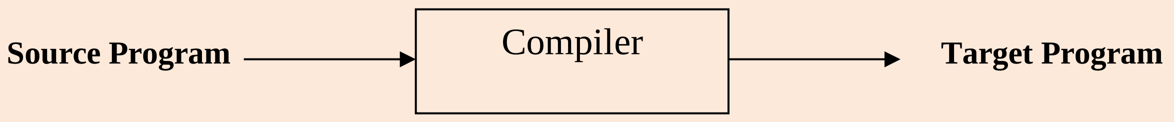
UNIT – 1

Introduction Language Processing, Structure of a compiler the evaluation of Programming language, The Science of building a Compiler application of Compiler Technology. Programming Language Basics. Lexical Analysis-: The role of lexical analysis buffing, specification of tokens. Recognitions of tokens the lexical analyzer generator lexical

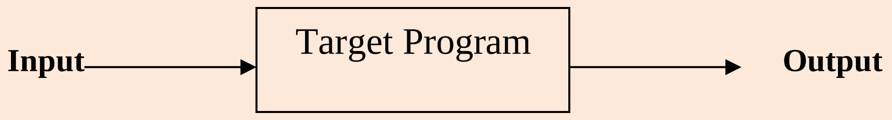
1.1 Language Processors or Language Translators:

The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. Before a program can be run, it must be translated into a form, which can be executed by a computer. The software systems that do this translation is called Compiler. Any software which converts one programming language into another programming language is called as Language Translator.

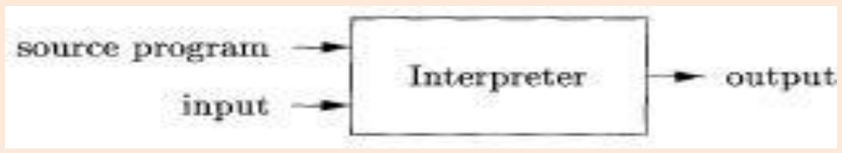
Compiler: - A compiler is a program which takes a program written in a source language and translates it into an equivalent program in a target language. An important role of the compiler is to report any errors in the source program that it detects during the translation process.



If the Target Program is an executable machine language program, it can then be called by the user to process input and produce output.



Interpreter:-An interpreter is another common kind of language processor. Instead of producing a target program it directly executes the operations specified in the source program on inputs supplied by the user.

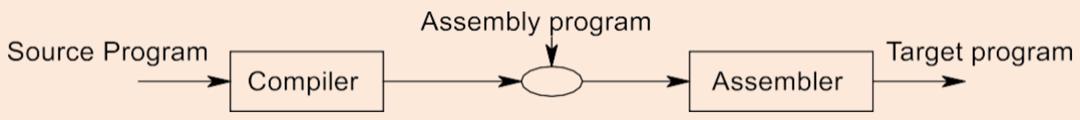


❖ ***<u>Note:</u>***

- The machine language target program produced by compiler is much faster than an interpreter at mapping input to output.
- An interpreter gives better error diagnostics than compiler, because it executes the source program statement-by-statement.

Preprocessor: - A preprocessor is a program that processes its input data (i.e. source program) to produce output (i.e. modified source program) that is used as input to the compiler. The preprocessor expand shorthand's, called macros, into source language statements.

Assembler: If the source program is assembly language and the target language is machine language then the translator is called an assembler.



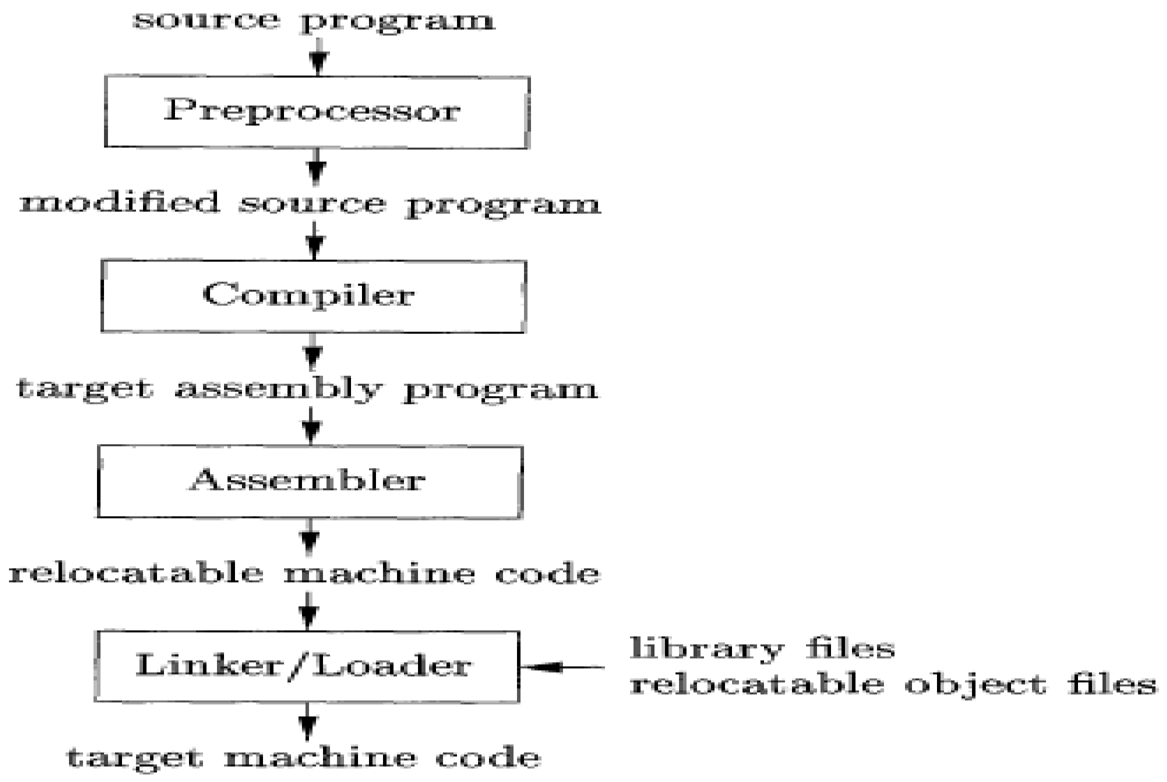
The assembly language is then processed by a program called on Assembler that produces relocatable machine code as its output.

Linker: Resolves external memory addresses, where the code in one file may refer to a location in another file. It links the relocatable object file with the system wide startup object file and makes an executable file. **Reloacatable machine** code means that it can be loaded starting at any location ***L*** in memory; i.e., if L is added to all addresses in the code, then all references will be correct.

Loader: The loader puts together all of the executable object files into memory for execution. It loads the executable code into the memory for execution. The process of loading takes relocatable machine code, alter the relocatable addresses and place the altered instructions and data in memory at the proper locations.

1.2 Language Processing System

The preprocessor may also expand shorthand's, called macros, into source language statements. The modified source program is then fed to a compiler. The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug. The assembly language is then processed by a program called an *assembler* that produces relocatable machine code as its output. Reloacatable machine code means that it can be loaded starting at any location ***L*** in memory; i.e., if L is added to all addresses in the code, then all references will be correct.



A language-processing system

Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine. The *linker* resolves external memory addresses, where the code in one file may refer to a location in another file. The *loader* then puts together all of the executable object files into memory for execution.

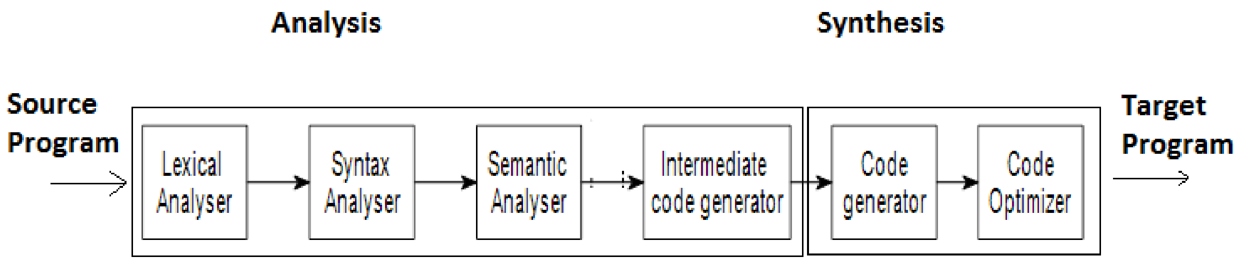
1.3 The Structure of a Compiler

A compiler maps a source program into semantically equivalent target program. There are two parts to this mapping **Analysis** and **Synthesis.**

The compilation process of a compiler can be subdivided into main parts. They are

1. Analysis and 2. Synthesis

Analysis phase: - The analysis part is often called the **front end** of the compiler. In analysis phase, an intermediate representation is created from the given source program. Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and Intermediate Code Generator are the parts of this phase. It breaks up the source program and checks whether the source program is either syntactically or semantically correct. It provides informative error messages, so that the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.



Synthesis phase: - The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. Code Generator and Code Optimizer are the parts of this phase. The synthesis part is the **back end** of the compiler. The backend deals with machine-specific details like allocation of registers, number of allowable operators and so on.

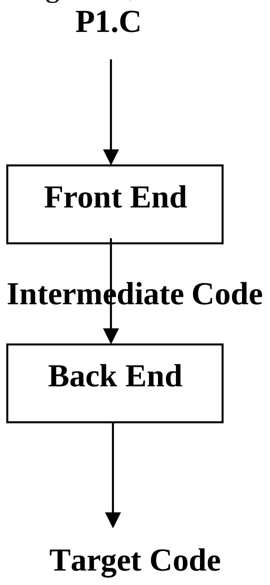


Fig: Front end and Back end of Compiler

The above fig. shows the two stage design approach of a compiler using C language source File as input. The main advantages of having this two stage design are as follows:

i. The compiler can be extended to support an additional processor by adding the required back end of the compiler. The existing front end is completely re-used in this case. This is shown in fig below**.**

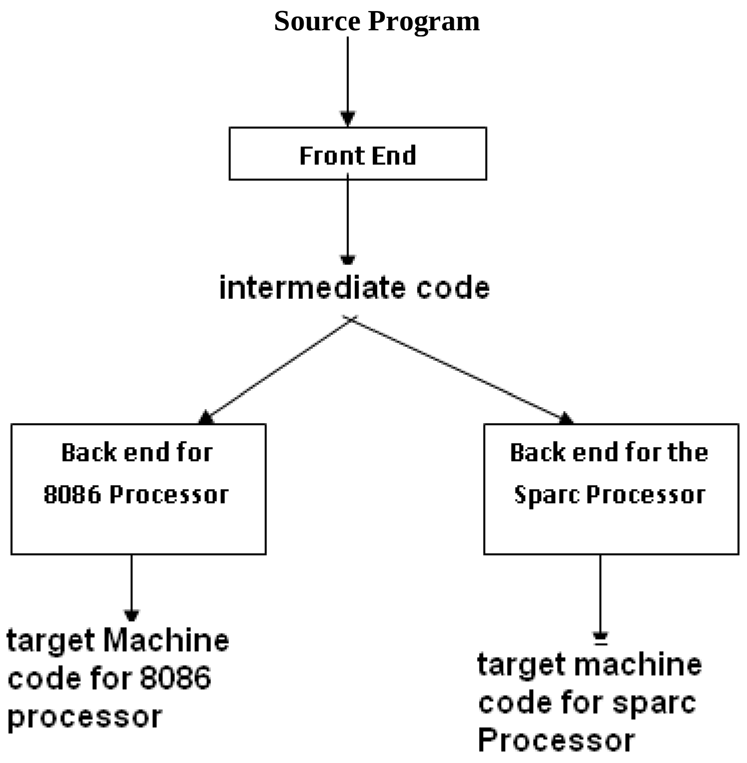

Source Program
Front End
intermediate code
Back end for 8086 Processor
Back end for the Sparc Processor
target Machine code for 8086 processor
target machine code for sparc Processor

Fig. supporting an additional processor by adding back end.

ii. The compiler can be easily extended to support an additional input source language by adding required front end. In this case, the back end is completely re-used. This is shown in fig below.

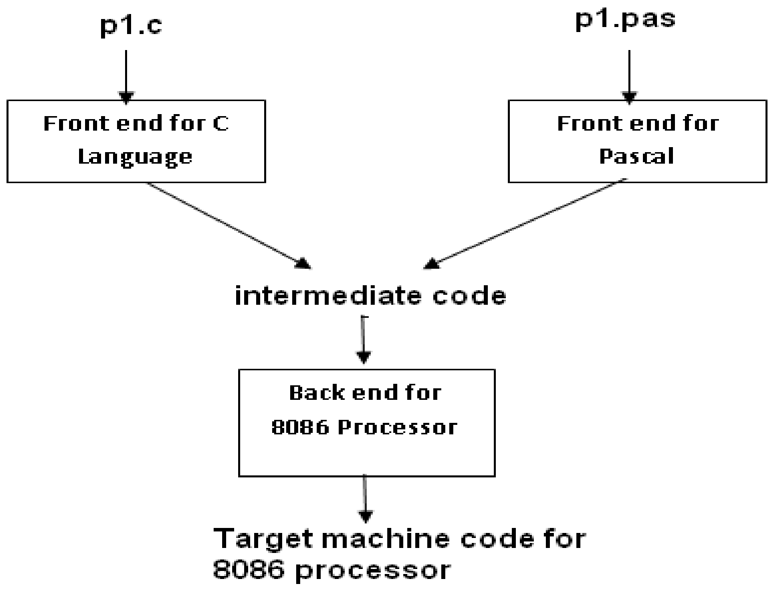


Fig. supporting an additional Languages by adding front end.

1.4 Phases of a Compiler

The compilation process operates as a sequence of phases. Each phase transforms the source program from one representation into another representation. A compiler is decomposed into phasesas shown in Fig. The symbol table, which stores information about the entire source program, is used by all phases of the compiler. During Compilation process, each phase can encounter errors. The error handler is a data structure that reports the presence of errors clearly and accurately. It specifies how the errors can be recovered quickly to detect subsequent errors.

Lexical Analyzer: - The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the source program and groups the characters into meaningful sequences called lexemes (i.e tokens). For each lexeme, the lexical analyzer produces output in the form

<token-name, attribute-value>

That is passed to the next phase. In token the first component token name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol-table entry is needed for semantic analysis and code generator.

Example: suppose a source program contains the assignment statement

Position = initial + rate * 60

The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

1. **Position** is a lexeme that would be mapped into a token **<id,** 1>, where **id** is an abstract symbol standing for *identifier* and 1 points to the symbol table entry for **Position**. The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

2. The assignment symbol = is a lexeme that is mapped into the token < = >. Since this token needs no attribute-value, we have omitted the second component. We could have used any abstract symbol such as **assign** for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

3. initial is a lexeme that is mapped into the token **<id,** 2>, where 2 points to the symbol-table entry for **initial .**

4. + is a lexeme that is mapped into the token < + >.

5. **rate** is a lexeme that is mapped into the token **<id,** 3>, where 3 points to the symbol-table entry for **rate** .

6. * is a lexeme that is mapped into the token <*>.

7. 60 is a lexeme that is mapped into the token <60>.

After lexical analysis the assignment statement can be represented as a sequence of tokens as follows:

< i d , l > < = > <id, 2> <+> <id, 3> <*> <60>

Blanks separating the lexemes would be discarded by the lexical analyzer.

Syntax Analyzer: - The second phase of the compiler is syntax analysis or parsing. The parser uses the tokens produced by the lexical analyzer to create a syntax tree. In syntax tree, each interior node represents an operation and the children of the node represent the arguments of the operation.

A syntax tree for the token stream is shown as the output of syntax analyzer.

The tree below shows the order in which the operations in assignment are to be performed.

Position=initial + rate * 60

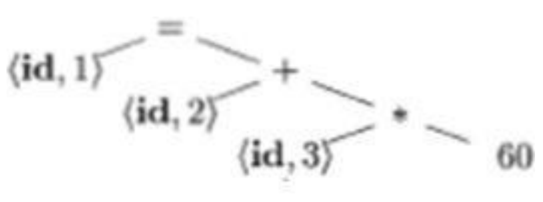
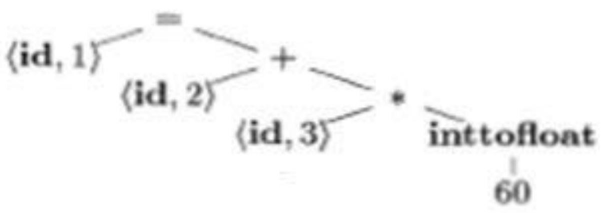


Fig: Syntax Tree

Semantic Analyzer: - The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic errors. It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. Some language allows type conversion called coercion. In our assignment statement the type checker converts integer value into floating point number. The output of the semantic analyzer has an extra node for operator ***inttofloat*** which explicitly converts its integer argument into a floating point number.



***Intermediate Code Generator*: -** In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms. Syntax trees are a form of intermediate representation; they are commonly used during syntax and semantic analysis. After syntax and semantic analysis of the source program, many compilers generate an intermediate representation. This intermediate representation should have two important properties:

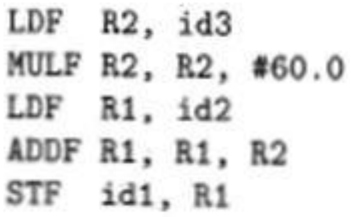
- It should be easy to produce and
- It should be easy to translate into the target code.

One form of intermediate code is three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction.
The output of Intermediate code generator for our assignment statement consists of Three-address code sequence.
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimization: - The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. The target code generated must be executed faster and must consume less power.
Example***:***

t1 = id3 * 60.0

id1 = id2 + t1

Code Generator: - The code generator takes intermediate representation of the source program and coverts into the target code. If the target language is machine code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.



```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

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

Error Handler: - Error handler should report the presence of an error. It must report the place in the source program where an error is detected. Common programming errors can occur at many different levels.

- Lexical errors include misspellings of identifiers, keywords, or operators.
- Syntax errors include misplaced semicolons or extra or missing braces.
- Semantic errors include type mismatches between operators and operands.
- Logical errors can be anything from incorrect reasoning on the part of the programmer to the use in a C program of the assignment operator = instead of the comparison operator ==.

The main goal of error handler is

1. Report the presence of errors clearly and accurately.
2. Recover from each error quickly enough to detect subsequent errors.
3. Add minimal overhead to the processing of correct programs.

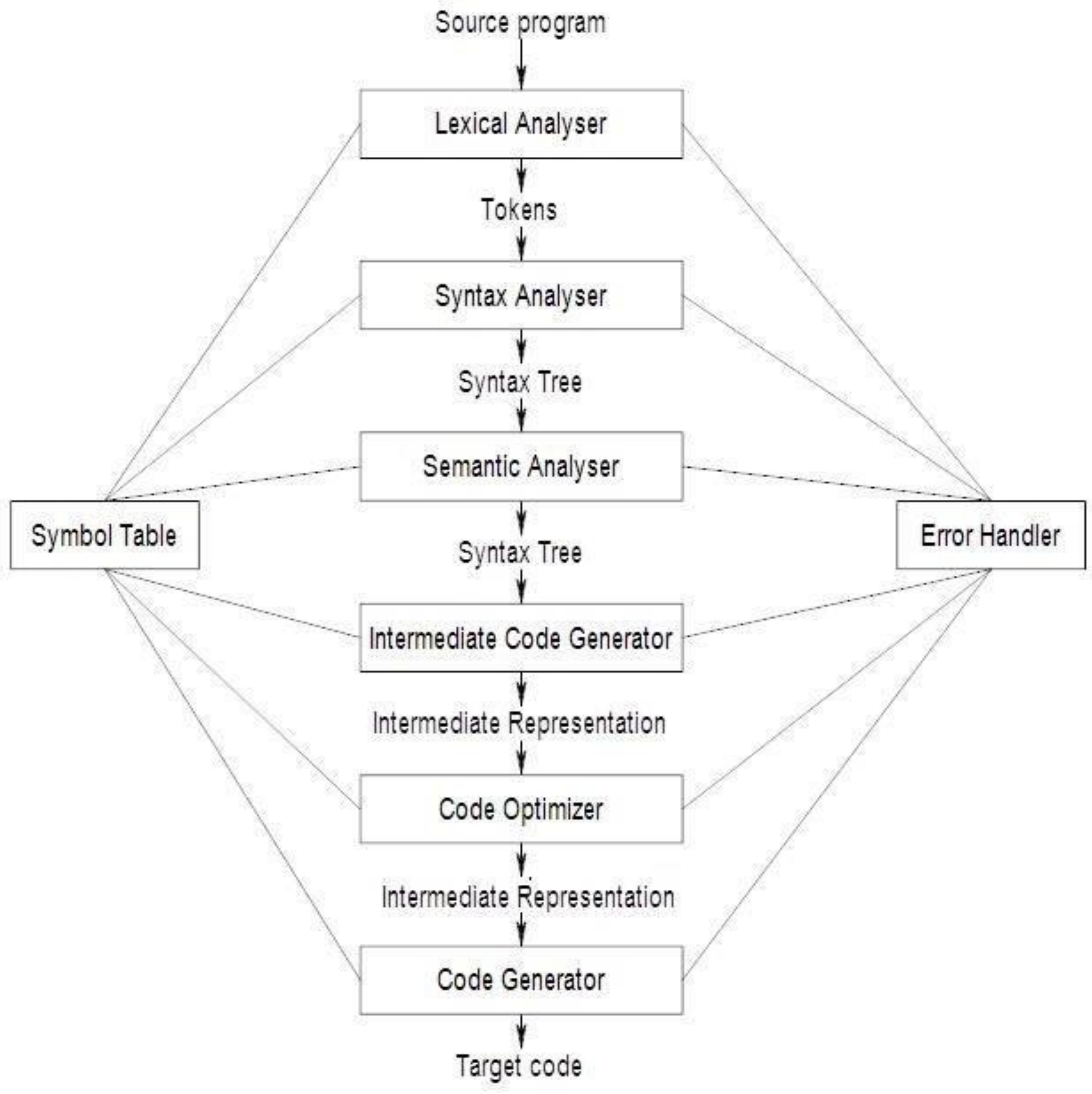
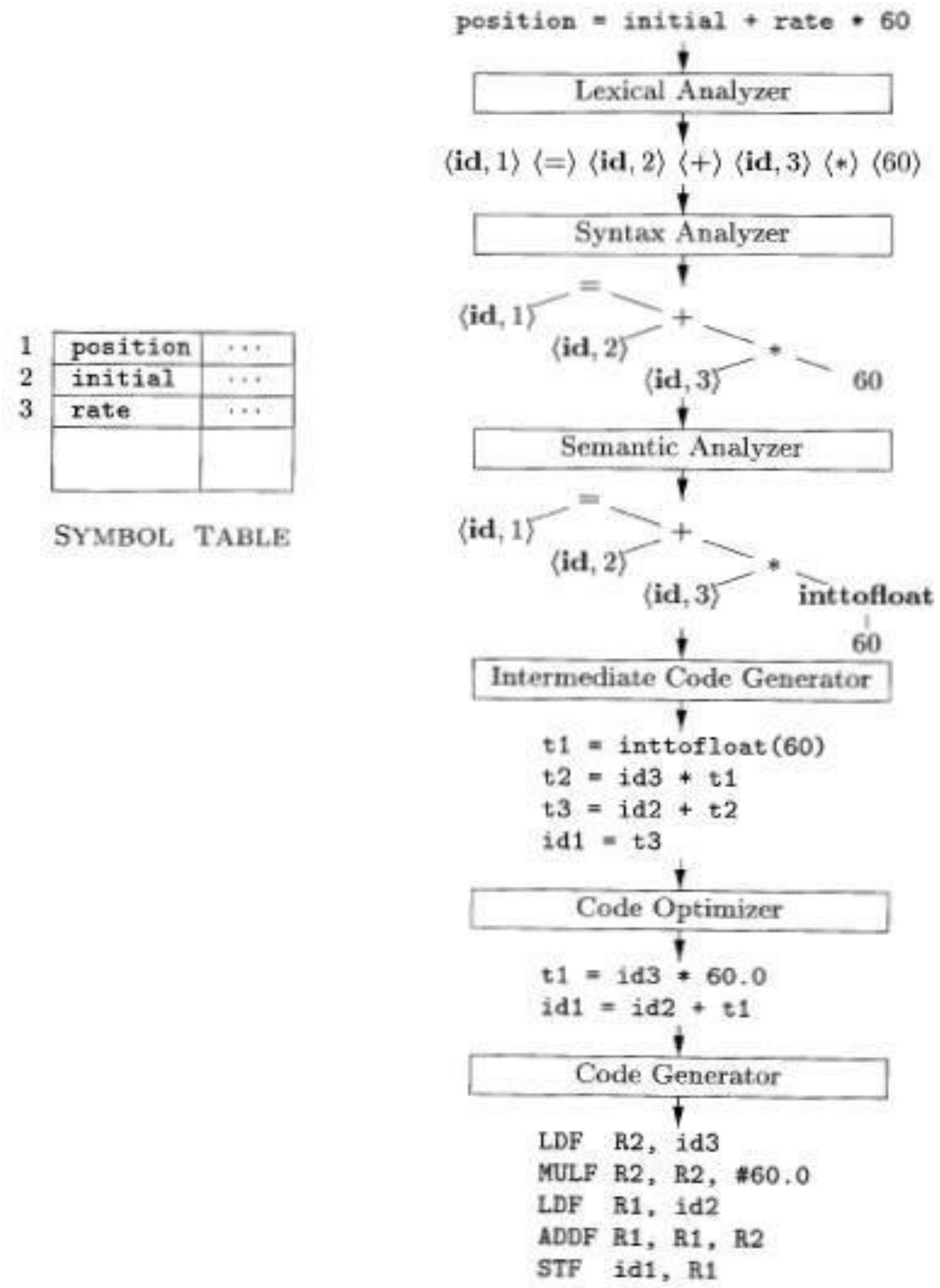


Figure: - Phases of a compiler

Example: - Compile the statement **position = initial + rate * 60**


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
⟨id, 1⟩ = + ⟨id, 2⟩ * ⟨id, 3⟩ 60
Semantic Analyzer
⟨id, 1⟩ = + ⟨id, 2⟩ * ⟨id, 3⟩ inttofloat 60
Intermediate Code Generator
Code Optimizer
Code Generator
1 position ...
2 initial ...
3 rate ...
SYMBOL TABLE

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Fig: - Output of each phase of compiler

1.5 Pass and Phase

Phases deals with the logical organization of a compiler. In an implementation, activities from several phases may be grouped together into a *pass* that reads an input file and writes an output file. For

example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass. Code optimization might be an optional pass. Then there could be a back-end pass consisting of code generation for a particular target machine.
Multi-pass Compilation requires more memory since we need to store the output of each phase in totality. Multi-pass compiler also takes a longer time to compile since it involves reading of the input in different forms (tokens, parse tree, etc) multiple number of times.
In practice, Compilers are designed with the idea of keeping the number of passes as minimum as possible. The number of passes required in a compiler to process an input source program depends on the structure of programming language. C language compilers can be implemented in a single pass, while ALGOL-68 compilers cannot be implemented in a single pass.

Comparison between Interpreter and Compiler.

S.NO	INTERPRETER	COMPILER
1.	An interpreter is a program which translates each statement of the source program and executes the machine code of that statement.	A Compiler is a program which translates the entire source program and generates the machine code of the source program.
2.	Executing the programs using interpreter is slower.	Executing the programs using compiler is faster.
3.	The source program gets interpreted every time it is to be executed. Hence interpretation is less efficient than compilation.	The source program gets compiled once and the object code of it is stored on the hard disk. It can be used every time the program is to be executed.
4.	Developing interpreter is an easier task.	Developing compiler is a complicated task and is difficult.
5.	Interpreter is simpler and they require less amount of memory.	Compiler is a complex program and it requires large amount of memory.

1.6 Evaluation of Programming Language

In olden days, computers were programmed in machine language by explicitly specifying the operations to be done. These operations were very low level and includes operations like data transfer, adding contents of registers, comparing the contents and so on. Machine language programs are complex to write, difficult to understand, modify and more error prone. To reduce such complexities, high level programming languages were developed.
The first step towards the development of high level programming language was the development of assembly language instructions called mnemonics. A major step towards development of higher-level language was initiated in mid of 1950 with FORTRAN language. It was developed for scientific computation. Cobol was developed for business data processing. Lisp was developed for symbolic computations. The main idea behind these languages was to provide higher level notations to the programmers for performing numerical computations, implementing business applications and symbolic programs easily.

Today, there are thousands of programming languages are in use. These languages can be classified in variety of ways.
One classification is made based on generations.

i) First Generation- Machine languages
ii) Second Generation- Assembly languages
iii) Third Generation- Higher level languages like FORTRAN, COBOL, Lisp, C, C++ and Java.
iv) Fourth Generation- Designed for NOMAD for report generation, SQL for database queries and Postscript for text formatting.
v) Fifth Generation- logic and constraint based languages like Prolog and OPS5.

Another Classification of languages are:

i) ***Imperative languages***- program specifies how computation is to be done. Example- C, C++, C# and Java.
ii) ***Declarative languages***- program specifies what is to be done. Example- SQL.
iii) ***Functional languages***- Designed for symbolic computations and list processing applications. Example- ML and Haskell.
iv) ***Neumann languages-*** these are the programming languages whose computational model is based on the von Neumann computer architecture. Example- FORTRAN and C are von Neumann languages.
v) ***Object Oriented Programming languages***- these languages supports object-oriented programming, a programming style in which a program consists of a collection of objects that interact with one another. Example- Simula 67, Smalltalk, C++, C#, Java and Ruby.
vi) ***Scripting Languages***- Developed for a special run-time environment. Example- AWK, Perl, PHP, Ruby etc.

1.7 The Science of building Compiler

A compiler is a large program which translates high level language into an equivalent machine code. A compiler must accept all source programs that conform to the specification of the language; the set of source programs is infinite and any program can be very large, consisting of possibly millions of lines of code. Any transformation performed by the compiler while translating a source program must preserve the meaning of the program being compiled.

i) Modeling in compiler design and implementation

Implementation of compiler involves the design of right mathematical models and selection of right algorithms. Some of most fundamental models are finite-state machines and regular expressions. These models are useful for describing the lexical units of programs (keywords, identifiers, and such) and for describing the algorithms used by the compiler to recognize those units. Also among the most fundamental models are context-free grammars, used to describe the syntactic structure of programming languages.

ii) The Science of Code Optimization

Code Optimization- It is a program transformation technique which improves the code such that the resultant target code will get execute faster by consuming less resource.
The optimization technique should not influence the semantics of the input source program. There are number of code optimization techniques that can be applied for developing the efficient compilers. Code optimization schemes must meet the following design objectives.

- Optimization must preserve the meaning of the compiled program.
- Optimization must improve the performance.
- Compilation time must be kept minimum.
- The engineering effort required must be manageable.

There is an infinite number of program optimizations that we could implement, and it takes a nontrivial amount of effort to create a correct and effective optimization. We must prioritize the optimizations, implementing only those that lead to the greatest benefits on source programs encountered in practice.

1.8 Applications of Compiler Technology

Compiler design impacts various areas of computer science

i) **Implementation of Higher-Level Programming Languages**

A high-level programming language defines a programming abstraction: the programmer expresses an algorithm using the language, and the compiler must translate that program to the target language. Generally, higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Optimizing compilers include techniques to improve the performance of generated code, thus offsetting the inefficiency introduced by high-level abstractions.

ii) **Optimization for Computer Architectures**

The rapid evolution of computer architectures has also led to an insatiable demand for new compiler technology. Almost all high-performance systems take advantage of the same two basic techniques: *parallelism* and *memory hierarchies.* Parallelism can be found at several levels: at the *instruction level,* where multiple operations are executed simultaneously andat the *processor level,* where different threads of the same application are run on different processors. Memory hierarchies are a response to the basic limitation that we can build very fast storage or very large storage, but not storage that is both fast and large.

iii) **Design of New Computer Architecture**

In the early days of computer architecture design, compilers were developed after the machines were built. Now a days such practice is not in use. Since many high level programming languages are coming up with advanced features, the performance of a computer system is not determined its speed but also by considering how well compilers can exploit its features. Thus, in modern computer architecture development, compilers are developed in the processor-design stage and also used to evaluate the proposed architectural features. Compiler technology is needed not only to support programming for these architectures, but also to evaluate proposed architectural designs.

iv) **Program Translations**

Compilation is a process of translation in which a high-level language is translated into the machine level instructions, the same technology can be applied to translate between different kinds of languages.

- **Binary Translation:** Compiler technology can be used to translate the binary code for one machine to that of another, allowing a machine to run programs originally compiled for another instruction set. Binary translation technology has been used to increase the availability of software for different machines. It also provide backward compatibility.
- **Hardware Synthesis:** Not only the softwares are written in high-level languages; even hardware designs are also described in high-level hardware description languages like Verilog and VHDL (Very high-speed integrated circuit Hardware Description Language). Hardware designs are typically described at the register transfer level (RTL), where variables represent registers and expressions represent combinational logic. Hardware-synthesis tools translate RTL descriptions automatically into gates, which are then mapped to transistors and eventually to a physical layout.
- **Database Query Interpreters**

  SQL is most one of the most widely used declarative language for searching databases. Database queries consists of relational and boolean operators. These queries are compiled or interpreted into commands to search database for records satisfying the given constrains.
- **Compiled Simulators**

  Simulation is a general technique used in many scientific and engineering disciplines to understand a phenomenon or to validate a design. Inputs to a simulator usually include the description of the design and specific input parameters for that particular simulation run. Simulations can be very expensive. Instead of writing a simulator that interprets the design, it is faster to compile the design to produce machine code that simulates that particular design natively. Compiled simulation can run orders of magnitude faster than an interpreter-based approach. Compiled simulation is used in many state-of-the- art tools that simulate designs written in Verilog or VHDL.

v) Software Productivity Tools

Programs are the most complicated engineering artifacts; they consist of many details, every detail must be correct and precise before the program will work completely. Errors are rampant in programs; errors may crash the system, produce wrong results, may cause security breaches, or even may lead to catastrophic failures in critical systems. So, testing is the primary technique required for locating errors in programs. The problem of finding all errors in a program is undecidable. The following approaches can be used for locating errors:

- ***Dataflow Analysis:*** A data-flow analysis may be designed to warn the programmers of all possible statements violating a particular category of errors.
- ***Type Checking:*** Type checking is an effective and well-established technique to catch inconsistencies in programs.
- ***Bounds Checking:*** many security breaches in systems are caused by buffer overflows in programs written in C. Because C does not have arraybounds checks, it is up to the user to ensure that the arrays are not accessed out of bounds. Failing to check that the data supplied by the user can overflow a buffer, the program may be tricked into storing user data outside of the buffer. An attacker can manipulate the input data that causes the program to misbehave and compromise the security of the system.
- ***Memory Management Tools:*** Garbage Collection is a process of reclaiming the unused memory automatically. This automatic memory management scheme eliminates all the memory management errors. Purify is a tool that dynamically catches memory management errors.

1.9 Programming Language Basics

i) Static scope and dynamic scope

Scope refers to a place in a program where a variable is visible and can be referenced. Static scope is also called lexical scope. In lexical scoping (also known as static scoping), the scope of a variable is determined by the lexical (i.e., textual) structure of a program or by looking only at program. C & Java uses static scope. In dynamic scoping, a variable is bound to the most recent value assigned to that variable, i.e., the most recent assignment during the program’s execution.

ii) Environments and State

The *environment* is a mapping from names to locations in the store. Since variables refer to locations ("1-values" in the terminology of C), we could alternatively define an environment as a mapping from names to variables. The *state* is a mapping from locations in store to their values. That is, the state maps 1-values to their corresponding r-values, in the terminology of C. Environments change according to the scope rules of a language.

iii) Names, identifiers and Variables

An *identifier* is a string of characters, typically letters or digits that refers to (identifies) an entity, such as a data object, a procedure, a class, or a type. All identifiers are names, but not all names are identifiers. Names can also be expressions. Here, *x* and *y* are identifiers, while *x.y* is a name, but not an identifier. Composite names like *x.y* are called *qualified* names. A *variable* refers to a particular location of the store. It is common for the same identifier to be declared more than once; each such declaration introduces a new variable.

Even if each identifier is declared just once, an identifier local to a recursive procedure will refer to different locations of the store at different times.

iv) Block Structure

A *block* is a grouping of declarations and statements. Languages that allow blocks to be nested are said to have *block structure.*

v) Procedures, Functions and Methods

A subprogram that may be called is referred as procedure. A function usually returns a value of some type, whereas procedure does not return any value. Object-oriented languages like Java and C + + use the term "methods." These can behave like either functions or procedures, but are associated with a particular class.

vi) Explicit Access Control

The use of keywords like **public, private,** and **protected,** object oriented languages such as C + + or Java provide explicit control over access to member names in a superclass. These keywords support *encapsulation* by restricting access. Thus, private names are purposely given a scope that includes only the method declarations and definitions associated with that class and any "friend" classes (the C + + term). Protected names are accessible to subclasses. Public names are accessible from outside the class.

vii) Declarations and Definitions

Declarations tell us about the types of things, while definitions tell us about their values. Thus, ***int i*** is a declaration of ***i,*** while ***i = 1*** is a definition of ***i.*** The difference is more significant when we deal with methods or other procedures. In C++, a method is declared in a class definition, by giving the types of the arguments and result of the method (often called the *signature* for the method. The method is then defined, i.e., the code for executing the method is given, in another place.

viii) Aliasing

There is an interesting consequence of call-by-reference parameter passing or its simulation, as in Java, where references to objects are passed by value. It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other.

ix) Parameter Passing Mechanisms

There are two types of parameters actual parameters (the parameters used in the call of a procedure) are associated with the formal parameters (those used in the procedure definition). The most common parameter passing mechanisms are:

a) Call by Value

In this parameter passing mechanism, the changes made to formal parameters will not be reflected on the actual parameters because both actual and formal parameters have separate storage locations.

b) Call by reference

In this parameter passing mechanism, the changes made to formal parameters will be reflected on the actual parameters because formal parameters references the storage locations of actual parameters. Hence, the any change made on formal parameter will be reflected on actual parameter.

c) Call by Name

This technique is used in programming language such as Algol. In this technique, symbolic “name” of a variable is passed, which allows it both to be accessed and update.

x) Aliasing

It is possible that two formal parameters can refer to the same location; such variables are said to be *aliases* of one another. As a result, any two variables, which may appear to take their values from two distinct formal parameters, can become aliases of each other, as well. Suppose *a* is an array belonging to a procedure *p,* and *p* calls another procedure *q(x,y)* with a call *q(a,a).* Suppose also that parameters are passed by value, but that array names are really references to the location where the array is stored, as in C or similar languages. Now, *x* and *y* have become aliases of each other.

************Important Questions************

1. Define language processor? Differentiate between Compiler and Interpreter.
2. Explain in detail about various Phases of Compiler.

   (Or)

   Explain the different Phases of a Compiler, showing the output of each phase for the statement "**position=initial+rate*60**".

   ii) x=(a+b)*(c+d)

   iii) Fahrenheit=Celsius*1.8+32
3. Differentiate between Pass and Phase.
4. Explain about Language Processing System.
5. Define Loader, Linker and Assembler. Explain the structure of a Compiler & its advantages.
6. Describe various parameter passing techniques.
7. Summarize the role of compiler technology in various applications.

2. *Introduction to Lexical Analysis:*

In this chapter we will discuss how to construct a lexical analyzer.
There are three ways to implement Lexical Analyzer:

i) Using State Transition diagrams to recognize various tokens.

ii) We can write a code to identify each occurrences of each lexeme on the input and to return information about the token identified.

iii) We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical analyzer generator and compiling those patterns into code that functions as lexical analyzer. This approach makes it easier to modify a lexical analyzer, since we have only to rewrite the affected patters, not the entire program. Lexical analyzer generator called **LEX.**

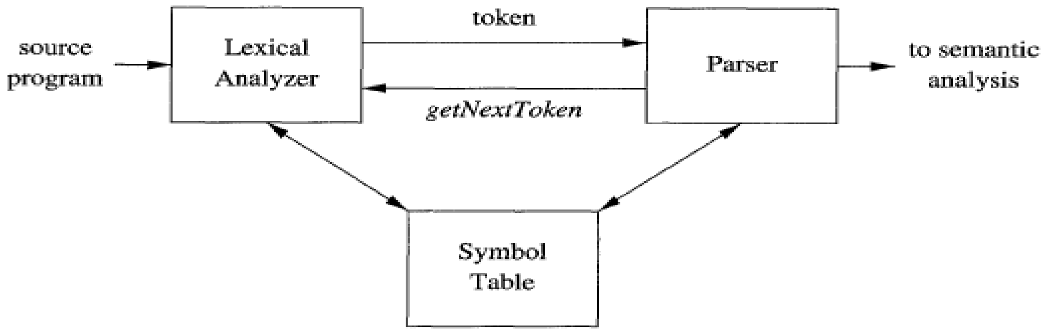
Lexical Analysis: - The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the source program and groups the characters into meaningful sequences called lexemes. It identifies the category (i.e tokens) to which this lexeme belongs. For each lexeme, the lexical analyzer produces output in the form

<token-name, attribute-value>

This output is passed to the subsequent phase i.e syntax analysis.

3. *Role of the Lexical Analyzer*

Lexical analyzer is the first phase of a compiler. The main task of the lexical analyzer is to read the input characters of the source program, group them into lexemes, and produce tokens for each lexeme in the source program. The stream of tokens is sent to the parser for syntax analysis. When lexical analyzer discovers a lexeme constituting an identifier, it interacts with the symbol table to enter that lexeme into the symbol table. Commonly, the interaction is implemented by having the parser call the lexical analyzer. The getNextToken command given by the parser, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce the next token, which it returns to the parser.



Since the lexical analyzer is the part of the compiler that reads the source text, it may perform certain other tasks besides identification of lexemes. One such task is stripping out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input). Another task is correlating error messages generated by the compiler with the source program. For instance, the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message. In some compilers, the lexical analyzer makes a copy of the source program with the error messages inserted at the appropriate positions. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer. Sometimes, lexical analyzers are divided into two processes:

a. Scanning consists of the simple processes that perform such as deletion of comments and eliminating excessive whitespace characters into one.
b. Lexical analysis is the more complex portion, which produces the sequence of tokens as output.

4. **Lexical Analysis Vs. Parsing**

There are a number of reasons for separating Lexical Analysis and Parsing.

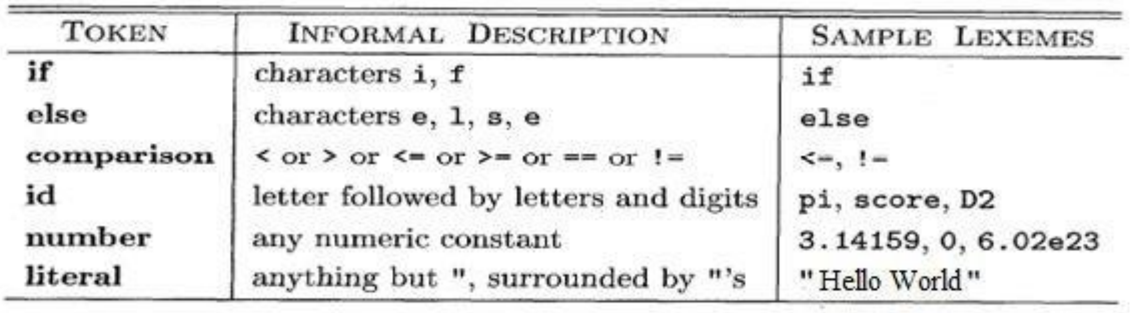
i) To simplify the overall design of the compiler.
ii) Compiler efficiency is improved. A separate lexical analyzer allows us to apply specialized techniques that serve only the lexical task, not the job of parsing. In addition, specialized buffering techniques for reading input characters can speed up the compiler significantly.
iii) Compiler portability is enhanced.

5. **Token, patterns and Lexemes: -**

- A token is a sequence of characters having a collective meaning.
  A **token** is a pair consisting of a token name and an optional attribute value. The token name is the category of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier etc.
- A **pattern** is a description that specify the rules that the lexemes should follow in order to belong to that token.
- A **lexeme** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token.

Example : printf("total = %d\n", score);

printf and ***score*** are lexemes matching the pattern of token **ID**.



TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	"Hello World"

In many programming languages, the following classes cover most or all of the tokens:

i) One token for each keyword. The pattern for a keyword is the same as the keyword itself.
ii) Tokens for the operators, either individually or in classes such as the token comparison mentioned.
iii) One token representing all identifiers.
iv) One or more tokens representing constants, such as numbers and literal strings.
v) Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

6. Input Buffering

The lexical analyzer scans the input from left to right one character at a time. It uses two pointers- **lexemeBegin** pointer and **forward pointer**. Lexeme begin pointer is set to the beginning of the lexeme. Whereas, forward pointer moves ahead to search for the end of the lexeme. As soon as the blank space is encountered, it indicates end of lexeme. Once the next lexeme is determined, **forward** is set to the character at its right end. Then, after the lexeme is recognized, an attribute value of a token is returned to the parser, **lexemeBegin** is set to the character immediately after the lexeme justfound.

Buffer Pairs (or) Two Buffer Scheme

The input character is thus read from secondary storage, but reading in this way from secondary storage is costly. Hence, buffering technique is used. In this method n this method two buffers are used to store the input string. Each buffer is of the same size *N,* and *N* is usually the size of a disk block, e.g., 4096 bytes. The first buffer and second buffer are scanned alternately. When end of current buffer is reached the other buffer is filled. Advancing **forward** requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move **forward** to the beginning of the newly loaded buffer. We must check, each time we advance **forward,** that we have not moved off one of the buffers; if we do, then we must also reload the other buffer. Thus, for each character read, we make two tests: one for the end of the buffer, and one to determine what character is read (the latter may be a multiway branch). We can combine the buffer-end test with the test for the current character if we extend each buffer 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 considered as sentinel**.**

7. Lexical Errors

Lexical errors include misspellings of identifiers, keywords, or operators. It is hard for a lexical analyzer to tell, without the help of other components, that there is a source-code error. For instance, if the string **fi** i.e fi( ) is encountered for the first time in a C program in the context a lexical analyzer cannot tell whether fi is a misspelling of the keyword if or an undeclared function identifier. Since fi isa valid lexeme for the token id, the lexical analyzer must return the token id to the parser and the parser handle an error due to transposition of the letters. However, suppose a situation arises where the lexical analyzer is unable to proceed because the lexeme doesn't matches any of the patterns for tokens. The simplest recovery strategy is ***"panic mode" recovery.*** We delete successive characters from the remaining input, until the lexical analyzer can find a well-formed token at the beginning of what input is left. This recovery technique may confuse the parser, but in an interactive computing environment it may be quite adequate. Other possible error-recovery actions are:

i) Delete one character from the remaining input.
ii) Insert a missing character into the remaining input.
iii) Replace a character by another character.
iv) Transpose two adjacent characters.

8. Regular Expressions

A regular expression is a pattern that describes a set of strings.
Regular expressions are used to describe the languages.
The regular expression is the one that matches a single character. For example: 's' matches any input string where letter ***s*** is present like ***sink, base***.

Meta characters in Regular Expression:

. → matches any character except a new line.
^ → matches the start of the line.
$ → matches end – of- the – line.
[ ] → A character class- matches any letter within the parenthesis. [0123456789] matches 0 or 1 or 2 etc.
[^ abcd] → ^ inside the square bracket represents the match of any character except the ones in the bracket.
| → matches either preceding Regular Expression or Succeeding Regular Expression.
( ) → used for grouping Regular Expressions
***,+,?** → for specifying repetitions in Regular Expressions
***** zero or more occurances
+ one or more occurrences
? zero or one occurrences
{ } → indicates how many times the previous pattern is matched. Eg. A {1,3} represents a match of one to three occurrences of 'a'. The strings that matches are 'dad','daad','daaad' etc.

Regular expressions are an important notation for specifying lexeme patterns. To describe the set of valid C identifiers use a notation called regular expressions. In this notation, if letter_ is established to stand for any letter or the underscore, and digit is established to stand for any digit. Then the identifiers of C language are defined by

letter (letter| digit)*

The vertical bar above means union, the parentheses are used to group subexpressions, the star means "zero or more occurrences of". The letter at the beginning indicates that the identifier can contain any letter at the beginning. The regular expressions are built recursively out of smaller regular expressions. The regular expressions are built recursively out of smaller regular expressions, using the rules described below. Each regular expression r denotes a language *L(r),* which is also defined recursively from the languages denoted by r ' s subexpressions. Here are the rules that define the regular expressions over some alphabet ∑ and the languages that those expressions denote.

BASIS: There are two rules that form the basis:

1. ∈ (epsilon) is a regular expression, and *L(∈)* is {∈}, that is, the language whose sole member is the empty string.
2. If *a* is a symbol in ∑, then **a** is a regular expression, and L**(a) =** *{a},* that is, the language with one string, of length one, with *a* in its one position.

INDUCTION: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and *s* are regular expressions denoting languages *L(r)* and *L(s),* respectively.

1. **(r)|(s)** is a regular expression denoting the language *L(r)* U *L(s).*
2. **(r)(s)** is a regular expression denoting the language *L(r)L(s).*
3. **(r)*** is a regular expression denoting (L(r))*.
4. **(r)** is a regular expression denoting *L(r).*

This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

As defined, regular expressions often contain unnecessary pairs of parentheses. We may drop certain pairs of parentheses if we adopt the conventions that:

a) The unary operator * has highest precedence and is left associative.
b) Concatenation has second highest precedence and is left associative.
c) | has lowest precedence and is left associative.

There are a number of algebraic laws for regular expressions; each law asserts that expressions of two different forms are equivalent. Figure below shows some of the algebraic laws that hold for arbitrary regular expressions *r, s,* and *t.*

LAW	DESCRIPTION
$r \mid s = s \mid r$	$\mid$ is commutative
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ is associative
$r(st) = (rs)t$	Concatenation is associative
$r(s \mid t) = rs \mid rt;\quad (s \mid t)r = sr \mid tr$	Concatenation distributes over $\mid$
$\in r = r \in = r$	$\in$ is the identity for concatenation
$r^* = (r \mid \in)^*$	$\in$ is guaranteed in a closure
$r^{**} = r^*$	* is idempotent

9. Regular definitions

Regular Definitions are names given to certain regular expressions and those names can be used in subsequent expressions as symbols of the language. If Σ is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form:

$$
\begin{array}{lcl}
d_1 & \rightarrow & r_1 \\
d_2 & \rightarrow & r_2 \\
 & \cdots & \\
d_n & \rightarrow & r_n
\end{array}
$$

where

1. Each **d**$_i$ is a new symbol, not in Σ and not the same as any other of the **d**$_i$'s, and
2. Each **r**$_i$ is a regular expression over the alphabet Σ U {d$_l$, d$_2$,. . . , d$_{i-l}$}.

***Example 1* :** C identifiers are strings of letters, digits, and underscores. Write a regular definition for the language of C identifiers.

$$
\begin{array}{rcl}
\textit{letter} & \rightarrow & \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \mid \\
\textit{digit} & \rightarrow & \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\
\textit{id} & \rightarrow & \textit{letter}\ (\ \textit{letter}\ \mid \textit{digit}\ )^*
\end{array}
$$

Using shorthand notations, the regular definition can be rewritten as:

$$
\begin{array}{rcl}
\textit{letter} & \rightarrow & [\mathtt{A\text{-}Za\text{-}z\_}] \\
\textit{digit} & \rightarrow & [\mathtt{0\text{-}9}] \\
\textit{id} & \rightarrow & \textit{letter}\ (\textit{letter} \mid \textit{digit}\ )^*
\end{array}
$$

***Example 2* :** Unsigned numbers (integer or floating point) are strings such as 5280, 0.01234, 6.336E4, or 1.89E-4. Write a regular definition for unsigned numbers in C language.

$$
\begin{array}{rcl}
\textit{digit} & \rightarrow & [\mathtt{0-9}] \\
\textit{digits} & \rightarrow & \textit{digit}^+ \\
\textit{number} & \rightarrow & \textit{digits}\ (.\ \textit{digits})?\ (\ \mathtt{E}\ [+-]?\ \textit{digits}\ )?
\end{array}
$$

10. Recognition of Tokens

In the previous section we learned how to express patterns using regular expressions. Now, we study how to take the patterns for all the needed tokens and build a piece of code that examines the input string and finds a prefix that is a lexeme matching one of the patterns.

The below example describes a simple form of branching statements and conditional expressions. This syntax is similar to that of the language Pascal, in that **then** appears explicitly after conditions.

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt} \\
 & \mid & \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{stmt}\ \textbf{else}\ \textit{stmt} \\
 & & e \\
\textit{expr} & \rightarrow & \textit{term}\ \textbf{relop}\ \textit{term} \\
 & \mid & \textit{term} \\
\textit{term} & \rightarrow & \textbf{id} \\
 & \mid & \textbf{number}
\end{array}
$$

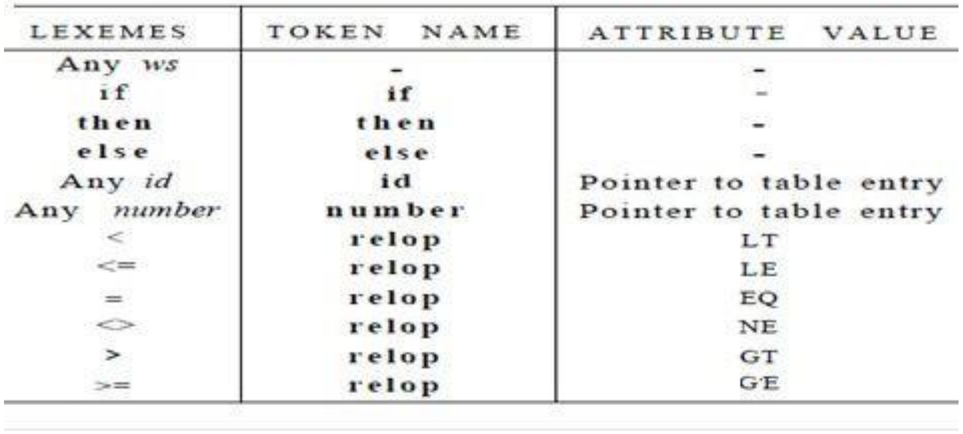
The terminals of the grammar, which are **if, then, else, relop, id,** and **number,** are the names of tokens as far as the lexical analyzer is concerned. The patterns for these tokens are described using regular definitions.

digit	---->	[0-9]
digits	---->	*digit*+
number	---->	*digits* (. *digits*)? (E [+-]? *digits*)?
letter	---->	[A-Za-z]
id	---->	*letter* (*letter* \ *digit*)*
if	---->	if
then	---->	then
else	---->	else
relop	---->	< | > | <= | >= | <> | =

For this language, the lexical analyzer will recognize the keywords **if**, **then,** and **e l s e ,** as well as lexemes that match the patterns for *relop, id,* and *number.* To simplify matters, we make the common assumption that keywords are also *reserved words:* that is, they are not identifiers, even though their lexemes match the pattern for identifiers.
In addition, we assign the lexical analyzer the job of stripping out whitespace, by recognizing the "token" *ws* defined by:

ws → (tab|blank space|new line)

Here, **blank, tab,** and **newline** are abstract symbols that we use to express the ASCII characters of the same names. Token *ws* is different from the other tokens in that, when we recognize it, we do not return it to the parser, but rather restart the lexical analysis from the character that follows the whitespace. It is the following token that gets returned to the parser.
The table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what attribute value is returned.



LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	-	-
if	if	-
then	then	-
else	else	-
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Transition Diagrams

Compiler converts regular-expression patterns to transition diagrams. Transition diagrams have a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns. Edges are directed from one state of the transition diagram to another. Each edge is labeled by a symbol or set of symbols. If we are in some state s, and the next input symbol is a, we look for an edge out of state s labeled by a (and perhaps by other symbols, as well). If we find such an edge, we enter the state of the transition diagram to which that edge leads.
Some important conventions about transition diagrams are:

i) Certain states are said to be *accepting,* or *final.* These states indicate that a lexeme has been found, although the actual lexeme may not consist of all positions between the ***lexemeBegin*** and *forward* pointers.

ii) In addition, if it is necessary to retract the *forward* pointer one position (i.e., the lexeme does not include the symbol that got us to the accepting state), then we shall additionally place a * near that accepting state.

iii) One state is designated the *start state,* or *initial state;* it is indicated by an edge, labeled "start," entering from nowhere. The transition diagram always begins in the start state before any input symbols have been read.

Below transition diagram that recognizes the lexemes matching the token **relop.** We begin in state 0, the start state. If we see < as the first input symbol, then among the lexemes that match the pattern for **relop** we can only be looking at <, <>, or <=. We therefore go to state 1, and look at
the next character. If it is =, then we recognize lexeme <=, enter state 2, and return the token **relop** with attribute LE, the symbolic constant representing this particular comparison operator. If in state 1 the next character is >, then instead we have lexeme <>, and enter state 3 to return an indication that the not-equals operator has been found. On any other character, the lexeme is <, and we enter state 4 to return that information. Note, however, that state 4 has a * to indicate that we must retract the input one position. On the other hand, if in state 0 the first character we see is =, then this one character must be the lexeme. We immediately return that fact from state 5. The remaining possibility is that the first character is >. Then, we must enter state 6 and decide, on the basis of the next character, whether the lexeme is >= (if we next see the = sign), or just > (on any other character).

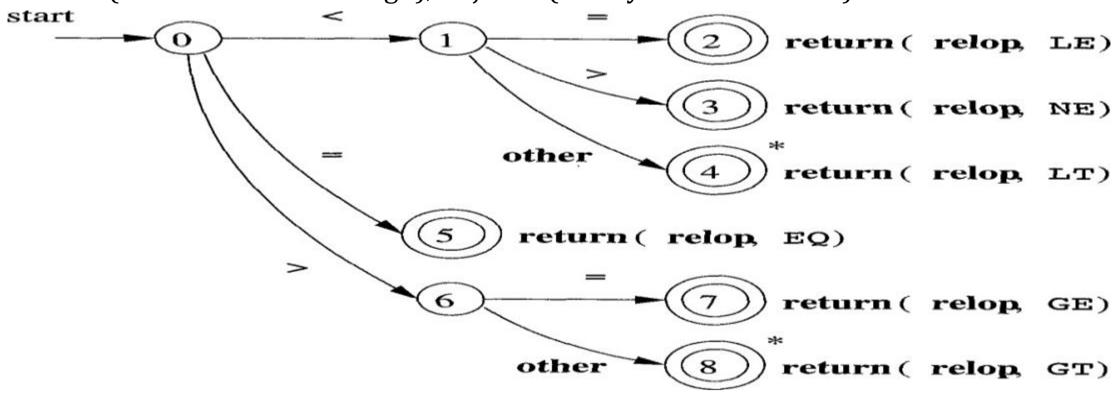


Figure: Transition diagram for relop

11. **Recognition of Reserved Words and Identifiers**

Recognizing keywords and identifiers presents a problem. Usually, keywords like if or **then** are reserved (as they are in our running example), so they are not identifiers even though they *look* like identifiers. The below diagram will recognize the keywords if, **then,** and **e l s e** of our running example.

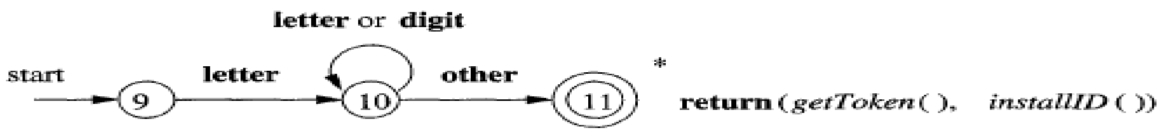


Figure: Transition diagram for identifier

There are two ways that we can handle reserved words that look like identifiers:

i) Install the reserved words in the symbol table initially. A field of the symbol-table entry indicates that these strings are never ordinary identifiers, and tells which token they represent. When we find an identifier, a call to *installlD* places it in the symbol table if it is not already there and returns a pointer to the symbol-table entry for the lexeme found. Of course, any identifier not in the symbol table during lexical analysis cannot be a reserved word, so its token is **id.** The function*getToken* examines the symbol table entry for the lexeme found, and returns whatever token name the symbol table says this lexeme represents — either **id** or one of the keyword tokens that was initially installed in the table.

ii) Create separate transition diagrams for each keyword; an example for the keyword **then** is shown in Fig.

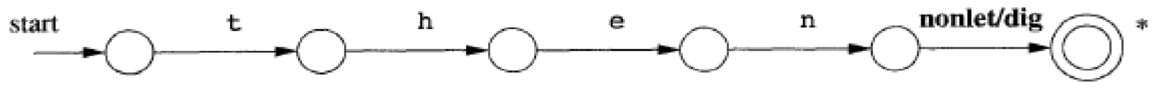


Figure: Transition diagram for then

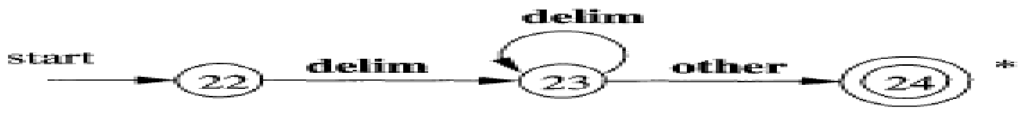


Figure: Transition diagram for whitespace

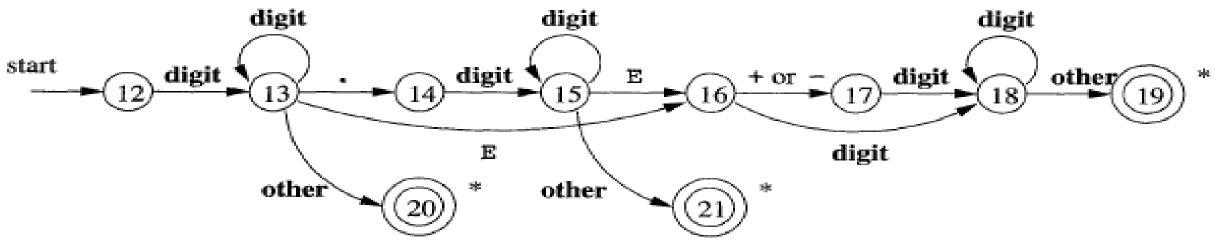


Figure: Transition diagram for unsigned numbers

12. **LEXICAL ANALYZER GENERATOR LEX**

Lex is a tool or in a more recent implementation Flex, that allows us to specify a lexical analyzer by specifying regular expressions to describe patterns for tokens. The input notation for the Lex tool is referred to as the *Lex language* and the tool itself is the *Lex compiler.* Behind the scenes, the Lex compiler transforms the input patterns into a transition diagram and generates code, in a file called ***lex.yy.c*** that simulates this transition diagram.

USE OF LEX:

Figure below shows how LEX is used. An input file, which we call ***lex.l***, is written in the Lex language and describes the lexical analyzer to be generated. This file is given as input to the LEX Compiler. The LEX compiler transforms ***lex.1*** to a C program, in a file that is always named ***lex.yy.c.*** The latter file is compiled by the C compiler into a file called ***a.out***. The C-compiler outputis a working lexical analyzer that can take a stream of input characters and produce a stream of tokens.

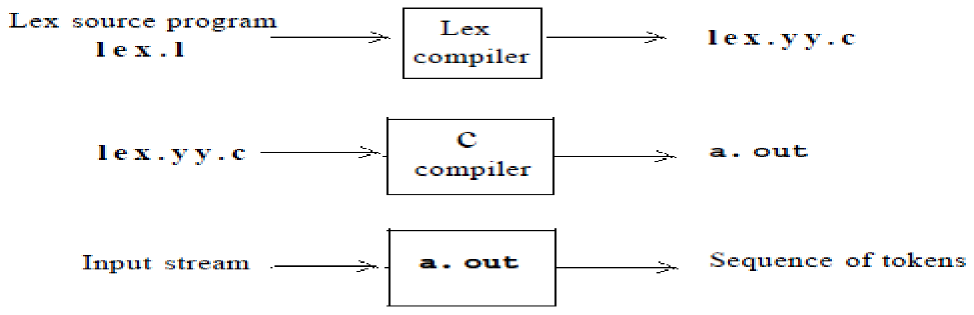


Figure Creating a lexical analyzer with Lex

Structure of LEX Program:

The LEX program consists of the following sections.

```
declarations
 %%
translation rules
 %%
auxiliary functions
```

- **Declaration Section:** Consists of regular definitions that can be used in translation rules.

Example: letter{a-zA-Z}
Apart from the regular definitions, the declaration section usually contains the # defines,C prototype declaration of functions used in translation rules and some # include statements for C library functions used in translation rules. all these statements are mentioned between special brackets %{ and %}.
Example: %{
 # define WORD 1 %}
These statements are copied into **lex.yy.c.**

- **Translation Rules Section**: consists of statements in the following form

Pattern 1 { Action 1 }
Pattern 2 { Action 2 }
....
Pattern N { Action N }

Each pattern is a regular expression, which may use the regular definitions of the declaration section. Where Pattern 1, Pattern 2,…,Pattern N are regular expressions and the Action 1,Action 2,…Action N are all program segments describing the action to be taken when the pattern matches.

- **Auxiliary Functions section:** usually contains the definition of the C functions used in the action statements. The whole section is copied as is into **lex.yy.c.** These functions can be compiled separately and loaded with the lexical analyzer.

The lexical analyzer created by **Lex** behaves in concert with the parser as follows. When called by the parser, the lexical analyzer begins reading its remaining input, one character at a time, until it finds the longest prefix of the input that matches one of the patterns *Pi.* It then executes the associated action *Ai.* Typically, *Ai* will return to the parser, but if it does not (e.g., because *Pi* describes whitespace or comments), then the lexical analyzer proceeds to find additional lexemes, until one of the corresponding actions causes a return to the parser. The lexical analyzer returns a single value, the token name, to the parser, but uses the shared, integer variable ***yylval*** to pass additional information about the lexeme found, if needed.
The actions taken when ***id*** is matched are listed below:

1. Function ***installID()*** is called to place the lexeme found in the symbol table.
2. This function returns a pointer to the symbol table, which is placed in global variable ***yylval*** , where it can be used by the parser or a later component of the compiler. Note that ***installID()*** has available to it two variables that are set automatically by the lexical analyzer:
   (a) ***yytext*** is a pointer to the beginning of the lexeme.
   (b) ***yyleng*** is the length of the lexeme found.
3. The token name **ID** is returned to the parser.
   The action taken when a lexeme matching the pattern *number* is similar, using the auxiliary function installNumO.

```
%{
     /* definitions of manifest constants
     LT, LE, EQ, NE, GT, GE,
     IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim       [ \t\n]
ws          {delim}+
letter      [A-Za-z]
digit       [0-9]
id          {letter}({letter}|{digit})*
number      {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
%%
{ws}        {/* no action and no return */}
if          {return(IF);}
then        {return(THEN);}
else        {return(ELSE);}
{id}        {yylval = (int) installID(); return(ID);}
{number}    {yylval = (int) installNumO ; return(NUMBER);}
"<"         {yylval = LT; return(RELOP) ;}
"<="        {yylval = LE; return(RELOP) ;}
"="         {yylval = EQ; return(RELOP) ;}
"<>"        {yylval = NE; return(RELOP) ;}
">"         {yylval = GT; return(RELOP) ;}
">="        {yylval = GE; return(RELOP) ;}
%%
int installID()    {/* function to install the lexeme, whose
                      first character is pointed to by yytext
                      arid whose length is yyleng, into the
                      symbol table and return a pointer

                     thereto */}
int installNumO {/* similar to installlD, but puts numer-
                   ical constants into a separate table

                  */}
```

Example: Write a LEX Program to count the number of words.

```
%{
int wc=0;
%}
char [a-zA-Z]
words {char}+
%%
{words}  {wc++;}
%%
main()
{
  yylex(); /* to invoke lexical analyzer */
printf("The no. of words are %d", wc);
}
 yywrap()
{
 return 1; /* returns 1 when the end of input is found */
}
```

OUTPUT:

```
$ lex words.l
$ cc lex.yy.c
$ ./a.out
Hai this is SMCE
^D
The no. of words are 4.
```

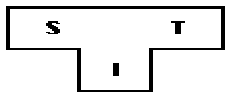
*****Additional Concepts*****

Bootstrapping:

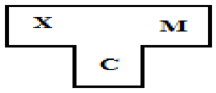
A compiler is characterized by three languages:

1. Source Language
2. Target Language
3. Implementation Language

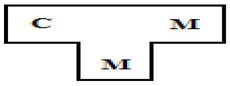
Notation: ${}^{S}C_{I}^{T}$ represents a compiler for Source *S*, Target *T*, implemented in *I*.
The *T-diagram* shown below is also used to depict the same compiler.



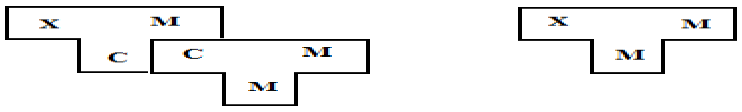
Consider you want to develop a compiler for your dream language X which must run on machine M and written in M language. The compiler for language X can be implemented in programming language like C, as show in fig (a).



The above compiler is compiled by using a C compiler which is implemented in M language, and compiles the C program into M language (see Figure b).



This produces the compiler for language X in language M which produces target code in M language. The resulting compiler for X is in M language as shown in below fig.



What is Cross Compiler?
A compiler that runs on one computer but produces object code for a different type of computer. Cross compilers are used to generate software that can run on computers with a new architecture.
A **cross compiler** is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
Example: a compiler that runs on a Windows 7 PC but generates code that runs on Android smartphone is a cross compiler.
A cross compiler is necessary to compile for multiple platforms from one machine.

******************IMPORTANT QUESTIONS******************

1. A) Explain the role of Lexical Analyzer.
   B) What are the reasons for separating Lexical analysis from the Syntax Analysis?
2. A) What is Regular Expression? Explain algebraic laws of Regular Expression.
   B) Draw the State Transition Diagram for recognizing the Identifiers and relational operators.
3. A) Explain the structure of Lex Program.
   B) Lex program for recognizing various tokens.

UNIT – II

Syntax Analysis -: The Role of a parser, Context free Grammars Writing A grammar, top down passing bottom up parsing Introduction to Lr Parser.

Syntax Analysis

2.1 The Role of the Syntax Analysis (Parser)

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, as shown in Fig. 2.1, and verifies that the string of token names can be generated by the grammar for the source language.

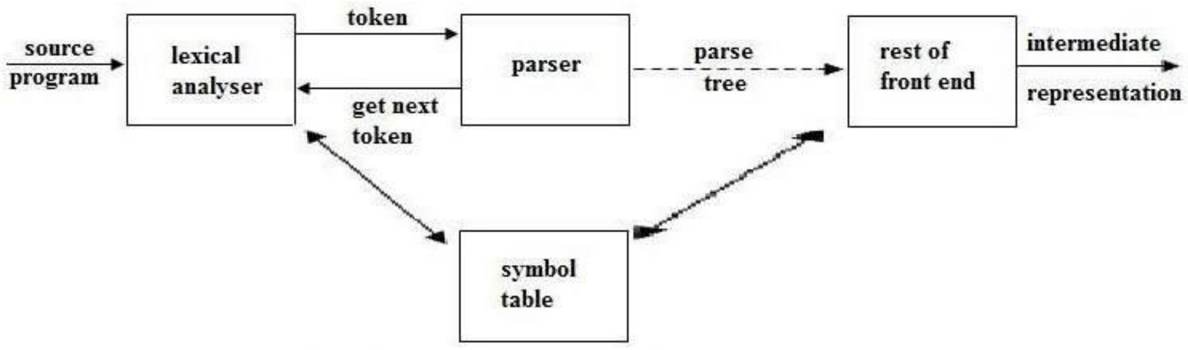
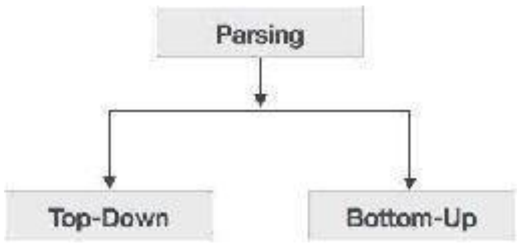


Fig 2.1 Position of parser in compiler model

The function of parser is to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing. In fact, the parse tree need not be constructed explicitly, since checking and translation actions can be interspersed with parsing.

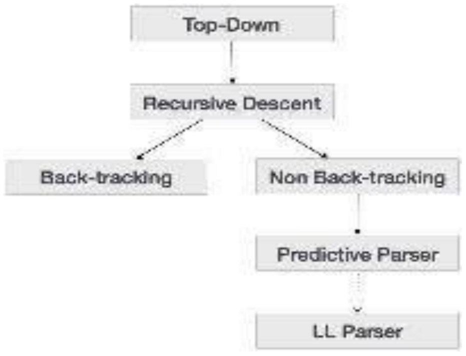
There are **three general types of parsers** for grammars: **universal, top-down, and bottom-up**. Universal parsing methods can parse any grammar. These general methods are too **inefficien**t to use in production compilers.

The **methods commonly used in compilers** can be classified as being either **top-down or bottom-up**. Syntax analyzers follow production rules definedby means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing.



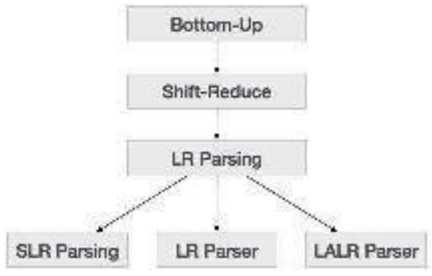
- **Top-down Parsing**

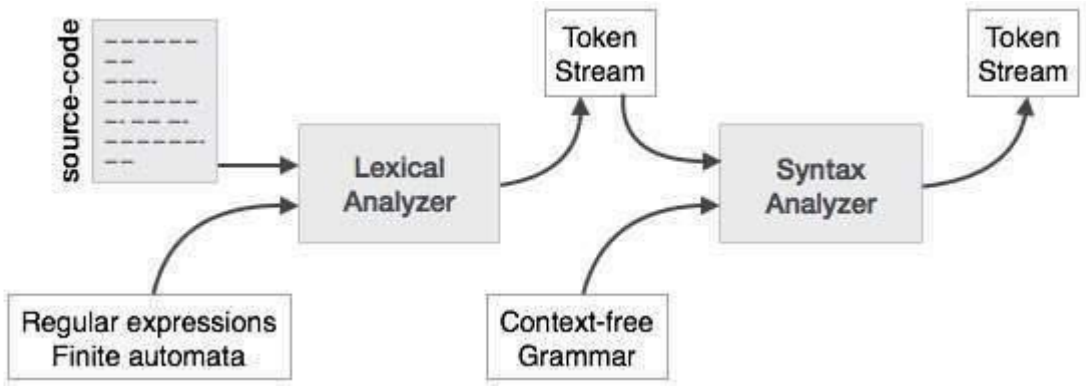
When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing. The types of top-down parsing are depicted:



- **Bottom-up Parsing**

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The types of bottom-up parsing are depicted:





2.1.1 Representative Grammars

Some of the grammars are presented here for ease of reference. Associativity and precedence are captured in the following grammar. *E* represents expressions consisting of terms separated by + signs, *T* represents terms consisting of factors separated by * signs and *F* represents factors that can be either parenthesized expressions or identifiers:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id \text{ ....(2.1)}$$

Expression grammar (2.1) belongs to the class of LR grammars that are suitable for bottom-up parsing. This grammar can be adapted to handle additional operators and additional levels of precedence. However, it cannot be used for top-down parsing because it is left recursive.

The following non-left-recursive variant of the expression grammar (2.1) will be used for top-down parsing:

$$E \rightarrow T E'$$

$$E' \rightarrow + T E' / \varepsilon$$
$$T \rightarrow F T'$$
$$T' \rightarrow * F T' / \varepsilon$$
$$F \rightarrow (E) \mid id \qquad \dots (2.2)$$

The following grammar treats + and * alike, so it is useful for illustrating techniques for handling ambiguities during parsing:

$$E \rightarrow E + E \mid E * E \mid (E) \mid id \qquad \dots (2.3)$$

Here, ***E*** represents expressions of all types. Grammar (2.3) permits more than one parse tree for expressions like ***a + b*c***.

2.1.2 Syntax Error Handling

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 programmay have the following kinds of errors at various stages:

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

The error handler in a parser has goals that are simple to state but challenging to realize:

- 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 correct programs.

2.1.3 Error-Recovery Strategies

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

- **Panic-Mode Recovery**

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 semi- colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

- **Phrase-Level Recovery**

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

- **Error Productions**

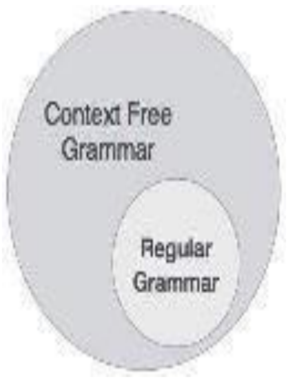
Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

- **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 a 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.

2.2 Context-Free Grammars

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:

Grammars systematically describe the syntax of programming language constructs like expressions and statements. Using a syntactic variable ***stmt*** to denote statements and variable ***expr*** to denote expressions, the production

stmt* —> if (*expr*) *stmt* else *stmt **… (2.4)**

specifies the structure of this form of conditional statement. Other productions then define precisely what an *expr* is and what else a *stmt* can be.

2.2.1 The Formal Definition of a Context-Free Grammar

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 2.1: Grammar for simple arithmetic expressions

The grammar defines simple arithmetic expressions. In this grammar, the terminal symbols are **id + - * / ( ).** The nonterminal symbols are ***expression, term*** **and** ***factor,*** and *expression* is the start symbol. ***expression*** $\rightarrow$ ***expression + term***

expression $\rightarrow$ ***expression - term***

expression $\rightarrow$ ***term***

term $\rightarrow$ ***term * factor***

term $\rightarrow$ ***term / factor***

term $\rightarrow$ ***factor***

factor $\rightarrow$ ***(expression)***

factor $\rightarrow$ **id** **… (2.5)**

2.2.2 Notational Conventions

1. ***These symbols are terminals***:

(a) Lowercase letters early in the alphabet, such as *a, b, c.*

(b) Operator symbols such as +, *, and so on.

(c) Punctuation symbols such as parentheses, comma, and so on.

(d) The digits 0, 1. . . 9.

(e) Boldface strings such as **id** or **if**, each of which represents a single terminal symbol.

2. ***These symbols are nonterminals***:

(a) Uppercase letters early in the alphabet, such as *A, B, C.*

(b) The letter *S,* which, when it appears, is usually the **start** symbol.

(c) Lowercase, italic names such as ***expr*** **or** ***stmt.***

(d) When discussing programming constructs, uppercase letters may be used to represent nonterminals for the constructs. For example, nonterminals for expressions, terms, and factors are often represented by *E, T,* and *F,* respectively.

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

4. Lowercase letters late in the alphabet, chiefly *u,v,..., z,* represent (possibly empty) strings of terminals.

5. Lowercase Greek letters, **α, β, γ** for example, represent (possibly empty) strings of grammar symbols. Thus, a generic production can be written as $A \rightarrow \alpha$, where ***A*** is the head and **α** is the body.

6. A set of productions $A \rightarrow \alpha 1$, $A \rightarrow \alpha_2, \dots$, $A \rightarrow \alpha_k$ with a common head *A* (call them *A-productions)*, may be written $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_k$. Call $\alpha_1, \alpha_2, \dots, \alpha_k$ the ***alternatives*** **for A**.

7. Unless stated otherwise, the head of the first production is the start symbol.

Example 2.2: Using these conventions, the grammar of Example 2.1 can be rewritten concisely as ***E*** $\rightarrow$ ***E + T | E - T | T***

T $\rightarrow$ ***T * F |T / F | F***

F $\rightarrow$ ***(E) | id*** ***… (2.6)***

The notational conventions tell us that *E,* T, and F are nonterminals, with *E* the start symbol. The remaining symbols are terminals.

2.2.3 Derivations

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
```

Rightmost derivations are sometimes called ***canonical*** **derivations**.

2.2.4 Parse Trees and Derivations

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace nonterminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the nonterminal *A* in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this *A* was replaced during the derivation.

For example, the parse tree for **- (id + id)** in Fig. 2.2, results from the derivation (2.8) as well as derivation (2.9).

The leaves of a parse tree are labeled by nonterminals or terminals and, read from left to right, constitute a sentential form, called the ***yield* or *frontier*** of the tree.

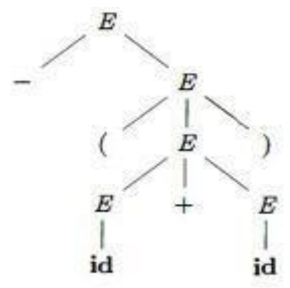


Figure 2.2: Parse tree for - (i d + id)

To see the relationship between derivations and parse trees, consider the sequence of parse trees constructed from the derivation is shown in Fig. 2.3.

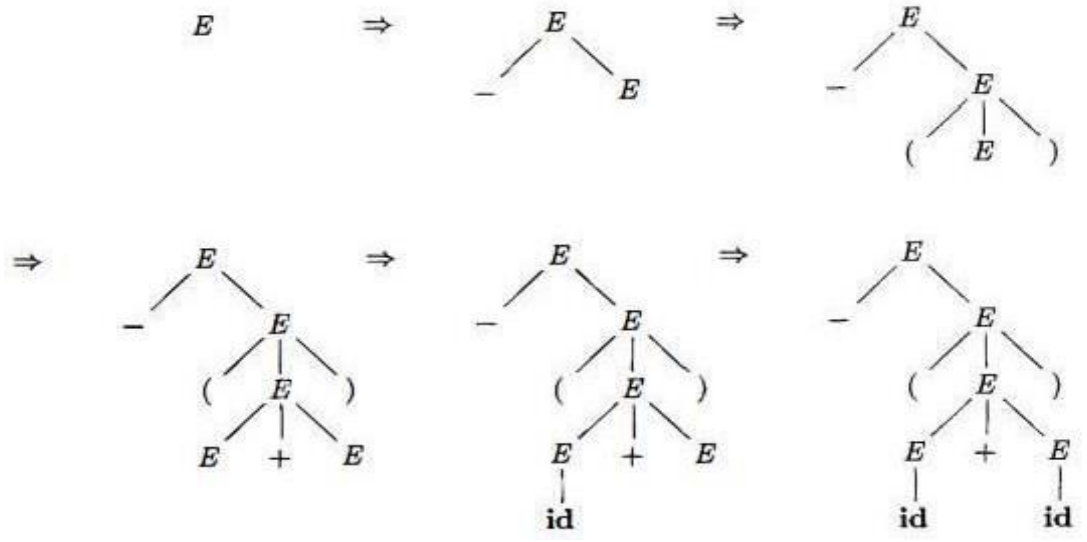


Fig: 2.3 Sequence of parse trees for derivation

In the first step of the derivation, *E =>-E.* To model this step, add two children, labeled - and *E,* to the root *E* of the initial tree. The result is the second tree.

In the second step of the derivation, *—E => -(E).* Consequently, add three children, labeled *(, E ,* and), to the leaf labeled *E* of the second tree, to obtain the third tree with yield -*(E).* Continuing in this fashion we obtain the complete parse tree as the sixth tree.

Since a parse tree ignores variations in the order in which symbols in sentential forms are replaced, there is a many-to-one relationship between derivations and parse trees. For example, both derivations (2.8) and (2.9), are associated with the same final parse tree of Fig. 2.3.

2.2.5 Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ***ambiguous***. Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Example 2.3: The arithmetic expression grammar permits two distinct leftmost derivations for the sentence **id + id * id:**

$$
\begin{array}{rll}
E & \Rightarrow & E + E \\
 & \Rightarrow & \mathbf{id} + E \\
 & \Rightarrow & \mathbf{id} + E * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
\qquad
\begin{array}{rll}
E & \Rightarrow & E * E \\
 & \Rightarrow & E + E * E \\
 & \Rightarrow & \mathbf{id} + E * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * E \\
 & \Rightarrow & \mathbf{id} + \mathbf{id} * \mathbf{id}
\end{array}
$$

The corresponding parse trees appear in Fig. 2.4

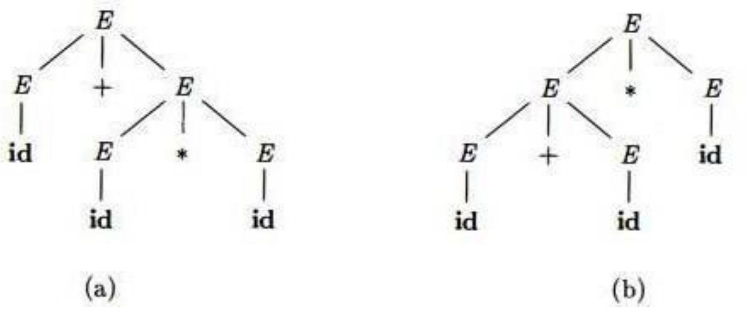


Fig. 2.4: Two parse trees for **id+id*id**

2.3 Writing a Grammar

Grammars are capable of describing most of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser.

2.3.1 Lexical Versus Syntactic Analysis

Everything that can be described by a regular expression can also be described by a grammar. There are several reasons for using regular expressions to define the lexical syntax of a language:

1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.
2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.
3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.
4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. **Regular expressions** are most useful for ***describing the structure of constructs such as identifiers, constants, keywords, and white space***. **Grammars**, on the other hand, are most useful for ***describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on***. These nested structures cannot be described by regular expressions.

2.3.2 Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "danglingelse" grammar:

stmt —> **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*
| other ….. **(2.7)**

Here **"other"** stands for any other statement. According to this grammar, the compound conditional statement

if *E1* **then** *S1* **else if** *E2* **then** *S2* **else** S3

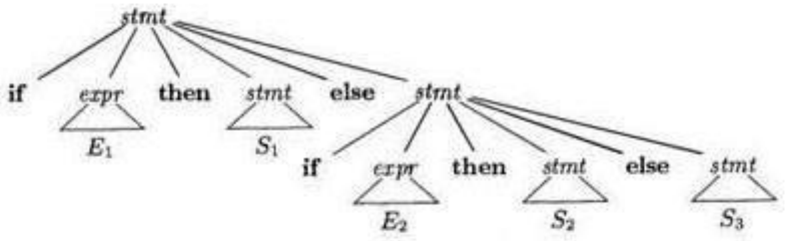


Figure 2.5: Parse tree for a conditional statement

has the parse tree shown in Fig. 2.5. Grammar (2.7) is ambiguous since the string

if *E1* **then if** *E2* **then** S1 **else** S2 … **(2.8)**

has the two parse trees shown in Fig. 2.6.

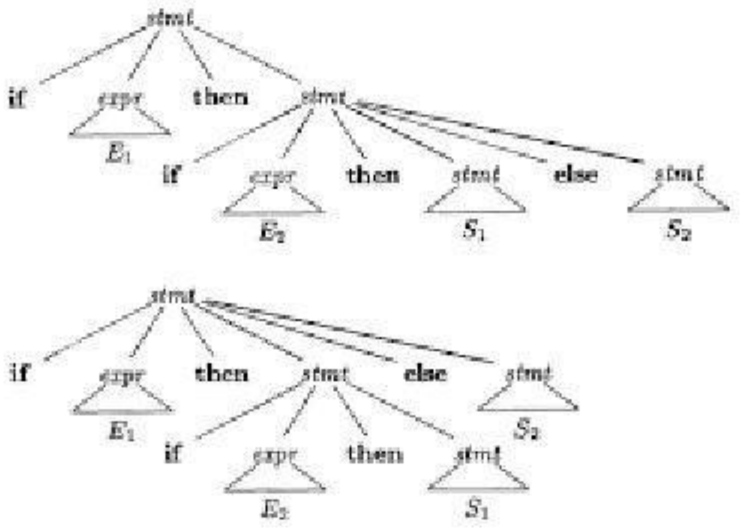


Figure 2.6: Two parse trees for an ambiguous sentence

In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, "Match each **else** with the closest unmatched **then."** This disambiguating rule can theoretically be incorporated directly into a grammar, but in practice it is rarely built into the productions.

Example 2.4: We can rewrite the dangling-else grammar (2.7) as the following unambiguous grammar. The idea is that a statement appearing between a **then** and an **else** must be "matched"; that is, the interior statement must not end with an unmatched or open **then.** A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement. Thus, we may use the grammar in Fig. 2.7. This grammar generates the same strings as the dangling-else grammar (2.10), but it allows only one parsing for string (2.11); namely,the one that associates each **else** with the closest previous unmatched **then.**

$$
\begin{array}{rcl}
\mathit{stmt} & \rightarrow & \mathit{matched\_stmt} \\
& | & \mathit{open\_stmt} \\
\mathit{matched\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{matched\_stmt} \\
& | & \textbf{other} \\
\mathit{open\_stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\
& | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{matched\_stmt} \textbf{ else } \mathit{open\_stmt}
\end{array}
$$

Figure 2.7: Unambiguous grammar for if-then-else statements

2.3.3 Elimination of Left Recursion

A grammar is ***left recursive*** if it has a nonterminal *A* such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string *α.* Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion.

Example 2.5: The non-left-recursive expression grammar (2.2), repeated here,

$$
\begin{array}{l}
E \rightarrow T\,E' \\
E' \rightarrow +\,T\,E' \,/\, \varepsilon \\
T \rightarrow F\,T' \\
T' \rightarrow *\,F\,T' \,/\, \varepsilon \\
F \rightarrow (E) \mid id
\end{array}
$$

is obtained by eliminating immediate left recursion from the expression grammar (421). The left-recursive pair of productions $E \rightarrow E + T \mid T$ are replaced by $E \rightarrow T\,E'$ and $E' \rightarrow +\,T\,E' \,/\, \varepsilon$. The new productions for *T* and *T'* are obtained similarly by eliminating immediate left recursion.

Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

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

where no β begins with an *A.* Then, replace the A-productions by

$$
\begin{array}{l}
A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A' \\
A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon
\end{array}
$$

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

$$
\begin{array}{ll}
S \rightarrow A\,a \mid b & \\
A \rightarrow A\,c \,/\, S\,d \mid \varepsilon & \quad \ldots (2.9)
\end{array}
$$

The nonterminal ***S*** is left recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive. Algorithm 2.1, below, systematically eliminates left recursion from a grammar.

Algorithm 2.1: Eliminating left recursion.

INPUT: Grammar *G* with no cycles or e-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 2.8 to *G.* Note that the resulting non-left-recursive grammar may have ε-productions.

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

Figure 2.8: Algorithm to eliminate left recursion from a grammar

Example 2.6: Let us apply Algorithm 2.1 to the grammar (2.9). Technically, the algorithm is not guaranteed to work, because of the ε-production, but in this case, the production *A*→ ε turns out to be harmless.
We order the nonterminals *S, A.* There is no immediate left recursion among the S-productions, so nothing happens during the outer loop for *i* = 1. For *i* = 2, we substitute for *S* in *A* —> *S d* to obtain the following A-productions.

A→ A c | A a d | b d | ε

Eliminating the immediate left recursion among these A-productions yields the following grammar.

S → A a | b
A → b d A' | A'
A' → c A' | a d A' | ε

2.3.4 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammarsuitable for predictive or top-down parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

stmt →**if** *expr* **then** *stmt* **else** *stmt*
| **if** *expr* **then** *stmt*

on seeing the input **if,** we cannot immediately tell which production to choose toexpand *stmt.* In general, if A → $\alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding *A* to *αA'.* Then, after seeing the input derived from α, we expand *A'* to β_1 or to β_2. That is, left-factored, the original productions become

A → αA'
A' → $\beta_1 \mid \beta_2$

Algorithm 2.2: Left factoring a grammar.
INPUT: Grammar *G.*
OUTPUT: An equivalent left-factored grammar.
METHOD: For each nonterminal *A,* find the longest prefix *α* common to two or more of its alternatives. If *α ≠ ε* i.e., there is a nontrivial common prefix — replace all of the A-productions *A* → $\alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \Upsilon$, where ϒ represents all alternatives that do not begin with *α,* by
A → *αA' |ϒ*

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

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example 2.7: The following grammar abstracts the "dangling-else" problem:

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$
$$E \rightarrow b \qquad \textit{… (2.10)}$$

Here, i, *t,* and e stand for if, then , and else; *E* and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

$$S \rightarrow i\ E\ t\ S\ S' \mid a$$
$$S' \rightarrow e\ S \mid \varepsilon$$
$$E \rightarrow b \qquad \textbf{… (2.11)}$$

Thus, we may expand S to *iEtSS'* on input i, and wait until iEtS has been seen to decide whether to expand *S'* to *eS* or to ε. Of course, these grammars are both ambiguous, and on input e, it will not be clear which alternative for *S'* should be chosen.

2.4 Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example 2.8: The sequence of parse trees in Fig. 2.9 for the input **id+id*id** is a top-down parse according to grammar (2.2), repeated here:

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

This sequence of trees corresponds to a leftmost derivation of the input.

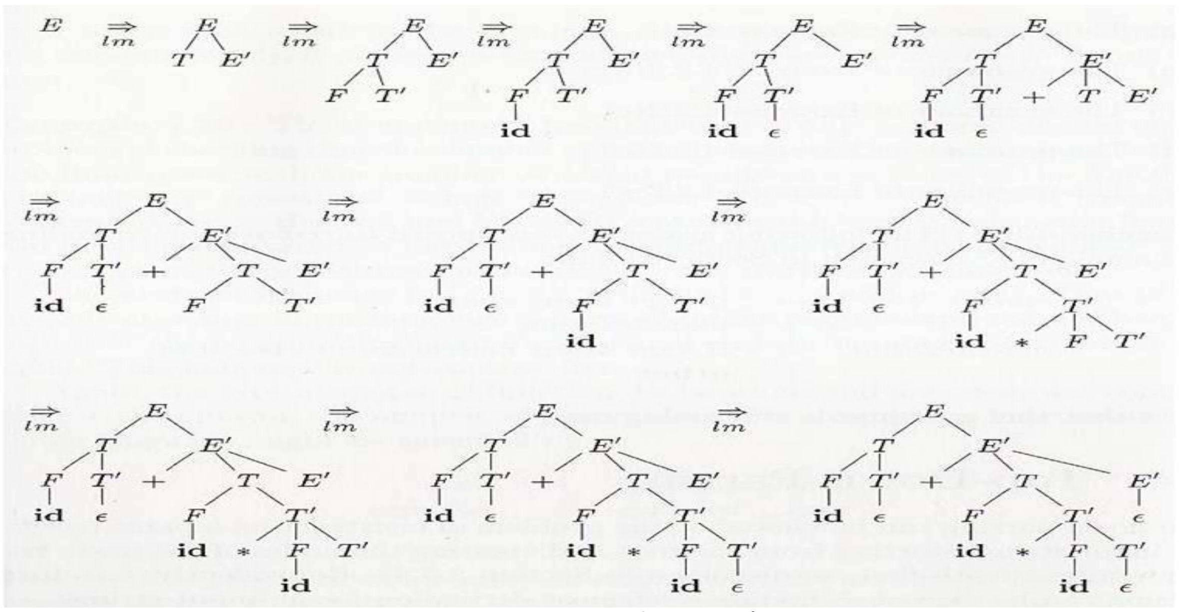


*Figure 2.9: Top-down parse for id + id * id*

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say *A.* Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

For example, consider the top-down parse in Fig. 2.9, which constructs a tree with two nodes labeled *E'.* At the first *E'* node (in preorder), the production *E' —> +TE'* is chosen; at the second *E'* node, the production *E' —> ε* is chosen. A predictive parser can choose between E'-productions by looking at the next input symbol.

The class of grammars for which we can construct predictive parsers looking *k* symbols ahead in the input is sometimes called the ***LL(k)* class**.

2.4.1 Recursive-Descent Parsing

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Pseudocode for a typical nonterminal appears in Fig. 2.10. Note that this pseudocode is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

```
void A() {
        Choose an A-production, $A \to X_1 X_2 \cdots X_k$;
        for ( $i$ = 1 to $k$ ) {
                if ( $X_i$ is a nonterminal )
                        call procedure $X_i()$;
                else if ( $X_i$ equals the current input symbol $a$ )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Figure 2.10: A typical procedure for a nonterminal in a top-down parser

General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently. Even for situations like natural language parsing, backtracking is not very efficient.

Example 2.9: Consider the grammar S → c A d
A → ab | a

To construct a parse tree top-down for the input string *w = cad,* begin with a tree consisting of a single node labeled *S,* and the input pointer pointing to c, the first symbol *of w. S* has only one production, so we use it to expand *S* and obtain the tree of Fig. 2.11(a). The leftmost leaf, labeled c, matches the first symbol of input *w,* so we advance the input pointer to a, the second symbol of *w,* and consider the next leaf, labeled *A.*

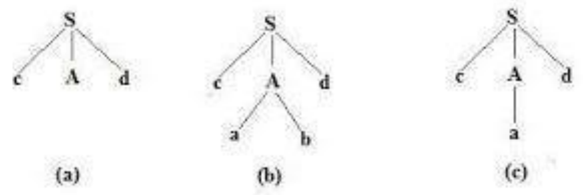


Figure 2.11: Steps in a top-down parse

Now, we expand *A* using the first alternative *A* → *a b* to obtain the tree of Fig. 2.1(b). We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol, and compare *d* against the next leaf, labeled *b.* Since *b* does not match *d,* we report failure and go back to *A* to see whether there is another alternative for *A* that has not been tried, but that might produce a match.

The second alternative for *A* produces the tree of Fig. 2.11(c). The leaf *a* matches the second symbol of *w* and the leaf *d* matches the third symbol. Since we have produced a parse tree for *w,* we halt and announce successful completion of parsing.

A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand a nonterminal *A,* we may eventually find ourselves again trying to expand *A* without having consumed any input.

2.4.2 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, **FIRST** and **FOLLOW,** associated with a grammar *G.*

- **FIRST Rules**:

Define ***FIRST(α),*** where ***α*** is any string of grammar symbols, to be the set of terminals that begin strings derived from ***α.*** If ***α***$=^{*}>$ε, then ε is also in **FIRST(α).** For example, *A*$=^{*}>$ *cϒ,* so c is in **FIRST**(A).

To compute FIRST*(X)* for all grammar symbols *X,* apply the following rules until no more terminals or e can be added to any FIRST set.

1. If *X* is a terminal, then FIRST(X) = *{X}.*
2. If *X* is a nonterminal and *X* —> $Y_1, Y_2 \ldots Y_k$ is a production for some *k* > =1, then place *a* in FIRST(X) if for some *i, a* is in FIRST(Y_i), and ε is in all of FIRST(Y_1),... , FIRST(Y_{i-1}); that is, $Y_1 \ldots Y_{i-1} =^{*}> \varepsilon$. If ε is in FIRST (Yj) for all *j* = 1,2,... , *k,* then add ε to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ε, then we add nothing more to FIRST(X), but if Yi $=^{*}>$ e, then we add FIRST(Y_2) , and so on.
3. If *X* -> ε is a production, then add ε to FIRST(X).

Now, we can compute FIRST for any string $X_1 X_2 \ldots X_n$ as follows. Add to FIRST($X_1 X_2 \ldots X_n$) all non-ε symbols of FIRST(X_1). Also add the non-ε symbols of FIRST(X_2), if ε is in FIRST(X_1); the non-ε symbols of FIRST(X_3) , if ε is in FIRST(X_1) and FIRST(X_2) ; and so on. Finally, add ε to FIRST($X_1 X_2 \ldots X_n$) if, for all i, ε is in FIRST (X_i).

- **FOLLOW Rules**

Define ***FOLLOW****(A),* for nonterminal *A,* to be the set of terminals *a* that can appear immediately to the right of *A* in some sentential form; that is, the set of terminals *a* such that there exists a derivation of the form $S =^{*}> \alpha A a \beta$, for some *α* and β. Note that there may have been symbols between *A* and a, at some time during the derivation, but if so, they derived ε and disappeared. In addition, if *A* can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A); recall that $ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.
1. Place $ in FOLLOW(S), where *S* is the start symbol, and $ is the input right endmarker.
2. If there is a production *A*→ *αB*β, then everything in FIRST(β) except ε is in FOLLOW(S).
3. If there is a production *A*→ *αB,* or a production *A*→ *αB*β, where FIRST(β) contains ε, then everything in **FOLLOW**(A) is in **FOLLOW**(B) .

Example 2.10: Consider again the non-left-recursive grammar. Then:

E→ T E'
E'→ + T E' / ε
T→ F T'
T'→ * F T' / ε
F→ (E) | id

1. **FIRST(F) = FIRST(T) = FIRST(E) = {(,id}.** To see why, note that the two productions for *F* have bodies that start with these two terminal symbols, id and the left parenthesis. *T* has only one production, and its body starts with *F.* Since *F* does not derive e, FIRST(T) must be the same as FIRST *(F).* The same argument covers FIRST *(E).*

2. **FIRST(E') = {+, ε}.** The reason is that one of the two productions for *E'* has a body that begins with terminal +, and the other's body is ε. whenever a nonterminal derives e, we place e in F I R S T for that nonterminal.

3. **FIRST(T') = {*,ε}.** The reasoning is analogous to that for FIRST(E').

4. **FOLLOW(E) = FOLLOW(E') = {),$}.** Since *E* is the start symbol, FOLLOW(E) must contain $. The production body (*E)* explains why the right parenthesis is in FOLLOW(E). For *E',* note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

5. **FOLLOW(T) = FOLLOW(T') = { + ,) , $}.** Notice that T appears in bodies only followed by *E'.* Thus, everything except ε that is in FIRST(E') must be in FOLLOW(T); that explains the symbol +. However, since FIRST(E') contains ε (i.e., *E'* =>* ε), and *E'* is the entire string following *T* in the bodies of the E-productions, everything in FOLLOW(E) must alsobe in FOLLOW(T). That explains the symbols $ and the right parenthesis. As for T', sinceit appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

6. **FOLLOW(F) = { + , * ,) , $ }** . The reasoning is analogous to that for T in point (5).

2.4.3 LL(1) Grammars

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the " 1 " for using one input symbol of lookahead at each step to make parsing actiondecisions.

A grammar *G* is LL(1) if and only if whenever *A* —> *α*| β are two distinct productions of *G,* the following conditions hold:

1. For no terminal *a* do both *α* and β derive strings beginning with *a.*
2. At most one of *α* and β can derive the empty string.
3. If β=*>ε, then *α* does not derive any string beginning with a terminal in FOLLOW (A). Likewise, if α=*>ε, then β does not derive any string beginning with a terminal in FOLLOW(A).

The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets. The third condition is equivalent to stating that if ε is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ε is in FIRST(α).

Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol. Flow-of-control constructs, with their distinguishing keywords, generally satisfy the LL(1) constraints. For instance, if we have the productions

stmt **-> if (** *expr) stmt* **else** *stmt*
| **while (** *expr) stmt*
| *{ stmtJist }*

then the keywords **if, while,** and the symbol { tell us which alternative is the only one that could possibly succeed if we are to find a statement.

The next algorithm collects the information from FIRST and FOLLOW sets into a **predictive parsing table** *M[A,a],* a two-dimensional array, where A is a nonterminal, and a is a terminal or the symbol $, the input endmarker. The algorithm is based on the following idea: the production *A —> α* is chosen if the next input symbol *a* is in FIRST(α). The only complication occurs when *α = ε* or, more generally, *α =*> ε.* In this case,we should again choose *A → α,* if the current input symbol is in FOLLOW(A), or if the $ on the input has been reached and $ is in FOLLOW (A).

Algorithm 2.3 : Construction of a predictive parsing table.
INPUT: Grammar *G.*
OUTPUT: Parsing table *M.*
METHOD: For each production A -> *α* of the grammar, do the following:

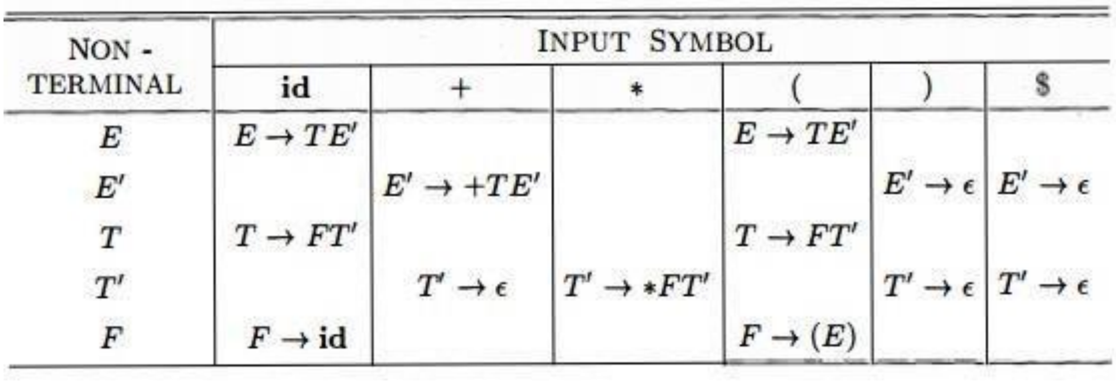
1. For each terminal *a* in FIRST(α), add A → α to M[A, *a].*
2. If ε is in FIRST(α), then for each terminal b in FOLLOW(A), add A → *α* to *M[A,b].* If ε is in FIRST (α) and $ is in FOLLOW(A), A → α to M[A, *$*]as well.

If, after performing the above, there is no production at all in *M[A,* a], then set *M[A, a]* to **error** (which we normally represent by an empty entry in the table).

Example 2.11: For the expression grammar

E → T E'
E' → + T E' / ε
T → F T'
T' → * F T' / ε
F → (E) | id

Algorithm 2.3 produces the parsing table in Fig. 2.12. Blanks are error entries; non-blanks indicate a production with which to expand a nonterminal.



NON - TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \text{id}$			$F \to (E)$		

Figure2.12: Parsing table M for Example 2.11

- Consider production ***E → T E'.*** Since
  FIRST(TE') = FIRST(T**) = {(,id}**
  this production is added to *M[E,(]* and *M[E, id].*

- Production ***E' → + T E'*** is added to *M[E',+]* since FIRST(+TE') = {+}.
- Since FOLLOW(E') = {) , $}, production ***E' → ε*** is added to *M[E',)*] and *M[E', $].*

2.4.4 Nonrecursive Predictive Parsing

A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If *w* is the input that has been matched so far, then the stack holds a sequence of grammar symbols *α* such that

$$S \overset{*}{\underset{lm}{\Rightarrow}} w\alpha$$

The table-driven parser in Fig. 2.13 has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm 2.3, and an output stream. The input buffer contains the string to be parsed, followed by the endmarker $. We reuse the symbol $ to mark the bottom of t he stack, which initially contains the start symbol of the grammar on top of $.

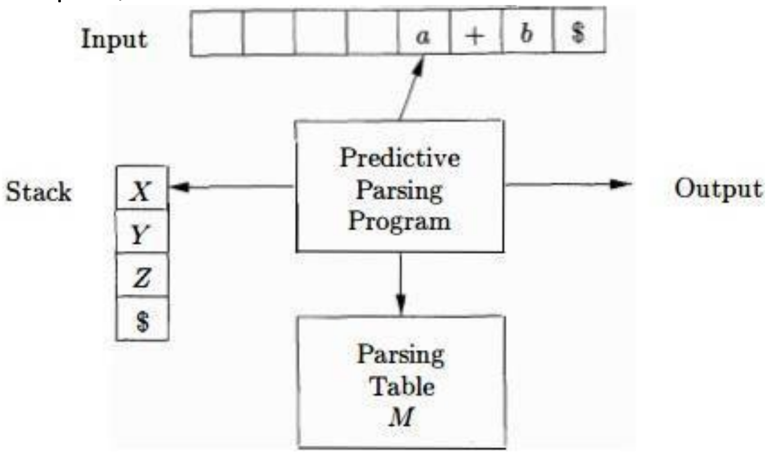


Figure 2.13: Model of a table-driven predictive parser

The parser is controlled by a program that considers *X,* the symbol on top of the stack, and *a,* the current input symbol. If *X* is a nonterminal, the parser chooses an X- production by consulting entry *M[X, a]* of the parsing table *M.* Otherwise, it checks for a match between the terminal *X* and current input symbol *a.*

The behavior of the parser can be described in terms of its ***configurations,*** which give the stack contents and the remaining input.

Algorithm 2.4 : Table-driven predictive parsing.

INPUT: A string *w* and a parsing table *M* for grammar *G.*

OUTPUT: If *w* is in *L(G),* a leftmost derivation of *w;* otherwise, an error indication.

METHOD: Initially, the parser is in a configuration with *w$* in the input buffer and the start symbol *S* of *G* on top of the stack, above $. The program uses the predictive parsing table *M* to produce a predictive parse for the input.

```
Let a be the first symbol of w;
Let X to the top stack symbol;
while ( X≠ $ ) { /* stack is not empty */
if ( X = a ) pop the stack and let a be the next symbol of w;
else if ( X is a terminal ) error();
else if ( M[X,a] is an error entry ) error();
else if ( M[X,a] = X -> Y1Y2 •••Yk) {
output the production X -> Y1 Y2 • • • Yk;
pop the stack;
push Yk, YK-1,... ,Yi onto the stack, with Y1 on top;
}
let X to the top stack symbol;
}
```

Example 2.12: Consider grammar

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

On input **id + id * id,** the nonrecursive predictive parser of Algorithm 2.4 makes the sequence of moves in Fig. 2.14. These moves correspond to a leftmost derivation

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $T' \rightarrow \epsilon$
$\mathbf{id}$	$+ TE'\$$	$+ \mathbf{id} * \mathbf{id}\$$	output $E' \rightarrow + TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \rightarrow FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \rightarrow * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \rightarrow \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \rightarrow \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \rightarrow \epsilon$

Figure 2.14: Moves made by a predictive parser on input id + id * id

2.5 Bottom-Up Parsing

A bottom-up parsing constructs a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top). The sequence of tree snapshots in Fig. 2.15 illustrates a bottom-up parse of the token stream **id * id,** with respect to the expression grammar. ***E → E + T / T***

T → T * F / F

F → (E) / id

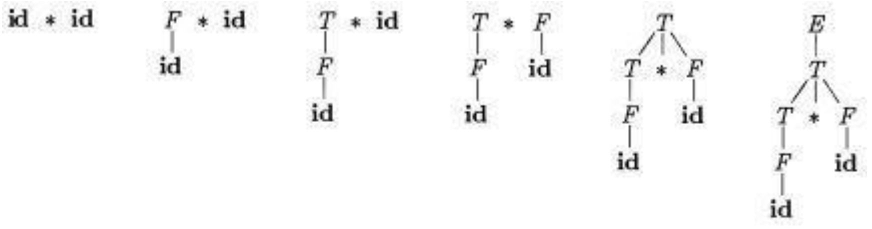


Figure 2.15: A bottom-up parse for id * id

2.5.1 Reductions

A bottom-up parsing is the process of "reducing" a string *w* to the start symbol of the grammar. At each ***reduction*** step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

Example 2.13: The snapshots in Fig. 2.15 illustrate a sequence of reductions; the grammar is the expression grammar. The reductions will be discussed in terms of the sequence of strings

id * id, F *** id ,** T *** id,** *T * F, T, E*

The strings in this sequence are formed from the roots of all the subtrees in the snapshots.

The sequence starts with the input string **id*id.** The first reduction produces *F**id*** by reducing the leftmost **id** to *F,* Using the production *F* → **id.** The second reduction produces *T* *** id** by reducing *F* to *T.*

Now, we have a choice between reducing the string T, which is the body of *E* → *T,* and the string consisting of the second **id,** which is the body of *F* → **id.** Rather than reduce *T* to *E,* the second **id** is reduced to T, resulting in the string *T * F.* This string then reduces to *T.* The parse completes with the reduction of *T* to the start symbol *E.*

By definition, a reduction is the reverse of a step in a derivation. The goal of bottom-up parsing is therefore to construct a derivation in reverse. The following derivation corresponds to the parse in Fig. 2.15:

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

This derivation is in fact a rightmost derivation.

2.5.2 Handle Pruning

Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse. Informally, a "handle" is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

For example, adding subscripts to the tokens **id** for clarity, the handles during the parse of $\textbf{id}_1 * \textbf{id}_2$ according to the expression grammar **(2.1)** are as in Fig. 2.16. Although T is the body of the production $E \rightarrow T$, the symbol T is not a handle in the sentential form $T * id_2$. If T were indeed replaced by E, we would get the string $E * \textbf{id}_2$, which cannot be derived from the start symbol E. Thus, the leftmost substring that matches the body of some production need not be a handle.

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

Figure 2.16: Handles during a parse of id1 * id2

A rightmost derivation in reverse can be obtained by "handle pruning." That is, we start with a string of terminals *w* to be parsed.

4.5.3 Shift-Reduce Parsing

Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds the rest of the string to be parsed. The handle always appears at the top of the stack just before it is identified as the handle.

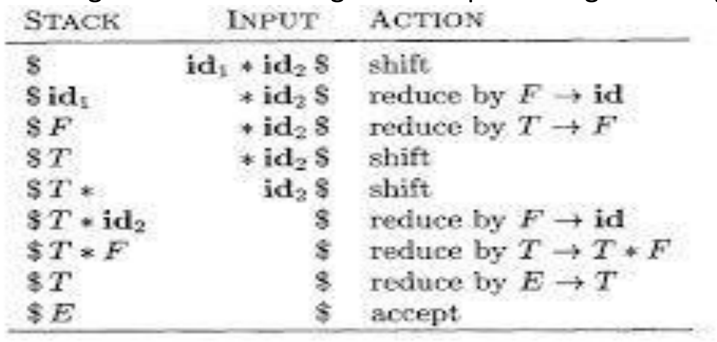
We use $ to mark the bottom of the stack and also the right end of the input. Conventionally, when discussing bottom-up parsing, we show the top of the stack on the right, rather than on the left as we did for top-down parsing. Initially, the stack is empty, and the string *w* is on the input, as follows:

STACK	INPUT
$	w $

During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

STACK	INPUT
$ S	$

Upon entering this configuration, the parser halts and announces successful completion of parsing. Figure 2.17 steps through the actions a shift-reduce parser might take in parsing the input string **id_1 * id_2** according to the expression grammar (2.1).



STACK	INPUT	ACTION
$\$$	$\mathbf{id}_1 * \mathbf{id}_2 \$$	shift
$\$\,\mathbf{id}_1$	$* \mathbf{id}_2 \$$	reduce by $F \to \mathbf{id}$
$\$\,F$	$* \mathbf{id}_2 \$$	reduce by $T \to F$
$\$\,T$	$* \mathbf{id}_2 \$$	shift
$\$\,T *$	$\mathbf{id}_2 \$$	shift
$\$\,T * \mathbf{id}_2$	$\$$	reduce by $F \to \mathbf{id}$
$\$\,T * F$	$\$$	reduce by $T \to T * F$
$\$\,T$	$\$$	reduce by $E \to T$
$\$\,E$	$\$$	accept

Figure 2.17: Configurations of a shift-reduce parser on input id1*id2

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

1. ***Shift***. Shift the next input symbol onto the top of the stack.
2. ***Reduce***. The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
3. **Accept**. Announce successful completion of parsing.
4. ***Error***. Discover a syntax error and call an error recovery routine.

2.6 Introduction to LR Parsing:

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where

- L stands for left-to-right scanning of the input stream;
- R stands for the construction of right-most derivation in reverse, and
- k denotes the number of lookahead symbols to make decisions.

There are three widely used algorithms available for constructing an LR parser:

- **SLR() – Simple LR Parser**:
  - Works on smallest class of grammar
  - Few number of states, hence very small table
  - Simple and fast construction
- **CLR() – Canonical LR Parser**:
  - Works on complete set of LR(1) Grammar
  - Generates large table and large number of states
  - Slow construction
- **LALR() – Look-Ahead LR Parser**:
  - Works on intermediate size of grammar
  - Number of states are same as in SLR(1)

2.6.1 Why LR Parsers?

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written.
- The LR-parsing method is the most general non backtracking shift-reduce parsing method known.

- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The principal drawback of the LR method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. A specialized tool, an LR parser generator, is needed. Most commonly used is **Yacc**. Such a generator takes a context-free grammar and automatically produces a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

UNIT – III

More Powerful LR parser (LR1, LALR) Using Armigers Grammars Equal Recovery in Lr parser Syntax Directed Transactions Definition, Evolution order of SDTS Application of SDTS. Syntax Directed Translation Schemes.

3.1. Simple LR (SLR)

1 Items and the LR(0) Automaton

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

$$A \rightarrow \cdot XYZ$$
$$A \rightarrow X \cdot YZ$$
$$A \rightarrow XY \cdot Z$$
$$A \rightarrow XYZ \cdot$$

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

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example, the item $A \rightarrow \cdot XYZ$ indicates that we hope to see a string derivable from *XYZ* next on the input. Item $A \rightarrow X \cdot YZ$ indicates that we have just seen on the input a string derivable from *X* and that we hope next to see a string derivable from *YZ.* Item $A \rightarrow XYZ \cdot$ indicates that we have seen the body *XYZ* and that it may be time to reduce *XYZ* to *A.*

One collection of sets of LR(0) items, called the ***canonical*** **LR(0)** collection, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an ***LR(0) automaton****.*

In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection. The automaton for the expression grammar (2.1), shown in Fig. 2.18, will serve as the running example for discussing the canonical LR(0) collection for a grammar.

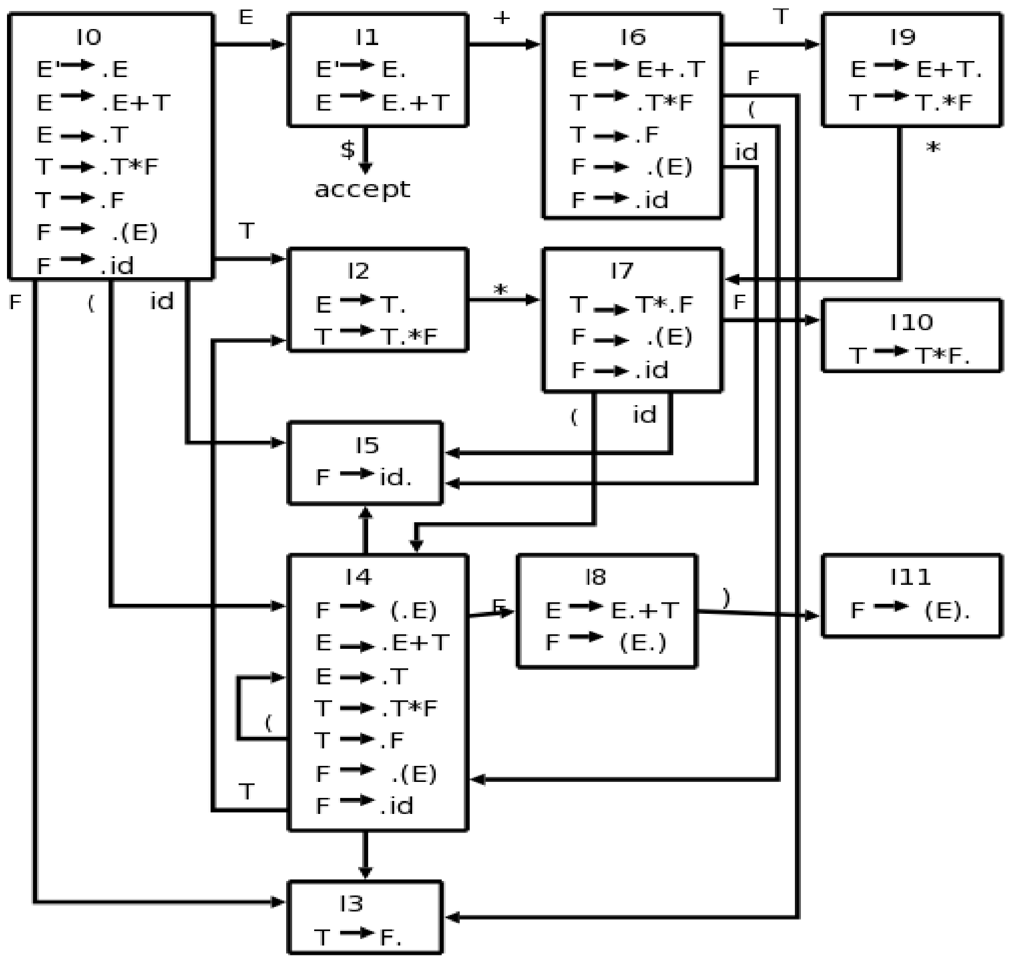


Figure 2.18: LR(O) automaton for the expression grammar (4.1)

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

- **Closure of Item Sets**

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, add every item in **I** to CLOSURE(**I**).
2. If *A* → α.*B*β is in CLOSURE(**I**) and *B* → ϒ is a production, then add the item *B* → .ϒ to CLOSURE(**I**), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(**I**).

The closure can be computed as .

```
function closure ( I )
begin
          J := I;
          repeat
                    for each item A → α.Bβ in J and each production
                              B→γ of G such that B→.γ is not in J do
                                        add B→.γ to J
                    until no more items can be added to J
end
```

- **The Function GOTO**

The second useful function is GOTO *(I, X)* where *I* is a set of items and X is a grammar symbol. GOTO (I , *X)* is defined to be the closure of the set of all items *[A* → *αX*·β*]* such that *[A* → *α*·*X*β*]* is in **I***.* Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I,X) specifies the transition from the state for I under input *X.*

The algorithm to construct *C,* the canonical collection of sets of LR(0) items for an augmented grammar *G'* is given by:

```
void items(G') {
          C = CLOSURE({[S' -> S]});
        repeat
            for ( each set of items I in C )
                  for ( each grammar symbol X )
                            if ( GOTO (I, X) is not empty and not in C )
                                  add GOTO(I, X) to C;
      until no new sets of items are added to C on a round;
}
```

The LR-Parsing Algorithm

A schematic of an LR parser is shown in Fig. 2.20. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (ACTION and GOTO). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. Where a shift-reduce parser would shift a symbol, an LR parser shifts a *state.* Each state summarizes the information contained in the stack below it.

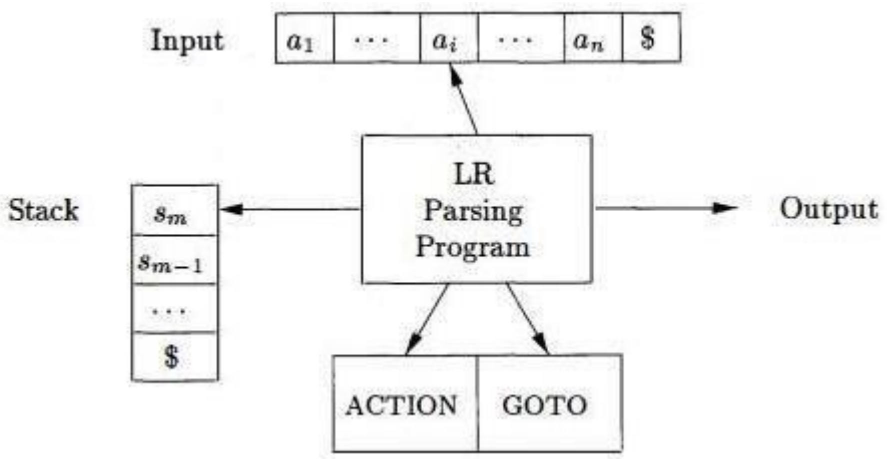


Figure 2.20: Model of an LR parser

The stack holds a sequence of states, $s_0s_1...s_m$, where s_m is on top. In the SLR method, the stack holds states from the LR(0) automaton; the canonical- LR and LALR methods are similar. By construction, each state has a corresponding grammar symbol. Recall that states correspond to sets of items, and that there is a transition from state *i* to state *j* if $GOTO(I_i,X) = I_j$. All transitions to state *j* must be for the same grammar symbol *X.* Thus, each state, except the start state 0, has a unique grammar symbol associated with it.

- **Structure of the LR Parsing Table**

The parsing table consists of two parts: a parsing-action function ACTION and a goto function GOTO.

1. The ACTION function takes as arguments a state *i* and a terminal a (or \$, the input endmarker). The value of ACTION[i,a] can have one of four forms:
   (a) Shift *j,* where *j* is a state. The action taken by the parser effectively shifts input *a* to the stack, but uses state *j* to represent *a.*
   (b) Reduce *A*→ β*.* The action of the parser effectively reduces β on the top of the stack to head *A.*
   (c) Accept. The parser accepts the input and finishes parsing.
   (d) Error. The parser discovers an error in its input and takes some corrective action.
2. We extend the GOTO function, defined on sets of items, to states: if GOTO[I_i, *A] =* I_j*,* then GOTO also maps a state *i* and a nonterminal *A* to state *j .*

- **LR-Parser Configurations**

To describe the behavior of an LR parser, it helps to have a notation representing the complete state of the parser: its stack and the remaining input. A *configuration* of an LR parser is a pair:

$$(s_0s_1...s_m,\ a_ia_{i+1}...a_n\$)$$

where the first component is the stack contents (top on the right), and the second component is the remaining input. This configuration represents the right-sentential form

$$X_1X_2...X_ma a_ia_{i+1}...a_n$$

in essentially the same way as a shift-reduce parser would; the only difference is that instead of grammar symbols, the stack holds states from which grammar symbols can be recovered. That is, X_i is the grammar symbol represented by state s_i. Note that s_0, the start state of the parser, does not represent a grammar symbol, and serves as a bottom-of-stack marker, as well as playing an important role in the parse.

- **Behavior of the LR Parser**

The next move of the parser from the configuration above is determined by reading a_i, the current input symbol, and s_m , the state on top of the stack, and then consulting the entry ACTION[s_m , a_i**]** in the parsing action table. The configurations resulting after each ofthe four types of move are as follows:

1. If ACTION[s_m , a_i] = shift ***s,*** the parser executes a shift move; it shifts the next state ***s*** onto the stack, entering the configuration

$$(s_0 s_1 \ldots s_m,\ a, a_{i+1} \ldots a_n \$)$$

The symbol a_i need not be held on the stack, since it can be recovered from ***s,*** if needed (which in practice it never is). The current input symbol i s now a_{i+1}.

2. If ACTION s_m , a_i***]*** = reduce $A \rightarrow \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 s_1 \ldots s_{m-r} s,\ a, a_{i+1} \ldots a_n \$)$$

where *r* is the length of **β,** and ***s = GOTO[s_{m-r},*** *A].* Here the parser first popped r state symbols off the stack, exposing state ***s_{m-r}***. The parser then pushed s, the entry for ***GOTO[s_{m-r},*** *A],* onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \ldots X_m$, the sequence of grammar symbolscorresponding to the states popped off the stack, will always match β*,* the right side of the reducing production.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production.

3. If ACTlON[s_m , a_i] = accept, parsing is completed.

4. If ACTlON[s_m , a_i] = error, the parser has discovered an error and calls an error recovery routine.

The LR-parsing algorithm is summarized below. All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the ACTION and GOTO fields of the parsing table.

Algorithm : LR-parsing algorithm.

INPUT: An input string *w* and an LR-parsing table with functions ACTION and GOTO for a grammar *G.*

OUTPUT: If *w* is in *L(G),* the reduction steps of a bottom-up parse for *w;* otherwise, an error indication.

METHOD: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and *w$* in the input buffer. The parser then executes the program.

```
let a be the first symbol of w$;
    while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
            push t onto the stack;
            let a be the next input symbol;
```

```
                } else if ( ACTION [s, a] = reduce A →β) {
                        pop |β| symbols off t he stack;
                        let state t now be on top of the stack;
                        push GOTO[t, A] onto the stack;
                        output the production A →β;
                } else if ( ACTION[s,a] = accept ) break; /* parsing is done */
                else call error-recovery routine;
        }
```

Example 2.15: Figure 2.21 shows the **ACTION** and **GOTO** functions of an LR-parsing table for the expression grammar productions numbered:

1) E→ E + T / T	***4) T→ F***
2) E→ T	***5) F→ (E) / id***
3) T→ T * F / F	***6) F→ id***

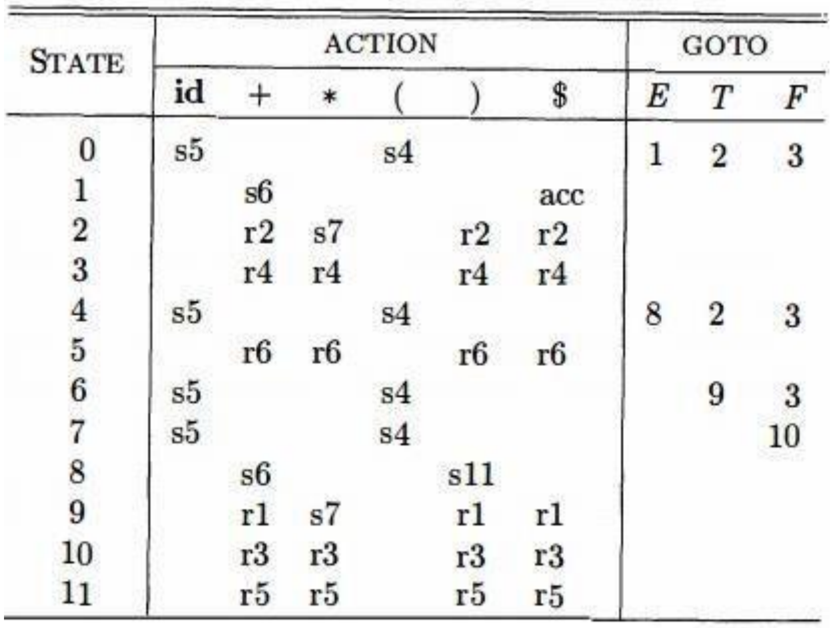
The codes for the actions are:

1. *si* means shift and stack state i,

2. rj means reduce by the production numbered *j,*

3. acc means accept,

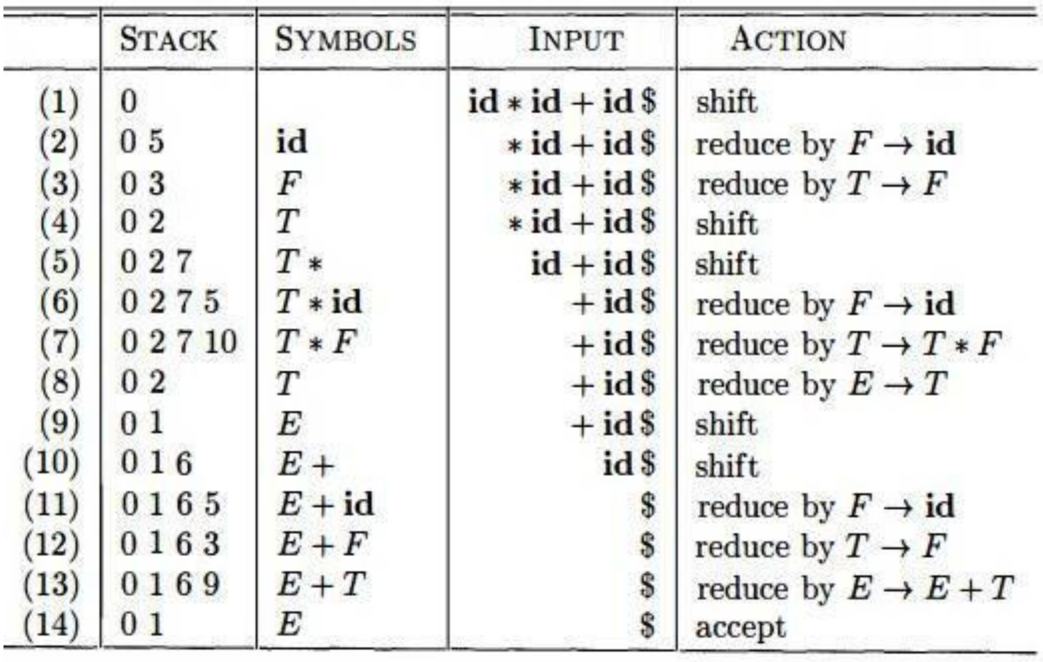
4. blank means error.



STATE	ACTION						GOTO		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Figure 2.21: Parsing table for expression grammar

On input **id * id + id,** the sequence of stack and input contents is shown in Fig. 2, 22. Also shown for clarity, are the sequences of grammar symbols corresponding to the states held on the stack. For example, at line (1) the LR parser is in state 0, the initial state with no grammar symbol, and with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 2.21 is s5, meaning shift by pushing state 5. That is what has happened at line (2): the state symbol 5 has been pushed onto the stack, and **id** has been removed from the input.



	STACK	SYMBOLS	INPUT	ACTION
(1)	0		**id** $*$ **id** $+$ **id** $	shift
(2)	0 5	**id**	$*$ **id** $+$ **id** $	reduce by $F \rightarrow$ **id**
(3)	0 3	F	$*$ **id** $+$ **id** $	reduce by $T \rightarrow F$
(4)	0 2	T	$*$ **id** $+$ **id** $	shift
(5)	0 2 7	$T *$	**id** $+$ **id** $	shift
(6)	0 2 7 5	$T *$ **id**	$+$ **id** $	reduce by $F \rightarrow$ **id**
(7)	0 2 7 10	$T * F$	$+$ **id** $	reduce by $T \rightarrow T * F$
(8)	0 2	T	$+$ **id** $	reduce by $E \rightarrow T$
(9)	0 1	E	$+$ **id** $	shift
(10)	0 1 6	$E +$	**id** $	shift
(11)	0 1 6 5	$E +$ **id**	$	reduce by $F \rightarrow$ **id**
(12)	0 1 6 3	$E + F$	$	reduce by $T \rightarrow F$
(13)	0 1 6 9	$E + T$	$	reduce by $E \rightarrow E + T$
(14)	0 1	E	$	accept

Figure 4.38: Moves of an LR parser on id * id + id

Then, * becomes the current input symbol, and the action of state 5 on input * is to reduce by *F* **id.** One state symbol is popped off the stack. State 0 is then exposed. Since the goto of state 0 on *F* is 3, state 3 is pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly.

3.2 More Powerful LR Parsers

3.2.1 Canonical LR(1) Items

The most general technique for constructing an LR parsing table from a grammar called "**canonical-LR**" or just "**LR**" method, which makes full use of the lookahead symbol(s). This method uses a large set of items, called the LR(1) items.

Formally, we say LR(1) item *[A* → *α. β, a]* is *valid* for a viable prefix ϒ if there is a derivation $S =_{rm}^{*}> {}^{\delta}A\omega = {}_{rm}^{*}> {}^{\delta}\alpha\beta\omega$, where

1. ϒ = $^{\delta}a$, and
2. Either *a* is the first symbol of *w,* or *w* is ε and *a* is $.

Example 3.1: Let us consider the grammar

S → B B

B → a B | b

There is a rightmost derivation $S =_{rm}^{*}> aaBab \;=_{rm}> aaaBab$. We see that item [B → *α. β, a*] is valid for a viable prefix ϒ = *aaa* by letting $^{\delta}$ *= aa, A = B, w = ab, a = a,* and β= B in the above definition. There is also a rightmost derivation $S=_{rm}^{*}> BaB=_{rm}> BaaB$. From this

derivation we see that item *[B*→ a.B,$] is valid for viable prefix *Baa.*

Constructing LR(1) Sets of Items

```
SetOfItems CLOSURE(I) {
    repeat
        for ( each item [A → α·Bβ, a] in I )
            for ( each production B → γ in G' )
                for ( each terminal b in FIRST(βa) )
                    add [B → ·γ, b] to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for ( each item [A → α·Xβ, a] in I )
        add item [A → αX·β, a] to set J;
    return CLOSURE(J);
}

void items(G') {
    initialize C to CLOSURE({[S' → ·S, $]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

Figure 3.1: Sets-of-LR(l)-items construction for grammar G'

Algorithm 3.1: Construction of the sets of LR(1) items.
INPUT: An augmented grammar *G'.*
OUTPUT: The sets of LR(1) items that are the set of items valid for one or more viable prefixes of *G'.*
METHOD: The procedures **CLOSURE** and **GOTO** and the main routine *items* for constructing the sets of items were shown in Fig. 3.1

Example 3.2: Consider the following augmented grammar.

$$S' \rightarrow S$$
$$S \rightarrow CC \qquad \text{.... (3.1)}$$
$$C \rightarrow cC \mid d$$

We begin by computing the closure of *{[S'* → .S,$]}. To close, we match the item *[S'* → .S,$] with the item *[A* → *α. B*β*, a]* in the procedure **CLOSURE.** That is, *A* = *S',* **a** = ε, B = *S,* β = ε, and **a** = $. Function **CLOSURE** tells us to add *[B*→ .ϒ,b] for each production *B* -> ϒ and terminal b in **FIRST** (βa). In terms of the present grammar, B → ϒ must be S → CC, and since β is ε and
a is $, *b* may only be $. Thus we add *[S*→ .CC,$].

We continue to compute the closure by adding all items [C → .ϒ, b] for b in **FIRST**(C$) . That is, matching *[S* → .CC, $] against [*A* → *α. B*β*, a*], we have A = S, α = ε, *B* = C, β = C, and a = $. Since C does not derive the empty string, **FIRST**(C$) = **FIRST**(C) . Since **FIRST(C**) contains terminals c and d, we add items [C → *.cC,c], [C* → *.cC,* d], [C → .d, c] and [C → .d, d]. None of the new items has a nonterminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial set of items is

I_0 :
S → .S,$
S → .CC. $
C → .cC, c/d
C → .d, c/d

We now give the LR(1) sets of items construction. Figure 3.2 shows the ten sets of items with their goto's.

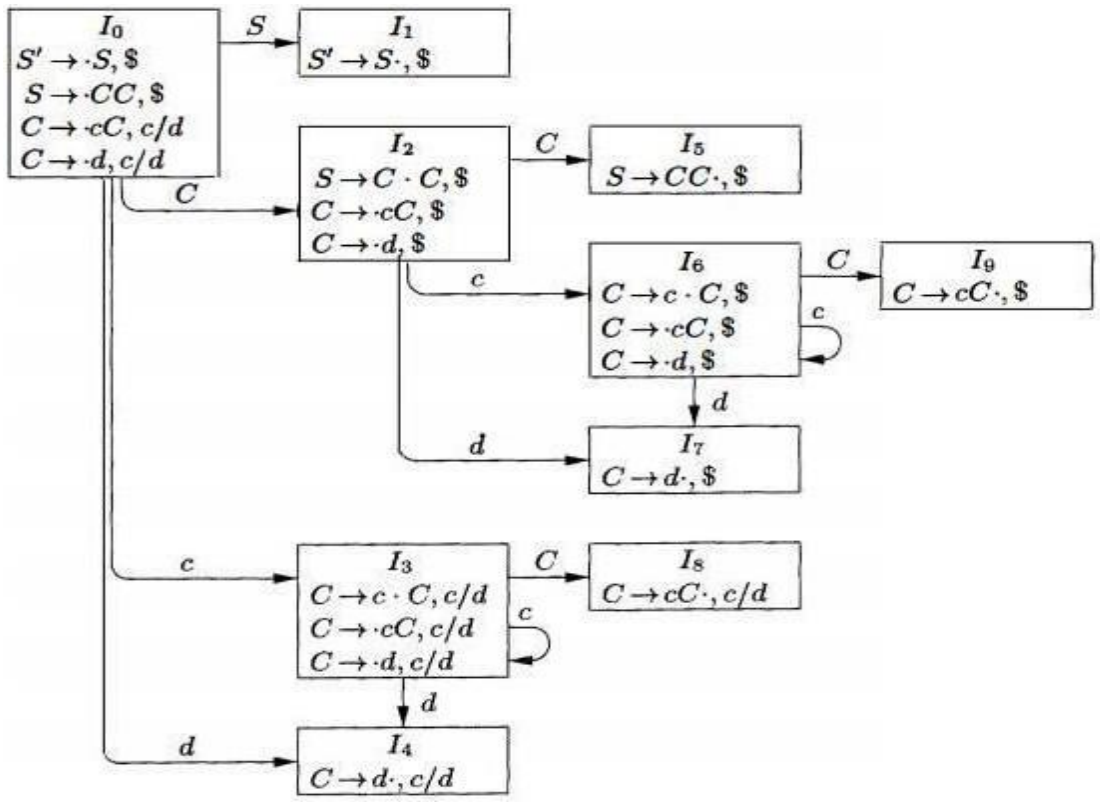

I_0
$S' \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \$$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$
I_1
$S' \rightarrow S\cdot, \$$
I_2
$S \rightarrow C \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$
I_5
$S \rightarrow CC\cdot, \$$
I_6
$C \rightarrow c \cdot C, \$$
$C \rightarrow \cdot cC, \$$
$C \rightarrow \cdot d, \$$
I_9
$C \rightarrow cC\cdot, \$$
I_7
$C \rightarrow d\cdot, \$$
I_3
$C \rightarrow c \cdot C, c/d$
$C \rightarrow \cdot cC, c/d$
$C \rightarrow \cdot d, c/d$
I_8
$C \rightarrow cC\cdot, c/d$
I_4
$C \rightarrow d\cdot, c/d$
S
C
c
d

Figure 3.2: The GOTO graph for grammar (3.1)

Canonical LR(1) Parsing Tables

The rules for constructing the LR(1) ACTION and GOTO functions from the sets of LR(1) items. These functions are represented by a table, as before. The only difference is in the values of the entries.

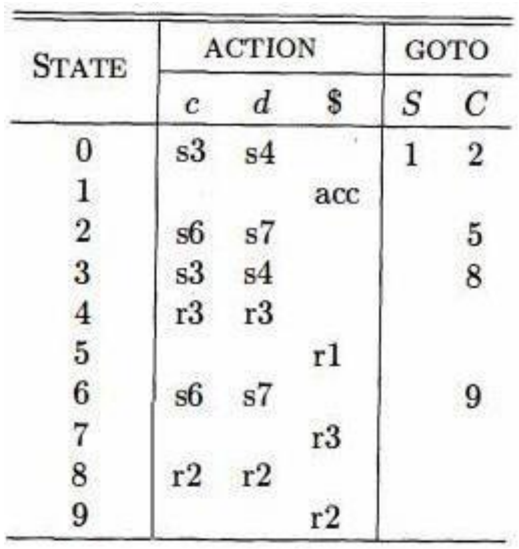
Algorithm 3.2: Construction of canonical-LR parsing tables.
INPUT: An augmented grammar *G'.*
OUTPUT: The canonical-LR parsing table functions ACTION and GOTO for *G'.*
METHOD:

1. Construct *C'* = *{I_0,I_1,..., I_n},* the collection of sets of LR(1) items for *G'*
2. State *i* of the parser is constructed from I_i. The parsing action for state *i* is determined as follows.
    (a) If *[A → α.aβ,b]* is in *I_i* and GOTO(*I_i*,a) = *I_j*, then set ACTION[i, a] to "shift j . " Here a must be a terminal.
    (b) If *[A → α.,a]* is in *I_i*, *A ≠ S',* then set ACTI0N[i, a*]* to "reduce *A* -> a."
    (c) If *[S'→ S.,* $] is in *$I_i$*, then set ACTION[i, a*]* to "accept."

    If any conflicting actions result from the above rules, we say the grammar is not LR(1). The algorithm fails to produce a parser in this case.
3. The goto transitions for state *i* are constructed for all nonterminals *A* using the rule: If GOTO(I_j, *A)* = *I_j*, 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, $].

The table formed from the parsing action and goto functions produced by Algorithm 3.2 is called the ***canonical*** **LR(1) parsing table**. An LR parser using this table is called a **canonical-LR(1) parser**. If the parsing action function has no multiply defined entries, thenthe given grammar is called an ***LR(1) grammar***. As before, we omit the "(1)" if it is understood.

Example 3.3: The canonical parsing table for grammar (3.1) is shown in Fig. 3.3. Productions 1, 2, and 3 are S → *CC, C* → *cC,* and *C* → *d,* respectively.
Every SLR(1) grammar is an LR(1) grammar, but for an SLR(1) grammar the canonical LR parser may have more states than the SLR parser for the same grammar.



STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Figure 3.3: Canonical parsing table for grammar (3.1)

3.2.2 Constructing LALR Parsing Tables

The last parser construction method, called, the **LALR *(lookahead-* LR) technique**. This method is often used in practice, because the tables obtained by it are considerably smaller than the canonical LR tables, yet most common syntactic constructs of programming languages can be expressed conveniently by an LALR grammar.

Algorithm 3.3: Construction of Look Ahead -LR parsing tables.
INPUT: An augmented grammar *G'.*
OUTPUT: The LALR parsing-table functions ACTION and GOTO for *G'.*
METHOD:

1. Construct $C = \{ I_0, I_1, \ldots, I_n\}$, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let $C = \{J_0, J_1, \ldots, J_m\}$ be the resulting sets of LR(1) items. The parsing actions for state *i* are constructed from J_i in the same manner as in Algorithm 3.2. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(l).
4. The GOTO table is constructed as follows. If *J* is the union of one or more sets of LR(1) items, that is, $J = I_1 \cup I_2 \cup \ldots I_k$, then the cores of GOTO(I_1, *X),* GOTO(I_2 , *X),*... , GOTO(I_k , *X)* are the same, since I_1, I_2 ..., I_k all have the same core. Let *K* be the union of all sets of items having the same core as GOTO(I_1,X). Then GOTO(J,X) = *K.*

The table produced by Algorithm 3.3 is called the ***LALR parsing table*** for *G.* If there are no parsing action conflicts, then the given grammar is said to be an ***LALR(1) grammar***. The collection of sets of items constructed in step (3) is called the ***LALR(1) collection***.

Example 3.4: Again consider grammar (3.1) whose **GOTO** graph was shown in Fig. 3.2. As we mentioned, there are three pairs of sets of items that can be merged. I_3 and I_6 are replaced
by their union:

I_{36} : C → c.C, c/d/$
C → .cC, c/d/$
C → .d, c/c/$

I_4 and I_7 are replaced by their union:

I_{47} : *C → d., c/d/$*

and I_8 and I_9 are replaced by their union:

I_{89} : C → cC., c/d/$

The LALR action and goto functions for the condensed sets of items are shown in Fig. 3.4.

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*c*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Figure 3.4: LALR parsing table for the grammar of Example3.1

3.3 Using Ambiguous Grammars

Ambiguous grammars are quite useful in the specification and implementation of languages. For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar. Another use of ambiguous grammars is in isolating commonly occurring syntactic constructs for special-case optimization. With an ambiguous grammar, we can specify the special-case constructs by carefully adding new productions to the grammar.

Precedence and Associativity to Resolve Conflicts

- **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.

3.4 Error Recovery in LR Parsing

An LR parser will detect an error **when it consults the parsing action table and find a blank or error entry**. Errors are never detected by consulting the goto table.
An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far scanned.

- A canonical LR parser will not make even a single reduction before announcing the error.
- SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.
- We can implement two Modes of Error Recovery :
  1. **Panic-mode Error Recovery**
  2. **Phrase-level Recovery**

Panic-mode Error Recovery

We can implement panic-mode error recovery **by scanning down the stack until a state s with a goto on a particular nonterminal A is found**.

- Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A.
- The parser then stacks the state GOTO(s, A) and resumes normal parsing.
- The situation might exist where there is more than one choice for the nonterminal A.
- Normally these would be nonterminals representing major program pieces, e.g. an expression, a statement, or a block.

For example, if A is the nonterminal stmt, a might be semicolon or }, which marks the end of a statement sequence.
This method of error recovery **attempts to eliminate the phrase containing the syntactic error**.

- The parser determines that a string derivable from A contains an error.
- Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack.
- The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow A.
- By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that if has found an instance of A and resumes normal parsing.

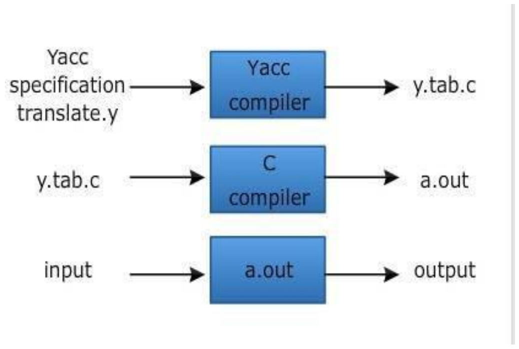
Phrase-level Recovery
Phrase-level recovery is implemented **by examining each error entry in the LR action table and deciding on the basis of language** usage the most likely programmer error that would give rise to that error.
An appropriate recovery procedure can then be constructed;

- presumably the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry.
- In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.
- The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols.
- We must make our choices so that the LR parser will not get into an infinite loop.
- A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached.
- Popping a stack state that covers a nonterminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

3.5 Parser Generators

YACC is an automatic tool that generates the parser program

1. YACC stands for Yet Another Compiler Compiler.
2. YACC provides a tool to produce a parser for a given grammar.
3. YACC is a program designed to compile a LALR (1) grammar.
4. It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
5. The input of YACC is the rule or grammar and the output is a C program.



A translator can be constructed using **YACC**.

- First, a file, say **translate.y,** containing a **YACC** specification of the translator is prepared. The UNIX system command **yacc translate.y**
- transforms the file **translate.y** into a C program called **y.tab.c** using the LALR method.
- The program **y.tab.c** is a representation of an LALR parser written in C, along with other C routines that the user may have prepared.

- By compiling **y.tab.c** along with the **ly** library that contains the LR parsing program using the command.
    - **cc y.tab.c -ly**
- We obtain the desired object program a.out that performs the translation specified by the original **Yacc** program.
- If other procedures are needed, they can be compiled or loaded with **y.tab.c,** just as with any C program.

A **YACC** source program has three parts:

Declarations

%%

translation rules

%%

auxiliary functions

```
%token DIGIT

 %{
 #include <ctype.h>
 %}
 %token DIGIT
 %%
 line      : expr '\n'             { printf("%d\n", $1); }
           ;
 expr      : expr '+' term         { $$ = $1 + $3; }
           | term
           ;
 term      : term '*' factor       { $$ = $1 * $3; }
           | factor
           ;
 factor    : '(' expr ')'          { $$ = $2; }
           | DIGIT
           ;
 %%
 yylex() {
       int c;
       c = getchar();
       if (isdigit(c)) {
           yylval = c-'0';
           return DIGIT;
       }
       return c;
 }
```

Syntax Directed Definition

A ***syntax-directed definition*** (SDD) is a context-free grammar **together with attributes and rules**. **Attributes are associated with grammar symbols and rules are associatedwith productions**. If *X* is a symbol and *a* is one of its attributes, then we write *X.a* to denote the value of a at a particular parse-tree node labeled *X.* For example, an infix-to- postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE	
E → E_1 + T	**E.code = E_1.code || T.code || '+'**	**...(3.3.1)**

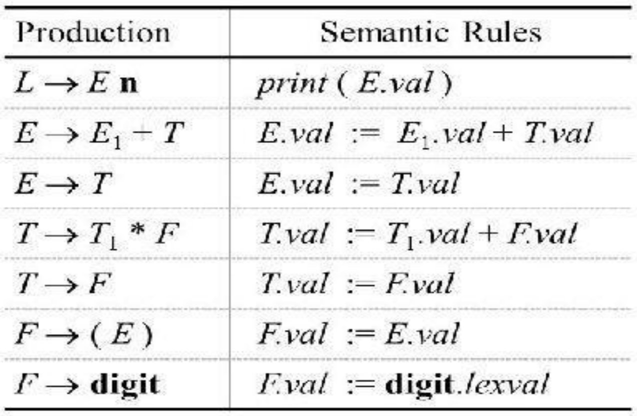
This production has two nonterminals, *E* and T; the subscript in E_1 distinguishes the occurrence of *E* in the production body from the occurrence of *E* as the head. Both *E* and *T* have a string-valued attribute *code.* The semantic rule specifies that the string *E.code* is formed by concatenating *E_1.code, T.code,* and the character '+'.

1.1 Inherited and Synthesized Attributes

Two kinds of **attributes for nonterminals**:

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

Example 3.3.1: The SDD in Fig. 3.3.1 is based on our familiar grammar for arithmetic expressions with operators + and *. It evaluates expressions terminated by an endmarker **n.** In the SDD, each of the nonterminals has a single synthesized attribute, called *val.* We also suppose that the terminal **digit** has a synthesized attribute *lexval,* which is an integer value returned by the lexical analyzer.



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

Figure 3.3.1 : Syntax-directed definition of a simple desk calculator

- The rule for **production 1**, *L* → *E* **n,** sets *L.val* to *E.val,* which we shall see is the numerical value of the entire expression.
- **Production 2**, *E* → *E_1* + *T,* also has one rule, which computes the *val* attribute for the head *E* as the sum of the values at *E_1* and *T.* At any parse tree node *N* labeled

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

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

An SDD that involves only synthesized attributes is called ***S-attributed;*** the SDD in Fig. 3.3.1 has this property. In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

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

1.2 Evaluating an SDD at the Nodes of a Parse Tree

A parse tree, showing the value(s) of its attribute(s) is called an ***annotated parse tree***. For SDD's with both inherited and synthesized attributes, there is no guarantee that there is even one order in which to evaluate attributes at nodes.

Example 3.3.2: Figure 3.3.2 shows an annotated parse tree for the input string 3 * 5 + 4 **n,** constructed using the grammar and rules of Fig. 3.3.1. The values of *lexval* are presumed supplied by the lexical analyzer. Each of the nodes for the nonterminals has attribute *val* computed in a bottom-up order, and we see the resulting values associated with each node. For instance, at the node with a child labeled *, after computing *T.val* = 3 and *F.val* = 5 at its first and third children, we apply the rule that says *T.val* is the product of these two values, or 15.

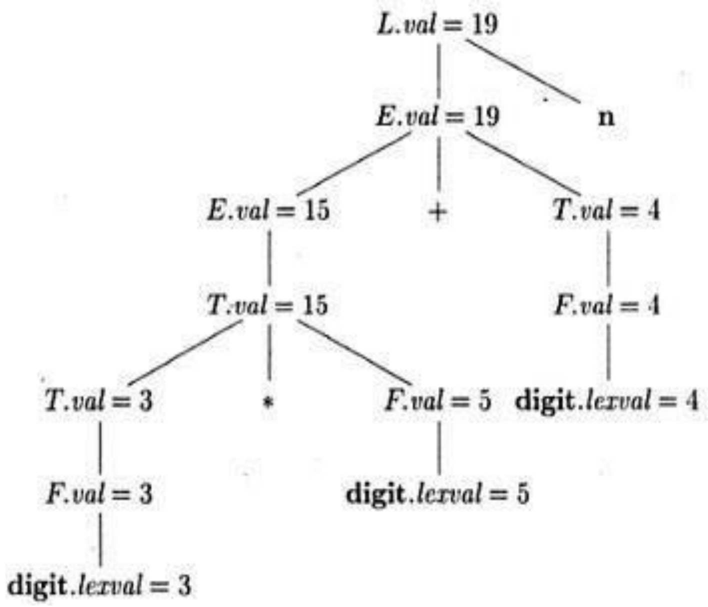


Figure 3.3.2: An SDD based on a grammar suitable for top-down parsing

Example 3.3.3: The SDD in Fig. 3.3.3 computes terms like 3 * 5 and 3 * 5 * 7. The top- down parse of input 3*5 begins with the production *T* → *FT'.* Here, *F* generates the digit 3, but the operator * is generated by T'. Thus, the left operand 3 appears in a different subtree of the parse tree from *. An inherited attribute will therefore be used to pass the operand to the operator.

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

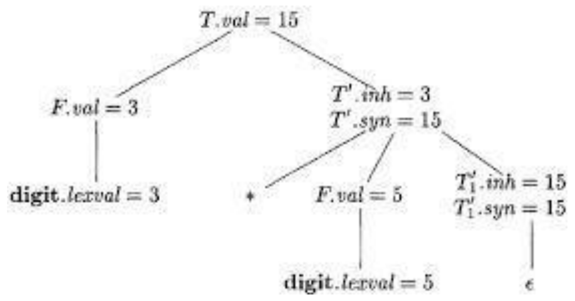
Figure 3.3.3: An SDD based on a grammar suitable for top-down parsing

Each of the nonterminals *T* and *F* has a synthesized attribute *val;* the terminal **digit** has a synthesized attribute *lexval.* The nonterminal *T* has two attributes: an inherited attribute *inh* and a synthesized attribute *syn.*

The semantic rules are based on the idea that the left operand of the operator * is inherited. More precisely, the head T' of the production *T'* → * *F* T_1' inherits the left operand of * in the production body. Given a term *x * y * z,* the root of the subtree for * *y* * *z* inherits *x.* Then, the root of the subtree for * *z* inherits the value of *x * y,* and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for 3 * 5 in Fig. 3.3.5. The leftmost leaf in the parse tree, labeled **digit,** has attribute value *lexval* = 3, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, *F* **-> digit.** The only semantic rule associated with this production defines *F.val* **= digit.***lexval,* which equals 3.

Figure 3.3.5: Annotated parse tree for 3 * 5



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

The production at the node for *T* is T' → * *FT₁'.* (We retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for *T'.)* The inherited attributeT_1'*.inh* is defined by the semantic rule $T_1'.inh = T'.inh \times F.val$ associated with production 2.

With *T'.inh* = 3 and *F.val* = 5, we get $T_1'.inh = 15$. At the lower node for T_1', the production is T'-> ε. The semantic rule *T'.syn = T'.inh* defines $T_1'.syn = 15$. The *syn* attributes at the nodes for T' pass the value 15 up the tree to the node for T, where *T.val* = 15.

Evaluation Orders for SDD's

1. Dependency Graphs

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

- For each parse-tree node, say a node labeled by grammar symbol *X,* the dependency graph has a node for each attribute associated with *X.*
- Suppose that a semantic rule associated with a production *p* defines the value of synthesized attribute *A.b* in terms of the value of *X.c.* Then, the dependency graph has an edge from *X.c* to *A.b.* More precisely, at every node *N* labeled *A* where production *p* is applied; create an edge to attribute *b* at *N,* from the attribute c at the child of *N* corresponding to this instance of the symbol *X* in the body of the production.
- Suppose that a semantic rule associated with a production *p* defines the value of inherited attribute *B.c* in terms of the value of *X.a.* Then, the dependency graph has an edge from *X.a* to *B.c.* For each node *N* labeled *B* that corresponds to an occurrence of this *B* in the body of production *p,* create an edge to attribute c at *N* from the attribute *a* at the node *M* that corresponds to this occurrence of *X.* Note that *M* could be either the parent or a sibling of *N.*

Example 3.4.1: Consider the following production and rule:

PRODUCTION	SEMANTIC RU LE
$E \to E_1 + T$	$E.val = E_1.val + T.val$

At every node *N* labeled *E,* with children corresponding to the body of this production, the synthesized attribute *val* at *N* is computed using the values of *val* at the two children, labeled *E* and *T.* Thus, a portion of the dependency graph for every parse tree in which this production is used looks like Fig. 3.4.1. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

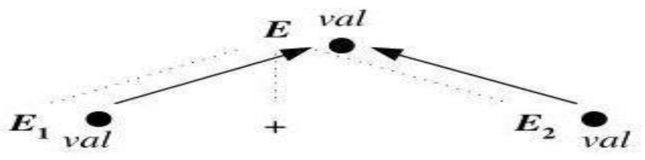


Figure 3.4.1: E.val is synthesized from Ei.val and T.val

Example 3.4.2: An example of a complete dependency graph appears in Fig. 3.4.2. The nodes of the dependency graph, represented by the numbers 1 through 9, correspond to the attributes in the annotated parse tree in Fig. 3.3.5

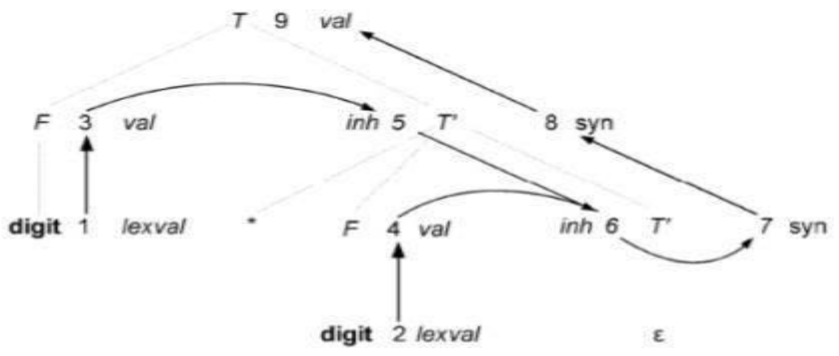


Figure 3.4.2: Dependency graph for the annotated parse tree of Fig. 3.3.5

- Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled digit.
- Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled *F.* The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines *F.val* in terms of *digit.lexval* In fact, *F.val* equals digit *.lexval,* but the edge represents dependence, not equality.
- Nodes 5 and 6 represent the inherited attribute *T'.inh* associated with each of the occurrences of nonterminal *T'.* The edge to 5 from 3 is due to the rule *T'.inh = F.val,* which defines *T'.inh* at the right child of the root from *F.val* at the left child. We see edges to 6 from node 5 for *T'.inh* and from node 4 for *F.val,* because these values are multiplied to evaluate the attribute *inh* at node 6.
- Nodes 7 and 8 represent the synthesized attribute *syn* associated with the occurrences of X". The edge to node 7 from 6 is due to the semantic rule *T'.syn = T'.inh* associated with production 3 in Fig. 3.3.4. The edge to node 8 from 7 is due to a semantic rule associated with production 2.
- Finally, node 9 represents the attribute *T.val.* The edge to 9 from 8 is due to the semantic rule, *T.val = T'.syn,* associated with production 1.

The dependency graph of Fig. 3.4.2 has no cycles. One topological sort is the order in which the nodes have already been numbered: 1,2,... ,9. Notice that every edge of the graph goes from a node to a higher-numbered node, so this order is surely a topological sort. There are other topological sorts as well, such as 1,3,5,2,4,6,7,8,9.

2. S-Attributed Definitions
Given an SDD, it is very hard to tell whether there exist any parse trees whose dependency graphs have cycles. In practice, translations can be implemented using classes of SDD's that guarantee an evaluation order, since they do not permit dependencygraphs with cycles. Moreover, the two classes can be implemented efficiently in connection with top-down or bottom-up parsing.
The first class is defined as follows:

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

Example 3.4.2: The SDD of Fig 3.3.1 is an example of an S-attributed definition. Each attribute, *L.val, E.val, T.val,* and *F.val* is synthesized.

When an SDD is S-attributed, we can evaluate its attributes in any bottom up order of the nodes of the parse tree. It is often especially simple to evaluate the attributes by performing a postorder traversal of the parse tree and evaluating the attributes at a node *N* when the traversal leaves *N* for the last time. That is, we apply the function *postorder,* defined below, to the root of the parse tree

```
postorder(N) {
        for ( each child C of N, from the left ) postorder(C);
        evaluate the attributes associated with node N;
}
```

S-attributed definitions can be implemented during bottom-up parsing, since a bottom-up parse corresponds to a postorder traversal. Specifically, postorder corresponds exactly to the order in which an LR parser reduces a production body to its head.

3. L-Attributed Definitions

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

1. Synthesized, or
2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1X_2 \ldots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only:

    (a) Inherited attributes associated with the head *A.*

    (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1X_2 \ldots X_n$ located to the left of X_i.

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

Example 3.4.2: The SDD in Fig. 3.3.4 is L-attributed. To see why, consider the semantic rules for inherited attributes, which are repeated here for convenience:

PRODUCTION	SEMANTIC RULE
$T \rightarrow FT'$	***T'.inh = F.val***
$T' \rightarrow *F T_1'$	$T_1'.Inh = T'.inh \times F.val$

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

In each of these cases, the rules use information "from above or from the left," asrequired by the class. The remaining attributes are synthesized. Hence, the SDD is L- attributed.

Applications of Syntax-Directed Translation

The main application in the construction of syntax trees is some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string intoa tree. We consider two SDD's for constructing syntax trees for expressions. The first, an

S-attributed definition, is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing.

3.1 Construction of Syntax Trees

Each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression *E1 + E2* has label + and two children representing the subexpressions *E1* and *E2.*

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

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

Example 3.5.1: The S-attributed definition in Fig. 3.5.1 constructs syntax trees for a simple expression grammar involving only the binary operators + and -. As usual, these operators are at the same precedence level and are jointly left associative. All nonterminals have one synthesized attribute *node,* which represents a node of the syntax tree.

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

Figure 3.5.1 : Constructing syntax trees for simple expressions

Every time the first production $E \rightarrow E_1 + T$ is used, its rule creates a node with ' + ' for *op* and two children, $E_1.node$ and *T.node,* for the subexpressions. The second production has a similar rule.

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

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

Figure 3.5.2 shows the construction of a syntax tree for the input *a* — 4 + *c.*

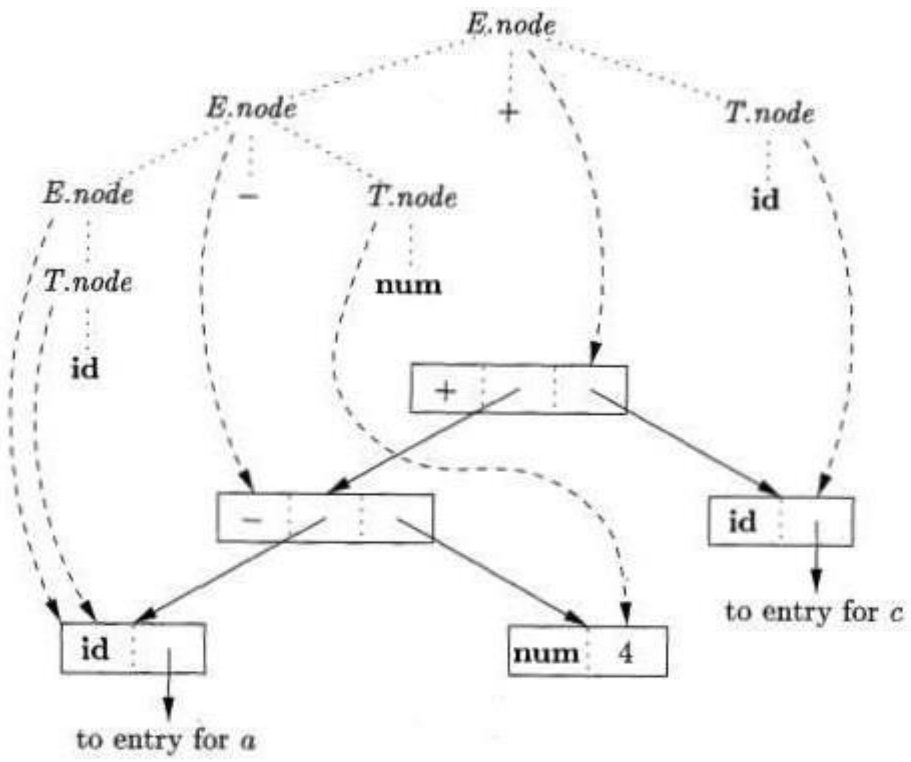


Figure 3.5.2: Syntax tree for a — 4 + c

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

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

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

If the rules are evaluated during a postorder traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown in Fig. 3.5.3 ends with p5 pointing to the root of the constructed syntax tree.

1) p_1 = **new** *Leaf*(**id**, *entry-a*);
2) p_2 = **new** *Leaf*(**num**, 4);
3) p_3 = **new** *Node*('−', p_1, p_2);
4) p_4 = **new** *Leaf*(**id**, *entry-c*);
5) p_5 = **new** *Node*('+', p_3, p_4);

Figure 3.5.3: Steps in the construction of the syntax tree for a — 4 + c

Example 3.5.2: The L-attributed definition in Fig. 3.5.4 performs the same translation as the S-attributed definition in Fig. 3.5.1. The attributes for the grammar symbols *E, T,* **id,** and **num** are as discussed in Example 3.5.2.

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

Figure 3.5.4 : Constructing syntax trees during top-down parsing

The rules for building syntax trees in this example are similar to the rules for the desk calculator. In the desk-calculator example, a term *x* * *y* was evaluated by passing *x* as an inherited attribute, since *x* and * *y* appeared in different portions of the parse tree. Here, *the* idea is to build a syntax tree for *x* + *y* by passing *x* as an inherited attribute, since *x* and + *y* appear in different subtrees. Nonterminal *E'* is the counterpart of nonterminal T'.

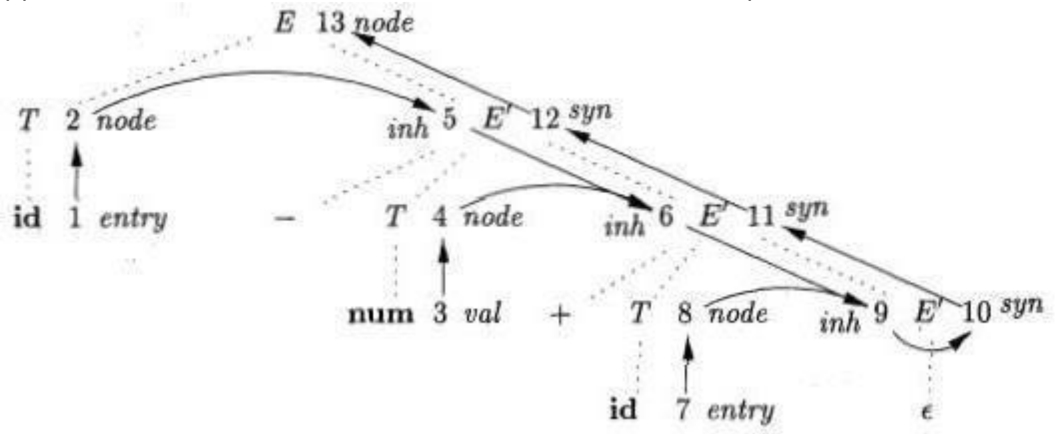


Figure 3.5.5 : Dependency graph for a - 4 + c, with the SDD of Fig. 3.5.2

Nonterminal *E'* has an inherited attribute *inh* and a synthesized attribute *syn.* Attribute *E'.inh* represents the partial syntax tree constructed so far. Specifically, it represents the root of the tree for the prefix of the input string that is to the left of the subtree for *E'.* At node 5 in the dependency graph in Fig. 3.5.5, *E'.inh* denotes the root of the partial syntax tree for the identifier a; that is, the leaf for *a.* At node 6, *E'.inh* denotes the root for the partial syntax tree for the input *a* — 4. At node 9, *E'.inh* denotes the syntax tree for *a* — 4 + c.

Since there is no more input, at node 9, *E'.inh* points to the root of the entire syntax tree. The *syn* attributes pass this value back up the parse tree until it becomes the value of *E.node.* Specifically, the attribute value at node 10 is defined by the rule *E'.syn = E'.inh* associated with the production *E' —> e.* The attribute value at node 11 is defined by the rule *E'.syn* = E_1*' .syn* associated with production 2 in Fig.3.5.4. Similar rules define the attribute values at nodes 12 and 13.

Syntax-Directed Translation Schemes

Syntax-directed translation schemes are a complementary notation to syntax directed definitions. A ***syntax-directed translation scheme*** (SDT) is a context free grammar with program fragments embedded within production bodies. The program fragments are called ***semantic actions*** and can appear at any position within a production body. By convention, we place curly braces around actions; if braces are needed as grammarsymbols, then we quote them.

Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth-first order; that is, during a preorder traversal.

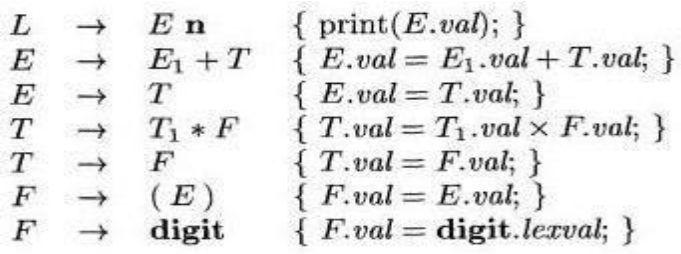
Typically, SDT's are implemented during parsing, without building a parse tree. To implement SDT's, two important classes of SDD's:

1. The underlying grammar is LR-parsable, and the SDD is S-attributed.
2. The underlying grammar is LL-parsable, and the SDD is L-attributed.

1. Postfix Translation Schemes

By far the simplest SDD implementation occurs when we can parse the grammar bottom-up and the SDD is S-attributed. In that case, we can construct an SDT in which each action is placed at the end of the production and is executed along with the reduction of the body to the head of that production. SDT's with all actions at the right ends of the production bodies are called ***postfix*** *SDT's.*

Example 3.6.1: The postfix SDT in Fig. 3.6.1 implements the desk calculator SDD of Fig. 3.3.1, with one change: the action for the first production prints a value. The remaining actions are exact counterparts of the semantic rules. Since the underlying grammar is LR, and the SDD is S-attributed, these actions can be correctly performed along with the reduction steps of the parser.



L	$\rightarrow$	E **n**	$\{ \text{print}(E.val); \}$
E	$\rightarrow$	$E_1 + T$	$\{ E.val = E_1.val + T.val; \}$
E	$\rightarrow$	T	$\{ E.val = T.val; \}$
T	$\rightarrow$	$T_1 * F$	$\{ T.val = T_1.val \times F.val; \}$
T	$\rightarrow$	F	$\{ T.val = F.val; \}$
F	$\rightarrow$	(E)	$\{ F.val = E.val; \}$
F	$\rightarrow$	**digit**	$\{ F.val = \textbf{digit}.lexval; \}$

Figure 3.6.1: Postfix SDT implementing the desk calculator

2. Parser-Stack Implementation of Postfix SDT's

Postfix SDT's can be implemented during LR parsing by executing the actions when reductions occur. The attribute(s) of each grammar symbol can be put on the stack in a place where they can be found during the reduction. The best plan is to place the attributes along with the grammar symbols (or the LR states that represent these symbols) in records on the stack itself.

In Fig. 3.6.2, the parser stack contains records with a field for a grammar symbol (or parser state) and, below it, a field for an attribute. The three grammar symbols *X Y Z* are on top of the stack; perhaps they are about to be reduced according to a production like*A —> X Y Z.* Here, we show *X.x* as the one attribute of *X,* and so on. In general, we can allow for more attributes, either by making the records large enough or by putting pointers to records on the stack. With small attributes, it may be simpler to make the records large enough, even if some fields go unused some of the time. However, if one or more attributes are of unbounded size — say, they are character strings — then it would be better to put a pointer to the attribute's value in the stack record and store the actual value in some larger, shared storage area that is not part of the stack.

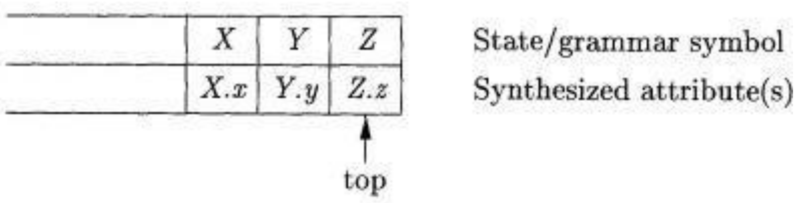


Figure 3.6.2: Parser stack with a field for synthesized attributes

If the attributes are all synthesized, and the actions occur at the ends of the productions, then we can compute the attributes for the head when we reduce the body to the head. If we reduce by a production such *as A → X Y Z ,* then we have all the attributes of *X, Y,* and *Z* available, at known positions on the stack, as in Fig.3.6.2. After the action, *A* and its attributes are at the top of the stack, in the position of the record for *X.*

Example 3.6.2: Let us rewrite the actions of the desk-calculator SDT of Example 3.6.1 so that they manipulate the parser stack explicitly. Such stack manipulation is usually done automatically by the parser.

PRODUCTION	ACTIONS
$L \rightarrow E\ \mathbf{n}$	$\{\ \text{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$E \rightarrow E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \rightarrow F$	
$F \rightarrow (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \rightarrow \mathbf{digit}$	

Figure 3.6.3: Implementing the desk calculator on a bottom-up parsing stack

Suppose that the stack is kept in an array of records called *stack,* with *top* a cursor to the top of the stack. Thus, *stack [top]* refers to the top record on the stack, *stack [top* - 1] to the record below that, and so on. Also, we assume that each record has a field called *val,* which holds the attribute of whatever grammar symbol is represented in that record. Thus, we may refer to the attribute *E.val* that appears at the third position on the stack as *stack[top - 2].val.* The entire SDT is shown in Fig. 3.6.3.

For instance, in the second production, *E* $\rightarrow$ E_1 *+ T,* we go two positions below the top to get the value of E_1, and we find the value of *T* at the top. The resulting sum is placed where the head *E* will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one. After computing *E.val,* we pop two symbols off the top of the stack, so the record where we placed *E.val* will now be at the top of the stack.

In the third production, *E* $\rightarrow$ *T,* no action is necessary, because the length of the stack does not change, and the value of *T.val* at the stack top will simply become the value of *E.val.* The same observation applies to the productions T $\rightarrow$ *F* and *F* $\rightarrow$ digit. Production *F* $\rightarrow$ (*E*) is slightly different. Although the value does not change, two positions are removed from the stack during the reduction, so the value has to move to the position after the reduction.

3. SDT's With Actions inside Productions

An action may be placed at any position within the body of a production. It is performed immediately after all symbols to its left are processed. Thus, if we have a production *B* $\rightarrow$ *X {a} Y,* the action *a* is done after we have recognized *X* (if *X* is a terminal) or all the terminals derived from *X* (if *X* is a nonterminal). More precisely,

- If the parse is bottom-up, then we perform action *a* as soon as this occurrence of *X* appears on the top of the parsing stack.
- If the parse is top-down, we perform *a* just before we attempt to expand this occurrence of *Y* (if *Y* a nonterminal) or check for *Y* on the input (if *Y* is a terminal).

SDT's that can be implemented during parsing include postfix SDT's and a class of SDT's that implements L-attributed definitions. Not all SDT's can be implemented during parsing, as we shall see in the next example.

Example 3.6.3: As an extreme example of a problematic SDT, suppose that we turn our desk-calculator running example into an SDT that prints the prefix form of an expression, rather than evaluating the expression. The productions and actions are shown in Fig. 3.6.4.

$$\begin{array}{llll}
1) & L & \rightarrow & E\ \mathbf{n} \\
2) & E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T \\
3) & E & \rightarrow & T \\
4) & T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F \\
5) & T & \rightarrow & F \\
6) & F & \rightarrow & (\ E\ ) \\
7) & F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.lexval);\ \}
\end{array}$$

Figure 3.6.4: Problematic SDT for infix-to-prefix translation during parsing

Unfortunately, it is impossible to implement this SDT during either top down or bottom- up parsing, because the parser would have to perform critical actions, like printing instances of * or +, long before it knows whether these symbols will appear in its input.

Using marker nonterminals M2 and M4 for the actions in productions 2 and 4, respectively, on input 3, a shift-reduce parser has conflicts between reducing by M2 → ε, reducing by M4 → ε, and shifting the digit.

Any SDT can be implemented as follows:

1. Ignoring the actions, parse the input and produce a parse tree as a result.
2. Then, examine each interior node *N,* say one for production *A* → *α.* Add additional children to *N* for the actions in *α,* so the children of *N* from left to right have exactly the symbols and actions of *α.*
3. Perform a preorder traversal of the tree, and as soon as a node labeled by an action is visited, perform that action.

For instance, Fig. 3.6.5 shows the parse tree for expression 3 * 5 + 4 with actions inserted. If we visit the nodes in preorder, we get the prefix form of the expression: + * 3 5 4.

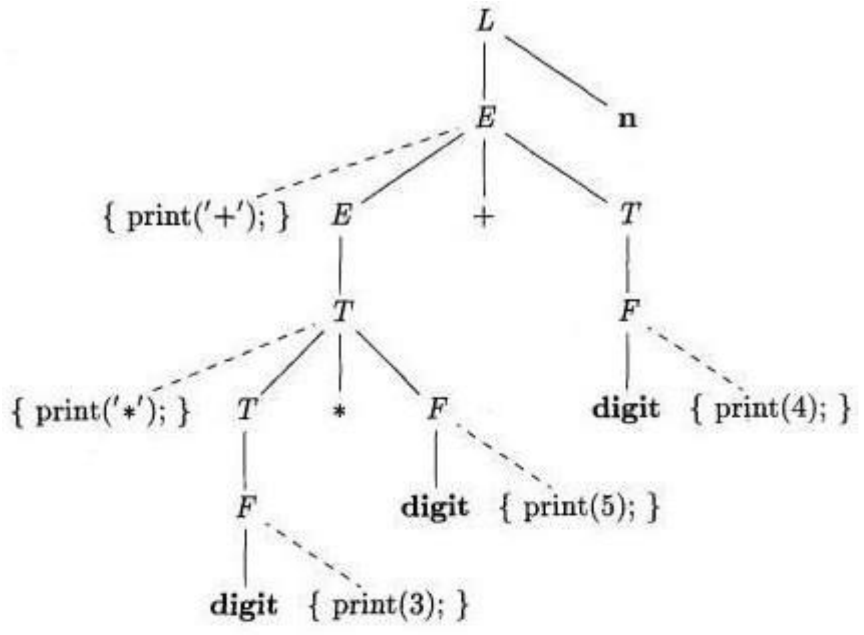


Figure 3.6.5: Parse tree with actions embedded

4. Eliminating Left Recursion from SDT's

Since no grammar with left recursion can be parsed deterministically top-down, we examined left-recursion elimination. When the grammar is part of an SDT, we also need to worry about how the actions are handled.

First, consider the simple case, in which the only thing we care about is the order in which the actions in an SDT are performed. For example, if each action simply prints a string, we care only about the order in which the strings are printed. In this case, the following principle can guide us:

• When transforming the grammar, treat the actions as if they were terminal symbols.

This principle is based on the idea that the grammar transformation preserves the order of the terminals in the generated string. The actions are therefore executed in the same order in any left-to-right parse, top-down or bottom-up. The "trick" for eliminating left recursion is to take two productions

$$A \rightarrow A\alpha \mid \beta$$

that generate strings consisting of a β and any number of α's, and replace them by productions that generate the same strings using a new nonterminal *R* (for "remainder") of the first production:

$$A \rightarrow \beta R$$
$$R \rightarrow \alpha R \mid \varepsilon$$

If β does not begin with *A,* then *A* no longer has a left-recursive production. In regular-definition terms, with both sets of productions, *A* is defined by **β***(α)*.*

Example 3.6.4: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

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

If we apply the standard transformation to *E,* the remainder of the left-recursive production is $\alpha = + T \text{ \{ print('+'); \}}$

and the body of the other production is *T.* If we introduce *R* for the remainder of *E,* we get the set of productions:

$$E \rightarrow T\ R$$
$$R \rightarrow +T \text{ \{ print('+'); \} } R$$
$$R \rightarrow \varepsilon$$

When the actions of an SDD compute attributes rather than merely printing output, we must be more careful about how we eliminate left recursion from a grammar. However, if the SDD is S-attributed, then we can always construct an SDT by placing attribute-computing actions at appropriate positions in the new productions.

5. SDT's for L-Attributed Definitions

We converted S-attributed SDD's into postfix SDT's, with actions at the right ends of productions.

The rules for turning an L-attributed SDD into an SDT are as follows:

1. Embed the action that computes the inherited attributes for a nonterminal *A* immediately before that occurrence of *A* in the body of the production. If several inherited attributes for *A* depend on one another in an acyclic fashion, order the evaluation of attributes so that those needed first are computed first.
2. Place the actions that compute a synthesized attribute for the head of a production at the end of the body of that production.

UNIT IV

Intermediated Code: Generation Variants of Syntax trees 3 Address code, Types and Deceleration, Translation of Expressions, Type Checking. Canted Flow Back patching?

4.1 INTRODUCTION

What is Intermediate Code?

Intermediate Code is a modified input source program which is stored in some data structure. The front end translates a source program into an intermediate representation from which the back end generates target code.

Why Intermediate Code Generation is required?

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimization can be applied to the intermediate representation.

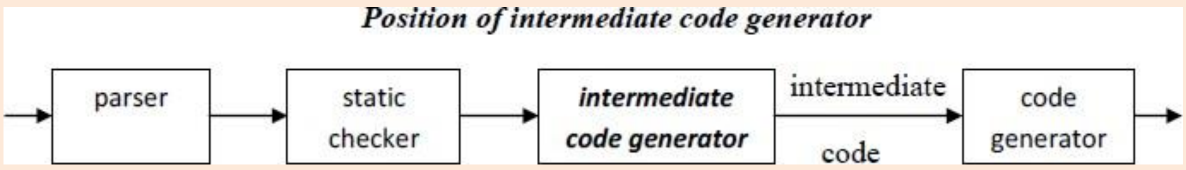

Position of intermediate code generator
parser
static checker
intermediate code generator
intermediate code
code generator

Fig. 4.1: Position of Intermediate Code Generator

Intermediate code can be represented by using the following:

i) Three Address Code
ii) Syntax Tree

4.1.1 Three-Address Code:

Three-address code is a sequence of statements of the general form

x = y *op* z

Where x, y and z are names, constants, or compiler-generated temporaries; *op* stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean valued data. Thus a source language expression like x+ y*z might be translated into a sequence

t1= y * z

t2= x + t1

Where t1 and t2 are compiler-generated temporary names.

The reason for the term "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result.

An address can be one of the following:

• ***A name.*** For convenience, we allow source-program names to appear as addresses in three-address code. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.

• ***A constant.*** In practice, a compiler must deal with many different types of constants and variables.

• ***A compiler-generated temporary.*** It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form **x = y *op* z**, where ***op*** is a binary arithmetic or logical operation.
2. Assignment instructions of the form **x = *op* y**, where ***op*** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.
3. *Copy statements* of the form **x = y** where the value of *y* is assigned to *x*.
4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.
5. Conditional jumps such as **if *x relop y* goto L**. This instruction applies a relational operator (<, =, >=, etc.) to *x* and *y*, and executes the statement with label L next if *x* stands in relation *relop to y*. If not, the three-address statement following if *x relop y* goto L is executed next, as in the usual sequence.
6. *param x* and *call p, n* for procedure calls and *return y*, where y representing a returned value is optional. For example,
   param x1
   param x2
   . . .
   param xn
   call p,n
   generated as part of a call of the procedure p(x1, x2, …. ,xn).
7. Indexed assignments of the form x:= y[i] and x[i] = y.
8. Address and pointer assignments of the form x = &y, x= *y, and *x= y.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands.
Three such representations are:

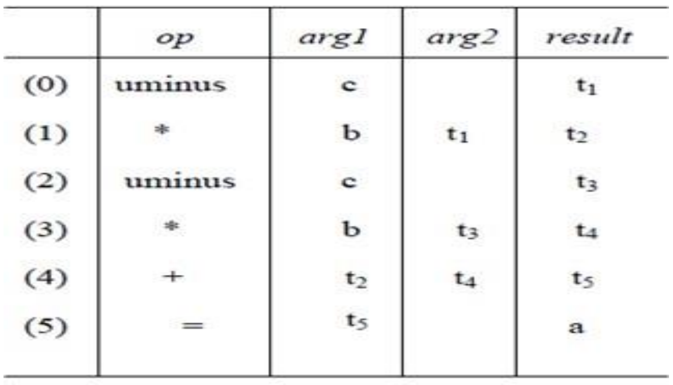
- Quadruples
- Triples
- Indirect triples

Quadruples:

Three-Address Code does not specify the internal representation of 3-Address instructions. This limitation is overcome by Quadruple.

- A quadruple is a record structure with four fields, which are, ***op, arg1, arg2*** and ***result.***
- The *op* field contains an internal code for the operator. The three-address statement ***x = y op z*** is represented by placing *y* in *arg1*, *z* in *arg2* and *x* in *result.*
- The contents of field's arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Example*: a: =b*-c+b*-c* represent the expression using Quadruple, Triples and Indirect Triples.

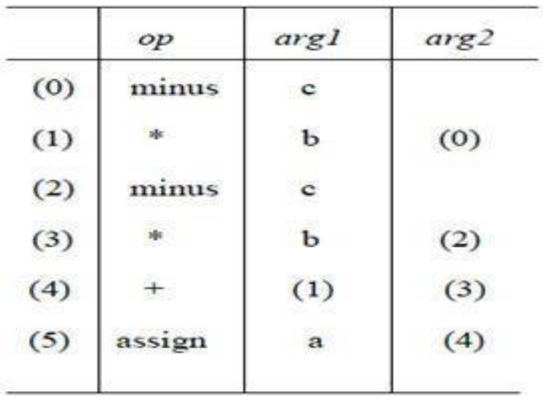


	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		a

(a) Quadruples

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: ***op, arg1* and *arg2.***
- The fields *arg1* and *arg2*, for the arguments of *op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as *triples*.



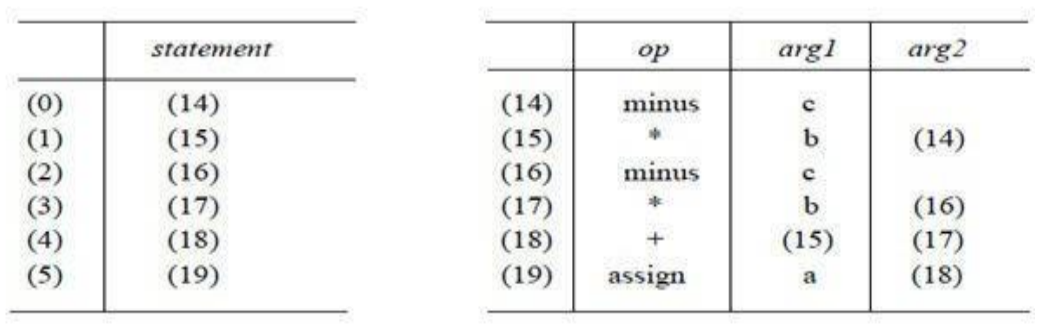
	op	*arg1*	*arg2*
(0)	minus	c	
(1)	*	b	(0)
(2)	minus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

****Note:** The benefit of quadruples over triples can be seen in an optimizing compiler, where instructions are often moved around. With Quadruples if we move an instruction that computes a temporary t, then the instruction that uses t require no change. With Triples the result of an operation is referred to by its position, so moving an instruction may require us to change all references to that result. This problem does not occur in Indirect Triples.

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order.



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

	op	*arg1*	*arg2*
(14)	minus	c	
(15)	*	b	(14)
(16)	minus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

4.1.2 ABSTRACT SYNTAX TREES:

Abstract Syntax tree is a condensed version of a Syntax Tree eliminating all syntactic elements of the language. Abstract Syntax Tree is also used to represent the Intermediate Code. The procedure for constructing abstract syntax tree is same as the procedure that we used to convert an expression into a postfix notation. The operators act as the parent nodes and variables, constants, identifiers as leaf nodes. Abstract Syntax tree is constructed form bottom to top.

- Every node in a syntax tree is a record with many fields. For example an operator will have two operands.
- The three functions **makeleaf(identifier,entry)**, **makeleaf(number,value)** and **makenode(operator,operand1,operand2)** are used while constructing the abstract syntax trees.

1. **Makeleaf(identifier,entry)**
   This function creates an identifier node with the name or label "identifier" and a pointer to symbol table entry given by "entry".
2. **Makeleaf(number,value)**
   This function creates a leaf node "number" and "value".
3. **Makenode(Operator,Operand1,Operand2)**
   This function creates an operator node with a name "operator" and a pointer to the left child (Operand1) and a pointer to the right child (Operand 2).The left and right child can be again an operator node.

Example: Construct Abstract Syntax Tree for the Expression *a*b-(c+d).*

1. Pointer1=Makeleaf(identifier, entry a);
2. Pointer2= Makeleaf(identifier, entry b);
3. Pointer3= Makenode('*', Pointer1,Pointer2);
4. Pointer4= Makeleaf(identifier, entry c);
5. Pointer5= Makeleaf(identifier, entry d);
6. Pointer6= Makenode('+', Pointer4,Pointer5);
7. Pointer7= Makenode('-', Pointer3,Pointer6);

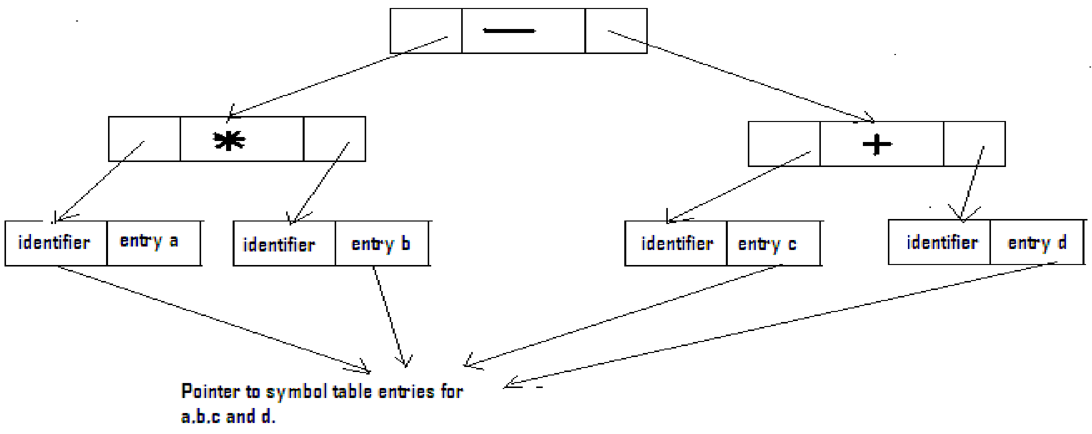
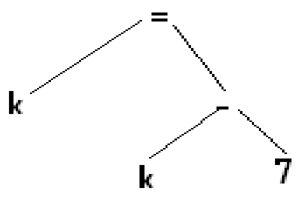


Fig.4.2: Abstract Syntax Tree

4.1.3 Directed Acyclic Graph:

An important derivative of abstract syntax tree is known as Directed Acyclic Graph. It is used to reduce the amount of memory used for storing the Abstract Syntax Tree data structure.
Consider an expression:
k=k-7;
The AST and DAG is shown in the fig below



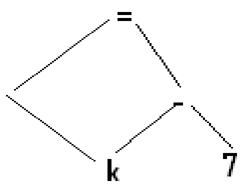


Fig. 4.3: AST **Fig.4.4: DAG**

Note: There are 2 nodes for the identifier **'k'** in fig 1, one representing **k** on the LHS of the expression and the other representing the k on the RHS. The DAG identifies such common nodes and eliminates their duplication in the AST. The DAG for the above expression is shown in fig2.
In DAG a node may have multiple parents. In fig2 node 'k' has two parents (- node and = node).
The creation of DAG is identical to the AST except for the extra check to determine whether a node with identical properties already exists. In the event of the node already created before, it ischained to the existing node avoiding a duplicate node.

Example 2: a + a * (b - c) + (b - c) * d
The leaf for **a** has two parents, because **a** appears twice in the expression. More interestingly, the two occurrences of the common subexpression **b - c** are represented by one node, the node labeled **—**. That node has two parents, representing its two uses in the subexpressions **a* (b - c)** and **(b- c)*d**. Even though **b** and **c** appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression **b - c**.

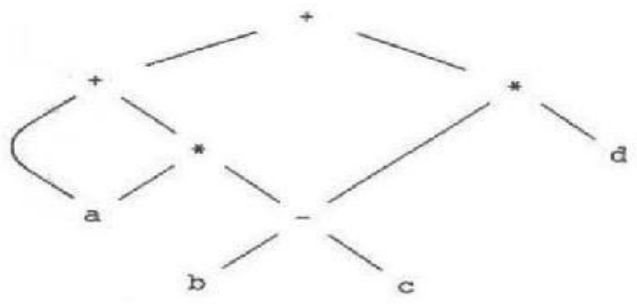


Fig 4.5: DAG for a + a * (b - c) + (b - c) * d

The Value-Number Method for Constructing DAG's
Often, the nodes of a syntax tree or DAG are stored in an array of records, as suggested by Fig. 4.6. Each row of the array represents one record, and therefore one node. In each record, the first field is an operation code, indicating the label of the node. In Fig. 4.6(b), leaves have one additional field, which holds the lexical value (either a symbol-table pointer or a constant, in this case), and interior nodes have two additional fields indicating the left and right children.

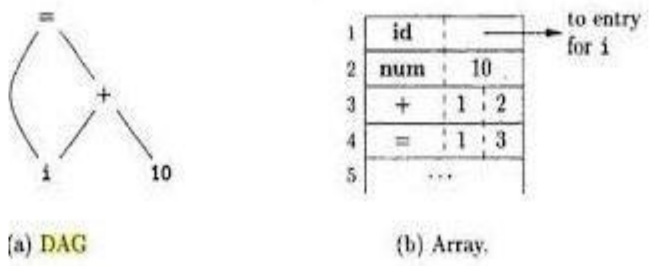


Fig. 4.6: Steps for constructing the DAG of Fig. 4.3

In this array, we refer to nodes by giving the integer index of the record for that node within the array. This integer historically has been called the ***value number*** for the node or for the expression represented by the node. For instance, in Fig. 4.6, the node labeled **+** has value number 3, and it's left and right children have value numbers 1 and 2, respectively. In practice, we could use pointers to records or references to objects instead of integer indexes, but we shall still refer to the reference to a node as its "value number." If stored in an appropriate data structure, value numbers help us construct expression DAG's efficiently.

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

- ***Type checking*** uses logical rules to reason about the behavior of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.
- ***Translation Applications***. From the type of a name, a compiler can determine the storage that will be needed for that name at run time.

4.2.1 Type Expressions
Types have structure, which we shall represent using *type expressions:* a type expression is either a basic type or is formed by applying an operator called a *type constructor* to a type expression. The sets of basic types and constructors depend on the language to be checked.
Example: The array type int [2] [3] can be read as "array of 2 arrays of 3 integers each" and written as a type expression *array (2, array (3, integer)).* This type is represented by the tree in Fig. 4.12. The operator *array* takes two parameters, a number and a type.

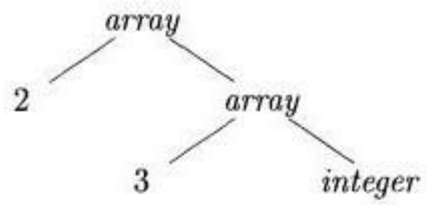


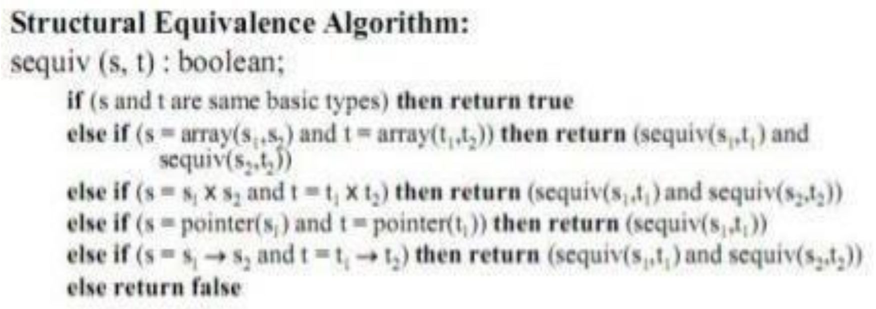
Figure 4.7: Type expression for int [2][3]

We shall use the following definition of type expressions:
• A *basic type* is a type expression. Typical basic types for a language include *boolean, char, integer, float,* and *void.*
• A *type name* is a type expression.

• A *type expression* can be formed by applying the array type constructor to a number and a type expression.
• A *record* is a data structure with named fields. A type expression can be formed by applying the *record* type constructor to the field names and their types.
• A type expression can be formed by using the type constructor → for function types. We write *s* → *t* for "function from type *s* to type t."
• If *s* and *t* are type expressions, then their Cartesian product *s X t* is a type expression.
• Type expressions may contain variables whose values are type expressions.

4.2.2 Type Equivalence

- The basic question is "when are two type expressions equivalent?"
- Two expressions are structurally equivalent if there are two expressions of same basic type or are formed by applying same constructor.



Structural Equivalence Algorithm:

```
sequiv (s, t) : boolean;
    if (s and t are same basic types) then return true
    else if (s = array(s_1,s_2) and t = array(t_1,t_2)) then return (sequiv(s_1,t_1) and
            sequiv(s_2,t_2))
    else if (s = s_1 X s_2 and t = t_1 X t_2) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))
    else if (s = pointer(s_1) and t = pointer(t_1)) then return (sequiv(s_1,t_1))
    else if (s = s_1 → s_2 and t = t_1 → t_2) then return (sequiv(s_1,t_1) and sequiv(s_2,t_2))
    else return false
```

- **Example**: int a, b
  Here a and b are structurally equivalent

4.2.3 Declarations

Types and declarations using a simplified grammar that declares just one name at a time; declarations with lists of names can also be handled. The grammar is

D →T id ; D | ε
T → B C | record '{' D '}'
B → int | float
C → ε | [num] C

- Nonterminal *D* generates a sequence of declarations.
- Nonterminal *T* generates basic, array, or record types.
- Nonterminal *B* generates one of the basic types int and float.
- Nonterminal C, for "component," generates strings of zero or more integers, each integer surrounded by brackets.

An array type consists of a basic type specified by *B,* followed by array components specified by nonterminal *C.*
A record type (the second production for *T)* is a sequence of declarations for the fields of the record, all surrounded by curly braces.

4.2.4 Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that will be needed for the name at run time. At compile time, we can use these amounts to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name. Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data.
The *width* of a type is the number of storage units needed for objects of that type. A basic type, such as a character, integer, or float, requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes is allocated in one contiguous block of bytes.

4.3 Type Checking

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an element along with the value of the element. A ***sound* type** system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is ***strongly typed*** if a compiler guarantees that the programs it accepts will run without type errors.

4.3.1 Rules for Type Checking

Type checking can take on two forms: **synthesis and inference**.

1. ***Type synthesis*** builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the types of E_1 *and* E_2.
2. ***Type inference*** determines the type of a language construct from the way it is used. Let ***null*** be a function that tests whether a list is empty. Then, from the usage *null(x),* we can tell that *x* must be a list. The type of the elements of *x* is not known; all we know is that *x* must be a list of elements of some type that is presently unknown.

4.3.2 Type Conversions

Consider expressions like *x + i,* where *x* is of type float and *i* is of type integer. Since the expression has two different types of operands, the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs. Suppose that integers are converted to floats when necessary, using a unary operator (float). For example, the integer **2** is converted to a float in the code for the expression **2*3.14:**

$$t_1 = (float)\ 2$$
$$t_2 = t_1 * 3.14$$

Type synthesis will be illustrated by extending the scheme for translating expressions. We introduce another attribute *E.type,* whose value is either *integer* or *float.* The rule associated with $E \rightarrow E1+E2$ builds on the pseudocode

```
if ( $E_1.type = integer$ and $E_2.type = integer$ ) $E.type = integer$:
else if ( $E_1.type = float$ and $E_2.type = integer$ ) . . .
```

Type conversion rules vary from language to language. The rules for Java in Fig. 4.16 distinguish between ***widening* conversions**, which are intended to preserve information, and ***narrowing* conversions**, which can lose information.

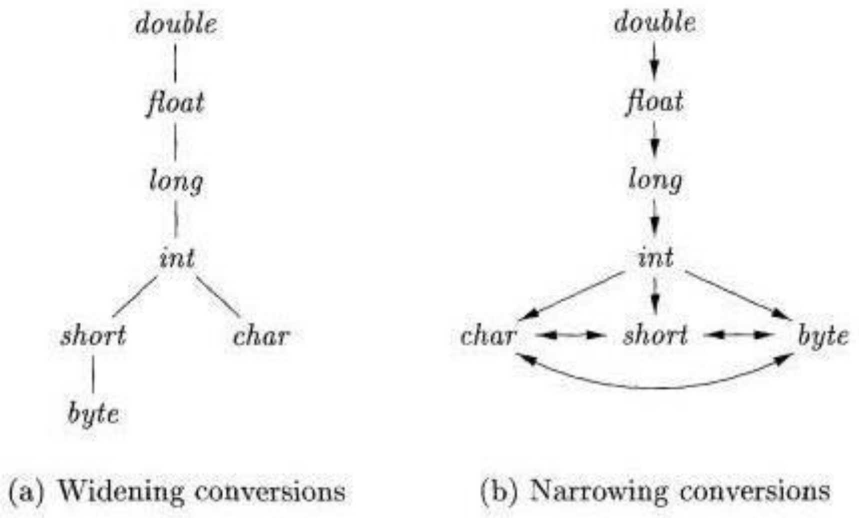


Figure 4.8: Conversions between primitive types in Java

The widening rules are given by the hierarchy in Fig. 4.8(a): any type lower in the hierarchy can be widened to a higher type. Thus, a *char* can be widened to an *int* or to a *float,* but a *char* cannot be widened to a *short.* The narrowing rules are illustrated by the graph in Fig. 4.8(b): a type *s* can be narrowed to a type *t* if there is a path from *s* to *t.* Note that *char, short,* and *byte* are pair wise convertible to each other.

Conversion from one type to another is said to be ***implicit*** if it is done automatically by the compiler. Implicit type conversions, also called ***coercions,*** are limited in many languages to widening conversions. Conversion is said to be ***explicit*** if the programmer must write something to cause the conversion. Explicit conversions are also called ***casts.***

The semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. ***max(t_1,t_2)*** takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.
2. ***widen(a, t, w)*** generates type conversions if needed to widen an address *a* of type t into a value of type *w.* It returns *a* itself if *t* and *w* are the same type. Otherwise, it generates an instruction to dothe conversion and place the result in a temporary *t,* which is returned as the result.

4.4 Control Flow

4.4.1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote &&, II, and ! using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions.
Relational expressions are of the form *E1* rel *E2,* where *E1* and *E2* are arithmetic expressions. We consider boolean expressions generated by the following grammar:

B → B || B | B && B | ! B | (B) | E **rel** E | **true** | **false**

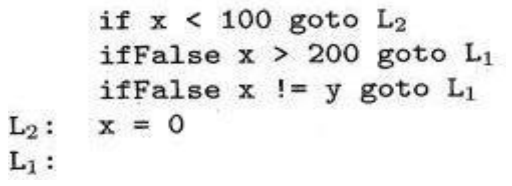
We use the attribute **rel** *op* to indicate which of the six comparison operators <, < = , =, ! =, >, or >= is represented by **rel.** As is customary, we assume that **II** and && are left-associative, and that II has lowest precedence, then &&, then !.
Given the expression *B1* **||** *B2,* if we determine that *B1* is true, and then we can conclude that the entire expression is true without having to evaluate *B2.* Similarly, given *B1&&B2,* if *B1* is false, then the entire expression is false.

4.4.2 Short-Circuit Code

In ***short-circuit*** **(or *jumping)* code**, the boolean operators &&, ||, and ! translate into jumps. The operators themselves do not appear in the code; instead, the value of a boolean expression is represented by a position in the code sequence.
Example: The statement

if (x < 100 || x > 200 && x != y) x = 0;

might be translated into the code.



```
     if x < 100 goto L2
     ifFalse x > 200 goto L1
     ifFalse x != y goto L1
L2:  x = 0
L1:
```

In this translation, the boolean expression is true if control reaches label *L2.* If the expression is false, control goes immediately to *L1,* skipping *L2* and the assignment x = 0.

4.4.4. Flow-of-Control Statements

We now consider the translation of boolean expressions into three-address code in the context of statements such as those generated by the following grammar:

S → **if (** *B*) S_1
S → **if (** *B*) S_1 **else** S_2
S → **while** (B) S_1

In these productions, nonterminal *B* represents a boolean expression and nonterminal S represents a statement.

The translation of **if** *(B)* S_1 consists of *B.code* followed by *S_1.code,* as illustrated in Fig. 6.35(a). Within *B.code* are jumps based on the value of *B.* If *B* is true, control flows to the first instruction of *S_1.code,* and if *B* is false, control flows to the instruction immediately following S_1*.code.*

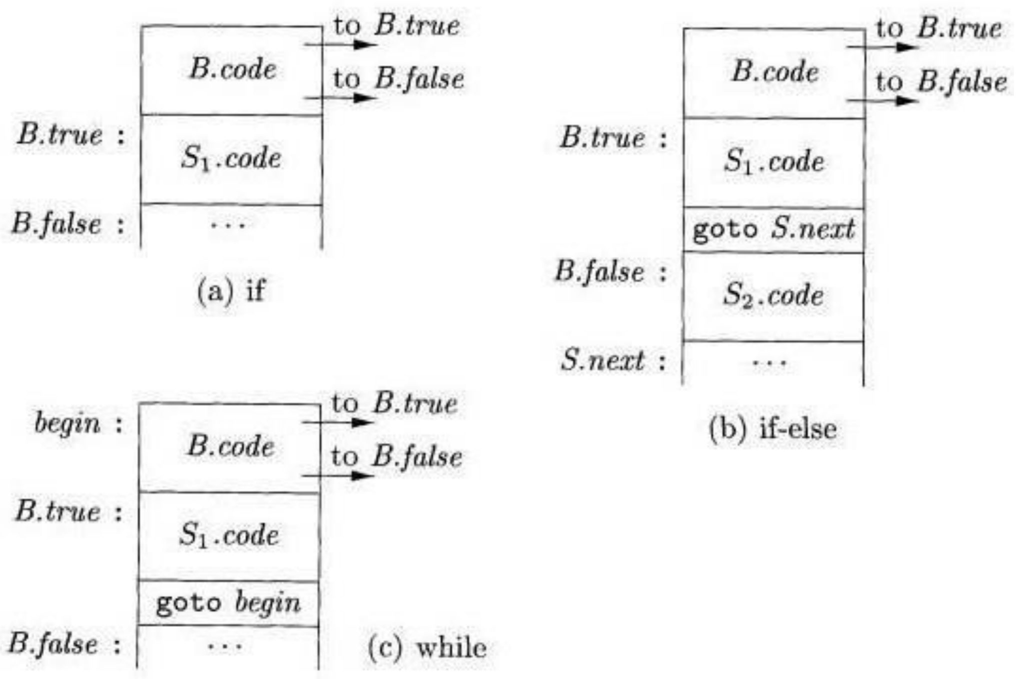


Figure 4.9: Code for if-, if-else-, and while-statements

The labels for the jumps in *B.code* and *S.code* are managed using inherited attributes. With a boolean expression *B,* we associate two labels: *B.true,* the label to which control flows if *B* is true, and *B.false,* the label to which control flows if *B* is false. With a statement S, we associate an inherited attribute *S.next* denoting a label for the instruction immediately after the code for *S.* In some cases, the instruction immediately following *S.code* is a jump to some label *L.* A jump to a jump to *L* from within *S.code* is avoided using *S.next.*

4.5 Backpatching:

It is the process of filling up the unspecified labels.

The following functions are required for backpatching:

1. ***makelist(i)*** creates a new list containing only *i,* an index into the array of instructions; *makelist* returns a pointer to the newly created list.
2. ***merge(p₁,p₂)*** concatenates the lists pointed to by p_1 and p_2, and returns a pointer to the concatenated list.
3. ***backpatch(p,i)*** inserts *i* as the target label for each of the instructions on the list pointed to by *p.*

We now construct a translation scheme suitable for generating code for boolean expressions during bottom-up parsing. A marker nonterminal *M* in the grammar causes a semantic action to pick up, at appropriate times, the index of the next instruction to be generated. The grammar is as follows:

$B \rightarrow B_1$ || M B_2 | B_1 && M B_2 | ! B_1 | (B_1) || E1 rel E2 | true | false

$M \rightarrow \varepsilon$

The translation scheme is in below figure 4.10.

1)	$B \rightarrow B_1 \mid\mid M\ B_2$	$\{\ backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;\ \}$
2)	$B \rightarrow B_1\ \&\&\ M\ B_2$	$\{\ backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);\ \}$
3)	$B \rightarrow\ !\ B_1$	$\{\ B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;\ \}$
4)	$B \rightarrow (\ B_1\ )$	$\{\ B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist;\ \}$
5)	$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $emit(\texttt{'if'}\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ \texttt{'goto \_'});$ $emit(\texttt{'goto \_'});\ \}$
6)	$B \rightarrow \mathbf{true}$	$\{\ B.truelist = makelist(nextinstr);$ $emit(\texttt{'goto \_'});\ \}$
7)	$B \rightarrow \mathbf{false}$	$\{\ B.falselist = makelist(nextinstr);$ $emit(\texttt{'goto \_'});\ \}$
8)	$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

Figure 4.10: Translation scheme for boolean expressions

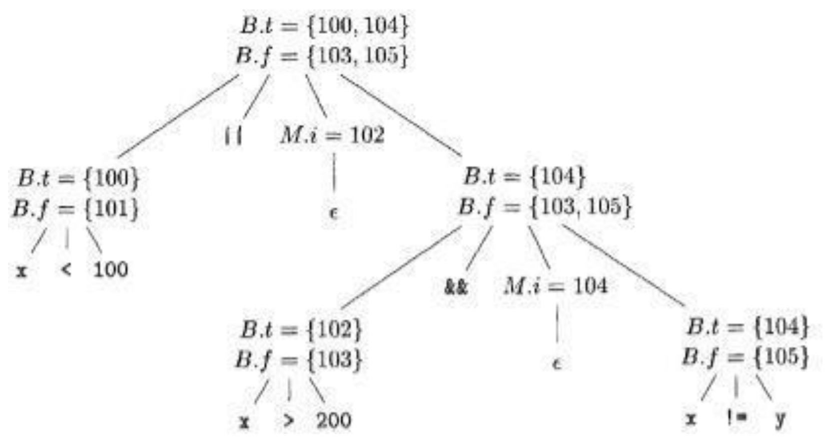
Consider semantic action (1) for the production $B \rightarrow B_1 \mid\mid M\ B_2$. If B_1 is true, then B is also true, so the jumps on B_1*.truelist* become part of *B.truelist.* If B_1 is false, however, we must next test B_2, so the target for the jumps B_1*.falselist* must be the beginning of the code generated for B_2. This target is obtained using the marker nonterminal *M.* That nonterminal produces, as a synthesized attribute *M.instr,* the index of the next instruction, just before B_2 code starts being generated.
To obtain that instruction index, we associate with the production $M \rightarrow \varepsilon$ the semantic action

{ M.instr = nextinstr; }

The variable *nextinstr* holds the index of the next instruction to follow. This value will be backpatched onto the B_1*.falselist* (i.e., each instruction on the list B_1*.falselist* will receive *M.instr* as its target label) when we have seen the remainder of the production $B \rightarrow B_1 \mid\mid M\ B_2$.

Semantic action (2) for $B \rightarrow B_1\ \&\&\ M\ B_2$ is similar to (1). Action (3) for $B \rightarrow\ !B_1$ swaps the true and false lists. Action (4) ignores parentheses. For simplicity, semantic action (5) generates two instructions, a conditional goto and an unconditional one. Neither has its target filled in. These instructions are put on new lists, pointed to by *B.truelist* and *B.falselist,* respectively.

Example: x<100 || x>200 && x!=y
Solution: Generate TAC for the given expression
100 if x<100 goto ______
101 goto
102 if x>200 goto ______
103 goto
104 if x!=y goto ________
105 goto



100 if x<100 goto ___
101 goto 102
102 if x>200 goto 104
103 goto ___
104 if x!=y goto ___
105 goto ____

UNIT – V

Runtime Environments, Stack allocation of space, access to Non Local date on the stack Heap Management code generation – Issues in design of code generation the target Language Address in the target code Basic blocks and Flow graphs. A Simple Code generation.

5.1 Storage Organization

From the perspective of the compiler writer, the executing target program runs in its own logical address space in which each program value has a location. The management and organization of this logical address space is shared between the compiler, operating system, and target machine. The operating system maps the logical addresses into physical addresses, which are usually spread throughout memory.

The run-time representation of an object program in the logical address space consists of data and program areas as shown in Fig. 5.1.

The size of the generated target code is fixed at compile time, so the compiler can place the executable target code in a statically determined area ***Code,*** usually in the low end of memory.

Similarly, the size of some program data objects, such as global constants, and data generated by the compiler, such as information to support garbage collection, may be known at compile time, and these data objects can be placed in another statically determined area called ***Static***. One reason for statically allocating as many data objects as possible is that the addresses of these objects can be compiled into the target code.

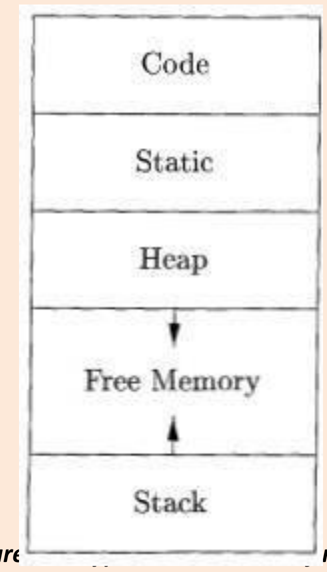


***Figure run-
time memory into code and data areas***

To maximize the utilization of space at run time, the other two areas, ***Stack*** **and** ***Heap,*** are at the opposite ends of the remainder of the address space. These areas are dynamic; their size can change as the program executes. These areas grow towards each other as needed. The stack is used to store data structures called activation records that get generated during procedure calls. In practice, the stack grows towards lower addresses, the heap towards higher.

5.1.1 Static versus Dynamic Storage Allocation

The layout and allocation of data to memory locations in the run-time environment are key issues in storage management. We say that a storage-allocation decision is ***static***, if it can be made by the compiler looking only at the text of the program, not at what the program does when it executes. Conversely, a decision is ***dynamic*** if it can be decided only while the program is running. Many compilers use some combination of the following two strategies for dynamic storage allocation:

1. ***Stack storage***. Names local to a procedure are allocated space on a stack. The stack supports the normal call/return policy for procedures.

2. ***Heap storage***. Data that may outlive the call to the procedure that created it is usually allocated on a "heap" of reusable storage.

5.2 Stack Allocation of Space

Almost all compilers for languages that use procedures, functions, or methods as units of user-defined actions manage at least part of their run-time memory as a stack. Each time a procedure is called, space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack. This arrangement not only allows space to be shared by procedure calls whose durations do not overlap in time, but it allows us to compile code for a procedure in such a way that the relative addresses of its nonlocal variables are always the same, regardless of the sequence of procedure calls.

5.2.1 Activation Trees

Stack allocation would not be feasible if procedure calls, or *activations* of procedures, did not nest in time. The following example illustrates nesting of procedure calls.

Example 5.1: Figure 5.2 contains a sketch of a program that reads nine integers into an array *a* and sorts them using the recursive quicksort algorithm.

```
int a[11];
void readArray() { /* Reads 9 integers into a[1],...,a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Figure 5.2: Sketch of a quicksort program

The main function has three tasks. It calls *readArray,* sets the sentinels, and then calls *quicksort* on the entire data array.

Figure 5.3 suggests a sequence of calls that might result from an execution of the program. In this execution, the call to *partition(1, 9)* returns 4, so a[1]through a[3] hold elements less than its chosen separator value *v,* while the larger elements are in a[5] through a[9].

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Figure 5.3: Possible activations for the program of Fig.5.2

In this example, procedure activations are nested in time. If an activation of procedure *p* calls procedure *q,* then that activation of *q* must end before the activation of *p* can end. There are three common cases:

1. The activation of *q* terminates normally. Then in essentially any language, control resumes just after the point of *p* at which the call to *q* was made.
2. The activation of *q,* or some procedure *q* called, either directly or indirectly, aborts; i.e., it becomes impossible for execution to continue. In that case, *p* ends simultaneously with *q.*
3. The activation of q terminates because of an exception that *q* cannot handle. Procedure p may handle the exception, in which case the activation of *q* has terminated while the activation of *p* continues, although not necessarily from the point at which the call to *q* was made. If *p* cannot handle the exception, then this activation of *p* terminates at the same time as the activation of *q,* and presumably the exception will be handled by some other open activation of a procedure.

We therefore can represent the activations of procedures during the running of an entire program by a tree, called an ***activation tree***. Each node corresponds to one activation, and the root is the activation of the "main" procedure that initiates execution of the program. At a node for an activation of procedure *p,* the children correspond to activations of the procedures called by this activation of *p.* We show these activations in the order that they are called, from left to right. Notice that one child must finish before the activation to its right can begin.

One possible activation tree that completes the sequence of calls and returns suggested in Fig. 5.3 is shown in Fig. 5.4. Functions are represented by the first letters of their names. Remember that this tree is only one possibility, since the arguments of subsequent calls, and also the number of calls along any branch is influenced by the values returned by ***partition***.

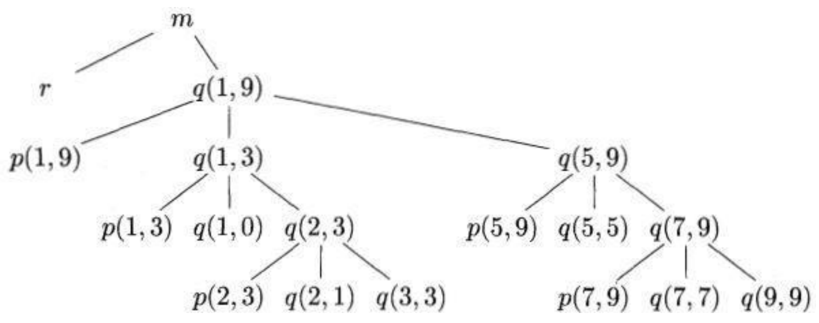


Figure 5.4: Activation tree representing calls during an execution of quicksort

The use of a run-time stack is enabled by several useful relationships between the activation tree and the behavior of the program:

1. The sequence of procedure calls corresponds to a preorder traversal of the activation tree.
2. The sequence of returns corresponds to a postorder traversal of the activation tree.
3. Suppose that control lies within a particular activation of some procedure, corresponding to a node *N* of the activation tree. Then the activations that are currently open *(live)* are those that correspond to node *N* and its ancestors. The order in which these activations were called is the order in which they appear along the path to *N,* starting at the root, and they will return in the reverse of that order.

5.2.2 Activation Records

Procedure calls and returns are usually managed by a run-time stack called the ***control stack****.* Each live activation has an ***activation record*** (sometimes called a ***frame****)* on the control stack, with the root of the activation tree at the bottom, and the entire sequence of activation records on the stack corresponding to the path in the activation tree to the activation where control currently resides. The latter activation has its record at the top of the stack.

If control is currently in the activation *q(2,*3) of the tree of Fig. 5.4, then the activation record for *q(2,*3) is at the top of the control stack. Just below is the activation record for q(1,3), the parent of q(2,3) in the tree. Below that is the activation record q(1,9), and at the bottom is the activation record for m, the main function and root of the activation tree.

We shall conventionally draw control stacks with the bottom of the stack higher than the top, so the elements in an activation record that appear lowest on the page are actually closest to the top of the stack.

The contents of activation records vary with the language being implemented. Here is a list of the kinds of data that might appear in an activation record, as shown in Fig.5.5:

1. ***Temporary values***, such as those arising from the evaluation of expressions, in cases where those temporaries cannot be held in registers.
2. ***Local data*** belonging to the procedure whose activation record this is.
3. A ***saved machine status***, with information about the state of the machine just before the call to the procedure. This information typically includes the *return address* (value of the program counter, to which the called procedure must return) and the contents of registers that were used by the calling procedure and that must be restored when the return occurs.
4. An "***access link***" may be needed to locate data needed by the called procedure but found elsewhere, e.g., in another activation record.
5. ***A control link,*** pointing to the activation record of the caller.

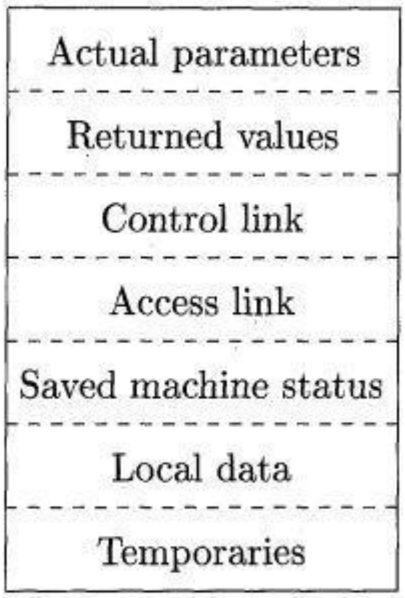


Figure 5.5: A general activation record

6. Space for the ***return value*** of the called function, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.

7. The ***actual parameters*** used by the calling procedure. Commonly, these values are not placed in the activation record but rather in registers, when possible, for greater efficiency. However, we show a space for them to be completely general.

Figure 5.6 shows snapshots of the run-time stack as control flows through the activation tree of Fig. 5.4. Dashed lines in the partial trees go to activations that have ended. Since array *a* is global, space is allocated for it before execution begins with an activation of procedure *main,* as shown in Fig. 5.6(a).

When control reaches the first call in the body of *main,* procedure r is activated, and its activation record is pushed onto the stack (Fig. 5.6(b)). The activation record for r contains space for local variable *i.* Recall that the top of stack is at the bottom of diagrams. When control returns from this activation, its record is popped, leaving just the record for *main* on the stack.

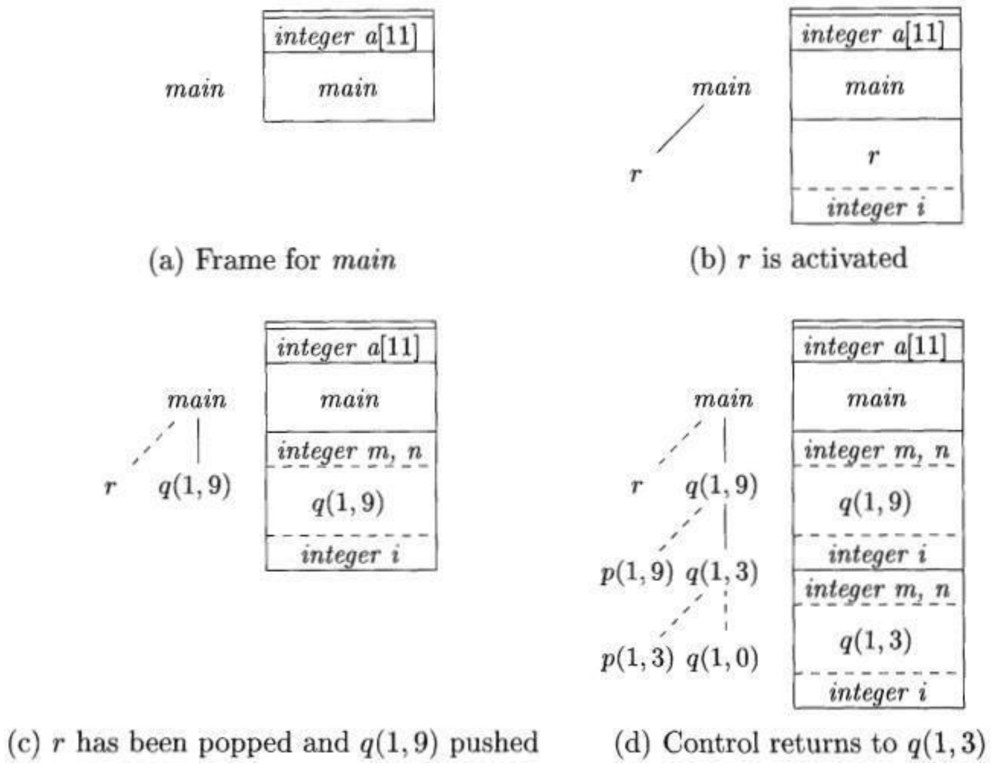


Figure 5.6: Downward-growing stack of activation records

Control then reaches the call to *q* (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack, as in Fig. 5.6(c). The activation record for *q* contains space for the parameters *m* and *n* and the local variable *i,* following the general layout in Fig. 5.5. Notice that space once used by the call of r is reused on the stack. No trace of data local to r will be available to *q(1,* 9). When *q(1,* 9) returns, the stack again has only the activation record for *main.*

Several activations occur between the last two snapshots in Fig. 5.6. A recursive call to q(1,3) was made. Activations p(1,3) and *q(1,0)* have begun and ended during the lifetime of *q(1,*3), leaving the activation record for *q(1,*3) on top (Fig.5.6(d)). Notice that when a procedure is recursive, it is normal to have several of its activation records on the stack at the same time.

5.2.3 Calling Sequences

Procedure calls are implemented by what are known as ***calling sequences,*** which consists of code that allocates an activation record on the stack and enters information into its fields. A ***return sequence*** is similar code to restore the state of the machine so the calling procedure can continue its execution after the call.

The code in a calling sequence is often divided between the calling procedure (the "caller") and the procedure it calls (the "callee").

An example of how caller and callee might cooperate in managing the stack is suggested by Fig. 5.7. A register *top_sp* points to the end of the machine status field in the current top activation record. This position within the callee's activation record is known to the caller, so the caller can be made responsible for setting *top_sp* before control is passed to the callee.

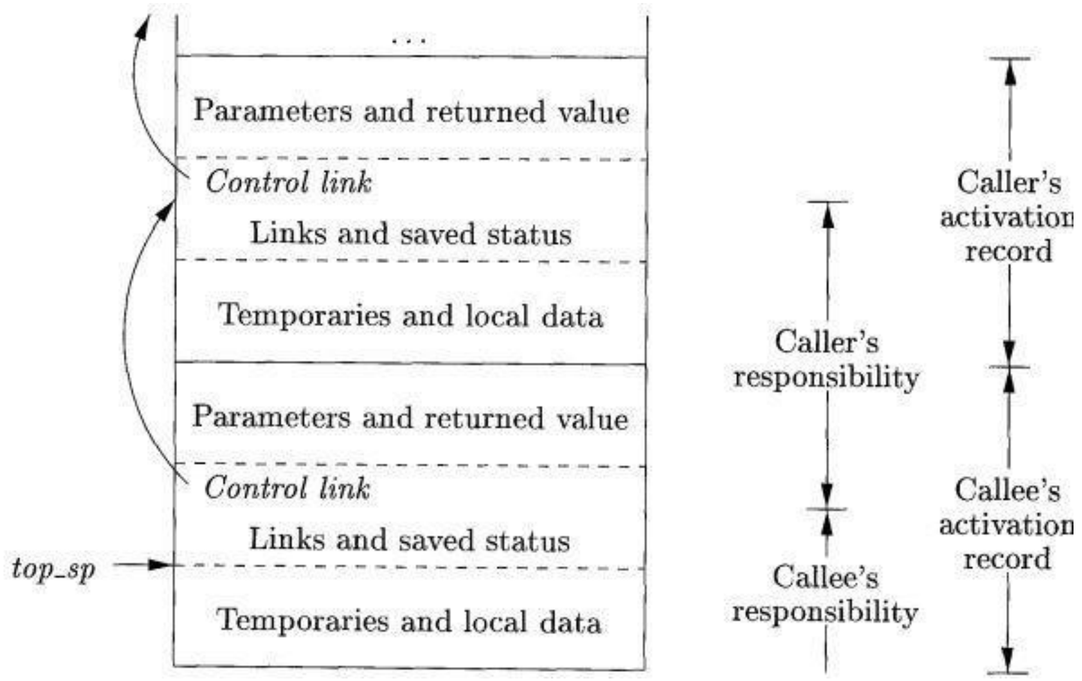

. . .
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
Parameters and returned value
Control link
Links and saved status
Temporaries and local data
top_sp
Caller's responsibility
Callee's responsibility
Caller's activation record
Callee's activation record

Figure 5.7: Division of tasks between caller and callee

The **calling sequence** and its division between caller and callee is as follows:

1. The caller evaluates the actual parameters.
2. The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments *to_sp* to the position. That is, *top_sp* is moved past the caller's local data and temporaries and the callee's parameters and status fields.
3. The callee saves the register values and other status information.
4. The callee initializes its local data and begins execution.

A suitable, corresponding **return sequence** is:

1. The callee places the return value next to the parameters, as in Fig. 5.5.
2. Using information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
3. Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp;* the caller therefore may use that value.

The above calling and return sequences allow the number of arguments of the called procedure to vary from call to call.

5.2.4 Variable-Length Data on the Stack

The run-time memory-management system must deal frequently with the allocation of space for objects the sizes of which are not known at compile time, but which are local to a procedure and thus may be allocated on the stack.

A common strategy for allocating variable-length arrays is shown in Fig. 5.8. The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.

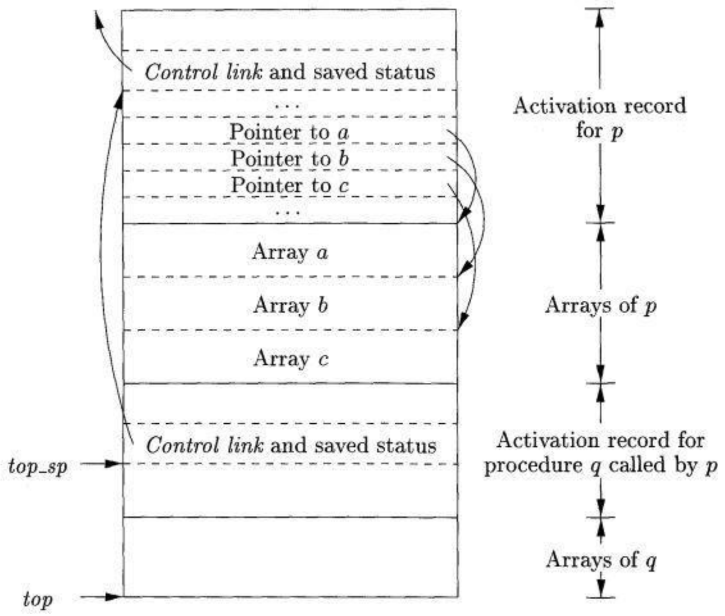


Figure 5.8: Access to dynamically allocated arrays

Also shown in Fig. 5.8 is the activation record for a procedure *q,* called by *p.* The activation record for *q* begins after the arrays of p, and any variable-length arrays of *q* are located beyond that.

Access to the data on the stack is through two pointers, ***top*** and ***top_sp***. Here,

- ***top*** marks the actual top of stack; it points to the position at which the next activation record will begin.
- ***top_sp*** is used to find local, fixed-length fields of the top activation record.

In Fig. 5.8, *top_sp* points to the end of this field in the activation record for *q.* From there, we can find the control-link field for *q,* which leads us to the place in the activation record for *p* where *top_sp* pointed when *p* was on top. The code to reposition *top* and *top_sp* can be generated at compile time, in terms of sizes that will become known at run time. When *q* returns, *top_sp* can be restored from the saved control link in the activation record for *q.* The new value of *top* is (the old unrestored value of) *top_sp* minus the length of the machine-status, control and access link, return-value, and parameter fields (as in Fig. 5.5) in q's activation record. This length is known at compile time to the caller, although it may depend on the caller, if the number of parameters can vary across calls to *q.*

5.3 Access to Nonlocal Data on the Stack

5.3.1 Data Access without Nested Procedures

In the C family of languages, all variables are defined either within a single function or outside any function ("globally"). Variables declared within a function have a scope consisting of that function only, or part of it, if the function has nested blocks.

For languages that do not allow nested procedure declarations, allocation of storage for variables and access to those variables is simple:

1. Global variables are allocated static storage. The locations of these variables remain fixed and are known at compile time. So to access any variable that is not local to the currently executing procedure, we simply use the statically determined address.
2. Any other name must be local to the activation at the top of the stack.

We may access these variables through the *top_sp* pointer of the stack. With the C static-scoping rule, and without nested procedures, any name nonlocal to one procedure is nonlocal to all procedures, regardless of how they are activated. Similarly, if a procedure is returned as a result, then any nonlocal name refers to the storage statically allocated for it.

5.3.2 Issues with Nested Procedures

Access becomes far more complicated when a language allows procedure declarations to be nested and also uses the normal static scoping rule; that is, a procedure can access variables ofthe procedures whose declarations surround its own declaration, following the nested scoping rule.

Finding the declaration that applies to a nonlocal name x in a nested procedure p is a **static decision**; it can be done by an extension of the static-scope rule for blocks. Suppose x is declaredin the enclosing procedure q. Finding the relevant activation of q from an activation of p is a **dynamic decision**; it requires additional run-time information about activations. One possible solution to this problem is to use "**access links**."

5.3.3 Access Links

A direct implementation of the normal static scope rule for nested functions is obtained by adding a pointer called the ***access link*** to each activation record. If procedure *p* is nested immediately within procedure *q* in the source code, then the access link in any activation of *p* points to the most recent activation of *q*. **Note that the nesting depth of *q* must be exactly one less than the nesting depth of p.**

Suppose that the procedure *p* at the top of the stack is at nesting depth n_p, and *p* needs to access *x,* which is an element defined within some procedure *q* that surrounds *p* and has nesting depth n_q. Note that $n_q < n_p$, with equality only if *p* and *q* are the same procedure. To find *x,* we start at the activation record for *p* at the top of the stack and follow the access link $n_p — n_q$ times, from activation record to activation record. Finally, we wind up at an activation record for *q,* and it will always be the most recent (highest) activation record for *q* that currently appears on the stack. This activation record contains the element *x* that we want. Since the compiler knows the layout of activation records, *x* will be found at some fixed offset from the position in g's activation record that we can reach by following the last access link.

Figure 5.9 shows a sequence of stacks that might result from execution of the function *sort.* As before, we represent function names by their first letters, and we show some of the data that might appear in the various activation records, as well as the access link for each activation. In Fig. 5.9(a), we see the situation after *sort* has called *readArray* to load input into the array *a* and then called *quicksort(1,*9) to sort the array. The access link from *quicksort(1,*9) points to the activation record for *sort,* not because *sort* called *quicksort* but because *sort* is the most closely nested function.

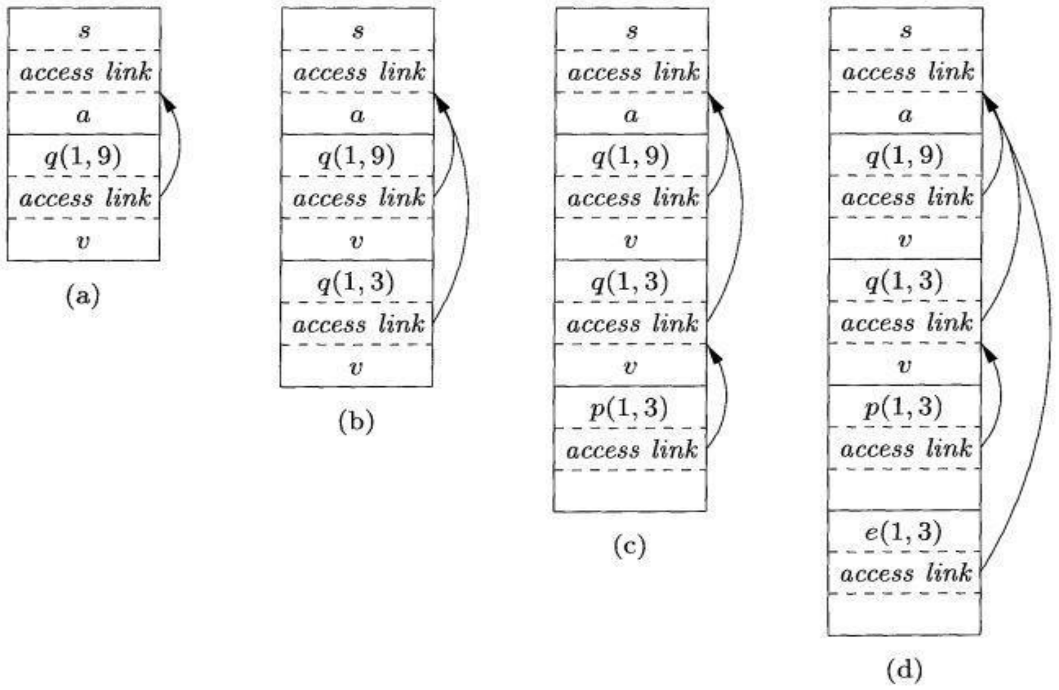


Figure 5.9: Access links for finding nonlocal data

In successive steps of Fig. 5.9 we see a recursive call to *quicksort(1* 3), followed by a call to *partition,* which calls *exchange.* Notice that *quicksort(1,* 3)'s access link points to *sort,* for the samereason that *quicksort(1,* 9)'s does.

In Fig. 5.9(d), the access link for *exchange* bypasses the activation records for *quicksort* and *partition,* since *exchange* is nested immediately within *sort.* That arrangement is fine, since *exchange* needs to access only the array *a,* and the two elements it must swap are indicated by its own parameters *i* and *j.*

5.4 Heap Management

The heap is the portion of the store that is used for data that lives indefinitely, or until the program explicitly deletes it.

5.4.1 The Memory Manager

The memory manager keeps track of all the free space in heap storage at all times. It performs two basic functions:

- ***Allocation.*** When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size. If possible, itsatisfies an allocation request using free space in the heap; if no chunk of the needed size is available, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.
- ***Deallocation.*** The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests.

Properties of memory manager:

- ***Space Efficiency***. A memory manager should minimize the total heap space needed by a program. Doing so allows larger programs to run in a fixed virtual address space. Space efficiency is achieved by minimizing "fragmentation."
- ***Program Efficiency***. A memory manager should make good use of the memory subsystem to allow programs to run faster.
- ***Low Overhead***. Because memory allocations and deallocations are frequent operations in many programs, it is important that these operations be as efficient as possible.

5.4.2 The Memory Hierarchy of a Computer

Memory management and compiler optimization must be done with an awareness of how memory behaves. The large variance in memory access times is due to the fundamental limitation in hardware technology; we can build small and fast storage, or large and slow storage, but not storage that is both large and fast. All modern computers arrange their storage as a ***memory hierarchy***. A memory hierarchy, as shown in Fig. 5.10, consists of a series of storage elements,with the smaller faster ones "closer" to the processor, and the larger slower ones further away.

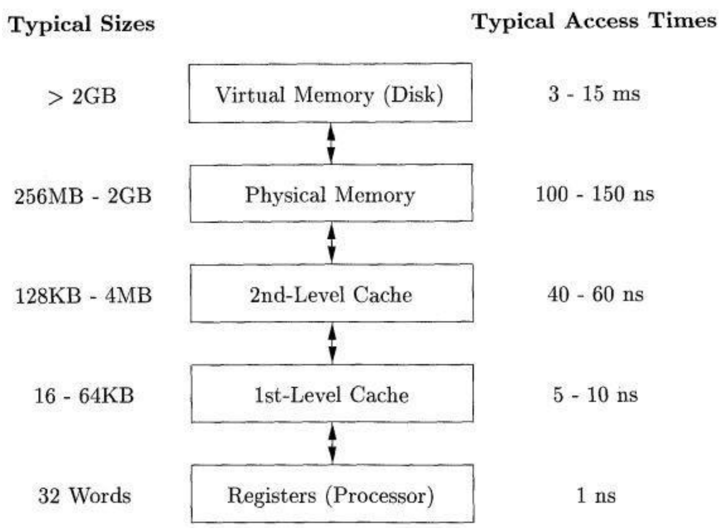


Figure 5.10: Typical Memory Hierarchy Configurations

Typically, a processor has a small number of registers, whose contents are under software control. Next, it has one or more levels of cache, usually made out of static RAM, that are kilobytes to several megabytes in size. The next level of the hierarchy is the physical (main) memory, made out of hundreds of megabytes or gigabytes of dynamic RAM. The physical memory is then backed up by virtual memory, which is implemented by gigabytes of disks. Upon a memory access, the machine first looks for the data in the closest (lowest-level) storage and, if the data is not there, looks in the next higher level, and so on.

5.4.3 Locality in Programs

Most programs exhibit a high degree of ***locality;*** that is, they spend most of their time executing a relatively small fraction of the code and touching only a small fraction of the data.

- We say that a program has ***temporal locality,*** if at one point a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.
- We say that a program has ***spatial locality,*** if a particular storage location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.

The conventional wisdom is that programs spend 90% of their time executing 10% of the code. Here is why:

- Programs often contain many instructions that are never executed.
- Only a small fraction of the code that could be invoked is actually executed in a typical run of the program.
- The typical program spends most of its time executing innermost loops and tight recursive cycles in a program.

Locality allows us to take advantage of the memory hierarchy of a modern computer. By placing the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage, we can lower the average memory-access time of a program significantly.

Optimization Using the Memory Hierarchy
The policy of keeping the most recently used instructions in the cache tends to work well. When a new instruction is executed, there is a high probability that the next instruction also will be executed. This phenomenon is an example of spatial locality. One effective technique to improve the spatial locality of instructions is to have the compiler place basic blocks (sequences of instructions that are always executed sequentially) that are likely to follow each other contiguously — on the same page, or even the same cache line, if possible. Instructions belonging to the same loop or same function also have a high probability of being executed together.

We can also improve the temporal and spatial locality of data accesses in a program by changing the data layout or the order of the computation. For example, programs that visit large amounts of data repeatedly, each time performing a small amount of computation, do not perform well. Itis better if we can bring some data from a slow level of the memory hierarchy to a faster level (e.g., disk to main memory) once, and perform all the necessary computations on this data while it resides at the faster level. This concept can be applied recursively to reuse data in physicalmemory, in the caches and in the registers.

5.4.4 Reducing Fragmentation
At the beginning of program execution, the heap is one contiguous unit of free space. As the program allocates and deallocates memory, this space is broken up into free and used chunks of memory, and the free chunks need not reside in a contiguous area of the heap. We refer to the free chunks of memory as ***holes.*** With each allocation request, the memory manager must *place* the requested chunk of memory into a large-enough hole. Unless a hole of exactly the right size is found, we need to *split* some hole, creating a yet smaller hole.

With each deallocation request, the freed chunks of memory are added back to the pool of free space. We combine contiguous holes into larger holes, as the holes can only get smaller otherwise. If we are not careful, the memory may end up getting ***fragmented,*** consisting of large numbers of small, noncontiguous holes. It is then possible that no hole is large enough to satisfy a future request, even though there may be sufficient aggregate free space.

We reduce fragmentation by controlling how the memory manager places new objects in the heap. It has been found empirically that a good strategy for minimizing fragmentation for real-life programs is to allocate the requested memory in the smallest available hole that is large enough. This ***best-fit*** algorithm tends to spare the large holes to satisfy subsequent, larger requests. An alternative, called ***first-fit,*** where an object is placed in the first (lowest-address) hole in which it fits, takes less time to place objects, but has been found inferior to best-fit in overall performance.

Coalescing Free Space

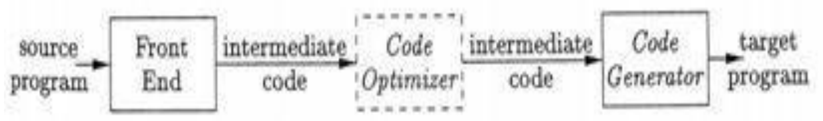
When an object is deallocated manually, the memory manager must make its chunk free, so it can be allocated again. In some circumstance, it may also be possible to combine (coalesce) that chunk with adjacent chunks of the heap, to form a larger chunk. The advantage is that a larger chunk can hold a larger object whereas smaller chunk cannot.

Problems with Manual Deallocation

Manual memory management is error-prone. The common mistakes take two forms: failing ever to delete data that cannot be referenced is called a *memory leak* error, and referencing deleted data is a *dangling-pointer-dereference* error.

Code Generation

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



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

A code generator has three primary tasks:

1. **Instruction selection** involves choosing appropriate target-machine instructions to implement the IR statements.
2. **Register allocation and assignment** involves deciding what values to keep in which registers.
3. **Instruction ordering** involves deciding in what order to schedule the execution of instructions.

5.5 Issues in the Design of a Code Generator

The most important criterion for a code generator is that it produces correct code. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are

- **RISC** (reduced instruction set computer): A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction- set architecture.
- **CISC** (complex instruction set computer): In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.
- **Stack based**: In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers.

5.5.1 Instruction Selection

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

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

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

```
LD R0, y          // R0 = y (load y into register R0)
ADD R0, R0, z     // R0 = R0 + z (add z t o R0)
ST x, R0          // x = R0 (store R0 into x)
```

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

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

would be translated into

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

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

The quality of the generated code is usually determined by its speed and size. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

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

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

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

5.5.2 Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two sub-problems:

1. ***Register allocation,*** during which we select the set of variables that will reside in registers at each point in the program.
2. ***Register assignment,*** during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Certain machines require *register-pairs* (an even and next odd numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs.

The multiplication instruction is of the form

M x, y

where x, the multiplicand, is the even register of an even/odd register pair and y, the multiplier, is the odd register. The product occupies the entire even/odd register pair.

The division instruction is of the form

D x, y

where the dividend occupies an even/odd register pair whose even register is x; the divisor is y. After division, the even register holds the remainder and the odd register the quotient.

Now, consider the two three-address code sequences in which the only difference in (a) and (b) is the operator in the second statement.

```
t = a + b          t = a + b
t = t * c          t = t + c
t = t / d          t = t / d
(a)                  (b)
```

The shortest assembly-code sequences for (a) and (b) are given

```
L R1,a             L R0, a
A R1,b             A R0, b
M R0,c             A R0, c
D R0,d             SRDA R0, 32
ST R1,t            D R0, d
                   ST R1, t
(a)                 (b)
```

R_i stands for register *i.* SRDA stands for Shift-Right-Double-Arithmetic and SRDA R0, 32 shifts the dividend into R1 and clears R0 so all bits equal its sign bit. L, ST, and A stand for load, store, and add, respectively. Note that the optimal choice for the register into which a is to be loaded depends on what will ultimately happen to t.

5.5.3 Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator.

5.6 The Target Language

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

5.6.1 A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with *n* general-purpose registers, R0,R1,... ,Rn - 1.

We assume the following **kinds of instructions** are available:

- ***Load operations***: The instruction **LD *dst, addr*** loads the value in location *addr* into location *dst.* The most common form of this instruction is **LD** *r, x* which loads the value in location *x* into register r. An instruction of the form **LD** *r1, r2* is a ***register-to-register copy*** in which the contents of register *r2* are copied into register *r1.*
- ***Store* operations**: The instruction **ST *x, r*** stores the value in register *r* into the location *x.* This instruction denotes the assignment *x = r.*
- ***Computation* operations** of the form ***OP dst, src1, src2,*** where *OP* is an operator like **ADD** or **SUB,** and *dst, src1,* and *src2* are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP to* the values in locations *src1* and *src2,* and place the result of this operation in location *dst.* For example, **SUB** *n, r2, r3* computes *n = r2 - r3.* Any value formerly stored in *n* is lost, but if *r1* is r2 or r3, the old value is read first. Unary operators that take only one operand do not have a *src2.*
- ***Unconditional jumps****:* The instruction **BR *L*** causes control to branch to the machine instruction with label *L.* (BR stands for *branch.)*
- ***Conditional jumps*** of the form *Bcond* r, *L,* where *r* is a register, *L* is a label, and *cond* stands for any of the common tests on values in the register r. For example, BLTZ r, *L* causes a jump to label *L* if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of **addressing modes**:

- In instructions, a location can be a **variable name *x* referring to the memory location** that is reserved for *x* (that is, the l-value of *x).*
- A location can also be an **indexed address** of the form ***a(r),*** where *a* is a variable and r is a register. The memory location denoted by *a(r)* is computed by taking the I-value of *a* and adding to it the value in register r. For example, the instruction ***LD R1, a(R2)*** has the effect of setting **R1 = *contents* (a + *contents* (R2)),** where *contents (x)* denotes the contents of the register or memory location represented by *x.* This addressing mode is useful for accessing arrays, where *a* is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array *a.*
- A memory location can be an integer **indexed by a register**. For example, ***LD R1, 100(R2)*** has the effect of setting ***R1 = contents(100 + contents(R2)),*** that is, of loading into R1 the value in the memory location obtained by adding **100** to the contents of register **R2.**
- We also allow **two indirect addressing modes**: **r* means the memory location found in the location represented by the contents of register r and **100(r)* means the memory location found in the location obtained by adding **100** to the contents of r. For example, LD R1, ***100(R2)** has the effect of setting R1 = *contents(contents(10Q + contents***(R2))),** that is, of loading into R1 the value in the memory location stored in the memory location obtained by adding **100** to the contents of register **R2.**

- Finally, we allow an **immediate constant addressing mode**. The constant is prefixed by #. The instruction ***LD Rl, #100*** loads the integer **100** into register R1, and ***ADD R1, R1, #100*** adds the integer **100** into register R1. '

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

Example 1: The three-address statement x = y - z can be implemented by the machine instructions:

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

One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible. For example, y and/or z may have been computed in a register, and if so we can avoid the **LD** step(s). Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed.

Example 2: Suppose 'a' is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of a are indexed starting at 0. We may execute the three-address instruction b = a [i] by the machine instructions:

```
LD R1, I                 // R1 = i
MUL R1, R1, 8            // R1 = R1 * 8
LD R2, a(R1)             // R2 = contents(a + contents(R1))
ST b, R2                // b = R2
```

That is, the second step computes **8**$_i$, and the third step places in register **R2** the value in the ith element of a — the one found in the location that is **8**$_i$ bytes past the base address of the array a.

Example 3*:* Similarly, the assignment into the array a represented by three-address instruction *a [j] = c* is implemented by:

```
LD R1, c            // R1 = c
LD R2, j            // R2 = j
MUL R2, R2, 8       // R2 = R2 * 8
ST a(R2), R1         // contents(a + contents(R2)) = R1
```

Example 4*:* To implement a simple pointer indirection, such as the three-address statement *x = *p*, we can use machine instructions like:

```
LD R1, p             // R1 = p
LD R2, 0(R1)         // R2 = contents(0 + contents(R1))
ST x, R2            // x = R2
```

Example 5*:* The assignment through a pointer **p = y* is similarly implemented in machine code by:

```
LD R1, p            // R1 = p
LD R2, y            // R2 = y
ST 0(R1), R2        // contents(0 + contents(R1)) = R2
```

Example 6*:* Finally, consider a conditional-jump three-address instruction like if x < y goto L
The machine-code equivalent would be something like:

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

Here, M is the label that represents the first machine instruction generated from the three-address instruction that has label L. As for any three-address instruction, we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored.

5.6.2 Program and Instruction Costs

For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction. Some examples:

- The instruction ***LD R0, R1*** copies the contents of register R1 into register R0. This **instruction has a cost of one** because no additional memory words are required.
- The instruction ***LD R0, M*** loads the contents of memory location M into register R0. The **cost is two** since the address of memory location M is in the word following the instruction.
- The instruction ***LD R1, *100(R2)*** loads into register R1 the value given by *contents(contents(100 + contents(R2))).* The **cost is three** because the constant 100 is stored in the word following the instruction.

Good code-generation algorithms seek to ***minimize the sum of the costs of the instructions*** executed by the generated target program on typical inputs.

5.7 Addresses in the Target Code

Executing program runs in its own logical address space that was partitioned into four code and data areas:

1. A statically determined area ***Code*** that holds the executable target code. The size of the target code can be determined at compile time.
2. A statically determined data area ***Static*** for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.
3. A dynamically managed area ***Heap*** for holding data objects that are allocated and freed during program execution. The size of the *Heap* cannot be determined at compile time.
4. A dynamically managed area ***Stack*** for holding activation records as they are created and destroyed during procedure calls and returns. Like the *Heap,* the size of the *Stack* cannot be determined at compile time.

To illustrate code generation for simplified procedure calls and returns, we shall focus on the following three-address statements:

- **call** *callee*
- **return**
- **halt**
- **action**, which is a placeholder for other three-address statements.

The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table. We shall first illustrate how to store the return address in an activation record on a procedure call and how to return control to it after the procedure call. For convenience, we assume the first location in the activation holds the return address.

Let us first consider the code needed to implement the simplest case, **static allocation**. Here, a **call** *callee* statement in the intermediate code can be implemented by a sequence of two target-machine instructions:

```
ST callee.staticArea, #here + 20
BR callee.codeArea
```

The **ST** instruction saves the return address at the beginning of the activation record for *callee,* and the **BR** transfers control to the target code for the called procedure *callee.* The attribute before *callee.staticArea* is a constant that gives the address of the beginning of the activation record for *callee,* and the attribute *callee.codeArea* is a constant referring to the address of the first instruction of the called procedure *callee* in the *Code* area of the run-time memory.

The operand *#here+* 20 in the **ST** instruction is the literal return address; it is the address of the instruction following the **BR** instruction. We assume that *#here* is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of 5 words or 20 bytes.

Suppose we have the following three-address code:

```
                    // code for c
action₁
call p
action₂
halt
                    // code for p
action₃
return
```

The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is **HALT,** which returns control to the operating system. A **return** *callee* statement can be implemented by a simple jump instruction

BR **callee.staticArea*

which transfers control to the address saved at the beginning of the activation record for *callee.*

5.8 Basic Blocks and Flow Graphs

The representation is constructed as follows:

1. Partition the intermediate code into ***basic blocks,*** which are maximal sequences of consecutive three-address instructions with the properties that
   a) The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block.
   b) Control will leave the block without halting or branching, except possibly at the last instruction in the block.
2. The basic blocks become the nodes of a ***flow graph,*** whose edges indicate which blocks can follow which other blocks.

5.8.1 Basic Blocks

Our first job is to partition a sequence of three-address instructions into basic blocks. We begin a new basic block with the first instruction and keep adding instructions until we meet either a jump, a conditional jump, or a label on the following instruction. In the absence of jumps and labels, control proceeds sequentially from one instruction to the next. This idea is formalized in the following algorithm.

Algorithm 5.8.1: Partitioning three-address instructions into basic blocks.
INPUT: A sequence of three-address instructions.
OUTPUT: A list of the basic blocks for that sequence in which each instruction is assigned to exactly one basic block.
METHOD: First, we determine those instructions in the intermediate code that are ***leaders,*** that is, the first instructions in some basic block. The instruction just past the end of the intermediate program is not included as a leader. The rules for finding leaders are:

1. The first three-address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows a conditional or unconditional jump is a leader.

Then, for each leader, its basic block consists of itself and all instructions up to but not including the next leader or the end of the intermediate program.

```
1) i = 1
2) j = 1
3) t1 = 10 * i
4) t2 = t1 + J
5) t3 = 8 * t2
6) t4 = t3 - 88
7) a [t4] = 0.0
8) j = j + 1
9) i f j <= 10 goto (3)
10) i = i + 1
11) i f i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) i f i <= 10 goto
```

Figure 5.8.1: Intermediate code to set a 10 x 10 matrix to an identity matrix

The intermediate code in Fig. 5.8.1 turns a 10 x 10 matrix a into an identity matrix. Although it is not important where this code comes from, it might be the translation of the pseudocode in Fig. 5.8.2. In generating the intermediate code, we have assumed that the real-valued array elements take 8 bytes each, and that the matrix a is stored in row-major form.

```
for i from 1 to 10 do
        for j from 1 to 10 do
                a[i,j] = 0.0;
for i from 1 to 10 do
                a[i, i] = 1.0;
```

Figure 5.8.2: Source code for Fig. 5.8.1

First, instruction 1 is a leader by rule (1) of Algorithm 5.8.1. To find the other leaders, we first need to find the jumps. In this example, there are three jumps, all conditional, at instructions 9, 11, and 17. By rule (2), the targets of these jumps are leaders; they are instructions 3, 2, and 13, respectively. Then, by rule (3), each instruction following a jump is a leader; those are instructions

10 and 12. Note that no instruction follows 17 in this code, but if there were code following, the 18th instruction would also be a leader.

We conclude that the leaders are instructions 1, 2, 3, 10, 12, and 13. The basic block of eachleader contains all the instructions from itself until just before the next leader. Thus, the basic block of 1 is just 1, for leader 2 the block is just 2. Leader 3, however, has a basic block consisting of instructions 3 through 9, inclusive. Instruction 10's block is 10 and 11; instruction 12's block is just 12, and instruction 13's block is 13 through 17.

5.8.2 Flow Graphs

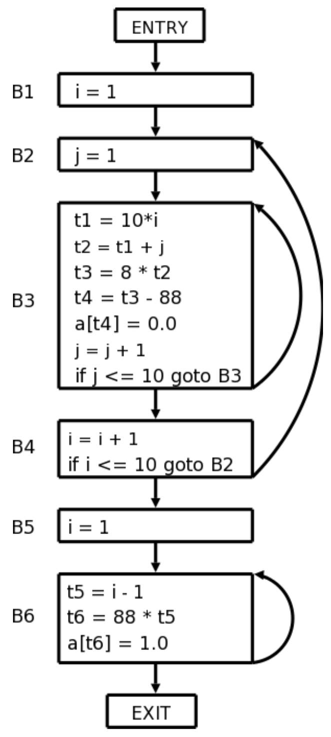
Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph. The nodes of the flow graph are the basic blocks. There is an edge from block *B* to block *C* if and only if it is possible for the first instruction in block *C* to immediately follow the last instruction in block *B.* There are two ways that such an edge could be justified:

- There is a conditional or unconditional jump from the end of *B* to the beginning of *C.*
- *C* immediately follows *B* in the original order of the three-address instructions, and *B* does not end in an unconditional jump.

We say that *B* is a *predecessor* of *C,* and *C* is a *successor* of *B.*

Often we add two nodes, called the ***entry* and *exit*** *that* do not correspond to executable intermediate instructions. There is an edge from the entry to the first executable node of the flow graph, that is, to the basic block that comes from the first instruction of the intermediate code. There is an edge to the exit from any basic block that contains an instruction that could be the last executed instruction of the program. If the final instruction of the program is not an unconditional jump, then the block containing the final instruction of the program is one predecessor of the exit, but so is any basic block that has a jump to code that is not part of the program.

The set of basic blocks constructed in Example 8.6 yields the flow graph of Fig. 8.9. The entry points to basic block B1, since B1 contains the first instruction of the program. The only successor of B1 is B2, because B1 does not end in an unconditional jump, and the leader of B2 immediately follows the end of B1.



Block B3 has two successors. One is itself, because the leader of B3, instruction 3, is the target of the conditional jump at theend of B3, instruction 9. The other successor is B4, because control can fall through the conditional jump at the end of B3 and next enter the leader of B4.

Only B6 points to the exit of the flow graph, since the only way to get to code that follows the program from which we constructed the flow graph is to fall through the conditional jump that ends B6.

UNIT – VI

Machine Independent Optimization. The principle sources of Optimization peephole Optimization, Introduction to Date flow Analysis.

MACHINE INDEPENDENT CODE OPTIMIZATION:

The optimization which is done on an intermediate code is called machine independent code optimization because intermediate code does not depend on any target machine.

6.1 Semantic Preserving Transformation:

There are number of optimization techniques that a compiler can use to improve the performance of the program without changing its output. The optimization technique should not influence the semantics of the program. Common-sub expression elimination, copy propagation, dead-code elimination, and constant folding are common examples of such semantic preserving transformations

6.1.1 Common Sub-Expression Elimination:

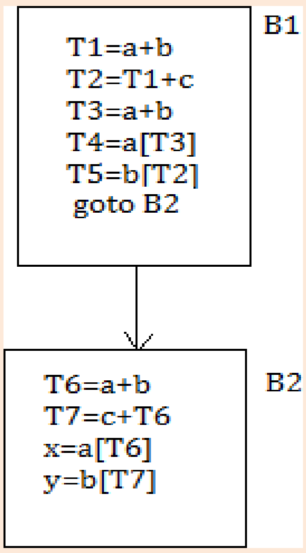
The code can be improved by eliminating common sub expressions from the code. An expression whose value is previously computed and the values of variables in the expression are not changed, since its computation can be avoided to by using the earlier computed value.

The below example shows the optimized code after eliminating the common sub expressions.

```
a : = b + c                          a : = b + c
b : = a – d      ──────────►         b : = a - d
c : = b + c                          c : = b + c
d : = a – d                          d : = b
```

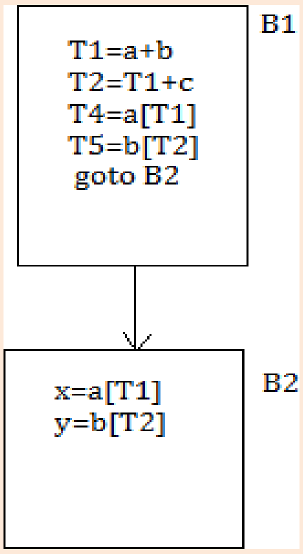
Fig (a): Before Elimination **Fig (b)** After Common sub exp elimination

In above code d=a-d can be replaced by d=b. This technique optimizes the intermediate code. If we eliminate common sub-expression in one block of code then it is called local common sub-expression elimination. If we eliminate common sub-expressions among multiple blocks then it is called Global Common sub-expression elimination. The above example shows local common sub-expression elimination. In the below example this optimization is performed first locally and then globally.



```
T1=a+b
T2=T1+c
T3=a+b
T4=a[T3]
T5=b[T2]
goto B2
```

```
T6=a+b
T7=c+T6
x=a[T6]
y=b[T7]
```



```
T1=a+b
T2=T1+c
T4=a[T1]
T5=b[T2]
goto B2
```

```
x=a[T1]
y=b[T2]
```

Fig a) Before Global Common sub expression elimination **Fig b) After Elimination**

Example 2: Quick Sort Example

In this example a fragment of a sorting program called *quicksort is used* to illustrate several important code-improving transformations.

void **quicksort**(int m, int n)
/ *recursively sorts a[m] through a[n] */
{

```
            int i,j ;
            int v,x;
            if (n <= m) return;
            / * fragment begins here */
            i = m - 1 ; j = n ; v = a[n] ;
            while (1) {
                    do i = i + 1 ; while ( a[i] < v );
                    do j = j - 1 ; while ( a[j] > v );
                    i f (i >= j) break;
                    x = a[i]; a[i] = a[j]; a[j] = x; /* swap a [i],a [j] */
                    }
            x = a [i]; a[i] = a[n]; a[n] = x; /* swap a[i],a[n] */
            /* fragment ends here */
            quicksort(m,j) ; quicksort(i+1,n );
    }
```

Figure 6.1: C code for quicksort

Intermediate code for the marked fragment of the program is shown in Fig. 6.2.

```
(1)     i = m-1                  (16)    t7 = 4*i
(2)     j = n                    (17)    t8 = 4*j
(3)     t1 = 4*n                 (18)    t9 = a[t8]
(4)     v = a[t1]                (19)    a[17] = t9
(5)     i = i+1                  (20)    t10 = 4*j
(6)     t2 = 4*i                 (21)    a[t10] = x
(7)     t3 = a[t2]               (22)    goto (5)
(8)     if t3<v goto (5)         (23)    t11 = 48i
(9)     j = j-1                  (24)    x = a[t11]
(10)     t4 = 4*i                (25)    t12 = 4*i
(11)     t5 = a[t4]              (26)    t13 = 4*n
(12)     if t5>v goto (9)        (27)    t14 = a[t13]
(13)     if i>=j goto (23)       (28)    a[t12] = t14
(14)     t6 = 4*i                (29)    t15 = 4*n
(15)     x=a[t6]                 (30)    a[t15] = x
```

Figure 6.2: Three-address code for fragment in Fig. 6.1

In this example we assume that integers occupy four bytes. The assignment x = a[i] is translated into the two three-address statements

```
 t6 = 4*i
x = a[t6]
```

as shown in steps (14) and (15).
Similarly, a[j] = x becomes

```
t10 = 4*j
a[t10] = x
```

in steps (20) and (21).

Figure 6.3 is the flow graph for the program in Fig. 6.2. Block *B1* is the entry node. All conditional and unconditional jumps to statements in Fig. 6.2 have been replaced in Fig. 6.3 by jumps to the block of which the statements are leaders. In Fig. 6.3, there are three loops. Blocks *B2* and B3 are loops by themselves. Blocks *B2, B3, B4,* and *B5* together form a loop,
with *B2* the only entry point.

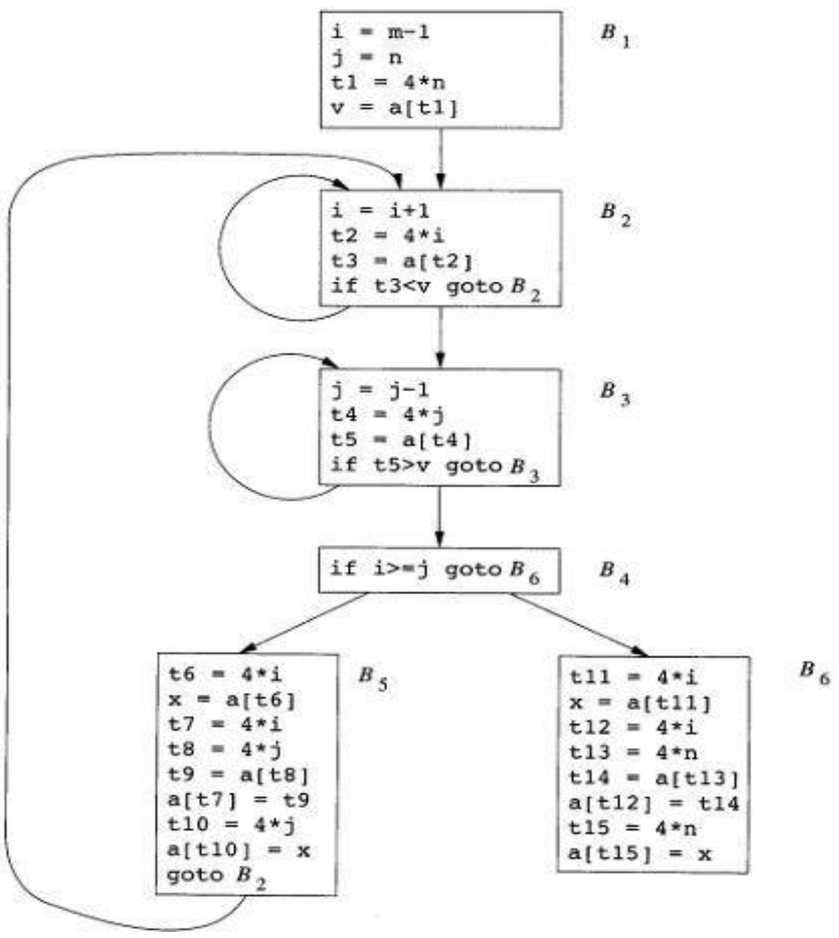


Figure 6.3: Flow graph for the quicksort fragment

*t6 = 4*i*	*t6 = 4*i*
x = a[t6]	*x = a[t6]*
*t7 = 4*i*	*t8 = 4*j*
*t8 = 4*j*	*t9 = a[t8]*
t9 = a[t8]	*a[t6] = t9*
a[t7] = t9	*a[t8] = x*
*t10 = 4*j*	*goto B2*
a[t10] = x	
goto B2	
(a) Before.	***(b) After.***

Figure 6.4: Local common-subexpression elimination

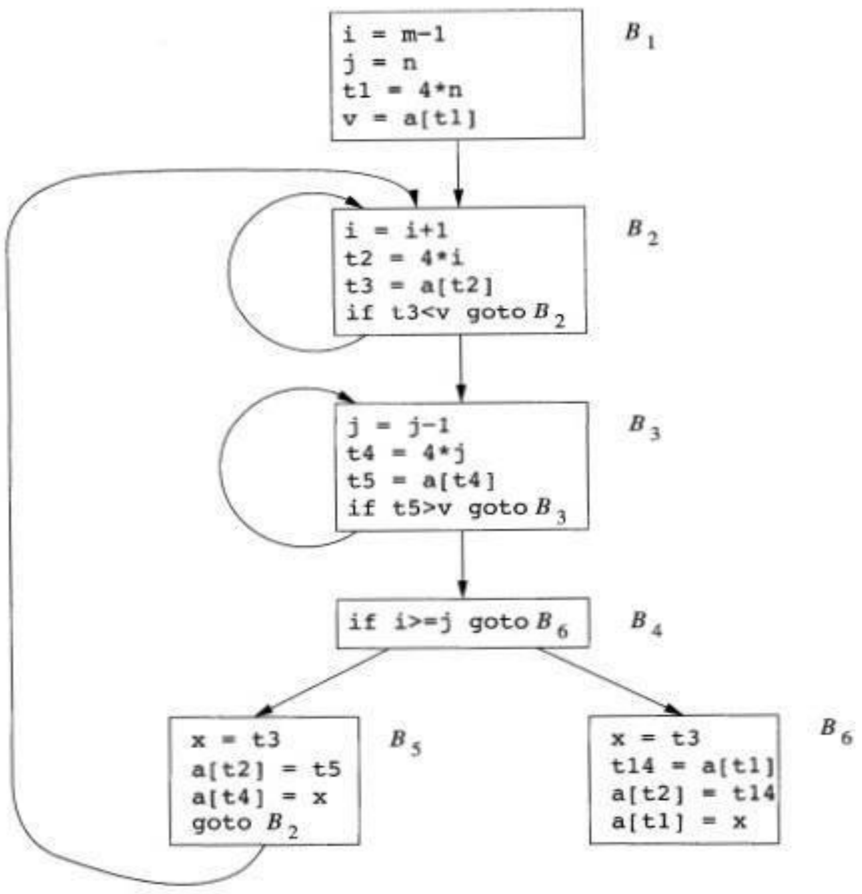


Figure 6.5: B5 and B6 after common-subexpression elimination

6.1.2 Constant Folding:

This is an optimization technique that evaluates constant expressions at compile time and replaces such expressions by their computed values.

In this technique expressions with constant operands can be evaluated at compile time, thus improving the run-time performance and reducing the code size by avoiding evaluation at compile-time.

Example:

In the code fragment below, the expression (3 + 5) can be evaluated at compile time and replaced with the constant 8.

```
int f (void)
{
 return 3 + 5;
}
```

Below is the code fragment after constant folding.

```
int f (void)
{
 return 8;
}
```

Notes:

Constant folding is a relatively easy optimization.

Programmers generally do not write expressions such as (3 + 5) directly, but these expressions are relatively common after macro expansion and other optimizations such as constant propagation.

6.1.3 Copy Propagation:

It is the process of replacing the occurrences of targets of direct assignments with their values. A direct assignment is an instruction of the form x = y, which simply assigns the value of y to x.
From the following code:

```
y = x
z = 3 + y
```

Copy propagation would yield:

```
z = 3 + x
```

Example 2: Quick Sort Example
The idea behind the copy-propagation transformation is to use *v* for u, wherever possible after the copy statement u = v. For example, the assignment x = t3 in block *B5* of Fig. 6.5 is a copy. Copy propagation applied to *B5* yields the code in Fig. 6.7. This change may not appear to be an improvement, but, it gives us the opportunity to eliminate the assignment to *x.*

```
x = t3
a[t2] = t5
a[t4] = t3
goto B2
```

Figure 6.7: Basic block B5 after copy propagation

6.1.4 Dead Code Elimination:

Dead code elimination (also known as **dead code removal**, **dead code stripping**, or **dead code strip**) is an optimization technique to remove the code which does not affect the program results. Removing such code has two benefits: it shrinks the program size and it allows the running program to avoid executing irrelevant operations, which reduces its running time. *Dead code* includes code that can never be executed (*unreachable code*), and code that only affects *dead variables*, that is, variables that are irrelevant to the program.
Consider the following example written in C.

```c
int foo(void)
{
  int a = 24;
  int b = 25; /* Assignment to dead variable */
  int c;
  c = a << 2;
  return c;
  b = 24; /* Unreachable code */
  return 0;
}
```

Simple analysis of the uses of values would show that the value of b after the first assignment is not used inside foo. Furthermore, b is declared as a local variable inside foo, so its value cannot be used outside foo. Thus, the variable b is *dead* and an optimizer can reclaim its storage space and eliminate its initialization.
Furthermore, because the first return statement is executed unconditionally, no feasible execution path reaches the second assignment to b. Thus, the assignment is *unreachable* and can be removed.

Example 2:
In the example below, the value assigned to i is never used, and the dead store can be eliminated. The first assignment to global is dead, and the third assignment to global is unreachable; both can be eliminated.

```
int global;
void f ()
{
 int i;
 i = 1;        /* dead store */
 global = 1;   /* dead store */
 global = 2;
 return;
 global = 3;   /* unreachable */
}
```

Below is the code fragment after dead code elimination.

```
int global;
void f ()
{
 global = 2;
 return;
}
```

Example 3: Quick Sort Example

One advantage of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to *x* and transforms the code in Fig 6.7 into

a[t2] = t5
a[t4] = t3
goto B2

This code is a further improvement of block *B5* in Fig. 6.5.

6.1.5 Strength Reduction:

Strength reduction is an optimization technique in which expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. By doing this the execution speed can be increased.

Example 1: Consider the following code

```
for(i=1;i<=5;i++)
{
x=4*i;
}
```

The instruction ***x=4*i*** in the loop can be replaced by equivalent additions instruction as ***x=x+4*;**

Code after strength reduction is shown below.

```
x=0;
for(i=1;i<=5;i++)
x=x+4;
```

Example 2: Quick Sort

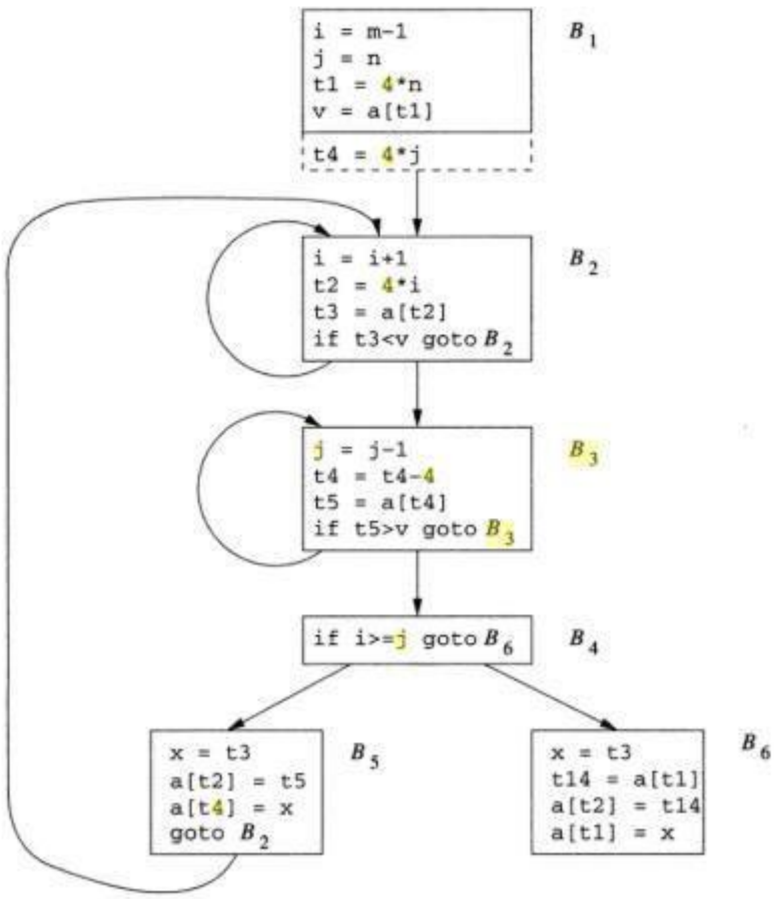


*Figure 6.8: Strength reduction applied to 4 * j in block B3*

6.2 Loop Optimization:

Loops are a very important place for optimizations, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
There are three techniques:

1. Code Motion
2. Elimination of induction variables
3. Strength reduction

1. Code Motion:

Code motion reduces the number of instructions in a loop by moving instructions outside a loop. It moves loop invariant computations i.e, those instructions or expressions that result in the same value independent of the number of times a loop is executed and places them at the beginning of the loop. The relocated expressions become an entry for the loop.

Example:

```
While(X!=n-2)
{
X=X+2;
}
```

here the expression n-2 is a loop invariant computation i.e, the value evaluated by this expression is independent of the number of times the while loop is executed. In other words the value of n remains unchanged. The code relocation places the expressions n-2 before the while loop begins as shown below.

```
M=n-2;
While(X!=M)
{
```

```
X=X+2;
}
```

2. **Elimination of Induction Variables:**
   An induction variable is a loop control variable or any other variable that depends on the induction variable in some fixed way. It can also be defined as variable which is incremented or decremented by a fixed number in a loop each time the loop is executed. If there are two or more induction variables in a loop then by the induction variable elimination process all can be eliminated except one.
   Example:
   ```
   void fun(void)
   {
    int i,j,k;
   for(i=0,j=0,k=0;i<10;i++)
   a[j++]=b[k++];
   return;
   }
   ```
   In the above code there are three induction variables i,j and k which take on the values 1,2,3…10 each time through the beginning of the loop. Suppose that the values of the variables j and k are not used after the end of the loop then we can eliminate them from the function fun() by replacing them by variable i.
   After induction variable elimination, the above code becomes
   ```
   void fun(void)
   {
    int i,j,k;
   for(i=0;i<10;i++)
   a[i]=b[i];
   return;
   }
   ```
   Thus induction variable elimination reduces the code and improves the run time performance.
3. **Strength Reduction:**
   Strength reduction is an optimization technique in which expensive operations are replaced with equivalent but less expensive operations. The classic example of strength reduction converts "strong" multiplications inside a loop into "weaker" additions – something that frequently occurs in array addressing. By doing this the execution speed can be increased. For example consider the code
   ```
   for(i=1;i<=5;i++)
   {
   x=4*i;
   }
   ```
   The instruction **x=4*i** in the loop can be replaced by equivalent additions instruction as ***x=x+4***;

6.3 Instruction Scheduling:

In this technique the instructions are rearranged to generate an efficient code using minimum number of registers. By changing the order in which computations are done we can obtain the object code with minimum cost. The technique of code optimization done by rearrangement of some sequence of instructions is called Instruction Scheduling.

Example:

```
TAC
T1= a+b
T2=c-d
T3=e+T2
T4=T1+T3
```

```
TARGET CODE
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, T1
MOV e, R0
ADD R0, R1
MOV T1, R0
ADD R1, R0
MOV R0, T4
```

Fig (a): Before Instruction Scheduling.

```
TAC
T2=c-d
T3=e+T2
T1= a+b
T4=T1+T3
```

```
TARGET CODE
MOV c, R0
SUB d, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, T4
```

Fig (b): After Instruction Scheduling

In above diagram TAC with its equivalent Target code is shown.
Now if we rearrange the instructions of TAC then the Target code will get reduced. In the first case the assembly code contains 10 lines. After rearranging the TAC then the assembly code contains 8 Lines. So by rearranging some sequence of instructions we can generate an efficient code using minimum number of registers.

6.4 Interprocedural Optimization:
It is a kind of code optimization in which collection of optimization techniques are used to improve the performance of the program. IPO reduces or eliminates duplicate calculations, inefficient use of memory and simplify the loops. IPO may reorder instructions for better memory utilization. IPO also checks the branches or code that never get executed and removes them from the program (dead code elimination). In this technique first we apply optimization techniques on each block i.e local optimization and then we apply optimization techniques on all the blocks of program i.e global optimization.
The example for this techniques is shown in the figure below.

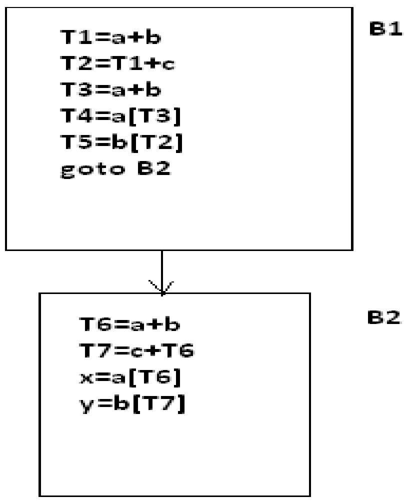


Fig a: Before local Optimization

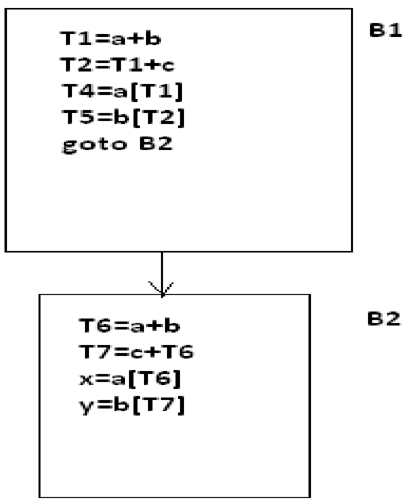


fig b After local Optimization

First we apply local optimization on B1 and B2 independently. As shown in above figure. B1 contains common sub-expression and copy propagations they are eliminated. After B1, optimization is performed independently on B2.

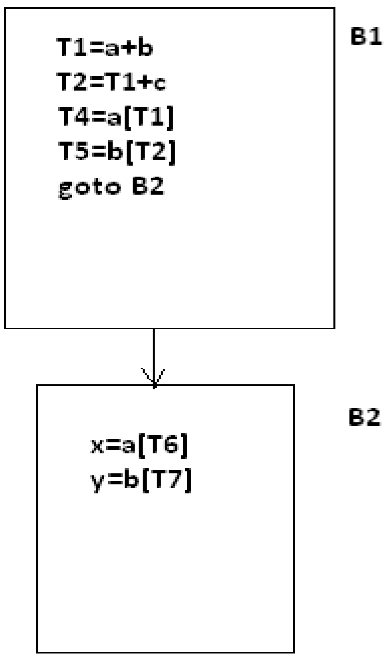


Fig. After Global Optimization

After performing local optimization, global optimization is performed on B1 and B2. In this case B2 contains T6=a+b and T7=c+T6 whose values are already calculated in B1. So instead of recomputing these expressions we can use T1 and T2 directly in B2. After global Optimization the modified code is shown above figure.

6.5 Peephole Optimization

A simple but effective technique for locally improving the target code is ***peephole optimization***, which is done by examining a sliding window of target instructions (called the *peephole)* and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Peephole optimization can also be applied directly after intermediate code generation to improve theintermediate representation.

The following peephole optimization techniques may be applied to improve the performance of the target program:

- Redundant-instruction elimination
- Elimination of Unreachable Code.
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

i) Eliminating Redundant Loads and Stores

If we see the instruction sequence

LD a, R0
ST R0, a

in a target program, we can delete the store instruction. Note that if the store instruction has a label, we could not be sure that the first instruction is always executed before the second, so we could not remove the store instruction. The two instructions have to be in the same basic block for this transformation to be safe.

ii) Eliminating Unreachable Code

Another opportunity for peephole optimization is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed.

Goto L2
Print Debug Information
L2:

In above example Print Statement can be eliminated.

iii) Flow-of-Control Optimizations

Simple intermediate code-generation algorithms frequently produce jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations.

We can replace the sequence

goto L1
...
Ll: goto L2

by the sequence

goto L2
...
Ll: goto L2

If there are now no jumps to L1, then it may be possible to eliminate the statement **L1: goto L2** provided it is preceded by an unconditional jump.

Similarly, the sequence

If a< b goto L1

L1: goto L2

can be replaced by the sequence

If a< b goto L2

L1: goto L2

iv) Algebraic Simplification and Reduction in Strength

The algebraic identities can also be used by a peephole optimizer to eliminate three-address statements such as X = X + 0 and X = X * 1 in the peephole. Similarly, reduction-in-strength transformations can be applied in the peephole to replace expensive operations by equivalent cheaper ones. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x^2 can be implemented by using x*x. Similarly x*x can be implemented by using x+x.

v) Use of Machine Idioms

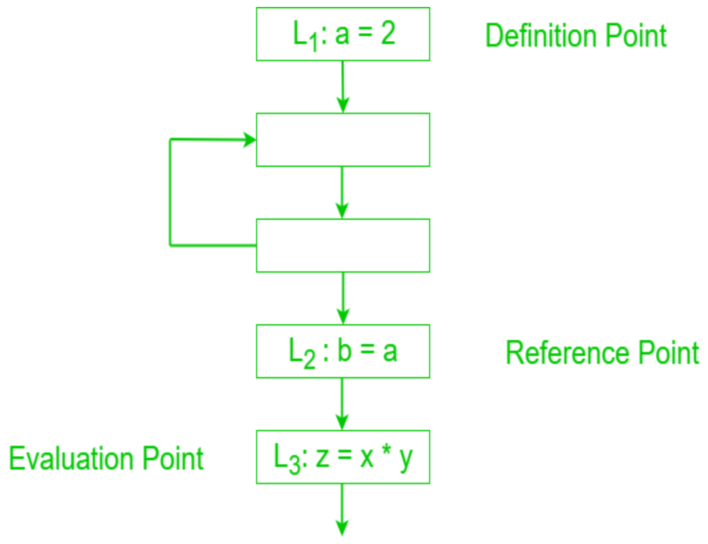
The target machine may have hardware instructions to implement certain specific operations efficiently. Detecting situations that permit the use of these instructions can reduce execution time significantly. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of the modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like x = x + 1.

Data Flow Analysis:

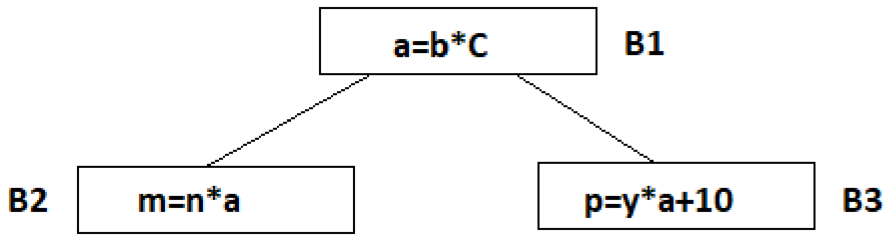
Data flow analysis is a technique of gathering all the information about the program and distributing this information to all the blocks of a flow graph.

It determines the information regarding the definition and use of data in program. This technique is used for optimization.

- **Definition Point:** A point X in a program which contains definition or where definition is specified.
- **Reference Point**: A point X in a program where data item is referred.
- **Evaluation Point:** A point X in a program where an expression is evaluated.

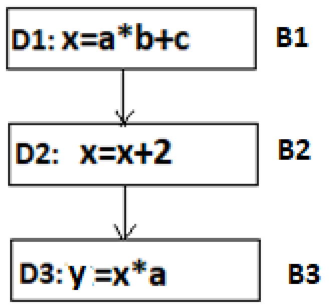


Available Expression: An expression is said to be available at program point X if along the path it reaches X. Also an expression is said to be available if none of its operands gets modified before its use.



In above figure expression b*C is available in Blocks B2 and B3. It is used to eliminate common sub expressions.

Reaching Definition: A definition D reaches point X, if there is a path from the point immediately following D to X such that D is not killed or not redefined along that path.



D1 is a reaching definition for B2 but not for B3 because it is killed by D2.

Live Variable

A variable **v** is live at point **p** if the value of v is used along some path in the flow graph starting at p. Otherwise, the variable is dead.

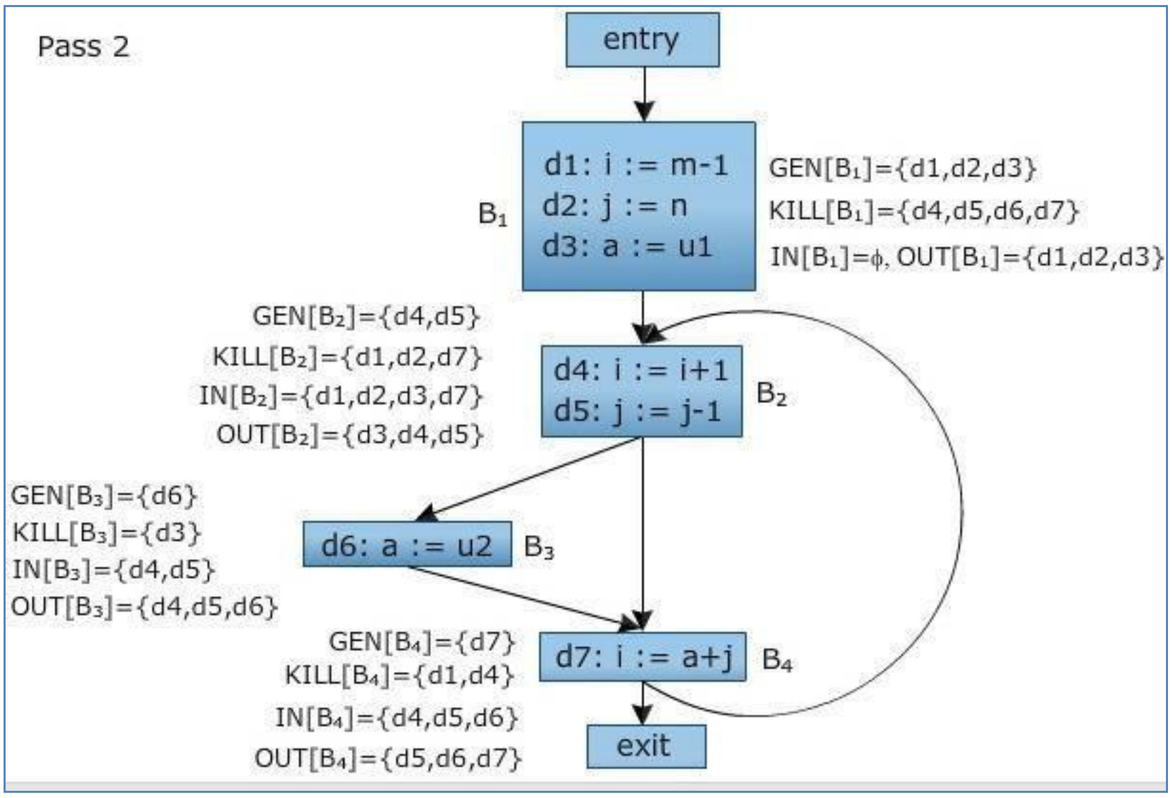
Data Flow Analysis Equation:

OUT[B]= GEN[B] U { IN[B] – KILL[B] }

IN[B]= If some definition reaches B1 entry then IN[B1] is initialised to that set.

GEN[B]= set of all definition defined inside B and that are visible after that block.

KILL[B]=Union of all definitions in all the basic blocks of flow graph that are killed by individual statements in B.



Frequently Asked Questions

1. What is Optimization? Explain various semantic preserving transformations with example.
2. Explain about Loop Optimization with suitable examples.
3. Discuss the importance of Instruction scheduling in Optimization.
4. Explain about various Peephole Optimization Techniques with suitable examples
5. Discuss the importance of Data Flow Analysis with example.