

Ambo University @ Woliso Campus
School of Technology and Informatics

Computer Science Department



Compiler Design Course Full hondout

Program: **Regular** Batch: **3rd** Semester: **II**

Course Code: **CoSc 3112**

*Prepared and Compiled by: **Yoobsan Bechera (BSc)***

May/25/2020, AU, Oromia, Ethiopia

Table of Contents

Chapter 1 Introduction to Compiler Design.....	1
What is a compiler?	1
Programs related to compilers.....	1
The translation process	5
Analysis (front end)	5
Synthesis (back end)	5
Grouping of phases	10
Major Data and Structures in a Compiler	11
Compiler construction tools	12
Review Exercise.....	13
Chapter 2.....	14
Lexical analysis.....	14
Introduction.....	14
Token, pattern, lexeme.....	14
Specification of patterns using regular expressions.....	15
Regular expression: Language Operations.....	16
Regular expressions for tokens	18
Automata.....	20
Deterministic Finite Automata (DFA)	24
From regular expression to an NFA.....	28
From an NFA to a DFA (subset construction algorithm)	29
The Lexical- Analyzer Generator: Lex	31
Summary of Chapter 2	32
Review Exercise.....	34
Chapter – 3.....	35
Syntax analysis.....	35
Introduction.....	35
Context free grammar (CFG).....	36
Derivation	37

Parse tree.....	37
Elimination of ambiguity Precedence/Association	38
Left Recursion.....	39
Left factoring	40
Top-down parsing	41
Non-recursive predictive parsing	43
FIRST and FOLLOW	45
LL (1) Grammars... ..	48
Bottom-Up and Top-Down Parsers.....	49
The Parser Generator: Yacc	61
Review Exercise.....	63
CHAPTER 4 Syntax-Directed Translation	65
Introduction.....	65
Syntax-Directed Definitions and Translation Schemes	65
Annotated Parse Tree	67
Annotating a Parse Tree with Depth-First Traversals.....	67
Dependency Graphs for Attributed Parse Trees	69
Evaluation Order.....	72
S-Attributed Definitions.....	72
Bottom-Up Evaluation of S-Attributed Definitions.....	74
Canonical LR(0) Collection for The Grammar	75
CHAPTER 5 Type checking.....	76
Position of type checker.....	76
Static versus Dynamic Checking	76
Type systems.....	80
Specification of a simple type checker	83
A Simple Language example	83
CHAPTER 6 Intermediate Code Generation.....	88
Intermediate Representations	88
Types of Intermediate Representations.....	89
Intermediate languages	89
Syntax-Directed Translation of Abstract Syntax Trees	90

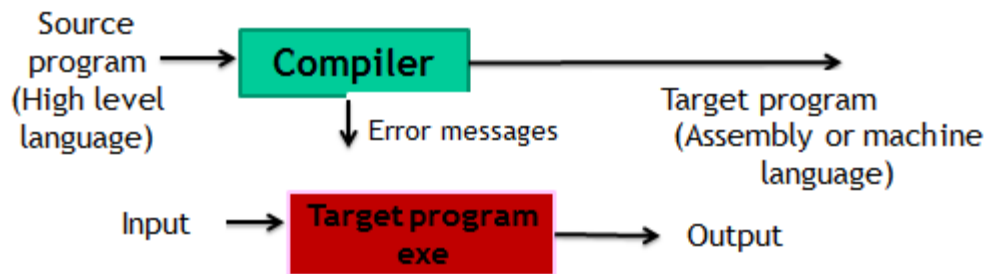
Abstract Syntax Trees versus DAGs.....	91
Stack Machine Code	91
Three-Address Statements	93
Syntax-Directed Translation into Three-Address Code	95
Implementation of Three-Address Statements.....	97
Implementation of Three-Address Statements: Quads	97
Implementation of Three-Address Statements: Triples	98
More triplet representations	98
Implementation of Three-Address Statements: Indirect Triples	98
CHAPTER 7 Code Generation	103
Introduction.....	103
Issue in the Design of a Code Generator.....	103
The Target Machine: Addressing Modes.....	108
Program and Instruction Costs.....	110

Chapter 1

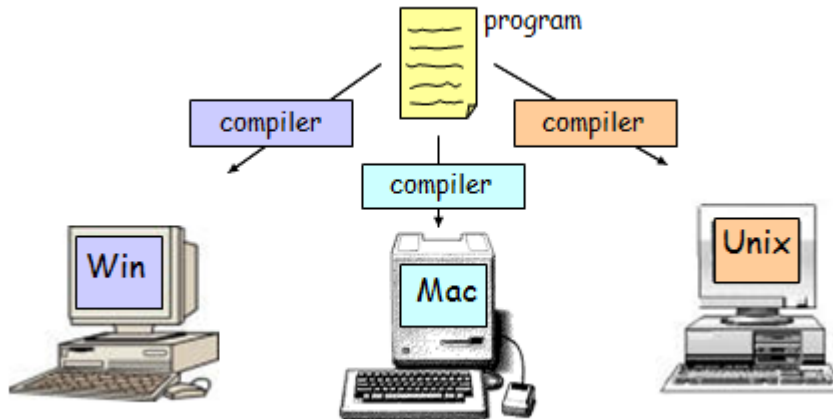
Introduction to Compiler Design

What is a compiler?

- A program that reads a program written in one language (the source language) and translates it into an equivalent program in another language (the target language).
- Why we design compiler?
- Why we study compiler construction techniques?
 - ✚ Compilers provide an essential interface between applications and architectures
 - ✚ Compilers embody a wide range of theoretical techniques



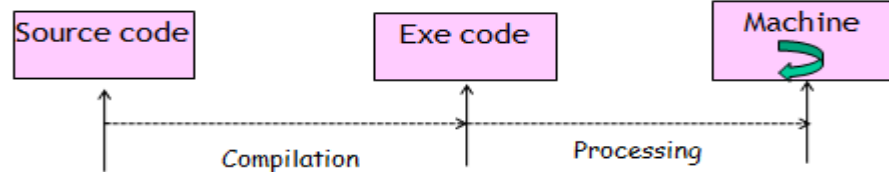
- ✚ Since different *platforms*, or hardware architectures along with the operating systems (Windows, Macs, Unix), require different machine code, you must compile most programs separately for each platform.



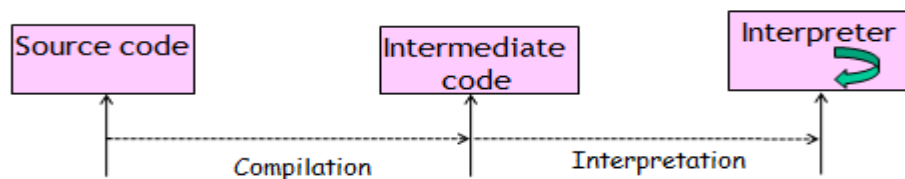
Programs related to compilers

- **Interpreter:**
 - Is a program that reads a source program and executes it
 - Works by analyzing and executing the source program commands *one at a time*

- Does not translate the whole source program into object code
- Interpretation is important when:
 - ✚ Programmer is working in interactive mode and needs to view and update variables
 - ✚ Running speed is not important
 - ✚ Commands have simple formats, and thus can be quickly analyzed and executed
 - ✚ Modification or addition to user programs is required as execution proceeds



a) Compiler



b) Interpreter

Interpreter and compiler differences

➤ Interpreter:

- Interpreter takes one statement then translates it and executes it and then takes another statement.
- Interpreter will stop the translation after it gets the first error.
- Interpreter takes less time to analyze the source code.
- Overall execution speed is less.

➤ Compiler:

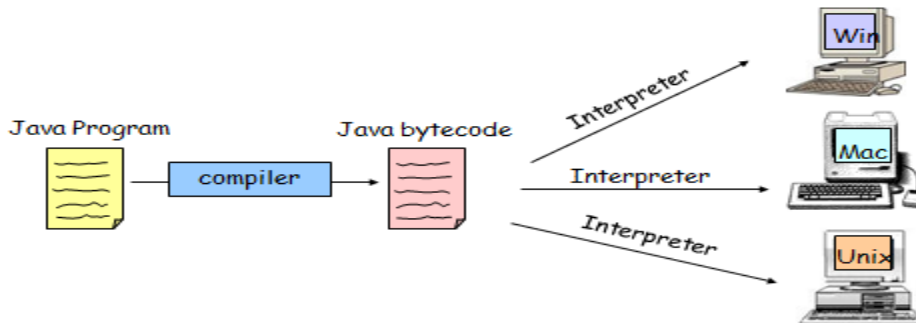
- While compiler translates the entire program in one go and then executes it.
- Generates the error report after the translation of the entire program.
- Takes a large amount of time in analyzing and processing the high level language code.
- Overall execution time is faster.

➤ **Interpreter:**

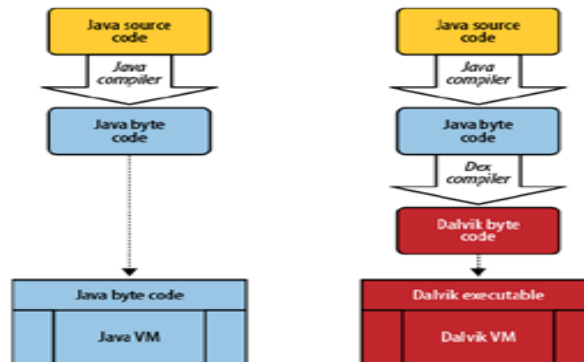
- ✚ Well-known examples of interpreters:
 - Basic interpreter, Lisp interpreter, UNIX shell command interpreter, SQL interpreter, java interpreter...
- ✚ In principle, any programming language can be either interpreted or compiled:
 - Some languages are designed to be interpreted, others are designed to be compiled
- ✚ Interpreters involve large overheads:
 - Execution speed degradation can vary from 10:1 to 100:1
 - Substantial space overhead may be involved

E.g., Compiling Java Programs

- ✓ The Java compiler produces *bytecode* not machine code
- ✓ Bytecode is converted into machine code using a *Java Interpreter*
- ✓ You can run bytecode on any computer that has a Java Interpreter installed



Android and Java



➤ **Assemblers:**

- ✚ Translator for the assembly language.
- ✚ Assembly code is translated into machine code
- ✚ Output is relocatable machine code.

➤ **Linker**

- ✚ Links object files separately compiled or assembled
- ✚ Links object files to standard library functions
- ✚ Generates a file that can be loaded and executed

➤ **Loader**

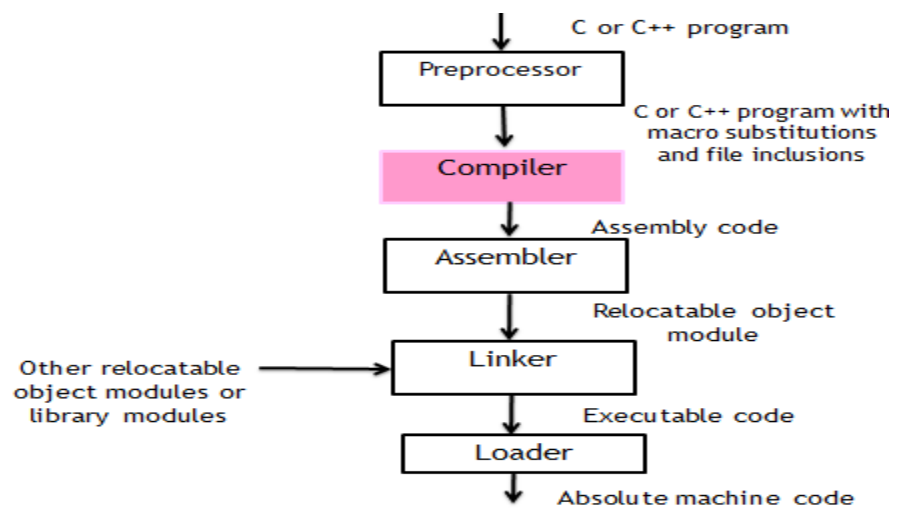
- ✚ Loading of the executable codes, which are the outputs of linker, into main memory.

➤ **Pre-processors**

- ✚ A pre-processor is a separate program that is called by the compiler before actual translation begins.

- ✚ Such a pre-processor:

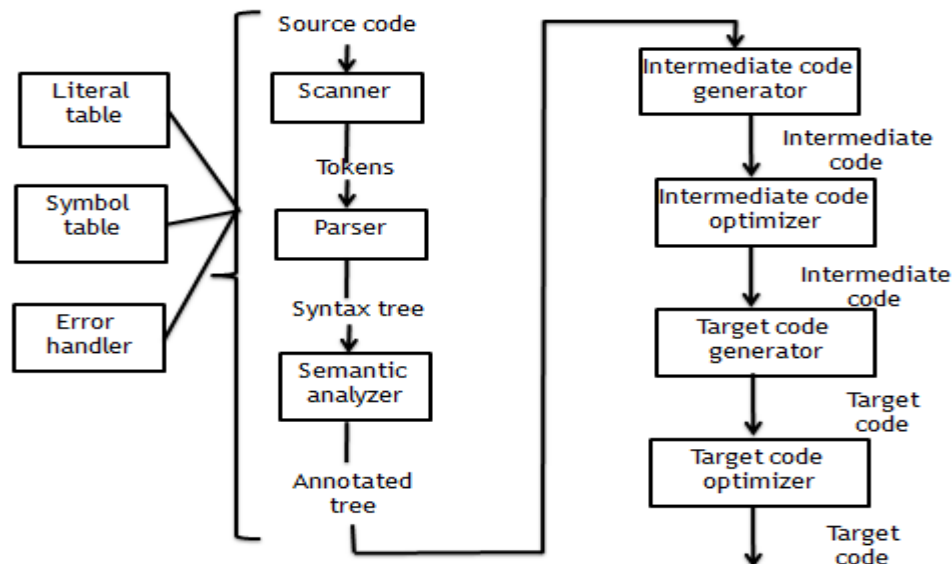
- Produce input to a compiler
- can delete comments,
- Macro processing (substitutions)
- Include other files...



The translation process

- ✚ A compiler consists of internally of a number of steps, or phases, that perform distinct logical operations.
- ✚ The phases of a compiler are shown in the next slide, together with three auxiliary components that interact with some or all of the phases:
 - ✓ The symbol table,
 - ✓ the literal table,
 - ✓ and error handler.
- ✚ There are two important parts in compilation process:

Analysis and Synthesis



Analysis and Synthesis

Analysis (front end)

- ✚ Breaks up the source program into constituent pieces and
- ✚ Creates an intermediate representation of the source program.
- ✚ During analysis, the operations implied by the source program are determined and recorded in hierarchical structure called a *tree*.

Synthesis (back end)

- ✚ The synthesis part constructs the desired program from the intermediate representation.

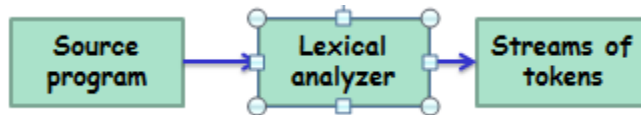
Analysis of the source program

➤ **Analysis consists of three phases:**

- 1) Linear/Lexical analysis
- 2) Hierarchical/Syntax analysis
- 3) Semantic analysis

Lexical analysis or Scanning

- ✚ The stream of characters making up the source program is read from left to right and is grouped into *tokens*.
- ✚ A **token** is a sequence of characters having a collective meaning.
- ✚ A **lexical analyzer**, also called a **lexer** or a **scanner**, receives a stream of characters from the source program and groups them into tokens.
- ✚ Examples:
 - Identifiers
 - Keywords
 - Symbols (+, -, ...)
 - Numbers ...



- ✚ Blanks, new lines, tabulation marks will be removed during lexical analysis.
- ✚ Example:

a[index] = 4 + 2;

a	identifier	
[left bracket	all are tokens
index	identifier	
]	right bracket	
=	assignment operator	
4	number	
+	plus operator	
2	number	
;	semicolon	

- ✚ A **scanner** may perform other operations along with the recognition of tokens.

- It may inter identifiers into the symbol table, and
- It may inter literals into literal table.

Lexical Analysis Tools

There are tools available to assist in the writing of lexical analyzers.

- lex - produces C source code (UNIX/linux).
- flex - produces C source code (gnu).
- JLex - produces Java source code.

We will use Lex.

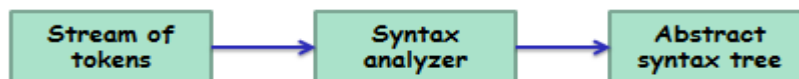
Syntax analysis or Parsing

The parser receives the source code in the form of **tokens** from the scanner and performs syntax analysis.

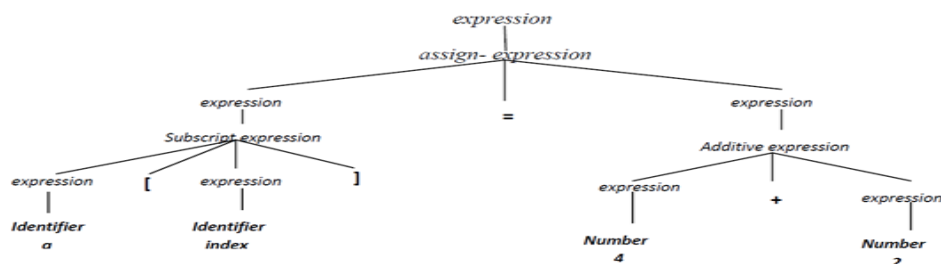
The results of syntax analysis are usually represented by a parse tree or a syntax tree.

Syntax tree → each interior node represents an operation and the children of the node represent the arguments of the operation.

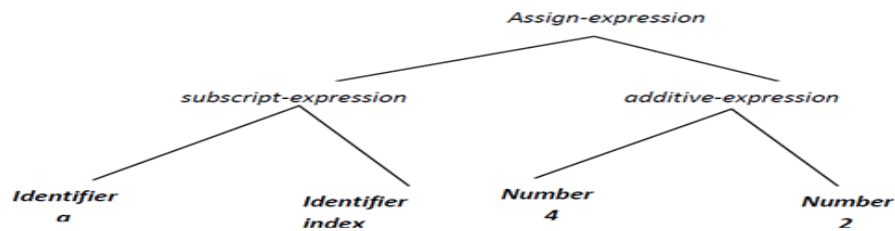
The syntactic structure of a programming language is determined by context free grammar (CFG).



Ex. Consider again the line of C code: **a[index] = 4 + 2**



Sometimes syntax trees are called **abstract syntax trees**, since they represent a further abstraction from parse trees. Example is shown in the following figure.



✚ Syntax Analysis Tools

✚ There are tools available to assist in the writing of parsers.

- yacc - produces C source code (UNIX/Linux).
- bison - produces C source code (gnu).
- CUP - produces Java source code.

✚ We will use yacc.

Semantic analysis

✚ The semantics of a program are its **meaning** as opposed to syntax or structure

✚ The semantics consist of:

- **Runtime semantics** – behavior of program at runtime
- **Static semantics** – checked by the compiler

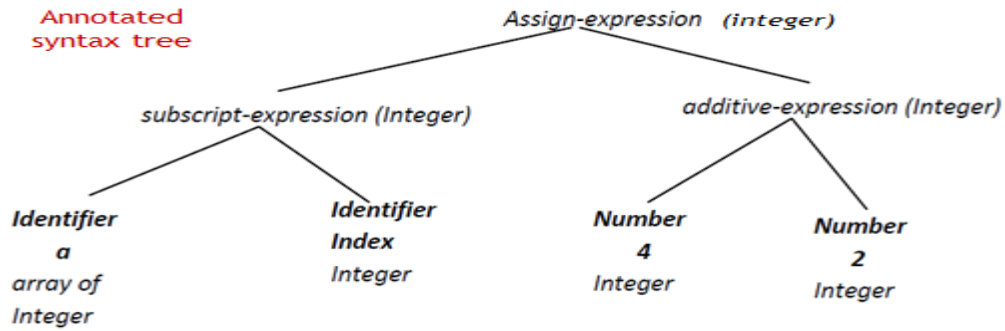
✚ Static semantics include:

- Declarations of variables and constants before use
- Calling functions that exist (predefined in a library or defined by the user)
- Passing parameters properly
- Type checking.

✚ The semantic analyzer does the following:

- Checks the static semantics of the language
- Annotates the syntax tree with type information

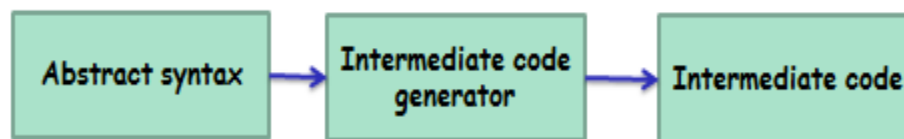
✚ Ex. Consider again the line of C code: **a[index] = 4 + 2**



- ✚ Synthesis of the target program
 - ✓ The target code generator
 - ✓ Intermediate code generator

Intermediate code generator

- ✚ Comes after syntax and semantic analysis
- ✚ Separates the compiler front end from its backend
- ✚ Intermediate representation should have 2 important properties:
 - Should be easy to produce
 - Should be easy to translate into the target program
- ✚ Intermediate representation can have a variety of forms:
 - Three-address code, P-code for an abstract machine, Tree or DAG representation



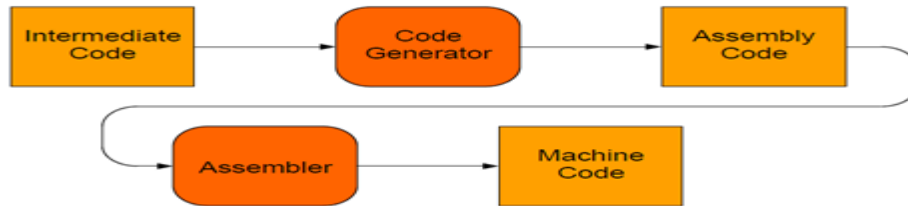
Code generator

- ✚ The machine code generator receives the (optimized) intermediate code, and then it produces either:
 - Machine code for a specific machine, or

- Assembly code for a specific machine and assembler.

Code generator

- Selects appropriate machine instructions
- Allocates memory locations for variables
- Allocates registers for intermediate computations



- ✚ The code generator takes the IR code and generates code for the target machine.
- ✚ Here we will write target code in assembly language: `a[index]=6`

```

MOV R0, index      ;; value of index -> R0
MUL R0, 2          ;; double value in R0
MOV R1, &a         ;; address of a ->R1
ADD R1, R0         ;; add R0 to R1
MOV *R1, 6         ;; constant 6 -> address in R1
  
```

- ✚ `&a` –the address of a (the base address of the array)
- ✚ `*R1`-indirect registers addressing (the last instruction stores the value 6 to the address contained in R1)

Grouping of phases

- ✚ The discussion of phases deals with the logical organization of a compiler.
- ✚ In practice most compilers are divided into:
 - **Front end** - language-specific and machine-independent.
 - **Back end** - **machine**-specific and language-independent.

Compiler passes:

- ✚ A **pass** consists of reading an input file and writing an output file.
- ✚ Several phases may be grouped in one pass.
- ✚ For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.

Single pass:

- Is a compiler that passes through the source code of each compilation unit only once
- A one-pass compiler does not "look back" at code it previously processed.
- A one-pass compilers is faster than multi-pass compilers
- They are unable to generate efficient programs, due to the limited scope available.

Multi pass:

- Is a type of compiler that processes the source code or abstract syntax tree of a program several times
- A collection of phases is done multiple times

Major Data and Structures in a Compiler

✚ Token

- Represented by an integer value or an enumeration literal
- Sometimes, it is necessary to preserve the string of characters that was scanned
- For example, name of an identifiers or value of a literal

✚ Syntax Tree

- Constructed as a pointer-based structure
- Dynamically allocated as parsing proceeds
- Nodes have fields containing information collected by the parser and semantic analyzer

✚ Symbol Table

- Keeps information associated with all kinds of tokens:
 - Identifiers, numbers, variables, functions, parameters, types, fields, etc.

- Tokens are entered by the scanner and parser
- Semantic analyzer adds type information and other attributes
- Code generation and optimization phases use the information in the symbol table

✚ Performance Issues

- Insertion, deletion, and search operations need to be efficient because they are frequent
- More than one symbol table may be used

✚ Literal Table

- Stores constant values and string literals in a program.
- One literal table applies globally to the entire program.
- Used by the code generator to:
 - Assign addresses for literals.
- Avoids the replication of constants and strings.
- Quick insertion and lookup are essential

Compiler construction tools

✚ Various tools are used in the construction of the various parts of a compiler.

✚ **Scanner generators**

- Ex. Lex, flex, JLex
- These tools generate a scanner /lexical analyzer/ if given a regular expression.

✚ **Parser Generators**

- Ex. Yacc, Bison, CUP
- These tools produce a parser /syntax analyzer/ if given a Context Free Grammar (CFG) that describes the syntax of the source language.

✚ **Syntax directed translation engines**

- Ex. Cornell Synthesizer Generator
- It produces a collection of routines that walk the parse tree and execute some tasks.

✚ Automatic code generators

- Take a collection of rules that define the translation of the IC to target code and produce a code generator.

❖ *This completes our brief description of the phases of compiler (chapter 1).*

❖ *For any unclear, comment, question, doubt things and etc please don't hesitate to announce me as you can.*

Review Exercise

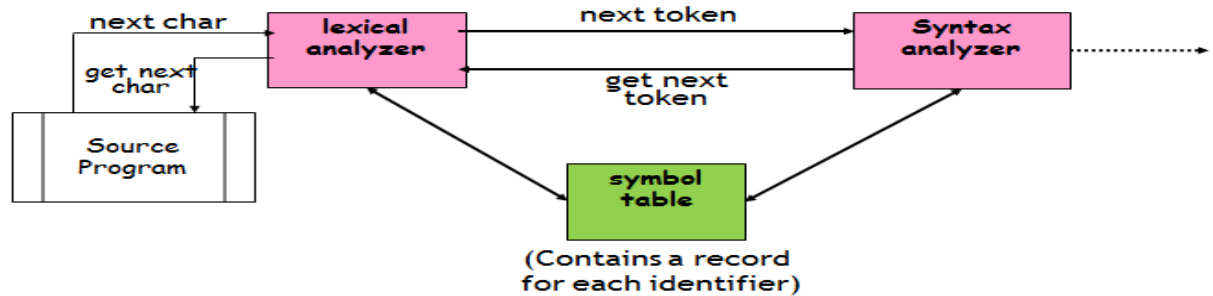
- 1) What is compiler?
- 2) Why designing it is necessary?
- 3) What are the main phases of compiler construction? Explain each.
- 4) Consider the line of C++ code: **float [index] = a-c. write its:**
 - A. Lexical analysis
 - B. Semantic analysis
 - C. Code generator
 - D. Syntax analysis
 - E. Intermediate code generator
- 5) Consider the line of C code: **int [index] = (4 + 2) / 2. write its:**
 - A. Lexical analysis
 - B. Semantic analysis
 - C. Syntax analysis
 - D. Intermediate code generator
 - E. Code generator
- 6) Explain diagrammatically how high level languages can be changed to machine understandable language (machine code).
 - a) What is the role of compilers in this action?
 - b) What is another programs used in this process and what makes them different from compilers.

Chapter 2

Lexical analysis

Introduction

- ✚ The role of lexical analyzer is:
 - to read a sequence of characters from the source program
 - group them into lexemes and
 - Produce as output a sequence of tokens for each lexeme in the source program.
- ✚ The scanner can also perform the following secondary tasks:
 - stripping out blanks, tabs, new lines
 - stripping out comments
 - keep track of line numbers (for error reporting)
- ✚ Interaction of the Lexical Analyzer with the Parser



- ✚ **token: smallest meaningful sequence of characters of interest in source program**

Token, pattern, lexeme

- ✚ A **token** is a sequence of characters from the source program having a collective meaning.
- ✚ A token is a classification of lexical units.
 - For example: **id** and **num**
- ✚ Lexemes are the specific character strings that make up a token.
 - For example: abc and 123A

✚ Patterns are rules describing the set of lexemes belonging to a token.

- For example: “*letter followed by letters and digits*”

✚ Patterns are usually specified using regular expressions. [a-zA-Z]*

✚ Example: `printf("Total = %d\n", score);`

✚ Example: The following table shows some tokens and their lexemes in Pascal (a high level, case insensitive programming language)

Token	Some lexemes	pattern
begin	Begin, Begin, BEGIN, <u>beGin...</u>	Begin in small or capital letters
if	If, IF, <u>iF</u> , If	If in small or capital letters
<u>ident</u>	Distance, F1, x, Dist1,...	Letters followed by zero or more letters and/or digits

✚ In general, in programming languages, the following are tokens:

- Keywords, operators, identifiers, constants, literals, punctuation symbols...

Specification of patterns using regular expressions

- Regular expressions
- Regular expressions for tokens

Regular expression: Definitions

✚ Represents patterns of strings of characters.

✚ An *alphabet* Σ is a finite set of symbols (characters)

✚ A *string* s is a finite sequence of symbols from Σ

- $|s|$ denotes the length of string s
- ϵ denotes the empty string, thus $|\epsilon| = 0$

✚ A language L is a specific set of strings over some fixed alphabet Σ

✚ A regular expression is one of the following:

✚ **Symbol:** a basic regular expression consisting of a single character \mathbf{a} , where \mathbf{a} is from:

- an alphabet Σ of legal characters;
- the metacharacter ϵ : or
- the metacharacter \emptyset .

✚ In the first case, $L(a)=\{a\}$; in the second case, $L(\epsilon)=\{\epsilon\}$; and in the third case, $L(\emptyset)=\{\}$.

✚ $\{\}$ – contains no string at all and $\{\epsilon\}$ – contains the single string consists of no character

✚ **Alternation:** an expression of the form $r|s$, where r and s are regular expressions.

- In this case, $L(r|s) = L(r) \cup L(s) = \{r,s\}$

✚ **Concatenation:** An expression of the form rs , where r and s are regular expressions.

- In this case, $L(rs) = L(r)L(s)=\{rs\}$

✚ **Repetition:** An expression of the form r^* , where r is a regular expression.

- In this case, $L(r^*) = L(r)^* = \{\epsilon, r, \dots\}$

Regular expression: Language Operations

✚ Union of L and M

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

✚ Concatenation of L and M

- $LM = \{xy \mid x \in L \text{ and } y \in M\}$

✚ Exponentiation of L

- $L^0 = \{\epsilon\}$; $L^i = L^{i-1}L$

✚ Kleene closure of L

- $L^* = \bigcup_{i=0, \dots, \infty} L^i$

✚ Positive closure of L

- $L^+ = \bigcup_{i=1, \dots, \infty} L^i$

Note: The following short hands are often used:

$$r^+ = r r^*$$

$$r^* = r^+ | \epsilon$$

$$r? = r | \epsilon$$

RE's: Examples

a) $L(01) = ?$

b) $L(01|0) = ?$

c) $L(0(1|0)) = ?$

- Note order of precedence of operators.

✚ $L(0^*) = ?$

✚ $L((0|10)^*(\epsilon|1)) = ?$

RE's: Examples Solution

✚ $L(01) = \{01\}$.

✚ $L(01|0) = \{01, 0\}$.

✚ $L(0(1|0)) = \{01, 00\}$.

- Note order of precedence of operators.

✚ $L(0^*) = \{\epsilon, 0, 00, 000, \dots\}$.

✚ $L((0|10)^*(\epsilon|1)) = \text{all strings of 0's and 1's without two consecutive 1's.}$

RE's: Examples (more)

1- $a | b = ?$

2- $(a|b)a = ?$

3- $(ab) | \epsilon = ?$

4- $((a|b)a)^* = ?$

❖ Reverse

1 – Even binary numbers =?

2 – An alphabet consisting of just three alphabetic characters: $\Sigma = \{a, b, c\}$. Consider the set of all strings over this alphabet that contains exactly one b.

RE's: Examples (more) Solutions

- 1- $a \mid b = \{a, b\}$
- 2- $(a|b)a = \{aa, ba\}$
- 3- $(ab) \mid \varepsilon = \{ab, \varepsilon\}$
- 4- $((a|b)a)^* = \{\varepsilon, aa, ba, aaaa, baba, \dots\}$

❖ Reverse

- 1 – Even binary numbers $(0|1)^*0$
- 2 – An alphabet consisting of just three alphabetic characters: $\Sigma = \{a, b, c\}$. Consider the set of all strings over this alphabet that contains exactly one b.
 $(a \mid c)^*b(a|c)^* \quad \{b, abc, abaca, baaaac, ccbaca, cccccb\}$

Exercises

✚ Describe the languages denoted by the following regular expressions:

- 1- $a(a|b)^*a$
- 2- $((\varepsilon|a)b^*)^*$
- 3- $(a|b)^*a(a|b)(a|b)$
- 4- $a^*ba^*ba^*ba^*$

Regular Expressions (Summary)

✚ Definition: A regular expression is a string over Σ if the following conditions hold:

- ε, \emptyset , and $a \in \Sigma$ are regular expressions
- If α and β are regular expressions, so is $\alpha\beta$
- If α and β are regular expressions, so is $\alpha+\beta$
- If α is a regular expression, so is α^*
- Nothing else is a regular expression if it doesn't follow from (1) to (4)

✚ Let α be a regular expression, the language represented by α is denoted by $L(\alpha)$.

Regular expressions for tokens

✚ Regular expressions are used to specify the patterns of tokens.

✚ Each pattern matches a set of strings. It falls into different categories:

Reserved (Key) words: They are represented by their fixed sequence of characters,

- Ex. if, while and do...

✚ If we want to collect all the reserved words into one definition, we could write it as follows:

Reserved = if | while | do |...

Special symbols: including arithmetic operators, assignment and equality such as =, :=, +, -, *

Identifiers: which are defined to be a sequence of letters and digits beginning with letter,

- we can express this in terms of regular definitions as follows:

letter = A|B|...|Z|a|b|...|z

digit = 0|1|...|9

or

letter = [a-zA-Z]

digit = [0-9]

identifiers = letter(letter|digit)*

Numbers: Numbers can be:

- sequence of digits (natural numbers), or
- decimal numbers, or
- numbers with exponent (indicated by an e or E).

✚ Example: **2.71E-2** represents the number **0.0271**.

✚ We can write regular definitions for these numbers as follows:

nat = [0-9]+

signedNat = (+/-)? *Nat*

number = *signedNat*("." *nat*)?(*E signedNat*)?

Literals or constants: This can include:

- numeric constants such as 42, and
- String literals such as “hello, world”.

relop → < / <= / = / <> / > / >=

Comments: Ex. /* this is a C comment*/

Delimiter → *newline / blank / tab / comment*

White space = (*delimiter*)⁺

✚ Example: Divide the following Java program into appropriate tokens.

```
public class Dog {  
    private String name;  
    private String color;  
    public Dog(String n, String c) {  
        name = n;  
        color = c;  
    }  
    public String getName() { return name; }  
    public String getColor() { return color; }  
    public void speak() {  
        System.out.println("Woof");  
    }  
}
```

Automata Abstract machines

Characteristics

✚ **Input:** input values (from an input alphabet Σ) are applied to the machine

✚ **Output:** outputs of the machine

✚ **States:** at any instant, the automation can be in one of the several states

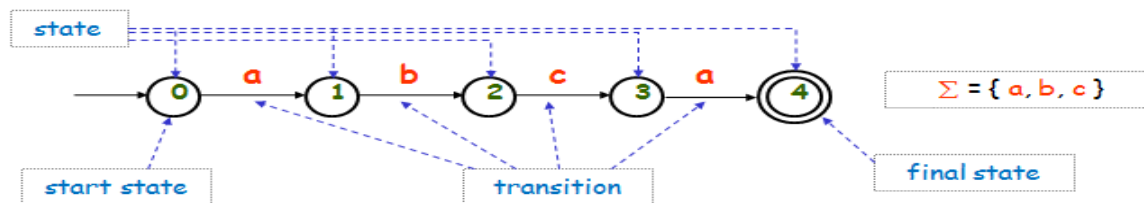
✚ **State relation:** the next state of the automation at any instant is determined by the present state and the present input

Types of automata

- Finite State Automata (FSA)
 - Deterministic FSA (DFSA)
 - Nondeterministic FSA (NFSA)
- Push Down Automata (PDA)
 - Deterministic PDA (DPDA)
 - Nondeterministic PDA (NPDA)

✚ Finite State Automaton

- Finite Automaton, Finite State Machine, FSA or FSM
- An abstract machine which can be used to implement regular expressions (etc.).
- Have a finite number of states, and a finite amount of memory (i.e., the current state).
- Can be represented by directed graphs or transition tables



• Representation

- An FSA may also be represented with a **state-transition table**. The table for the above FSA:

State	Input		
	a	b	c
0	1	∅	∅
1	∅	2	∅
2	∅	∅	3
3	4	∅	∅
4	∅	∅	∅

Design of a Lexical Analyzer/Scanner

Finite Automata

- ✚ **Lex** – turns its input program into lexical analyzer.
- ✚ Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.
- ✚ Finite automata come in two flavors:

a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges.

ϵ , the empty string, is a possible label.

b) Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

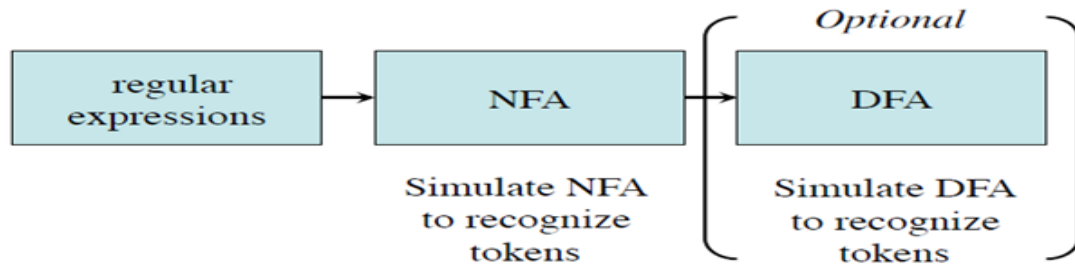
The Whole Scanner Generator Process

Overview

- ✚ Direct construction of Nondeterministic finite Automaton (NFA) to recognize a given regular expression.
 - Easy to build in an algorithmic way
 - Requires ϵ -transitions to combine regular sub expressions
- ✚ Construct a deterministic finite automaton (DFA) to simulate the NFA
 - Use a set-of-state construction
- ✚ Minimize the number of states in the DFA (**optional**)
- ✚ Generate the scanner code.

Design of a Lexical Analyzer ...

- Token → Pattern
- Pattern → Regular Expression
- Regular Expression → NFA
- NFA → DFA
- DFA's or NFA's for all tokens → **Lexical Analyzer**



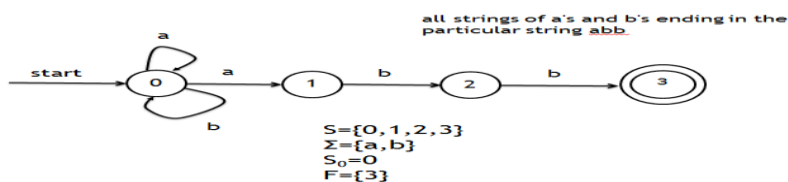
Non-Deterministic Finite Automata (NFA)

Definition:

- ✚ An NFA M consists of five tuples: (Σ, S, T, S_0, F)
 - A set of input symbols Σ , the input alphabet
 - a finite set of states' S ,
 - a transition function $T: S \times (\Sigma \cup \{ \epsilon \}) \rightarrow S$ (next state),
 - a start state S_0 from S , and
 - a set of accepting/final states F from S .
- ✚ The language accepted by M , written $L(M)$, is defined as:
- ✚ The set of strings of characters $c_1c_2\dots c_n$ with each c_i from $\Sigma \cup \{ \epsilon \}$ such that there exist states s_1 in $T(s_0, c_1)$, s_2 in $T(s_1, c_2)$, ..., s_n in $T(s_{n-1}, c_n)$ with s_n an element of F .
- ✚ It is a finite automata which has choice of edges
 - The same symbol can label edges from one state to several different states.
- ✚ An edge may be labeled by ϵ , the empty string
 - We can have transitions without any input character consumption.

Transition Graph

- ✚ The transition graph for an NFA recognizing the language of regular expression $(a|b)^*abb$



Transition Table

- The mapping T of an NFA can be represented in a *transition table*

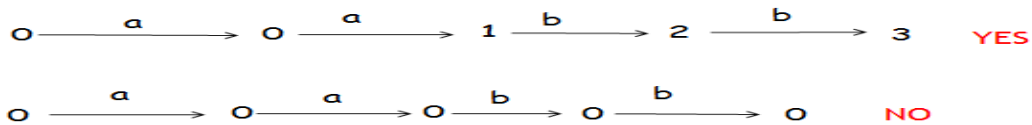
$T(0, a) = \{0, 1\}$	→
$T(0, b) = \{0\}$	
$T(1, b) = \{2\}$	
$T(2, b) = \{3\}$	

State	Input a	Input b	Input ϵ
0	{0,1}	{0}	\emptyset
1	\emptyset	{2}	\emptyset
2	\emptyset	{3}	\emptyset
3	\emptyset	\emptyset	\emptyset

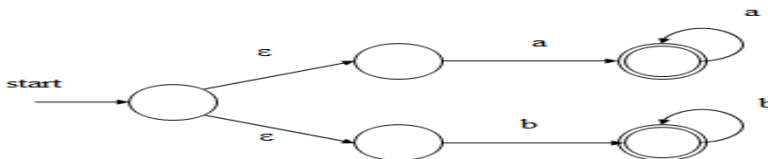
- The language defined by an NFA is the set of input strings it accepts, such as $(a|b)^*abb$ for the example NFA

Acceptance of input strings by NFA

- An NFA *accepts* input string x if and only if there is some path in the transition graph from the start state to one of the accepting states
- The string $aabb$ is accepted by the NFA:



Another NFA



- An ϵ -transition is taken without consuming any character from the input.
- What does the NFA above accept?

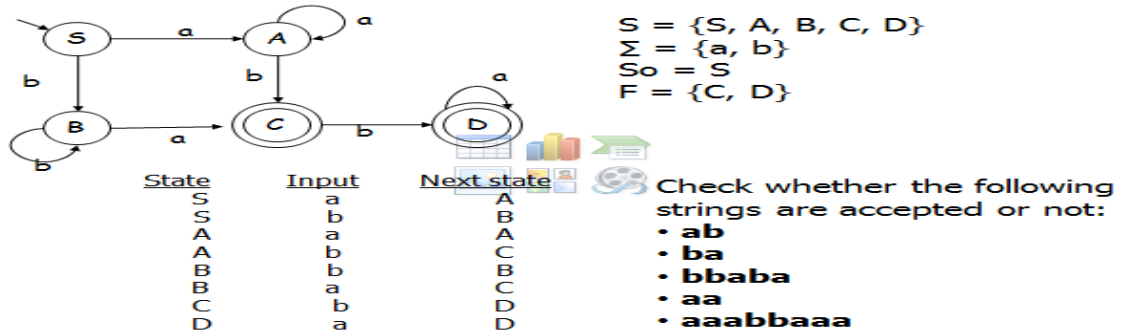
$aa^*|bb^*$

Deterministic Finite Automata (DFA)

- A *deterministic finite automaton* is a special case of an NFA
 - No state has an ϵ -transition
 - For each state S and input symbol a there is at most one edge labeled a leaving S

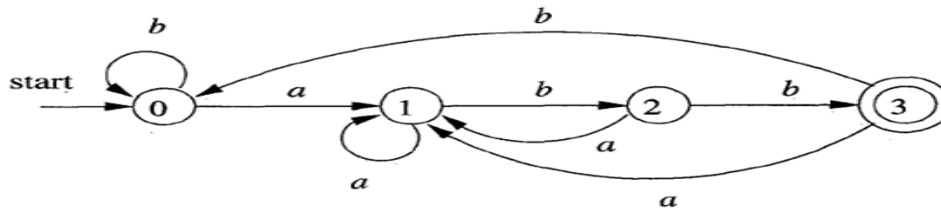
- ✚ Each entry in the transition table is a single state
 - At most one path exists to accept a string
 - Simulation algorithm is simple

DFSA: Example



DFA example

- ✚ A DFA that accepts $(a|b)^*abb$



Simulating a DFA: Algorithm

How to apply a DFA to a string?

INPUT:

- An input string x terminated by an end-of-file character eof.
- A DFA D with start state S_0 , accepting states F , and transition function $move$.

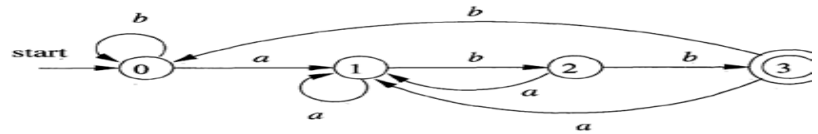
OUTPUT: Answer "yes" if D accepts x ; "no" otherwise

METHOD


- Apply the algorithm in (next slide) to the input string x .
- The function $move(s, c)$ gives the state to which there is an edge from state s on input c .
- The function $nextChar()$ returns the next character of the input string x .

Example:

```
s = s0;  
c = nextchar();  
while ( c != eof ) {  
  s = move(s, c);  
  c = nextchar();  
}  
if ( s is in F ) return  
  "yes";  
else return "no";
```



DFA accepting $(a|b)^*abb$

Given the input string **ababb**, this DFA enters the sequence of states **0,1,2,1,2,3** and returns "yes" 

DFA: Exercise

- ✚ Construct DFAs for the string matched by the following definition:

$digit = [0-9]$

$nat = digit^+$

$signednat = (+|-)?nat$

$number = signednat("."nat)?(E signedNat)?$

- ✚ Why do we study RE, NFA, DFA?

- ✚ Goal: To scan the given source program

- ✚ Process:

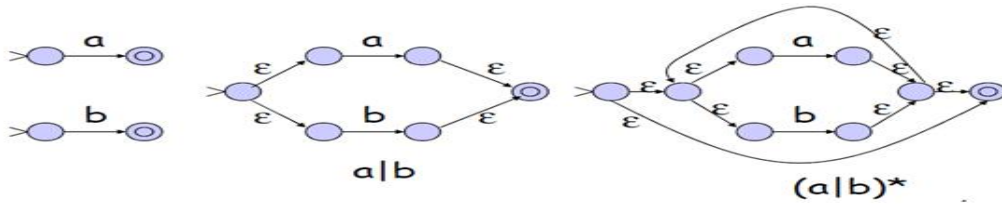
- Start with Regular Expression (RE)
- Build a DFA
 - **How?**
 - ✓ We can build a non-deterministic finite automaton, NFA (Thompson's construction)
 - ✓ Convert that to a deterministic one, DFA (Subset construction)
 - ✓ Minimize the DFA (optional) (different algorithms)
 - ✓ Implement it
 - ✓ Existing scanner generator: Lex/Flex

RE → NFA → DFA → Minimize DFA states

Step 1: Come up with a Regular Expression

(a|b)*ab

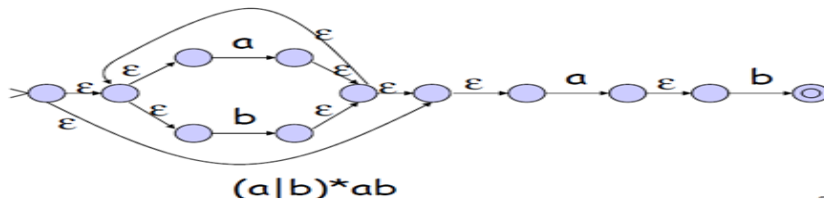
Step 2: Use Thompson's construction to create an NFA for that expression



r RE → NFA → DFA → Minimize DFA states

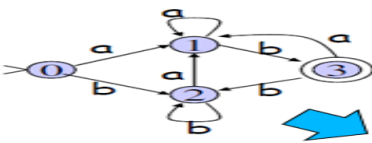
Step 1: Come up with a Regular Expression **(a|b)*ab**

Step 2: Use Thompson's construction to create an NFA for that expression



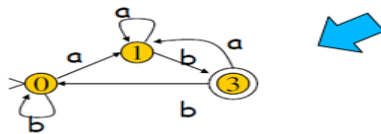
RE → NFA → DFA → Minimize DFA states

Step 3: Use subset construction to convert the NFA to a DFA



States 0 and 2 behave the same way, so they can be merged.

Step 4: Minimize the DFA states



Design of a Lexical Analyzer Generator

Two algorithms:

1- Translate a regular expression into an NFA (**Thompson's construction**)

2- Translate NFA into DFA (**Subset construction**)

From regular expression to an NFA

✚ It is known as **Thompson's construction**.

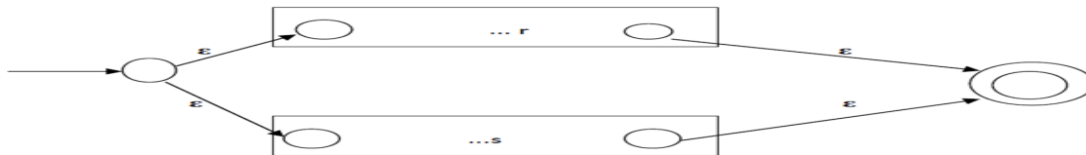
Rules:

1- For a ϵ , **a** regular expressions, construct:

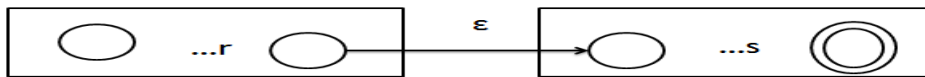


2- For a composition of regular expression:

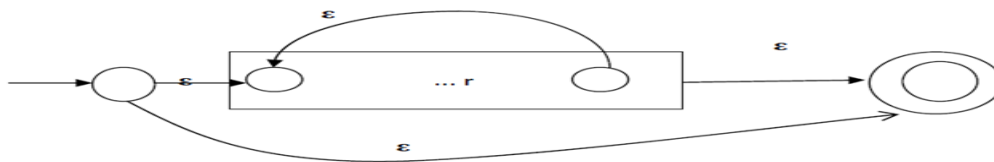
Case 1: Alternation: regular expression $(s|r)$, assume that NFAs equivalent to r and s have been constructed.



Case 2: Concatenation: regular expression sr .



Case 3: Repetition r^*



From RE to NFA: Exercises

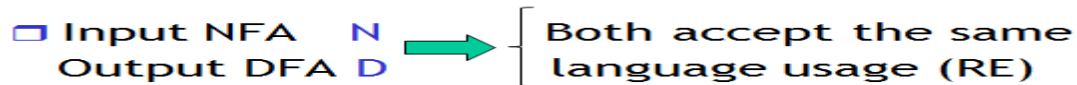
✚ Construct NFA for token identifier.

letter(letter|digit)*

✚ Construct NFA for the following regular expression:

(a|b)*abb

From an NFA to a DFA (subset construction algorithm)



Rules:

- Start state of D is assumed to be unmarked.
- Start state of D is = ϵ -closer (S_0),
- Where S_0 -start state of N .

ϵ - closure

- ϵ -closure (S') – is a set of states with the following characteristics:
 - $S' \in \epsilon$ -closure(S') itself
 - if $t \in \epsilon$ -closure (S') and if there is an edge labeled ϵ from t to v , then $v \in \epsilon$ -closure (S')
 - Repeat step 2 until no more states can be added to ϵ -closure (S').

E.g: for NFA of $(a|b)^*abb$

ϵ -closure (0)= {0, 1, 2, 4, 7} and ϵ -closure (1)= {1, 2, 4}

Algorithm

While there is unmarked state

$X = \{ s_0, s_1, s_2, \dots, s_n \}$ of D do

Begin

Mark X

For each input symbol 'a' do

Begin

Let T be the set of states to which there is a transition 'a' from state s_i in X .

$Y = \epsilon$ -Closer (T)

If Y has not been added to the set of states of D then {

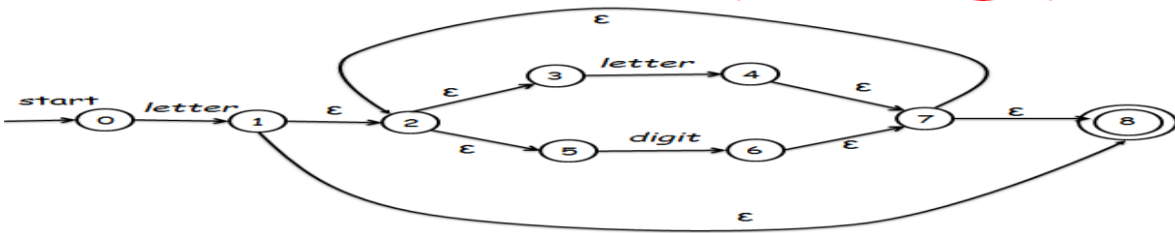
Mark Y an “Unmarked” state of D add a transition from X to Y labeled ‘a’ if not already presented

}

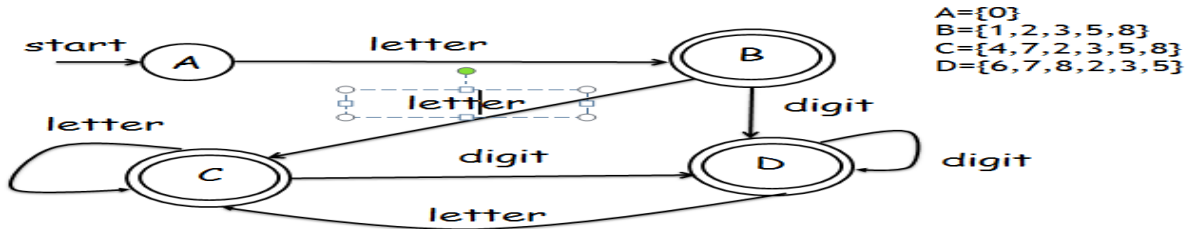
End

End

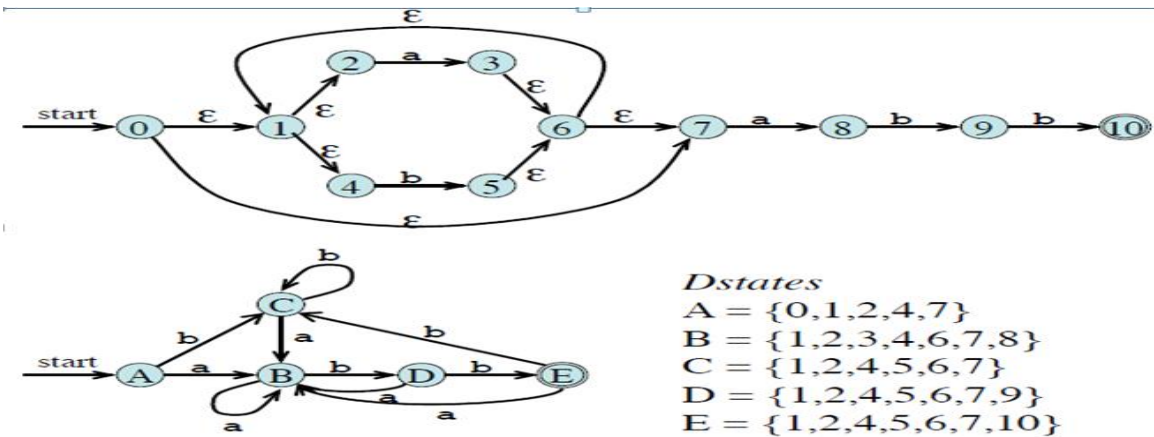
NFA for identifier: letter(letter/digit)*



Example: Convert the following NFA into the corresponding DFA. letter (letter/digit)*



Exercise: convert NFA of (a|b)*abb in to DFA.



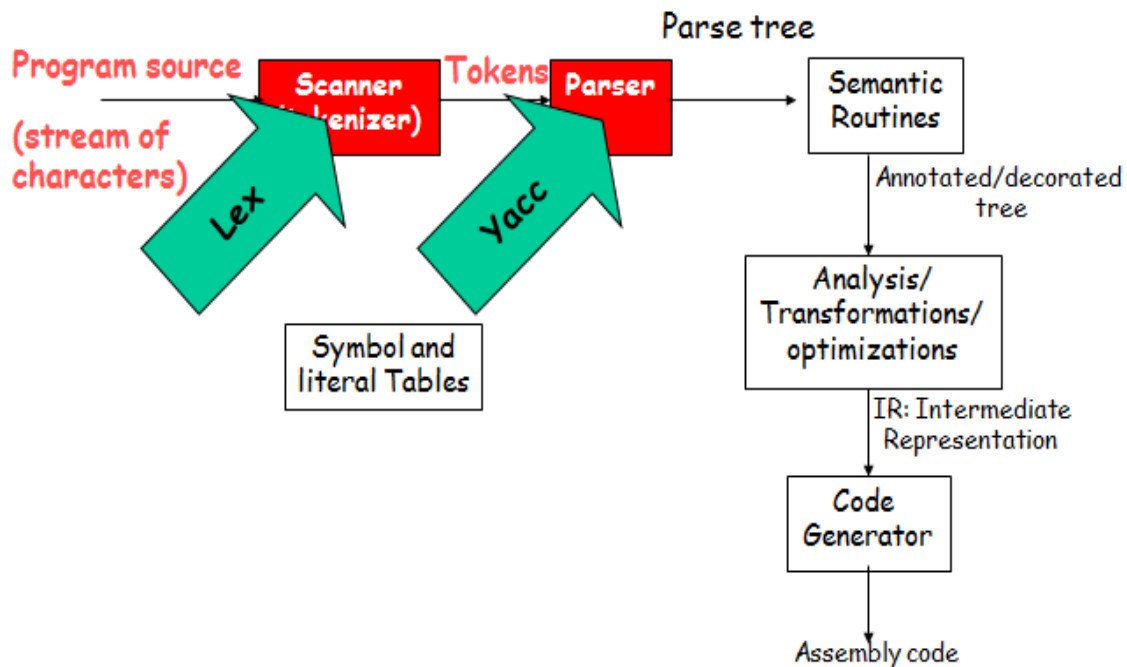
Other Algorithms

- ❖ How to minimize a DFA? (see Dragon Book 3.9, pp.173)
- ❖ How to convert RE to DFA directly? (see Dragon Book 3.9.5 pp.179)

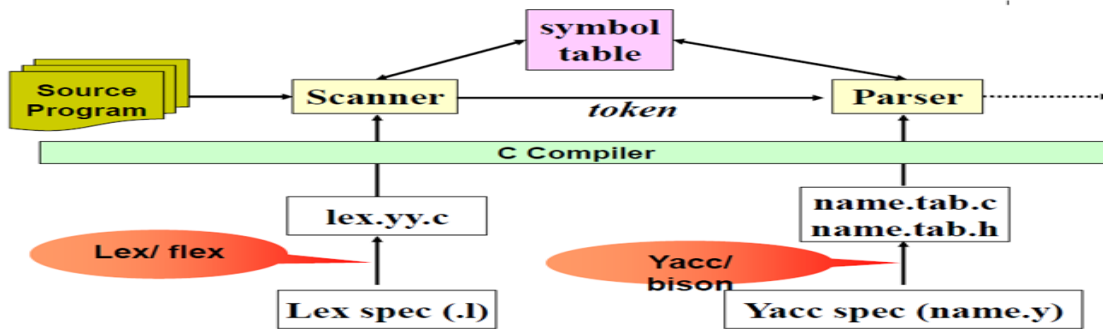
The Lexical- Analyzer Generator: Lex

- ✦ The first phase in a compiler is, it reads the input source and converts strings in the source to tokens.
- ✦ **Lex**: generates a scanner (lexical analyzer or lexer) given a specification of the tokens using REs.
 - The input notation for the Lex tool is referred to as the *Lex language* and
 - The tool itself is the *Lex compiler*.
- ✦ The Lex compiler **transforms** the **input patterns** into a **transition diagram** and **generates code**, in a file called **lex.yy.c**, that simulates this transition diagram.
- ✦ By using regular expressions, we can specify patterns to lex that allow it to scan and match strings in the input.
- ✦ Each pattern in lex has an associated action.
- ✦ Typically an action returns a token, representing the matched string, for subsequent use by the parser.
- ✦ It uses patterns that match strings in the input and converts the strings to **tokens**.

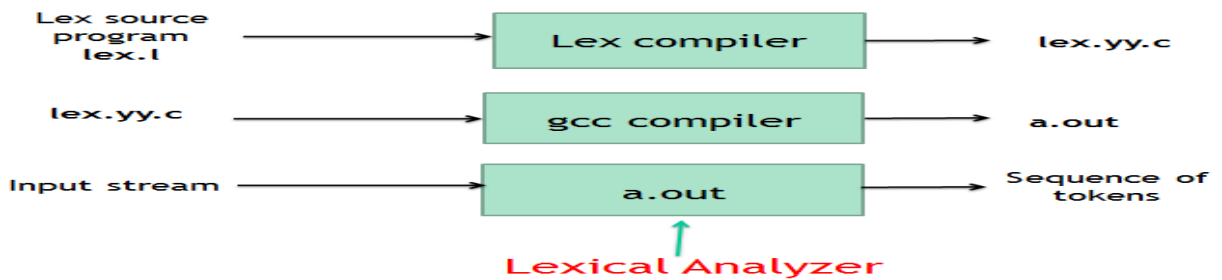
General Compiler Infra-structure



Scanner, Parser, Lex and Yacc



Generating a Lexical Analyzer using Lex



✚ We will see more about lex and its construction in lab case.

Summary of Chapter 2

Tokens: The lexical analyzer scans the source program and produces as output a sequence of tokens, which are normally passed, one at a time to the parser. Some tokens may consist only of a token name while others may also have an associated lexical value that gives information about the particular instance of the token that has been found on the input.

Lexemes: Each time the lexical analyzer returns a token to the parser, it has an associated lexeme - the sequence of input characters that the token represents.

Buffering: Because it is often necessary to scan ahead on the input in order to see where the next lexeme ends, it is usually necessary for the lexical analyzer to buffer its input. Using a pair of buffers cyclically and ending each buffer's contents with a sentinel that warns of its end are two techniques that accelerate the process of scanning the input. + Patterns. Each token has a pattern that describes which sequences of characters can form the lexemes corresponding to that token. The set of words or strings of characters that match a given pattern is called a language.

Regular Expressions: These expressions are commonly used to describe patterns. Regular expressions are built from single characters, using union, concatenation, and the Kleene closure, or any-number-of, operator.

Regular Definitions: Complex collections of languages, such as the patterns that describe the tokens of a programming language, are often defined by a regular definition, which is a sequence of statements that each define one variable to stand for some regular expression. The regular expression for one variable can use previously defined variables in its regular expression.

Transition Diagrams: The behavior of a lexical analyzer can often be described by a transition diagram. These diagrams have states, each of which represents something about the history of the characters seen during the current search for a lexeme that matches one of the possible patterns. There are arrows, or transitions, from one state to another, each of which indicates the possible next input characters that cause the lexical analyzer to make that change of state. + Finite Automata. These are a formalization of transition diagrams that include a designation of a start state and one or more accepting states, as well as the set of states, input characters, and transitions among states. Accepting states indicate that the lexeme for some token has been found. Unlike transition diagrams, finite automata can make transitions on empty input as well as on input characters.

Deterministic Finite Automata: A DFA is a special kind of finite automaton that has exactly one transition out of each state for each input symbol. Also, transitions on empty input are disallowed. The DFA is easily simulated and makes a good implementation of a lexical analyzer, similar to a transition diagram.

Nondeterministic Finite Automata: Automata that are not DFA's are called nondeterministic. NFA's often are easier to design than are DFA's. Another possible architecture for a lexical analyzer is to tabulate all the states that NFA's for each of the possible patterns can be in, as we scan the input characters.

Conversion among Pattern Representations: It is possible to convert any regular expression into an NFA of about the same size, recognizing the same language as the regular expression defines. Further, any NFA can be converted to a DFA for the same pattern, although in the worst case (never encountered in common programming languages) the size of the automaton can grow exponentially. It is also possible to convert any nondeterministic or deterministic finite automaton into a regular expression that defines the same language recognized by the finite automaton.

Lex: There is a family of software systems, including Lex and Flex, that are lexical-analyzer generators. The user specifies the patterns for tokens using an extended regular-expression notation. Lex converts these expressions into a lexical analyzer that is

essentially a deterministic finite automaton that recognizes any of the patterns.
Minimization of Finite Automata: For every DFA there is a minimum state DM accepting the same language. Moreover, the minimum-state DFA for a given language is unique except for the names given to the various states.

Review Exercise

1) Divide the following C++ program:

```
float limitedSquare(x) float x {  
    /* returns x-squared, but never more than 100 */  
    return (x<=-10.01 || x>=10.0)?100:x*x;
```

into appropriate lexemes. Which lexemes should get associated lexical values? What should those values be?

2) Write regular definitions for the following languages:

- a) All strings of lowercase letters that contain the five vowels in order.
- b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
- c) Comments, consisting of a string surrounded by /* and */, without an intervening */, unless it is inside double-quotes (").
- d) All strings of digits with no repeated digits. Hint: Try this problem first with a few digits, such as {0, 1, 2}. !!
- e) All strings of digits with at most one repeated digit. !!
- f) All strings of a's and b's with an even number of a's and an odd number of b's.
- g) The set of Chess moves, in the informal notation, such as p-k4 or kbp x qn.!!
- h) All strings of a's and b's that do not contain the substring abb.
- i) All strings of a's and b's that do not contain the subsequence abb.

3) Construct the minimum-state DFA7s for the following regular expressions:

- a) $(a|b)^*a(a|b)$
- b) $(a|b)^*a(a|b)(a|b)$
- c) $(a|b)^*a(a|b)(a|b)(a|b)$

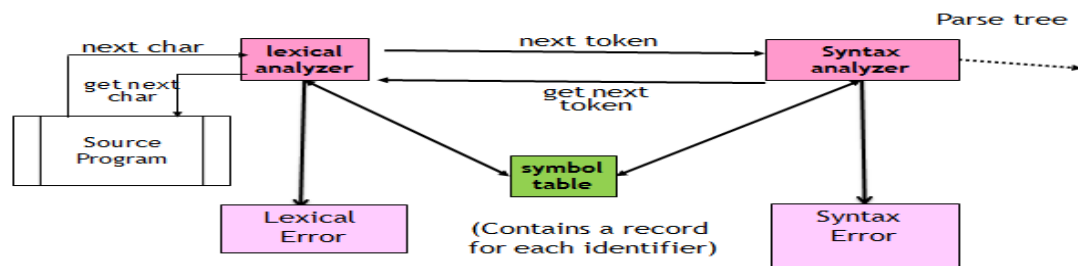
Chapter – 3

Syntax analysis

Introduction

- ✚ **Syntax:** the way in which tokens are put together to form expressions, statements, or blocks of statements.
 - The rules governing the formation of statements in a programming language.
- ✚ **Syntax analysis:** the task concerned with fitting a sequence of tokens into a specified syntax.
- ✚ **Parsing:** To break a sentence down into its component parts with an explanation of the form, function, and syntactical relationship of each part.
- ✚ The syntax of a programming language is usually given by the grammar rules of a context free grammar (CFG).

Parser



- ✚ The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a CFG or not.
 - If it satisfies, the parser creates the parse tree of that program.
 - Otherwise, the parser gives the error messages.

A CFG:

- Gives a precise syntactic specification of a programming language.
 - A grammar can be directly converted in to a parser by some tools (**yacc**).
- ✚ The parser can be categorized into two groups:
 - ✚ Top-down parser

- The parse tree is created top to bottom, starting from the root to leaves.
- ✚ Bottom-up parser
 - The parse tree is created bottom to top, starting from the leaves to root.
- ✚ Both top-down and bottom-up parser scan the input from left to right (one symbol at a time).
- ✚ Efficient top-down and bottom-up parsers can be implemented by making use of context-free- grammar.
 - LL for top-down parsing
 - LR for bottom-up parsing

Context free grammar (CFG)

- ✚ A context-free grammar is a specification for the syntactic structure of a programming language.
- ✚ Context-free grammar has 4-tuples:

$G = (T, N, P, S)$ where

 - **T** is a finite set of **terminals** (a set of tokens)
 - **N** is a finite set of **non-terminals** (syntactic variables)
 - **P** is a finite set of **productions** of the form $A \rightarrow \alpha$ where A is non-terminal and α is a strings of terminals and non-terminals (including the empty string)
- ✚ $S \in N$ is a designated start symbol (one of the non- terminal symbols)

Example: grammar for simple arithmetic expressions

<i>expression</i> \rightarrow <i>expression</i> + <i>term</i>	Terminal symbols
<i>expression</i> \rightarrow <i>expression</i> - <i>term</i>	id + - * / ()
<i>expression</i> \rightarrow <i>term</i>	
<i>term</i> \rightarrow <i>term</i> * <i>factor</i>	Non-terminals
<i>term</i> \rightarrow <i>term</i> / <i>factor</i>	expression
<i>term</i> \rightarrow <i>factor</i>	term
<i>factor</i> \rightarrow (<i>expression</i>)	Factor
<i>factor</i> \rightarrow id	Start symbol
	expression

Derivation

✚ A derivation is a sequence of replacements of structure names by choices on the right hand sides of grammar rules.

✚ Example: $E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid -E$

$$E \rightarrow (E)$$

$$E \rightarrow \mathbf{id}$$

$E \Rightarrow E + E$ means that $E + E$ is derived from E

- we can replace E by $E + E$
- we have to have a production rule $E \rightarrow E + E$ in our grammar.

$E \Rightarrow E + E \Rightarrow \mathbf{id} + E \Rightarrow \mathbf{id} + \mathbf{id}$ means that a sequence of replacements of non-terminal symbols is called a derivation of $\mathbf{id} + \mathbf{id}$ from E .

✚ If we always choose the left-most non-terminal in each derivation step, this derivation is called left-most derivation.

Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(\mathbf{id} + E) \Rightarrow -(\mathbf{id} + \mathbf{id})$

✚ If we always choose the right-most non-terminal in each derivation step, this derivation is called right-most derivation.

Example: $E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E + E) \Rightarrow -(E + \mathbf{id}) \Rightarrow -(\mathbf{id} + \mathbf{id})$

✚ We will see that the top-down parser try to find the left-most derivation of the given source program.

✚ We will see that the bottom-up parser try to find right-most derivation of the given source program in the reverse order.

Parse tree

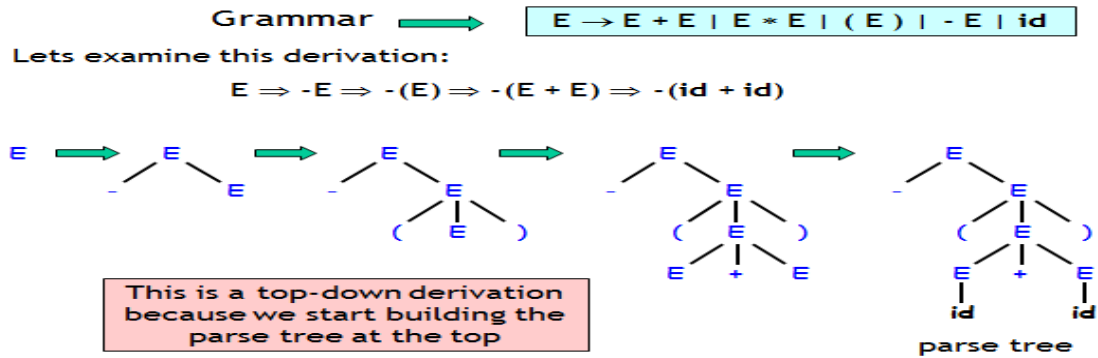
✚ A parse tree is a graphical representation of a derivation.

✚ It filters out the order in which productions are applied to replace non-terminals.

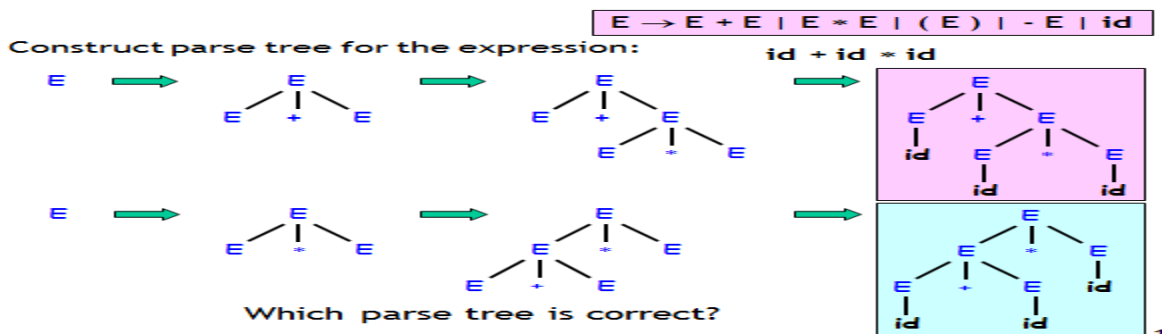
✚ A parse tree corresponding to a derivation is a labeled tree in which:

- the interior nodes are labeled by non-terminals,
- the leaf nodes are labeled by terminals, and
- the children of each internal node represent the replacement of the associated non-terminal in one step of the derivation.

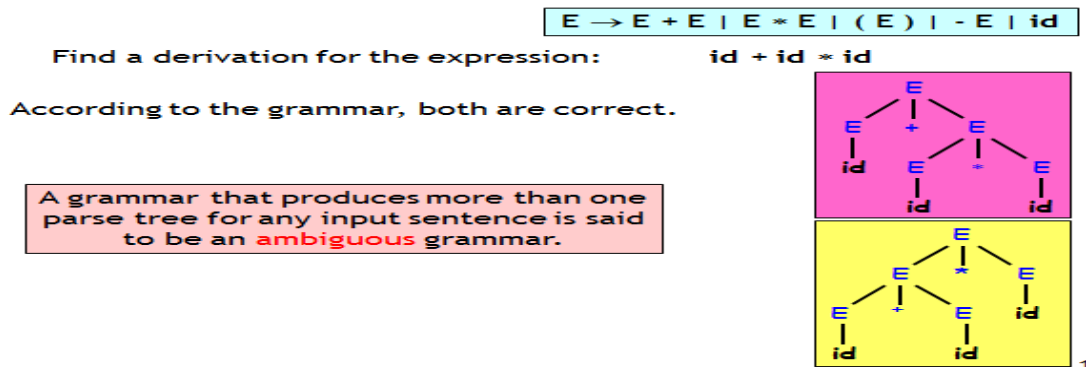
Parse tree and Derivation



Ambiguity: example



Ambiguity: example...



Elimination of ambiguity Precedence/Association

- ✚ These two derivations point out a problem with the grammar:
- ✚ The grammar do not have notion of precedence, or implied order of evaluation

To add precedence

- Create a non-terminal for each *level of precedence*
- Isolate the corresponding part of the grammar

- Force the parser to recognize high precedence sub expressions first

For algebraic expressions

- Multiplication and division, first (*level one*)
- Subtraction and addition, next (*level two*)

To add association

- **Left-associative** : The next-level (higher) non-terminal places at the last of a production
- Elimination of ambiguity
- To disambiguate the grammar :

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

- we can use precedence of operators as follows:
 - * Higher precedence (left associative)
 - + Lower precedence (left associative)
- We get the following unambiguous grammar:

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

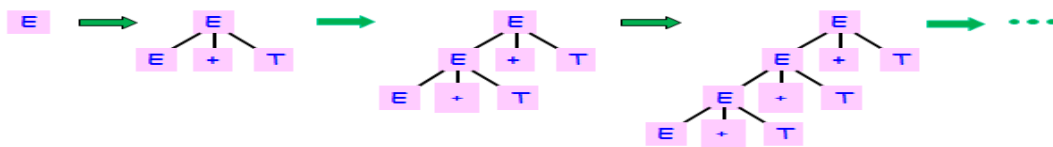
id + id * id

Left Recursion

Consider the grammar:

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

A top-down parser might loop forever when parsing an expression using this grammar



Elimination of Left recursion

✚ A grammar is left recursive, if it has a non-terminal A such that there is a derivation

$$A \Rightarrow^+ A\alpha \quad \text{for some string } \alpha.$$

✚ **Top-down parsing** methods cannot handle left-recursive grammar. So a transformation that eliminates left-recursion is needed.

✚ To eliminate left recursion for single production $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left- recursive productions

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \epsilon$$

This left-recursive grammar:

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

Can be re-written to eliminate the immediate left recursion:

$$\begin{array}{l} E \rightarrow TE' \\ E' \rightarrow +TE' \mid \epsilon \\ T \rightarrow FT' \\ T' \rightarrow *FT' \mid \epsilon \\ F \rightarrow (E) \mid id \end{array}$$

✚ Generally, we can eliminate immediate left recursion from them by the following technique.

✚ First we group the A-productions as:

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

Where no β_i begins with A. then we replace the A productions by:

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon$$

Left factoring

✚ When a non-terminal has two or more productions whose right-hand sides start with the same grammar symbols, the grammar is not LL(1) and cannot be used for predictive parsing

✚ A predictive parser (a top-down parser without backtracking) insists that the grammar must be **left-factored**.

In general: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$, where α -is a non-empty and the first symbol of β_1 and β_2 .

✚ When processing α we do not know whether to expand A to $\alpha\beta_1$ or to $\alpha\beta_2$, but if we re-write the grammar as follows:

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2 \text{ so, we can immediately expand A to } \alpha A'.$$

✚ Example: given the following grammar:

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$

✚ Left factored, this grammar becomes:

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

The following grammar:

$stmt \rightarrow if\ expr\ then\ stmt\ else\ stmt$
 $\mid\ if\ expr\ then\ stmt$

Cannot be parsed by a predictive parser that looks one element ahead.

But the grammar can be re-written:

$stmt \rightarrow if\ expr\ then\ stmt\ stmt'$
 $stmt' \rightarrow else\ stmt \mid \varepsilon$

Where ε is the empty string.

Rewriting a grammar to eliminate multiple productions starting with the same token is called **left factoring**.

Syntax analysis

- ✚ Every language has rules that prescribe the syntactic structure of well-formed programs.
- ✚ The syntax can be described using Context Free Grammars (CFG) notation.
- ✚ The use of CFGs has several advantages:
 - helps in identifying ambiguities
 - it is possible to have a tool which produces automatically a parser using the grammar
 - a properly designed grammar helps in modifying the parser easily when the language changes

Top-down parsing Recursive Descent Parsing (RDP)

- ✚ This method of top-down parsing can be considered as an attempt to find the left most derivation for an input string. It may involve backtracking.
- ✚ To construct the parse tree using RDP:
 - We create one node tree consisting of S.
 - Two pointers, one for the tree and one for the input, will be used to indicate where the parsing process is.

- Initially, they will be on S and the first input symbol, respectively.
- Then we use the first S-production to expand the tree. The tree pointer will be positioned on the left most symbol of the newly created sub-tree.
- ✚ As the symbol pointed by the tree pointer matches that of the symbol pointed by the input pointer, both pointers are moved to the right.
- ✚ Whenever the tree pointer points on a non-terminal, we expand it using the first production of the non-terminal.
- ✚ Whenever the pointers point on different terminals, the production that was used is not correct, thus another production should be used. We have to go back to the step just before we replaced the non-terminal and use another production.
- ✚ If we reach the end of the input and the tree pointer passes the last symbol of the tree, we have finished parsing.

Example: $G: S \rightarrow cAd$

$A \rightarrow ab|a$

- ✚ Draw the parse tree for the input string cad using the above method.

Exercise:

1) Consider the following grammar:

$S \rightarrow A$

$A \rightarrow A + A \mid B++$

$B \rightarrow y$

- ✚ Draw the parse tree for the input “y+++y++”

2) Using the grammar below, construct a parse tree for the following string using RDP algorithm: $((\mathbf{id} . \mathbf{id}) \mathbf{id} (\mathbf{id}) (()))$

$S \rightarrow E$

$E \rightarrow id$

$| (E . E)$

$| (L)$

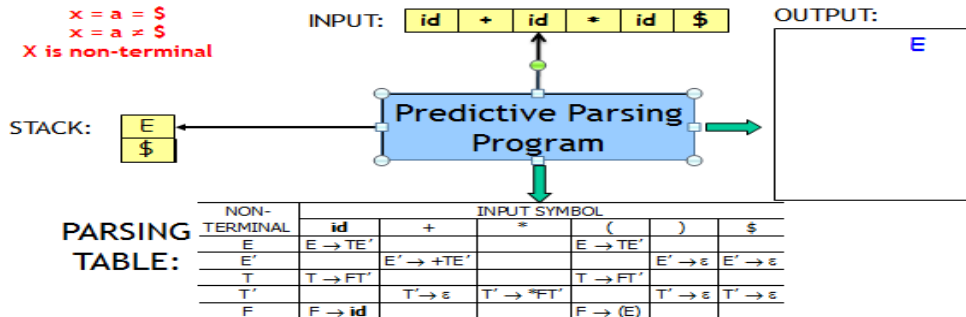
$| ()$

$$L \rightarrow L E$$

$$| E$$

Non-recursive predictive parsing

- It is possible to build a non-recursive parser by explicitly maintaining a stack.
- This method uses a parsing table that determines the next production to be applied. The input buffer contains the string to be parsed followed by \$ (the right end marker)



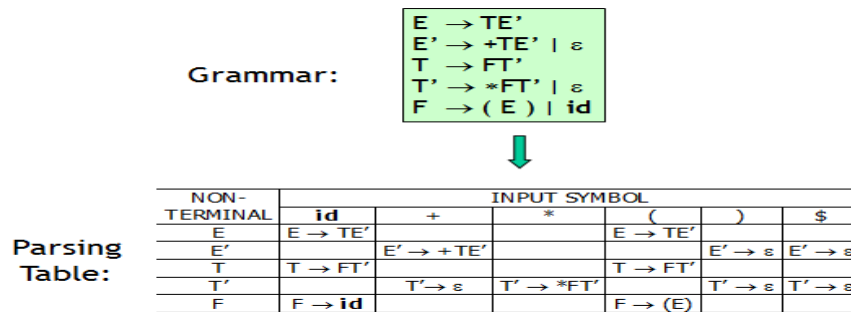
- The stack contains a sequence of grammar symbols with \$ at the bottom.
- Initially, the stack contains the start symbol of the grammar followed by \$.
- The parsing table is a two dimensional array $M[A, a]$ where A is a non-terminal of the grammar and a is a terminal or \$.
- The parser program behaves as follows.
- The program always considers
 - X, the symbol on top of the stack and
 - a, the current input symbol.

Predictive Parsing...

- There are three possibilities:
 - $x = a = \$$: the parser halts and announces a successful completion of parsing
 - $x = a \neq \$$: the parser pops x off the stack and advances the input pointer to the next symbol
 - X is a non-terminal**: the program consults entry $M[X, a]$ which can be an X-production or an error entry.

- ✦ If $M[X, a] = \{X \rightarrow uvw\}$, X on top of the stack will be replaced by uvw (u at the top of the stack).
- ✦ As an output, any code associated with the X -production can be executed.
- ✦ If $M[X, a] = \text{error}$, the parser calls the error recovery method.

A Predictive Parser table



A Predictive Parser: Example

Input	Stack	Output
<u>id</u> +id*id\$	E\$	Parse tree

Non-recursive predictive parsing Example: G:

$E \rightarrow TR$

$R \rightarrow +TR$ Input: 1+2

$R \rightarrow -TR$

$R \rightarrow \varepsilon$

$T \rightarrow 0|1|\dots|9$

<u>X a</u>	0	1	...	9	+	-	\$
E	$E \rightarrow TR$	$E \rightarrow TR$...	$E \rightarrow TR$	Error	Error	Error
R	Error	Error	...	Error	$R \rightarrow +TR$	$R \rightarrow -TR$	$R \rightarrow \varepsilon$
T	$T \rightarrow 0$	$T \rightarrow 1$...	$T \rightarrow 9$	Error	Error	Error

Parsing 1 + 2

Input	Stack
1+2\$	E\$
1+2\$	TR\$
1+2\$	1R\$
+2\$	R\$
+2\$	+TR\$
2\$	TR\$
2\$	2R\$
\$	R\$
\$	\$

FIRST and FOLLOW

- ✚ The construction of both top-down and bottom-up parsers are aided by two functions, FIRST and FOLLOW, associated with a grammar G.
- ✚ During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.
- ✚ We need to build a FIRST set and a FOLLOW set for each symbol in the grammar. The elements of FIRST and FOLLOW are terminal symbols.
 - $FIRST(\alpha)$ is the set of terminal symbols that can begin any string derived from α .
 - $FOLLOW(\alpha)$ is the set of terminal symbols that can follow α : $t \in FOLLOW(\alpha) \leftrightarrow \exists$ derivation containing αt

Construction of a predictive parsing table

- ✚ Makes use of two functions: FIRST and FOLLOW.

FIRST

- $FIRST(\alpha)$ = set of terminals that begin the strings derived from α .
- If $\alpha \Rightarrow \epsilon$ in zero or more steps, ϵ is in $FIRST(\alpha)$.
- $FIRST(X)$ where X is a grammar symbol can be found using the following rules:

1- If X is a terminal, then $FIRST(x) = \{x\}$

2- If X is a non-terminal: two cases

a) If $X \rightarrow \epsilon$ is a production, then add ϵ to $FIRST(X)$

b) For each production $X \rightarrow y_1y_2\dots y_k$, place a in $FIRST(X)$ if for some i, $a \in FIRST(y_i)$ and $\epsilon \in FIRST(y_j)$, for $1 < j < i$. If $\epsilon \in FIRST(y_j)$, for $j=1, \dots, k$ then $\epsilon \in FIRST(X)$

For any string $y = x_1x_2\dots x_n$

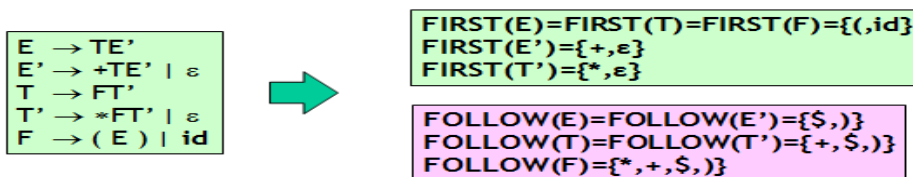
- a- Add all non- ϵ symbols of $FIRST(X_i)$ in $FIRST(y)$
- b- Add all non- ϵ symbols of $FIRST(X_j)$ for $i \neq j$ if for all $j < i$, $\epsilon \in FIRST(X_j)$
- c- $\epsilon \in FIRST(y)$ if $\epsilon \in FIRST(X_i)$ for all i

FOLLOW

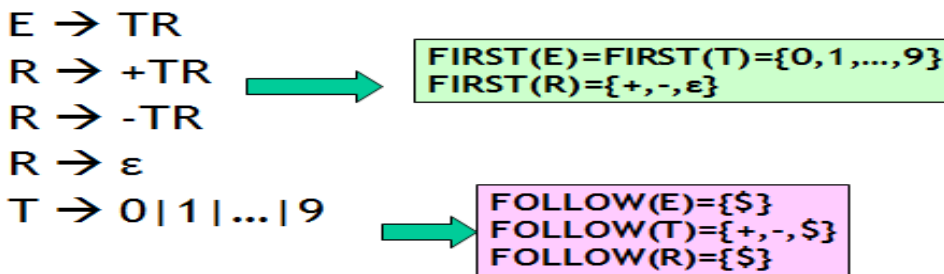
- ✚ FOLLOW(A) = set of terminals that can appear immediately to the right of A in some sentential form.
 - Place \$ in FOLLOW(A), where A is the start symbol.
 - If there is a production $B \rightarrow \alpha A \beta$, then everything in $FIRST(\beta)$, except ϵ , should be added to FOLLOW(A).
 - If there is a production $B \rightarrow \alpha A$ or $B \rightarrow \alpha A \beta$ and $\epsilon \in FIRST(\beta)$, then all elements of FOLLOW(B) should be added to FOLLOW(A).

Exercises:

1) Consider the following grammars G, find FIRST and FOLLOW sets.



2) Find FIRST and FOLLOW sets for the following grammar G:



3) Consider the following grammar over the alphabet { g,h,i,b}

- $A \rightarrow BCD$
- $B \rightarrow bB \mid \epsilon$
- $C \rightarrow Cg \mid g \mid Ch \mid i$
- $D \rightarrow AB \mid \epsilon$

Fill in the table below with the FIRST and FOLLOW sets for the non-terminals in this grammar:

	FIRST	FOLLOW
A		
B		
C		
D		

Construction of predictive parsing table

- Input Grammar G
- Output Parsing table M

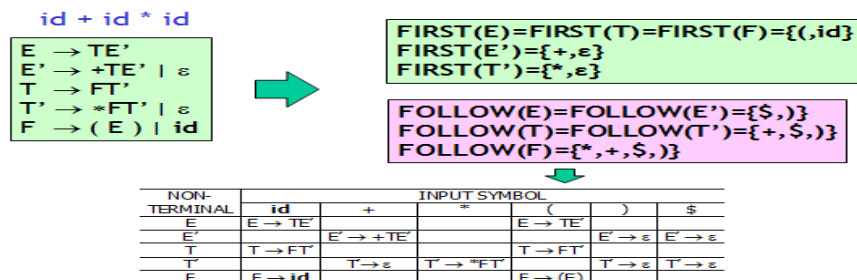
✚ For each production of the form $A \rightarrow \alpha$ of the grammar do:

- ✓ For each terminal a in $FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
- ✓ If $\epsilon \in FIRST(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each b in $FOLLOW(A)$
- ✓ If $\epsilon \in FIRST(\alpha)$ and $\$ \in FOLLOW(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$
- ✓ Make each undefined entry of M be an error.

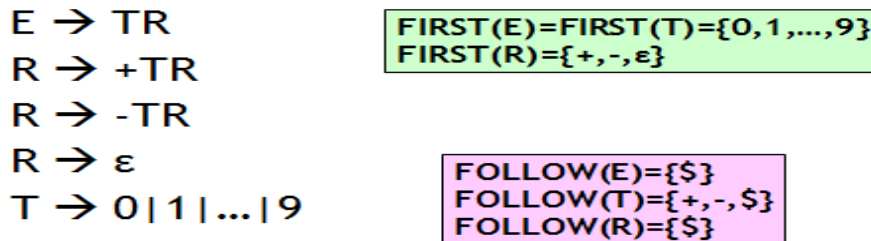
✚ Non-recursive predictive parsing...

Exercise:

- 1) Consider the following grammars G, Construct the predictive parsing table and parse the input symbols:



- 2) Construct the predictive parsing table for the grammar G:



- 3) Let G be the following grammar:

$$S \rightarrow [SX] | a$$

$$X \rightarrow \varepsilon | +SY | Yb$$

$$Y \rightarrow \varepsilon | -SXc$$

A – Find FIRST and FOLLOW sets for the non-terminals in this grammar.

B – Construct predictive parsing table for the grammar above.

C – Show a top down parse of the string **[a+a-ac]**

LL (1) Grammars...

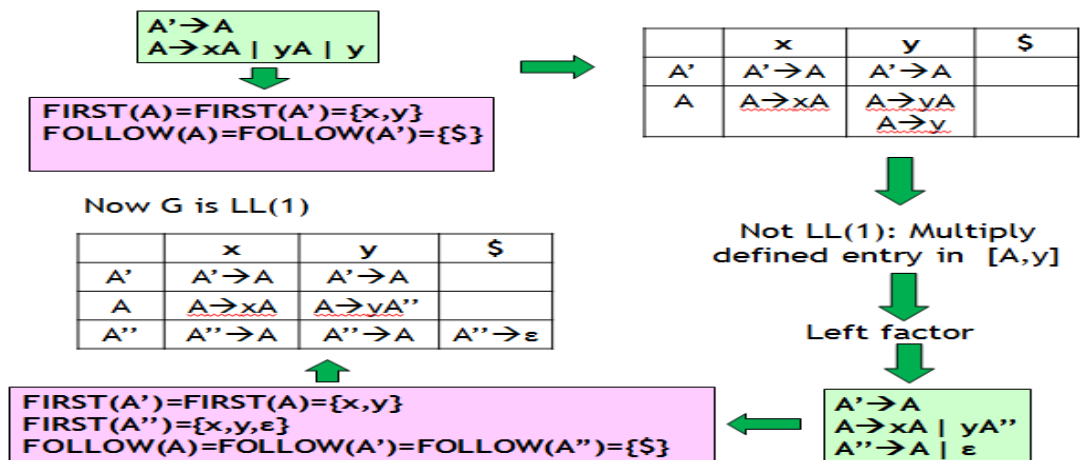
Exercises: 1) Consider the following grammar G:

$$A' \rightarrow A$$

$$A \rightarrow xA | yA | y$$

- Find FIRST and FOLLOW sets for G:
- Construct the LL(1) parse table for this grammar.
- Explain why this grammar is not LL(1).
- Transform the grammar into a grammar that is LL(1).
- Give the parse table for the grammar created in (d).

Solution:



2) Given the following grammar:

$S \rightarrow WAB \mid ABCS$

$A \rightarrow B \mid WB$

$B \rightarrow \varepsilon \mid yB$

$C \rightarrow z$

$W \rightarrow x$

- a) Find FIRST and FOLLOW sets of the grammar.
 - b) Construct the LL(1) parse table.
 - c) Is the grammar LL(1)? Justify your answer.
- 3) Consider the following grammar:

$S \rightarrow ScB \mid B$

$B \rightarrow e \mid efg \mid efCg$

$C \rightarrow SdC \mid S$

- a) Justify whether the grammar is LL(1) or not?
- b) If not, translate the grammar into LL(1).
- c) Construct predictive parsing table for the above grammar.

Bottom-Up and Top-Down Parsers

Top-down parsers:

- ✚ Starts constructing the parse tree at the top (root) of the tree and move down towards the leaves.
- ✚ Easy to implement by hand, but work with restricted grammars.

Example: predictive parsers

Bottom-up parsers:

- ✚ Build the nodes on the bottom of the parse tree first.
- ✚ Suitable for automatic parser generation, handle a larger class of grammars.

Example: shift-reduce parser (or LR (k) parsers)

- ✚ A bottom-up parser, or a shift-reduce parser, begins at the leaves and works up to the top of the tree. The reduction steps trace a rightmost derivation on reverse.

Consider the Grammar:

S	→	aABe
A	→	Abc b
B	→	d

- ✚ We want to parse the input string abcde. This parser is known as an LR Parser because it scans the input from Left to right, and it constructs a rightmost derivation in reverse order.

Example of Bottom-up parser (LR parsing)

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$abcde \leftarrow aAbcde \leftarrow aAde \leftarrow aABe \leftarrow S$

- ✚ At each step, we have to find α such that α is a substring of the sentence and replace α by A, where $A \rightarrow \alpha$

Stack implementation of shift/reduce parsing

- ✚ In LR parsing the two major problems are:
 - locate the substring that is to be reduced
 - locate the production to use
- ✚ A shift/reduce parser operates:
 - By shifting zero or more input into the stack until the right side of the handle is on top of the stack.
 - The parser then replaces handle by the non-terminal of the production.
 - This is repeated until the start symbol is in the stack and the input is empty, or until error is detected.
- ✚ Four actions are possible:
 - Shift: the next input is shifted on to the top of the stack
 - Reduce: the parser knows the right end of the handle is at the top of the stack. It should then decide what non-terminal should replace that substring

- Accept: the parser announces successful completion of parsing
- Error: the parser discovers a syntax error

Example: An example of the operations of a shift/reduce parser G: $E \rightarrow E + E \mid E * E \mid (E) \mid id$

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

Conflict during shift/reduce parsing

✚ Grammars for which we can construct an LR (k) parsing table are called LR (k) grammars.

✚ Most of the grammars that are used in practice are LR (1).

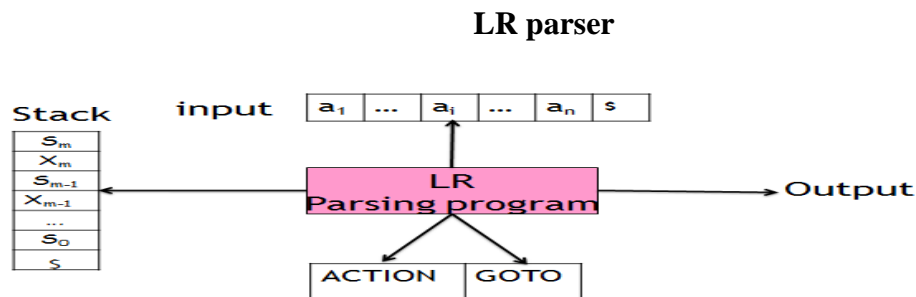
✚ There are two types of conflicts in shift/reduce parsing:

- **Shift/reduce conflict:** when we have a situation where the parser knows the entire stack content and the next k symbols but cannot decide whether it should shift or reduce. Ambiguity
- **Reduce/reduce conflict:** when the parser cannot decide which of the several productions it should use for a reduction.

$$E \rightarrow T$$

$$E \rightarrow id \quad \text{with an id on the top of stack}$$

$$T \rightarrow id$$



✚ The LR(k) stack stores strings of the form: $S_0 X_0 S_1 X_1 \dots X_m S_m$ where

- S_i is a new symbol called state that summarizes the information contained in the stack
- S_m is the state on top of the stack
- X_i is a grammar symbol

✚ The parser program decides the next step by using:

- ✓ the top of the stack (S_m),
- ✓ the input symbol (a_i), and
- ✓ the parsing table which has two parts: ACTION and GOTO.
- ✓ then consulting the entry $ACTION[S_m, a_i]$ in the parsing action table

Structure of the LR Parsing Table

✚ The parsing table consists of two parts:

- a parsing-action function ACTION and
- a goto function GOTO.

✚ The ACTION function takes as arguments a state i and a terminal a (or $\$,$ the input endmarker).

✚ The value of $ACTION[i, a]$ can have one of four forms:

- Shift j , where j is a state. The action taken by the parser shifts input a on the top of the stack, but uses state j to represent a .
- Reduce $A \rightarrow \beta$, The action of the parser reduces β on the top of the stack to head A .
- Accept, The parser accepts the input and finishes parsing.
- Error, The parser discovers an error

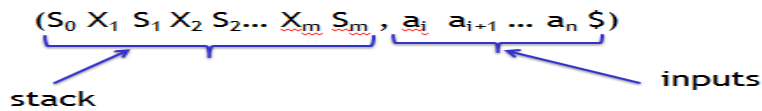
✚ GOTO function, defined on sets of items, to states.

- $GOTO[I_i, A] = I_j$, then GOTO maps a state i and a non-terminal A to state j .

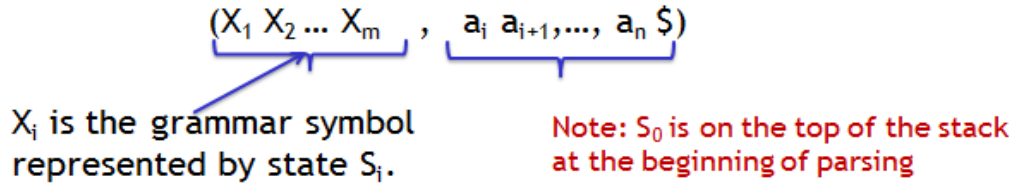
LR parser configuration

✚ Behavior of an LR parser \rightarrow describes the complete state of the parser.

✚ A configuration of an LR parser is a pair:



✚ This configuration represents the right-sentential form



64

Behavior of LR parser

✚ The parser program decides the next step by using:

- the top of the stack (S_m),
- the input symbol (a_i), and
- the parsing table which has two parts: ACTION and GOTO.
- then consulting the entry ACTION[S_m, a_i] in the parsing action table

1. If Action[S_m, a_i] = shift S , the parser program shifts both the current input symbol a_i and state S on the top of the stack, entering the configuration

$$(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$$

2. Action[S_m, a_i] = reduce $A \rightarrow \beta$: the parser pops the first $2r$ symbols off the stack, where $r = |\beta|$ (at this point, S_{m-r} will be the state on top of the stack), entering the configuration,

$$(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} A S, a_i a_{i+1} \dots a_n \$)$$

- Then A and S are pushed on top of the stack where $S = \text{goto}[S_{m-r}, A]$. The input buffer is not modified.

3. Action[S_m, a_i] = accept, parsing is completed.

4. Action[S_m, a_i] = error, parsing has discovered an error and calls an error recovery routine.

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] = shift t ) {

    push t 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;

    output the production A →β;

} else if ( ACTION[S, a] = accept ) break; /* parsing is done */

else call error-recovery routine; }

```

Example: Let G1 be:

- 1) $E \rightarrow E + T$
- 2) $E \rightarrow T$
- 3) $T \rightarrow T * F$
- 4) $T \rightarrow F$
- 5) $F \rightarrow (E)$
- 6) $F \rightarrow id$

The codes for the actions are:

1. s_i means shift and stack state i ,
2. r_j means reduce by the production numbered j ,
3. acc means accept,
4. $blank$ means error.

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

Legend: S_i means shift to state i , R_j means reduce production by j

✚ The following grammar can be parsed with this **action** and **goto** table as bellow.

- (1) $E \rightarrow E + T$
- (2) $E \rightarrow T$
- (3) $T \rightarrow T * F$
- (4) $T \rightarrow F$
- (5) $F \rightarrow (E)$
- (6) $F \rightarrow id$

State	action						goto		
	id	+	*	()	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2				
3		r4	r4		r4				
4	s5			s4			8	2	3
5		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			

Example: The following example shows how a shift/reduce parser parses an input string $w = id * id + id$ using the parsing table shown above.

Stack	Input	Action
0	id * id + id \$	shift 5
0 id 5	* id + id \$	reduce 6 ($F \rightarrow id$)
0 F 3	* id + id \$	reduce 4 ($T \rightarrow F$)
0 T 2	* id + id \$	shift 7
0 T 2 * 7	id + id \$	shift 5
0 T 2 * 7 id 5	+ id \$	reduce 6 ($F \rightarrow id$)
0 T 2 * 7 F 10	+ id \$	reduce 3 ($T \rightarrow T * F$)
0 T 2	+ id \$	reduce 2 ($E \rightarrow T$)
0 E 1	+ id \$	shift 6
0 E 1 + 6	id \$	shift 5
0 E 1 + 6 id 5	\$	reduce 6 ($F \rightarrow id$)
0 E 1 + 6 F 3	\$	reduce 4 ($T \rightarrow F$)
0 E 1 + 6 T 9	\$	reduce 1 ($E \rightarrow E + T$)
0 E 1	\$	accept

Constructing SLR parsing tables

This method is the simplest of the three methods used to construct an LR parsing table. It is called SLR (simple LR) because it is the easiest to implement. However, it is also the weakest in terms of the number of grammars for which it succeeds. A parsing table constructed by this method is called SLR table. A grammar for which an SLR table can be constructed is said to be an SLR grammar.

LR (0) item

An LR (0) item (item for short) is a production of a grammar G with a **dot** at some position of the right side.

For example for the production $A \rightarrow X Y Z$ we have four items:

$$A \rightarrow \cdot X Y Z$$

$$A \rightarrow X \cdot Y Z$$

$$A \rightarrow X Y \cdot Z$$

$$A \rightarrow X Y Z \cdot$$

For the production $A \rightarrow \epsilon$ we only have one item:

$$A \rightarrow \cdot$$

- ✚ An item indicates what is the part of a production? That we have seen and what we hope to see. The central idea in the SLR method is to construct, from the grammar, a deterministic finite automaton to recognize viable prefixes.
- ✚ A viable prefix is a prefix of a right sentential form that can appear on the stack of a shift/reduce parser.
 - If you have a viable prefix in the stack it is possible to have inputs that will reduce to the start symbol.
 - If you don't have a viable prefix on top of the stack you can never reach the start symbol; therefore you have to call the error recovery procedure.

The closure operation

- ✚ If I is a set of items of G , then **Closure (I)** is the set of items constructed by two rules:
 - Initially, every item in I is added to closure (I)
 - If $A \rightarrow \alpha.B\beta$ is in Closure of (I) and $B \rightarrow \gamma$ is a production, then add $B \rightarrow .\gamma$ to I .
 - This rule is applied until no more new item can be added to Closure (I).

Example G_1 ':

$E' \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

✚ $I = \{[E' \rightarrow .E]\}$

✚ $\text{Closure}(I) = \{[E' \rightarrow .E], [E \rightarrow .E + T], [E \rightarrow .T], [T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id]\}$

The Goto operation

- ✚ The second useful function is Goto (I, X) where I is a set of items and X is a grammar symbol. Goto (I, X) is defined as the closure of all items $[A \rightarrow \alpha X \beta]$ such that $[A \rightarrow \alpha X \beta]$ is in I.

Example:

$I = \{[E' \rightarrow E.], [E \rightarrow E . + T]\}$ Then $\text{goto}(I, +) = \{[E \rightarrow E + . T], [T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)] [F \rightarrow .id]\}$

The set of Items construction

- ✚ Below is given an algorithm to construct C, the canonical collection of sets of LR(0) items for augmented grammar G' .

Procedure Items (G');

Begin

$C := \{\text{Closure}(\{[S' \rightarrow . S]\})\}$

Repeat

- ✚ For Each item of I in C and each grammar symbol X such that Goto (I, X) is not empty and not in C do

 Add Goto (I, X) to C;

Until no more sets of items can be added to C

End

Example: Construction of the set of Items for the augmented grammar above $G1'$.

✚ $I_0 = \{[E' \rightarrow .E], [E \rightarrow .E + T], [E \rightarrow .T], [T \rightarrow .T * F],$

✚ $[T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id]\}$

✚ $I_1 = \text{Goto}(I_0, E) = \{[E' \rightarrow E.], [E \rightarrow E . + T]\}$

✚ $I_2 = \text{Goto}(I_0, T) = \{[E \rightarrow T.], [T \rightarrow T . * F]\}$

✚ $I_3 = \text{Goto}(I_0, F) = \{[T \rightarrow F.]\}$

✚ $I_4 = \text{Goto}(I_0, () = \{[F \rightarrow (.E)], [E \rightarrow .E + T], [E \rightarrow .T],$

 ○ $[T \rightarrow .T * F], [T \rightarrow .F], [F \rightarrow .(E)], [F \rightarrow .id]\}$

✚ $I_5 = \text{Goto}(I_0, id) = \{[F \rightarrow id.]\}$

✚ $I_6 = \text{Goto}(I_1, +) = \{[E \rightarrow E + \cdot T], [T \rightarrow \cdot T * F], [T \rightarrow \cdot F],$

○ $[F \rightarrow \cdot (E)], [F \rightarrow \cdot \text{id}]\}$

✚ $I_7 = \text{Goto}(I_2, *) = \{[T \rightarrow T * \cdot F], [F \rightarrow \cdot (E)],$

○ $[F \rightarrow \cdot \text{id}]\}$

✚ $I_8 = \text{Goto}(I_4, E) = \{[F \rightarrow (E \cdot)], [E \rightarrow E \cdot + T]\}$

○ $\text{Goto}(I_4, T) = \{[E \rightarrow T \cdot], [T \rightarrow T \cdot * F]\} = I_2;$

○ $\text{Goto}(I_4, F) = \{[T \rightarrow F \cdot]\} = I_3;$

○ $\text{Goto}(I_4, () = I_4;$

○ $\text{Goto}(I_4, \text{id}) = I_5;$

✚ $I_9 = \text{Goto}(I_6, T) = \{[E \rightarrow E + T \cdot], [T \rightarrow T \cdot * F]\}$

○ $\text{Goto}(I_6, F) = I_3;$

○ $\text{Goto}(I_6, () = I_4;$

○ $\text{Goto}(I_6, \text{id}) = I_5;$

✚ $I_{10} = \text{Goto}(I_7, F) = \{[T \rightarrow T * F \cdot]\}$

○ $\text{Goto}(I_7, () = I_4;$

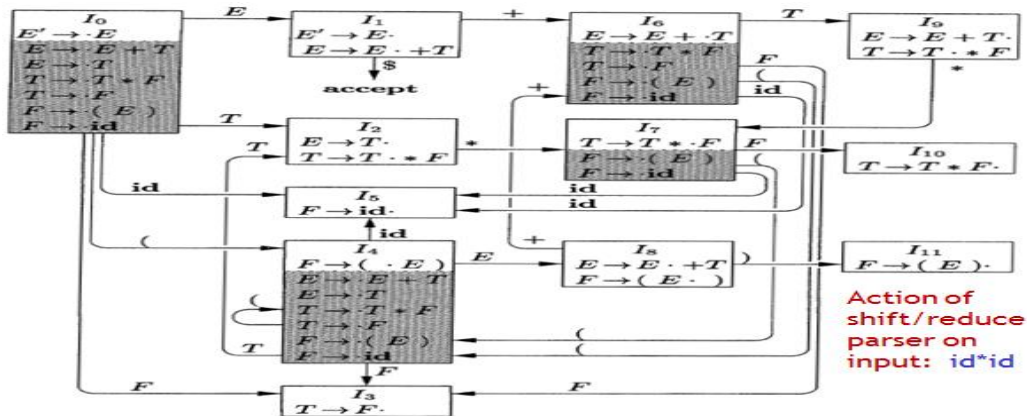
○ $\text{Goto}(I_7, \text{id}) = I_5;$

✚ $I_{11} = \text{Goto}(I_8,) = \{[F \rightarrow (E \cdot)]\}$

○ $\text{Goto}(I_8, +) = I_6;$

○ $\text{Goto}(I_8, *) = I_7;$

LR (0) automation



SLR table construction algorithm

1. Construct $C = \{I_0, I_1, \dots, I_N\}$ the collection of the set of LR (0) items for G' .
2. State i is constructed from I_i and
 - a) If $[A \rightarrow \alpha.a\beta]$ is in I_i and $\text{Goto}(I_i, a) = I_j$ (a is a terminal) then action $[i, a] = \text{shift } j$
 - b) If $[A \rightarrow \alpha.]$ is in I_i then action $[i, a] = \text{reduce } A \rightarrow \alpha$ for a in $\text{Follow}(A)$ for $A \neq S'$
 - c) If $[S' \rightarrow S.]$ is in I_i then action $[i, \$] = \text{accept}$.
 - If no conflicting action is created by 1 and 2 the grammar is SLR (1); otherwise it is not.
3. For all non-terminals A , if $\text{Goto}(I_i, A) = I_j$ then $\text{Goto}[i, A] = j$
4. All entries of the parsing table not defined by 2 and 3 are made error
5. The initial state is the one constructed from the set of items containing $[S' \rightarrow .S]$

Example: Construct the SLR parsing table for the grammar $G1'$

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

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

$E' \rightarrow E$

1 $E \rightarrow E + T$

2 $E \rightarrow T$

3 $T \rightarrow T * F$

4 $T \rightarrow F$

5 $F \rightarrow (E)$

6 $F \rightarrow \text{id}$

✚ By following the method we find the Parsing table used earlier.

State	action						goto		
	id	+	*	()	\$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Legend: S_i means shift to state i , R_j means reduce production by j

Exercise: Construct the SLR parsing table for the following grammar: /* Grammar G_2 */

$S' \rightarrow S$

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

Answer

✚ $C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9\}$

✚ $I_0 = \{[S' \rightarrow .S], [S \rightarrow .L = R], [S \rightarrow .R], [L \rightarrow .*R],$

○ $[L \rightarrow .id], [R \rightarrow .L]\}$

✚ $I_1 = \text{goto}(I_0, S) = \{[S' \rightarrow S.]\}$

✚ $I_2 = \text{goto}(I_0, L) = \{[S \rightarrow L. = R], [R \rightarrow L.]\}$

✚ $I_3 = \text{goto}(I_0, R) = \{[S \rightarrow R.]\}$

✚ $I_4 = \text{goto}(I_0, *) = \{[L \rightarrow *. R], [L \rightarrow .*R], [L \rightarrow .id],$

○ $[R \rightarrow .L]\}$

✚ $I_5 = \text{goto}(I_0, id) = \{[L \rightarrow id.]\}$

✚ $I_6 = \text{goto}(I_2, =) = \{[S \rightarrow L = . R], [R \rightarrow . L], [L \rightarrow .*R],$

- $[L \rightarrow \text{id}]$
- ✚ $I7 = \text{goto}(I4, R) = \{[L \rightarrow * R .]\}$
- ✚ $I8 = \text{goto}(I4, L) = \{[R \rightarrow L .]\}$
 - $\text{goto}(I4, *) = I4$
 - $\text{goto}(I4, \text{id}) = I5$
- ✚ $I9 = \text{goto}(I6, R) = \{[S \rightarrow L = R .]\}$
 - $\text{goto}(I6, L) = I8$
 - $\text{goto}(I6, *) = I4$
 - $\text{goto}(I6, \text{id}) = I5$
- ✚ Follow (S) = {\$} Follow (R) = {\$, =} Follow (L) = {\$, =}. We have shift/reduce conflict since = is in Follow (R) and $R \rightarrow L$ is in I2 and $\text{Goto}(I2, =) = I6$. Every SLR(1) grammar is unambiguous, but there are many unambiguous grammars that are not SLR(1).
- ✚ $G2'$ is not an ambiguous grammar. However, it is not SLR. This is because the SLR parser is not powerful enough to remember enough left context to decide whether to shift or reduce when it sees an =.

Exercise

1) Given the following Grammar:

- (1) $S \rightarrow A$
- (2) $S \rightarrow B$
- (3) $A \rightarrow a A b$
- (4) $A \rightarrow 0$
- (5) $B \rightarrow a B b b$
- (6) $B \rightarrow 1$

A. Construct the SLR parsing table.

B. Write the action of an LR parse for the following string aa1bbbb

The Parser Generator: Yacc

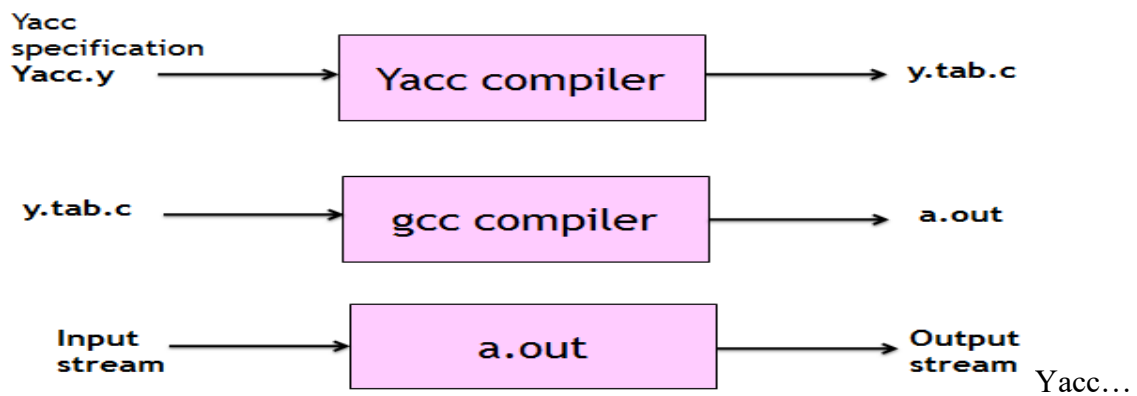
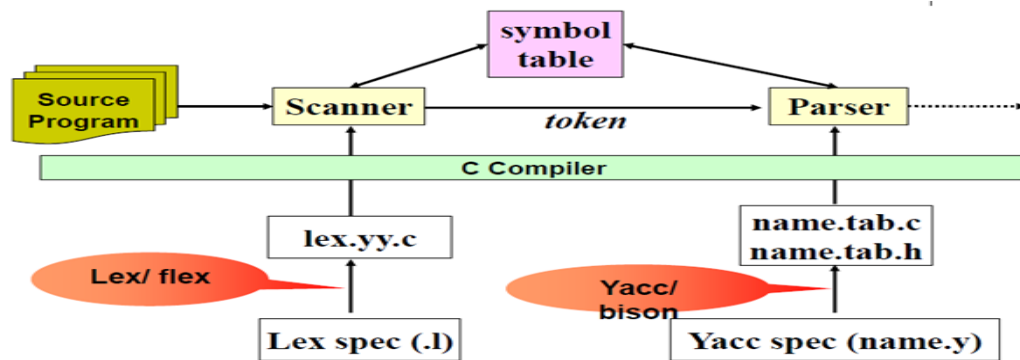
- ✚ **Yacc** stands for "yet another compiler-compiler". **Yacc**: a tool for automatically generating a parser given a grammar written in a yacc specification (.y file). **Yacc parser**

– calls lexical analyzer to **collect tokens** from input stream. Tokens are **organized** using grammar rules. When a rule is recognized, its **action is executed**

Note:

- o *lex* tokenizes the input and *yacc* parses the tokens, taking the right actions, in context.

Scanner, Parser, Lex and Yacc



✚ There are four steps involved in creating a compiler in Yacc:

- 1) Generate a parser from Yacc by running Yacc over the grammar file.
- 2) Specify the grammar:
 - o Write the grammar in a .y file (also specify the actions here that are to be taken in C).
 - o Write a lexical analyzer to process input and pass tokens to the parser. This can be done using Lex.
 - o Write a function that starts parsing by calling yyparse().
 - o Write error handling routines (like yyerror()).

- 3) Compile code produced by Yacc as well as any other relevant source files.
- 4) Link the object files to appropriate libraries for the executable parser.

Review Exercise

- 1) Consider the context-free grammar and the string: **aa + a***.
 - a) Give a leftmost derivation for the string.
 - b) Give a rightmost derivation for the string.
 - c) Give a parse tree for the string.
 - d) Is the grammar ambiguous or unambiguous? Justify your answer.
 - e) Describe the language generated by this grammar.
- 2) Design grammars for the following languages:
 - a) The set of all strings of 0s and 1s such that every 0 is immediately followed by at least one 1.
 - b) The set of all strings of 0s and 1s that are palindromes; that is, the string reads the same backward as forward.
 - c) The set of all strings of 0s and 1s with an equal number of 0s and 1s.
 - d) The set of all strings of 0s and 1s with an unequal number of 0s and 1s.
 - e) The set of all strings of 0s and 1s in which 011 does not appear as a substring.
 - f) The set of all strings of 0s and 1s of the form xy , where $x \neq y$ and x and y are of the same length.
- 3) The following is a grammar for regular expressions over symbols a and b only, using $+$ in place of $|$ for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

$$\begin{aligned} \text{rexpr} &\rightarrow \text{rexpr} + \text{rterm} \mid \text{rterm} \\ \text{rterm} &\rightarrow \text{rterm} \text{ rfactor} \mid \text{rfactor} \\ \text{rfactor} &\rightarrow \text{rfactor} * \mid \text{rprimary} \\ \text{rprimary} &\rightarrow a \mid b \end{aligned}$$

- a) Left factor this grammar.
 - b) Does left factoring make the grammar suitable for top-down parsing?
 - c) In addition to left factoring, eliminate left recursion from the original grammar.
 - d) Is the resulting grammar suitable for top-down parsing?
- 4) The grammar $S \rightarrow a S a \mid a a$ generates all even-length strings of a 's. We can devise a recursive-descent parser with backtrack for this grammar. If we choose to expand by production $S \rightarrow a a$ first, then we shall only recognize the string aa . Thus, any reasonable recursive-descent parser will try $S \rightarrow a S a$ first.
 - a) Show that this recursive-descent parser recognizes inputs aa , $aaaa$, and $aaaaaaaa$, but not $aaaaaa$.
 - b) What language does this recursive-descent parser recognize?
 - 5) Show that the following grammar is LL(1) but not SLR(1).

$S \rightarrow AaAb | BbBa$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

6) Show that the following grammar is SLR(1) but not LL(1).

$S \rightarrow SA | A$

$A \rightarrow a$

CHAPTER 4

Syntax-Directed Translation

Introduction

✚ Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent. Values of these attributes are evaluated by the **semantic rules** associated with the production rules.

✚ **Evaluation of these semantic rules:**

- may generate intermediate codes
- may put information into the symbol table
- may perform type checking
- may issue error messages
- may perform some other activities
- in fact, they may perform almost any activities.

Syntax-Directed Definitions and Translation Schemes

✚ When we associate semantic rules with productions, we use two notations:

➤ **Syntax-Directed Definitions**

➤ **Translation Schemes**

✚ **Syntax-Directed Definitions:**

- give high-level specifications for translations
- hide many implementation details such as order of evaluation of semantic actions.
- We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

✚ **Translation Schemes:**

- indicate the order of evaluation of semantic actions associated with a production rule.
- In other words, translation schemes give more information about implementation details.

Syntax-Directed Definitions

✚ A **syntax-directed definition** is a generalization of a context-free grammar in which:

- Each grammar symbol is associated with a set of **attributes**. This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol. Each production rule is associated with a set of **semantic rules**.

✚ An attribute can represent anything we choose:

- a string, a number, a type, a memory location, intermediate program representation etc...
- The value of a **synthesized attribute** is computed from the values of attributes at the children of that node in the parse tree.
- The value of an **inherited attribute** is computed from the values of attributes at the siblings and parent of that node in the parse tree.

✚ **Semantic rules** set up dependencies between attributes which can be represented by a *dependency graph*. This *dependency graph* determines the evaluation order of these semantic rules. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

✚ A *depth-first traversal* algorithm traverses the parse tree thereby executing semantic rules to assign attribute values. After the traversal is completed the attributes contain the translated form of the input.

✚ In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

$$b = f(c_1, c_2, \dots, c_n) \quad \text{where } f \text{ is a function, and } b \text{ can be one of the followings:}$$

➔ b is a synthesized attribute of A and c_1, c_2, \dots, c_n are attributes of the grammar symbols in the production

$$(A \rightarrow \alpha). \quad \text{For } A \rightarrow C \quad A.b = C.c$$

OR

➔ b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and c_1, c_2, \dots, c_n are attributes of the grammar symbols in the production $(A \rightarrow \alpha)$.
For $A \rightarrow C \quad C.c = A.b$

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**. The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree. Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Annotating a Parse Tree with Depth-First Traversals
procedure visit(n : node);

Begin

for each child m of n, from left to right **do**

visit(m);

evaluate semantic rules at node n

end

Example 4.1: Synthesized Attributed grammar that calculate the value of expression

<u>Production</u>	<u>Semantic Rules</u>
$L \rightarrow E \ n$	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

- It specifies a simple calculator that reads an input line containing an arithmetic expression involving:
 - digits, parenthesis, the operator + and *, followed by a new line character n, and
 - prints the value of expression.

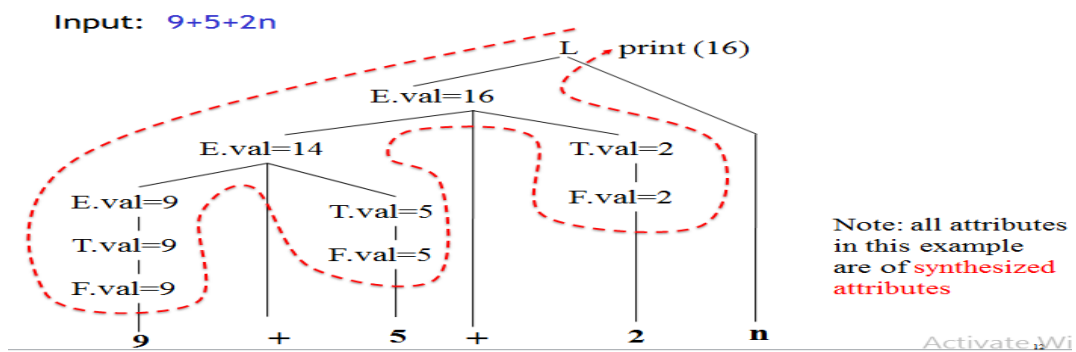
Example 4.2: Synthesized Attributed grammar that calculate the value of expression

<u>Production</u>	<u>Semantic Rules</u>
-------------------	-----------------------

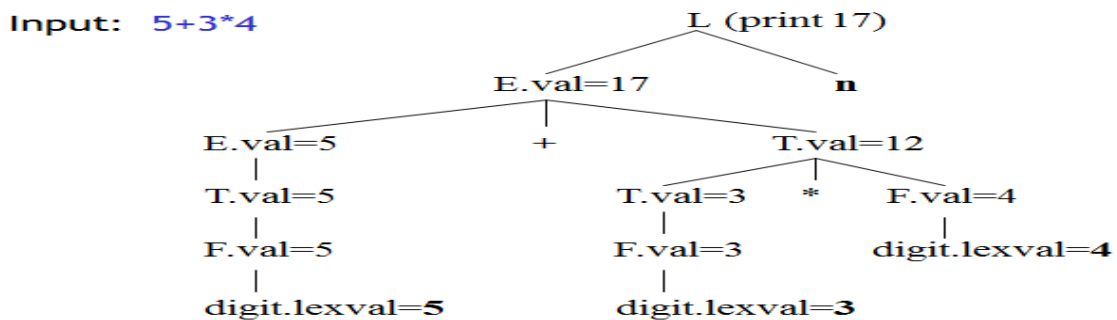
$L \rightarrow E n$	$\text{print}(E.\text{val})$	
$E \rightarrow E_1 + T$	$E.\text{val} = E_1.\text{val} + T.\text{val}$	
$E \rightarrow T$	$E.\text{val} = T.\text{val}$	
$T \rightarrow T_1 * F$	$T.\text{val} = T_1.\text{val} * F.\text{val}$	
$T \rightarrow F$	$T.\text{val} = F.\text{val}$	
$F \rightarrow (E)$	$F.\text{val} = E.\text{val}$	
$F \rightarrow \text{digit}$	$F.\text{val} = \text{digit}.\text{lexval}$	Input: 9+5+2n

✚ Symbols E, T, and F are associated with a synthesized attribute val. The token **digit** has a synthesized attribute lexval (it is assumed that it is evaluated by the lexical analyzer).

Depth-First Traversals: Example



Annotated Parse Tree: Example



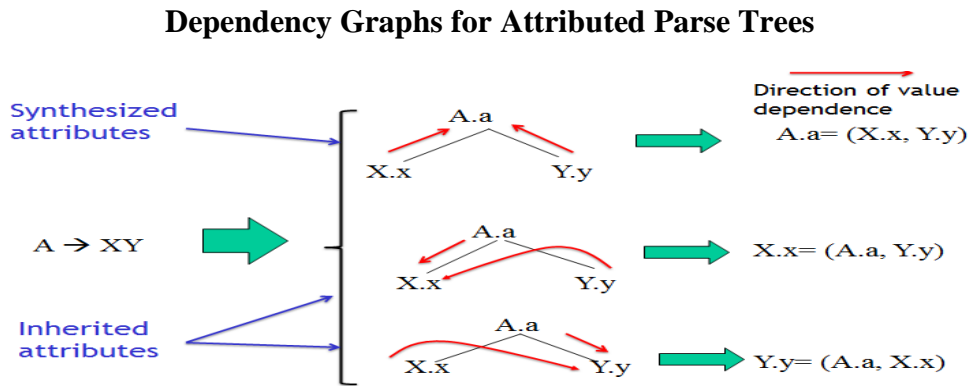
Exercise: 1) Synthesized Attributed grammar that calculate the value of expression

- Given the expression $5+3*4$ followed by new line, the program prints 17.
- Draw the decorated parse tree for input: $1*2n$

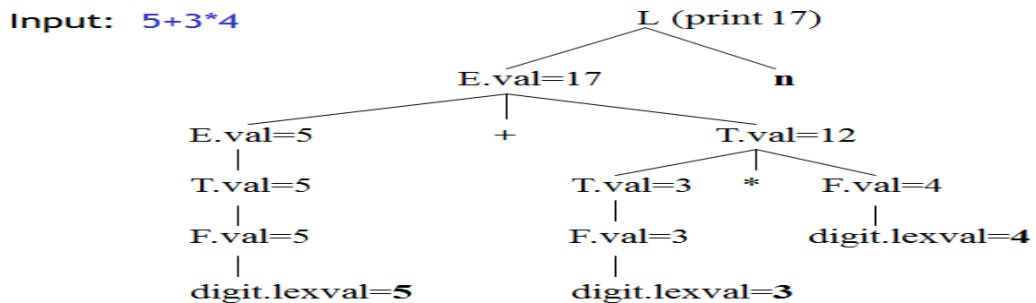
- c) Draw the annotated parse tree for input: $5*3+4n$
 - d) Draw the annotated parse tree for input: $5*(3+4)n$
- 2) Synthesized Attributed grammar that calculate the value of expression. By making use of SDD of example 4.2 give annotated parse trees for the following expressions:
- a) $(3+4) * (5+6)n$
 - b) $7*5*9*(4+5)n$
 - c) $(9+8*(7+6)+5)*4n$

Dependency Graphs for Attributed Parse Trees

✚ Annotated parse tree shows the values of attributes. Dependency graph helps us to determine how those values can be computed. The attributes should be evaluated in a given order because they depend on one another. The dependency of the attributes is represented by a dependency graph. $b(j) \text{ ----}D()\text{ ----} a(i)$ if and only if there exists a semantic action such as $a(i) := f(\dots b(j) \dots)$

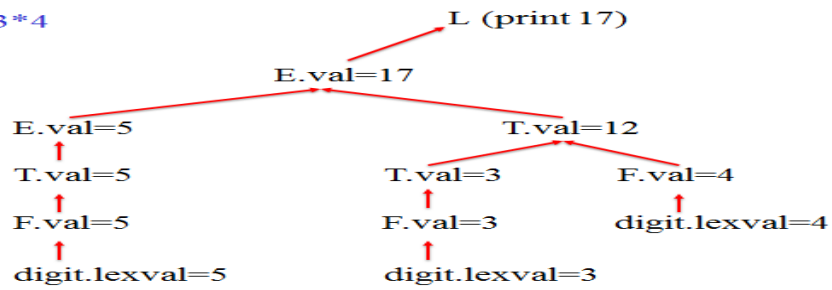


Annotated Parse Tree: Example



Dependency Graph

Input: $5+3*4$



val is synthesized attribute

Syntax-Directed Definition: Inherited Attributes

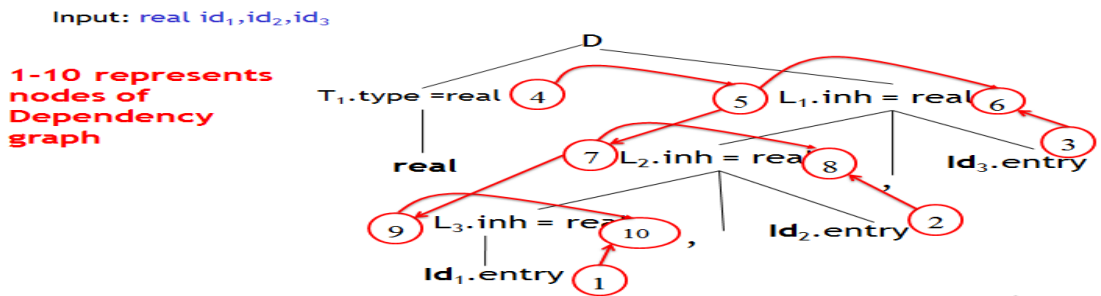
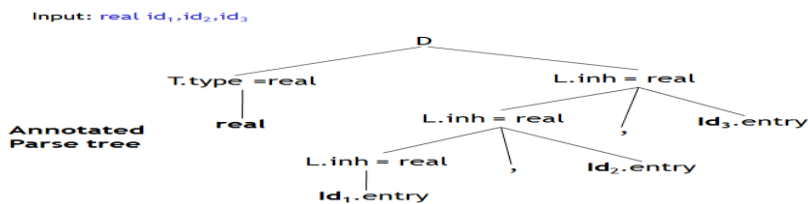
<u>Production</u>	<u>Semantic Rules</u>
$D \rightarrow T L$	$L.inh = T.type$
$T \rightarrow \text{int}$	$T.type = \text{integer}$
$T \rightarrow \text{real}$	$T.type = \text{real}$
$L \rightarrow L_1, \text{id}$	$L_1.inh = L.inh,$ $\text{addtype}(\text{id.entry}, L.inh)$
$L \rightarrow \text{id}$	$\text{addtype}(\text{id.entry}, L.inh)$

Inherited

synthesized

- Symbol T is associated with a synthesized attribute *type*. Symbol L is associated with an inherited attribute *inh*. *Input: real id₁, id₂, id₃*

A Dependency Graph – Inherited Attributes



SDD based on a grammar suitable for top-down parsing

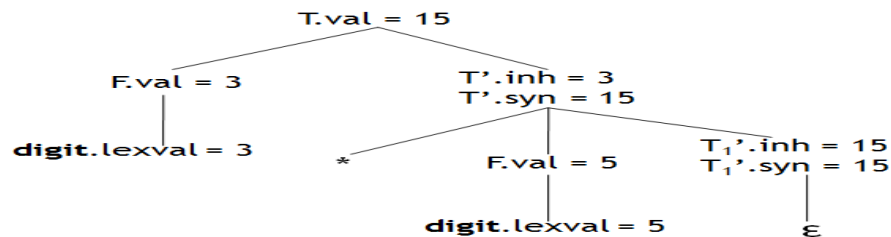
Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

✚ The SDD above computes terms like $3 * 5$ and $3 * 5 * 7$. Each of the non-terminals T and F has a synthesized attribute val ; The terminal **digit** has a synthesized attribute $lexval$.

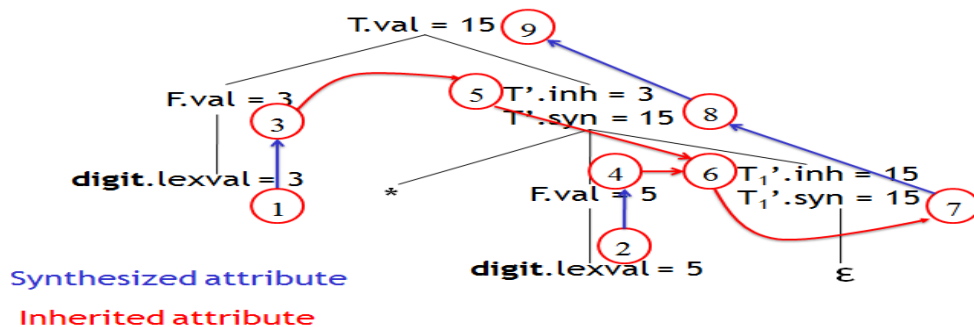
✚ The non-terminal T' has two attributes:

- an inherited attribute inh and
- a synthesized attribute syn .

Annotated parse tree for $3*5$



Dependency graph for the annotated parse tree of $3*5$



Exercises

Production

$N \rightarrow L1.L2$

$L1 \rightarrow L2 B$

$L \rightarrow B$

$B \rightarrow 0$

$B \rightarrow 1$

Semantic Rules

$N.v = L1.v + L2.v / (2^{L2.l})$

$L1.v = 2 * L2.v + B.v$

$L1.l = L2.l + 1$

$L.v = B.v$

$L1.l = 1$

$B.v = 0$

$B.v = 1$

✚ Draw the decorated parse tree and draw the dependency graph for input:

a - 1011.01

b - 11.1

c - 1001.001

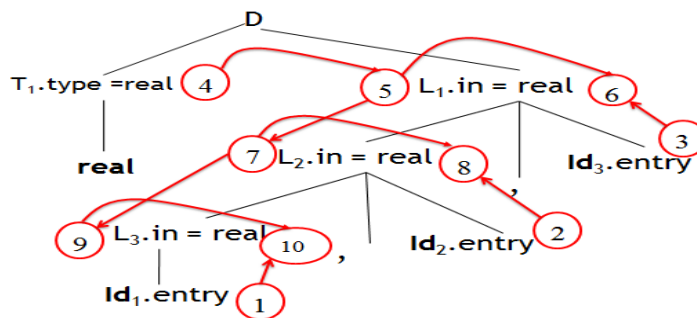
Evaluation Order

- ✚ A topological sort of a directed acyclic graph (DAG) is any ordering m_1, m_2, \dots, m_n of the nodes of the graph, such that: if $m_i \rightarrow m_j$ is an edge, then m_i appears before m_j
- ✚ Any topological sort of a dependency graph gives a valid evaluation order of the semantic rules.

Example Parse Tree with Topologically Sorted Actions

Topological sort:

1. Get $id_1.entry$
2. Get $id_2.entry$
3. Get $id_3.entry$
4. $T1.type = real$
5. $L1.in = T1.type$
6. $addtype(id_3.entry, L1.in)$
7. $L2.in = L1.in$
8. $addtype(id_2.entry, L2.in)$
9. $L3.in = L2.in$
10. $addtype(id_1.entry, L3.in)$



S-Attributed Definitions

- ✚ Syntax-directed definitions are used to specify syntax-directed translations that guarantee an evaluation order. We would like to evaluate the semantic rules during parsing (i.e. in a single pass, we will parse and we will also evaluate semantic rules during the parsing).

- ✚ We will look at two sub-classes of the syntax-directed definitions:
 - **S-Attributed Definitions:** only synthesized attributes used in the syntax-directed definitions.
 - **L-Attributed Definitions:** in addition to synthesized attributes, we may also use inherited attributes.
- ✚ These classes of SDD can be implemented efficiently in connection with top-down and bottom-up parsing.

S-Attributed Definitions

- ✚ A syntax-directed definition that uses synthesized attributes exclusively is called an *S-attributed definition (or S-attributed grammar)*. A parse tree of an S-attributed definition is annotated with a single bottom-up traversal.
- ✚ Bottom-up parser uses depth first traversal. A new stack is maintained to store the values of the attributes as in the following example. Yacc/Bison only support S-attributed definitions

Example: Attribute Grammar in Yacc

```

%{
#include <stdio.h>

void yyerror(char *);
%}

%token INTEGER

%%

program:
    program expr '\n'  { printf("%d\n", $2); }
    |
    ;

expr:
    INTEGER          { $$=$1;}
    | expr '+' expr  { $$ = $1 + $3; }
  
```

```

| expr '-' expr { $$ = $1 - $3; }
;
%%

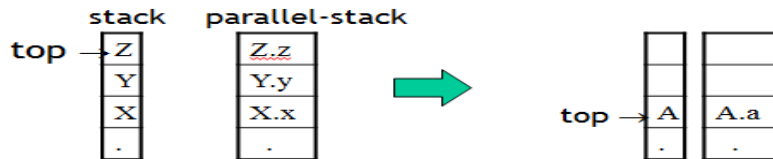
```

Synthesized attribute of parent node expr

Bottom-Up Evaluation of S-Attributed Definitions

- ✚ We put the values of the synthesized attributes of the grammar symbols into a parallel stack.
 - When an entry of the parser stack holds a grammar symbol X (terminal or non-terminal), the corresponding entry in the parallel stack will hold the synthesized attribute(s) of the symbol X.
- ✚ We evaluate the values of the attributes during reductions.

$A \rightarrow XYZ$ $A.a=f(X.x,Y.y,Z.z)$ where all attributes are synthesized.



Production

$L \rightarrow E n$

$E \rightarrow E_1 + T$

$E \rightarrow T$

$T \rightarrow T_1 * F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow \text{digit}$

Semantic Rules

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

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] + \text{val}[\text{top}]$

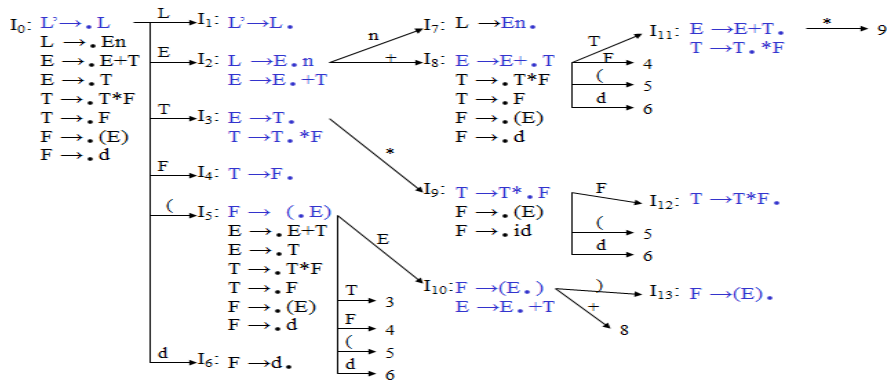
$$$ = \$1 + \$3; \text{ in yacc}$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-2] * \text{val}[\text{top}]$

$\text{val}[\text{ntop}] = \text{val}[\text{top}-1]$

- ✚ At each shift of **digit**, we also push **digit.lexval** into *val-stack*. At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Canonical LR(0) Collection for The Grammar



Bottom-Up Evaluation of S-Attributed Definitions in Yacc: Example

At each shift of **digit**, we also push **digit.lexval** into *val-stack*.

<u>stack</u>	<u>val-stack</u>	<u>input</u>	<u>action</u>	<u>semantic rule</u>
0		5+3*4n	s6	d.lexval(5) into val-stack
0d6	5	+3*4n	F→d	F.val=d.lexval – do nothing
0F4	5	+3*4n	T→F	T.val=F.val – do nothing
0T3	5	+3*4n	E→T	E.val=T.val – do nothing
0E2	5	+3*4n	s8	push empty slot into val-stack
0E2+8	5-	3*4n	s6	d.lexval(3) into val-stack
0E2+8d6	5-3	*4n	F→d	F.val=d.lexval – do nothing
0E2+8F4	5-3	*4n	T→F	T.val=F.val – do nothing
0E2+8T11	5-3	*4n	s9	push empty slot into val-stack
0E2+8T11*9	5-3-	4n	s6	d.lexval(4) into val-stack
0E2+8T11*9d6	5-3-4	n	F→d	F.val=d.lexval – do nothing
0E2+8T11*9F12	5-3-4	n	T→T*F	T.val=T ₁ .val*F.val
0E2+8T11	5-12	n	E→E+T	E.val=E ₁ .val+T.val
0E2	17	n	s7	push empty slot into val-stack
0E2n7	17-	\$	L→En	print(17), pop empty slot from val-stack
0L1	17	\$	acc	

CHAPTER 5

Type checking

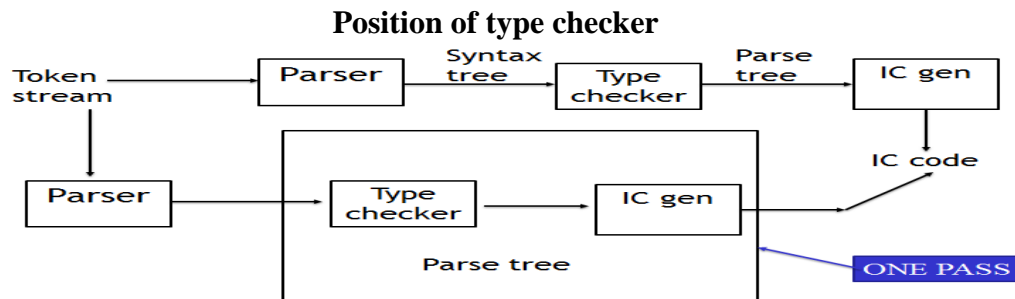
Introduction

✚ The compiler must check that the source program follows both the syntactic and semantic conventions of the source language.

✚ Semantic Checks

- Static – done during compilation
- Dynamic – done during run-time

✚ This checking is called *static checking* (to distinguish it from dynamic checking executed during execution of the target program). Static checking ensures that certain kind of errors will be detected and reported.



Static versus Dynamic Checking

Static checking: the compiler enforces programming language's static semantics

- Program properties that can be checked at compile time

Dynamic semantics: checked at run time

- Compiler generates verification code to enforce programming language's dynamic semantics

✚ Type checking is one of these static checking operations.

- We may not do all type checking at compile-time.
- Some systems also use dynamic type checking too.

✚ Why static checking?

✚ Parsing finds syntactic errors

- An input that can't be derived from the grammar

✚ **Static checking finds semantic errors**

- Calling a function with the wrong number/kind of arguments
- Applying operators to the wrong kinds of arguments
- Using undeclared variables
- Invalid conditions (not Boolean) in conditionals
- inappropriate instruction
 - ✓ return, break, continue used in wrong place

Other Static Checks

✚ A variety of other miscellaneous static checks can be performed

- Check for return statements outside of a function
- Check for case statements outside of a switch statement
- Check for duplicate cases in a case statement
- Check for break or continue statements outside of any loop
- Check for goto statements that jump to undefined labels
- Check for goto statements that jump to labels not in scope

✚ Most such checks can be done using 1 or 2 traversals of (part of) the parse tree

The Need for Type checking

✚ **We want to generate machine code**

✚ **Memory layout**

- Different data types have different sizes
 - ✓ In C, char, short, int, long, float, double usually have different sizes
 - ✓ Need to allocate different amounts of memory for different types

✚ **Choice of instructions**

- Machine instructions are different for different types
 - ✓ add (for i386 ints)
 - ✓ fadd (for i386 floats)

✚ One important kind of static checking is **type checking**

- Do operators match their operands? Do types of variables match the values assigned to them? Do function parameters match the function declarations? Have called function and variable names been declared?
- ✚ Not all languages can be completely type checked. All compiled languages must be at least partially type checked
- ✚ Type checking can be done bottom up using the parse tree. For convenience, we may create one or more pseudo-types for error handling purposes
 - Error type can be generated when a type checking error occurs
 - ✓ e.g., adding a number and a string
 - Unknown type can be generated when the type of an expression is unknown
 - ✓ e.g., an undeclared variable

Static checking

✚ **Typical examples of static checking are:**

- Type checks
- Flow-of-control checks
- Uniqueness checks...

Type checks:

- A compiler should report an error if an operator is applied to an incompatible operand.

Example: if an array variable and a function variable are added together.

```
int a, c[10],d;
d = c + d;
```

Flow of control check:

- Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control.
- Example: a break statement in C causes control to leave the smallest enclosing while, for, or switch statement.
- An error occurs if such an enclosing statement does not exist.

```

for(i=0;i<attempts;i++) {
    cout<<"Please enter your password:";
    Cin>>password;
    if(verify(password))
        break;//OK
    cout<<"incorrect\n";
}

```

Flow of control example...

```

foo ()
{
    ...
    break;//ERROR
}

```

```

foo ()
{...
    while (n)
    {...
        if (i>10)
            break;// OK
    }
}

```

```

foo ()
{...
    switch(a)
    { case 0:
        ...
        break;// OK
      case 1:
        ...
    }
}

```

Uniqueness check:

- Variables or objects must be defined exactly once.
- Example: in most PL, an identifier must be declared uniquely.

```

foo()
{
    int i, j, i;//ERROR
    ...
}

```

```

foo(int a, int a)//ERROR
{
    ...
}

```

```

struct myrec
{ int name;
};
struct myrec //ERROR
{int id;
};

```

One-Pass versus Multi-Pass Static Checking

- ✚ **One-pass compiler:** static checking in C, Pascal, Fortran, and many other languages is performed in one pass while intermediate code is generated
 - Influences design of a language: placement constraints
- ✚ **Multi-pass compiler:** static checking in Ada, Java, and C# is performed in a separate phase, sometimes by traversing a syntax tree multiple times. A separate type-checking pass between parsing and intermediate code generation.
- ✚ In this chapter, we focus on type checking. A type checker verifies that the type construct matches that expected by its context.
- ✚ For example:
 - The type checker should verify that the type value assigned to a variable is compatible with the type of the variable.
 - Built in operator mod requires integer operands.
 - Indexing is done only to array.
 - Dereference to pointer
 - A user-defined function is applied to the correct number and type of arguments...

Type systems

- ✚ A type system is a collection of rules for assigning type expressions to the parts of a program. A type checker implements a type system. A *sound* type system eliminates run-time type checking for type errors.
- ✚ A programming language is *strongly-typed*, if every program its compiler accepts will execute without type errors.
 - In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly-typed).
 - Ex: `int x [100]; ... x[i]` → most of the compilers cannot guarantee that `i` will be between 0 and 99

Type expressions

- ✚ The type of a language construct is denoted by a type expression.
- ✚ **A type expression is either:**

- a basic type or
- Formed by applying an operator called type constructor to the type expression.

✚ **The followings are type expressions:**

- A basic type is a type expression:

Example: Boolean, char, integer, and real

- A special basic type, *type_error*, will signal an error during type checking.
- A basic type *void* denoting “the absence of a value” allows statement to be checked.

✚ **The following are type constructors:**

- **Arrays:** if I in an index set and T is a type expression, then Array (I, T) is a TE:

In java, `int[] A = new int[10];`

In C++ `int A[10];`

In pascal `var A: array [1..10] of integer;` associates the type expression `array(1..10, integer)` with A.

- **Products:** if T1 and T2 are type expressions, the Cartesian product T1 x T2 is a TE. X is left associative.

Example: `foo(int, char), int x char` → (1, 'a'), (2, 'b')...

- **Pointers:** if T is a TE then pointer (T) is a TE. Denotes the type “pointer to an object of type T.”

`var p: ^integer` → `pointer(integer)`

`int *a;`

✚ **Functions:** the TE of a function has the form D → R where:

- D is the type expression of the parameters and
- R is the TE of the returned value.

✚ For example:

- mod function has domain type `int x int`, a pair of integers and range type `int`, thus `mod` has `int x int` → `int`

- ✚ The TE corresponding to the Pascal declaration:

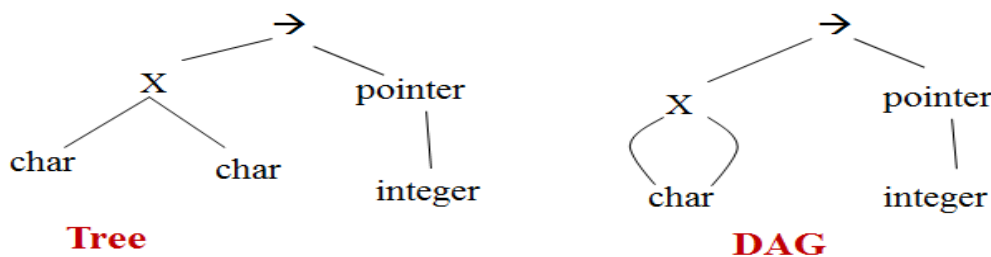
function f (a, b : char): ^Integer:

char x char → pointer (integer)

- ✚ A convenient way to represent a type expression is to use a graph (tree or DAG).

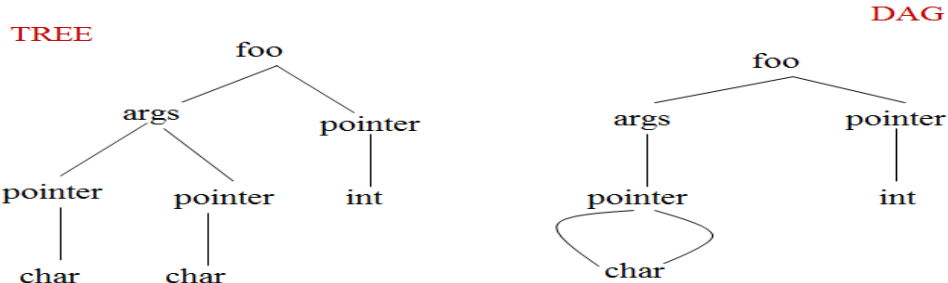
For example, the type expression corresponding to the above function declaration can be represented with the tree shown below:

TE: char x char → pointer (integer)



Example: tree and DAG

int *foo(char *, char *)



Type Expression (summary)

- ✚ The type of a language construct is denoted by a *type expression*.
- ✚ A *type expression* can be:
 - **A basic type**
 - ✓ a primitive data type such as *integer*, *real*, *char*, *boolean*, ...
 - ✓ *type-error* to signal a type error
 - ✓ *void* : no type

- **A type name**
 - ✓ a name can be used to denote a type expression.
- **A type constructor applies to other type expressions.**
 - ✓ **arrays:** If T is a type expression, then $array(I,T)$ is a type expression where I denotes index range. Ex: $array(0..99,int)$
 - ✓ **products:** If T_1 and T_2 are type expressions, then their cartesian product $T_1 \times T_2$ is a type expression. Ex: $int \times int$
 - ✓ **pointers:** If T is a type expression, then $pointer(T)$ is a type expression. Ex: $pointer(int)$
 - ✓ **functions:** We may treat functions in a programming language as mapping from a domain type D to a range type R . So, the type of a function can be denoted by the type expression $D \rightarrow R$ where D and R are type expressions. Ex: $int \rightarrow int$ represents the type of a function which takes an int value as parameter, and its return type is also int .

Specification of a simple type checker

✚ In this section, we specify a type checker for a simple language. The type of each identifier must be declared before the identifier is used. The type checker is a translation scheme:

- Synthesizes the type of each expression from the types of its sub expressions.

✚ The type checker can handle:

- arrays,
- pointers,
- statements, and
- Functions.

A Simple Language example

✚ This grammar generates programs, represented by the non-terminal P consisting of sequence of declarations D followed by a single expression E or statement S .

$$P \rightarrow D ; E \mid D ; S$$

$$D \rightarrow D ; D \mid \text{id} : T$$

$$T \rightarrow \text{boolean} \mid \text{char} \mid \text{integer} \mid \text{array} \{ \text{num} \} \text{ of } T \mid \wedge T$$

$$E \rightarrow \text{true} \mid \text{false} \mid \text{literal} \mid \text{num} \mid \text{id} \mid E \text{ mod } E \mid E [E] \mid E \wedge$$

$$\mid E = E \mid E + E$$

$$S \rightarrow \text{id} := E \mid \text{if } E \text{ then } S \mid \text{while } E \text{ do } S \mid S ; S$$

✚ One program generated by the grammar is:

```
key : integer;
key mod 100
```

Types in the language

✚ The language has three basic types: *boolean*, *char* and *integer*. *Type_error* used to signal error. *Void* used to check statements. All arrays start at 1. **For example:**

array [256] of char => leads to the TE array (1...256, char)

✚ Consisting of the constructor *array* applied to the sub range 1...256 and the type *char*. The prefix operator \wedge in declarations builds a pointer type, so $\wedge \text{integer}$ leads to the TE pointer (*integer*), consisting of the constructor *pointer* applied to the type *integer*.

Specification of a simple type checker

✚ **Translation schemes for Declarations**

$$P \rightarrow D ; E \mid D ; S$$

$$D \rightarrow D ; D$$

$$D \rightarrow \text{id} : T \quad \{ \text{addtype} (\text{id.entry}, T. \text{Type}) \}$$

$$T \rightarrow \text{boolean} \quad \{ T.\text{type} := \text{boolean} \}$$

$$T \rightarrow \text{char} \quad \{ T.\text{type} := \text{char} \}$$

$$T \rightarrow \text{integer} \quad \{ T.\text{type} := \text{integer} \}$$

$$T \rightarrow \wedge T1 \quad \{ T.\text{type} := \text{pointer} (T1.\text{type}) \}$$

$$T \rightarrow \text{array} [\text{num}] \text{ of } T1 \quad \{ T.\text{type} := \text{array} (1 \dots \text{num.val}, T1.\text{type}) \}$$

✚ The purpose of the above semantic actions is:

- to synthesize the type expression corresponding to a declaration of a type and

- add the type expression in the symbol table entry corresponding to the variable identifier.

✚ Translation scheme for type checking of Expressions:

$E \rightarrow \text{true}$ {E.type:= boolean}
 $E \rightarrow \text{false}$ {E.type:= boolean}
 $E \rightarrow \text{literal}$ {E.type := char}
 $E \rightarrow \text{num}$ {E.type := integer}
 $E \rightarrow \text{id}$ {E.type := lookup (id.entry)}
 $E \rightarrow E_1 \text{ mod } E_2$ {E.type = **if** E1.type = integer **and** E2.type := integer **then**
 integer
 else type_error}
 $E \rightarrow E_1 [E_2]$ {E.type := **if** E2.type = integer **and** E1.type := array (s,t) **then**
 t
 else type_error}
 $E \rightarrow E_1 \wedge$ {E.type := **if** E1.type = pointer (t) **then** t
 else type_error}

The objects pointed to by its operand

$E \rightarrow E_1 + E_2$ { **if** (E₁.type=int and E₂.type=int) **then** E.type=int
 else if (E₁.type=int and E₂.type=real) **then** E.type=real
 else if (E₁.type=real and E₂.type=int) **then** E.type=real
 else if (E₁.type=real and E₂.type=real) **then** E.type=real
 else E.type=type-error }
 $E \rightarrow E_1 = E_2$ { E.type := **if** E1.type = boolean **and** E2.type =
 boolean **then** boolean
 else type_error}

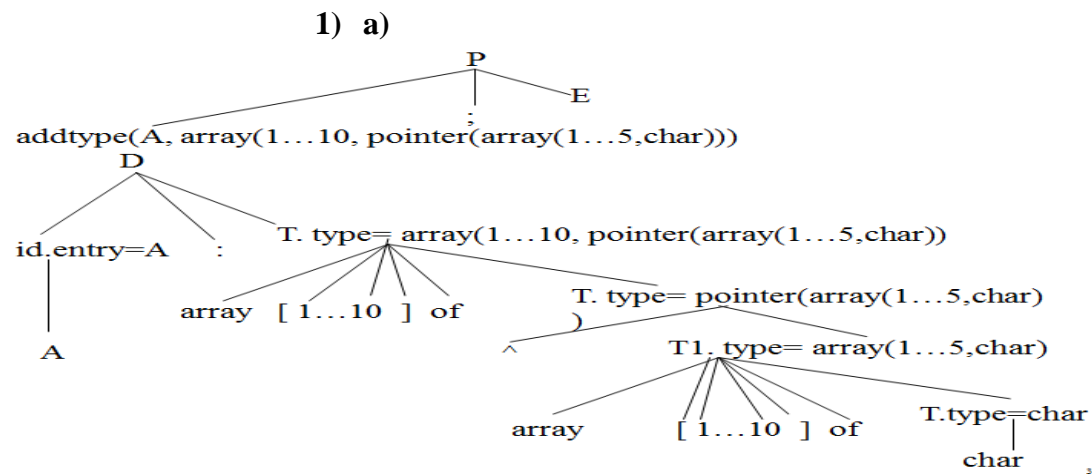
Translation scheme for type checking of Statements:

- $S \rightarrow id := E \quad \{S.type := \text{if } id.type = E.type \text{ then void} \\ \text{else type_error} \}$
- $S \rightarrow \text{if } E \text{ then } S1 \quad \{S.type := \text{if } E.type = \text{boolean} \text{ then} \\ S1.type \\ \text{else type_error} \}$
- $S \rightarrow \text{while } E \text{ do } S1 \quad \{S.type := \text{if } E.type = \text{boolean} \text{ then} \\ S1.type \\ \text{else type_error} \}$
- $S \rightarrow S1 ; S2 \quad \{S.type := \text{if } S1.type = \text{void} \text{ and } S2.type = \text{void} \\ \text{then void} \\ \text{else type_error} \}$

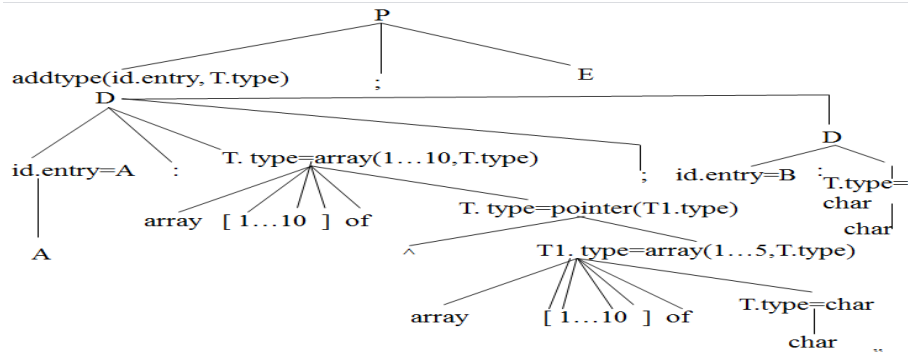
Exercises:

- 1) For the translation scheme of a simple type checker presented above, draw the decorated parse tree for:
 - a) A: array [1...10] of ^array [1...5] of char
 - b) A: array [1...10] of ^ array [1...5] of char; B: char

Solutions:



1) b)



2) For the translation scheme of a simple type checker presented above, draw the decorated parse tree for:

a: array [1...5] of integer;

b: integer;

c: char;

b=a[1];

if b = a[10] then

b = b + 1;

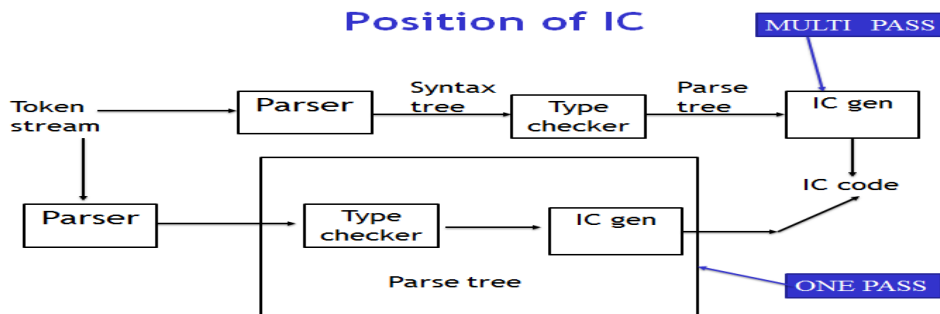
c = a[5];

CHAPTER 6

Intermediate Code Generation

Intermediate Representations

- ✚ In a compiler, the front end translates source program into an intermediate representation, and the back end generates the target code from this intermediate representation. The use of a machine independent intermediate code (IC) is:
 - retargeting to another machine is facilitated
 - the optimization can be done on the machine independent code
- ✚ Type checking is done in another pass → **Multi – pass**. IC generation and type checking can be done at the same time → **one pass**.
- ✚ Decisions in IR design affect the speed and efficiency of the compiler. Some important IR properties:
 - Ease of generation
 - Ease of manipulation
 - Procedure size
 - Level of abstraction
- ✚ The importance of different properties varies between compilers
 - Selecting an appropriate IR for a compiler is critical



Intermediate Code Generation

- ✚ Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
 - Syntax tree can be used as an intermediate language.

- Postfix notation can be used as an intermediate language.
- Three-address code (Quadruples) can be used as an intermediate language
 - ✓ We will use three address to discuss intermediate code generation
 - ✓ Three addresses are close to machine instructions, but they are not actual machine instructions.
- Some programming languages have well defined intermediate languages.
 - ✓ java – java virtual machine
 - ✓ prolog – warren abstract machine
 - ✓ In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Types of Intermediate Representations

✚ Three major categories:

- Structural
 - ✓ Graphically oriented
 - ✓ Heavily used in source-to-source translators
 - ✓ Tend to be large
 - ✓ **Examples:** Trees, DAG
- Linear
 - ✓ Pseudo-code for an abstract machine
 - ✓ Level of abstraction varies
 - ✓ Simple, compact data structures
 - ✓ Easier to rearrange
 - ✓ **Examples:** 3 address code and Stack machine code
- Hybrid
 - ✓ Combination of graphs and linear code
 - ✓ Example: control-flow graph

Intermediate languages

Syntax tree

✚ While parsing the input, a syntax tree can be constructed. A syntax tree (abstract tree) is a condensed form of parse tree useful for representing language constructs. For example,

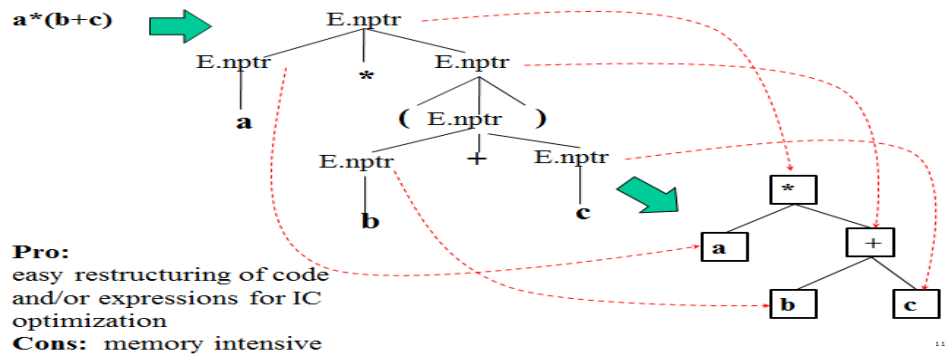
for the string **a+b**, the parse tree in (a) below can be represented by the syntax tree shown in (b); the keywords (syntactic sugar) that existed in the parse tree will no longer exist in the syntax tree.



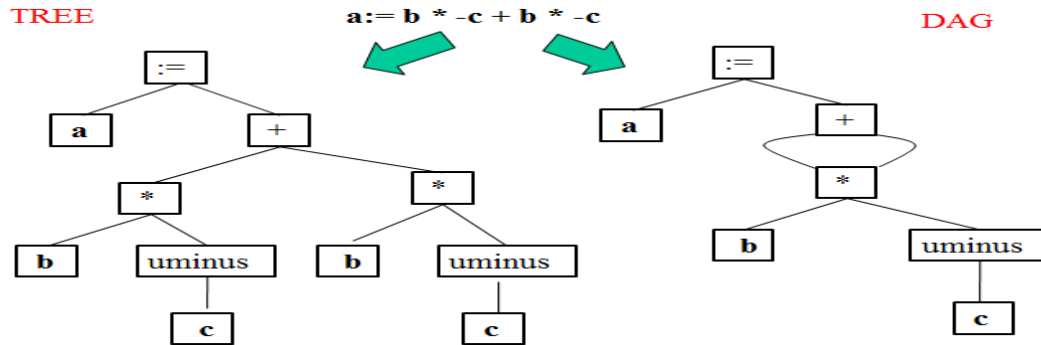
Syntax-Directed Translation of Abstract Syntax Trees

<u>Production</u>	<u>Semantic Rules</u>
$S \rightarrow \mathbf{id} := E$	$S.nptr := \text{mknode}(':=', \text{mkleaf}(\mathbf{id}, \mathbf{id.entry}), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := \text{mknode}('+', E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := \text{mknode}('*', E_1.nptr, E_2.nptr)$
$E \rightarrow - E_1$	$E.nptr := \text{mknode}(\text{'uminus'}, E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$E \rightarrow \mathbf{id}$	$E.nptr := \text{mkleaf}(\mathbf{id}, \mathbf{id.entry}) \dots\dots\dots ***$

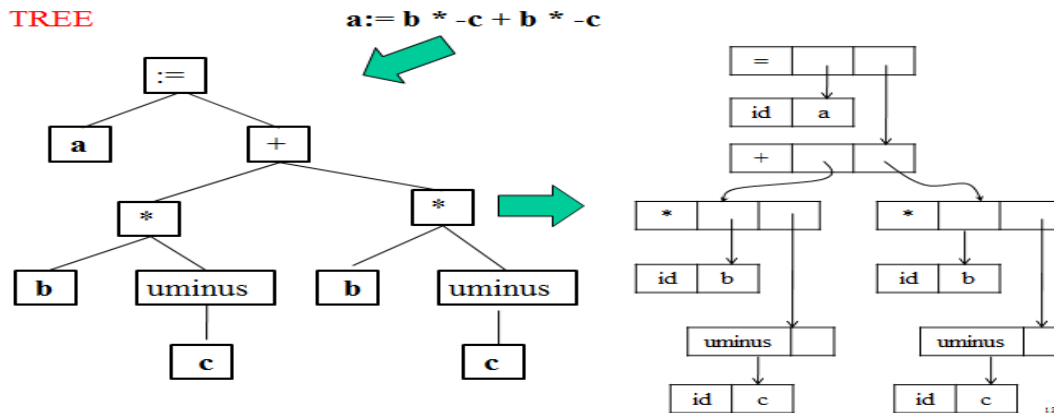
Abstract Syntax Trees



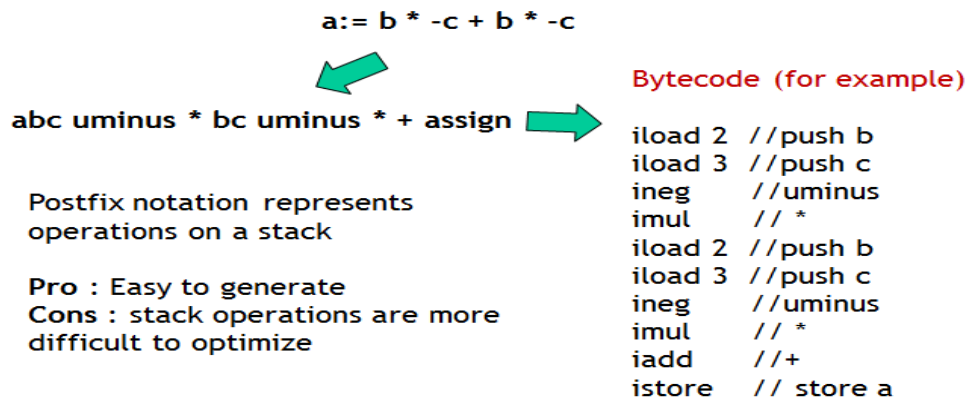
Abstract Syntax Trees versus DAGs



Syntax Tree representation



Postfix notation



Stack Machine Code

Originally used for stack-based computers, now Java

Example:

13+5* becomes

push 1
push 3
add
push 5
multiply

The value on the top of the stack at z end (here 20) is a value of the entire expression

Pros:

- Compact form
- Introduced names are **implicit**, not explicit
- Simple to generate and execute code

Implicit names take up no space, where explicit ones do!

Three-Address Code

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

t1 := -c
 t2 := b * t1
 t3 := -c
 t4 := b * t3
 t5 := t2 + t4
 a := t5

Linearized representation of a syntax tree

t1 := -c
 t2 := b * t1
 t5 := t2 + t2
 a := t5

Linearized representation of a syntax DAG

✚ A three address code is: $x := y \text{ op } z$ where x , y and z are names, constants or compiler-generated temporaries; **op** is any operator. But we may also use the following notation for three address code (much better notation because it looks like a machine code instruction)

op y,z,x apply operator **op** to **y** and **z**, and store the result in **x**.

✚ We use the term “three-address code” because each statement usually contains three addresses (two for operands, one for the result).

✚ **In three-address code:**

- Only one operator at the right side of the assignment is possible, i.e. $x + y * z$ is not possible
- Similar to postfix notation, the three address code is a linear representation of a syntax tree.
- It has been given the name three-address code because such an instruction usually contains three addresses (the two operands and the result)

$$t_1 = y * z$$

$$t_2 = x + t_1$$

Three-Address Statements

Binary Operator:

op y,z,result or **result := y op z**

- Where op is a binary arithmetic or logical operator. This binary operator is applied to y and z, and the result of the operation is stored in result.

Ex: *add a,b,c*
 mul a,b,c
 addr a,b,c
 addi a,b,c

Unary Operator:

op y,, result or **result := op y**

- Where op is a unary arithmetic or logical operator. This unary operator is applied to y, and the result of the operation is stored in result.

Ex: *uminus a,,c*
 not a,,c
 inttoreal a,,c

Copy/ Move Operator:

mov y,,result or **result := y** where the content of y is copied into **result**.

Ex: **mov a,,c**
 movi a,,c
 movr a,,c

Unconditional Jumps:

jmp ,,L or **goto L**

- ✚ We will jump to the three-address code with the label L, and the execution continues from that statement.

Ex: **jmp** ,,L1 // jump to L1
 jmp ,,7 // jump to the statement 7

Conditional Jumps:

jmprelop y,z,L or **if y relop z goto L**

- ✚ We will jump to the three-address code with the label L if the result of **y relop z** is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Ex: **jmpgt** y,z,L1 // jump to L1 if y>z
 jmpgte y,z,L1 // jump to L1 if y>=z
 jmpe y,z,L1 // jump to L1 if y==z
 jmpne y,z,L1 // jump to L1 if y!=z


- ✚ Our relational operator can also be a unary operator.

jmpnz y,,L1 // jump to L1 if y is not zero
jmpz y,,L1 // jump to L1 if y is zero
jmpnt y,,L1 // jump to L1 if y is true
jmpf y,,L1 // jump to L1 if y is false

Procedure Parameters:

param x,, or **param** x

Procedure Calls: **call** p,n, or **call** p,n where x is an actual parameter, we invoke the procedure p with n parameters.

Ex: **param** x1,,
 param x2,,
  **p(x1,...,x_n)**
 param x_n,,
 call p,n,

$f(x+1,y) \longrightarrow \text{add } x,1,t_1$

param $t_1,$

param $y,$

call $f,2,$

Indexed Assignments:

move $y[i],,x$ or $x := y[i]$

move $x,,y[i]$ or $y[i] := x$

Address and Pointer Assignments:

moveaddr $y,,x$ or $x := \&y$

movecont $y,,x$ or $x := *y$

Three Address Statements (summary)

- Assignment statements: $x := y \text{ op } z, x := \text{op } y$
- Indexed assignments: $x := y[i], x[i] := y$
- Pointer assignments: $x := \&y, x := *y, *x := y$
- Copy statements: $x := y$
- Unconditional jumps: **goto** L
- Conditional jumps: **if** $y \text{ relop } z$ **goto** L
- Function calls: **param** $x... \text{ call } p, n$

✓ **return** y

Syntax-Directed Translation into Three-Address Code

✚ Syntax directed translation can be used to generate the three-address code. Generally, either:

- the three-address code is generated as an attribute of the attributed parse tree or
- the semantic actions have side effects that write the three-address code statements in a file.

✚ When the three-address code is generated, it is often necessary to use temporary variables and temporary names. The following functions are used to generate 3-address code:

newtemp() - each time this function is called, it gives distinct names that can be used for temporary variables.

- returns t_1, t_2, \dots, t_n in response to successive calls

newlabel() - each time this function is called, it gives distinct names that can be used for label names.

gen() to generate a single three address statement given the necessary information.

- variable names and operations.

✚ **gen** will produce a three-address code after concatenating all the parameters.

✚ For example: If **id1.lexeme = x**, **id2.lexeme = y** and **id3.lexeme = z**:

gen (id1.lexeme, ':=', id2.lexeme, '+', id3.lexeme) will produce the three-address code : **x := y + z**

✚ **Note:** variables and attribute values are evaluated by **gen** before being concatenated with the other parameters.

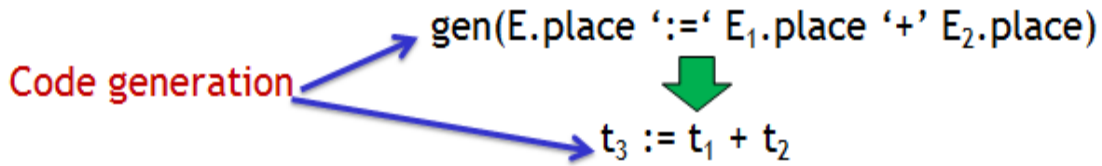
✚ Use attributes:

E.place: the name that will hold the value of E.

- Identifier will be assumed to already have the place attribute defined.

E.code: hold the three address code statements that evaluate E (this is the 'translation' attribute).

<u>Production</u>	<u>Semantic Rules</u>
$S \rightarrow \mathbf{id} := E$	S.code three address code for S
while E do S	S.begin lable to start of S or nil
$E \rightarrow E + E$	S.after lable to end of S or nil
$E * E$	E.code three-address code for E
$- E$	E.place a name holding the value of E
(E)	
id	
num	



Implementation of Three-Address Statements

The description of three-address instructions specifies the components of each type of instruction. However, it does not specify the representation of these instructions in a data structure. In a compiler, these statements can be implemented as objects or as records with fields for the operator and the operands.

Three such representations are:

- Quadruples
- Triples and
- Indirect triples

Quadruples: A quadruple (or just "quad") has four fields, which we call op, arg₁, arg₂, and result

Triples: A triple has only three fields, which we call op, arg₁, and arg₂.

Indirect Triples: consists of a listing of pointers to triples, rather than a listing of triples themselves.

The 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 to change all references to that result. *This problem does not occur with indirect triples.*

Implementation of Three-Address Statements: Quads

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

Three address code

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

The original FORTRAN compiler used "quads"

Quads (quadruples)

#	op	Arg1	Arg2	Res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Pro: easy to rearrange code for global optimization, explicit names
Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

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

Three address code

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Implicit names occupy
no space

Triples

#	op	Arg1	Arg2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Pro: temporaries are implicit
25% less space consumed than quads
Cons: difficult to rearrange codes

More triplet representations

#	op	Arg1	Arg2
(0)	[]=	x	i
(1)	assign	(0)	y

$x [i] = y$

#	op	Arg1	Arg2
(0)	= []	y	i
(1)	assign	x	(0)

$x = y [i]$

✚ Major tradeoff between quads and triples is compactness versus ease of manipulation

- In the past compile-time and space was critical
- Today, speed may be more important

Implementation of Three-Address Statements: Indirect Triples

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

Pointers to Triples

#	stmt	#	op	Arg1	Arg2
(0)	(14)	(14)	uminus	c	
(1)	(15)	(15)	*	b	(14)
(2)	(16)	(16)	uminus	c	
(3)	(17)	(17)	*	b	(16)
(4)	(18)	(18)	+	(15)	(17)
(5)	(19)	(19)	:=	a	(18)

Pro: temporaries are implicit & easier to rearrange code
Cons: Uses more space than triples

Exercises

✚ Translate the arithmetic expression $a + -(b + c)$ into

- A syntax tree and DAG.

b) Quadruples.

c) Triples.

d) Indirect triples by making use of translation scheme given in*** above.

✚ Three address code for an assignment statement and an expression

Productions

Semantic actions

$S \rightarrow \mathbf{id} := E$ $S.code := E.code \parallel \text{gen}(\text{id.lexeme } ' := ' E.place); S.begin = S.after = \text{nil}$

$E \rightarrow E1 + E2$ $E.place := \text{newtemp}();$
 $E.code := E1.code \parallel E2.code \parallel \text{gen}(E.place, ' := ', E1.place, '+', E2.place)$

$E \rightarrow E1 * E2$ $E.place := \text{newtemp}();$
 $E.code := E1.code \parallel E2.code \parallel \text{gen}(E.place, ' := ', E1.place, '*', E2.place)$

$E \rightarrow - E1$ $E.place := \text{newtemp}();$
 $E.code := E1.code \parallel \text{gen}(E.place, ' := \text{uminus } ' E1.place)$

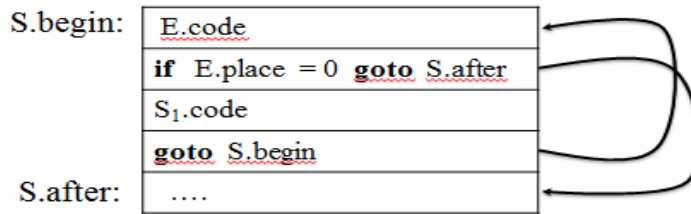
$E \rightarrow (E1)$ $E.place := E1.place$
 $E.code := E1.code$

$E \rightarrow \mathbf{id}$ $E.place := \text{id.lexeme}$
 $E.code := ' /* empty code */$

$E \rightarrow \mathbf{num}$ $E.place := \text{newtemp}();$
 $E.code := \text{gen}(E.place ' := ' \mathbf{num}.value)$

✚ Three address code for an assignment statement and an expression

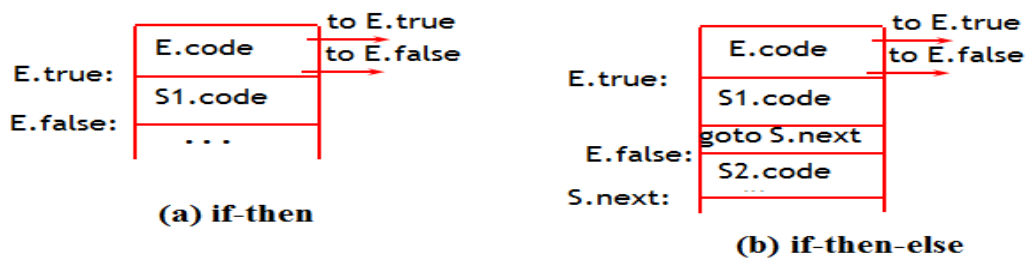
$S \rightarrow \mathbf{while} E \mathbf{do} S_1$ $S.begin = \text{newlabel}();$
 $S.after = \text{newlabel}();$
 $S.code = \text{gen}(S.begin " :") \parallel E.code \parallel$
 $\text{gen}(' \text{if } E.place ' = ' 0 ' \text{goto } S.after) \parallel S_1.code \parallel$
 $\text{gen}(' \text{goto } S.begin) \parallel$
 $\text{gen}(S.after " :")$



$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$ $S.\text{else} = \text{newlabel}();$
 $S.\text{after} = \text{newlabel}();$
 $S.\text{code} = E.\text{code} \parallel$
 $\text{gen}(\text{'if' } E.\text{place} \text{'='} 0 \text{' goto' } S.\text{else}) \parallel$
 $S_1.\text{code} \parallel$
 $\text{gen}(\text{'goto' } S.\text{after}) \parallel$
 $\text{gen}(S.\text{else} \text{' :'}) \parallel S_2.\text{code} \parallel$
 $\text{gen}(S.\text{after} \text{' :'})$

$E \rightarrow E < E$ $E.\text{place} = \text{newtemp}();$
 $E.\text{code} = E_1.\text{code} \parallel E_2.\text{code} \parallel$
 $\text{gen}(E.\text{place}, \text{'='}, E_1.\text{place}, \text{'<'}, E_2.\text{place})$

Code for flow-of-control statements



Syntax-Directed Translation (cont.)

$S \rightarrow \text{while } E \text{ do } S_1$ $S.\text{begin} = \text{newlabel}();$
 $S.\text{after} = \text{newlabel}();$
 $S.\text{code} = \text{gen}(S.\text{begin} \text{' :'}) \parallel E.\text{code} \parallel$


```

gen('jmpf' E.place ',', S.after) || S1.code ||
gen('jmp' ',', S.begin) ||
gen(S.after ':')

```

S → if E then S₁ else S₂ S.else = newlabel();

 S.after = newlabel();

 S.code = E.code ||

```

gen('jmpf' E.place ',', S.else) || S1.code ||
gen('jmp' ',', S.after) ||
gen(S.else ':') || S2.code ||
gen(S.after ':')

```

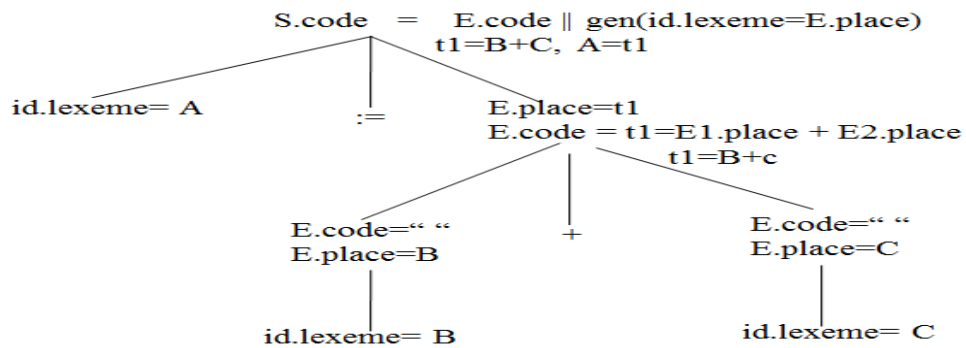
Exercises:

1) Draw the decorated parse tree and generate three-address code by using the translation schemes given:

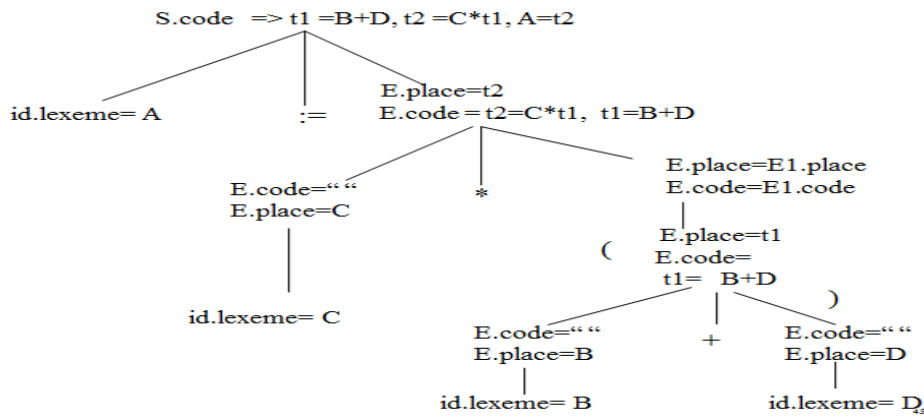
- a) A := B + C
- b) A := C * (B + D)
- c) while a < b do a := (a + b) * c
- d) while a < b do a := a + b
- e) a:= b * -c + b * -c

Solutions for:

Three address code of A := B + C



Three address code of A := C * (B + D)



**while a < b
a = (a + b) * c**



**L1: t1 := a < b
if t1 = 0 goto L2
t2 := a + b
t3 := t2 * c
a := t3
goto L1
L2:**

How come ? Draw the decorated parse tree

**i := 2 * n + k
while i do
i := i - k**



**t1 := 2
t2 := t1 * n
t3 := t2 + k
i := t3
L1: if i = 0 goto L2
t4 := i - k
i := t4
goto L1
L2:**

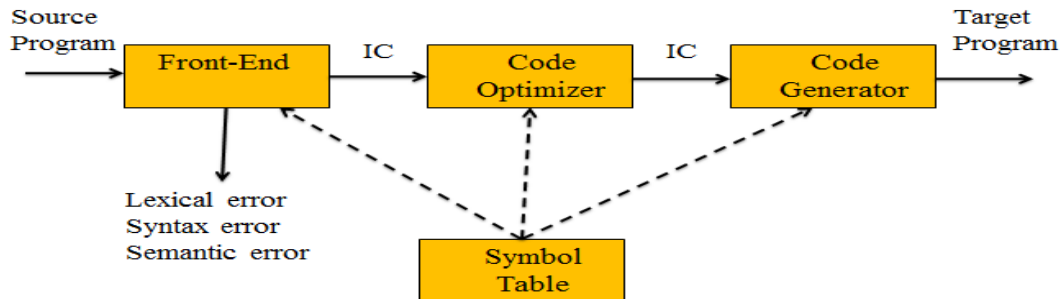
Note: Please do for the rest questions.

CHAPTER 7

Code Generation

Introduction

Position of a Code Generator



✚ The final phase in our compiler model is **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**.

✚ Requirements imposed on a code generator

- Preserving the semantic meaning of the source program and being of high quality
- Making effective use of the available resources of the target machine
- The code generator itself must run efficiently.

✚ A code generator has three primary tasks:

- Instruction selection, register allocation and assignment, and instruction ordering

Issue in the Design of a Code Generator

✚ General tasks in almost all code generators:

- instruction selection,
- register allocation and assignment and
- instruction ordering

✚ The details are also dependent on:

- the specifics of the intermediate representation,
- the target language, and

- the run-time system.

✚ The most important criterion for a code generator is that it should produce correct code.

✚ Most serious issues in the design of a code generator are:

- Input to the Code Generator
- The Target Program
- Instruction Selection
- Register Allocation
- Choice of Evaluation Order

Input to the Code Generator

✚ **The input to the code generator is**

- the intermediate representation of the source program produced by the frontend along with
- information in the symbol table that is used to determine the run-time address of the data objects denoted by the names in the IR.

✚ **Choices for the IR**

- Three-address representations: quadruples, triples, indirect triples
- Virtual machine representations: such as byte codes and stack-machine code
- Linear representations: such as postfix notation
- Graphical representation: such as syntax trees and DAG's

✚ **Assumptions**

- Front end has scanned, parsed and translated into relatively lower level IR
- All syntactic and static semantic errors are detected.

The Target Program

✚ The most common target-machine architectures are RISC, CISC, and stack based.

- A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture.

- A CISC machine typically has few registers, two-address instructions, and variety of addressing modes, variable-length instructions.
- 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.

✚ Producing the target program as

- Absolute machine code (executable code)
- Relocatable machine code (Object files for linker and loader)
- Assembly language (assembler)
- Byte code forms for interpreters (e.g. JVM)

✚ In this chapter

- Use very simple RISC-like computer as the target machine.
- Add some CISC-like addressing modes
- Use assembly code as the target language.

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 the 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, use code templates to translate each IR statement into a sequence of machine instruction.

- Produces poor code, needs further optimization.

✚ If the IR reflects some of the low-level details of the underlying machine, then it can use this information to generate more efficient code sequence. 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.
- Instruction speeds are another important factor.

- If we do not care about the efficiency of the target program, instruction selection is straightforward.

```

x = y + z ⇒ LD R0, y
            ADD R0, R0, z
            ST x, R0

a = b + c ⇒ LD R0, b
d = a + e ⇒ ADD R0, R0, c
            ST a, R0 Redundant
            LD R0, a
            ADD R0, R0, e
            ST d, R0
  
```

✚ Example: consider the following statement: $x := x + 1$

- Use **ADD** instruction (straight forward)
 - ✓ costly
- Use **INC** instruction
 - ✓ Less costly

✚ Straight forward translation may not always be the best one, which leads to unacceptably inefficient target code.

✚ Suppose we translate three-address code:

$x := y + z$	➔	LD R0 , y ADD R0 , z ST x , R0	
$x := x + 1$	➔	LD R0 , x ADD R0 , #1 ST x , R0	cost = 6
Better	➔	ADD x , #1	cost = 3
Best	➔	INC x	cost = 2

Register Allocation

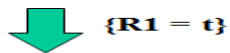
✚ Efficient and careful management of registers results in a faster program. A key problem in code generation is deciding what values to hold in what registers.

- Use of registers imposes two problems:
 - **Register allocation:** select the variables that will reside in registers.
 - **Register assignment:** pick the register that a variable will reside in.

✚ Finding an optimal assignment of registers to variables is mathematically difficult. In addition, the hardware/OS may require some register usage rules to be followed.

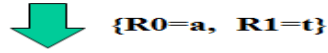
Example:

```
t := a * b
t := t + a
t := t / d
```



```
LD R1, a
MUL R1, b
ADD R1, a
DIV R1, d
ST t, R1
```

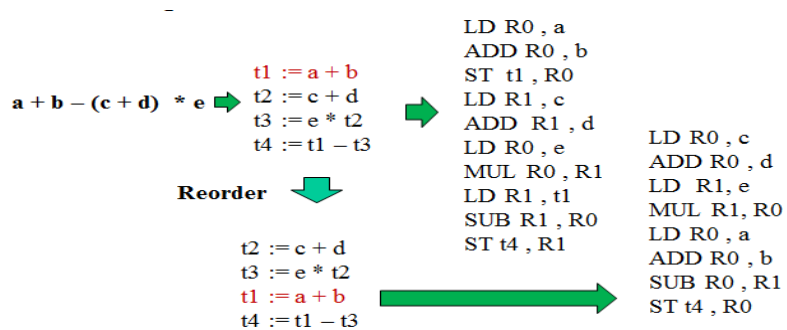
```
t := a * b
t := t + a
t := t / d
```



```
LD R0, a
LD R1, R0
MUL R1, b
ADD R1, R0
DIV R1, d
ST t, R1
```

Choice of Evaluation Order

- The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. However, Selection of the best evaluation order is also mathematically difficult. When instructions are independent, their evaluation order can be changed.



A Simple Target Machine Model

- Implementing code generation requires complete understanding of the target machine architecture and its instruction set.
- Our (hypothetical) machine:
 - Byte-addressable (word = 4 bytes)
 - Has n general purpose registers R_0, R_1, \dots, R_{n-1}
 - All operands are integers
 - Three-address instructions of the form $op \text{ dest}, src_1, src_2$
- Assume the following kinds of instructions are available:
 - Load operations
 - Store operations

- Computation operations
- Unconditional jumps
- Conditional jumps

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 *r*₁, *r*₂** is a register-to-register copy in which the contents of register ***r*₂** are copied into register ***r*₁**.

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

- ✚ Has the form ***OP dst, src₁, src₂***, where ***OP*** is an operator like **ADD** or **SUB**, and ***dst, src₁, src₂*** are locations, not necessarily distinct.
- ✚ The effect of this machine instruction is to apply the operation represented by ***OP*** to the values in locations ***src₁*** and ***src₂***, and place the result of this operation in location ***dst***.
- ✚ For example, **SUB *r*₁, *r*₂, *r*₃** computes $r_1 = r_2 - r_3$ any value formerly stored in ***r*₁** is lost, but if ***r*₁** is ***r*₂** or ***r*₃** the old value is read first. Unary operators that take only one operand do not have a ***src₂***.

Unconditional Jumps

- ✚ The instruction **BR *L*** causes control to branch to the machine instruction with label ***L***. (**BR** stands for branch)

Conditional Jumps

- ✚ Has the form **Bcond *r*, *L***, where: ***r*** is a register, ***L*** is a label, and **cond** is 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.

The Target Machine: Addressing Modes

- ✚ We assume that 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 .
- Indexed address, $a(r)$, where a is a variable and r is a register.

LD R1, a (R2) \longrightarrow R1 = contents (a + contents (R2))

✚ This addressing mode is useful for **accessing arrays**.

- A memory location can be an integer indexed by a register, for example,

LD R1, 100(R2) \longrightarrow R1 = contents (100 + contents (R2))

✚ Useful for following pointers.

- Two indirect addressing modes: $*r$ and $*100(r)$

LD R1, *100 (R2) \longrightarrow R1 = contents (contents (100 + contents (R2)))

✚ Loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2. 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.

$R1 = R1 + 100$

✚ Comments at the end of instructions are preceded by //.

✚ Op-codes (op), for example

LD and **ST** (move content of source to destination)

ADD (add content of source to destination)

SUB (subtract content of source from dest.)

Address modes

Mode	Form	Address	Added Cost
Absolute	M	M	1
Register	R	R	0
Indexed	$a(\mathbf{R})$	$a + \text{contents}(\mathbf{R})$	1
Indirect Register	$*\mathbf{R}$	$\text{contents}(\mathbf{R})$	0
Indirect Indexed	$*a(\mathbf{R})$	$\text{contents}(a + \text{contents}(\mathbf{R}))$	1
Literal	$\#c$	c	1

A Simple Target language (assembly language)

Example:

```
x = y - z => LD R1, y           //R1=y
              LD R2, z           // R2=z
              SUB R1, R1, R2      //R1=R1-R2
              ST x, R1           //x=R1

b = a[i] => LD R1, i             // R1=i
            MUL R1, R1, 8        // R1=R1*8
            LD R2, a(R1)        // R2=content(a+content(R1))
            ST b, R2            // b=R2

a[j] = c => LD R1, c             //R1=c
            LD R2, j             //R2=j
            MUL R2, R2, 8        //R2=R2*8
            ST a(R2), R1        // content(a+content(R2))=R1

x = *p => LD R1, p               //R1=p
          LD R2, 0(R1)          //R2=content(0+content(R1))
          ST x, R2              //x=R2

*p = y => LD R1, p               //R1=p
          LD R2, y               //R2=y
          ST 0(R1), R2          //content(0+content(R1))=R2

if x < y goto L => LD R1, x       //R1=x
                  LD R2, y       //R2=y
                  SUB R1, R1, R2 //R1=R1-R2
                  BLTZ R1, L      //if R1 < 0 jump to L
```

Program and Instruction Costs

✚ **Cost** is associated with compiling and running a program. It measures are:

- The length of compilation time and the size
- Running time and power consumption of the target program

✚ For simplicity, we take:

- **The cost of an instruction = one + the costs associated with the addressing modes of the operands.**

✚ Addressing modes involving:

- registers have zero additional cost,
- a memory location or constant in them have an additional cost of one.

Examples

Instruction	Operation	Cost
LD R1 , R0	Load <i>content(R0)</i> into register R1	1
ST M , R0	Store <i>content(R0)</i> into memory location M	2
LD R0 , M	Load <i>content(M)</i> into register R0	2
ST M , 4(R0)	Store <i>contents(4+contents(R0))</i> into M	3
ST M , *4(R0)	Store <i>contents(contents(4+contents(R0)))</i> into M	3
LD R0, #1	Load 1 into R0	2
ADD *12(R1) , 4(R0)	Add <i>contents(4+contents(R0))</i> to value at location <i>contents(12+contents(R1))</i>	3