An Overview of a Compiler

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- About the course
- Why should we study compiler design?
- Compiler overview with block diagrams

About the Course

- A detailed look at the internals of a compiler
- Does not assume any background but is intensive
- Doing programming assignments and solving theoretical problems are both essential
- A compiler is an excellent example of theory translated into practice in a remarkable way

Why Should We Study Compiler Design?

- Compilers are everywhere!
- Many applications for compiler technology
  - Parsers for HTML in web browser
  - Interpreters for javascript/flash
  - Machine code generation for high level languages
  - Software testing
  - Program optimization
  - Malicious code detection
  - Design of new computer architectures
    - Compiler-in-the-loop hardware development
  - Hardware synthesis: VHDL to RTL translation
  - Compiled simulation
    - Used to simulate designs written in VHDL
    - No interpretation of design, hence faster

About the Complexity of Compiler Technology

- A compiler is possibly the most complex system software and writing it is a substantial exercise in software engineering
- The complexity arises from the fact that it is required to map a programmer's requirements (in a HLL program) to architectural details
- It uses algorithms and techniques from a very large number of areas in computer science
- Translates intricate theory into practice - enables tool building

About the Nature of Compiler Algorithms

- Draws results from mathematical logic, lattice theory, linear algebra, probability, etc.
  - type checking, static analysis, dependence analysis and loop parallelization, cache analysis, etc.
- Makes practical application of
  - Greedy algorithms - register allocation
  - Heuristic search - list scheduling
  - Graph algorithms - dead code elimination, register allocation
  - Dynamic programming - instruction selection
  - Optimization techniques - instruction scheduling
  - Finite automata - lexical analysis
  - Pushdown automata - parsing
  - Fixed point algorithms - data-flow analysis
  - Complex data structures - symbol tables, parse trees, data dependence graphs
  - Computer architecture - machine code generation

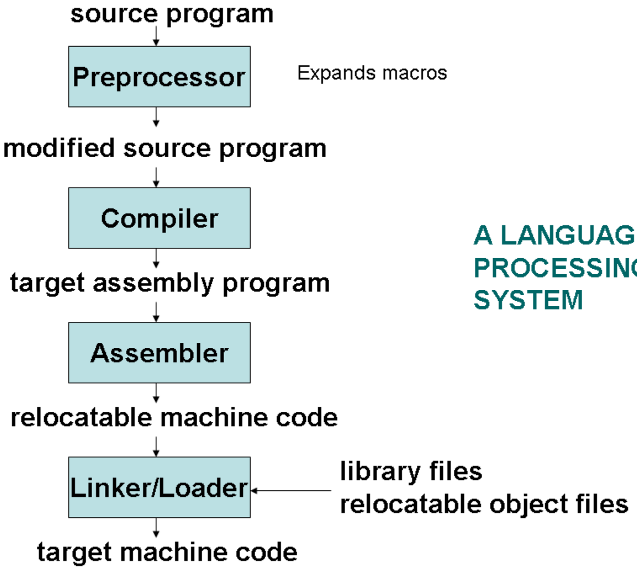
Other Uses of Scanning and Parsing Techniques

- Assembler implementation
- Online text searching (GREP, AWK) and word processing
- Website filtering
- Command language interpreters
- Scripting language interpretation (Unix shell, Perl, Python)
- XML parsing and document tree construction
- Database query interpreters

Other Uses of Program Analysis Techniques

- Converting a sequential loop to a parallel loop
- Program analysis to determine if programs are data-race free
- Profiling programs to determine busy regions
- Program slicing
- Data-flow analysis approach to software testing
  - Uncovering errors along all paths
  - Dereferencing null pointers
  - Buffer overflows and memory leaks
- Worst Case Execution Time (WCET) estimation and energy analysis

Language Processing System



source program

Preprocessor

Expands macros

modified source program

Compiler

target assembly program

Assembler

relocatable machine code

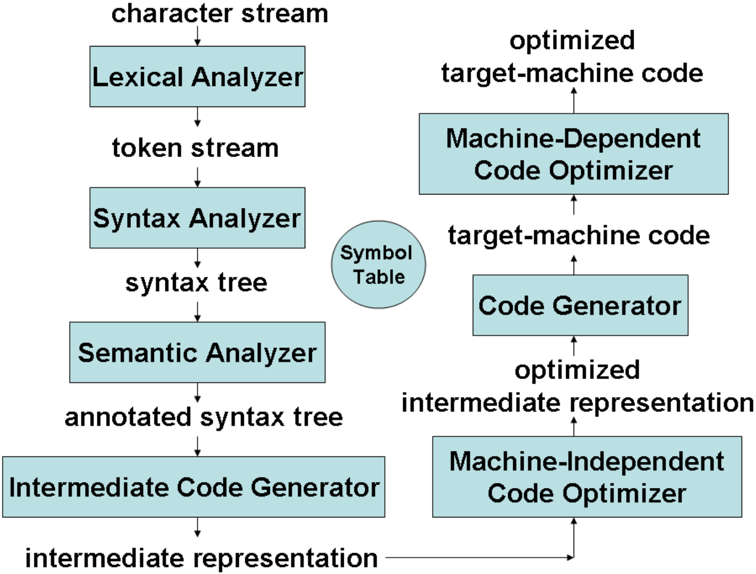
Linker/Loader

library files
relocatable object files

target machine code

A LANGUAGE PROCESSING SYSTEM

Compiler Overview



character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

annotated syntax tree

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

optimized intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

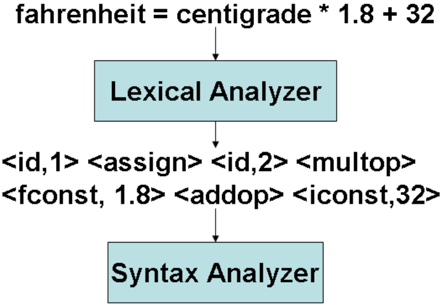
optimized target-machine code

Symbol Table

Compilers and Interpreters

- Compilers generate machine code, whereas interpreters interpret intermediate code
- Interpreters are easier to write and can provide better error messages (symbol table is still available)
- Interpreters are at least 5 times slower than machine code generated by compilers
- Interpreters also require much more memory than machine code generated by compilers
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP

Translation Overview - Lexical Analysis



fahrenheit = centigrade * 1.8 + 32

Lexical Analyzer

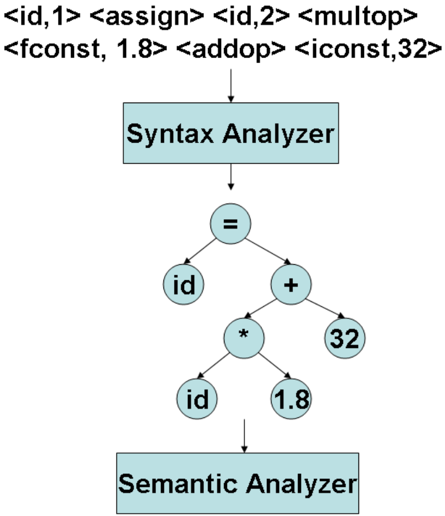
<id,1> <assign> <id,2> <multop>
<fconst, 1.8> <addop> <iconst,32>

Syntax Analyzer

Lexical Analysis

- LA can be generated automatically from regular expression specifications
    - LEX and Flex are two such tools
- LA is a deterministic finite state automaton
- Why is LA separate from parsing?
    - Simplification of design - software engineering reason
    - I/O issues are limited LA alone
    - LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

Translation Overview - Syntax Analysis



<id,1> <assign> <id,2> <multop>
<fconst, 1.8> <addop> <iconst,32>

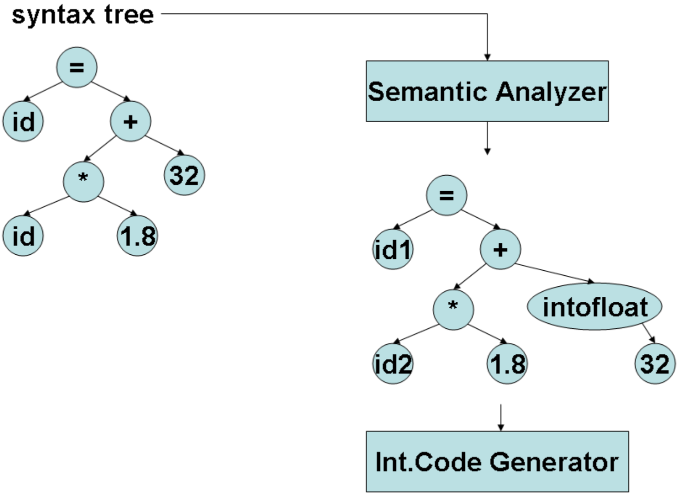
Syntax Analyzer

Semantic Analyzer

Parsing or Syntax Analysis

- Syntax analyzers (parsers) can be generated automatically from several variants of context-free grammar specifications
    - LL(1), and LALR(1) are the most popular ones
    - ANTLR (for LL(1)), YACC and Bison (for LALR(1)) are such tools
- Parsers are deterministic push-down automata
- Parsers cannot handle context-sensitive features of programming languages; e.g.,
    - Variables are declared before use
    - Types match on both sides of assignments
    - Parameter types and number match in declaration and use

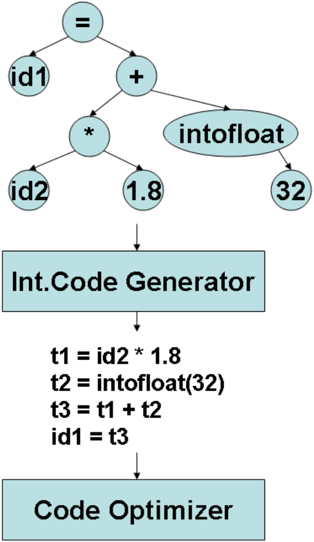
Translation Overview - Semantic Analysis



Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Type checking of various programming language constructs is one of the most important tasks
- Stores type information in the symbol table or the syntax tree
    - Types of variables, function parameters, array dimensions, etc.
    - Used not only for semantic validation but also for subsequent phases of compilation
- Static semantics of programming languages can be specified using attribute grammars

Translation Overview - Intermediate Code Generation



Int.Code Generator

```
t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3
```

Code Optimizer

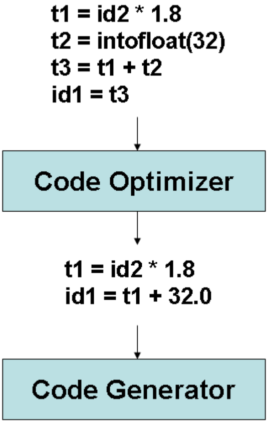
Intermediate Code Generation

- While generating machine code directly from source code is possible, it entails two problems
  - With m languages and n target machines, we need to write $m \times n$ compilers
  - The code optimizer which is one of the largest and very-difficult-to-write components of any compiler cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)

Different Types of Intermediate Code

- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
  - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

Translation Overview - Code Optimization



t1 = id2 * 1.8
t2 = intofloat(32)
t3 = t1 + t2
id1 = t3

Code Optimizer

t1 = id2 * 1.8
id1 = t1 + 32.0

Code Generator

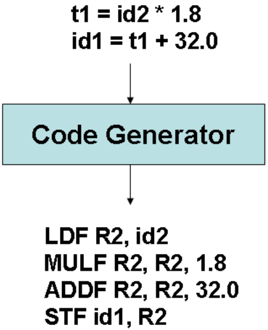
Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
    - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
    - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)

Examples of Machine-Independant Optimizations

- Common sub-expression elimination
- Copy propagation
- Loop invariant code motion
- Partial redundancy elimination
- Induction variable elimination and strength reduction
- Code opimization needs information about the program
    - which expressions are being recomputed in a function?
    - which definitions reach a point?
- All such information is gathered through data-flow analysis

Translation Overview - Code Generation



```
t1 = id2 * 1.8
id1 = t1 + 32.0
```

Code Generator

```
LDF R2, id2
MULF R2, R2, 1.8
ADDF R2, R2, 32.0
STF id1, R2
```

Code Generation

- Converts intermediate code to machine code
- Each intermediate code instruction may result in many machine instructions or vice-cersa
- Must handle all aspects of machine architecture
    - Registers, pipelining, cache, multiple function units, etc.
- Generating efficient code is an NP-complete problem
    - Tree pattern matching-based strategies are among the best
    - Needs tree intermediate code
- Storage allocation decisions are made here
    - Register allocation and assignment are the most important problems

Machine-Dependent Optimizations

- Peephole optimizations
  - Analyze sequence of instructions in a small window (*peephole*) and using preset patterns, replace them with a more efficient sequence
  - Redundant instruction elimination
    e.g., replace the sequence [LD A,R1][ST R1,A] by [LD A,R1]
  - Eliminate “jump to jump” instructions
  - Use machine idioms (use INC instead of LD and ADD)
- Instruction scheduling (reordering) to eliminate pipeline interlocks and to increase parallelism
- Trace scheduling to increase the size of basic blocks and increase parallelism
- Software pipelining to increase parallelism in loops

Lexical Analysis - Part 1

Y.N. Srikant

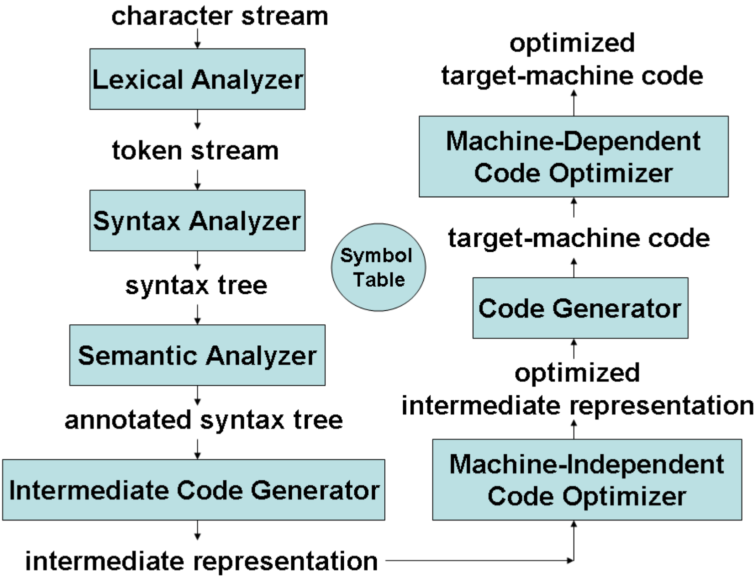
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is lexical analysis?
- Why should LA be separated from syntax analysis?
- Tokens, patterns, and lexemes
- Difficulties in lexical analysis
- Recognition of tokens - finite automata and transition diagrams
- Specification of tokens - regular expressions and regular definitions
- LEX - A Lexical Analyzer Generator

Compiler Overview



character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

annotated syntax tree

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

optimized intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

optimized target-machine code

Symbol Table

What is Lexical Analysis?

- The input is a high level language program, such as a 'C' program in the form of a sequence of characters
- The output is a sequence of *tokens* that is sent to the parser for syntax analysis
- Strips off blanks, tabs, newlines, and comments from the source program
- Keeps track of line numbers and associates error messages from various parts of a compiler with line numbers
- Performs some preprocessor functions such as `#define` and `#include` in 'C'

Separation of Lexical Analysis from Syntax Analysis

- Simplification of design - software engineering reason
- I/O issues are limited LA alone
- More compact and faster parser
  - Comments, blanks, etc., need not be handled by the parser
  - A parser is more complicated than a lexical analyzer and shrinking the grammar makes the parser faster
    - No rules for numbers, names, comments, etc., are needed in the parser
- LA based on finite automata are more efficient to implement than pushdown automata used for parsing (due to stack)

Tokens, Patterns, and Lexemes

- Running example: *float abs_zero_Kelvin = -273;*
- Token (also called *word*)
  - A string of characters which logically belong together
  - **float**, **identifier**, **equal**, **minus**, **intnum**, **semicolon**
  - Tokens are treated as terminal symbols of the grammar specifying the source language
- Pattern
  - The set of strings for which the *same* token is produced
  - The pattern is said to *match* each string in the set
  - *float*, *l(l+d+_)**, *=*, *-*, *d+*, *;*
- Lexeme
  - The sequence of characters matched by a pattern to form the corresponding token
  - “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;”

Tokens in Programming Languages

- Keywords, operators, identifiers (names), constants, literal strings, punctuation symbols such as parentheses, brackets, commas, semicolons, and colons, etc.
- A unique integer representing the token is passed by LA to the parser
- Attributes for tokens (apart from the integer representing the token)
    - *identifier*: the lexeme of the token, or a pointer into the symbol table where the lexeme is stored by the LA
    - *intnum*: the value of the integer (similarly for *floatnum*, etc.)
    - *string*: the string itself
    - The exact set of attributes are dependent on the compiler designer

Difficulties in Lexical Analysis

- Certain languages do not have any reserved words, *e.g.,* **while, do, if, else**, etc., are reserved in 'C', but not in PL/1
- In FORTRAN, some keywords are context-dependent
  - In the statement, *DO 10 I = 10.86*, **DO10I** is an identifier, and **DO** is not a keyword
  - But in the statement, *DO 10 I = 10, 86*, **DO** is a keyword
  - Such features require substantial *look ahead* for resolution
- Blanks are not significant in FORTRAN and can appear in the midst of identifiers, but not so in 'C'
- LA cannot catch any significant errors except for simple errors such as, illegal symbols, etc.
- In such cases, LA skips characters in the input until a well-formed token is found

Specification and Recognition of Tokens

- Regular definitions, a mechansm based on *regular expressions* are very popular for specification of tokens
  - Has been implemented in the lexical analyzer generator tool, LEX
  - We study regular expressions first, and then, token specification using LEX
- Transition diagrams, a variant of finite state automata, are used to implement regular definitions and to recognize tokens
  - Transition diagrams are usually used to model LA before translating them to programs by hand
  - LEX automatically generates optimized FSA from regular definitions
  - We study FSA and their generation from regular expressions in order to understand transition diagrams and LEX

Languages

- **Symbol**: An abstract entity, not defined
  - Examples: letters and digits
- **String**: A finite sequence of juxtaposed symbols
  - **abcb, caba** are strings over the symbols *a,b,* and *c*
  - $|w|$ is the length of the string *w*, and is the #symbols in it
  - ϵ is the empty string and is of length 0
- **Alphabet**: A *finite* set of symbols
- **Language**: A set of strings of symbols from some alphabet
  - Φ and $\{\epsilon\}$ are languages
  - The set of palindromes over {0,1} is an infinite language
  - The set of strings, {01, 10, 111} over {0,1} is a finite language
- If Σ is an alphabet, Σ^* is the set of all strings over Σ

Language Representations

- Each subset of Σ^* is a language
- This set of languages over Σ^* is uncountably infinite
- Each language must have by a finite representation
  - A finite representation can be encoded by a finite string
  - Thus, each string of Σ^* can be thought of as representing some language over the alphabet Σ
  - Σ^* is countably infinite
  - Hence, there are more languages than language representations
- **Regular expressions** (type-3 or regular languages), **context-free grammars** (type-2 or context-free languages), **context-sensitive grammars** (type-1 or context-sensitive languages), and **type-0 grammars** are finite representations of respective languages
- RL $<<$ CFL $<<$ CSL $<<$ type-0 languages

Examples of Languages

Let $\Sigma = \{a, b, c\}$

- $L_1 = \{a^m b^n | m, n \geq 0\}$ is regular
- $L_2 = \{a^n b^n | n \geq 0\}$ is context-free but not regular
- $L_3 = \{a^n b^n c^n | n \geq 0\}$ is context-sensitive but neither regular nor context-free
- Showing a language that is type-0, but none of CSL, CFL, or RL is very intricate and is omitted

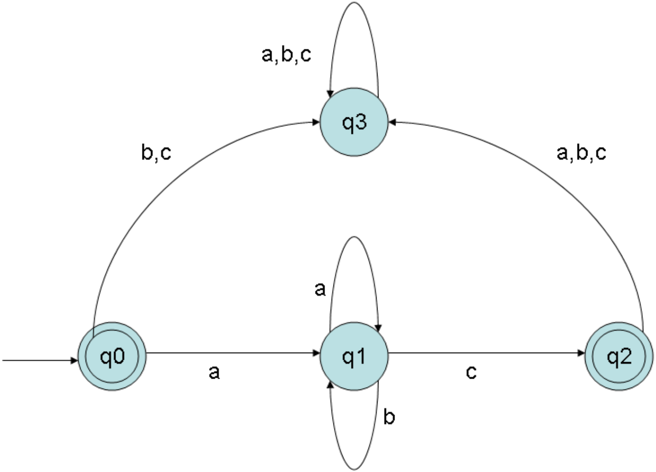
Automata

- Automata are machines that accept languages
  - Finite State Automata accept RLs (corresponding to REs)
  - Pushdown Automata accept CFLs (corresponding to CFGs)
  - Linear Bounded Automata accept CSLs (corresponding to CSGs)
  - Turing Machines accept type-0 languages (corresponding to type-0 grammars)
- Applications of Automata
  - Switching circuit design
  - Lexical analyzer in a compiler
  - String processing (*grep, awk*), etc.
  - State charts used in object-oriented design
  - Modelling control applications, e.g., elevator operation
  - Parsers of all types
  - Compilers

Finite State Automaton

- An FSA is an **acceptor** or **recognizer** of regular languages
- An FSA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where
  - Q is a finite set of states
  - Σ is the input alphabet
  - δ is the transition function, $\delta : \; Q \times \Sigma \rightarrow Q$
    That is, $\delta(q, a)$ is a state for each state q and input symbol a
  - q_0 is the start state
  - F is the set of *final* or *accepting* states
- In one move from some state q, an FSA reads an input symbol, changes the state based on δ, and gets ready to read the next input symbol
- An FSA **accepts** its input string, if starting from q_0, it consumes the entire input string, and reaches a final state
- If the last state reached is not a final state, then the input string is rejected

FSA Example - 1



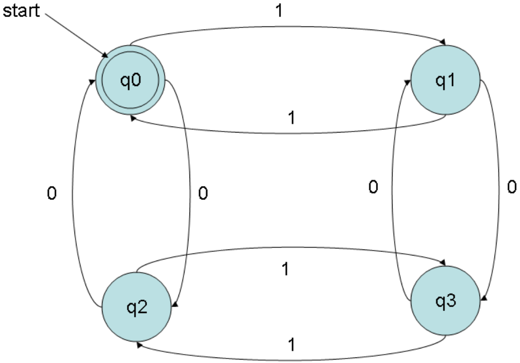
FSA Example -1 (Contd.)

- $Q = \{q_0, q_1, q_2, q_3\}$
- $\Sigma = \{a, b, c\}$
- q_0 is the start state and $F = \{q_0, q_2\}$
- The transition function δ is defined by the table below

state	symbol		
	a	b	c
q_0	q_1	q_3	q_3
q_1	q_1	q_1	q_2
q_2	q_3	q_3	q_3
q_3	q_3	q_3	q_3

The accepted language is the set of all strings beginning with an 'a' and ending with a 'c' (ϵ is also accepted)

FSA Example - 2



- $Q = \{q_0, q_1, q_2, q_3\}$, q_0 is the start state
- $F = \{q_0\}, \delta$ is as in the figure
- Language accepted is the set of all strings of 0's and 1's, in which the no. of 0's and the no. of 1's are even numbers

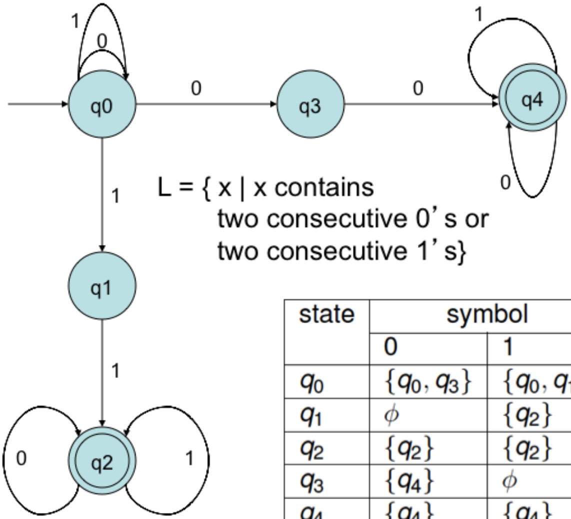
Regular Languages

- The language **accepted** by an FSA is the set of all strings accepted by it, i.e., $\delta(q_0, x) \epsilon F$
- This is a **regular language** or a **regular set**
- Later we will define **regular expressions** and **regular grammars** which are **generators** of regular languages
- It can be shown that for every regular expression, an FSA can be constructed and vice-versa

Nondeterministic FSA

- NFAs are FSA which allow 0, 1, or more transitions from a state on a given input symbol
- An NFA is a 5-tuple as before, but the transition function δ is different
- $\delta(q, a)$ = the set of all states p, such that there is a transition labelled a from q to p
- $\delta : \; Q \times \Sigma \rightarrow 2^Q$
- A string is accepted by an NFA if there *exists* a sequence of transitions corresponding to the string, that leads from the start state to some final state
- Every NFA can be converted to an equivalent deterministic FA (DFA), that accepts the same language as the NFA

Nondeterministic FSA Example - 1



L = { x | x contains
two consecutive 0's or
two consecutive 1's}

state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$

Lexical Analysis - Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

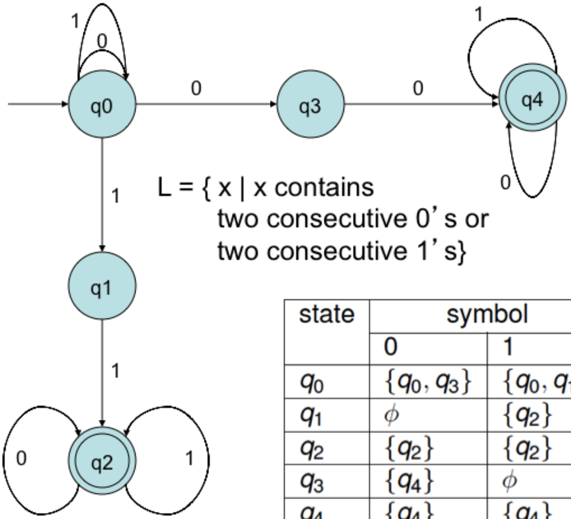
Outline of the Lecture

- What is lexical analysis? (covered in part 1)
- Why should LA be separated from syntax analysis? (covered in part 1)
- Tokens, patterns, and lexemes (covered in part 1)
- Difficulties in lexical analysis (covered in part 1)
- Recognition of tokens - finite automata and transition diagrams
- Specification of tokens - regular expressions and regular definitions
- LEX - A Lexical Analyzer Generator

Nondeterministic FSA

- NFAs are FSA which allow 0, 1, or more transitions from a state on a given input symbol
- An NFA is a 5-tuple as before, but the transition function δ is different
- $\delta(q, a)$ = the set of all states p, such that there is a transition labelled a from q to p
- $\delta : \; Q \times \Sigma \rightarrow 2^Q$
- A string is accepted by an NFA if there *exists* a sequence of transitions corresponding to the string, that leads from the start state to some final state
- Every NFA can be converted to an equivalent deterministic FA (DFA), that accepts the same language as the NFA

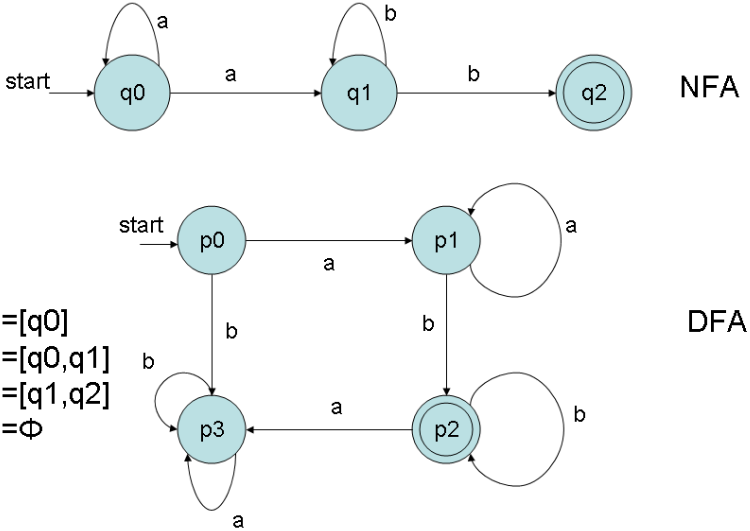
Nondeterministic FSA Example - 1



L = { x | x contains
two consecutive 0's or
two consecutive 1's}

state	symbol	
	0	1
q_0	$\{q_0, q_3\}$	$\{q_0, q_1\}$
q_1	ϕ	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$
q_3	$\{q_4\}$	ϕ
q_4	$\{q_4\}$	$\{q_4\}$

An NFA and an Equivalent DFA



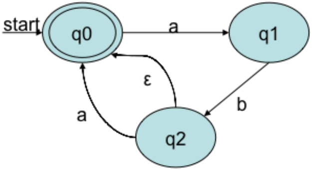
p0=[q0]
p1=[q0,q1]
p2=[q1,q2]
p3=Φ

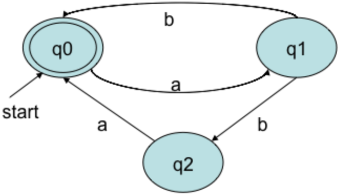
Example of NFA to DFA conversion

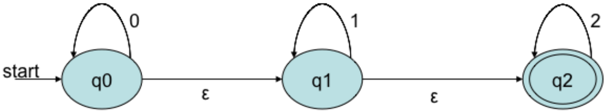
- The start state of the DFA would correspond to the set $\{q_0\}$ and will be represented by $[q_0]$
- Starting from $\delta([q_0], a)$, the new states of the DFA are constructed on *demand*
- Each subset of NFA states is a *possible* DFA state
- All the states of the DFA containing some final state as a member would be final states of the DFA
- For the NFA presented before (whose equivalent DFA was also presented)
  - $\delta[q_0], a) = [q_0, q_1], \ \ \delta([q_0], b) = \phi$
  - $\delta([q_0, q_1], a) = [q_0, q_1], \ \ \delta([q_0, q_1], b) = [q_1, q_2]$
  - $\delta(\phi, a) = \phi, \ \ \delta(\phi, b) = \phi$
  - $\delta([q_1, q_2], a) = \phi, \ \ \delta([q_1, q_2], b) = [q_1, q_2]$
  - $[q_1, q_2]$ is the final state
- In the worst case, the converted DFA may have 2^n states, where n is the no. of states of the NFA

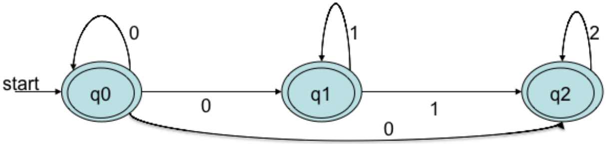
NFA with ϵ-Moves

ϵ-NFA is equivalent to NFA in power









Regular Expressions

Let Σ be an alphabet. The REs over Σ and the languages they denote (or generate) are defined as below

1. ϕ is an RE. $L(\phi) = \phi$
2. ϵ is an RE. $L(\epsilon) = \{\epsilon\}$
3. For each $a \in \Sigma$, a is an RE. $L(a) = \{a\}$
4. If r and s are REs denoting the languages R and S, respectively
    - (rs) is an RE, $L(rs) = R.S = \{xy \mid x \in R \wedge y \in S\}$
    - $(r + s)$ is an RE, $L(r + s) = R \cup S$
    - (r^*) is an RE, $L(r^*) = R^* = \bigcup_{i=0}^{\infty} R^i$

      (L^* is called the *Kleene closure* or *closure* of L)

Examples of Regular Expressions

1. L = set of all strings of 0's and 1's
   $r = (0+1)^*$
   - How to generate the string 101 ?
   - $(0+1)^* \Rightarrow^4 (0+1)(0+1)(0+1)\epsilon \Rightarrow^4 101$
2. L = set of all strings of 0's and 1's, with at least two consecutive 0's
   $r = (0+1)^*00(0+1)^*$
3. $L = \{w \in \{0,1\}^* \mid w$ has two or three occurrences of 1, the first and second of which are not consecutive$\}$
   $r = 0^*10^*010^*(10^* + \epsilon)$
4. $r = (1+10)^*$
   L = set of all strings of 0's and 1's, beginning with 1 and not having two consecutive 0's
5. $r = (0+1)^*011$
   L = set of all strings of 0's and 1's ending in 011

Examples of Regular Expressions

6. $r = c^*(a + bc^*)^*$
   L = set of all strings over {a,b,c} that do not have the substring *ac*
7. $L = \{w \mid w \in \{a, b\}^* \wedge w$ ends with $a\}$
   $r = (a + b)^* a$
8. L = {if, then, else, while, do, begin, end}
   $r = if + then + else + while + do + begin + end$

Examples of Regular Definitions

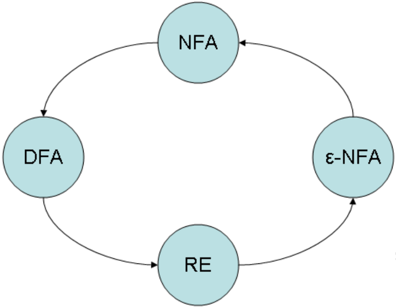
A *regular definition* is a sequence of "equations" of the form $d_1 = r_1;\ d_2 = r_2;\ \dots;\ d_n = r_n$, where each d_i is a distinct name, and each r_i is a regular expression over the symbols $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

1. identifiers and integers
   $letter = a + b + c + d + e;\ \ digit = 0 + 1 + 2 + 3 + 4;$
   $identifier = letter(letter + digit)^*;\ \ number = digit\ digit^*$
2. unsigned numbers
   $digit = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9;$
   $digits = digit\ digit^*;$
   $optional\_fraction = .digits + \epsilon;$
   $optional\_exponent = (E(+|-|\epsilon)digits) + \epsilon$
   $unsigned\_number =$
   $digits\ optional\_fraction\ optional\_exponent$

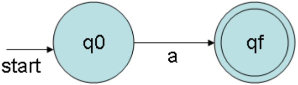
Equivalence of REs and FSA

- Let r be an RE. Then there exists an NFA with ϵ-transitions that accepts $L(r)$. The proof is by construction.
- If L is accepted by a DFA, then L is generated by an RE. The proof is tedious.

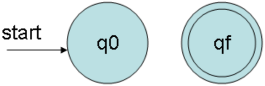
Construction of FSA from RE - r = ϕ, ϵ, or *a*



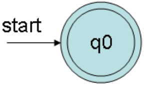
r=a



r=Φ

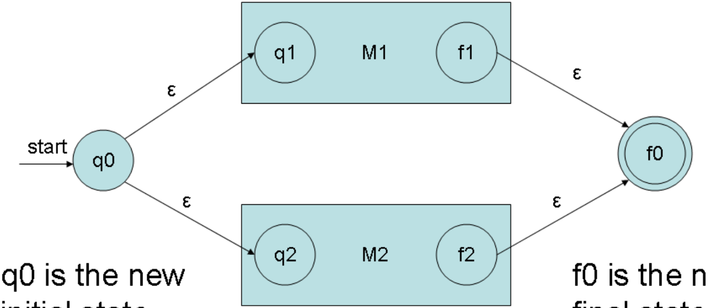


r=ε



FSA for r = r1 + r2

FSA for the RE r = r1 + r2



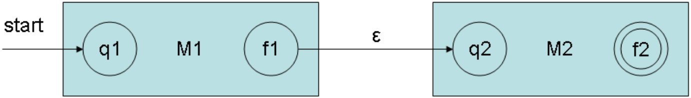
q0 is the new initial state

f0 is the new final state

q1,q2 are no more initial states
f1,f2 are no more final states

FSA for r = r1 r2

FSA for RE r = r1 r2



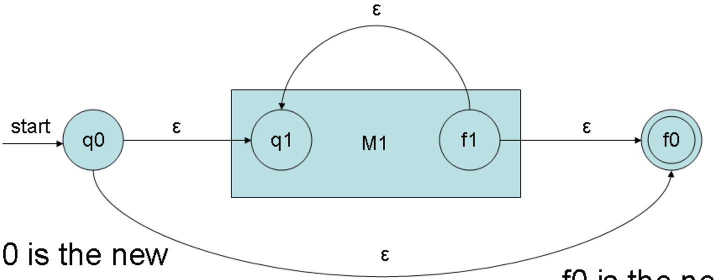
q1 is the new
start state

f2 is the new
final state

f1 is no more a final state
q2 is no more a start state

FSA for r = r1*

FSA for r = r1*

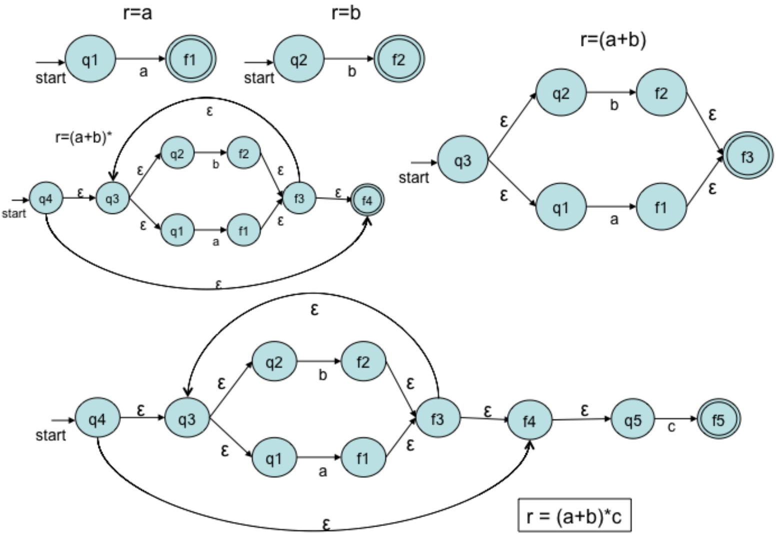


q0 is the new
start state

f0 is the new
final state

q1 is no more a start state
f1 is no more a final state

NFA Construction for r = (a+b)*c

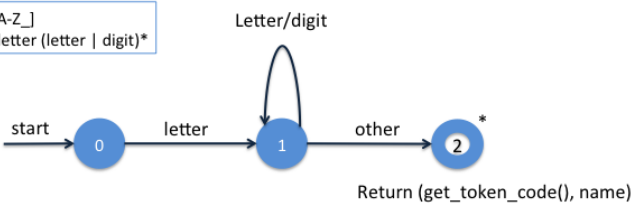


Transition Diagrams

- Transition diagrams are generalized DFAs with the following differences
  - Edges may be labelled by a symbol, a set of symbols, or a regular definition
  - Some accepting states may be indicated as *retracting states*, indicating that the lexeme does not include the symbol that brought us to the accepting state
  - Each accepting state has an action attached to it, which is executed when that state is reached. Typically, such an action returns a token and its attribute value
- Transition diagrams are not meant for machine translation but only for manual translation

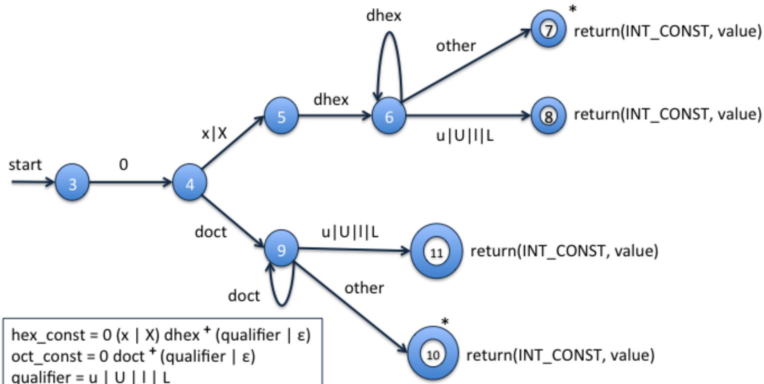
Transition Diagram for Identifiers and Reserved Words

letter = [a-zA-Z_]
Identifier = letter (letter | digit)*



- '*' indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)

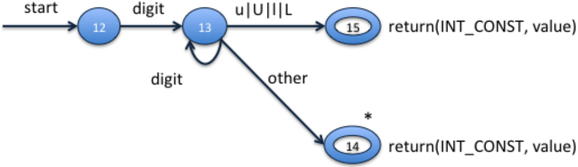
Transition Diagrams for Hex and Oct Constants



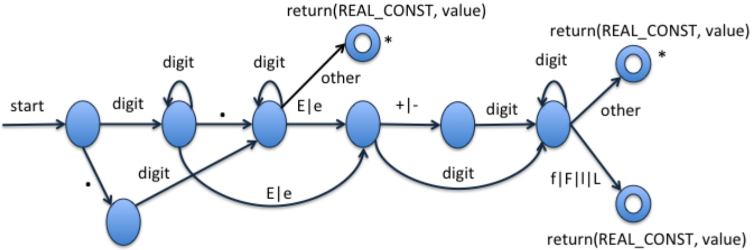
hex_const = 0 (x | X) dhex$^+$ (qualifier | ε)
oct_const = 0 doct$^+$ (qualifier | ε)
qualifier = u | U | l | L
dhex = [0-9A-F]
doct = [0-7]

Transition Diagrams for Integer Constants

int_const = digit $^+$ (qualifier | ε)
qualifier = u | U | l | L
digit = [0-9]

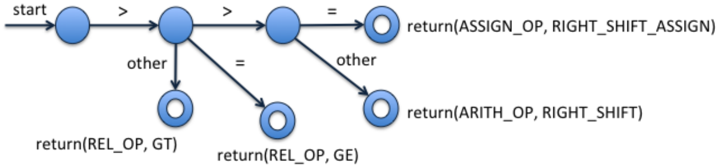


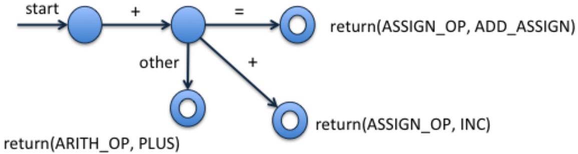
Transition Diagrams for Real Constants



real_const = (digit$^+$ exponent (qualifier | ε)) |
(digit* "." digit$^+$ (exponent | ε) (qualifier | ε)) |
(digit$^+$ "." digit* (exponent | ε) (qualifier | ε))
exponent = (E|e)(+|-|ε) digit$^+$
qualifier = f | F | l | L
digit = [0-9]

Transition Diagrams for a few Operators



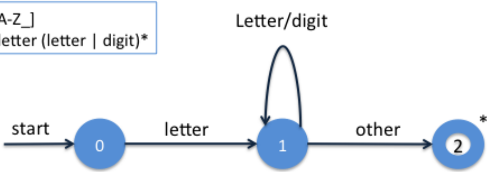


Lexical Analyzer Implementation from Trans. Diagrams

```
TOKEN gettoken() {
   TOKEN mytoken; char c;
   while(1) { switch (state) {
     /* recognize reserved words and identifiers */
       case 0: c = nextchar(); if (letter(c))
               state = 1; else state = failure();
               break;
       case 1: c = nextchar();
               if (letter(c) || digit(c))
               state = 1; else state = 2; break;
       case 2: retract(1);
               mytoken.token = search_token();
               if (mytoken.token == IDENTIFIER)
               mytoken.value = get_id_string();
               return(mytoken);
```

Transition Diagram for Identifiers and Reserved Words

letter = [a-zA-Z_]
Identifier = letter (letter | digit)*



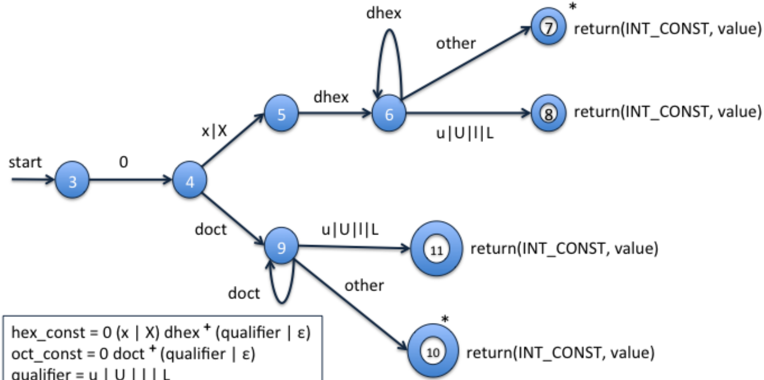
Return (get_token_code(), name)

- '*' indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)

Lexical Analyzer Implementation from Trans. Diagrams

```
/* recognize hexa and octal constants */
  case 3: c = nextchar();
          if (c == '0') state = 4; break;
          else state = failure();
  case 4: c = nextchar();
          if ((c == 'x') || (c == 'X'))
          state = 5; else if (digitoct(c))
          state = 9; else state = failure();
          break;
  case 5: c = nextchar(); if (digithex(c))
          state = 6; else state = failure();
          break;
```

Transition Diagrams for Hex and Oct Constants



hex_const = 0 (x | X) dhex $^+$ (qualifier | ε)
oct_const = 0 doct $^+$ (qualifier | ε)
qualifier = u | U | l | L
dhex = [0-9A-F]
doct = [0-7]

Lexical Analyzer Implementation from Trans. Diagrams

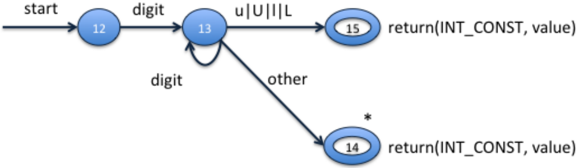
```
    case 6: c = nextchar(); if (digithex(c))
            state = 6; else if ((c == 'u')||
            (c == 'U')||(c == 'l')||
            (c == 'L')) state = 8;
            else state = 7; break;
    case 7: retract(1);
/* fall through to case 8, to save coding */
    case 8: mytoken.token = INT_CONST;
            mytoken.value = eval_hex_num();
            return(mytoken);
    case 9: c = nextchar(); if (digitoct(c))
            state = 9; else if ((c == 'u')||
            (c == 'U')||(c == 'l')||(c == 'L'))
            state = 11; else state = 10; break;
```

Lexical Analyzer Implementation from Trans. Diagrams

```
    case 10: retract(1);
/* fall through to case 11, to save coding */
    case 11: mytoken.token = INT_CONST;
             mytoken.value = eval_oct_num();
             return(mytoken);
```

Transition Diagrams for Integer Constants

int_const = digit $^+$ (qualifier | ε)
qualifier = u | U | l | L
digit = [0-9]



Lexical Analyzer Implementation from Trans. Diagrams

```
       /* recognize integer constants */
           case 12: c = nextchar(); if (digit(c))
                    state = 13; else state = failure();
           case 13: c = nextchar(); if (digit(c))
                    state = 13;else if ((c == ’u’)||
                    (c == ’U’)||(c == ’l’)||(c == ’L’))
                    state = 15; else state = 14; break;
           case 14: retract(1);
       /* fall through to case 15, to save coding */
           case 15: mytoken.token = INT_CONST;
                    mytoken.value = eval_int_num();
                    return(mytoken);
           default: recover();
           }
       }
    }
```

Lexical Analysis - Part 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is lexical analysis? (covered in part 1)
- Why should LA be separated from syntax analysis? (covered in part 1)
- Tokens, patterns, and lexemes (covered in part 1)
- Difficulties in lexical analysis (covered in part 1)
- Recognition of tokens - finite automata and transition diagrams (covered in part 2)
- Specification of tokens - regular expressions and regular definitions (covered in part 2)
- LEX - A Lexical Analyzer Generator

Transition Diagrams

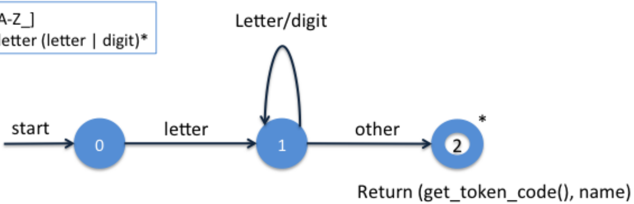
- Transition diagrams are generalized DFAs with the following differences
  - Edges may be labelled by a symbol, a set of symbols, or a regular definition
  - Some accepting states may be indicated as *retracting states*, indicating that the lexeme does not include the symbol that brought us to the accepting state
  - Each accepting state has an action attached to it, which is executed when that state is reached. Typically, such an action returns a token and its attribute value
- Transition diagrams are not meant for machine translation but only for manual translation

Lexical Analyzer Implementation from Trans. Diagrams

```
TOKEN gettoken() {
   TOKEN mytoken; char c;
   while(1) { switch (state) {
     /* recognize reserved words and identifiers */
       case 0: c = nextchar(); if (letter(c))
               state = 1; else state = failure();
               break;
       case 1: c = nextchar();
               if (letter(c) || digit(c))
               state = 1; else state = 2; break;
       case 2: retract(1);
               mytoken.token = search_token();
               if (mytoken.token == IDENTIFIER)
               mytoken.value = get_id_string();
               return(mytoken);
```

Transition Diagram for Identifiers and Reserved Words

letter = [a-zA-Z_]
Identifier = letter (letter | digit)*



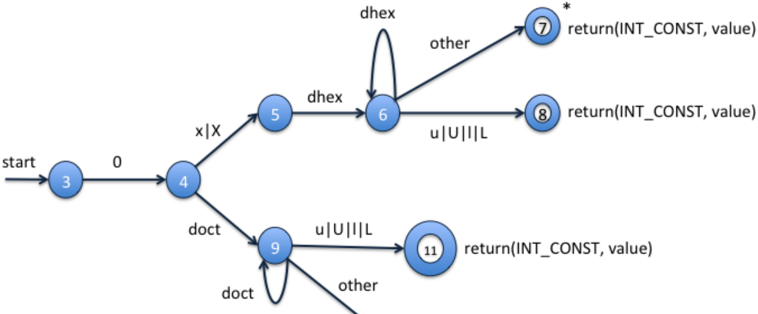
Return (get_token_code(), name)

- '*' indicates retraction state
- get_token_code() searches a table to check if the name is a reserved word and returns its integer code, if so
- Otherwise, it returns the integer code of IDENTIFIER token, with name containing the string of characters forming the token (name is not relevant for reserved words)

Lexical Analyzer Implementation from Trans. Diagrams

```
/* recognize hexa and octal constants */
  case 3: c = nextchar();
          if (c == ’0’) state = 4; break;
          else state = failure();
  case 4: c = nextchar();
          if ((c == ’x’) || (c == ’X’))
          state = 5; else if (digitoct(c))
          state = 9; else state = failure();
          break;
  case 5: c = nextchar(); if (digithex(c))
          state = 6; else state = failure();
          break;
```

Transition Diagrams for Hex and Oct Constants



hex_const = 0 (x | X) dhex $^+$ (qualifier | ε)
oct_const = 0 doct $^+$ (qualifier | ε)
qualifier = u | U | l | L
dhex = [0-9A-F]
doct = [0-7]

Lexical Analyzer Implementation from Trans. Diagrams

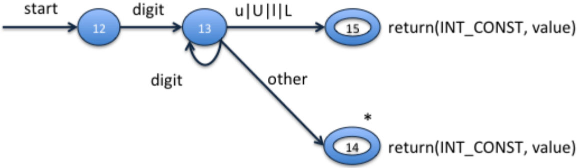
```
    case 6: c = nextchar(); if (digithex(c))
            state = 6; else if ((c == ’u’)||
            (c == ’U’)||(c == ’l’)||
            (c == ’L’)) state = 8;
            else state = 7; break;
    case 7: retract(1);
/* fall through to case 8, to save coding */
    case 8: mytoken.token = INT_CONST;
            mytoken.value = eval_hex_num();
            return(mytoken);
    case 9: c = nextchar(); if (digitoct(c))
            state = 9; else if ((c == ’u’)||
            (c == ’U’)||(c == ’l’)||(c == ’L’))
            state = 11; else state = 10; break;
```

Lexical Analyzer Implementation from Trans. Diagrams

```
    case 10: retract(1);
/* fall through to case 11, to save coding */
    case 11: mytoken.token = INT_CONST;
             mytoken.value = eval_oct_num();
             return(mytoken);
```

Transition Diagrams for Integer Constants

int_const = digit $^+$ (qualifier | ε)
qualifier = u | U | l | L
digit = [0-9]



Lexical Analyzer Implementation from Trans. Diagrams

```
    /* recognize integer constants */
        case 12: c = nextchar(); if (digit(c))
                 state = 13; else state = failure();
        case 13: c = nextchar(); if (digit(c))
                 state = 13;else if ((c == ’u’)||
                 (c == ’U’)||(c == ’l’)||(c == ’L’))
                 state = 15; else state = 14; break;
        case 14: retract(1);
    /* fall through to case 15, to save coding */
        case 15: mytoken.token = INT_CONST;
                 mytoken.value = eval_int_num();
                 return(mytoken);
        default: recover();
        }
    }
}
```

Combining Transition Diagrams to form LA

- Different transition diagrams must be combined appropriately to yield an LA
  - Combining TDs is not trivial
  - It is possible to try different transition diagrams one after another
  - For example, TDs for reserved words, constants, identifiers, and operators could be tried in that order
  - However, this does not use the “longest match" characteristic (*thenext* would be an identifier, and not reserved word *then* followed by identifier *ext*)
  - To find the longest match, all TDs must be tried and the longest match must be used
- Using LEX to generate a lexical analyzer makes it easy for the compiler writer

LEX - A Lexical Analyzer Generator

- LEX has a language for describing regular expressions
- It generates a pattern matcher for the regular expression specifications provided to it as input
- General structure of a LEX program
  {definitions} – Optional
  %%
  {rules} – Essential
  %%
  {user subroutines} – Essential
- Commands to create an LA
  - lex ex.l – creates a C-program *lex.yy.c*
  - gcc -o ex.o lex.yy.c – produces ex.o
  - ex.o is a *lexical analyzer*, that carves tokens from its input

LEX Example

```
/* LEX specification for the Example */
%%
[A-Z]+    {ECHO; printf("\n");}
.|\n       ;
%%
yywrap(){}
main(){yylex();}
```

```
/* Input */
wewevWEUFWIGhHkkH
sdcwehSDWEhTkFLksewT
```

```
/* Output */
WEUFWIG
H
H
SDWE
T
FL
T
```

Definitions Section

- Definitions Section contains definitions and included code
  - Definitions are like macros and have the following form: **name translation**

```
digit [0-9]
number {digit} {digit}*
```

  - Included code is all code included between **%{** and **%}**

```
%{
    float number; int count=0;
%}
```

Rules Section

- Contains patterns and C-code
- A line starting with white space or material enclosed in %{ and %} is C-code
- A line starting with anything else is a pattern line
- Pattern lines contain a pattern followed by some white space and C-code
  $\{pattern\} \quad \{action\ (C - code)\}$
- C-code lines are copied verbatim to the the generated C-file
- Patterns are translated into NFA which are then converted into DFA, optimized, and stored in the form of a table and a driver routine
- The action associated with a pattern is executed when the DFA recognizes a string corresponding to that pattern and reaches a final state

Strings and Operators

- **Examples of strings**: integer a57d hello
- **Operators**:

  `" \ [] ^ - ? . * + | () $ {} % <>`

  \ can be used as an escape character as in C
- **Character classes**: enclosed in [and]
  Only \, -, and ^ are special inside [ ]. All other operators are irrelevant inside [ ]
  Examples:

```
[-+][0-9]+    ---> (-|+)(0|1|2|3|4|5|6|7|8|9)+
[a-d][0-4][A-C]    ---> a|b|c|d|0|1|2|3|4|A|B|C
[^abc]  ---> all char except a,b, or c,
             including special and control char
[+\-][0-5]+  ---> (+|-)(0|1|2|3|4|5)+
[^a-zA-Z]  ---> all char which are not letters
```

Operators - Details

- **. operator**: matches any character except newline
- **? operator**: used to implement ϵ option
  $ab?c$ stands for $a(b \mid \epsilon)c$
- **Repetition, alternation, and grouping**:
  $(ab \mid cd+)?(ef)*$ —> $(ab \mid c(d)^{+} \mid \epsilon)(ef)^{*}$
- **Context sensitivity**: `/`, `^`, `$`, are context-sensitive operators
  - `^`: If the first char of an expression is `^`, then that expression is matched only at the beginning of a line. Holds only outside [ ] operator
  - $: If the last char of an expression is $, then that expression is matched only at the end of a line
  - /: Look ahead operator, indicates trailing context

```
^ab  ---> line beginning with ab
ab$  ---> line ending with ab (same as ab/\n)
DO/({letter}|{digit})* = ({letter}|{digit})*,
```

LEX Actions

- Default action is to copy input to output, those characters which are unmatched
- We need to provide patterns to **catch** characters
- **yytext**: contains the text matched against a pattern copying **yytext** can be done by the action **ECHO**
- **yyleng**: provides the number of characters matched
- LEX always tries the rules in the order written down and the *longest match* is preferred

  ```
  integer    action1;
  [a-z]+     action2;
  ```

  The input *integers* will match the second pattern

LEX Example 1: EX-1.lex

```
%%
[A-Z]+     {ECHO; printf("\n";}
.|\n       ;
%%
yywrap(){}
main(){yylex();}

/* Input */                    /* Output */
wewevWEUFWIGhHkkH              WEUFWIG
sdcwehSDWEhTkFLksewT           H
                               H
                               SDWE
                               T
                               FL
                               T
```

LEX Example 2: EX-2.lex

```
%%
^[ ]*\n
\n    {ECHO; yylineno++;}
.*    {printf("%d\t%s",yylineno,yytext);}
%%

yywrap(){}
main(){ yylineno = 1; yylex(); }
```

LEX Example 2 (contd.)

```
/* Input and Output */
========================
kurtrtotr
dvure

     123456789

euhoyo854
shacg345845nkfg
========================
1 kurtrtotr
2 dvure
3      123456789
4 euhoyo854
5 shacg345845nkfg
```

LEX Example 3: EX-3.lex

```
%{
FILE *declfile;
%}

blanks [ \t]*
letter  [a-z]
digit [0-9]
id    ({letter}|_)({letter}|{digit}|_)*
number {digit}+
arraydeclpart {id}"["{number}"]"
declpart ({arraydeclpart}|{id})
decllist ({declpart}{blanks}","{blanks})*
                 {blanks}{declpart}{blanks}
declaration (("int")|("float")){blanks}
                 {decllist}{blanks};
```

LEX Example 3 (contd.)

```
%%
{declaration} fprintf(declfile,"%s\n",yytext);
%%

yywrap(){
fclose(declfile);
}
main(){
declfile = fopen("declfile","w");
yylex();
}
```

LEX Example 3: Input, Output, Rejection

```
wjwkfblwebg2; int ab, float cd, ef;
ewl2efo24hg2jhrto;ty;
int ght,asjhew[37],fuir,gj[45]; sdkvbwrkb;
float ire,dehj[80];
sdvjkjkw
==========================================
float cd, ef;
int ght,asjhew[37],fuir,gj[45];
float ire,dehj[80];
==========================================
wjwkfblwebg2; int ab,
ewl2efo24hg2jhrto;ty;
 sdkvbwrkb;
sdvjkjkw
```

LEX Example 4: Identifiers, Reserved Words, and Constants (id-hex-oct-int-1.lex)

```
%{
int hex = 0; int oct = 0; int regular =0;
%}
letter           [a-zA-Z_]
digit            [0-9]
digits           {digit}+
digit_oct        [0-7]
digit_hex        [0-9A-F]
int_qualifier    [uUlL]
blanks          [ \t]+
identifier       {letter}({letter}|{digit})*
integer         {digits}{int_qualifier}?
hex_const       0[xX]{digit_hex}+{int_qualifier}?
oct_const       0{digit_oct}+{int_qualifier}?
```

LEX Example 4: (contd.)

```
%%
if            {printf("reserved word:%s\n",yytext);}
else          {printf("reserved word:%s\n",yytext);}
while         {printf("reserved word:%s\n",yytext);}
switch        {printf("reserved word:%s\n",yytext);}
{identifier} {printf("identifier :%s\n",yytext);}
{hex_const}  {sscanf(yytext,"%i",&hex);
     printf("hex constant: %s = %i\n",yytext,hex);}
{oct_const}  {sscanf(yytext,"%i",&oct);
     printf("oct constant: %s = %i\n",yytext,oct);}
{integer}    {sscanf(yytext,"%i",&regular);
    printf("integer : %s = %i\n",yytext, regular);}
.|\n ;
%%
yywrap(){}
int main(){yylex();}
```

LEX Example 4: Input and Output

```
uorme while
0345LA 456UB 0x786lHABC
b0x34
========================
identifier :uorme
reserved word:while
oct constant: 0345L = 229
identifier :A
integer : 456U = 456
identifier :B
hex constant: 0x786l = 1926
identifier :HABC
identifier :b0x34
```

LEX Example 5: Floats in C (C-floats.lex)

```
digits          [0-9]+
exp             ([Ee](\+|\-)?{digits})
blanks          [ \t\n]+
float_qual      [fFlL]
%%
{digits}{exp}{float_qual}?/{blanks}
        {printf("float no fraction:%s\n",yytext);}
[0-9]*\.{digits}{exp}?{float_qual}?/{blanks}
        {printf("float with optional
                 integer part :%s\n",yytext);}
{digits}\.[0-9]*{exp}?{float_qual}?/{blanks}
        {printf("float with
                 optional fraction:%s\n",yytext);}
.|\n         ;
%%
yywrap(){} int main(){yylex();}
```

LEX Example 5: Input and Output

```
123 345.. 4565.3 675e-5 523.4e+2 98.1e5  234.3.4
345.   .234E+09L  987E-6F  5432.E7l
============================================
float with optional integer part : 4565.3
float no fraction: 675e-5
float with optional integer part : 523.4e+2
float with optional integer part : 98.1e5
float with optional integer part : 3.4
float with optional fraction: 345.
float with optional integer part : .234E+09L
float no fraction: 987E-6F
float with optional fraction: 5432.E7l
```

LEX Example 6: LA for Desk Calculator

```
number [0-9]+\.?|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] {/* skip blanks */}
{number} {sscanf(yytext,"%lf",&yylval.dval);
            return NUMBER;}
{name} {struct symtab *sp =symlook(yytext);
             yylval.symp = sp; return NAME;}
"++" {return POSTPLUS;}
"--" {return POSTMINUS;}
"$" {return 0;}
\n|. {return yytext[0];}
```

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing

Part - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis?
- Specification of programming languages: context-free grammars
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Grammars

- Every programming language has precise grammar rules that describe the syntactic structure of well-formed programs
  - In C, the rules state how functions are made out of parameter lists, declarations, and statements; how statements are made of expressions, etc.
- Grammars are easy to understand, and parsers for programming languages can be constructed automatically from certain classes of grammars
- Parsers or syntax analyzers are generated *for* a particular grammar
- Context-free grammars are usually used for syntax specification of programming languages

What is Parsing or Syntax Analysis?

- A parser for a grammar of a programming language
  - verifies that the string of tokens for a program in that language can indeed be generated from that grammar
  - reports any syntax errors in the program
  - constructs a parse tree representation of the program (not necessarily explicit)
  - usually calls the lexical analyzer to supply a token to it when necessary
  - could be hand-written or automatically generated
  - is based on *context-free* grammars
- Grammars are generative mechanisms like regular expressions
- Pushdown automata are machines recognizing context-free languages (like FSA for RL)

Context-free Grammars

- A CFG is denoted as $G = (N, T, P, S)$
  - N: Finite set of non-terminals
  - T: Finite set of terminals
  - $S \in N$: The start symbol
  - P: Finite set of productions, each of the form $A \rightarrow \alpha$, where $A \in N$ and $\alpha \in (N \cup T)^*$
- Usually, only P is specified and the first production corresponds to that of the start symbol
- Examples

(1)	(2)	(3)	(4)
$E \rightarrow E + E$	$S \rightarrow 0S0$	$S \rightarrow aSb$	$S \rightarrow aB \mid bA$
$E \rightarrow E * E$	$S \rightarrow 1S1$	$S \rightarrow \epsilon$	$A \rightarrow a \mid aS \mid bAA$
$E \rightarrow (E)$	$S \rightarrow 0$		$B \rightarrow b \mid bS \mid aBB$
$E \rightarrow id$	$S \rightarrow 1$		
	$S \rightarrow \epsilon$		

Derivations

- $E \Rightarrow^{E \rightarrow E+E} E + E \Rightarrow^{E \rightarrow id} id + E \Rightarrow^{E \rightarrow id} id + id$
  is a derivation of the terminal string $id + id$ from E
- In a derivation, a production is applied at each step, to replace a nonterminal by the right-hand side of the corresponding production
- In the above example, the productions $E \rightarrow E + E$, $E \rightarrow id$, and $E \rightarrow id$, are applied at steps 1,2, and, 3 respectively
- The above derivation is represented in short as, $E \Rightarrow^{*} id + id$, and is read as S **derives** $id + id$

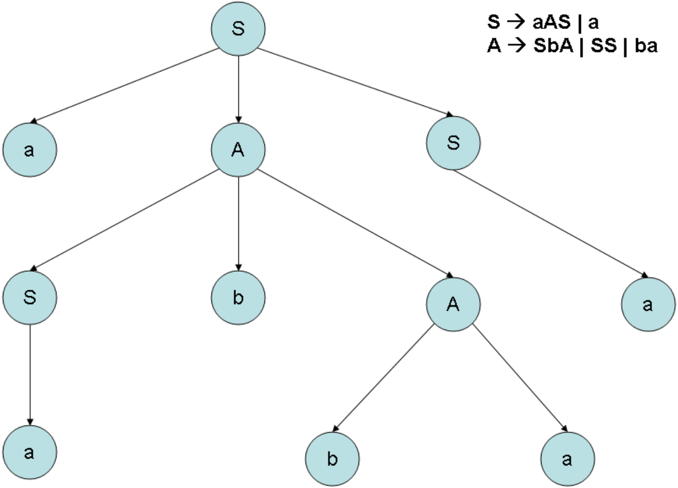
Context-free Languages

- Context-free grammars generate context-free languages (grammar and language resp.)
- The *language generated by* G, denoted $L(G)$, is
  $L(G) = \{w \mid w \ \in T^*, \ and\ S \Rightarrow^* \ w\}$
  i.e., a string is in $L(G)$, if
  1. the string consists solely of terminals
  2. the string can be derived from S
- Examples
  1. $L(G_1)$ = Set of all expressions with +, *, names, and balanced '(' and ')'
  2. $L(G_2)$ = Set of palindromes over 0 and 1
  3. $L(G_3) = \{a^n b^n \mid n \geq 0\}$
  4. $L(G_4) = \{x \mid x\ has\ equal\ no.\ of\ a's\ and\ b's\}$
- A string $\alpha \ \in \ (N \cup T)^*$ is a **sentential form** if $S \Rightarrow^* \ \alpha$
- Two grammars G_1 and G_2 are equivalent, if $L(G_1) = L(G_2)$

Derivation Trees

- Derivations can be displayed as trees
- The internal nodes of the tree are all nonterminals and the leaves are all terminals
- Corresponding to each internal node A, there exists a production $\in$ P, with the RHS of the production being the list of children of A, read from left to right
- The **yield** of a derivation tree is the list of the labels of all the leaves read from left to right
- If α is the yield of some derivation tree for a grammar G, then $S \Rightarrow^* \alpha$ and conversely

Derivation Tree Example



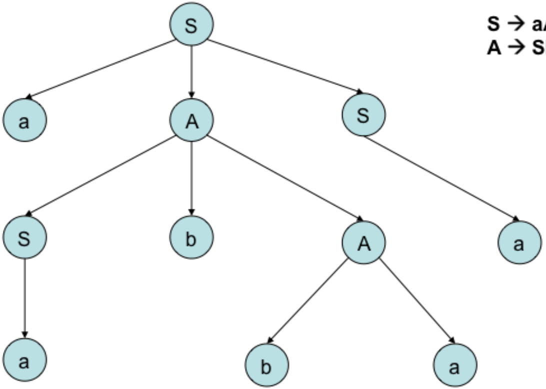
S → aAS | a
A → SbA | SS | ba

S => aAS => aSbAS => aabAS => aabbaS => aabbaa

Leftmost and Rightmost Derivations

- If at each step in a derivation, a production is applied to the leftmost nonterminal, then the derivation is said to be **leftmost**. Similarly **rightmost derivation**.
- If $w \in L(G)$ for some G, then w has at least one parse tree and corresponding to a parse tree, w has unique leftmost and rightmost derivations
- If some word w in $L(G)$ has two or more parse trees, then G is said to be **ambiguous**
- A CFL for which every G is ambiguous, is said to be an **inherently ambiguous** CFL

Leftmost and Rightmost Derivations: An Example



S → aAS | a
A → SbA | SS | ba

Leftmost derivation: **S => aAS => aSbAS => aabAS => aabbaS => aabbaa**

Rightmost derivation: **S => aAS => aAa => aSbAa => aSbbaa => aabbaa**

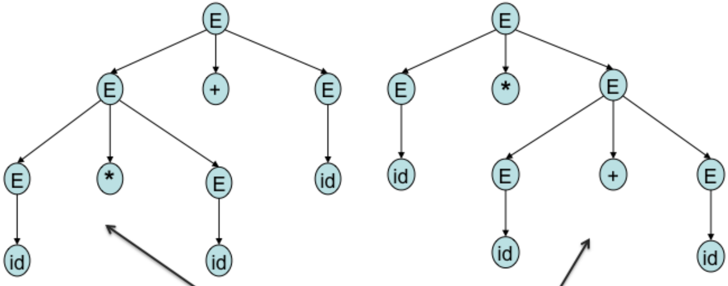
Ambiguous Grammar Examples

- The grammar, $E \rightarrow E + E | E * E | (E) | id$
  is ambiguous, but the following grammar for the same language is unambiguous
  $E \rightarrow E + T | T,\ T \rightarrow T * F | F,\ F \rightarrow (E) | id$
- The grammar,
  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ stmt\ ELSE\ stmt | other\_stmt$

  is ambiguous, but the following equivalent grammar is not

  $stmt \rightarrow IF\ expr\ stmt | IF\ expr\ matched\_stmt\ ELSE\ stmt$
  $matched\_stmt \rightarrow$
  $IF\ expr\ matched\_stmt\ ELSE\ matched\_stmt | other\_stmt$
- The language,
  $L = \{a^n b^n c^m d^m \mid n, m \geq 1\} \cup \{a^n b^m c^m d^n \mid n, m \geq 1\}$,
  is inherently ambiguous

Ambiguity Example 1

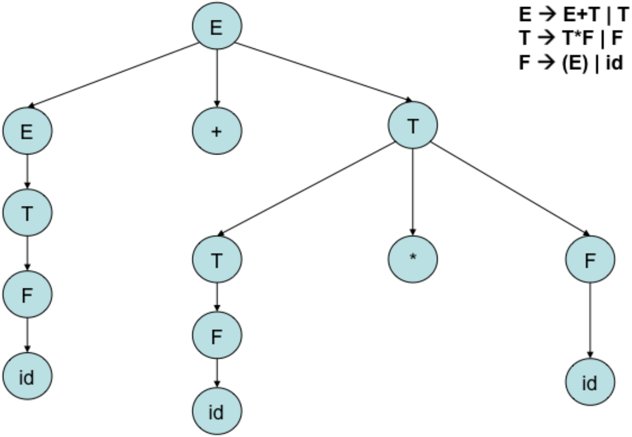


E => E+E => E*E+E => id*E+E => id*id+E => id*id+id

E => E*E => id*E => id*E+E => id*id+E => id*id+id

E → E+E | E*E | (E) | id

Equivalent Unambiguous Grammar



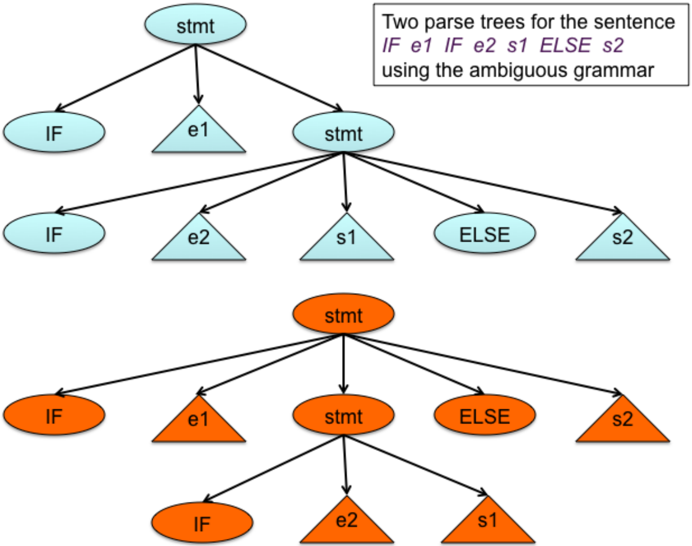
E → E+T | T
T → T*F | F
F → (E) | id

E => E+T => T+T => F+T => id+T => id+T*F => id+F*F => id+id*F => id+id*id

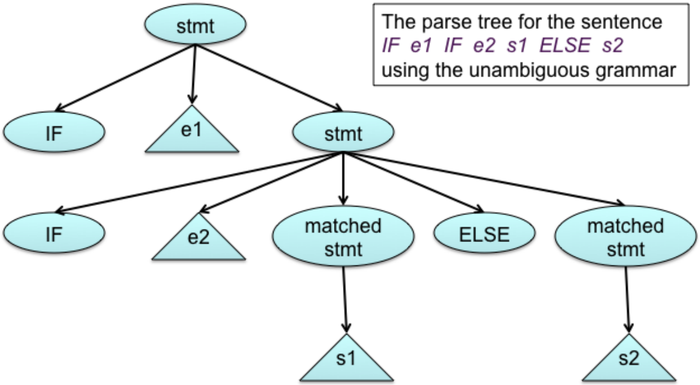
E => T*F => F*F => (E)*F => (E+T)*F => (T+T)*F => (F+T)*F => (id+T)*F
=> (id+F)*id => (id+id)*F => (id+id)*id

Ambiguity Example 2

Two parse trees for the sentence
IF e1 IF e2 s1 ELSE s2
using the ambiguous grammar



Ambiguity Example 2 (contd.)



The parse tree for the sentence
IF e1 IF e2 s1 ELSE s2
using the unambiguous grammar

s→ IF e s | IF e ms ELSE s
ms → IF e ms ELSE ms | other_s

Fragment of C-Grammar (Statements)

```
program --> VOID MAIN '(' ')' compound_stmt
compound_stmt --> '{' '}' | '{' stmt_list '}'
          | '{' declaration_list stmt_list '}'
stmt_list --> stmt | stmt_list stmt
stmt --> compound_stmt| expression_stmt
            | if_stmt | while_stmt
expression_stmt --> ';'| expression ';'
if_stmt --> IF '(' expression ')' stmt
   | IF '(' expression ')' stmt ELSE stmt
while_stmt --> WHILE '(' expression ')' stmt
expression --> assignment_expr
             | expression ',' assignment_expr
```

Fragment of C-Grammar (Expressions)

```
assignment_expr --> logical_or_expr
        | unary_expr  assign_op  assignment_expr
assign_op --> '='| MUL_ASSIGN| DIV_ASSIGN
                 | ADD_ASSIGN| SUB_ASSIGN
                 | AND_ASSIGN| OR_ASSIGN
unary_expr --> primary_expr
        | unary_operator   unary_expr
unary_operator --> '+'| '-'| '!'
primary_expr --> ID| NUM| '(' expression ')'
logical_or_expr --> logical_and_expr
        | logical_or_expr   OR_OP logical_and_expr
logical_and_expr --> equality_expr
        | logical_and_expr   AND_OP  equality_expr
equality_expr --> relational_expr
             | equality_expr EQ_OP relational_expr
             | equality_expr NE_OP relational_expr
```

Fragment of C-Grammar (Expressions and Declarations)

```
relational_expr --> add_expr
                 | relational_expr '<' add_expr
                 | relational_expr '>' add_expr
                 | relational_expr LE_OP add_expr
                 | relational_expr GE_OP add_expr
add_expr --> mult_expr| add_expr '+' mult_expr
                      | add_expr '-' mult_expr
mult_expr --> unary_expr| mult_expr '*' unary_expr
                        | mult_expr '/' unary_expr
declarationlist --> declaration
                 | declarationlist  declaration
declaration --> type  idlist ';'
idlist --> idlist ',' ID | ID
type --> INT_TYPE | FLOAT_TYPE | CHAR_TYPE
```

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Pushdown Automata (contd.)

- The leftmost symbol of γ_i will be the new top of stack
- a in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\},\ \ \delta(q_1, 0, 0) = \{(q_1, 00)\},$
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\},\ \ \delta(q_2, 1, 0) = \{(q_2, \epsilon)\},$
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Pushdown Automata

A PDA M is a system $(Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Γ is the stack alphabet
- $q_0 \in Q$ is the start state
- $z_0 \in \Gamma$ is the start symbol on stack (initialization)
- $F \subseteq Q$ is the set of final states
- δ is the transition function, $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

A typical entry of δ is given by
$\delta(q, a, z) = \{(p_1, \gamma_1), ((p_2, \gamma_2), ..., (p_m, \gamma_m)\}$
The PDA in state q, with input symbol a and top-of-stack symbol z, can enter any of the states p_i, replace the symbol z by the string γ_i, and advance the input head by one symbol.

Pushdown Automata (contd.)

- The leftmost symbol of γ_i will be the new top of stack
- *a* in the above function δ could be ϵ, in which case, the input symbol is not used and the input head is not advanced
- For a PDA M, we define $L(M)$, the language accepted by M **by final state**, to be $L(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \gamma)$, for some $p \in F$ and $\gamma \in \Gamma^*\}$
- We define $N(M)$, the language accepted by M **by empty stack**, to be $N(M) = \{w \mid (q_0, w, Z_0) \vdash^* (p, \epsilon, \epsilon)$, for some $p \in Q$
- When acceptance is by empty stack, the set of final states is irrelevant, and usually, we set $F = \phi$

PDA - Examples

- $L = \{0^n 1^n \mid n \geq 0\}$
  $M = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{Z, 0\}, \delta, q_0, Z, \{q_0\})$, where δ is defined as follows
  $\delta(q_0, 0, Z) = \{(q_1, 0Z)\},\ \ \delta(q_1, 0, 0) = \{(q_1, 00)\},$
  $\delta(q_1, 1, 0) = \{(q_2, \epsilon)\},\ \ \delta(q_2, 1, 0) = \{(q_2, \epsilon)\},$
  $\delta(q_2, \epsilon, Z) = \{(q_0, \epsilon)\}$
- $(q_0, 0011, Z) \vdash (q_1, 011, 0Z) \vdash (q_1, 11, 00Z) \vdash (q_2, 1, 0Z) \vdash (q_2, \epsilon, Z) \vdash (q_0, \epsilon, \epsilon)$
- $(q_0, 001, Z) \vdash (q_1, 01, 0Z) \vdash (q_1, 1, 00Z) \vdash (q_2, \epsilon, 0Z) \vdash$ *error*
- $(q_0, 010, Z) \vdash (q_1, 10, 0Z) \vdash (q_2, 0, Z) \vdash$ *error*

PDA - Examples (contd.)

- $L = \{ww^R \mid w \in \{a, b\}^+\}$
  $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{Z, a, b\}, \delta, q_0, Z, \{q_2\})$, where δ is defined as follows
  $\delta(q_0, a, Z) = \{(q_0, aZ)\}, \ \delta(q_0, b, Z) = \{(q_0, bZ)\},$
  $\delta(q_0, a, a) = \{(q_0, aa), (q_1, \epsilon)\}, \ \delta(q_0, a, b) = \{(q_0, ab)\},$
  $\delta(q_0, b, a) = \{(q_0, ba)\}, \delta(q_0, b, b) = \{(q_0, bb), (q_1, \epsilon)\},$
  $\delta(q_1, a, a) = \{(q_1, \epsilon)\}, \ \ \delta(q_1, b, b) = \{(q_1, \epsilon)\},$
  $\delta(q_1, \epsilon, Z) = \{(q_2, \epsilon)\}$
- $(q_0, abba, Z) \vdash (q_0, bba, aZ) \vdash (q_0, ba, baZ) \vdash (q_1, a, aZ) \vdash (q_1, \epsilon, Z) \vdash (q_2, \epsilon, \epsilon)$
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_0, a, aaZ) \vdash (q_1, \epsilon, aZ) \vdash$ *error*
- $(q_0, aaa, Z) \vdash (q_0, aa, aZ) \vdash (q_1, a, Z) \vdash$ *error*

Nondeterministic and Deterministic PDA

- Just as in the case of NFA and DFA, PDA also have two versions: NPDA and DPDA
- However, NPDA are strictly more powerful than the DPDA
- For example, the language, $L = \{ww^R \mid w \in \{a, b\}^+\}$ can be recognized only by an NPDA and not by any DPDA
- In the same breath, the language, $L = \{wcw^R \mid w \in \{a, b\}^+\}$, can be recognized by a DPDA
- In practice we need DPDA, since they have exactly one possible move at any instant
- Our parsers are all DPDA

Parsing

- Parsing is the process of constructing a parse tree for a sentence generated by a given grammar
- If there are no restrictions on the language and the form of grammar used, parsers for context-free languages require $O(n^3)$ time (n being the length of the string parsed)
  - Cocke-Younger-Kasami's algorithm
  - Earley's algorithm
- Subsets of context-free languages typically require $O(n)$ time
  - Predictive parsing using $LL(1)$ grammars (top-down parsing method)
  - Shift-Reduce parsing using $LR(1)$ grammars (bottom-up parsing method)

Top-Down Parsing using LL Grammars

- Top-down parsing using predictive parsing, traces the left-most derivation of the string while constructing the parse tree
- Starts from the start symbol of the grammar, and “predicts” the next production used in the derivation
- Such “prediction” is aided by parsing tables (constructed off-line)
- The next production to be used in the derivation is determined using the next input symbol to lookup the parsing table (look-ahead symbol)
- Placing restrictions on the grammar ensures that no slot in the parsing table contains more than one production
- At the time of parsing table constrcution, if two productions become eligible to be placed in the same slot of the parsing table, the grammar is declared unfit for predictive parsing

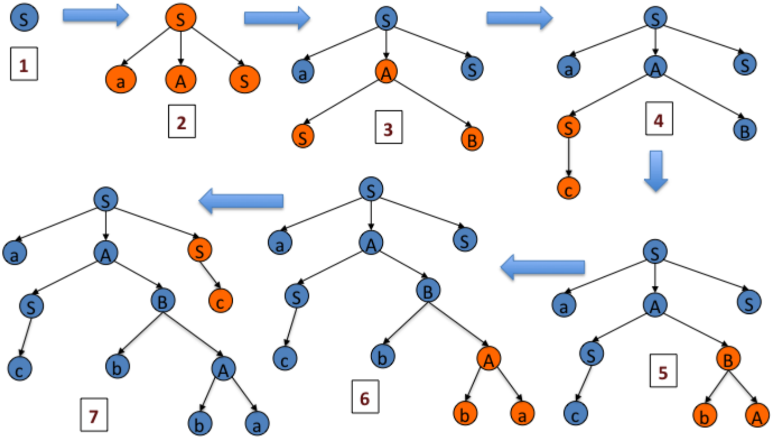
Top-Down LL-Parsing Example

S → aAS | c
A → ba | SB
B → bA | S

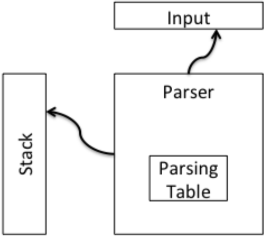
Leftmost derivation of the string *acbbac*

S => aAS => aSBS => acBS => acbAS => acbbaS => acbbac

1 2 3 4 5 6 7



LL(1) Parsing Algorithm



```
Initial configuration: Stack = S, Input = w$,
where, S = start symbol, $ = end of file marker
repeat {
   let X be the top stack symbol;
   let a be the next input symbol /*may be $*/;
   if X is a terminal symbol or $ then
        if X == a then {
             pop X from Stack;
             remove a from input;
        } else ERROR();
   else  /* X is a non-terminal symbol */
        if M[X,a] == X → Y_1Y_2... Y_k then {
              pop X from Stack;
              push Y_k, Y_k-1, ..., Y_1 onto Stack;
                     (Y_1 on top)
        }
} until Stack has emptied;
```

LL(1) Parsing Algorithm Example

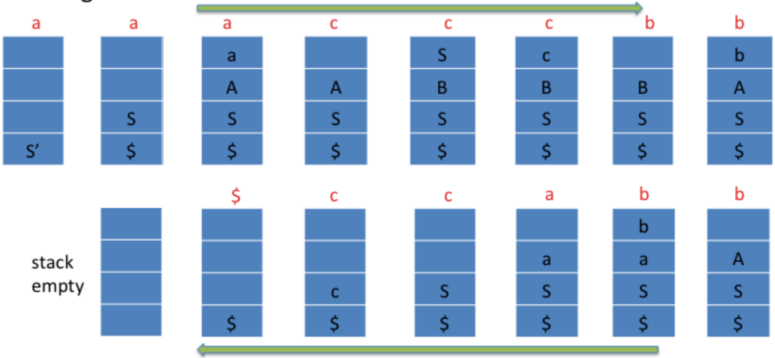
Grammar

S′ → S$
S → aAS | c
A → ba | SB
B → bA | S

string: *acbbac*

LL(1) Parsing Table

	a	b	c	$
S′	S′ → S$		S′ → S$	
S	S → aAS		S → c	
A	A → SB	A → ba	A → SB	
B	B → S	B → bA	B → S	



Strong LL(k) Grammars

Let the given grammar be G

- Input is extended with k symbols, $\$^k$, k is the lookahead of the grammar
- Introduce a new nonterminal S', and a production, $S' \rightarrow S\$^k$, where S is the start symbol of the given grammar
- Consider leftmost derivations only and assume that the grammar has no *useless symbols*
- A production $A \rightarrow \alpha$ in G is called a *strong LL*(k) production, if in G
  $S' \Rightarrow^* wA\gamma \Rightarrow w\alpha\gamma \Rightarrow^* wzy$
  $S' \Rightarrow^* w'A\delta \Rightarrow w'\beta\delta \Rightarrow^* w'zx$
  $|z| = k,\ z \in \Sigma^*, w \text{ and } w' \in \Sigma^*$, then $\alpha = \beta$
- A grammar (nonterminal) is strong LL(k) if all its productions are strong LL(k)

Strong LL(k) Grammars (contd.)

- Strong LL(k) grammars do not allow different productions of the same nonterminal to be used even in two different derivations, if the first *k* symbols of the strings produced by $\alpha\gamma$ and $\beta\delta$ are the same
- Example: $S \rightarrow Abc|aAcb,\ \ A \rightarrow \epsilon|b|c$
  S is a strong LL(1) nonterminal
  - $S' \Rightarrow S\$ \Rightarrow Abc\$ \Rightarrow bc\$,\ \ bbc\$$, and $cbc\$$, on application of the productions, $A \rightarrow \epsilon$, $A \rightarrow b$, and, $A \rightarrow c$, respectively. $z = b,\ \ b$, or c, respectively
  - $S' \Rightarrow S\$ \Rightarrow aAcb\$ \Rightarrow acb\$,\ \ abcb\$$, and $accb\$$, on application of the productions, $A \rightarrow \epsilon$, $A \rightarrow b$, and, $A \rightarrow c$, respectively. $z = a$, in all three cases
  - In this case, $w = w' = \epsilon$, $\alpha = Abc$, $\beta = aAcb$, but z is different in the two derivations, in all the derived strings
  - Hence the nonterminal S is strong LL(1)

Strong LL(k) Grammars (contd.)

A is not strong LL(1)

- $S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}c\$$, $w = \epsilon$, $z = b$, $\alpha = \epsilon$ $(A \rightarrow \epsilon)$
  $S' \Rightarrow^* Abc\$ \Rightarrow \underline{b}bc\$$, $w' = \epsilon$, $z = b$, $\beta = b$ $(A \rightarrow b)$
- Even though the lookaheads are the same ($z = b$), $\alpha \neq \beta$, and therefore, the grammar is not strong LL(1)

A is not strong LL(2)

- $S' \Rightarrow^* Abc\$ \Rightarrow \underline{bc}\$$, $w = \epsilon$, $z = bc$, $\alpha = \epsilon$ $(A \rightarrow \epsilon)$
  $S' \Rightarrow^* aAcb\$ \Rightarrow a\underline{bc}b\$$, $w' = a$, $z = bc$, $\beta = b$ $(A \rightarrow b)$
- Even though the lookaheads are the same ($z = bc$), $\alpha \neq \beta$, and therefore, the grammar is not strong LL(2)

A is strong LL(3) because all the six strings (*bc\$, bbc, cbc, cb\$, bcb, ccb*) can be distinguished using 3-symbol lookahead (details are for home work)

Testable Conditions for LL(1)

- We call strong LL(1) as LL(1) from now on and we will not consider lookaheads longer than 1
- The classical condition for LL(1) property uses *FIRST* and *FOLLOW* sets
- If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
  $FIRST(\alpha) = \{a \mid a \in T,\ and\ \alpha \Rightarrow^* ax,\ x \in T^*\}$
  $FIRST(\epsilon) = \{\epsilon\}$
- If A is any nonterminal, then
  $$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*,\ a \in T \cup \{\$\}\}$$
- $FIRST(\alpha)$ is determined by α alone, but $FOLLOW(A)$ is determined by the "context" of A, i.e., the derivations in which A occurs

FIRST and *FOLLOW* Computation Example

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$
- $FIRST(S') = FIRST(S) = \{a, c\}$ because
  $S' \Rightarrow S\$ \Rightarrow \underline{c}\$$, and $S' \Rightarrow S\$ \Rightarrow \underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$$
- $FIRST(A) = \{a, b, c\}$ because
  $A \Rightarrow \underline{b}a$, and $A \Rightarrow SB$, and therefore all symbols in $FIRST(S)$ are in $FIRST(A)$
- $FOLLOW(S) = \{a, b, c, \$\}$ because
  $S' \Rightarrow \underline{S\$}$,
  $S' \Rightarrow^* aAS\$ \Rightarrow a\underline{S}BS\$ \Rightarrow aS\underline{b}AS\$$,
  $S' \Rightarrow^* a\underline{S}BS\$ \Rightarrow a\underline{S}SS\$ \Rightarrow aS\underline{a}ASS\$$,
  $S' \Rightarrow^* a\underline{S}SS\$ \Rightarrow aS\underline{c}S\$$
- $FOLLOW(A) = \{a, c\}$ because
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$$,
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Computation of *FIRST*: Terminals and Nonterminals

```
{
 for each ($a \in T$) FIRST($a$) = $\{a\}$; FIRST($\epsilon$) = $\{\epsilon\}$;
 for each ($A \in N$) FIRST($A$) = $\emptyset$;
 while (FIRST sets are still changing) {
     for each production $p$ {
         Let $p$ be the production $A \rightarrow X_1 X_2 ... X_n$;
         FIRST($A$) = FIRST($A$) $\cup$ (FIRST($X_1$) - $\{\epsilon\}$);
         $i$ = 1;
         while ($\epsilon \in$ FIRST($X_i$) && $i \leq n - 1$) {
             FIRST($A$) = FIRST($A$) $\cup$ (FIRST($X_{i+1} - \{\epsilon\}$); $i++$;
         }
         if ($i == n$) && ($\epsilon \in$ FIRST($X_n$))
             FIRST($A$) = FIRST($A$) $\cup \{\epsilon\}$
 }
}
```

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing

Part - 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) and recursive-descent parsing
- Bottom-up parsing: LR-parsing

Testable Conditions for LL(1)

- We call strong LL(1) as LL(1) from now on and we will not consider lookaheads longer than 1
- The classical condition for LL(1) property uses *FIRST* and *FOLLOW* sets
- If α is any string of grammar symbols ($\alpha \in (N \cup T)^*$), then
  $FIRST(\alpha) = \{a \mid a \in T,\ and\ \alpha \Rightarrow^* ax,\ x \in T^*\}$
  $FIRST(\epsilon) = \{\epsilon\}$
- If A is any nonterminal, then
  $$FOLLOW(A) = \{a \mid S \Rightarrow^* \alpha A a \beta,\ \alpha, \beta \in (N \cup T)^*,\ a \in T \cup \{\$\}\}$$
- $FIRST(\alpha)$ is determined by α alone, but $FOLLOW(A)$ is determined by the "context" of A, i.e., the derivations in which A occurs

FIRST and *FOLLOW* Computation Example

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ B \rightarrow bA \mid S$
- $FIRST(S') = FIRST(S) = \{a, c\}$ because
  $S' \Rightarrow S\$ \Rightarrow \underline{c}\$$, and $S' \Rightarrow S\$ \Rightarrow \underline{a}AS\$ \Rightarrow \underline{a}baS\$ \Rightarrow \underline{a}bac\$$
- $FIRST(A) = \{a, b, c\}$ because
  $A \Rightarrow \underline{b}a$, and $A \Rightarrow SB$, and therefore all symbols in $FIRST(S)$ are in $FIRST(A)$
- $FOLLOW(S) = \{a, b, c, \$\}$ because
  $S' \Rightarrow \underline{S\$}$,
  $S' \Rightarrow^* aAS\$ \Rightarrow a\underline{S}BS\$ \Rightarrow aS\underline{b}AS\$$,
  $S' \Rightarrow^* a\underline{S}BS\$ \Rightarrow a\underline{S}SS\$ \Rightarrow aS\underline{a}ASS\$$,
  $S' \Rightarrow^* a\underline{S}SS\$ \Rightarrow aS\underline{c}S\$$
- $FOLLOW(A) = \{a, c\}$ because
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{a}AS\$$,
  $S' \Rightarrow^* a\underline{A}S\$ \Rightarrow aA\underline{c}$

Computation of *FIRST*: Terminals and Nonterminals

```
{
 for each ($a \in T$) FIRST($a$) = $\{a\}$; FIRST($\epsilon$) = $\{\epsilon\}$;
 for each ($A \in N$) FIRST($A$) = $\emptyset$;
 while (FIRST sets are still changing) {
     for each production $p$ {
         Let $p$ be the production $A \rightarrow X_1 X_2 ... X_n$;
         FIRST($A$) = FIRST($A$) $\cup$ (FIRST($X_1$) - $\{\epsilon\}$);
         $i$ = 1;
         while ($\epsilon \in$ FIRST($X_i$) && $i \leq n - 1$) {
             FIRST($A$) = FIRST($A$) $\cup$ (FIRST($X_{i+1} - \{\epsilon\}$); $i++$;
         }
         if ($i == n$) && ($\epsilon \in$ FIRST($X_n$))
             FIRST($A$) = FIRST($A$) $\cup\{\epsilon\}$
 }
}
```

Computation of *FIRST*(β): β, a string of Grammar Symbols

```
{ /* It is assumed that FIRST sets of terminals and nonterminals
    are already available /*
    FIRST($\beta$) = $\emptyset$;
    while (FIRST sets are still changing) {
        Let $\beta$ be the string $X_1 X_2 ... X_n$;
        FIRST($\beta$) = FIRST($\beta$) $\cup$ (FIRST($X_1$) - {$\epsilon$});
        $i$ = 1;
        while ($\epsilon \in$ FIRST($X_i$) && $i \leq n - 1$) {
            FIRST($\beta$) = FIRST($\beta$) $\cup$ (FIRST($X_{i+1} - \{\epsilon\}$); $i++$;
        }
        if ($i == n$) && ($\epsilon \in$ FIRST($X_n$))
            FIRST($\beta$) = FIRST($\beta$) $\cup \{\epsilon\}$
    }
}
```

FIRST Computation: Algorithm Trace - 1

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid \epsilon,\ A \rightarrow ba \mid SB,\ B \rightarrow cA \mid S$
- Initially, FIRST(S) = FIRST(A) = FIRST(B) = $\emptyset$
- Iteration 1
  - FIRST(S) = $\{a, \epsilon\}$ from the productions $S \rightarrow aAS \mid \epsilon$
  - FIRST(A) = $\{b\} \cup$ FIRST(S) - $\{\epsilon\} \cup$ FIRST(B) - $\{\epsilon\}$ = $\{b, a\}$
    from the productions $A \rightarrow ba \mid SB$
    (since $\epsilon \in$ FIRST(S), FIRST(B) is also included;
    since FIRST(B)=ϕ, ϵ is not included)
  - FIRST(B) = $\{c\} \cup$ FIRST(S) - $\{\epsilon\}$ $\cup\{\epsilon\}$ = $\{c, a, \epsilon\}$
    from the productions $B \rightarrow cA \mid S$
    (ϵ is included because $\epsilon \in$ FIRST(S))

FIRST Computation: Algorithm Trace - 2

- The grammar is
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid \epsilon,\ A \rightarrow ba \mid SB,\ B \rightarrow cA \mid S$
- From the first iteration,
  FIRST(S) = $\{a, \epsilon\}$, FIRST(A) = $\{b, a\}$, FIRST(B) = $\{c, a, \epsilon\}$
- Iteration 2
  (values stabilize and do not change in iteration 3)
  - FIRST(S) = $\{a, \epsilon\}$ (no change from iteration 1)
  - FIRST(A) = $\{b\} \cup$ FIRST(S) - $\{\epsilon\} \cup$ FIRST(B) - $\{\epsilon\} \cup \{\epsilon\}$
    = $\{b, a, c, \epsilon\}$ (changed!)
  - FIRST(B) = $\{c, a, \epsilon\}$ (no change from iteration 1)

Computation of *FOLLOW*

```
{ for each ($X \in N \cup T$) FOLLOW($X$) = $\emptyset$;
  FOLLOW($S$) = {\$}; /* $S$ is the start symbol of the grammar */
  repeat {
    for each production $A \rightarrow X_1X_2...X_n$ {/* $X_i \neq \epsilon$ */
      FOLLOW($X_n$) = FOLLOW($X_n$) $\cup$ FOLLOW($A$);
      REST = FOLLOW($A$);
      for $i$ = $n$ downto 2 {
        if ($\epsilon \in$ FIRST($X_i$)) { FOLLOW($X_{i-1}$) =
              FOLLOW($X_{i-1}$) $\cup$ (FIRST ($X_i$) $- \{\epsilon\}$)$\cup$ REST;
              REST = FOLLOW($X_{i-1}$);
        } else { FOLLOW($X_{i-1}$) = FOLLOW($X_{i-1}$) $\cup$ FIRST ($X_i$) ;
                    REST = FOLLOW($X_{i-1}$); }
      }
    }
  } until no FOLLOW set has changed
}
```

FOLLOW Computation: Algorithm Trace

- Consider the following grammar
  $S' \rightarrow S\$,\ S \rightarrow aAS \mid \epsilon,\ A \rightarrow ba \mid SB,\ B \rightarrow cA \mid S$
- Initially, $follow(S)$ = {\$}; $follow(A) = follow(B) = \emptyset$
  $first(S) = \{a, \epsilon\}$; $first(A) = \{a, b, c, \epsilon\}$; $first(B) = \{a, c, \epsilon\}$;
- Iteration 1 /* In the following, $x \cup = y$ means $x = x \cup y$ */
  - $S \rightarrow aAS$: $follow(S)\cup = \{\$\}$; $rest = follow(S) = \{\$\}$
    $follow(A)\cup = (first(S) - \{\epsilon\}) \cup rest = \{a, \$\}$
  - $A \rightarrow SB$: $follow(B)\cup = follow(A) = \{a, \$\}$
    $rest = follow(A) = \{a, \$\}$
    $follow(S)\cup = (first(B) - \{\epsilon\}) \cup rest = \{a, c, \$\}$
  - $B \rightarrow cA$: $follow(A)\cup = follow(B) = \{a, \$\}$
  - $B \rightarrow S$: $follow(S)\cup = follow(B) = \{a, c, \$\}$
  - At the end of iteration 1
    $follow(S) = \{a, c, \$\}$; $follow(A) = follow(B) = \{a, \$\}$

FOLLOW Computation: Algorithm Trace (contd.)

- $\mathit{first}(S) = \{a, \epsilon\}$; $\mathit{first}(A) = \{a, b, c, \epsilon\}$; $\mathit{first}(B) = \{a, c, \epsilon\}$;
- At the end of iteration 1
  $\mathit{follow}(S) = \{a, c, \$\}$; $\mathit{follow}(A) = \mathit{follow}(B) = \{a, \$\}$
- Iteration 2
- $S \rightarrow aAS$: $\mathit{follow}(S) \cup = \{a, c, \$\}$;
  $\mathit{rest} = \mathit{follow}(S) = \{a, c, \$\}$
  $\mathit{follow}(A) \cup = (\mathit{first}(S) - \{\epsilon\}) \cup \mathit{rest} = \{a, c, \$\}$ (changed!)
- $A \rightarrow SB$: $\mathit{follow}(B) \cup = \mathit{follow}(A) = \{a, c, \$\}$ (changed!)
  $\mathit{rest} = \mathit{follow}(A) = \{a, c, \$\}$
  $\mathit{follow}(S) \cup = (\mathit{first}(B) - \{\epsilon\}) \cup \mathit{rest} = \{a, c, \$\}$ (no change)
- At the end of iteration 2
  $\mathit{follow}(S) = \mathit{follow}(A) = \mathit{follow}(B) = \{a, c, \$\}$;
- The *follow* sets do not change any further

LL(1) Conditions

- Let G be a context-free grammar
- G is LL(1) iff for every pair of productions $A \rightarrow \alpha$ and $A \rightarrow \beta$, the following condition holds
  - $dirsymb(\alpha) \cap dirsymb(\beta) = \emptyset$, where
    $dirsymb(\gamma)$ = if ($\epsilon \in first(\gamma)$) then
    $((first(\gamma) - \{\epsilon\}) \cup follow(A))$ else $first(\gamma)$
    (γ stands for α or β)
  - *dirsymb* stands for "direction symbol set"
- An equivalent formulation (as in ALSU's book) is as below
  - $first(\alpha.follow(A)) \cap first(\beta.follow(A)) = \emptyset$
- Construction of the LL(1) parsing table

```
for each production $A \rightarrow \alpha$
    for each symbol $s \in dirsymb(\alpha)$
    /* s may be either a terminal symbol or $ */
        add $A \rightarrow \alpha$ to $LLPT[A, s]$
Make each undefined entry of LLPT as error
```

LL(1) Table Construction using *FIRST* and *FOLLOW*

```
for each production $A \rightarrow \alpha$
    for each terminal symbol $a \in first(\alpha)$
        add $A \rightarrow \alpha$ to $LLPT[A, a]$
    if $\epsilon \in first(\alpha)$ {
        for each terminal symbol $b \in follow(A)$
            add $A \rightarrow \alpha$ to $LLPT[A, b]$
        if $\$ \in follow(A)$
            add $A \rightarrow \alpha$ to $LLPT[A, \$]$
    }
Make each undefined entry of LLPT as error
```

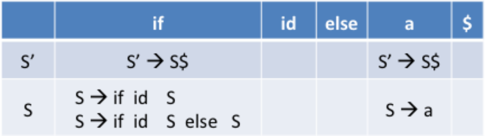
- After the construction of the LL(1) table is complete (following any of the two methods), if any slot in the LL(1) table has two or more productions, then the grammar is NOT LL(1)

Simple Example of LL(1) Grammar

- P1: $S \rightarrow$ *if* (a) *S else S* $|$ *while* (a) S $|$ *begin SL end*
  P2: $SL \rightarrow S\ S'$
  P3: $S' \rightarrow ;\ SL \mid \epsilon$
- {if, while, begin, end, a, (,), ;} are all terminal symbols
- Clearly, all alternatives of P1 start with distinct symbols and hence create no problem
- P2 has no choices
- Regarding P3, dirsymb(;SL) = {;}, and dirsymb(ϵ) = {end}, and the two have no common symbols
- Hence the grammar is LL(1)

LL(1) Table Construction Example 1

LL(1) Parsing Table for the original grammar



	if	id	else	a	$
S’	S’ → S$			S’ → S$	
S	S → if id S S → if id S else S			S → a	

Original Grammar

S’ → S$
S → if id S |
if id S else S |
a

Grammar is not LL(1)

tokens: if, id, else, a

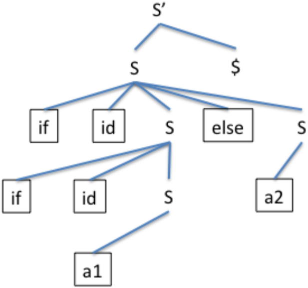
dirsymb(S$) = {if,a}; dirsymb (a) = {a}
dirsymb(if id S) = {if}
dirsymb(if id S else S) = {if}

$$dirsymb(if\ id\ S) \bigcap dirsymb(a) = \varnothing$$

$$dirsymb(if\ id\ S\ else\ S) \bigcap dirsymb(a) = \varnothing$$

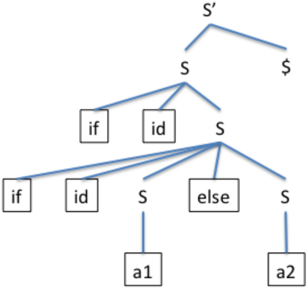
$$dirsymb(if\ id\ S) \bigcap dirsymb(if\ id\ S\ else\ S) \neq \varnothing$$

LL(1) Table Problem Example 1



string: if id (if id a1) else a2

parentheses are not part of the string



string: if id (if id a1 else a2)

parentheses are not part of the string

LL(1) Table Construction Example 2

Original Grammar

S′ → S$
S → if id S |
if id S else S |
a

LL(1) Parsing Table for modified grammar

	if	else	a	$
S′	S′ → S$		S′ → S$	
S	S → if id S S1		S → a	
S1		S1 → ε S1 → else S		S1 → ε

dirsymb(S$) = {if, a}; dirsymb (a) = {a}
dirsymb(if id S S1) = {if}
dirsymb(else S) = {else}
dirsymb(ε) = {else, $}

Left-Factored Grammar

S′ → S$
S → if id S S1 | a
S1 → ε | else S

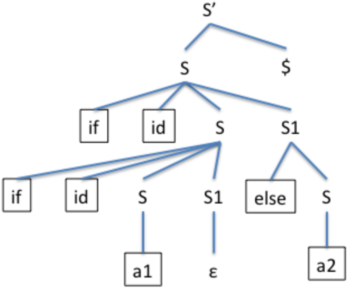
Grammar is not LL(1)

tokens: if, id, else, a

$$dirsymb(if\ id\ S\ S1) \bigcap dirsymb(a) = \varnothing$$

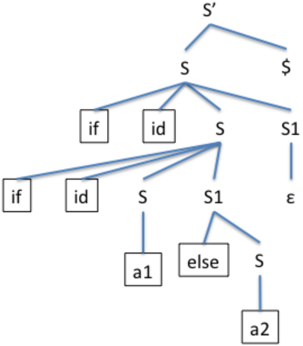
$$dirsymb(\varepsilon) \bigcap dirsymb(else\ S) \neq \varnothing$$

LL(1) Table Problem Example 2



string: if id (if id a1) else a2

parentheses are not part of the string



string: if id (if id a1 else a2)

parentheses are not part of the string

LL(1) Table Construction Example 3

S′ → S$
S → aAS | c
A → ba | SB
B → bA | S
Grammar is LL(1)

LL(1) Parsing Table

	a	b	c	$
S′	S′ → S$		S′ → S$	
S	S → aAS		S → c	
A	A → SB	A → ba	A → SB	
B	B → S	B → bA	B → S	

first(S) = {a,c}
first(A) = {a,b,c}
first(B) = {a,b,c}

$$dirsymb(aAS) \bigcap dirsymb(c) = \varnothing$$

$$dirsymb(ba) \bigcap dirsymb(SB) = \varnothing$$

$$dirsymb(bA) \bigcap dirsymb(S) = \varnothing$$

follow(S) = {a,b,c,$}
follow(A) = {a,c}
follow(B) = {a,c}

dirsymb(S$) = {a,c}
dirsymb(aAS) = {a}
dirsymb(c) = {c}

dirsymb(ba) = {b}
dirsymb(SB) = {a,c}
dirsymb(bA) = {b}
dirsymb(S) = {a,c}

LL(1) Table Construction Example 4

Left-Recursive Grammar for Statement List

S′ → SL $
SL → SL S | S
S → a

dirsymb(SL $) = {a}
dirsymb (a) = {a}
dirsymb(SL S) = {a}
dirsymb(S) = {a}

$$dirsymb(SL\ \ S) \bigcap dirsymb(S) \neq \emptyset$$

LL(1) Parsing Table for Left-Recursive Grammar

	a
S′	S′ → SL $
SL	SL → SL S SL → S
S	S → a

Grammar is not LL(1)

Right-Recursive Grammar for Statement List

S′ → SL $
SL → S A
A → S A | ε
S → a

dirsymb(SL $) = {a}
dirsymb (a) = {a}
dirsymb(S A) = {a}
dirsymb(ε) = {$}

Grammar is LL(1)

LL(1) Parsing Table for Right-Recursive Grammar

	a	$
S′	S′ → SL $	
SL	SL → S A	
A	A → S A	A → ε
S	S → a	

$$dirsymb(S\ A) \bigcap dirsymb(\varepsilon) = \emptyset$$

Elimination of Useless Symbols

Now we study the *grammar transformations*, elimination of useless symbols, elimination of left recursion and left factoring

- Given a grammar $G = (N, T, P, S)$, a non-terminal X is *useful* if $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$, where, $w \in T^*$
  Otherwise, X is useless
- Two conditions have to be met to ensure that X is useful
  1. $X \Rightarrow^* w, w \in T^*$ (X derives some terminal string)
  2. $S \Rightarrow^* \alpha X \beta$ (X occurs in some string derivable from S)
- Example: $S \rightarrow AB \mid CA,\ B \rightarrow BC \mid AB,\ A \rightarrow a,$
  $C \rightarrow aB \mid b,\ D \rightarrow d$
  1. $A \rightarrow a,\ C \rightarrow b,\ D \rightarrow d,\ S \rightarrow CA$
  2. $S \rightarrow CA,\ A \rightarrow a,\ C \rightarrow b$

Testing for $X \Rightarrow^* w$

```
G' = (N',T',P',S') is the new grammar
N_OLD = $\phi$;
N_NEW = {$X \mid X \rightarrow w, w \in T^*$ }
while N_OLD $\neq$ N_NEW do {
  N_OLD = N_NEW;
  N_NEW = N_OLD $\cup \{X \mid X \rightarrow \alpha, \alpha \in (T \cup N\_OLD)^*\}$
}
N' = N_NEW; T' = T; S' = S;
P' = {$p \mid$ all symbols of $p$ are in $N' \cup T'$}
```

Testing for $S \Rightarrow^* \alpha X \beta$

```
G' = (N',T',P',S') is the new grammar
N' = {S};
Repeat {
  for each production $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$ with $A \in N'$ do
     add all nonterminals of $\alpha_1, \alpha_2, ..., \alpha_n$ to N' and
     all terminals of $\alpha_1, \alpha_2, ..., \alpha_n$ to T'
} until there is no change in N' and T'
P' = {$p \mid$ all symbols of $p$ are in $N' \cup T'$};  S' = S
```

Syntax Analysis:
Context-free Grammars, Pushdown Automata and Parsing
Part - 4

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)
- Recursive-descent parsing
- Bottom-up parsing: LR-parsing

Elimination of Left Recursion

- A *left-recursive* grammar has a non-terminal A such that $A \Rightarrow^{+} A\alpha$
- Top-down parsing methods (LL(1) and RD) cannot handle reft-recursive grammars
- Left-recursion in grammars can be eliminated by transformations
- A simpler case is that of grammars with *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$
    - Two productions $A \rightarrow A\alpha \mid \beta$ can be transformed to $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$
    - In general, a group of productions:
      $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$
      can be transformed to
      $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_n A'$, $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_m A' \mid \epsilon$

Left Recursion Elimination - An Example

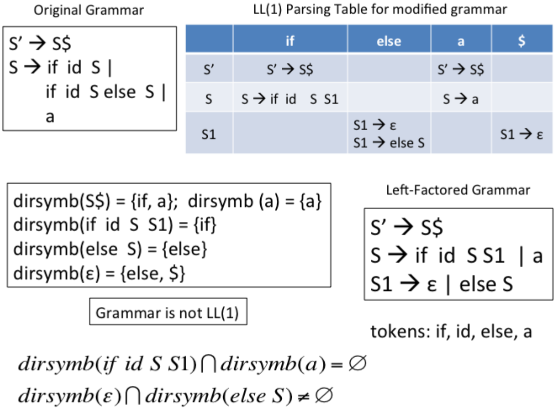
$A \rightarrow A\alpha \mid \beta \Rightarrow A \rightarrow \beta A',\ A' \rightarrow \alpha A' \mid \epsilon$

- The following grammar for regular expressions is ambiguous:
  $E \rightarrow E + E \mid E\ E \mid E* \mid (E) \mid a \mid b$
- Equivalent left-recursive but unambiguous grammar is:
  $E \rightarrow E + T \mid T,\ T \rightarrow T\ F \mid F,\ F \rightarrow F* \mid P,\ P \rightarrow (E) \mid a \mid b$
- Equivalent non-left-recursive grammar is:
  $E \rightarrow T\ E',\ E' \rightarrow +T\ E' \mid \epsilon,\ T \rightarrow F\ T',\ T' \rightarrow F\ T' \mid \epsilon,$
  $F \rightarrow P\ F',\ F' \rightarrow *F' \mid \epsilon,\ P \rightarrow (E) \mid a \mid b$

Left Factoring

- If two alternatives of a production begin with the same string, then the grammar is not LL(1)
- Example: $S \rightarrow 0S1 \mid 01$ *is not LL(1)*
  - After left factoring: $S \rightarrow 0S'$, $S' \rightarrow S1 \mid 1$ *is LL(1)*
- General method: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \Rightarrow A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$
- Another example: a grammar for logical expressions is given below
  $E \rightarrow T\ or\ E \mid T$, $T \rightarrow F\ and\ T \mid F$,
  $F \rightarrow not\ F \mid (E) \mid true \mid false$
  - This grammar is not LL(1) but becomes LL(1) after left factoring
  - $E \rightarrow TE'$, $E' \rightarrow or\ E \mid \epsilon$, $T \rightarrow FT'$, $T' \rightarrow and\ T \mid \epsilon$,
    $F \rightarrow not\ F \mid (E) \mid true \mid false$

Grammar Transformations may not help!



Original Grammar

S′ → S$
S → if id S |
if id S else S |
a

LL(1) Parsing Table for modified grammar

	if	else	a	$
S′	S′ → S$		S′ → S$	
S	S → if id S S1		S → a	
S1		S1 → ε S1 → else S		S1 → ε

dirsymb(S$) = {if, a}; dirsymb (a) = {a}
dirsymb(if id S S1) = {if}
dirsymb(else S) = {else}
dirsymb(ε) = {else, $}

Grammar is not LL(1)

Left-Factored Grammar

S′ → S$
S → if id S S1 | a
S1 → ε | else S

tokens: if, id, else, a

$$dirsymb(if\ id\ S\ S1) \bigcap dirsymb(a) = \emptyset$$

$$dirsymb(\varepsilon) \bigcap dirsymb(else\ S) \neq \emptyset$$

Choose *S*1 → *else S* instead of *S*1 → ϵ on lookahead *else*.
This resolves the conflict. Associates *else* with the innermost *if*

Recursive-Descent Parsing

- Top-down parsing strategy
- One function/procedure for each nonterminal
- Functions call each other recursively, based on the grammar
- Recursion stack handles the tasks of LL(1) parser stack
- LL(1) conditions to be satisfied for the grammar
- Can be automatically generated from the grammar
- Hand-coding is also easy
- Error recovery is superior

An Example

Grammar: $S' \rightarrow S\$$, $S \rightarrow aAS \mid c$, $A \rightarrow ba \mid SB$, $B \rightarrow bA \mid S$

```
/* function for nonterminal S' */
void main(){/* S' --> S$ */
   fS(); if (token == eof) accept();
         else error();
}
/* function for nonterminal S */
void fS(){/* S --> aAS | c */
   switch token {
     case a : get_token(); fA(); fS();
              break;
     case c : get_token(); break;
     others : error();
   }
}
```

An Example (contd.)

```
void fA(){/* A --> ba | SB */
   switch token {
     case b : get_token();
              if (token == a) get_token();
              else error(); break;
     case a,c : fS(); fB(); break;
     others : error();
   }
}
void fB(){/* B --> bA | S */
   switch token {
     case b : get_token(); fA(); break;
     case a,c : fS(); break;
     others : error();
   }
}
```

Automatic Generation of RD Parsers

- Scheme is based on structure of productions
- Grammar must satisfy LL(1) conditions
- function *get_token*() obtains the next token from the lexical analyzer and places it in the global variable *token*
- function *error*() prints out a suitable error message
- In the next slide, for each grammar component, the code that must be generated is shown

Automatic Generation of RD Parsers (contd.)

1. ϵ : ;
2. $a \in T$: if (*token* == a) *get_token*(); else *error*();
3. $A \in N$: *fA*(); /* function call for nonterminal A */
4. $\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

   ```
   switch token {
     case dirsym($\alpha_1$): program_segment($\alpha_1$); break;
     case dirsym($\alpha_2$): program_segment($\alpha_2$); break;
     ...
     others: error();
   }
   ```

5. $\alpha_1 \alpha_2 \ldots \alpha_n$:
   program_segment(α_1); program_segment(α_2); ... ;
   program_segment(α_n);
6. $A \rightarrow \alpha$: void *fA*() { program_segment(α); }

Bottom-Up Parsing

- Begin at the leaves, build the parse tree in small segments, combine the small trees to make bigger trees, until the root is reached
- This process is called *reduction* of the sentence to the start symbol of the grammar
- One of the ways of “reducing” a sentence is to follow the rightmost derivation of the sentence in *reverse*
  - *Shift-Reduce* parsing implements such a strategy
  - It uses the concept of a *handle* to detect when to perform reductions

Shift-Reduce Parsing

- **Handle**: A *handle* of a right sentential form γ, is a production $A \rightarrow \beta$ *and* a position in γ, where the string β may be found and replaced by A, to produce the previous right sentential form in a rightmost derivation of γ
  That is, if $S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$, then $A \rightarrow \beta$ in the position following α is a handle of $\alpha\beta w$
- A handle will always eventually appear on the top of the stack, never submerged inside the stack
- In S-R parsing, we locate the handle and reduce it by the LHS of the production repeatedly, to reach the start symbol
- These reductions, in fact, trace out a rightmost derivation of the sentence in reverse. This process is called handle pruning
- *LR-Parsing* is a method of shift-reduce parsing

Examples

1. $S \rightarrow aAcBe$, $A \rightarrow Ab \mid b$, $B \rightarrow d$
   For the string = *abbcde*, the rightmost derivation marked with handles is shown below

$$
\begin{aligned}
S \; & \Rightarrow \underline{aAcBe} \;\; (aAcBe,\ S \rightarrow aAcBe) \\
& \Rightarrow aAc\underline{d}e \;\; (d,\ \ B \rightarrow d) \\
& \Rightarrow a\underline{Ab}cde \;\; (Ab,\ \ A \rightarrow Ab) \\
& \Rightarrow a\underline{b}bcde \;\; (b,\ \ A \rightarrow b)
\end{aligned}
$$

The handle is unique if the grammar is unambiguous!

Examples (contd.)

2. $S \rightarrow aAS \mid c,\ A \rightarrow ba \mid SB,\ \ B \rightarrow bA \mid S$
   For the string = *acbbac*, the rightmost derivation marked with handles is shown below

$$
\begin{aligned}
S \quad & \Rightarrow \underline{aAS}\ \ (aAS,\ S \rightarrow aAS) \\
& \Rightarrow aA\underline{c}\ \ (c,\ \ S \rightarrow c) \\
& \Rightarrow a\underline{SB}c\ \ (SB,\ \ A \rightarrow SB) \\
& \Rightarrow aS\underline{bA}c\ \ (bA,\ \ B \rightarrow bA) \\
& \Rightarrow aSb\underline{ba}c\ \ (ba,\ \ A \rightarrow ba) \\
& \Rightarrow a\underline{c}bbac\ \ (c,\ \ S \rightarrow c)
\end{aligned}
$$

Examples (contd.)

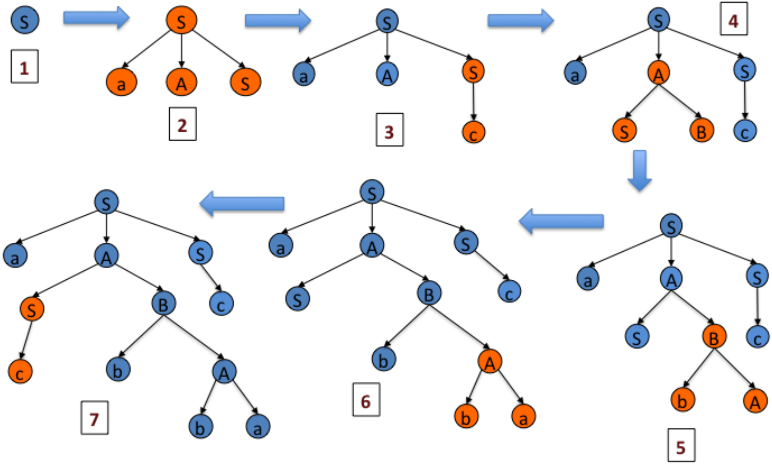
3. $E \rightarrow E + E$, $E \rightarrow E * E$, $E \rightarrow (E)$, $E \rightarrow id$
   For the string = $id + id * id$, two rightmost derivation marked with handles are shown below

$$
\begin{aligned}
E \quad & \Rightarrow \underline{E + E} \;\; (E + E,\;\; E \rightarrow E + E) \\
& \Rightarrow E + \underline{E * E} \;\; (E * E,\;\; E \rightarrow E * E) \\
& \Rightarrow E + E * \underline{id} \;\; (id,\;\; E \rightarrow id) \\
& \Rightarrow E + \underline{id} * id \;\; (id,\;\; E \rightarrow id) \\
& \Rightarrow \underline{id} + id * id \;\; (id,\;\; E \rightarrow id) \\
E \quad & \Rightarrow \underline{E * E} \;\; (E * E,\;\; E \rightarrow E * E) \\
& \Rightarrow E * \underline{id} \;\; (id,\;\; E \rightarrow id) \\
& \Rightarrow \underline{E + E} * id \;\; (E + E,\;\; E \rightarrow E + E) \\
& \Rightarrow E + \underline{id} * id \;\; (id,\;\; E \rightarrow id) \\
& \Rightarrow \underline{id} + id * id \;\; (id,\;\; E \rightarrow id)
\end{aligned}
$$

Rightmost Derivation and Bottom-UP Parsing

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac*
S => aAS => aAc => aSBc => aSbAc => aSbbac => acbbac
1 2 3 4 5 6 7

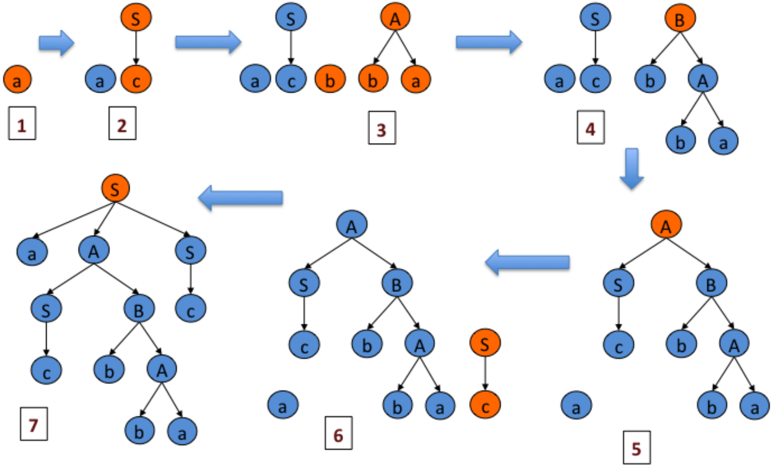


Rightmost Derivation and Bottom-UP Parsing (contd.)

S → aAS | c
A → ba | SB
B → bA | S

Rightmost derivation of the string *acbbac* in reverse

S <= aAS <= aAc <= aSBc <= aSbAc <= aSbbac <= acbbac
7 6 5 4 3 2 1



Shift-Reduce Parsing Algorithm

- How do we locate a handle in a right sentential form?
  - An LR parser uses a DFA to detect the condition that a handle is now on the stack
- Which production to use, in case there is more than one with the same RHS?
  - An LR parser uses a parsing table similar to an LL parsing table, to choose the production
- A stack is used to implement an S-R parser, The parser has four actions
  1. **shift**: the next input symbol is shifted to the top of stack
  2. **reduce**: the right end of the handle is the top of stack; locates the left end of the handle inside the stack and replaces the handle by the LHS of an appropriate production
  3. **accept**: announces successful completion of parsing
  4. **error**: syntax error, error recovery routine is called

S-R Parsing Example 1

$ marks the bottom of stack and the right end of the input

Stack	**Input**	**Action**
$	*acbbac*$	shift
$ *a*	*cbbac*$	shift
$ *ac*	*bbac*$	reduce by $S \rightarrow c$
$ *aS*	*bbac*$	shift
$ *aSb*	*bac*$	shift
$ *aSbb*	*ac*$	shift
$ *aSbba*	*c*$	reduce by $A \rightarrow ba$
$ *aSbA*	*c*$	reduce by $B \rightarrow bA$
$ *aSB*	*c*$	reduce by $A \rightarrow SB$
$ *aA*	*c*$	shift
$ *aAc*	$	reduce by $S \rightarrow c$
$ *aAS*	$	reduce by $S \rightarrow aAS$
$ *S*	$	accept

S-R Parsing Example 2

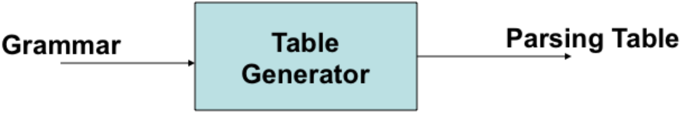
$\$$ marks the bottom of stack and the right end of the input

Stack	**Input**	**Action**
$\$$	$id_1 + id_2 * id_3 \$$	shift
$\$ \; id_1$	$+id_2 * id_3 \$$	reduce by $E \rightarrow id$
$\$ \; E$	$+id_2 * id_3 \$$	shift
$\$ \; E+$	$id_2 * id_3 \$$	shift
$\$ \; E + id_2$	$*id_3 \$$	reduce by $E \rightarrow id$
$\$ \; E + E$	$*id_3 \$$	shift
$\$ \; E + E*$	$id_3 \$$	shift
$\$ \; E + E * id_3$	$\$$	reduce by $E \rightarrow id$
$\$ \; E + E * E$	$\$$	reduce by $E \rightarrow E * E$
$\$ \; E + E$	$\$$	reduce by $E \rightarrow E + E$
$\$ \; E$	$\$$	accept

LR Parsing

- LR(k) - *L*eft to right scanning with *R*ightmost derivation in reverse, *k* being the number of lookahead tokens
    - $k = 0, 1$ are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written quite easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracking shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR grammars - an LL(k) grammar is also LR(k), but not vice-versa

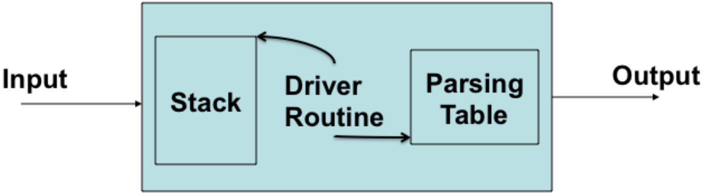
LR Parser Generation



Grammar

Table Generator

Parsing Table



Input

Stack

Driver Routine

Parsing Table

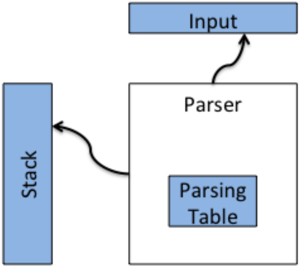
Output

LR Parser Generator

LR Parser Configuration

- A configuration of an LR parser is:
  $(\underbrace{s_0 X_1 s_2 X_2 ... X_m s_m}_{\textbf{stack}}, \quad \underbrace{a_i a_{i+1} ... a_n \ \$}_{\textbf{unexpended input}})$, where,
  $s_0, s_1, ..., s_m$, are the states of the parser, and $X_1, X_2, ..., X_m$, are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 ... a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 ... a_n$ is the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entries: **shift, reduce, accept**, or **error**
  - The *GOTO* table provides the next state information to be used after a *reduce* move

LR Parsing Algorithm



```
Initial configuration: Stack = state 0, Input = w$,
a = first input symbol;
repeat {
   let s be the top stack state;
   let a be the next input symbol;
   if ( ACTION[s, a] == shift p) {
         push a and p onto the stack (in that order);
         advance input pointer;
   } else if (ACTION[s,a] == reduce $A \rightarrow \alpha$) then {
                pop $2*|\alpha|$ symbols off the stack;
                let s' be the top of stack state now;
                push A and GOTO[s', A] onto the stack
                (in that order);
         } else if (ACTION[s, a] == accept) break;
                /* parsng is over */
                else error();
} until true; /* for ever */
```

LR Parsing Example 1 - Parsing Table

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S’ → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

Syntax Analysis:
Context-free Grammars, Pushdown Automata and Parsing
Part - 5

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

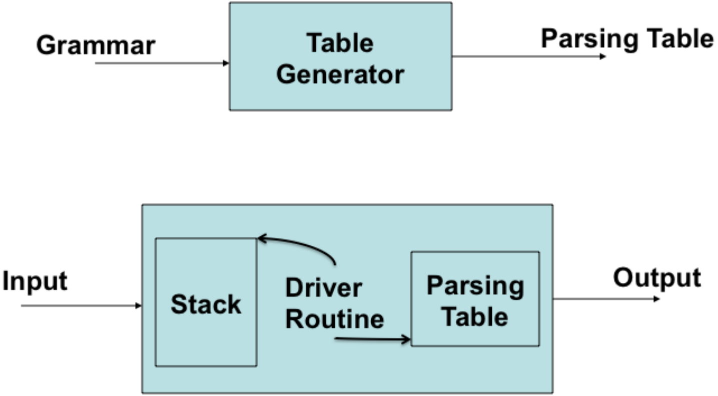
Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing
  (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing

LR Parsing

- LR(k) - *L*eft to right scanning with *R*ightmost derivation in reverse, *k* being the number of lookahead tokens
    - $k = 0, 1$ are of practical interest
- LR parsers are also automatically generated using parser generators
- LR grammars are a subset of CFGs for which LR parsers can be constructed
- LR(1) grammars can be written quite easily for practically all programming language constructs for which CFGs can be written
- LR parsing is the most general non-backtracking shift-reduce parsing method (known today)
- LL grammars are a strict subset of LR grammars - an LL(k) grammar is also LR(k), but not vice-versa

LR Parser Generation

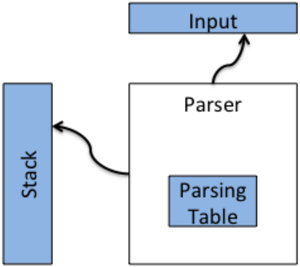


LR Parser Generator

LR Parser Configuration

- A configuration of an LR parser is:
  $(s_0 X_1 s_2 X_2 ... X_m s_m, \quad a_i a_{i+1} ... a_n \, \$)$, where,
  stack **unexpended input**
  $s_0, s_1, ..., s_m$, are the states of the parser, and $X_1, X_2, ..., X_m$, are grammar symbols (terminals or nonterminals)
- Starting configuration of the parser: $(s_0, a_1 a_2 ... a_n \$)$, where, s_0 is the initial state of the parser, and $a_1 a_2 ... a_n$ is the string to be parsed
- Two parts in the parsing table: *ACTION* and *GOTO*
  - The *ACTION* table can have four types of entries: **shift, reduce, accept**, or **error**
  - The *GOTO* table provides the next state information to be used after a *reduce* move

LR Parsing Algorithm



Input

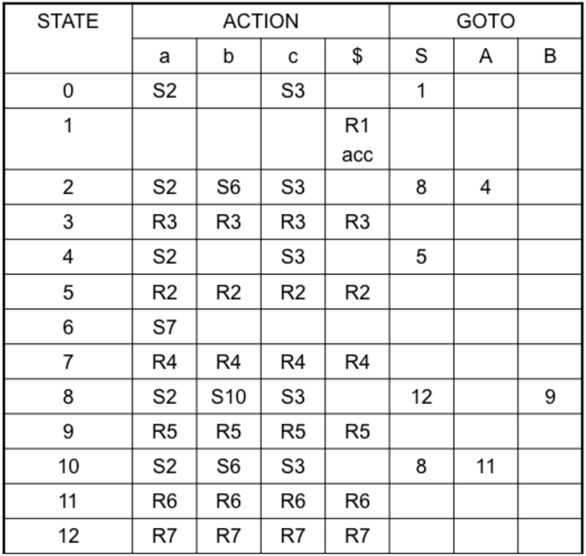
Parser

Parsing Table

Stack

```
Initial configuration: Stack = state 0, Input = w$,
a = first input symbol;
repeat {
   let s be the top stack state;
   let a be the next input symbol;
   if ( ACTION[s, a] == shift p) {
         push a and p onto the stack (in that order);
         advance input pointer;
   } else if (ACTION[s,a] == reduce $A \rightarrow \alpha$) then {
                pop $2*|\alpha|$ symbols off the stack;
                let s' be the top of stack state now;
                push A and GOTO[s', A] onto the stack
                (in that order);
            } else if (ACTION[s, a] == accept) break;
                    /* parsng is over */
                    else error();
} until true; /* for ever */
```

LR Parsing Example 1 - Parsing Table



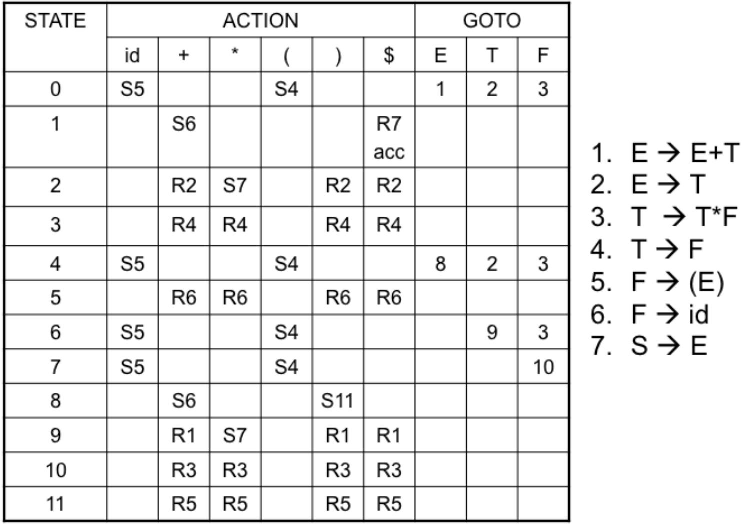
STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S’ → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

LR Parsing Example 1 (contd.)

Stack	**Input**	**Action**
0	$acbbac\$$	S2
$0a2$	$cbbac\$$	S3
$0a2c3$	$bbac\$$	R3 ($S \rightarrow c$, goto(2,S) = 8)
$0a2S8$	$bbac\$$	S10
$0a2S8b10$	$bac\$$	S6
$0a2S8b10b6$	$ac\$$	S7
$0a2S8b10b6a7$	$c\$$	R4 ($A \rightarrow ba$, goto(10,A) = 11)
$0a2S8b10A11$	$c\$$	R6 ($B \rightarrow bA$, goto(8,B) = 9)
$0a2S8B9$	$c\$$	R5 ($A \rightarrow SB$, goto(2,A) = 4)
$0a2A4$	$c\$$	S3
$0a2A4c3$	$\$$	R3 ($S \rightarrow c$, goto(4,S) = 5)
$0a2A4S5$	$\$$	R2 ($S \rightarrow aAS$, goto(0,S) = 1)
$0S1$	$\$$	R1 ($S' \rightarrow S$), and accept

LR Parsing Example 2 - Parsing Table



STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				R7 acc			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. E → E+T
2. E → T
3. T → T*F
4. T → F
5. F → (E)
6. F → id
7. S → E

LR Parsing Example 2(contd.)

Stack	**Input**	**Action**
0	$id + id * id\$$	S5
$0\ id\ 5$	$+id * id\$$	R6 ($F \rightarrow id$, G(0,F) = 3)
$0\ F\ 3$	$+id * id\$$	R4 ($T \rightarrow F$, G(0,T) = 2)
$0\ T\ 2$	$+id * id\$$	R2 ($E \rightarrow T$, G(0,E) = 1)
$0\ E\ 1$	$+id * id\$$	S6
$0\ E\ 1 + 6$	$id * id\$$	S5
$0\ E\ 1 + 6\ id\ 5$	$*id\$$	R6 ($F \rightarrow id$, G(6,F) = 3)
$0\ E\ 1 + 6F3$	$*id\$$	R4 ($T \rightarrow F$, G(6,T) = 9)
$0\ E\ 1 + 6T9$	$*id\$$	S7
$0\ E\ 1 + 6T9 * 7$	$id\$$	S5
$0\ E\ 1 + 6T9 * 7\ id\ 5$	$\$$	R6 ($F \rightarrow id$, G(7,F) = 10)
$0\ E\ 1 + 6T9 * 7F10$	$\$$	R3 ($T \rightarrow T * F$, G(6,T) = 9)
$0\ E\ 1 + 6T9$	$\$$	R1 ($E \rightarrow E + T$, G(0,E) = 1)
$0\ E\ 1$	$\$$	R7 ($S \rightarrow E$) and accept

LR Grammars

- Consider a rightmost derivation:
  $S \Rightarrow^*_{rm} \phi Bt \Rightarrow_{rm} \phi\beta t$,
  where the production $B \rightarrow \beta$ has been applied
- A grammar is said to be **LR(k)**, if for any given input string, at each step of any rightmost derivation, the handle β can be detected by examining the string $\phi\beta$ and scanning *at most*, first *k* symbols of the unused input string *t*

LR Grammars (contd.)

- Example: The grammar, $\{S \rightarrow E,\ E \rightarrow E + E \mid E * E \mid id\}$, is not LR(2)
  - $S \Rightarrow^{1} \underline{E} \Rightarrow^{2} \underline{E + E} \Rightarrow^{3} E + \underline{E * E} \Rightarrow^{4} E + E * \underline{id} \Rightarrow^{5}$ $E + \underline{id} * id \Rightarrow^{6} \underline{id} + id * id$
  - $S \Rightarrow^{1'} \underline{E} \Rightarrow^{2'} \underline{E * E} \Rightarrow^{3'} E * \underline{id} \Rightarrow^{4'} \underline{E + E} * id \Rightarrow^{5'}$ $E + \underline{id} * id \Rightarrow^{6'} \underline{id} + id * id$
  - In the above two derivations, the handle at steps 6 & 6' and at steps 5 & 5', is $E \rightarrow id$, and the position is underlined (with the same lookahead of two symbols, $id+$ and $+id$)
  - However, the handles at step 4 and at step 4' are different ($E \rightarrow id$ and $E \rightarrow E + E$), even though the lookahead of 2 symbols is the same ($*id$), and the stack is also the same ($\phi = E + E$)
  - That means that the handle cannot be determined using the lookahead

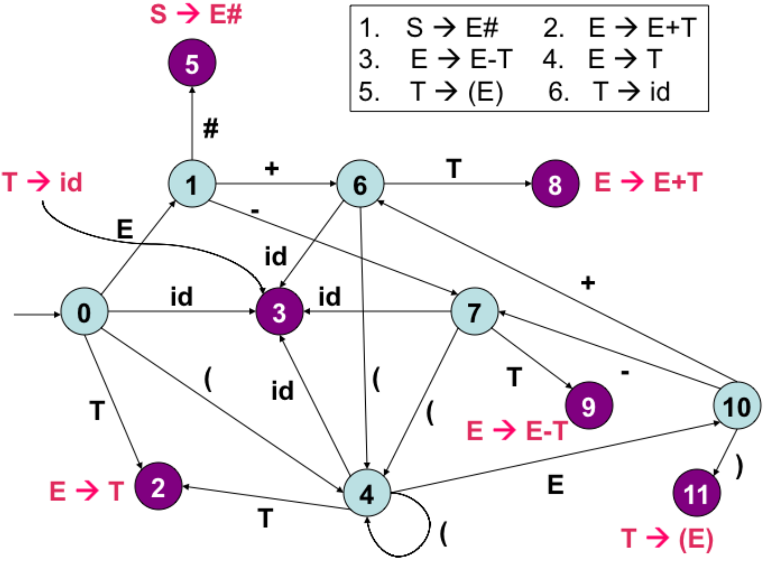
LR Grammars (contd.)

- A **viable prefix** of a sentential form $\phi\beta t$, where β denotes the handle, is any prefix of $\phi\beta$. A viable prefix cannot contain symbols to the right of the handle
- Example: $S \rightarrow E\#$, $E \rightarrow E + T \mid E - T \mid T$, $T \rightarrow id \mid (E)$
  $S \Rightarrow E\# \Rightarrow E + T\# \Rightarrow E + (E)\# \Rightarrow E + (T)\#$
  $\Rightarrow E + (id)\#$
  $E,\ E+,\ E + ($, and $E + (id$, are all viable prefixes of the right sentential form $E + (id)\#$
- It is always possible to add appropriate terminal symbols to the end of a viable prefix to get a right-sentential form
- Viable prefixes characterize the prefixes of sentential forms that can occur on the stack of an LR parser

LR Grammars (contd.)

- **Theorem**: The set of all viable prefixes of all the right sentential forms of a grammar is a regular language
- The DFA of this regular language can detect handles during LR parsing
- When this DFA reaches a “reduction state”, the corresponding viable prefix cannot grow further and thus signals a reduction
- This DFA can be constructed by the compiler using the grammar
- All LR parsers have such a DFA incorporated in them
- We construct an augmented grammar for which we construct the DFA
    - If S is the start symbol of G, then G' contains all productions of G and also a new production $S' \rightarrow S$
    - This enables the parser to halt as soon as S' appears on the stack

DFA for Viable Prefixes - LR(0) Automaton



1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

Items and *Valid* Items

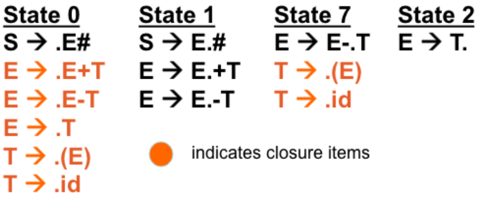
- A finite set of *items* is associated with each state of DFA
  - An *item* is a marked production of the form $[A \rightarrow \alpha_1.\alpha_2]$, where $A \rightarrow \alpha_1\alpha_2$ is a production and '.' denotes the mark
  - Many items may be associated with a production *e.g.,* the items $[E \rightarrow .E + T],\ [E \rightarrow E. + T],\ [E \rightarrow E + .T]$, and $[E \rightarrow E + T.]$ are associated with the production $E \rightarrow E + T$
- An item $[A \rightarrow \alpha_1.\alpha_2]$ is *valid* for some viable prefix $\phi\alpha_1$, iff, there exists some rightmost derivation $S \Rightarrow^* \phi A t \Rightarrow \phi\alpha_1\alpha_2 t$, where $t \in \Sigma^*$
- There may be several items valid for a viable prefix
  - The items $[E \rightarrow E - .T],\ [T \rightarrow .id]$, and $[T \rightarrow .(E)]$ are all valid for the viable prefix "$E-$" as shown below
    $S \Rightarrow E\# \Rightarrow E - T\#$, $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - id\#$,
    $S \Rightarrow E\# \Rightarrow E - T\# \Rightarrow E - (E)\#$

Valid Items and States of LR(0) DFA

- An item indicates how much of a production has already been seen and how much remains to be seen
    - $[E \rightarrow E - .T]$ indicates that we have already seen a string derivable from "$E-$" and that we hope to see next, a string derivable from T
- Each state of an LR(0) DFA contains only those items that are valid for the *same set of viable prefixes*
    - All items in state 7 are valid for the viable prefixes "$E-$" and "$(E-$" (and many more)
    - All items in state 4 are valid for the viable prefix "(" (and many more)
    - In fact, the set of all viable prefixes for which the items in a state *s* are valid is the set of strings that can take us from state 0 (initial) to state *s*
- Constructing the LR(0) DFA using sets of items is very simple

Closure of a Set of Items

```
Itemset closure(I){ /* I is a set of items */
    while (more items can be added to I) {
        for each item [$A \to \alpha.B\beta$] $\in$ I {
            /* note that B is a nonterminal and is right after the "." */
            for each production $B \to \gamma \in G$
                if (item [$B \to .\gamma$] $\notin$ I) add item [$B \to .\gamma$] to I
    }
    return I
}
```

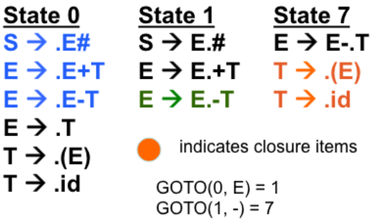


State 0	State 1	State 7	State 2
S → .E#	S → E.#	E → E-.T	E → T.
E → .E+T	E → E.+T	T → .(E)	
E → .E-T	E → E.-T	T → .id	
E → .T			
T → .(E)			
T → .id			

● indicates closure items

GOTO set computation

```
Itemset GOTO(I, X){ /* I is a set of items
    X is a grammar symbol, a terminal or a nonterminal */
    Let $I' = \{[A \rightarrow \alpha X.\beta] \mid [A \rightarrow \alpha.X\beta] \in I\}$;
    return (closure(I'))
}
```



State 0	**State 1**	**State 7**
S → .E#	**S → E.#**	**E → E-.T**
E → .E+T	**E → E.+T**	**T → .(E)**
E → .E-T	**E → E.-T**	**T → .id**
E → .T		
T → .(E)		
T → .id		

● indicates closure items

GOTO(0, E) = 1
GOTO(1, -) = 7

Intuition behind *closure* and *GOTO*

- If an item $[A \to \alpha.B\delta]$ is in a state (i.e., item set I), then, some time in the future, we expect to see in the input, a string derivable from $B\delta$
  - This implies a string derivable from B as well
  - Therefore, we add an item $[B \to .\beta]$ corresponding to each production $B \to \beta$ of B, to the state (i.e., item set I)
- If I is the set of items valid for a viable prefix γ
  - All the items in *closure*(I) are also valid for γ
  - *GOTO*(I, X) is the set items valid for the viable prefix γX
    - If $[A \to \alpha.B\delta]$ (in item set I) is valid for the viable prefix $\phi\alpha$, and $B \to \beta$ is a production, we have $S \Rightarrow^* \phi At \Rightarrow \phi\alpha B\delta t \Rightarrow^* \phi\alpha Bxt \Rightarrow \phi\alpha\beta xt$ demonstrating that the item $[B \to .\beta]$ (in the closure of I) is valid for $\phi\alpha$
    - The above derivation also shows that the item $[A \to \alpha B.\delta]$ (in GOTO(I, B) is valid for the viable prefix $\phi\alpha B$

Construction of Sets of Canonical LR(0) Items

```
void Set_of_item_sets($G'$){ /* G' is the augmented grammar */
  $C$ = {closure($\{S' \rightarrow .S\}$)};/* $C$ is a set of item sets */
  while (more item sets can be added to $C$) {
    for each item set $I \in C$ and each grammar symbol $X$
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}
```

- Each set in *C* (above) corresponds to a state of a DFA (LR(0) DFA)
- This is the DFA that recognizes viable prefixes

Construction of an LR(0) Automaton - Example 1

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

State 11
T → (E).

● indicates closure items

● indicates kernel items

Shift and Reduce Actions

- If a state contains an item of the form $[A \rightarrow \alpha.]$ ("reduce item"), then a reduction by the production $A \rightarrow \alpha$ is the action in that state
- If there are no "reduce items" in a state, then shift is the appropriate action
- There could be shift-reduce conflicts or reduce-reduce conflicts in a state
  - Both shift and reduce items are present in the same state (S-R conflict), or
  - More than one reduce item is present in a state (R-R conflict)
  - It is normal to have more than one shift item in a state (no shift-shift conflicts are possible)
- If there are no S-R or R-R conflicts in any state of an LR(0) DFA, then the grammar is LR(0), otherwise, it is not LR(0)

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 6

Y.N. Srikant

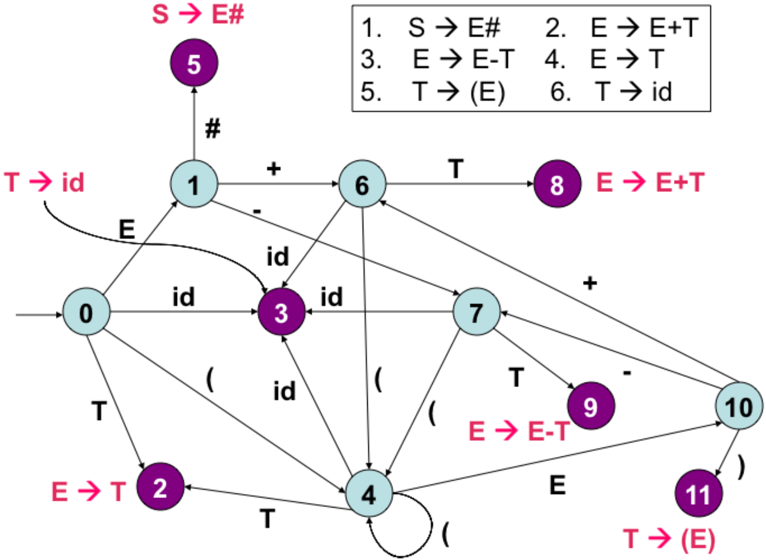
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing
  (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing (continued)

DFA for Viable Prefixes - LR(0) Automaton



1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

Construction of Sets of Canonical LR(0) Items

```
void Set_of_item_sets(G'){ /* G' is the augmented grammar */
    $C = \{closure(\{S' \rightarrow .S\})\}$;/* C is a set of item sets */
    while (more item sets can be added to C) {
        for each item set $I \in C$ and each grammar symbol X
        /* X is a grammar symbol, a terminal or a nonterminal */
            if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
                $C = C \cup GOTO(I, X)$
    }
}
```

- Each set in *C* (above) corresponds to a state of a DFA (LR(0) DFA)
- This is the DFA that recognizes viable prefixes

Construction of an LR(0) Automaton - Example 1

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 6
E → E+.T
T → .(E)
T → .id

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

State 11
T → (E).

🟠 indicates closure items

● indicates kernel items

Shift and Reduce Actions

- If a state contains an item of the form $[A \rightarrow \alpha.]$ (“reduce item”), then a reduction by the production $A \rightarrow \alpha$ is the action in that state
- If there are no “reduce items” in a state, then shift is the appropriate action
- There could be shift-reduce conflicts or reduce-reduce conflicts in a state
  - Both shift and reduce items are present in the same state (S-R conflict), or
  - More than one reduce item is present in a state (R-R conflict)
  - It is normal to have more than one shift item in a state (no shift-shift conflicts are possible)
- If there are no S-R or R-R conflicts in any state of an LR(0) DFA, then the grammar is LR(0), otherwise, it is not LR(0)

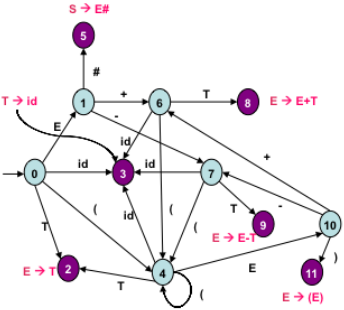
LR(0) Parser Table - Example 1

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0			S4		S3			1	2
1	S6	S7				S5			
2	R4	R4	R4	R4	R4	R4			
3	R6	R6	R6	R6	R6	R6			
4			S4		S3			10	2
5	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc			
6			S4		S3				8
7			S4		S3				9
8	R2	R2	R2	R2	R2	R2			
9	R3	R3	R3	R3	R3	R3			
10	S6	S7		S11					
11	R5	R5	R5	R5	R5	R5			

1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

Construction of an LR(0) Parser Table - Example 1

STATE	ACTION						GOTO		
	+	-	(	)	id	#	S	E	T
0			S4		S3			1	2
1	S6	S7				S5			
2	R4	R4	R4	R4	R4	R4			
3	R6	R6	R6	R6	R6	R6			
4			S4		S3			10	2
5	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc	R1 acc			
6			S4		S3				8
7			S4		S3				9
8	R2	R2	R2	R2	R2	R2			
9	R3	R3	R3	R3	R3	R3			
10	S6	S7		S11					
11	R5	R5	R5	R5	R5	R5			



1. S → E#
2. E → E+T
3. E → E-T
4. E → T
5. T → (E)
6. T → id

State 0
S → .E#
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.#
E → E.+T
E → E.-T

State 2
E → T.

State 3
T → id.

State 6
E → E+.T
T → .(E)
T → .id

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
S → E#.

State 7
E → E-.T
T → .(E)
T → .id

State 8
E → E+T.

State 9
E → E-T.

State 10
T → (E.)
E → E.+T
E → E.-T

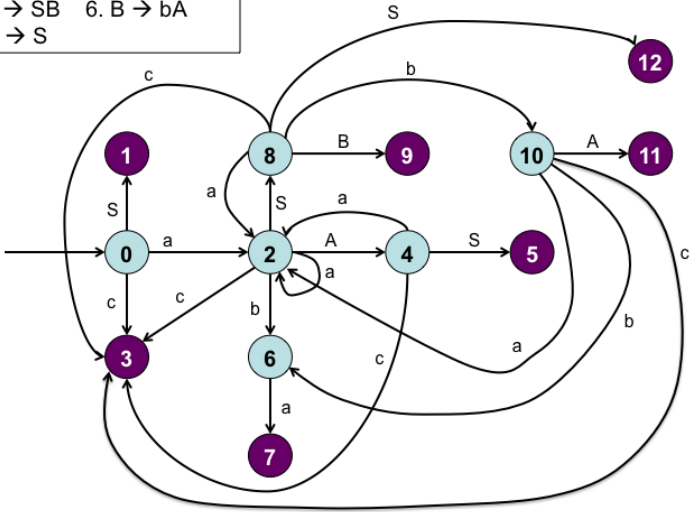
State 11
T → (E).

🟠 indicates closure items

● indicates kernel items

LR(0) Automaton - Example 2

1. S' → S   2. S → aAS
3. S → c   4. A → ba
5. A → SB   6. B → bA
7. B → S



Construction of an LR(0) Automaton - Example 2

State 0
S’ → .S
S → .aAS
S → .c

State 1
S’ → S.

State 2
S → a.AS
A → .ba
A → .SB
S → .aAS
S → .c

State 3
S → c.

State 4
S → aA.S
S → .aAS
S → .c

State 5
S → aAS.

State 6
A → b.a

State 7
A → ba.

State 8
A → S.B
B → .bA
B → .S
S → .aAS
S → .c

State 9
A → SB.

State 10
B → b.A
A → .ba
A → .SB
S → .aAS
S →.c

State 11
B → bA.

State 12
B → S.

🟠 indicates closure items

● indicates kernel items

1. S’ → S 2. S → aAS 3. S → c
4. A → ba 5. A → SB
6. B → bA 7. B → S

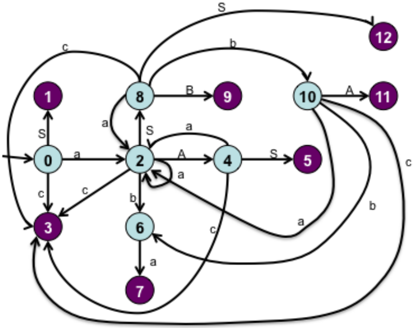
LR(0) Parser Table - Example 2

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			

1. S’ → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

Construction of an LR(0) Parser Table - Example 2

STATE	ACTION				GOTO		
	a	b	c	$	S	A	B
0	S2		S3		1		
1				R1 acc			
2	S2	S6	S3		8	4	
3	R3	R3	R3	R3			
4	S2		S3		5		
5	R2	R2	R2	R2			
6	S7						
7	R4	R4	R4	R4			
8	S2	S10	S3		12		9
9	R5	R5	R5	R5			
10	S2	S6	S3		8	11	
11	R6	R6	R6	R6			
12	R7	R7	R7	R7			



1. S' → S
2. S → aAS
3. S → c
4. A → ba
5. A → SB
6. B → bA
7. B → S

State 0
S' → .S
S → .aAS
S → .c

State 1
S' → S.

State 2
S → a.AS
A → .ba
A → .SB
S → .aAS
S → .c

State 3
S → c.

State 4
S → aA.S
S → .aAS
S → .c

State 5
S → aAS.

State 6
A → b.a

State 7
A → ba.

State 9
A → SB.

State 8
A → S.B
B → .bA
B → .S
S → .aAS
S → .c

State 11
B → bA.

State 10
B → b.A
A → .ba
A → .SB
S → .aAS
S → .c

State 12
B → S.

● indicates closure items ● indicates kernel items

A Grammar that is not LR(0) - Example 1

State 0
S → .E
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 1
S → E.
E → E.+T
E → E.-T

shift-reduce conflicts in state 1

State 2
E → T.

State 3
T → id.

State 4
T → (.E)
E → .E+T
E → .E-T
E → .T
T → .(E)
T → .id

State 5
E → E+.T
T → .(E)
T → .id

State 6
E → E-.T
T → .(E)
T → .id

State 7
E → E+T.

State 8
E → E-T.

State 9
T → (E.)
E → E.+T
E → E.-T

State 10
T → (E).

● indicates closure items

● indicates kernel items

follow(S) = {$}, where $ is EOF
Reduction on $, and shifts on + and - , will resolve the conflicts
This is similar to having an end marker such as #

Grammar is not LR(0), but is SLR(1)

SLR(1) Parsers

- If the grammar is not LR(0), we try to resolve conflicts in the states using one look-ahead symbol
- Example: The expression grammar that is not LR(0)
  The state containing the items $[T \rightarrow F.]$ and $[T \rightarrow F. * T]$ has S-R conflicts
  - Consider the reduce item $[T \rightarrow F.]$ and the symbols in $FOLLOW(T)$
  - $FOLLOW(T) = \{+,), \$\}$, and reduction by $T \rightarrow F$ can be performed on seeing one of these symbols in the input (look-ahead), since shift requires seeing $*$ in the input
  - Recall from the definition of $FOLLOW(T)$ that symbols in $FOLLOW(T)$ are the only symbols that can legally follow T in any sentential form, and hence reduction by $T \rightarrow F$ when one of these symbols is seen, is correct
  - If the S-R conflicts can be resolved using the FOLLOW set, the grammar is said to be SLR(1)

A Grammar that is not LR(0) - Example 2

State 0
S → .E
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 1
S → E.
E → E.+T
Shift-reduce conflict

State 2
E → T.

State 3
T → F.*T
T → F.
Shift-reduce conflict

State 4
F → (.E)
E → .E+T
E → .T
T → .F*T
T → .F
F → .(E)
F → .id

State 5
F → id.

State 6
E → E+.T
T → .F*T
T → .F
F → .(E)
F → .id

State 7
T → F*.T
T → .F*T
T → .F
F → .(E)
F → .id

State 8
F → (E.)
E → E.+T

State 9
E → E+T.

State 10
E → F*T.

State 11
F → (E).

follow(S) = {$}, Reduction on $ and shift on +, eliminates conflicts
follow(T) = {$,), +}, where $ is EOF
Reduction on $,), and +, and shift on *, eliminates conflicts

Grammar is not LR(0), but is SLR(1)

Construction of an SLR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(0) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n
Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S]$)
Parsing actions for state *i* are determined as follows

1. If $([A \rightarrow \alpha.a\beta] \in I_i)$ && $([A \rightarrow \alpha a.\beta] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$, for all $a \in follow(A)$
3. If $([S' \rightarrow S.] \in I_i)$ set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not SLR(1)

4. If $([A \rightarrow \alpha.A\beta] \in I_i)$ && $([A \rightarrow \alpha A.\beta] \in I_j)$
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

A Grammar that is not LR(0) - Example 3

Grammar
S' → S, S → aSb, S → ε

follow(S) = {$, b}

State 0
S' → .S
S → .aSb
S → .

State 3
S → aS.b

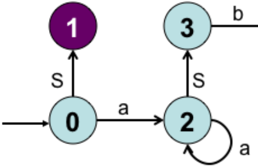
State 1
S' → S.

State 4
S → aSb.

State 2
S → a.Sb
S → .aSb
S → .

shift-reduce conflict in states 0, 2

	a	b	$	S
0	S2	reduce S → ε	reduce S → ε	1
1			accept	
2	S2	reduce S → ε	reduce S → ε	3
3		S4		
4		reduce S → aSb	reduce S → aSb	



Grammar is not LR(0), but is SLR(1)

● indicates closure items

● indicates kernel items

A Grammar that is not SLR(1) - Example 1

Grammar: S' → S,
S → aSb, S → ab, S → ε

State 0
S' → .S
S → .aSb
S → .ab
S → .

State 1
S' → S.

State 2
S → a.Sb
S → a.b
S → .aSb
S → .ab
S → .

State 3
S → aS.b

State 4
S → aSb.

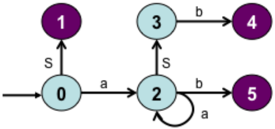
State 5
S → ab.

shift-reduce conflict in states 0, 2

Grammar is neither LR(0) nor SLR(1)

follow(S) = {$, b}
State 0: Reduction on *$* and *b*, by S → ε, and shift on *a* resolves conflicts
State 2: S-R conflict on *b* still remains

	a	b	$	S
0	S2	R: S → ε	R: S → ε	1
1			accept	
2	S2	S5, R: S → ε	R: S → ε	3
3		S4		
4		R: S → aSb	R: S → aSb	
5		R: S → ab	R: S → ab	



A Grammar that is not SLR(1) - Example 2

Grammar
S’ → S
S → L=R
S → R
L → *R
L → id
R → L

Grammar is neither LR(0) nor SLR(1)

State 0
S’ → .S
S → .L=R
S → .R
L → .*R
L → .id
R → .L

State 1
S’ → S.

State 3
S → R.

State 2
S → L .=R
R → L .
shift-reduce conflict

State 4
L → *.R
R → .L
L → .*R
L → .id

State 5
L → id.

State 6
S → L=.R
R → .L
L → .*R
L → .id

State 7
L → *R.

State 8
R → L .

State 9
S → L=R.

Follow(R) = {$,=} does not resolve S-R conflict

The Problem with SLR(1) Parsers

- SLR(1) parser construction process does not remember enough left context to resolve conflicts
  - In the "$L = R$" grammar (previous slide), the symbol '=' got into follow(R) because of the following derivation: $S' \Rightarrow S \Rightarrow L = R \Rightarrow L = L \Rightarrow L = id \Rightarrow *\underline{R =} id \Rightarrow ...$
  - The production used is $L \rightarrow *R$
  - The following rightmost derivation in *reverse* does not exist (and hence reduction by $R \rightarrow L$ on '=' in state 2 is illegal) $id = id \Leftarrow L = id \Leftarrow R = id ...$
- Generalization of the above example
  - In some situations, when a state *i* appears on top of the stack, a viable prefix $\beta\alpha$ may be on the stack such that βA cannot be followed by '*a*' in any right sentential form
  - Thus, the reduction by $A \rightarrow \alpha$ would be invalid on '*a*'
  - In the above example, β = ϵ, α = L, and $A = R$; L cannot be reduced to R on '=', since it would lead to the above illegal derivation sequence

LR(1) Parsers

- LR(1) items are of the form $[A \rightarrow \alpha.\beta,\ a]$, a being the "lookahead" symbol
- Lookahead symbols have no part to play in shift items, but in reduce items of the form $[A \rightarrow \alpha.,\ a]$, reduction by $A \rightarrow \alpha$ is valid only if the next input symbol is 'a'
- An LR(1) item $[A \rightarrow \alpha.\beta,\ a]$ is *valid* for a viable prefix γ, if there is a derivation $S \Rightarrow^*_{rm} \delta A w \Rightarrow_{rm} \delta\alpha\beta w$, where, $\gamma = \delta\alpha$, $a = first(w)$ or $w = \epsilon$ and $a = \$$
- Consider the grammar: $S' \rightarrow S,\ S \rightarrow aSb \mid \epsilon$
  - $[S \rightarrow a.Sb,\ \$]$ is valid for the VP a, $S' \Rightarrow S \Rightarrow aSb$
  - $[S \rightarrow a.Sb,\ b]$ is valid for the VP aa, $S' \Rightarrow S \Rightarrow aSb \Rightarrow aaSbb$
  - $[S \rightarrow .,\ \$]$ is valid for the VP ϵ, $S' \Rightarrow S \Rightarrow \epsilon$
  - $[S \rightarrow aSb.,\ b]$ is valid for the VP $aaSb$, $S' \Rightarrow S \Rightarrow aSb \Rightarrow aaSbb$

LR(1) Grammar - Example 1

Grammar
S’ → S, S → aSb, S → ε

State 0
S’ → .S , $
S → .aSb , $
S → . , $

State 1
S’ → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

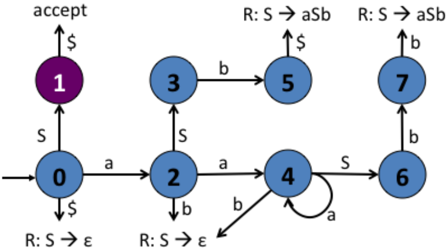
State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

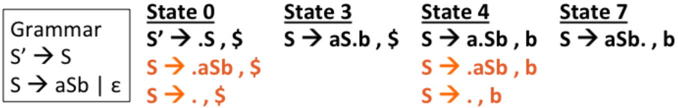
Grammar is LR(1)

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		



Closure of a Set of LR(1) Items

```
Itemset closure(I){ /* I is a set of LR(1) items */
    while (more items can be added to I) {
        for each item [$A \rightarrow \alpha.B\beta,\ a$] $\in I$ {
            for each production $B \rightarrow \gamma \in G$
                for each symbol $b \in first(\beta a)$
                    if (item [$B \rightarrow .\gamma,\ b$] $\notin I$) add item [$B \rightarrow .\gamma,\ b$] to $I$
    }
    return I
}
```

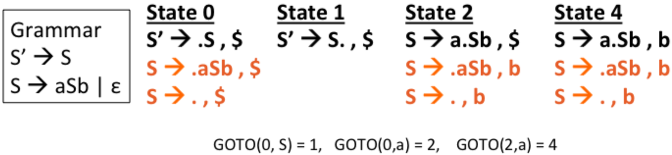


Grammar
S' → S
S → aSb | ε

State 0	**State 3**	**State 4**	**State 7**
S' → .S , $	**S → aS.b , $**	**S → a.Sb , b**	**S → aSb. , b**
S → .aSb , $		**S → .aSb , b**	
S → . , $		**S → . , b**	

GOTO set computation

```
$Itemset\ GOTO(I,\ X)$\{ /* I is a set of LR(1) items
    X is a grammar symbol, a terminal or a nonterminal */
    Let $I' = \{[A \rightarrow \alpha X.\beta,\ a] \mid [A \rightarrow \alpha.X\beta,\ a] \in I\}$;
    return ($closure(I')$)
}
```



Grammar
S’ → S
S → aSb | ε

State 0
S’ → .S , $
S → .aSb , $
S → . , $

State 1
S’ → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

GOTO(0, S) = 1,   GOTO(0,a) = 2,   GOTO(2,a) = 4

Construction of Sets of Canonical of LR(1) Items

```
void Set_of_item_sets(G'){ /* G' is the augmented grammar */
    C = {closure({$S' \rightarrow .S,\ \$$})};/* C is a set of LR(1) item sets */
    while (more item sets can be added to C) {
        for each item set $I \in C$ and each grammar symbol X
        /* X is a grammar symbol, a terminal or a nonterminal */
            if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
                $C = C \cup GOTO(I, X)$
    }
}
```

- Each set in *C* (above) corresponds to a state of a DFA (LR(1) DFA)
- This is the DFA that recognizes viable prefixes

LR(1) DFA Construction - Example 1

Grammar
S' → S, S → aSb, S → ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

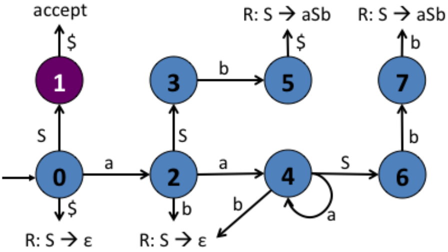
State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

Grammar is LR(1)

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		



Construction of an LR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(1) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n
Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S,\ \$]$)
Parsing actions for state *i* are determined as follows
1. If $([A \rightarrow \alpha.a\beta,\ b] \in I_i)$ && $([A \rightarrow \alpha a.\beta,\ b] \in I_j)$
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If $([A \rightarrow \alpha.,\ a] \in I_i)$
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$
3. If $([S' \rightarrow S.,\ \$] \in I_i)$ set ACTION[i, \$] = *accept*

S-R or R-R conflicts in the table imply grammar is not LR(1)

4. If $([A \rightarrow \alpha.A\beta,\ a] \in I_i)$ && $([A \rightarrow \alpha A.\beta,\ a] \in I_j)$
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

Syntax Analysis:

Context-free Grammars, Pushdown Automata and Parsing
Part - 7

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

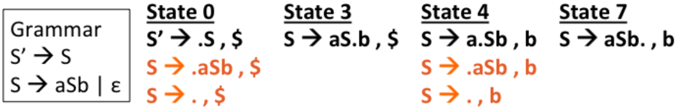
NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is syntax analysis? (covered in lecture 1)
- Specification of programming languages: context-free grammars (covered in lecture 1)
- Parsing context-free languages: push-down automata (covered in lectures 1 and 2)
- Top-down parsing: LL(1) parsing (covered in lectures 2 and 3)
- Recursive-descent parsing (covered in lecture 4)
- Bottom-up parsing: LR-parsing (continued)
- YACC Parser generator

Closure of a Set of LR(1) Items

```
$Itemset\ closure(I)$\{ /* I is a set of LR(1) items */
    while (more items can be added to I) {
        for each item $[A \rightarrow \alpha.B\beta,\ a] \in I$ {
            for each production $B \rightarrow \gamma \in G$
                for each symbol $b \in first(\beta a)$
                    if (item $[B \rightarrow .\gamma,\ b] \notin I$) add item $[B \rightarrow .\gamma,\ b]$ to $I$
    }
    return $I$
}
```



Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

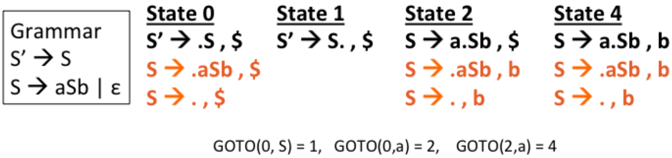
State 3
S → aS.b , $

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 7
S → aSb. , b

GOTO set computation

```
$Itemset\ GOTO(I,\ X)$\{ /* I is a set of LR(1) items
    X is a grammar symbol, a terminal or a nonterminal */
    Let $I' = \{[A \rightarrow \alpha X.\beta,\ a] \mid [A \rightarrow \alpha.X\beta,\ a] \in I\}$;
    return ($closure(I')$)
}
```



Grammar
S' → S
S → aSb | ε

State 0
S' → .S , $
S → .aSb , $
S → . , $

State 1
S' → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 4
S → a.Sb , b
S → .aSb , b
S → . , b

GOTO(0, S) = 1, GOTO(0,a) = 2, GOTO(2,a) = 4

Construction of Sets of Canonical of LR(1) Items

```
void Set_of_item_sets(G′){ /* G’ is the augmented grammar */
  C = {closure({S′ → .S, $})};/* C is a set of LR(1) item sets */
  while (more item sets can be added to C) {
    for each item set $I \in C$ and each grammar symbol X
    /* X is a grammar symbol, a terminal or a nonterminal */
      if (($GOTO(I, X) \neq \emptyset$) && ($GOTO(I, X) \notin C$))
        $C = C \cup GOTO(I, X)$
  }
}
```

- Each set in *C* (above) corresponds to a state of a DFA (LR(1) DFA)
- This is the DFA that recognizes viable prefixes

Construction of an LR(1) Parsing Table

Let $C = \{I_0, I_1, ..., I_i, ..., I_n\}$ be the canonical LR(1) collection of items, with the corresponding states of the parser being 0, 1, ... , i, ... , n

Without loss of generality, let 0 be the initial state of the parser (containing the item $[S' \rightarrow .S,\ \$]$)

Parsing actions for state *i* are determined as follows

1. If ($[A \rightarrow \alpha.a\beta,\ b] \in I_i$) && ($[A \rightarrow \alpha a.\beta,\ b] \in I_j$)
   set ACTION[i, a] = *shift j* /* *a* is a terminal symbol */
2. If ($[A \rightarrow \alpha.,\ a] \in I_i$)
   set ACTION[i, a] = *reduce* $A \rightarrow \alpha$
3. If ($[S' \rightarrow S.,\ \$] \in I_i$) set ACTION[i, $] = *accept*

S-R or R-R conflicts in the table imply grammar is not LR(1)

4. If ($[A \rightarrow \alpha.A\beta,\ a] \in I_i$) && ($[A \rightarrow \alpha A.\beta,\ a] \in I_j$)
   set GOTO[i, A] = *j* /* *A* is a nonterminal symbol */

All other entries not defined by the rules above are made *error*

LR(1) Grammar - Example 2

Grammar
S' → S
S → L=R | R
L → *R | id
R → L

State 0
S' → .S, $
S → .L=R, $
S → .R, $
L → .*R, =
L → .id, =
R → .L, $
L → .*R, $
L → .id, $

State 1
S' → S., $

State 2
S → L .=R, $
R → L ., $

State 3
S → R., $

State 4
L → *.R, =/$
R → .L, =/$
L → .*R, =/$
L → .id, =/$

State 5
L → id., =/$

State 6
S → L=.R, $
R → .L, $
L → .*R, $
L → .id, $

State 7
L → *R., =/$

State 8
R → L ., =/$

State 9
S → L=R., $

State 10
R → L ., $

State 11
L → *.R, $
R → .L, $
L → .*R, $
L → .id, $

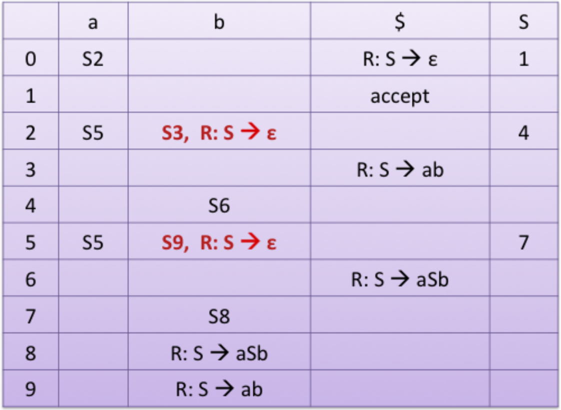
State 12
L → id., $

State 13
L → *R., $

Grammar is not SLR(1), but is LR(1)

A non-LR(1) Grammar

Grammar
S’ → S
S → aSb
S → ab
S → ε



	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S5	**S3, R: S → ε**		4
3			R: S → ab	
4		S6		
5	S5	**S9, R: S → ε**		7
6			R: S → aSb	
7		S8		
8		R: S → aSb		
9		R: S → ab		

This grammar is neither SLR(1) nor LR(1), because it is ambiguous

LALR(1) Parsers

- LR(1) parsers have a large number of states
  - For C, many thousand states
  - An SLR(1) parser (or LR(0) DFA) for C will have a few hundred states (with many conflicts)
- LALR(1) parsers have exactly the same number of states as SLR(1) parsers for the same grammar, and are derived from LR(1) parsers
  - SLR(1) parsers may have many conflicts, but LALR(1) parsers may have very few conflicts
  - If the LR(1) parser had no S-R conflicts, then the corresponding derived LALR(1) parser will also have none
  - However, this is not true regarding R-R conflicts
- LALR(1) parsers are as compact as SLR(1) parsers and are almost as powerful as LR(1) parsers
- Most programming language grammars are also LALR(1), if they are LR(1)

Construction of LALR(1) parsers

- The core part of LR(1) items (the part after leaving out the lookahead symbol) is the same for several LR(1) states (the loohahead symbols will be different)
    - Merge the states with the same core, along with the lookahead symbols, and rename them
- The ACTION and GOTO parts of the parser table will be modified
    - Merge the rows of the parser table corresponding to the merged states, replacing the old names of states by the corresponding new names for the merged states
    - For example, if states 2 and 4 are merged into a new state 24, and states 3 and 6 are merged into a new state 36, all references to states 2,4,3, and 6 will be replaced by 24,24,36, and 36, respectively
- LALR(1) parsers may perform a few more reductions (but not shifts) than an LR(1) parser before detecting an error

LALR(1) Parser Construction - Example 1

Grammar
S’ → S, S → aSb, S → ε

Grammar is LALR(1)

State 0
S’ → .S , $
S → .aSb , $
S → . , $

State 1
S’ → S. , $

State 2
S → a.Sb , $
S → .aSb , b
S → . , b

State 3
S → aS.b , $

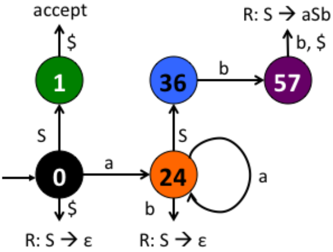
State 4
S → a.Sb , b
S → .aSb , b
S → . , b

State 5
S → aSb. , $

State 6
S → aS.b , b

State 7
S → aSb. , b

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	



LALR(1) Parser Construction - Example 1 (contd.)

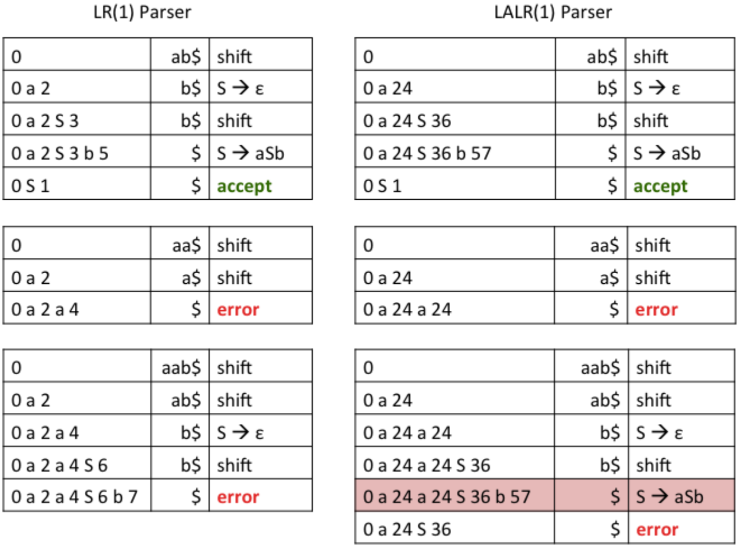
LR(1) Parser Table

	a	b	$	S
0	S2		R: S → ε	1
1			accept	
2	S4	R: S → ε		3
3		S5		
4	S4	R: S → ε		6
5			R: S → aSb	
6		S7		
7		R: S → aSb		

LALR(1) Parser Table

	a	b	$	S
0	S24		R: S → ε	1
1			accept	
24	S24	R: S → ε		36
36		S57		
57		R: S → aSb	R: S → aSb	

LALR(1) Parser Error Detection



LR(1) Parser

0	ab$	shift
0 a 2	b$	S → ε
0 a 2 S 3	b$	shift
0 a 2 S 3 b 5	$	S → aSb
0 S 1	$	accept

0	aa$	shift
0 a 2	a$	shift
0 a 2 a 4	$	error

0	aab$	shift
0 a 2	ab$	shift
0 a 2 a 4	b$	S → ε
0 a 2 a 4 S 6	b$	shift
0 a 2 a 4 S 6 b 7	$	error

LALR(1) Parser

0	ab$	shift
0 a 24	b$	S → ε
0 a 24 S 36	b$	shift
0 a 24 S 36 b 57	$	S → aSb
0 S 1	$	accept

0	aa$	shift
0 a 24	a$	shift
0 a 24 a 24	$	error

0	aab$	shift
0 a 24	ab$	shift
0 a 24 a 24	b$	S → ε
0 a 24 a 24 S 36	b$	shift
0 a 24 a 24 S 36 b 57	$	S → aSb
0 a 24 S 36	$	error

Characteristics of LALR(1) Parsers

- If an LR(1) parser has no S-R conflicts, then the corresponding derived LALR(1) parser will also have none
  - LR(1) and LALR(1) parser states have the same core items (lookaheads may not be the same)
  - If an LALR(1) parser state *s*1 has an S-R conflict, it must have two items $[A \rightarrow \alpha., a]$ and $[B \rightarrow \beta.a\gamma, b]$
  - One of the states *s*1′, from which *s*1 is generated, must have the same core items as *s*1
  - If the item $[A \rightarrow \alpha., a]$ is in *s*1′, then *s*1′ must also have the item $[B \rightarrow \beta.a\gamma, c]$ (the lookahead need not be *b* in *s*1′ - it may be *b* in some other state, but that is not of interest to us)
  - These two items in *s*1′ still create an S-R conflict in the LR(1) parser
  - Thus, merging of states with common core can never introduce a new S-R conflict, because shift depends only on core, not on lookahead

Characteristics of LALR(1) Parsers (contd.)

- However, merger of states may introduce a new R-R conflict in the LALR(1) parser even though the original LR(1) parser had none
- Such grammars are rare in practice
- Here is one from ALSU's book. Please construct the complete sets of LR(1) items as home work:
  $S' \rightarrow S\$,\ S \rightarrow aAd \mid bBd \mid aBe \mid bAe$
  $A \rightarrow c,\ B \rightarrow c$
- Two states contain the items:
  $\{[A \rightarrow c.,\ d], [B \rightarrow c.,\ e]\}$ and
  $\{[A \rightarrow c.,\ e], [B \rightarrow c.,\ d]\}$
- Merging these two states produces the LALR(1) state:
  $\{[A \rightarrow c.,\ d/e], [B \rightarrow c.,\ d/e]\}$
- This LALR(1) state has a reduce-reduce conflict

Error Recovery in LR Parsers - Parser Construction

- Compiler writer identifies *major* non-terminals such as those for *program, statement, block, expression*, etc.
- Adds to the grammar, *error productions* of the form $A \rightarrow$ *error* α, where A is a major non-terminal and α is a suitable string of grammar symbols (usually terminal symbols), possibly empty
- Associates an error message routine with each error production
- Builds an LALR(1) parser for the new grammar with error productions

Error Recovery in LR Parsers - Parser Operation

- When the parser encounters an error, it scans the stack to find the topmost state containing an *error item* of the form $A \rightarrow .error\ \alpha$
- The parser then shifts a token *error* as though it occurred in the input
- If $\alpha = \epsilon$, reduces by $A \rightarrow \epsilon$ and invokes the error message routine associated with it
- If $\alpha \neq \epsilon$, discards input symbols until it finds a symbol with which the parser can proceed
- Reduction by $A \rightarrow .error\ \alpha$ happens at the appropriate time
  Example: If the error production is $A \rightarrow .error\ ;$, then the parser skips input symbols until ';' is found, performs reduction by $A \rightarrow .error\ ;$, and proceeds as above
- Error recovery is not perfect and parser may abort on end of input

LR(1) Parser Error Recovery

State 0
S -> .rhyme, $
rhyme -> .sound place, $
rhyme -> .error DELL, $
sound -> .DING DONG, $

State 1
S -> rhyme. , $

State 2
rhyme -> sound.place, $
place -> .DELL, $
place -> .error DELL, $

State 3
rhyme -> error.DELL, $

State 4
rhyme -> error DELL. , $

State 5
sound -> DING.DONG, $

State 6
sound -> DING DONG. , $

State 7
rhyme -> sound place. , $

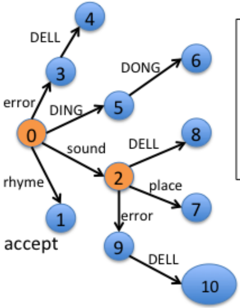
State 8
place -> DELL. , $

State 9
place -> error.DELL, $

State 10
place -> error DELL. , $

S -> rhyme
rhyme -> sound place | error DELL
sound -> DING DONG
place -> DELL | error DELL

DING DELL $
0 -> 5 -> error; pops 5;
0 contains error item;
shifts error, reads DELL, enters 4;
reduces by *rhyme -> error DELL*;
reduces by *S -> rhyme*; accepts



DING DONG DING DELL $
0 -> 5 -> 6 -> reduce -> 2 -> error;
2 contains error item;
skips DING; shifts error, reads DELL;
enters 10; reduces by *place -> error DELL*;
enters 7; reduces by *rhyme -> sound place*;
reduces by *S -> rhyme*; accepts

DING $; 0 -> 5 -> error; pops 5;
0 contains error item;
hits $; aborts; solution: add
rhyme -> error instead of
rhyme -> error DELL

YACC:

Yet Another Compiler Compiler
A Tool for generating Parsers

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

YACC Example

```
%token DING DONG DELL
%start rhyme
%%
rhyme : sound place '\n'
          {printf("string valid\n"); exit(0);};
sound : DING DONG ;
place : DELL ;
%%
#include "lex.y.c"

int yywrap(){return 1;}
yyerror( char* s)
{ printf("%s\n",s);}
main() {yyparse(); }
```

LEX Specification for the YACC Example

```
%%
ding return DING;
dong return DONG;
dell return DELL;
[ ]* ;
\n|. return yytext[0];
```

Compiling and running the parser

```
lex ding-dong.l
yacc ding-dong.y
gcc -o ding-dong.o y.tab.c
ding-dong.o
```

Sample inputs	Sample outputs
`ding dong dell`	`string valid`
`ding dell`	`syntax error`
`ding dong dell$`	`syntax error`

Form of a YACC file

- YACC has a language for describing context-free grammars
- It generates an LALR(1) parser for the CFG described
- Form of a YACC program
  %{ declarations – optional
  %}
  %%
  rules – compulsory
  %%

  programs – optional
- YACC uses the lexical analyzer generated by LEX to match the **terminal symbols** of the CFG
- YACC generates a file named **y.tab.c**

Declarations and Rules

- **Tokens**: %token name1 name2 name3, $\cdots$
- **Start Symbol**: %start name
- **names** in rules: *letter*(*letter* | *digit* | . | _)*
  letter is either a lower case or an upper case character
- **Values of symbols** and **actions**: Example

```
A   :   B
        {$$ = 1;}
        C
        {x = $2; y = $3; $$ = x+y;}
        ;
```

  - Now, value of A is stored in `$$` (second one), that of B in `$1`, that of action 1 in `$2`, and that of C in `$3`.

Declarations and Rules (contd.)

- Intermediate action in the above example is translated into an ϵ-production as follows:

```
$ACT1 : /* empty */
        {$$ =1;}
      ;
   A  : B $ACT1 C
        {x = $2; y = $3; $$ = x+y;}
      ;
```

- Intermediate actions can return values
  For example, the first `$$` in the previous example is available as `$2`
- However, intermediate actions cannot refer to values of symbols to the left of the action
- Actions are translated into C-code which are executed just before a reduction is performed by the parser

Lexical Analysis

- LA returns integers as token numbers
- Token numbers are assigned automatically by YACC, starting from 257, for all the tokens declared using **%token** declaration
- Tokens can return not only token numbers but also other information (e.g., value of a number, character string of a name, pointer to symbol table, etc.)
- Extra values are returned in the variable, **yylval**, known to YACC generated parsers

Ambiguity, Conflicts, and Disambiguation

- $E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid id$
- Ambiguity with left or right associativity of '-' and '/'
- This causes shift-reduce conflicts in YACC: (E-E-E) – shift or reduce on -?
- Disambiguating rule in YACC:
  - Default is **shift** action in S-R conflicts
  - Reduce by **earlier** rule in R-R conflicts
  - Associativity can be specified explicitely
- Similarly, precedence of operators causes S-R conflicts. Precedence can also be specified
- Example

```
%right '='
%left '+' '-'        --- same precedence for +, -
%left '*' '/'        --- same precedence for *, /
%right ^             --- highest precedence
```

Symbol Values

- Tokens and nonterminals are both stack symbols
- Stack symbols can be associated with values whose **types** are declared in a **%union** declaration in the YACC specification file
- YACC turns this into a union type called YYSTYPE
- With **%token** and **%type** declarations, we inform YACC about the types of values the tokens and nonterminals take
- Automatically, references to `$1,$2,yylval`, etc., refer to the appropriate member of the union (see example below)

YACC Example : YACC Specification (desk-3.y)

```
%{
#define NSYMS 20
struct symtab {
       char *name; double value;
     }symboltab[NSYMS];
struct symtab *symlook();
#include <string.h>
#include <ctype.h>
#include <stdio.h>
%}
```

YACC Example : YACC Specification (contd.)

```
%union {
    double dval;
    struct symtab *symp;
    }
%token <symp> NAME
%token <dval> NUMBER
%token POSTPLUS
%token POSTMINUS
%left '='
%left '+' '-'
%left '*' '/'
%left POSTPLUS
%left POSTMINUS
%right UMINUS
%type <dval> expr
```

YACC Example : YACC Specification (contd.)

```
%%
lines: lines expr ’\n’ {printf("%g\n",$2);}
     | lines ’\n’     | /* empty */
     | error ’\n’
       {yyerror("reenter last line:"); yyerrok; }
     ;
expr : NAME ’=’ expr {$1 -> value = $3; $$ = $3;}
     | NAME {$$ = $1 -> value;}
     | expr ’+’ expr {$$ = $1 + $3;}
     | expr ’-’ expr {$$ = $1 - $3;}
     | expr ’*’ expr {$$ = $1 * $3;}
     | expr ’/’ expr {$$ = $1 / $3;}
     | ’(’ expr ’)’ {$$ = $2;}
     | ’-’ expr   %prec UMINUS {$$ = - $2;}
     | expr POSTPLUS {$$ = $1 + 1;}
     | expr POSTMINUS {$$ = $1 - 1;}
     | NUMBER
```

YACC Example : LEX Specification (desk-3.l)

```
number [0-9]+\.?|[0-9]*\.[0-9]+
name [A-Za-z][A-Za-z0-9]*
%%
[ ] {/* skip blanks */}
{number} {sscanf(yytext,"%lf",&yylval.dval);
            return NUMBER;}
{name} {struct symtab *sp =symlook(yytext);
            yylval.symp = sp; return NAME;}
"++" {return POSTPLUS;}
"--" {return POSTMINUS;}
"$" {return 0;}
\n|. {return yytext[0];}
```

YACC Example : Support Routines

```
%%
void initsymtab()
{int i = 0;
 for(i=0; i<NSYMS; i++) symboltab[i].name = NULL;
}
int yywrap(){return 1;}
yyerror( char* s) { printf("%s\n",s);}
main() {initsymtab(); yyparse(); }

#include "lex.yy.c"
```

YACC Example : Support Routines (contd.)

```
struct symtab* symlook(char* s)
{struct symtab* sp = symboltab; int i = 0;
 while ((i < NSYMS) && (sp -> name != NULL))
  { if(strcmp(s,sp -> name) == 0) return sp;
    sp++; i++;
  }
 if(i == NSYMS) {
   yyerror("too many symbols"); exit(1);
 }
 else { sp -> name = strdup(s);
        return sp;
      }
}
```

Error Recovery in YACC

- In order to prevent a cascade of error messages, the parser remains in error state (after entering it) until three tokens have been successfully shifted onto the stack
- In case an error happens before this, no further messages are given and the input symbol (causing the error) is quietly deleted
- The user may identify **major** nonterminals such as those for **program**, **statement**, or **block**, and add error productions for these to the grammar
- Examples
  statement → **error** {action1}
  statement → **error ';'** {action2}

YACC Error Recovery Example

```
%token DING DONG DELL
%start S
%%
S     :  rhyme{printf("string valid\n"); exit(0);}
rhyme : sound place
rhyme : error DELL{yyerror("msg1:token skipped");}
sound : DING DONG ;
place : DELL ;
place : error DELL{yyerror("msg2:token skipped");}
%%
```

Semantic Analysis with Attribute Grammars
Part 1

Y.N. Srikant

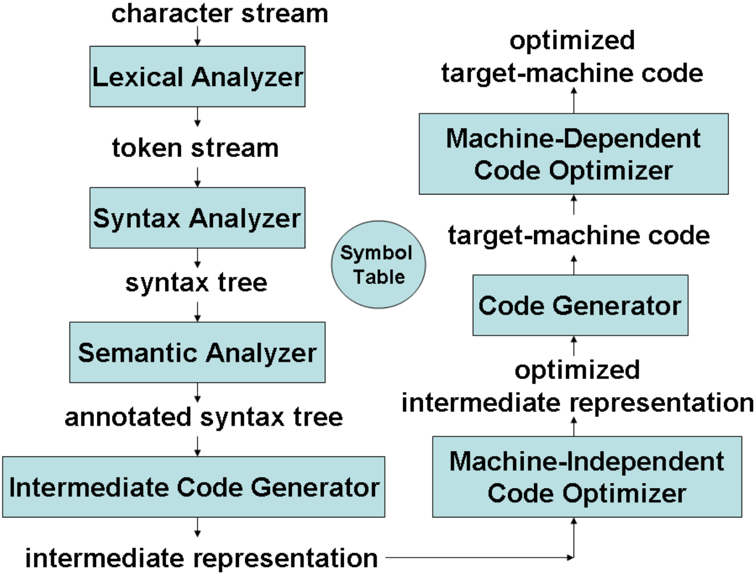
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

Compiler Overview



character stream

Lexical Analyzer

token stream

Syntax Analyzer

syntax tree

Semantic Analyzer

annotated syntax tree

Intermediate Code Generator

intermediate representation

Machine-Independent Code Optimizer

optimized intermediate representation

Code Generator

target-machine code

Machine-Dependent Code Optimizer

optimized target-machine code

Symbol Table

Semantic Analysis

- Semantic consistency that cannot be handled at the parsing stage is handled here
- Parsers cannot handle context-sensitive features of programming languages
- These are *static semantics* of programming languages and can be checked by the semantic analyzer
    - Variables are declared before use
    - Types match on both sides of assignments
    - Parameter types and number match in declaration and use
- Compilers can only generate code to check *dynamic semantics* of programming languages at runtime
    - whether an overflow will occur during an aritmetic operation
    - whether array limits will be crossed during execution
    - whether recursion will cross stack limits
    - whether heap memory will be insufficient

Static Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of static semantic checks in *main*

- Types of *p* and return type of *dot_prod* match
- Number and type of the parameters of *dot_prod* are the same in both its declaration and use
- *p* is declared before use, same for *a* and *b*

Static Semantics: Errors given by gcc Compiler

```
int dot_product(int a[], int b[]) {...}

1 main(){int a[10]={1,2,3,4,5,6,7,8,9,10};
2 int b[10]={1,2,3,4,5,6,7,8,9,10};
3 printf("%d", dot_product(b));
4 printf("%d", dot_product(a,b,a));
5 int p[10]; p=dotproduct(a,b); printf("%d",p);}
```

```
In function ‘main’:
error in 3: too few arguments to fn ‘dot_product’
error in 4: too many arguments to fn ‘dot_product’
error in 5: incompatible types in assignment
warning in 5: format ‘%d’ expects type ‘int’, but
              argument 2 has type ‘int *’
```

Static Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of static semantic checks in *dot_prod*

- *d* and *i* are declared before use
- Type of *d* matches the return type of *dot_prod*
- Type of *d* matches the result type of “$*$”
- Elements of arrays *x* and *y* are compatible with “$*$”

Dynamic Semantics

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Samples of dynamic semantic checks in *dot_prod*

- Value of *i* does not exceed the declared range of arrays *x* and *y* (both lower and upper)
- There are no overflows during the operations of "$*$" and "$+$" in `d += x[i]*y[i]`

Dynamic Semantics

```
int fact(int n){
   if (n==0) return 1;
   else return (n*fact(n-1));
}
main(){int p; p = fact(10); }
```

Samples of dynamic semantic checks in *fact*

- Program stack does not overflow due to recursion
- There is no overflow due to “∗” in `n*fact(n-1)`

Semantic Analysis

- Type information is stored in the symbol table or the syntax tree
    - Types of variables, function parameters, array dimensions, etc.
    - Used not only for semantic validation but also for subsequent phases of compilation
- If declarations need not appear before use (as in C++), semantic analysis needs more than one pass
- Static semantics of PL can be specified using attribute grammars
- Semantic analyzers can be generated semi-automatically from attribute grammars
- Attribute grammars are extensions of context-free grammars

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of *attributes* (denoted by $X.a$, $X.b$, etc.)
- Two types of attributes: *inherited* (denoted by $AI(X)$)and *synthesized* (denoted by $AS(X)$)
- Each attribute takes values from a specified domain (finite or infinite), which is its *type*
  - Typical domains of attributes are, integers, reals, characters, strings, booleans, structures, etc.
  - New domains can be constructed from given domains by mathematical operations such as *cross product, map*, etc.
  - *array*: a map, $\mathcal{N} \rightarrow \mathcal{D}$, where, $\mathcal{N}$ and $\mathcal{D}$ are domains of natural numbers and the given objects, respectively
  - *structure*: a cross-product, $A_1 \times A_2 \times \ldots \times A_n$, where n is the number of fields in the structure, and A_i is the domain of the i^{th} field

Attribute Computation Rules

- A production $p \in P$ has a set of attribute computation rules (functions)
- Rules are provided for the computation of
  - Synthesized attributes of the LHS non-terminal of p
  - Inherited attributes of the RHS non-terminals of p
- These rules can use attributes of symbols from the production p only
  - Rules are strictly local to the production p (no side effects)
- Restrictions on the rules define different types of attribute grammars
  - L-attribute grammars, S-attribute grammars, ordered attribute grammars, absolutely non-circular attribute grammars, circular attribute grammars, etc.

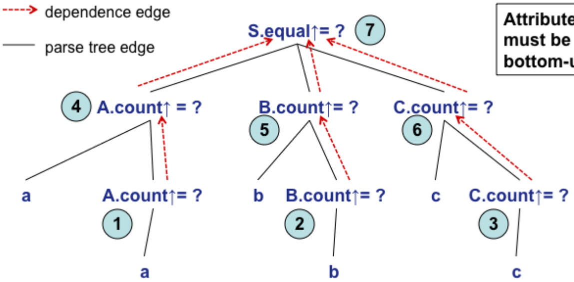
Synthesized and Inherited Attributes

- An attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes
- Attributes of symbols are evaluated over a parse tree by making passes over the parse tree
- Synthesized attributes are computed in a bottom-up fashion from the leaves upwards
    - Always synthesized from the attribute values of the children of the node
    - Leaf nodes (terminals) have synthesized attributes initialized by the lexical analyzer and cannot be modified
    - An AG with only synthesized attributes is an *S-attributed grammar (SAG)*
    - YACC permits only SAGs
- Inherited attributes flow down from the parent or siblings to the node in question

Attribute Grammar - Example 1

- The following CFG
  $S \rightarrow A\ B\ C,\ \ A \rightarrow aA \mid a,\ \ B \rightarrow bB \mid b,\ \ C \rightarrow cC \mid c$
  generates: $L(G) = \{a^m b^n c^p \mid m, n, p \geq 1\}$
- We define an AG (attribute grammar) based on this CFG to generate $L = \{a^n b^n c^n \mid n \geq 1\}$
- All the non-terminals will have only synthesized attributes
  - $AS(S) = \{equal \uparrow\colon\ \{T, F\}\}$
  - $AS(A) = AS(B) = AS(C) = \{count \uparrow\colon\ integer\}$

Attribute Grammar - Example 1 (contd.)

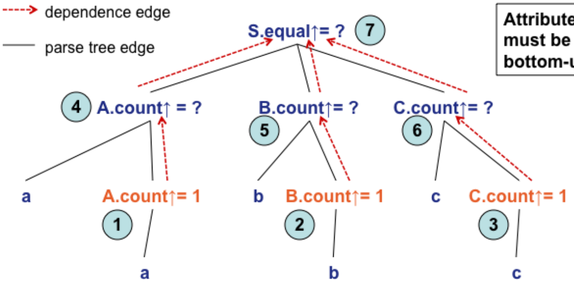


Attribute evaluation must be done in a bottom-up manner

S → ABC
A → aA | a
B → bB | b
C → cC | c

1. $S \rightarrow ABC\ \{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \&$
   $B.count \uparrow = C.count \uparrow \ \text{then } T \text{ else } F\}$
2. $A_1 \rightarrow aA_2\ \{A_1.count \uparrow := A_2.count \uparrow +1\}$
3. $A \rightarrow a\ \{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2\ \{B_1.count \uparrow := B_2.count \uparrow +1\}$
5. $B \rightarrow b\ \{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2\ \{C_1.count \uparrow := C_2.count \uparrow +1\}$
7. $C \rightarrow c\ \{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)

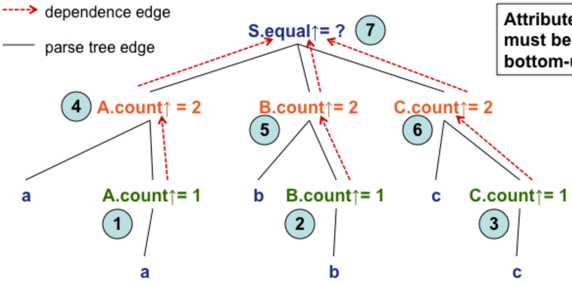


Attribute evaluation must be done in a bottom-up manner

S → ABC
A → aA | a
B → bB | b
C → cC | c

1. $S \rightarrow ABC$ $\{S.equal \uparrow := if\ A.count \uparrow = B.count \uparrow\ \&$ $B.count \uparrow = C.count \uparrow\ then\ T\ else\ F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count \uparrow := B_2.count \uparrow + 1\}$
5. $B \rightarrow b$ $\{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count \uparrow := C_2.count \uparrow + 1\}$
7. $C \rightarrow c$ $\{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)

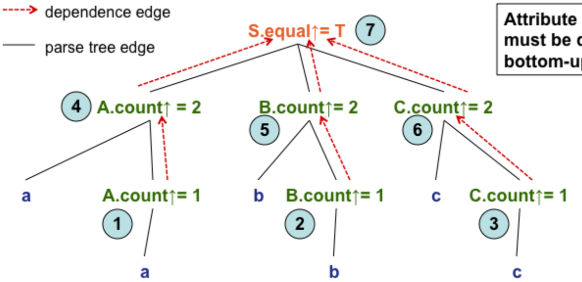


Attribute evaluation must be done in a bottom-up manner

S → ABC
A → aA | a
B → bB | b
C → cC | c

1. $S \rightarrow ABC$ $\{S.equal \uparrow := \textit{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \ \textit{then } T \textit{ else } F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count \uparrow := B_2.count \uparrow + 1\}$
5. $B \rightarrow b$ $\{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count \uparrow := C_2.count \uparrow + 1\}$
7. $C \rightarrow c$ $\{C.count \uparrow := 1\}$

Attribute Grammar - Example 1 (contd.)



Attribute evaluation must be done in a bottom-up manner

S → ABC
A → aA | a
B → bB | b
C → cC | c

1. $S \rightarrow ABC$ $\{S.equal \uparrow := \text{if } A.count \uparrow = B.count \uparrow \ \& \ B.count \uparrow = C.count \uparrow \ \text{then } T \text{ else } F\}$
2. $A_1 \rightarrow aA_2$ $\{A_1.count \uparrow := A_2.count \uparrow + 1\}$
3. $A \rightarrow a$ $\{A.count \uparrow := 1\}$
4. $B_1 \rightarrow bB_2$ $\{B_1.count \uparrow := B_2.count \uparrow + 1\}$
5. $B \rightarrow b$ $\{B.count \uparrow := 1\}$
6. $C_1 \rightarrow cC_2$ $\{C_1.count \uparrow := C_2.count \uparrow + 1\}$
7. $C \rightarrow c$ $\{C.count \uparrow := 1\}$

Attribute Dependence Graph

- Let T be a parse tree generated by the CFG of an AG, G.
- The *attribute dependence graph* (dependence graph for short) for T is the directed graph, $DG(T) = (V, E)$, where

  $V = \{b | b$ is an attribute instance of some tree node$\}$, and

  $E = \{(b, c) | b, c \in V$, b and c are attributes of grammar symbols in the same production p of B, and the value of b is used for computing the value of c in an attribute computation rule associated with production $p\}$

Attribute Dependence Graph

- An AG G is *non-circular*, iff for all trees T derived from G, $DG(T)$ is acyclic
    - Non-circularity is very expensive to determine (exponential in the size of the grammar)
    - Therefore, our interest will be in subclasses of AGs whose non-circularity can be determined efficiently
- Assigning consistent values to the attribute instances in DG(T) is *attribute evaluation*

Attribute Evaluation Strategy

- Construct the parse tree
- Construct the dependence graph
- Perform topological sort on the dependence graph and obtain an evaluation order
- Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective productions
- Multiple attributes at a node in the *parse tree* may result in that node to be visited multiple number of times
    - Each visit resulting in the evaluation of at least one attribute

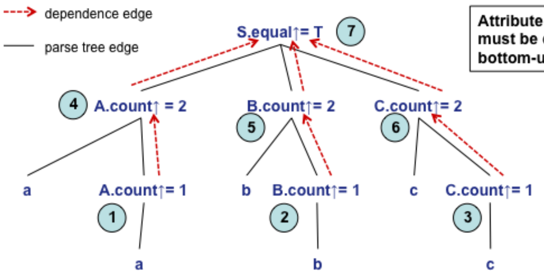
Attribute Evaluation Algorithm

Input: A parse tree T with unevaluated attribute instances

Output: T with consistent attribute values

```
{ Let $(V, E) = DG(T)$;
  Let $W = \{b \mid b \in V \; \& \; indegree(b) = 0\}$;
  while $W \neq \phi$ do
    { remove some $b$ from $W$;
      $value(b)$ := value defined by appropriate attribute
                    computation rule;
      for all $(b, c) \in E$ do
        { $indegree(c) := indegree(c) - 1$;
          if $indegree(c) = 0$ then $W := W \cup \{c\}$;
        }
    }
}
```

Dependence Graph for Example 1



Attribute evaluation must be done in a bottom-up manner

S → ABC
A → aA | a
B → bB | b
C → cC | c

1,2,3,4,5,6,7 and 2,3,6,5,1,4,7 are two possible evaluation orders. 1,4,2,5,3,6,7 can be used with LR-parsing. The right-most derivation is below (its reverse is LR-parsing order)

S => ABC => ABcC => ABcc => AbBcc => Abbcc => aAbbcc => aabbcc

1. A.count = 1 {A → a, {A.count := 1}}
4. A.count = 2 {A_1 → aA_2, {A_1.count := A_2.count + 1}}
2. B.count = 1 {B → b, {B.count :=1}}
5. B.count = 2 {B_1 → bB_2, {B_1.count := B_2.count + 1}}
3. C.count = 1 {C → c, {C.count :=1}}
6. C.count = 2 {C_1 → cC_2, {C_1.count := C_2.count + 1}}
7. S.equal = 1 {S → ABC, {S.equal := *if* A.count = B.count & B.count = C.count *then* T else F}}

Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation
  Example: 110.101 = 6.625
- $N \rightarrow L.R,\ L \rightarrow BL \mid B,\ R \rightarrow BR \mid B,\ B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value \uparrow : real\}$,
  $AS(L) = \{length \uparrow : integer,\ value \uparrow : real\}$
  1. $N \rightarrow L.R$ $\{N.value \uparrow := L.value \uparrow + R.value \uparrow\}$
  2. $L \rightarrow B$ $\{L.value \uparrow := B.value \uparrow;\ L.length \uparrow := 1\}$
  3. $L_1 \rightarrow BL_2$ $\{L_1.length \uparrow := L_2.length \uparrow + 1;$
     $L_1.value \uparrow := B.value \uparrow * 2^{L_2.length\uparrow} + L_2.value \uparrow\}$
  4. $R \rightarrow B$ $\{R.value \uparrow := B.value \uparrow /2\}$
  5. $R_1 \rightarrow BR_2$ $\{R_1.value \uparrow := (B.value \uparrow + R_2.value \uparrow)/2\}$
  6. $B \rightarrow 0$ $\{B.value \uparrow := 0\}$
  7. $B \rightarrow 1$ $\{B.value \uparrow := 1\}$

Semantic Analysis with Attribute Grammars
Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of *attributes*
- Two types of attributes: *inherited* and *synthesized*
- Each attribute takes values from a specified domain
- A production $p \in P$ has a set of attribute computation rules for
  - synthesized attributes of the LHS non-terminal of p
  - inherited attributes of the RHS non-terminals of p
- Rules are strictly local to the production p (no side effects)

Synthesized and Inherited Attributes

- An attribute cannot be both synthesized and inherited, but a symbol can have both types of attributes
- Attributes of symbols are evaluated over a parse tree by making passes over the parse tree
- Synthesized attributes are computed in a bottom-up fashion from the leaves upwards
    - Always synthesized from the attribute values of the children of the node
    - Leaf nodes (terminals) have synthesized attributes (only) initialized by the lexical analyzer and cannot be modified
- Inherited attributes flow down from the parent or siblings to the node in question

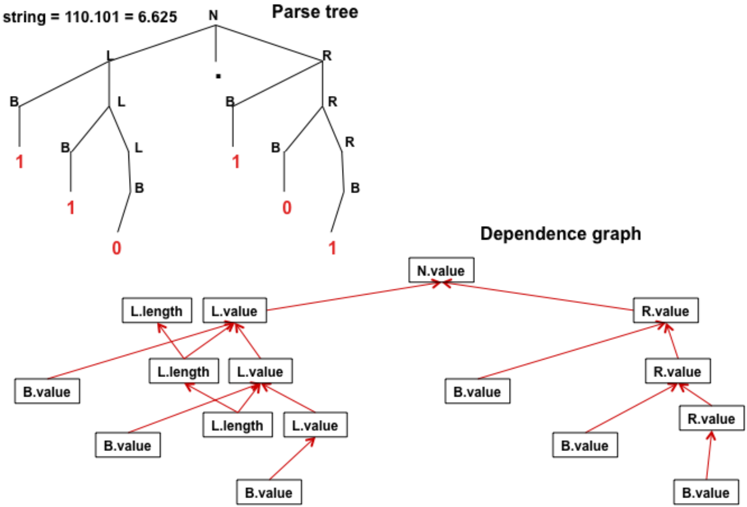
Attribute Evaluation Strategy

- Construct the parse tree
- Construct the dependence graph
- Perform topological sort on the dependence graph and obtain an evaluation order
- Evaluate attributes according to this order using the corresponding attribute evaluation rules attached to the respective productions

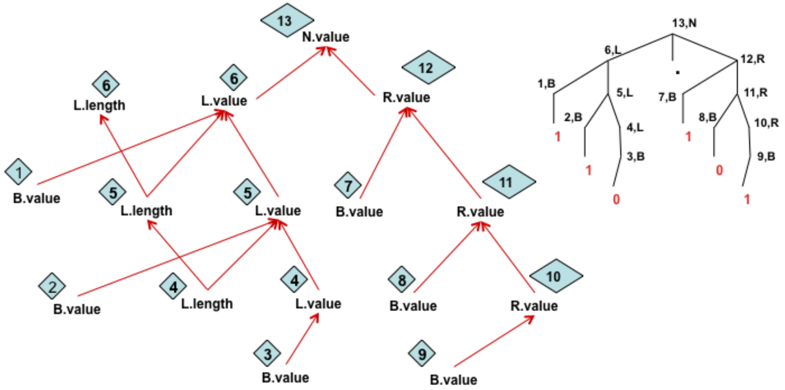
Attribute Grammar - Example 2

- AG for the evaluation of a real number from its bit-string representation
  Example: 110.101 = 6.625
- $N \rightarrow L.R,\ \ L \rightarrow BL \mid B,\ \ R \rightarrow BR \mid B,\ \ B \rightarrow 0 \mid 1$
- $AS(N) = AS(R) = AS(B) = \{value\uparrow : real\}$,
  $AS(L) = \{length\uparrow : integer,\ \ value\uparrow : real\}$
  1. $N \rightarrow L.R\ \{N.value\uparrow := L.value\uparrow + R.value\uparrow\}$
  2. $L \rightarrow B\ \{L.value\uparrow := B.value\uparrow;\ \ L.length\uparrow := 1\}$
  3. $L_1 \rightarrow BL_2\ \{L_1.length\uparrow := L_2.length\uparrow + 1;$
     $L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$
  4. $R \rightarrow B\ \{R.value\uparrow := B.value\uparrow /2\}$
  5. $R_1 \rightarrow BR_2\ \{R_1.value\uparrow := (B.value\uparrow + R_2.value\uparrow)/2\}$
  6. $B \rightarrow 0\ \{B.value\uparrow := 0\}$
  7. $B \rightarrow 1\ \{B.value\uparrow := 1\}$

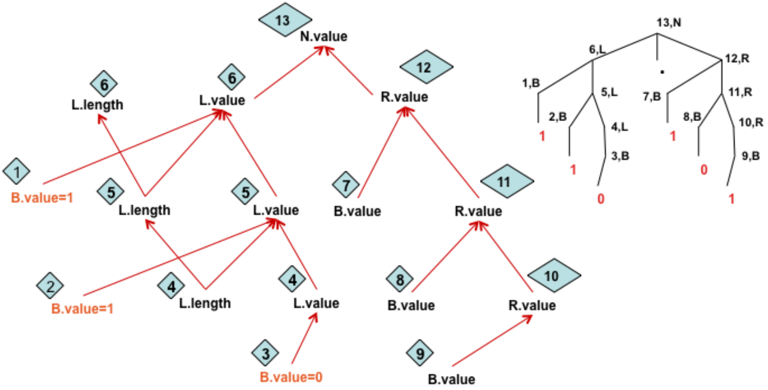
Dependence Graph for Example 2



Attribute Evaluation for Example 2 - 1

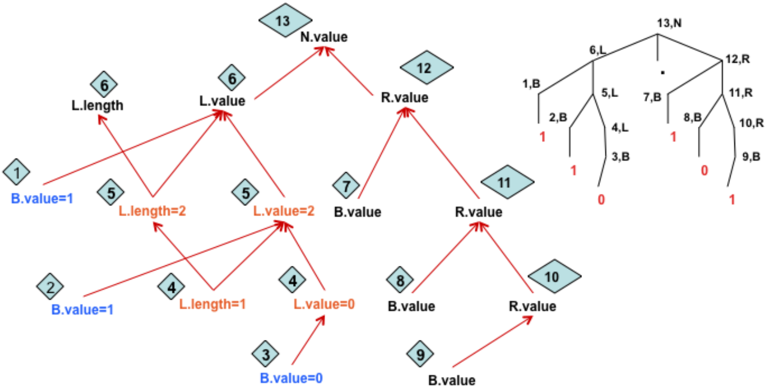


Attribute Evaluation for Example 2 - 2



Nodes 1,2: $B \rightarrow 1$ $\{B.value \uparrow := 1\}$
Node 3: $B \rightarrow 0$ $\{B.value \uparrow := 0\}$

Attribute Evaluation for Example 2 - 3

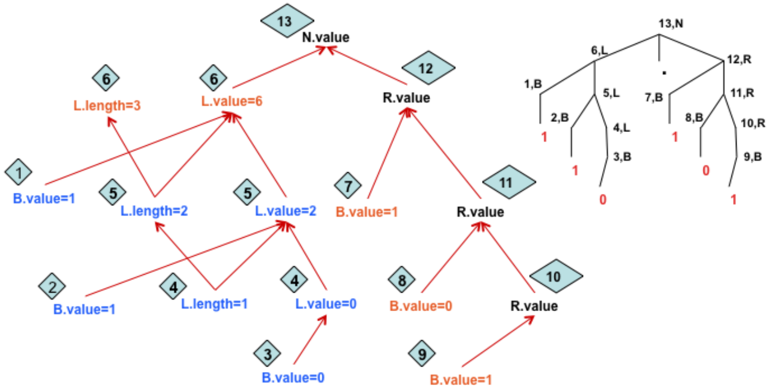


Node 4: $L \rightarrow B$ $\{L.value\uparrow := B.value\uparrow;\ L.length\uparrow := 1\}$

Node 5: $L_1 \rightarrow BL_2$ $\{L_1.length\uparrow := L_2.length\uparrow + 1;$

$L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$

Attribute Evaluation for Example 2 - 4



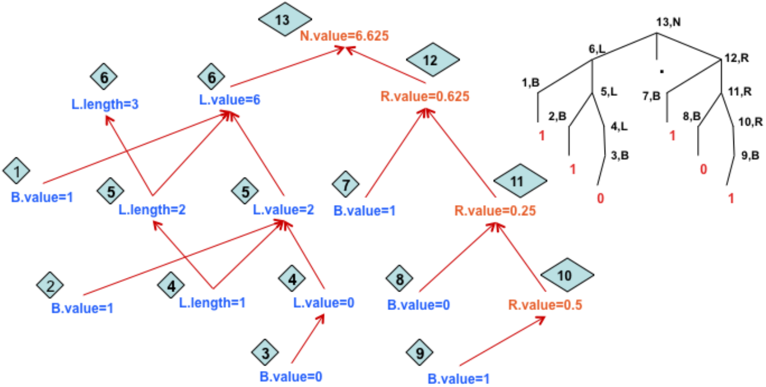
Node 6: $L_1 \rightarrow BL_2$ $\{L_1.length\uparrow := L_2.length\uparrow + 1;$

$L_1.value\uparrow := B.value\uparrow * 2^{L_2.length\uparrow} + L_2.value\uparrow\}$

Nodes 7,9: $B \rightarrow 1$ $\{B.value\uparrow := 1\}$

Node 8: $B \rightarrow 0$ $\{B.value\uparrow := 0\}$

Attribute Evaluation for Example 2 - 5



Node 10: $R \rightarrow B$ $\{R.value\uparrow := B.value\uparrow /2\}$

Nodes 11,12:

$R_1 \rightarrow BR_2$ $\{R_1.value\uparrow := (B.value\uparrow + R_2.value\uparrow)/2\}$

Node 13: $N \rightarrow L.R$ $\{N.value\uparrow := L.value\uparrow + R.value\uparrow\}$

Attribute Grammar - Example 3

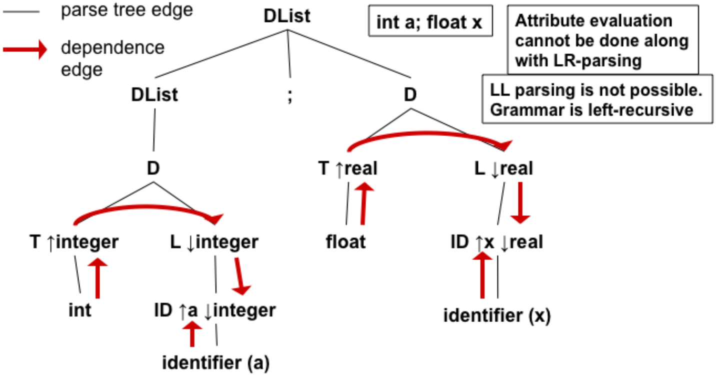
- A simple AG for the evaluation of a real number from its bit-string representation
  Example: $110.1010 = 6 + 10/2^4 = 6 + 10/16 = 6 + 0.625 = 6.625$
- $N \rightarrow X.X,\ X \rightarrow BX \mid B,\ B \rightarrow 0 \mid 1$
- $AS(N) = AS(B) = \{value \uparrow : real\}$,
  $AS(X) = \{length \uparrow : integer,\ value \uparrow : real\}$
  1. $N \rightarrow X_1.X_2\ \{N.value \uparrow := X_1.value \uparrow + X_2.value \uparrow / 2^{X_2.length}\}$
  2. $X \rightarrow B\ \{X.value \uparrow := B.value \uparrow;\ X.length \uparrow := 1\}$
  3. $X_1 \rightarrow BX_2\ \{X_1.length \uparrow := X_2.length \uparrow + 1;$
     $X_1.value \uparrow := B.value \uparrow * 2^{X_2.length \uparrow} + X_2.value \uparrow\}$
  4. $B \rightarrow 0\ \{B.value \uparrow := 0\}$
  5. $B \rightarrow 1\ \{B.value \uparrow := 1\}$

Attribute Grammar - Example 4

- An AG for associating *type* information with names in variable declarations
- $AI(L) = AI(ID) = \{type\downarrow: \{integer, real\}\}$
  $AS(T) = \{type\uparrow: \{integer, real\}\}$
  $AS(ID) = AS(identifier) = \{name\uparrow: string\}$
  1. $DList \rightarrow D \mid DList\ ;\ D$
  2. $D \rightarrow T\ L\ \{L.type\downarrow := T.type\uparrow\}$
  3. $T \rightarrow int\ \{T.type\uparrow := integer\}$
  4. $T \rightarrow float\ \{T.type\uparrow := real\}$
  5. $L \rightarrow ID\ \{ID.type\downarrow := L.type\downarrow\}$
  6. $L_1 \rightarrow L_2\ ,\ ID\ \{L_2.type\downarrow := L_1.type\downarrow; ID.type\downarrow := L_1.type\downarrow\}$
  7. $ID \rightarrow identifier\ \{ID.name\uparrow := identifier.name\uparrow\}$

Example: *int* a,b,c; *float* x,y
a,b, and c are tagged with type *integer*
x,y, and z are tagged with type *real*

Attribute Evaluation for Example 4



int a; float x

Attribute evaluation cannot be done along with LR-parsing

LL parsing is not possible. Grammar is left-recursive

1. $DList \rightarrow D \mid DList\ ;\ D$ **2.** $D \rightarrow T\ L\ \{L.type \downarrow := T.type \uparrow\}$

3. $T \rightarrow int\ \{T.type \uparrow := integer\}$ **4.** $T \rightarrow float\ \{T.type \uparrow := real\}$

5. $L \rightarrow ID\ \{ID.type \downarrow := L.type \downarrow\}$

6. $L_1 \rightarrow L_2\ ,\ ID\ \{L_2.type \downarrow := L_1.type \downarrow; ID.type \downarrow := L_1.type \downarrow\}$

7. $ID \rightarrow identifier\ \{ID.name \uparrow := identifier.name \uparrow\}$

Attribute Grammar - Example 5

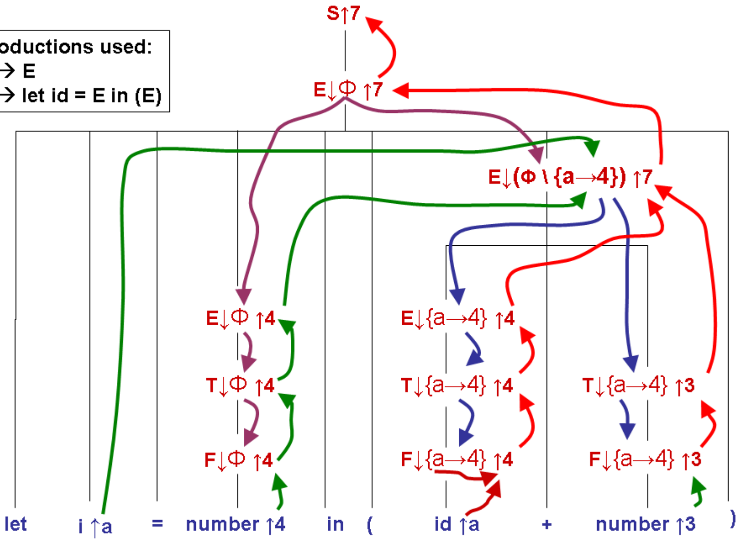
- Let us first consider the CFG for a simple language
  1. $S \longrightarrow E$
  2. $E \longrightarrow E + T \mid T \mid let\ id = E\ in\ (E)$
  3. $T \longrightarrow T * F \mid F$
  4. $F \longrightarrow (E) \mid number \mid id$
- This language permits expressions to be nested inside expressions and have scopes for the names
  - *let A = 5 in ((let A = 6 in (A*7)) - A)* evaluates correctly to 37, with the scopes of the two instances of *A* being different
- It requires a scoped symbol table for implementation
- An abstract attribute grammar for the above language uses both inherited and synthesized attributes
- Both inherited and synthesized attributes can be evaluated in one pass (from left to right) over the parse tree
- Inherited attributes cannot be evaluated during LR parsing

Attribute Grammar - Example 5

1. $S \longrightarrow E$ $\{E.symtab \downarrow := \phi; S.val \uparrow := E.val \uparrow\}$
2. $E_1 \longrightarrow E_2 + T$ $\{E_2.symtab \downarrow := E_1.symtab \downarrow;$
   $E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow; T.symtab \downarrow := E_1.symtab \downarrow\}$
3. $E \longrightarrow T$ $\{T.symtab \downarrow := E.symtab \downarrow; E.val \uparrow := T.val \uparrow\}$
4. $E_1 \longrightarrow let\ id = E_2\ in\ (E_3)$
   $\{E_1.val \uparrow := E_3.val \uparrow; E_2.symtab \downarrow := E_1.symtab \downarrow;$
   $E_3.symtab \downarrow := E_1.symtab \downarrow \backslash \{id.name \uparrow \rightarrow E_2.val \uparrow\}\}$
5. $T_1 \longrightarrow T_2 * F$ $\{T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow;$
   $T_2.symtab \downarrow := T1.symtab \downarrow; F.symtab \downarrow := T_1.symtab \downarrow\}$
6. $T \longrightarrow F$ $\{T.val \uparrow := F.val \uparrow; F.symtab \downarrow := T.symtab \downarrow\}$
7. $F \longrightarrow (E)$ $\{F.val \uparrow := E.val \uparrow; E.symtab \downarrow := F.symtab \downarrow\}$
8. $F \longrightarrow number$ $\{F.val \uparrow := number.val \uparrow\}$
9. $F \longrightarrow id$ $\{F.val \uparrow := F.symtab \downarrow [id.name \uparrow]\}$

Attribute Flow and Evaluation - Example 5

productions used:
S → E
E → let id = E in (E)



L-Attributed and S-Attributed Grammars

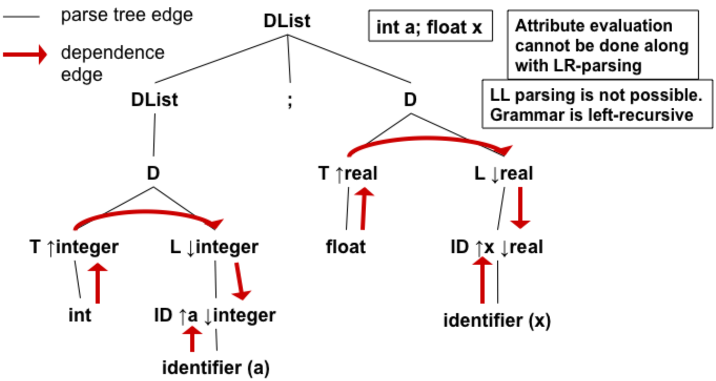
- An AG with only synthesized attributes is an S-attributed grammar
    - Attributes of SAGs can be evaluated in any bottom-up order over a parse tree (single pass)
    - Attribute evaluation can be combined with LR-parsing (YACC)
- In L-attributed grammars, attribute dependencies always go from *left to right*
- More precisely, each attribute must be
    - Synthesized, or
    - Inherited, but with the following limitations:
      consider a production $p: \ A \rightarrow X_1X_2...X_n$. Let $X_i.a \in AI(X_i)$.
      $X_i.a$ may use only
        - elements of $AI(A)$
        - elements of $AI(X_k)$ or $AS(X_k)$, $k = 1, ..., i-1$
          (*i.e.,* attibutes of $X_1, ..., X_{i-1}$)
- We concentrate on SAGs, and 1-pass LAGs, in which attribute evaluation can be combined with LR, LL or RD parsing

Attribute Evaluation Algorithm for LAGs

Input: A parse tree T with unevaluated attribute instances
Output: T with consistent attribute values

```
void dfvisit(n: node)
{  for each child m of n, from left to right do
     {  evaluate inherited attributes of m;
        dfvisit(m)
     };
   evaluate synthesized attributes of n
}
```

Example of LAG - 1



int a; float x

Attribute evaluation cannot be done along with LR-parsing

LL parsing is not possible. Grammar is left-recursive

1. $DList \rightarrow D \mid DList\ ;\ D$ **2.** $D \rightarrow T\ L\ \{L.type\downarrow := T.type\uparrow\}$

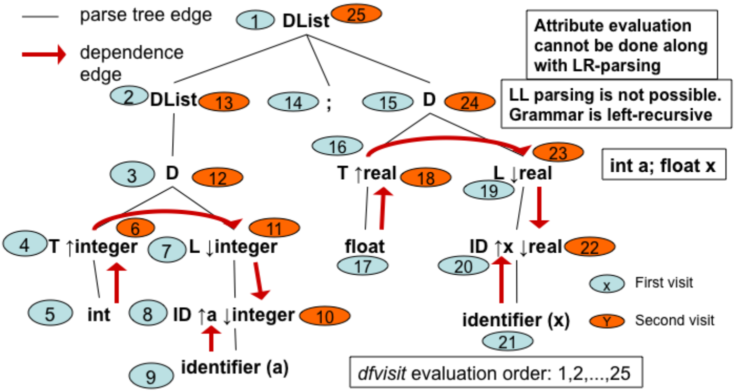
3. $T \rightarrow int\ \{T.type\uparrow := integer\}$ **4.** $T \rightarrow float\ \{T.type\uparrow := real\}$

5. $L \rightarrow ID\ \{ID.type\downarrow := L.type\downarrow\}$

6. $L_1 \rightarrow L_2\ ,\ ID\ \{L_2.type\downarrow := L_1.type\downarrow;\ ID.type\downarrow := L_1.type\downarrow\}$

7. $ID \rightarrow identifier\ \{ID.name\uparrow := identifier.name\uparrow\}$

Example of LAG - 1, Evaluation Order



Attribute evaluation cannot be done along with LR-parsing

LL parsing is not possible. Grammar is left-recursive

int a; float x

dfvisit evaluation order: 1,2,...,25

1. $DList \rightarrow D \mid DList\ ;\ D$ **2.** $D \rightarrow T\ L\ \{L.type\downarrow := T.type\uparrow\}$

3. $T \rightarrow int\ \{T.type\uparrow := integer\}$ **4.** $T \rightarrow float\ \{T.type\uparrow := real\}$

5. $L \rightarrow ID\ \{ID.type\downarrow := L.type\downarrow\}$

6. $L_1 \rightarrow L_2\ ,\ ID\ \{L_2.type\downarrow := L_1.type\downarrow;\ ID.type\downarrow := L_1.type\downarrow\}$

7. $ID \rightarrow identifier\ \{ID.name\uparrow := identifier.name\uparrow\}$

Semantic Analysis with Attribute Grammars
Part 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars
- Attributed translation grammars
- Semantic analysis with attributed translation grammars

Attribute Grammars

- Let $G = (N, T, P, S)$ be a CFG and let $V = N \cup T$.
- Every symbol X of V has associated with it a set of *attributes*
- Two types of attributes: *inherited* and *synthesized*
- Each attribute takes values from a specified domain
- A production $p \in P$ has a set of attribute computation rules for
  - synthesized attributes of the LHS non-terminal of p
  - inherited attributes of the RHS non-terminals of p
- Rules are strictly local to the production p (no side effects)

L-Attributed and S-Attributed Grammars

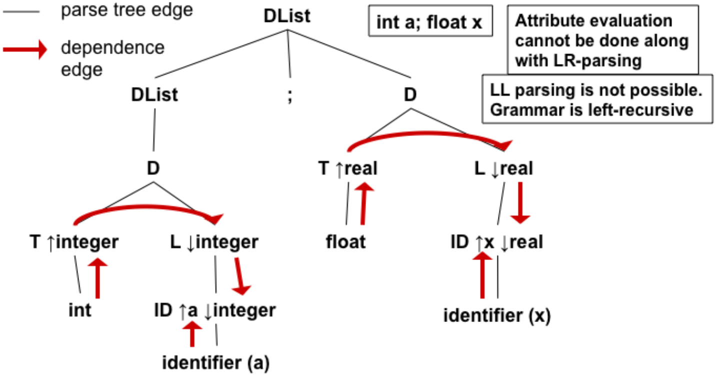
- An AG with only synthesized attributes is an S-attributed grammar
  - Attributes of SAGs can be evaluated in any bottom-up order over a parse tree (single pass)
  - Attribute evaluation can be combined with LR-parsing (YACC)
- In L-attributed grammars, attribute dependencies always go from *left to right*
- More precisely, each attribute must be
  - Synthesized, or
  - Inherited, but with the following limitations:
    consider a production $p: \ A \rightarrow X_1 X_2 ... X_n$. Let $X_i.a \in AI(X_i)$.
    $X_i.a$ may use only
    - elements of $AI(A)$
    - elements of $AI(X_k)$ or $AS(X_k), \ k = 1, ..., i-1$
      (*i.e.,* attibutes of $X_1, ..., X_{i-1}$)
- We concentrate on SAGs, and 1-pass LAGs, in which attribute evaluation can be combined with LR, LL or RD parsing

Attribute Evaluation Algorithm for LAGs

Input: A parse tree *T* with unevaluated attribute instances
Output: *T* with consistent attribute values

```
void dfvisit(n: node)
{  for each child m of n, from left to right do
     {  evaluate inherited attributes of m;
        dfvisit(m)
     };
   evaluate synthesized attributes of n
}
```

Example of LAG - 1



int a; float x

Attribute evaluation cannot be done along with LR-parsing

LL parsing is not possible. Grammar is left-recursive

1. $DList \rightarrow D \mid DList\ ;\ D$ **2.** $D \rightarrow T\ L\ \{L.type\downarrow := T.type\uparrow\}$

3. $T \rightarrow int\ \{T.type\uparrow := integer\}$ **4.** $T \rightarrow float\ \{T.type\uparrow := real\}$

5. $L \rightarrow ID\ \{ID.type\downarrow := L.type\downarrow\}$

6. $L_1 \rightarrow L_2\ ,\ ID\ \{L_2.type\downarrow := L_1.type\downarrow; ID.type\downarrow := L_1.type\downarrow\}$

7. $ID \rightarrow identifier\ \{ID.name\uparrow := identifier.name\uparrow\}$

Example of Non-LAG

- An AG for associating *type* information with names in variable declarations
- $AI(L) = AI(ID) = \{type\downarrow: \{integer, real\}\}$
  $AS(T) = \{type\uparrow: \{integer, real\}\}$
  $AS(ID) = AS(identifier) = \{name\uparrow: string\}$
  1. $DList \rightarrow D \mid DList\ ;\ D$
  2. $D \rightarrow L\ :\ T\ \{L.type\downarrow := T.type\uparrow\}$
  3. $T \rightarrow int\ \{T.type\uparrow := integer\}$
  4. $T \rightarrow float\ \{T.type\uparrow := real\}$
  5. $L \rightarrow ID\ \{ID.type\downarrow := L.type\downarrow\}$
  6. $L_1 \rightarrow L_2\ ,\ ID\ \{L_2.type\downarrow := L_1.type\downarrow; ID.type\downarrow := L_1.type\downarrow\}$
  7. $ID \rightarrow identifier\ \{ID.name\uparrow := identifier.name\uparrow\}$

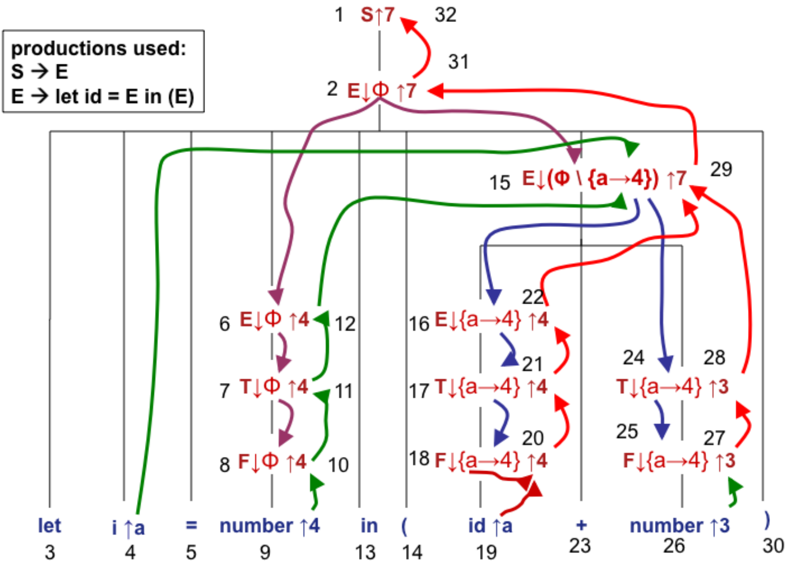
Example: a,b,c: *int*; x,y: *float*
a,b, and c are tagged with type *integer*
x,y, and z are tagged with type *real*

Example of LAG - 2

1. $S \longrightarrow E$ $\{E.symtab \downarrow := \phi;\ S.val \uparrow := E.val \uparrow\}$
2. $E_1 \longrightarrow E_2 + T$ $\{E_2.symtab \downarrow := E_1.symtab \downarrow;$ $E_1.val \uparrow := E_2.val \uparrow + T.val \uparrow;\ T.symtab \downarrow := E_1.symtab \downarrow\}$
3. $E \longrightarrow T$ $\{T.symtab \downarrow := E.symtab \downarrow;\ E.val \uparrow := T.val \uparrow\}$
4. $E_1 \longrightarrow let\ id = E_2\ in\ (E_3)$ $\{E_1.val \uparrow := E_3.val \uparrow;\ E_2.symtab \downarrow := E_1.symtab \downarrow;$ $E_3.symtab \downarrow := E_1.symtab \downarrow \backslash \{id.name \uparrow \rightarrow E_2.val \uparrow\}\}$

   Note: changing the above production to: $E_1 \rightarrow return\ (E_3)\ with\ id = E_2$ **(with the same computation rules) changes this AG into non-LAG**
5. $T_1 \longrightarrow T_2 * F$ $\{T_1.val \uparrow := T_2.val \uparrow * F.val \uparrow;$ $T_2.symtab \downarrow := T1.symtab \downarrow;\ F.symtab \downarrow := T_1.symtab \downarrow\}$
6. $T \longrightarrow F$ $\{T.val \uparrow := F.val \uparrow;\ F.symtab \downarrow := T.symtab \downarrow\}$
7. $F \longrightarrow (E)$ $\{F.val \uparrow := E.val \uparrow;\ E.symtab \downarrow := F.symtab \downarrow\}$
8. $F \longrightarrow number$ $\{F.val \uparrow := number.val \uparrow\}$
9. $F \longrightarrow id$ $\{F.val \uparrow := F.symtab \downarrow [id.name \uparrow]\}$

Example of LAG - 2, Evaluation Order



productions used:
S → E
E → let id = E in (E)

Attributed Translation Grammar

- Apart from attribute computation rules, some program segment that performs either output or some other side effect-free computation is added to the AG
- Examples are: symbol table operations, writing generated code to a file, etc.
- As a result of these *action code segments*, evaluation orders may be constrained
- Such constraints are added to the attribute dependence graph as *implicit edges*
- These actions can be added to both SAGs and LAGs (making them, SATG and LATG resp.)
- Our discussion of semantic analysis will use LATG(1-pass) and SATG

Example 1: SATG for Desk Calculator

```
%%
lines: lines expr '\n' {printf("%g\n",$2);}
     | lines '\n'
     | /* empty */
     ;
expr : expr '+' expr {$$ = $1 + $3;}
/*Same as: expr(1).val = expr(2).val+expr(3).val */
     | expr '-' expr {$$ = $1 - $3;}
     | expr '*' expr {$$ = $1 * $3;}
     | expr '/' expr {$$ = $1 / $3;}
     | '(' expr ')' {$$ = $2;}
     | NUMBER /* type double */
     ;
%%
```

Example 2: SATG for Modified Desk Calculator

```
%%
lines: lines expr ’\n’ {printf("%g\n",$2);}
     | lines ’\n’
     | /* empty */
     ;
expr : NAME ’=’ expr {sp = symlook($1);
       sp->value = $3; $$ = $3;}
     | NAME {sp = symlook($1);$$ = sp->value;}
     | expr ’+’ expr {$$ = $1 + $3;}
     | expr ’-’ expr {$$ = $1 - $3;}
     | expr ’*’ expr {$$ = $1 + $3;}
     | expr ’/’ expr {$$ = $1 - $3;}
     | ’(’ expr ’)’ {$$ = $2;}
     | NUMBER /* type double */
     ;
%%
```

Example 3: LAG, LATG, and SATG

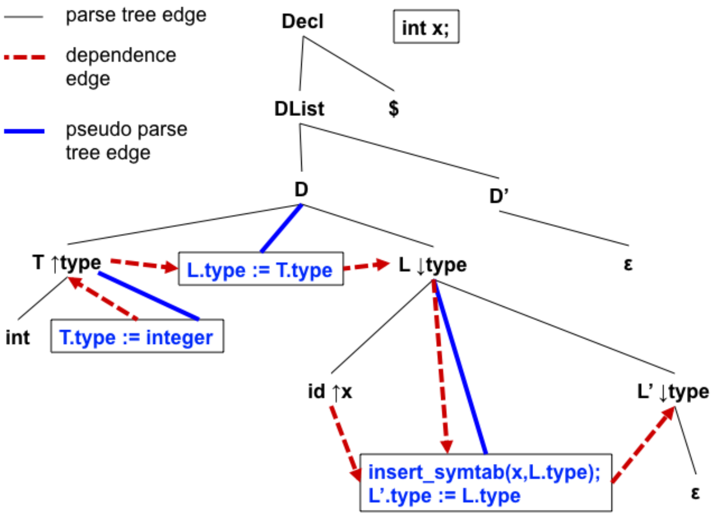
LAG (notice the changed grammar)

1. $Decl \rightarrow DList\$$ **2.** $DList \rightarrow D\ D'$ **3.** $D' \rightarrow \epsilon \mid ;\ DList$
4. $D \rightarrow T\ L\ \{L.type \downarrow := T.type \uparrow\}$
5. $T \rightarrow int\ \{T.type \uparrow := integer\}$ **6.** $T \rightarrow float\ \{T.type \uparrow := real\}$
7. $L \rightarrow ID\ L' \{ID.type \downarrow := L.type \downarrow;\ \ L'.type \downarrow := L.type \downarrow; \}$
8. $L' \rightarrow \epsilon \mid ,\ L\ \{L.type \downarrow := L'.type \downarrow; \}$
9. $ID \rightarrow identifier\ \{ID.name \uparrow := identifier.name \uparrow\}$

LATG (notice the changed grammar)

1. $Decl \rightarrow DList\$$ **2.** $DList \rightarrow D\ D'$ **3.** $D' \rightarrow \epsilon \mid ;\ DList$
4. $D \rightarrow T\ \{L.type \downarrow := T.type \uparrow\}\ L$
5. $T \rightarrow int\ \{T.type \uparrow := integer\}$ **6.** $T \rightarrow float\ \{T.type \uparrow := real\}$
7. $L \rightarrow id\ \{insert\_symtab(id.name \uparrow, L.type \downarrow);$
$L'.type \downarrow := L.type \downarrow; \}\ L'$
8. $L' \rightarrow \epsilon \mid ,\ \{L.type \downarrow := L'.type \downarrow; \}\ L$

Example - 3: LATG Dependence Example



Example 3: LAG, LATG, and SATG (contd.)

SATG

1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D \mid DList\ ;\ D$
3. $D \rightarrow T\ L\ \{patchtype(T.type\uparrow, L.namelist\uparrow);\}$
4. $T \rightarrow int\ \{T.type\uparrow := integer\}$
5. $T \rightarrow float\ \{T.type\uparrow := real\}$
6. $L \rightarrow id\ \{sp = insert\_symtab(id.name\uparrow);$
$L.namelist\uparrow = makelist(sp);\}$
7. $L_1 \rightarrow L_2\ ,\ id\ \{sp = insert\_symtab(id.name\uparrow);$
$L_1.namelist\uparrow = append(L_2.namelist\uparrow, sp);\}$

Integrating LATG into RD Parser - 1

```
/* Decl --> DList $*/
void Decl(){Dlist();
            if mytoken.token == EOF return
            else error(); }
/* DList --> D D' */
void DList(){D(); D'(); }
/* D --> T {L.type := T.type} L */
void D(){vartype type = T(); L(type); }
/* T --> int {T.type := integer}
       | float {T.type := real} */
vartype T(){if mytoken.token == INT
               {get_token(); return(integer);}
            else if mytoken.token == FLOAT
                   {get_token(); return(real); }
                 else error();
}
```

Integrating LATG into RD Parser - 2

```
/* L --> id {insert_symtab(id.name, L.type);
             L'.type := L.type} L' */
void L(vartype type){if mytoken.token == ID
            {insert_symtab(mytoken.value, type);
             get_token(); L'(type); } else error();
}
/* L' --> empty | ,{L.type := L'.type} L */
void L'(vartype type){if mytoken.token == COMMA
                {get_token(); L(type);} else ;
}
/* D' --> empty | ; DList */
void D'(){if mytoken.token == SEMICOLON
             {get_token(); DList(); } else ; }
```

Example 4: SATG with Scoped Names

```
1. S --> E  { S.val := E.val }
2. E --> E + T  { E(1).val := E(2).val + T.val }
3. E --> T  { E.val := T.val }
/* The 3 productions below are broken parts
   of the prod.: E --> let id =  E in (E) */
4. E --> L B  { E.val := B.val; }
5. L --> let id = E { //scope initialized to 0;
         scope++; insert (id.name, scope, E.val) }
6. B --> in (E) { B.val := E.val;
         delete_entries (scope); scope--;  }
7. T --> T * F  { T(1).val := T(2).val * F.val }
8. T --> F  { T.val := F.val }
9. F --> (E) { F.val := E.val }
10. F --> number  { F.val := number.val }
11. F --> id { F.val := getval (id.name, scope) }
```

LATG for Sem. Analysis of Variable Declarations - 1

1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D \mid D\ ;\ DList$
3. $D \rightarrow T\ L$
4. $T \rightarrow int \mid float$
5. $L \rightarrow ID\_ARR \mid ID\_ARR\ ,\ L$
6. $ID\_ARR \rightarrow id \mid id\ [\ DIMLIST\ ] \mid id\ BR\_DIMLIST$
7. $DIMLIST \rightarrow num \mid num,\ DIMLIST$
8. $BR\_DIMLIST \rightarrow [\ num\ ] \mid [\ num\ ]\ BR\_DIMLIST$

Note: array declarations have two possibilities

`int a[10,20,30]; float b[25][35];`

LATG for Sem. Analysis of Variable Declarations - 2

- The grammar is not LL(1) and hence an LL(1) parser cannot be built from it.
- We assume that the parse tree is available and that attribute evaluation is performed over the parse tree
- Modifications to the CFG to make it LL(1) and the corresponding changes to the AG are left as exercises
- The attributes and their rules of computation for productions 1-4 are as before and we ignore them
- We provide the AG only for the productions 5-7; AG for rule 8 is similar to that of rule 7
- Handling constant declarations is similar to that of handling variable declarations

Identifier Type Information in the Symbol Table

Identifier type information record

name	type	eletype	dimlist_ptr

1. *type*: (simple, array)
2. *type* = simple for non-array names
3. The fields *eletype* and *dimlist_ptr* are relevant only for arrays. In that case, *type* = array
4. *eletype*: (integer, real, errortype), is the type of a simple id or the type of the array element
5. *dimlist_ptr* points to a list of ranges of the dimensions of an array. C-type array declarations are assumed

   Ex. float my_array[5][12][15]

   dimlist_ptr points to the list (5,12,15), and the total number elements in the array is 5x12x15 = 900, which can be obtained by *traversing* this list and multiplying the elements.

Semantic Analysis with Attribute Grammars
Part 4

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars (covered in lectures 2 and 3)
- Attributed translation grammars (covered in lecture 3)
- Semantic analysis with attributed translation grammars

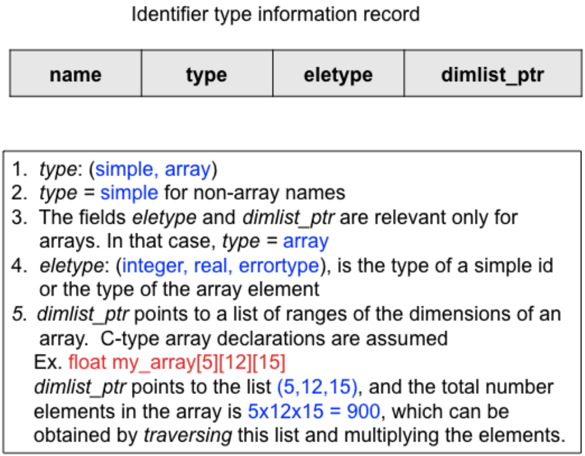
LATG for Sem. Analysis of Variable Declarations - 1

1. $Decl \rightarrow DList\$$
2. $DList \rightarrow D \mid D\ ;\ DList$
3. $D \rightarrow T\ L$
4. $T \rightarrow int \mid float$
5. $L \rightarrow ID\_ARR \mid ID\_ARR\ ,\ L$
6. $ID\_ARR \rightarrow id \mid id\ [\ DIMLIST\ ] \mid id\ BR\_DIMLIST$
7. $DIMLIST \rightarrow num \mid num,\ DIMLIST$
8. $BR\_DIMLIST \rightarrow [\ num\ ] \mid [\ num\ ]\ BR\_DIMLIST$

LATG for Sem. Analysis of Variable Declarations - 2

- The grammar is not LL(1) and hence an LL(1) parser cannot be built from it.
- We assume that the parse tree is available and that attribute evaluation is performed over the parse tree
- Modifications to the CFG to make it LL(1) and the corresponding changes to the AG are left as exercises
- The attributes and their rules of computation for productions 1-4 are as before and we ignore them
- We provide the AG only for the productions 5-7; AG for rule 8 is similar to that of rule 7
- Handling constant declarations is similar to that of handling variable declarations

Identifier Type Information in the Symbol Table



Identifier type information record

name	**type**	**eletype**	**dimlist_ptr**

1. *type*: (simple, array)
2. *type* = simple for non-array names
3. The fields *eletype* and *dimlist_ptr* are relevant only for arrays. In that case, *type* = array
4. *eletype*: (integer, real, errortype), is the type of a simple id or the type of the array element
5. *dimlist_ptr* points to a list of ranges of the dimensions of an array. C-type array declarations are assumed

   Ex. float my_array[5][12][15]

   dimlist_ptr points to the list (5,12,15), and the total number elements in the array is 5x12x15 = 900, which can be obtained by *traversing* this list and multiplying the elements.

LATG for Sem. Analysis of Variable Declarations - 3

1. $L_1 \rightarrow$ {ID_ARR.type↓ := L_1.type↓ } *ID_ARR* ,
   {L_2.type↓ := L_1.type↓;} L_2
2. $L \rightarrow \{ID\_ARR.type \downarrow := L.type \downarrow\}$ *ID_ARR*
3. *ID_ARR* → *id*

```
{ search_symtab(id.name↑, found);
  if (found) error('identifier already declared');
  else { typerec* t; t->type := simple;
           t->eletype := ID_ARR.type↓;
           insert_symtab(id.name↑, t);}
}
```

LATG for Sem. Analysis of Variable Declarations - 4

4. $ID\_ARR \rightarrow id\ [\ DIMLIST\ ]$

```
{ search ...; if (found) ...;
  else { typerec* t; t->type := array;
              t->eletype := ID_ARR.type↓;
              t->dimlist_ptr := DIMLIST.ptr↑;
              insert_symtab(id.name↑, t)}
}
```

5. $DIMLIST \rightarrow num$

   {DIMLIST.ptr↑ := makelist(num.value↑)}

6. $DIMLIST_1 \rightarrow num,\ DIMLIST_2$

   {$DIMLIST_1.ptr\uparrow$:= append(num.value↑, $DIMLIST_2.ptr\uparrow$)}

Storage Offset Computation for Variables

- The compiler should compute
  - the offsets at which variables and constants will be stored in the activation record (AR)
- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations
- Example: `float c; int d[10]; float e[5,15]; int a,b;`
  The offsets are: c-0, d-8, e-48, a-648, b-652,
  assuming that `int` takes 4 bytes and `float` takes 8 bytes

LATG for Storage Offset Computation

1. $Decl \rightarrow DList\$$

   $Decl \rightarrow \{\ \text{DList.inoffset}\downarrow\ := 0;\ \}\ DList\$$
2. $DList \rightarrow D$

   $DList \rightarrow \{\ \text{D.inoffset}\downarrow\ := \text{DList.inoffset}\downarrow;\ \}\ D$
3. $DList_1 \rightarrow D\ ;\ DList_2$

   $DList_1 \rightarrow \{\ \text{D.inoffset}\downarrow\ := DList_1\text{.inoffset}\downarrow;\ \}\ \ D\ ;$

   $\qquad\qquad \{\ DList_2\text{.inoffset}\downarrow\ := \text{D.outoffset}\uparrow;\}\ DList_2$
4. $D \rightarrow T\ L$

   $D \rightarrow T\ \{\ \text{L.inoffset}\downarrow\ := \text{D.inoffset}\downarrow;\ \ \text{L.typesize}\downarrow\ := \text{T.size}\uparrow;\}$

   $\qquad\quad L\ \{\ \text{D.outoffset}\uparrow\ := \text{L.outoffset}\uparrow;\}$
5. $T \rightarrow int\ \mid\ float$

   $T \rightarrow int\ \{\text{T.size}\uparrow\ := 4;\ \}\ \mid\ float\ \{\text{T.size}\uparrow\ := 8;\ \}$

Storage Offset Example

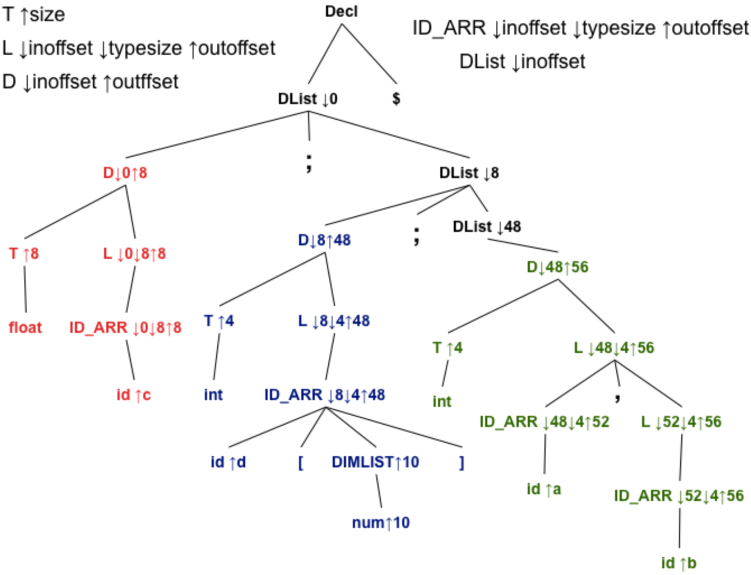
T ↑size

L ↓inoffset ↓typesize ↑outoffset

D ↓inoffset ↑outffset

ID_ARR ↓inoffset ↓typesize ↑outoffset

DList ↓inoffset



LATG for Storage Offset Computation(contd.)

6. $L \rightarrow ID\_ARR$
   $L \rightarrow$ { ID_ARR.inoffset↓ := L.inoffset↓;
   ID_ARR.typesize↓ := L.typesize↓; }
   ID_ARR { L.outoffset↑ := ID_ARR.outoffset↑; }
7. $L_1 \rightarrow ID\_ARR\ ,\ L_2$
   $L_1 \rightarrow$ { ID_ARR.inoffset↓ := L_1.inoffset↓;
   ID_ARR.typesize↓ := L_1.typesize↓; }
   ID_ARR , { L_2.inoffset↓ := ID_ARR.outoffset↑;
   L_2.typesize↓ := L_1.typesize↓; }
   L_2{ L_1.outoffset↑ := L_2.outoffset↑; }
8. $ID\_ARR \rightarrow id$
   $ID\_ARR \rightarrow id$ { insert_offset(id.name, ID_ARR.inoffset↓);
   ID_ARR.outoffset↑ := ID_ARR.inoffset↓ +
   ID_ARR.typesize↓ }

Storage Offset Example

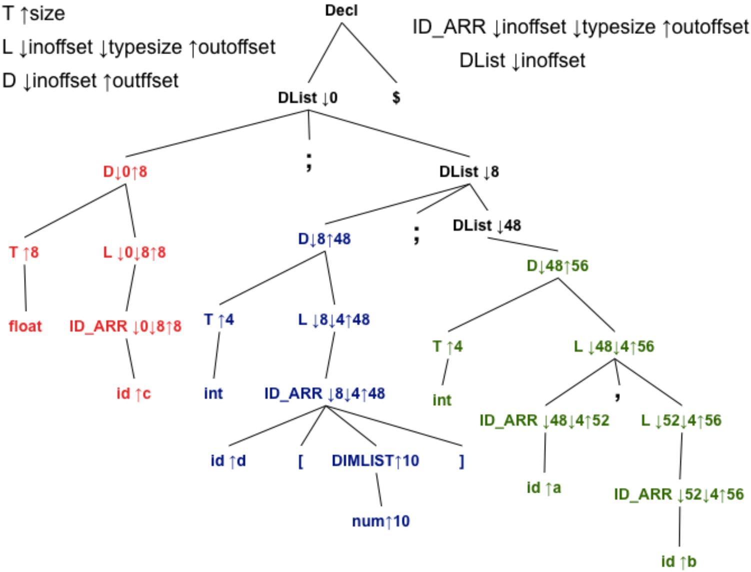
T ↑size

L ↓inoffset ↓typesize ↑outoffset

D ↓inoffset ↑outffset

ID_ARR ↓inoffset ↓typesize ↑outoffset

DList ↓inoffset


Decl

DList ↓0

$

D↓0↑8

;

DList ↓8

T ↑8

L ↓0↓8↑8

D↓8↑48

;

DList ↓48

float

ID_ARR ↓0↓8↑8

T ↑4

L ↓8↓4↑48

D↓48↑56

id ↑c

int

ID_ARR ↓8↓4↑48

T ↑4

L ↓48↓4↑56

int

id ↑d

[

DIMLIST↑10

]

ID_ARR ↓48↓4↑52

,

L ↓52↓4↑56

num↑10

id ↑a

ID_ARR ↓52↓4↑56

id ↑b

LATG for Storage Offset Computation(contd.)

9. *ID_ARR* → *id* [*DIMLIST*]
   ID_ARR → *id* { insert_offset(id.name, ID_ARR.inoffset↓);
   [*DIMLIST*] ID_ARR.outoffset↑ :=
   ID_ARR.inoffset↓ + ID_ARR.typesize↓ × DIMLIST.num }
10. *DIMLIST* → *num* { DIMLIST.num↑ := num.value↑; }
11. $\mathit{DIMLIST}_1$ → *num* , $\mathit{DIMLIST}_2$
    { $\mathit{DIMLIST}_1$.num↑ := $\mathit{DIMLIST}_2$.num↑ × num.value↑; }
12. *ID_ARR* → *id BR_DIMLIST*
13. *BR_DIMLIST* → [*num*] | [*num*] *BR_DIMLIST*
    Processing productions 12 and 13 is similar to that of the previous productions, 9-11

Storage Offset Example

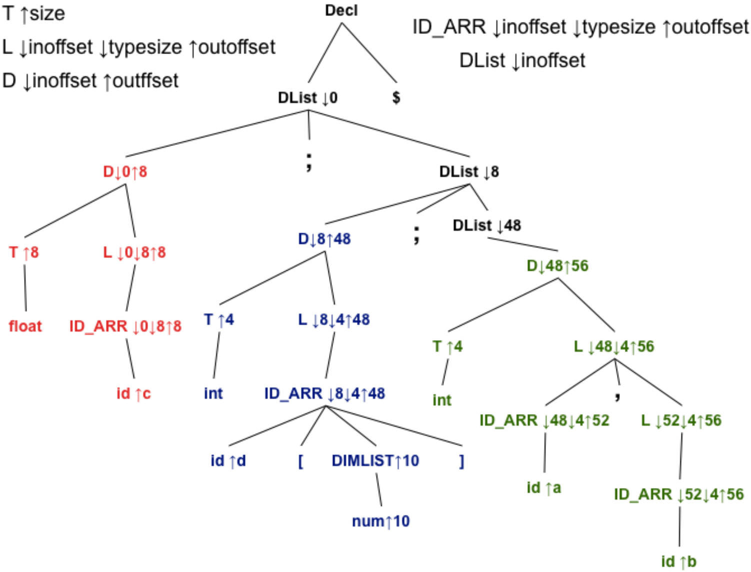
T ↑size

L ↓inoffset ↓typesize ↑outoffset

D ↓inoffset ↑outffset

ID_ARR ↓inoffset ↓typesize ↑outoffset

DList ↓inoffset



SATG for Sem. Analysis of Statements and Expr. - 1

1. $S \rightarrow if\ E\ then\ S \mid if\ E\ then\ S\ else\ S$
2. $S \rightarrow while\ E\ do\ S$
3. $S \rightarrow L\ :=\ E$
4. $L \rightarrow id \mid id\ [\ ELIST\ ]$
5. $ELIST \rightarrow E \mid ELIST\ ,\ E$
6. $E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid -E \mid (E) \mid L \mid num$
7. $E \rightarrow E || E \mid E \&\& E \mid \sim E$
8. $E \rightarrow E < E \mid E > E \mid E == E$

- We assume that the parse tree is available and that attribute evaluation is performed over the parse tree
- The grammar above is ambiguous and changing it appropriately to suit parsing is necessary
- Actions for similar rules are skipped (to avoid repetition)

SATG for Sem. Analysis of Statements and Expr. - 2

All attributes are synthesized and therefore ↑ symbol is dropped (for brevity)

- *E*, *L*, and *num*: *type*: {integer, real, boolean, errortype}
  /* Note: *num* will also have *value* as an attribute */
- *ELIST*: *dimnum*: integer

1. $S \rightarrow IFEXP\ then\ S$
2. $IFEXP \rightarrow if\ E$ {if (E.type $\neq$ *boolean*)
   error('boolean expression expected');}
3. $S \rightarrow WHILEEXP\ do\ S$
4. $WHILEEXP \rightarrow while\ E$ {if (E.type $\neq$ *boolean*)
   error('boolean expression expected');}

SATG for Sem. Analysis of Statements and Expr. - 3

5. $S \rightarrow L \; := \; E$

```
{if (L.type $\neq$ errortype && E.type $\neq$ errortype)
   if $\sim$coercible(L.type, E.type)
      error('type mismatch of operands
                in assignment statement');}
```

```
int coercible( types type_a, types type_b ){
    if ((type_a == integer || type_a == real) &&
      (type_b == integer || type_b == real))
      return 1; else return 0;
}
```

SATG for Sem. Analysis of Statements and Expr. - 4

Identifier type information record

name	type	eletype	dimlist_ptr

1. *type*: (simple, array)
2. *type* = simple for non-array names
3. The fields *eletype* and *dimlist_ptr* are relevant only for arrays. In that case, *type* = array
4. *eletype*: (integer, real, errortype), is the type of a simple id or the type of the array element
5. *dimlist_ptr* points to a list of ranges of the dimensions of an array. C-type array declarations are assumed

   Ex. float my_array[5][12][15]

   dimlist_ptr points to the list (5,12,15), and the total number elements in the array is 5x12x15 = 900, which can be obtained by *traversing* this list and multiplying the elements.

SATG for Sem. Analysis of Statements and Expr. - 5

6. $E \rightarrow num$ {E.type := num.type;}
7. $L \rightarrow id$

```
{ typerec* t; search_symtab(id.name, missing, t);
  if (missing) { error('identifier not declared');
                 L.type := errortype;}
  else if (t->type == array)
            { error('cannot assign whole arrays');
              L.type := errortype;}
        else L.type := t->eletype;}
```

SATG for Sem. Analysis of Statements and Expr. - 6

8. $L \rightarrow id\ [\ ELIST\ ]$

```
{ typerec* t; search_symtab(id.name, missing, t);
  if (missing) { error('identifier not declared');
                   L.type := errortype}
  else { if (t->type $\neq$ array)
             { error('identifier not of array type');
               L.type := errortype;}
         else { find_dim(t->dimlist_ptr, dimnum);
                  if (dimnum $\neq$ ELIST.dimnum)
                    { error('mismatch in array
                         declaration and use; check index list');
                      L.type := errortype;}
                  else L.type := t->eletype;}
```

SATG for Sem. Analysis of Statements and Expr. - 7

9. $ELIST \rightarrow E$ {If (E.type $\neq$ integer)

   ```
       error('illegal subscript type'); ELIST.dimnum := 1;}
   ```

10. $ELIST_1 \rightarrow ELIST_2 \, , \; E$ {If (E.type $\neq$ integer)

    ```
        error('illegal subscript type');
           $ELIST_1$.dimnum := $ELIST_2$.dimnum+1;}
    ```

11. $E_1 \rightarrow E_2 + E_3$

    ```
    {if ($E_2$.type $\neq$ errortype && $E_3$.type $\neq$ errortype)
      if ($\sim$coercible($E_2$.type, $E_3$.type)||
        $\sim$(compatible_arithop($E_2$.type, $E_3$.type))
         {error('type mismatch in expression');
           $E_1$.type := errortype;}
      else $E_1$.type := compare_types($E_2$.type, $E_3$.type);
    else $E_1$.type := errortype;}
    ```

SATG for Sem. Analysis of Statements and Expr. - 8

```
int compatible_arithop( types type_a, types type_b ){
    if ((type_a == integer || type_a == real) &&
      (type_b == integer || type_b == real))
      return 1; else return 0;
}
types compare_types( types type_a, types type_b ){
        if (type_a == integer && type_b == integer)
          return integer;
        else if (type_a == real && type_b == real)
          return real;
        else if (type_a == integer && type_b == real)
          return real;
        else if (type_a == real && type_b == integer)
          return real;
        else return error_type;
}
```

SATG for Sem. Analysis of Statements and Expr. - 9

⑫ $E_1 \rightarrow E_2 \,||\, E_3$

```
{if ($E_2$.type $\neq$ errortype && $E_3$.type $\neq$ errortype)
    if (($E_2$.type == boolean || $E_2$.type == integer) &&
        ($E_3$.type == boolean || $E_3$.type == integer))
         $E_1$.type := boolean;
    else {error('type mismatch in expression');
              $E_1$.type := errortype;}
 else $E_1$.type := errortype;}
```

⑬ $E_1 \rightarrow E_2 \; < \; E_3$

```
{if ($E_2$.type $\neq$ errortype && $E_3$.type $\neq$ errortype)
   if ($\sim$coercible($E_2$.type, $E_3$.type)||
      $\sim$(compatible_arithop($E_2$.type, $E_3$.type))
       {error('type mismatch in expression');
         $E_1$.type := errortype;}
   else $E_1$.type := boolean;
 else $E_1$.type := errortype;}
```

Semantic Analysis with Attribute Grammars
Part 5

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars (covered in lectures 2 and 3)
- Attributed translation grammars (covered in lecture 3)
- Semantic analysis with attributed translation grammars

Symbol Table Data Structure

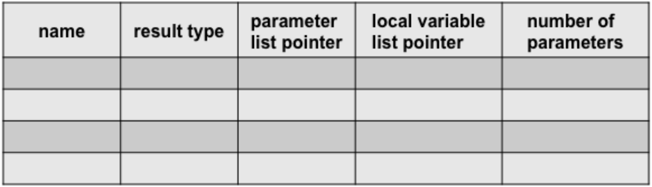
- A symbol table (in a compiler) stores names of all kinds that occur in a program along with information about them
    - Type of the name (int, float, function, etc.), level at which it has been declared, whether it is a declared parameter of a function or an ordinary variable, etc.
    - In the case of a function, additional information about the list of parameters and their types, local variables and their types, result type, etc., are also stored
- It is used during semantic analysis, optimization, and code generation
- Symbol table must be organized to enable a search based on the level of declaration
- It can be based on:
    - Binary search tree, hash table, array, etc.

A Simple Symbol Table - 1

- A very simple symbol table (quite restricted and not really fast) is presented for use in the semantic analysis of functions
- An array, *func_name_table* stores the function name records, assuming no nested function definitions
- Each function name record has fields: name, result type, parameter list pointer, and variable list pointer
- Parameter and variable names are stored as lists
- Each parameter and variable name record has fields: name, type, parameter-or-variable tag, and level of declaration (1 for parameters, and 2 or more for variables)

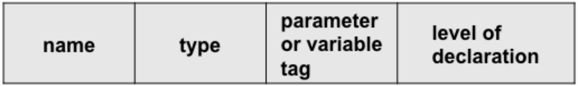
A Simple Symbol Table - 2

func_name_table



name	result type	parameter list pointer	local variable list pointer	number of parameters

Parameter/Variable name record

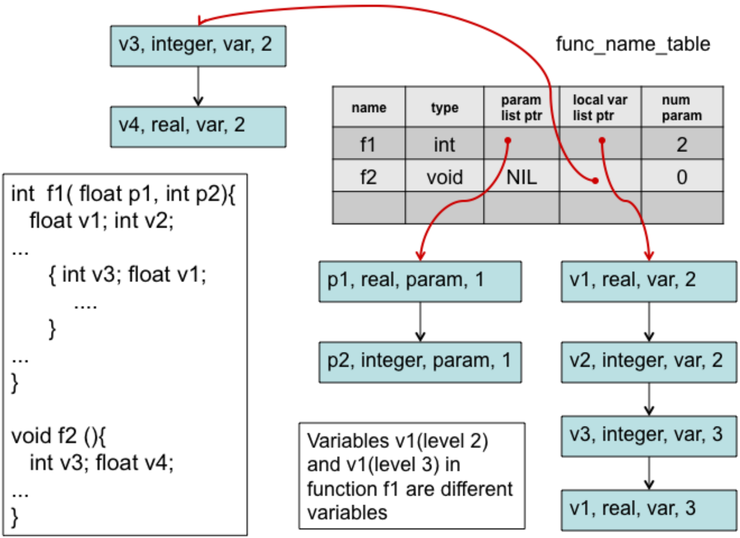


name	type	parameter or variable tag	level of declaration

A Simple Symbol Table - 3

- Two variables in the same function, with the same name but different declaration levels, are treated as different variables (in their respective scopes)
- If a variable (at level > 2) and a parameter have the same name, then the variable name overrides the parameter name (only within the corresponding scope)
- However, a declaration of a variable at level 2, with the same name as a parameter, is flagged as an error
- The above two cases must be checked carefully
- A search in the symbol table for a given name must always consider the names with the declaration levels *l, l-1, ... , 2*, in that order, where *l* is the current level

A Simple Symbol Table - 4



```
int  f1( float p1, int p2){
   float v1; int v2;
...
      { int v3; float v1;
          ....
      }
...
}

void f2 (){
   int v3; float v4;
...
}
```

func_name_table

name	type	param list ptr	local var list ptr	num param
f1	int	●	●	2
f2	void	NIL	●	0

Variables v1(level 2) and v1(level 3) in function f1 are different variables

A Simple Symbol Table - 5

- The global variable, *active_func_ptr*, stores a pointer to the function name entry in *func_name_table* of the function that is currently being compiled
- The global variable, *level*, stores the current nesting level of a statement block
- The global variable, *call_name_ptr*, stores a pointer to the function name entry in *func_name_table* of the function whose call is being currently processed
- The function $search\_func(n, found, fnptr)$ searches the function name table for the name *n* and returns *found* as T or F; if found, it returns a pointer to that entry in *fnptr*

A Simple Symbol Table - 6

- The function *search_param*$(p, fnptr, found, pnptr)$ searches the parameter list of the function at *fnptr* for the name *p*, and returns *found* as T or F; if found, it returns a pointer to that entry in the parameter list, in *pnptr*
- The function *search_var*$(v, fnptr, l, found, vnptr)$ searches the variable list of the function at *fnptr* for the name *v* at level *l* or lower, and returns *found* as T or F; if found, it returns a pointer to that entry in the variable list, in *vnptr*. Higher levels are preferred
- The other symbol table routines will be explained during semantic analysis

SATG for Sem. Analysis of Functions and Calls - 1

1. $FUNC\_DECL \rightarrow FUNC\_HEAD \; \{ \; VAR\_DECL \; BODY \; \}$
2. $FUNC\_HEAD \rightarrow RES\_ID \; (\; DECL\_PLIST \;)$
3. $RES\_ID \rightarrow RESULT \;\; id$
4. $RESULT \rightarrow int \mid float \mid void$
5. $DECL\_PLIST \rightarrow DECL\_PL \mid \epsilon$
6. $DECL\_PL \rightarrow DECL\_PL \; , \; DECL\_PARAM \mid DECL\_PARAM$
7. $DECL\_PARAM \rightarrow T \; id$
8. $VAR\_DECL \rightarrow DLIST \mid \epsilon$
9. $DLIST \rightarrow D \mid DLIST \; ; \; D$
10. $D \rightarrow T \; L$
11. $T \rightarrow int \mid float$
12. $L \rightarrow id \mid L \; , \; id$

SATG for Sem. Analysis of Functions and Calls - 2

13. $BODY \rightarrow \{\ VAR\_DECL\ STMT\_LIST\ \}$
14. $STMT\_LIST \rightarrow STMT\_LIST\ ;\ STMT \mid STMT$
15. $STMT \rightarrow BODY \mid FUNC\_CALL \mid ASG \mid$ /* others */
    /* BODY may be regarded as a *compound statement* */
    /* Assignment statement is being singled out */
    /* to show how function calls can be handled */
16. $ASG \rightarrow LHS\ :=\ E$
17. $LHS \rightarrow id$ /* array expression for exercises */
18. $E \rightarrow LHS \mid FUNC\_CALL \mid$/* other expressions */
19. $FUNC\_CALL \rightarrow id\ (\ PARAMLIST\ )$
20. $PARAMLIST \rightarrow PLIST \mid \epsilon$
21. $PLIST \rightarrow PLIST\ ,\ E \mid E$

SATG for Sem. Analysis of Functions and Calls - 3

1. *FUNC_DECL* → *FUNC_HEAD* { *VAR_DECL BODY* }
   {delete_var_list(active_func_ptr, level);
   active_func_ptr := NULL; level := 0;}
2. *FUNC_HEAD* → *RES_ID* (*DECL_PLIST*) {level := 2}
3. *RES_ID* → *RESULT* *id*
   { search_func(id.name, found, namptr);
   if (found) error('function already declared');
   else enter_func(id.name, RESULT.type, namptr);
   active_func_ptr := namptr; level := 1}
4. *RESULT* → *int* {**action1**} | *float* {**action2**}
   | *void* {**action3**}
   {action 1:} {RESULT.type := integer}
   {action 2:} {RESULT.type := real}
   {action 3:} {RESULT.type := void}

SATG for Sem. Analysis of Functions and Calls - 4

5. $DECL\_PLIST \rightarrow DECL\_PL \mid \epsilon$
6. $DECL\_PL \rightarrow DECL\_PL \; , \; DECL\_PARAM \mid DECL\_PARAM$
7. $DECL\_PARAM \rightarrow T \; id$
   {search_param(id.name, active_func_ptr, found, pnptr);
    if (found) {error('parameter already declared')}
    else {enter_param(id.name, T.type, active_func_ptr)}
8. $T \rightarrow int$ {T.type := integer} | $float$ {T.type := real}
9. $VAR\_DECL \rightarrow DLIST \mid \epsilon$
10. $DLIST \rightarrow D \mid DLIST \; ; \; D$
    /* We show the analysis of simple variable declarations. Arrays can be handled using methods desribed earlier. Extension of the symbol table and SATG to handle arrays is left as an exercise. */

SATG for Sem. Analysis of Functions and Calls - 5

11. *D* $\rightarrow$ *T L* {patch_var_type(T.type, L.list, level)}
    /* Patch all names on L.list with declaration level, *level*, with T.type */
12. *L* $\rightarrow$ *id*

```
{search_var(id.name, active_func_ptr, level, found, vn);
 if (found && vn -> level == level)
     {error('variable already declared at the same level');
        L.list := makelist(NULL);}
 else if (level==2)
{search_param(id.name, active_func_ptr, found, pn);
 if (found) {error('redeclaration of parameter as variable');
                  L.list := makelist(NULL);}
} /* end of if (level == 2) */
 else {enter_var(id.name, level, active_func_ptr, vnptr);
          L.list := makelist(vnptr);}}
```

SATG for Sem. Analysis of Functions and Calls - 6

⑬ $L_1 \rightarrow L_2$, *id*

```
{search_var(id.name, active_func_ptr, level, found, vn);
 if (found && vn -> level == level)
     {error(‘variable already declared at the same level’);
        $L_1$.list := $L_2$.list;}
 else if (level==2)
{search_param(id.name, active_func_ptr, found, pn);
 if (found) {error(‘redclaration of parameter as variable’);
                  $L_1$.list := $L_2$.list;}
} /* end of if (level == 2) */
 else {enter_var(id.name, level, active_func_ptr, vnptr);
           $L_1$.list := append($L_2$.list, vnptr);}}
```

⑭ *BODY* → ‘{’{level++;} *VAR_DECL* *STMT_LIST*
  {delete_var_list(active_func_ptr, level); level- -;}‘}’

⑮ *STMT_LIST* → *STMT_LIST* ; *STMT* | *STMT*

⑯ *STMT* → *BODY* | *FUNC_CALL* | *ASG* | /* others */

SATG for Sem. Analysis of Functions and Calls - 7

17. $ASG \rightarrow LHS\ :=\ E$

    ```
    {if (LHS.type ≠ errortype && E.type ≠ errortype)
       if (LHS.type ≠ E.type) error(‘type mismatch of
             operands in assignment statement’)}
    ```

18. $LHS \rightarrow id$

    ```
    {search_var(id.name, active_func_ptr, level, found, vn);
      if (∼found)
        {search_param(id.name, active_func_ptr, found, pn);
          if (∼found){ error(‘identifier not declared’);
                            LHS.type := errortype}
          else LHS.type := pn -> type}
      else LHS.type := vn -> type}
    ```

19. $E \rightarrow LHS$ {E.type := LHS.type}
20. $E \rightarrow FUNC\_CALL$ {E.type := FUNC_CALL.type}

SATG for Sem. Analysis of Functions and Calls - 8

21. *FUNC_CALL → id (PARAMLIST)*

```
{ search_func(id.name, found, fnptr);
  if (∼found) {error(‘function not declared’);
                    call_name_ptr := NULL;
                    FUNC_CALL.type := errortype;}
  else {FUNC_CALL.type := get_result_type(fnptr);
           call_name_ptr := fnptr;
  if (call_name_ptr.numparam ≠ PARAMLIST.pno)
     error(‘mismatch in mumber of parameters
             in declaration and call’);}
```

22. *PARAMLIST → PLIST* {PARAMLIST.pno := PLIST.pno }
    | ϵ {PARAMLIST.pno := 0 }

SATG for Sem. Analysis of Functions and Calls - 9

㉓ $PLIST \rightarrow E$ {PLIST.pno := 1;
check_param_type(call_name_ptr, 1, E.type, ok);
if (∼ok) error('parameter type mismatch
in declaration and call');}

㉔ $PLIST_1 \rightarrow PLIST_2\ ,\ E$ {$PLIST_1$.pno := $PLIST_2$.pno + 1;
check_param_type(call_name_ptr, $PLIST_2$.pno + 1,
E.type, ok);
if (∼ok) error('parameter type mismatch
in declaration and call');}

Semantic Analysis of Arrays

Multi-dimensional arrays

- length of each dimension must be stored in the symbol table and connected to the array name, while processing declarations
- C allows assignment of array slices. Therefore, size and type of slices must be checked during semantic analysis of assignments
- `int a[10][20], b[20], c[10][10];`
  `a[5] = b; c[7] = a[8];`
  In the above code fragment, the first assignment is valid, but the second one is not
- The above is called *structure equivalence* and it is different from *name equivalance*

Semantic Analysis of Structs

- Names inside structs belong to a higher level
- Equivalance of structs is based on *name equivalance* and not on *structure equivalence*
- `struct {int a,b; float c[10]; char d} x,y;`
  `struct {char d; float c[10]; int a,b} a,b;`
  `x = y; a = x;`
- In the code fragment above
  - In the second struct, the fields `a, b` of the struct are different from the struct variables *a* and *b*
  - The assignment `x = y;` is valid but `a = x;` is not valid, even though both structs have the same fields (but permuted)
- For a `struct` variable, an extra pointer pointing to the fields of the struct variable, along with their levels, can be maintained in the symbol table

Operator Overloading

- Operators such as '+' are usually overloaded in most languages
    - For example, the same symbol '+' is used with integers and reals
    - Programmers can define new functions for the existing operators in C++
    - This is operator overloading
    - Examples are defining '+' on complex numbers, rational numbers, or *time*

```
Complex operator+(const Complex& lhs,
                  const Complex& rhs)
{   Complex temp = lhs;
    temp.real += rhs.real;
    temp.imaginary += rhs.imaginary;
    return temp;
}
```

Function Overloading

- C++ also allows function overloading
- Overloaded functions with the same name (or same operator)
  - return results with different *types*, or
  - have different number of parameters, or
  - differ in parameter types
- The meaning of overloaded operators (in C++) with built-in types as parameters cannot be redefined
  - E.g., '+' on integers cannot be overloaded
  - Further, overloaded '+' must have exactly two operands
- Both operator and function overloading are resolved at compile time
- Either of them is different from *virtual functions* or *function overriding*

Function Overloading Example

```
// area of a square
int area(int s) { return s*s; }

// area of a rectangle
int area(int l, int b) { return l*b; }

// area of a circle
float area(float radius)
{ return 3.1416*radius*radius; }

int main()
{   std::cout << area(10);
    std::cout << area(12, 8);
    std::cout << area(2.5);
}
```

Implementing Operator Overloading

- A list of operator functions along with their parameter types is needed
- This list may be stored in a hash table, with the hash function designed to take the operator and its parameter types into account
- While handling a production such as $E \rightarrow E_1 + E_2$, the above hash table is searched with the signature $+(E_1.type, E_2.type)$
- If there is only one exact match (with the same operand types), then the overloading is resolved in favor of the match
- In case there is more than one exact match, an error is flagged
- The situation gets rather complicated in C++, due to possible conversions of operand types (char to int, int to float, etc.)

Implementing Function Overloading

- The symbol table should store multiple instances of the same function name along with their parameter types (and other information)
- While resolving a function call such as, $test(a, b, c)$, all the overloaded functions with the name *test* are collected and the closest possible match is chosen
  - Suppose the parameters a, b, c are all of `int` type
  - And the available overloaded functions are: `int test(int a, int b, float c)` and `int test(float a, int b, float c)`
  - In this case, we may choose the first one because it entails only one conversion from `int` to `float` (faster)
- If there is no match (or more than one match) even after conversions, an error is flagged

SATG for 2-pass Sem. Analysis of Func. and Calls

- $FUNC\_DECL \rightarrow FUNC\_HEAD \; \{ \; VAR\_DECL \;\; BODY \}$
  $BODY \rightarrow \{ \; VAR\_DECL \;\; STMT\_LIST \; \}$
  - Variable declarations appear stricty before their use
- $FUNC\_DECL \rightarrow$
  $FUNC\_HEAD \; \{ \; VAR\_DECL \;\; BODY \;\; VAR\_DECL \; \}$
  $BODY \rightarrow \{ \; VAR\_DECL \;\; STMT\_LIST \;\; VAR\_DECL \; \}$
  - permits variable declarations before *and after their use*
- Semantic analysis in this case requires two passes
  - Symbol table is constructed in the 1st pass
  - Declarations are all processed in the 1st pass
  - 1st pass can be integrated with LR-parsing during which a parse tree is built
  - Statements are analyzed in the 2nd pass
  - Sem. errors in statements are reported only in the 2nd pass
  - This effectively presents all the variable declarations before their use
  - 2nd pass can be made over the parse tree

Symbol Table for a 2-pass Semantic Analyzer

block_table (indexed by blk.num)

blk. num	name	result type	param. list ptr	local var. list ptr	num. param	surr. blk. num
1						
2						
3						
4						

Parameter/Variable name record

name	type	parameter or variable tag	level of declaration	blk.num

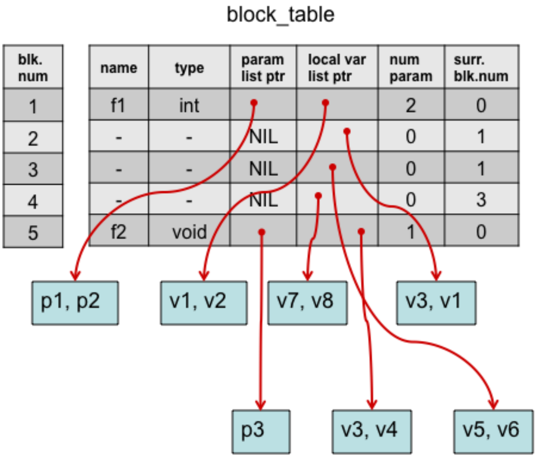
- The symbol table has to be *persistent*
- Cannot be destroyed after the block/function is processed in pass 1
- Should be stored in a form that can be accessed according to levels in pass 2

Symbol Table for a 2-pass Semantic Analyzer(contd.)

- The symbol table(ST) is indexed by block number
- In the previous version of the ST, there were no separate entries for blocks
- The surrounder block number (*surr.blk.num*) is the block number of the enclosing block
- All the blocks below a function entry *f* in the ST, upto the next function entry, belong to the function *f*
- To get the name of the parent function for a given block *b*, we go up table using surrounder block numbers until the surrounder block number becomes zero

Symbol Table for a 2-pass Semantic Analyzer(contd.)

```
1  int  f1( float p1, int p2){
       float v1; int v2;
      ...
2     { int v3; float v1;
          ....
      }
      ...
3     { float v5,v6;
          ....
4         { char v7,v8;
              ...
          }
      }
   } /* end of f1 */
5  void f2 (char p3){
       int v3; float v4;
     ...
   }
```



block_table

blk. num	name	type	param list ptr	local var list ptr	num param	surr. blk.num
1	f1	int			2	0
2	-	-	NIL		0	1
3	-	-	NIL		0	1
4	-	-	NIL		0	3
5	f2	void			1	0

Symbol Table for a 2-pass Semantic Analyzer(contd.)

- Block numbers begin from 1, and a counter *last_blk_num* generates new block numbers by incrementing itself
- *curr_blk_num* is the currently open block
- While opening a new block, *curr_blk_num* becomes its surrounder block number
- Similarly, while closing a block, its *surr.blk.num* is copied into *curr_blk_num*

Symbol Table for a 2-pass Semantic Analyzer(contd.)

- Apart from *active_func_ptr*, and *call_name_ptr*, we also need an *active_blk_ptr*
- *level* remains the same (nesting level of the current block)
- $search\_func(n, found, fnptr)$ remains the same, except that it searches entries corresponding to functions only (with $surr.blk.num = 0$)
- $search\_param(p, fnptr, found, pnptr)$ remains the same
- $search\_var(v, fnptr, l, found, vnptr)$ is similar to the old one, but the method of searching is now different
    - The variables of each block are stored separately under different block numbers
    - The parameter *level* is now replaced by *active_blk_ptr*
    - The search starts from *active_blk_ptr* and proceeds upwards using surrounder block numbers until the enclosing function is reached (with $surr.blk.num = 0$)

Intermediate Code Generation - Part 1

Y.N. Srikant

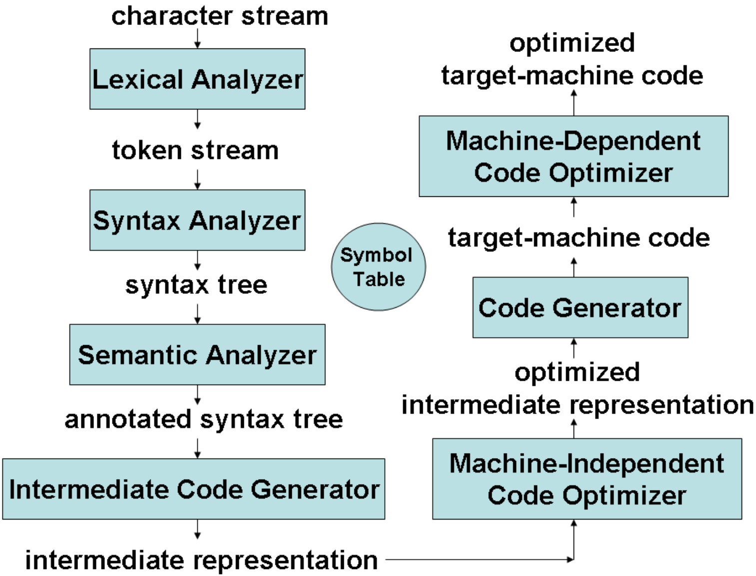
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction
- Different types of intermediate code
- Intermediate code generation for various constructs

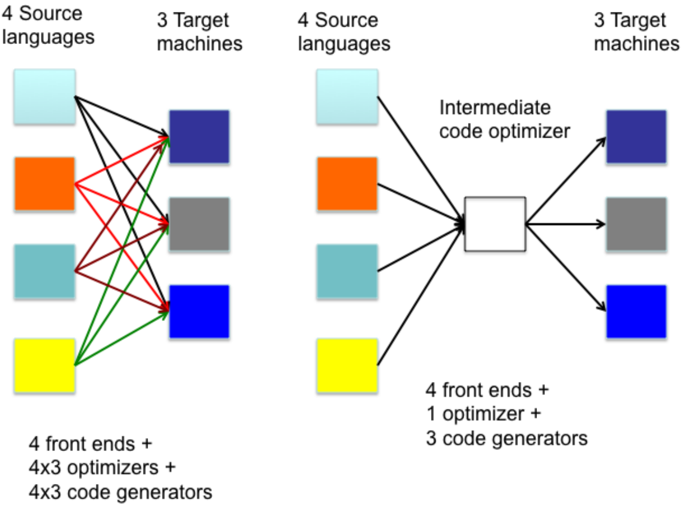
Compiler Overview



Compilers and Interpreters

- Compilers generate machine code, whereas interpreters interpret intermediate code
- Interpreters are easier to write and can provide better error messages (symbol table is still available)
- Interpreters are at least 5 times slower than machine code generated by compilers
- Interpreters also require much more memory than machine code generated by compilers
- Examples: Perl, Python, Unix Shell, Java, BASIC, LISP

Why Intermediate Code? - 1



4 Source languages

3 Target machines

4 front ends +
4x3 optimizers +
4x3 code generators

4 Source languages

Intermediate code optimizer

3 Target machines

4 front ends +
1 optimizer +
3 code generators

Why Intermediate Code? - 2

- While generating machine code directly from source code is possible, it entails two problems
    - With *m* languages and *n* target machines, we need to write *m* front ends, $m \times n$ optimizers, and $m \times n$ code generators
    - The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused
- By converting source code to an intermediate code, a machine-independent code optimizer may be written
- This means just *m* front ends, *n* code generators and 1 optimizer

Different Types of Intermediate Code

- Intermediate code must be easy to produce and easy to translate to machine code
  - A sort of universal assembly language
  - Should not contain any machine-specific parameters (registers, addresses, etc.)
- The type of intermediate code deployed is based on the application
- Quadruples, triples, indirect triples, abstract syntax trees are the classical forms used for machine-independent optimizations and machine code generation
- Static Single Assignment form (SSA) is a recent form and enables more effective optimizations
  - Conditional constant propagation and global value numbering are more effective on SSA
- Program Dependence Graph (PDG) is useful in automatic parallelization, instruction scheduling, and software pipelining

Three-Address Code

- Instructions are very simple
- Examples: `a = b + c`, `x = -y`, `if a > b goto L1`
- LHS is the target and the RHS has at most two sources and one operator
- RHS sources can be either variables or constants
- Three-address code is a generic form and can be implemented as quadruples, triples, indirect triples, tree or DAG
- Example: The three-address code for `a+b*c-d/(b*c)` is below
  1. t1 = b*c
  2. t2 = a+t1
  3. t3 = b*c
  4. t4 = d/t3
  5. t5 = t2-t4

Implementations of 3-Address Code

3-address code

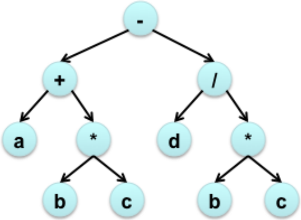
1 t1 = b*c
2 t2 = a+t1
3 t3 = b*c
4 t4 = d/t3
5 t5 = t2-t4

Quadruples

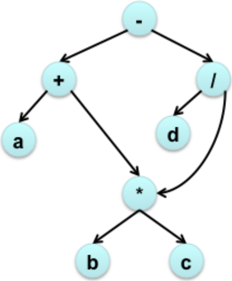
op	arg_1	arg_2	result
*	b	c	t1
+	a	t1	t2
*	b	c	t3
/	d	t3	t4
-	t2	t4	t5

Triples

	op	arg_1	arg_2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)



Syntax tree



DAG

Instructions in Three-Address Code - 1

1. *Assignment instructions*:
   `a = b biop c`, `a = uop b`, and `a = b` (copy), where
   - *biop* is any binary arithmetic, logical, or relational operator
   - *uop* is any unary arithmetic (-, shift, conversion) or logical operator ($\sim$)
   - Conversion operators are useful for converting integers to floating point numbers, etc.
2. *Jump instructions*:
   `goto L` (unconditional jump to L),
   `if t goto L` (it *t* is *true* then jump to L),
   `if a relop b goto L` (jump to L if *a* relop *b* is *true*),
   where
   - *L* is the label of the next three-address instruction to be executed
   - *t* is a boolean variable
   - *a* and *b* are either variables or constants

Instructions in Three-Address Code - 2

3. *Functions*:
   `func begin <name>` (beginning of the function),
   `func end` (end of a function),
   `param p` (place a value parameter *p* on stack),
   `refparam p` (place a reference parameter *p* on stack),
   `call f, n` (call a function *f* with *n* parameters),
   `return` (return from a function),
   `return a` (return from a function with a value *a*)
4. *Indexed copy instructions*:
   `a = b[i]` (*a* is set to contents(contents(*b*)+contents(*i*)), where *b* is (usually) the base address of an array
   `a[i] = b` (i^{th} location of array *a* is set to *b*)
5. *Pointer assignments*:
   `a = &b` (*a* is set to the address of *b*, i.e., *a* points to *b*)
   `*a = b` (contents(contents(*a*)) is set to contents(b))
   `a = *b` (*a* is set to contents(contents(*b*)))

Intermediate Code - Example 1

C-Program

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
    dot_prod = 0;         |    T6 = T4[T5]
    i = 0;                |    T7 = T3*T6
L1: if(i >= 10)goto L2    |    T8 = dot_prod+T7
    T1 = addr(a)          |    dot_prod = T8
    T2 = i*4              |    T9 = i+1
    T3 = T1[T2]           |    i = T9
    T4 = addr(b)          |    goto L1
    T5 = i*4              |L2:
```

Intermediate Code - Example 2

C-Program

```
int a[10], b[10], dot_prod, i; int* a1; int* b1;
dot_prod = 0; a1 = a; b1 = b;
for (i=0; i<10; i++) dot_prod += *a1++ * *b1++;
```

Intermediate code

```
    dot_prod = 0;          |    b1 = T6
    a1 = &a                |    T7 = T3*T5
    b1 = &b                |    T8 = dot_prod+T7
    i = 0                  |    dot_prod = T8
L1: if(i>=10)goto L2       |    T9 = i+1
    T3 = *a1               |    i = T9
    T4 = a1+1              |    goto L1
    a1 = T4                |L2:
    T5 = *b1
    T6 = b1+1
```

Intermediate Code - Example 3

C-Program (function)

```
int dot_prod(int x[], int y[]){
  int d, i; d = 0;
  for (i=0; i<10; i++) d += x[i]*y[i];
  return d;
}
```

Intermediate code

```
    func begin dot_prod |    T6 = T4[T5]
    d = 0;              |    T7 = T3*T6
    i = 0;              |    T8 = d+T7
L1: if(i >= 10)goto L2  |    d = T8
    T1 = addr(x)        |    T9 = i+1
    T2 = i*4            |    i = T9
    T3 = T1[T2]         |    goto L1
    T4 = addr(y)        |L2: return d
    T5 = i*4            |    func end
```

Intermediate Code - Example 3 (contd.)

C-Program (main)

```
main(){
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

Intermediate code

```
    func begin main
    refparam a
    refparam b
    refparam result
    call dot_prod, 3
    p = result
    func end
```

Intermediate Code - Example 4

C-Program (function)

```
int fact(int n){
   if (n==0) return 1;
   else return (n*fact(n-1));
}
```

Intermediate code

```
func begin fact     |       T3 = n*result
if (n==0) goto L1   |       return T3
T1 = n-1            |   L1: return 1
param T1            |       func end
refparam result     |
call fact, 2        |
```

Code Templates for *If-Then-Else* Statement

Assumption: No short-circuit evaluation for E (i.e., no jumps within the intermediate code for E)

```
If (E) S1 else S2
      code for E (result in T)
      if T≤ 0 goto L1 /* if T is false, jump to else part */
      code for S1 /* all exits from within S1 also jump to L2 */
      goto L2 /* jump to exit */
L1:   code for S2 /* all exits from within S2 also jump to L2 */
L2:   /* exit */

If (E) S
      code for E (result in T)
      if T≤ 0 goto L1 /* if T is false, jump to exit */
      code for S /* all exits from within S also jump to L1 */
L1:   /* exit */
```

Code Template for *While-do* Statement

Assumption: No short-circuit evaluation for E (i.e., no jumps within the intermediate code for E)

```
while  (E)  do  S
L1:       code for E (result in T)
          if  T≤ 0  goto  L2 /* if T is false, jump to exit */
          code for S /* all exits from within S also jump to L1 */
          goto L1 /* loop back */
L2:       /* exit */
```

Translations for *If-Then-Else* Statement

Let us see the code generated for the following code fragment.
A_i are all assignments, and E_i are all expressions
if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

```
1           code for E1 /* result in T1 */
10          if (T1 <= 0), goto L1 (61)
            /* if T1 is false jump to else part */
11          code for E2 /* result in T2 */
35          if (T2 <= 0), goto L2 (43)
            /* if T2 is false jump to else part */
36          code for A1
42          goto L3 (82)
43    L2:   code for A2
60          goto L3 (82)
61    L1:   code for A3
82    L3:   code for A4
```

Translations for *while-do* Statement

Code fragment:

while (E_1) do {if (E_2) then A_1; else A_2;} A_3;

```
1    L1:   code for E1 /* result in T1 */
15         if (T1 <= 0), goto L2 (79)
           /* if T1 is false jump to loop exit */
16         code for E2 /* result in T2 */
30         if (T2 <= 0), goto L3 (55)
           /* if T2 is false jump to else part */
31         code for A1
54         goto L1 (1)/* loop back */
55   L3:   code for A2
78         goto L1 (1)/* loop back */
79   L2:   code for A3
```

SATG - Attributes

- S.next, N.next: list of quads indicating where to jump; target of jump is still undefined
- IFEXP.falselist: quad indicating where to jump if the expression is false; target of jump is still undefined
- E.result: pointer to symbol table entry
  - All temporaries generated during intermediate code generation are inserted into the symbol table
  - In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
  - However, textual examples will use names

SATG - Auxiliary functions/variables

- nextquad: global variable containing the number of the next quadruple to be generated
- backpatch(list, quad_number): patches target of all ‘goto’ quads on the ‘list’ to ‘quad_number’
- merge(list-1, list-2,...,list-n): merges all the lists supplied as parameters
- gen(‘quadruple’): generates ‘quadruple’ at position ‘nextquad’ and increments ‘nextquad’
    - In quadruple/triple/tree representation, pointers to symbol table entries for variables and temporaries are used in place of names
    - However, textual examples will use names
- newtemp(temp-type): generates a temporary name of type *temp-type*, inserts it into the symbol table, and returns the pointer to that entry in the symbol table

SATG for *If-Then-Else* Statement

- $IFEXP \rightarrow if \;\; E$
  { IFEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __'); }
- $S \rightarrow IFEXP \;\; S_1; \;\; N \;\; else \;\; M \;\; S_2$
  { backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S_1.next, S_2.next, N.next); }
- $S \rightarrow IFEXP \;\; S_1;$
  { S.next := merge(S_1.next, IFEXP.falselist); }
- $N \rightarrow \epsilon$
  { N.next := makelist(nextquad);
  gen('goto __'); }
- $M \rightarrow \epsilon$
  { M.quad := nextquad; }

Intermediate Code Generation - Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in part 1)
- Different types of intermediate code (covered in part 1)
- Intermediate code generation for various constructs

SATG for *If-Then-Else* Statement

- $IFEXP \rightarrow if\ \ E$
  { IFEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __'); }
- $S \rightarrow IFEXP\ \ S_1;\ \ N\ \ else\ \ M\ \ S_2$
  { backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S_1.next, S_2.next, N.next); }
- $S \rightarrow IFEXP\ \ S_1;$
  { S.next := merge(S_1.next, IFEXP.falselist); }
- $N \rightarrow \epsilon$
  { N.next := makelist(nextquad);
  gen('goto __'); }
- $M \rightarrow \epsilon$
  { M.quad := nextquad; }

SATG for Other Statements

- *$S \rightarrow$ '{' L '}'*
  { S.next := L.next; }
- *$S \rightarrow A$*
  { S.next := makelist(nil); }
- *$S \rightarrow$ return E*
  { gen('return E.result'); S.next := makelist(nil); }
- *$L \rightarrow L_1$ ';' M S*
  { backpatch(L_1.next, M.quad);
  L.next := S.next; }
- *$L \rightarrow S$*
  { L.next := S.next; }
- When the body of a procedure ends, we perform the following actions in addition to other actions:
  { backpatch(S.next, nextquad); gen('func end'); }

Translation Trace for *If-Then-Else* Statement

A_i are all assignments, and E_i are all expressions

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow IFEXP \;\; S_1; \;\; N_1 \;\; else \;\; M_1 \;\; S_2$

$\Rightarrow^* IFEXP_1 \;\; IFEXP_2 \;\; S_{21}; \;\; N_2 \;\; else \;\; M_2 \;\; S_{22}; \;\; N_1 \; else \;\; M_1 \;\; S_2$

1. Consider outer if-then-else
   Code generation for E_1
2. gen('if E_1.result $\leq$ 0 goto __')
   on reduction by $IFEXP_1 \rightarrow if \;\; E_1$
   Remember the above quad address in $IFEXP_1$.falselist
3. Consider inner if-then-else
   Code generation for E_2
4. gen('if E_2.result $\leq$ 0 goto __')
   on reduction by $IFEXP_2 \rightarrow if \;\; E_2$
   Remember the above quad address in $IFEXP_2$.falselist

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;
$S \Rightarrow^* \mathit{IFEXP}_1\ \mathit{IFEXP}_2\ S_{21};\ N_2\ \mathit{else}\ M_2\ S_{22};\ N_1\ \mathit{else}\ \ M_1\ S_2$
Code generated so far:

Code for E_1; if E_1.result $\leq$ 0 goto __ (on IFEXP_1.falselist);
Code for E_2; if E_2.result $\leq$ 0 goto __ (on IFEXP_2.falselist);

5. Code generation for S_{21}
6. gen('goto __'), on reduction by $N_2 \rightarrow \epsilon$
   (remember in N_2.next)
7. L1: remember in M_2.quad, on reduction by $M_2 \rightarrow \epsilon$
8. Code generation for S_{22}
9. backpatch(IFEXP_2.falselist, L1) (processing E_2 == false)
   on reduction by $S_1 \rightarrow \mathit{IFEXP}_2\ \ S_{21}\ \ N_2\ \ \mathit{else}\ \ M_2\ \ S_{22}$
   N_2.next is not yet patched; put on S_1.next

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow IFEXP\ \ S_1;\ \ N_1\ \ else\ \ M_1\ \ S_2$

$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$

Code generated so far:

Code for E_1; if E_1.result $\leq$ 0 goto __ (on $IFEXP_1$.falselist)

Code for E_2; if E_2.result $\leq$ 0 goto L1

Code for S_{21}; goto __ (on S_1.next)

L1: Code for S_{22}

10. gen('goto __'), on reduction by $N_1 \rightarrow \epsilon$ (remember in N_1.next)
11. L2: remember in M_1.quad, on reduction by $M_1 \rightarrow \epsilon$
12. Code generation for S_2
13. backpatch(IFEXP.falselist, L2) (processing E_1 == false) on reduction by $S \rightarrow IFEXP\ \ S_1\ \ N_1\ \ else\ \ M_1\ \ S_2$
    N_1.next is merged with S_1.next, and put on S.next

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;

$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$

$L \Rightarrow^* L_1$ ‘;’ $M_3\ \ S_4\ \ \Rightarrow^*\ S_3$ ‘;’ $M_3\ \ S_4$

Code generated so far (for S_3/L_1 above):

Code for E_1; if E_1.result $\leq$ 0 goto L2
Code for E_2; if E_2.result $\leq$ 0 goto L1
Code for S_{21}; goto __ (on S_3.next/L_1.next)
L1: Code for S_{22}
goto __ (on S_3.next/L_1.next)
L2: Code for S_2

14. L3: remember in M_3.quad, on reduction by $M_3 \rightarrow \epsilon$
15. Code generation for S_4
16. backpatch(L_1.next, L3), on reduction by $L \rightarrow L_1$ ‘;’ $M_3\ \ S_4$
17. L.next is empty

Translation Trace for *If-Then-Else* Statement(contd.)

if (E_1) { if (E_2) A_1; else A_2; }else A_3; A_4;
$S \Rightarrow^* IFEXP_1\ IFEXP_2\ S_{21};\ N_2\ else\ M_2\ S_{22};\ N_1\ else\ M_1\ S_2$
$L \Rightarrow^* L_1$ ‘;’ $M_3\ \ S_4\ \ \Rightarrow^* S_3$ ‘;’ $M_3\ \ S_4$

Final generated code

Code for E_1; if E_1.result $\leq$ 0 goto L2
Code for E_2; if E_2.result $\leq$ 0 goto L1
Code for S_{21}; goto L3
L1: Code for S_{22}
goto L3
L2: Code for S_2
L3: Code for S_4

SATG for *While-do* Statement

- *WHILEXEP* $\rightarrow$ *while M E*
  { WHILEEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __');
  WHILEEXP.begin := M.quad; }
- *S* $\rightarrow$ *WHILEXEP do* S_1
  { gen('goto WHILEEXP.begin');
  backpatch(S_1.next, WHILEEXP.begin);
  S.next := WHILEEXP.falselist; }
- *M* $\rightarrow \epsilon$ (repeated here for convenience)
  { M.quad := nextquad; }

Code Template for *Function* Declaration and Call

Assumtion: No nesting of functions

```
result foo(parameter list){ variable declarations; Statement list; }
func begin foo
/* creates activation record for foo - */
/* - space for local variables and temporaries */
code for Statement list
func end /* releases activation record and return */

x = bar(p1,p2,p3);
code for evaluation of p1, p2, p3 (result in T1, T2, T3)
/* result is supposed to be returned in T4 */
param T1;  param T2;  param T3; refparam T4;
call bar, 4
/* creates appropriate access links, pushes return address */
/* and jumps to code for bar */
x = T4
```

SATG for *Function* Call

Assumtion: No nesting of functions

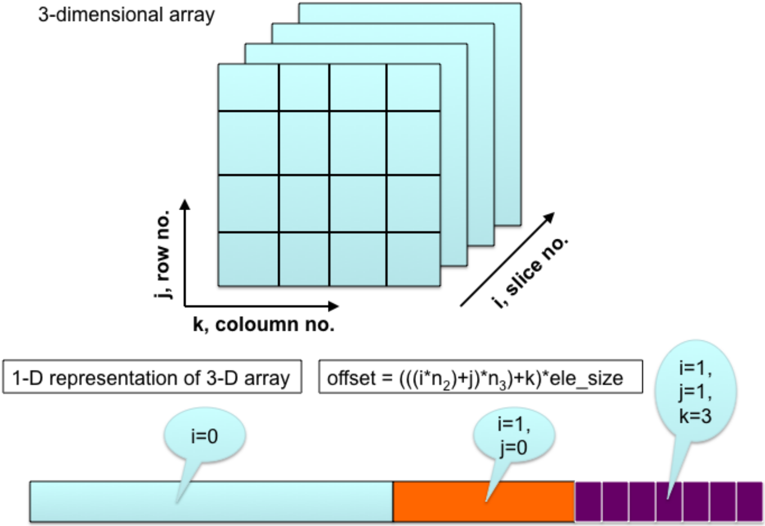
- *FUNC_CALL* → *id* **{action 1}** (*PARAMLIST*) **{action 2}**
  {action 1:} {search_func(id.name, found, fnptr);
  call_name_ptr := fnptr }
  {action 2:}
  { result_var := newtemp(get_result_type(call_name_ptr));
  gen('refparam result_var');
  /* Machine code for return a places a in result_var */
  gen('call call_name_ptr, PARAMLIST.pno+1'); }
- *PARAMLIST* → *PLIST* { PARAMLIST.pno := PLIST.pno }
- *PARAMLIST* → ϵ {PARAMLIST.pno := 0 }
- *PLIST* → *E* { PLIST.pno := 1; gen('param E.result'); }
- $PLIST_1$ → $PLIST_2$, *E*
  { $PLIST_1$.pno := $PLIST_2$.pno + 1; gen('param E.result'); }

SATG for *Function* Declaration

Assumtion: No nesting of functions

- *FUNC_DECL* → *FUNC_HEAD* { *VAR_DECL BODY* }
  { backpatch(BODY.next, nextquad);
  gen('func end');}
- *FUNC_HEAD* → *RESULT id* (*DECL_PLIST*)
  { search_func(id.name, found, namptr);
  active_func_ptr := namptr;
  gen('func begin active_func_ptr'); }

1-D Representation of 3-D Array



1-D representation of 3-D array

offset = $(((i*n_2)+j)*n_3)+k)*\text{ele\_size}$

Code Template for *Expressions and Assignments*

```
int a[10][20][35], b;
b = exp1;
code for evaluation of exp1 (result in T1)
b = T1
/* Assuming the array access to be, a[i][j][k] */
/* base address = addr(a), offset = (((i*n2)+j)*n3)+k)*ele_size */
a[exp2][exp3][exp4] = exp5;

10: code for exp2 (result in T2)  || 141: T8 = T7+T6
70: code for exp3 (result in T3)  || 142: T9 = T8*intsize
105: T4 = T2*20                   || 143: T10 = addr(a)
106: T5 = T4+T3                   || 144: code for exp5 (result in T11)
107: code for exp4 (result in T6)|| 186: T10[T9] = T11
140: T7 = T5*35
```

SATG for *Expressions and Assignments*

- $S \rightarrow L \ := \ E$
  /* L has two attributes, L.place, pointing to the name of the variable or temporary in the symbol table, and L.offset, pointing to the temporary holding the offset into the array (NULL in the case of a simple variable) */
  { if (L.offset == NULL) gen('L.place = E.result');
  else gen('L.place[L.offset] = E.result');}
- $E \rightarrow (\ E_1 \ )$ {E.result := E_1.result; }
- $E \rightarrow L$ { if (L.offset == NULL) E.result := L.place;
  else { E.result := newtemp(L.type);
  gen('E.result = L.place[L.offset]'); }
- $E \rightarrow num$ { E.result := newtemp(num.type);
  gen('E.result = num.value'); }

SATG for *Expressions and Assignments* (contd.)

- $E \rightarrow E_1 + E_2$

```
{ result_type := compatible_type($E_1$.type, $E_2$.type);
  E.result := newtemp(result_type);
  if ($E_1$.type == result_type) operand_1 := $E_1$.result;
  else if ($E_1$.type == integer && result_type == real)
        { operand_1 := newtemp(real);
          gen('operand_1 = cnvrt_float($E_1$.result); };
  if ($E_2$.type == result_type) operand_2 := $E_2$.result;
  else if ($E_2$.type == integer && result_type == real)
        { operand_2 := newtemp(real);
          gen('operand_2 = cnvrt_float($E_2$.result); };
  gen('E.result = operand_1 + operand_2');
}
```

SATG for *Expressions and Assignments* (contd.)

- $E \rightarrow E_1 || E_2$
  { E.result := newtemp(integer);
  gen('E.result = E_1.result || E_2.result');
- $E \rightarrow E_1 < E_2$
  { E.result := newtemp(integer);
  gen('E.result = 1');
  gen('if E_1.result < E_2.result goto nextquad+2');
  gen('E.result = 0');
  }
- $L \rightarrow id$ { search_var_param(id.name, active_func_ptr,
  level, found, vn); L.place := vn; L.offset := NULL; }

Note: *search_var_param*() searches for *id.name* in the variable list first, and if not found, in the parameter list next.

SATG for *Expressions and Assignments* (contd.)

- *ELIST* $\rightarrow$ *id* [*E*
  { search_var_param(id.name, active_func_ptr,
  level, found, vn); ELIST.dim := 1;
  ELIST.arrayptr := vn; ELIST.result := E.result; }
- *L* $\rightarrow$ *ELIST*] { L.place := ELIST.arrayptr;
  temp := newtemp(int); L.offset := temp;
  ele_size := ELIST.arrayptr -> ele_size;
  gen('temp = ELIST.result * ele_size'); }
- *ELIST* $\rightarrow$ *ELIST*$_1$, *E*
  { ELIST.dim := *ELIST*$_1$.dim + 1;
  ELIST.arrayptr := *ELIST*$_1$.arrayptr
  num_elem := get_dim(*ELIST*$_1$.arrayptr, *ELIST*$_1$.dim + 1);
  temp1 := newtemp(int); temp2 := newtemp(int);
  gen('temp1 = *ELIST*$_1$.result * num_elem');
  ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }

Short Circuit Evaluation for Boolean Expressions

- (exp1 && exp2): value = if ($\sim$exp1) then FALSE else exp2
  - This implies that exp2 need not be evaluated if exp1 is FALSE
- (exp1 || exp2):value = if (exp1) then TRUE else exp2
  - This implies that exp2 need not be evaluated if exp1 is TRUE
- Since boolean expressions are used mostly in conditional and loop statements, it is possible to realize perform short circuit evaluation of expressions using control flow constructs
- In such a case, there are no explicit '||' and '&&' operators in the intermediate code (as earlier), but only jumps
- Much faster, since complete expression is not evaluated
- If unevaluated expressions have side effects, then program may have non-deterministic behaviour

Control-Flow Realization of Boolean Expressions

```
if ((a+b < c+d) || ((e==f) && (g > h-k))) A1; else A2; A3;
```

```
100:         T1 = a+b
101:         T2 = c+d
103:         if T1 < T2 goto L1
104:         goto L2
105:L2:      if e==f goto L3
106:         goto L4
107:L3:      T3 = h-k
108:         if g > T3 goto L5
109:         goto L6
110:L1:L5:   code for A1
111:         goto L7
112:L4:L6:   code for A2
113:L7:      code for A3
```

Intermediate Code Generation - Part 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in part 1)
- Different types of intermediate code (covered in part 1)
- Intermediate code generation for various constructs

Short Circuit Evaluation for Boolean Expressions

- (exp1 && exp2): value = if ($\sim$exp1) then FALSE else exp2
  - This implies that exp2 need not be evaluated if exp1 is FALSE
- (exp1 || exp2):value = if (exp1) then TRUE else exp2
  - This implies that exp2 need not be evaluated if exp1 is TRUE
- Since boolean expressions are used mostly in conditional and loop statements, it is possible to realize perform short circuit evaluation of expressions using control flow constructs
- In such a case, there are no explicit '||' and '&&' operators in the intermediate code (as earlier), but only jumps
- Much faster, since complete expression is not evaluated
- If unevaluated expressions have side effects, then program may have non-deterministic behaviour

Control-Flow Realization of Boolean Expressions

```
if ((a+b < c+d) || ((e==f) && (g > h-k))) A1; else A2; A3;
```

```
100:          T1 = a+b
101:          T2 = c+d
103:          if T1 < T2 goto L1
104:          goto L2
105:L2:       if e==f goto L3
106:          goto L4
107:L3:       T3 = h-k
108:          if g > T3 goto L5
109:          goto L6
110:L1:L5:    code for A1
111:          goto L7
112:L4:L6:    code for A2
113:L7:       code for A3
```

SATG for *Control-Flow Realization of Boolean Expressions*

- $E \rightarrow E_1 \; || \; M \; E_2$ { backpatch(E_1.falselist, M.quad);
  E.truelist := merge(E_1.truelist, E_2.truelist);
  E.falselist := E_2.falselist }
- $E \rightarrow E_1 \; \&\& \; M \; E_2$ { backpatch(E_1.truelist, M.quad);
  E.falselist := merge(E_1.falselist, E_2.falselist);
  E.truelist := E_2.truelist }
- $E \rightarrow \sim E_1$ { E.truelist := E_1.falselist;
  E.falselist := E_1.truelist }
- $M \rightarrow \epsilon$ {M.quad := nextquad; }
- $E \rightarrow E_1 \; < \; E_2$ { E.truelist := makelist(nextquad);
  E.falselist := makelist(nextquad+1);
  gen('if E_1.result **<** E_2.result goto __');
  gen('goto __'); }

SATG for *Control-Flow Realization of Boolean Expressions*

- *$E \rightarrow (E_1)$*
  { E.truelist := E_1.truelist; E.falselist := E_1.falselist }
- *$E \rightarrow$ true* { E.truelist := makelist(nextquad); gen('goto __');}
- *$E \rightarrow$ false*
  { E.falselist := makelist(nextquad); gen('goto __');}
- *$S \rightarrow$ IFEXP S_1 N else M S_2*
  { backpatch(IFEXP.falselist, M.quad);
  S.next := merge(S_1.next, S_2.next, N.next); }
- *$S \rightarrow$ IFEXP S_1*
  { S.next := merge(S_1.next, IFEXP.falselist); }
- *IFEXP $\rightarrow$ if E* { backpatch(E.truelist, nextquad);
  IFEXP.falselist := E.falselist;}
- *$N \rightarrow \epsilon$* { N.next := makelist(nextquad); gen('goto __'); }

SATG for *Control-Flow Realization of Boolean Expressions*

- $S \rightarrow WHILEXEP\ \ do\ \ S_1$
  { gen('goto WHILEEXP.begin');
  backpatch(S_1.next, WHILEEXP.begin);
  S.next := WHILEEXP.falselist; }
- $WHILEXEP \rightarrow while\ \ M\ \ E$
  { WHILEEXP.falselist := E.falselist;
  backpatch(E.truelist, nextquad);
  WHILEEXP.begin := M.quad; }
- $M \rightarrow \epsilon$ (repeated here for convenience)
  { M.quad := nextquad; }

Code Template for *Switch* Statement

```
switch (exp) {
    case $l_1$ : $SL_1$
    case $l_{2_1}$ : case $l_{2_2}$ : $SL_2$
    ...
    case $l_{n-1}$ : $SL_{n-1}$
    default: $SL_n$
}
```

This code template can be used for switch statements with 10-15 cases. Note that statement list SL_i must incorporate a ‘break’ statement, if necessary

```
           code for exp (result in T)
           goto TEST
$L_1$:     code for $SL_1$
$L_2$:     code for $SL_2$
           ...
$L_n$:     code for $SL_n$
           goto NEXT
TEST:      if T==$l_1$ goto $L_1$
           if T==$l_{2_1}$ goto $L_2$
           if T==$l_{2_2}$ goto $L_2$
           ...
           if T==$l_{n-1}$ goto $L_{n-1}$
           if default_yes goto $L_n$
NEXT:
```

Grammar for Switch Statement

The grammar for the ‘switch’ statement according to ANSI standard C is:

selection_statement → SWITCH ‘(’ expression ‘)’ statement

However, a more intuitive form of the grammar is shown below

- *STMT* → *SWITCH_HEAD   SWITCH_BODY*
- *SWITCH_HEAD* → *switch* (*E*)/* E must be `int` type */
- *SWITCH_BODY* → { *CASE_LIST* }
- *CASE_LIST* → *CASE_ST* | *CASE_LIST   CASE_ST*
- *CASE_ST* → *CASE_LABELS   STMT_LIST* ;
- *CASE_LABELS* → ϵ | *CASE_LABELS   CASE_LABEL*
- *CASE_LABEL* → *case   CONST_INTEXPR* : | *default* : /* CONST_INTEXPR must be of `int` or `char` type */
- *STMT* → *break* /* also an option */

SATG for *Switch* Statement

- *SWITCH_HEAD* → *switch* (*E*)
  { SWITCH_HEAD.result := E.result;
  SWITCH_HEAD.test := nextquad;
  gen('goto __'); }
- *STMT* → *break*
  { STMT.next := makelist(nextquad);
  gen('goto __'); }
- *CASE_LABEL* → *case CONST_INTEXPR* :
  { CASE_LABEL.val := CONST_INTEXPR.val;
  CASE_LABEL.default := false; }
- *CASE_LABEL* → *default* : {CASE_LABEL.default := true; }
- *CASE_LABELS* → ϵ { CASE_LABELS.default := false;
  { CASE_LABELS.list := makelist(NULL); }

SATG for *Switch* Statement (contd.)

- *CASE_LABELS* → *CASE_LABELS*$_1$ *CASE_LABEL*
  { if (∼CASE_LABEL.default) CASE_LABELS.list :=
  append(*CASE_LABELS*$_1$.list, CASE_LABEL.val);
  else CASE_LABELS.list := *CASE_LABELS*$_1$.list;
  if (*CASE_LABELS*$_1$.default || CASE_LABEL.default)
  CASE_LABEL.default := true; }
- *CASE_ST* → *CASE_LABELS M STMT_LIST ;*
  { CASE_ST.next := STMT_LIST.next; CASE_ST.list :=
  add_jump_target(CASE_LABELS.list, M.quad);
  if (CASE_LABELS.default) CASE_ST.default := M.quad;
  else CASE_ST.default := -1; }
- *CASE_LIST* → *CASE_ST*
  { CASE_LIST.next := CASE_ST.next;
  CASE_LIST.list := CASE_ST.list;
  CASE_LIST.default := CASE_ST.default; }

Code Template for *Switch* Statement

```
switch (exp) {
    case $l_1$ : $SL_1$
    case $l_{2_1}$: case $l_{2_2}$ : $SL_2$
    ...
    case $l_{n-1}$ : $SL_{n-1}$
    default: $SL_n$
}
```

This code template can be used for switch statements with 10-15 cases. Note that statement list SL_i must incorporate a ‘break’ statement, if necessary

```
            code for exp (result in T)
            goto TEST
$L_1$:      code for $SL_1$
$L_2$:      code for $SL_2$
            ...
$L_n$:      code for $SL_n$
            goto NEXT
TEST:       if T==$l_1$ goto $L_1$
            if T==$l_{2_1}$ goto $L_2$
            if T==$l_{2_2}$ goto $L_2$
            ...
            if T==$l_{n-1}$ goto $L_{n-1}$
            if default_yes goto $L_n$
NEXT:
```

SATG for *Switch* Statement (contd.)

- $CASE\_LIST \rightarrow CASE\_LIST_1 \quad CASE\_ST$
  { CASE_LIST.next :=
        merge($CASE\_LIST_1$.next, CASE_ST.next);
  CASE_LIST.list :=
        merge($CASE\_LIST_1$.list, CASE_ST.list);
  CASE_LIST.default := $CASE\_LIST_1$.default == -1 ?
    CASE_ST.default : $CASE\_LIST_1$.default; }
- $SWITCH\_BODY \rightarrow \{ \ CASE\_LIST \ \}$
  { SWITCH_BODY.next :=
      merge(CASE_LIST.next, makelist(nextquad));
  gen('goto __');
  SWITCH_BODY.list := CASE_LIST.list;
  SWITCH_BODY.default := CASE_LIST.default; }

SATG for *Switch* Statement (contd.)

- *STMT → SWITCH_HEAD SWITCH_BODY*

```
{ backpatch(SWITCH_HEAD.test, nextquad);
  for each (value, jump) pair in SWITCH_BODY.list do {
    (v,j) := next (value, jump) pair from SWITCH_BODY.list;
    gen('if SWITCH_HEAD.result == v goto j');
  }
  if (SWITCH_BODY.default != -1)
       gen('goto SWITCH_BODY.default');
  STMT.next := SWITCH_BODY.next;
}
```

C For-Loop

The for-loop of C is very general

- for (*expression*$_1$; *expression*$_2$; *expression*$_3$) *statement*
  This statement is equivalent to
  expression$_1$;
  while (*expression*$_2$) { *statement* *expression*$_3$; }
- All three expressions are optional and any one (or all) may be missing
- Code generation is non-trivial because the order of execution of *statement* and *expression*$_3$ are reversed compared to their occurrance in the for-statement
- Difficulty is due to 1-pass bottom-up code generation
- Code generation during parse tree traversals mitigates this problem by generating code for *expression*$_3$ before that of *statement*

Code Generation Template for *C For-Loop*

```
for ( $E_1$; $E_2$; $E_3$ ) $S$
        code for $E_1$
L1:     code for $E_2$ (result in T)
        goto L4
L2:     code for $E_3$
        goto L1
L3:     code for $S$ /* all jumps out of S goto L2 */
        goto L2
L4:     if  T == 0  goto  L5 /* if T is zero, jump to exit */
        goto L3
L5:     /* exit */
```

Code Generation for C For-Loop

- $STMT \rightarrow for\ (\ E_1;\ M\ E_2;\ N\ E_3\ )\ P\ STMT_1$
  { gen('goto N.quad+1'); Q1 := nextquad;
   gen('if E_2.result == 0 goto __');
   gen('goto P.quad+1');
   backpatch(N.quad, Q1);
   backpatch($STMT_1$.next, N.quad+1);
   backpatch(P.quad, M.quad);
   STMT.next := makelist(Q1); }
- $M \rightarrow \epsilon$ { M.quad := nextquad; }
- $N \rightarrow \epsilon$ { N.quad := nextquad; gen('goto __'); }
- $P \rightarrow \epsilon$ { P.quad := nextquad; gen('goto __'); }

ALGOL For-Loop

- Let us also consider a more restricted form of the for-loop
  - $STMT \rightarrow for\ id = EXP_1\ to\ EXP_2\ by\ EXP_3\ do\ STMT_1$ where, EXP_1, EXP_2, and EXP_3 are all arithmetic expressions, indicating starting, ending and increment values of the iteration index
  - EXP_3 may have either positive or negative values
  - All three expressions are evaluated before the iterations begin and are stored. They are not evaluated again during the loop-run
  - All three expressions are mandatory (unlike in the C-for-loop)

Code Generation Template for ALGOL For-Loop

$STMT \rightarrow$ for id $=$ EXP_1 to EXP_2 by EXP_3 do $STMT_1$

```
        Code for $EXP_1$ (result in T1)
        Code for $EXP_2$ (result in T2)
        Code for $EXP_3$ (result in T3)
        goto L1
L0:     Code for $STMT_1$
        id = id + T3
        goto L2
L1:     id = T1
L2:     if (T3 $\leq$ 0) goto L3
        if (id > T2) goto L4 /* positive increment */
        goto L0
L3:     if (id < T2) goto L4 /* negative increment */
        goto L0
L4:
```

Code Generation for ALGOL For-Loop

$M \rightarrow \epsilon$ { M.quad := nextquad; gen('goto __'); }

$STMT \rightarrow for\ id = EXP_1\ to\ EXP_2\ by\ EXP_3\ M\ do\ STMT_1$
{ search(id.name, idptr); gen('idptr = idptr + EXP_3.result');
Q1 := nextquad; gen('goto __'); backpatch(M.quad, nextquad);
gen('idptr = EXP_1.result'); backpatch(Q1, nextquad);
Q2 := nextquad; gen('if EXP_3.result $\leq$ 0 goto __');
gen('if idptr > EXP_2.result goto __');
gen('goto M.quad+1'); backpatch(Q2, nextquad);
Q3 := nextquad; gen('if idptr < EXP_2.result goto __');
gen('goto M.quad+1');
STMT.next :=
merge(makelist(Q2+1), makelist(Q3), $STMT_1$.next);

Another Code Generation Template for ALGOL For-Loop

$STMT \rightarrow for\ id = EXP_1\ to\ EXP_2\ by\ EXP_3\ do\ STMT_1$

```
      Code for $EXP_1$ (result in T1)
      Code for $EXP_2$ (result in T2)
      Code for $EXP_3$ (result in T3)
      id = T1
L1:   if (T3 $\leq$ 0) goto L2
      if (id > T2) goto L4 /* positive increment */
      goto L3
L2:   if (id < T2) goto L4 /* negative increment */
L3:   Code for STMT
      id = id + T3
      goto L1
L4:
```

Code generation using this template is left as an exercise

Run-Time Array Range Checking

`int b[10][20]; a = b[`exp_1`][`exp_2`];`

The code generated for this assignment with run-time array range checking is as below:

```
    code for $exp_1$ /* result in T1 */
    if T1 < 10 goto L1
    'error: array overflow in dimension 1'
    T1 = 9 /* max value for dim 1 */
L1: code for $exp_2$ /* result in T2 */
    if T2 < 20 goto L2
    'error: array overflow in dimension 2'
    T2 = 19 /* max value for dim 2 */
L2: T3 = T1*20
    T4 = T3+T2
    T5 = T4*intsize
    T6 = addr(b)
    a = T6[T5]
```

Code Generation with Array Range Checking

- $S \rightarrow L := E$
  { if (L.offset == NULL) gen(‘L.place = E.result’);
  else gen(‘L.place[L.offset] = E.result’);}
- $E \rightarrow L$ { if (L.offset == NULL) E.result := L.place;
  else { E.result := newtemp(L.type);
  gen(‘E.result = L.place[L.offset]’); }
- $ELIST \rightarrow id\ [\ E$ { search_var(id.name, active_func_ptr,
  level, found, vn); ELIST.arrayptr := vn;
  ELIST.result := E.result; ELIST.dim := 1;
  num_elem := get_dim(vn, 1); Q1 := nextquad;
  gen(‘if E.result < num_elem goto Q1+3’);
  gen(‘error(“array overflow in dimension 1”)’);
  gen(‘E.result = num_elem-1’);}

Code Generation with Array Range Checking(contd.)

- $L \rightarrow ELIST$] { L.place := ELIST.arrayptr;
  temp := newtemp(int); L.offset := temp;
  ele_size := ELIST.arrayptr -> ele_size;
  gen('temp = ELIST.result * ele_size'); }
- $ELIST \rightarrow ELIST_1$, E
  { ELIST.dim := $ELIST_1$.dim + 1;
  ELIST.arrayptr := $ELIST_1$.arrayptr
  num_elem := get_dim($ELIST_1$.arrayptr, $ELIST_1$.dim + 1);
  Q1 := nextquad;
  gen('if E.result < num_elem goto Q1+3');
  gen('error("array overflow in ($ELIST_1$.dim + 1)")');
  gen('E.result = num_elem-1');
  temp1 := newtemp(int); temp2 := newtemp(int);
  gen('temp1 = $ELIST_1$.result * num_elem');
  ELIST.result := temp2; gen('temp2 = temp1 + E.result'); }

Intermediate Code Generation - Part 4

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in part 1)
- Different types of intermediate code (covered in part 1)
- Intermediate code generation for various constructs

break and *continue* Statements

- `break` statements can occur only within `while, for, do-while` and `switch` statements
- `continue` statements can occur only within `while, for,` and `do-while` statements (i.e., only loops)
- All other occurrences are flagged as errors by the compiler
- Examples (incorrect programs)

  -
    ```
    main(){
      int a=5;
      if (a<5) {break; printf("hello-1");};
      printf("hello-2");}
    }
    ```
  - Replacing `break` with `continue` in the above program is also erroneous

break and *continue* Statements (correct programs)

- The program below prints 6

```
main(){int a,b=10; for(a=1;a<5;a++) b--;
           printf("%d",b);}
```

- The program below prints 8

```
main(){int a,b=10; for(a=1;a<5;a++)
   { if (a==3) break; b--;} printf("%d",b);}
```

- The program below prints 7

```
main(){int a,b=10; for(a=1;a<5;a++)
   { if (a==3) continue; b--;} printf("%d",b);}
```

- This program also prints 8

```
main(){int a,b=10; for(a=1;a<5;a++)
   { while (1) break;
     if (a==3) break; b--;} printf("%d",b);}
```

Handling *break* and *continue* Statements

- We need extra attributes for the non-terminal *STMT*
  - *STMT.break* and *STMT.continue*, along with *STMT.next*(existing one), all of which are lists of quadruples with unfilled branch targets
- *STMT* → *break*

  { STMT.break := makelist(nextquad); gen('goto __');
  STMT.next := makelist(NULL);
  STMT.continue := makelist(NULL); }
- *STMT* → *continue*

  { STMT.continue := makelist(nextquad); gen('goto __');
  STMT.next := makelist(NULL);
  STMT.break := makelist(NULL); }

SATG for *While-do* Statement with *break* and *continue*

- *WHILEXEP* $\rightarrow$ *while M E*
  { WHILEEXP.falselist := makelist(nextquad);
  gen('if E.result $\leq$ 0 goto __');
  WHILEEXP.begin := M.quad; }
- *STMT* $\rightarrow$ *WHILEXEP do* $STMT_1$
  { gen('goto WHILEEXP.begin');
  backpatch($STMT_1$.next, WHILEEXP.begin);
  backpatch($STMT_1$.continue, WHILEEXP.begin);
  STMT.continue := makelist(NULL);
  STMT.break := makelist(NULL);
  STMT.next := merge(WHILEEXP.falselist, $STMT_1$.break); }
- *M* $\rightarrow \epsilon$
  { M.quad := nextquad; }

Code Generation Template for *C For-Loop* with *break* and *continue*

```
for ( $E_1$; $E_2$; $E_3$ ) $S$
            code for $E_1$
L1:         code for $E_2$ (result in T)
            goto L4
L2:         code for $E_3$
            goto L1
L3:         code for $S$ /* all breaks out of S goto L5 */
/* all continues and other jumps out of S goto L2 */
            goto L2
L4:         if  T == 0  goto  L5 /* if T is zero, jump to exit */
            goto L3
L5:         /* exit */
```

Code Generation for C For-Loop with *break* and *continue*

- $STMT \rightarrow for\ (\ E_1;\ M\ E_2;\ N\ E_3\ )\ P\ STMT_1$
  { gen('goto N.quad+1'); Q1 := nextquad;
  gen('if E_2.result == 0 goto __'); gen('goto P.quad+1');
  backpatch(makelist(N.quad), Q1);
  backpatch(makelist(P.quad), M.quad);
  backpatch($STMT_1$.continue, N.quad+1);
  backpatch($STMT_1$.next, N.quad+1);
  STMT.next := merge($STMT_1$.break, makelist(Q1));
  STMT.break := makelist(NULL);
  STMT.continue := makelist(NULL); }
- $M \rightarrow \epsilon$ { M.quad := nextquad; }
- $N \rightarrow \epsilon$ { N.quad := nextquad; gen('goto __'); }
- $P \rightarrow \epsilon$ { P.quad := nextquad; gen('goto __'); }

LATG for *If-Then-Else* Statement

Assumption: No short-circuit evaluation for E

```
If  (E)  S1  else  S2
        code for E (result in T)
        if  T≤ 0  goto  L1 /* if T is false, jump to else part */
        code for S1 /* all exits from within S1 also jump to L2 */
        goto L2 /* jump to exit */
L1:     code for S2 /* all exits from within S2 also jump to L2 */
L2:     /* exit */
```

```
$S \rightarrow$ if E { N := nextquad; gen(‘if E.result <= 0 goto __’); }
        $S_1$ else { M := nextquad; gen(‘goto __’);
                     backpatch(N, nextquad); }
        $S_2$ { S.next := merge(makelist(M), $S_1$.next, $S_2$.next); }
```

LATG for *While-do* Statement

Assumption: No short-circuit evaluation for E

```
while  (E)  do  S
L1:       code for E (result in T)
          if  T≤ 0  goto  L2 /* if T is false, jump to exit */
          code for S /* all exits from within S also jump to L1 */
          goto L1 /* loop back */
L2:       /* exit */
```

```
S → while { M := nextquad; }
      E { N := nextquad; gen('if E.result <= 0 goto __'); }
      do S_1 { backpatch(S_1.next, M); gen('goto M');
                S.next := makelist(N); }
```

LATG for Other Statements

- $S \rightarrow A$ { S.next := makelist(NULL); }
- $S \rightarrow \{\ SL\ \}$ { S.next := SL.next; }
- $SL \rightarrow \epsilon$ { SL.next := makelist(NULL); }
- $SL \rightarrow S;$ { backpatch(S.next, nextquad); }
  SL_1 { SL.next := SL_1.next; }
- When a function ends, we perform { gen('func end'); }. No backpatching of SL.next is required now, since this list will be empty, due to the use of $SL \rightarrow \epsilon$ as the last production.
- LATG for function declaration and call, and return statement are left as exercises

LATG for Expressions

- $A \rightarrow L = E$
  { if (L.offset == NULL) /* simple id */
  gen('L.place = E.result');
  else gen('L.place[L.offset] = E.result'); }
- $E \rightarrow T$ { E'.left := T.result; }
  E' { E.result := E'.result; }
- $E' \rightarrow + T$ { temp := newtemp(T.type);
  gen('temp = E'.left + T.result'); E'_1.left := temp; }
  E'_1 { E'.result := E'_1.result; }
  Note: Checking for compatible types, etc., are all required here as well. These are left as exercises.
- $E' \rightarrow \epsilon$ { E'.result := E'.left; }
- Processing $T \rightarrow F\ T'$, $T' \rightarrow *F\ T' \mid \epsilon$, $F \rightarrow (\ E\ )$, boolean and relational expressions are all similar to the above productions

LATG for Expressions(contd.)

- *F* $\rightarrow$ *L* { if (L.offset == NULL) F.result := L.place;
  else { F.result := newtemp(L.type);
  gen(‘F.result = L.place[L.offset]’); }
- *F* $\rightarrow$ *num* { F.result := newtemp(num.type);
  gen(‘F.result = num.value’); }
- *L* $\rightarrow$ *id* { search(id.name, vn); INDEX.arrayptr := vn; }
  INDEX { L.place := vn; L.offset := INDEX.offset; }
- *INDEX* $\rightarrow \epsilon$ { INDEX.offset := NULL; }
- *INDEX* $\rightarrow$ [{ ELIST.dim := 1;
  ELIST.arrayptr := INDEX.arrayptr; }
  ELIST]
  { temp := newtemp(int); INDEX.offset := temp;
  ele_size := INDEX.arrayptr -> ele_size;
  gen(‘temp = ELIST.result * ele_size’); }

LATG for Expressions(contd.)

- *ELIST* → *E* { INDEXLIST.dim := ELIST.dim+1;
  INDEXLIST.arrayptr := ELIST.arrayptr;
  INDEXLIST.left := E.result; }
  INDEXLIST { ELIST.result := INDEXLIST.result; }
- *INDEXLIST* → ϵ { INDEXLIST.result := INDEXLIST.left; }
- *INDEXLIST* → **,** { **action 1** }
  ELIST { gen('temp = temp + ELIST.result');
  INDEXLIST.result := temp; }

action 1:

```
{ temp := newtemp(int);
  num_elem := rem_num_elem(INDEXLIST.arrayptr,
                                    INDEXLIST.dim);
  gen('temp = INDEXLIST.left * num_elem');
  ELIST.arrayptr := INDEXLIST.arrayptr;
  ELIST.dim := INDEXLIST.dim; }
```

LATG for Expressions(contd.)

- The function rem_num_elem(arrayptr, dim) computes the product of the dimensions of the array, starting from dimension *dim*. For example, consider the expression, `a[i,j,k,l]`, and its declaration `int a[10,20,30,40]`. The expression translates to $i * 20 * 30 * 40 + j * 30 * 40 + k * 40 + l$. The above function returns, 24000(dim=2), 1200(dim=3), and 40(dim=3).

Run-time Environments - 1

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support?
- Parameter passing methods
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

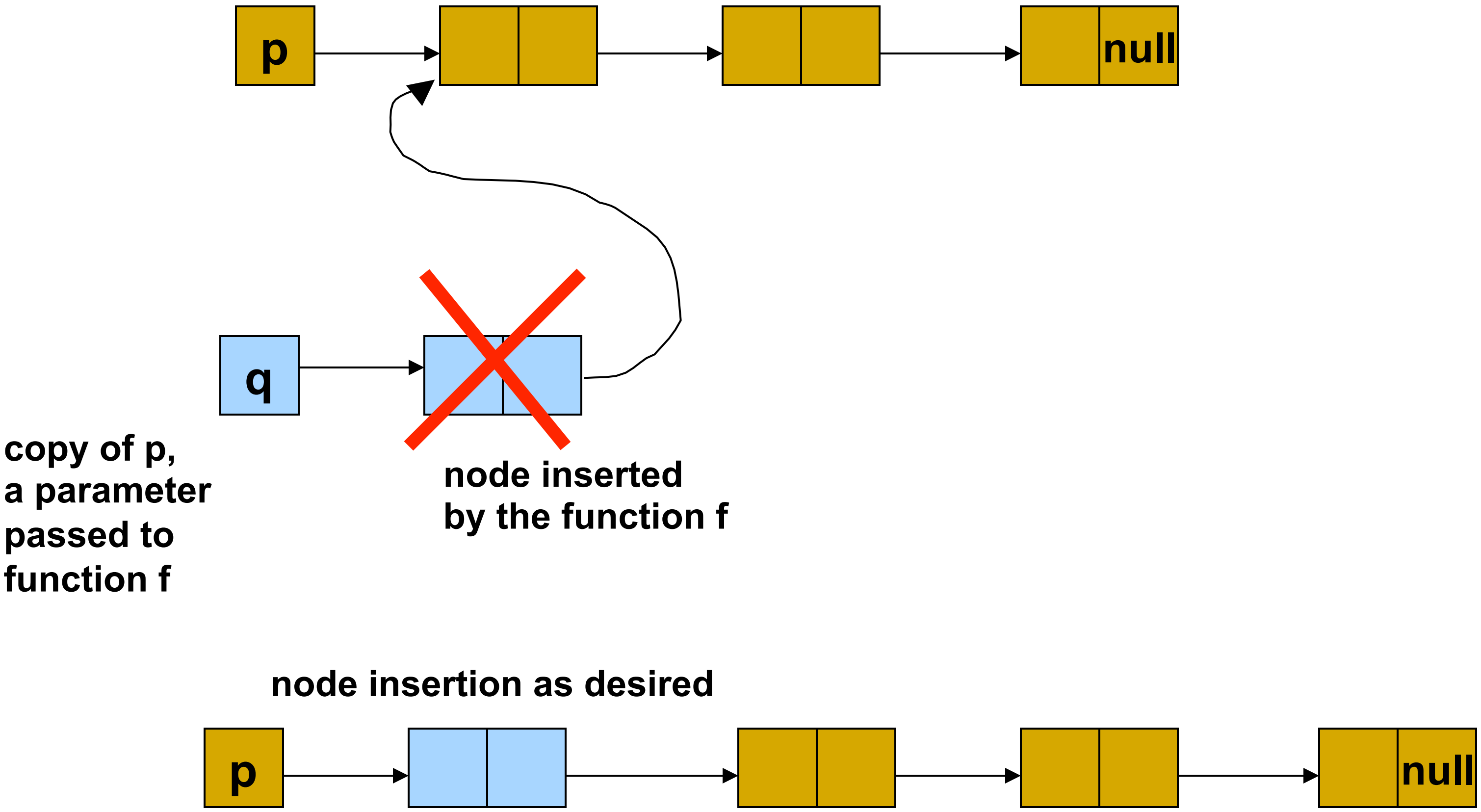
What is Run-time Support?

- It is not enough if we generate machine code from intermediate code
- Interfaces between the program and computer system resources are needed
    - There is a need to manage memory when a program is running
        - This memory management must connect to the data objects of programs
        - Programs request for memory blocks and release memory blocks
        - Passing parameters to fucntions needs attention
    - Other resources such as printers, file systems, etc., also need to be accessed
- These are the main tasks of run-time support
- In this lecture, we focus on memory management

Parameter Passing Methods - Call-by-value

- At runtime, prior to the call, the parameter is evaluated, and its actual value is put in a location private to the called procedure
  - Thus, there is no way to change the actual parameters.
  - Found in C and C++
  - C has only call-by-value method available
    - Passing pointers does not constitute call-by-reference
    - Pointers are also copied to another location
    - Hence in C, there is no way to write a function to insert a node at the front of a linked list (just after the header) without using pointers to pointers

Problem with Call-by-Value



copy of p,
a parameter
passed to
function f

node inserted
by the function f

node insertion as desired

Parameter Passing Methods
- Call-by-Reference

- At runtime, prior to the call, the parameter is evaluated and put in a temporary location, if it is not a variable
- The **address** of the variable (or the temporary) is passed to the called procedure
- Thus, the actual parameter may get changed due to changes to the parameter in the called procedure
- Found in C++ and Java

Call-by-Value-Result

- ***Call-by-value-result*** is a hybrid of Call-by-value and Call-by-reference
- Actual parameter is calculated by the calling procedure and is copied to a local location of the called procedure
- Actual parameter's value is not affected during execution of the called procedure
- At return, the value of the formal parameter is copied to the actual parameter, if the actual parameter is a variable
- Becomes different from call-by-reference method
  - when global variables are passed as parameters to the called procedure and
  - the same global variables are also updated in another procedure invoked by the called procedure
- Found in Ada

Difference between Call-by-Value, Call-by-Reference, and Call-by-Value-Result

```
int a;
void Q()
    { a = a+1; }
void R(int x);
    { x = x+10; Q(); }
main()
    { a = 1; R(a); print(a); }
```

call-by-value	call-by-reference	call-by-value-result
2	**12**	**11**

Value of a printed

Note: In Call-by-V-R, value of x is copied into a, when proc R returns. Hence a=11.

Parameter Passing Methods - Call-by-Name

- Use of a call-by-name parameter implies a **textual** substitution of the formal parameter name by the **actual** parameter
- For example, if the procedure

  void R (int X, int I);

  { I = 2; X = 5; I = 3; X = 1; }

  is called by R(B[J*2], J)

  this would result in (effectively) changing the body to

  { J = 2; B[J*2] = 5; J = 3; B[J*2] = 1; }

  just before executing it

Parameter Passing Methods - Call by Name

- Note that the actual parameter corresponding to *X* changes whenever *J* changes
  - Hence, we cannot evaluate the address of the actual parameter just once and use it
  - It must be recomputed every time we reference the formal parameter within the procedure
- A separate routine (called *thunk*) is used to evaluate the parameters whenever they are used
- Found in Algol and functional languages

Example of Using the Four Parameter Passing Methods

```
void swap (int x, int y)
{ int temp;
   temp = x;
   x = y;
   y = temp;
} /*swap*/
...
{ i = 1;
   a[i] =10; /* int a[5]; */
   print(i,a[i]);
   swap(i,a[i]);
   print(i,a[1]); }
```

- Results from the 4 parameter passing methods (print statements)

call-by-value	call-by-reference	call-by-val-result	call-by-name
1 10	1 10	1 10	1 10
1 10	10 1	10 1	error!

Reason for the error in the Call-by-name Example

The problem is in the swap routine

temp = i; /* => temp = 1 */
i = a[i]; /* => i =10 since a[i] ==10 */
a[i] = temp; /* => a[10] = 1 => index out of bounds */

Run-time Environments - 2

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation
- Activation records
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Code and Data Area in Memory

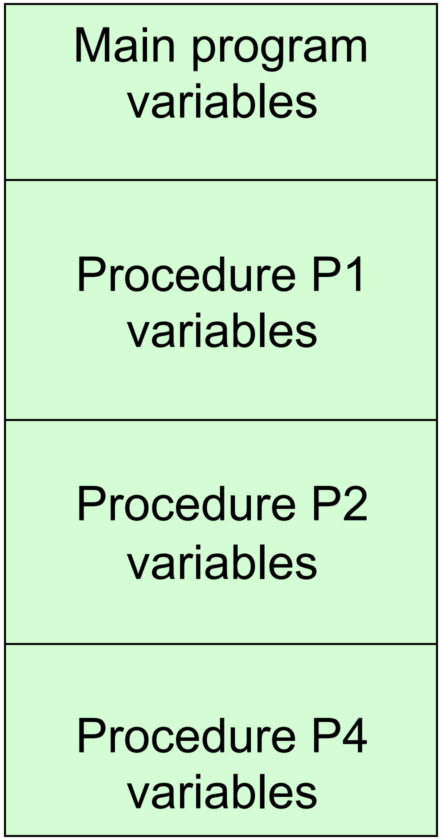
- Most programming languages distinguish between code and data
- Code consists of only machine instructions and normally does not have embedded data
    - Code area normally does not grow or shrink in size as execution proceeds
        - Unless code is loaded dynamically or code is produced dynamically
            - As in Java – dynamic loading of classes or producing classes and instantiating them dynamically through reflection
    - Memory area can be allocated to code statically
        - We will not consider Java further in this lecture
- Data area of a program may grow or shrink in size during execution

Static Versus Dynamic Storage Allocation

- Static allocation
  - Compiler makes the decision regarding storage allocation by looking only at the program text
- Dynamic allocation
  - Storage allocation decisions are made only while the program is running
  - Stack allocation
    - Names local to a procedure are allocated space on a stack
  - Heap allocation
    - Used for data that may live even after a procedure call returns
    - Ex: dynamic data structures such as symbol tables
    - Requires memory manager with garbage collection

Static Data Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at compile time
    - No stack/heap allocation; no overheads
    - Ex: Fortran IV and Fortran 77
    - Variable access is fast since addresses are known at compile time
    - No recursion


Main program variables

Procedure P1 variables

Procedure P2 variables

Procedure P4 variables

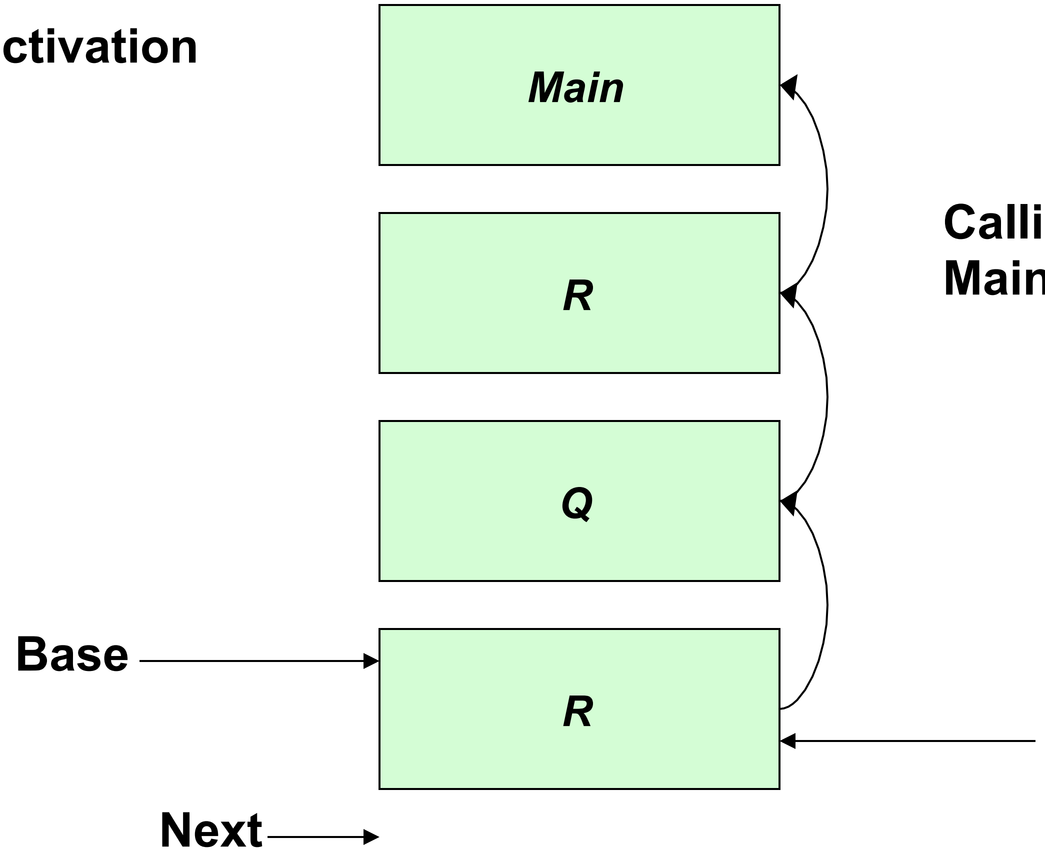
Main memory

Dynamic Data Storage Allocation

- Compiler allocates space only for global variables at compile time
- Space for variables of procedures will be allocated at run-time
  - Stack/heap allocation
  - Ex: C, C++, Java, Fortran 8/9
  - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
  - Recursion can be implemened

Dynamic Stack Storage Allocation

Stack of activation records



Main

R

Q

Base

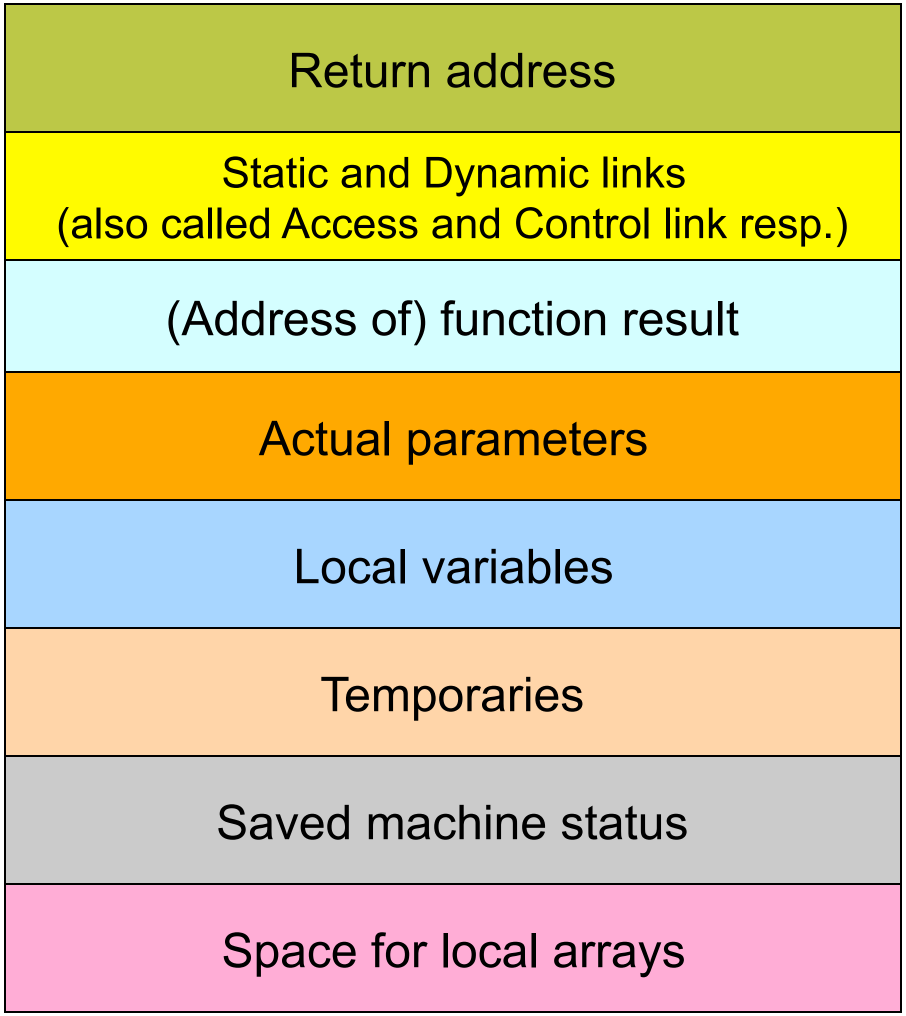
R

Next

Calling sequence:
Main → R → Q → R

Currently active procedure

Activation Record Structure



Return address

Static and Dynamic links
(also called Access and Control link resp.)

(Address of) function result

Actual parameters

Local variables

Temporaries

Saved machine status

Space for local arrays

Note:

The position of the fields of the act. record as shown are only notional.

Implementations can choose different orders; e.g., function result could be after local var.

Variable Storage Offset Computation

- The compiler should compute
    - the offsets at which variables and constants will be stored in the activation record (AR)
- These offsets will be with respect to the pointer pointing to the beginning of the AR
- Variables are usually stored in the AR in the declaration order
- Offsets can be easily computed while performing semantic analysis of declarations

Overlapped Variable Storage for Blocks in C

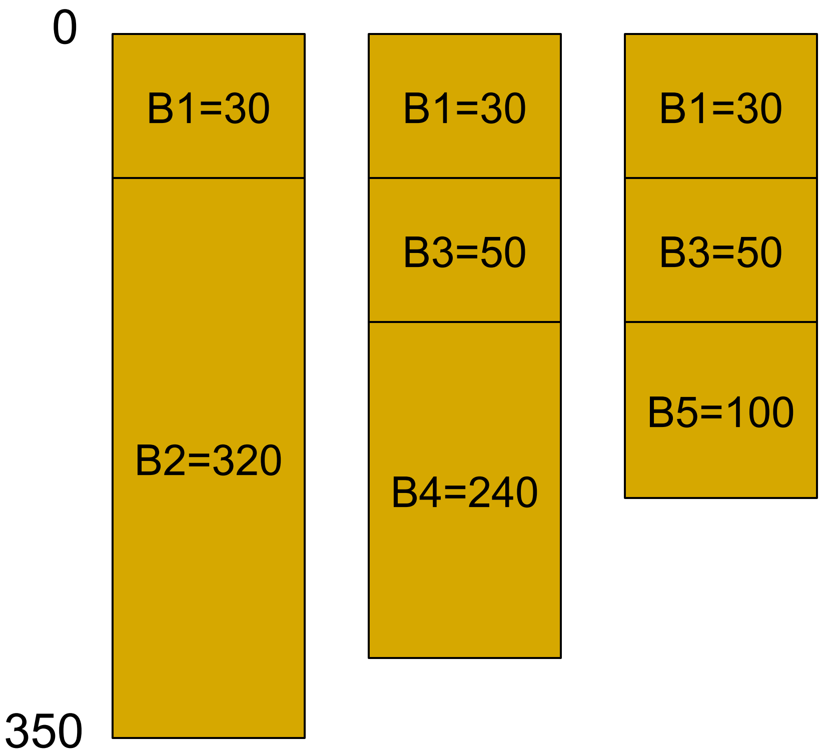
```
Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
         30+max(320, (50+240)) =
         30+max(320,290) = 350
```

```
int example(int p1, int p2)
B1 { a,b,c; /* sizes - 10,10,10;
              offsets 0,10,20 */
         ...
         B2 { d,e,f; /* sizes - 100, 180, 40;
                       offsets 30, 130, 310 */
                 ...}
         B3 { g,h,i; /* sizes - 20,20,10;
                       offsets 30, 50, 70 */
                ...
                B4 { j,k,l; /* sizes - 70, 150, 20;
                              offsets 80, 150, 300 */
                        ... }
                B5 { m,n,p; /* sizes - 20, 50, 30;
                               offsets 80, 100, 150 */
                        ... }
            }
   }
```

Overlapped storage

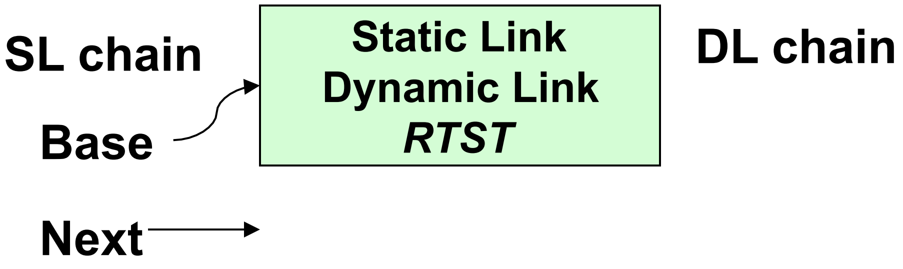
Overlapped storage

Overlapped Variable Storage for Blocks in C (Ex.)



Storage required =
B1+max(B2,(B3+max(B4,B5))) =
30+max(320,(50+max(240,100))) =
30+max(320, (50+240)) =
30+max(320,290) = 350

Allocation of Activation Records (nested procedures)


SL chain

Base

Next

Static Link
Dynamic Link
RTST

DL chain

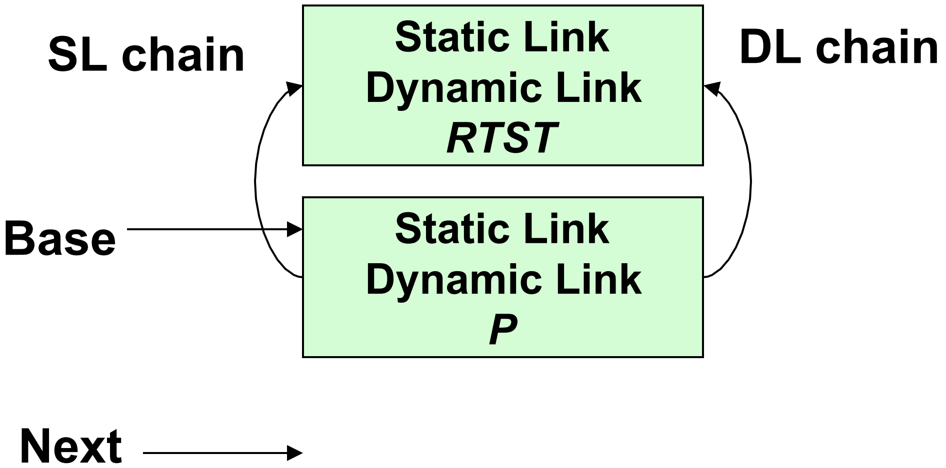
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```

Activation records are created at procedure entry time and destroyed at procedure exit time

RTST -> P -> R -> Q -> R

Allocation of Activation Records (contd.)

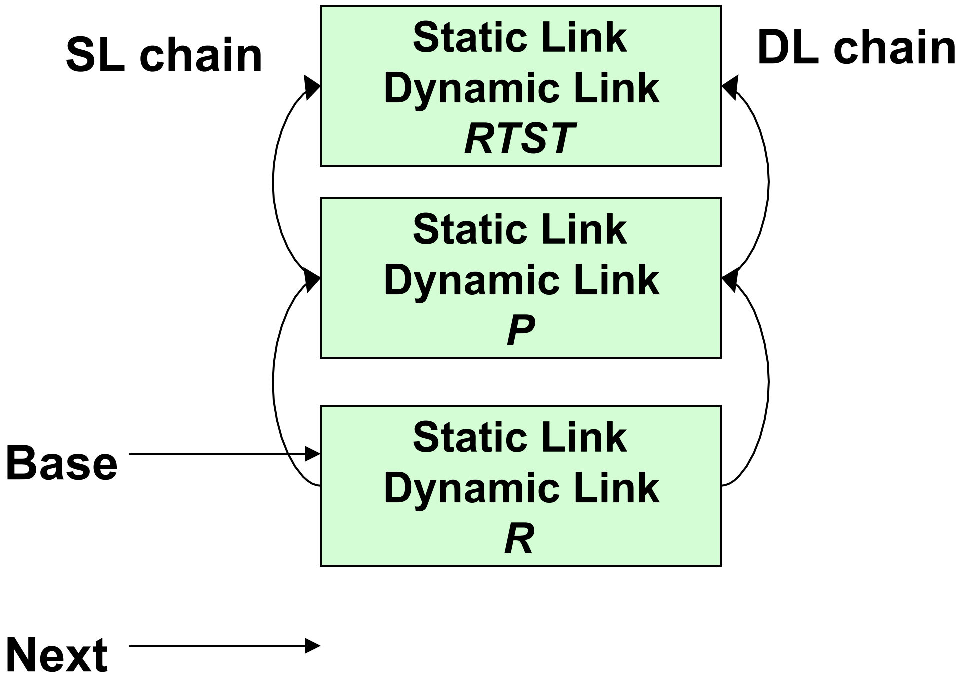
program *RTST*;
  procedure *P*;
    procedure *Q*;
      begin *R*; end
    procedure *R*;
      begin *Q*; end
  begin *R*; end
begin *P*; end



RTST -> **P** -> R -> Q -> R

Allocation of Activation Records (contd.)

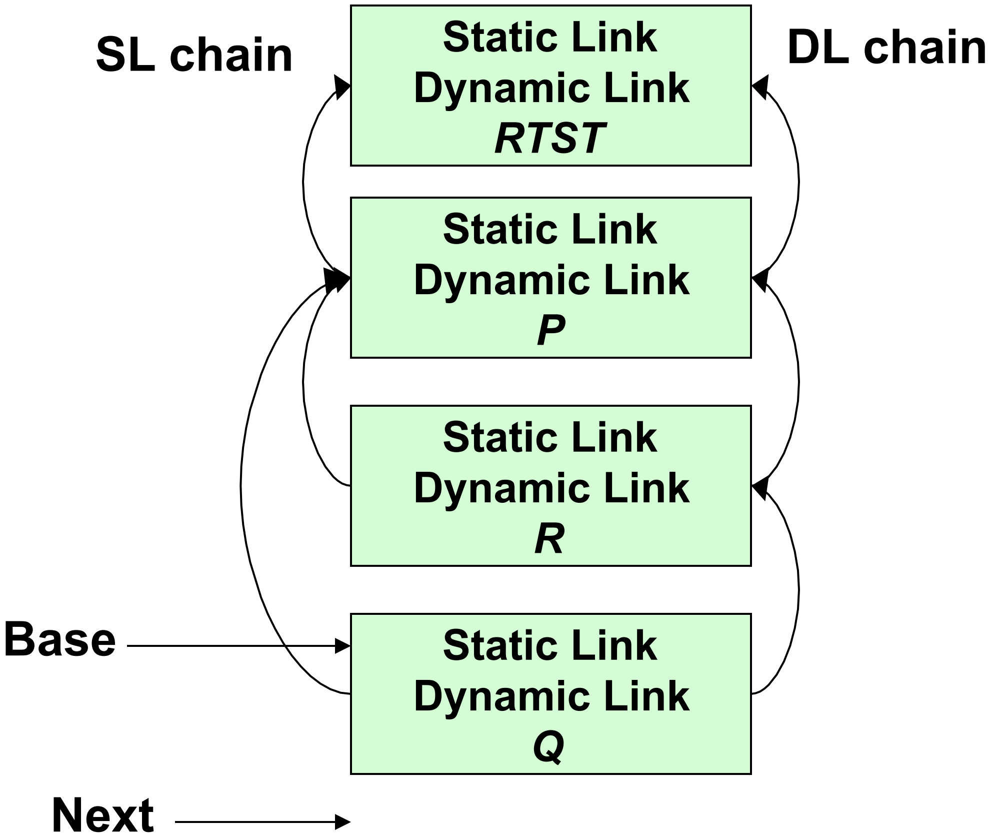
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> **R** -> Q -> R

Allocation of Activation Records (contd.)

```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```

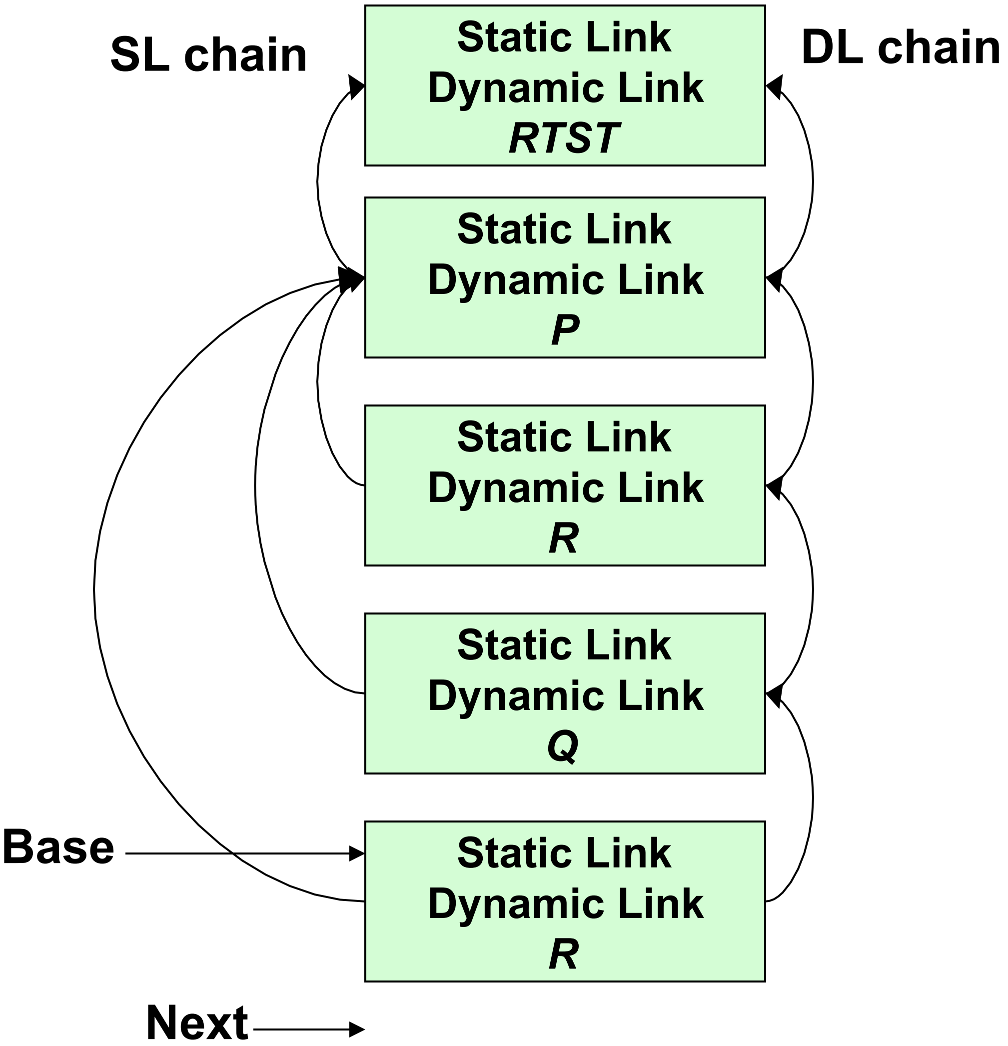


RTST -> **P** -> **R** -> **Q** -> R

Allocation of Activation Records (contd.)

1 program *RTST*;
2   procedure *P*;
3   procedure *Q*;
        begin *R*; end
3   procedure *R*;
        begin *Q*; end
    begin *R*; end
  begin *P*; end

RTST^1 -> P^2 -> R^3 -> Q^3 -> R^3



Allocation of Activation Records (contd.)

Skip L_1-L_2+1 records starting from the caller's AR and establish the static link to the AR reached

L_1 – caller, L_2 – Callee

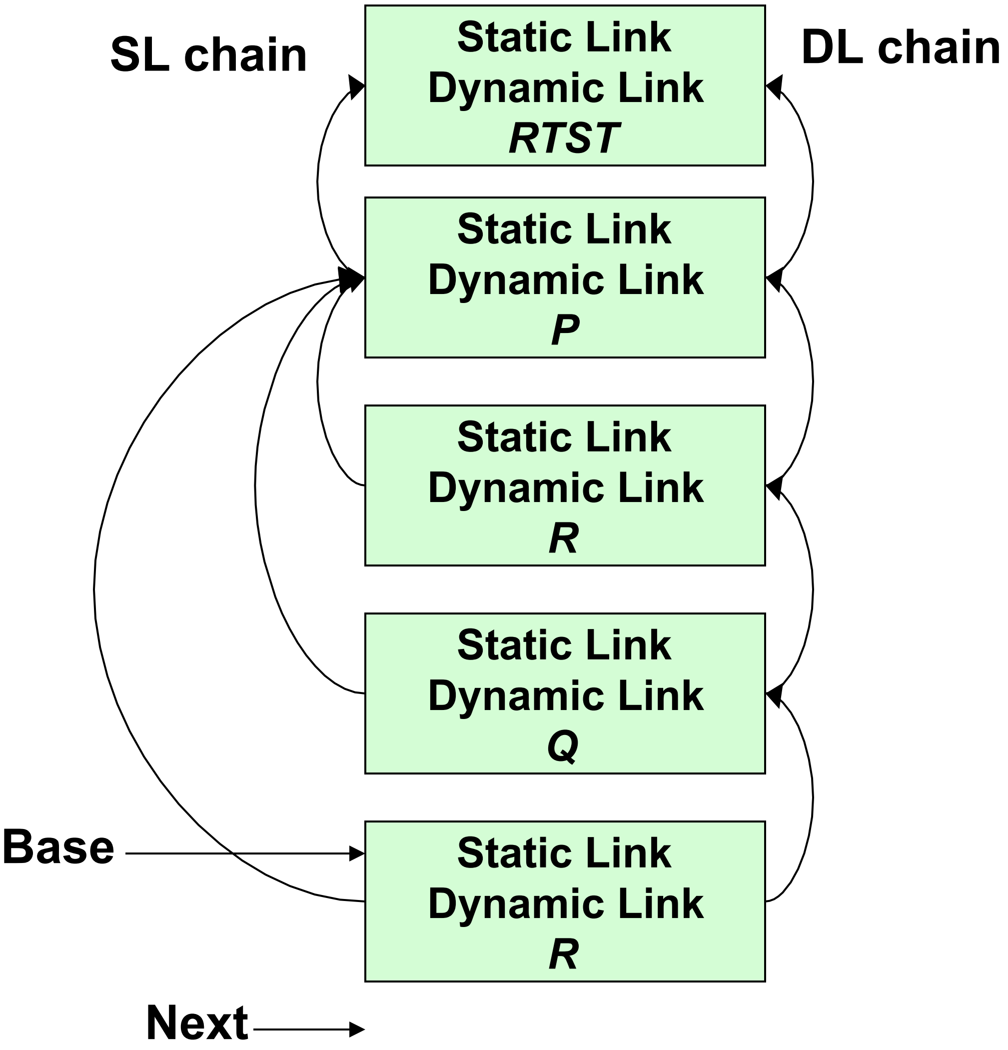
RTST^1 -> P^2 -> R^3 -> Q^3 -> R^3

Ex: Consider P^2 -> R^3

2-3+1=0; hence the SL of R points to P

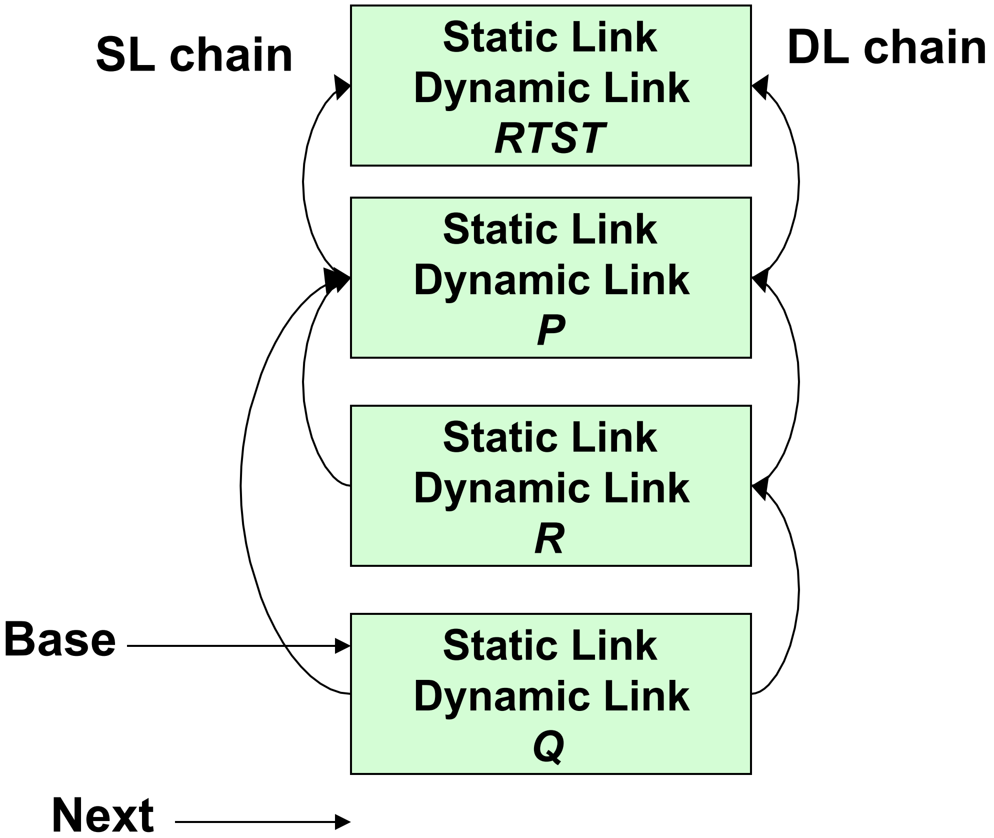
Consider R^3 -> Q^3

3-3+1=1; hence skipping one link starting from R, we get P; SL of Q points to P



Allocation of Activation Records (contd.)

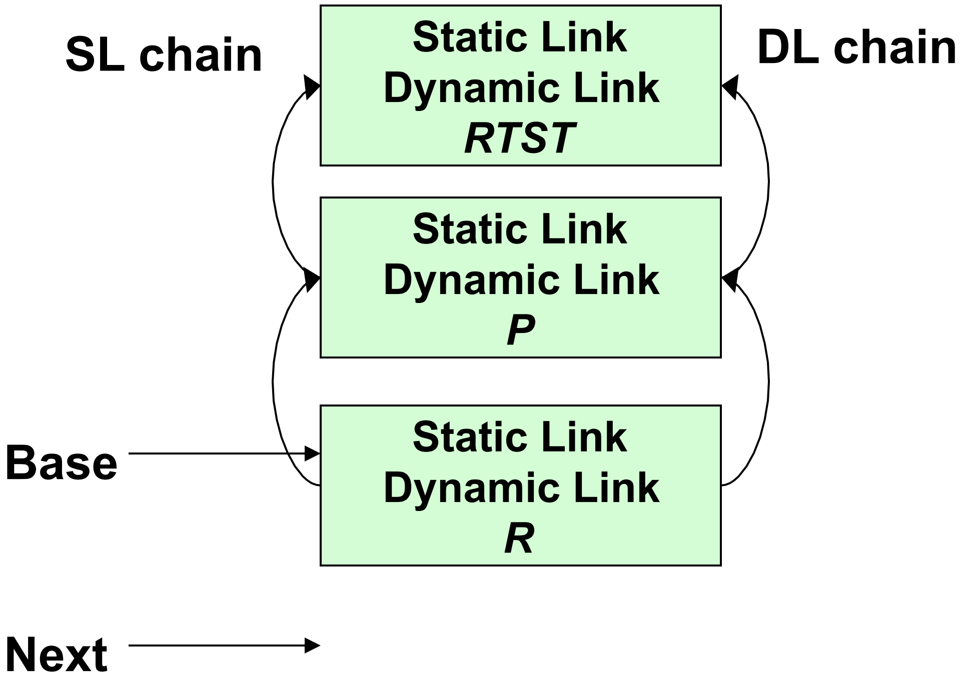
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> **R** -> **Q** <- R **Return from R**

Allocation of Activation Records (contd.)

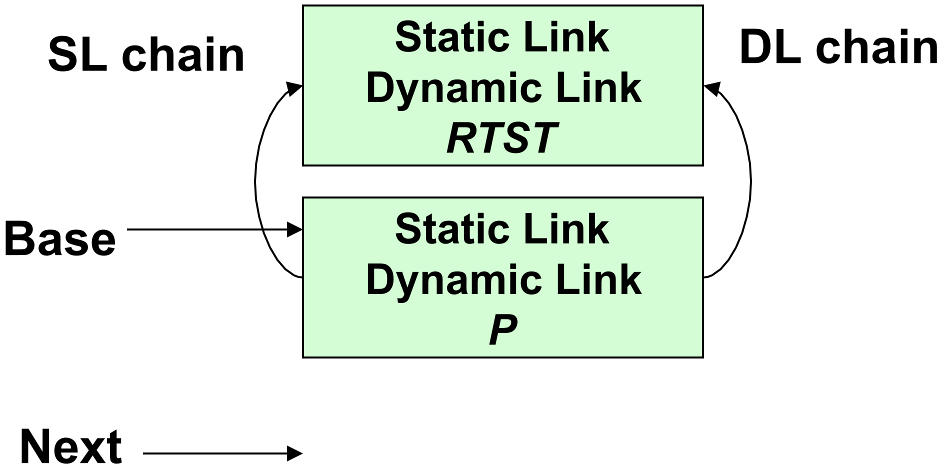
```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** -> **R** <- Q   **Return from Q**

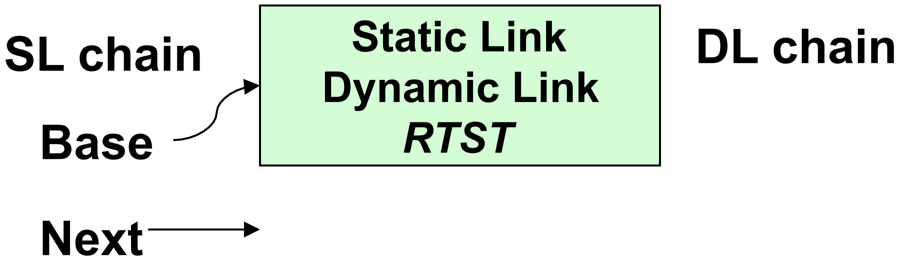
Allocation of Activation Records (contd.)

```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```



RTST -> **P** <- R **Return from R**

Allocation of Activation Records (contd.)



```
program RTST;
  procedure P;
    procedure Q;
      begin R; end
    procedure R;
      begin Q; end
  begin R; end
begin P; end
```

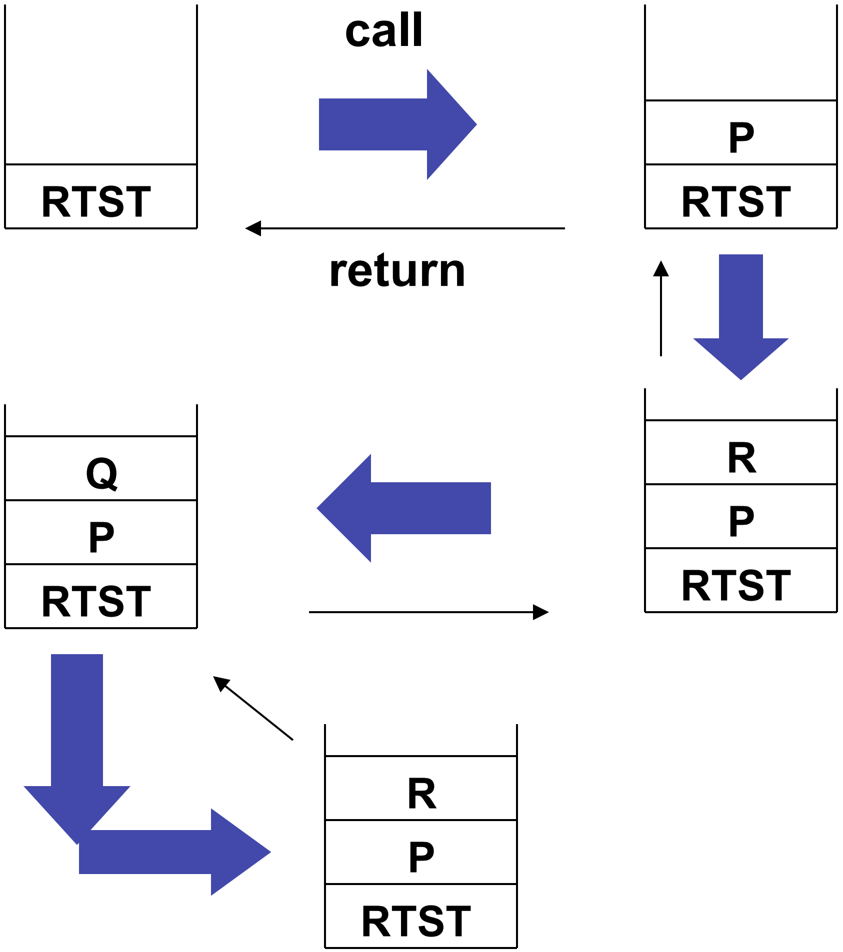
RTST <- P **Return from P**

Display Stack of Activation Records

```
1  program RTST;
2    procedure P;
3      procedure Q;
         begin R; end
3      procedure R;
         begin Q; end
     begin R; end
   begin P; end
```

Pop L_1-L_2+1 records off the display of the caller and push the pointer to AR of callee (L_1 – caller, L_2 – Callee)

The popped pointers are stored in the AR of the caller and restored to the DISPLAY after the callee returns



Static Scope and Dynamic Scope

- *Static Scope*
  - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
  - Uses the *static* (unchanging) relationship between blocks in the program text
- *Dynamic Scope*
  - A global identifier refers to the identifier associated with the most recent activation record
  - Uses the actual sequence of calls that are executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

Static Scope and Dynamic Scope : An Example

```
int x = 1, y = 0;
int g(int z)
  { return x+z;}
int f(int y) {
    int x; x = y+1;
    return g(y*x);
}
y = f(3);
---------------------------------
```

After the call to g,
Static scope: x = 1
Dynamic scope: x = 4

x	**1**	**outer block**
y	**0**	

y	**3**	**f(3)**
x	**4**	

z	**12**	**g(12)**

Stack of activation records
after the call to *g*

Static Scope and Dynamic Scope: Another Example

```
float r = 0.25;
void show() { printf(“%f”,r); }
void small() {
   float r = 0.125; show();
}
int main (){
show(); small(); printf(“\n”);
show(); small(); printf(“\n”);
}
```

- Under static scoping, the output is

  ```
  0.25  0.25
  0.25  0.25
  ```

- Under dynamic scoping, the output is

  ```
  0.25  0.125
  0.25  0.125
  ```

Run-time Environments - 3

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation (in part 2)
- Activation records (in part 2)
- Static scope and dynamic scope
- Passing functions as parameters
- Heap memory management
- Garbage Collection

Static Scope and Dynamic Scope

- *Static Scope*
  - A global identifier refers to the identifier with that name that is declared in the closest enclosing scope of the program text
  - Uses the *static* (unchanging) relationship between blocks in the program text
- *Dynamic Scope*
  - A global identifier refers to the identifier with that name associated with the most recent activation record
  - Uses the actual sequence of calls that is executed in the *dynamic* (changing) execution of the program
- Both are identical as far as local variables are concerned

Static Scope and Dynamic Scope : An Example

```
int x = 1, y = 0;
int g(int z)
  { return x+z;}
int f(int y) {
    int x; x = y+1;
    return g(y*x);
}
y = f(3);
```

After the call to g,
Static scope: x = 1
Dynamic scope: x = 4

x	**1**
y	**0**

outer block

y	**3**
x	**4**

f(3)

z	**12**

g(12)

Stack of activation records
after the call to *g*

Static Scope and Dynamic Scope: Another Example

```
float r = 0.25;
void show() { printf(“%f”,r); }
void small() {
   float r = 0.125; show();
}
int main (){
show(); small(); printf(“\n”);
show(); small(); printf(“\n”);
}
```

- Under static scoping, the output is

```
0.25  0.25
0.25  0.25
```

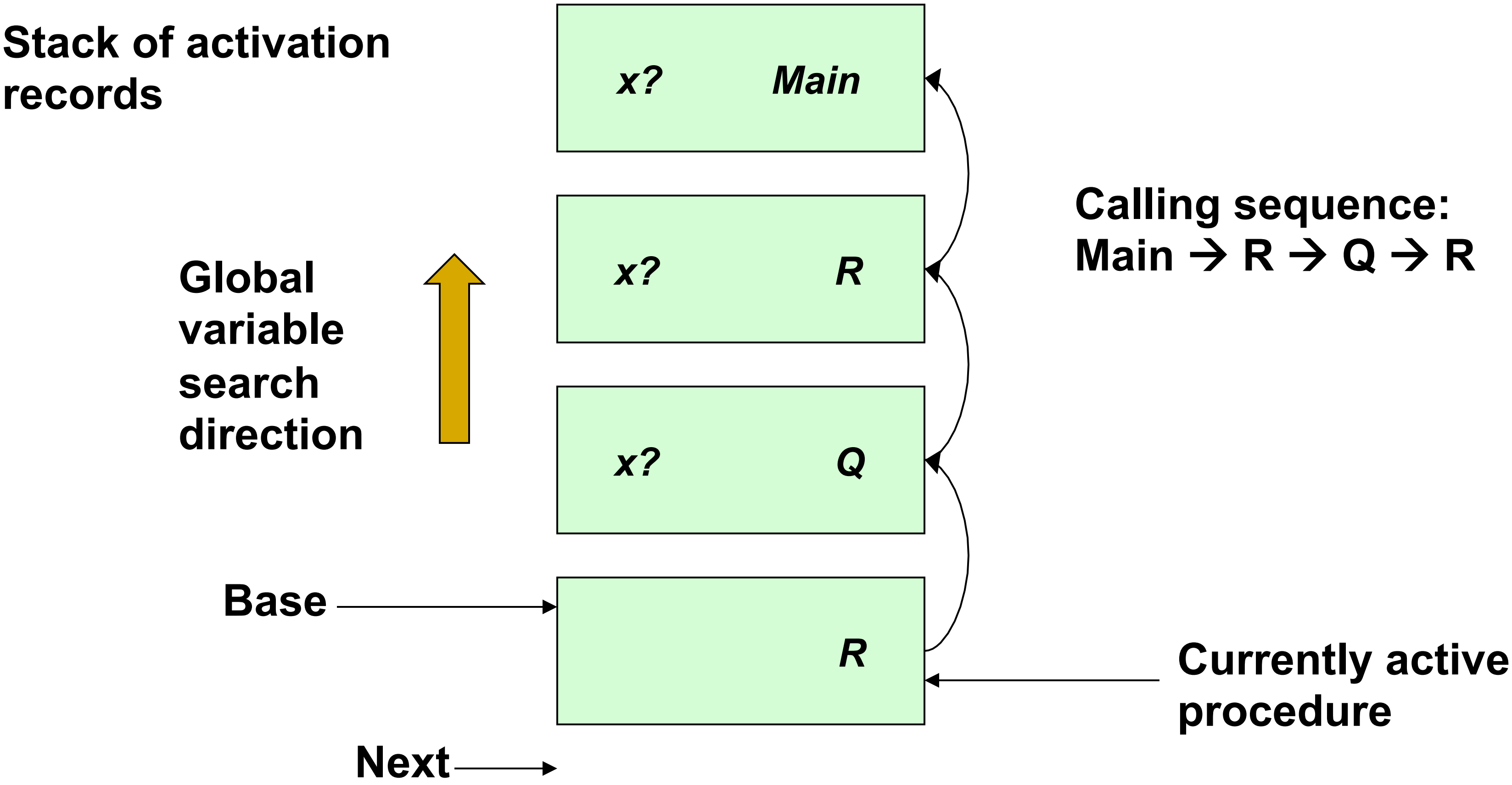
- Under dynamic scoping, the output is

```
0.25  0.125
0.25  0.125
```

Implementing Dynamic Scope – Deep Access Method

- Use *dynamic link* as *static link*
- Search activation records on the stack to find the first AR containing the non-local name
- The depth of search depends on the input to the program and cannot be determined at compile time
- Needs some information on the identifiers to be maintained at runtime within the ARs
- Takes longer time to access globals, but no overhead when activations begin and end

Deep Access Method - Example



Stack of activation records

Global variable search direction

Calling sequence:
Main → R → Q → R

Base

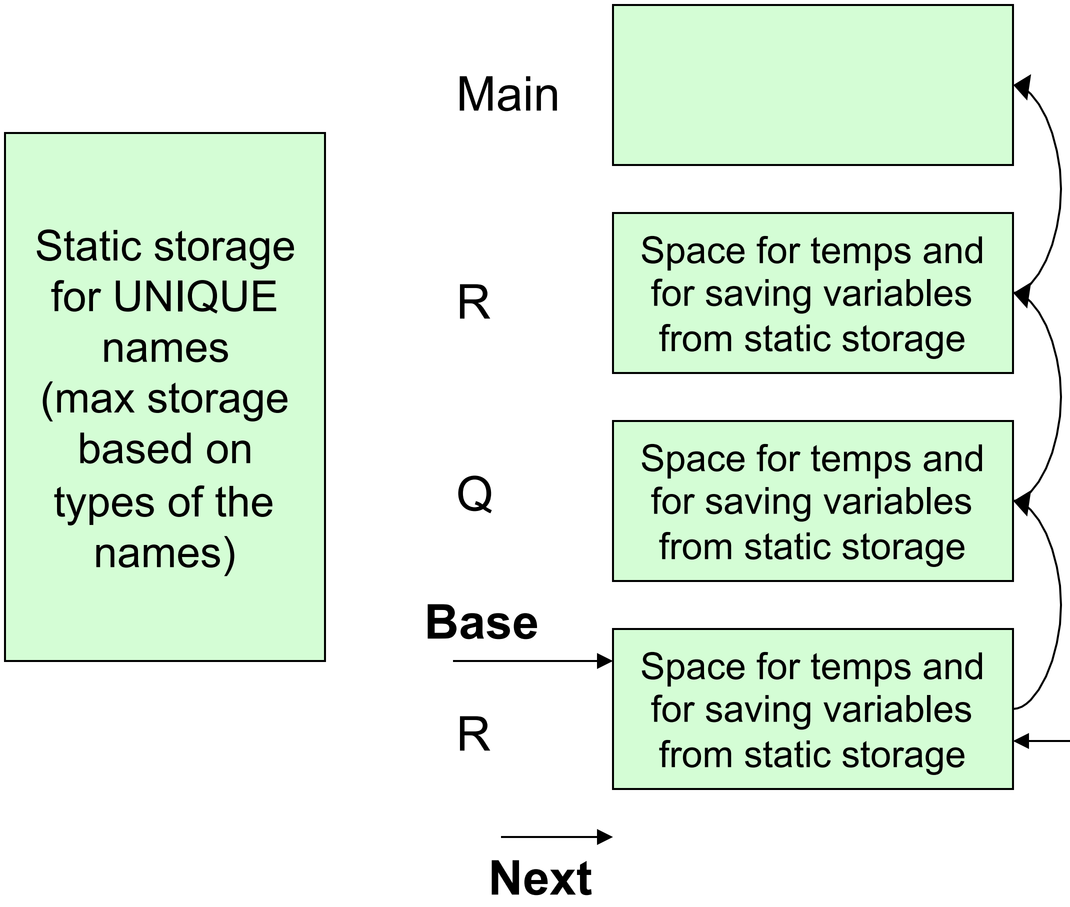
Next

Currently active procedure

Implementing Dynamic Scope – Shallow Access Method

- Allocate maximum static storage needed for *each* name (based on the types)
- When a new AR is created for a procedure *p*, a local name *n* in *p* takes over the static storage allocated to name *n*
    - Global variables are also accessed from the static storage
    - Temporaries are located in the AR
    - Therefore, all variable (not temp) accesses use static addresses
- The previous value of *n* held in static storage is saved in the AR of *p* and is restored when the activation of *p* ends
- Direct and quick access to globals, but some overhead is incurred when activations begin and end

Shallow Access Method - Example



Stack of activation records

Calling sequence:
Main → R → Q → R

Currently active procedure

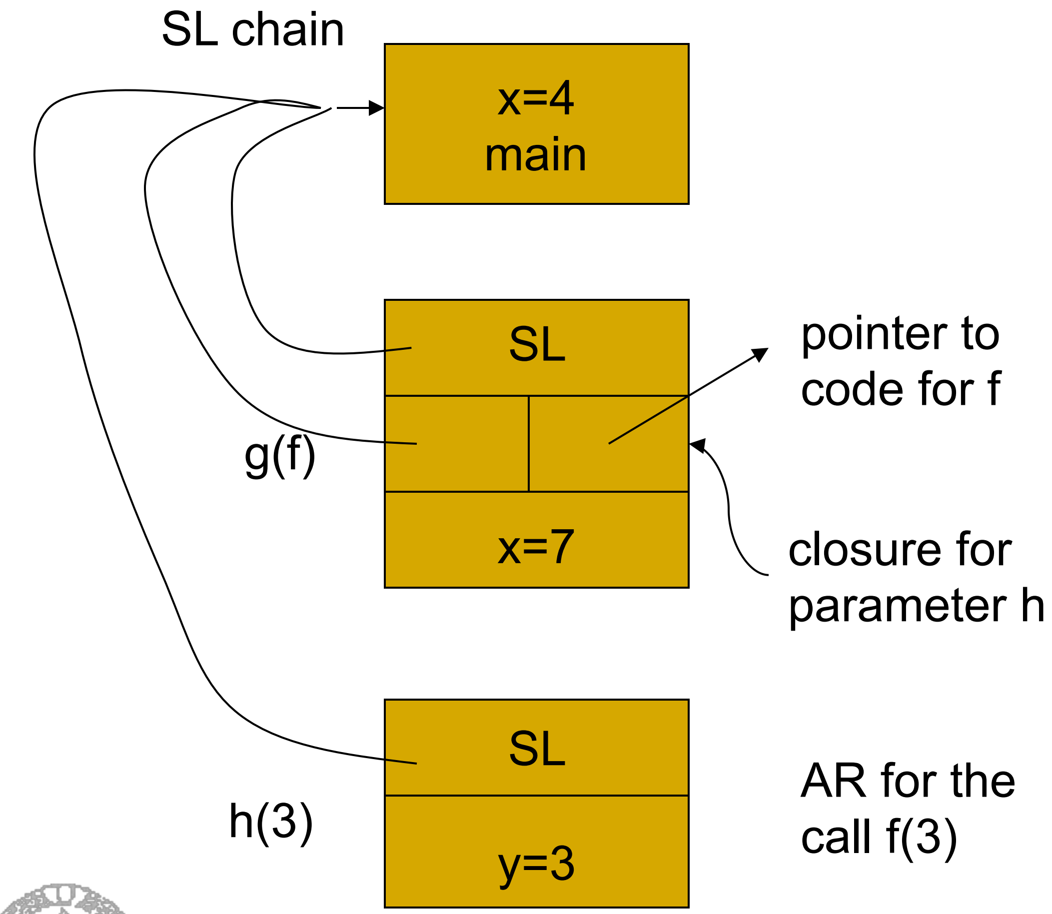
Passing Functions as Parameters

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

- A language has first-class functions if functions can be
  - declared within any scope
  - passed as arguments to other functions
  - returned as results of functions
- In a language with first-class functions and static scope, a function value is generally represented by a closure
  - a pair consisting of a pointer to function code and
  - a pointer to an activation record
- Passing functions as arguments is very useful in structuring of systems using callbacks

Passing Functions as Parameters – Implementation



SL chain

x=4
main

g(f)

SL

x=7

pointer to code for f

closure for parameter h

h(3)

SL

y=3

AR for the call f(3)

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

Passing Functions as Parameters: Implementation

An example:

```
main()
{ int x = 4;
  int f (int y) {
     return x*y;
  }
  int g (int → int h){
     int x = 7;
     return h(3) + x;
   }
   g(f);
}
```

- In this example, when executing the call h(3), h is really f and 3 is the parameter y of f
- Without passing a closure, the AR of the main program cannot be accessed, and hence, the value of x within f will not be 4
- In the call g(f), f is passed as a closure
- Closure may also contain information needed to set up AR (e.g., size of space for local variables, etc.)
- When processing the call h(3), after setting up an AR for h (i.e., f), the SL for the AR is set up using the AR pointer in the closure for f that has been passed to the call g(f)

Heap Memory Management

- Heap is used for allocating space for objects created at run time
  - For example: nodes of dynamic data structures such as linked lists and trees
- Dynamic memory allocation and deallocation based on the requirements of the program
  - *malloc*() and *free*() in C programs
  - *new*() and *delete*() in C++ programs
  - *new*() and garbage collection in Java programs
- Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic* (Java), or *fully automatic* (Lisp)

Memory Manager

- Manages heap memory by implementing mechanisms for allocation and deallocation, both manual and automatic
- Goals
    - Space efficiency: minimize fragmentation
    - Program efficiency: take advantage of locality of objects in memory and make the program run faster
    - Low overhead: allocation and deallocation must be efficient
- Heap is maintained either as a doubly linked list or as bins of free memory chunks (more on this later)

Allocation and Deallocation

- In the beginning, the heap is one large and contiguous block of memory
- As allocation requests are satisfied, chunks are cut off from this block and given to the program
- As deallocations are made, chunks are returned to the heap and are free to be allocated again (*holes*)
- After a number of allocations and deallocations, memory becomes fragmented and is not contiguous
- Allocation from a fragmented heap may be made either in a *first-fit* or *best-fit* manner
- After a deallocation, we try to *coalesce* contiguous holes and make a bigger hole (free chunk)

First-Fit and Best-Fit Allocation Strategies

- The *first-fit* strategy picks the **first** available chunk that satisfies the allocation request
- The *best-fit* strategy searches and picks the smallest (**best**) possible chunk that satisfies the allocation request
- Both of them chop off a block of the required size from the chosen chunk, and return it to the program
- The rest of the chosen chunk remains in the heap

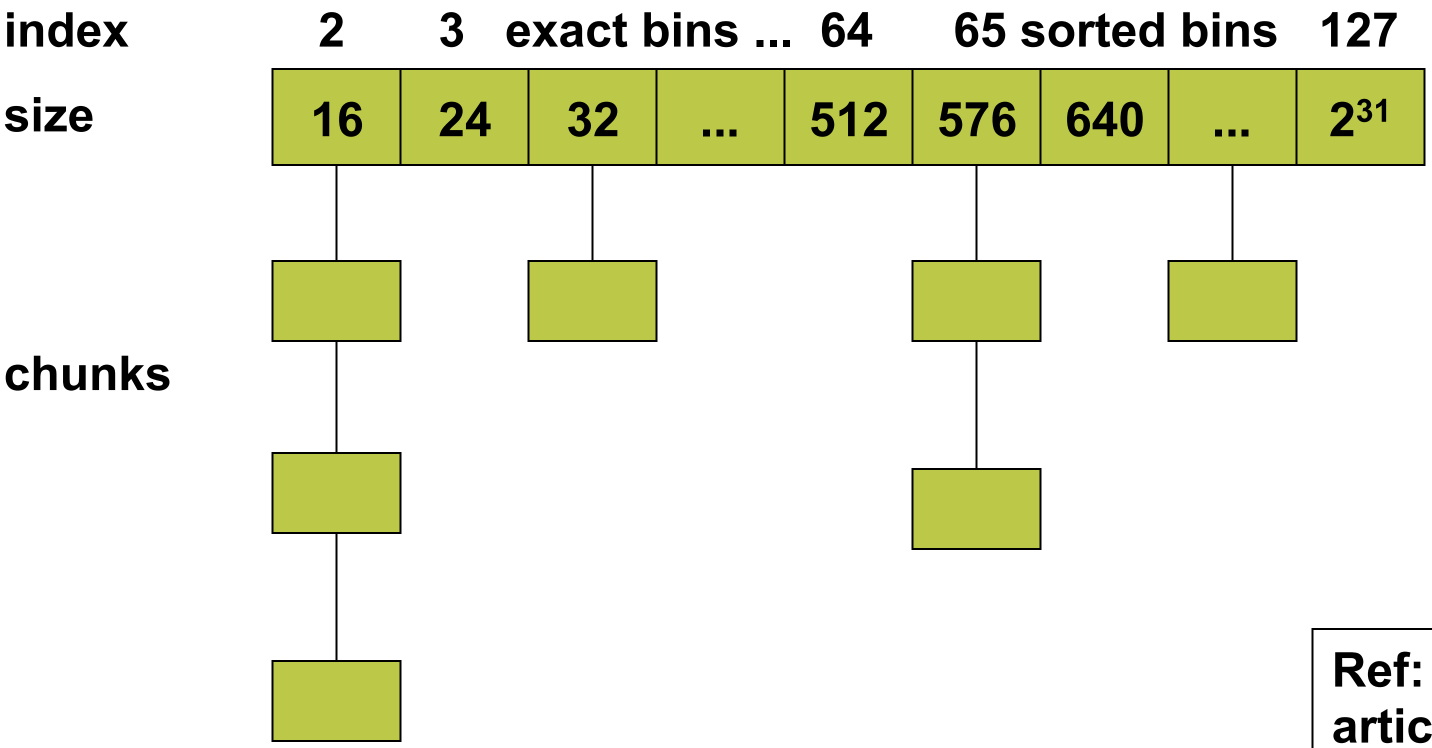
First-Fit and Best-Fit Allocation Strategies

- Best-fit strategy has been shown to reduce fragmentation in practice, better than first-fit strategy
- *Next-fit* strategy tries to allocate the object in the chunk that has been split recently
  - Tends to improve speed of allocation
  - Tends to improve spatial locality since objects allocated at about the same time tend to have similar reference patterns and life times (cache behaviour may be better)

Bin-based Heap

- Free space is organized into *bins* according to their sizes (Lea Memory Manager in GCC)
  - Many more bins for smaller sizes, because there are many more small objects
  - A bin for every multiple of 8-byte chunks from 16 bytes to 512 bytes
  - Then approximately logarithmically (double previous size)
  - Within each "small size bin", chunks are all of the same size
  - In others, they are ordered by size
  - The last chunk in the last bin is the *wilderness chunk*, which gets us a chunk by going to the operating system

Bin-based Heap – An Example



index 2 3 exact bins ... 64 65 sorted bins 127

size 16 24 32 ... 512 576 640 ... 2^{31}

chunks

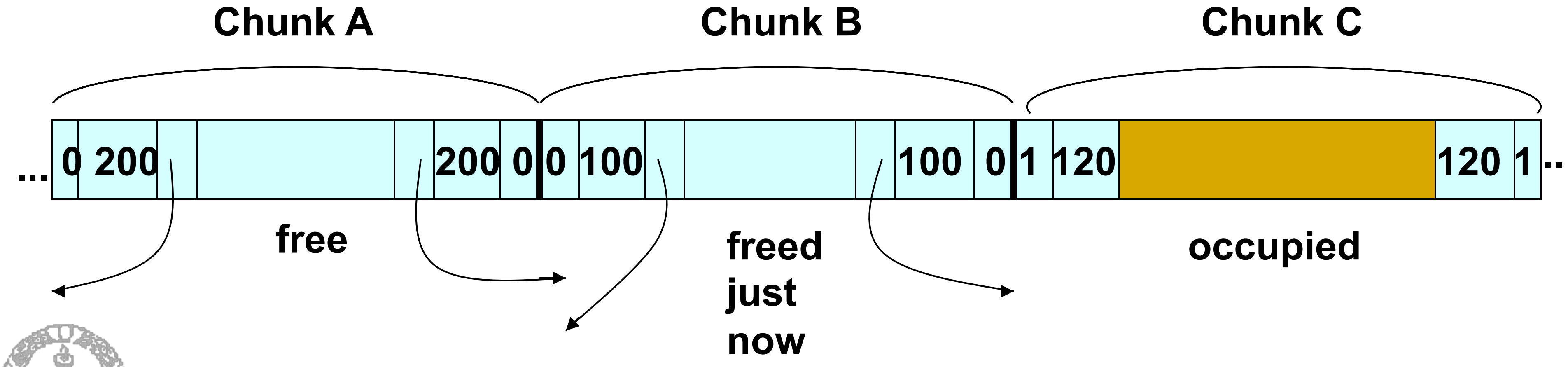
Ref: From Lea's article on memory manager in GCC

Managing and Coalescing Free Space

- Should coalesce adjacent chunks and reduce fragmentation
    - Many small chunks together cannot hold one large object
    - In the Lea memory manager, no coalescing in the exact size bins, only in the sorted bins
    - *Boundary tags* (free/used bit and chunk size) at each end of a chunk (for both used and free chunks)
    - A *doubly linked list* of free chunks

Boundary Tags and Doubly Linked List

3 adjacent chunks. Chunk B has been freed just now and returned to the free list. Chunks A and B can be merged, and this is done just before inserting it into the linked list. The merged chunk AB may have to be placed in a different bin.



Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data
- Both are serious and hard to debug
- Solution: automatic garbage collection

Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
  - Then, size and pointer fields of objects can be determined by the GC
  - Languages in which types of objects can be determined at compile time or run-time are type safe
    - Java is type safe
    - C and C++ are not type safe because they permit type casting, which creates new pointers
    - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

Run-time Environments - 4

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is run-time support? (in part 1)
- Parameter passing methods (in part 1)
- Storage allocation (in part 2)
- Activation records (in part 2)
- Static scope and dynamic scope (in part 3)
- Passing functions as parameters (in part 3)
- Heap memory management (in part 3)
- Garbage Collection

Problems with Manual Deallocation

- Memory leaks
  - Failing to delete data that cannot be referenced
  - Important in long running or nonstop programs
- Dangling pointer dereferencing
  - Referencing deleted data
- Both are serious and hard to debug
- Solution: automatic garbage collection

Garbage Collection

- Reclamation of chunks of storage holding objects that can no longer be accessed by a program
- GC should be able to determine types of objects
  - Then, size and pointer fields of objects can be determined by the GC
  - Languages in which types of objects can be determined at compile time or run-time are type safe
    - Java is type safe
    - C and C++ are not type safe because they permit type casting, which creates new pointers
    - Thus, any memory location can be (theoretically) accessed at any time and hence cannot be considered inaccessible

Reachability of Objects

- The *root set* is all the data that can be accessed (reached) directly by a program without having to dereference any pointer
- Recursively, any object whose reference is stored in a field of a member of the root set is also reachable
- New objects are introduced through object allocations and add to the set of reachable objects
- Parameter passing and assignments can propagate reachability
- Assignments and ends of procedures can terminate reachability

Reachability of Objects

- Similarly, an object that becomes *unreachable* can cause more objects to become unreachable
- A garbage collector periodically finds all unreachable objects by one of the two methods
  - Catch the transitions as reachable objects become unreachable
  - Or, periodically locate all reachable objects and infer that all *other* objects are unreachable

Reference Counting Garbage Collector

- This is an approximation to the first approach mentioned before
- We maintain a count of the references to an object, as the mutator (program) performs actions that may change the reachability set
- When the count becomes zero, the object becomes unreachable
- Reference count requires an extra field in the object and is maintained as below

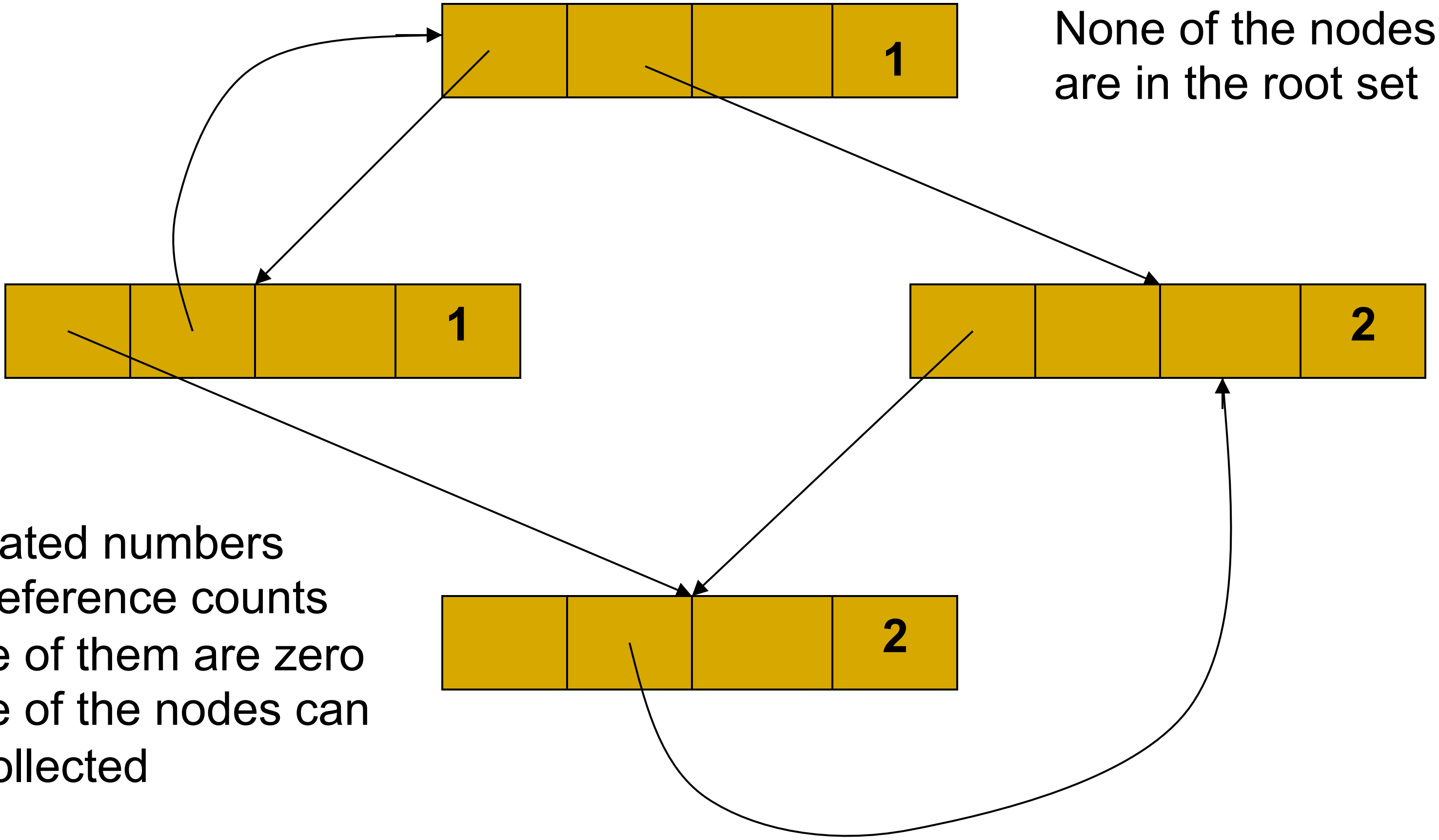
Maintaining Reference Counts

- *New object allocation.* ref_count=1 for the new object
- *Parameter passing.* ref_count++ for each object passed into a procedure
- *Reference assignments.* For u:=v, where u and v are references, ref_count++ for the object *v, and ref_count-- for the object *u
- *Procedure returns.* ref_count-- for each object pointed to by the local variables
- *Transitive loss of reachability.* Whenever ref_count of an object becomes zero, we must also decrement the ref_count of each object pointed to by a reference within the object

Reference Counting GC: Disadvantages and Advantages

- High overhead due to reference maintenance
- Cannot collect unreachable cyclic data structures (ex: circularly linked lists), since the reference counts never become zero
- Garbage collection is incremental
    - overheads are distributed to the mutator’s operations and are spread out throughout the life time of the mutator
- Garbage is collected immediately and hence space usage is low
- Useful for real-time and interactive applications, where long and sudden pauses are unacceptable

Unreachable Cyclic Data Structure



None of the nodes are in the root set

Indicated numbers are reference counts
None of them are zero
None of the nodes can be collected

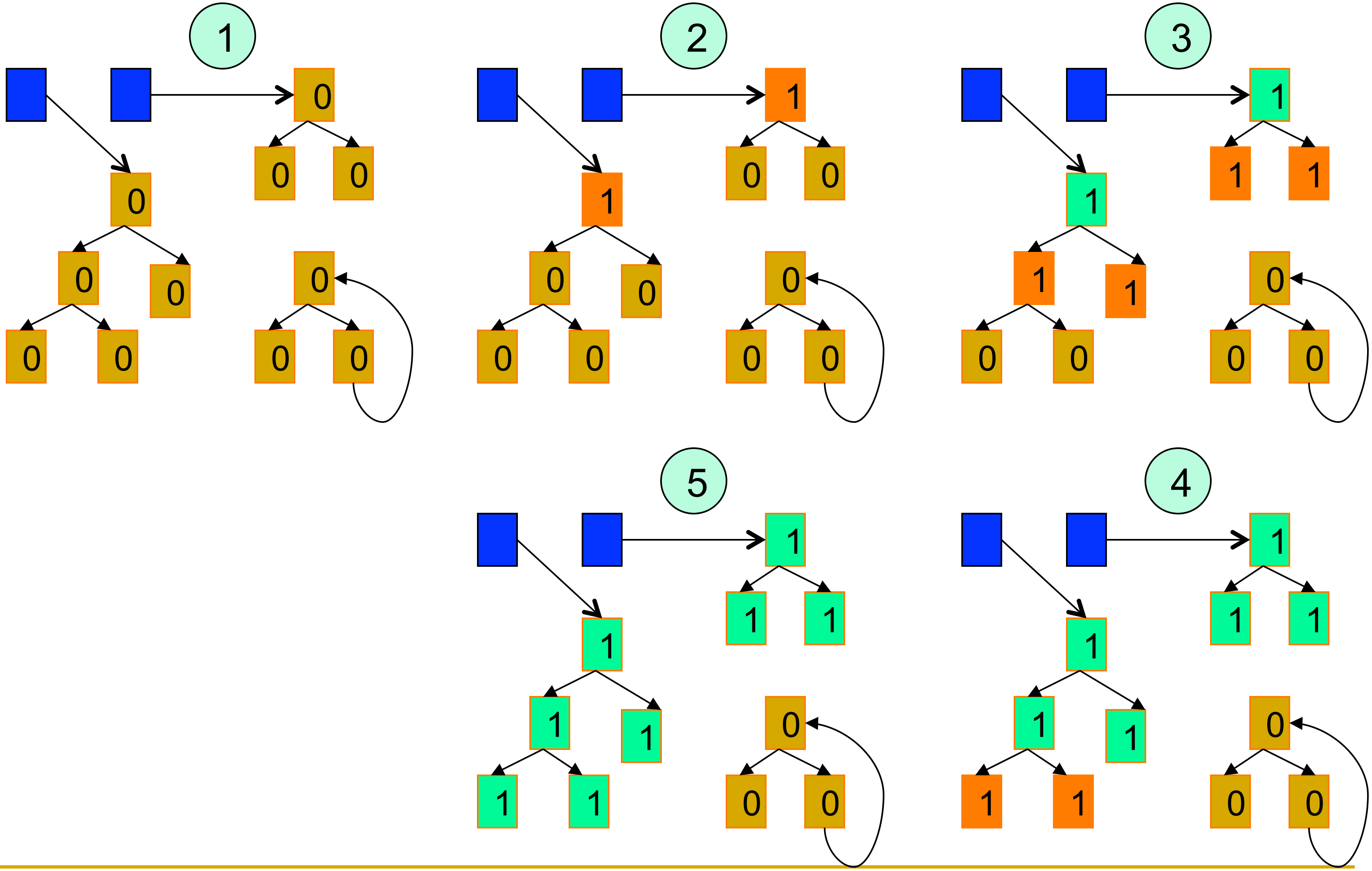
Mark-and-Sweep Garbage Collector

- Memory recycling steps
  - Program runs and requests memory allocations
  - GC traces and finds reachable objects
  - GC reclaims storage from unreachable objects
- Two phases
  - Marking reachable objects
  - Sweeping to reclaim storage
- Can reclaim unreachable cyclic data structures
- Stop-the-world algorithm

Mark-and-Sweep Algorithm - Mark

```
/* marking phase */
1. Start scanning from root set, mark all reachable objects
   (set reached-bit = 1), place them on the list Unscanned
2. while (Unscanned ≠ Φ) do
   { object o = delete(Unscanned);
     for (each object $o_1$ referenced in o) do
        { if (reached-bit($o_1$) == 0)
            { reached-bit($o_1$) = 1; place $o_1$ on Unscanned;}
        }
   }
```

Mark-and-Sweep GC Example - Mark

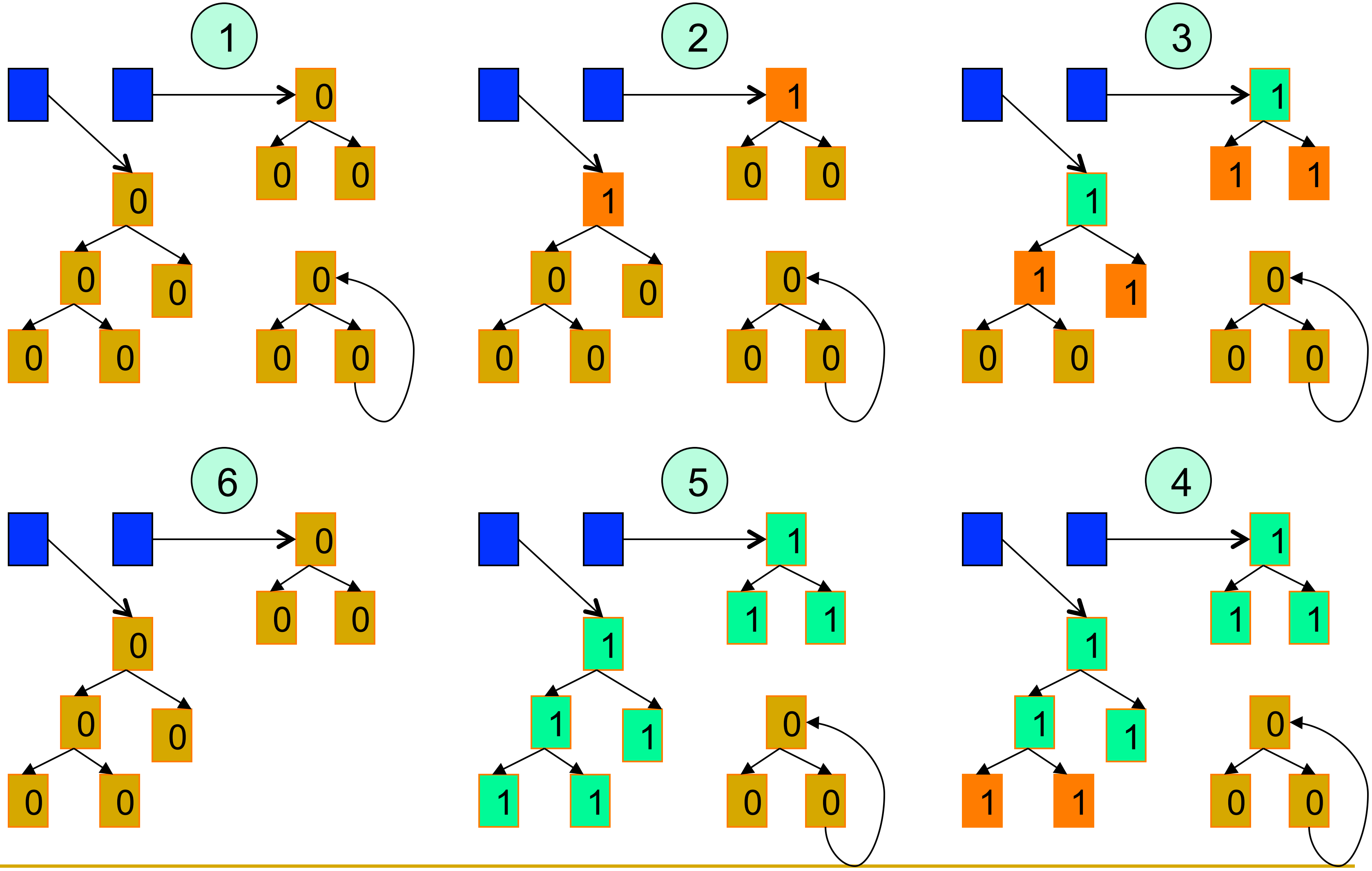


Mark-and-Sweep Algorithm - Sweep

- /* Sweeping phase, each object in the heap is inspected only once */

```
3.  Free = Φ;
    for (each object o in the heap) do
      { if (reached-bit(o) == 0)    add(Free, o);
        else reached-bit(o) = 0;
      }
```

Mark-and-Sweep GC Example - Sweep



Control-Flow Graph and Local Optimizations - Part 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization and why is it needed?
- Types of optimizations
- Basic blocks and control flow graphs
- Local optimizations
- Building a control flow graph
- Directed acyclic graphs and value numbering

Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)
- Optimizations may be classified as *local* and *global*

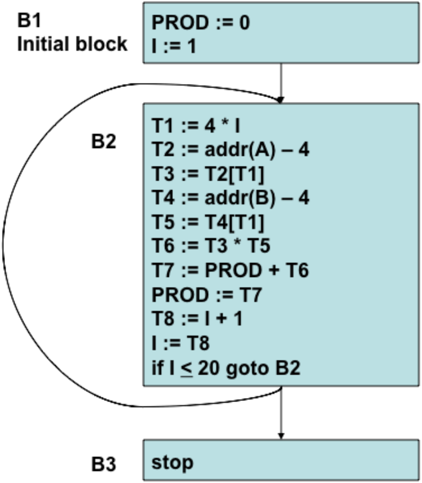
Local and Global Optimizations

- Local optimizations: within basic blocks
  - Local common subexpression elimination
  - Dead code (instructions that compute a value that is never used) elimination
  - Reordering computations using algebraic laws
- Global optimizations: on whole procedures/programs
  - Global common sub-expression elimination
  - Constant propagation and constant folding
  - Loop invariant code motion
  - Partial redundancy elimination
  - Loop unrolling and function inlining
  - Vectorization and Concurrentization

Basic Blocks and Control-Flow Graphs

- Basic blocks are sequences of intermediate code with a *single entry* and a single exit
- We consider the quadruple version of intermediate code here, to make the explanations easier
- Control flow graphs show control flow among basic blocks
- Basic blocks are represented as *directed acyclic blocks*(DAGs), which are in turn represented using the value-numbering method applied on quadruples
- Optimizations on basic blocks

Example of Basic Blocks and Control Flow Graph



B1
Initial block

```
PROD := 0
I := 1
```

B2

```
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
```

B3

```
stop
```

High level language code:

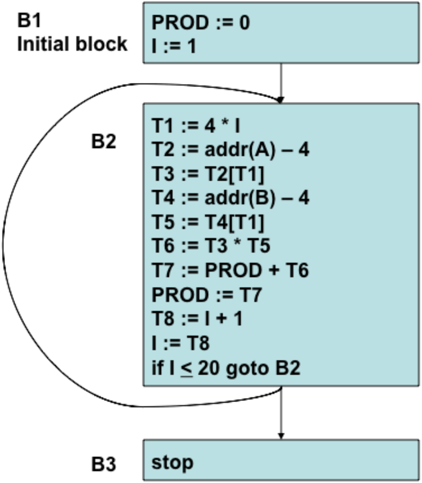
```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```

Algorithm for Partitioning into Basic Blocks

1. Determine the set of *leaders*, the first statements of basic blocks
    - The first statement is a leader
    - Any statement which is the target of a conditional or unconditional *goto* is a leader
    - Any statement which immediately follows a *conditional goto* is a leader
2. A leader and all statements which follow it upto but not including the next leader (or the end of the procedure), is the basic block corresponding to that leader
3. Any statements, not placed in a block, can never be executed, and may now be removed, if desired

Example of Basic Blocks and CFG



B1
Initial block

```
PROD := 0
I := 1
```

B2

```
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
```

B3

```
stop
```

High level language code:

```
{ PROD = 0;
  for ( I = 1; I <= 20; I++)
    PROD = PROD + A[I] * B[I];
}
```

```
PROD := 0
I := 1
T1 := 4 * I
T2 := addr(A) – 4
T3 := T2[T1]
T4 := addr(B) – 4
T5 := T4[T1]
T6 := T3 * T5
T7 := PROD + T6
PROD := T7
T8 := I + 1
I := T8
if I ≤ 20 goto B2
stop
```

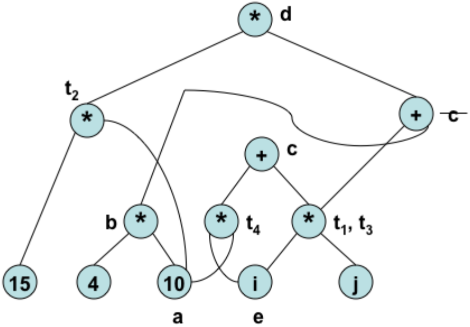
Control Flow Graph

- The nodes of the CFG are basic blocks
- One node is distinguished as the initial node
- There is a directed edge $B1 \longrightarrow B2$, if B2 can immediately follow B1 in some execution sequence; i.e.,
    - There is a conditional or unconditional jump from the last statement of B1 to the first statement of B2, or
    - B2 immediately follows B1 in the order of the program, and B1 does not end in an unconditional jump
- A basic block is represented as a record consisting of
    1. a count of the number of quadruples in the block
    2. a pointer to the leader of the block
    3. pointers to the predecessors of the block
    4. pointers to the successors of the block

  Note that jump statements point to basic blocks and not quadruples so as to make code movement easy

Example of a Directed Acyclic Graph (DAG)

1. $a = 10$
2. $b = 4 * a$
3. $t1 = i * j$
4. $c = t1 + b$
5. $t2 = 15 * a$
6. $d = t2 * c$
7. $e = i$
8. $t3 = e * j$
9. $t4 = i * a$
10. $c = t3 + t4$



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values:
  HashTable, ValnumTable, and *NameTable*
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

Control-Flow Graph and Local Optimizations - Part 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

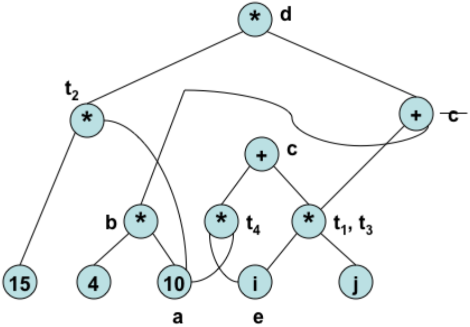
NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization and why is it needed? (in part 1)
- Types of optimizations (in part 1)
- Basic blocks and control flow graphs (in part 1)
- Local optimizations (in part 1)
- Building a control flow graph (in part 1)
- Directed acyclic graphs and value numbering

Example of a Directed Acyclic Graph (DAG)

1. $a = 10$
2. $b = 4 * a$
3. $t1 = i * j$
4. $c = t1 + b$
5. $t2 = 15 * a$
6. $d = t2 * c$
7. $e = i$
8. $t3 = e * j$
9. $t4 = i * a$
10. $c = t3 + t4$



Value Numbering in Basic Blocks

- A simple way to represent DAGs is via *value-numbering*
- While searching DAGs represented using pointers etc., is inefficient, *value-numbering* uses hash tables and hence is very efficient
- Central idea is to assign numbers (called value numbers) to expressions in such a way that two expressions receive the same number if the compiler can prove that they are equal for all possible program inputs
- We assume quadruples with binary or unary operators
- The algorithm uses three tables indexed by appropriate hash values:
  HashTable, ValnumTable, and *NameTable*
- Can be used to eliminate common sub-expressions, do constant folding, and constant propagation in basic blocks
- Can take advantage of commutativity of operators, addition of zero, and multiplication by one

Data Structures for Value Numbering

In the field *Namelist*, first name is the defining occurrence and replaces all other names with the same value number with itself (or its constant value)

HashTable entry
(indexed by expression hash value)

Expression	Value number

ValnumTable entry
(indexed by name hash value)

Name	Value number

NameTable entry
(indexed by value number)

Name list	Constant value	Constflag

Example of Value Numbering

HLL Program	Quadruples before Value-Numbering	Quadruples after Value-Numbering
$a = 10$	1. $a = 10$	1. $a = 10$
$b = 4 * a$	2. $b = 4 * a$	2. $b = 40$
$c = i * j + b$	3. $t1 = i * j$	3. $t1 = i * j$
$d = 15 * a * c$	4. $c = t1 + b$	4. $c = t1 + 40$
$e = i$	5. $t2 = 15 * a$	5. $t2 = 150$
$c = e * j + i * a$	6. $d = t2 * c$	6. $d = 150 * c$
	7. $e = i$	7. $e = i$
	8. $t3 = e * j$	8. $t3 = i * j$
	9. $t4 = i * a$	9. $t4 = i * 10$
	10. $c = t3 + t4$	10. $c = t1 + t4$
		(Instructions 5 and 8 can be deleted)

Running the algorithm through the example (1)

1. $a = 10$:
   - *a* is entered into *ValnumTable* (with a *vn* of 1, say) and into *NameTable* (with a constant value of 10)
2. $b = 4 * a$:
   - *a* is found in *ValnumTable*, its constant value is 10 in *NameTable*
     - We have performed *constant propagation*
     - $4 * a$ is evaluated to 40, and the quad is rewritten
     - We have now performed *constant folding*
     - *b* is entered into *ValnumTable* (with a *vn* of 2) and into *NameTable* (with a constant value of 40)
3. $t1 = i * j$:
   - *i* and *j* are entered into the two tables with new *vn* (as above), but with no constant value
   - $i * j$ is entered into *HashTable* with a new *vn*
   - $t1$ is entered into *ValnumTable* with the same *vn* as $i * j$

Running the algorithm through the example (2)

4. Similar actions continue till $e = i$
    - e gets the same *vn* as i
5. $t3 = e * j$:
    - e and i have the same *vn*
    - hence, $e * j$ is detected to be the same as $i * j$
    - since $i * j$ is already in the HashTable, we have found a *common subexpression*
    - from now on, all uses of $t3$ can be replaced by $t1$
    - quad $t3 = e * j$ can be deleted
6. $c = t3 + t4$:
    - $t3$ and $t4$ already exist and have *vn*
    - $t3 + t4$ is entered into *HashTable* with a new *vn*
    - this is a reassignment to c
    - c gets a different *vn*, same as that of $t3 + t4$
7. Quads are renumbered after deletions

Example: *HashTable* and *ValNumTable*

HashTable

Expression	Value-Number
$i * j$	5
$t1 + 40$	6
$150 * c$	8
$i * 10$	9
$t1 + t4$	11

ValNumTable

Name	Value-Number
a	1
b	2
i	3
j	4
$t1$	5
c	6,11
$t2$	7
d	8
e	3
$t3$	5
$t4$	10

Handling Commutativity etc.

- When a search for an expression $i + j$ in *HashTable* fails, try for $j + i$
- If there is a quad $x = i + 0$, replace it with $x = i$
- Any quad of the type, $y = j * 1$ can be replaced with $y = j$
- After the above two types of replacements, value numbers of x and y become the same as those of i and j, respectively
- Quads whose LHS variables are used later can be marked as *useful*
- All unmarked quads can be deleted at the end

Handling Array References

Consider the sequence of quads:

1. $X = A[i]$
2. $A[j] = Y$: i and j could be the same
3. $Z = A[i]$: in which case, $A[i]$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = A[i];\ A[j] = Y;\ Z = X$
- When $A[j] = Y$ is processed during value numbering, ALL references to array A so far are searched in the tables and are marked KILLED - this kills quad 1 above
- When processing $Z = A[i]$, killed quads not used for CSE
- Fresh table entries are made for $Z = A[i]$
- However, if we know apriori that $i \neq j$, then $A[i]$ can be used for CSE

Handling Pointer References

Consider the sequence of quads:

1. $X = *p$
2. $*q = Y$: p and q could be pointing to the same object
3. $Z = *p$: in which case, $*p$ is not a common subexpression here

- The above sequence cannot be replaced by: $X = *p;\ *q = Y;\ Z = X$
- Suppose no pointer analysis has been carried out
  - p and q can point to *any* object in the basic block
  - Hence, When $*q = Y$ is processed during value numbering, ALL table entries created so far are marked KILLED - this kills quad 1 above as well
  - When processing $Z = *p$, killed quads not used for CSE
  - Fresh table entries are made for $Z = *p$

Handling Pointer References and Procedure Calls

- However, if we know apriori which objects *p* and *q* point to, then table entries corresponding to only those objects need to killed
- Procedure calls are similar
- With no dataflow analysis, we need to assume that a procedure call can modify any object in the basic block
    - changing call-by-reference parameters and global variables within procedures will affect other variables of the basic block as well
- Hence, while processing a procedure call, ALL table entries created so far are marked KILLED
- Sometimes, this problem is avoided by making a procedure call a separate basic block

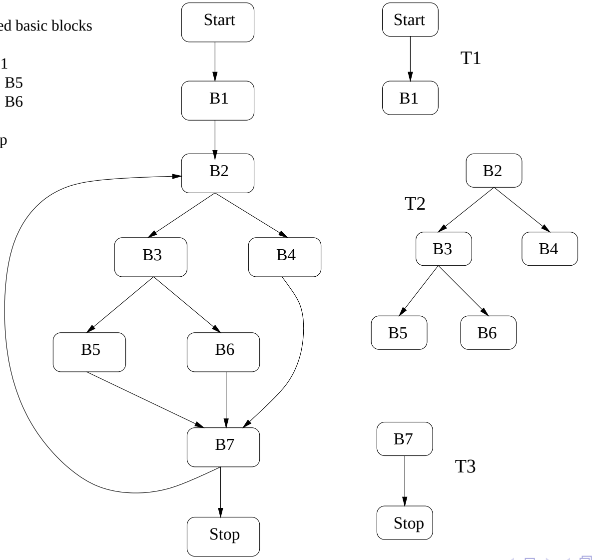
Extended Basic Blocks

- A sequence of basic blocks $B_1, B_2, ..., B_k$, such that B_i is the unique predecessor of $B_{i+1} (i \leq i < k)$, and B_1 is either the start block or has no unique predecessor
- Extended basic blocks with shared blocks can be represented as a tree
- Shared blocks in extended basic blocks require scoped versions of tables
- The new entries must be purged and changed entries must be replaced by old entries
- Preorder traversal of extended basic block trees is used

Extended Basic Blocks and their Trees

Extended basic blocks

Start, B1
B2, B3, B5
B2, B3, B6
B2, B4
B7, Stop



Value Numbering with Extended Basic Blocks

```
function $visit\text{-}ebb\text{-}tree(e)$ // $e$ is a node in the tree
begin
   // From now on, the new names will be entered with a new scope into the tables.
   // When searching the tables, we always search beginning with the current scope
   // and move to enclosing scopes. This is similar to the processing involved with
   // symbol tables for lexically scoped languages
   $value\text{-}number(e.B)$;
   // Process the block $e.B$ using the basic block version of the algorithm
   if $(e.left \neq null)$ then $visit\text{-}ebb\text{-}tree(e.left)$;
   if $(e.right \neq null)$ then $visit\text{-}ebb\text{-}tree(e.right)$;
   remove entries for the new scope from all the tables
   and undo the changes in the tables of enclosing scopes;
end

begin // main calling loop
   for each tree $t$ do $visit\text{-}ebb\text{-}tree(t)$;
   // $t$ is a tree representing an extended basic block
end
```

Machine Code Generation - 1

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Machine code generation – main issues
- Samples of generated code
- Two Simple code generators
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

Code Generation – Main Issues (1)

- Transformation:
  - Intermediate code → m/c code (binary or assembly)
  - We assume quadruples and CFG to be available
- Which instructions to generate?
  - For the quadruple A = A+1, we may generate
    - Inc A *or*
    - Load A, R1
      Add #1, R1
      Store R1, A
  - One sequence is faster than the other (cost implication)

Code Generation – Main Issues (2)

- In which order?
  - Some orders may use fewer registers and/or may be faster
- Which registers to use?
  - Optimal assignment of registers to variables is difficult to achieve
- Optimize for memory, time or power?
- Is the code generator easily retargetable to other machines?
  - Can the code generator be produced automatically from specifications of the machine?

Samples of Generated Code

- B = A[i]

```
Load   i, R1 // R1 = i
Mult    R1,4,R1// R1 = R1*4
// each element of array
// A is 4 bytes long
Load   A(R1), R2// R2=(A+R1)
Store  R2, B// B = R2
```

- X[j] = Y

```
Load  Y, R1// R1 = Y
Load  j, R2// R2 = j
Mult    R2, 4, R2// R2=R2*4
Store  R1, X(R2)// X(R2)=R1
```

- X = *p

```
Load   p, R1
Load   0(R1), R2
Store  R2, X
```

- *q = Y

```
Load   Y, R1
Load   q, R2
Store   R1, 0(R2)
```

- if X < Y goto L

```
Load  X, R1
Load  Y, R2
Cmp   R1, R2
Bltz     L
```

Samples of Generated Code – Static Allocation (no JSR instruction)

Three Adress Code

// Code for function F1
action code seg 1
call F2
action code seg 2
Halt

// Code for function F2
action code seg 3
return

Activation Record for F1 (48 bytes)

0	return address
4	data array A
40	variable x
44	variable y

Activation Record for F2 (76 bytes)

0	return address
4	parameter 1
	data array B
72	variable m

Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move val1, 652
256:    Jump 400 // Call F2
264:    Action code seg 2
280:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1
440:    Jump @648
        ...
```

```
//Activation record for F1
//from 600-647
600:    //return address
604:    //space for array A
640:    //space for variable x
644:    //space for variable y
//Activation record for F2
//from 648-723
648:    //return address
652:   // parameter 1
656:    //space for array B
        ...
720:    //space for variable m
```

Machine Code Generation - 2

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Mach. code generation – main issues (in part 1)
- Samples of generated code
- Two Simple code generators
- Optimal code generation
    - Sethi-Ullman algorithm
    - Dynamic programming based algorithm
    - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

Samples of Generated Code – Static Allocation (no JSR instruction)

Three Adress Code

// Code for function F1
action code seg 1
call F2
action code seg 2
Halt

// Code for function F2
action code seg 3
return

Activation Record for F1 (48 bytes)

0	return address
4	data array A
40	variable x
44	variable y

Activation Record for F2 (76 bytes)

0	return address
4	parameter 1
	data array B
72	variable m

Samples of Generated Code – Static Allocation (no JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now store return address
240:    Move #264, 648
252:    Move val1, 652
256:    Jump 400 // Call F2
264:    Action code seg 2
280:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1
440:    Jump @648
        ...
```

```
//Activation record for F1
//from 600-647
600:    //return address
604:    //space for array A
640:    //space for variable x
644:    //space for variable y
//Activation record for F2
//from 648-723
648:    //return address
652:    // parameter 1
656:    //space for array B
        ...
720:    //space for variable m
```

Samples of Generated Code – Static Allocation (with JSR instruction)

Three Adress Code

// Code for function F1
action code seg 1
call F2
action code seg 2
Halt

// Code for function F2
action code seg 3
return

Activation Record for F1 (44 bytes)

0	data array A
36	variable x
40	variable y

Activation Record for F2 (72 bytes)

0	data array B
68	variable m

return address need not be stored

Samples of Generated Code – Static Allocation (with JSR instruction)

```
// Code for function F1
200:    Action code seg 1
// Now jump to F2, return addr
// is stored on hardware stack
240:    JSR 400 // Call F2
248:    Action code seg 2
268:    Halt
        ...
// Code for function F2
400:    Action code seg 3
// Now return to F1 (addr 248)
440:    return
        ...
```

```
//Activation record for F1
//from 600-643
600:    //space for array A
636:    //space for variable x
640:    //space for variable y
//Activation record for F2
//from 644-715
644:    //space for array B
        ...
712:    //space for variable m
```

Samples of Generated Code – Dynamic Allocation (no JSR instruction)

Three Adress Code

```
// Code for function F1
action code seg 1
call F2
action code seg 2
return
```

```
// Code for function F2
action code seg 3
call F1
action code seg 4
call F2
action code seg 5
return
```

Activation Record for F1 (68 bytes)

0	return address
4	local data and other information
64	

Activation Record for F2 (96 bytes)

0	return address
4	parameter 1
	local data and other information
92	

Samples of Generated Code – Dynamic Allocation (no JSR instruction)

```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #96, SP
238:    Move #258, @SP
246:    Move val1, @SP+4
250:    Jump 300
258:    Sub #96, SP
266:    Action code seg 2
296:    Jump   @SP
```

```
//Code for F2
300:    Action code seg 3
340:    Add #68, SP
348:    Move #364, @SP
356:    Jump 200
364:    Sub #68, SP
372:    Action code seg 4
400:    Add #96, SP
408:    Move #424, @SP
416:    Move val2, @SP+4
420:    Jump 300
428:    Sub #96, SP
436:    Action code seg 5
480:    Jump @SP
```

Samples of Generated Code – Dynamic Allocation (with JSR instruction)

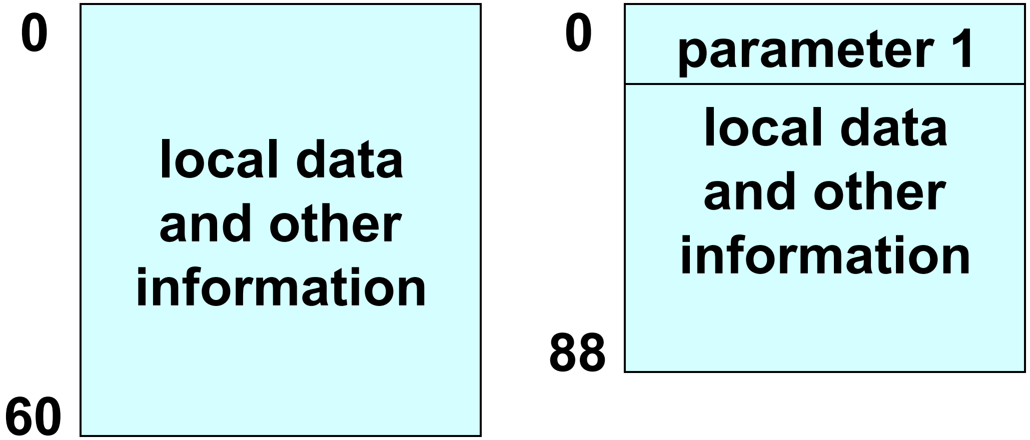
Three Adress Code

// Code for function F1
action code seg 1
call F2
action code seg 2
return

// Code for function F2
action code seg 3
call F1
action code seg 4
call F2
action code seg 5
return

Activation Record for F1 (64 bytes)

Activation Record for F2 (92 bytes)



Samples of Generated Code – Dynamic Allocation (with JSR instruction)

```
//Initialization
100:    Move #800, SP
        ...
//Code for F1
200:    Action code seg 1
230:    Add #92, SP
238:    Move val1, @SP
242:    JSR 290
250:    Sub #92, SP
258:    Action code seg 2
286:    return
```

```
//Code for F2
290:    Action code seg 3
330:    Add #64, SP
338:    JSR 200
346:    Sub #64, SP
354:    Action code seg 4
382:    Add #92, SP
390:    Move val2, @SP
394:    JSR 290
402:    Sub #92, SP
410:    Action code seg 5
454:    return
```

A Simple Code Generator – Scheme A

- Treat each quadruple as a ‘macro’
  - Example: The quad *A := B + C* will result in

```
Load B, R1   OR   Load B, R1
Load C, R2
Add R2, R1        Add C, R1
Store R1, A       Store R1, A
```

  - Results in inefficient code
    - Repeated load/store of registers
  - Very simple to implement

A Simple Code Generator – Scheme B

- Track values in registers and reuse them
  - If any operand is already in a register, take advantage of it
  - Register descriptors
    - Tracks <register, variable name> pairs
    - A single register can contain values of multiple names, if they are all copies
  - Address descriptors
    - Tracks <variable name, location> pairs
    - A single name may have its value in multiple locations, such as, memory, register, and stack

A Simple Code Generator – Scheme B

- Leave computed result in a register as long as possible
- Store only at the end of a basic block or when that register is needed for another computation
    - A variable is live at a point, if it is used later, possibly in other blocks – obtained by dataflow analysis
    - On exit from a basic block, store only live variables which are not in their memory locations already (use address descriptors to determine the latter)
    - If liveness information is not known, assume that all variables are live at all times

Example

- A := B+C
  - If B and C are in registers R1 and R2, then generate
    - *ADD R2,R1* (cost = 1, result in R1)
      - legal only if B is *not live* after the statement
  - If R1 contains B, but C is in memory
    - *ADD C,R1* (cost = 2, result in R1) **or**
    - *LOAD C, R2*

      ADD R2,R1 (cost = 3, result in R1)
      - legal only if B is *not live* after the statement
      - attractive if the value of C is subsequently used (it can be taken from R2)

Next Use Information

- Next use info is used in code generation and register allocation
- Next use of *A* in quad *i* is *j* if

Quad *i* : A = ... (assignment to A)

↓ (control flows from *i* to *j* with no assignments to A)

Quad *j* : = A op B (usage of A)

- In computing next use, we assume that on exit from the basic block
    - All temporaries are considered non-live
    - All programmer defined variables (and non-temps) are live
- Each procedure/function call is assumed to start a basic block
- Next use is computed on a backward scan on the quads in a basic block, starting from the end
- Next use information is stored in the symbol table

Example of computing Next Use

3	T1 := 4 * I	T1 – (nlv, lu 0, nu 5), I – (lv, lu 3, nu 10)
4	T2 := addr(A) – 4	T2 – (nlv, lu 0, nu 5)
5	T3 := T2[T1]	T3 – (nlv, lu 0, nu 8), T2 – (nlv, lu 5, nnu), T1 – (nlv, lu 5, nu 7)
6	T4 := addr(B) – 4	T4 – (nlv, lu 0, nu 7)
7	T5 := T4[T1]	T5 – (nlv, lu 0, nu 8), T4 – (nlv, lu 7, nnu), T1 – (nlv, lu 7, nnu)
8	T6 := T3 * T5	T6 – (nlv, lu 0, nu 9),T3 – (nlv, lu 8, nnu), T5 – (nlv, lu 8, nnu)
9	PROD := PROD + T6	PROD – (lv, lu 9, nnu), T6 – (nlv, lu 9, nnu)
10	I := I + 1	I – (lv, lu 10, nu 11)
11	if I ≤ 20 goto 3	I – (lv, lu 11, nnu)

Scheme B – The algorithm

- We deal with one basic block at a time
- We assume that there is no global register allocation
- For each quad *A := B op C* do the following
  - Find a location *L* to perform *B op C*
    - Usually a register returned by *GETREG*() (could be a mem loc)
  - Where is *B*?
    - *B*' , found using address descriptor for *B*
    - Prefer register for *B*' , if it is available in memory and register
    - Generate *Load B' , L* (if *B*' is not in *L*)
  - Where is *C*?
    - *C*' , found using address descriptor for *C*
    - Generate *op C' , L*
  - Update descriptors for *L* and *A*
  - If *B/C* have no next uses, update descriptors to reflect this information

Function *GETREG*( )

Finds *L* for computing *A := B op C*

1. If *B* is in a register (say *R*), R holds no other names, and
   - *B* has no next use, and *B* is not live after the block, then return *R*
2. Failing (1), return an empty register, if available
3. Failing (2)
   - If *A* has a next use in the block, OR
     if *B op C* needs a register (*e.g.*, *op* is an indexing operator)
     - Use a *heuristic* to find an occupied register
       - **a register whose contents are referenced farthest in future, or**
       - **the number of next uses is smallest etc.**
     - *Spill* it by generating an instruction, *MOV R, mem*
       - ***mem* is the memory location for the variable in *R***
       - **That variable is not already in *mem***
     - Update *Register* and *Address* descriptors
4. If *A* is not used in the block, or no suitable register can be found
   - Return a memory location for *L*

Example

T,U, and V are temporaries - not live at the end of the block
W is a non-temporary - live at the end of the block, 2 registers

Statements	Code Generated	Register Descriptor	Address Descriptor
T := A * B	Load A,R0 Mult B, R0	R0 contains T	T in R0
U := A + C	Load A, R1 Add C, R1	R0 contains T R1 contains U	T in R0 U in R1
V := T - U	Sub R1, R0	R0 contains V R1 contains U	U in R1 V in R0
W := V * U	Mult R1, R0	R0 contains W	W in R0
	Store R0, W		W in memory (restored)

Optimal Code Generation
- The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form *op R,R* or *op M,R*
- **Always computes the left subtree into a register and reuses it immediately**
- Two phases
  - Labelling phase
  - Code generation phase

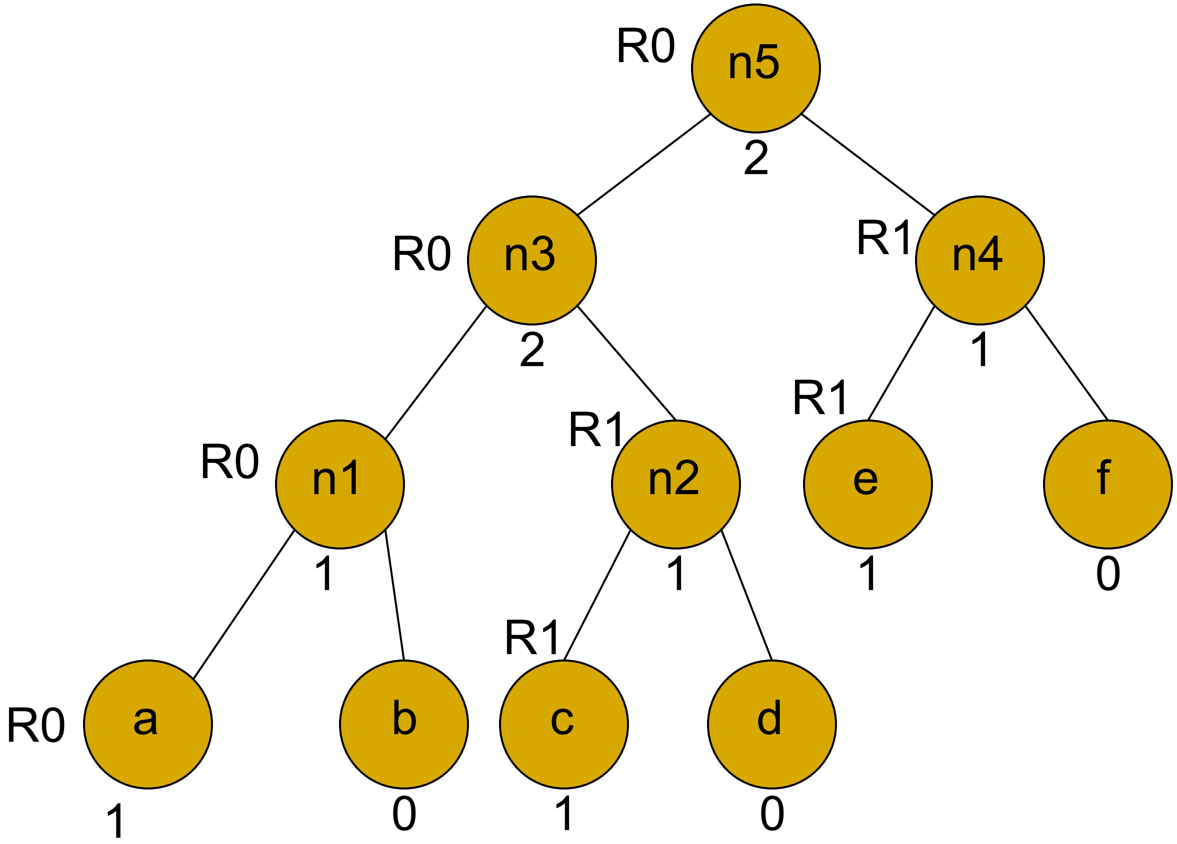
The Labelling Algorithm

- Labels each node of the tree with an integer:
  - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
  - Consider binary trees
- For leaf nodes
  - ***if*** **n** is the leftmost child of its parent ***then***

    label(n) := 1 *else* label(n) := 0
- For internal nodes
  - **label(n) = max (l_1, l_2), if l_1<> l_2**

    = l_1 + 1, if l_1 = l_2

Labelling - Example



Machine Code Generation - 3

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Mach. code generation – main issues (in part 1)
- Samples of generated code (in part 2)
- Two Simple code generators (in part 2)
- Optimal code generation
  - Sethi-Ullman algorithm
  - Dynamic programming based algorithm
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

Optimal Code Generation
- The Sethi-Ullman Algorithm

- Generates the shortest sequence of instructions
  - Provably optimal algorithm (w.r.t. length of the sequence)
- Suitable for expression trees (basic block level)
- Machine model
  - All computations are carried out in registers
  - Instructions are of the form *op R,R* or *op M,R*
- **Always computes the left subtree into a register and reuses it immediately**
- Two phases
  - Labelling phase
  - Code generation phase

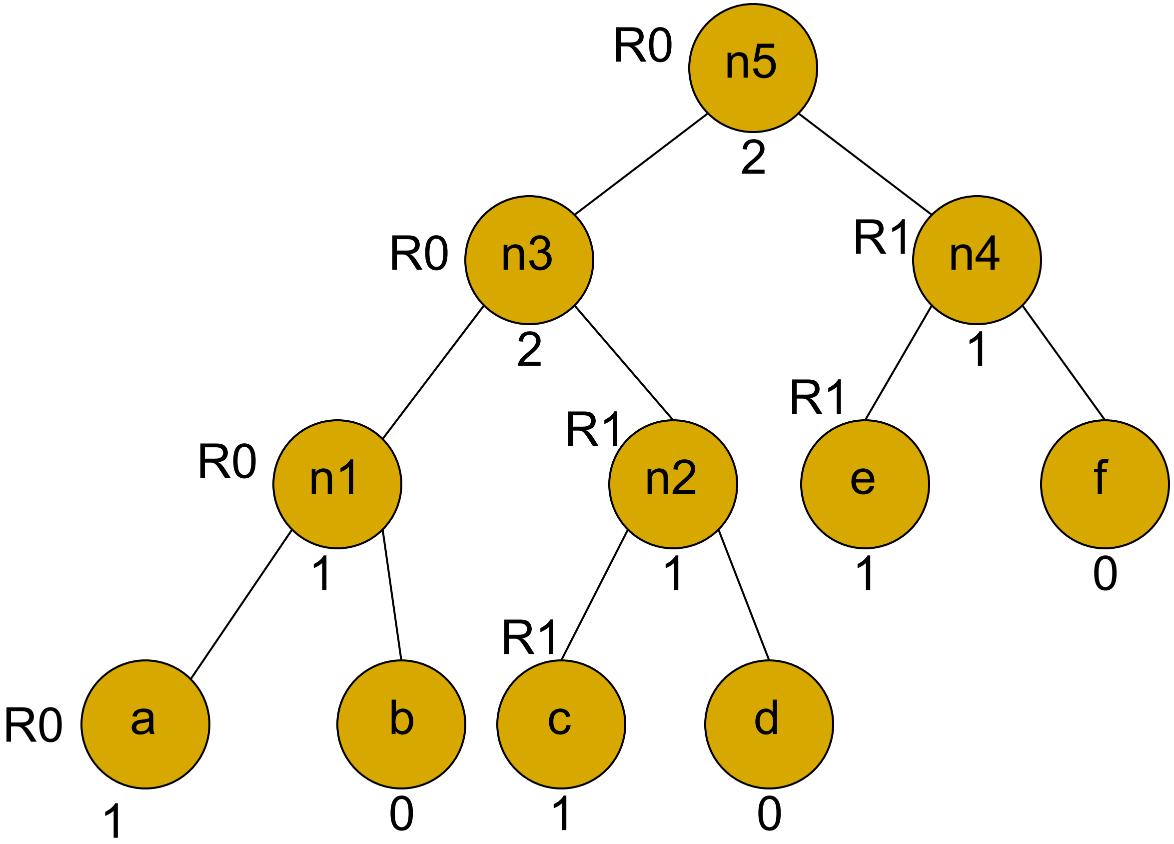
The Labelling Algorithm

- Labels each node of the tree with an integer:
    - fewest no. of registers required to evaluate the tree with no intermediate stores to memory
    - Consider binary trees
- For leaf nodes
    - ***if*** **n** is the leftmost child of its parent ***then***

      label(n) := 1 *else* label(n) := 0
- For internal nodes
    - **label(n) = max** (l_1, l_2)**, if** $l_1 <> l_2$

      = $l_1 + 1$**, if** $l_1 = l_2$

Labelling - Example

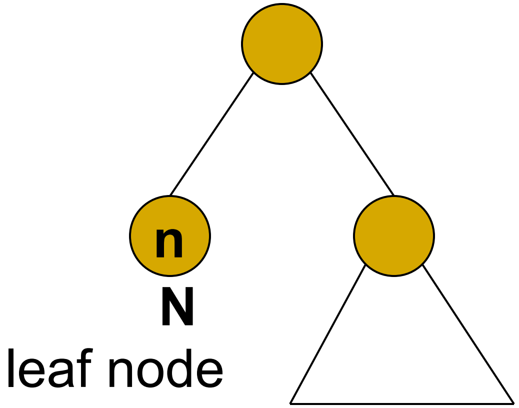


Code Generation Phase – Procedure GENCODE(n)

- RSTACK – stack of registers, $R_0,\ldots,R_{(r-1)}$
- TSTACK – stack of temporaries, $T_0,T_1,\ldots$
- A call to Gencode(n) generates code to evaluate a tree T, rooted at node n, into the register top (RSTACK) ,and
  - the rest of RSTACK remains in the same state as the one before the call
- A swap of the top two registers of RSTACK is needed at some points in the algorithm to ensure that a node is evaluated into the same register as its left child.

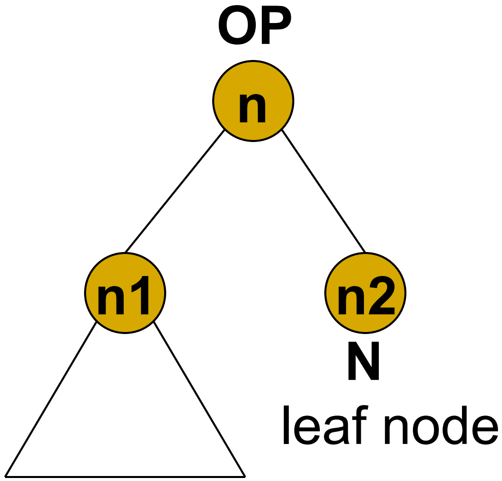
The Code Generation Algorithm (1)

```
Procedure gencode(n);
{ /* case 0 */
  if
      n is a leaf representing
      operand N and is the
      leftmost child of its parent
  then
      print(LOAD N, top(RSTACK))
```



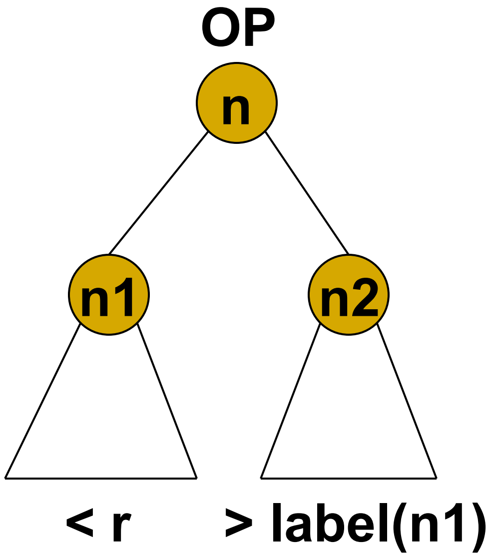
The Code Generation Algorithm (2)

```
/* case 1 */
else if
  n is an interior node with operator
  OP, left child n1, and right child n2
then
  if  label(n2) == 0 then {
      let N be the operand for n2;
      gencode(n1);
      print(OP N, top(RSTACK));
  }
```



The Code Generation Algorithm (3)

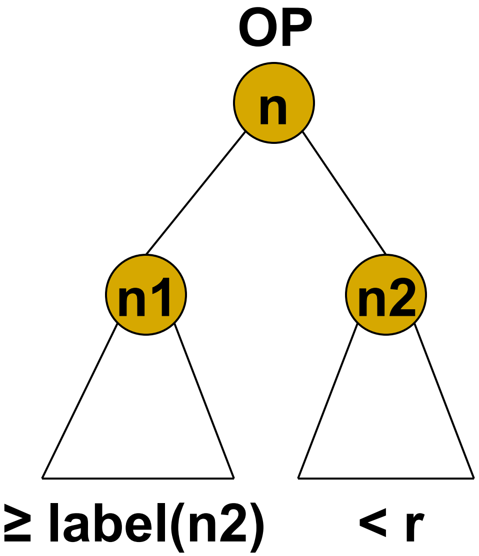
```
/* case 2 */
else if ((1 ≤ label(n1) < label(n2))
          and( label(n1) < r))
then {
  swap(RSTACK); gencode(n2);
  R := pop(RSTACK); gencode(n1);
  /* R holds the result of n2 */
  print(OP R, top(RSTACK));
  push (RSTACK,R);
  swap(RSTACK);
  }
```



The swap() function ensures that a node is evaluated into the same register as its left child

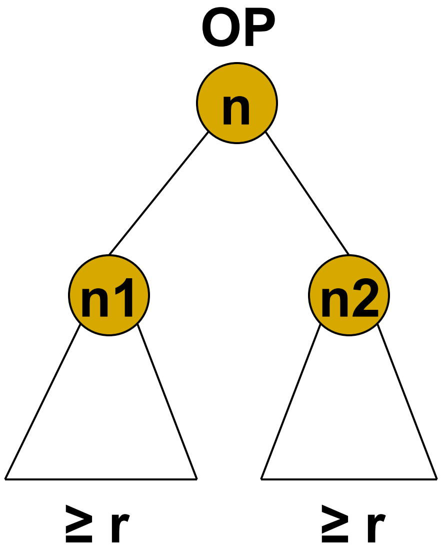
The Code Generation Algorithm (4)

```
/* case 3 */
else if ((1 ≤ label(n2) ≤ label(n1))
            and( label(n2) < r))
then {
   gencode(n1);
   R := pop(RSTACK); gencode(n2);
   /* R holds the result of n1 */
   print(OP  top(RSTACK), R);
   push (RSTACK,R);
   }
```



The Code Generation Algorithm (5)

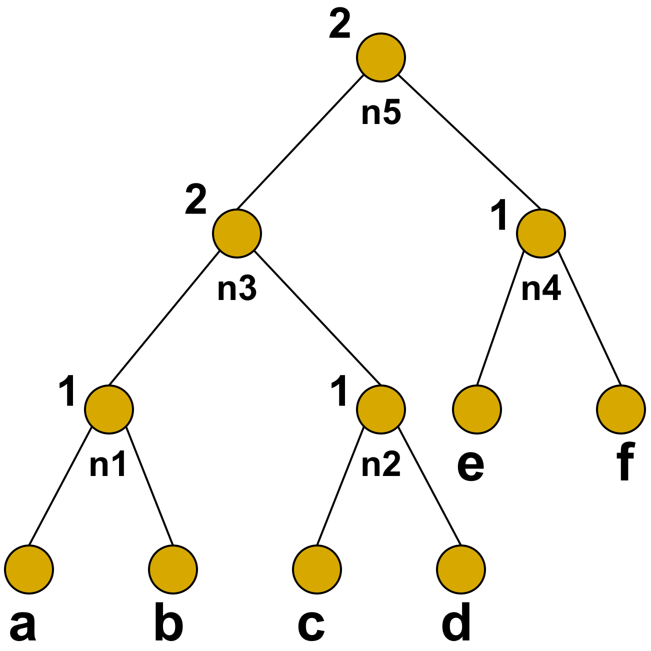
```
/* case 4, both labels are ≥ r */
else {
   gencode(n2); T:= pop(TSTACK);
   print(LOAD top(RSTACK), T);
   gencode(n1);
   print(OP T, top(RSTACK));
   push(TSTACK, T);
   }
}
```



Code Generation Phase – Example 1

No. of registers = r = 2

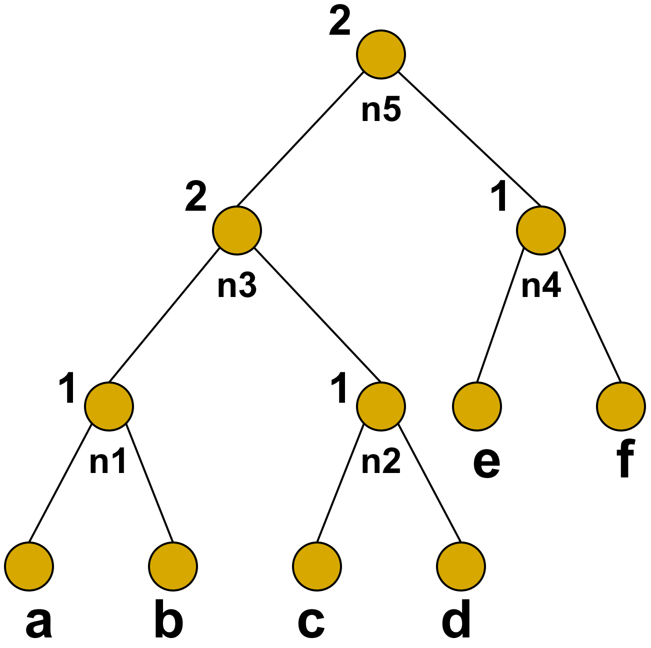
n5 → n3 → n1 → a → Load a, R0
→ op_{n1} b, R0
→ n2 → c → Load c, R1
→ op_{n2} d, R1
→ op_{n3} R1, R0
→ n4 → e → Load e, R1
→ op_{n4} f, R1
→ op_{n5} R1, R0



Code Generation Phase – Example 2

No. of registers = r = 1.
Here we choose *rst* first so that *lst* can be computed into R0 later (case 4)

n5 → n4 → e → Load e, R0
→ op_{n4} f, R0
→ Load R0, T0 {release R0}
→ n3 → n2 → c → Load c, R0
→ op_{n2} d, R0
→ Load R0, T1 {release R0}
→ n1 → a → Load a, R0
→ op_{n1} b, R0
→ op_{n3} T1, R0 {release T1}
→ op_{n5} T0, R0 {release T0}

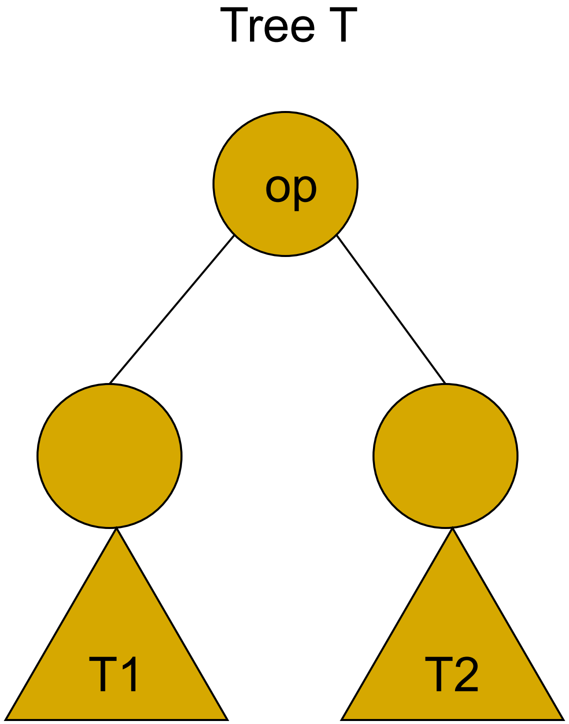


Dynamic Programming based Optimal Code Generation for Trees

- Broad class of register machines
  - *r* interchangeable registers, $R_0,\ldots,R_{r-1}$
  - Instructions of the form $R_i := E$
    - If *E* involves registers, R_i must be one of them
    - $R_i := M_j$, $R_i := R_i \ op\ R_j$, $R_i := R_i \ op\ M_j$, $R_i := R_j$, $M_i := R_j$
- Based on principle of contiguous evaluation
- Produces optimal code for trees (basic block level)
- Can be extended to include a different cost for each instruction

Contiguous Evaluation

- First evaluate subtrees of *T* that need to be evaluated into memory. Then,
  - Rest of *T1, T2, op*, in that order, *OR,*
  - Rest of *T2, T1, op*, in that order
- Part of *T1*, part of *T2*, part of *T1* again, etc., is *not* contiguous evaluation
- Contiguous evaluation is optimal!
  - No higher cost and no more registers than optimal evaluation



The Algorithm (1)

1. Compute in a bottom-up manner, for each node *n* of *T,* an array of costs, *C*
   - *C[i]* = *min* cost of computing the complete subtree rooted at *n*, assuming *i* registers to be available
     - Consider each machine instruction that matches at *n* and consider all possible contiguous evaluation orders (using dynamic programming)
     - Add the cost of the instruction that matched at node *n*

The Algorithm (2)

- Using *C*, determine the subtrees that must be computed into memory (based on cost)
- Traverse *T*, and emit code
  - memory computations first
  - rest later, in the order needed to obtain optimal cost
- Cost of computing a tree into memory = cost of computing the tree using all registers + 1 (store cost)

An Example

Max no. of registers = 2

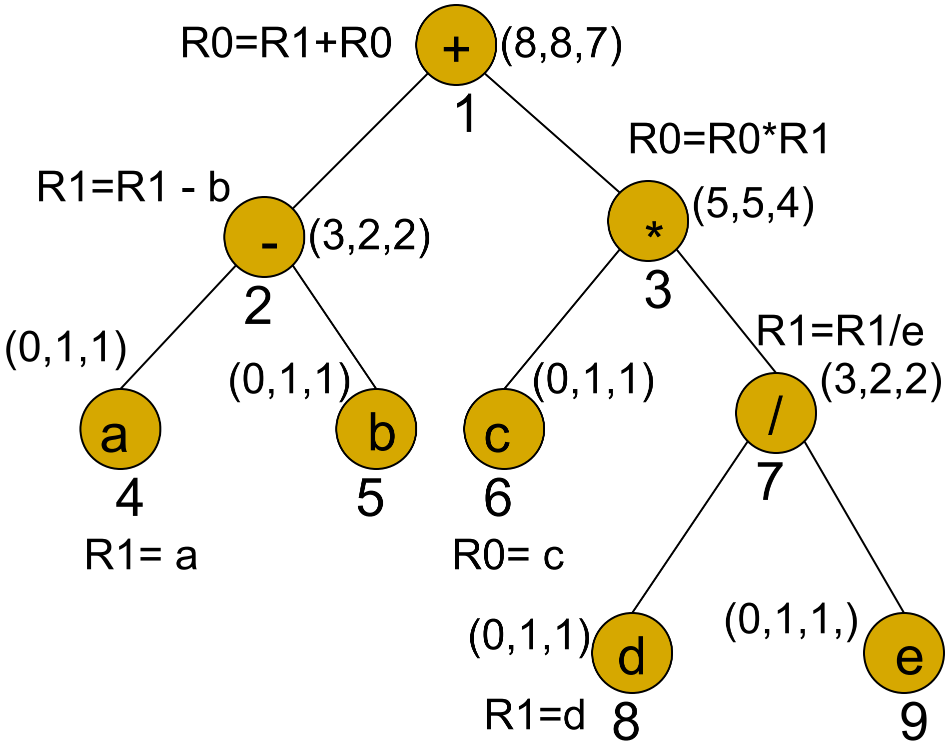
Node 2: matching instructions

R*i* = R*i* – M (*i* = 0,1) and
R*i* = R*i* – R*j* (*i,j* = 0,1)

C2[1] = C4[1] + C5[0] + 1
= 1+0+1 = 2

C2[2] = Min{ C4[2] + C5[1] + 1,
C4[2] + C5[0] + 1,
C4[1] + C5[2] + 1,
C4[1] + C5[1] + 1,
C4[1] + C5[0] + 1}
= Min{1+1+1,1+0+1,1+1+1,
1+1+1,1+0+1}
= Min{3,2,3,3,2} = 2

C2[0] = 1+ C2[2] = 1+2 = 3



R0 = c
R1 = d
R1 = R1 / e
R0 = R0 * R1
R1 = a
R1 = R1 – b
R0 = R1 + R0

Generated sequence of instructions

Example – continued
Cost of computing node 3 with 2 registers

#regs for node 6	#regs for node 7	cost for node 3
2	0	1+3+1 = 5
2	1	1+2+1 = 4
1	0	1+3+1 = 5
1	1	1+2+1 = 4
1	2	1+2+1 = 4
	min value	4

Cost of computing with 1 register = 5 (row 4, red)
Cost of computing into memory = 4 + 1 = 5

Triple = (5,5,4)

Example – continued
Traversal and Generating Code

Min cost for node 1=7, Instruction: R0 := R1+R0

Compute RST(3) with 2 regs into R0

Compute LST(2) into R1

For node 3, instruction: R0 := R0 * R1

Compute RST(7) with 2 regs into R1

Compute LST(6) into R0

For node 7, instruction: R1 := R1 / e

Compute RST(9) into memory (already available)

Compute LST(8) into R1

For node 8, instruction: R1 := d

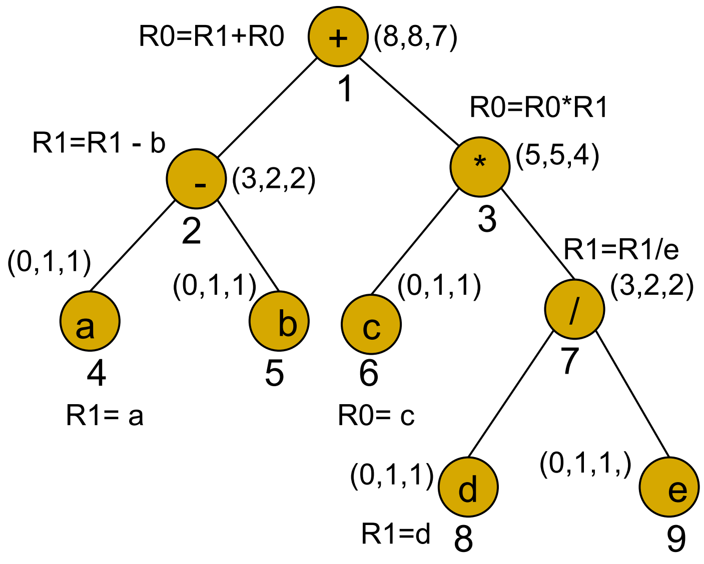
For node 6, instruction: R0 := c

For node 2, instruction: R1 := R1 – b

Compute RST(5) into memory (available already)

Compute LST(4) into R1

For node 4, instruction: R1 := a



Machine Code Generation - 4

Y. N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Mach. code generation – main issues (in part 1)
- Samples of generated code (in part 2)
- Two Simple code generators (in part 2)
- Optimal code generation
  - Sethi-Ullman algorithm (in part 3)
  - Dynamic programming based algorithm (in part 3)
  - Tree pattern matching based algorithm
- Code generation from DAGs
- Peephole optimizations

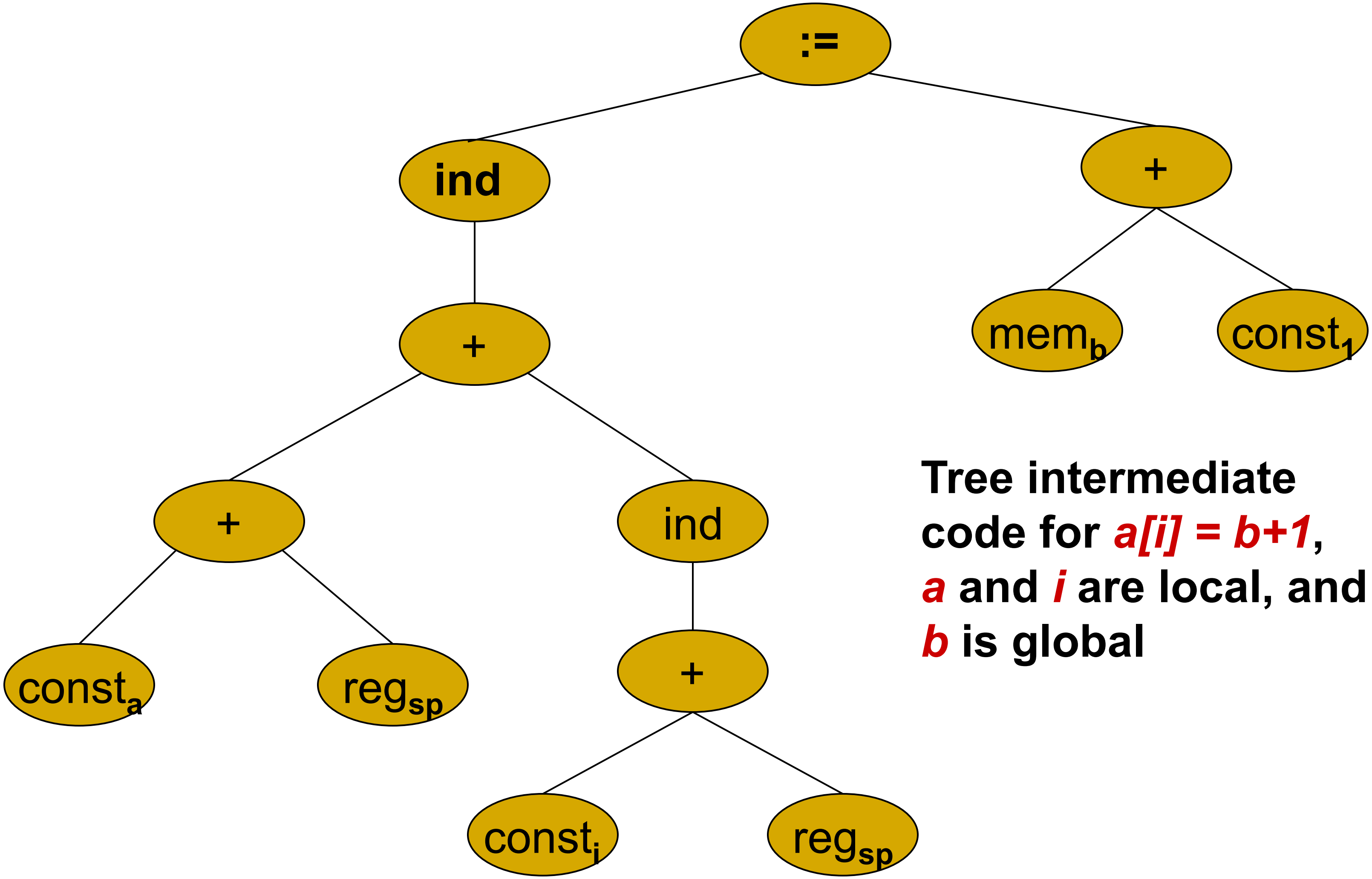
Code Generation based on Dynamic Programming - Limitations

- Several instructions require even-odd register pairs – (R_0,R_1), (R_2,R_3), etc.
  - example: multiplication in x86
  - may require non-contiguous evaluation to ensure optimality
  - cannot be handled by DP

Code Generation by Tree Rewriting

- Caters to complex instruction sets and very general machine models
- Can produce locally optimal code (basic block level)
- Non-contiguous evaluation orders are possible without sacrificing optimality
- Easily retargetable to different machines
- Automatic generation from specifications is possible

Example



Tree intermediate code for *a[i] = b+1*, *a* and *i* are local, and *b* is global

Some Tree Rewriting Rules and Associated Actions

1. reg$_i$ ← const$_a$ { Load #a, reg$_i$ }
2. reg$_i$ ← +(reg$_i$, reg$_j$) { Add reg$_i$, reg$_j$ }
3. reg$_i$ ← ind (+(const$_c$, reg$_j$)) { Load #c(reg$_j$), reg$_i$ }
4. reg$_i$ ← +(reg$_i$, ind (+(const$_c$, reg$_j$)))
   { Add #c(reg$_j$), reg$_i$ }
5. reg$_i$ ← mem$_a$ { Load b, reg$_i$ }

6. reg$_i$ ← +(reg$_i$, const$_1$) { Inc reg$_i$ }
7. mem ← :=(ind (reg$_i$) , reg$_j$) { Load reg$_j$, *reg$_i$ }

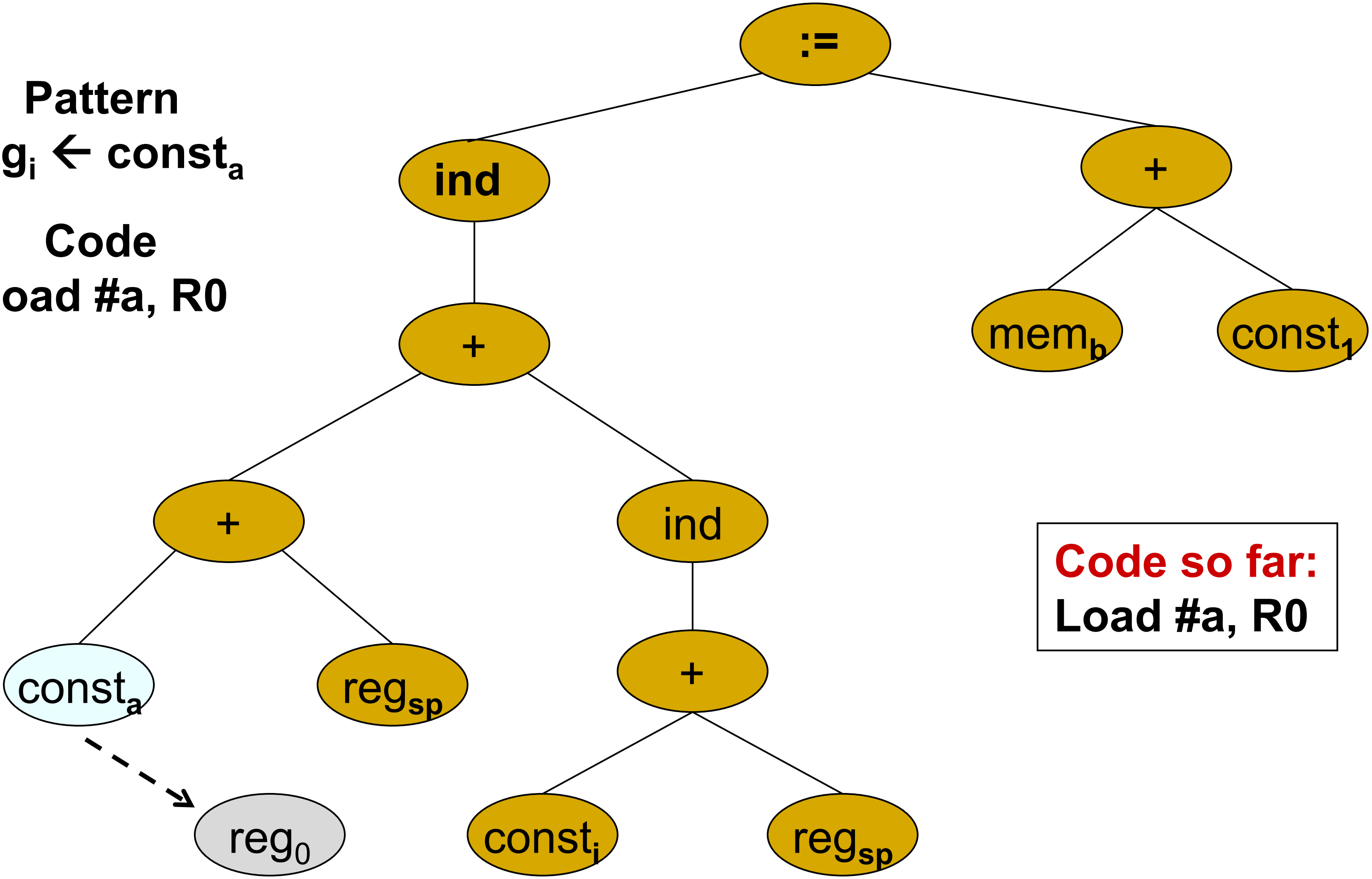
Match #1

Pattern

$reg_i \leftarrow const_a$

Code

Load #a, R0



Code so far:
Load #a, R0

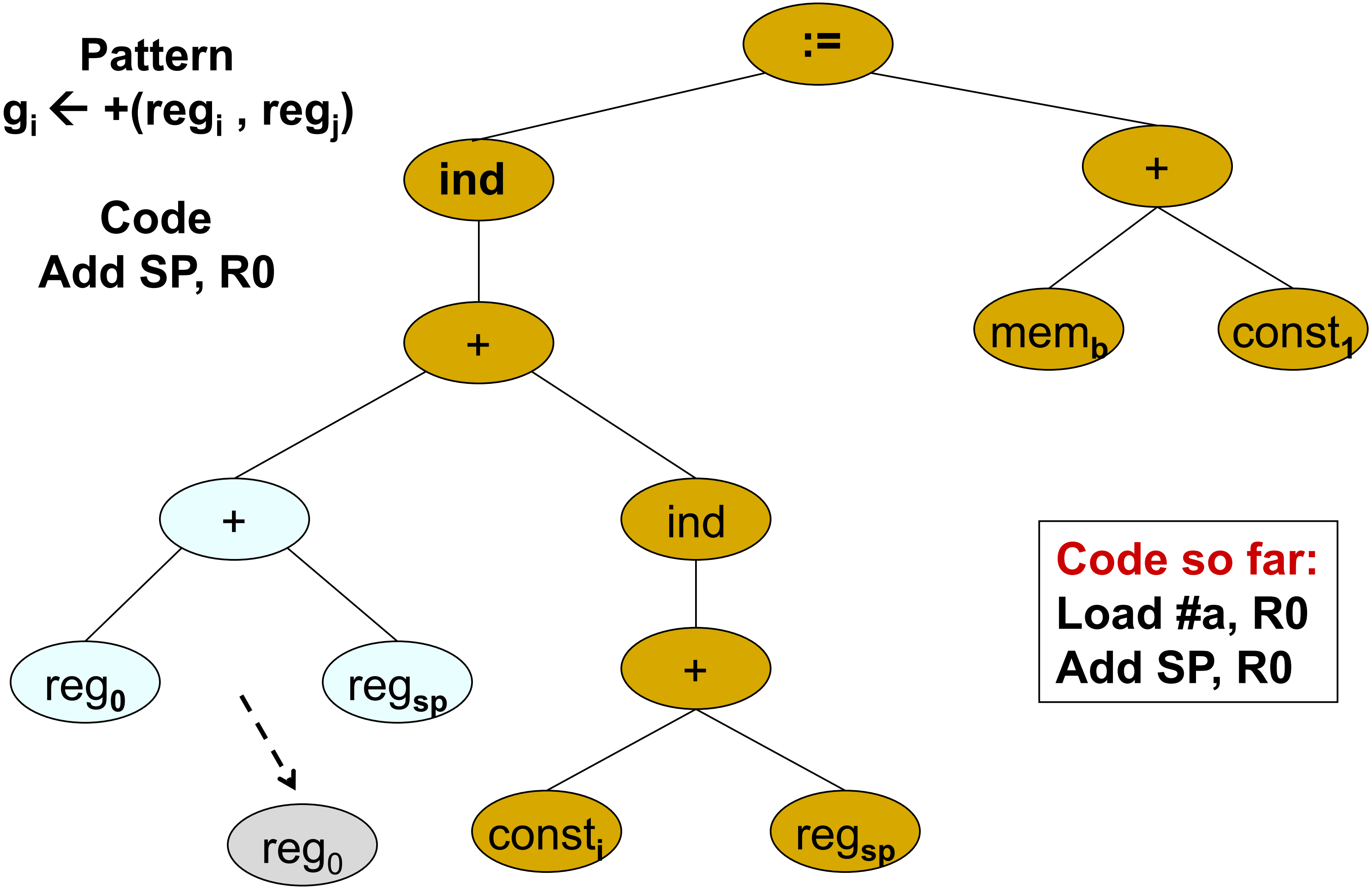
Match #2

Pattern

$reg_i \leftarrow +(reg_i\ ,\ reg_j)$

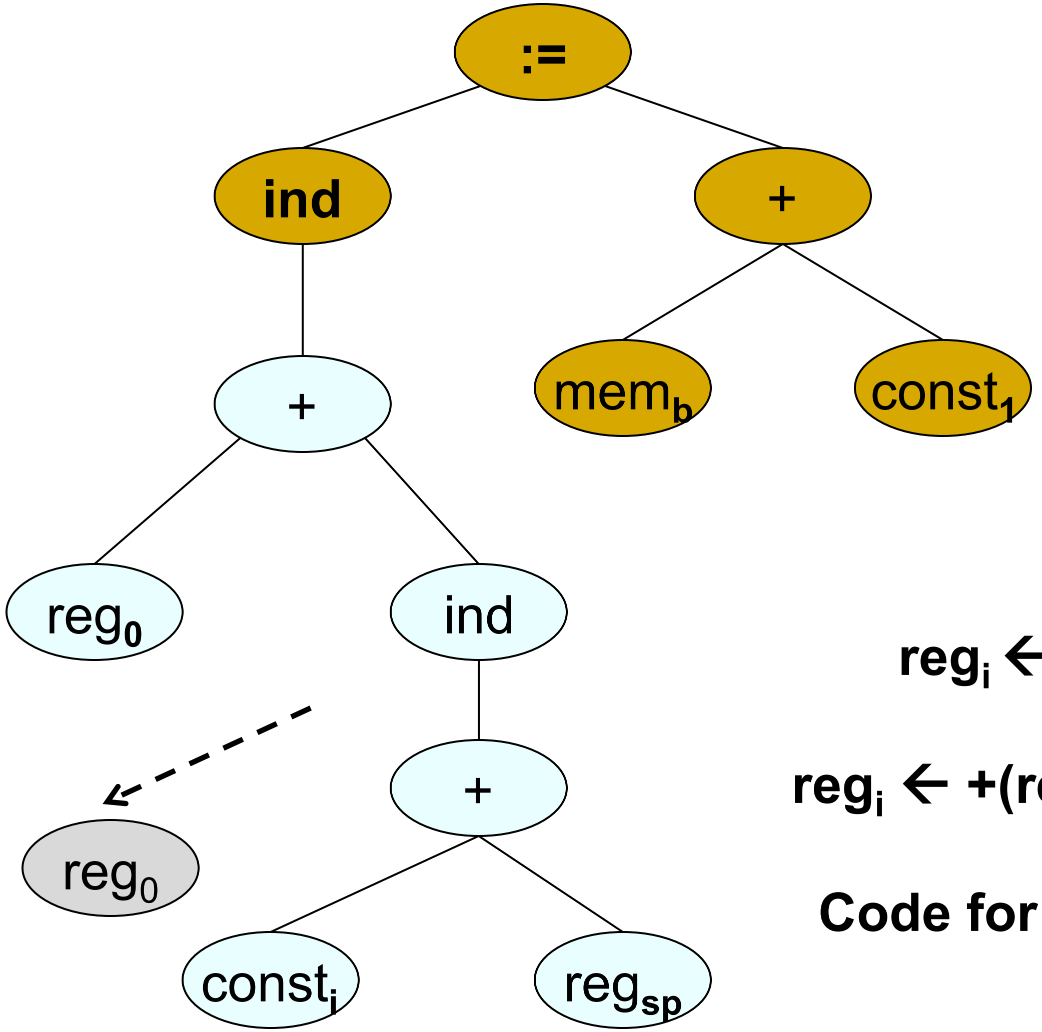
Code

Add SP, R0



Code so far:
Load #a, R0
Add SP, R0

Match #3



Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0

Pattern

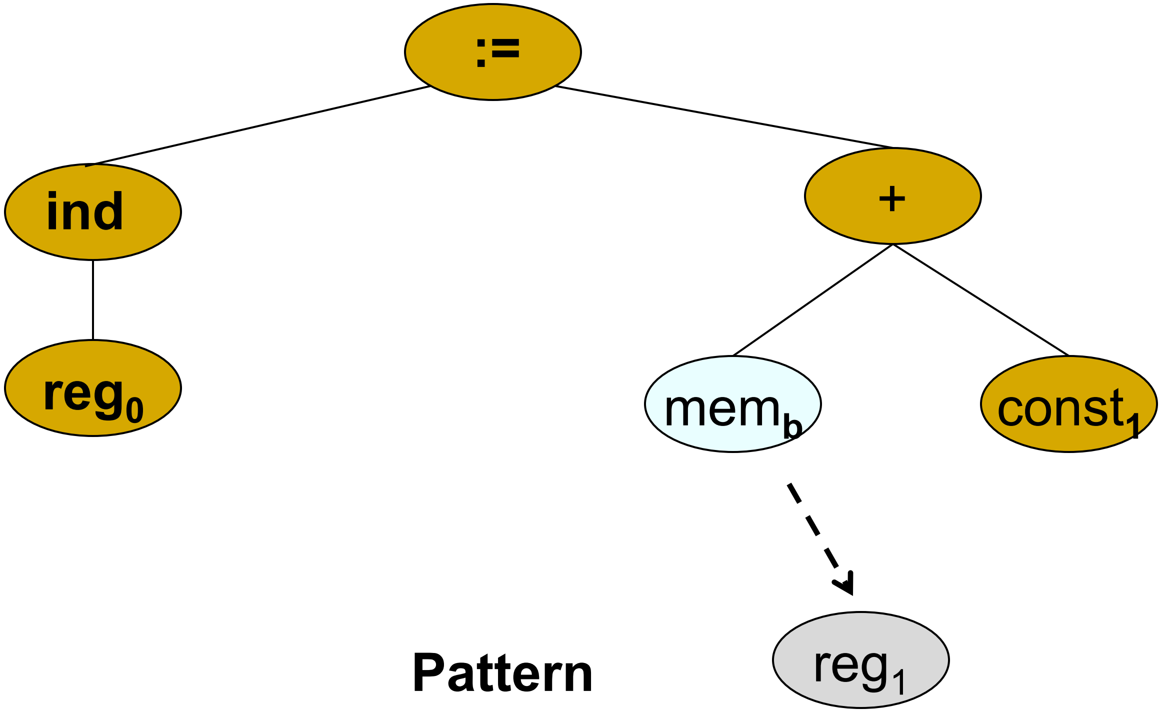
$reg_i \leftarrow ind(+(const_c, reg_j))$

OR

$reg_i \leftarrow +(reg_i, ind(+(const_c, reg_j)))$

Code for 2nd alternative (chosen)
Add #i(SP), R0

Match #4

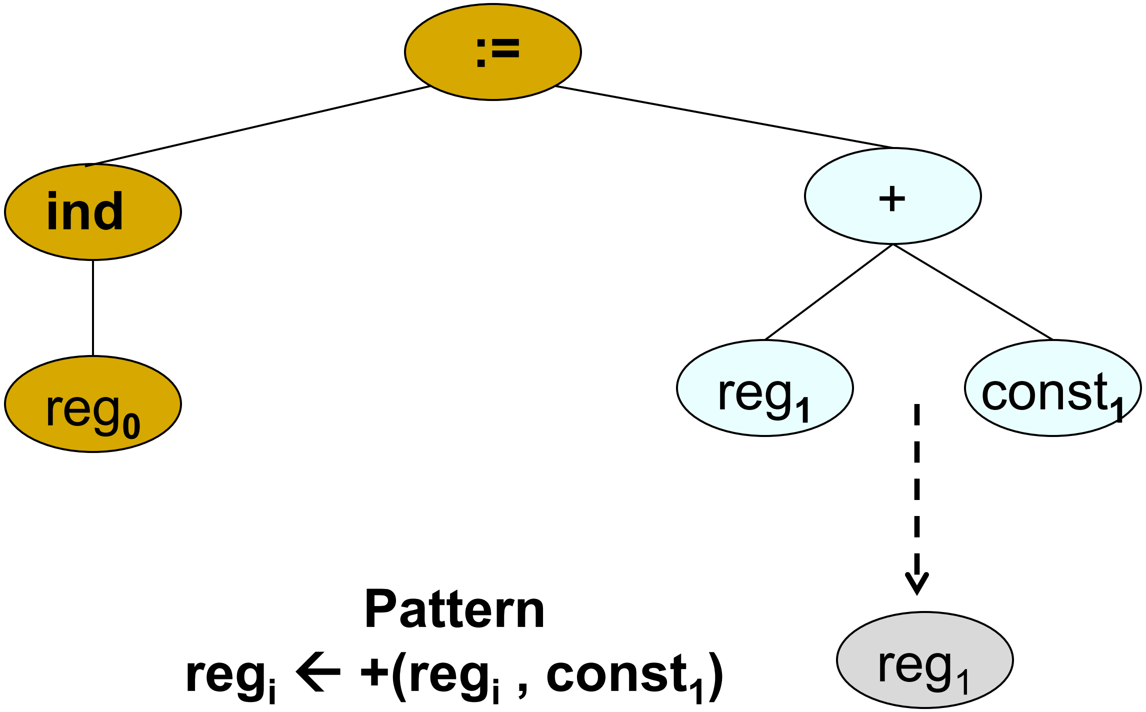


Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1

Pattern
$reg_i \leftarrow mem_a$

Code
Load b, R1

Match #5

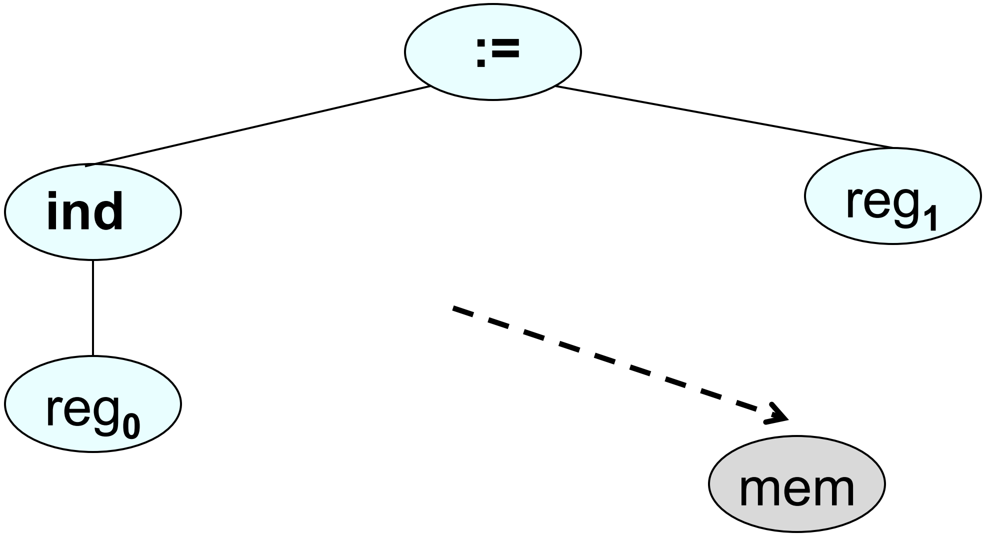


Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1
Inc R1

Pattern
$\mathbf{reg_i \leftarrow +(reg_i\ ,\ const_1)}$

Code
Inc R1

Match #6



Code so far:
Load #a, R0
Add SP, R0
Add #i(SP), R0
Load b, R1
Inc R1
Load R1, *R0

Pattern
mem ← :=(ind (reg_i) , reg_j)

Code
Load R1, *R0

Code Generator Generators (CGG)

- Based on tree pattern matching and dynamic programming
- Accept tree patterns, associated costs, and semantic actions (for register allocation and object code emission)
- Produce tree matchers that produce a cover of minimum cost
- Make two passes
    - First pass is a bottom-up pass and finds a set of patterns that cover the tree with minimum cost
    - Second pass executes the semantic actions associated with the minimum cost patterns at the nodes they matched
- Twig, BURG, and IBURG are such CGGs

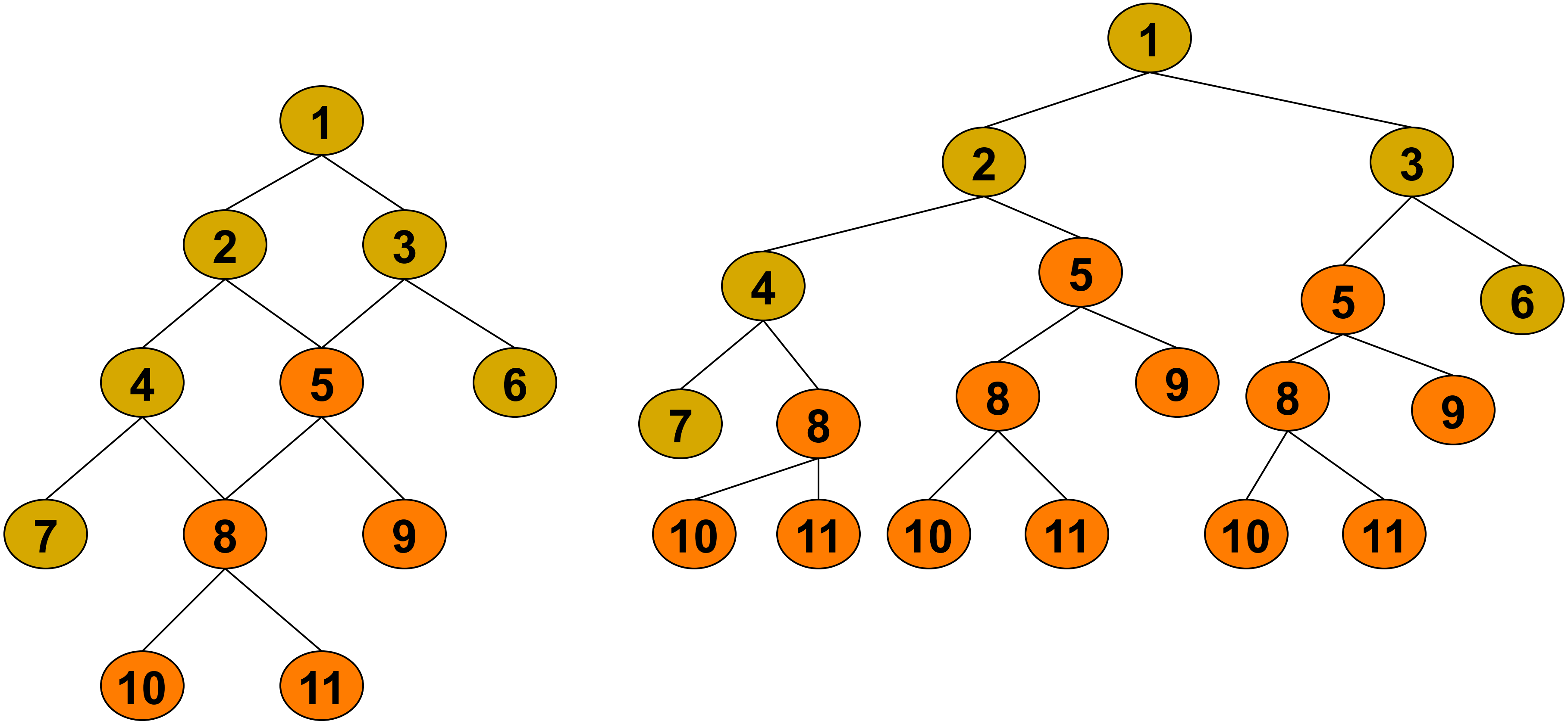
Code Generator Generators (2)

- IBURG
  - Uses dynamic programming (DP) at compile time
  - Costs can involve arbitrary computations
  - The matcher is hard coded
- TWIG
  - Uses a table-driven tree pattern matcher based on Aho-Corasick string pattern matcher
  - High overheads, could take $O(n^2)$ time, n being the number of nodes in the subject tree
  - Uses DP at compile time
  - Costs can involve arbitrary computations
- BURG
  - Uses BURS (bottom-up rewrite system) theory to move DP to compile-compile time (matcher generation time)
  - Table-driven, more complex, but generates optimal code in $O(n)$ time
  - Costs must be constants

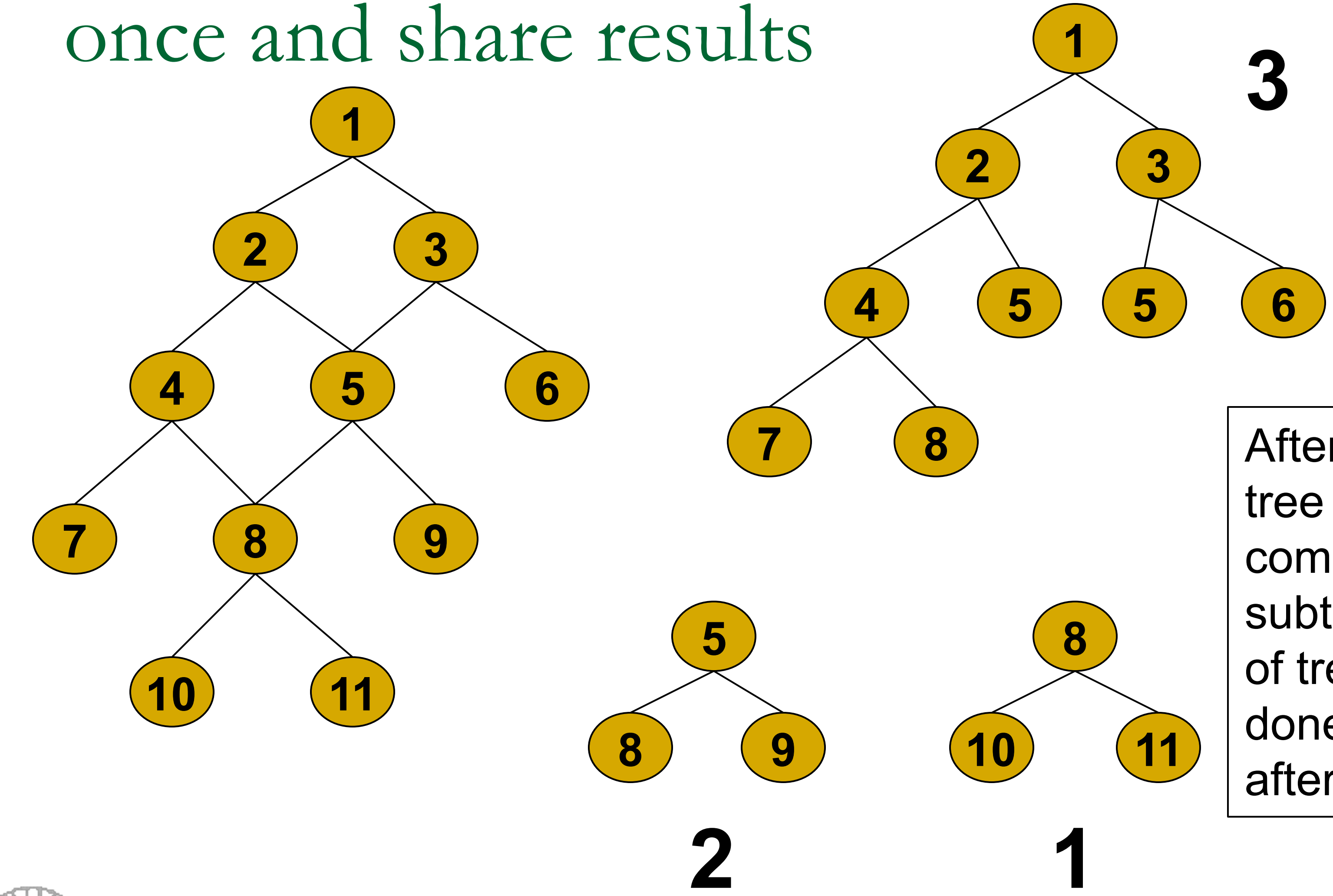
Code Generation from DAGs

- Optimal code generation from DAGs is NP-Complete
- DAGs are divided into trees and then processed
- We may replicate shared trees
  - Code size increases drastically
- We may store result of a tree (root) into memory and use it in all places where the tree is used
  - May result in sub-optimal code

DAG example: Duplicate shared trees



DAG example: Compute shared trees once and share results



After computing tree 1, the computation of subtree 4-7-8 of tree 3 can be done before or after tree 2

Peephole Optimizations

- Simple but effective local optimization
- Usually carried out on machine code, but intermediate code can also benefit from it
- Examines a sliding window of code (peephole), and replaces it by a shorter or faster sequence, if possible
- Each improvement provides opportunities for additional improvements
- Therefore, repeated passes over code are needed

Peephole Optimizations

- Some well known peephole optimizations
  - eliminating redundant instructions
  - eliminating unreachable code
  - eliminating jumps over jumps
  - algebraic simplifications
  - strength reduction
  - use of machine idioms

Elimination of Redundant Loads and Stores

Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Store R0, X
```

Store instruction can be deleted

Basic block B

```
Load X, R0
{no modifications
to X or R0 here}
Load X, R0
```

Second Load instr can be deleted

Basic block B

```
Store R0, X
{no modifications
to X or R0 here}
Load X, R0
```

Load instruction can be deleted

Basic block B

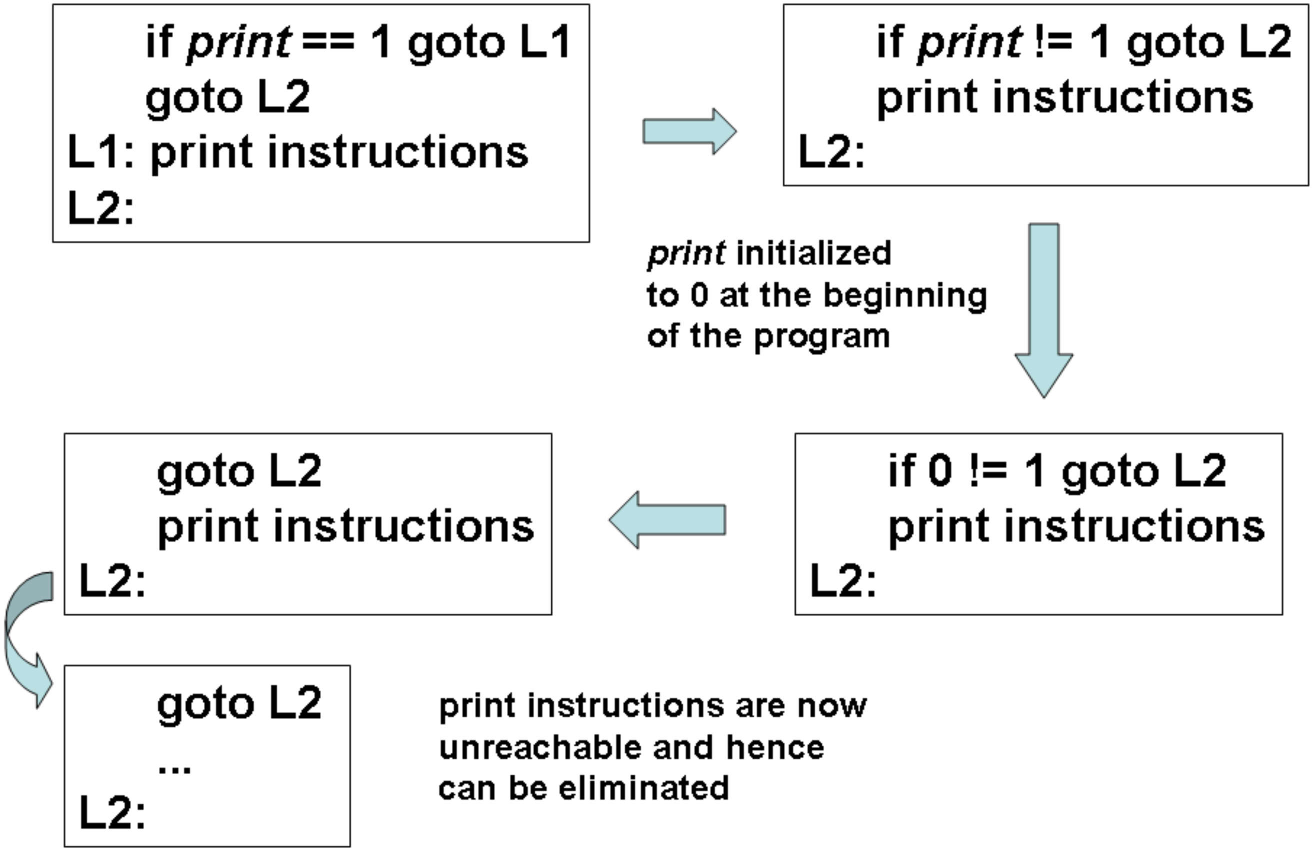
```
Store R0, X
{no modifications
to X or R0 here}
Store R0, X
```

Second Store instr can be deleted

Eliminating Unreachable Code

- An unlabeled instruction immediately following an unconditional jump may be removed
  - May be produced due to debugging code introduced during development
  - Or due to updates to programs (changes for fixing bugs) without considering the whole program segment

Eliminating Unreachable Code



```
    if print == 1 goto L1
    goto L2
L1: print instructions
L2:
```

```
    if print != 1 goto L2
    print instructions
L2:
```

print initialized to 0 at the beginning of the program

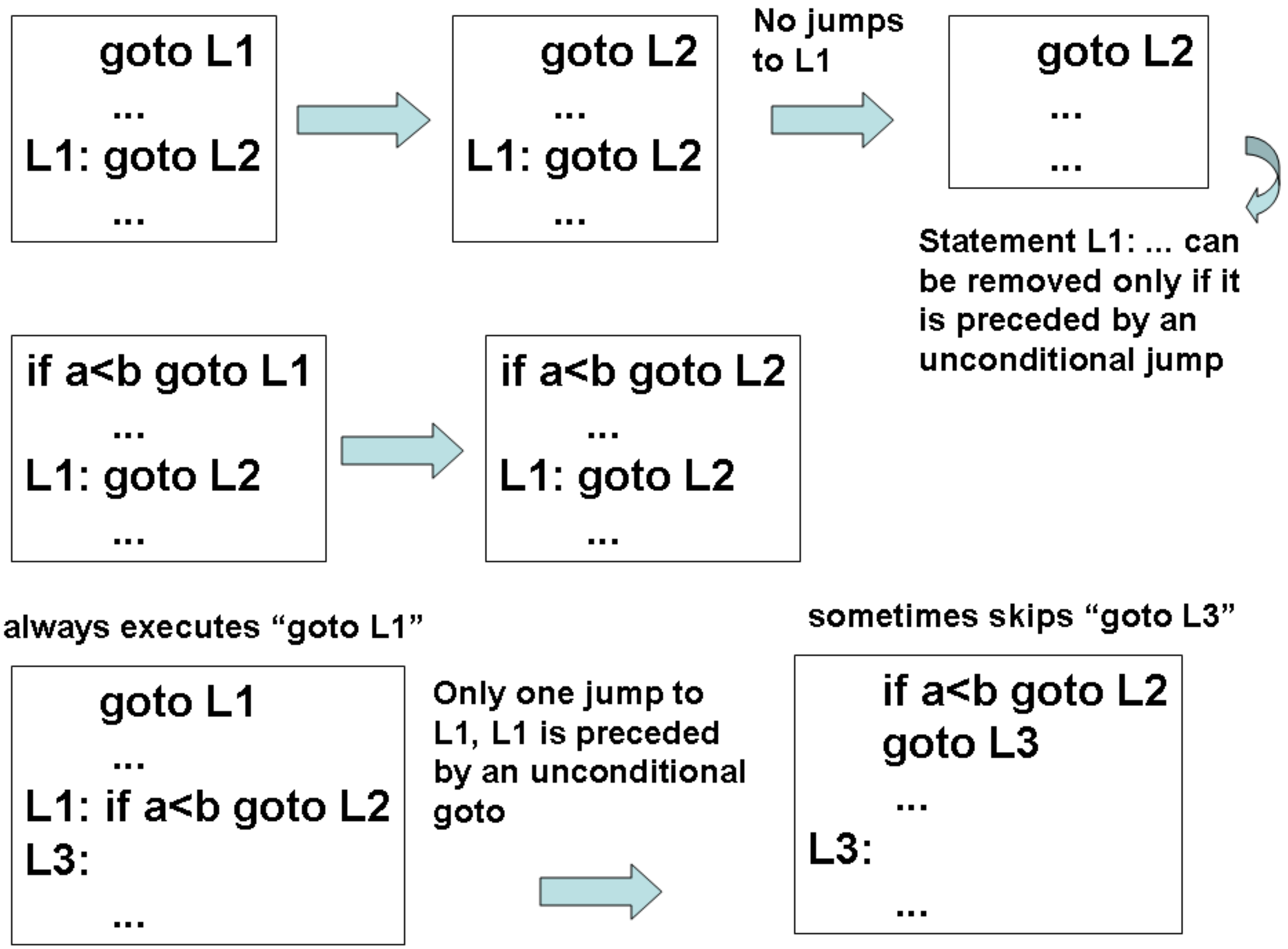
```
    if 0 != 1 goto L2
    print instructions
L2:
```

```
    goto L2
    print instructions
L2:
```

```
    goto L2
    ...
L2:
```

print instructions are now unreachable and hence can be eliminated

Flow-of-Control Optimizations



```
      goto L1
      ...
L1: goto L2
      ...
```

```
      goto L2
      ...
L1: goto L2
      ...
```

No jumps to L1

```
      goto L2
      ...
      ...
```

Statement L1: ... can be removed only if it is preceded by an unconditional jump

```
if a<b goto L1
      ...
L1: goto L2
      ...
```

```
if a<b goto L2
      ...
L1: goto L2
      ...
```

always executes “goto L1”

```
      goto L1
      ...
L1: if a<b goto L2
L3:
      ...
```

Only one jump to L1, L1 is preceded by an unconditional goto

sometimes skips “goto L3”

```
      if a<b goto L2
      goto L3
      ...
L3:
      ...
```

Reduction in Strength and Use of Machine Idioms

- x^2 is cheaper to implement as x*x, than as a call to an exponentiation routine
- For integers, $x*2^3$ is cheaper to implement as x << 3 (x left-shifted by 3 bits)
- For integers, $x/2^2$ is cheaper to implement as x >> 2 (x right-shifted by 2 bits)

Reduction in Strength and Use of Machine Idioms

- Floating point division by a constant can be approximated as multiplication by a constant
- Auto-increment and auto-decrement addressing modes can be used wherever possible
  - Subsume INCREMENT and DECREMENT operations (respectively)
- Multiply and add is a more complicated pattern to detect

Implementing Object-Oriented Languages

Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

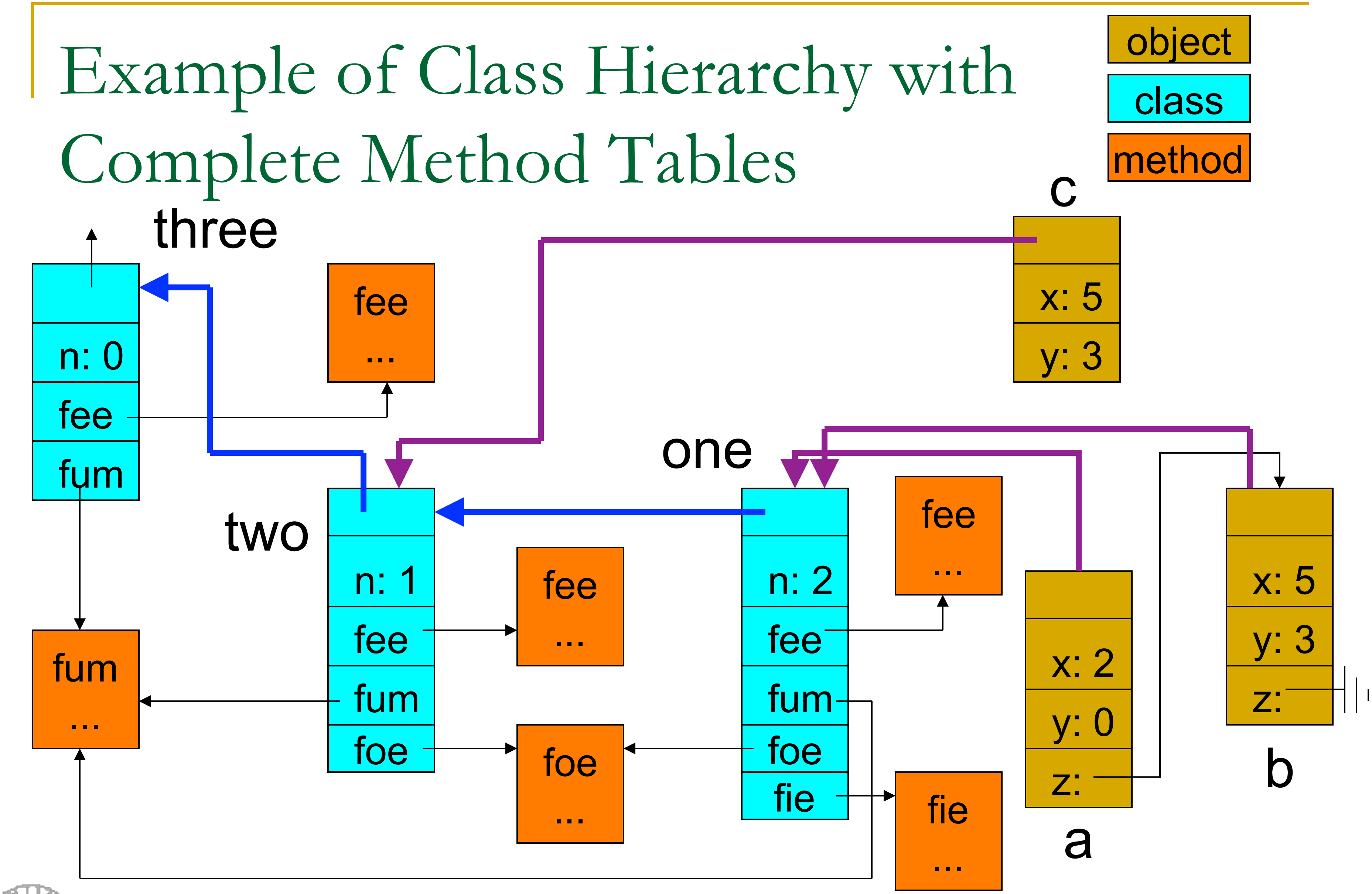
Outline of the Lecture

- Language requirements
- Mapping names to methods
- Variable name visibility
- Code generation for methods
- Simple optimizations
- **Parts of this lecture are based on the book, "Engineering a Compiler", by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.**

Language Requirements

- Objects and Classes
- Inheritance, subclasses and superclasses
- Inheritance requires that a subclass have all the instance variables specified by its superclass
  - Necessary for superclass methods to work with subclass objects
- If A is B's superclass, then some or all of A's methods/instance variables may be redefined in B

Example of Class Hierarchy with Complete Method Tables



Mapping Names to Methods

- Method invocations are not always static calls
- *a.fee*() invokes *one.fee*(), *a.foe*() invokes *two.foe*(), and *a.fum*() invokes *three.fum*()
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

Mapping Names to Methods

- If the class structure can be determined wholly at compile time, then the method tables can be statically built for each class
- If classes can be created at run-time or loaded dynamically (class definition can change too)
  - full lookup in the class hierarchy can be performed at run-time or
  - use complete method tables as before, and include a mechanism to update them when needed

Implementing Object-Oriented Languages - 2

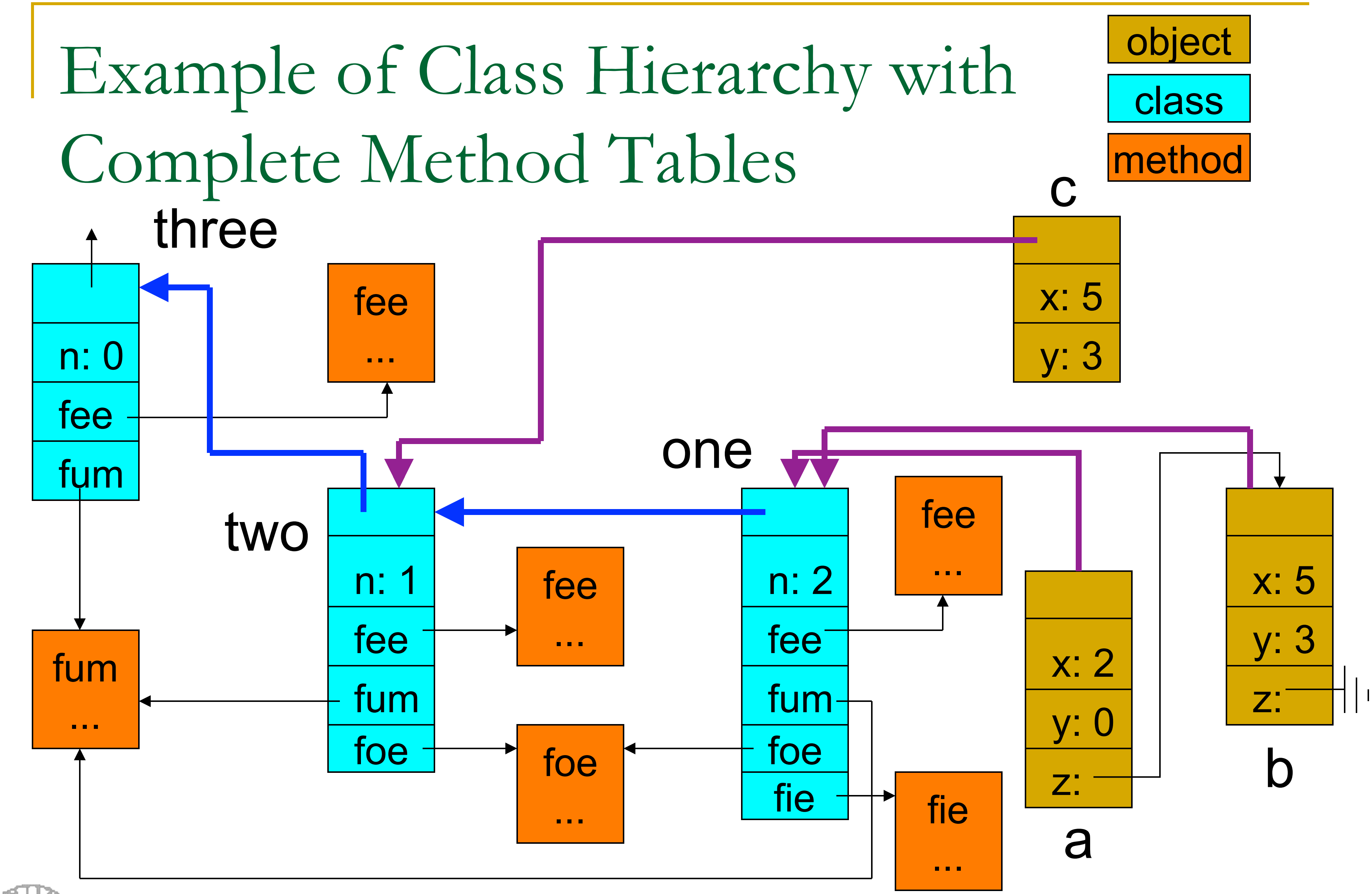
Y.N. Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Language requirements (in part 1)
- Mapping names to methods (in part 1)
- Variable name visibility
- Code generation for methods
- Simple optimizations
- **Parts of this lecture are based on the book, "Engineering a Compiler", by Keith Cooper and Linda Torczon, Morgan Kaufmann, 2004, sections 6.3.3 and 7.10.**

Example of Class Hierarchy with Complete Method Tables



Mapping Names to Methods

- Method invocations are not always static calls
- *a.fee*() invokes *one.fee*(), *a.foe*() invokes *two.foe*(), and *a.fum*() invokes *three.fum*()
- Conceptually, method lookup behaves as if it performs a search for each procedure call
  - These are called virtual calls
  - Search for the method in the receiver's class; if it fails, move up to the receiver's superclass, and further
  - To make this search efficient, an implementation places a complete method table in each class
  - Or, a pointer to the method table is included (virtual tbl ptr)

Rules for Variable Name Visibility

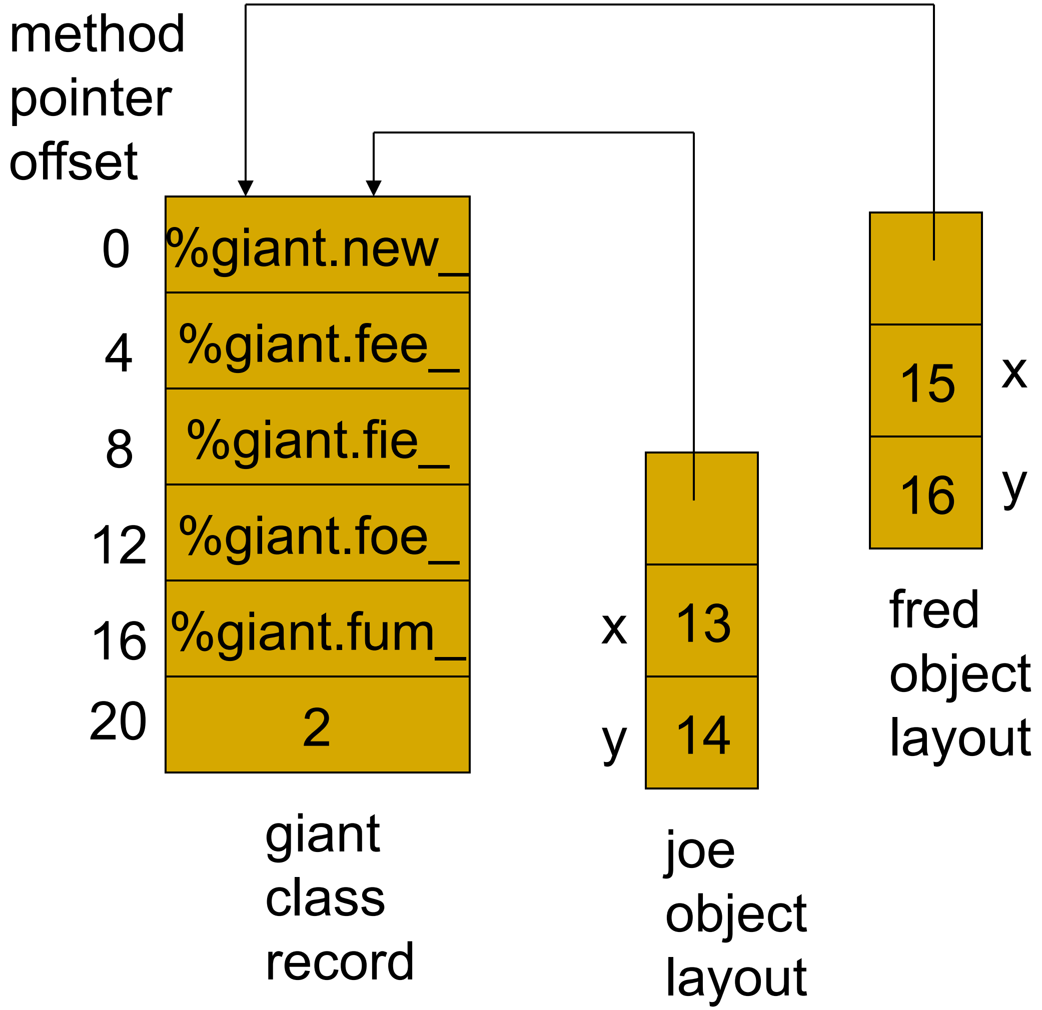
- Invoking b.fee() allows fee() to access all of b's instance variables (x,y,z), (since fee and b are both declared by class one), and also all class variables of classes one, two, and three
- However, invoking b.foe() allows foe() access only to instance variables x and y of b (not z), since foe() is declared by class two, and b by class one
  - foe() can also access class variables of classes two and three, but not class variables of class one

Code Generation for Methods

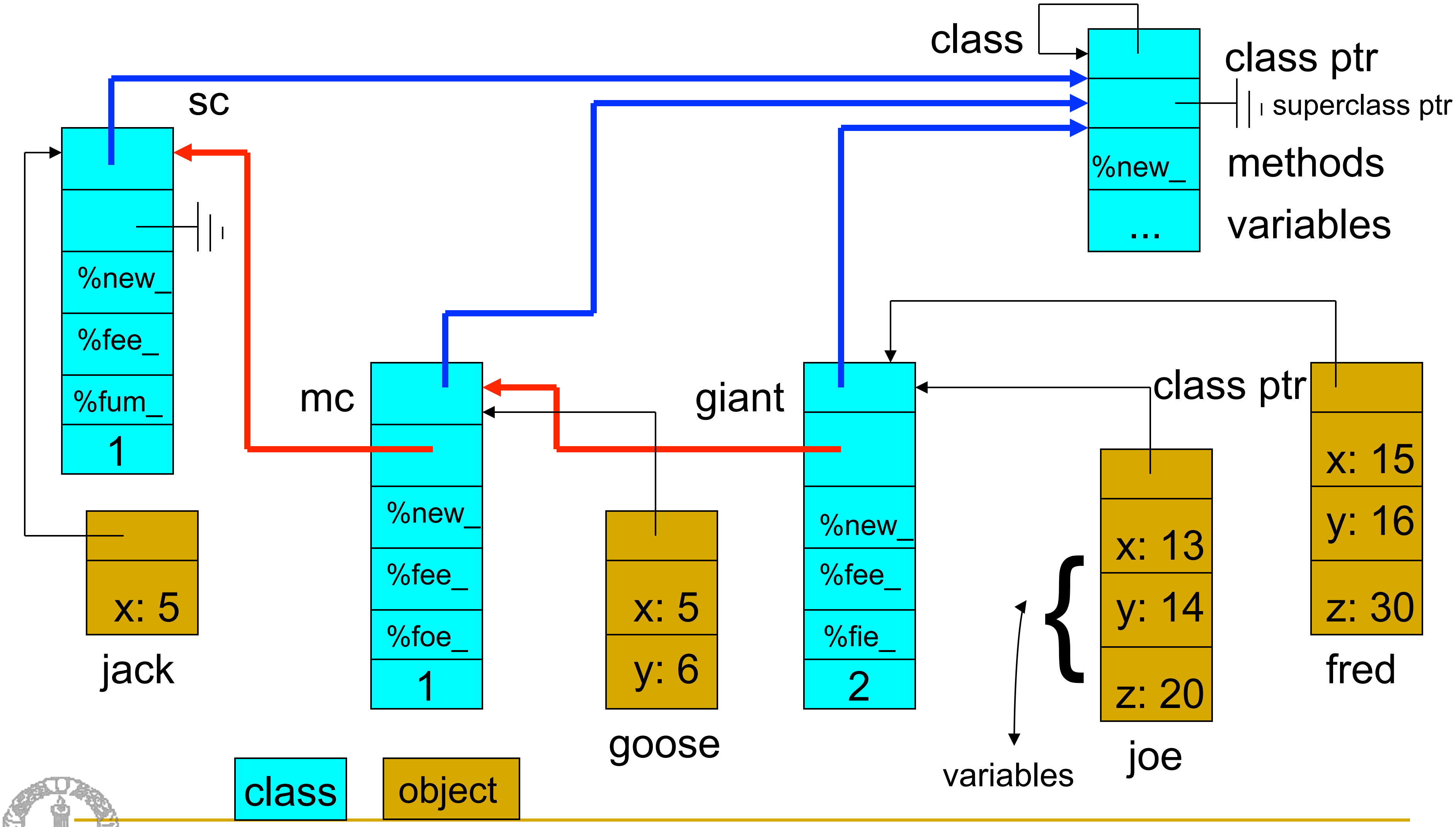
- Methods can access any data member of any object that becomes its receiver
  - receiver - every object that can find the method
  - subject to class hierarchy restrictions
- Compiler must establish an offset for each data member that applies uniformly to every receiver
- The compiler constructs these offsets as it processes the declarations for a class
  - Objects contain no code, only data

Single Class, No Inheritance

```
Example:
Class giant {
    int fee() {...}
    int fie() {...}
    int foe() {...}
    int fum() {...}
static n;
int x,y;
}
```



Implementing Single Inheritance



Single Inheritance Object Layout

class pointer	*sc* data members	*mc* data members	*giant* data members

- Now, an instance variable has the same offset in every class where it exists up in its superclass
- Method tables also follow a similar sequence as above
- When a class redefines a method defined in one of its superclasses
  - the method pointer for that method implementation must be stored at the same offset as the previous implementation of that method in the superclasses

Single Inheritance Object Layout (Complete Method Tables)

Object layout for joe/fred (giant)

class pointer	*sc* data members (x)	*mc* data members (y)	*giant* data members (z)

class record for class giant

class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fum_ pointer	%foe_ pointer	%fie_ pointer	2

Object layout for goose (mc)

class pointer	*sc* data members (x)	*mc* data members (y)

class record for class mc

class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fum_ pointer	%foe_ pointer	1

Object layout for jack (sc)

class pointer	*sc* data members (x)

class record for class sc

class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fum_ pointer	1

Single Inheritance Object Layout (including only changed and extra methods)

Object layout for joe/fred (giant)

class pointer	*sc* data members (x)	*mc* data members (y)	*giant* data members (z)

class record for class giant

class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fie_ pointer	2

Object layout for goose (mc)

class pointer	*sc* data members (x)	*mc* data members (y)

class record for class mc

class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%foe_ pointer	1

Object layout for jack (sc)

class pointer	*sc* data members (x)

class record for class sc

class pointer	superclass pointer	%new_ pointer	%fee_ pointer	%fum_ pointer	1

Fast Type Inclusion Tests – The need

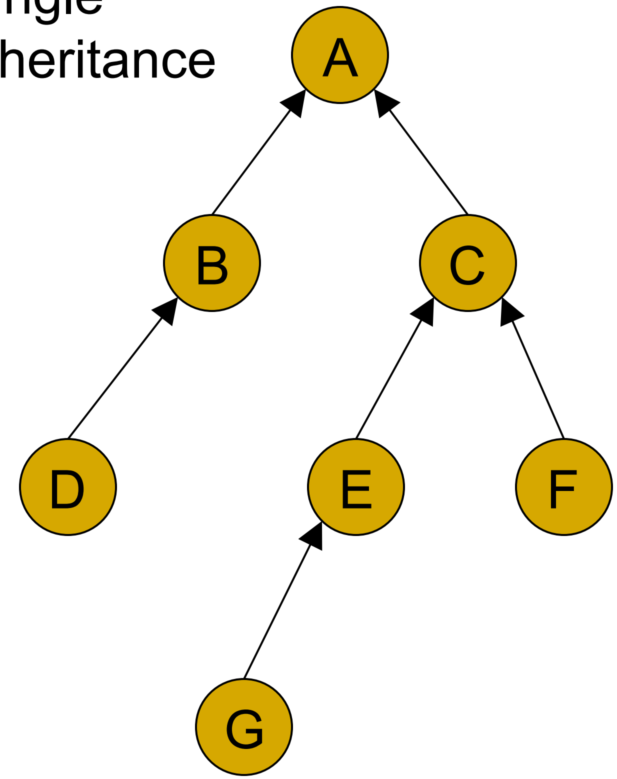
- If class Y is a subclass of class X
  - X a = new Y(); //a is of type base class of Y, okay
    // other code omitted
    Y b = a; // a holds a value of type Y
  - The above assignment is valid, but stmt 2 below is not
  - 1. X a = new X();
    // other code omitted
    2. Y b = a; // a holds a value of type X
- Runtime type checking to verify the above is needed
- Java has an explicit *instanceof* test that requires a runtime type checking

Fast Type Inclusion Tests – Searching the Class Hierarchy Graph

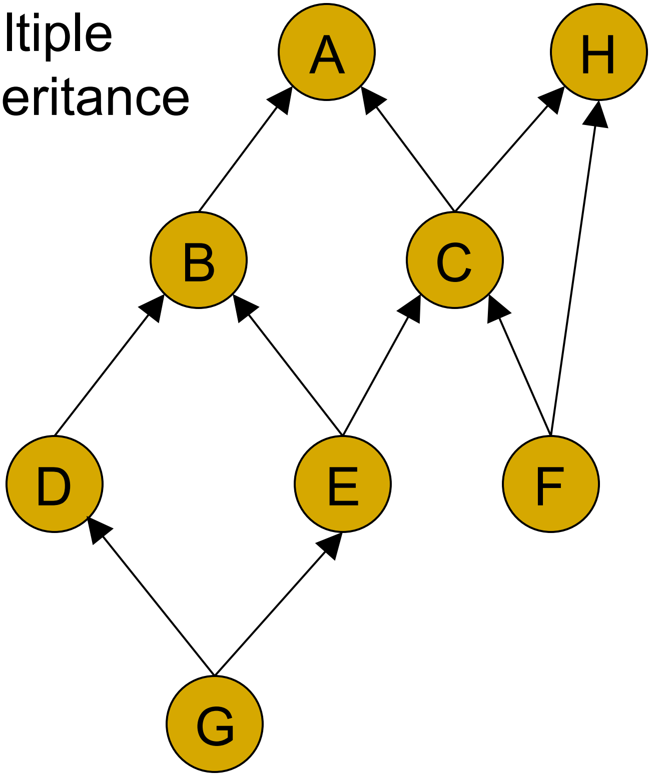
- Store the class hierarchy graph in memory
- Search and check if one node is an ancestor of another
- Traversal is straight forward to implement only for single inheritance
- Cumbersome and slow for multiple inheritance
- Execution time increases with depth of class hierarchy

Class Hierarchy Graph - Example

Single inheritance



Multiple inheritance



Fast Type Inclusion Tests – Binary Matrix

Class types

		C1	C2	C3	C4	C5
	C1	**0**	**1**	**0**	**0**	**1**
	C2	**0**	**0**	**1**	**0**	**1**
Class types	C3	**1**	**0**	**0**	**1**	**0**
	C4	**1**	**0**	**0**	**0**	**1**
	C5	**0**	**0**	**1**	**0**	**0**

Tests are efficient, but Matrix will be large in practice. The matrix can be compacted, but this increases access time. This can handle multiple inheritance also.

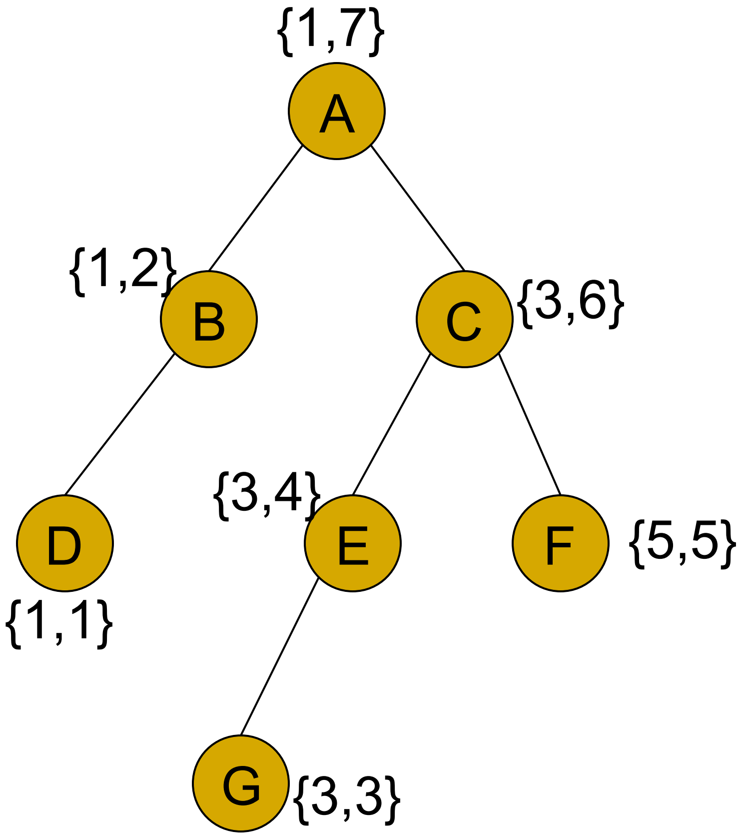
BM [C_i , C_j] = 1, iff C_i is a subclass of C_j

Relative (Schubert's) Numbering

{ l_a, r_a } for a node a :
r_a is the ordinal number of the node a in a postorder traversal of the tree. Let $\underline{\blacktriangleleft}$ denote "subtype of" relation. All descendants of a node are subtypes of that node. $\underline{\blacktriangleleft}$ is reflexive and transitive.
$l_a = \min \{ r_p \mid p \text{ is a descendant of } a \}$.

Now, $a \, \underline{\blacktriangleleft} \, b$, iff $l_b \leq r_a \leq r_b$.

This test is very fast and is O(1).
Works only for single inheritance.
Extensions to handle multiple inheritance are complex.

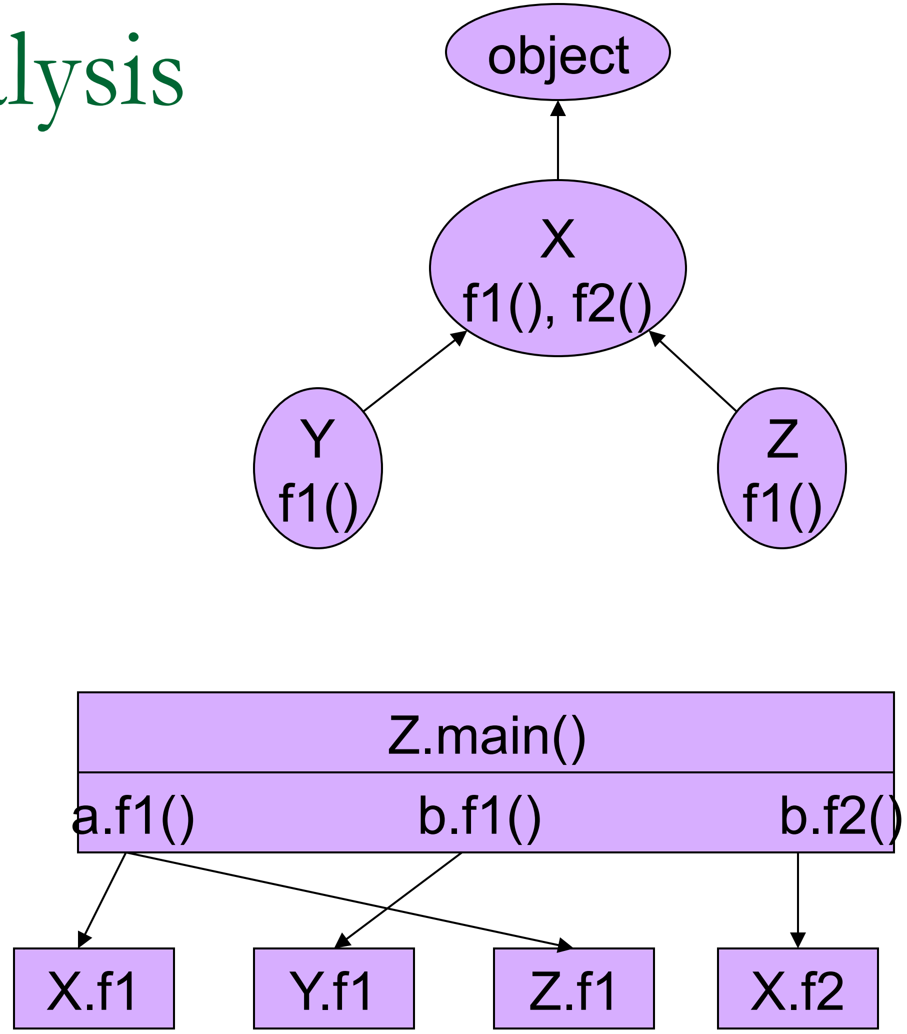


Devirtualization – Class Hierarchy Analysis

- Reduces the overhead of virtual method invocation
- Statically determines which virtual method calls resolve to a single method
- Such calls are either inlined or replaced by static calls
- Builds a class hierarchy and a call graph

Class Hierarchy Analysis

```
class X extends object {
        void f1() {. . .}
        void f2() {. . .}
}
class Y extends X {
        void f1() {. . .}
}
class Z extends X {
        void f1() {. . .}
        public static void main(...) {
                X a = new X(); Y b = new Y();
                Z c = new Z();
                 if (...) a = c;
                // other code
                a.f1();   b.f1();   b.f2();
        }
}
```



Global Register Allocation - 1

Y N Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Principles of Compiler Design

Outline

- Issues in Global Register Allocation
- The Problem
- Register Allocation based on Usage Counts
- Linear Scan Register allocation
- Chaitin's graph colouring based algorithm

Some Issues in Register Allocation

- Which values in a program reside in registers? (register allocation)
- In which register? (register assignment)
  - The two together are usually loosely referred to as register allocation
- What is the unit at the level of which register allocation is done?
  - Typical units are basic blocks, functions and regions.
  - RA within basic blocks is called local RA
  - The other two are known as global RA
  - Global RA requires much more time than local RA

Some Issues in Register Allocation

- Phase ordering between register allocation and instruction scheduling
  - Performing RA first restricts movement of code during scheduling – not recommended
  - Scheduling instructions first cannot handle spill code introduced during RA
    - Requires another pass of scheduling
- Tradeoff between speed and quality of allocation
  - In some cases, e.g., in Just-In-Time compilation, cannot afford to spend too much time in register allocation
  - Only local or both local and global allocation?

The Problem

- Global Register Allocation assumes that allocation is done beyond basic blocks and usually at function level
- Decision problem related to register allocation :
  - Given an intermediate language program represented as a control flow graph and a number ***k***, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads or stores are introduced, and at most ***k*** registers are used?
- This problem has been shown to be NP-hard (Sethi 1970).
- Graph colouring is the most popular heuristic used.
- However, there are simpler algorithms as well

Global Register Allocation - 2

Y N Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Principles of Compiler Design

Outline

- Issues in Global Register Allocation (in part 1)
- The Problem (in part 1)
- Register Allocation based in Usage Counts
- Linear Scan Register allocation
- Chaitin’s graph colouring based algorithm

The Problem

- Global Register Allocation assumes that allocation is done beyond basic blocks and usually at function level
- Decision problem related to register allocation :
  - Given an intermediate language program represented as a control flow graph and a number ***k***, is there an assignment of registers to program variables such that no conflicting variables are assigned the same register, no extra loads or stores are introduced, and at most ***k*** registers are used.
- This problem has been shown to be NP-hard (Sethi 1970).
- Graph colouring is the most popular heuristic used.
- However, there are simpler algorithms as well

Conflicting variables

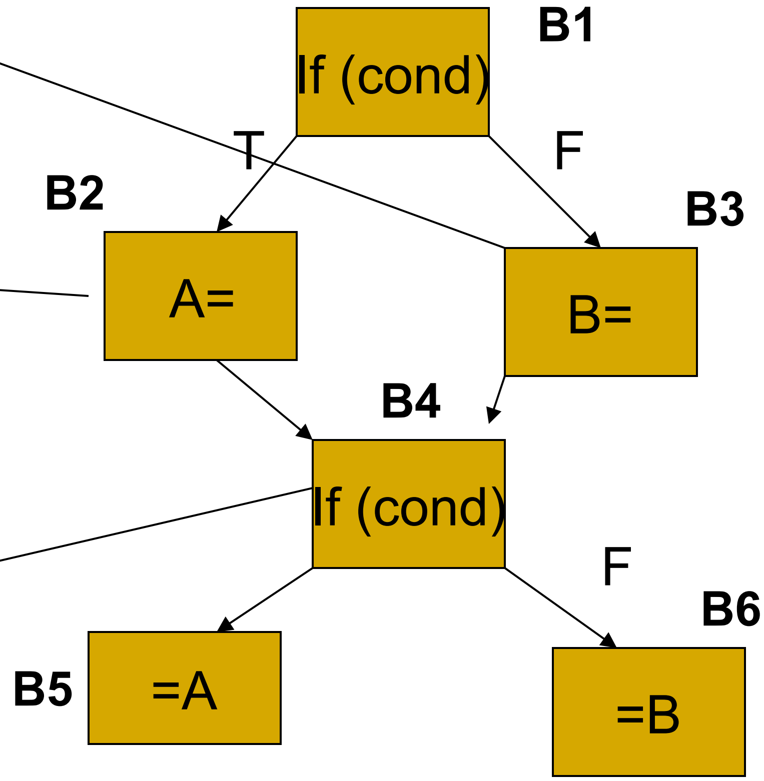
- Two variables interfere or conflict if their live ranges intersect
  - A variable is live at a point *p* in the flow graph, if there is a use of that variable in the path from *p* to the end of the flow graph
  - The live range of a variable is the smallest set of program points at which it is live.
  - Typically, instruction no. in the basic block along with the basic block no. is the representation for a point.

Example

Live range of A: B2, B4 B5
Live range of B: B3, B4, B6

```
If (cond)           A not live
    then A =
    else B =
X: if (cond)        B not live
     then = A
     else = B
```

A and B both live



Global Register Allocation via Usage Counts (for Single Loops)

- Allocate registers for variables used within loops
- Requires information about liveness of variables at the entry and exit of each basic block (BB) of a loop
- Once a variable is computed into a register, it stays in that register until the end of of the BB (subject to existence of next-uses)
- Load/Store instructions cost 2 units (because they occupy two words)

Global Register Allocation via Usage Counts (for Single Loops)

1. For every usage of a variable ***v*** in a BB, until it is first defined, do:
   - savings(v) = savings(v) + 1
   - after v is defined, it stays in the register any way, and all further references are to that register
2. For every variable **v** computed in a BB, if it is live on exit from the BB,
   - count a savings of 2, since it is not necessary to store it at the end of the BB

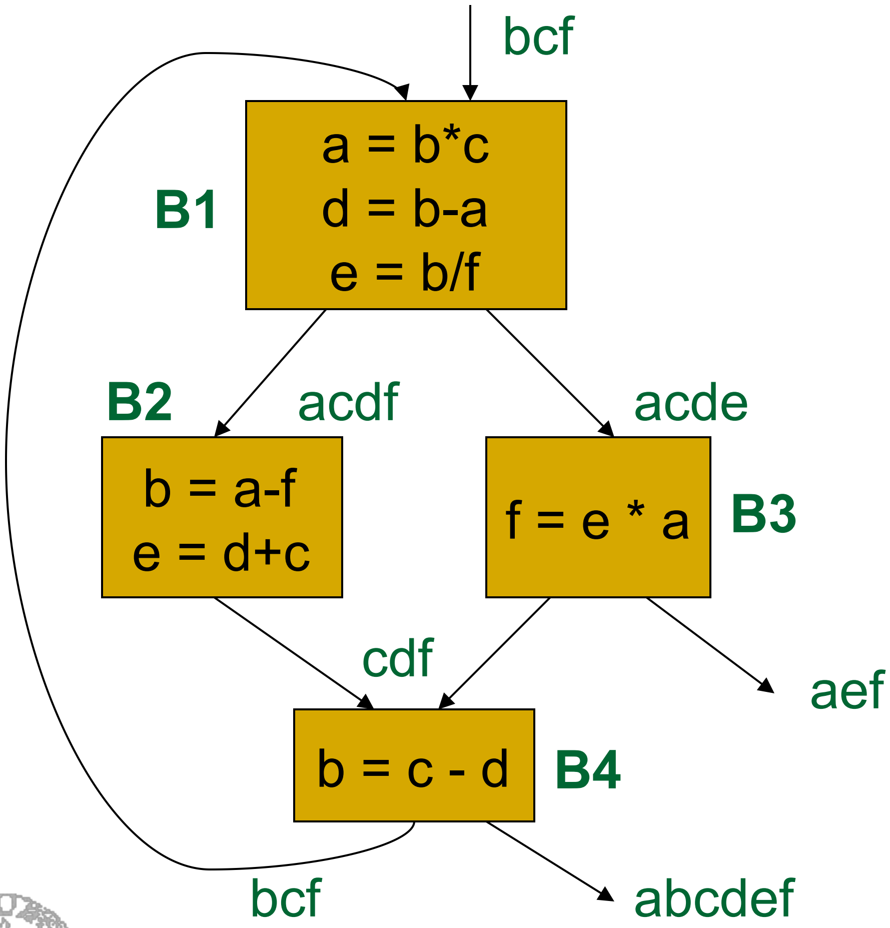
Global Register Allocation via Usage Counts (for Single Loops)

- Total savings per variable ***v*** are

$$\sum_{B \in Loop} (savings(v,B) + 2 * liveandcomputed(v,B))$$

  - *liveandcomputed(v,B)* in the second term is 1 or 0
- On entry to (exit from) the loop, we load (store) a variable live on entry (exit), and lose 2 units for each
  - But, these are “one time” costs and are neglected
- Variables, whose savings are the highest will reside in registers

Global Register Allocation via Usage Counts (for Single Loops)



Savings for the variables

B1 B2 B3 B4

a: (0+2)+(1+0)+(1+0)+(0+0) = 4

b: (3+0)+(0+0)+(0+0)+(0+2) = 5

c: (1+0)+(1+0)+(0+0)+(1+0) = 3

d: (0+2)+(1+0)+(0+0)+(1+0) = 4

e: (0+2)+(0+0)+(1+0)+(0+0) = 3

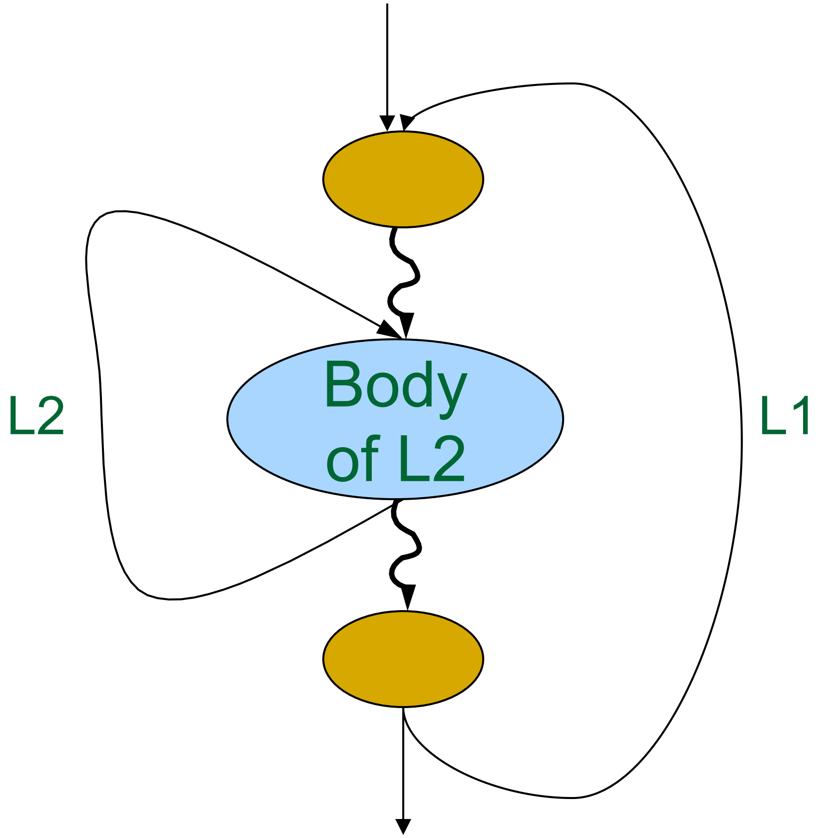
f: (1+0)+(1+0)+(0+2)+(0+0) = 4

If there are 3 registers, they will be allocated to the variables, a, b, and d

Global Register Allocation via Usage Counts (for Nested Loops)

- We first assign registers for inner loops and then consider outer loops. Let L1 nest L2
- For variables assigned registers in L2, but not in L1
    - load these variables on entry to L2 and store them on exit from L2
- For variables assigned registers in L1, but not in L2
    - store these variables on entry to L2 and load them on exit from L2
- All costs are calculated keeping the above rules

Global Register Allocation via Usage Counts (for Nested Loops)



- case 1: variables x,y,z assigned registers in L2, but not in L1
    - Load x,y,z on entry to L2
    - Store x,y,z on exit from L2
- case 2: variables a,b,c assigned registers in L1, but not in L2
    - Store a,b,c on entry to L2
    - Load a,b,c on exit from L2
- case 3: variables p,q assigned registers in both L1 and L2
    - No special action

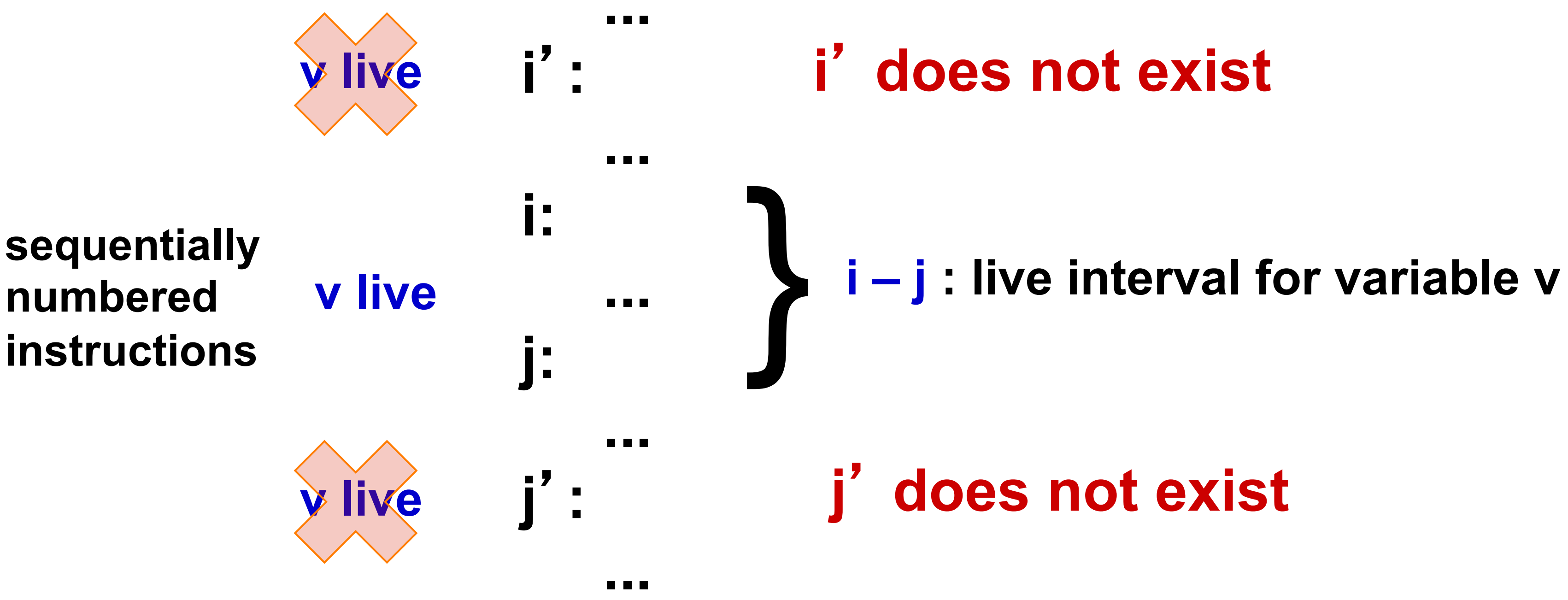
A Fast Register Allocation Scheme

- Linear scan register allocation(Poletto and Sarkar 1999) uses the notion of a live interval rather than a live range.
- Is relevant for applications where compile time is important, such as in dynamic compilation and in just-in-time compilers.
- Other register allocation schemes based on graph colouring are slow and are not suitable for JIT and dynamic compilers

Linear Scan Register Allocation

- Assume that there is some numbering of the instructions in the intermediate form
- An interval [i,j] is a ***live interval*** for variable v if there is no instruction with number j' > j such that v is live at j' and no instruction with number i' < i such that v is live at i
- This is a conservative approximation of live ranges: there may be subranges of [i,j] in which v is not live but these are ignored

Live Interval Example



…

~~v live~~ i’ : i’ does not exist

…

sequentially numbered instructions

i:

v live …

j:

i – j : live interval for variable v

…

~~v live~~ j’ : j’ does not exist

…

Example

If (cond)

then A=

else B=

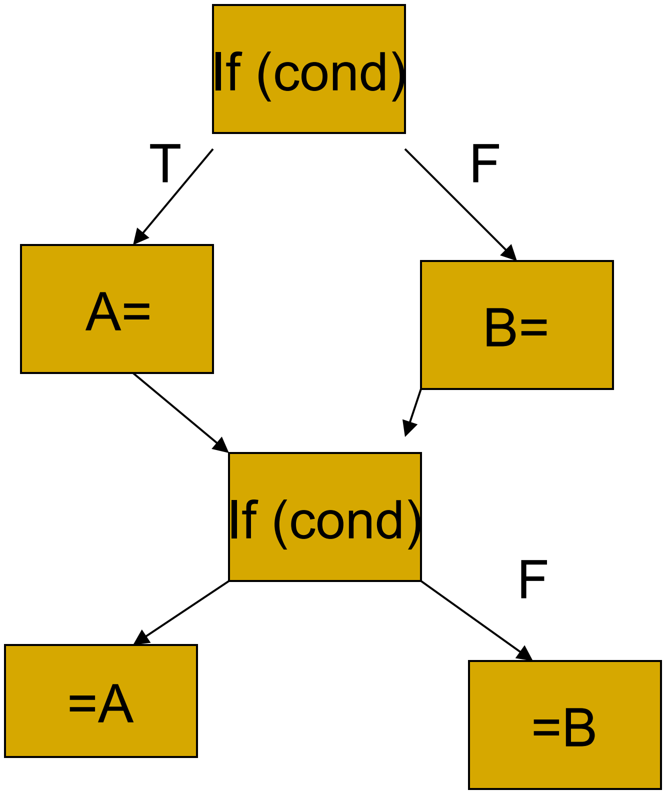
X: if (cond)

then =A

else = B

A NOT LIVE HERE

LIVE INTERVAL FOR A



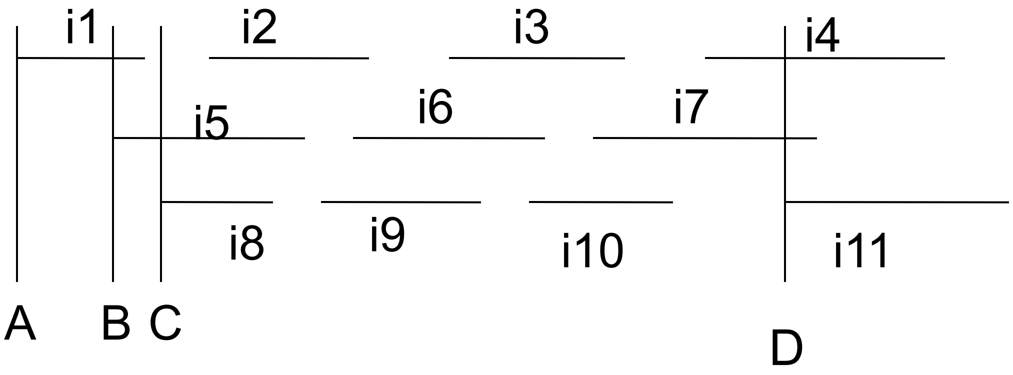
Live Intervals

- Given an order for pseudo-instructions and live variable information, live intervals can be computed easily with one pass through the intermediate representation.
- Interference among live intervals is assumed if they overlap.
- Number of overlapping intervals changes only at start and end points of an interval.

The Data Structures

- Live intervals are stored in the sorted order of increasing start point.
- At each point of the program, the algorithm maintains a list (active list) of live intervals that overlap the current point and that have been placed in registers.
- active list is kept in the sorted order of increasing end point.

Example



Active lists (in order of increasing end pt)

Active(A)= {i1}
Active(B)={i1,i5}
Active(C)={i8,i5}
Active(D)= {i7,i4,i11}

Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers are enough for computation without spills

The Algorithm (1)

```
{ active :=  [ ];
  for each live interval i, in order of increasing
        start point do
  { ExpireOldIntervals (i);
    if length(active) == R then SpillAtInterval(i);
    else { register[i] := a register removed from the
                                   pool of free registers;
              add i to active, sorted by increasing end point
           }
   }
}
```

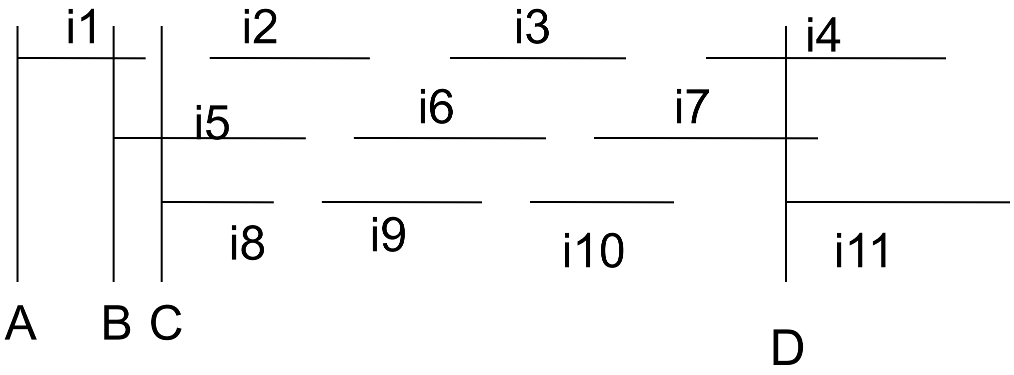
The Algorithm (2)

```
ExpireOldIntervals (i)
{ for each interval j in active, in order of
      increasing end point do
    { if endpoint[j] $\geq$ startpoint[i] then continue
      else { remove j from active;
                add register[j] to pool of free registers;
             }
   }
}
```

The Algorithm (3)

```
SpillAtInterval (i)
{ spill := last interval in active; /* last ending interval */
  if endpoint [spill] $\geq$ endpoint [i] then
    { register [i] := register [spill];
      location [spill] := new stack location;
      remove spill from active;
      add i to active, sorted by increasing end point;
    } else location [i] := new stack location;
}
```

Example 1



Active lists (in order of increasing end pt)

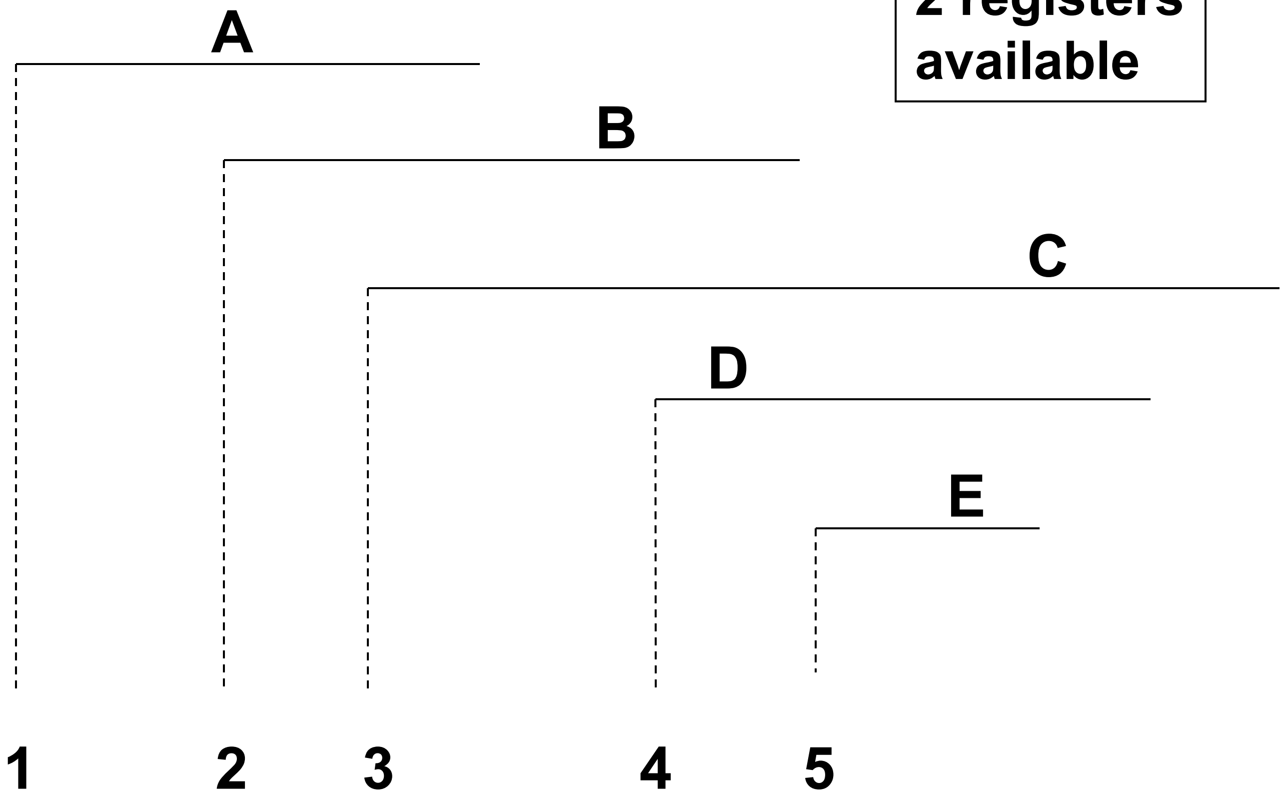
Active(A)= {i1}
Active(B)={i1,i5}
Active(C)={i8,i5}
Active(D)= {i7,i4,i11}

Sorted order of intervals (according to start point):
i1, i5, i8, i2, i9, i6, i3, i10, i7, i4, i11

Three registers are enough for computation without spills

Example 2

2 registers available

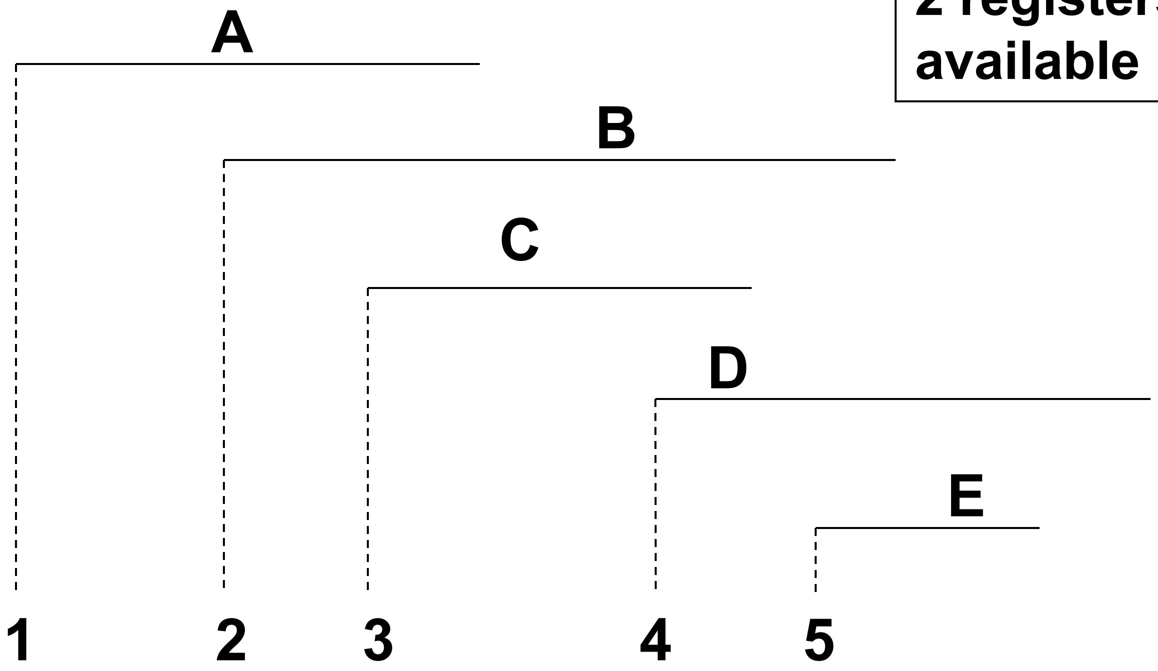


1,2 : give A,B register
3: Spill C since endpoint[C] > endpoint [B]

4: A expires, give D register
5: B expires, E gets register

Example 3

2 registers available



1,2 : give A,B register
3: Spill B since endpoint[B] > endpoint [C]
give register to C

4: A expires, give D register
5: C expires, E gets register

Complexity of the Linear Scan Algorithm

- If V is the number of live intervals and R the number of available physical registers, then if a balanced binary tree is used for storing the active intervals, complexity is O(V log R).
  - Active list can be at most ‘R’ long
  - Insertion and deletion are the important operations
- Empirical results reported in literature indicate that linear scan is significantly faster than graph colouring algorithms and code emitted is at most 10% slower than that generated by an aggressive graph colouring algorithm.

Chaitin's Formulation of the Register Allocation Problem

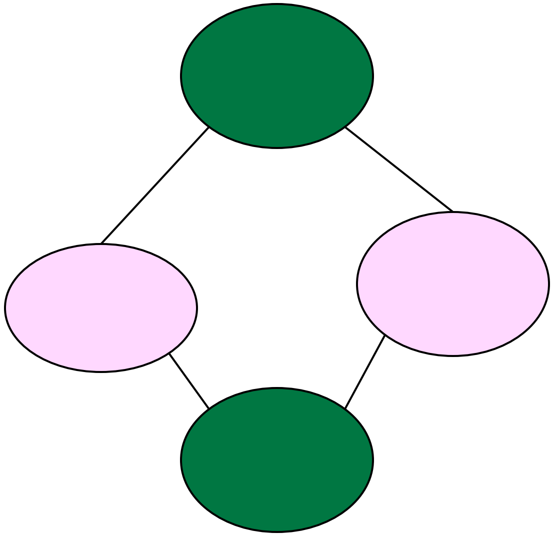
- A graph colouring formulation on the interference graph
- Nodes in the graph represent either live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists are used to represent the graph.

Chaitin's Formulation of the Register Allocation Problem

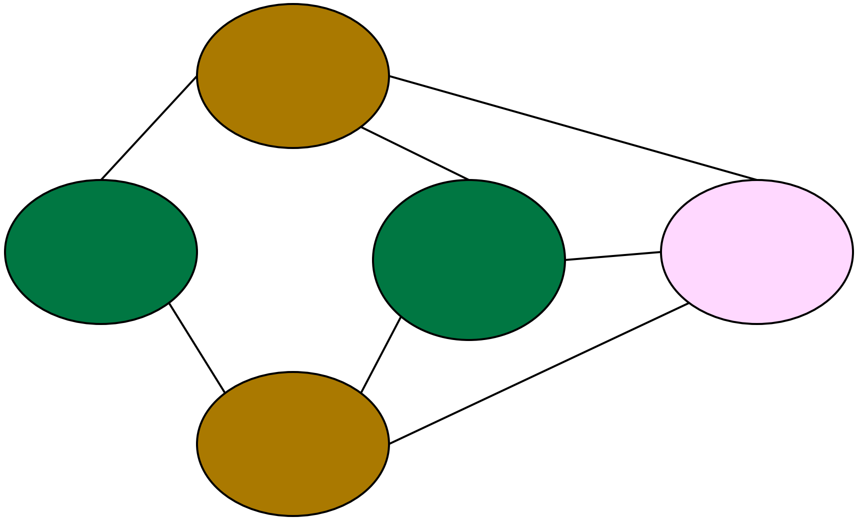
- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour
  - The number of colours available is the number of registers available on the machine
  - A k-colouring of the interference graph is mapped onto an allocation with k registers

Example

- Two colourable



Three colourable



Idea behind Chaitin's Algorithm

- Choose an arbitrary node of degree less than k and put it on the stack
- Remove that vertex and all its edges from the graph
  - This may decrease the degree of some other nodes and cause some more nodes to have degree less than k
- At some point, if all vertices have degree greater than or equal to k, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in a colour not used by neighbours (reuse colours as far as possible)

Global Register Allocation - 3

Y N Srikant
Computer Science and Automation
Indian Institute of Science
Bangalore 560012

NPTEL Course on Principles of Compiler Design

Outline

- Issues in Global Register Allocation (in part 1)
- The Problem (in part 1)
- Register Allocation based in Usage Counts (in part 2)
- Linear Scan Register allocation (in part 2)
- Chaitin's graph colouring based algorithm

Chaitin's Formulation of the Register Allocation Problem

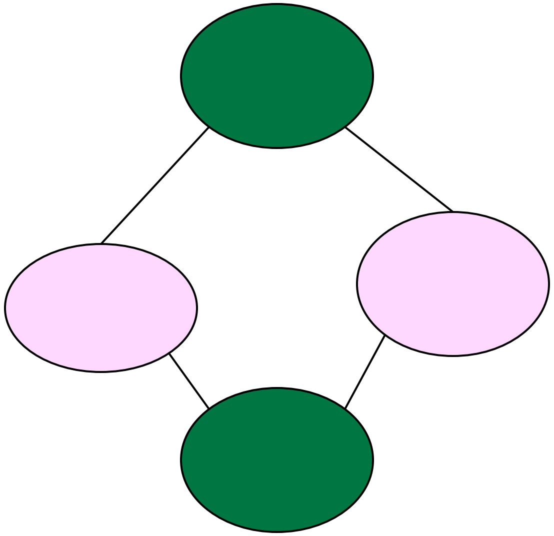
- A graph colouring formulation on the interference graph
- Nodes in the graph represent either live ranges of variables or entities called webs
- An edge connects two live ranges that interfere or conflict with one another
- Usually both adjacency matrix and adjacency lists are used to represent the graph.

Chaitin's Formulation of the Register Allocation Problem

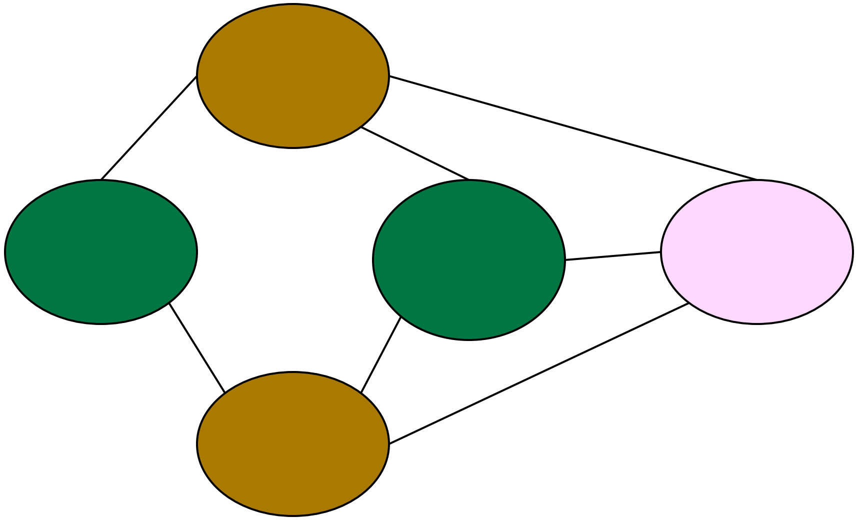
- Assign colours to the nodes such that two nodes connected by an edge are not assigned the same colour
  - The number of colours available is the number of registers available on the machine
  - A k-colouring of the interference graph is mapped onto an allocation with k registers

Example

- Two colourable



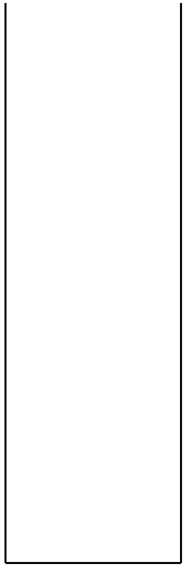
Three colourable



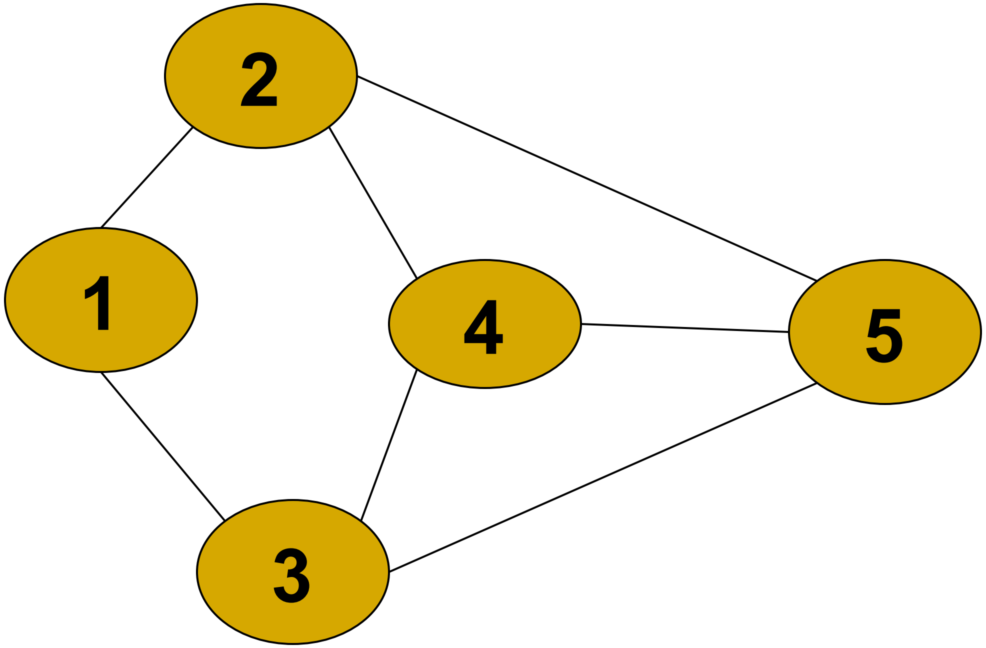
Idea behind Chaitin's Algorithm

- Choose an arbitrary node of degree less than k and put it on the stack
- Remove that vertex and all its edges from the graph
    - This may decrease the degree of some other nodes and cause some more nodes to have degree less than k
- At some point, if all vertices have degree greater than or equal to k, some node has to be spilled
- If no vertex needs to be spilled, successively pop vertices off stack and colour them in a colour not used by neighbours (reuse colours as far as possible)

Simple example – Given Graph

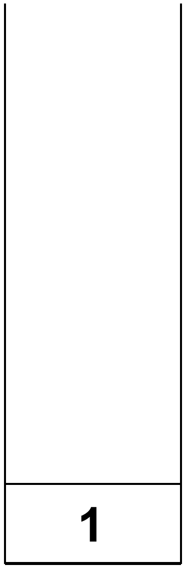


STACK

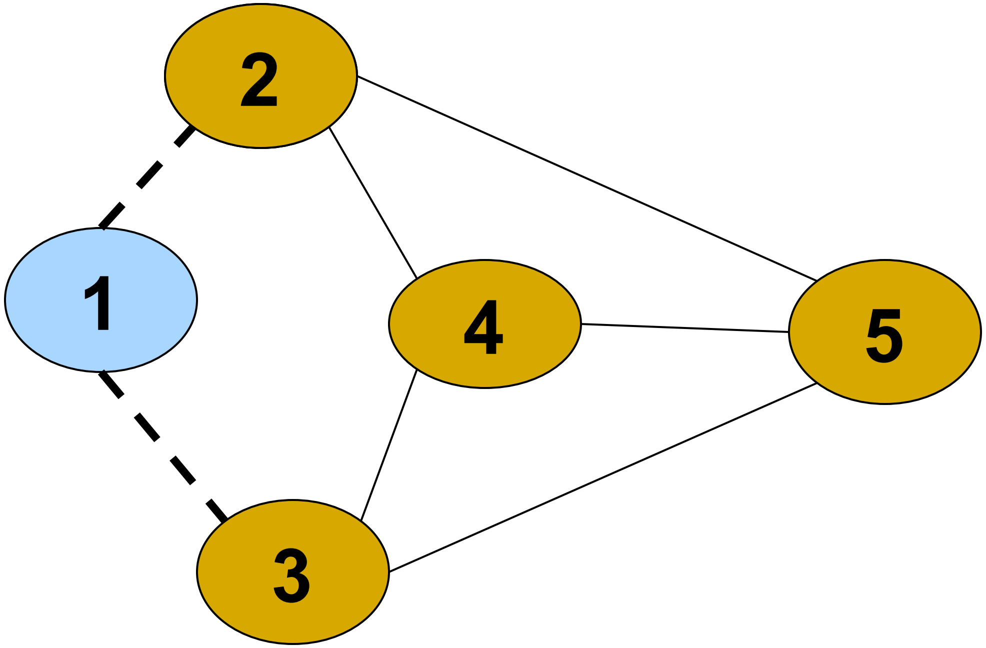


3 REGISTERS

Simple Example – Delete Node 1

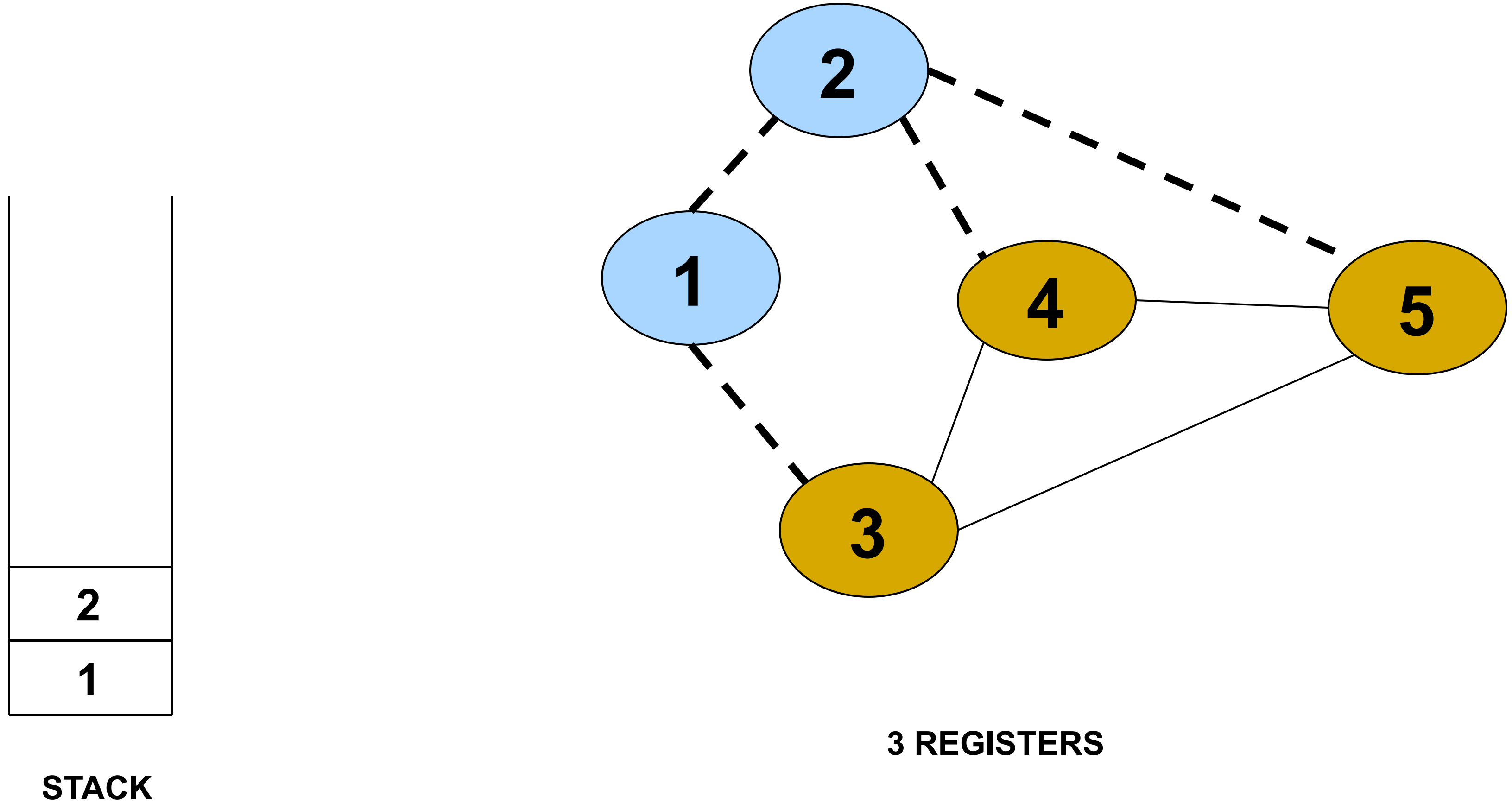


STACK

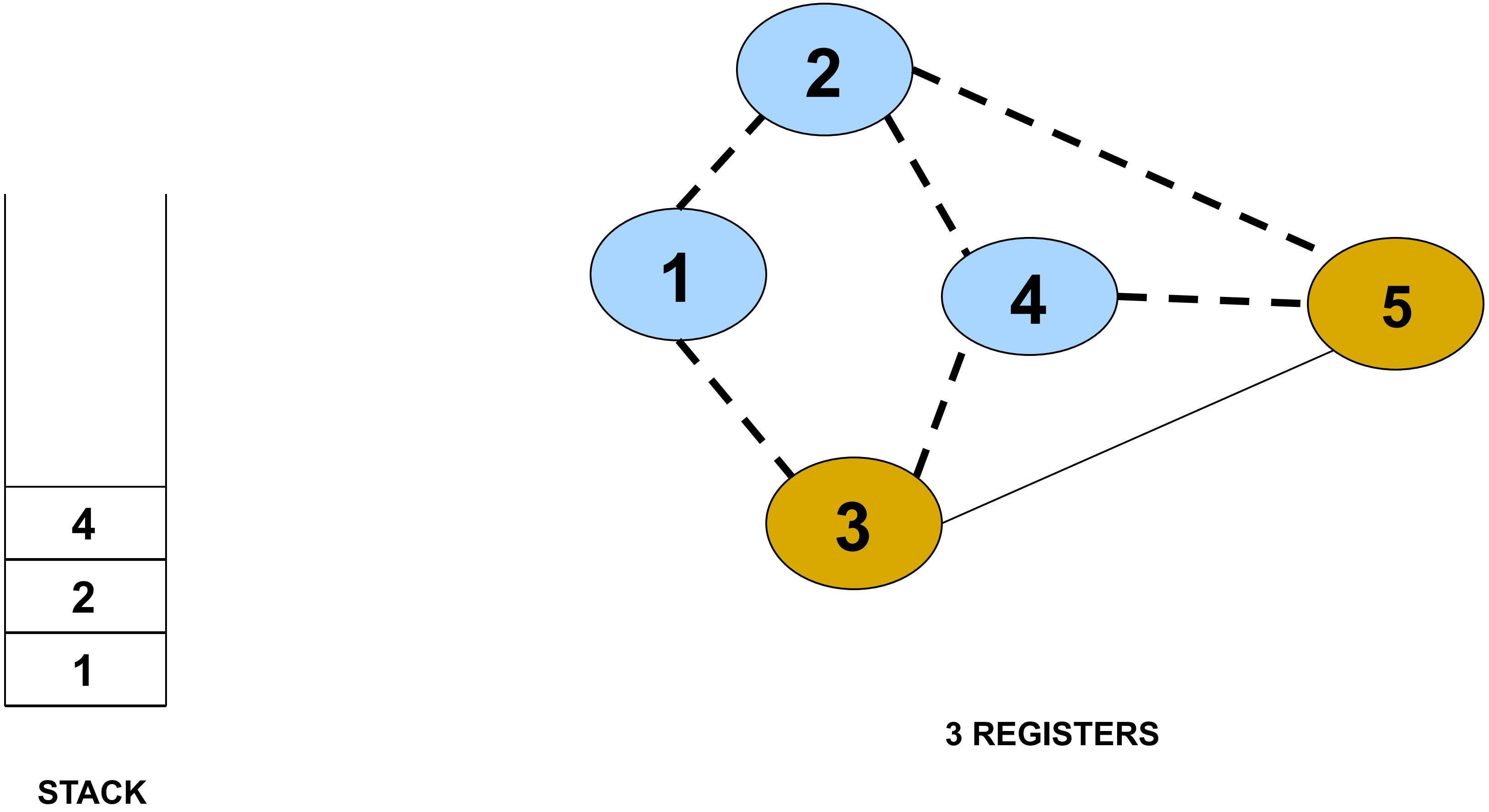


3 REGISTERS

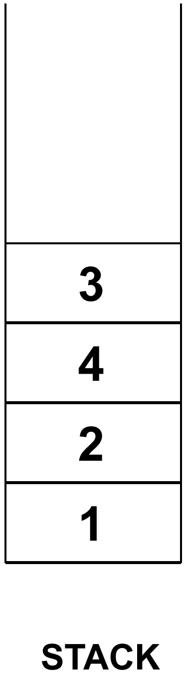
Simple Example – Delete Node 2

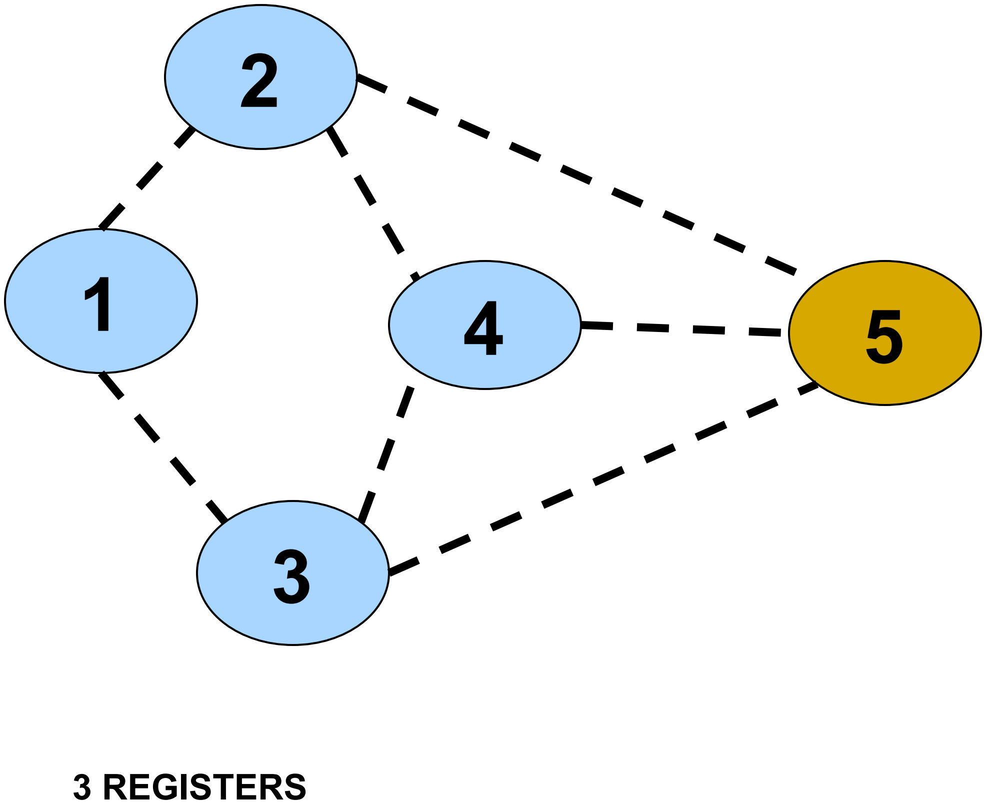


Simple Example – Delete Node 4

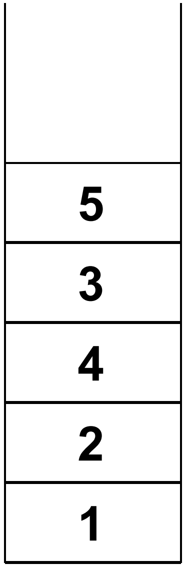


Simple Example – Delete Nodes 3

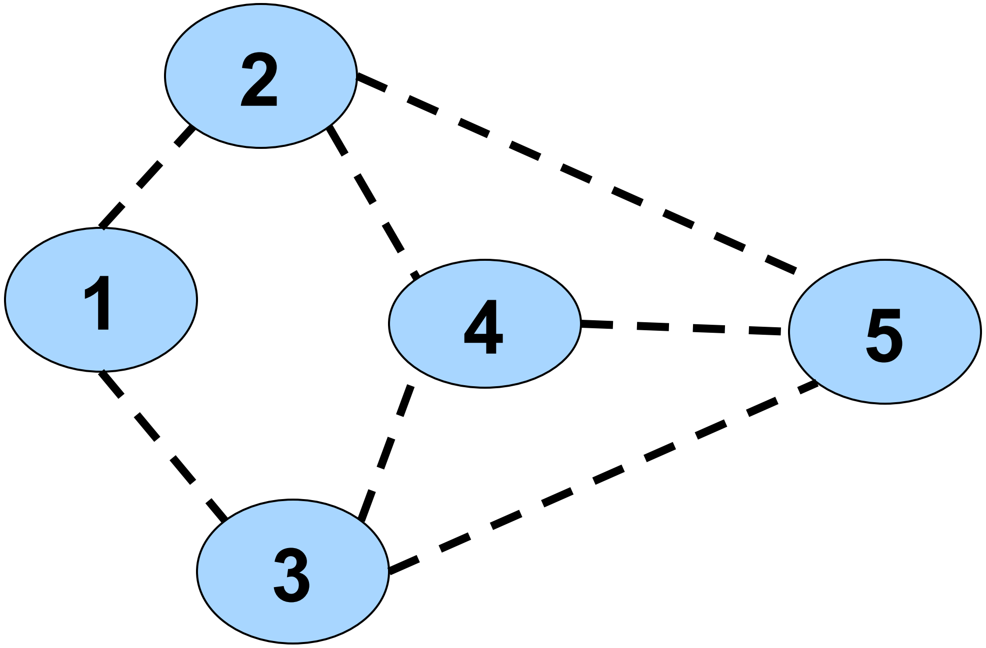




Simple Example – Delete Nodes 5

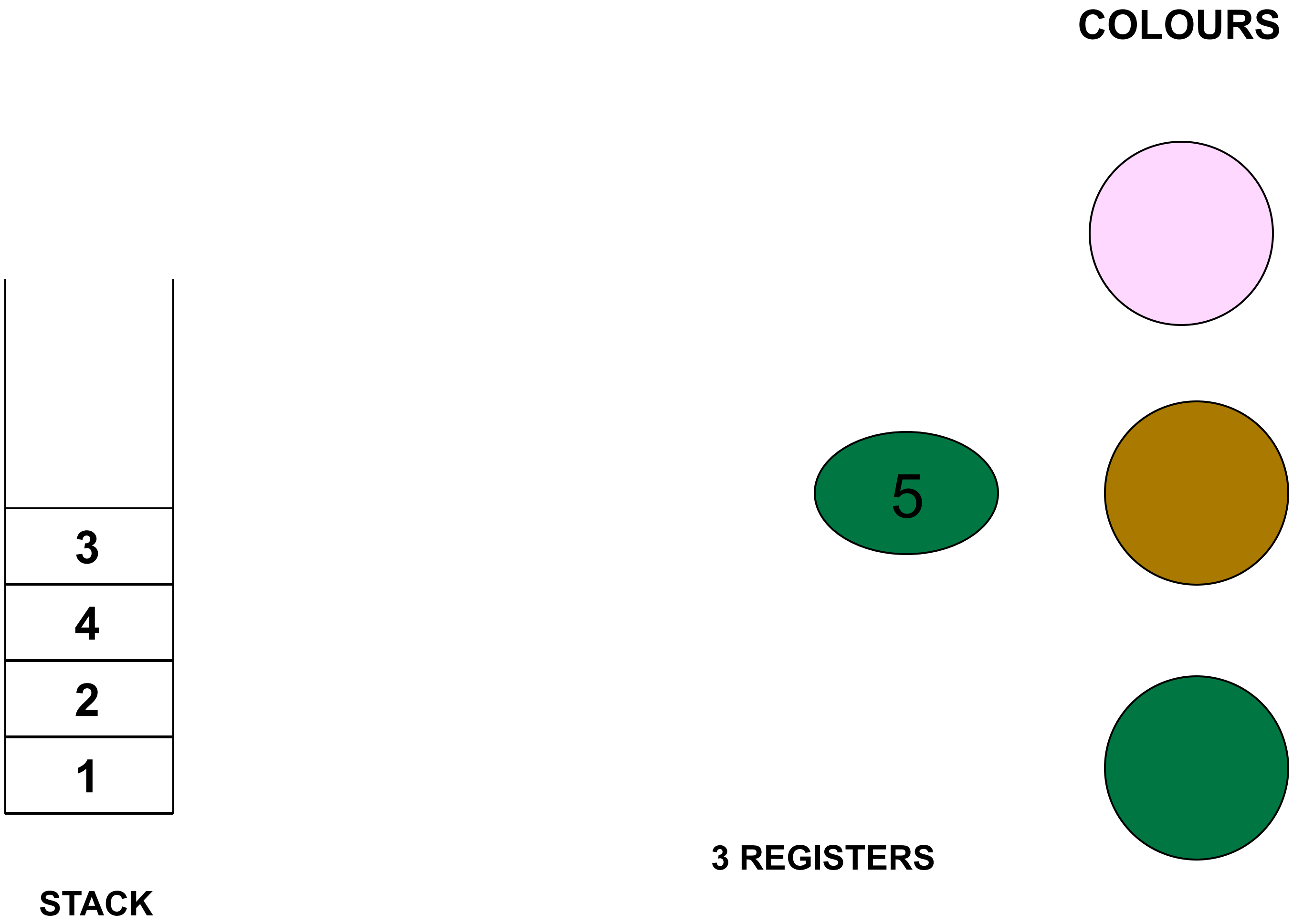


STACK

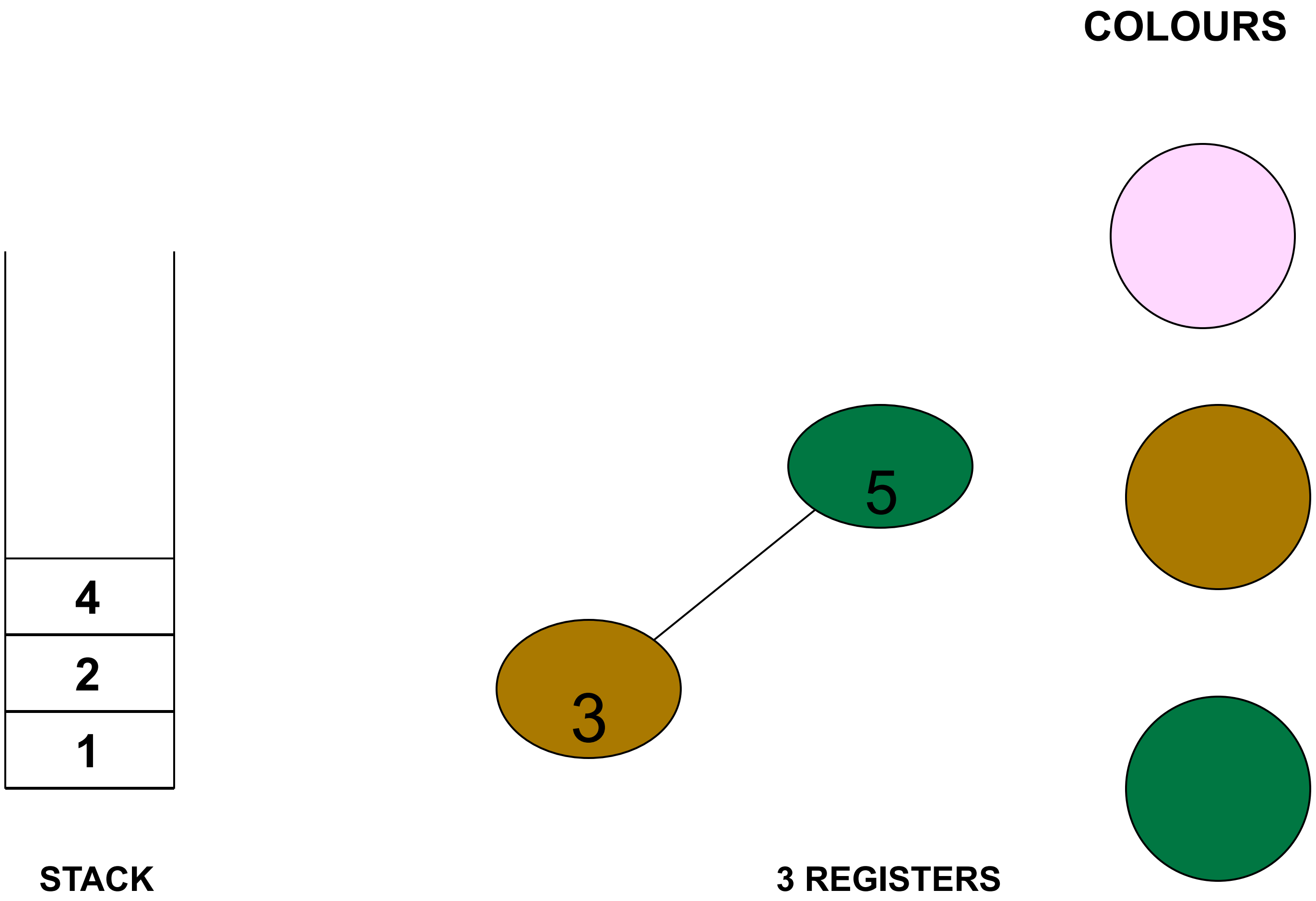


3 REGISTERS

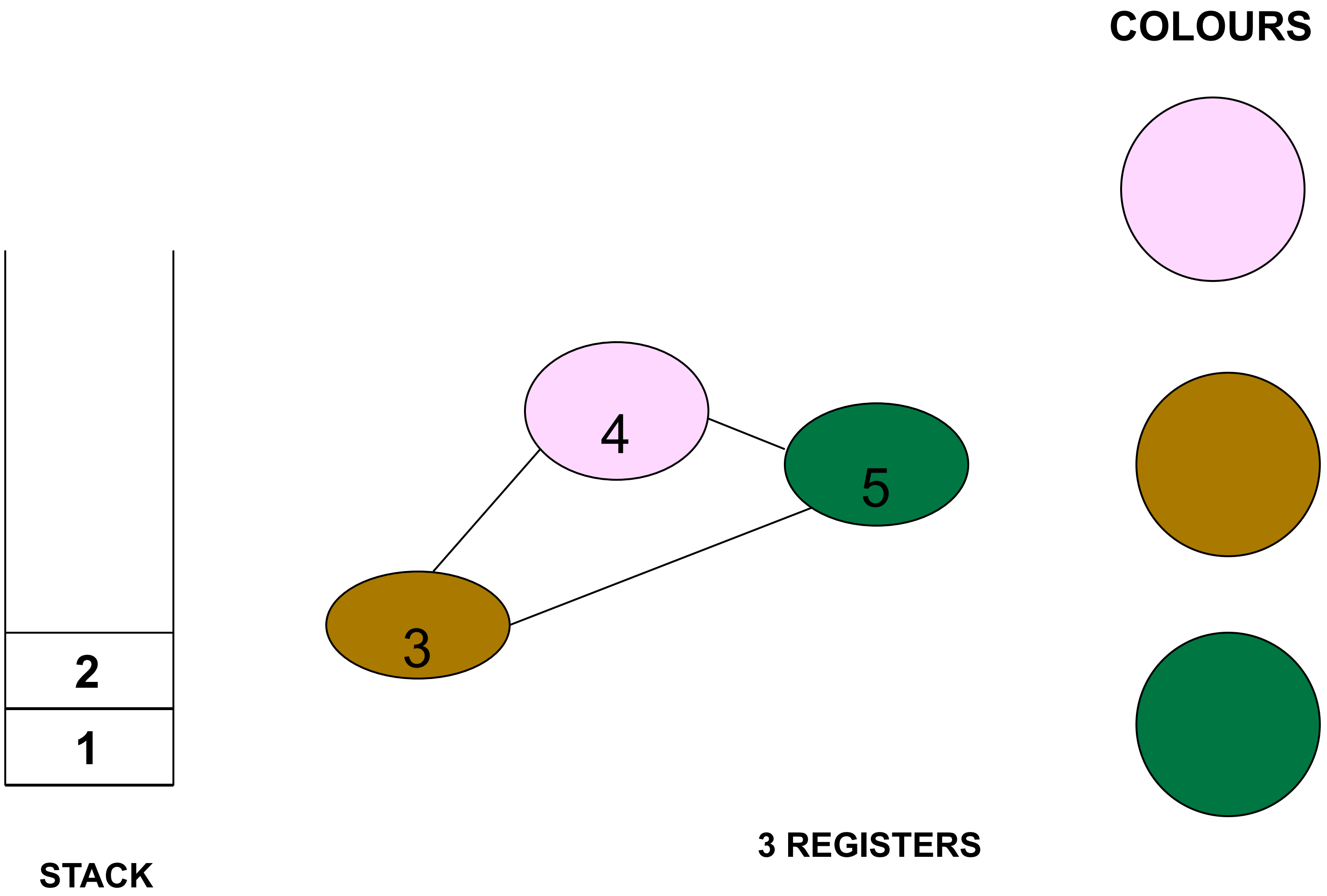
Simple Example – Colour Node 5



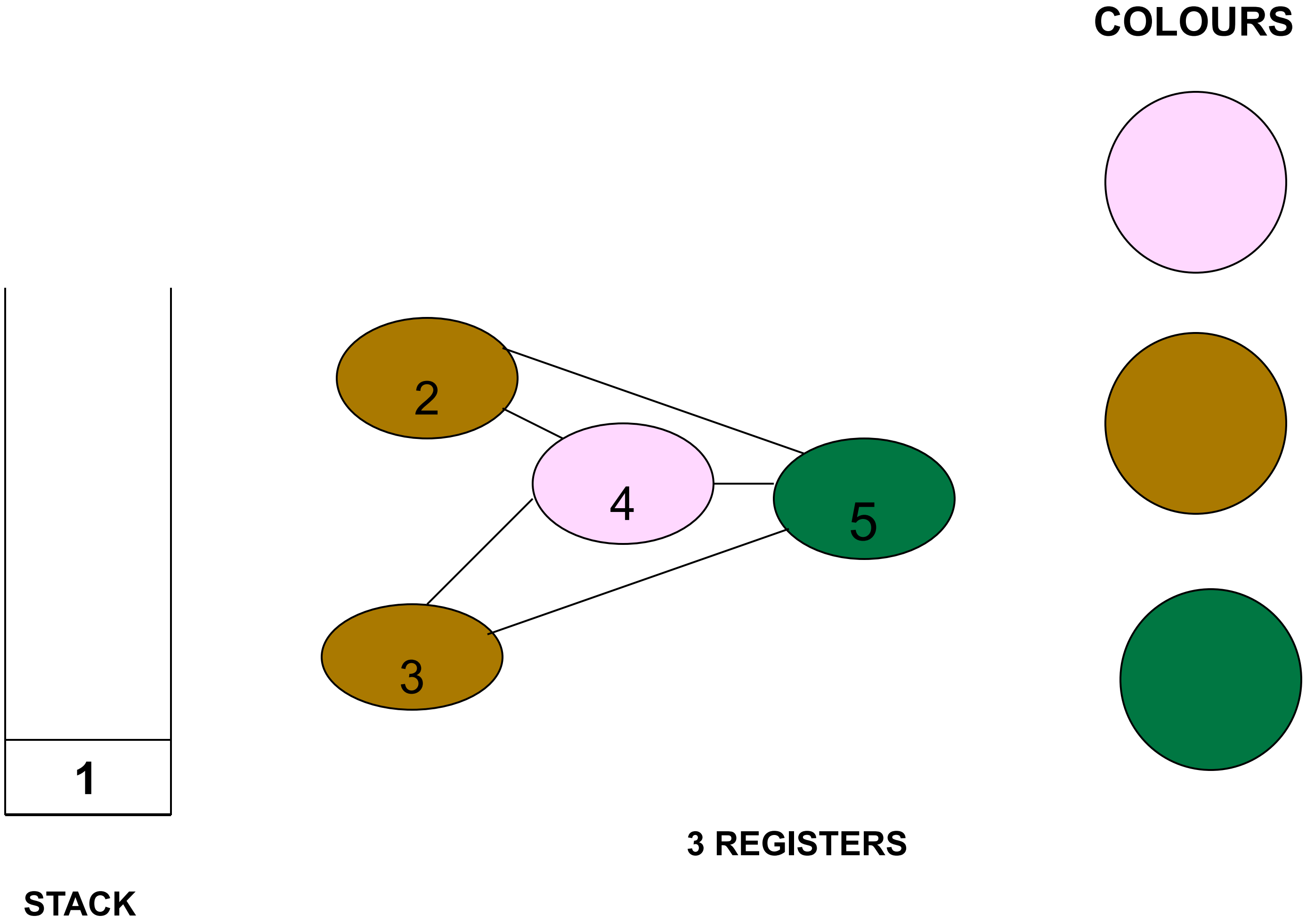
Simple Example – Colour Node 3



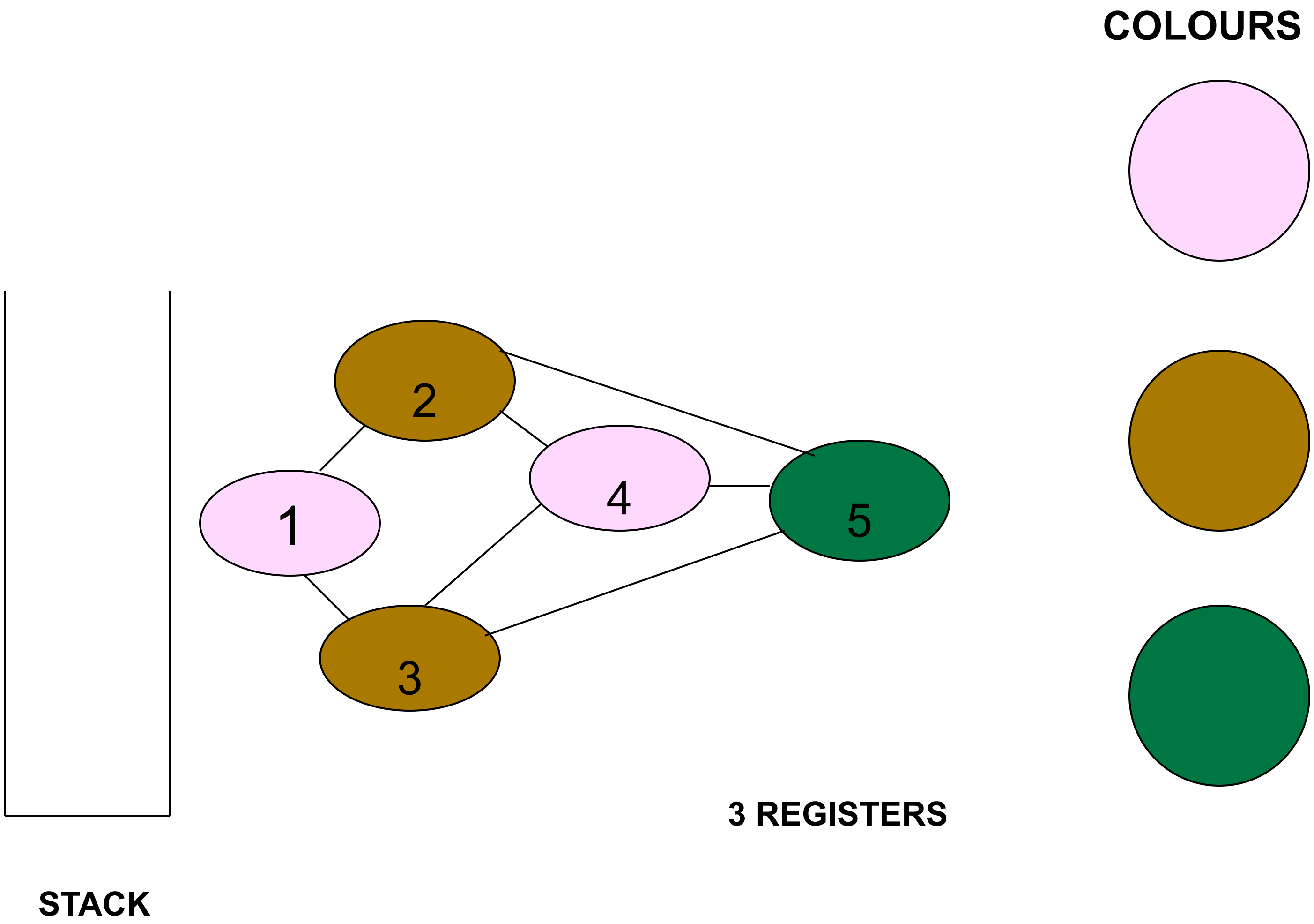
Simple Example – Colour Node 4



Simple Example – Colour Node 2



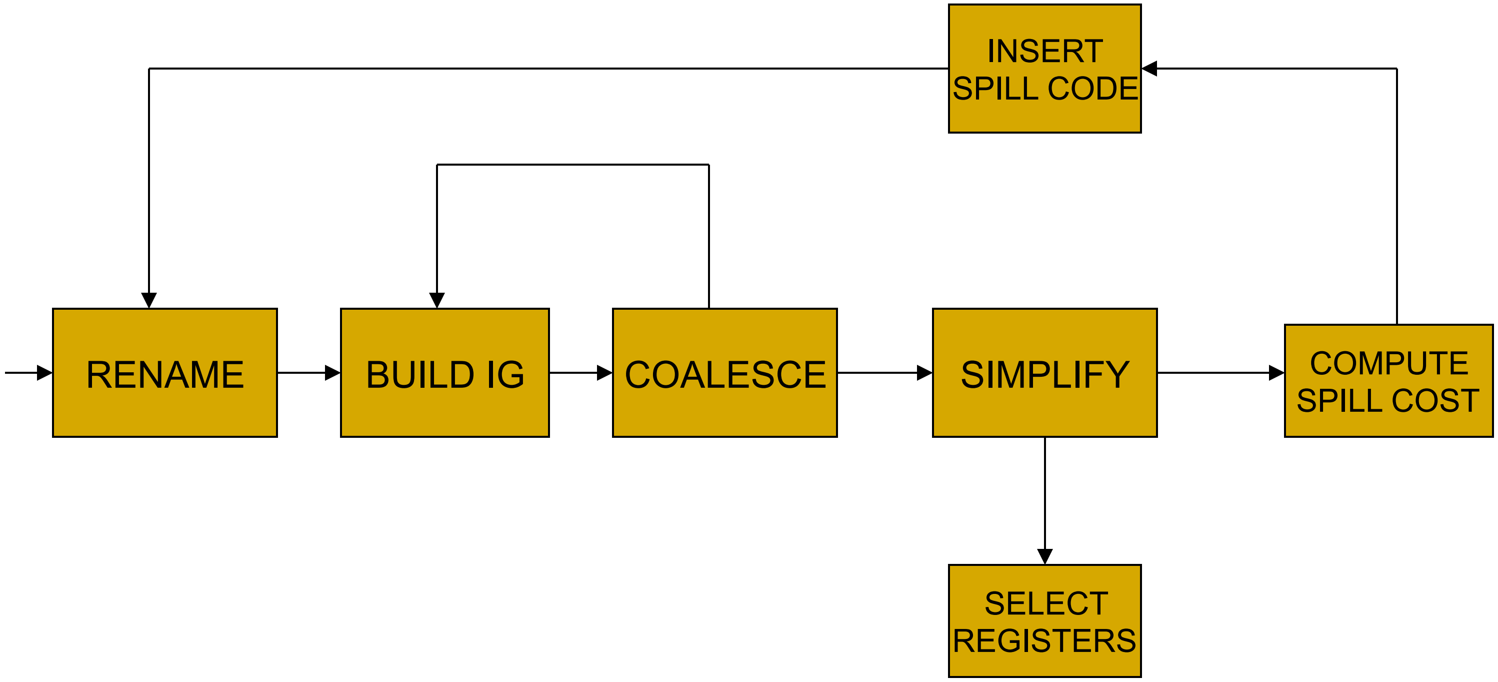
Simple Example – Colour Node 1



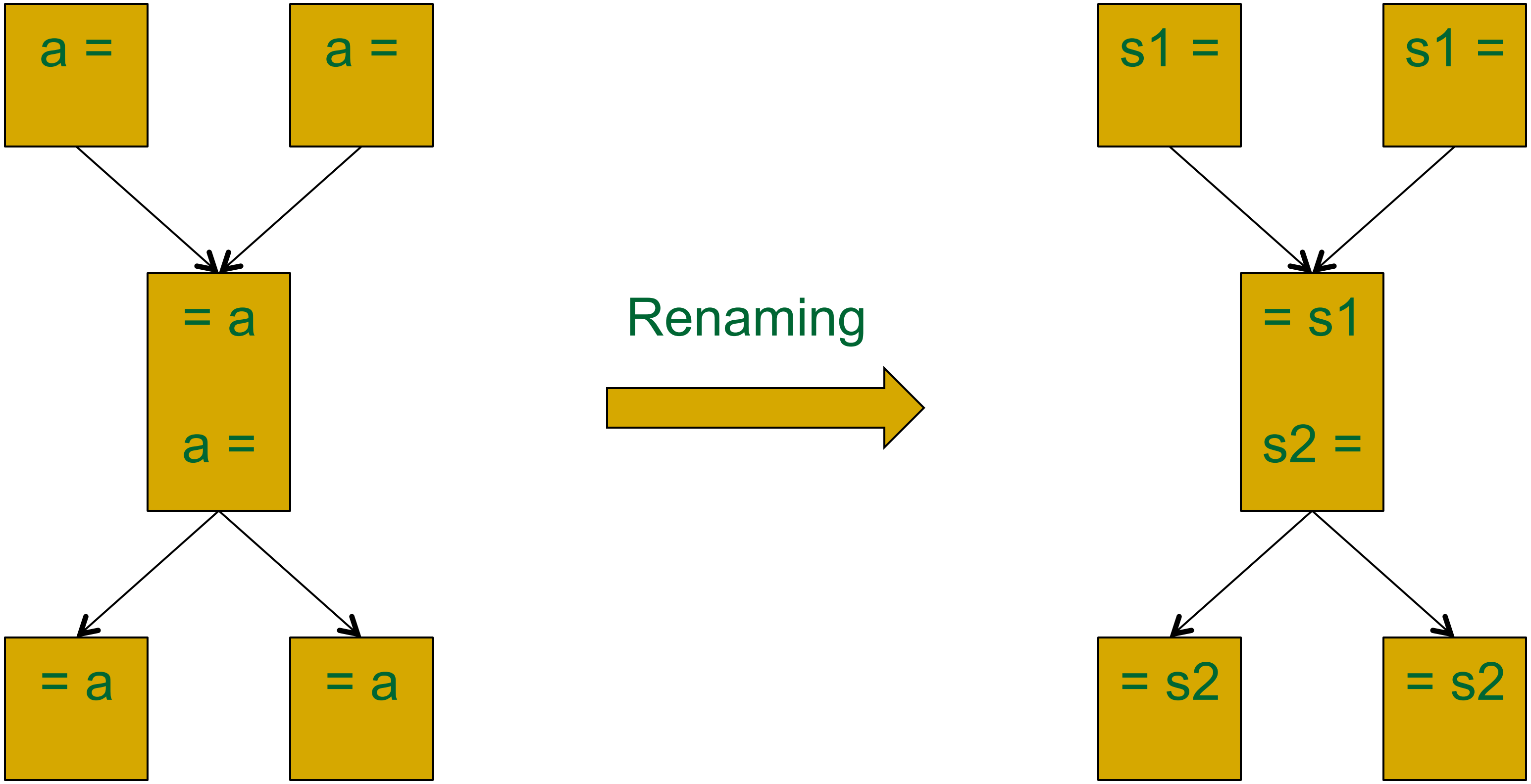
Steps in Chaitin's Algorithm

- Identify units for allocation
  - Renames variables/symbolic registers in the IR such that each live range has a unique name (number)
  - A live range is entitled to get a register
- Build the interference graph
- Coalesce by removing unnecessary move or copy instructions
- Colour the graph, thereby selecting registers
- Compute spill costs, simplify and add spill code till graph is colourable

Chaitin’s Framework

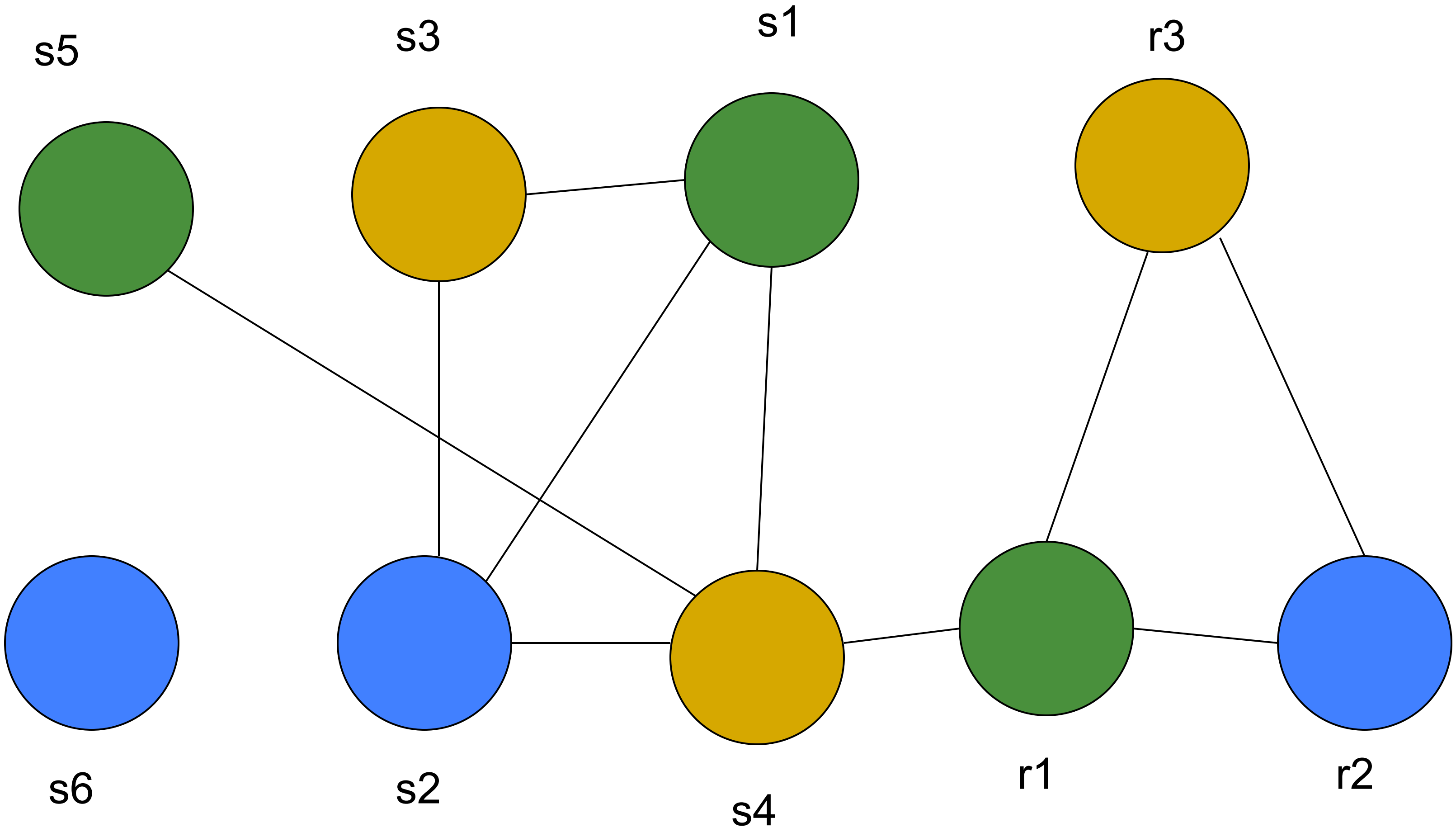


Example of Renaming



An Example

Original code	Code with symbolic registers
1. x= 2	1. s1=2; (lv of s1: 1-5)
2. y = 4	2. s2=4; (lv of s2: 2-5)
3. w = x+ y	3. s3=s1+s2; (lv of s3: 3-4)
4. z = x+1	4. s4=s1+1; (lv of s4: 4-6)
5. u = x*y	5. s5=s1*s2; (lv of s5: 5-6)
6. x= z*2	6. s6=s4*2; (lv of s6: 6- ...)



INTERFERENCE GRAPH
HERE ASSUME VARIABLE Z (s4) CANNOT OCCUPY r1

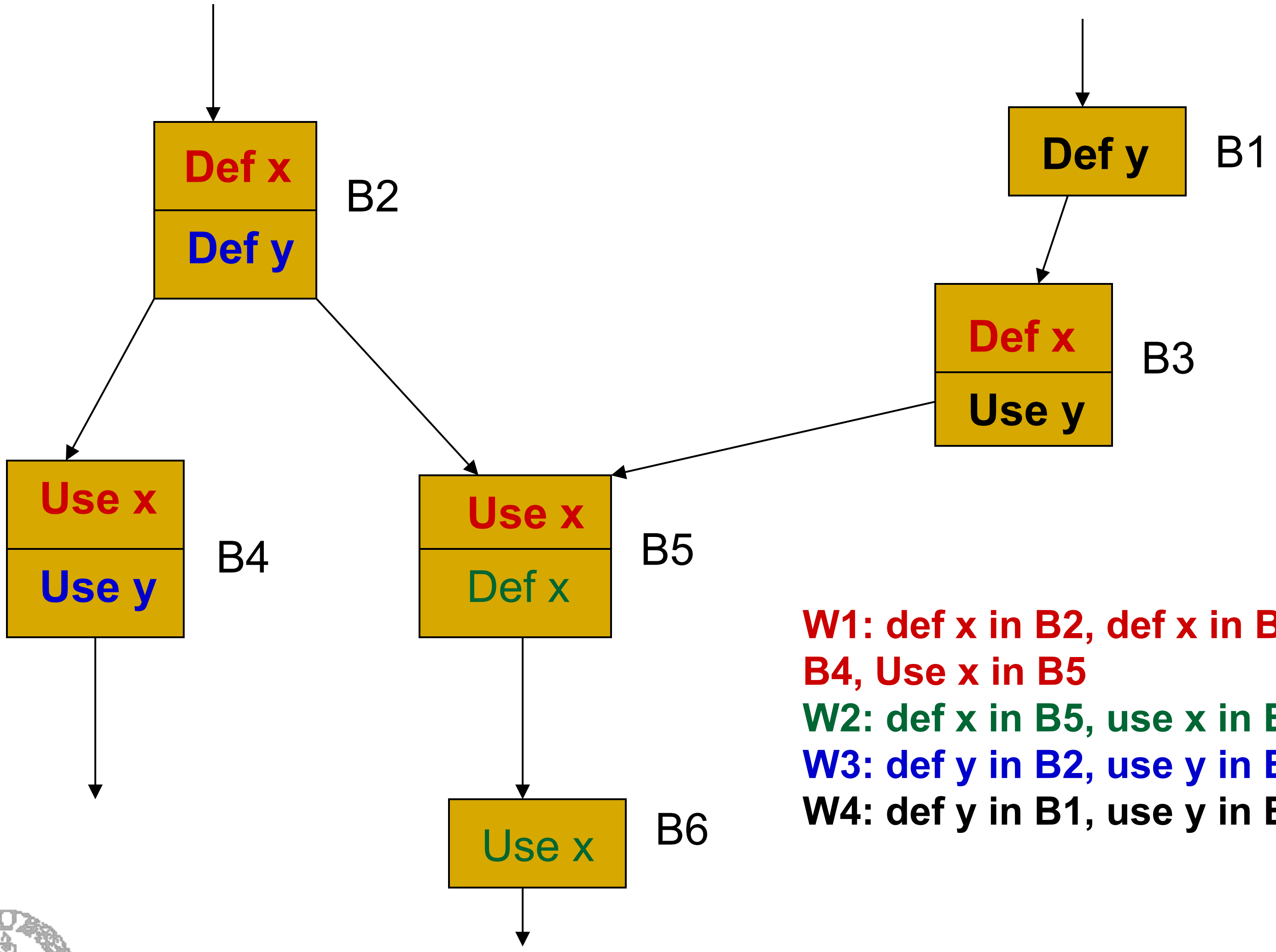
Example(continued)

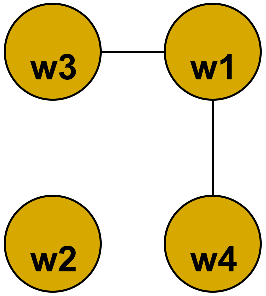
Final register allocated code

```
r1 = 2
r2= 4
r3= r1+r2
r3= r1+1
r1= r1 *r2
r2= r3+r2
```

Three registers are sufficient for no spills

More Complex Example





W1: def x in B2, def x in B3, use x in B4, Use x in B5
W2: def x in B5, use x in B6
W3: def y in B2, use y in B4
W4: def y in B1, use y in B3

Build Interference Graph

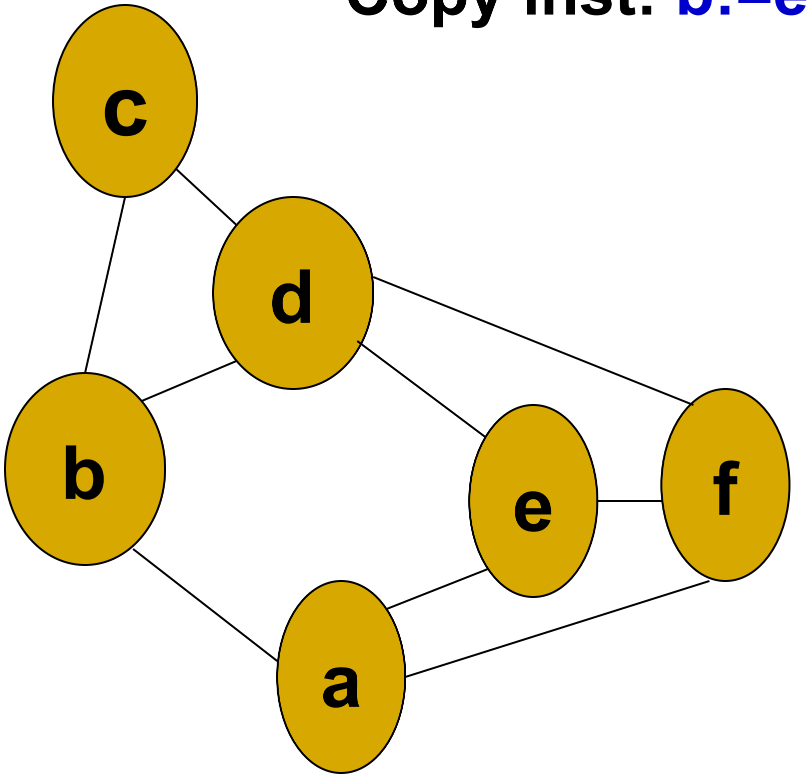
- Create a node for each LV and for each physical register in the interference graph
- If two distinct LVs interfere, that is, a variable associated with one LV is live at a definition point of another add an edge between the two LVs
- If a particular variable cannot reside in a register, add an edge between all LVs associated with that variable and the register

Copy Subsumption or Coalescing

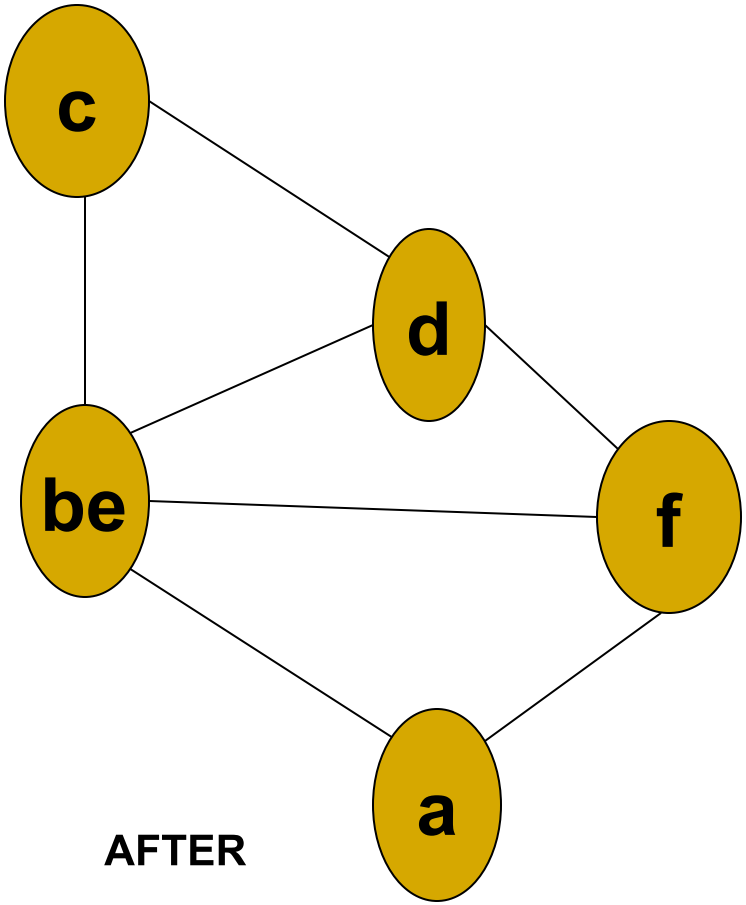
- Consider a copy instruction: b := e in the program
- If the live ranges of b and e do not overlap, then b and e can be given the same register (colour)
  - Implied by lack of any edges between b and e in the interference graph
- The copy instruction can then be removed from the final program
- Coalesce by merging b and e into one node that contains the edges of both nodes

Example of coalescing

Copy inst: b:=e

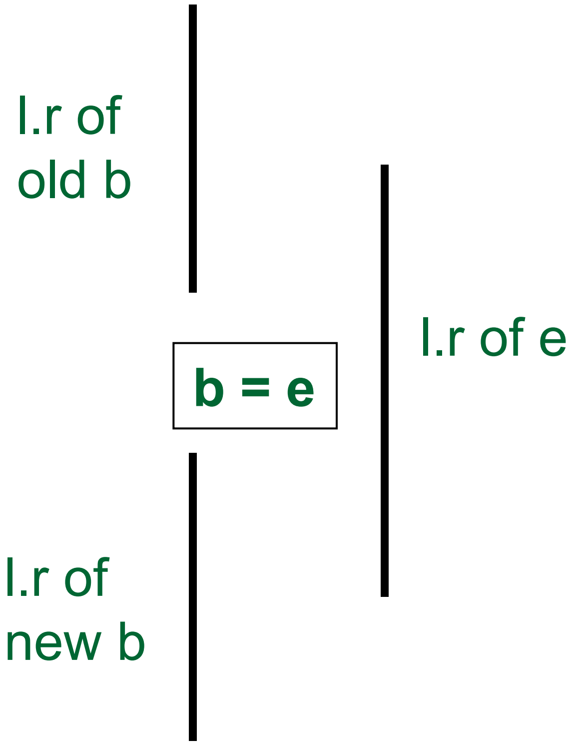


BEFORE

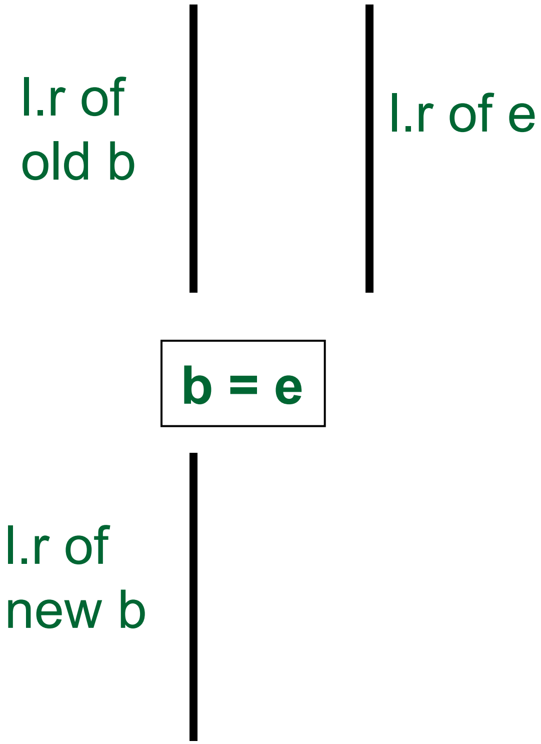


AFTER

Copy Subsumption or Coalescing

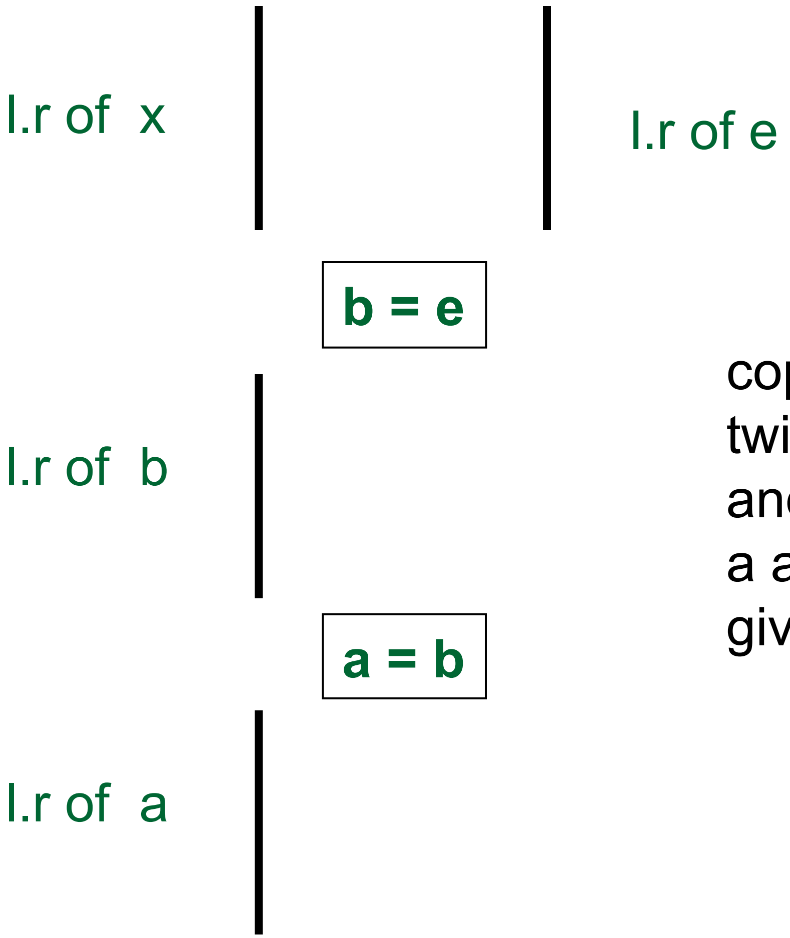


copy subsumption
is not possible; lr(e)
and lr(new b) interfere



copy subsumption is
possible; lr(e) and lr(new b)
do not interfere

Copy Subsumption Repeatedly



copy subsumption happens twice - once between b and e, and second time between a and b. e, b, and a are all given the same register.

Coalescing

- Coalesce all possible copy instructions
  - Rebuild the graph
    - may offer further opportunities for coalescing
    - build-coalesce phase is repeated till no further coalescing is possible.
- Coalescing reduces the size of the graph and possibly reduces spilling

Simple fact

- Suppose the no. of registers available is R.
- If a graph G contains a node ***n*** with fewer than R neighbors then removing ***n*** and its edges from G will not affect its R-colourability
- If G' = G-{n} can be coloured with R colours, then so can G.
  - After colouring G', just assign to ***n,*** a colour different from its R-1 neighbours.

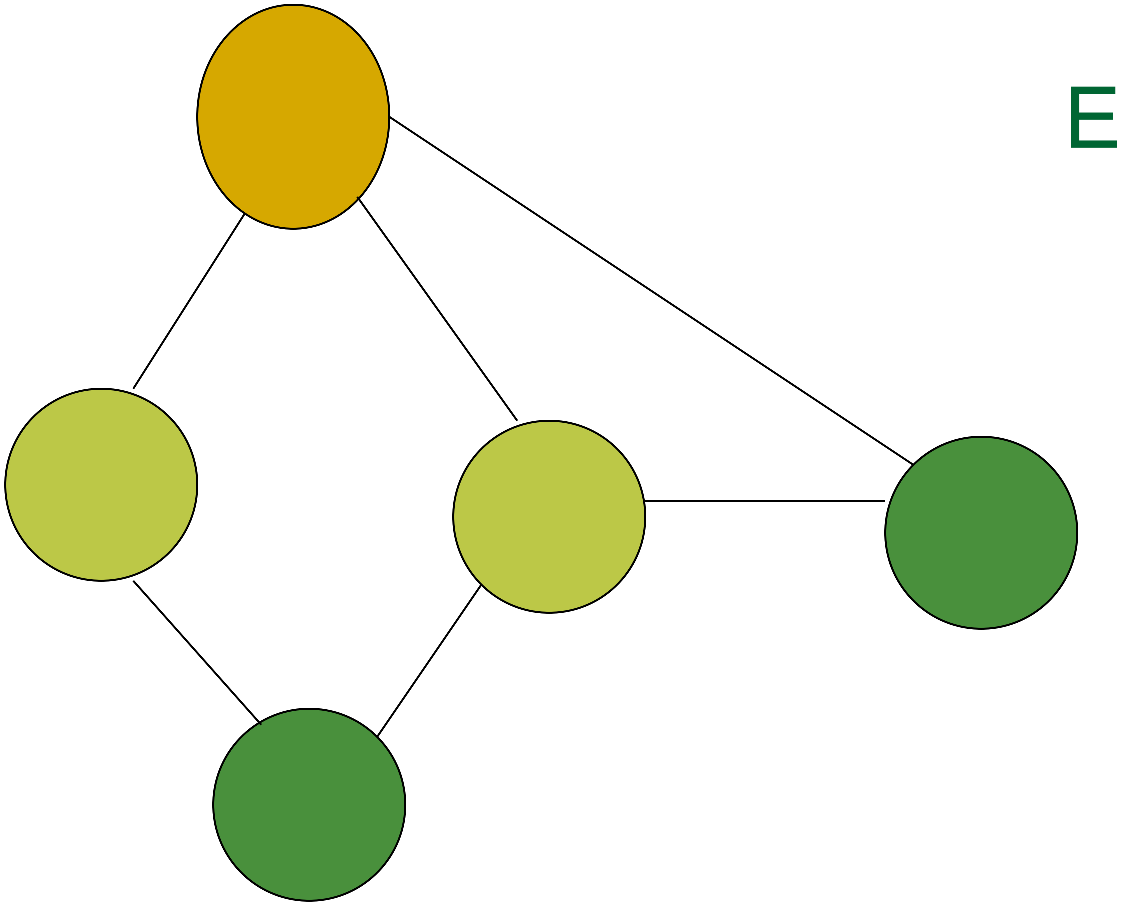
Simplification

- If a node ***n*** in the interference graph has degree less than R, remove ***n*** and all its edges from the graph and place ***n*** on a colouring stack.
- When no more such nodes are removable then we need to spill a node.
- Spilling a variable x implies
  - loading x into a register at every use of x
  - storing x from register into memory at every definition of x

Spilling Cost

- The node to be spilled is decided on the basis of a spill cost for the live range represented by the node.
- Chaitin's estimate of spill cost of a live range v
  - cost(v) = $$\sum_{\substack{\text{all load or store}\\ \text{operations in}\\ \text{a live range v}}} c * 10^{d}$$
  - where *c* is the cost of the op and *d*, the loop nesting depth.
  - 10 in the eqn above approximates the no. of iterations of any loop
  - The node to be spilled is the one with MIN(cost(v)/deg(v))

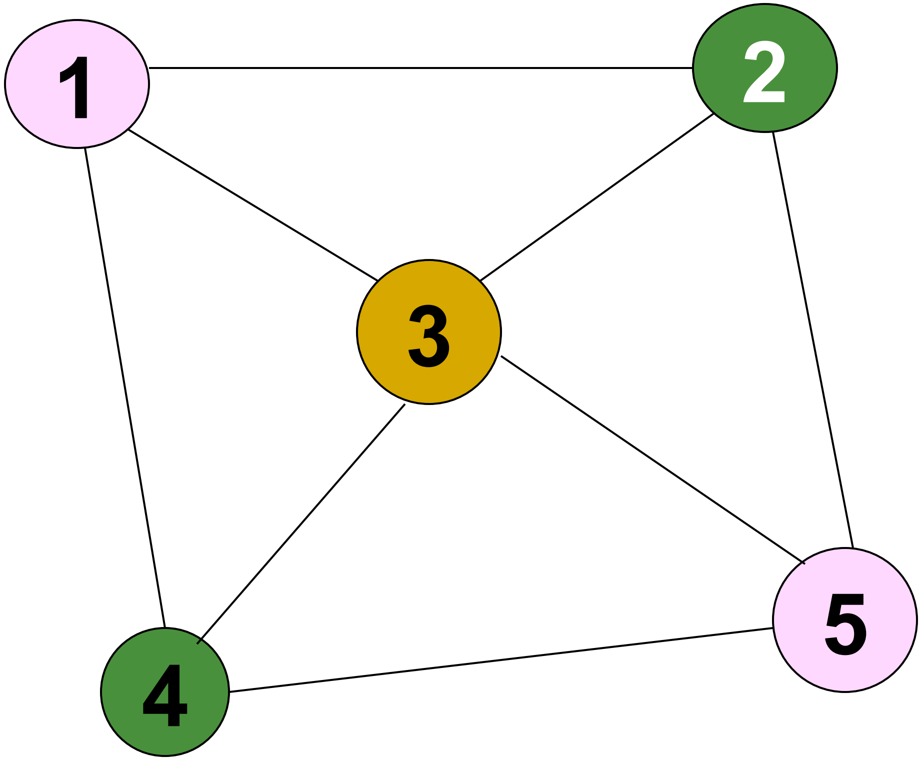
Example



Here R = 3 and the graph is 3-colourable
No spilling is necessary

Example

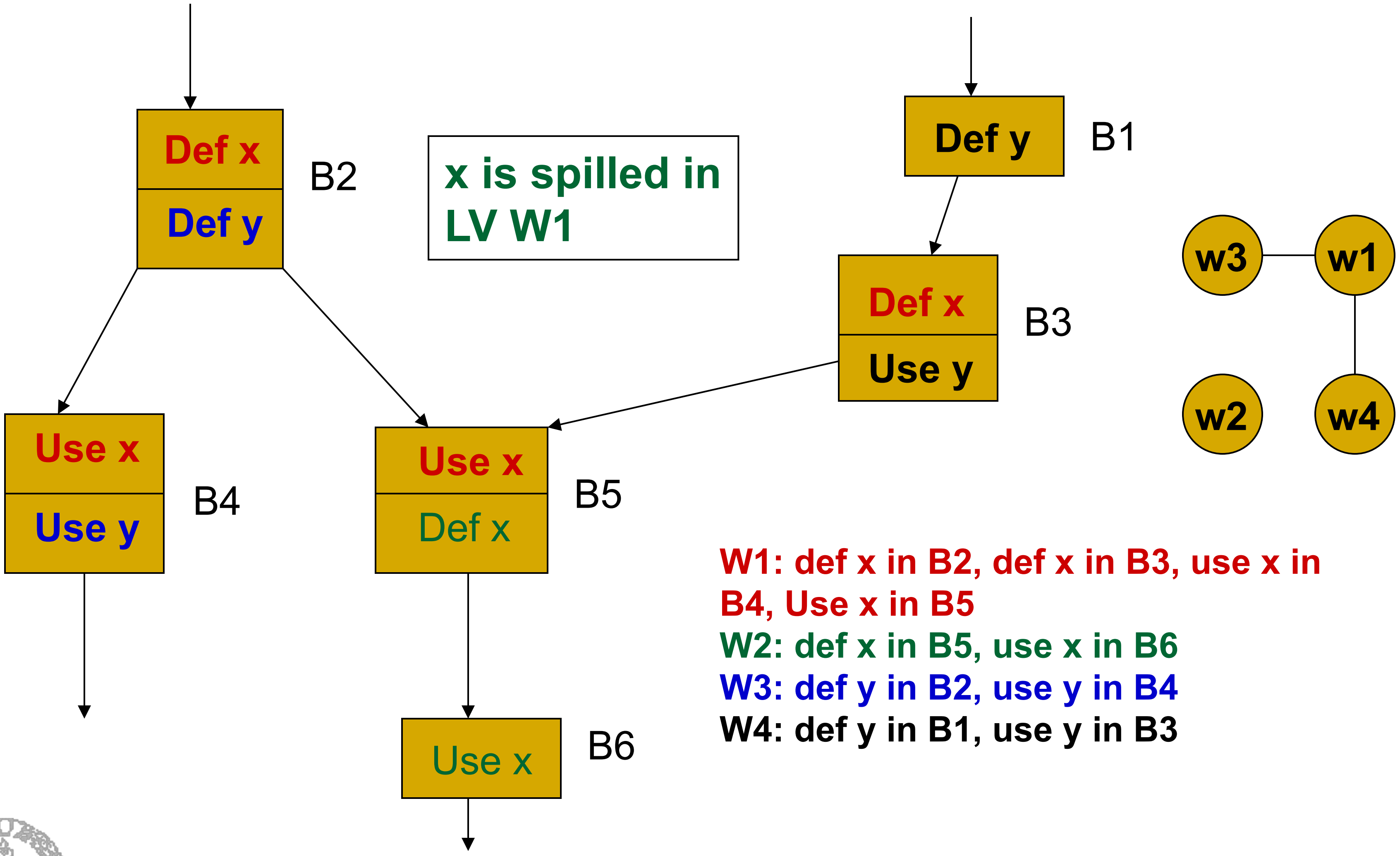
A 3-colourable graph which is not 3-coloured by colouring heuristic



Spilling a Node

- To spill a node we remove it from the graph and represent the effect of spilling as follows (It cannot be simply removed from the graph).
  - Reload the spilled object at each use and store it in memory at each definition point
  - This creates new small live ranges which will also need registers.
- After all spill decisions are made, insert spill code, rebuild the interference graph and then repeat the attempt to colour.
- When simplification yields an empty graph then select colours, that is, registers

Effect of Spilling



x is spilled in LV W1

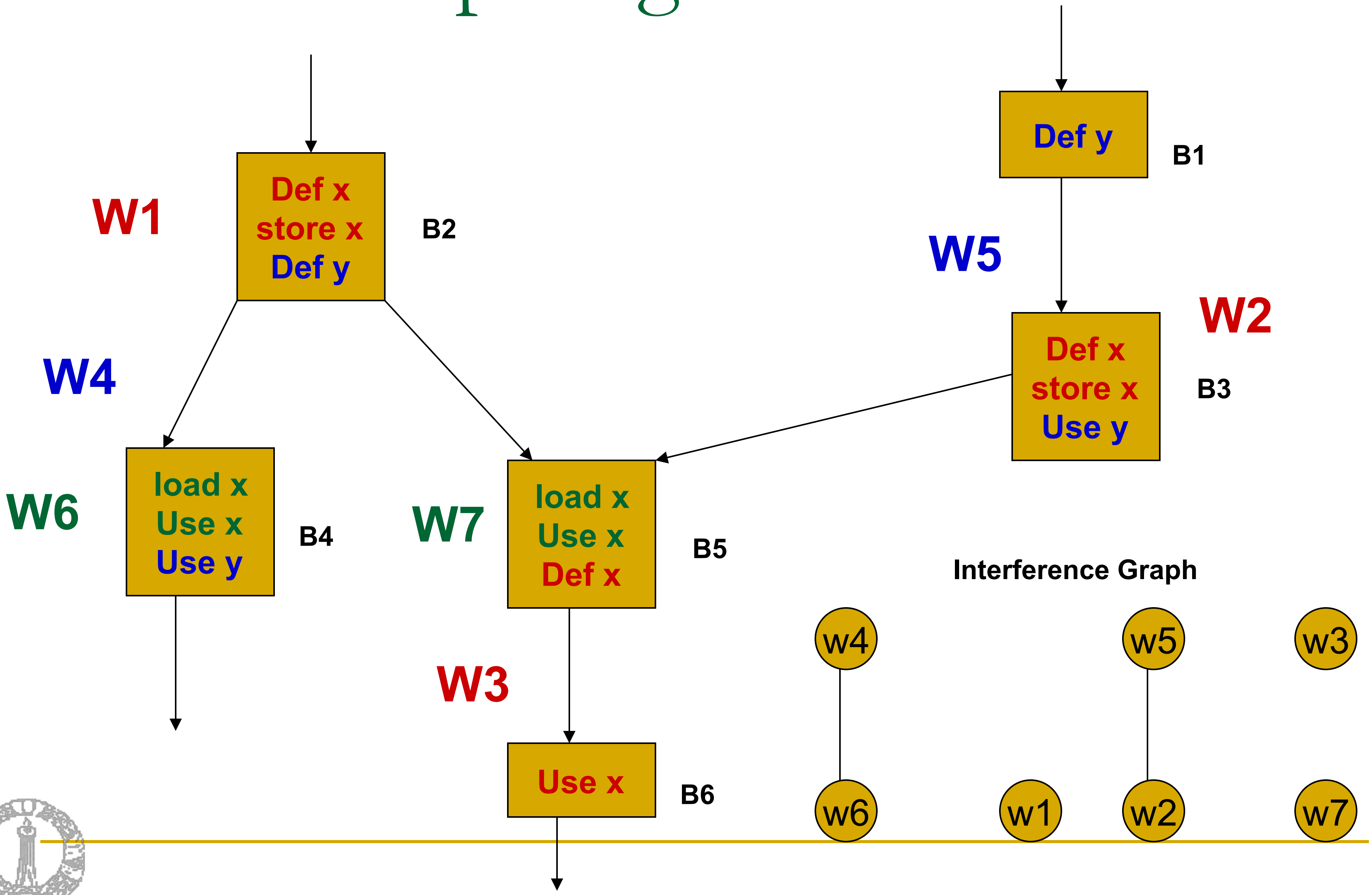
W1: def x in B2, def x in B3, use x in B4, Use x in B5

W2: def x in B5, use x in B6

W3: def y in B2, use y in B4

W4: def y in B1, use y in B3

Effect of Spilling



Colouring the Graph(selection)

```
Repeat
v= pop(stack).
Colours_used(v) = colours used by neighbours of v.
Colours_free(v) = all colours - Colours_used(v).
Colour (v) = choose any colour in Colours_free(v).
Until stack is empty
```

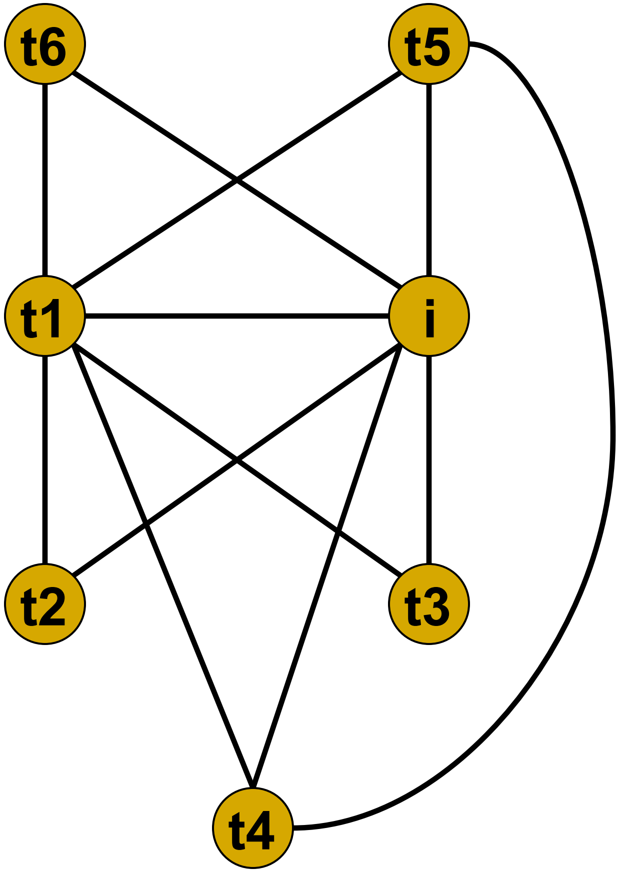
- Convert the colour assigned to a symbolic register to the corresponding real register's name in the code.

A Complete Example

```
1.        t1 = 202
2.        i = 1
3.  L1:   t2 = i>100
4.        if t2 goto L2
5.        t1 = t1-2
6.        t3 = addr(a)
7.        t4 = t3 - 4
8.        t5 = 4*i
9.        t6 = t4 + t5
10.       *t6 = t1
11.       i = i+1
12.       goto L1
13. L2:
```

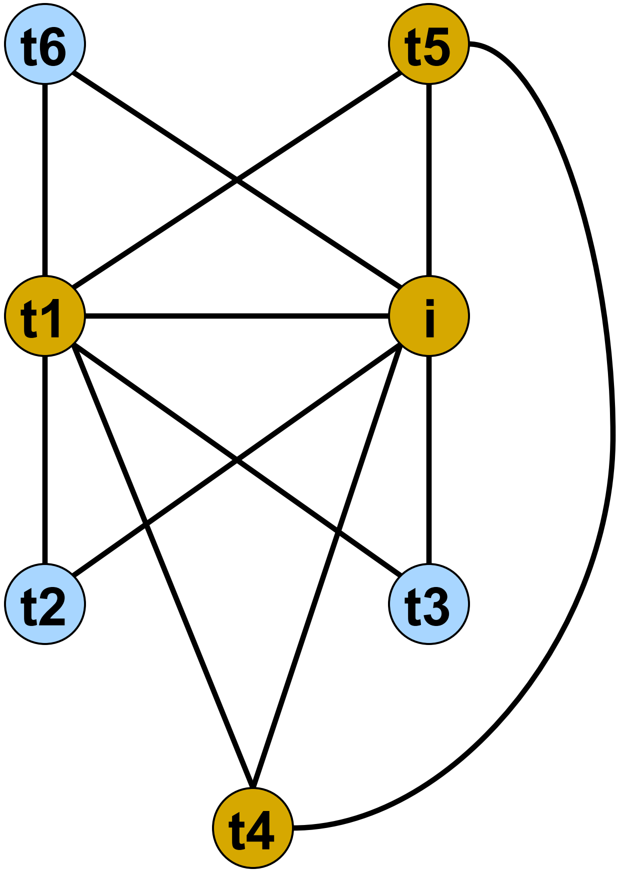
variable	live range
t1	1-10
i	2-11
t2	3-4
t3	6-7
t4	7-9
t5	8-9
t6	9-10

A Complete Example

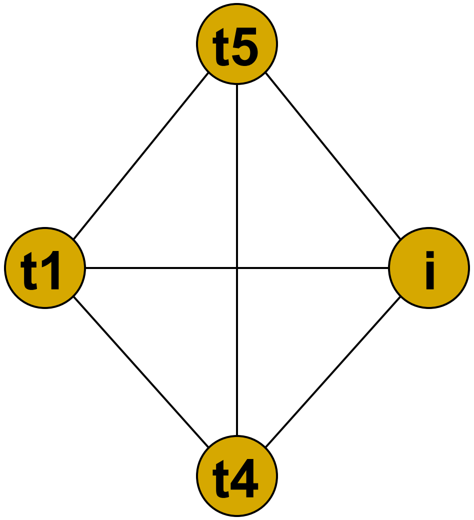


variable	**live range**
t1	1-10
i	2-11
t2	3-4
t3	6-7
t4	7-9
t5	8-9
t6	9-10

A Complete Example



Assume 3 registers. Nodes t6,t2, and t3 are first pushed onto a stack during reduction.



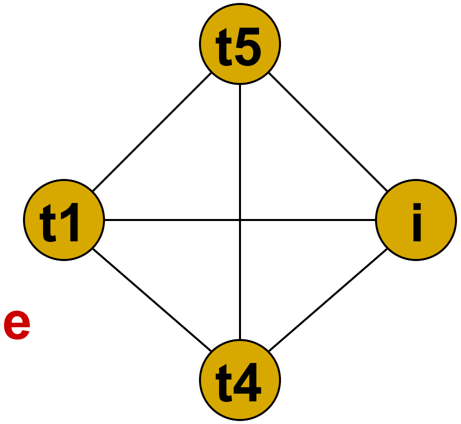
This graph cannot be reduced further. Spilling is necessary.

A Complete Example

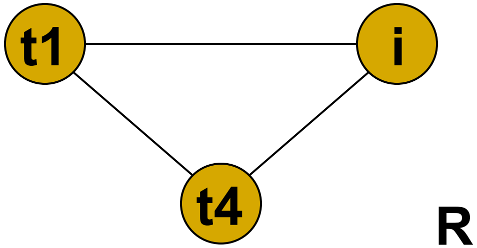
Node V	Cost(v)	deg(v)	$h_0(v)$
t1	31	3	10
i	41	3	14
t4	20	3	7
t5	20	3	7

```
        t1 = 202
        i = 1
L1:     t2 = i>100
        if t2 goto L2
        t1 = t1-2
        t3 = addr(a)
        t4 = t3 - 4
        t5 = 4*i
        t6 = t4 + t5
       *t6 = t1
        i = i+1
        goto L1
L2:
```

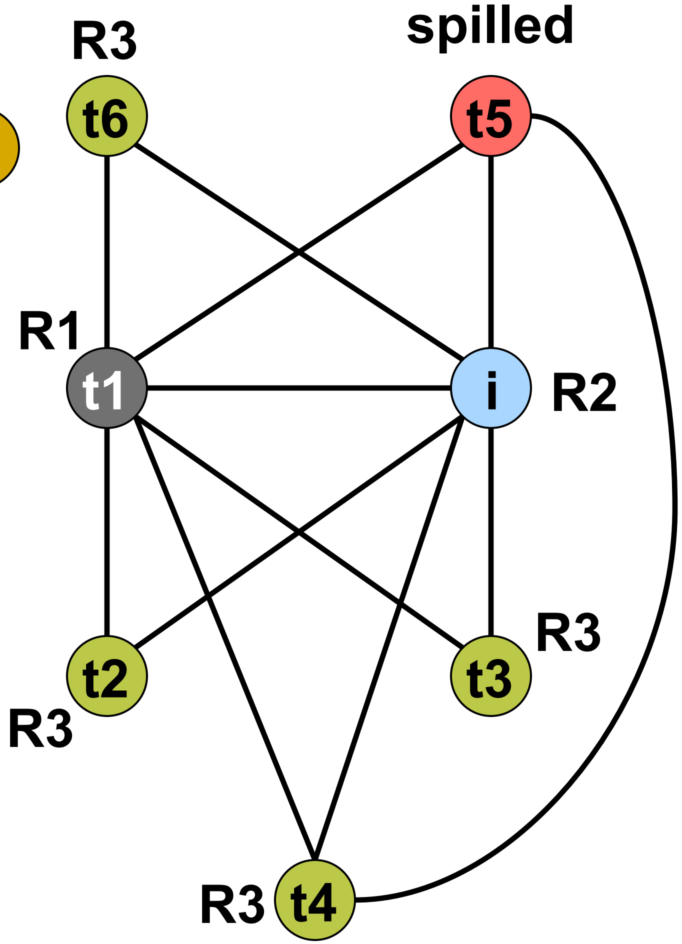
t1: 1+(1+1+1)*10 = 31
i : 1+(1+1+1+1)*10 = 41
t4: (1+1)*10 = 20
t5: (1+1)*10 = 20
t5 will be spilled. Then the graph can be coloured.



A Complete Example



i
t1
t4
t3
t2
t6



```
        R1 = 202
        R2 = 1
L1:     R3 = i>100
        if R3 goto L2
        R1 = R1 - 2
        R3 = addr(a)
        R3 = R3 - 4
        t5 = 4*R2
        R3 = R3 + t5
        *R3 = R1
        R2 = R2+1
        goto L1
L2:
```

t5: spilled node, will be provided with a temporary register during code generation

Drawbacks of the Algorithm

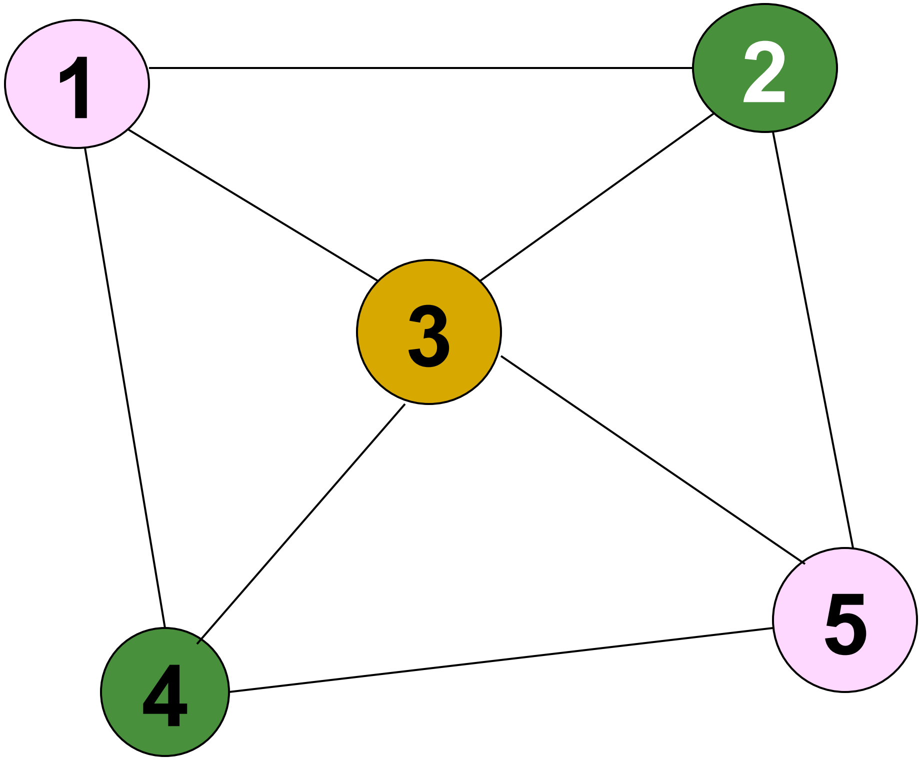
- Constructing and modifying interference graphs is very costly as interference graphs are typically huge.
- For example, the combined interference graphs of procedures and functions of gcc in mid-90’s have approximately 4.6 million edges.

Some modifications

- Careful coalescing: Do not coalesce if coalescing increases the degree of a node to more than the number of registers
- Optimistic colouring: When a node needs to be spilled, push it onto the colouring stack instead of spilling it right away
  - spill it only when it is popped and if there is no colour available for it
  - this could result in colouring graphs that need spills using Chaitin’s technique.

Example

A 3-colourable graph which is not 3-coloured by colouring heuristic, but coloured by optimistic colouring



Say, 1 is chosen for spilling. Push it onto the stack, and remove it from the graph. The remaining graph (2,3,4,5) is 3-colourable. Now, when 1 is popped from the colouring stack, there is a colour with which 1 can be coloured. It need not be spilled.

Introduction to Machine-Independent Optimizations - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization?
- Illustrations of code optimizations
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

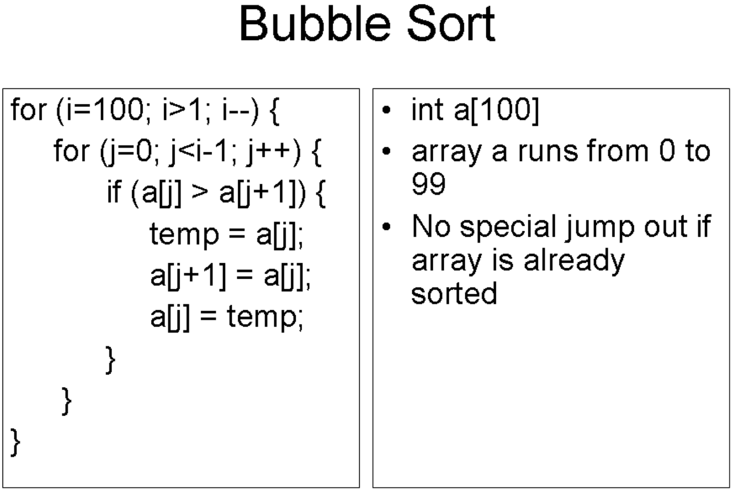
Machine-independent Code Optimization

- Intermediate code generation process introduces many inefficiencies
  - Extra copies of variables, using variables instead of constants, repeated evaluation of expressions, etc.
- Code optimization removes such inefficiencies and improves code
- Improvement may be time, space, or power consumption
- It changes the structure of programs, sometimes of beyond recognition
  - Inlines functions, unrolls loops, eliminates some programmer-defined variables, etc.
- Code optimization consists of a bunch of heuristics and percentage of improvement depends on programs (may be zero also)

Examples of Machine-Independant Optimizations

- Global common sub-expression elimination
- Copy propagation
- Constant propagation and constant folding
- Loop invariant code motion
- Induction variable elimination and strength reduction
- Partial redundancy elimination
- Loop unrolling
- Function inlining
- Tail recursion removal
- Vectorization and Concurrentization
- Loop interchange, and loop blocking

Bubble Sort Running Example

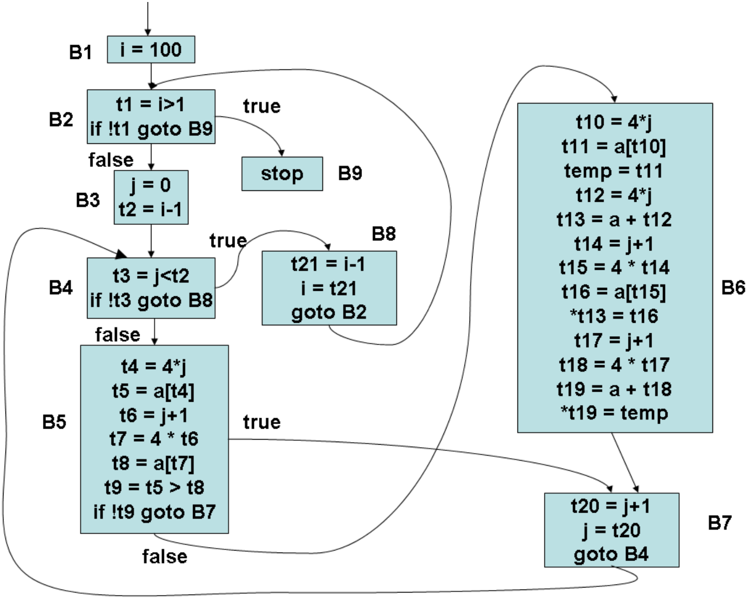


Bubble Sort

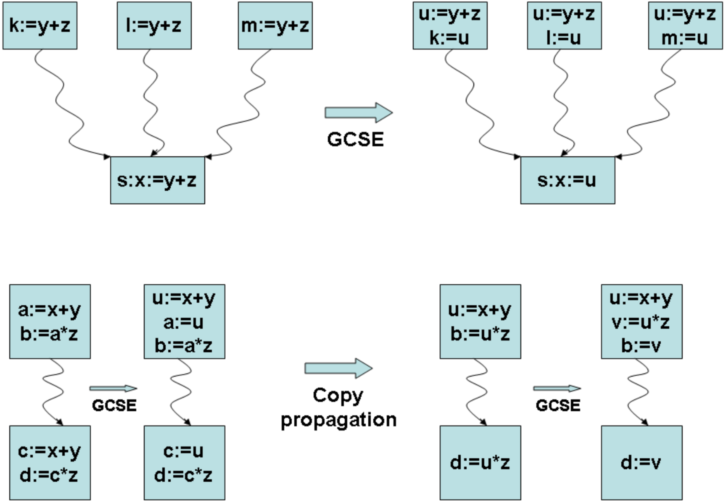
```
for (i=100; i>1; i--) {
    for (j=0; j<i-1; j++) {
        if (a[j] > a[j+1]) {
            temp = a[j];
            a[j+1] = a[j];
            a[j] = temp;
        }
    }
}
```

- int a[100]
- array a runs from 0 to 99
- No special jump out if array is already sorted

Control Flow Graph of Bubble Sort



GCSE Conceptual Example



k:=y+z

l:=y+z

m:=y+z

s:x:=y+z

GCSE

u:=y+z
k:=u

u:=y+z
l:=u

u:=y+z
m:=u

s:x:=u

a:=x+y
b:=a*z

c:=x+y
d:=c*z

GCSE

u:=x+y
a:=u
b:=a*z

c:=u
d:=c*z

Copy propagation

u:=x+y
b:=u*z

d:=u*z

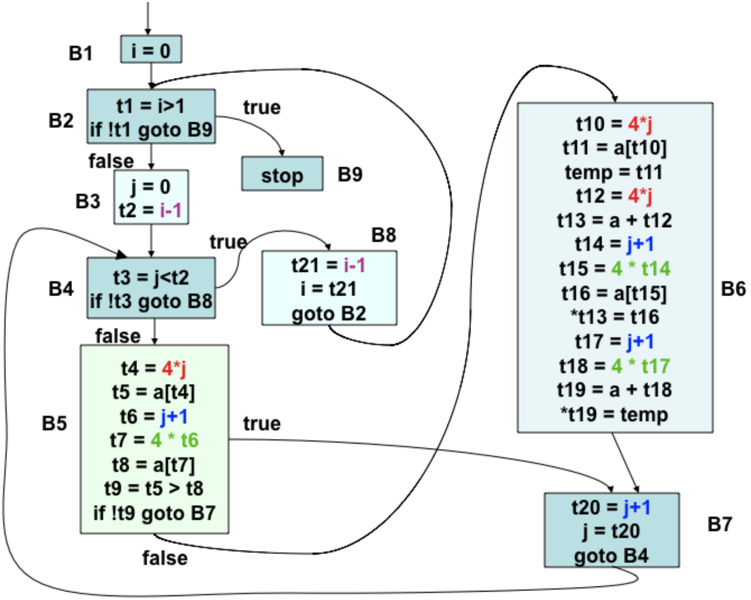
GCSE

u:=x+y
v:=u*z
b:=v

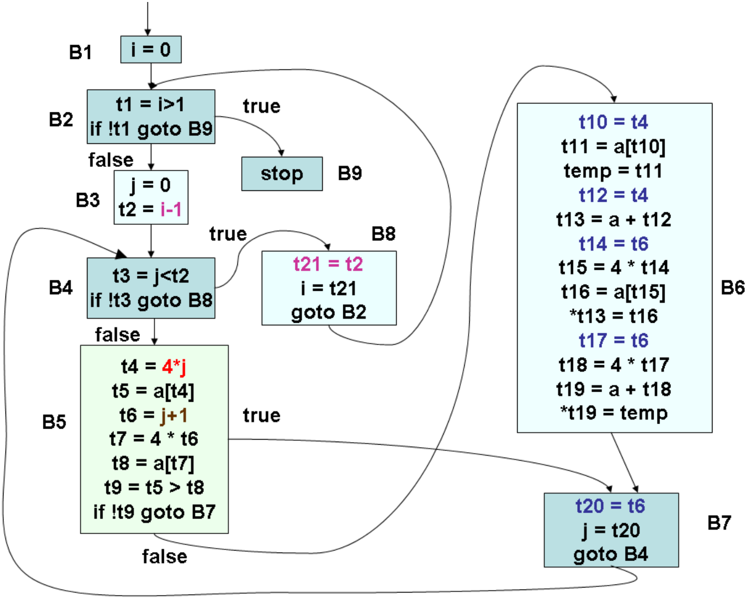
d:=v

Demonstrating the need for repeated application of GCSE

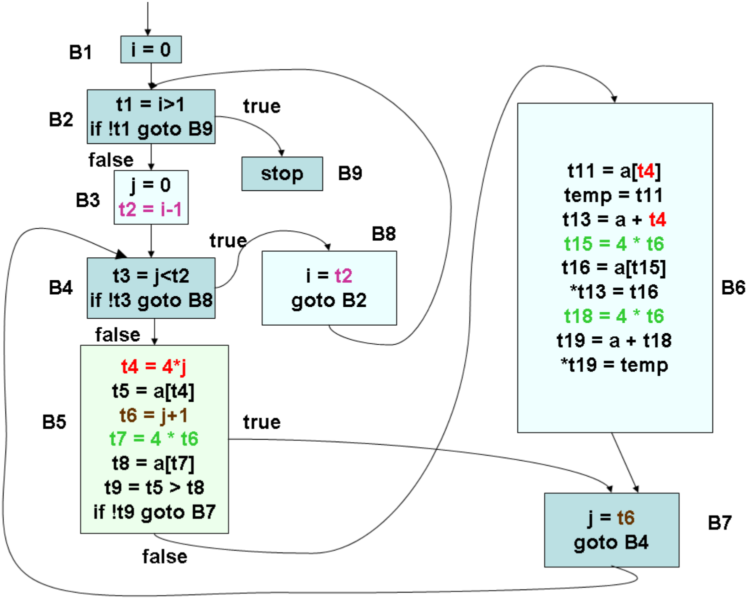
GCSE on Running Example - 1



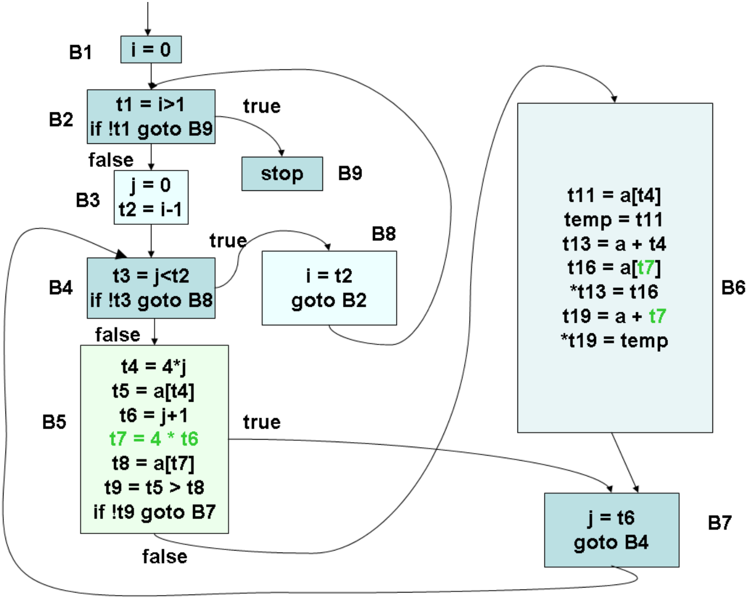
GCSE on Running Example - 2



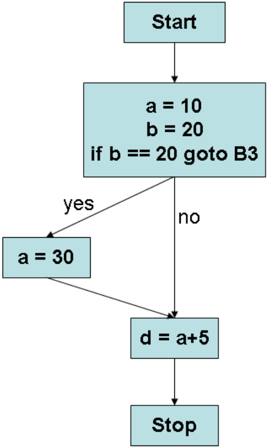
Copy Propagation on Running Example



GCSE and Copy Propagation on Running Example



Constant Propagation and Folding Example



Start

a = 10
b = 20
if b == 20 goto B3

yes

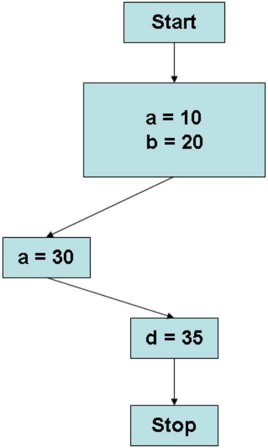
no

a = 30

d = a+5

Stop

Before constant propagation



Start

a = 10
b = 20

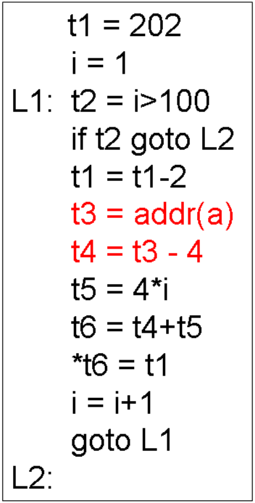
a = 30

d = 35

Stop

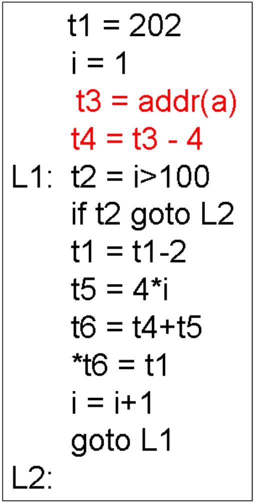
After constant propagation and folding

Loop Invariant Code motion Example



```
      t1 = 202
      i = 1
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t3 = addr(a)
      t4 = t3 - 4
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before LIV code motion



```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

After LIV code motion

Strength Reduction

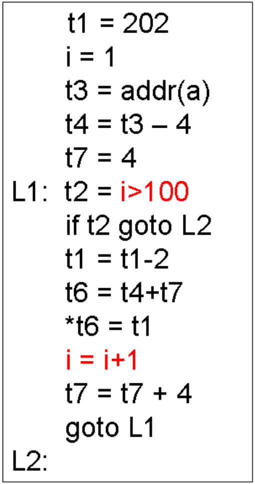
```
      t1 = 202
      i = 1
       t3 = addr(a)
      t4 = t3 - 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t5 = 4*i
      t6 = t4+t5
      *t6 = t1
      i = i+1
      goto L1
L2:
```

Before strength reduction for t5

```
      t1 = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1 = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

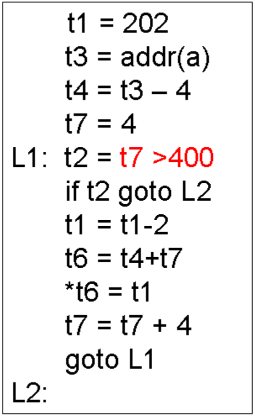
After strength reduction for t5 and copy propagation

Induction Variable Elimination



```
      t1  = 202
      i = 1
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = i>100
      if t2 goto L2
      t1  = t1-2
      t6 = t4+t7
      *t6 = t1
      i = i+1
      t7 = t7 + 4
      goto L1
L2:
```

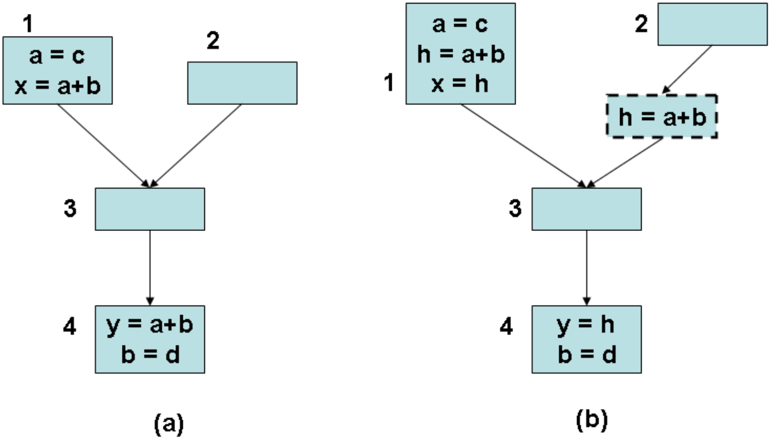
Before induction variable elimination (i)



```
      t1  = 202
      t3 = addr(a)
      t4 = t3 – 4
      t7 = 4
L1:   t2 = t7 >400
      if t2 goto L2
      t1  = t1-2
      t6 = t4+t7
      *t6 = t1
      t7 = t7 + 4
      goto L1
L2:
```

After eliminating i and replacing it with t7

Partial Redundancy Elimination



Unrolling a For-loop

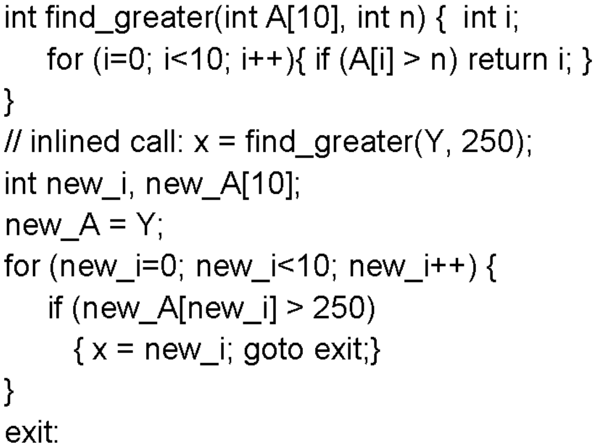
```
for (i = 0; i<N; i++) { S_1(i); S_2(i); }

for (i = 0; i+3 < N; i+=3) {
    S_1(i); S_2(i);
    S_1(i+1); S_2(i+1);
    S_1(i+2); S_2(i+2);
}
// remaining few iterations, 1,2, or 3:
// (((N-1) mod 3)+1)
for (k=i; k<N; k++) { S_1(k); S_2(k); }
```

Unrolling While and Repeat loops

```
while (C) { $S_1$; $S_2$; }

while (C) {
    $S_1$; $S_2$;
    if (!C) break;
    $S_1$; $S_2$;
    if (!C) break;
    $S_1$; $S_2$;
}
```

```
repeat { $S_1$; $S_2$; } until C;

repeat {
    $S_1$; $S_2$;
    if (C) break;
    $S_1$; $S_2$;
    if (C) break;
    $S_1$; $S_2$;
} until C;
```

Function Inlining



```
int find_greater(int A[10], int n) {  int i;
    for (i=0; i<10; i++){ if (A[i] > n) return i; }
}
// inlined call: x = find_greater(Y, 250);
int new_i, new_A[10];
new_A = Y;
for (new_i=0; new_i<10; new_i++) {
    if (new_A[new_i] > 250)
      { x = new_i; goto exit;}
}
exit:
```

Tail Recursion Removal

```
void sum (int A[], int n, int* x) {
    if (n==0) *x = *x+ A[0]; else {
        *x = *x+A[n];  sum(A, n-1, x);
    }
}
// after removal of tail recursion
void sum (int A[], int n, int* x) {
   while (true) { if (n==0) {*x=*x+A[0]; break;}
                  else{ *x=*x + A[n]; n=n-1; continue;}
   }
}
```

Vectorization and Concurrentization Example 1

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

X(1:100) = X(1:100) + Y(1:100)

or

forall I = 1 to 100 do X(I) = X(I) + Y(I)
```

Vectorization Example 2

```
for I = 1 to 100 do {
   X(I+1) = X(I) + Y(I)
}

cannot be converted to

X(2:101) = X(1:100) + Y(1:100)
or equivalent concurrent code

because of dependence as shown below

X(2) = X(1) + Y(1)
X(3) = X(2) + Y(2)
X(4) = X(3) + Y(3)
...
```

Loop Interchange for parallelizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is not parallelizable, but inner loop is

Less work per thread

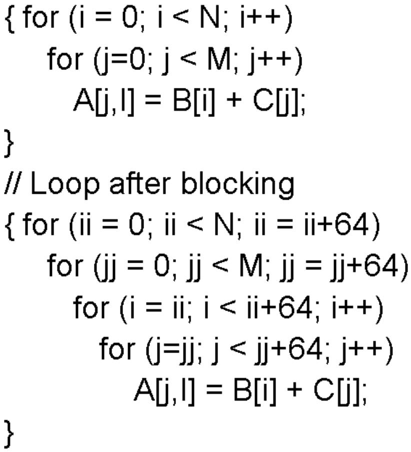
```
for J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is parallelizable but inner loop is not

More work per thread

```
forall J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Loop Blocking



```
{ for (i = 0; i < N; i++)
    for (j=0; j < M; j++)
      A[j,I] = B[i] + C[j];
}
// Loop after blocking
{ for (ii = 0; ii < N; ii = ii+64)
    for (jj = 0; jj < M; jj = jj+64)
      for (i = ii; i < ii+64; i++)
        for (j=jj; j < jj+64; j++)
          A[j,I] = B[i] + C[j];
}
```

Fundamentals of Data-flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Data-flow analysis

- These are techniques that derive information about the flow of data along program execution paths
- An *execution path* (or *path*) from point p_1 to point p_n is a sequence of points $p_1, p_2, ..., p_n$ such that for each $i = 1, 2, ..., n-1$, either
    1. p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
    2. p_i is the end of some block and p_{i+1} is the beginning of a successor block
- In general, there is an infinite number of paths through a program and there is no bound on the length of a path
- Program analyses summarize all possible program states that can occur at a point in the program with a finite set of facts
- No analysis is necessarily a perfect representation of the state

Uses of Data-flow Analysis

- Program debugging
    - Which are the definitions (of variables) that *may* reach a program point? These are the *reaching definitions*
- Program optimizations
    - Constant folding
    - Copy propagation
    - Common sub-expression elimination etc.

Data-Flow Analysis Schema

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
  - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of of definitions in the program
  - A particular data-flow value is a set of definitions
- $IN[s]$ and $OUT[s]$: data-flow values *before* and *after* each statement s
- The *data-flow problem* is to find a solution to a set of constraints on $IN[s]$ and $OUT[s]$, for all statements s

Introduction to Machine-Independent Optimizations - 2

Data-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

Data-Flow Analysis Schema

- A *data-flow value* for a program point represents an abstraction of the set of all possible program states that can be observed for that point
- The set of all possible data-flow values is the *domain* for the application under consideration
    - Example: for the *reaching definitions* problem, the domain of data-flow values is the set of all subsets of of definitions in the program
    - A particular data-flow value is a set of definitions
- *IN*[*s*] and *OUT*[*s*]: data-flow values *before* and *after* each statement *s*
- The *data-flow problem* is to find a solution to a set of constraints on *IN*[*s*] and *OUT*[*s*], for all statements *s*

Data-Flow Analysis Schema (2)

- Two kinds of constraints
  - Those based on the semantics of statements (*transfer functions*)
  - Those based on flow of control
- A DFA schema consists of
  - A control-flow graph
  - A direction of data-flow (forward or backward)
  - A set of data-flow values
  - A confluence operator (usually set union or intersection)
  - Transfer functions for each block
- We always compute *safe* estimates of data-flow values
- A decision or estimate is *safe* or *conservative*, if it never leads to a change in what the program computes (after the change)
- These safe values may be either subsets or supersets of actual values, based on the application

The Reaching Definitions Problem

- We *kill* a definition of a variable *a*, if between two points along the path, there is an assignment to *a*
- A definition *d* reaches a point *p*, if there is a path from the point immediately following *d* to *p*, such that *d* is not *killed* along that path
- Unambiguous and ambiguous definitions of a variable

  ```
  a := b+c
  ```

  (unambiguous definition of 'a')

  ```
  ...
  *p := d
  ```

  (ambiguous definition of 'a', if 'p' may point to variables other than 'a' as well; hence does not kill the above definition of 'a')

  ```
  ...
  a := k-m
  ```

  (unambiguous definition of 'a'; kills the above definition of 'a')

The Reaching Definitions Problem(2)

- We compute supersets of definitions as *safe* values
- It is safe to assume that a definition reaches a point, even if it does not.
- In the following example, we assume that both `a=2` and `a=4` reach the point after the complete if-then-else statement, even though the statement `a=4` is not reached by control flow

  ```
  if (a==b) a=2; else if (a==b) a=4;
  ```

The Reaching Definitions Problem (3)

- The data-flow equations (constraints)

$$\begin{aligned} IN[B] &= \bigcup_{P \text{ is a predecessor of } B} OUT[P] \\ OUT[B] &= GEN[B] \bigcup (IN[B] - KILL[B]) \\ IN[B] &= \phi, \text{for all } B \text{ (initialization only)} \end{aligned}$$

- If some definitions reach B_1 (entry), then $IN[B_1]$ is initialized to that set
- Forward flow DFA problem (since $OUT[B]$ is expressed in terms of $IN[B]$), confluence operator is $\cup$
  - Direction of flow does not imply traversing the basic blocks in a particular order
  - The final result does not depend on the order of traversal of the basic blocks

The Reaching Definitions Problem (4)

- *GEN*[*B*] = set of all definitions inside *B* that are “visible” immediately after the block - *downwards exposed* definitions
  - If a variable *x* has two or more defintions in a basic block, then only the last definition of *x* is downwards exposed; all others are not visible outside the block
- *KILL*[*B*] = union of the definitions in all the basic blocks of the flow graph, that are killed by individual statements in *B*
  - If a variable *x* has a definition d_i in a basic block, then d_i kills all the definitions of the variable *x* in the program, except d_i

Reaching Definitions Analysis: GEN and KILL

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

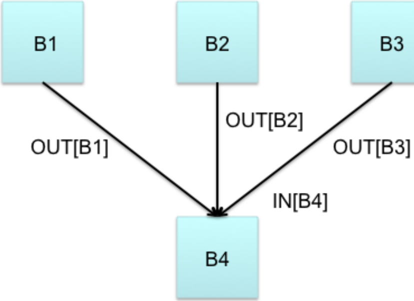
d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

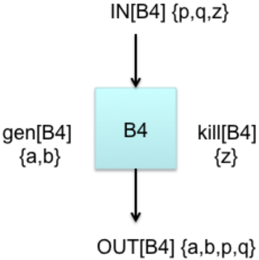
B

Set of all definitions = {d1,d2,d3,d4,d5,d6,d7,d8,d9,10}

GEN[B] = {d2,d3,d4}
KILL[B] = {d4,d9,d5,d10,d1}

Reaching Definitions Analysis: DF Equations





IN[B4] = OUT[B1] U OUT[B2] U OUT[B3]

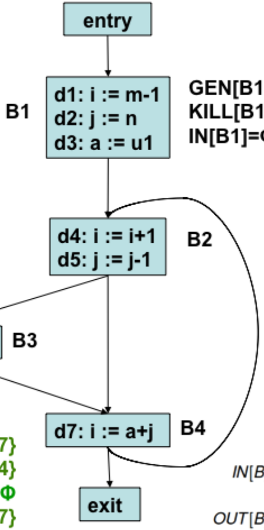
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

OUT[B4] = gen[B4] U (IN[B4] – kill[B4])

Reaching Definitions Analysis: An Example - Pass 1

Pass 1



entry

B1

d1: i := m-1
d2: j := n
d3: a := u1

GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]=Φ
OUT[B2]={d4,d5}

d4: i := i+1
d5: j := j-1

B2

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]=Φ
OUT[B3]={d6}

d6: a := u2

B3

d7: i := a+j

B4

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]=Φ
OUT[B4]={d7}

exit

Adapted from the
“Dragon Book”,
A-W, 1986

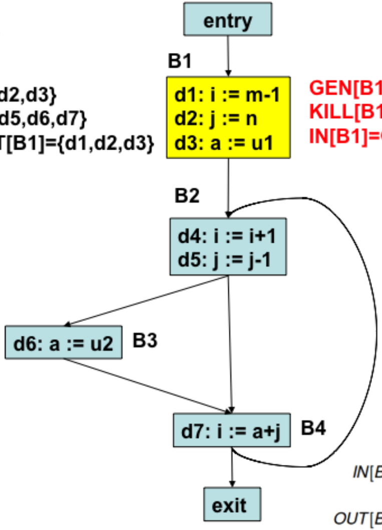
$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Pass 2.1

Pass 2

GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}



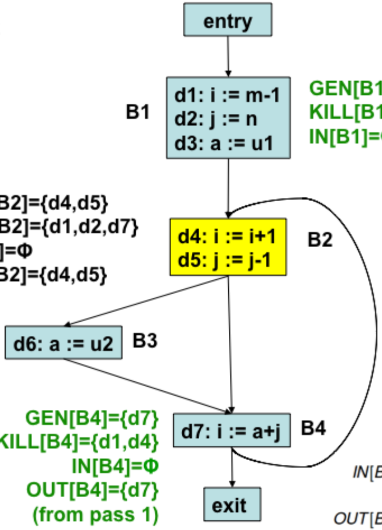
GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Pass 2.2

Pass 2



entry

B1

d1: i := m-1
d2: j := n
d3: a := u1

GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]=Φ
OUT[B2]={d4,d5}

d4: i := i+1
d5: j := j-1

B2

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

d6: a := u2

B3

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]=Φ
OUT[B4]={d7}
(from pass 1)

d7: i := a+j

B4

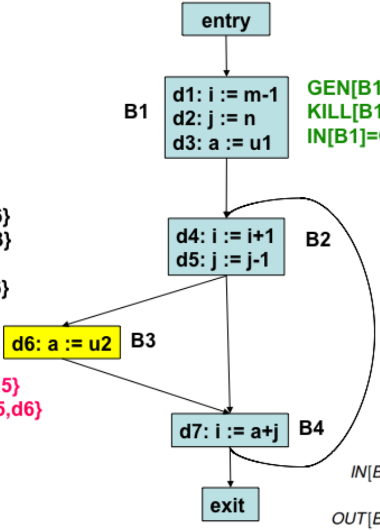
exit

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Pass 2.3

Pass 2



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]=Φ
OUT[B3]={d6}

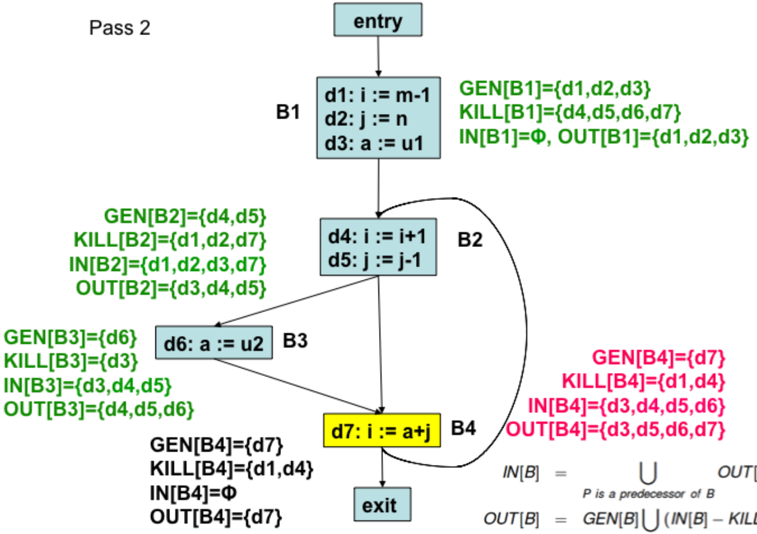
GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5}
OUT[B3]={d4,d5,d6}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Pass 2.4

Pass 2



GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d7}
OUT[B2]={d3,d4,d5}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5}
OUT[B3]={d4,d5,d6}

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]={d3,d4,d5,d6}
OUT[B4]={d3,d5,d6,d7}

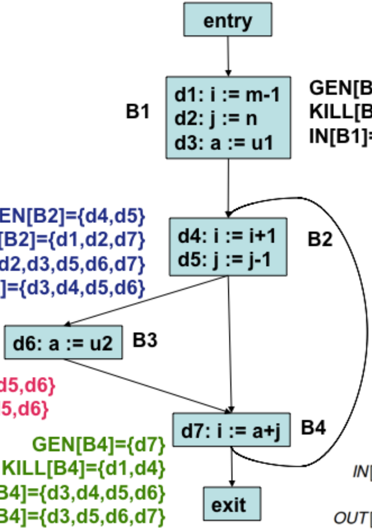
GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]=Φ
OUT[B4]={d7}

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

Reaching Definitions Analysis: An Example - Final

Final



entry

B1

d1: i := m-1
d2: j := n
d3: a := u1

GEN[B1]={d1,d2,d3}
KILL[B1]={d4,d5,d6,d7}
IN[B1]=Φ, OUT[B1]={d1,d2,d3}

B2

d4: i := i+1
d5: j := j-1

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d5,d6,d7}
OUT[B2]={d3,d4,d5,d6}

B3

d6: a := u2

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5,d6}
OUT[B3]={d4,d5,d6}

B4

d7: i := a+j

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]={d3,d4,d5,d6}
OUT[B4]={d3,d5,d6,d7}

exit

Adapted from the "Dragon Book", A-W, 1986

$$IN[B] = \bigcup_{P \text{ is a predecessor of } B} OUT[P]$$

$$OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B])$$

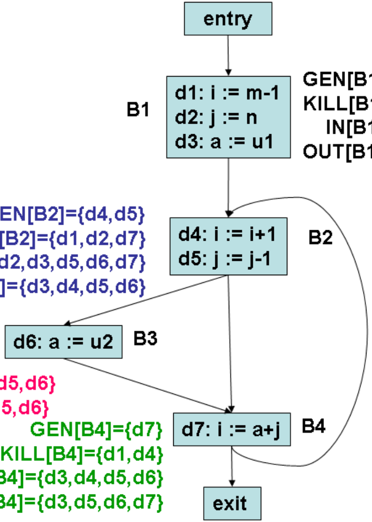
An Iterative Algorithm for Computing Reaching Def.

```
for each block $B$ do { $IN[B] = \phi;\ OUT[B] = GEN[B];$ }
$change = true$;
while $change$ do { $change = false$;
  for each block $B$ do {
      $IN[B] = \bigcup_{P\ a\ predecessor\ of\ B} OUT[P];$
      $oldout = OUT[B];$
      $OUT[B] = GEN[B] \bigcup (IN[B] - KILL[B]);$
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}
```

- *GEN*, *KILL*, *IN*, and *OUT* are all represented as bit vectors with one bit for each definition in the flow graph

Reaching Definitions: Bit Vector Representation

Final dataflow value sets shown in bit vector format



d1: i := m-1
d2: j := n
d3: a := u1

d4: i := i+1
d5: j := j-1

d6: a := u2

d7: i := a+j

	d1	d2	d3	d4	d5	d6	d7
GEN[B1]=	1	1	1	0	0	0	0
KILL[B1]=	0	0	0	1	1	1	1
IN[B1]=	0	0	0	0	0	0	0
OUT[B1]=	1	1	1	0	0	0	0

GEN[B2]={d4,d5}
KILL[B2]={d1,d2,d7}
IN[B2]={d1,d2,d3,d5,d6,d7}
OUT[B2]={d3,d4,d5,d6}

GEN[B3]={d6}
KILL[B3]={d3}
IN[B3]={d3,d4,d5,d6}
OUT[B3]={d4,d5,d6}

GEN[B4]={d7}
KILL[B4]={d1,d4}
IN[B4]={d3,d4,d5,d6}
OUT[B4]={d3,d5,d6,d7}

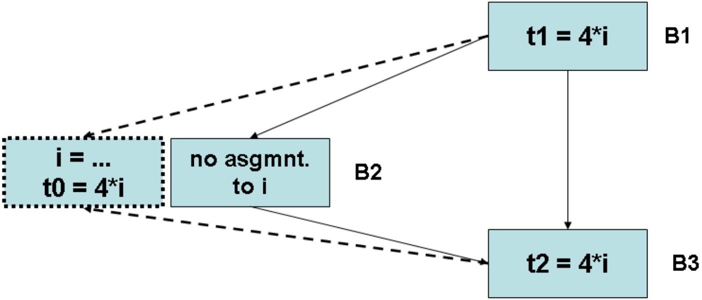
Adapted from the "Dragon Book", A-W, 1986

Available Expression Computation

- Sets of expressions constitute the domain of data-flow values
- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates $x + y$, and does not subsequently redefine x or y

Available Expression Computation(2)

- Useful for global common sub-expression elimination
- $4 * i$ is a CSE in $B3$, if it is available at the entry point of $B3$ *i.e.,* if i is not assigned a new value in $B2$ or $4 * i$ is recomputed after i is assigned a new value in $B2$ (as shown in the dotted box)



Computing e_gen and e_kill

- For statements of the form $x = a$, step 1 below does not apply
- The set of all expressions appearing as the RHS of assignments in the flow graph is assumed to be available and is represented using a hash table and a bit vector

e_gen[q] = A **q ▪**
x = y + z
p ▪

Computing e_gen[p]

1. A = A **U** {y+z}
2. A = A – {all expressions involving x}
3. e_gen[p] = A

e_kill[q] = A **q ▪**
x = y + z
p ▪

Computing e_kill[p]

1. A = A – {y+z}
2. A = A **U** {all expressions involving x}
3. e_kill[p] = A

Introduction to Machine-Independent Optimizations - 3

Data-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

Available Expression Computation

- Sets of expressions constitute the domain of data-flow values
- Forward flow problem
- Confluence operator is $\cap$
- An expression $x + y$ is *available* at a point p, if every path (not necessarily cycle-free) from the initial node to p evaluates $x + y$, and after the last such evaluation, prior to reaching p, there are no subsequent assignments to x or y
- A block *kills* $x + y$, if it assigns (or may assign) to x or y and does not subsequently recompute $x + y$.
- A block *generates* $x + y$, if it definitely evaluates $x + y$, and does not subsequently redefine x or y

Available Expression Computation - EGEN and EKILL

In other blocks:

d5: b = a+4
d6: f = e+c
d7: e = b+d
d8: d = a+b
d9: a = c+f
d10: c = e+a

d1: a = f + 1
d2: b = a + 7
d3: c = b + d
d4: a = d + c

B

Set of all expressions = {f+1,a+7,b+d,d+c,a+4,e+c,a+b,c+f,e+a}

EGEN[B] = {f+1,b+d,d+c}
EKILL[B] = {a+4,a+b,e+a,e+c,c+f,a+7}

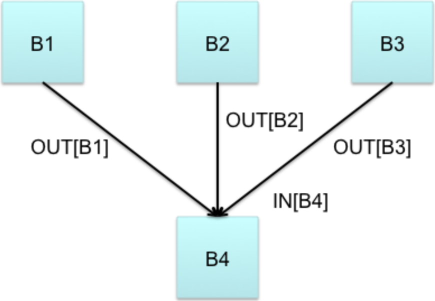
Available Expression Computation - DF Equations (1)

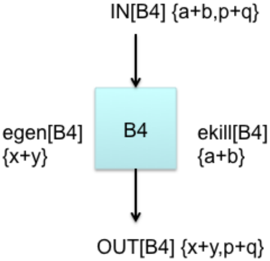
- The data-flow equations

$$\begin{aligned}
IN[B] &= \bigcap_{P\ is\ a\ predecessor\ of\ B} OUT[P],\ B\ not\ initial \\
OUT[B] &= e\_gen[B] \bigcup (IN[B] - e\_kill[B]) \\
IN[B1] &= \phi \\
IN[B] &= U,\ for\ all\ B \neq B1\ (initialization\ only)
\end{aligned}$$

- $B1$ is the intial or entry block and is special because nothing is available when the program begins execution
- $IN[B1]$ is always ϕ
- U is the universal set of all expressions
- Initializing $IN[B]$ to ϕ for all $B \neq B1$, is restrictive

Available Expression Computation - DF Equations (2)





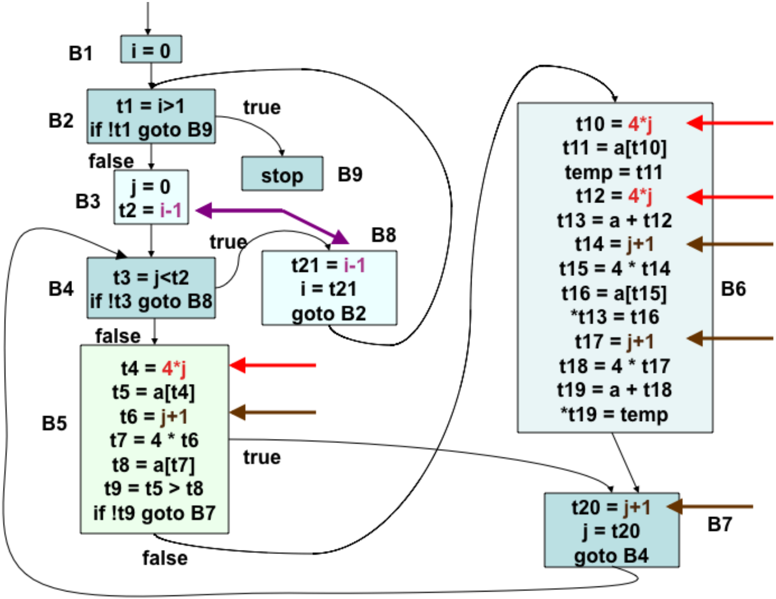
$$\mathbf{IN[B4] = OUT[B1] \bigcap OUT[B2] \bigcap OUT[B3]}$$

$$\mathbf{OUT[B4] = egen[B4] \bigcup (IN[B4] - ekill[B4])}$$

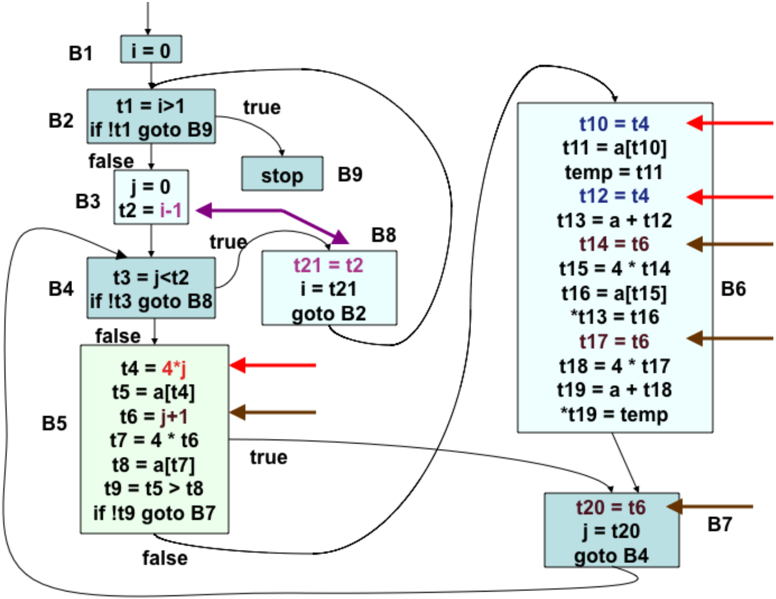
$$IN[B] = \bigcap_{P \text{ is a predecessor of } B} OUT[P], \; B \text{ not initial}$$

$$OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$$

Available Expression Computation - An Example



Available Expression Computation - An Example (2)



An Iterative Algorithm for Computing Available Expressions

```
for each block $B \neq B1$ do {$OUT[B] = U - e\_kill[B]$; }
/* You could also do $IN[B] = U$;*/
/* In such a case, you must also interchange the order of */
/* $IN[B]$ and $OUT[B]$ equations below */
$change = true$;
while $change$ do { $change = false$;
  for each block $B \neq B1$ do {
        $IN[B] = \bigcap_{P\ a\ predecessor\ of\ B} OUT[P]$;
        $oldout = OUT[B]$;
        $OUT[B] = e\_gen[B] \bigcup (IN[B] - e\_kill[B])$;
    if ($OUT[B] \neq oldout$) $change = true$;
  }
}
```

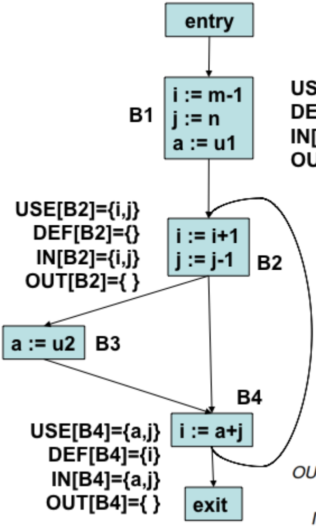
Live Variable Analysis

- The variable x is *live* at the point p, if the value of x at p could be used along some path in the flow graph, starting at p; otherwise, x is *dead* at p
- Sets of variables constitute the domain of data-flow values
- Backward flow problem, with confluence operator $\bigcup$
- $IN[B]$ is the set of variables live at the beginning of B
- $OUT[B]$ is the set of variables live just after B
- $DEF[B]$ is the set of variables definitely assigned values in B, prior to any use of that variable in B
- $USE[B]$ is the set of variables whose values may be used in B prior to any definition of the variable

$$
\begin{aligned}
OUT[B] &= \bigcup_{S\ is\ a\ successor\ of\ B} IN[S] \\
IN[B] &= USE[B] \bigcup (OUT[B] - DEF[B]) \\
IN[B] &= \phi, for\ all\ B\ (initialization\ only)
\end{aligned}
$$

Live Variable Analysis: An Example - Pass 1

Pass 1



USE[B1]={m,n,u1}
DEF[B1]={i,j,a}
IN[B1]={m,n,u1}
OUT[B1]={ }

USE[B2]={i,j}
DEF[B2]={}
IN[B2]={i,j}
OUT[B2]={ }

USE[B3]={u2}
DEF[B3]={a}
IN[B3]={u2}
OUT[B3]={ }

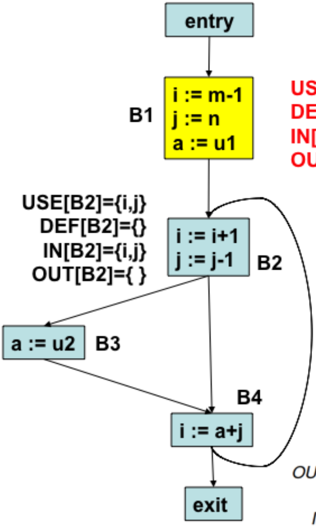
USE[B4]={a,j}
DEF[B4]={i}
IN[B4]={a,j}
OUT[B4]={ }

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis: An Example - Pass 2.1

Pass 2



USE[B1]={m,n,u1}
DEF[B1]={i,j,a}
IN[B1]={m,n,u1}
OUT[B1]={i,j}

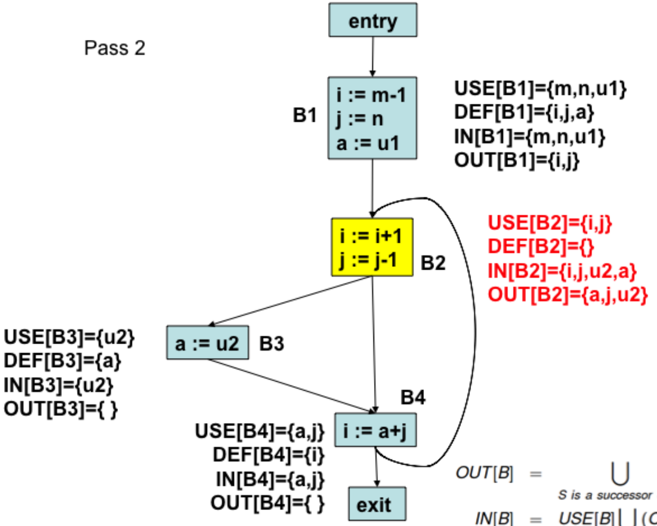
USE[B2]={i,j}
DEF[B2]={}
IN[B2]={i,j}
OUT[B2]={ }

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis: An Example - Pass 2.2

Pass 2

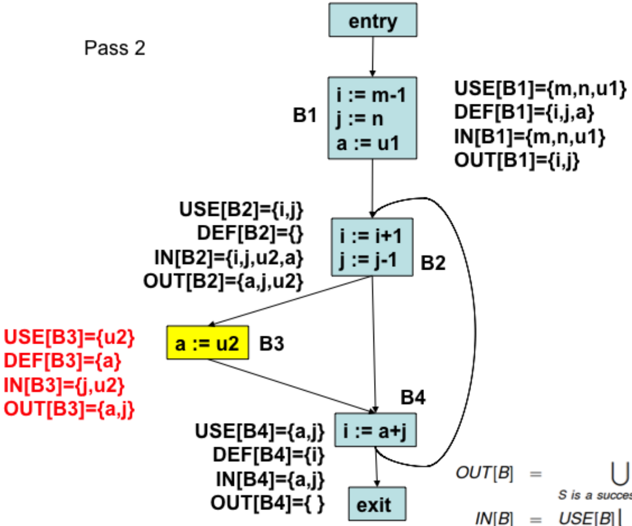


$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis: An Example - Pass 2.3

Pass 2



USE[B1]={m,n,u1}
DEF[B1]={i,j,a}
IN[B1]={m,n,u1}
OUT[B1]={i,j}

USE[B2]={i,j}
DEF[B2]={}
IN[B2]={i,j,u2,a}
OUT[B2]={a,j,u2}

USE[B3]={u2}
DEF[B3]={a}
IN[B3]={j,u2}
OUT[B3]={a,j}

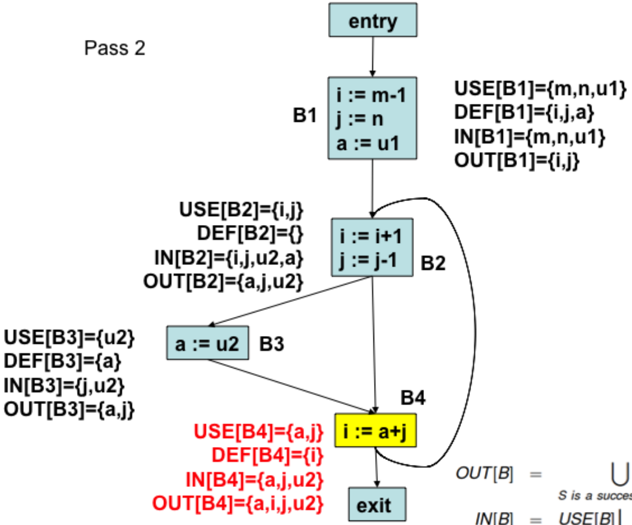
USE[B4]={a,j}
DEF[B4]={i}
IN[B4]={a,j}
OUT[B4]={ }

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis: An Example - Pass 2.4

Pass 2



entry

B1

i := m-1
j := n
a := u1

USE[B1]={m,n,u1}
DEF[B1]={i,j,a}
IN[B1]={m,n,u1}
OUT[B1]={i,j}

USE[B2]={i,j}
DEF[B2]={}
IN[B2]={i,j,u2,a}
OUT[B2]={a,j,u2}

i := i+1
j := j-1

B2

USE[B3]={u2}
DEF[B3]={a}
IN[B3]={j,u2}
OUT[B3]={a,j}

a := u2

B3

B4

USE[B4]={a,j}
DEF[B4]={i}
IN[B4]={a,j,u2}
OUT[B4]={a,i,j,u2}

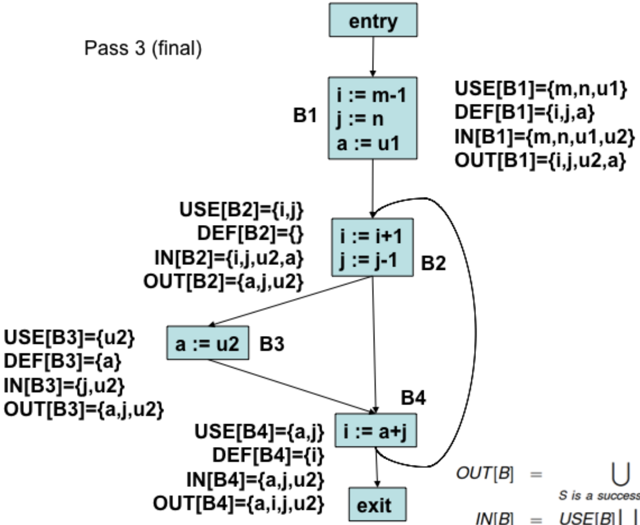
i := a+j

exit

$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Live Variable Analysis: An Example - Final pass



$$OUT[B] = \bigcup_{S \text{ is a successor of } B} IN[S]$$

$$IN[B] = USE[B] \bigcup (OUT[B] - DEF[B])$$

Data-flow Analysis: Theoretical Foundations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Foundations of Data-flow Analysis

- Basic questions to be answered
  1. In which situations is the iterative DFA algorithm correct?
  2. How precise is the solution produced by it?
  3. Will the algorithm converge?
  4. What is the meaning of a “solution”?
- The above questions can be answered accurately by a DFA framework
- Further, reusable components of the DFA algorithm can be identified once a framework is defined
- A DFA framework $(D, V, \wedge, F)$ consists of

D : A direction of the dataflow, either forward or backward
V : A domain of values
$\wedge$: A meet operator; $(V, \wedge)$ form a semi-lattice
F : A family of transfer functions, $V \longrightarrow V$
F includes constant transfer functions for the ENTRY/EXIT nodes as well

Semi-Lattice

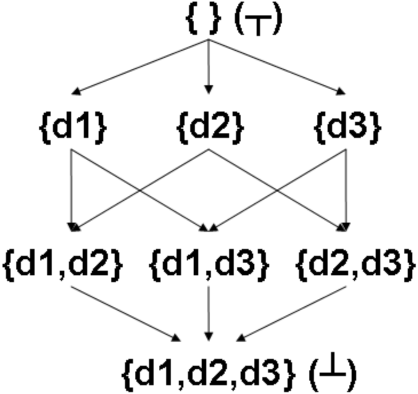
- A semi-lattice is a set V and a binary operator $\wedge$, such that the following properties hold
  1. V is closed under $\wedge$
  2. $\wedge$ is idempotent ($x \wedge x = x$), commutative ($x \wedge y = y \wedge x$), and associative ($x \wedge (y \wedge z) = (x \wedge y) \wedge z$)
  3. It has a *top* element, $\top$, such that $\forall\ x \in V,\ \top \wedge x = x$
  4. It may have a *bottom* element, $\bot$, such that $\forall x \in\ V,\ \bot \wedge x = \bot$
- The operator $\wedge$ defines a partial order $\leq$ on V, such that $x \leq y$ *iff* $x \wedge y = x$

Semi-Lattice of Reaching Definitions

- 3 definitions, {d1,d2,d3}
- V is the set of all subsets of {d1,d2,d3}
- $\wedge$ is $\cup$
- The diagram (next slide) shows the partial order relation induced by $\wedge$ (i.e., $\cup$)
- Partial order relation is $\supseteq$
- An arrow, $y \rightarrow x$ indicates $x \supseteq y$ ($x \leq y$)
- Each set in the diagram is a data-flow value
- Transitivity is implied in the diagram ($a \rightarrow b$ & $b \rightarrow c$ imples $a \rightarrow c$)
- An ascending chain: ($x_1 < x_2 < ... < x_n$)
- Height of a semi-lattice: largest number of '<' relations in any ascending chain
- Semi-lattices in our DF frameworks will be of finite height

Lattice Diagram of Reaching Definitions

$y \rightarrow x$ indicates $x \supseteq y$ ($x \leq y$)



Transfer Functions

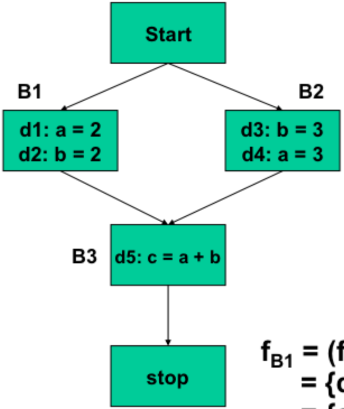
$F: \ V \rightarrow V$ has the following properties

1. F has an identity function, $I(x) = x$, for all $x \in V$
2. F is closed under composition, *i.e.,* for $f, g \in F, \ f.g \in F$

Example: Again considering the R-D problem

- Assume that each quadruple is in a separate basic block
- $OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$
- In its general form, this becomes $f(x) = G \cup (x - K)$
- F consists of such functions f, one for each basic block
- Identity function exists here (when both G and K (*GEN* and *KILL*) are empty)

Reaching Definitions Framework - Example



Transfer functions:

$f_{d1}(x) = \{d1\} \cup (x - \{d4\})$
$f_{d2}(x) = \{d2\} \cup (x - \{d3\})$
$f_{d3}(x) = \{d3\} \cup (x - \{d2\})$
$f_{d4}(x) = \{d4\} \cup (x - \{d1\})$
$f_{d5}(x) = \{d5\} \cup (x - \Phi)$

Transfer functions for start and stop blocks are identity functions

$$\begin{aligned} f_{B1} &= (f_{d2}.f_{d1})(x) \\ &= \{d2\} \cup (\{d1\} \cup (x - \{d4\}) - \{d3\}) \\ &= \{d1,d2\} \cup (x - \{d3,d4\}) \\ f_{B2} &= (f_{d4}.f_{d3})(x) \\ &= \{d3,d4\} \cup (x - \{d1,d2\}) \\ f_{B3} &= f_{d5} = \{d5\} \cup x \end{aligned}$$

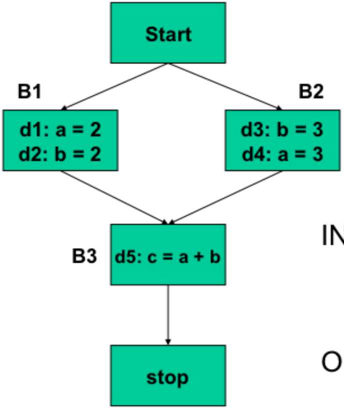
Monotone and Distributive Frameworks

- A DF framework $(D, F, V, \wedge)$ is monotone, if $\forall x, y \in V,\ f \in F,\ x \leq y \Rightarrow f(x) \leq f(y)$, OR $f(x \wedge y) \leq f(x) \wedge f(y)$
- The reaching definitions framework is monotone
- A DF framework is distributive, if $\forall x, y \in V,\ f \in F,\ f(x \wedge y) = f(x) \wedge f(y)$
- Distributivity $\Rightarrow$ monotonicity, but not vice-versa
- The reaching definitions lattice is distributive

Iterative Algorithm for DFA (forward flow)

```
{$OUT[B1] = v_{init}$;
for each block $B \neq B1$ do $OUT[B] = \top$;
while (*changes to any OUT occur*) do
  for each block $B \neq B1$ do {
      $IN[B] = \bigwedge_{P\ a\ predecessor\ of\ B} OUT[P]$;
      $OUT[B] = f_B(IN[B])$;
  }
}
```

Reaching Definitions Framework - Example contd.



f_{B1} = {d1,d2} U (x – {d3,d4})

f_{B2} = {d3,d4} U (x – {d1,d2})

f_{B3} = {d5} U x

$$IN[B] = \wedge_{P,\ \text{a predecessor of B}}\ OUT[P]$$

$$= \cup_{P,\text{a predecessor of B}}\ OUT[P]$$

$$OUT[B] = f_B(IN[B])$$

Needs 2 iterations to converge
IN[B1] = IN[B2] = Φ; OUT[B1] = {d1,d2}; OUT[B2] = {d3,d4}
IN[B3] = OUT[B1] U OUT[B2] = {d1,d2,d3,d4}
OUT[B3] = {d5} U IN[B3] = {d1,d2,d3,d4,d5}

Introduction to
Machine-Independent Optimizations - 4
Data-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

Foundations of Data-flow Analysis

- Basic questions to be answered
  1. In which situations is the iterative DFA algorithm correct?
  2. How precise is the solution produced by it?
  3. Will the algorithm converge?
  4. What is the meaning of a “solution”?
- A DFA framework $(D, V, \wedge, F)$ consists of

D	:	A direction of the dataflow, either forward or backward
V	:	A domain of values
$\wedge$	:	A meet operator; $(V, \wedge)$ form a semi-lattice
F	:	A family of transfer functions, $V \longrightarrow V$ F includes constant transfer functions for the ENTRY/EXIT nodes as well

Properties of the Iterative DFA Algorithm

1. If the iterative algorithm converges, the result is a solution to the DF equations
2. If the framework is monotone, then the solution found is the maximum fixpoint (MFP) of the DF equations
    - An MFP solution is such that in any other solution, values of $IN[B]$ and $OUT[B]$ are $\leq$ the corresponding values of the MFP (i.e., less precise)
3. If the semi-lattice of the framework is monotone and is of finite height, then the algorithm is guaranteed to converge
    - Dataflow values decrease with each iteration
      Max no. of iterations = height of the lattice $\times$ no. of nodes in the flow graph

Meaning of the Ideal Data-flow Solution

- Find all possible execution paths from the start node to the beginning of *B*
- (Assuming forward flow) Compute the data-flow value at the end of each path (using composition of transfer functions)
- No execution of the program can produce a *smaller* value for that program point than

$$IDEAL[B] = \bigwedge_{P,\ a\ possible\ execution\ path\ from\ start\ node\ to\ B} f_P(v_{init})$$

- Answers greater (in the sense of $\leq$) than IDEAL are incorrect (one or more execution paths have been ignored)
- Any value smaller than or equal to IDEAL is conservative, *i.e.,* safe (one or more infeasible paths have been included)
- Closer the value to IDEAL, more precise it is

Meaning of the Meet-Over-Paths Data-flow Solution

- Since finding all execution paths is an undecidable problem, we approximate this set to include all paths in the flow graph

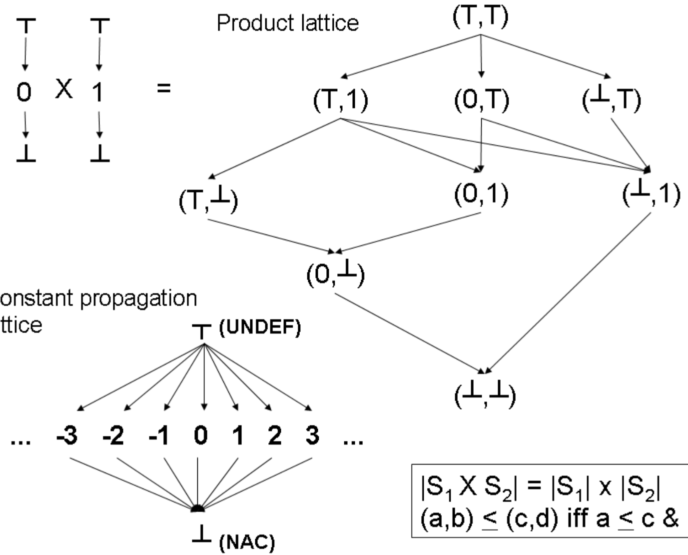
$$MOP[B] = \bigwedge_{P,\ a\ path\ from\ start\ node\ to\ B} f_P(v_{init})$$

- $MOP[B] \leq IDEAL[B]$, since we consider a superset of the set of execution paths

Meaning of the Maximum Fixpoint Data-flow Solution

- Finding all paths in a flow graph may still be impossible, if it has cycles
- The iterative algorithm does not try this
    - It visits all basic blocks, not necessarily in execution order
    - It applies the $\wedge$ operator at each join point in the flow graph
    - The solution obtained is the Maximum Fixpoint solution (MFP)
- If the framework is distributive, then the MOP and MFP solutions will be identical
- Otherwise, with just monotonicity, $MFP \leq MOP \leq IDEAL$, and the solution provided by the iterative algorithm is safe

Product of Two Lattices and Lattice of Constants



$|S_1 \text{ X } S_2| = |S_1| \text{ x } |S_2|$
$(a,b) \leq (c,d)$ iff $a \leq c$ & $b \leq d$

The Constant Propagation Framework

- The lattice of the DF values in the CP framework is the product of the semi-lattices of the variables (one lattice for each variable)
- In a product lattice, $(a_1, b_1) \leq (a_2, b_2)$ iff $a_1 \leq_A a_2$ and $b_1 \leq_B b_2$ assuming $a_1, a_2 \in A$ and $b_1, b_2 \in B$
- Each variable v is associated with a map m, and $m(v)$ is its abstract value (as in the lattice)
- Each element of the product lattice has a similar, but “larger” map m
    - Thus, $m \leq m'$ (in the product lattice), iff for all variables v, $m(v) \leq m'(v)$

Transfer Functions for the CP Framework

- Assume one statement per basic block
- Transfer functions for basic blocks containing many statements may be obtained by composition
- $m(v)$ is the abstract value of the variable v in a map m.
- The set F of the framework contains transfer functions which accept maps and produce maps as outputs
- F contains an identity map
- Map for the *Start* block is $m_0(v) = UNDEF$, for all variables v
- This is reasonable since all variables are undefined before a program begins

Transfer Functions for the CP Framework

- Let f_s be the transfer function of the statement s
- If $m' = f_s(m)$, then f_s is defined as follows
  1. If s is not an assignment, f_s is the identity function
  2. If s is an assignment to a variable x, then $m'(v) = m(v)$, for all $v \neq x$, and,
     - (a) If the RHS of s is a constant c, then $m'(x) = c$
     - (b) If the RHS is of the form $y + z$, then

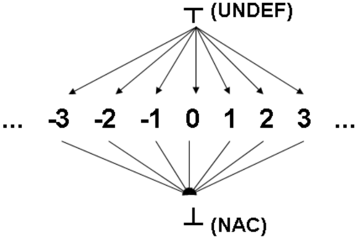
$$\begin{aligned} m'(x) &= m(y) + m(z),\ \textit{if } m(y) \textit{ and } m(z) \textit{ are constants} \\ &= NAC,\ \textit{if either } m(y) \textit{ or } m(z) \textit{ is } NAC \\ &= UNDEF,\ \textit{otherwise} \end{aligned}$$

     - (c) If the RHS is any other expression, then $m'(x) = NAC$

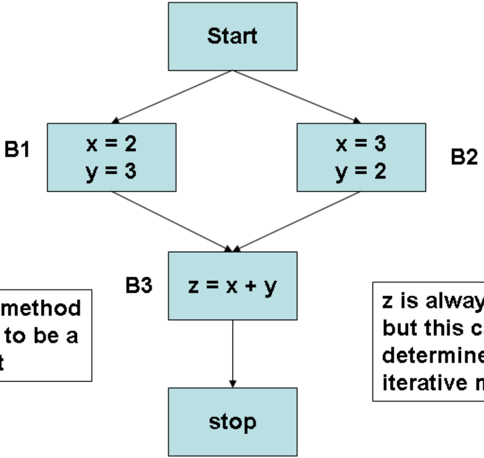
Monotonicity of the CP Framework

It must be noted that the transfer function ($m' = f_s(m)$) always produces a "lower" or same level value in the CP lattice, whenever there is a change in inputs

$m(y)$	$m(z)$	$m'(x)$
UNDEF	UNDEF	UNDEF
	c_2	UNDEF
	NAC	NAC
c_1	UNDEF	UNDEF
	c_2	$c_1 + c_2$
	NAC	NAC
NAC	UNDEF	NAC
	c_2	NAC
	NAC	NAC



Non-distributivity of the CP Framework



The iterative method determines z to be a non-constant

z is always a constant but this cannot be determined by the iterative method

Non-distributivity of the CF Framework - Example

- If f_1, f_2, f_3 are transfer functions of $B1, B2, B3$ (resp.), then $f_3(f_1(m_0) \wedge f_2(m_0)) < f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$ as shown in the table, and therefore the CF framework is non-distributive

m	$m(x)$	$m(y)$	$m(z)$
m_0	UNDEF	UNDEF	UNDEF
$f_1(m_0)$	2	3	UNDEF
$f_2(m_0)$	3	2	UNDEF
$f_1(m_0) \wedge f_2(m_0)$	NAC	NAC	UNDEF
$f_3(f_1(m_0) \wedge f_2(m_0))$	NAC	NAC	NAC
$f_3(f_1(m_0))$	2	3	5
$f_3(f_2(m_0))$	3	2	5
$f_3(f_1(m_0)) \wedge f_3(f_2(m_0))$	NAC	NAC	5

Introduction to Control-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Why control-flow analysis?
- Dominators and natural loops
- Depth of a control-flow graph

Why Control-Flow Analysis?

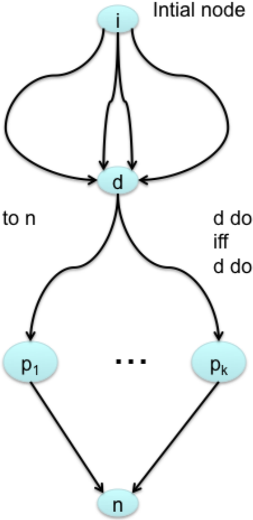
Control-flow analysis (CFA) helps us to understand the structure of control-flow graphs (CFG)

- To determine the loop structure of CFGs
- To compute dominators - useful for code motion
- To compute dominance frontiers - useful for the construction of the static single assignment form (SSA)
- To compute control dependence - needed in parallelization

Dominators

- We say that a node d in a flow graph *dominates* node n, written d *dom* n, if every path from the initial node of the flow graph to n goes through d
- Initial node is the root, and each node dominates only its descendents in the dominator tree (including itself)
- The node x *strictly dominates* y, if x dominates y and $x \neq y$
- x is the *immediate dominator* of y (denoted $idom(y)$), if x is the closest strict dominator of y
- A *dominator tree* shows all the immediate dominator relationships
- Principle of the dominator algorithm
  - If $p_1, p_2, ..., p_k$, are all the predecessors of n, and $d \neq n$, then d *dom* n, iff d *dom* p_i for each i

Dominator Algorithm Principle



All paths from i to n
go through d

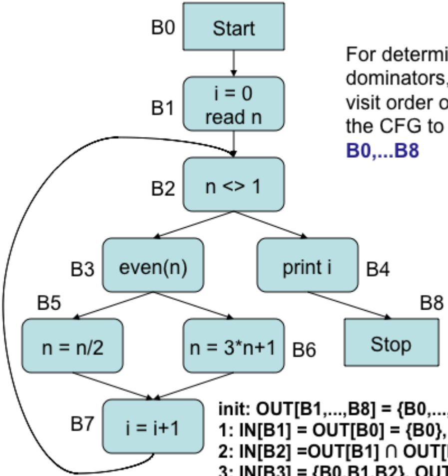
d dom n
iff
d dom p_i for all i

An Algorithm for finding Dominators

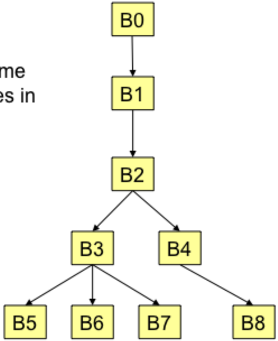
- $D(n) = OUT[n]$ for all n in N (the set of nodes in the flow graph), after the following algorithm terminates
- { /* n_0 = initial node; N = set of all nodes; */

```
{ /* $n_0$ = initial node; $N$ = set of all nodes; */
  $OUT[n_0] = \{n_0\};$
  for $n$ in $N - \{n_0\}$ do $OUT[n] = N;$
  while (changes to any $OUT[n]$ or $IN[n]$ occur) do
    for $n$ in $N - \{n_0\}$ do
      $IN[n] = \bigcap_{P\ a\ predecessor\ of\ n} OUT[P];$
      $OUT[n] = \{n\} \cup IN[n]$
}
```

Dominator Example - 1

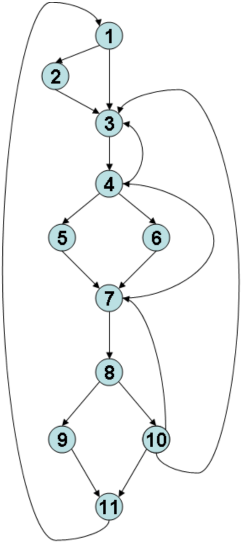


For determining dominators, assume visit order of nodes in the CFG to be **B0,...B8**

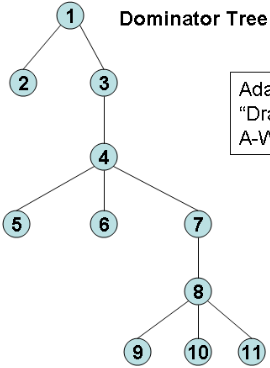


init: OUT[B1,...,B8] = {B0,...,B8}, OUT[B0] = {B0}
1: IN[B1] = OUT[B0] = {B0}, OUT[B1] = {B0,B1}
2: IN[B2] =OUT[B1] ∩ OUT[B7] = {B0,B1} , OUT[B2] = {B0,B1,B2}
3: IN[B3] = {B0,B1,B2}, OUT[B3] = {B0,B1,B2,B3}
IN[B4] = {B0,B1,B2}, OUT[B4] = {B0,B1,B2,B4} = IN[B8]
4: IN[B5] = {B0,B1,B2,B3} = IN[B6], OUT[B5] = {B0,B1,B2,B3,B5}
OUT[B6] = {B0,B1,B2,B3,B6}, OUT[B8] = {B0,B1,B2,B4,B8}
5: IN[B7] = OUT[B5] ∩ OUT[B6] = {B0,B1,B2,B3}
OUT[B7] = {B0,B1,B2,B3,B7}

Dominator Example - 2



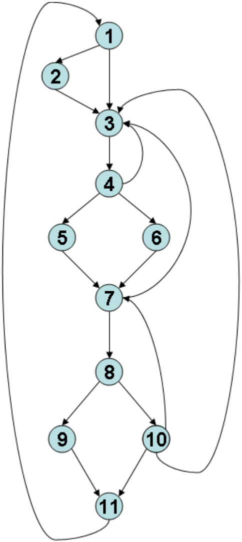
Flow Graph



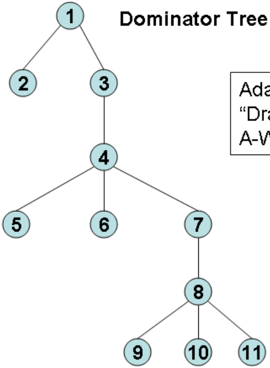
Dominator Tree

Adapted from the "Dragon Book", A-W, 1986

Dominator Example - 3



Flow Graph



Dominator Tree

Adapted from the
“Dragon Book”,
A-W, 1986

Dominators and Natural Loops

- Edges whose heads dominate their tails are called *back edges* ($a \rightarrow b$: $b = head$, $a = tail$)
- Given a back edge $n \rightarrow d$
    - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
    - d is the header of the loop
        - A single entry point to the loop that dominates all nodes in the loop
        - At least one path back to the header exists (so that the loop can be iterated)

Introduction to Machine-Independent Optimizations - 5

Control-Flow Analysis

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis
- Algorithms for two machine-independent optimizations
- SSA form and optimizations

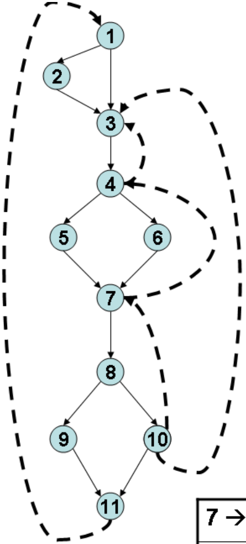
Dominators and Natural Loops

- Edges whose heads dominate their tails are called *back edges* ($a \rightarrow b: \; b = head, \; a = tail$)
- Given a back edge $n \rightarrow d$
  - The *natural loop* of the edge is d plus the set of nodes that can reach n without going through d
  - d is the header of the loop
    - A single entry point to the loop that dominates all nodes in the loop
    - At least one path back to the header exists (so that the loop can be iterated)

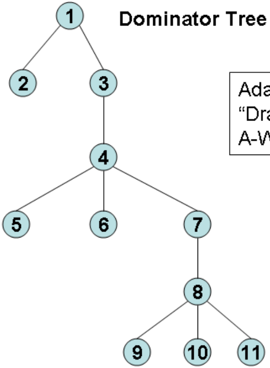
Algorithm for finding the Natural Loop of a Back Edge

```
/* The back edge under consideration is $n \rightarrow d$ /*
{ stack = empty; loop = {$d$};
 /* This ensures that we do not look at predecessors of $d$ */
  insert($n$);
  while (stack is not empty) do {
    pop($m$, stack);
    for each predecessor $p$ of $m$ do insert($p$);
  }
}

  procedure insert($m$) {
    if $m \notin$ loop then {
      loop = loop $\cup$ {$m$};
      push($m$, stack);
    }
  }
```

Dominators, Back Edges, and Natural Loops



Flow Graph



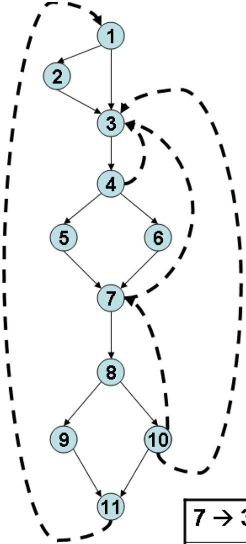
Dominator Tree

Adapted from the
“Dragon Book”,
A-W, 1986

Back edges and their natural loops

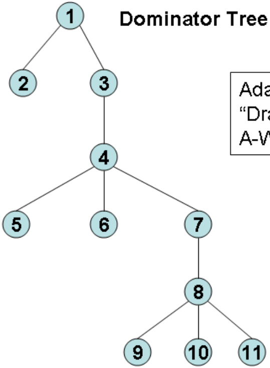
7 → 4	10→7	4→3	10→3	11→1
{4,5,6,7,8,10}	{7,8,10}	{3,4,5,6,7,8,10}	{3,4,5,6,7,8,10}	{1,2,3,4,5,6,7,8,9,10,11}

Dominators, Back Edges, and Natural Loops



Flow Graph

Dominator Tree



Adapted from the
“Dragon Book”,
A-W, 1986

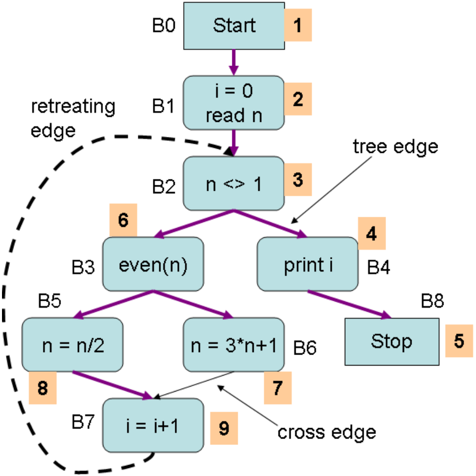
Back edges and their natural loops

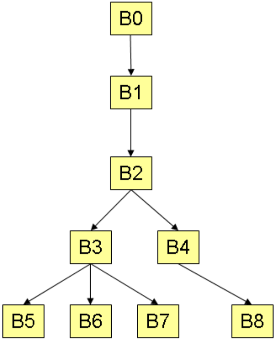
7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6,7,8, 10}	{7,8,10}	{3,4}	{3,4,5,6,7, 8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Depth-First Numbering of Nodes in a CFG

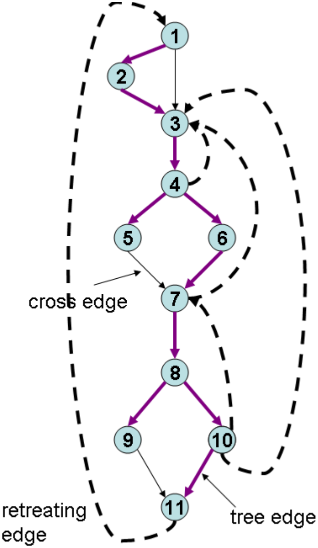
```
void dfs-num(int n) {
    mark node n "visited";
    for each node s adjacent to n do {
        if s is "unvisited" {
            add edge n →s to dfs tree T;
            dfs-num(s);
        }
    depth-first-num[n] = i ; i-- ;
}
// Main program
{  T = empty; mark all nodes of CFG as "unvisited";
   i = number of nodes of CFG;
   dfs-num($n_0$);// $n_0$ is the entry node of the CFG
}
```

Depth-First Numbering Example 1

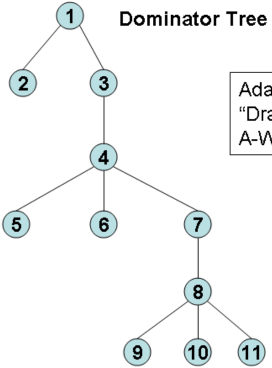




Depth-First Numbering Example 2



Flow Graph



Dominator Tree

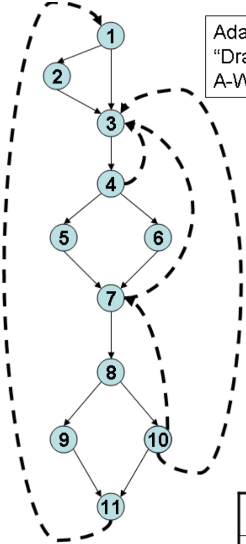
Adapted from the "Dragon Book", A-W, 1986

Nodes of the CFG show the DF-numbering

Inner Loops

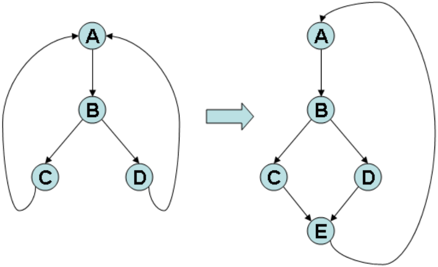
- Unless two loops have the same header, they are either disjoint or one is nested within the other
- Nesting is checked by testing whether the set of nodes of a loop A is a subset of the set of nodes of another loop B
- Similarly, two loops are disjoint if their sets of nodes are disjoint
- When two loops share a header, neither of these may hold (see next slide)
- In such a case the two loops are combined and transformed as in the next slide

Inner Loops and Loops with the same header



Adapted from the
“Dragon Book”,
A-W, 1986

C→A	D→A	E→A
{A,B,C}	{A,B,D}	{A,B,C, D,E}

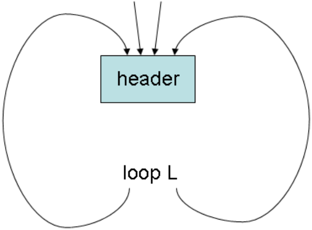


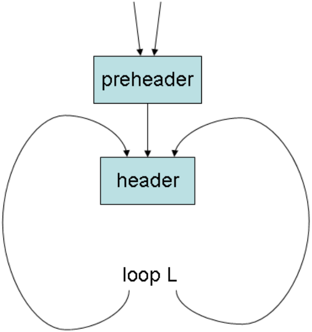
E is a dummy node

Back edges and their natural loops

7 → 3	10→7	4→3	10→3	11→1
{3,4,5,6, 7,8,10}	{7,8,10}	{3,4}	{3,4,5,6, 7,8,10}	{1,2,3,4,5, 6,7,8,9,10,11}

Preheader

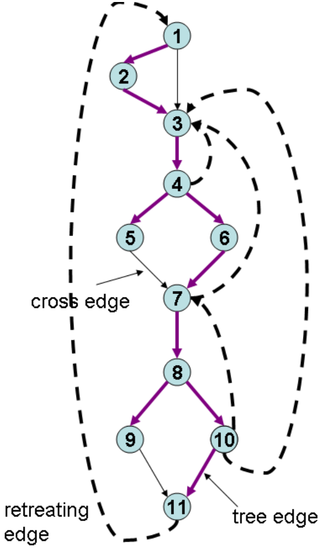




Depth of a Flow Graph and Convergence of DFA

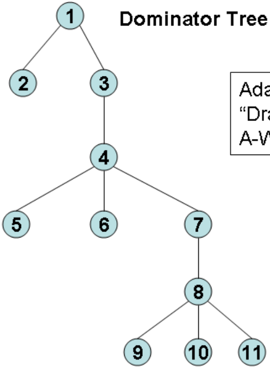
- Given a depth-first spanning tree of a CFG, the largest number of retreating edges on any cycle-free path is the *depth* of the CFG
- The number of passes needed for convergence of the solution to a forward DFA problem is (1 + depth of CFG)
- One more pass is needed to determine *no change*, and hence the bound is actually (2 + depth of CFG)
- This bound can be actually met if we traverse the CFG using the *depth-first numbering* of the nodes
- For a backward DFA, the same bound holds, but we must consider the reverse of the depth-first numbering of nodes
- Any other order will still produce the correct solution, but the number of passes may be more

Depth of a CFG - Example 1



Flow Graph

Dominator Tree

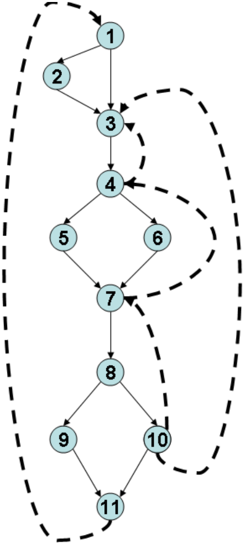


Adapted from the
“Dragon Book”,
A-W, 1986

Nodes of the CFG show the DF-numbering

Depth of the CFG = 2 (10-7-3)

Depth of a CFG - Example 2



Flow Graph

Adapted from the “Dragon Book”, A-W, 1986

Depth of the CFG = 3 (10-7-4-3)

Algorithms for Machine-Independent Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

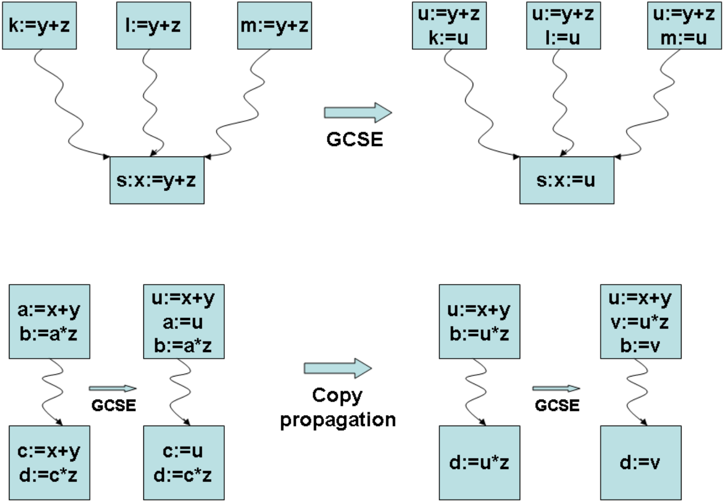
Outline of the Lecture

- Global common sub-expression elimination
- Copy propagation
- Simple constant propagation
- Loop invariant code motion

Elimination of Global Common Sub-expressions

- Needs available expression information
- For every $s: \; x := y + z$, such that $y + z$ is available at the beginning of s' block, and neither y nor z is defined prior to s in that block, do the following
    1. Search backwards from s' block in the flow graph, and find first block in which $y + z$ is evaluated. We need not go *through* any block that evaluates $y + z$.
    2. Create a new variable u and replace each statement $w := y + z$ found in the above step by the code segment $\{u := y + z; w := u\}$, and replace s by $x := u$
    3. Repeat 1 and 2 above for every predecessor block of s' block
- Repeated application of GCSE may be needed to catch "deep" CSE

GCSE Conceptual Example



k:=y+z

l:=y+z

m:=y+z

s:x:=y+z

GCSE

u:=y+z
k:=u

u:=y+z
l:=u

u:=y+z
m:=u

s:x:=u

a:=x+y
b:=a*z

c:=x+y
d:=c*z

GCSE

u:=x+y
a:=u
b:=a*z

c:=u
d:=c*z

Copy propagation

u:=x+y
b:=u*z

d:=u*z

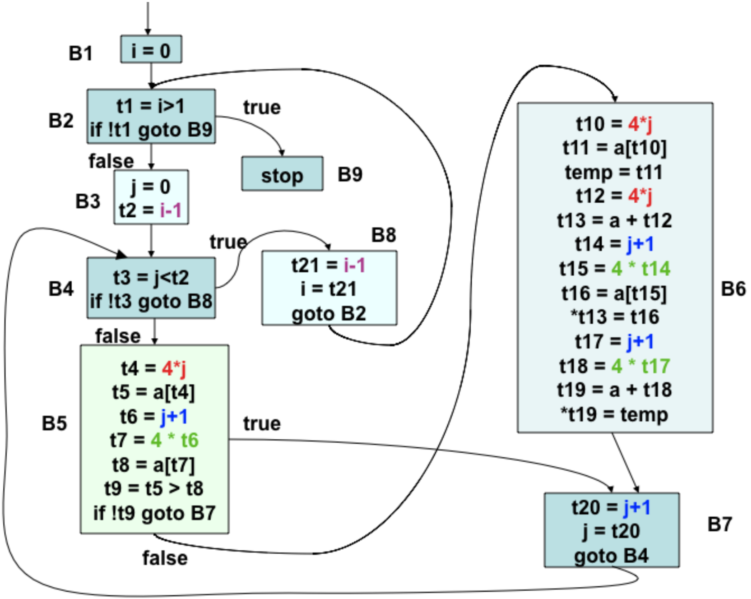
GCSE

u:=x+y
v:=u*z
b:=v

d:=v

Demonstrating the need for repeated application of GCSE

GCSE on Running Example - 1



B1

i = 0

B2

t1 = i>1
if !t1 goto B9

true

stop

B9

false

B3

j = 0
t2 = i-1

true

B8

t21 = i-1
i = t21
goto B2

B4

t3 = j<t2
if !t3 goto B8

false

B5

t4 = 4*j
t5 = a[t4]
t6 = j+1
t7 = 4 * t6
t8 = a[t7]
t9 = t5 > t8
if !t9 goto B7

true

false

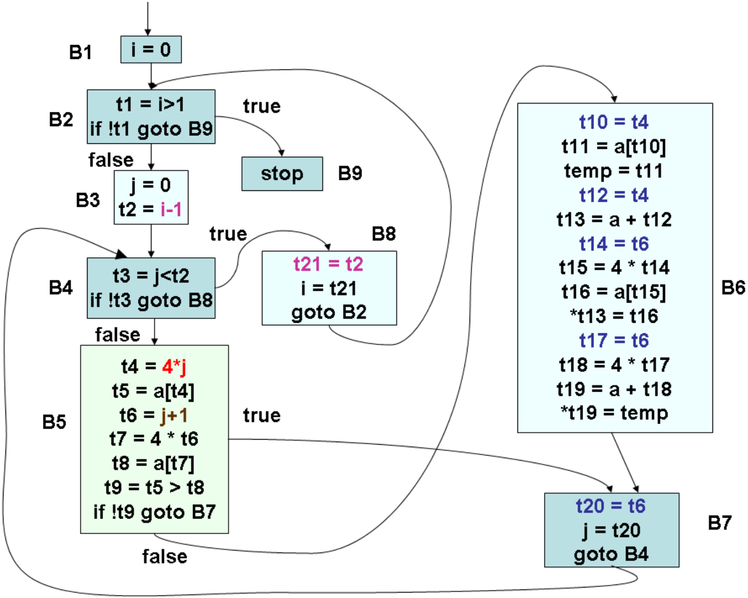
B6

t10 = 4*j
t11 = a[t10]
temp = t11
t12 = 4*j
t13 = a + t12
t14 = j+1
t15 = 4 * t14
t16 = a[t15]
*t13 = t16
t17 = j+1
t18 = 4 * t17
t19 = a + t18
*t19 = temp

B7

t20 = j+1
j = t20
goto B4

GCSE on Running Example - 2



Copy Propagation

- Eliminate copy statements of the form $s: \ x := y$, by substituting y for x in all uses of x reached by this copy
- Conditions to be checked
    1. u-d chain of use u of x must consist of s only. Then, s is the only definition of x reaching u
    2. On every path from s to u, including paths that go through u several times (but do not go through s a second time), there are no assignments to y. This ensures that the copy is valid
- The second condition above is checked by using information obtained by a new data-flow analysis problem
    - *c_gen*[B] is the set of all copy statements, $s : x := y$ in B, such that there are no subsequent assignments to either x or y within B, after s
    - *c_kill*[B] is the set of all copy statements, $s : x := y$, s not in B, such that either x or y is assigned a value in B
    - Let U be the universal set of all copy statements in the program

Copy Propagation - The Data-flow Equations

- $c\_in[B]$ is the set of all copy statements, $x := y$ reaching the beginning of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path
- $c\_out[B]$ is the set of all copy statements, $x := y$ reaching the end of B along every path such that there are no assignments to either x or y following the last occurrence of $x := y$ on the path

$$\begin{aligned}
c\_in[B] &= \bigcap_{P\ is\ a\ predecessor\ of\ B} c\_out[P],\ B\ not\ initial \\
c\_out[B] &= c\_gen[B] \bigcup (c\_in[B] - c\_kill[B]) \\
c\_in[B1] &= \phi,\ where\ B1\ is\ the\ initial\ block \\
c\_out[B] &= U - c\_kill[B],\ for\ all\ B \neq B1\ (initialization\ only)
\end{aligned}$$

Algorithm for Copy Propagation

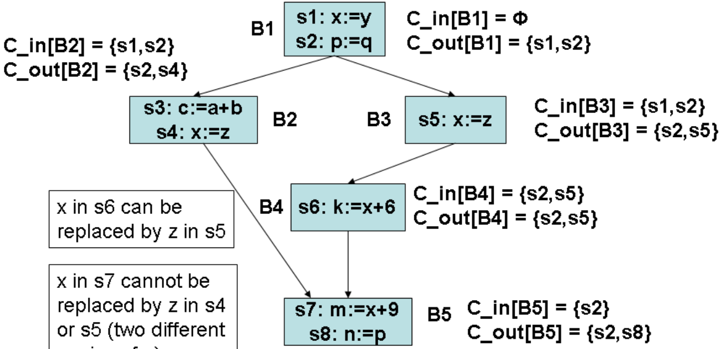
For each copy, $s : x := y$, do the following

1. Using the $du - chain$, determine those uses of x that are reached by s
2. For each use u of x found in (1) above, check that
   (i) u-d chain of u consists of s only
      - This implies that s is the only definition of x that reaches this block

   (ii) s is in $c\_in[B]$, where B is the block to which u belongs.
      - This ensures that no definitions of x or y appear on this path from s to B

   (iii) no definitions x or y occur within B prior to u found in (1) above
3. If s meets the conditions above, then remove s and replace all uses of x found in (1) above by y

Copy Propagation Example 1



B1

s1: x:=y
s2: p:=q

C_in[B1] = Φ
C_out[B1] = {s1,s2}

C_in[B2] = {s1,s2}
C_out[B2] = {s2,s4}

s3: c:=a+b
s4: x:=z

B2

B3

s5: x:=z

C_in[B3] = {s1,s2}
C_out[B3] = {s2,s5}

B4

s6: k:=x+6

C_in[B4] = {s2,s5}
C_out[B4] = {s2,s5}

s7: m:=x+9
s8: n:=p

B5

C_in[B5] = {s2}
C_out[B5] = {s2,s8}

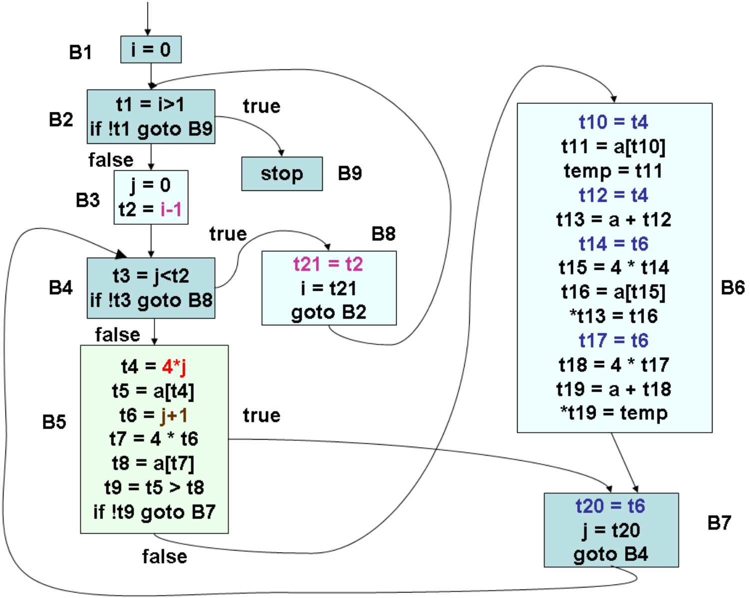
x in s6 can be replaced by z in s5

x in s7 cannot be replaced by z in s4 or s5 (two different copies of z)

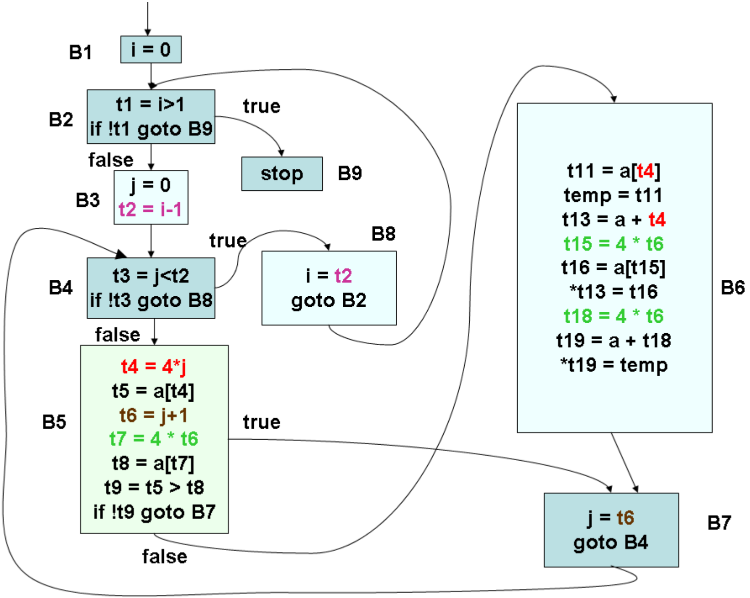
p in s8 can be replaced by q in s2 (s2 reaches B5 thro' both the paths)

Adapted from
“The Dragon Book”
A-W 1986

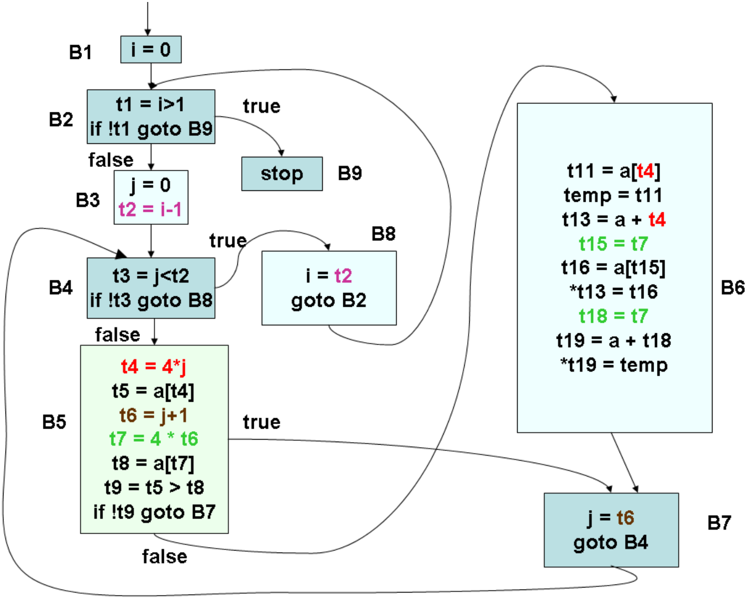
Copy Propagation on Running Example 1.1



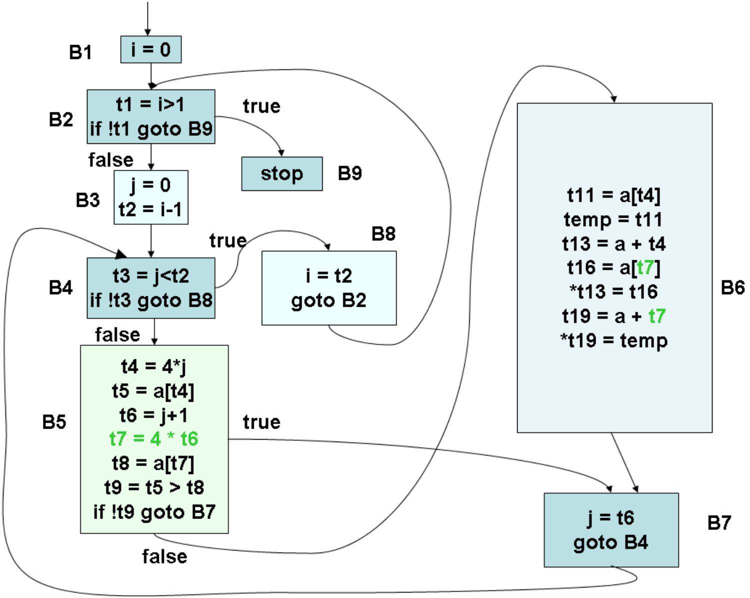
Copy Propagation on Running Example 1.2


B1
i = 0
B2
t1 = i>1
if !t1 goto B9
true
false
stop
B9
B3
j = 0
t2 = i-1
true
B8
i = t2
goto B2
B4
t3 = j<t2
if !t3 goto B8
false
B5
t4 = 4*j
t5 = a[t4]
t6 = j+1
t7 = 4 * t6
t8 = a[t7]
t9 = t5 > t8
if !t9 goto B7
true
false
B6
t11 = a[t4]
temp = t11
t13 = a + t4
t15 = 4 * t6
t16 = a[t15]
*t13 = t16
t18 = 4 * t6
t19 = a + t18
*t19 = temp
B7
j = t6
goto B4

GCSE and Copy Propagation on Running Example 1.1



GCSE and Copy Propagation on Running Example 1.2

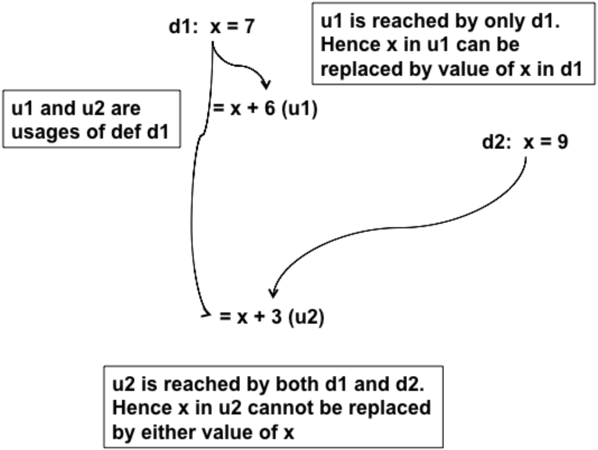


Simple Constant Propagation

```
{  Stmtpile = {S|S is a statement in the program}
   while Stmtpile is not empty {
      S = remove(Stmtpile);
      if S is of the form $x = c$ for some constant $c$
         for all statements T in the du-chain of $x$ do
               if usage of $x$ in T is reachable only by S
                  { substitute $c$ for $x$ in T; simplify T
                    Stmtpile = Stmtpile $\cup$ {T}
                  }
}
```

Note: If all usages of x are replaced by c, then $x = c$ becomes dead code and a separate dead code elimination pass will remove it.

Simple Constant Propagation Example



d1: x = 7

u1 is reached by only d1. Hence x in u1 can be replaced by value of x in d1

u1 and u2 are usages of def d1

= x + 6 (u1)

d2: x = 9

= x + 3 (u2)

u2 is reached by both d1 and d2. Hence x in u2 cannot be replaced by either value of x

Introduction to Machine-Independent Optimizations - 6

Machine-Independent Optimization Algorithms

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis (in parts 4 and 5)
- Algorithms for machine-independent optimizations
- SSA form and optimizations

Detection of Loop-invariant Computations

Mark as “invariant”, those statements whose operands are all either constant or have all their reaching definitions outside *L*

```
Repeat {
  Mark as “invariant” all those statements not previously so marked all of whose operands are constants, or have all their reaching definitions outside $L$, or have exactly one reaching definition, and that definition is a statement in $L$ marked “invariant”
} until no new statements are marked “invariant”
```

Loop Invariant Code motion Example

```
     t1 = 202
     i = 1
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t3 = addr(a)
     t4 = t3 - 4
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

Before LIV code motion

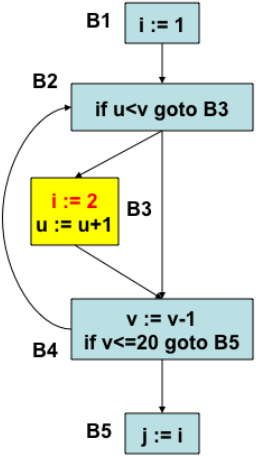
```
     t1 = 202
     i = 1
     t3 = addr(a)
     t4 = t3 - 4
L1:  t2 = i>100
     if t2 goto L2
     t1 = t1-2
     t5 = 4*i
     t6 = t4+t5
     *t6 = t1
     i = i+1
     goto L1
L2:
```

After LIV code motion

Loop-Invariant Code Motion Algorithm

1. Find loop-invariant statements
2. For each statement *s* defining *x* found in step (1), check that
   - (a) it is in a block that dominates all exits of *L*
   - (b) *x* is not defined elsewhere in *L*
   - (c) all uses in *L* of *x* can only be reached by the definition of *x* in *s*
3. Move each statement *s* found in step (1) and satisfying conditions of step (2) to a newly created preheader
   - provided any operands of *s* that are defined in loop *L* have previously had their definition statements moved to the preheader

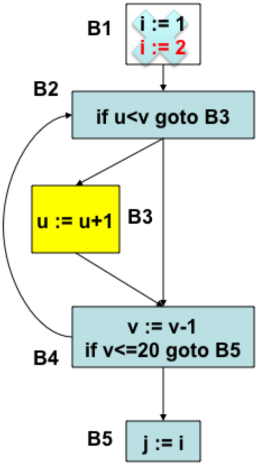
Code Motion - Violation of condition 2(a)-1



The statement i:=2 from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4)
The computation gets altered due to code movement
i always gets value 2, and never 1, and hence j always gets value 2

Condition 2(a):
s dominates all exits of *L*

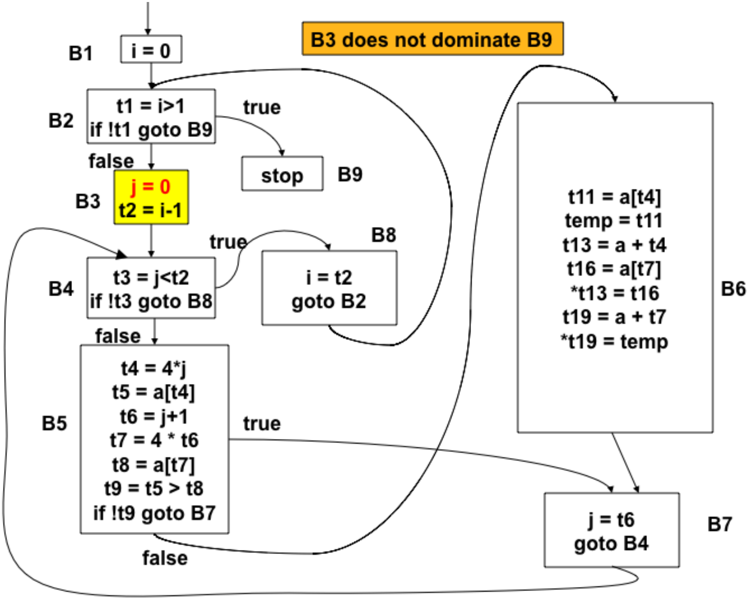
Code Motion - Violation of condition 2(a)-2



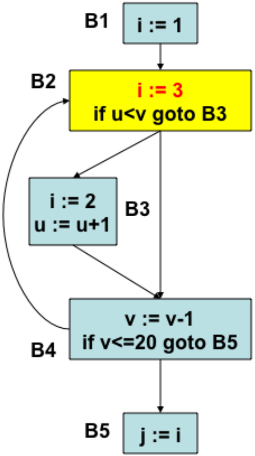
The statement i:=2 from B3 cannot be moved to a preheader since condition 2(a) is violated (B3 does not dominate B4)
The computation gets altered due to code movement
i always gets value 2, and never 1, and hence j always gets value 2

Condition 2(a):
s dominates all exits of *L*

Violation of condition 2(a) - Running Example



Code Motion - Violation of condition 2(b)



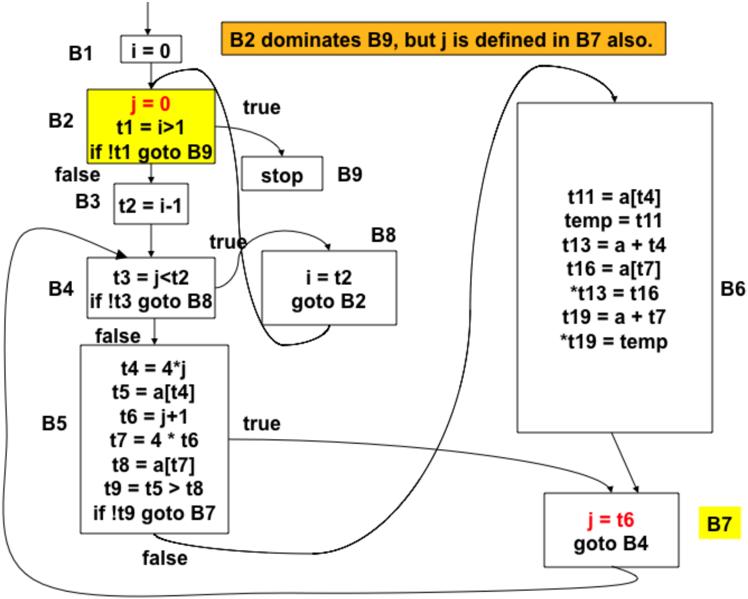
B2 dominates B4 and hence condition 2(a) is satisfied for i:=3 in B2. However statement i:=3 from B2 cannot be moved to a preheader since condition 2(b) is violated (i is defined in B3)

The computation gets altered due to code movement
If the loop is executed twice, i may pass its value of 3 from B2 to j in the original loop.
In the revised loop, i gets the value 2 in the second iteration and retains it forever

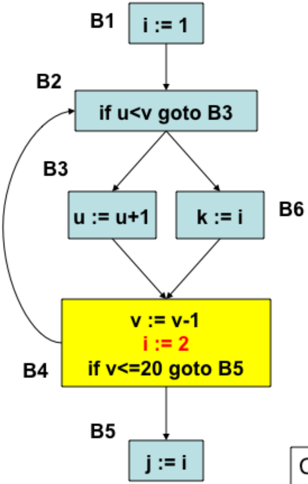
Condition 2(a):
s dominates all exits of *L*

Condition 2(b):
x is not defined elsewhere in *L*

Violation of condition 2(b) - Running Example



Code Motion - Violation of condition 2(c)



Conditions 2(a) and 2(b) are satisfied. However statement i:=2 from B4 cannot be moved to a preheader since condition 2(c) is violated (use of i in B6 is reached by defs of i in B1 and B4)

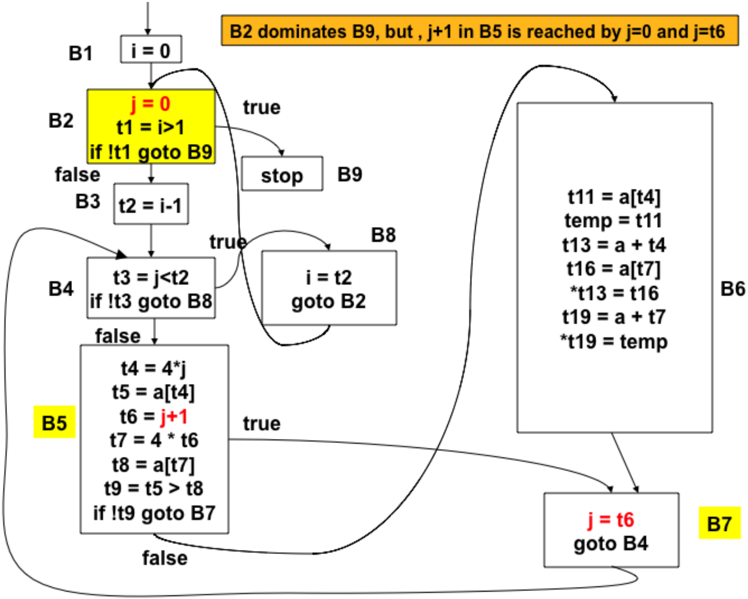
The computation gets altered due to code movement

In the revised loop, i gets the value 2 from the def in the preheader and k becomes 2.

However, k could have received the value of either 1 (from B1) or 2 (from B4) in the original loop

Condition 2(a): *s* dominates all exits of *L*
Condition 2(b): *x* is not defined elsewhere in *L*
Condition 2(c): All uses of *x* in *L* can only be reached by the definition of *x* in *s*

Violation of condition 2(c) - Running Example



B2 dominates B9, but , j+1 in B5 is reached by j=0 and j=t6

The Static Single Assignment Form:

Application to Program Optimizations

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

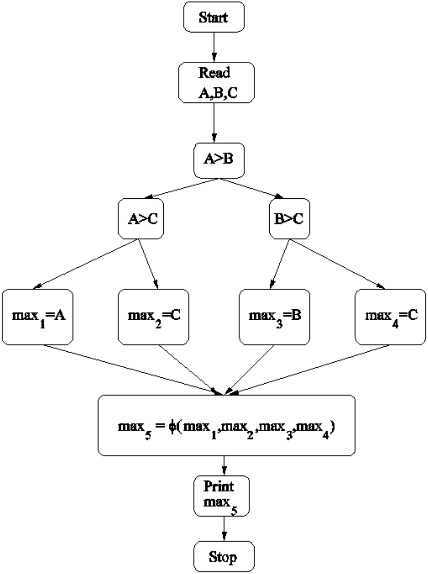
- SSA form - definition and examples
- Optimizations with SSA forms
    - Dead-code elimination
    - Simple constant propagation
    - Copy propagation
    - Conditional constant propagation and constant folding

The SSA Form: Introduction

- A new intermediate representation
- Incorporates *def-use* information
- Every variable has exactly one definition in the program text
  - This does not mean that there are no loops
  - This is a *static* single assignment form, and not a *dynamic* single assignment form
- Some compiler optimizations perform better on SSA forms
  - Conditional constant propagation and global value numbering are faster and more effective on SSA forms
- A *sparse* intermediate representation
  - If a variable has N uses and M definitions, then *def-use chains* need space and time proportional to $N.M$
  - But, the corresponding instructions of uses and definitions are only $N + M$ in number
  - SSA form, for most realistic programs, is linear in the size of the original program

A Program in non-SSA Form and its SSA Form

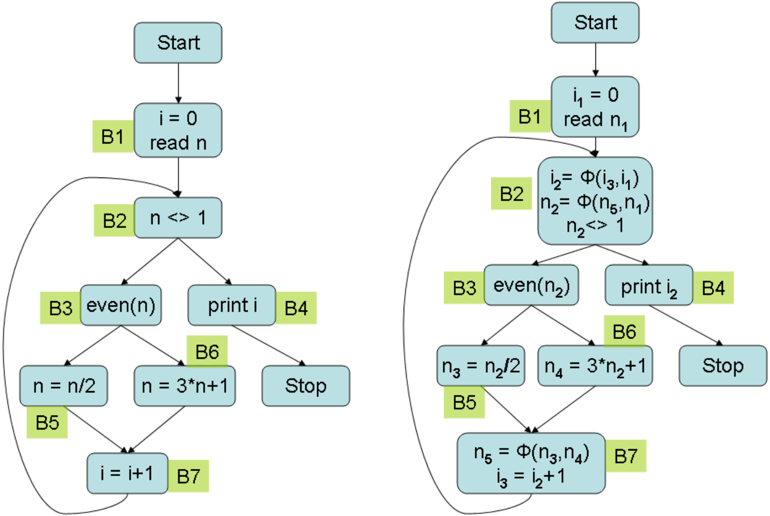
```
read A,B,C
if (A>B)
   if (A>C) max = A
   else max = C
else if (B>C) max = B
      else max = C
printf (max)
```



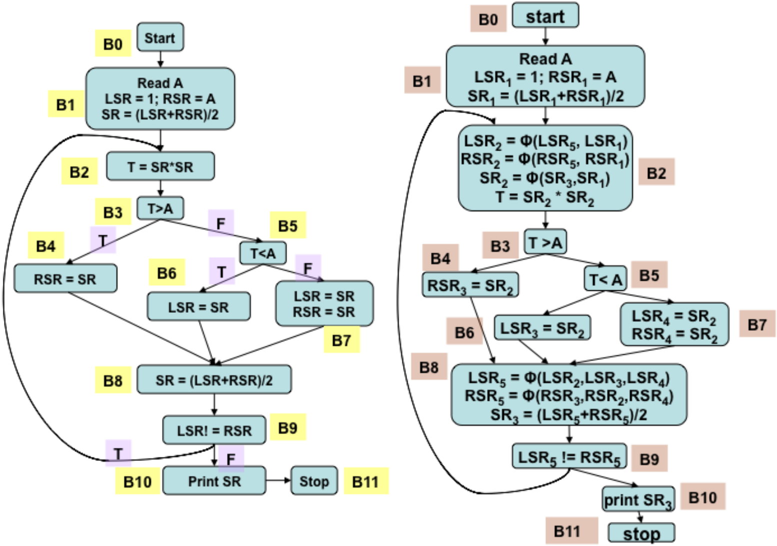
SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, ϕ, is used for selection of values in join nodes
- Not every join node needs a ϕ operator for every variable
- No need for a ϕ operator, if the same definition of the variable reaches the join node along all incoming edges
- Often, an SSA form is augmented with *u-d* and *d-u* chains to facilitate design of faster algorithms
- Translation from SSA to machine code introduces copy operations, which may introduce some inefficiency

Program 2 in non-SSA and SSA Form



Program 3 in non-SSA and SSA Form



Optimization Algorithms with SSA Forms

- Dead-code elimination
  - Very simple, since there is exactly one definition reaching each use
  - Examine the *du-chain* of each variable to see if its use list is empty
  - Remove such variables and their definition statements
  - If a statement such as $x = y + z$ (or $x = \phi(y_1, y_2)$) is deleted, care must be taken to remove the deleted statement from the *du-chains* of y and z (or y_1 and y_2)
- Simple constant propagation
- Copy propagation
- Conditional constant propagation and constant folding
- Global value numbering

Simple Constant Propagation

```
{   Stmtpile = {S|S is a statement in the program}
    while Stmtpile is not empty {
        S = remove(Stmtpile);
        if S is of the form $x = \phi(c, c, ..., c)$ for some constant $c$
              replace S by $x = c$
        if S is of the form $x = c$ for some constant $c$
            delete S from the program
            for all statements T in the du-chain of $x$ do
                substitute $c$ for $x$ in T; simplify T
                Stmtpile = Stmtpile $\cup$ {T}
}
```

Copy propagation is similar to constant propagation

- A single-argument ϕ-function, $x = \phi(y)$, or a copy statement, $x = y$ can be deleted and y substituted for every use of x

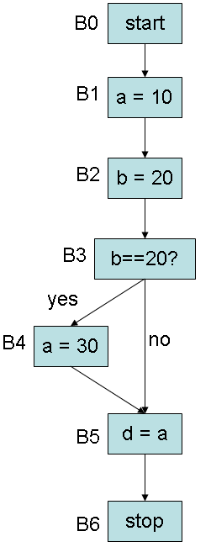
Conditional Constant Propagation - 1

- SSA forms along with extra edges corresponding to *d-u* information are used here
    - Edge from every definition to each of its uses in the SSA form (called henceforth as *SSA edges*)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile* , resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

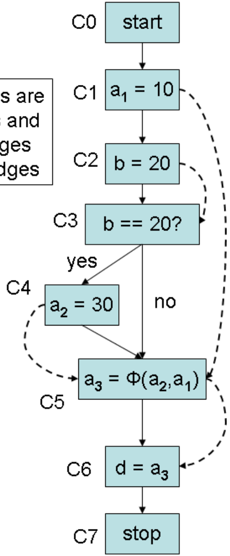
Conditional Constant Propagation - 2

- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs much lesser storage compared to its non-SSA counterpart
- Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to *true* or *false*) or both edges (corresponding to $\perp$) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.

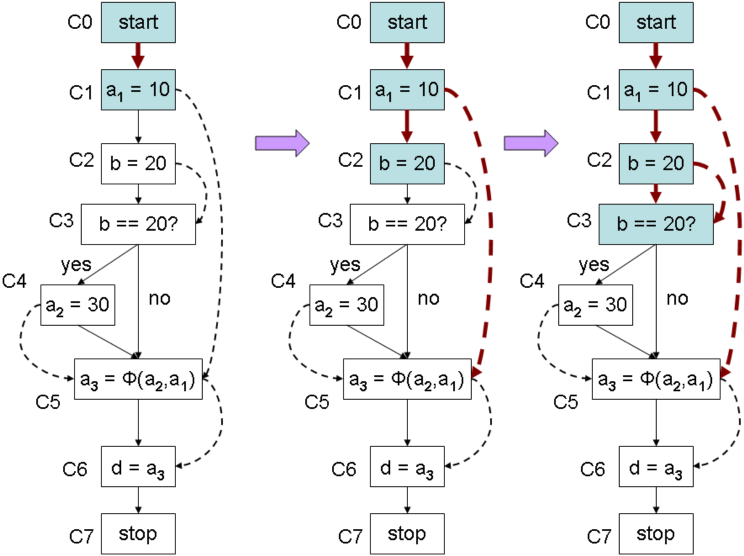
CCP Algorithm - Example - 1



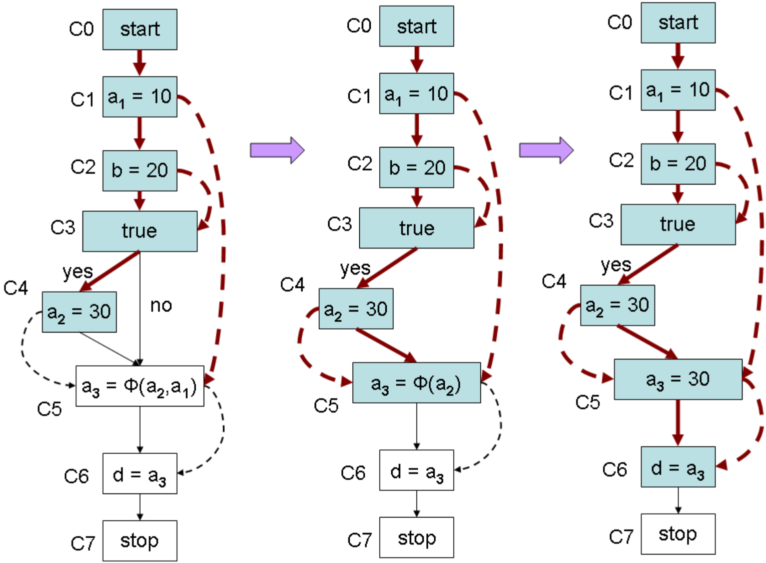
Solid edges are flow edges and dashed edges are SSA edges



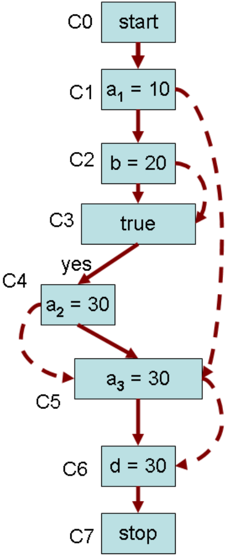
CCP Algorithm - Example 1 - Trace 1



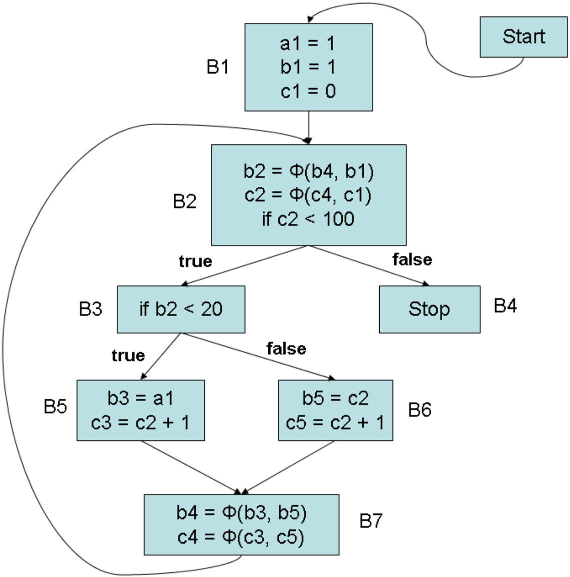
CCP Algorithm - Example 1 - Trace 2



CCP Algorithm - Example 1 - Trace 3



CCP Algorithm - Example 2



Introduction to Machine-Independent Optimizations - 7

Program Optimizations and the SSA Form

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- What is code optimization? (in part 1)
- Illustrations of code optimizations (in part 1)
- Examples of data-flow analysis (in parts 2,3, and 4)
- Fundamentals of control-flow analysis (in parts 4 and 5)
- Algorithms for machine-independent optimizations (in part 6)
- SSA form and optimizations

SSA Form: A Definition

- A program is in SSA form, if each use of a variable is reached by exactly one definition
- Flow of control remains the same as in the non-SSA form
- A special merge operator, ϕ, is used for selection of values in join nodes
- Conditional constant propagation is faster and more effective on SSA forms

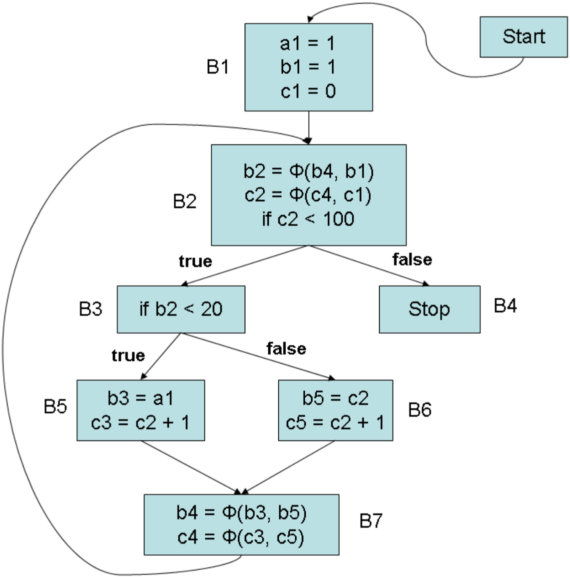
Conditional Constant Propagation - 1

- SSA forms along with extra edges corresponding to *d-u* information are used here
  - Edge from every definition to each of its uses in the SSA form (called henceforth as *SSA edges*)
- Uses both flow graph and SSA edges and maintains two different work-lists, one for each (*Flowpile* and *SSApile* , resp.)
- Flow graph edges are used to keep track of reachable code and SSA edges help in propagation of values
- Flow graph edges are added to *Flowpile*, whenever a branch node is symbolically executed or whenever an assignment node has a single successor

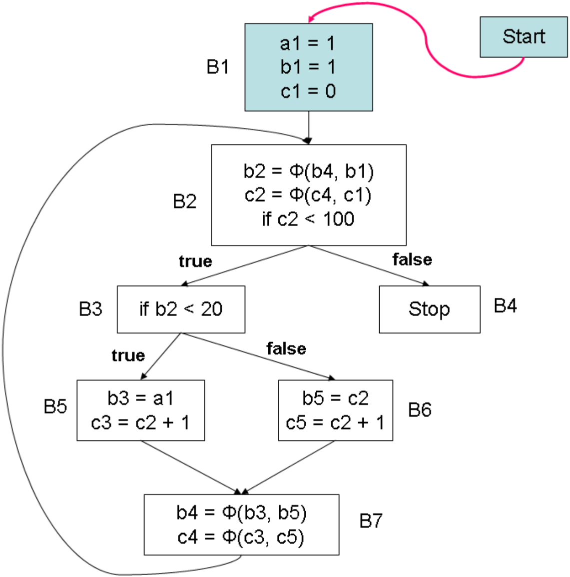
Conditional Constant Propagation - 2

- SSA edges coming out of a node are added to the SSA work-list whenever there is a change in the value of the assigned variable at the node
- This ensures that all *uses* of a definition are processed whenever a definition changes its lattice value.
- This algorithm needs much lesser storage compared to its non-SSA counterpart
- Conditional expressions at branch nodes are evaluated and depending on the value, either one of outgoing edges (corresponding to *true* or *false*) or both edges (corresponding to $\perp$) are added to the worklist
- However, at any join node, the *meet* operation considers only those predecessors which are marked *executable*.

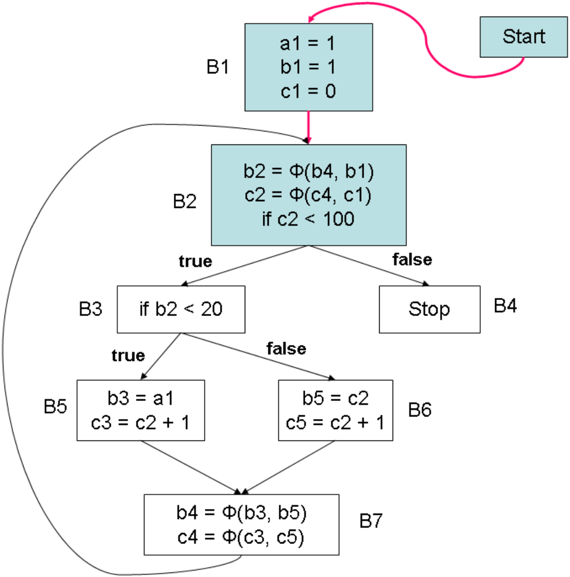
CCP Algorithm - Example 2



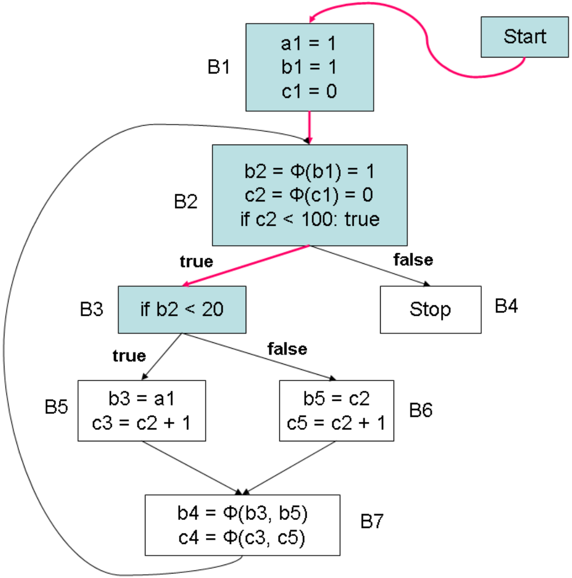
CCP Algorithm - Example 2 - Trace 1



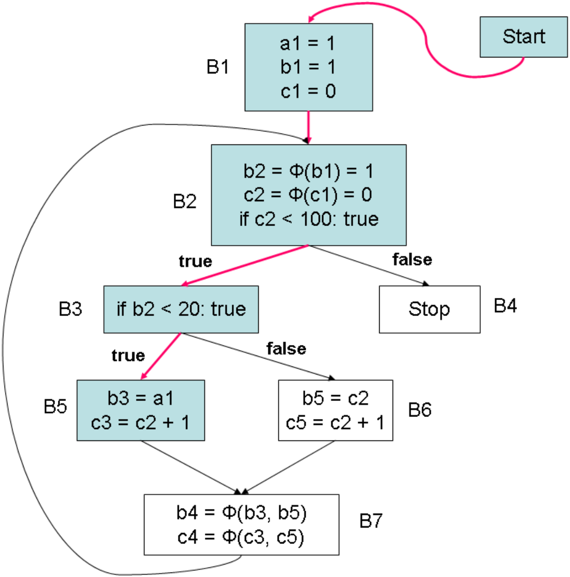
CCP Algorithm - Example 2 - Trace 2



CCP Algorithm - Example 2 - Trace 3



CCP Algorithm - Example 2 - Trace 4



Start

B1
a1 = 1
b1 = 1
c1 = 0

B2
$b2 = \Phi(b1) = 1$
$c2 = \Phi(c1) = 0$
if $c2 < 100$: true

true

false

B3
if $b2 < 20$: true

B4
Stop

true

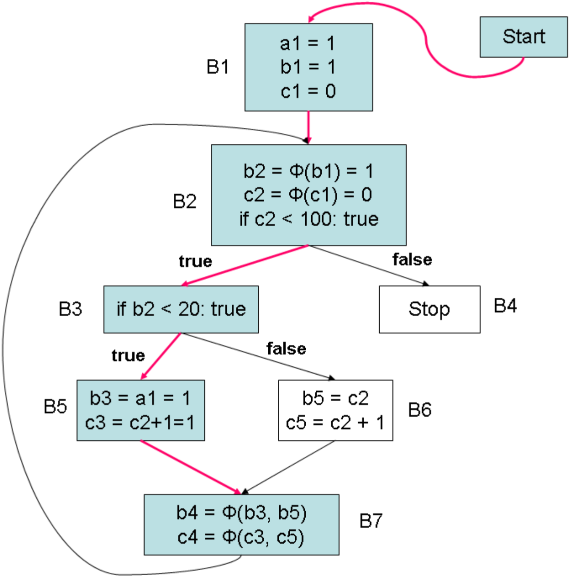
false

B5
b3 = a1
c3 = c2 + 1

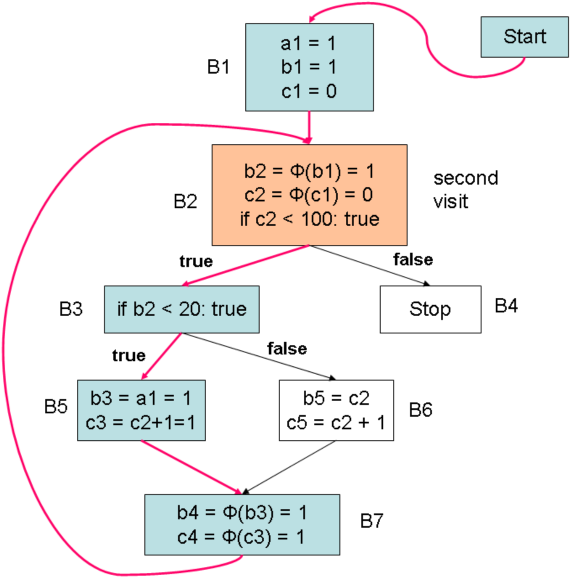
B6
b5 = c2
c5 = c2 + 1

B7
$b4 = \Phi(b3, b5)$
$c4 = \Phi(c3, c5)$

CCP Algorithm - Example 2 - Trace 5



CCP Algorithm - Example 2 - Trace 6



Start

B1
a1 = 1
b1 = 1
c1 = 0

B2
b2 = Φ(b1) = 1
c2 = Φ(c1) = 0
if c2 < 100: true

second visit

true

false

B3
if b2 < 20: true

B4
Stop

true

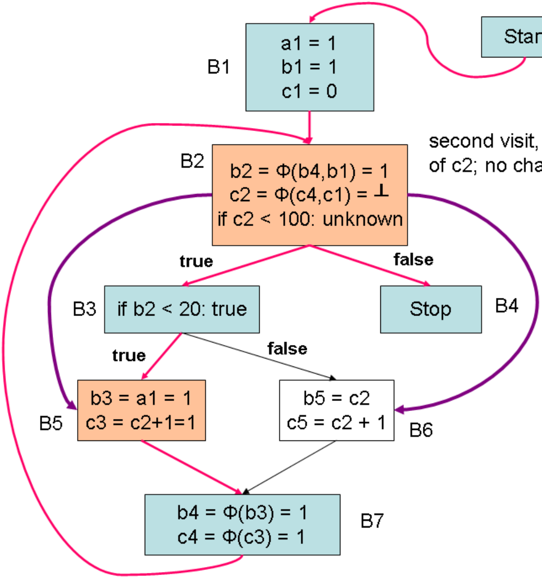
false

B5
b3 = a1 = 1
c3 = c2+1=1

B6
b5 = c2
c5 = c2 + 1

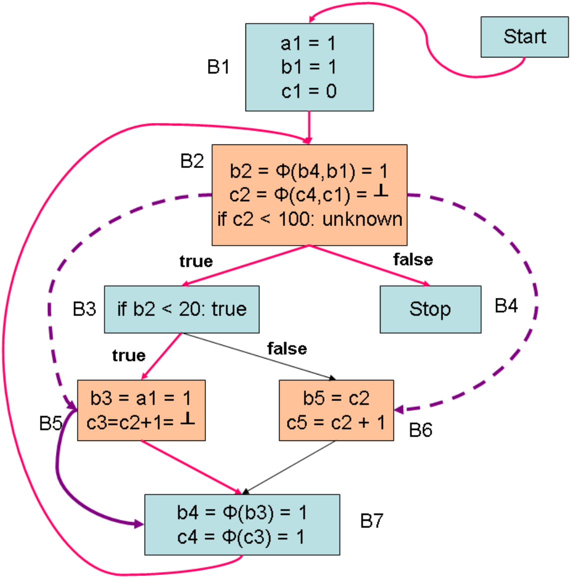
B7
b4 = Φ(b3) = 1
c4 = Φ(c3) = 1

CCP Algorithm - Example 2 - Trace 7

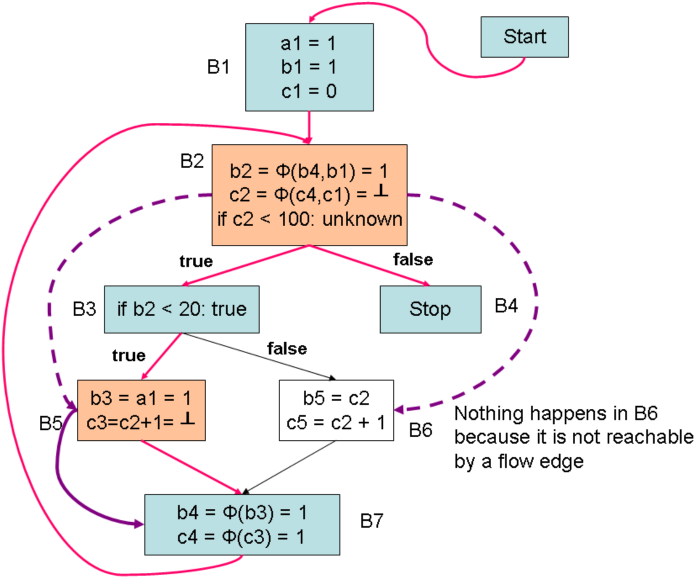


second visit, change in value of c2; no change in value of b2

CCP Algorithm - Example 2 - Trace 8

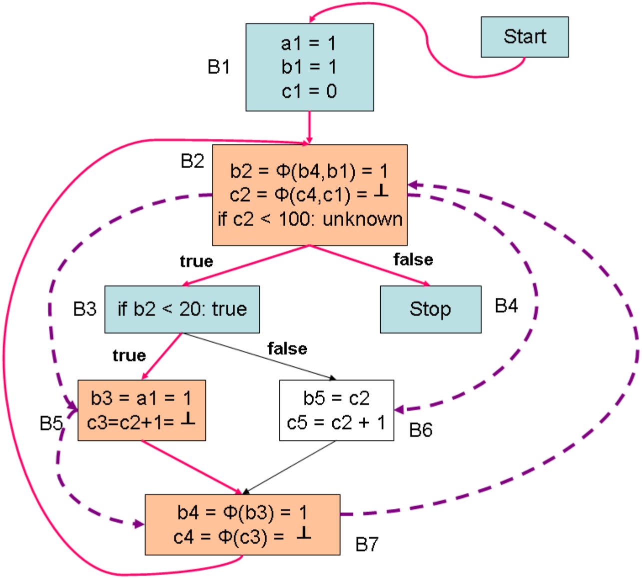


CCP Algorithm - Example 2 - Trace 9

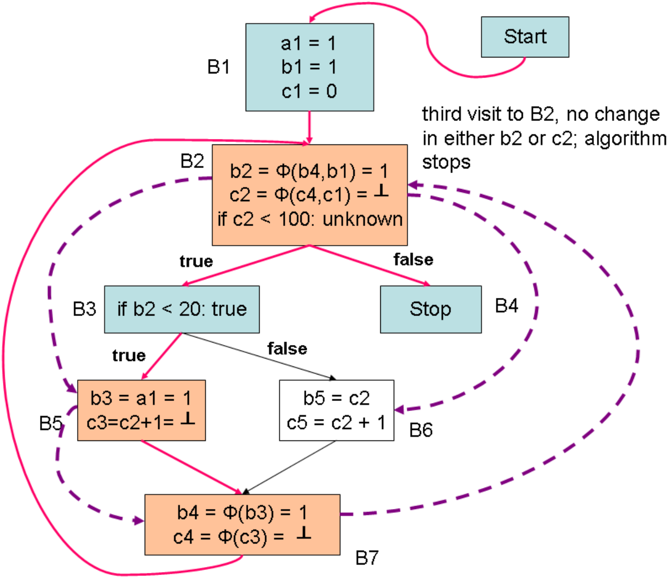


Nothing happens in B6 because it is not reachable by a flow edge

CCP Algorithm - Example 2 - Trace 10



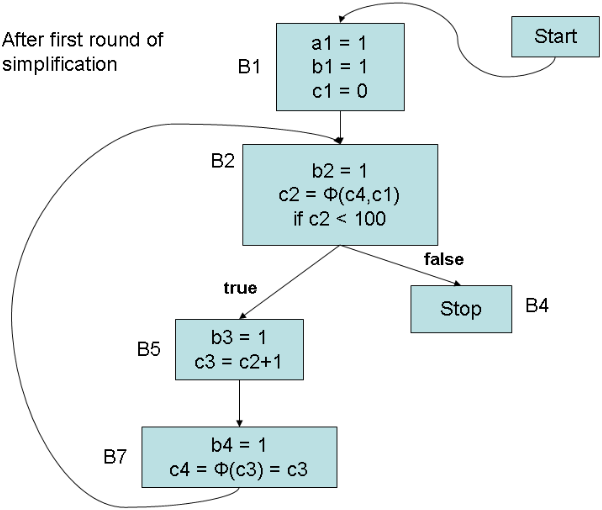
CCP Algorithm - Example 2 - Trace 11



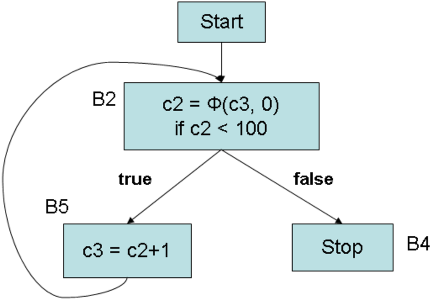
third visit to B2, no change in either b2 or c2; algorithm stops

CCP Algorithm - Example 2 - Trace 12

After first round of simplification



CCP Algorithm - Example 2 - Trace 13



After second round of simplification –
elimination of dead code, elimination
of trivial Φ-functions, copy propagation etc.

Instruction Scheduling and Software Pipelining - 1

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline

- Instruction Scheduling
    - Simple Basic Block Scheduling
    - Trace, Superblock and Hyperblock scheduling
- Software pipelining

Instruction Scheduling

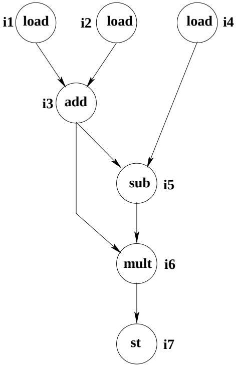
- Reordering of instructions so as to keep the pipelines of functional units full with no stalls
- NP-Complete and needs heuristcs
- Applied on basic blocks (local)
- Global scheduling requires elongation of basic blocks (super-blocks)

Instruction Scheduling - Motivating Example

- time: load - 2 cycles, op - 1 cycle
- This code has 2 stalls, at i3 and at i5, due to the loads

i1:	r1	←	load a
i2:	r2	←	load b
i3:	r3	←	r1 + r2
i4:	r4	←	load c
i5:	r5	←	r3 - r4
i6:	r6	←	r3 * r5
i7:	d	←	st r6

(a) Sample Code Sequence



(b) DAG

Scheduled Code - no stalls

- There are no stalls, but dependences are indeed satisfied

```
i1: | r1  ←  load a
i2: | r2  ←  load b
i4: | r4  ←  load c
i3: | r3  ←  r1 + r2
i5: | r5  ←  r3 - r4
i6: | r6  ←  r3 * r5
i7: | d   ←  st r6
```

Definitions - Dependences

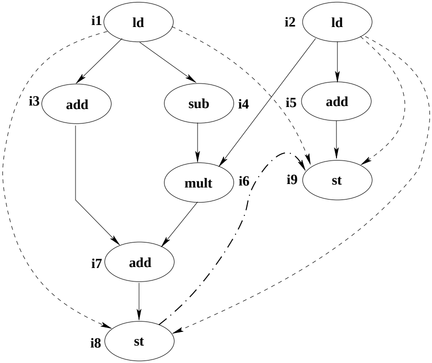
- Consider the following code:
  $i_1 :\ r1 \leftarrow load(r2)$
  $i_2 :\ r3 \leftarrow r1 + 4$
  $i_3 :\ r1 \leftarrow r4 + r5$
- The dependences are
  $i_1\ \delta\ i_2$ (flow dependence) $i_2\ \overline{\delta}\ i_3$ (anti-dependence)
  $i_1\ \delta^o\ i_3$ (output dependence)
- anti- and ouput dependences can be eliminated by register renaming

Dependence DAG

- full line: *flow* dependence, dash line: *anti*-dependence
  dash-dot line: *output* dependence
- some anti- and output dependences are because memory disambiguation could not be done

i1:	t1	←	load a
i2:	t2	←	load b
i3:	t3	←	t1 + 4
i4:	t4	←	t1 - 2
i5:	t5	←	t2 + 3
i6:	t6	←	t4 * t2
i7:	t7	←	t3 + t6
i8:	c	←	st t7
i9:	b	←	st t5

(a) Instruction Sequence



(b) DAG

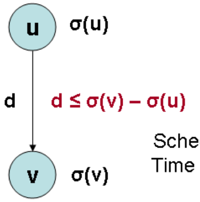
Basic Block Scheduling

- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
  - PC's relate to data dependences and execution delays
  - RC's relate to limited availability of shared resources

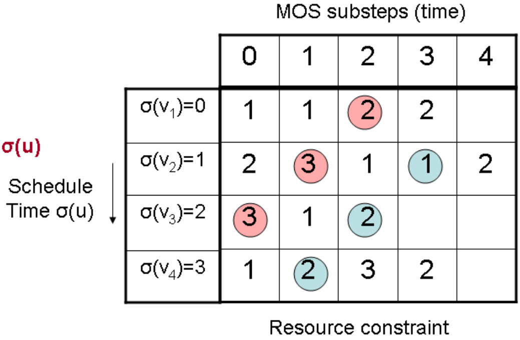
The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, $G = (V, E)$
    - R: number of resources
    - Nodes (V): MOS; Edges (E): Precedence
    - Label on node v
        - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with v
        - length $l(v)$ of node v
    - Label on edge e: Execution delay of the MOS, $d(e)$
- Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that $\forall e = (u, v) \in E,\ \sigma(v) - \sigma(u) \geq d(e)$ and $\forall i, \sum^{v \in V} \rho_v(i - \sigma(v)) \leq R$, where length of the schedule is $\max_{v \in V}\{\sigma(v) + l(v)\}$

Instruction Scheduling - Precedence and Resource Constraints



Precedence constraint


MOS substeps (time)

Schedule Time σ(u)

	0	1	2	3	4
$\sigma(v_1)=0$	1	1	🔴2	2	
$\sigma(v_2)=1$	2	🔴3	1	🔵1	2
$\sigma(v_3)=2$	🔴3	1	🔵2		
$\sigma(v_4)=3$	1	🔵2	3	2	

Resource constraint

Consider R = 5. Each MOS substep takes 1 time unit.

🔵 At i=4, $\varsigma_{v4}(1)+\varsigma_{v3}(2)+\varsigma_{v2}(3)+\varsigma_{v1}(4) = 2+2+1+0 = 5 \leq R$, satisfied

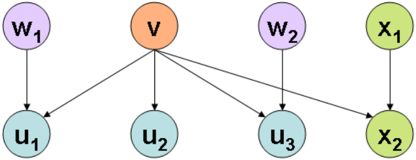
🔴 At i=2, $\varsigma_{v3}(0)+\varsigma_{v2}(1)+\varsigma_{v1}(2) = 3+3+2 = 8 > R$, NOT satisfied

A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

```
FUNCTION ListSchedule (V,E)
BEGIN
  Ready = root nodes of V; Schedule = $\phi$;
  WHILE Ready $\neq \phi$ DO
  BEGIN
    v = highest priority node in Ready;
    Lb = SatisfyPrecedenceConstraints ($v$, $Schedule$, $\sigma$);
    $\sigma$(v) = SatisfyResourceConstraints ($v$, $Schedule$, $\sigma$, $Lb$);
    $Schedule = Schedule + \{v\}$;
    $Ready = Ready - \{v\} + \{u \mid NOT\ (u \in Schedule)$
                  $AND\ \forall\ (w,u) \in E,\ w \in Schedule\}$;
  END
  RETURN $\sigma$;
END
```

List Scheduling - Ready Queue Update



Already scheduled nodes: w

Unscheduled nodes which will get into the Ready queue now: u

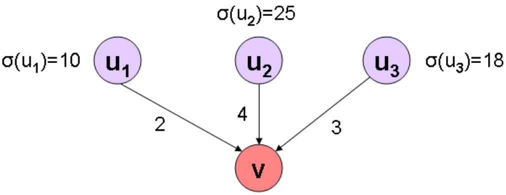
Currently scheduled node: v

Unscheduled nodes: x

Constraint Satisfaction Functions

```
FUNCTION SatisfyPrecedenceConstraint(v, Sched, $\sigma$)
BEGIN
  RETURN ( $\max_{u \in Sched} \sigma(u) + d(u, v)$)
END

FUNCTION SatisfyResourceConstraint(v, Sched, $\sigma$, Lb)
BEGIN
  FOR i := Lb TO $\infty$ DO
    IF $\forall 0 \leq j < l(v),\ \rho_v(j) + \sum^{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R$ THEN
      RETURN (i);
END
```

Precedence Constraint Satisfaction



Lower bound for $\sigma(v)$ = 29

Already scheduled nodes **u**

Node to be scheduled **v**

Precedence constraint satisfaction:

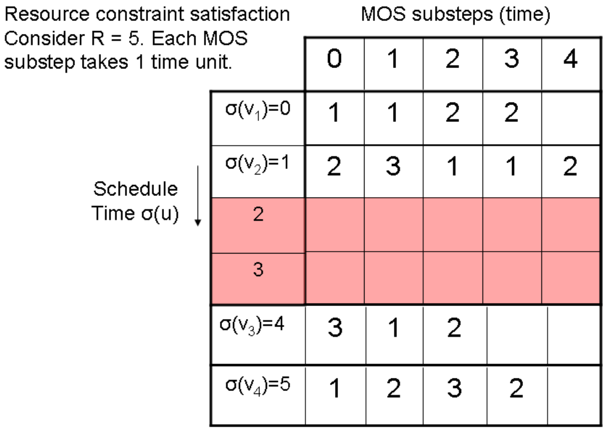
v can be scheduled only after all of u_1, u_2, and, u_3, finish

Lower bound for $\sigma(v)$

$= \max(10+2,\ 25+4,\ 18+3)$

$= \max(12,\ 29,\ 21) = 29$

Resource Constraint Satisfaction



Resource constraint satisfaction
Consider R = 5. Each MOS substep takes 1 time unit.

Schedule Time σ(u) ↓ \ MOS substeps (time)	0	1	2	3	4
$\sigma(v_1)=0$	1	1	2	2	
$\sigma(v_2)=1$	2	3	1	1	2
2					
3					
$\sigma(v_3)=4$	3	1	2		
$\sigma(v_4)=5$	1	2	3	2	

Time slots 2 and 3 are vacant because scheduling node v_3 in either of them violates resource constraints

Instruction Scheduling and Software Pipelining - 2

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline

- Instruction Scheduling
    - Simple Basic Block Scheduling
    - Trace, Superblock and Hyperblock scheduling
- Software pipelining

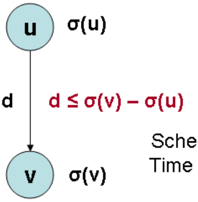
Basic Block Scheduling

- Basic block consists of micro-operation sequences (MOS), which are indivisible
- Each MOS has several steps, each requiring resources
- Each step of an MOS requires one cycle for execution
- Precedence constraints and resource constraints must be satisfied by the scheduled program
    - PC's relate to data dependences and execution delays
    - RC's relate to limited availability of shared resources

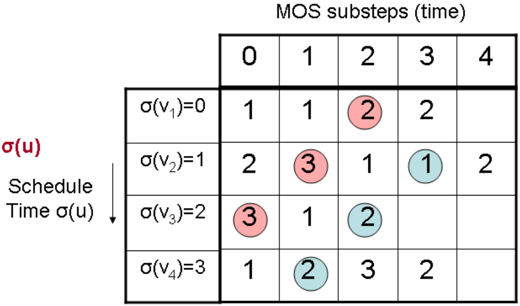
The Basic Block Scheduling Problem

- Basic block is modelled as a digraph, $G = (V, E)$
  - R: number of resources
  - Nodes (V): MOS; Edges (E): Precedence
  - Label on node v
    - resource usage functions, $\rho_v(i)$ for each step of the MOS associated with v
    - length $l(v)$ of node v
  - Label on edge e: Execution delay of the MOS, $d(e)$
- Problem: Find the shortest schedule $\sigma : V \rightarrow N$ such that $\forall e = (u, v) \in E,\ \sigma(v) - \sigma(u) \geq d(e)$ and $\forall i, \sum^{v \in V} \rho_v(i - \sigma(v)) \leq R$, where length of the schedule is $\max\limits_{v \in V}\{\sigma(v) + l(v)\}$

Instruction Scheduling - Precedence and Resource Constraints



Precedence constraint



	0	1	2	3	4
$\sigma(v_1)=0$	1	1	🔴 2	2	
$\sigma(v_2)=1$	2	🔴 3	1	🔵 1	2
$\sigma(v_3)=2$	🔴 3	1	🔵 2		
$\sigma(v_4)=3$	1	🔵 2	3	2	

Resource constraint

Consider R = 5. Each MOS substep takes 1 time unit.

🔵 At i=4, $\varsigma_{v4}(1)+\varsigma_{v3}(2)+\varsigma_{v2}(3)+\varsigma_{v1}(4) = 2+2+1+0 = 5 \leq R$, satisfied

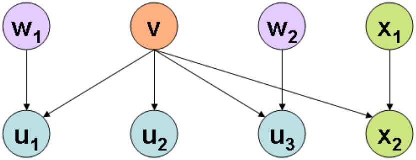
🔴 At i=2, $\varsigma_{v3}(0)+\varsigma_{v2}(1)+\varsigma_{v1}(2) = 3+3+2 = 8 > R$, NOT satisfied

A Simple List Scheduling Algorithm

Find the shortest schedule $\sigma : V \rightarrow N$, such that precedence and resource constraints are satisfied. Holes are filled with NOPs.

```
FUNCTION ListSchedule (V,E)
BEGIN
  Ready = root nodes of V; Schedule = $\phi$;
  WHILE Ready $\neq \phi$ DO
  BEGIN
    v = highest priority node in Ready;
    Lb = SatisfyPrecedenceConstraints ($v, Schedule, \sigma$);
    $\sigma$(v) = SatisfyResourceConstraints ($v, Schedule, \sigma, Lb$);
    $Schedule = Schedule + \{v\}$;
    $Ready = Ready - \{v\} + \{u \mid NOT\ (u \in Schedule)$
                 $AND\ \forall\ (w, u) \in E,\ w \in Schedule\}$;
  END
  RETURN $\sigma$;
END
```

List Scheduling - Ready Queue Update



Already scheduled nodes w

Unscheduled nodes which will get into the Ready queue now u

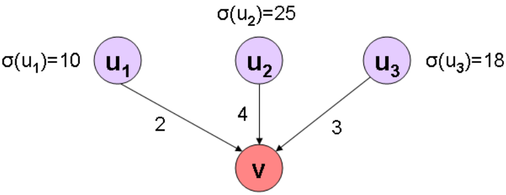
Currently scheduled node v

Unscheduled nodes x

Constraint Satisfaction Functions

```
FUNCTION SatisfyPrecedenceConstraint(v, Sched, $\sigma$)
BEGIN
  RETURN ( $\max_{u \in Sched} \sigma(u) + d(u, v)$)
END

FUNCTION SatisfyResourceConstraint(v, Sched, $\sigma$, Lb)
BEGIN
  FOR i := Lb TO $\infty$ DO
    IF $\forall 0 \leq j < l(v),\ \rho_v(j) + \sum^{u \in Sched} \rho_u(i + j - \sigma(u)) \leq R$ THEN
      RETURN (i);
END
```

Precedence Constraint Satisfaction



Lower bound for $\sigma(v)$ = 29

Already scheduled nodes

Node to be scheduled

Precedence constraint satisfaction:

v can be scheduled only after all of u_1, u_2, and, u_3, finish

Lower bound for $\sigma(v)$

$$= \max(10+2,\ 25+4,\ 18+3)$$
$$= \max(12,\ 29,\ 21) = 29$$

Resource Constraint Satisfaction

Resource constraint satisfaction
Consider R = 5. Each MOS
substep takes 1 time unit.

MOS substeps (time)

Schedule Time σ(u)

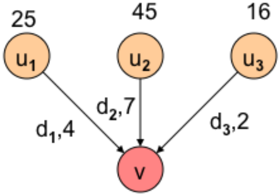
Schedule Time σ(u)	0	1	2	3	4
$\sigma(v_1)=0$	1	1	2	2	
$\sigma(v_2)=1$	2	3	1	1	2
2					
3					
$\sigma(v_3)=4$	3	1	2		
$\sigma(v_4)=5$	1	2	3	2	

Time slots 2 and 3 are vacant because scheduling node v_3 in either of them violates resource constraints

List Scheduling - Priority Ordering for Nodes

1. Height of the node in the DAG (*i.e.,* longest path from the node to a terminal node
2. *Estart*, and *Lstart*, the earliest and latest start times
   - Violating *Estart* and *Lstart* may result in pipeline stalls
   - $Estart(v) = \max_{i=1,\cdots,k} (Estart(u_i) + d(u_i, v))$ where $u_1, u_2, \cdots, u_k$ are predecessors of v. *Estart* value of the source node is 0.
   - $Lstart(u) = \min_{i=1,\cdots,k} (Lstart(v_i) - d(u, v_i))$ where $v_1, v_2, \cdots, v_k$ are successors of u. *Lstart* value of the sink node is set as its *Estart* value.
   - *Estart* and *Lstart* values can be computed using a top-down and a bottom-up pass, respectively, either statically (before scheduling begins), or dynamically during scheduling

Estart Computation

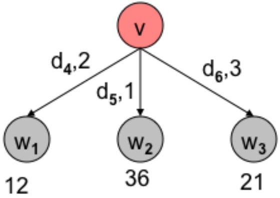


$$\textit{Estart}\,(v) = \max_{i=1,\ldots,3} (\textit{Esart}\,(u_i) + d_i)$$

$$= \max(25+4, 45+7, 16+2)$$

$$= \max(29, 52, 18) = 52$$

Lstart Computation



$$\mathit{Lstart}\,(v) = \min_{i = 4,\dots,6} (\mathit{Lsart}\,(w_i) - d_i)$$

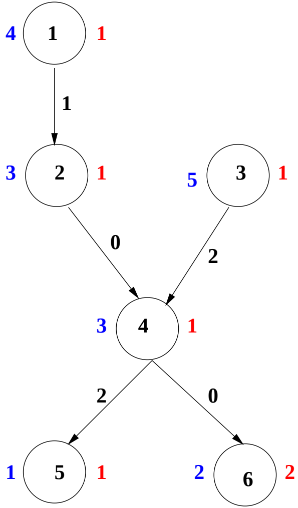
$$= \min(12-2,\ 36-1,\ 21-3)$$
$$= \min(10,\ 35,\ 18) = 10$$

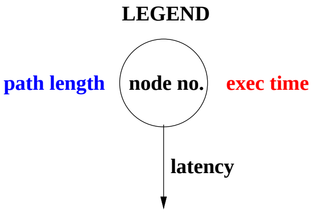
List Scheduling - Slack

1. A node with a lower *Estart* (or *Lstart*) value has a higher priority
2. $Slack = Lstart - Estart$
    - Nodes with lower slack are given higher priority
    - Instructions on the critical path may have a slack value of zero and hence get priority

Simple List Scheduling - Example - 1

INSTRUCTION SCHEDULING - EXAMPLE





path length (n) = exec time (n) , if n is a leaf

= max { latency (n,m) + path length (m) }
m ε succ (n)

Schedule = {3, 1, 2, 4, 6, 5}

Simple List Scheduling - Example - 2

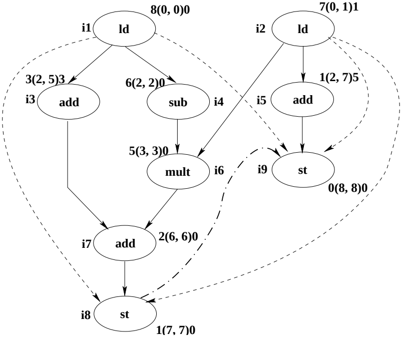
- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
- *path length* and *slack* are shown on the left side and right side of the pair of numbers in parentheses

```
c = (a+4)+(a-2)*b;
b = b+3;
```

(a) High-Level Code

```
i1:  t1  ←  load a
i2:  t2  ←  load b
i3:  t3  ←  t1 + 4
i4:  t4  ←  t1 - 2
i5:  t5  ←  t2 + 3
i6:  t6  ←  t4 * t2
i7:  t7  ←  t3 + t6
i8:  c   ←  st t7
i9:  b   ←  st t5
```

(b) 3-Address Code



(c) DAG with *(Estart, Lstart)* Values

Simple List Scheduling - Example - 2 (contd.)

- latencies
  - *add,sub,store*: 1 cycle; *load*: 2 cycles; *mult*: 3 cycles
  - 2 Integer units and 1 Multiplication unit, all capable of load and store as well
- Heuristic used: height of the node or slack

int1	int2	mult	Cycle #	Instr.No.	Instruction
1	1	0	0	i1, i2	$t_1 \leftarrow load\ a, t_2 \leftarrow load\ b$
1	1	0	1		
1	1	0	2	i4, i3	$t_4 \leftarrow t_1 - 2, t_3 \leftarrow t_1 + 4$
1	0	1	3	i6, i5	$t_5 \leftarrow t_2 + 3, t_6 \leftarrow t_4 * t_2$
0	0	1	4		i5 not sched. in cycle 2
0	0	1	5		due to shortage of *int* units
1	0	0	6	i7	$t_7 \leftarrow t_3 + t_6$
1	0	0	7	i8	$c \leftarrow st\ t_7$
1	0	0	8	i9	$b \leftarrow st\ t_5$

Resource Usage Models - Instruction Reservation Table

	r_0	r_1	r_2	r_3	r_4
t_0	1	0	1	2	0
t_1	1	1	0	0	1
t_2	0	0	0	2	1
t_3	0	1	0	0	1

No. of resources in the machine: 4

Resource Usage Models - Global Reservation Table

	r_0	r_1	r_2	$\cdots$	r_M
t_0	1	0	1		0
t_1	1	1	0		1
t_2	0	0	0		1
t_T					

M: No. of resources in the machine
T: Length of the schedule

Resource Usage Models - Global Reservation Table

- GRT is constructed as the schedule is built (cycle by cycle)
- All entries of GRT are initialized to 0
- GRT maintains the state of all the resources in the machine
- GRTs can answer questions of the type:
  "can an instruction of class I be scheduled in the current cycle (say t_k)?"
- Answer is obtained by ANDing RT of I with the GRT starting from row t_k
  - If the resulting table contains only 0's, then YES, otherwise NO
- The GRT is updated after scheduling the instruction with a similar OR operation

Simple List Scheduling - Disadvantages

- Checking resource constraints is inefficient here because it involves repeated ANDing and ORing of bit matrices for many instructions in each scheduling step
- Space overhead may become considerable, but still manageable

Global Acyclic Scheduling

- Average size of a basic block is quite small (5 to 20 instructions)
  - Effectiveness of instruction scheduling is limited
  - This is a serious concern in architectures supporting greater ILP
    - VLIW architectures with several function units
    - superscalar architectures (multiple instruction issue)
- Global scheduling is for a set of basic blocks
  - Overlaps execution of successive basic blocks
  - Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.

Trace Scheduling

- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
    - Identify the most frequently executed basic block
    - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

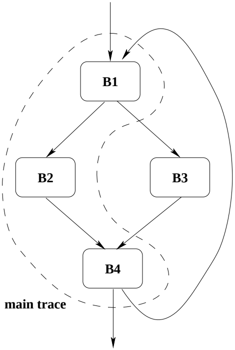
Trace Example

```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		%% r1 ← 0 %% r5 ← 0 %% r6 ← 400 %% r7 ← s
B1:	i1:	r2 ← load a(r1)
	i2:	if (r2 != 0) goto i7
B2:	i3:	r3 ← load b(r1)
	i4:	r4 ← r3 + r7
	i5:	b(r1) ← r4
	i6:	goto i9
B3:	i7:	r4 ← r2
	i8:	b(r1) ← r2
B4:	i9:	r5 ← r5 + r4
	i10:	r1 ← r1 + 4
	i11:	if (r1 < r6) goto i1

(b) Assembly Code



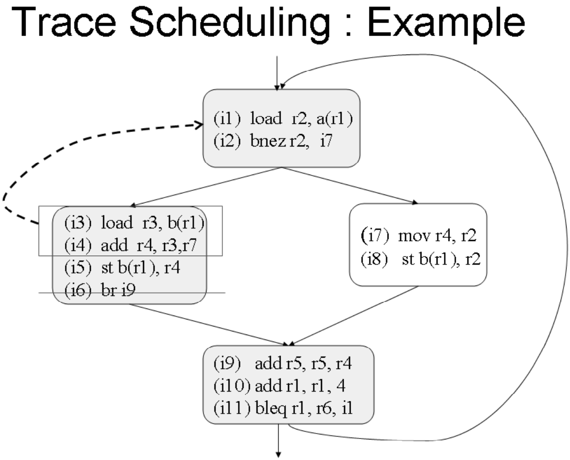
(c) Control Flow Graph

Trace - Basic Block Schedule

- 2-way issue architecture with 2 integer units
- *add, sub, store*: 1 cycle, *load*: 2 cycles, *goto*: no stall
- 9 cycles for the main trace and 6 cycles for the off-trace

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)		
1				
2	i2:	if (r2 != 0) goto i7		
3	i3:	r3 ← load b(r1)		
4				
5	i4:	r4 ← r3 + r7		
6	i5:	b(r1) ← r4	i6:	goto i9
3	i7:	r4 ← r2	i8:	b(r1) ← r2
7 (4)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
8 (5)	i11:	if (r1 < r6) goto i1		

Trace Schedule



Trace Scheduling : Example

```
(i1 ) load r2, a(r1 )
(i2) bnez r2, i7
```

```
(i3) load r3, b(r1)
(i4) add r4, r3,r7
(i5) st b(r1 ), r4
(i6) br i9
```

```
(i7) mov r4, r2
(i8) st b(r1 ), r2
```

```
(i9) add r5, r5, r4
(i10) add r1, r1, 4
(i11) bleq r1, r6, i1
```

Trace Schedule

- 6 cycles for the main trace and 7 cycles for the off-trace

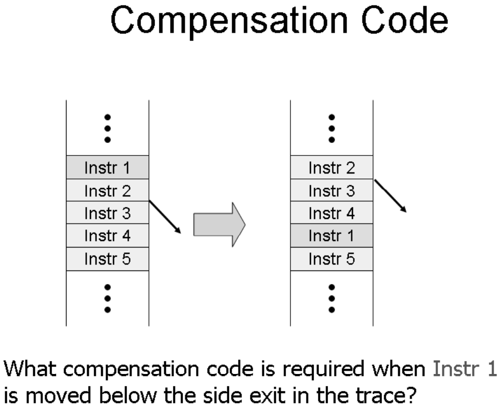
Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2 != 0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4		
4 (5)	i9:	r5 ← r5 + r4	i10:	r1 ← r1 + 4
5 (6)	i11:	if (r1 < r6) goto i1		

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i12:	goto i9		

Trace Scheduling - Issues

- *Side exits* and *side entrances* are ignored during scheduling of a trace
- Required compensation code is inserted during book-keeping (after scheduling the trace)
- Speculative code motion - *load* instruction moved ahead of conditional branch
    - Example: Register r3 should not be live in block B3 (off-trace path)
    - May cause unwanted exceptions
        - Requires additional hardware support!

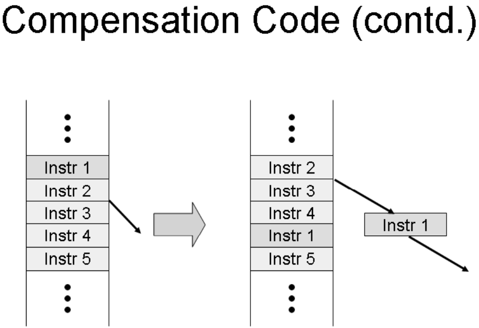
Compensation Code - Side Exit



Compensation Code

What compensation code is required when Instr 1 is moved below the side exit in the trace?

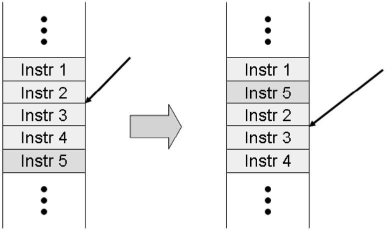
Compensation Code - Side Exit



Compensation Code (contd.)

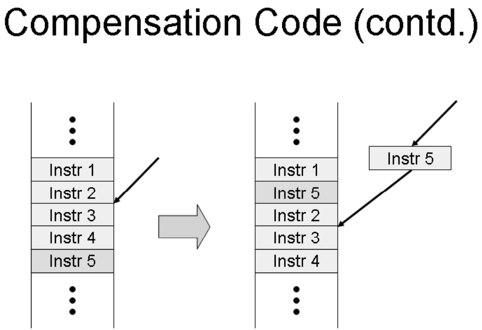
Compensation Code - Side Entry

Compensation Code (contd.)



What compensation code is required when Instr 5 moves above the side entrance in the trace?

Compensation Code - Side Entry



Compensation Code (contd.)

Instruction Scheduling and Software Pipelining - 3

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline

- Instruction Scheduling
    - Simple Basic Block Scheduling
    - Trace, Superblock and Hyperblock scheduling
- Software pipelining

Global Acyclic Scheduling

- Average size of a basic block is quite small (5 to 20 instructions)
  - Effectiveness of instruction scheduling is limited
  - This is a serious concern in architectures supporting greater ILP
    - VLIW architectures with several function units
    - superscalar architectures (multiple instruction issue)
- Global scheduling is for a set of basic blocks
  - Overlaps execution of successive basic blocks
  - Trace scheduling, Superblock scheduling, Hyperblock scheduling, Software pipelining, etc.

Trace Scheduling

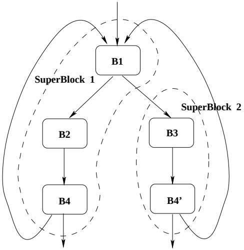
- A Trace is a frequently executed acyclic sequence of basic blocks in a CFG (part of a path)
- Identifying a trace
  - Identify the most frequently executed basic block
  - Extend the trace starting from this block, forward and backward, along most frequently executed edges
- Apply list scheduling on the trace (including the branch instructions)
- Execution time for the trace may reduce, but execution time for the other paths may increase
- However, overall performance will improve

Superblock Scheduling

- A Superblock is a trace without side entrances
    - Control can enter only from the top
    - Many exits are possible
    - Eliminates several book-keeping overheads
- Superblock formation
    - Trace formation as before
    - Tail duplication to avoid side entrances into a superblock
    - Code size increases

Superblock Example

- 5 cycles for the main trace and 6 cycles for the off-trace



(a) Control Flow Graph

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2:	if (r2!=0) goto i7	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4	i10:	r1 ← r1 + 4
4	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

3	i7:	r4 ← r2	i8:	b(r1) ← r2
4	i9':	r5 ← r5 + r4	i10':	r1 ← r1 + 4
5	i11':	if (r1<r6) goto i1		

(b) Superblock Schedule

Hyperblock Scheduling

- Superblock scheduling does not work well with control-intensive programs which have many control flow paths
- Hyperblock scheduling was proposed to handle such programs
- Here, the control flow graph is IF-converted to eliminate conditional branches
- IF-conversion replaces conditional branches with appropriate predicated instructions
- Now, control dependence is changed to a data dependence

IF-Conversion Example

```
for I = 1 to 100 do {
   if (A(I) <= 0) then contnue
   A(I) = B(I) + 3
}
```

```
for I = 1 to 100 do {
   p = (A(I) <= 0)
   (!p)  A(I) = B(I) + 3
}
```

```
          for I = 1 to N do {
S1:          A(I) = D(I) + 1
S2:          if (B(I) > 0) then
S3:              C(I) = C(I) + A(I)
S4:            else   D(I+1) = D(I+1) + 1
               end if
          }
```

```
          for I = 1 to N do {
S1:          A(I) = D(I) + 1
S2:          p = (B(I) > 0)
S3:          (p)  C(I) = C(I) + A(I)
S4:          (!p)  D(I+1) = D(I+1) + 1
          }
```

Hyperblock Example Code

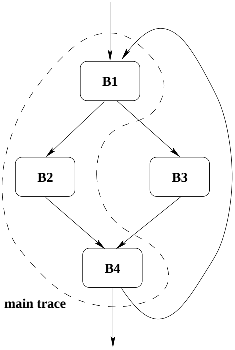
```
for (i=0; i < 100; i++)
{
   if (A[i] == 0)
     B[i] = B[i] + s;
   else
     B[i] = A[i];
   sum = sum + B[i];
}
```

(a) High-Level Code

		`%% r1 ← 0`
		`%% r5 ← 0`
		`%% r6 ← 400`
		`%% r7 ← s`

B1:	i1:	`r2 ← load a(r1)`
	i2:	`if (r2 != 0) goto i7`
B2:	i3:	`r3 ← load b(r1)`
	i4:	`r4 ← r3 + r7`
	i5:	`b(r1) ← r4`
	i6:	`goto i9`
B3:	i7:	`r4 ← r2`
	i8:	`b(r1) ← r2`
B4:	i9:	`r5 ← r5 + r4`
	i10:	`r1 ← r1 + 4`
	i11:	`if (r1 < r6) goto i1`

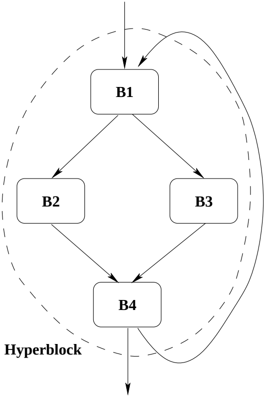
(b) Assembly Code



(c) Control Flow Graph

Hyperblock Example

- 6 cycles for the entire set of predicated instructions
- Instructions i3 and i4 can be executed speculatively and can be moved up, instead of being scheduled after cycle 2



(a) Control Flow Graph

Time	Int. Unit 1		Int. Unit 2	
0	i1:	r2 ← load a(r1)	i3:	r3 ← load b(r1)
1				
2	i2':	p1 ← (r2 == 0)	i4:	r4 ← r3 + r7
3	i5:	b(r1) ← r4, if p1	i8:	b(r1) ← r2, if !p1
4	i10:	r1 ← r1 + 4	i7:	r4 ← r2, if !p1
5	i9:	r5 ← r5 + r4	i11:	if (r1<r6) goto i1

(b) Hyperblock Schedule

Introduction to Software Pipelining

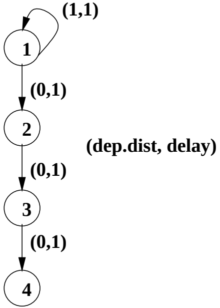
- Overlaps execution of instructions from multiple iterations of a loop
- Executes instructions from different iterations in the same pipeline, so that pipelines are kept busy without stalls
- Objective is to sustain a high initiation rate
    - Initiation of a subsequent iteration may start even before the previous iteration is complete
- Unrolling loops several times and performing global scheduling on the unrolled loop
    - Exploits greater ILP within unrolled iterations
    - Very little or no overlap across iterations of the loop

Introduction to Software Pipelining - contd.

- More complex than instruction scheduling
- NP-Complete
- Involves finding initiation interval for successive iterations
  - Trial and error procedure
  - Start with minimum II, schedule the body of the loop using one of the approaches below and check if schedule length is within bounds
    - Stop, if yes
    - Try next value of II, if no
- Requires a modulo reservation table (GRT with II columns and R rows)
- Schedule lengths are dependent on II, dependence distance between instructions and resource contentions

Software Pipelining Example-1

```
for (i=1; i<=n; i++) {
  a[i+1] = a[i] + 1;
  b[i] = a[i+1]/2;
  c[i] = b[i] + 3;
  d[i] = c[i]
}
```



Iterations

TIME							
1	S1						
2	S2	S1					
3	S3	S2	S1				
4	S4	S3	S2	S1			
5		S4	S3	S2	S1		
6			S4	S3	S2	S1	
7				S4	S3	S2	S1
8					S4	S3	S2
9						S4	S3
10							S4

Software Pipelining Example-2.1

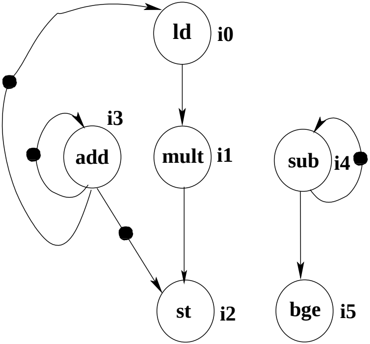
No. of tokens present on an arc indicates the dependence distance

```
for (i = 0; i < n; i++) {
    a[i] = s * a[i];
}
```

(a) High-Level Code

```
     % t0 ← 0 %
     % t1 ← (n-1) %
     % t2 ← s %
i0:  t3      ←  load a(t0)
i1:  t4      ←  t2 * t3
i2:  a(t0)   ←  t4
i3:  t0      ←  t0 + 4
i4:  t1      ←  t1 - 1
i5:  if (t1 ≥ 0) goto i0
```

(b) Instruction Sequence



(c) Dependence graph

Software Pipelining Example

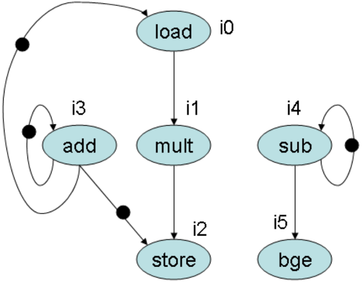
Software Pipelining Example-2.2

- Number of tokens present on an arc indicates the dependence distance
- Assume that the possible dependence from i2 to i0 can be disambiguated
- Assume 2 INT units (latency 1 cycle), 2 FP units (latency 2 cycles), and 1 LD/STR unit (latency 2 cycles/1 cycle)
- Branch can be executed by INT units
- Acyclic schedule takes 5 cycles (see figure)
- Corresponds to an initiation rate of 1/5 iteration per cycle
- Cyclic schedule takes 2 cycles (see figure)

Acyclic and Cyclic Schedules

Acyclic Schedule

0	i0: load
1	
2	i1: mult, i3: add, i4: sub
3	
4	i2: store, i5: bge



Cyclic Schedule

4	i4: sub	i1: mult	i0: load
5	i2: store i5: bge	i3: add	

Software Pipelining Example-2.3

Time Step	Iter. 0	Iter. 1	Iter. 2	
0	i0 : ld			**Prolog**
1				
2	i1 : mult	i0 : ld		
3	i3 : add			
4	i4 : sub	i1 : mult	i0 : ld	**Kernel**
5	i2 : st i5 : bge	i3 : add		
6		i4 : sub	i1 : mult	**Epilog**
7		i2 : st i5 : bge	i3 : add	
8			i4 : sub	
9			i2 : st i5 : bge	

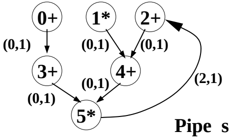
A Software Pipelined Schedule with II = 2

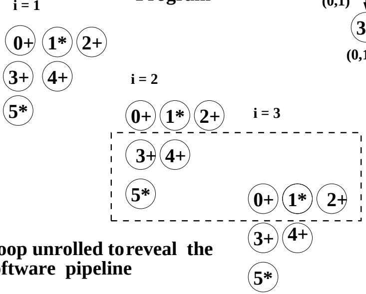
Software Pipelining Example-3

```
for i = 1 to n {
   0: t0[i] = a[i] + b[i];
   1: t1[i] = c[i] * const1;
   2: t2[i] = d[i] + e[i−2];
   3: t3[i] = t0[i] + c[i];
   4: t4[i] = t1[i] + t2[i];
   5:   e[i] = t3[i] * t4[i];
}
```

Program

Dependence Graph





Loop unrolled to reveal the software pipeline

Pipe stages

time	PS0	PS1
0	3+ 4+	
1	5*	0+ 1* 2+

2 multipliers, 2 adders, 1 cluster, single cycle operations

Automatic Parallelization - 1

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Automatic Parallelization

- Automatic conversion of sequential programs to parallel programs by a compiler
- Target may be a vector processor (vectorization), a multi-core processor (concurrentization), or a cluster of loosely coupled distributed memory processors (parallelization)
- Parallelism extraction process is normally a source-to-source transformation
- Requires dependence analysis to determine the dependence between statements
- Implementation of available parallelism is also a challenge
  - For example, can all the iterations of a 2-nested loop be run in parallel?

Example 1

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

X(1:100) = X(1:100) + Y(1:100)
```

The above code can be run on a vector processor in O(1) time. The vectors X and Y are fetched first and then the vector X is written into

Example 2

```
for I = 1 to 100 do {
   X(I) = X(I) + Y(I)
}

can be converted to

forall I = 1 to 100 do {
   X(I) = X(I) + Y(I)
```

The above code can be run on a multi-core processor with all the 100 iterations running as separate threads. Each thread “owns” a different I value

Example 3

```
for I = 1 to 100 do {
   X(I+1) = X(I) + Y(I)
}
```

`cannot be converted to`

```
X(2:101) = X(1:100) + Y(1:100)
```

`because of dependence as shown below`

```
X(2) = X(1) + Y(1)
X(3) = X(2) + Y(2)
X(4) = X(3) + Y(3)
...
```

Data Dependence Relations

Flow or true dependence

S1: X = ...

↓

S2: ... = X

δ

Anti-dependence

S1: ... = X

↓

S2: X = ...

$\bar{\delta}$

Output dependence

S1: X = ...

↓

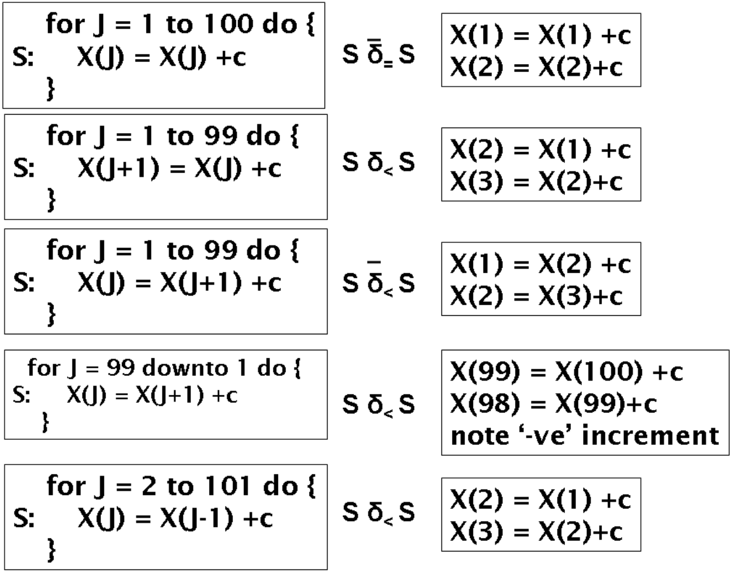
S2: X = ...

δ^o

Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence (direction vector)
- There is one direction vector component for each loop in a nest of loops
- The *data dependence direction vector* (or direction vector) is $\Psi = (\Psi_1, \Psi_2, ..., \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- Forward or "<" direction means dependence from iteration i to $i+k$ (*i.e.,* computed in iteration i and used in iteration $i+k$)
- Backward or ">" direction means dependence from iteration i to $i-k$ (*i.e.,* computed in iteration i and used in iteration $i-k$). This is not possible in single loops and possible in two or higher levels of nesting
- Equal or "=" direction means that dependence is in the same iteration (*i.e.,* computed in iteration i and used in iteration i)

Direction Vector Example 1



```
   for J = 1 to 100 do {
S:    X(J) = X(J) +c
   }
```

$S\ \bar{\delta}_{=}\ S$

```
X(1) = X(1) +c
X(2) = X(2)+c
```

```
   for J = 1 to 99 do {
S:    X(J+1) = X(J) +c
   }
```

$S\ \delta_{<}\ S$

```
X(2) = X(1) +c
X(3) = X(2)+c
```

```
   for J = 1 to 99 do {
S:    X(J) = X(J+1) +c
   }
```

$S\ \bar{\delta}_{<}\ S$

```
X(1) = X(2) +c
X(2) = X(3)+c
```

```
   for J = 99 downto 1 do {
S:    X(J) = X(J+1) +c
   }
```

$S\ \delta_{<}\ S$

```
X(99) = X(100) +c
X(98) = X(99)+c
```

note ‘-ve’ increment

```
   for J = 2 to 101 do {
S:    X(J) = X(J-1) +c
   }
```

$S\ \delta_{<}\ S$

```
X(2) = X(1) +c
X(3) = X(2)+c
```

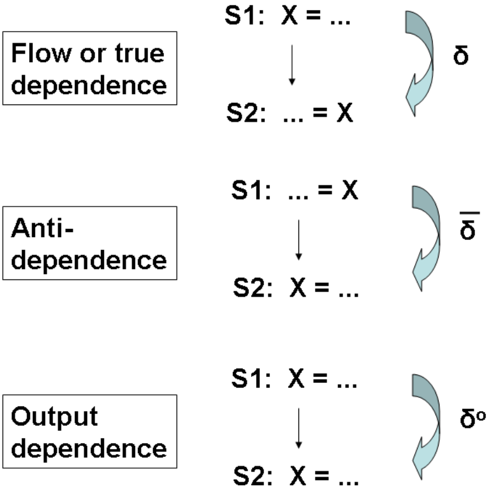
Automatic Parallelization - 2

Y.N. Srikant

Department of Computer Science
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Data Dependence Relations



Flow or true dependence

S1: X = ...

S2: ... = X

δ

Anti-dependence

S1: ... = X

S2: X = ...

$\bar{\delta}$

Output dependence

S1: X = ...

S2: X = ...

δ^o

Data Dependence Direction Vector

- Data dependence relations are augmented with a direction of data dependence (direction vector)
- There is one direction vector component for each loop in a nest of loops
- The *data dependence direction vector* (or direction vector) is $\Psi = (\Psi_1, \Psi_2, ..., \Psi_d)$, where $\Psi_k \in \{<, =, >, \leq, \geq, \neq, *\}$
- Forward or "<" direction means dependence from iteration i to $i + k$ (*i.e.,* computed in iteration i and used in iteration $i + k$)
- Backward or ">" direction means dependence from iteration i to $i - k$ (*i.e.,* computed in iteration i and used in iteration $i - k$). This is not possible in single loops and possible in two or higher levels of nesting
- Equal or "=" direction means that dependence is in the same iteration (*i.e.,* computed in iteration i and used in iteration i)

Direction Vector Example 1

```
   for J = 1 to 100 do {
S:     X(J) = X(J) +c
   }
```

$S \, \bar{\delta}_{=} \, S$

X(1) = X(1) +c
X(2) = X(2)+c

```
   for J = 1 to 99 do {
S:     X(J+1) = X(J) +c
   }
```

$S \, \delta_{<} \, S$

X(2) = X(1) +c
X(3) = X(2)+c

```
   for J = 1 to 99 do {
S:     X(J) = X(J+1) +c
   }
```

$S \, \bar{\delta}_{<} \, S$

X(1) = X(2) +c
X(2) = X(3)+c

```
  for J = 99 downto 1 do {
S:    X(J) = X(J+1) +c
  }
```

$S \, \delta_{<} \, S$

X(99) = X(100) +c
X(98) = X(99)+c
note '-ve' increment

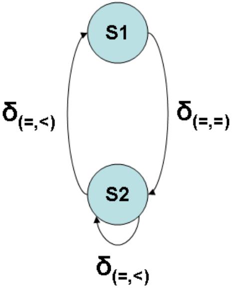
```
   for J = 2 to 101 do {
S:     X(J) = X(J-1) +c
   }
```

$S \, \delta_{<} \, S$

X(2) = X(1) +c
X(3) = X(2)+c

Direction Vector Example 2

```
        for I = 1 to 5 do {
            for J = 1 to 4 do {
S1:             A(I, J) = B(I, J) + C(I, J)
S2:             B(I, J+1) = A(I, J) + B(I, J)
            }
        }
```



Demonstration of direction vector

```
I=1, J=1:  A(1,1)=B(1,1)+C(1,1)
           B(1,2)=A(1,1)+B(1,1)
     J=2:  A(1,2)=B(1,2)+C(1,2)
           B(1,3)=A(1,2)+B(1,2)
     J=3:  A(1,3)=B(1,3)+C(1,3)
           B(1,4)=A(1,3)+B(1,3)
```

S1 $\delta_{(=,=)}$ S2

S2 $\delta_{(=,<)}$ S1

S2 $\delta_{(=,<)}$ S2

Direction Vector Example 3

$S1\ \delta_{(<,>)}\ S2$

```
     for I = 1 to N do {
        for J = 1 to N do {
S1:        A(I+1, J) = ...
S2:        ... = A(I, J+1)
        }
     }
```

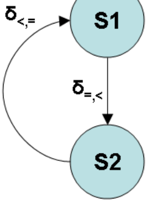
```
I = 1, J = 2
S1:    A(2,2) = ...

I = 2, J = 1
S2:    ... = A(2,2)
```

$S2\ \delta_{(<,>)}\ S1$

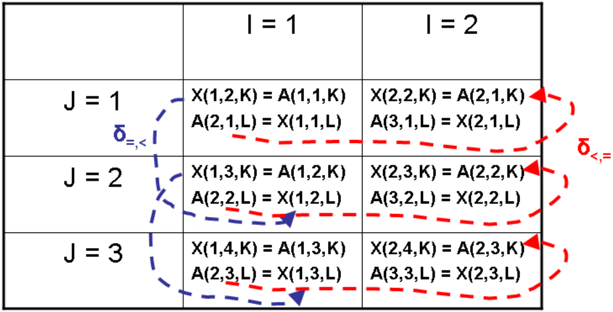
```
     for I = 1 to N do {
        for J = 1 to N do {
S1:        ... = A(I, J+1)
S2:        A(I+1, J) = ...
        }
     }
```

```
I = 1, J = 2
S2:    A(2,2) = ...

I = 2, J = 1
S1:    ... = A(2,2)
```

Direction Vector Example 4



```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
         for K = 1 to 100 do {
S1:         X(I, J+1, K) = A(I, J, K) + 10
         }
         for L = 1 to 50 do {
S2:         A(I+1, J, L) = X(I, J, L) +5
         }
      }
    }
```



	I = 1	I = 2
J = 1	X(1,2,K) = A(1,1,K) A(2,1,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,1,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,2,L) = X(1,2,L)	X(2,3,K) = A(2,2,K) A(3,2,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,3,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,3,L) = X(2,3,L)

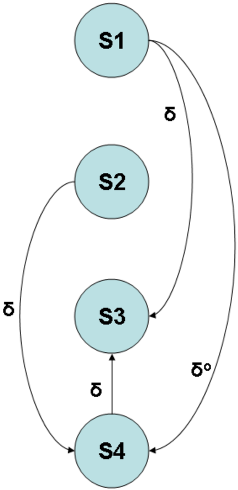
Data Dependence Graph and Vectorization

- Individual nodes are statements of the program and edges depict data dependence among the statements
- If the DDG is acyclic, then vectorization of the program is possible and is straightforward
    - Vector code generation can be done using a topological sort order on the DDG
- Otherwise, find all the strongly connected components of the DDG, and reduce the DDG to an acyclic graph by treating each SCC as a single node
    - SCCs cannot be fully vectorized; the final code will contain some sequential loops and possibly some vector code

Data Dependence Graph and Vectorization

- If all the dependence relations in a loop nest have a direction vector value of "=" for a loop, then the iterations of that loop can be executed in parallel with no synchronization between iterations
- Any dependence with a forward (<) direction in an outer loop will be satisfied by the serial execution of the outer loop
- If an outer loop L is run in sequential mode, then all the *dependences* with a forward (<) direction at the outer level (of L) will be automatically satisfied (even those of the loops inner to L)
- However, this is not true for those dependences with with (=) direction at the outer level; the dependences of the inner loops will have to be satisfied by appropriate statement ordering and loop execution order

Vectorization Example 1



```
      for I = 1 to 99 {
S1:    X(I) = I
S2:    B(I) = 100 – I
      }
      for I = 1 to 99 {
S3:    A(I) = F(X(I))
S4:    X(I+1) = G(B(I))
      }
```

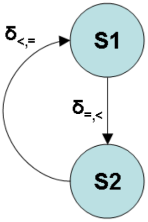
Loop A

Loop B

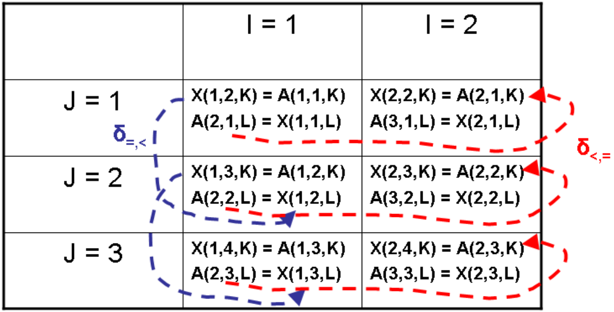
```
X(1:99) = (/1:99/)
B(1:99) = (/99:1:-1/)
X(2:100) = G(B(1:99))
A(1:99) = F(X(1:99))
```

Loop A is parallelizable, but loop B is not, due to forward dependence of S3 on S4

Vectorization Example 2.1

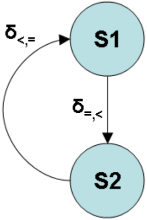


```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
         for K = 1 to 100 do {
S1:         X(I, J+1, K) = A(I, J, K) + 10
         }
         for L = 1 to 50 do {
S2:         A(I+1, J, L) = X(I, J, L) +5
         }
      }
    }
```



	I = 1	I = 2
J = 1	X(1,2,K) = A(1,1,K) A(2,1,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,1,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,2,L) = X(1,2,L)	X(2,3,K) = A(2,2,K) A(3,2,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,3,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,3,L) = X(2,3,L)

Vectorization Example 2.2



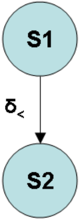
```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
         for K = 1 to 100 do {
S1:            X(I, J+1, K) = A(I, J, K) + 10
         }
         for L = 1 to 50 do {
S2:            A(I+1, J, L) = X(I, J, L) +5
         }
      }
    }
```

I loop cannot be vectorized due to the cycle.

I and J loops cannot be parallelized, due to '<' direction vector. K and L loops can be parallelized

```
for I = 1 to 100 do {
     X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
     A(I+1, 1:100, 1:50) = X(I, 1:100, 1:50) + 5
}
```

Vectorization Example 2.3

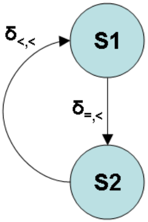


```
    for I = 1 to 100 do {
      for J = 1 to 100  do {
         for K = 1 to 100 do {
S1:          X(I, J+1, K) = A(I, J, K) + 10
         }
         for L = 1 to 50 do {
S2:          A(I+1, J, L) = X(I, J, L) +5
         }
      }
    }
```

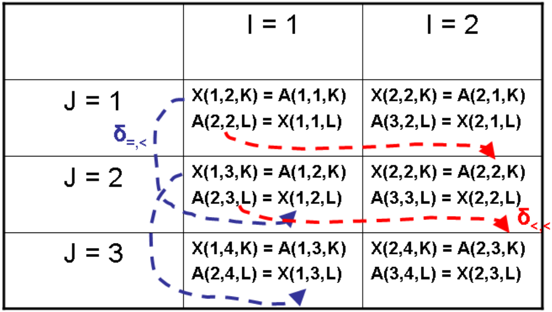
If the I loop is run sequentially, the I-loop dependences are satisfied; J-loop dependences change as shown and there are no more cycles. The loops can be vectorized. However, J-loop cannot be (still) parallelized.

```
for I = 1 to 100 do {
     X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
     A(I+1, 1:100, 1:50) = X(I, 1:100, 1:50) + 5
}
```

Vectorization Example 2.4

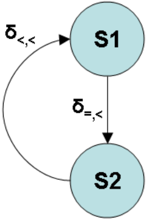


```
     for I = 1 to 100 do {
       for J = 1 to 100  do {
          for K = 1 to 100 do {
S1:          X(I, J+1, K) = A(I, J, K) + 10
          }
          for L = 1 to 50 do {
S2:          A(I+1, J+1, L) = X(I, J, L) +5
          }
       }
     }
```



	I = 1	I = 2
J = 1	X(1,2,K) = A(1,1,K) A(2,2,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,2,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,3,L) = X(1,2,L)	X(2,2,K) = A(2,2,K) A(3,3,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,4,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,4,L) = X(2,3,L)

Vectorization Example 2.5



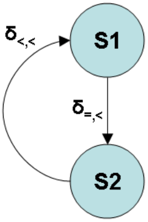
If the program is changed slightly, then dependences change as shown. I and J loops are not parallelizable. If I and J loops are interchanged and J-loop is run sequentially, I-loop can be parallelized. K and L loops are always parallelizable.

```
      for I = 1 to 100 do {
        for J = 1 to 100  do {
           for K = 1 to 100 do {
S1:             X(I, J+1, K) = A(I, J, K) + 10
           }
           for L = 1 to 50 do {
S2:             A(I+1, J+1, L) = X(I, J, L) +5
           }
        }
      }
```

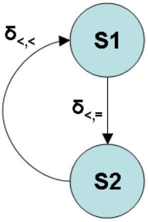
```
for I = 1 to 100 do {
     X(I, 2:101, 1:100) = A(I, 1:100, 1:100) + 10
     A(I+1, 2:101, 1:50) = X(I, 1:100, 1:50) + 5
}
```

Vectorization Example 2.6

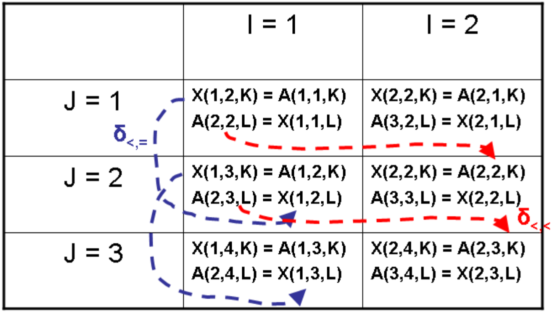
Before interchange



```
      for J = 1 to 100 do {
        for I = 1 to 100  do {
          for K = 1 to 100 do {
S1:          X(I, J+1, K) = A(I, J, K) + 10
          }
          for L = 1 to 50 do {
S2:          A(I+1, J+1, L) = X(I, J, L) +5
          }
        }
      }
```



After interchange



	I = 1	I = 2
J = 1	X(1,2,K) = A(1,1,K) A(2,2,L) = X(1,1,L)	X(2,2,K) = A(2,1,K) A(3,2,L) = X(2,1,L)
J = 2	X(1,3,K) = A(1,2,K) A(2,3,L) = X(1,2,L)	X(2,2,K) = A(2,2,K) A(3,3,L) = X(2,2,L)
J = 3	X(1,4,K) = A(1,3,K) A(2,4,L) = X(1,3,L)	X(2,4,K) = A(2,3,K) A(3,4,L) = X(2,3,L)

Concurrentization Examples

```
      for I = 2 to N do {
        for J = 2 to N do {
S1:       A(I,J) = B(I,J) + 2
S2:       B(I,J) = A(I-1, J-1) – B(I,J)
        }
      }
```

$S1\ \delta_{(<,<)}\ S2,\ S1\ \overline{\delta}_{(=,=)}\ S2,\ S2\ \overline{\delta}_{(=,=)}\ S2$

	I = 1	I = 2
J = 1	A(2,2)= = A(1,1)	A(3,2)= = A(2,1)
J = 2	A(2,3)= = A(1,2)	A(3,3)= = A(2,2)
J = 3	A(2,4)= = A(1,3)	A(3,4)= = A(2,3)

If the I loop is run in serial mode then, the J loop can be run in parallel mode

```
      for I = 2 to N do {
        for J = 2 to N do {
S1:       A(I,J) = B(I,J) + 2
S2:       B(I,J) = A(I, J-1) – B(I,J)
        }
      }
```

$S1\ \delta_{(=,<)}\ S2,\ S1\ \overline{\delta}_{(=,=)}\ S2,\ S2\ \overline{\delta}_{(=,=)}\ S2$

	I = 1	I = 2
J = 1	A(2,2)= = A(2,1)	A(3,2)= = A(3,1)
J = 2	A(2,3)= = A(2,2)	A(3,3)= = A(3,2)
J = 3	A(2,4)= = A(2,3)	A(3,4)= = A(3,3)

The J loop cannot be run in parallel mode. However, the I loop can be run in parallel mode

Loop Transformations for increasing Parallelism

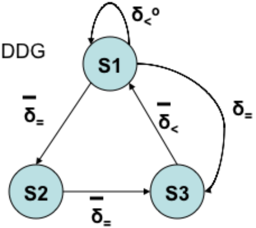
- Recurrence breaking
  - Ignorable cycles
  - Scalar expansion
  - Scalar renaming
  - Node splitting
  - Threshold detection and index set splitting
  - If-conversion
- Loop interchanging
- Loop fission
- Loop fusion

Scalar Expansion

Not vectorizable or parallelizable

```
for I = 1 to N do {
  S1:    T = A(I)
  S2:    A(I) = B(I)
  S3:    B(I) = T
}
```

Cyclic DDG



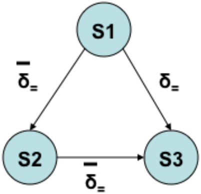
Vectorizable due to scalar expansion

```
for I = 1 to N do {
  S1:    Tx(I) = A(I)
  S2:    A(I) = B(I)
  S3:    B(I) = Tx(I)
}
```

Parallelizable due to privatization

```
forall I = 1 to N do {
  private temp
  S1:    temp = A(I)
  S2:    A(I) = B(I)
  S3:    B(I) = temp
}
```

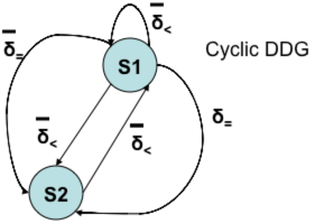
Acyclic DDG



Scalar Expansion is not always profitable

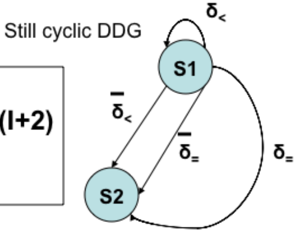
Not vectorizable or parallelizable

```
for I = 1 to N do {
  S1:    T = T + A(I) + A(I+2)
  S2:    A(I) = T
}
```



Not vectorizable even
after scalar expansion

```
for I = 1 to N do {
  S1:    Tx(I) = Tx(I-1)+A(I)+A(I+2)
  S2:    A(I) = Tx(I)
}
```



Scalar Renaming

The output dependence S1 δ^o S3 cannot be broken by scalar expansion

1.

```
for I = 1 to N do {
  S1:    T = A(I) + B(I)
  S2:    C(I) = T*2
  S3:    T = D(I) * B(I)
  S4:    A(I+2) = T + 5
}
```

The output dependence S1 δ^o S3 CAN be broken by scalar renaming

2.

```
for I = 1 to N do {
  S1:    T1 = A(I) + B(I)
  S2:    C(I) = T1*2
  S3:    T2 = D(I) * B(I)
  S4:    A(I+2) = T2 + 5
}
```

3.

```
S3:    T2(1:100) = D(1:100) * B(1:100)
S4:    A(3:102) = T2(1:100) + 5(1:100)
S1:    T1(1:100) = A(1:100) + B(1:100)
S2:    C(1:100) = T1(1:100)*2(1:100)
        T = T2(100)
```

5(1:100) and 2(1:100) are vectors of constants

If-Conversion

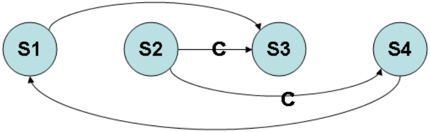
```
for I = 1 to 100 do {
   if (A(I) <= 0) then contnue
   A(I) = B(I) + 3
}
```

```
for I = 1 to 100 do {
   BR(I) = (A(I) <= 0)
   if (~ BR(I))  then
      A(I) = B(I) + 3
}
```

```
BR(1:N) = (A(1:N) <= 0)
where (~ BR(1:N))
   A(1:N) = B(1:N) + 3
```

```
           for I = 1 to N do {
S1:           A(I) = D(I) + 1
S2:           if (B(I) > 0) then
S3:              C(I) = C(I) + A(I)
S4:              D(I+1) = D(I+1) + 1
              end if
           }
```

```
           for I = 1 to N do {
S2:           temp(1:N) = B(1:N) > 0
S4:           where (temp(1:N))
                   D(2:N+1) = D(2:N+1) + 1
S1:           A(1:N) = D(1:N) + 1
S3:           where (temp(1:N))
                   C(1:N) = C(1:N) + A(1:N)
           }
```



Loop Interchange

- For machines with vector instructions, inner loops are preferrable for vectorization, and loops can be interchanged to enable this
- For multi-core and multi-processor machines, parallel outer loops are preferred and loop interchange may help to make this happen
- Requirements for simple loop interchange
    1. The loops L1 and L2 must be tightly nested (no statements between loops)
    2. The loop limits of L2 must be invariant in L1
    3. There are no statements S_v and S_w (not necessarily distinct) in L1 with a dependence $S_v \ \delta^*_{(<,>)} \ S_w$

Loop Interchange for Vectorizability

```
for I = 1 to N do {
   for J = 1 to N do {
S:       A(I,J+1) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Inner loop is not vectorizable

$S\ \delta_{(=,<)}\ S$

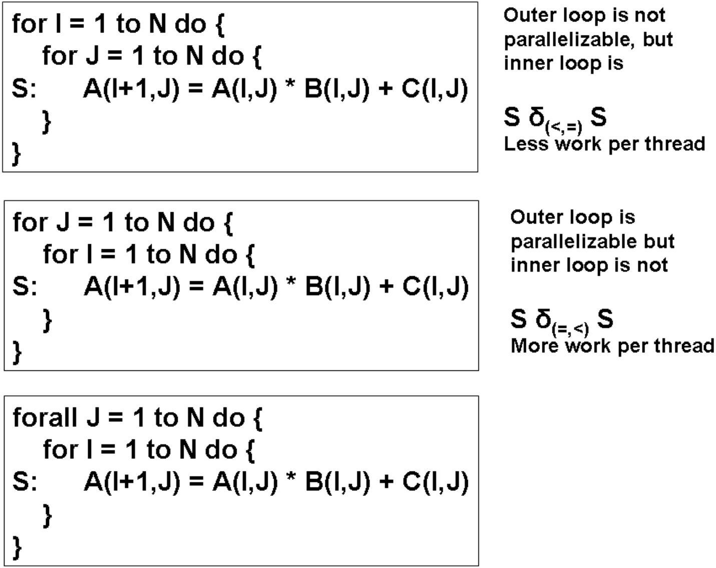
```
for J = 1 to N do {
   for I = 1 to N do {
S:       A(I,J+1) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Inner loop is vectorizable

$S\ \delta_{(<,=)}\ S$

```
for J = 1 to N do {
S:       A(1:N, J+1) = A(1:N, J) * B(1:N, J) + C(1:N, J)
}
```

Loop Interchange for parallelizability



```
for I = 1 to N do {
   for J = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Outer loop is not parallelizable, but inner loop is

$S\ \delta_{(<,=)}\ S$

Less work per thread

```
for J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

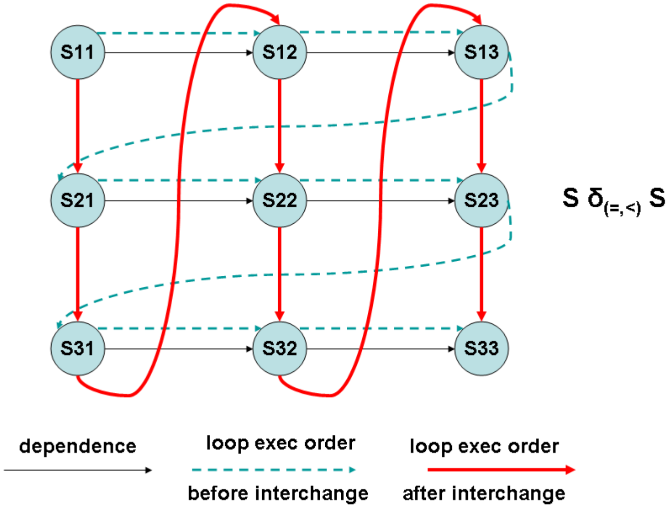
Outer loop is parallelizable but inner loop is not

$S\ \delta_{(=,<)}\ S$

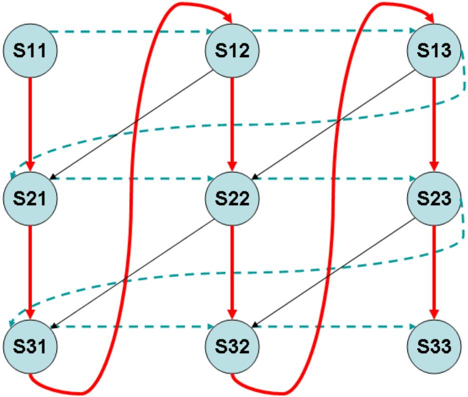
More work per thread

```
forall J = 1 to N do {
   for I = 1 to N do {
S:      A(I+1,J) = A(I,J) * B(I,J) + C(I,J)
   }
}
```

Legal Loop Interchange



Illegal Loop Interchange



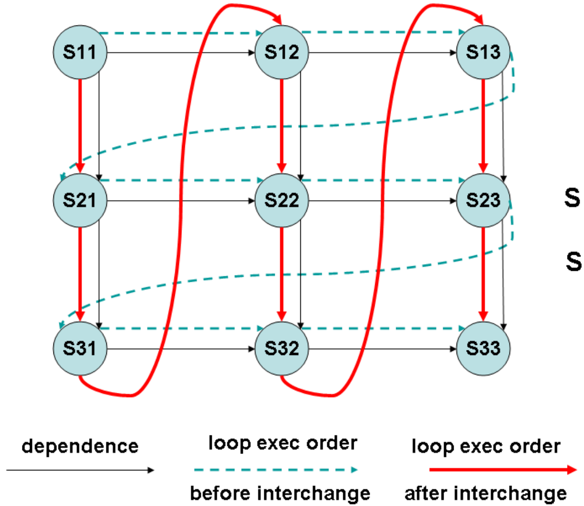
$S\ \delta_{(<,>)}\ S$

dependence

loop exec order before interchange

loop exec order after interchange

Legal but not beneficial Loop Interchange



S $\delta_{(=,<)}$ S
and
S $\delta_{(<,=)}$ S

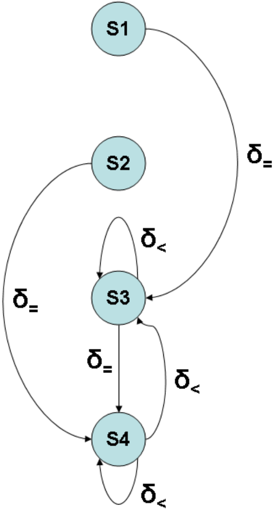
Loop Fission - Motivation

```
     for I = 1 to N do {
S1:      A(I) = E(I) + 1
S2:      B(I) = F(I) * 2
S3:      C(I+1) = C(I) * A(I) + D(I)
S4:      D(I+1) = C(I+1) * B(I) + D(I)
     }
```

The above loop cannot be vectorized

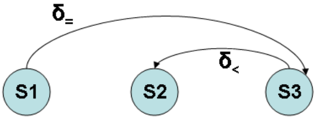
```
L1:  for I = 1 to N do {
S1:      A(I) = E(I) + 1
S2:      B(I) = F(I) * 2
     }

L2:  for I = 1 to N do {
S3:      C(I+1) = C(I) * A(I) + D(I)
S4:      D(I+1) = C(I+1) * B(I) + D(I)
     }
```

L1 can be vectorized, but L2 cannot be



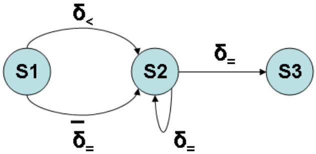
Loop Fission: Legal and Illegal

```
     for I = 1 to N do {
S1:      A(I) = D(I) * T
S2:      B(I) = (C(I) + E(I))/2
S3:      C(I+1) = A(I) + 1
     }
```



In the above loop, S3 $\delta_<$ S2, and S3 follows S2. Therefore, cutting the loop between S2 and S3 is illegal. However, cutting the loop between S1 and S2 is legal.

```
     for I = 1 to N do {
S1:      A(I+1) = B(I) +D(I)
S2:      B(I) = (A(I) + B(I))/2
S3:      C(I) = B(I) + 1
     }
```



The above loop can be cut between S1 and S2, and also between S2 and S3