MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution

PG DEPARTMENT OF COMPUTER SCIENCE

SUBJECT NAME: SYSTEM SOFTWARE

SUBJECT CODE: PSD1C

SEMESTER: I

PREPARED BY: PROF.W.SHARMILA

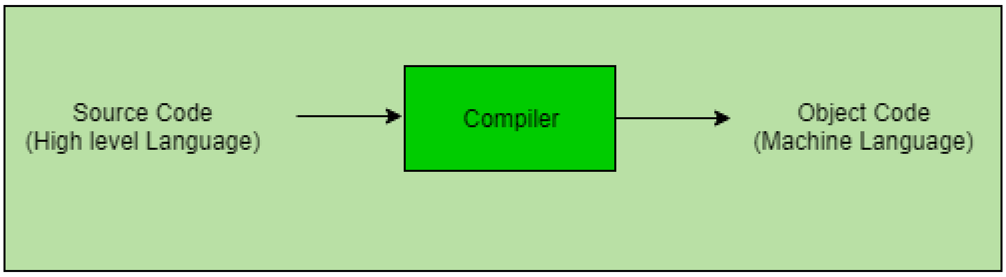
UNIT – I

Language Processors –
Assembly language is machine dependent yet mnemonics that are being used to represent instructions in it are not directly understandable by machine and high Level language is machine independent. A computer understands instructions in machine code, i.e. in the form of 0s and 1s. It is a tedious task to write a computer program directly in machine code. The programs are written mostly in high level languages like Java, C++, Python etc. and are called **source code**. These source codes cannot be executed directly by the computer and must be converted into machine language to be executed. Hence, a special translator system software is used to translate the program written in high-level language into machine code is called **Language Processor** and the program after translated into machine code (object program / object code).

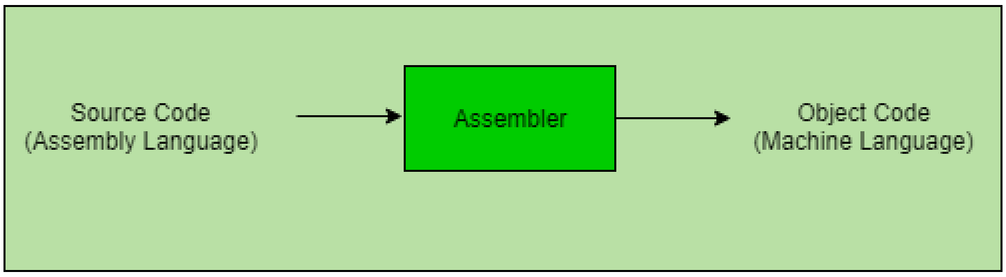
The language processors can be any of the following three types:

1. **Compiler –**
   The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.
   Example: C, C++, C#, Java

In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again.>



2. **Assembler** –
   The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.



3. **Interpreter** –
The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves on to the next line for execution only after removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code. **Example:** Perl, Python and Matlab.

Difference between Compiler and Interpreter –

Compiler	Interpreter
A compiler is a program which coverts the entire source code of a programming language into executable machine code for a CPU.	interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met
Generates intermediate object code.	No intermediate object code is generated.
Examples: C, C++, Java	Examples: Python, Perl

Language processing activities arise to bridge the ideas of software designer with actual execution on the computer system. Due to the differences between the manners in which a software designer describes the ideas concerning the behavior of software and the manner in which these ideas are implemented in a computer system. The designer expresses the ideas in terms related to the *application domain of* the software. To implement these ideas, their description has to be interpreted in terms related to the *execution domain* of the computer system. We use the term *semantics* to represent the rules of meaning of a domain, and the term *semantic gap* to represent the difference between the semantics of two domains. The fundamental language processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap.

· **Program Generation Activities**

· **Program Execution Activities**

A program generation activity aims at automatic generation of a program. The source language is a specification language of an application domain and the target language is typically a procedure oriented PL. A program execution activity, organizes this execution of a program written in a PL on a computer system. Its source language could be a procedure-oriented language or a problem oriented language.

o **<u>Program Generation</u>**

The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL. We call this the program generator domain. The specification gap is now the gap between the application domain and the program generator domain. This gap is smaller than the gap between the application domain and the target PL domain.

Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.

Fig. 1.3: Program generator domain

The harder task of bridging the gap to the PL domain is performed by the generator. This would be performed while implementing the generator. To test an application generated by using the generator, it is necessary to only verify the correctness of the specification input to the program generator. This is a much simpler task than verifying correctness of the generated program. This task can be further simplified by providing a good diagnostic (i.e. error indication) capability in the program generator, which would detect inconsistencies in the specification.

It is more economical to develop a program generator than to develop a problem-oriented language. This is because a problem oriented language suffers a very large execution gap between the PL domain and the execution domain, whereas the program generator has a smaller semantic gap to the target PL domain, j which is the domain of a standard procedure oriented language. The execution gap between the target PL domain and the execution domain is bridged by the compiler or interpreter for the PL.

o **Program Execution**

Two popular models for program execution are

· Translation

· Interpretation

Program Translation

Program translation model bridges the execution gap by translating a program written in a PL, called the source program (SP), into an equivalent program in the machine or assembly language of the computer system, called the target program (TP).

A **specification language** is a formal language in computer science used during systems analysis, requirements analysis, and systems design to describe a system at a much higher level than a programming language, which is used to produce the executable code for a system.

Specification languages are generally not directly executed. They are meant to describe the *what*, not the *how*. Indeed, it is considered as an error if a requirement specification is cluttered with unnecessary implementation detail.

A common fundamental assumption of many specification approaches is that programs are modelled as algebraic or model-theoretic structures that include a collection of sets of data values together with functions over those sets. This level of abstraction coincides with the view that the correctness of the input/output behaviour of a program takes precedence over all its other properties.

In the *property-oriented* approach to specification (taken e.g. by CASL), specifications of programs consist mainly of logical axioms, usually in a logical system in which equality has a prominent role, describing the properties that the functions are required to satisfy—often just by their interrelationship.

This is in contrast to so-called model-oriented specification in frameworks like VDM and Z, which consist of a simple realization of the required behaviour.

Specifications must be subject to a process of *refinement* (the filling-in of implementation detail) before they can actually be implemented. The result of such a refinement process is an executable algorithm, which is either formulated in a programming language, or in an executable subset of the specification language at hand. For example, Hartmann pipelines, when properly applied, may be considered a dataflow specification which *is* directly executable. Another example is the actor model which has no specific application content and must be *specialized* to be executable.

Software Development Tools

- 1 Text Editors. A text editor is a program that allows us to create or edit programs and text files. ...
- 2 Assemblers and Compilers. ...
- 3 Simulators. ...
- 4 High-Level Language Simulators. ...
- 5 Simulators With Hardware Simulation. ...
- 6 Integrated **Development** Environment (IDE)
- Software tool classification

Tool type	**Examples**
Language-processing **tools**	Compilers, interpreters
Program analysis **tools**	Cross reference generators, static analysers, dynamic analysers
Testing **tools**	Test data generators, file comparators
Debugging **tools**	Interactive debugging systems

Data Structures for Language Processing
Data Structures used in Language Processing are classified as

- Nature of Data Structure - Linear or Non Linear
- Purpose of a Data Structure - Search and allocated
- Lifetime of Data Structure - Used during Language Processing or during target program

Linear Data Structure:

Data structures are categorised into two classes : linear and non-linear

1) Linear Data Structures

In linear data structure, member elements form a sequence. Such linear structures can be represented in memory by using one of the two basic strategies

1. By having the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called **arrays.**
2. By having relationship between the elements represented by pointers. These structures are called **linked lists.**

2)Non Linear Data Structures

There are various non-linear structures, such as, trees and graphs and various operations can be performed on these data structures such as:

- **Traversal** - One of the most important operations which involves processing each element in the list.
- **Searching** - Searching or finding any element with a given value or the record with a given key.
- **Insertion** - Adding a new element to the list
- **Deletion** - Removing an element from the list
- **Sorting** - Arranging the elements in some order
- **Merging** - Combining two lists into a single list.

Purpose of Data structure:

1) Search Data Structure: Binary Search

```
intbinary_search(int A[],int key,intimin,intimax)
{
// test if array is empty
if(imax<imin):
// set is empty, so return value showing not found
return KEY_NOT_FOUND;
else
{
// calculate midpoint to cut set in half
intimid= midpoint(imin,imax);
// three-way comparison
if(A[imid]> key)
// key is in lower subset
returnbinary_search(A, key,imin, imid-1);
elseif(A[imid]< key)
// key is in upper subset
returnbinary_search(A, key, imid+1,imax);
else
// key has been found
returnimid;
}
}
```

2) HASH TABLE ORGANISATION

8.3.1 Direct Address Tables

If we have a collection of **n** elements whose keys are unique integers in (1,**m**), where **m** >= **n**,

then we can store the items in a *direct address* table, **T[m]**, where $\mathbf{T_i}$ is either empty or contains one of the elements of our collection.Searching a direct address table is clearly an **O(1)** operation: for a key, **k**, we access $\mathbf{T_k}$,

- if it contains an element, return it,
- if it doesn't then return a NULL.

There are two constraints here:

1. the keys must be unique, and
2. the range of the key must be severely bounded.

If the keys are not unique, then we can simply construct a set of **m** lists and store the heads of these lists in the direct address table. The time to find an element matching an input key will still be **O(1)**.However, if each element of the collection has some other distinguishing feature (other than its key), and if the maximum number of duplicates is $\mathbf{n_{dup}^{max}}$, then searching for a specific element is $\mathbf{O(n_{dup}^{max})}$. If duplicates are the exception rather than the rule, then $\mathbf{n_{dup}^{max}}$ is much smaller than **n** and a direct address table will provide good performance. But if $\mathbf{n_{dup}^{max}}$ approaches **n**, then the time to find a specific element is **O(n)** and a tree structure will be more efficient.

The range of the key determines the size of the direct address table and may be too large to be practical. For instance it's not likely that you'll be able to use a direct address table to store elements which have arbitrary 32-bit integers as their keys for a few years yet!

Direct addressing is easily generalised to the case where there is a function,

h(k) => (1,**m**)

which maps each value of the key, **k**, to the range (1,**m**). In this case, we place the element in **T[h(k)]** rather than **T[k]** and we can search in **O(1)** time as before.

8.3.2 Mapping functions

The direct address approach requires that the function, **h(k)**, is a one-to-one mapping from each **k** to integers in (1,**m**). Such a function is known as a **perfect hashing function**: it maps each key to a distinct integer within some manageable range and enables us to trivially build an **O(1)** search time table.

Unfortunately, finding a perfect hashing function is not always possible. Let's say that we can find a **hash function**, **h(k)**, which maps *most* of the keys onto unique integers, but maps a small number of keys on to the same integer. If the number of **collisions** (cases where multiple keys map onto the same integer), is sufficiently small, then *hash tables* work quite well and give **O(1)** search times.

Handling the collisions

In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table. It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key. But there may be more than one element which should be stored in a single slot of the table. Various techniques are used to manage this problem:

1. chaining,
2. overflow areas,
3. re-hashing,
4. using neighbouring slots (linear probing),
5. quadratic probing,
6. random probing, ...

Chaining

One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require *a priori* knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Re-hashing

Re-hashing schemes use a second hashing operation when there is a collision. If there is a further collision, we *re-hash* until an empty "slot" in the table is found.The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located.

Linear probing

One of the simplest re-hashing functions is +1 (or -1), *ie* on a collision, look in the neighbouring slot in the table. It calculates the new address extremely quickly and may be extremely efficient on a modern RISC processor due to efficient cache utilisation (*cf.* the discussion of linked list efficiency).The animation gives you a practical demonstration of the effect of linear probing: it also implements a quadratic re-hash function so that you can compare the difference.

h(j)=h(k), so the next hash function, **h1** is used. A second collision occurs, so **h2** is used.

Clustering

Linear probing is subject to a **clustering** phenomenon. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashed can become large.

Quadratic Probing

Better behaviour is usually obtained with **quadratic probing**, where the secondary hash function depends on the re-hash index:

address = h(key) + c $\boldsymbol{i^2}$

on the $\boldsymbol{i^{th}}$ re-hash. (A more complex function of ***i*** may also be used.) Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows **secondary clustering**. However, secondary clustering is not nearly as severe as the clustering shown by linear probes.

Re-hashing schemes use the originally allocated table space and thus avoid linked list overhead, but require advance knowledge of the number of items to be stored.

However, the collision elements are stored in slots to which other key values map directly, thus the potential for multiple collisions increases as the table becomes full.

Overflow area

Another scheme will divide the pre-allocated table into two sections: the *primary area* to which keys are mapped and an area for collisions, normally termed the *overflow area*.

> When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system. This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access. As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.

Of course, it is possible to design systems with multiple overflow tables, or with a mechanism for handling overflow out of the overflow area, which provide flexibility without losing the advantages of the overflow scheme.

Summary: Hash Table Organization

Organization	**Advantages**	**Disadvantages**
Chaining	• Unlimited number of elements • Unlimited number of collisions	• Overhead of multiple linked lists
Re-hashing	• Fast re-hashing • Fast access through use of main table space	• Maximum number of elements must be known • Multiple collisions may become probable
Overflow area	• Fast access • Collisions don't use primary table space	• Two parameters which govern performance need to be estimated •

hash table

> Tables which can be searched for an item in **O(1)** time using a hash function to form an address from the key.

hash function

> Function which, when applied to the key, produces a integer which can be used as an address in a hash table.

collision

When a hash function maps two different keys to the same table address, a collision is said to occur.

linear probing

A simple re-hashing scheme in which the next slot in the table is checked on a collision.

quadratic probing

A re-hashing scheme in which a higher (usually 2nd) order function of the hash index is used to calculate the address.

clustering.

Tendency for clusters of adjacent slots to be filled when linear probing is used.

secondary clustering.

Collision sequences generated by addresses calculated with quadratic probing.

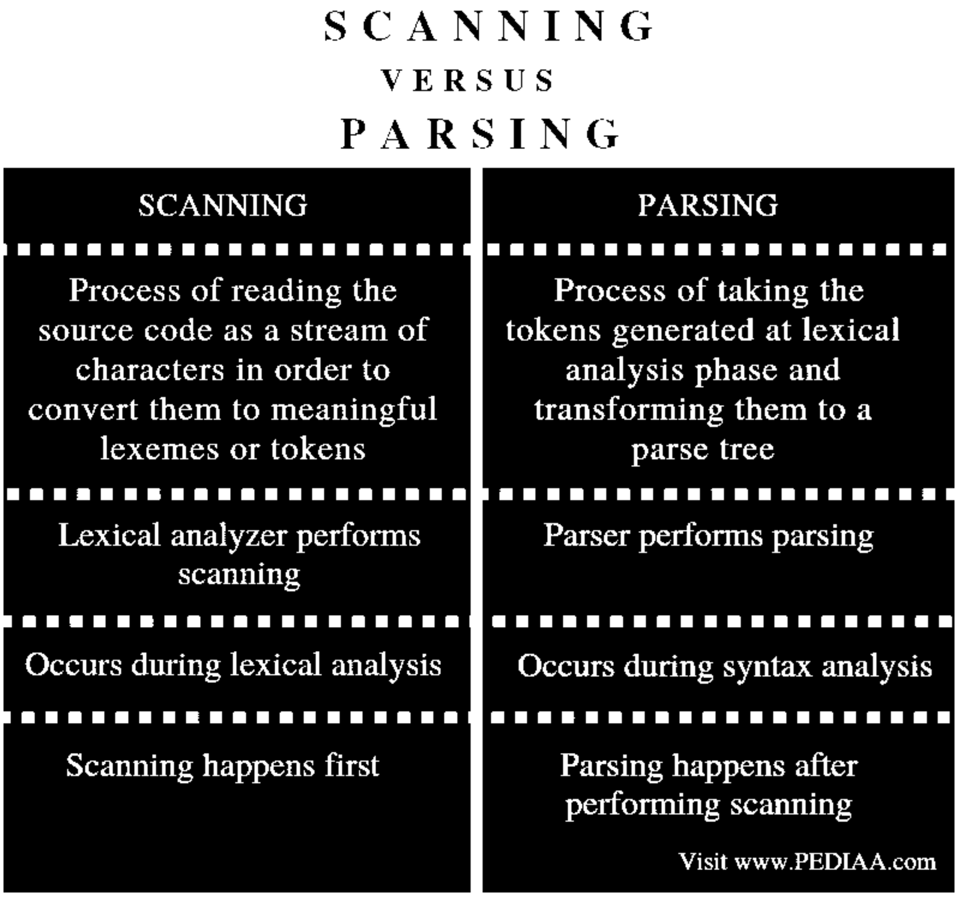
perfect hash function

Function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range.

Scanners and Parsers

The **main difference** between scanning and parsing is that **scanning is the process of reading the source code one character at a time in a methodical manner to convert them into tokens while parsing is the process of taking the tokens and generating a parse tree as the output.**

Generally, a compiler is a software program that is capable of converting the source code into machine code so that the computer can execute that machine code. The compiler goes through multiple phases to compile a program. Scanning and parsing are two activities that occur during this compilation process. Overall, scanning occurs at the lexical analysis phase, whereas parsing occurs at the syntax analysis phase. Furthermore, the lexical analyzer performs scanning while the parser performs parsing.



SCANNING VERSUS PARSING

SCANNING	PARSING
Process of reading the source code as a stream of characters in order to convert them to meaningful lexemes or tokens	Process of taking the tokens generated at lexical analysis phase and transforming them to a parse tree
Lexical analyzer performs scanning	Parser performs parsing
Occurs during lexical analysis	Occurs during syntax analysis
Scanning happens first	Parsing happens after performing scanning

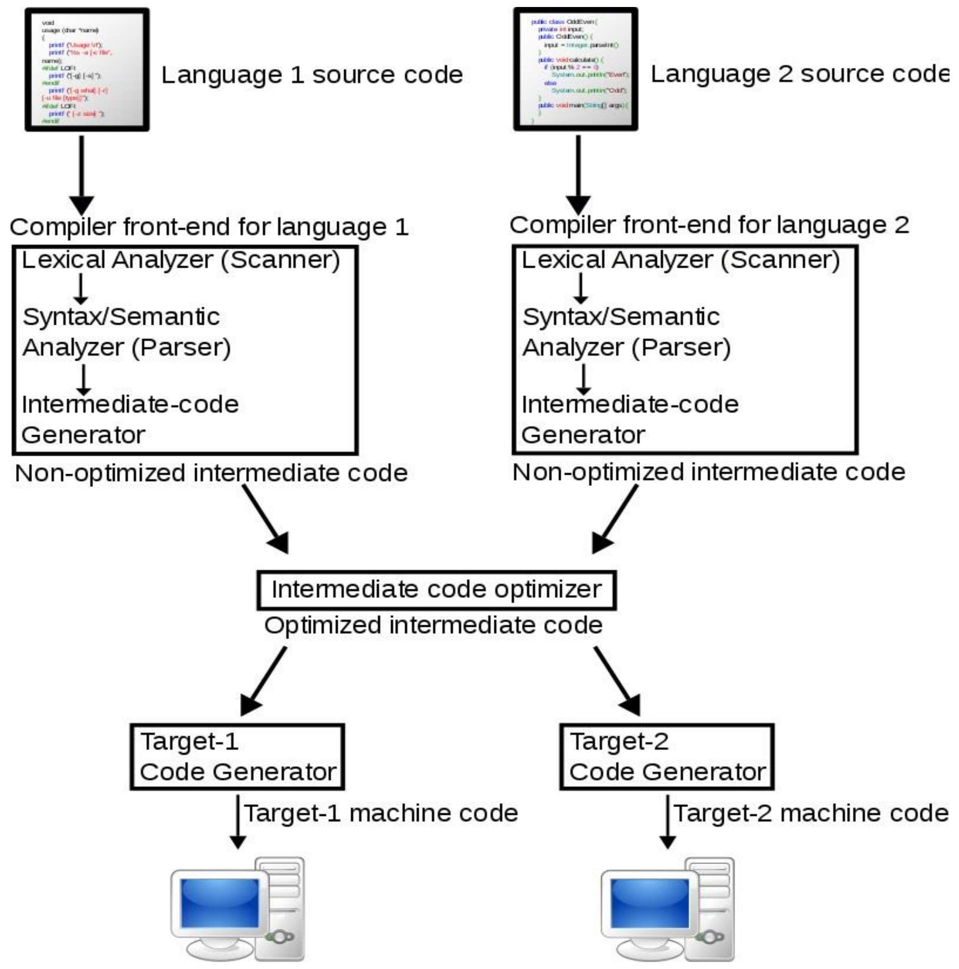
Visit www.PEDIAA.com

What is Scanning

The first phase of compilation is lexical analysis. The lexical analyzer performs this task. It takes the source code as the input. Lexical analyzer reads the source program a character at a time and then converts it into meaningful tokens. The process of reading the source code methodically is called scanning. In this process, the lexical analyzer considers specific information of the source code.

What is Parsing

The tokens generated from lexical analysis goes to the next phase, which is syntax analysis. The parser performs this task. It takes the tokens as input and generates a parse tree as output. Thus, this process is called parsing. Furthermore, the parser checks whether the expression made by the tokens is syntactically correct or not.


Language 1 source code
Language 2 source code
Compiler front-end for language 1
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Compiler front-end for language 2
Lexical Analyzer (Scanner)
Syntax/Semantic Analyzer (Parser)
Intermediate-code Generator
Non-optimized intermediate code
Intermediate code optimizer
Optimized intermediate code
Target-1 Code Generator
Target-1 machine code
Target-2 Code Generator
Target-2 machine code

Moreover, in addition to lexical analysis and syntax analysis, there are other phases such as semantic analysis, intermediate code generation, code optimization etc. After performing all of the above phases, the source code will be converted into the equivalent machine code.

Difference Between Scanning and Parsing

Definition

Scanning is the process of reading the source code as a stream of characters to convert them to meaningful lexemes or tokens. In contrast, parsing is the process of taking the tokens generated at the lexical analysis phase and transforming them into a parse tree. Thus, this is the main difference between scanning and parsing.

Performed by

Further, the lexical analyzer performs scanning while parser performs parsing.

Associated Phase of the Compilation

Besides, scanning occurs during lexical analysis, whereas parsing occurs during syntax analysis. Hence, this is another difference between scanning and parsing.

Occurrence

Moreover, scanning happens first, while parsing happens after performing scanning.

Conclusion

In overall, a compiler is a software program that is responsible for converting the source code into equivalent machine code. It goes through several phases to accomplish this task. Here, the scanning and parsing are two activities that occur during this compilation process. However, the main difference between scanning and parsing is that scanning is the process of reading the source code one character at a time in a methodical manner to convert them to tokens while parsing is the process of taking the tokens and generating a parse tree as the output.

UNIT II

ASSEMBLERS

What is an Assembler?

The first idea a new computer programmer has of how a computer works is learned from a programming language. Invariably, the language is a textual or symbolic method of encoding programs to be executed by the computer. In fact, this language is far removed from what the computer hardware actually "understands". At the hardware level, after all, computers only understand bits and bit patterns. Somewhere between the programmer and the hardware the symbolic programming language must be translated to a pattern of bits. The language processing software which accomplishes this translation is usually centered around either an assembler, a compiler, or an interpreter. The difference between these lies in how much of the meaning of the language is "understood" by the language processor.

An interpreter is a language processor which actually executes programs written in its source language. As such, it can be considered to fully understand that language. At the lowest level of any computer system, there must always be some kind of interpreter, since something must ultimately execute programs. Thus, the hardware may be considered to be the interpreter for the machine language itself. Languages such as BASIC, LISP, and SNOBOL are typically implemented by interpreter programs which are themselves interpreted by this lower level hardware interpreter.

Interpreters running as machine language programs introduce inefficiency because each instruction of the higher level language requires many machine instructions to execute. This motivates the translation of high level language programs to machine language. This translation is accomplished by either assemblers or compilers. If the translation can be accomplished with no attention to the meaning of the source language, then the language is called an assembly or low level language, and the translator is called an assembler. If the meaning must be considered, the translator is called a compiler and the source language is called a high level language. The distinction between high and low level languages is somewhat artificial since there is a continuous spectrum of possible levels of complexity in language design. In fact, many assembly languages contain some high level features, and some high level languages contain low level features.

Since assemblers are the simplest of symbolic programming languages, and since high level languages are complex enough to be the subject of entire texts, only assembly languages will be discussed here. Although this simplifies the discussion of language processing, it does not limit its applicability; most of the problems faced by an implementor of an assembly language are also faced in high level language implementations. Furthermore, most of these problems are present in even the simplest of assembly languages. For this reason, little reference will be made to the comparatively complex assembly languages of real machines in the following sections.

The Assembly Process

It is useful to consider how a person would process a program before trying to think about how it is done by a program. For this purpose, consider the program in Figure 2.1. It is important to note that the assembly process does not require any understanding of the program being assembled. Thus, it is unnecessary to understand the algorithm implemented by the code in Figure 2.1, and little understanding of the particular machine code being used is needed (for those who are curious, the code is written for an R6502 microprocessor, the processor used in the historically important Apple II family of personal computers from the late 1970's).

```
; UNSIGNED INTEGER DIVIDE ROUTINE
;   Takes dividend in A, divisor in Y
;   Returns remainder in A, quotient in Y
START: STA IDENDL     ;Store the low half of the dividend
       STY ISOR       ;Store the divisor
       LDA #0         ;Zero the high half of the dividend (in register A)
       TAX            ;Zero the loop counter (in register X)
  LOOP:  ASL IDENDL    ;Shift the dividend left (low half first)
         ROL           ;                     (high half second)
```

```
        CMP ISOR       ;Compare high dividend with divisor
        BCC NOSUB      ;If IDEND < ISOR don't subtract
         SBC ISOR      ;Subtract ISOR from IDEND
         INC IDENDL     ;Put a one bit in the quotient
  NOSUB: INX           ;Count times through the loop
        CPX #8
        BNE LOOP       ;Repeat loop 8 times
       LDY IDENDL     ;Return quotient in Y
       RTS           ;Return remainder in A

IDENDL:B 0           ;Reserve storage for the low dividend/quotient
ISOR:  B 0           ;Reserve storage for the divisor
```

Figure 2.1. An example assembly language program.

When a person who knows the Roman alphabet looks at text such as that illustrated in Figure 2.1, an important, almost unconscious processing step takes place: The text is seen not as a random pattern on the page, but as a sequence of lines, each composed of a sequence of punctuation marks, numbers, and word-like strings. This processing step is formally called *lexical analysis,* and the words and similar structures recognized at this level are called *lexemes.*

If the person knows the language in which the text is written, a second and still possibly unconscious processing step will occur: Lexical elements of the text will be classified into structures according to their function in the text. In the case of an assembly language, these might be labels, opcodes, operands, and comments; in English, they might be subjects, objects, verbs, and subsidiary phrases. This level of analysis is called *syntactic analysis,* and is performed with respect to the *grammar* or *syntax* of the language in question.

A person trying to hand translate the above example program must know that the R6502 microprocessor has a 16 bit memory address, that memory is addressed in 8 bit (one byte) units, and that instructions have a one byte opcode field followed optionally by additional bytes for the operands. The first step would typically involve looking at each instruction to find out how many bytes of memory it occupies. Table 2.1 lists the instructions used in the above example and gives the necessary information for this step.

Opcode	Bytes	Hex Code
ASL	3	0E aa aa
B	1	cc

BCC	2	90 oo
BNE	2	D0 oo
CMP	3	CD aa aa
CPX #	2	E0 cc
INC	3	EE aa aa
INX	1	E8
LDA #	2	A9 cc
LDY	3	AC aa aa
ROL	1	2A
RTS	1	60
SBC	3	ED aa aa
STA	3	8D aa aa
STY	3	8C aa aa
TAX	1	AA

Notes: aa aa - two byte address, least significant byte first.
oo - one byte relative address.
cc - one byte of constant data.

Table 2.1. Opcodes on the R6502.

To begin the translation of the example program to machine code, we take the data from table 2.1 and attach it to each line of code. Each significant line of an assembly language program includes the symbolic name of one machine instruction, for example, STA. This is called the *opcode* or *operation code* for that line. The programmer, of course, needs to know what the program is supposed to do and what these opcodes are supposed to do, but the translator has no need to know this! Here, we show the numerical equivalent of each opcode code in hexadecimal, or base 16. We could have used any number base; inside the computer, the bytes are stored in binary, and because hexidecimal to binary conversion is trivial, we use that base here. While we're at it, we will strip off all the irrelevant commentary and formatting that was only included only for the human reader, and leave only the textual description of the program.

```
8D START: STA IDENDL
aa
aa
8C      STY ISOR
aa
aa
A9      LDA #0
cc
```

```
AA       TAX
0E LOOP:  ASL IDENDL
aa
aa
2A       ROL
CD       CMP ISOR
aa
aa
90       BCC NOSUB
oo
ED       SBC ISOR
aa
aa
EE       INC IDENDL
aa
aa
E8 NOSUB: INX
E0       CPX #8
cc
D0       BNE LOOP
oo
AC       LDY IDENDL
aa
aa
60       RTS
cc IDENDL:B 0
cc ISOR:  B 0
```

Figure 2.2. Partial translation of the example to machine language

The result of this first step in the translation is shown in Figure 2.2. This certainly does not complete the job! Table 2.1 included constant data, relative offsets and addresses, as indicated by the lower case notatons cc, oo and aaaa, and to finish the translation to machine code, we must substitute numeric values for these!

Constants are the easiest. We simply incorporate the appropriate constants from the source code into the machine code, translating each to hexadecimal. Relative offsets are a bit more difficult! These give the number of bytes ahead (if positive) or behind (if negative) the location immediately after the location that references the offset. Negative offsets are represented using 2's complement notation.

```
8D START: STA IDENDL
aa
aa
8C       STY ISOR
aa
aa
A9       LDA #0
00
AA       TAX
0E LOOP:  ASL IDENDL
aa
aa
2A       ROL
CD       CMP ISOR
aa
aa
90       BCC NOSUB
06
ED       SBC ISOR
aa
aa
EE       INC IDENDL
aa
aa
E8 NOSUB: INX
E0       CPX #8
08
D0       BNE LOOP
EC
AC       LDY IDENDL
aa
aa
60       RTS
00 IDENDL:B 0
00 ISOR:  B 0
```

Figure 2.3. Additional translation of the example to machine language

The result of this next translation step is shown in boldface in Figure 2.3. We cannot complete the translation without determining where the code will be placed in memory. Suppose, for example, that we place this code in memory starting at location 0200_{16}. This allows us to determine which byte goes in what memory location, and it allows us to assign values to the two labels IDENDL and ISOR, and thus, fill out the values of all of the 2-byte address fields to complete the translation.

```
0200: 8D START: STA IDENDL
0201: 21
0202: 02
0203: 8C        STY ISOR
0204: 22
0205: 02
0206: A9        LDA #0
0207: 00
0208: AA        TAX
0209: 0E LOOP:  ASL IDENDL
020A: 21
020B: 02
020C: 2A        ROL
020D: CD        CMP ISOR
020E: 22
020F: 02
0210: 90        BCC NOSUB
0211: 06
0212: ED        SBC ISOR
0213: 22
0214: 02
0215: EE        INC IDENDL
0216: 21
0217: 02
0218: E8 NOSUB: INX
0219: E0        CPX #8
021A: 08
021B: D0        BNE LOOP
021C: EC
021D: AC        LDY IDENDL
021E: 21
021F: 02
0220: 60        RTS
```

```
0221: 00 IDENDL:B 0
0222: 00 ISOR:  B 0
```

Figure 2.4. Complete translation of the example to machine language

Again, in completing the translation to machine code, the changes from Figure 2.3 to Figure 2.4 are shown in boldface. For hand assembly of a small program, we don't need anything additional, but if we were assembling a program that ran on for pages and pages, it would be helpful to read through it once to find the numerical addresses of each label in the program, and then read through it again, substituting those numerical values into the code where they are needed.

symbol	address
START	0200
LOOP	0209
NOSUB	0218
IDENDL	0221
ISOR	0222

Table 2.2. The symbol table for Figure 2.4.

Table 2.2 shows the *symbol table* for this small example, sorted into numerical order. For a really large program, we might rewrite the table into alphabetical order to before using it to finish the assembly.

It is worth noting the role which the meaning of the assembly code played in the assembly process. None! The programmer writing the line STA IDENDL must have understood its meaning, "store the value of the A register in the location labeled IDENDL", and the CPU, when it executes the corresponding binary instruction 8D 21 02 must know that this means "store the value of the A register in the location 0221", but there is no need for the person or computer program that translates assembly code to machine code to understand this!

To the translator performing the assembly process, the line STA IDENDL means "allocate 3 bytes of memory, put 8D in the first byte, and put the 16 bit value of the symbol IDENDL in the remaining 2 bytes." If the symbol IDENDL is mapped to the value 0221 by the symbol table, then the interpretation of the result of the assembler's interpretation of the source code is the same as the programmers interpretation. These relationships may be illustrated in Figure 2.5.

Source Text

```
                  /   \
     programmer's   /     \    assembler's
   view of meaning  /       \  view of meaning
                 /         \
       Abstract Meaning ----- Machine Code


                 hardware's
               view of meaning
```

Figure 2.5. Views of the meaning of a program.

A Simple Assembly Language

In order to simplify this discussion of the translation process, an assembly language less complex than that used in the previous example will be used. The R6502 language used there is complicated by the fact that a single symbolic instruction may assemble in many different ways; for example, the symbolic instruction LDA assembles to either A9, AD, A5, or others depending on the form of the operand field. For example, if the operand field begins with a hash mark (#), the immediate form, A9 is used, while if the operand is an expression with a 16 bit value but is not preceded by a hash mark, the direct addressing form, AD is used. In a simplified assembly language, these differences in the address mode can be indicated by different symbolic names.

Another problem with using the R6502 assembly language is its size; it has 56 different symbolic instructions. None of the basic functions of the assembler depend on the number of different instructions, so a simple assembly language with two instructions will be used as an example for the remainder of this chapter. These instructions are B, which means, initialize one byte (8 bits) of memory, and W, which means initialize one word (16 bits) of memory. These correspond to the .BYTE and .WORD directives in the MACRO-11 assembly language for the PDP-11 (circa 1970), or to variants of the DC directive in the IBM 360 (and 370) assembly language (circa 1965). The syntax of most modern assembly languages can be traced back to one or the other of these older languages, although many minor changes have been introduced in the years since the widespread use of these older languages.

These two simple instructions could be used to assemble code for the R6502 processor by composing however many B and W directives as are needed to make up each actual machine instruction, as is illustrated in Figure 2.6.

```
; --  DEFINE SYMBOLIC INSTRUCTION NAMES  --
STA = #8D           ;STA direct addressing
```

```
STY = #8C           ;STY direct addressing
LDAI= #A9           ;LDA immediate operand
TAX = #AA           ;TAX
ASL = #0E           ;ASL

; --  THE PROGRAM ITSELF  --
START: B STA         ; Store
         W IDENDL    ; ... the low half of the dividend
       B STY         ; Store
         W ISOR      ; ... the divisor
       B LDAI        ; Load register A (the high half of the dividend)
         B 0         ; ... with zero
       B TAX         ; Zero the loop counter (in register X)
LOOP:  B ASL         ; Shift left
         W IDENDL    ; ... the dividend
```

Figure 2.6. Part of Figure 2.1 recoded in the simple assembly language.

Figure 2.6 completes the first 5 instructions of the original example, except that the programmer has had to remember the instruction format and write one line per byte or per 16-bit word in the program, and the programmer had to begin his or her efforts by explicitly defining to the assembler the values to be assembled for each machine instruction. In the Figure, indenting has been used to distinguish between instructions and their operands.

Informally, each line of this simple assembly language is either a definition or a statement. Definitions assign values to symbolic names and do not imply the loading of any values in memory; each of the two statements we have defined loads values in memory in its own way.

Each statements consists of an optional label followed by an opcode and an operand. Labels end with a colon and may begin anywhere on the line. Note that the freedom to indent labels is not common. Many assemblers require that labels begin at the left margin.

The valid opcodes are B and W; these mean, respectively, assemble one byte and assemble one word. The operand field, which is the same as the value field in a definition, may be either an identifier, a symbolic name, a decimal number, or a hexadecimal number; the latter is indicated by the use of the # symbol as a prefix (this should not be confused with the use of the # prefix in the official R6502 assembly language, where it means an immediate constant). If an identifier or symbolic name is used, it must be defined elsewhere in the program, either by its use as a label, or by its use in a definition.

Formal Definitions

The above informal definition is accurate as far as it goes, but its very informality leads to difficulties. If two different programmers used this definition and wrote their own assemblers, it is likely that they would end up supporting slightly different languages. With definitions of larger languages, the differences between independently written processors frequently become insurmountable.

Over the years, a number of formal definition techniques have been developed which help to overcome this problem. Perhaps the oldest of these is *BNF notation*.

The initials BNF stand for either Backus-Naur Form or Backus Normal Form (depending on who is talking). This notation became widely used after Peter Naur used it in the definition of Algol 60; Naur modified a notation used by John Backus (the developer of FORTRAN). Since Backus has claimed that he did not invent the notation himself, but merely used it, and since the notation is not (technically speaking) a normal form, perhaps it is best to forget what the initials BNF stand for.

An important limitation of this notation is that it only defines the syntax of a language, while informal definitions such as the one given above indicate something about the meaning or semantics involved. Thus, a BNF definition can describe how to construct an assembly language program, but it can notdescribe the meaning of the result. The small assembly language used here is defined in Figure 2.7, with added informal comments.

```
<program> ::= <line><end of file> | <line><program>
   -- a program is a sequence of 1 or more lines

<line> ::= <definition> | <statement> | <comment>
   -- a line is either a definition, statement or comment

<definition> ::= <identifier> = <operand><comment>
   -- a definition is an identifier, followed by an
      equals sign, followed by an operand

<statement> ::= <label><instruction> | <instruction>
   -- the label part of a statement is optional

<instruction> ::= <opcode><operand><comment> | <comment>
   -- the opcode, operand part of an instruction is optional
```

<comment> ::= ;<text><line end> | <line end>

-- comments at ends of lines are optional

<label> ::= <identifier> :

-- a label is a symbol followed by a colon

<opcode> ::= B | W

-- the legal opcodes are B and W

<operand> ::= <identifier> | <number>

-- an operand is either an identifier or a number

Figure 2.7. BNF definition of the small assembly language.

Each line in the formal part of the above definition is called a production rule because it defines how to produce an object in the language from simpler objects. For example, a definition is made by concatenating a symbol, an equals sign, an operand, and a comment. Similarly, a comment is made by either a line end or a semicolon followed by any text followed by a line end.

In BNF, the symbols <> | and ::= have special meanings. The ::= symbol is used to indicate that the object on the left is defined by the "expression" to the right. The vertical bar is used to separate alternatives, while the angle brackets are used to enclose "nonterminal" symbols (those which must be further defined elsewhere). All of these special symbols are called *metasymbols* because they are used to "speak about" symbols in the language being defined.

This definition has two faults: It is wordy, and it omits lexical details such as the rules governing spacing and the construction of identifiers and numbers. Using BNF, the latter details can be defined as shown in Figure 2.8:

<identifier> ::= <letter> | <symbol><letter or digit>

-- identifiers start with a letter

<letter> ::= A | B | C | ... | X | Y | Z

<digit> ::= 0 | 1 | 2 | ... | 7 | 8 | 9

<letter or digit> ::= <letter> | <digit>

<number> ::= <decimal> | #<hexadecimal>

<decimal> ::= <digit> | <digit><decimal>
    -- a decimal number is a sequence of digits

<hexadecimal> ::= <hexdigit> | <hexdigit><hexadecimal>

<hexdigit> ::= <digit> | A | B | C | D | E | F

Figure 2.8: Lexical details of the example language.

Note that Figure 2.8 does not mention the spaces between lexemes! It is fairly common to leave this detail out of the formal description of programming languages. Instead, the informal statement is made that spaces may be included before any lexeme or between lexemes but may not be included within them. It is sometimes necessary to include the additional rule that successive identifiers or numbers must be separated by at least one space.

There are a number of ways of formally including the treatment of spaces in the definition of the syntax of a language, but it is more common to do this in a formal description of the lexical structure, as will be discussed later.

The primary problem with the BNF definitions given above is that they are wordy. There are too many nonterminal symbols. The most common solution to this is to introduce new metasymbols which allow many BNF production rules to be combined into a single rule in the new notation. The symbols which are generally introduced are [], {}, and (). Square brackets enclose optional constructs, curly brackets enclose constructs which may be repeated zero or more times, and parentheses group alternatives.

Notations such as this are commonly called extended BNF or EBNF notations; this one derives from a merger of BNF with the form of definition used originally for COBOL, in which vertical groupings of symbols indicated alternatives, and the different kinds of brackets were used as they are here. Figure 2.9 gives the definition of the example assembly language in this notation.

<program> ::= <line> { <line> } <end of file>
    -- a program is a line followed by zero or more lines

<line> ::= (<definition> | <statement>) [;<text>] <line end>
    -- a line is a definition or statement with an optional comment

<definition> ::= <identifier> = <operand>

<statement> ::= [<identifier> :] [(B | W) <operand>]

<operand> ::= <identifier> | <number>

Figure 2.9. An Extended BNF grammar for the example language.

The difficulty with the definition given in Figure 2.9 is that, by omitting nonterminal symbols such as <comment> and <label>, less of the meaning of the grammar has been conveyed by this definition of the syntax. Of course, if meaningless symbols such as <a> and <b> had been substituted for <comment> and <label> in the original BNF grammar, the same difficulty would have arisen. This illustrates that, by carefully naming nonterminal symbols in a grammar, the grammar can be made to informally describe the meaning of a language at the same time that it formally describes the syntax.

A third notation for the formal definition of the syntax of a language is known as *RTN (Recursive Transition Network)* notation. Definitions in this form are also frequently called *syntax diagrams* or *railroad charts,* and are frequently used for the definition of languages descended from Pascal. The syntax diagrams for the example assembly language are given in Figure 2.10.

```
program      ------
   ---------| line |-------(end of file)----
       /    ------    \
       \_______________/


           ------------
         --| definition |--
line    /    ------------    \
   ----                       ------------------------(line end)--
       \    -----------     /  \       ---------   /
         --| statement |---      -(;)-| comment |-
           -----------            ---------


definition  ------------         ---------
   --------| identifier |---(=)---| operand |----------
          ------------         ---------


statement
   ---------------------------------------------------------
```

```
  \   ------------       / \                   /
   -| identifier |--(:)-    \ --(B)--    ---------  /
     ------------            \        --| operand |-
                            -(W)--    ---------
```

Figure 2.10. RTN notation for the example language.

In RTN notation, nonterminal symbols are boxed, while terminal symbols (those which appear in the language) are circled. These syntax diagrams are essentially translations of the Extended BNF grammar given previously. The term "railroad chart" comes from the similarity of these diagrams to the schematic descriptions of railroad networks frequently used in railroad control towers and dispatching centers. As with flowcharts, poorly structured syntax diagrams are possible which are not easily translated to a structured form such as Extended BNF.

RTN notation has an important property: The RTN diagrams for a language are isomorphic to the flowchart of a program which reads input in that language! Such a program is called a *parser*. The same observations can be made about Extended BNF notation. In that case, the relation to be noted is that there are operations for selection between alternatives (a|b is like if ? then a else b), for repetition ({a} is like while ? do a), and for conditional inclusion ([a] is like if ? then a). Additionally, in both Extended BNF and RTN notation, the inclusion of a nonterminal symbol in the definition is equivalent to a procedure or function call in a program (hence the R in RTN).

There is a problem with the relationship between language definitions and programs which process that language. This problem is hinted at by the question marks in the parenthetic remarks in the last paragraph. The problem is that, although the form of the parsing program is specified by the language definition, the conditions to be tested at each branch in the flowchart are not specified. This is the crux of the parsing problem.

Before discussing some solutions to the parsing problem, it is interesting to consider the reverse problem, that of writing a program which generates programs in the language being defined. In that case, each terminal symbol in the language definition maps to a write statement. A simple program generator for random programs would request a new random number to be used as the basis of each branch in the program. For example, if "random" is a function returning a random boolean value each time it is called, the random generator for lines of assembly code would have the form given in Figure 2.11.

```
procedure line;                   void line()
begin                             {
   if random then begin        if (random()) {
      if random                   if (random())
```

```
        then definition                    definition();
        else statement;             else
    end;                                   statement();
    if random then begin          }
      write(';');                 if (random()) {
      text;                                    putchar(';')
    end                                  text();
end {line};                                }
                                         } /* line */
```

Figure 2.11. A random program generator in Pascal and C.

Most of the "computer poetry" which is the subject of occasional jokes is produced using essentially this technique, except that the basic grammar is that of a language such as English, and variables are added to control such things as rhyme and meter. In artificial intelligence work, an RTN grammar with added variables is referred to as an ATN or Augmented Transition Network grammar. The use of ATN grammars is at the center of much work with natural language understanding.

Parsing

A program (or part of a program) which reads text in an input language and classifies its components according to their grammatic derivation is called a *parser*. In this section, we will deal only with parsers and not with the problem of what to do with the output of the parser. A language processing program where the parser directs the translation process is said to be a *syntax directed translator*; later sections will describe these. As has already been mentioned, the flowchart of a parser can be derived from the grammar of a language; there are other forms of parsers, for example, table driven ones, but these will not be discussed here.

The parsers discussed here are sometimes called *top-down parsers* because they begin with the assumption that the input will be a program and they operate by trying to decide which of the ways of constructing a program matches the input. An alternative, *bottom-up parsing,* involves putting pieces of the input together to see what they make, hoping eventually to reduce the entire input to a single object and then making sure that the result is a program. The differences between these two approaches are most apparent in the context of expression analysis, where they will be discussed in more detail. An important property of both techniques, however, is that parsing is accomplished as the input text is read; computer programming languages are designed so that a parser can operate by reading only a few lexemes at a time, without any need to hold the entire text in memory at once.

The basic problem faced in a top-down parser is that of differentiating between the various alternate forms that may be substituted for some nonterminal symbol. The example given in Figure 2.12 demonstrates this problem in the context of the nonterminal symbol <line> from the extended BNF grammar given in Figure 2.9:

```
B = 5
B : B 5
B 5
W 5
```

Figure 2.12. A parsing problem for the nonterminal <line>.

The first line in Figure 2.12 is a <definition> while the others are <statement>s. Clearly, these cannot be distinguished by their first lexeme, but the second lexeme does the job. The first lexemes of the second and third lines are the same, but they serve different purposes; again, the second lexeme distinguishes between these purposes. Only in the last two lines is the first lexeme sufficient to distinguish between the forms. These examples suggest (correctly) that the example assembly language can be parsed by reading one lexeme at a time, from left to right, with the added ability to peek ahead at the next lexeme from time to time when that is needed to distinguish between forms which do not differ in their first lexeme.

This process of 'peeking ahead' at the next lexeme is conventionally called *looking ahead*, or *looking right* in the input. The number of lexemes ahead of the current lexeme which must be examined in order to parse a language is commonly used as a measure of the complexity of the grammar for that language. Thus, a grammar which allows a language to be parsed without looking ahead is the simplest; such grammars are called LL0 grammars (for Left-to-right parsing, Leftmost reduction first, looking right 0 places'). The example assembly language is in the class LL1 because it requires one symbol look-ahead. It is interesting to speculate about how far ahead one must look in order to parse English; is English an LL6 language?

Most grammars for English appear to require infinite look-ahead, but example sentences illustrating the need for more than a few words of look-ahead are very hard for real people to follow even though they may be correct under the commonplace grammars people use to describe natural languages. It may be that the human capacity for look-ahead is limited by the fact that human short-term memory can hold about 'seven plus or minus two' things at any time; if this is the case, it becomes reasonable to speculate that a grammar requiring somewhere between 5 and 9 symbols of lookahead might be adequate to describe English as it is actually used.

For the example assembly language, the main body of the parser is easy to propose. This is simply a loop which processes lines until the end of a file. Prior to the 1970's, of course, most parsers were written in assembly language or even machine language, but today, it is common to write language processors in decent high-level languages. Figure 2.13 shows how this might look in Pascal and C.

```
procedure program;                      void program()
begin                                   {
   repeat                         do {
      line;                             line();
   until eof(input);              } while (!feof(stdin));
end {program};                          }
```

Figure 2.13. The main body of a parser in Pascal and C.

The predicates "eof(input)" or "feof(stdin)" can be formally treated as asking if the current lexeme is a special, invisible, "end of file" lexeme, although it would probably be implemented as a simple test for end of file. Note that the parser given in Figure 2.13 has not been coded to anticipate an empty input file; thus, it may well produce unexpected results for an empty file.

Processing a line is more complex, since there must be some way to examine the current and next lexeme. To allow this, we will use two variables, "lex.this" and "lex.next"; the variable "lex.this" always holds the current lexeme, while the variable "lex.next" always holds the lexeme that comes next after the current one. Thus, examining the contents of "lex.next" corresponds to looking ahead in the input. The procedure "lex.scan" will be used to advance the state of the lexical analyzer.

Formally, the lexical analyzer is an object with two read-only public variables, "lex.this" and "lex.next", and one public procedure, "lex.scan". In a language that doesn't support objects, we can simply make these variables global, naming them "lex_this" and "lex_next", with no loss of utility, because we have no intention of ever introducing multiple instances of the lexical analyzer. In fact, if our programming environment requires that we name the lexical analyzer class and then instantiate it, our environment is forcing us to do something inappropriate by suggesting the possibility of multiple instances of this class.

For now, we will assume that the values of "lex.next" and "lex.this" are strings, although this would rarely be the case in a production parser; instead, in production, these really ought to be values of type "lexeme", where values of type lexeme carry compact encodings of the attributes of the lexeme as they are computed.

Using the extended BNF grammar of the example assembly language as a basis, a procedure to parse one line can be written as shown in Figure 2.14.

```
procedure line;                    void line()
begin                              {
  if lex.next = "="             if (!strcmp(lex.next,"="))
    then definition                        definition();
    else statement;                  else
skipline;                          statement();
end {line};                        skipline();
                                   }
```

Figure 2.14. A parser for lines in Pascal and C.

Note that the inclusion of a comment after the body of the definition or statement has been ignored! Whatever follows the definition or statement up to the end of line has simply been skipped over by the call to "skipline". Detection of errors significantly complicates this code; as is illustrated in Figure 2.15.

```
procedure line {with error detection};
begin
  if is_identifier(lex.this) then
    if lex.next = "="
      then definition
      else statement;
  if (lex.this = ";") or is_eol(lex.this) then begin
        skipline;
  end else begin
    error("comment expected, something else found");
        skipline;
  end;
end {line};
```

Figure 2.15. A parser with error detection.

Here, the predicate "is_identifier" has been used to check that the line begins with a valid identifier, since all legal nonblank lines start with a valid identifier. Similarly, the predicate "iseol" has been used to check to see if the current lexeme is an end-of-line marker. In the remainder of this discussion of parsing, this extra code to handle errors will be ignored, but it should be kept in mind that this code frequently dominates the structure of production-quality parsers because users demand good error detection and reporting.

The procedures for parsing definitions and statements which were called from the above routines can easily be written as shown in Figure 2.16.

```
void definition ()
{
lex_scan(); /* skip over identifier */
lex_scan(); /* skip over equals sign */
    operand;
} /* definition */

void statement ()
{
    if (!strcmp(lex.next,":")) {
lex_scan(); /* skip over identifier */
lex_scan(); /* skip over colon */
    }
    if (!strcmp(lex.this,"B")) {
lex_scan(); /* skip over B */
        operand();
    } else if (!strcmp(lex.this,"W")) {
lex_scan(); /* skip over W */
        operand;
    }
} /* statement */
```

Figure 2.16. Parsers for definitions and statements.

It is interesting to note that these versions of definition and statement would require no additional error checking code if called from the error checking version of line given in Figure 2.12, assuming that the operand procedure performs appropriate checks for malformed operands.

A Syntax Directed Assembler

The parser given in the previous section provides a convenient scaffolding on which to build the rest of an assembler. In order to do this, there must be a place to store the assembled code; here, this will "M", standing for memory, an array of bytes.

It should be noted that most production assemblers do not directly store assembled code in memory, but store it in special files called object files; these will be discussed in detail in

Chapter 7. When assembly is directly into memory, it becomes necessary to violate the "sane usage" constraints on pointers, perhaps by using a small assembly language routine that directly interprets an integer memory address as a pointer. A classic name for this routine would be "poke", after the common name for the built-in procedure in many early microcomputer BASIC implementations that did this. Typically, "poke(b,a)" has the effect of "M[a]:=b".

We also need a mechanism to store the association of symbols with values in the symbol table. Logically, the symboltable is an object, perhaps named "st", with two access routines, "st.define" and "st.lookup"; the former defines (or redefines) a symbol by associating a value with it, while the latter returns the value associated with a symbol. Appropriate implementations for these routines will not be discussed until the next chapter, but it is worth noting that, again, the object-oriented paradigm poses minor problems. We don't really want to create a symbol-table class, with the suggestion that there might be multiple coexisting symbol tables in our assembler; rather, we want a guarantee that there will always be exactly one object, the symbol table, that is the only instance of thisr class. Furthermore, with only one instance, the need to prefix each use of an access routine for that instance with the instance name becomes annoying.

We can now rewrite the procedures "definition" and "statement" as shown in Figure 2.17 for use in a real assembler.

```
procedure definition;
begin
    s := lex.this {save the symbol to be st_defined};
lex_scan      {skip that symbol};
lex_scan      {skip the equals sign};
    v := operand;
st_define(s,v);
end {definition};

procedure statement;
begin
    if lex.next = ":" then begin
        s := lex.this {save symbol used as label};
lex_scan  {skip label};
lex_scan  {skip colon};
st_define(s,location);
    end;
    if lex.this = "B" then begin
lex_scan {skip B};
        M[location] := operand;
        location := location + 1;
    end else if lex.this = "W" then begin
lex_scan {skip W};
              o := operand
        M[location] := first_byte_of(o);
        M[location + 1] := second_byte_of(o);
        location := location + 2;
    end;
end {statement};
```

Figure 2.17. The heart of an assembler.

To paraphrase the actions taken by these procedures, when a definition is found, the identifier is set equal to the associated operand. In a statement, when a label is found, it is set equal to the current location. The opcode B causes the operand to be stored in the current location, after which the current location is incremented by one. The opcode W causes the operand to be stored in the current and next location (taken as a 16 bit word), after which the current location is incremented by two.

The variable called "location" above is an important component of any assembler. It is commonly called the *location counter* in the assembler, by analogy with the program counter maintained by the computer when it runs a program. The assembler uses the location counter to determine where to place assembled instructions in memory during the assembly process, while the computer uses the program counter to determine where to fetch instructions from in memory when it runs a program.

Lexical Analysis

Before the shortcomings of the above basic assembler are examined, We will examine the implementation of the lexical analysis package, with the access procedure "lex.scan" and the variables "lex.this" and "lex.next". The "lex.scan" procedure identifies lexemes (words, tokens, or other logical units) from the lexicon (vocabulary) of a language. Although the syntactic structures (grammars) of computer languages differ greatly, their lexical structures are very similar to each other and to the written forms of natural languages which use the same alphabet. Thus, spaces serve to delimit lexemes, as do punctuation marks, which are themselves lexemes. It is important to note that the process of lexical analysis never depends on the meaning of the language or on syntactic issues such as whether or not some lexeme is allowed in a particular context.

The lexical structure of the example assembly language can be summarized as follows: All lexemes are either symbolic names, numbers, or punctuation marks. B and W are simply symbolic names. A symbolic name is a letter followed by zero or more letters or digits. A number is either a string of digits or a pound sign followed by a string of hexadecimal digits. The allowed punctuation marks are the equals sign, colon, semicolon, line-end and end-of-file. Any number of spaces may be inserted between lexemes without changing the lexical structure of a string, but at least one space must initially separate successive symbolic names or numbers. The extended BNF grammar given in Figure 2.18 describes the lexical level of the example assembly language in more detail than that in Figure 2.8.

```
<program> ::= <lexeme> { <lexeme> }
    -- a program is a string of one or more lexemes

<lexeme> ::= { <blank> } ( <identifier> | <number> | <punctuation> )
    -- any lexeme may be preceded by blanks

<identifier> ::= <letter> { <letter> | <digit> }

<number> ::= # <hexdigit> { <hexdigit> } | <digit> { <digit> }

<punctuation> ::= : | ; | = | <line end> | <end of file>
```

Figure 2.18. Lexical details in EBNF.

This definition of the lexical level does not include the rule that consecutive identifiers or decimal numbers must be separated by spaces; thus, it is ambiguous. This does not cause a problem in lexical analysis, but programmers must be aware that the string "B12" will be interpreted as one identifier, even though the above rules would allow it to be interpreted as starting with the identifiers "B" or "B1" followed by the numbers "12" or "2". The reason this causes no problem in lexical analysis is that, for both parsers and lexical analyzers, a so called greedy approach is commonly used. That is, we assume that the parser or lexical analyzer will construct the largest identifier or number it can by following the rules for <identifier> or <number> before it returns to the level where it looks for the start of the next lexeme.

An alternate way of formalizing the description of the lexical level of a language rests on the use of finite state transition diagrams or simple state transition networks. In such a definition, state changes are caused by the processing of successive input characters, and some state changes also signal the completion of the analysis of some lexeme. The notation used is very similar to RTN notation, and is shown in Figure 2.19.

```
      __________________________________________________
     /                                                 \
start  \                              identifier   /|
 -------->----------(letter)-------->-------------------  |
   /       \ \          /         \            |
   \       /  |        |\         /|           |
    (blank)    |        |  -(letter)-  |           |
              |         \         /            |
              |           -(digit)--              |
              |\                   number      /|
              |  (#)----(hexdigit)---->---------------  |
              |      /          \                |
              |      \_____________/                  |
              |\                   number      /|
              |  ----(digit)---------->---------------  |
              |   /       \                    |
              |   \_________/                      |
              |\                   punctuation  /|
              |\ ---------(:)--------->--------------- /|
              |\ -----(;)---------------------------- /|
               \ -------------(line end)------------- /
                 -(end of file)----------------------
```

Figure 2.19. Finite state description of the lexical level.

None of the rules given up to this point mention anything about a maximum length for identifiers, maximum value for numbers, maximum number of characters in a line, or maximum program size. These are frequently considered to be outside of the realm of formal definition, and may even vary from one implementation of a language to another. Typically, the informal part of the language specification will include minimum values for the line length, number of significant characters in an identifier, and the maximum number of digits allowed in a number.

A typical lexical analyzer will contain, as a private component, a line buffer which holds one line of input (a string variable or an array of characters). With this buffer is associated a variable which points to or indexes the first character in the buffer which has not yet been processed at the lexical level. Because of the need for look-ahead, processing at the lexical level will generally be a few lexemes ahead of processing at the syntactic level. We will use the variable "pos" to serve this purpose.

In addition, we need a more sophisticated way to represent the current lexemes than simple character strings! Instead, we will represent lexemes with a record or structure that contains information about the lexeme. Figure 2.20 illustrates appropriate type definitions:

```
type lextypes = (identifier, number, punctuation);
type lexeme = record
           start: integer { index of start of lexeme on line };
                     stop: integer  { index of end of lexeme on line };
typ: lextypes;
         end;


enumlextypes { identifier, number, punctuation };
struct lexeme {
int start; /* index of start of lexeme on line */
int stop;  /* index of end of lexeme on line */
lextypestyp;  /* index of end of lexeme on line */
}
```

Figure 2.20. Type definitions for lexeme types in Pascal and C.

A programming language such as Ada allows a clear definition of the interface between the lexical analyzer and the rest of the world, as shown in Figure 2.21.

```
package lex is
  type lextype is (identifier, number, punctuation);
  type lexeme is
    record
      start: integer; -- starting position of lexeme on line
      stop: integer;  -- ending position of lexeme on line
typ: lextype;   -- nature of this lexeme
    end record;

  this: lexeme; -- the current lexeme
  next: lexeme; -- the lexeme following the current one

  procedure init;     -- called to start the lexical analyzer
  procedure nextline; -- called to advance to the next line
```

```
            -- after a call to either of the above, this and next will
            -- be the first and second lexeme on the current line

   procedure scan;     -- called to advance to the next lexeme on the line
            -- after a call to next, this and next will advance one lexeme
            -- within the current line
end lex;
```

Figure 2.21: An Ada interface to the Lexical Analyzer

As with C++ and Java, the Ada language allows interface specificiations to be given separately from the implementation of an abstraction. All of the definitions in an Ada package declaration are publically available to the rest of the program, including type definitions, variables and functions. Unlike C++ and Java, however, Ada packages are objects, not classes; Ada does include something called a generic package that corresponds to classes, but the purpose of this discussion is not to teach all of Ada.

It is fair to ask, why didn't we add a string field to the lexeme structure to hold the text of the current lexeme? The answer to this is that we are interested in writing efficient software, and copying strings is something that should be avoided if it is not necessary. Therefore, what we want in the lexeme data structure is not the text of the lexeme, but rather, the numerical value of numeric lexemes, some equally concise indication of what identifer is represented, and in the case of punctuation, a quick and easy way to determine what mark is involved. We will deal with these issues later.

Given an interface specification, we can go on to define the functions and private variables of the lexical analyzer as shown in Figure 2.22:

```
package body lex is
   line: array (0 .. linelen) of char;
pos: integer; -- current position in line

   ...  -- we omit a few details (initialization etc)

   procedure scan is
   begin
      this := next;
      while line(pos) = ' ' loop
pos := pos + 1;
endloop
```

```
next.start := pos; -- mark start of lexeme
    if line(pos) in 'A' .. 'Z' then
next.typ := identifier;
      loop
          pos := pos + 1;
             exit when   (line(pos) not in 'A' .. 'Z')
               and then (line(pos) not in '0' .. '9');
endloop;
elsif line(pos) in '0' .. '9' then
next.typ := number;
      repeat pos := pos + 1;
      loop
          pos := pos + 1;
             exit when line(pos) not in '0' .. '9';
endloop;
elsiflinebuf[pos] = '#' then
next.typ := number;
      loop
          pos := pos + 1;
             exit when   (line(pos) not in '0' .. '9')
               and then (line(pos) not in 'A' .. 'F');
endloop;
    else
      -- we treat everything else as punctuation
next.typ := punctuation;
pos := pos + 1;
endif;
next.stop := pos - 1 {remember where lexeme ends};
  end scan;
end lex;
```

Figure 2.22. A lexical analyzer.

Note that important details have been ignored in this version of "lex.scan" such as initialization, checking for the end of a line, or handling of invalid characters; furthermore, we've provided no way for the user to inspect the current lexeme to determine if it is a particular identifier or a particular punctuation mark!

The version of "lex.scan" given in Figure 2.22 makes it clear that the cost of one lexeme look-ahead is a single assignment statement per lexeme processed, plus an extra variable to store the value of one lexeme. In fact, the assignment statement is not free, since it actually involves copying an entire record that is several words long, but we can afford this.

The fact that the cost of look-ahead is low was not understood in the design of some early programming, where the need for look-ahead was eliminated by having a leading keyword on each line to identify the type of that line. For example, all early versions of BASIC required the keyword "LET" at the start of each assignment statement.

It is common to make the lexical analyzer responsible for skipping comments; thus, semicolon would not be considered a lexeme type in the example assembly language; rather, the end of line lexeme would be considered to include the comments leading up to the end of line. In languages such as Pascal and PL/I, where comments may be interspersed between any lexemes, the lexical analyzer would identify and skip comments as part of the code responsible for skipping spaces between lexemes.

It is also common to integrate the production of a listing with the lexical analyzer. Thus, the routine to print a line is typically called from within the lexical analyzer as a consequence of finishing the analysis of the previous line, and error message formatting is tied to the lexical analyzer so that error messages can be printed under the lexeme to which they apply.

Alternatives

The assembler presented up to this point is incomplete, since it lacks any symbol table mechanism, and even if that were provided, it would not be able to handle identifiers which are defined after their first use. These problems will be solved in the next two chapters, but before solving them, it is useful to look at the alternatives which have been avoided in this presentation of parsing techniques.

A natural objection to the above presentation is that it avoids using powerful high level language features; specifically, it makes little use of string operations which are supposed to greatly simplify text processing. In fact, the extensive use of string operations can lead to trouble, as the following example illustrates:

Consider an assembler which, after reading a line in as a string, searches the line, using a string search operator, for any semicolon and uses substring operations to remove that and all following characters (the comment) from the line. The next step might be to use a search operation for an equals sign in order to distinguish between statements and definitions. For statements, a second search operation could be used to see if there is a colon, and if there is, substring operations could be used to remove the colon from the line and process it. Although there is no doubt that a working assembler could be written this way, this approach is also computationally expensive: Each substring operation is typically implemented by a loop which copies one character at a time, and string searches are typically implemented by sequentially testing successive characters. Even if these are done by hardware, the above approach leads to testing each character on a line many times, requiring many memory cycles where the lexical analyzer given requires only one.

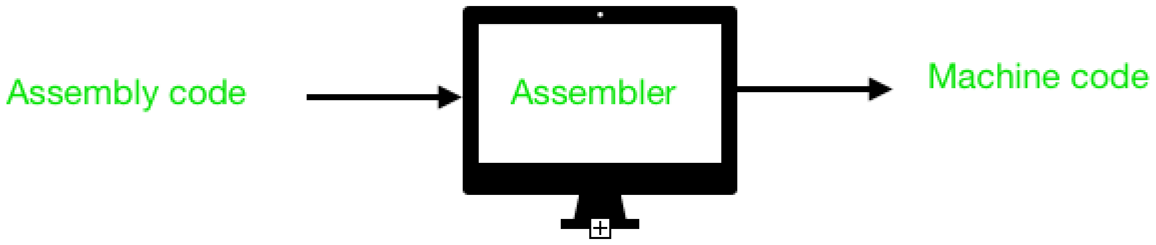
Actually, there is an appropriate way to use string functions in the lexical analysis routine presented above. The key is to use the string function to do exactly the same processing as is explicitly indicated in the code given above; for example:

```
while linebuf[pos] = ' ' do pos := pos + 1;
can be replaced by
pos := (pos-1) + verify(substr(linebuf,pos),' ');
```

assuming the PL/I string functions "verify" and "substr", which return the position of some character in a string and take a substring, respectively. Unfortunately, unless a good optimizing compiler is used, the "substr" operation will involve making an unnecessary copy of part of the line buffer, and it is not much harder to write explicit code for the operation in the first place.

TWO PASS ASSEMBLER

Assembler is a program for converting instructions written in low-level assembly code into relocatable machine code and generating along information for the loader.



It generates instructions by evaluating the mnemonics (symbols) in operation field and find the value of symbol and literals to produce machine code. Now, if assembler do all this work in one scan then it is called single pass assembler, otherwise if it does in multiple scans then called multiple pass assembler. Here assembler divide these tasks in two passes:

- **Pass-1:**
  1. Define symbols and literals and remember them in symbol table and literal table respectively.
  2. Keep track of location counter
  3. Process pseudo-operations
- **Pass-2:**
  1. Generate object code by converting symbolic op-code into respective numeric op-code

2. Generate data for literals and look for values of symbols

Firstly, We will take a small assembly language program to understand the working in their respective passes. Assembly language statement format:

[Label] [Opcode] [operand]

Example: M ADD R1, ='3'

where, M - Label; ADD - symbolic opcode;

R1 - symbolic register operand; (='3') - Literal

Assembly Program:

```
Label  Op-code   operand   LC value(Location counter)
JOHN   START     200
       MOVER     R1, ='3'    200
       MOVEM     R1, X       201
L1     MOVER     R2, ='2'    202
       LTORG                 203
X      DS        1           204
       END                   205
```

Let's take a look on how this program is working:

1. **START:** This instruction starts the execution of program from location 200 and label with START provides name for the program.(JOHN is name for program)
2. **MOVER:** It moves the content of literal(='3′) into register operand R1.
3. **MOVEM:** It moves the content of register into memory operand(X).
4. **MOVER:** It again moves the content of literal(='2′) into register operand R2 and its label is specified as L1.
5. **LTORG:** It assigns address to literals(current LC value).
6. **DS(Data Space):** It assigns a data space of 1 to Symbol X.
7. **END:** It finishes the program execution.

Working of Pass-1: Define Symbol and literal table with their addresses. Note: Literal address is specified by LTORG or END.

Step-1: START 200 (here no symbol or literal is found so both table would be empty)

Step-2: MOVER R1, ='3′ 200 (='3′ is a literal so literal table is made)

Literal	Address

Literal	Address
=’3′	– – –

Step-3: MOVEM R1, X 201

X is a symbol referred prior to its declaration so it is stored in symbol table with blank address field.

Symbol	Address
X	– – –

Step-4: L1 MOVER R2, =’2′ 202

L1 is a label and =’2′ is a literal so store them in respective tables

Symbol	Address
X	– – –
L1	202

Literal	Address
=’3′	– – –
=’2′	– – –

Step-5: LTORG 203

Assign address to first literal specified by LC value, i.e., 203

Literal	Address
=’3′	203
=’2′	– – –

Step-6: X DS 1 204

It is a data declaration statement i.e X is assigned data space of 1. But X is a symbol which was referred earlier in step 3 and defined in step 6.This condition is called Forward Reference

Problem where variable is referred prior to its declaration and can be solved by back-patching. So now assembler will assign X the address specified by LC value of current step.

Symbol	Address
X	204
L1	202

Step-7: **END** **205**

Program finishes execution and remaining literal will get address specified by LC value of END instruction. Here is the complete symbol and literal table made by pass 1 of assembler.

Symbol	Address
X	204
L1	202

Literal	Address
=’3′	203
=’2′	205

Now tables generated by pass 1 along with their LC value will go to pass-2 of assembler for further processing of pseudo-opcodes and machine op-codes.

Working of Pass-2:

Pass-2 of assembler generates machine code by converting symbolic machine-opcodes into their respective bit configuration(machine understandable form). It stores all machine-opcodes in MOT table (op-code table) with symbolic code, their length and their bit configuration. It will also process pseudo-ops and will store them in POT table(pseudo-op table).

Various Data bases required by pass-2:

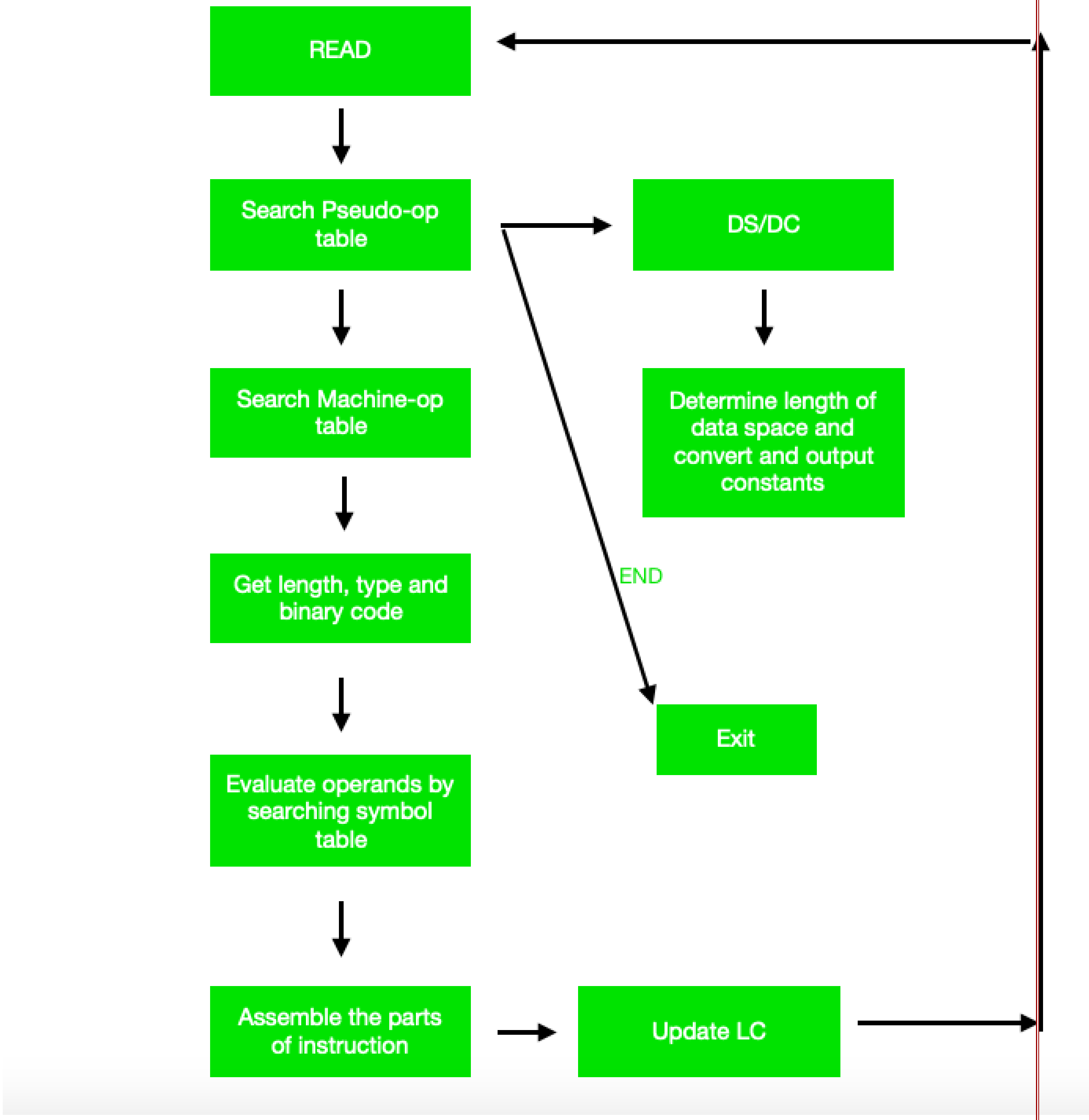
1. MOT table(machine opcode table)

2. POT table(pseudo opcode table)

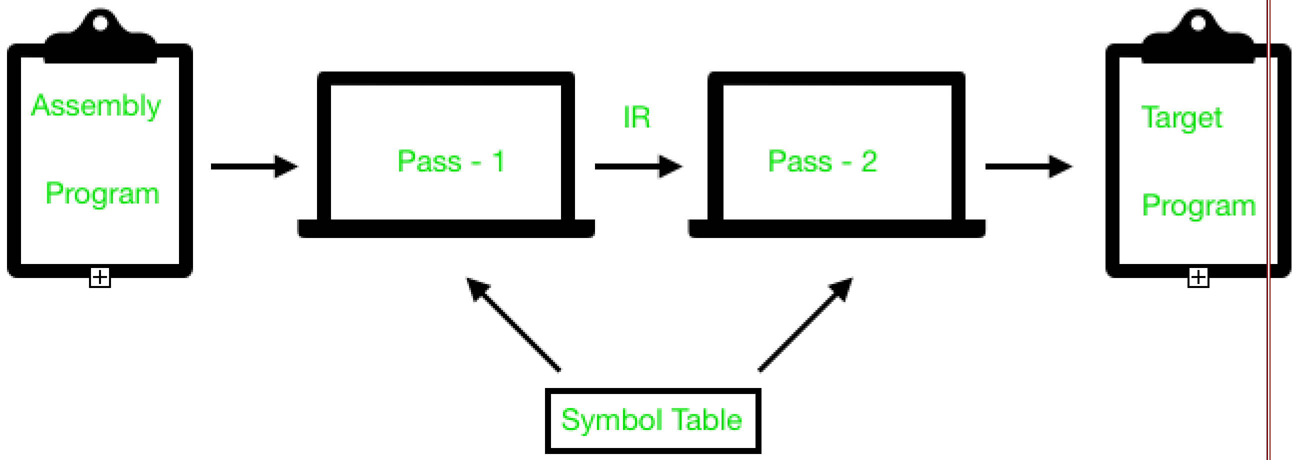
3. Base table(storing value of base register)

4. LC (location counter)

Take a look at flowchart to understand:


READ
Search Pseudo-op table
DS/DC
Search Machine-op table
Determine length of data space and convert and output constants
Get length, type and binary code
END
Exit
Evaluate operands by searching symbol table
Assemble the parts of instruction
Update LC

As a whole assembler works as:



A Single pass ASSEMBLER for IBM PC

A Single Pass Assembler for IBM PC Single pass assembler for the Intel 8088 processor used in IBM PC. Focuses on the design features for handling the forward reference problem in an environment using segment-based addressing. 1. Architecture of Intel 8088. 2. Intel 8088 Instructions. 3. Assembly Language of Intel 8088. 4. Problems of Single pass assembly 5. Design of the Assembler.

1. The Architecture of Intel 8088

 Supports 8 and 16 bit arithmetic. Provides special instructions for string manipulation. The CPU contains following features – 1. Data registers AX, BX, CX and DX 2. Index registers SI and DI 3. Stack pointer registers BP and SP 4. Segment registers Code, Stack, Data and Extra.

a)

AH	BH	CH	DH
AL	BL	CL	DL

b)

BP	SP

c)

SI DI

d)

Code Stack Data Extra

AX BX CX DX

Fig:- a) Data b) Base c) Index d) Segment registers

⎫ Each data register is 16 bits in size, split into upper and lower halves. ⎫ Either half can be used for 8 bit arithmetic, while the two halves together constitute the data register for 16 bit arithmetic. ⎫ Architecture supports stacks for storing subroutine and interrupt return addresses, parameters and other data. ⎫ The index registers SI and DI are used to index the source and destination addresses in string manipulation instructions. ⎫ Two stack pointer registers called SP and BP are provided to address the stack. Push and Pop instructions are provided.

⎫ The Intel 8088 provides addressing capability for 1 MB of primary memory. ⎫ The memory is used to store three components of program, Program code, Data and Stack. ⎫ The Code, Stack and Data segment registers are used to contain the start addresses of these three components. ⎫ The Extra segment register points to another memory area which can be used to store data. ⎫ The size of each segment is limited to 216 i.e 64 K bytes.

⎫ The 8088 architecture provides 24 addressing modes. ⎫ In the Immediate addressing mode, the instruction itself contains the data that is to participate in the instruction. This data can be 8 or 16 bits in length. ⎫ In the Direct addressing mode, the instruction contains 16 bit number which is taken to be displacement from the segment base contained in segment register. ⎫ In the Indexed mode, contents of the index register indicated in the instruction (SI or DI) are added to the 8 or 16 bit displacement contained in the instruction.

⎫ In the Based mode, contents of the base register are added to the displacement. ⎫ The based-and-indexed with displacement mode combines the effect of the based and indexed modes.

Addressing mode

Example

Remarks

Immediate

MOV SUM, 1234H

Data= 1234H

Register

MOV SUM, AX

AX contains the data

Direct

MOV SUM, [1234H]

Data disp.= 1234H

Register indirect

MOV SUM, [BX]

Data disp.= (BX)

Based

MOV SUM, 12H [BX]

Indexed

MOV SUM, 34H [SI]

Data disp.= 12H+ (BX) Data disp.= 34H+ (SI)

Based & Indexed MOV SUM, 56H [SI] [BX] Data disp.= 56H+ (SI) + (BX) Addressing modes of 8088

2. Intel 8088 Instructions Arithmetic Instructions Operands can be in one of the four 16 bit registers or in memory location designated by one of the 24 addressing modes. Three instruction formats are as shown in figure. The mod and r/m fields specify first operand, which can be in register or in memory. The reg field describes the second operand, which is always a register. The instruction opcode indicates which instruction format is applicable.

 The direction field (d) indicates which operand is the destination operand. If d=0, the register/memory operand is the destination, else the register operand indicated by reg is the destination. The width field (w) indicates whether 8 or 16 bit arithmetic is to be used.

a) Register/Memory to Register opcode d

w

mod reg r/m

b) Immediate to Register/Memory opcode d

w

mod reg r/m

data

c) Immediate to Accumulator opcode w

data

data

data

r/m

mod= 00

mod= 01

mod= 10

mod= 11 w=0 w=1

000

(BX)+(SI)

(BX)+(SI)+ d8

Note 2

AL

AX

001

(BX)+(DI)

(BX)+(DI)+d8

Note 2

CL

CX

010

(BP)+(SI)

(BP)+(SI)+ d8

Note 2

DL

DX

011		
(BP)+(DI)		
(BP)+(DI)+		d8
Note		2
BL		
BX		
100		
(SI)		
(SI)	+	d8
Note		2
AH		
SP		
101		
(DI)		
(DI)	+	d8
Note		2
CH		
BP		
110		

Note 1

(BP) + d8

Note 2

DH

SI

111

(BX)

(BX) + d8

Note 2

BH

DI

Note 1: (BP)+ DISP for indirect addressing, d16 for direct Note 2: Same as previous column, except d16 instead of d8 reg

Register 8 bit (w=0)

16 bit (w=1)

000

AL

AX

001

CL

CX

010

DL

DX

011

BL

BX

100

AH

SP

101

CH

BP

110

DH

SI

111

BH

DI

Control Transfer Instructions Two groups of control transfer instructions are supported. 1. Calls, jumps and returns 2. Iteration control instructions Calls, jumps and returns can occur within the same segment or can cross segment boundaries. Intra-segment transfers are preferably assembled using a self-relative displacement. The longer form of intra-segment transfers uses a 16 bit logical address within the segment. Inter-segment transfers indicate a new segment base and an offset.

 Control transfers can be both direct and indirect. Their instruction formats are :a) Intra-segment Opcode

Disp. low

Disp. high

b) Inter-segment Opcode

Offset

Offset

Segment

Base

c) Indirect

Opcode

mod 100 r/m

Disp. low

Disp. high

Formats of Control Transfer Instruction

 Iteration control operations perform looping decisions in string operations. Example:- Consider the program MOV MOVMOV CLD REP

SI, 100H DI, 200H CX, 50H

MOVSB

; Source address ; Destination address ; No. of bytes ; Clear direction flag ; Move 80 bytes

3. The Assembly Language of Intel 8088 1) Statement Format [Label:] opcode operand(s) ; comment string 2) Assembler Directives a) Declarations - Declaration of constants and reservation of storage are both achieved in the same direction A DB 25 ; Reserve byte & initialize B DW ? ; Reserve word, no initialization C DD 6DUP(0) ; 6 Double words, all 0's

b) EQU and PURGE EQU defines symbolic names to represent values PURGE undefined the symbolic names. That name can be reused for other purpose later in the program. Example:XYZ DB ? ABC EQU XYZ ; ABC represents name XYZ PURGE ABC ; ABC no longer XYZ ABC EQU 25 ; ABC now stands for '25'

UNIT III

MACROS and MACRO PROCESSORS

Macro Processor

- Last Updated : 06 Oct, 2020

A Macro instruction is the notational convenience for the programmer. For every occurrence of macro the whole macro body or macro block of statements gets expanded in the main source code. Thus Macro instructions make writing code more convenient.

Salient features of Macro Processor:

- **Macro** represents a group of commonly used statements in the source programming language.
- Macro Processor replaces each macro instruction with the corresponding group of source language statements. This is known as the expansion of macros.
- Using Macro instructions programmer can leave the mechanical details to be handled by the macro processor.
- Macro Processor designs are not directly related to the computer architecture on which it runs.
- Macro Processor involves definition, invocation, and expansion.

Macro Definition and Expansion:

```
Line        Label        Opcode        Operand

5           COPY         START         0
10          RDBUFF       MACRO         &INDEV, &BUFADR
15
.
.
90
95                       MEND
```

- **Line 10:**
  RDBUFF (Read Buffer) in the Label part is the name of the Macro or definition of the Macro. &INDEV and &BUFADR are the parameters present in the Operand part. Each parameter begins with the character &.
- **Line 15 – Line 90:**
  From Line 15 to Line 90 Macro Body is present. Macro directives are the statements that make up the body of the macro definition.
- **Line 95:**
  MEND is the assembler directive that means the end of the macro definition.

Macro Invocation:

```
Line        Label        Opcode        Operand

180         FIRST        STL           RETADR
190         CLOOP        RDBUFF        F1, BUFFER
15
.
.
255                      END           FIRST
```

Line 190:
RDBUFF is the Macro invocation or Macro Call that gives the name of the macro instruction being invoked and F1, BUFFER are the arguments to be used in expanding the macro. The statement that form the expansion of a macro are generated each time the macro is invoked.

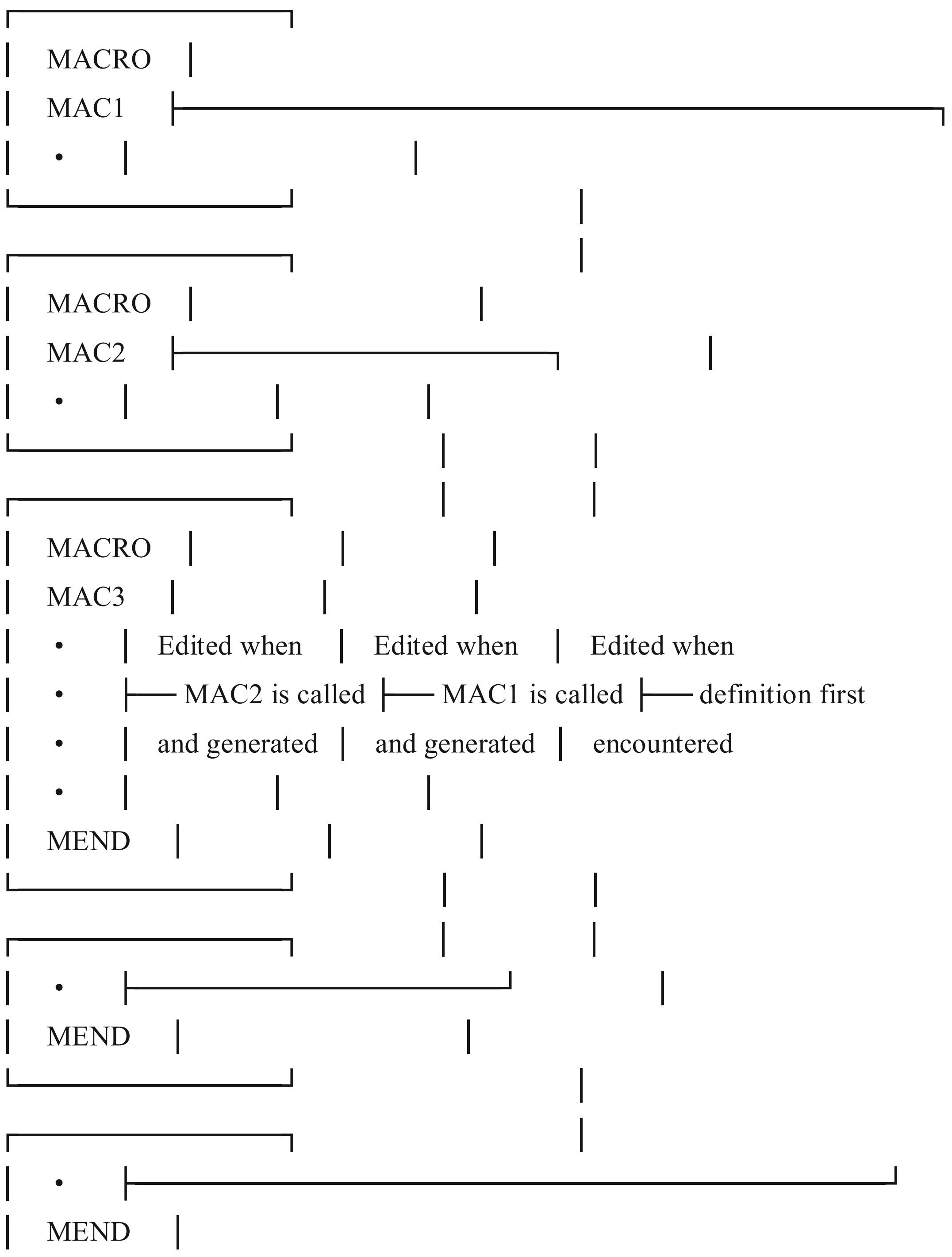
Nesting macro instruction definitions

A nested macro instruction definition is a macro instruction definition you can specify as a set of model statements in the body of an enclosing macro definition. This lets you create a macro definition by expanding the outer macro that contains the nested definition.

All nested inner macro definitions are effectively "black boxes": there is no visibility to the outermost macro definition of any variable symbol or sequence symbol within any of the nested macro definitions. This means that you cannot use an enclosing macro definition to tailor or parameterize the contents of a nested inner macro definition.

High Level Assembler allows both inner macro instructions and inner macro definitions. The inner macro definition is not edited until the outer macro is generated as the result of a macro instruction calling it, and then only if the inner macro definition is encountered during the generation of the outer macro. If the outer macro is not called, or if the inner macro is not encountered in the generation of the outer macro, the inner macro definition is never edited. Figure 1 shows the editing of inner macro definitions.

Figure 1. Editing inner macro definitions


MACRO
MAC1
•
MACRO
MAC2
•
MACRO
MAC3
•
•
•
•
MEND
Edited when MAC2 is called and generated
Edited when MAC1 is called and generated
Edited when definition first encountered
•
MEND
•
MEND

First MAC1 is edited, and MAC2 and MAC3 are not. When MAC1 is called, MAC2 is edited (unless its definition is bypassed by an AIF or AGO branch); when MAC2 is called, MAC3 is edited. No macro can be called until it has been edited.

There is no limit to the number of nestings allowed for inner macro definitions.

The lack of parameterization can be overcome in some cases by using the AINSERT statement. This lets you generate a macro definition from within another macro generation. A simple example is shown at Where to define a macro in a source module. In Figure 2, macro ainsert_test_macro generates the macro mac1 using a combination of AINSERT and AREAD instructions. The mac1 macro is then called with a list of seven parameters.

NESTED MACRO CALLS :-

```
e.g    MACRO
       INCR & R, & MEM , & VAL, & R
       MOVER & R, & MEM                   Definition of MACRO INCR,
       ADD & R, & VAL                     which is called in CALCULATE
       MOVEM & R, & MEM                   Macro definition
       MEND
       MACRO
       CALCULATE                      Definition of MACRO CALCULATE
       MOVER CREG, LOC                which is calling INCR macro
       INCR A, B, CREG, LOC
       MEND
       START 100
       CALCULATE  ------->            calling of Macro CALCULATE
       STOP
       END
  Expansion of MACRO CALCULATE:-
             START 100
             MOVER CREG, LOC
             MOVER CREG, A
             ADD CREG, B             Expansion of          Expansion of
             MOVEM CREG, A           INCR Macro            CALCULATE
             MOVEM CREG, LOC                               Macro
                  STOP
                  END
```

Advanced macro facilities are aimed at supporting semantic expansion. These facilities can be grouped into :

a) Facilities for alteration of flow of control during expansion

b) Expansion Time Variables

c) Attributes of parameters.

d) Facilities for alteration of flow of control during expansion.

a) Facilities for alteration of flow of control during expansion

Expansion with statements AIF, AGO and ANOP.

A sequencing symbol (SS) has the syntax

< ordinary String >

As SS is defined by putting it in the field ' LABEL' of a statement in the macro body. This LABEL field will act as target address on which control is transferred for conditional as well as unconditional way. It never appear in the expanded form of a model statement.

Syntax of AIF :-   conditional jump

AIF (< expression >) < LABEL sequential symbol >

Where < expression > is formal parameters and their attributes like T, L,S (Type , Length and size). If expression is true, control is transferred to LABEL or sequential symbol

Syntax of AGO   Unconditional jump

AGO < sequential symbol >

Without checking condition control is transferred to LABEL.

An ANOT statement is written as

< Sequential symbol >   ANOP.

Which will simply act as LABEL.

Example of altering flow of control during expansion :-

```
        MACRO

        EVAL & X, & Y

AIF ( & Y EQ & X ) AGAIN

        AGO NEXT

AGAI N : ANOP

          MOVER AREG, BREG

NEXT : ANOP

        MEND
```

1. AIF (& Y EQ & X) AGAIN If value of X = Y then it will jump on label again i.e conditional jump.
2. AGO NEXT Unconditionally it will go on NEXT
3. Every label is having first statement as ANOP.

b) Expansion Time Variables :-

EV are used during expansion of macros A local EV is created for use inside a particular MACRO. A global EV exists across all macro calls. Syntax for local and global EV's

```
LCL    < EV    specification >

GBL    < EV    specification >
```

Where < EV specification > has the syntax &< EV Name >

Where < EV Name > is an ordinary string.

Values of EV's can be manipulated by SET statement . A SET statement is written as

< EV Specification > SET < SET – expression >

Here < EV specification > appears in the label field and SET in mnemonic field . A SET statement assigns the value of < SET- expression > to the < EV specification.

e.g.

```
        MACRO

        CALC

        LAL         & A, & B

& A     SET         1

& B     SET         5

        MEND
```

A call on macro CALC is expanded by creating two local EV A & B . The first SET statement assigns value '1' to A and second SET statement assigns value 's' to B.

c) Attributes of formal parameters:-

An attribute is written using the syntax

< attribute name >' < formal parameter > and represents information about the value of the formal parameter. These attributes are type, length and size have the names T, L and S

e.g

```
        MACRO

        CALC & B

        AIF ( L' & A EQ 1 )    NEXT
```

```
      NEXT :

                MEND
```

Here expression control is transferred to NEXT only if Length of A. is equal to 1.

A **general-purpose macro processor** or **general purpose** **preprocessor** is a macro processor that is not tied to or integrated with a particular language or piece of software.

A macro processor is a program that copies a stream of text from one place to another, making a systematic set of replacements as it does so. Macro processors are often embedded in other programs, such as assemblers and compilers. Sometimes they are standalone programs that can be used to process any kind of text.

Macro processors have been used for language expansion (defining new language constructs that can be expressed in terms of existing language components), for systematic text replacements that require decision making, and for text reformatting

UNIT IV

COMPILERS AND INTERPRETERS

Principles of Compilers

Introduction

The word *compilation* is used to denote the task of translating *high level language* (HLL) programs into machine language programs. Though the objective of this task of translation is similar to that of an *assembler*, the problem of compilation is much more complex than that of an assembler. A *compiler* is a program that does the compilation task. A compiler recognises programs in a particular HLL and produces equivalent output programs appropriate for some particular computer configuration (hardware and OS). Thus, an HLL program is to a great extent independent of the configuration of the machine it will eventually run on, as long as it is ensured that the program is compiled by a compiler that recognises that HLL and produces output for the required machine configuration. It is common for a machine to have compilers that would translate programs to produce executables for that machine (*hosts*). But there also are compilers that runs on one type of machine but the output of which are programs that shall run on some other machine configuration, such as generating an MS-DOS executable program by compiling an HLL program in UNIX. Such a compiler is called a *cross compiler*. Another kind of translator that accepts programs in HLL are known as *interpreters*. An interpreter translates an input HLL program and *also runs the program on the same machine*. Hence the output of running an interpreter is actually the output of the program that it translates.

Important phases in Compilation

The following is a typical breakdown of the overall task of a compiler in an approximate sequence -

Lexical analysis, Syntax analysis, Intermediate code generation, Code optimisation, Code generation.

Like an assembler, a compiler usually performs the above tasks by making multiple passes over the input or some intermediate representation of the same. The compilation task calls for intensive processing of information extracted from the input programs, and hence data structures for representing such information needs to be carefully selected. During the process of translation a compiler also detects certain kinds of errors in the input, and may try to take some recovery steps for these.

Lexical Analysis

Lexical analysis in a compiler can be performed in the same way as in an assembler. Generally in an HLL there are more number of tokens to be recognised - various keywords (such as, *for, while, if, else*, etc.), punctuation symbols (such as, comma, semi-colon, braces, etc.), operators (such as arithmatic operators, logical operators, etc.), identifiers, etc. Tools like *lex* or *flex* are used to create lexcical analysers.

Syntax Analysis

Syntax analysis deals with recognising the structure of input programs according to known set of *syntax rules* defined for the HLL. This is the most important aspect in which HLLs are significantly different from lower level languages such as assembly language. In assembly languages the syntax rules are simple which roughly requires that a program should be a sequence of statements, and each statement should esentially contain a mnemonic followed by zero or more operands depending on the mnemonic. Optionally, there can be also be an identifier preceding the mnemonic. In case of HLLs, the syntax rules are much more complicated. In most HLLs the notion of a statement itself is very flexible, and often allows *recursion*, making nested constructs valid. These languages usually support multiple data types and often allow programmers to define abstruct data types to be used in the programs. These and many other such features make the process of creating software easier and less error prone compared to assembly language programming. But, on the other hand, these features make the process of compilation complicated.

The non-trivial syntax rules of HLLs need to be cleverly specified using some suitable notation, so that these can be encoded in the compiler program. One commonly used formalism for this purpose is the *Context Free Grammar (CFG)*. CFG is a formalism that is more powerful than *regular grammars* (used to write regular expressions to describe *tokens* in a lexical analyser). Recursion, which is a common feature in most constructs of HLLs, can be defined using a CFG in a concise way, whereas a regular grammar is incapable of doing so. It needs to be noted that there are certain constructs that cannot be adequately described using CFG, and may require other more powerful formalisms, such as *Context Sensitive Grammars (CSG)*. A common notation used to write the rules of CFG or CSG is the *BNF (Backus Naur Form)*.

During syntax analysis, the compiler tries to apply the rules of the grammar of the input HLL given using BNF, to recognise the structure of the input program. This is called *parsing* and the module that performs this task is called a *parser*. From a somewhat abstract point of view, the output of this phase is a *parse tree* that depicts how various rules of the grammar can be repetitively applied to recognise the input program. If the parser cannot create a parse tree for some given input program, then the input program is not valid according to the syntax of the HLL.

The soundness of the CFG formalism and the BNF notation makes it possible to create different types of efficient parsers to recognise input according to a given language. These parsers can be broadly classified as *top-down parsers* and *bottom-up parsers*. *Recursive descent parsers* and *Predictive parsers* are two examples of top-down parsers. *SLR parsers* and *LALR parser* are two examples of bottom-up parsers. For certain simple context free languages (languages that can be defined using CFG) simpler bottom-up parsers can be written. For example, for recognising mathematical expressions, an *operator precedence parser* can be created.

In creating a compiler, a parser is often built using tools such as *yacc* and *bison*. To do so the CFG of the input language is written in BNF notation, and given as input to the tool (along with other details).

Intermediate Code Generation

Having recognised a given input program as valid, a compiler tries to create the equivalent program in the language of the target environment. In case of an assembler this translation was somewhat simpler since the operation implied by the mnemonic opcode in each statement in the input program, there is some equivalent machine opcode. The number of operands applicable for each operation in the machine language is the same as allowed for the corresponding assembly language mnemonic opcodes. Thus for the assembly language the translation for each statement can be done for each statement almost independently of the rest of the program. But, in case of an HLL, it is futile to try to associate a single machine opcode for each statement of the input language. One of the reasons for this is, as stated above, the extent of a statement is not always fixed and may contain recursion. Moreover, data references in HLL programs can assume significant levels of abstractions in comparision to what the target execution environment may directly support. The task of associating meanings (in terms of primitive operations that can be supported by a machine) to programs or segments of a program is called *semantic processing*.

Syntax Directed Translation

Though it is not entirely straightforward to associate target language operations to statements in the HLL programs, the CFG for the HLL allows one to associate *semantic actions* (or implications) for the various syntactic rules. Hence in the broad task of translation, when the input program is parsed, a compiler also tries to perform certain semantic actions corresponding to the various syntactic rules that are eventually applied. However, most HLLs contain certain syntactic features for which the semantic actions are to be determined using some additional information, such as the contents of the symbol table. Hence, building and usage of data-structures such as the symbol table are an important part of the semantic action that are performed by the compiler.

Upon carrying out the semantic processing a more manageable equivalent form of the input program is obtained. This is stored (represented) using some *Intermediate code* representation that makes further processing easy. In this representation, the compiler often has to introduce several temporary variables to store intermediate results of various operations. The language used for the intermediate code is generally not any particular machine language, but is such which can be efficiently converted to a required machine language (some form of assembly language can be considered for such use).

Code Optimisation

The programs represented in the intermediate code form usually contains much scope for optimisation both in terms of storage space as well as run time efficiency of the intended output program. Sometimes the input program itself contains such scope. Besides that, the process of generating the intermediate code representation usually leaves much room for such optimisation. Hence, compilers usually implement explicit steps to optimise the intermediate code.

Code Generation

Finally, the compiler converts the (optimised) program in the intermediate code representation to the required machine language. It needs to be noted that if the program being translated by the compiler actually has dependencies on some external modules, then *linking* has to be performed to the output of the compiler. These activities are independent of whether the input program was in HLL or assembly language.

One of the important aspects of the semantic actions of a compiler is to ensure an efficient and

error free run-time storage model to the output

Importance of binding times

- The binding time of an entity's attributes determines the manner in which a language processor can handle use of the entity in the program.
- A compiler can tailor the code generated to access an entity if a relevant binding was performed before or during compilation time.
- However,suchtailoringisnotpossibleifthebindingisperformedlaterthan compilation time. Sothecompilerhastogenerateageneralpurposecodethatwould findinformation

  about the relevant binding during its execution and use it to access the entity appropriately. It affects execution efficiency of the target program.

emory Allocation

program. Most modern programming languages allows some extent of *block-structuring*, nesting of constructs, and recursion of subroutines. All these calls for an efficient modelling of data storage that is dynamic, and it turns out that a *stack* meets much of the criteria. Thus allocation and access of storage for program variables, subroutine parameters, and compiler generated internal variables on the stack is an important part of the task of a compiler.

Memory allocation is primarily a computer hardware operation but is managed through operating **system** and **software** applications. ... Once the program has finished its operation or is idle, the **memory** is released and **allocated** to another program or merged within the primary **memory**.

There are two **types of memory allocation**. 1) Static **memory allocation** -- **allocated** by the compiler. Exact size and **type of memory** must be known at compile time. 2) Dynamic **memory allocation** -- **memory allocated** during run time.

Memory allocation is the process of assigning blocks of **memory** on request. Typically the allocator receives **memory** from the operating system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks. It must also make any returned blocks available for reuse.

A partition **allocation method** is considered **better** if it avoids internal fragmentation. When it is time to load a **process** into the main **memory** and if there is more than one free block of **memory** of sufficient size then the OS decides which free block to **allocate**. 1.06-Nov-2020 time.

Memory Allocation in block structured language

- The block is a sequence of statements containing the local data and declarations which are enclosed within thedelimiters.

Ex:

```
A
{
    Statements
    .....
}
```

- The delimiters mark the beginning and the end of the block. There can be nested blocks for ex: block B2 can be completely defined within the block B1.
- A block structured language usesdynamicmemoryallocation.
- Findingthescopeofthevariablemeanscheckingthevisibilitywithintheblock
- Following are the rulesusedtodetermine the scope of the variable:
  1. Variable X is accessed within the block B1 if it can be accessed by any statement situated in blockB1.
  2. Variable X is accessed by any statement in block B2 and block B2 is situated in block B1.
- Therearetwotypesofvariablesituated in theblockstructuredlanguage
  1. Local variable
  2. Non localvariable
- To understand local and non-local variable consider the following example Procedure A

```
{
    int
    x,y
    ,z
    Pr
    oc
    ed
    ur
    e B
    {
      Inta,b
  }
    Procedure C
    {
```

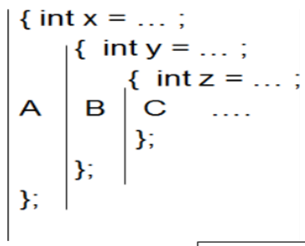
```
                    Intm,n
                }
            }
```

Procedure	Local variables	Nonlocalvariables
A	x,y,z	
B	a,b	x,y,z
C	m,n	x,y,z

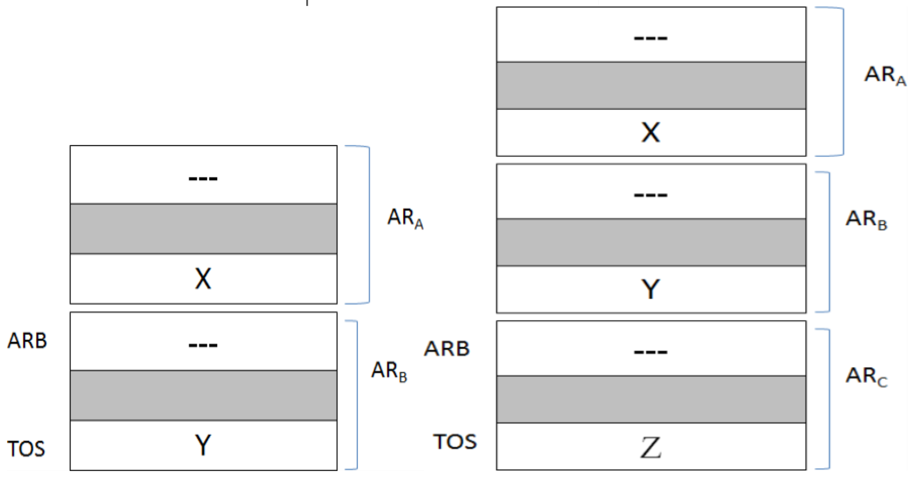
- Variables x, y and z are local variables to procedure A but those are non-local to block B and C because these variable are not defined locally within the block B and C but are accessible within theseblocks.
- Automaticdynamicallocationisimplementedusingtheextendedstackmodel.
- Each record in the stackhastworeserved pointers instead of one.
- Each stack record accommodates the variable for one activation of a block, which we call an activation record(AR).

Dynamic pointer

- The first reserved pointer in block's AR points to the activation record of its dynamic parent. This is called dynamicpointerand hastheaddress0 (ARB).
- The dynamic pointer is used for de-allocating an AR.
- Followingexampleshowsmemoryallocationforprogramgivenbelow.



```
{ int x = ... ;
     { int y = ... ;
          { int z = ... ;
A    B    C     ....
          };
     };
};
```



Static pointer

- Access to non local variable is implemented using the second reserved pointer in AR.

This pointer which hastheaddress 1 (ARB) is calledthestaticpointer.

Activation record

- Theactivationrecordisablockofmemoryusedformanaginginformationneeded bya single execution of a procedure.

Return value
Actual parameter
Control link
Access link
Saved M/c status
Local variables
Temporaries

1. Temporary values: The temporary variables are needed during the evaluation of expressions. Such variables are stored in the temporary field of activation record.
2. Localvariables:Thelocaldataisadatathatislocaltotheexecutionprocedure is stored in this field of activation record.
3. Saved machine registers: This field holds the information regarding the status of machine just before the procedure is called. This field contains the registers and program counter.
4. Control link: Thisfieldisoptional. Itpointstotheactivation record ofthe calling procedure. This link is also called dynamic link.
5. Access link: This field is also optional. It refers to the non local data in other activation record. This field is also called static link field.
6. Actualparameters:Thisfieldholdstheinformationabouttheactual parameters. Theseactualparametersarepassedtothe called procedure.
7. Return values:Thisfieldisusedtostoretheresultofa function call.

Compilation of Expression

Operand Descriptor

An operand descriptor has the following fields:

1. **1.** Attributes:Containsthesubfieldstype,lengthandmiscellaneousinformation
2. **2.** Addressability: Specifies where the operand is located, and how it can be accessed. It has two subfields

- Addressabilitycode:Takesthevalues'M'(operandisinmemory),and'R'(operandis in register). Other addressability codes, e.g. address in register ('AR') and address in memory ('AM'), are alsopossible,

- Address:Address of a CPU register or memory word.
- Ex: a*b

  MOVER AREG,
  A MULT
  AREG, B
  Three operand descriptors are used during code generation. Assuming a, b to be integers occupying 1 memory word, these are:

Attribute	Addressability
(int, 1)	Address(a)
(int, 1)	Address(b)
(int, 1)	Address(AREG)

Register descriptors

A register descriptor has two fields

1. Status: Containsthecodefreeoroccupiedtoindicateregisterstatus.
2. Operand descriptor #: If status = occupied, this field contains the descriptor for the operand contained in theregister.

- Register descriptors are stored in an array called Register_descriptor. One register descriptor exists for each CPU register.
- InaboveExampletheregisterdescriptorforAREGaftergeneratingcodefora*bwould be

  Occupied #3
- ThisindicatesthatregisterAREGcontainstheoperanddescribedbydescriptor#3.

Intermediate code for expression

There are two types of intermediate representation

1. Postfix notation
2. Three addresscode.

1) Postfix notation

- Postfix notation is a linearized representation of a syntax tree.
- italistofnodesofthetreeinwhichanodeappearsimmediatelyafteritschildren
- thepostfixnotationofx=-a*b+-
  a*bwillbe x a –b * a-b*+=

2) Three address code

- In three address code form at the most three addresses are used to represent statement. The generalformofthreeaddresscoderepresentation is -a:=bopc
- Wherea,b or c are theoperandsthat can benames, constants.

- For the expression like a = b+c+d the three address code will be t1=b+c
  t2=t1+d
- Heret1andt $_2$arethetemporarynamesgeneratedbythecompiler.Therearemost three addressesallowed.Hence,thisrepresentationisthree-addresscode.
- There are three representations used for three code such as quadruples, triples and indirect triples.

Quadruple representation

- Thequadrupleisastructurewithatthemosttourfieldssuchasop,arg1,arg2andresult.
- The op field is used to represent the internal code for operator, the arg1 and arg2 represent the two operands used and result field is used to store the result of an expression.
- Consider the input statement x:= -a*b + -a*b

		Op	Arg1	Arg2	result
t1=uminus a	(0)	uminus	a		t1
t_2:= t1 * b	(1)	*	t1	b	t2
t_3= - a	(2)	uminus	a		t3
t_4:= t_3* b	(3)	*	t3	b	t4
t_5:= t_2+ t_4	(4)	+	t2	t4	t5
x= t_5	(5)	:=	t5		X

Triples

- The triple representation the use of temporary variables is avoided by referring the pointers in the symbol table.
- theexpression x : = - a * b + - a * b thetriple representation isasgivenbelow

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	:=	X	(4)

Indirect Triples

- Theindirecttriplerepresentationthelistingoftriplesisbeendone.Andlisting pointers are used instead of using statements.

Number	Op	Arg1	Arg2
(0)	uminus	a	
(1)	*	(11)	b
(2)	uminus	a	
(3)	*	(13)	b
(4)	+	(12)	(14)
(5)	:=	X	(15)

	Statement
(0)	(11)
(1)	(12)
(2)	(13)
(3)	(14)
(4)	(15)
(5)	(16)

Code Optimization

I. <u>Compile Time Evaluation</u>

- Compiletimeevaluationmeansshiftingofcomputationsfromruntimetocompilation.
- Therearetwomethodsused to obtain the compile time evaluation.

1. Folding

- Inthefoldingtechniquethecomputationofconstantisdoneatcompiletime instead of runtime.

example : length = (22/7) * d

- Here foldingis implied byperformingthe computation of22/7at compiletime

2. Constant propagation

- In this technique the value of variable is replaced and computation of an expression is done at the compilation time.

example:pi=3.14;r=5;

Area=pi*r*r

- Hereatthecompilationtimethevalueofpiisreplacedby3.14andrby5 then computation of 3.14 * 5 * 5 is done during compilation.

II. <u>Common Sub Expression Elimination</u>

- The common sub expression is an expression appearing repeatedly in the program which is computedpreviously.
- Theniftheoperandsofthissubexpressiondonotgetchangedatallthenresultof such sub expression isused instead of recomputing it each time

```
Example: t1 = 4 * i
         t2 = a[t1]
         t3 = 4 * j
         t4 = 4 * i
```

```
t5 = n
t6 = b[t4]+t5
```

- The above code can be optimized using common sub expression elimination

```
t1=4*i
t2=a[t1]
t3=4*j
t5=n
t6=b[t1]+t5
```

- Thecommonsubexpressiont4:=4*iiseliminatedasitscomputationisalreadyint1 and value of i is not been changed from definition to use.

  }

III. Loop invariant computation (Frequency reduction)

- Loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop.
- This method is also called code motion.

- Exam

  ple:

```
while(i

<=max

-1)
{
        sum=sum+a[i];
}
```

Can be

optimized as

```
a N=max-1;
While(i<=N)
{
sum=sum+a[i];
}
```

IV. <u>Strength Reduction</u>

- Strength of certain operators is higher than others.
- For instance strength of * is higher than +.

- In this technique the higher strength operators can be replaced by lower strength operators.
- Example:
```
 for(i=1;i<=50;i++)
{
        count = i x 7;
}
```
- Here we get the count values as7,14,21and soonuptolessthan 50.
- This code can be replaced by using strength reduction as follows temp=7
```
for(i=l;i<=50;i++)
{
        count
        =
        temp;
        temp
        =
        temp+
        7;
 }
```

V. **Dead Code Elimination**

- Avariableissaidtobe**live**inaprogramifthevaluecontainedintoissubsequently.
- On the other hand, the variable is said to be **dead** at a point in a program if the value contained into it is never been used. The code containing ***such a variable*** supposed to be adeadcode.Andanoptimizationcanbeperformedbyeliminating suchadeadcode.
- Example :
```
        i=0;
        if(i==1)
        {
                a=x+5;
        }
```
- **if** statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

UNIT-IV (COMPILERS AND INTERPRETERS)

Overview of Interpretation OR Write a note on Interpreter

An interpreter is system software that translates a given High-Level Language (HLL) program into a low-level one, but it differs from compilers. Interpretation is a real-time activity where an interpreter takes the program, one statement at a time, and translates each line before executing it.

Comparison between Compilers and Interpreters

Compilers	**Interpreters**
Compilers are language processors that are based on the language translation-linking-loading model.	Interpreters are a class of language processors based on the interpretation model.
Generate a target output program as an output, which can be run independently from the source program written in Source Language.	Do not generate any output program; rather theyevaluatethesourceprogramateachtime for execution.
Program execution is separated from compilation and performed only after the entire output program is produced.	Program execution is a part of interpretation and performed on a statement by statement basis.
Target program executes independently and doesnotneedthepresenceofcompilerinthe memory.	The interpreter exists in the memory during interpretation, i.e. it coexists with the source program to be interpreted.
Do not generate the output program for execution if there is any error in any of the source program statements.	Can evaluate and execute program statement unti an error is found.
Need recompilation for generating a fresh output program in target language after each modification in the source program.	The interpreter is independent of program modification issues as it processes the source program each time during execution.
Compilers are suitable for production environment.	Interpreters are suited for program development environment.
Compilers are bound to a specific target machine and cannot be ported.	Canbemadeportablebycarefullycodingthem in higher level language.

Comparing the Performance of Compilers and Interpreters

- Comparative performance of a compiler and an interpreter can be realized by inspecting the averageCPUtimecostfordifferentkindsofprocessingofa statement.
- Let t_i, t_c and t_e be the interpretation-time statement, compilation-time statement, and execution-time of compiled statement, respectively.

Itisassumedthatt $_c \approx t_i$sincebooththecompilersandinterpretersinvolvelexical,syntax, and semantic analyses of the source statement. In addition, the code generation effort for a statement performed by the compiler is of the same order of magnitude as the effort involved in the interpretation of the statement.

- If a 400-statement program is executed on a test data with only 80 statements being visited during the test run, the total CPU time in compilation followed by the execution of the programis400*t $_c$+80*t $_e$,whilethetotalCPUtimeininterpretation oftheprogram is80 *t_i= 80 *t_c This shows that the interpretation will be cheaper in such cases. However, if more than 400 statements are to be executed, compilation followed by execution will be cheaper, which means that using interpreter is advantageous up to the executionof400 statements duringtheexecution.This clearly indicates thatfromthe point of view of the CPU time cost, interpreters are a better choice at least for the program development environment.

Benefits of Interpretation

The distinguished benefits of interpretation are as follows:

- Executes the source code directly. It translates the source code into some efficient Intermediate Code (IC) and immediately executes it. The process of execution can be performed in a single stage without theneed of acompilation stage.
- Handles certain language featuresthat cannot be compiled.
- Ensuresportabilitysinceitdoesnotproducemachinelanguageprogram.
- Suited for development environment where a program is modified frequently. This means alteration of code can be performed dynamically.
- Suitedfordebuggingofthecodeandfacilitatesinteractivecodedevelopment.

Java Language Environment

- Java language environment has four key features:
  - o TheJavavirtualmachine(JVM),whichprovidesportabilityofJavaprograms.
  - o An impure interpretive scheme, whose flexibility is exploited to provide a capability for inclusion of program modules dynamically, i.e., during interpretation.
  - o A Java bytecode verifier, which provides security by ensuring that dynamically loadedprogrammodulesdonotinterferewiththeoperationofthe programandthe operating system.
  - o AnoptionalJavajust-in-time(JIT)compiler,whichprovidesefficientexecution.
- Figure 8.1 shows a schematic of the Java language environment. The Java compiler converts a source language program into the Java bytecode, which is a program in the machine

language of the Java virtual machine.

- The Java virtual machine is implemented by a software layer on a computer, which is itself calledtheJavavirtualmachineforsimplicity.Thisschemeprovidesportabilityas theJava bytecodecanbe'executed'onanycomputerthatimplementstheJavavirtual machine.
- The Java virtual machine essentially interprets the bytecode form of a program. The Java compilerandtheJavavirtualmachinethusimplementtheimpureinterpretation scheme.
- Use of an interpretive scheme allows certain elements of a program to be specified during interpretation. This feature is exploited to provide a capability for including program modules calledJavaclassfilesduringinterpretationofaJavaprogram.
- The class loader is invoked whenever a new class file is to be dynamically included in program.TheclassloaderlocatesthedesiredclassfileandpassesittotheJavabytecode verifier.
- The Java bytecode verifier checks whether
    - o The program forges pointers, thereby potentially accessing invalid data or performing branches to invalid locations.
    - o Theprogramviolatesaccessrestrictions,e.g.,byaccessingprivatedata.
    - o The program has type-mismatches whereby it may access data in an invalid manner.
    - o Theprogrammayhavestackoverflowsorunderflowsduringexecution.
- The Java language environment provides the two compilation schemes shown in the lower half of Figure 8.1. The Java Just-In-Time compiler compiles parts of the Java bytecode that are consuming a significant fraction of the execution time into the machine language of the computer to improve their execution efficiency. It is implemented using the schemeof dynamic compilation.
- Afterthe just-in-time compiler has compiled some part of the program, some parts of the Java source program has been converted into the machine language while the remainder of the program still exists in the bytecode form. Hence the Java virtual machineusesamixed- mode executionapproach.

  The other compilation option uses the Java native code compiler shown in the lower partof. It simply compiles the complete Java program into the machine language of a computer. This scheme provides fast execution of the Java program; however, it cannot provide any of the benefits of interpretation or just-in- time compilation.

Java Virtual Machine

- AJavacompilerproducesabinaryfilecalledaclassfilewhichcontainsthebytecodefor a Java program. The Java virtual machine loads one or more class files and executes programs contained in them. To achieve it, the JVM requires the support of the class loader,which locates a required class file, and abytecodeverifier,which ensures that execution ofthe bytecode would not cause any breaches of security.

- The Java virtual machine is a stack machine. By contrast, a stack machine performs computationsbyusingthe values existinginthetop few entrieson a stack and leaving their results on the stack. This arrangement requires that a program should load the values on which it wishes to operate on the stack before performing operations on themandshould take their results from the stack.
- The stackmachine has the following three kinds of operations:
    - o Push operation: This operation has one operand, which is the address of a memory location. The operation creates a new entry at the top of the stack and copies the value that is contained in the specified memory location into thisentry.
    - o Pop operation: This operation also has the address of a memory location as its operand. It performs the converse of the push operation—it copies the value contained in the entry that is at the top of the stack into the specified memory location and also deletes that entry from the stack.
    - o n-ary operation: This operation operates on the values existing in the top n entriesof the stack, deletes the top n entries from the stack, and leaves the result, if any, in the top entry of the stack. Thus, a unary operation operates only on the value contained in the top entry of the stack, a binary operation operates on values contained in the top two entries of the stack, etc.
- A stack machine can evaluate expressions very efficiently because partial results need not be stored in memory—they can be simply left on the stack.

Types of Errors

Syntax Error

- Syntax errors occur due to the fact that the syntax of the programming language is not followed.
- The errors in token formation, missing operators, unbalanced parenthesis, etc., constitute syntax errors.
- Thesearegenerallyprogrammerinducedduetomistakesandnegligencewhile writinga program.

  Syntaxerrorsaredetectedearlyduringthecompilationprocessandrestrictthecompiler to proceed for code generation.
- LetusseethesyntaxerrorswithJavalanguageinthefollowingexamples.

  Example 1: Missing punctuation-

"semicolon" intage=50 // note
here semicolon is missing ***Example2:***
Errorsinexpressionsyntax
x=(30-15; // note the missing closing parenthesis ")"

Semantic Error

- Semanticerrorsoccurduetoimproperuseofprogramminglanguagestatements.
- They include operands whose types are incompatible, undeclared variables, incompatible arguments to function or procedures, etc.
- Semantic errors are mentioned in the following examples.

  Example: Type incompatibility between operands
  intmsg="hello"; //note the types String and int are incompatible

Logical Error

- Logical errors occur due to the fact that the software specification is not followed while writing the program. Although the program is successfully compiled and executed error free, the desired results are not obtained.
- Let us look into some logical errors with Java language.

  Example : Errors in
  computation public
  staticintmul(inta,int
  b){

  ```
      return a + b ;
  }
  ```

  //thismethodreturnstheincorrectvaluewithrespecttothespecificationthat
  requiresto multiply twointegers
  Example : Non-
  terminating loops String
  str=br.readLine();

  ```
  while (str != null) {
      System.out.println(str);
  } // the loop in the code did not terminate
  ```

- Logical errors may causeundesirable effect andprogram behaviors. Sometimes, these errors remain undetected unlesstheresultsare analyzed carefully.

Debugging Procedures

- Wheneverthereisagapbetweenanexpectedoutputandanactualoutputofa program, the program needs to be debugged.

UNIT-IV (COMPILERS AND INTERPRETERS)

- Anerrorinaprogramiscalledbug,anddebuggingmeansfindingandremovingthe errors present in theprogram.
- Debugging involves executingtheprogramina controlled fashion.
- Duringdebugging,theexecutionofaprogramcanbemonitoredateverystep.
- Inthedebugmode,activitiessuchasstartingtheexecutionandstoppingtheexecution are in the hands of the debugger.
- Thedebuggerprovidesthefacilitytoexecuteaprogramuptothespecifiedinstruction by inserting abreakpoint.
- Itgivesachancetoexaminethevaluesassignedtothevariablespresentintheprogram at any instant and, if required, offers an opportunity toupdatetheprogram.
- Types of debugging procedures:
  - o Debug Monitors:
    A debug monitor is a program that monitors the execution of a program and reports the state of a program during its execution. It may interfere in the execution process, depending upon the actions carried out by a debugger (person). In order to initiate the process of debugging, a programmer must compiletheprogramwiththedebug option first. This option, along with other information,generatesatablethatstorestheinformationaboutthevariables usedinaprogramandtheiraddresses.
  - o Assertions:
    Assertions are mechanisms used by a debugger to catch the errors at a stage before the execution of a program. Sometimes, while programming, some assumptions are made about the data involved in computation. If these assumptions went wrong during the execution of the program, it may lead to erroneous results. For this, a programmer can make use of an assert statement. Assertions are the statements used in programs, which are always associated with Boolean conditions. If an assert() statement is evaluated to be true, nothing happens. But if it is realized that the statement is false, the execution program halts.

Classification of Debuggers

Static Debugging

- Static debugging focuses on semantic analysis.
- Inacertainprogram,supposetherearetwovariables:var1andvar2.Thetypeofvar1is an integer,andthetypeofvar2isafloat.Now,theprogramassignsthevalueofvar2to var1; then, there is a possibility that it may not get correctly assigned to the variable due to truncation. This type of analysis fallsunderstaticdebugging.
- Staticdebuggingdetectserrorsbeforethe actual execution.

- Static code analysis may include detection of the following situations:
  - o Dereferencing of variable before assigning a value to it
  - o Truncation of value due to wrong assignment
  - o Redeclarationofvariables
  - o Presence of unreachablecode

Dynamic/Interactive Debugger

Dynamic analysis is carried out during program execution. An interactive debugging system provides programmerswithfacilitiesthataidintestinganddebuggingprogramsinteractively.

A dynamic debugging system should provide the following facilities:

- Execution sequencing: It is nothing but observation and control of the flow of program execution. For example, the program may be halted after a fixed number of instructionsare executed.
- Breakpoints:Breakpointsspecifytheposition within aprogramtillwhichtheprogram gets executedwithoutdisturbance.Oncethecontrolreachessuchaposition,itallows theuser to verify the contents of variables declared in the program.
- Conditional expressions: A debugger can include statements in a program to ensure that certain conditions are reached in the program. These statements, known as assertions, can be used to check whether some pre-condition or post-condition has been met in the program duringexecution.
- Tracing: Tracing monitors step by step the execution of all executable statements present in aprogram.Theothernameforthisprocessis"stepinto".Anotherpossiblevariationis "step over"debuggingthat can be executed atthelevelofprocedureorfunction.This can be implemented by adding a breakpoint at the last executable statement in a program.
- Traceback: This gives a user the chance to traceback over the functions, and the traceback utility uses stack data structure. Traceback utility should show the path by which the current statement in the program was reached.
- Program-display capabilities: While debugging is in progress, the program being debugged must be made visible onthe screen along with the linenumbers.
- Multilingual capability: The debugging system must also consider the language in which the debugging is done. Generally, different programming languages involve differentuser environments and applicationssystems.
- Optimization: Sometimes, to make a program efficient, programmers may use an optimized code. Debugging of such statements can betricky. However, to simplify the debugging process, a debugger may use an optimizing compiler that deals with the followingissues:
  - o Removing invariant expressions from a loop
  - o Merging similarloops
  - o Eliminating unnecessarystatements
  - o Removing branchinstructions

UNIT V

LINKERS

Linker is a program in a system which helps to link a object modules of program into a single object file. It performs the process of linking. Linker are also called link editors. Linking is process of collecting and maintaining piece of code and data into a single file. Linker also link a particular module into system library. It takes object modules from assembler as input and forms an executable file as output for loader.

Linking is performed at both compile time, when the source code is translated into machine code and load time, when the program is loaded into memory by the loader. Linking is performed at the last step in compiling a program.

Source code -> compiler -> Assembler -> Object code -> Linker -> Executable file -> Loader

Linking is of two types:
1. Static Linking –
It is performed during the compilation of source program. Linking is performed before execution in static linking. It takes collection of relocatable object file and command-line argument and generate fully linked object file that can be loaded and run.
Static linker perform two major task:

- **Symbol resolution** – It associates each symbol reference with exactly one symbol definition .Every symbol have predefined task.
- **Relocation** – It relocate code and data section and modify symbol references to the relocated memory location.

The linker copy all library routines used in the program into executable image. As a result, it require more memory space. As it does not require the presence of library on the system when it is run . so, it is faster and more portable. No failure chance and less error chance.

2. Dynamic linking – Dynamic linking is performed during the run time. This linking is accomplished by placing the name of a shareable library in the executable image. There is

more chances of error and failure chances. It require less memory space as multiple program can share a single copy of the library.
Here we can perform code sharing. it means we are using a same object a number of times in the program. Instead of linking same object again and again into the library, each module share information of a object with other module having same object. The shared library needed in the linking is stored in virtual memory to save RAM. In this linking we can also relocate the code for the smooth running of code but all the code is not relocatable.It fixes the address at run time.

Design of a linker

Relocation and linking requirements in segmented addressing

The relocation requirements of a program are influenced by the addressing structure of the computer system on which it is to execute. Use of the segmented addressing structure reduces the relocation requirements of program.

A Linker for MS-DOS

Example : Consider the program of written in the assembly language of intel 8088. The ASSUME statement declares the segment registers CS and DS to the available for memory addressing. Hence all memory addressing is performed by using suitable displacements from their contents. Translation time address o A is 0196. In statement 16, a reference to A is assembled as a displacement of 196 from the contents of the CS register. This avoids the use of an absolute address, hence the instruction is not address sensitive. Now no relocation is needed if segment SAMPLE is to be loaded with address 2000 by a calling program (or by the OS). The effective operand address would be calculated as <CS>+0196, which is the correct address 2196. A similar situation exists with the reference to B in statement 17. The reference to B is assembled as a displacement of 0002 from the contents of the DS register. Since the DS register would be loaded with the execution time address of DATA_HERE, the reference to B would be automatically relocated to the correct address.

Though use of segment register reduces the relocation requirements, it does not completely eliminate the need for relocation. Consider statement 14 .

MOV AX, DATA_HERE

Which loads the segment base of DATA_HERE into the AX register preparatory to its transfer into the DS register . Since the assembler knows DATA_HERE to be a segment, it makes provision to load the higher order 16 bits of the address of DATA_HERE into the AX register. However it does not know the link time address of DATA_HERE, hence it assembles the MOV instruction in the immediate operand format and puts zeroes in the operand field. It also makes an entry for this instruction in RELOCTAB so that the linker would put the appropriate address in the operand field. Inter-segment calls and jumps are handled in a similar way.

Relocation is somewhat more involved in the case of intra-segment jumps assembled in the FAR format. For example, consider the following program :

FAR_LAB EQU THIS FAR ; FAR_LAB is a FAR label

JMP FAR_LAB ; A FAR jump

Here the displacement and the segment base of FAR_LAB are to be put in the JMP instruction itself. The assembler puts the displacement of FAR_LAB in the first two operand bytes of the instruction , and makes a RELOCTAB entry for the third and fourth operand bytes which are to hold the segment base address. A segment like

ADDR_A DW OFFSET A

(which is an ‘address constant’) does not need any relocation since the assemble can itself put the required offset in the bytes. In summary, the only RELOCATAB entries that must exist for a program using segmented memory addressing are for the bytes that contain a segment base address.

For linking, however both segment base address and offset of the external symbol must be computed by the linker. Hence there is no reduction in the linking requirements.

Self-relocation is similar to the relocation process employed by the linker-loader when a program is copied from external storage into main memory; the difference is that it is the loaded program itself rather than the loader in the operating system or shell that performs the relocation.

One form of self-relocation occurs when a program copies the code of its instructions from one sequence of locations to another sequence of locations within the main memory of a single computer, and then transfers processor control from the instructions found at the source locations of memory to the instructions found at the destination locations of memory. As such, the data operated upon by the algorithm of the program is the sequence of bytes which define the program.

Self-relocation typically happens at load-time (after the operating system has loaded the software and passed control to it, but still before its initialization has finished), sometimes also when changing the program's configuration at a later stage during runtime.[3][4]

Examples[edit]

Boot loaders[**edit**]

As an example, self-relocation is often employed in the early stages of bootstrapping operating systems on architectures like IBM PC compatibles, where lower-level chain boot loaders (like the master boot record (MBR), volume boot record (VBR) and initial boot stages of operating systems such as DOS) move themselves out of place in order to load the next stage into memory.

x86 DOS drivers[**edit**]

Under DOS, self-relocation is sometimes also used by more advanced drivers and RSXs/TSRs loading themselves "high" into upper memory more effectively than possible for externally provided "high"-loaders (like LOADHIGH/HILOAD, INSTALLHIGH/HIINSTALL or DEVICEHIGH/HIDEVICE et c.[5] since DOS 5) in order to maximize the memory available for applications. This is down to the fact that the operating system has no knowledge of the inner workings of a driver to be loaded and thus has to load it into a free memory area large enough to hold the whole driver as a block including its initialization code, even if that would be freed after the initialization. For TSRs, the operating system also has to allocate a Program Segment Prefix (PSP) and an environment segment.[6] This might cause the driver not to be loaded into the most suitable free memory area or even prevent it from being loaded high at all. In contrast to this, a self-relocating driver can be loaded anywhere (including into conventional memory) and then relocate only its (typically much smaller) resident portion into a suitable free memory area in upper memory. In addition, advanced self-relocating TSRs (even if already loaded into upper memory by the operating system) can relocate over most of their own PSP segment and command line buffer and free their environment segment in order to further reduce the resulting memory footprint and avoid fragmentation. Some self-relocating TSRs can also dynamically change their "nature" and morph into device drivers even if originally loaded as TSRs, thereby typically also freeing some memory.[4] Finally, it is technically impossible for an external loader to relocate drivers into expanded memory (EMS), the high memory area (HMA) or extended memory (via DPMS or CLOAKING), because these methods require small driver-specific stubs to remain in conventional or upper memory in order to coordinate the access to the relocation target area,[7][nb 1][nb 2] and in the case of device drivers also because the driver's header must always remain in the first megabyte.[7][6] In order to achieve this, the drivers must be specially designed to support self-relocation into these areas.[7]

Some advanced DOS drivers also contain both a device driver (which would be loaded at offset +0000h by the operating system) and TSR (loaded at offset +0100h) sharing a common code portion internally as fat binary.[6] If the shared code is not designed to be position-independent, it requires some form of internal address fix-up similar to what would otherwise have been carried out by a relocating loader already; this is similar to the fix-up stage of self-relocation but with the code already being loaded at the target location by the operating system's loader (instead of done by the driver itself).

LOADERS

A loader is a major component of an operating system that ensures all necessary programs and libraries are loaded, which is essential during the startup phase of running a program. It places the libraries and programs into the main memory in order to prepare them for execution. Loading involves reading the contents of the executable file that contains the instructions of the program and then doing other preparatory tasks that are required in order to prepare the executable for running, all of which takes anywhere from a few seconds to minutes depending on the size of the program that needs to run.

Techopedia Explains Loader

The loader is a component of an operating system that carries out the task of preparing a program or application for execution by the OS. It does this by reading the contents of the executable file and then storing these instructions into the RAM, as well as any library elements that are required to be in memory for the program to execute. This is the reason a splash screen appears right before most programs start, often showing what is happening in the background, which is what the loader is currently loading into the memory. When all of that is done, the program is ready to execute. For small programs, this process is almost instantaneous, but for large and complex applications with large libraries required for execution, such as games as well as 3D and CAD software, this could take longer. The loading speed is also dependent on the speed of the CPU and RAM.

Not all code and libraries are loaded at program startup, only the ones required for actually running the program. Other libraries are loaded as the program runs, or only as required. This is especially true for applications such as games that only need assets loaded for the current level or location that the player is in.

Though loaders in different operating systems might have their own nuances and specialized functions native to that particular operating system, they still serve basically the same function. The following are the responsibilities of a loader:

1. Validate the program for memory requirements, permissions, etc.
2. Copy necessary files, such as the program image or required libraries, from the disk into the memory
3. Copy required command-line arguments into the stack
4. Link the starting point of the program and link any other required library
5. Initialize the registers
6. Jump to the program starting point in memory

SOFTWARE TOOLS FOR PROGRAM DEVELOPMENT

A programming tool or software development tool is a program or application that software developers use to create, debug, maintain, or otherwise support other programs and applications. The term usually refers to relatively simple programs that can be combined together to accomplish a task, much as one might use multiple hand tools to fix a physical object.

The history of software tools began with the first computers in the early 1950s that used linkers, loaders, and control programs. Tools became famous with Unix in the early 1970s with tools like grep, awk and make that were meant to be combined flexibly with pipes. The term "software tools" came from the book of the same name by Brian Kernighan and P. J. Plauger.

Tools were originally simple and light weight. As some tools have been maintained, they have been integrated into more powerful integrated development environments (IDEs). These environments consolidate functionality into one place, sometimes increasing simplicity and productivity, other times sacrificing flexibility and extensibility. The workflow of IDEs is routinely contrasted with alternative approaches, such as the use of Unix shell tools with text editors like Vim and Emacs

Software development tools can be roughly divided into the following categories:

1) Performance analysis tools

2) Debugging tools

3) Static analysis and formal verification tools

4) Correctness checking tools

5) Memory usage tools

6) Application build tools

7) Integrated development environment

Software Development Tools also called Programming Tools, Integrated Development Tools, Software Development Kits, Software Developer's Kits, Design Tools, Application Development Software, Application Deployment Tools, Application Development Tools, SDK, Development Tools, Tools, Software Engineering Tools, Applications Frameworks, Development Kits, Program Development Tools, IT Tools, Frameworks, Software Tools, and Information Technology Tools .

A software developer's kit (SDK) is a set of programs used by a computer programmer to write application programs. Typically, an SDK includes a visual screen builder, an editor, a compiler, a linker, and sometimes other facilities. The term is used by Microsoft, Sun Microsystems, and a number of other companies.This term is sometimes seen as software development kit.

Debug monitors give debugging support for a program. A debug monitor executes the program being debugged in its own control thereby giving execution efficiency throughout debugging. There are debug monitors which are language independent and can handle programs written in several languages. For illustration-DEC-10.

Debug monitor give the following facilities for dynamic debugging:

1. Setting breakpoints into the program

2. Initiating a debug conversation while control reaches a breakpoint.

3. Displaying variable's values

4. Assigning new values to variables.

5. Testing in defined assertions and predicates including program variables.

Programming Environments

The term *programming environment* is sometimes reserved for environments containing language specific editors and source level debugging facilities; here, the term will be used in its broader sense to refer to all of the hardware and software in the environment used by the programmer. All programming can therefore be properly described as takin place in a programming environment.

Programming environments may vary considerably in complexity. An example of a simple environment might consist of a text editor for program preparation, an assembler for translating programs to machine language, and a simple operating system consisting of input-output drivers and a file system. Although card input and non-interactive operation characterized most early computer systems, such simple environments were supported on early experimental time-sharing systems by 1963.

Although such simple programming environments are a great improvement over the bare hardware, tremendous improvements are possible. The first improvement which comes to mind is the use of a high level language instead of an assembly language, but this implies other changes. Most high level languages require more complicated run-time support than just input-output drivers and a file system. For example, most require an extensive library of predefined procedures and functions, many require some kind of automatic storage management, and some require support for concurrent execution of threads, tasks or processes within the program.

Many applications require additional features, such as window managers or elaborate file access methods. When multiple applications coexist, perhaps written by different programmers, there is frequently a need to share files, windows or memory segments between applications. This is typical of today's electronic mail, database, and spreadsheet applicatons, and the programming environments that support such applications can be extremely complex, particularly if they attempt to protect users from malicious or accidental damage caused by program developers or other users.

A programming environment may include a number of additional features which simplify the programmer's job. For example, library management facilities to allow programmers to extend the set of predefined procedures and functions with their own routines. Source level debugging facilities, when available, allow run-time errors to be interpreted in terms of the source program instead of the machine language actually run by the hardware. As a final example, the text editor may be language specific, with commands which operate in terms of the syntax of the language being used, and mechanisms which allow syntax errors to be detected without leaving the editor to compile the program.

A Unifying Framework

In all programming environments, from the most rudimentary to the most advanced, it is possible to identify two distinct components, the program preparation component and the program execution component. On a bare machine, the program preparation component consists of the switches or push buttons by which programs and data may be entered into the memory of the machine; more advanced systems supplement this with text editors, compilers, assemblers, object library managers, linkers, and loaders. On a bare machine, the program execution component consists of the hardware of the machine, the central processors, any peripheral processors, and the various memory resources; more advanced systems supplement this with operating system services, libraries of predefined procedures, functions and objects, and interpreters of various kinds.

Within the program execution component of a programming environment, it is possible to distinguish between those facilities needed to support a single user process, and those which are introduced when resources are shared between processes. Among the facilities which may be used to support a single process environment are command language interpreters, input-output, file systems, storage allocation, and virtual memory. In a multiple process environment, processor allocation, interprocess communication, and resource protection may be needed. Figure 1.1 lists and classifies these components.

```
Editors
Compilers
Assemblers                    Program Preparation
Linkers
Loaders
=======================================================
Command Languages
Sequential Input/Output
Random Access Input/Output
File Systems               Used by a Single Process
Window Managers
Storage Allocation
Virtual Memory
----------------------------- Program Execution Support
Process Scheduling
```

Interprocess Communication

Resource Sharing Used by Multiple Processes

Protection Mechanisms

Figure 1.1. Components of a programming environment.

This text is divided into three basic parts based on the distinctions illustrated in Figure 1.1. The distinction between preparation and execution is the basis of the division between the first and second parts, while the distinction between single process and multiple process systems is the basis of the division between the second and third parts.

USER INTERFACE

User interface is the front-end application view to which user interacts in order to use the software. User can manipulate and control the software as well as hardware by means of user interface. Today, user interface is found at almost every place where digital technology exists, right from computers, mobile phones, cars, music players, airplanes, ships etc.

User interface is part of software and is designed such a way that it is expected to provide the user insight of the software. UI provides fundamental platform for human-computer interaction.

UI can be graphical, text-based, audio-video based, depending upon the underlying hardware and software combination. UI can be hardware or software or a combination of both.

The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interfacing screens

UI is broadly divided into two categories:

- Command Line Interface
- Graphical User Interface

Command Line Interface (CLI)

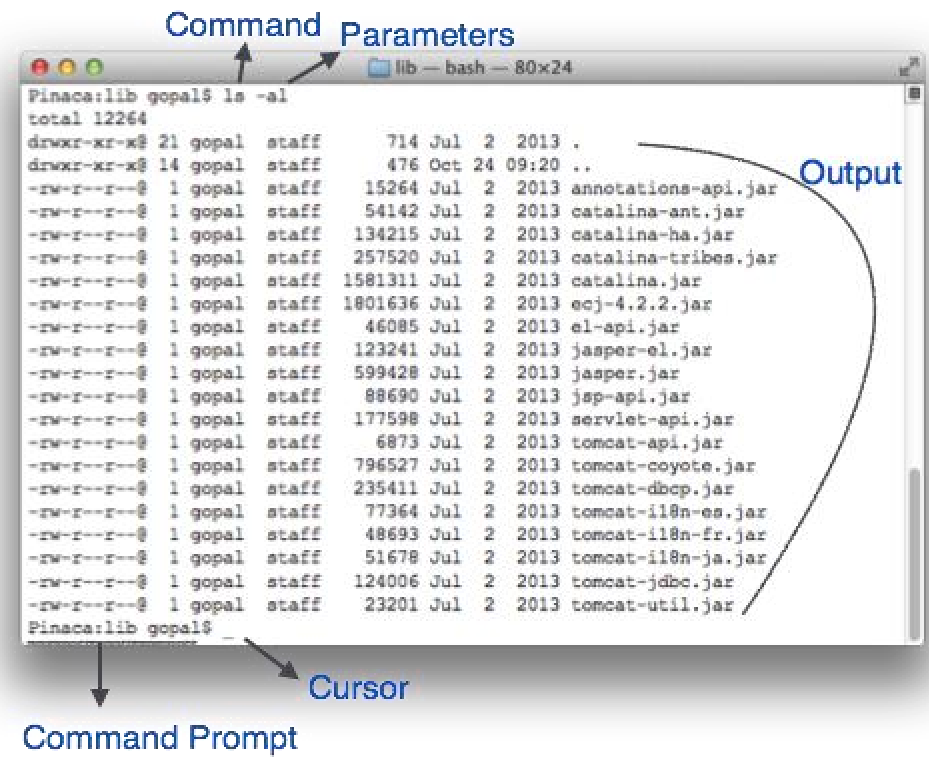
CLI has been a great tool of interaction with computers until the video display monitors came into existence. CLI is first choice of many technical users and programmers. CLI is minimum interface a software can provide to its users.

CLI provides a command prompt, the place where the user types the command and feeds to the system. The user needs to remember the syntax of command and its use. Earlier CLI were not programmed to handle the user errors effectively.

A command is a text-based reference to set of instructions, which are expected to be executed by the system. There are methods like macros, scripts that make it easy for the user to operate.

CLI uses less amount of computer resource as compared to GUI.

CLI Elements



```
Pinaca:lib gopal$ ls -al
total 12264
drwxr-xr-x@ 21 gopal  staff      714 Jul  2  2013 .
drwxr-xr-x@ 14 gopal  staff      476 Oct 24 09:20 ..
-rw-r--r--@  1 gopal  staff    15264 Jul  2  2013 annotations-api.jar
-rw-r--r--@  1 gopal  staff    54142 Jul  2  2013 catalina-ant.jar
-rw-r--r--@  1 gopal  staff   134215 Jul  2  2013 catalina-ha.jar
-rw-r--r--@  1 gopal  staff   257520 Jul  2  2013 catalina-tribes.jar
-rw-r--r--@  1 gopal  staff  1581311 Jul  2  2013 catalina.jar
-rw-r--r--@  1 gopal  staff  1801636 Jul  2  2013 ecj-4.2.2.jar
-rw-r--r--@  1 gopal  staff    46085 Jul  2  2013 el-api.jar
-rw-r--r--@  1 gopal  staff   123241 Jul  2  2013 jasper-el.jar
-rw-r--r--@  1 gopal  staff   599428 Jul  2  2013 jasper.jar
-rw-r--r--@  1 gopal  staff    88690 Jul  2  2013 jsp-api.jar
-rw-r--r--@  1 gopal  staff   177598 Jul  2  2013 servlet-api.jar
-rw-r--r--@  1 gopal  staff     6873 Jul  2  2013 tomcat-api.jar
-rw-r--r--@  1 gopal  staff   796527 Jul  2  2013 tomcat-coyote.jar
-rw-r--r--@  1 gopal  staff   235411 Jul  2  2013 tomcat-dbcp.jar
-rw-r--r--@  1 gopal  staff    77364 Jul  2  2013 tomcat-i18n-es.jar
-rw-r--r--@  1 gopal  staff    48693 Jul  2  2013 tomcat-i18n-fr.jar
-rw-r--r--@  1 gopal  staff    51678 Jul  2  2013 tomcat-i18n-ja.jar
-rw-r--r--@  1 gopal  staff   124006 Jul  2  2013 tomcat-jdbc.jar
-rw-r--r--@  1 gopal  staff    23201 Jul  2  2013 tomcat-util.jar
Pinaca:lib gopal$ _
```

A text-based command line interface can have the following elements:

- **Command Prompt** - It is text-based notifier that is mostly shows the context in which the user is working. It is generated by the software system.
- **Cursor** - It is a small horizontal line or a vertical bar of the height of line, to represent position of character while typing. Cursor is mostly found in blinking state. It moves as the user writes or deletes something.
- **Command** - A command is an executable instruction. It may have one or more parameters. Output on command execution is shown inline on the screen. When output is produced, command prompt is displayed on the next line.

Graphical User Interface

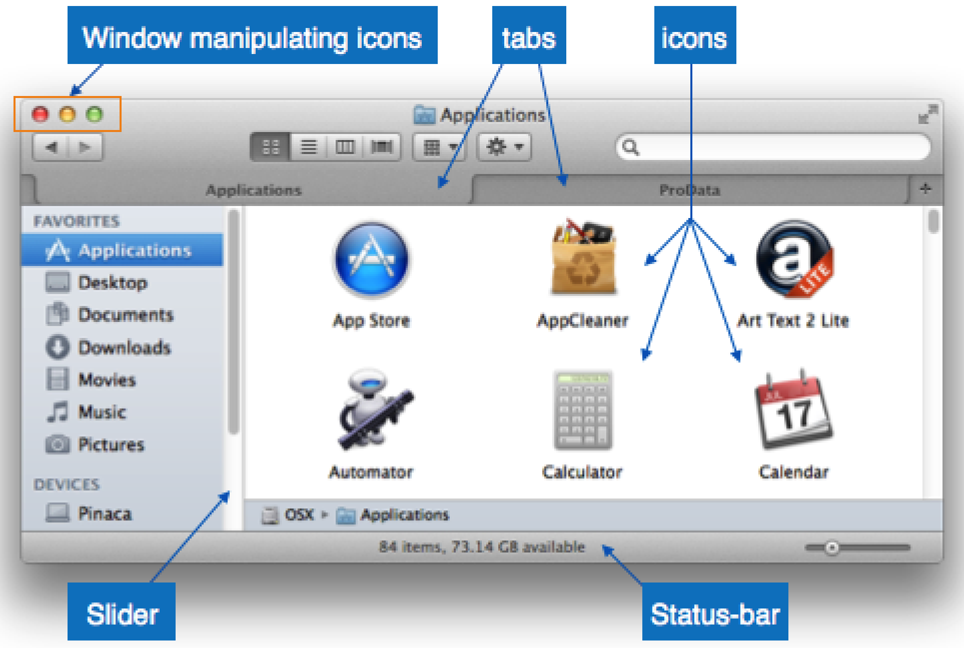
Graphical User Interface provides the user graphical means to interact with the system. GUI can be combination of both hardware and software. Using GUI, user interprets the software.

Typically, GUI is more resource consuming than that of CLI. With advancing technology, the programmers and designers create complex GUI designs that work with more efficiency, accuracy and speed.

GUI Elements

GUI provides a set of components to interact with software or hardware.

Every graphical component provides a way to work with the system. A GUI system has following elements such as:



- **Window** - An area where contents of application are displayed. Contents in a window can be displayed in the form of icons or lists, if the window represents file structure. It is easier for a user to navigate in the file system in an exploring window. Windows can be minimized, resized or maximized to the size of screen. They can be moved anywhere on the screen. A window may contain another window of the same application, called child window.

- **Tabs** - If an application allows executing multiple instances of itself, they appear on the screen as separate windows. **Tabbed Document Interface** has come up to open multiple documents in the same window. This interface also helps in viewing preference panel in application. All modern web-browsers use this feature.

- **Menu** - Menu is an array of standard commands, grouped together and placed at a visible place (usually top) inside the application window. The menu can be programmed to appear or hide on mouse clicks.

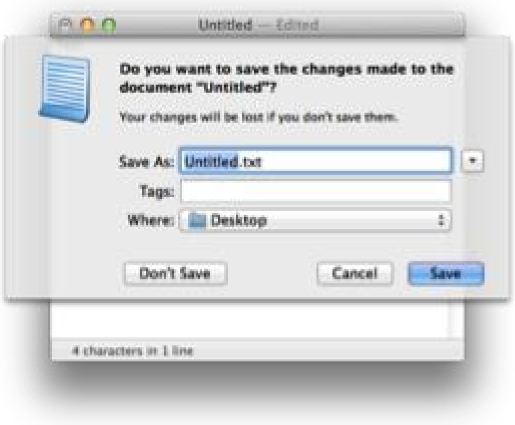
- **Icon** - An icon is small picture representing an associated application. When these icons are clicked or double clicked, the application window is opened. Icon displays application and programs installed on a system in the form of small pictures.

- **Cursor** - Interacting devices such as mouse, touch pad, digital pen are represented in GUI as cursors. On screen cursor follows the instructions from hardware in almost real-time. Cursors are also named pointers in GUI systems. They are used to select menus, windows and other application features.

Application specific GUI components

A GUI of an application contains one or more of the listed GUI elements:

- **Application Window** - Most application windows uses the constructs supplied by operating systems but many use their own customer created windows to contain the contents of application.
- **Dialogue Box** - It is a child window that contains message for the user and request for some action to be taken. For Example: Application generate a dialogue to get confirmation from user to delete a file.



Do you want to save the changes made to the document "Untitled"?

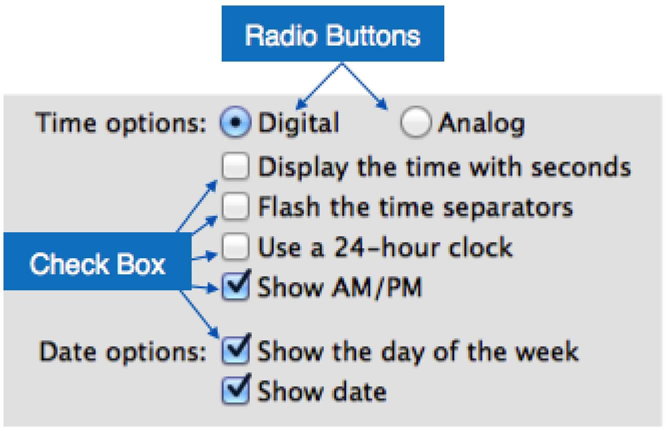
Your changes will be lost if you don't save them.

Save As: Untitled.txt

Tags:

Where: Desktop

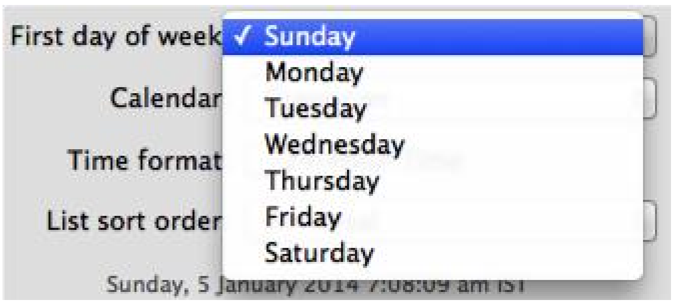
- **Text-Box** - Provides an area for user to type and enter text-based data.
- **Buttons** - They imitate real life buttons and are used to submit inputs to the software.



Time options: [● Digital] ○ Analog
☐ Display the time with seconds
☐ Flash the time separators
☐ Use a 24-hour clock
☑ Show AM/PM

Date options: ☑ Show the day of the week
☑ Show date

- **Radio-button** - Displays available options for selection. Only one can be selected among all offered.
- **Check-box** - Functions similar to list-box. When an option is selected, the box is marked as checked. Multiple options represented by check boxes can be selected.
- **List-box** - Provides list of available items for selection. More than one item can be selected.



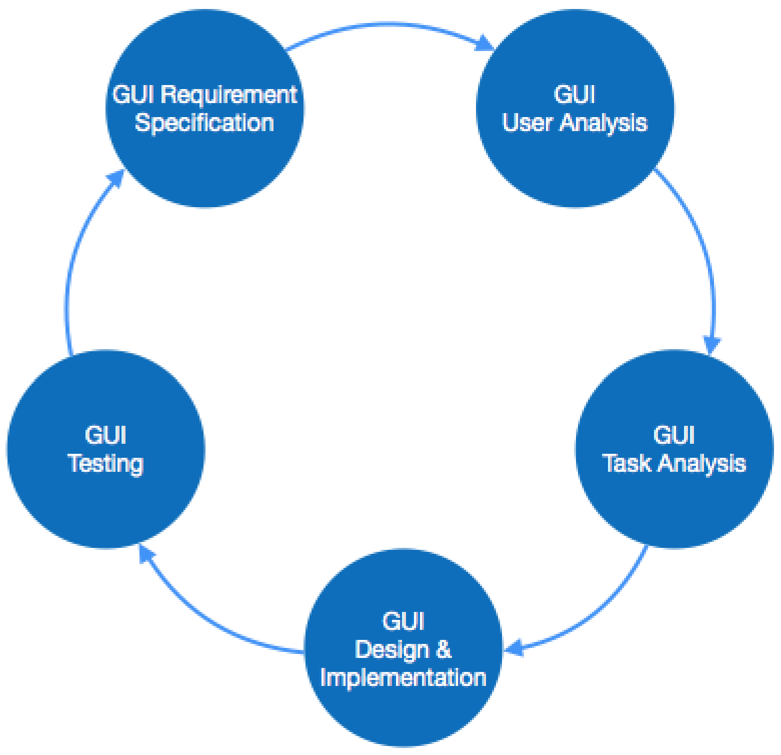
Other impressive GUI components are:

- Sliders
- Combo-box
- Data-grid
- Drop-down list

User Interface Design Activities

There are a number of activities performed for designing user interface. The process of GUI design and implementation is alike SDLC. Any model can be used for GUI implementation among Waterfall, Iterative or Spiral Model.

A model used for GUI design and development should fulfill these GUI specific steps.



- **GUI Requirement Gathering** - The designers may like to have list of all functional and non-functional requirements of GUI. This can be taken from user and their existing software solution.
- **User Analysis** - The designer studies who is going to use the software GUI. The target audience matters as the design details change according to the knowledge and competency level of the user. If user is technical savvy, advanced and complex GUI can be incorporated. For a novice user, more information is included on how-to of software.
- **Task Analysis** - Designers have to analyze what task is to be done by the software solution. Here in GUI, it does not matter how it will be done. Tasks can be represented in hierarchical manner taking one major task and dividing it further into smaller sub-tasks. Tasks provide goals for GUI presentation. Flow of information among sub-tasks determines the flow of GUI contents in the software.
- **GUI Design & implementation** - Designers after having information about requirements, tasks and user environment, design the GUI and implements into code and embed the GUI with working or dummy software in the background. It is then self-tested by the developers.

- **Testing** - GUI testing can be done in various ways. Organization can have in-house inspection, direct involvement of users and release of beta version are few of them. Testing may include usability, compatibility, user acceptance etc.

GUI Implementation Tools

There are several tools available using which the designers can create entire GUI on a mouse click. Some tools can be embedded into the software environment (IDE).

GUI implementation tools provide powerful array of GUI controls. For software customization, designers can change the code accordingly.

There are different segments of GUI tools according to their different use and platform.

Example

Mobile GUI, Computer GUI, Touch-Screen GUI etc. Here is a list of few tools which come handy to build GUI:

- FLUID
- AppInventor (Android)
- LucidChart
- Wavemaker
- Visual Studio

User Interface Golden rules

The following rules are mentioned to be the golden rules for GUI design, described by Shneiderman and Plaisant in their book (Designing the User Interface).

- **Strive for consistency** - Consistent sequences of actions should be required in similar situations. Identical terminology should be used in prompts, menus, and help screens. Consistent commands should be employed throughout.
- **Enable frequent users to use short-cuts** - The user's desire to reduce the number of interactions increases with the frequency of use. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.

- **Offer informative feedback** - For every operator action, there should be some system feedback. For frequent and minor actions, the response must be modest, while for infrequent and major actions, the response must be more substantial.
- **Design dialog to yield closure** - Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and this indicates that the way ahead is clear to prepare for the next group of actions.
- **Offer simple error handling** - As much as possible, design the system so the user will not make a serious error. If an error is made, the system should be able to detect it and offer simple, comprehensible mechanisms for handling the error.
- **Permit easy reversal of actions** - This feature relieves anxiety, since the user knows that errors can be undone. Easy reversal of actions encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.
- **Support internal locus of control** - Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.
- **Reduce short-term memory load** - The limitation of human information processing in short-term memory requires the displays to be kept simple, multiple page displays be consolidated, window-motion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.

*****