

# Com S 440/540

Principles and Practice of Compiling

Spring 2006

# Com S 440

Lecturer:

Dr. Markus Lumpe  
Department of Computer Science  
113 Atanasoff Hall  
<http://www.cs.iastate.edu/~cs440>  
TR 11 – 12:20, GILMAN 1104  
W 2:10 – 3 (440), PEARSON 1102

TA's:

N/A

Grading:

problem sets, midterm, project, written report

Assignments:

on a weekly basis

# Overview

Tentative course program:

- Lexical analysis and parsing
- Tool-based parser generation (JavaCC, yacc, lex)
- Abstract syntax representation
- Semantic analysis and symbol table organization
- Intermediate code translation
- Instruction selection and register allocation
- Target code generation
- Optimization of intermediate and target code
- Polymorphic type representation
- Garbage collection

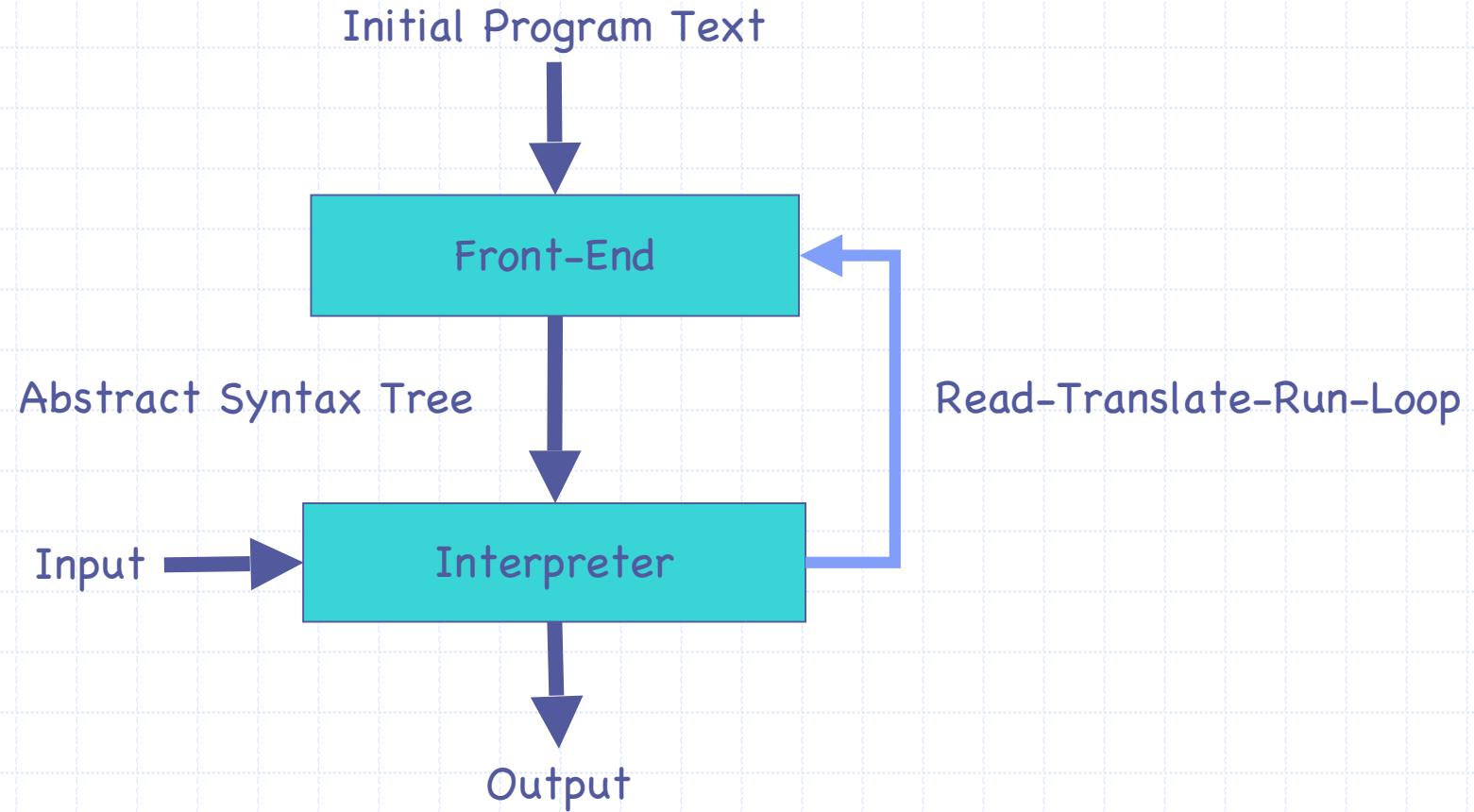
# A Language Interpreter

- An interpreter consists of two parts:
  - A front end that converts program text (a program in the source language) to a abstract syntax tree (the internal representation of the program text) and
  - An evaluator (the actual interpreter) that looks at a data structure and performs some associated actions, which depend on the actual data structure. In case of a language-processing system, the interpreter takes the abstract syntax tree and converts it, possibly using external inputs, to an answer.

## Examples:

- A calculator
- Basic
- Perl, Python, sh, awk, Tcl
- JVM

# Execution via Interpreter



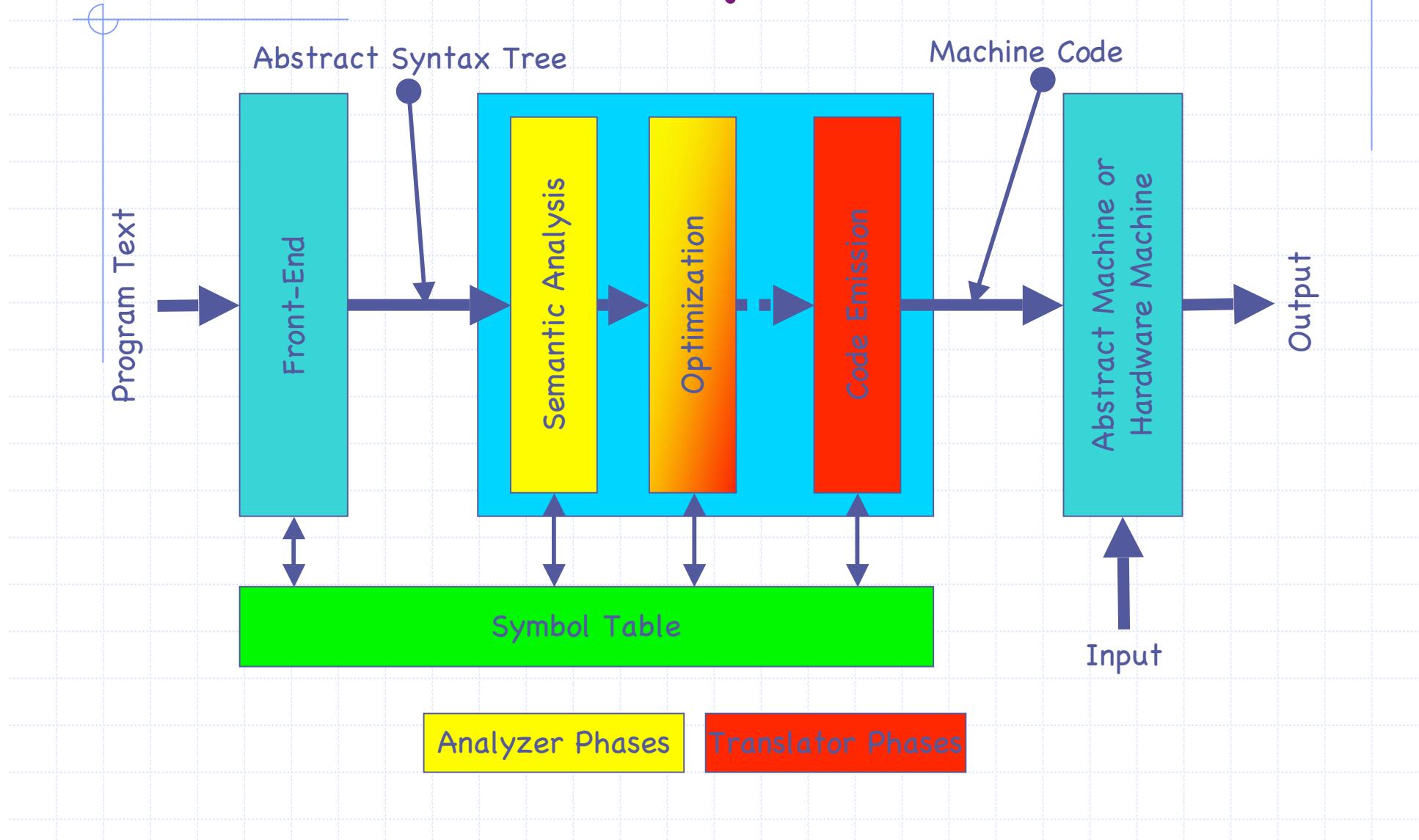
# A Language Compiler

- A compiler translates program text into some other language (the target language)
- The building blocks of a compiler are:
  - A front end that converts program text (a program in the source language) to a abstract syntax tree (the internal representation of the program text),
  - A set of independent compiler phases, each has assigned a particular task in the compilation process (e.g. semantics analysis, optimization, register allocation, code emission), and
  - The evaluator of a compiled languages may be an interpreter (e.g. JVM) or simply a hardware machine (e.g. von Neumann computer).

Examples of compiled languages:

- C/C++, C#
- Pascal, Java

# Execution via Compiler



# Programming Language Values

- In the specification of programming languages we have always at least two sets of values:
  - *Expressed values* – values that can be specified by means of (literal) expressions in the given programming language  
Examples: numbers, pairs, characters, strings
  - *Denoted values* – values that are bound to names  
Examples: variables, parameters, procedures

# Source, Host, and Target Language

- The **source language** is the language in which we write programs that should be evaluated by an interpreter or compiled by a compiler.
- The **host language** is the language in which we specify the interpreter or compiler.
- The **target language** is the language a source language is translated into by a compiler. A target language may be a higher-level programming language (e.g. C) or assembly language (or machine language).

# Warm-up Example

A straight-line programming language:

*Stm* ::= *Stm ; Stm*

| *Identifier* ::= *Exp*

| *print* ( *ExpList* )

*Exp* ::= *Identifier*

| *Integer*

| *Exp Binop Exp*

| ( *Stm , Exp* )

*ExpList* ::= *Exp , ExpList*

| *Exp*

*Binop* ::= +

| -

| \*

| /

# Lexical Issues

## ◆ Identifiers

- An *identifier* is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase.

## ◆ Integers

- A sequence of decimal digits is an *integer constant* that denotes the corresponding integer value.

## ◆ Comments

- A comment may appear between any two tokens. There are two forms of comments: One starts with /\*, end with \*/, and may be nested; another begins with // and goes to the end of the line.

# Sample Program

Input:

```
a := 5 + 3; b := (print (a, a-1), 10 * a); print (b)
```

Output:

```
8 7  
80
```

# Intermediate Representation

- We translate the grammar directly into tree data structure definitions. Each grammar symbol corresponds to an abstract class.

Grammar Symbol	(Abstract) Java Class
Stm	Statement
Exp	Expression
ExpList	ExpressionList
Identifier	String
Integer	int

# Java Specification

```
public abstract class Statement
{
    abstract void Eval( java.util.Hashtable aSymbolTable );
}

public abstract class Expression
{
    abstract Integer Eval( java.util.Hashtable aSymbolTable );
}

public abstract class ExpressionList
{
    abstract Integer[] Eval( java.util.Hashtable aSymbolTable
);
}
```

# Abstract Syntax Classes

// Statements

```
public class CompoundStatement extends Statement { ... }  
public class AssignmentStatement extends Statement { ... }  
public class PrintStatement extends Statement { ... }
```

// Expressions

```
public class VariableExpression extends Expression { ... }  
public class IntegerExpression extends Expression { ... }  
public final class Binops { ... }  
public class BinaryExpression extends Expression { ... }  
public class CommaExpression extends Expression { ... }
```

// ExpressionLists

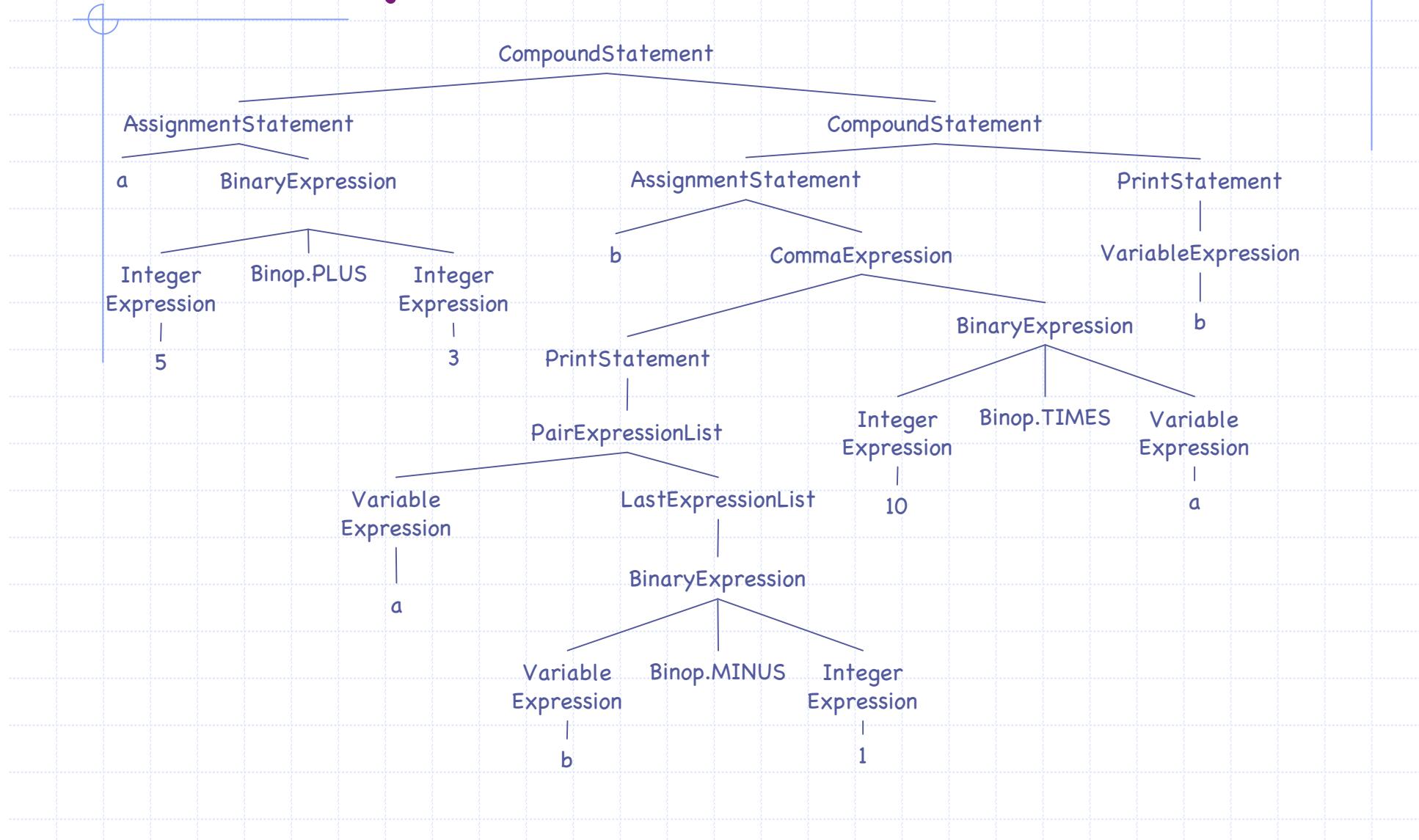
```
public class PairExpressionList extends ExpressionList { ... }  
public class LastExpressionList extends ExpressionList { ... }
```

# Constructing a Program

```
// a := 5 + 3; b := (print (a, a-1), 10 * a); print (b)
Statement s =
    new CompoundStatement(
        new AssignmentStatement( "a", new BinaryExpression( new IntegerExpression( new Integer( 5 ) ),
            new IntegerExpression( new Integer( 3 ) ),
            Binops.PLUS ) ),
        new CompoundStatement (
            new AssignmentStatement( "b",
                new CommaExpression(
                    new PrintStatement(
                        new PairExpressionList( new VariableExpression( "a" ),
                            new LastExpressionList(
                                new BinaryExpression( new VariableExpression( "a" ),
                                    new IntegerExpression( new
                                        Integer( 1 ) ),
                                Binops_MINUS ) ) ),
                            new BinaryExpression( new IntegerExpression( new Integer( 10 ) ),
                                new VariableExpression( "a" ),
                                Binops_TIMES ) ) ),
            new PrintStatement( new LastExpressionList( new VariableExpression( "b" ) ) ) );
    );

// evaluate s with an empty symbol table
s.Eval( new java.util.Hashtable() );
```

# Tree Representation



# Lexical Analysis

- The front end of a compiler/interpreter performs analysis, whereas the back end does synthesis.
- The analysis is usually broken up into:
  - Lexical analysis, which breaks the input into individual tokens,
  - Syntax analysis, which parses the structure of a program, and
  - Semantic analysis, which assigns the program a meaning.

# Syntax

- The syntax of a programming language is concerned with the form of programs, that is , how expressions, commands, declarations, etc., are put together to form programs.
- A well-designed programming language will have a well-designed syntax. However, the syntax definition given for a specific language is not power-full enough to define a programming language completely. The purpose of a well-defined syntax is to guide the programmer to understand the language's semantics.

# Semantics

- The semantics of a programming language is concerned with the meaning of programs, that is, how they behave when executed on computers.
- The semantics of a programming language assigns a precise meaning to every sentence of the language that can be formed using the given syntax definition. There are three approaches to define the semantics of a programming language:
  - Axiomatic semantics,
  - Operational semantics,
  - Denotational semantics.

# Lexical Tokens

- A lexical token is a sequence of characters, which are treated as a unit in the grammar of a programming language. A programming language classifies lexical tokens into a finite set of token types:

Type	Examples
Identifier	foo N14 symbol?
Integer	0 239495
Real	.0102 3.141 2E45 4.4E-67
Keywords	if while return class
Operators	+ - >>    :=
Punctuations	( ) . ; { }

# Specifying Token Recognition

- We can use any programming language to specify an ad hoc scanner. Consider, for example, the following Haskell code:

```
getNextToken :: String -> (Terminal, String)
getNextToken []          = (T_EOF, [])
getNextToken ('/'('/:cs) = getNextToken cs'      -- skip comment
             where cs' = dropWhile (\x -> x /= '\n') cs
getNextToken (c:cs)
| isSpace c = getNextToken cs                  -- skip white space
| isAlpha c || c == '_' = (checkKeyword (c:ns), cs')
| otherwise = getNextToken cs                 -- skip illegal character (no error)
             where nameChar x = (isAlphaNum x) || (x == '_')
                   (ns, cs') = ( takeWhile nameChar cs, dropWhile nameChar cs )

checkKeyword :: String -> Terminal
checkKeyword s
| s == "load" = (T_KEYWORD s)
| otherwise = (T_NAME s)\
```

However, we will specify lexical tokens using the formal language of *regular expressions* and implement token recognition using *finite deterministic automata* (DFA).

# Finite Deterministic Automata

A deterministic automaton is a quintuple  $(\Sigma, Q, q_0, \sigma, F)$  with:

- a set  $\Sigma$  of actions (sometimes called an alphabet),
- a set  $Q = \{q_0, q_1, \dots\}$  of states,
- a subset  $F$  of  $Q$  called the accepting states,
- a subset  $\sigma$  of  $Q \times \Sigma \times Q$  called the transitions,
- a designated start state  $q_0$ .

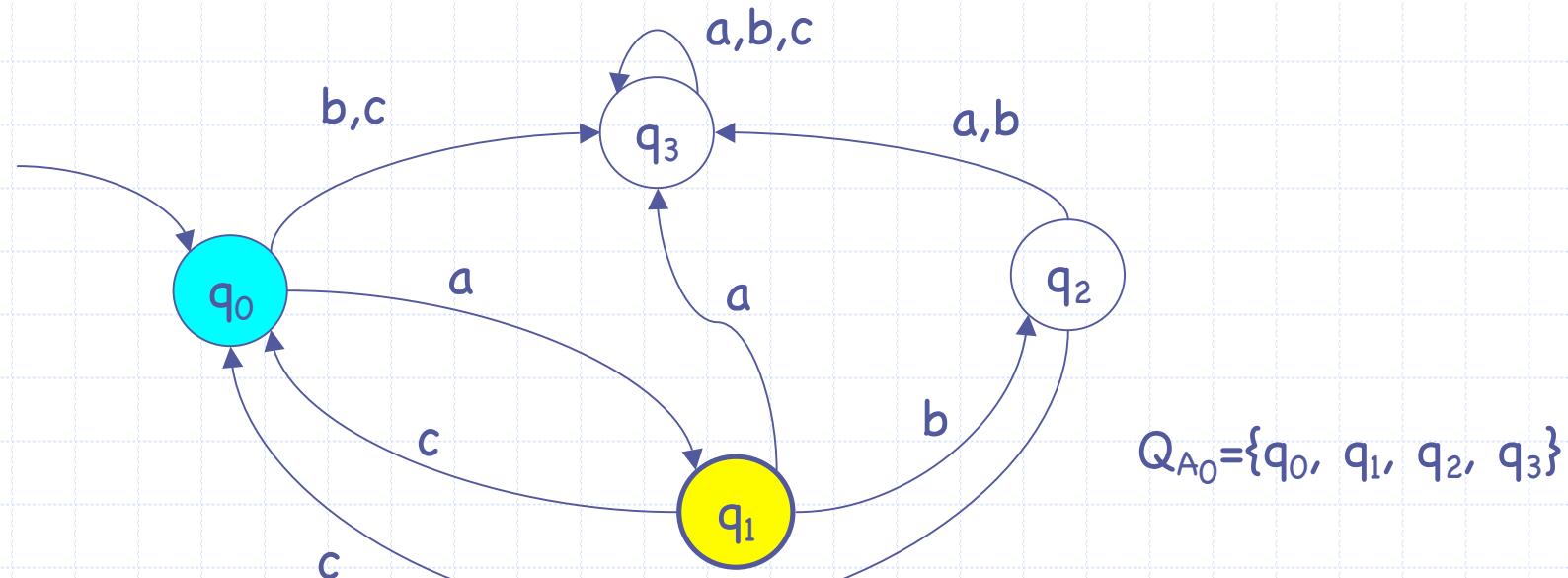
A transition  $(q, a, q') \in \sigma$  is usually written  $q \xrightarrow{a} q'$ .

The automaton  $A$  is said to be finite if  $Q$  is finite.

# Transition Graph

- An automaton is usually represented by a transition graph, whose nodes are states and whose arcs are transitions.

Example:  $A_0$  is a finite automaton over the alphabet  $\Sigma = \{a,b,c\}$ :



# Machine Table

- We can describe  $A_0$  formally also by writing  $A = (\Sigma, Q, q_0, \sigma, F)$ , where

- $\Sigma = \{a,b,c\}$ ,
- $Q = \{q_0, q_1, q_2, q_3\}$ ,
- $\sigma$  is described as

Q/S	a	b	c
$q_0$	$q_1$	$q_3$	$q_3$
$q_1$	$q_3$	$q_2$	$q_0$
$q_2$	$q_3$	$q_3$	$q_0$
$q_3$	$q_3$	$q_3$	$q_3$

- $q_0$  is the start state, and
- $F = \{ q_2 \}$

# Language of an Automaton

- A string over an alphabet  $\Sigma$  is a finite sequence of symbols from that alphabet, usually written to one another and not separated by any characters. If  $w$  is a string over  $\Sigma$ , the length of  $w$ , written  $|w|$ , is the number of symbols that it contains. The string of length zero is called the empty string and is denoted by  $\epsilon$ .
- Let  $A$  be an automaton over  $\Sigma$ , and  $s = a_1 \dots a_n$  a string over  $\Sigma$ . Then  $A$  is said to accept  $s$  if there is a path in  $A$ , from  $q_0$  to an accepting state, whose arcs are labeled successively  $a_1, \dots, a_n$ . The language of  $A$ , denoted by  $L(A)$ , is the set of strings accepted by  $A$ .
- The language  $L(A)$  of any finite-state automaton  $A$  is regular.

# Regular Sets

- Operations for building sets of strings:
  - Alternation
$$S_1 \mid S_2 = \{ s \mid s \in S_1 \vee s \in S_2 \}$$
  - Concatenation
$$S_1 \cdot S_2 = \{s_1s_2 \mid s_1 \in S_1, s_2 \in S_2\}$$
  - Iteration
$$\begin{aligned} S^* &= \{\epsilon\} \mid S \mid S \cdot S \mid S \cdot S \cdot S \mid \dots \\ &= S^0 \mid S^1 \mid S^2 \mid S^3 \mid \dots \end{aligned}$$
- A set of strings over  $\Sigma$  is said to be regular if it can be built from the empty set  $\emptyset$  and the singleton set  $\{a\}$  (for each  $a \in \Sigma$ ), using just the operations of alternation, concatenation, and iteration.

# Arden's Rule

- The following equations hold:

$$(S_1 \cdot S_2) \cdot S_3 = S_1 \cdot (S_2 \cdot S_3)$$

$$(S_1 | S_2) \cdot T = S_1 \cdot T | S_2 \cdot T$$

$$T \cdot (S_1 | S_2) = T \cdot S_1 | T \cdot S_2$$

$$S \cdot \varepsilon = S$$

$$S \cdot \emptyset = \emptyset$$

$$S \cdot (T \cdot S)^* = (S \cdot T)^* \cdot S$$

Note,  $\emptyset$  means "no path", whereas  $\varepsilon$  means "empty path".

## Arden's Rule:

For any two sets of strings  $S$  and  $T$ , the equation  $X = S \cdot X | T$  has  $X = S^* \cdot T$  as a solution. Moreover, this solution is unique if  $\varepsilon \notin S$ .

$$S \bullet (T \bullet S)^* = (S \bullet T)^* \bullet S$$

Proof by induction on the length of  $(T \bullet S)^*$ :

- Case 0:

$\Rightarrow$  We have  $S \bullet \varepsilon = S$

$\Leftarrow$  We have  $\varepsilon \bullet S = S$ . The result follows by a simple sub-induction to establish the fact that  $\varepsilon$  is a left-neutral element for  $\bullet$ .

- Case n:

We assume  $S \bullet (T \bullet S)^n = (S \bullet T)^n \bullet S$ .

We need to show that  $S \bullet (T \bullet S)^{n+1} = (S \bullet T)^{n+1} \bullet S$ .

$$\begin{aligned}
 & S \bullet (T \bullet S)^{n+1} \\
 &= S \bullet ((T \bullet S)^n \bullet (T \bullet S)) && \text{by def of } * \\
 &= (S \bullet (T \bullet S)^n) \bullet (T \bullet S) && \text{by assoc} \\
 &= ((S \bullet T)^n \bullet S) \bullet (T \bullet S) && \text{by assumption} \\
 &= (S \bullet T)^n \bullet (S \bullet (T \bullet S)) && \text{by assoc} \\
 &= (S \bullet T)^n \bullet ((S \bullet T) \bullet S) && \text{by assoc} \\
 &= ((S \bullet T)^n \bullet (S \bullet T)) \bullet S && \text{by assoc} \\
 &= (S \bullet T)^{n+1} \bullet S && \text{by def of } *
 \end{aligned}$$

Q.E.D.

# Monoid $(A^*, \cdot)$

A monoid is a pair  $(M, *)$ , that is, a set  $M$  with binary operation  $* : M \times M \rightarrow M$ , obeying the following axioms:

- Associativity: for all  $a, b, c$  in  $M$ ,  $(a * b) * c = a * (b * c)$
- Identity element: there exists an element  $e$  in  $M$ , such that for all  $a$  in  $M$ ,  $a * e = e * a = a$ .

$\Rightarrow (A^*, \cdot)$  is a monoid.

Moreover,  $(A^*, \cdot)$  is a free monoid, because the elements of  $A^*$  are all the finite sequences (or strings) of zero or more elements from  $A$ , with the binary operation of concatenation.

# Regular Expression Notation

a	An ordinary character stands for itself.
	Empty string (we write e)
M   N	Alternation, choosing from M or N.
M N	Concatenation, an M followed by an N.
M*	Repetition, zero or more times.
M+	Repetition, one or more times.
M?	Optional, zero or one occurrence of M.
[a-zA-Z]	Character set alternation.
.	A period stands for any single character (except newline).
"while"	Quotation, a string in quotes stands for itself.

# Examples of Regular Expressions

if

<IF>

[a-zA-Z][a-zA-Z0-9]\*

<IDENTIFIER>

[0-9]+

<INTEGER>

([0-9]+ \." [0-9]\*) | ([0-9]\* \." [0-9]+)

<REAL>

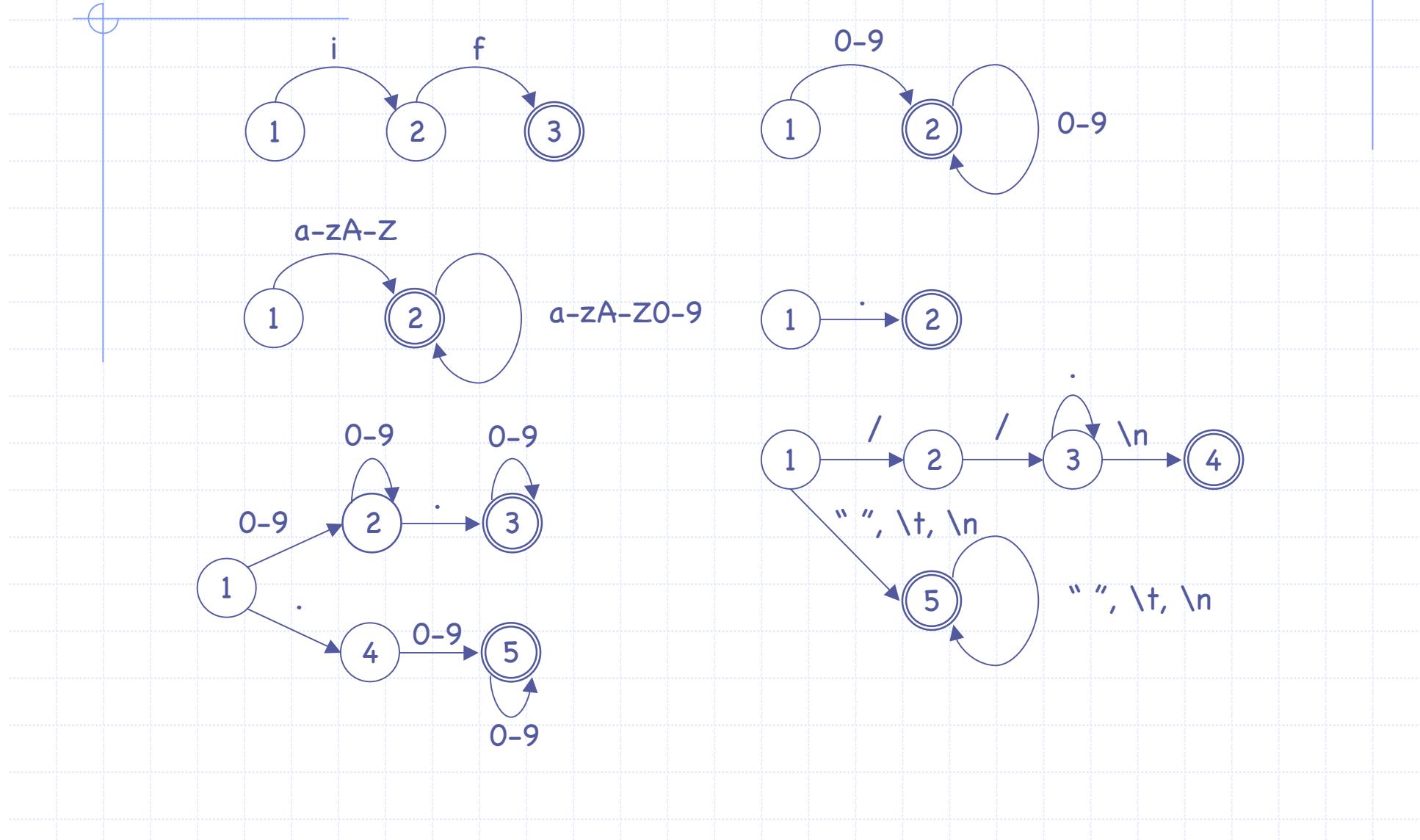
("//".\* "\n") | (" " | "\n" | "\t")

no token, just white space

.

error

# Finite Automata for Tokens



# Disambiguation Rules

- Longest match:

The longest initial substring of the input that can match any regular expression is taken as the next token.

"596.354" => Real, "85893a" => Integer (85893)

- Rule priority:

For a particular longest initial substring, the first regular expression that can match determines its token type.

"if" => Keyword if,

but only if we have defined the regular expression for keyword if before expression for identifier

# Finite Nondeterministic Automata

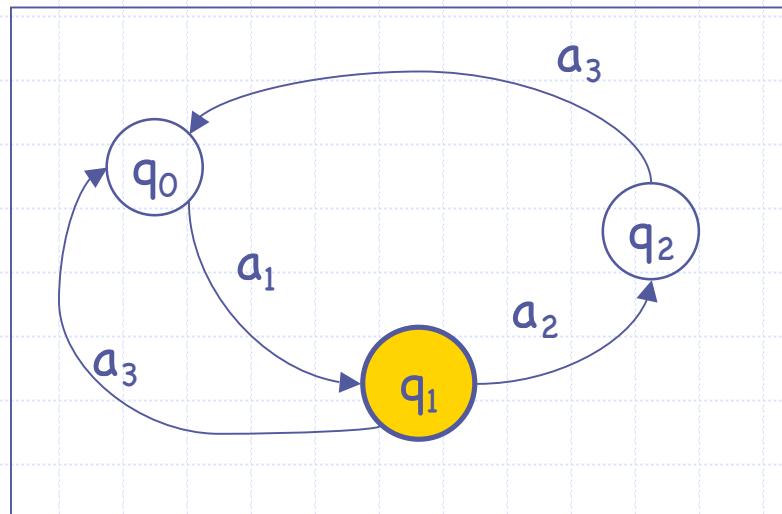
A finite nondeterministic automaton is a quintuple  $(\Sigma, Q, q_0, \sigma, F)$  with:

- a set  $\Sigma$  of *actions* (sometimes called an *alphabet*),
- a set  $Q = \{q_0, q_1, \dots\}$  of *states*,
- a subset  $F$  of  $Q$  called the *accepting states*,
- a subset  $\sigma$  of  $Q \times \Sigma \times P(Q)$  called the *transitions*,
- a designated start state  $q_0$ .

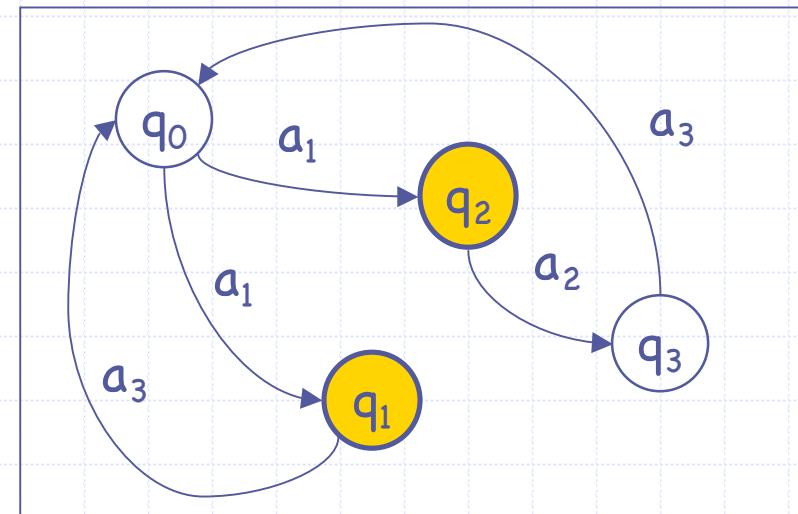
# DFA's versus NFA's

An automaton is deterministic if for each pair  $(q, a) \in Q \times \Sigma$  there is exactly one transition  $q \xrightarrow{a} q'$ .

deterministic automaton:

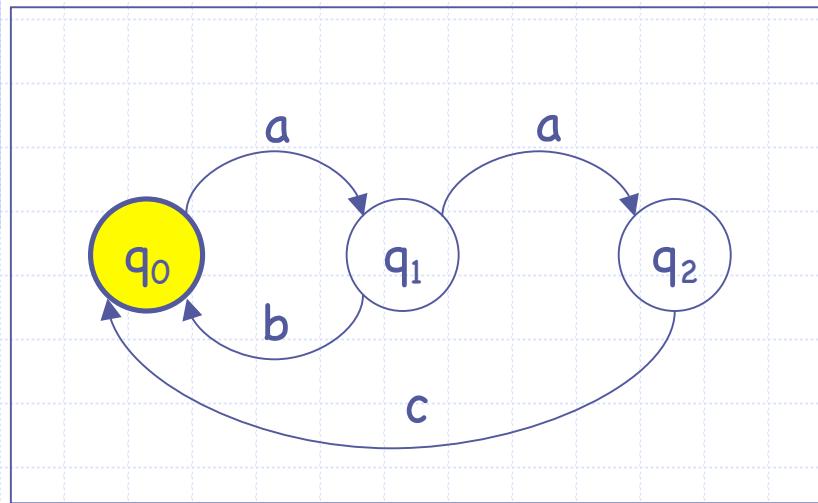


non-deterministic automaton:

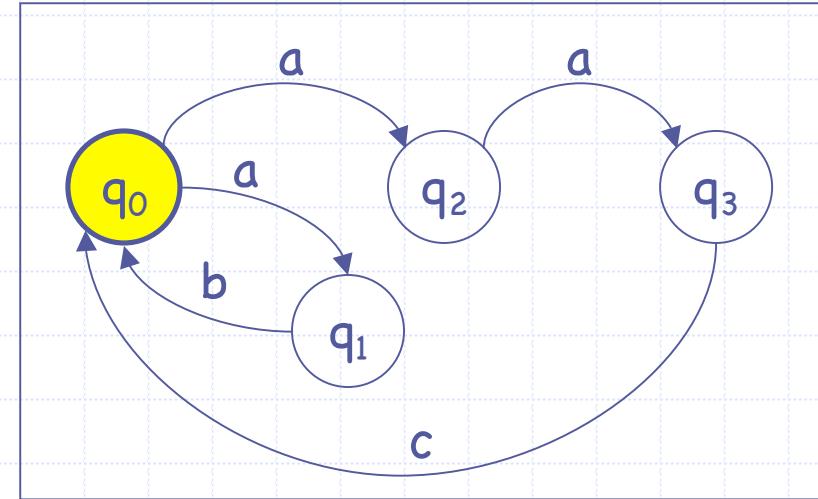


# Example

Deterministic system  $S_1$ :



Non-deterministic system  $S_2$ :



Are both systems equivalent?

# Machine Tables

Deterministic system  $S_1$ :

$Q/\Sigma$	a	b	c
$q_0$	$q_1$		
$q_1$	$q_2$	$q_0$	
$q_2$			$q_0$

Non-deterministic system  $S_2$ :

$Q/\Sigma$	a	b	c
$q_0$	$\{q_1, q_2\}$		
$q_1$			$q_0$
$q_2$		$q_3$	
$q_3$			$q_0$

# **S<sub>1</sub> = S<sub>2</sub>?**

S<sub>1</sub>:

$$\begin{aligned} q_0 &= a \cdot q_1 \mid e \\ q_1 &= b \cdot q_0 \mid a \cdot q_2 \\ q_2 &= c \cdot q_0 \end{aligned}$$

$$\begin{aligned} q_1 &= b \cdot q_0 \mid a \cdot c \cdot q_0 \\ q_0 &= a \cdot (b \cdot q_0 \mid a \cdot c \cdot q_0) \mid e \\ q_0 &= a \cdot (b \mid a \cdot c) \cdot q_0 \mid e \\ q_0 &= (a \cdot (b \mid a \cdot c))^* \end{aligned}$$

S<sub>2</sub>:

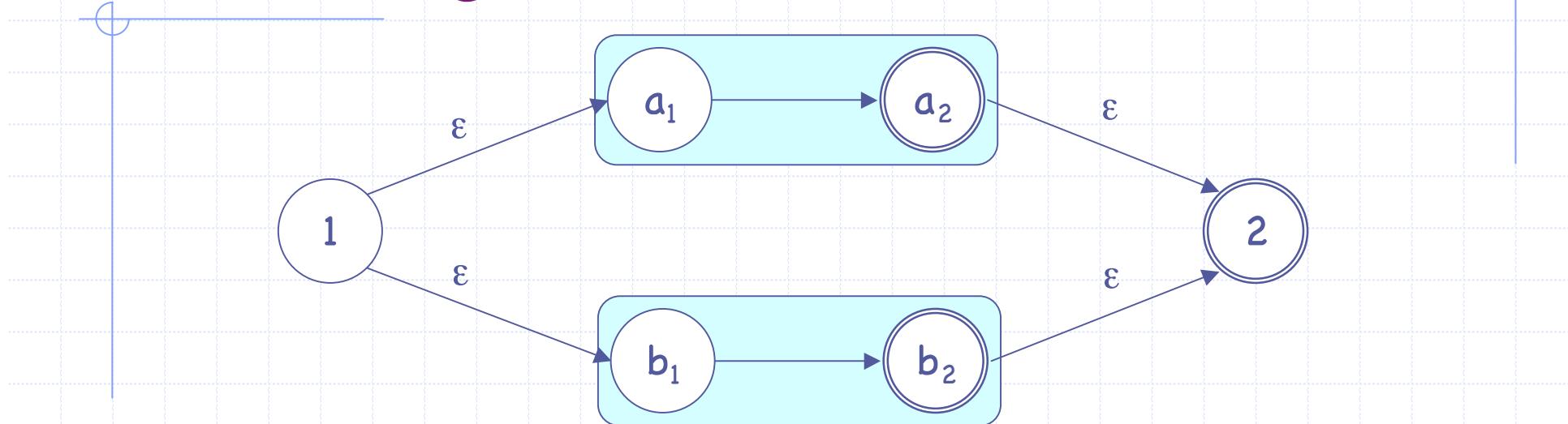
$$\begin{aligned} q_0 &= a \cdot q_1 \mid a \cdot q_2 \mid e \\ q_1 &= b \cdot q_0 \\ q_2 &= a \cdot q_3 \\ q_3 &= c \cdot q_0 \\ q_2 &= a \cdot c \cdot q_0 \\ q_0 &= a \cdot b \cdot q_0 \mid a \cdot a \cdot c \cdot q_0 \mid e \\ q_0 &= (a \cdot b \mid a \cdot a \cdot c) \cdot q_0 \mid e \\ q_0 &= a \cdot (b \mid a \cdot c) \cdot q_0 \mid e \\ q_0 &= (a \cdot (b \mid a \cdot c))^* \end{aligned}$$

The systems S<sub>1</sub> and S<sub>2</sub> are language-equivalent. The accepted language is L(S<sub>1</sub>) = L(S<sub>2</sub>) = (a · (b | a · c))\*

# Language Equivalence

- Language equivalence is blind for non-determinism. Thus, each non-deterministic automaton can be transformed into an equivalent deterministic one.
- However, language equivalence is blind for deadlocks. That is, the equivalence definition has only limited applicability.

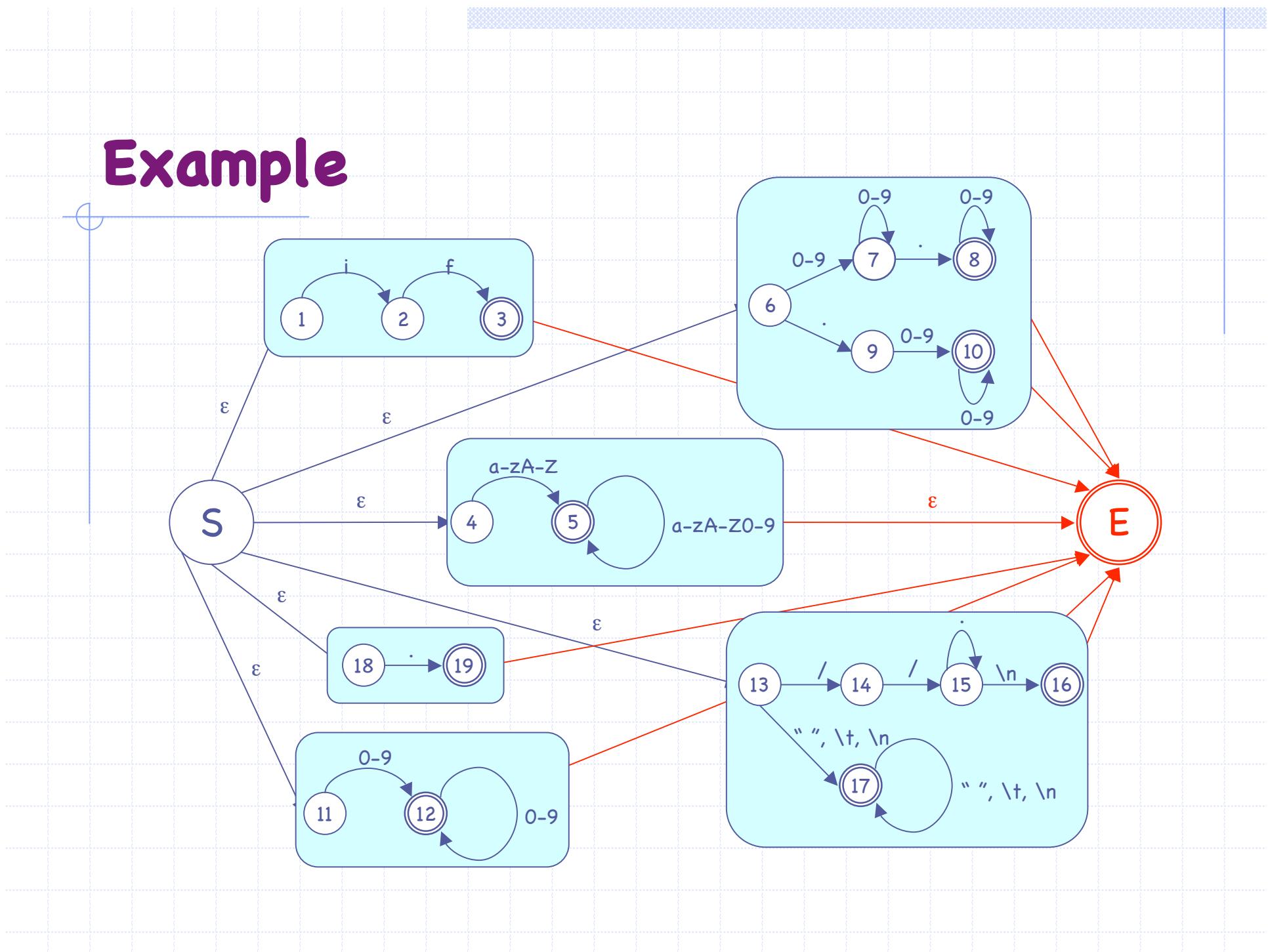
# Combining DFA's



We can define a new NFA to combine the DFA's defined for the lexical tokens. That is, we introduce a new start and end state and connect them with their corresponding states in the DFA's through an  $\epsilon$  transition.

Note that we also have to rename the states in the DFA's in order to make them unique.

# Example



# Conversion of an NFA into a DFA

Multivalued transitions make it hard to implement an NFA, since most computers do not have good "guessing" hardware.

The problem:  $\epsilon$  transitions.

The solution: We have to find a deterministic  $\epsilon$ -free automaton.

Proposition: For each finite automaton there exists an equivalent  $\epsilon$ -free automaton.

The approach: We can compute the  $\epsilon$ -closure for each reachable state. By combining the states in the  $\epsilon$ -closure, that is, defining a new state, we can construct from an NFA a DFA that accepts the same language.

# The $\varepsilon$ -closure

- Let  $A = (\Sigma, Q, q_0, \sigma, F)$  be an NFA. Then

$$\text{edge}(s, c) = \{ q \mid (s, c, Q_{(s,c)}) \in \sigma \wedge q \in Q_{(s,c)} \}$$

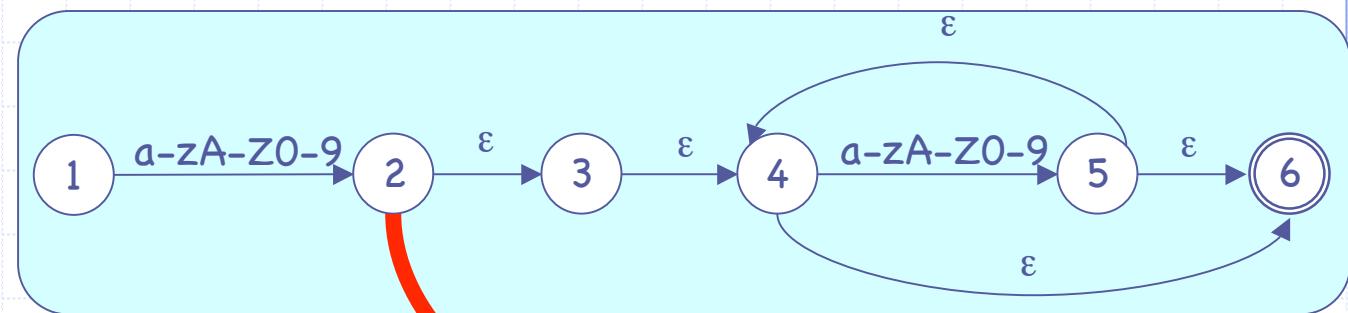
That is,  $\text{edge}(s, c)$  is the set of all NFA states reachable by following a single edge with label  $c$  from state  $s$ .

- For a set of states  $S$ , the  $\varepsilon$ -closure( $S$ ) is the set of states that can be reached from a state in  $S$  going only through  $\varepsilon$ -edges. That is,  $\varepsilon$ -closure( $S$ ) is the smallest set  $T$  such that

$$T = S \cup \left( \bigcup_{s \in T} \text{edge}(s, \varepsilon) \right)$$

# Iterative Calculation

```
T ← S  
repeat  
    T' ← T  
    T ← T' ∪ (Us ∈ T' edge(s, e))  
until T = T'
```



$S = \text{edge}(2, e)$

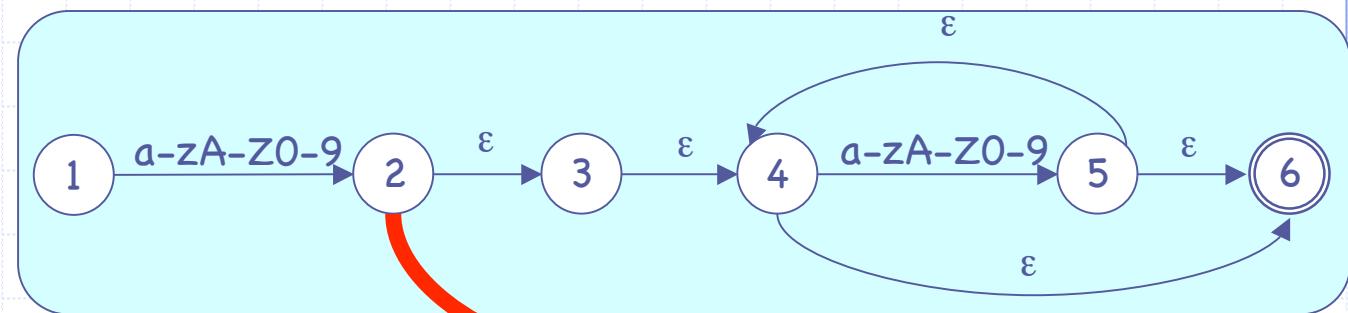
0:  $S = \{3\}$   
1:  $T = \{3\}; T' = \{3\}; T = \{3, 4\}$   
2:  $T = \{3, 4\}; T' = \{3, 4\}; T = \{3, 4, 6\}$   
3:  $T = \{3, 4, 6\}; T' = \{3, 4, 6\}; T = \{3, 4, 6\}$

# Consuming Input

- Suppose we are in a set  $d = \{s_1, s_2, s_3\}$  consisting of the NFA states  $s_1, s_2, s_3$ . Starting in  $d$  and consuming the input symbol  $c$ , we reach a new set of NFA states:

$$\text{DFAedge}(d, c) = \varepsilon\text{-closure}(\bigcup_{s \in d} \text{edge}(s, c))$$

# DFA-edge Calculation



```
d ← e-closure( { 2 } )
foreach c in [a-zA-Z0-9]
    d ← DFAedge(d, c)
```

d = e-closure ( { 2 } )

0: d = { 2, 3, 4, 6 }  
1: d = { 2, 3, 4, 6 };  
 d' ← e-closure( $\bigcup_{s \in d} \text{edge}(s, c)$ );  
 d' = {4, 5, 6}  
2: d = { 4, 5, 6 }

# DFA construction

- Let  $A = (\Sigma, Q, q_0, \sigma, F)$ .

$\text{states}[0] \leftarrow \{\}; \text{states}[1] \leftarrow \varepsilon\text{-closure}(\{q_0\})$

$p \leftarrow 1; j \leftarrow 0;$

**while**  $j \leq p$

**foreach**  $c \in \Sigma$

$e \leftarrow \text{DFAedge}(\text{states}[j], c)$

**if**  $e = \text{states}[i]$  **for some**  $i \leq p$

**then**  $\text{trans}[j, c] \leftarrow i$

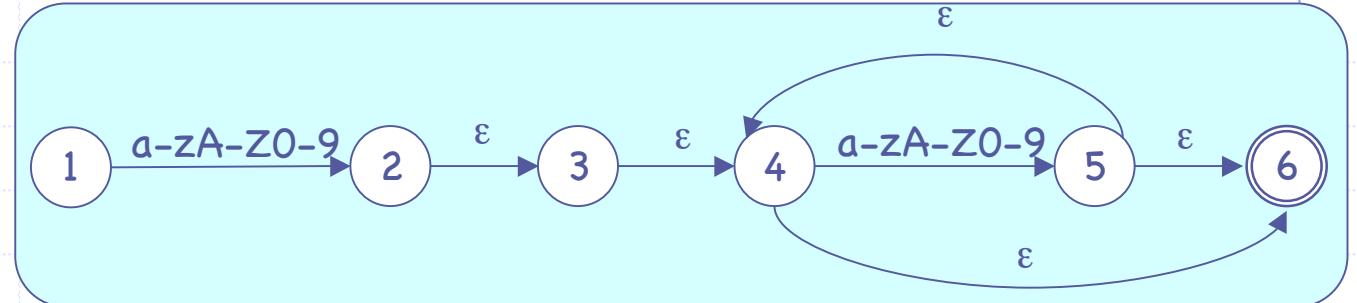
**else**  $p \leftarrow p + 1$

$\text{states}[p] \leftarrow e$

$\text{trans}[j, c] \leftarrow p$

$j \leftarrow j + 1$

# Example



$$A = (\{a-zA-Z0-9\}, \{1, 2, 3, 4, 5, 6\}, 1, \sigma, \{6\})$$

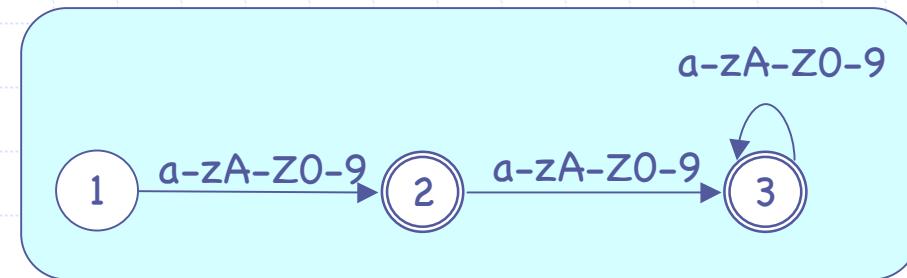
```
states[0] ← {};
states[1] ← {1}
j ← 0; e ← DFAedge( {}, Σ) = {}
trans[j,Σ] ← 0
j ← 1; e ← DFAedge( {1}, Σ) = {2, 3, 4, 6}
p ← 2; states[p] ← {2, 3, 4, 6}; trans[j,Σ] ← 2
j ← 2; e ← DFAedge( {2, 3, 4, 6}, Σ) = { 4, 5, 6}
p ← 3; states[p] <- { 4, 5, 6 }; trans[j,Σ] ← 3
j ← 3; e ← DFAedge( {4, 5, 6}, Σ) = { 4, 5, 6 }
trans[j,Σ] ← 3
j ← 4
```

# New DFA

```
states[0] = {}  
states[1] = {1}  
states[2] = {2, 3, 4, 6}  
states[3] = { 4, 5, 6 }
```

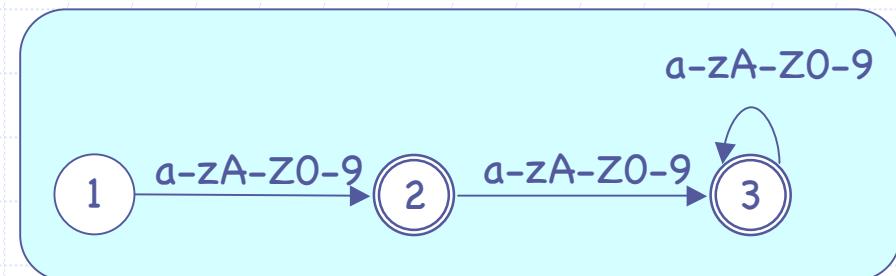
```
trans[0, $\Sigma$ ] = 0  
trans[1, $\Sigma$ ] = 2  
trans[2, $\Sigma$ ] = 3  
trans[3, $\Sigma$ ] = 3
```

Q/ $\Sigma$	a-zA-Z0-9
0	0
1	2
2	3
3	3



The generated automaton is suboptimal, since it is not the smallest one that accept the same language. The states 2 and 3 are equivalent and can therefore be merged (see Hopcroft and Ullman, Theorem 3.10).

# State Partitions



Let  $A = (\Sigma, Q, q_0, \sigma, F)$  be a DFA.

The states  $q, r \in Q$  are called

- 0-equivalent,  $q \equiv r$ , if  $q, r \in F$  or  $q, r \in Q - F$ .
- $k$ -equivalent,  $q \equiv^k r$   $\text{card}(Q) \geq k > 0$ , if
  - $q \equiv^{k-1} r$ , and
  - for all  $c \in \Sigma$  it holds either  
 $\sigma(q, c) \equiv^{k-1} \sigma(r, c)$  or  $\sigma(q, c) = \sigma(r, c) = \emptyset$ .

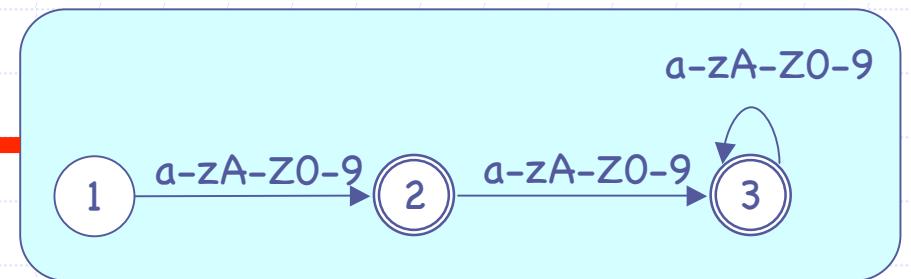
$\Pi_{0,0} = \{1\}$   
 $\Pi_{0,1} = \{2, 3\}$   
 $\Pi_{1,0} = \{1\}$   
 $\Pi_{1,1} = \{2, 3\}$

# Building a Minimal DFA

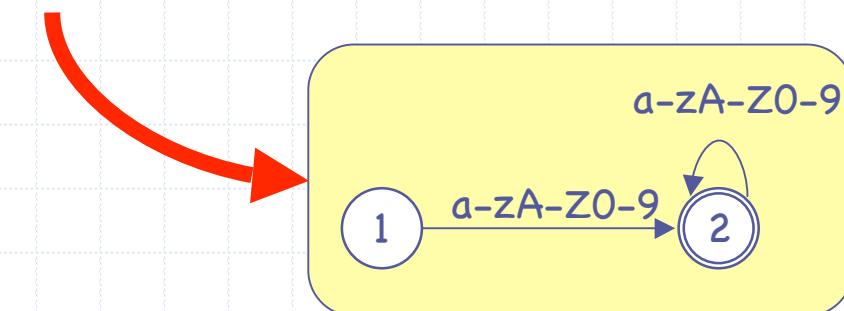
Let  $A = (\Sigma, Q, q_0, \sigma, F)$  be a DFA.

1. Determine the  $\Pi_k$  of equivalent states with  $k \leq \text{card}(Q)$ .
2. Choose one state in each group  $\Pi_{k,n}$  as representative of  $\Pi_{k,n}$  and build  $Q'$ .
3. Determine the transition function  $\sigma'$ :  
Let  $q$  be the representative state. For all  $c \in \Sigma$  if  $\sigma(q, c) = r$  with  $r \in \Pi_{k,m}$ , then add  $\sigma(q, c)$  to  $\sigma'$ .
4. Let the start state of  $A'$  be the representative of the group  $\Pi_{k,n}$  containing state  $q_0$ .
5. Let the accepting states of  $A'$  be the representatives that are in  $F$ .
6.  $A' = (\Sigma, Q', q_0', \sigma', F')$  is the minimal DFA.

# Example



1.  $\Pi_{1,0} = \{1\}, \Pi_{1,1} = \{2, 3\}$
2.  $Q' = \{1, 2\}$
3.  $\sigma' = \{(1, a-zA-Z0-9, 2), (2, a-zA-Z0-9, 2)\}$
4.  $q_0' = 1$
5.  $F' = \{2\}$
6.  $A' = (\{a-zA-Z0-9\}, \{1, 2\}, 1, \{(1, a-zA-Z0-9, 2), (2, a-zA-Z0-9, 2)\}, \{2\})$



# JavaCC Specification

```
PARSER_BEGIN(FirstExample)
    class FirstExample {} // parser class specification
PARSER_END(FirstExample)

// token specification
TOKEN : {   < IF : "if" >
            | < IDENTIFIER : ["a"-"z","A"-"Z"] (["a"-"z","A"-"Z","0"-"9"])* >
            | < INTEGER : ([0"-"9])+ >
            | < REAL : ( ([0"-"9])+ "." ([0"-"9])* ) | ( ([0"-"9])* "." ([0"-"9])+ ) > }

// white space
SKIP :
{ < //> (~["\n","\r"])* ("\\n"|"\\r"|"\\r\\n") > | " " | "\\t" | "\\r" | "\\n" }

// test
void Scan() :
{
{ ( <IF> | <IDENTIFIER> | <INTEGER> | <REAL> )* }
```

# Test Program

```
public class test
{
    public static void main( String[] args )
    {
        try
        {
            FirstExample lScanner = new FirstExample( System.in );
            lScanner.Scan();
            System.out.println( "Success" );
        }
        catch (ParseException e)
        {
            System.out.println( e );
        }
    }
}
```

# JavaCC Specification Version 2

```
PARSER_BEGIN(FirstExample)
    class FirstExample {} // parser class specification
PARSER_END(FirstExample)

// token specification
TOKEN : {   < IF : "if" >
            | < #DIGIT : ["0"- "9"] >
            | < IDENTIFIER : ["a"- "z", "A"- "Z"] (["a"- "z", "A"- "Z"] | <DIGIT>)* >
            | < INTEGER : (<DIGIT>)+ >
            | < REAL : ( (<DIGIT>)+ "." (<DIGIT>)* ) | ( (<DIGIT>)* ":" (<DIGIT>)+ ) > }

// white space
SKIP :
{ <("//" (~["\n","\r"])* ("\n|\r|\r\n") | " " | "\t" | "\r" | "\n" }

// test
void Scan() :
{
{ ( <IF> | <IDENTIFIER> | <INTEGER> | <REAL> )* }
```

# Syntax Analysis

The role of the parser:

The parser obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. That is, parsing the string of tokens succeeds if and only if there exists a derivation sequence from the start symbol of the grammar to the string of token.

The result of the syntax analysis is an abstract syntax tree.

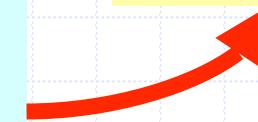
# Need for more Flexibility

Valid regular expression specification:

$\langle \text{Digits} \rangle = [0-9]^+$

$\langle \text{Sum} \rangle = (\langle \text{Digits} \rangle "+")^* \langle \text{Digits} \rangle$

28+301+9



Invalid regular expression specification:

$\langle \text{Digits} \rangle = [0-9]^+$

$\langle \text{Sum} \rangle = \langle \text{Exp} \rangle "+" \langle \text{Exp} \rangle$

$\langle \text{Exp} \rangle = "(" \langle \text{Sum} \rangle ")" | \langle \text{Digits} \rangle$

(1+(250+3))



We cannot construct a DFA for the second specification!

# Pumping Lemma for Regular Languages

If  $L$  is a regular language (e.g., the language of a finite-state automaton), then there is a number  $p$  (the pumping length) where, if  $s$  is any string in  $L$  of length at least  $p$ , then  $s$  may be divided into three pieces,  $s = xyz$ , satisfying the following conditions:

1. For each  $i \geq 0$ ,  $xy^i z \in L$ ,
2.  $|y| > 0$ , and
3.  $|xy| \leq p$ .

When  $s$  is divided into  $xyz$ , either  $x$  or  $z$  may be  $\epsilon$ , but condition 2 says that  $y \neq \epsilon$ . Condition 3 states that the pieces  $x$  and  $y$  together have length at most  $p$ . This is an extra condition that is very useful when proving that a given language is not regular.

# Is $L = \{a^n b^n \mid n > 0\}$ regular?

Let  $L$  be the language  $\{a^n b^n \mid n > 0\}$ . We use the pumping lemma to prove that  $L$  is not regular. The proof is by contradiction.

Assume  $L$  is regular. Let  $p$  be the pumping length given by the pumping lemma. Choose  $s$  to be the string  $a^p b^p$ . Because  $s$  is a member of  $L$  and  $s$  has length more than  $p$ , the pumping lemma guarantees that  $s$  can be split into three pieces  $s = xyz$ , where for any  $i \geq 0$ ,  $xy^i z$  is in  $L$ . We consider the three cases to show that this is impossible.

- The string  $y$  consists only of  $a$ 's. Thus,

$$a^p b^p = xyz \text{ with } x = a^r, y = a^s, \text{ and } z = a^t b^p \quad r, t \geq 0, s \geq 1, r+s+t=p$$

According to case 1 of the pumping lemma, it is also required that all strings  $a^{r+is+t} b^p = a^{p+(i-1)s} b^p$  with  $i \geq 0$  have to be elements of  $L$ . But this is only true for  $i = 1$ , hence a contradiction.

- The string  $y$  consists only of  $b$ 's. This case gives also a contradiction.
- The string  $y$  consists of both  $a$ 's and  $b$ 's. Thus,

$$a^p b^p = xyz \text{ with } x = a^r, y = a^s b^t, \text{ and } z = b^u \quad r, u \geq 0, s, t \geq 1, r+s=t+u=p$$

Now it is required that all strings  $a^r (a^s b^t)^i b^u$  with  $i \geq 0$  have to be elements of  $L$ . But this is not true for  $i \geq 2$ , hence a contradiction.

Q.E.D.

# Grammar

Definition:

A vocabulary (or alphabet)  $V$  is a finite, nonempty set of symbols. A sentence over  $V$  is a string of finite length of elements of  $V$ . The empty string, denoted by  $\epsilon$ , is the string containing no symbols. The set of all sentences over  $V$  is denoted by  $V^*$ .

Definition:

A grammar is a quadruple  $G = (V, T, S, P)$  with

- A vocabulary  $V$ ,
- A subset  $T \subset V$  consisting of terminal elements (tokens),
- A start symbol  $S \in (V - T)$ , and
- A set of productions  $P \subseteq (V^* - T^*) \times V^*$ .

# Language

Definition:

Let  $G = (V, T, S, P)$  be a grammar. The *language* generated by  $G$  (or the *language of*  $G$ ), denoted by  $L(G)$ , is the set of all strings of terminals that are derivable from the start symbol  $S$ . In other words,

$$L(G) = \{ w \in T^* \mid S \Rightarrow^+ w \}$$

# Types of Grammars

- Type 0: A grammar that has no restrictions on its productions.
- Type 1 - context-sensitive: A grammar can only have productions of the form  $(w_1, w_2)$ , where the length of  $w_2$  is greater than or equal to the length of  $w_1$ , or the form  $(w_1, \epsilon)$ .
- Type 2 - context-free: A grammar can only have productions of the form  $(w_1, w_2)$ , where  $w_1$  is a single nonterminal.
- Type 3 - regular: A grammar can only have productions of the form  $(w_1, w_2)$ , with  $w_1$  is a nonterminal and  $w_2$  is either  $aB$ ,  $Ba$ ,  $a$ , or  $\epsilon$ , where  $B$  is a nonterminal and  $a$  is a terminal.

# $L = \{a^n b^n \mid n > 0\}$ Revisited

$\langle \text{Digits} \rangle = [0-9]^+$

$\langle \text{Sum} \rangle = \langle \text{Exp} \rangle "+" \langle \text{Exp} \rangle$

$\langle \text{Exp} \rangle = "(" \langle \text{Sum} \rangle ")" \mid \langle \text{Digits} \rangle$

Let  $G = (\{s, a, b\}, \{a, b\}, s, \{(s, asb), (s, ab)\})$  be a grammar.

Then the language generated by  $G$  is

$$L(G) = \{a^n b^n \mid n > 0\}.$$

Furthermore, the production  $(s, asb)$  makes  $G$  a context-free grammar. Therefore,  $L(G)$  is a context-free language.

# The Backus-Naur Form

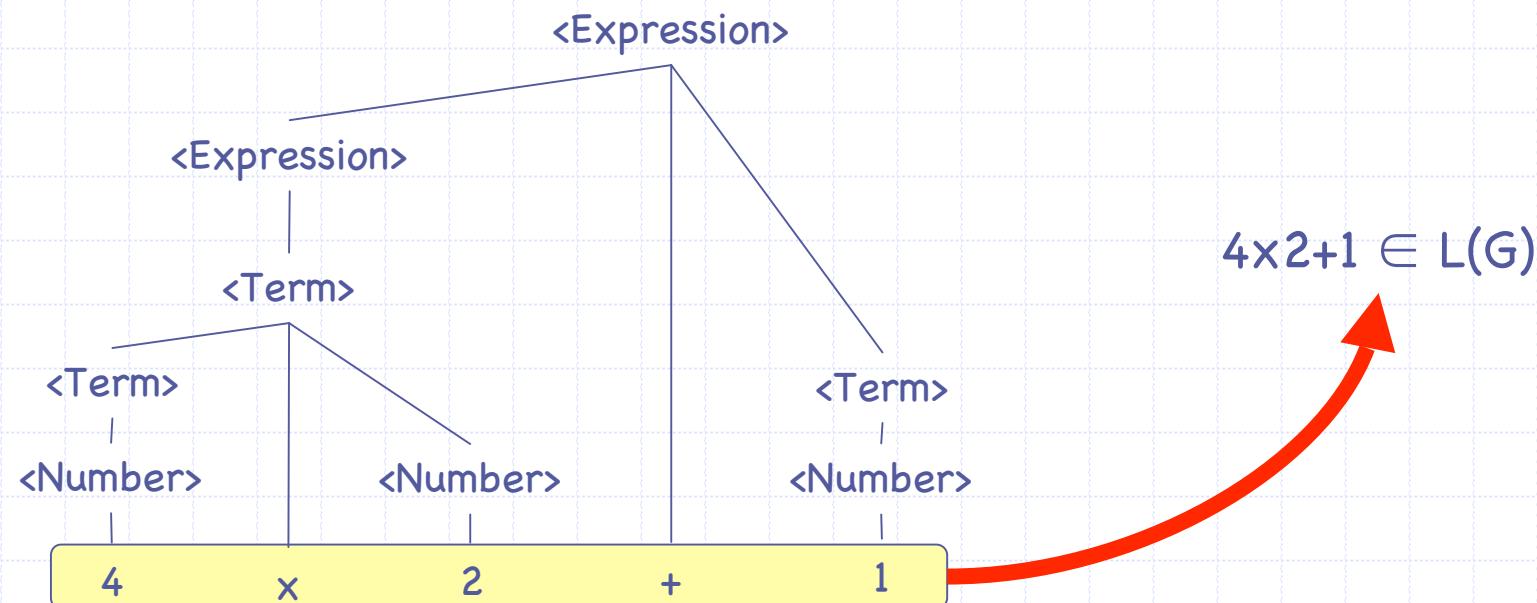
- In general, we specify the grammar of a programming language using a context-free grammar (or type 2 grammar).
- We use a notation called Backus-Naur Form (BNF).
- The general rule format is:

*lhs ::= rhs*

where *lhs* is a nonterminal, and *rhs* may be a list, separated with “|” of strings of terminals and nonterminals.

# Syntax Tree

```
G = ( V = { <Expression>, <Term>, <Number>, + , -, ×, / },
      T = { <Number>, +, -, ×, / },
      S = <Expression>,
      P = { p1 = (<Expression> := <Expression> (+|-) <Term>), p2 =(<Expression> :=
<Term>),
      p3 = (<Term> := <Term> (×|/) <Number>), p4 = (<Term> := <Number>) }
    )
```



# Derivation

```
G = ( V = { <Expression>, <Term>, <Number>, + , -, ×, / },
      T = { <Number>, +, -, ×, / },
      S = <Expression>,
      P = { p1 = (<Expression> := <Expression> (+|-) <Term>), p2 =(<Expression> := <Term>),
             p3 = (<Term> := <Term> (x|/) <Number>), p4 = (<Term> := <Number>) } )
```

<Expression>  
→ p<sub>1</sub>    <Expression> + <Term>  
→ p<sub>2</sub>    <Term> + <Term>  
→ p<sub>3</sub>    <Term> × <Number> + <Term>  
→ p<sub>4</sub>    <Number> × <Number> + <Term>  
→ p<sub>4</sub>    <Number> × <Number> + <Number>

=                  4            ×            2            +            1

# Ambiguous Grammars

A grammar is ambiguous if we can derive a sentence  $s \in L(G)$  with two or more different syntax trees.

Consider:

$\langle \text{Expression} \rangle ::= \langle \text{Identifier} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Number} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle + \langle \text{Expression} \rangle$

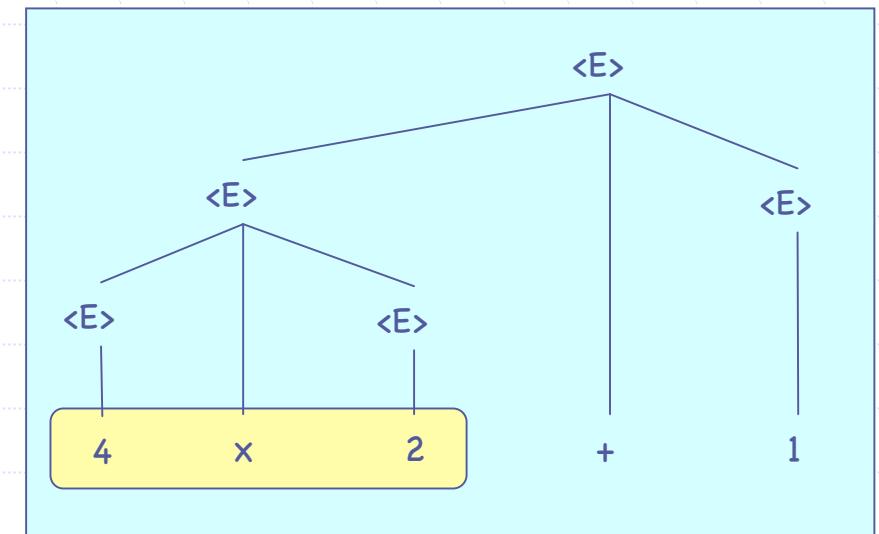
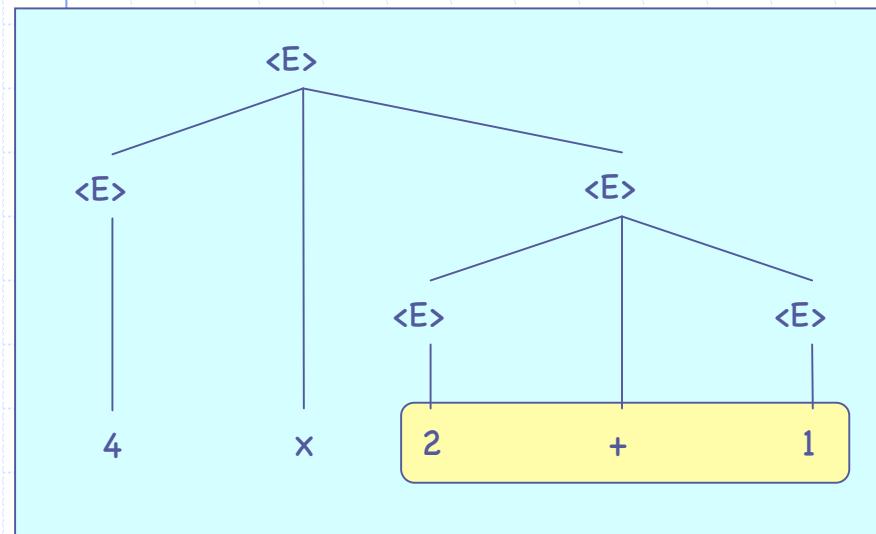
$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle - \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle \times \langle \text{Expression} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle / \langle \text{Expression} \rangle$

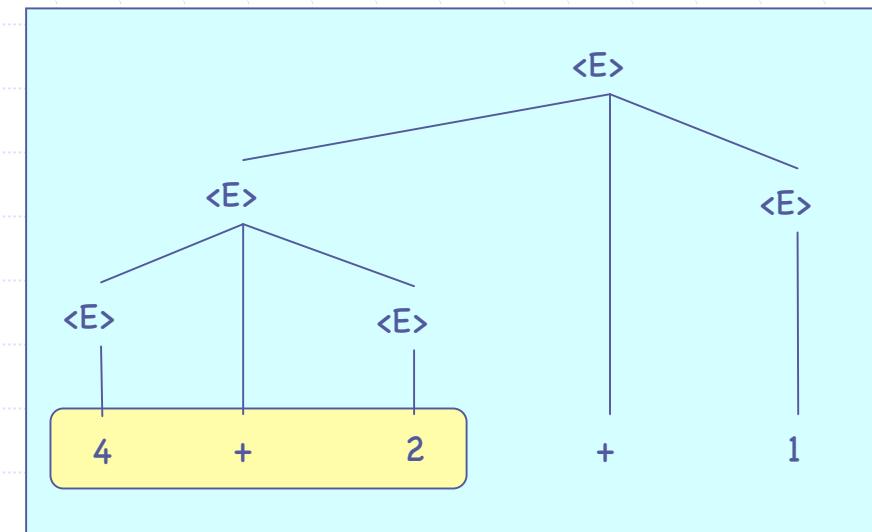
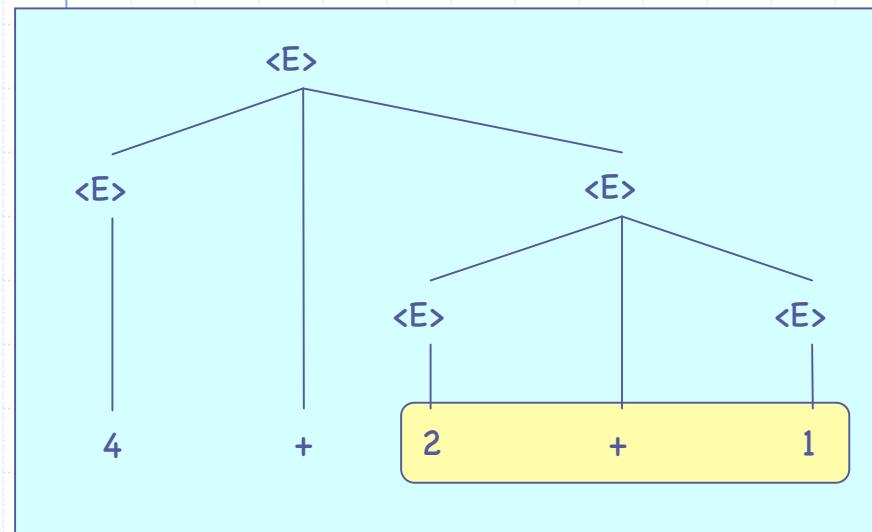
$\langle \text{Expression} \rangle ::= ( \langle \text{Expression} \rangle )$

**4 × 2 + 1**



Which syntax tree is the correct one?

4 + 2 + 1



Which syntax tree is the correct one?

# Precedence & Associativity

## Precedence:

An operator has a higher precedence if the syntax rule that defines its application occurs nested in another rule. That is, an operator binds tighter if its syntax rule is defined deeper in the hierarchy.

## Associativity:

An operator *associates to the left* if the syntax rule that defines its application uses a left-recursive rule structure. An operator *associates to the right* if the syntax rule that defines its application uses a right-recursive rule structure. Non-recursive rules result in operators with no associativity.

# An Expression Language

`<Expression> ::= <Expression> + <Term>`

`<Expression> ::= <Expression> - <Term>`

`<Expression> ::= <Term>`

`<Term> ::= <Term> × <PrimaryExpression>`

`<Term> ::= <Term> / <PrimaryExpression>`

`<Term> ::= <PrimaryExpression>`

`<PrimaryExpression> ::= <Identifier>`

`<PrimaryExpression> ::= <Number>`

`<PrimaryExpression> ::= ( <Expression> )`

# Recursive-descent Parsing

Some grammars are easy to parse using a simple approach called *predictive* or *recursive-descent* parsing.

## Recursive-decent parsing:

- Each grammar rule is transformed into a recursive function.
- Success is indicated by a Boolean value. That is, a function representing a grammar rule returns true upon successful parsing of a given input string.
- If a grammar rule contains an  $\epsilon$ -clause, the corresponding recursive function always succeeds (i.e., no result value).

# Example Grammar

`<Statement> ::= if <Expression> then <Statement> else <Statement>`

`<Statement> ::= begin <Statement> <StatementRest>`

`<Statement> ::= print <Expression>`

`<StatementRest> ::= end`

`<StatementRest> ::= ; <Statement> <StatementRest>`

`<Expression> ::= <Number> = <Number>`

# Lexical Analysis

```
PARSER_BEGIN(SimpleParser)
    public class SimpleParser {}
PARSER_END(SimpleParser)
```

```
TOKEN:
{
    < IF : "if" > | < THEN : "then" > | < ELSE : "else" > |
    < BEGIN : "begin" > | < END : "end" > | < PRINT : "print" >
    < SEMICOLON : ";" > | < EQUAL : "=" > |
    < NUMBER : ("0"- "9")+ >
}
```

```
SKIP :
{
    < //> (~["\n", "\r"])* ("\n" | "\r" | "\r\n")
    | " " | "\t" | "\r" | "\n" >
}
```

```
public class SimpleParser
{
    public static Token getNextToken() { ... }
}
```

# Parser Structure

```
public class Parser implements SimpleParserConstants
{
    private SimpleParser fSimpleParser;
    private Token CurrentToken;

    public static void main( String[] args ) { ... }

    public Parser( java.io.InputStream stream )
    {
        fSimpleParser = new SimpleParser( stream );
        nextToken();
    }

    private void nextToken() { CurrentToken = SimpleParser.getNextToken(); }

    public boolean Program() { ... }
    public boolean Statement() { ... }
    public boolean StatementRest() { ... }
    public boolean Expression() { ... }
}
```

# Program()

```
if 2 = 3 then print 4 = 5 else print 6 = 7
```

```
public boolean Program()
{
    if ( Statement() )
        return ( CurrentToken.kind == EOF );
    else
        return false;
}
```

A parser must also check for EOF.

# Statement()

```
public boolean Statement() {  
    switch ( CurrentToken.kind ) {  
        case IF:  
            nextToken();  
            if ( Expression() ) {  
                if ( CurrentToken.kind == THEN ) {  
                    nextToken();  
                    if ( Statement() ) {  
                        if ( CurrentToken.kind == ELSE ) {  
                            nextToken(); return Statement(); } } } } }  
            break;  
        case BEGIN:  
            nextToken();  
            if ( Statement() ) return StatementRest();  
            break;  
        case PRINT:  
            nextToken();  
            if ( Expression() ) return true;  
            break; }  
    return false;  
}
```

consume token

return failure

return success

# StatementRest()

```
public boolean StatementRest()
{
    switch ( CurrentToken.kind )
    {
        case END:
            nextToken();
            return true;
        case SEMICOLON:
            nextToken();
            if ( Statement() )
                return StatementRest();
            break;
    }
    return false;
}
```

# Expression()

```
public boolean Expression()
{
    if ( CurrentToken.kind == NUMBER )
    {
        nextToken();
        if ( CurrentToken.kind == EQUAL )
        {
            nextToken();
            if ( CurrentToken.kind == NUMBER
)
            {
                nextToken();
                return true;
            }
        }
    }
    return false;
}
```

# Problem: Left Recursion

Consider the expression language:

direct left recursion

```
public boolean Expression()
{
    if ( Expression() )
    {
        if ( CurrentToken.kind == PLUS ) return Term();
    }
    else
    {
        if ( Expression() )
        {
            if ( CurrentToken.kind == MINUS ) return
Term();
        }
        else return Term();
    }
    return false;
}
```

# Problem: Choosing Rule

Consider the expression language:

```
public boolean Expression()
{
    switch ( CurrentToken.kind )
    {
        case ?: if ( Expression() )
                  if ( CurrentToken.kind == PLUS ) return Term();
                  break;
        case ?: if ( Expression() )
                  if ( CurrentToken.kind == MINUS ) return Term();
                  break;
        case ?: return Term();
    }
    return false;
}
```

Which clause should we choose?

# LL(1) Property

LL(1)-property:

- By using one lookahead symbol only, we have to be able to determine the next action.
- The first symbols of all right-hand-sides must be pairwise disjoint. For example, if  $\text{lhs} ::= \text{rhs}_1 \mid \text{rhs}_2$ , then it must hold that  $\text{FIRST}(\text{rhs}_1) \cap \text{FIRST}(\text{rhs}_2) = \emptyset$ .
- All production must not have direct or indirect left-recursive application of the same production. For example, the following rule is not LL(1):

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle + \langle \text{Term} \rangle$

# FIRST(X)

$$\text{FIRST}(X) = \{ a \mid X \Rightarrow^* aw_1, a \in T, w_1 \in V^* \}$$

$$\text{FIRST}(X) = \{ X \mid X \in T \}$$

$$\text{FIRST}(X) = \{ \epsilon \mid X = \epsilon \}$$

$$\begin{aligned} \text{FIRST}(X) = & \{ a \mid X \in (V-T), (X, w_1w_2\dots w_n) \in P, \\ & a = \text{FIRST}(w_1 w_2 \dots w_n) \\ & \text{if } a = \text{FIRST}(w_i), 1 \leq i \leq n, \text{ and} \\ & \epsilon \in \text{FIRST}(w_1), \dots, \text{FIRST}(w_{i-1}), \text{ or} \\ & a = \epsilon \text{ if } \epsilon \in \text{FIRST}(w_1), \dots, \text{FIRST}(w_n) \} \end{aligned}$$

# FOLLOW(X)

$$\text{FOLLOW}(X) = \{ a \mid S \Rightarrow^* w_1 X a w_2, a \in T, w_1, w_2 \in V^* \}$$

- Place EOF in FOLLOW(S), i.e.,  $\text{FOLLOW}(X) = \{\text{EOF} \mid X = S\}$ .
- If there is a production  $(X, w_1 B w_2)$ , then everything in  $\text{FIRST}(w_2)$  except for  $\epsilon$  is placed in  $\text{FOLLOW}(B)$ .
- If there is a production  $(X, w_1 B)$ , or a production  $(X, w_1 B w_2)$  where  $\text{FIRST}(w_2)$  contains  $\epsilon$ , then everything in  $\text{FOLLOW}(X)$  is in  $\text{FOLLOW}(B)$ .

# Example

```
Z ::= d | X Y Z  
Y ::=   | c  
X ::= Y | a
```

	FIRST
Z	{a, c, d}
Y	{ε, c}
X	{ε, a, c}

	FOLLOW
Z	{<EOF>}
Y	{a, c ,d}
X	{a, c, d}

# Predictive Parser Table

Construction of a Predictive Parser Table M

For each production  $(A, w_1)$ :

- ♦ For each terminal  $a$  in  $\text{FIRST}(w_1)$ , add  $(A, w_1)$  to  $M[A, a]$ .
- ♦ If  $\epsilon$  is in  $\text{FIRST}(w_1)$ , add  $(A, w_1)$  to  $M[A, b]$  for each terminal  $b$  in  $\text{FOLLOW}(A)$  and if  $\langle\text{EOF}\rangle$  is in  $\text{FOLLOW}(A)$  add  $(A, w_1)$  to  $M[A, \langle\text{EOF}\rangle]$ .

Make each undefined entry in M an error.

# Example

$Z ::= d \mid X Y Z$   
 $Y ::= \quad \mid c$   
 $X ::= Y \mid a$

	FIRST
$Z$	$\{a, c, d\}$
$Y$	$\{\epsilon, c\}$
$X$	$\{\epsilon, a, c\}$

	FOLLOW
$Z$	$\{\text{EOF}\}$
$Y$	$\{a, c, d\}$
$X$	$\{a, c, d\}$

M	a	c	d	<EOF>
$Z$	$(Z, X Y Z)$	$(Z, X Y Z)$	$(Z, d)$	$(Z, d)$ $(Z, X Y Z)$
$Y$	$(Y, )$	$(Y, c)$ $(Y, )$	$(Y, )$	error
$X$	$(X, a)$ $(X, Y)$	$(X, Y)$	$(X, Y)$	error

# Ambiguous Grammar

	FIRST
Z	{a, c, d}
Y	{ε, c}
X	{ε, a, c}

M	a	c	d	<EOF>
Z	(Z,X Y Z)	(Z,X Y Z)	(Z,d) (Z,X Y Z)	(Z,d) (Z,X Y Z)
Y	(Y, )	(Y, )	(Y, )	error
X	(X,a) (X,Y)	(X,Y)	(X, Y)	error

If  $M[X,a]$  contains multiple entries, then the grammar cannot be parsed with a recursive-descent parser!

# Elimination of Direct Left Recursion

```
<E> ::= <E> + <T>
<E> ::= <E> - <T>
<E> ::= <T>
```

```
<T> ::= <T> x <P>
<T> ::= <T> / <P>
<T> ::= <P>
```

```
<P> ::= <Id>
<P> ::= <N>
<P> ::= ( <E> )
```

There exists a simple way to transform left-recursive grammar rules into right-recursive ones.

```
<E> ::= <T> + <E>
<E> ::= <T> - <E>
<E> ::= <T>
```

```
<T> ::= <P> x <T>
<T> ::= <P> / <T>
<T> ::= <P>
```

```
<P> ::= <Id>
<P> ::= <N>
<P> ::= ( <E> )
```

However, this transformation does not preserve the semantics of the rules. For example, the operators associate to the right now.

# Recursive and Terminating Rules

Let  $G = (V, T, S, P)$  be a context-free grammar that contains direct left-recursive rules for  $z \in (V - T)$ .

We can build two sets:

- $P_{z_{\text{Rec}}} = \{ (z, zw) \mid (z, zw) \in P \text{ and } w \in V^+ \}$
- $P_{z_{\text{Term}}} = \{ (z, v) \mid (z, v) \in P, v \neq zv', \text{ and } v, v' \in V^* \}$

Note that rules of the form  $(z, z)$  are deleted immediately.

# Constructing a New Grammar

Let  $z', z''$  be two non-terminals with  $z', z'' \notin (V - T)$ . Based on  $G$ , we can construct a new equivalent grammar  $G'$  that does not contain left-recursive rules for  $z \in (V - T)$ :

$G' = (V \cup \{z', z''\}, T, S, P')$  with

$$P' = P - P_{z_{\text{Rec}}} - P_{z_{\text{Term}}} \cup \{(z', ), (z', z''z')\} \cup P_{z_A} \cup P_{z_B}$$

$$P_{z_A} = \{(z'', w) \mid (z, zw) \in P_{z_{\text{Rec}}}\}$$

$$P_{z_B} = \{(z, vz') \mid (z, v) \in P_{z_{\text{Term}}}\}$$

# Example

```
 $\langle E \rangle ::= \langle E \rangle + \langle T \rangle$ 
 $\langle E \rangle ::= \langle E \rangle - \langle T \rangle$ 
 $\langle E \rangle ::= \langle T \rangle$ 
```

```
 $\langle T \rangle ::= \langle T \rangle \times \langle P \rangle$ 
 $\langle T \rangle ::= \langle T \rangle / \langle P \rangle$ 
 $\langle T \rangle ::= \langle PE \rangle$ 
```

```
 $\langle PE \rangle ::= \langle Id \rangle$ 
 $\langle PE \rangle ::= \langle N \rangle$ 
 $\langle PE \rangle ::= ( \langle E \rangle )$ 
```

```
 $\langle E \rangle ::= \langle T \rangle \langle E' \rangle$ 
 $\langle E'' \rangle ::= + \langle T \rangle$ 
 $\langle E'' \rangle ::= - \langle T \rangle$ 
 $\langle E' \rangle ::= | \langle E'' \rangle \langle E' \rangle$ 
```

```
 $\langle T \rangle ::= \langle PE \rangle \langle T' \rangle$ 
 $\langle T'' \rangle ::= \times \langle P \rangle$ 
 $\langle T'' \rangle ::= / \langle PE \rangle$ 
 $\langle T' \rangle ::= | \langle T'' \rangle \langle T' \rangle$ 
```

```
 $\langle PE \rangle ::= \langle Id \rangle$ 
 $\langle PE \rangle ::= \langle N \rangle$ 
 $\langle PE \rangle ::= ( \langle E \rangle )$ 
```

$$P_{E_R} = \{(\langle E \rangle, \langle E \rangle + \langle T \rangle), (\langle E \rangle, \langle E \rangle - \langle T \rangle)\}$$

$$P_{E_T} = \{(\langle E \rangle, \langle T \rangle)\}$$

$$P_{T_R} = \{(\langle T \rangle, \langle T \rangle \times \langle P \rangle), (\langle T \rangle, \langle T \rangle / \langle P \rangle)\}$$

$$P_{T_T} = \{(\langle T \rangle, \langle P \rangle)\}$$

$$P_{E_A} = \{(\langle E'' \rangle, + \langle T \rangle), (\langle E'' \rangle, - \langle T \rangle)\}$$

$$P_{E_B} = \{(\langle E \rangle, \langle T \rangle \langle E' \rangle)\}$$

$$P_{T_A} = \{(\langle T'' \rangle, \times \langle P \rangle), (\langle T'' \rangle, / \langle P \rangle)\}$$

$$P_{T_B} = \{(\langle T \rangle, \langle P \rangle \langle T' \rangle)\}$$

Note that all operators continue to associate to the left in the new grammar  $G'$ .

# Elimination of Indirect Left Recursion

Let  $G = (V, T, S, P)$  be a context-free grammar:

1. Eliminate all direct left-recursive rules.
2. Let  $Z_{rec} = \{ z \mid z \Rightarrow^+ zw, z \in (V - T), \text{ and } w \in V^* \}$
3. If  $Z_{rec} = \emptyset$ , then  $G$  does not contain indirect left-recursive rules. STOP.
4. For all  $z \in Z_{rec}$  build  $z_0 \xrightarrow{p_1} z_1w_1 \xrightarrow{p_2} z_2w_2 \xrightarrow{p_3} \dots \xrightarrow{p_n} z_nw_n$  with  $z_0 = z_n = z$ ,  $z_i \neq z_j$ , and  $0 \leq i < j < n$ .
5. For the production  $p_n = (z_{n-1}, zw')$  build
$$P' = \{(z_{n-1}, ww') \mid (z, w) \in P\} - \{(z, z) \mid z \in (V - T)\}$$
6. Let  $G = (V, T, S, P - \{p_n = (z_{n-1}, zw')\} \cup P')$  and continue with step 1.

# Left Factoring

$\text{FIRST}(p_1) \cap \text{FIRST}(p_2) = \{\text{if}\}$

`<Statement> ::= if <Expression> then <Statement> else <Statement> fi  
<Statement> ::= if <Expression> then <Statement> fi`

`<Statement> ::= if <Expression> then <Statement> <StatementRest>`

`<StatementRest> ::= else <Statement> fi  
<StatementRest> ::= fi`

# Parsing Errors

- Using the predictive parser table, it is easy to write a recursive-descent parser.
- Each blank entry in  $M[A,x]$  (i.e., marked with error) means that the procedure for the non-terminal A may see a token x, which it is not expecting. In this case, we have to report an error.
- Error recovery in recursive-descent parsing is not easy. The simplest solution is to print an error message and to terminate the parsing process.
- Error recovery requires an extra mechanism to resynchronize the parser. In general, this is accomplished by skipping tokens until a token in the FOLLOW set is reached.

# LR Parsing

- A shortcoming of the LL( $k$ ) parsing technique is that one must predict, which production to use, having seen only  $k$  tokens of the right-hand side.
- The LR( $k$ ) parsing technique, on the other hand, is able to postpone the decision until it has seen all input tokens of a right-hand side (plus  $k$  more input tokens).

# Finite Deterministic Stack Automata

A finite deterministic stack automaton is a septuple

$$(\Sigma, G, Q, q_0, \gamma_0, \sigma, F)$$

with:

- a set  $\Sigma$  of input symbols (or input alphabet),
- a set  $G$  of stack symbols (or stack alphabet),
- a set  $Q = \{q_0, q_1, \dots\}$  of states,
- a designated start state  $q_0$ .
- a designated start symbol  $\gamma_0$ .
- a subset  $\sigma$  of  $(Q \times (\Sigma \cup \{\epsilon\}) \times G) \times (Q \times G^*)$  called the transitions,
- a subset  $F$  of  $Q$  called the accepting states.

# Shift-Reduce Parsing

- Let  $G = (V, T, S, P)$  be a grammar. Then the LR parsing technique tries to reduce a string  $w \in T^*$  to the start symbol  $S$ .
- The LR parsing technique uses a finite deterministic stack automaton to analyze a given input string. The stack automaton distinguishes between four actions:
  - In a **shift** action, the next input symbol is shifted onto the top of the stack.
  - In a **reduce** action, the sequence on the top of the stack corresponds to a right-hand side of a production. The automaton reduces this sequence to the left-hand side symbol of the production and pushes the symbol onto the top of the stack.
  - In an **accept** action, the stack only contains the root symbol  $S$ . The automaton terminates successfully.
  - In an **error** action, the automaton discovers that an error has occurred and calls an error recovery routine.

# Example Grammar

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle + \langle \text{Term} \rangle$

$\langle \text{Expression} \rangle ::= \langle \text{Term} \rangle$

$\langle \text{Term} \rangle ::= \langle \text{Term} \rangle \times \langle \text{PrimaryExpression} \rangle$

$\langle \text{Term} \rangle ::= \langle \text{PrimaryExpression} \rangle$

$\langle \text{PrimaryExpression} \rangle ::= \langle \text{Identifier} \rangle$

$\langle \text{PrimaryExpression} \rangle ::= ( \langle \text{Expression} \rangle )$

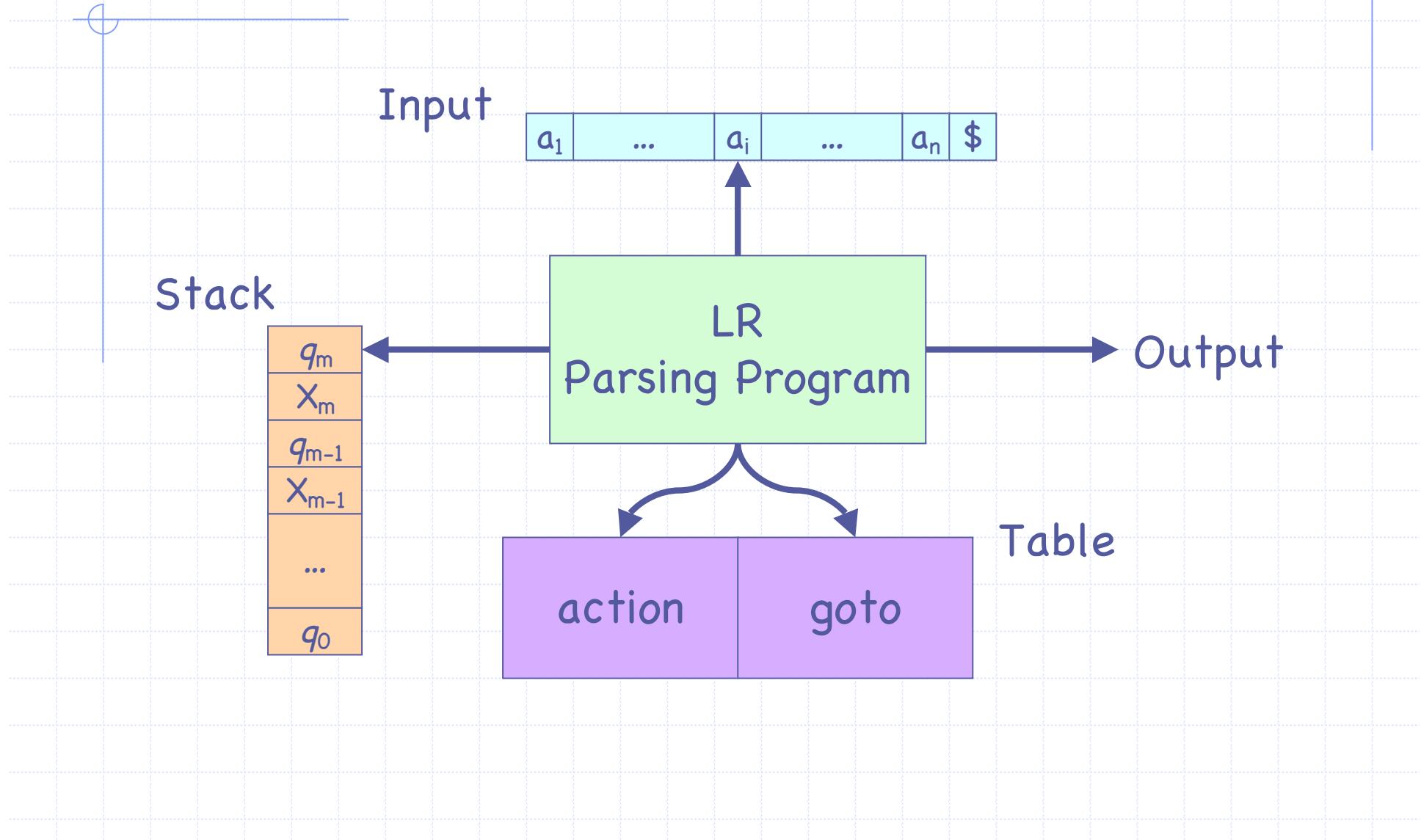
In order to perform LR( $k$ ) parsing, we have to augment the grammar with a new start production:

$\langle \text{Expression}' \rangle ::= \langle \text{Expression} \rangle \$$

$(a + b)^* c$

Stack	Input	Action
	$(a + b)^* c \$$	shift (
(	$a + b)^* c \$$	shift a
a (	$+ b)^* c \$$	reduce $\langle PE \rangle ::= \langle Identifier \rangle$
$\langle PE \rangle ($	$+ b)^* c \$$	reduce $\langle T \rangle ::= \langle PE \rangle$
$\langle T \rangle ($	$+ b)^* c \$$	reduce $\langle E \rangle ::= \langle T \rangle$
$\langle E \rangle ($	$+ b)^* c \$$	shift +
$+ \langle E \rangle ($	$b)^* c \$$	shift b
$b + \langle E \rangle ($	$)^* c \$$	reduce $\langle PE \rangle ::= \langle Identifier \rangle$
$\langle PE \rangle + \langle E \rangle ($	$)^* c \$$	reduce $\langle T \rangle ::= \langle PE \rangle$
$\langle T \rangle + \langle E \rangle ($	$)^* c \$$	reduce $\langle E \rangle ::= \langle E \rangle + \langle T \rangle$
$\langle E \rangle ($	$)^* c \$$	shift )
$) \langle E \rangle ($	$* c \$$	reduce $\langle PE \rangle := (\langle E \rangle)$
$\langle PE \rangle$	$* c \$$	reduce $\langle T \rangle ::= \langle PE \rangle$
$\langle T \rangle$	$* c \$$	shift *
$* \langle T \rangle$	$c \$$	shift c
$c * \langle T \rangle$	$\$$	reduce $\langle PE \rangle ::= \langle Identifier \rangle$
$\langle PE \rangle * \langle T \rangle$	$\$$	reduce $\langle T \rangle ::= \langle T \rangle * \langle PE \rangle$
$\langle T \rangle$	$\$$	reduce $\langle E \rangle ::= \langle T \rangle$
$\langle E \rangle$	$\$$	accept

# Model of an LR Parser



# Elements of an LR Parser Table

An LR parser uses four kinds of actions:

- $sn$ : Shift into state  $n$ ,
- $gn$ : Goto state  $n$ ,
- $rk$ : Reduce rule  $k$ , and
- accept: Stop (success)

Errors are denoted by a blank entries in the table.

# A LR Parsing Table

$\langle \text{Expression} \rangle ::= \langle \text{Expression} \rangle + \langle \text{Term} \rangle$	(1)
$\langle \text{Expression} \rangle ::= \langle \text{Term} \rangle$	(2)
$\langle \text{Term} \rangle ::= \langle \text{Term} \rangle \times \langle \text{PrimaryExpression} \rangle$	(3)
$\langle \text{Term} \rangle ::= \langle \text{PrimaryExpression} \rangle$	(4)
$\langle \text{PrimaryExpression} \rangle ::= \langle \text{Identifier} \rangle$	(5)
$\langle \text{PrimaryExpression} \rangle ::= (\langle \text{Expression} \rangle)$	(6)

	$\langle \text{Id} \rangle$	+	*	(	)	\$	$\langle E \rangle$	$\langle T \rangle$	$\langle PE \rangle$
0	s4				s5		g1	g2	g3
1			s6			accept			
2		r2	s7			r2	r2		
3		r4	r4			r4	r4		
4		r5	r5			r5	r5		
5	s4				s5		g8	g2	g3
6	s4				s5			g9	g3
7	s4				s5				g10
8			s6			s11			
9		r1	s7			r1	r1		
10		r3	r3			r3	r3		
11		r6	r6			r6	r6		

# LR(0) Grammars

- An LR( $k$ ) parser uses the contents of its stack and the next  $k$  input tokens to decide, which action to take.
- In practice,  $k > 1$  is not used, because this would result in huge parsing tables. Moreover, most modern programming languages can be described by LR(1) grammars.
- LR(0) grammars are grammars that can be parsed by looking at the stack only, making shift/reduce decisions without any lookahead.

# LR(0) Item

- LR(0) item:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, the production ( $\langle \text{Expression} \rangle$ ,  $\langle \text{Expression} \rangle + \langle \text{Term} \rangle$ ) yields four items:

( $\langle \text{Expression} \rangle$ , •  $\langle \text{Expression} \rangle + \langle \text{Term} \rangle$ )

( $\langle \text{Expression} \rangle$ ,  $\langle \text{Expression} \rangle$  • +  $\langle \text{Term} \rangle$ )

( $\langle \text{Expression} \rangle$ ,  $\langle \text{Expression} \rangle +$  •  $\langle \text{Term} \rangle$ )

( $\langle \text{Expression} \rangle$ ,  $\langle \text{Expression} \rangle + \langle \text{Term} \rangle$  •)

- An LR(0) item set is a collection of LR(0) items.

# **closure( $I$ )**

- If  $I$  is a set of LR(0) items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of LR(0) items constructed from  $I$  by the following two rules:
  - Every LR(0) item in  $I$  is also in  $\text{closure}( I )$ .
  - If  $(A, w_1 \bullet B w_2)$  is in  $\text{closure}( I )$  and  $(B, w_3)$  is a production, then add  $(B, \bullet w_3)$  to  $I$ , if it is not already there. Repeat this step until no more new items can be added to  $\text{closure}( I )$ .

# $\text{goto}( I, X )$

- The function  $\text{goto}( I, X )$  is defined to be the closure of the set of all items  $(A, w_1 X \bullet w_2)$  such that  $(A, w_1 \bullet X w_2)$  is in  $I$ .

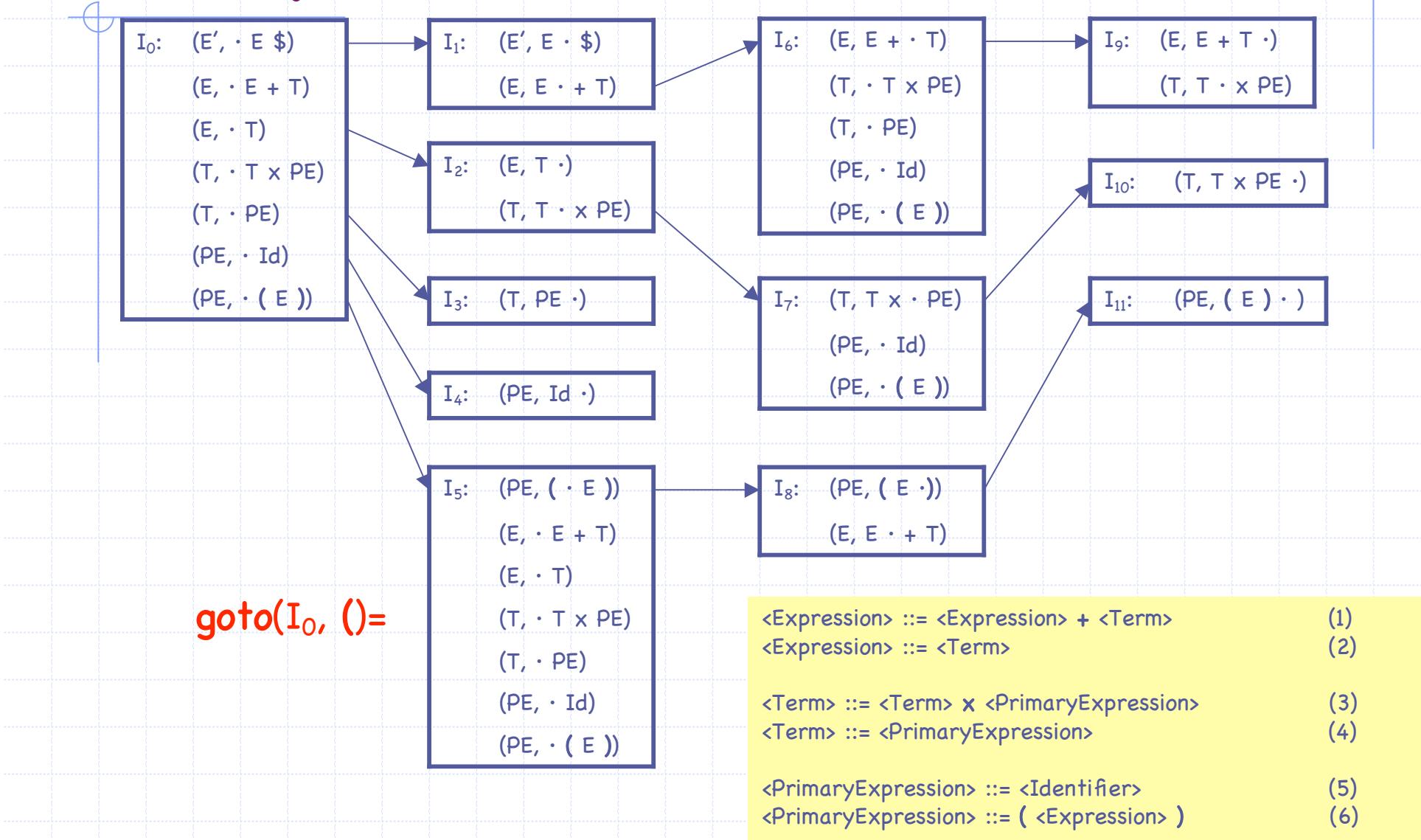
$$\text{goto}( I, X ) = \text{closure}(\{(A, w_1 X \bullet w_2) \mid (A, w_1 \bullet X w_2) \in I\})$$

# Canonical LR(0) Collection

Let  $G = (V, T, S, P)$  be a grammar.

- Augment the grammar  $G$  with an auxiliary production  $(S', S \$)$ .
- set  $T = \{ \text{closure}( (S', \bullet S \$) ) \}$
- repeat
  - for each item set  $I$  in  $T$ 
    - for each LR(0) item  $(A, w_1 \bullet X \ w_2)$  in  $I$ 
      - let  $J$  be  $\text{goto}( I, X )$
      - if  $J \neq \emptyset$  and  $J \notin T$ 
        - then add  $J$  to  $T$  add new state
  - until no more elements have been added to  $T$

# Example: $T =$



# LR(0) Parser Construction

Let  $G = (V, T, S, P)$  be a grammar.

- Augment the grammar  $G$  with an auxiliary production  $(S', S \$)$ .
- Construct  $I = \{I_0, I_1, \dots, I_n\}$ , the canonical LR(0) set for augmented  $G$ .
- State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follows:
  - ◆ If  $(A, w_1 \bullet a w_2)$  with  $a \in T$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift j" (i.e.,  $sj$ ).
  - ◆ If  $(A, w \bullet)$  with  $w \in (V - T)$ ,  $w \neq S'$  is in  $I_i$ , then set  $\text{action}[i, w]$  to "reduce  $(A, w)$ " (i.e.,  $rk$  and  $k$  is the number of  $(A, w)$ ).
  - ◆ If  $(S', S \bullet \$)$  is in , then set  $\text{action}[i,\$]$  to "accept".
- The goto transitions for state  $i$  are constructed for all nonterminals  $X$  using the rule: If  $\text{goto}(I_i, X) = I_j$ , then  $\text{goto}[i, X] = j$ .
- All other entries are set to "error".
- The start state of the parser is the one constructed from the LR(0) item set containing  $(S', \bullet S \$)$ .

# A LR(0) Parsing Table

$\langle E \rangle ::= \langle E \rangle + \langle T \rangle$  (1)

$\langle E \rangle ::= \langle T \rangle$  (2)

$\langle T \rangle ::= \langle T \rangle \times \langle PE \rangle$  (3)

$\langle T \rangle ::= \langle PE \rangle$  (4)

$\langle PE \rangle ::= \langle Id \rangle$  (5)

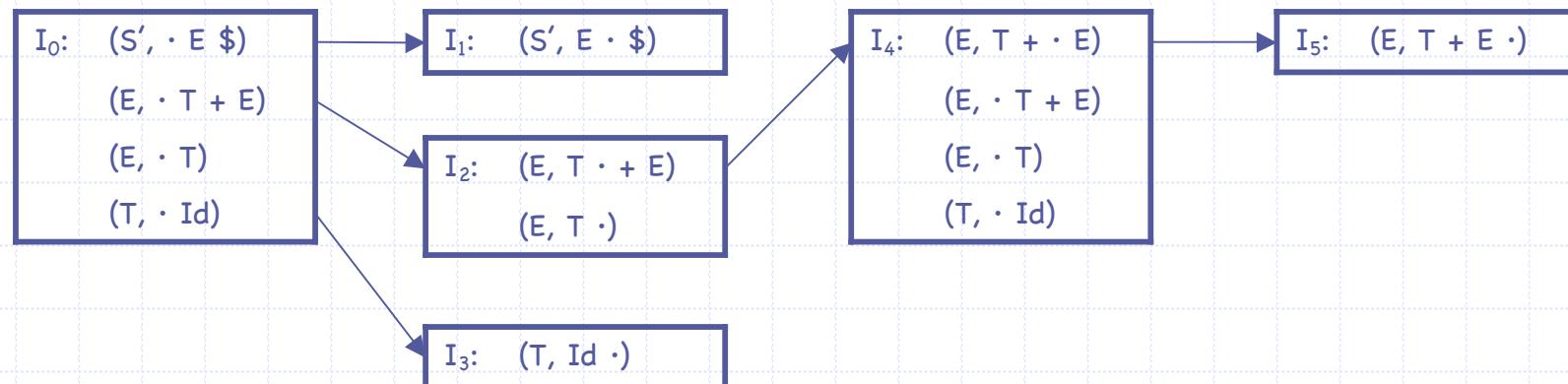
$\langle PE \rangle ::= ( \langle E \rangle )$  (6)

	<Id>	+	*	(	)	\$	<E>	<T>	<PE>
0	s4			s5			g1	g2	g3
1		s6				accept			
2	r2	r2	s7/r <sub>2</sub>	r2	r2	r2			
3	r4	r4	r4	r4	r4	r4			
4	r5	r5	r5	r5	r5	r5			
5	s4			s5			g8	g2	g3
6	s4			s5				g9	g3
7	s4			s5					g10
8		s6			s11				
9	r1	r1	s7/r1	r1	r1	r1			
10	r3	r3	r3	r3	r3	r3			
11	r6	r6	r6	r6	r6	r6			

# LR(0) Conflict

- 1:  $(S', \cdot E \$)$
- 2:  $(E, T + E)$
- 3:  $(E, T)$
- 4:  $(T, Id)$

Consider the following augmented grammar:



	Id	+	\$	E	T
0	s3			g1	g2
1			accept		
2	r3	s4,r3	r3		
3	r4	r4	r4	g4	g2
4	s3				
5	r2	r2	r2		

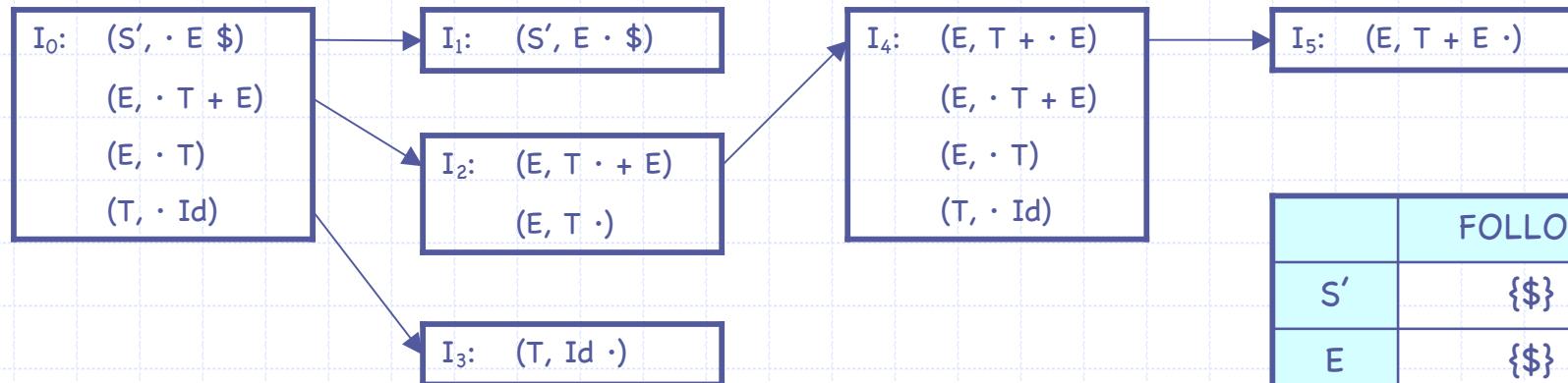
# SLR Parser Construction

Let  $G = (V, T, S, P)$  be a grammar.

- Augment the grammar  $G$  with an auxiliary production  $(S', S \$)$ .
- Construct  $I = \{I_0, I_1, \dots, I_n\}$ , the canonical LR(0) set for augmented  $G$ .
- State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follows:
  - ◆ If  $(A, w_1 \bullet a w_2)$  with  $a \in T$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift j" (i.e.,  $sj$ ).
  - ◆ If  $(A, w \bullet)$  with  $w \in (V - T)$ ,  $w \neq S'$  is in  $I_i$ , then set  $\text{action}[i, w]$  to "reduce  $(A, w)$ " for all  $a$  in  $\text{ FOLLOW}(A)$ ;  $A \neq S'$ .
  - ◆ If  $(S', S \bullet \$)$  is in , then set  $\text{action}[i,\$]$  to "accept".
- The goto transitions for state  $i$  are constructed for all nonterminals  $X$  using the rule: If  $\text{goto}(I_i, X) = I_j$ , then  $\text{goto}[i, X] = j$ .
- All other entries are set to "error".
- The start state of the parser is the one constructed from the LR(0) item set containing  $(S', \bullet S \$)$ .

# SLR Parser

Consider the following augmented grammar:



- 1: (S', E \$)
- 2: (E, T + E)
- 3: (E, T)
- 4: (T, Id)

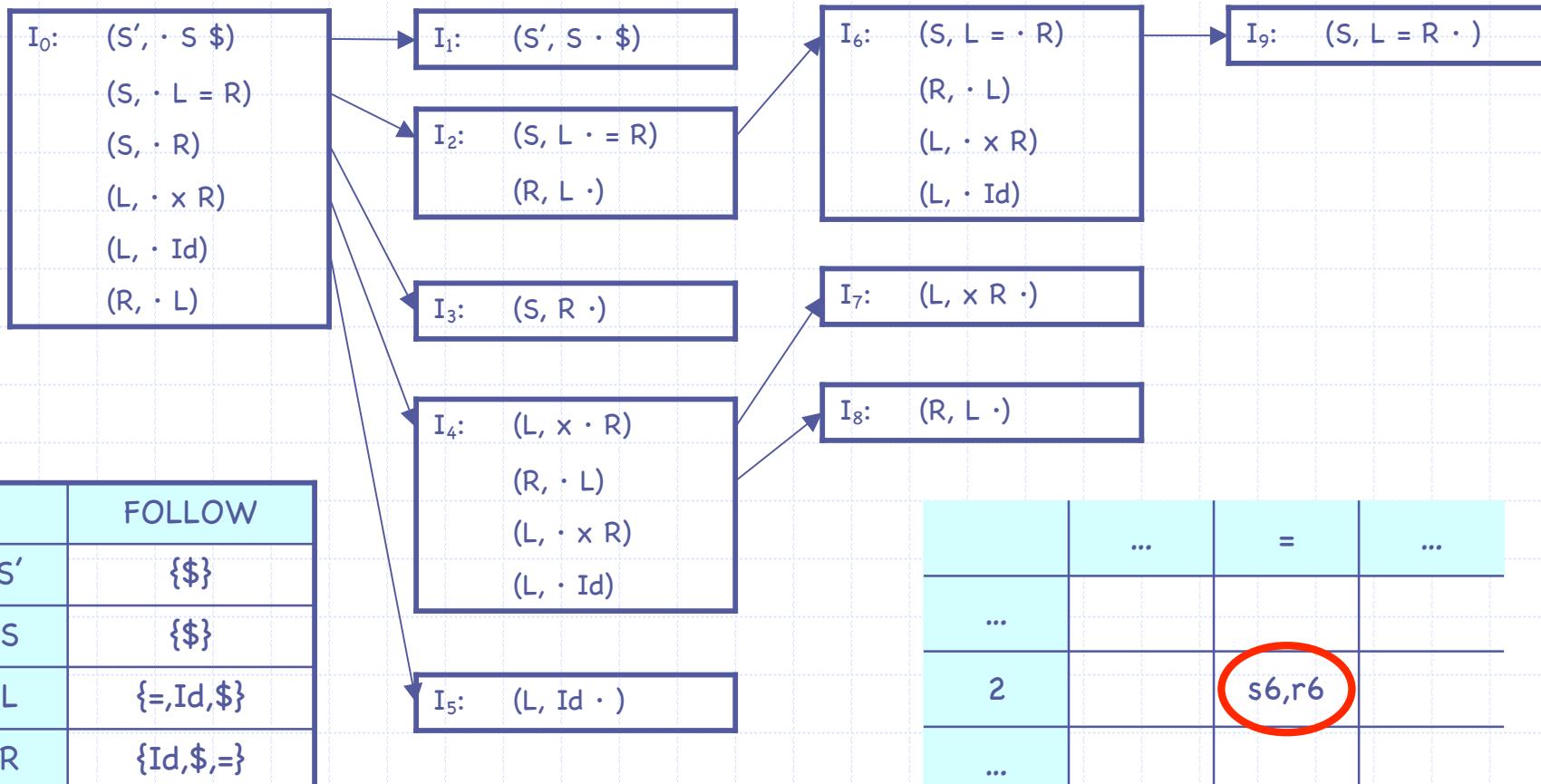
	FOLLOW
S'	{\$}
E	{\$}
T	{+, \$}

	Id	+	\$	E	T
0	s3			g1	g2
1			accept		
2		s4	r3		
3		r4	r4	g4	g2
4	s3				
5			r2		

# SLR Conflict

Consider the following augmented grammar:

- 1:  $(S', S \$)$
- 2:  $(S, L = R)$
- 3:  $(S, R)$
- 4:  $(L, x R)$
- 5:  $(L, Id)$
- 6:  $(R, L)$



# LR(1) Item

- LR(1) item:

An LR(1) item of a grammar G is a production of G with a dot at some position of the right side and a lookahead symbol. For example

(<Expression>, • <Expression> + <Term>; ))

An LR(1) item ( $A, w_1 \bullet X w_2; a$ ) indicates that the sequence  $w_1$  is the top of the stack, and at the head of the input is a string derivable from  $Xa$ .

- An LR(1) item set is a collection of LR(1) items.

# **closure( $I$ )**

- If  $I$  is a set of LR(1) items for a grammar  $G$ , then  $\text{closure}(I)$  is the set of LR(1) items constructed from  $I$  by the following two rules:
  - Every LR(1) item in  $I$  is also in  $\text{closure}( I )$ .
  - If  $(A, w_1 \bullet B w_2; a)$  is in  $\text{closure}( I )$  and  $(B, w_3)$  is a production, then for each  $b \in \text{FIRST}(w_2 a)$  add  $(B, \bullet w_3; b)$  to  $I$ , if it is not already there. Repeat this step until no more new items can be added to  $\text{closure}( I )$ .

# goto( $I$ , $X$ )

- The function  $\text{goto}( I, X )$  is defined to be the closure of the set of all items  $(A, w_1 X \bullet w_2; a)$  such that  $(A, w_1 \bullet X w_2; a)$  is in  $I$ .

$$\text{goto}( I, X ) = \text{closure}(\{(A, w_1 X \bullet w_2; a) \mid (A, w_1 \bullet X w_2; a) \in I\})$$

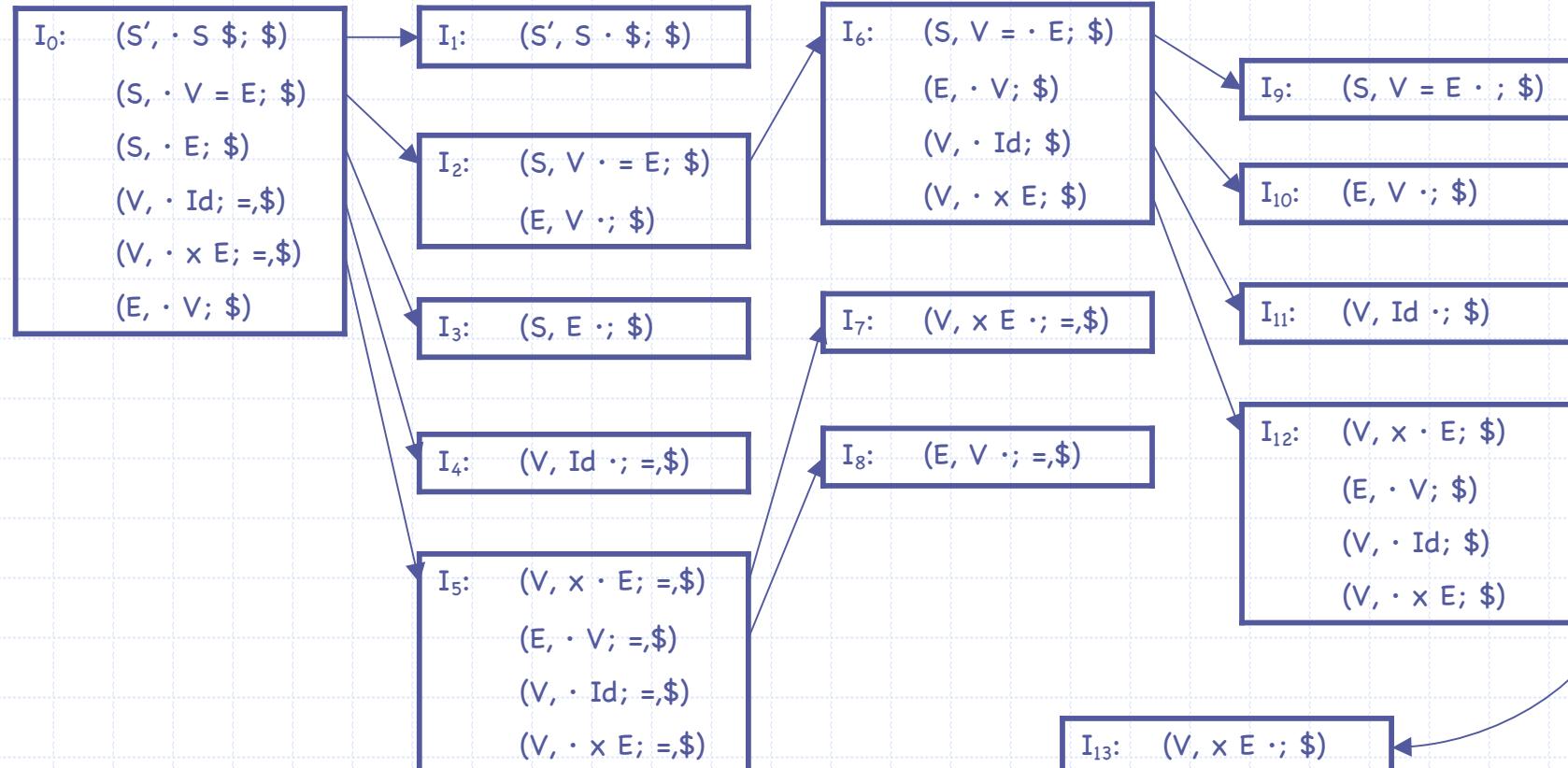
# Canonical LR(1) Collection

Let  $G = (V, T, S, P)$  be a grammar.

- Augment the grammar  $G$  with an auxiliary production  $(S', S \$)$ .
- set  $T = \{ \text{closure}( (S', \bullet S \$; \$) ) \}$
- repeat
  - for each item set  $I$  in  $T$ 
    - for each LR(1) item  $(A, w_1 \bullet X w_2; a)$  in  $I$ 
      - let  $J$  be  $\text{goto}( I, X )$
      - if  $J \neq \emptyset$  and  $J \notin T$ 
        - then add  $J$  to  $T$  add new state
  - until no more elements have been added to  $T$

# Example:

- 1:  $(S', S \$)$
- 2:  $(S, V = E)$
- 3:  $(S, E)$
- 4:  $(E, V)$
- 5:  $(V, Id)$
- 6:  $(V, \times E)$



# LR(1) Parser Construction

Let  $G = (V, T, S, P)$  be a grammar.

- Augment the grammar  $G$  with an auxiliary production  $(S', S \$)$ .
- Construct  $I = \{I_0, I_1, \dots, I_n\}$ , the canonical LR(1) set for augmented  $G$ .
- State  $i$  is constructed from  $I_i$ . The parsing actions for state  $i$  are determined as follows:
  - ◆ If  $(A, w_1 \bullet a w_2; b)$  with  $a \in T$  is in  $I_i$  and  $\text{goto}(I_i, a) = I_j$ , then set  $\text{action}[i, a]$  to "shift j" (i.e.,  $sj$ ).
  - ◆ If  $(A, w \bullet; b)$  with  $w \in (V - T)$ ,  $w \neq S'$  is in  $I_i$ , then set  $\text{action}[i, b]$  to "reduce  $(A, w)$ " (i.e.,  $rk$  and  $k$  is the number of  $(A, w)$ ).
  - ◆ If  $(S', S \bullet \$; \$)$  is in , then set  $\text{action}[i,\$]$  to "accept".
- The goto transitions for state  $i$  are constructed for all nonterminals  $X$  using the rule: If  $\text{goto}(I_i, X) = I_j$ , then  $\text{goto}[i, X] = j$ .
- All other entries are set to "error".
- The start state of the parser is the one constructed from the LR(1) item set containing  $(S', \bullet S \$; \$)$ .

# LR(1) Parser Table

- 1: ( $S'$ ,  $S$  \$)
- 2: ( $S$ ,  $V$  =  
E)
- 3: ( $S$ ,  $E$ )
- 4: ( $E$ ,  $V$ )
- 5: ( $V$ , Id)
- 6: ( $V$ ,  $x E$ )

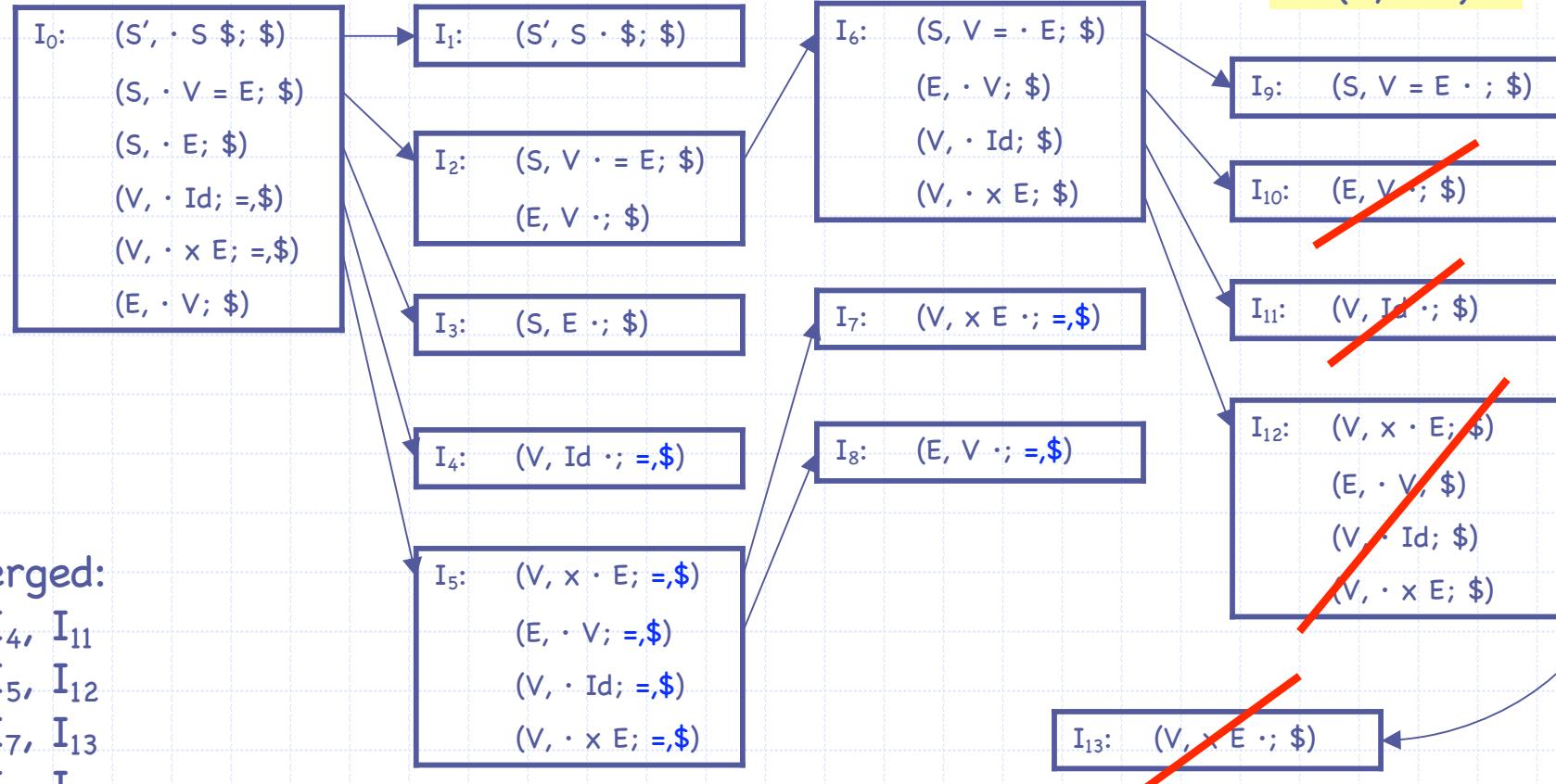
	=	Id	x	\$	S	E	V
0		s4	s5		g1	g3	g2
1				accept			
2	s6			r4			
3				r3			
4	r5			r5			
5		s4	s5			g7	g8
6		s11	s12			g9	g10
7	r6			r6			
8	r4			r4			
9				r2			
10				r4			
11				r5			
12		s11	s12			g13	g10
13				r6			

# Union of LR(1) Item Sets

$\text{union}(A, B) = \{(A, w_1 \bullet w_2; a \cup b) \mid$   
 $(A, w_1 \bullet w_2; a) \in A \text{ and}$   
 $(A, w_1 \bullet w_2; b) \in B\}$

# Merged States

- 1:  $(S', S \$)$
- 2:  $(S, V = E)$
- 3:  $(S, E)$
- 4:  $(E, V)$
- 5:  $(V, Id)$
- 6:  $(V, \times E)$



merged:

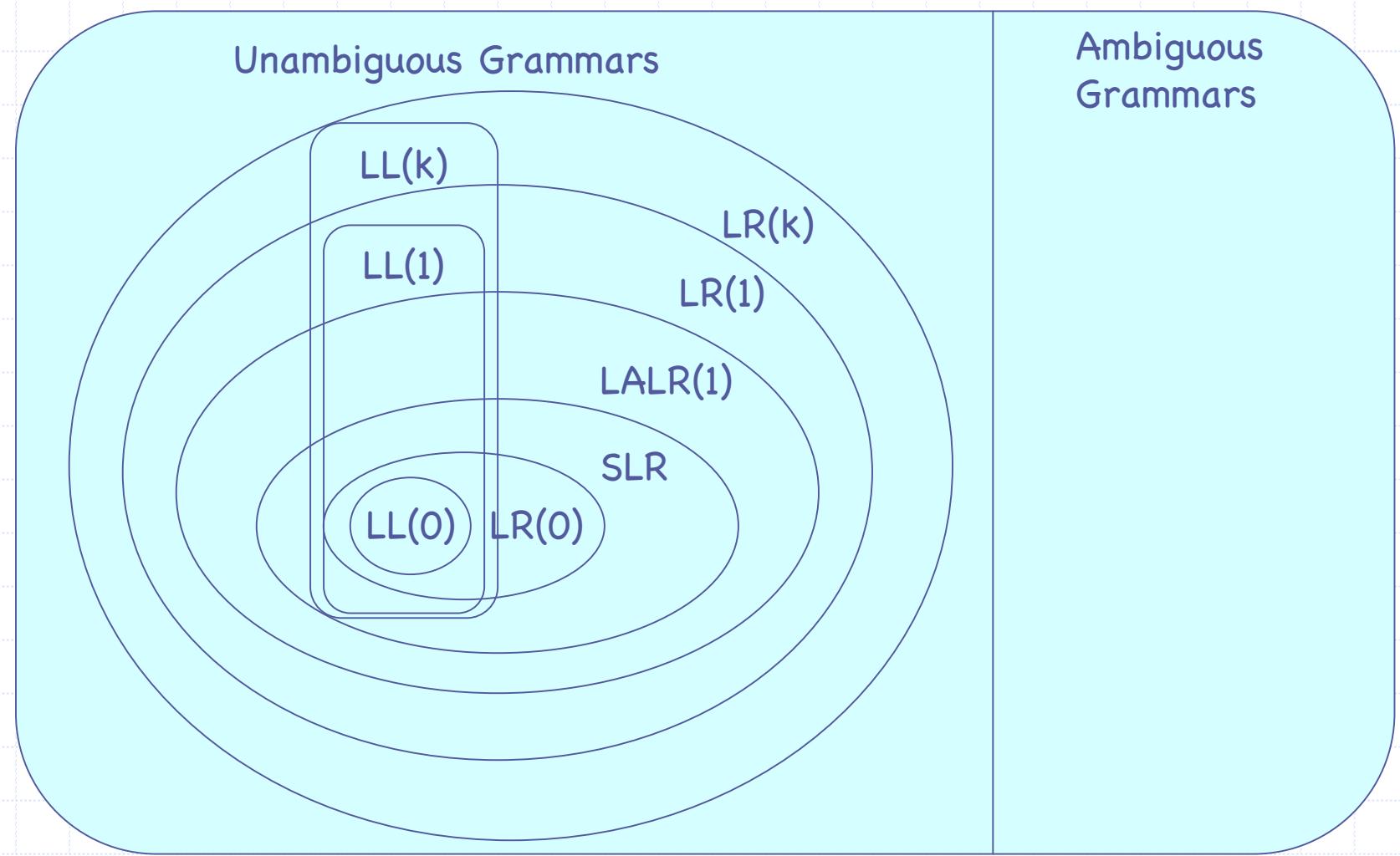
- $I_4, I_{11}$
- $I_5, I_{12}$
- $I_7, I_{13}$
- $I_8, I_{10}$

# LALR(1) Parser Table

- 1: ( $S'$ ,  $S \ $$ )
- 2: ( $S$ ,  $V = E$ )
- 3: ( $S$ ,  $E$ )
- 4: ( $E$ ,  $V$ )
- 5: ( $V$ ,  $Id$ )
- 6: ( $V$ ,  $x E$ )

	=	Id	x	\$	S	E	V
0		s4	s5		g1	g3	g2
1				accept			
2	s6			r4			
3				r3			
4	r5			r5			
5		s4	s5			g7	g8
6		s4	s5			g9	g8
7	r6			r6			
8	r4			r4			
9				r2			

# Hierarchy of Grammar Classes



# Parser Generators

- Java Compiler Compiler:
  - JavaCC is an LL( $k$ ) parser generator that generates a recursive-descent parser.
  - By default, JavaCC generates an LL(1) parser. However, JavaCC offers a syntactic and semantic lookahead capability to resolve ambiguities.
- Yet Another Compiler Compiler:
  - YACC is an LALR(1) parser generator that generates a table-based bottom-up parser. The first version of YACC was created in the early 1970's.
  - YACC and its GNU version bison are written in C and they generate C-code. Versions for other languages exist also (e.g., SableCC).

# JavaCC Grammar File

```
<JavaCC Input> ::=  
    <JavaCC-Options>  
    PARSER_BEGIN ( <Identifier> )  
    <Java-Compilation-Unit>  
    PARSER_END ( <Identifier> )  
    <Production>*  
    <EOF>
```

# JavaCC Production

```
<Production> ::=  
    <Java-Type> <Identifier> ( <Java-Parameter-List> ) :  
    <Java-Block>  
    { (<Expansion> | <Expansion>)* }*
```

```
void Rule( Grammar aGrammar ) :  
{  
    Token lhs;  
}  
{  
    lhs=<ID> <COLON> { aGrammar.addNonterminal( lhs.image ); }  
    RHSs( lhs.image, aGrammar ) <SEMICOLON>  
}
```

Rule ::= <Identifier> : <RHSs> ;

# YACC Grammar File

<YACC-Input> ::= <Declarations>  
%%  
<Productions>  
%%  
<C-Routines>

# YACC Declarations

- `%{ C-Declarations %}`
- `%start <Identifier>`
- `%token [<<TypeName>>] <Identifier> [<Integer>]`
- `%type <<TypeName>> <Identifier>`
- `%union { <Member-Declaration-List> }`
- `%left <Operator>`  
`%right <Operator>`  
`%nonassoc <Operator>`

# YACC Production

<Production> ::=

<Identifier> : (<RHS> | <RHS>\*)\* [ ; ]

```
%token <Ids> ID      1  
%token COLON          2  
%token SEMICOLON      3
```

```
%union { String* Ids; }
```

```
%type <Ids> Rule  
%type <Ids> RHSs
```

```
%%
```

```
Rule :  
    ID COLON RHSs SEMICOLON  
    { $$ = $3; }
```

Rule ::= <Identifier> : <RHSs> ;

RHS's may also contain:

**%prec <Terminal>**

# Syntax-Directed Definition

- A syntax-directed definition is a generalization of a context-free grammar in which each grammar symbol has an associated set of attributes, partitioned into two categories:
  - Synthesized Attributes
  - Inherited Attributes
- In a syntax-directed definition, each production  $(A, w)$  is equipped with a set of semantic rules of the form  $b = f(c_1, c_2, \dots, c_n)$  where  $f$  is a function, and either
  - $b$  is a synthesized attribute of  $A$  and  $c_1, c_2, \dots, c_n$  are attributes belonging the grammar symbols in  $w$ , or
  - $b$  is an inherited attribute of one of the grammar symbols in  $w$  and  $c_1, c_2, \dots, c_n$  are attributes belonging to the grammar symbols in  $w$ .

# Compiler Passes

- Several compilation phases are usually implemented in a single pass consisting of reading an input file and writing an output file.
- It is desirable to have relatively few phases, since the communication between passes usually utilizes intermediate files.
- However, the minimal number of passes is determined by the equation  $b = f( c_1, c_2, \dots, c_n )$ , which expresses the attribute dependencies between  $b$  and  $c_1, c_2, \dots, c_n$ .

# A Simple Calculator

```
<Lines> ::= | <Expression> \n <Lines>
```

```
<Expression> ::= <Expression> + <Expression>
| <Expression> - <Expression>
| <Expression> x <Expression>
| <Expression> / <Expression>
| ( <Expression> )
| - <Expression>
| <Number>
```

This grammar is ambiguous and contains left-recursive rules.

# JavaCC: Step 1

`<Lines> ::= ( <Expression> \n )*`

`<Expression> ::= <Term> ( (+| -) <Term>)*`

`<Term> ::= <Primary> ( (x | /) <Primary> )*`

`<Primary> ::= ( <Expression> )  
| - <Expression>  
| <Number>`

# JavaCC: Step 2

## Lexical Analysis:

```
PARSER_BEGIN (SimpleCalculator)

public class SimpleCalculator
{
    public static void main( String[] Args ) {
        try {
            new SimpleCalculator(System.in).Start(); }
        catch (ParseException e) {
            System.out.println(e.toString()); } }

PARSER_END (SimpleCalculator)

TOKEN : { < NUMBER : ([0"-9])+ > }

SKIP : { < " " | "\t" | "\r" > }
```

## Grammar:

```
void Start() :
{}

{ Lines() <EOF> }

void Lines() :
{}

{ ( Expression() "\n" )* }

void Expression() :
{}

{ Term() (( "+" | "-" ) Term())* }

void Term() :
{}

{ Primary() (( "x" | "/" ) Primary())* }

void Primary() :
{}

{ "(" Expression() ")" | <NUMBER> | "-" Expression() }
```

# Error Message

```
javacc SimpleCalculator.jj
Java Compiler Compiler Version 3.2 (Parser Generator)
(type "javacc" with no arguments for help)
Reading from file SimpleCalculator.jj . .
Warning: Choice conflict in (...)*
construct at line 41, column 10.
    Expansion nested within construct and expansion following
    construct
        have common prefixes, one of which is: "+"
        Consider using a lookahead of 2 or more for nested expansion.
Warning: Choice conflict in (...)*
construct at line 45, column 13.
    Expansion nested within construct and expansion following
    construct
        have common prefixes, one of which is: "x"
        Consider using a lookahead of 2 or more for nested expansion.
Parser generated with 0 errors and 2 warnings.
```

# FOLLOW-Conflict

GrammarCheck:

FOLLOW conflict on PLUS

ExpressionRest ::= <<EPSILON>>

ExpressionRest ::= PLUS Term ExpressionRest

FOLLOW conflict on MINUS

ExpressionRest ::= <<EPSILON>>

ExpressionRest ::= MINUS Term ExpressionRest

FOLLOW conflict on TIMES

TermRest ::= <<EPSILON>>

TermRest ::= TIMES Primary TermRest

FOLLOW conflict on DIV

TermRest ::= <<EPSILON>>

TermRest ::= DIV Primary TermRest

Read: return on e, continue on "+"

# JavaCC: Step 2a

`<Lines> ::= ( <Expression> \n )*`

`<Expression> ::= <Term> ( (+| -) <Term>)*`

`<Term> ::= <Primary> ( (x | /) <Primary> )*`

`<Primary> ::= ( <Expression> )  
| - <Primary>  
| <Number>`

# JavaCC: Step 2a - No Warning

## Lexical Analysis:

```
PARSER_BEGIN (SimpleCalculator)

public class SimpleCalculator
{
    public static void main( String[] Args ) {
        try {
            new SimpleCalculator(System.in).Start(); }
        catch (ParseException e) {
            System.out.println(e.toString()); } }

PARSER_END (SimpleCalculator)

TOKEN : { < NUMBER : ([0"-9])+ > }

SKIP : { < " " | "\t" | "\r" > }
```

## Grammar:

```
void Start() :
{}

{ Lines() <EOF> }

void Lines() :
{}

{ ( Expression() "\n" )* }

void Expression() :
{}

{ Term() (( "+" | "-" ) Term())* }

void Term() :
{}

{ Primary() (( "x" | "/" ) Primary())* }

void Primary() :
{}

{ "(" Expression() ")" | <NUMBER> | "-" Primary() }
```

# JavaCC: Step 3 – Attributes

```
void Start() : {} { Lines() <EOF> }

void Lines() :
{ int value; }
{ ( value=Expression() "\n" { System.out.println( "-> " + value ); } )* }

int Expression() :
{ int value; int top; }
{ value=Term( 0, 0 ) (( "+" { top = 1; } | "-" { top = 2; } ) value=Term( value, top ))*
  { return value; }
}
```

# JavaCC: Step 3 - Attributes II

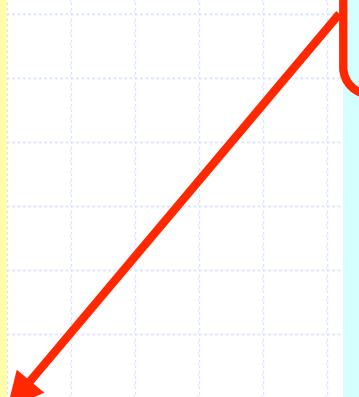
```
(1) int Term( int left, int op ) :  
{ int value; int pop; }  
{ value=Primary( 0, 0 ) (("x" { pop = 1; } | "/" {pop = 2; }) value=Primary( value, pop ))*  
{ switch ( op ) {  
    case 1: value = left + value; break;  
    case 2: value = left - value; break; }  
return value; }  
}  
  
int Primary( int left, int op ) :  
{ int value; Token n; }  
{ ( "(" value=Expression() ")" | "-" value=Primary( 0, 0 ) { value = value * -1; }  
| n=<NUMBER> { value = (new Integer( n.image )).intValue(); } )  
{ switch ( op ) {  
    case 1: value = left * value; break;  
    case 2: value = left / value; break; }  
return value; }  
}
```

# Trace

```
Call: Start
Call: Lines
4 × 2 - - 1
Call: Expression
Call: Term
Call: Primary
Consumed token: <<NUMBER>>: "4"
Return: Primary
Consumed token: <"x">
Call: Primary
Consumed token: <<NUMBER>>: "2"
Return: Primary
Return: Term
Consumed token: <"-">
Call: Term
Call: Primary
Consumed token: <"-">
Call: Primary
Consumed token: <<NUMBER>>: "1"
Return: Primary
Return: Primary
Return: Term
Return: Expression
Consumed token: <"\n": "
">
-> 9
```

4 × 2 - - 1  
-> 9

4 × 2 + 1  
-> 9  
2 × 2 + 3 × 3  
-> 13  
2 × (2 + 3) × 3  
-> 30  
2 × (2 + 3) / 3  
-> 3  
^Z



# JavaCC LOOKAHEAD

- A local lookahead specification is used to temporarily increase the number of tokens to be considered when the generated parser has to choose the next RHS.

```
<Local-Lookahead> ::=  
    LOOKAHEAD ( [ <Java-Integer-Literal> ] [ , ]  
                [ <Expansion-Choices> ] [ , ]  
                [ { <Java-Expression> } ] )
```

Note: If syntactic lookahead is used and no lookahead limit is provided, then the lookahead limit is set to the largest integer value (i.e., infinite lookahead).

# Example: Common Prefixes

```
void TermC() :
```

```
{}
```

```
{
```

Form()

|  
LOOKAHEAD ( 2 )

<LP> <RES> <IDENTIFIER> <ARROW> Term() <RP>

<LP> Term() <RP>

```
}
```

LL(2) at this point only

# Example: Context

```
void FormE() :  
{  
{  
    <EMPTY>  
  
    // Identifier not followed by "="  
    LOOKAHEAD( { getToken(1).kind == IDENTIFIER &&  
                getToken(2).kind != EQUAL } )  
    <IDENTIFIER>  
}
```

# YACC Version

```
%token NUMBER
%left '+' '-'
%left 'x' '/'
%right UMINUS
%start lines
%%

lines :    | lines expression '\n'

expression : expression '+' expression
            | expression '-' expression
            | expression 'x' expression
            | expression '/' expression
            | '(' expression ')'
            | '-' expression %prec UMINUS
            | NUMBER
```

# YACC Specification

```
%{  
#include <stdio.h>  
#define YYSTYPE int  
%}  
  
%token NUMBER  
%start lines  
  
%%  
  
lines : | lines expression '\n'  
  
expression : expression '+' term  
| expression '-' term  
| term  
  
term : term 'x' primary  
| term '/' primary  
| primary  
  
primary : '(' expression ')'  
| '-' expression  
| NUMBER  
  
%%
```

```
int main() { yyparse(); }  
  
void yyerror( char* aMessage ) {  
    printf( "Syntax error: %s\n", aMessage );  
}  
  
int yylex() {  
    int c;  
  
    while ( ((c = getchar()) == ' ') || (c == '\t') );  
  
    if ( c == '.' ) return -1;  
  
    if ( isdigit( c ) ) {  
        yylval=0;  
        while ( isdigit( c ) ) {  
            yylval = yylval * 10 + (c - '0');  
            c = getchar(); }  
  
        ungetc( c, stdin );  
        return NUMBER;  
    }  
    else  
        return c;  
}
```

# Errors Induced by UMINUS

- 8 shift/reduce conflicts:

6: shift/reduce conflict (shift 13, reduce 5) on 'x'

6: shift/reduce conflict (shift 14, reduce 5) on '/'

state 6

expression : term . (5)

term : term . 'x' primary (6)

term : term . '/' primary (7)

'x' shift 13

'/' shift 14

'\n' reduce 5

'+' reduce 5

'-' reduce 5

')' reduce 5

lines :  
| lines expression '\n'

expression :  
  expression '+' term  
| expression '-' term  
| term

term :  
  term 'x' primary  
| term '/' primary  
| primary

primary :  
  '(' expression ')'  
| '-' expression  
| NUMBER

# YACC Specification

```
%{  
#include <stdio.h>  
#define YYSTYPE int  
%}  
  
%token NUMBER  
%start lines  
  
%%  
  
lines : | lines expression '\n'  
  
expression : expression '+' term  
| expression '-' term  
| term  
  
term : term 'x' primary  
| term '/' primary  
| primary  
  
primary : '(' expression ')'  
| '-' primary  
| NUMBER  
  
%%
```

```
int main() { yyparse(); }  
  
void yyerror( char* aMessage ) {  
    printf( "Syntax error: %s\n", aMessage );  
}  
  
int yylex() {  
    int c;  
  
    while ( ((c = getchar()) == ' ') || (c == '\t') );  
  
    if ( c == '.' ) return -1;  
  
    if ( isdigit( c ) ) {  
        yylval=0;  
        while ( isdigit( c ) ) {  
            yylval = yylval * 10 + (c - '0');  
            c = getchar(); }  
  
        ungetc( c, stdin );  
        return NUMBER;  
    }  
    else  
        return c;  
}
```

# Yacc Step 2 – Attributes

lines :

```
| lines expression '\n' { printf( "-> %d\n", $2 ); }
```

expression :

```
expression '+' term { $$ = $1 + $3; }
| expression '-' term { $$ = $1 - $3; }
| term
```

term :

```
term 'x' primary { $$ = $1 * $3; }
| term '/' primary { $$ = $1 / $3; }
| primary
```

primary :

```
(' expression ')' { $$ = $2; }
| '-' primary      { $$ = - $2; }
| NUMBER
```

# 0-Attributes

- We can simulate inherited attributes in YACC (i.e., in an LALR(1)-parser) by providing so-called 0-attributes.
- Now, we have to introduce attribute types:

```
%union
{
    int N;
    struct { int n; int op; } LEFT;
}

%token <N> NUMBER

%type <N> expression term primary
```

# Using 0-Attributes

lines :

```
| lines { $<LEFT>$.op = 0; } expression '\n' { printf( "-> %d\n", $3 ); }
```

expression :

```
expression '+' { $<LEFT>$.n = $1; $<LEFT>$.op = 1; } term { $$ = $4; }
| expression '-' { $<LEFT>$.n = $1; $<LEFT>$.op = 2; } term { $$ = $4; }
| term
```

term :

```
term 'x' { $<LEFT>$.n = $1; $<LEFT>$.op = 3; } primary
{ $$ = eval( $<LEFT>0.n, $4, $<LEFT>0.op ); }
| term '/' { $<LEFT>$.n = $1; $<LEFT>$.op = 4; } primary
{ $$ = eval( $<LEFT>0.n, $4, $<LEFT>0.op ); }
| primary
```

primary :

```
(' expression ')' { $$ = eval( $<LEFT>0.n, $2, $<LEFT>0.op ); }
| '-' primary { $$ = eval( $<LEFT>0.n, - $2, $<LEFT>0.op ); }
| NUMBER { $$ = eval( $<LEFT>0.n, $1, $<LEFT>0.op ); }
```

```
int eval( int l, int r, int op )
{
    switch ( op )
    {
        case 0: return r;
        case 1: return l + r;
        case 2: return l - r;
        case 3: return l * r;
        case 4: return l / r;
    }
}
```

# Error Handling

- A compiler has to reject erroneous programs and must be able to handle all occurrences of errors in a meaningful way (that is, it must not crash).
- The quality of error detection, error handling, and error reporting affects greatly a programmer's productivity.
- A good error reporting mechanism facilitates error diagnosis and error correction. Programmers need to be guided towards the solution of the problem.

# Poor Error Message

```
javacc SimpleCalculator.jj
```

```
Java Compiler Compiler Version 3.2 (Parser Generator)
```

```
(type "javacc" with no arguments for help)
```

```
Reading from file SimpleCalculator.jj . . .
```

```
Warning: Choice conflict in (...)*) construct at line 41, column 10.
```

Expansion nested within construct and expansion following  
construct

have common prefixes, one of which is: "+"

Consider using a lookahead of 2 or more for nested expansion.

```
Warning: Choice conflict in (...)*) construct at line 45, column 13.
```

Expansion nested within construct and expansion following  
construct

have common prefixes, one of which is: "x"

Consider using a lookahead of 2 or more for nested expansion.

```
Parser generated with 0 errors and 2 warnings.
```

# Types of Errors

- Lexical errors:
  - Misspelling of an identifier, keyword, or operator
- Syntax errors:
  - Unbalanced parentheses in arithmetic expressions
- Semantic errors:
  - Operators applied to incompatible operands
- Logical errors:
  - Infinite recursions

# Error Handling Goals

- The presence of errors has to be reported as clearly and accurately as possible.
- A compiler has to recover from each error as quickly as possible to be able to detect subsequent errors.
- The error handling mechanism should not introduce new, possibly artificial, errors.
- The error handling process should not significantly slow down the processing of correct programs.

# Error Messages

- An error message has to lead a user to the problem. That is, a good message has to include the cause of the error.
- An error message has to be based on the source code. That is, it should include references to the location of the error, the context of the error, but free of any internal reasons why the error may have occurred.
- Error message must be complete. That is, an error message has to enable a programmer to fix the problem immediately.
- Error messages must be readable (i.e., plain English) and should be restrained and polite. Being a Servant, the compiler has to treat the programmer as Master and gently ask for a clarification.

# Syntax Error Classes

- `if ( a == 1 ) z = 0 else z = 1;`

The error (missing semicolon between 0 and else) is unique and can be located.

- `x = x + * 5;`

The error is ambiguous, but it can be located.

- `x = x * 7 + (a - b) + 2;`

The error can be neither located nor identified.

# Definition of Syntax Error

- A syntax error denotes the minimal Hamming distance between a correct program and the erroneous program. The Hamming distance considers three operations:
  - Delete tokens
  - Insert tokens
  - Replace tokens
- A syntax error is given by the “parser defined error”, which is the earliest detectable location of an error a backtrack-free parser can determine.

# Prefix

Let  $G = (V, T, S, P)$  be a grammar.

- A string  $a$  is called a “prefix” of  $L(G)$  if there exists a string  $b \in T^*$  such that  $ab \in L(G)$ .
- If  $d \notin L(G)$ , then it is possible to determine the longest correct prefix of  $d$  that is still a prefix in  $L(G)$ . The token following the longest correct prefix of  $d$  is called the error location:

$$d = a_1 a_2 \dots a_j a_k \dots a_m, a_i \in T, 1 \leq i \leq m$$

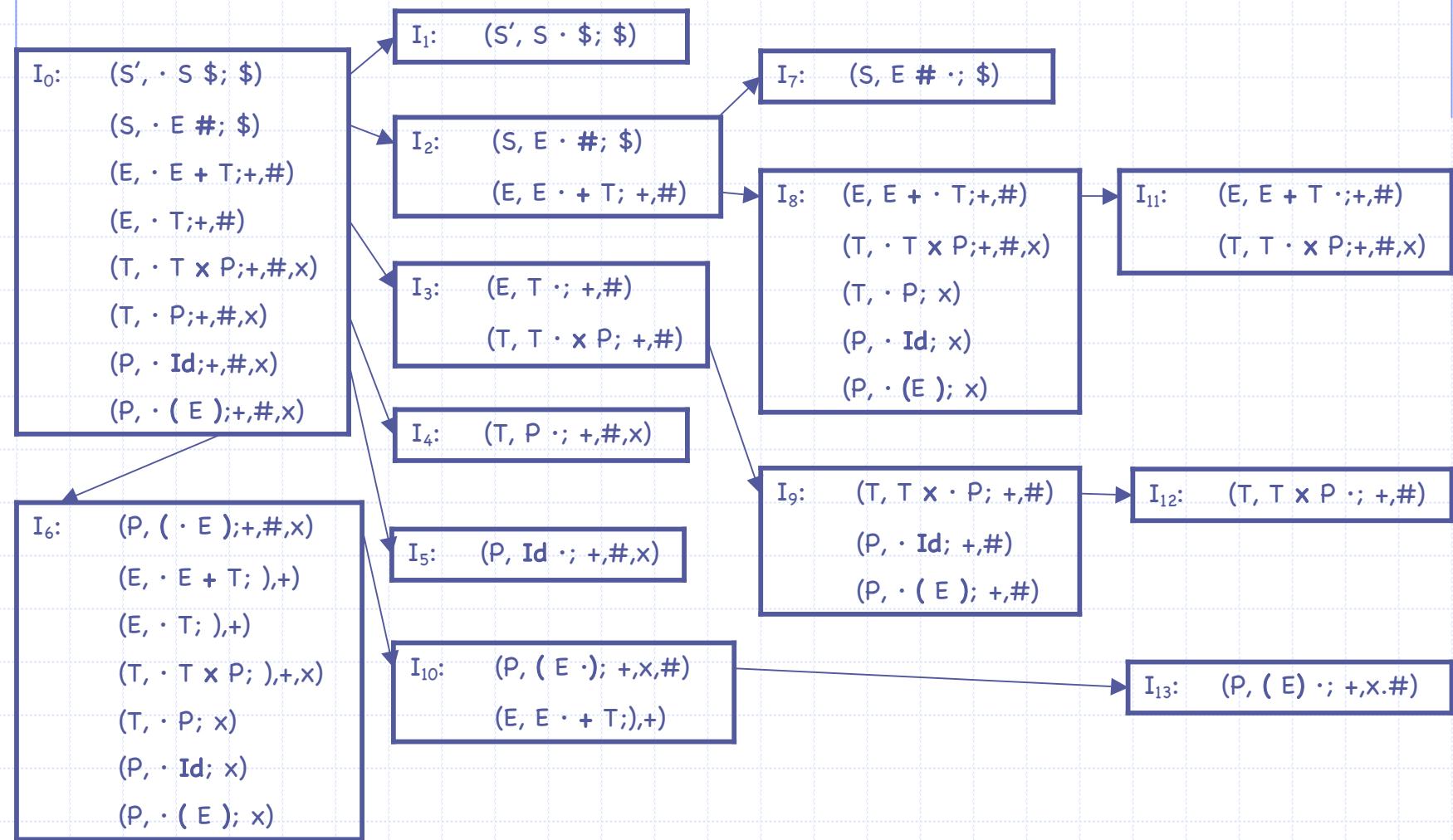
$a_1 a_2 \dots a_j$  - longest correct prefix  
 $a_k$  - error location

# IEDP

- A parser is said to have the “immediate error detection property”, if the parser stops immediately on encountering a prefix  $d \notin L(G)$ .
- Every recursive-descent parser satisfies this property.
- The SLR-, LR(1)-, and LALR(1)-parser construction algorithm generates a parser with IEDP.
- YACC does not have IEDP.

# Example:

- |                     |                      |
|---------------------|----------------------|
| 1: $(S', S \$; \$)$ | 5: $(T, T \times P)$ |
| 2: $(S, E \#; \$)$  | 6: $(T, P)$          |
| 3: $(E, E + T)$     | 7: $(P, Id)$         |
| 4: $(E, T)$         | 8: $(P, ( E ))$      |



# LR(1) Parser Table

- |                       |                        |
|-----------------------|------------------------|
| 1: ( $S'$ , $S \ $$ ) | 5: ( $T, T \times P$ ) |
| 2: ( $S, E \ #$ )     | 6: ( $T, P$ )          |
| 3: ( $E, E + T$ )     | 7: ( $P, Id$ )         |
| 4: ( $E, T$ )         | 8: ( $P, ( E )$ )      |

	#	Id	(	)	+	x	\$	S	E	T	P
0		s5	s6					g1	g2	g3	g4
1							accept				
2	s7				s8						
3	r4				r4	s9					
4	r6				r6	r6					
5	r7				r7	r7					
6		s5	s6						g10	g3	g4
7							r7				
8		s5	s6							g11	g4
9		s5	s6								g12
10				s13	s8						
11	r3					s9					
12	r5				r5						
13	r8				r8	r8					

# YACC Parser Table

1: ( $S'$ , $S \$$ )	5: ( $T, T \times P$ )
2: ( $S, E \#$ )	6: ( $T, P$ )
3: ( $E, E + T$ )	7: ( $P, Id$ )
4: ( $E, T$ )	8: ( $P, ( E )$ )

	#	Id	(	)	+	x	\$	S	E	T	P
0		s5	s6					g1	g2	g3	g4
1							accept				
2	s7				s8						
3	r4	r4	r4	r4	r4	s9	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6		s5	s6						g10	g3	g4
7	r7	r7	r7	r7	r7	r7	r7				
8		s5	s6							g11	g4
9		s5	s6								g12
10				s13	s8						
11	r3	r3	r3	r3	r3	s9	r3				
12	r5	r5	r5	r5	r5	r5	r5				
13	r8	r8	r8	r8	r8	r8	r8				

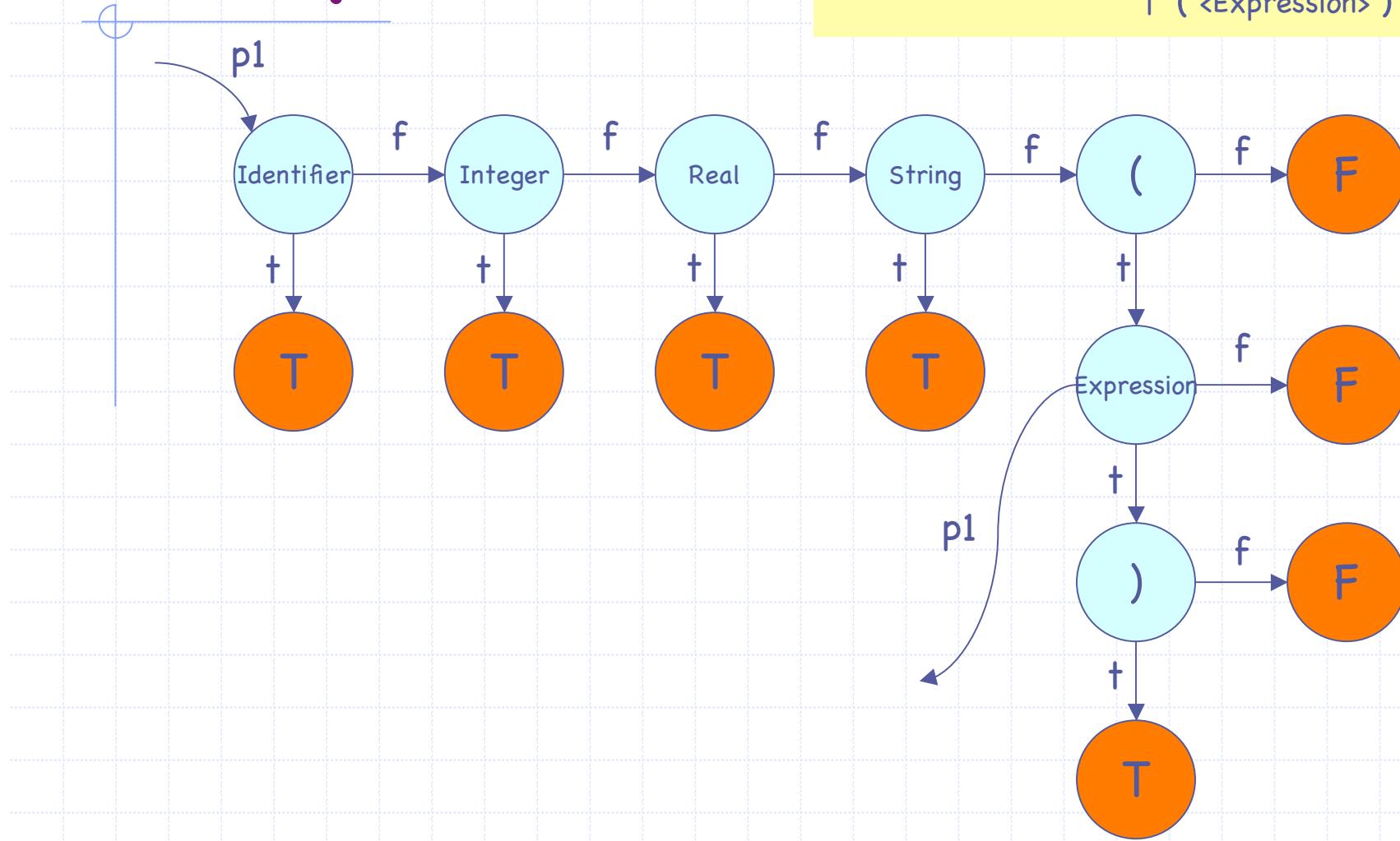
# Annotated Syntax Tree

An annotated directed syntax graph  $\alpha$  for grammar  $G = (V, T, S, P)$  is quadruple  $(N, E, n_0, \gamma)$  with:

- A set  $N = N_T \cup N_P \cup N_F$  with:
  - ◆ The set of terminal vertices  $N_T$ ,
  - ◆ The set of production vertices  $N_P$ , and
  - ◆ The set of final vertices  $N_F = \{\text{True}, \text{False}, \text{Error}\}$ .
- A set  $E \subseteq (N_P \cup N_T) \times N$  of edges,
- A root  $n_0$ ,
- A set  $\gamma \subseteq E \times \{t, f, p\}$ , called FOLLOW-transitions, where  $t$  is the TRUE-successor,  $f$  is the FALSE-successor, and  $p$  is a production definition.

# Example

```
<PrimaryExpression> ::= <Identifier>
| <Integer>
| <Real>
| <String>
| ( <Expression> )
```



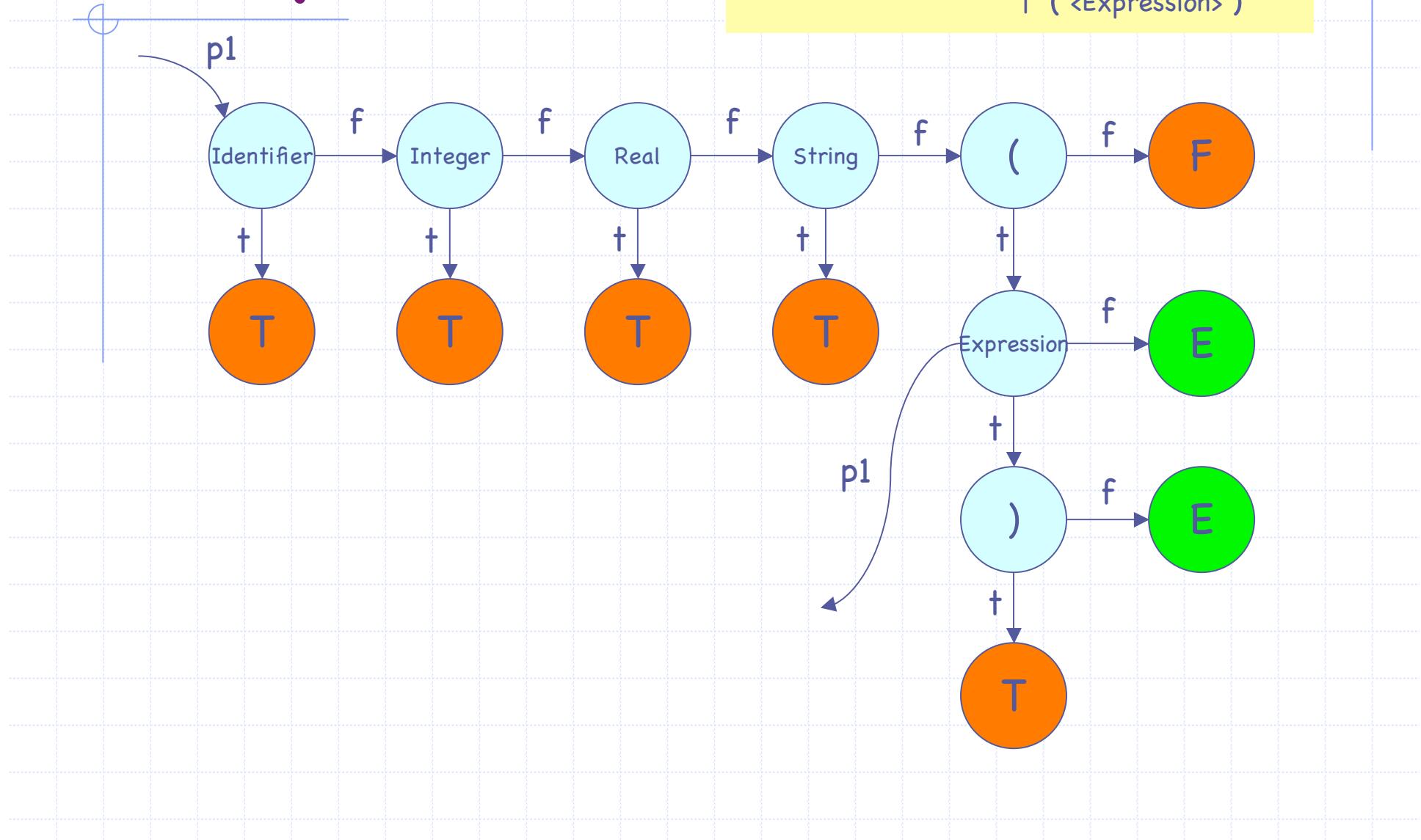
# Constructing Error Vertices

Let  $G = (V, T, S, P)$  be a grammar.

- Construct  $\text{AST} = (N, E, n_0, \gamma)$  a directed annotated syntax graph for  $G$ .
- Let  $F$  is a FALSE-successor of  $u$  and  $S \Rightarrow^+ w_1uw_2$  with  $w_1, w_2 \in V^*$  and  $u \in V$ . If there exists no alternative derivation sequence  $S \Rightarrow^+ w_1vw_2$  with  $w_1, w_2 \in V^*$  and  $v \in V$ , then replace  $F$  with  $E$ . In other words, replace final vertices  $F$  with  $E$ , if no alternative production applications are allowed.
- Do not change  $F$  into  $E$  if an alternative production application must be reachable.

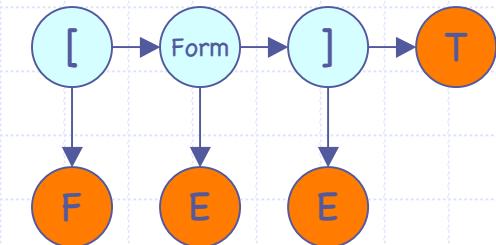
# Example

```
<PrimaryExpression> ::= <Identifier>
| <Integer>
| <Real>
| <String>
| ( <Expression> )
```



# Recursive-Descent Parser

```
public boolean Term2() throws ParseException {  
    if ( consume( LB ) ) {  
        if ( Form() ) {  
            if ( consume( RB ) ) {  
                return true; }  
            else {  
                throw SyntaxError( "]" missing! ); } } }  
        else {  
            throw SyntaxError( "form expected!" ); } } }  
    return false;  
}
```



Term2 ::= <LB> Form <RB>

# Error Recovery Experiment

```
public void Declarations() throws ParseException
{
    boolean lContinue = true;

    while ( lContinue ) {
        try { lContinue = Declaration(); }
        catch (ParseException e)
        {
            System.out.println( "Syntax error at " + e.getMessage() );
        }
        // error recovery
        skipUntil( new int[] { BINDER, IN, EOF } );
    }
}
```

FOLLOW(Declaration) = {<BINDER>, <IN>}

# Error Recovery in YACC

- YACC uses a predefined symbol “error” to mark error recovery points in the grammar. That is, if an error occurs while parsing a program, then YACC searches the parser stack for non-terminals defining an error rule. If no such non-terminal can be found, YACC terminates.
- YACC’s error recovery mechanism will discard all symbols on the stack until the recovery point is found. Furthermore, after a syntax error YACC will skip all input tokens until a sequence of three tokens has successfully been accepted.

# Example

```
%token ID  
%%  
S : E '#' | error  
  
E : E '+' T | T | error  
  
T : T 'x' P | P | error  
  
P : ID | '(' E ')' | error
```

→ 14 reduce/reduce conflicts

12: reduce/reduce conflict (red'ns 5 and 8 ) on PLUS  
12: reduce/reduce conflict (red'ns 5 and 8 ) on RP  
12: reduce/reduce conflict (red'ns 5 and 11 ) on PLUS  
12: reduce/reduce conflict (red'ns 8 and 11 ) on TIMES  
12: reduce/reduce conflict (red'ns 5 and 11 ) on RP  
state 12

E : error\_ (5)  
T : error\_ (8)  
P : error\_ (11)

TIMES reduce 8  
. reduce 5

# Valid Error Annotation

%token ID

%%

S : E '#' | error

E : E '+' T | T

T : T 'x' P | P

P : ID | '(' E ')' | error

Symbols '+' or '#' expected!

- $a \times ( b + c ) \wedge_{\text{Error}}$

- $a \times ( b + c ) \#$

- $a \times ( b + \wedge_{\text{Error}} )$

Primary expression expected!

# Semantic Analysis

The semantic analysis phase

- Connects variable definitions to their uses,
- Checks that all expressions are well-typed, and
- Translates the abstract syntax into a simpler intermediate representation more suitable for code generation.

# Abstract Syntax

- The concrete syntax describes the external representation of statements, expressions, and values.
- The abstract syntax describes the internal representation of statements, expressions, and values.
- In the abstract syntax terminals disappear entirely.
- The building blocks of abstract syntax are tokens rather than terminals.
- The Abstract syntax generates ambiguous syntax trees, but it is not meant for parsing.

# Concrete Syntax

```
<Lines> ::= ( <Expression> \n )*
```

```
<Expression> ::= <Term> ( (+| -) <Term>)*
```

```
<Term> ::= <Primary> ( (x | /) <Primary> )*
```

```
<Primary> ::= ( <Expression> )
              | - <Primary>
              | <Number>
```

# Abstract Syntax

```
<Lines> ::= | <Expression> <Lines>
```

```
<Expression> ::= <Expression> + <Expression>
              | <Expression> - <Expression>
              | <Expression> × <Expression>
              | <Expression> / <Expression>
              | - <Expression>
              | <Number>
```

Note, there is no representation for nested expressions.

# Simple Calculator

```
int Expression() :  
{ int value; int right; }  
{ value=Term()  
  ( "+" right=Term() { value = value + right; } | "-" right=Term() { value = value - right; } )*  
  { return value; }  
}  
  
int Term() :  
{ int value; int right; }  
{ value=Primary()  
  ( "x" right=Primary() { value = value * right; } | "/" right=Primary() { value = value /  
    right; } )*  
  { return value; }  
}  
  
int Primary() :  
{ int value; Token n; }  
{ ( "(" value=Expression() ")" ) { return value; }  
  | n=<NUMBER> { return (new Integer( n.image )).intValue(); }  
  | "-" value=Primary() { return - value; }  
}
```

# Abstract Syntax Classes

```
public abstract class Expression  
{  
    public abstract int eval();  
}
```

ALGEBRAIC DATA TYPE

```
public class PlusExpression extends Expression { ... }  
public class MinusExpression extends Expression { ... }  
public class TimesExpression extends Expression { ... }  
public class DivideExpression extends Expression { ... }  
public class UnaryMinusExpression extends Expression { ... }  
public class IntegerLiteral extends Expression { ... }
```

# PlusExpression

```
public class PlusExpression extends Expression
{
    private Expression fLeft;
    private Expression fRight;

    public PlusExpression( Expression aLeft, Expression aRight )
    {
        fLeft = aLeft;
        fRight = aRight;
    }

    public int eval()
    {
        return fLeft.eval() + fRight.eval();
    }
}
```

# Simple Calculator

```
void Lines() :  
{ Expression value; }  
{ ( value=Expression() "\n" { System.out.println( "-> " + value.eval() ); } )* }
```

```
Expression Expression() :  
{ Expression value; Expression right; }  
{ value=Term() ( "+" right=Term() { value = new PlusExpression( value, right ); }  
| "-" right=Term() { value = new MinusExpression( value, right ); } )* }  
{ return value; } }
```

```
Expression Term() :  
{ Expression value; Expression right; }  
{ value=Primary() ( "x" right=Primary() { value = new TimesExpression( value, right ); }  
| "/" right=Primary() { value = new DivideExpression( value, right ); } )* }  
{ return value; } }
```

```
Expression Primary() :  
{ Expression value; Token n; }  
{ ( "(" value=Expression() ")" { return value; }  
| n=<NUMBER> { return new IntegerLiteral( n.image ); }  
| "-" value=Primary() { return new UnaryMinusExpression( value ); } ) }
```

# Position

- If there is an error that must be reported to the user, the *current* position of the lexical analyzer is a reasonable approximation of the source position of the error.
- A compiler that uses abstract syntax tree data structures can perform the required analysis phases in multiple passes. This means that the scanner has already reached the end the input before an error may occur. Therefore, the source position of each node of the abstract syntax tree must be remembered.

# Positions in Expression

```
public abstract class  
    Expression  
{  
    public int fBeginLine;  
    public int fBeginColumn;  
    public int fEndLine;  
    public int fEndColumn;  
  
    public abstract int eval();  
}
```

# PlusExpression

```
public class PlusExpression extends Expression
{
    private Expression fLeft;
    private Expression fRight;

    public PlusExpression( Expression aLeft, Expression aRight )
    {
        fBeginLine = aLeft.fBeginLine;
        fBeginColumn = aLeft.fBeginColumn;
        fEndLine = aRight.fEndLine;
        fEndColumn = aRight.fEndColumn;

        fLeft = aLeft;
        fRight = aRight;
    }

    public int eval() { return fLeft.eval() + fRight.eval(); }
}
```

# UnaryMinusExpression

```
public class UnaryMinusExpression extends Expression
{
    private Expression fExp;

    public UnaryMinusExpression( Token aMinus, Expression aExp )
    {
        fBeginLine = aMinus.beginLine;
        fBeginColumn = aMinus.beginColumn;
        fEndLine = aExp.fEndLine;
        fEndColumn = aExp.fEndColumn;

        fExp = aExp;
    }

    public int eval()
    {
        return - fExp.eval();
    }
}
```

# DivideExpression

```
public class DivideExpression extends Expression
{
    private Expression fLeft;
    private Expression fRight;

    public DivideExpression( Expression aLeft, Expression aRight ) { ... }

    public int eval()
    {
        int lLeft = fLeft.eval();
        int lRight = fRight.eval();

        if ( lRight == 0 )
            { System.out.println( "DivisionByZero, expression starting in " + "line " +
                fRight.fBeginLine + " column " + fRight.fBeginColumn +
                " evaluates to zero." );
                System.exit( 1 ); }
        return lLeft / lRight;
    }
}
```

# Simple Calculator

```
void Lines() :  
{ Expression value; }  
{ ( value=Expression() "\n" { System.out.println( "-> " + value.eval() ); } )* }
```

```
Expression Expression() :  
{ Expression value; Expression right; }  
{ value=Term() ( "+" right=Term() { value = new PlusExpression( value, right ); }  
| "-" right=Term() { value = new MinusExpression( value, right ); } )*  
{ return value; } }
```

```
Expression Term() :  
{ Expression value; Expression right; }  
{ value=Primary() ( "x" right=Primary() { value = new TimesExpression( value, right ); }  
| "/" right=Primary() { value = new DivideExpression( value, right ); } )*  
{ return value; } }
```

```
Expression Primary() :  
{ Expression value; Token t; }  
{ ( "(" value=Expression() ")" { return value; }  
| t=<NUMBER> { return new IntegerLiteral( t ); }  
| t="-" value=Primary() { return new UnaryMinusExpression( t, value ); } ) }
```

# Example

```
>java SimpleCalculator  
2 x 3  
-> 6  
- 2  
-> -2  
3 / - (2 - 2)  
DivisionByZero, expression starting in line 3 column 5 evaluates to zero.
```

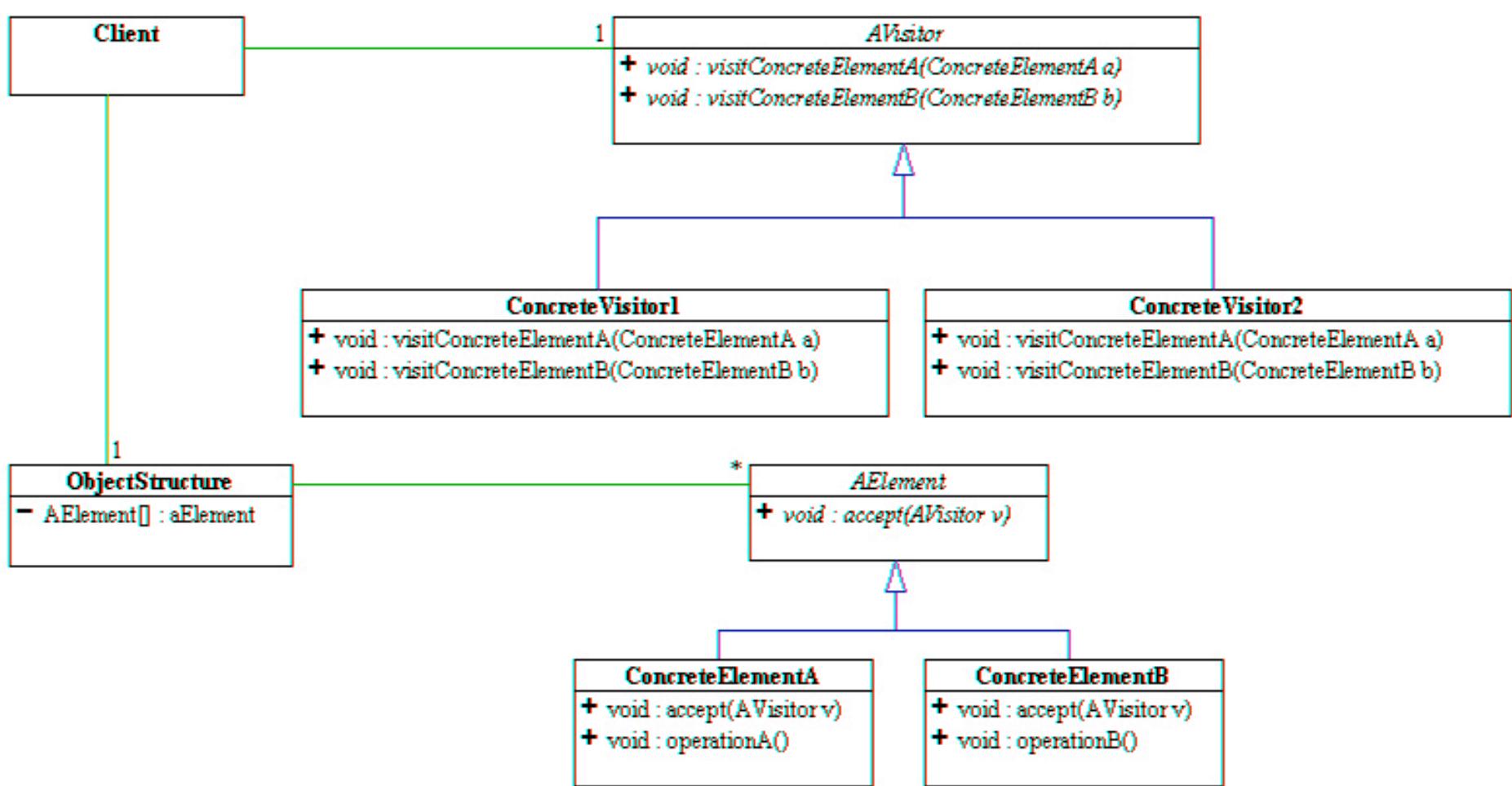
# Orthogonal Directions of Modularity

	Optimize	Eliminate unused variables	...
PlusExpression	•	•	•
MinusExpression	•	•	•
TimesExpression	•	•	•
DivideExpression	•	•	•
UnaryMinusExpression	•	•	•
IntegerLiteral	•	•	•
...	...	...	...

# The Visitor Design Pattern

- The purpose of the Visitor Pattern is to encapsulate an operation that needs to be performed on (all) instances of a class hierarchy.
- Using the Visitor pattern allows for decoupling the classes and the interpretations used upon them.
- The Visitor Pattern allows for modular class extensions.
- Each instance “accepts” a Visitor and sends a message to the Visitor, which includes the instance's class. The visitor will then execute its algorithm for that instance. This process is known as “Double Dispatch.” The instance makes a call to the Visitor, passing itself in, and the Visitor executes its algorithm on the instance. In Double Dispatch, the call made depends upon the type of the Visitor and of the Host (the class of the instance), not just of one component.

# The Visitor Design Pattern



# IVisitor

```
public interface IVisitor
{
    public int visit( PlusExpression e );
    public int visit( MinusExpression e );
    public int visit( TimesExpression e );
    public int visit( DivideExpression e );
    public int visit( UnaryMinusExpression e );
    public int visit( IntegerLiteral e );
}
```

# ExpressionVisitor

```
public class ExpressionVisitor implements IVisitor
{
    public int visit( PlusExpression e ) { return e.fLeft.accept( this ) + e.fRight.accept( this ); }
    public int visit( MinusExpression e ) { return e.fLeft.accept( this ) - e.fRight.accept( this ); }
    public int visit( TimesExpression e ) { return e.fLeft.accept( this ) * e.fRight.accept( this ); }
    public int visit( DivideExpression e )
    {
        int lLeft = e.fLeft.accept( this );
        int lRight = e.fRight.accept( this );

        if ( lRight == 0 )
            { System.out.println( "DivisionByZero, expression starting in line " + e.fRight.fBeginLine +
                " column " + e.fRight.fBeginColumn + " evaluates to zero." );
                System.exit( 1 );
            }
        return lLeft / lRight;
    }
    public int visit( UnaryMinusExpression e ) { return - e.fExp.accept( this ); }
    public int visit( IntegerLiteral e ) { return e.getValue(); }
}
```

# PlusExpression

```
public class PlusExpression extends Expression
{
    public Expression fLeft;
    public Expression fRight;

    public PlusExpression( Expression aLeft, Expression aRight )
    {
        fBeginLine = aLeft.fBeginLine;
        fBeginColumn = aLeft.fBeginColumn;
        fEndLine = aRight.fEndLine;
        fEndColumn = aRight.fEndColumn;

        fLeft = aLeft;
        fRight = aRight;
    }

    public int accept( IVisitor aVisitor ) { return aVisitor.visit( this ); }
}
```

# Symbol Tables

- A symbol table is an environment that maps identifiers to their meanings (i.e., types, values, locations, etc.).
- Each variable in a program has a scope in which it is visible. A symbol table manages this visibility.
- A given identifier can have multiple meanings within a program. Languages that allow for multiple meanings of identifiers assign these identifiers an additional attribute called *namespace*.

# Efficient Implementation

- A program can contains thousands of distinct identifiers. Thus, a symbol table must permit an efficient lookup.
- Identifiers are stored in a hash table that minimizes the search effort in the lookup process.
- In order to maintain a good performance, the underlying hash table should only be filled up to a certain percentage.

# Table of Hash Tables

Level 1:



Level 2:



...

Level N-1:



Most recent level:

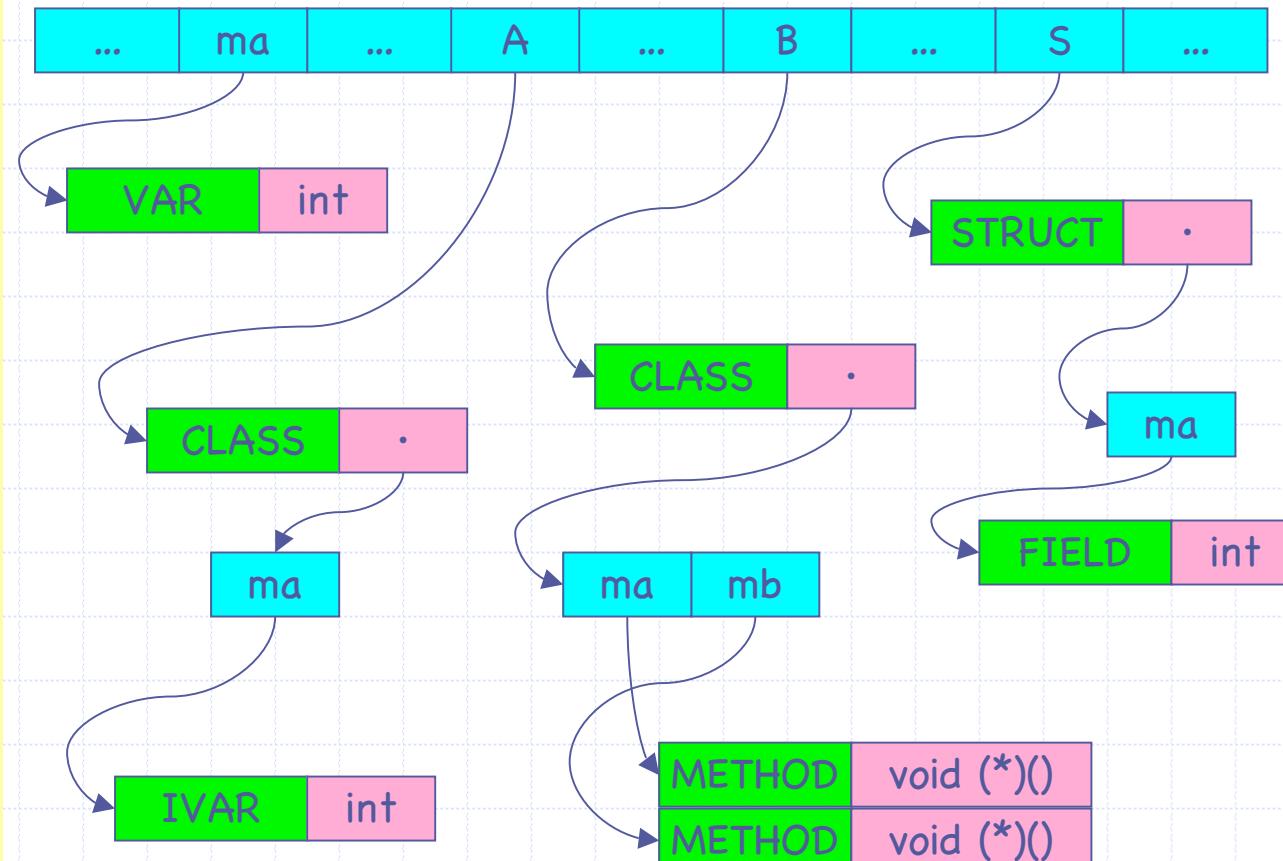


A symbol table can be organized as a table of hash table each not loaded more than 75%. Note, Java's Hashtable does not have this property, as a Java Hashtable object can grow when the maximum load has been reached.

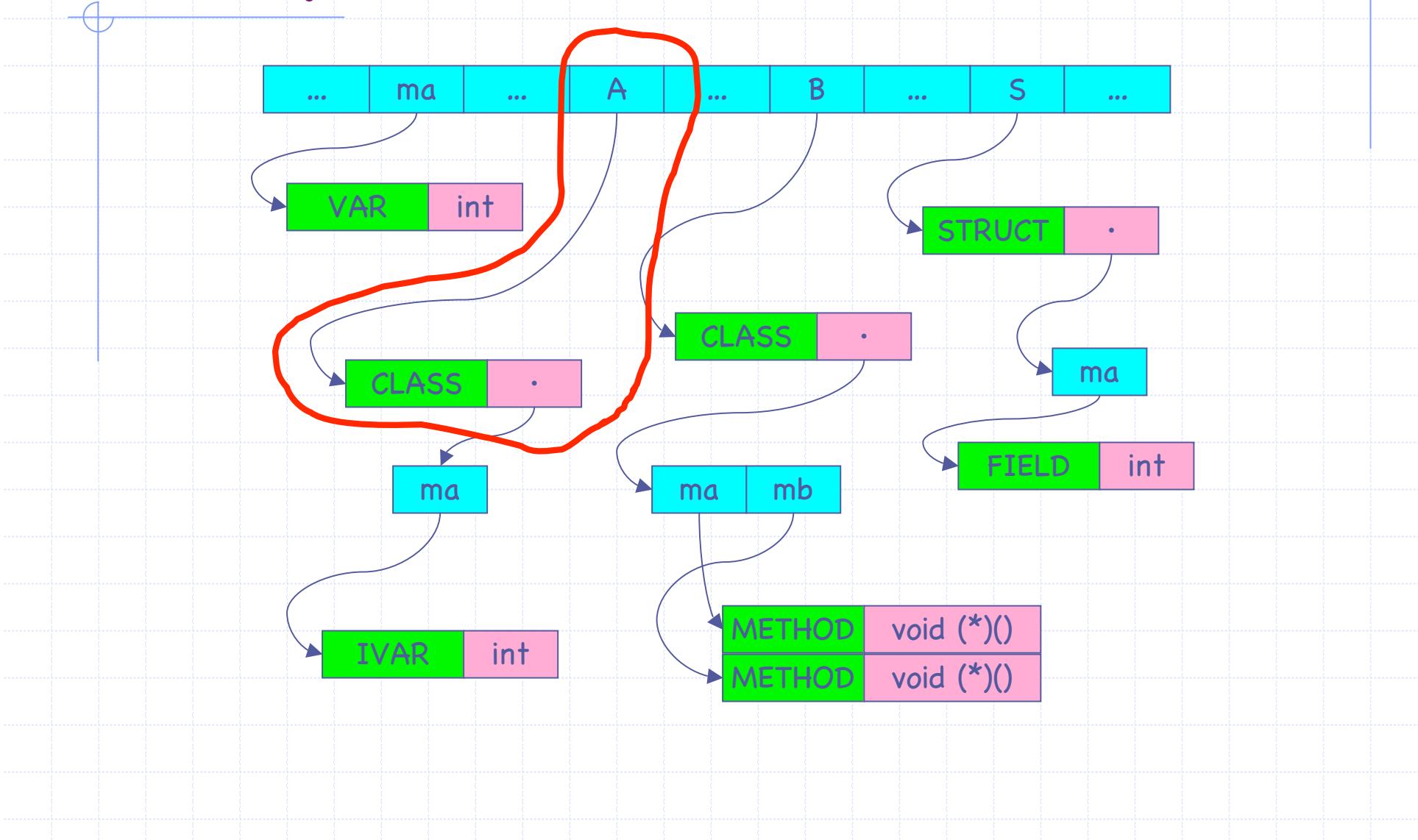
# Namespaces

```
int ma;  
  
class A  
{  
    int ma;  
}  
  
struct  
{  
    int ma;  
} S;  
  
class B  
{  
public:  
    void ma() {}  
    void mb() { ma(); }  
}
```

Namespaces are organized as nested symbol tables.

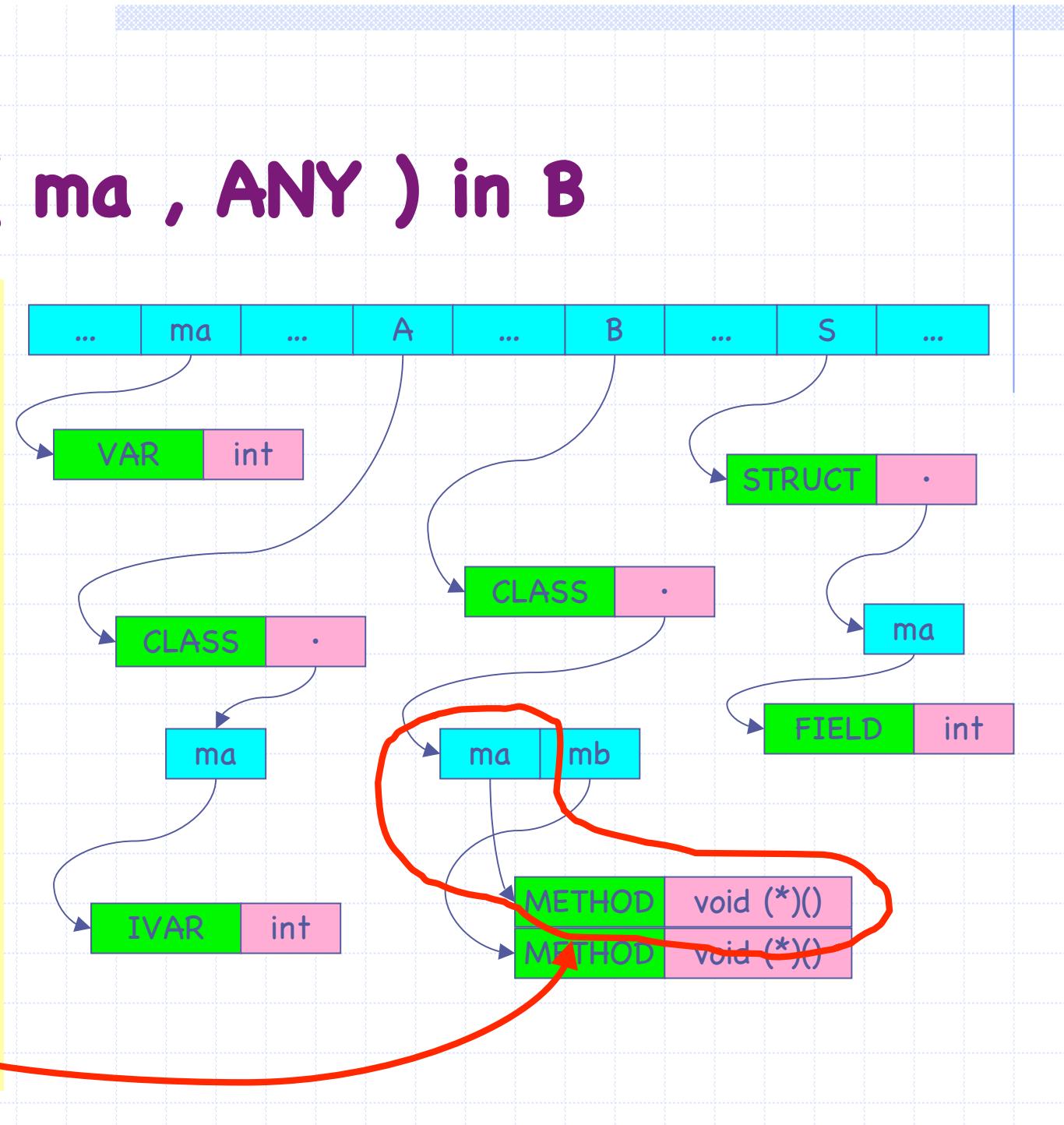


# Lookup( A, CLASS)



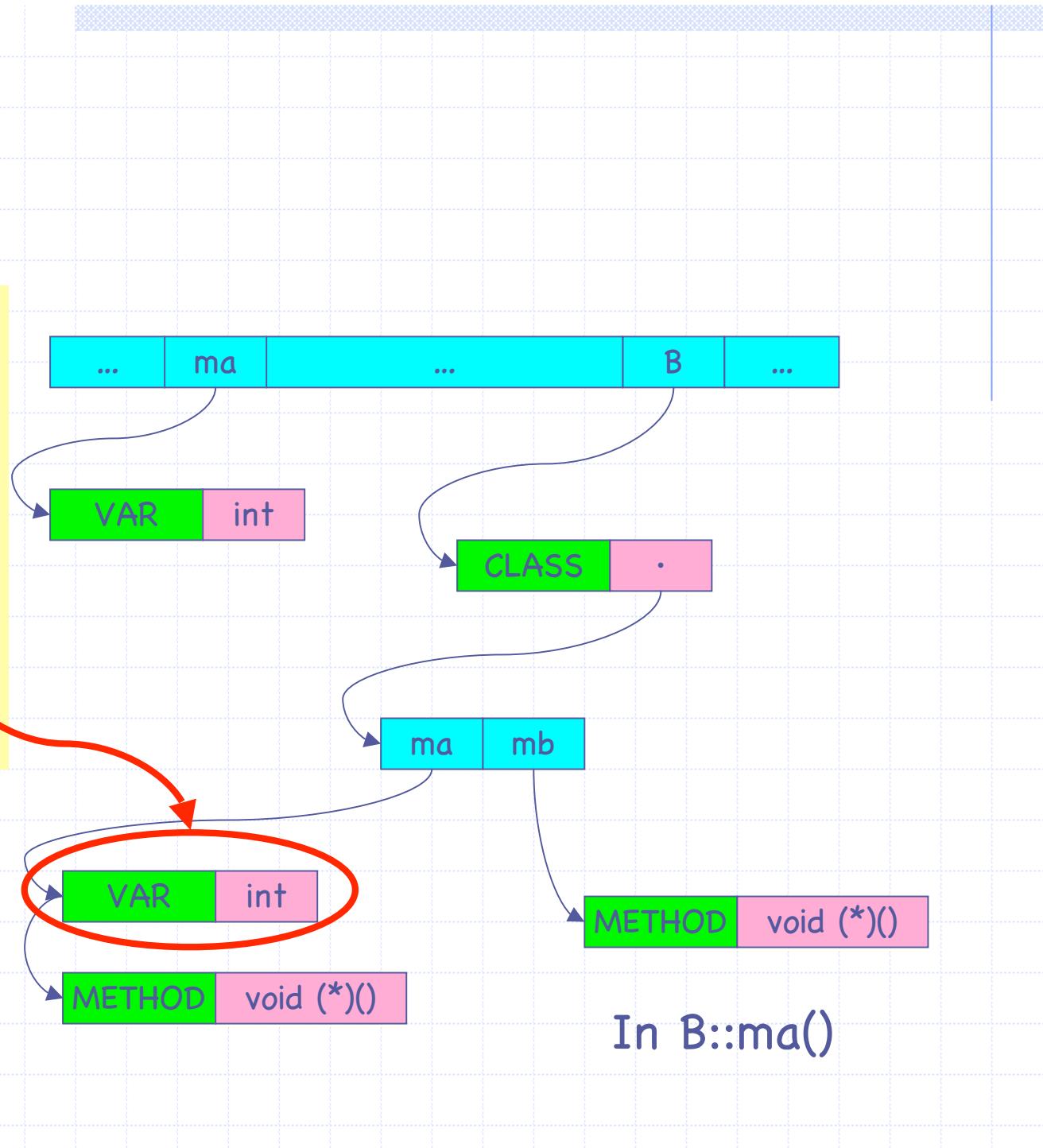
# LOOKUP( ma , ANY ) in B

```
int ma;  
  
class A  
{  
    int ma;  
}  
  
struct  
{  
    int ma;  
} S;  
  
class B  
{  
public:  
    void ma() {}  
    void mb() {ma();}  
}
```



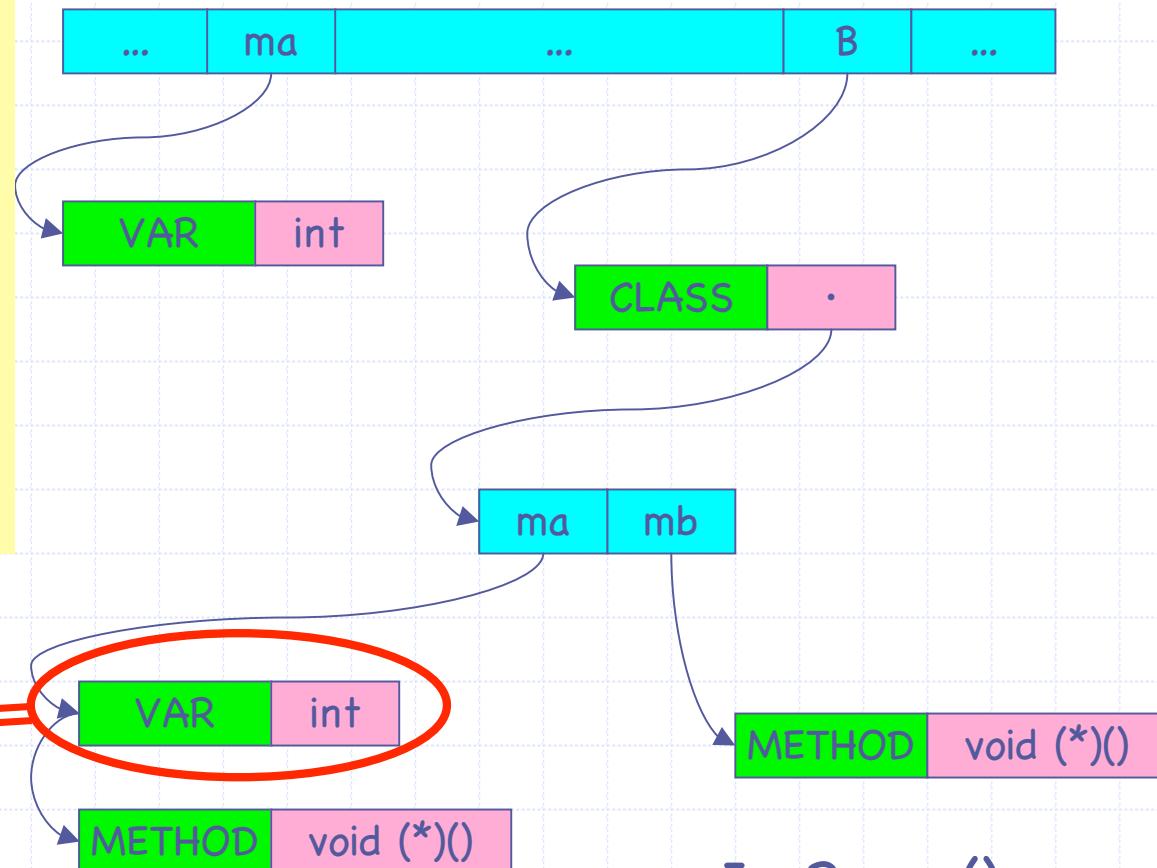
# Hiding

```
int ma;  
...  
  
class B  
{  
public:  
    void ma() { int ma; }  
  
    void mb() { ma(); }  
}
```



# LOOKUP( ma, METHOD ) in B::ma()

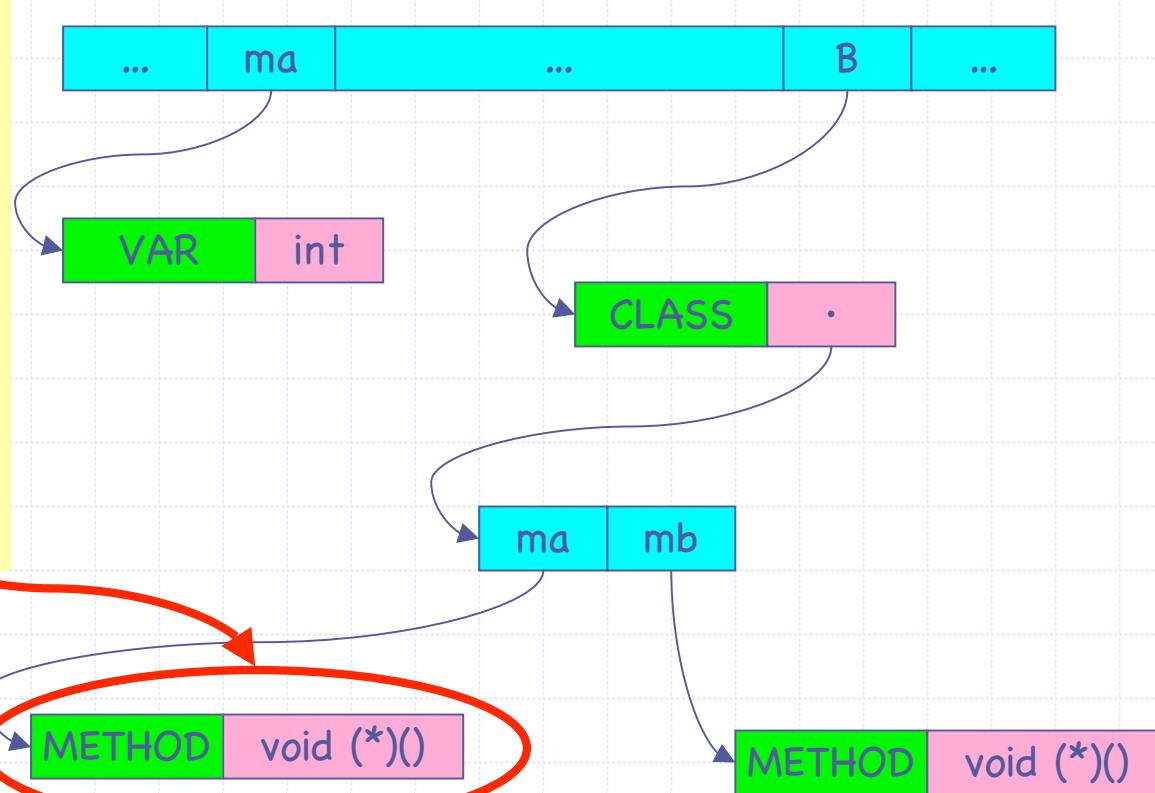
```
int ma;  
...  
  
class B  
{  
public:  
    void ma() { float ma; }  
  
    void mb() { ma(); }  
}
```



In B::ma()

# LOOKUP( ma, METHOD ) in B

```
int ma;  
...  
  
class B  
{  
public:  
    void ma() { float ma; }  
  
    void mb() { ma(); }  
}
```

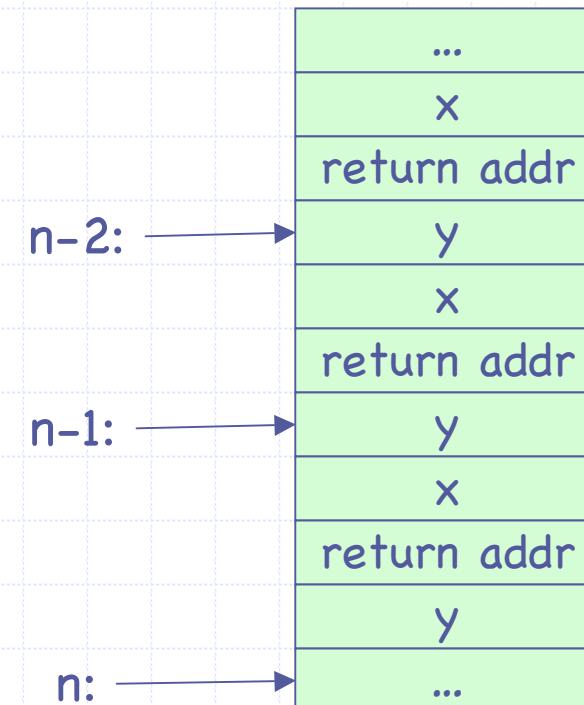


In B::mb()

# Activation Records

```
int f( int x )
{
    int y = x + x;
    if ( y < 10 )
        return f( y );
    else
        return y-1;
}
```

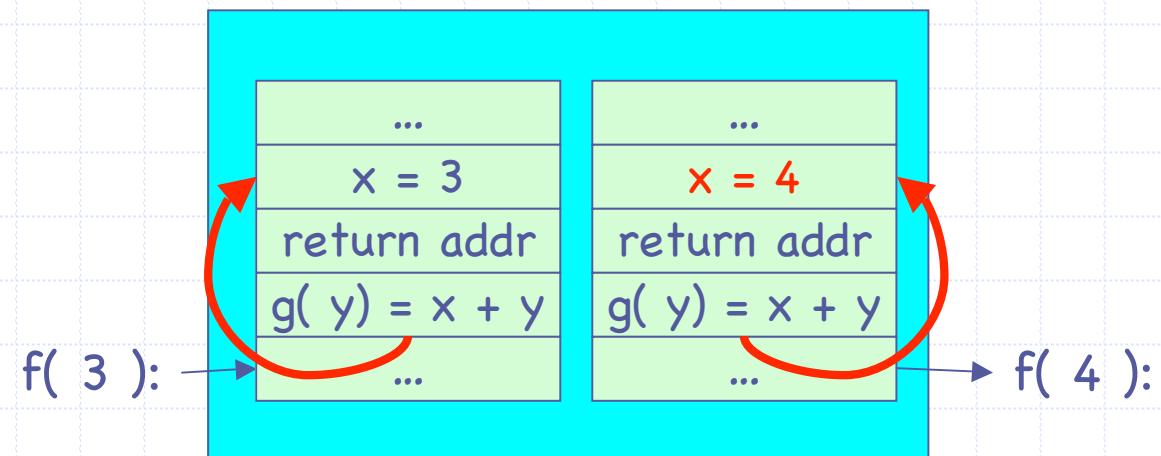
Stack:



# Higher-order Functions

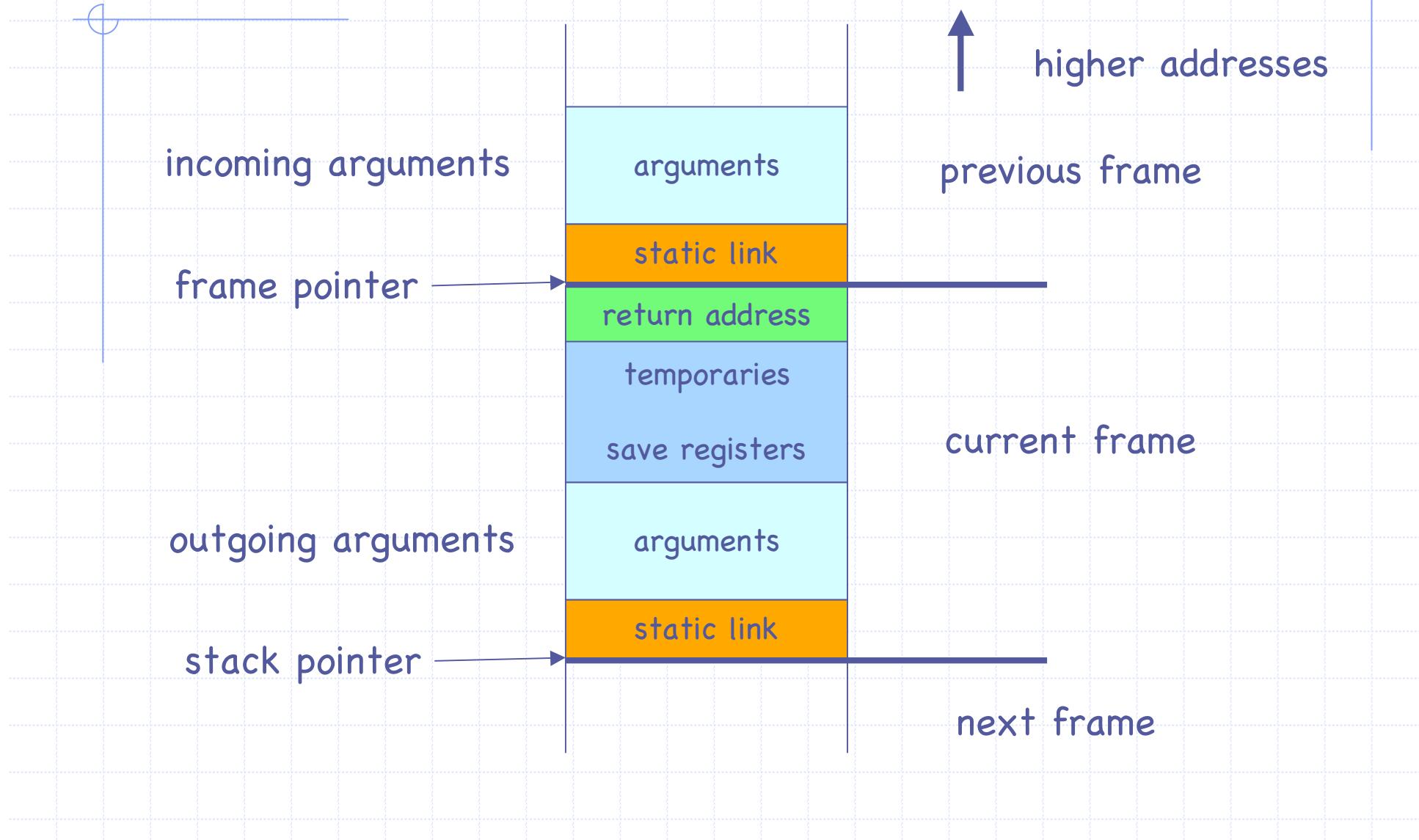
```
let f( x ) =  
    let g( y ) = x + y  
in g  
  
h = f( 3 )  
j = f( 4 )  
  
u = h( 5 )  
v = j( 7 )
```

Stack:

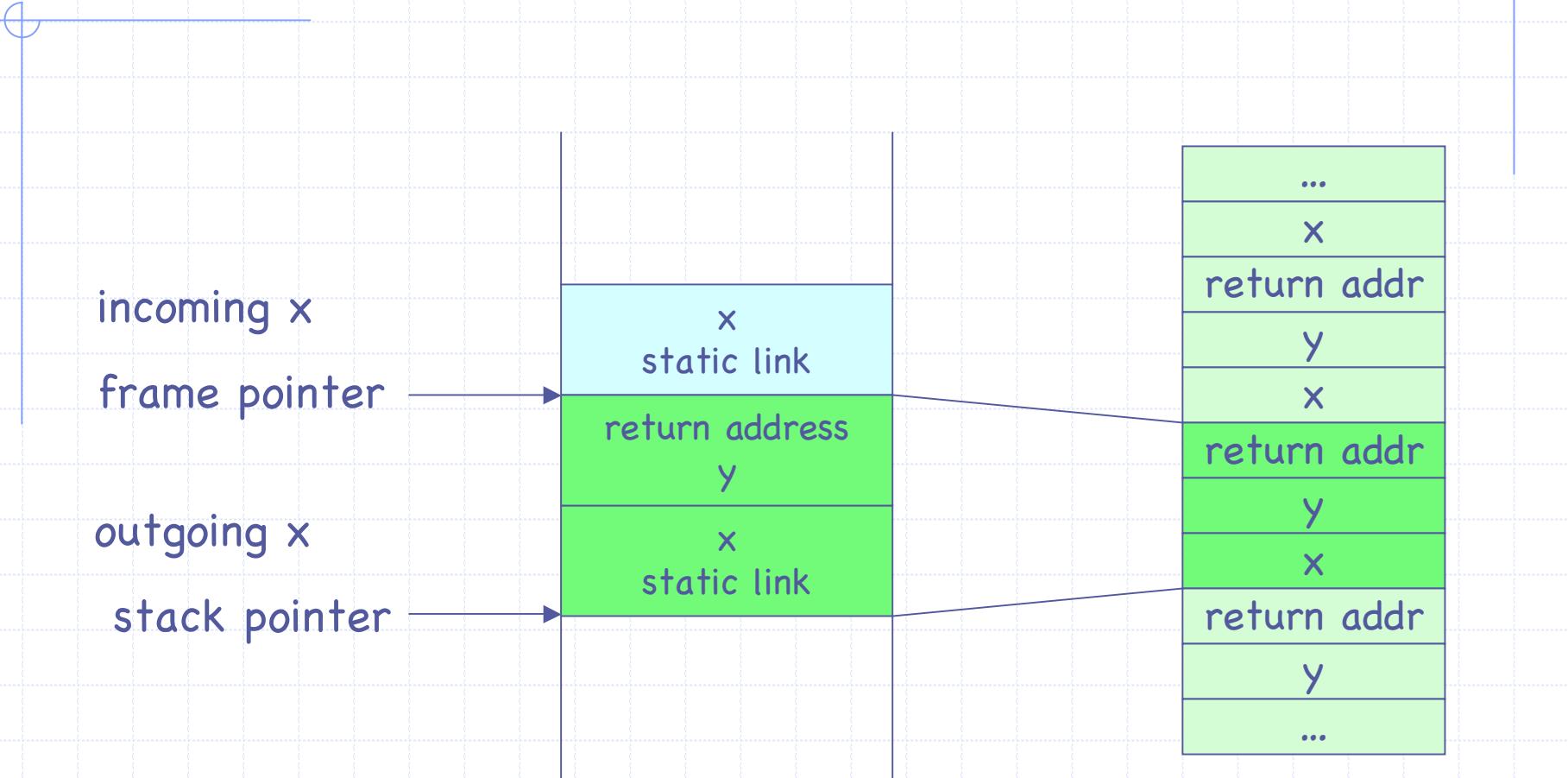


Nested functions may use local variables that have a longer lifetime than their enclosing function invocation.

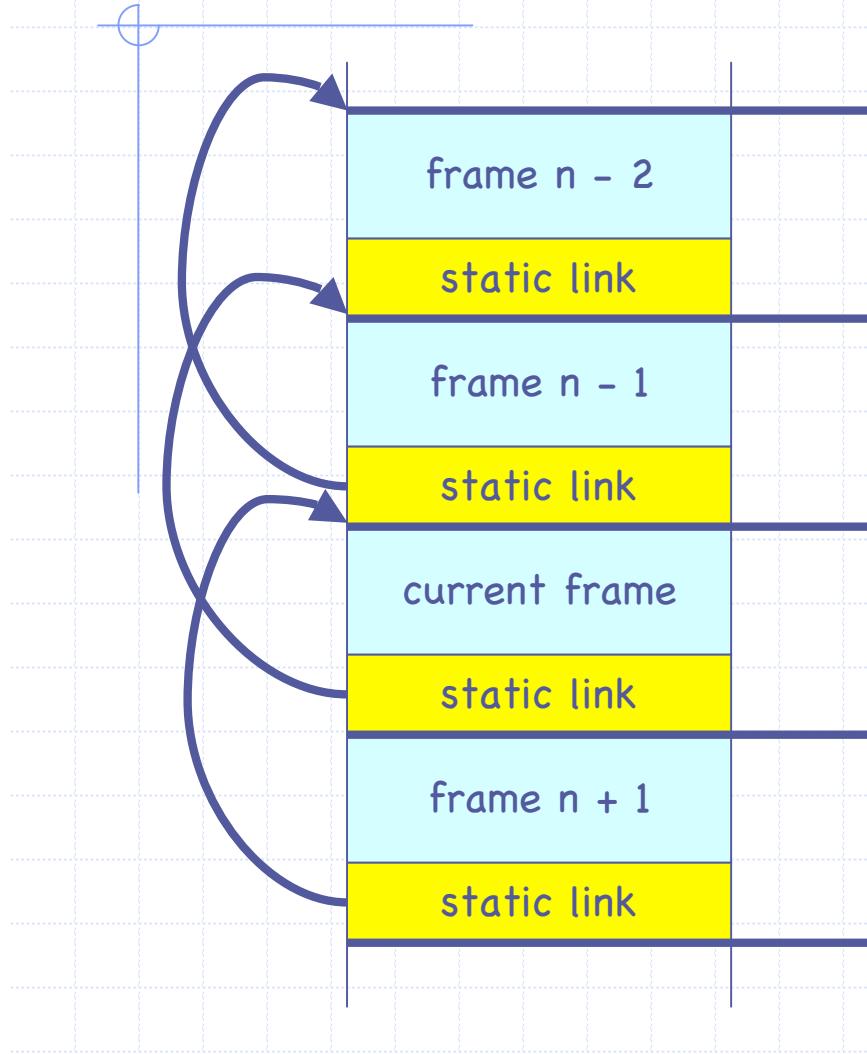
# Stack Frames



# Stack Frames



# Frame Pointer



If a function g calls the function f, then g is the *caller* and f is the *callee*.

On entry, f allocates a frame by subtracting the frame size from the stack pointer.

Old stack pointer becomes new frame pointer.

# Registers

- Registers provide a fast medium to store local variables, arguments, and results of expressions.
- Modern machines often have a large number of registers.
- A register is called caller-save, if the caller must save and restore the register.
- A register is called callee-save, if it is the callee's responsibility to guarantee the register's integrity.

# Parameter Passing

- Parameter passing utilizes the stack. However, some parameters may also be passed in registers to the called function.
- Some languages assume a contiguous memory block in which arguments reside (e.g., C). C also allows one to take the address of a formal argument.
- Arguments can be passed in registers, but one usually must reserve some space to save them if the register hosting them is needed elsewhere.

# Return Addresses

- When a function g calls a function f, then f eventually returns. Therefore, f needs to know where to go back.
- The return address is stored in the current frame.
  - The machine instruction CALL usually pushes the contents of the instruction pointer onto the stack. At this point, the instruction pointer points to the instruction immediately following the CALL.
  - The machine instruction RET usually removes the return address from the stack and loads it into the instruction pointer, which results in the fact that the program returns from the call.

# Static Links

```
function prettyprint( BinTree ) =  
let  
    output = ""
```

```
        function write( S ) = output = concat( output, s )
```

```
function show( N, T ) =  
let  
    function indent( S ) = for i = 1 to N do write( " " );  
                      output = concat( output, S );  
                      write( "\n" );  
in if T==nil then indent( "." )  
    else indent( T.key ); show( N+1, T.left ); show( N+1, T.right )  
in show( 0, BinTree ); output
```

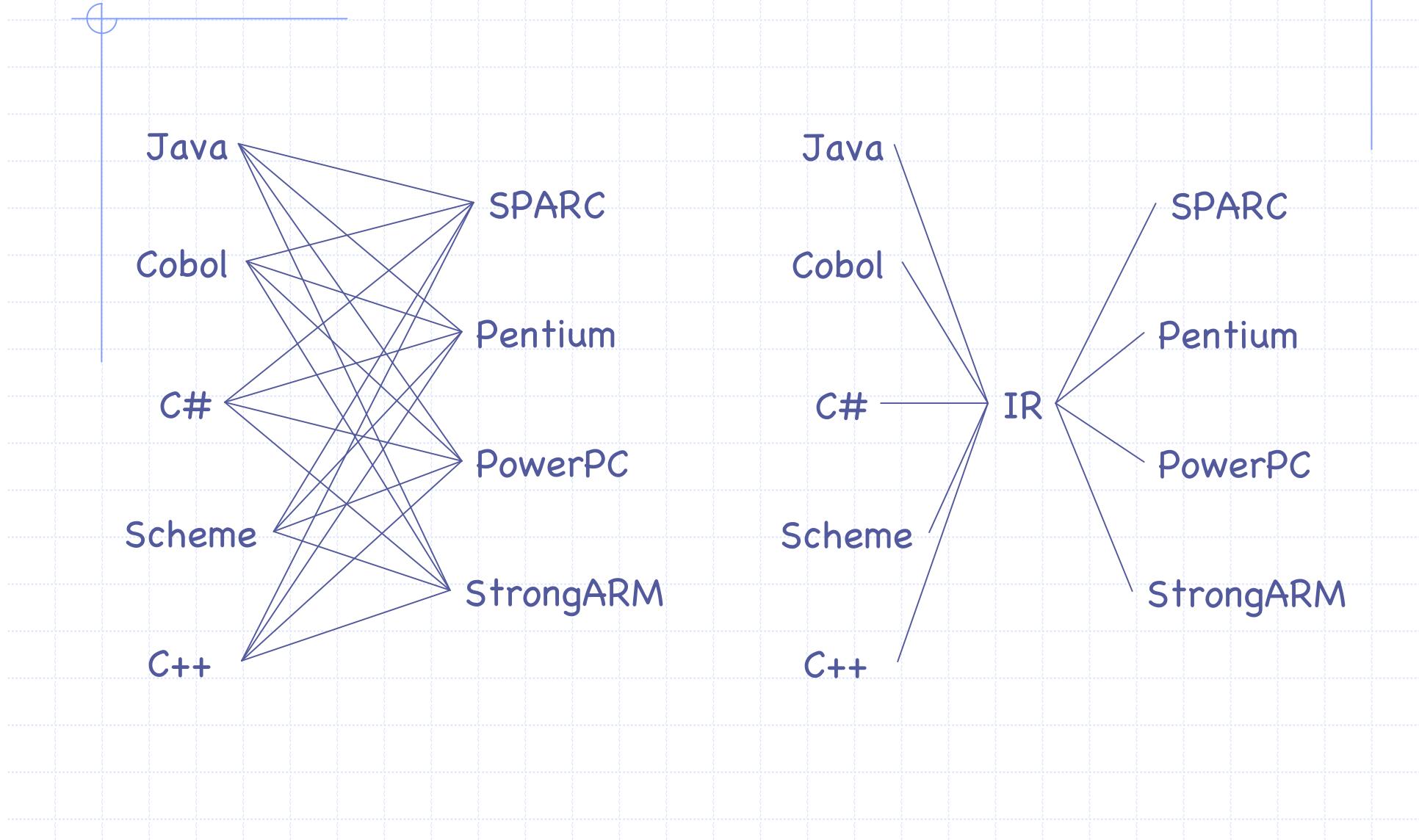
To access output in indent we must load ident's static link, which gives us access to show's static link, which in turn gives us access to the frame of prettyprint in which output is located.

# Intermediate Code Generation

Although a source program can be translated directly into the target language, it is often better to use a machine-independent intermediate form:

- Retargeting is facilitated. A compiler for a different machine can be created by attaching a new back-end for the new machine to an existing front-end.
- A machine-independent code optimizer can be applied to the intermediate representation, before the concrete machine code is generated.

# Front-end vs. Back-End

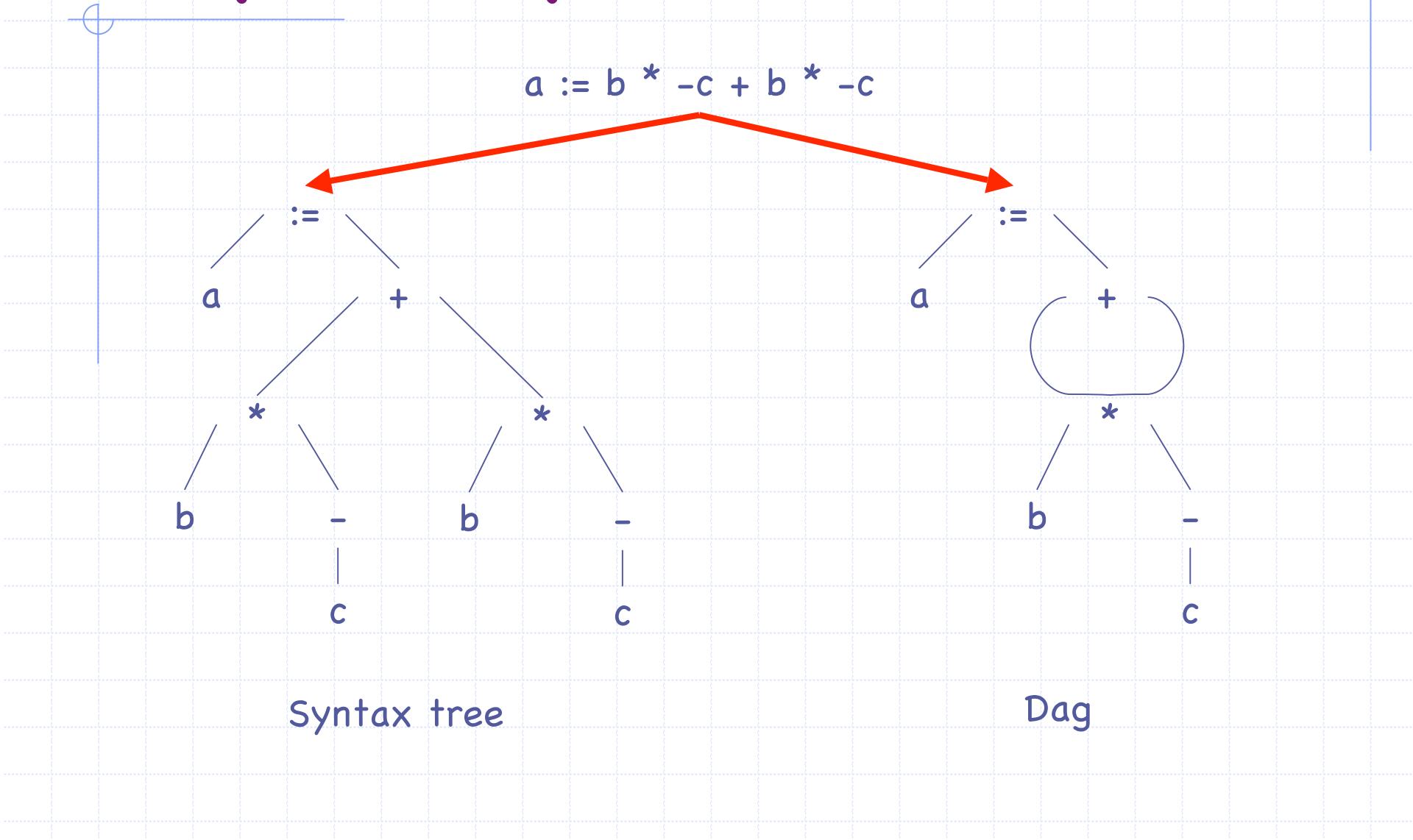


# Intermediate Representation

A good intermediate representation:

- Should be designed in a way that the semantic analysis phase can easily produce it,
- Should be designed in a way that is easy to translate it into a real machine language for all desired target machines,
- Should provide a clear and simple instruction set, so that intermediate code optimization becomes feasible.

# Graphical Representation



# Three-Address Code

- Three-address code is a sequence of statements of the general form

$$x := y \text{ op } z$$

where  $x$ ,  $y$ , and  $z$  are names, constants, or compiler-generated temporaries;  $\text{op}$  stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data.

# Three-Address Example

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

$t_1 = -c$   
 $t_2 = b * t_1$   
 $t_3 = -c$   
 $t_4 = b * t_3$   
 $t_5 = t_2 + t_4$   
 $a = t_5$

Syntax tree

$t_1 = -c$   
 $t_2 = b * t_1$   
 $t_3 = t_2 + t_2$   
 $a = t_3$

Dag

# RPN

- In the 1920's, Jan Lukasiewicz developed a formal logic system which allowed mathematical expressions to be specified without parentheses by placing the operators before (prefix notation) or after (postfix notation) the operands. For example the (infix notation) expression

$$(4 + 5) \times 6$$

- prefix notation:  $\times + 4 5 6$
- postfix notation:  $4 5 + 6 \times$
- Prefix notation is also known as Polish Notation. Hewlett-Packard adjusted the postfix notation for a calculator keyboard, added a stack to hold the operands and functions to reorder the stack. HP called the result Reverse Polish Notation (RPN).

# RPN Example

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

b

c  
b

-c  
b

b \* -c

b  
b \* -c

c  
b  
b \* -c

-c  
b  
b \* -c

b \* -c  
b \* -c

b \* -c + b \* -c

$\emptyset, \text{MEM}(a) = b * -c + b * -c$

# Three-Address Statements

- Three-address statements can have symbolic labels. A symbolic label represents the index of a three-address statement in the array holding the intermediate code.

L1:  $x := y \ op \ z$

- There are control flow three-address statements like unconditional jumps and conditional jumps.

# Assignment Statement

- An assignment statement is a three-address statement of the form

$$[ l: ] x := y \text{ op } z$$

where  $\text{op}$  is a binary operator supported by the abstract machine model underlying the intermediate code.

# Assignment Instruction

- An assignment instruction is a three-address statement of the form

$$[ l: ] x := op y$$

where  $op$  is a unary operator supported by the abstract machine model underlying the intermediate code.

# Copy Statement

- An copy statement is a three-address statement of the form

[ l: ] x := y

where x is called L-value and y is called R-value.

An L-value has to be an address that denotes the location in which the R-value is going to be stored.

# Jump Statements

- The unconditional jump

[ l: ] goto m

is a three-address statement in which label m denotes the next statement to be executed.

- Conditional jumps

[ l: ] if x relop y goto m

apply a relational operator (e.g., <, ==, >=, etc.) to x and y, and execute the statement labeled with m next if x relop y evaluates to TRUE. Otherwise, the statement following the conditional statement is executed next.

# Procedure Statements

- To represent procedures, one usually needs three different statements:
  - **param x :**  
Stores a parameter x into its corresponding frame-specific place. In a stack-based system, param x means push x onto the stack.
  - **call p:**  
Calls the procedure p.
  - **return y:**  
Return y as result of a procedure. The value y is optional.

# Address Statements

- Address statements are for indirect address modes in which a value denotes the address of another value.
  - $x := \&y$   
The value of  $x$  is set to be the location of  $y$ .
  - $x := *y$   
The value of  $x$  is set to value of the location denoted by  $y$ .  
This statement is often called load-indirect.
  - $*x := y$   
The value in the location denoted by  $x$  is set to  $y$ . This statement is often called store-indirect.

# Example

```
if (0 < x)
{
    fact = 1;
    do
    {
        fact = fact *
x;
        x = x - 1;
    } while (x > 0);
}
```

```
L2: if 0 >= x goto L1
fact := 1
t1 := fact * x
fact := t1
t2 := x - 1
x := t2
if x <= 0 goto L1
goto L2
```

L2:

L1:



# Array Access

```
int[] a = new int[5];
```

...

```
for ( int i = 0; i < 5; i++)
    a[i] = i;
```

L2:

L1:

{ create array a}

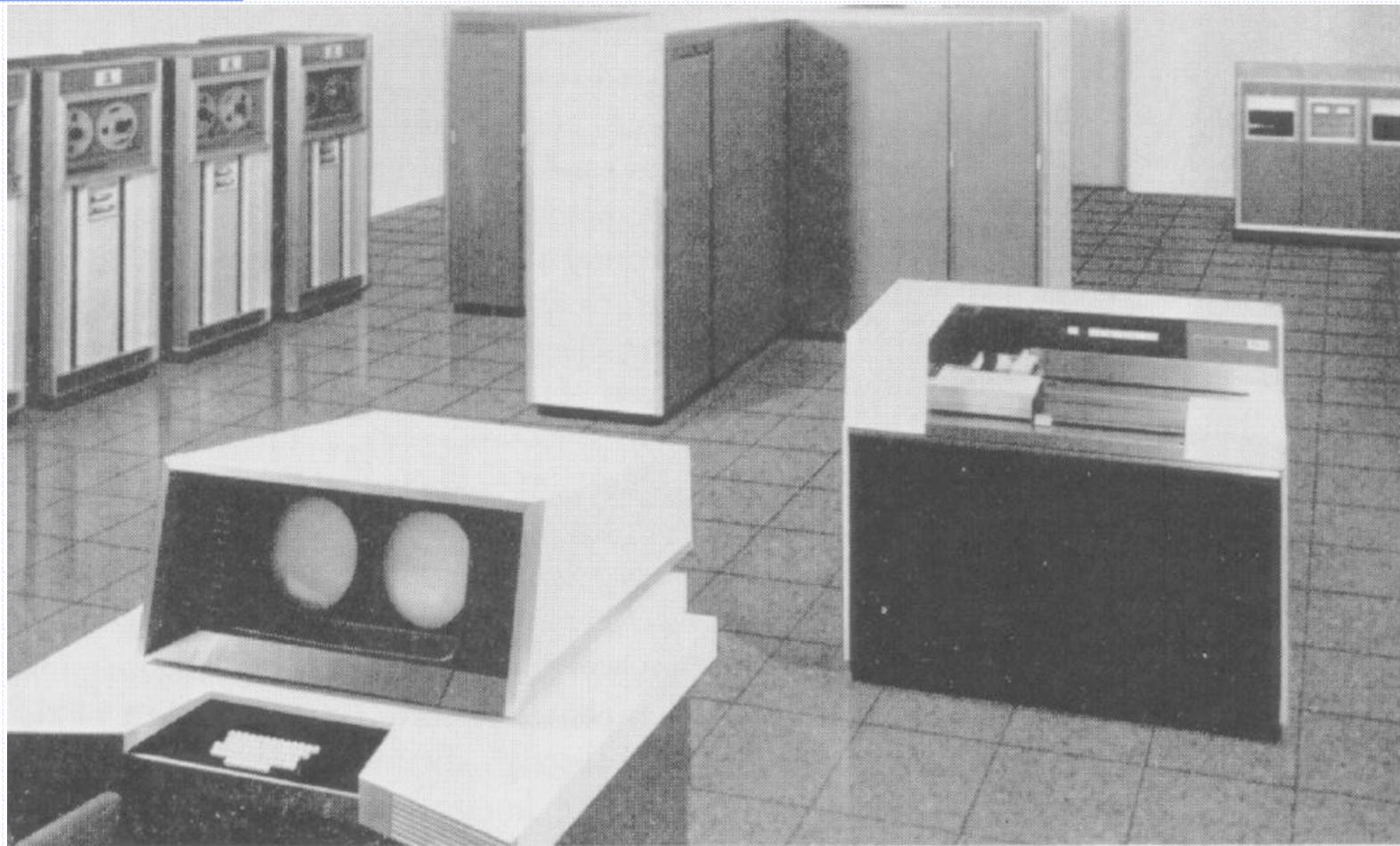
...

```
i := 0
if i >= 5 goto L1
t1 := &a
t2 := t1 + i
*t2 := i
t3 := i + 1
i := t3
goto L2
```

# P-Code

- P-code is the intermediate language produced by a number of Pascal compilers in the 1970s and early 1980s.
- P-code is the assembly language for a hypothetical stack machine.
- The first Pascal compiler was designed in Zurich for the CDC 6000 computer family, and it became operational in 1970.

# CDC 6000 Family



Source: <http://www.moorecad.com/standardpascal/cdc6400.html>

# Call-by-Name vs. Call-by-Value

Niklaus Wirth:

Europeans tend to pronounce his name properly, as *Nih-klaus Virt*, while Americans usually mangle it into something like *Nickles Worth*. That has led to the programmer joke saying Europeans call him by name while Americans call him by value.

# P-code Example

x := y + 1

```
lda x ; load address of x  
lod y ; load value of y  
ldc 1 ; load constant 1  
adi ; add  
sto ; store top to address  
; below top and pop both
```

# Simple Expression Trees

## Expressions:

- CONST( int value )
- NAME( Label label )
- TEMP( Temp temp )
- BINOP( int binop, Exp left, Exp right )
- MEM( Exp exp )
- CALL( Exp fun, ExpList args )
- ESEQ( Stm stm, Exp exp )

## Statements:

- MOVE( Exp src, Exp dest )
- EXP( Exp exp )
- JUMP( Exp exp, LabelList labels )
- CJUMP( int relop, Exp left, Exp right, Label true, Label false )
- SEQ( Stm left, Stm right )
- LABEL( Label label )

## Auxiliaries:

- ExpList( Exp head, ExpList tail )
- StmList( Stm head, StmList tail )

# Expressions

- **CONST( i )**  
The integer constant i.
- **NAME( n )**  
The symbolic constant n (i.e., a label).
- **TEMP( t )**  
A temporary t. A temporary in an abstract machine is similar to a register in a real machine.
- **BINOP( op, left, right )**  
The application of the binary operator op to the operands left and right. The subexpression left is evaluated before right.
- **MEM( e )**  
The contents of word-size byte of memory starting at address e.
- **CALL( f, args )**  
A procedure call: the application of f to argument list args. Evaluation is from left to right.
- **ESEQ( Stmt stmt, Exp exp )**  
The statement stmt is evaluated for side effects, then e is evaluated for a result.

# Statements

- **MOVE( Temp t, e )**  
Evaluate e and move it into temporary t.
- **MOVE( MEM( e<sub>1</sub> ), e<sub>2</sub> )**  
Evaluate e<sub>1</sub>, which yields address a, then evaluate e<sub>2</sub>, and store the result into word-size bytes of memory starting at a.
- **EXP( e )**  
Evaluate e and discard it.
- **JUMP( e, labels )**  
Transfer control (jump) to address e.
- **CJUMP( relop, left, right, t, f )**  
Evaluate left, right in that order, yielding values a, b. Then compare a,b using relop, and jump to t if the result is true. Otherwise jump to f.
- **SEQ( left, right )**  
The statement left followed by statement right.
- **LABEL( n )**  
Define the constant value of name n to be the current machine code address.

# Translating Conditionals

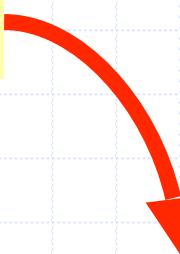
```
if (a < b && c < d)
{ true block }
else
{ else block }
```



```
SEQ( CJUMP( <, a, b, NAME( z ), f ),
      SEQ( LABEL( z ),
            CJUMP( >, c, d, t, f ) ) )
```

# Translating Booleans

flag = (a < b && c < d);



```
MOVE( MEM( NAME( flag ) ),
      ESEQ( SEQ( MOVE( TEMP( r ), CONST( 1 ) ),
                  SEQ( CJUMP( <, a, b, NAME( z ), NAME( f ) ),
                      SEQ( SEQ( LABEL( z ),
                                CJUMP( >, c, d, NAME( t ), NAME( f ) ) ),
                      SEQ( SEQ( LABEL( f ),
                                MOVE( TEMP( r ), CONST( 0 ) ) ),
                                LABEL( t ) ) ) ) ),
      TEMP( r ) ) )
```

# Translating While Loops

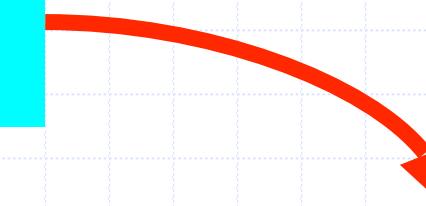
```
while ( condition )  
    body
```

```
test:  
    if not( condition ) goto done  
    body  
    goto test  
done:
```

# Translating For Loops

```
for ( i=lo; i <= hi; i++ )  
    body
```

If limit <> MaxInt



```
i=lo;  
limit=hi;  
while ( i <= limit )  
{  
    body  
    i++;  
}
```

# Translating For Loops MaxInt

```
for ( i=lo; i <= hi; i++ )  
    body
```

If limit == MaxInt



```
i=lo;  
limit=hi;  
while ( i <= limit )  
{  
    body  
    if ( i < limit )  
        i++;  
    else  
        break;  
}
```

# AOSD

- Aspect-oriented software development is a technology for separation of concerns (SOC) in software development.
- The techniques of AOSD make it possible to modularize crosscutting aspects of a system.
- Like objects, aspects may arise at any stage of the software lifecycle, including requirements specification, design, implementation, etc. Common examples of crosscutting aspects are design or architectural constraints, systemic properties or behaviors (e.g., logging and error recovery), and features.

# Case Study: Aspect J#

- Aspect J# is a fictional extension of J#.
- We use AspectJ as our motivating example.
- In order to weave an aspect A into a given class C, we construct a subclass D : C, where all methods with a matching signature (join points) are overridden using the aspect A.
- We construct a tool that implements a particular aspect weaver.

# Fibonacci

```
package Fibonacci;  
javac Fibonacci.java  
  
public class Fibonacci  
{  
    public long fib( long n )  
    {  
        if ( n < 2 )  
            return n;  
        else  
            return fib( n - 1 ) + fib( n - 2 );  
    }  
}
```

# FibonacciTest

```
package FibonacciTest;
```

```
import Fibonacci.*;
```

```
public class FibonacciTest  
{
```

```
    public static void main( String[] args )  
    {
```

```
        System.out.print( "fib( 5 ): " );
```

```
        System.out.println( (new Fibonacci()).fib( 5 ) );
```

```
    }
```

```
}
```

```
javac -classpath ../ FibonacciTest.java
```

```
>java -classpath ../ FibonacciTest.FibonacciTest  
fib( 5 ): 5
```

# Tracing Aspect

```
public aspect Tracing {  
    // Join points at execution of methods.  
    pointcut pc(long n) : execution(long * (long)) && args(n);  
  
    // Trace the beginning of methods.  
    before( long n ) : pc(n) {  
        entryIndent();  
        System.out.println( "calling " + fThisJoinPoint.getName() +  
                            "( " + n + " )" ); }  
  
    // Trace the end of methods.  
    after(long n) returning(long ret) : pc(n) {  
        exitIndent();  
        System.out.println( "" + ret + " == " + fThisJoinPoint.getName() +  
                            "( " + n + " )" ); }  
  
    ...  
}
```

# TracingFibonacci

```
public class TracingFibonacci extends Fibonacci
{
    ...
    // aspect-related code
    // join point
    public long fib( long n )
    {
        try
        {
            fThisJoinPoint = getClass().getMethod( "fib", new Class[] { Long.TYPE } );
            before( n );
            return after( super.fib( n ), n );
        }
        catch ( Exception e ) { System.out.println( e.getMessage() );
                               System.exit( 0 ); }

        return 0;
    }
}
```

javac -classpath ../ TracingFibonacci.cs

# Test

```
import FibonacciTrace.TracingFibonacci;

public class FibonacciTraceTest
{
    public static void main(String[] args)
    {
        System.out.print( "fib( 5 ): " );
        System.out.println( new TracingFibonacci().fib( 5 ) );
    }
}
```

```
javac -classpath ../ TracingFibonacciTest.cs
```

# Output

```
>java -classpath ../ FibonacciTraceTest.FibonacciTraceTest
fib( 5 ):
calling fib( 5 )
calling fib( 4 )
calling fib( 3 )
calling fib( 2 )
calling fib( 1 )
1 == fib( 1 )
calling fib( 0 )

...
calling fib( 1 )
1 == fib( 1 )
2 == fib( 3 )
5 == fib( 5 )
5
```

# Aspect Weaver

- We build an aspect weaver tool, which automatically weaves a predefined aspect into a class.
- The tool iterates over all public members of a given class. If it can find a method with a matching signature, the weaver creates a new overridden method (i.e., incorporate the predefined aspect).
- We will use the packages `java.lang.reflect` and `com.sun.tools`.

# Main

```
public static void main( String[] args )
{
    try
    {
        System.out.print( "Class: " );
        byte[] lBuffer = new byte[1024];
        int lCount = System.in.read( lBuffer );
        String lClassName = new String( lBuffer, 0, lCount - 2 ); // kill \r\n

        Class lClass = Class.forName( lClassName ); // load class
        if ( Modifier.isFinal( lClass.getModifiers() ) )
            System.out.println( "Error, cannot modify a final class." );
        else
        {
            Weaver lWeaver = new Weaver( lClass );
            lWeaver.buildNewClass();
        }
    }
    catch (Exception e)
    { System.out.println( e.getClass().getName() + ":" + e.getMessage() ); }
```

# Weaver Constructor

```
public Weaver( Class aClass )
{
    fClass = aClass;
    fClassName = aClass.getName();
    fPureClassName = fClassName.substring( fClassName.lastIndexOf( '.' ) + 1 );
    fNewClassName = "Trace" + fPureClassName;
    fName = fNewClassName + ".java";

    // not supported in J#
    Package lPackage = aClass.getPackage();
    fPackageName = lPackage.getName();
}
```

# BuildNewClass

```
public void buildNewClass() throws FileNotFoundException
{
    System.out.println( "Processing " + fNewClassName + "..." );

    fOutput = new PrintWriter( new FileOutputStream( fFileName ) );

    // write prolog...
    createConstructors();
    // write trace methods...
    weaveMethods();
    // write epilog...

    System.out.println( "Compiling " + fNewClassName + "..." );
    String[] lFiles = new String[1];
    lFiles[0] = fFileName;
    int res = com.sun.tools.javac.Main.compile( lFiles );
    if ( res == 0 ) System.out.println( "SUCCESS" );
    else System.out.println( "Oops, try again." );
}
```

# Weave Class

```
// write prolog...
fOutput.println( "public class " + fNewClassName +
                  " extends " + fPureClassName );
fOutput.println( "{" );

createConstructors();
createNewInstanceVariables();

createIndent();
createEntryIndent();
createExitIndent();
createBefore();
createAfter();

weaveMethods();
```

# weaveMethods

```
private void weaveMethods()
{
    Method[] lMethods = fClass.getMethods();

    for ( int i = 0; i < lMethods.length; i++ )
    {
        if ( mustWeave( lMethods[i] ) )
            weaveMethod( lMethods[i].getName() );
    }
}
```

# MustWeave

```
private boolean mustWeave( Method aMethod )
{
    Class IResType = aMethod.getReturnType();
    Class IParamTypes[] = aMethod.getParameterTypes();

    return
        (IParamTypes.length == 1) &&
        (IParamTypes[0].getName().equals( Long.TYPE.getName() )) &&
        (!Modifier.isFinal( aMethod.getModifiers() )) &&
        (IResType.getName().equals( Long.TYPE.getName() ));
}
```

Pointcut (long \* (long))?

# WeaveMethod

```
private void weaveMethod( String aMethodName )
{
    fOutput.println( "    public long " + aMethodName + "( long n )");
    fOutput.println( "    {");
    fOutput.println( "        try" );
    fOutput.println( "        {" );
    fOutput.println( "            __fThisJoinPoint = getClass().getMethod( \\" + aMethodName +
        "\\", " + "new Class[] { Long.TYPE } );" );
    fOutput.println( "            __before( n );" );
    fOutput.println( "            return __after( super." + aMethodName + "( n ), n );" );
    fOutput.println( "        }" );
    fOutput.println( "        catch ( Exception e )" );
    fOutput.println( "        {" );
    fOutput.println( "            System.out.println( e.getClass().getName() + \": \"+ e.getMessage() );" );
    fOutput.println( "            System.exit( 0 );" );
    fOutput.println( "        }" );
    fOutput.println( "        return 0;" );
    fOutput.println( "    }" );
}
```

# Reflection on Constructors

```
private void createConstructors()
{
    Constructor[] lConstructors = fClass.getConstructors();

    for ( int i = 0; i < lConstructors.length; i++ )
    {
        createConstructor( lConstructors[i] );
    }
}
```

# Create Constructor

```
private void createConstructor( Constructor aConstructor )
{
    if ( Modifier.isPublic( aConstructor.getModifiers() ) )
    {
        Class[] lExceptions = aConstructor.getExceptionTypes();
        Class[] lParamTypes = aConstructor.getParameterTypes();

        fOutput.print( "    public " + fNewClassName + "(" );
        // generate args...
        fOutput.print( ")" );

        // generate exceptions...

        fOutput.println( "\n    {" );
        fOutput.print( "        super(" );
        // generate args...
        fOutput.println( ";" );
        fOutput.println( "    }\n" );
    }
}
```

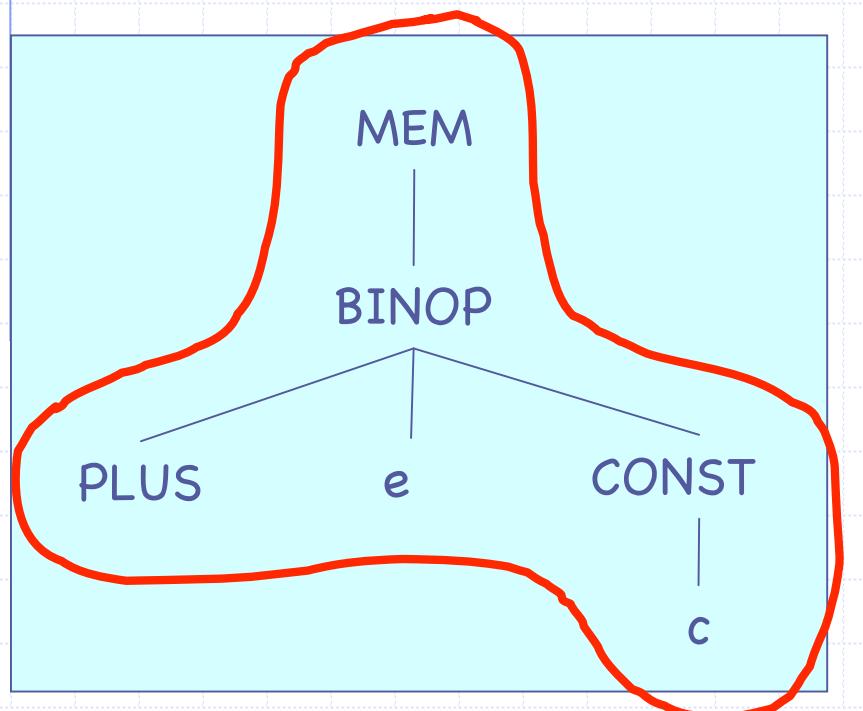
# Run Weaver

```
>java -classpath c:/j2sdk1.4.1_03/lib/tools.jar;../  
    TraceWeaver.MainClass  
Class: Fibonacci.Fibonacci  
Processing TraceFibonacci...  
Compiling TraceFibonacci...  
SUCCESS
```

# Code Generation

- The intermediate language (e.g., Three-Address Code, Simple Expression Trees) expresses only one operation in each node: memory fetch or store, addition or subtraction, conditional jump, etc.
- A real machine instruction can often perform several of these primitive operations.
- Finding the appropriate machine instructions to implement a given intermediate representation is the job of the *instruction selection* phase of a compiler.

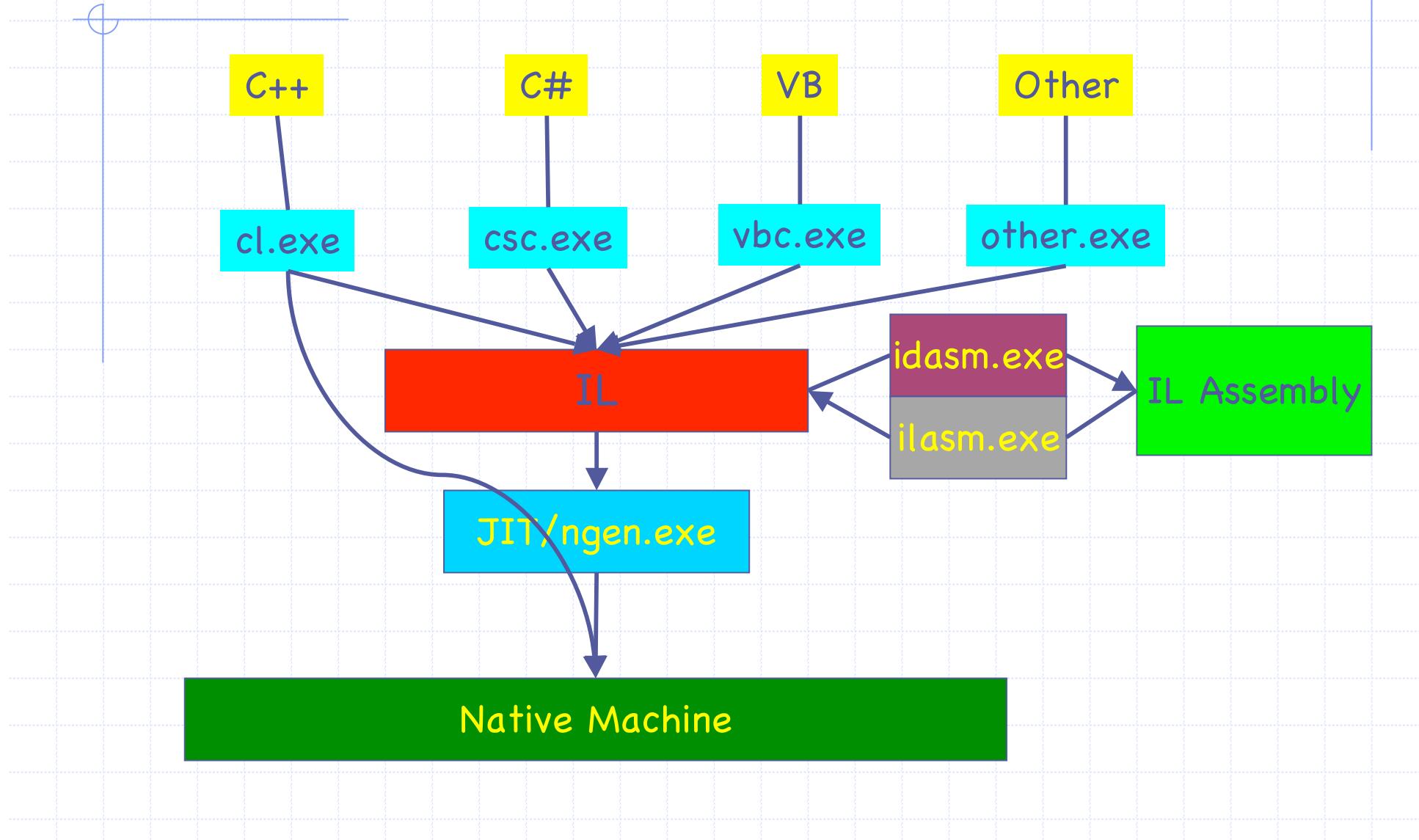
# Load Indirect



Pentium:  
MOV EAX, c(e)

CLR:  
ldc.i4 c  
ldloc.0  
add ; ldc.i4 e+c  
ldind.i4

# CLR Model



# HelloWorld IL Program

```
.assembly extern mscorlib {}

.assembly HelloWorld
{
    .ver 1:0:1:0
}

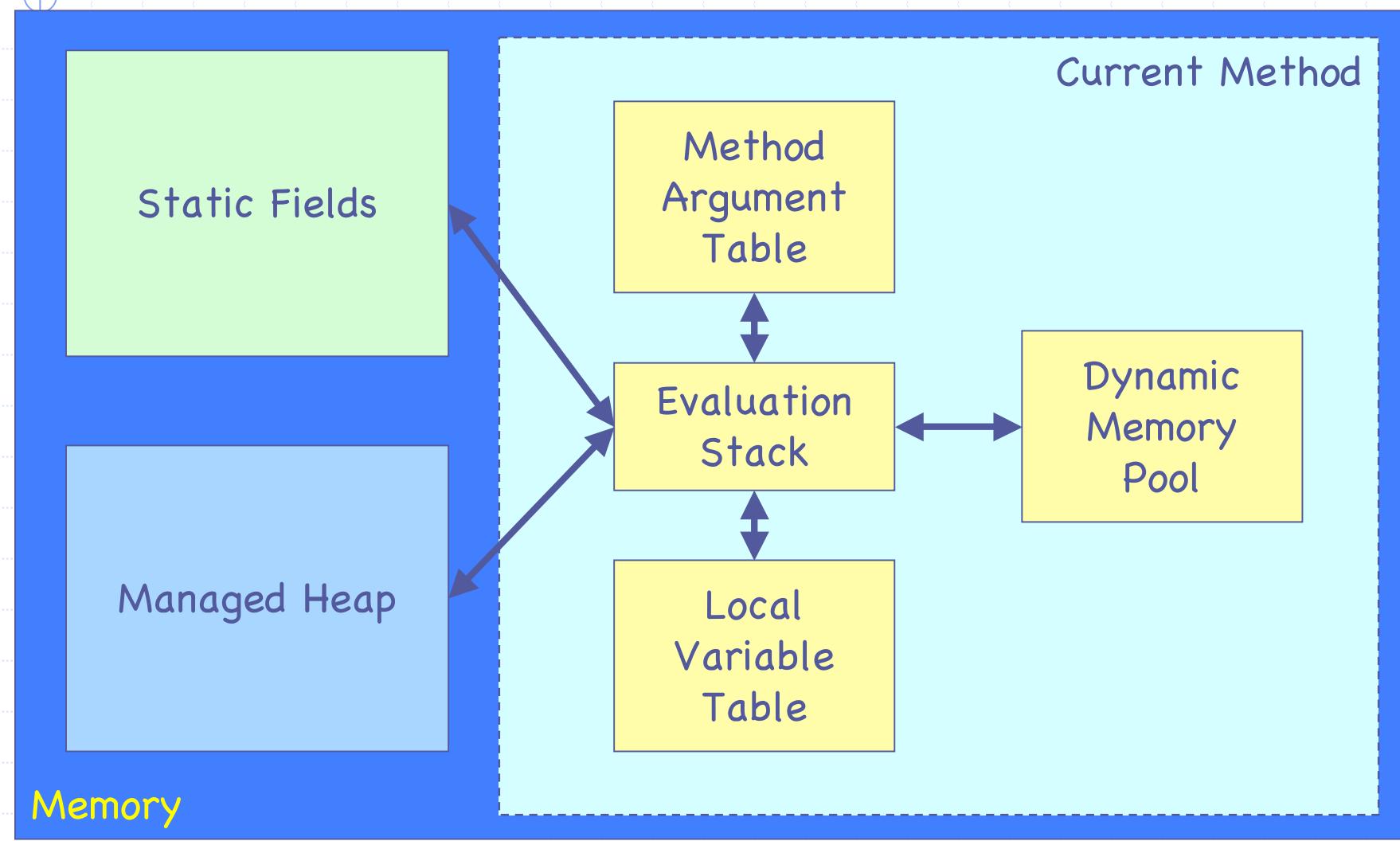
.module HelloWorld.exe

.method static void Main() cil managed
{
    .entrypoint
    .maxstack 1

    ldstr "Hello, World"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

iladm HelloWorld.il

# The .NET Machine Model



# Adding Two Numbers

```
.assembly extern mscorlib {}

.assembly AddConstants
{
    .ver 1:0:1:0
}
.module AddConstants.exe

.method static void Main() cil managed
{
    .entrypoint
    .maxstack 2

    ldstr "The sum of the numbers is"
    call void [mscorlib]System.Console::WriteLine(string)
    ldc.i4.s 47
    ldc.i4 345
    add
    call void [mscorlib]System.Console::WriteLine(int32)
    ret
}
```

# 00 HelloWorld IL Program

```
.assembly extern mscorlib {}

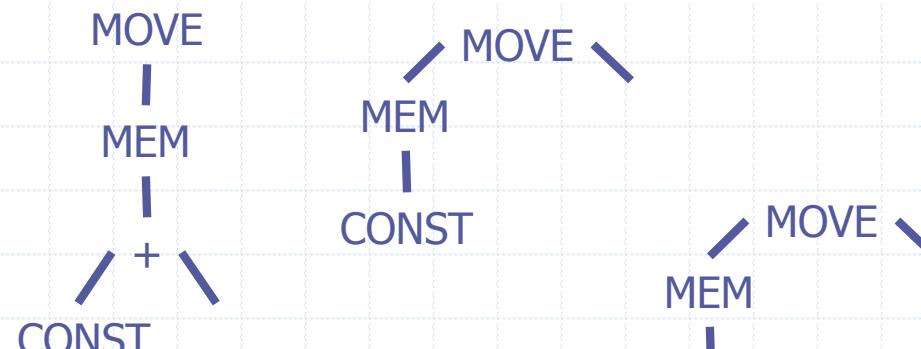
.assembly HelloWorldClass { .ver 1:0:1:0 }
.module HelloWorldClass.exe

.namespace Edu.IaState.Cs.Cs440.HelloWorldClass
{
    .class public auto ansi EntryPoint extends [mscorlib]System.Object
    {
        .method public static void DisplayHelloWorld() cil managed
        {
            .entrypoint
            .maxstack 1
            ldstr      "Hello, World"
            call       void [mscorlib]System.Console::WriteLine(string)
            ret
        }
    }
}
```

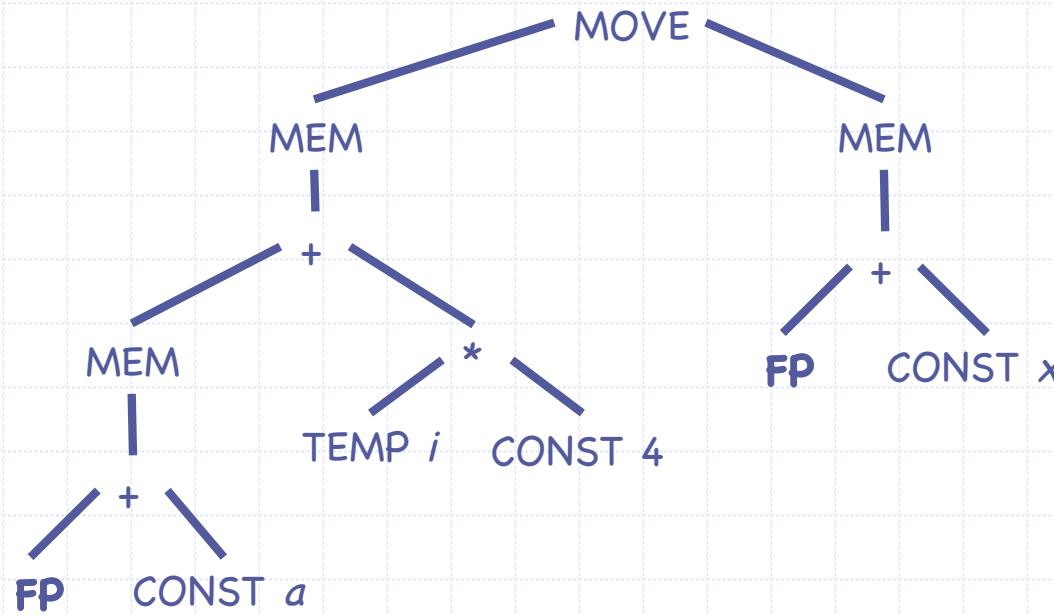
# Jouette

Name	Effect	Trees
-	..., value	TEMP
ADD SUB MUL DIV	... value, value → ..., result	<pre> graph TD     op --- plus[+]     plus --- const1[CONST]     plus --- const2[CONST]     </pre>
ADDI SUBI	..., value, c → ..., result	<pre> graph LR     plus1[CONST + CONST] --- plus1plus[+]     plus1plus --- plus1const1[CONST]     plus1plus --- plus1const2[CONST]     plus2[CONST +/- CONST] --- plus2minus[-]     plus2minus --- plus2const1[CONST]     plus2minus --- plus2const2[CONST]     </pre>
LOAD	... → ..., value	<pre> graph LR     plus3[CONST + MEM] --- plus3plus[+]     plus3plus --- plus3const3[CONST]     plus3plus --- plus3mem1[MEM]     plus4[CONST +/- MEM] --- plus4minus[-]     plus4minus --- plus4const4[CONST]     plus4minus --- plus4mem2[MEM]     </pre>

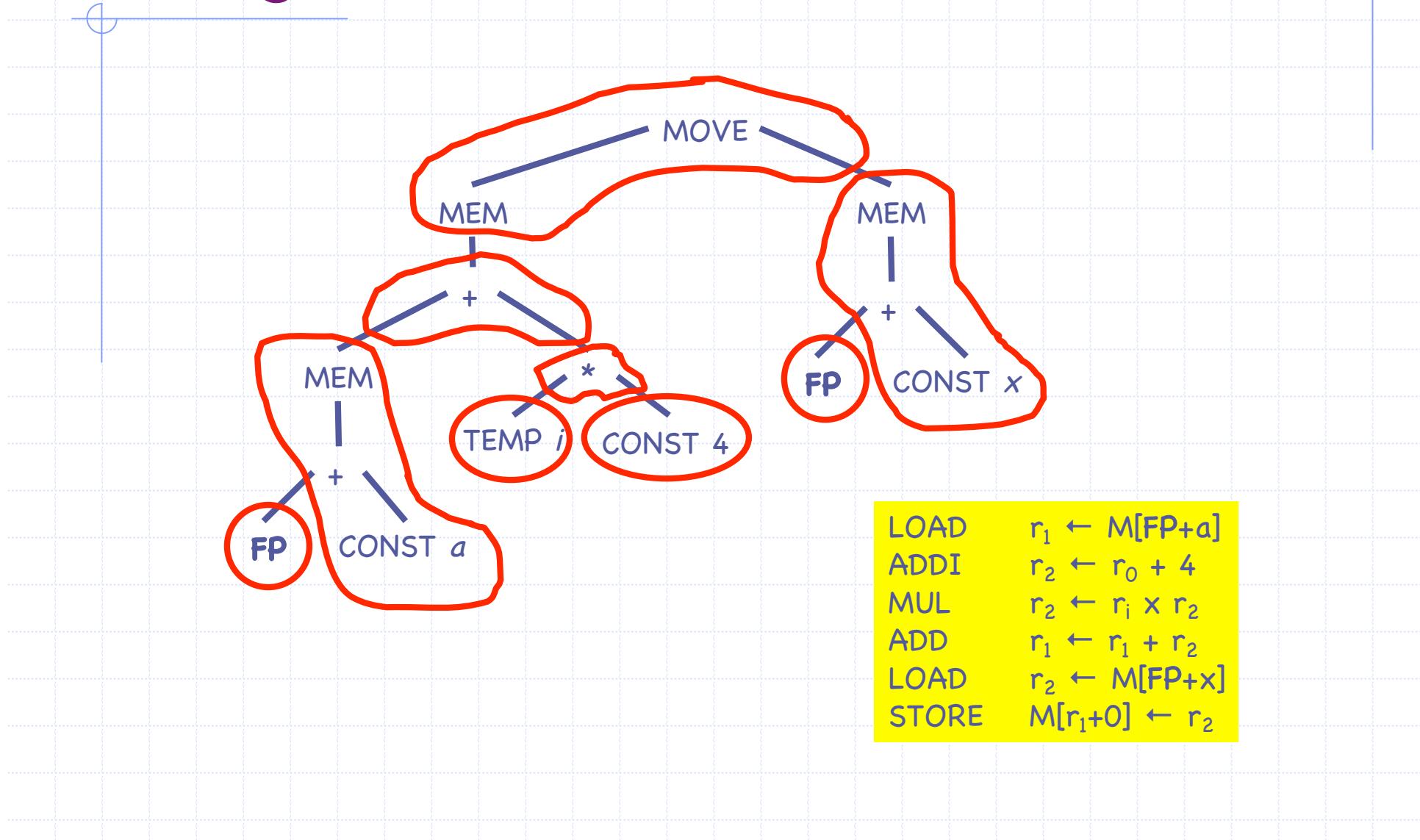
# Jouette II

Name	Effect	Trees
STORE	..., value → ...	
MOVEM	... → ...	

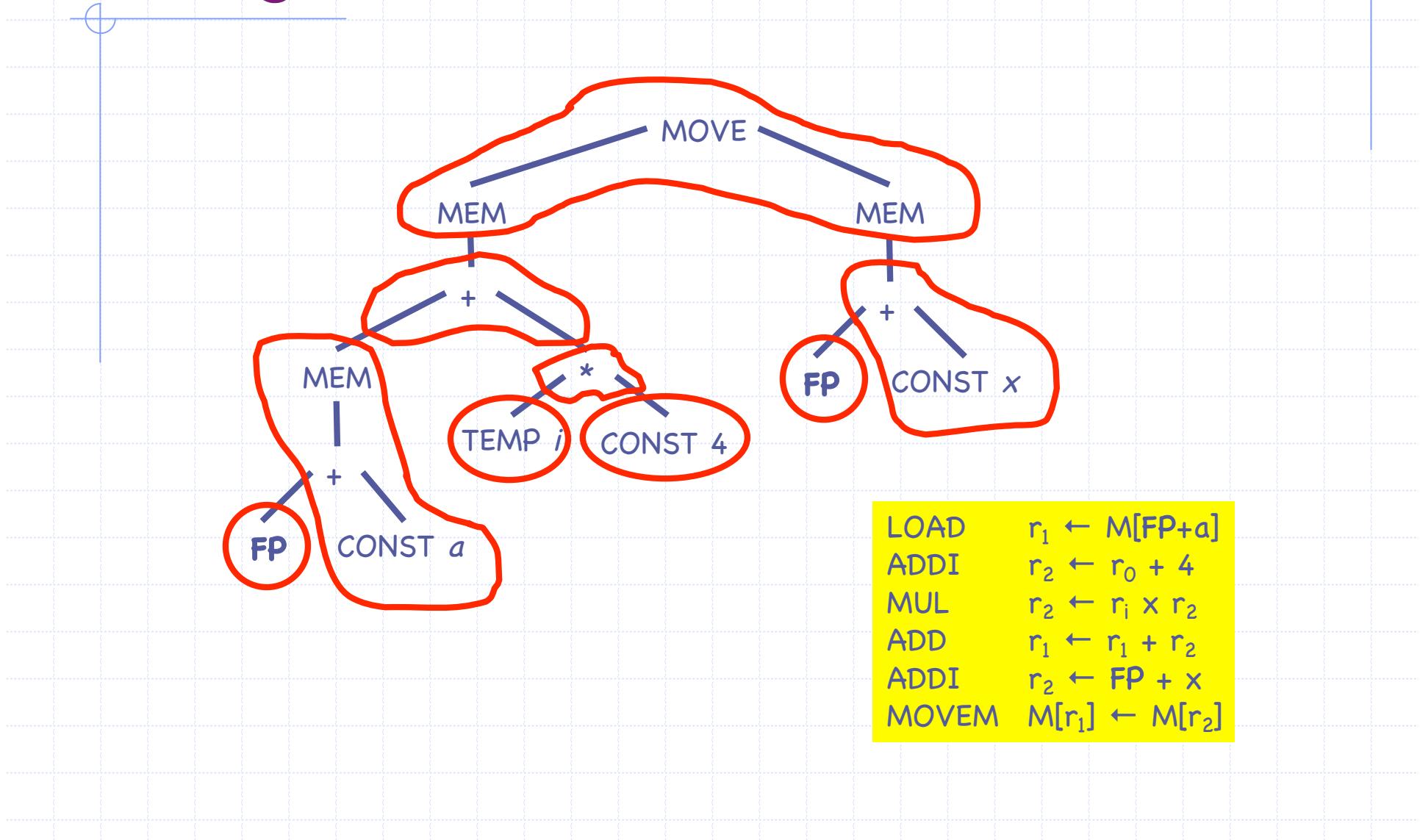
# A Expression Tree Example



# Tiling Version 1



# Tiling Version 2



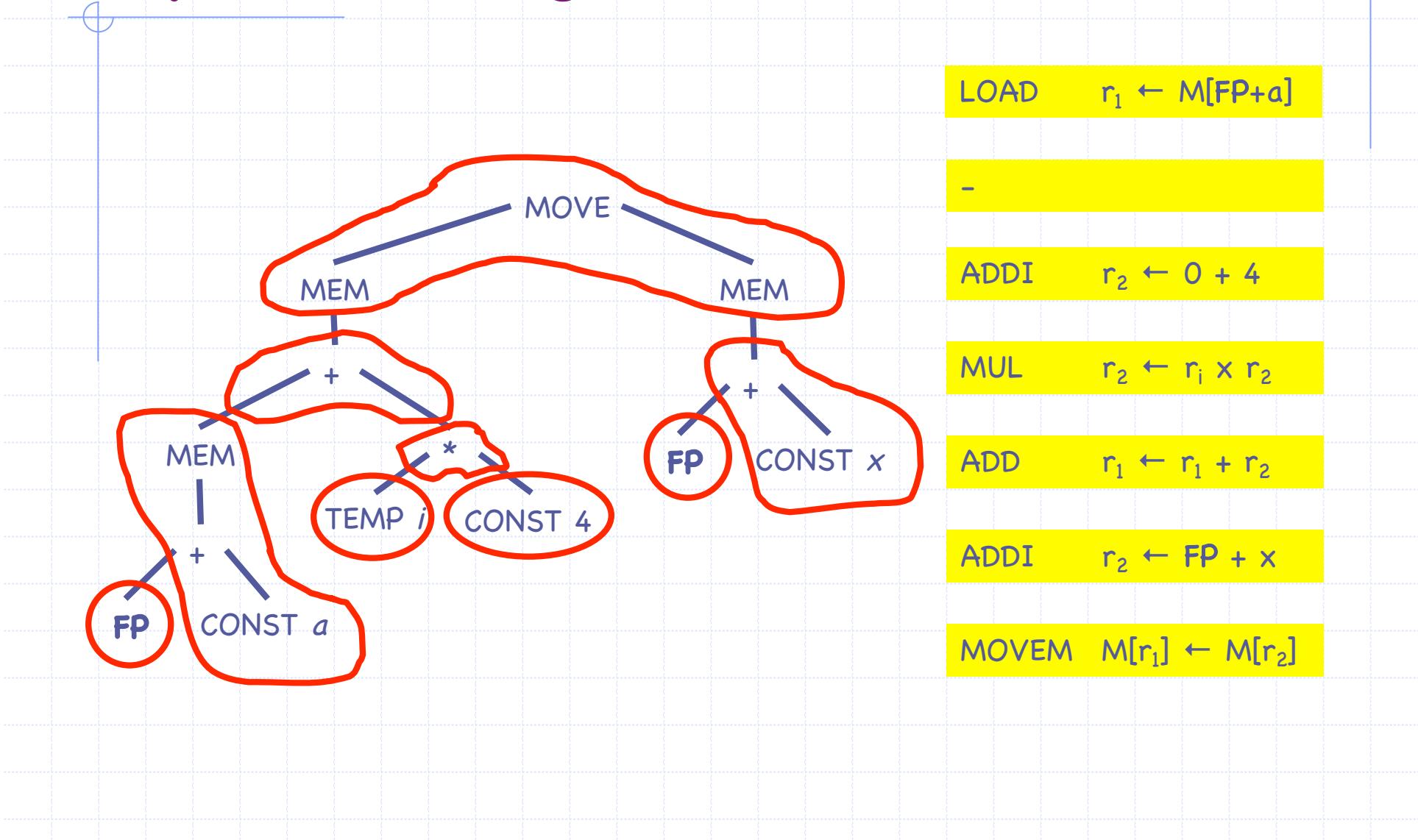
# Optimal and Optimum Tiling

- The best tiling of a intermediate representation tree corresponds to an instruction sequence of least cost:
  - the shortest sequence of instructions, or
  - if the instructions take different amounts of time to execute, the least-cost sequence has the lowest total time.
- An optimum tiling is one whose tiles sum to the lowest possible value.
- An optimal tiling is one where no two adjacent tiles can be combined into a single tile of lower cost. (Note, the tiling has to be deterministic.)
- Every optimum tiling is also optimal, but not vice versa.

# Maximal Munch Algorithm

- Optimal tiling:
  - Starting at the root of the simple expression tree, find the largest tile that fits.
  - Cover the root node and possibly several neighbor nodes with this tile, leaving several subtrees.
  - Repeat process for each remaining subtree.
- As each tile is placed, the instruction corresponding to that tile is generated.
- The maximal munch generates the instructions in reverse order – the instruction at the root is generated first, but it can only execute after the other instructions have produced operand values.

# Optimal Tiling



# CLR Code

Jouette	Stack Transitions	CLR
LOAD $r_1 \leftarrow M[FP+a]$	..., FP+c, → ..., value	ldloc <b>FP</b> ldc.i4    c add ldind.i4
ADDI $r_2 \leftarrow 0 + 4$	..., value, 0, 4 → ..., value, 4	ldc.i4.0 ldc.i4    4 add
MUL $r_2 \leftarrow r_i \times r_2$	..., value, 4, i → ..., value, 4*i	ldloc    i mul
ADD $r_1 \leftarrow r_1 + r_2$	..., value, 4*i → ..., L-ADDR	add
ADDI $r_2 \leftarrow FP + x$	..., L-ADDR, FP, x → ..., L-ADDR, R-ADDR	ldloc <b>FP</b> ldc.i4    x add
MOVEM $M[r_1] \leftarrow M[r_2]$	..., L-ADDR, R-ADDR → ...	ldind.i4 stind.i4

# Dynamic Programming

- The maximal munch algorithm finds the optimal tiling, but not necessarily an optimum tiling.
- A dynamic-programming algorithm can find the optimum.
- Dynamic programming is a technique for finding optimum solutions for a whole problem based on the optimum solution of each subproblem.

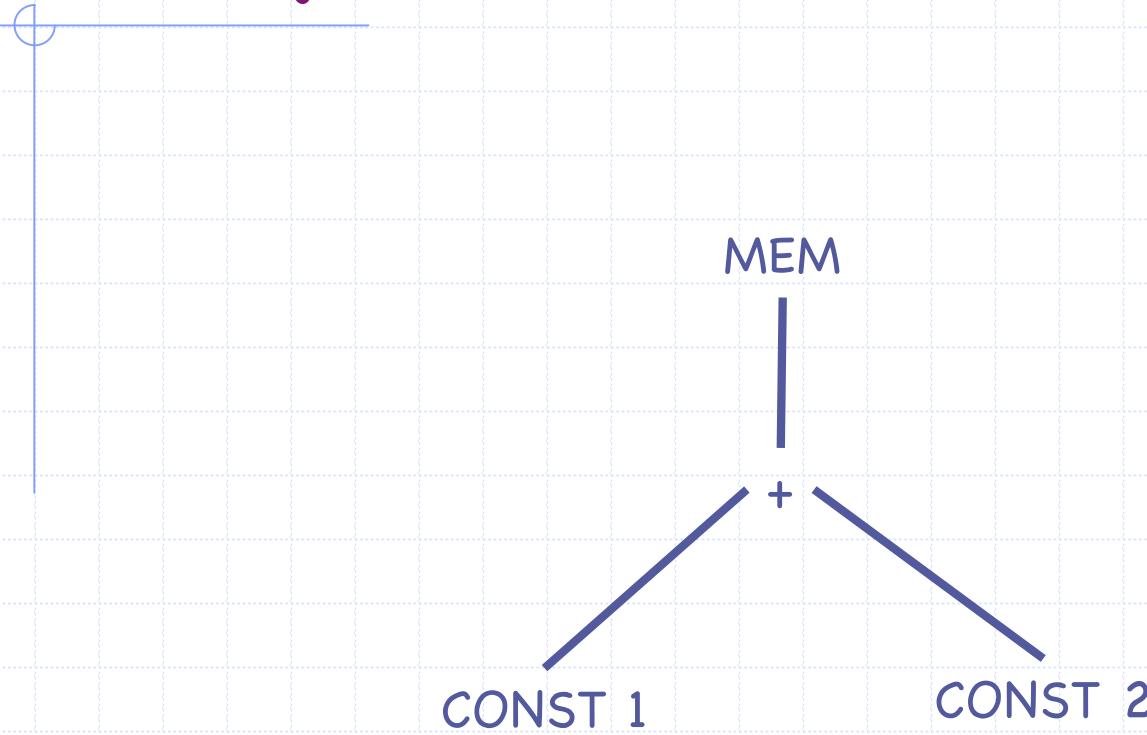
# Costs

- We assign every node in the simple expression tree representation a cost.
- The cost is the sum of the instruction costs of the best instruction sequence that can tile rooted at that node.
- Cost-assignment works bottom-up, in contrast to maximal munch, which works top-down. First, the costs of all children of node  $n$  are determined recursively. Then, each tree pattern (tile kind) is against node  $n$ .

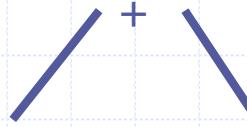
# Cost Structure

- Each tile has zero or more *leaves*. The leaves are represented as edges whose bottom ends exit the tile.
- For each tile  $t$  of cost  $c$  that matches at node  $n$ , there will be zero or more subtrees  $s_i$  corresponding to the leaves of the tile.
- The cost  $c_i$  of each subtree has already been computed, so the cost matching tile  $t$  is just  $c + \sum c_i$ .
- Of all the tiles  $t_j$  that match at node  $n$ , the one with the minimum-cost match is chosen, and the minimum cost of node  $n$  is computed.

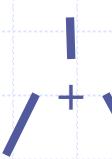
# Example



# PLUS Node

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
	ADD	1	1+1	3
 CONST	ADDI	1	1	2
 CONST	ADDI	1	1	2

# MEM Node

Tile	Instruction	Tile Cost	Leaves Cost	Total Cost
MEM 	LOAD	1	2	3
MEM  CONST	LOAD	1	1	2
MEM  CONST	LOAD	1	1	2

# Instruction Emission

- $\text{Emission}(\text{ node } n ):$   
For each leaf  $l_i$  of the tile selected at node  $n$ , perform  $\text{Emission}( l_i )$ . Then emit the instruction at node  $n$ .
- $\text{Emission}( n )$  does not recur on the children of node  $n$ , but on the *leaves* of the tile that matched at  $n$ .



# Garbage Collection

- Heap-allocated objects that are unreachable by any chain of references from program variables are *garbage*.
- The process of garbage collection, performed by the runtime system, reclaims memory occupied by garbage.
- The compiler has to guarantee that any *live* object is *reachable*. The compiler has to minimize the number of reachable object that are *not live*.

# Halting Problem

- It is not always possible to determine whether a variable is live or not.
- Given an arbitrary Turing machine and its input tape, will the machine eventually halt? → UNDECIDABLE

*last-try( x ) = will-stop( last-try ) && loop( x )*

The function *will-stop* returns true if the argument eventually stops. Otherwise, it returns false.

# Mark-And-Sweep Collection

- Program variables and heap-allocated objects form a directed graph.
- The variables are the *roots* of this graph.
- A node  $n$  is reachable if there is a path of directed edges  $r \rightarrow \dots \rightarrow n$  starting as some root  $r$ .
- Mark-And-Sweep need to consider both Stack-based variables and Non-Stack-based variables (e.g. CLR's Local Variable Table and Method Argument Table).

# Depth-First Search Mark

```
function DFS(x)
    if x is a reference into the
        heap
        if record x is not marked
            mark x
            foreach field f in record x
                DFS( f )
```

DFS is a recursive algorithm that uses a stack. In worst case the size of the stack is  $H$  activation records, which is larger than the entire heap.

# Sweep

```
function Sweep
    p ← first address in heap
    while p < last address in heap
        if record p is marked
            unmark x
        else let
            f1 be the first field in
            p
            in p.f1 ← freelist
            freelist ← p
        p ← p + sizeof p
```

# Cost of Garbage Collection

- Depth-first search takes time proportional to the amount of reachable data.
- The sweep phase takes time proportional to the size of the heap.
- Let  $R$  be the size of reachable data and  $H$  be the size of the heap. Then the cost of one garbage collection is

$$c_1 R + c_2 H$$

# Amortized Costs

amortized costs =  $\frac{\text{time spent collecting garbage}}{\text{amount of garbage reclaimed}}$



$$\text{amortized costs} = \frac{c_1 R + c_2 H}{H - R}$$

- If  $R$  is close to  $H$ , the cost becomes very large, since the garbage collection phase only reclaims a few words of garbage.
- If the ratio  $R/H$  after a collection is greater than 0.5, then the collector should increase the heap by asking the operating system for more memory.

# Explicit Stack

```
function DFS(x)
    if x is an unmarked reference into the heap
        mark x
        t ← 1
        stack[t] ← x
        while t > 0
            x ← stack[t]; t ← t - 1
            foreach field f that is an unmarked reference
            in x
                mark f
                t ← t + 1; stack[t] ← f
```

There could be a path of length  $H$  such that the stack of activation records required by DFS would be larger than the entire heap.

The stack can still grow to  $H$  words.

# Pointer Reversal

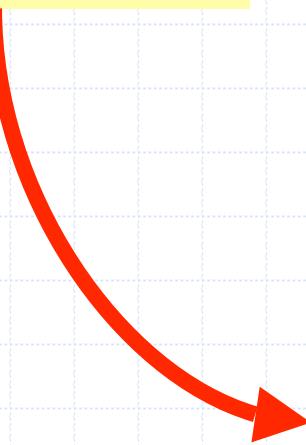
```
function DFS(x)
    if x is an unmarked reference into the heap
        t ← nil; mark x; done[x] ← 0
        while true
            i ← done[x]
            if i < # of fields in record x
                y ← x.fi
                if y is an unmarked reference into the heap
                    x.fi ← t; t ← x; x ← y; mark x; done[x] ← 0
                else
                    done[x] ← I + 1
                else
                    y ← x; x ← t
                    if x = nil then return
                    i ← done[x]; t ← x.fi; x.fi ← y; done[x] ← i + i
```

# Reference Counting

- We can count the numbers of references to a specific record in the heap.
- Each time a heap-based object is assigned to a variable the object's reference count is incremented and the reference count of what the variable previously pointed to is decremented.
- The compiler can emit the necessary code.

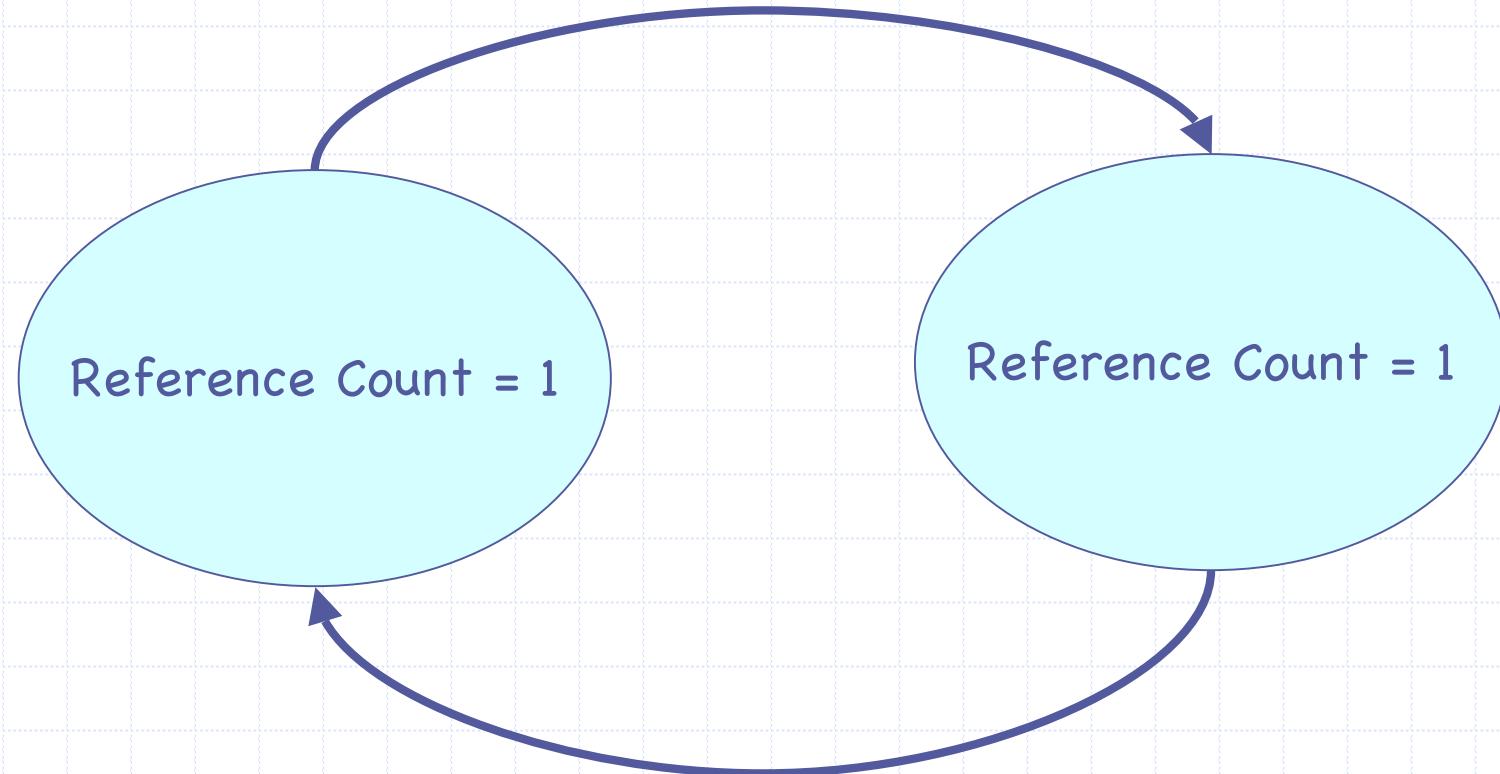
# Maintaining the Counts

x = heap object



```
temp ← x
count ← temp.count
count ← count - 1
temp.count ← count
if count == 0 call releaseObject
x ← heap object
count ← heap object.count
count ← count +1
heap object.count ← count
```

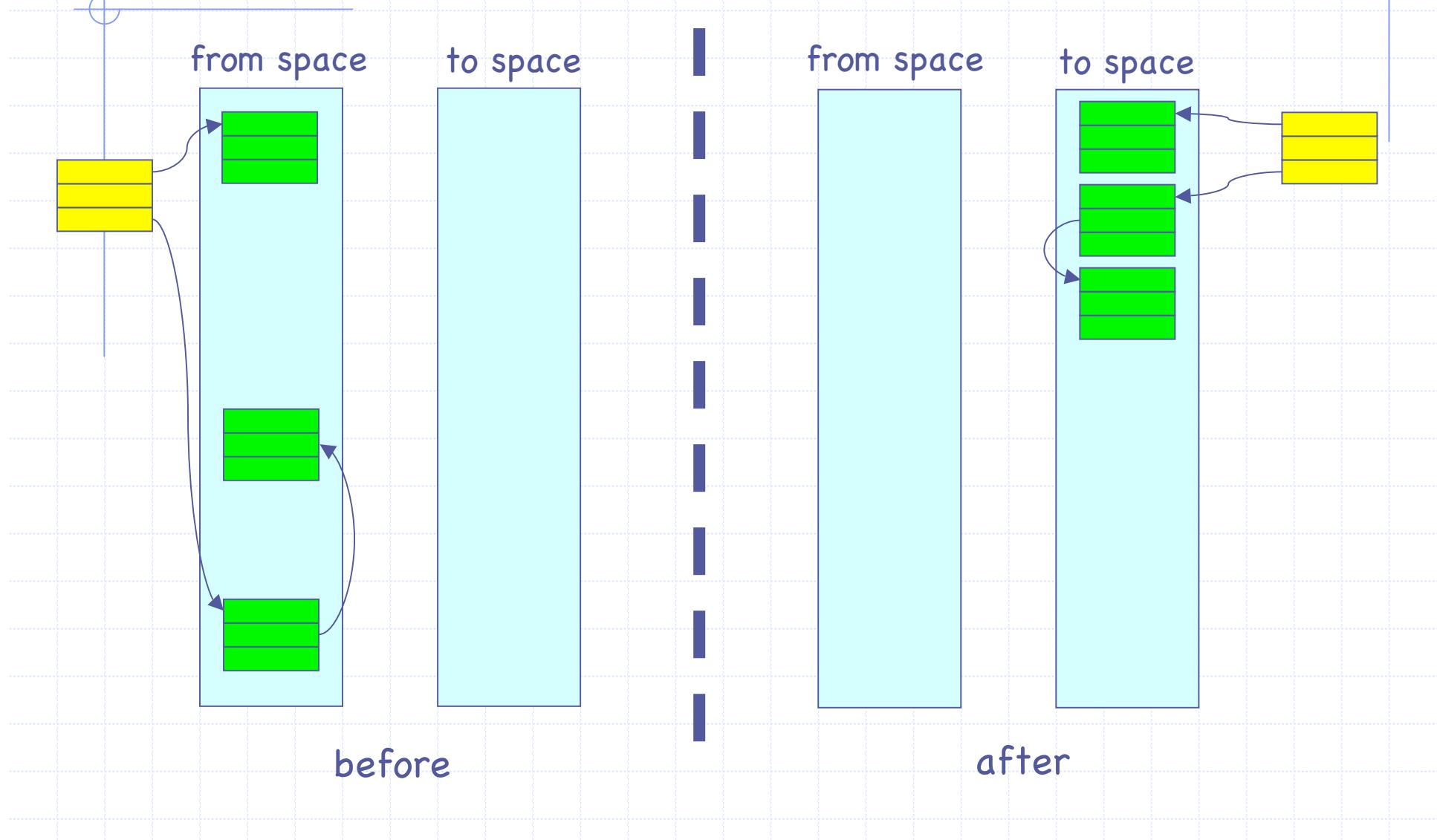
# Cycles



# Conclusion

- The problems of reference counting outweigh its advantages, and it is rarely used for automatic storage management in programming language environments.
- What about COM/ActiveX?

# Copying Collection



# Variations

- Generational collection (managing lifetime)
- Incremental collection (tri-color marking)