Teach yourself **System Software**

Dr. Kalyani A. Patel

Content

Compilers
Aspects of Compilation, various phases of a compiler ,Memory Allocation, Compilation of Expressions, Compilation of Control Structures, Code Optimization.

Ch.1

System Software

System software (**systems software**) is computer **software** designed to operate and control the computer hardware and to provide a platform for running application **software**. **System software** can be separated into two different categories, operating **systems** and utility **software**.

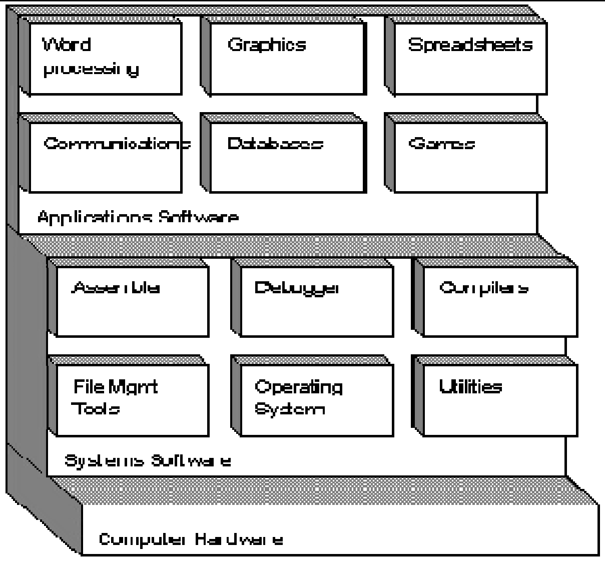


Figure1. Computer System as a Layered Model

System software and Application software are the two main types of computer software. Unlike system software, application software performs a particular function for the user.

System software is a type of computer program that is designed to run a computer's hardware and application programs. If we think of the computer system as a layered model as shown in figure.1, the system software is the interface between the hardware and user applications.

The operating system (OS) is the best-known example of system software. The OS manages all the other programs in a computer.

Each program in the system software is called a system program. System program performs various tasks such as editing a program, compiling and arranging it for execution by linking it with other programs.

Examples of system software and what each does:

- The boot program loads the operating system into the computer's main memory or random access memory (RAM).

- The BIOS (basic input/output system) gets the computer system started after you turn it on and manages the data flow between the operating system and attached devices such as the hard disk, video adapter, keyboard, mouse, and printer.
- According to some definitions, system software also includes system utilities, such as the disk defragmenter and System Restore, and development tools such as debuggers.
- Compiler and is language translator.
- Interpreter is a language processor that bridges the execution gap of a program written in a programming language without generating target program.
- Assembler takes basic computer instructions and converts them into a pattern of bits i.e. machine language that the computer's processor can use to perform its basic operations.
- Macro is a language preprocessor which bridges and execution gap but is not a language translator.
- Language Migrator bridges the specification gap between two programming languages.
- Linker is a system program that puts main program and library programs together for meaning full execution.
- Loader is a system program loads ready to run program into the computer's memory for execution.
- A Device Driver controls a particular type of device that is attached to your computer, such as a keyboard or a mouse. The driver program converts the more general input/output instructions of the operating system to messages that the device type can understand.

1.1 Inception

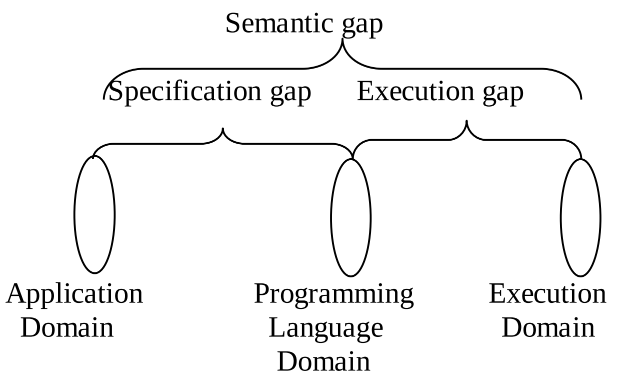
User does not wish to interact with the computer in terms of machine language i.e. in strings of 0s and 1s. The gap between computer and user is known as semantic gap. That is user's territory is application domain and computer's territory is execution domain. This gap between application domain and execution domain is known as semantic gap. The semantics of the domains are very different so the task of bridging the gap is complex and required large amount of time and effort to convert specification of one domain in to the specification of the other domain. The Language Processor bridges this semantic gap.

1.2 Language Processors

A language Processor is software which bridges a specification or execution gap. The specification gap is the semantic gap between two specifications of the same task. And the execution gap is the semantic gap between the semantics of programs that perform the same task but are written in different language.

1.2.1 Techniques to bridge the semantic gap

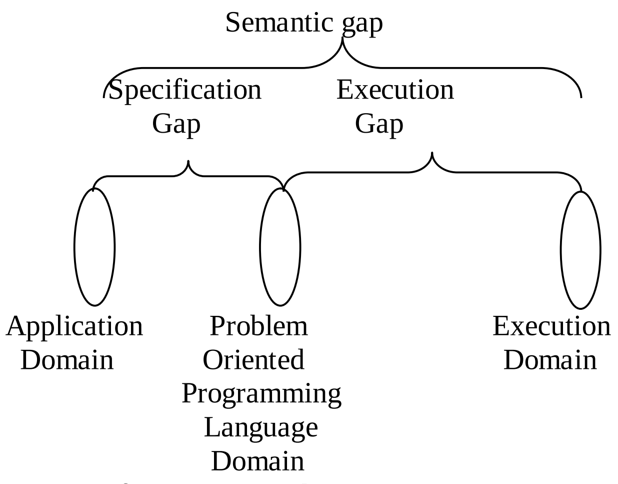
1. Insert Programming Language domain in between application and execution domain. Therefore, the programming language domain split the semantic gap in to two smaller gaps.



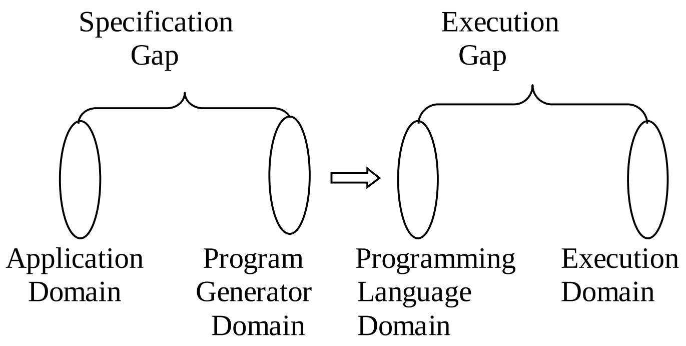
The ease of specification depends on the language in which the specification is written.

- A problem oriented programming language where a user specifies a computation using data and operations that are meaningful in the application area of the computation.
- A procedure oriented programming language provides some standard methods of creating data and performing intended computational operations.

2. Classical solution to reduce the specification gap is by developing a programming language domain close to application domain i.e. generate problem oriented programming language domain.



3. Reduce specification gap by inserting program generation domain in between application and programming language domain. So the harder task of bridging the gap to the programming language domain is performed by the program generator.



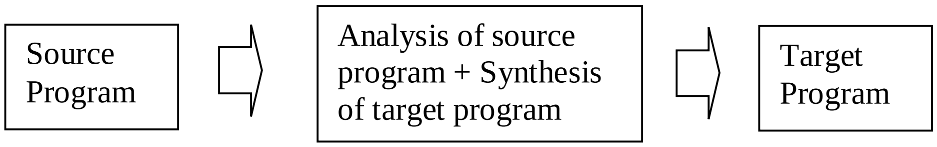
- A program generator – converts the specification written by the user in application domain into a program in a procedure oriented language.
- An assembler / compiler / interpreter bridge the execution gap by implementing a program written in a programming language.
    - Assembler converts an assembly language (machine dependent language) program into machine language program.
    - Compiler translates a source program into the target language, which is either the machine language of a computer or a language that is close to it.
    - An interpreter analysis a source program and performs the computations described in it with the help of computer.

The compiler and interpreter are used depends upon the needs of different kinds of programming environment.

- Cross Translator bridges an execution gap by implementing the machine language of one computer in to the machine language of another computer.
- A de translator bridges the same execution gap as the language translator, but in the opposite direction.
- A language migrator bridges the specification gap between two programming languages. i.e from c to c++.
- A preprocessor is a language processor but is not a language translator.

1.3 Fundamental of language Processing

Language Processing = Analysis of source program + Synthesis of the target program



Analysis of source program consists three components:

1. Lexical Rule : use to form valid lexical unit in Source Program such as operator, Identifier, Constants
2. Syntax Rule : use to verify valid statements in Source Program
3. Semantic Rule: associate meaning with the valid statement of source program.

The Synthesis of target program generates target language statements which would implement the meaning of a source statement. It consists of two main tasks:

1. If the statement is a data declaration statement, decide how the declared data should be represented in the target language and generate corresponding declaration statements.
2. If the statement is imperative statement, i.e some action should be performed, then generate corresponding statements for the action in the target language.

This gives the impression that language processing can be performed on a statement by statement basis.

1.3.1 Multi-pass organization of Language Processing

However statement by statement processing of the source program may not feasible due to

- Program may contain forward reference
- Statement-by-statement processing may require more memory

In case of forward reference the language processor would not know the attribute of the entity when it encounters the forward reference.
Ex: 1.1 Consider the following program:

```
long p_p,c_p;
p_p = (p*100)/c_p;  \\use of p before declaring p i.e forward reference
…..

long p;
```

In the multi-pass organization, the language processor splits the task of language processing into a sequence of stages. The problem caused by forward references is solved by analyzing the program in one stage of the language processor and generating the target program in a later stage.

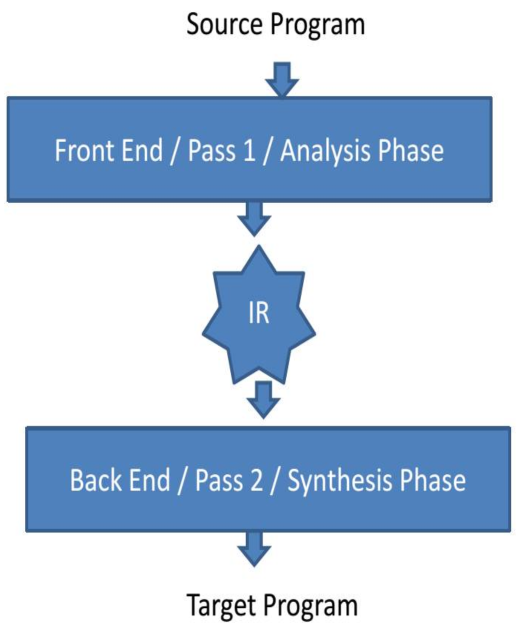
Also, the statement-by-statement language processor requires source code in memory for both the analysis and synthesis phases. It is eliminating in the multi-pass organization because the source code is required in analysis stage only.

Pass -1 Perform analysis of the source program and deduce information in the form of intermediate representation
Pass -2 Perform synthesis of the target program.

Forward Reference: a forward reference of a program entity is a reference to the entity in some statement of the program that occurs before the statement containing the definition or declaration of the entity.
Intermediate Representation (IR): an intermediate representation is a representation of a source program which reflects the effects of some but not all, analysis and synthesis functions performed during language processing.
Properties of IR :

- Ease of use : easy to construct and analysis
- Processing Efficiency : simple representation
- Memory efficiency : compact representation

Benefit of IR:
Memory Requirement Reduced.. Now; No need of co-existence of front end & back end in memory


Source Program
Front End / Pass 1 / Analysis Phase
IR
Back End / Pass 2 / Synthesis Phase
Target Program

Exercise:

1. What is system software? Write down examples of system software and is functions.
2. Write a short note on language processor.
3. Explain fundamentals of language processing.
4. What are the problems of language processor? How to overcome that problem?
5. Define IR. And list out its properties.

Ch.2
Compilers

A compiler is a language processor that translates programs written in any high level language into its equivalent machine language program.

• It bridges the semantic gap between a programming language domain and the execution domain.

• Two aspects of compilation are:
  - o Generate code to implement meaning of a source program in the execution domain.
  - o Provide diagnostics for violation of programming language, semantics in a source program.

• The program instructions are taken as a whole.

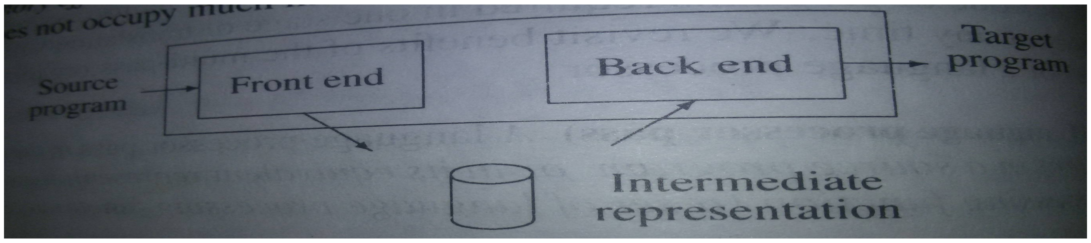
High level language $\Longrightarrow$ Compiler $\Longrightarrow$ Machine language program / a language that is close to it.

For the purpose of compiler construction, a high level programming language is described in terms of a grammar. This grammar specifies the formal description of the syntax or legal statements in the language.

Example: Assignment statement in C is defined as:

< Variable > = < Expression >;

The compiler has to match statement written by the programmer to the structure defined by the grammars and generates appropriate object code for each statement. The compilation process is so complex that it is not reasonable to implement it in one single step. The two-pass language processor is used to implement compilation process. As shown in figure.



The first pass performs analysis of the source program and reflects its result in the intermediate representation. The second pass reads and analysis the intermediate representation to perform synthesis of the target program.

2.1 A Toy Compiler

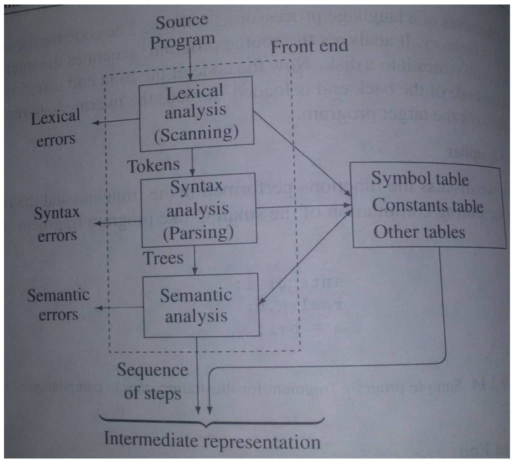
Front End performs lexical, syntax and semantic analysis of the source program.
Functions of Front End:

- Determine validity of source statement from view point of analysis.

- Determine 'content' of source statement.
- Construct a suitable representation of source statement for the use of subsequent analysis function or synthesis phase of a language processor.

Output of Front End:

- IR : Intermediate Representation.
  It has two things with it.
    1. Table of Information.
    2. Intermediate Code (IC) : Description of source statement.



Lexical Analysis of Front End Also called Scanning, Identifies lexical units and then classifies these units further into lexical classes like id's, constants, identifiers, reserved keywords and then enter them into different tables.

Output: Token
Token is the description of lexical analysis. It has two parts 1. Class Code, 2. Number

Class code identifies class to which a lexical unit belongs. Number in class is the entry number in lexical unit in relevant table.

Example:

```
i : integer;
a, b : real;
a = b + i;
```

IR produced by Lexical analysis phase of program:

Table of Information contains information obtained during different analysis of source program, known as 'Symbol Table'. It contains all the identifiers used in source program.

Symbol (Identifier) Table: class code ID

Sr. No.	Symbol	Type	Length	Address
1	i	Integer	-	-
2	a	Real	-	-
3	b	Real	-	-

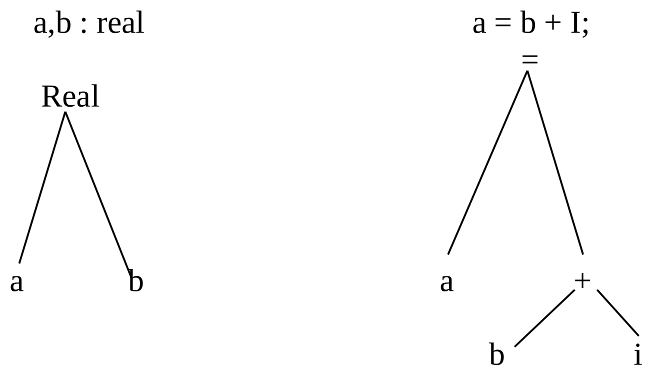
Operator Table: class code OP

Sr.No.	Operator
1	;
2	-
3	+
4	=

String of token for a = b + i; is Id#2 op#4 id#3 op#3 id#1 op#1

Syntax Analysis of Front End Also called Parsing. Syntax analysis process the string of tokens built by lexical analyzer to determine the statement class. i.e. Assignment Statement Class, For Loop Class, Switch Case, etc. It builds IC representing the structure of statements in tree form. Tree can represent hierarchical structure of PL statement appropriately. IC is then passed to semantic analysis to determine meaning of the statement.

Example of IC after syntax analysis



Semantic analysis adds more information to symbol table. So, Symbol table have information about lexical analysis + new names designated by temporary results.

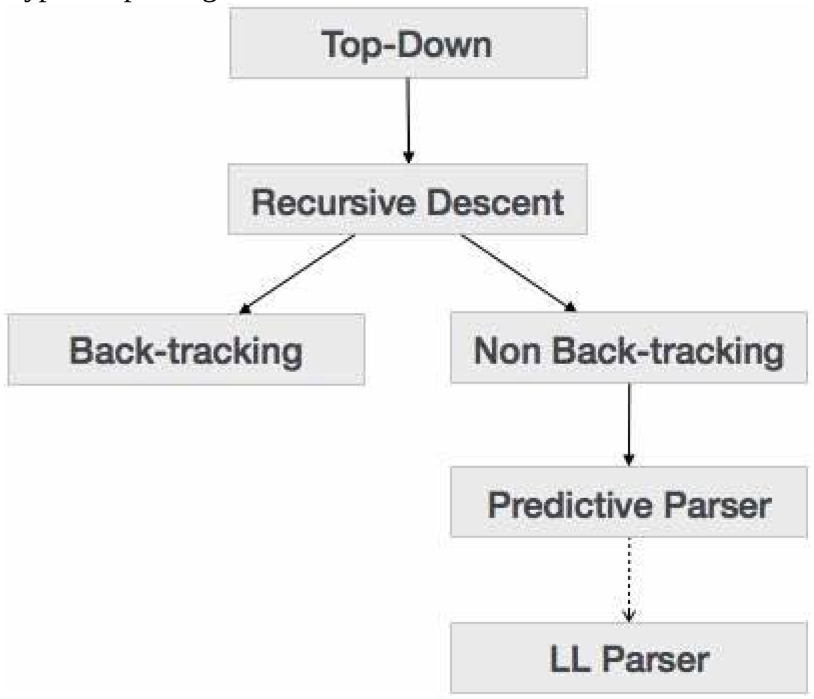
Sr. No.	Symbol	Type	Length	Address
4	i*	Real	-	-
5	temp	Real	-	-

Dfa from book

Grammer from ch1.pdf

Parsing

Types of parsing



Top down parsing with back tracking

From book

Top down parsing without backtracking >>> from book

<u>Left Factoring</u>	<u>Eliminating left-recursion</u>
Consider the grammar E ☛T + E | T T ☛V * T | V V ☛<id>	Consider the grammar E☛E+T|T T☛T*V|V V☛<id>
Common prefix requires ☞ applying left factoring E ☞T + E | T T ☞V * T | V Left factor E E ☞T + E | T ☞ E ☞ TE” E” ☞ + E |ε Left factor T T ☞ V * T | V	Apply left recursion elimination E ☞E+T|T T☞T*V|V Eliminate left recursion of E E☞TE’ E’☞ ε | + TE’ Eliminating left recursion of T T☞VT’ T’☞ ε | + VT’

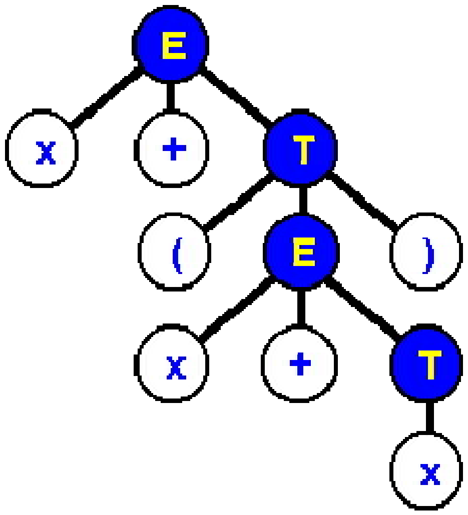
☞ T☞ VT” T”☞ * T | ε	

RD parser

- One of the most straightforward forms of parsing is recursive descent parsing.
- A top-down process in which the parser attempts to verify that the syntax of the input stream is correct as it is read from left to right.
- A basic operation necessary for this involves reading characters from the input stream and matching then with terminals from the grammar that describes the syntax of the input.
- Here recursive descent parsers will look ahead one character and advance the input stream reading pointer when proper matches occur.
- This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking.
- But the grammar associated with it (if not left factored) cannot avoid back-tracking.A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.
- This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.
- What a recursive descent parser actually does is to perform a depth-first search of the derivation tree for the string being parsed. for example, consider the simple grammar

E ® x+T
T ® (E)
T ® x

and the derivation tree in figure for the expression x+(x+x)



```
#include <iostream.h>
```

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <stdlib.h>


typedef struct treenode
{
    char info;
    treenode *left;
    treenode *right;
}node;


    node * proc_e(char input[],int &ssm);
    node * proc_t(char input[],int &ssm);
    node * proc_v(char input[],int &ssm);
    void traversal(node *temp);

void main()
{
    char input[20];
    int ssm=0;
    node *root;
    clrscr();

    cout<<endl<<"ENTER THE STRING TO BE PARSED:";
    cin>>input;

    root = proc_e(input,ssm);
    cout<<endl<<"CONTENTS OF THE PARSE TREE:";
    traversal(root);


    getch();
}


void traversal(node *temp)
{


   if(temp != NULL)
   {
    traversal(temp->left);
    cout<<temp->info;
    traversal(temp->right);
    }
}


//PROCEDURE FOR NT E:

node * proc_e(char input[],int &ssm)
{
    node *a,*b;

    a=proc_t(input,ssm);

    while(input[ssm]=='+')
    {
        ssm++;
        b=proc_t(input,ssm);

        node *temp;
        temp=new node; //(node*)malloc(sizeof(node));
        temp->info='+';
        temp->left=a;
        temp->right=b;

        a=temp;

    }

    return a;
}




//PROCEDURE FOR NT T:

node * proc_t(char input[],int &ssm)
{
    node *a,*b;

    a=proc_v(input,ssm);
    ssm = ssm + 1;

    while(input[ssm] == '*')
    {
        ssm++;

        b=proc_v(input,ssm);
        node *temp;

        temp=new node;//(node *)malloc(sizeof(node));
        temp->info='*';
        temp->left=a;
        temp->right=b;

        a=temp;
        ssm = ssm +1;
    }

    return a;


}

//PROCEDURE FOR NT V:

node * proc_v(char input[],int &ssm)
{
    node *a;

    if(input[ssm]=='I')
    {
        node *temp;
        temp=new node; //(node*)malloc(sizeof(node));
        temp->info='I';
        temp->left=NULL;
        temp->right=NULL;
        return temp;
    }
    else
    {
        cout<<endl<<"ERROR. INVALID SYMBOL "<<input[ssm];
        getch();
        exit(0);
    }
}
```

LL1 parser

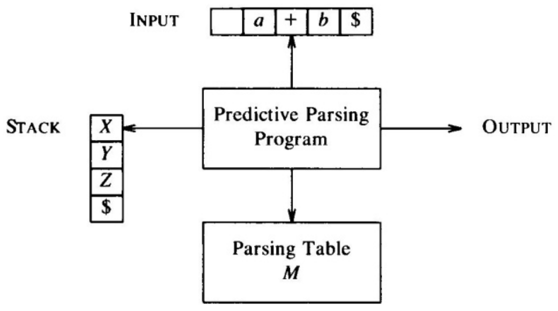


Fig. 4.13. Model of a nonrecursive predictive parser.

- Using the parsing table, the predictive parsing program works like this:
  - A stack of grammar symbols ($ on the bottom)
  - A string of input tokens ($ at the end)
  - A parsing table, M[NT, T] of productions
  - Algorithm:
  - put '$ Start' on the stack ($ is the end of input string).

```
1) if top == input == $ then accept
2) if top == input then
       pop top of the stack; advance to next input symbol; goto 1;
3) If top is nonterminal
       if M[top, input] is a production then replace top with the production; goto 1
       else error
4) else error
```

(1) E->TE'
(2) E'->+TE'
(3) E'->e
(4) T->FT'
(5) T'->*FT'
(6) T'->e
(7) F->(E)
(8) F->id

	id	+	*	(	)	$
E	(1)			(1)		
E'		(2)			(3)	(3)
T	(4)			(4)		
T'		(6)	(5)		(6)	(6)
F	(8)			(7)		

Stack	input	production
$E	id+id*id$	
$E'T	id+id*id$	E->TE'
$E'T'F	id+id*id$	T->FT'
$E'T'id	id+id*id$	F->id
$E'T'	+id*id$	
…...		

This produces leftmost derivation:
E=>TE'=>FT'E'=>idT'E'=>….=>id+id*id

Bottom up parser

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. It uses two unique steps for bottom-up parsing. These steps are known as shift step and reduce-step. The shift step refers to the advancement of the input pointer to the next input symbol, which is called the shifted symbol. When the parser finds a complete grammar rule *RHS* and replaces it to *LHS* , it is known as reduce - step.

1. Precedence Relations

Bottom-up parsers for a large class of context-free grammars can be easily developed using *operator grammars.*

Operator grammars have the property that no production right side is empty or has two adjacent nonterminals. This property enables the implementation of efficient *operator-precedence parsers*. These parser rely on the following three precedence relations:

Relation	Meaning
a <· *b*	*a* yields precedence to *b*
a =· *b*	*a* has the same precedence as *b*
a ·> *b*	*a* takes precedence over *b*

These operator precedence relations allow to delimit the handles in the right sentential forms: <· marks the left end, =· appears in the interior of the handle, and ·> marks the right end.

Let assume that between the symbols a_i and a_{i+1} there is exactly one precedence relation. Suppose that $ is the end of the string. Then for all terminals we can write: $ <· *b* and *b* ·> $. If we remove all nonterminals and place the correct precedence relation: <·, =·, ·> between the remaining terminals, there remain strings that can be analyzed by easily developed parser.

For example, the following operator precedence relations can be introduced for simple expressions:

	id	+	*	$
id		·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	·>

Example: The input string:

$$\mathbf{id}_1 + \mathbf{id}_2 * \mathbf{id}_3$$

after inserting precedence relations becomes

$$\$ <\cdot \mathbf{id}_1 \cdot> + <\cdot \mathbf{id}_2 \cdot> * <\cdot \mathbf{id}_3 \cdot> \$$$

Having precedence relations allows to identify handles as follows:

- scan the string from left until seeing ·>

- scan backwards the string from right to left until seeing <·

- everything between the two relations <· and ·> forms the handle

LPDT

Ch1.pdf

Assembler

Assemblers: • Programmers found it difficult to write or red programs in machine language. In a quest for a convenient language, they began to use a mnemonic (symbol) for each machine instructions which would subsequently be translated into machine language. • Such a mnemonic language is called Assembly language. • Programs known as Assemblers are written to automate the translation of assembly language into machine language.
Assembly language program
Assembler
Machine language program • Fundamental functions: 1. Translating mnemonic operation codes to their machine language equivalents. 2. Assigning machine addresses to symbolic tables used by the programmers.