DEBRE TABOR UNIVERSITY - DTU

Faculty of Technology

Department of Computer Science

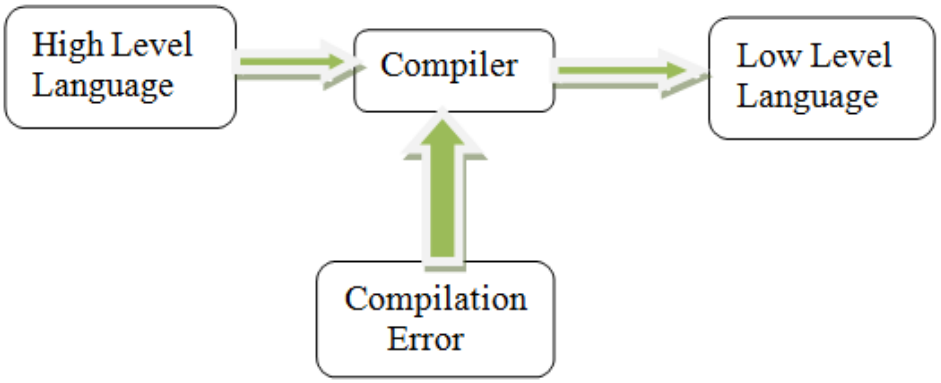
Course Module for Compiler Design (CoSc4103)

Debre Tabor,
Ethiopia
2014 E.C.

Chapter One: Introduction to Compiling

1. Introduction

Compiler is a computer program or software that used to translate high-level programming language (HLPL) or source program to Computer understandable language or machine language. A compiler takes a program written in source language as its input and produces an equivalent program called object code. Usually, the source program is written in a high-level language, such as C, C++, Java, C#, PHP, Perl, Android, etc and the target program is object code (**Machine code**) for the target machine (operating system and processor type).



Types of Compilers

i. **Cross-Compiler:** that runs on a machine 'A' and produces a code for another machine 'B'. It is capable of creating code for a platform other than the one which the compiler running on. The output of a cross compiler is **designed** to run on a different platform. It is a program that translates into an object code format that is not supported on the **compilation** machine and is commonly used to prepare code for **embedded** applications (Systems).

ii. **Source to Source Compiler (Transpiler or Transcompiler):** is a compiler that translates the source program (code) written in one programming language into the source code for another programming language.

iii. **Native Compiler (Host Compiler):** a compiler that translates source program to object code on the same platform and in which output is intended to directly run on the same type of computer and operating system.

Brainstorming, Programs Related to Compilers

- *Briefly Explain the following Programs?*

Interpreters, Assemblers, Linkers, Loaders, Pre-processors, Editors, Debuggers, Profilers, and Project Managers.

- Give examples of compiler for the above types?

1.1. Phases of a Compiler

A compiler operates in phases. There are two major phases of compilation, which in turn have many parts. Each of them is taking input from the output of the previous level and work in a coordinated way. A phase is a compiler module (subprograms) that are a logically interrelated operation that takes source program in one representation and produces output in another representation.

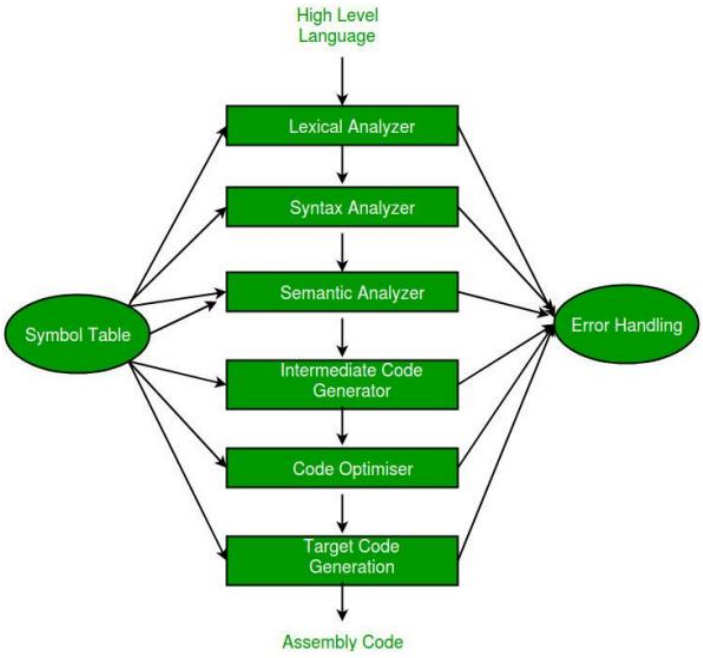
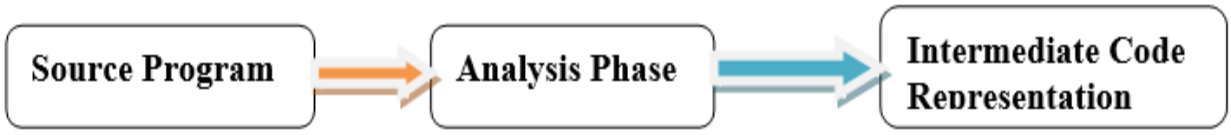


fig.1.1. The phases of a compiler

The two phases of compilation.

- **Analysis Phase**: An Intermediate representation is created from the source program. This phase is Machine Independent and Language-Dependent. This phase has three parts. It is also termed as the **front end** of the compiler.
  - Lexical Analysis,
  - Syntactical (Syntax) Analysis,
  - Semantic Analysis.

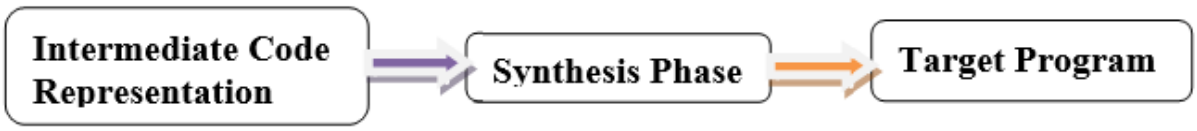


Lexical analyzer divides the program into "tokens", Syntax analyze recognizes "sentences" in the program using the syntax of the language and semantic analyzer checks static semantics of each construct.

- **Synthesis Phase**: Equivalent target program is created from the intermediate representation. Synthesis part takes the **intermediate** representation as input and transforms it into the **target** program. This phase is Machine Dependent and Language independent. It has three parts. It is also termed as the **back end** of the compiler.

- Intermediate Code Generation,
- Code Optimization.
- Code Generation

Intermediate Code Generation generates "abstract" code, Code Optimizer optimizes the abstract code, and final Code generation translates abstract intermediate code into specific machine instructions.



i. **Lexical Analysis (Scanning):** In this phase source program will get a scan at once and a finite set of tokens can be generated. Programs responsible for this task are called lexical Analyzer(**lexer**) or **Scanner** or **Tokenizer**. The process is called Tokenization. Lexical Analyzer or Scanners reads the source program one character at a time, carving the source program into a sequence of atomic units called **tokens**. **Tokens** are independent program elements which are meaningful by themselves. Token in Lexical Analysis can be represented as two attributes: Lexical analysis is the first phase of the compiler which is also termed as **scanning**. The source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

• **Token:** Token is a sequence of characters that represent a lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc.

• **Lexeme:** Lexeme is an instance of a token i.e., group of characters forming a token.

• **Pattern:** Pattern describes the rule that the lexemes of a token take. It is the structure that must be matched by strings. Once a token is generated the corresponding entry is made in the symbol table.

Input: a stream of characters

Output: Token

*Token Template: <**token-name, attribute-value**>*

If the symbols are given in the standard format the LA accepts and produces token as output. Each token is a sub-string of the program that is to be treated as a single and meaningful unit.

Token are two types.

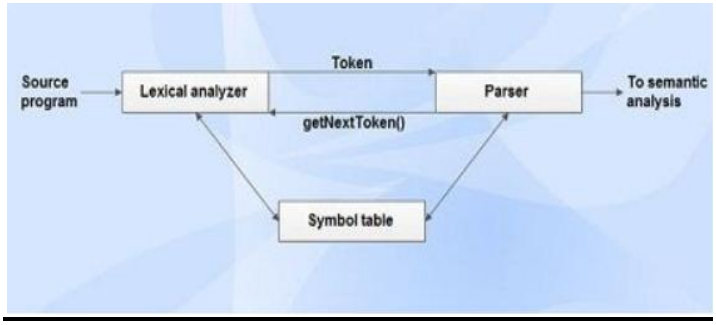
1. Specific strings such as if (or) semicolon.
2. Classes of string such as identifiers, label, constants,

For example, for the expression, c= a + b*5; after lexical analysis, the output will be:

Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

Role of the lexical analyzer



Lexical analyzer performs the following tasks (Summary):

- Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
- Enters the identified token into the symbol table.
- Strips out white spaces and comments from the source program.
- Correlates error messages with the source program i.e., displays an error message with its occurrence by specifying the line number.
- Expands the macros if it is found in the source program.

Tasks of the lexical analyzer can be divided into two processes:

Scanning: Performs reading of input characters, removal of white spaces and comments.

Lexical Analysis: Produce tokens as the output.

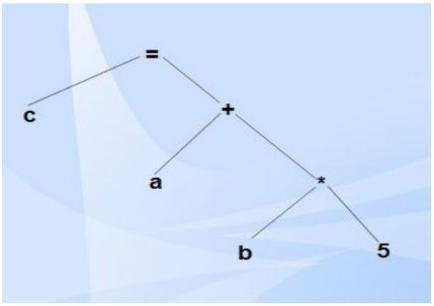
ii.Syntax Analysis (Parsing): The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of the lexical analysis. Syntax analysis is aided by using techniques based on the formal grammar of the programming language. Syntax analysis is the second phase of the compiler which is also called as parsing. It checks if the program to be compiled is syntactically correct. The target program called Syntax Analyzer or **Parser** converts the tokens produced by the lexical analyzer into a tree-like representation called parse tree. A parse tree describes the syntactic structure of the input or source program. The syntax tree is a compressed representation of the parse tree in

which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.

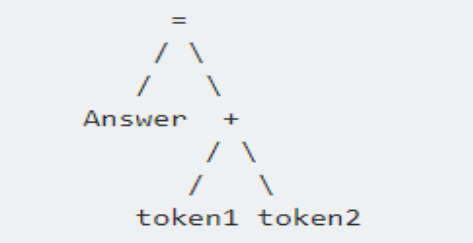
Input: Tokens

Output: Syntax tree or parse tree

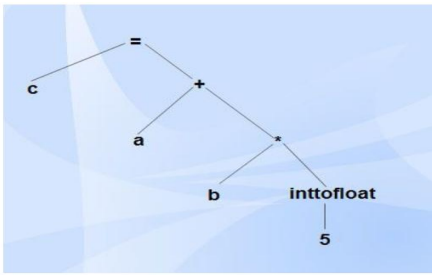
For example: for the expression c=a+b*5; the corresponding syntax tree is:



Consider following example, Answer=token1+token2;



iii. Semantic Analysis (Semantic Analyzer): Semantic analysis is the third phase of the compiler. It determines its **runtime behavior**. It checks for the semantic consistency (meaning of the statements in the program). **Semantics** help interprets symbols, their types, and their relations with each other. **Semantic analysis** judges whether the syntax structure constructed in the source program derives any meaning or not. Type information is gathered and stored in symbol table or in the syntax tree. Performs type checking. It takes syntax tree as input and produces annotated or attributed syntax tree as output.



iv. Intermediate Code Generations: An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation. Intermediate code generation produces intermediate representations for the source program which are of the following forms:

- **Postfix Notation**

- **Three Address Code (**TAC or 3AC): is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations.

Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example, t1: = t2 + t3.

- **Syntax Tree**

Most commonly used form is the three-address code.

Calculate one solution to the [[quadratic equation]].

x = (-b + sqrt (b^2 - 4*a*c)) / (2*a)

Three address code representations:

t1: = b * b	t4: = t1 - t3	t8: = 2 * a
t2: = 4 * a	t5: = sqrt(t4)	t9: = t7 / t8
t3: = t2 * c	t6: = 0 - b	x: = t9
	t7: = t5 + t6	

Properties of intermediate code

• It should be easy to produce.

• It should be easy to translate into the target program.

v.Code Optimization: This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Optimization is a program transformation technique, which tries to improve the code by making it consume less resource (i.e., CPU, memory) and deliver high speed. Code optimization phase gets the intermediate code as **input** and produces optimized intermediate code as **output**. This phase reduces the redundant code and attempts to improve the intermediate code so that the output or machine code will run faster, takes less processor unit and takes **less** space. During the code optimization, the **result** of the program or meaning is not affected.

• To improve the code generation, the optimization involves

- o Deduction and removal of dead code (unreachable code).
- o Calculation of constants in expressions and terms.
- o Collapsing of repeated expression into a temporary string.
- o **Loop unrolling?**
- **o Moving code outside the loop.**
- o Removal of unwanted temporary variables.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes.

Loop Optimization: Another important source of optimization concerns about increasing the speed of loops. A typical loop improvement is to move a computation that produces the same result each time around the loop to a point, in the program just before the loop is entered.

For Example:

```
do {
item=10;
value=value + item;
} while(value<100);
```

This code involves the repeated assignment of the identifier item, which if we put this way:

```
item=10;
do {
value=value + item;
} while(value<100);
```

Local Optimization: There are local transformations that can be applied to a program to make an improvement. For example,

If A > B goto L2

Goto L3

L2:

This can be replaced by a single statement If A < B goto L3

Another important local optimization is the elimination of common sub-expressions

A: = B + C + D

E: = B + C + F

Might be evaluated as

T1: = B + C

A: = T1 + D

E: = T1 + F

Take this advantage of the common sub-expressions B + C.

Question: Find any sample code try to optimize it accordingly?

vi. **Code Generation:** The last phase of translation is code generation. A number of optimizations to reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer. Code generation is the final phase of a compiler, readily executed by a machine. It gets input from code optimization phase and produces the target code or **object** code as result. Intermediate instructions are translated into a sequence of machine instructions that perform the same task.

• The code generation involves

o Allocation of register and memory.

o Generation of correct references.

o Generation of correct data types.

o Generation of missing code.

LDF R_2, id_3 **ADDF** R_1, R_2

MULF R_2, # 5.0 **STF** id_1, R_1

LDF R_1, id_2

Note: The above instructions are 8086 assembly language instructions which can be run on a 8086 assembler. For example, every three-address statement of the form x: = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

- MOV y, R0 /* load y into register R0 */
- ADD z, R0 /* add z to R0 */
- MOV R0, x /* store R0 into x */
- a := b + c
- d := a + e

Would be translated into:

- MOV b, R0
- ADD c, R0
- MOV R0, a
- MOV a, R0
- ADD e, R0
- MOV R0, d

All of the aforementioned phases involve the following tasks:

Symbol Table Manager: This is the portion compiler phases to keep the track of the **names** used by the program and records essential information about each name. It is a simple data structure used to store/record this information called a 'Symbol Table'. Information about the source program is collected and stored in a data structure called a symbol table. contains/stores/ necessary information of the program; **identifiers, functions, data types, literals** (numeric constants 3.14 and quoted strings "hi"). The symbol table is used to store all the information about identifiers used in the program. It allows finding the record for each identifier quickly and to store or retrieve data from that record. Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Example 1:

int a, b; float c;

char z;

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Example 2:

```
extern double test (double x);
double sample (int count)
{
double sum= 0.0;
for (int i = 1; i < = count; i++)
sum+= test((double) i);
return sum;}
```

Symbol name	Type	Scope
test	function, double	extern
x	double	function parameter
sample	function, double	global
count	int	function parameter
sum	double	block local
i	int	for-loop statement

vii. Error Handlers: It is invoked when a flaw error in the source program is detected. The output of Lexical Analysis is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The Syntax Analyzer groups the tokens together into a syntactic structure called as an expression. The expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

The parser has two functions. It checks if the tokens from the lexical analyzer, occur in a pattern that is permitted by the **specification** for the source language. It also imposes on tokens a tree-like structure that is used by the subsequent phases of the compiler.

Example: if a program contains the expression A+/B after lexical analysis this expression might appear to the syntax analyzer as the token sequence id+/id. On seeing the /, the syntax analyzer should detect an error situation, because the presence of these two adjacent binary operators violates the formulations rule of an expression.

Syntax analysis is to make explicit the hierarchical structure of the incoming token stream by identifying which parts of the token stream should be grouped.

Example, (A/B*C has two possible interpretations.)

1. Divide A by B and then multiply by C or
2. Multiply B by C and then use the result to divide A.

Each of these two interpretations can be represented in terms of a parse tree.

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler. Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Expecting Errors at each phase of the compiler are:

• A compiler is expected to help the programmer in locating and tracking down errors!

Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

• In lexical analysis, errors occur in the separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In the semantic analysis, errors may occur in the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when the code is missing etc.

The figure illustrates the translation of source code through each phase, considering the statement

c =*a*+ *b* * 5.

Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must somehow deal with the error, so that compilation can proceed.

Lexical Errors: A character sequence that cannot be scanned into any valid token is a lexical error. Lexical errors are uncommon, but they still must be handled by a scanner, It includes incorrect or Misspelling of identifiers, keyword, or operators typed incorrectly are considered as lexical errors. Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning **of a token.** Misspellings of ids, keywords and missing quotes around text intended as a string.

Syntax Errors: It includes missing semicolon or unbalanced parenthesis. Syntactic errors are handled by syntax analyzer (parser).

When an error is detected, it must be handled by the parser to enable the parsing of the rest of the input. In general, errors may be expected at various stages of compilation but most of the errors

are syntactic errors and hence the parser should be able to detect and report those errors in the program.

The goals of error handler in parser are:

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

Semantically Errors: These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are Type mismatch, Undeclared variable, Reserved identifier misuse, Multiple declarations of a variable in a scope, Accessing an out-of-scope variable, Actual and formal parameter mismatch, Multiple declaration of a variable in a scope, Type mismatches between operator and operands.

Logical errors

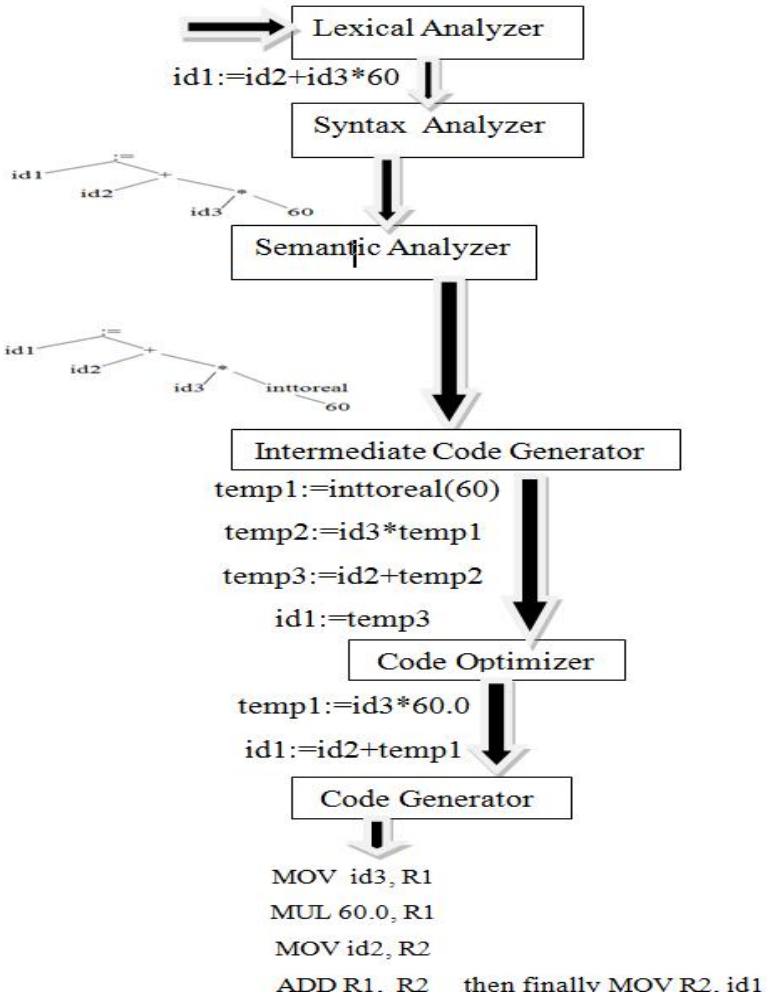
- These errors occur due to not reachable code-infinite loop, Incorrect reasoning, = instead of ==, Program may be well-formed but not what the programmer wants.

For example:

- int a = “value”;
- should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.
- The following tasks should be performed in semantic analysis:
- Scope resolution, Type checking, Array-bound checking.

Summary on Phases of Compiler, show what will expect at each phase of for the expression below: position: =initial + rate*60?

position: = initial + rate*60



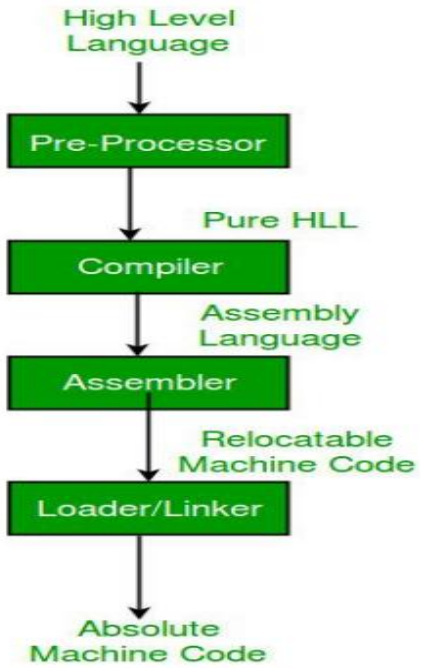
Corresponding Symbol Table

1	Position	....
2	Initial	....
3	Rate	.....

1.2. Computer Language Representation (Processing Systems)

A high-level language (HLL) is a programming language such as C, FORTRAN, or Pascal that enables a programmer to write programs that are more or less independent of a particular type of computer. Such languages are considered high-level because they are closer to human languages (a human can understand better) and further from machine languages. In contrast, assembly languages are considered low level because they are very close to machine languages (more understood by machine).

We know a computer is a logical assembly of software and hardware. The hardware knows a language, that is hard for us to grasp, consequently, we tend to write programs in high-level language, that is much less complicated for us to comprehend and maintain in thoughts. now, these programs go through a serious of transformation so that they can readily be used machines. This is where language procedure systems come in handy.



- **High-Level Language:** if a program contains #define or #include directives such as # include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called pre-processor directives. They direct the pre-processor about what to do.
- **Pre-Processor:** The pre-processor removes all the #include directives by including the files called files inclusion and all the #define directives using macro expansion. it performs file inclusion, augmentation, macro-processing etc.
- **Assembly Language:** It's neither in binary form nor high level. it is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- **Assembler**: For every platform (hardware + OS) we will have an assembler. they are not universal since for each platform we have one. The output of the assembler is called object file. It translates assembly language to machine code.
- **Interpreter:** An interpreter converts high-level language into low-level machine language. Compiler and Interpreter different from each other by in a way they read input. The compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line. compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a line. Interpreted programs are usually slower with respect to compiled ones.
- **Relocatable Machine Code**: It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate for the program movement.
- **Loader/Linker:** It converts the relocatable code into the absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). linker load a variety of object files into a single file to make it executable. The loader loads it in memory and executes it.

▪ **Compiler Design: Linker:** linker is a program in a system which helps to link object modules of a program into a single object file. It performs the process of linking. The linker is also called link editors. Linking is the process of collecting and maintaining a piece of code and data into a single file. Linker also link a particular module into system library. It takes object module from assembler as input and from the executable file as output for the loader. Linking is performed at both compile time when the source code is translated to object code and load time when the program loaded into main memory by the loader. liking is performed at the last step in compiling a program.

Source Code ⇨ Pre-processor ⇨ Compiler ⇨ Assembler Linker ⇨ Loader

Question

What are the functions and procedure of linker, loader, preprocessor and compiler?

• In computer systems, a **loader** is the part of an operating system that is responsible for loading programs and libraries. It is one of the essential stages in the process of starting a program, as it places programs into memory and prepares them for execution.

• A linker combines one or more object files and possible some library code into either some executable, some library or a list of error messages.

• How **the Linker Works**. The compiler compiles a single high-level language file (C language, for example) into a single object module file. The **linker** (Id) can only work with object modules to **link** them together. Object modules are the smallest unit that the **linker works** with.

• After linking, you obtain the actual executable that can run. A **compiler** generates object code files (machine language) from source code. A **linker** combines these object code files into an executable. ... Some languages/**compilers** do not have a distinct **linker** and linking is done by the **compiler** as part of its work.

• **The linker** is a program that takes one or more objects generated by a **compiler** and combines them into a single executable program. • The loader is the part of an operating **system** that is responsible for loading programs from executables (i.e., executable files) into memory, preparing them for execution and then executing them.

• The key **difference between linker and loader** is that the **linker** generates the executable file of a program whereas, the **loader** loads the executable file obtained from the **linker** into main memory for execution. ... On the other hands, **loader** allocates space to an executable module in main memory.

Linking has two phases:

1. Static Linking: It is performed during the compilation of the source program. Linking is performed before execution in static linking. It takes a collection of the relocatable object file and command-line argument and generally fully linked object file that can be loaded and run.

Static linker performs two major tasks:

- **Symbol resolution**- it associates each symbol reference with exactly one symbol definition. Every symbol has a predefined task.
- **Relocation**- It relocates code and data section and modifies symbol reference to the relocated memory location.

The linker copies all library routines used in the program into the executable image. as a result, it is faster and more portable. No failure chance and less error chance.

2. Dynamic Linking- Dynamic linking is performed during the runtime. This linking is accomplished by placing the name of a shareable library in the executable image. There are more chances of error and failure chance. It requires less memory space as multiple programs can share a single copy of the library.

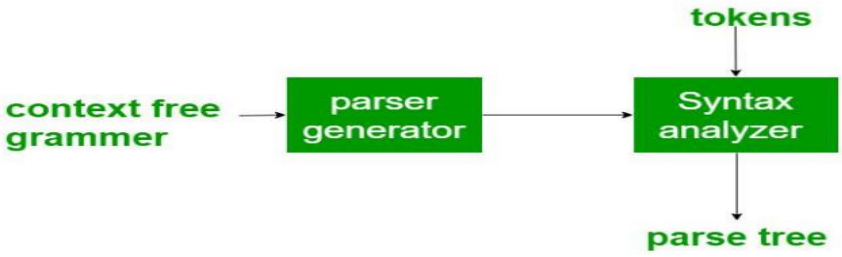
Here we can perform code sharing. It means we are using the same object a number of times in the program, instead of linking the same object again and again into the library, each module share information of an object with other module having the same object. The shared library needed in the linking 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 but all the code is not relocatable. it fixes the address at runtime.

1.3. Compiler Construction Tools

The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. These are specialized tools or programs that have been developed for helping implement various phases of a compiler. Some commonly used compiler construction tools include:

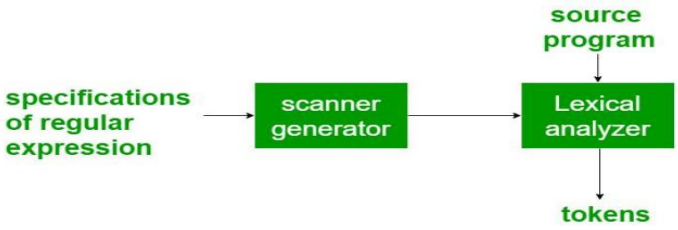
1) **Parser Generator:** -It produces syntax analyzers (parsers), normally from input that is based on a grammatical description of context-free grammar or programming language. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time. It consumes a large fraction of the running time of a compiler. Example-YACC (Yet Another Compiler-Compiler).

Example: PIC, EQM



2) **Scanner Generator:** It generates lexical analyzers, normally from the input a specification based on regular expressions. The basic organization of lexical analyzers is based on finite automation. It generates a finite automation to recognize the regular expression.

Example: Lex



3) **Syntax-Directed Translation Engines:** -These produce routines that walk the parse tree and as a result generate intermediate code with three address formats from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate. In this each translation is defined in terms of translations at its neighbor nodes in the tree.

4) **Automatic Code Generators:** It takes a collection of rules to translate intermediate language into machine language for the target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. The rules must include sufficient details to handle different possible access methods for data. Template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.

5) **Data-Flow analysis Engines:** It used in code optimization using data-flow analysis. It is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another.

Reading Assignment: Refer data flow analysis in compiler.

6) **Compile Construction Toolkits:** It provides an integrated set of routines that aids in build compiler components or in the construction of various phases of compiler.

Chapter Two: Token Specification

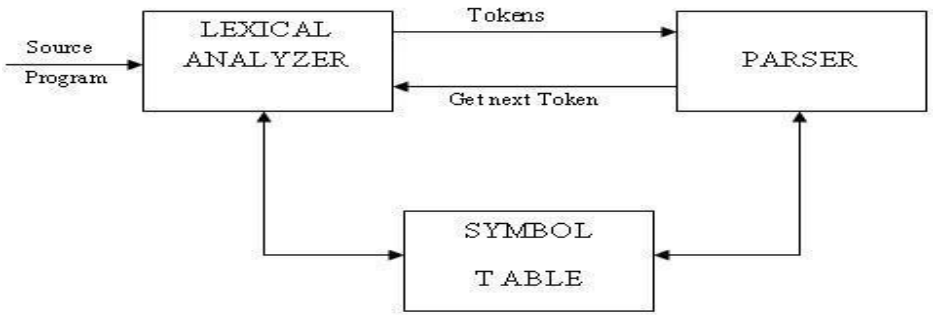
2. OVERVIEW OF LEXICAL ANALYSIS

To identify the tokens, we need some method of describing the possible tokens that can appear in the input stream. For this purpose, we introduce **regular expression**, a notation that can be used to describe essentially all the tokens of programming language.

Secondly, having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are **designed using transition diagrams and finite automata.**

Role of Lexical Analyzer

The Lexical Analyzer (LA) is the first phase of a compiler. Its main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



Upon receiving a 'get next token' command forms the parser; the lexical analyzer reads the input character until it can identify the next token. The LA return to the parser representation for the token it has found. The representation will be an integer code if the token is a simple construct such as parenthesis, comma or colon.

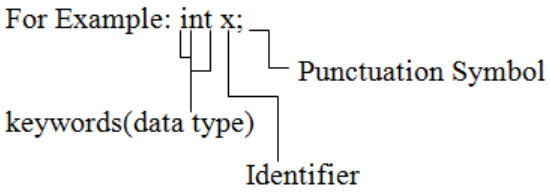
LA may also perform certain secondary tasks as the user interface. One such task is striping out from the source program the commands and white spaces in the form of blank, tab and newline characters. Another is correlating error message from the compiler with the source program**.**

2.1. Token Specification

TOKENS: A token is a string of characters, categorized according to the rules as a symbol (e.g., IDENTIFIER, NUMBER, COMMA). The process of forming tokens from an input stream of characters is called **tokenization**.

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language:

To specify the tokens the regular expressions are used. When a pattern is matched with the regular expression then tokens will be generated.



2.2. Recognition of Tokens

For programming language there are various types of tokens, i.e., keywords, identifiers, constants, operators, and punctuations etc.

generally, Tokens are described in a pair. (Token type, token value)

Token Type:	**Token Value:**
Describes the category or class of tokens. e.g., constant, identifiers, operators, punctuation symbols, keyword, etc	Gives information about the tokens. e.g., Name of the variable, current variable (or) pointer to symbol table

sum=3+2;

Lexeme	**Token type**
Sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

LEXEME: Collection or group of characters forming tokens is called Lexeme.

PATTERN: A pattern is a description of the form that the lexemes of a token may take. In the case of a keyword as a token, the pattern is just the sequence of characters that form the keyword. For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings.

The need of Lexical Analyzer

- ✓ The simplicity of design of compiler:
  The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.
- ✓ Compiler efficiency is improved Specialized buffering techniques for reading characters speed up the compiler process.
- ✓ Compiler portability is enhanced

Issues in Lexical Analysis

Lexical analysis is the process of producing tokens from the source program. It has the following issues:

• **Look ahead**

• **Ambiguities**

- **Look ahead:** *Look-ahead* is required to decide when one token will end and the next token will begin. The simple example which has look ahead issues are i *vs. if,* = *vs.* ==. Therefore, a way to describe the lexemes of each token is required.

A way needed to resolve ambiguities

• Is if it is two variables *i* and *f* or if?

• Is == is two equal signs =, = or ==?

• arr (5, 4) vs. fn (5, 4) *// as* array reference syntax and function call syntax are similar.

Hence, the number of looks ahead to be considered and a way to describe the lexemes of each token is also needed.

Regular expressions are one of the most popular ways of representing tokens.

- **Ambiguities**: The lexical analysis programs written with lex accept ambiguous specifications and choose the longest match possible at each input point. Lex can handle ambiguous specifications. When more than one expression can match the current input, lex chooses as follows:

• The longest match is preferred.

• Among rules which matched the same number of characters, the rule gave first is preferred.

Input Buffering:

To ensure that a right lexeme is found, one or more characters have to be looked up beyond the next lexeme.

• Hence a two-buffer scheme is introduced to handle large look-ahead safely.

• Techniques for speeding up the process of the lexical analyzer such as the use of sentinels to mark the buffer end have been adopted. There are three general approaches for the implementation of a lexical analyzer:

(i) By using a lexical analyzer generator, such as lex compiler to produce the lexical analyzer from a regular expression-based specification. In this, the generator provides routines for reading and buffering the input.

(ii) By writing the lexical analyzer in a conventional systems-programming language, using I/O facilities of that language to read the input.

(iii) By writing the lexical analyzer in assembly language and explicitly managing the reading of input.

Buffer Pairs: Because of a large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Fig shows the buffer pairs which are used to hold the input data.

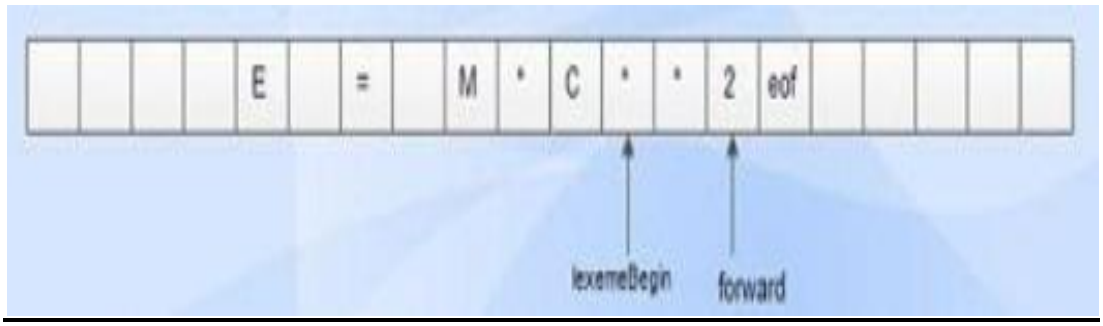


Fig: 1.3. Input Buffer

Scheme: One buffer scheme and Two buffer scheme.

• Consists of two buffers, each consists of N-character size which is reloaded alternatively.

• N-Number of characters on one disk block, e.g., 4096.

• N characters are read from the input file to the buffer using one system read command.

• *EOF* is inserted at the end if the number of characters is less than N.

Pointers

Two pointers *lexemeBegin* and *forward* are maintained.

lexeme Begin points to the beginning of the current lexeme which is yet to be found.

forward scans ahead until a match for a pattern is found.

• Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.

• A current lexeme is the set of characters between two pointers.

Disadvantages of this scheme

• This scheme works well most of the time, but the amount of look-ahead is limited.

• This limited look-ahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

(e.g.) DECLARE (ARGl, ARG2, . . ., ARGn) in PL/1 program;

• It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels (*EOF*)

• In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.

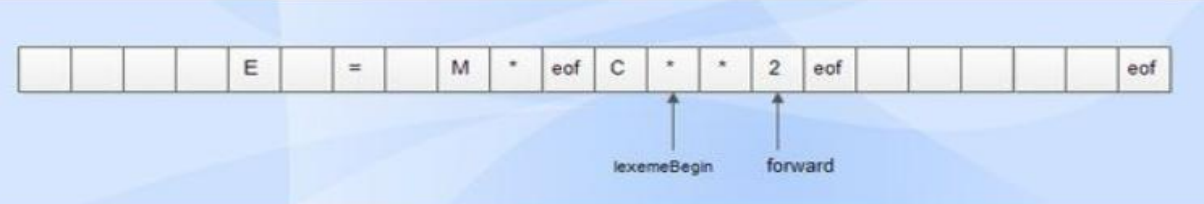
• Therefore, the ends of the buffer halves require two tests for each advance of the forward pointer.

Test 1: For the end of the buffer.

Test 2: To determine what character is read.

• The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.

• The Sentinel is a special character that cannot be part of the source program. *(EOF* character is used as Sentinel).



Advantages: Most of the time, it performs only one test to see whether the forward pointer points to an *EOF.* Only when it reaches the end of the buffer half or *EOF,* it performs more tests. Since N input characters are encountered between *EOF's,* the average number of tests per input character is very close to 1.

Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

Question: Differentiate between the drawbacks of one buffer scheme and two buffer schemes?

Example: Consider the following grammar fragment:

stmt → if expr then stmt

| if expr then stmt else stmt

| ε

expr → term relop term

| term

term → id

| num

Where the terminals **if, then, else, relop, id and num** generate sets of strings given by the following regular definitions:

if → if

then → then

else → else

relop → <|<=|=|<>|>|>=

id → letter (letter | digit) *

num → digit

For this language fragment, the lexical analyzer will recognize the keywords if, then, else, as well as the lexemes denoted by relop, id, and num. To simplify matters, we assume keywords are reserved; that is, they cannot be used as **identifiers**.

Example: Consider the following piece of code:

if(a<10)

i=i+2;

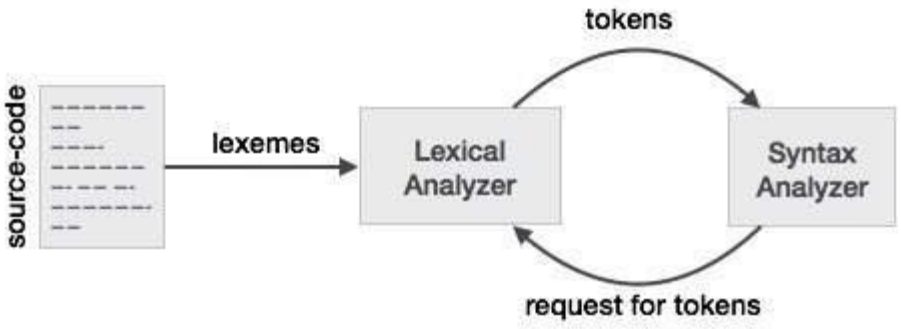
else

i=i-2;

Show how the lexical analyzer will generate the tokens streams.

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code.

If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.



Tokens

Lexemes are said to be a sequence of characters (alphanumeric) in a token. There are some predefined rules for every lexeme to be identified as a valid token. These rules are defined by grammar rules, by means of a pattern. A pattern explains what can be a token, and these patterns are defined by means of regular expressions.

In programming language, keywords, constants, identifiers, strings, numbers, operators and punctuations symbols can be considered as tokens.

For example, in C language, the variable declaration line

```
int value = 100;
```

contains the tokens:

```
int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).
```

Specifications of Tokens

Let us understand how the language theory undertakes the following terms:

Alphabets

Any finite set of symbols {0,1} is a set of binary alphabets, {0,1,2,3,4,5,6,7,8,9, A, B, C, D, E, F} is a set of Hexadecimal alphabets, {a-z, A-Z} is a set of English language alphabets.

Strings

Any finite sequence of alphabets is called a string. Length of the string is the total number of occurrence of alphabets, e.g., the length of the string **compiler** is 8 and is denoted by | **compiler** | = 8. A string having no alphabets, i.e., a string of zero length is known as an empty string and is denoted by ε (epsilon).

Special Symbols

A typical high-level language contains the following symbols: -

Arithmetic Symbols	Addition (+), Subtraction (-), Modulo (%), Multiplication (*), Division (/)
Punctuation	Comma (,), Semicolon (;), Dot(.), Arrow (→)
Assignment	=
Special Assignment	+=, /=, *=, -=
Comparison	==, !=, <, <=, >, >=
Preprocessor	#
Location Specifier	&
Logical	&, &&, |, ||, !
Shift Operator	>>, >>>, <<, <<<

Language

A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

For example:

```
int intvalue;
```

While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier int value.

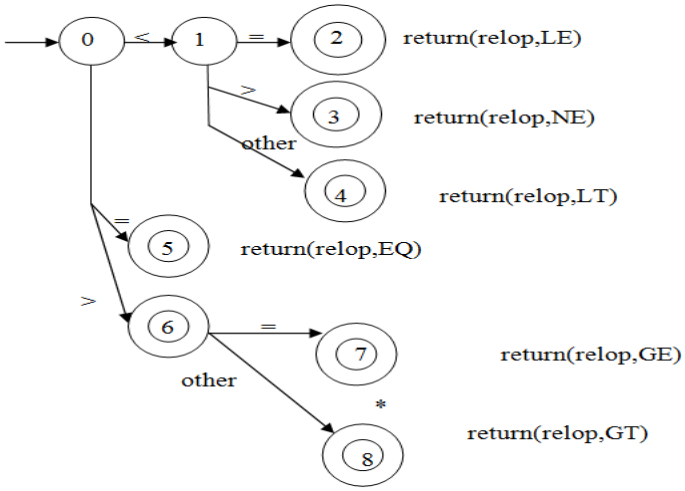
The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

The lexical analyzer also follows **rule priority** where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

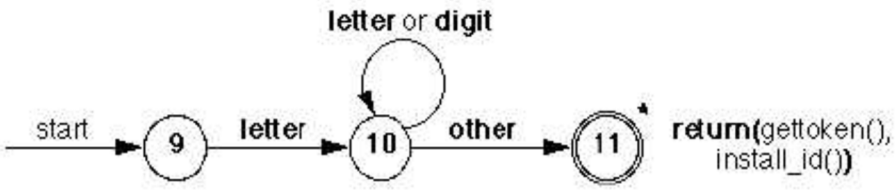
Transition diagrams

It is a diagrammatic representation to depict or represent the action that will take place when a lexical analyzer is **called** by the **parser** to get the next **token**. It is used to keep track of information about the characters that are seen as the forward pointer scans the input.

Below is example for transition diagram for Relational operators:



Transition diagram for identifiers and keywords:



2.3. Recognition Machines: Discuss NFA and DFA briefly**?**

2.4. NFA to DFA Conversion

Convert an NFA to DFA using the subset construction?

DFA → RE→NFA with λ→NFA→DFA show the conversion of each?

2.5. Error Recovery

Compiler Design - Error Recovery

A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input. Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:

- **Lexical**: name of some identifier typed incorrectly
- **Syntactical**: missing semicolon or unbalanced parenthesis
- **Semantically**: incompatible value assignment
- **Logical**: code not reachable, infinite loop

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as a semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement Mode

When a parser encounters an error, it tries to take corrective measures so that the rest of the inputs of statements allow the parser to parse ahead. For example, inserting a missing semicolon, replacing the comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error Productions

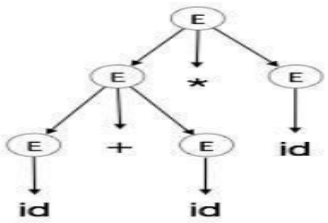
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global Correction

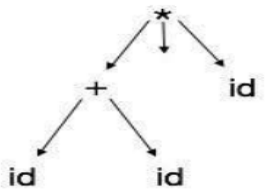
The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out the closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Abstract Syntax Trees

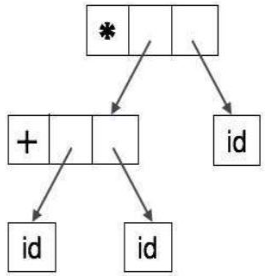
Parse tree representations are not easy to be parsed by the compiler, as they contain more details than actually needed. Take the following parse tree as an example:



If watched closely, we find most of the leaf nodes are a single child to their parent nodes. This information can be eliminated before feeding it to the next phase. By hiding extra information, we can obtain a tree as shown below:



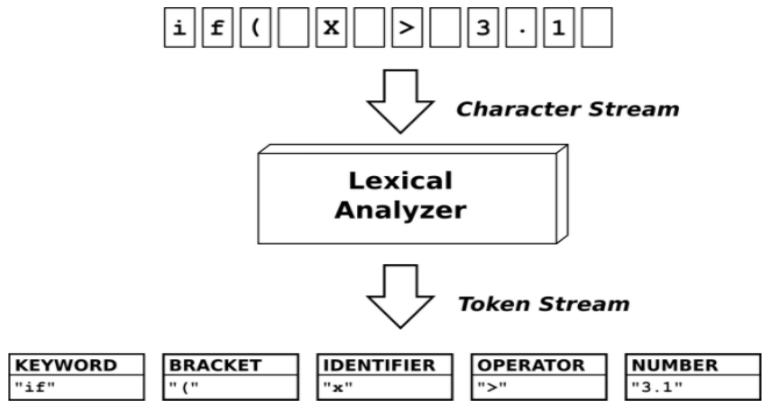
The abstract tree can be represented as:



ASTs are important data structures in a compiler with the least unnecessary information. ASTs are more compact than a parse tree and can be easily used by a compiler.

2.6. A Typical Lexical Analyzer Generator

It is a tool to generate lexical analyzers. A lexical analyzer is a program that transforms a stream of characters into a stream of 'atomic chunks of meaning', so-called tokens.



It is licensed under LGPL Version 2.1.

It does ...

- generate directly coded lexical analyzers, rather than table-based engines.
- respond to queries on Unicode properties and regular expressions on the command line.
- generate state transition graphs of the generated engines.
- It has ...
- Many examples.
- Modes that can be inherited and mode transitions that can be controlled.
- Line and column number counting as part of the generated engine.
- Include stacks for include file management.
- Support for indentation-based analysis (INDENT, DEDENT, NODENT tokens).
- Path and Template compression for minimal code size.

- Two token passing strategies: 'queue' and 'single token'.
- Support for customized token types.
- Event handlers for fine-grained control over analyzer actions.
- Buffer management providing tell & seek based on character indices even with dynamic size character coding (e.g., UTF-8, UTF-16).

2.7. DFA Analysis - Reading assignment?

Finite automata are a state machine that takes a string of symbols as input and changes its state accordingly. Finite automata are a recognizer for regular expressions. When a regular expression string is fed into finite automata, it changes its state for each literal. If the input string is successfully processed and the automata reaches its final state, it is accepted, i.e., the string just fed was said to be a valid token of the language in hand.

The mathematical model of finite automata consists of:

- Finite set of states (Q)
- Finite set of input symbols (Σ)
- One Start state (q0)
- Set of final states (qf)
- Transition function (δ)

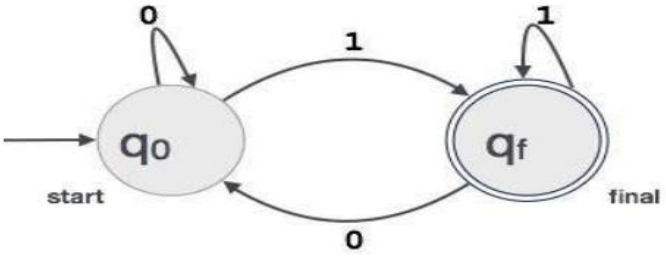
The transition function (δ) maps the finite set of state (Q) to a finite set of input symbols (Σ), Q × Σ ➔ Q

Finite Automata Construction

Let L(r) be a regular language recognized by some finite automata (FA).

- **States**: States of FA are represented by circles. State names are written inside circles.
- **Start state**: The state from where the automata start, is known as the start state. Start state has an arrow pointed towards it.
- **Intermediate states**: All intermediate states have at least two arrows; one pointing to and another pointing out from them.
- **Final state**: If the input string is successfully parsed, the automata is expected to be in this state. Final state is represented by double circles. It may have any odd number of arrows pointing to it and even number of arrows pointing out from it. The number of odd arrows is one greater than even, i.e., **odd = even+1**.
- **Transition**: The transition from one state to another state happens when a desired symbol in the input is found. Upon transition, automata can either move to the next state or stay in the same state. Movement from one state to another is shown as a directed arrow, where the arrows point to the destination state. If automata stay on the same state, an arrow pointing from a state to itself is drawn.

Example: We assume FA accepts any three-digit binary value ending in digit 1. FA = {Q (q_0, q_f), Σ (0,1), q_0, q_f, δ}



Chapter Three: Syntax Analysis

3. Parsing: Syntax analysis is the second phase of the compiler. It gets tokens as the input from lexical analysis and generates a **syntax tree or parse tree.**

•Syntax analyzers follow production rules defined by means of a context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types:

- Top-down parsing and
- bottom-up parsing.

The parse tree is constructed

- From top to bottom and from left to right.

Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.

2. An efficient parser can be constructed automatically from a properly designed grammar.

3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.

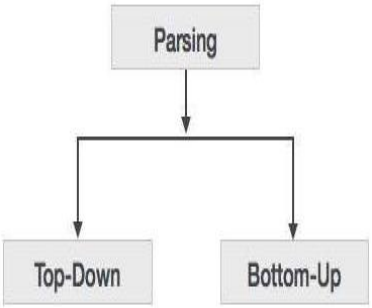
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

Parsing: It is the process of analyzing a continuous stream of input in order to determine its grammatical structure with respect to a given formal grammar.

Parse tree: Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals. The first node of a parse tree is the starting symbols of the grammar in the production rule and the last or least nodes of a parse tree are terminals.

3.1. Types of Parsing:

1. **Top-Down Parsing:** When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing. Example: **LL Parsers.**
2. **Bottom-Up Parsing:** A parser can start with input and attempt to rewrite it into the start symbol. Example: **LR Parsers.**



3.2. Top-Down Parsing

It can be viewed as an attempt to find a left-most derivation from left to right for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of Top-Down Parsing:

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING(RDP)

Recursive descent parsing is one of the top-down parsing techniques that uses a set of recursive procedures to scan its input**.** This parsing method may suffer from **backtracking**, that is, making repeated scans of the input.

Backtracking: It means, if one production fails, the syntax analyzer restarts the process using different rules of the same product. This technique may process the input string more than once to determine the right production.

Example: for backtracking

Consider the grammar G: S → cAd

A → ab | a, and the input string w=cad.

In this case, the parser parses the input "cad" at the first it will try the production:

S → cAd → cabd, in this case, the parsing doesn't get the right one result so it will backtrack and attempt the second production. so, s→cAd→cad. Now parsing is a success.

Exercise:

S → rXd | rZd

X→ oa | ea

Z → ai, using this CFG, parse the input 'read','raid','reid'?

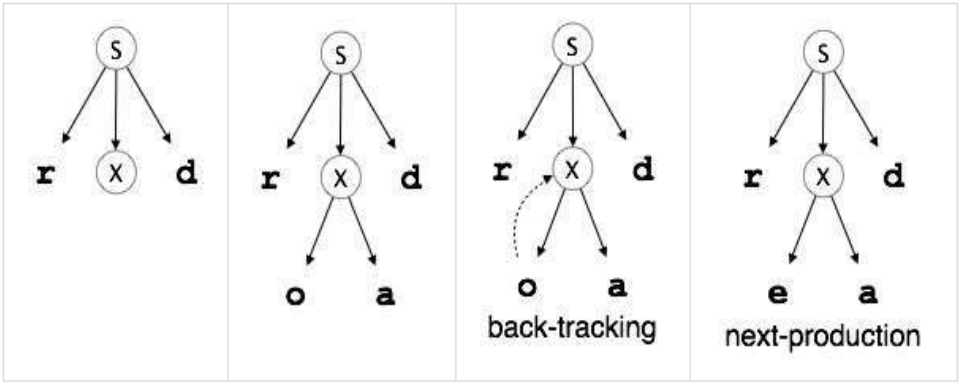
The parse tree can be constructed using the top-down approach?

Solution:

- For an input string: read, a top-down parser, will behave like this:
- It will start with S from the production rules and will match its yield to the left-most letter of the input, i.e., 'r'. The very production of S (S → rXd) matches with it. So, the top-down parser advances to the next input letter (i.e., 'e'). The parser tries to expand non-

terminal ‘X’ and checks its production from the left (X → oa). It does not match with the next input symbol. So, the top-down parser backtracks to obtain the next production rule of X, (X → ea).

- Now the parser matches all the input letters in an ordered manner. The string is accepted.



Example for recursive descent parsing:

Input string: a + b * c

Production rules:

S → E

E → E + T| E → E * T| E → T

T → id

Use top-down parsing to parse the above input?

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop. Hence, **Elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

2. Predictive Parsing

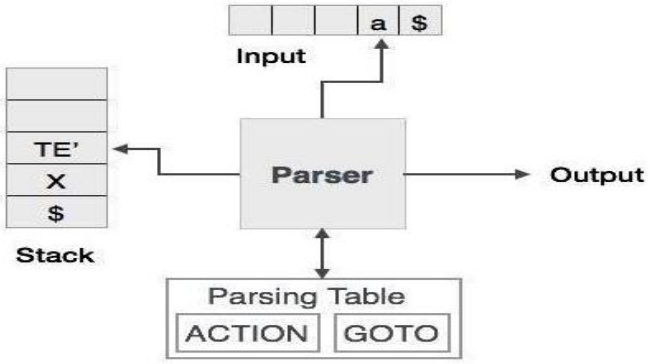
Predictive parsing is a special case of recursive descent parsing where no backtracking is required. The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives. The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream. Predictive parser is a recursive descent parser, which

has the capability to predict which production is to be used to replace the input string. The predictive parser does not suffer from **backtracking**. To accomplish its tasks, the predictive parser uses a look-ahead pointer, which points to the next input symbols. To make the parser back-tracking free, the predictive parser puts some constraints on the grammar and accepts only a class of grammar known as LL(k) grammar.

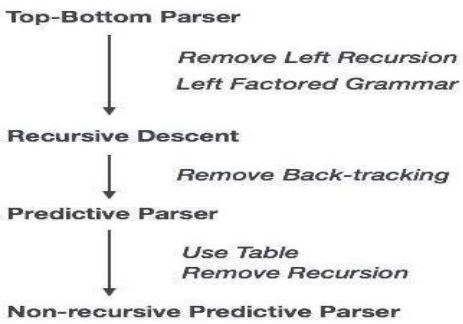
Input buffer: It consists of strings to be parsed, followed by $ to indicate the end of the input string.

Stack: It contains a sequence of grammar symbols preceded by $ to indicate the bottom of the stack. Initially, the stack contains the start symbol on top of $.

Parsing table: It is a two-dimensional array *M* [*A*, *a*], where **'*A*'** is a non-terminal and **'*a*'** is a terminal.



Predictive parsing uses a stack and a parsing table to parse the input and generate a parse tree. Both the stack and the input contain an end symbol **$** to denote that the stack is empty and the input is consumed. The parser refers to the parsing table to take any decision on the input and stack element combination.



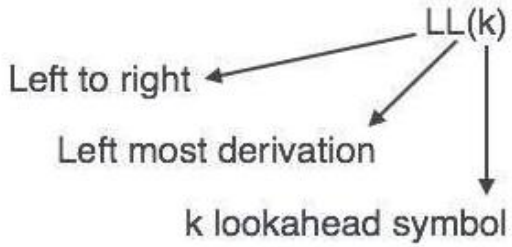
In recursive descent parsing, the parser may have more than one production to choose from for a single instance of input, whereas in predictive parser, each step has at most one production to choose.

There might be instances where there is no production matching the input string, making the parsing procedure to fail.

LL Parser

An LL Parser accepts LL grammar. LL grammar is a subset of context-free grammar but with some restrictions to get the simplified version, in order to achieve easy implementation. LL grammar can be implemented by means of both algorithms namely, recursive-descent or table-driven.

LL parser is denoted as LL(k). The first L in LL(k) is parsing the input from left to right, the second L in LL(k) stands for left-most derivation and k itself represents the number of look ahead. Generally, k = 1, so LL(k) may also be written as LL (1).



The goal of predictive parsing is to construct a top-down parser that never backtracks. To do so, we must transform a grammar in two ways:

- eliminate left recursion, and
- perform left factoring.

These rules eliminate most common causes for backtracking although they do not guarantee a completely backtrack-free parsing (called LL (1)

LL Parsing Algorithm

- We may stick to deterministic LL (1) for parser explanation, as the size of table grows exponentially with the value of k. Secondly, if a given grammar is not LL (1), then usually, it is not LL(k), for any given k.
- Given below is an algorithm for LL (1) Parsing:

Input: string ω

parsing table M for grammar G

Output: If ω is in L(G) then left-most derivation of ω,

error otherwise.

Initial State: $S, on stack (with S being start symbol) ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat let X be the top stack symbol and the symbol pointed by ip.

if X∈ V_t or $ if X = a POP X and advance ip.

else error () endif

else /* X is non-terminal */

if M [X, a] = X → Y1, Y2... Yk

POP X PUSH Yk, Yk-1... Y1 /* Y1 on top */

Output the production X → Y1, Y2... Yk

else error ()

endif

endif

until X = $ /* empty stack */

A grammar G is LL (1) if A → α | β are two distinct productions of G:

for no terminal, both α and β derive strings beginning with a.

at most one of α and β can derive empty string.

if β → t, then α does not derive any string beginning with a terminal in FOLLOW(A).

Predictive parsing table construction:

The construction of a predictive parser is aided by two functions associated with a grammar G:

1. FIRST
2. FOLLOW

Rules for first ():

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If *X* → ε is a production, then add ε to FIRST(*X*).
3. If *X* is non-terminal and X → *a*α is a production then add *a* to FIRST(X).
4. If X is non-terminal and *X* → *Y1 Y2…Yk* is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(*Yi*), and ε is in all of FIRST(*Y1*) …, FIRST(*Yi-1*); that is, *Y1…. Yi-1* => ε. If ε is in FIRST(*Yj*) for all j=1, 2..., k, then add ε to FIRST(*X*).

Rules for follow ():

1. If *S* is a start symbol, then FOLLOW(*S*) contains $.
2. If there is a production *A* → α*B*β, then everything in FIRST(β) except ε is placed in follow(*B*).
3. If there is a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

Algorithm for construction of predictive parsing table:

Input : Grammar *G*

Output : Parsing table *M*

Method:

1. For each production *A* → α of the grammar, do steps 2 and 3.
2. For each terminal *a* in FIRST(α), add *A* → α to *M* [*A*, *a*].
3. If ε is in FIRST(α), add A → α to *M* [*A*, *b*] for each terminal *b* in FOLLOW(*A*). If ε is in FIRST(α) and $ is in FOLLOW(*A*) , add *A* → α to *M* [*A*, $].
4. Make each undefined entry of *M* be **error**.

Example:

Consider the following grammar:

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating left-recursion the grammar is

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

First ():

FIRST(E) = {(, id}

FIRST(E’) = {+, ε}

FIRST(T) = {(, id}

FIRST(T’) = {*, ε}

FIRST(F) = {(, id}

Follow ():

FOLLOW(E) = {$,)}

FOLLOW(E’) = {$,)}

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

FOLLOW(T’) = {+, $,)}

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

Predictive parsing table :

NON-TERMINAL	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’ → ε	T’ → ε
F	F → id			F → (E)		

Stack implementation:

stack	Input	Output
$E	id+id*id $	
$E'T	id+id*id $	E → TE'
$E'T'F	id+id*id $	T → FT'
$E'T'id	id+id*id $	F → id
$E'T'	+id*id $	
$E'	+id*id $	T' → ε
$E'T+	+id*id $	E' → +TE'
$E'T	id*id $	
$E'T'F	id*id $	T → FT'
$E'T'id	id*id $	F → id
$E'T'	*id $	
$E'T'F*	*id $	T' → *FT'
$E'T'F	id $	
$E'T'id	id $	F → id
$E'T'	$	
$E'	$	T' → ε
$	$	E' → ε

3.3. Regular Expressions Vs Context Free Grammar

Compiler Design - Regular Expressions: The lexical analyzer needs to scan and identify only a finite set of valid string/token/lexeme that belong to the language in hand. It searches for the pattern defined by the language rules.

Regular expressions have the capability to express finite languages by defining a pattern for finite strings of symbols. The grammar defined by regular expressions is known as **regular grammar**. The language defined by regular grammar is known as **regular language**.

Regular expression is an important notation for specifying patterns. Each pattern matches a set of strings, so regular expressions serve as names for a set of strings. Programming language tokens can be described by regular languages. The specification of regular expressions is an example of a recursive definition. Regular languages are easy to understand and have efficient implementation.

There are a number of algebraic laws that are obeyed by regular expressions, which can be used to manipulate regular expressions into equivalent forms.

Operations

The various operations on languages are:

- Union of two languages L and M is written as
  L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as
  LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as

L* = Zero or more occurrence of language L.

Notations

If r and s are regular expressions denoting the languages L(r) and L(s), then

- **Union**: (r)|(s) is a regular expression denoting L(r) U L(s)
- **Concatenation**: (r)(s) is a regular expression denoting L(r)L(s)
- **Kleene closure**: (r)* is a regular expression denoting (L(r)) *
- (r) is a regular expression denoting L(r)

Precedence and Associativity

- *, concatenation (.), and | (pipe sign) are left associative
- * Has the highest precedence
- Concatenation (.) has the second highest precedence.
- | (pipe sign) has the lowest precedence of all.

Representing valid tokens of a language in regular expression

If x is a regular expression, then:

- x* means zero or more occurrence of x.
  i.e., it can generate {e, x, xx, xxx, xxxx, …}
- x+ means one or more occurrence of x.
  i.e., it can generate {x, xx, xxx, xxxx …} or x.x*
- x? means at most one occurrence of x
  i.e., it can generate either {x} or {e}.
  [a-z] is all lower-case alphabets of English language.
  [A-Z] is all upper-case alphabets of English language.
  [0-9] is all natural digits used in mathematics.

Representing occurrence of symbols using regular expressions

letter = [a – z] or [A – Z]

digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]

sign = [+ | -]

Representing language tokens using regular expressions

Decimal = $(sign)^{?}$ $(digit)^{+}$

Identifier = (letter) (letter | digit) *

The only problem left with the lexical analyzer is how to verify the validity of a regular expression used in specifying the patterns of keywords of a language. A well-accepted solution is to use finite automata for verification.

Context-free Grammar principles:

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

It implies that every Regular Grammar is also context-free, but there exist some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

Context-Free Grammar

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
- A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Example

We take the problem of palindrome language, which cannot be described by means of Regular Expression. That is, $L = \{w \mid w = w^R\}$ is not a regular language. But it can be described by means of CFG, as illustrated below:

$G = (V, \Sigma, P, S)$

Where:

$V = \{Q, Z, N\}$

$\Sigma = \{0, 1\}$

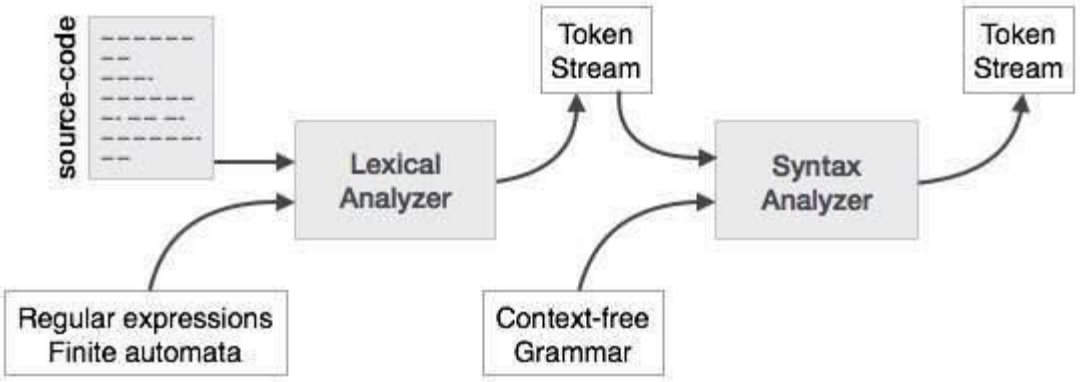
$P = \{Q \rightarrow Z \mid Q \rightarrow N \mid Q \rightarrow \varepsilon \mid Z \rightarrow 0Q0 \mid N \rightarrow 1Q1\}$

$S = \{Q\}$

This grammar describes palindrome language, such as: 1001, 11100111, 00100, 1010101, 11111, etc.

Syntax Analyzers

A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies, which we will learn later in this chapter.

Derivation

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

- Deciding the non-terminal which is to be replaced.
- Deciding the production rule, by which, the non-terminal will be replaced.

To decide which non-terminal to be replaced with production rule, we can have two options.

Left-most Derivation

If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

Right-most Derivation

If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example: Production rules:

E → E + E

E → E * E

E → id

Input string: id + id * id, the left-most derivation is:

E → E * E

E → E + E * E

E → id + E * E

E → id + id * E

E → id + id * id

Notice that the left-most side non-terminal is always processed first. The right-most derivation is:

E → E + E

E → E + E * E

E → E + E * id

E → E + id * id

E → id + id * id

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

We take the left-most derivation of a + b * c, The left-most derivation is:

E → E * E

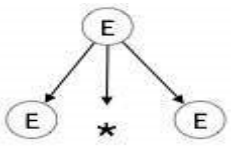
E → E + E * E

E → id + E * E

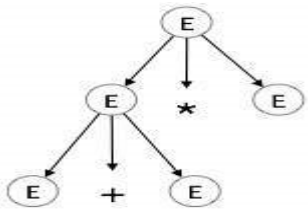
E → id + id * E

E → id + id * id

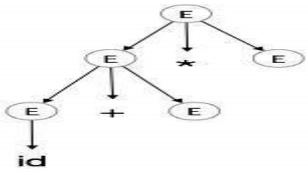
Step 1:

E → E * E	

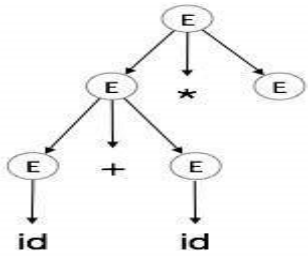
Step 2:

E → E + E * E	

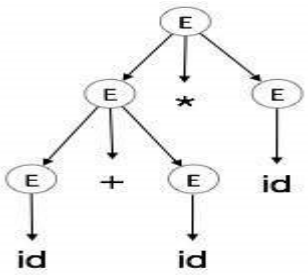
Step 3:

E → id + E * E	

Step 4:

E → id + id * E	

Step 5:

E → id + id * id	

In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.

- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

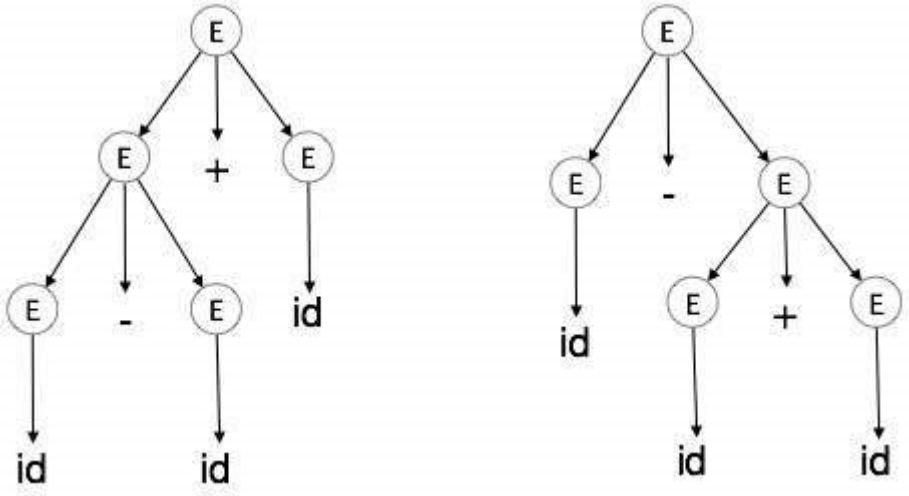
Ambiguity

A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.

Example

```
E → E + E
E → E – E
E → id
```

For the string id + id – id, the above grammar generates two parse trees:



The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

Associativity

If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.

Example

Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

```
id op id op id
```

it will be evaluated as:

```
(id op id) op id
```

For example, (id + id) + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

```
id op (id op id)
```

For example, id ^ (id ^ id)

Precedence

If two different operators share a common operand, the precedence of operators decides which will take the operand. That is, 2+3*4 can have two different parse trees, one corresponding to (2+3) *4 and another corresponding to 2+(3*4). By setting precedence among operators, this problem can be easily removed. As in the previous example, mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be interpreted as:

```
2 + (3 * 4)
```

These methods decrease the chances of ambiguity in a language or its grammar.

REGULAR EXPRESSION	CONTEXT-FREE GRAMMAR
It is used to describe the tokens of programming languages.	It consists of a quadruple where S → start symbol, P → production, T → terminal, V → variable or non- terminal.
It is used to check whether the given input is valid or not using **transition diagram.**	It is used to check whether the given input is valid or not using **derivation**.
The transition diagram has set of states and edges.	The context-free grammar has set of productions.
It has no start symbol.	It has start symbol.
It is useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.	It is useful in describing nested structures such as balanced parentheses, matching begin-end's and so on.

LL (1) Grammar

The parsing table entries are single entries. So, each location has not more than one entry. This type of grammar is called LL (1) grammar.

Consider this following grammar:

S → iEtS | iEtSeS | a

E → b

After eliminating left factoring, we have

S → iEtSS' | a

S'→ eS | ε

E → b

To construct a parsing table, we need FIRST () and FOLLOW () for all the non-terminals.

FIRST(S) = {i, a}

FIRST(S') = {e, ε}

FIRST(E) = {b}

FOLLOW(S) = {$, e}

FOLLOW(S') = {$, e}

FOLLOW(E) = {t}

Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS'		
S'			S' → eS S' → ε			S' → ε
E		E → b				

Since there are more than one production, the grammar is not LL (1) grammar.

Bottom-up parsing

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing. A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

Example:

Consider the grammar:

S → aABe

A → Abc | b

B → d

The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)

a**b**bcde (A → b)

a**Abc**de (A → Abc)

aA**d**e (B → d)

aABe (S → aABe)

RIGHTMOST DERIVATION

S → aA**B**e

→ a**A**de

→ a**A**bcde

→ abbcde

The reductions trace out the right-most derivation in reverse.

Handles

A handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Example:

Consider the grammar:

E → E+E

E → E*E

E → (E)

E → id

And the input string id1+id2*id3

The rightmost derivation is:

E → **E**+**E**

→ E+**E***E**

→ E+E***id3**

→ E+**id2***id3

→ **id1**+id2*id3

In the above derivation the underlined substrings are called **handles.**

Stack Implementation of Shift reduce Parsing

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Stack	Input	Action
$	id_1+id_2*id_3 $	shift
$ id_1	+id_2*id_3 $	reduce by E→id
$ E	+id_2*id_3 $	shift
$ E+	id_2*id_3 $	shift
$ E+id_2	*id_3 $	reduce by E→id
$ E+E	*id_3 $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

LR parsers-Implementation – LR Parsing Algorithm

An efficient bottom-up syntax analysis technique that can be used to parse a large class of CFG is called LR(*k*) parsing. The 'L' is for left-to-right scanning of the input, the 'R' for constructing a rightmost derivation in reverse, and the '*k*' for the number of input symbols. When '*k*' is omitted, it is assumed to be 1.

Types of LR parsing method:

1. SLR- Simple LR -Easiest to implement, least powerful.

2. CLR- Canonical LR- Most powerful, most expensive.

3. LALR- Look-Ahead LR Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:

Input: An input string *w* and an LR parsing table with functions *action* and *goto* for grammar G.

Output: If *w* is in L(G), a bottom-up-parse for *w*; otherwise, an error indication.

Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and *w*$ in the input buffer. The parser then executes the following program:

set *ip* to point to the first input symbol of *w*$;

repeat forever begin

let *s* be the state on top of the stack and

a the symbol pointed to by *ip*;

if *action* [*s*, *a*] = shift *s*' **then begin**

push *a* then *s*' on top of the stack;

advance *ip* to the next input symbol

end

else if *action* [*s*, *a*] = reduce A→β **then begin**

pop 2* | β | symbols off the stack;

let *s*' be the state now on top of the stack;

push A then *goto* [*s*', A] on top of the stack;

output the production A→ β

end

else if *action* [*s*, *a*] = accept **then**

return

else *error* ()

end

SLR

To perform SLR parsing, take grammar as input and do the following:

1. Find LR (0) items.

2. Completing the closure.

3. Compute *goto* (I, X), where, I is set of items and X is grammar symbol.

LR (0) items:

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

For example, production A → XYZ yields the four items:

A → **.** XYZ

A → X **.** YZ

$A \rightarrow XY \cdot Z$

$A \rightarrow XYZ \cdot$

Closure operation:

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. Initially, every item in I is added to closure(I).

2. If $A \rightarrow \alpha \cdot B\beta$ is in closure(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot \gamma$ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto (I, X) is defined to be the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X\beta]$ is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'

2. Construct the canonical collection of set of items C for G'

3. Construct the parsing action function *action* and *goto* using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

Output : The SLR parsing table functions *action* and *goto* for G'

Method:

1. Construct C = {I0, I1, …. In}, the collection of sets of LR (0) items for G'.

2. State *i* is constructed from I*i*.. The parsing functions for state *i* are determined as follows:

(a) If $[A \rightarrow \alpha \cdot a\beta]$ is in Ii and goto (Ii, *a*) = Ij, then set *action* [*i*, *a*] to "shift j". Here *a* must be terminal.

(b) If $[A \rightarrow \alpha \cdot]$ is in Ii , then set *action* [*i*, *a*] to "reduce $A \rightarrow \alpha$" for all *a* in FOLLOW(A).

(c) If [S'→S.] is in Ii, then set *action* [*i*, $] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR (1).

3. The *goto* transitions for state *i* are constructed for all non-terminals A using the rule:

If *goto* (Ii, A) = Ij, then *goto* [i, A] = *j*.

4. All entries not defined by rules (2) and (3) are made "error"

5. The initial state of the parser is the one constructed from the set of items containing [S'→. S].

Canonical & LALR

LALR PARSER

ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

Input: G'

Output: LALR parsing table functions with action and goto for G'.

Method:

1. Construct C = {I0, I1, ..., In} the collection of sets of LR (1) items for G'.
2. For each core present among the set of LR (1) items, find all sets having that core and replace these sets by the union.
3. Let C' = {J0, J1, ..., Jm} be the resulting sets of LR (1) items. The parsing actions for state i are constructed from Ji in the same manner as in the construction of the canonical LR parsing table.
4. If there is a conflict, the grammar is not LALR (1) and the algorithm fails.
5. The goto table is constructed as follows: If J is the union of one or more sets of LR (1) items, that is, J = I0U I1 U ... U Ik, then the cores of goto (I0, X), goto (I1, X), ..., goto (Ik, X) are the same, since I0, I1 , ..., Ik all have the same core. Let K be the union of all sets of items having the same core as goto (I1, X).
6. Then goto (J, X) = K.

Summary:

- Top Down Parsing
- Predictive Parsing
- Top-down parsing – Principles of CFG
- Regular Expressions Vs Context- Free Grammar
- Top-down parsing implementation-Recursive Descent Parsing
- Non recursive Predictive Parsing
- LL (1) Grammar
- Bottom-up parsing
- Handles
- Stack Implementation of Shift reduce Parsing
- LR parsers-Implementation – LR Parsing Algorithm
- SLR
- Canonical & LALR
- Error recovery
- Parser generator

Chapter IV: Syntax-Directed Translation (SDT)

4. Introduction

Brainstorming Question: Differentiate between Syntax directed definition and syntax directed translation?

4.1. STD Introduction: Parser uses a CFG(Context-free-Grammar) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation.

Definition: Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax directed translation rules use 1) lexical values of nodes, 2) constants & 3) attributes associated to the non-terminals in their definitions.

More over, Syntax-Directed Definitions are a generalization of context-free grammars in which:

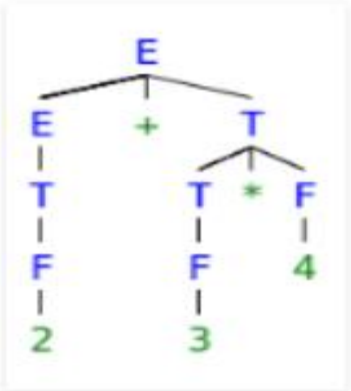
1. Grammar symbols have an associated set of Attributes;
2. Productions are associated with Semantic Rules for computing the values of attributes.

- Such formalism generates Annotated Parse-Trees where each node of the tree is a record with a field for each attribute. e.g., X.a indicates the attribute a of the grammar symbol X.
- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

4.2. Construction of Syntax Trees

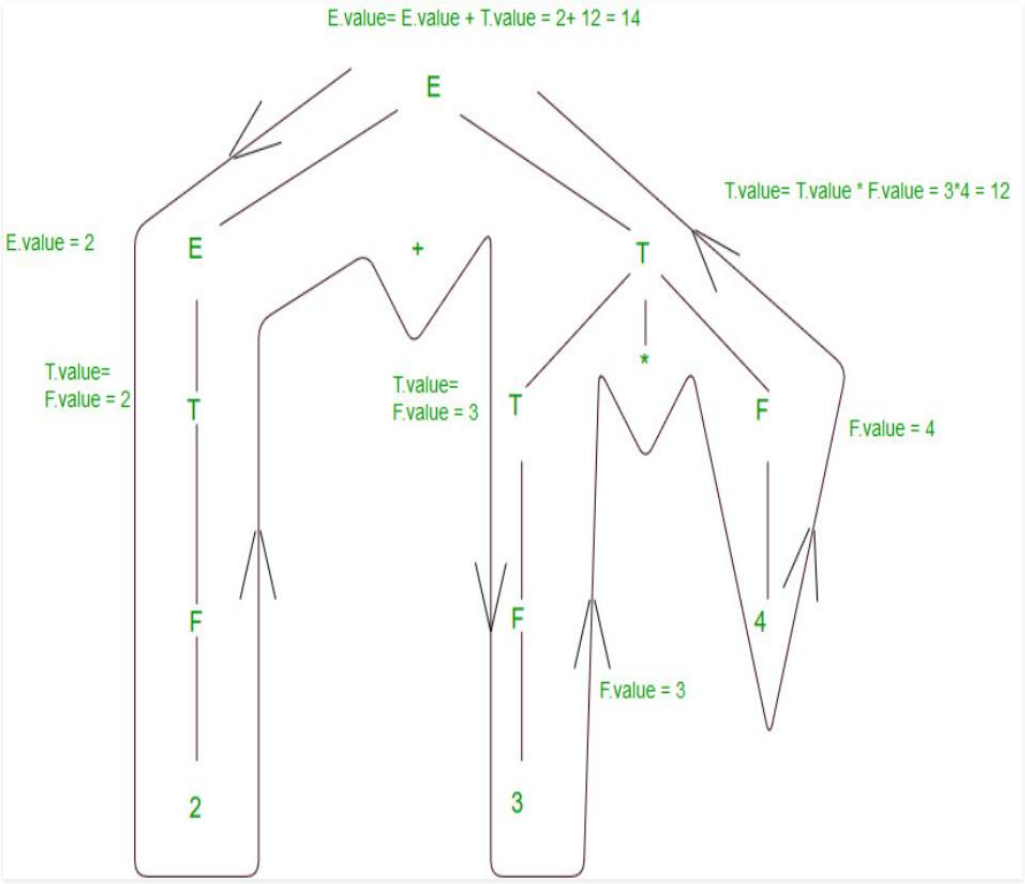
Let us consider the Grammar for arithmetic expressions. The Syntax Directed Definition associates to each non terminal a synthesized attribute called *val.*

Let's take a string to see how semantic analysis happens – S = 2+3*4. Parse tree corresponding to S would be



To evaluate translation rules, we can employ one depth first search traversal on the parse tree. This is possible only because SDT rules don't impose any specific order on evaluation until

children attributes are computed before parents for a grammar having all synthesized attributes. Otherwise, we would have to figure out the best suited plan to traverse through the parse tree and evaluate all the attributes in one or more traversals. For better understanding, we will move bottom up in left to right fashion for computing translation rules of our example.



Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents, as discussed above. Right hand side nodes are sometimes annotated with subscript 1 to distinguish between children and parent.

Additional Information

. We distinguish between two kinds of attributes:

1. Synthesized Attributes. They are computed from the values of the attributes of the children's nodes.

2. Inherited Attributes. They are computed from the values of the attributes of both the siblings and the parent nodes

Example:

E → E+T | T

T → T*F | F

F→ INTLIT

This is a grammar to syntactically validate an expression having additions and multiplications in it. Now, to carry out semantic analysis we will augment SDT rules to this grammar, in order to pass some information up the parse tree and check for semantic errors, if any. In this example we will focus on evaluation of the given expression, as we don't have any semantic assertions to check in this very basic example.

E → E+T	{E.val = E.val + T.val }	PR#1
E → T	{E.val = T.val}	PR#2
T → T*F	{ T.val = T.val * F.val }	PR#3
T → F	{ T.val = F.val }	PR#4
F → INTLIT	{F.val = INTLIT.lexval}	PR#5

For understanding translation rules further, we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has **Val** as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right-side nodes of the production rule and vice-versa. Generalizing, SDT are augmented rules to a CFG that associate 1) set of attributes to every node of the grammar and 2) set of translation rules to every production rule using attributes, constants and lexical values.

Synthesized Attributes: are such attributes that depend only on the attribute values of children nodes.

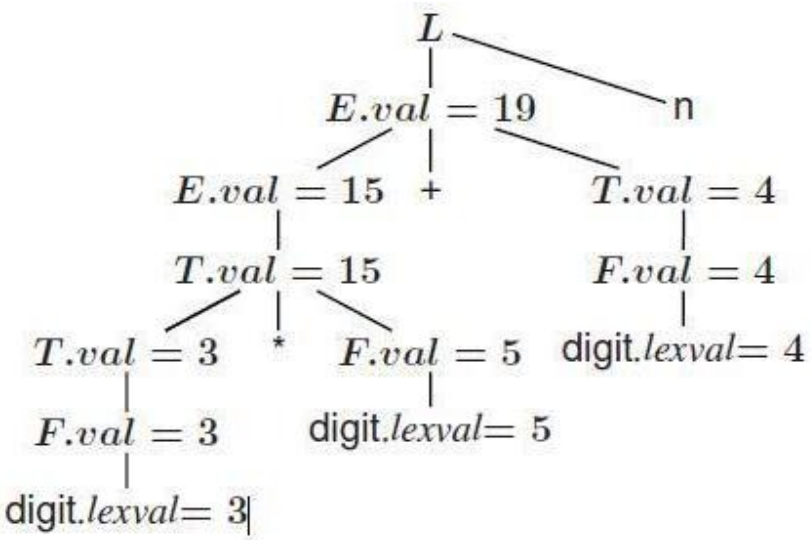
Thus [E -> E+T {E.val = E.val + T.val}] has a synthesized attribute val. corresponding to node E. If all the semantic attributes in an augmented grammar are synthesized, one depth first search traversal in any order is sufficient for semantic analysis phase.

Inherited Attributes: are such attributes that depend on parent and/or siblings attributes. Thus [Ep -> E+T {Ep. val = E.val + T.val, T.val = Ep. val}], where E & Ep are same production symbols annotated to differentiate between parent and child, has an inherited attribute val corresponding to node T.

S-ATTRIBUTED DEFINITION: An S-Attributed Definition is a Syntax Directed Definition that uses only synthesized attributes.

Evaluation Order: Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or Post Order, traversal of the parse-tree.

Example. The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



L-attributed definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A! X1 : :Xn depends only on attributes of X1; X2; : : : ;Xi 1 (symbols to the left of Xi in the RHS)

PRODUCTION	SEMANTIC RULE
$L \rightarrow En$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$digit.$lexval$

1. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.
2. An action must not refer to synthesized attribute of any symbol to the right of that action.
3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

S – attributed and L – attributed SDTs in SDT

Before coming up to S-attributed and L-attributed SDTs, here is a brief intro to Synthesized or Inherited attributes.

4.3. Types of attributes

Attributes may be of two types – Synthesized or Inherited.

1. **Synthesized Attributes –** A Synthesized attribute is an attribute of the non-terminal on the left-hand side of a production. Synthesized attributes represent information that is being passed up the parse tree. The attribute can take value only from its children (Variables in the RHS of the production).

For e.g., let's say A -> BC is a production of a grammar, and A's attribute is dependent on B's attributes or C's attributes than it will be synthesized attribute.

2. **Inherited Attributes:** An attribute of a non terminal on the right-hand side of a production is called an inherited attribute. The attribute can take value either from its children or from its siblings (variables in the LHS or RHS of the production).

For example, let's say A -> BC is a production of a grammar and B's attribute is dependent on A's attributes or C's attributes than it will be inherited attribute.

Now, let's discuss about S – attribute and L – attribute.

1. **S-attributed SDT –** If an SDT uses only synthesized attributes, it is called as S-attributed SDT. S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

   S-attributed SDT As depicted above, attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (RHS).

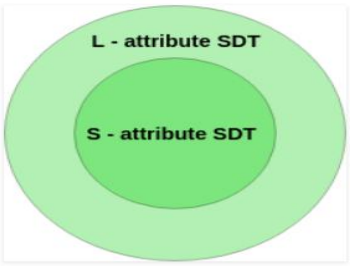
2. **L-attributed SDT –** If an SDT uses either synthesized attributes or inherited attributes with a restriction that it can inherit values from left siblings only, it is called as L-attributed SDT. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner. L-attributed SDT is form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.

   For example,

   ```
   A -> XYZ {Y.S = A.S, Y.S = X.S, Y.S = Z.S}
   ```

   is not an L-attributed grammar since Y.S = A.S and Y.S = X.S are allowed but Y.S = Z.S violates the L-attributed SDT definition as attributed is inheriting the value from its right sibling.

Note – If a definition is S-attributed, then it is also L-attributed but **NOT** vice-versa.



Example – Consider the given below SDT.

P1: S -> MN {S.val= M.val + N.val}

P2: M -> PQ {M.val = P.val * Q.val and P.val =Q.val}

Select the correct option.

A. Both P1 and P2 are S attributed.

B. P1 is S attributed and P2 is L-attributed.

C. P1 is L attributed but P2 is not L-attributed.

D. None of the above

Explanation –

The correct answer is option **C** as, In P1, S is a synthesized attribute and in L-attribute definition synthesized is allowed. So P1 follows the L-attributed definition. But P2 doesn't follow L-attributed definition as P is depending on Q which is RHS to it.

Chapter V: Type Checking

Outlines:

- **Type Systems**
- **Type Expressions**
- **Type casting and Type Conversion**

5. Introduction

A compiler must check that the source program follows both syntactic and semantic conventions of the source language. This checking, called *static checking,* detects and reports programming errors. Some examples of static checks:

1. Type checks – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.

2. Flow-of-control checks – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

5.1. Type Systems

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example: "if both operands of the arithmetic operators of +, - and * are of type integer, then the result is of type integer.

5.2. Type Conversions

Here, we specify a type checker for a simple language in which the type of each identifier must be declared before the identifier is used. The type checker is a translation scheme that synthesizes the type of each expression from the types of its sub expressions. The type checker can handle arrays, pointers, statements and functions.

A Simple Language

Consider the following grammar:

P → D ; E

D → D ; D | id : T

T → char | integer | array [num] of T | ↑ T

E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E

D → D ; D

D → id : T { *addtype* (id.*entry* , T.*type*) }

T → char { T.*type* : = char }

T → integer { T.*type* : = integer }

T → ↑ T1 { T.*type* : = pointer(T1.*type*) }

T → array [num] of T1 { T.*type* : = array (1… num.val , T1.*type*) }

In the above language,

→ There are two basic types : char and integer ;

→ *type_error* is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer)**.

TYPE CASTING IN C LANGUAGE

float f=10.10;

int i=(int)f;

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can typecast long to int. You can convert values from one type to another explicitly using the cast operator.

New data type should be mentioned before the variable name or value in brackets which to be typecast.

C type casting example program:

In the below C program, 7/5 alone will produce integer value as 1. So, type cast is done before division to retain float value (1.4).

```
1  #include <stdio.h>
2  int main ()
3  {
4      float x;
5      x = (float) 7/5;
6      printf("%f",x);
7  }
```

Output: 1.400000

WHAT IS TYPE CASTING IN C LANGUAGE?

Converting an expression of a given type into another type is known as type-casting. typecasting is more use in c language programming. Here, It is best practice to convert lower data type to higher data type to avoid data loss. Data will be truncated when the higher data type is converted to lower. For example, if a float is converted to int, data which is present after the decimal point will be lost.

There are two types of typecasting in c language.

TYPES OF TYPECASTING IN C

S.No	Types of typecasting in C Programming
1	Implicit Conversion
2	Explicit Conversion

1. IMPLICIT CONVERSION

Implicit conversions do not require any operator for converted. They are automatically performed when a value is copied to a compatible type in the program.

Here, the value of a has been promoted from int to double and we have not had to specify any type-casting operator. This is known as a standard conversion.

Example :-

```
1  #include<stdio.h>
2  #include<conio.h>
3  void main()
4  {
5      int i=20;
6      double p;
7      clrscr();
8
9      p=i; // implicit conversion
10
11     printf("implicit value is %d",p);
12
13     getch();
14 }
```

Output:- implicit value is 20.

2. EXPLICIT CONVERSION

In C language, Many conversions, especially those that imply a different interpretation of the value, require an explicit conversion. We have already seen two notations for explicit type conversion.

They are not automatically performed when a value is copied to a compatible type in the program.

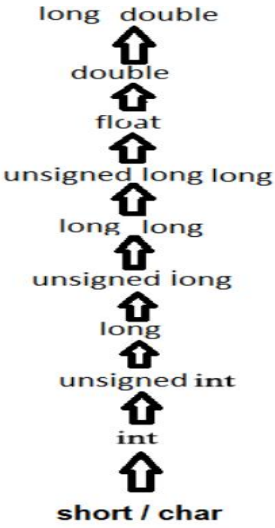
Example :-

```
1  #include<stdio.h>
2  #include<conio.h>
3  void main()
4  {
5      int i=20;
6      short p;
7      clrscr();
8
9      p = (short) i; // Explicit conversion
10
11     printf("Explicit value is %d",p);
12
13     getch();
14 }
```

Output:- Explicit value is 20.

Usual Arithmetic Conversion

The usual arithmetic conversions are implicitly performed to cast their values in a common type, C uses the rule that, in all expressions except assignments, any implicit type conversions made from a lower size type to a higher size type as shown below:



Inbuilt Typecast Functions In C:

There are many inbuilt typecasting functions available in C language which performs data type conversion from one type to another.

S.No	Typecast Function	Description
1	atof()	Convert string to Float
2	atoi()	Convert string to int
3	atol()	Convert string to long
4	itoa()	Convert int to string
5	ltoa()	Convert long to string

We are recently taking a survey from different programmers who are available in google plus social media. We did a survey for type casting in c language.

DIFFERENCE BETWEEN TYPE CASTING AND TYPE CONVERSION

Whenever there is a need to convert one data type to another the two terms comes in our mind “type casting” and “type conversion”. When the two types are compatible with each other, then the conversion of one type to other is done automatically by the compiler that is cold type conversion but remember we can store a large data type into the other for example we cannot store a float into int because a float is greater than int. whereas, type casting is to be explicitly done by the programmer.

BASIS FOR COMPARISON	TYPE CASTING	TYPE CONVERSION
Definition	When a user can convert the one data type into other then it is called as the type casting.	Type Conversion is that which automatically converts the one data type into another.
Implemented	Implemented on two 'incompatible' data types.	Implemented only when two data types are 'compatible'.
Operator	For casting a data type to another, a casting operator '()' is required.	No operator required.
Implemented	It is done during program designing.	It is done explicitly while compiling.
Conversion type	Narrowing conversion.	Widening conversion.

Example	int x;
	byte y;
	…
	…
	y= (byte) x;
		int x=3;
		float y;
		y=a; // value in y=3.000.

Chapter VI: Intermediate Code Generation (ICG)

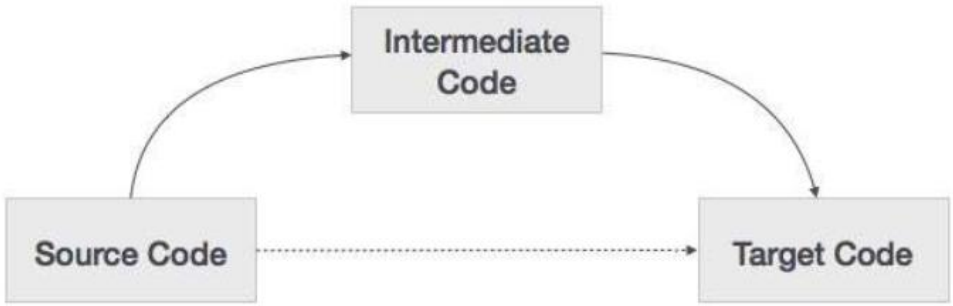
Outlines:

- **Intermediate languages**
- **Intermediate Language Representations**
- **Declarations**
- **Declarations in procedures**
- **The flow of Control Statements**
- **Back Patching**
- **Procedure Call**

6. Introduction

6.1. Intermediate Languages

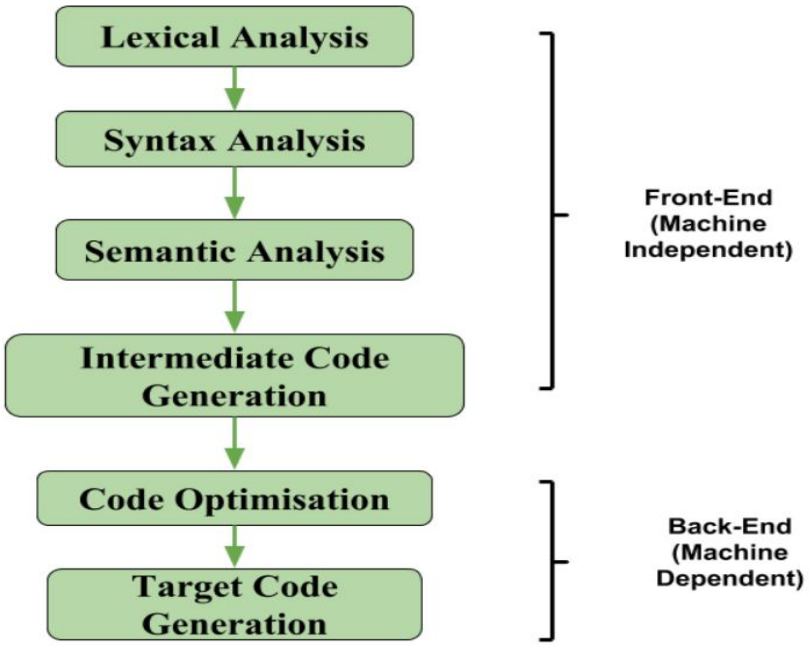
A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code. Here is the secret. In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code which can be understood by the machine. This facilitates ***retargeting***: enables attaching a backend for the new machine to an existing front end.



1. If a compiler translates the source language to its target machine language without having the option for generating an intermediate code, then for each new machine, **a full native compiler is required.**
2. Intermediate code **eliminates** the need for a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
3. The second part of compiler, synthesis, is changed according to the target machine.
4. It becomes easier to apply the source code modifications to improve code performance by applying code **optimization** techniques on the intermediate code.

The benefits of using machine independent (intermediate codes) are:

- Because of the machine independent intermediate code, portability will be enhanced. For example, suppose, if a compiler translates the source language to its target machine language without having the option for generating an intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications.
- **Retargeting** is facilitated
- It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code. The following are phases of a compiler as two parts of front and back end.



Logical Structure of a Compiler Front End

A compiler front-end is organized as in the figure above, where parsing, static checking, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. All schemes can be implemented by creating a syntax tree and then walking the tree.

6.2. Intermediate Code Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

- **High-Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

- **Low-Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code). If we generate machine code directly from source code then for 'N' target machine we need to have 'N' optimizers and 'N' code generators but if we will have a machine-independent intermediate code, we will have only **one** optimizer.

The following are commonly used intermediate code representation:

1. Postfix Notation
2. Three Address Code
3. Syntax Tree

1. Postfix Notation:

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a + b The postfix notation for the same expression places the operator at the right end as ab +. In general, if e1 and e2 are any postfix expressions, and + is any binary operator, the result of applying + to the values denoted by e1 and e2 is postfix notation by e1e2 +. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permit only one way to decode a postfix expression. In postfix notation, the operator follows the operand.

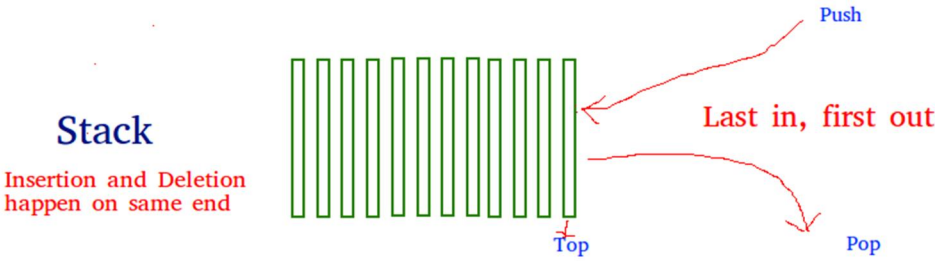
Example: The postfix representation of the expression (a – b) * (c + d) + (a – b) **is**: ab – cd + ab -+*.

1) Stack Data Structure (Introduction and Program)

The stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last in First Out) or FILO (First in Last Out).

Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.
- **Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
- **Peek or Top:** Returns the top element of the stack.
- **isEmpty:** Returns true if the stack is empty, else false.



How to understand a stack practically? There are many real-life examples of the stack. Consider the simple example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e., the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO/FILO order.

Time Complexities of operations on the stack:

push (), pop (), isEmpty () and peek () all take O (1) time. We do not run any loop in any of these operations.

Applications of the stack:

- Balancing of symbols
- Infix to Postfix /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, stock span problem, histogram problem.
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver
- In Graph Algorithms like Topological Sorting and Strongly Connected Components

Implementation:

There are two ways to implement a stack:

- Using array
- Using linked list

Assignment, Show the Implementation of Stack using Arrays and linked list?

2) Stack | (Infix to Postfix)

Infix Expression: The expression of the form "a op b". When an operator is in-between every pair of operands.

Postfix Expression: The expression of the form "a b op". When an operator is followed for every pair of operands.

Why postfix representation of the expression? The compiler scans the expression either from left to right or from right to left.

Consider the below expression: a op1 b op2 c op3 d If op1 = +, op2 = *, op3 = +

The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it. The result is then added to d after another scan.

The repeated scanning makes it very inefficient. It is better to convert the expression to postfix (or prefix) form before evaluation.

The corresponding expression in postfix form is abc*+d+. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.

2. If the scanned character is an operand, output it.

3. Else,

3.1 If the precedence of the scanned operator is greater than the precedence of the operator in the stack (or the stack is empty), push it.

3.2 Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.

4. If the scanned character is an '(', push it to the stack.

5. If the scanned character is an ')', pop and output from the stack until an '('is encountered.

6. Repeat steps 2-6 until infix expression is scanned.

7. Pop and output from the stack until it is not empty.

Given an infix expression. Convert the infix expression to postfix expression.

Infix expression: The expression of the form "a op b". When an operator is in-between every pair of operands.

Postfix expression: The expression of the form "a b op". When an operator is followed for every pair of operands.

Input: The first line of input contains an integer T denoting the number of test cases. The next T lines contain an infix expression. The expression contains all characters and ^,*,/,+,-.

Output: Output the infix expression to postfix expression.

Constraints:

1<=T<=100

1<=length of expression<=30

Example: Input: 2

a+b*(c^d-e) ^(f+g*h)-i

A*(B+C)/D

Output: abcd^e-fgh*+^*+i-

ABC+*D/

Assignment, Show the implementations of Infix to postfix and infix to prefix?

2.Three Address Code (TAC or 3AC) Rules

A statement involving no more than three references (two for operands and one for the result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of form x = y op z, here x, y, z will have an address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement. The general form is x: = y op z, where "op" is an operator, x is the result, and y and z are operands. x, y, z are **variables**, **constants**, or "**temporaries**". A three-address instruction consists of at most 3 addresses for each statement. It is a linear representation of a binary syntax tree. Explicit names correspond to the interior nodes of the graph. E.g., for a looping statement, syntax tree represents components of the statement, whereas three-address code contains labels and jump instructions to represent the flow-of-control as in machine language. A TAC instruction has at most one operator on the RHS of an instruction; no built-up arithmetic expressions are permitted.

Generally, The Three-address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by the compiler. The compiler decides the order of operation given by three address code.

General representation:

```
x = y op z
```

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator.

Example – 1) The three-address code for the expression a + b * c + d:
T 1 = b * c
T 2 = a + T 1
T 3 = T 2 + d
T 1, T 2, T 3 are temporary variables.

2) x + y * z can be translated as t1 = y * z, t2 = x + t, where t1 & t2 are compiler–generated temporary names. Since it unravels multi-operator arithmetic expressions and nested control-flow statements, it is useful for target code generation and optimization.

3) Convert the expression a * – (b + c) into three address code. T1=b+c, T2=Uminus T1, T3=a*T2.

4) Write three address code for following code

```
for (i = 1; i<=10; i++)
 {a[i] = x * 5;}
```

$$
\begin{aligned}
& i = 1 \\
L : \; & t_1 = x * 5 \\
& t_2 = \&a \\
& t_3 = sizeof(int) \\
& t_4 = t_3 * i \\
& t_5 = t_2 + t_4 \\
& *t_5 = t_1 \\
& i = i + 1 \\
& \text{if } i<=10 \text{ goto } L
\end{aligned}
$$

Implementation of Three Address Code: There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

2.1. Quadruples: It is structure with consist of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage:

- Easy to rearrange code for global optimization.
- One can quickly access the value of temporary variables using symbol table.

Disadvantage:

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

Example – Consider expression a = b * – c + b * – c. The three-address code is:

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

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

Quadruple representation

Mathematical and Boolean expressions

a + b => (+, a, b, tmp1)

a * (b + c) => (+, b, c, tmp1), (*, a, tmp1, tmp2)

a < b => ((uminus, a, b, tmp)

- Unary operators

-a => (uminus, a, , tmp)

Assignment: a = a + b => (+, a, b, tmp), (:=, tmp, , a)

Mathematical and Boolean expressions

a + b => (+, a, b, tmp1)

a * (b + c) => (+, b, c, tmp1), (*, a, tmp1, tmp2)

a < b => ((uminus, a, , tmp)

Unary operators: -a => (uminus, a, , tmp)

Assignment: a = a + b => (+, a, b, tmp), (:=, tmp, , a)

Declarations: int a, b => -

int a=5 => (:=, 5, , a)

Array reference (?? problem here)

a = x[i] => ([]=, x, i, tmp1), (:=, tmp1, , a)

x[i] = a => ([]=, x, i, tmp1), (:=, a, , tmp1)

Type conversion (it of = 'int to float')

a = 1; b = a + 0.7 => (:=, 1, , a)

(itof, a, , tmp1) (+, tmp1, 0.7, tmp2) (:=, tmp2, , b)

The semantic actions which generate quads from the parse tree thus needs to check the type of each element to ensure that arguments are of the same type. If not, generate a quad to convert types.

Unconditional Jump

(jmp, <jump_address>, ,)

Conditional Jump

(<, i, 5, t1) ; condition
(jtrue, L10, t1,)

Remember:

- Whether conditional or unconditional, the destination is in the 2nd argument.
- The destination address will be a quad number (each quad has a number, the first is 1.

If-else statement

if (a < b) then c = 1 else c = 2

=>

```
1       (<, a, b, t1)
2       (jfalse, 5, t1, )
3       (:=, 1, , c)
4       (jmp, 6, , )
5       (:=, 2, , c)
6
```

For loop

```
for (i=0, i<5, i++)  do <body>  end

=> 1 (:=, 0,   , i)      ; initialisation
   2 (<,  i, 5, t1)      ; condition
   3 (jfalse, 7, t1, )
   … <body of loop>
   5 (+, i, 1, i)        ; Iteration
   6 (jmp, 2, , )
   7
```

While statement

```
while (a < b) do a = a +1

=>

1       (<, a, b, t1)
2       (jfalse, 6, t1, , )
3       (+, a, 1, tmp1)
4       (:=, tmp1, , a)
5       (jmp, 1, , )
6
```

Exercise:

Given the following program:

```
int a = 2, b = 8, c = 4, d;
for(j=0; j<=10; j++){
          a = a * (j* (b/c));
          d = a * (j* (b/c));
}
```

Show the intermediate code generated from this source code in the form of quadruples.

2.2. Triples: This representation doesn't make use of an extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consists of only three fields namely op, arg1 and arg2.

Disadvantage:

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Example: Consider expression a = b * – c + b * – c

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

Triples representation

2.3. Indirect Triples: This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to quadruple

representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Example: Consider expression a = b * – c + b * – c

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

#	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Indirect Triples representation

Question: Write quadruple, triples and indirect triples for following expression: (x + y) * (y + z) + (x + y + z)

Explanation – The three-address code is:

t1 = x + y

t2 = y + z

t3 = t1 * t2

t4 = t1 + z, t5 = t3 + t4

#	Op	Arg1	Arg2	Result
(1)	+	x	y	t1
(2)	+	y	z	t2
(3)	*	t1	t2	t3
(4)	+	t1	z	t4
(5)	+	t3	t4	t5

Quadruple representation

#	Op	Arg1	Arg2
(1)	+	x	y
(2)	+	y	z
(3)	*	(1)	(2)
(4)	+	(1)	z
(5)	+	(3)	(4)

Triples representation

List of pointers to table

#	Op	Arg1	Arg2
(14)	+	x	y
(15)	+	y	z
(16)	*	(14)	(15)
(17)	+	(14)	z
(18)	+	(16)	(17)

#	Statement
(1)	(14)
(2)	(15)
(3)	(16)
(4)	(17)
(5)	(18)

Indirect Triples representation

Exercise

Write quadruple, triples and indirect triples for following expression:

1. A: =-B(C/D)

2. -(a+b) *(c*d)/(a+b+c)

Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res. res = arg1 op arg2.

Example: a = b + c, b is represented as arg1, c is represented as arg2, + as the op and an as res.

Detection of a Loop in Three-Address Code

Loop optimization is the phase after the Intermediate Code Generation. The main intention of this phase is to reduce the number of lines in a program. In any program majority of the time is spent by any program is actually inside the loop for an iterative program. In case of the recursive program a block will be there and the majority of the time will present inside the block.

Loop Optimization:

1. To apply loop optimization, we must first detect loops.
2. For detecting loops, we use Control Flow Analysis (CFA) using Program Flow Graph (PFG).
3. To find program flow graph we need to find Basic Block

Basic Block: A basic block is a sequence of three address statements where control enters at the beginning and leaves only at the end without any jumps or halts.

Finding the Basic Block: In order to find the basic blocks, we need to find the leaders in the program. Then a basic block will start from one leader to the next leader but not including the next leader. Which means if you find out that lines no 1 is a leader and line no 15 is the next leader, then the line from 1 to 14 is a basic block, but not including line no 15.

Identifying leader in a Basic Block –

1. The First statement is always a leader
2. A Statement that is the target of conditional or unconditional statement is a leader

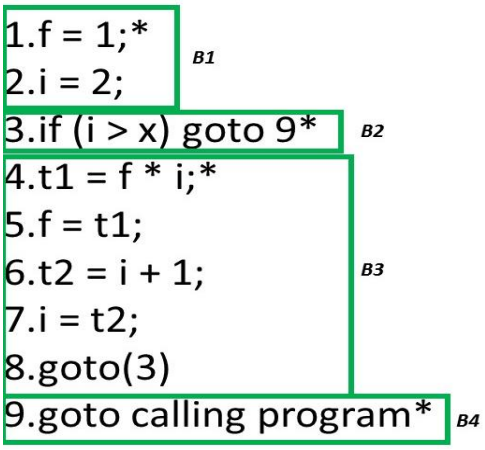
3. A Statement that follows immediately a conditional or unconditional statement is a leader

```
fact(x)
{int f = 1;
   for (i = 2; i <= x; i++)
      f = f * i;
   return f;}
```

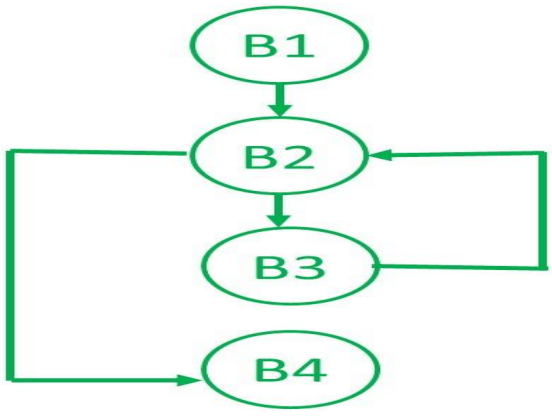
Three Address Code of the above C code:

1. f = 1;
2. i = 2;
3. if (i > x) goto 9
4. t1 = f * i;
5. f = t1;
6. t2 = i + 1;
7. i = t2;
8. goto (3)
9. goto calling program

Leader and Basic Block:



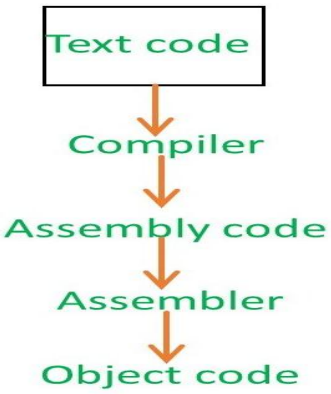
Control Flow Analysis –



If the control enters B1 there is no other option after B1, it has to enter B2. Now, if control enters B2, then depending on the condition control will flow, if the condition is true we are going to line no 9, which means 9 is nothing but B4. But if the condition is a false control goes to next block B3. After B3, there is no condition at all we are direct goes to 3rd statement B2. Above control flow graph have a cycle between B2 and B3 which is nothing but a loop.

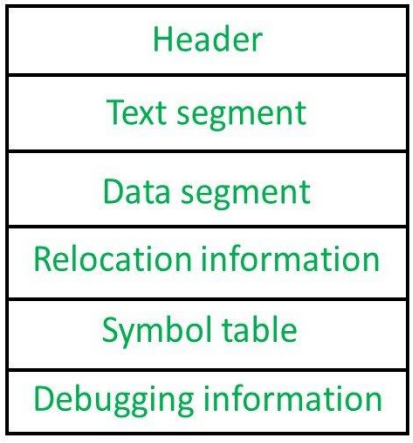
Introduction of Object Code

Let assume that, you have a c program, then you give the C program to compiler and compiler will produce the output in assembly code. Now, that assembly language code will give to the assembler and assembler is going to produce you some code. That is known as **Object Code**.



But, when you compile a program, then you are not going to use both compiler and assembler. You just take the program and give it to the compiler and compiler will give you the directly executable code. The compiler is actually combined inside the assembler along with loader and linker. So, all the module kept together in the compiler software itself. So, when you calling gcc, you are actually not just calling the compiler, you are calling the compiler, then assembler, then linker and loader.

Once you call the compiler, then your object code is going to present in Hard-disk. This object code contains various part –



OBJECT CODE

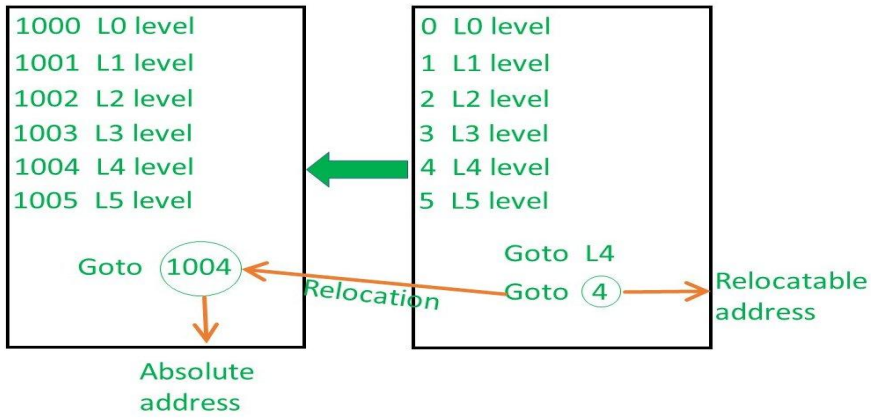
- **Header** –

  The header will say what are the various parts present in this object code and then point those parts. So, header will say where the text segment is going to start and a pointer to it and where the data segment going to start and it say where the relocation information and symbol information there.

It is nothing but like an index, like you have a textbook, there an index page will contains at what page number each topic present. Similarly, the header will tell you, what are the palaces at which each information is present. So that later for other software it will be useful to directly go into those segments.

- **Text segment –**It is nothing but the set of instruction.
- **Data segment –**Data segment will contain whatever data you have used. For example, you might have used something constraint, then that going to be present in the data segment.
- **Relocation Information –**Whenever you try to write a program, we generally use symbol to specify anything. Let us assume you have instruction 1, instruction 2, instruction 3, instruction 4….

Now if you say somewhere Goto L4 (Even if you don't write Goto statement in the high-level language, the output of the compiler will write it), then that code will be converted into object code and L4 will be replaced by Goto 4. Now Goto 4 for the level L4 is going to work fine, as long as the program is going to be loaded starting at address no 0. But most of the cases, the initial part of the RAM is going to be dedicated to the operating system. Even if it is not dedicated to the operating system, then might be some other process which will already be running at address no 0. So, when you are going to load the program into memory, means if the program has to be load in the main memory, it might be loaded anywhere. Let us say 1000 is the new starting address, then all the addresses has to be changed, that is known as **Reallocation**.



The original address is known as **Relocatable address** and the final address which we get after loading the program into main memory is known as the **Absolute address**.

Symbol table –

It contains every symbol that you have in your program. for example, int a, b, c then, a, b, c is the symbol. it will show what are the variables that your program contains.

Debugging information –

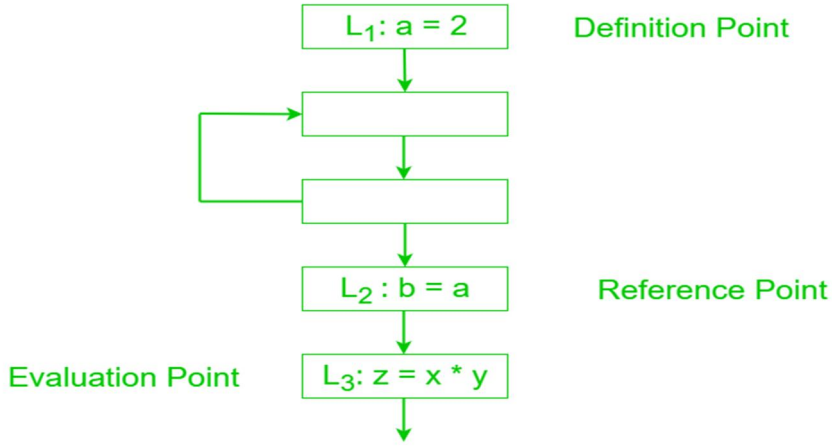
It will help to find how a variable is keeping on changing.

Data Flow Analysis in Compiler

It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done. In general, its process in which values are computed using data flow analysis. The data flow property represents information which can be used for optimization.

Basic Terminologies –

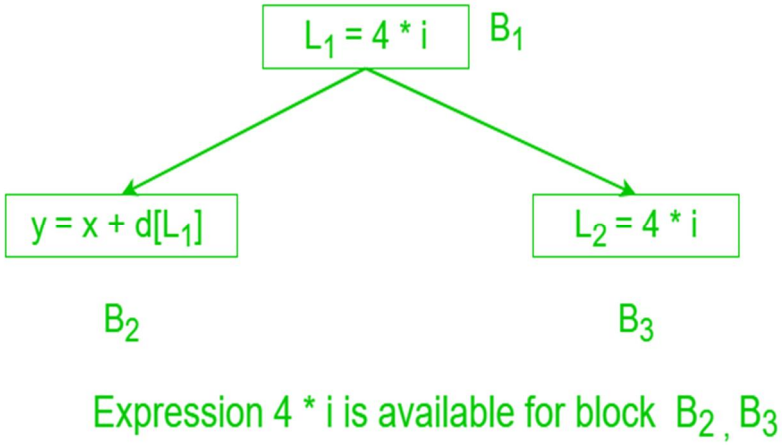
- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



Data Flow Properties –

- **Available Expression –** A expression is said to be available at a program point x iff along paths its reaching to x. An Expression is available at its evaluation point. An expression a+b is said to be available if none of the operands gets modified before their use.

Example –

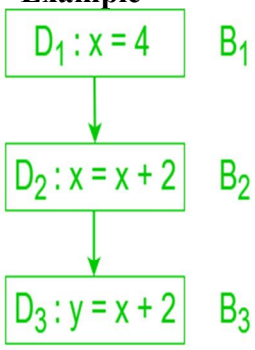


Advantage –

It is used to eliminate common sub expressions.

▪ **Reaching Definition –** A definition D is reaching a point x if there is path from D to x in which D is not killed, i.e., not redefined.

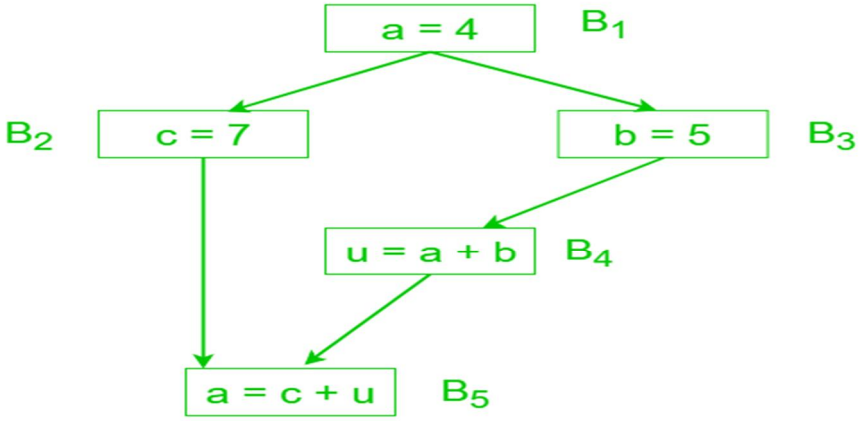
Example –



Advantage –It is used in constant and variable propagation.

Live variable – A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.

Example –



a is live at block B_1 , B_3 , B_4 but killed at B_5

Advantage –

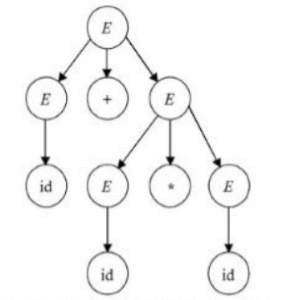
1. It is useful for register allocation.
2. It is used in dead code elimination.

- **Busy Expression –** An expression is busy along a path iff its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.

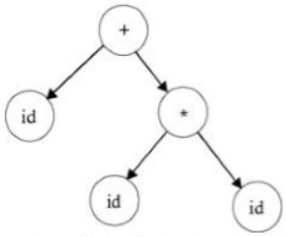
  Advantage –

  It is used for performing code movement optimization.

3) **Syntax Tree:** Syntax tree is nothing more than condensed form of a parse tree. The operator and keyword nodes of the parse tree are moved to their parents and a chain of single productions is replaced by single link in syntax tree the internal nodes are operators and child nodes are operands. To form syntax tree put parentheses in the expression, this way it's easy to recognize which operand should come first.



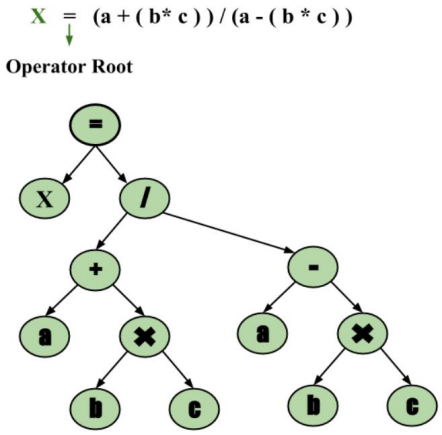
Parse tree for the string id+id*id.



Syntax tree for id+id*id.

Example:

x = (a + b * c) / (a – b * c)



6.3. Declarations

A variable or procedure has to be declared before it can be used. Declaration involves allocation of space in memory and entry of type and name in the symbol table. A program may be coded and designed keeping the target machine structure in mind, but it may not always be possible to accurately convert a source code to its target language.

Taking the whole program as a collection of procedures and sub-procedures, it becomes possible to declare all the names local to the procedure. Memory allocation is done in a consecutive manner and names are allocated to memory in the sequence they are declared in the program. We use offset variable and set it to zero {offset = 0} that denote the base address.

The source programming language and the target machine architecture may vary in the way names are stored, so relative addressing is used. While the first name is allocated memory starting from the memory location 0 {offset=0}, the next name declared later, should be allocated memory next to the first one.

Example:

We take the example of C programming language where an integer variable is assigned 2 bytes of memory and a float variable is assigned 4 bytes of memory.

```
int a;
float b;
Allocation process:
{offset = 0}
  int a;
  id.type = int
  id.width = 2
offset = offset + id.width
{offset = 2}
  float b;
  id.type = float
  id.width = 4
```

```
offset = offset + id.width
```

```
{offset = 6}
```

To enter this detail in a symbol table, a procedure *enter* can be used. This method may have the following structure:

```
enter(name, type, offset)
```

This procedure should create an entry in the symbol table, for the variable *name*, having its type set to type and relative address *offset* in its data area.

Complicated declarations in C: Most of the times declarations are simple to read, but it is hard to read some declarations which involve pointer to functions. For example, consider the following declaration from "signal.h".

void (*bsd_signal(int, void (*)(int)))(int);

Let us see the steps to read complicated declarations.

1) Convert C declaration to postfix format and read from left to right.
2) To convert expression to postfix, start from innermost parenthesis, If innermost parenthesis is not present then start from declarations name and go right first. When first ending parenthesis encounters then go left. Once the whole parenthesis is parsed then come out from parenthesis.
3) Continue until complete declaration has been parsed.

Let us start with a simple example. Below examples are from "K & R" book.

1) int (*fp) ();

Let us convert above expression to postfix format. For the above example, there is no innermost parenthesis, that's why we will print declaration name i.e., "fp". Next step is, go to the right side of the expression, but there is nothing on the right side of "fp" to parse, that's why go to the left side. On the left side, we found "*", now print "*" and come out of parenthesis. We will get postfix expression as below.

```
fp * () int
```

Now read postfix expression from left to right. e.g., fp is a pointer to function returning an int.

Let us see some more examples.

2) int (*daytab) [13]

Postfix: daytab * [13] int, meaning: daytab is pointer to array of 13 integers.

3) void (*f[10]) (int, int)

Run on IDE

Postfix : f[10] * (int, int) void, Meaning : f is an array of 10 of pointer to function(which takes 2 arguments of type int) returning void

4) char (*(*x())[]) ()

ostfix : x () * [] * () char, Meaning : x is a function returning pointer to array of pointers to function returnging char

5) char (*(*x[3])())[5]

Postfix : x[3] * () * [5] char, Meaning : x is an array of 3 pointers to function returning pointer to array of 5 char's

6) int *(*(*arr[5])()) ()

Postfix : arr[5] * () * () * int, Meaning : arr is an array of 5 pointers to functions returning pointer to function returning pointer to integer

7) void (*bsd_signal(int sig, void (*func)(int)))(int);

Postfix : bsd_signal(int sig, void(*func)(int)) * (int) void, Meaning : bsd_signal is a function that takes integer & a pointer to a function(that takes integer as argument and returns void) and returns pointer to a function(that take integer as argument and returns void)

Declarations in Procedures

Declarations : Single Procedure:

Each local name has a symbol-table entry

Entry contains

– Type : type attribute

– Relative addressin procedure's activation record (e.g, Java int takesfour bytes, double takes eight) : .width attribute

Code-generating part of compiler uses a global variable offset that isreset to zero as new procedures are declared Code-generator proc. enter (name, type, width) createssymboltable entry with thisinfo Consider just declarations part of program.

P--> D { offset := 0 } // initializaion

D--> D ; D

D --> id : T { enter(id.name, T.type, offset); offset := offset + T.width) }

T-->integer { T.type := integer; T.width := 4 }

T--> double { T.type := double; T.width := 8 }

Problem : (look at first production)

"offset := 0" code will be performed aslast action, instead of first

Solution: "dummy" non-terminal: P M D M ∈ { offset :=0}

Declarations: Nested Procedures

ML Example:

let fun f(x) =

let fun g(x, y) = x + y

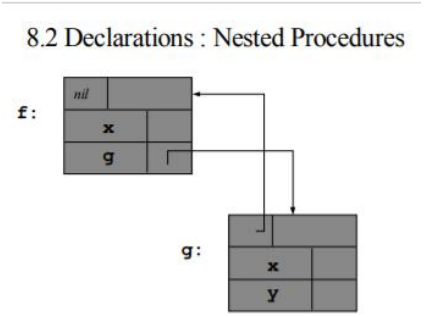
in f(g(3, 4)) end end

Abstractly:

P D D D , D | id : T | fun id (D) = S

Each procedure getsits own symbol table

Table for "inner" declaration pointsto table for "outer" – linked list, searched from end to begnning.



Semantic rules are a little more complicated:

– mktable (previous) creates and returns a new s.t., pointing to previously created one

– enter(table, name, type, offset) actslike enter, but using a specified table

– addwidth (table, width) records cumulative width of all entriesin table

– enterproc (table, name, newtable) creates new entry for procedure name in existing table, and assigns newtable to be the s.t. for this new proc.

Can do it all in one pass, if we use stacks tblptr, offset to keep track of symbol tables and offsets.

P-->M D { addwidth(top(tblptr), top(offset)); pop(tblptr); pop(offset); } // done after M's // actions

M--> ∈ { t := mktable(nil); push(t, tblptr); push (0, offset) }

D-->D , D

D fun id N (D1) = S { t := top (tblptr); addwidth (t, top (offset)); pop(tblptr); pop(offset); enterproc(top (tblptr), id.name, t) } continues.

D-->id : T { enter (top (tblptr), id.name, T.type, top (offset)); top(offset) := top(offset) + T.width) }

N-->∈ { t := mktable(top (tblptr)); push(t, tblptr); push (0, offset) }

Let's try it with let fun f(x:int) = let fun g(x:int, y:int) = x + y in f(g(3, 4)) end

6.5. Flow Control Statements: We now use back patching to translate flow-of-control statements in one pass.

- Consider statements generated by the following grammar:

  S →→ if (B)S | if (B) S else S | while (B) S| {L}|A;
  L → LS |S

  Here S denotes a statement, L a statement, L a statement list, A an assignment-statement, and B a Boolean expression.

  Note that there must be other productions, such as

```
100:  if x < 100 goto _
101:  goto _
102:  if x > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

  (a) After back patching 104 into instruction 102.

```
100:  if x < 100 goto _
101:  goto 102
102:  if y > 200 goto 104
103:  goto _
104:  if x != y goto _
105:  goto _
```

  (b) After back patching 102 into instruction 101.

  those for assignment-statements.

- The productions gave, however, are sufficient to illustrate the techniques used to translate flow-of-control statements.
- We make the tacit assumption that the code sequence in the instruction array reflects the natural flow of control from one instruction to the next.
- If not, then explicit jumps must be inserted to implement the natural sequential flow of control.

6.6. Backpatching: Backpatching usually refers to the process of resolving forward branches that have been planted in the code, e.g., at 'if' statements, when the value of the target becomes known, e.g., when the closing brace or matching 'else' is encountered. **Backpatching** comes into play in the intermediate code generation step of the compiler. There are times when the compiler has to execute a **jump** instruction but it doesn't know where to yet. So, it will fill in some kind of filler or blank value at this point and remember that this happened. Then once the value is found that will actually be used the compiler will go back "**backpatch**" and fill in the correct value.

Exercise: show pack patching with Examples?

Procedure Calls: Explain Procedure call? Differentiate between Local Procedure Call (LPC) and Remote Method Invocation vs. Remote procedure call (RPC)?

Procedure Calls: What you need to know?

Anything related to calling a procedure is considered so basic that you should understand it thoroughly.
Runtime Stack, and how it relates to a program's address space for a typical processor and operating system.

Stack top of memory = bottom of stack stack frame, activation record caller-saved registers procedure arguments return address callee-saved registers automatic local variables temporary register storage allocated at runtime initial contents determined by OS frame pointer in register stack pointer in register
heap (dynamic storage) allocated storage (new, malloc) allocated at runtime, by explicit request pointer or reference in program sometimes with garbage collection
data (static storage) global variables static local variables constants allocated at compile time address + offset in program global pointer in register
text, code program, machine instructions allocated at compile time program counter (instruction address register) in processor

Call / Return mechanism

- instructions
- register usage conventions
- stack management
    - stack frame, activation record
    - stack base and limit

Stack frame

- storage allocated on stack for one procedure call
- general organization same for all procedures
- specific contents defined for each procedure
- frame pointer is constant during procedure execution
- stack pointer may vary during procedure execution

On the caller's side

- save registers for later use
    - stack push
    - caller-save registers
- arrange for the procedure arguments to be found by the procedure
    - copy some arguments to registers

    - push additional arguments onto stack
- call the procedure
    - save return address in register or on stack
    - jump to procedure starting address
- later, the return value (if any) is in a pre-determined place, and control is transferred back to the return address
- copy the return value
    - move to another register or store to memory (pop the stack if necessary)
- throw away the procedure arguments
    - adjust stack pointer
- restore saved registers
    - stack pop
- continue

On the callee's side

- control is transferred from the caller to the starting address
    - return address is in register or on stack
    - arguments are in registers and on stack
- save registers for later use
    - stack push
    - arguments and return address
    - caller's frame pointer
    - other callee-save registers
- allocate storage for automatic local variables
    - set the frame pointer, adjust the stack pointer
    - assign initial values
- allocate storage for temporaries
    - adjust the stack pointer
- work
    - stack may be used for additional temporaries
    - push, use, pop
- put return value in appropriate place
    - register or stack
- throw away the local variables and temporaries
    - adjust the stack pointer
- restore saved registers
    - stack pop
    - arguments and return address
    - caller's frame pointer
    - other callee-saved registers
- jump through the return address register

Some additional details

- Why do the registers need to be saved and restored?
- How do the caller and callee agree on which registers to use for which purposes, and how to organize the runtime stack?
    - procedure call and register usage conventions are determined by the processor architecture and compilers
- How does the program know the procedure's starting address?
    - compiler maintains addresses symbolically or as symbol + offset
    - linker replaces symbols and symbol + offset with virtual addresses
    - virtual memory address translation yields physical address
- How does the compiler give names to procedures, so the linker can find them?
    - This is important if you are linking to C, C++ and Fortran modules.
- How does the debugger find out where things are?

    - stack frame, symbol table
- How does a performance measurement program find out where the time goes?
    - compiler and operating system support
    - time and event counters in the processor
- How does the system decide how much space to allocate to the stack?

How to write a procedure in an assembly language

- Design in C, write and debug in C, then translate to assembly language.
    - first, use the compiler
    - second, use the compiler switches effectively, especially -O
    - if that's not good enough, translate by hand
- Read the compiler output
    - -S option on most compilers, to generate assembly code from C.
- Use the compiler to generate a template if it is not possible to do the whole job in C.
- Why are you still using C?
    - compiler output from C is easier to read than compiler output from C++
    - many operating systems are written in C

MIPS generic procedure

- arguments in registers and on stack, return address in register

```
    # int procedure_name(int arg0) { ... }
.text   .align 2
.globl procedure_name
procedure_name:
      # arg0 is found in register $a0
...#
return value is placed in register $v0
  jr $ra    # register $ra holds return address
```

- Where do the additional arguments go?
- How are other argument types treated?

Intel IA-32 (x86) generic procedure

- arguments and return address on stack
- very few general registers
    - caller-save, %eax, %edx, %ecx
    - callee-save, %ebx, %esi, %edi
    - frame pointer, %ebp
    - stack pointer, %esp
    - integer or pointer return value in %eax

C Standard main procedure

```
int main(int argc, char * argv[]) { ... }
```

- argc, argv assigned from the command line or mouse clicks by the command interpreter
    - in Unix, from arguments to one of the exec() functions
- How does getopt() work?
    - iterate through argv[], with state information
- How does getenv() work?
    - global pointer environ

Typical parts of an object file

- this is the .o file on Unix, there is usually a system command to print information about the file
    - Solaris, elfdump
    - GNU/Linux, objdump
    - Mac OS X, otool
- object file header, describes the parts of the file
- text module, machine instructions
- data module, static and dynamic parts

- relocation information, describes addresses that may need to be modified by the linker
- symbol table, undefined labels, such as external references
- debugging information
- Unix tools
    - nm (symbol table)
    - strings (printable strings)

Optimizations by the compiler

- defer the stack pop on the caller's side
- simplify register use for leaf procedures
- eliminate tail recursion
- omit the frame pointer
- inline simple functions
- align the procedure's starting address to a word boundary (branches are faster)
- align the stack top to a word boundary (local variable access is faster)

Other compiler and linker support

- bounds-checking on stack accesses and stack growth
- insert profiling code
- retain symbol table for debugger (usually requires frame pointer)
- generate testing and code coverage files
- dynamic linking of shared libraries
- generate position-independent code for use in dynamic shared libraries

Further subdivisions of the process address space

- user space
    - runtime stack, fixed stack bottom allocated at program start, initial values pushed according to runtime environment and command-line arguments for main(), stack top moves as procedures are called and as registers are stored temporarily
    - dynamic data storage, allocated/deallocated at runtime by program request
    - static data storage, fixed locations allocated at program start, relative addresses computed by compiler and linker
        - initialized. Special cases include:
            - read-only, useful for character string storage
        - uninitialized (this is often called BSS, block starting segment). Choices are:
            - truly uninitialized (dangerous)
            - set to zero by the loader
            - set to some pattern of values by the loader
    - text segment, contains the user program, allocation and addressing similar to initialized read-only static data storage
    - shared libraries, position-independent system libraries, mapped to the process address space at run time from a single copy in shared physical memory
    - memory-mapped files
    - unallocated and unused, often the largest part of the user space
- kernel space
    - not directly accessible by the user program, addressed via system calls
    - organization is similar to user space, but managed by the operating system on behalf of the user program or for itself
- memory-mapped I/O registers
- reserved space
    - for example, to catch bad pointers

How many bits in an address?

- Intel IA-32, 32 bits, 36 bit page extension mode available
- Intel IA-64, Core i5, 64 bits logical, 48 bits virtual, 36 bits physical
- AMD X86-64, Opteron, 64 bits logical, 48 bits virtual, 48 bits physical

- Sun Solaris, 32 bits on 32-bit processor, 44 bits on 64-bit processor, both modes available on same processor

Architectural variations

- stack grows downward - Intel, MIPS, Sparc
- stack grows upward - HP PA-RISC

How does the system (hardware or software) know when

- the stack and heap collide (if the stack grows downward)
- the stack goes beyond the end of the allowed space

Position-independent code

- when and why do we need it?
- how does it work?

Procedures with a variable number of arguments

- when and why do we need it?
    - C, printf(), scanf(), execl()
- how does it work?
    - C, stdarg.h, see stdarg(3head) on Solaris
- what restrictions are there, and why?
- what dangers are there?
    - loss of type safety, etc.

We left out

- in C, setjmp(), longjmp(), which bypass the normal procedure call-return mechanism
- in C++, exception handlers
- how to hijack an operating system that does not have adequate protection against buffer overflows
- multiple stacks, for threaded programs
- initial stack contents
    - OS- and environment-dependent
    - language-dependent

Chapter VII: Run-Time Environment (CG)

Outline:

- ✓ **Symbol Table**
- ✓ **Storage/Memory Allocation**
- ✓ **Activation Tree**
- ✓ **Parameters Passing**
- ✓ **Static and Dynamic Scope Management**

7. Introduction

By **runtime**, we mean a program in execution. **Runtime environment** is a state of the target machine, which may include software libraries, **environment** variables, etc., to provide services to the processes running in the system. It is a configuration of hardware and software. It includes the CPU type, operating system and any **runtime** engines or system software required by a particular category of applications. Compiler didn't **allocate memory** at compile time. Instead, program asked user, at **runtime**, number of marks to process and **allocated** them **memory** and processed them. This technique is **run time allocation**. Though, **run time allocation** is efficient but program retained **memory** until it exited. A translation needs to relate the static source text of a program to the dynamic actions that must occur at runtime to implement the program. The program consists of names for procedures, identifiers etc., that require mapping with the actual memory location at runtime. A program as a source code is merely a collection of text (code, statements etc.) and to make it alive, it requires actions to be performed on the target machine. A program needs memory resources to execute instructions.

7.1. Symbol Table

A symbol table is a global data structure that can be used in all stages/phases/passes of a compiler. This means that it can be used/accessed from both the **lex** and **yacc** generated components. A symbol table is a major data structure used in a compiler. Associates attribute with identifiers used in a program. For instance, a type attribute is usually associated with each identifier. A symbol table is a necessary component Definition (declaration) of identifiers appears once in a program.Use of identifiers may appear in many places of the program text Identifiers and attributes are entered by the analysis phases. When processing a definition (declaration) of an identifier. In simple languages with only global variables and implicit declarations. The scanner can enter an identifier into a symbol table if it is not already there In block-structured languages with scopes and explicit declarations:

The parser and/or semantic analyzer enter identifiers and corresponding attributes.

Symbol table information is used by the analysis and synthesis phases

To verify that used identifiers have been defined (declared)

To verify that expressions and assignments are semantically correct – type checking

To generate intermediate or target code

Symbol Table Interface

The basic operations defined on a symbol table include:

allocate – to allocate a new empty symbol table

free – to remove all entries and free the storage of a symbol table

insert – to insert a name in a symbol table and return a pointer to its entry

lookup – to search for a name and return a pointer to its entry

set attribute – to associate an attribute with a given entry

get attribute – to get an attribute associated with a given entry

Other operations can be added depending on requirement For example, a delete operation removes a name previously inserted Some identifiers become invisible (out of scope) after exiting a block

Hash Tables

A hash table is an array with index range: 0 to *Table_Size* – 1

Most commonly used data structure to implement symbol tables

Insertion and lookup can be made very fast – *O* (1)

A hash function maps an identifier name into a table index

A hash function, *h*(*name*), should depend solely on *name*

h(*name*) should be computed quickly

h should be uniform and randomize in distributing names

All table indices should be mapped with equal probability.

Similar names should not cluster to the same table index

Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc. Symbol table is used by both the analysis and the synthesis parts of a compiler.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

A symbol table is simply a table which can be either linear or a hash table. It maintains an entry for each name in the following format:

```
<symbol name, type, attribute>
```

For example, if a symbol table has to store information about the following variable declaration:

```
static int interest;
```

then it should store the entry such as:

```
<interest, int, static>
```

The attribute clause contains the entries related to the name.

Implementation

If a compiler is to handle a small amount of data, then the symbol table can be implemented as an unordered list, which is easy to code, but it is only suitable for small tables only. A symbol table can be implemented in one of the following ways:

- Linear (sorted or unsorted) list
- Binary Search Tree
- Hash table

Among all, symbol tables are mostly implemented as hash tables, where the source code symbol itself is treated as a key for the hash function and the return value is the information about the symbol.

Operations

A symbol table, either linear or hash, should provide the following operations.

insert ()

This operation is more frequently used by analysis phase, i.e., the first half of the compiler where tokens are identified and names are stored in the table. This operation is used to add information in the symbol table about unique names occurring in the source code. The format or structure in which the names are stored depends upon the compiler in hand.

An attribute for a symbol in the source code is the information associated with that symbol. This information contains the value, state, scope, and type about the symbol. The insert() function takes the symbol and its attributes as arguments and stores the information in the symbol table.

For example:

```
int a;
```

should be processed by the compiler as:

```
insert (a, int);
```

lookup ()

lookup () operation is used to search a name in the symbol table to determine:

- if the symbol exists in the table.
- if it is declared before it is being used.
- if the name is used in the scope.
- if the symbol is initialized.

- if the symbol declared multiple times.

The format of lookup () function varies according to the programming language. The basic format should match the following:

```
lookup(symbol)
```

This method returns 0 (zero) if the symbol does not exist in the symbol table. If the symbol exists in the symbol table, it returns its attributes stored in the table.

Scope Management

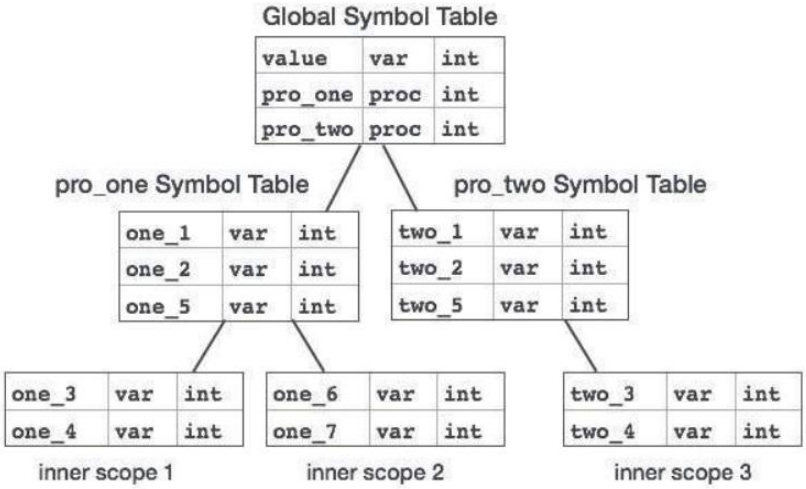
A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.

To determine the scope of a name, symbol tables are arranged in hierarchical structure as shown in the example below:

```
. . .
int value=10;
void pro_one()
  {
  int one_1;
  int one_2;
      {           \
    int one_3;    |_ inner scope 1
    int one_4;    |
    }           /
      int one_5;
      {           \
    int one_6;    |_ inner scope 2
    int one_7;    |
    }           /
  }
  void pro_two()
  {
  int two_1;
  int two_2;
      {           \
    int two_3;    |_ inner scope 3
    int two_4;    |
```

```
    }           /
    int two_5;
  }
. . .
```

The above program can be represented in a hierarchical structure of symbol tables:



The global symbol table contains names for one global variable (int value) and two procedure names, which should be available to all the child nodes shown above. The names mentioned in the pro_one symbol table (and all its child tables) are not available for pro_two symbols and its child tables.

This symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

- first a symbol will be searched in the current scope, i.e. current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found or global symbol table has been searched for the name.

Activation Trees

A program is a sequence of instructions combined into a number of procedures. Instructions in a procedure are executed sequentially. A procedure has a start and an end delimiter and everything inside it is called the body of the procedure. The procedure identifier and the sequence of finite instructions inside it make up the body of the procedure.

The execution of a procedure is called its activation. An activation record contains all the necessary information required to call a procedure. An activation record may contain the following units (depending upon the source language used).

Temporaries	Stores temporary and intermediate values of an expression.
Local Data	Stores local data of the called procedure.

Machine Status	Stores machine status such as Registers, Program Counter etc., before the procedure is called.
Control Link	Stores the address of activation record of the caller procedure.
Access Link	Stores the information of data which is outside the local scope.
Actual Parameters	Stores actual parameters, i.e., parameters which are used to send input to the called procedure.
Return Value	Stores return values.

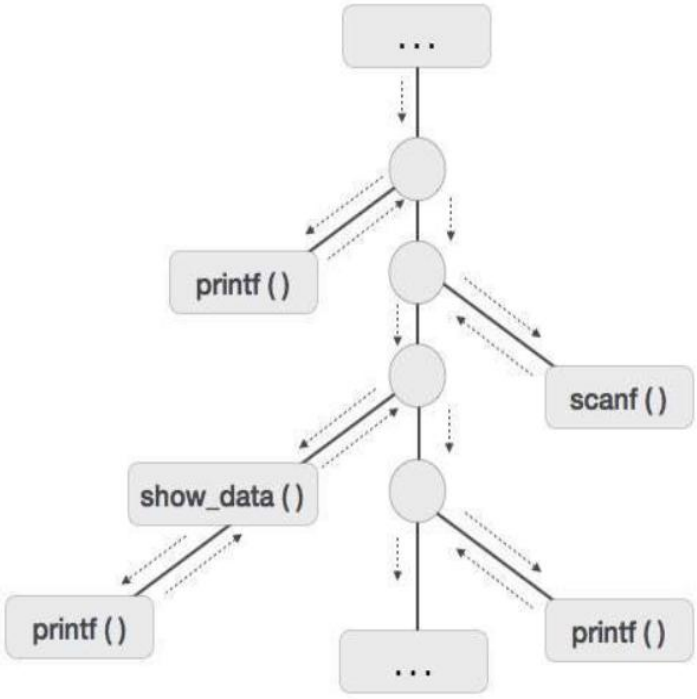
Whenever a procedure is executed, its activation record is stored on the stack, also known as control stack. When a procedure calls another procedure, the execution of the caller is suspended until the called procedure finishes execution. At this time, the activation record of the called procedure is stored on the stack.

We assume that the program control flows in a sequential manner and when a procedure is called, its control is transferred to the called procedure. When a called procedure is executed, it returns the control back to the caller. This type of control flow makes it easier to represent a series of activations in the form of a tree, known as the **activation tree**.

To understand this concept, we take a piece of code as an example:

```
. . .
printf("Enter Your Name: ");
scanf("%s", username);
show_data(username);
printf("Press any key to continue…");
. . .
int show_data(char *user)
  {
  printf("Your name is %s", username);
  return 0;
  }
. . .
```

Below is the activation tree of the code given.



Now we understand that procedures are executed in depth-first manner, thus stack allocation is the best suitable form of storage for procedure activations.

A program consists of procedures, a procedure definition is a declaration that, in its simplest form, associates an identifier (procedure name) with a statement (body of the procedure). Each execution of procedure is referred to as an activation of the procedure. Lifetime of an activation is the sequence of steps present in the execution of the procedure. If 'a' and 'b' be two procedures then their activations will be non-overlapping (when one is called after other) or nested (nested procedures). A procedure is recursive if a new activation begins before an earlier activation of the same procedure has ended. An activation tree shows the way control enters and leaves activations.

Properties of activation trees are: -

- Each node represents an activation of a procedure.
- The root shows the activation of the main function.
- The node for procedure 'x' is the parent of node for procedure 'y' if and only if the control flows from procedure x to procedure y.

Example – Consider the following program of Quicksort

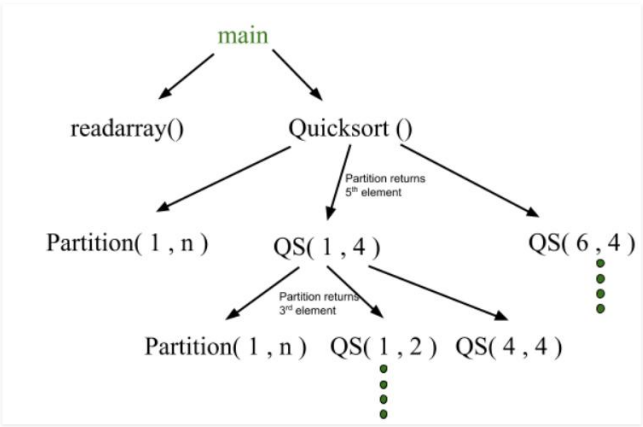
```
main() {

    Int n;
    readarray();
    quicksort(1,n);
}
```

```
quicksort(int m, int n) {

    Int i= partition(m,n);
    quicksort(m,i-1);
    quicksort(i+1,n);
}
```

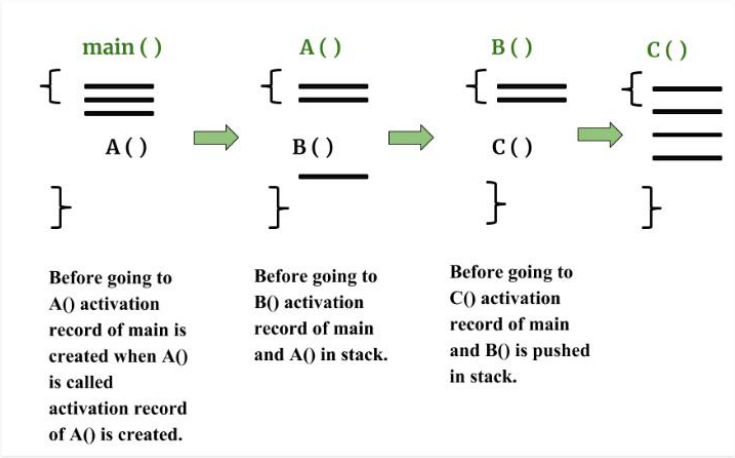
The activation tree for this program will be:



First main function as root then main calls readarray and quicksort. Quicksort in turn calls partition and quicksort again. The flow of control in a program corresponds to the depth first traversal of activation tree which starts at the root.

CONTROL STACK AND ACTIVATION RECORDS

Control stack or runtime stack is used to keep track of the live procedure activations i.e the procedures whose execution have not been completed. A procedure name is pushed on to the stack when it is called (activation begins) and it is popped when it returns (activation ends). Information needed by a single execution of a procedure is managed using an activation record or frame. When a procedure is called, an activation record is pushed into the stack and as soon as the control returns to the caller function the activation record is popped.

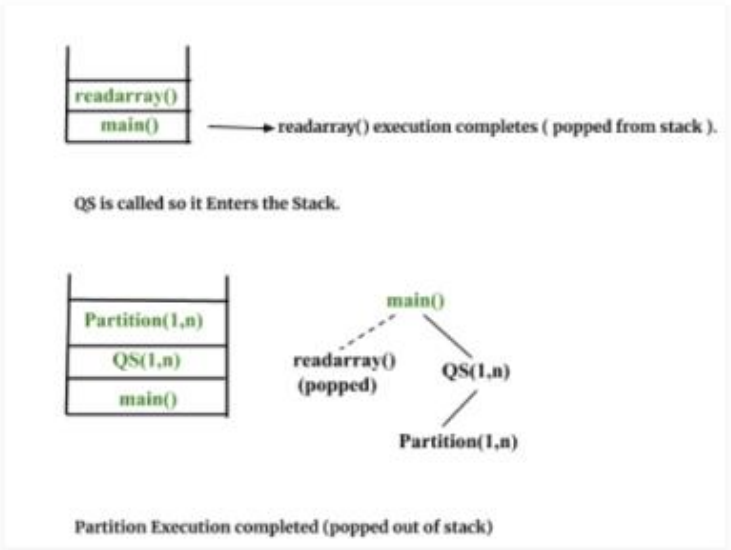


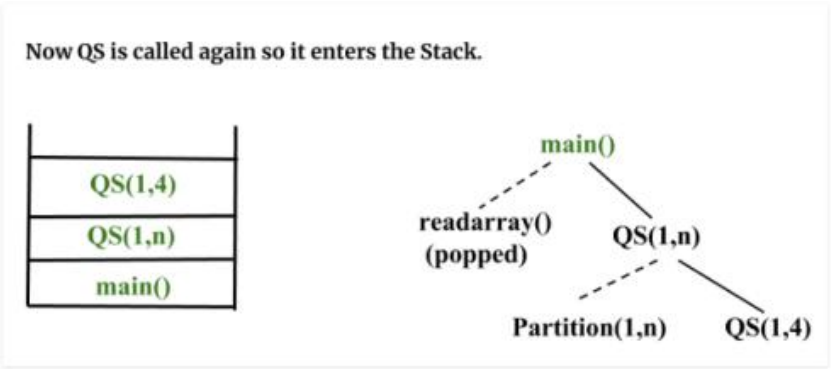
A general activation record consists of the following things:

- **Local variables:** hold the data that is local to the execution of the procedure.
- **Temporary values**: stores the values that arise in the evaluation of an expression.
- **Machine status:** holds the information about status of machine just before the function call.

- **Access link (optional):** refers to non-local data held in other activation records.
- **Control link (optional):** points to activation record of caller.
- **Return value:** used by the called procedure to return a value to calling procedure
- Actual parameters

Control stack for the above quick sort example:

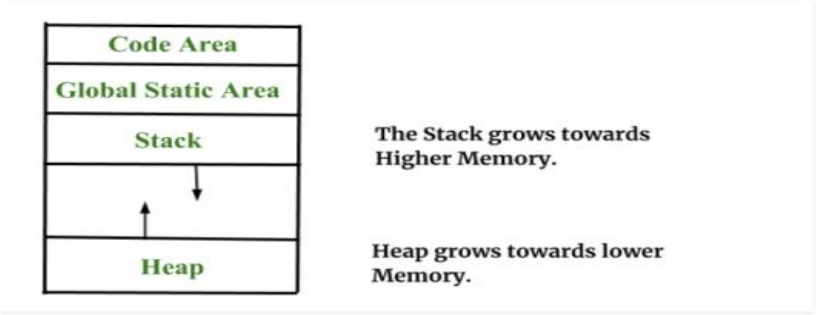




SUBDIVISION OF RUNTIME MEMORY

Runtime storage can be subdividing to hold:

- Target code- the program code, it is static as its size can be determined at compile time
- Static data objects
- Dynamic data objects- heap
- Automatic data objects- stack



Storage Allocation

Runtime environment manages runtime memory requirements for the following entities:

- **Code**: It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

- **Procedures**: Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.
- **Variables**: Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

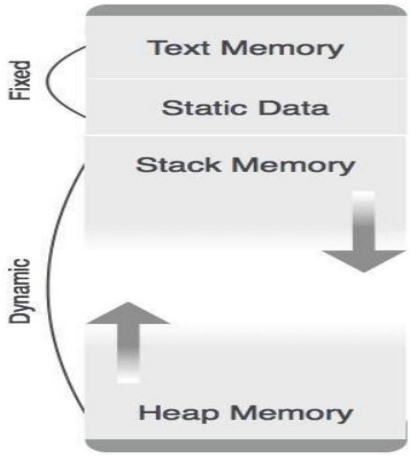
Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed amount of memory in the system.



As shown in the image above, the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program. Both shrink and grow against each other.

STORAGE ALLOCATION TECHNIQUES

I. Static Storage Allocation

- For any program if we create memory at compile time, memory will be created in the static area.
- For any program if we create memory at compile time only, memory is created only once.
- It doesn't support dynamic data structure i.e memory is created at compile time and deallocated after program completion.
- The drawback with static storage allocation is recursion is not supported.
- Another drawback is size of data should be known at compile time

Eg- FORTRAN was designed to permit static storage allocation.

II. Stack Storage Allocation

- Storage is organized as a stack and activation records are pushed and popped as activation begin and end respectively. Locals are contained in activation records so they are bound to fresh storage in each activation.
- Recursion is supported in stack allocation

III. Heap Storage Allocation

- Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user.
- Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.
- Recursion is supported.

Parameter Passing

PARAMETER PASSING

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism.

Basic terminology:

- **R- value:** The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right side of the assignment operator. R-value can always be assigned to some other variable.
- **L-value:** The location of the memory(address) where the expression is stored is known as the l-value of that expression. It always appears on the left side if the assignment operator.

The communication medium among procedures is known as parameter passing. The values of the variables from a calling procedure are transferred to the called procedure by some mechanism. Before moving ahead, first go through some basic terminologies pertaining to the values in a program.

r-value

The value of an expression is called its r-value. The value contained in a single variable also becomes an r-value if it appears on the right-hand side of the assignment operator. r-values can always be assigned to some other variable.

l-value

The location of memory (address) where an expression is stored is known as the l-value of that expression. It always appears at the left-hand side of an assignment operator.

For example:

```
day = 1;

week = day * 7;

month = 1;

year = month * 12;
```

From this example, we understand that constant values like 1, 7, 12, and variables like day, week, month and year, all have r-values. Only variables have l-values as they also represent the memory location assigned to them.

For example:

```
7 = x + y;
```

is an l-value error, as the constant 7 does not represent any memory location.

- **i. Formal Parameter:** Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.
  ii. Actual Parameter: Variables whose values and functions are passed to the called function are called actual parameters. These variables are specified in the function call as arguments.

Different ways of passing the parameters to the procedure

Formal Parameters

Variables that take the information passed by the caller procedure are called formal parameters. These variables are declared in the definition of the called function.

Actual Parameters

Variables whose values or addresses are being passed to the called procedure are called actual parameters. These variables are specified in the function call as arguments.

Example:

```
fun_one()
{
  int actual_parameter = 10;
  call fun_two(int actual_parameter);
}
  fun_two(int formal_parameter)
{
  print formal_parameter;
}
```

Formal parameters hold the information of the actual parameter, depending upon the parameter passing technique used. It may be a value or an address.

Pass/Call by Value

In pass by value mechanism, the calling procedure passes the r-value of actual parameters and the compiler puts that into the called procedure's activation record. Formal parameters then hold the values passed by the calling procedure. If the values held by the formal parameters are changed, it should have no impact on the actual parameters.

In call by value the calling procedure pass the r-value of the actual parameters and the compiler puts that into called procedure's activation record. Formal parameters hold the values passed by the calling procedure, thus any changes made in the formal parameters does not affect the actual parameters.

Pass by Reference

In pass by reference mechanism, the l-value of the actual parameter is copied to the activation record of the called procedure. This way, the called procedure now has the address (memory location) of the actual parameter and the formal parameter refers to the same memory location.

Therefore, if the value pointed by the formal parameter is changed, the impact should be seen on the actual parameter as they should also point to the same value. **Call by Reference** In call by reference the formal and actual parameters refers to same memory location. The l-value of actual parameters is copied to the activation record of the called function. Thus, the called function has the address of the actual parameters. If the actual parameters do not have a l-value (e.g.- i+3) then it is evaluated in a new temporary location and the address of the location is passed. Any changes made in the formal parameter is reflected in the actual parameters (because changes are made at the address).

Pass by Copy-restore

This parameter passing mechanism works similar to 'pass-by-reference' except that the changes to actual parameters are made when the called procedure ends. Upon function call, the values of actual parameters are copied in the activation record of the called procedure. Formal parameters if manipulated have no real-time effect on actual parameters (as l-values are passed), but when the called procedure ends, the l-values of formal parameters are copied to the l-values of actual parameters. **Call by Copy Restore,** in call by copy restore compiler copies the value in formal parameters when the procedure is called and copy them back in actual parameters when control returns to the called function. The r-values are passed and on return r-value of formals are copied into l-value of actual.

Example:

```
int y;
calling_procedure()
{
  y = 10;
  copy_restore(y); //l-value of y is passed
  printf y; //prints 99
}
copy_restore(int x)
{
  x = 99; // y still has value 10 (unaffected)
  y = 0; // y is now 0
}
```

When this function ends, the l-value of formal parameter x is copied to the actual parameter y. Even if the value of y is changed before the procedure ends, the l-value of x is copied to the l-value of y making it behave like call by reference.

Pass by Name

Languages like Algol provide a new kind of parameter passing mechanism that works like preprocessor in C language. In pass by name mechanism, the name of the procedure being called is replaced by its actual body. Pass-by-name textually substitutes the argument expressions in a procedure call for the corresponding parameters in the body of the procedure so that it can now work on actual parameters, much like pass-by-reference.

Call by Name

In call by name the actual parameters are substituted for formals in all the places formals occur in the procedure. It is also referred as lazy evaluation because evaluation is done on parameters only when needed.

7.3. Representing Scope Information

Every name possesses a region of validity within the source program, called the "scope" of that name. The rules governing the scope of names in a block-structured language are as follows:

1. A name declared within a block *B* is valid only within *B*.
2. If block *B*1 is nested within *B*2, then any name that is valid for *B*2 is also valid for *B*1, unless the identifier for that name is re-declared in *B*1.

These scope rules require a more complicated symbol table organization than simply a list of associations between names and attributes. One technique that can be used is to keep multiple symbol tables, one for each active block, such as the block that the compiler is currently in. Each table is a list of names and their associated attributes, and the tables are organized into a stack. Whenever a new block is entered, a new empty table is pushed onto the stack for holding the names that are declared as local to this block. And when a declaration is compiled, the table on the stack is searched for a name. If the name is not found, then the new name is inserted. When a reference to a name is translated, each table is searched, starting from the top table on the stack, ensuring compliance with static scope rules. For example, consider the following program structure. The symbol table organization will be as shown in Figure 1.

```
Program                                  .
main                                     .
Var x,y : integer :                      end
Procedure P :                            begin :
Var x,a : boolean;                       end
Procedure q                              begin :
Var x,y,z : real;                        end
Begin
```

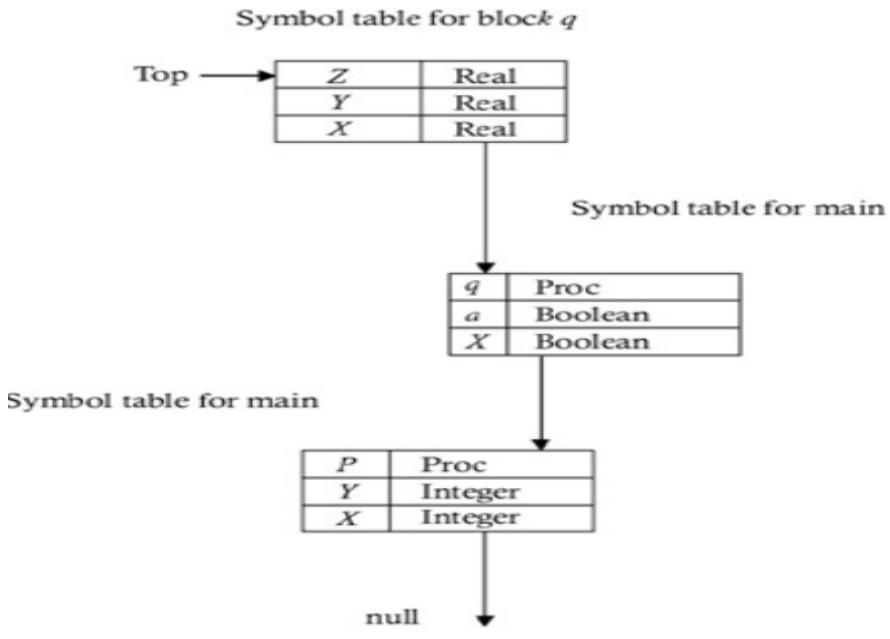


Figure 1: Symbol table organization that complies with static scope information rules.

Another technique can be used to represent scope information in the symbol table. We store the nesting depth of each procedure block in the symbol table and use the [procedure name, nesting depth] pair as the key to accessing the information from the table. A nesting depth of a procedure is a number that is obtained by starting with a value of one for the main and adding one to it every time we go from an enclosing to an enclosed procedure. This number is basically a count of how many procedures are there in the referencing environment of the procedure.

Chapter VIII: Introduction to Code Optimization

8. Introduction to Code Optimization

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e., CPU, Memory) and deliver high speed. In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e., CPU, Memory) so that faster-running machine code will result.

A code optimizing process should meet the following objectives:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.
- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

When to Optimize? Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

Efforts for an optimized code can be made at various levels of compiling the process.

- At the beginning, users can change/rearrange the code or use better algorithms to write the code.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Types of Code Optimization: Optimization can be categorized broadly into two types : machine independent and machine dependent.

1. Machine Independent Optimization – This code optimization phase the compiler attempts to improve the intermediate code to get a better target code as the output. The part of the

intermediate code which is transformed here does not involve any CPU registers or absolute memory locations.

For example:

```
do
{
  item = 10;
  value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

2. Machine-dependent Optimization: Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of the memory hierarchy.

Basic Blocks: Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

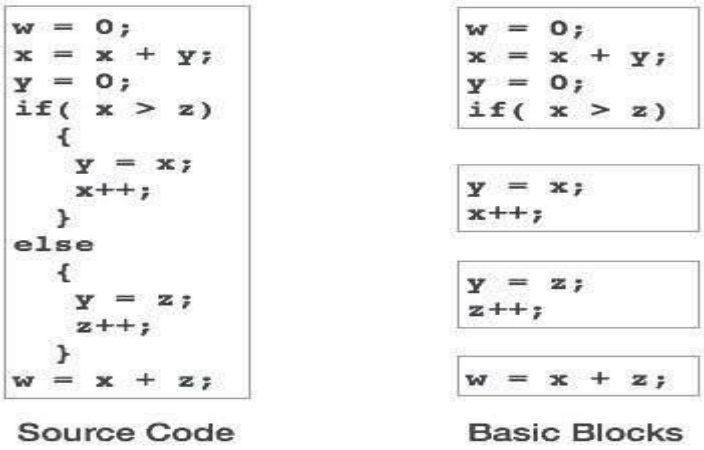
We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
    - First statement of a program.

- Statements that are target of any branch (conditional/unconditional).
- Statements that follow any branch statement.

- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



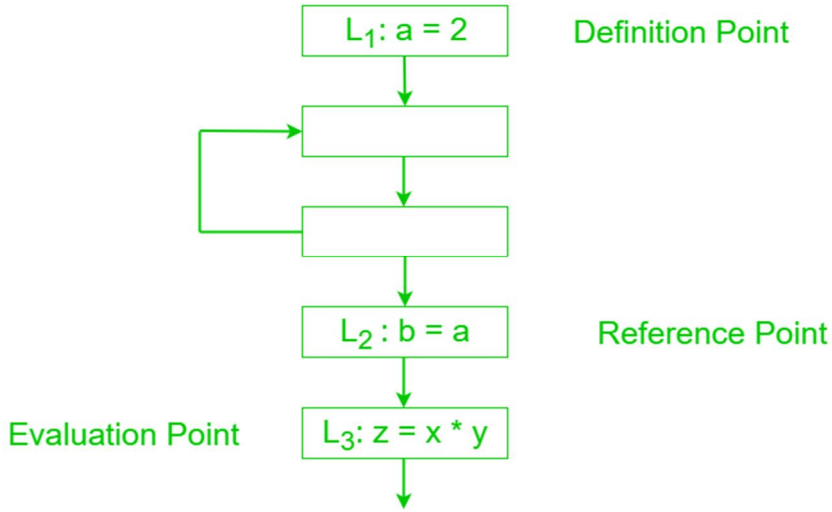
Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Data flow analysis in Compiler

It is the analysis of flow of data in control flow graph, i.e., the analysis that determines the information regarding the definition and use of data in program. With the help of this analysis optimization can be done. In general, its process in which values are computed using data flow analysis. The data flow property represents information which can be used for optimization.

Basic Terminologies –

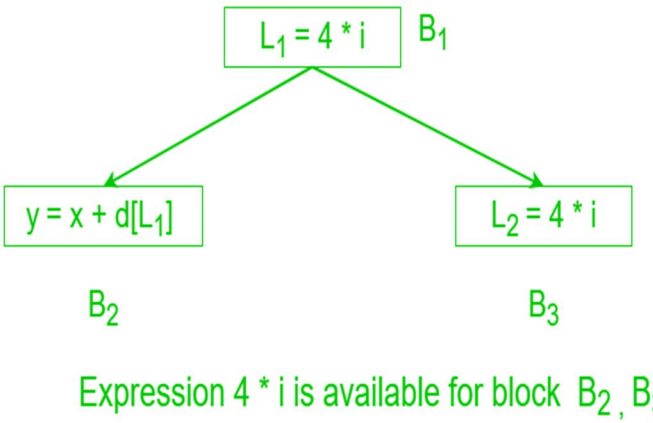
- **Definition Point:** a point in a program containing some definition.
- **Reference Point:** a point in a program containing a reference to a data item.
- **Evaluation Point:** a point in a program containing evaluation of expression.



Data Flow Properties –

- **Available Expression –** A expression is said to be available at a program point x iff along paths its reaching to x. A Expression is available at its evaluation point. A expression a+b is said to be available if none of the operands gets modified before their use.

Example –

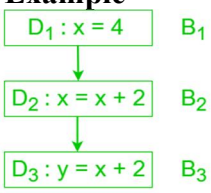


Expression 4 * i is available for block B_2 , B_3

Advantage –
It is used to eliminate common sub expressions.

- **Reaching Definition –** A definition D is reaching a point x if there is path from D to x in which D is not killed, i.e., not redefined.

Example –

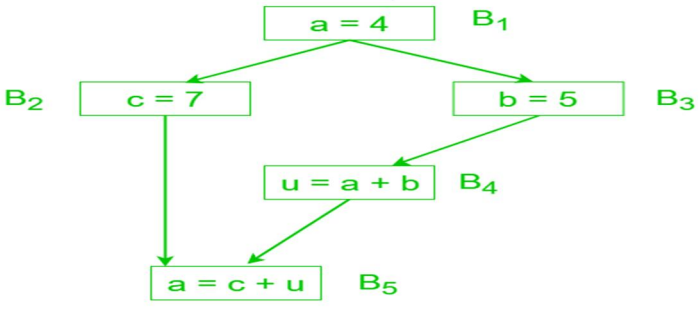


D_1 is reaching definition for B_2 but not for B_3 since it is killed by D_2

Advantage –
It is used in constant and variable propagation.

- **Live variable –** A variable is said to be live at some point p if from p to end the variable is used before it is redefined else it becomes dead.

Example –



a is live at block B_1 , B_3 , B_4 but killed at B_5

Advantage –

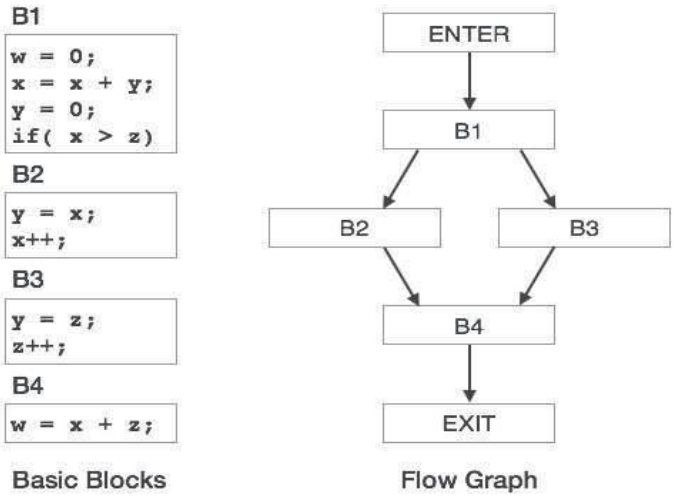
1. It is useful for register allocation.
2. It is used in dead code elimination.

Busy Expression – An expression is busy along a path iff its evaluation exists along that path and none of its operand definition exists before its evaluation along the path.

Advantage –
It is used for performing code movement optimization.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.



Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Dead-code Elimination

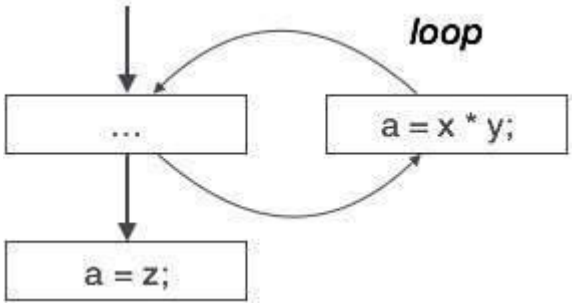
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

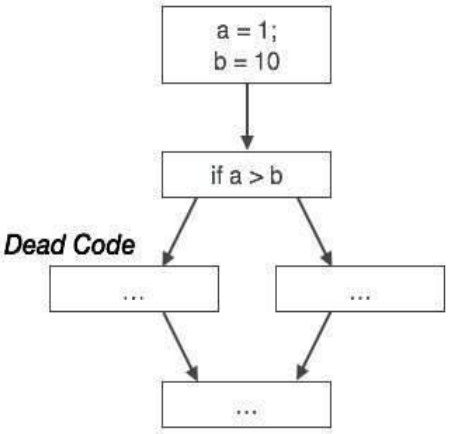
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



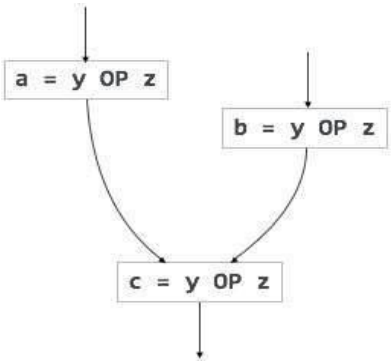
The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’. Let us assume that the value assigned to ‘a’ is never used inside the loop.Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.



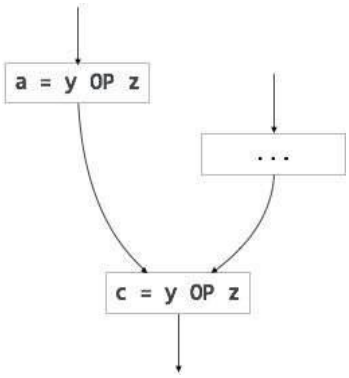
Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands, whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,



[redundant expression]



[partially redundant expression]

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

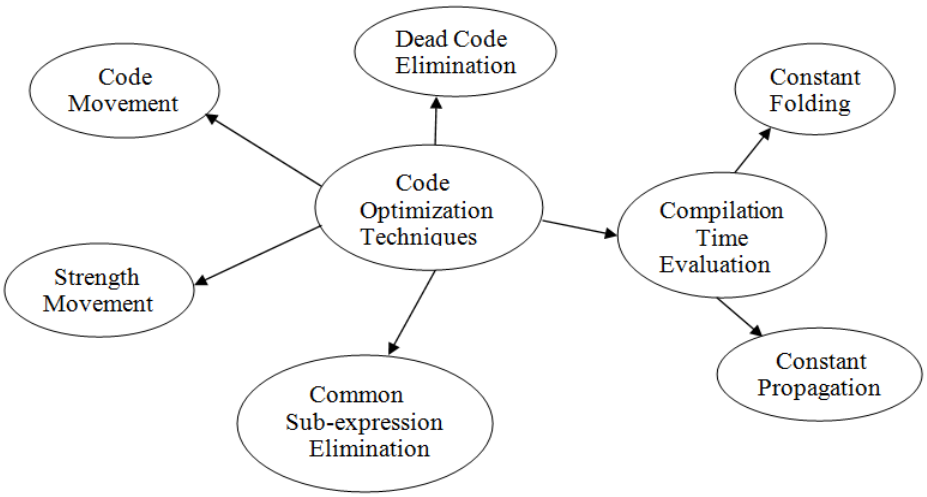
```
If (condition)
{   a = y OP z;}
else{
   ...}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{   ...
   tmp = y OP z;
   a = tmp;
   ...}
else{
   ...
   tmp = y OP z;}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

Code Optimization is done in the following different ways :



Question: ***Briefly define and explain with example for the above compilation techniques?***

1. **Compile Time Evaluation:**
   (i) A = 2*(22.0/7.0)*r
   Perform 2*(22.0/7.0)*r at compile time.
   (ii) x = 12.4
   y = x/2.3
   Evaluate x/2.3 as 12.4/2.3 at compile time.
2. **Variable Propagation:**
   //Before Optimization
   c = a * b
   x = a
   till
   d = x * b + 4
   //After Optimization
   c = a * b
   x = a
   till
   d = a * b + 4
   Hence, after variable propagation, a*b and x*b will be identified as common sub-expression.
3. **Dead code elimination:** Variable propagation often leads to making assignment statement into dead code
   c = a * b
   x = a
   till
   d = a * b + 4
   //After elimination:
   c = a * b
   till
   d = a * b + 4
4. **Code Motion:**
   Reduce the evaluation frequency of expression.
   • Bring loop invariant statements out of the loop.

```
a = 200;
 while(a>0)
 {
    b = x + y;
```

```
    if (a % b == 0}
    printf ("%d", a);
   }

//This code can be further optimized as
a = 200;
b = x + y;
while(a>0)
 {
    if (a % b == 0}
    printf ("%d", a);
   }
```

5. **Induction Variable and Strength Reduction:** An induction variable is used in the loop for the following kind of assignment i = i + constant. Strength reduction means replacing the high strength operator by the low strength.

```
i = 1;
while (i<10)
{
   y = i * 4;
}
 //After Reduction
i = 1
t = 4
{
  while(t<40)
  y = t;
  t = t + 4;
}
```

Peephole Optimization Techniques

In compiler theory, **peephole optimization** is a kind of optimization performed over a very small set of instructions in a segment of generated code. The set is called a "peephole" or a "window". It works by recognizing sets of instructions that can be replaced by shorter or faster sets of instructions. A statement-by-statement code-generation strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying "optimizing" transformations to the target program.

A simple but effective technique for improving the target code is peephole optimization, a method for trying to improve the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

We shall give the following examples of program transformations that are characteristic of peephole optimizations:

- Redundant-instructions elimination

- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable Code
- Elimination of redundant loads and stores
- Elimination of multiple jumps
- Elimination of unreachable code
- Algebraic Simplifications
- Reducing strength
- Use of machine idioms

Common techniques applied in peephole optimization:

- Null sequences – Delete useless operations.
- Combine operations – Replace several operations with one equivalent.
- Algebraic laws – Use algebraic laws to simplify or reorder instructions.
- Special case instructions – Use instructions designed for special operand cases.
- Address mode operations – Use address modes to simplify code.

There can be other types of peephole optimizations.

Eliminating Redundant Loads and Stores

If the target code contains the instruction sequence:

1. MOV R, a
2. MOV a, R

we can delete the second instruction if it an unlabeled instruction. This is because the first instruction ensures that the value of *a* is already in the register *R*. If it is labeled, there is no guarantee that step 1 will always be executed before step 2.

Eliminating Multiple Jumps

If we have jumped to other jumps, then the unnecessary jumps can be eliminated in either intermediate code or the target code. If we have a jump sequence:

```
    goto L1
    ...
L1: goto L2
```

then this can be replaced by:

```
    goto L2
    ...
L1: goto L2
```

If there are now no jumps to $L1$, then it may be possible to eliminate the statement, provided it is preceded by an unconditional jump. Similarly, the sequence:

```
    if a < b goto L1
    ...
L1: goto L2
```

can be replaced by:

```
    if a < b goto L2
    ...
L1: goto L2
```

Eliminating Unreachable Code

An unlabeled instruction that immediately follows an unconditional jump can possibly be removed, and this operation can be repeated in order to eliminate a sequence of instructions. For debugging purposes, a large program may have within it certain segments that are executed only if a debug variable is one. For example, the source code may be:

```
#Define debug 0
...
if (debug)
    {
      print debugging information
    }
```

This if statement is translated in the intermediate code to:

goto $L2$

$L1$: print debugging information

$L2$:

One of the optimizations is to replace the pair:

if debug = 1 goto $L1$

goto $L2$

within the statements with a single conditional go to a statement by negating the condition and changing its target, as shown below:

Print debugging information

$L2$:

Since debug is a constant zero by constant propagation, this code will become:

if $0 \neq 1$ goto $L2$

Print debugging information

*L*2 :

Since $0 \neq 1$ is always true this will become:

goto *L*2

Print debugging information

*L*2 :

Therefore, the statements that print the debugging information are unreachable and can be eliminated, one at a time.

Algebraic Simplifications

If statements like: are generated in the code, they can be eliminated, because zero is an additive identity, and one is a multiplicative identity.

Reducing Strength

Certain machine instructions are considered to be cheaper than others. Hence, if we replace expensive operations with equivalent cheaper ones on the target machine, then the efficiency will be better. For example, x^2 is invariably cheaper to implement as $x * x$ than as a call to an exponentiation routine. Similarly, fixed-point multiplication or division by a power of two is cheaper to implement as a shift.

Using Machine Idioms

The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. Using these modes can greatly improve the quality of the code when pushing or popping a stack. These modes can also be used for implementing statements like $a = a + 1$.

Code generation – Simple Code generator

In computing, code generation is the process by which a compiler's code generator converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine. Sophisticated compilers typically perform multiple passes over various intermediate forms. The code generator is part of an extra security feature called login approvals. If you turn on login approvals, you'll be asked for a special security code each time you try to log into your Workplace account from a new device.

Register Allocation

Register Allocation and Assignment

1 Global Register Allocation

2 Usage Counts

3 Register Assignment for Outer Loops

4 Register Allocation by Graph Coloring

Instructions involving only register operands are faster than those involving memory operands. On modern machines, processor speeds are often an order of magnitude or faster than memory speeds. Therefore, efficient utilization of registers is vitally important in generating good code. This section presents various strategies for deciding at each point in a program what values should reside in registers (register allocation) and in which register each value should reside (register assignment).

One approach to register allocation and assignment is to assign specific values in the target program to certain registers. For example, we could decide to assign base addresses to one group of registers, arithmetic computations to another, the top of the stack to a fixed register, and so on.

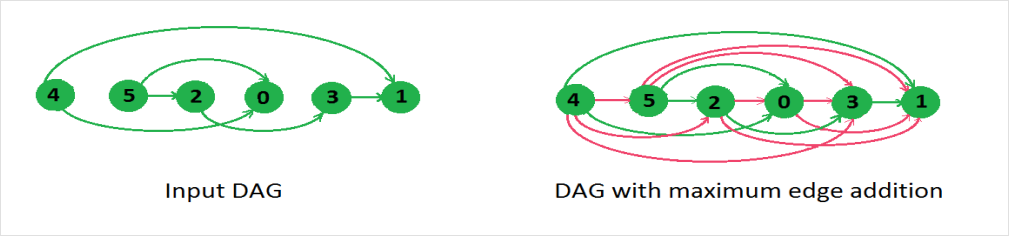
This approach has the advantage that it simplifies the design of a code generator. Its disadvantage is that applied too strictly, it uses registers inefficiently; certain registers may go unused over substantial portions of code, while unnecessary loads and stores are generated into the other registers. Nevertheless, it is reasonable in most computing environments to reserve a few registers for base registers, stack pointers, and the like, and to allow the remaining registers to be used by the code generator as it sees fit.

DAG Representation

A directed acyclic graph (DAG) is a directed graph that contains no cycles. A rooted tree is a special kind of DAG and a DAG is a special kind of directed graph. For example, a DAG may be used to represent common subexpressions in an optimizing compiler.

Maximum edges that can be added to DAG so that remains DAG

A DAG is given to us, we need to find a maximum number of edges that can be added to this DAG, after which new graph still remain a DAG that means the reformed graph should have the maximal number of edges, adding even single edge will create a cycle in the graph.



The Output for above example should be following edges in any order.

4-2, 4-5, 4-3, 5-3, 5-1, 2-0, 2-1, 0-3, 0-1

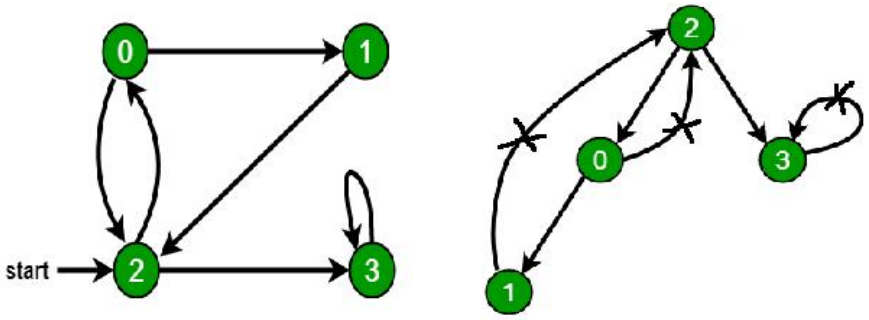
As shown in the above example, we have added all the edges in one direction only to save ourselves from making a cycle. This is the trick to solve this question. We sort all our nodes in topological order and create edges from a node to all nodes to the right if not there already. How can we say that it is not possible to add any more edge? the reason is we have added all possible edges from left to right and if we want to add more edge we need to make that from right to left, but adding edge from right to left we surely create a cycle because its counterpart left to right edge is already been added to graph and creating cycle is not what we needed. So solution proceeds as follows, we consider the nodes in topological order and if any edge is not there from left to right, we will create it. Below is the solution, we have printed all the edges that can be added to given DAG without making any cycle.

Detect Cycle in a Directed Graph

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles 0->2->0, 0->1->2->0 and 3->3, so your function must return true.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with a cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for a cycle in individual trees by checking back edges.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. We have used recStack[] array to keep track of vertices in the recursion stack.

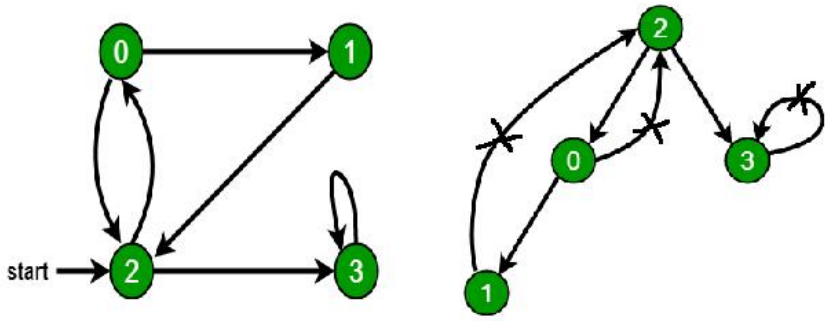
Time Complexity of this method is same as time complexity of DFS traversal which is O(V+E).

Depth First Search or DFS for a Graph

Depth First Traversal (or Search) for a graph is similar to Depth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a boolean visited array.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited

vertices, then 2 will be processed again and it will become a non-terminating process. A Depth First Traversal of the following graph is 2, 0, 1, 3.



See this post

for all applications of Depth-First Traversal. Following are implementations of simple Depth-First Traversal. The C++ implementation uses an adjacency list representation of graphs. STL's list container is used to store lists of adjacent nodes.

How to handle disconnected graph?

The above code traverses only the vertices reachable from a given source vertex. All the vertices may not be reachable from a given vertex (example Disconnected graph). To do complete DFS traversal of such graphs, we must call DFSUtil() for every vertex. Also, before calling DFSUtil(), we should check if it is already printed by some other call of DFSUtil(). Following implementation does the complete graph traversal even if the nodes are unreachable.

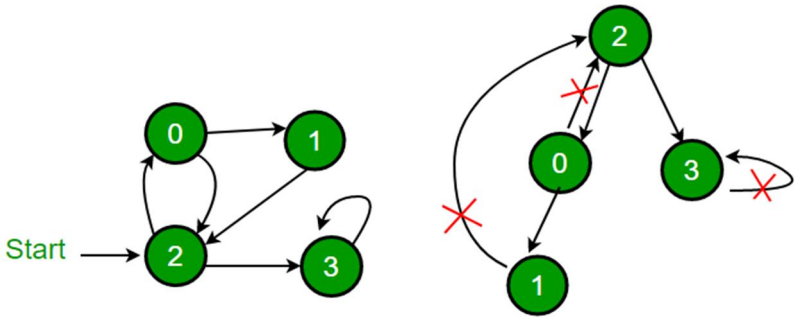
Time Complexity: O(V+E) where V is number of vertices in the graph and E is number of edges in the graph.

Detect Cycle in a directed graph using colors

Given a directed graph, check whether the graph contains a cycle or not. Your function should return true if the given graph contains at least one cycle, else return false. For example, the following graph contains three cycles 0->2->0, 0->1->2->0 and 3->3, so your function must return true.

Solution

Depth First Traversal can be used to detect cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (selfloop) or one of its ancestor in the tree produced by DFS. In the following graph, there are 3 back edges, marked with cross sign. We can observe that these 3 back edges indicate 3 cycles present in the graph.



For a disconnected graph, we get the DFS forest as output. To detect cycle, we can check for cycle in individual trees by checking back edges.

The idea is to do DFS of given graph and while doing traversal, assign one of the below three colors to every vertex.

WHITE : Vertex is not processed yet. Initially all vertices are WHITE.

GRAY : Vertex is being processed (DFS for this vertex has started, but not finished which means that all descendants (ind DFS tree) of this vertex are not processed yet (or this vertex is in function call stack)

BLACK : Vertex and all its descendants are processed.

While doing DFS, if we encounter an edge from current vertex to a GRAY vertex, then this edge is back edge and hence there is a cycle.