UNIT – I

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 buffering, specification of tokens. Recognitions of tokens the lexical analyzer generator lexical

UNIT -1

TRANSLATOR

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of HLL specification would be detected and reported to the programmers. Important role of translator are:

1 Translating the HLL program input into an equivalent machine language program.
2 Providing diagnostic messages wherever the programmer violates specification of the HLL.

A translator is a program that takes as input a program written in one language and produces as output a program in another language. Beside program translation, the translator performs another very important role, the error-detection. Any violation of HLL specification would be detected and reported to the programmers. Important role of translator are:

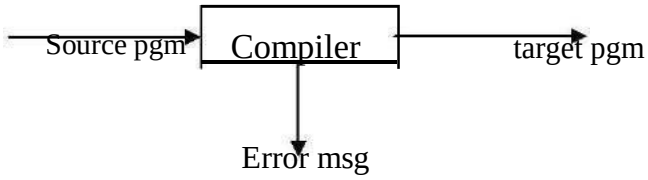
1 Translating the hll program input into an equivalent ml program.
2 Providing diagnostic messages wherever the programmer violates specification of the hll.

TYPE OF TRANSLATORS:-

a. **Compiler**
b. **Interpreter**
c. **Preprocessor**

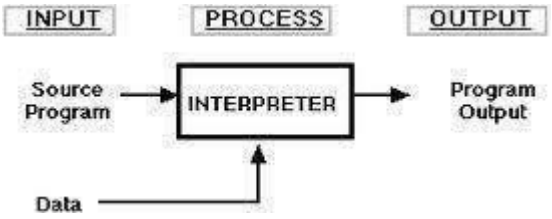
Compiler

Compiler is a translator program that translates a program written in (HLL) the source program and translate it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



Executing a program written n HLL programming language is basically of two parts. The source program must first be compiled and translated into a object program. Then the resulting object program is loaded into a memory executed.

Interpreter*:* An interpreter is a program that appears to execute a source program as if it were machine language.



Languages such as BASIC, SNOBOL, LISP can be translated using interpreters. JAVA also uses interpreter. The process of interpretation can be carried out in following phases.

1. Lexical analysis
2. Synatx analysis
3. Semantic analysis
4. Direct Execution

Advantages:

Modification of user program can be easily made and implemented as execution proceeds.

5. Type of object that denotes a various may change dynamically.

Debugging a program and finding errors is simplified task for a program used for interpretation.

The interpreter for the language makes it machine independent.

Disadvantages:

The execution of the program is *slower*.

Memory consumption is more.

OVERVIEW OF LANGUAGE PROCESSING SYSTEM

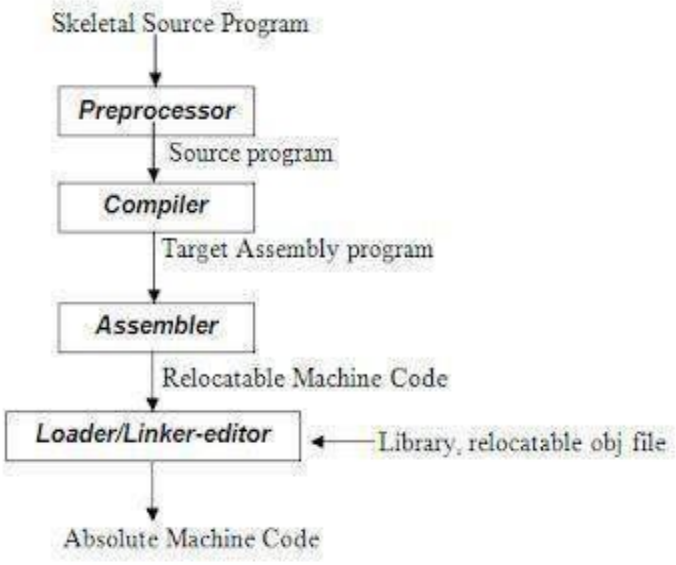


Fig 1.1 Language –processing System

Preprocessor

A preprocessor produce input to compilers. They may perform the following functions.

1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
4. *Language Extensions:* These preprocessor attempts to add capabilities to the language by certain amounts to build-in macro

Assembler*:* programmers found it difficult to write or read programs in machine language. They begin to use a mnemonic (symbols) for each machine instruction, which they would subsequently translate into machine language. Such a mnemonic machine language is now called an assembly language. Programs known as assembler were written to automate the translation of assembly language in to machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

Loader and Link-editor:

Once the assembler procedures an object program, that program must be placed into memory and executed. The assembler could place the object program directly in memory and transfer control to it, thereby causing the machine language program to be execute. This would waste core by leaving the assembler in memory while the user‟s program was being executed. Also the programmer would have to retranslate his program with each execution, thus wasting translation time. To overcome this problems of wasted translation time and memory. System programmers developed another component called loader

"A loader is a program that places programs into memory and prepares them for execution." It would be more efficient if subroutines could be translated into object form the loader could"relocate" directly behind the user‟s program. The task of adjusting programs othey may be placed in arbitrary core locations is called relocation.

STRUCTURE OF A COMPILER

Phases of a compiler: A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)
b. Synthesis(Machine Dependent/Language independent)

Compilation process is partitioned into no-of-sub processes called p**hases'**.

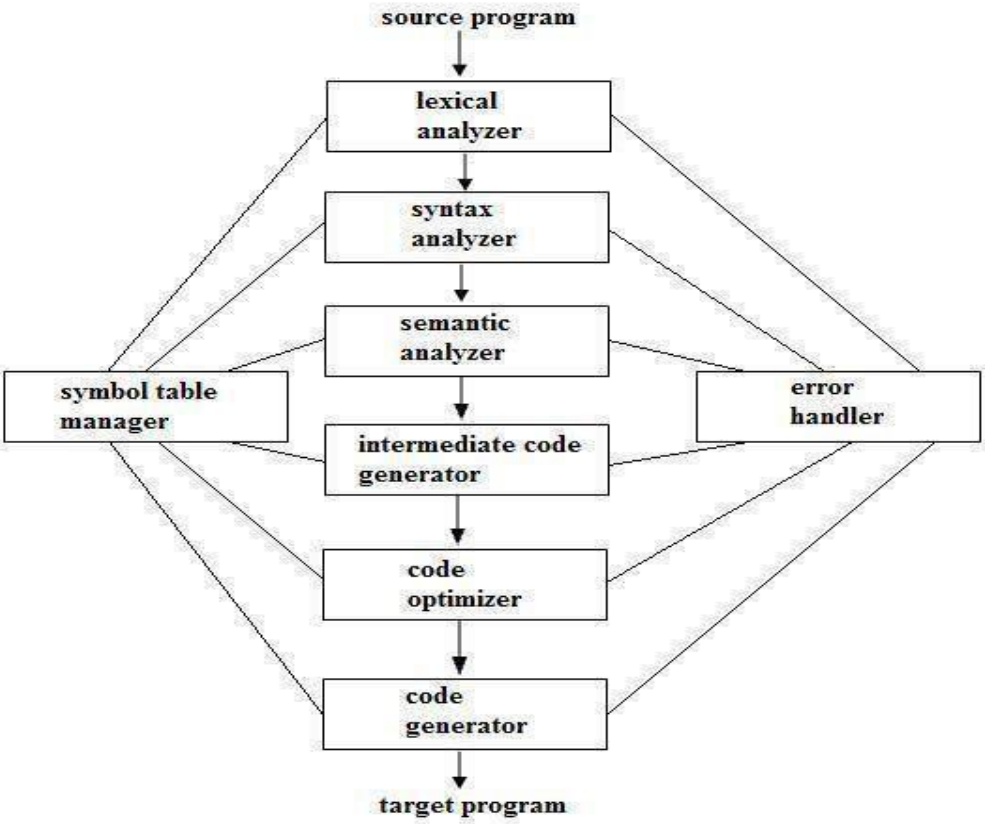


Fig 1.5 Phases of a compiler

Lexical Analysis:-

LA or Scanners reads the source program one character at a time, carving the source program into a sequence of automic units called **tokens.**

Syntax Analysis:-

The second stage of translation is called Syntax analysis or parsing. In this phase expressions, statements, declarations etc… are identified by using the results of lexical analysis. Syntax analysis is aided by using techniques based on formal grammar of the programming language.

Intermediate Code Generations:-

An intermediate representation of the final machine language code is produced. This phase bridges the analysis and synthesis phases of translation.

Code Optimization :-

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space.

Code Generation:-

The last phase of translation is code generation. A number of optimizations to **reduce the length of machine language program** are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:-

This is the portion to **keep the names** used by the program and records essential information about each. The data structure used to record this information called a „Symbol Table".

Error Handlers:-

It is invoked when a flaw error in the source program is detected.

The output of **LA** is a stream of tokens, which is passed to the next phase, the syntax analyzer or parser. The SA groups the tokens together into syntactic structure called as **expression.** Expression may further be combined to form statements. The syntactic structure can be regarded as a tree whose leaves are the token called as parse trees.

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

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

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

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

1, divide A by B and then multiply by C or

2, multiply B by C and then use the result to divide A.

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

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are possible. One common style uses instruction with one operator and a small number of operands.

The output of the syntax analyzer is some representation of a parse tree. the intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization

This is optional phase described to improve the intermediate code so that the output runs faster and takes less space. Its output is another intermediate code program that does the some job as the original, but in a way that saves time and / or spaces.

1, Local Optimization:-

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

If **A** > **B** goto **L2**

Goto **L3**

L2 :

This can be replaced by a single statement

If **A < B** goto **L3**

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

A := B + C + D
E := B + C + F

Might be evaluated as

T1 := B + C

A := T1 + D
E := T1 + F

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

2, Loop Optimization:-

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

Code Generator :-

Code Generator produces the object code by deciding on the memory locations for data, selecting code to access each datum and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would attempt to utilize registers as efficiently as possible.

Table Management OR Book-keeping :-

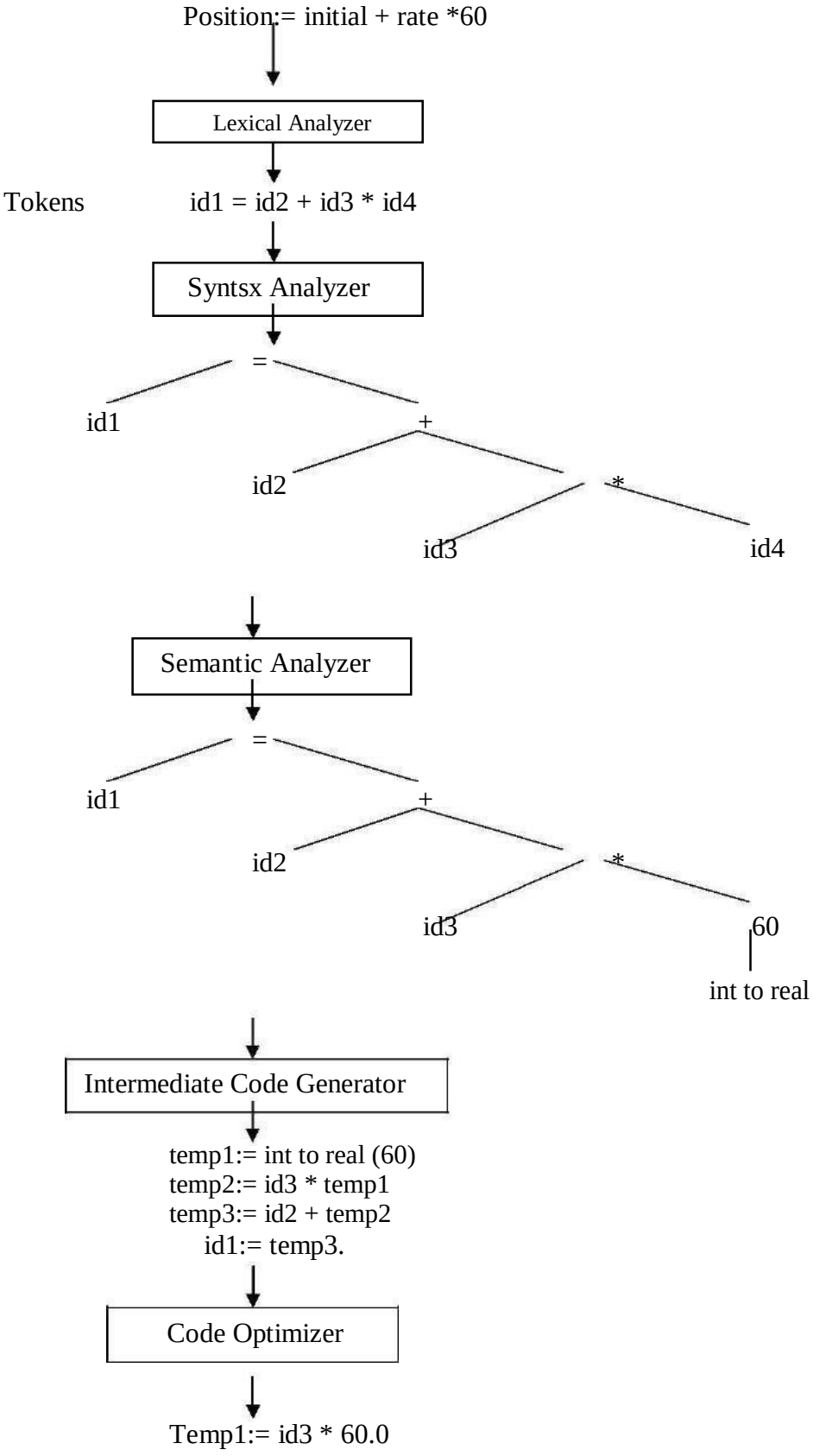
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

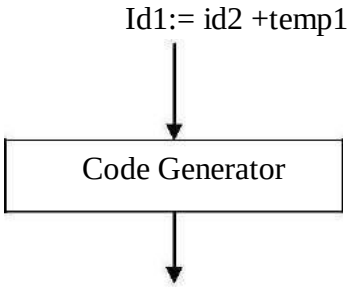
Error Handing :-

One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred. Errors may occur in all or the phases of a compiler.

Whenever a phase of the compiler discovers an error, it must report the error to the error handler, which issues an appropriate diagnostic msg. Both of the table-management and error-Handling routines interact with all phases of the compiler.

Example:


Position:= initial + rate *60
Lexical Analyzer
Tokens
id1 = id2 + id3 * id4
Syntsx Analyzer
=
id1
+
id2
*
id3
id4
Semantic Analyzer
=
id1
+
id2
*
id3
60
int to real
Intermediate Code Generator
temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
Code Optimizer
Temp1:= id3 * 60.0




```
MOVF id3, r2
MULF *60.0, r2
MOVF   id2, r2
ADDF   r2, r1
MOVF   r1, id1
```

Evolution of Programming languages

The **history of programming languages** spans from documentation of early mechanical computers to modern tools for software development. Early programming languages were highly specialized, relying on mathematical notation

The move to Higher Level Languages

The first step towards more people friendly programming languages was the development of mnemonic assembly languages in the early 1950's.The instructions in assembly languages were just mnemonic representations of machine instructions.
A major step towards higher level languages was made in the later half of the 1950's with the development of FORTRAN for scientific computation, Cobol for business data Processing and Lisp for symbolic computation.

In the following decades many more languages were created with innovative features to help make programming easier, more natural, and more robust.

Languages can also be classified in variety of ways.

Classification by Generation: Ist generation are the machine languages, 2nd generation are the assembly languages, 3rd generation are the higher level languages like Fortran,cobol, Lisp,C etc.4th generation are the languages designed for specific application like NOMAD,SQL,POST The term fifth generation language has been applied to logic and the constraint based language like prolog and OPS5.

Classification by the use: imperative languages in which your program specifies How computation is to be done the declarative for languages in which your program specifies what computation is to be done.

Examples:

Imperative languages: C,C++,C#,Java.
Declarative languages: ML, Haskell, Prolog

Object oriented language is one that supports Object oriented programming, a Programming style in which a program consists of a collection of objects that interact with one another.

Examples: Simula 67, small talk, C ++, Java,Ruby

Scripting languages are interpreted languages with high level operators designed for "gluing together" computations These computations originally called Scripts

Example: JavaScript, Perl, PHP, python, Ruby, TCL

The Science of building a Compiler

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. Compiler writers thus have influence over not just the compilers they create, but all the programs that their compilers compile. This leverage makes writing compilers particularly rewarding; however, it also makes compiler development challenging.

Modelling in compiler design and implementation: The study of compilers is mainly a study of how we design the right mathematical models and choose the right algorithms.Some of most fundamental models are finite-state machines and regular expressions.These models are useful for de-scribing 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 such as the nesting of parentheses or control constructs. Similarly, trees are an important model for representing the structure of programs and their translation into object code.

The science of code optimization: The term "optimization" in compiler design refers to the attempts that a com-piler makes to produce code that is more efficient than the obvious code. In modern times, the optimization of code that a compiler performs has become both more important and more complex. It is more complex because processor architectures have become more complex, yielding more opportunities to improve the way code executes. It is more important because massively par-allel computers require substantial optimization, or their performance suffers by orders of magnitude.

Compiler optimizations must meet the following design objectives:

1. The optimization must be correct, that is, preserve the meaning of the compiled program,
2. The optimization must improve the performance of many programs,
3. The compilation time must be kept reasonable, and
4. The engineering effort required must be manageable.

Thus, in studying compilers, we learn not only how to build a compiler, but also the general methodology of solving complex and open-ended problems.

Applications of Compiler Technology

Compiler design impacts several other areas of computer science.

Implementation of high-level programming language: 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. higher-level programming languages are easier to program in, but are less efficient, that is, the target programs run more slowly. Programmers using a low-level language have more control over a computation and can, in principle, produce more efficient code.

Language features that have stimulated significant advances in compiler technology.

Practically all common programming languages, including C, Fortran and Cobol, support user-defined aggregate data types, such as arrays and structures, and high-level control flow, such as loops and procedure invocations. If we just take each high-level construct or data-access operation and translate it directly to machine code, the result would be very inefficient. A body of compiler optimizations, known as *data-flow optimizations*, has been developed to analyze the flow of data through the program and removes redundancies across these constructs. They are effective in generating code that resembles code written by a skilled programmer at a lower level.

Object orientation was first introduced in Simula in 1967, and has been incorporated in languages such as Smalltalk, C + + , C # , and Java. The key ideas behind object orientation are

1. Data abstraction and
2. Inheritance of properties,

Java has many features that make programming easier, many of which have been introduced previously in other languages. Compiler optimizations have been developed to reduce the overhead, for example, by eliminating unnecessary range checks and by allocating objects that are not accessible beyond a procedure on the stack instead of the heap. Effective algorithms also have been developed to minimize the overhead of garbage collection.

In dynamic optimization, it is important to minimize the compilation time as it is part of the execution overhead. A common technique used is to only compile and optimize those parts of the program that will be frequently executed.

Optimizations for Computer Architecture: 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 and at 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.

Design of New Computer Architectures: in modern computer architecture development, compilers are developed in the processor-design stage, and compiled code, running on simulators, is used to evaluate the proposed architectural features. One of the best known examples of how compilers influenced the design of computer architecture was the invention of the RISC (Reduced Instruction-Set Computer) architecture.

Compiler optimizations often can reduce these instructions to a small number of simpler operations by eliminating the redundancies across complex instructions. Thus, it is desirable to build simple instruction sets; compilers can use them effectively and the hardware is much easier to optimize. Most general-purpose processor architectures, including PowerPC, SPARC, MIPS, Alpha, and PA-RISC, are based on the RISC concept.

Specialized Architectures Over the last three decades, many architectural concepts have been proposed. They include data flow machines, vector machines, VLIW (Very Long Instruction Word) machines, SIMD (Single Instruction, Multiple Data) arrays of processors, systolic arrays, multiprocessors with shared memory, and multiprocessors with distributed memory. The development of each of these architectural concepts was accompanied by the research and development of corresponding compiler technology.

Program Translations: The following are some of the important applications of program-translation techniques.

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 by various computer companies to increase the availability of software for their machines.

Hardware Synthesis: Not only is most software written in high-level languages; even hardware de-signs are mostly described in high-level hardware description languages like Verilog and VHDL. Hardware designs are typically described at the register trans-fer level (RTL), where variables represent registers and expressions represent combinational logic.

Database Query Interpreters: Besides specifying software and hardware, languages are useful in many other applications. For example, query languages, especially SQL (Structured Query Language), are used to search databases. Database queries consist of predicates containing relational and boolean operators. They can be interpreted or com-piled into commands to search a database for records satisfying that predicate.

Programming Language Basics:

1 The Static/Dynamic Distinction

2 Environments and States

3 Static Scope and Block Structure

4 Explicit Access Control

5 Dynamic Scope

6 Parameter Passing Mechanisms

The Static/Dynamic Distinction: Among the most important issues that we face when designing a compiler for a language is what decisions can the compiler make about a program. If a language uses a policy that allows the compiler to decide an issue, then we say that the language uses a *static* policy or that the issue can be decided at *compile time.* On the other hand, a policy that only allows a decision to be made when we execute the program is said to be a *dynamic policy.* One issue is the scope of declarations. The *scope* of a declaration of *x* is the region of the program in which uses of *x* refer to this declaration. A language uses *static scope* or *lexical scope* if it is possible to determine the scope of a declaration by looking only at the program. Otherwise, the language uses *dynamic scope.* With dynamic scope, as the program runs, the same use of *x* could refer to any of several different declarations of *x.*

Environments and States:

The *environment* is a mapping from names to locations in the store. Since variables refer to locations, 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.

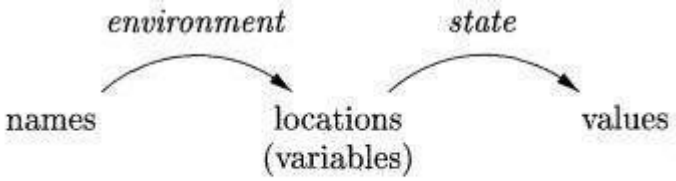


Figure 1.8: Two-stage mapping from names to values

Static Scope and Block Structure

Most languages, including C and its family, use static scope. we consider static-scope rules for a language with blocks, where a *block* is a grouping of declarations and statements. C uses braces { and } to delimit a block; the alternative use of **begin** and **end** for the same purpose dates back to Algol.

A C program consists of a sequence of top-level declarations of variables and functions.Functions may have variable declarations within them, where variables include local variables and parameters. The scope of each such declaration is restricted to the function in which it appears. The scope of a top-level declaration of a name *x* consists of the entire program that follows, with the exception of those statements that lie within a function that also has a declaration of *x.*

A block is a sequence of declarations followed by a sequence of statements, all surrounded by braces. a declaration *D* "belongs" to a block *B* if *B* is the most closely nested block containing *D;* that is, *D* is located within *B,* but not within any block that is nested within *B.* The static-scope rule for variable declarations in a block-structured lan-guages is as follows. If declaration *D* of name *x* belongs to block *B,* then the scope of D is all of B, except for any blocks B' nested to any depth within J5, in which x is redeclared. Here, x is redeclared in B' if some other declaration D' of the same name x belongs to B'.

An equivalent way to express this rule is to focus on a use of a name x. Let Bi, i?2, • • • , Bk be all the blocks that surround this use of x, with Bk the smallest, nested within Bk-i, which is nested within Bk-2, and so on. Search for the largest i such that there is a declaration of x belonging to B^. This use of x refers to the declaration in B{. Alternatively, this use of x is within the scope of the declaration in Bi.

Explicit Access Control

Through 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.

Dynamic Scope

Any scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes. The term *dynamic scope,* however, usually refers to the following policy: a use of a name *x* refers to the declaration of *x* in the most recently called procedure with such a declaration. Dynamic scoping of this type appears only in special situations. We shall consider two ex-amples of

dynamic policies: macro expansion in the C preprocessor and method resolution in object-oriented programming.

Declarations and Definitions

Declarations tell us about the types of things, while definitions tell us about their values. Thus, i n t 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. Similarly, it is common to define a C function in one file and declare it in other files where the function is used.

Parameter Passing Mechanisms

In this section, we shall consider how the *actual parameters* (the parameters used in the call of a procedure) are associated with the *formal parameters* (those used in the procedure definition). Which mechanism is used determines how the calling-sequence code treats parameters. The great majority of languages use either "call-by-value," or "call-by-reference," or both.

Call - by - Value

In *call-by-value,* the actual parameter is evaluated (if it is an expression) or copied (if it is a variable). The value is placed in the location belonging to the corresponding formal parameter of the called procedure. This method is used in C and Java, and is a common option in C + + , as well as in most other languages. Call-by-value has the effect that all computation involving the formal parameters done by the called procedure is local to that procedure, and the actual parameters themselves cannot be changed.

Note, however, that in C we can pass a pointer to a variable to allow that variable to be changed by the callee. Likewise, array names passed as param eters in C, C + + , or Java give the called procedure what is in effect a pointer or reference to the array itself. Thus, if a is the name of an array of the calling procedure, and it is passed by value to corresponding formal parameter *x,* then an assignment such as x [i] = 2 really changes the array element a[2]. The reason is that, although *x* gets a copy of the value of *a,* that value is really a pointer to the beginning of the area of the store where the array named *a* is located.

Similarly, in Java, many variables are really references, or pointers, to the things they stand for. This observation applies to arrays, strings, and objects of all classes. Even though Java uses call-by-value exclusively, whenever we pass the name of an object to a called procedure, the value received by that procedure is in effect a pointer to the object. Thus, the called procedure is able to affect the value of the object itself.

Call - by - Reference

In *call-by-reference,* the address of the actual parameter is passed to the callee as the value of the corresponding formal parameter. Uses of the formal parameter in the code of the callee are implemented by following this pointer to the location indicated by the caller. Changes to the formal parameter thus appear as changes to the actual parameter.

If the actual parameter is an expression, however, then the expression is evaluated before the call, and its value stored in a location of its own. Changes to the formal parameter change this location, but can have no effect on the data of the caller.

Call-by-reference is used for "ref" parameters in C + + and is an option in many other languages. It is almost essential when the formal parameter is a large object, array, or structure. The reason is that strict call-by-value requires that the caller copy the entire actual parameter into the space belonging to the corresponding formal parameter. This copying gets expensive when the parameter is large. As we noted when discussing call-by-value, languages such as Java solve the problem of passing arrays, strings, or other objects by copying only a reference to those objects. The effect is that Java behaves as if it used call-by-reference for anything other than a basic type such as an integer or real.

Call - by - Name

A third mechanism — call-by-name — was used in the early programming language Algol **60.** It requires that the callee execute as if the actual parameter were substituted literally for the formal parameter in the code of the callee, as if the formal parameter were a macro standing for the actual parameter (with renaming of local names in the called procedure, to keep them distinct). When the actual parameter is an expression rather than a variable, some unintuitive behaviors occur, which is one reason this mechanism is not favored today.

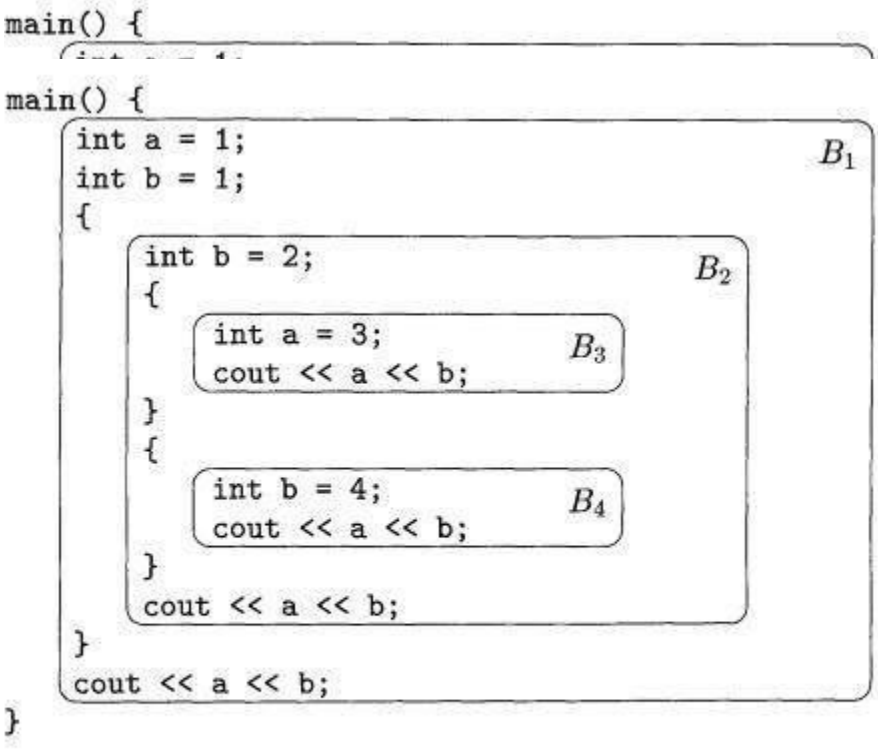


Figure 1.10: Blocks in a C++ program

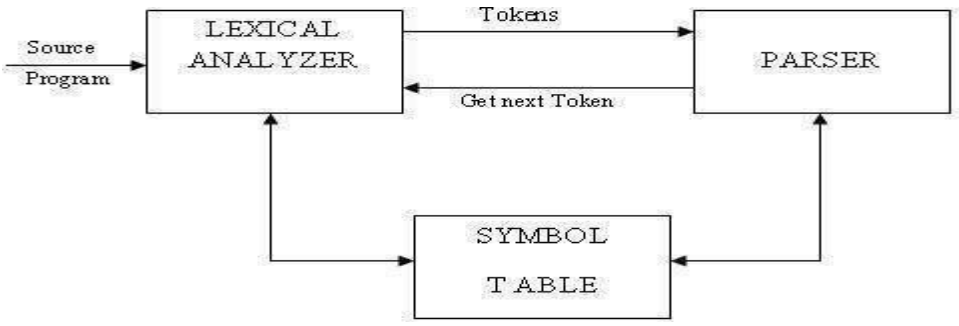
LEXICAL ANALYSIS

OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

ROLE OF LEXICAL ANALYZER

the LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.



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

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

LEXICAL ANALYSIS VS PARSING:

Lexical analysis	Parsing
A Scanner simply turns an input String (say a file) into a list of tokens. These tokens represent things like identifiers, parentheses, operators etc. The lexical analyzer (the "lexer") parses individual symbols from the source code file into tokens. From there, the "parser" proper turns those whole tokens into sentences of your grammar	A parser converts this list of tokens into a Tree-like object to represent how the tokens fit together to form a cohesive whole (sometimes referred to as a sentence). A parser does not give the nodes any meaning beyond structural cohesion. The next thing to do is extract meaning from this structure (sometimes called contextual analysis).

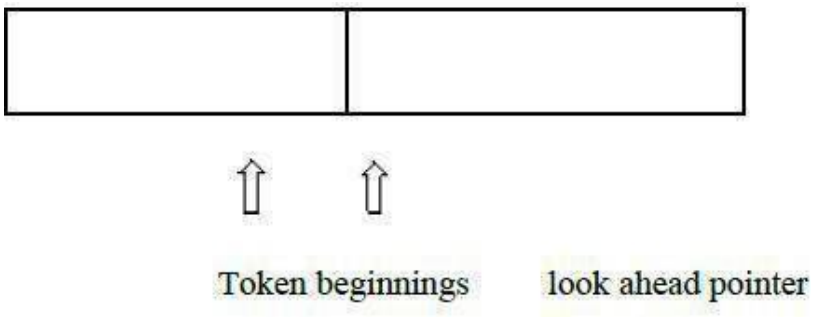
INPUT BUFFERING

The LA scans the characters of the source pgm one at a time to discover tokens. Because of large amount of time can be consumed scanning characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens. Often, however, many characters beyond the next token many have to be examined before the next token itself can be determined. For this and other reasons, it is desirable for thelexical analyzer to read its input from an input buffer. Figure shows a buffer divided into two haves of, say 100 characters each. One pointer marks the beginning of the token being discovered. A look ahead pointer scans ahead of the beginning point, until the token is discovered .we view the position of each pointer as being between the character last read and thecharacter next to be read. In practice each buffering scheme adopts one convention either apointer is at the symbol last read or the symbol it is ready to read.



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. For example, in a PL/I program we may see: DECALRE (ARG1, ARG2… ARG *n*) Without knowing whether DECLARE is a keyword or

an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example:

Description of token

Token	**lexeme**	**pattern**
const	const	const
if	if	If
relation	<,<=,= ,< >,>=,>	< or <= or = or < > or >= or letter followed by letters & digit
i	pi	any numeric constant
nun	3.14	any character b/w “and “except"
literal	"core"	pattern

A pattern is a rule describing the set of lexemes that can represent a particular token in source program.

LEXICAL ERRORS:

Lexical errors are the errors thrown by your lexer when unable to continue. Which means that there's no way to recognise a *lexeme* as a valid *token* for you lexer. Syntax errors, on the other side, will be thrown by your scanner when a given set of **already** recognised valid tokens don't match any of the right sides of your grammar rules. simple panic-mode error handling system requires that we return to a high-level parsing function when a parsing or lexical error is detected.

Error-recovery actions are:

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

DIFFERENCE BETWEEN COMPILER AND INTERPRETER

A compiler converts the high level instruction into machine language while an interpreter converts the high level instruction into an intermediate form.

Before execution, entire program is executed by the compiler whereas after translating the first line, an interpreter then executes it and so on.

List of errors is created by the compiler after the compilation process while an interpreter stops translating after the first error.

An independent executable file is created by the compiler whereas interpreter is required by an interpreted program each time.

The compiler produce object code whereas interpreter does not produce object code. In the process of compilation the program is analyzed only once and then the code is generated whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. hence interpreter is less efficient than compiler.

Examples of interpreter: A *UPS Debugger* is basically a graphical source level debugger but it contains built in C interpreter which can handle multiple source files.

Example of compiler: *Borland c compiler* or Turbo C compiler compiles the programs written in C or C++.

REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string.

Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R2R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- o is a regular expression denoting { € }, that is, the language containing only the empty string.
- o For each „a‟ in ∑, is a regular expression denoting { a }, the language with only one string consisting of the single symbol „a‟ .
- o If R and S are regular expressions, then

(R) | (S) means LrULs
R.S means Lr.Ls
R* denotes Lr*

REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.

Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | ……| Z | a | b |……| z|

Digits - 0 | 1 | 2 | …. | 9

letter (letter / digit)* I

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

```
Stmt  ->  if expr then stmt
        | If expr then else stmt
        | є
Expr -->  term relop term
        |term

Term -->id
```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes. The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

```
digit     -->[0,9]
digits    -->digit+
number -->digit(.digit)?(e.[+-]?digits)?
letter    -->[A-Z,a-z]
id        -->letter(letter/digit)*
if        --> if
then      -->then
else      -->else
relop     --></>/<=/>=/==/< >
```

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

ws --> (blank/tab/newline)$^{+}$

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 parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	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	ET
< >	relop	NE

TRANSITION DIAGRAM:

Transition Diagram has 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.

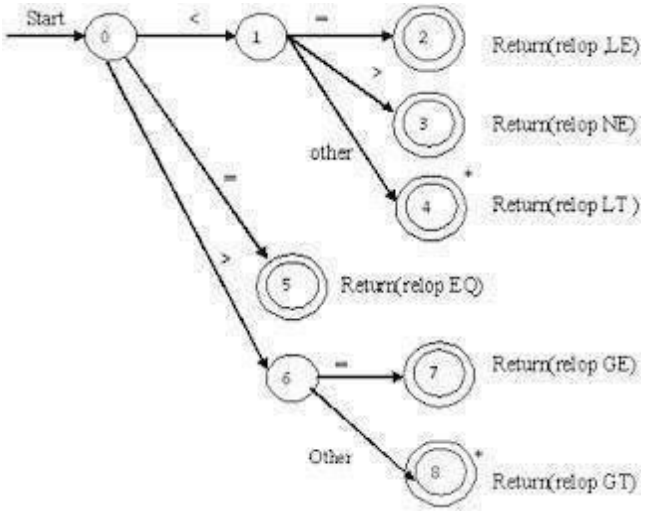
If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

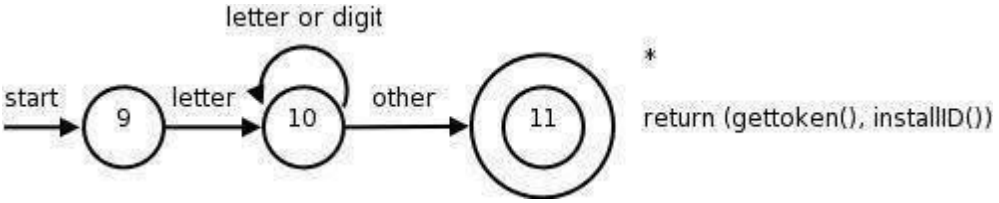
1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.

3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.



As an intermediate step in the construction of a LA, we first produce a stylized flowchart, called a transition diagram. Position in a transition diagram, are drawn as circles and are called as states.



The above TD for an identifier, defined to be a letter followed by any no of letters or digits.A sequence of transition diagram can be converted into program to look for the tokens specified by the diagrams. Each state gets a segment of code.

If	=	if
Then	=	then
Else	=	else
Relop	=	< | <= | = | > | >=
Id	=	letter (letter | digit) *|
Num	=	digit |

2.10 AUTOMATA

An automation is defined as a system where information is transmitted and used for performing some functions without direct participation of man.

1, an automation in which the output depends only on the input is **called an automation without memory.**

2, an automation in which the output depends on the input and state also is **called as automation with memory**.

3, an automation in which the output depends only on the state of the machine is **called a Moore machine**.

3, an automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

DESCRIPTION OF AUTOMATA

1, an automata has a mechanism to read input from input tape,

2, any language is recognized by some automation, Hence these automation are basically language „acceptors" or „language recognizers".

Types of Finite Automata

- Deterministic Automata
- Non-Deterministic Automata.

DETERMINISTIC AUTOMATA

A deterministic finite automata has at most one transition from each state on any input. A DFA is a special case of a NFA in which:-

1, it has no transitions on input € ,

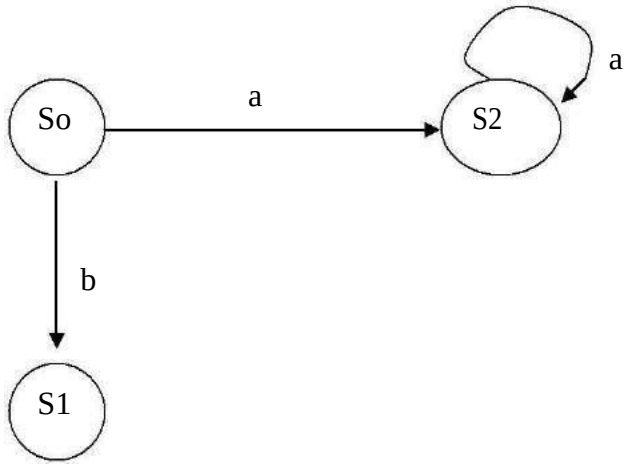
2, each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where
Q is a finite „set of states", which is non empty.
∑ is „input alphabets", indicates input set.
qo is an „initial state" and qo is in Q ie, qo, ∑, Q
F is a set of „Final states",
δ is a „transmission function" or mapping function, using this function the next state can be determined.

The regular expression is converted into minimized DFA by the following procedure:

Regular expression → NFA → DFA → Minimized DFA

The Finite Automata is called DFA if there is only one path for a specific input from current state to next state.



From state S0 for input „a" there is only one path going to S2. similarly from S0 there is only one path for input going to S1.

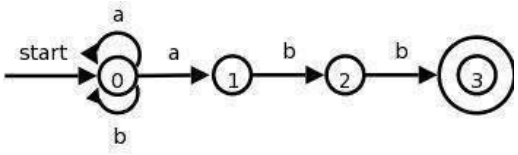
NONDETERMINISTIC AUTOMATA

- A NFA is a mathematical model that consists of
  - A set of states S.
  - A set of input symbols ∑.
  - A transition for move from one state to an other.
  - A state so that is distinguished as the start (or initial) state.
  - A set of states F distinguished as accepting (or final) state.
  - A number of transition to a single symbol.

- A NFA can be diagrammatically represented by a labeled directed graph, called a transition graph, In which the nodes are the states and the labeled edges represent the transition function.

- This graph looks like a transition diagram, but the same character can label two or more transitions out of one state and edges can be labeled by the special symbol € as well as by input symbols.

- The transition graph for an NFA that recognizes the language (a | b) * abb is shown



DEFINITION OF CFG

It involves four quantities.

CFG contain terminals, N-T, start symbol and production.

- Terminal are basic symbols form which string are formed.
- N-terminals are synthetic variables that denote sets of strings
- In a Grammar, one N-T are distinguished as the start symbol, and the set of string it denotes is the language defined by the grammar.
- The production of the grammar specify the manor in which the terminal and N-T can be combined to form strings.
- Each production consists of a N-T, followed by an arrow, followed by a string of one terminal and terminals.

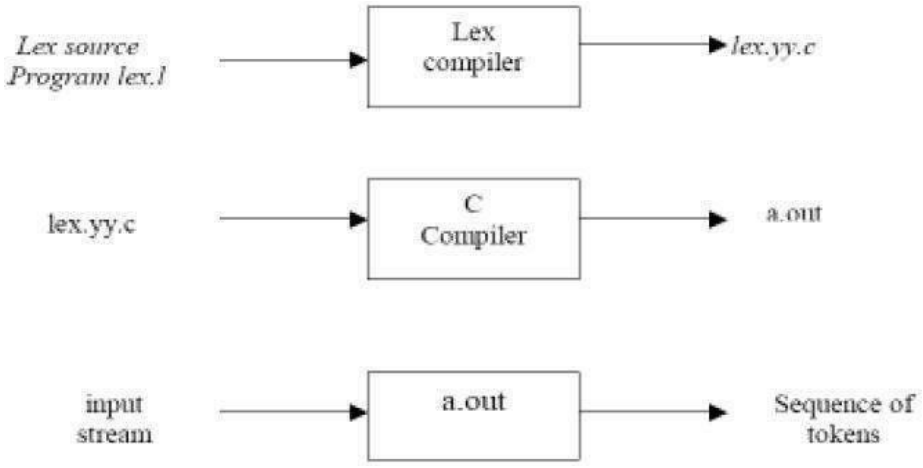
DEFINITION OF SYMBOL TABLE

- An extensible array of records.
- The identifier and the associated records contains collected information about the identifier.

  FUNCTION identify (Identifier name)

  RETURNING a pointer to identifier information contains
- The actual string
- A macro definition A
- keyword definition
- A list of type, variable & function definition
- A list of structure and union name definition
- A list of structure and union field selected definitions.

Creating a lexical analyzer with Lex



Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :

   p1 *{action 1}*
   p2 *{action 2}*
   p3 *{action 3}*
   ... ...
   ... ...

   where each *p* is a regular expression and each *action* is a program fragment describing what action the lexical analyzer should take when a pattern *p* matches a lexeme. In Lex the actions are written in C.

3. The third section holds whatever *auxiliary procedures* are needed by the *actions.*Alternatively these procedures can be compiled separately and loaded with the lexical analyzer.

Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

UNIT –II

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

UNIT -2

SYNTAX ANALYSIS

ROLE OF THE PARSER

Parser obtains a string of tokens from the lexical analyzer and verifies that it can be generated by the language for the source program. The parser should report any syntax errors in an intelligible fashion. The two types of parsers employed are:

1.Top down parser: which build parse trees from top(root) to bottom(leaves)

2.Bottom up parser: which build parse trees from leaves and work up the root.

Therefore there are two types of parsing methods– top-down parsing and bottom-up parsing

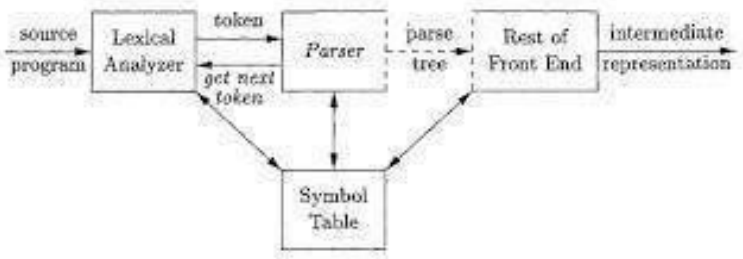


Figure 4.1: Position of parser in compiler model

Context free Grammars(CFG)

CFG is used to specify the syntax of a language. A grammar naturally describes the hierarchical structure of most program-ming language constructs.

Formal Definition of Grammars

A context-free grammar has four components:

1. A set of terminal symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.
2. A set of nonterminals, sometimes called "syntactic variables." Each non-terminal represents a set of strings of terminals, in a manner we shall describe.
3. A set of productions, where each production consists of a nonterminal, called the head or left side of the production, an arrow, and a sequence of terminals and1or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body

represents a written form of the construct.

.

4. A designation of one of the nonterminals as the *start* symbol.

Production is for a nonterminal if the nonterminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as ***E* ,** is called the empty string.

Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

Leftmost and Rightmost Derivation of a String

- **Leftmost derivation** − A leftmost derivation is obtained by applying production to the leftmost variable in each step.
- **Rightmost derivation** − A rightmost derivation is obtained by applying production to the rightmost variable in each step.
- **Example**

Let any set of production rules in a CFG be

$$X \rightarrow X+X \mid X*X \mid X \mid a$$

over an alphabet {a}.

The leftmost derivation for the string **"a+a*a"** is

$$X \rightarrow X+X \rightarrow a+X \rightarrow a + X*X \rightarrow a+a*X \rightarrow a+a*a$$

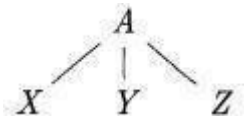
The rightmost derivation for the above string **"a+a*a"** is

$$X \rightarrow X*X \rightarrow X*a \rightarrow X+X*a \rightarrow X+a*a \rightarrow a+a*a$$

Derivation or Yield of a Tree

The derivation or the yield of a parse tree is the final string obtained by concatenating the labels of the leaves of the tree from left to right, ignoring the Nulls. However, if all the leaves are Null, derivation is Null.

parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production A → XYZ , then a parse tree may have an interior node labeled A with three children labeled X, Y, and Z, from left to right:



Given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.
2. Each leaf is labeled by a terminal or by e.
3. Each interior node is labeled by a nonterminal

.

If A is the nonterminal labeling some interior node and X I , Xz, . . . ,Xn are the labels of the children of that node from left to right, then there must be a production A → X1X2 . . Xn . Here, X1, X2,. . . , Xn, each stand for a symbol that is either a terminal or a nonterminal. As a special case, if A → **c** is a production, then a node labeled A may have a single child labeled ***E***

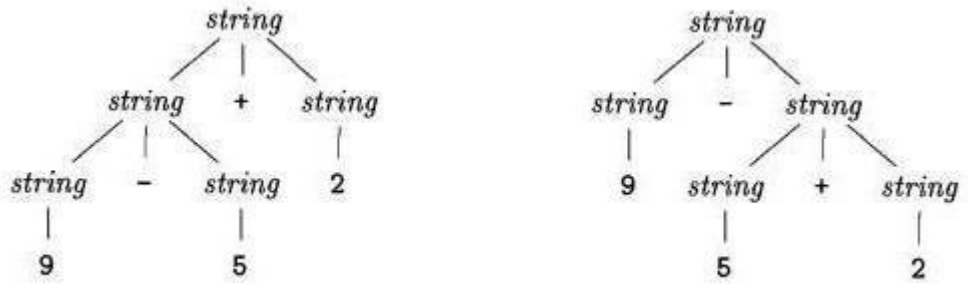
Ambiguity

A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous.* To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

Example :: Suppose we used a single nonterminal *string* and did not distinguish between digits and lists,

string → *string* + *string* | *string* − *string* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Fig. shows that an expression like 9-5+2 has more than one parse tree with this grammar. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5) +2 and 9- (5+2) . This second parenthesization gives the expression the unexpected value 2 rather than the customary value 6.



Two parse trees for 9-5+2

Verifying the language generated by a grammar

The set of all strings that can be derived from a grammar is said to be the language generated from that grammar. A language generated by a grammar **G** is a subset formally defined by

$$L(G)=\{W|W \in \sum^*, S \Rightarrow G\ \mathbf{W}\}$$

If **L(G1) = L(G2)**, the Grammar **G1** is equivalent to the Grammar **G2**.

Example

If there is a grammar

G: N = {S, A, B} T = {a, b} P = {S → AB, A → a, B → b}

Here **S** produces **AB**, and we can replace **A** by **a**, and **B** by **b**. Here, the only accepted string is **ab**, i.e.,

L(G) = {ab}

.

Writing a grammar

A *grammar* consists of a number of *productions*. Each production has an abstract symbol called a *nonterminal* as its *left-hand side*, and a sequence of one or more nonterminal and *terminal* symbols as its *right-hand side*. For each grammar, the terminal symbols are drawn from a specified *alphabet*.

Starting from a sentence consisting of a single distinguished nonterminal, called the *goal symbol*, a given context-free grammar specifies a *language*, namely, the set of possible sequences of terminal symbols that can result from repeatedly replacing any nonterminal in the sequence with a right-hand side of a production for which the nonterminal is the left-hand side.

There are **four categories** in writing a grammar :

1. Lexical Vs Syntax Analysis
2. Eliminating ambiguous grammar.
3. Eliminating left-recursion
4. Left-factoring.

Each parsing method can handle grammars only of a certain form hence, the initial grammar may have to be rewritten to make it parsable

1. Lexical Vs Syntax Analysis

Reasons for using the regular expression to define the lexical syntax of a language

a) Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
b) The lexical rules of a language are simple and to describe them, we donot need notation as powerful as grammars.
c) Efficient lexical analyzers can be constructed automatically from RE than from grammars.
d) Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end into two manageable-sized components.

2. Eliminating ambiguous grammar.

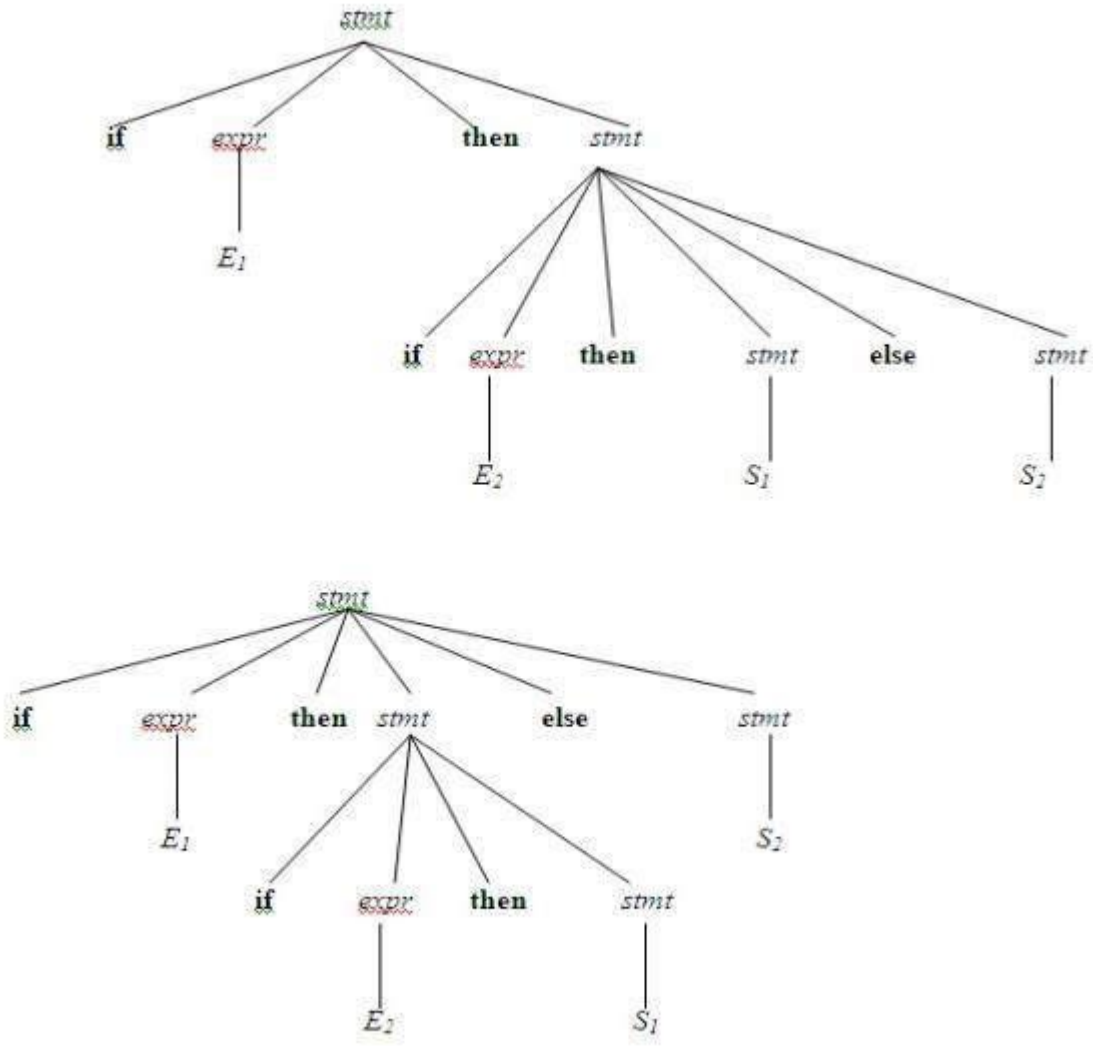
Ambiguity of the grammar that produces more than one parse tree for leftmost or rightmost derivation can be eliminated by re-writing the grammar.

Consider this example,

G: *stmt*→**if** *expr* **then** *stmt*
|**if** *expr* **then** *stmt* **else** *stmt*
|**other**

This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2 has the** following two parse trees for leftmost derivation

.



Two parse trees for an ambiguous sentence

The general rule is "Match each else with the closest unmatched then.This disambiguating rule can be used directly in the grammar,

To eliminate ambiguity, the following grammar may be used:

stmt→matched | unmatchedstmt

matched→**if** *expr stmt* **then** *matched* **else** *matchedstmt* | **other**

unmatched→ **if** *expr* **then** *stmt* | **if** *expr* **then** *matched* **else** *unmatchedstmt*

3. Eliminating left-recursion

.

Because we try to generate a leftmost derivation by scanning the input from left to right, grammars of the form $A \rightarrow A\ x$ may cause endless recursion.Such grammars are called left-recursive and they must be transformed if we want to use a top-down parser.

- A grammar is left recursive if for a non-terminal A, there is a derivation $A \Rightarrow^{+} A\alpha$

- **To eliminate direct left recursion replace**

1) $A \rightarrow A\alpha \mid \beta$ **with** $A' \rightarrow \alpha A' \mid \varepsilon$

2) $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_n$

with

$A \rightarrow \beta_1 B \mid \beta_2 B \mid ... \mid \beta_n B$

$B \rightarrow \alpha_1 B \mid \alpha_2 B \mid ... \mid \alpha_m B \mid \varepsilon$

4. Left-factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

- Consider $S \rightarrow$ if E then S else S | if E then S
  - Which of the two productions should we use to expand non-terminal S when the next token is if?

We can solve this problem by factoring out the common part in these rules.

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \gamma$

becomes

$A \rightarrow \alpha B \mid \gamma$

$B \rightarrow \beta_1 \mid \beta_2 \mid ... \mid \beta_n$

Consider the grammar , G : $S \rightarrow$ iEtS | iEtSeS | a

$E \rightarrow b$

Left factored, this grammar becomes

$S \rightarrow$ iEtSS' | a

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

PARSING

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

Types of parsing:

1. Top down parsing
2. Bottom up parsing

Top-down parsing : A parser can start with the start symbol and try to transform it to the input string. **Example :** LL Parsers.

.

Bottom-up parsing : A parser can start with input and attempt to rewrite it into the start symbol.
Example : LR Parsers.

TOP-DOWN PARSING

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

Types of TOP-DOWN PARSING

1. Recursive descent parsing
2. Predictive parsing

RECURSIVE DESCENT PARSING

Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right. It uses procedures for every non-terminal entity. This parsing technique recursively parses the input to make a parse tree, which may or may not require back-tracking. But the grammar associated with it (if not left factored) cannot avoid back-tracking. A form of recursive-descent parsing that does not require any back-tracking is known as **predictive parsing**.

This parsing technique is regarded recursive as it uses context-free grammar which is recursive in nature.

This parsing method may involve backtracking.

Example for :backtracking

Consider the grammar G : S → cAd

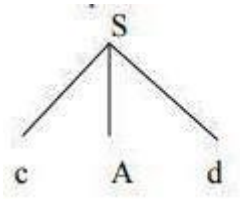
A→ab|a

and the input string w=cad.

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

Step1:

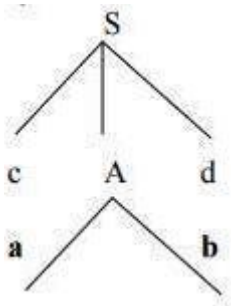
Initially create a tree with single node labeled S. An input pointer points to 'c', the first symbol of w. Expand the tree with the production of S.



Step2:

The leftmost leaf 'c' matches the first symbol of w, so advance the input pointer to the second symbol of w 'a' and consider the next leaf 'A'. Expand A using the first alternative.

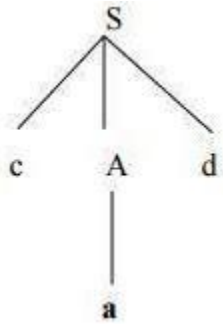
.



Step3:

The second symbol 'a' of w also matches with second leaf of tree. So advance the input pointer to third symbol of w 'd'**.** But the third leaf of tree is b which does not match with the input symbol **d.** Hence discard the chosen production and reset the pointer to second **backtracking.**

Step4: Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Predictive parsing

It is possible to build a nonrecursive predictive parser by maintaining a stack explicitly, rather than implicitly via recursive calls. The key problem during predictive parsing is that of determining the production to be applied for a nonterminal . The nonrecursive parser in figure looks up the production to be applied in parsing table. In what follows, we shall see how the table can be constructed directly from certain grammars.

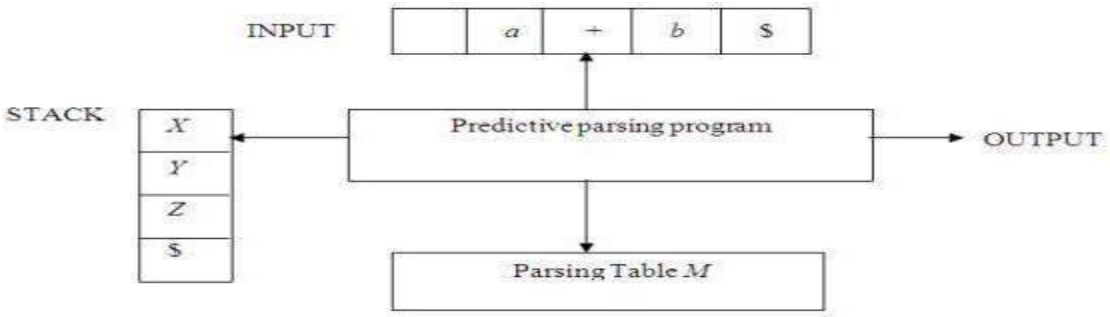


Fig. 2.4 Model of a nonrecursive predictive parser

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream. The input buffer contains the string to be parsed, followed by $, a symbol used as a right endmarker to indicate the end of the input string. The stack contains a sequence of grammar symbols

.

with $ on the bottom, indicating the bottom of the stack. Initially, the stack contains the start symbol of the grammar on top of $. The parsing table is a two dimensional array M[A,a] where A is a nonterminal, and a is a terminal or the symbol $. The parser is controlled by a program that behaves as follows. The program considers X, the symbol on the top of the stack, and a, the current input symbol. These two symbols determine the action of the parser. There are three possibilities.

1 If X= a=$, the parser halts and announces successful completion of parsing.
2 If X=a!=$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3 If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry. If, for example, M[X,a]={X- >UVW}, the parser replaces X on top of the stack by WVU(with U on top). As output, we shall assume that the parser just prints the production used; any other code could be executed here. If M[X,a]=error, the parser calls an error recovery routine

Implementation of predictive parser:

1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table

Algorithm for Nonrecursive 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 in which it has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is shown in Fig.

```
set ip to point to the first symbol of w$. repeat

    let X be the top stack symbol and a the symbol pointed to by ip. if X is a terminal of $ then
      if X=a then
         pop X from the stack and advance ip else error()
      else
      if M[X,a]=X->Y1Y2...Yk then begin pop X from the stack;
         push Yk,Yk-1...Y1 onto the stack, with Y1 on top; output the production X-> Y1Y2...Yk
      end
      else error()
  until X=$ /* stack is empty */
```

.

FIRST AND FOLLOW

The construction of a predictive parsing table is aided by two functions associated with a grammar:

1. FIRST
2. FOLLOW

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.

Rules for FIRST **( ):**

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

Rules for FOLLOW ( ):

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

Algorithm for construction of predictive parsing table:

Input : Grammar G
Output : Parsing table M
Method :

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

.

Example:

Consider the following grammar :

E→E+T|T

T→T*F|F

F→(E)|id

After eliminating left-recursion the grammar is

E →TE'

E' → +TE' | ε

T →FT'

T' → *FT' | ε

F → (E)|id

First( ) :

FIRST(E) = { (, id}

FIRST(E') ={+ , ε }

FIRST(T) = { (, id}

FIRST(T') = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

FOLLOW(E') = { $,) }

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

FOLLOW(T') = { +, $,) }

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

Predictive parsing Table

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

.

Stack Implementation

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

.

LL(1) GRAMMAR

The parsing table algorithm can be applied to any grammar G to produce a parsing table M. For some Grammars, for example if G is left recursive or ambiguous, then M will have at least one multiply-defined entry. A grammar whose parsing table has no multiply defined entries is said to be LL(1). It can be shown that the above algorithm can be used to produce for every LL(1) grammar G, a parsing table M that parses all and only the sentences of G. LL(1) grammars have several distinctive properties. No ambiguous or left recursive grammar can be LL(1). There remains a question of what should be done in case of multiply defined entries. One easy solution is to eliminate all left recursion and left factoring, hoping to produce a grammar which will produce no multiply defined entries in the parse tables. Unfortunately there are some grammars which will give an LL(1) grammar after any kind of alteration. In general, there are no universal rules to convert multiply defined entries into single valued entries without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use the translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence for expressions.however, if an lr parser generator is available, one can get all the benefits of predictive parsing and operator precedence automatically.

ERROR RECOVERY IN PREDICTIVE PARSING

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and the parsing table entry M[A,a] is empty.

Consider error recovery predictive parsing using the following two methods

Panic-Mode recovery
Phrase Level recovery

Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice.

As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the

.

synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.

If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

Phrase Level recovery

This involves, defining the blank entries in the table with pointers to some error routines which may

Change, delete or insert symbols in the input or
May also pop symbols from the stack

BOTTOM-UP PARSING

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

2.6.1 SHIFT-REDUCE PARSING

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

Example:

Consider the grammar: $S \rightarrow aABe$
$A \rightarrow Abc \mid b$
$B \rightarrow d$

The sentence to be recognized is abbcde.

REDUCTION (LEFTMOST)	**RIGHTMOST DERIVATION**
abbcde ($A \rightarrow b$)	$S \rightarrow aABe$
aAbcde($A \rightarrow Abc$)	$\rightarrow aAde$
aAde ($B \rightarrow d$)	$\rightarrow aAbcde$
aABe ($S \rightarrow aABe$)	$\rightarrow abbcde$
S	

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

.

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

Example:

Consider the grammar:

E→E+E
E→E*E
E→(E)
E→id

And the input string id1+id2*id3

The rightmost derivation is :

E→E+E
→ E+E*E
→ E+E*id3
→ E+id2*id3
→ id1+id2*

In the above derivation the underlined substrings are called handles.

Handle pruning:

A rightmost derivation in reverse can be obtained by "handle pruning". (i.e.) if w is a sentence or string of the grammar at hand, then w = γn, where γn is the nth right sentential form of some rightmost derivation.

Actions in shift-reduce parser:

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

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.
2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

.

Stack implementation of shift-reduce parsing :

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

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	
$E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$E	*id $	Shift	$E+E*	id $	Shift
$E*	id $	Shift	$E+E*id	$	Reduce by E→id
$E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$E			$E		

.

2. Reduce-reduce conflict:

Consider the grammar: M→R+R|R+c|R

R→c

input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$c	+c $	Reduce by R→c	$c	+c $	Reduce by R→c
$R	+c $	Shift	$R	+c $	Shift
$R+	c $	Shift	$R+	c $	Shift
$R+c	$	Reduce by R→c	$R+c	$	Reduce by M→R+c
$R+R	$	Reduce by M→R+R	$M	$	
$M	$				

INTRODUCTION TO LR PARSERS

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

Advantages of LR parsing:

1. It recognizes virtually all programming language constructs for which CFG can be written.
2. It is an efficient non-backtracking shift-reduce parsing method.
3. A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser
4. It detects a syntactic error as soon as possible.

Drawbacks of LR method:

It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR

   Easiest to implement, least powerful.

2. CLR- Canonical LR

   Most powerful, most expensive.

3. LALR- Look-Ahead LR

   Intermediate in size and cost between the other two methods.

.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:

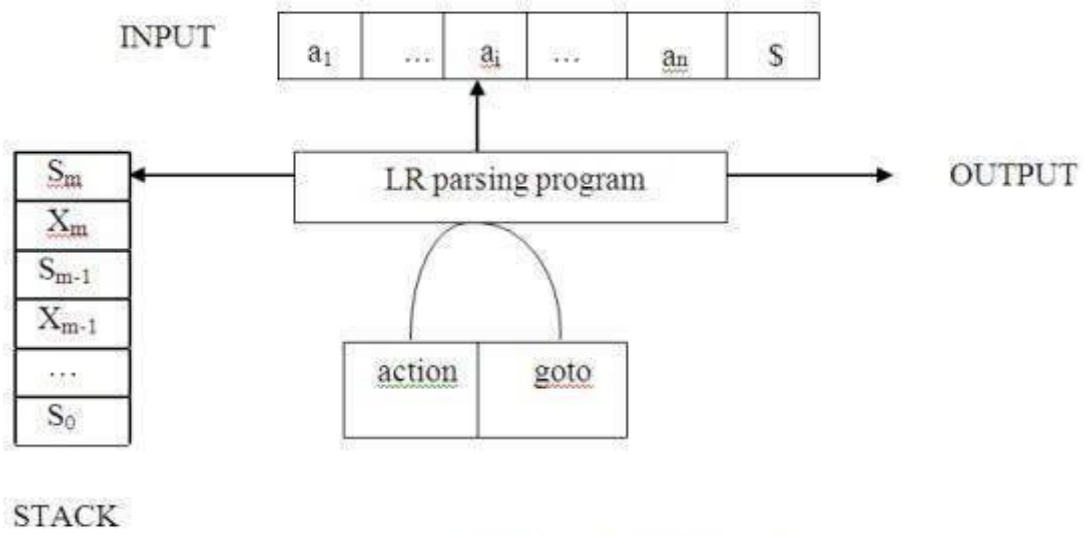


Fig. 2.5 Model of an LR parser

It consists of an input, an output, a stack, a driver program, and a pa parts (action and goto).

a. The driver program is the same for all LR parser.
b. The parsing program reads characters from an input buffer one at a time.
c. The program uses a stack to store a string of the form s0X1s1X2s2…Xmsm, where sm is on top. Each Xi is a grammar symbol and each si is a state.
d. The parsing table consists of two parts : action and goto functions.

Action : The parsing program determines sm, the state currently on top of stack, and ai, the current input symbol. It then consults action[sm,ai] in the action table which can have one of four values:

1. shift s, where s is a state,
2. reduce by a grammar production A → β,
3. accept,
4. Error.

Goto : The function goto takes a state and grammar symbol as arguments and produces a state.

LR Parsing algorithm:

Input: An input string w and an LR parsing table with functions action and goto for grammar G.
Output: If w is in L(G), a bottom-up-parse for w; otherwise, an error indication.
Method: Initially, the parser has s0 on its stack, where s0 is the initial state, and w$ in the input buffer.

The parser then executes the following program:

.

```
set ip to point to the first input symbol of w$; repeat forever begin
        let s be the state on top of the stack and a the symbol pointed to by ip;
if action[s, a] = shift s' then begin
        push a then s' on top of the stack; advance ip to the next input symbol end
        else if action[s, a] = reduce A→β then begin pop 2* | β | symbols off the stack;
        let s' be the state now on top of the stack; push A then goto[s', A] on top of the stack; output the
        production A→ β
        end
        else if action[s, a] = accept then
        return
        else error( )
        end
```

CONSTRUCTING SLR(1) PARSING TABLE

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

1. Find LR(0) items.
2. Completing the closure.
3. Compute goto(I,X), where, I is set of items and X is grammar symbol.

LR(0) items:

An LR(0) item of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A→.XYZ
A → X . YZ
A → XY . Z
A → XYZ .

Closure operation:

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

1. Initially, every item in I is added to closure(I).
2. If A → α . Bβ is in closure(I) and B → γ is a production, then add the item B → . γ to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ αX . β] such that [A→ α . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function action and goto using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

.

Algorithm for construction of SLR parsing table:

Input : An augmented grammar G'

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

Method :

1. Construct C = {I0, I1, …. In}, the collection of sets of LR(0) items for G'.
2. State i is constructed from Ii.. The parsing functions for state i are determined as follows:
   (a) If [A→α·aβ] is in Ii and goto(Ii,a) = Ij, then set action[i,a] to "shift j". Here a must be terminal.
   (b) If [A→α·] is in Ii , then set action[i,a] to "reduce A→α" for all a in FOLLOW(A).
   (c) If [S'→S.] is in Ii, then set action[i,$] to "accept".
   If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).
3. The goto transitions for state i are constructed for all non-term
   If goto(Ii,A) = Ij, then goto[i,A] = j.
4. All entries not defined by rules (2) and (3) are made "error"
5. The initial state of the parser is the one constructed from the [S'→.S].

Example on SLR (1) Grammar

S → E
E → E + T | T
T → T * F | F
F → id

Add Augment Production and insert '•' symbol at the first position for every production in G

S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I0 State:

Add Augment production to the I0 State and Compute the Closure

I0 = Closure (S` → •E)

Add all productions starting with E in to I0 State because "." is followed by the non-terminal. So, the I0 State becomes

I0 = S` → •E
E → •E + T
E → •T

Add all productions starting with T and F in modified I0 State because "." is followed by the non-terminal. So, the I0 State becomes.

I0= S` → •E
E → •E + T
E → •T
T → •T * F
T → •F
F → •id

I1= Go to (I0, E) = closure (S` → E•, E → E• + T)
I2= Go to (I0, T) = closure (E → T•T, T• → * F)
I3= Go to (I0, F) = Closure (T → F•) = T → F•
I4= Go to (I0, id) = closure (F → id•) = F → id•
I5= Go to (I1, +) = Closure (E → E +•T)

Add all productions starting with T and F in I5 State because "." is followed by the non-terminal. So, the I5 State becomes

I5 = E → E +•T
T → •T * F
T → •F
F → •id

Go to (I5, F) = Closure (T → F•) = (same as I3)
Go to (I5, id) = Closure (F → id•) = (same as I4)

I6= Go to (I2, *) = Closure (T → T * •F)

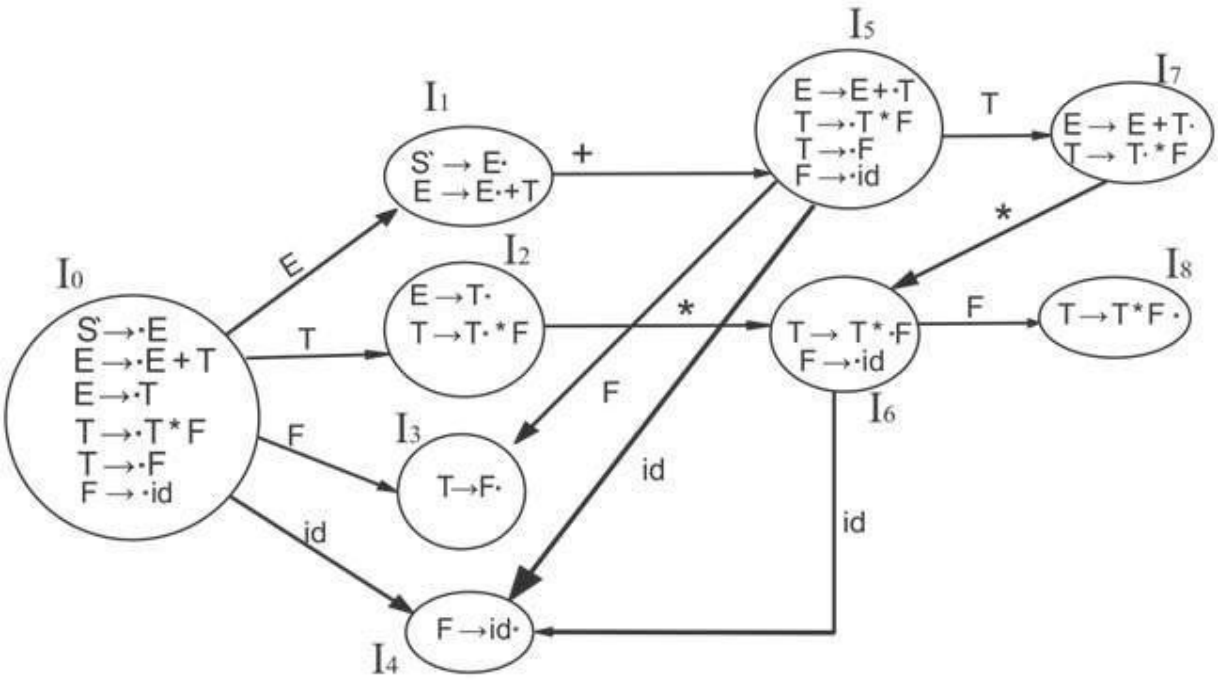
Add all productions starting with F in I6 State because "." is followed by the non-terminal. So, the I6 State becomes

I6 = T → T * •F
F → •id

Go to (I6, id) = Closure (F → id•) = (same as I4)

I7= Go to (I5, T) = Closure (E → E + T•) = E → E + T•
I8= Go to (I6, F) = Closure (T → T * F•) = T → T * F•

Drawing DFA



SLR (1) Table

States	Action				Go to		
	id	+	*	**$**	**E**	**T**	**F**
I_0	S_4				1	2	3
I_1		S_5		Accept			
I_2		R_2	S_6	R2			
I_3		R_4	R_4	R4			
I_4		R_5	R5	R5			
I_5	S4					7	3
I_6	S4						8
I_7		R1	S6	R1			
I_8		R3	R3	R3			

Explanation:

First (E) = First (E + T) U First (T)
First (T) = First (T * F) U First (F)
First (F) = {id}
First (T) = {id}
First (E) = {id}
Follow (E) = First (+T) U {$} = {+, $}
Follow (T) = First (*F) U First (F)
= {*, +, $}
Follow (F) = {*, +, $}

1) I1 contains the final item which drives S → E• and follow (S) = {$}, so action {I1, $} = Accept
2) I2 contains the final item which drives E → T• and follow (E) = {+, $}, so action {I2, +} = R2, action {I2, $} = R2
3) I3 contains the final item which drives T → F• and follow (T) = {+, *, $}, so action {I3, +} = R4, action {I3, *} = R4, action {I3, $} = R4
4) I4 contains the final item which drives F → id• and follow (F) = {+, *, $}, so action {I4, +} = R5, action {I4, *} = R5, action {I4, $} = R5
5) I7 contains the final item which drives E → E + T• and follow (E) = {+, $}, so action {I7, +} = R1, action {I7, $} = R1
6) I8 contains the final item which drives T → T * F• and follow (T) = {+, *, $}, so action {I8, +} = R3, action {I8, *} = R3, action {I8, $} = R3.

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.

UNIT -3

CANONICAL LR PARSING

CLR refers to canonical lookahead. CLR parsing use the canonical collection of LR (1) items to build the CLR (1) parsing table. CLR (1) parsing table produces the more number of states as compare to the SLR (1) parsing.

In the CLR (1), we place the reduce node only in the lookahead symbols.

Various steps involved in the CLR (1) Parsing:

1) For the given input string write a context free grammar
2) Check the ambiguity of the grammar
3) Add Augment production in the given grammar
4) Create Canonical collection of LR (0) items
5) Draw a data flow diagram (DFA)
6) Construct a CLR (1) parsing table

In the SLR method we were working with LR(0)) items. In CLR parsing we will be using LR(1) items. LR(k) item is defined to be an item using lookaheads of length k. So ,the LR(1) item is comprised of two parts : the LR(0) item and the lookahead associated with the item. The look ahead is used to determine that where we place the final item. The look ahead always add $ symbol for the argument production.

LR(1) parsers are more powerful parser.
for LR(1) items we modify the Closure and GOTO function.

Closure Operation

Closure(I)

repeat

for (each item [A -> ?.B?, a] in I)

for (each production B -> ? in G')

for (each terminal b in FIRST(?a))

add [B -> .? , b] to set I;

until no more items are added to I;

return I;

Goto Operation

```
Goto(I, X)
Initialise J to be the empty set;
for ( each item A -> ?.X?, a ] in I )
   Add item A -> ?X.?, a ] to se J;  /* move the dot one step */
return Closure(J);    /* apply closure to the set */
```

LR(1) items

```
Void items(G')
Initialise 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 set of items are added to C;
```

ALGORITHM FOR CONSTRUCTION OF THE CANONICAL LR PARSING TABLE

Input: grammar G'
Output: canonical LR parsing table functions action and goto

1. Construct C = {I0, I1 , ..., In} the collection of sets of LR(1) items for G'.State i is constructed from Ii.
2. if [*A* -> a.ab, b>] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.
3. if [*A* -> a., a] is in Ii, then set action[i, a] to "reduce *A* -> a" for all a in FOLLOW(*A*). Here *A* may *not* be *S*'.
4. if [*S*' -> *S*.] is in Ii, then set action[i, $] to "accept"
5. If any conflicting actions are generated by these rules, the grammar is not LR(1) and the algorithm fails to produce a parser.
6. The goto transitions for state i are constructed for all *nonterminal*s *A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j.
7. All entries not defined by rules 2 and 3 are made "error".
8. The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S, $].

Example,

Consider the following grammar,

S‟->S
S->CC
C->cC
C->d

Sets of LR(1) items

I0: S‟->.S,$
S->.CC,$
C->.Cc,c/d
C->.d,c/d

I1: S‟->S.,$

I2: S->C.C,$
C->.Cc,$
C->.d,$

I3: C->c.C,c/d C->.Cc,c/dC->.d,c/d

I4: C->d.,c/d

I5: S->CC.,$

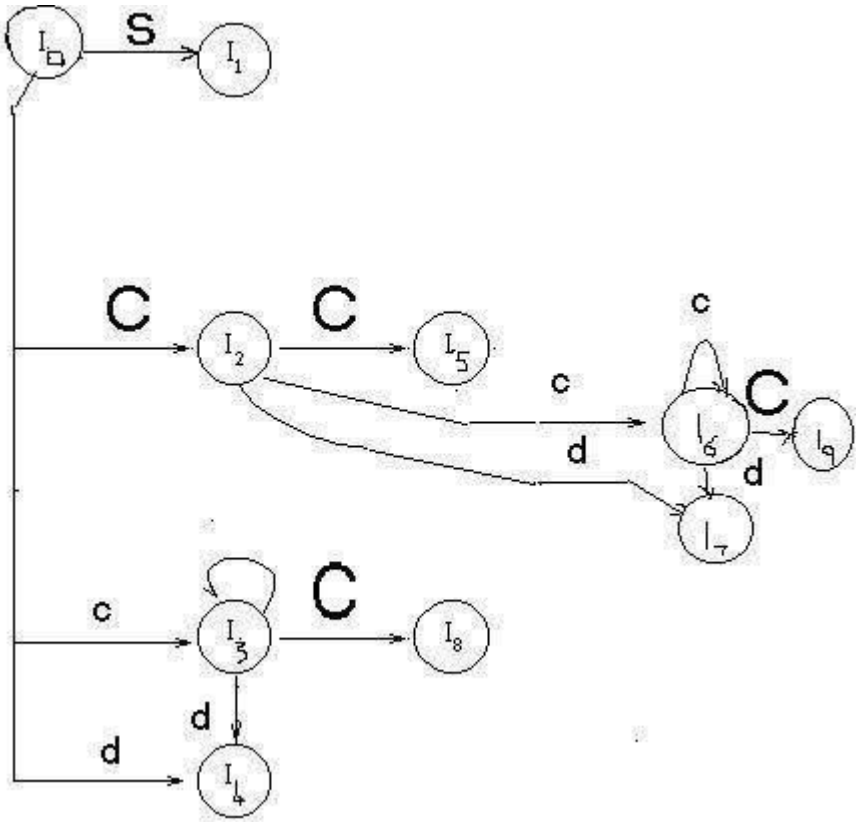
I6: C->c.C,$
C->.cC,$
C->.d,$

I7: C->d.,$

I8: C->cC.,c/d

I9: C->cC.,$

Here is what the corresponding DFA looks like



Parsing Table:state	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		

3.3.LALR PARSER:

We begin with two observations. First, some of the states generated for LR(1) parsing have the same set of core (or first) components and differ only in their second component, the lookahead symbol. Our intuition is that we should be able to merge these states and reduce the number of states we have, getting close to the number of states that would be generated for LR(0) parsing. This observation suggests a hybrid approach: We can construct the canonical LR(1) sets of items and then look for sets of items having the same core. We merge these sets with common cores into one set of items. The merging of states with common cores can never produce a shift/reduce conflict that was not present in one of the original states because shift actions depend only on the core, not the lookahead. But it is possible for the merger to produce a reduce/reduce conflict.

Our second observation is that we are really only interested in the lookahead symbol in places where there is a problem. So our next thought is to take the LR(0) set of items and add lookaheads only where they are needed. This leads to a more efficient, but much more complicated method.

ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

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

Method:

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

6. Then goto(J, X) = K.

Consider the above example,

I3 & I6 can be replaced by their union I36:C->c.C,c/d/$

C->.Cc,C/D/$

C->.d,c/d/$

I47:C->d.,c/d/$

I89:C->Cc.,c/d/$

Parsing Table

state	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

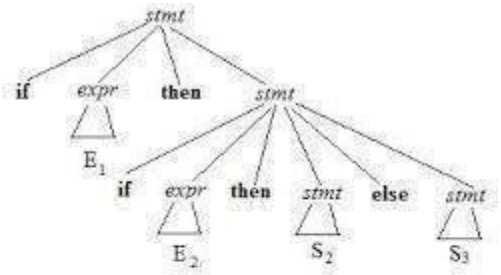
HANDLING ERRORS

The LALR parser may continue to do reductions after the LR parser would have spotted an error, but the LALR parser will never do a shift after the point the LR parser would have discovered the error and will eventually find the error.

DANGLING ELSE

The dangling else is a problem in computer programming in which an optional else clause in an If–then(–else) statement results in nested conditionals being ambiguous. Formally, the context-free grammar of the language is ambiguous, meaning there is more than one correct parse tree.

In many programming languages one may write conditionally executed code in two forms: the if-then form, and the if-then-else form – the else clause is optional:



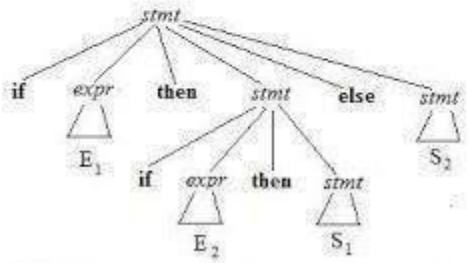


Fig 2.4 Two parse trees for an ambiguous sentence

Consider the grammar:

S ::= E $

E ::= E + E

| E * E

| (E)

| id

| num

and four of its LALR(1) states:

I0: S ::= . E $?

E ::= . E + E +*$

E ::= . E * E +*$

E ::= . (E) +*$

E ::= . id +*$

E ::= . num +*$

I1: S ::= E . $?

E ::= E . + E +*$

E ::= E . * E +*$

I2: E ::= E * . E +*$

E ::= . E + E +*$

E ::= . E * E +*$

E ::= . (E) +*$

E ::= . id +*$

E ::= . num +*$

I3: E ::= E * E . +*$

E ::= E . + E +*$

E ::= E . * E +*$

-

E ::= E . * E +*$

Here we have a shift-reduce error. Consider the first two items in I3. If we have a*b+c and we parsed a*b, do we reduce using E ::= E * E or do we shift more symbols? In the former case we get a parse tree (a*b)+c; in the latter case we get a*(b+c). To resolve this conflict, we can specify that * has higher precedence than +. The precedence of a grammar production is equal to the precedence of the rightmost token at the rhs of the production. For example, the precedence of the production E ::= E * E is equal to the precedence of the operator *, the precedence of the production E ::= (E) is equal to the precedence of the token), and the precedence of the production E ::= if E then E else E is equal to the precedence of the token else. The idea is that if the look ahead has higher precedence than the production currently used, we shift. For example, if we are parsing E + E using the production rule E ::= E + E and the look ahead is *, we shift *. If the look ahead has the same precedence as that of the current production and is left associative, we reduce, otherwise we shift. The above grammar is valid if we define the precedence and associativity of all the operators. Thus, it is very important when you write a parser using CUP or any other LALR(1) parser generator to specify associativities and precedence‟s for most tokens (especially for those used as operators). Note: you can explicitly define the precedence of a rule in CUP using the %prec directive:

E ::= MINUS E %prec UMINUS

where UMINUS is a pseudo-token that has higher precedence than TIMES, MINUS etc, so that -1*2 is equal to (-1)*2, not to -(1*2).

Another thing we can do when specifying an LALR(1) grammar for a parser generator is error recovery. All the entries in the ACTION and GOTO tables that have no content correspond to syntax errors. The simplest thing to do in case of error is to report it and stop the parsing. But we would like to continue parsing finding more errors. This is called *error recovery*. Consider the grammar:

```
S ::= L = E ;
   | { SL }
   ; | error ;
SL ::= S ; |
    SL S ;
```

The special token error indicates to the parser what to do in case of invalid syntax for S (an invalid statement). In this case, it reads all the tokens from the input stream until it finds the first semicolon. The way the parser handles this is to first push an error state in the stack. In case of an error, the parser pops out elements from the stack until it finds an error state where it can proceed. Then it discards tokens from the input until a restart is possible. Inserting error handling productions in the proper places in a grammar to do good error recovery is considered very hard.

LR ERROR RECOVERY

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.

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 non terminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

Syntax Directed Translations

We associate information with a language construct by attaching attributes to the grammar symbol(s) representing the construct, A syntax-directed definition specifies the values of attributes by associating semantic rules with the grammar productions. For example, an infix-to-postfix translator might have a production and rule

PRODUCTION	SEMANTIC RULE
$E \rightarrow Ei + T$	$E.code = Ei.code \parallel T.code \parallel '+'$

This production has two nonterminals, E and T; the subscript in E1 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 Ei.code, T.code, and the character '+'. While the rule makes it explicit that the translation of E is built up from the translations of E1, T, and '+', it may be inefficient to implement the translation directly by manipulating strings.

a syntax-directed translation scheme embeds program fragments called semantic actions within production bodies
There are two notations for attaching semantic rules:

1. **Syntax Directed Definitions.** High-level specification hiding many implementation details (also called **Attribute Grammars**).

2. **Translation Schemes.** More implementation oriented: Indicate the order in which semantic rules are to be evaluated.

Syntax Directed Definitions

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

1. Grammar symbols have an associated set of **Attributes**;
2. Productions are associated with **Semantic Rules** for computing the values of attributes Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammar symbol X).

The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

We distinguish between two kinds of attributes:

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

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

Syntax Directed Definitions: An Example

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

PRODUCTION	SEMANTIC RULE
PRODUCTION	SEMANTIC RULE
E -> Ei + T	E.code = Ei.code ‖ T.code ‖ '+'
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow$ digit	$F.val :=$ digit.*lexval*

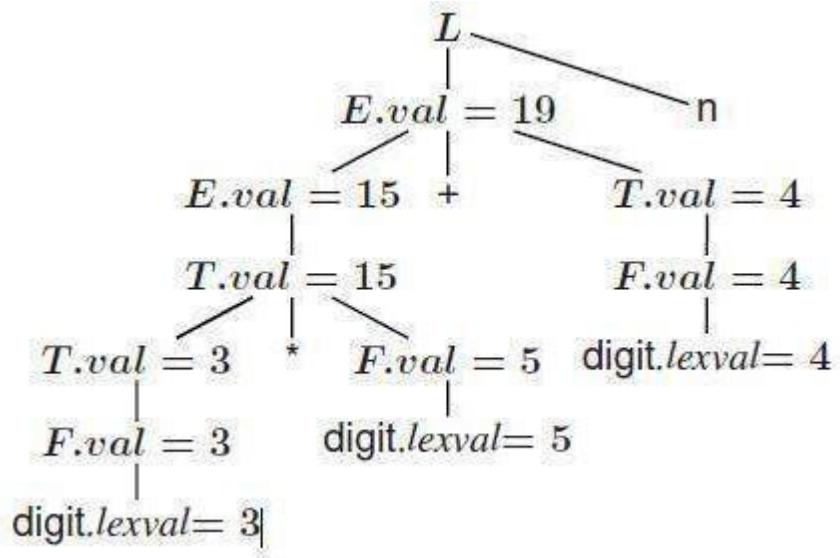
SDD of a simple desk calculator

S-ATTRIBUTED DEFINITIONS

Definition. An **S-Attributed Definition** is a Syntax Directed Definition that uses only synthesized attributes.

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

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



L-attributed definition

Definition: A SDD its *L-attributed* if each inherited attribute of Xi in the RHS of A ! X1 :

:Xn depends only on

1. attributes of X1;X2; : : : ;Xi 1 (symbols to the left of Xi in the RHS)
2. inherited attributes of A.

Restrictions for translation schemes:

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

Evaluation order of SDTS

"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

1 Dependency Graphs

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

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.*

Since a node ***N*** can have several children labeled ***X,*** we again assume that subscripts distinguish among uses of the same symbol at different places in the production.

Example: Consider the following production and rule:

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$

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. 5.6. As a convention, we shall show the parse tree edges as dotted lines, while the edges of the dependency graph are solid.

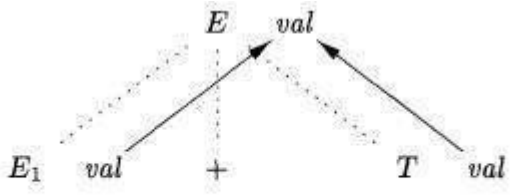


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

2. Ordering the Evaluation of Attributes

The dependency graph characterizes the possible orders in which we can evalu-ate the attributes at the various nodes of a parse tree. If the dependency graph has an edge from node M to node N, then the attribute corresponding to M must be evaluated before the attribute of N. Thus, the only allowable orders of evaluation are those sequences of nodes N1, N2,... ,Nk such that if there is an edge of the dependency graph from Ni to Nj, then i < j. Such an ordering embeds a directed graph into a linear order, and is called a topological sort of the graph.

If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree. If there are no cycles, however, then there is always at least one topological sort

3. S-Attributed Definitions

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

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.

4 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 -> X1 X2 ..... Xn, and that there is an inherited attribute Xi.a computed by a rule associated with this production.

Then the rule may use only:
Inherited attributes associated with the head A.
Either inherited or synthesized attributes associated with the occurrences of symbols X1, X2,... , X(i-1) located to the left of Xi.
Inherited or synthesized attributes associated with this occurrence of Xi itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X i

Application of SDTS

1 Construction of Syntax Trees
2 The Structure of a Type

The main application is the construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree.

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 $E1 + E_2$ has label + and two children representing the subexpressions *E1* and E_2

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,ci,c2,... ,ck) creates an object with first field op and k additional fields for the k children c1,... , .

Example

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

Figure 5.10: Constructing syntax trees for simple expressions

Figure 5.1 1 shows the construction of a syntax tree for the input *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.

.

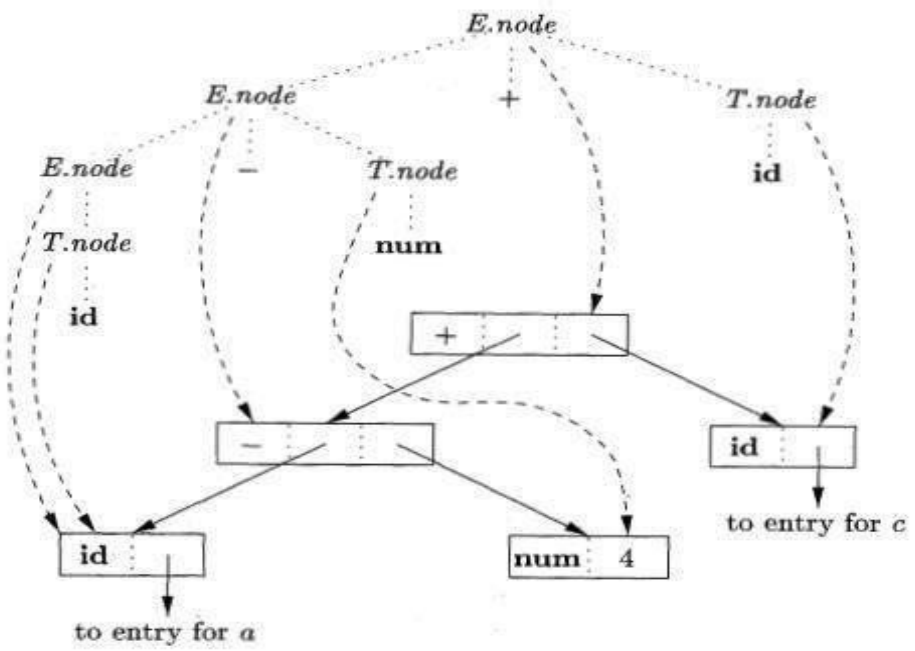


Figure 5.11: Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}a});$
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}c});$
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4);$

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

2 The Structure of a Type

The type int [2][3] can be read as, "array of 2 arrays of 3 integers." The corresponding type expression array(2, array(3, integer)) is represented by the tree in Fig. 5.15. The operator array takes two parameters, a number and a type. If types are represented by trees, then this operator returns a tree node labeled array with two children for a number and a type.

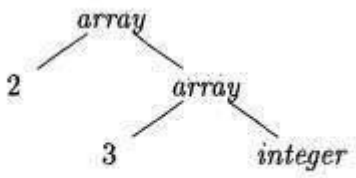


Figure 5.15: Type expression for **int**[2][3]

Nonterminal *B* generates one of the basic types **int** and **float.** *T* generates a basic type when *T* derives *B C* and *C* derives e. Otherwise, *C* generates array components consisting of a sequence of integers, each integer surrounded by brackets.

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

An annotated parse tree for the input string **int [** 2] [3] is shown in Fig. 5.17. The corresponding type expression in Fig. 5.15 is constructed by passing the type *integer* from *B,* down the chain of C's through the inherited attributes *b.* The array type is synthesized up the chain of C's through the attributes *t.*

In more detail, at the root for *T* -» *B C,* nonterminal *C* inherits the type from *B,* using the inherited attribute *C.b.* At the rightmost node for C, the production is C e, so C.t equals C.6. The semantic rules for the production C [num] C1 form C.t by applying the operator array to the operands num.ua/ and C1.t.

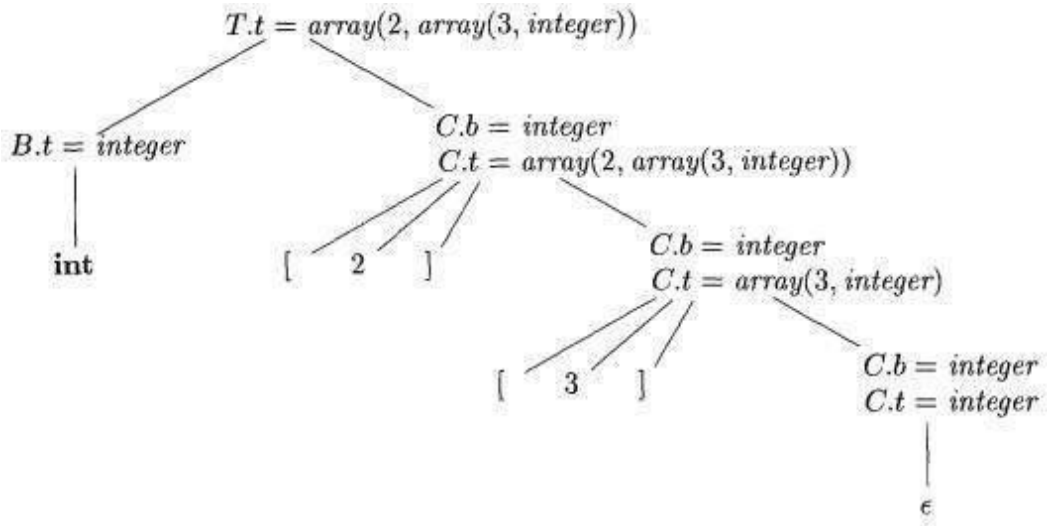


Figure 5.17: Syntax-directed translation of array types

Syntax Directed Translation Schemes.

1 Postfix Translation Schemes
2 Parser-Stack Implementation of Postfix SDT's
3 SDT's With Actions Inside Productions
4 Eliminating Left Recursion From SDT's

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 grammar symbols, then we quote them.SDT's are implemented during parsing, without building a parse tree.

Two important classes of SDD's are

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

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 5.14 : The postfix SDT in Fig. 5.18 implements the desk calculator SDD of Fig. 5.1, 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.

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \mathrm{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Figure 5.18: Postfix SDT implementing the desk calculator

2 Parser-Stack Implementation of Postfix SDT's

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. 5.19, 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 YZ* are on top of the stack; perhaps they

are about to be reduced according to a production like *A —> X YZ.* 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.

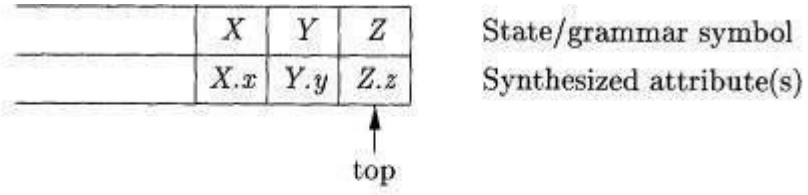


Figure 5.19: Parser stack with a field for synthesized attributes

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 -» 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).

4 Eliminating Left Recursion From SDT's

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

$$A \rightarrow Aa \mid b$$

that generate strings consisting of a *j3* and any number of en's, and replace them by productions that generate the same strings using a new nonterminal *R* (for "remainder") of the first production:

A->bR

R —»• aR | e

If *(3* 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 ***0(a)****.

Example **5** . 1 7 : Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

E -> E i + T { print('+'); }

E -> T

If we apply the standard transformation to E, the remainder of the left-recursive production is

a = + T { print('-r'); }

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 --> T R

R --> + T { printC-h'); } R

R -> e

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.