

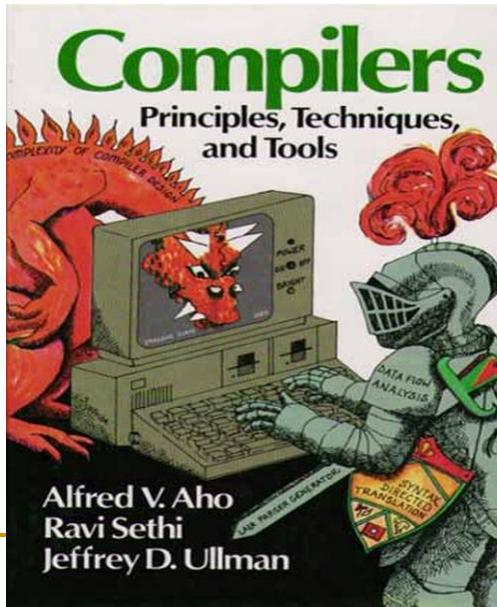
# Compiler Design

---

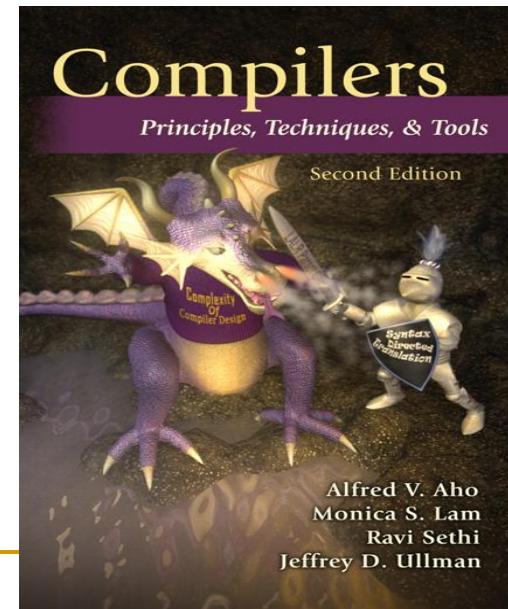
Dr. S. Suresh  
Assistant Professor  
Department of Computer Science  
Banaras Hindu University  
Varanasi – 211 005, India.

# Textbook

- Compilers: Principles, Techniques, and Tools, 2/E.
  - Alfred V. Aho, *Columbia University*
  - Monica S. Lam, *Stanford University*
  - Ravi Sethi, *Avaya Labs*
  - Jeffrey D. Ullman, *Stanford University*



Dragon



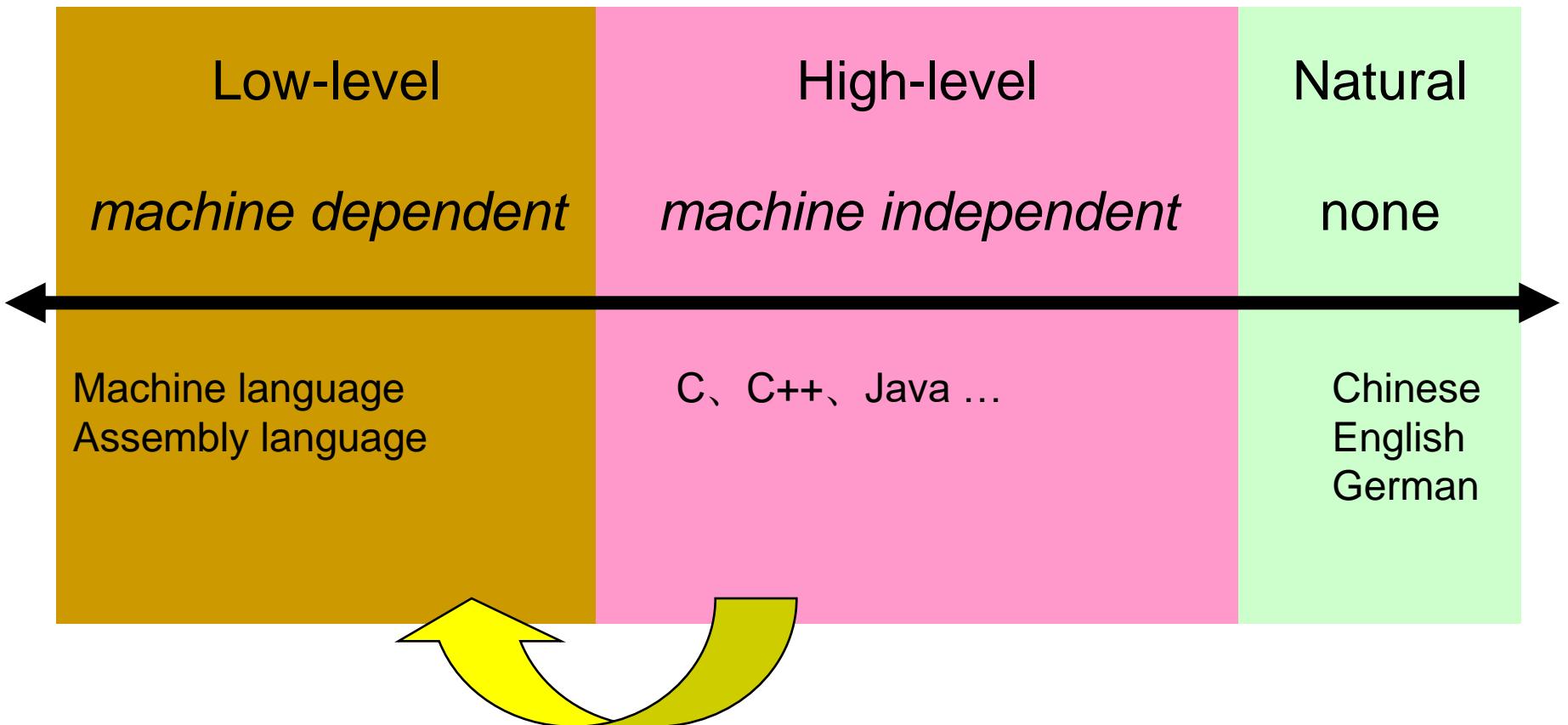
# Assessment

- Total Credits: 06 (Theory 4 credits & Lab 2 Credits)
- Theory :
  - Sessional 30% (midterm 20% + attendance 10%)
  - Final exam 70%
- Laboratory :
  - Sessional 30% (regular lab performance, assignment, report, etc.)
  - Final exam 70% (viva + lab performance)

# Objectives

- Know the various phases of compiling process
- Know how to use compiler construction tools, such as generators of scanners and parsers
- Be able to define LL(1), LR(1), and LALR(1) grammars
- Be familiar with compiler analysis and optimization techniques
- Be able to build a compiler for a (simplified) (programming) language

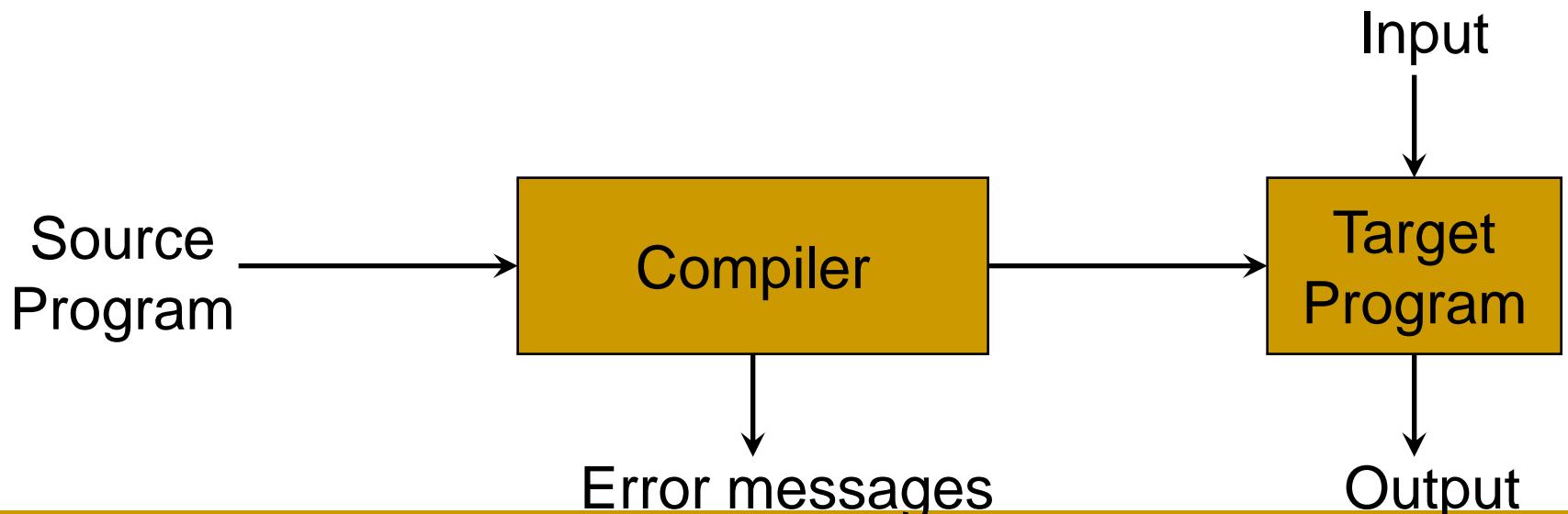
# Programming Languages



# Compilers

## ■ “*Compilation*”

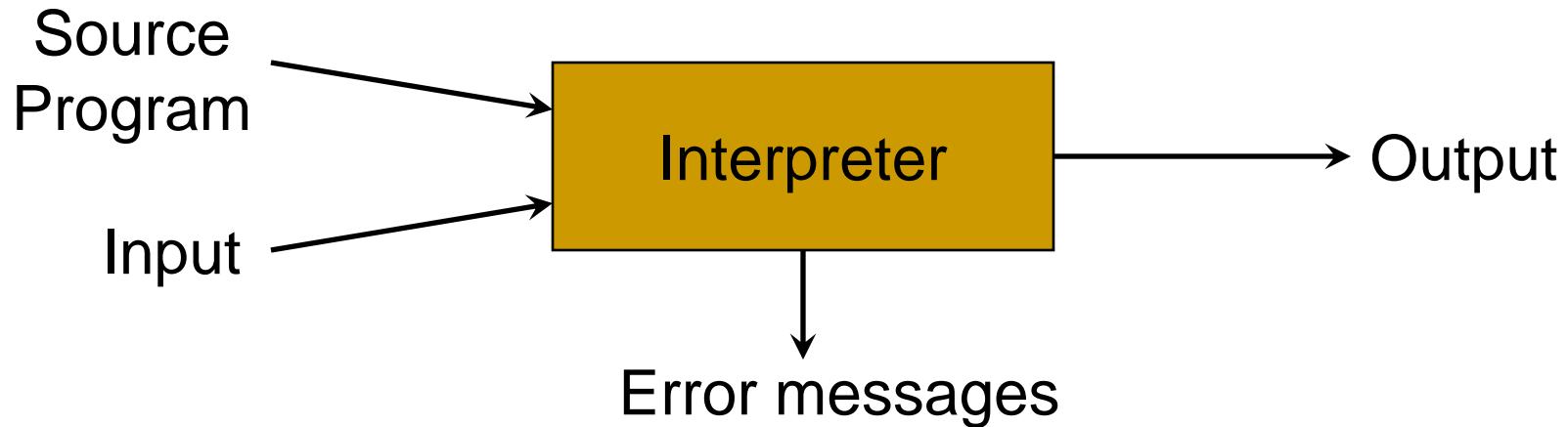
- Translation of a program written in a source language into a semantically equivalent program written in a target language



# Interpreters

## ■ “*Interpretation*”

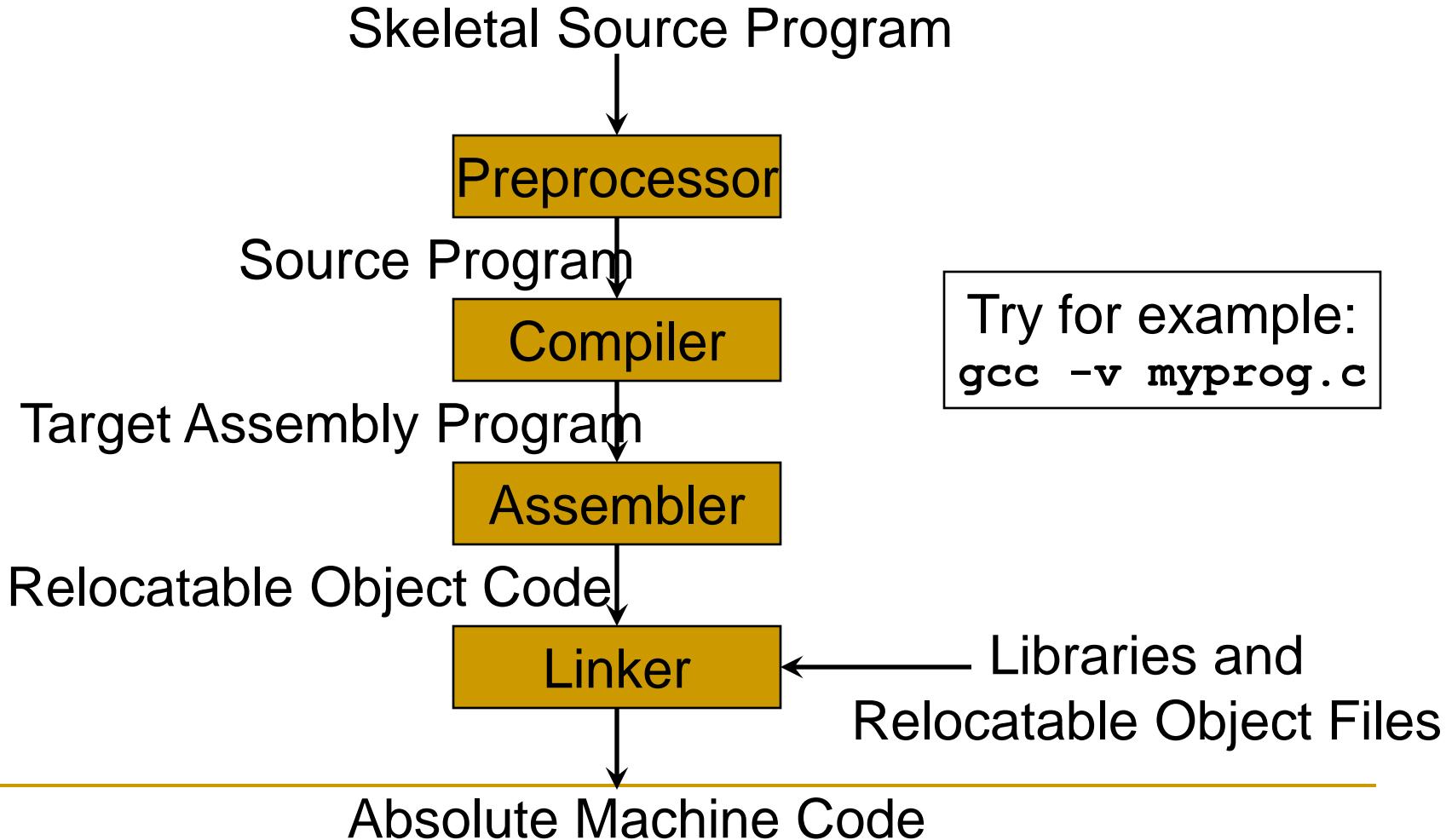
- Performing the operations implied by the source program



# Compiler vs. Interpreter

	Compiler	Interpreter
Translation method	Translates program as a whole	One statement at a time
Debugging	Harder	Easier
Intermediate code generation	Yes	No

# Preprocessors, Compilers, Assemblers, and Linkers



# The Analysis-Synthesis Model of Compilation

- There are two parts to compilation:
  - *Analysis* determines the operations implied by the source program which are recorded in a tree structure
  - *Synthesis* takes the tree structure and translates the operations therein into the target program

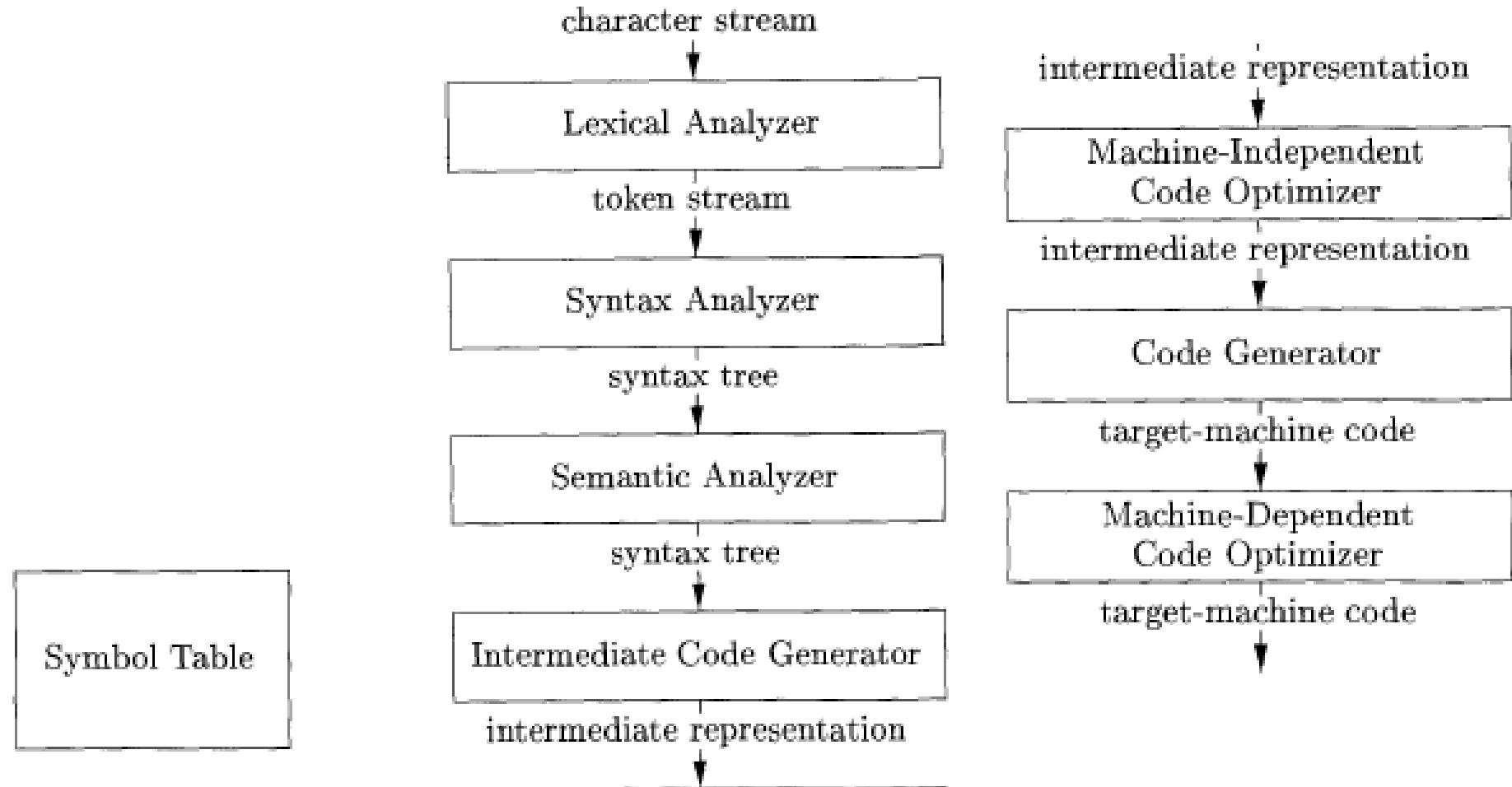
# Tools that Use the Analysis-Synthesis Model

- *Programming Languages* (C, C++...)
- *Scripts* (Javascript, bash)
- *Editors* (syntax highlighting)
- *Pretty printers* (e.g. Doxygen)
- *Static checkers* (e.g. Lint and Splint)
- *Interpreters*
- *Text formatters* (e.g. TeX and LaTeX)
- *Silicon compilers* (e.g. VHDL)
- *Query interpreters/compilers* (Databases)

# The Phases of a Compiler

Phase	Output	Sample
<i>Programmer (source code producer)</i>	Source string	<b>A=B+C;</b>
<i>Scanner (performs lexical analysis)</i>	Token string	'A', '=', 'B', '+', 'C', ';' And <i>symbol table</i> with names
<i>Parser (performs syntax analysis based on the grammar of the programming language)</i>	Parse tree or abstract syntax tree	<pre> ;     =  / \  A   +       / \       B   C     </pre>
<i>Semantic analyzer (type checking, etc)</i>	Annotated parse tree or abstract syntax tree	
<i>Intermediate code generator</i>	Three-address code, quads, or RTL	<pre> int2fp B          t1 +      t1        C      t2 :=     t2        A   </pre>
<i>Optimizer</i>	Three-address code, quads, or RTL	<pre> int2fp B          t1 +      t1        #2.3  A   </pre>
<i>Code generator</i>	Assembly code	<pre> MOVF  #2.3,r1 ADDF2 r1,r2 MOVF  r2,A   </pre>
<i>Peephole optimizer</i>	Assembly code	<pre> ADDF2 #2.3,r2 MOVF  r2,A   </pre>

# The Phases of a Compiler



# The Grouping of Phases

- Compiler *front* and *back ends*:
  - Front end: *analysis (machine independent)*
  - Back end: *synthesis (machine dependent)*
- Compiler *passes*:
  - A collection of phases is done only once (*single pass*) or multiple times (*multi pass*)
    - Single pass: usually requires everything to be defined before being used in source program
    - Multi pass: compiler may have to keep entire program representation in memory

# Compiler-Construction Tools

- Software development tools are available to implement one or more compiler phases
  - *Scanner generators*
  - *Parser generators*
  - *Syntax-directed translation engines*
  - *Automatic code generators*
  - *Data-flow engines*

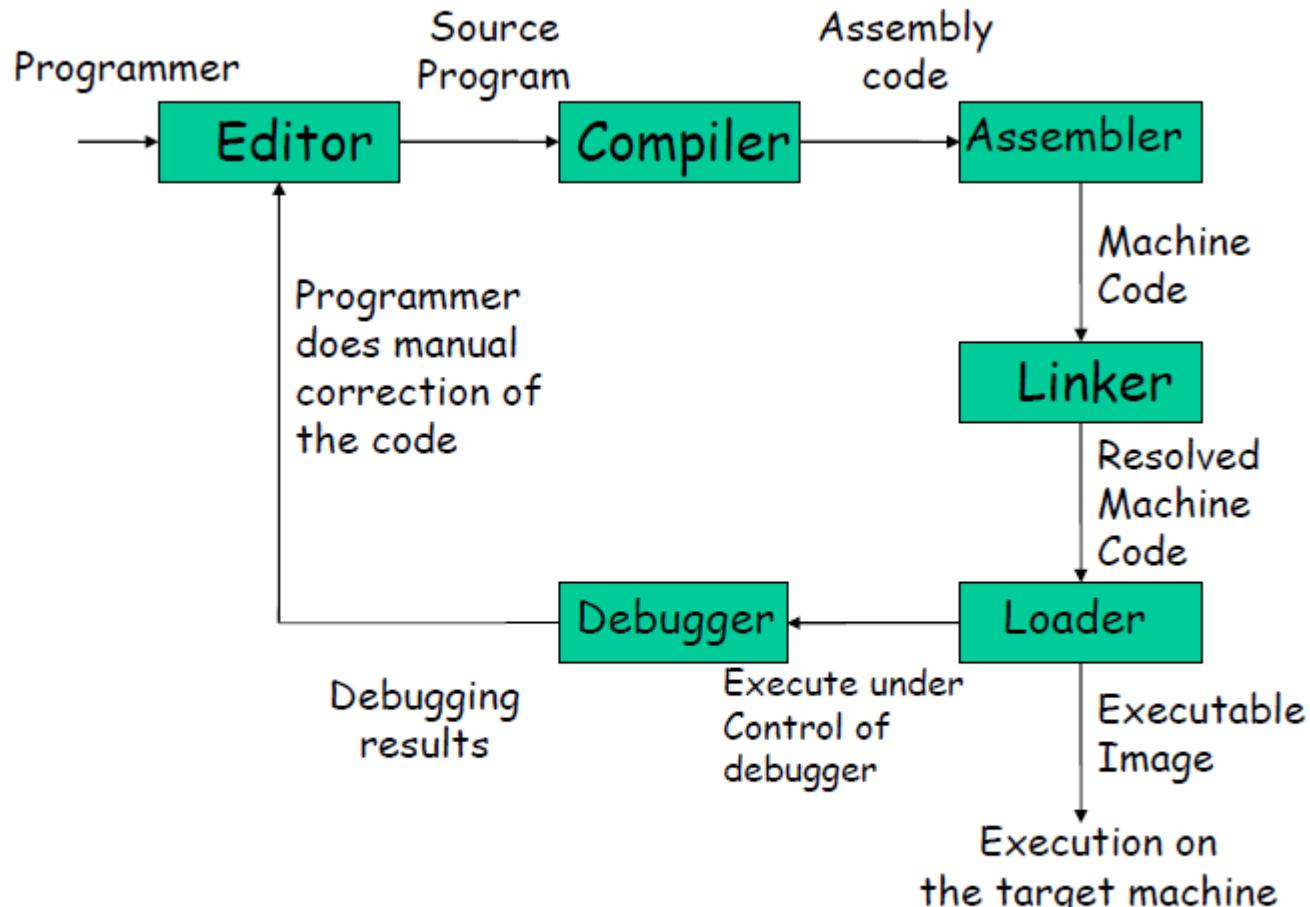
# Some early machines and implementations

- IBM developed 704 in 1954. All programming was done in assembly language. Cost of software development far exceeded cost of hardware. Low productivity.
- Speedcoding interpreter (1953): programs ran about 10 times slower than hand written assembly code.
- John Backus (in 1954): Proposed a program that translated high level expressions into native machine code. Skepticism all around. Most people thought it was impossible.
- Fortran I project (1954-1957): The first compiler was released.

# Fortran I

- The first compiler had a huge impact on the programming languages and computer science. The whole new field of compiler design was started.
- More than half the programmers were using Fortran by 1958.
- The development time was cut down to half.
- Led to enormous amount of theoretical work (lexical analysis, parsing, optimization, structured programming, code generation, error recovery etc.).
- Modern compilers preserve the basic structure of the Fortran I compiler !!!

# Program development and processing



Normally end  
up with error

# Syllabus Outline

- Introduction
- Lexical Analysis and Lex/Flex
- Syntax Analysis and Yacc/Bison
- Syntax-Directed Translation
- Type Checking
- Run-Time Environments
- Intermediate Code Generation
- Code Generation and Optimization

# Overview Of Compilers

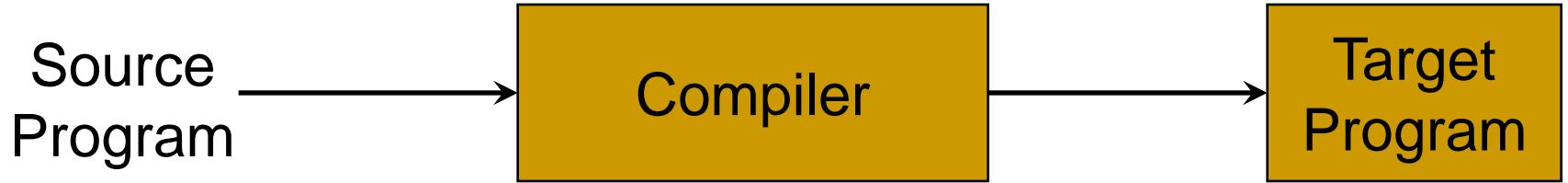
---

Dr. S. Suresh  
Assistant Professor  
Department of Computer Science  
Banaras Hindu University  
Varanasi – 211 005, India.

# What are Compilers?

- Translates from one representation of the program to another
- Typically from high level source code to low level machine code or object code
- Source code is normally optimized for human readability
  - Expressive: matches our notion of languages (and application?!)
  - Redundant to help avoid programming errors
- Machine code is optimized for hardware
  - Redundancy is reduced
  - Information about the intent is lost

# Compiler as a Translator



# Goals of translation

- Good compile time performance
- Good performance for the generated code
- Correctness
  - A very important issue.
  - Can compilers be proven to be correct?
    - Tedium even for toy compilers! Undecidable in general.
  - However, the correctness has an implication on the development cost

# How to translate?

- Direct translation is difficult. Why?
- Source code and machine code mismatch in level of abstraction
  - Variables vs Memory locations/registers
  - Functions vs jump/return
  - Parameter passing
  - structs
- Some languages are farther from machine code than others
  - For example, languages supporting Object Oriented Paradigm

# How to translate easily?

- Translate in steps. Each step handles a reasonably simple, logical, and well defined task.
- Design a series of program representations.
- Intermediate representations should be amenable to program manipulation of various kinds (type checking, optimization, code generation etc.).
- Representations become more machine specific and less language specific as the translation proceeds.

# The first few steps

- The first few steps can be understood by analogies to how humans comprehend a natural language.
- The first step is recognizing/knowing alphabets of a language. For example
  - English text consists of lower and upper case alphabets, digits, punctuations and white spaces
  - Written programs consist of characters from the ASCII characters set (normally 9-13, 32-126).

# The first few steps

- The next step to understand the sentence is recognizing words
  - How to recognize English words?
  - Words found in standard dictionaries
  - Dictionaries are updated regularly



ABOUT ▾ OXFORD GLOBAL LANGUAGES ▾ THE OED PRESS AND NEWS

December 2016 -

Around 500 new words, phrases, and senses have entered the *Oxford English Dictionary* this quarter, including *glam-ma*, *Youtuber*, and *upstander*.

We have a selection of release notes this December, each of which takes a closer look at some of our additions. The last few years have seen the emergence of the word *Brexit*, and you can read more about the huge increase in the use of the word, and how we go about defining it, in [this article](#) by Craig Leyland.

# The first few steps

- How to recognize words in a programming language?
  - a dictionary (of keywords etc.)
  - rules for constructing words (identifiers, numbers etc.)
- This is called lexical analysis
- Recognizing words is not completely trivial. For example:
  - w hat ist his se nte nce?

# Lexical Analysis: Challenges

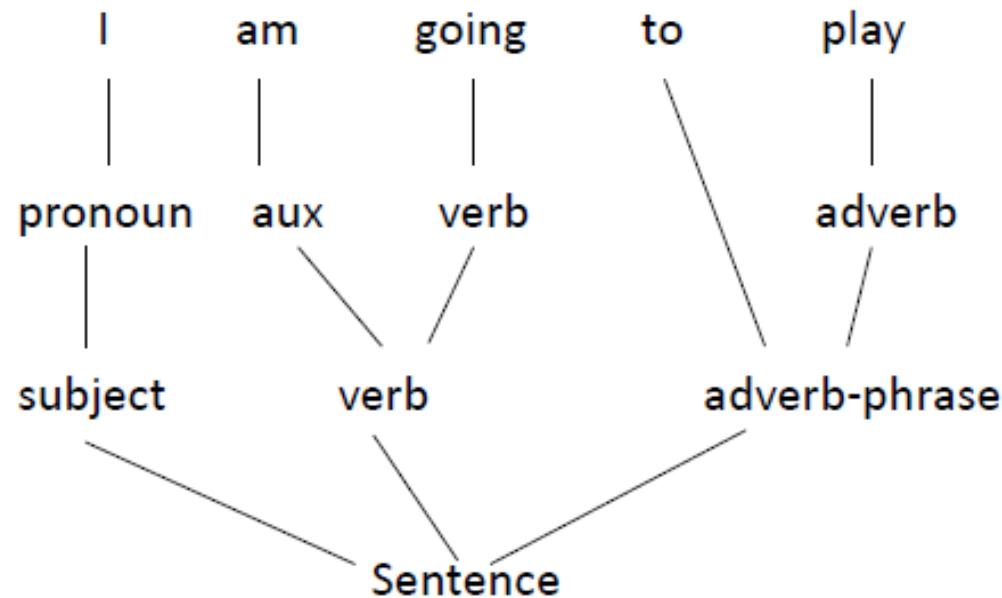
- We must know what the word separators are
- The language must define rules for breaking a sentence into a sequence of words.
- Normally white spaces and punctuations are word separators in languages.

# Lexical Analysis: Challenges

- In programming languages a character from a different class may also be treated as word separator.
- The lexical analyzer breaks a sentence into a sequence of words or tokens:
  - If a == b then a = 1 ; else a = 2 ;
  - Sequence of words (total 14 words)
    - if a == b then a = 1 ; else a = 2 ;

# The next step

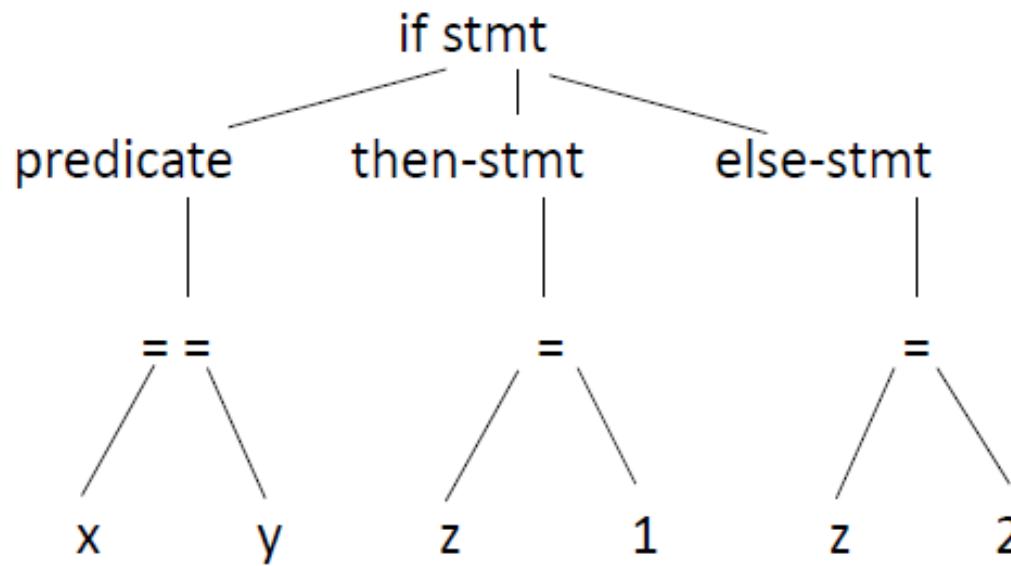
- Once the words are understood, the next step is to understand the structure of the sentence
- The process is known as *syntax checking* or *parsing*



# Parsing

- Parsing a program is exactly the same process as shown in previous slide.
- Consider an expression

**if x == y then z = 1 else z = 2**



# Understanding the meaning

- Once the sentence structure is understood we try to understand the meaning of the sentence (semantic analysis)
- A challenging task
- Example:

Prateek said Nitin left his assignment at home

- What does his refer to? Prateek or Nitin?

# Understanding the meaning

- Worse case
  - Amit said Amit left his assignment at home
- Even worse
  - Amit said Amit left Amit's assignment at home
- How many **Amits** are there? Which one left the assignment? Whose assignment got left?

# Semantic Analysis

- Too hard for compilers. They do not have capabilities similar to human understanding
- However, compilers do perform analysis to understand the meaning and catch inconsistencies
- Programming languages define strict rules to avoid such ambiguities

```
{ int Amit = 3;  
    { int Amit = 4;  
        cout << Amit;  
    }  
}
```

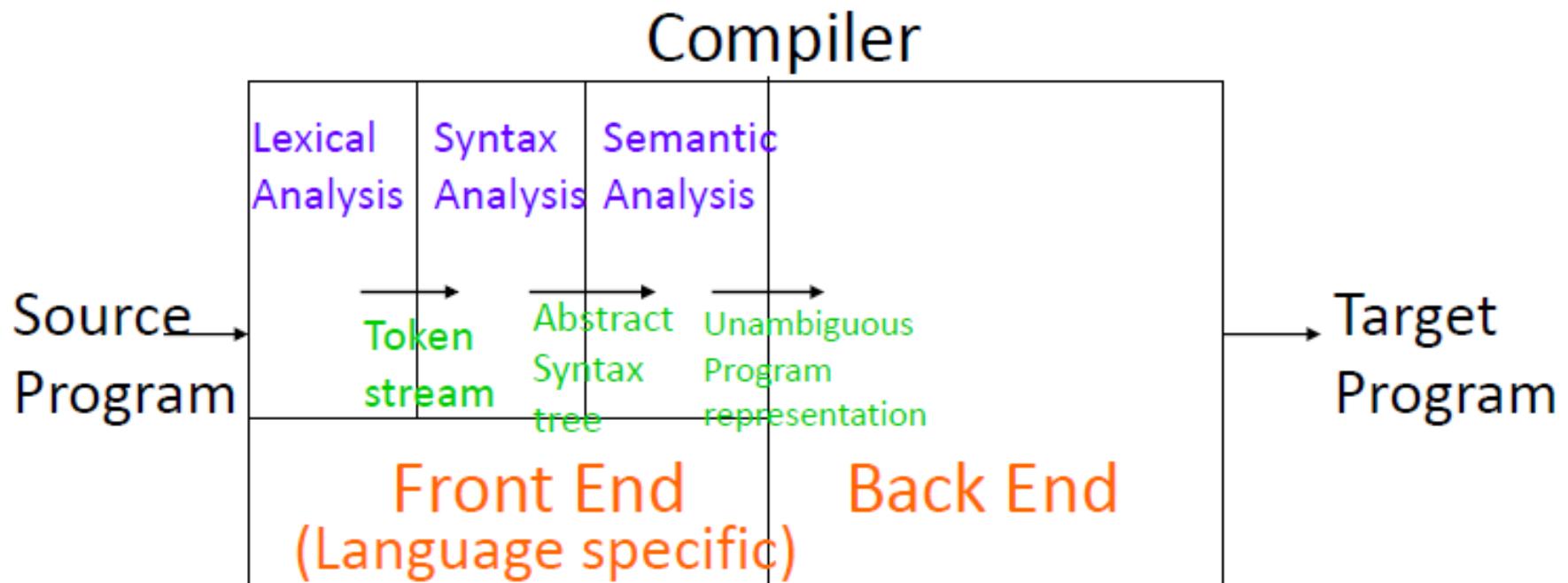
# More on Semantic Analysis

- Compilers perform many other checks besides variable bindings
- Type checking
  - Amit left her work at home
- There is a type mismatch between **her** and **Amit**. Presumably **Amit** is a male. And they are not the same person.

## Example from Mahabharat

- अश्वथामा हतः इतत नरो वा कुंजरो वा
- “Ashwathama hathaha iti, narova kunjarova”
- Ashwathama is dead. But, I am not certain whether it was a human or an elephant

# Compiler structure once again



# Back End



# Code Optimization

- No strong counter part with English, but is similar to editing/précis writing
- Automatically modify programs so that they
  - Run faster
  - Use less resources (memory, registers, space, fewer fetches etc.)

# Code Optimization

- Some common optimizations
  - Common sub-expression elimination
  - Copy propagation
  - Dead code elimination
  - Code motion
  - Strength reduction
  - Constant folding
- Example:  $x = 15 * 3$  is transformed to  $x = 45$

# Example of Optimizations

A : assignment   M : multiplication   D : division   E : exponent

PI = 3.14159

Area = 4 \* PI \* R^2

Volume = (4/3) \* PI \* R^3

3A+4M+1D+2E

---

X = 3.14159 \* R \* R

Area = 4 \* X

Volume = 1.33 \* X \* R

3A+5M

---

Area = 4 \* 3.14159 \* R \* R

Volume = ( Area / 3 ) \* R

2A+4M+1D

---

Area = 12.56636 \* R \* R

Volume = ( Area / 3 ) \* R

2A+3M+1D

---

X = R \* R

Area = 12.56636 \* X

Volume = 4.18879 \* X \* R

3A+4M

# Code Generation

- Usually a two step process
  - Generate intermediate code from the semantic representation of the program
  - Generate machine code from the intermediate code
- The advantage is that each phase is simple
- Requires design of intermediate language
- Most compilers perform translation between successive intermediate representations
- Intermediate languages are generally ordered in decreasing level of abstraction from highest (source) to lowest (machine)

# Code Generation

- Abstractions at the source level  
identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level  
memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems
- Code generation is mapping from source level abstractions to target machine abstractions

# Code Generation

- Map identifiers to locations (memory/storage allocation)
- Explicate variable accesses (change identifier reference to relocatable/absolute address)
- Map source operators to opcodes or a sequence of opcodes

# Code Generation

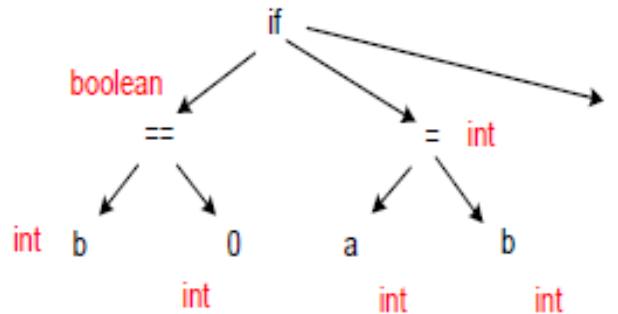
- Convert conditionals and iterations to a test/jump or compare instructions
- Layout parameter passing protocols: locations for parameters, return values, layout of activations frame etc.
- Interface calls to library, runtime system, operating systems

# Post translation Optimizations

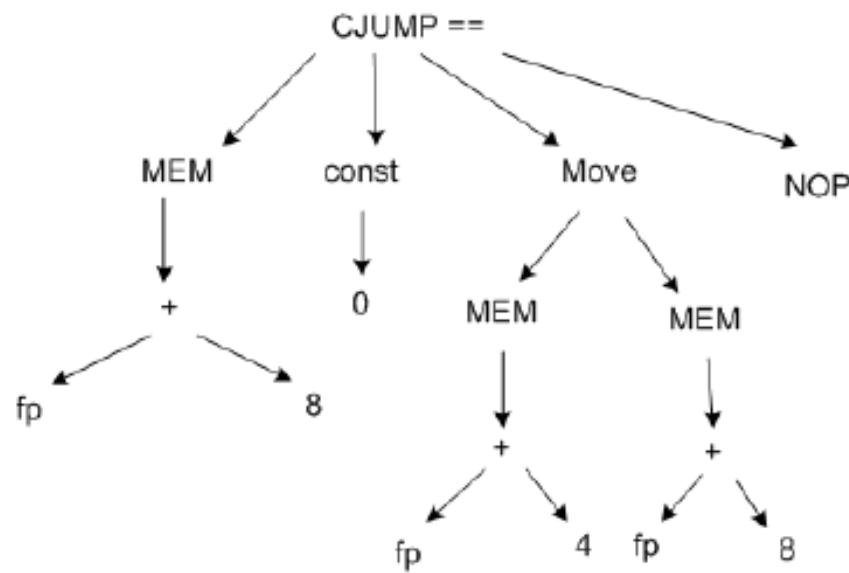
- Algebraic transformations and reordering
  - Remove/simplify operations like
    - Multiplication by 1
    - Multiplication by 0
    - Addition with 0
  - Reorder instructions based on
    - Commutative properties of operators
    - For example  $x+y$  is same as  $y+x$  (always?)

# Post translation Optimizations

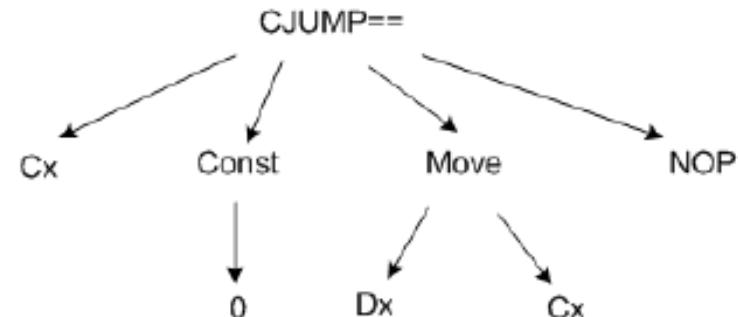
- Instruction selection
  - Addressing mode selection
  - Opcode selection
  - Peephole optimization
    - kind of optimization performed over a very small set of instructions, called a "peephole"



## Intermediate code generation



## Optimization



## Code Generation

**CMP Cx, 0**  
**CMOVZ Dx,Cx**

# References

- Compiler Design by Amey Karkare, IIT Kanpur  
<https://karkare.github.io/cs335/>
- Compilers: Principles, Techniques, and Tools, Second edition, 2006. by Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman

# Lexical Analysis

---

Dr. S. Suresh  
Assistant Professor  
Department of Computer Science  
Banaras Hindu University  
Varanasi – 211 005, India.

# Lexical Analyzer

- Reads the input characters of the source program, group them into lexemes and produce as output a sequence of tokens for each lexeme in the source program.
- Interacts with symbol table.
- Stripping out the comments and whitespaces.
- Correlates the error messages generated by the compiler with source program.
- Models with regular expressions.
- Recognizes using Finite State Automata.

# Lexemes, Tokens and Patterns

- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token.
- It is identified by the lexical analyzer as an instance of that token.

# Lexemes, Tokens and Patterns

- A **token** is a pair consisting of a token name and an optional attribute value.
- The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or sequence of input characters denoting an identifier.
- The token names are the input symbols that the parser processes.
- Token form <token-name, attribute-value>
  - Token-name is an abstract symbol
  - attribute-value points to an entry in symbol table (describes the lexeme represented by the token)

# Lexemes, Tokens and Patterns

- A **pattern** is a description of the form that the lexemes of a token may take.
- In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword.
- For identifiers and some other tokens, the pattern is more complex structure that is matched by many strings.

# Lexemes, Tokens and Patterns

- For Example,

position = initial + rate \* 60

- Lexemes are

position, =, initial, rate, \*, 60

- Tokens are

`<id,1> <=> <id,2> <+> <id, 3> <*> <60>`

# Lexical Analysis

- Sentences consist of string of tokens (a syntactic category)
- For example, number, identifier, keyword, string
- Task: Identify Tokens and corresponding Lexemes

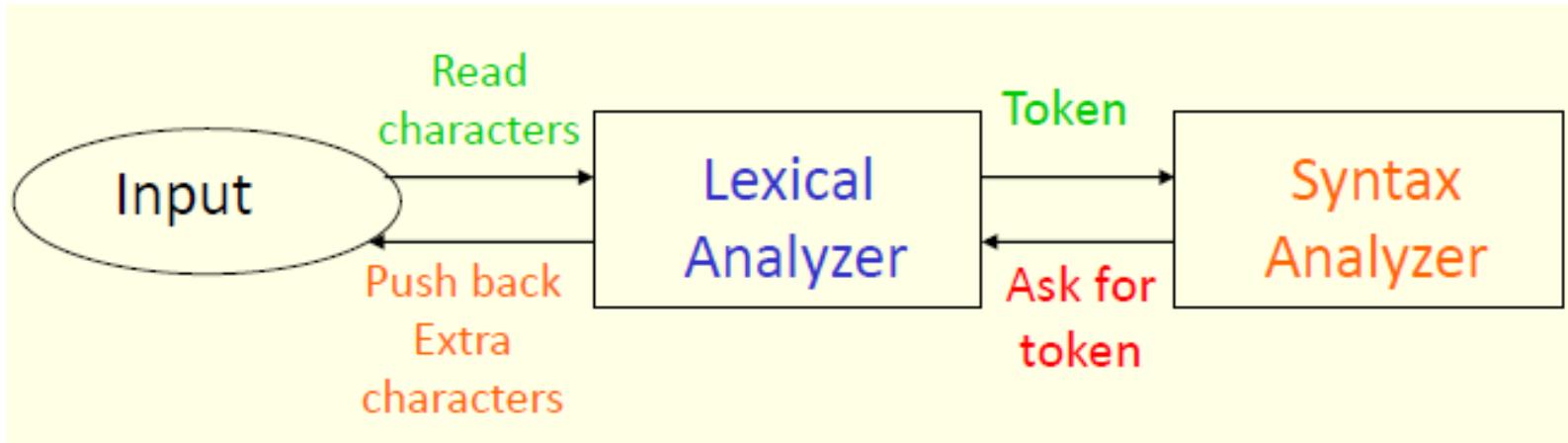
# Types of tokens

- In many programming languages, the following classes covers most or all of the tokens:
- One token for each keyword. The pattern for a keyword is the same as the keyword itself.
- Tokens for the operators, either individually or in classes.
- One token representing all identifiers.
- One or more tokens representing constants, such as numbers and literal strings.
- Tokens for each punctuation symbol, such as left and right parenthesis, comma and semicolon.

# Approaches to implementation

- **Use assembly language**  
Most efficient but most difficult to implement
- **Use high level languages like C**  
Efficient but difficult to implement
- **Use tools like lex, flex**  
Easy to implement but not as efficient as the first two cases

# Interface to other phases



- Why do we need Push back?
- Required due to look-ahead
  - for example, to recognize  $\geq$  and  $>$
- Typically implemented through a buffer
  - Keep input in a buffer
  - Move pointers over the input

# Construct a lexical analyzer

- Allow white spaces, numbers and arithmetic operators in an expression
- Return tokens and attributes to the syntax analyzer
- A global variable `tokenval` is set to the value of the number
- Design requires that
  - A finite set of tokens be defined
  - Describe strings belonging to each token

# Problems

- Scans text character by character
- Look ahead character determines what kind of token to read and when the current token ends
- First character cannot determine what kind of token we are going to read

# Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - Insert(s,t): save lexeme s and token t and return pointer
  - Lookup(s): return index of entry for lexeme s or 0 if s is not found

# Implementation of Symbol Table

- Fixed amount of space to store lexemes.
  - Not advisable as it waste space.
- Store lexemes in a separate array.
  - Each lexeme is separated by eos.
  - Symbol table has pointers to lexemes.

Usually 32 bytes

Fixed space for lexemes	Other attributes

Usually 4 bytes

	Other attributes

lexeme1 eos lexeme2 eos lexeme3 .....

# How to handle keywords?

- Consider token DIV and MOD with lexemes div and mod.
- Initialize symbol table with insert( “div” , DIV ) and insert( “mod” , MOD).
- Any subsequent insert fails (unguarded Insert)
- Any subsequent lookup returns a nonzero value, therefore, cannot be used as identifier.

# Difficulties in design of lexical analyzers

- Is it as simple as it sounds?
- Lexemes in a fixed position. Fix format vs. free format languages
- FORTRAN Fixed Format
  - 80 columns per line
  - Column 1-5 for the statement number/label column
  - Column 6 for continuation mark (?)
  - Column 7-72 for the program statements
  - Column 73-80 Ignored (Used for other purpose)
  - Letter C in Column 1 meant the current line is a comment

# Difficulties in design of lexical analyzers

- Handling of blanks
  - in Pascal and C, blanks separate identifiers
  - in Fortran blanks are important only in literal strings  
for example variable **counter** is same as **count er**
  - Another example
    - DO 10 I = 1.25                    DO10I=1.25
    - DO 10 I = 1,25                    DO10I=1,25

- ❑ The first line is variable assignment  
DO10I=1.25
- ❑ Second line is beginning of a  
Do loop
- ❑ Reading from left to right one can not distinguish  
between the two until the ";" or "." is reached
- Fortran white space and fixed format rules  
came into force due to punch cards and  
errors in punching



# Punched card from a Fortran program

		DO 10 I=1,3																			
		STATEMENT NO.	CONT.																		
		FORTAN		STATEMENT		IDENTIFICATION															
		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
		2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
		4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	IBM	1154	PRINTED	A1	FORTAN CODE CARD																
					5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
					6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
					7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
					8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
					9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
		21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
		41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
		61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80

# PL/1 Problems

- Keywords are not reserved in PL/1
  - if then then then = else else else = then
  - if if then then = then + 1
- PL/1 declarations
  - Declare(arg<sub>1</sub>,arg<sub>2</sub>,arg<sub>3</sub>,.....,arg<sub>n</sub>)
- Can not tell whether **Declare** is a keyword or array reference until after “)”
- Requires **arbitrary lookahead** and **very large buffers**. Worse, the buffers may have to be reloaded.

# Problem continues even today!!

- C++ template syntax: Foo<Bar>
- C++ stream syntax: cin >> var;
- Nested templates: Foo<Bar<Bazz>>
- Can these problems be resolved by lexical analyzers alone?

# How to specify tokens?

- How to describe tokens

2.e0      20.e-01      2.000

- How to break text into token

```
if (x==0) a = x << 1;  
iff (x==0) a = x < 1;
```

- How to break input into token efficiently
  - Tokens may have similar prefixes
  - Each character should be looked at only once

# How to describe tokens?

- Programming language tokens can be described by regular languages
- Regular languages
  - Are easy to understand
  - There is a well understood and useful theory
  - They have efficient implementation
- Regular languages are discussed in great detail in the “Theory of Computation” course

# Strings and Languages

- An *alphabet* is any finite set of symbols.
- Example
  - $\{0, 1\}$  is the binary alphabet
  - ASCII, Unicode are the important examples
- A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet.
- Sentence, word and strings are Interchangeably used in language theory.
- Length of the string  $s$ , denoted by  $|s|$ , is a no. of occurrences of symbols in the string.
- The *empty string*, denoted  $\epsilon$ , is the string of length zero.

# Strings and Languages

- A *language* is any countable set of strings over some fixed alphabet.
- The language alphabet  $\{\epsilon\}$  contains only the empty string.
- Definition of language does not require that any meaning be associated to the strings in the language.

# Operations on languages

- Union

- $L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$

- Concatenation

- $LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$

- Closure

- Kleene closure  $L^* = \text{Union of } L^i \text{ such that } 0 \leq i \leq \infty,$   
where  $L^0 = \epsilon$  and  $L^i = L^{i-1}L$

- Positive closure  $L^+ = \text{Union of } L^i \text{ such that } 1 \leq i \leq \infty,$   
i.e.,  $L^+ = L^*$  without  $\epsilon$

# Example

- Let  $L = \{a, b, \dots, z\}$  and  $D = \{0, 1, 2, \dots, 9\}$  then
- LUD is set of letters and digits
- LD is set of strings consisting of a letter followed by a digit
- $L^*$  is a set of all strings of letters including  $\epsilon$
- $L(LUD)^*$  is set of all strings of letters and digits beginning with a letter
- $D^+$  is the set of strings of one or more digits

# Notation

- Let  $\Sigma$  be a set of characters. A language over  $\Sigma$  is a set of strings of characters belonging to  $\Sigma$
- A regular expression  $r$  denotes a language  $L(r)$
- Rules that define the regular expressions over  $\Sigma$ 
  - $\epsilon$  is a regular expression that denotes  $\{\epsilon\}$  the set containing the empty string
  - If  $a$  is a symbol in  $\Sigma$  then  $a$  is a regular expression that denotes  $\{a\}$

- If  $r$  and  $s$  are regular expressions denoting the languages  $L(r)$  and  $L(s)$  then
- $(r)|(s)$  is a regular expression denoting  $L(r) \cup L(s)$
- $(r)(s)$  is a regular expression denoting  $L(r)L(s)$
- $(r)^*$  is a regular expression denoting  $(L(r))^*$
- $(r)$  is a regular expression denoting  $L(r)$

- Let  $\Sigma = \{a, b\}$
- The regular expression  $a|b$  denotes the set  $\{a, b\}$
- The regular expression  $(a|b)(a|b)$  denotes  $\{aa, ab, ba, bb\}$
- The regular expression  $a^*$  denotes the set of all strings  $\{\epsilon, a, aa, aaa, \dots\}$
- The regular expression  $(a|b)^*$  denotes the set of all strings containing  $\epsilon$  and all strings of a's and b's
- The regular expression  $a|a^*b$  denotes the set containing the string  $a$  and all strings consisting of zero or more a's followed by a b

- Precedence and associativity
- \*, concatenation, and | are left associative
- \* has the highest precedence
- Concatenation has the second highest precedence
- | has the lowest precedence

# How to specify tokens

## ■ Regular definitions

- For notational convenience, names may be given to the regular expressions and use those names in subsequent expressions
- The names themselves can be used as symbols in the subsequent expressions

# How to specify tokens

## ■ Regular definitions

- Let  $r_i$  be a regular expression and  $d_i$  be a distinct name and  $\Sigma$  is an alphabet of basic symbols

- Regular definition is a sequence of definitions of the form

$$d_1 \rightarrow r_1$$

$$d_2 \rightarrow r_2$$

.....

$$d_n \rightarrow r_n$$

- Where  $d_i$  is a new symbol, not in  $\Sigma$  and not same as any other  $d$ 's, and

- each  $r_i$  is a regular expression over  $\Sigma \cup \{d_1, d_2, \dots, d_{i-1}\}$

# Examples

- My fax number  
91-(512)-259-7586
- $\Sigma = \text{digits} \cup \{-, (, )\}$
- Country  $\rightarrow \text{digit}^+$       digit<sup>2</sup>
- Area  $\rightarrow ( \text{digit}^+ )$       digit<sup>3</sup>
- Exchange  $\rightarrow \text{digit}^+$       digit<sup>3</sup>
- Phone  $\rightarrow \text{digit}^+$       digit<sup>4</sup>
- Number  $\rightarrow \text{country - area - exchange - phone}$

# Examples ...

- My email address

`csdept@bhu.ac.in`

- $\Sigma = \text{letter} \cup \{@, .\}$

- Letter  $\rightarrow a | b | \dots | z | A | B | \dots | Z$

- Name  $\rightarrow \text{letter}^*$

- Address  $\rightarrow \text{name} @ \text{name} . \text{Name} . \text{name}$

# Examples ...

- Identifier

letter  $\rightarrow$  a| b| ...|z| A| B| ...| Z | \_

digit  $\rightarrow$  0| 1| ...| 9

identifier  $\rightarrow$  letter(letter|digit)\*

- Unsigned number in Pascal

digit  $\rightarrow$  0| 1| ...|9

digits  $\rightarrow$  digit+ or digit digit\*

fraction  $\rightarrow$  . digits | ε

exponent  $\rightarrow$  (E ( + | - | ε) digits) | ε

number  $\rightarrow$  digits fraction exponent

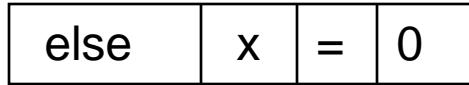
# Regular expressions in specifications

- Regular expressions describe many useful languages
- Regular expressions are only specifications; implementation is still required
- Given a string  $s$  and a regular expression  $R$ , does  $s \in L(R)$  ?
- Solution to this problem is the basis of the lexical analyzers
- However, just the yes/no answer is not important
- Goal: Partition the input into tokens

1. Write a regular expression for lexemes of each token
  - number  $\rightarrow$  digit<sup>+</sup>
  - identifier  $\rightarrow$  letter(letter|digit)<sup>+</sup>
2. Construct R matching all lexemes of all tokens
  - $R = R_1 + R_2 + R_3 + \dots$
3. Let input be  $x_1 \dots x_n$ 
  - for  $1 \leq i \leq n$  check  $x_1 \dots x_i \in L(R)$
4.  $x_1 \dots x_i \in L(R) \Rightarrow x_1 \dots x_i \in L(R_j)$  for some j
  - smallest such j is token class of  $x_1 \dots x_i$
5. Remove  $x_1 \dots x_i$  from input; go to (3)

- The algorithm gives priority to tokens listed earlier
  - Treats “if” as keyword and not identifier
- How much input is used? What if
  - $x_1 \dots x_i \in L(R)$
  - $x_1 \dots x_j \in L(R)$
  - Pick up the longest possible string in  $L(R)$
  - The principle of “maximal munch”
- Regular expressions provide a concise and useful notation for string patterns
- Good algorithms require single pass over the input

# How to break up text

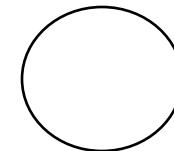
- Elsex=0  
- Regular expressions alone are not enough
- Normally longest match wins
- Ties are resolved by prioritizing tokens
- Lexical definitions consist of regular definitions, priority rules and maximal munch or longest match principle

# Finite Automata

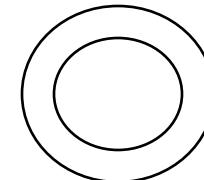
- Regular expression are declarative specifications
- Finite automata is implementation
- A finite automata consists of
  - An input alphabet belonging to  $\Sigma$
  - A set of states  $S$
  - A set of transitions  $state_i \xrightarrow{input} state_j$
  - A set of final states  $F$
  - A start state  $n$
- Transition  $s1 \xrightarrow{a} s2$  is read:  
in state  $s1$  on input  $a$  go to state  $s2$
- If end of input is reached in a final state then accept
- Otherwise, reject

# Pictorial notation

- A state



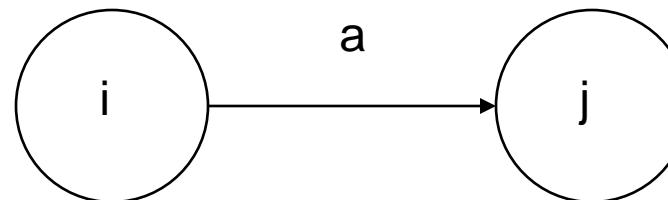
- A final state



- Transition



- Transition from state  $i$  to state  $j$  on input  $a$



# How to recognize tokens

- Consider

relOp → < | <= | = | <> | >= | >

id → letter(letter|digit)\*

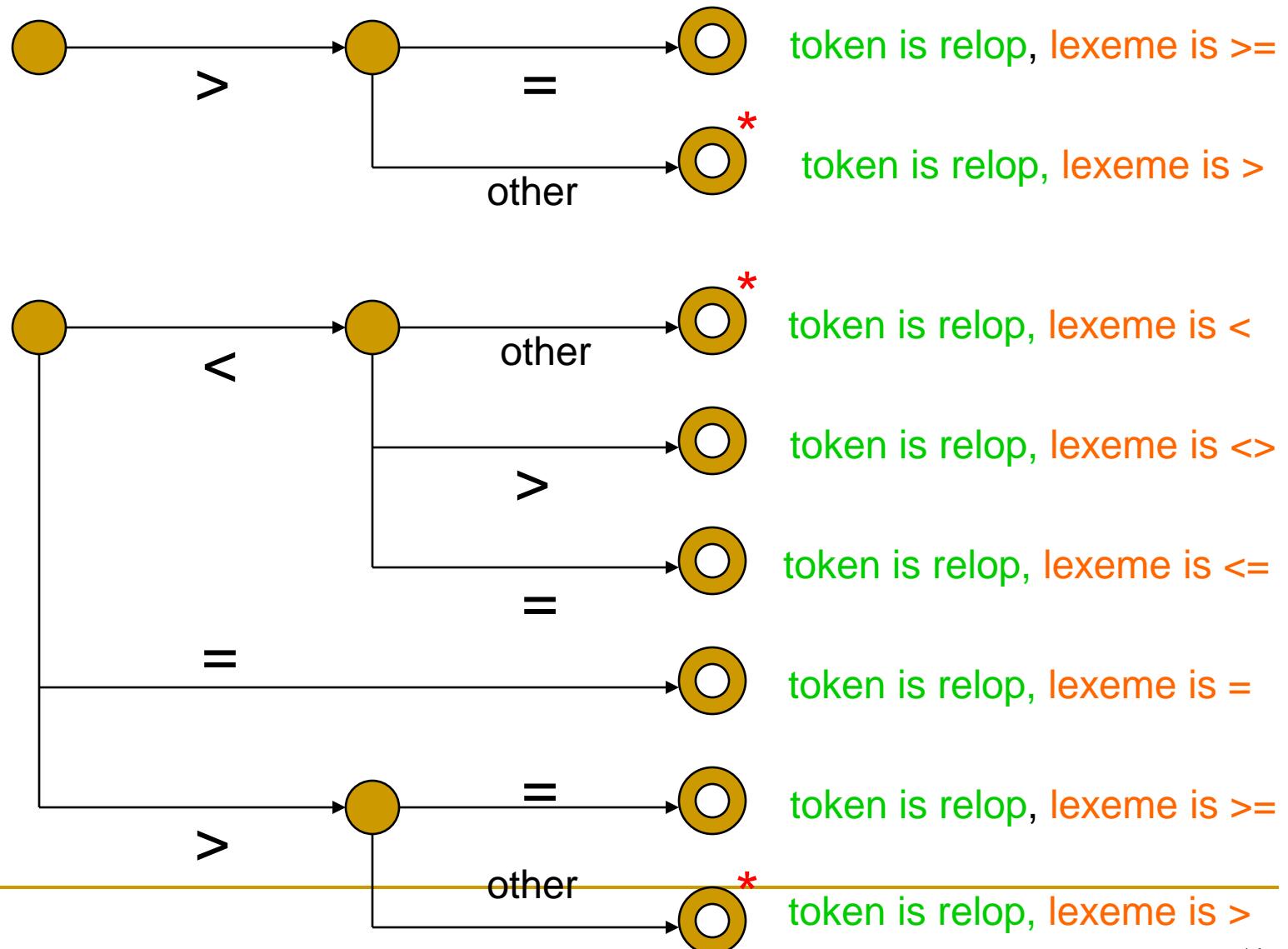
num → digit+ ('.' digit+)? (E('+'|'-'))? digit+)?

delim → blank | tab | newline

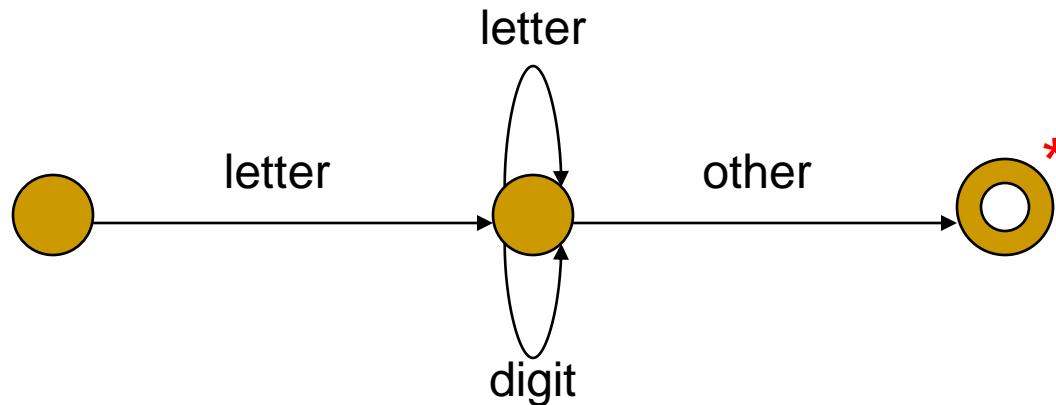
ws → delim<sup>+</sup>

- Construct an analyzer that will return  
<token, attribute> pairs

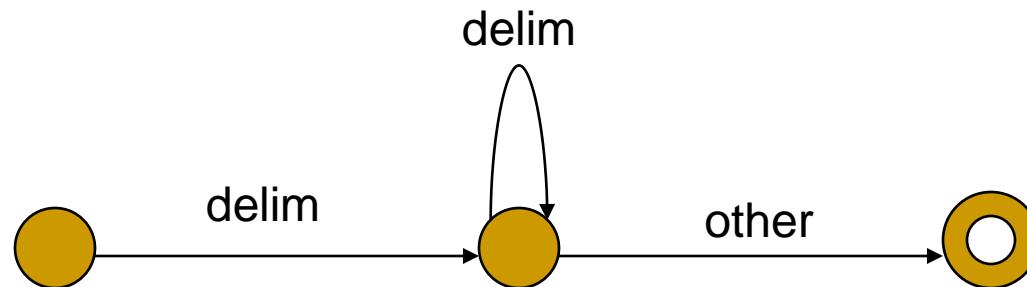
# Transition diagram for relops



## Transition diagram for identifier

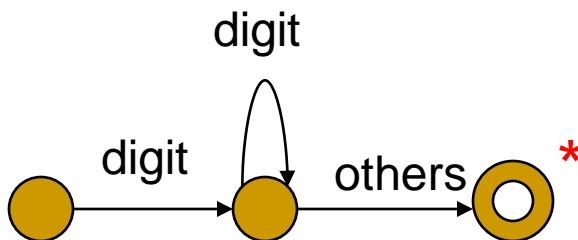
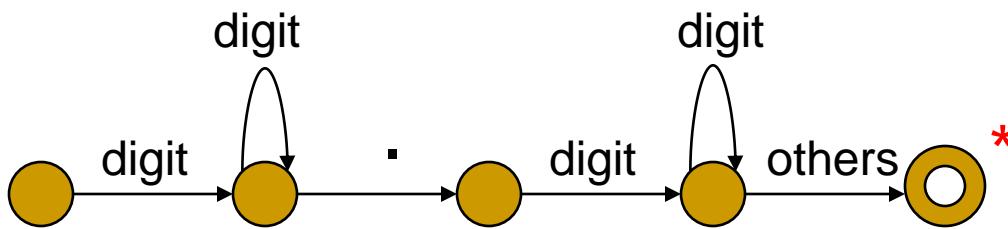
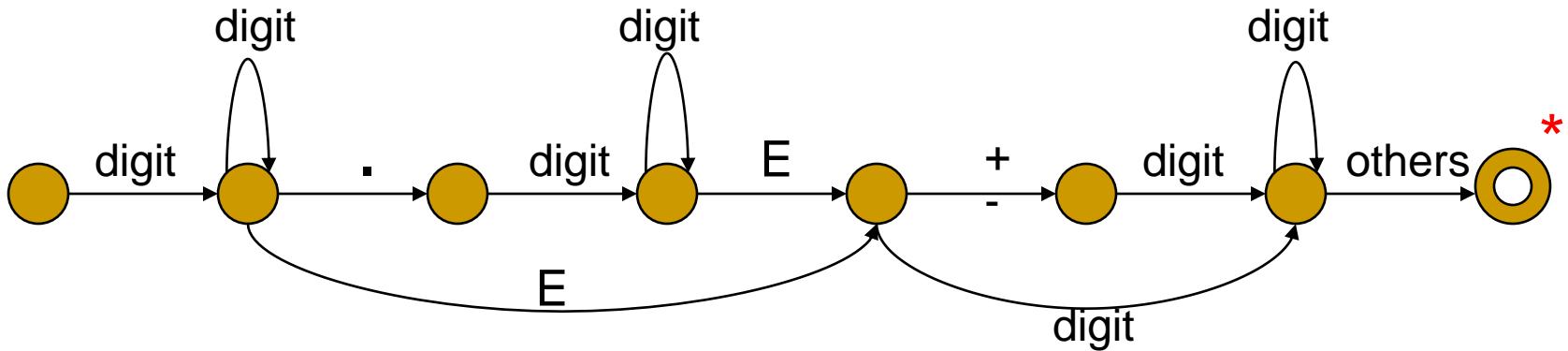


## Transition diagram for white spaces



\* Indicates that, we must retract the input one position

# Transition diagram for unsigned numbers



- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- Starting in the third diagram the accept state will be reached after 12
- Therefore, the matching should always start with the first transition diagram
- If failure occurs in one transition diagram then retract the forward pointer to the start state and activate the next diagram
- If failure occurs in all diagrams then a lexical error has occurred

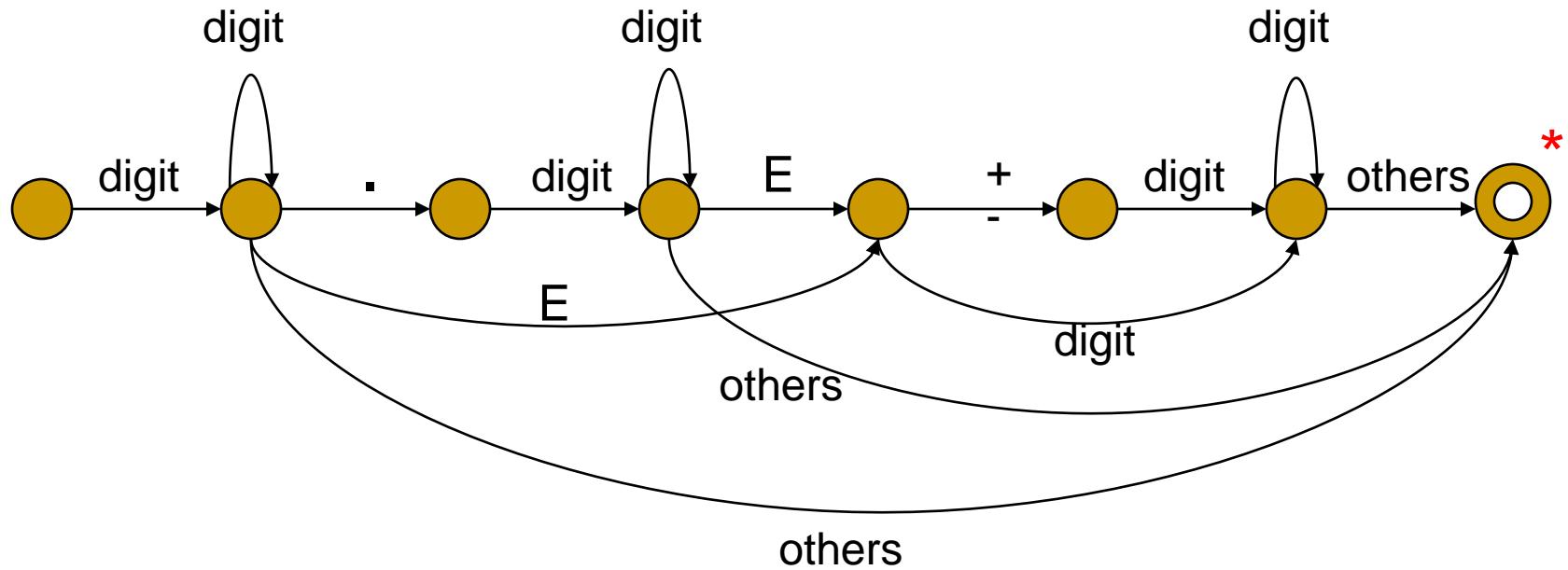
# Implementation of transition diagrams

```
Token nexttoken() {
    while(1) {
        switch (state) {

            .....
            case 10: c=nextchar();
                if(isletter(c)) state=10;
                elseif (isdigit(c)) state=10;
                else state=11;
                break;

            .....
        }
    }
}
```

# Another transition diagram for unsigned numbers



A more complex transition diagram  
is difficult to implement and  
may give rise to errors during coding

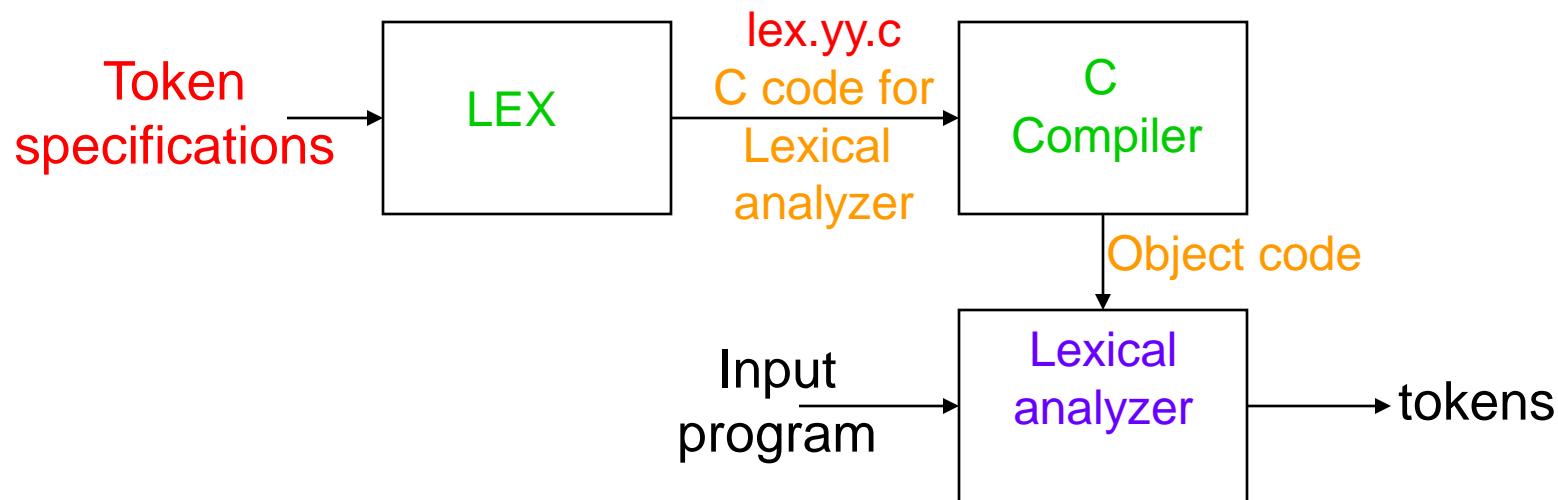
# Lexical analyzer generator - Lex

- Lex is a tool for Lexical analyzer generator
- Recent implement is Flex
- Allows to specify a Lexical analyzer by specifying regular expressions
- The input notation for the Lex is referred as the Lex language and the tool is Lex compiler.
- The Lex compiler transforms the input patterns into transition diagram and generates code.

# Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression  
(generates kind of token and other book keeping information)
  
- Output of the generator
  - Program that reads input character stream and breaks that into tokens
  - Reports lexical errors (unexpected characters)

# LEX: A lexical analyzer generator



Refer to LEX User's Manual

# Structure of Lex Programs

- A Lex program has the following form:

declarations

%%

translation rules

%%

auxiliary functions

# Example

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any ws	—	—
if	if	—
then	then	—
else	else	—
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

# Example

```
%{  
    /* definitions of manifest constants  
    LT, LE, EQ, NE, GT, GE,  
    IF, THEN, ELSE, ID, NUMBER, RELOP */  
}  
  
/* regular definitions */  
delim      [ \t\n]  
ws         {delim}+  
letter     [A-Za-z]  
digit      [0-9]  
id         {letter}{(letter}|{digit})*  
number     {digit}+({.}{digit}+)?(E[+-]?)?{digit}+)?  
  
%%  
  
{ws}      /* no action and no return */  
if        {return(IF);}  
then      {return(THEN);}  
else      {return(ELSE);}  
{id}       {yyval = (int) installID(); return(ID);}  
{number}   {yyval = (int) installNum(); return(NUMBER);}  
"<"       {yyval = LT; return(RELOP);}  
"<="      {yyval = LE; return(RELOP);}  
"=="      {yyval = EQ; return(RELOP);}  
"<>"     {yyval = NE; return(RELOP);}  
">"      {yyval = GT; return(RELOP);}  
">="      {yyval = GE; return(RELOP);}  
  
%%  
  
int installID() /* function to install the lexeme, whose  
                  first character is pointed to by yytext,  
                  and whose length is yyleng, into the  
                  symbol table and return a pointer  
                  thereto */  
}  
  
int installNum() /* similar to installID, but puts numer-  
                  ical constants into a separate table */  
}
```

# How does LEX work?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non deterministic finite automaton (NFA)
- Convert the NFA into equivalent DFA
- Minimize DFA to reduce number of states
- Emit code driven by DFA tables

# References

- Compiler Design by Amey Karkare, IIT Kanpur  
<https://karkare.github.io/cs335/>
- Compilers: Principles, Techniques, and Tools, Second edition, 2006. by Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman

# Syntax Analysis

---

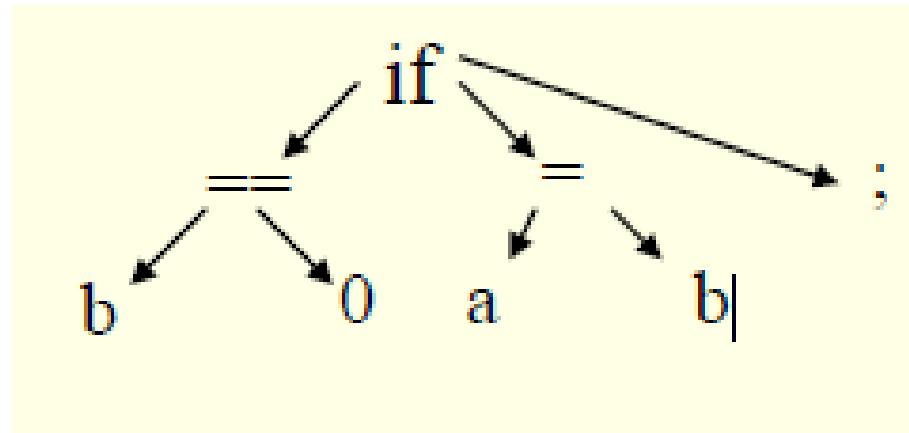
Dr. S. Suresh  
Assistant Professor  
Department of Computer Science  
Banaras Hindu University  
Varanasi – 211 005, India.

# Syntax Analyzer

- Input: Sequence of Tokens
- Output: a representation of program
  - Often AST, but could be other things
- Error reporting and recovery
- Model using context free grammars
- Recognize using Push down automata/Table Driven Parsers

# Syntax Analyzer

if	(	b	==	0	)	a	=	b	;
----	---	---	----	---	---	---	---	---	---



- Check syntax and construct abstract syntax tree and Error reporting

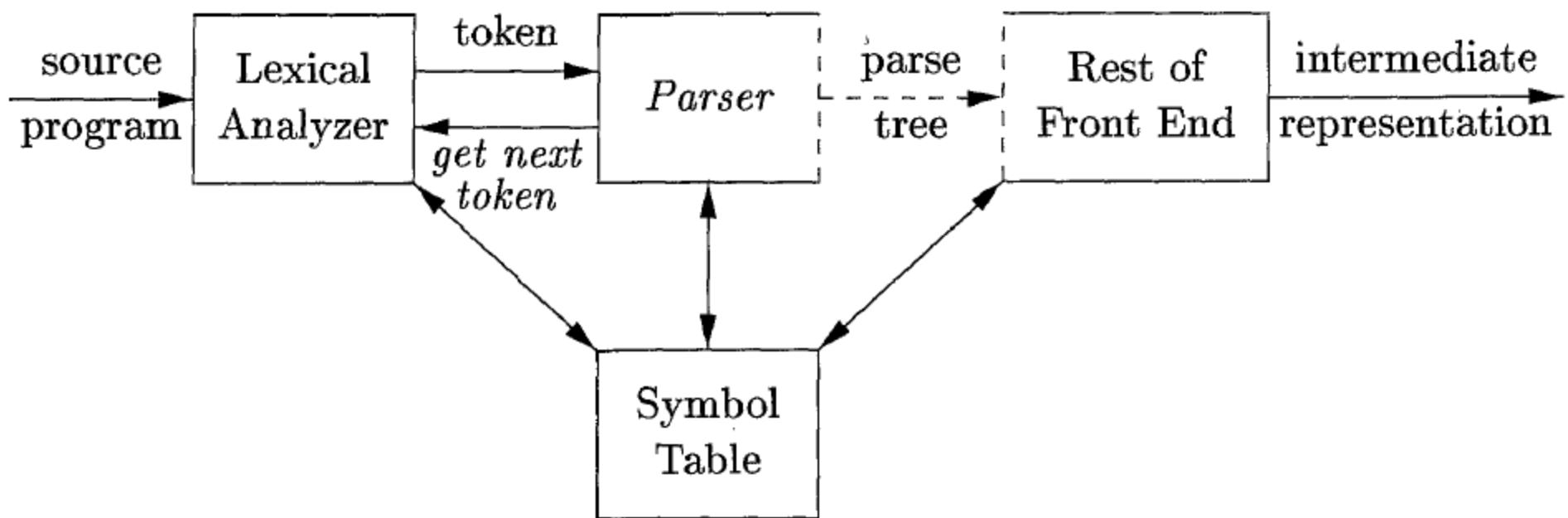
# Benefits of Grammars

- Offers benefit for both language designers and compiler writers.
- Gives a precise, yet easy-to-understand, syntactic specification of a language
- For certain classes of grammars, parser can be constructed automatically
- Parser construction process can reveal syntactic ambiguities and trouble spots in the initial design of a language
- Useful for translating source programs into correct object code and for detecting errors
- Allows a language can to be evolved or developed iteratively

# The role of the Parser

- Obtains a string of tokens from lexical analyzer
- Verifies that the string of token names can be generated by the grammar for the source language and constructs the parse tree
- Report syntax errors
- Recovers from commonly occurring errors to continue processing the remainder of the program
- Parsers can be combined with other phases of front end since they interacts often.

# Position of parser in compiler model



# Types of Parsers

- Universal
  - Cocke-Younger-Kasami algorithms
  - Earley's algorithm
  - Can parse any grammar
  - Too inefficient to use in production compilers
- Top-down
  - Builds parse tree from the top (root) to the bottom (leaves)
- Bottom-up
  - Builds parse tree from the bottom (leaves) to the top (root)
- Both top-down and bottom-up case, input is scanned from left to right, one symbol at a time

# Limitations of regular languages

- How to describe language syntax precisely and conveniently. Can regular expressions be used?
- Many languages are not regular, for example, string of balanced parentheses
  - (((...))))
  - $\{ (i)^i \mid i \geq 0 \}$
  - There is no regular expression for this language
- A finite automata may repeat states, however, it cannot remember the number of times it has been to a particular state
- A more powerful language is needed to describe a valid string of tokens

# Context Free Grammars (CFGs)

- Context free grammars  $\langle T, N, P, S \rangle$ 
  - $T$ : a set of **tokens** (terminal symbols)
  - $N$ : a set of **nonterminal** symbols
  - $P$ : a set of **productions** or **rule** of the form
    - nonterminal  $\rightarrow$  String of terminals & non terminals
  - $S$ : a **start** symbol
- A grammar derives strings by beginning with a start symbol and repeatedly replacing a nonterminal by the right hand side of a production for that non terminal.
- The strings that can be derived from the start symbol of a grammar  $G$  form the language  $L(G)$  defined by the grammar.

# Context Free Grammars (CFGs)

- *Terminals* are the basic symbols from which strings are formed.
- “token name” is a synonym for “terminal”
- *Nonterminals* are syntactic variables that denote the sets of strings
- *Nonterminals* impose a hierarchical structure on language that is key to syntax analysis and translation
- One *nonterminal* is distinguished as start symbol and the set of strings it denotes is the language generated by the grammar

# Notational Conventions

- Terminals
  - lowercase letters,
  - operators symbols (eg. +, -, \*, etc.),
  - digits,
  - punctuation symbols, etc.
- Nonterminals
  - uppercase letters,
  - letters S for start symbol

# Examples

- Grammar for arithmetic expressions
  - terminals are  $id + - * / ()$
  - nonterminals are ***expression, term*** and ***factor***
  - start symbol is ***expression***
- Grammar

$expression \rightarrow expression + term$

$expression \rightarrow expression - term$

$expression \rightarrow term$

$term \rightarrow term * factor$

$term \rightarrow term / factor$

$term \rightarrow factor$

$factor \rightarrow (expression)$

$factor \rightarrow id$

# Examples

- String of balanced parentheses

$$S \rightarrow ( S ) S | \epsilon$$

- Grammar

$$\text{list} \rightarrow \text{list} + \text{digit}$$

$$| \text{list} - \text{digit}$$

$$| \text{digit}$$

$$\text{digit} \rightarrow 0 | 1 | \dots | 9$$

- Consists of the language which is a list of digit separated by + or -.

# Examples

list  $\rightarrow \underline{\text{list}} + \text{digit}$

$\rightarrow \underline{\text{list}} - \text{digit} + \text{digit}$

$\rightarrow \underline{\text{digit}} - \text{digit} + \text{digit}$

$\rightarrow 9 - \underline{\text{digit}} + \text{digit}$

$\rightarrow 9 - 5 + \underline{\text{digit}}$

$\rightarrow 9 - 5 + 2$

- Therefore, the string 9-5+2 belongs to the language specified by the grammar
- The name context free comes from the fact that use of a production  $X \rightarrow \dots$  does not depend on the context of X

# Examples

## ■ Simplified Grammar for C block

block → '{'decls statements'}

statements → stmt-list |  $\epsilon$

stmt-list → stmt-list stmt ;

| stmt ;

decls → decls declaration |  $\epsilon$

declaration → ...

# Syntax analyzers

- Testing for membership whether  $w$  belongs to  $L(G)$  is just a “yes” or “no” answer
- However the syntax analyzer
  - Must generate the parse tree
  - Handle errors gracefully if string is not in the language
- Form of the grammar is important
  - Many grammars generate the same language
  - Tools are sensitive to the grammar

# What syntax analysis cannot do!

- To check whether variables are of types on which operations are allowed
- To check whether a variable has been declared before use
- To check whether a variable has been initialized
- These issues will be handled in semantic analysis

# Derivation

- If there is a production  $A \rightarrow \alpha$  then we say that  $A$  derives  $\alpha$  and is denoted by  $A \Rightarrow \alpha$
- $\Rightarrow$  means “derives in one step”
- $\alpha A \beta \Rightarrow \alpha y \beta$  if  $A \Rightarrow y$  is a production
- If  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$  rewrites  $\alpha_1$  to  $\alpha_n$   
we say that  $\alpha_1$  derives  $\alpha_n$  (derives in zero or more steps)
- $\Rightarrow^*$  means “derives in one or more steps”
- $\Rightarrow^+$  means “derives in one or more steps”

# Derivation

- Thus  $\alpha \Rightarrow^* \alpha$ , for any string  $\alpha$ , and
- If  $\alpha \Rightarrow^* \beta$  and  $\beta \Rightarrow \delta$ , then  $\alpha \Rightarrow^* \delta$
- Given a grammar  $G$  and a string  $w$  of terminals in  $L(G)$  we can write  $S \Rightarrow^+ w$
- If  $S \Rightarrow^* \alpha$  where  $\alpha$  is a string of terminals and non terminals of  $G$  then we say that  $\alpha$  is a **sentential form** of  $G$
- Sentential form may contain both terminal and nonterminals and may be empty

# Derivation

- Consider the following grammar

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid id$$

- The string  $- ( id + id )$  is a sentence of grammar because

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id)$$

- $E, -E, -(E), -(E + E), -(id + E), -(id + id)$  are the sentential form of this grammar

# Derivation

- If in a sentential form only the leftmost non terminal is replaced then it becomes **leftmost derivation**
- Every leftmost step can be written as

$$wA\gamma \Rightarrow^{lm^*} w\delta\gamma$$

where  $w$  is a string of terminals,  $A \rightarrow \delta$  is a production and  $\gamma$  is a string of grammar symbols

- Similarly, **rightmost derivation**, **left-sentential** and **right-sentential** are also can be defined
- An **ambiguous** grammar is one that produces more than one leftmost(rightmost) derivation of a sentence

# Parse Tree

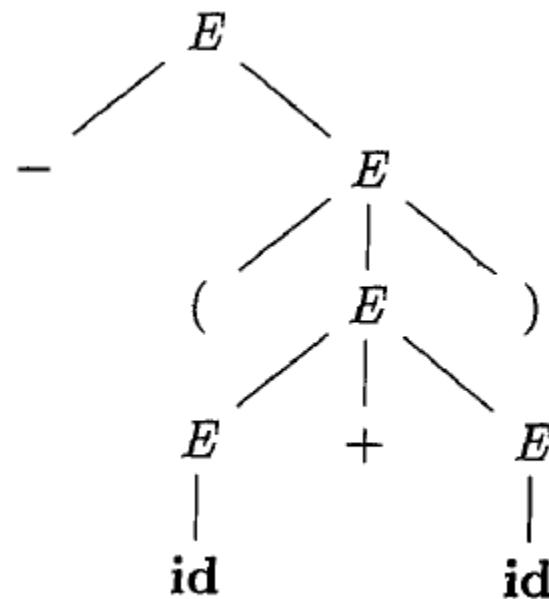
- It is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals
- Shows how the start symbol of a grammar derives a string in the language
- Root is labeled by the start symbol
- Leaf nodes are labeled by tokens
- Each internal node is labeled by a non terminal

# Parse Tree

- If  $A$  is the label of a node and  $x_1, x_2, \dots, x_n$  are labels of the children of that node then  $A \rightarrow x_1 x_2 \dots x_n$  is a production in the grammar
- To see the relationship between derivations and parse trees, consider any derivations  $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$  where  $\alpha_1$  is a single nonterminal  $A$ .
- For each sentential form  $\alpha_i$  in a derivation, we can construct a parse tree whose yield is  $\alpha_i$
- The process is an induction on  $i$ .

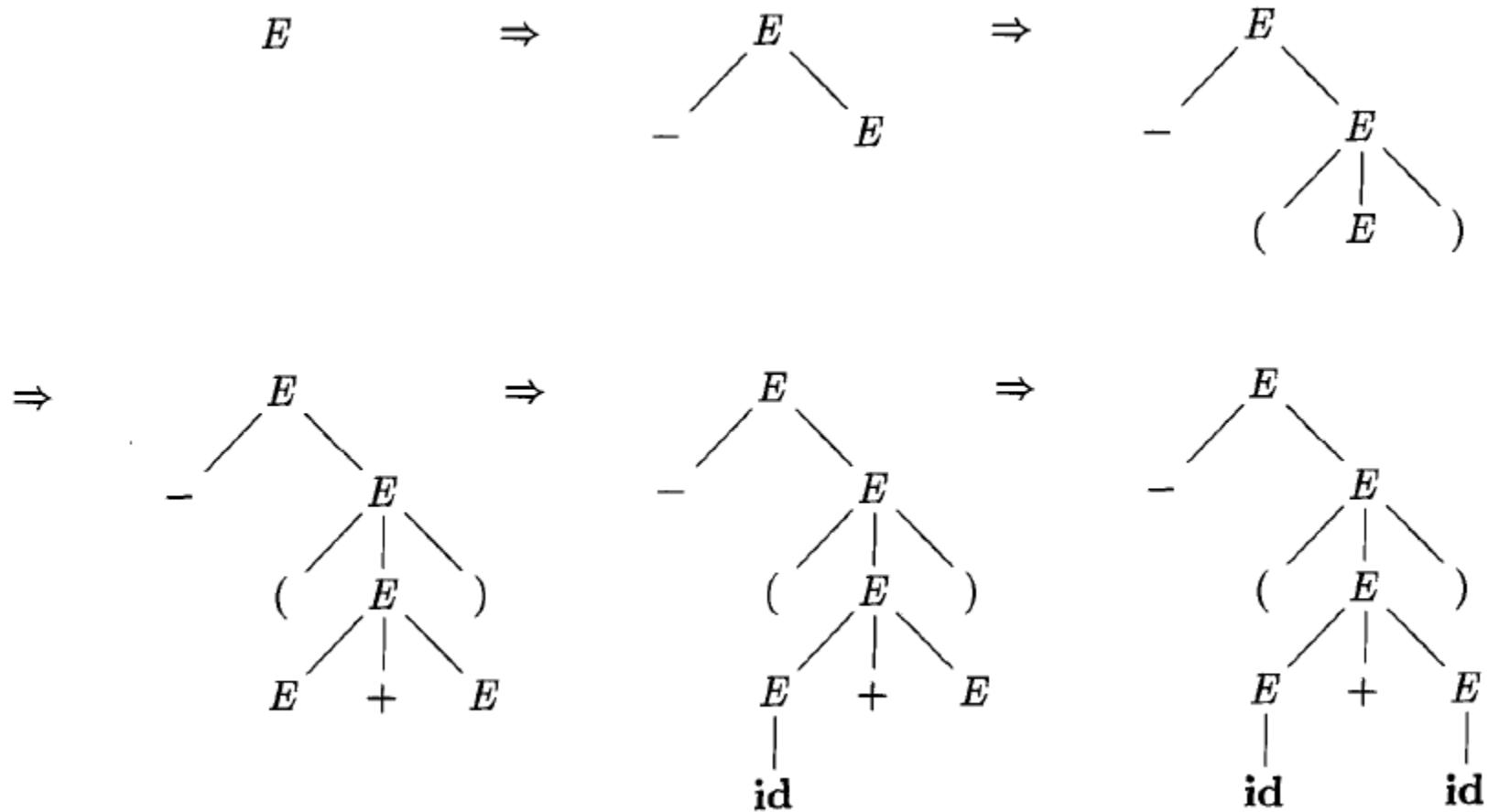
# Example

- Parse tree for  $- ( id + id )$



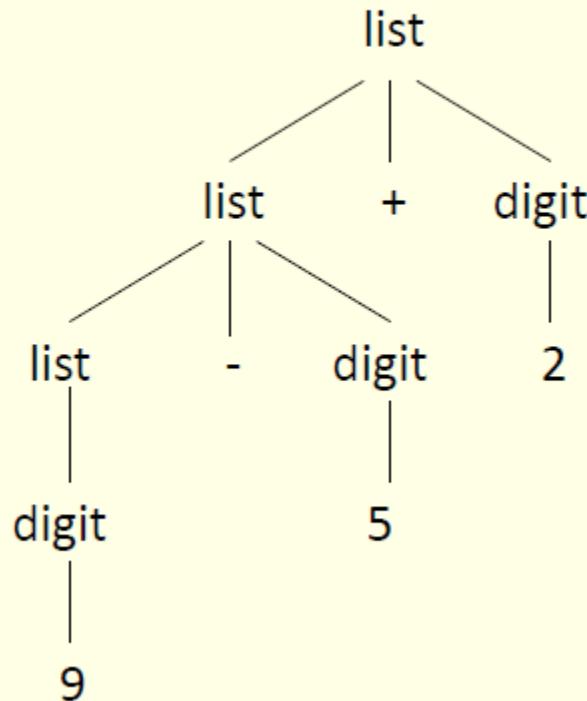
# Example

## ■ Sequence of parse trees



# Example

Parse tree for  $9-5+2$

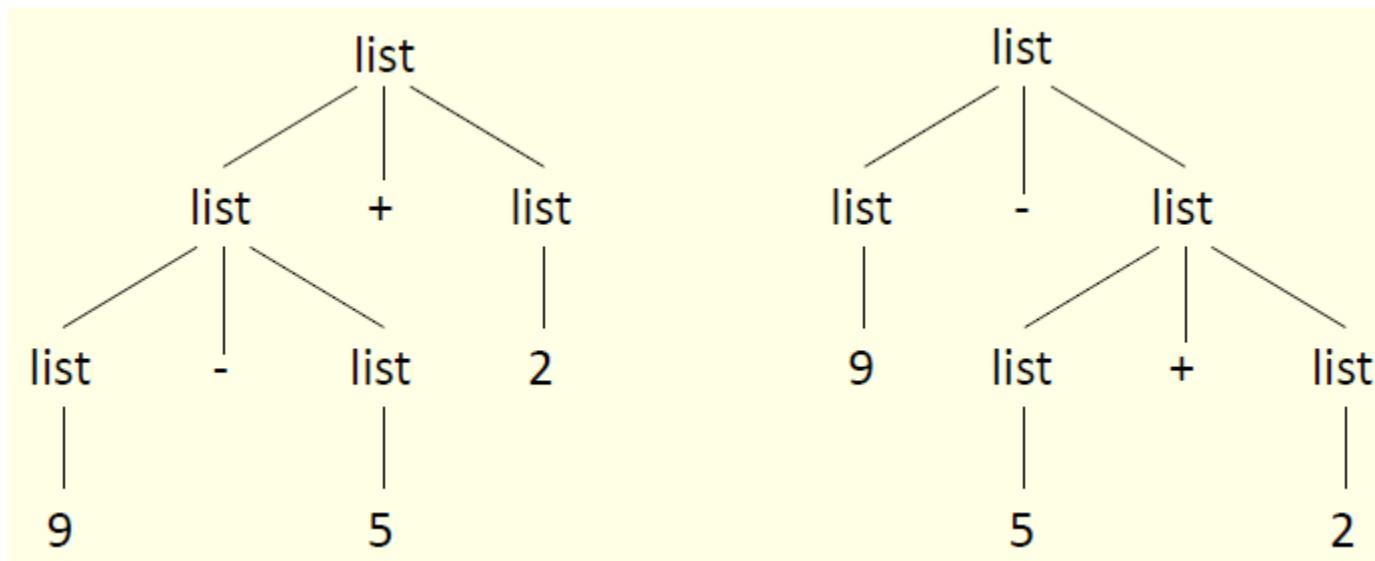


# Ambiguity

- A Grammar that produces more than one parse tree for some sentence is said ambiguous.
- Consider grammar
  - list  $\rightarrow$  list+ list
  - | list–list
  - | 0 | 1 | ... | 9
- String 9-5+2 has two parse trees

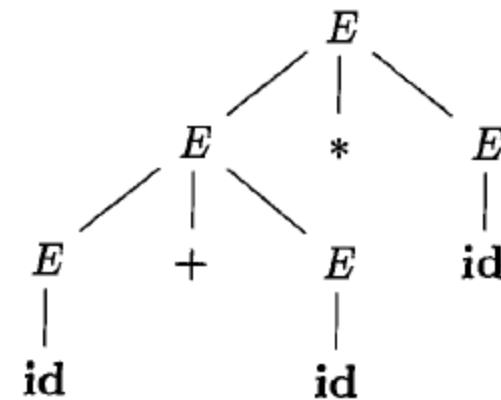
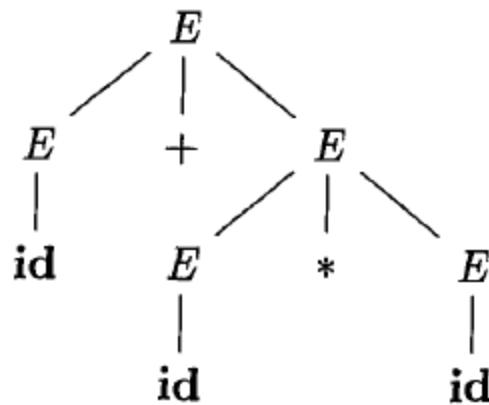
# Example

- String 9-5+2 has two parse trees



# Example

- String `id + id * id` has two parse trees



# Ambiguity ...

- Ambiguity is problematic because meaning of the programs can be incorrect
- Ambiguity can be handled in several ways
  - Enforce associativity and precedence
  - Rewrite the grammar (cleanest way)
- There is no algorithm to convert automatically any ambiguous grammar to an unambiguous grammar accepting the same language
- Worse, there are inherently ambiguous languages!

# Ambiguity ...

- For most parsers, it is desirable that the grammar be made unambiguous
- Otherwise, we can't uniquely determine the parse tree
- However, it is convenient to use carefully chosen ambiguous grammars with disambiguating rules that throw away undesirable parse trees, leaving only one tree for each sentence

# Ambiguity in Programming Lang.

- Dangling else problem

`stmt → if expr stmt`

| `if expr stmt`  
| `else stmt`  
| `other`

`other` means any other statement

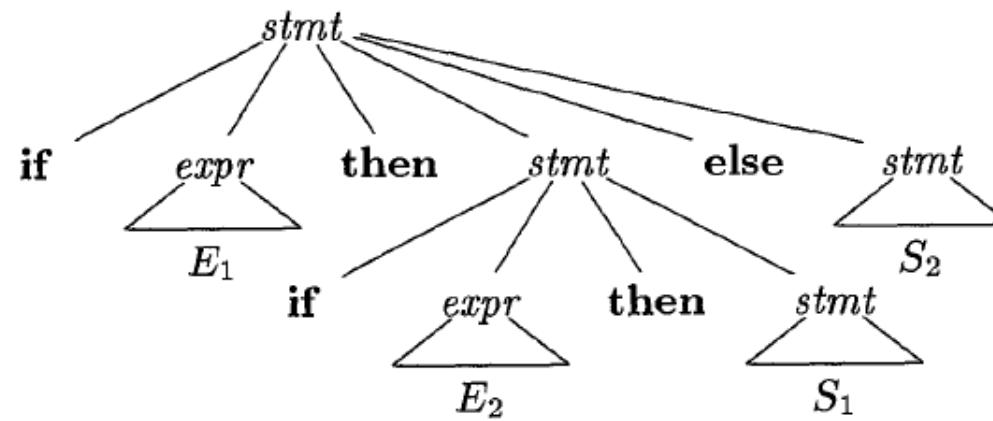
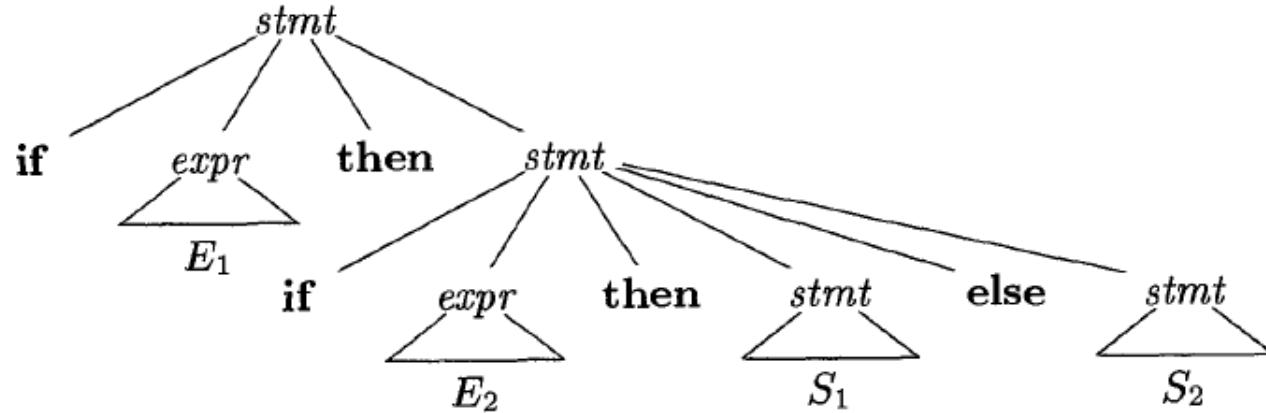
- For this grammar, the string

`if e1 then if e2 then s1 else s2`

has two parse trees

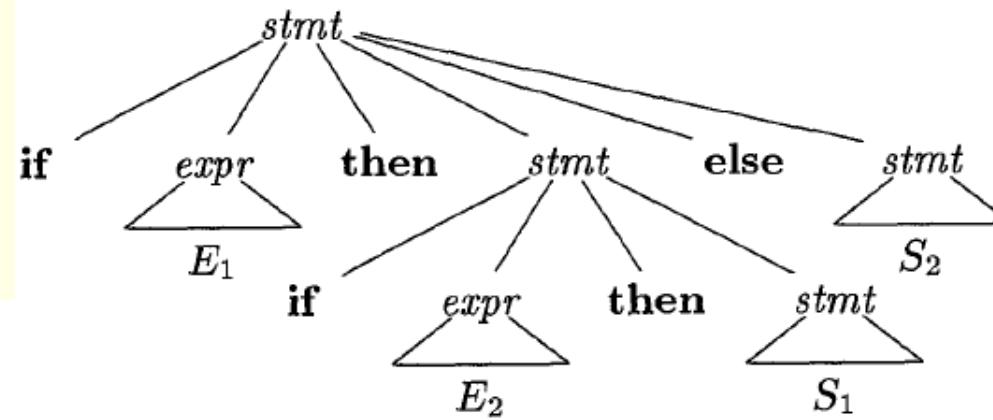
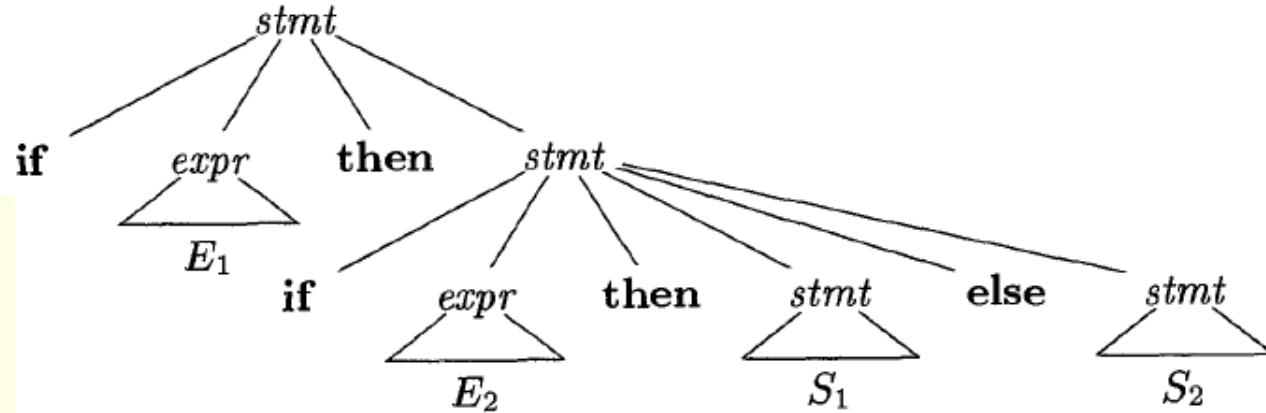
# Ambiguity in Programming Lang.

- Parse trees of `if e1 then if e2 then s1 else s2`



# Ambiguity in Programming Lang.

- Parse trees of `if e1 then if e2 then s1 else s2`



```
if e1  
  if e2  
    s1  
  else s2
```

```
if e1  
  if e2  
    s1  
  else s2
```

# Resolving dangling else problem

- General rule: match each **else** with the closest previous **unmatched if**. The grammar can be rewritten as

$\text{stmt} \rightarrow \text{matched-stmt}$

$| \text{ unmatched-stmt}$

$\text{matched-stmt} \rightarrow \text{if expr matched-stmt}$

$\text{else matched-stmt}$

$| \text{ others}$

$\text{unmatched-stmt} \rightarrow \text{if expr stmt}$

$| \text{ if expr matched-stmt}$

$\text{else unmatched-stmt}$

# Associativity

- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator
- In  $a+b+c$  b is taken by left +
- +, -, \*, / are left associative
- ^, = are right associative
- Grammar to generate strings with right associative operators
  - right  $\rightarrow$  letter = right | letter
  - letter  $\rightarrow$  a| b |...| z

# Precedence

- String  $a+5^*2$  has two possible interpretations because of two different parse trees corresponding to
$$(a+5)^*2 \text{ and } a+(5^*2)$$
- Precedence determines the correct interpretation.
- Next, an example of how precedence rules are encoded in a grammar

# Precedence/Associativity in the Grammar for Arithmetic Expressions

Ambiguous

$$E \rightarrow E + E \\ | \quad E * E$$

$$| \quad (E)$$

$$| \quad \text{num} \mid \text{id}$$

3 + 2 + 5

3 + 2 \* 5

- Unambiguous, with precedence and associativity rules honored

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow ( E ) \mid \text{num} \\ | \quad \text{id}$$

# Parsing

- Process of determination whether a string can be generated by a grammar
- Parsing falls in two categories:
  - Top-down parsing:

Construction of the parse tree starts at the root (from the start symbol) and proceeds towards leaves (token or terminals)
  - Bottom-up parsing:

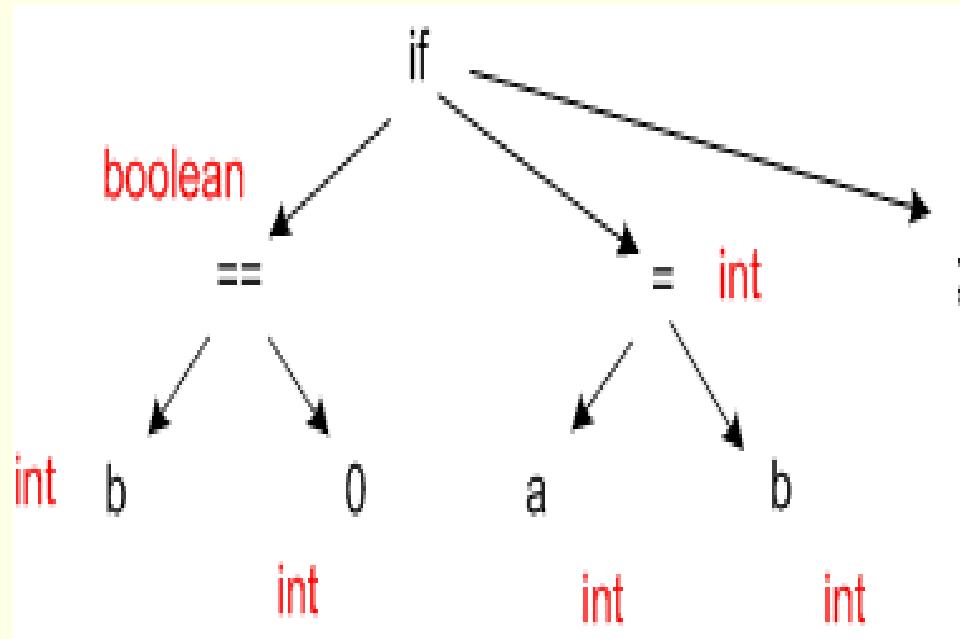
Construction of the parse tree starts from the leaf nodes (tokens or terminals of the grammar) and proceeds towards root (start symbol)

# References

- Compiler Design by Amey Karkare, IIT Kanpur  
<https://karkare.github.io/cs335/>
- Compilers: Principles, Techniques, and Tools, Second edition, 2006. by Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman

# Semantic Analysis

- Static checking
  - Type checking
  - Control flow checking
  - Uniqueness checking
  - Name checks
- Disambiguate overloaded operators
- Type coercion
- Error reporting



# Beyond syntax analysis

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
  - This problem is of identifying a language  $\{w\omega w \mid w \in \Sigma^*\}$
  - This language is not context free

# Beyond syntax ...

- Examples

```
string x; int y;
```

```
y = x + 3
```

the use of x could be a type error

```
int a, b;
```

```
a = b + c
```

c is not declared

- An identifier may refer to different variables in different parts of the program
- An identifier may be usable in one part of the program but not another

# Compiler needs to know?

- Whether a variable has been declared?
- Are there variables which have not been declared?
- What is the type of the variable?
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like  $A[i,j,k]$  is consistent with the declaration? Does it have three dimensions?

- How many arguments does a function take?
  - Are all invocations of a function consistent with the declaration?
  - If an operator/function is overloaded, which function is being invoked?
  - Inheritance relationship
  - Classes not multiply defined
  - Methods in a class are not multiply defined
- 
- The exact requirements depend upon the language

# How to answer these questions?

- These issues are part of semantic analysis phase
- Answers to these questions depend upon values like type information, number of parameters etc.
- Compiler will have to do some computation to arrive at answers
- The information required by computations may be non local in some cases

# How to ... ?

- Use formal methods
  - Context sensitive grammars
  - Extended attribute grammars
- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code
- Something in between !!!
  - Use attributes
  - Do analysis along with parsing
  - Use code for attribute value computation
  - However, code is developed systematically

# Why attributes ?

- For lexical analysis and syntax analysis formal techniques were used.
- However, we still had code in form of actions along with regular expressions and context free grammar
- The attribute grammar formalism is important
  - However, it is very difficult to implement
  - But makes many points clear
  - Makes “ad-hoc” code more organized
  - Helps in doing non local computations

# Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules

# Attribute Grammar Framework

- Two notations for associating semantic rules with productions
- **Syntax directed definition**
  - high level specifications
  - hides implementation details
  - explicit order of evaluation is not specified
- **Translation scheme**
  - indicate order in which semantic rules are to be evaluated
  - allow some implementation details to be shown

# Attribute Grammar Framework

- Conceptually both:
  - parse input token stream
  - build parse tree
  - traverse the parse tree to evaluate the semantic rules at the parse tree nodes
- Evaluation may:
  - save information in the symbol table
  - issue error messages
  - generate code
  - perform any other activity

# Example

- Consider a grammar for signed binary numbers

number  $\rightarrow$  sign list

sign  $\rightarrow$  + | -

list  $\rightarrow$  list bit | bit

bit  $\rightarrow$  0 | 1

- Build attribute grammar that annotates **number** with the value it represents

# Example

- Associate attributes with grammar symbols

**symbol**

number

sign

list

bit

**attributes**

value

negative

position, value

position, value

**production**

**Attribute rule**

symbol	attributes
number	value
sign	negative
list	position, value
bit	position, value

$\text{number} \rightarrow \text{sign } \text{list}$

$\text{list.position} \leftarrow 0$

$\text{if sign.negative}$

$\text{number.value} \leftarrow -\text{list.value}$

$\text{else}$

$\text{number.value} \leftarrow \text{list.value}$

$\text{sign} \rightarrow +$

$\text{sign.negative} \leftarrow \text{false}$

$\text{sign} \rightarrow -$

$\text{sign.negative} \leftarrow \text{true}$

# production

# Attribute rule

symbol	attributes
number	value
sign	negative
list	position, value
bit	position, value

$\text{list} \rightarrow \text{bit}$

$\text{bit.position} \leftarrow \text{list.position}$

$\text{list.value} \leftarrow \text{bit.value}$

$\text{list}_0 \rightarrow \text{list}_1 \text{ bit}$

$\text{list}_1.\text{position} \leftarrow \text{list}_0.\text{position} + 1$

$\text{bit.position} \leftarrow \text{list}_0.\text{position}$

$\text{list}_0.\text{value} \leftarrow \text{list}_1.\text{value} + \text{bit.value}$

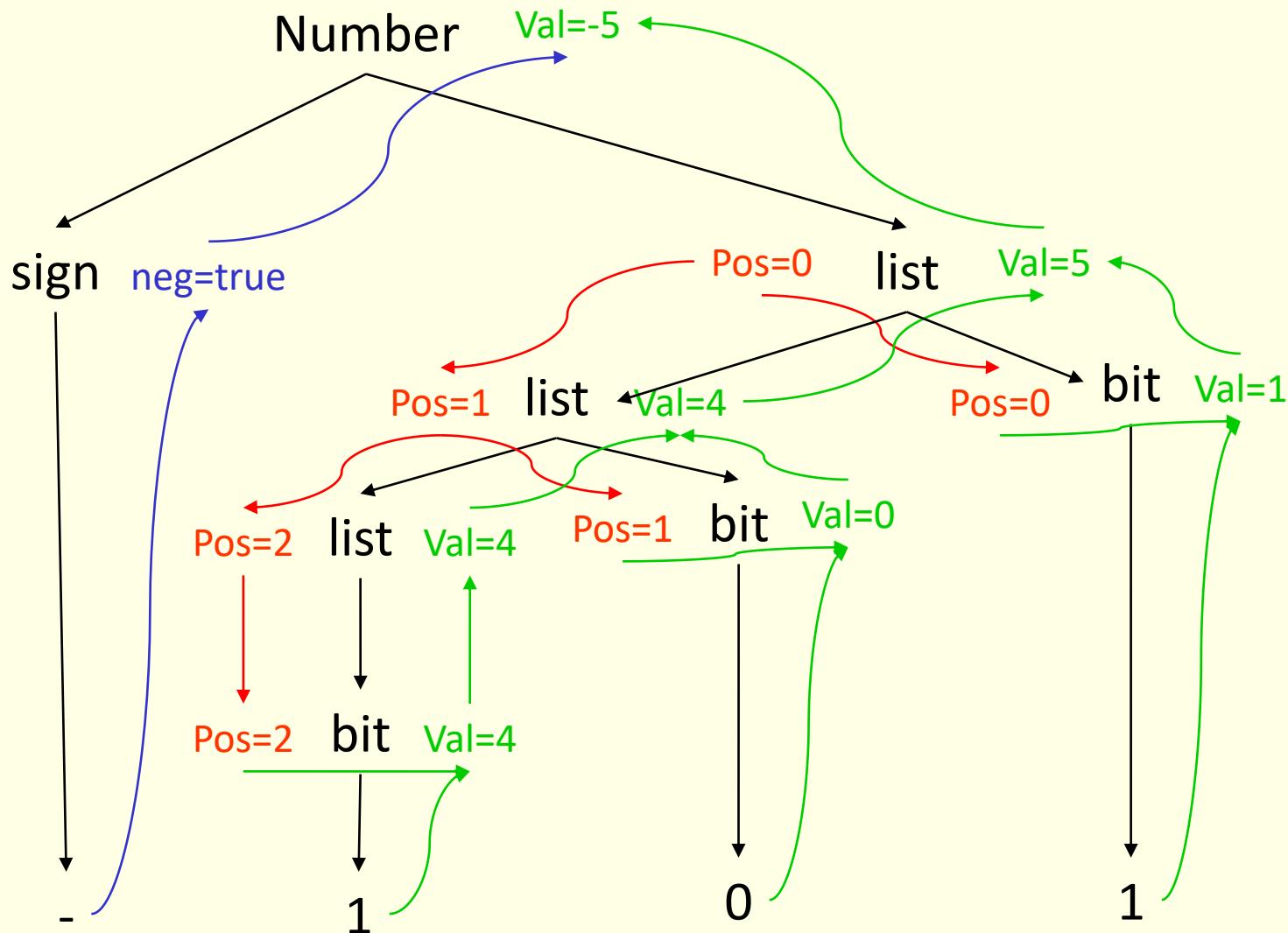
$\text{bit} \rightarrow 0$

$\text{bit.value} \leftarrow 0$

$\text{bit} \rightarrow 1$

$\text{bit.value} \leftarrow 2^{\text{bit.position}}$

# Parse tree and the dependence graph



# Attributes ...

- Attributes fall into two classes: *Synthesized* and *Inherited*
- Value of a synthesized attribute is computed from the values of children nodes
  - Attribute value for LHS of a rule comes from attributes of RHS
- Value of an inherited attribute is computed from the sibling and parent nodes
  - Attribute value for a symbol on RHS of a rule comes from attributes of LHS and RHS symbols

# Attributes ...

- Each grammar production  $A \rightarrow \alpha$  has associated with it a set of semantic rules of the form
$$b = f(c_1, c_2, \dots, c_k)$$
where  $f$  is a function, and  $x$ 
  - Either  $b$  is a synthesized attribute of  $A$
  - OR  $b$  is an inherited attribute of one of the grammar symbols on the right
- Attribute  $b$  depends on attributes  $c_1, c_2, \dots, c_k$

# Synthesized Attributes

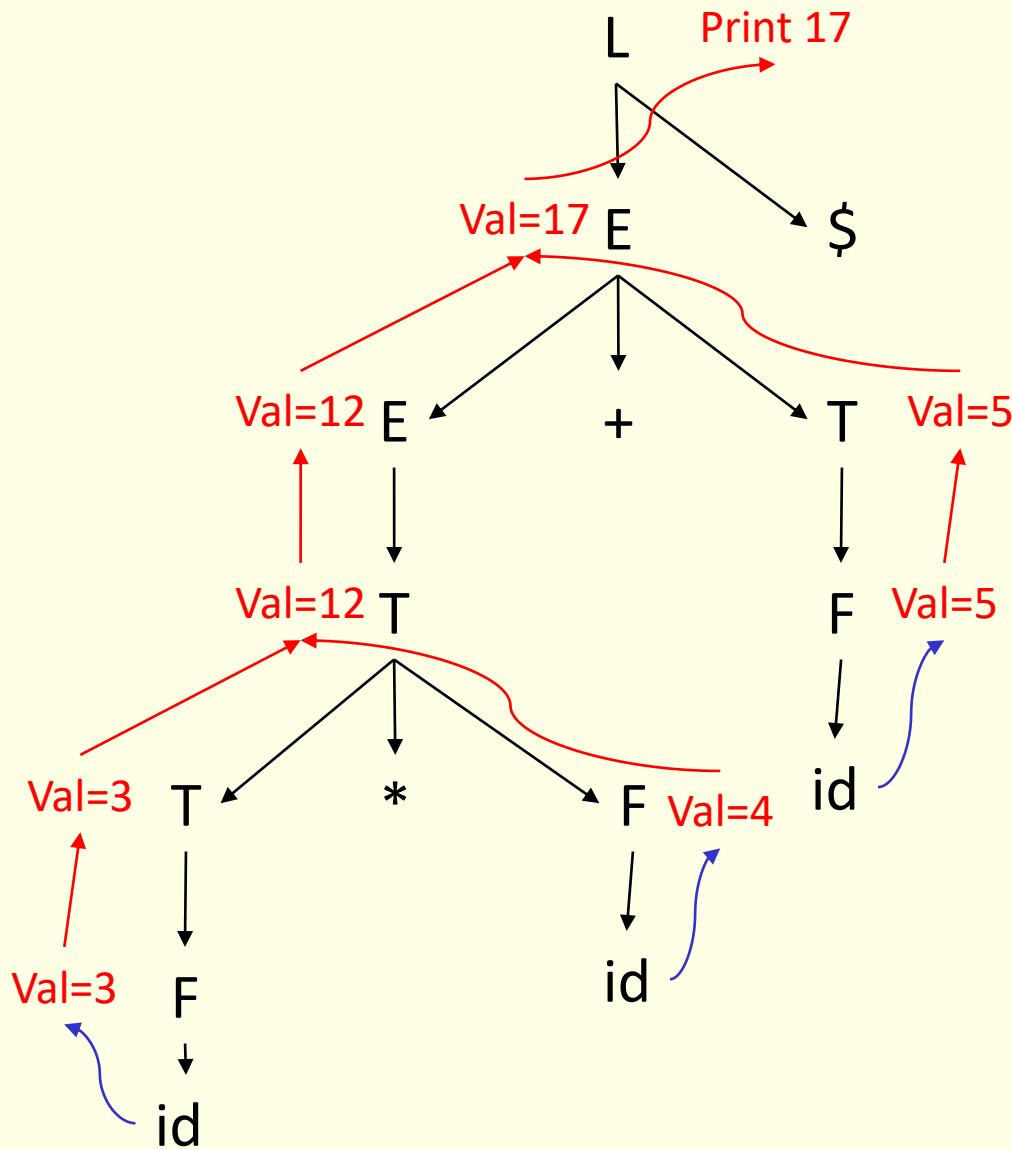
- a syntax directed definition that uses only synthesized attributes is said to be an S-attributed definition
- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

# Syntax Directed Definitions for a desk calculator program

$L \rightarrow E \$$	Print (E.val)
$E \rightarrow E + T$	E.val = E.val + T.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow T * F$	T.val = T.val * F.val
$T \rightarrow F$	T.val = F.val
$F \rightarrow (E)$	F.val = E.val
$F \rightarrow \text{digit}$	F.val = digit.lexval

- terminals are assumed to have only synthesized attribute values of which are supplied by lexical analyzer
- start symbol does not have any inherited attribute

# Parse tree for $3 * 4 + 5 \text{ n}$



# Inherited Attributes

- an inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings
- Used for finding out the context in which it appears
- possible to use only S-attributes but more natural to use inherited attributes

# Inherited Attributes

$D \rightarrow T\ L$        $L.in = T.type$

$T \rightarrow \text{real}$        $T.type = \text{real}$

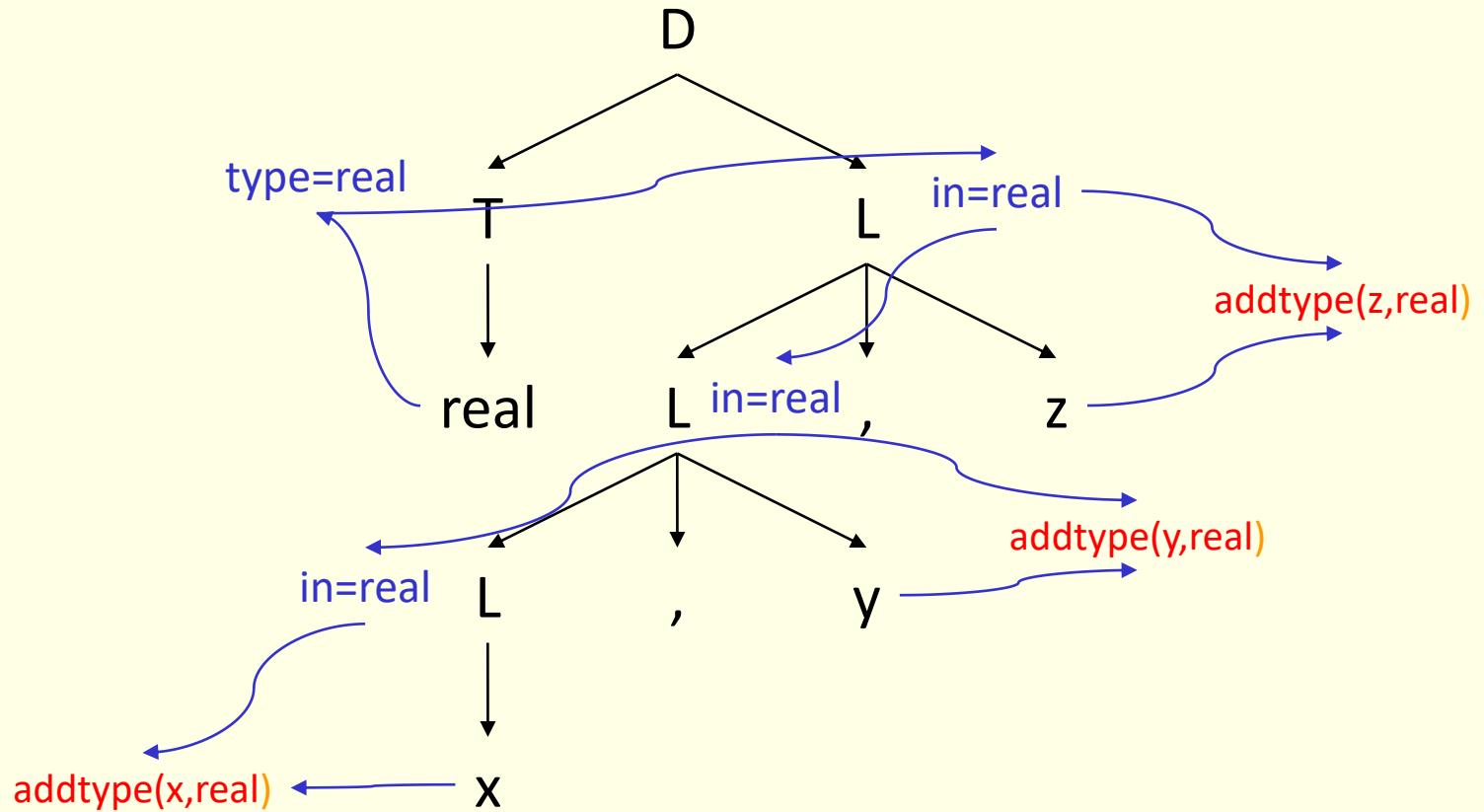
$T \ int$        $T.type = \text{int}$

$L \rightarrow L_1, \text{id}$        $L_1.in = L.in;$   
 $\text{addtype}(\text{id.entry}, L.in)$

$L \rightarrow \text{id}$        $\text{addtype}(\text{id.entry}, L.in)$

# Parse tree for

real x, y, z



# Dependence Graph

- If an attribute  $b$  depends on an attribute  $c$  then the semantic rule for  $b$  must be evaluated after the semantic rule for  $c$
- The dependencies among the nodes can be depicted by a directed graph called dependency graph

# Algorithm to construct dependency graph

for each node **n** in the parse tree do

    for each attribute **a** of the grammar symbol do  
        construct a node in the dependency graph

            for **a**

for each node **n** in the parse tree do

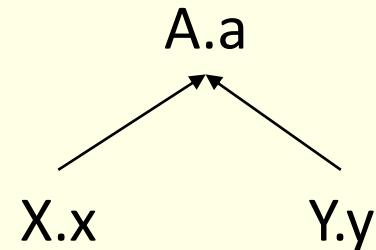
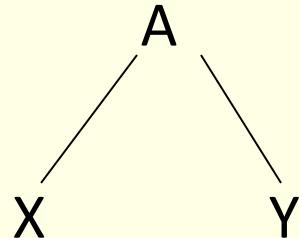
    for each semantic rule **b = f (c<sub>1</sub>, c<sub>2</sub>, ..., c<sub>k</sub>)**  
    { associated with production at **n** } do

        for i = 1 to k do

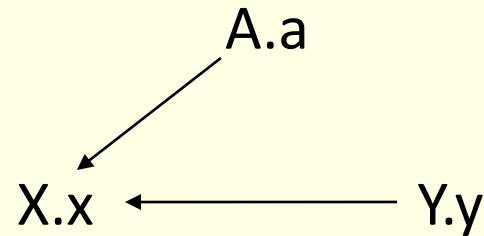
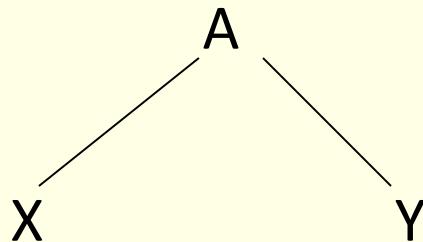
            construct an edge from **c<sub>i</sub>** to **b**

# Example

- Suppose  $A.a = f(X.x, Y.y)$  is a semantic rule for  $A \rightarrow X Y$

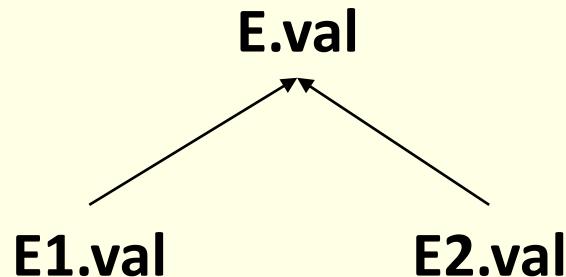


- If production  $A \rightarrow X Y$  has the semantic rule  $X.x = g(A.a, Y.y)$



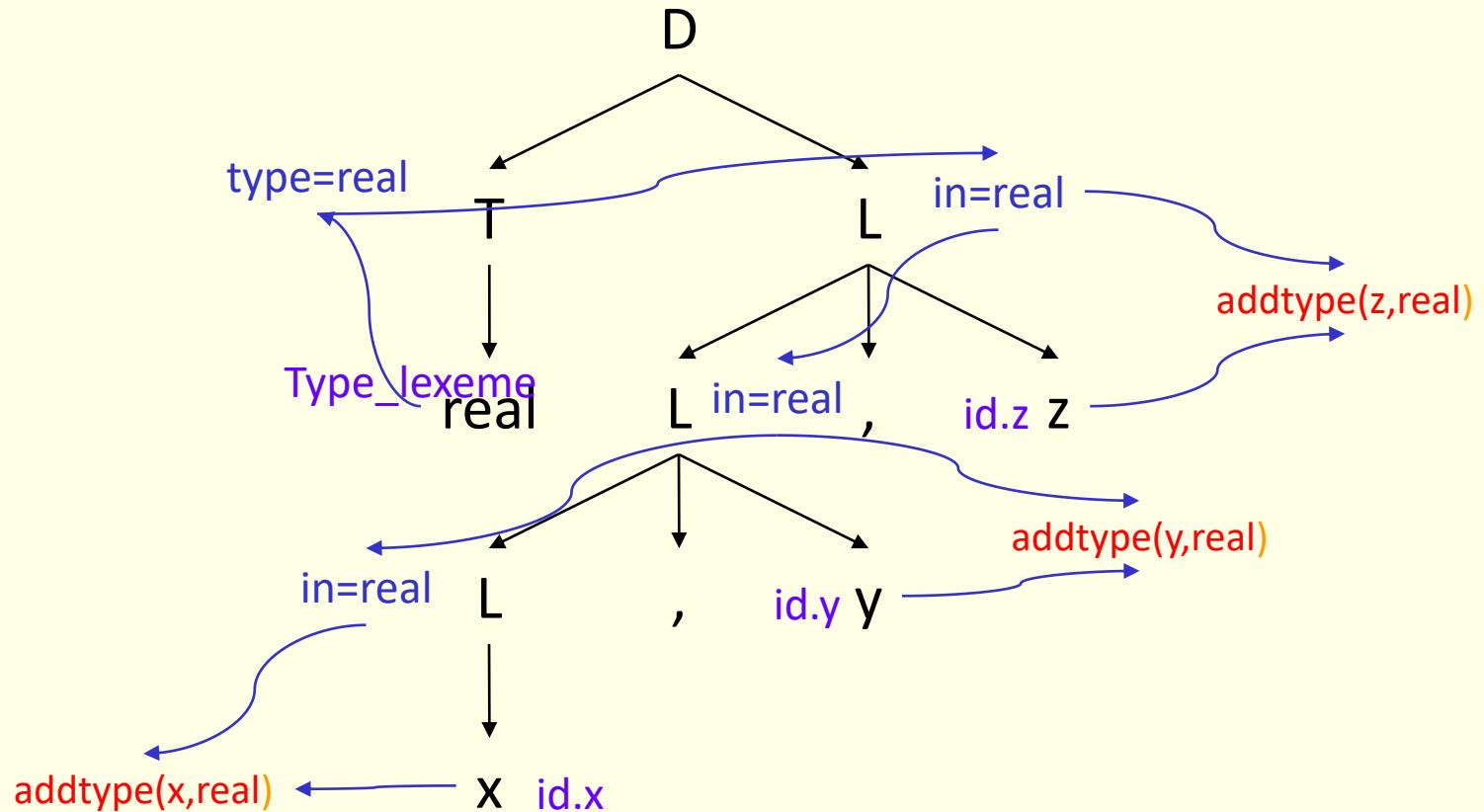
# Example

- Whenever following production is used in a parse tree  
 $E \rightarrow E_1 + E_2 \quad E.\text{val} = E_1.\text{val} + E_2.\text{val}$   
we create a dependency graph



# Example

- dependency graph for real id1, id2, id3
- put a dummy node for a semantic rule that consists of a procedure call



# Evaluation Order

- Any topological sort of dependency graph gives a valid order in which semantic rules must be evaluated

$a4 = \text{real}$

$a5 = a4$

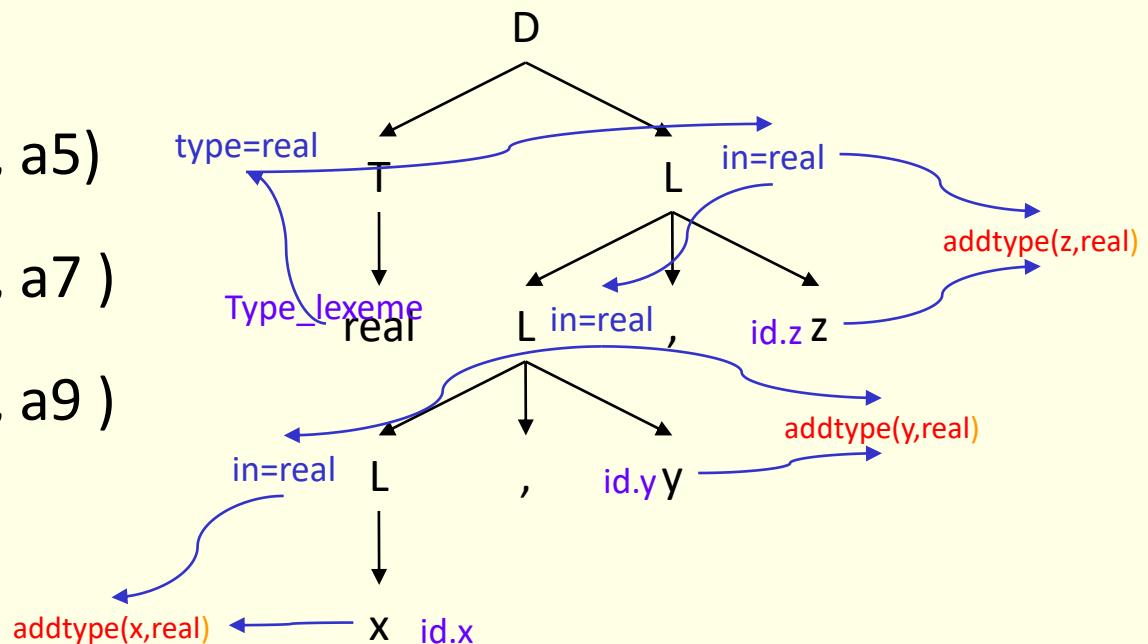
$\text{addtype}(\text{id3.entry}, a5)$

$a7 = a5$

$\text{addtype}(\text{id2.entry}, a7)$

$a9 := a7$

$\text{addtype}(\text{id1.entry}, a9)$



# Top Down Parsing

---

Dr. S. Suresh  
Assistant Professor  
Department of Computer Science  
Banaras Hindu University  
Varanasi – 211 005, India.

# Top down Parsing

- Construction of parse tree for the input, starting from root and creating the nodes of the parse tree in preorder
- Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

# Example: Top down Parsing

- Following grammar generates types of Pascal

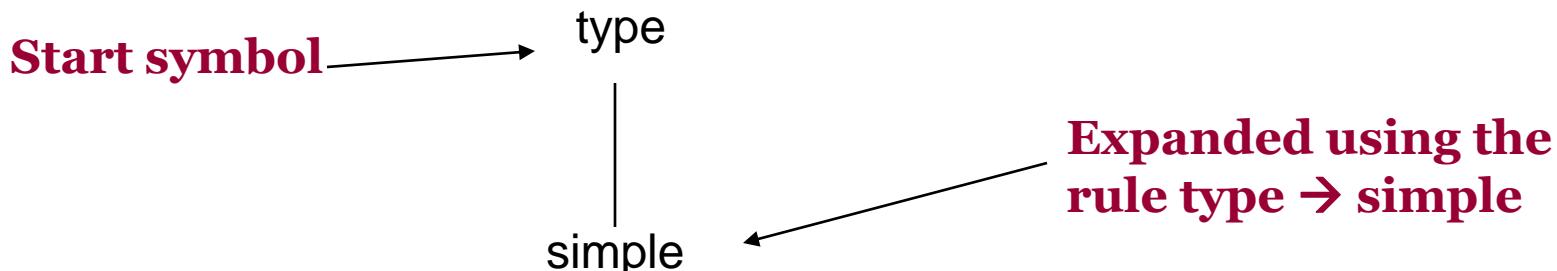
type  $\rightarrow$  simple  
|  $\uparrow$  id  
| array [ simple] of type

simple  $\rightarrow$  integer  
| char  
| num dotdot num

# Example ...

- Construction of parse tree is done by starting root labeled by start symbol
- Repeat following two steps
  - at node labeled with non terminal A select one of the production of A and construct children nodes  
**(Which production?)**
  - find the next node at which subtree is Constructed  
**(Which node?)**

- Parse  
array [ num dotdot num ] of integer



- Can not proceed as non terminal “simple” never generates a string beginning with token “array”. Therefore, requires back-tracking.
- Back-tracking is not desirable therefore, take help of a “look-ahead” token. The current token is treated as look-ahead token. (**restricts the class of grammars**)

array [ num dotdot num ] of integer

look-ahead

type

array

[

simple

]

of

type

Left most non terminal

num

dotdot

num

simple

integer

Start symbol

Expand using the rule  
 $\text{type} \rightarrow \text{array} [ \text{simple} ] \text{ of type}$

all the tokens exhausted  
Parsing completed

Left most non terminal  
Expand using the rule  
 $\text{type} \rightarrow \text{simple}$

Left most non terminal

Expand using the rule  
 $\text{simple} \rightarrow \text{integer}$

# Recursive descent parsing

First set:

Let there be a production

$$A \rightarrow \alpha$$

then  $\text{First}(\alpha)$  is set of tokens that appear as the first token in the strings generated from  $\alpha$

For example :

$$\text{First}(\text{simple}) = \{\text{integer}, \text{char}, \text{num}\}$$

$$\text{First}(\text{num dotdot num}) = \{\text{num}\}$$

# Define a procedure for each non terminal

```
procedure type;
  if lookahead in {integer, char, num}
    then simple
  else if lookahead =  $\uparrow$ 
    then begin match( $\uparrow$ );
           match(id)
        end
  else if lookahead = array
    then begin match(array);
           match([]);
           simple;
           match([]);
           match(of);
           type
        end
  else error;
```

```
procedure simple;
  if lookahead = integer
    then match(integer)
  else if lookahead = char
    then match(char)
  else if lookahead = num
    then begin match(num);
              match(dotdot);
              match(num)
            end
  else
    error;
```

---

```
procedure match(t:token);
  if lookahead = t
    then lookahead = next token
  else error;
```

# Left recursion

- A grammar is left recursive if it has a nonterminal  $A$  such that there is a derivation  $A \Rightarrow^* A\alpha$  for some string  $\alpha$
- Top-down parsing methods cannot handle left-recursive grammars
- Transformation is needed to eliminate left recursion.

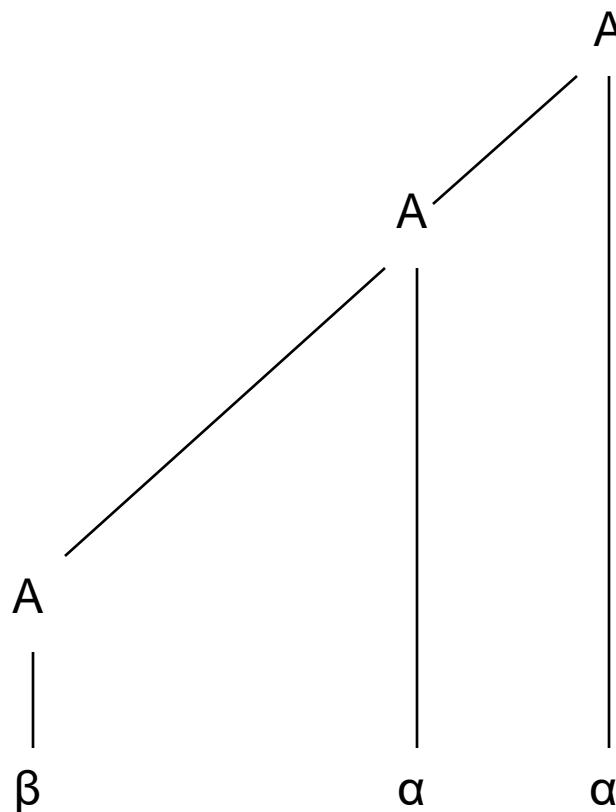
# Left recursion

- A top down parser with production  
 $A \rightarrow A \alpha$  may loop forever
- From the grammar  $A \rightarrow A \alpha \mid \beta$   
left recursion may be eliminated by transforming the grammar to

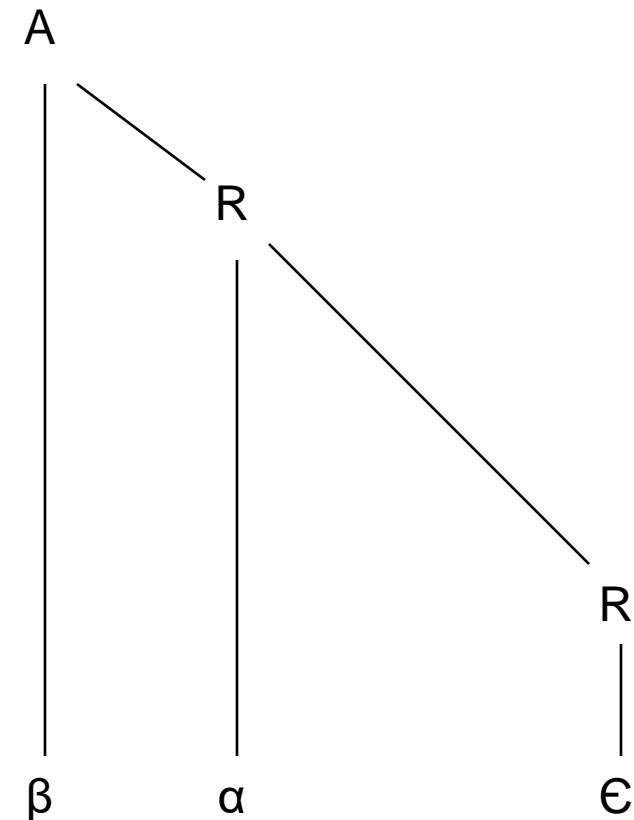
$$A \rightarrow \beta \ R$$

$$R \rightarrow \alpha \ R \mid \varepsilon$$

**Parse tree corresponding  
to left recursive grammar**



**Parse tree corresponding  
to the modified grammar**



**Both the trees generate string  $\beta\alpha^*$**

# Example

- Consider grammar for arithmetic expressions

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow ( E ) \mid id$$

- After removal of left recursion the grammar becomes

$$E \rightarrow T E'$$
$$E' \rightarrow + T E' \mid \epsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T' \mid \epsilon$$
$$F \rightarrow ( E ) \mid id$$

# Removal of left recursion

In general

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \\ \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

where no  $\beta_i$  begins with an A.

transforms to

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

# Left recursion hidden due to many productions

- Left recursion may also be introduced by two or more grammar rules. For example:

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \mid \epsilon \end{aligned}$$

there is a left recursion because

$$S \rightarrow Aa \rightarrow Sda$$

- In such cases, left recursion is removed systematically
  - Starting from the first rule and replacing all the occurrences of the first non terminal symbol
  - Removing left recursion from the modified grammar

# Removal of left recursion due to many productions ...

- After the first step (substitute S by its rhs in the rules) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Aad \mid bd \mid \epsilon$$

- After the second step (removal of left recursion) the grammar becomes

$$S \rightarrow Aa \mid b$$
$$A \rightarrow bdA' \mid A'$$
$$A' \rightarrow cA' \mid adA' \mid \epsilon$$

# Left factoring

- In top-down parsing when it is not clear which production to choose for expansion of a symbol  
**defer the decision till we have seen enough input.**

In general if  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

defer decision by expanding  $A$  to  $\alpha A'$

we can then expand  $A'$  to  $\beta_1$  or  $\beta_2$

- Therefore  $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2$

transforms to

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

# Dangling else problem again

Dangling else problem can be handled by left factoring

$\text{stmt} \rightarrow \text{if expr then stmt else stmt}$   
|  $\text{if expr then stmt}$

can be transformed to

$\text{stmt} \rightarrow \text{if expr then stmt S'}$   
 $S' \rightarrow \text{else stmt} \mid \epsilon$

# Predictive parsers

- A non recursive top down parsing method
- Parser “predicts” which production to use
- It removes backtracking by fixing one production for every non-terminal and input token(s)
- Predictive parsers accept LL( $k$ ) languages
  - First L stands for left to right scan of input
  - Second L stands for leftmost derivation
  - $k$  stands for number of lookahead token
- In practice LL(1) is used

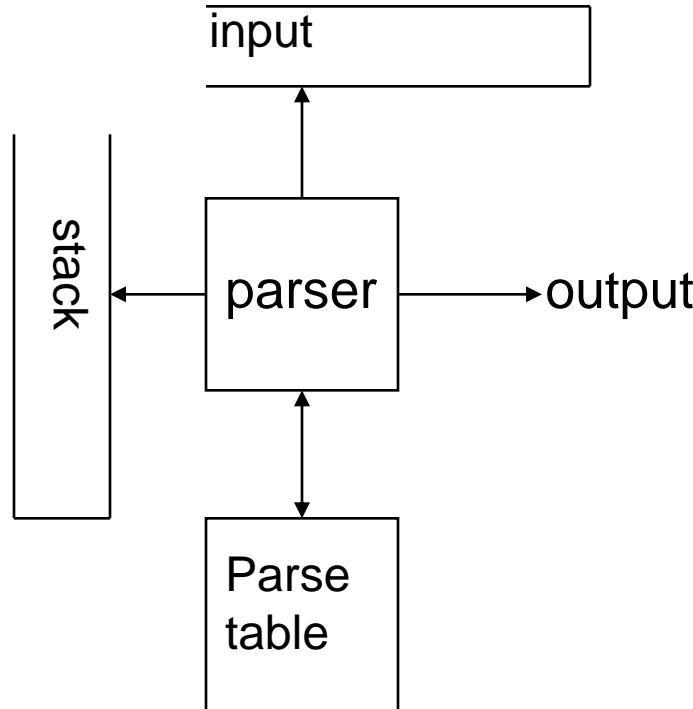
# LL(1) grammars

A grammar  $G$  is **LL(1)** if and only if whenever  $A \rightarrow \alpha \mid \beta$  are two distinct productions of  $G$ , the following conditions hold:

- For no terminal  $a$  do both  $\alpha$  and  $\beta$  derive strings beginning with  $a$
- At most one of  $\alpha$  and  $\beta$  can derive the empty string
- If  $\beta \Rightarrow^* \epsilon$ , then  $\alpha$  does not derive any string beginning with a terminal in  $\text{Follow}(A)$ .  
Likewise, if  $\alpha \Rightarrow^* \epsilon$ , then  $\beta$  does not derive any string beginning with a terminal in  $\text{Follow}(A)$

# Predictive parsing

- Predictive parser can be implemented by maintaining an external stack



**Parse table is a  
two dimensional array  
 $M[X,a]$  where “X” is a  
non terminal and “a” is  
a terminal of the grammar**

# Parsing algorithm

- The parser considers 'X' the symbol on top of stack, and 'a' the current input symbol
- These two symbols determine the action to be taken by the parser
- Assume that '\$' is a special token that is at the bottom of the stack and terminates the input string

if  $X = a = \$$  then halt

if  $X = a \neq \$$  then pop( $x$ ) and  $ip++$

if  $X$  is a non terminal

    then if  $M[X,a] = \{X \rightarrow UVW\}$

        then begin pop( $X$ ); push( $W,V,U$ )

        end

    else error

# Example

- Consider the grammar

$$E \rightarrow T E'$$
$$E' \rightarrow +T E' \mid \epsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T' \mid \epsilon$$
$$F \rightarrow ( E ) \mid \text{id}$$

# Parse table for the grammar

	<b>id</b>	+	*	(	)	\$
<b>E</b>	$E \rightarrow TE'$			$E \rightarrow TE'$		
<b>E'</b>		$E' \rightarrow +TE$ ,			$E' \rightarrow E$	$E' \rightarrow E$
<b>T</b>	$T \rightarrow FT'$			$T \rightarrow FT'$		
<b>T'</b>		$T' \rightarrow E$	$T' \rightarrow *FT'$		$T' \rightarrow E$	$T' \rightarrow E$
<b>F</b>	$F \rightarrow id$			$F \rightarrow (E)$		

**Blank entries are error states. For example  
E can not derive a string starting with ‘+’**

# Example

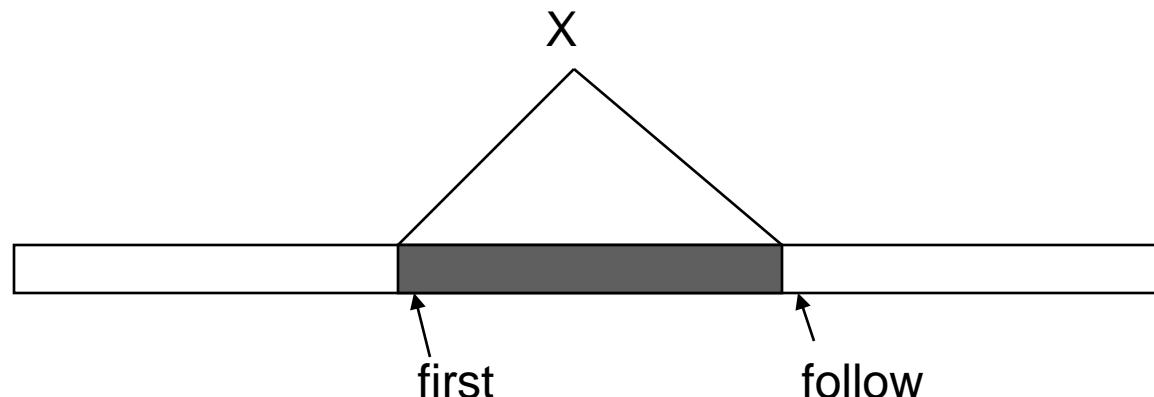
Stack	input	action
\$E	id + id * id \$	expand by $E \rightarrow TE'$
\$E'T	id + id * id \$	expand by $T \rightarrow FT'$
\$E'T'F	id + id * id \$	expand by $F \rightarrow id$
\$E'T'id	id + id * id \$	pop id and ip++
\$E'T'	+ id * id \$	expand by $T' \rightarrow \epsilon$
\$E'	+ id * id \$	expand by $E' \rightarrow +TE'$
\$E'T+	+ id * id \$	pop + and ip++
\$E'T	id * id \$	expand by $T \rightarrow FT'$

# Example ...

Stack	input	action
\$E'T'F	id * id \$	expand by F → id
\$E'T'id	id * id \$	pop id and ip++
\$E'T'	* id \$	expand by T' → *FT'
\$E'T'F*	* id \$	pop * and ip++
\$E'T'F	id \$	expand by F → id
\$E'T'id	id \$	pop id and ip++
\$E'T'	\$	expand by T' → ε
\$E'	\$	expand by E' → ε
\$	\$	halt

# Constructing parse table

- Table can be constructed if for every non terminal, every lookahead symbol can be handled by at most one production
- First( $\alpha$ ) for a string of terminals and non terminals  $\alpha$  is
  - Set of symbols that might begin the fully expanded (made of only tokens) version of  $\alpha$
- Follow( $X$ ) for a non terminal  $X$  is
  - set of symbols that might follow the derivation of  $X$  in the input stream



# First and Follow

- **First( $\alpha$ )** for a string of terminals and non terminals  $\alpha$  is
  - The set of terminals that begin strings derived from  $\alpha$
  - If  $\alpha \Rightarrow^* \epsilon$ , then  $\epsilon$  is also in **First( $\alpha$ )**
  - For example,  $A \Rightarrow^* c\beta$ , so  $c$  is in **First( $\alpha$ )**
- **Follow( $X$ )** for a non terminal  $X$  is
  - The set of terminals  $a$  that can appear immediately to the right of  $A$  in some sentential form;
  - That is, the set of terminals  $a$  such that there exists a derivation of the form  $S \Rightarrow^* \alpha A a \beta$  for some  $\alpha$  and  $\beta$
  - In addition, if  $A$  is the right most symbol in some sentential form, then  $\$$  is in **Follow( $X$ )**
  - Note: there may have been any other symbol between  $A$  and  $a$  at some time of derivation

# Compute first sets

- If  $X$  is a terminal symbol then  $\text{First}(X) = \{X\}$
- If  $X \rightarrow \epsilon$  is a production then  $\epsilon$  is in  $\text{First}(X)$
- If  $X$  is a non terminal
  - and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$  then
    - if for some  $i$ ,  $a$  is in  $\text{First}(Y_i)$  and  $\epsilon$  is in all of  $\text{First}(Y_j)$  (such that  $j < i$ ) then  $a$  is in  $\text{First}(X)$
    - i.e., if  $a$  is in  $\text{First}(Y_i)$  and  $Y_1 Y_2 \dots Y_{i-1} \Rightarrow^* \epsilon$ , then  $a$  is in  $\text{First}(X)$
  - If  $\epsilon$  is in  $\text{First}(Y_1) \dots \text{First}(Y_k)$  then  $\epsilon$  is in  $\text{First}(X)$

# Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow ( E ) \mid \text{id}$$

$$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{ (, \text{id} \} \}$$

$$\text{First}(E') = \{ +, \epsilon \}$$

$$\text{First}(T') = \{ *, \epsilon \}$$

# Compute follow sets

1. Place \$ in  $\text{follow}(S)$  where S is a start symbol
2. If there is a production  $A \rightarrow \alpha B \beta$   
then everything in  $\text{first}(\beta)$  (except  $\epsilon$ ) is in  $\text{follow}(B)$
3. If there is a production  $A \rightarrow \alpha B$   
then everything in  $\text{follow}(A)$  is in  $\text{follow}(B)$
4. If there is a production  $A \rightarrow \alpha B \beta$   
and  $\text{First}(\beta)$  contains  $\epsilon$   
then everything in  $\text{follow}(A)$  is in  $\text{follow}(B)$

Since follow sets are defined in terms of follow sets last two steps have to be repeated until follow sets converge

# Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow ( E ) \mid \text{id}$$

$$\text{follow}(E) = \text{follow}(E') = \{ \$, ) \}$$

$$\text{follow}(T) = \text{follow}(T') = \{ \$, ), + \}$$

$$\text{follow}(F) = \{ \$, ), +, * \}$$

# Construction of parse table

- for each production  $A \rightarrow \alpha$  do
  - for each terminal 'a' in  $\text{first}(\alpha)$   
 $M[A,a] = A \rightarrow \alpha$
  - If  $\epsilon$  is in  $\text{First}(\alpha)$   
 $M[A,b] = A \rightarrow \alpha$   
for each terminal b in  $\text{follow}(A)$
  - If  $\epsilon$  is in  $\text{First}(\alpha)$  and  $\$$  is in  $\text{follow}(A)$   
 $M[A,\$] = A \rightarrow \alpha$
- A grammar whose parse table has no multiple entries is called LL(1)

# Practice Assignment

- Construct LL(1) parse table for the expression grammar

bexpr  $\rightarrow$  bexpr or bterm | bterm

bterm  $\rightarrow$  bterm and bfactor | bfactor

bfactor  $\rightarrow$  not bfactor | ( bexpr ) | true | false

- Steps to be followed

- Remove left recursion
  - Compute first sets
  - Compute follow sets
  - Construct the parse table

- Not to be submitted

# Error handling

- Stop at the first error and print a message
  - Compiler writer friendly
  - But not user friendly
- Every reasonable compiler must recover from error and identify as many errors as possible
- However, multiple error messages due to a single fault must be avoided
- Error recovery methods
  - Panic mode
  - Phrase level recovery
  - Error productions
  - Global correction

# Panic mode

- Simplest and the most popular method
- Most tools provide for specifying panic mode recovery in the grammar
- When an error is detected
  - Discard tokens one at a time until a set of tokens is found whose role is clear
  - Skip to the next token that can be placed reliably in the parse tree

# Panic mode ...

- Consider following code  
begin

```
a = b + c;  
x = p r ;  
h = x < 0;
```

```
end;
```

- The second expression has syntax error
- Panic mode recovery for begin-end block  
skip ahead to next ';' and try to parse the next expression
- It discards one expression and tries to continue parsing
- May fail if no further ';' is found

# Phrase level recovery

- Make local correction to the input
- Works only in limited situations
  - A common programming error which is easily detected
  - For example insert a ";" after closing "}" of a class definition
- Does not work very well!

# Error productions

- Add erroneous constructs as productions in the grammar
- Works only for most common mistakes which can be easily identified
- Essentially makes common errors as part of the grammar
- Complicates the grammar and does not work very well

# Global corrections

- Considering the program as a whole find a correct “nearby” program
- Nearness may be measured using certain metric
- PL/C compiler implemented this scheme: anything could be compiled!
- It is complicated and not a very good idea!

# Error Recovery in LL(1) parser

- Error occurs when a parse table entry  $M[A,a]$  is empty
- Skip symbols in the input until a token in a selected set (synch) appears
- Place symbols in  $\text{follow}(A)$  in synch set. Skip tokens until an element in  $\text{follow}(A)$  is seen.  
Pop(A) and continue parsing
- Add symbol in  $\text{first}(A)$  in synch set. Then it may be possible to resume parsing according to A if a symbol in  $\text{first}(A)$  appears in input.

# Assignment

- **Reading assignment:** Read about error recovery in LL(1) parsers
- **Practice Assignment:**
  - introduce synch symbols (using both follow and first sets) in the parse table created for the boolean expression grammar in the previous assignment
  - Parse “not (true and or false)” and show how error recovery works

# Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root  
OR
- Reduce a string w of input to start symbol of grammar

Consider a grammar

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \quad | \quad b \\ B &\rightarrow d \end{aligned}$$

And reduction of a string

a b b c d e  
a A b c d e  
a A d e  
a A B e  
S

**Right most derivation**

**S → a A B e**  
**→ a A d e**  
**→ a A b c d e**  
**→ a b b c d e**

# Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character ":"
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is .w

# Shift reduce parsing ...

- Bottom up parsing has two actions
- Shift: move terminal symbol from right string to left string
  - if string before shift is                 $\alpha.pqr$
  - then string after shift is     $\alpha.p.qr$
- Reduce: immediately on the left of “.” identify a string same as RHS of a production and replace it by LHS
  - if string before reduce action is     $\alpha\beta.pqr$
  - and  $A \rightarrow \beta$  is a production
  - then string after reduction is  $\alpha A.pqr$

# Example

Assume grammar is  
Parse  $\text{id}^*\text{id}+\text{id}$

$$E \rightarrow E+E \mid E^*E \mid \text{id}$$

String	action
. $\text{id}^*\text{id}+\text{id}$	shift
$\text{id}.$ $\text{id}^*\text{id}+\text{id}$	reduce $E \rightarrow \text{id}$
$E.$ $\text{id}^*\text{id}+\text{id}$	shift
$E^*.$ $\text{id}^*\text{id}+\text{id}$	shift
$E^*\text{id}.$ $\text{id}+\text{id}$	reduce $E \rightarrow \text{id}$
$E^*\text{E}.$ $\text{id}+\text{id}$	reduce $E \rightarrow E^*E$
$E.$ $\text{id}$	shift
$E.$ $\text{id}$	shift
$E+\text{id}.$	Reduce $E \rightarrow \text{id}$
$E+E.$	Reduce $E \rightarrow E+E$
E.	ACCEPT

# Shift reduce parsing ...

- Symbols on the left of “.” are kept on a stack
  - Top of the stack is at “.”
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issue: when to shift and when to reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

# Bottom up parsing ...

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery

# Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - Which production to use for reduction?
- Sometimes parser can reduce but it should not:  
 $X \rightarrow \epsilon$  can always be reduced!
- Sometimes parser can reduce in different ways!
- Given stack  $\delta$  and input symbol  $a$ , should the parser
  - Shift  $a$  onto stack (making it  $\delta a$ )
  - Reduce by some production  $A \rightarrow \beta$  assuming that stack has form  $\alpha \beta$  (making it  $\alpha A$ )
  - Stack can have many combinations of  $\alpha \beta$
  - How to keep track of length of  $\beta$ ?

# Handle

- A string that matches right hand side of a production and whose replacement gives a step in the reverse right most derivation
- If  $S \xrightarrow{rm^*} \alpha Aw \xrightarrow{rm} \alpha\beta w$  then  $\beta$  (corresponding to production  $A \rightarrow \beta$ ) in the position following  $\alpha$  is a handle of  $\alpha\beta w$ . The string  $w$  consists of only terminal symbols
- We only want to reduce handle and not any rhs
- **Handle pruning:** If  $\beta$  is a handle and  $A \rightarrow \beta$  is a production then replace  $\beta$  by  $A$
- A right most derivation in reverse can be obtained by handle pruning.

# Handles ...

- Handles always appear at the top of the stack and never inside it
- This makes stack a suitable data structure
- Consider two cases of right most derivation to verify the fact that handle appears on the top of the stack
  - $S \rightarrow \alpha A z \rightarrow \alpha \beta B y z \rightarrow \alpha \beta \gamma y z$
  - $S \rightarrow \alpha B x A z \rightarrow \alpha B x y z \rightarrow \alpha \gamma x y z$
- Bottom up parsing is based on recognizing handles

# Handle always appears on the top

Case I:  $S \rightarrow \alpha A z \rightarrow \alpha \beta B y z \rightarrow \alpha \beta \gamma y z$

stack	input	action
$\alpha \beta \gamma$	yz	reduce by $B \rightarrow \gamma$
$\alpha \beta B$	yz	shift y
$\alpha \beta B y$	z	reduce by $A \rightarrow \beta B y$
$\alpha A$	z	

Case II:  $S \rightarrow \alpha B x A z \rightarrow \alpha B x y z \rightarrow \alpha \gamma x y z$

stack	input	action
$\alpha \gamma$	xyz	reduce by $B \rightarrow \gamma$
$\alpha B$	xyz	shift x
$\alpha B x$	yz	shift y
$\alpha B x y$	z	reduce $A \rightarrow y$
$\alpha B x A$	z	

# Conflicts

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- However, what happens when there is a choice
  - What action to take in case both shift and reduce are valid?  
**shift-reduce conflict**
  - Which rule to use for reduction if reduction is possible by more than one rule?  
**reduce-reduce conflict**
- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

# Shift reduce conflict

Consider the grammar  $E \rightarrow E+E \mid E^*E \mid id$   
and input       $id+id^*id$

stack	input	action
$E+E$	$*id$	reduce by $E \rightarrow E+E$
$E$	$*id$	shift
$E^*$	$id$	shift
$E^*id$		reduce by $E \rightarrow id$
$E^*E$		reduce by $E \rightarrow E^*EE$

stack	input	action
$E+E$	$*id$	shift
$E+E^*$	$id$	shift
$E+E^*id$		reduce by $E \rightarrow id$
$E+E^*E$		reduce by $E \rightarrow E^*E$
$E+E$		reduce by $E \rightarrow E+E$

# Reduce reduce conflict

Consider the grammar  $M \rightarrow R+R \mid R+c \mid R$

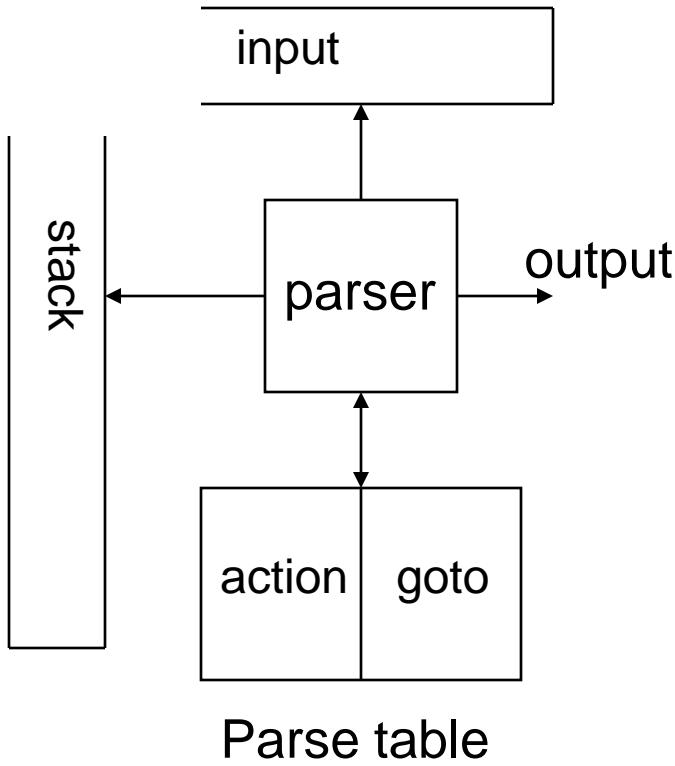
$$R \rightarrow c$$

and input  $c+c$

Stack	input	action
	$c+c$	shift
$c$	$+c$	reduce by $R \rightarrow c$
$R$	$+c$	shift
$R+$	$c$	shift
$R+c$		reduce by $R \rightarrow c$
$R+R$		reduce by $\rightarrow R+RM$

Stack	input	action
	$c+c$	<b>shift</b>
$c$	$+c$	<b>reduce by <math>R \rightarrow c</math></b>
$R$	$+c$	<b>shift</b>
$R+$	$c$	<b>shift</b>
$R+c$		<b>reduce by <math>M \rightarrow R+cM</math></b>

# LR parsing



- Input contains the input string.
- Stack contains a string of the form  $S_0X_1S_1X_2\dots\dots X_nS_n$  where each  $X_i$  is a grammar symbol and each  $S_i$  is a state.
- Tables contain action and goto parts.
- action table is indexed by state and terminal symbols.
- goto table is indexed by state and non terminal symbols.

# Actions in an LR (shift reduce) parser

- Assume  $S_i$  is top of stack and  $a_i$  is current input symbol
- Action  $[S_i, a_i]$  can have four values
  1. shift  $a_i$  to the stack and goto state  $S_j$
  2. reduce by a rule
  3. Accept
  4. error

# Configurations in LR parser

Stack:  $S_0 X_1 S_1 X_2 \dots X_m S_m$       Input:  $a_i a_{i+1} \dots a_n \$$

- If  $\text{action}[S_m, a_i] = \text{shift } S$

Then the configuration becomes

Stack:  $S_0 X_1 S_1 \dots X_m S_m a_i S$       Input:  $a_{i+1} \dots a_n \$$

- If  $\text{action}[S_m, a_i] = \text{reduce } A \rightarrow \beta$

Then the configuration becomes

Stack:  $S_0 X_1 S_1 \dots X_{m-r} S_{m-r} AS$       Input:  $a_i a_{i+1} \dots a_n \$$

Where  $r = |\beta|$  and  $S = \text{goto}[S_{m-r}, A]$

- If  $\text{action}[S_m, a_i] = \text{accept}$

Then parsing is completed. HALT

- If  $\text{action}[S_m, a_i] = \text{error}$

Then invoke error recovery routine.

# LR parsing Algorithm

Initial state:      Stack:  $S_0$       Input:  $w\$$

Loop{

  if action[S,a] = shift  $S'$

    then push(a); push( $S'$ ); ip++

  else if action[S,a] = reduce  $A \rightarrow \beta$

    then pop ( $2^*|\beta|$ ) symbols;

    push(A); push (goto[ $S''$ ,A])

    ( $S''$  is the state after popping symbols)

  else if action[S,a] = accept

    then exit

  else error

}

# Example

Consider the grammar  
And its parse table

$$\begin{array}{l}
 E \rightarrow E + T \quad | \quad T \\
 T \rightarrow T^* F \quad | \quad F \\
 F \rightarrow ( E ) \quad | \quad id
 \end{array}$$

State	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				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			

# Parse id + id \* id

Stack	Input	Action
0	id+id*id\$	shift 5
0 id 5	+id*id\$	reduce by F→id
0 F 3	+id*id\$	reduce by T→F
0 T 2	+id*id\$	reduce by E→T
0 E 1	+id*id\$	shift 6
0 E 1 + 6	id*id\$	shift 5
0 E 1 + 6 id 5	*id\$	reduce by F→id
0 E 1 + 6 F 3	*id\$	reduce by T→F
0 E 1 + 6 T 9	*id\$	shift 7
0 E 1 + 6 T 9 * 7	id\$	shift 5
0 E 1 + 6 T 9 * 7 id 5	\$	reduce by F→id
0 E 1 + 6 T 9 * 7 F 10	\$	reduce by T→T*F
0 E 1 + 6 T 9	\$	reduce by E→E+T
0 E 1	\$	ACCEPT

# Parser states

- Goal is to know the valid reductions at any given point
- Summarize all possible stack prefixes  $\alpha$  as a parser state
- Parser state is defined by a DFA state that reads in the stack  $\alpha$
- Accept states of DFA are unique reductions

# Constructing parse table

## Augment the grammar

- G is a grammar with start symbol S
- The augmented grammar G' for G has a new start symbol S' and an additional production  $S' \rightarrow S$
- When the parser reduces by this rule it will stop with accept

# Viable prefixes

- $\alpha$  is a viable prefix of the grammar if
  - There is a  $w$  such that  $\alpha w$  is a right sentential form
  - $\alpha.w$  is a configuration of the shift reduce parser
- As long as the parser has viable prefixes on the stack no parser error has been seen
- The set of viable prefixes is a regular language (not obvious)
- Construct an automaton that accepts viable prefixes

# LR(0) items

- An LR(0) item of a grammar G is a production of G with a special symbol “.” at some position of the right side
- Thus production  $A \rightarrow XYZ$  gives four LR(0) items
  - $A \rightarrow .XYZ$
  - $A \rightarrow X.YZ$
  - $A \rightarrow XY.Z$
  - $A \rightarrow XYZ.$
- An item indicates how much of a production has been seen at a point in the process of parsing
  - Symbols on the left of “.” are already on the stacks
  - Symbols on the right of “.” are expected in the input

# Start state

- Start state of DFA is empty stack corresponding to  $S' \Rightarrow^* S$  item
  - This means no input has been seen
  - The parser expects to see a string derived from  $S$
- Closure of a state adds items for all productions whose LHS occurs in an item in the state, just after “.”
  - Set of possible productions to be reduced next
  - Added items have “.” located at the beginning
  - No symbol of these items is on the stack as yet

# Closure operation

- If  $I$  is a set of items for a grammar  $G$  then  $\text{closure}(I)$  is a set constructed as follows:
  - Every item in  $I$  is in  $\text{closure}(I)$
  - If  $A \rightarrow \alpha.B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production then  $B \rightarrow .\gamma$  is in  $\text{closure}(I)$
- Intuitively  $A \rightarrow \alpha.B\beta$  indicates that we might see a string derivable from  $B\beta$  as input
- If input  $B \rightarrow \gamma$  is a production then we might see a string derivable from  $\gamma$  at this point

# Example

Consider the grammar

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow ( E ) \mid \text{id} \end{aligned}$$

If I is  $\{ E' \rightarrow .E \}$  then closure(I) is

$$\begin{aligned} E' &\rightarrow .E \\ E &\rightarrow .E + T \\ E &\rightarrow .T \\ T &\rightarrow .T^* F \\ T &\rightarrow .F \\ F &\rightarrow .\text{id} \\ F &\rightarrow .(E) \end{aligned}$$

# Applying symbols in a state

- In the new state include all the items that have appropriate input symbol just after the “.”
  -
- Advance “.” in those items and take closure

# Goto operation

- Goto( $I, X$ ) , where  $I$  is a set of items and  $X$  is a grammar symbol,
  - is closure of set of item  $A \rightarrow \alpha X \beta$
  - such that  $A \rightarrow \alpha X \beta$  is in  $I$
- Intuitively if  $I$  is set of items for some valid prefix  $\alpha$  then  $\text{goto}(I, X)$  is set of valid items for prefix  $\alpha X$
- If  $I$  is {  $E' \rightarrow E.$  ,  $E \rightarrow E. + T$  } then  $\text{goto}(I, +)$  is

$E \rightarrow E + .T$

$T \rightarrow .T^* F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

# Sets of items

C : Collection of sets of LR(0) items for grammar G'

$C = \{ \text{closure} (\{ S' \rightarrow .S \}) \}$

repeat

for each set of items I in C

and each grammar symbol X

such that goto (I,X) is not empty and not in C

ADD goto(I,X) to C

until no more additions

# Example

Grammar:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + T \mid T \\ T &\rightarrow T^* F \mid F \\ F &\rightarrow (E) \mid id \end{aligned}$$

$I_0$ : closure( $E' \rightarrow .E$ )

$$\begin{aligned} E' &\rightarrow .E \\ E &\rightarrow .E + T \\ E &\rightarrow .T \\ T &\rightarrow .T^* F \\ T &\rightarrow .F \\ F &\rightarrow .(E) \\ F &\rightarrow .id \end{aligned}$$

$I_1$ : goto( $I_0, E$ )

$$\begin{aligned} E' &\rightarrow E. \\ E &\rightarrow E. + T \end{aligned}$$

$I_2$ : goto( $I_0, T$ )  
 $E \rightarrow T.$   
 $T \rightarrow T. * F$

$I_3$ : goto( $I_0, F$ )  
 $T \rightarrow F.$

$I_4$ : goto( $I_0, ( )$ )  
 $F \rightarrow (.E)$   
 $E \rightarrow .E + T$   
 $E \rightarrow .T$   
 $T \rightarrow .T^* F$   
 $T \rightarrow .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .id$

$I_5$ : goto( $I_0, id$ )  
 $F \rightarrow id.$

$I_6: \text{goto}(I_1, +)$   
 $E \rightarrow E + .T$   
 $T \rightarrow .T * F$   
 $T \rightarrow .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .id$

$I_7: \text{goto}(I_2, *)$   
 $T \rightarrow T * .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .id$

$I_8: \text{goto}(I_4, E)$   
 $F \rightarrow (E.)$   
 $E \rightarrow E. + T$

$\text{goto}(I_4, T) \text{ is } I_2$   
 $\text{goto}(I_4, F) \text{ is } I_3$   
 $\text{goto}(I_4, ()) \text{ is } I_4$   
 $\text{goto}(I_4, id) \text{ is } I_5$

$I_9: \text{goto}(I_6, T)$   
 $E \rightarrow E + T.$   
 $T \rightarrow T. * F$

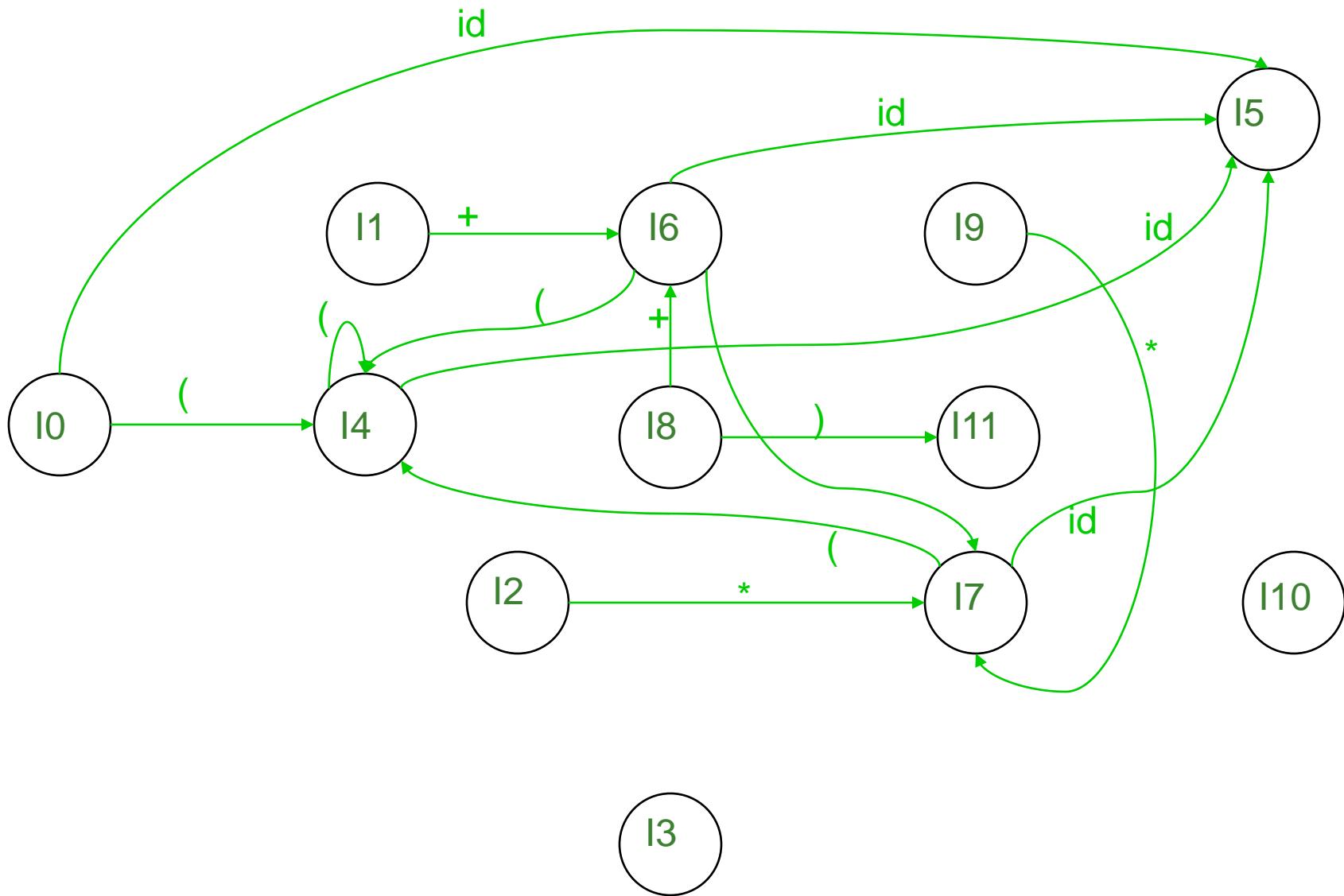
**goto( $I_6, F$ ) is  $I_3$**   
**goto( $I_6, ()$ ) is  $I_4$**   
**goto( $I_6, id$ ) is  $I_5$**

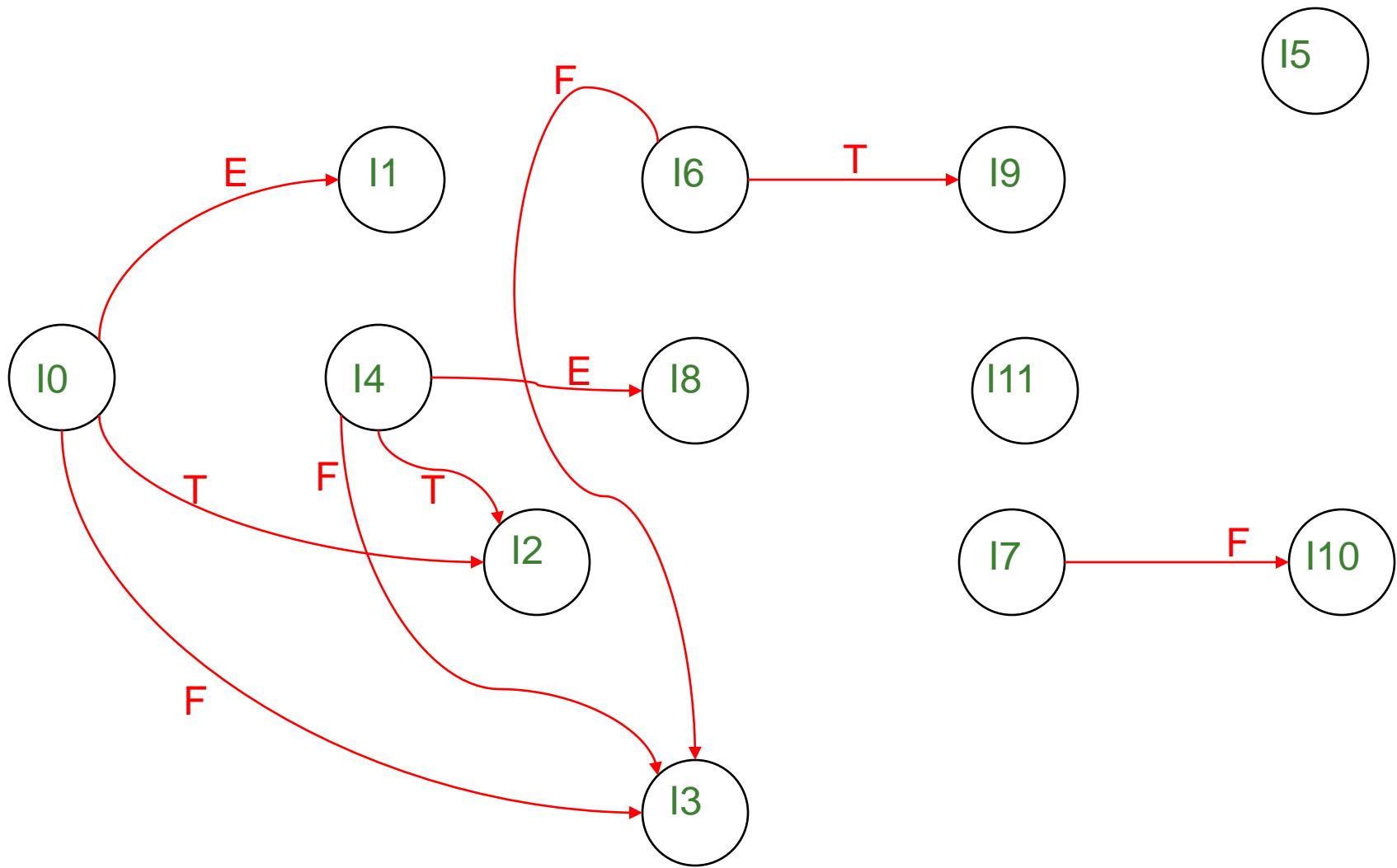
$I_{10}: \text{goto}(I_7, F)$   
 $T \rightarrow T * F.$

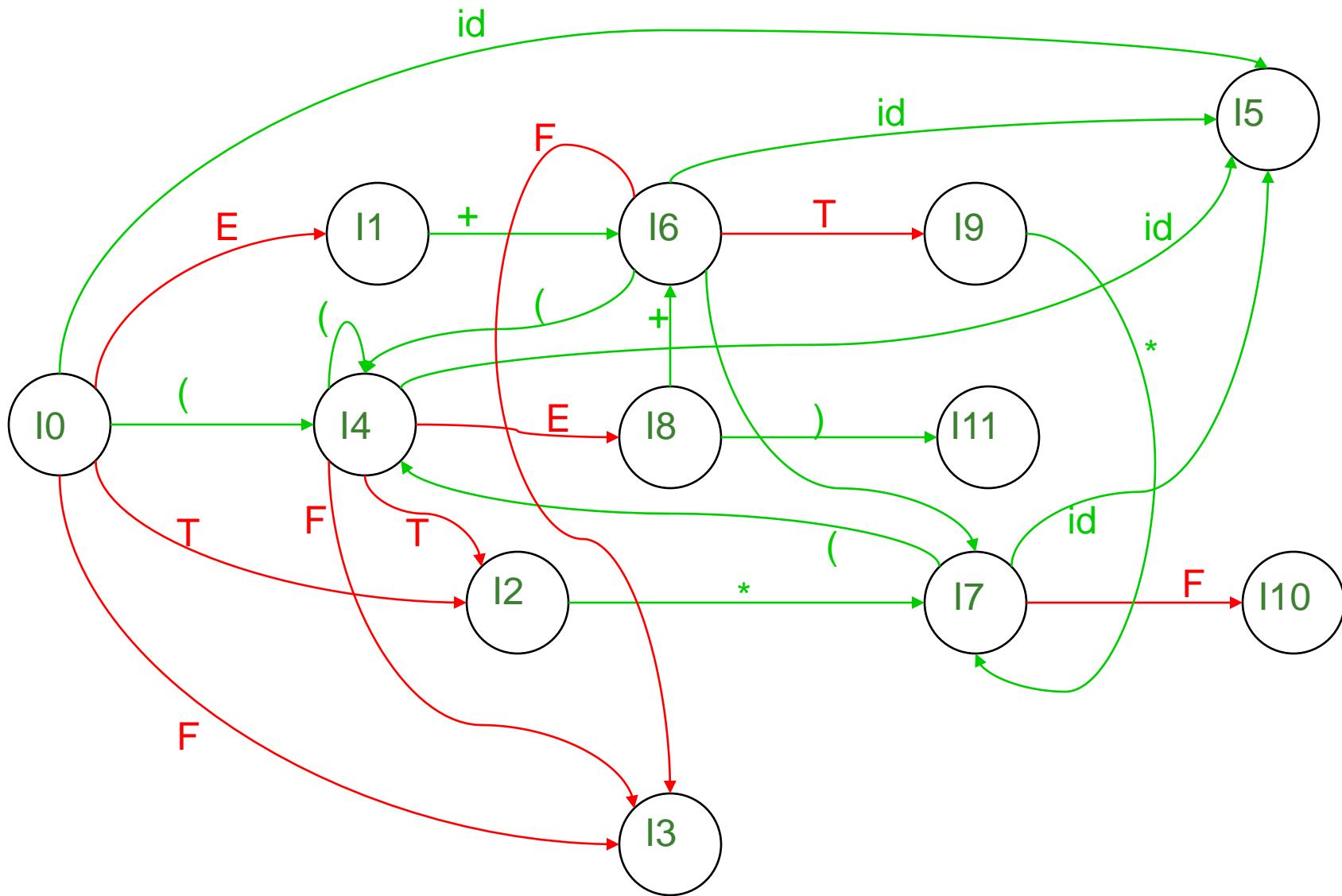
**goto( $I_7, ()$ ) is  $I_4$**   
**goto( $I_7, id$ ) is  $I_5$**

$I_{11}: \text{goto}(I_8, )$   
 $F \rightarrow (E).$

**goto( $I_8, +$ ) is  $I_6$**   
**goto( $I_9, *$ ) is  $I_7$**







# Construct SLR parse table

- Construct  $C = \{I_0, \dots, I_n\}$  the collection of sets of LR(0) items
- If  $A \rightarrow \alpha.a\beta$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$   
then  $\text{action}[i, a] = \text{shift } j$
- If  $A \rightarrow \alpha.$  is in  $I_i$   
then  $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$  for all  $a$  in  $\text{follow}(A)$
- If  $S' \rightarrow S.$  is in  $I_i$  then  $\text{action}[i, \$] = \text{accept}$
- If  $\text{goto}(I_i, A) = I_j$   
then  $\text{goto}[i, A] = j$  for all non terminals  $A$
- All entries not defined are errors

# Notes

- This method of parsing is called SLR (Simple LR)
- LR parsers accept LR( $k$ ) languages
  - L stands for left to right scan of input
  - R stands for rightmost derivation
  - $k$  stands for number of lookahead token
- SLR is the simplest of the LR parsing methods. It is too weak to handle most languages!
- If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous
- All SLR grammars are unambiguous
- Are all unambiguous grammars in SLR?

# Assignment

Construct SLR parse table for following grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid ( E ) \mid \text{digit}$$

Show steps in parsing of string  
 $9^*5+(2+3^*7)$

- Steps to be followed
  - Augment the grammar
  - Construct set of LR(0) items
  - Construct the parse table
  - Show states of parser as the given string is parsed
- Due on todate+5

# Example

- Consider following grammar and its SLR parse table:

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$I_1: goto(I_o, S)$   
 $S' \rightarrow S.$

$I_2: goto(I_o, L)$   
 $S \rightarrow L.=R$   
 $R \rightarrow L.$

$I_o: S' \rightarrow .S$   
 $S \rightarrow .L=R$   
 $S \rightarrow .R$   
 $L \rightarrow .*R$   
 $L \rightarrow .id$   
 $R \rightarrow .L$

**Assignment (not to be submitted):**  
**Construct rest of the items and the parse table.**

## SLR parse table for the grammar

	=	*	id	\$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6,r6			r6			
3				r3			
4		s4	s5			8	7
5	r5			r5			
6		s4	s5			8	9
7	r4			r4			
8	r6			r6			
9				r2			

The table has multiple entries in action[2,=]

- There is both a shift and a reduce entry in action[2,=]. Therefore state 2 has a shift-reduce conflict on symbol “=”, However, the grammar is not ambiguous.

- Parse id=id assuming reduce action is taken in [2,=]

Stack	input	action
0	id=id	shift 5
0 id 5	=id	reduce by L→id
0 L 2	=id	reduce by R→L
0 R 3	=id	error

- if shift action is taken in [2,=]

Stack	input	action
0	id=id\$	shift 5
0 id 5	=id\$	reduce by L→id
0 L 2	=id\$	shift 6
0 L 2 = 6	id\$	shift 5
0 L 2 = 6 id 5	\$	reduce by L→id
0 L 2 = 6 L 8	\$	reduce by R→L
0 L 2 = 6 R 9	\$	reduce by S→L=R
0 S 1	\$	ACCEPT

# Problems in SLR parsing

- No sentential form of this grammar can start with  $R=...$
- However, the reduce action in  $\text{action}[2,=]$  generates a sentential form starting with  $R=$
- Therefore, the reduce action is incorrect
- In SLR parsing method state  $i$  calls for reduction on symbol “a”, by rule  $A \rightarrow \alpha$  if  $I_i$  contains  $[A \rightarrow \alpha.]$  and “a” is in  $\text{follow}(A)$
- However, when state  $I$  appears on the top of the stack, the viable prefix  $\beta\alpha$  on the stack may be such that  $\beta A$  can not be followed by symbol “a” in any right sentential form
- Thus, the reduction by the rule  $A \rightarrow \alpha$  on symbol “a” is invalid
- SLR parsers can not remember the left context

# Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by  $A \rightarrow \alpha$  will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol)
- The general form of the item becomes  $[A \rightarrow \alpha.\beta, a]$  which is called LR(1) item.
- Item  $[A \rightarrow \alpha., a]$  calls for reduction only if next input is a. The set of symbols "a"s will be a subset of  $\text{Follow}(A)$ .

# Closure(I)

repeat

    for each item  $[A \rightarrow \alpha.B\beta, a]$  in I

        for each production  $B \rightarrow \gamma$  in  $G'$

        and for each terminal b in  $\text{First}(\beta a)$

            add item  $[B \rightarrow .\gamma, b]$  to I

until no more additions to I

# Example

Consider the following grammar

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow CC \\ C &\rightarrow cC \mid d \end{aligned}$$

Compute closure(I) where I= {[S' → .S, \$]}

$$\begin{array}{ll} S' \rightarrow .S, & \$ \\ S \rightarrow .CC, & \$ \\ C \rightarrow .cC, & c \\ C \rightarrow .cC, & d \\ C \rightarrow .d, & c \\ C \rightarrow .d, & d \end{array}$$

# Example

Construct sets of LR(1) items for the grammar on previous slide

$I_0: S' \rightarrow .S,$   
 $S \rightarrow .CC,$   
 $C \rightarrow .cC,$   
 $C \rightarrow .d,$

\$  
\$  
c/d  
c/d

$I_1: \text{ goto}(I_0, S)$   
 $S' \rightarrow S.,$

\$

$I_2: \text{ goto}(I_0, C)$   
 $S \rightarrow C.C,$   
 $C \rightarrow .cC,$   
 $C \rightarrow .d,$

\$  
\$  
\$

$I_3: \text{ goto}(I_0, c)$   
 $C \rightarrow c.C,$   
 $C \rightarrow .cC,$   
 $C \rightarrow .d,$

c/d  
c/d  
c/d

$I_4: \text{ goto}(I_0, d)$   
 $C \rightarrow d.,$

c/d

$I_5: \text{ goto}(I_2, C)$   
 $S \rightarrow CC.,$

\$

$I_6: \text{ goto}(I_2, c)$   
 $C \rightarrow c.C,$   
 $C \rightarrow .cC,$   
 $C \rightarrow .d,$

\$  
\$  
\$

$I_7: \text{ goto}(I_2, d)$   
 $C \rightarrow d.,$

\$

$I_8: \text{ goto}(I_3, C)$   
 $C \rightarrow cC.,$

c/d

$I_9: \text{ goto}(I_6, C)$   
 $C \rightarrow cC.,$

\$

# Construction of Canonical LR parse table

- Construct  $C = \{I_0, \dots, I_n\}$  the sets of LR(1) items.
- If  $[A \rightarrow \alpha.a\beta, b]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a] = \text{shift } j$
- If  $[A \rightarrow \alpha., a]$  is in  $I_i$  then  $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$
- If  $[S' \rightarrow S., \$]$  is in  $I_i$  then  $\text{action}[i, \$] = \text{accept}$
- If  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$  for all non terminals  $A$

# Parse table

State	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

# Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is  $c^*dc^*d$ .
- When reading input  $cc\dots dc\dots d$  the parser shifts  $cs$  into stack and then goes into state 4 after reading  $d$ . It then calls for reduction by  $C \rightarrow d$  if following symbol is  $c$  or  $d$ .
- IF  $\$$  follows the first  $d$  then input string is  $c^*d$  which is not in the language; parser declares an error
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error
- **Problem:** Canonical LR parse table has a large number of states

# LALR Parse table

- Look Ahead LR parsers
- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items  
 $I_4: C \rightarrow d. , c/d$        $I_7: C \rightarrow d., \$$
- Replace  $I_4$  and  $I_7$  by a new state  $I_{47}$  consisting of  $(C \rightarrow d., c/d/\$)$
- Similarly  $I_3 & I_6$  and  $I_8 & I_9$  form pairs
- Merge LR(1) items having the same core

# Construct LALR parse table

- Construct  $C = \{I_0, \dots, I_n\}$  set of LR(1) items
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union
- Let  $C' = \{J_0, \dots, J_m\}$  be the resulting set of items
- Construct action table as was done earlier
- Let  $J = I_1 \cup I_2 \cup \dots \cup I_k$

since  $I_1, I_2, \dots, I_k$  have same core,  $\text{goto}(J, X)$  will have the same core

Let  $K = \text{goto}(I_1, X) \cup \text{goto}(I_2, X) \cup \dots \cup \text{goto}(I_k, X)$  then  $\text{goto}(J, X) = K$

# LALR parse table ...

State	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

# Notes on LALR parse table

- Modified parser behaves as original except that it will reduce  $C \rightarrow d$  on inputs like ccd. The error will eventually be caught before any more symbols are shifted.
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
- SLR and LALR parse tables have same number of states.

# Notes on LALR parse table...

- Merging items may result into conflicts in LALR parsers which did not exist in LR parsers
- New conflicts can not be of shift reduce kind:
  - Assume there is a shift reduce conflict in some state of LALR parser with items $\{[X \rightarrow \alpha., a], [Y \rightarrow \gamma.a\beta, b]\}$
  - Then there must have been a state in the LR parser with the same core
  - Contradiction; because LR parser did not have conflicts
- LALR parser can have new reduce-reduce conflicts
  - Assume states $\{[X \rightarrow \alpha., a], [Y \rightarrow \beta., b]\}$  and $\{[X \rightarrow \alpha., b], [Y \rightarrow \beta., a]\}$
  - Merging the two states produces $\{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., a/b]\}$

# Notes on LALR parse table...

- LALR parsers are not built by first making canonical LR parse tables
- There are direct, complicated but efficient algorithms to develop LALR parsers
- Relative power of various classes
  - $\text{SLR}(1) \leq \text{LALR}(1) \leq \text{LR}(1)$
  - $\text{SLR}(k) \leq \text{LALR}(k) \leq \text{LR}(k)$
  - $\text{LL}(k) \leq \text{LR}(k)$

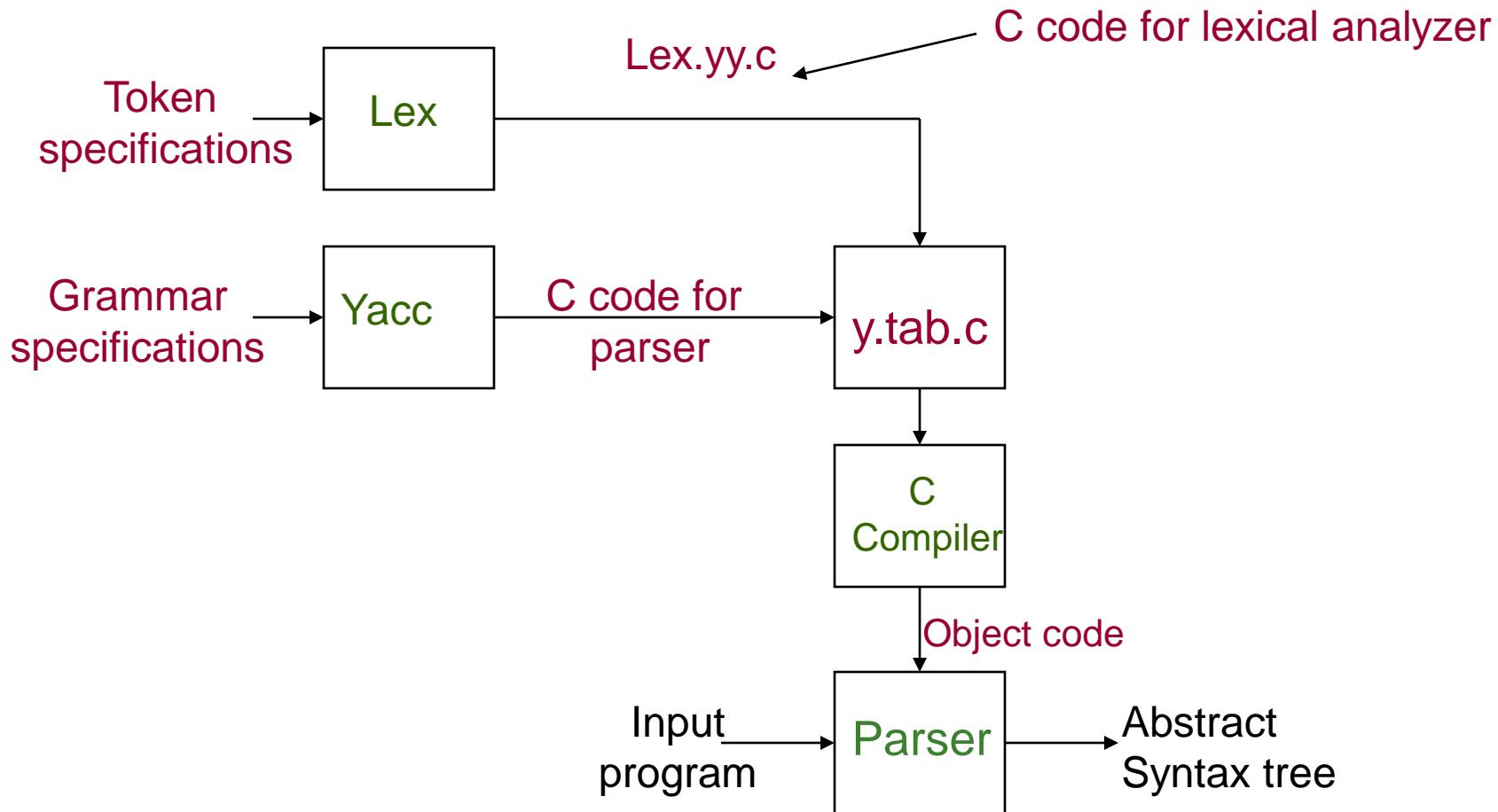
# Error Recovery

- An error is detected when an entry in the action table is found to be empty.
- Panic mode error recovery can be implemented as follows:
  - scan down the stack until a state S with a goto on a particular nonterminal A is found.
  - discard zero or more input symbols until a symbol a is found that can legitimately follow A.
  - stack the state  $\text{goto}[S, A]$  and resume parsing.
- Choice of A: Normally these are non terminals representing major program pieces such as an expression, statement or a block. For example if A is the nonterminal stmt, a might be semicolon or end.

# Parser Generator

- Some common parser generators
  - YACC: Yet Another Compiler Compiler
  - Bison: GNU Software
  - ANTLR: ANother Tool for Language Recognition
- Yacc/Bison source program specification (accept LALR grammars)
  - declaration
  - %%
  - translation rules
  - %%
  - supporting C routines

# Yacc and Lex schema



Refer to YACC Manual

# References

- Compiler Design by Amey Karkare, IIT Kanpur  
<https://karkare.github.io/cs335/>
- Compilers: Principles, Techniques, and Tools, Second edition, 2006. by Alfred V. Aho , Monica S. Lam , Ravi Sethi , Jeffrey D. Ullman

# Bottom up parsing

- Construct a parse tree for an input string beginning at leaves and going towards root OR
- Reduce a string w of input to start symbol of grammar  
Consider a grammar

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \quad | \quad b \\ B &\rightarrow d \end{aligned}$$

And reduction of a string

a b b c d e  
a A b c d e  
a A d e  
a A B e  
S

The sentential forms happen to be a *right most derivation in the reverse order.*

$\begin{aligned} S &\rightarrow a A \underline{B} e \\ &\rightarrow a A \underline{d} e \\ &\rightarrow a A \underline{b} c d e \\ &\rightarrow a b b c d e \end{aligned}$

# Shift reduce parsing

- Split string being parsed into two parts
  - Two parts are separated by a special character “.”
  - Left part is a string of terminals and non terminals
  - Right part is a string of terminals
- Initially the input is .w

# Shift reduce parsing ...

- Bottom up parsing has two actions
- Shift: move terminal symbol from right string to left string
  - if string before shift is      a.pqr
  - then string after shift is    ap.qr

# Shift reduce parsing ...

- **Reduce:** immediately on the left of “.” identify a string same as RHS of a production and replace it by LHS

if string before reduce action is  $\alpha\beta.pqr$   
and  $A \rightarrow \beta$  is a production  
then string after reduction is  $\alpha A.pqr$

# Example

Assume grammar is  $E \rightarrow E+E \mid E^*E \mid id$

Parse  $id^*id+id$

Assume an oracle tells you when to shift / when to reduce

<b>String</b>	<b>action (by oracle)</b>
. $id^*id+id$	shift
$id.^*id+id$	reduce $E \rightarrow id$
$E.^*id+id$	shift
$E^*.id+id$	shift
$E^*id.+id$	reduce $E \rightarrow id$
$E^*E.+id$	reduce $E \rightarrow E^*E$
$E.+id$	shift
$E+.id$	shift
$E+id.$	Reduce $E \rightarrow id$
$E+E.$	Reduce $E \rightarrow E+E$
$E.$	ACCEPT

# Shift reduce parsing ...

- Symbols on the left of “.” are kept on a stack
  - Top of the stack is at “.”
  - Shift pushes a terminal on the stack
  - Reduce pops symbols (rhs of production) and pushes a non terminal (lhs of production) onto the stack
- The most important issue: when to shift and when to reduce
- Reduce action should be taken only if the result can be reduced to the start symbol

# Issues in bottom up parsing

- How do we know which action to take
  - whether to shift or reduce
  - Which production to use for reduction?
- Sometimes parser can reduce but it should not:  
 $x \rightarrow \epsilon$  can always be used for reduction!

# Issues in bottom up parsing

- Sometimes parser can reduce in different ways!
- Given stack  $\delta$  and input symbol  $a$ , should the parser
  - Shift  $a$  onto stack (making it  $\delta a$ )
  - Reduce by some production  $A \rightarrow \beta$  assuming that stack has form  $\alpha\beta$  (making it  $\alpha A$ )
  - Stack can have many combinations of  $\alpha\beta$
  - How to keep track of length of  $\beta$ ?

# Handles

- The basic steps of a bottom-up parser are
  - to identify a *substring* within a *rightmost sentential* form which matches the RHS of a rule.
  - when this substring is replaced by the LHS of the matching rule, it must produce the previous rightmost-sentential form.
- Such a substring is called a *handle*

# Handle

- A *handle* of a right sentential form  $\gamma$  is
  - a production rule  $A \rightarrow \beta$ , and
  - an occurrence of a sub-string  $\beta$  in  $\gamma$
- such that
- when the occurrence of  $\beta$  is replaced by  $A$  in  $\gamma$ , we get the previous right sentential form in a rightmost derivation of  $\gamma$ .

# Handle

Formally, if

$$S \xrightarrow{rm^*} \alpha Aw \xrightarrow{rm} \alpha\beta w,$$

then

- $\beta$  in the position following  $\alpha$ ,
- and the corresponding production  $A \rightarrow \beta$  is a handle of  $\alpha\beta w$ .
- The string  $w$  consists of only terminal symbols

# Handle

- We only want to reduce handle and not any RHS
- Handle pruning: If  $\beta$  is a handle and  $A \rightarrow \beta$  is a production then replace  $\beta$  by  $A$
- A right most derivation in reverse can be obtained by handle pruning.

## Handle: Observation

- *Only terminal symbols can appear to the right of a handle in a rightmost sentential form.*
- Why?

# Handle: Observation

Is this scenario possible:

- $\alpha\beta\gamma$  is the content of the stack
- $A \rightarrow \gamma$  is a handle
- The stack content reduces to  $\alpha\beta A$
- Now  $B \rightarrow \beta$  is the handle

In other words, handle is not on top, but buried *inside* stack

Not Possible! Why?

# Handles ...

- Consider two cases of right most derivation to understand the fact that handle appears on the top of the stack

$$S \rightarrow \alpha A z \rightarrow \alpha \beta B y z \rightarrow \alpha \beta \gamma y z$$

$$S \rightarrow \alpha B x A z \rightarrow \alpha B x y z \rightarrow \alpha \gamma x y z$$

# Handle always appears on the top

Case I:  $S \rightarrow \alpha A z \rightarrow \alpha \beta B y z \rightarrow \alpha \beta \gamma y z$

stack	input	action
$\alpha \beta \gamma$	yz	reduce by $B \rightarrow \gamma$
$\alpha \beta B$	yz	shift y
$\alpha \beta B y$	z	reduce by $A \rightarrow \beta B y$
$\alpha A$	z	

Case II:  $S \rightarrow \alpha B x A z \rightarrow \alpha B x y z \rightarrow \alpha y x y z$

stack	input	action
$\alpha \gamma$	xyz	reduce by $B \rightarrow \gamma$
$\alpha B$	xyz	shift x
$\alpha B x$	yz	shift y
$\alpha B x y$	z	reduce $A \rightarrow y$
$\alpha B x A$	z	

# Shift Reduce Parsers

- The general shift-reduce technique is:
  - if there is no handle on the stack then shift
  - If there is a handle then reduce
- Bottom up parsing is essentially the process of detecting handles and reducing them.
- Different bottom-up parsers differ in the way they detect handles.

# Conflicts

- What happens when there is a choice
  - What action to take in case both shift and reduce are valid?  
**shift-reduce conflict**
  - Which rule to use for reduction if reduction is possible by more than one rule?  
**reduce-reduce conflict**

# Conflicts

- Conflicts come either because of ambiguous grammars or parsing method is not powerful enough

# Shift reduce conflict

Consider the grammar  $E \rightarrow E+E \mid E^*E \mid id$   
and the input  $id+id^*id$

stack	input	action
$E+E$	$*id$	reduce by $E \rightarrow E+E$
$E$	$*id$	shift
$E^*$	$id$	shift
$E^*id$		reduce by $E \rightarrow id$
$E^*E$		reduce by $E \rightarrow E^*E$
$E$		

stack	input	action
$E+E$	$*id$	shift
$E+E^*$	$id$	shift
$E+E^*id$		reduce by $E \rightarrow id$
$E+E^*E$		reduce by $E \rightarrow E^*E$
$E+E$		reduce by $E \rightarrow E+E$
$E$		

# Reduce reduce conflict

Consider the grammar  $M \rightarrow R+R \mid R+c \mid R$   
 $R \rightarrow c$

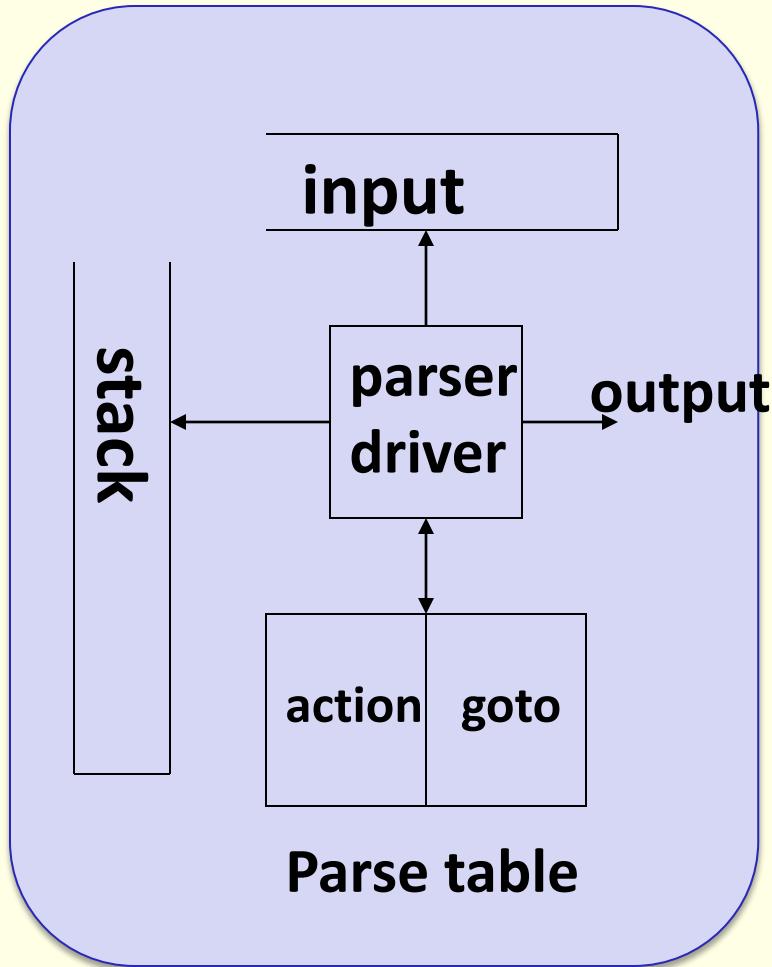
and the input

c+c

Stack	input	action
	c+c	shift
c	+c	reduce by $R \rightarrow c$
R	+c	shift
R+	c	shift
R+c		reduce by $R \rightarrow c$
R+R		reduce by $M \rightarrow R+R$
M		

Stack	input	action
	c+c	shift
c	+c	reduce by $R \rightarrow c$
R	+c	shift
R+	c	shift
R+c		reduce by $M \rightarrow R+R$
M		

# LR parsing



- Input buffer contains the input string.
- Stack contains a string of the form  $S_0X_1S_1X_2.....X_nS_n$  where each  $X_i$  is a grammar symbol and each  $S_i$  is a state.
- Table contains action and goto parts.
- action table is indexed by state and terminal symbols.
- goto table is indexed by state and non terminal symbols.

# Example

Consider a grammar  
and its parse table

$$\begin{array}{l}
 E \rightarrow E + T \quad | \quad T \\
 T \rightarrow T * F \quad | \quad F \\
 F \rightarrow ( E ) \quad | \quad id
 \end{array}$$

State	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				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			

action

goto

# Actions in an LR (shift reduce) parser

- Assume  $S_i$  is top of stack and  $a_i$  is current input symbol
- Action  $[S_i, a_i]$  can have four values
  1. sj: shift  $a_i$  to the stack, goto state  $S_j$
  2. rk: reduce by rule number k
  3. acc: Accept
  4. err: Error (empty cells in the table)

# Driving the LR parser

Stack:  $S_0 X_1 S_1 X_2 \dots X_m S_m$       Input:  $a_i a_{i+1} \dots a_n \$$

- If  $\text{action}[S_m, a_i] = \text{shift } S$

Then the configuration becomes

Stack:  $S_0 X_1 S_1 \dots \dots X_m S_m a_i S$       Input:  $a_{i+1} \dots a_n \$$

- If  $\text{action}[S_m, a_i] = \text{reduce } A \rightarrow \beta$

Then the configuration becomes

Stack:  $S_0 X_1 S_1 \dots X_{m-r} S_{m-r} AS$       Input:  $a_i a_{i+1} \dots a_n \$$

Where  $r = |\beta|$  and  $S = \text{goto}[S_{m-r}, A]$

# Driving the LR parser

Stack:  $S_0 X_1 S_1 X_2 \dots X_m S_m$     Input:  $a_i a_{i+1} \dots a_n \$$

- If  $\text{action}[S_m, a_i] = \text{accept}$   
Then parsing is completed. HALT
- If  $\text{action}[S_m, a_i] = \text{error}$  (or empty cell)  
Then invoke error recovery routine.

# Parse id + id \* id

Stack	Input	Action
0	id+id*id\$	shift 5
0 id 5	+id*id\$	reduce by F→id
0 F 3	+id*id\$	reduce by T→F
0 T 2	+id*id\$	reduce by E→T
0 E 1	+id*id\$	shift 6
0 E 1 + 6	id*id\$	shift 5
0 E 1 + 6 id 5	*id\$	reduce by F→id
0 E 1 + 6 F 3	*id\$	reduce by T→F
0 E 1 + 6 T 9	*id\$	shift 7
0 E 1 + 6 T 9 * 7 id\$		shift 5
0 E 1 + 6 T 9 * 7 id 5	\$	reduce by F→id
0 E 1 + 6 T 9 * 7 F 10	\$	reduce by T→T*F
0 E 1 + 6 T 9	\$	reduce by E→E+T
0 E 1	\$	ACCEPT

# Configuration of a LR parser

- The tuple  
 $\langle \text{Stack Contents}, \text{Remaining Input} \rangle$   
defines a *configuration* of a LR parser
- Initially the configuration is  
 $\langle S_0, a_0 a_1 \dots a_n \$ \rangle$
- Typical final configuration on a successful parse is  
 $\langle S_0 X_1 S_i, \$ \rangle$

# LR parsing Algorithm

Initial state:      Stack:  $S_0$       Input:  $w\$$

```
while (1) {
    if (action[S,a] = shift S') {
        push(a); push(S'); ip++
    } else if (action[S,a] = reduce A → β) {
        pop (2*|β|) symbols;
        push(A); push (goto[S'',A])
        ( $S''$  is the state at stack top after popping symbols)
    } else if (action[S,a] = accept) {
        exit
    } else { error }
```

# Constructing parse table

## Augment the grammar

- $G$  is a grammar with start symbol  $S$
- The augmented grammar  $G'$  for  $G$  has a new start symbol  $S'$  and an additional production  $S' \rightarrow S$
- When the parser reduces by this rule it will stop with accept

# Production to Use for Reduction

- How do we know which production to apply in a given configuration
- We can guess!
  - May require backtracking
- Keep track of “ALL” possible rules that can apply at a given point in the input string
  - But in general, there is no upper bound on the length of the input string
  - Is there a bound on number of applicable rules?

# Some hands on!

- $E' \rightarrow E$  Strings to Parse
- $E \rightarrow E + T$ 
  - id + id + id + id
- $E \rightarrow T$ 
  - id \* id \* id \* id
- $T \rightarrow T * F$ 
  - id \* id + id \* id
- $T \rightarrow F$ 
  - id \* (id + id) \* id
- $F \rightarrow (E)$
- $F \rightarrow id$

# Parser states

- Goal is to know the valid reductions at any given point
- Summarize all possible stack prefixes  $\alpha$  as a parser state
- Parser state is defined by a DFA state that reads in the stack  $\alpha$
- Accept states of DFA are unique reductions

# Viable prefixes

- $\alpha$  is a viable prefix of the grammar if
  - $\exists w$  such that  $\alpha w$  is a right sentential form
  - $\langle \alpha, w \rangle$  is a configuration of the parser
- As long as the parser has viable prefixes on the stack no parser error has been seen
- The set of viable prefixes is a regular language
- We can construct an automaton that accepts viable prefixes

# LR(0) items

- An LR(0) item of a grammar G is a production of G with a special symbol “.” at some position of the right side
- Thus production  $A \rightarrow XYZ$  gives four LR(0) items

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

# LR(0) items

- An item indicates how much of a production has been seen at a point in the process of parsing
  - Symbols on the left of “.” are already on the stacks
  - Symbols on the right of “.” are expected in the input

# Start state

- Start state of DFA is an empty stack corresponding to  $S' \xrightarrow{} .S$  item
- This means no input has been seen
- The parser expects to see a string derived from S

# Closure of a state

- **Closure** of a state adds items for all productions whose LHS occurs in an item in the state, just after “.”
  - Set of possible productions to be reduced next
  - Added items have “.” located at the beginning
  - No symbol of these items is on the stack as yet

# Closure operation

- Let  $I$  be a set of items for a grammar  $G$
- $\text{closure}(I)$  is a set constructed as follows:
  - Every item in  $I$  is in  $\text{closure}(I)$
  - If  $A \rightarrow \alpha.B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production then  $B \rightarrow .\gamma$  is in  $\text{closure}(I)$
- Intuitively  $A \rightarrow \alpha.B\beta$  indicates that we expect a string derivable from  $B\beta$  in input
- If  $B \rightarrow \gamma$  is a production then we might see a string derivable from  $\gamma$  at this point

# Example

For the grammar

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow ( E ) \mid id$$

If I is {  $E' \rightarrow .E$  } then  
closure(I) is

$$E' \rightarrow .E$$

$$E \rightarrow .E + T$$

$$E \rightarrow .T$$

$$T \rightarrow .T * F$$

$$T \rightarrow .F$$

$$F \rightarrow .id$$

$$F \rightarrow .(E)$$

# Goto operation

- $\text{Goto}(I, X)$  , where  $I$  is a set of items and  $X$  is a grammar symbol,
  - is closure of set of item  $A \rightarrow \alpha X . \beta$
  - such that  $A \rightarrow \alpha . X \beta$  is in  $I$
- Intuitively if  $I$  is a set of items for some valid prefix  $\alpha$  then  $\text{goto}(I, X)$  is set of valid items for prefix  $\alpha X$

# Goto operation

If I is {  $E' \rightarrow E.$  ,  $E \rightarrow E. + T$  } then  
goto(I,+) is

$E \rightarrow E + .T$

$T \rightarrow .T * F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

# Sets of items

C : Collection of sets of LR(0) items for grammar G'

$$C = \{ \text{closure} (\{ S' \rightarrow .S \}) \}$$

repeat

for each set of items I in C

for each grammar symbol X

if goto (I,X) is not empty and not in C

ADD goto(I,X) to C

until no more additions to C

# Example

Grammar:

$$E' \rightarrow E$$

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T^* F \mid F$$

$$F \rightarrow (E) \mid id$$

$I_0$ : closure( $E' \rightarrow .E$ )

$$E' \rightarrow .E$$

$$E \rightarrow .E + T$$

$$E \rightarrow .T$$

$$T \rightarrow .T^* F$$

$$T \rightarrow .F$$

$$F \rightarrow .(E)$$

$$F \rightarrow .id$$

$I_1$ : goto( $I_0, E$ )

$$E' \rightarrow E.$$

$$E \rightarrow E. + T$$

$I_2$ : goto( $I_0, T$ )

$$E \rightarrow T.$$

$$T \rightarrow T. * F$$

$I_3$ : goto( $I_0, F$ )

$$T \rightarrow F.$$

$I_4$ : goto( $I_0, ( )$ )

$$F \rightarrow (.E)$$

$$E \rightarrow .E + T$$

$$E \rightarrow .T$$

$$T \rightarrow .T^* F$$

$$T \rightarrow .F$$

$$F \rightarrow .(E)$$

$$F \rightarrow .id$$

$I_5$ : goto( $I_0, id$ )

$$F \rightarrow id.$$

$I_6: \text{goto}(I_1, +)$   
 $E \rightarrow E + .T$   
 $T \rightarrow .T * F$   
 $T \rightarrow .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .\text{id}$

$I_7: \text{goto}(I_2, *)$   
 $T \rightarrow T * .F$   
 $F \rightarrow .(E)$   
 $F \rightarrow .\text{id}$

$I_8: \text{goto}(I_4, E)$   
 $F \rightarrow (E.)$   
 $E \rightarrow E. + T$

$\text{goto}(I_4, T) \text{ is } I_2$   
 $\text{goto}(I_4, F) \text{ is } I_3$   
 $\text{goto}(I_4, ( ) \text{ is } I_4$   
 $\text{goto}(I_4, \text{id}) \text{ is } I_5$

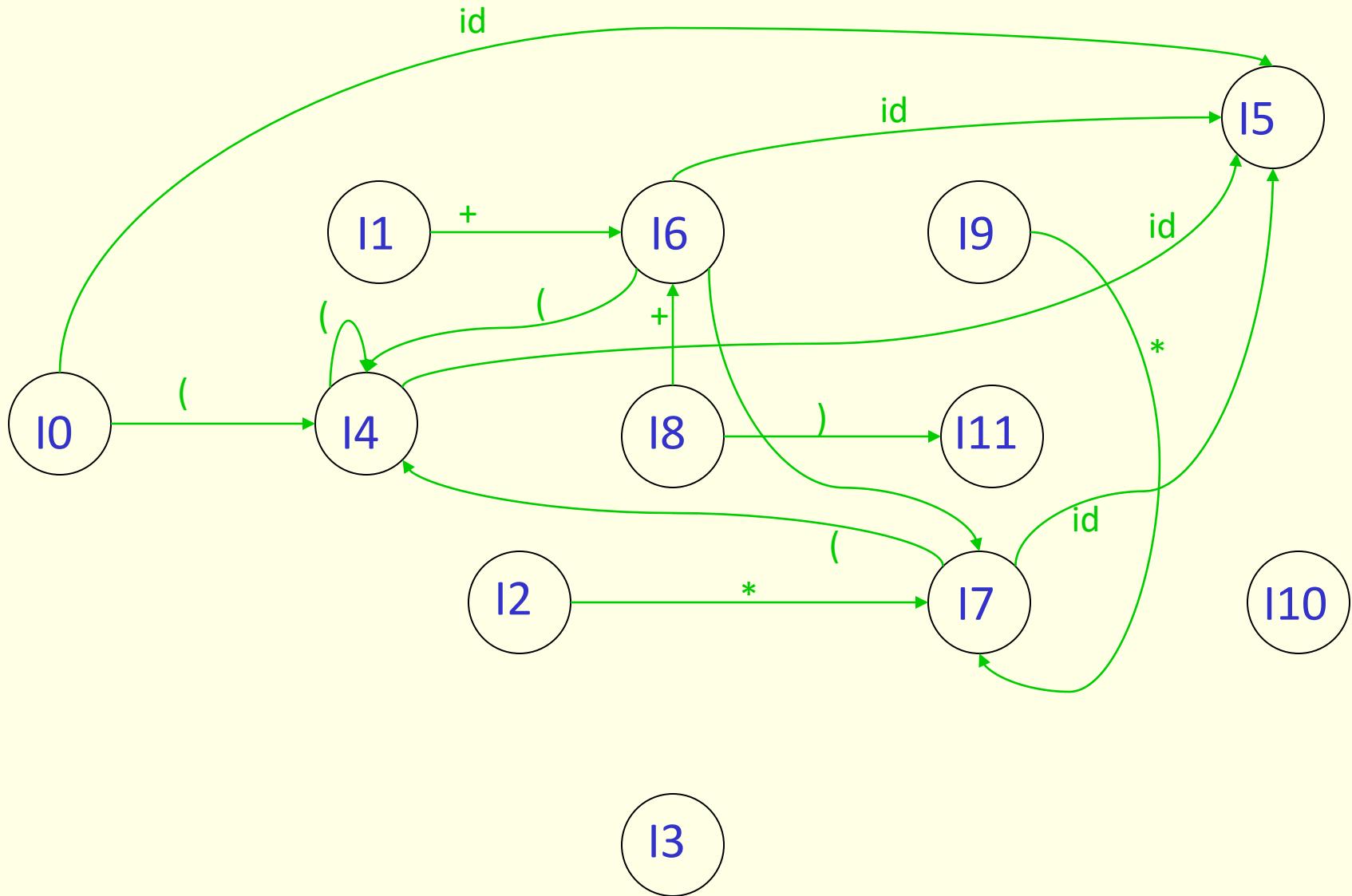
$I_9: \text{goto}(I_6, T)$   
 $E \rightarrow E + T.$   
 $T \rightarrow T. * F$   
  
 $\text{goto}(I_6, F) \text{ is } I_3$   
 $\text{goto}(I_6, ( ) \text{ is } I_4$   
 $\text{goto}(I_6, \text{id}) \text{ is } I_5$

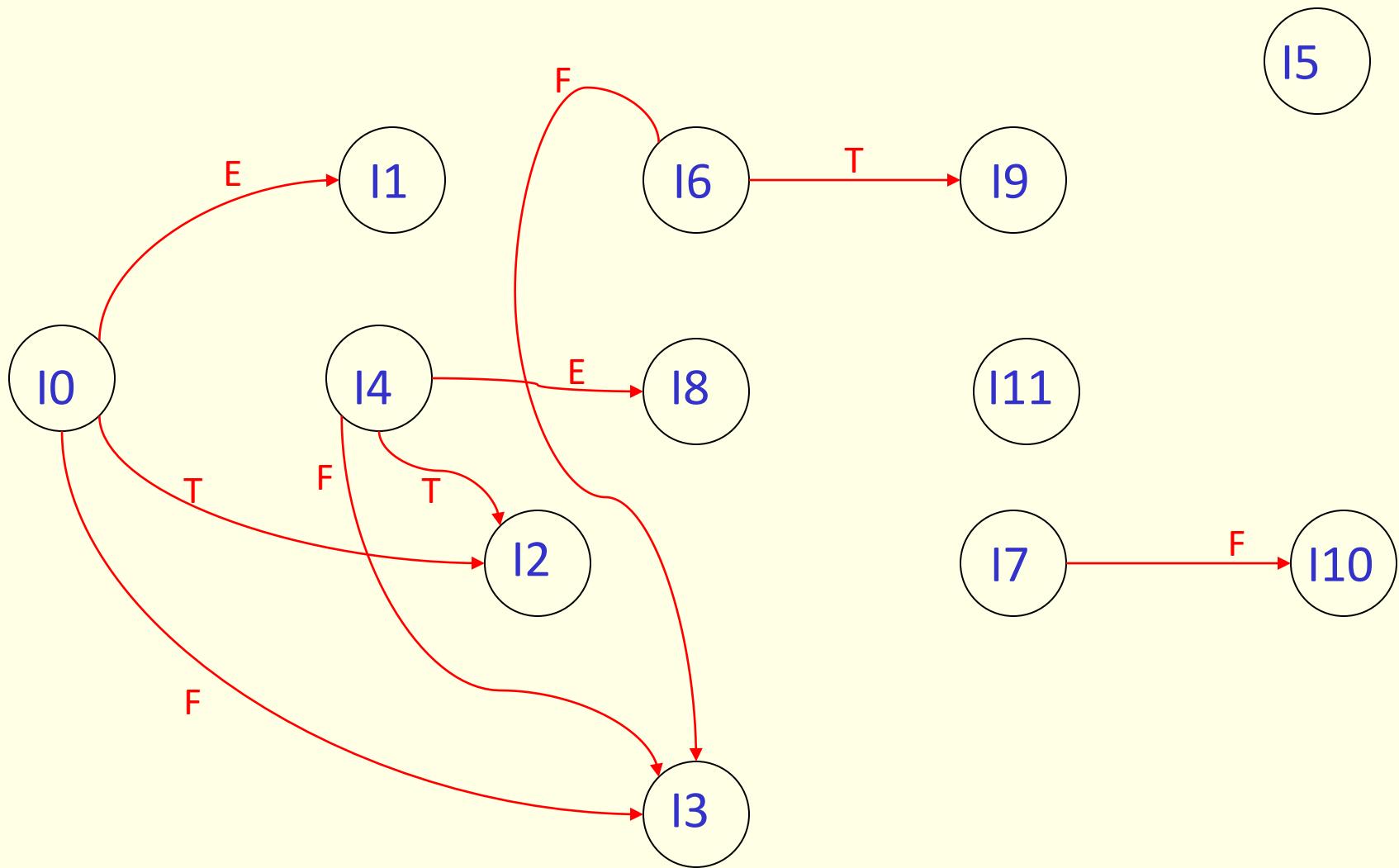
$I_{10}: \text{goto}(I_7, F)$   
 $T \rightarrow T * F.$

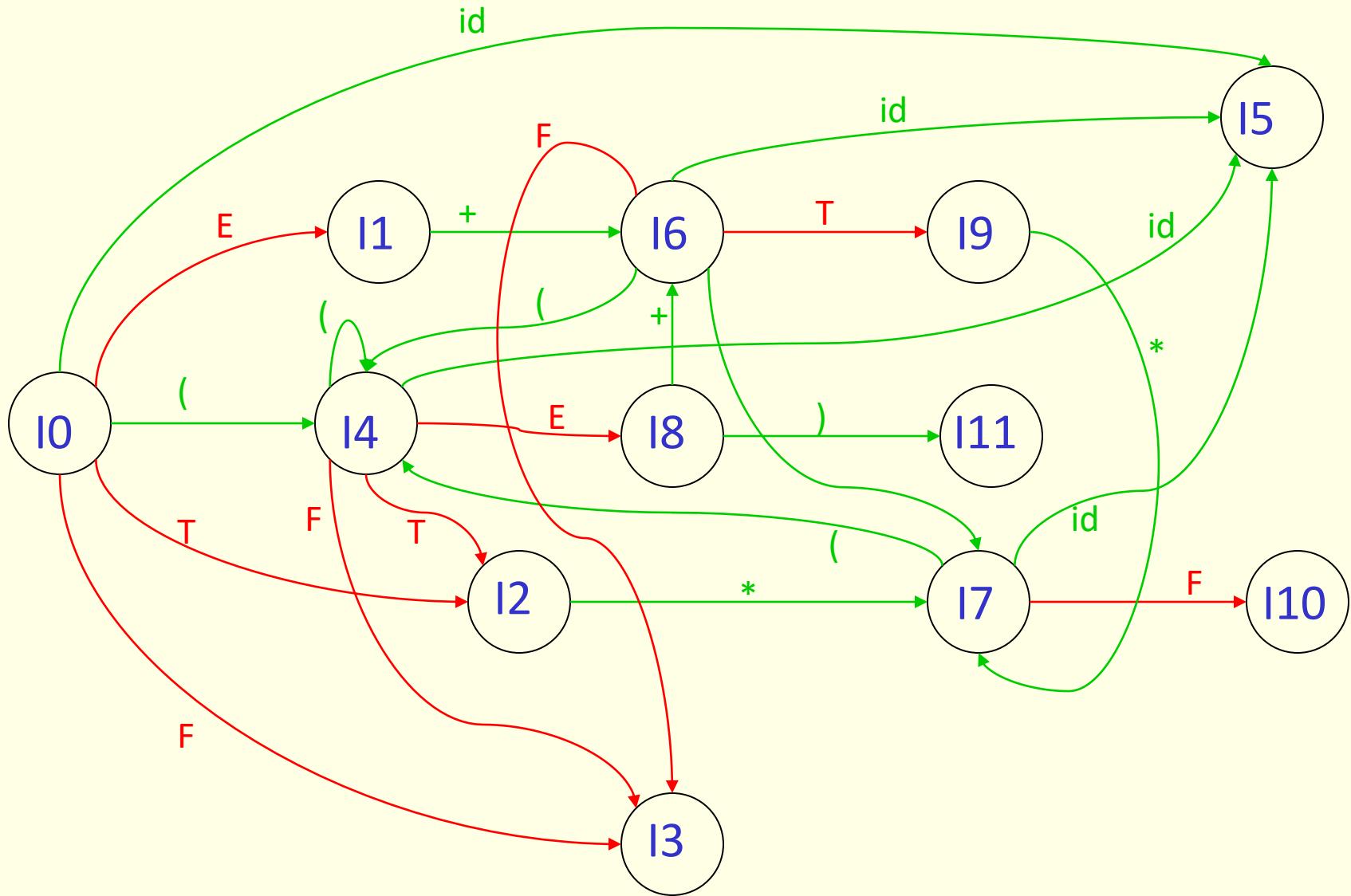
$\text{goto}(I_7, ( ) \text{ is } I_4$   
 $\text{goto}(I_7, \text{id}) \text{ is } I_5$

$I_{11}: \text{goto}(I_8, )$   
 $F \rightarrow (E).$

$\text{goto}(I_8, +) \text{ is } I_6$   
 $\text{goto}(I_9, *) \text{ is } I_7$







## LR(0) (?) Parse Table

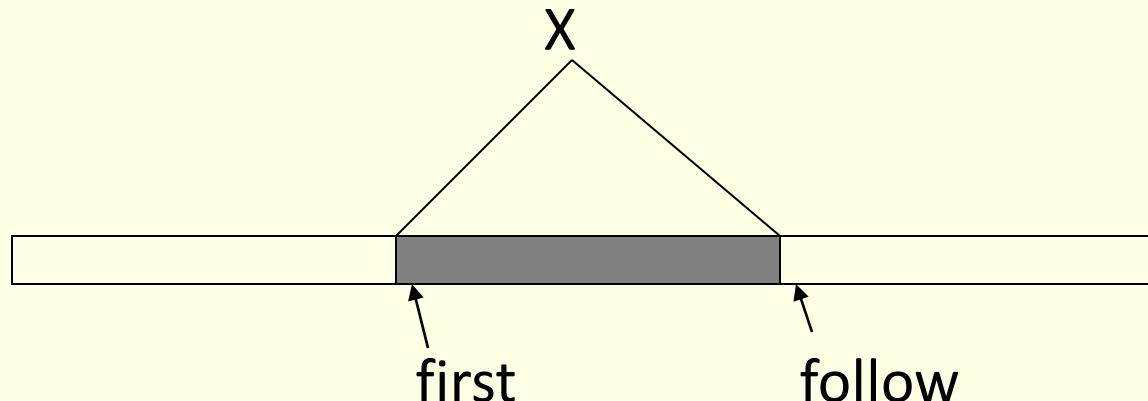
- The information is still not sufficient to help us resolve shift-reduce conflict.  
For example the state:

$$\begin{aligned}I_1: E' &\rightarrow E. \\E &\rightarrow E. + T\end{aligned}$$

- We need some more information to make decisions.

# Constructing parse table

- $\text{First}(\alpha)$  for a string of terminals and non terminals  $\alpha$  is
  - Set of symbols that might begin the fully expanded (made of only tokens) version of  $\alpha$
- $\text{Follow}(X)$  for a non terminal  $X$  is
  - set of symbols that might follow the derivation of  $X$  in the input stream



# Compute first sets

- If  $X$  is a terminal symbol then  $\text{first}(X) = \{X\}$
- If  $X \rightarrow \epsilon$  is a production then  $\epsilon$  is in  $\text{first}(X)$
- If  $X$  is a non terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production, then
  - if for some  $i$ ,  $a$  is in  $\text{first}(Y_i)$  and  $\epsilon$  is in all of  $\text{first}(Y_j)$  (such that  $j < i$ ) then  $a$  is in  $\text{first}(X)$
- If  $\epsilon$  is in  $\text{first}(Y_1) \dots \text{first}(Y_k)$  then  $\epsilon$  is in  $\text{first}(X)$
- Now generalize to a string  $\alpha$  of terminals and non-terminals

# Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow +T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow ( E ) \mid \text{id}$$

$$\text{First}(E) = \text{First}(T) = \text{First}(F)$$

$$= \{ (, \text{id} \} \}$$

$$\text{First}(E')$$

$$= \{ +, \epsilon \}$$

$$\text{First}(T')$$

$$= \{ *, \epsilon \}$$

# Compute follow sets

1. Place  $\$$  in  $\text{follow}(S)$  //  $S$  is the start symbol
  2. If there is a production  $A \rightarrow \alpha B \beta$   
then everything in  $\text{first}(\beta)$  (except  $\epsilon$ ) is in  
 $\text{follow}(B)$
  3. If there is a production  $A \rightarrow \alpha B \beta$  and  $\text{first}(\beta)$   
contains  $\epsilon$   
then everything in  $\text{follow}(A)$  is in  $\text{follow}(B)$
  4. If there is a production  $A \rightarrow \alpha B$   
then everything in  $\text{follow}(A)$  is in  $\text{follow}(B)$
- Last two steps have to be repeated until the follow sets converge.

# Example

- For the expression grammar

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' \mid \epsilon$$

$$T \rightarrow F T'$$

$$T' \rightarrow * F T' \mid \epsilon$$

$$F \rightarrow ( E ) \mid \text{id}$$

$$\text{follow}(E) = \text{follow}(E') = \{ \$, ) \ }$$

$$\text{follow}(T) = \text{follow}(T') = \{ \$, ), + \}$$

$$\text{follow}(F) = \{ \$, ), +, * \}$$

# Construct SLR parse table

- Construct  $C = \{I_0, \dots, I_n\}$  the collection of sets of LR(0) items
- If  $A \rightarrow \alpha.a\beta$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a] = \text{shift } j$
- If  $A \rightarrow \alpha.$  is in  $I_i$  then  $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$  for all  $a$  in  $\text{follow}(A)$
- If  $S' \rightarrow S.$  is in  $I_i$  then  $\text{action}[i, \$] = \text{accept}$
- If  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$  for all non terminals  $A$
- All entries not defined are errors

# Notes

- This method of parsing is called SLR (Simple LR)
- LR parsers accept LR( $k$ ) languages
  - L stands for left to right scan of input
  - R stands for rightmost derivation
  - $k$  stands for number of lookahead token
- SLR is the simplest of the LR parsing methods.  
SLR is too weak to handle most languages!
- If an SLR parse table for a grammar does not have multiple entries in any cell then the grammar is unambiguous
- All SLR grammars are unambiguous
- Are all unambiguous grammars in SLR?

# Practice Assignment

Construct SLR parse table for following grammar

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid ( E ) \mid \text{digit}$$

Show steps in parsing of string

$$9 * 5 + (2 + 3 * 7)$$

- Steps to be followed
  - Augment the grammar
  - Construct set of LR(0) items
  - Construct the parse table
  - Show states of parser as the given string is parsed

# Example

- Consider following grammar and its SLR parse table:

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

$I_1: goto(I_0, S)$   
 $S' \rightarrow S.$

$I_2: goto(I_0, L)$   
 $S \rightarrow L.=R$   
 $R \rightarrow L.$

$I_0: S' \rightarrow .S$

$S \rightarrow .L=R$

$S \rightarrow .R$

$L \rightarrow .*R$

$L \rightarrow .id$

$R \rightarrow .L$

**Assignment (not to be submitted):**  
Construct rest of the items and the parse table.

## SLR parse table for the grammar

	=	*	id	\$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6,r6			r6			
3				r3			
4		s4	s5			8	7
5	r5			r5			
6		s4	s5			8	9
7	r4			r4			
8	r6			r6			
9				r2			

The table has multiple entries in action[2,=]

- There is both a shift and a reduce entry in action[2,=]. Therefore state 2 has a shift-reduce conflict on symbol “=”, However, the grammar is not ambiguous.
- Parse id=id assuming reduce action is taken in [2,=]

<b>Stack</b>	<b>input</b>	<b>action</b>
0	id=id	shift 5
0 id 5	=id	reduce by $L \rightarrow id$
0 L 2	=id	reduce by $R \rightarrow L$
0 R 3	=id	<b>error</b>

- if shift action is taken in [2,=]

<b>Stack</b>	<b>input</b>	<b>action</b>
0	id=id\$	shift 5
0 id 5	=id\$	reduce by L→id
0 L 2	=id\$	shift 6
0 L 2 = 6	id\$	shift 5
0 L 2 = 6 id 5	\$	reduce by L→id
0 L 2 = 6 L 8	\$	reduce by R→L
0 L 2 = 6 R 9	\$	reduce by S→L=R
0 S 1	\$	ACCEPT

# Problems in SLR parsing

- No sentential form of this grammar can start with  $R=...$
- However, the reduce action in  $\text{action}[2,=]$  generates a sentential form starting with  $R=$
- Therefore, the reduce action is incorrect
- In SLR parsing method state  $i$  calls for reduction on symbol “ $a$ ”, by rule  $A \rightarrow \alpha$  if  $I_i$  contains  $[A \rightarrow \alpha.]$  and “ $a$ ” is in  $\text{follow}(A)$
- However, when state  $I$  appears on the top of the stack, the viable prefix  $\beta\alpha$  on the stack may be such that  $\beta A$  can not be followed by symbol “ $a$ ” in any right sentential form
- Thus, the reduction by the rule  $A \rightarrow \alpha$  on symbol “ $a$ ” is invalid
- **SLR parsers cannot remember the left context**

# Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by  $A \rightarrow \alpha$  will be ruled out
- Redefine LR items to include a terminal symbol as a second component (look ahead symbol)
- The general form of the item becomes  $[A \rightarrow \alpha.\beta, a]$  which is called LR(1) item.
- Item  $[A \rightarrow \alpha., a]$  calls for reduction only if next input is a. The set of symbols “a”s will be a subset of  $\text{Follow}(A)$ .

# Closure( $I$ )

repeat

    for each item  $[A \rightarrow \alpha.B\beta, a]$  in  $I$

        for each production  $B \rightarrow \gamma$  in  $G'$

        and for each terminal  $b$  in  $\text{First}(\beta a)$

            add item  $[B \rightarrow .\gamma, b]$  to  $I$

until no more additions to  $I$

# Example

Consider the following grammar

$$S' \rightarrow S$$

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

Compute closure(I) where I={ $[S' \rightarrow .S, \$]$ }

$S' \rightarrow .S,$	\$
$S \rightarrow .CC,$	\$
$C \rightarrow .cC,$	c
$C \rightarrow .cC,$	d
$C \rightarrow .d,$	c
$C \rightarrow .d,$	d

# Example

Construct sets of LR(1) items for the grammar on previous slide

$I_0: S' \rightarrow .S,$ $S \rightarrow .CC,$ $C \rightarrow .cC,$ $C \rightarrow .d,$	\$ \$ c/d c/d	$I_4: \text{goto}(I_0, d)$ $C \rightarrow d.,$	c/d
$I_1: \text{goto}(I_0, S)$ $S' \rightarrow S.,$	\$	$I_5: \text{goto}(I_2, C)$ $S \rightarrow CC.,$	\$
$I_2: \text{goto}(I_0, C)$ $S \rightarrow C.C,$ $C \rightarrow .cC,$ $C \rightarrow .d,$	\$ \$ \$	$I_6: \text{goto}(I_2, c)$ $C \rightarrow c.C,$ $C \rightarrow .cC,$ $C \rightarrow .d,$	\$ \$ \$
$I_3: \text{goto}(I_0, c)$ $C \rightarrow c.C,$ $C \rightarrow .cC,$ $C \rightarrow .d,$	c/d c/d c/d	$I_7: \text{goto}(I_2, d)$ $C \rightarrow d.,$	\$
		$I_8: \text{goto}(I_3, C)$ $C \rightarrow cC.,$	c/d
		$I_9: \text{goto}(I_6, C)$ $C \rightarrow cC.,$	\$

# Construction of Canonical LR parse table

- Construct  $C = \{I_0, \dots, I_n\}$  the sets of LR(1) items.
- If  $[A \rightarrow \alpha.a\beta, b]$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$  then  $\text{action}[i, a] = \text{shift } j$
- If  $[A \rightarrow \alpha., a]$  is in  $I_i$  then  $\text{action}[i, a] = \text{reduce } A \rightarrow \alpha$
- If  $[S' \rightarrow S., \$]$  is in  $I_i$  then  $\text{action}[i, \$] = \text{accept}$
- If  $\text{goto}(I_i, A) = I_j$  then  $\text{goto}[i, A] = j$  for all non terminals  $A$

# Parse table

State	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

# Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is  $c^*dc^*d$ .
- When reading input cc...dcc...d the parser shifts cs into stack and then goes into state 4 after reading d. It then calls for reduction by  $C \rightarrow d$  if following symbol is c or d.
- IF \$ follows the first d then input string is  $c^*d$  which is not in the language; parser declares an error
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error
- **Problem:** Canonical LR parse table has a large number of states

# LALR Parse table

- Look Ahead LR parsers
- Consider a pair of similar looking states (same kernel and different lookaheads) in the set of LR(1) items  
 $I_4: C \rightarrow d. , c/d$                      $I_7: C \rightarrow d., \$$
- Replace  $I_4$  and  $I_7$  by a new state  $I_{47}$  consisting of  $(C \rightarrow d., c/d/\$)$
- Similarly  $I_3$  &  $I_6$  and  $I_8$  &  $I_9$  form pairs
- Merge LR(1) items having the same core

# Construct LALR parse table

- Construct  $C = \{I_0, \dots, I_n\}$  set of LR(1) items
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union
- Let  $C' = \{J_0, \dots, J_m\}$  be the resulting set of items
- Construct action table as was done earlier
- Let  $J = I_1 \cup I_2 \cup \dots \cup I_k$

since  $I_1, I_2, \dots, I_k$  have same core,  $\text{goto}(J, X)$  will have the same core

Let  $K = \text{goto}(I_1, X) \cup \text{goto}(I_2, X) \cup \dots \cup \text{goto}(I_k, X)$  then  $\text{goto}(J, X) = K$

# LALR parse table ...

State	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

# Notes on LALR parse table

- Modified parser behaves as original except that it will reduce  $C \rightarrow d$  on inputs like ccd. The error will eventually be caught before any more symbols are shifted.
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
- SLR and LALR parse tables have same number of states.

# Notes on LALR parse table...

- Merging items may result into conflicts in LALR parsers which did not exist in LR parsers
- New conflicts can not be of shift reduce kind:
  - Assume there is a shift reduce conflict in some state of LALR parser with items $\{[X \rightarrow \alpha., a], [Y \rightarrow \gamma.a\beta, b]\}$
  - Then there must have been a state in the LR parser with the same core
  - Contradiction; because LR parser did not have conflicts
- LALR parser can have new reduce-reduce conflicts
  - Assume states $\{[X \rightarrow \alpha., a], [Y \rightarrow \beta., b]\}$  and  $\{[X \rightarrow \alpha., b], [Y \rightarrow \beta., a]\}$
  - Merging the two states produces $\{[X \rightarrow \alpha., a/b], [Y \rightarrow \beta., a/b]\}$

# Notes on LALR parse table...

- LALR parsers are not built by first making canonical LR parse tables
- There are direct, complicated but efficient algorithms to develop LALR parsers
- Relative power of various classes
  - $\text{SLR}(1) \leq \text{LALR}(1) \leq \text{LR}(1)$
  - $\text{SLR}(k) \leq \text{LALR}(k) \leq \text{LR}(k)$
  - $\text{LL}(k) \leq \text{LR}(k)$

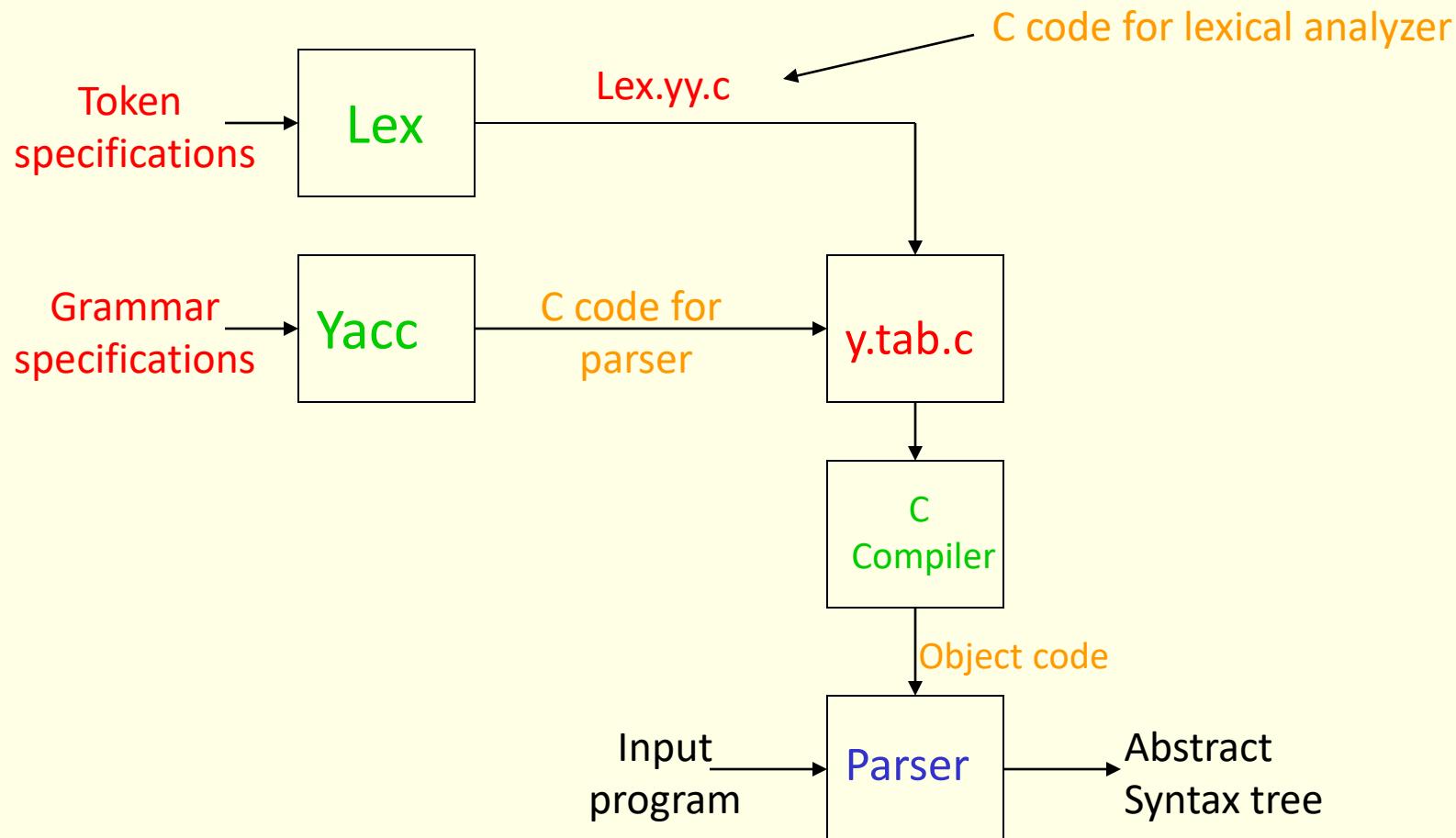
# Error Recovery

- An error is detected when an entry in the action table is found to be empty.
- Panic mode error recovery can be implemented as follows:
  - scan down the stack until a state S with a goto on a particular nonterminal A is found.
  - discard zero or more input symbols until a symbol a is found that can legitimately follow A.
  - stack the state  $\text{goto}[S,A]$  and resume parsing.
- **Choice of A:** Normally these are non terminals representing major program pieces such as an expression, statement or a block. For example if A is the nonterminal stmt, a might be semicolon or end.

# Parser Generator

- Some common parser generators
  - YACC: Yet Another Compiler Compiler
  - Bison: GNU Software
  - ANTLR: ANother Tool for Language Recognition
- Yacc/Bison source program specification (accept LALR grammars)  
declaration  
%%  
translation rules  
%%  
supporting C routines

# Yacc and Lex schema



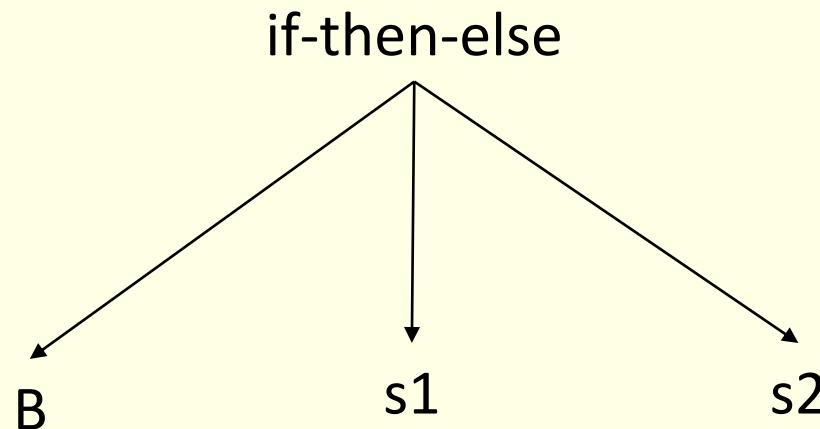
Refer to YACC Manual

# Bottom up parsing ...

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery

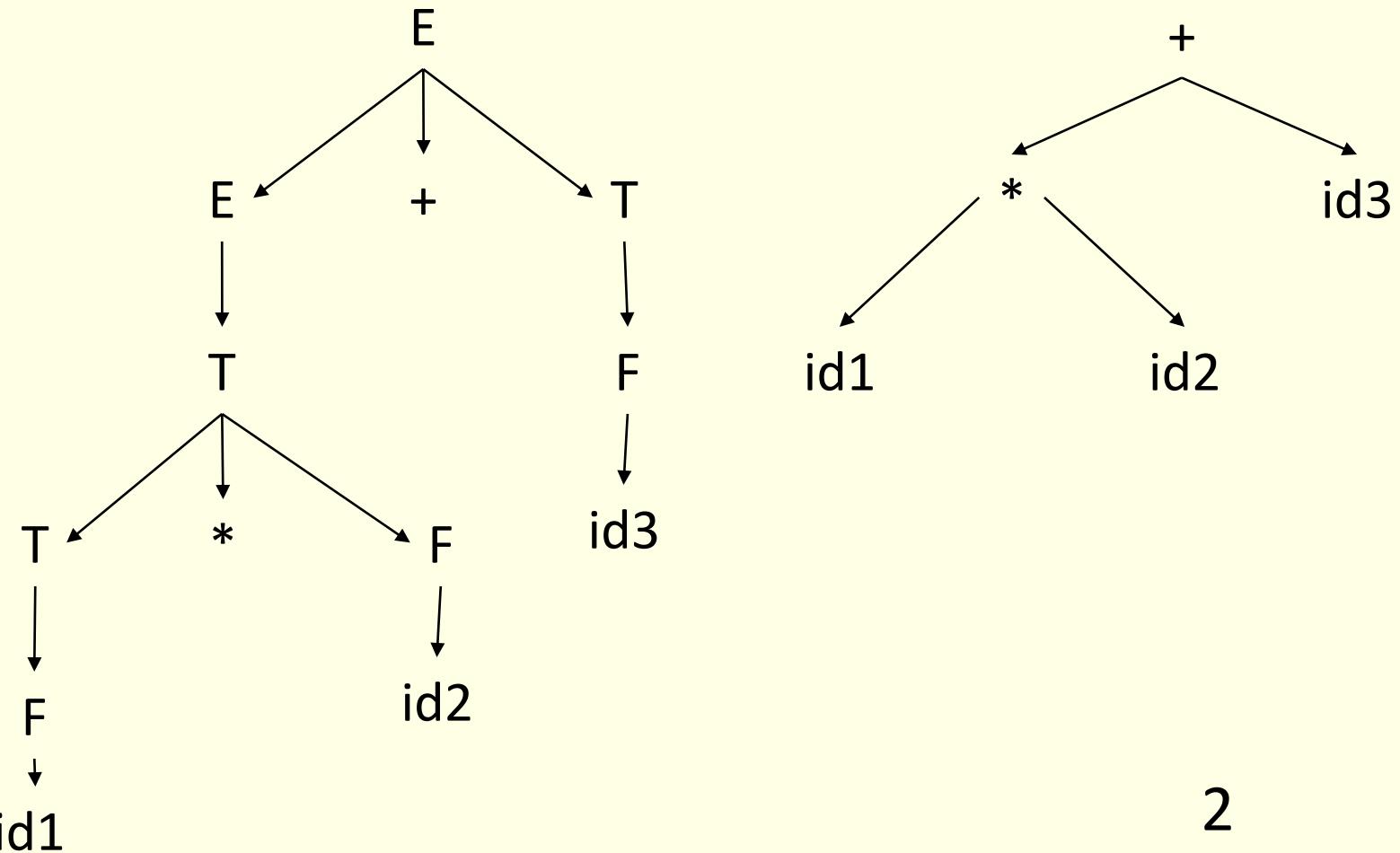
# Abstract Syntax Tree

- Condensed form of parse tree,
- useful for representing language constructs.
- The production  $S \rightarrow \text{if } B \text{ then } s1 \text{ else } s2$   
may appear as



# Abstract Syntax tree ...

- Chain of single productions may be collapsed, and operators move to the parent nodes



# Constructing Abstract Syntax Tree for expression

- Each node can be represented as a record
- *operators*: one field for operator, remaining fields ptrs to operands  
`mknode(op,left,right )`
- *identifier*: one field with label id and another ptr to symbol table  
`mkleaf(id,entry)`
- *number*: one field with label num and another to keep the value of the number  
`mkleaf(num,val)`

# Example

the following sequence of function calls creates a parse tree for  $a - 4 + c$

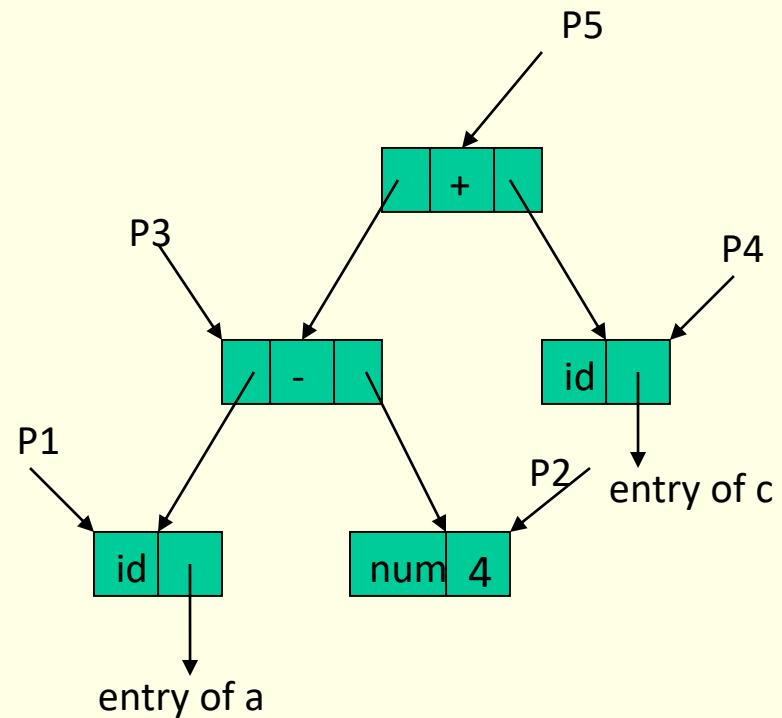
$P_1 = \text{mkleaf}(\text{id}, \text{entry.a})$

$P_2 = \text{mkleaf}(\text{num}, 4)$

$P_3 = \text{mknnode}(-, P_1, P_2)$

$P_4 = \text{mkleaf}(\text{id}, \text{entry.c})$

$P_5 = \text{mknnode}(+, P_3, P_4)$



# A syntax directed definition for constructing syntax tree

$E \rightarrow E_1 + T$	$E.\text{ptr} = \text{mknode}(+, E_1.\text{ptr}, T.\text{ptr})$
$E \rightarrow T$	$E.\text{ptr} = T.\text{ptr}$
$T \rightarrow T_1 * F$	$T.\text{ptr} := \text{mknode}(*, T_1.\text{ptr}, F.\text{ptr})$
$T \rightarrow F$	$T.\text{ptr} := F.\text{ptr}$
$F \rightarrow (E)$	$F.\text{ptr} := E.\text{ptr}$
$F \rightarrow \text{id}$	$F.\text{ptr} := \text{mkleaf}(\text{id}, \text{entry.id})$
$F \rightarrow \text{num}$	$F.\text{ptr} := \text{mkleaf}(\text{num}, \text{val})$

# DAG for Expressions

Expression  $a + a * ( b - c ) + ( b - c ) * d$

make a leaf or node if not present,  
otherwise return pointer to the existing node

$P_1 = \text{makeleaf(id,a)}$

$P_2 = \text{makeleaf(id,a)}$

$P_3 = \text{makeleaf(id,b)}$

$P_4 = \text{makeleaf(id,c)}$

$P_5 = \text{makenode}(-, P_3, P_4)$

$P_6 = \text{makenode}(*, P_2, P_5)$

$P_7 = \text{makenode}(+, P_1, P_6)$

$P_8 = \text{makeleaf(id,b)}$

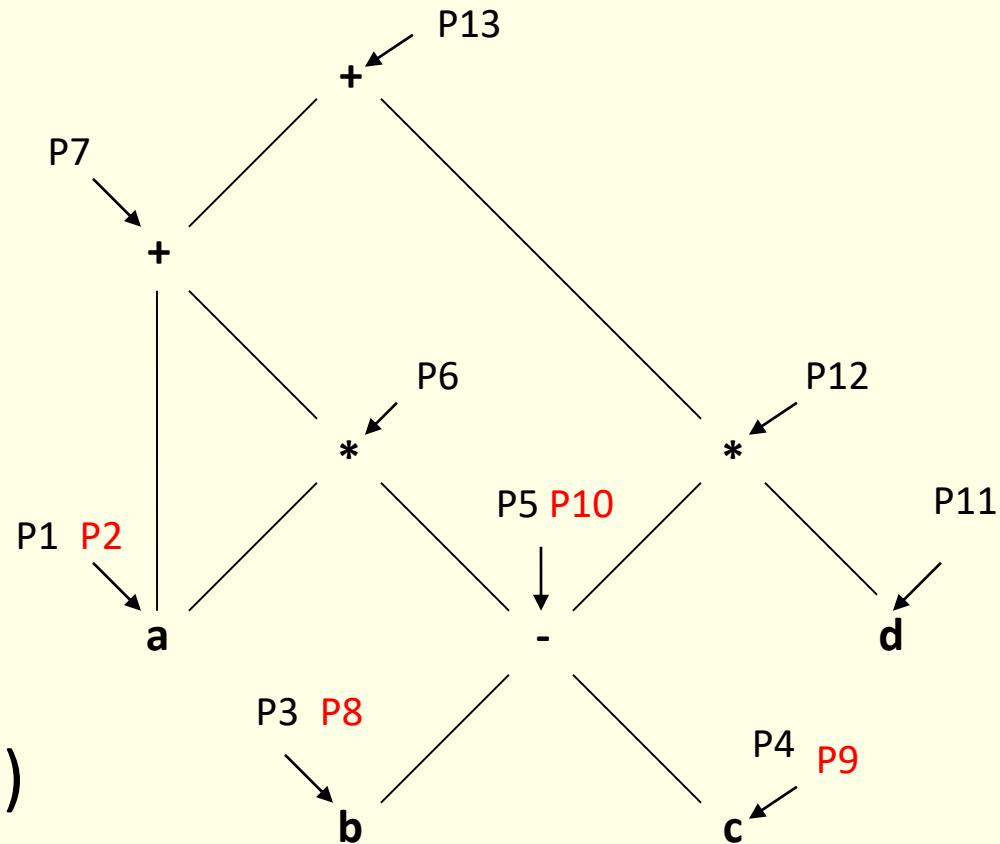
$P_9 = \text{makeleaf(id,c)}$

$P_{10} = \text{makenode}(-, P_8, P_9)$

$P_{11} = \text{makeleaf(id,d)}$

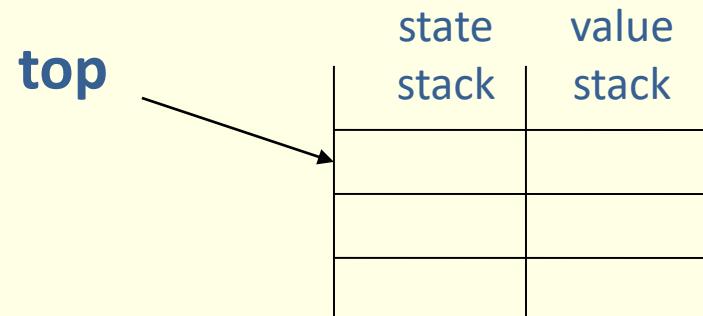
$P_{12} = \text{makenode}(*, P_{10}, P_{11})$

$P_{13} = \text{makenode}(+, P_7, P_{12})$



# Bottom-up evaluation of S-attributed definitions

- Can be evaluated while parsing
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by **top** pointer



# Bottom-up evaluation of S-attributed definitions

- Suppose semantic rule

$$A.a = f(X.x, Y.y, Z.z)$$

is associated with production

$$A \rightarrow XYZ$$

- Before reducing  $XYZ$  to  $A$ , value of  $Z$  is in  $\text{val}(\text{top})$ , value of  $Y$  is in  $\text{val}(\text{top-1})$  and value of  $X$  is in  $\text{val}(\text{top-2})$
- If symbol has no attribute then the entry is undefined
- After the reduction,  $\text{top}$  is decremented by 2 and state covering  $A$  is put in  $\text{val}(\text{top})$

# Example: desk calculator

$L \rightarrow E \$$	Print (E.val)
$E \rightarrow E + T$	$E.val = E.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T * F$	$T.val = T.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{digit}$	$F.val = \text{digit}.lexval$

# Example: desk calculator

$L \rightarrow E\$$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Before reduction

$\text{ntop} = \text{top} - r + 1$

After code reduction  $\text{top} = \text{ntop}$

$r$  is the #symbols on RHS

INPUT	STATE	Val	PROD
3*5+4\$			
*5+4\$	digit	3	
*5+4\$	F	3	$F \rightarrow \text{digit}$
*5+4\$	T	3	$T \rightarrow F$
5+4\$	$T^*$	3 □	
+4\$	$T^* \text{digit}$	3 □ 5	
+4\$	$T^* F$	3 □ 5	$F \rightarrow \text{digit}$
+4\$	T	15	$T \rightarrow T^* F$
+4\$	E	15	$E \rightarrow T$
4\$	$E^+$	15 □	
\$	$E^+ \text{digit}$	15 □ 4	
\$	$E^+ F$	15 □ 4	$F \rightarrow \text{digit}$
\$	$E^+ T$	15 □ 4	$T \rightarrow F$
\$	E	19	$E \rightarrow E^+ T$

# YACC Terminology

$E \rightarrow E + T \quad \text{val(ntop)} = \text{val}(top-2) + \text{val}(top)$

In YACC

$E \rightarrow E + T \quad \$\$ = \$1 + \$3$

$\$\$$  maps to  $\text{val}[top - r + 1]$

$\$k$  maps to  $\text{val}[top - r + k]$

$r = \#$ symbols on RHS ( here 3)

$\$\$ = \$1$  is the *default* action in YACC

# L-attributed definitions

- When translation takes place during parsing, order of evaluation is linked to the order in which nodes are created
- In S-attributed definitions parent's attribute evaluated after child's.
- A natural order in both top-down and bottom-up parsing is depth first-order
- **L-attributed** definition: where attributes can be evaluated in depth-first order

# L attributed definitions ...

- A syntax directed definition is L-attributed if each inherited attribute of  $X_j$  ( $1 \leq j \leq n$ ) at the right hand side of  $A \rightarrow X_1 X_2 \dots X_n$  depends only on
  - Attributes of symbols  $X_1 X_2 \dots X_{j-1}$  and
  - Inherited attribute of A
- Examples (i inherited, s synthesized)

$A \rightarrow LM$

$L.i = f_1(A.i)$

$M.i = f_2(L.s)$

$A.s = f_3(M.s)$



$A \rightarrow QR$

$R.i = f4(A.i)$

$Q.i = f5(R.s)$

$A.s = f6(Q.s)$



# Translation schemes

- A CFG where semantic actions occur within the rhs of production
- Example: A translation scheme to map infix to postfix

$E \rightarrow T\ R$

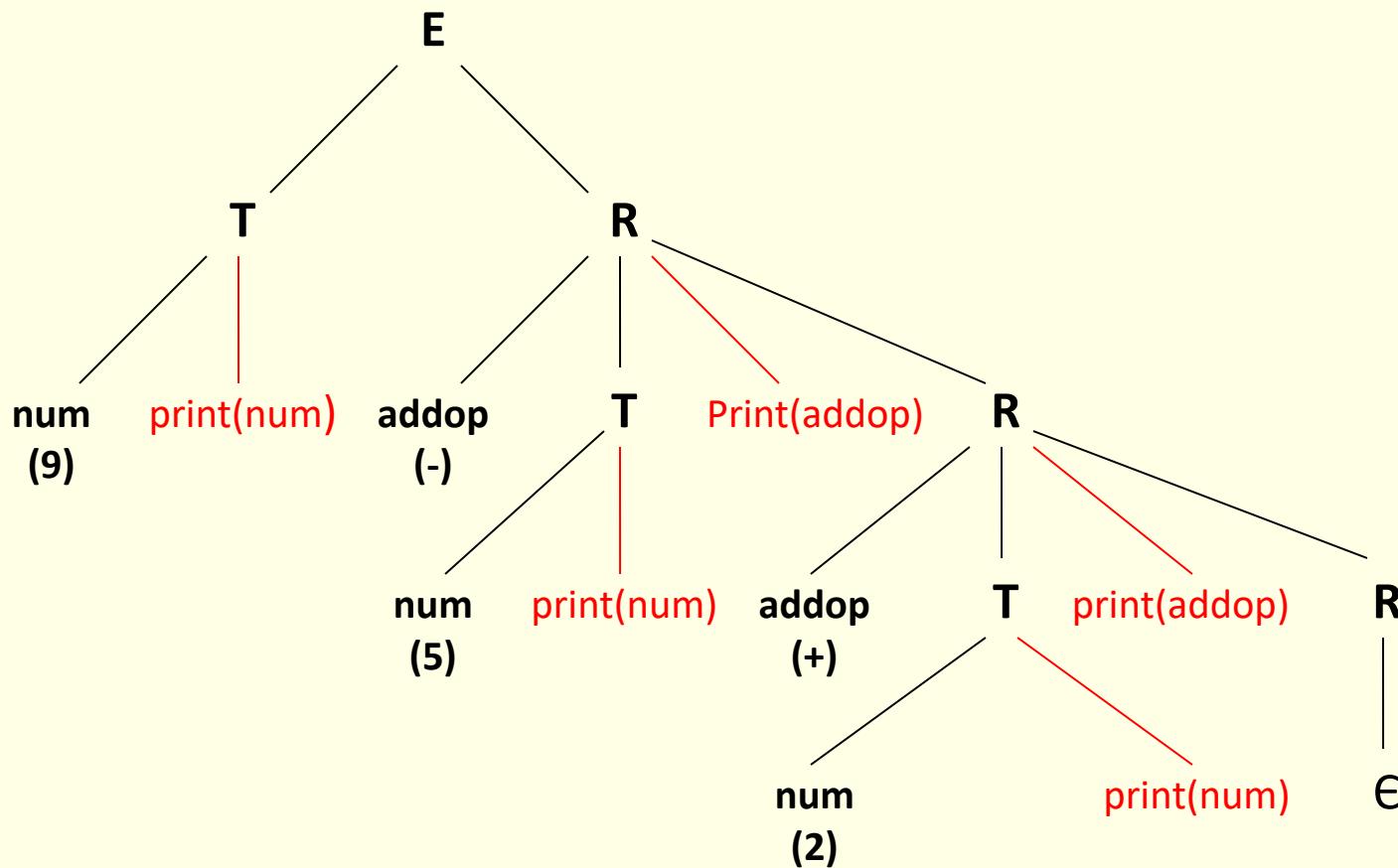
$R \rightarrow \text{addop}\ T\ R \mid \epsilon$

$T \rightarrow \text{num}$

$\text{addop} \rightarrow + \mid -$

Exercise: Create Parse Tree for  $9 - 5 + 2$

# Parse tree for 9-5+2



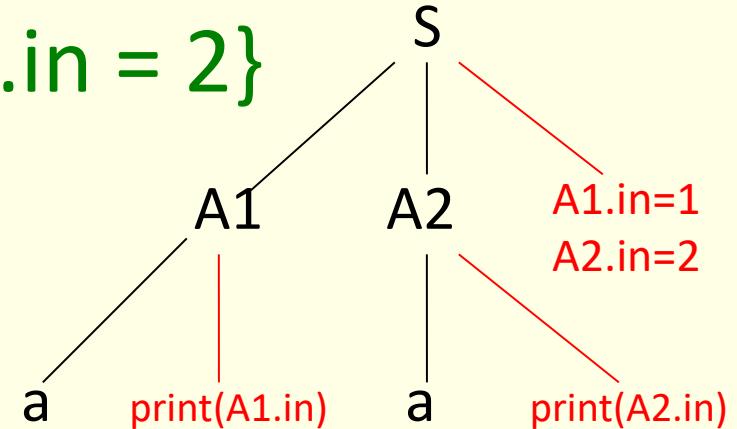
# Evaluation of Translation Schemes

- Assume actions are terminal symbols
- Perform depth first order traversal to obtain 9 5 – 2 +
- When designing translation scheme, **ensure** attribute value is available when referred to
- In case of synthesized attribute it is trivial (**why ?**)

- An inherited attribute for a symbol on RHS of a production must be computed in an action before that symbol

$$S \rightarrow A_1 A_2 \quad \{A_1.\text{in} = 1, A_2.\text{in} = 2\}$$

$$A \rightarrow a \quad \{\text{print}(A.\text{in})\}$$



depth first order traversal gives error (*undef*)

- A synthesized attribute for the non terminal on the LHS can be computed after all attributes it references, have been computed. **The action normally should be placed at the end of RHS.**

# Bottom up evaluation of inherited attributes

- Remove embedded actions from translation scheme
- Make transformation so that embedded actions occur only at the ends of their productions
- Replace each action by a distinct marker non terminal M and attach action at end of  $M \rightarrow \epsilon$

$E \rightarrow TR$   
 $R \rightarrow + T \{ \text{print } (+) \} R$   
 $R \rightarrow - T \{ \text{print } (-) \} R$   
 $R \rightarrow \epsilon$   
 $T \rightarrow \text{num } \{ \text{print}(\text{num.val}) \}$

transforms to

$E \rightarrow TR$   
 $R \rightarrow + T M R$   
 $R \rightarrow - T N R$   
 $R \rightarrow \epsilon$   
 $T \rightarrow \text{num} \quad \{ \text{print}(\text{num.val}) \}$   
 $M \rightarrow \epsilon \quad \{ \text{print}(+) \}$   
 $N \rightarrow \epsilon \quad \{ \text{print}(-) \}$

## Inheriting attribute on parser stacks

- bottom up parser reduces rhs of  $A \rightarrow XY$  by removing  $XY$  from stack and putting  $A$  on the stack
- synthesized attributes of  $Xs$  can be inherited by  $Y$  by using the copy rule  
 $Y.i=X.s$

# Inherited Attributes: SDD

$D \rightarrow T\ L$        $L.in = T.type$

$T \rightarrow \text{real}$        $T.type = \text{real}$

$T \ int$        $T.type = \text{int}$

$L \rightarrow L_1, id$        $L_1.in = L.in;$   
                         $\text{addtype}(id.entry, L.in)$

$L \rightarrow id$        $\text{addtype } (id.entry, L.in)$

Exercise: Convert to Translation Scheme

# Inherited Attributes: Translation Scheme

$D \rightarrow T \{L.in = T.type\} L$

$T \rightarrow \text{int } \{T.type = \text{integer}\}$   
 $T \rightarrow \text{real } \{T.type = \text{real}\}$

$L \rightarrow \{L_1.in = L.in\} L_1, id \{addtype(id.entry, L_{in})\}$

$L \rightarrow id \{addtype(id.entry, L_{in})\}$

**Example:** take string      real p,q,r

<b>State stack</b>	<b>INPUT</b>	<b>PRODUCTION</b>
real	real p,q,r	
T	p,q,r	
Tp	p,q,r	$T \rightarrow \text{real}$
TL	,q,r	
TL,	,q,r	$L \rightarrow \text{id}$
TL,q	q,r	
TL	,r	$L \rightarrow L,\text{id}$
TL,	r	
TL,r	-	
TL	-	$L \rightarrow L,\text{id}$
D	-	$D \rightarrow TL$

Every time a string is reduced to L, T.val is just below it on the stack

# Example ...

- Every time a reduction to L is made value of T type is just below it
- Use the fact that T.val (type information) is at a known place in the stack
- When production  $L \rightarrow id$  is applied, id.entry is at the top of the stack and T.type is just below it, therefore,

$\text{addtype}(id.\text{entry}, L.\text{in}) \Leftrightarrow$

$\text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-1])$

- Similarly when production  $L \rightarrow L_1$ , id is applied id.entry is at the top of the stack and T.type is three places below it, therefore,

$\text{addtype}(id.\text{entry}, L.\text{in}) \Leftrightarrow$

$\text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-3])$

## Example ...

Therefore, the translation scheme becomes

$D \rightarrow T\ L$

$T \rightarrow \text{int}$

$\text{val}[\text{top}] = \text{integer}$

$T \rightarrow \text{real}$

$\text{val}[\text{top}] = \text{real}$

$L \rightarrow L, id$

$\text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-3])$

$L \rightarrow id$

$\text{addtype}(\text{val}[\text{top}], \text{val}[\text{top}-1])$

# Simulating the evaluation of inherited attributes

- The scheme works only if grammar allows position of attribute to be predicted.
- Consider the grammar

$$\begin{array}{ll} S \rightarrow aAC & C_i = A_s \\ S \rightarrow bABC & C_i = A_s \\ C \rightarrow c & C_s = g(C_i) \end{array}$$

- C inherits  $A_s$
- there may or may not be a B between A and C on the stack when reduction by rule  $C \rightarrow c$  takes place
- When reduction by  $C \rightarrow c$  is performed the value of  $C_i$  is either in [top-1] or [top-2]

# Simulating the evaluation ...

- Insert a marker M just before C in the second rule and change rules to

$$S \rightarrow aAC$$

$$S \rightarrow bABMC$$

$$C \rightarrow c$$

$$M \rightarrow \epsilon$$

$$C_i = A_s$$

$$M_i = A_s; C_i = M_s$$

$$C_s = g(C_i)$$

$$M_s = M_i$$

- When production  $M \rightarrow \epsilon$  is applied we have  
 $M_s = M_i = A_s$
- Therefore value of  $C_i$  is always at val[top-1]

## Simulating the evaluation ...

- Markers can also be used to simulate rules that are not copy rules

$$S \rightarrow aAC$$

$$C_i = f(A.s)$$

- using a marker

$$S \rightarrow aANC$$

$$N \rightarrow \epsilon$$

$$N_i = A_s; C_i = N_s$$

$$N_s = f(N_i)$$

# General algorithm

- **Algorithm:** Bottom up parsing and translation with inherited attributes
- **Input:** L attributed definitions
- **Output:** A bottom up parser
- Assume every non terminal has one inherited attribute and every grammar symbol has a synthesized attribute
- For every production  $A \rightarrow X_1 \dots X_n$  introduce n markers  $M_1 \dots M_n$  and replace the production by
$$\begin{array}{c} A \rightarrow M_1 X_1 \dots M_n X_n \\ M_1 \dots M_n \rightarrow \epsilon \end{array}$$
- Synthesized attribute  $X_{j,s}$  goes into the value entry of  $X_j$
- Inherited attribute  $X_{j,i}$  goes into the value entry of  $M_j$

# Algorithm ...

- If the reduction is to a marker  $M_j$  and the marker belongs to a production

$A \rightarrow M_1 X_1 \dots M_n X_n$  then

$A_i$  is in position top-2j+2

$X_{1,i}$  is in position top-2j+3

$X_{1,s}$  is in position top-2j+4

- If reduction is to a non terminal A by production  $A \rightarrow M_1 X_1 \dots M_n X_n$  then compute  $A_s$  and push on the stack

# Space for attributes at compile time

- Lifetime of an attribute begins when it is first computed
- Lifetime of an attribute ends when all the attributes depending on it, have been computed
- Space can be conserved by assigning space for an attribute only during its lifetime

# Example

- Consider following definition

$D \rightarrow T\ L$

$L.in := T.type$

$T \rightarrow \text{real}$

$T.type := \text{real}$

$T \rightarrow \text{int}$

$T.type := \text{int}$

$L \rightarrow L_1, l$

$L_1.in := L.in; l.in = L.in$

$L \rightarrow l$

$l.in = L.in$

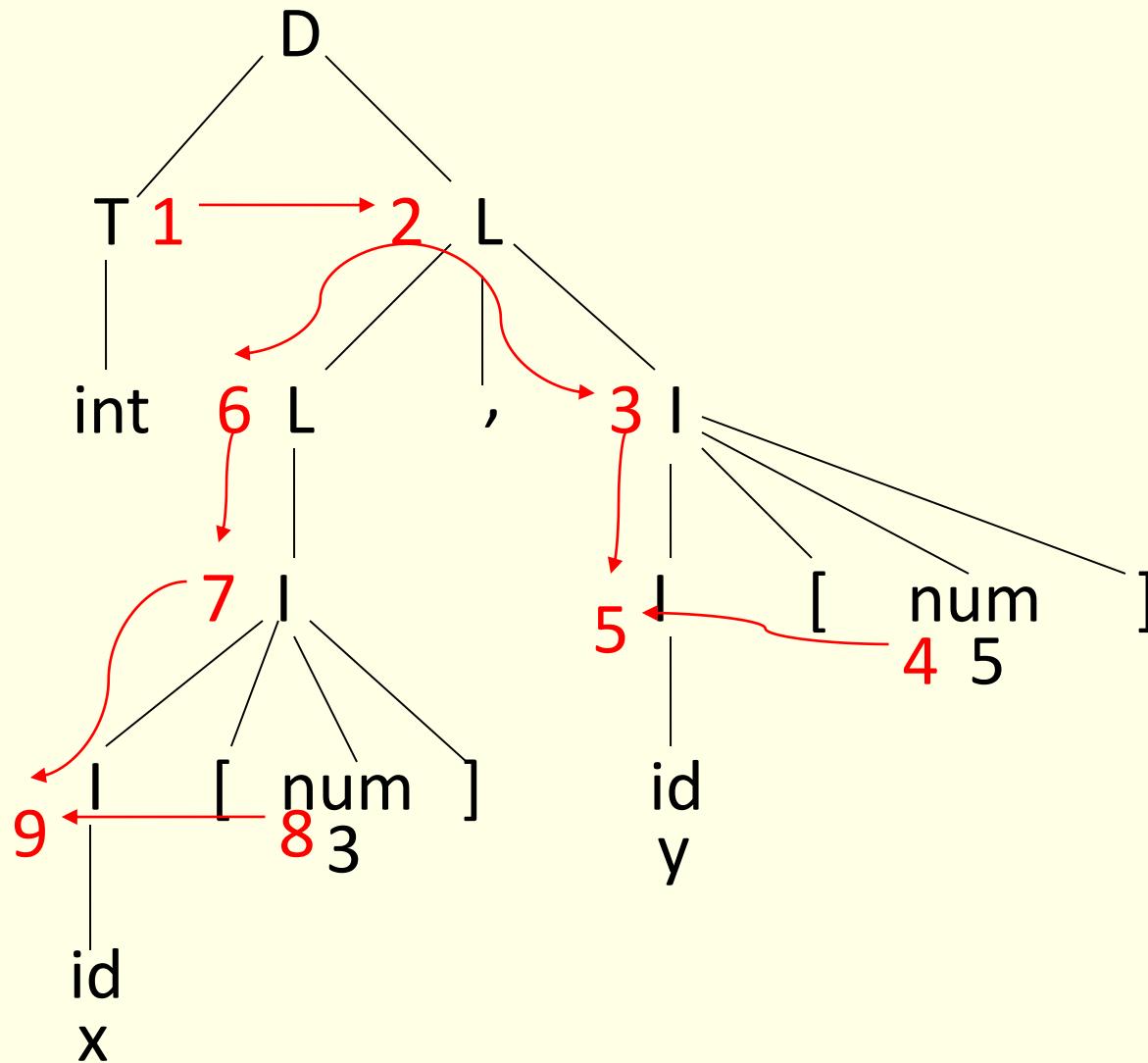
$l \rightarrow l_1[\text{num}]$

$l_1.in = \text{array}(\text{numeral}, l.in)$

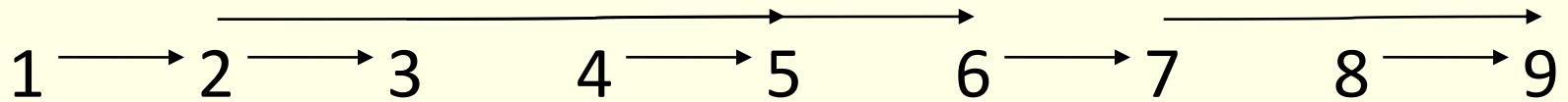
$l \rightarrow id$

$\text{addtype}(id.entry, l.in)$

Consider string int x[3], y[5]  
its parse tree and dependence graph



# Resource requirement



Allocate resources using life time information

R1    R1    R2    R3    R2    R1    R1    R2    R1

Allocate resources using life time and copy information

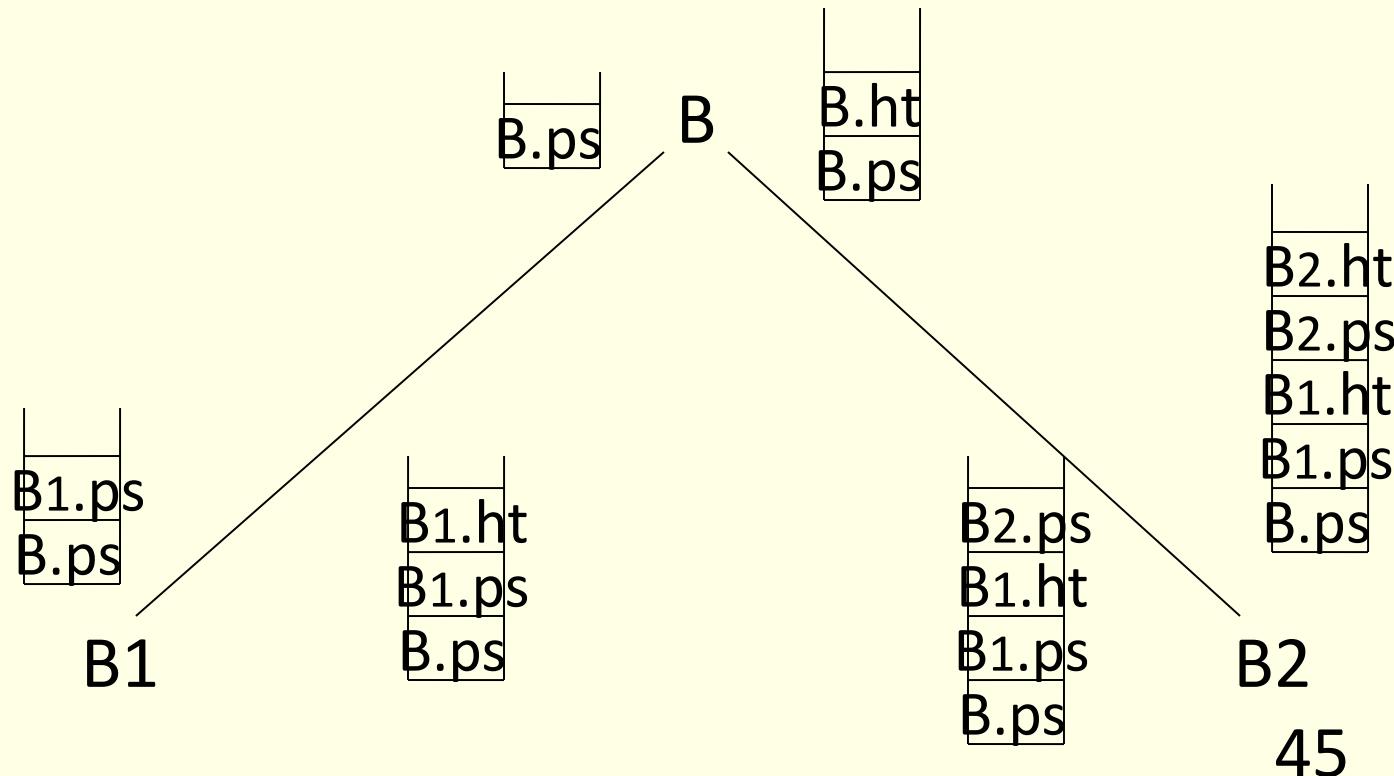
R1    =R1    =R1    R2    R2    =R1    =R1    R2    R1

# Space for attributes at compiler Construction time

- Attributes can be held on a single stack. However, lot of attributes are copies of other attributes
- For a rule like  $A \rightarrow B C$  stack grows up to a height of five (assuming each symbol has one inherited and one synthesized attribute)
- Just before reduction by the rule  $A \rightarrow B C$  the stack contains  $I(A) I(B) S(B) I(C) S(C)$
- After reduction the stack contains  $I(A) S(A)$
-

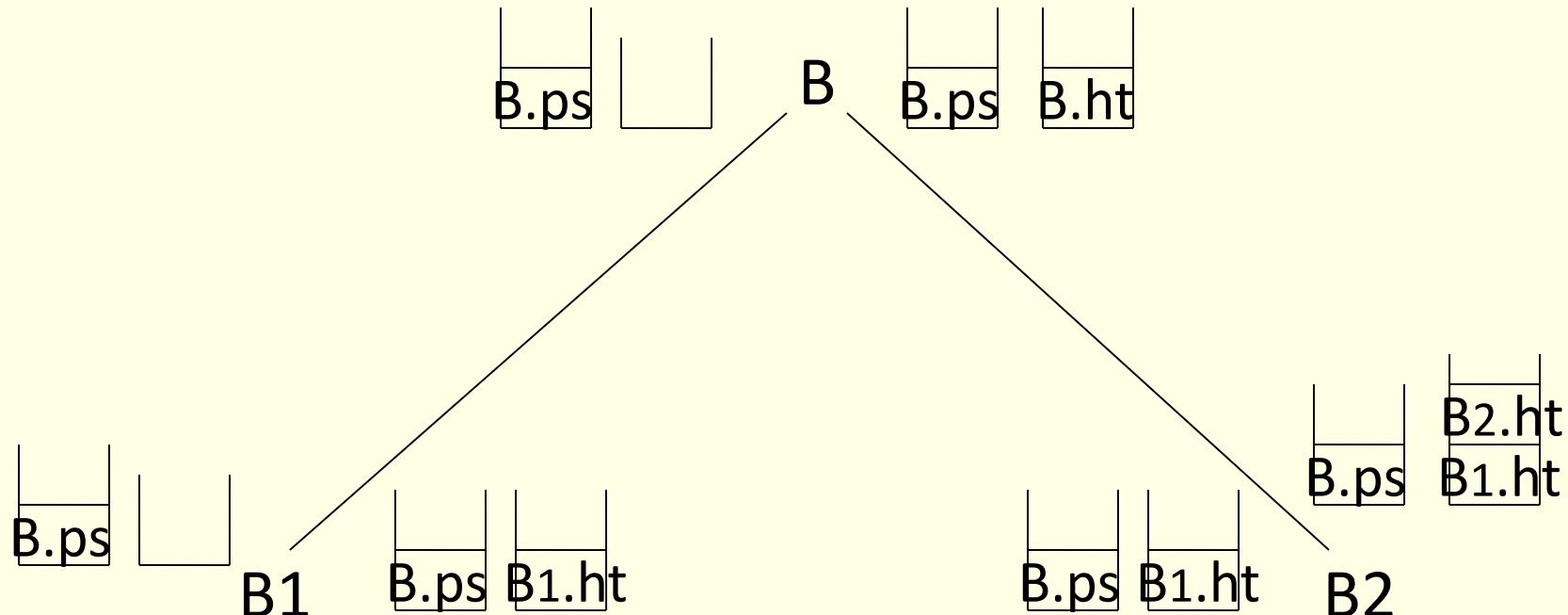
# Example

- Consider rule  $B \rightarrow B_1 B_2$  with inherited attribute ps and synthesized attribute ht
- The parse tree for this string and a snapshot of the stack at each node appears as



# Example ...

- However, if different stacks are maintained for the inherited and synthesized attributes, the stacks will normally be smaller



# Type system

- A type is a set of values and operations on those values
- A language's type system specifies which operations are valid for a type
- The aim of type checking is to ensure that operations are used on the variable/expressions of the correct types

# Type system ...

- Languages can be divided into three categories with respect to the type:
  - “untyped”
    - No type checking needs to be done
    - Assembly languages
  - Statically typed
    - All type checking is done at compile time
    - Algol class of languages
    - Also, called strongly typed
  - Dynamically typed
    - Type checking is done at run time
    - Mostly functional languages like Lisp, Scheme etc.

# Type systems ...

- Static typing
  - Catches most common programming errors at compile time
  - Avoids runtime overhead
  - May be restrictive in some situations
  - Rapid prototyping may be difficult
- Most code is written using static types languages
- In fact, developers for large/critical system insist that code be strongly type checked at compile time even if language is not strongly typed (use of Lint for C code, code compliance checkers)

# Type System

- A type system is a collection of rules for assigning type expressions to various parts of a program
- Different type systems may be used by different compilers for the same language
- In Pascal type of an array includes the index set. Therefore, a function with an array parameter can only be applied to arrays with that index set
- Many Pascal compilers allow index set to be left unspecified when an array is passed as a parameter

# Type system and type checking

- If both the operands of arithmetic operators +, -, x are integers then the result is of type integer
- The result of unary & operator is a pointer to the object referred to by the operand.
  - If the type of operand is  $X$  the type of result is ***pointer to X***
- **Basic types:** integer, char, float, boolean
- **Sub range type:** 1 ... 100
- **Enumerated type:** (violet, indigo, red)
- **Constructed type:** array, record, pointers, functions

# Type expression

- Type of a language construct is denoted by a type expression
- It is either a basic type OR
- it is formed by applying operators called *type constructor* to other type expressions
- A basic type is a type expression. There are two special basic types:
  - *type error*: error during type checking
  - *void*: no type value
- A type constructor applied to a type expression is a type expression

# Type Constructors

- **Array:** if  $T$  is a type expression then  $\text{array}(I, T)$  is a type expression denoting the type of an array with elements of type  $T$  and index set  $I$

`int A[10];`

- $A$  can have type expression  $\text{array}(0 .. 9, \text{integer})$
- C does not use this type, but uses equivalent of `int*`

- **Product:** if  $T_1$  and  $T_2$  are type expressions then their Cartesian product  $T_1 * T_2$  is a type expression
  - `Pair/tuple`

# Type constructors ...

- **Records**: it applies to a tuple formed from field names and field types. Consider the declaration

```
type row = record
    addr : integer;
    lexeme : array [1 .. 15] of char
end;
```

```
var table: array [1 .. 10] of row;
```

- The type `row` has type expression

```
record ((addr * integer) * (lexeme * array(1 .. 15,
char)))
```

and type expression of `table` is `array(1 .. 10, row)`

# Type constructors ...

- **Pointer**: if  $T$  is a type expression then  $\text{pointer}(T)$  is a type expression denoting type pointer to an object of type  $T$
- **Function**: function maps domain set to range set. It is denoted by type expression  $D \rightarrow R$ 
  - For example  $\%$  has type expression  $\text{int} * \text{int} \rightarrow \text{int}$
  - The type of function  $\text{int}^* f(\text{char } a, \text{char } b)$  is denoted by  $\text{char} * \text{char} \rightarrow \text{pointer}(\text{int})$

# Specifications of a type checker

- Consider a language which consists of a sequence of declarations followed by a single expression

$$P \rightarrow D ; E$$
$$D \rightarrow D ; D \mid id : T$$
$$T \rightarrow \text{char} \mid \text{integer} \mid T[\text{num}] \mid T^*$$
$$E \rightarrow \text{literal} \mid \text{num} \mid E\%E \mid E [E] \mid *E$$

# Specifications of a type checker ...

- A program generated by this grammar is

```
key : integer;  
key %1999
```

- Assume following:
  - basic types are char, int, type-error
  - all arrays start at 0
  - char[256] has type expression  
array(0 .. 255, char)

# Rules for Symbol Table entry

$D \rightarrow id : T$

addtype(id.entry, T.type)

$T \rightarrow char$

T.type = char

$T \rightarrow integer$

T.type = int

$T \rightarrow T_1^*$

T.type = pointer( $T_1$ .type)

$T \rightarrow T_1 [num]$

T.type = array(0..num-1,  $T_1$ .type)

# Type checking of functions

$E \rightarrow E_1 (E_2)$   $E.\text{type} =$

$(E_1.\text{type} == s \rightarrow t \text{ and } E_2.\text{type} == s)$

?  $t : \text{type-error}$

# Type checking for expressions

$E \rightarrow \text{literal}$

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

$E \rightarrow E_1 \% E_2$

$E \rightarrow E_1[E_2]$

$E \rightarrow *E_1$

# Type checking for expressions

$E \rightarrow \text{literal}$	$E.\text{type} = \text{char}$
$E \rightarrow \text{num}$	$E.\text{type} = \text{integer}$
$E \rightarrow \text{id}$	$E.\text{type} = \text{lookup(id.entry)}$
$E \rightarrow E_1 \% E_2$	$E.\text{type} = \begin{cases} \text{if } E_1.\text{type} == \text{integer} \text{ and } E_2.\text{type} == \text{integer} \\ \quad \text{then integer} \\ \quad \text{else type\_error} \end{cases}$
$E \rightarrow E_1[E_2]$	$E.\text{type} = \begin{cases} \text{if } E_2.\text{type} == \text{integer} \text{ and } E_1.\text{type} == \text{array(s,t)} \\ \quad \text{then t} \\ \quad \text{else type\_error} \end{cases}$
$E \rightarrow *E_1$	$E.\text{type} = \begin{cases} \text{if } E_1.\text{type} == \text{pointer(t)} \\ \quad \text{then t} \\ \quad \text{else type\_error} \end{cases}$

# Type checking for statements

- Statements typically do not have values. Special basic type *void* can be assigned to them.

$S \rightarrow id := E$

$S \rightarrow if\ E\ then\ S1$

$S \rightarrow while\ E\ do\ S1$

$S \rightarrow S1 ; S2$

# Type checking for statements

- Statements typically do not have values. Special basic type *void* can be assigned to them.

$S \rightarrow id := E$

$S.Type = \begin{cases} \text{if } id.type == E.type \\ \text{then void} \\ \text{else type\_error} \end{cases}$

$S \rightarrow \text{if } E \text{ then } S1$

$S.Type = \begin{cases} \text{if } E.type == \text{boolean} \\ \text{then } S1.type \\ \text{else type\_error} \end{cases}$

$S \rightarrow \text{while } E \text{ do } S1$

$S.Type = \begin{cases} \text{if } E.type == \text{boolean} \\ \text{then } S1.type \\ \text{else type\_error} \end{cases}$

$S \rightarrow S1 ; S2$

$S.Type = \begin{cases} \text{if } S1.type == \text{void} \\ \text{and } S2.type == \text{void} \\ \text{then void} \\ \text{else type\_error} \end{cases}$

# Equivalence of Type expression

- Structural equivalence: Two type expressions are equivalent if
  - either these are same basic types
  - or these are formed by applying same constructor to equivalent types
- Name equivalence: types can be given names
  - Two type expressions are equivalent if they have the same name

# Function to test structural equivalence

boolean sequiv(type s, type t) :

If s and t are same basic types

then return true

elseif s == array(s1, s2) and t == array(t1, t2)

then return sequiv(s1, t1) && sequiv(s2, t2)

elseif s == s1 \* s2 and t == t1 \* t2

then return sequiv(s1, t1) && sequiv(s2, t2)

elseif s == pointer(s1) and t == pointer(t1)

then return sequiv(s1, t1)

elseif s == s1 → s2 and t == t1 → t2

then return sequiv(s1,t1) && sequiv(s2,t2)

else return false;

# Efficient implementation

- Bit vectors can be used to represent type expressions. Refer to: A Tour Through the Portable C Compiler: S. C. Johnson, 1979.

Basic type	Encoding
Boolean	0000
Char	0001
Integer	0010
real	0011

Type constructor	encoding
pointer	01
array	10
function	11

# Efficient implementation ...

Basic type	Encoding	Type constructor	Encoding
Boolean	0000	pointer	01
Char	0001		10
Integer	0010		11
real	0011		

Type expression

char

function( char )

pointer( function( char ) )

array( pointer( function( char ) ) )

encoding

000000 0001

000011 0001

000111 0001

100111 0001

This representation saves space and keeps  
track of constructors

# Checking name equivalence

- Consider following declarations

```
typedef cell* link;  
link next, last;  
cell *p, *q, *r;
```
- Do the variables next, last, p, q and r have identical types ?
- Type expressions have names and names appear in type expressions.
- Name equivalence views each type name as a distinct type

# Name equivalence ...

variable	type expression
next	link
last	link
p	pointer(cell)
q	pointer(cell)
r	pointer(cell)

- Under name equivalence  $\text{next} = \text{last}$  and  $\text{p} = \text{q} = \text{r}$  , however,  $\text{next} \neq \text{p}$
- Under structural equivalence all the variables are of the same type

# Name equivalence ...

- Some compilers allow type expressions to have names.
- However, some compilers assign **implicit type names**.
- A fresh implicit name is created every time a type name appears in declarations.
- Consider  
`type link = ^ cell;`  
`var next : link;`  
    `last : link;`  
    `p, q : ^ cell;`  
    `r : ^ cell;`
- In this case type expression of q and r are given different implicit names and therefore, those are not of the same type

# Name equivalence ...

The previous code is equivalent to

```
type link = ^ cell;  
    np = ^ cell;  
    nr = ^ cell;  
var next : link;  
    last : link;  
    p, q: np;  
    r : nr;
```

# Cycles in representation of types

- Data structures like linked lists are defined recursively
- Implemented through structures which contain pointers to structure
- Consider following code

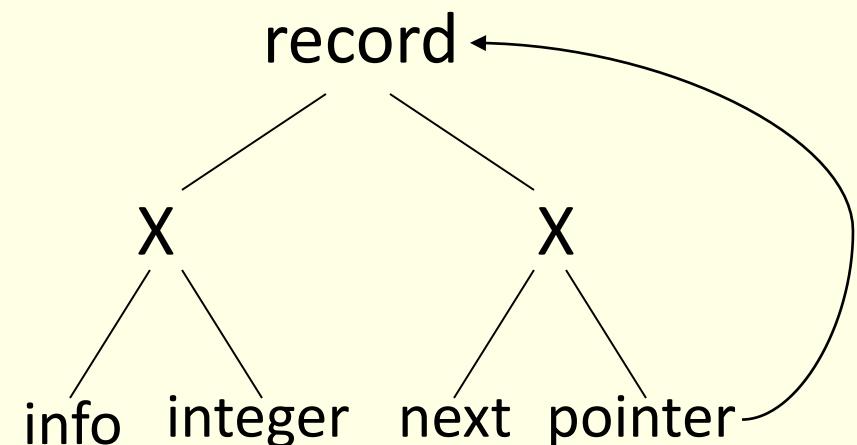
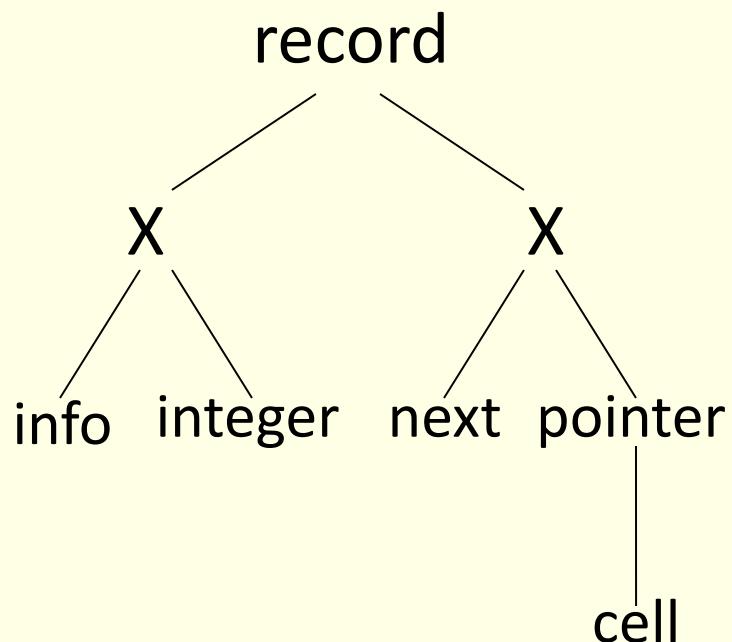
```
type link = ^ cell;
cell = record
    info : integer;
    next : link
end;
```

- The type name **cell** is defined in terms of **link** and **link** is defined in terms of **cell** (recursive definitions)

# Cycles in representation of ...

- Recursively defined type names can be substituted by definitions
- However, it introduces cycles into the type graph

```
link = ^ cell;  
cell = record  
    info : integer;  
    next : link  
end;
```



# Cycles in representation of ...

- C uses structural equivalence for all types except records (struct)
- It uses the acyclic structure of the type graph
- Type names must be declared before they are used
  - However, allow pointers to undeclared record types
  - All potential cycles are due to pointers to records
- Name of a record is part of its type
  - Testing for structural equivalence stops when a record constructor is reached

# Type conversion

- Consider expression like  $x + i$  where  $x$  is of type real and  $i$  is of type integer
- Internal representations of integers and reals are different in a computer
  - different machine instructions are used for operations on integers and reals
- The compiler has to convert both the operands to the same type
- Language definition specifies what conversions are necessary.

# Type conversion ...

- Usually conversion is to the type of the left hand side
- Type checker is used to insert conversion operations:  
 $x + i$   
 $\Rightarrow x \text{ real} + \text{inttoreal}(i)$
- Type conversion is called implicit/coercion if done by compiler.
- It is limited to the situations where no information is lost
- Conversions are explicit if programmer has to write something to cause conversion

# Type checking for expressions

$E \rightarrow \text{num}$

$E.\text{type} = \text{int}$

$E \rightarrow \text{num.num}$

$E.\text{type} = \text{real}$

$E \rightarrow \text{id}$

$E.\text{type} = \text{lookup( id.entry )}$

$E \rightarrow E_1 \text{ op } E_2$

$E.\text{type} =$   
if  $E_1.\text{type} == \text{int} \text{ && } E_2.\text{type} == \text{int}$   
then  $\text{int}$   
elif  $E_1.\text{type} == \text{int} \text{ && } E_2.\text{type} == \text{real}$   
then  $\text{real}$   
elif  $E_1.\text{type} == \text{real} \text{ && } E_2.\text{type} == \text{int}$   
then  $\text{real}$   
elif  $E_1.\text{type} == \text{real} \text{ && } E_2.\text{type} == \text{real}$   
then  $\text{real}$

# Overloaded functions and operators

- Overloaded symbol has different meaning depending upon the context
- In math, `+` is overloaded; used for integer, real, complex, matrices
- In Ada, `()` is overloaded; used for array, function call, type conversion
- Overloading is resolved when a **unique** meaning for an occurrence of a symbol is determined

# Overloaded functions and operators

- In Ada standard interpretation of \* is multiplication of integers
- However, it may be overloaded by saying function “\*” (i, j: integer) return complex;  
function “\*” (i, j: complex) return complex;
- Possible type expression for “\*” include:  
integer **x** integer → integer  
integer **x** integer → complex  
complex **x** complex → complex

# Overloaded function resolution

- Suppose only possible type for  $2$ ,  $3$  and  $5$  is integer
- $Z$  is a complex variable
- $3*5$  is either integer or complex depending upon the context
  - in  $2*(3*5)$ :  $3*5$  is integer because  $2$  is integer
  - in  $Z*(3*5)$  :  $3*5$  is complex because  $Z$  is complex

# Type resolution

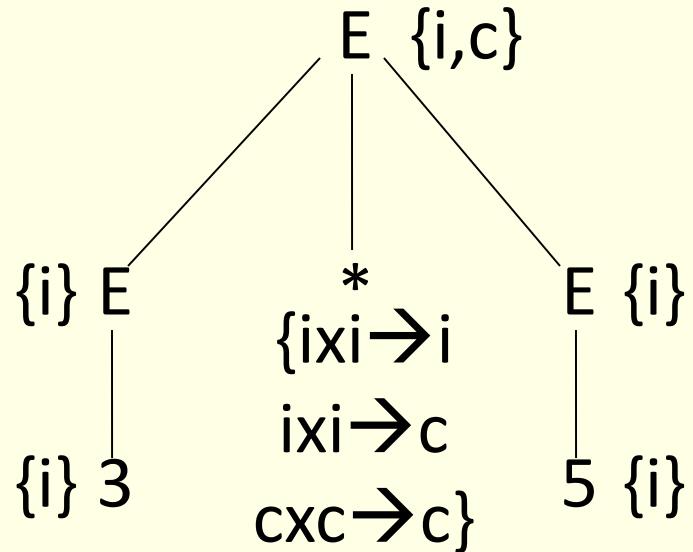
- Try all possible types of each overloaded function (possible but brute force method!)
- Keep track of all possible types
- Discard invalid possibilities
- At the end, check if there is a single unique type
- Overloading can be resolved in two passes:
  - Bottom up: compute set of all possible types for each expression
  - Top down: narrow set of possible types based on what could be used in an expression

# Determining set of possible types

$E' \rightarrow E$        $E'.types = E.types$

$E \rightarrow id$        $E.types = \text{lookup}(id)$

$E \rightarrow E_1(E_2)$        $E.types =$   
 $\{t \mid \exists s \text{ in } E_2.types \text{ && } s \rightarrow t \text{ is in } E_1.types\}$



# Narrowing the set of possible types

- Ada requires a complete expression to have a unique type
- Given a unique type from the context we can narrow down the type choices for each expression
- If this process does not result in a unique type for each sub expression then a type error is declared for the expression

# Narrowing the set of ...

$E' \rightarrow E$        $E'.types = E.types$

$E \rightarrow id$        $E.types = \text{lookup}(id)$

$E \rightarrow E_1(E_2)$        $E.types =$   
                                   $\{ t \mid \exists s \text{ in } E_2.types \ \&\& \ s \rightarrow t \text{ is in } E_1.types \}$

# Narrowing the set of ...

$E' \rightarrow E$	$E'.types = E.types$ $E.unique = \text{if } E'.types == \{t\} \text{ then } t$ else type_error
$E \rightarrow id$	$E.types = \text{lookup}(id)$
$E \rightarrow E_1(E_2)$	$E.types = \{ t \mid \exists s \text{ in } E_2.types \text{ && } s \rightarrow t \text{ is in } E_1.types\}$ $t = E.unique$ $S = \{s \mid s \in E2.types \text{ and } (s \rightarrow t) \in E1.types\}$ $E_2.unique = \text{if } S == \{s\} \text{ then } s \text{ else type_error}$ $E_1.unique = \text{if } S == \{s\} \text{ then } s \rightarrow t$ else type_error

# Polymorphic functions

- A function can be invoked with arguments of different types
- Built in operators for indexing arrays, applying functions, and manipulating pointers are usually polymorphic
- Extend type expressions to include expressions with type variables
- Facilitate the implementation of algorithms that manipulate data structures (regardless of types of elements)
  - Determine length of the list without knowing types of the elements

# Polymorphic functions ...

- Strongly typed languages can make programming very tedious
- Consider identity function written in a language like Pascal
  - `function identity (x: integer): integer;`
  - This function is the identity on integers: `int → int`
  - If we want to write identity function on char then we must write
    - `function identity (x: char): char;`
  - This is the same code; only types have changed. However, in Pascal a new identity function must be written for each type
  - Templates solve this problem somewhat, for end-users
    - For compiler, multiple definitions still present!

# Type variables

- Variables can be used in type expressions to represent unknown types
- **Important use:** check consistent use of an identifier in a language that does not require identifiers to be declared
- An inconsistent use is reported as an error
- If the variable is always used as of the same type then the use is consistent and has lead to type inference
- Type inference: determine the type of a variable/language construct from the way it is used
  - Infer type of a function from its body

```
function deref(p) { return *p; }
```

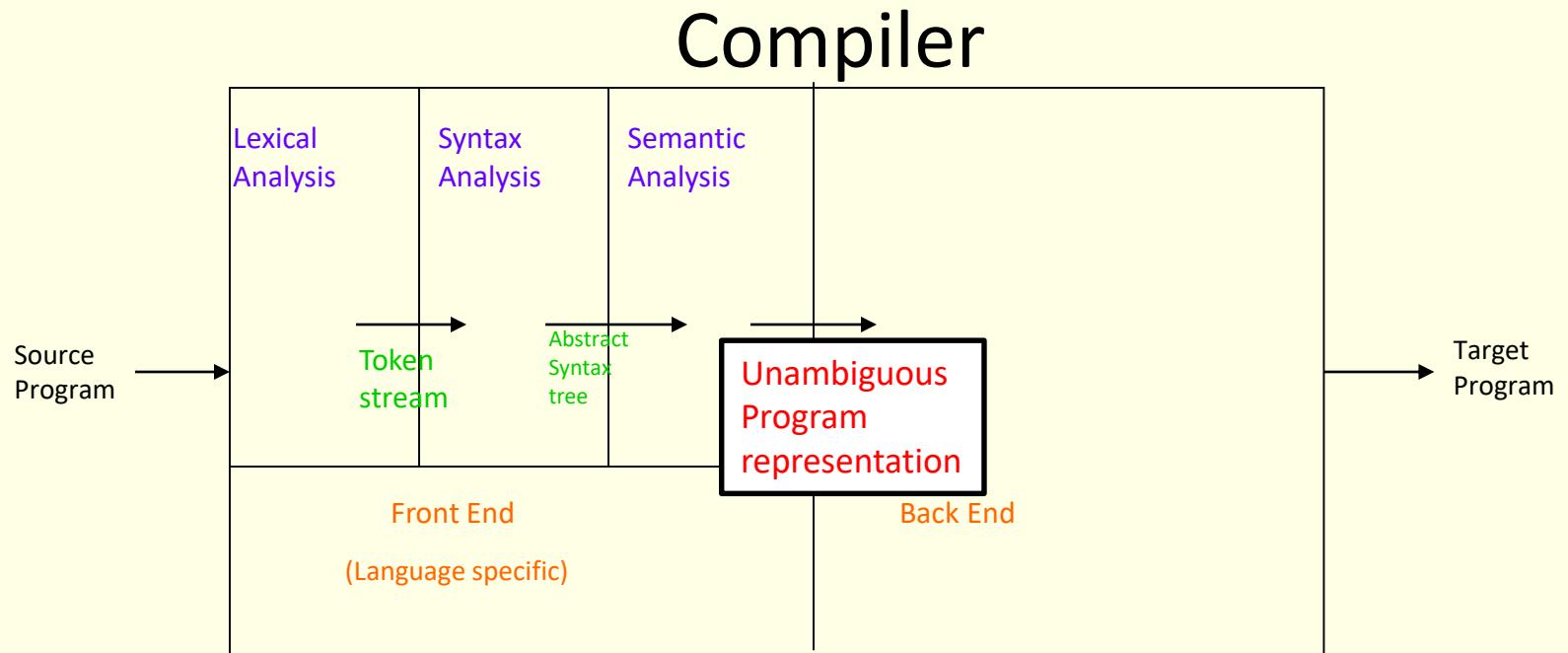
- Initially, nothing is known about type of p
  - Represent it by a type variable
- Operator `*` takes pointer to an object and returns the object
- Therefore, `p` must be pointer to an object of unknown type  $\alpha$ 
  - If type of `p` is represented by  $\beta$  then  
 $\beta = \text{pointer}(\alpha)$
  - Expression `*p` has type  $\alpha$
- Type expression for function `deref` is  
**for any type  $\alpha$ :  $\text{pointer}(\alpha) \rightarrow \alpha$**
- For identity function, the type expression is  
**for any type  $\alpha$ :  $\alpha \rightarrow \alpha$**

# Reading assignment

- Rest of Section 6.6 and Section 6.7 of Old Dragonbook [Aho, Sethi and Ullman]

# Principles of Compiler Design

## Intermediate Representation



# Intermediate Representation Design

- More of a wizardry rather than science
- Compiler commonly use 2-3 IRs
- HIR (high level IR) preserves loop structure and array bounds
- MIR (medium level IR) reflects range of features in a set of source languages
  - language independent
  - good for code generation for one or more architectures
  - appropriate for most optimizations
- LIR (low level IR) low level similar to the machines

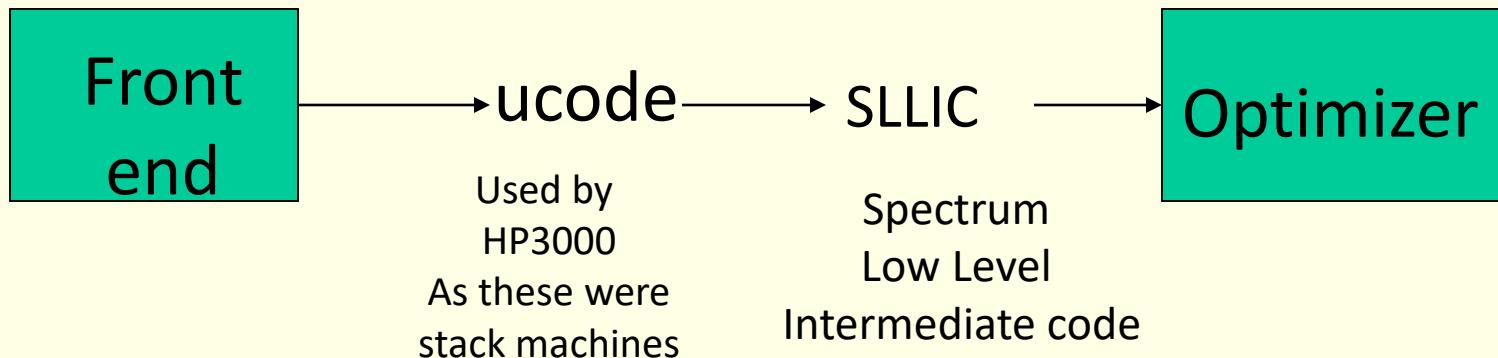
- Compiler writers have tried to define Universal IRs and have failed. (UNCOL in 1958)
- There is no standard Intermediate Representation. IR is a step in expressing a source program so that machine understands it
- As the translation takes place, IR is repeatedly analyzed and transformed
- Compiler users want analysis and translation to be fast and correct
- Compiler writers want optimizations to be simple to write, easy to understand and easy to extend

# Issues in IR Design

- source language and target language
- porting cost or reuse of existing design
- whether appropriate for optimizations
- U-code IR used on PA-RISC and Mips.  
Suitable for expression evaluation on stacks  
but less suited for load-store architectures
- both compilers translate U-code to another form
  - HP translates to very low level representation
  - Mips translates to MIR and translates back to U-code for code generator

# Issues in new IR Design

- how much machine dependent
- expressiveness: how many languages are covered
- appropriateness for code optimization
- appropriateness for code generation
- Use more than one IR (like in PA-RISC)



# Issues in new IR Design ...

- Use more than one IR for more than one optimization
- represent subscripts by list of subscripts: suitable for dependence analysis
- make addresses explicit in linearized form:
  - suitable for constant folding, strength reduction, loop invariant code motion, other basic optimizations

float a[20][10];  
use a[i][j+2]

HIR

$t1 \leftarrow a[i, j+2]$

MIR

$t1 \leftarrow j+2$

$t2 \leftarrow i * 20$

$t3 \leftarrow t1 + t2$

$t4 \leftarrow 4 * t3$

$t5 \leftarrow \text{addr } a$

$t6 \leftarrow t4 + t5$

$t7 \leftarrow *t6$

LIR

$r1 \leftarrow [fp-4]$

$r2 \leftarrow r1 + 2$

$r3 \leftarrow [fp-8]$

$r4 \leftarrow r3 * 20$

$r5 \leftarrow r4 + r2$

$r6 \leftarrow 4 * r5$

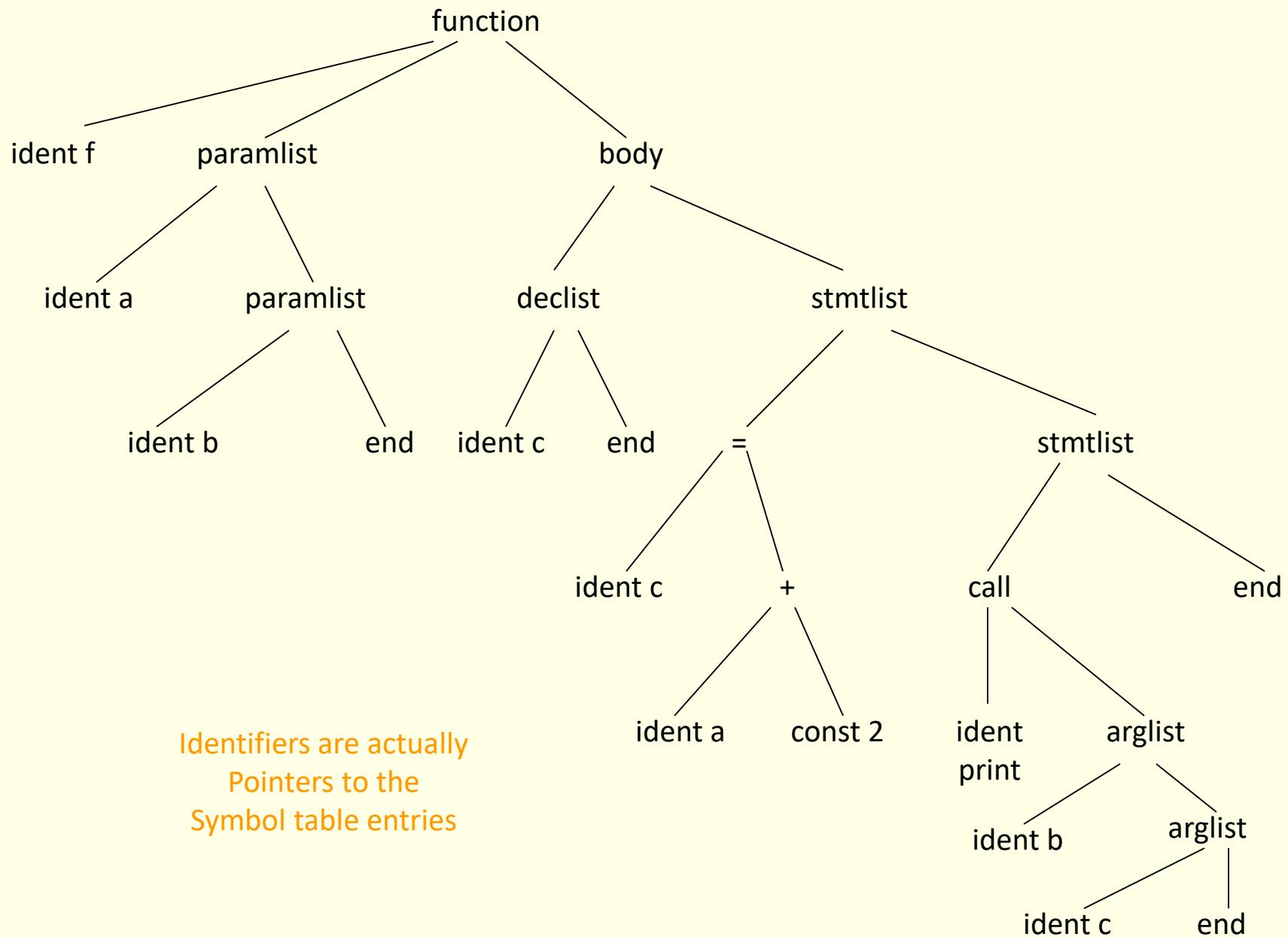
$r7 \leftarrow fp-216$

$f1 \leftarrow [r7 + r6]$

# High level IR

```
int f(int a, int b) {  
    int c;  
    c = a + 2;  
    print(b, c);  
}
```

- Abstract syntax tree
  - keeps enough information to reconstruct source form
  - keeps information about symbol table



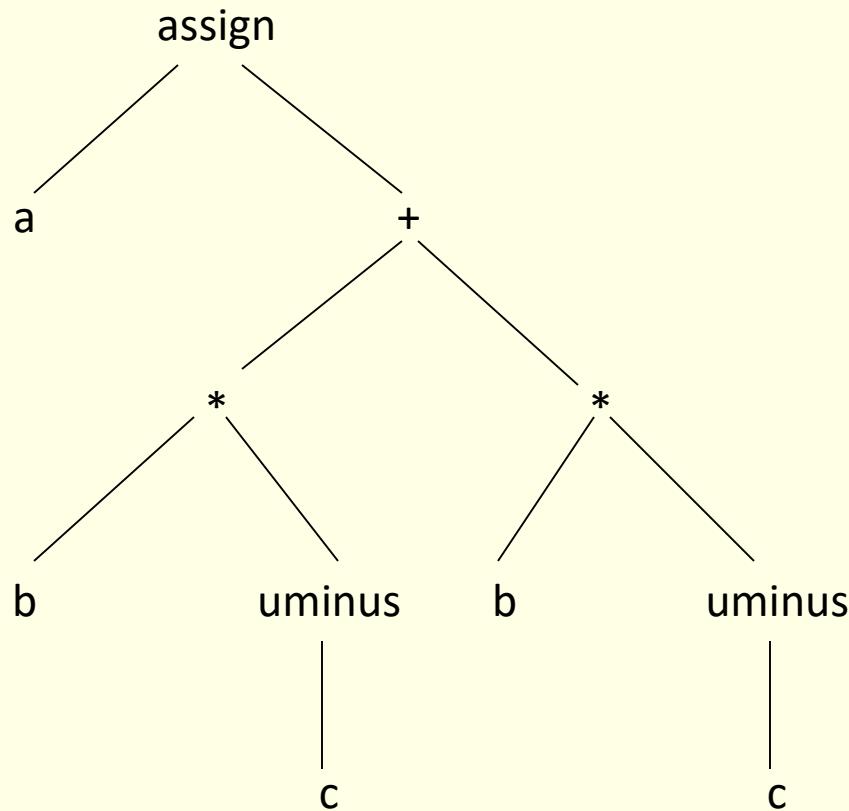
- Medium level IR
  - reflects range of features in a set of source languages
  - language independent
  - good for code generation for a number of architectures
  - appropriate for most of the optimizations
  - normally three address code
- Low level IR
  - corresponds one to one to target machine instructions
  - architecture dependent
- Multi-level IR
  - has features of MIR and LIR
  - may also have some features of HIR

# Abstract Syntax Tree/DAG

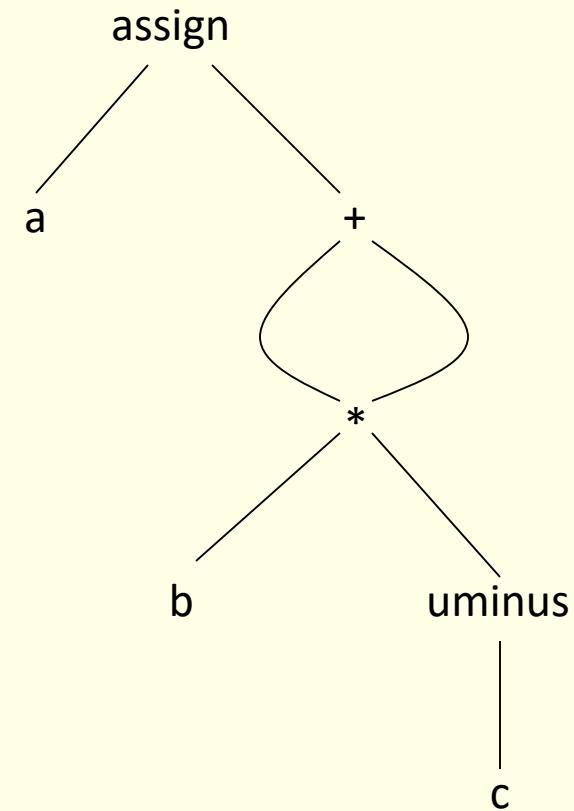
- Condensed form of a parse tree
- useful for representing language constructs
- Depicts the natural hierarchical structure of the source program
  - Each internal node represents an operator
  - Children of the nodes represent operands
  - Leaf nodes represent operands
- DAG is more compact than abstract syntax tree because common sub expressions are eliminated

$$a := b * -c + b * -c$$

Abstract syntax tree



Directed Acyclic Graph



# Three address code

- A linearized representation of a syntax tree where explicit names correspond to the interior nodes of the graph
- Sequence of statements of the general form

$$X := Y \text{ op } Z$$

- X, Y or Z are names, constants or compiler generated temporaries
- op stands for any operator such as a fixed- or floating-point arithmetic operator, or a logical operator
- Extensions to handle arrays, function call

# Three address code ...

- Only one operator on the right hand side is allowed
- Source expression like  $x + y * z$  might be translated into

$$t_1 := y * z$$

$$t_2 := x + t_1$$

where  $t_1$  and  $t_2$  are compiler generated temporary names

- Unraveling of complicated arithmetic expressions and of control flow makes 3-address code desirable for code generation and optimization
- The use of names for intermediate values allows 3-address code to be easily rearranged

# Three address instructions

- Assignment
  - $x = y \text{ op } z$
  - $x = \text{op } y$
  - $x = y$
- Jump
  - goto L
  - if  $x \text{ relop } y$  goto L
- Indexed assignment
  - $x = y[i]$
  - $x[i] = y$
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - $x = \&y$
  - $x = *y$
  - $*x = y$

# Other IRs

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes
- The list goes on .....

# Symbol Table

- Compiler uses symbol table to keep track of scope and binding information about names
- changes to table occur
  - if a new name is discovered
  - if new information about an existing name is discovered
- Symbol table must have mechanism to:
  - add new entries
  - find existing information efficiently

# Symbol Table

- Two common mechanism:
  - linear lists
    - simple to implement, poor performance
  - hash tables
    - greater programming/space overhead, good performance
- Compiler should be able to grow symbol table dynamically
  - If size is fixed, it must be large enough for the largest program

# Data Structures for SymTab

- List data structure
  - simplest to implement
  - use a single array to store names and information
  - search for a name is linear
  - entry and lookup are independent operations
  - cost of entry and search operations are very high and lot of time goes into book keeping
- Hash table
  - The advantages are obvious

# Symbol Table Entries

- each entry corresponds to a declaration of a name
- format need not be uniform because information depends upon the usage of the name
- each entry is a record consisting of consecutive words
  - If uniform records are desired, some entries may be kept outside the symbol table (e.g. variable length strings)

# Symbol Table Entries

- information is entered into symbol table at various times
  - keywords are entered initially
  - identifier lexemes are entered by lexical analyzer
  - attribute values are filled in as information is available
- a name may denote several objects in the same block

```
int x;  
struct x {float y, z; }
```

  - lexical analyzer returns the name itself and not pointer to symbol table entry
  - record in the symbol table is created when role of the name becomes clear
  - in this case two symbol table entries will be created

- attributes of a name are entered in response to declarations
- labels are often identified by colon (:)
- syntax of procedure/function specifies that certain identifiers are formals
- there is a distinction between token id, lexeme and attributes of the names
  - it is difficult to work with lexemes
  - if there is modest upper bound on length then lexemes can be stored in symbol table
  - if limit is large store lexemes separately

# Storage Allocation Information

- information about storage locations is kept in the symbol table
  - if target is assembly code then assembler can take care of storage for various names
- compiler needs to generate data definitions to be appended to assembly code
- if target is machine code then compiler does the allocation
- for names whose storage is allocated at runtime no storage allocation is done
  - compiler plans out activation records

# Representing Scope Information

- entries are declarations of names
- when a lookup is done, entry for appropriate declaration must be returned
- scope rules determine which entry is appropriate
- maintain separate table for each scope
- symbol table for a procedure or scope is compile time equivalent an activation record
- information about non local is found by scanning symbol table for the enclosing procedures
- symbol table can be attached to abstract syntax of the procedure (integrated into intermediate representation)

# Symbol attributes and symbol table entries

- Symbols have associated attributes
- typical attributes are name, type, scope, size, addressing mode etc.
- a symbol table entry collects together attributes such that they can be easily set and retrieved
- example of typical names in symbol table

Name	Type
name	character string
class	enumeration
size	integer
type	enumeration

# Nesting structure of an example Pascal program

```
program e;
var a, b, c: integer;

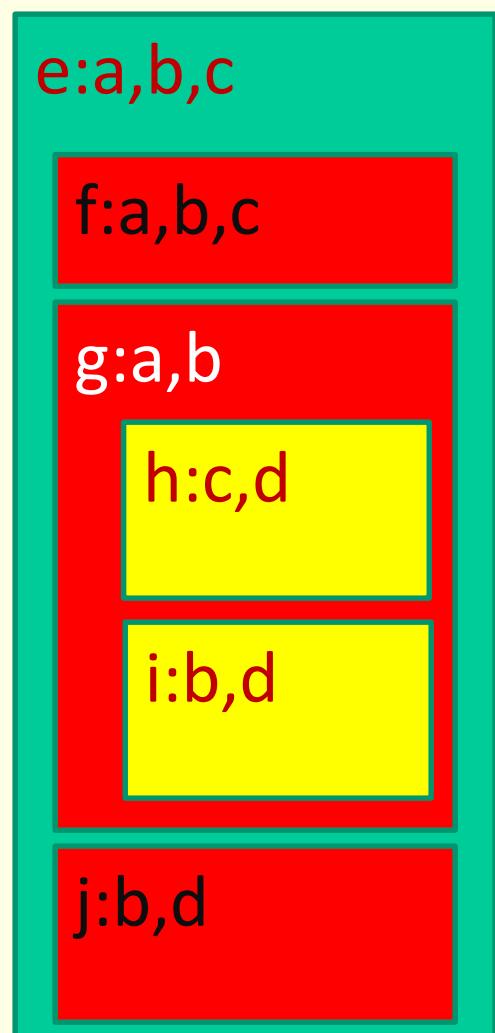
procedure f;
var a, b, c: integer;
begin
  a := b+c
end;

procedure g;
var a, b: integer;

procedure h;
var c, d: integer;
begin
  c := a+d
end;

procedure i;
var b, d: integer;
begin
  b:= a+c
end;
begin
  ....
end;

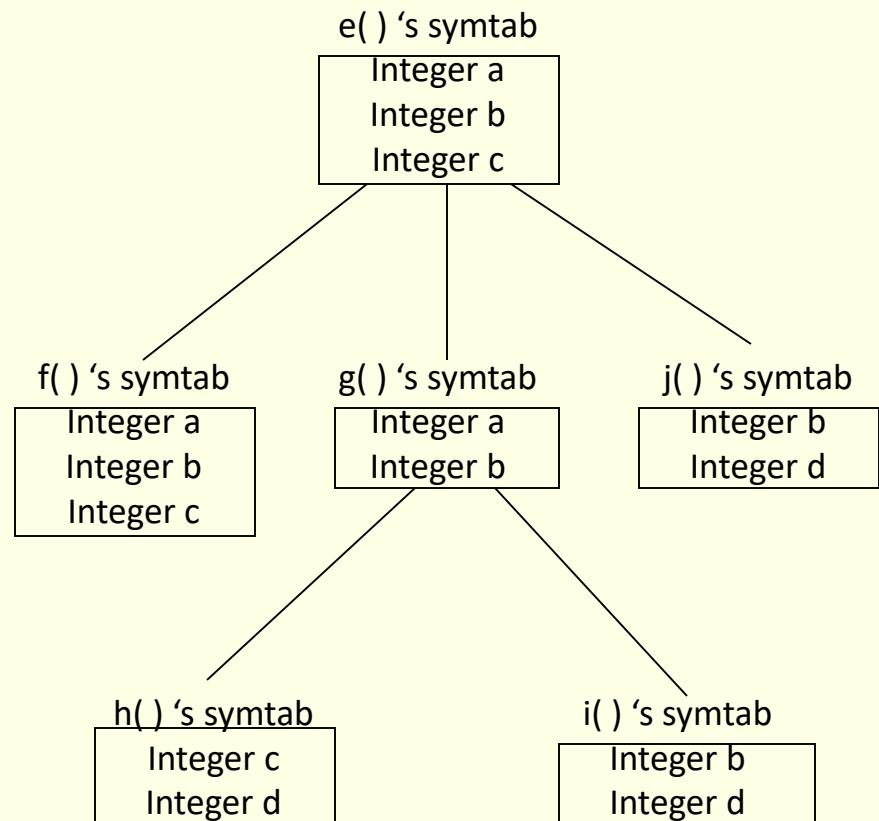
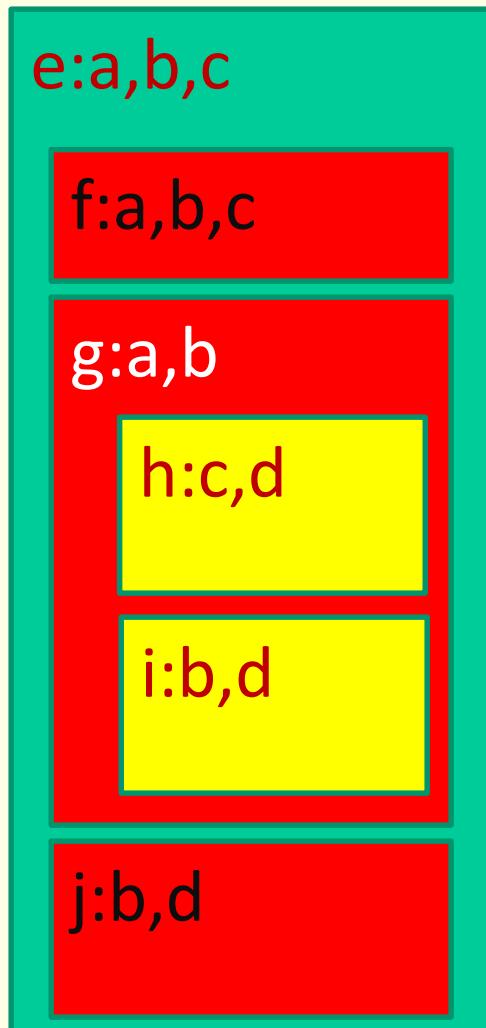
procedure j;
var b, d: integer;
begin
  b := a+d
end;
begin
  a := b+c
end.
```



# Global Symbol table structure

- scope and visibility rules determine the structure of global symbol table
- for Algol class of languages scoping rules structure the symbol table as tree of local tables
  - global scope as root
  - tables for nested scope as children of the table for the scope they are nested in

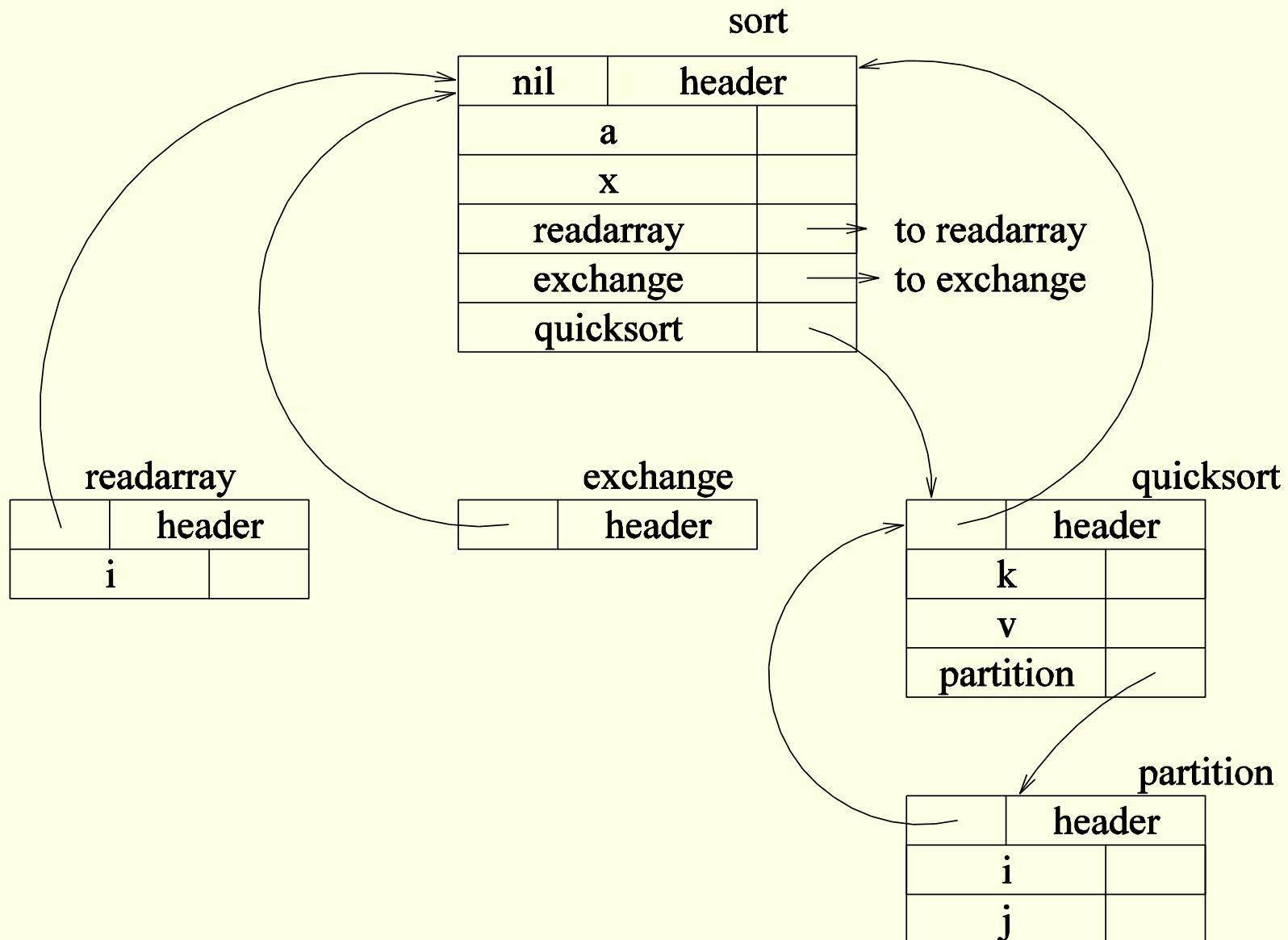
# Global Symbol table structure



# Example

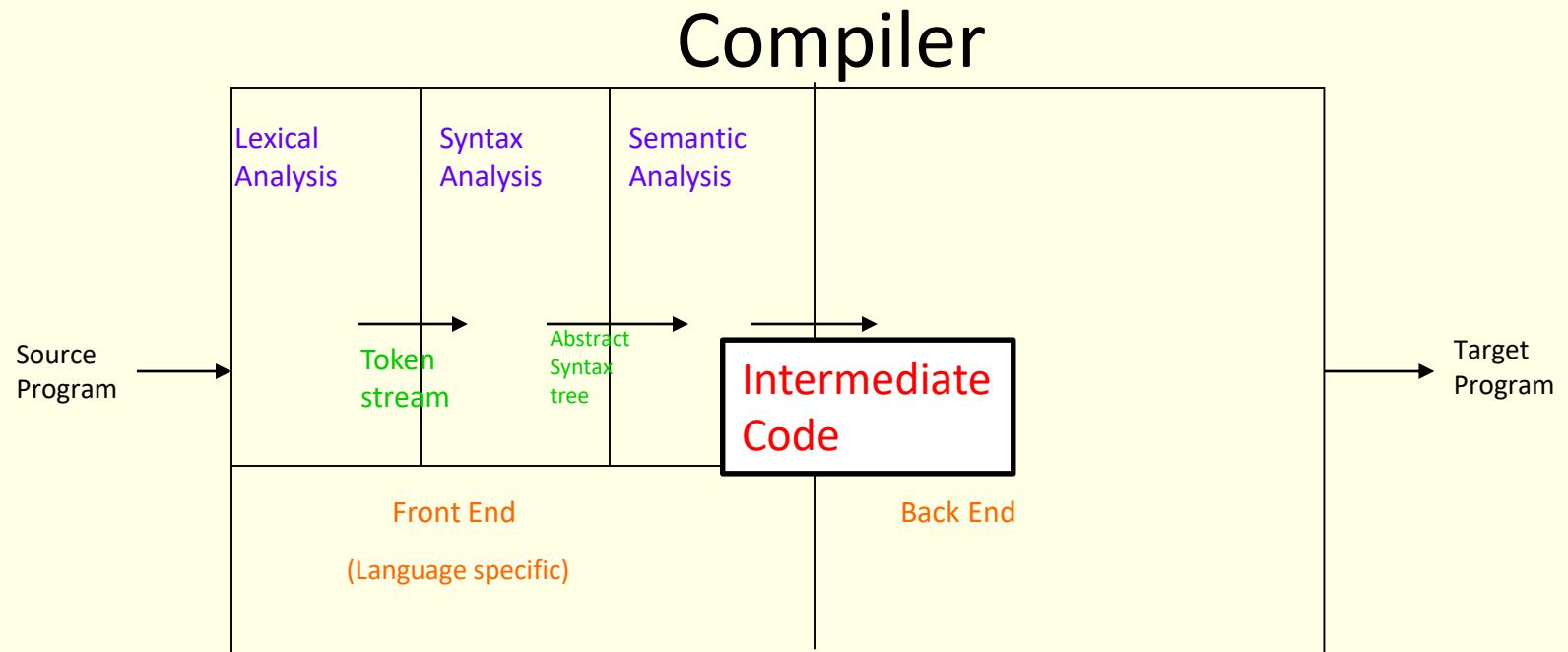
```
program sort;  
var a : array[0..10] of integer;  
  
procedure readarray;  
var i :integer;  
:  
  
procedure exchange(i, j  
:integer)  
:  
:
```

```
procedure quicksort (m, n :integer);  
var i :integer;  
function partition (y, z  
:integer) :integer;  
var i, j, x, v :integer;  
:  
i:= partition (m,n);  
quicksort (m,i-1);  
quicksort(i+1, n);  
:  
begin{main}  
readarray;  
quicksort(1,9)  
end.
```



# Principles of Compiler Design

## Intermediate Representation

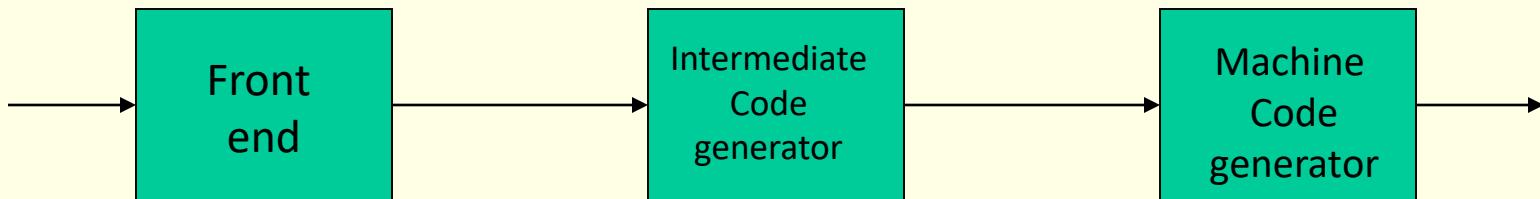


# Intermediate Code Generation

- Code generation is a mapping from source level abstractions to target machine abstractions
- Abstraction at the source level  
identifiers, operators, expressions, statements, conditionals, iteration, functions (user defined, system defined or libraries)
- Abstraction at the target level  
memory locations, registers, stack, opcodes, addressing modes, system libraries, interface to the operating systems

# Intermediate Code Generation ...

- Front end translates a source program into an intermediate representation
- Back end generates target code from intermediate representation
- Benefits
  - Retargeting is possible
  - Machine independent code optimization is possible



# Three address code

- Assignment
  - $x = y \text{ op } z$
  - $x = \text{op } y$
  - $x = y$
- Jump
  - goto L
  - if  $x \text{ relop } y$  goto L
- Indexed assignment
  - $x = y[i]$
  - $x[i] = y$
- Function
  - param x
  - call p,n
  - return y
- Pointer
  - $x = \&y$
  - $x = *y$
  - $*x = y$

# Syntax directed translation of expression into 3-address code

- Two attributes
- $E.place$ , a name that will hold the value of E, and
- $E.code$ , the sequence of three-address statements evaluating E.
- A function **gen(...)** to produce sequence of three address statements
  - The statements themselves are kept in some data structure, e.g. list
  - SDD operations described using pseudo code

# Syntax directed translation of expression into 3-address code

$S \rightarrow id := E$

S.code := E.code ||  
gen(id.place:= E.place)

$E \rightarrow E_1 + E_2$

E.place:= newtmp  
E.code:= E<sub>1</sub>.code || E<sub>2</sub>.code ||  
gen(E.place := E<sub>1</sub>.place + E<sub>2</sub>.place)

$E \rightarrow E_1 * E_2$

E.place:= newtmp  
E.code := E<sub>1</sub>.code || E<sub>2</sub>.code ||  
gen(E.place := E<sub>1</sub>.place \* E<sub>2</sub>.place)

# Syntax directed translation of expression ...

$E \rightarrow -E_1$

E.place := newtmp  
E.code := E<sub>1</sub>.code ||  
gen(E.place := - E<sub>1</sub>.place)

$E \rightarrow (E_1)$

E.place := E<sub>1</sub>.place  
E.code := E<sub>1</sub>.code

$E \rightarrow id$

E.place := id.place  
E.code := ‘ ’

# Example

For  $a = b * -c + b * -c$

following code is generated

$$t_1 = -c$$

$$t_2 = b * t_1$$

$$t_3 = -c$$

$$t_4 = b * t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

# Flow of Control

$S \rightarrow \text{while } E \text{ do } S_1$

Desired Translation is

$S.\text{begin} :$

$E.\text{code}$

  if  $E.\text{place} = 0$  goto  $S.\text{after}$

$S_1.\text{code}$

  goto  $S.\text{begin}$

$S.\text{after} :$

$S.\text{begin} := \text{newlabel}$

$S.\text{after} := \text{newlabel}$

$S.\text{code} := \text{gen}(S.\text{begin}:) \parallel$   
   $E.\text{code} \parallel$   
   $\text{gen}(\text{if } E.\text{place} = 0 \text{ goto } S.\text{after}) \parallel$   
   $S_1.\text{code} \parallel$   
   $\text{gen}(\text{goto } S.\text{begin}) \parallel$   
   $\text{gen}(S.\text{after}:)$

# Flow of Control ...

$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

E.code

if E.place = 0 goto S.else

$S_1\text{.code}$

goto S.after

**S.else:**

$S_2\text{.code}$

**S.after:**

S.else := newlabel

S.after := newlabel

S.code = E.code ||

gen(if E.place = 0 goto S.else) ||

$S_1\text{.code}$  ||

gen(goto S.after) ||

gen(S.else :) ||

$S_2\text{.code}$  ||

gen(S.after :)

# Declarations

P → D

D → D ; D

D → id : T

T → integer

T → real

# Declarations

For each name create symbol table entry with information like type and relative address

P → D

D → D ; D

D → id : T

enter(id.name, T.type, offset);  
offset = offset + T.width

T → integer

T.type = integer; T.width = 4

T → real

T.type = real; T.width = 8

# Declarations

For each name create symbol table entry with information like type and relative address

P → {offset=0} D

D → D ; D

D → id : T

enter(id.name, T.type, offset);  
offset = offset + T.width

T → integer

T.type = integer; T.width = 4

T → real

T.type = real; T.width = 8

# Declarations ...

$T \rightarrow \text{array} [ \text{num} ] \text{ of } T_1$

$T.\text{type} = \text{array}(\text{num.val}, T_1.\text{type})$

$T.\text{width} = \text{num.val} \times T_1.\text{width}$

$T \rightarrow \uparrow T_1$

$T.\text{type} = \text{pointer}(T_1.\text{type})$

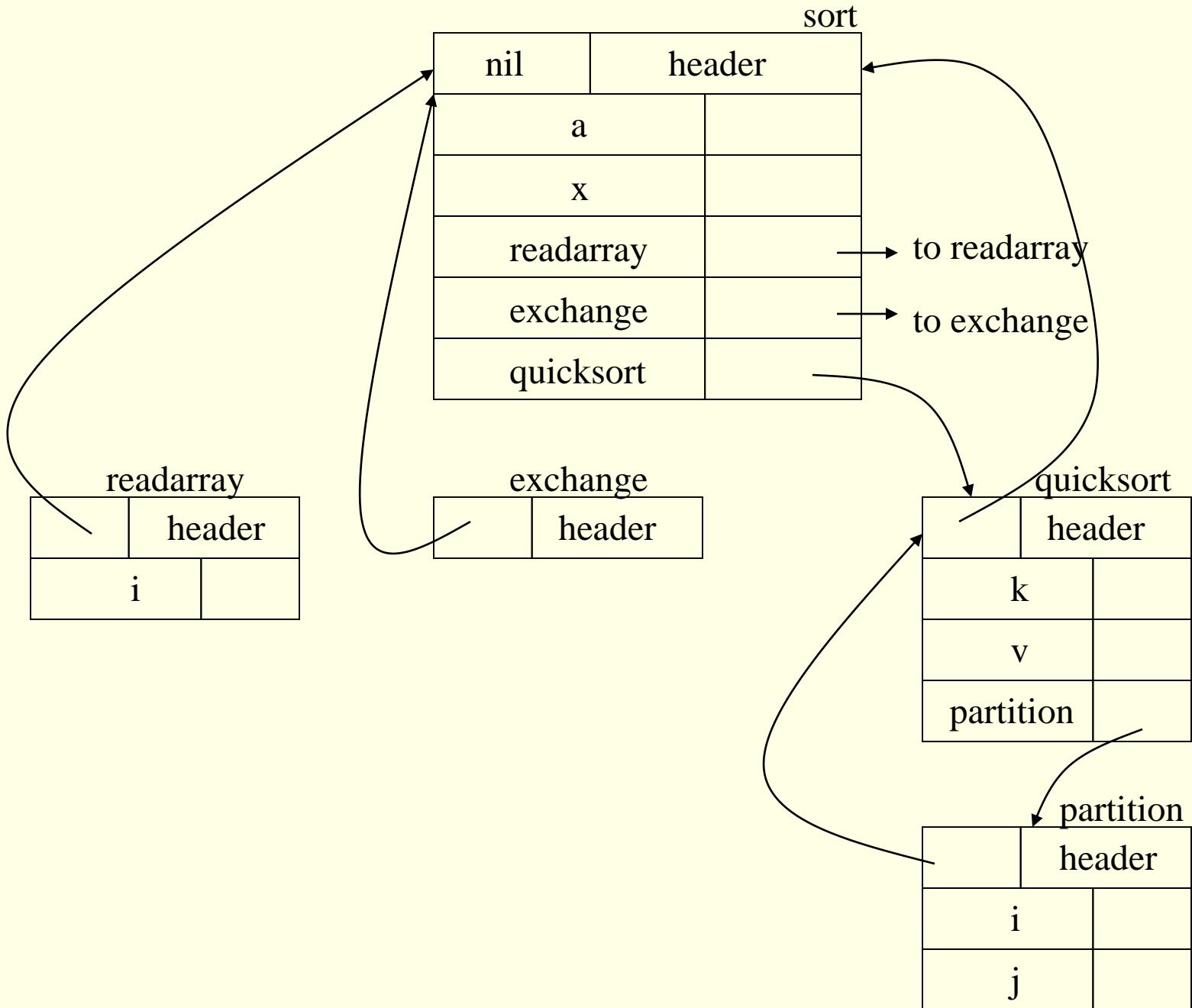
$T.\text{width} = 4$

# Keeping track of local information

- when a nested procedure is seen, processing of declaration in enclosing procedure is temporarily suspended
- assume following language
  - $P \rightarrow D$
  - $D \rightarrow D ; D \mid id : T \mid proc\ id\ ; D \mid S$
- a new symbol table is created when procedure declaration
  - $D \rightarrow proc\ id\ ; D_1 \mid S$  is seen
- entries for  $D_1$  are created in the new symbol table
- the name represented by id is local to the enclosing procedure

# Example

```
program sort;
    var a : array[1..n] of integer;
        x : integer;
    procedure readarray;
        var i : integer;
        .....
    procedure exchange(i,j:integers);
        .....
    procedure quicksort(m,n : integer);
        var k,v : integer;
        function partition(x,y:integer):integer;
            var i,j: integer;
            .....
        .....
begin{main}
    .....
end.
```



# Creating symbol table: Interface

- **mktble (previous)**  
create a new symbol table and return a pointer to the new table. The argument previous points to the enclosing procedure
- **enter (table, name, type, offset)**  
creates a new entry
- **addwidth (table, width)**  
records cumulative width of all the entries in a table
- **enterproc (table, name, newtable)**  
creates a new entry for procedure name. newtable points to the symbol table of the new procedure
- Maintain two stacks: (1) symbol tables and (2) offsets
- Standard stack operations: push, pop, top

# Creating symbol table ...

D → proc id;  
    {t = mkttable(top(tblptr));  
      push(t, tblptr); push(0, offset)}

D<sub>1</sub>; S  
    {t = top(tblptr);  
      addwidth(t, top(offset));  
      pop(tblptr); pop(offset);  
      enterproc(top(tblptr), id.name, t)}

D → id: T  
    {enter(top(tblptr), id.name, T.type, top(offset));  
      top(offset) = top (offset) + T.width}

# Creating symbol table ...

P →

```
{t=mkttable(nil);  
push(t,tblptr);  
push(0,offset)}
```

D

```
{addwidth(top(tblptr),top(offset));  
pop(tblptr); // save it somewhere!  
pop(offset)}
```

D → D ; D

# Field names in records

T → record

```
{t = mkttable(nil);  
 push(t, tblptr); push(0, offset)}
```

D end

```
{T.type = record(top(tblptr));  
 T.width = top(offset);  
 pop(tblptr); pop(offset)}
```

# Names in the Symbol table

$S \rightarrow id := E$

```
{p = lookup(id.place);  
if p <> nil then emit(p := E.place)  
else error}
```

$E \rightarrow id$

```
{p = lookup(id.name);  
if p <> nil then E.place = p  
else error}
```

emit is like gen, but instead of returning code, it generates code as a side effect in a list of three address instructions.

# Type conversion within assignments

$E \rightarrow E_1 + E_2$

E.place = newtmp;

if  $E_1.type = \text{integer}$  and  $E_2.type = \text{integer}$

then emit(E.place ':=' E<sub>1</sub>.place 'int+' E<sub>2</sub>.place);

E.type = integer;

...

similar code if both  $E_1.type$  and  $E_2.type$  are real

...

else if  $E_1.type = \text{int}$  and  $E_2.type = \text{real}$

then

u = newtmp;

emit(u ':=' inttoreal E<sub>1</sub>.place);

emit(E.place ':=' u 'real+' E<sub>2</sub>.place);

E.type = real;

...

similar code if  $E_1.type$  is real and  $E_2.type$  is integer

# Example

```
real x, y;  
int i, j;  
x = y + i * j
```

generates code

```
t1 = i int* j  
t2 = inttoreal t1  
t3 = y real+ t2  
x = t3
```

# Boolean Expressions

- compute logical values
- change the flow of control
- boolean operators are: and or not

$$\begin{array}{l} E \rightarrow E \text{ or } E \\ | \quad E \text{ and } E \\ | \quad \text{not } E \\ | \quad (E) \\ | \quad \text{id relop id} \\ | \quad \text{true} \\ | \quad \text{false} \end{array}$$

# Methods of translation

- Evaluate similar to arithmetic expressions
  - Normally use 1 for true and 0 for false
- implement by flow of control
  - given expression  $E_1$  or  $E_2$ 
    - if  $E_1$  evaluates to true
    - then  $E_1$  or  $E_2$  evaluates to true
    - without evaluating  $E_2$

# Numerical representation

- a or b and not c

$$t_1 = \text{not } c$$

$$t_2 = b \text{ and } t_1$$

$$t_3 = a \text{ or } t_2$$

- relational expression  $a < b$  is equivalent to  
if  $a < b$  then 1 else 0

1. if  $a < b$  goto 4.

2.  $t = 0$

3. goto 5

4.  $t = 1$

- 5.

# Syntax directed translation of boolean expressions

$E \rightarrow E_1 \text{ or } E_2$

E.place := newtmp

emit(E.place ':=' E<sub>1</sub>.place 'or' E<sub>2</sub>.place)

$E \rightarrow E_1 \text{ and } E_2$

E.place:= newtmp

emit(E.place ':=' E<sub>1</sub>.place 'and' E<sub>2</sub>.place)

$E \rightarrow \text{not } E_1$

E.place := newtmp

emit(E.place ':=' 'not' E<sub>1</sub>.place)

$E \rightarrow (E_1)$

E.place = E<sub>1</sub>.place

# Syntax directed translation of boolean expressions

$E \rightarrow id1 \text{ relop } id2$

```
E.place := newtmp  
emit(if id1.place relop id2.place goto nextstat+3)  
emit(E.place = 0)  
emit(goto nextstat+2)  
emit(E.place = 1)
```

$E \rightarrow \text{true}$

```
E.place := newtmp  
emit(E.place = '1')
```

$E \rightarrow \text{false}$

```
E.place := newtmp  
emit(E.place = '0')
```

"nextstat" is a global variable; a pointer to the statement to be emitted. emit also updates the nextstat as a side-effect.

# Example:

## Code for $a < b$ or $c < d$ and $e < f$

100: if  $a < b$  goto 103

101:  $t_1 = 0$

102: goto 104

103:  $t_1 = 1$

104:

if  $c < d$  goto 107

105:  $t_2 = 0$

106: goto 108

107:  $t_2 = 1$

108:

if  $e < f$  goto 111

109:  $t_3 = 0$

110: goto 112

111:  $t_3 = 1$

112:

$t_4 = t_2$  and  $t_3$

113:  $t_5 = t_1$  or  $t_4$

# Short Circuit Evaluation of boolean expressions

- Translate boolean expressions without:
  - generating code for boolean operators
  - evaluating the entire expression

- Flow of control statements

$S \rightarrow \text{if } E \text{ then } S_1$   
|  $\text{if } E \text{ then } S_1 \text{ else } S_2$   
|  $\text{while } E \text{ do } S_1$

Each Boolean expression  $E$  has two attributes, **true** and **false**. These attributes hold the label of the **target stmt** to jump to.

# Control flow translation of boolean expression

if E is of the form:  $a < b$

then code is of the form: if  $a < b$  goto E.true  
                                  goto E.false

$E \rightarrow id_1 \text{ relop } id_2$

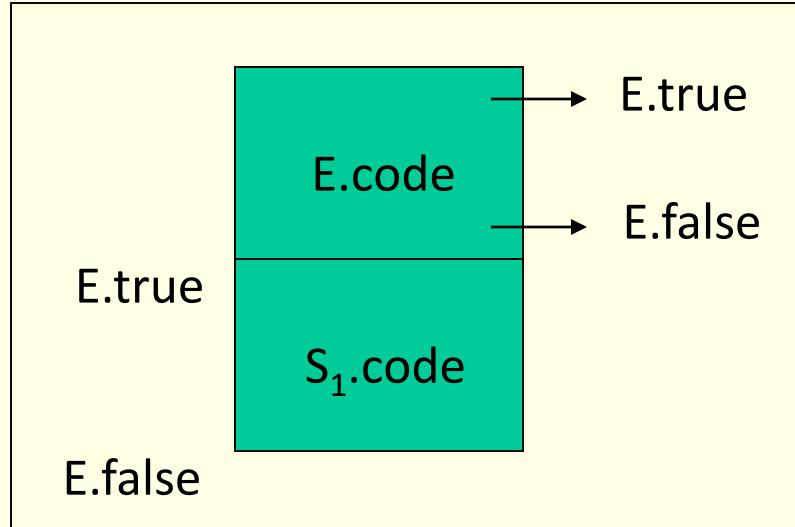
$E.\text{code} = \text{gen}(\text{ if } id_1 \text{ relop } id_2 \text{ goto } E.\text{true}) \parallel$   
 $\text{gen}(\text{goto } E.\text{false})$

$E \rightarrow \text{true}$

$E.\text{code} = \text{gen}(\text{goto } E.\text{true})$

$E \rightarrow \text{false}$

$E.\text{code} = \text{gen}(\text{goto } E.\text{false})$



$S \rightarrow \text{if } E \text{ then } S_1$

$E.\text{true} = \text{newlabel}$

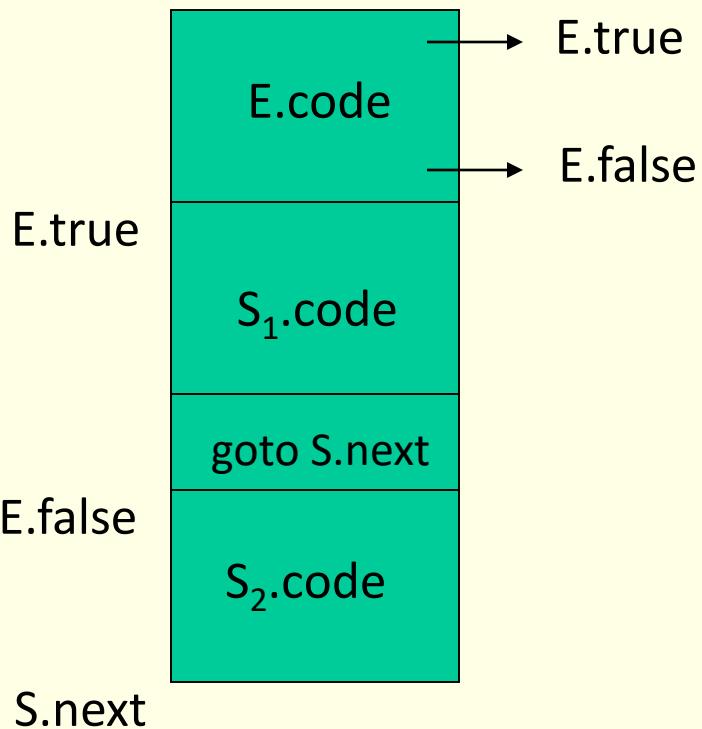
$E.\text{false} = S.\text{next}$

$S_1.\text{next} = S.\text{next}$

$S.\text{code} = E.\text{code} \parallel$

$\text{gen}(E.\text{true} ':') \parallel$

$S_1.\text{code}$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

$E.\text{true} = \text{newlabel}$

$E.\text{false} = \text{newlabel}$

$S_1.\text{next} = S.\text{next}$

$S_2.\text{next} = S.\text{next}$

$S.\text{code} = E.\text{code} \mid \mid$

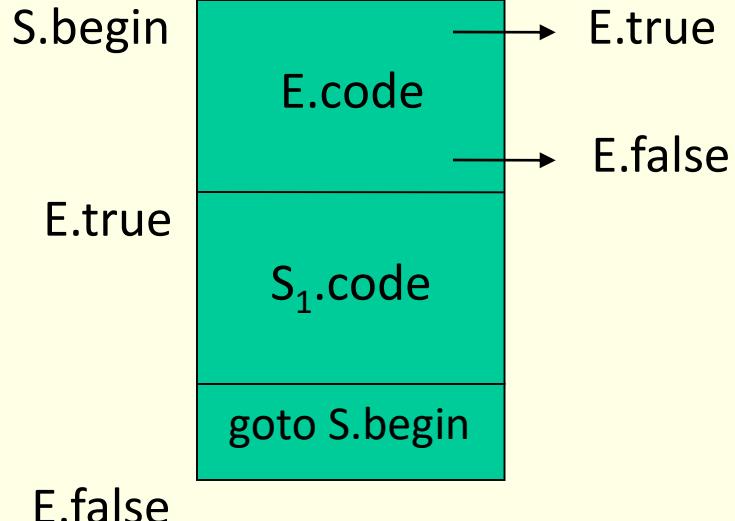
$\text{gen}(E.\text{true} ':') \mid \mid$

$S_1.\text{code} \mid \mid$

$\text{gen}(\text{goto } S.\text{next}) \mid \mid$

$\text{gen}(E.\text{false} ':') \mid \mid$

$S_2.\text{code}$



$S \rightarrow \text{while } E \text{ do } S_1$

$S.\text{begin} = \text{newlabel}$

$E.\text{true} = \text{newlabel}$

$E.\text{false} = S.\text{next}$

$S_1.\text{next} = S.\text{begin}$

$S.\text{code} = \text{gen}(S.\text{begin} ':') || E.\text{code} ||$

$\text{gen}(E.\text{true} ':') ||$

$S_1.\text{code} ||$

$\text{gen}(\text{goto } S.\text{begin})$

# Control flow translation of boolean expression

$E \rightarrow E_1 \text{ or } E_2$

$E_1.\text{true} := E.\text{true}$

$E_1.\text{false} := \text{newlabel}$

$E_2.\text{true} := E.\text{true}$

$E_2.\text{false} := E.\text{false}$

$E.\text{code} := E_1.\text{code} || \text{gen}(E_1.\text{false}) || E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.\text{true} := \text{newlabel}$

$E_1.\text{false} := E.\text{false}$

$E_2.\text{true} := E.\text{true}$

$E_2.\text{false} := E.\text{false}$

$E.\text{code} := E_1.\text{code} || \text{gen}(E_1.\text{true}) || E_2.\text{code}$

# Control flow translation of boolean expression ...

$E \rightarrow \text{not } E_1$

$E_1.\text{true} := E.\text{false}$

$E_1.\text{false} := E.\text{true}$

$E.\text{code} := E_1.\text{code}$

$E \rightarrow (E_1)$

$E_1.\text{true} := E.\text{true}$

$E_1.\text{false} := E.\text{false}$

$E.\text{code} := E_1.\text{code}$

# Example

Code for     $a < b$  or  $c < d$  and  $e < f$

    if  $a < b$  goto Ltrue

    goto L1

L1:    if  $c < d$  goto L2

    goto Lfalse

L2:    if  $e < f$  goto Ltrue

    goto Lfalse

Ltrue:

Lfalse:

# Example ...

Code for

```
while a < b do
    if c < d then x=y+z
    else      x=y-z
```

L1: if a < b goto L2  
 goto Lnext

L2: if c < d goto L3  
 goto L4

L3:  $t_1 = Y + Z$   
  $X = t_1$   
 goto L1

L4:  $t_1 = Y - Z$   
  $X = t_1$   
 goto L1

Lnext:

# Case Statement

- switch expression
  - begin
    - case value: statement
    - case value: statement
    - ....
    - case value: statement
    - default: statement
  - end

- evaluate the expression
- find which value in the list of cases is the same as the value of the expression.
  - Default value matches the expression if none of the values explicitly mentioned in the cases matches the expression
- execute the statement associated with the value found

# Translation

```
code to evaluate E into t  
if t <> V1 goto L1  
code for S1  
goto next  
L1    if t <> V2 goto L2  
      code for S2  
      goto next  
L2: .....  
Ln-2  if t <> Vn-1 goto Ln-1  
      code for Sn-1  
      goto next  
Ln-1: code for Sn  
next:
```

```
code to evaluate E into t  
      goto test  
L1: code for S1  
      goto next  
L2: code for S2  
      goto next  
.....  
Ln: code for Sn  
      goto next  
test:   if t = V1 goto L1  
          if t = V2 goto L2  
          ....  
          if t = Vn-1 goto Ln-1  
          goto Ln  
next:
```

# BackPatching

- way to implement boolean expressions and flow of control statements in one pass
- code is generated as quadruples into an array
- labels are indices into this array
- **makelist(i)**: create a newlist containing only i, return a pointer to the list.
- **merge(p1,p2)**: merge lists pointed to by p1 and p2 and return a pointer to the concatenated list
- **backpatch(p,i)**: insert i as the target label for the statements in the list pointed to by p

# Boolean Expressions

$E \rightarrow E_1 \text{ or } M \quad E_2$

|  $E_1 \text{ and } M \quad E_2$

|  $\text{not } E_1$

|  $(E_1)$

|  $\text{id}_1 \text{ relop } id_2$

|  $\text{true}$

|  $\text{false}$

$M \rightarrow \epsilon$

- Insert a marker non terminal M into the grammar to pick up index of next quadruple.
- attributes **truelist** and **falselist** are used to generate jump code for boolean expressions
- incomplete jumps are placed on lists pointed to by E.truelist and E.falselist

# Boolean expressions ...

- Consider  $E \rightarrow E_1$  and  $M \ E_2$ 
  - if  $E_1$  is false then  $E$  is also false so statements in  $E_1.\text{falselist}$  become part of  $E.\text{falselist}$
  - if  $E_1$  is true then  $E_2$  must be tested so target of  $E_1.\text{truelist}$  is beginning of  $E_2$
  - target is obtained by marker  $M$
  - attribute  $M.\text{quad}$  records the number of the first statement of  $E_2.\text{code}$

$E \rightarrow E_1 \text{ or } M E_2$

backpatch( $E_1.\text{falselist}$ ,  $M.\text{quad}$ )

$E.\text{truelist} = \text{merge}(E_1.\text{truelist}, E_2.\text{truelist})$

$E.\text{falselist} = E_2.\text{falselist}$

$E \rightarrow E_1 \text{ and } M E_2$

backpatch( $E_1.\text{truelist}$ ,  $M.\text{quad}$ )

$E.\text{truelist} = E_2.\text{truelist}$

$E.\text{falselist} = \text{merge}(E_1.\text{falselist}, E_2.\text{falselist})$

$E \rightarrow \text{not } E_1$

$E.\text{truelist} = E_1.\text{falselist}$

$E.\text{falselist} = E_1.\text{truelist}$

$E \rightarrow ( E_1 )$

$E.\text{truelist} = E_1.\text{truelist}$

$E.\text{falselist} = E_1.\text{falselist}$

$E \rightarrow id_1 \text{ relop } id_2$

E.truelist = makelist(nextquad)

E.falselist = makelist(nextquad+ 1)

emit(if id<sub>1</sub> relop id<sub>2</sub> goto --- )

emit(goto ---)

$E \rightarrow \text{true}$

E.truelist = makelist(nextquad)

emit(goto ---)

$E \rightarrow \text{false}$

E.falselist = makelist(nextquad)

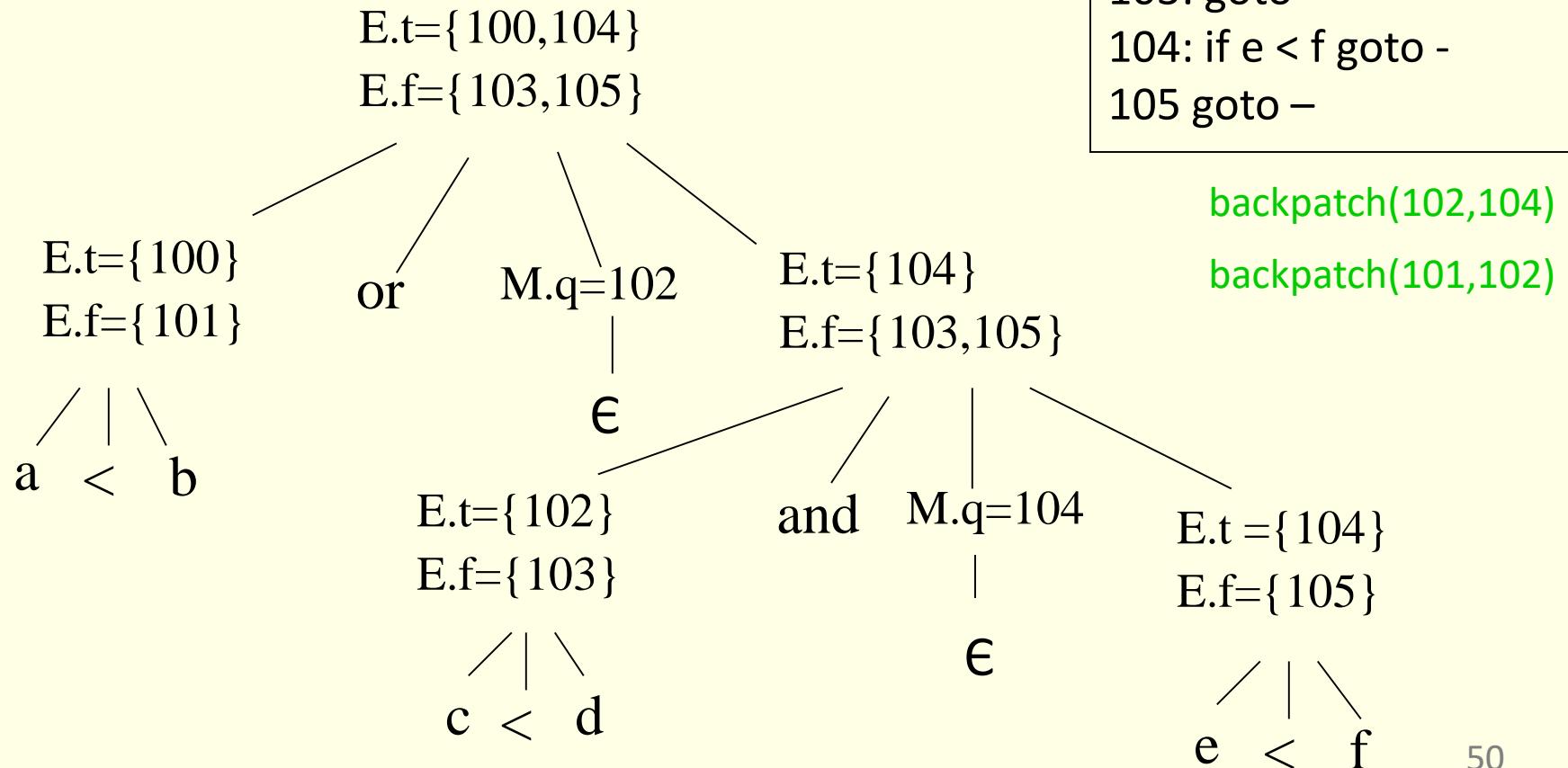
emit(goto ---)

$M \rightarrow \epsilon$

M.quad = nextquad

# Generate code for a < b or c < d and e < f

Initialize nextquad to 100



# Flow of Control Statements

$$\begin{aligned} S \rightarrow & \text{ if } E \text{ then } S_1 \\ | & \text{ if } E \text{ then } S_1 \text{ else } S_2 \\ | & \text{ while } E \text{ do } S_1 \\ | & \text{ begin L end} \\ | & A \end{aligned}$$
$$\begin{aligned} L \rightarrow & L ; S \\ | & S \end{aligned}$$

S : Statement

A : Assignment

L : Statement list

# Scheme to implement translation

- E has attributes truelist and falselist
- L and S have a list of unfilled quadruples to be filled by backpatching
- $S \rightarrow \text{while } E \text{ do } S_1$   
requires labels S.begin and E.true
  - markers  $M_1$  and  $M_2$  record these labels
  - $S \rightarrow \text{while } M_1 E \text{ do } M_2 S_1$
  - when while. . . is reduced to S
    - backpatch  $S_1.\text{nextlist}$  to make target of all the statements to  $M_1.\text{quad}$
    - E.truelist is backpatched to go to the beginning of  $S_1$  ( $M_2.\text{quad}$ )

## Scheme to implement translation ...

$S \rightarrow \text{if } E \text{ then } M S_1$   
    backpatch(E.truelist, M.quad)  
 $S.\text{nextlist} = \text{merge}(E.\text{falselist},$   
                                 $S_1.\text{nextlist})$

$S \rightarrow \text{if } E \text{ then } M_1 S_1 \text{ N else } M_2 S_2$   
    backpatch(E.truelist, M<sub>1</sub>.quad)  
    backpatch(E.falselist, M<sub>2</sub>.quad )  
 $S.\text{next} = \text{merge}(S_1.\text{nextlist},$   
                                 $N.\text{nextlist},$   
                                 $S_2.\text{nextlist})$

## Scheme to implement translation ...

```
S → while M1 E do M2 S1
    backpatch(S1.nextlist, M1.quad)
    backpatch(E.truelist, M2.quad)
    S.nextlist = E.falselist
    emit(goto M1.quad)
```

## Scheme to implement translation ...

$S \rightarrow \text{begin } L \text{ end}$   $S.\text{nextlist} = L.\text{nextlist}$

$S \rightarrow A$   $S.\text{nextlist} = \text{makelist}()$

$L \rightarrow L_1 ; M S$   $\text{backpatch}(L_1.\text{nextlist},$   
 $M.\text{quad})$

$L.\text{nextlist} = S.\text{nextlist}$

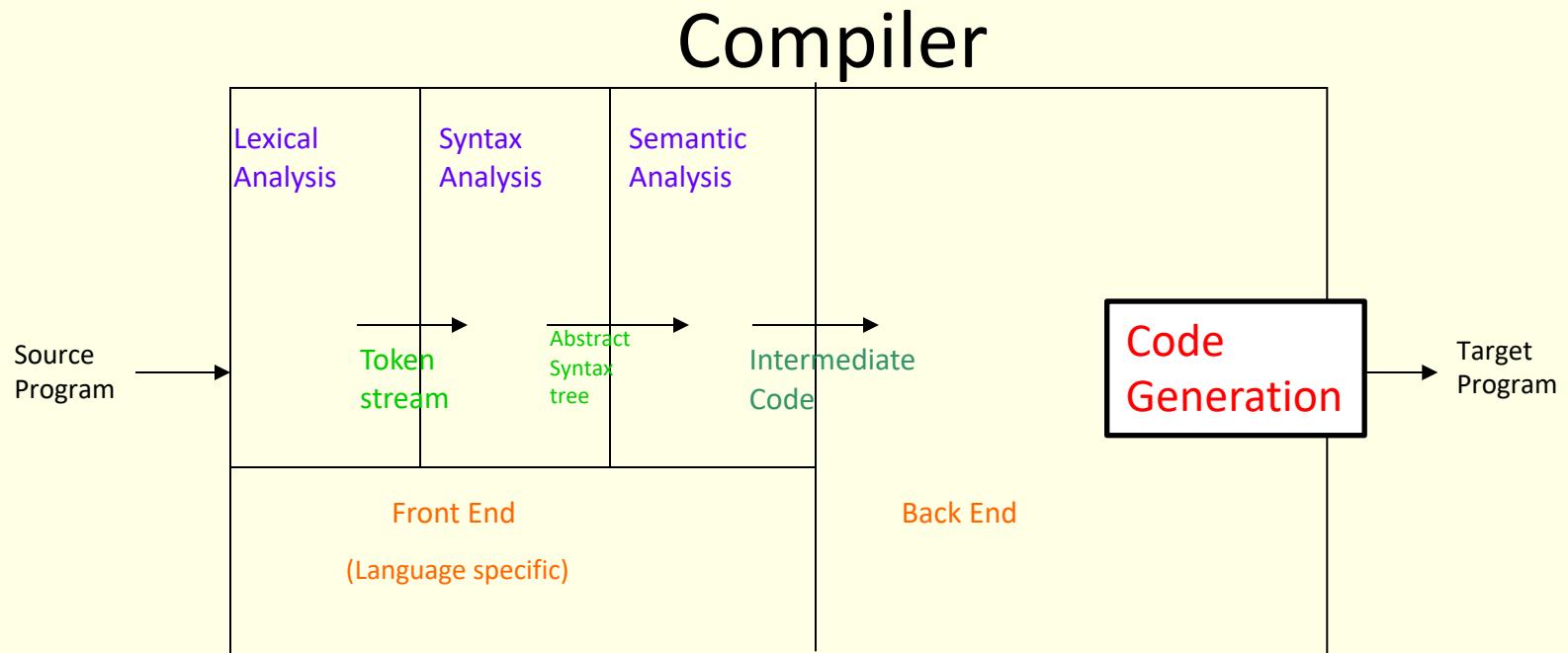
$L \rightarrow S$   $L.\text{nextlist} = S.\text{nextlist}$

$N \rightarrow \in$   $N.\text{nextlist} = \text{makelist(nextquad)}$   
 $\text{emit(goto ---)}$

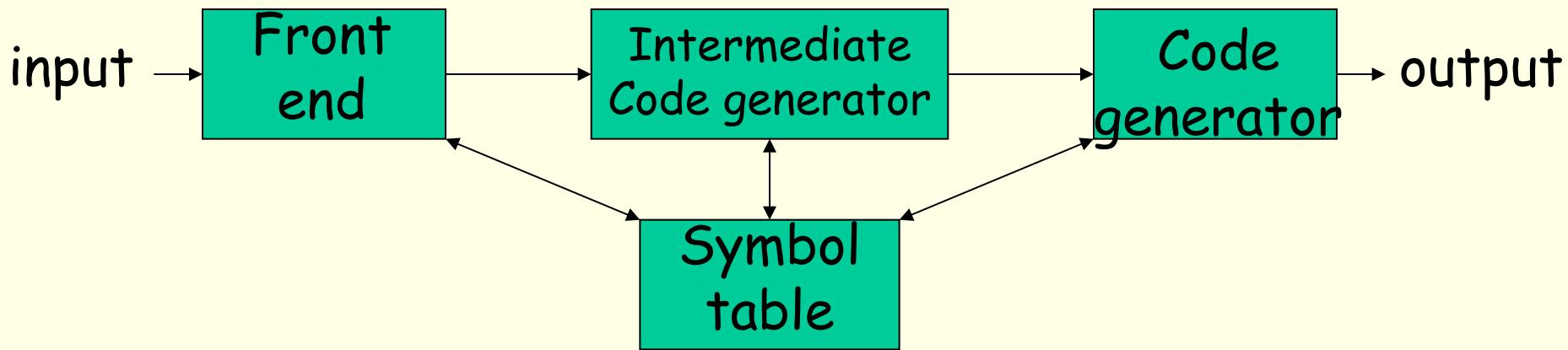
$M \rightarrow \in$   $M.\text{quad} = \text{nextquad}$

# Principles of Compiler Design

## Code Generation



# Code generation and Instruction Selection



## Requirements

- output code must be correct
- output code must be of high quality
- code generator should run efficiently

# Design of code generator: Issues

- Input: Intermediate representation with symbol table
  - assume that input has been validated by the front end
- Target programs :
  - absolute machine language
    - fast for small programs
  - relocatable machine code
    - requires linker and loader
  - assembly code
    - requires assembler, linker, and loader

# More Issues...

- Instruction selection
  - Uniformity
  - Completeness
  - Instruction speed, power consumption
- Register allocation
  - Instructions with register operands are faster
  - store long life time and counters in registers
  - temporary locations
  - Even odd register pairs
- Evaluation order

# Instruction Selection

- straight forward code if efficiency is not an issue

$a = b + c$	Mov b, R <sub>0</sub>
$d = a + e$	Add c, R <sub>0</sub>
	Mov R <sub>0</sub> , a
	Mov a, R <sub>0</sub>
	Add e, R <sub>0</sub>
	Mov R <sub>0</sub> , d

$a = a + 1$	Mov a, R <sub>0</sub>	Inc a
	Add #1, R <sub>0</sub>	
	Mov R <sub>0</sub> , a	

# Example Target Machine

- Byte addressable with 4 bytes per word
- $n$  registers  $R_0, R_1, \dots, R_{n-1}$
- Two address instructions of the form  
**opcode source, destination**
- Usual opcodes like move, add, sub etc.
- Addressing modes

MODE	FORM	ADDRESS
Absolute	M	M
register	R	R
index	c(R)	c+content(R)
indirect register	*R	content(R)
indirect index	*c(R)	content(c+content(R))
literal	#c	c

# Flow Graph

- Graph representation of three address code
- Useful for understanding code generation (and for optimization)
- Nodes represent computation
- Edges represent flow of control

# Basic blocks

- (maximum) sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end

Algorithm to identify basic blocks

- determine leader
  - first statement is a leader
  - any target of a goto statement is a leader
  - any statement that follows a goto statement is a leader
- for each leader its basic block consists of the leader and all statements up to next leader

# Flow graphs

- add control flow information to basic blocks
- nodes are the basic blocks
- there is a directed edge from  $B_1$  to  $B_2$  if  $B_2$  can follow  $B_1$  in some execution sequence
  - there is a jump from the last statement of  $B_1$  to the first statement of  $B_2$
  - $B_2$  follows  $B_1$  in natural order of execution
- initial node: block with first statement as leader

# Next use information

- for register and temporary allocation
- remove variables from registers if not used
- statement  $X = Y \text{ op } Z$   
defines X and uses Y and Z
- scan each basic blocks backwards
- assume all temporaries are dead on exit and all user variables are live on exit

# Computing next use information

Suppose we are scanning

$i : X := Y \text{ op } Z$

in backward scan

1. attach to statement  $i$ , information in symbol table about  $X, Y, Z$
2. set  $X$  to “not live” and “no next use” in symbol table
3. set  $Y$  and  $Z$  to be “live” and next use as  $i$  in symbol table

# Example

1:  $t_1 = a * a$

2:  $t_2 = a * b$

3:  $t_3 = 2 * t_2$

4:  $t_4 = t_1 + t_3$

5:  $t_5 = b * b$

6:  $t_6 = t_4 + t_5$

7:  $X = t_6$

## STATEMENT

1:  $t_1 = a * a$  ←  
2:  $t_2 = a * b$  ←  
3:  $t_3 = 2 * t_2$  ←  
4:  $t_4 = t_1 + t_3$  ←  
5:  $t_5 = b * b$  ←  
6:  $t_6 = t_4 + t_5$  ←  
7:  $X = t_6$  ←

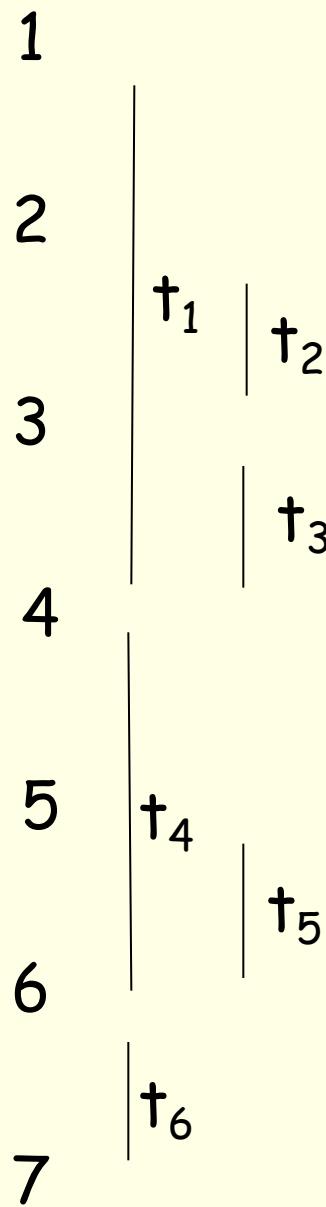
7: no temporary is live  
6:  $t_6$ :use(7),  $t_4$   $t_5$  not live  
5:  $t_5$ :use(6)  
4:  $t_4$ :use(6),  $t_1$   $t_3$  not live  
3:  $t_3$ :use(4),  $t_2$  not live  
2:  $t_2$ :use(3)  
1:  $t_1$ :use(4)

## Example

### Symbol Table

$t_1$	dead	Use in 4
$t_2$	dead	Use in 3
$t_3$	dead	Use in 4
$t_4$	dead	Use in 6
$t_5$	dead	Use in 6
$t_6$	dead	Use in 7

# Example ...



STATEMENT
1: $t_1 = a * a$
2: $t_2 = a * b$
3: $t_3 = 2 * t_2$
4: $t_4 = t_1 + t_3$
5: $t_5 = b * b$
6: $t_6 = t_4 + t_5$
7: $X = t_6$

- 1:  $t_1 = a * a$
- 2:  $t_2 = a * b$
- 3:  $t_3 = 2 * t_2$
- 4:  $t_4 = t_1 + t_3$
- 5:  $t_5 = b * b$
- 6:  $t_6 = t_4 + t_5$
- 7:  $X = t_6$

# Code Generator

- consider each statement
- remember if operands are in registers
- **Register descriptor**
  - Keep track of what is currently in each register.
  - Initially all the registers are empty
- **Address descriptor**
  - Keep track of location where current value of the name can be found at runtime
  - The location might be a register, stack, memory address or a set of those

# Code Generation Algorithm

**for each  $X = Y \text{ op } Z$  do**

- invoke a function **getreg** to determine location L where X must be stored. Usually L is a register.
- Consult address descriptor of Y to determine Y'. Prefer a register for Y'. If value of Y not already in L generate

**Mov Y', L**

# Code Generation Algorithm

- Generate  
op Z', L  
Again prefer a register for Z. Update address descriptor of X to indicate X is in L.
- If L is a register, update its descriptor to indicate that it contains X and remove X from all other register descriptors.
- If current value of Y and/or Z have no next use and are dead on exit from block and are in registers, change register descriptor to indicate that they no longer contain Y and/or Z.

# Function getreg

1. If Y is in register (that holds no other values) and Y is not live and has no next use after

$$X = Y \text{ op } Z$$

then return register of Y for L.

2. Failing (1) return an empty register
3. Failing (2) if X has a next use in the block or op requires register then get a register R, store its content into M (by Mov R, M) and use it.
4. else select memory location X as L

# Example

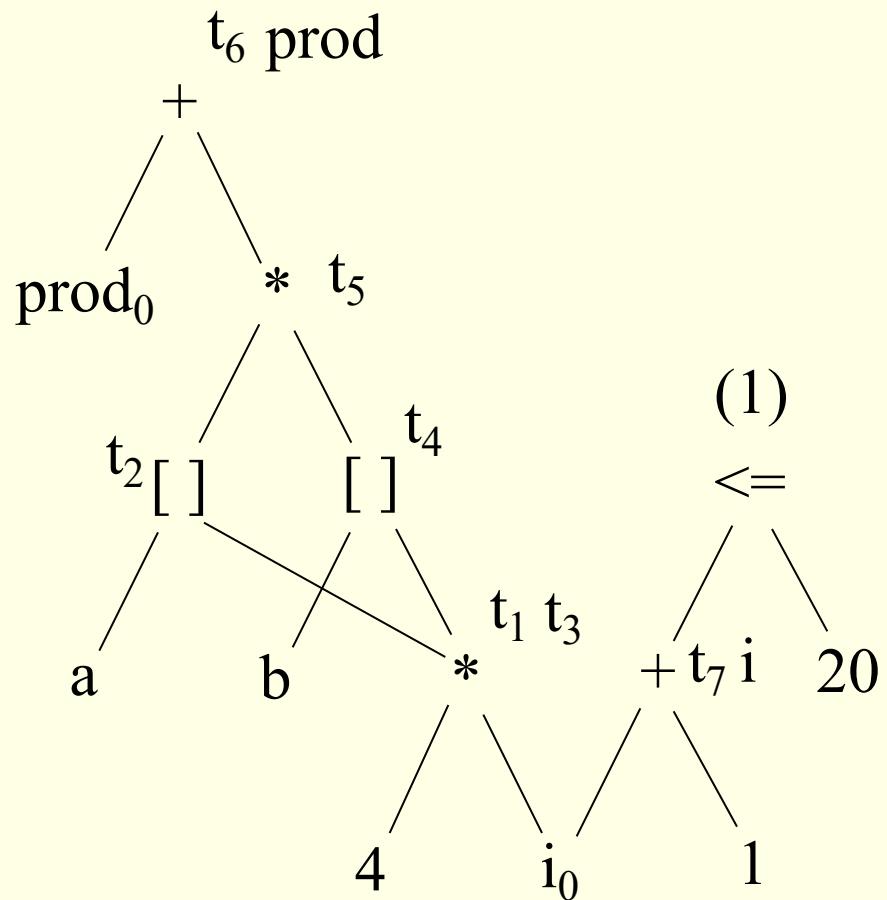
Stmt	code	reg desc	addr desc
$t_1=a-b$	$\text{mov } a, R_0$		
	$\text{sub } b, R_0$	$R_0$ contains $t_1$	$t_1$ in $R_0$
$t_2=a-c$	$\text{mov } a, R_1$	$R_0$ contains $t_1$	$t_1$ in $R_0$
	$\text{sub } c, R_1$	$R_1$ contains $t_2$	$t_2$ in $R_1$
$t_3=t_1+t_2$	$\text{add } R_1, R_0$	$R_0$ contains $t_3$	$t_3$ in $R_0$
		$R_1$ contains $t_2$	$t_2$ in $R_1$
$d=t_3+t_2$	$\text{add } R_1, R_0$	$R_0$ contains $d$	$d$ in $R_0$
	$\text{mov } R_0, d$		$d$ in $R_0$ and memory

# DAG representation of basic blocks

- useful data structures for implementing transformations on basic blocks
- gives a picture of how value computed by a statement is used in subsequent statements
- good way of determining common sub-expressions
- A dag for a basic block has following labels on the nodes
  - leaves are labeled by unique identifiers, either variable names or constants
  - interior nodes are labeled by an operator symbol
  - nodes are also optionally given a sequence of identifiers for labels

# DAG representation: example

1.  $t_1 := 4 * i$
2.  $t_2 := a[t_1]$
3.  $t_3 := 4 * i$
4.  $t_4 := b[t_3]$
5.  $t_5 := t_2 * t_4$
6.  $t_6 := \text{prod} + t_5$
7.  $\text{prod} := t_6$
8.  $t_7 := i + 1$
9.  $i := t_7$
10. if  $i \leq 20$  goto (1)



# Code Generation from DAG

$S_1 = 4 * i$

$S_2 = \text{addr}(A)-4$

$S_3 = S_2[S_1]$

$S_4 = 4 * i$

$S_5 = \text{addr}(B)-4$

$S_6 = S_5[S_4]$

$S_7 = S_3 * S_6$

$S_8 = \text{prod} + S_7$

$\text{prod} = S_8$

$S_9 = I+1$

$I = S_9$

If  $I \leq 20$  goto (1)

$S_1 = 4 * i$

$S_2 = \text{addr}(A)-4$

$S_3 = S_2[S_1]$

$S_5 = \text{addr}(B)-4$

$S_6 = S_5[S_4]$

$S_7 = S_3 * S_6$

$\text{prod} = \text{prod} + S_7$

$I = I + 1$

If  $I \leq 20$  goto (1)

# Rearranging order of the code

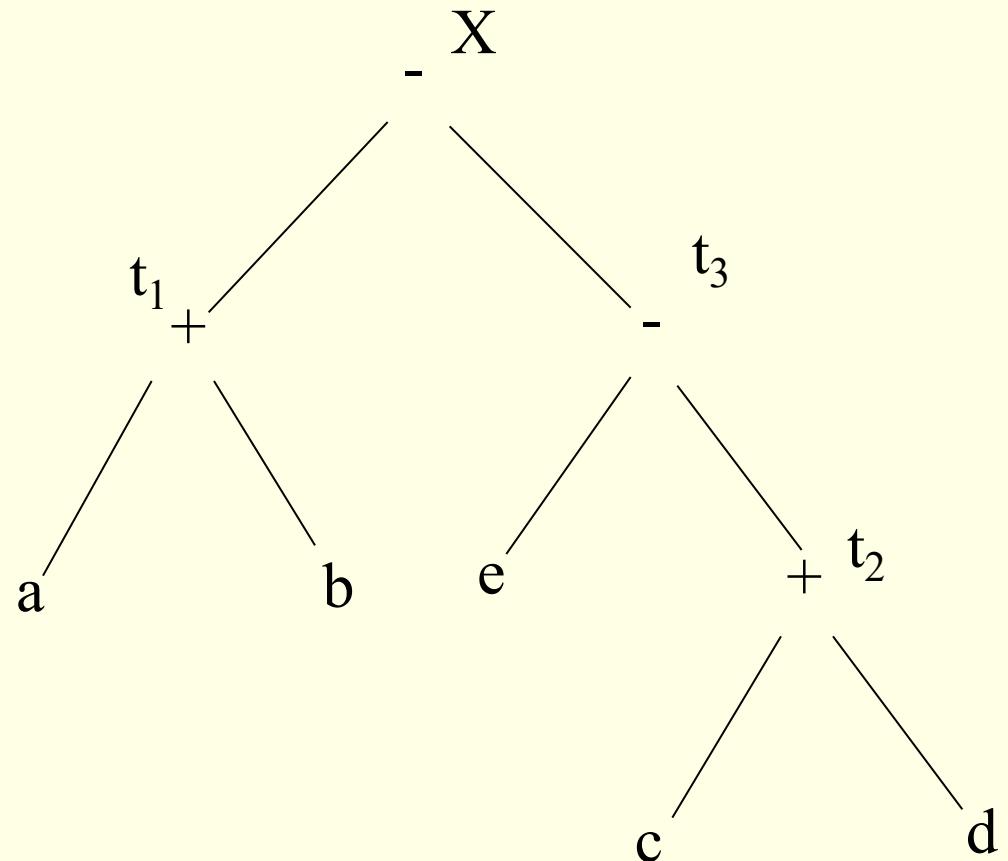
- Consider following basic block

$$t_1 = a + b$$

$$t_2 = c + d$$

$$t_3 = e - t_2$$

$$X = t_1 - t_3$$



and its DAG

# Rearranging order ...

Three address code for the DAG (assuming only two registers are available)

MOV a, R<sub>0</sub>

ADD b, R<sub>0</sub>

MOV c, R<sub>1</sub>

ADD d, R<sub>1</sub>

MOV R<sub>0</sub>, t<sub>1</sub>

MOV e, R<sub>0</sub>

SUB R<sub>1</sub>, R<sub>0</sub>

MOV t<sub>1</sub>, R<sub>1</sub>

SUB R<sub>0</sub>, R<sub>1</sub>

MOV R<sub>1</sub>, X

Register spilling

Register reloading

Rearranging the code as

$$t_2 = c + d$$

$$t_3 = e - t_2$$

$$t_1 = a + b$$

$$X = t_1 - t_3$$

gives

MOV c, R<sub>0</sub>

ADD d, R<sub>0</sub>

MOV e, R<sub>1</sub>

SUB R<sub>0</sub>, R<sub>1</sub>

MOV a, R<sub>0</sub>

ADD b, R<sub>0</sub>

SUB R<sub>1</sub>, R<sub>0</sub>

MOV R<sub>1</sub>, X 24



# Advanced Compiler Optimizations

## Code Optimization

Amey Karkare  
Dept of Computer Science and Engg  
IIT Kanpur

karkare@cse.iitk.ac.in

# Recap

- Optimizations
  - To improve efficiency of generated executable (time, space, resources ...)
  - Maintain semantic equivalence
- Two levels
  - Machine Independent
  - Machine Dependent

# Machine Independent Optimizations

- Scope of optimizations
    - Local
    - Global
    - Interprocedural
- 
- The diagram shows a horizontal brace with two vertical lines extending downwards from it, grouping the terms 'Local' and 'Global'. To the right of this group, the term 'Intraprocedural' is written, indicating that both local and global optimizations fall under this category.

# Local Optimizations

- Restricted to a basic block
- Simplifies the analysis
- Not all optimizations can be applied locally
  - E.g. Loop optimizations
- Gains are also limited
- Simplify global/interprocedural optimizations

# Global Optimizations

- Typically restricted within a procedure/function
  - Could be restricted to a smaller scope, e.g. a loop
- Most compiler implement up to global optimizations
  - Well founded theory
  - Practical gains

# Interprocedural Optimizations

- Spans multiple procedures, files
  - In some cases multiple languages!
- Not as popular as global optimizations
  - No single theory applicable to multiple scenarios
  - Time consuming

# *A Catalogue of Code Optimizations*

# Compile-time Evaluation

- Move run-time actions to compile-time
- Constant Folding:  
Volume =  $4/3 * \text{PI} * r * r * r;$ 
  - Compute  $4/3 * \text{PI}$  at compile time
  - Applied very frequently for linearizing indices of multidimensional arrays
  - When can we apply it?

# Compile-time Evaluation

- Constant Propagation
  - Replace a variable by its “constant” value

```
i = 5;  
...  
j = i * 4;  
...
```

Replaced by

```
i = 5;  
...  
j = 5 * 4;  
...
```

- May result in application of constant folding
- When can we apply it?

# Common Subexpression Elimination

- Reuse a computation if already "available"

```
x = u+v;  
...  
y = u+v+w;  
...
```

Replaced by

```
t0 = u+v;  
x = t0;  
...  
y = t0+w;  
...
```

- When can we do it?

# Copy Propagation

- Replace a variable by another
  - If they are guaranteed to have same value

```
i = k;
```

```
...
```

```
j = i*4;
```

```
...
```

Replaced by

```
i = k;
```

```
...
```

```
j = k*4;
```

- May result in dead code, common subexpr, ...
- When can we apply it?

# Code Movement

- Move the code in a program
- Benefits:
  - Code size reduction
  - Reduction in the frequency of execution
- Allowed only if the meaning of the program does not change.
  - May result in dead code, common subexpr, ...
  - When can we apply it?

# Code Movement

- Code size reduction

```
if (a < b)
    u = x*y;
else
    v = x*y;
```

Replaced by

```
t1 = x*y;
if (a < b)
    u = t1;
else
    v = t1;
```

# Code Movement

- Execution frequency reduction

```
if (a < b)
    u = ...;
else
    v = x*y;
w = x*y;
```

Replaced by

```
if (a < b) {
    t2 = x*y;
    u = ...;
} else {
    t2 = x*y;
    v = t2;
}
w = t2;
```

- Important: Safety of code motion!
- When can we do it?

# Loop Invariant Code Movement

- Execution frequency reduction

```
for (...) {  
    ...  
    u = a+b;  
    ...  
}
```

Replaced by

```
t3 = a+b;  
for (...) {  
    ...  
    u = t3;  
    ...  
}
```

- When can we do it?

# Other optimizations

- Dead code elimination
  - Remove unreachable, unused code.
  - Can we always do it?
- Strength reduction
  - Use of *low strength* operators in place of *high strength* operators.
    - $i*i$  instead of  $i^2$ ,  $\text{pow}(i,2)$
    - $i\ll 1$  instead of  $i*2$
  - Typically performed for integers only  
(Why?)

# Data Flow Analysis

- Class of techniques to derive information about flow of data
  - along program execution paths
- Used to answer questions such as:
  - whether two identical expressions evaluate to same value
    - used in common subexpression elimination
  - whether the result of an assignment is used later
    - used by dead code elimination

# Data Flow Abstraction

- Flow graph
  - Graph representation of paths that program may exercise during execution
  - Typically one graph per procedure
  - Graphs for separate procedure have to be combined/connected for interprocedural analysis
    - Later!
    - Single procedure, single flow graph for now.

# Data Flow Abstraction

- Basic Blocks (bb)
- Input state/Output state for Stmt
  - Program point before/after a stmt
  - Denoted  $\text{IN}[s]$  and  $\text{OUT}[s]$
  - Within a basic block:
    - Program point after a stmt is same as the program point before the next stmt

# Runtime Environment

- Relationship between names and data objects (of target machine)
- Allocation & de-allocation is managed by run time support package
- Each execution of a procedure is an activation of the procedure. If procedure is recursive, several activations may be alive at the same time.
  - If a and b are activations of two procedures then their lifetime is either non overlapping or nested
  - A procedure is recursive if an activation can begin before an earlier activation of the same procedure has ended

# Procedure

- A procedure definition is a declaration that associates an identifier with a statement (procedure body)
- When a procedure name appears in an executable statement, it is called at that point
- Formal parameters are the one that appear in declaration. Actual Parameters are the one that appear in when a procedure is called

# Activation tree

- Control flows sequentially
- Execution of a procedure starts at the beginning of body
- It returns control to place where procedure was called from
- A tree can be used, called an activation tree, to depict the way control enters and leaves activations
  - The root represents the activation of main program
  - Each node represents an activation of procedure
  - The node **a** is parent of **b** if control flows from **a** to **b**
  - The node **a** is to the left of node **b** if lifetime of **a** occurs before **b**

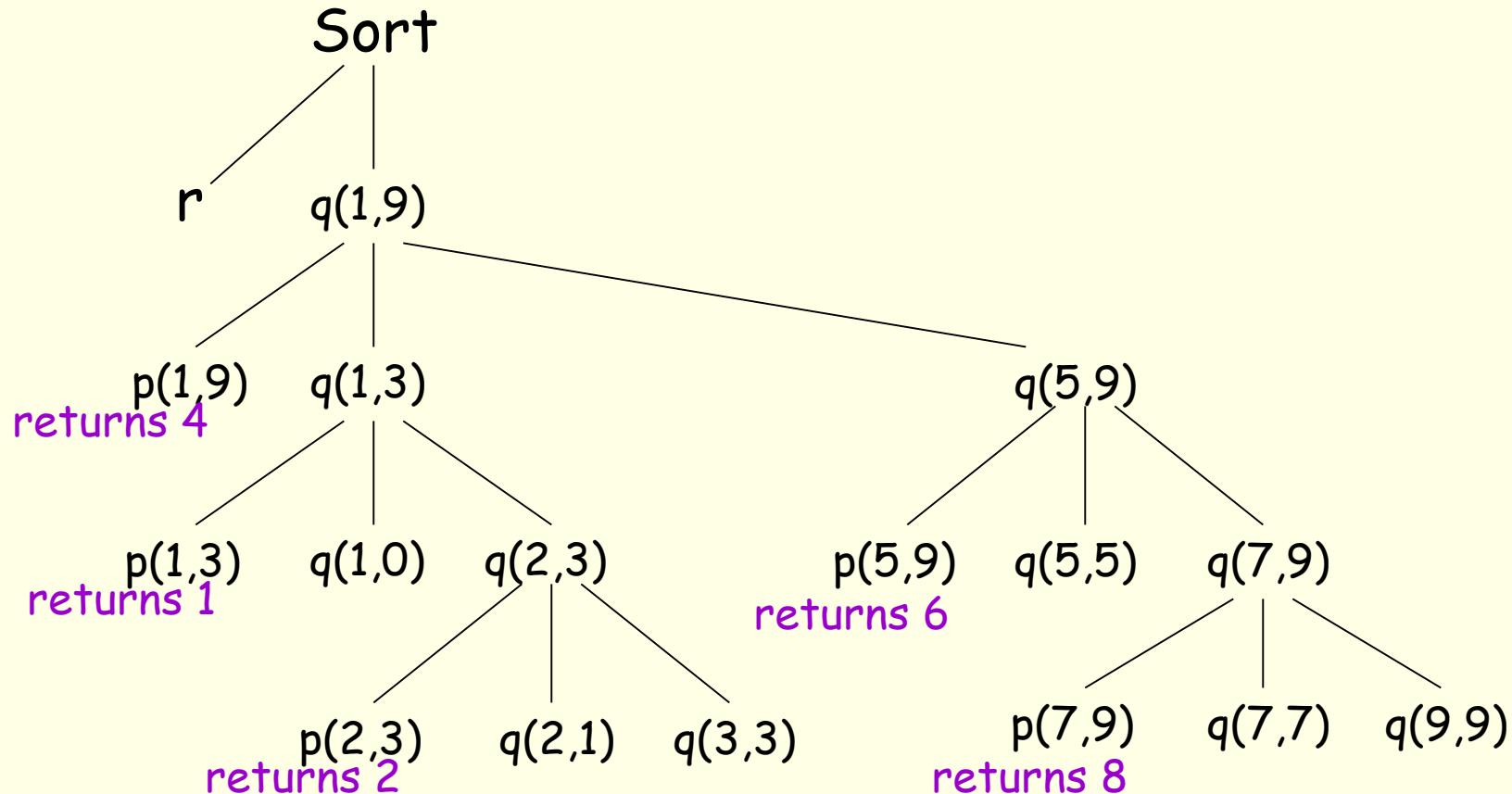
# Example

```
program sort;  
var a : array[0..10] of  
integer;
```

```
procedure readarray;  
var i :integer;  
:  
function partition (y, z  
:integer)  
:integer;  
var i, j ,x, v :integer;  
:
```

```
procedure quicksort (m, n  
:integer);  
var i :integer;  
:  
i:= partition (m,n);  
quicksort (m,i-1);  
quicksort(i+1, n);  
:  
begin{main}  
readarray;  
quicksort(1,9)  
end.
```

# Activation Tree



# Control stack

- Flow of control in program corresponds to depth first traversal of activation tree
- Use a stack called control stack to keep track of live procedure activations
- Push the node when activation begins and pop the node when activation ends
- When the node  $n$  is at the top of the stack the stack contains the nodes along the path from  $n$  to the root

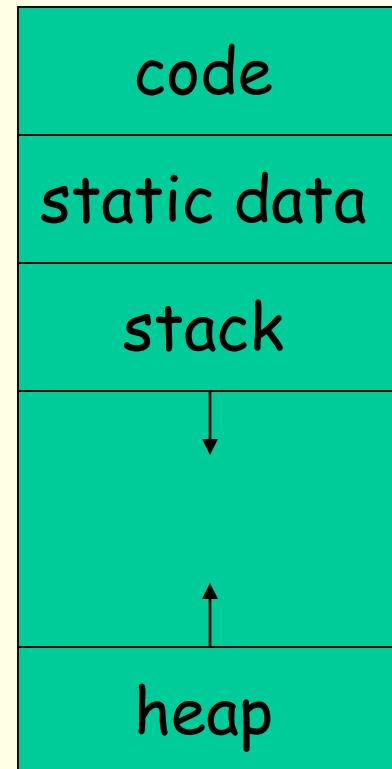
# Scope of declaration

- A declaration is a syntactic construct associating information with a name
  - Explicit declaration :Pascal (Algol class of languages)  
var i : integer
  - Implicit declaration: Fortran  
i is assumed to be integer
- There may be independent declarations of same name in a program.
- Scope rules determine which declaration applies to a name
- Name binding

name  $\xrightarrow{\text{environment}}$  storage  $\xrightarrow{\text{state}}$  value

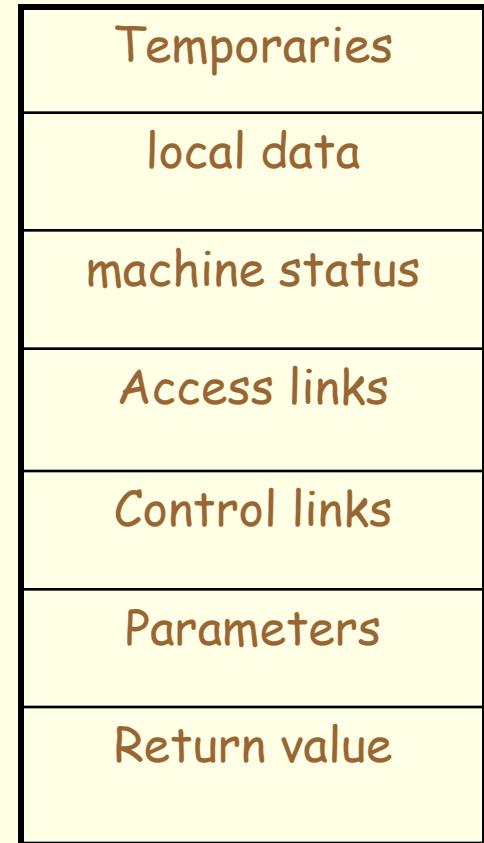
# Storage organization

- The runtime storage might be subdivided into
  - Target code
  - Data objects
  - Stack to keep track of procedure activation
  - Heap to keep all other information



# Activation Record

- **temporaries:** used in expression evaluation
- **local data:** field for local data
- **saved machine status:** holds info about machine status before procedure call
- **access link :** to access non local data
- **control link :** points to activation record of caller
- **actual parameters:** field to hold actual parameters
- **returned value:** field for holding value to be returned



# Activation Records: Examples

- Examples on the next few slides by Prof Amitabha Sanyal, IIT Bombay
- C/C++ programs with gcc extensions
- Compiled on x86\_64

# Example 1 – Vanilla Program in C

```
int a=1,b=2;  
main ()  
{  
    a = a + b;  
}
```

Global  
Memory

a	1
b	2

a and b have both been  
given absolute addresses

compilation of  $a = a + b$

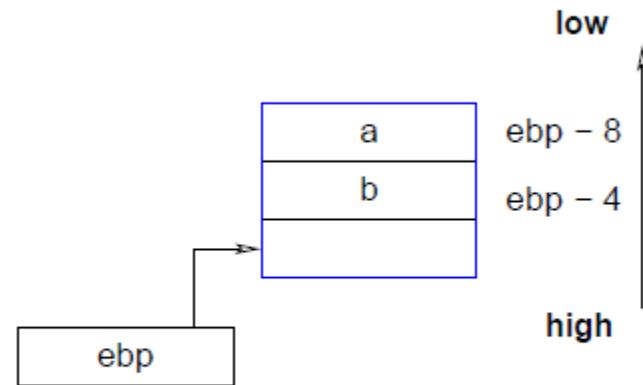
```
...  
movl a, %edx  
movl b, %eax  
addl %edx, %eax  
movl %eax, a  
...
```

# Example 2 – Function with Local Variables

```
void f()
{
    int a, b;
    a = a + b;
}
```

... compilation of  $a = a + b$

```
...
movl -4(%ebp), %eax
addl %eax, -8(%ebp)
...
```



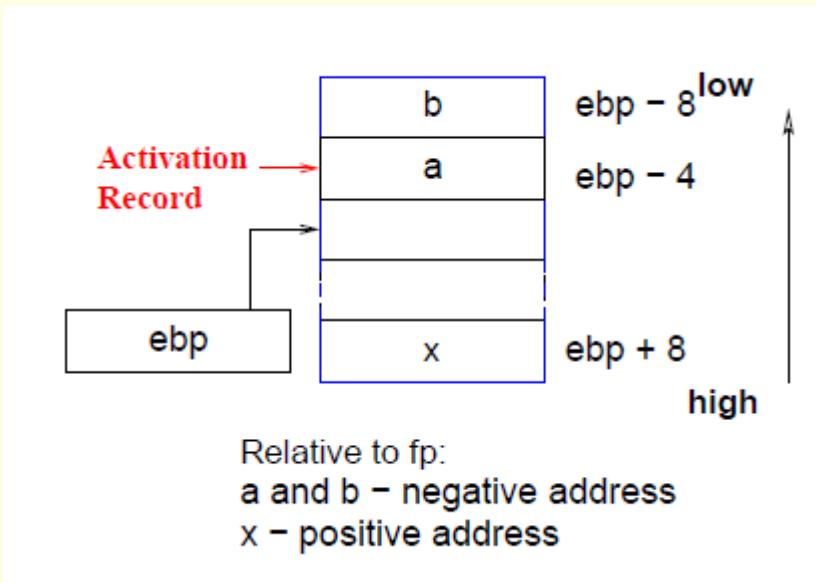
a and b have been given relative address on stack

# Example 3 – Function with Parameters

```
void f(int x)
{
    int a, b;
    a = x + b;
}
...
```

## Compilation of $a = x + b$

```
...
movl -8(%ebp), %eax
movl 8(%ebp), %edx
addl %edx, %eax
movl %eax, -4(%ebp)
...
```

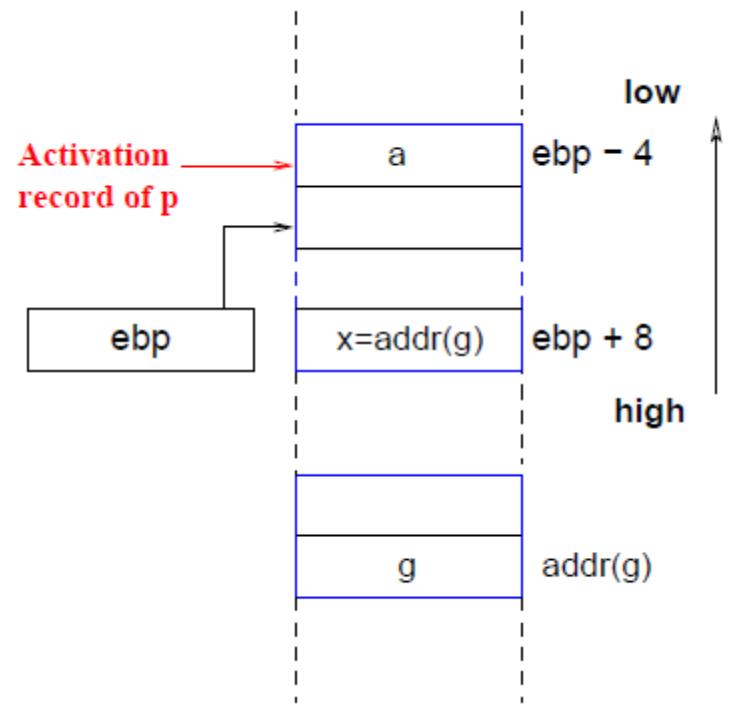


# Example 4 – Reference Parameters

```
int g;
void p(int& x)
{
    int a;
    a = x + 1;
}
main()
{
    p(g);
}
```

... compilation of  $a := x + 1$

```
movl 8(%ebp), %eax
movl (%eax), %eax
addl $1, %eax
movl %eax, -4(%ebp)
```

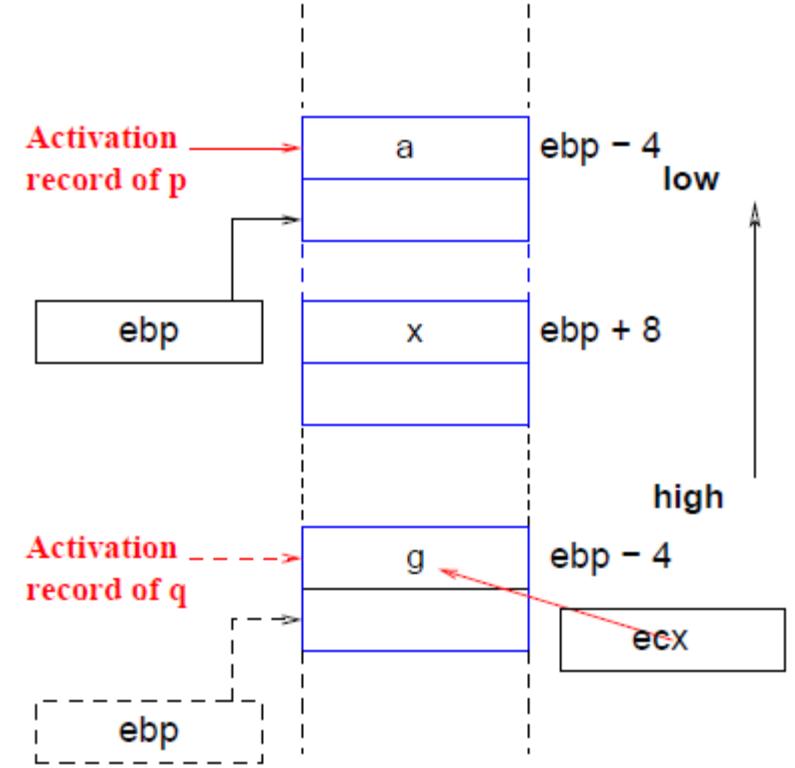


# Example 5 – Global Variables

```
void q()
{
    int g;
    void p(int x)
    {
        int a;
        a = x + g;
    };
    p(1);
}
```

**Compilation of  $a = x + g$**

```
...
movl %ecx, %eax ;static link
movl (%eax), %edx
movl 8(%ebp), %eax
addl %edx, %eax
movl %eax, -4(%ebp)
```



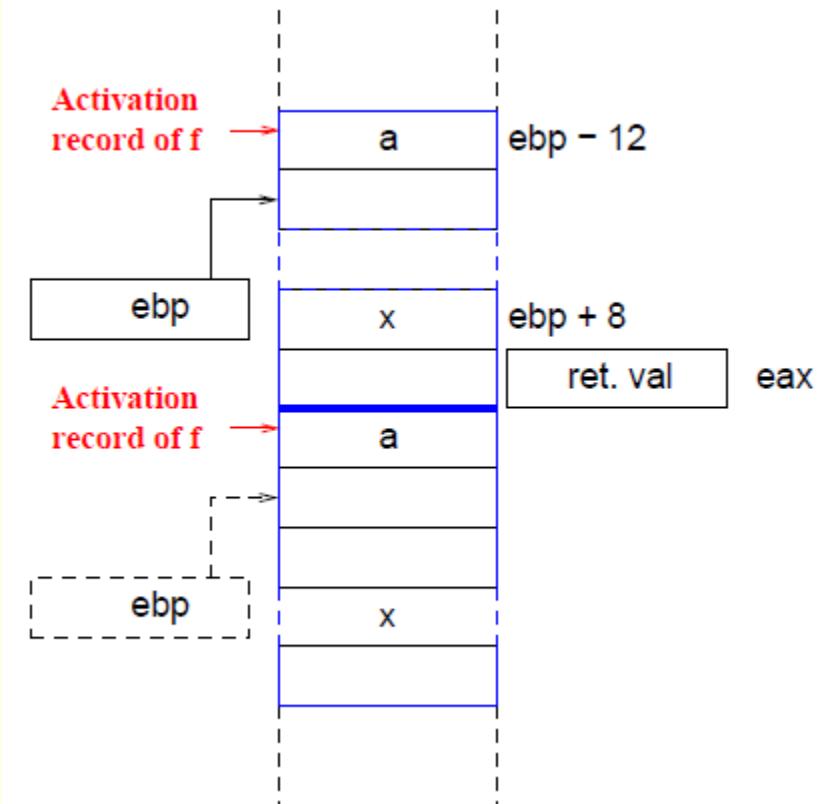
# Example 6 – Recursive Functions

```
int f(int x)
{
    int a;
    if (x==0) return 1;
    {
        a = f(x-1);
        return(x * a);
    }
}
```

... compilation of  $a = f(x-1);$   
 $\quad \quad \quad \text{return}(x * a)$

...

```
movl %eax, -12(%ebp)
movl 8(%ebp), %eax
imull -12(%ebp), %eax
```

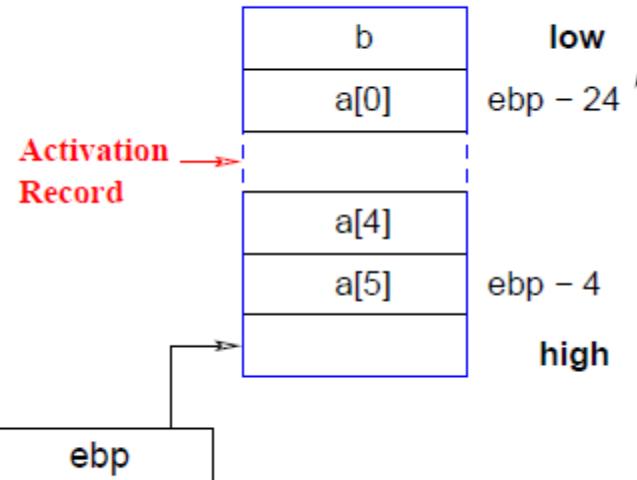


# Example 7 – Array Access

```
void p()
{
    int a[6], b;
    b = a[5];
}
```

... compilation of  $b = a[5]$

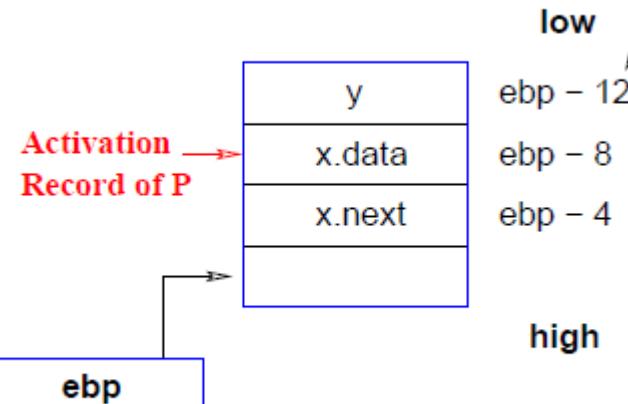
...  
movl -4(%ebp), %eax  
movl %eax, -28(%ebp)  
...



# Example 8 – Records and Pointers

```
typedef struct rec
{
    int data;
    struct rec* next;

} rec;
void p ()
{
    rec x; rec *y;
    x.next = y;
}
```



...compilation of `x.data = 5;`  
`x.next = y;`

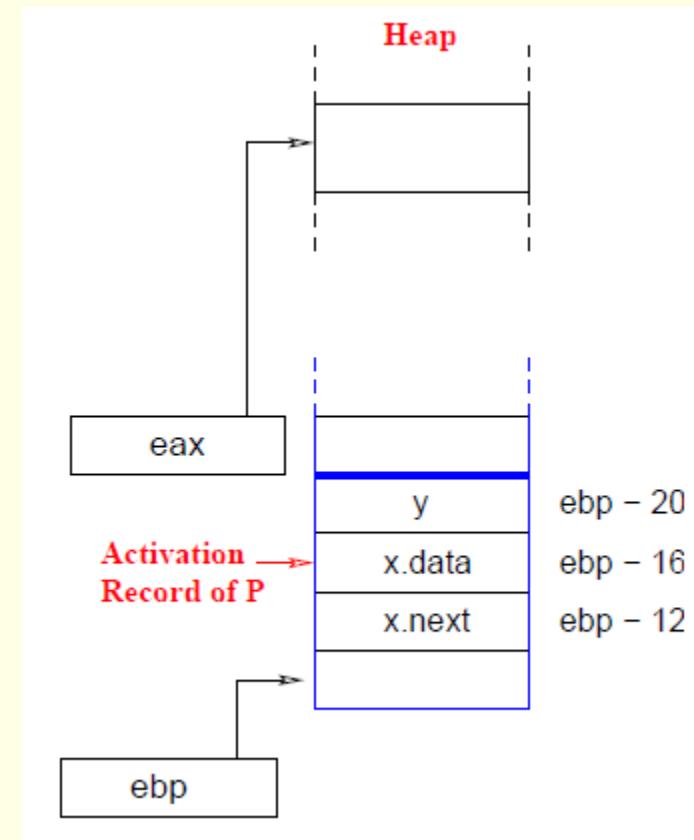
...  
movl -12(%ebp), %eax  
movl %eax, -4(%ebp)  
...

# Example 9 – Dynamically Created Data

```
typedef struct rec
{
    int data;
    struct rec* next;
} rec;
void p ()
{
    rec x; rec *y;
    y = malloc(4); x.next = y;
}
```

Compilation of `y = malloc...; x.next = y;`

```
call malloc
movl %eax, -20(%ebp)
movl -20(%ebp), %eax
movl %eax, -12(%ebp)
```



# Issues to be addressed

- Can procedures be recursive?
- What happens to locals when procedures return from an activation?
- Can procedure refer to non local names?
- How to pass parameters?
- Can procedure be parameter?
- Can procedure be returned?
- Can storage be dynamically allocated?
- Can storage be de-allocated?

# Layout of local data

- Assume byte is the smallest unit
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type
- Memory allocation is done as the declarations are processed
  - Keep a count of memory locations allocated for previous declarations
  - From the count *relative* address of the storage for a local can be determined
  - As an *offset* from some fixed position

# Layout of local data

- Data may have to be aligned (in a word) padding is done to have alignment.
- When space is important
  - Complier may pack the data so no padding is left
  - Additional instructions may be required to execute packed data
  - Tradeoff between space and execution time

# Storage Allocation Strategies

- Static allocation: lays out storage at compile time for all data objects
- Stack allocation: manages the runtime storage as a stack
- Heap allocation :allocates and de-allocates storage as needed at runtime from heap

# Static allocation

- Names are bound to storage as the program is compiled
- No runtime support is required
- Bindings do not change at run time
- On every invocation of procedure names are bound to the same storage
- Values of local names are retained across activations of a procedure

- Type of a name determines the amount of storage to be set aside
- Address of a storage consists of an offset from the end of an activation record
- Compiler decides location of each activation
- All the addresses can be filled at compile time
- Constraints
  - Size of all data objects must be known at compile time
  - Recursive procedures are not allowed
  - Data structures cannot be created dynamically

# Stack Allocation

Sort

Sort

Sort

readarray

Sort

readarray

Sort

readarray

qsort(1,9)

Sort

qsort(1,9)

Sort

readarray

qsort(1,9)

partition(1,9)    qsort(1,3)

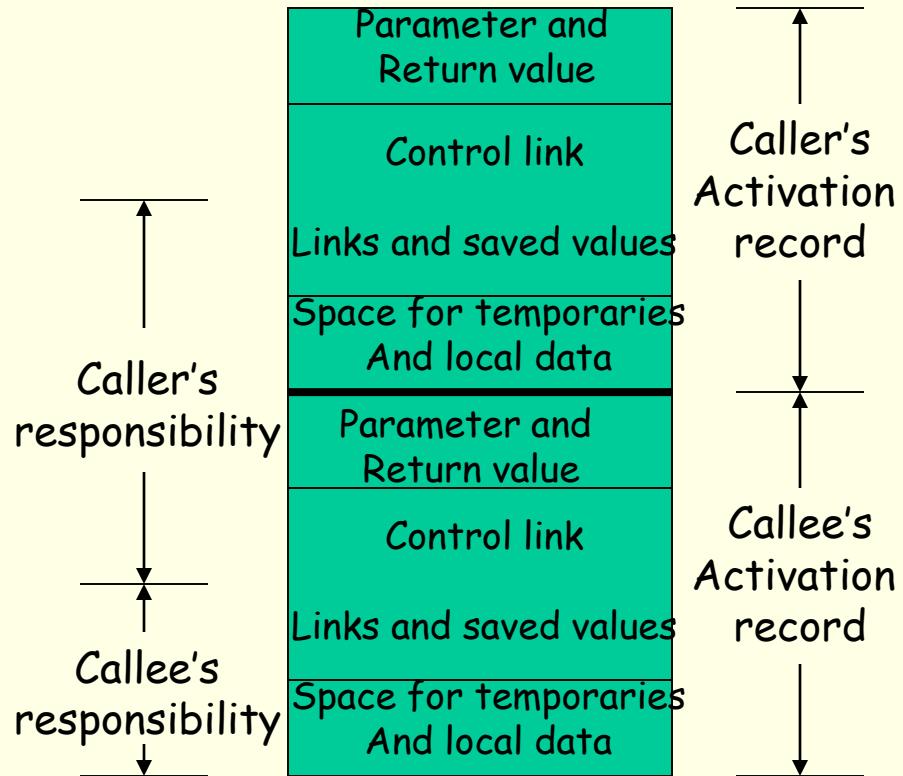
Sort

qsort(1,9)

qsort(1,3)

# Calling Sequence

- A call sequence allocates an activation record and enters information into its field
- A return sequence restores the state of the machine so that calling procedure can continue execution



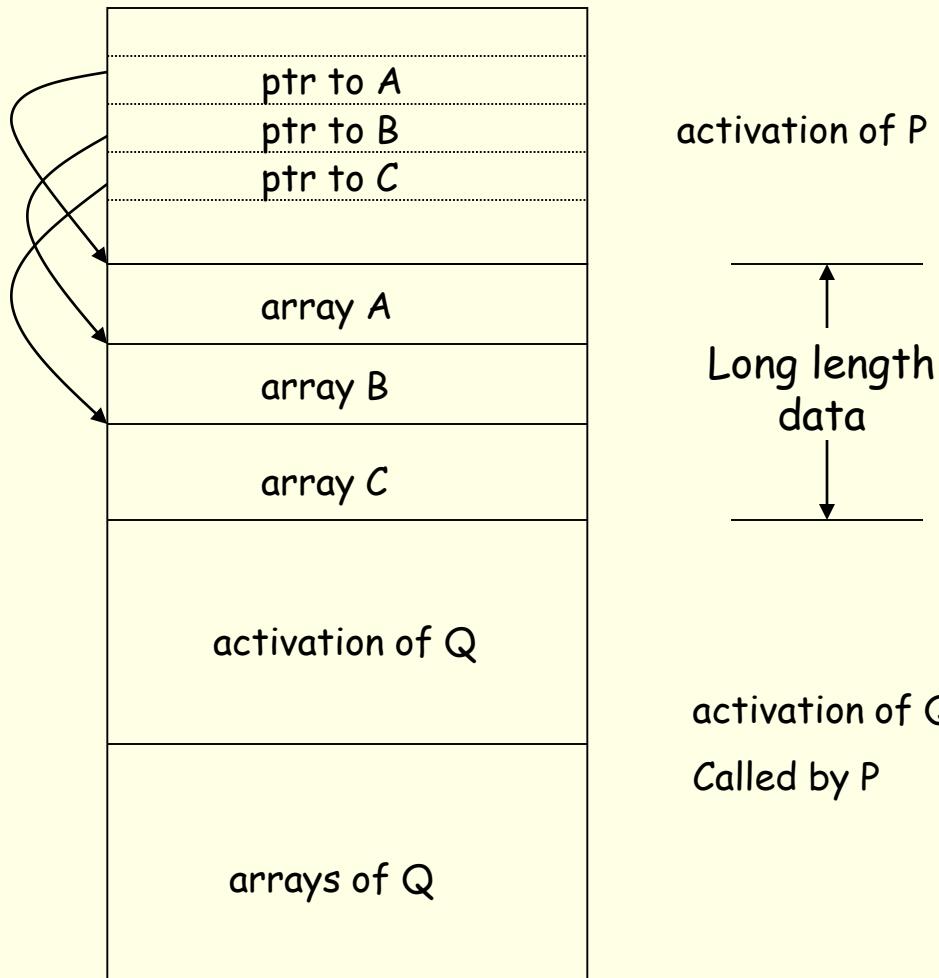
# Call Sequence

- Caller evaluates the actual parameters
- Caller stores return address and other values (control link) into callee's activation record
- Callee saves register values and other status information
- Callee initializes its local data and begins execution

# Return Sequence

- Callee places a return value next to activation record of caller
- Restores registers using information in status field
- Branch to return address
- Caller copies return value into its own activation record

# Long/Unknown Length Data



# Dangling references

Referring to locations which have been deallocated

```
main() {  
    int *p;  
    p = dangle(); /* dangling reference */  
}
```

```
int *dangle() {  
    int i=23;  
    return &i;  
}
```

# Heap Allocation

- Stack allocation cannot be used if:
  - The values of the local variables must be retained when an activation ends
  - A called activation outlives the caller
- In such a case de-allocation of activation record cannot occur in last-in first-out fashion
- Heap allocation gives out pieces of contiguous storage for activation records

# Heap Allocation ...

- Pieces may be de-allocated in any order
- Over time the heap will consist of alternate areas that are free and in use
- Heap manager is supposed to make use of the free space
- For efficiency reasons it may be helpful to handle small activations as a special case

# Heap Allocation ...

- For each size of interest keep a linked list of free blocks of that size
- Fill a request of size  $s$  with block of size  $s'$  where  $s'$  is the smallest size greater than or equal to  $s$ .
- When the block is deallocated, return it to the corresponding list

# Heap Allocation ...

- For large blocks of storage use heap manager
- For large amount of storage computation may take some time to use up memory
  - time taken by the manager may be negligible compared to the computation time

# Access to non-local names

- Scope rules determine the treatment of non-local names
- A common rule is *lexical scoping* or *static scoping* (most languages use lexical scoping)
  - Most closely nested declaration
- Alternative is *dynamic scoping*
  - Most closely nested activation

# Block

- Statement containing its own data declarations
- Blocks can be nested
  - also referred to as *block structured*
- Scope of the declaration is given by *most closely nested* rule
  - The scope of a declaration in block B includes B
  - If X is not declared in B then an occurrence of X in B is in the scope of declaration of X in B' such that
    - B' has a declaration of X
    - B' is most closely nested around B

# Example

```
main()
{
    int a=0
    int b=0
    {
        int b=1
        {
            int a=2
            print a, b
        }
        END of B2
    }
    BEGINNING of B3
    int b=3
    print a, b
}
END of B1
print a, b
}
END of B0
```

BEGINNING of B0 Scope B0, B1, B3

Scope B0

BEGINNING of B1 Scope B1, B2

BEGINNING of B2 Scope B2

END of B2

BEGINNING of B3 Scope B3

int b=3

print a, b

END of B3

END of B1

print a, b

END of B0

# Blocks ...

- Blocks are simpler to handle than procedures
- Blocks can be treated as parameter less procedures
- Either use stack for memory allocation
- OR allocate space for complete procedure body at one time

```
{ // a0
  { // b0
    { // b1
      { // a2
        }
      { // b3
        }
      }
    }
  }
```

a0
b0
b1
a2   b3

# Lexical scope without nested procedures

- A procedure definition cannot occur within another
- Therefore, all non local references are global and can be allocated at compile time
- Any name non-local to one procedure is non-local to all procedures
- In absence of nested procedures use stack allocation
  - Storage for non locals is allocated statically
    - Any other name must be local to the top of the stack
- Static allocation of non local has advantage:
  - Procedures can be passed/returned as parameters

# Scope with nested procedures

```
Program sort;  
var a: array[1..n] of integer;  
    x: integer;  
procedure readarray;  
var i: integer;  
begin  
  
end;  
procedure exchange(i,j:integer)  
begin  
  
end;
```

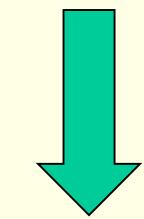
```
procedure quicksort(m,n:integer);  
var k,v : integer;  
  
function partition(y,z:integer): integer;  
var i,j: integer;  
begin  
  
end;  
begin  
  
end;  
begin  
  
end.
```

# Nesting Depth

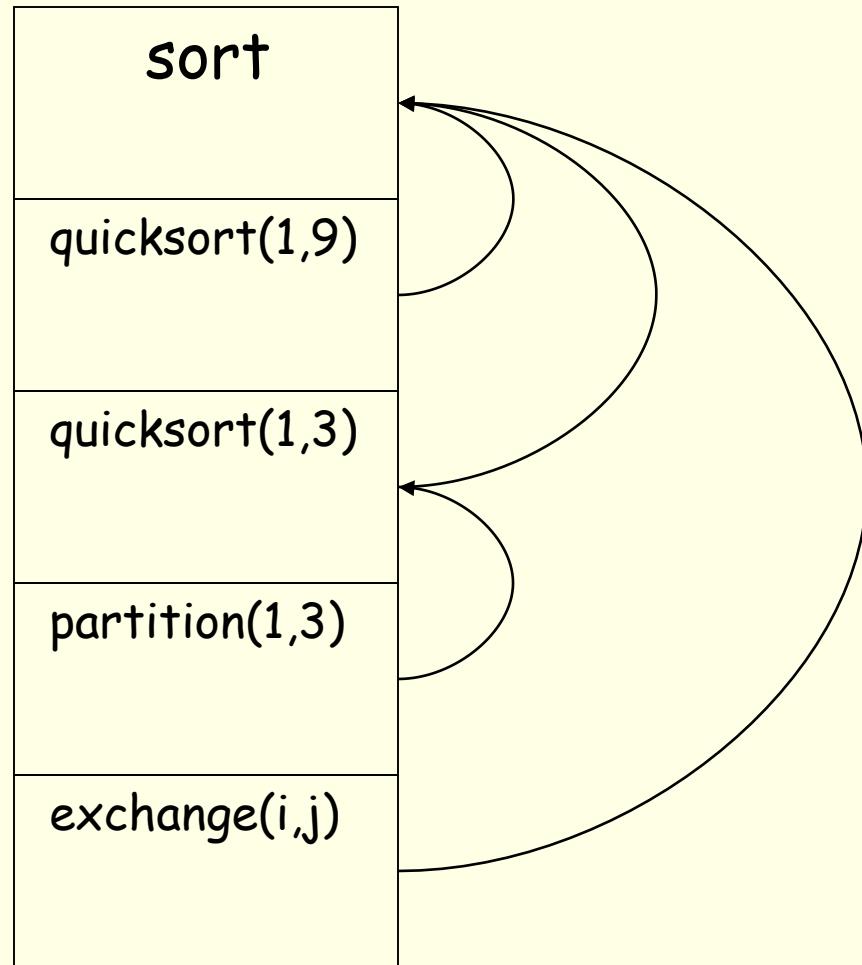
- Main procedure is at depth 1
- Add 1 to depth as we go from enclosing to enclosed procedure

## Access to non-local names

- Include a field ‘access link’ in the activation record
- If p is nested in q then access link of p points to the access link in most recent activation of q



Stack



# Access to non local names ...

- Suppose procedure p at depth np refers to a non-local  $a$  at depth na ( $na \leq np$ ), then storage for  $a$  can be found as
  - follow  $(np-na)$  access links from the record at the top of the stack
  - after following  $(np-na)$  links we reach procedure for which  $a$  is local
- Therefore, address of a non local  $a$  in p can be stored in symbol table as
  - $(np-na, \text{ offset of } a \text{ in record of activation having } a)$

# How to setup access links?

- Code to setup access links is part of the calling sequence.
- suppose procedure p at depth np calls procedure x at depth nx.
- The code for setting up access links depends upon *whether or not* the called procedure is nested within the caller.

# How to setup access links?

$$np < nx$$

- Called procedure x is nested more deeply than p.
- Therefore, x must be declared in p.
- The access link in x must point to the access link of the activation record of the caller just below it in the stack

# How to setup access links?

$$np \geq nx$$

- From scoping rules enclosing procedure at the depth  $1, 2, \dots, nx-1$  must be same.
- Follow  $np-(nx-1)$  links from the caller.
- We reach the most recent activation of the procedure that statically encloses both  $p$  and  $x$  most closely.
- The access link reached is the one to which access link in  $x$  must point.
- $np-(nx-1)$  can be computed at compile time.

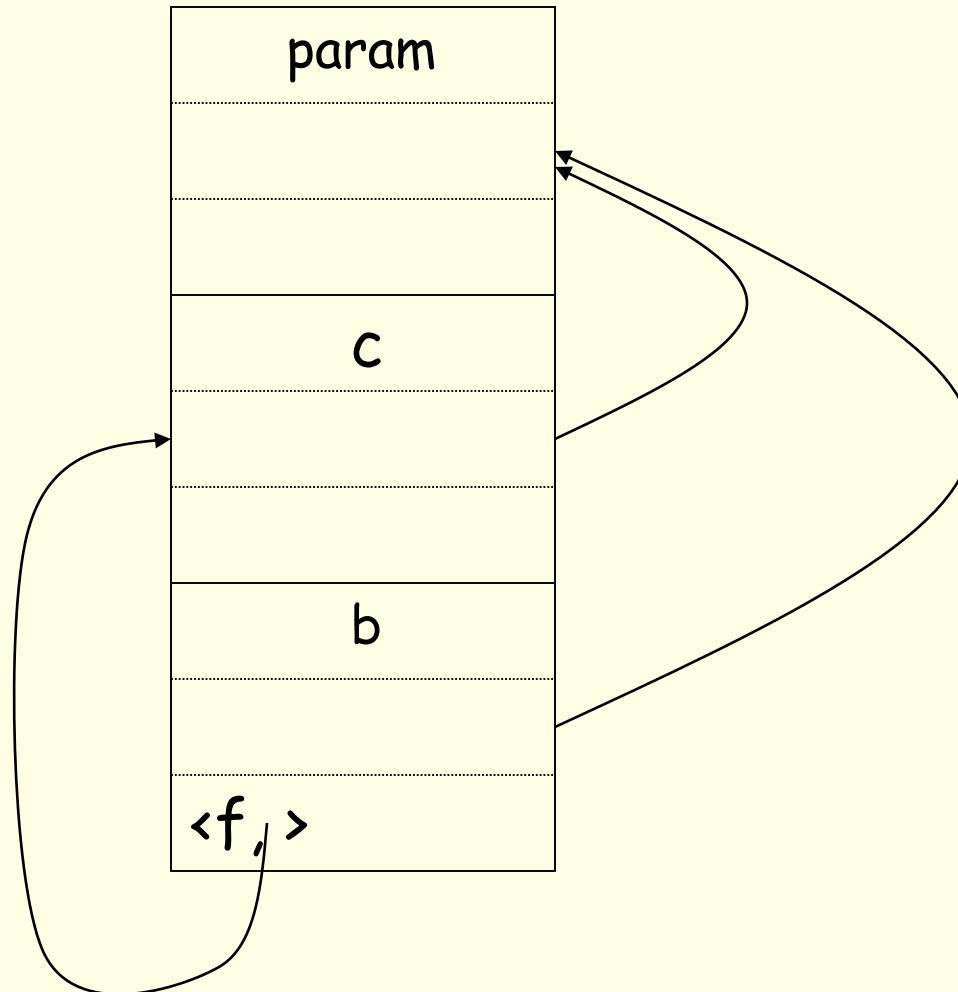
# Procedure Parameters

```
program param (input,output);
procedure b( function h(n:integer): integer);
begin
    print (h(2))
end;
procedure c;
var m: integer;
function f(n: integer): integer;
begin
    return m + n
end;
begin
    m :=0; b(f)
end;
begin
    c
end.
```

# Procedure Parameters ...

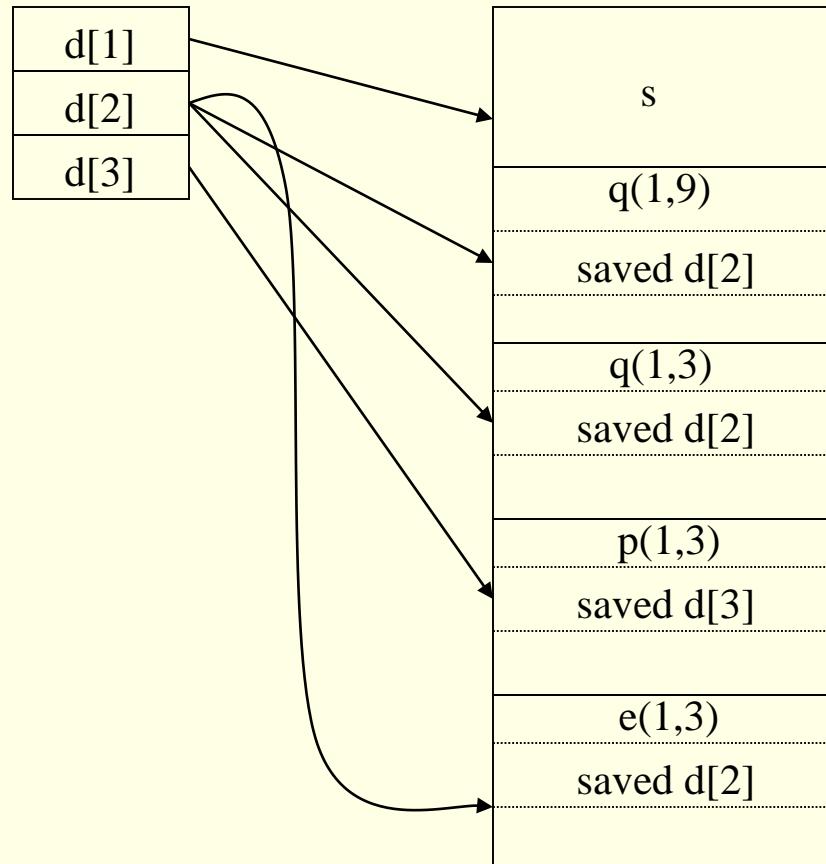
- Scope of m does not include procedure b
- within b, call h(2) activates f
- how is access link for activation of f is set up?
- a nested procedure must take its access link along with it
- when c passes f:
  - it determines access link for f as if it were calling f
  - this link is passed along with f to b
- When f is activated, this passed access link is used to set up the activation record of f

# Procedure Parameters ...



# Displays

- Faster access to non locals
- Uses an array of pointers to activation records
- Non locals at depth  $i$  are in the activation record pointed to by  $d[i]$



# Setting up Displays

- When a new activation record for a procedure at nesting depth  $i$  is set up:
- Save the value of  $d[i]$  in the new activation record
- Set  $d[i]$  to point to the new activation record
- Just before an activation ends,  $d[i]$  is reset to the saved value

# Justification for Displays

- Suppose procedure at depth  $j$  calls procedure at depth  $i$
- Case  $j < i$  then  $i = j + 1$ 
  - called procedure is nested within the caller
  - first  $j$  elements of display need not be changed
  - old value of  $d[i]$  is saved and  $d[i]$  set to the new activation record
- Case  $j \geq i$ 
  - enclosing procedure at depths  $1\dots i-1$  are same and are left un-disturbed
  - old value of  $d[i]$  is saved and  $d[i]$  points to the new record
  - display is correct as first  $i-1$  records are not disturbed

# Dynamic Scoping: Example

- Consider the following program

```
program dynamic (input, output);
```

```
  var r: real;
```

```
procedure show;
```

```
  begin write(r) end;
```

```
procedure small;
```

```
  var r: real;
```

```
  begin r := 0.125; show end;
```

```
begin
```

```
  r := 0.25;
```

```
  show; small; writeln;
```

```
  show; small; writeln;
```

```
end.
```

// writeln prints a newline character

## Example ...

- Output under lexical scoping

0.250      0.250

0.250      0.250

- Output under dynamic scoping

0.250      0.125

0.250      0.125

# Dynamic Scope

- Binding of non local names to storage do not change when new activation is set up
- A non local name  $x$  in the called activation refers to same storage that it did in the calling activation

# Implementing Dynamic Scope

- Deep Access
  - Dispense with access links
  - use control links to search into the stack
  - term deep access comes from the fact that search may go deep into the stack
- Shallow Access
  - hold current value of each name in static memory
  - when a new activation of p occurs a local name n in p takes over the storage for n
  - previous value of n is saved in the activation record of p

# Parameter Passing

- Call by value
  - actual parameters are evaluated and their r-values are passed to the called procedure
  - used in Pascal and C
  - formal is treated just like a local name
  - caller evaluates the actual parameters and places rvalue in the storage for formals
  - call has no effect on the activation record of caller

# Parameter Passing ...

- Call by reference (call by address)
  - the caller passes a pointer to each location of actual parameters
  - if actual parameter is a name then l-value is passed
  - if actual parameter is an expression then it is evaluated in a new location and the address of that location is passed

# Parameter Passing ...

- Copy restore (copy-in copy-out, call by value result)
  - actual parameters are evaluated, rvalues are passed by call by value, lvalues are determined before the call
  - when control returns, the current rvalues of the formals are copied into lvalues of the locals

# Parameter Passing ...

- Call by name (used in Algol)
  - names are copied
  - local names are different from names of calling procedure
  - Issue:

```
swap(x, y) {  
    temp = x  
    x = y  
    y = temp  
}
```

```
swap(i,a[i]):  
    temp = i  
    i = a[i]  
    a[i] = temp
```

# 3AC for Procedure Calls

$S \rightarrow \text{call id} ( \text{Elist} )$

$\text{Elist} \rightarrow \text{Elist}, E$

$\text{Elist} \rightarrow E$

- Calling sequence
  - allocate space for activation record
  - evaluate arguments
  - establish environment pointers
  - save status and return address
  - jump to the beginning of the procedure

# Procedure Calls ...

## Example

- parameters are passed by reference
- storage is statically allocated
- use param statement as place holder for the arguments
- called procedure is passed a pointer to the first parameter
- pointers to any argument can be obtained by using proper offsets

# Procedue Calls

- Generate three address code needed to evaluate arguments which are expressions
- Generate a list of param three address statements
- Store arguments in a list

$S \rightarrow \text{call id ( Elist )}$

for each item p on queue do emit('param' p)  
emit('call' id.place)

$Elist \rightarrow Elist , E$

append E.place to the end of queue

$Elist \rightarrow E$

initialize queue to contain E.place

# Procedure Calls

- Practice Exercise:

How to generate intermediate code for parameters passed by value? Passed by reference?

# Code Generation: Sethi Ullman Algorithm

Amey Karkare

karkare@cse.iitk.ac.in

March 28, 2019

# Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,
  - ▶ a store instruction, and

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,
  - ▶ a store instruction, and
  - ▶ binary operations involving either a register and a memory, or two registers.

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,
  - ▶ a store instruction, and
  - ▶ binary operations involving either a register and a memory, or two registers.
- ▶ Does not use algebraic properties of operators. If  $a * b$  has to be evaluated using  $r_1 \leftarrow r_1 * r_2$ , then  $a$  and  $b$  have to be necessarily loaded in  $r_1$  and  $r_2$  respectively.

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,
  - ▶ a store instruction, and
  - ▶ binary operations involving either a register and a memory, or two registers.
- ▶ Does not use algebraic properties of operators. If  $a * b$  has to be evaluated using  $r_1 \leftarrow r_1 * r_2$ , then  $a$  and  $b$  have to be necessarily loaded in  $r_1$  and  $r_2$  respectively.
- ▶ Extensions to take into account algebraic properties of operators.

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,
  - ▶ a store instruction, and
  - ▶ binary operations involving either a register and a memory, or two registers.
- ▶ Does not use algebraic properties of operators. If  $a * b$  has to be evaluated using  $r_1 \leftarrow r_1 * r_2$ , then  $a$  and  $b$  have to be necessarily loaded in  $r_1$  and  $r_2$  respectively.
- ▶ Extensions to take into account algebraic properties of operators.
- ▶ Generates optimal code – i.e. code with least number of instructions. There may be other notions of optimality.

## Sethi-Ullman Algorithm – Introduction

- ▶ Generates code for expression trees (not dags).
- ▶ Target machine model is simple. Has
  - ▶ a load instruction,
  - ▶ a store instruction, and
  - ▶ binary operations involving either a register and a memory, or two registers.
- ▶ Does not use algebraic properties of operators. If  $a * b$  has to be evaluated using  $r_1 \leftarrow r_1 * r_2$ , then  $a$  and  $b$  have to be necessarily loaded in  $r_1$  and  $r_2$  respectively.
- ▶ Extensions to take into account algebraic properties of operators.
- ▶ Generates optimal code – i.e. code with least number of instructions. There may be other notions of optimality.
- ▶ Complexity is linear in the size of the expression tree.  
Reasonably efficient.

# Expression Trees

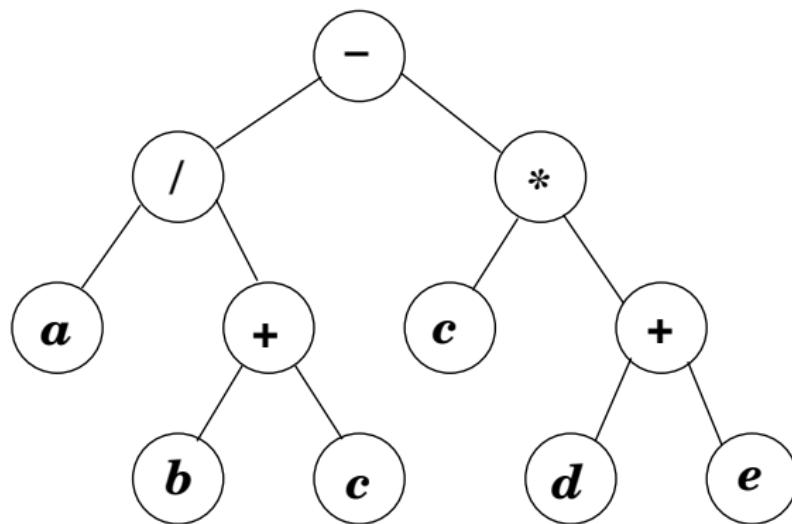
- ▶ Here is the expression  $a/(b + c) - c * (d + e)$  represented as a tree:

# Expression Trees

- ▶ Here is the expression  $a/(b + c) - c * (d + e)$  represented as a tree:

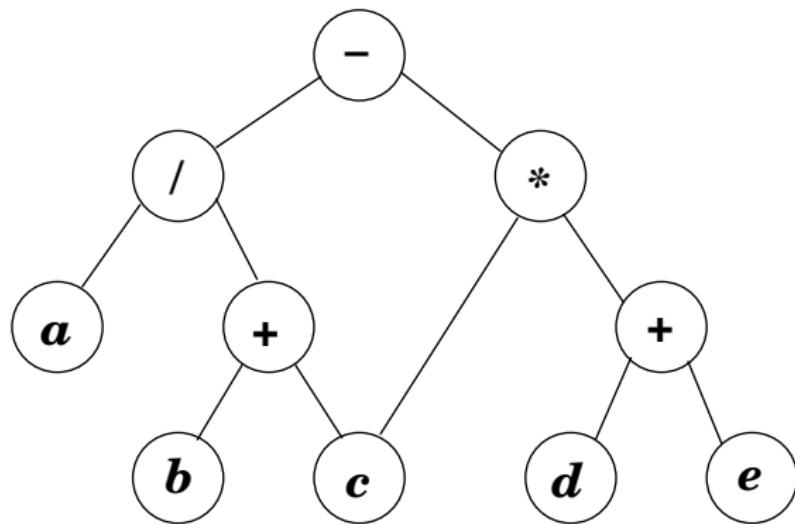
## Expression Trees

- ▶ Here is the expression  $a/(b+c) - c * (d+e)$  represented as a tree:



## Expression Trees

- ▶ We have not identified common sub-expressions; else we would have a directed acyclic graph (DAG):



# Expression Trees

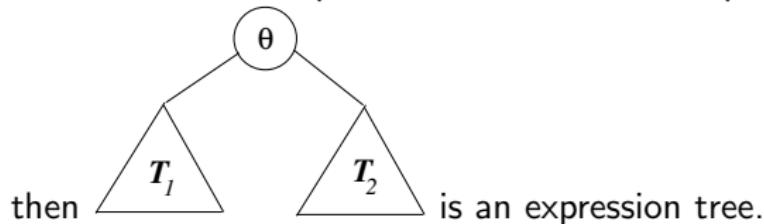
- ▶ Let  $\Sigma$  be a countable set of variable names, and  $\Theta$  be a finite set of binary operators. Then,

# Expression Trees

- ▶ Let  $\Sigma$  be a countable set of variable names, and  $\Theta$  be a finite set of binary operators. Then,
  1. A single vertex labeled by a name from  $\Sigma$  is an expression tree.

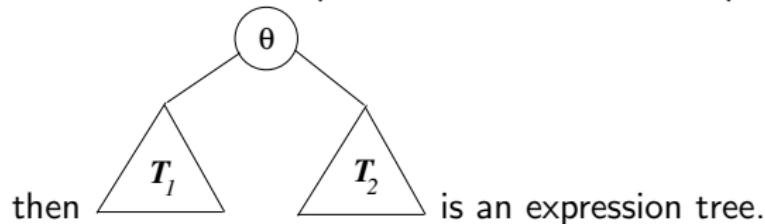
# Expression Trees

- ▶ Let  $\Sigma$  be a countable set of variable names, and  $\Theta$  be a finite set of binary operators. Then,
  1. A single vertex labeled by a name from  $\Sigma$  is an expression tree.
  2. If  $T_1$  and  $T_2$  are expression trees and  $\theta$  is a operator in  $\Theta$ ,



## Expression Trees

- ▶ Let  $\Sigma$  be a countable set of variable names, and  $\Theta$  be a finite set of binary operators. Then,
  1. A single vertex labeled by a name from  $\Sigma$  is an expression tree.
  2. If  $T_1$  and  $T_2$  are expression trees and  $\theta$  is a operator in  $\Theta$ ,



- ▶ In this example
- $$\Sigma = \{a, b, c, d, e, \dots\}, \text{ and } \Theta = \{+, -, *, /, \dots\}$$

## Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:

# Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:

1.  $m \leftarrow r$       (store instruction)

# Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:
  1.  $m \leftarrow r$  (store instruction)
  2.  $r \leftarrow m$  (load instruction)

# Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:
  1.  $m \leftarrow r$  (store instruction)
  2.  $r \leftarrow m$  (load instruction)
  3.  $r \leftarrow r \ op \ m$  (the result of  $r \ op \ m$  is stored in  $r$ )

# Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:
  1.  $m \leftarrow r$  (store instruction)
  2.  $r \leftarrow m$  (load instruction)
  3.  $r \leftarrow r \ op \ m$  (the result of  $r \ op \ m$  is stored in  $r$ )
  4.  $r_2 \leftarrow r_2 \ op \ r_1$  (the result of  $r_2 \ op \ r_1$  is stored in  $r_2$ )

# Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:
  1.  $m \leftarrow r$  (store instruction)
  2.  $r \leftarrow m$  (load instruction)
  3.  $r \leftarrow r \ op \ m$  (the result of  $r \ op \ m$  is stored in  $r$ )
  4.  $r_2 \leftarrow r_2 \ op \ r_1$  (the result of  $r_2 \ op \ r_1$  is stored in  $r_2$ )
- ▶ Note:

# Target Machine Model

- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:
  1.  $m \leftarrow r$  (store instruction)
  2.  $r \leftarrow m$  (load instruction)
  3.  $r \leftarrow r \ op \ m$  (the result of  $r \ op \ m$  is stored in  $r$ )
  4.  $r_2 \leftarrow r_2 \ op \ r_1$  (the result of  $r_2 \ op \ r_1$  is stored in  $r_2$ )
- ▶ Note:
  1. In instruction 3, the memory location is the right operand.

# Target Machine Model

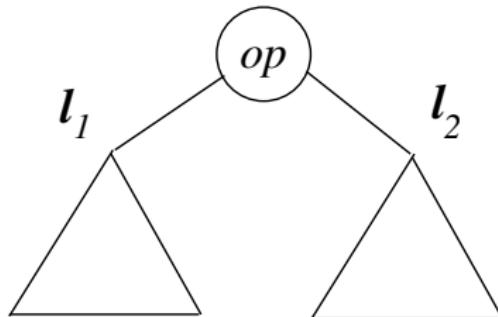
- ▶ We assume a machine with finite set of registers  $r_0, r_1, \dots, r_k$ , countable set of memory locations, and instructions of the form:
  1.  $m \leftarrow r$  (store instruction)
  2.  $r \leftarrow m$  (load instruction)
  3.  $r \leftarrow r \ op \ m$  (the result of  $r \ op \ m$  is stored in  $r$ )
  4.  $r_2 \leftarrow r_2 \ op \ r_1$  (the result of  $r_2 \ op \ r_1$  is stored in  $r_2$ )
- ▶ Note:
  1. In instruction 3, the memory location is the right operand.
  2. In instruction 4, the destination register is the same as the left operand register.

## Key Idea

- ▶ Determines an evaluation order of the subtrees which requires *minimum number of registers*.

## Key Idea

- ▶ Determines an evaluation order of the subtrees which requires *minimum number of registers*.
- ▶ If the left and right subtrees require  $l_1$ , and  $l_2$  ( $l_1 < l_2$ ) registers respectively, what should be the order of evaluation?



# Key Idea

- *Choice 1*

# Key Idea

- ▶ *Choice 1*

1. Evaluate left subtree first, leaving result in a register. This requires upto  $l_1$  registers.

# Key Idea

## ► *Choice 1*

1. Evaluate left subtree first, leaving result in a register. This requires upto  $l_1$  registers.
2. Evaluate the right subtree. During this we might require upto  $l_2 + 1$  registers ( $l_2$  registers for evaluating the right subtree and one register to hold the value of the left subtree.)

# Key Idea

- ▶ *Choice 1*
  1. Evaluate left subtree first, leaving result in a register. This requires upto  $l_1$  registers.
  2. Evaluate the right subtree. During this we might require upto  $l_2 + 1$  registers ( $l_2$  registers for evaluating the right subtree and one register to hold the value of the left subtree.)
- ▶ The maximum register requirement in this case is  
$$\max(l_1, l_2 + 1) = l_2 + 1.$$

# Key Idea

- ▶ *Choice 2*

# Key Idea

- ▶ *Choice 2*

1. Evaluate the right subtree first, leaving the result in a register.  
During this evaluation we shall require upto  $l_2$  registers.

# Key Idea

- ▶ *Choice 2*

1. Evaluate the right subtree first, leaving the result in a register.  
During this evaluation we shall require upto  $l_2$  registers.
2. Evaluate the left subtree. During this, we might require upto  $l_1 + 1$  registers.

# Key Idea

- ▶ *Choice 2*

1. Evaluate the right subtree first, leaving the result in a register.  
During this evaluation we shall require upto  $l_2$  registers.
2. Evaluate the left subtree. During this, we might require upto  $l_1 + 1$  registers.

- ▶ The maximum register requirement over the whole tree is

$$\max(l_1 + 1, l_2) = l_2$$

# Key Idea

- ▶ *Choice 2*

1. Evaluate the right subtree first, leaving the result in a register.  
During this evaluation we shall require upto  $l_2$  registers.
2. Evaluate the left subtree. During this, we might require upto  $l_1 + 1$  registers.

- ▶ The maximum register requirement over the whole tree is

$$\max(l_1 + 1, l_2) = l_2$$

# Key Idea

- ▶ *Choice 2*

1. Evaluate the right subtree first, leaving the result in a register.  
During this evaluation we shall require upto  $l_2$  registers.
2. Evaluate the left subtree. During this, we might require upto  $l_1 + 1$  registers.

- ▶ The maximum register requirement over the whole tree is

$$\max(l_1 + 1, l_2) = l_2$$

*Therefore the subtree requiring more registers should be evaluated first.*

## Labeling the Expression Tree

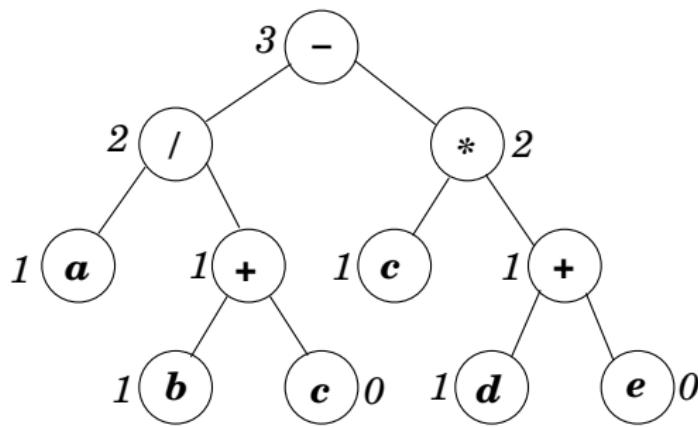
- ▶ Label each node by the number of registers required to evaluate it in a store free manner.

## Labeling the Expression Tree

- ▶ Label each node by the number of registers required to evaluate it in a store free manner.

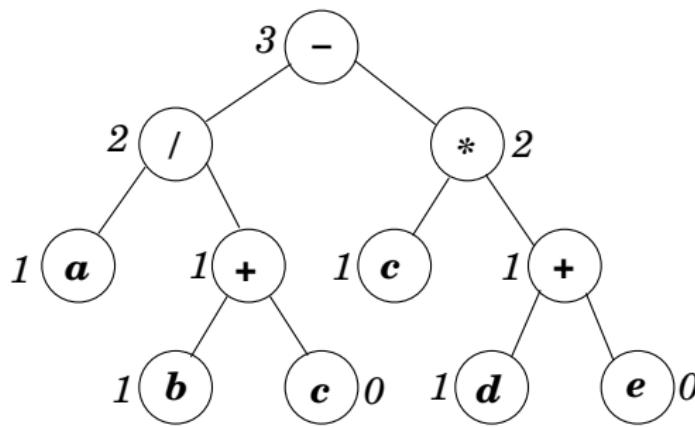
## Labeling the Expression Tree

- ▶ Label each node by the number of registers required to evaluate it in a store free manner.



## Labeling the Expression Tree

- ▶ Label each node by the number of registers required to evaluate it in a store free manner.



- ▶ Left and the right leaves are labeled 1 and 0 respectively, because the left leaf must necessarily be in a register, whereas the right leaf can reside in memory.

## Labeling the Expression Tree

- ▶ Visit the tree in post-order. For every node visited do:

## Labeling the Expression Tree

- ▶ Visit the tree in post-order. For every node visited do:
  1. Label each left leaf by 1 and each right leaf by 0.

## Labeling the Expression Tree

- ▶ Visit the tree in post-order. For every node visited do:
  1. Label each left leaf by 1 and each right leaf by 0.
  2. If the labels of the children of a node  $n$  are  $l_1$  and  $l_2$  respectively, then

$$\begin{aligned} \text{label}(n) &= \max(l_1, l_2), \text{ if } l_1 \neq l_2 \\ &= l_1 + 1, \text{ otherwise} \end{aligned}$$

## Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function  $\text{gencode}(n)$ , which produces code to evaluate the node labeled  $n$ .

## Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function  $\text{gencode}(n)$ , which produces code to evaluate the node labeled  $n$ .
2. Register allocation is done from a stack of register names  $rstack$ , initially containing  $r_0, r_1, \dots, r_k$  (with  $r_0$  on top of the stack).

## Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function  $\text{gencode}(n)$ , which produces code to evaluate the node labeled  $n$ .
2. Register allocation is done from a stack of register names  $rstack$ , initially containing  $r_0, r_1, \dots, r_k$  (with  $r_0$  on top of the stack).
3.  $\text{gencode}(n)$  evaluates  $n$  in the register on the top of the stack.

## Assumptions and Notational Conventions

1. The code generation algorithm is represented as a function  $\text{gencode}(n)$ , which produces code to evaluate the node labeled  $n$ .
2. Register allocation is done from a stack of register names  $rstack$ , initially containing  $r_0, r_1, \dots, r_k$  (with  $r_0$  on top of the stack).
3.  $\text{gencode}(n)$  evaluates  $n$  in the register on the top of the stack.
4. Temporary allocation is done from a stack of temporary names  $tstack$ , initially containing  $t_0, t_1, \dots, t_k$  (with  $t_0$  on top of the stack).

## Assumptions and Notational Conventions

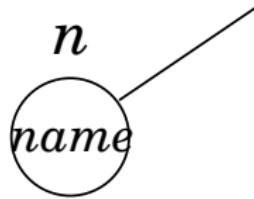
1. The code generation algorithm is represented as a function  $\text{gencode}(n)$ , which produces code to evaluate the node labeled  $n$ .
2. Register allocation is done from a stack of register names  $rstack$ , initially containing  $r_0, r_1, \dots, r_k$  (with  $r_0$  on top of the stack).
3.  $\text{gencode}(n)$  evaluates  $n$  in the register on the top of the stack.
4. Temporary allocation is done from a stack of temporary names  $tstack$ , initially containing  $t_0, t_1, \dots, t_k$  (with  $t_0$  on top of the stack).
5.  $\text{swap}(rstack)$  swaps the top two registers on the stack.

# The Algorithm

- ▶  $\text{gencode}(n)$  described by case analysis on the type of the node  $n$ .

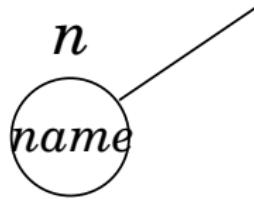
# The Algorithm

- ▶  $\text{gencode}(n)$  described by case analysis on the type of the node  $n$ .
  1.  $n$  is a left leaf:



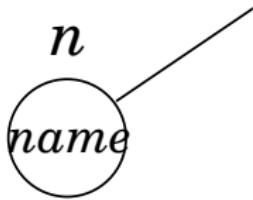
# The Algorithm

- ▶  $\text{gencode}(n)$  described by case analysis on the type of the node  $n$ .
  1.  $n$  is a left leaf:



# The Algorithm

- ▶  $\text{gencode}(n)$  described by case analysis on the type of the node  $n$ .
  1.  $n$  is a left leaf:

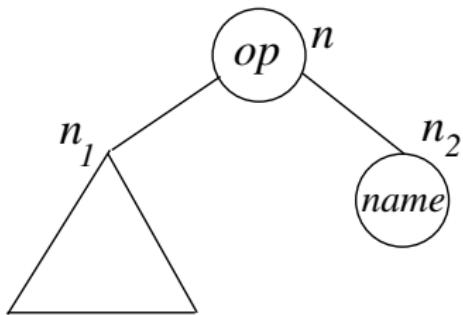


$\text{gen}(\text{top}(\text{rstack}) \leftarrow \text{name})$

*Comments:*  $n$  is named by a variable say  $\text{name}$ . Code is generated to load  $\text{name}$  into a register.

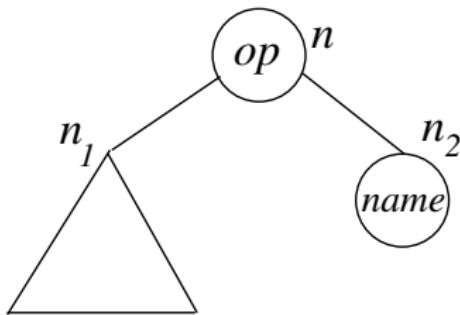
# The Algorithm

2. *n's right child is a leaf:*



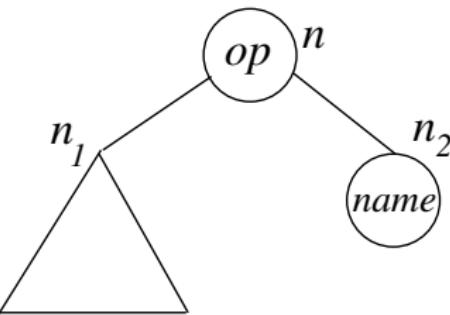
# The Algorithm

2. *n's right child is a leaf:*



# The Algorithm

2. *n's right child is a leaf:*



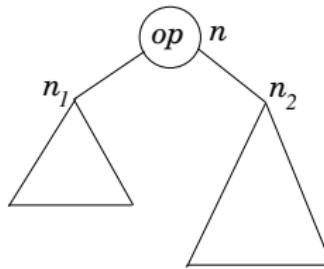
$\text{gencode}(n_1)$

$\text{gen}(\text{top}(\text{rstack}) \leftarrow \text{top}(\text{rstack}) \text{ op } \text{name})$

*Comments:*  $n_1$  is first evaluated in the register on the top of the stack, followed by the operation  $op$  leaving the result in the same register.

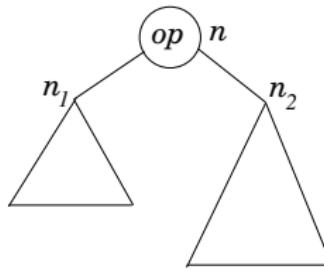
# The Algorithm

3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



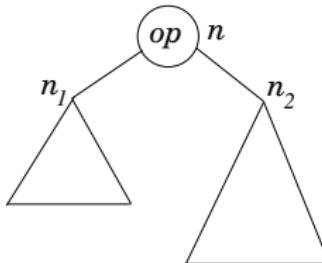
# The Algorithm

3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



# The Algorithm

3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*

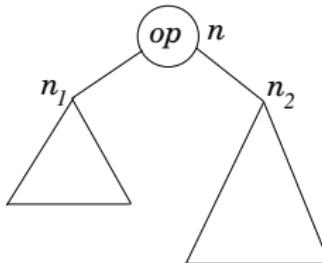


`swap(rstack);`

Right child goes into next to top register

# The Algorithm

3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



`swap(rstack);`

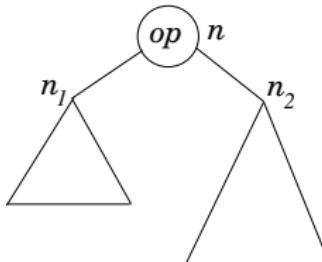
`gencode(n2);`

Right child goes into next to top register

Evaluate right child

# The Algorithm

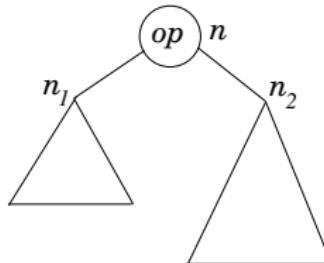
3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



`swap(rstack);`      Right child goes into next to top register  
`gencode( $n_2$ );`      Evaluate right child  
 $R := \text{pop}(\text{rstack});$

# The Algorithm

3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



`swap(rstack);` Right child goes into next to top register

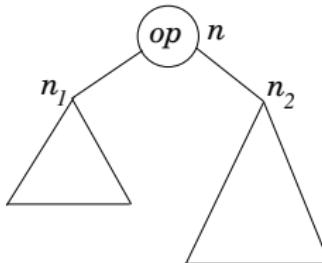
`gencode( $n_2$ );` Evaluate right child

$R := \text{pop}(rstack);$

`gencode( $n_1$ );` Evaluate left child

# The Algorithm

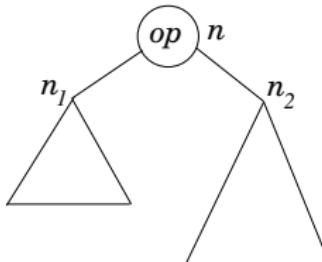
3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



`swap(rstack);`      Right child goes into next to top register  
`gencode( $n_2$ );`      Evaluate right child  
 $R := \text{pop}(\text{rstack});$   
`gencode( $n_1$ );`      Evaluate left child  
`gen(\text{top}(\text{rstack}) \leftarrow \text{top}(\text{rstack}) \text{ op } R);`      Issue  $op$

# The Algorithm

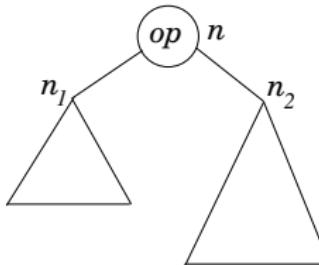
3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



`swap(rstack);`                    Right child goes into next to top register  
`gencode( $n_2$ );`                Evaluate right child  
 $R := \text{pop}(\text{rstack});$   
`gencode( $n_1$ );`                Evaluate left child  
`gen(\text{top}(\text{rstack}) \leftarrow \text{top}(\text{rstack}) \text{ op } R);`            Issue  $op$   
`push(rstack, R);`

# The Algorithm

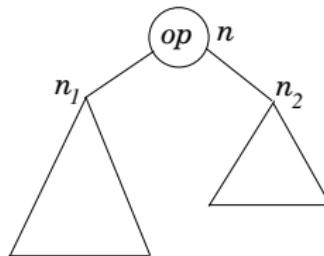
3. *The left child of  $n$  requires lesser number of registers. This requirement is strictly less than the available number of registers*



`swap(rstack);` Right child goes into next to top register  
`gencode( $n_2$ );` Evaluate right child  
 $R := \text{pop}(rstack);$   
`gencode( $n_1$ );` Evaluate left child  
`gen( $\text{top}(rstack)$ )  $\leftarrow \text{top}(rstack) \text{ op } R;$`  Issue  $op$   
`push(rstack,  $R$ );`  
`swap(rstack)` Restore register stack

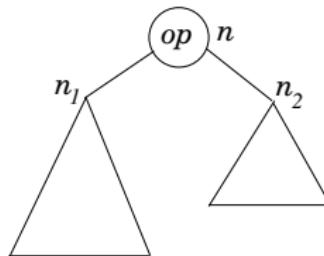
# The Algorithm

4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



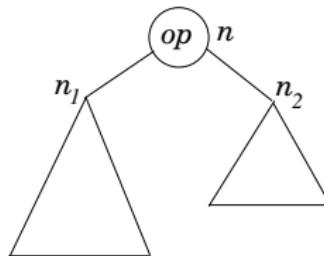
# The Algorithm

4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



# The Algorithm

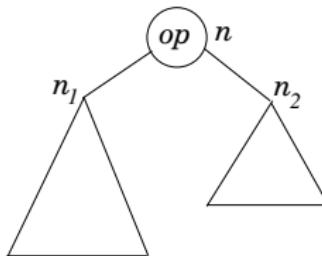
4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



$\text{gencode}(n_1);$

# The Algorithm

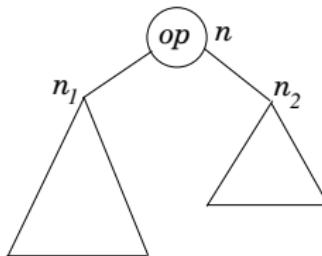
4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



*gencode( $n_1$ );  
 $R := pop(rstack);$*

# The Algorithm

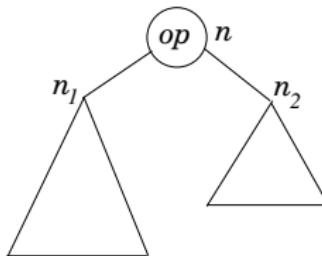
4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



```
gencode(n1);  
 $R := pop(rstack);$   
gencode(n2);
```

# The Algorithm

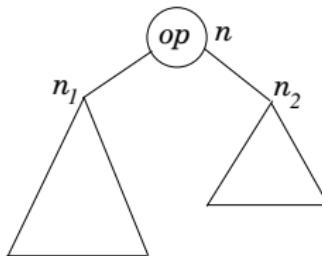
4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



```
gencode(n1);  
R := pop(rstack);  
gencode(n2);  
gen(R ← R op top(rstack));
```

# The Algorithm

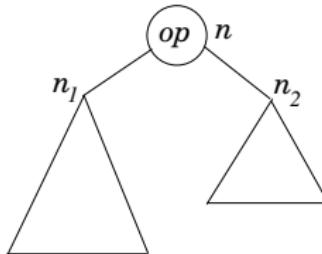
4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*



```
gencode(n1);  
R := pop(rstack);  
gencode(n2);  
gen(R ← R op top(rstack));  
push(rstack, R)
```

# The Algorithm

4. *The right child of  $n$  requires lesser (or the same) number of registers than the left child, and this requirement is strictly less than the available number of registers*

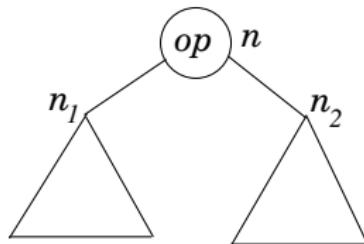


```
gencode(n1);
R := pop(rstack);
gencode(n2);
gen(R ← R op top(rstack));
push(rstack, R)
```

*Comments:* Same as case 3, except that the left sub-tree is evaluated first.

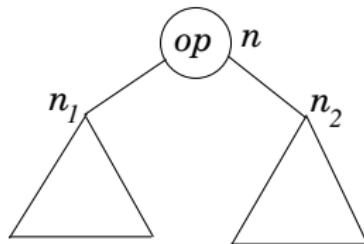
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



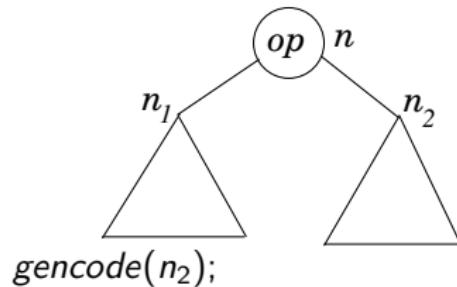
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



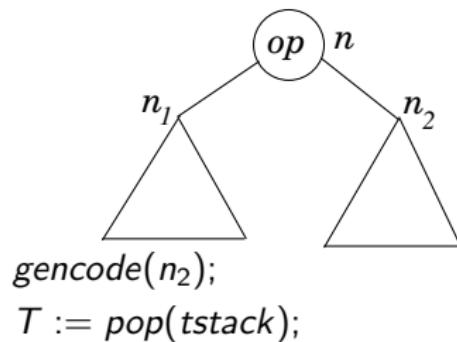
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



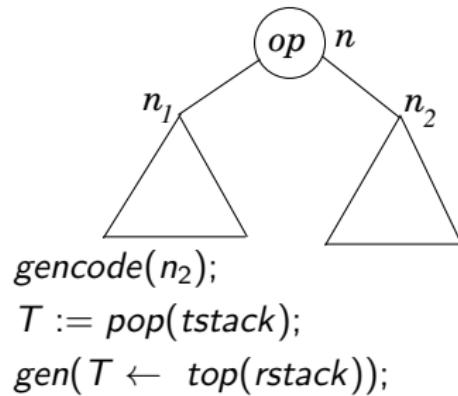
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



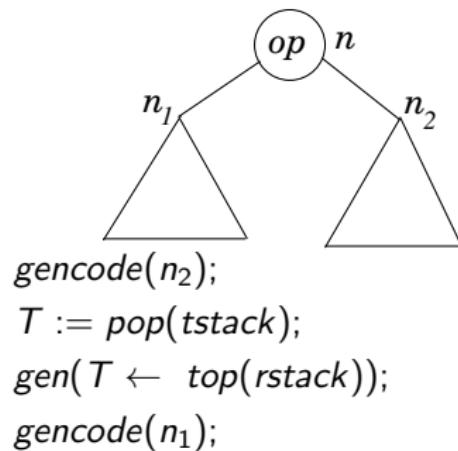
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



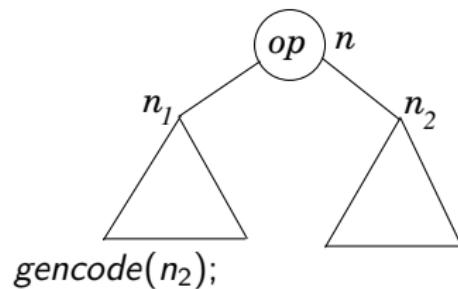
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



*gencode( $n_2$ );*

*$T := pop(tstack);$*

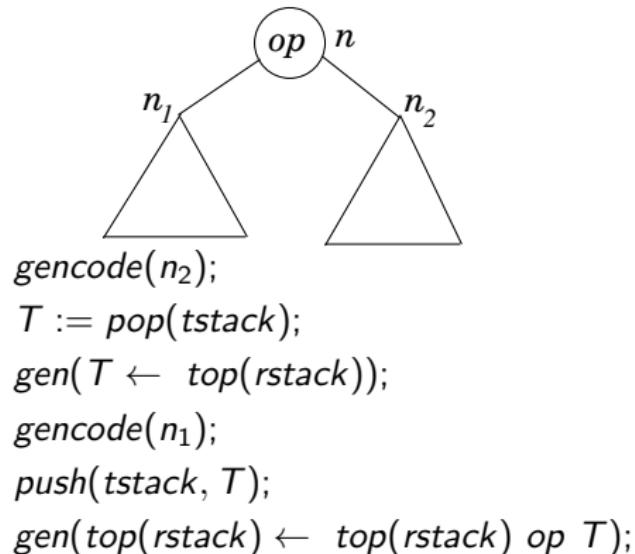
*gen( $T \leftarrow top(rstack)$ );*

*gencode( $n_1$ );*

*push( $tstack, T$ );*

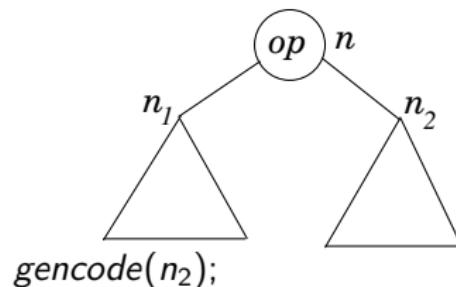
# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.



# The Algorithm

5. Both the children of  $n$  require registers greater or equal to the available number of registers.

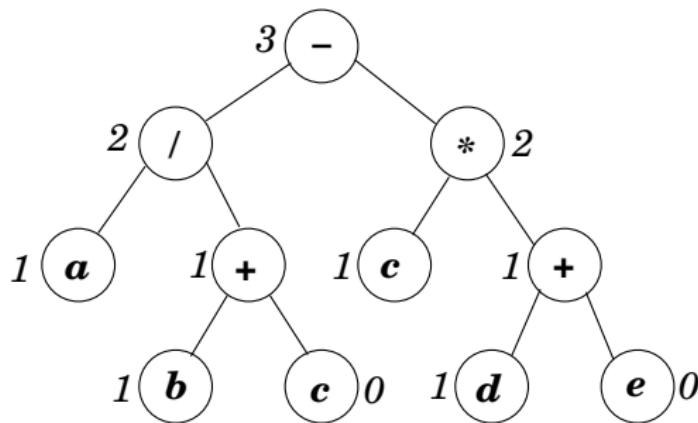


```
gencode(n2);
T := pop(tstack);
gen(T ← top(rstack));
gencode(n1);
push(tstack, T);
gen(top(rstack) ← top(rstack) op T);
```

*Comments:* In this case the right sub-tree is first evaluated into a temporary. This is followed by the evaluations of the left sub-tree and  $n$  into the register on the top of the stack.

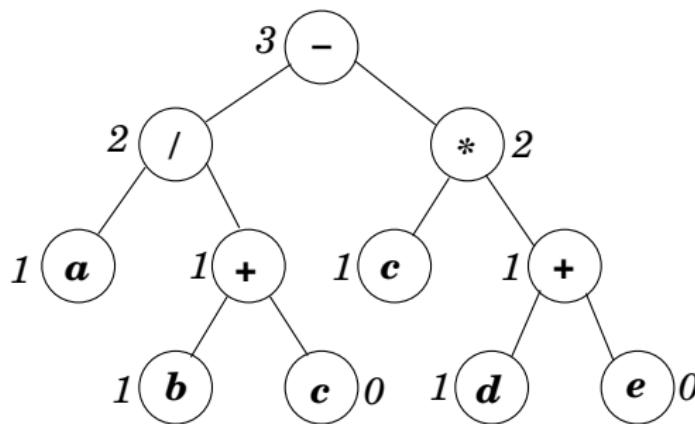
# An Example

For the example:



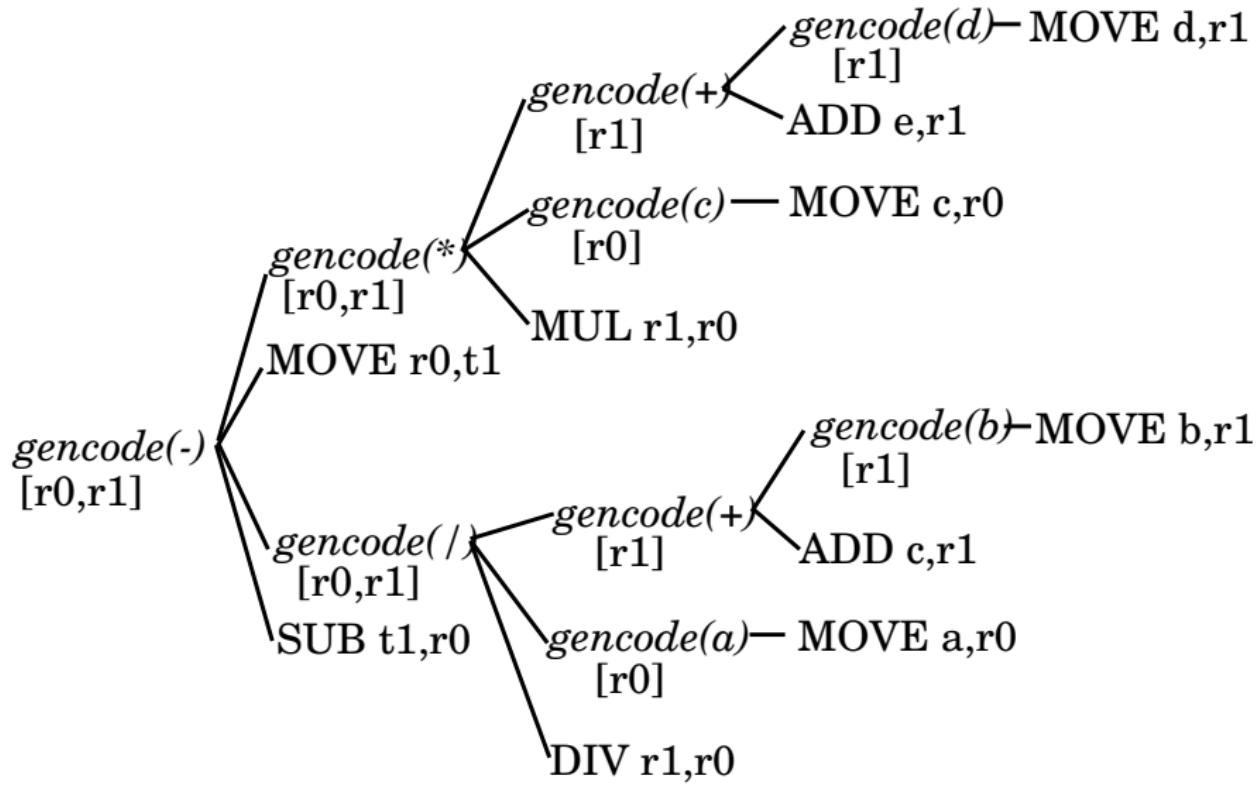
## An Example

For the example:



assuming two available registers  $r_0$  and  $r_1$ , the calls to gencode and the generated code are shown on the next slide.

## An Example



## SETHI-ULLMAN ALGORITHM: OPTIMALITY

- ▶ The algorithm is optimal because

## SETHI-ULLMAN ALGORITHM: OPTIMALITY

- ▶ The algorithm is optimal because
  1. The number of load instructions generated is optimal.

## SETHI-ULLMAN ALGORITHM: OPTIMALITY

- ▶ The algorithm is optimal because
  1. The number of load instructions generated is optimal.
  2. Each binary operation specified in the expression tree is performed only once.

# SETHI-ULLMAN ALGORITHM: OPTIMALITY

- ▶ The algorithm is optimal because
  1. The number of load instructions generated is optimal.
  2. Each binary operation specified in the expression tree is performed only once.
  3. The number of stores is optimal.

# SETHI-ULLMAN ALGORITHM: OPTIMALITY

- ▶ The algorithm is optimal because
  1. The number of load instructions generated is optimal.
  2. Each binary operation specified in the expression tree is performed only once.
  3. The number of stores is optimal.
- ▶ We shall now elaborate on each of these.

## SETHI-ULLMAN ALGORITHM: OPTIMALITY

1. It is easy to verify that the number of loads required by any program computing an expression tree is at least equal to the number of left leaves. This algorithm generates no more loads than this.

## SETHI-ULLMAN ALGORITHM: OPTIMALITY

1. It is easy to verify that the number of loads required by any program computing an expression tree is at least equal to the number of left leaves. This algorithm generates no more loads than this.
2. Each node of the expression tree is visited exactly once. If this node specifies a binary operation, then the algorithm branches into steps 2,3,4 or 5, and at each of these places code is generated to perform this operation exactly once.

## SETHI-ULLMAN ALGORITHM: OPTIMALITY

3. The number of stores is optimal: this is harder to show.

# SETHI-ULLMAN ALGORITHM: OPTIMALITY

3. The number of stores is optimal: this is harder to show.
  - ▶ Define a *major node* as a node, each of whose children has a label at least equal to the number of available registers.

# SETHI-ULLMAN ALGORITHM: OPTIMALITY

3. The number of stores is optimal: this is harder to show.
  - ▶ Define a *major node* as a node, each of whose children has a label at least equal to the number of available registers.
  - ▶ If we can show that the number of stores required by *any program* computing an expression tree is at least equal the number of major nodes, then our algorithm produces minimal number of stores (Why?)

## SETHI-ULLMAN ALGORITHM

- ▶ To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.

## SETHI-ULLMAN ALGORITHM

- ▶ To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- ▶ Assume that the tree has  $M$  major nodes.

## SETHI-ULLMAN ALGORITHM

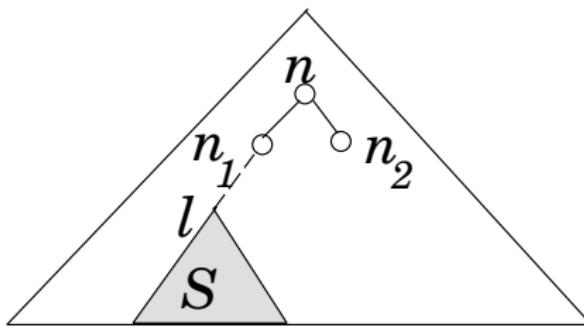
- ▶ To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- ▶ Assume that the tree has  $M$  major nodes.
- ▶ Now consider a tree formed by replacing the subtree  $S$  evaluated by the first store, with a leaf labeled by a name  $l$ .

## SETHI-ULLMAN ALGORITHM

- ▶ To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- ▶ Assume that the tree has  $M$  major nodes.
- ▶ Now consider a tree formed by replacing the subtree  $S$  evaluated by the first store, with a leaf labeled by a name  $l$ .

## SETHI-ULLMAN ALGORITHM

- ▶ To see this, consider an expression tree and the code generated by any optimal algorithm for this tree.
- ▶ Assume that the tree has  $M$  major nodes.
- ▶ Now consider a tree formed by replacing the subtree  $S$  evaluated by the first store, with a leaf labeled by a name  $l$ .



- ▶ Let  $n$  be the major node in the original tree, just above  $S$ , and  $n_1$  and  $n_2$  be its immediate descendants ( $n_1$  could be  $l$  itself).

## SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of  $n_1$  might have decreased but the label of  $n_2$  remains unaffected ( $\geq k$ , the available number of registers).

## SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of  $n_1$  might have decreased but the label of  $n_2$  remains unaffected ( $\geq k$ , the available number of registers).
2. The label of  $n$  is  $\geq k$ .

## SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of  $n_1$  might have decreased but the label of  $n_2$  remains unaffected ( $\geq k$ , the available number of registers).
2. The label of  $n$  is  $\geq k$ .
3. The node  $n$  may no longer be a major node *but all other major nodes in the original tree continue to be major nodes in the modified tree.*

## SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of  $n_1$  might have decreased but the label of  $n_2$  remains unaffected ( $\geq k$ , the available number of registers).
2. The label of  $n$  is  $\geq k$ .
3. The node  $n$  may no longer be a major node *but all other major nodes in the original tree continue to be major nodes in the modified tree.*
4. Therefore the number of major nodes in the modified tree is  $M - 1$ .

## SETHI-ULLMAN ALGORITHM

1. In the modified tree, the (modified) label of  $n_1$  might have decreased but the label of  $n_2$  remains unaffected ( $\geq k$ , the available number of registers).
2. The label of  $n$  is  $\geq k$ .
3. The node  $n$  may no longer be a major node *but all other major nodes in the original tree continue to be major nodes in the modified tree.*
4. Therefore the number of major nodes in the modified tree is  $M - 1$ .
5. If we assume as induction hypothesis that the number of stores for the modified tree is at least  $M - 1$ , then the number of stores for the original tree is at least  $M$ .

## SETHI-ULLMAN ALGORITHM: COMPLEXITY

Since the algorithm visits every node of the expression tree twice – once during labeling, and once during code generation, the complexity of the algorithm is  $O(n)$ .

# Code Generation: Aho Johnson Algorithm

Amey Karkare

karkare@cse.iitk.ac.in

April 5, 2019

# Aho-Johnson Algorithm

# Characteristics of the Algorithm

- ▶ Considers expression trees.

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction
  - ▶ can have a root of any arity.

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction
  - ▶ can have a root of any arity.
  - ▶ can have as leaves registers or memory locations appearing in any order.

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction
  - ▶ can have a root of any arity.
  - ▶ can have as leaves registers or memory locations appearing in any order.
  - ▶ can be of of any height

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction
  - ▶ can have a root of any arity.
  - ▶ can have as leaves registers or memory locations appearing in any order.
  - ▶ can be of any height
- ▶ Does not use algebraic properties of operators.

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction
  - ▶ can have a root of any arity.
  - ▶ can have as leaves registers or memory locations appearing in any order.
  - ▶ can be of any height
- ▶ Does not use algebraic properties of operators.
- ▶ Generates optimal code, where, once again, the cost measure is the number of instructions in the code.

## Characteristics of the Algorithm

- ▶ Considers expression trees.
- ▶ The target machine model is general enough to generate code for a large class of machines.
- ▶ Represented as a tree, an instruction
  - ▶ can have a root of any arity.
  - ▶ can have as leaves registers or memory locations appearing in any order.
  - ▶ can be of of any height
- ▶ Does not use algebraic properties of operators.
- ▶ Generates optimal code, where, once again, the cost measure is the number of instructions in the code.
- ▶ Complexity is linear in the size of the expression tree.

## Expression Trees Defined

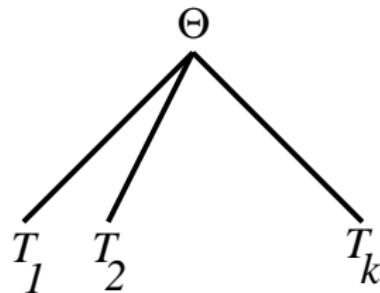
- ▶ Let  $\Sigma$  be a countable set of operands, and  $\Theta$  be a finite set of operators. Then,

## Expression Trees Defined

- ▶ Let  $\Sigma$  be a countable set of operands, and  $\Theta$  be a finite set of operators. Then,
  1. A single vertex labeled by a name from  $\Sigma$  is an expression tree.

## Expression Trees Defined

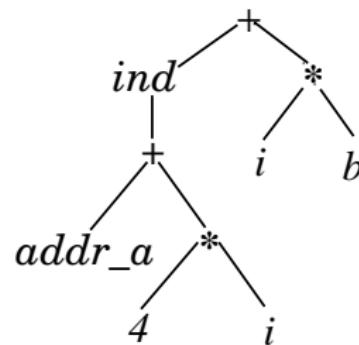
- ▶ Let  $\Sigma$  be a countable set of operands, and  $\Theta$  be a finite set of operators. Then,
  1. A single vertex labeled by a name from  $\Sigma$  is an expression tree.
  2. If  $T_1, T_2, \dots, T_k$  are expression trees whose leaves all have distinct labels and  $\theta$  is a k-ary operator in  $\Theta$ , then



is an expression tree.

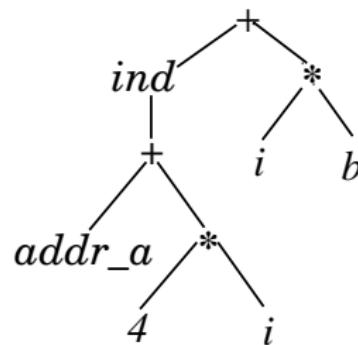
## Example

- ▶ An example of an expression tree is



## Example

- An example of an expression tree is



- Notation:** If  $T$  is an expression tree, and  $S$  is a subtree of  $T$ , then  $T/S$  is the tree obtained by replacing  $S$  in  $T$  by a single leaf labeled by a distinct name from  $\Sigma$ .

# The Machine Model

1. The machine has  $n$  general purpose registers (no special registers).

# The Machine Model

1. The machine has  $n$  general purpose registers (no special registers).
2. Countable sequence of memory locations.

# The Machine Model

1. The machine has  $n$  general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:

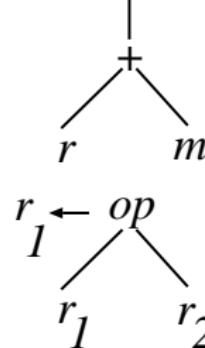
# The Machine Model

1. The machine has  $n$  general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:
  - a.  $r \leftarrow E$ ,  $r$  is a register and  $E$  is an expression tree whose operators are from  $\Theta$  and operands are registers, memory locations or constants. Further,  $r$  should be one of the register names occurring (if any) in  $E$ .

# The Machine Model

1. The machine has  $n$  general purpose registers (no special registers).
2. Countable sequence of memory locations.
3. Instructions are of the form:
  - a.  $r \leftarrow E$ ,  $r$  is a register and  $E$  is an expression tree whose operators are from  $\Theta$  and operands are registers, memory locations or constants. Further,  $r$  should be one of the register names occurring (if any) in  $E$ .
  - b.  $m \leftarrow r$ , a store instruction.

## Example Of A Machine

$r \leftarrow c$	$\{\text{MOV } \#c, r\}$
$r \leftarrow m$	$\{\text{MOV } m, r\}$
$m \leftarrow r$	$\{\text{MOV } r, m\}$
$r \leftarrow \text{ind}$	$\{\text{MOV } m(r), r\}$
	
$r_1 \leftarrow \text{op}$	$\{\text{op } r_2, r_1\}$
$r_1$	
$r_2$	

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

$$r_1 \leftarrow 4$$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

$r_1 \leftarrow 4$

$r_1 \leftarrow r_1 * i$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

$r_1 \leftarrow 4$

$r_1 \leftarrow r_1 * i$

$r_2 \leftarrow \text{addr\_}a$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

$r_1 \leftarrow 4$

$r_1 \leftarrow r_1 * i$

$r_2 \leftarrow \text{addr\_}a$

$r_2 \leftarrow r_2 + r_1$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

```
r1 ← 4  
r1 ← r1 * i  
r2 ← addr_a  
r2 ← r2 + r1  
r2 ← ind(r2)
```

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

$r_1 \leftarrow 4$

$r_1 \leftarrow r_1 * i$

$r_2 \leftarrow \text{addr\_}a$

$r_2 \leftarrow r_2 + r_1$

$r_2 \leftarrow \text{ind}(r_2)$

$r_3 \leftarrow i$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

$r_1 \leftarrow 4$

$r_1 \leftarrow r_1 * i$

$r_2 \leftarrow \text{addr\_}a$

$r_2 \leftarrow r_2 + r_1$

$r_2 \leftarrow \text{ind}(r_2)$

$r_3 \leftarrow i$

$r_3 \leftarrow r_3 * b$

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

```
r1 ← 4  
r1 ← r1 * i  
r2 ← addr_a  
r2 ← r2 + r1  
r2 ← ind(r2)  
r3 ← i  
r3 ← r3 * b  
r2 ← r2 + r3
```

# MACHINE PROGRAM

- ▶ A *machine program* consists of a finite sequence of instructions  $P = I_1 I_2 \dots I_q$ .
- ▶ The machine program below evaluates  $a[i] + i * b$

```
r1 ← 4  
r1 ← r1 * i  
r2 ← addr_a  
r2 ← r2 + r1  
r2 ← ind(r2)  
r3 ← i  
r3 ← r3 * b  
r2 ← r2 + r3
```

# VALUE OF A PROGRAM

- ▶ We need to define the value  $v(P)$  computed by a program  $P$ .

# VALUE OF A PROGRAM

- ▶ We need to define the value  $v(P)$  computed by a program  $P$ .
  1. We want to specify what it means to say that a *program P computes an expression tree T*. This is when the value of the program  $v(P)$  is the same as  $T$ .

# VALUE OF A PROGRAM

- ▶ We need to define the value  $v(P)$  computed by a program  $P$ .
  1. We want to specify what it means to say that a *program P computes an expression tree T*. This is when the value of the program  $v(P)$  is the same as  $T$ .
  2. We also want to talk of *equivalence* of two programs  $P_1$  and  $P_2$ . This is true when  $v(P_1) = v(P_2)$ .

# VALUE OF A PROGRAM

- ▶ What is the *value of a program*  $P = I_1, I_2, \dots, I_q$ ?

# VALUE OF A PROGRAM

- ▶ What is the *value of a program*  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:

# VALUE OF A PROGRAM

- ▶ What is the *value of a program*  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:
- ▶ First define  $v_t(z)$ , the value of a memory location or register  $z$  after the execution of the instruction  $I_t$ .

# VALUE OF A PROGRAM

- ▶ What is the *value of a program*  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:
- ▶ First define  $v_t(z)$ , the value of a memory location or register  $z$  after the execution of the instruction  $I_t$ .
  - a. Initially  $v_0(z)$  is  $z$  if  $z$  is a memory location, else it is undefined.

# VALUE OF A PROGRAM

- ▶ What is the *value of a program*  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:
- ▶ First define  $v_t(z)$ , the value of a memory location or register  $z$  after the execution of the instruction  $I_t$ .
  - a. Initially  $v_0(z)$  is  $z$  if  $z$  is a memory location, else it is undefined.
  - b. If  $I_t$  is  $r \leftarrow E$ , then  $v_t(r)$  is the tree obtained by taking the tree representing  $E$ , and substituting for each leaf  $l$  the value of  $v_{t-1}(l)$ .

# VALUE OF A PROGRAM

- ▶ What is the *value* of a program  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:
- ▶ First define  $v_t(z)$ , the value of a memory location or register  $z$  after the execution of the instruction  $I_t$ .
  - a. Initially  $v_0(z)$  is  $z$  if  $z$  is a memory location, else it is undefined.
  - b. If  $I_t$  is  $r \leftarrow E$ , then  $v_t(r)$  is the tree obtained by taking the tree representing  $E$ , and substituting for each leaf  $l$  the value of  $v_{t-1}(l)$ .
  - c. If  $I_t$  is  $m \leftarrow r$ , then  $v_t(m)$  is  $v_{t-1}(r)$ .

# VALUE OF A PROGRAM

- ▶ What is the *value of a program*  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:
- ▶ First define  $v_t(z)$ , the value of a memory location or register  $z$  after the execution of the instruction  $I_t$ .
  - a. Initially  $v_0(z)$  is  $z$  if  $z$  is a memory location, else it is undefined.
  - b. If  $I_t$  is  $r \leftarrow E$ , then  $v_t(r)$  is the tree obtained by taking the tree representing  $E$ , and substituting for each leaf  $l$  the value of  $v_{t-1}(l)$ .
  - c. If  $I_t$  is  $m \leftarrow r$ , then  $v_t(m)$  is  $v_{t-1}(r)$ .
  - d. Otherwise  $v_t(z) = v_{t-1}(z)$ .

# VALUE OF A PROGRAM

- ▶ What is the *value* of a program  $P = I_1, I_2, \dots, I_q$ ?
- ▶ It is a tree, defined as follows:
- ▶ First define  $v_t(z)$ , the value of a memory location or register  $z$  after the execution of the instruction  $I_t$ .
  - a. Initially  $v_0(z)$  is  $z$  if  $z$  is a memory location, else it is undefined.
  - b. If  $I_t$  is  $r \leftarrow E$ , then  $v_t(r)$  is the tree obtained by taking the tree representing  $E$ , and substituting for each leaf  $l$  the value of  $v_{t-1}(l)$ .
  - c. If  $I_t$  is  $m \leftarrow r$ , then  $v_t(m)$  is  $v_{t-1}(r)$ .
  - d. Otherwise  $v_t(z) = v_{t-1}(z)$ .
- ▶ If  $I_q$  is  $z \leftarrow E$ , then the value of  $P$  is  $v_q(z)$ .

# EXAMPLE

- ▶ For the program:

$$r_1 \leftarrow b$$
$$r_1 \leftarrow r_1 + c$$
$$r_2 \leftarrow a$$
$$r_2 \leftarrow r_2 * \text{ind}(r_1)$$

# EXAMPLE

- ▶ For the program:

$$r_1 \leftarrow b$$

$$r_1 \leftarrow r_1 + c$$

$$r_2 \leftarrow a$$

$$r_2 \leftarrow r_2 * \text{ind}(r_1)$$

- ▶ the values of  $r_1$ ,  $r_2$ ,  $a$ ,  $b$  and  $c$  at different time instants are:

	$r_1$	$r_2$	$a$	$b$	$c$
<b>before 1</b>	$U$	$U$	$a$	$b$	$c$
<b>after 1</b>	$b$	$U$	$a$	$b$	$c$
<b>after 2</b>		$U$	$a$	$b$	$c$
		$a$	$a$	$b$	$c$
<b>after 3</b>		$a$	$a$	$b$	$c$
		$a$	$a$	$b$	$c$
<b>after 4</b>		$a$	$b$	$c$	
		$b$	$c$		

## EXAMPLE

- ▶ For the program:

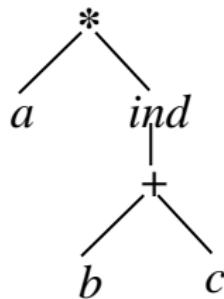
$$r_1 \leftarrow b$$

$$r_1 \leftarrow r_1 + c$$

$$r_2 \leftarrow a$$

$$r_2 \leftarrow r_2 * \text{ind}(r_1)$$

- ▶ The values of of the program is



## USELESS INSTRUCTIONS

- ▶ An instruction  $I_t$  in a program  $P$  is said to be *useless*, if the program  $P_1$  formed by removing  $I_t$  from  $P$  is equivalent to  $P$ .

## USELESS INSTRUCTIONS

- ▶ An instruction  $I_t$  in a program  $P$  is said to be *useless*, if the program  $P_1$  formed by removing  $I_t$  from  $P$  is equivalent to  $P$ .
- ▶ NOTE: We shall assume that our programs do not have any useless instructions.

# SCOPE OF INSTRUCTIONS

- ▶ The *scope of an instruction*  $I_t$  in a program  $P = I_1 I_2 \dots I_q$  is the sequence of instructions  $I_{t+1}, \dots, I_s$ , where  $s$  is the largest index such that

# SCOPE OF INSTRUCTIONS

- ▶ The *scope of an instruction*  $I_t$  in a program  $P = I_1 I_2 \dots I_q$  is the sequence of instructions  $I_{t+1}, \dots, I_s$ , where  $s$  is the largest index such that
  - a. The register or memory location defined by  $I_t$  is used by  $I_s$ , and

# SCOPE OF INSTRUCTIONS

- ▶ The *scope of an instruction*  $I_t$  in a program  $P = I_1 I_2 \dots I_q$  is the sequence of instructions  $I_{t+1}, \dots, I_s$ , where  $s$  is the largest index such that
  - a. The register or memory location defined by  $I_t$  is used by  $I_s$ , and
  - b. This register/memory location is not redefined by the instructions between  $I_t$  and  $I_s$ .

# SCOPE OF INSTRUCTIONS

- ▶ The *scope of an instruction*  $I_t$  in a program  $P = I_1 I_2 \dots I_q$  is the sequence of instructions  $I_{t+1}, \dots, I_s$ , where  $s$  is the largest index such that
  - a. The register or memory location defined by  $I_t$  is used by  $I_s$ , and
  - b. This register/memory location is not redefined by the instructions between  $I_t$  and  $I_s$ .
- ▶ The relation between  $I_s$  and  $I_t$  is expressed by saying that  $I_s$  is the *last use* of  $I_t$ , and is denoted by  $s = U_p(t)$ .

# REARRANGABILITY OF PROGRAMS

- ▶ We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.

# REARRANGABILITY OF PROGRAMS

- ▶ We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.
- ▶ Why is this result important? This is because our algorithm considers programs which are in strong normal form only. The above result assures us that by doing so, we shall not miss out an optimal solution.

# REARRANGABILITY OF PROGRAMS

- ▶ We shall show that each program can be rearranged to obtain an equivalent program (of the same length) in *strong normal form*.
- ▶ Why is this result important? This is because our algorithm considers programs which are in strong normal form only. The above result assures us that by doing so, we shall not miss out an optimal solution.
- ▶ To show the above result, we shall have to consider the kinds of rearrangements which retain program equivalence.

## Rearrangement Theorem

- ▶ Let  $P = I_1, I_2, \dots, I_q$  be a program which computes an expression tree.

## Rearrangement Theorem

- ▶ Let  $P = I_1, I_2, \dots, I_q$  be a program which computes an expression tree.
- ▶ Let  $\pi$  be a permutation on  $\{1 \dots q\}$  with  $\pi(q) = q$ .

# Rearrangement Theorem

- ▶ Let  $P = I_1, I_2, \dots, I_q$  be a program which computes an expression tree.
- ▶ Let  $\pi$  be a permutation on  $\{1 \dots q\}$  with  $\pi(q) = q$ .
- ▶  $\pi$  induces a rearranged program  $Q = J_1, J_2, \dots, J_q$  with  $I_i$  in  $P$  becoming  $J_{\pi(i)}$  in  $Q$ .

# Rearrangement Theorem

- ▶ Let  $P = I_1, I_2, \dots, I_q$  be a program which computes an expression tree.
- ▶ Let  $\pi$  be a permutation on  $\{1 \dots q\}$  with  $\pi(q) = q$ .
- ▶  $\pi$  induces a rearranged program  $Q = J_1, J_2, \dots, J_q$  with  $I_i$  in  $P$  becoming  $J_{\pi(i)}$  in  $Q$ .
- ▶ Then  $Q$  is equivalent to  $P$  if  $\pi(U_P(t)) = U_Q(\pi(t))$ .

## Rearrangement Theorem: Notes

- ▶ The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.

## Rearrangement Theorem: Notes

- ▶ The rearrangement theorem merely states that a rearrangement retains program equivalence, if any variable defined by an instruction in the original program is last used by the same instructions in both the original and rearranged program.
- ▶ To see why the statement of the theorem is true, reason as follows.

## Rearrangement Theorem: Notes

- a.  $P$  is equivalent to  $Q$ , if the operands used by the last instruction  $I_q$  (also  $J_q$ ) have the same value in  $P$  and  $Q$ .

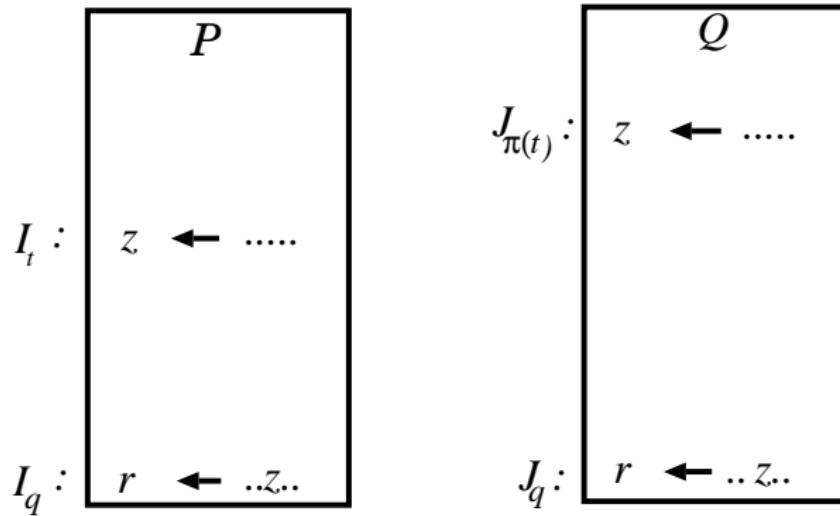
## Rearrangement Theorem: Notes

- a.  $P$  is equivalent to  $Q$ , if the operands used by the last instruction  $I_q$  (also  $J_q$ ) have the same value in  $P$  and  $Q$ .
- b. Consider any operand in  $I_q$ , say  $z$ . By the rearrangement theorem, This must have been defined by the same instruction (though in different positions say  $I_t$  and  $J_{\pi(t)}$ ) in  $P$  and  $Q$ . So  $z$  in  $I_q$  and  $J_q$  have the same value, if the operands used by  $I_t$  and  $J_{\pi(t)}$  have the same value in  $P$  and  $Q$ .

## Rearrangement Theorem: Notes

- a.  $P$  is equivalent to  $Q$ , if the operands used by the last instruction  $I_q$  (also  $J_q$ ) have the same value in  $P$  and  $Q$ .
- b. Consider any operand in  $I_q$ , say  $z$ . By the rearrangement theorem, This must have been defined by the same instruction (though in different positions say  $I_t$  and  $J_{\pi(t)}$ ) in  $P$  and  $Q$ . So  $z$  in  $I_q$  and  $J_q$  have the same value, if the operands used by  $I_t$  and  $J_{\pi(t)}$  have the same value in  $P$  and  $Q$ .
- c. Repeat this argument, till you come across an instruction with all constants on the right hand side.

# Rearrangement Theorem: Notes



# WIDTH

- ▶ The *width* of a program is a measure of the minimum number of registers required to execute the program.

## WIDTH

- ▶ The *width* of a program is a measure of the minimum number of registers required to execute the program.
- ▶ Formally, if  $P$  is a program, then the *width of an instruction*  $I_t$  is the number of distinct  $j$ ,  $1 \leq j \leq t$ , with  $U_P(j) > t$ , and  $I_j$  not a store instruction.

## WIDTH

- ▶ The *width* of a program is a measure of the minimum number of registers required to execute the program.
- ▶ Formally, if  $P$  is a program, then the *width of an instruction*  $I_t$  is the number of distinct  $j$ ,  $1 \leq j \leq t$ , with  $U_P(j) > t$ , and  $I_j$  not a store instruction.

# WIDTH

- ▶ The *width* of a program is a measure of the minimum number of registers required to execute the program.
- ▶ Formally, if  $P$  is a program, then the *width of an instruction*  $I_t$  is the number of distinct  $j$ ,  $1 \leq j \leq t$ , with  $U_P(j) > t$ , and  $I_j$  not a store instruction.

$r_1 \leftarrow$   
 $r_2 \leftarrow$   
 $I_t :$                     Width = 2  
                           $\leftarrow r_1$   
                           $\leftarrow r_2$

# WIDTH

- ▶ The *width* of a program is a measure of the minimum number of registers required to execute the program.
- ▶ Formally, if  $P$  is a program, then the *width of an instruction*  $I_t$  is the number of distinct  $j$ ,  $1 \leq j \leq t$ , with  $U_P(j) > t$ , and  $I_j$  not a store instruction.

$r_1 \leftarrow$   
 $r_2 \leftarrow$   
 $I_t :$                     Width = 2  
                           $\leftarrow r_1$   
                           $\leftarrow r_2$

- ▶ The *width of a program*  $P$  is the maximum width over all instructions in  $P$ .

## WIDTH

- ▶ A program of width  $w$  (but possibly using more than  $w$  registers) can be rearranged into an equivalent program using exactly  $w$  registers.

# WIDTH

- ▶ A program of width  $w$  (but possibly using more than  $w$  registers) can be rearranged into an equivalent program using exactly  $w$  registers.
- ▶ EXAMPLE:

$$r_1 \leftarrow a$$
$$r_2 \leftarrow b$$
$$r_1 \leftarrow r_1 + r_2$$
$$r_3 \leftarrow c$$
$$r_3 \leftarrow r_3 + d$$
$$r_1 \leftarrow r_1 * r_3$$

# WIDTH

- ▶ A program of width  $w$  (but possibly using more than  $w$  registers) can be rearranged into an equivalent program using exactly  $w$  registers.
- ▶ EXAMPLE:

$$r_1 \leftarrow a$$
$$r_2 \leftarrow b$$
$$r_1 \leftarrow r_1 + r_2$$
$$r_3 \leftarrow c$$
$$r_3 \leftarrow r_3 + d$$
$$r_1 \leftarrow r_1 * r_3$$
$$r_1 \leftarrow a$$
$$r_2 \leftarrow b$$
$$r_1 \leftarrow r_1 + r_2$$
$$r_2 \leftarrow c$$
$$r_2 \leftarrow r_2 + d$$
$$r_1 \leftarrow r_1 * r_2$$

# WIDTH

- ▶ A program of width  $w$  (but possibly using more than  $w$  registers) can be rearranged into an equivalent program using exactly  $w$  registers.
- ▶ EXAMPLE:

$$r_1 \leftarrow a$$

$$r_2 \leftarrow b$$

$$r_1 \leftarrow r_1 + r_2$$

$$r_3 \leftarrow c$$

$$r_3 \leftarrow r_3 + d$$

$$r_1 \leftarrow r_1 * r_3$$

$$r_1 \leftarrow a$$

$$r_2 \leftarrow b$$

$$r_1 \leftarrow r_1 + r_2$$

$$r_2 \leftarrow c$$

$$r_2 \leftarrow r_2 + d$$

$$r_1 \leftarrow r_1 * r_2$$

- ▶ In the example above, the first program has width 2 but uses 3 registers. By suitable renaming, the number of registers in the second program has been brought down to 2.

## LEMMA

Let  $P$  be a program of width  $w$ , and let  $R$  be a set of  $w$  distinct registers. Then, by renaming the registers used by  $P$ , we may construct an equivalent program  $P'$ , with the same length as  $P$ , which uses only registers in  $R$ .

# PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.

## PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?

# PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
  - a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).

# PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
  - a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).
  - b. There is no question of choice for the register  $r$  in the instruction  $r \leftarrow E$ , where  $E$  has some register operands.  $r$  has to be one of the registers occurring in  $E$ .

# PROOF OUTLINE

1. The relabeling algorithm should be consistent, that is, when a variable which is defined is relabeled, its use should also be relabeled.
2. Assume that we are renaming the registers in the instructions in order starting from the first instruction. At which points will there be a question of a choice of registers?
  - a. There is no question of choice for the registers on the RHS of an instruction. These had been decided at the point of their definitions (consistent relabeling).
  - b. There is no question of choice for the register  $r$  in the instruction  $r \leftarrow E$ , where  $E$  has some register operands.  $r$  has to be one of the registers occurring in  $E$ .
  - c. The only instructions involving a choice of registers are instructions of the form  $r \leftarrow E$ , where  $E$  has no register operands.

## PROOF OUTLINE

3. Since the width of  $P$  is  $w$ , the width of the instruction just before  $r \leftarrow E$  is at most  $w - 1$ . (Why?)

## PROOF OUTLINE

3. Since the width of  $P$  is  $w$ , the width of the instruction just before  $r \leftarrow E$  is at most  $w - 1$ . (Why?)
4. Therefore a register can always be found for  $r$  in the rearranged program  $P'$ .

## CONTIGUITY AND STRONG CONTIGUITY

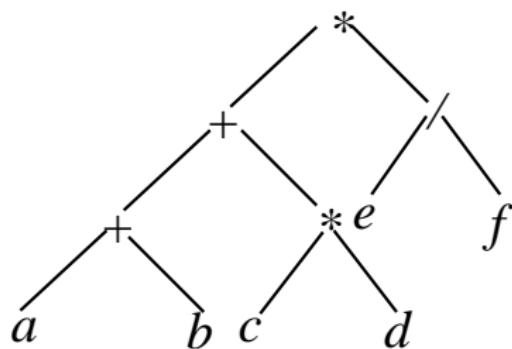
- ▶ Can one decrease the width of a program?

## CONTIGUITY AND STRONG CONTIGUITY

- ▶ Can one decrease the width of a program?
- ▶ For *storeless programs*, there is an arrangement which has minimum width.

# CONTIGUITY AND STRONG CONTIGUITY

- ▶ Can one decrease the width of a program?
- ▶ For *storeless programs*, there is an arrangement which has minimum width.
- ▶ EXAMPLE: All the three programs  $P_1$ ,  $P_2$ , and  $P_3$  compute the expression tree shown below:

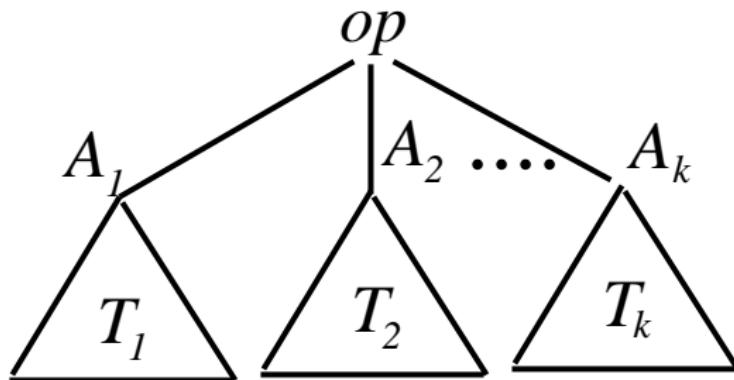


<u><math>P_1</math></u>	<u><math>P_2</math></u>	<u><math>P_3</math></u>
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

$P_1$	$P_2$	$P_3$
$r_1 \leftarrow a$	$r_1 \leftarrow a$	$r_1 \leftarrow a$
$r_2 \leftarrow b$	$r_2 \leftarrow b$	$r_2 \leftarrow b$
$r_3 \leftarrow c$	$r_3 \leftarrow c$	$r_1 \leftarrow r_1 + r_2$
$r_4 \leftarrow d$	$r_4 \leftarrow d$	$r_2 \leftarrow c$
$r_5 \leftarrow e$	$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow d$
$r_6 \leftarrow f$	$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow r_2 * r_3$
$r_5 \leftarrow r_5 / r_6$	$r_1 \leftarrow r_1 + r_3$	$r_1 \leftarrow r_1 + r_2$
$r_3 \leftarrow r_3 * r_4$	$r_2 \leftarrow e$	$r_2 \leftarrow e$
$r_1 \leftarrow r_1 + r_2$	$r_3 \leftarrow f$	$r_3 \leftarrow f$
$r_1 \leftarrow r_1 + r_3$	$r_2 \leftarrow r_2 / r_3$	$r_2 \leftarrow r_2 / r_3$
$r_1 \leftarrow r_1 * r_5$	$r_1 \leftarrow r_1 * r_2$	$r_1 \leftarrow r_1 * r_2$

The program  $P_2$  has a width less than  $P_1$ , whereas  $P_3$  has the least width of all three programs.  $P_2$  is a *contiguous* program whereas  $P_3$  is a *strongly contiguous* program.

# CONTIGUITY AND STRONG CONTIGUITY



# CONTIGUITY AND STRONG CONTIGUITY

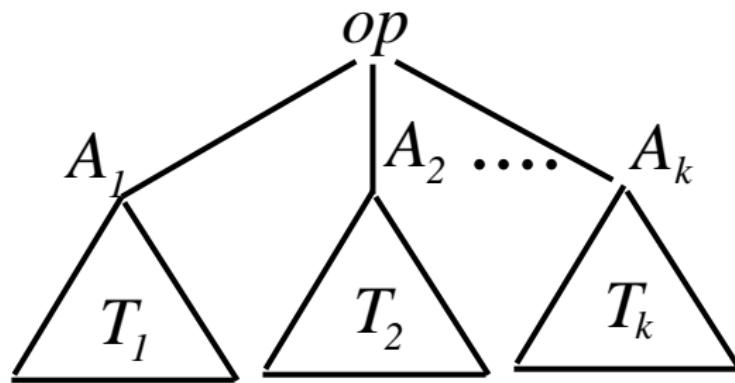
**THEOREM:** Let  $P = I_1, I_2, \dots, I_q$  be a program of width  $w$  with no stores.  $I_q$  uses  $k$  registers whose values at time  $q - 1$  are  $A_1, \dots, A_k$ . Then there exists an equivalent program

$Q = J_1, J_2, \dots, J_q$ , and a permutation  $\pi$  on  $\{1, \dots, k\}$  such that

- i.  $Q$  has width at most  $w$ .
- ii.  $Q$  can be written as  $P_1 \dots P_k J_q$  where  $v(P_i) = A_{\pi(i)}$  for  $1 \leq i \leq k$ , and the width of  $P_i$ , by itself, is at most  $w - i + 1$ .

# CONTIGUITY AND STRONG CONTIGUITY

Consider an evaluation of the expression tree::



This tree can be evaluated in the order mentioned below:

# CONTIGUOUS AND STRONG CONTIGUOUS EVALUATION

1. Q computes the entire subtree  $T_1$  first using  $P_1$ . In the process all the  $w$  registers could be used.
2. After computing  $T_1$  all registers except one are freed. Therefore  $T_2$  is free to use  $w - 1$  registers and its width is at most  $w - 1$ .  $T_2$  is computed by  $P_2$ .
3.  $T_3$  is similarly computed by  $P_3$ , whose width is  $w - 2$ .

Of course  $A_1, \dots, A_3$  need not necessarily be computed in this order. This is what brings the permutation  $\pi$  in the statement of the theorem.

# CONTIGUOUS AND STRONG CONTIGUOUS EVALUATION

A program in the form  $P_1 \dots P_k J_q$  is said to be in *contiguous form*. If each of the  $P_i$ s is, in turn, contiguous, then the program is said to be in *strong contiguous form*.

**THEOREM:** Every program *without stores* can be transformed into strongly contiguous form.

**PROOF OUTLINE:** Apply the technique in the previous theorem recursively to each of the  $P_i$ s.

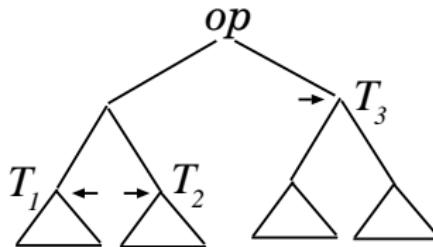
# AHO-JOHNSON ALGORITHM

## STRONG NORMAL FORM PROGRAMS

A program requires stores if there are not enough registers to hold intermediate values or if an instruction requires some of its operands to be in memory locations. Such programs can also be cast in a certain form called *strong normal form*.

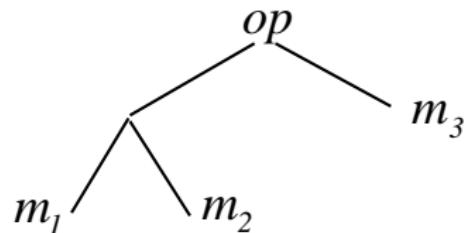
## AHO-JOHNSON ALGORITHM

Consider the following evaluation of tree shown, in which the marked nodes require stores.



1. Compute  $T_1$  using program  $P_1$ . Store the value in memory location  $m_1$ .
2. Compute  $T_2$  using program  $P_2$ . Store the value in memory location  $m_2$ .
3. Compute  $T_3$  using program  $P_3$ . Store the value in memory location  $m_3$ .
4. Compute the tree shown below using a storeless program  $P_4$ .

# AHO-JOHNSON ALGORITHM



A program in such a form is called a *normal form program*.

# AHO-JOHNSON ALGORITHM

Let  $P = I_1 \dots I_q$  be a machine program. We say  $P$  is in *normal form*, if it can be written as  $P = P_1 J_1 P_2 J_2 \dots P_{s-1} J_{s-1} P_s$ , such that

1. Each  $J_i$  is a store instruction and no  $P_i$  contains a store instruction.
2. No registers are active immediately after a store instruction.

Further,  $P$  is in *strong normal form*, if each  $P_i$  is strongly contiguous.

## AHO-JOHNSON ALGORITHM

LEMMA: Let  $P$  be an optimal program which computes an expression tree. Then there exists a permutation of  $P$ , which computes the same value and is in normal form.

# AHO-JOHNSON ALGORITHM

LEMMA: Let  $P$  be an optimal program which computes an expression tree. Then there exists a permutation of  $P$ , which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let  $I_f$  be the first store instruction of  $P$ .

# AHO-JOHNSON ALGORITHM

LEMMA: Let  $P$  be an optimal program which computes an expression tree. Then there exists a permutation of  $P$ , which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let  $I_f$  be the first store instruction of  $P$ .
2. Identify the instructions between  $I_1$  and  $I_{f-1}$  which do not contribute towards the computation of the value of  $I_f$ .

# AHO-JOHNSON ALGORITHM

LEMMA: Let  $P$  be an optimal program which computes an expression tree. Then there exists a permutation of  $P$ , which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let  $I_f$  be the first store instruction of  $P$ .
2. Identify the instructions between  $I_1$  and  $I_{f-1}$  which do not contribute towards the computation of the value of  $I_f$ .
3. Shift these instructions, in order, after  $I_f$ .

# AHO-JOHNSON ALGORITHM

LEMMA: Let  $P$  be an optimal program which computes an expression tree. Then there exists a permutation of  $P$ , which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let  $I_f$  be the first store instruction of  $P$ .
2. Identify the instructions between  $I_1$  and  $I_{f-1}$  which do not contribute towards the computation of the value of  $I_f$ .
3. Shift these instructions, in order, after  $I_f$ .
4. We now have a program  $P_1 J_1 Q$ , where  $P_1$  is storeless,  $J_1$  is the first store instruction (previously denoted by  $I_f$ ), and no registers are active after  $J_1$ .

# AHO-JOHNSON ALGORITHM

LEMMA: Let  $P$  be an optimal program which computes an expression tree. Then there exists a permutation of  $P$ , which computes the same value and is in normal form.

PROOF OUTLINE:

1. Let  $I_f$  be the first store instruction of  $P$ .
2. Identify the instructions between  $I_1$  and  $I_{f-1}$  which do not contribute towards the computation of the value of  $I_f$ .
3. Shift these instructions, in order, after  $I_f$ .
4. We now have a program  $P_1 J_1 Q$ , where  $P_1$  is storeless,  $J_1$  is the first store instruction (previously denoted by  $I_f$ ), and no registers are active after  $J_1$ .
5. Repeat this for the program  $Q$ .

## AHO-JOHNSON ALGORITHM

THEOREM: Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

## AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.

## AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.
2.  $Q$  has width at most  $w$ , and

# AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.
2.  $Q$  has width at most  $w$ , and
3.  $Q$  is in strong normal form.

# AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.
2.  $Q$  has width at most  $w$ , and
3.  $Q$  is in strong normal form.

# AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.
2.  $Q$  has width at most  $w$ , and
3.  $Q$  is in strong normal form.

**PROOF OUTLINE:**

1. Given a program, first apply the previous lemma to get a program in normal form.

# AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.
2.  $Q$  has width at most  $w$ , and
3.  $Q$  is in strong normal form.

**PROOF OUTLINE:**

1. Given a program, first apply the previous lemma to get a program in normal form.
2. Convert each  $P_i$  to strongly contiguous form.

# AHO-JOHNSON ALGORITHM

**THEOREM:** Let  $P$  be an optimal program of width  $w$ . We can transform  $P$  into an equivalent program  $Q$  such that:

1.  $P$  and  $Q$  have the same length.
2.  $Q$  has width at most  $w$ , and
3.  $Q$  is in strong normal form.

**PROOF OUTLINE:**

1. Given a program, first apply the previous lemma to get a program in normal form.
2. Convert each  $P_i$  to strongly contiguous form.
3. None of the above transformations increase the width or length of the program.

# AHO-JOHNSON ALGORITHM

## OPTIMALITY CONDITION

Not all programs in strong normal form are optimal. We need to specify under what conditions is a program in strong normal form optimal. This will allow us later to prove the optimality of our code generation algorithm.

# AHO-JOHNSON ALGORITHM

## OPTIMALITY CONDITION

Not all programs in strong normal form are optimal. We need to specify under what conditions is a program in strong normal form optimal. This will allow us later to prove the optimality of our code generation algorithm.

1. If an expression tree can be evaluated without stores, then the optimal program will do so. Moreover it will use minimal number of instructions for this purpose.

# AHO-JOHNSON ALGORITHM

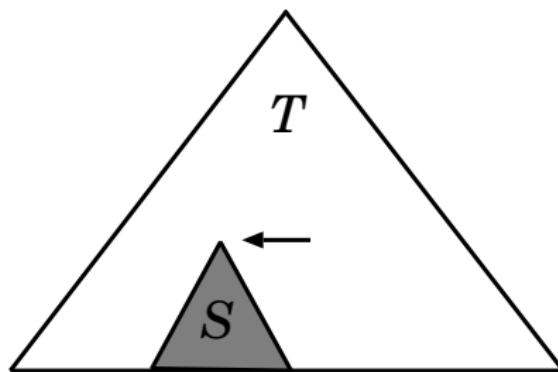
## OPTIMALITY CONDITION

Not all programs in strong normal form are optimal. We need to specify under what conditions is a program in strong normal form optimal. This will allow us later to prove the optimality of our code generation algorithm.

1. If an expression tree can be evaluated without stores, then the optimal program will do so. Moreover it will use minimal number of instructions for this purpose.
2. Now assume that a program necessarily requires stores at certain points of the tree, as shown next. For simplicity, assume that this is the only store required to evaluate the tree.

# AHO-JOHNSON ALGORITHM

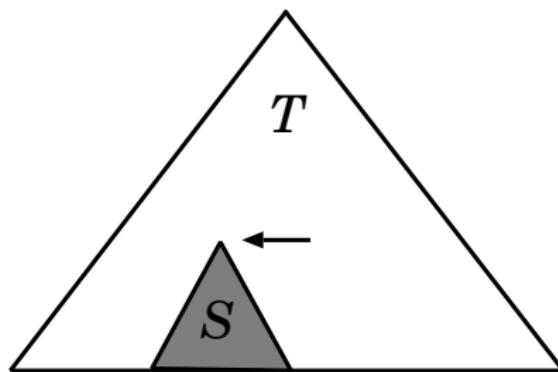
## OPTIMALITY CONDITION



3. then the optimal program should

# AHO-JOHNSON ALGORITHM

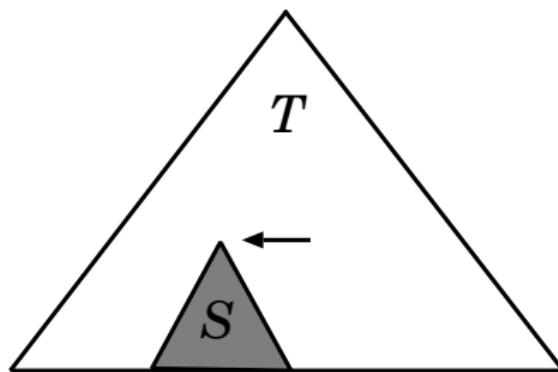
## OPTIMALITY CONDITION



3. then the optimal program should
  - a. Evaluate  $S$  (optimally, by condition 1).

# AHO-JOHNSON ALGORITHM

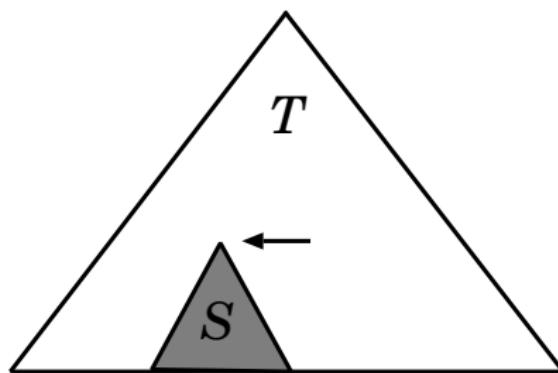
## OPTIMALITY CONDITION



3. then the optimal program should
  - a. Evaluate  $S$  (optimally, by condition 1).
  - b. Store the value in a memory location.

# AHO-JOHNSON ALGORITHM

## OPTIMALITY CONDITION



3. then the optimal program should
  - a. Evaluate  $S$  (optimally, by condition 1).
  - b. Store the value in a memory location.
  - c. Evaluate the rest of the (storeless) tree  $T/S$  (once again optimally, due to condition 1).

# AHO-JOHNSON ALGORITHM

## THE ALGORITHM

The algorithm makes three passes over the expression tree.

**Pass 1** Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.

# AHO-JOHNSON ALGORITHM

## THE ALGORITHM

The algorithm makes three passes over the expression tree.

- Pass 1** Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.
- Pass 2** Identifies the subtrees which must be evaluated in memory locations.

# AHO-JOHNSON ALGORITHM

## THE ALGORITHM

The algorithm makes three passes over the expression tree.

- Pass 1 Computes an array of costs for each node. This helps to select an instruction to evaluate the node, and the evaluation order to evaluate the subtrees of the node.
- Pass 2 Identifies the subtrees which must be evaluated in memory locations.
- Pass 3 Actually generates code.

## AHO-JOHNSON ALGORITHM: COVER

- ▶ An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.

## AHO-JOHNSON ALGORITHM: COVER

- ▶ An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.
- ▶ The algorithm which decides whether an instruction covers a node also provides a related information

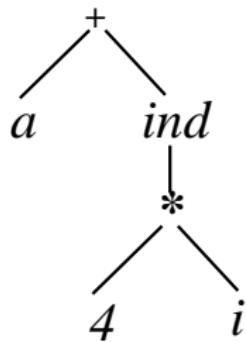
## AHO-JOHNSON ALGORITHM: COVER

- ▶ An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.
- ▶ The algorithm which decides whether an instruction covers a node also provides a related information
  - ▶ which of the subtrees of the node should be evaluated in registers (regset)

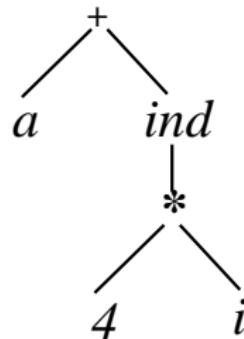
## AHO-JOHNSON ALGORITHM: COVER

- ▶ An instruction *covers a node* in an expression tree, if it can be used to evaluate the node.
- ▶ The algorithm which decides whether an instruction covers a node also provides a related information
  - ▶ which of the subtrees of the node should be evaluated in registers (regset)
  - ▶ which should be evaluated in memory locations (memset).

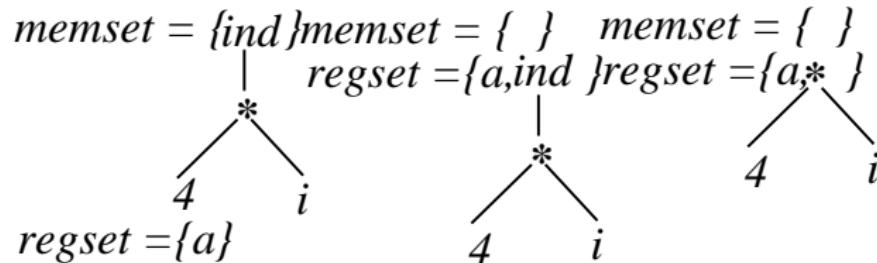
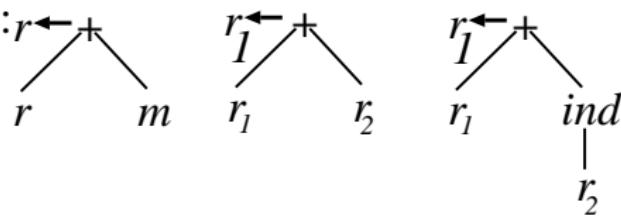
## EXAMPLE



# EXAMPLE



Instruction:



# ALGORITHM FOR COVER

```
function cover( $E, S$ );
```

(\* decides whether  $z \leftarrow E$  covers the expression tree  $S$ . If so, then  
*regset* and *memset* will contain the subtrees of  $S$  to be evaluated  
in register and memory \*)

# ALGORITHM FOR COVER

```
function cover( $E, S$ );
```

(\* decides whether  $z \leftarrow E$  covers the expression tree  $S$ . If so, then *regset* and *memset* will contain the subtrees of  $S$  to be evaluated in register and memory \*)

1. If  $E$  is a single register node, add  $S$  to *regset* and return *true*.

# ALGORITHM FOR COVER

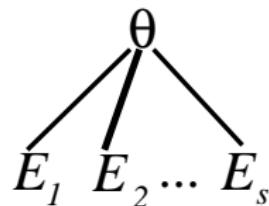
```
function cover( $E, S$ );
```

(\* decides whether  $z \leftarrow E$  covers the expression tree  $S$ . If so, then *regset* and *memset* will contain the subtrees of  $S$  to be evaluated in register and memory \*)

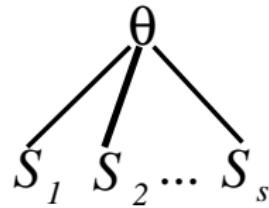
1. If  $E$  is a single register node, add  $S$  to *regset* and return *true*.
2. If  $E$  is a single memory node, add  $S$  to *memset* and return *true*.

# ALGORITHM FOR COVER

3. If  $E$  has the form



then, if the root of  $S$  is not  $\theta$ , return *false*. Else, write  $S$  as



For all  $i$  from 1 to  $s$  do  $\text{cover}(E_i, S_i)$ . Return *true*, only if all invocations return *true*.

## AHO-JOHNSON ALGORITHM

Calculates an array of costs  $C_j(S)$  for every subtree  $S$  of  $T$ , whose meaning is to be interpreted as follows:

# AHO-JOHNSON ALGORITHM

Calculates an array of costs  $C_j(S)$  for every subtree  $S$  of  $T$ , whose meaning is to be interpreted as follows:

- ▶  $C_0(S)$  : cost of evaluating  $S$  in a memory location.

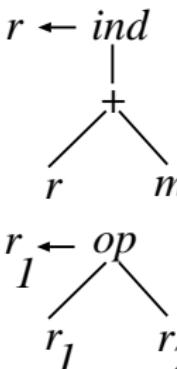
## AHO-JOHNSON ALGORITHM

Calculates an array of costs  $C_j(S)$  for every subtree  $S$  of  $T$ , whose meaning is to be interpreted as follows:

- ▶  $C_0(S)$  : cost of evaluating  $S$  in a memory location.
- ▶  $C_j(S), j \neq 0$  is the minimum cost of evaluating  $S$  using  $j$  registers.

## EXAMPLE

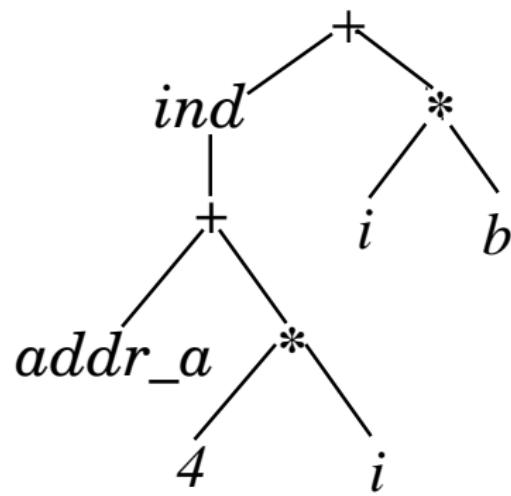
Consider a machine with the instructions shown below.

$r \leftarrow c$	{MOV #c, r}
$r \leftarrow m$	{MOV m, r}
$m \leftarrow r$	{MOV r, m}
$r \leftarrow ind$	{MOV m(r), r}
	
$r_1 \leftarrow op$	{op r <sub>2</sub> , r <sub>1</sub> }

Note that there are no instructions of the form  $op\ m,\ r$  OR  
 $op\ r,\ m$ .

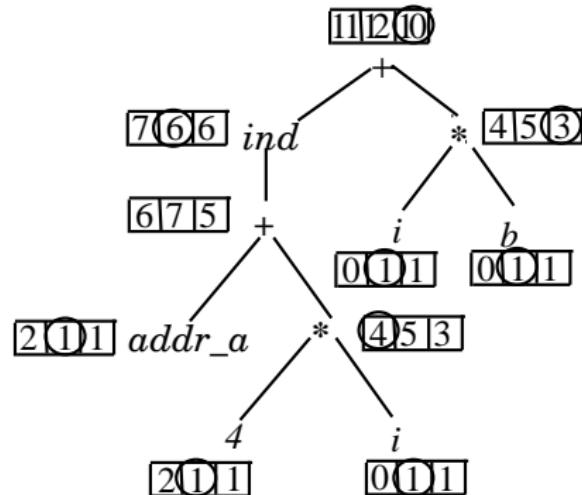
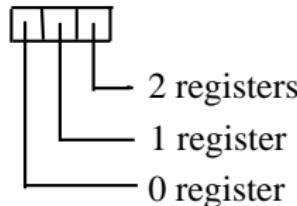
# AHO-JOHNSON ALGORITHM

Cost computation with 2 registers for the expression tree

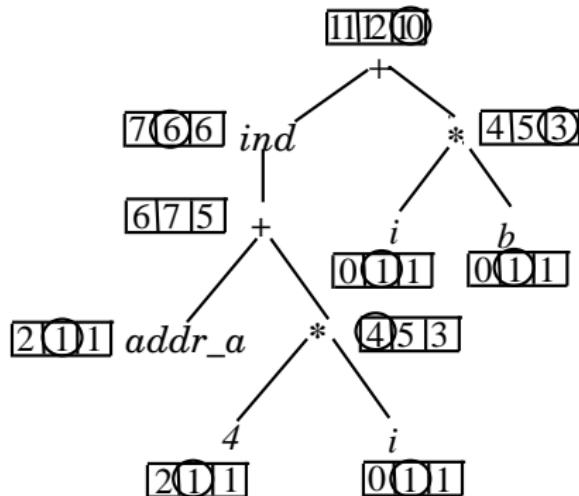
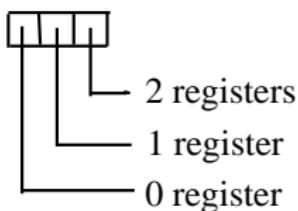


Assume that 4, being a literal, does not reside in memory.

# AHO-JOHNSON ALGORITHM



# AHO-JOHNSON ALGORITHM



In this example, we assume that 4, being a literal, does not reside in memory. The circles around the costs indicate the choices at the children which resulted in the circled cost of the parent. The next slide explains how to calculate the cost at each node.

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.

## AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.
3.  $C[0] = 2$ , load into a register, and then store in memory location.

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.
3.  $C[0] = 2$ , load into a register, and then store in memory location.

## AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.
3.  $C[0] = 2$ , load into a register, and then store in memory location.

For the leaf labeled  $i$ ,

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.
3.  $C[0] = 2$ , load into a register, and then store in memory location.

For the leaf labeled  $i$ ,

1.  $C[1] = 1$ , load the variable into a register.

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c, m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.
3.  $C[0] = 2$ , load into a register, and then store in memory location.

For the leaf labeled  $i$ ,

1.  $C[1] = 1$ , load the variable into a register.
2.  $C[2] = 1$ ,

# AHO-JOHNSON ALGORITHM

Consider the subtree  $4 * i$ . For the leaf labeled 4,

1.  $C[1] = 1$ , load the constant into a register using the MOVE  $c$ ,  $m$  instruction.
2.  $C[2] = 1$ , the extra register does not help.
3.  $C[0] = 2$ , load into a register, and then store in memory location.

For the leaf labeled  $i$ ,

1.  $C[1] = 1$ , load the variable into a register.
2.  $C[2] = 1$ ,
3.  $C[0] = 0$ , do nothing,  $i$  is already in a memory location.

# AHO-JOHNSON ALGORITHM

For the node labeled \*,

# AHO-JOHNSON ALGORITHM

For the node labeled \*,

1.  $C[2] = 3$ , evaluate each of the operands in registers and use the op  $r_1, r_2$  instruction.

# AHO-JOHNSON ALGORITHM

For the node labeled \*,

1.  $C[2] = 3$ , evaluate each of the operands in registers and use the op  $r_1, r_2$  instruction.
2.  $C[0] = 4$ , evaluate the node using two registers as above and store in a memory location.

# AHO-JOHNSON ALGORITHM

For the node labeled \*,

1.  $C[2] = 3$ , evaluate each of the operands in registers and use the op  $r_1, r_2$  instruction.
2.  $C[0] = 4$ , evaluate the node using two registers as above and store in a memory location.
3.  $C[1] =$

# AHO-JOHNSON ALGORITHM

For the node labeled \*,

1.  $C[2] = 3$ , evaluate each of the operands in registers and use the op  $r_1, r_2$  instruction.
2.  $C[0] = 4$ , evaluate the node using two registers as above and store in a memory location.
3.  $C[1] =$

# AHO-JOHNSON ALGORITHM

For the node labeled \*,

1.  $C[2] = 3$ , evaluate each of the operands in registers and use the op  $r_1, r_2$  instruction.
2.  $C[0] = 4$ , evaluate the node using two registers as above and store in a memory location.
3.  $C[1] = 5$ , notice that our machine has no op  $m, r$  instruction.

So we can use two registers to perform the operation and store the result in a memory location releasing the registers.

When we want to use the result, we can load it in a register.

The cost in this case is  $C[0] + 1 = 5$ .

## AHO-JOHNSON ALGORITHM

0. Let  $n$  denote the max number of available registers. Set  $C_j(s) = \infty$  for all subtrees  $S$  of  $T$  and for all  $j$ ,  $0 \leq j \leq n$ . Visit the tree in postorder. For each node  $S$  in the tree do steps 1–3.

## AHO-JOHNSON ALGORITHM

0. Let  $n$  denote the max number of available registers. Set  $C_j(s) = \infty$  for all subtrees  $S$  of  $T$  and for all  $j$ ,  $0 \leq j \leq n$ . Visit the tree in postorder. For each node  $S$  in the tree do steps 1–3.
1. If  $S$  is a leaf (variable), set  $C_0(S) = 0$ .

## AHO-JOHNSON ALGORITHM

0. Let  $n$  denote the max number of available registers. Set  $C_j(S) = \infty$  for all subtrees  $S$  of  $T$  and for all  $j$ ,  $0 \leq j \leq n$ . Visit the tree in postorder. For each node  $S$  in the tree do steps 1–3.
  1. If  $S$  is a leaf (variable), set  $C_0(S) = 0$ .
  2. Consider each instruction  $r \leftarrow E$  which covers  $S$ . For each instruction obtain the *regset*  $\{S_1, \dots, S_k\}$  and *memset*  $\{T_1, \dots, T_l\}$ . Then for each permutation  $\pi$  of  $\{1, \dots, k\}$  and for all  $j$ ,  $k \leq j \leq n$ , compute

$$C_j(S) = \min(C_j(S), \sum_{i=1}^k C_{j-i+1}(S_{\pi(i)}) + \sum_{i=1}^l C_0(T_i) + 1)$$

Remember the  $\pi$  that gives minimum  $C_j(S)$ .

## AHO-JOHNSON ALGORITHM

0. Let  $n$  denote the max number of available registers. Set  $C_j(S) = \infty$  for all subtrees  $S$  of  $T$  and for all  $j$ ,  $0 \leq j \leq n$ . Visit the tree in postorder. For each node  $S$  in the tree do steps 1–3.
  1. If  $S$  is a leaf (variable), set  $C_0(S) = 0$ .
  2. Consider each instruction  $r \leftarrow E$  which covers  $S$ . For each instruction obtain the *regset*  $\{S_1, \dots, S_k\}$  and *memset*  $\{T_1, \dots, T_l\}$ . Then for each permutation  $\pi$  of  $\{1, \dots, k\}$  and for all  $j$ ,  $k \leq j \leq n$ , compute
$$C_j(S) = \min(C_j(S), \sum_{i=1}^k C_{j-i+1}(S_{\pi(i)}) + \sum_{i=1}^l C_0(T_i) + 1)$$
Remember the  $\pi$  that gives minimum  $C_j(S)$ .
  3. Set  $C_0(S) = \min(C_0(S), C_n(S) + 1)$ , and
$$C_j(S) = \min(C_j(S), C_0(S) + 1).$$

# AHO-JOHNSON ALGORITHM: NOTES

1. In step 2,
  - ▶  $\sum_{i=1}^k C_{j-i+1}(S_{\pi(i)})$  is the cost of computing the subtrees  $S_i$  in registers,
  - ▶  $\sum_{i=1}^l C_0(T_i)$  is the cost of computing the subtrees  $T_i$  in memory,
  - ▶ 1 is the cost of the instruction at the root.
2.  $C_0(S) = \min(C_0(S), C_n(S) + 1)$  is the cost of evaluating a node in memory location by first using  $n$  registers and then storing it.

## AHO-JOHNSON ALGORITHM: NOTES

3.  $C_j(S) = \min(C_j(S), C_0(S) + 1)$  is the cost of evaluating a node by first evaluating it in a memory location and then loading it.
4. The algorithm also records at each node, the minimum cost, and
  - a. The instruction which resulted in the minimum cost.
  - b. The permutation which resulted in the minimum cost.

## AHO-JOHNSON ALGORITHM: PASS2

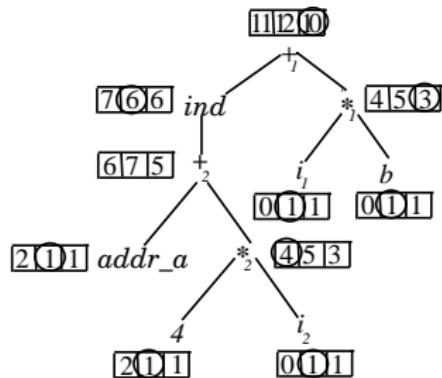
- ▶ This pass marks the nodes which have to be evaluated into memory.
- ▶ The algorithm is initially invoked as  $\text{mark}(T, n)$ , where  $T$  is the given expression tree and  $n$  the number of registers supported by the machine.
- ▶ It returns a sequence of nodes  $x_1, \dots, x_{s-1}$ , where  $x_1, \dots, x_{s-1}$  represent the nodes to be evaluated in memory. For purely technical reasons, after  $\text{mark}$  returns,  $x_s$  is set to  $T$  itself.

## function $mark(S, j)$

1. Let  $z \leftarrow E$  be the optimal instruction associated with  $C_j(S)$ , and  $\pi$  be the optimal permutation. Invoke  $cover(E, S)$  to obtain regset  $\{S_1, \dots, S_k\}$  and memset  $\{T_1, \dots, T_l\}$  of  $S$ .
2. For all  $i$  from 1 to  $k$  do  $mark(S_{\pi(i)}, j - i + 1)$ .
3. For all  $i$  from 1 to  $l$  do  $mark(T_i, n)$ .
4. If  $j$  is  $n$  and the instruction  $z \leftarrow E$  is a store, increment  $s$  and set  $x_s$  to the root of  $S$ .
5. Return.

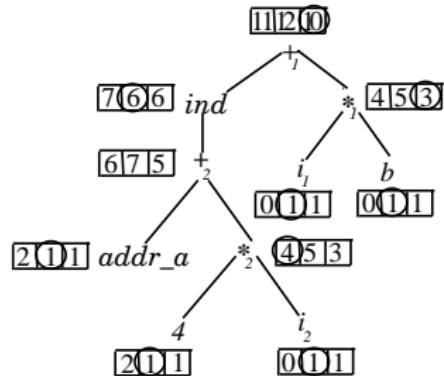
# AHO-JOHNSON ALGORITHM

$mark(+_1, 2)$

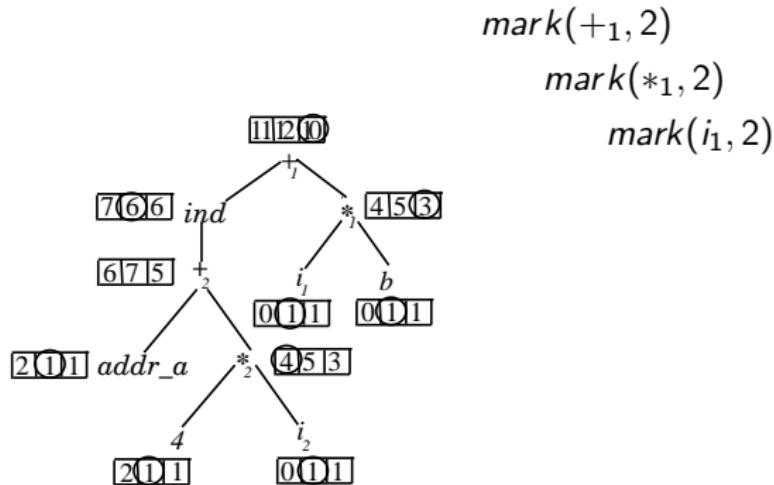


# AHO-JOHNSON ALGORITHM

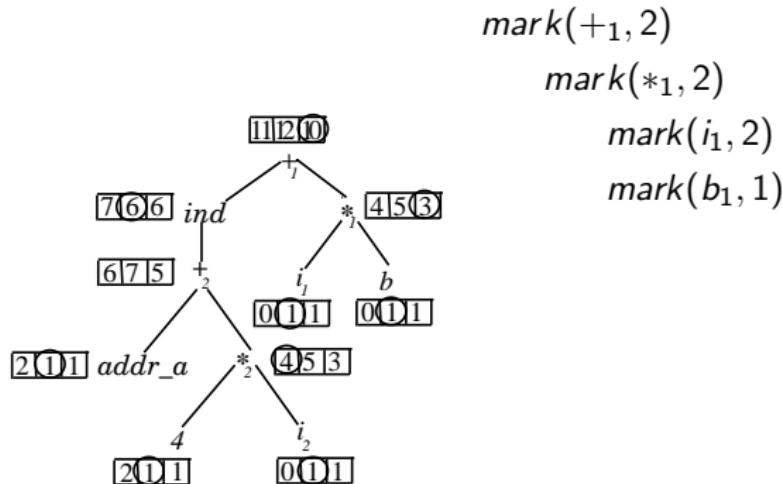
$mark(+_1, 2)$   
 $mark(*_1, 2)$



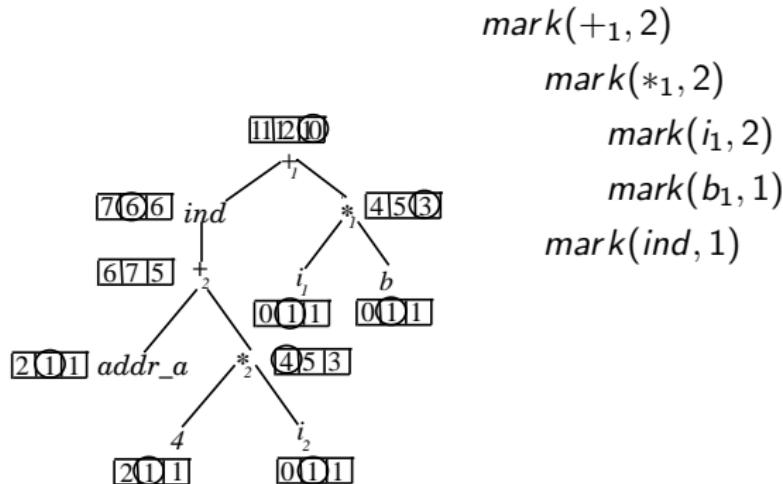
# AHO-JOHNSON ALGORITHM



# AHO-JOHNSON ALGORITHM



# AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

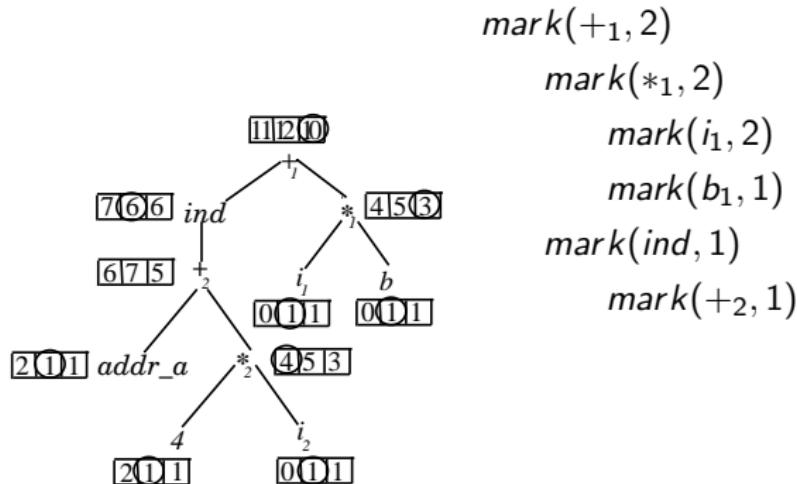
$mark(*_1, 2)$

$mark(i_1, 2)$

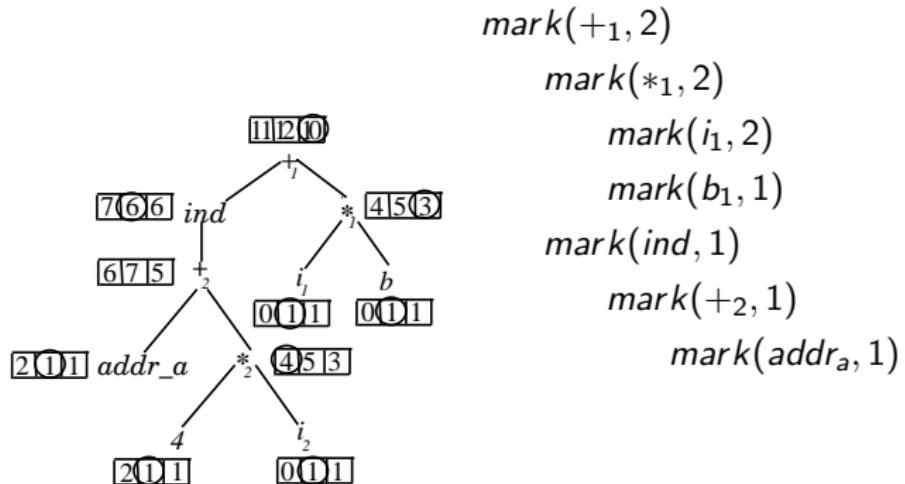
$mark(b_1, 1)$

$mark(ind, 1)$

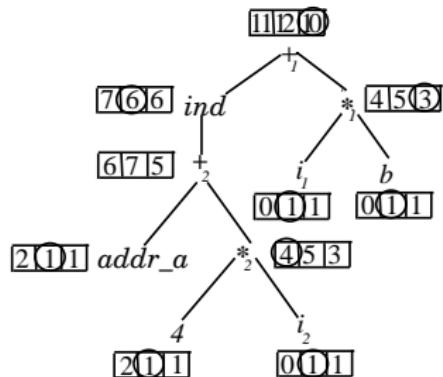
# AHO-JOHNSON ALGORITHM



# AHO-JOHNSON ALGORITHM



# AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

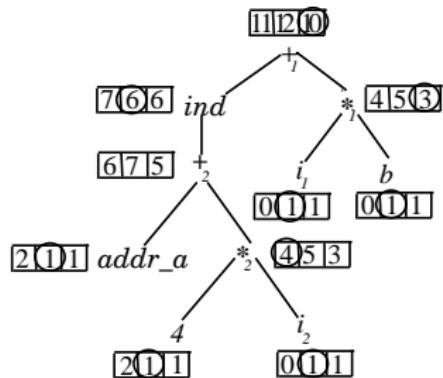
$mark(+_2, 1)$

$mark(addr_a, 1)$

$mark(*_2, 2)$  //the covering

//instruction is  $m \leftarrow \dots$

# AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

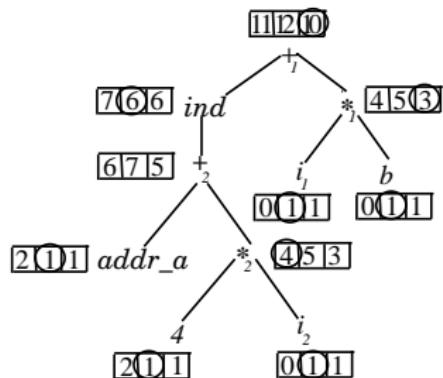
$mark(addr_a, 1)$

$mark(*_2, 2)$  //the covering

//instruction is  $m \leftarrow \dots$

$mark(4, 2)$

# AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

$mark(addr_a, 1)$

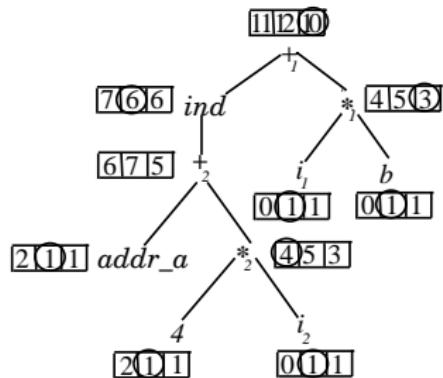
$mark(*_2, 2)$  //the covering

//instruction is  $m \leftarrow \dots$

$mark(4, 2)$

$mark(i_2, 1)$

# AHO-JOHNSON ALGORITHM



$mark(+_1, 2)$

$mark(*_1, 2)$

$mark(i_1, 2)$

$mark(b_1, 1)$

$mark(ind, 1)$

$mark(+_2, 1)$

$mark(addr_a, 1)$

$mark(*_2, 2)$  //the covering

//instruction is  $m \leftarrow \dots$

$mark(4, 2)$

$mark(i_2, 1)$

$x_1 = *_2$  //  $*_2$  needs to be stored

## AHO-JOHNSON ALGORITHM: PASS 3

- ▶ The algorithm generates code for the subtrees rooted at  $x_1, \dots, x_s$ , in that order.

## AHO-JOHNSON ALGORITHM: PASS 3

- ▶ The algorithm generates code for the subtrees rooted at  $x_1, \dots, x_s$ , in that order.
- ▶ After generating code for  $x_i$ , the algorithm replaces the node with a distinct memory location  $m_i$ .

## AHO-JOHNSON ALGORITHM: PASS 3

- ▶ The algorithm generates code for the subtrees rooted at  $x_1, \dots, x_s$ , in that order.
- ▶ After generating code for  $x_i$ , the algorithm replaces the node with a distinct memory location  $m_i$ .
- ▶ The algorithm uses the following unspecified routines

## AHO-JOHNSON ALGORITHM: PASS 3

- ▶ The algorithm generates code for the subtrees rooted at  $x_1, \dots, x_s$ , in that order.
- ▶ After generating code for  $x_i$ , the algorithm replaces the node with a distinct memory location  $m_i$ .
- ▶ The algorithm uses the following unspecified routines
  - ▶ *alloc* {\*allocates a register\*}

## AHO-JOHNSON ALGORITHM: PASS 3

- ▶ The algorithm generates code for the subtrees rooted at  $x_1, \dots, x_s$ , in that order.
- ▶ After generating code for  $x_i$ , the algorithm replaces the node with a distinct memory location  $m_i$ .
- ▶ The algorithm uses the following unspecified routines
  - ▶ *alloc* {\*allocates a register\*}
  - ▶ *free* {\*frees a register\*}

# AHO-JOHNSON ALGORITHM

The main program is:

1. Set  $i = 1$  and invoke  $\text{code}(x_i, n)$ . Let  $\alpha$  be the register returned. Issue the instruction  $m_i \leftarrow \alpha$ , invoke  $\text{free}(\alpha)$ , and rewrite  $x_i$  to represent  $m_i$ . Repeat this step for  $i = 2, \dots, s - 1$ .

# AHO-JOHNSON ALGORITHM

The main program is:

1. Set  $i = 1$  and invoke  $\text{code}(x_i, n)$ . Let  $\alpha$  be the register returned. Issue the instruction  $m_i \leftarrow \alpha$ , invoke  $\text{free}(\alpha)$ , and rewrite  $x_i$  to represent  $m_i$ . Repeat this step for  $i = 2, \dots, s - 1$ .
2. Invoke  $\text{code}(x_s, n)$ .

# AHO-JOHNSON ALGORITHM

The main program is:

1. Set  $i = 1$  and invoke  $\text{code}(x_i, n)$ . Let  $\alpha$  be the register returned. Issue the instruction  $m_i \leftarrow \alpha$ , invoke  $\text{free}(\alpha)$ , and rewrite  $x_i$  to represent  $m_i$ . Repeat this step for  $i = 2, \dots, s - 1$ .
2. Invoke  $\text{code}(x_s, n)$ .

# AHO-JOHNSON ALGORITHM

The main program is:

1. Set  $i = 1$  and invoke  $\text{code}(x_i, n)$ . Let  $\alpha$  be the register returned. Issue the instruction  $m_i \leftarrow \alpha$ , invoke  $\text{free}(\alpha)$ , and rewrite  $x_i$  to represent  $m_i$ . Repeat this step for  $i = 2, \dots, s - 1$ .
2. Invoke  $\text{code}(x_s, n)$ .

This uses the function  $\text{code}(S, j)$  which generates code for the tree  $S$  using  $j$  registers, and also returns the register in which the code was evaluated. This is described in the following slide.

function *code*( $S, j$ )

1. Let  $z \leftarrow E$  be the optimal instruction for  $C_j(S)$ , and  $\pi$  be the optimal permutation. Invoke *cover*( $E, S$ ) to obtain the regset  $\{S_1, \dots, S_k\}$ .

## function $code(S, j)$

1. Let  $z \leftarrow E$  be the optimal instruction for  $C_j(S)$ , and  $\pi$  be the optimal permutation. Invoke  $cover(E, S)$  to obtain the regset  $\{S_1, \dots, S_k\}$ .
2. For  $i = 1$  to  $k$ , do  $code(S_{\pi(i)}, j - i + 1)$ . Let  $\alpha_1, \dots, \alpha_k$  be the registers returned.

## function $code(S, j)$

1. Let  $z \leftarrow E$  be the optimal instruction for  $C_j(S)$ , and  $\pi$  be the optimal permutation. Invoke  $cover(E, S)$  to obtain the regset  $\{S_1, \dots, S_k\}$ .
2. For  $i = 1$  to  $k$ , do  $code(S_{\pi(i)}, j - i + 1)$ . Let  $\alpha_1, \dots, \alpha_k$  be the registers returned.
3. If  $k = 0$ , call alloc to obtain an unused register to return.

## function $\text{code}(S, j)$

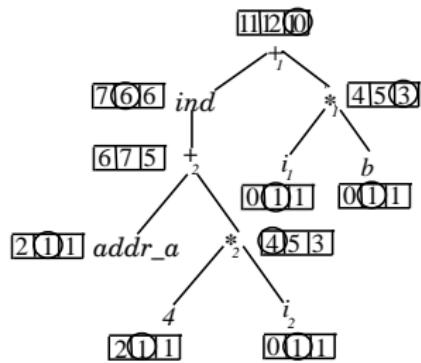
1. Let  $z \leftarrow E$  be the optimal instruction for  $C_j(S)$ , and  $\pi$  be the optimal permutation. Invoke  $\text{cover}(E, S)$  to obtain the regset  $\{S_1, \dots, S_k\}$ .
2. For  $i = 1$  to  $k$ , do  $\text{code}(S_{\pi(i)}, j - i + 1)$ . Let  $\alpha_1, \dots, \alpha_k$  be the registers returned.
3. If  $k = 0$ , call  $\text{alloc}$  to obtain an unused register to return.
4. Issue  $\alpha \leftarrow E$  with  $\alpha_1, \dots, \alpha_k$  substituted for the registers of  $E$ . Memory locations of  $E$  are substituted by some  $m_i$  or leaves of  $T$ .

## function $code(S, j)$

1. Let  $z \leftarrow E$  be the optimal instruction for  $C_j(S)$ , and  $\pi$  be the optimal permutation. Invoke  $cover(E, S)$  to obtain the regset  $\{S_1, \dots, S_k\}$ .
2. For  $i = 1$  to  $k$ , do  $code(S_{\pi(i)}, j - i + 1)$ . Let  $\alpha_1, \dots, \alpha_k$  be the registers returned.
3. If  $k = 0$ , call *alloc* to obtain an unused register to return.
4. Issue  $\alpha \leftarrow E$  with  $\alpha_1, \dots, \alpha_k$  substituted for the registers of  $E$ . Memory locations of  $E$  are substituted by some  $m_i$  or leaves of  $T$ .
5. Call *free* on  $\alpha_1, \dots, \alpha_k$  except  $\alpha$ . Return  $\alpha$  as the register for  $code(S, j)$ .

# AHO-JOHNSON ALGORITHM

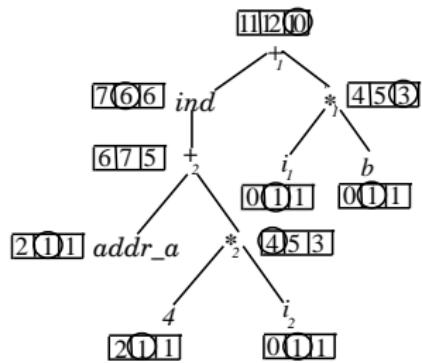
EXAMPLE: For the expression tree shown below, the code generated will be:



# AHO-JOHNSON ALGORITHM

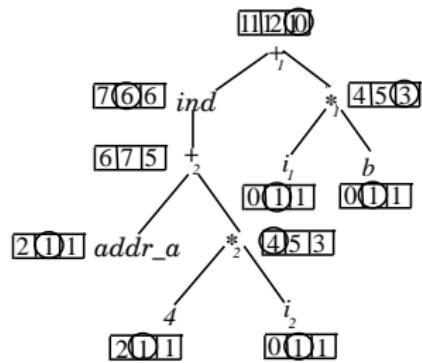
EXAMPLE: For the expression tree shown below, the code generated will be:

*MOVE #4, r<sub>1</sub> (evaluate 4 \* i first, since  
MOVE i, r<sub>2</sub> this node has to be stored)  
MUL r<sub>2</sub>, r<sub>1</sub>*



# AHO-JOHNSON ALGORITHM

EXAMPLE: For the expression tree shown below, the code generated will be:



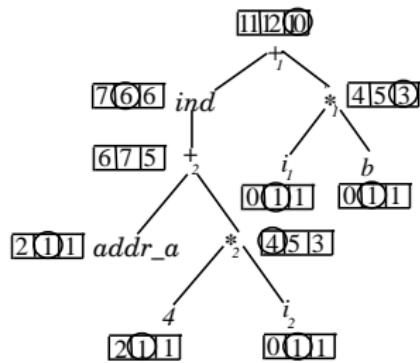
MOVE #4,  $r_1$  (evaluate  $4 * i$  first, since  
MOVE  $i$ ,  $r_2$  this node has to be stored)

MUL  $r_2$ ,  $r_1$

MOVE  $r_1$ ,  $m_1$

# AHO-JOHNSON ALGORITHM

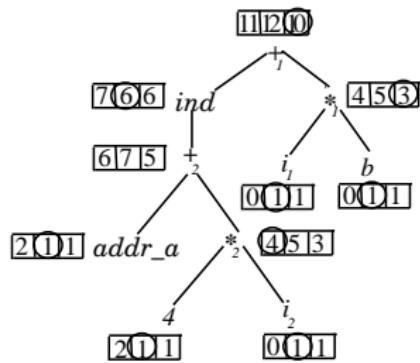
EXAMPLE: For the expression tree shown below, the code generated will be:



*MOVE #4,  $r_1$  (evaluate  $4 * i$  first, since  
 $MOVE i, r_2$  this node has to be stored)*  
*MUL  $r_2, r_1$*   
*MOVE  $r_1, m_1$*   
*MOVE  $i, r_1$  (evaluate  $i * b$  next, since this  
 $MOVE b, r_2$  requires 2 registers)*  
*MUL  $r_2, r_1$*

# AHO-JOHNSON ALGORITHM

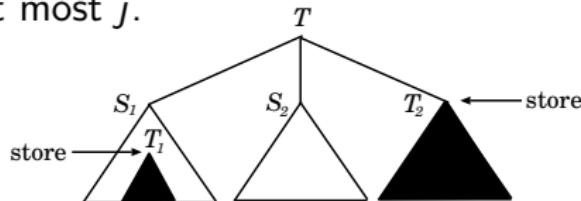
EXAMPLE: For the expression tree shown below, the code generated will be:



*MOVE #4, r<sub>1</sub>* (evaluate  $4 * i$  first, since  
*MOVE i, r<sub>2</sub>* this node has to be stored)  
*MUL r<sub>2</sub>, r<sub>1</sub>*  
*MOVE r<sub>1</sub>, m<sub>1</sub>*  
*MOVE i, r<sub>1</sub>* (evaluate  $i * b$  next, since this  
*MOVE b, r<sub>2</sub>* requires 2 registers)  
*MUL r<sub>2</sub>, r<sub>1</sub>*  
*MOVE #addr\_a, r<sub>1</sub>*  
*MOVE m<sub>1</sub>(r<sub>1</sub>), r<sub>1</sub>* (evaluate the *ind* node)  
*ADD r<sub>1</sub>, r<sub>2</sub>* (evaluate the root)

# PROOF OF OPTIMALITY

THEOREM:  $C_j(T)$  is the minimal cost over all strong normal form programs  $P_1J_1 \dots P_{s-1}J_{s-1}P_s$  which compute  $T$  such that the width of  $P_s$  is at most  $j$ .



- ▶ Consider an optimal program  $P_1J_1P_2J_2P_1$  in strong normal form.
- ▶ Now  $P$  is a strongly contiguous program which evaluates in registers values required by  $J$ . So  $P$  might be written as a sequence of contiguous programs, say  $P_3P_4$ .
- ▶ For instance,  $P_3$  could be the program computing the portion of  $S_1$  in figure the figure which is not shaded, using  $j$  registers, and  $P_4$  could be computing  $S_2$  using  $j - 1$  registers. Also  $P_1J_1$  and  $P_2J_2$  must be computing the shaded subtrees  $T_1$  and  $T_2$ .

## AHO-JOHNSON ALGORITHM

Now let us calculate the cost of this program.

- ▶  $P_1 J_1 P_3$  is a program in strong normal form, evaluating the subtree  $S_1$ . Since the width of  $P_3$  is  $j$ , as induction hypothesis we can assume that the cost of  $P_1 J_1 P_3$  is atleast  $C_j(S_1)$ .
- ▶  $P_4$  is also a program in strong normal form, evaluating  $S_2$  and the width of  $P_4$  is  $j - 1$ . Once again, as induction hypothesis, we can assume that the cost of  $P_4$  is atleast  $C_{j-1}(S_2)$ .
- ▶ Finally  $P_2 J_2$  is a program which computes the subtree  $T_2$  and stores it in memory. The cost of this is no more than  $C_0(T_2)$ .

Therefore the cost of this optimal program is

$1 + C_j(S_1) + C_{j-1}(S_2) + C_0(T_2)$ . The program generated by our algorithm is no costlier than this (Pass 1, step 2), and is therefore optimal.

# AHO-JOHNSON ALGORITHM

## COMPLEXITY OF THE ALGORITHM

1. The time required by Pass 1 is  $an$ , where  $a$  is a constant depending
  - ▶ linearly on the size of the instruction set
  - ▶ exponentially on the arity of the machine, and
  - ▶ linearly on the number of registers in the machineand  $n$  is the number of nodes in the expression tree.
2. Time required by Passes 2 and 3 is proportional to  $n$

Therefore the complexity of the algorithm is  $O(n)$ .