

# Module-1

## Introduction to System Software

# 1.1 What is System Software?

System Software consists of a variety of programs that supports the operation of a computer. This software makes its possible for the user to focus on an application without needing to know the detail of how the machine works internally.

## Difference between system software & application software:

System Software	Application Software
It is a program or group of programs written for a computer system management.	It is a program or collection of programs written to solve a particular problem.
These are developed by the manufacturers.	These are developed by users.
System software control & manage the hardware.	Application software uses the services of the system software to interact with the hardware.
Development of system software is complex task.	Development of application software is relatively easier.
Ex: Operating System, Compilers, Assemblers, Loaders, Linkers, Text editors etc.	MS-WORD, MS-EXCEL, Payroll inventory system, Student management system, Library Management System etc.

# 1.2 System Software v/s. Machine Architecture

- **Machine Dependent**

- The most important characteristic in which most system software differ from application software is **machine Dependency**.

e.g. Assembler translate **mnemonic instructions** into machine code

e.g. Compilers Translate **Program in high-Level languages** like C,C++, JAVA etc. into machine code.

- Machine architecture differs in:

- Instruction Set

- Instruction formats

- Addressing Modes

- Registers support

- **Machine independent**

- There are aspects of system software that **do not directly depend upon the type of computing system /Machine**

e.g. General design and logic of an assembler is same on most computers.

e.g. Code optimization techniques used by compilers.

## 1.3 The Simplified Instructional Computer (SIC)

- SIC is a hypothetical computer that includes the hardware features most often found on real machines.
- Why the simplified instructional computer:
  - To focus on central, fundamental, and commonly encountered features and concepts.
- Two versions of SIC:
  1. Standard model (SIC)
  2. Extension version (SIC/XE)
- **Upward compatible**
  - Program for SIC can run on SIC/XE

# SIC Machine Architecture( Standard version)

- Memory
  - $2^{15}$  (32,768) bytes in the computer memory.
  - 3 consecutive bytes form a **word**.
  - byte addressable memory
- Registers ( 5 Nos, 24 bits in length )

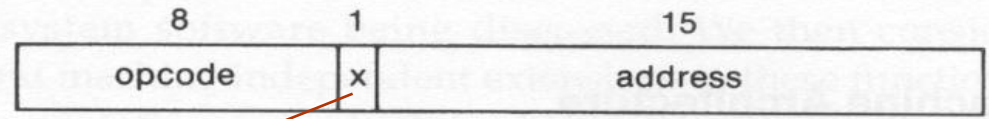
Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

# SIC Machine Architecture Contd...

- Data Format

- Integers are stored as **24-bit binary numbers**; 2's complement representation is used for negative values.
- 8-bit character support using ASCII code.
- **No floating-point hardware**

- Instruction Formats



x: indicate indexed - addressing mode

- Addressing Modes:

Mode	Indication	Target address calculation
Direct	$x = 0$	$TA = \text{address}$
Indexed	$x = 1$	$TA = \text{address} + (X)$

# SIC Machine Architecture Contd...

- Instruction Set

- **Load and Store:** LDA, LDX, STA, STX etc.

- **Integer arithmetic operations:** ADD, SUB, MUL, DIV, etc.

- All arithmetic operations involve register A and a word in memory, with the result being left in the register(A).

- **Comparison:** COMP

- COMP compares the value in register A with a word in memory, this instruction **sets a condition code CC to indicate the result.**

# SIC Machine Architecture Contd...

- **Conditional jump instructions:** JLT, JEQ, JGT

These instructions test the setting of CC and jump accordingly.

- **Subroutine linkage:** JSUB, RSUB

- JSUB jumps to the subroutine, placing the return address in register L (**Linkage register**)

- RSUB returns by jumping to the address contained in register L



# SIC Machine Architecture Contd...

- Input and Output(IO)

- Input and output are performed by transferring 1 byte at a time to or from the rightmost 8 bits of register A.
- **The Test Device (TD)** instruction tests whether the addressed device is ready to send or receive a byte of data
- Read Data (RD)
- Write Data (WD)

# SIC Programming Example

```
LDA      FIVE      LOAD CONSTANT 5 INTO REGISTER A
STA      ALPHA     STORE IN ALPHA
LDCH     CHARZ     LOAD CHARACTER 'Z' INTO REGISTER A
STCH     C1        STORE IN CHARACTER VARIABLE C1
.
```

Address labels

```
ALPHA   RESW      1      ONE-WORD VARIABLE
FIVE    WORD      5      ONE-WORD CONSTANT
CHARZ   BYTE      C'Z'   ONE-BYTE CONSTANT
C1      RESB      1      ONE-BYTE VARIABLE
```

Assembler directives  
for defining storage

# SIC/XE Machine Architecture

- **Memory**
  - $2^{20}$  bytes in the computer memory
  - 3 consecutive bytes form a **word**.
  - byte addressable memory
- **Registers(Additional)**

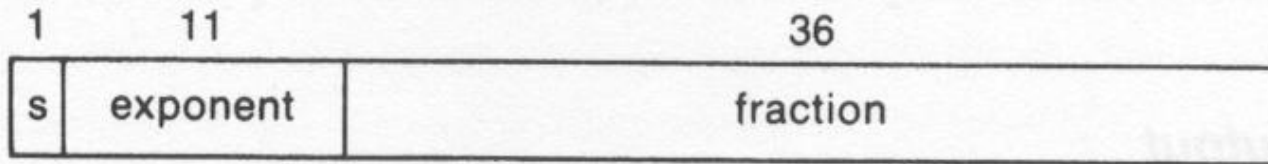
Mnemonic	Number	Special use
A	0	Accumulator; used for arithmetic operations
X	1	Index register; used for addressing
L	2	Linkage register; the Jump to Subroutine (JSUB) instruction stores the return address in this register
PC	8	Program counter; contains the address of the next instruction to be fetched for execution
SW	9	Status word; contains a variety of information, including a Condition Code (CC)

SIC

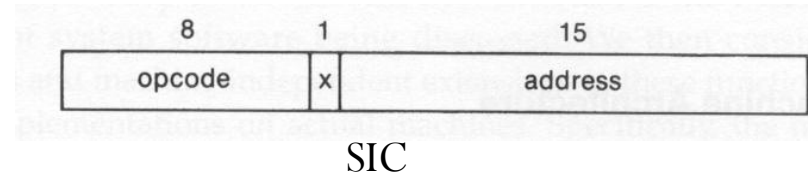
Mnemonic	Number	Special use
B	3	Base register; used for addressing
S	4	General working register—no special use
T	5	General working register—no special use
F	6	Floating-point accumulator (48 bits)

# SIC/XE Machine Architecture Contd..

- Data Format(Additional)
  - Floating-point data type

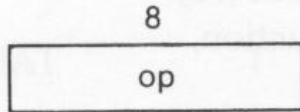


# SIC/XE Machine Architecture Contd...



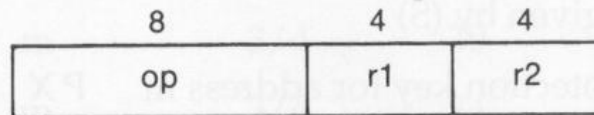
## • Instruction formats

Format 1 (1 byte):



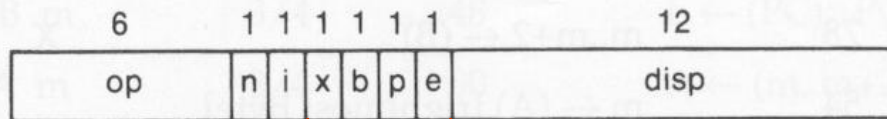
No memory reference

Format 2 (2 bytes):



No memory reference

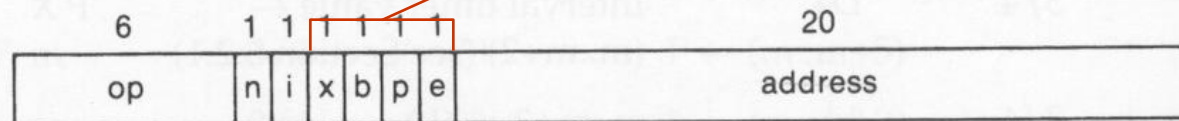
Format 3 (3 bytes):



Relative addressing

e=0

Format 4 (4 bytes):



Extended address field

for target address calculation

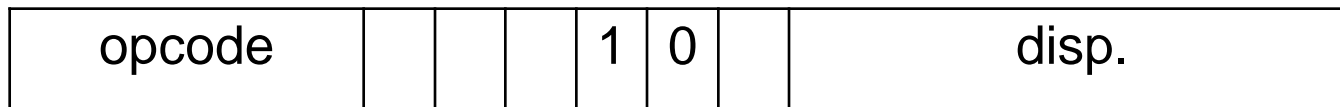
e=1

# SIC/XE Machine Architecture Contd...

## • Addressing Modes:

### ➤ Base Relative Addressing Mode

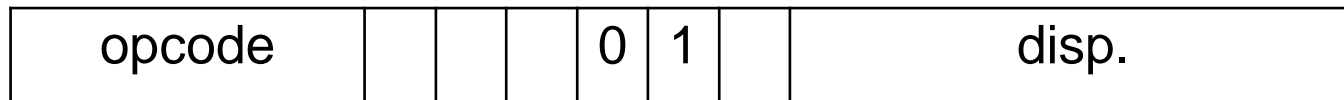
n i x b p e



$b=1, p=0, TA=(B)+disp \quad (0 \leq disp \leq 4095)$

### ➤ Program-Counter Relative Addressing Mode

n i x b p e

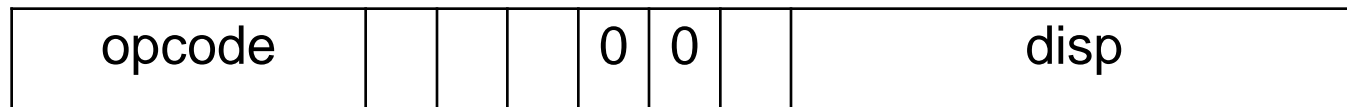


$b=0, p=1, TA=(PC)+disp \quad (-2048 \leq disp \leq 2047)$

# SIC/XE Machine Architecture Contd...

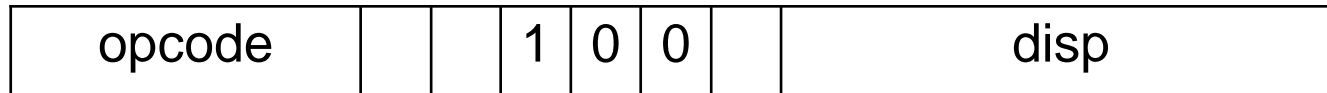
## ➤ Direct Addressing Mode

n i x b p e



$b=0, p=0, TA=disp$  ( $0 \leq disp \leq 4095$ )

n i x b p e



$b=0, p=0, TA=(X)+disp$   
(with index addressing mode)

# SIC/XE Machine Architecture Contd...



## ➤ Immediate Addressing Mode

n i x b p e



$n=0, i=1, x=0, \text{Operand}=\text{disp}$

## ➤ Indirect Addressing Mode



n i x b p e



$n=1, i=0, x=0, \text{TA}=(\text{disp})$

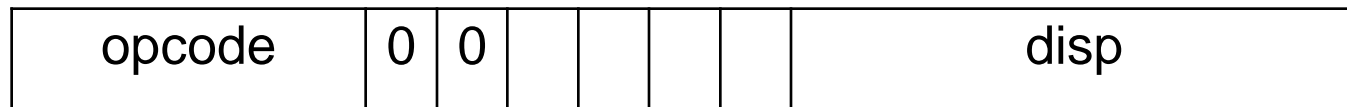
Note: Indexing cannot be used with **immediate** or **indirect** addressing modes.



# SIC/XE Machine Architecture Contd...

## ➤ Simple Addressing Mode

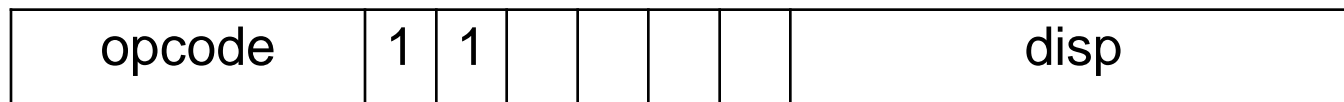
n i x b p e



$i=0, n=0, TA = bpe+disp$  (SIC standard)

$opcode+n+i = \text{SIC standard opcode (8-bit)}$

n i x b p e



$i=1, n=1, TA=disp$  (SIC/XE standard)

## SIC/XE Machine Architecture Contd...

### • Instruction set

- Load and store registers
  - **LDA, LDX, STA, STX**, LDB, STB, ...
- Integer arithmetic operations
  - **ADD, SUB, MUL, DIV**, ADDF, SUBF, MULF, DIVF, ADDR, SUBR, MULR, DIVR
- Comparison **COMP**
- Conditional jump instructions (according to CC)
  - **JLE, JEQ, JGT**
- Subroutine linkage
  - **JSUB**
  - **RSUB**

### • Input and output

Three instructions:

- **Test device (TD)**
- **Read data (RD)**
- **Write data (WD)**

# SIC/XE Programming Example

LDA	#5	LOAD VALUE 5 INTO REGISTER A
STA	ALPHA	STORE IN ALPHA
LDA	#90	LOAD ASCII CODE FOR 'Z' INTO REG A
STCH	C1	STORE IN CHARACTER VARIABLE C1
.		
.		
.		
ALPHA	RESW	1 ONE-WORD VARIABLE
C1	RESB	1 ONE-BYTE VARIABLE

(b)

# Module-2

## Content

### **Introduction:**

Language Processors, The structure of a compiler, The evolution of programming languages, Applications of compiler technology.

### **Lexical Analysis:**

The role of lexical analyzer, Input buffering, Specifications of token, recognition of tokens.

**Text Book:** Compilers Principles, Techniques, and Tools by  
Alfred V.Aho, Ravi Sethi, and Jeffrey D. Ullman, Addison-Wesley

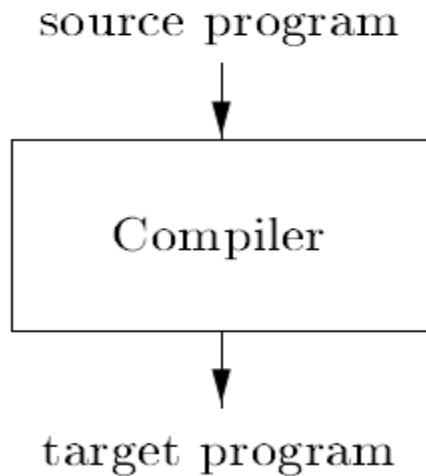
# Introduction

- Programming Languages are notations for describing computations to people and the Machines.
- The Software running on all the computers was written in some Programming Languages.
- Before a program can be run , it first must be translated into a form in which it can be executed by a Computer.
- The System Software that do this translation are called Compilers*

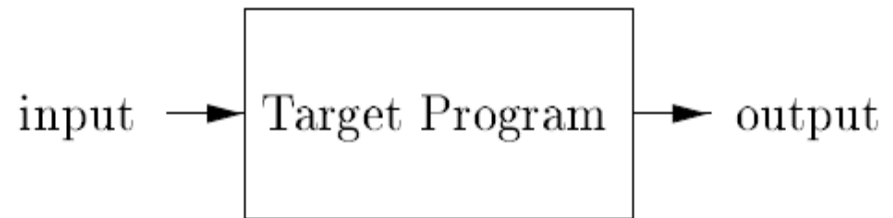
# Language Processor: Compiler.

A compiler is a program that can read a program in one language the *source* language - and translate it into an equivalent program in another language - the *target* language;

An important role of the compiler is to report any errors in the source program that it detects during the translation process



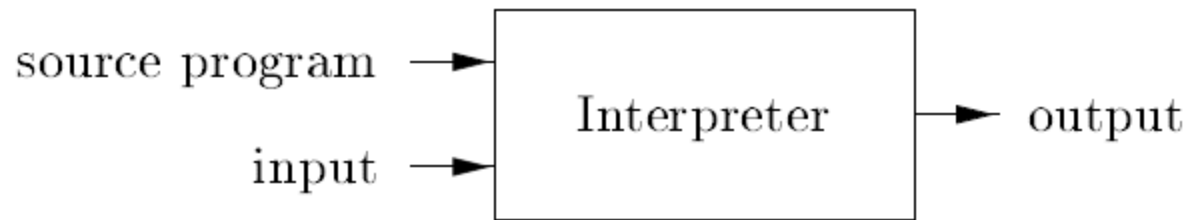
A Compiler



Running the Target Program

# Language Processor: Interpreter.

An interpreter is another common kind of language processor. Instead of producing a *target* program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.



**An Interpreter**

# Compiler *Versus* Interpreter

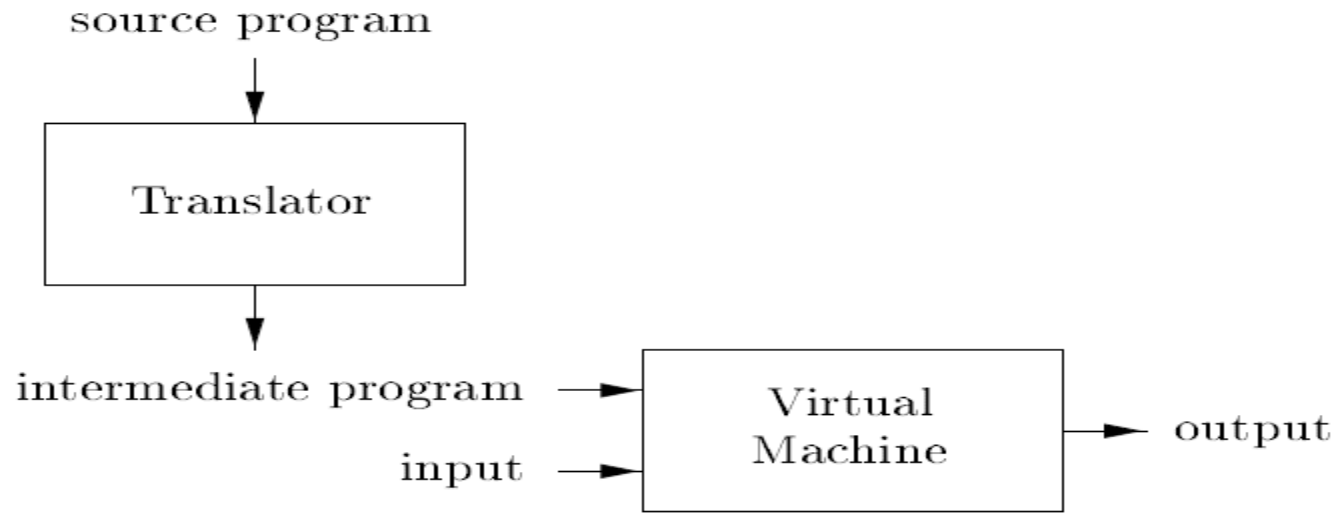
Interpreter	Compiler
Translates program one statement at a time.	Scans the entire program and translates it as a whole into machine code.
Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.	Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.
No Object Code is generated, hence are memory efficient.	Generates Object Code which further requires linking, hence requires more memory.
Programming languages like JavaScript, Python, Ruby use interpreters.	Programming languages like C, C++, Java use compilers.

Debugging is comparatively easy while working with an Interpreter.

Debugging of the program is comparatively complex while working with a compiler.



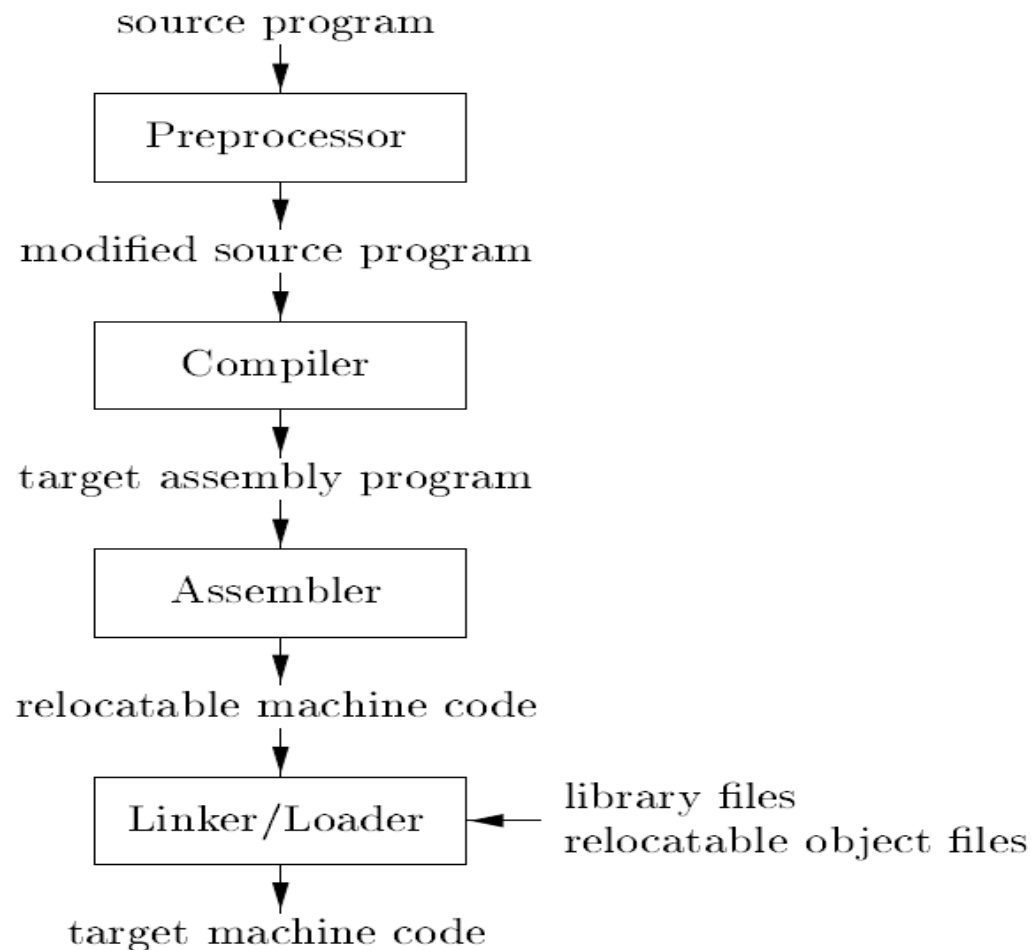
# Language Processor: Hybrid Translator



## A Hybrid Translator

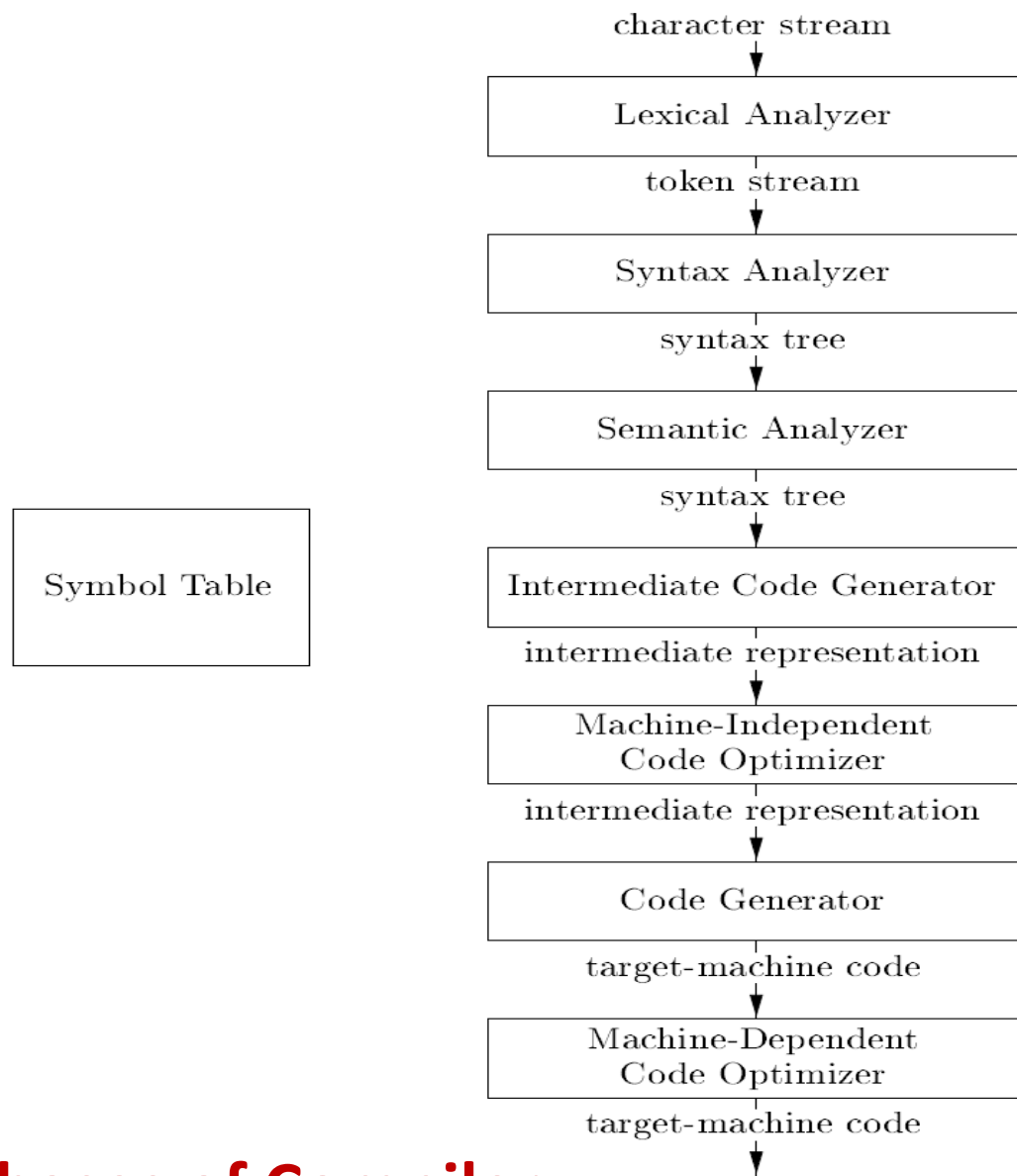
- A Java source program may first be compiled into an intermediate form called *bytecodes*.
- The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.
- In order to achieve faster processing of inputs to outputs, some Java compilers, called *just-in-time* compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

# Language Processing System



***In addition to a Compiler, many other programs(System) may required to create an executable Target Machine code as shown above fig.***

# The Structure of a Compiler



**The phases of Compiler**

# The Analysis-Synthesis Model of Compilation:

- There are two parts of compilation:
  - Analysis
  - Synthesis

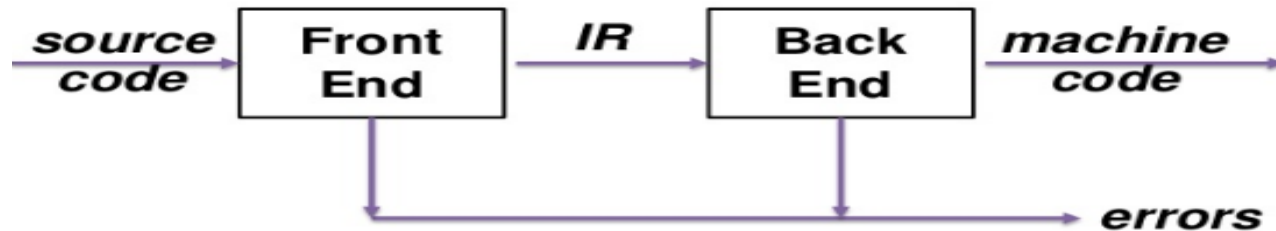
The *Analysis* part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program.

-If the analysis part detects that the source program contains an error, then it must provide informative messages, so the user can take corrective action.

-Collects information about source program and stores it in a data structure called a symbol table for later use by all phases.

The *Synthesis* part constructs the desired target program from the intermediate representation (IR) and the information in the symbol table.

The analysis part is often called the **Front end**(Machine-Independent but Language Dependent) of the compiler and synthesis part is called **Back end** (Machine dependent)



### **Lexical Analysis:**

Lexical Analyzer(Scanner) reads the stream of characters from the source program and groups characters into a meaningful sequences called Lexemes and produce as an output a **token** for each lexeme in the source program. Token format:  $\langle token-name, attribute-value \rangle$

### **Syntax Analysis:**

The second phase of compilation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc. are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language and produce a syntax tree (also called Parse tree)

## **Semantic Analysis:**

The semantic analyzer use the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It also does the type checking and type casting kind of activities.

## **Intermediate Code Generations:**

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine. This intermediate representation should have two important properties: it should be easy to produce and it should be easy to translate into the target machine.

## **Code Optimization :**

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

## **Code Generation:**

The code generator takes as input an intermediate representation of the source program and maps it into the target language.

## **Symbol Table :**

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly. Symbol tables are discussed in

This data structure is implemented using Hashing Technique and its Dynamic

## **Error Handler:**

It is invoked when an errors in the source program is detected by compiler to handle the errors.

# Translation of an assignment statement

position = initial + rate \* 60

Lexical Analyzer

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Syntax Analyzer

$\langle \text{id}, 1 \rangle = \langle \text{id}, 2 \rangle + \langle \text{id}, 3 \rangle * 60$

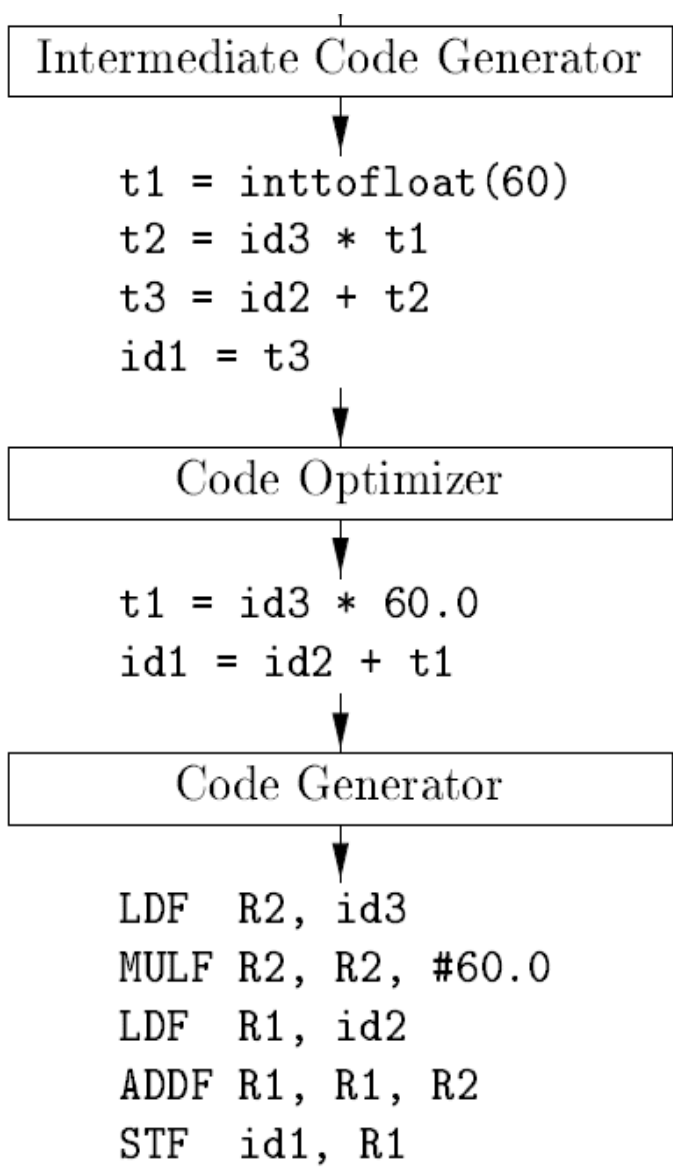
Semantic Analyzer

$\langle \text{id}, 1 \rangle = \langle \text{id}, 2 \rangle + \langle \text{id}, 3 \rangle * \text{inttofloat}(60)$

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE





# Compiler Construction Tools

Some commonly used compiler-construction tools include

1. *Parser generators* that automatically produce syntax analyzers from a grammatical description of a programming language.
2. *Scanner generators* that produce lexical analyzers from a regular-expression description of the tokens of a language.
3. *Syntax-directed translation engines* that produce collections of routines for walking a parse tree and generating intermediate code.
4. *Code-generator generators* that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. *Data-flow analysis engines* that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
6. *Compiler-construction toolkits* that provide an integrated set of routines for constructing various phases of a compiler.

# The Evolution of Programming Languages

**To build programs, people use languages that are similar to human language. The results are translated into machine code, which computers understand.**

**Programming languages fall into three broad categories:**

- **Machine languages**
- **Assembly languages**
- **Higher-level languages**

Advantages of Machine Language

- Translation Free
- High Speed

Disadvantages of Machine Language

- Machine Dependent
- Complex Language
- Error Prone
- Tedious

# *Assembly Language*

- **Assembly languages (second-generation languages) are only somewhat easier to work with than machine languages.**
- **To create programs in assembly language, developers use cryptic English-like phrases to represent strings of numbers.**
- **The code is then translated into object code, using a translator called an assembler.**

- Advantages of Assembly Language
  - Easy to understand and use
  - Less Error Prone
  - Efficiency
  - More control on Hardware
- Disadvantages of Assembly Language
  - Machine Dependent
  - Harder to Learn
  - Slow Development Time
  - Less Efficient

# *High-Level Language*

**Higher-level languages are more powerful than assembly language and allow the programmer to work in a more English-like environment.**

**Ex.:**

Procedure-Oriented Languages: FORTRAN, COBOL, PASCAL, C etc.

Object-Oriented Languages: C++, C#, JAVA, Python, Ruby etc.

Scripting Languages: JavaScript, Perl, PHP, Python etc

## Advantages of HLL

- Readability
- Machine Independent
- Easy debugging
- Easier to maintain
- Easy Documentation

## Disadvantages of HLL

- Poor control on Hardware
- Less Efficient

# Applications of Compiler Technology

Compiler technology has other important uses. Additionally, compiler design impacts several other areas of computer science. In this section, we review the most important interactions and applications of the technology.

## 1. Implementation of High-Level Programming Languages

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language.

## 2. Optimizations for Computer Architectures

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *Parallelism* and *Memory hierarchies*.

***Parallelism***: can be found at several levels: at the instruction level, where multiple operations are executed simultaneously and at the processor level, where different threads of the same application are run on different processors.

***Memory hierarchies***: Memory hierarchies consists of several levels of storage with different speeds and sizes, with the level closest to the processor being the fastest but smallest. Memory hierarchies are found in all machines to enhance the performance of the memory system.

### 3. Program Translations

While we normally think of compiling as a translation from a high-level language to the machine level, the same technology can be applied to translate between different kinds of languages. The following are some of the important applications of program-translation techniques.

**Binary Translation:** Compiler Technology can be used to translate the binary code for one machine to that of another.

**Hardware Synthesis:** Translating High level Hardware description written in Languages like Verilog and VHDL into Hardware (physical) Circuit Design.

**Query Interpreter:** Database Queries can be compiled into commands to search database for records.

### 4. Software Productivity Tools

Many of the **Data-flow –Analysis** Techniques , originally developed for compiler optimizations, can be used to assist the programmers in their software Engineering tasks to enhance the software productivity. Some of the important software productivity tools include:

**Type Checking:** Is an effective and well established technique to catch inconsistencies in programs. It can be used to catch errors, where an operation is applied to the wrong type of data object. It can also be used to catch a variety of security holes in software.

## Bounds Checking

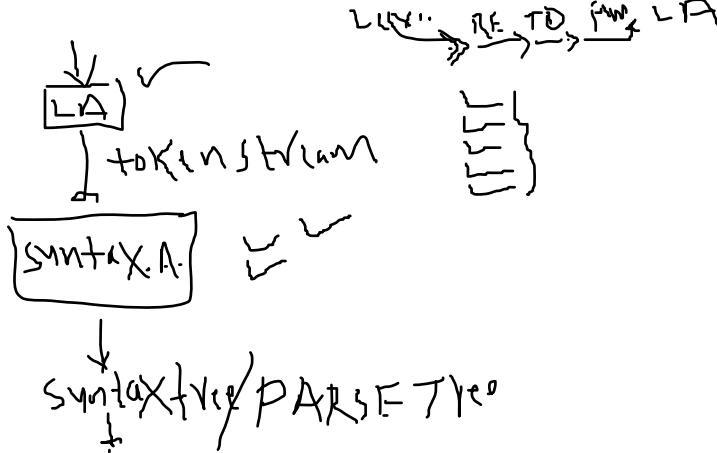
Many security breaches in systems are caused by buffer overflows in programs, written in languages like C. Because C does not have array bounds checks, it is up to the programmer to ensure that the arrays are not accessed out of bounds - unsafe Language( C) and may leads to the security compromise. The program written in safe Language(JAVA) that includes automatic bound checking, this problem would not have occurred.

## Memory-Management Tools

Garbage collection is another excellent example of the tradeoff between efficiency and a combination of ease of programming and software reliability. Automatic memory management eliminates all memory-management errors (e.g., "memory leaks"), which are a major source of problems in C and C++ programs. Various tools have been developed to help programmers find memory management errors. For example, *Purify* is a widely used tool that dynamically catches memory management errors as they occur.



END...



# Syntax Analysis (Parsing)

# Outline:



- ✓ The Role of the parser
- ✓ Error recovery Strategies or methods
- ✓ Context free Grammars: Derivations, Parse Trees
- ✓ Writing Context Free Grammars(CFG)
- ✓ Computing FIRST and FOLLOW sets
- ✓ Top-Down Parsers( RDP, Predictive)
- ✓ Bottom-Up parser(Shift-Reduce, ~~SLR~~)  
SRP

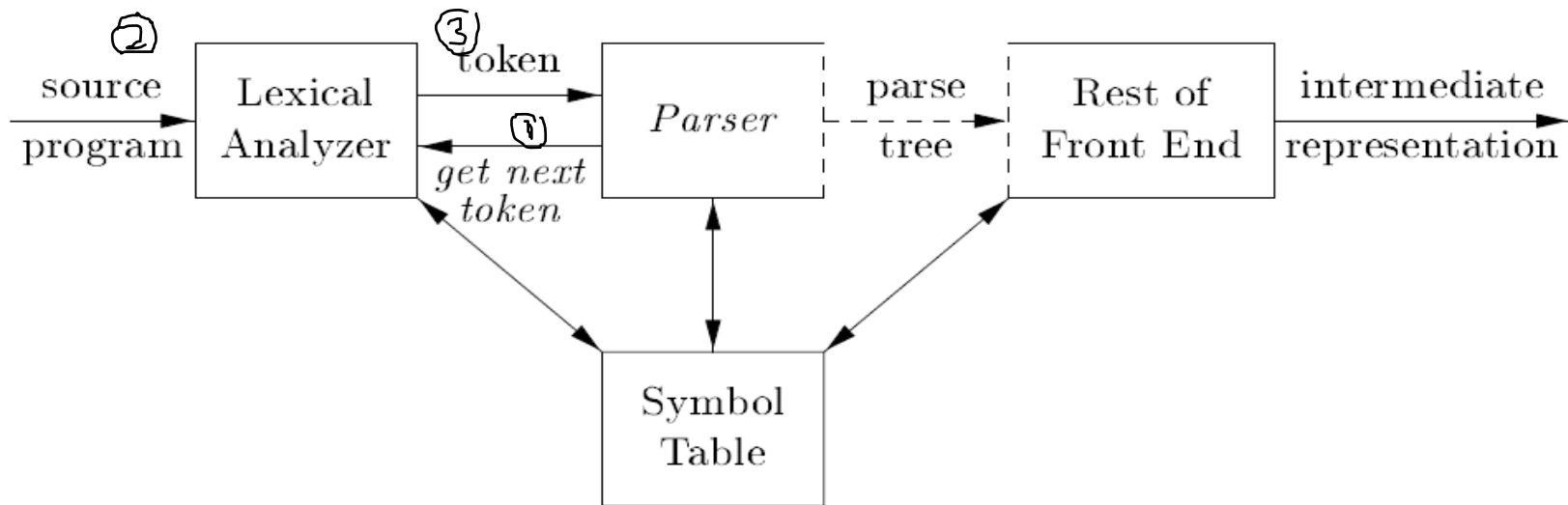
# Introduction: SYNTAX ANALYZER/PARSER

- ▶ *Syntax Analyzer* creates the syntactic structure of the given source program.
- ▶ This syntactic structure - *parse tree*.
- ▶ Syntax Analyzer is also known as *parser*.
- ▶ The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not.
  - ▶ If it satisfies, the parser creates the parse tree of that program.
  - ▶ Otherwise the parser gives the error messages.

**Parsing:** *The process of finding a parse tree for a given stream of tokens received ~~for~~<sup>by</sup> the Lexical Analyzer.*

- ▶ Every programming language has precise rules that prescribe the syntactic structure of well-formed programs.
  - ▶ Program is made up of functions, a function out of declarations and statements, a statement out of expressions
- ▶ The syntax of programming language constructs can be specified by context-free grammars (CFG)
- ▶ A context-free grammar
  - ▶ gives a precise syntactic specification of a programming language
  - ▶ the design of the grammar is an initial phase of the design of a compiler.
  - ▶ a grammar can be directly converted into a parser by some tools.

# The Role of The Parser



Position of parser in compiler model

Scanners are usually implemented to produce tokens only when requested by a parser.

Here is how it works

1. Get next token" is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next token.
3. It returns the token to Parser.

# Types of parsers:

There are three general types of parsers for grammars:

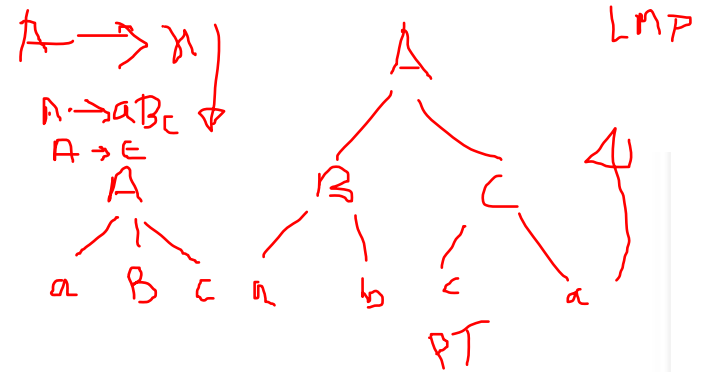
- (i) Universal parsing methods
- (ii) Top-down parsing methods and ✓
- (iii) Bottom-up parsing methods. ✓

The methods commonly used in compilers can be classified as being either top-down or bottom-up.

✗ **Universal parsing methods:** Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar (see the bibliographic notes). These general methods are, however, too inefficient to use in production compilers.

**Top-down parsing methods:** Top-down methods build parse trees from the top (root) to the bottom (leaves).

**Bottom-up parsing methods:** Bottom-up methods start from the leaves and work their way up to the root.



# Common Programming Errors:

We know that program may contain errors at many different levels. So error can be-

- (i) Lexical errors ✓
- (ii) Syntactic errors ✓
- (iii) Semantic errors ✓
- (iv) Logical errors ✓

*abmn* (circled with 'x' marks above it)  
*if = fi* (with 'x' marks above it)  
*hit* (with 'x' marks above it)

**Lexical errors:** Lexical errors include misspellings of identifiers, keywords, or operators e.g., the use of an identifier ellipse Size instead of ellipse Size—and missing quotes around text intended as a string.

**Syntactic errors:** It may occur due to grammatical error in statements of a program. For example an arithmetic expression with unbalanced parenthesis as  $((a+a)*c-d)$ . Another example is missing semicolon.

*(a + b*

*a = b + c,*



**Semantic errors:** Semantic errors include type mismatches between operators and operands. An example is a return statement in a Java method with result type void. For example, in c statement int c = a+2.5 ;

void f()

return;

**Logical errors:** it may occur due to an infinitely ~~recursive~~ call. For example in c statement-

LOOP

```
1) for (i=0; i<i+1; i++)
    {
        stmt
    }
```

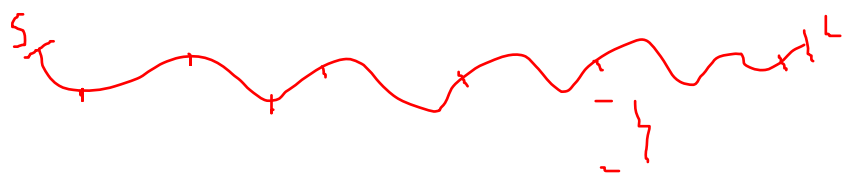
10 5  
 $a = b + c ;$   
 or  
 $a = b - c ;$

==

? 15  
 (5)<sup>x</sup>

2) if ( a=b ) in c.

## Goals of Error handler:



The error handler in a parser has goals that are simple to state but challenging to achieve.

- Report the presence of errors clearly and accurately in the source program.
- ✓ Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correct programs.

# Error-Recovery Strategies



Once an error is detected, how should the parser recover? Although no strategy has proven itself universally acceptable, a few methods have broad applicability.


There are many different strategies that a parser can employ to recover from a syntactic error. These are:

1. Panic-Mode recovery ✓
2. Phrase-Level recovery ✓
3. Error-Productions and ✓
4. Global Correction. ✗




**Panic-Mode:** This is the simplest method to implement and used by most parsing methods. In this method, on discovering an error, the parser discards input tokens one at a time until one of a designated set of <sup>skip</sup> synchronizing tokens is found. The synchronizing tokens are usually delimiters such as semicolon or }. The compiler designer must select the synchronizing tokens appropriate for the source language.


**Phrase-Level:** On discovering an error, a parser may perform local correction on the remaining input. A typical local correction is to replace a comma by a semicolon, delete an extraneous semicolon, or insert a missing semicolon. The choice of the local correction is left to the compiler designer.



**Error Productions:** Using an error productions, we can generate appropriate error diagnostic to indicate the erroneous construct that has been recognized in the input. We can augment the grammar for the language with productions that generate the erroneous constructs.



**Global Correction:** There are algorithms for choosing minimal sequence of changes to obtain a globally least-cost correction. Unfortunately, these algorithms are in general too costly to implement in terms of time and space, so these techniques are currently only of theoretical interest.



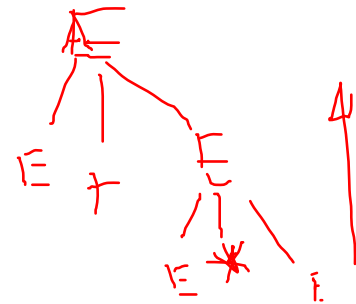
# Context-Free Grammar

$\left\{ \begin{array}{l} S \rightarrow \dots \\ A \rightarrow a \end{array} \right.$

Grammars very powerful notation to systematically describe the syntax of programming language constructs like expressions and statements. The syntactic structure of a language is defined using grammars.

Ex.: grammar that specifies the structure of the if-else statement:

$\left\{ \begin{array}{l} \text{stmt} \rightarrow \text{if} (\text{expr}) \text{stmt} \text{ else } \text{stmt} \\ \text{expr} \rightarrow b \end{array} \right.$



Ex.: grammar to describe arithmetic expression :

$\left\{ \begin{array}{l} E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid \text{id} \end{array} \right. \leftarrow \text{( Ambiguous)}$

$\text{id} + \text{id} * \text{id}$

Ex.: grammar to describe arithmetic expression :

$\left\{ \begin{array}{l} E \rightarrow E + T \mid E - T \mid T \\ T \rightarrow T * F \mid T / F \mid F \\ F \rightarrow -E \mid (E) \mid \text{id} \end{array} \right.$



$\leftarrow \text{( Unambiguous)}$

**Definition:** A CFG  $G$ , is a 4-tuple  $(V, T, P, S)$ ,  
 where:

$L = \{ \dots \}$   
 $L = \{ \dots \}$   
 $L = \{ \dots \}$

- V: a finite set of variables( Non-terminals)
- T: a finite set of terminal symbols or simply terminals / tokens
- P: a finite set of productions(Rules) of the  $A \rightarrow x$ , /  $A \rightarrow \epsilon$   
 where A is the Variable and x is any string of zero or more grammars symbols( i.e terminals and Non-terminals)
- S: the start variable.

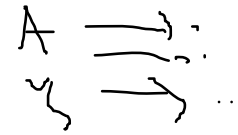
G:

$S \rightarrow aS \mid A$       S-productions  
 $A \rightarrow aA \mid bA \mid a$       A-productions

Where:

$V = \{ S, A \}$ ,  $T = \{ a, b \}$ ,  $P = \{ S \rightarrow aS \mid A, A \rightarrow aA \mid bA \mid a \}$ ,  
 S = start variable

# Notational Conventions:



terminal

## 1. These symbols are terminals:

1. Lowercase letters early in the alphabet, such as a, b, c.
2. Operator symbols such as +, \* , and so on.
3. Punctuation symbols such as parentheses, comma, and so on.
4. The digits 0,1,...9.
5. Boldface strings such as id or if, each of which represents a single terminal symbol.

## 2. These symbols are Non-terminals:

1. Uppercase letters early in the alphabet, such as A, B, C, ... ✓
2. The letter S, which, when it appears, is usually the start symbol.
3. Lowercase, italic names such as expr or stmt.

## 3. Uppercase letters late in the alphabet, such as X, Y, Z, represent grammar symbols; that is, either Non-terminals or terminals.

$$X = t, H$$
$$X = A, \text{,}$$

4. Lowercase letters late in the alphabet, chiefly  $u, v, \dots, z$ , represent (possibly empty) strings of terminals.  $\alpha = \underline{abbb}, \gamma = \underline{a+id\psi id}$
5. Lowercase Greek letters,  $\alpha, \beta, \gamma$  for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as  $A \rightarrow \alpha$ , where  $A$  is the head and  $\alpha$  the body.  $\alpha = \underline{"aAbbB"}$   $\alpha = \epsilon$
6. A set of productions  $A \rightarrow \alpha_1, A \rightarrow \alpha_2, \dots, A \rightarrow \alpha_k$  with a common head  $A$  (call them  $A$ -productions), may be written  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$ . Call  $\alpha_1, \alpha_2, \dots, \alpha_k$  the alternatives for  $A$ .

## In Short...

- **Terminals:** lower case letters, digits, punctuation
- **Nonterminals:** Upper case letters
- **Arbitrary Terminals/Nonterminals:**  $X, Y, Z$
- **Strings of Terminals:**  $u, v, w$
- **Strings of Terminals/Nonterminals:**  $\alpha, \beta, \gamma$
- **Start Symbol:**  $S$



# Derivation and Parse Tree:

i d + i d \* i d

The process of *deriving* a string is called as *derivation*.

- Starting with start symbol,

At each step in a derivation, there are two choices to be made. We need to choose which non-terminal to replace, and having made this choice, we must pick a production with that non-terminal as head

id + ✓ ∈ L

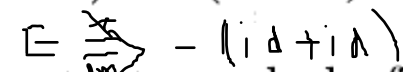
Consider the Grammar for generating Arithmetic Expressions:

$$E \rightarrow E + E \mid E * E \mid - E \mid ( E ) \mid id \quad \mathcal{L}(E) = \{id, id + id, id - id, \dots, id + (id * id) - id, \dots\}$$

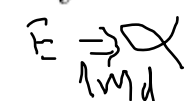
Suppose, we want to deriving string:  $-(id + id)$



$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(id + E) \Rightarrow -(id + id) \rightarrow \text{Sentence}$$



If  $S \xRightarrow{*} \alpha$ , where  $S$  is the start symbol of a grammar  $G$ , we say that  $\alpha$  is a *sentential form* of  $G$ .



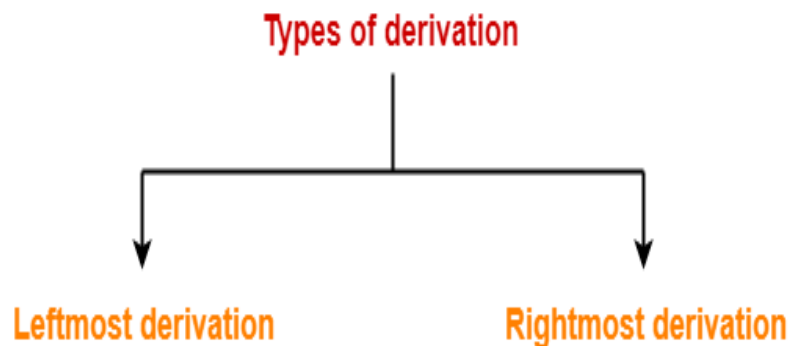
# Derivations Contd...



$\Rightarrow^*$  Means derive string in zero or more steps

$\Rightarrow \dots \Rightarrow$

$\Rightarrow^+$  Means derive string in one or more steps



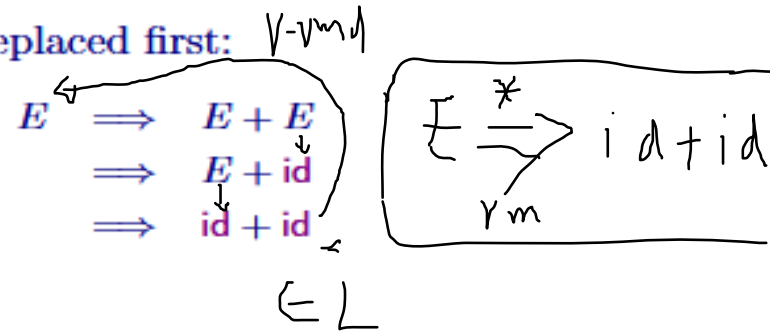
**Leftmost derivation(LMD):** The process of deriving a string by expanding the leftmost non-terminal at each step of derivation.

$$\underset{lm}{\alpha} \Rightarrow \beta$$

**Rightmost derivation(RMD):** The process of deriving a string by expanding the rightmost non-terminal at each step of derivation.

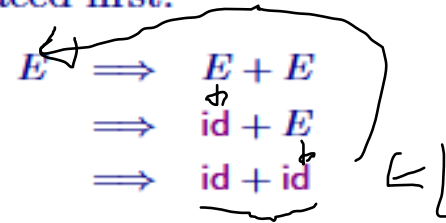
$$\underset{rm}{\alpha} \Rightarrow \beta$$

- **Rightmost derivation:** Rightmost nonterminal is replaced first:



Written as  $E \xrightarrow{*} id + id$

- **Leftmost derivation:** Leftmost nonterminal is replaced first:



Written as  $E \xrightarrow{*} id + id$

- ▶ We will see that the top-down parsers try to find the left-most derivation of the given source program.
- ▶ We will see that the bottom-up parsers try to find the right-most derivation of the given source program in the reverse order.

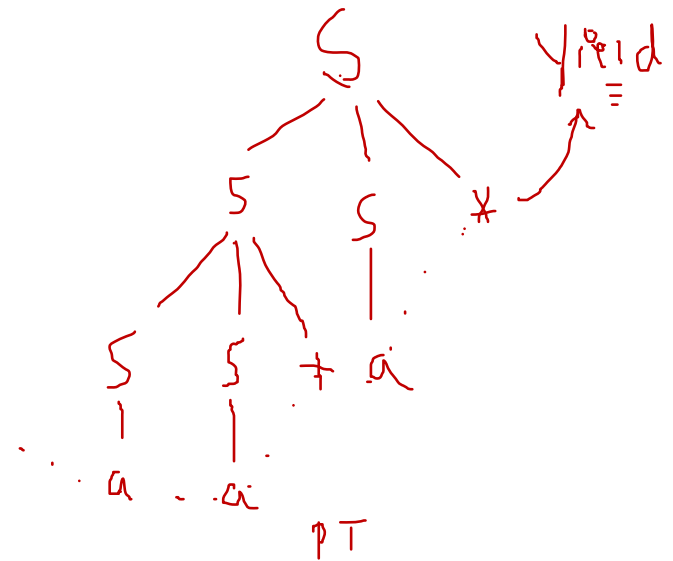
# Example:

For the context-free grammar:

$$S \rightarrow SS + \mid SS * \mid a$$

and the string  $aa + a*$ .

- Give a leftmost derivation for the string.
- Give a rightmost derivation for the string.
- Give a parse tree for the string.



a)  $S \Rightarrow S S * \Rightarrow \underline{S S + S *} \Rightarrow a S + S * \Rightarrow a a + S * \Rightarrow a a + a *$

$S \xrightarrow{*} a a + a *$   
 $\uparrow \text{rmd}$

$\Rightarrow a a + a *$

b)  $S \Rightarrow S S * \Rightarrow S a * \Rightarrow S S + a * \Rightarrow S a + a * \Rightarrow a a + a *$

$S \xrightarrow{*} a a + a *$   
 $\uparrow \text{rmd}$

# Practice Problems:

Pf ("a/b", a, b)!

For each of the following context-free grammar, Obtain LMD, RMD and Parse tree for the given string of Tokens(symbols)

Sum(a, b):

a)  $S \rightarrow 0 S 1 \mid 0 1$  with string 000111.

b)  $S \rightarrow + S S \mid * S S \mid a$  with string  $+ * \underline{aaa}$ .

! c)  $S \rightarrow S ( S ) S \mid \epsilon$  with string  $(( ))$ .

# Parse Tree:



- ▶ A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals.
- ▶ The interior node is labeled with the nonterminal  $A$  in the head of the production;
- ▶ The children of the node are labeled, from left to right, by the symbols in the body of the production
- ▶ The leaves of a parse tree are labeled by nonterminals or terminals / tokens
- ▶ Read from left to right, constitute a sentential form, called the yield or frontier of the tree.
- ▶ There is a many-to-one relationship between derivations and parse trees.

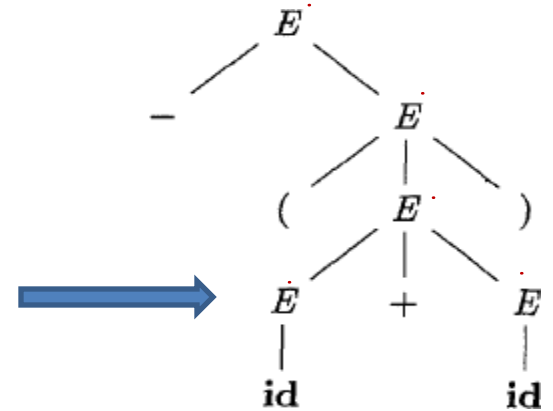
Alshaya, Govt. Engg. College, Thiruvandiyur

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

String:  $-(id + id)$

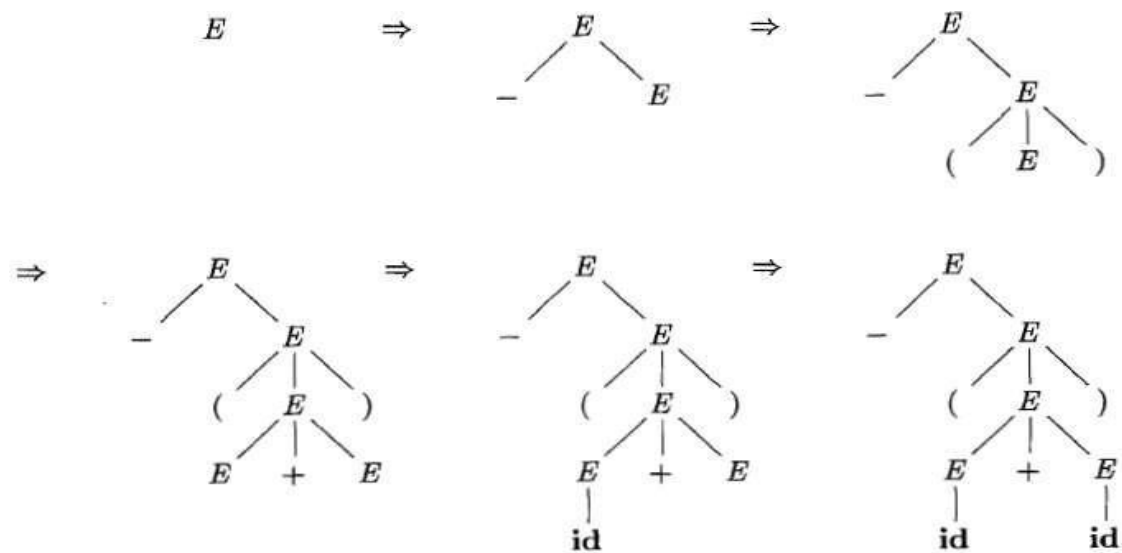
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + id) \Rightarrow -(id + id)$$

EMD



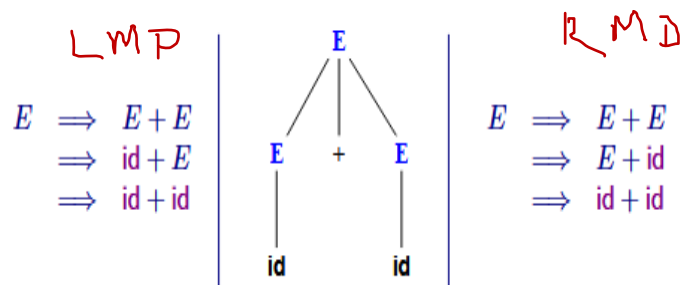
$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \text{id}) \Rightarrow -(\text{id} + \text{id})$$

Parse tree for  $-(\text{id} + \text{id})$



Sequence of parse trees for derivation

Graphical Representation of Derivations



# Writing a Grammar

$\left\{ \begin{array}{l} \leftarrow \\ \text{omit} \\ \text{unless} \end{array} \right\}$

Grammars are capable of describing most, but not all, of the syntax of programming languages.

-For instance, the requirement that variable be declared before they are used, cannot be described by a context-free grammar.

In this section, We consider some transformation techniques to get a grammar more suitable for parsing.

$G_2 \Rightarrow G_1$

$\rightarrow$   $\downarrow$   $\checkmark$   $-$   $=$   $-$

## Transformation Techniques:

1. Eliminating Ambiguity  $\checkmark$  (top, bottom)
2. Left Factoring of Grammar  $\checkmark$  (top)
3. Elimination of left recursion  $\checkmark$  (top)

$$S \rightarrow S S \overset{A}{\cancel{+ | x}} | a$$

$$A \rightarrow + | x$$

$$S S A = S S (+ | x)$$

$$= S S + | S S x$$

$$\bar{x} Y + \bar{x} Z$$

$$= \bar{x} (Y + Z)$$

$$S \rightarrow \underline{S S +} | \underline{S S x} | a$$

$$S \rightarrow \underline{S S} \bar{A} | a$$

$$S \Rightarrow A \rightarrow + | x$$

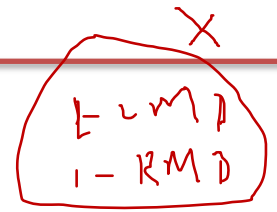


# Eliminating Ambiguity:

## Ambiguous Grammar:

$S \Rightarrow$

2 - LMD  
OR  
2 - RMD

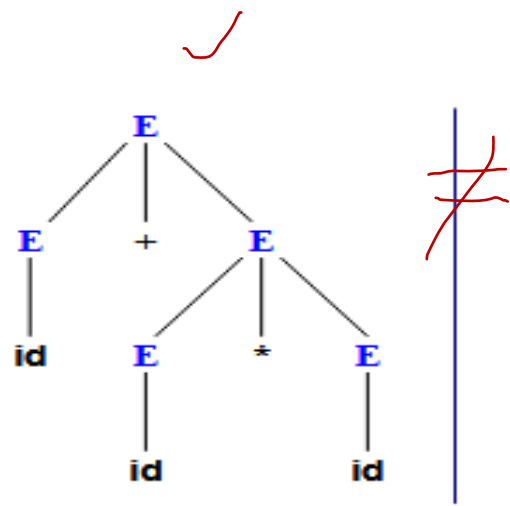


- A grammar that produce more than one parse tree for some sentence is said to be an ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.
- For most parsers, it is desirable that the grammar be made unambiguous.
- Fortunately, we can eliminate ambiguity from the grammars written for most of the program constructs.

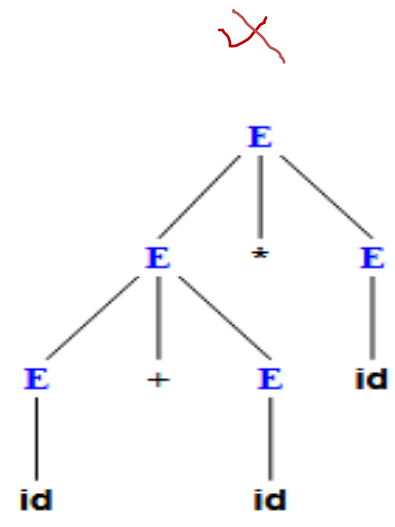
Ex.: The arithmetic expression grammar below permits two distinct leftmost derivations for the sentence id + id \* id.

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

	①		②
$E$	$\Rightarrow E + E$	$E$	$\Rightarrow E * E$
	$\Rightarrow id + E$		$\Rightarrow E + E * E$
	$\Rightarrow id + E * E$		$\Rightarrow id + E * E$
	$\Rightarrow id + id * E$		$\Rightarrow id + id * E$
	$\Rightarrow id + id * id$		$\Rightarrow id + id * id$



Parse tree-1  
(Correct)



Parse tree-2  
(Incorrect)

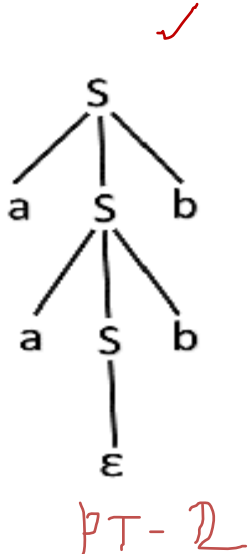
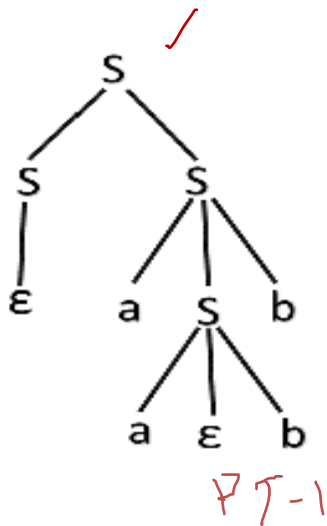
Two LMDs for same sentence

Two parse trees for **id+id\*id**

Another Example: Check whether the given grammar G is ambiguous or not.

$$S \rightarrow aSb \mid SS$$
$$S \rightarrow \epsilon$$

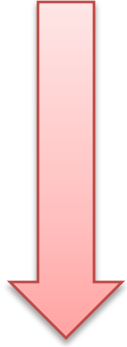
For the string "aabb" the above grammar can generate two parse trees:



Since there are two parse trees for a single string "aabb", the grammar G is ambiguous.

# 1. Eliminating Ambiguity from Grammar:

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



$+ , - \quad T$   
 $* , / \quad F$   
 $- , ( , ) , id$

Elimination of ambiguity by considering an operator precedence and associativity rules.

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

← Resulting Grammar

$id + id * id$



Where E, T, F are Non-Terminals, +, -, \*, / , ( , ) and id are Terminals.

Formally:

$G = ( \{E, T, F\}, \{+, -, *, /, (, ), id\}, P, E )$

# Another Example: if... else statement

- An ambiguous grammar can be rewritten to eliminate the ambiguity.

- Ex. Eliminating the ambiguity from the following dangling-else grammar:

$stmt \rightarrow$  **if** *expr* **then** *stmt*  
          | **if** *expr* **then** *stmt* **else** *stmt*  
          | **other**

- Compound conditional statement

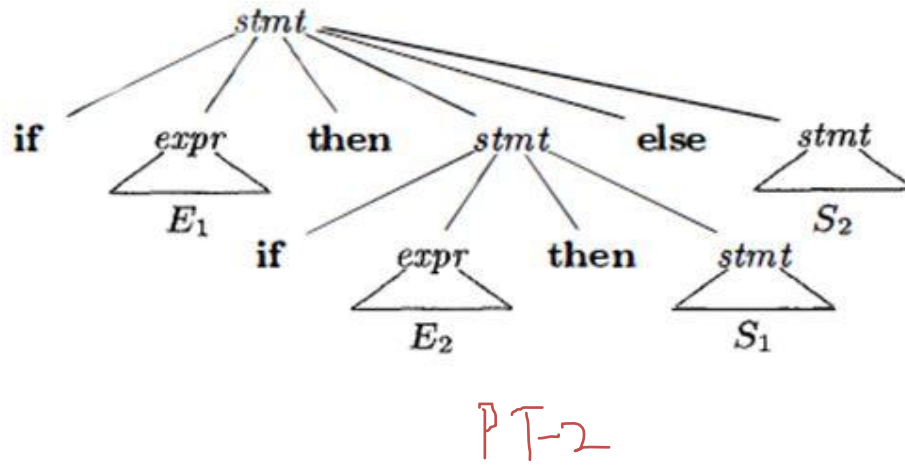
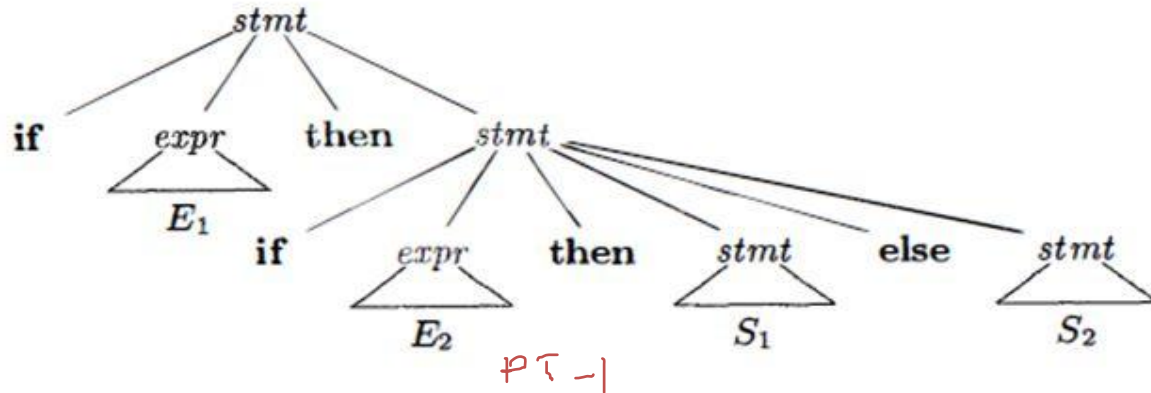
**if**  $E_1$  **then**  $S_1$  **else if**  $E_2$  **then**  $S_2$  **else**  $S_3$

PASCAL

```
if (E1)
{
  if (E2)
    S1;
  else
    S2;
}
```



# Two Parse Trees for the above string( refer Prev. slide)



We can rewrite the dangling-else grammar with the idea:

- A statement appearing between a **then** and an **else** must be *matched*

*stmt* → *matched\_stmt* ✓  
| *open\_stmt*  
*matched\_stmt* → **if** *expr* **then** *matched\_stmt* **else** *matched\_stmt*  
| **other**  
*open\_stmt* → **if** *expr* **then** *stmt*  
| **if** *expr* **then** *matched\_stmt* **else** *open\_stmt*

Unambiguous Grammar

## 2. Left factoring of Grammar:

$$\lambda y + \lambda z = \lambda (y+z)$$

$\lambda A$   
 $A = (y+z)$

Left factoring is a grammar transformation Technique that is useful for producing a grammar suitable for top-down parsing. (LMD)

(S, i)

Consider following grammar:

$$S \rightarrow iEtSeS \mid iEtSe \mid a$$

$$E \rightarrow b$$



$$\Rightarrow \left\{ \begin{array}{l} S \rightarrow iEtSeS \mid a \\ S' \rightarrow eS/E \\ E \rightarrow b \end{array} \right.$$

On seeing input i it is not clear for the parser which production to use.

In General, we can easily perform left factoring:

If we have  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \gamma$ , then we can replace it

with:

$$A \rightarrow \alpha A' \mid \gamma$$

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

$$\frac{A \rightarrow \alpha (\beta_1 \mid \beta_2) \mid \gamma}{A \rightarrow \alpha A' \mid \gamma}$$

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



## Left factoring of Grammar Contd...

**METHOD:** For each nonterminal  $A$ , find the longest prefix  $\alpha$  common to two or more of its alternatives. If  $\alpha \neq \epsilon$  — i.e., there is a nontrivial common prefix — replace all of the  $A$ -productions  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$ , where  $\gamma$  represents all alternatives that do not begin with  $\alpha$ , by

$$\left. \begin{array}{l} A \rightarrow \alpha A' \mid \gamma \\ A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{array} \right\}$$

### Example(1):

$$\left. \begin{array}{l} S \rightarrow i \underline{E} t S e S \mid i \underline{E} t S \mid a \\ E \rightarrow b \end{array} \right\} \quad \longrightarrow \quad \left. \begin{array}{l} S \rightarrow i E t S S' \mid a \\ S' \rightarrow e S \mid \epsilon \\ E \rightarrow b \end{array} \right\}$$

**Before Left-Factoring**

**After Left-Factoring**

## Another Example(2):

1. Eliminate left factor from the following grammar:

$$\begin{aligned} A &\rightarrow \underline{a}AB \mid aBc \mid \underline{a}Ac \\ B &\rightarrow b \end{aligned}$$

A — Prediction  
B — No

$$\begin{aligned} &\Downarrow \\ A &\rightarrow \underline{a}AA' \mid \underline{a}Bc \\ A' &\rightarrow B \mid c \\ B &\rightarrow b \end{aligned}$$

$\Rightarrow$

$$\begin{aligned} A &\rightarrow aA'' \\ A'' &\rightarrow AA' \mid Bc \\ A' &\rightarrow B \mid c \\ B &\rightarrow b \end{aligned}$$

[ Final ]

## Practice problem...

1. Perform the left factoring for the following grammar:

$$S \rightarrow bSSaaS \mid bSSaSb \mid bSb \mid a$$

2. Do left factoring in the following grammar:

$$S \rightarrow aAd \mid aB$$

$$A \rightarrow a \mid ab$$

$$B \rightarrow ccd \mid ddc$$



### 3. Elimination of left recursion:



- A grammar is **left recursive** if it has a non-terminal  $A$  such that there is a derivation  $A \xRightarrow{+} A\alpha$  (i.e.  $A$  derives  $A\alpha$  in one or more steps of derivations)
- Top down parsing methods can not handle left-recursive grammars.
- Left recursion is considered to be a problematic situation for all the Top down parsers.
- Therefore, left recursion has to be eliminated from the grammar.

#### Example:

$$S \rightarrow Sa \mid \epsilon$$

(Left Recursive Grammar)

$$S \Rightarrow - \Rightarrow \dots$$



$$S \xRightarrow{+} \underline{S\alpha}$$

➤ Left recursion can be of two types :

➤ 1) Direct or Immediate 2) Indirect or Intermediate

➤ A simple rule for Direct left recursion elimination:

For a rule like:

$$A \rightarrow Aa \mid \beta$$

L ✓

S ⇒ ... ⇒ ε

$$A \rightarrow \frac{AcBA}{\alpha} \mid \frac{cAd}{B}$$

We may replace it with (Equivalent):

$$\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow aA' \mid \varepsilon \end{array} \quad \Bigg|$$

**Note:**  $\alpha, \beta, \gamma$  etc. a Greek letters denotes string of terminals or Non-terminals or both.

# Elimination of Left-Recursion Contd...

$m+n$

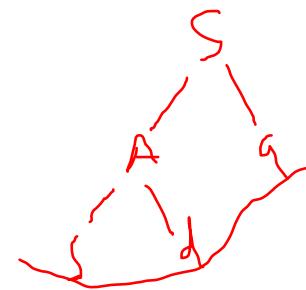
More Generally...

$$A \rightarrow \underline{A\alpha_1} \mid \underline{A\alpha_2} \mid \cdots \mid \underline{A\alpha_m} \mid \underline{\beta_1} \mid \underline{\beta_2} \mid \cdots \mid \underline{\beta_n}$$

where no  $\beta_i$  begins with an  $A$ . Then, replace the  $A$ -productions by

↓

$$\left\{ \begin{array}{l} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon \end{array} \right.$$



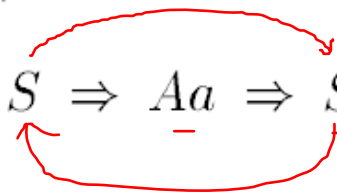
The nonterminal  $A$  generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the  $A$  and  $A'$  productions (provided no  $\alpha_i$  is  $\epsilon$ ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

**Indirect Left recursion:**

$$\begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow A c \mid S d \mid \epsilon \end{array}$$

**S is Indirect left-Recursive**

The nonterminal  $S$  is left recursive because  $S \Rightarrow Aa \Rightarrow Sda$ , but it is not immediately left recursive.



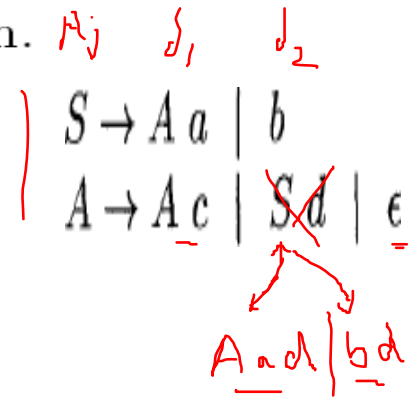
# Algorithm: To eliminate Left recursion from $G$ .

**INPUT:** Grammar  $G$  with no cycles or  $\epsilon$ -productions.

**OUTPUT:** An equivalent grammar with no left recursion.

## METHOD:

- 1) arrange the nonterminals in some order  $A_1, A_2, \dots, A_n$ .
- 2) for ( each  $i$  from 1 to  $n$  ) {
- 3)     for ( each  $j$  from 1 to  $i - 1$  ) {
- 4)         replace each production of the form  $A_i \rightarrow A_j \gamma$  by the productions  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ , where  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  are all current  $A_j$ -productions
- 5)     }
- 6)     eliminate the immediate left recursion among the  $A_i$ -productions
- 7) }



$$A \rightarrow S \delta$$

$\mathbb{D}$  is not

# Example-1: Eliminate Left recursion

$$S \rightarrow A a \mid b$$
$$A \rightarrow A c \mid \underline{S} d \mid \epsilon$$



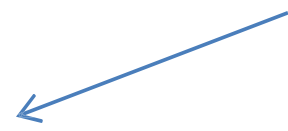
$$S \rightarrow A a \mid b$$
$$A \rightarrow A c \mid A a d \mid b d \mid \epsilon$$

*Handwritten annotations:  $\beta_1$  above  $A a$ ,  $\beta_2$  above  $A a d$ .  $A a$  and  $A a d$  are underlined.*



The grammar after eliminating left recursion

$$\left\{ \begin{array}{l} S \rightarrow A a \mid b \\ A \rightarrow b d A' \mid \epsilon \\ A' \rightarrow c A' \mid a d A' \mid \epsilon \end{array} \right.$$



~~Direct and LR-derivable~~



## Example-2: Eliminate Left recursion

$A_1$   $B_1$  ✓  
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid -E \mid id$



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

✓  
 $E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *TE' \mid \epsilon$   
 $F \rightarrow (E) \mid -E \mid id$

# Example-3: Eliminate Left recursion

$$\begin{aligned} S &\rightarrow (L) \mid a \quad \beta_1 \\ L &\rightarrow L, S \mid S \quad \alpha_1 \end{aligned}$$

↓

$$\begin{aligned} S &\rightarrow (L) \mid a \\ L &\rightarrow SL' \\ L' &\rightarrow \epsilon, SL' \mid \epsilon \end{aligned} //$$

$$\begin{aligned} A &\rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_m \\ &\parallel \\ &A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \\ &\parallel \\ A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \epsilon \end{aligned}$$

# Practice Problem...

$a|b|c$

1. Consider the following grammar and eliminate left recursion

$$\begin{aligned}
 S &\rightarrow aB \mid aC \mid Sd \mid Se \\
 B &\rightarrow bBc \mid f \\
 C &\rightarrow g
 \end{aligned}$$

$$\begin{aligned}
 S &\rightarrow SA \mid Se \mid aB \mid aC \\
 S &\rightarrow aB S' \mid aC S' \\
 S' &\rightarrow d S' \mid e S' \mid \epsilon \\
 B &\rightarrow \\
 C &\rightarrow
 \end{aligned}$$

2) Consider the following grammar and eliminate left recursion

$$\begin{aligned}
 S &\rightarrow Sb \mid Ba \mid a \\
 B &\rightarrow BSb \mid Sa \mid b
 \end{aligned}$$

$$\begin{aligned}
 S &\rightarrow BaS' \mid aS' \\
 S' &\rightarrow bS' \mid \epsilon \\
 B &\rightarrow BSb \mid Sa \mid b
 \end{aligned}$$

$$\begin{aligned}
 S &\rightarrow BaS' \mid aS' \\
 S' &\rightarrow bS' \mid \epsilon \\
 B &\rightarrow BSb \mid BaS'b \mid aSa
 \end{aligned}$$

Watch this Video: [https://www.youtube.com/watch?v=3\\_VCoBfrrt9c](https://www.youtube.com/watch?v=3_VCoBfrrt9c)

$$\begin{aligned}
 S &\rightarrow \\
 S' &\rightarrow
 \end{aligned}$$

$$\begin{aligned}
 B &\rightarrow aSaB' \mid aB' \\
 B' &\rightarrow SbB' \mid aS'bB' \mid \epsilon
 \end{aligned}$$



# Topic: Syntax Analysis

(Top- down and Bottom-up parsing)

# Introduction (Top-down parsing)

- Top-down parsing can be viewed as finding a leftmost derivation for an input string.
- A Top-down parser tries to create a parse tree from the root towards the leafs scanning input from left to right.
- The sequence of parse trees in Fig. for the input **id+id\*id** is a top-down parse according to the grammar below.

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

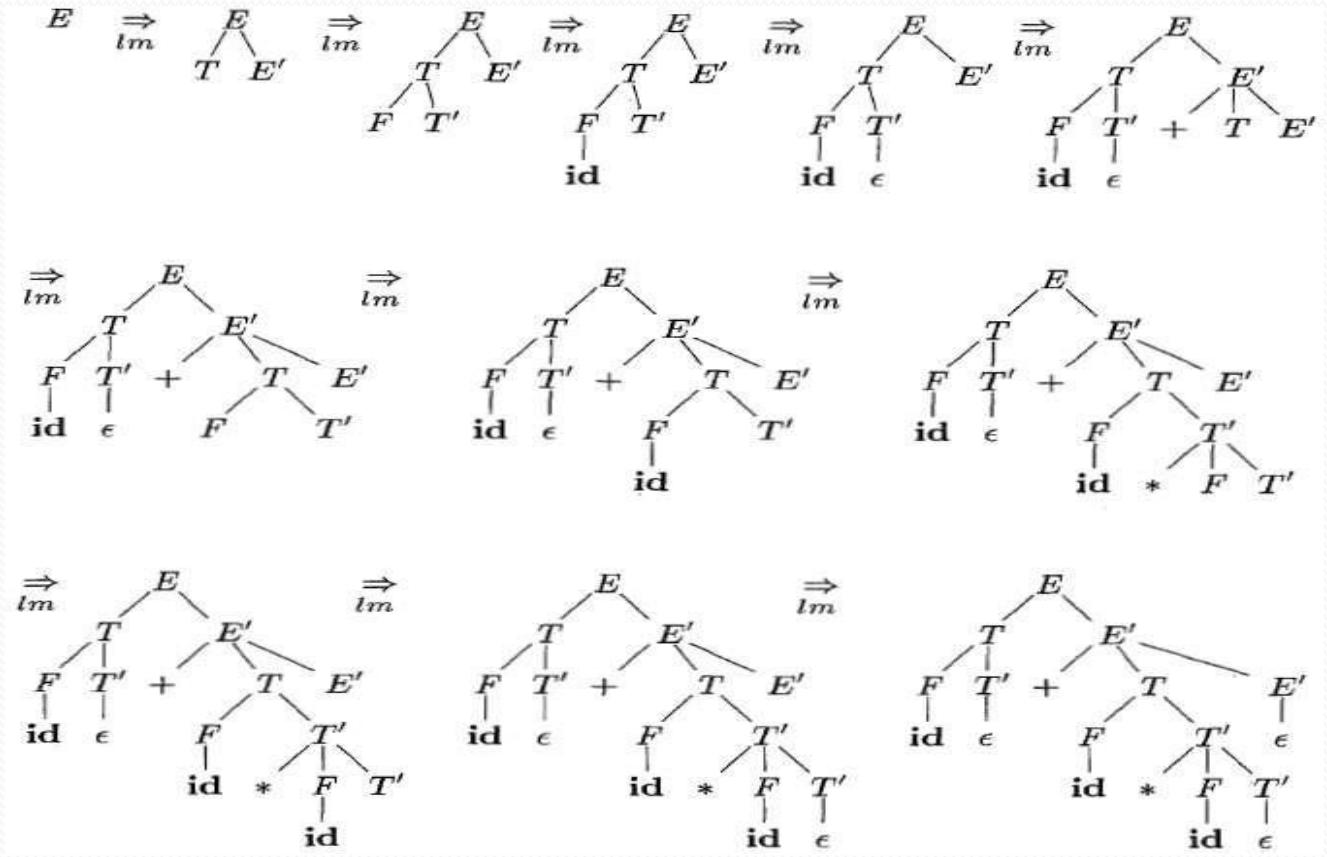
$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

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

# The top down Parse for : **id+id\*id**

- $E \rightarrow TE'$
- $E' \rightarrow +TE' \mid \epsilon$
- $T \rightarrow FT'$
- $T' \rightarrow *FT' \mid \epsilon$
- $F \rightarrow (E) \mid \mathbf{id}$



## Introduction Contd.

- Constructing a parse tree for an input string starting from the root.
- Finding Left-Most -Derivation(LMD)
- At each step of a derivation, the Key Problems are:
  1. Determine the correct production to be applied.
  2. Matching terminal symbols in the production's body with input string.



## Actions of Top-down Parser:

Action	Description
Expand	Expand leftmost Non-terminal in the Sentential form with correct A-production's body(i.e. LMD)
Match	Match the current input symbol with terminal symbol in the body of the production.
Accept	Announce successful completion of parsing on Input string.
Error	Discover a syntax error in the input.

- In this section, we study two top-down Parsing Techniques:
- 1. Recursive -Descent Parser(Procedure-driven)
  - General form of Top-down Parsing.
  - May require Backtracking to find correct production to be applied for expand action.  
(Drawback: repeated scan of the same input may required)
  - Try one production at a time for a Non-terminal.
- 2. Predictive-Parser/LL(1) Parser(Table -Driven)
  - A special case of Recursive- Descent Parsing Technique
  - Backtracking not required(Advantage)
  - Chooses correct A-production by looking ahead in the input a fixed number of symbols (Typically 1 symbol)

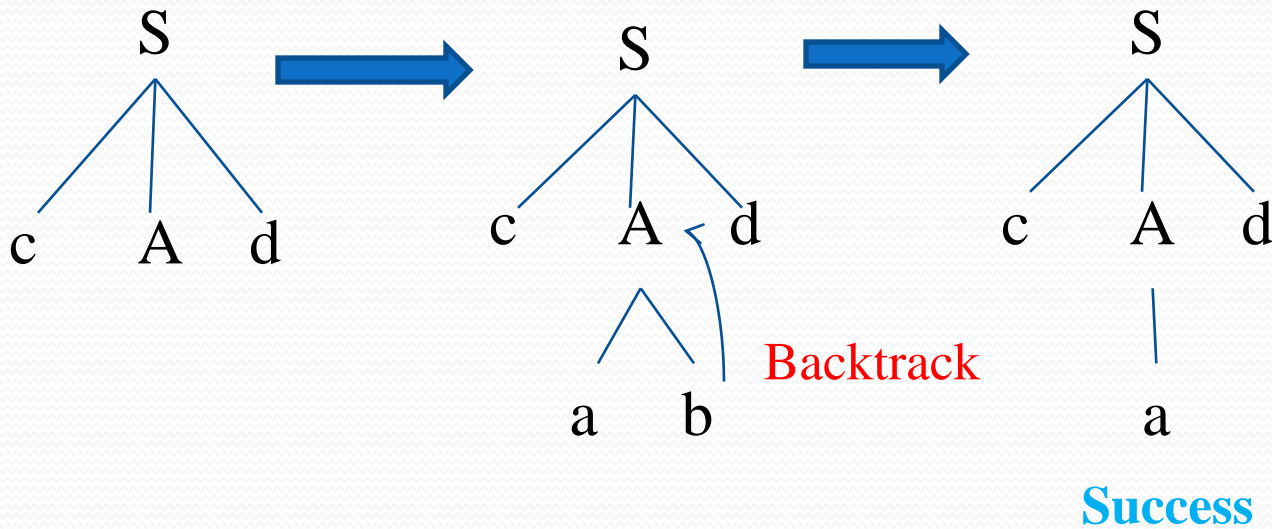
# Recursive-Descent Parsing:

## Example(Backtracking)

$S \rightarrow cAd$   
 $A \rightarrow ab \mid a$

Input: 

c	a	d	\$
---	---	---	----



**Note:** \$ is input right end marker

# Recursive-Descent Parser(RDP):

- Consists of a set of procedures(functions) one for each Non-terminal.
- Execution begins with the procedure for start symbol.
- A typical procedure for a Non-terminal  $A$  of  $G$ :

```
void A( )
{
    Choose an A-Production,  $A \rightarrow X_1X_2..X_k$ 
    for ( i =1 to k )
        {
            if ( $X_i$  is a Nonterminal )
                call procedure  $X_i( )$ ;
            else if ( $X_i =$  current input symbol a)
                advance the input to the next symbol;
            else error( ) ; /* an error has occurred */
        }
}
```

## Recursive-Descent parsing contd.

- In general RDP may require backtracking (that is, it may require repeated scans over the input)
- However, backtracking is rarely needed to parse programming language constructs (statements).
- The previous code needs to be modified to allow backtracking.
- In general, we can't choose an A-production easily.
- So, we must try each of several alternative productions in some order, until it finds correct one for expansion.
- In order to try another A-production, the input pointer needs to be reset to where it was before this production was tried.
- Recursive descent parsers can't be used for **left-recursive and left factor grammars**.

## Problem: Write a recursive-Descent Parser for the Grammar.

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

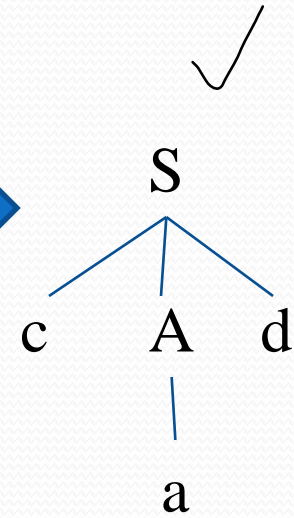
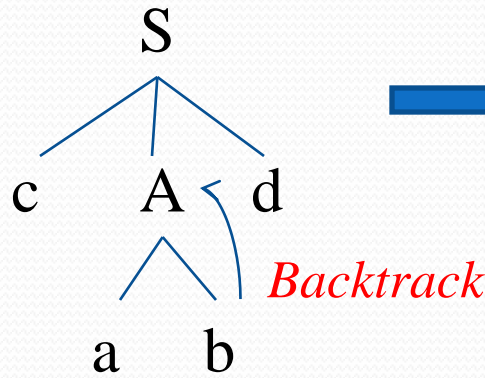
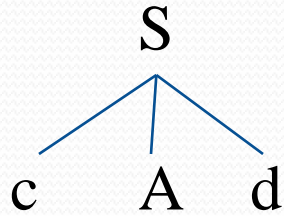
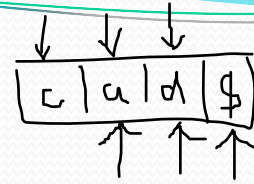
Procedure for S:  $S \rightarrow cAd$

```
void S( )
{
    if( input symbol == ' c ' )
    {
        advance the input to the next symbol;
        A( );
        if (input symbol == ' d ' )
            advance the input to the next symbol;
        else
            error( ); /* Parsing fails */
    }
    else
        error( ); /* parsing fails */
} /* end of S( ) */
```

## Procedure for A: $A \rightarrow ab \mid a$

```
void A()  
{  
  int prod = 1; /* try first production*/  
  while( 1 ) /* repeat until success or error */  
  {  
    switch(prod)  
    {  
  case 1: if(input symbol == 'a')  
          advance the input to the next symbol;  
          else { backtrack( ); prod = 2 ; break; } /* try second production*/  
          if(input symbol == 'b')  
            advance the input to the next symbol; /* success & return */  
          else { backtrack( ); prod = 2; break; } /* try second production*/  
          break;  
  case 2: if(input symbol == 'a')  
          advance the input to the next symbol; /* Success & return */  
          else error( ); /* both productions tried & Parsing fails */  
          break;  
    } /* end of switch */  
  }  
} /* end of A( ) */
```

$S \rightarrow cAd$   
 $A \rightarrow ab \mid a$



*Parsing success*

*Fig.:* Trace on the input : **cad\$**

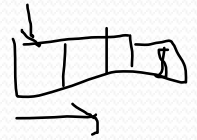
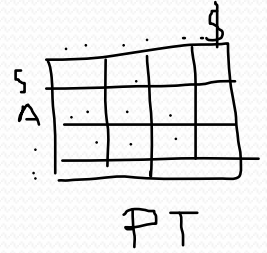


# Predictive parsing

LL(1)

LL(K) - PAR.

- Predictive parsing is Table-Driven method.
- It is also called **LL(1)** parsing method.  
(**L**: Scan input string from Left - right,  
**L**: using LMD, looking 1 symbol ahead at input)
- A special case of Recursive-Descent Parsing.
- Backtracking is not required.  
(i.e. no repeated scan of input - **Advantage**)
- Chooses correct A-production by **looking ahead** at the input a **fixed number** of symbols, typically we may look at 1 (that is, the next input symbol)



## Steps in Design of Predictive/LL(1) Parser...

- Step-1: Transform the Grammar  $G$ , to make it suitable for LL(1) parsing (left recursion, left factor etc.)
- Step-2: Compute FIRST and FOLLOW sets for each Nonterminals (Variables) of  $G$ .
- Step-3: Construct Predictive parsing/LL(1) Parsing table.
- Step-4: Apply parsing Algorithm to parse the given input string and **produce parse tree**.

# Computing FIRST and FOLLOW sets

Definitions:  $S \Rightarrow \dots \Rightarrow \dots$

- $FIRST(\alpha)$  is set of terminals that begins strings derived from  $\alpha$  (where  $\alpha$  is a string of grammar symbols)
- If  $\alpha \Rightarrow \epsilon$  then add  $\epsilon$  to  $FIRST(\alpha)$  also.
- $FOLLOW(B)$ , for any Nonterminal  $B$ , is set of terminals a that can appear immediately after B in some sentential form.
  - If we have,  $S \Rightarrow \alpha \underline{B} \alpha \beta$  for some  $\alpha$  and  $\beta$  then a is in  $FOLLOW(B)$ 

$\Rightarrow \alpha \underline{B} \alpha \beta$

$\alpha \Rightarrow \dots \Rightarrow \dots \Rightarrow \epsilon$

# Computing FIRST(): Procedure

$FIRST(A) = FIRST(B) =$   
 $A \rightarrow B C a$   
 $X \rightarrow Y_1 Y_2 Y_3$

- To compute  $FIRST(A)$  for all Non-terminal  $A$  of  $G$ , apply following rules until no more terminals or  $\epsilon$  can be added to any First set:

1. ✓ If  $X$  is a terminal then  $FIRST(X) = \{ X \}$ .

$FIRST(a) = \{ a \}$

2. If  $X$  is a Non-terminal and  $X \rightarrow Y_1 Y_2 \dots Y_k$  is a production for some  $k \geq 1$ , then place  $a$  in  $FIRST(X)$  if for some  $i$ ,  $a$  is in  $FIRST(Y_i)$  and  $\epsilon$  is in all of  $FIRST(Y_1), \dots, FIRST(Y_{i-1})$  that is  $Y_1 \dots Y_{i-1} \Rightarrow \epsilon$ .

✓ Also, if  $\epsilon$  is in  $FIRST(Y_j)$  for  $j=1, \dots, k$  then add  $\epsilon$  to  $FIRST(X)$ .

3. If  $X \rightarrow \epsilon$  is a production then add  $\epsilon$  to  $FIRST(X)$

# Computing FOLLOW(): Procedure

- To compute FOLLOW(B) for all Non-terminals B of  $G$ , apply following rules until nothing can be added to any follow set:
  - ✓ 1. Place \$ in FOLLOW(S), where S is the start symbol.
  2. If there is a production  $A \rightarrow \alpha B \beta$  then everything in FIRST( $\beta$ ) except  $\underline{\varepsilon}$  is in FOLLOW(B).  
$$E \cdot \varepsilon \cdot AB \cdot L \cdot L = \underline{\varepsilon} B$$
  3. If there is a production  $A \rightarrow \alpha \underline{B} \varepsilon$  or a production  $A \rightarrow \alpha B \beta$ , where FIRST( $\beta$ ) contains  $\varepsilon$ , then everything in FOLLOW(A) is in FOLLOW(B)

# Example-1

Compute FIRST and FOLLOW set for the Grammar below:

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

$\text{FIRST}(E) = \{ (, \text{id} \}$   
 $\text{FIRST}(E') = \{ +, \epsilon \}$   
 $\text{FIRST}(T) = \{ (, \text{id} \}$   
 $\text{FIRST}(T') = \{ *, \epsilon \}$   
 $\text{FIRST}(F) = \{ (, \text{id} \}$   
 $\text{FOLLOW}(E) = \{ \$, ) \}$   
 $\text{FOLLOW}(E') = \{ \$, ) \}$   
 $\text{FOLLOW}(T) = \{ +, \$, ) \}$   
 $\text{FOLLOW}(T') = \{ +, \$, ) \}$   
 $\text{FOLLOW}(F) = \{ *, +, ), \$ \}$

$\text{FIRST}(E) = \{ (, \text{id} \}$   
 $\text{FIRST}(E') = \{ +, \epsilon \}$   
 $\text{FIRST}(T) = \{ (, \text{id} \}$   
 $\text{FIRST}(T') = \{ *, \epsilon \}$   
 $\text{FIRST}(F) = \{ (, \text{id} \}$

$\text{FOLLOW}(E) = \{ ), \$ \}$   
 $\text{FOLLOW}(E') = \{ ), \$ \}$   
 $\text{FOLLOW}(T) = \{ +, ), \$ \}$   
 $\text{FOLLOW}(T') = \{ +, ), \$ \}$   
 $\text{FOLLOW}(F) = \{ *, +, ), \$ \}$

## Example-2

Reference.:

$S \rightarrow (L) \mid a$

$L \rightarrow L,S \mid S$

Compute FIRST and FOLLOW sets for the following grammar :

$S \rightarrow (L) \mid a$

$L \rightarrow SL'$

$L' \rightarrow ,SL' \mid \epsilon$

## Practice problem:

Compute FIRST and FOLLOW sets for the following grammar.

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$E \rightarrow b$$

Additional Examples....u may

[Watch this Video..](#)



# Design of predictive/LL(1) parsing table

NonTerminals	Input symbols/terminals			
	a	b	...	\$
S	$S \rightarrow \alpha$			
A		$A \rightarrow Xa$		
.			...	
.				
.				

$M[S,a]$  →

M

Structure of Predictive Parsing Table

# Construction of predictive parsing table

- For each production  $A \rightarrow \alpha$  in grammar do the following:
  1. For each terminal  $a$  in  $\text{FIRST}(\alpha)$  add  $A \rightarrow \alpha$  in  $M[A,a]$
  2. If  $\varepsilon$  is in  $\text{FIRST}(\alpha)$ , then for each terminal  $b$  in  $\text{FOLLOW}(A)$  add  $A \rightarrow \varepsilon$  to  $M[A,b]$ .
  3. If  $\varepsilon$  is in  $\text{FIRST}(\alpha)$  and  $\$$  is in  $\text{FOLLOW}(A)$ , add  $A \rightarrow \varepsilon$  to  $M[A,\$]$  as well.



Note: 1. After applying the above algorithm for each production, unfilled cells in the Parsing Table implies error case.

2. If there are any conflicting entries in Parsing Table then grammar is said to be not LL(1).

$$1) \text{FIRST}(TE') = \{ \checkmark, id \}, \text{FIRST}(+TE') = \{ + \}, \text{FIRST}(E) = \{ E \}$$

## Example: Constructing Paring table. $\text{FIRST}((E)) = \{ ( \}$

$$A \rightarrow \alpha$$

$$E \rightarrow TE' \quad \alpha$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

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

$$\text{FIRST}(E) = \{ (, id \}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T) = \{ (, id \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FIRST}(F) = \{ (, id \}$$

$$\text{FOLLOW}(E) = \{ ), \$ \}$$

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

$$\text{FOLLOW}(T) = \{ +, ), \$ \}$$

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

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

$$\text{FIRST}(id) = \{ id \}$$

Non-terminal \	Input Symbols					
	id	+	*	(	)	\$
E	E → TE' ✓			E → TE' ✓		
E'		E' → +TE' ✓			E' → ε ✓	E' → ε ✓
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id ✓			F → (E) ✓		

M

## Another example...

$S \rightarrow iEtSS' \mid a$   
 $S' \rightarrow eS \mid \epsilon$   
 $E \rightarrow b$

$FIRST(S) =$

$FIRST(S') =$

$FIRST(E) =$

$FOLLOW(S) =$

$FOLLOW(S') =$

$FOLLOW(E) =$

Conflicts, in any cell  $\Rightarrow$  implies that grammar is not LL(1)

Input Symbols

Non-terminal	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

M

Consider the following Grammar:

15M - 20M

$$S \rightarrow (L) \mid a$$

$$L \rightarrow L,S \mid S$$

- i) Make necessary changes to make it suitable for LL(1) parsing.
- ii) Compute FISRT and FOLLOW sets
- iii) Construct the Predictive/LL(1) parsing table.

**i) Eliminating Left-recursion from the grammar:**

$$S \rightarrow (L) \mid a$$

$$L \rightarrow SL'$$

$$L' \rightarrow ,SL' \mid \epsilon$$

## ii) Computing FIRST and FOLLOW sets

$$\text{FIRST}(S) = \{ (, a \}$$

$$\text{FIRST}(L) = \{ (, a \}$$

$$\text{FIRST}(L') = \{ , , \epsilon \}$$

$$\text{FOLLOW}(S) = \{ \$, \epsilon, ) \}$$

$$\text{FOLLOW}(L) = \{ ) \}$$

$$\text{FOLLOW}(L') = \{ ) \}$$

## iii) Design of Parsing Table

$$S \rightarrow (L) \mid a$$

$$L \rightarrow SL'$$

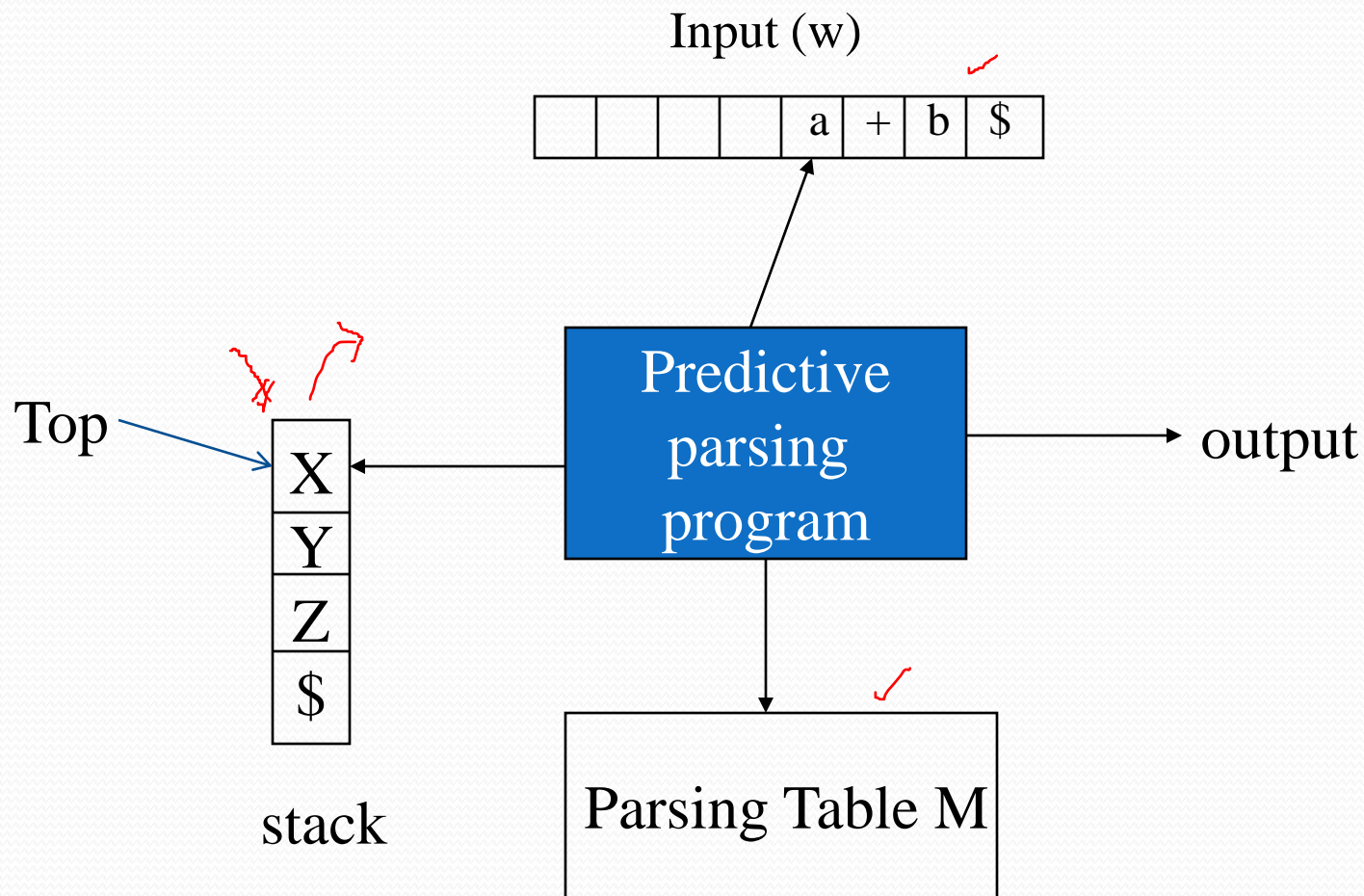
$$L' \rightarrow ,SL' \mid \epsilon$$

✓

NT	→ ,	(	)	a	\$
S		$S \rightarrow (L)$		$S \rightarrow a$	
L		$L \rightarrow SL'$		$L \rightarrow SL'$	
L'	$L' \rightarrow ,SL'$		$L' \rightarrow \epsilon$		

M

# predictive parser(Non-recursive): Model



# Predictive parser: Algorithm

Set  $ip$  point to the first symbol of  $w\$$ ;

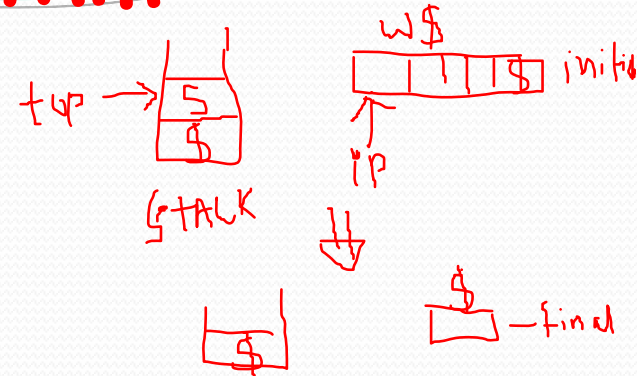
Set  $X \leftarrow \text{stacktop}()$ ;  $X = S$

While ( $X \neq \$$ ) /\* stack is not empty \*/

{  
if ( $X ==$  current input symbol )  
    pop( ) and advance  $ip$ ;  
else if ( $X$  is a terminal) error();  
else if ( $M[X,a]$  is blank entry) error();  
else if ( $M[X,a] = X \rightarrow \underline{Y_1 Y_2 \dots Y_k}$ )  
    {  
        pop( );  
        push  $Y_k, \dots, Y_2, Y_1$  on to the stack with  $Y_1$  on top;  
    }

$X \leftarrow \text{stacktop}()$ ;

}





$E \Rightarrow TE' \Rightarrow FT'E' \Rightarrow id+id \Rightarrow \dots \Rightarrow id+id * id$

Moves made by a predictive parser on input id + id \* id

$w = id + id * \$$

$x = E$   
 $T$

$M[E, id]$

←	STACK	INPUT	ACTION
	$E\$$	$id + id * id\$$	
	$TE'\$$	$id + id * id\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$id + id * id\$$	output $T \rightarrow FT'$
	$id T'E'\$$	$id + id * id\$$	output $F \rightarrow id$
	$T'E'\$$	$+ id * id\$$	match $id$
	$E'\$$	$+ id * id\$$	output $T' \rightarrow \epsilon$
	$+ TE'\$$	$+ id * id\$$	output $E' \rightarrow + TE'$
	$TE'\$$	$id * id\$$	match $+$
	$FT'E'\$$	$id * id\$$	output $T \rightarrow FT'$ ✓
	$id T'E'\$$	$id * id\$$	output $F \rightarrow id$
	$T'E'\$$	$* id\$$	match $id$
	$* FT'E'\$$	$* id\$$	output $T' \rightarrow * FT'$
	$FT'E'\$$	$id\$$	match $*$
	$id T'E'\$$	$id\$$	output $F \rightarrow id$
	$T'E'\$$	$\$$	match $id$
	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
	$\$$	$\$$	output $E' \rightarrow \epsilon$ ✓

→ success

( Refer Parsing-Table of expression Grammars)

**Show the moves made by a Predictive parser on input: id+id**

# PRACTICE PROBLEMS( predictive parsing)

1) Consider the Grammar

$S \rightarrow A$

$A \rightarrow aB \mid Ad$

$B \rightarrow b$

$C \rightarrow g$

|

- i) Make necessary changes to make it suitable for LL(1) parsing.
- ii) Compute FIRST and FOLLOW sets
- iii) Construct Predictive parsing table. Is Grammar LL(1) ? Justify
- iv) Show the moves made by the predictive parser on input: abdd

2) Construct predictive parsing table by making necessary changes to the grammar given below:

$S \rightarrow Ac$

$A \rightarrow Ac \mid \epsilon$

$B \rightarrow d \mid \epsilon$

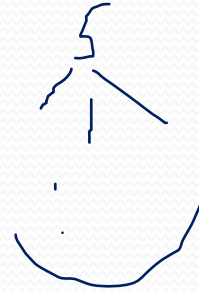
3) Construct predictive parsing table for the following grammar:

$S \rightarrow AA$

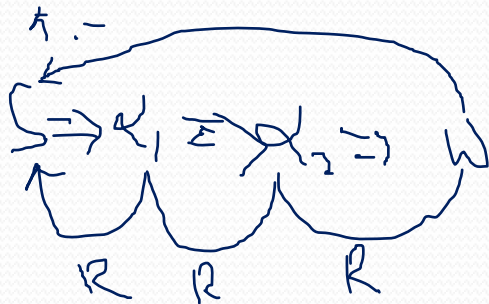
$T \rightarrow aA \mid b$

L M  $\bar{v}$

$S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \text{string}$



# Bottom-up Parsing



RMB (REPLY SET)



# Introduction...

- A bottom-up parse corresponds to the Construction of parse tree for an input string beginning at the leaves (the bottom) and working towards the root (the top)
- Uses reverse of right-most derivation
- This section, we study a general style of bottom-up parsing known as Shift-Reduce parsing (SRP)

RMD



• Example: token stream id \* id

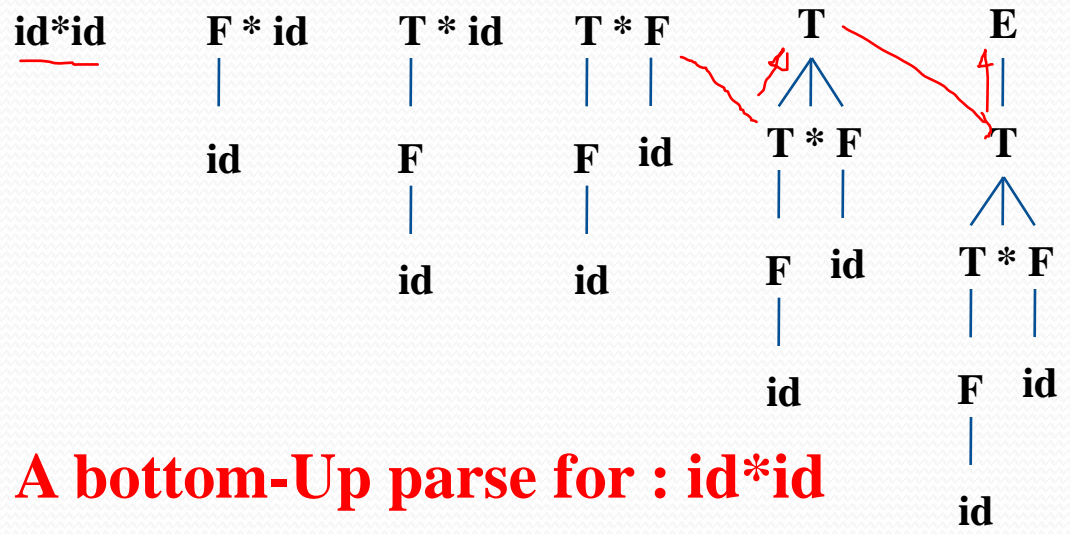
$E \rightarrow E + T \mid T$   
 $T \rightarrow T * F \mid F$   
 $F \rightarrow (E) \mid \text{id}$

RMD:

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \text{id} \Rightarrow F * \text{id} \Rightarrow \text{id} * \text{id}$

RSE

Munching



**A bottom-Up parse for : id\*id**

# Reductions:

$$A \rightarrow \alpha_1$$

 $\alpha_1$ 

-We can think of bottom-up parsing as the process of "reducing" a string w to the start symbol of the grammar.

-At each reduction step, a specific substring(handle) matching the body of a production is replaced by the Nonterminal at the head of that Production.

## Handle :

A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation(RMD).

## Handle pruning?

The process of discovering a **handle** & reducing it to the appropriate left-hand side of a production is called **handle pruning**. Handle pruning forms the basis for a **bottom-up parsing** method.

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

RMD:  $E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id}_2 \Rightarrow F * \mathbf{id}_2 \Rightarrow \mathbf{id}_1 * \mathbf{id}_2$

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	$F$	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$T \rightarrow T * F$
$T$	$T$	$E \rightarrow T$

Handles during a parse of  $\mathbf{id}_1 * \mathbf{id}_2$



$$A \rightarrow \alpha_1$$

$$B \rightarrow \alpha_2$$

$$R \rightarrow \alpha_1 \alpha_2 / \text{Red} \alpha_1 \alpha_2$$

$$\alpha_1^n \alpha_2^m : n \geq 1, m \geq 1$$

Given the grammar  $S \rightarrow OS1 \mid O1$

Show all the handles during a parse of the string: 000111

RMD:  $S \Rightarrow \underbrace{OS1}_3 \Rightarrow \underbrace{OOS11}_2 \Rightarrow \underbrace{OOO111}_1$

RST	Handle	Reducing Production
000111	01	$S \rightarrow O1$
00S11	OS1	$S \rightarrow OS1$
OS1	OS1	$S \rightarrow OS1$

↓  
S

Handles during a parse of 000111.

# Shift-Reduce parsing(SRP)

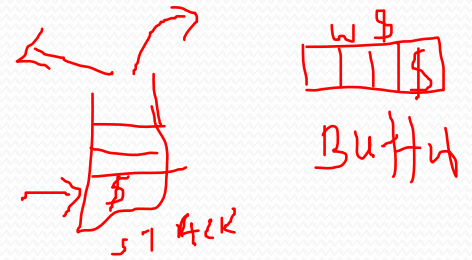
- Shift-Reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and input buffer holds string to be parsed.
- The general idea is to shift some symbols of input to the stack until a reduction can be applied.
- At each reduction step, a specific substring matching the body of a production is replaced by the Non-terminal at the head of the production.
- The key decisions during bottom-up parsing are about when to reduce and about what production to apply.
- The goal of a bottom-up parser is to construct a RMD in reverse.

# Actions of Shift Reduce Parser

- There are four possible actions of a shift-parser action:
  1. **Shift** : The next input symbol is shifted onto the top of the stack.
  2. **Reduce**: Replace the handle on the top of the stack by the non-terminal.
  3. **Accept**: Successful completion of parsing.
  4. **Error**: Parser discovers a syntax error, and calls an error recovery routine.

# Stack implementation of a Shift-reduce parser

- A stack is used to hold grammar symbols
- Handle always appear on top of the stack
- Initial configuration:



Stack      Input

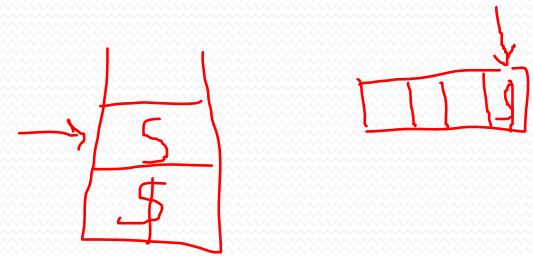
\$                      w\$

⋮                      ⋮

- Acceptance(final) configuration:

Stack                      Input

\$S                      \$



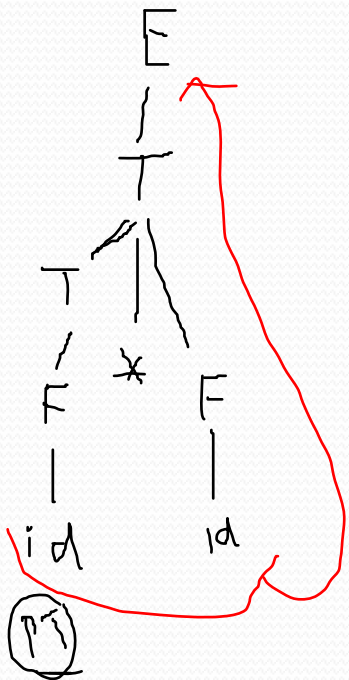
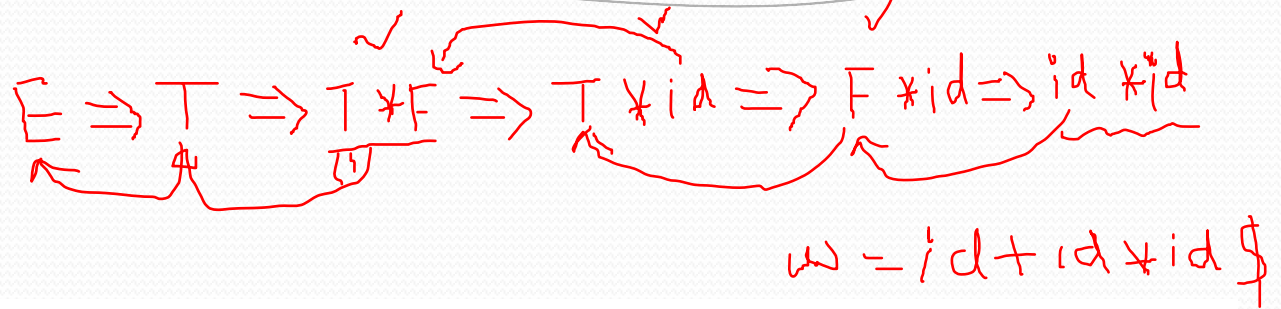
# Stack implementation of a Shift-reduce parser...

Ex.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid id$



STACK	INPUT	ACTION
$\$$	$\downarrow$ $id_1 * id_2 \$$	shift
$\$ id_1$	$* id_2 \$$	reduce by $F \rightarrow id$
$\$ F$	$* id_2 \$$	reduce by $T \rightarrow F$
$\$ T$	$* id_2 \$$	shift
$\$ T *$	$id_2 \$$	shift
$\$ T * id_2$	$\$$	reduce by $F \rightarrow id$
$\$ T * F$	$\$$	reduce by $T \rightarrow T * F$
$\$ T$	$\$$	reduce by $E \rightarrow T$
$\$ E$	$\$$	accept ✓

Configurations of a shift-reduce parser on input  $id_1 * id_2$

# Shift-Reduce Parsing -- Example

$S \rightarrow aABb$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

input string: aaabb

aaAbb

aAbb

aABb

S

↓ reduction

$w = aabb$

$S \xRightarrow{m} aABb \xRightarrow{m} aAbb \xRightarrow{m} aaAbb \xRightarrow{m} aaabb$

Stack	Input	Action
\$ aAB	b\$	shift
\$ aABb	\$	Reduce
\$ S	\$	Accept

STACK	INPUT	Action
\$	aaabb\$	shift
\$ a	abb\$	shift
\$ aa	bb\$	Reduce by $A \rightarrow a$
\$ aaA	bb\$	Reduce by $A \rightarrow aA$
\$ aA	bb\$	shift
\$ aAb	b\$	Reduce by $B \rightarrow b$

# Practice Problem.

Consider the grammar:

$$S \rightarrow 0S1 \mid 01$$

Show the moves made by shift-Reduce parse for the input string: 000111

*S*  $\rightarrow$  *0S1*      *0S1*  $\rightarrow$  *00S11*      *00S11*  $\rightarrow$  *000S111*

---

# Conflicts during shift reduce parsing

- Two kind of conflicts(during parsing)
  - Shift/Reduce conflict
  - Reduce/Reduce conflict
- Example: Shift/Reduce Conflict

$S \rightarrow iEtS \mid \underline{iEtSeS} \mid a$   
 $E \rightarrow b$

Stack

⋮

✓  $\$ \dots \underline{iEtS}$

⋮

Input

⋮

$e \dots \$$

⋮

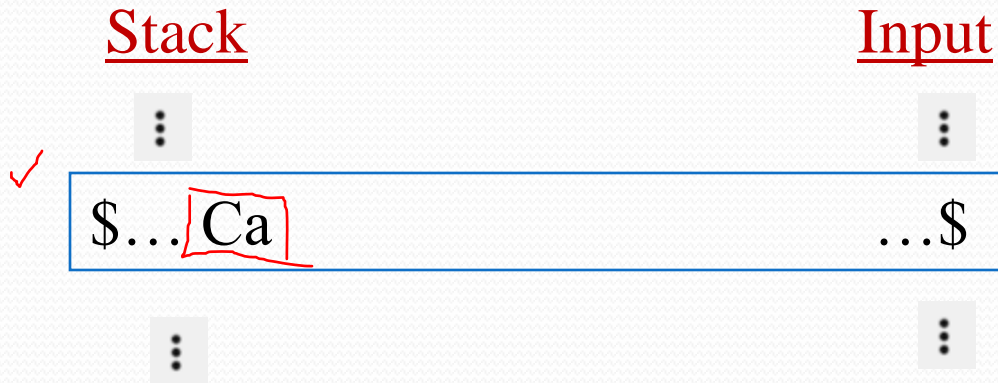
Parser's  
Configuration at  
some point in  
time

prefer Shift

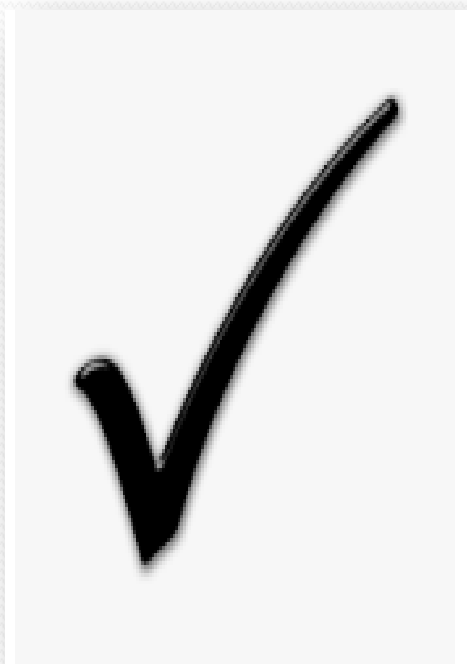


## Example: Reduce/Reduce conflict

✓  
A → Ca | Db  
✓  
B → Ca | b



Parser's  
Configuration at  
some point in  
time



# Introduction to LR Parsing: Simple LR

- The most prevalent type of bottom-up parsers
- LR(k), mostly interested on parsers with  $k=0$  or  $k=1$
- The “L” is for left-to-right scanning of input and “R” is constructing a RMD in reverse, and  $k$  is the number of input symbols of [lookahead](#) that are used in making parsing decisions( $k=1$  , for most practical applications).
- Why LR parsers?
  - Table driven
  - Can be constructed to recognize all programming language constructs
  - Most general shift-reduce parsing method
  - Can detect a syntactic error as soon as it is possible to do so

# Items and the LR(0) Automaton

How does a shift-reduce parser know when to shift and when to reduce? For example, with stack contents  $\$T$  and next input symbol  $*$  in Fig. 4.28, how does the parser know that  $T$  on the top of the stack is not a handle, so the appropriate action is to shift and not to reduce  $T$  to  $E$ ?

An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of “items.” An *LR(0) item* (*item* for short) of a grammar  $G$  is a production of  $G$  with a dot at some position of the body. Thus, production  $A \rightarrow XYZ$  yields the four items

$$A \rightarrow \cdot XYZ$$

$$A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z$$

$$A \rightarrow XYZ \cdot$$

The production  $A \rightarrow \epsilon$  generates only one item,  $A \rightarrow \cdot$ .

## Significance of $\cdot$ (dot) in body of production

Item  $A \rightarrow X \cdot YZ$  indicates that we have just seen on the input a string derivable from  $X$  and that we hope next to see a string derivable from  $YZ$ .

Item  $A \rightarrow XYZ \cdot$

indicates that we have seen the body  $XYZ$  and that it may be time to reduce  $XYZ$  to  $A$ .

One collection of sets of LR(0) items, called the *canonical* LR(0) collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an *LR(0) automaton*.<sup>3</sup> In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.

To construct the canonical LR(0) collection for a grammar, we define an augmented grammar and two functions, CLOSURE and GOTO. If  $G$  is a grammar with start symbol  $S$ , then  $G'$ , the *augmented grammar* for  $G$ , is  $G$  with a new start symbol  $S'$  and production  $S' \rightarrow S$ . The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by  $S' \rightarrow S$ .

# Constructing canonical LR(0) item sets

- Augment the grammar:
  - $G$  with addition of a production:  $S' \rightarrow S$
- CLOSURE of item sets:
  - If  $I$  is a set of items,  $\text{closure}(I)$  is a set of items constructed from  $I$  by the following rules:
    - Add every item in  $I$  to  $\text{closure}(I)$
    - If  $A \rightarrow \alpha.B\beta$  is in  $\text{closure}(I)$  and  $B \rightarrow \gamma$  is a production then add the item  $B \rightarrow \cdot\gamma$  to  $\text{closure}(I)$ .
- $\text{GOTO}(I, X)$  where  $I$  is an item set and  $X$  is a grammar symbol is closure of set of all items  $[A \rightarrow \alpha X \cdot \beta]$  where  $[A \rightarrow \alpha \cdot X \beta]$  is in  $I$

# SLR Parser Design for given Grammar G

G

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

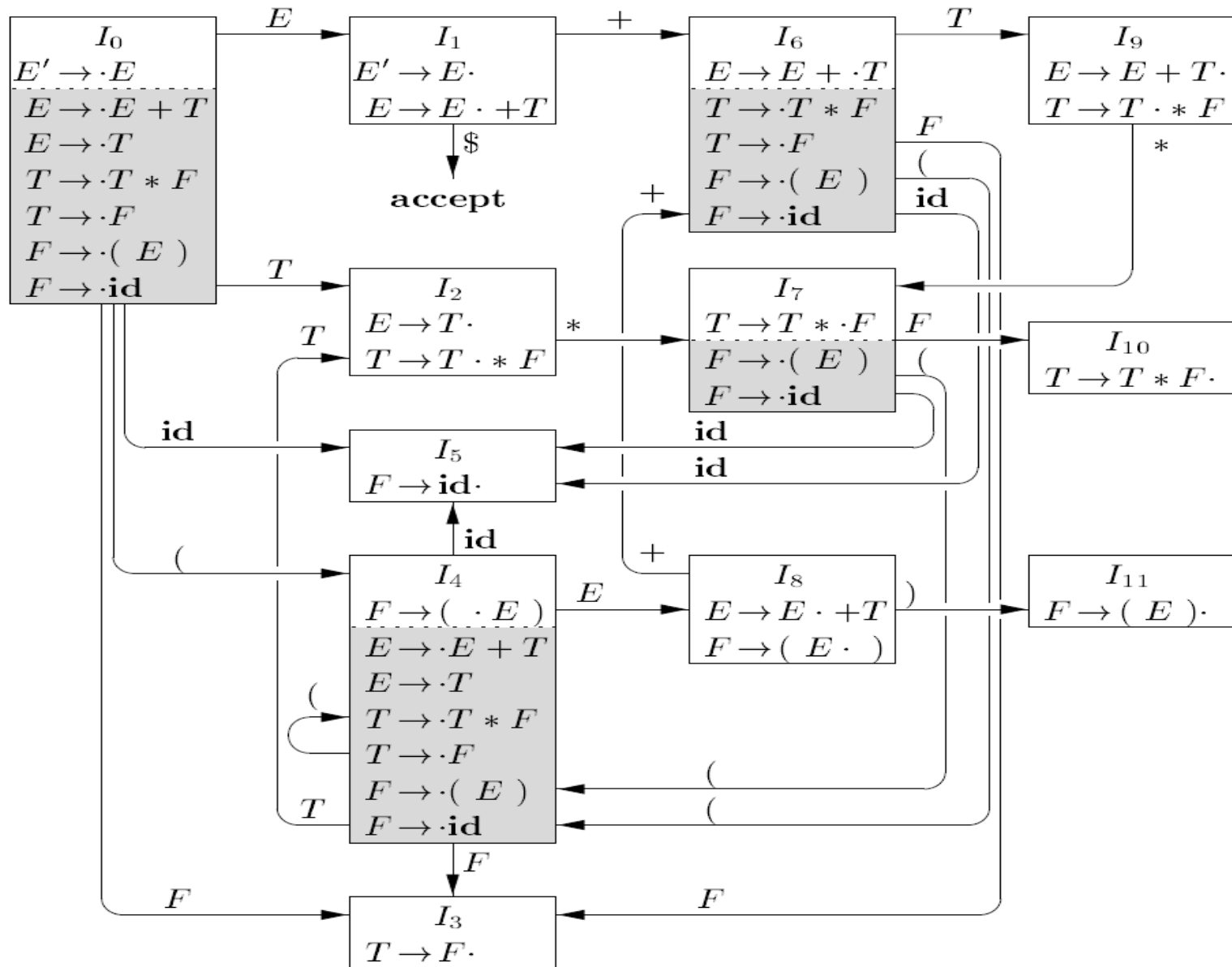
Augmentation



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



# Construct LR(0) Automata(DFA)



# Designing of SLR parsing table

- Method

- Construct  $C = \{I_0, I_1, \dots, I_n\}$ , the collection of LR(0) items for  $G'$
- State  $i$  is constructed from state  $I_i$ :
  - If  $[A \rightarrow \alpha.a\beta]$  is in  $I_i$  and  $GOTO(I_i, a) = I_j$ , then set  $ACTION[i, a]$  to “shift<sub>j</sub>”
  - If  $[A \rightarrow \alpha.]$  is in  $I_i$ , then set  $ACTION[i, a]$  to “reduce  $A \rightarrow \alpha$ ” for all  $a$  in  $follow(A)$
  - If  $[S' \rightarrow .S]$  is in  $I_i$ , then set  $ACTION[i, \$]$  to “Accept”
- **If any conflicts appears then we say that the grammar is not SLR(1).**
- If  $GOTO(I_i, A) = I_j$  then  $GOTO[i, A] = j$
- **All entries not defined by above rules are made “error”**
- The initial state of the parser is the one constructed from the set of items containing  $[S' \rightarrow .S]$

# SLR Parsing Table for G

Augmented grammar

$E' \rightarrow E$

$E \rightarrow E + T$  ---(1)

$E \rightarrow T$  ---(2)

$T \rightarrow T * F$  ---(3)

$T \rightarrow F$  ---(4)

$F \rightarrow (E)$  ---(5)

$F \rightarrow id$  ---(6)

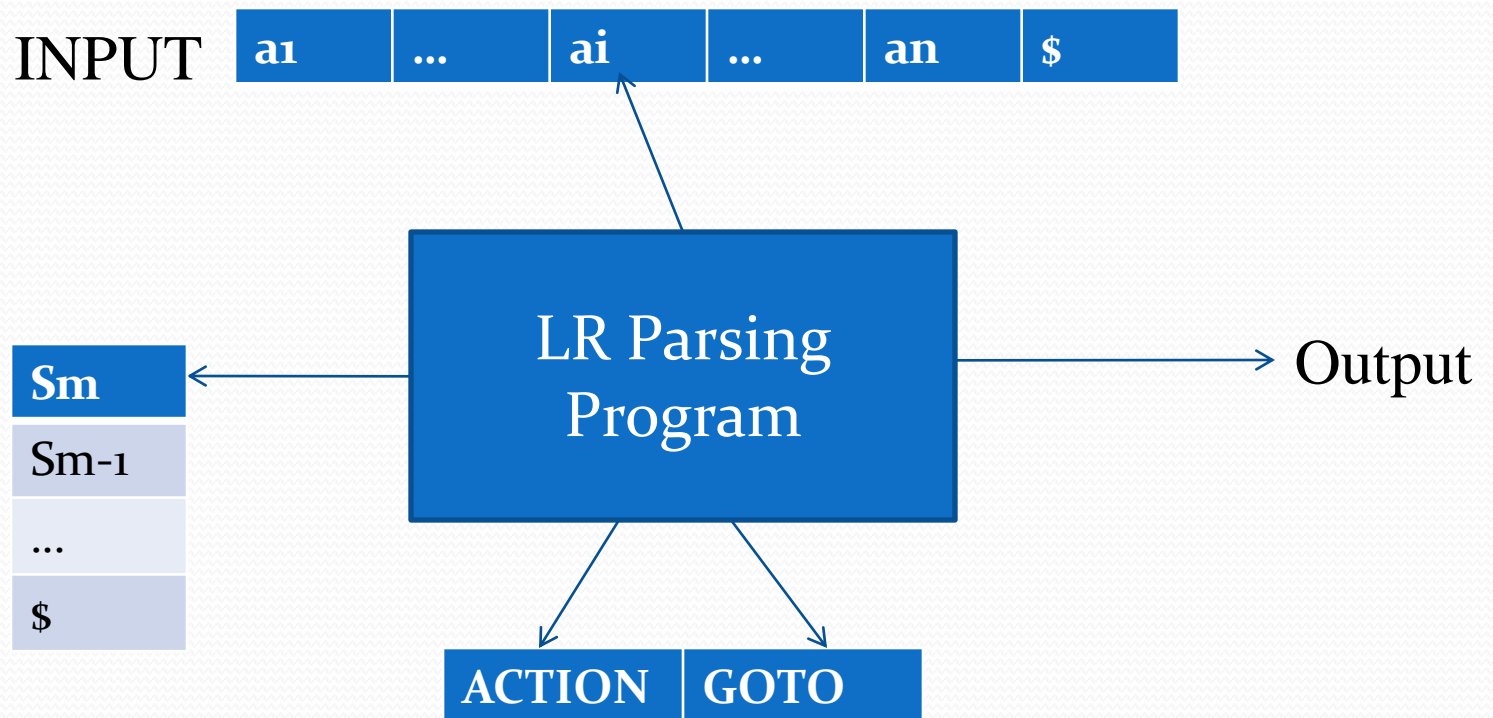
$Follow(E) = \{ \$, +, ) \}$

$Follow(T) = \{ \$, +, *, ) \}$

$Follow(F) = \{ \$, +, *, ) \}$

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

# LR-Parsing model



# LR parsing algorithm

```
let a be the first symbol of w$;
while(1)  { /*repeat forever */
    let s be the state on top of the stack;
    if (ACTION[s,a] = sj) {
        push j onto the stack;
        let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A → β) {
        pop |β| symbols off the stack;
        let state t now be on top of the stack;
        push GOTO[t, A] onto the stack;
    }
    else if (ACTION[s,a] = accept) break; /* parsing is done */
    else
        error(); /* Parsing is unsuccessful */
}
```

# Moves made by SLR parser on: $id*id+id$

Stack	Input	Action
\$0	$id*id+id\$$	Shift 5
\$05	$*id+id\$$	Reduce by $F \rightarrow id$ ( i.e 6 <sup>th</sup> prod.)
\$03	$*id+id\$$	Reduce by $T \rightarrow F$
\$02	$*id+id\$$	Shift 7
\$027	$id+id\$$	Shift 5
\$0275	$+id\$$	Reduce by $F \rightarrow id$
\$02710	$+id\$$	Reduce by $T \rightarrow T * F$
\$02	$+id\$$	Reduce by $E \rightarrow T$
\$01	$+id\$$	Shift 6
\$016	$id\$$	Shift 5
\$0165	$\$$	Reduce by $F \rightarrow id$
\$0163	$\$$	Reduce by $T \rightarrow F$
\$0169	$\$$	Reduce by $E \rightarrow E + T$
\$01	$\$$	accept

# SLR(1) parser design..another Example

Consider the following grammar:

$$S \rightarrow AA$$
$$A \rightarrow aA \mid b$$

- 1) Construct the DFA of LR(0) item sets.
- 2) Design/construct SLR parsing table.

# ✓ Augment the given Grammar

$S \rightarrow AA$   
 $A \rightarrow aA \mid b$

Given Grammar

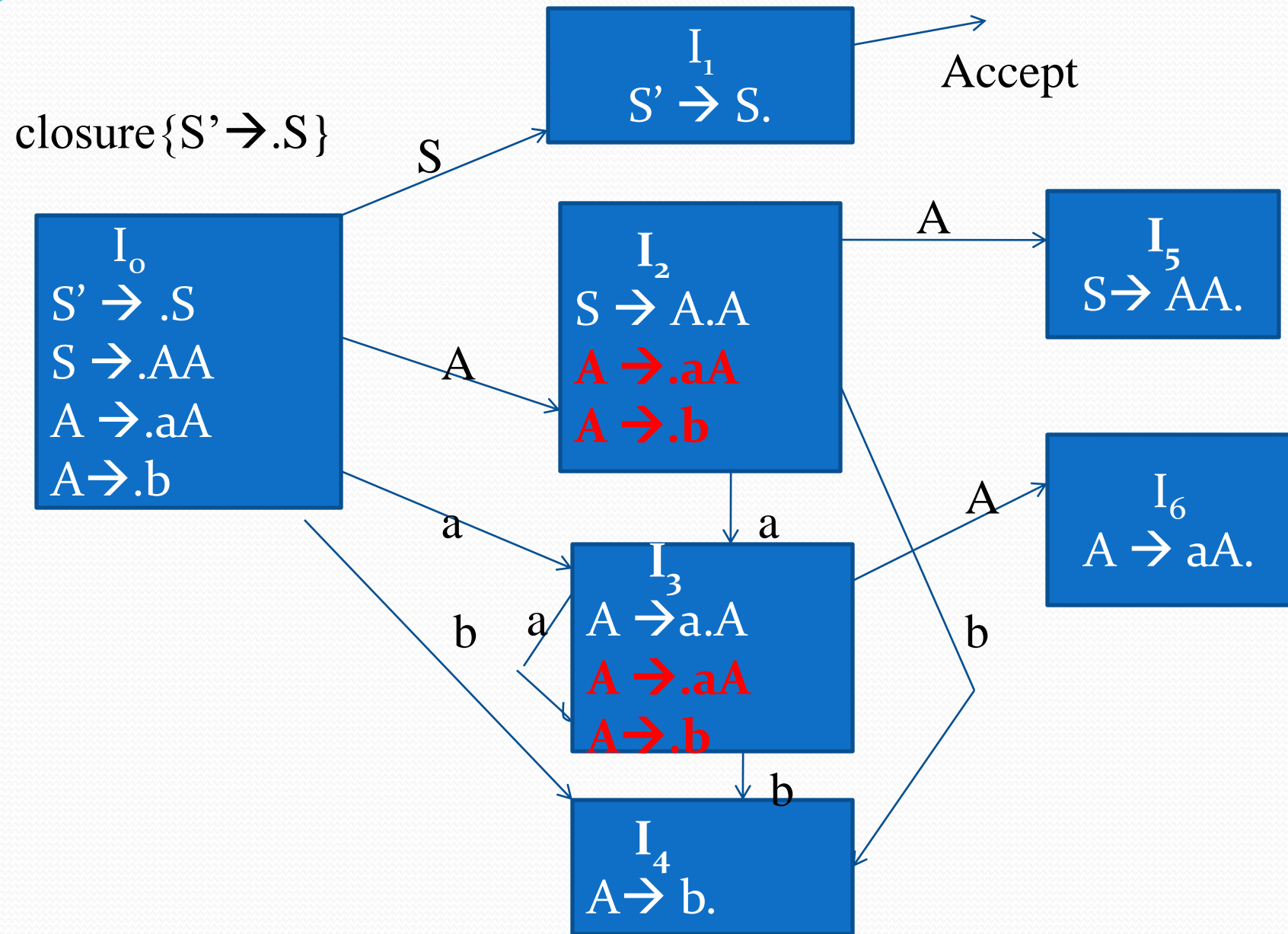


$S' \rightarrow S$   
 $S \rightarrow AA$   
 $A \rightarrow aA \mid b$

After augmentation



# ✓ Construct LR(0) items sets(automaton)



# ✓ Construction of SLR(1) Parsing Table...

$S' \rightarrow S$

$S \rightarrow AA$  ---(1)

$A \rightarrow aA$  --- (2)

$A \rightarrow b$  ----(3)

$FOLLOW(S) = \{ \$ \}$

$FOLLOW(A) = \{ a, b, \$ \}$

STATES	ACTION			GOTO	
	a	b	\$	S	A
0	S <sub>3</sub>	S <sub>4</sub>		1	2
1			Accept		
2	S <sub>3</sub>	S <sub>4</sub>			5
3	S <sub>3</sub>	S <sub>4</sub>			6
4	R <sub>3</sub>	R <sub>3</sub>	R <sub>3</sub>		
5			R <sub>1</sub>		
6	R <sub>2</sub>	R <sub>2</sub>	R <sub>2</sub>		

SLR(1) Parsing Table

# Practice Problem...

Consider the following grammar:

$$S \rightarrow AaBb \mid BbBa$$
$$A \rightarrow \varepsilon$$
$$B \rightarrow \varepsilon$$

- 1) Obtain the canonical collection of LR(0) items sets.
- 2) Construct the SLR parsing table
- 3) Is the grammar SLR(1)? If not, give reasons?



# Module-4

## Topic: Introduction to lex and yacc

**Lex and Yacc** –The Simplest Lex Program, Grammars, Parser-Lexer Communication, A YACC Parser, The Rules Section, Running LEX and YACC, LEX vs. Hand-written Lexers.

**Using LEX** - Regular Expressions, Examples of Regular Expressions, A Word Counting Program.

**Using YACC** – Grammars, Recursive Rules, Shift/Reduce Parsing, What YACC Cannot Parse, A YACC Parser - The Definition Section, The Rules Section, The LEXER, Compiling and Running a Simple Parser, Arithmetic Expressions and Ambiguity.

# A Step - Back: Phases of Compiler

- The main job of a *lexical analyzer* (*scanner*) is to break up an input stream into more usable elements (*tokens*)

1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

position = initial + rate \* 60

Lexical Analyzer

$\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle * \rangle \langle 60 \rangle$

Syntax Analyzer / *PARSER*

$\langle \text{id}, 1 \rangle = \langle \text{id}, 2 \rangle + \langle \text{id}, 3 \rangle * 60$

Semantic Analyzer

$\langle \text{id}, 1 \rangle = \langle \text{id}, 2 \rangle + \langle \text{id}, 3 \rangle * \text{inttofloat}$

60

# Chapter-1: Introduction

- Lex and Yacc help us to write a programs that transform structured input.
- Enormous range of applications from simple search application to a compiler that transform a source program into optimized object code.
- In a program with structured input, two tasks that occur over & over are:
  1. *Dividing the input into meaningful units called lexeme(tokens).*
  2. *Then discovering the relationship among these tokens.*

For a C program ,the tokens are variables, keywords, constants, strings, operators, punctuations and so forth.

- This division into tokens is known as Lexical Analysis or lexing for short.

The token descriptions that lex uses are known as **regular expressions** and lex turns these REs into a form that the lexar can use to scan the input text extremely fast to match and recognize lexmes.

- As the input is divided into lexemes, a program often needs to establish the relationship among the tokens - called **Parsing**.
- The list of Rules that define the relationship that program understands is called a **Grammar**.
- The Yacc takes a concise description of a grammar and produces a C -routine that can parse that grammar, a **Parser**.
- The Yacc parser automatically detects whenever a sequence of input tokens matches one of the rules in the grammar and also detects a **syntax error** whenever its input doesn't match with any rules(productions).

*"When a task involves dividing input into units and establishing some relationship among those units, you should think of **Lex and Yacc**"*



# Lex v.s. Yacc

- Lex

- Lex generates C code for a lexical analyzer, or scanner
- Lex uses patterns that match strings in the input and converts the strings to tokens

- Yacc

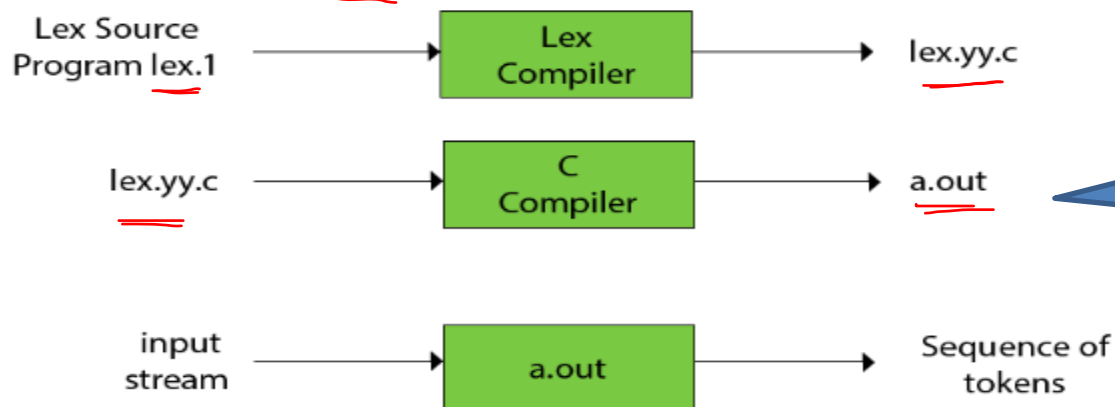
- Yacc generates C code for syntax analyzer, or parser.
- Yacc uses grammar rules that allow it to analyze tokens from Lex and create a syntax tree.

# LEX

- Lex is a program that generates lexical analyzer. It is used with YACC parser generator.
- The lexical analyzer is a program that transforms an input stream into a sequence of tokens.
- It reads the input stream and produces the source code as output through implementing the lexical analyzer in the C program.

## The function of Lex is as follows:

- Firstly lexical analyzer creates a program lex.1 in the Lex language. Then Lex compiler runs the lex.1 program and produces a C program lex.yy.c.
- Finally C compiler runs the lex.yy.c program and produces an object program a.out.
- a.out is lexical analyzer that transforms an input stream into a sequence of tokens.



Creating & Executing Lex Program

# The Simplest Lex Program

The simplest lex program copies its input to standard output:

rule section

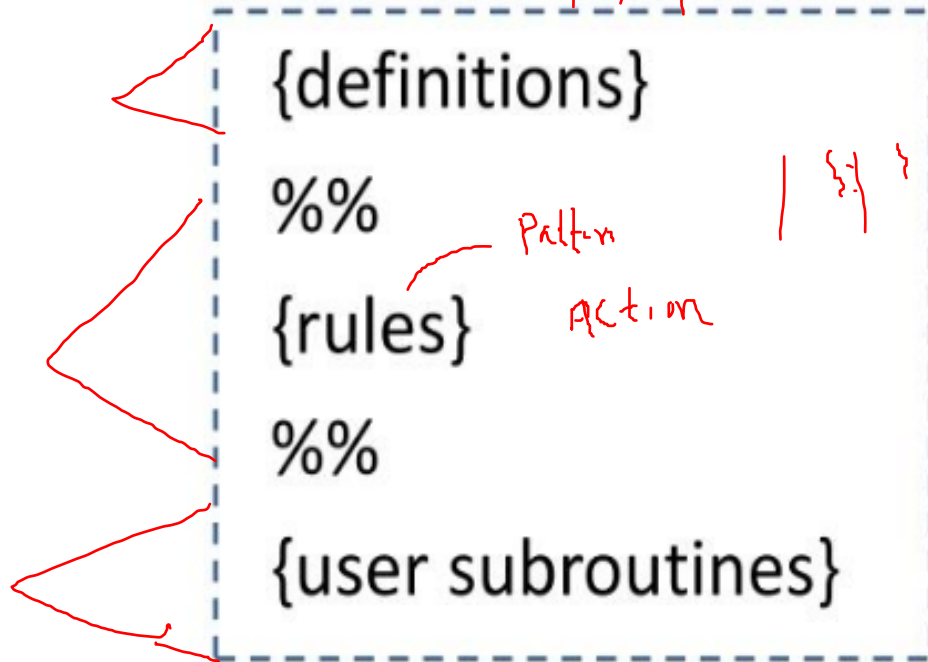
```
%%  
(. | \n) { printf("%s", yytext); }  
%%
```

```
main()  
{  
    printf(" Enter input string\n");  
    yylex();  
}
```

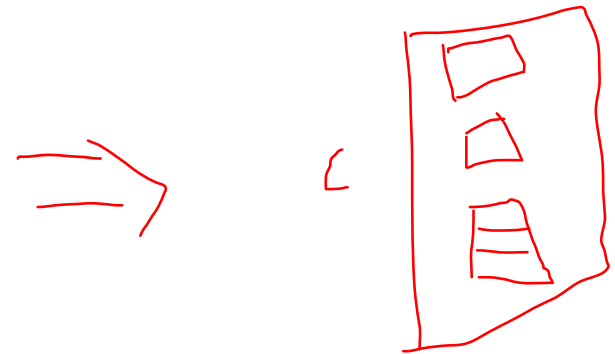
Lex automatically generate the actual C program code needed to handle reading the input and writing the output.

# Lex Source *Fig 9.11*

- General format



*#include <stdio.h>*  
*int main() {*  
*int n, y, z;*



A Lex program is separated into three sections by %% delimiters. The format of Lex source is as follows:

- The **definition** section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The **rules** section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- The **C code** section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

# Lex program to Recognizing C Keywords.

```
%{  
  
    #include<stdio.h>    /* Program to recognize C Keywords */  
  
    %}  
  
    %  
  
    {int|float| for | do | switch | if |else } { printf(“%s: Keyword”, yytext); }  
  
    %%  
  
    main( )  
    {  
        printf(“ Enter the string\n”);  
        yylex( );  
  
    }
```

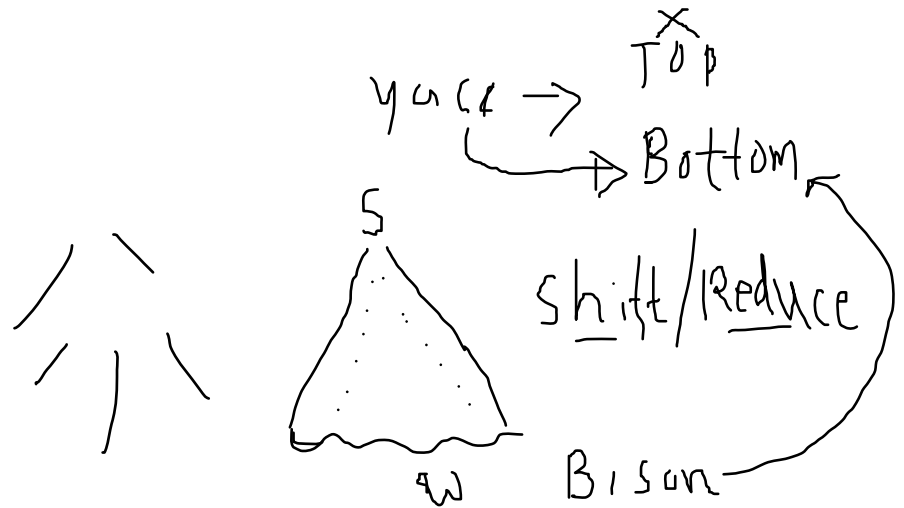
--// → int switch else ... main

# Grammar:



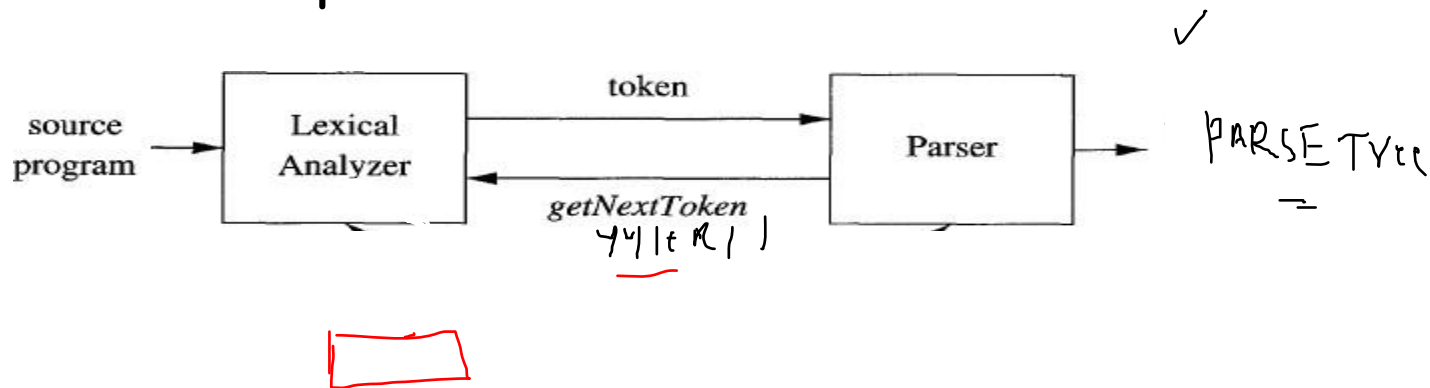
- For some applications, the simple kind of word recognition may be more than enough.
- But, others need to recognize specific sequences of tokens and establish relations among tokens using appropriate rules.
- Traditionally, a description of such a set of rules is known as a Grammar.

## Ex: Expression grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow ( E ) \mid \text{num} \end{aligned}$$


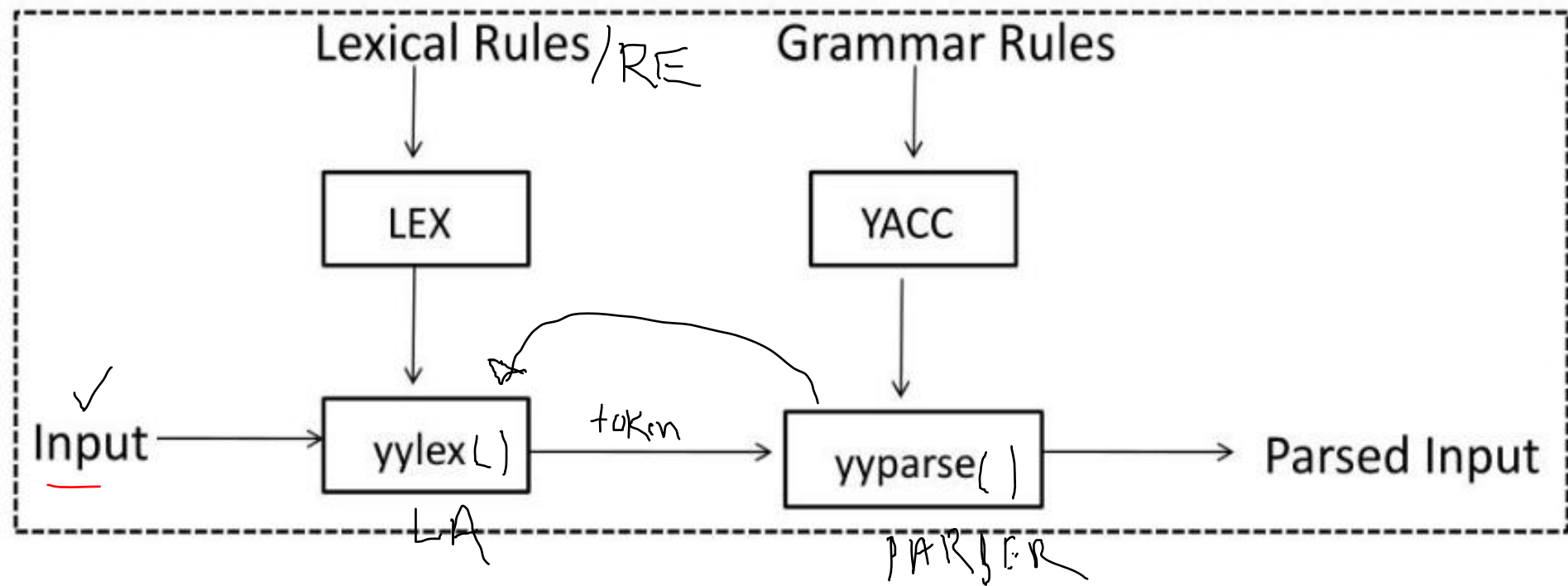
# Parser- Lexer Communication

- When we use a lex scanner and a yacc parser together, the parser is the high level routine .
- It calls the lexer `yylex( )` whenever it needs a token from the input. The lexer then scans through the input recognizing tokens.
- As soon as it finds a token of interest to the parser, it returns to the parser.



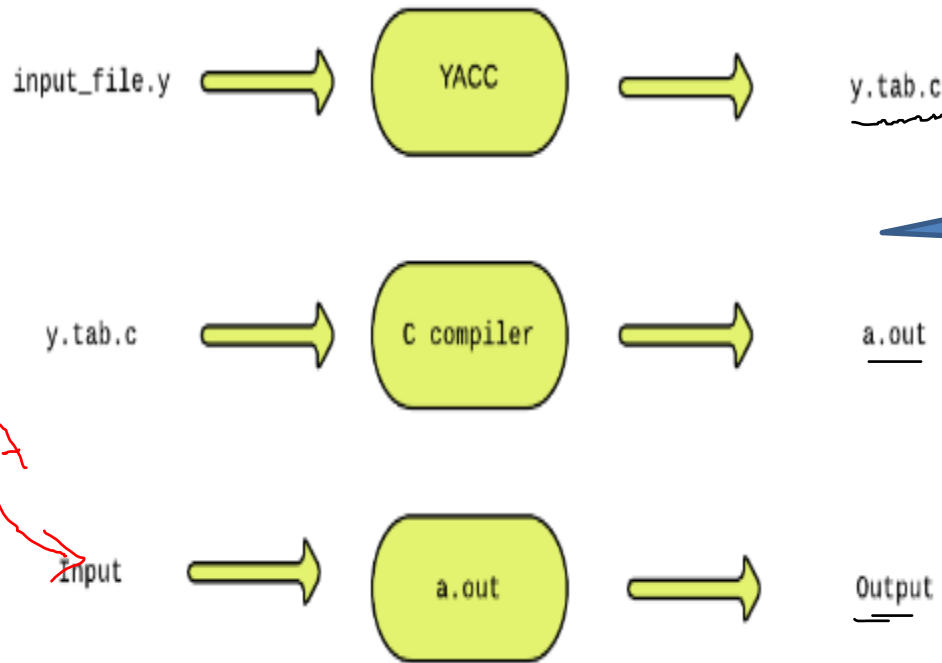


- Can also be used with a parser generator
- Lex programs recognize only regular expressions
- Yacc creates parsers that accepts a large class of context-free grammars



# Yacc

Yacc (Yet another compiler compiler) is a tool used to generate a parser. Yacc translates a given Context Free Grammar (CFG) specifications into a C implementation. This C program when compiled, yields an executable parser.



yy\_parser.c  
{  
}

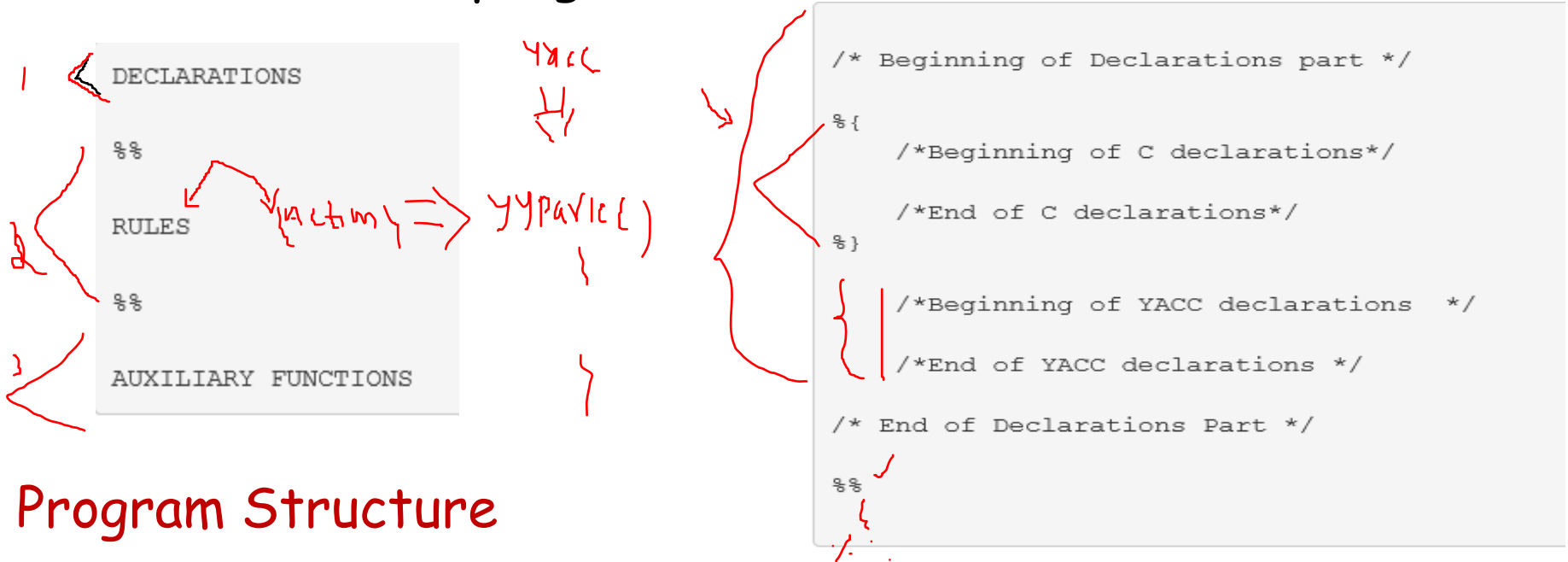
Steps in  
Creating and  
Executing Yacc  
Programs

LA

✓  
✓  
✓  
✓

# The Structure of Yacc Program

A YACC program consists of three sections: Declarations, Rules and Auxiliary functions. (Note the similarity with the structure of LEX programs).



## Program Structure



## Example

**Declarations:** The declarations section consists of two parts: (i) C declarations and (ii) YACC declarations. The C Declarations are delimited by %{ and %}.

**Rules:** A rule in a YACC program comprises of two parts (i) the productions/rules part and (ii) the action part. A rule in YACC is of the form:

Production\_head : production\_body { action in C }

### **Auxiliary Functions:**

The Auxiliary functions section contains the definitions of three mandatory functions main(), yylex() and yyerror(). You may wish to add your own functions (depending on the the requirement for the application) in the y.tab.c file. Such functions are written in the auxiliary functions section. The main( ) function must invoke yparse( ) to parse the input.

# Yacc program to recognize string with grammar $\{ a^n b^n \mid n \geq 0 \}$

$\int$   $\int$   $a$

```
%{
  /* Definition section */
  #include "y.tab.h"
}%

/* Rule Section */
%%

[a]      { return A; }
[b]      { return B; }

%%
```

**Lex program**

```
%{
  #include<stdio.h>      /* Definition section */
  #include<stdlib.h>
}%
%token  A B              /* A,B are tokens from lexer */
%%
S : X '\n' { printf("Valid String\n"); exit(0); }
  ;
X : A X B |
  ;
%%

int yyerror(char *msg)
{
  printf("Invalid string\n");
  exit(0);
}

main()
{
  printf("Enter the string\n");
  yyparse();
}
```

**Yacc program**

## Running Lex and Yacc( in Ubuntu)

```
$ lex file1.l
```

```
$ yacc -d file2.y
```

```
$ cc lex.yy.c y.tab.c
```

```
$ ./a.out
```

*bab* → Valid  
*bab* ⇒ Invalid

# USING Lex



- When we write a lex specification, we create a set of patterns which lex matches against input string.
- Each time one of the patterns matches, the lex program invokes C code in the action portion which does something with the matched text.
- This way a lex program divides the input into lexeme.
- Lex translates the lex specification into a file containing a C function called yylex( ).
- Using regular C compiler, we can compile the file (i.e. lex.yy.c) that lex produced.

# Regular Expressions

$$\Sigma = \{ a, b, \epsilon \}$$

+ , \* , |

$$(a|b)^* (ab)$$

$\overline{abba}$   
 $\overline{abab}$

You specify the patterns you are interested in with a notation called a regular expression. A regular expression is formed by stringing together characters with or without operators. The simplest regular expressions are strings of text characters with no operators at all.



# Regular Expressions

RE CHARACTERS	DESCRIPTION
.	Matches any single character except newline <u>character</u>
*	Matches zero or more copies of preceding expression
+	Matches one or more occurrences of preceding expression
^	Matches the beginning of a line as first character of a RE
\$	Matches the end of a line as the last character of a RE
{}	Indicates how many times the previous pattern is allowed to match
\	Used to escape meta-characters, as part of C escape sequences
?	Matches zero or one occurrence of the preceding expression
	Matches either the preceding RE or the following expression
[]	Character class, matches any character within the brackets
"....."	Interprets everything within quotation marks literally
a/b	Matches the preceding RE but only if followed by the following RE
()	Groups a series of RE's together into a new RE

$[abc] = a|b|c$

# Examples of Regular Expressions

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc , abcabc abcabcabc ...
a(bc)?	a , abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\ -z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[ \t\n]+	whitespace
[^ab]	anything except: a, b
[a^b]	one of: a, ^, b
[a b]	one of: a,  , b
a b	one of: a, b

$(A|B|C|\dots|z|a|b|\dots|z|0|1|\dots|9)^+$   
 $(A-Ba-b0-9)^+$

# Ex. Lex specification to recognize an identifiers.

```
%%  
[\n\t ]+ { ; }  
[a-zA-Z][a-zA-Z0-9_]* { printf(" Identifier\n"); }  
. {;}  
%%  
  
main ( )  
{  
  yylex( );  
}
```

$[a-zA-Z][a-zA-Z0-9_]*$



$\sqrt{a-zA-Z}$

96abc\*

# Ex. Lex specification

```
%%
[\\n\\t]+  {;}

[0-9]+  {
    printf(" INTEGER\\n");
}

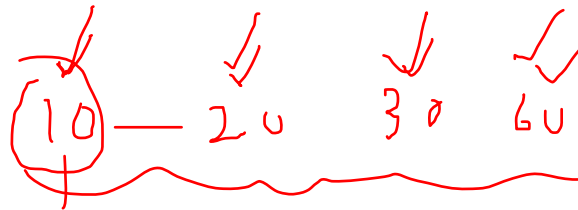
%%

main( )
{
    yylex( );
}
```

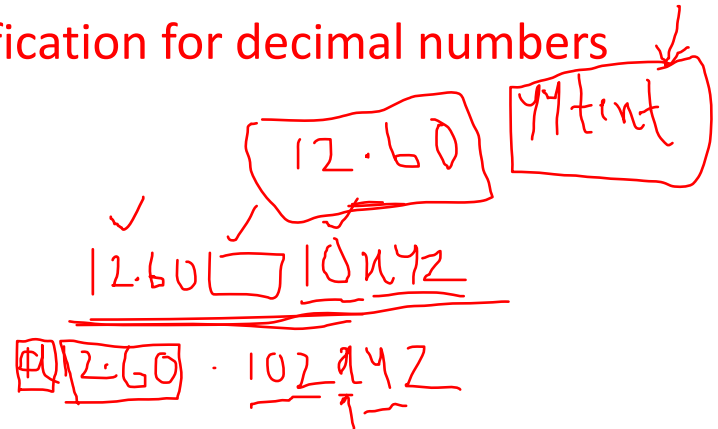
```
%% ✓
[\\n\\t]+  {;}
✓
( [0-9]+ ) | ( [0-9]* \\.[0-9]+ ) { printf("REAL No.\\n"); }
✓
.  {;}
%%

main( )
{
    yylex( );
}
```

Lex specification for integer numbers



Lex specification for decimal numbers



# A Word Counting Example

```
%{
    unsigned int charCount = 0, wordCount = 0, lineCount = 0;
}%}

word    [^\t\n]+
eol     \n      }  Substitution Definition

%%

{word}  {wordCount++; charCount+=yyleng;}

{eol}   {charCount++; lineCount++;}

.       charCount++;

%%

main()
{
    yylex();
    printf("%d %d %d\n", lineCount, charCount, wordCount);
}
```

# Lex program to match " Hello World"

```
%%
```

```
" Hello World" { printf(" GOOD BY\n"); }  
.  
{; }
```

```
%%
```

```
main ( )
```

```
{
```

```
printf( " Enter a string\n");
```

```
yylex( );
```

```
}
```

enl

## Write a lex program to find the number of vowels and consonants

```
%{      /* to find vowels and consonants*/
int vowels = 0; int consonants = 0;
%}
%%
[ \t\n]+  ;
[aeiouAEIOU]    { vowels++;}
[bcdfghjklmnpqrstvwxyzBCDFGHJKLMNPQRSTVWXYZ] { consonants++;}
.  ;
%%
main()
{
yylex();
printf(" The number of vowels = %d\n", vowels);
printf(" number of consonents = %d \n", consonents);
}

yywrap( )
{
return 1;
}
```

**Write a lex program to find the number of positive integer, negative integer number .**

```
%{
int posnum = 0,negnum = 0, posflo = 0, negflo = 0;
}%
%%
[\\n\\t]          ;
([0-9]+)          { posnum++;}
-?([0-9]+)        { negnum++; }
.                  ECHO;
%%
main()
{
yylex();
printf("Number of positive numbers = %d\\n", posnum); printf("number of
negative numbers = %d\\n", negnum);
}
```



## RECOMMENDED QUESTIONS:

1. write the specification of lex with an example?
2. what is regular expressions? With examples explain?
3. write a lex program to count the no of words , lines , space, characters?
4. write a lex program to count the no of vowels and consonants?
5. what is lexer- parser communication? Explain?
6. write a program to count no of words by the method of substitution?
7. Write a LEX program to eliminate *comment lines in a C program and copy the resulting program into a separate file.*
8. Develop a LEX program to count number of words, lines and characters in a given file.
9. Explain the general structure of YACC program with suitable example.?
10. What is grammar? How does yacc parse a tree?
11. How do you compile and run a yacc program? Explain.
12. Explain the ambiguity occurring in an grammar with an example?
13. Explain Shift - Reduce parsing ?
14. Write a Yacc program to recognize an valid variable which starts with letter followed by a digit. The letter should be in lowercase only.
15. Write a yacc program to recognize the grammar {  $a^n b$  for  $n \geq 0$ }.
16. Write YACC program to evaluate *arithmetic expression involving operators: +, -, \*, and /*
17. Write YACC program to recognize valid *identifier, operators and keywords in the given text (C program) file.*
18. Write a yacc program to test the validity of an arthimetic expressions

## Chapter 5

# Syntax-Directed Translation

This chapter develops the theme of Section 2.3: the translation of languages guided by context-free grammars. The translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. The techniques are also useful for implementing little languages for specialized tasks; this chapter includes an example from typesetting.

We associate information with a language construct by attaching *attributes* to the grammar symbol(s) representing the construct, as discussed in Section 2.3.2. A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

$$\begin{array}{ll} \text{PRODUCTION} & \text{SEMANTIC RULE} \\ E \rightarrow E_1 + T & E.\text{code} = E_1.\text{code} \parallel T.\text{code} \parallel '+' \end{array} \quad (5.1)$$

This production has two nonterminals,  $E$  and  $T$ ; the subscript in  $E_1$  distinguishes the occurrence of  $E$  in the production body from the occurrence of  $E$  as the head. Both  $E$  and  $T$  have a string-valued attribute *code*. The semantic rule specifies that the string  $E.\text{code}$  is formed by concatenating  $E_1.\text{code}$ ,  $T.\text{code}$ , and the character '+'. While the rule makes it explicit that the translation of  $E$  is built up from the translations of  $E_1$ ,  $T$ , and '+', it may be inefficient to implement the translation directly by manipulating strings.

From Section 2.3.5, a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies, as in

$$E \rightarrow E_1 + T \{ \text{print } '+' \} \quad (5.2)$$

By convention, semantic actions are enclosed within curly braces. (If curly braces occur as grammar symbols, we enclose them within single quotes, as in

{' and '}'.) The position of a semantic action in a production body determines the order in which the action is executed. In production (5.2), the action occurs at the end, after all the grammar symbols; in general, semantic actions may occur at any position in a production body.

Between the two notations, syntax-directed definitions can be more readable, and hence more useful for specifications. However, translation schemes can be more efficient, and hence more useful for implementations.

The most general approach to syntax-directed translation is to construct a parse tree or a syntax tree, and then to compute the values of attributes at the nodes of the tree by visiting the nodes of the tree. In many cases, translation can be done during parsing, without building an explicit tree. We shall therefore study a class of syntax-directed translations called “L-attributed translations” (L for left-to-right), which encompass virtually all translations that can be performed during parsing. We also study a smaller class, called “S-attributed translations” (S for synthesized), which can be performed easily in connection with a bottom-up parse.

## 5.1 Syntax-Directed Definitions

A *syntax-directed definition* (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions. If  $X$  is a symbol and  $a$  is one of its attributes, then we write  $X.a$  to denote the value of  $a$  at a particular parse-tree node labeled  $X$ . If we implement the nodes of the parse tree by records or objects, then the attributes of  $X$  can be implemented by data fields in the records that represent the nodes for  $X$ . Attributes may be of any kind: numbers, types, table references, or strings, for instance. The strings may even be long sequences of code, say code in the intermediate language used by a compiler.

### 5.1.1 Inherited and Synthesized Attributes

We shall deal with two kinds of attributes for nonterminals:

1. A *synthesized attribute* for a nonterminal  $A$  at a parse-tree node  $N$  is defined by a semantic rule associated with the production at  $N$ . Note that the production must have  $A$  as its head. A synthesized attribute at node  $N$  is defined only in terms of attribute values at the children of  $N$  and at  $N$  itself.
2. An *inherited attribute* for a nonterminal  $B$  at a parse-tree node  $N$  is defined by a semantic rule associated with the production at the parent of  $N$ . Note that the production must have  $B$  as a symbol in its body. An inherited attribute at node  $N$  is defined only in terms of attribute values at  $N$ 's parent,  $N$  itself, and  $N$ 's siblings.

### An Alternative Definition of Inherited Attributes

No additional translations are enabled if we allow an inherited attribute  $B.c$  at a node  $N$  to be defined in terms of attribute values at the children of  $N$ , as well as at  $N$  itself, at its parent, and at its siblings. Such rules can be “simulated” by creating additional attributes of  $B$ , say  $B.c_1, B.c_2, \dots$ . These are synthesized attributes that copy the needed attributes of the children of the node labeled  $B$ . We then compute  $B.c$  as an inherited attribute, using the attributes  $B.c_1, B.c_2, \dots$  in place of attributes at the children. Such attributes are rarely needed in practice.

While we do not allow an inherited attribute at node  $N$  to be defined in terms of attribute values at the children of node  $N$ , we do allow a synthesized attribute at node  $N$  to be defined in terms of inherited attribute values at node  $N$  itself.

Terminals can have synthesized attributes, but not inherited attributes. Attributes for terminals have lexical values that are supplied by the lexical analyzer; there are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

**Example 5.1:** The SDD in Fig. 5.1 is based on our familiar grammar for arithmetic expressions with operators  $+$  and  $*$ . It evaluates expressions terminated by an endmarker  $\mathbf{n}$ . In the SDD, each of the nonterminals has a single synthesized attribute, called *val*. We also suppose that the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer.

PRODUCTION	SEMANTIC RULES
1) $L \rightarrow E \mathbf{n}$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow ( E )$	$F.val = E.val$
7) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit.lexval}$

Figure 5.1: Syntax-directed definition of a simple desk calculator

The rule for production 1,  $L \rightarrow E \mathbf{n}$ , sets  $L.val$  to  $E.val$ , which we shall see is the numerical value of the entire expression.

Production 2,  $E \rightarrow E_1 + T$ , also has one rule, which computes the *val* attribute for the head  $E$  as the sum of the values at  $E_1$  and  $T$ . At any parse-

tree node  $N$  labeled  $E$ , the value of  $val$  for  $E$  is the sum of the values of  $val$  at the children of node  $N$  labeled  $E$  and  $T$ .

Production 3,  $E \rightarrow T$ , has a single rule that defines the value of  $val$  for  $E$  to be the same as the value of  $val$  at the child for  $T$ . Production 4 is similar to the second production; its rule multiplies the values at the children instead of adding them. The rules for productions 5 and 6 copy values at a child, like that for the third production. Production 7 gives  $F.val$  the value of a digit, that is, the numerical value of the token **digit** that the lexical analyzer returned.  $\square$

An SDD that involves only synthesized attributes is called *S-attributed*; the SDD in Fig. 5.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

For simplicity, the examples in this section have semantic rules without side effects. In practice, it is convenient to allow SDD's to have limited side effects, such as printing the result computed by a desk calculator or interacting with a symbol table. Once the order of evaluation of attributes is discussed in Section 5.2, we shall allow semantic rules to compute arbitrary functions, possibly involving side effects.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser. In fact, the SDD in Fig. 5.1 mirrors the Yacc program of Fig. 4.58, which illustrates translation during LR parsing. The difference is that, in the rule for production 1, the Yacc program prints the value  $E.val$  as a side effect, instead of defining the attribute  $L.val$ .

An SDD without side effects is sometimes called an *attribute grammar*. The rules in an attribute grammar define the value of an attribute purely in terms of the values of other attributes and constants.

### 5.1.2 Evaluating an SDD at the Nodes of a Parse Tree

To visualize the translation specified by an SDD, it helps to work with parse trees, even though a translator need not actually build a parse tree. Imagine therefore that the rules of an SDD are applied by first constructing a parse tree and then using the rules to evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an *annotated parse tree*.

How do we construct an annotated parse tree? In what order do we evaluate attributes? Before we can evaluate an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends. For example, if all attributes are synthesized, as in Example 5.1, then we must evaluate the  $val$  attributes at all of the children of a node before we can evaluate the  $val$  attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree; the evaluation of S-attributed definitions is discussed in Section 5.2.3.

For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes. For instance, consider nonterminals  $A$  and  $B$ , with synthesized and inherited attributes  $A.s$  and  $B.i$ , respectively, along with the production and rules

PRODUCTION	SEMANTIC RULES
$A \rightarrow B$	$A.s = B.i;$ $B.i = A.s + 1$

These rules are circular; it is impossible to evaluate either  $A.s$  at a node  $N$  or  $B.i$  at the child of  $N$  without first evaluating the other. The circular dependency of  $A.s$  and  $B.i$  at some pair of nodes in a parse tree is suggested by Fig. 5.2.

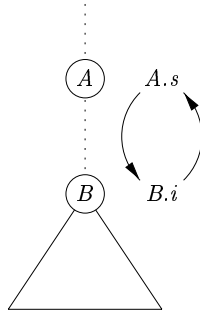


Figure 5.2: The circular dependency of  $A.s$  and  $B.i$  on one another

It is computationally difficult to determine whether or not there exist any circularities in any of the parse trees that a given SDD could have to translate.<sup>1</sup> Fortunately, there are useful subclasses of SDD's that are sufficient to guarantee that an order of evaluation exists, as we shall see in Section 5.2.

**Example 5.2:** Figure 5.3 shows an annotated parse tree for the input string  $3 * 5 + 4 \mathbf{n}$ , constructed using the grammar and rules of Fig. 5.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled  $*$ , after computing  $T.val = 3$  and  $F.val = 5$  at its first and third children, we apply the rule that says  $T.val$  is the product of these two values, or 15. □

Inherited attributes are useful when the structure of a parse tree does not “match” the abstract syntax of the source code. The next example shows how inherited attributes can be used to overcome such a mismatch due to a grammar designed for parsing rather than translation.

<sup>1</sup>Without going into details, while the problem is decidable, it cannot be solved by a polynomial-time algorithm, even if  $\mathcal{P} = \mathcal{NP}$ , since it has exponential time complexity.

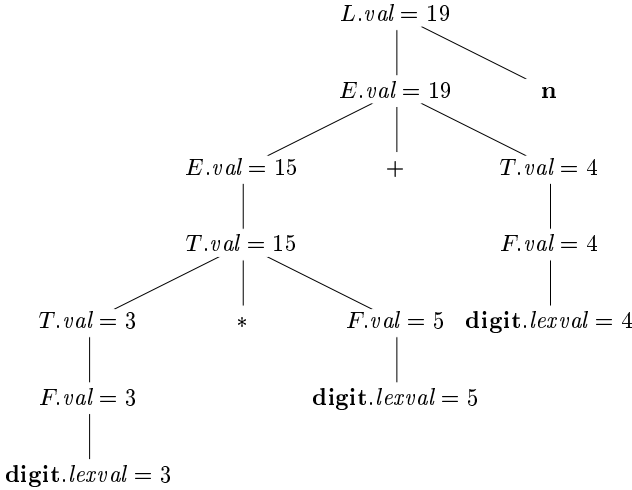


Figure 5.3: Annotated parse tree for  $3 * 5 + 4 n$

**Example 5.3:** The SDD in Fig. 5.4 computes terms like  $3 * 5$  and  $3 * 5 * 7$ . The top-down parse of input  $3 * 5$  begins with the production  $T \rightarrow F T'$ . Here,  $F$  generates the digit 3, but the operator  $*$  is generated by  $T'$ . Thus, the left operand 3 appears in a different subtree of the parse tree from  $*$ . An inherited attribute will therefore be used to pass the operand to the operator.

The grammar in this example is an excerpt from a non-left-recursive version of the familiar expression grammar; we used such a grammar as a running example to illustrate top-down parsing in Section 4.4.

PRODUCTION	SEMANTIC RULES
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Figure 5.4: An SDD based on a grammar suitable for top-down parsing

Each of the nonterminals  $T$  and  $F$  has a synthesized attribute  $val$ ; the terminal **digit** has a synthesized attribute  $lexval$ . The nonterminal  $T'$  has two attributes: an inherited attribute  $inh$  and a synthesized attribute  $syn$ .

The semantic rules are based on the idea that the left operand of the operator  $*$  is inherited. More precisely, the head  $T'$  of the production  $T' \rightarrow * F T'_1$  inherits the left operand of  $*$  in the production body. Given a term  $x * y * z$ , the root of the subtree for  $* y * z$  inherits  $x$ . Then, the root of the subtree for  $* z$  inherits the value of  $x * y$ , and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for  $3 * 5$  in Fig. 5.5. The leftmost leaf in the parse tree, labeled **digit**, has attribute value  $lexval = 3$ , where the 3 is supplied by the lexical analyzer. Its parent is for production 4,  $F \rightarrow \mathbf{digit}$ . The only semantic rule associated with this production defines  $F.val = \mathbf{digit}.lexval$ , which equals 3.

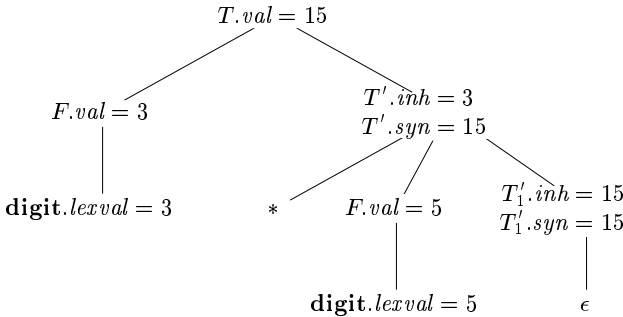


Figure 5.5: Annotated parse tree for  $3 * 5$

At the second child of the root, the inherited attribute  $T'.inh$  is defined by the semantic rule  $T'.inh = F.val$  associated with production 1. Thus, the left operand, 3, for the  $*$  operator is passed from left to right across the children of the root.

The production at the node for  $T'$  is  $T' \rightarrow * FT'_1$ . (We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for  $T'$ .) The inherited attribute  $T'_1.inh$  is defined by the semantic rule  $T'_1.inh = T'.inh \times F.val$  associated with production 2.

With  $T'.inh = 3$  and  $F.val = 5$ , we get  $T'_1.inh = 15$ . At the lower node for  $T'_1$ , the production is  $T' \rightarrow \epsilon$ . The semantic rule  $T'.syn = T'.inh$  defines  $T'_1.syn = 15$ . The  $syn$  attributes at the nodes for  $T'$  pass the value 15 up the tree to the node for  $T$ , where  $T.val = 15$ .  $\square$

### 5.1.3 Exercises for Section 5.1

**Exercise 5.1.1:** For the SDD of Fig. 5.1, give annotated parse trees for the following expressions:

- a)  $(3 + 4) * (5 + 6) \mathbf{n}$ .



b)  $1 * 2 * 3 * (4 + 5) \mathbf{n}$ .

c)  $(9 + 8 * (7 + 6) + 5) * 4 \mathbf{n}$ .

**Exercise 5.1.2:** Extend the SDD of Fig. 5.4 to handle expressions as in Fig. 5.1.

**Exercise 5.1.3:** Repeat Exercise 5.1.1, using your SDD from Exercise 5.1.2.

## 5.2 Evaluation Orders for SDD's

“Dependency graphs” are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

In this section, in addition to dependency graphs, we define two important classes of SDD's: the “S-attributed” and the more general “L-attributed” SDD's. The translations specified by these two classes fit well with the parsing methods we have studied, and most translations encountered in practice can be written to conform to the requirements of at least one of these classes.

### 5.2.1 Dependency Graphs

A *dependency graph* depicts the flow of information among the attribute instances in a particular parse tree; an edge from one attribute instance to another means that the value of the first is needed to compute the second. Edges express constraints implied by the semantic rules. In more detail:

- For each parse-tree node, say a node labeled by grammar symbol  $X$ , the dependency graph has a node for each attribute associated with  $X$ .
- Suppose that a semantic rule associated with a production  $p$  defines the value of synthesized attribute  $A.b$  in terms of the value of  $X.c$  (the rule may define  $A.b$  in terms of other attributes in addition to  $X.c$ ). Then, the dependency graph has an edge from  $X.c$  to  $A.b$ . More precisely, at every node  $N$  labeled  $A$  where production  $p$  is applied, create an edge to attribute  $b$  at  $N$ , from the attribute  $c$  at the child of  $N$  corresponding to this instance of the symbol  $X$  in the body of the production.<sup>2</sup>
- Suppose that a semantic rule associated with a production  $p$  defines the value of inherited attribute  $B.c$  in terms of the value of  $X.a$ . Then, the dependency graph has an edge from  $X.a$  to  $B.c$ . For each node  $N$  labeled  $B$  that corresponds to an occurrence of this  $B$  in the body of production  $p$ , create an edge to attribute  $c$  at  $N$  from the attribute  $a$  at the node  $M$

---

<sup>2</sup>Since a node  $N$  can have several children labeled  $X$ , we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

that corresponds to this occurrence of  $X$ . Note that  $M$  could be either the parent or a sibling of  $N$ .

**Example 5.4:** Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

At every node  $N$  labeled  $E$ , with children corresponding to the body of this production, the synthesized attribute  $val$  at  $N$  is computed using the values of  $val$  at the two children, labeled  $E_1$  and  $T$ . Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.  $\square$

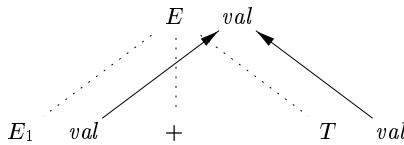


Figure 5.6:  $E.val$  is synthesized from  $E_1.val$  and  $T.val$

**Example 5.5:** An example of a complete dependency graph appears in Fig. 5.7. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 5.5.

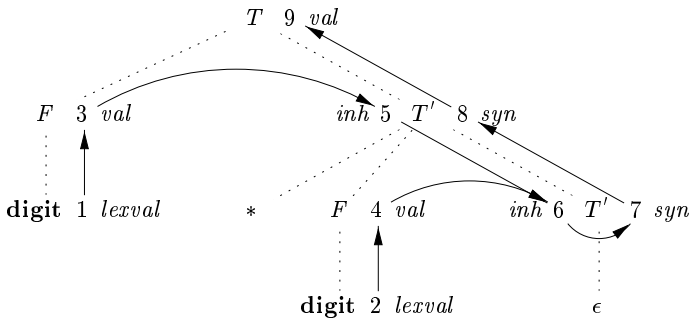


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled **digit**. Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled *F*. The edges to node 3 from 1 and to node 4 from 2 result

from the semantic rule that defines  $F.val$  in terms of  $\mathbf{digit.lexval}$ . In fact,  $F.val$  equals  $\mathbf{digit.lexval}$ , but the edge represents dependence, not equality.

Nodes 5 and 6 represent the inherited attribute  $T'.inh$  associated with each of the occurrences of nonterminal  $T'$ . The edge to 5 from 3 is due to the rule  $T'.inh = F.val$ , which defines  $T'.inh$  at the right child of the root from  $F.val$  at the left child. We see edges to 6 from node 5 for  $T'.inh$  and from node 4 for  $F.val$ , because these values are multiplied to evaluate the attribute  $inh$  at node 6.

Nodes 7 and 8 represent the synthesized attribute  $syn$  associated with the occurrences of  $T'$ . The edge to node 7 from 6 is due to the semantic rule  $T'.syn = T'.inh$  associated with production 3 in Fig. 5.4. The edge to node 8 from 7 is due to a semantic rule associated with production 2.

Finally, node 9 represents the attribute  $T.val$ . The edge to 9 from 8 is due to the semantic rule,  $T.val = T'.syn$ , associated with production 1.  $\square$

## 5.2.2 Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evaluate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node  $M$  to node  $N$ , then the attribute corresponding to  $M$  must be evaluated before the attribute of  $N$ . Thus, the only allowable orders of evaluation are those sequences of nodes  $N_1, N_2, \dots, N_k$  such that if there is an edge of the dependency graph from  $N_i$  to  $N_j$ , then  $i < j$ . Such an ordering embeds a directed graph into a linear order, and is called a *topological sort* of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort. To see why, since there are no cycles, we can surely find a node with no edge entering. For if there were no such node, we could proceed from predecessor to predecessor until we came back to some node we had already seen, yielding a cycle. Make this node the first in the topological order, remove it from the dependency graph, and repeat the process on the remaining nodes.

**Example 5.6:** The dependency graph of Fig. 5.7 has no cycles. One topological sort is the order in which the nodes have already been numbered: 1, 2,  $\dots$ , 9. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as 1, 3, 5, 2, 4, 6, 7, 8, 9.  $\square$

## 5.2.3 S-Attributed Definitions

As mentioned earlier, given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles. In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order,

since they do not permit dependency graphs with cycles. Moreover, the two classes introduced in this section can be implemented efficiently in connection with top-down or bottom-up parsing.

The first class is defined as follows:

- An SDD is *S-attributed* if every attribute is synthesized.

**Example 5.7:** The SDD of Fig. 5.1 is an example of an S-attributed definition. Each attribute, *L.val*, *E.val*, *T.val*, and *F.val* is synthesized.  $\square$

When an SDD is S-attributed, we can evaluate its attributes in any bottom-up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node *N* when the traversal leaves *N* for the last time. That is, we apply the function *postorder*, defined below, to the root of the parse tree (see also the box “Preorder and Postorder Traversals” in Section 2.3.4):

$$\begin{array}{l} \textit{postorder}(N) \{ \\ \quad \textbf{for} \text{ ( each child } C \text{ of } N, \text{ from the left ) } \textit{postorder}(C); \\ \quad \text{evaluate the attributes associated with node } N; \\ \} \end{array}$$

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head. This fact will be used in Section 5.4.2 to evaluate synthesized attributes and store them on the stack during LR parsing, without creating the tree nodes explicitly.

## 5.2.4 L-Attributed Definitions

The second class of SDD's is called *L-attributed definitions*. The idea behind this class is that, between the attributes associated with a production body, dependency-graph edges can go from left to right, but not from right to left (hence “L-attributed”). More precisely, each attribute must be either

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production  $A \rightarrow X_1 X_2 \cdots X_n$ , and that there is an inherited attribute  $X_i.a$  computed by a rule associated with this production. Then the rule may use only:
  - (a) Inherited attributes associated with the head *A*.
  - (b) Either inherited or synthesized attributes associated with the occurrences of symbols  $X_1, X_2, \dots, X_{i-1}$  located to the left of  $X_i$ .

- (c) Inherited or synthesized attributes associated with this occurrence of  $X_i$  itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this  $X_i$ .

**Example 5.8:** The SDD in Fig. 5.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \rightarrow F T'$	$T'.inh = F.val$
$T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$

The first of these rules defines the inherited attribute  $T'.inh$  using only  $F.val$ , and  $F$  appears to the left of  $T'$  in the production body, as required. The second rule defines  $T'_1.inh$  using the inherited attribute  $T'.inh$  associated with the head, and  $F.val$ , where  $F$  appears to the left of  $T'_1$  in the production body.

In each of these cases, the rules use information “from above or from the left,” as required by the class. The remaining attributes are synthesized. Hence, the SDD is L-attributed.  $\square$

**Example 5.9:** Any SDD containing the following production and rules cannot be L-attributed:

PRODUCTION	SEMANTIC RULES
$A \rightarrow B C$	$A.s = B.b;$ $B.i = f(C.c, A.s)$

The first rule,  $A.s = B.b$ , is a legitimate rule in either an S-attributed or L-attributed SDD. It defines a synthesized attribute  $A.s$  in terms of an attribute at a child (that is, a symbol within the production body).

The second rule defines an inherited attribute  $B.i$ , so the entire SDD cannot be S-attributed. Further, although the rule is legal, the SDD cannot be L-attributed, because the attribute  $C.c$  is used to help define  $B.i$ , and  $C$  is to the right of  $B$  in the production body. While attributes at siblings in a parse tree may be used in L-attributed SDD's, they must be to the left of the symbol whose attribute is being defined.  $\square$

### 5.2.5 Semantic Rules with Controlled Side Effects

In practice, translations involve side effects: a desk calculator might print a result; a code generator might enter the type of an identifier into a symbol table. With SDD's, we strike a balance between attribute grammars and translation schemes. Attribute grammars have no side effects and allow any evaluation order consistent with the dependency graph. Translation schemes impose left-to-right evaluation and allow semantic actions to contain any program fragment; translation schemes are discussed in Section 5.4.

We shall control side effects in SDD's in one of the following ways:

- Permit incidental side effects that do not constrain attribute evaluation. In other words, permit side effects when attribute evaluation based on any topological sort of the dependency graph produces a “correct” translation, where “correct” depends on the application.
- Constrain the allowable evaluation orders, so that the same translation is produced for any allowable order. The constraints can be thought of as implicit edges added to the dependency graph.

As an example of an incidental side effect, let us modify the desk calculator of Example 5.1 to print a result. Instead of the rule  $L.val = E.val$ , which saves the result in the synthesized attribute  $L.val$ , consider:

	PRODUCTION	SEMANTIC RULE
1)	$L \rightarrow E \mathbf{n}$	$print(E.val)$

Semantic rules that are executed for their side effects, such as  $print(E.val)$ , will be treated as the definitions of dummy synthesized attributes associated with the head of the production. The modified SDD produces the same translation under any topological sort, since the print statement is executed at the end, after the result is computed into  $E.val$ .

**Example 5.10:** The SDD in Fig. 5.8 takes a simple declaration  $D$  consisting of a basic type  $T$  followed by a list  $L$  of identifiers.  $T$  can be **int** or **float**. For each identifier on the list, the type is entered into the symbol-table entry for the identifier. We assume that entering the type for one identifier does not affect the symbol-table entry for any other identifier. Thus, entries can be updated in any order. This SDD does not check whether an identifier is declared more than once; it can be modified to do so.

	PRODUCTION	SEMANTIC RULES
1)	$D \rightarrow T L$	$L.inh = T.type$
2)	$T \rightarrow \mathbf{int}$	$T.type = \text{integer}$
3)	$T \rightarrow \mathbf{float}$	$T.type = \text{float}$
4)	$L \rightarrow L_1, \mathbf{id}$	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \rightarrow \mathbf{id}$	$addType(\mathbf{id}.entry, L.inh)$

Figure 5.8: Syntax-directed definition for simple type declarations

Nonterminal  $D$  represents a declaration, which, from production 1, consists of a type  $T$  followed by a list  $L$  of identifiers.  $T$  has one attribute,  $T.type$ , which is the type in the declaration  $D$ . Nonterminal  $L$  also has one attribute, which we call  $inh$  to emphasize that it is an inherited attribute. The purpose of  $L.inh$



### 5.2.6 Exercises for Section 5.2

**Exercise 5.2.1:** What are all the topological sorts for the dependency graph of Fig. 5.7?

**Exercise 5.2.2:** For the SDD of Fig. 5.8, give annotated parse trees for the following expressions:

- a) `int a, b, c.`
- b) `float w, x, y, z.`

**Exercise 5.2.3:** Suppose that we have a production  $A \rightarrow BCD$ . Each of the four nonterminals  $A$ ,  $B$ ,  $C$ , and  $D$  have two attributes:  $s$  is a synthesized attribute, and  $i$  is an inherited attribute. For each of the sets of rules below, tell whether (i) the rules are consistent with an S-attributed definition (ii) the rules are consistent with an L-attributed definition, and (iii) whether the rules are consistent with any evaluation order at all?

- a)  $A.s = B.i + C.s.$
- b)  $A.s = B.i + C.s$  and  $D.i = A.i + B.s.$
- c)  $A.s = B.s + D.s.$
- ! d)  $A.s = D.i$ ,  $B.i = A.s + C.s$ ,  $C.i = B.s$ , and  $D.i = B.i + C.i.$

! **Exercise 5.2.4:** This grammar generates binary numbers with a “decimal” point:

$$\begin{aligned} S &\rightarrow L . L \mid L \\ L &\rightarrow L B \mid B \\ B &\rightarrow 0 \mid 1 \end{aligned}$$

Design an L-attributed SDD to compute  $S.val$ , the decimal-number value of an input string. For example, the translation of string `101.101` should be the decimal number 5.625. *Hint:* use an inherited attribute  $L.side$  that tells which side of the decimal point a bit is on.

!! **Exercise 5.2.5:** Design an S-attributed SDD for the grammar and translation described in Exercise 5.2.4.

!! **Exercise 5.2.6:** Implement Algorithm 3.23, which converts a regular expression into a nondeterministic finite automaton, by an L-attributed SDD on a top-down parsable grammar. Assume that there is a token `char` representing any character, and that `char.lexval` is the character it represents. You may also assume the existence of a function `new()` that returns a new state, that is, a state never before returned by this function. Use any convenient notation to specify the transitions of the NFA.



## 5.3 Applications of Syntax-Directed Translation

The syntax-directed translation techniques in this chapter will be applied in Chapter 6 to type checking and intermediate-code generation. Here, we consider selected examples to illustrate some representative SDD's.

The main application in this section is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree. (Chapter 6 also discusses approaches to intermediate-code generation that apply an SDD without ever constructing a tree explicitly.)

We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

The final example of this section is an L-attributed definition that deals with basic and array types.

### 5.3.1 Construction of Syntax Trees

As discussed in Section 2.8.2, each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression  $E_1 + E_2$  has label  $+$  and two children representing the subexpressions  $E_1$  and  $E_2$ .

We shall implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function  $Leaf(op, val)$  creates a leaf object. Alternatively, if nodes are viewed as records, then  $Leaf$  returns a pointer to a new record for a leaf.
- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function  $Node$  takes two or more arguments:  $Node(op, c_1, c_2, \dots, c_k)$  creates an object with first field *op* and  $k$  additional fields for the  $k$  children  $c_1, \dots, c_k$ .

**Example 5.11:** The S-attributed definition in Fig. 5.10 constructs syntax trees for a simple expression grammar involving only the binary operators  $+$  and  $-$ . As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node*, which represents a node of the syntax tree.

Every time the first production  $E \rightarrow E_1 + T$  is used, its rule creates a node with  $+$  for *op* and two children,  $E_1.node$  and  $T.node$ , for the subexpressions. The second production has a similar rule.

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \mathbf{new} \text{Node}('+', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \mathbf{new} \text{Node}('-', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow ( E )$	$T.node = E.node$
5) $T \rightarrow \mathbf{id}$	$T.node = \mathbf{new} \text{Leaf}(\mathbf{id}, \mathbf{id}.entry)$
6) $T \rightarrow \mathbf{num}$	$T.node = \mathbf{new} \text{Leaf}(\mathbf{num}, \mathbf{num}.val)$

Figure 5.10: Constructing syntax trees for simple expressions

For production 3,  $E \rightarrow T$ , no node is created, since  $E.node$  is the same as  $T.node$ . Similarly, no node is created for production 4,  $T \rightarrow ( E )$ . The value of  $T.node$  is the same as  $E.node$ , since parentheses are used only for grouping; they influence the structure of the parse tree and the syntax tree, but once their job is done, there is no further need to retain them in the syntax tree.

The last two  $T$ -productions have a single terminal on the right. We use the constructor *Leaf* to create a suitable node, which becomes the value of  $T.node$ .

Figure 5.11 shows the construction of a syntax tree for the input  $a - 4 + c$ . The nodes of the syntax tree are shown as records, with the *op* field first. Syntax-tree edges are now shown as solid lines. The underlying parse tree, which need not actually be constructed, is shown with dotted edges. The third type of line, shown dashed, represents the values of  $E.node$  and  $T.node$ ; each line points to the appropriate syntax-tree node.

At the bottom we see leaves for  $a$ , 4 and  $c$ , constructed by *Leaf*. We suppose that the lexical value  $\mathbf{id}.entry$  points into the symbol table, and the lexical value  $\mathbf{num}.val$  is the numerical value of a constant. These leaves, or pointers to them, become the value of  $T.node$  at the three parse-tree nodes labeled  $T$ , according to rules 5 and 6. Note that by rule 3, the pointer to the leaf for  $a$  is also the value of  $E.node$  for the leftmost  $E$  in the parse tree.

Rule 2 causes us to create a node with *op* equal to the minus sign and pointers to the first two leaves. Then, rule 1 produces the root node of the syntax tree by combining the node for  $-$  with the third leaf.

If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown in Fig. 5.12 ends with  $p_5$  pointing to the root of the constructed syntax tree.  $\square$

With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

**Example 5.12:** The L-attributed definition in Fig. 5.13 performs the same translation as the S-attributed definition in Fig. 5.10. The attributes for the grammar symbols  $E$ ,  $T$ ,  $\mathbf{id}$ , and  $\mathbf{num}$  are as discussed in Example 5.11.

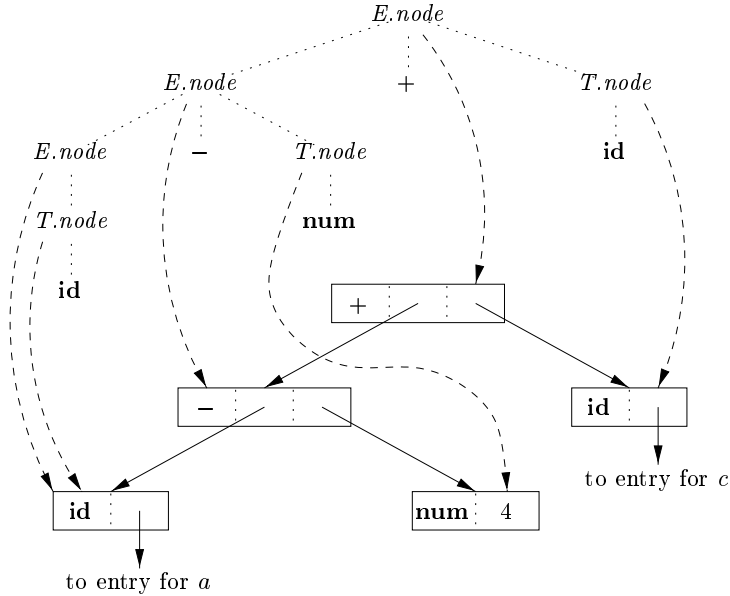


Figure 5.11: Syntax tree for  $a - 4 + c$

- 1)  $p_1 = \mathbf{new\ Leaf}(\mathbf{id}, \mathit{entry-a});$
- 2)  $p_2 = \mathbf{new\ Leaf}(\mathbf{num}, 4);$
- 3)  $p_3 = \mathbf{new\ Node}('-', p_1, p_2);$
- 4)  $p_4 = \mathbf{new\ Leaf}(\mathbf{id}, \mathit{entry-c});$
- 5)  $p_5 = \mathbf{new\ Node}('+', p_3, p_4);$

Figure 5.12: Steps in the construction of the syntax tree for  $a - 4 + c$

The rules for building syntax trees in this example are similar to the rules for the desk calculator in Example 5.3. In the desk-calculator example, a term  $x * y$  was evaluated by passing  $x$  as an inherited attribute, since  $x$  and  $* y$  appeared in different portions of the parse tree. Here, the idea is to build a syntax tree for  $x + y$  by passing  $x$  as an inherited attribute, since  $x$  and  $+ y$  appear in different subtrees. Nonterminal  $E'$  is the counterpart of nonterminal  $T'$  in Example 5.3. Compare the dependency graph for  $a - 4 + c$  in Fig. 5.14 with that for  $3 * 5$  in Fig. 5.7.

Nonterminal  $E'$  has an inherited attribute  $inh$  and a synthesized attribute  $syn$ . Attribute  $E'.inh$  represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for  $E'$ . At node 5 in the dependency graph in Fig. 5.14,  $E'.inh$  denotes the root of the partial syntax tree for the identifier  $a$ ; that is, the leaf for  $a$ . At node 6,  $E'.inh$  denotes the root for the partial syntax

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow T E'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \rightarrow + T E'_1$	$E'_1.inh = \mathbf{new Node}('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3) $E' \rightarrow - T E'_1$	$E'_1.inh = \mathbf{new Node}('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4) $E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5) $T \rightarrow ( E )$	$T.node = E.node$
6) $T \rightarrow \mathbf{id}$	$T.node = \mathbf{new Leaf}(\mathbf{id}, \mathbf{id.entry})$
7) $T \rightarrow \mathbf{num}$	$T.node = \mathbf{new Leaf}(\mathbf{num}, \mathbf{num.val})$

Figure 5.13: Constructing syntax trees during top-down parsing

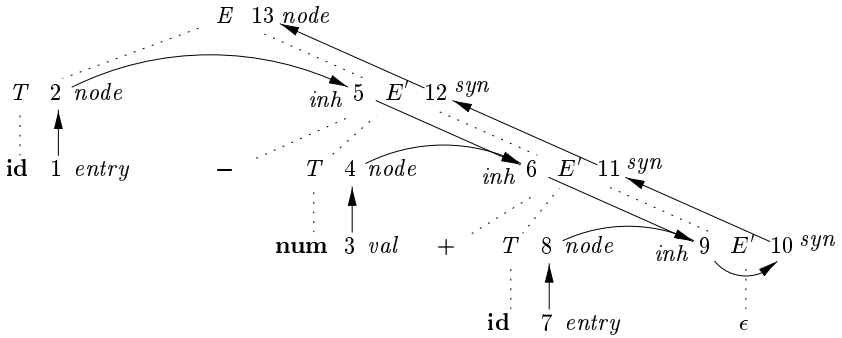


Figure 5.14: Dependency graph for  $a - 4 + c$ , with the SDD of Fig. 5.13

tree for the input  $a - 4$ . At node 9,  $E'.inh$  denotes the syntax tree for  $a - 4 + c$ .

Since there is no more input, at node 9,  $E'.inh$  points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of  $E.node$ . Specifically, the attribute value at node 10 is defined by the rule  $E'.syn = E'.inh$  associated with the production  $E' \rightarrow \epsilon$ . The attribute value at node 11 is defined by the rule  $E'.syn = E'_1.syn$  associated with production 2 in Fig. 5.13. Similar rules define the attribute values at nodes 12 and 13.  $\square$

### 5.3.2 The Structure of a Type

Inherited attributes are useful when the structure of the parse tree differs from the abstract syntax of the input; attributes can then be used to carry informa-

tion from one part of the parse tree to another. The next example shows how a mismatch in structure can be due to the design of the language, and not due to constraints imposed by the parsing method.

**Example 5.13:** In C, the type `int [2][3]` can be read as, “array of 2 arrays of 3 integers.” The corresponding type expression `array(2, array(3, integer))` is represented by the tree in Fig. 5.15. The operator `array` takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled `array` with two children for a number and a type.

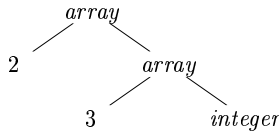


Figure 5.15: Type expression for `int[2][3]`

With the SDD in Fig. 5.16, nonterminal  $T$  generates either a basic type or an array type. Nonterminal  $B$  generates one of the basic types `int` and `float`.  $T$  generates a basic type when  $T$  derives  $BC$  and  $C$  derives  $\epsilon$ . Otherwise,  $C$  generates array components consisting of a sequence of integers, each integer surrounded by brackets.

PRODUCTION	SEMANTIC RULES
$T \rightarrow BC$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow \text{int}$	$B.t = \text{integer}$
$B \rightarrow \text{float}$	$B.t = \text{float}$
$C \rightarrow [\text{num}] C_1$	$C.t = \text{array}(\text{num.val}, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16:  $T$  generates either a basic type or an array type

The nonterminals  $B$  and  $T$  have a synthesized attribute  $t$  representing a type. The nonterminal  $C$  has two attributes: an inherited attribute  $b$  and a synthesized attribute  $t$ . The inherited  $b$  attributes pass a basic type down the tree, and the synthesized  $t$  attributes accumulate the result.

An annotated parse tree for the input string `int [2] [3]` is shown in Fig. 5.17. The corresponding type expression in Fig. 5.15 is constructed by passing the type `integer` from  $B$ , down the chain of  $C$ 's through the inherited attributes  $b$ . The array type is synthesized up the chain of  $C$ 's through the attributes  $t$ .

In more detail, at the root for  $T \rightarrow BC$ , nonterminal  $C$  inherits the type from  $B$ , using the inherited attribute  $C.b$ . At the rightmost node for  $C$ , the

production is  $C \rightarrow \epsilon$ , so  $C.t$  equals  $C.b$ . The semantic rules for the production  $C \rightarrow [\mathbf{num}] C_1$  form  $C.t$  by applying the operator *array* to the operands  $\mathbf{num.val}$  and  $C_1.t$ .  $\square$

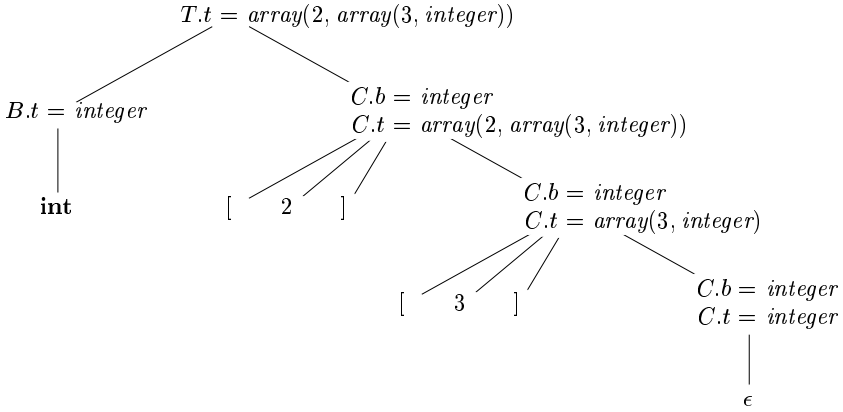


Figure 5.17: Syntax-directed translation of array types

### 5.3.3 Exercises for Section 5.3

**Exercise 5.3.1:** Below is a grammar for expressions involving operator + and integer or floating-point operands. Floating-point numbers are distinguished by having a decimal point.

$$\begin{aligned}
 E &\rightarrow E + T \mid T \\
 T &\rightarrow \mathbf{num} . \mathbf{num} \mid \mathbf{num}
 \end{aligned}$$

- a) Give an SDD to determine the type of each term  $T$  and expression  $E$ .
- b) Extend your SDD of (a) to translate expressions into postfix notation. Use the unary operator **intToFloat** to turn an integer into an equivalent float.

**! Exercise 5.3.2:** Give an SDD to translate infix expressions with + and \* into equivalent expressions without redundant parentheses. For example, since both operators associate from the left, and \* takes precedence over +,  $((a*(b+c))*(d))$  translates into  $a * (b + c) * d$ .

**! Exercise 5.3.3:** Give an SDD to differentiate expressions such as  $x * (3 * x + x * x)$  involving the operators + and \*, the variable  $x$ , and constants. Assume that no simplification occurs, so that, for example,  $3 * x$  will be translated into  $3 * 1 + 0 * x$ .

# Chapter 6

## Intermediate-Code Generation

In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code. Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. With a suitably defined intermediate representation, a compiler for language  $i$  and machine  $j$  can then be built by combining the front end for language  $i$  with the back end for machine  $j$ . This approach to creating suite of compilers can save a considerable amount of effort:  $m \times n$  compilers can be built by writing just  $m$  front ends and  $n$  back ends.

This chapter deals with intermediate representations, static type checking, and intermediate code generation. For simplicity, we assume that a compiler front end is organized as in Fig. 6.1, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. We shall use the syntax-directed formalisms of Chapters 2 and 5 to specify checking and translation. Many of the translation schemes can be implemented during either bottom-up or top-down parsing, using the techniques of Chapter 5. All schemes can be implemented by creating a syntax tree and then walking the tree.

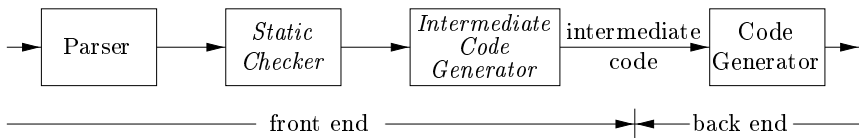


Figure 6.1: Logical structure of a compiler front end

Static checking includes *type checking*, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain

after parsing. For example, static checking assures that a break-statement in C is enclosed within a while-, for-, or switch-statement; an error is reported if such an enclosing statement does not exist.

The approach in this chapter can be used for a wide range of intermediate representations, including syntax trees and three-address code, both of which were introduced in Section 2.8. The term “three-address code” comes from instructions of the general form  $x = y \text{ op } z$  with three addresses: two for the operands  $y$  and  $z$  and one for the result  $x$ .

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as in Fig. 6.2. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking.



Figure 6.2: A compiler might use a sequence of intermediate representations

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high- to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial, as we shall see in Section 6.2.3. For looping statements, for example, a syntax tree represents the components of a statement, whereas three-address code contains labels and jump instructions to represent the flow of control, as in machine language.

The choice or design of an intermediate representation varies from compiler to compiler. An intermediate representation may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler. C is a programming language, yet it is often used as an intermediate form because it is flexible, it compiles into efficient machine code, and its compilers are widely available. The original C++ compiler consisted of a front end that generated C, treating a C compiler as a back end.

## 6.1 Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A directed acyclic graph (hereafter called a *DAG*) for an expression identifies the *common subexpressions* (subexpressions that occur more than once) of the expression. As we shall see in this section, DAG’s can be constructed by using the same techniques that construct syntax trees.



### 6.1.1 Directed Acyclic Graphs for Expressions

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators. The difference is that a node  $N$  in a DAG has more than one parent if  $N$  represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

**Example 6.1:** Figure 6.3 shows the DAG for the expression

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

The leaf for  $a$  has two parents, because  $a$  appears twice in the expression. More interestingly, the two occurrences of the common subexpression  $b-c$  are represented by one node, the node labeled  $-$ . That node has two parents, representing its two uses in the subexpressions  $a*(b-c)$  and  $(b-c)*d$ . Even though  $b$  and  $c$  appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression  $b-c$ .  $\square$

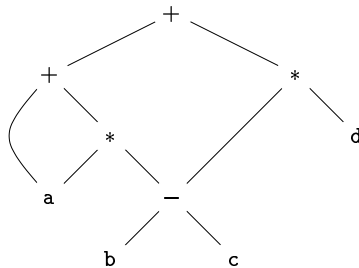


Figure 6.3: Dag for the expression  $a + a * (b - c) + (b - c) * d$

The SDD of Fig. 6.4 can construct either syntax trees or DAG's. It was used to construct syntax trees in Example 5.11, where functions *Leaf* and *Node* created a fresh node each time they were called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node,  $Node(op, left, right)$ , we check whether there is already a node with label  $op$ , and children  $left$  and  $right$ , in that order. If so, *Node* returns the existing node; otherwise, it creates a new node.

**Example 6.2:** The sequence of steps shown in Fig. 6.5 constructs the DAG in Fig. 6.3, provided *Node* and *Leaf* return an existing node, if possible, as

PRODUCTION	SEMANTIC RULES
1) $E \rightarrow E_1 + T$	$E.node = \mathbf{new} \text{Node}(' + ', E_1.node, T.node)$
2) $E \rightarrow E_1 - T$	$E.node = \mathbf{new} \text{Node}(' - ', E_1.node, T.node)$
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow ( E )$	$T.node = E.node$
5) $T \rightarrow \mathbf{id}$	$T.node = \mathbf{new} \text{Leaf}(\mathbf{id}, \mathbf{id}.entry)$
6) $T \rightarrow \mathbf{num}$	$T.node = \mathbf{new} \text{Leaf}(\mathbf{num}, \mathbf{num}.val)$

Figure 6.4: Syntax-directed definition to produce syntax trees or DAG's

- 1)  $p_1 = \text{Leaf}(\mathbf{id}, \text{entry-a})$
- 2)  $p_2 = \text{Leaf}(\mathbf{id}, \text{entry-a}) = p_1$
- 3)  $p_3 = \text{Leaf}(\mathbf{id}, \text{entry-b})$
- 4)  $p_4 = \text{Leaf}(\mathbf{id}, \text{entry-c})$
- 5)  $p_5 = \text{Node}(' - ', p_3, p_4)$
- 6)  $p_6 = \text{Node}('* ', p_1, p_5)$
- 7)  $p_7 = \text{Node}(' + ', p_1, p_6)$
- 8)  $p_8 = \text{Leaf}(\mathbf{id}, \text{entry-b}) = p_3$
- 9)  $p_9 = \text{Leaf}(\mathbf{id}, \text{entry-c}) = p_4$
- 10)  $p_{10} = \text{Node}(' - ', p_3, p_4) = p_5$
- 11)  $p_{11} = \text{Leaf}(\mathbf{id}, \text{entry-d})$
- 12)  $p_{12} = \text{Node}('* ', p_5, p_{11})$
- 13)  $p_{13} = \text{Node}(' + ', p_7, p_{12})$

Figure 6.5: Steps for constructing the DAG of Fig. 6.3

discussed above. We assume that *entry-a* points to the symbol-table entry for *a*, and similarly for the other identifiers.

When the call to *Leaf*(**id**, *entry-a*) is repeated at step 2, the node created by the previous call is returned, so  $p_2 = p_1$ . Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e.,  $p_8 = p_3$  and  $p_9 = p_4$ ). Hence the node returned at step 10 must be the same as that returned at step 5; i.e.,  $p_{10} = p_5$ .  $\square$

### 6.1.2 The Value-Number Method for Constructing DAG's

Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 6.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 6.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and

interior nodes have two additional fields indicating the left and right children.

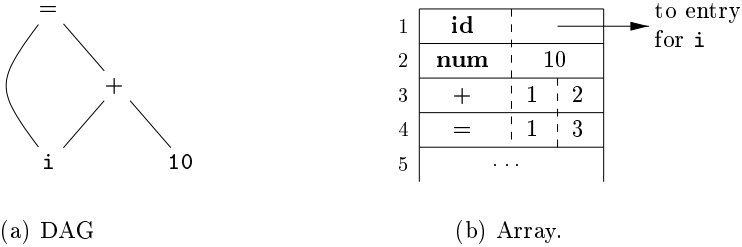


Figure 6.6: Nodes of a DAG for  $i = i + 10$  allocated in an array

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the *value number* for the node or for the expression represented by the node. For instance, in Fig. 6.6, the node labeled  $+$  has value number 3, and its left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its “value number.” If stored in an appropriate data structure, value numbers help us construct expression DAG’s efficiently; the next algorithm shows how.

Suppose that nodes are stored in an array, as in Fig. 6.6, and each node is referred to by its value number. Let the *signature* of an interior node be the triple  $\langle op, l, r \rangle$ , where  $op$  is the label,  $l$  its left child’s value number, and  $r$  its right child’s value number. A unary operator may be assumed to have  $r = 0$ .

**Algorithm 6.3:** The value-number method for constructing the nodes of a DAG.

**INPUT:** Label  $op$ , node  $l$ , and node  $r$ .

**OUTPUT:** The value number of a node in the array with signature  $\langle op, l, r \rangle$ .

**METHOD:** Search the array for a node  $M$  with label  $op$ , left child  $l$ , and right child  $r$ . If there is such a node, return the value number of  $M$ . If not, create in the array a new node  $N$  with label  $op$ , left child  $l$ , and right child  $r$ , and return its value number.  $\square$

While Algorithm 6.3 yields the desired output, searching the entire array every time we are asked to locate one node is expensive, especially if the array holds expressions from an entire program. A more efficient approach is to use a hash table, in which the nodes are put into “buckets,” each of which typically will have only a few nodes. The hash table is one of several data structures that support *dictionaries* efficiently.<sup>1</sup> A dictionary is an abstract data type that

<sup>1</sup>See Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983, for a discussion of data structures supporting dictionaries.

allows us to insert and delete elements of a set, and to determine whether a given element is currently in the set. A good data structure for dictionaries, such as a hash table, performs each of these operations in time that is constant or close to constant, independent of the size of the set.

To construct a hash table for the nodes of a DAG, we need a *hash function*  $h$  that computes the index of the bucket for a signature  $\langle op, l, r \rangle$ , in a way that distributes the signatures across buckets, so that it is unlikely that any one bucket will get much more than a fair share of the nodes. The bucket index  $h(op, l, r)$  is computed deterministically from  $op, l$ , and  $r$ , so that we may repeat the calculation and always get to the same bucket index for node  $\langle op, l, r \rangle$ .

The buckets can be implemented as linked lists, as in Fig. 6.7. An array, indexed by hash value, holds the *bucket headers*, each of which points to the first cell of a list. Within the linked list for a bucket, each cell holds the value number of one of the nodes that hash to that bucket. That is, node  $\langle op, l, r \rangle$  can be found on the list whose header is at index  $h(op, l, r)$  of the array.

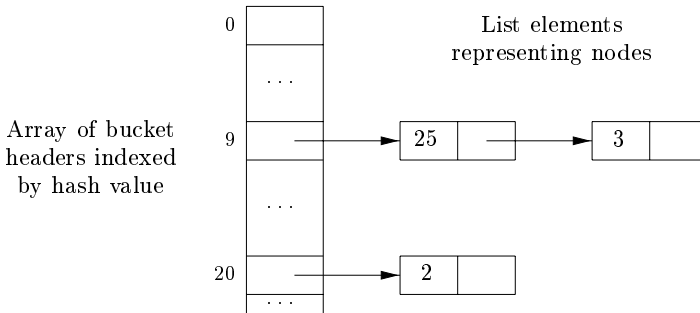


Figure 6.7: Data structure for searching buckets

Thus, given the input node  $op, l$ , and  $r$ , we compute the bucket index  $h(op, l, r)$  and search the list of cells in this bucket for the given input node. Typically, there are enough buckets so that no list has more than a few cells. We may need to look at all the cells within a bucket, however, and for each value number  $v$  found in a cell, we must check whether the signature  $\langle op, l, r \rangle$  of the input node matches the node with value number  $v$  in the list of cells (as in Fig. 6.7). If we find a match, we return  $v$ . If we find no match, we know no such node can exist in any other bucket, so we create a new cell, add it to the list of cells for bucket index  $h(op, l, r)$ , and return the value number in that new cell.

### 6.1.3 Exercises for Section 6.1

**Exercise 6.1.1:** Construct the DAG for the expression

$$((x + y) - ((x + y) * (x - y))) + ((x + y) * (x - y))$$

**Exercise 6.1.2:** Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming  $+$  associates from the left.

a)  $a + b + (a + b)$ .

b)  $a + b + a + b$ .

c)  $a + a + (a + a + a + (a + a + a + a))$ .

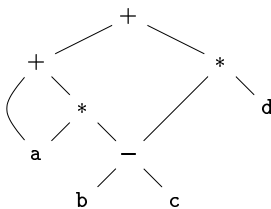
## 6.2 Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like  $x+y*z$  might be translated into the sequence of three-address instructions

$$\begin{aligned}t_1 &= y * z \\t_2 &= x + t_1\end{aligned}$$

where  $t_1$  and  $t_2$  are compiler-generated temporary names. This unraveling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization, as discussed in Chapters 8 and 9. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

**Example 6.4:** Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph. The DAG in Fig. 6.3 is repeated in Fig. 6.8, together with a corresponding three-address code sequence.  $\square$



(a) DAG

$$\begin{aligned}t_1 &= b - c \\t_2 &= a * t_1 \\t_3 &= a + t_2 \\t_4 &= t_1 * d \\t_5 &= t_3 + t_4\end{aligned}$$

(b) Three-address code

Figure 6.8: A DAG and its corresponding three-address code

## 6.2.1 Addresses and Instructions

Three-address code is built from two concepts: addresses and instructions. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses. Alternatively, three-address code can be implemented using records with fields for the addresses; records called quadruples and triples are discussed briefly in Section 6.2.2.

An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- *A constant.* In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered in Section 6.5.2.
- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

We now consider the common three-address instructions used in the rest of this book. Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by “backpatching,” discussed in Section 6.7. Here is a list of the common three-address instruction forms:

1. Assignment instructions of the form  $x = y \text{ op } z$ , where  $op$  is a binary arithmetic or logical operation, and  $x$ ,  $y$ , and  $z$  are addresses.
2. Assignments of the form  $x = op \ y$ , where  $op$  is a unary operation. Essential unary operations include unary minus, logical negation, and conversion operators that, for example, convert an integer to a floating-point number.
3. *Copy instructions* of the form  $x = y$ , where  $x$  is assigned the value of  $y$ .
4. An unconditional jump `goto L`. The three-address instruction with label  $L$  is the next to be executed.
5. Conditional jumps of the form `if x goto L` and `ifFalse x goto L`. These instructions execute the instruction with label  $L$  next if  $x$  is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.

6. Conditional jumps such as `if  $x$  relop  $y$  goto  $L$` , which apply a relational operator (`<`, `=`, `>`, etc.) to  $x$  and  $y$ , and execute the instruction with label  $L$  next if  $x$  stands in relation *relop* to  $y$ . If not, the three-address instruction following `if  $x$  relop  $y$  goto  $L$`  is executed next, in sequence.
7. Procedure calls and returns are implemented using the following instructions: `param  $x$`  for parameters; `call  $p, n$`  and  `$y$  = call  $p, n$`  for procedure and function calls, respectively; and `return  $y$` , where  $y$ , representing a returned value, is optional. Their typical use is as the sequence of three-address instructions

```

param  $x_1$ 
param  $x_2$ 
...
param  $x_n$ 
call  $p, n$ 

```

generated as part of a call of the procedure  $p(x_1, x_2, \dots, x_n)$ . The integer  $n$ , indicating the number of actual parameters in “`call  $p, n$` ,” is not redundant because calls can be nested. That is, some of the first `param` statements could be parameters of a call that comes after  $p$  returns its value; that value becomes another parameter of the later call. The implementation of procedure calls is outlined in Section 6.9.

8. Indexed copy instructions of the form  `$x$  =  $y$ [ $i$ ]` and  `$x$ [ $i$ ] =  $y$` . The instruction  `$x$  =  $y$ [ $i$ ]` sets  $x$  to the value in the location  $i$  memory units beyond location  $y$ . The instruction  `$x$ [ $i$ ] =  $y$`  sets the contents of the location  $i$  units beyond  $x$  to the value of  $y$ .
9. Address and pointer assignments of the form  `$x$  = & $y$` ,  `$x$  = * $y$` , and `* $x$  =  $y$` . The instruction  `$x$  = & $y$`  sets the  $r$ -value of  $x$  to be the location ( $l$ -value) of  $y$ .<sup>2</sup> Presumably  $y$  is a name, perhaps a temporary, that denotes an expression with an  $l$ -value such as `A[ $i$ ][ $j$ ]`, and  $x$  is a pointer name or temporary. In the instruction  `$x$  = * $y$` , presumably  $y$  is a pointer or a temporary whose  $r$ -value is a location. The  $r$ -value of  $x$  is made equal to the contents of that location. Finally, `* $x$  =  $y$`  sets the  $r$ -value of the object pointed to by  $x$  to the  $r$ -value of  $y$ .

**Example 6.5:** Consider the statement

```
do i = i+1; while (a[i] < v);
```

Two possible translations of this statement are shown in Fig. 6.9. The translation in Fig. 6.9(a) uses a symbolic label  $L$ , attached to the first instruction.

---

<sup>2</sup>From Section 2.8.3,  $l$ - and  $r$ -values are appropriate on the left and right sides of assignments, respectively.

The translation in (b) shows position numbers for the instructions, starting arbitrarily at position 100. In both translations, the last instruction is a conditional jump to the first instruction. The multiplication  $i * 8$  is appropriate for an array of elements that each take 8 units of space.  $\square$

<pre>L:  t<sub>1</sub> = i + 1     i = t<sub>1</sub>     t<sub>2</sub> = i * 8     t<sub>3</sub> = a [ t<sub>2</sub> ]     if t<sub>3</sub> &lt; v goto L</pre>		<pre>100: t<sub>1</sub> = i + 1 101: i = t<sub>1</sub> 102: t<sub>2</sub> = i * 8 103: t<sub>3</sub> = a [ t<sub>2</sub> ] 104: if t<sub>3</sub> &lt; v goto 100</pre>
(a) Symbolic labels.		(b) Position numbers.

Figure 6.9: Two ways of assigning labels to three-address statements

The choice of allowable operators is an important issue in the design of an intermediate form. The operator set clearly must be rich enough to implement the operations in the source language. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front end must generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations.

## 6.2.2 Quadruples

The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure. In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are called “quadruples,” “triples,” and “indirect triples.”

A *quadruple* (or just “*quad*”) has four fields, which we call *op*, *arg<sub>1</sub>*, *arg<sub>2</sub>*, and *result*. The *op* field contains an internal code for the operator. For instance, the three-address instruction  $x = y + z$  is represented by placing  $+$  in *op*,  $y$  in *arg<sub>1</sub>*,  $z$  in *arg<sub>2</sub>*, and  $x$  in *result*. The following are some exceptions to this rule:

1. Instructions with unary operators like  $x = \text{minus } y$  or  $x = y$  do not use *arg<sub>2</sub>*. Note that for a copy statement like  $x = y$ , *op* is  $=$ , while for most other operations, the assignment operator is implied.
2. Operators like *param* use neither *arg<sub>2</sub>* nor *result*.
3. Conditional and unconditional jumps put the target label in *result*.

**Example 6.6:** Three-address code for the assignment  $a = b * -c + b * -c$ ; appears in Fig. 6.10(a). The special operator *minus* is used to distinguish the



unary minus operator, as in `-c`, from the binary minus operator, as in `b - c`. Note that the unary-minus “three-address” statement has only two addresses, as does the copy statement `a = t5`.

The quadruples in Fig. 6.10(b) implement the three-address code in (a).  $\square$

<code>t<sub>1</sub> = minus c</code>	<table border="1" style="border-collapse: collapse; text-align: center;"> <thead> <tr> <th style="padding: 2px 5px;"><i>op</i></th> <th style="padding: 2px 5px;"><i>arg<sub>1</sub></i></th> <th style="padding: 2px 5px;"><i>arg<sub>2</sub></i></th> <th style="padding: 2px 5px;"><i>result</i></th> </tr> </thead> <tbody> <tr> <td style="padding: 2px 5px;">0</td> <td style="padding: 2px 5px;">minus</td> <td style="padding: 2px 5px;">c</td> <td style="padding: 2px 5px;">t<sub>1</sub></td> </tr> <tr> <td style="padding: 2px 5px;">1</td> <td style="padding: 2px 5px;">*</td> <td style="padding: 2px 5px;">b</td> <td style="padding: 2px 5px;">t<sub>1</sub> t<sub>2</sub></td> </tr> <tr> <td style="padding: 2px 5px;">2</td> <td style="padding: 2px 5px;">minus</td> <td style="padding: 2px 5px;">c</td> <td style="padding: 2px 5px;">t<sub>3</sub></td> </tr> <tr> <td style="padding: 2px 5px;">3</td> <td style="padding: 2px 5px;">*</td> <td style="padding: 2px 5px;">b</td> <td style="padding: 2px 5px;">t<sub>3</sub> t<sub>4</sub></td> </tr> <tr> <td style="padding: 2px 5px;">4</td> <td style="padding: 2px 5px;">+</td> <td style="padding: 2px 5px;">t<sub>2</sub> t<sub>4</sub></td> <td style="padding: 2px 5px;">t<sub>5</sub></td> </tr> <tr> <td style="padding: 2px 5px;">5</td> <td style="padding: 2px 5px;">=</td> <td style="padding: 2px 5px;">t<sub>5</sub></td> <td style="padding: 2px 5px;">a</td> </tr> <tr> <td colspan="4" style="padding: 2px 5px;">...</td> </tr> </tbody> </table>	<i>op</i>	<i>arg<sub>1</sub></i>	<i>arg<sub>2</sub></i>	<i>result</i>	0	minus	c	t <sub>1</sub>	1	*	b	t <sub>1</sub> t <sub>2</sub>	2	minus	c	t <sub>3</sub>	3	*	b	t <sub>3</sub> t <sub>4</sub>	4	+	t <sub>2</sub> t <sub>4</sub>	t <sub>5</sub>	5	=	t <sub>5</sub>	a	...			
<i>op</i>	<i>arg<sub>1</sub></i>	<i>arg<sub>2</sub></i>	<i>result</i>																														
0	minus	c	t <sub>1</sub>																														
1	*	b	t <sub>1</sub> t <sub>2</sub>																														
2	minus	c	t <sub>3</sub>																														
3	*	b	t <sub>3</sub> t <sub>4</sub>																														
4	+	t <sub>2</sub> t <sub>4</sub>	t <sub>5</sub>																														
5	=	t <sub>5</sub>	a																														
...																																	

(a) Three-address code

(b) Quadruples

Figure 6.10: Three-address code and its quadruple representation

For readability, we use actual identifiers like `a`, `b`, and `c` in the fields *arg<sub>1</sub>*, *arg<sub>2</sub>*, and *result* in Fig. 6.10(b), instead of pointers to their symbol-table entries. Temporary names can either be entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class *Temp* with its own methods.

### 6.2.3 Triples

A *triple* has only three fields, which we call *op*, *arg<sub>1</sub>*, and *arg<sub>2</sub>*. Note that the *result* field in Fig. 6.10(b) is used primarily for temporary names. Using triples, we refer to the result of an operation `x op y` by its position, rather than by an explicit temporary name. Thus, instead of the temporary `t1` in Fig. 6.10(b), a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself. In Section 6.1.2, positions or pointers to positions were called value numbers.

Triples are equivalent to signatures in Algorithm 6.3. Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

**Example 6.7:** The syntax tree and triples in Fig. 6.11 correspond to the three-address code and quadruples in Fig. 6.10. In the triple representation in Fig. 6.11(b), the copy statement `a = t5` is encoded in the triple representation by placing `a` in the *arg<sub>1</sub>* field and (4) in the *arg<sub>2</sub>* field.  $\square$

A ternary operation like `x[i] = y` requires two entries in the triple structure; for example, we can put `x` and `i` in one triple and `y` in the next. Similarly, `x = y[i]` can be implemented by treating it as if it were the two instructions

### Why Do We Need Copy Instructions?

A simple algorithm for translating expressions generates copy instructions for assignments, as in Fig. 6.10(a), where we copy  $t_5$  into  $a$  rather than assigning  $t_2 + t_4$  to  $a$  directly. Each subexpression typically gets its own, new temporary to hold its result, and only when the assignment operator  $=$  is processed do we learn where to put the value of the complete expression. A code-optimization pass, perhaps using the DAG of Section 6.1.1 as an intermediate form, can discover that  $t_5$  can be replaced by  $a$ .

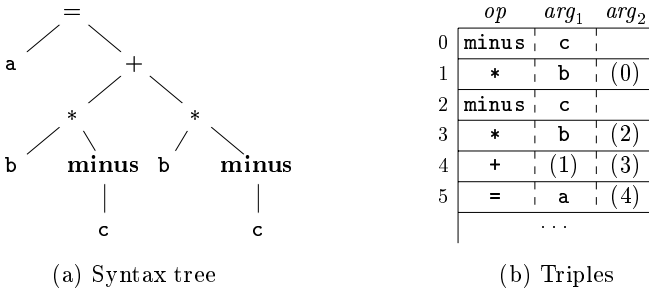


Figure 6.11: Representations of  $a = b * - c + b * - c$ ;

$t = y[i]$  and  $x = t$ , where  $t$  is a compiler-generated temporary. Note that the temporary  $t$  does not actually appear in a triple, since temporary values are referred to by their position in the triple structure.

A benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With quadruples, if we move an instruction that computes a temporary  $t$ , then the instructions that use  $t$  require no change. With triples, the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur with indirect triples, which we consider next.

*Indirect triples* consist of a listing of pointers to triples, rather than a listing of triples themselves. For example, let us use an array *instruction* to list pointers to triples in the desired order. Then, the triples in Fig. 6.11(b) might be represented as in Fig. 6.12.

With indirect triples, an optimizing compiler can move an instruction by reordering the *instruction* list, without affecting the triples themselves. When implemented in Java, an array of instruction objects is analogous to an indirect triple representation, since Java treats the array elements as references to objects.

<i>instruction</i>		<i>op</i>	<i>arg<sub>1</sub></i>	<i>arg<sub>2</sub></i>
35	(0)	0	minus	c
36	(1)	1	*	b (0)
37	(2)	2	minus	c
38	(3)	3	*	b (2)
39	(4)	4	+	(1) (3)
40	(5)	5	=	a (4)
	...			...

Figure 6.12: Indirect triples representation of three-address code

### 6.2.4 Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term *static single-assignment*. Figure 6.13 shows the same intermediate program in three-address code and in static single-assignment form. Note that subscripts distinguish each definition of variables  $p$  and  $q$  in the SSA representation.

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

(a) Three-address code.      (b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

The same variable may be defined in two different control-flow paths in a program. For example, the source program

```
if ( flag ) x = -1; else x = 1;
y = x * a;
```

has two control-flow paths in which the variable  $x$  gets defined. If we use different names for  $x$  in the true part and the false part of the conditional statement, then which name should we use in the assignment  $y = x * a$ ? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the  $\phi$ -function to combine the two definitions of  $x$ :

```
if ( flag ) x1 = -1; else x2 = 1;
x3 =  $\phi(x_1, x_2)$ ;
```

Here,  $\phi(x_1, x_2)$  has the value  $x_1$  if the control flow passes through the true part of the conditional and the value  $x_2$  if the control flow passes through the false part. That is to say, the  $\phi$ -function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the  $\phi$ -function.

### 6.2.5 Exercises for Section 6.2

**Exercise 6.2.1:** Translate the arithmetic expression  $a + -(b + c)$  into:

- a) A syntax tree.
- b) Quadruples.
- c) Triples.
- d) Indirect triples.

**Exercise 6.2.2:** Repeat Exercise 6.2.1 for the following assignment statements:

- i.*  $a = b[i] + c[j].$
- ii.*  $a[i] = b*c - b*d.$
- iii.*  $x = f(y+1) + 2.$
- iv.*  $x = *p + \&y.$

**! Exercise 6.2.3:** Show how to transform a three-address code sequence into one in which each defined variable gets a unique variable name.

## 6.3 Types and Declarations

The applications of types can be grouped under checking and translation:

- *Type checking* uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator. For example, the  $\&\&$  operator in Java expects its two operands to be booleans; the result is also of type boolean.
- *Translation Applications.* From the type of a name, a compiler can determine the storage that will be needed for that name at run time. Type information is also needed to calculate the address denoted by an array reference, to insert explicit type conversions, and to choose the right version of an arithmetic operator, among other things.

# Chapter 8

## Code Generation

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig. 8.1.

The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the *back end*, may make multiple passes over the IR before generating the target program. Code optimization is discussed in detail in Chapter 9. The techniques presented in this chapter can be used whether or not an optimization phase occurs before code generation.

A code generator has three primary tasks: instruction selection, register

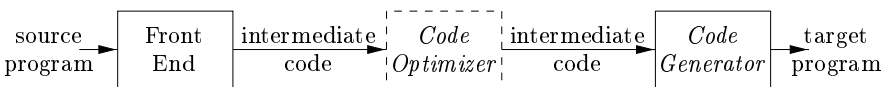


Figure 8.1: Position of code generator

allocation and assignment, and instruction ordering. The importance of these tasks is outlined in Section 8.1. Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

This chapter presents algorithms that code generators can use to translate the IR into a sequence of target language instructions for simple register machines. The algorithms will be illustrated by using the machine model in Section 8.2. Chapter 10 covers the problem of code generation for complex modern machines that support a great deal of parallelism within a single instruction.

After discussing the broad issues in the design of a code generator, we show what kind of target code a compiler needs to generate to support the abstractions embodied in a typical source language. In Section 8.3, we outline implementations of static and stack allocation of data areas, and show how names in the IR can be converted into addresses in the target code.

Many code generators partition IR instructions into “basic blocks,” which consist of sequences of instructions that are always executed together. The partitioning of the IR into basic blocks is the subject of Section 8.4. The following section presents simple local transformations that can be used to transform basic blocks into modified basic blocks from which more efficient code can be generated. These transformations are a rudimentary form of code optimization, although the deeper theory of code optimization will not be taken up until Chapter 9. An example of a useful, local transformation is the discovery of common subexpressions at the level of intermediate code and the resultant replacement of arithmetic operations by simpler copy operations.

Section 8.6 presents a simple code-generation algorithm that generates code for each statement in turn, keeping operands in registers as long as possible. The output of this kind of code generator can be readily improved by peephole optimization techniques such as those discussed in the following Section 8.7.

The remaining sections explore instruction selection and register allocation.

## 8.1 Issues in the Design of a Code Generator

While the details are dependent on the specifics of the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are encountered in the design of almost all code generators.

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that a code generator might face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

### 8.1.1 Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's. Many of the algorithms in this chapter are couched in terms of the representations considered in Chapter 6: three-address code, trees, and DAG's. The techniques we discuss can be applied, however, to the other intermediate representations as well.

In this chapter, we assume that the front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers. We also assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type-conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

### 8.1.2 The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.

However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The inter-

preter provides software compatibility across multiple platforms, a major factor in the success of Java.

To overcome the high performance penalty of interpretation, which can be on the order of a factor of 10, *just-in-time* (JIT) Java compilers have been created. These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine. Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine, bypassing the Java bytecodes entirely.

Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.

Producing a relocatable machine-language program (often called an *object module*) as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

In this chapter, we shall use a very simple RISC-like computer as our target machine. We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines. For readability, we use assembly code as the target language. As long as addresses can be calculated from offsets and other information stored in the symbol table, the code generator can produce relocatable or absolute addresses for names just as easily as symbolic addresses.

### 8.1.3 Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs



further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form  $x = y + z$ , where  $x$ ,  $y$ , and  $z$  are statically allocated, can be translated into the code sequence

```
LD  RO, y          // RO = y          (load y into register RO)
ADD RO, RO, z      // RO = RO + z      (add z to RO)
ST  x, RO          // x = RO          (store RO into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD  RO, b          // RO = b
ADD RO, RO, c      // RO = RO + c
ST  a, RO          // a = RO
LD  RO, a          // RO = a
ADD RO, RO, e      // RO = RO + e
ST  d, RO          // d = RO
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if  $a$  is not subsequently used.

The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

For example, if the target machine has an “increment” instruction (INC), then the three-address statement  $a = a + 1$  may be implemented more efficiently by the single instruction `INC a`, rather than by a more obvious sequence that loads  $a$  into a register, adds one to the register, and then stores the result back into  $a$ :

```
LD  R0, a      // R0 = a
ADD R0, R0, #1 // R0 = R0 + 1
ST  a, R0      // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine-code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

In Section 8.9 we shall see that instruction selection can be modeled as a tree-pattern matching process in which we represent the IR and the machine instructions as trees. We then attempt to “tile” an IR tree with a set of subtrees that correspond to machine instructions. If we associate a cost with each machine-instruction subtree, we can use dynamic programming to generate optimal code sequences. Dynamic programming is discussed in Section 8.11.

### 8.1.4 Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two subproblems:

1. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.
2. *Register assignment*, during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

**Example 8.1:** Certain machines require *register-pairs* (an even and next odd-numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

```
M x, y
```

where  $x$ , the multiplicand, is the odd register of an even/odd register pair and  $y$ , the multiplier, can be anywhere. The product occupies the entire even/odd register pair. The division instruction is of the form

D  $x, y$

where the dividend occupies an even/odd register pair whose even register is  $x$ ; the divisor is  $y$ . After division, the even register holds the remainder and the odd register the quotient.

Now, consider the two three-address code sequences in Fig. 8.2 in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 8.3.

$t = a + b$ $t = t * c$ $t = t / d$	$t = a + b$ $t = t + c$ $t = t / d$
(a)	(b)

Figure 8.2: Two three-address code sequences

L R1, a A R1, b M R0, c D R0, d ST R1, t	L R0, a A R0, b A R0, c SRDA R0, 32 D R0, d ST R1, t
(a)	(b)

Figure 8.3: Optimal machine-code sequences

$R_i$  stands for register  $i$ . SRDA stands for Shift-Right-Double-Arithmetic and SRDA R0, 32 shifts the dividend into R1 and clears R0 so all bits equal its sign bit. L, ST, and A stand for load, store, and add, respectively. Note that the optimal choice for the register into which  $a$  is to be loaded depends on what will ultimately happen to  $t$ .  $\square$

Strategies for register allocation and assignment are discussed in Section 8.8. Section 8.10 shows that for certain classes of machines we can construct code sequences that evaluate expressions using as few registers as possible.

### 8.1.5 Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid

the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator. In Chapter 10, we shall study code scheduling for pipelined machines that can execute several operations in a single clock cycle.

## 8.2 The Target Language

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine. In this chapter, we shall use as a target language assembly code for a simple computer that is representative of many register machines. However, the code-generation techniques presented in this chapter can be used on many other classes of machines as well.

### 8.2.1 A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with  $n$  general-purpose registers,  $R_0, R_1, \dots, R_{n-1}$ . A full-fledged assembly language would have scores of instructions. To avoid hiding the concepts in a myriad of details, we shall use a very limited set of instructions and assume that all operands are integers. Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction. We assume the following kinds of instructions are available:

- *Load* operations: The instruction `LD dst, addr` loads the value in location *addr* into location *dst*. This instruction denotes the assignment  $dst = addr$ . The most common form of this instruction is `LD r, x` which loads the value in location *x* into register *r*. An instruction of the form `LD r1, r2` is a *register-to-register copy* in which the contents of register *r*<sub>2</sub> are copied into register *r*<sub>1</sub>.
- *Store* operations: The instruction `ST x, r` stores the value in register *r* into the location *x*. This instruction denotes the assignment  $x = r$ .
- *Computation* operations of the form `OP dst, src1, src2`, where *OP* is a operator like `ADD` or `SUB`, and *dst*, *src*<sub>1</sub>, and *src*<sub>2</sub> are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP* to the values in locations *src*<sub>1</sub> and *src*<sub>2</sub>, and place the result of this operation in location *dst*. For example, `SUB r1, r2, r3` computes  $r_1 = r_2 - r_3$ . Any value formerly stored in *r*<sub>1</sub> is lost, but if *r*<sub>1</sub> is *r*<sub>2</sub> or *r*<sub>3</sub>, the old value is read first. Unary operators that take only one operand do not have a *src*<sub>2</sub>.

- *Unconditional jumps*: The instruction `BR L` causes control to branch to the machine instruction with label  $L$ . (BR stands for *branch*.)
- *Conditional jumps* of the form `Bcond r, L`, where  $r$  is a register,  $L$  is a label, and *cond* stands for any of the common tests on values in the register  $r$ . For example, `BLTZ r, L` causes a jump to label  $L$  if the value in register  $r$  is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a variable name  $x$  referring to the memory location that is reserved for  $x$  (that is, the  $l$ -value of  $x$ ).
- A location can also be an indexed address of the form  $a(r)$ , where  $a$  is a variable and  $r$  is a register. The memory location denoted by  $a(r)$  is computed by taking the  $l$ -value of  $a$  and adding to it the value in register  $r$ . For example, the instruction `LD R1, a(R2)` has the effect of setting  $R1 = contents(a + contents(R2))$ , where  $contents(x)$  denotes the contents of the register or memory location represented by  $x$ . This addressing mode is useful for accessing arrays, where  $a$  is the base address of the array (that is, the address of the first element), and  $r$  holds the number of bytes past that address we wish to go to reach one of the elements of array  $a$ .
- A memory location can be an integer indexed by a register. For example, `LD R1, 100(R2)` has the effect of setting  $R1 = contents(100 + contents(R2))$ , that is, of loading into  $R1$  the value in the memory location obtained by adding 100 to the contents of register  $R2$ . This feature is useful for following pointers, as we shall see in the example below.
- We also allow two indirect addressing modes:  $*r$  means the memory location found in the location represented by the contents of register  $r$  and  $*100(r)$  means the memory location found in the location obtained by adding 100 to the contents of  $r$ . For example, `LD R1, *100(R2)` has the effect of setting  $R1 = contents(contents(100 + contents(R2)))$ , that is, of loading into  $R1$  the value in the memory location stored in the memory location obtained by adding 100 to the contents of register  $R2$ .
- Finally, we allow an immediate constant addressing mode. The constant is prefixed by `#`. The instruction `LD R1, #100` loads the integer 100 into register  $R1$ , and `ADD R1, R1, #100` adds the integer 100 into register  $R1$ .

Comments at the end of instructions are preceded by `//`.

**Example 8.2:** The three-address statement  $x = y - z$  can be implemented by the machine instructions:

```

LD  R1, y           // R1 = y
LD  R2, z           // R2 = z
SUB R1, R1, R2      // R1 = R1 - R2
ST  x, R1           // x = R1

```

We can do better, perhaps. One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible. For example,  $y$  and/or  $z$  may have been computed in a register, and if so we can avoid the LD step(s). Likewise, we might be able to avoid ever storing  $x$  if its value is used within the register set and is not subsequently needed.

Suppose  $a$  is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of  $a$  are indexed starting at 0. We may execute the three-address instruction  $b = a[i]$  by the machine instructions:

```

LD  R1, i           // R1 = i
MUL R1, R1, 8       // R1 = R1 * 8
LD  R2, a(R1)       // R2 = contents(a + contents(R1))
ST  b, R2           // b = R2

```

That is, the second step computes  $8i$ , and the third step places in register  $R2$  the value in the  $i$ th element of  $a$  — the one found in the location that is  $8i$  bytes past the base address of the array  $a$ .

Similarly, the assignment into the array  $a$  represented by three-address instruction  $a[j] = c$  is implemented by:

```

LD  R1, c           // R1 = c
LD  R2, j           // R2 = j
MUL R2, R2, 8       // R2 = R2 * 8
ST  a(R2), R1       // contents(a + contents(R2)) = R1

```

To implement a simple pointer indirection, such as the three-address statement  $x = *p$ , we can use machine instructions like:

```

LD  R1, p           // R1 = p
LD  R2, 0(R1)       // R2 = contents(0 + contents(R1))
ST  x, R2           // x = R2

```

The assignment through a pointer  $*p = y$  is similarly implemented in machine code by:

```

LD  R1, p           // R1 = p
LD  R2, y           // R2 = y
ST  0(R1), R2       // contents(0 + contents(R1)) = R2

```

Finally, consider a conditional-jump three-address instruction like

```

if x < y goto L

```

The machine-code equivalent would be something like:

```
LD   R1, x           // R1 = x
LD   R2, y           // R2 = y
SUB  R1, R1, R2      // R1 = R1 - R2
BLTZ R1, M           // if R1 < 0 jump to M
```

Here, *M* is the label that represents the first machine instruction generated from the three-address instruction that has label *L*. As for any three-address instruction, we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored. □

### 8.2.2 Program and Instruction Costs

We often associate a cost with compiling and running a program. Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given source program is an undecidable problem in general, and many of the subproblems involved are NP-hard. As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

For the remainder of this chapter, we shall assume each target-language instruction has an associated cost. For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction. Some examples:

- The instruction `LD R0, R1` copies the contents of register *R1* into register *R0*. This instruction has a cost of one because no additional memory words are required.
- The instruction `LD R0, M` loads the contents of memory location *M* into register *R0*. The cost is two since the address of memory location *M* is in the word following the instruction.
- The instruction `LD R1, *100(R2)` loads into register *R1* the value given by *contents(contents(100 + contents(R2)))*. The cost is two because the constant 100 is stored in the word following the instruction.

In this chapter we assume the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input. Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs. We shall see that in some situations we can actually generate optimal code for expressions on certain classes of register machines.

### 8.2.3 Exercises for Section 8.2

**Exercise 8.2.1:** Generate code for the following three-address statements assuming all variables are stored in memory locations.

- a)  $x = 1$
- b)  $x = a$
- c)  $x = a + 1$
- d)  $x = a + b$
- e) The two statements

$$\begin{aligned}x &= b * c \\y &= a + x\end{aligned}$$

**Exercise 8.2.2:** Generate code for the following three-address statements assuming  $a$  and  $b$  are arrays whose elements are 4-byte values.

- a) The four-statement sequence

$$\begin{aligned}x &= a[i] \\y &= b[j] \\a[i] &= y \\b[j] &= x\end{aligned}$$

- b) The three-statement sequence

$$\begin{aligned}x &= a[i] \\y &= b[i] \\z &= x * y\end{aligned}$$

- c) The three-statement sequence

$$\begin{aligned}x &= a[i] \\y &= b[x] \\a[i] &= y\end{aligned}$$