### UNIT - I  INTRODUCTION TO COMPILERS
Translators-Compilation and Interpretation-Language processors -The Phases of Compiler-Errors Encountered in Different Phases - The Grouping of Phases-Compiler Construction Tools- Programming Language basics

### UNIT - II  LEXICAL ANALYSIS
Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions- Converting Regular Expression to DFA- Minimization of DFA-Language for Specifying Lexical Analyzers-LEX-Design of Lexical Analyzer for a sample Language.

### UNIT- III  SYNTAX ANALYSIS

### UNIT- IV  SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT
Syntax directed Definitions-Construction of Syntax Tree-Bottom-up Evaluation of S-Attribute Definitions-Design of predictive translator - Type Systems-Specification of a simple type checker- Equivalence of Type Expressions-Type Conversions.
RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation- Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.

### UNIT- V  CODE OPTIMIZATION AND CODE GENERATION

**TOTAL: 45 PERIODS**

### TEXT BOOKS

### REFERENCES

### SUBJECT IN-CHARGE

<table>
<thead>
<tr>
<th>L T P C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 0 0 3</td>
</tr>
</tbody>
</table>

M.I.E.T./CSE/III Yr/Compiler Design
COURSE OBJECTIVE
Students will be able to:
1. Learn the principles of compiler and to design it.
2. Understand various parsing techniques and different levels of translation.
4. Optimize and effectively learn to generate machine codes.

COURSE OUTCOMES
On completion of course the students will be able to:
1. Design and implement a prototype compiler to correct code and execute.
2. Diagnose the data flow anomalies.
3. Work with debugger.
4. Adapt parallel processing and explore architecture interface by customizing compilation process to application.
5. Apply the various code optimization techniques.
6. Use the compiler construction tools to perform predictable optimization.

Prepared by
STAFF NAME
G.MEGALA

Verified By
HOD

Approved by
PRINCIPAL
UNIT I - INTRODUCTION TO COMPILERS

1.1 Translator:
It is a program that translates one language to another.

![Diagram of Translator]

Types of Translator:
1. Interpreter
2. Compiler
3. Assembler

1. Interpreter:
It is one of the translators that translate high level language to low level language.

![Diagram of Interpreter]

During execution, it checks line by line for errors.
Example: Basic, Lower version of Pascal.

2. Assembler:
It translates assembly level language to machine code.

![Diagram of Assembler]

Example: Microprocessor 8085, 8086.

3. Compiler:
It is a program that translates one language (source code) to another language (target code).

![Diagram of Compiler]
It executes the whole program and then displays the errors. Example: C, C++, COBOL, higher version of Pascal.

1.2 Compilation & Interpretation:

**Difference between compiler and interpreter:**

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is a translator that translates high level to low level language</td>
<td>It is a translator that translates high level to low level language</td>
</tr>
<tr>
<td>It displays the errors after the whole program is executed.</td>
<td>It checks line by line for errors.</td>
</tr>
<tr>
<td>Examples: Basic, lower version of Pascal.</td>
<td>Examples: C, C++, Cobol, higher version of Pascal.</td>
</tr>
</tbody>
</table>

1.3 LANGUAGE PROCESSORS:

1.3.1 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
1.3.2 TRANSLATORS:

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 in HLL programming language is basically of two parts. The source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.
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 into machine language. The input to an assembler program is called source program, the output is a machine language translation (object program).

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 the following phases.
1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Direct Execution

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

1.3.3 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 programme 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 o they may be placed in arbitrary core locations is called relocation. Relocation loaders perform four functions.

1.3.4 Software tools used in Analysis part:

1) Structure editor:
   - Takes as input a sequence of commands to build a source program.
   - The structure editor not only performs the text-creation and modification functions of an ordinary text editor, but it also analyzes the program text, putting an appropriate hierarchical structure on the source program.
   - For example, it can supply key words automatically - while …. do and begin….. end.

2) Pretty printers:
   - A pretty printer analyzes a program and prints it in such a way that the structure of the program becomes clearly visible.
   - For example, comments may appear in a special font.

3) Static checkers:
   - A static checker reads a program, analyzes it, and attempts to discover potential bugs without running the program.
   - For example, a static checker may detect that parts of the source program can never be executed.

4) Interpreters:
   - Translates from high level language (BASIC, FORTRAN, etc..) into machine language.
   - An interpreter might build a syntax tree and then carry out the operations at the nodes as it walks the tree.
   - Interpreters are frequently used to execute command language since each operator executed in a command language is usually an invocation of a complex routine such as an editor or complier.
### 1.3.5 Analysis of the Source Program

Analysis consists of 3 phases:

**Linear/Lexical Analysis:**

- It is also called scanning. It is the process of reading the characters from left to right and grouping into tokens having a collective meaning.

- For example, in the assignment statement `a=b+c*2`, the characters would be grouped into the following tokens:
  1. The identifier `1` ‘a’
  2. The assignment symbol (=)
  3. The identifier `2` ‘b’
  4. The plus sign (+)
  5. The identifier `3` ‘c’
  6. The multiplication sign (*)
  7. The constant ‘2’

**Syntax Analysis:**

- It is called parsing or hierarchical analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize output.

- They are represented using a syntax tree as shown below:

```
        =
       / \
      a   +
     / \  /
   b   * 2
  /   /  /
 c   c  c
```

- A **syntax tree** is the tree generated as a result of syntax analysis in which the interior nodes are the operators and the exterior nodes are the operands.

- This analysis shows an error when the syntax is incorrect.

**Semantic Analysis:**

- It checks the source programs for semantic errors and gathers type information for the subsequent code generation phase. It uses the syntax tree to identify the operators and operands of statements.

- An important component of semantic analysis is **type checking**. Here the compiler checks that each operator has operands that are permitted by the source language specification.
1.4 THE PHASES OF COMPILER

A Compiler operates in phases, each of which transforms the source program from one representation into another. The following are the phases of the compiler:

Main phases:
1) Lexical analysis
2) Syntax analysis
3) Semantic analysis
4) Intermediate code generation
5) Code optimization
6) Code generation

Sub-Phases:
1) Symbol table management
2) Error handling

1.4.1 LEXICAL ANALYSIS:

- It is the first phase of the compiler. It gets input from the source program and produces tokens as output.
- It reads the characters one by one, starting from left to right and forms the tokens.
- **Token**: It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.
  Example: \( a + b = 20 \)
  Here, \( a, b, +, =, 20 \) are all separate tokens.
  Group of characters forming a token is called the **Lexeme**.
- The lexical analyser not only generates a token but also enters the lexeme into the symbol table if it is not already there.

1.4.2 SYNTAX ANALYSIS:
- It is the second phase of the compiler. It is also known as parser.
- It gets the token stream as input from the lexical analyser of the compiler and generates syntax tree as the output.
- Syntax tree:
  It is a tree in which interior nodes are operators and exterior nodes are operands.
- Example: For a=b+c*2, syntax tree is

  ![](syntax_tree.png)

1.4.3 SEMANTIC ANALYSIS:

- It is the third phase of the compiler.
- It gets input from the syntax analysis as parse tree and checks whether the given syntax is correct or not.
- It performs type conversion of all the data types into real data types.

1.4.4 INTERMEDIATE CODE GENERATION:

- It is the fourth phase of the compiler.
- It gets input from the semantic analysis and converts the input into output as intermediate code such as three-address code.
- The three-address code consists of a sequence of instructions, each of which has atmost three operands.
  Example: t1=t2+t3

1.4.5 CODE OPTIMIZATION:

- It is the fifth phase of the compiler.
- It gets the intermediate code as input and produces optimized intermediate code as output.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - deduction and removal of dead code (unreachable code).
  - calculation of constants in expressions and terms.
  - collapsing of repeated expression into temporary string.
  - loop unrolling.
  - moving code outside the loop.
  - removal of unwanted temporary variables.

1.4.6 CODE GENERATION:

- It is the final phase of the compiler.
- It gets input from code optimization phase and produces the target code or object code as
result.

- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - allocation of register and memory
  - generation of correct references
  - generation of correct data types
  - generation of missing code

1.4.7 SYMBOL TABLE MANAGEMENT:

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows to find the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

1.4.8 ERROR HANDLING:

- Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.
- In lexical analysis, errors occur in separation of tokens.
- In syntax analysis, errors occur during construction of syntax tree.
- In semantic analysis, errors occur when the compiler detects constructs with right syntactic structure but no meaning and during type conversion.
- In code optimization, errors occur when the result is affected by the optimization.
- In code generation, it shows error when code is missing etc.

To illustrate the translation of source code through each phase, consider the statement \( a = b + c \times 2 \).
The figure shows the representation of this statement after each phase:
a = b + c * 2

Lexical analyser

id1 = id2 + id3 * 2

Syntax analyser

id2 *

id3

id1 +

Semantic analyser

id1 +

id2 *

id3

Intermediate code generator

temp1 = inttoreal(2)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3

Code optimizer

temp1 = id3 * 2.0
id1 = id2 + temp1

Code generator

MOVFR3, R2
MULF #2.0, R2
MOVFR2, R1
ADDF R2, R1
MOVFR1, id1
1.5 ERRORS ENCOUNTERED IN DIFFERENT PHASES:

Compiler Errors

- Lexical errors (e.g. misspelled word)
- Syntax errors (e.g. unbalanced parentheses, missing semicolon)
- Semantic errors (e.g. type errors)
- Logical errors (e.g. infinite recursion)

Error Handling

- Report errors clearly and accurately
- Recover quickly if possible
- Poor error recover may lead to avalanche of errors

In Different Phases:

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

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

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

Panic mode

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

Statement mode

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

Error productions

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

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

1.6 GROUPING OF PHASES:

Compiler can be grouped into front and back ends:

- **Front end**: analysis (machine independent)
  These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code. It also includes error handling that goes along with each of these phases.

- **Back end**: synthesis (machine dependent)
  It includes code optimization phase and code generation along with the necessary error handling and symbol table operations.

Compiler passes

A collection of phases is done only once (single pass) or multiple times (multi pass)

- Single pass: usually requires everything to be defined before being used in source program.
- Multi pass: compiler may have to keep entire program representation in memory.

Several phases can be grouped into one single pass and the activities of these phases are interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass.

1.7 COMPILER CONSTRUCTION TOOLS

These are specialized tools that have been developed for helping implement various phases of a compiler. The following are the compiler construction tools:

1) **Parser Generators:**
   - These produce syntax analyzers, normally from input that is based on a context-free grammar.
   - It consumes a large fraction of the running time of a compiler.
   - Example- YACC (Yet Another Compiler-Compiler).

2) **Scanner Generator:**
   - These generate lexical analyzers, normally from a specification based on regular expressions.
   - The basic organization of lexical analyzers is based on finite automation.

3) **Syntax-Directed Translation:**
   - These produce routines that walk the parse tree and as a result generate intermediate code.
   - Each translation is defined in terms of translations at its neighbour nodes in the tree.
4) **Automatic Code Generators:**
- It takes a collection of rules to translate intermediate language into machine language. The rules must include sufficient details to handle different possible access methods for data.

5) **Data-Flow Engines:**
- It does code optimization using data-flow analysis, that is, the gathering of information about how values are transmitted from one part of a program to each other part.

1.8 Programming Language Basics:

**Static / dynamic distinction**

The scope of a name binding – an association of a name to an entity, such as a variable – is the part of a computer program where the binding is valid: where the name can be used to refer to the entity. In other parts of the program the name may refer to a different entity (it may have a different binding), or to nothing at all (it may be unbound).

The scope of a binding is also known as the visibility of an entity, particularly in older or more technical literature – this is from the perspective of the referenced entity, not the referencing name.

A scope is a part of a program that is or can be the scope for a set of bindings – a precise definition is tricky (see below), but in casual use and in practice largely corresponds to a block, a function, or a file, depending on language and type of entity.

**Lexical /Dynamic Scope:**

A fundamental distinction in scoping is what "part of a program" means. In languages with lexical scope (also called static scope), name resolution depends on the location in the source code and the lexical context, which is defined by where the named variable or function is defined.

In languages with dynamic scope the name resolution depends upon the program state when the name is encountered which is determined by the execution context or calling context. With lexical scope a variable's definition is resolved by searching its containing block or function, then if that fails searching the outer containing block, and so on, whereas with dynamic scope the calling function is searched, then the function which called that calling function, and so on.

Most modern languages use lexical scoping for variables and functions, though dynamic scoping is used in some languages, notably some dialects of Lisp, some "scripting" languages like Perl, and some template languages. Even in lexically scoped languages, scope for closures can be confusing to the uninitiated, as these depend on the lexical context where the closure is defined, not where it is called.

**Environments and States:**

A change in the program affects the values of data elements in the program.

Name -> location -> value.

The environment is a mapping from names to locations or mapping from names to variables. It can also be considered as l-value.

The state is defined as a mapping from locations to their values. It assigns the r-values to the l-values.

**Static scope and block structure:**

The scope of programming language such as C and its family use static scope. The scope rules for C asr based on the program structure where the scope is defined implicitly inside the declaration. The languages such as C++,JAVA and C# provide explicit control over the scope by using keywords like public,private and protected.

The blocks defined in programming languages,

C,C++,Java \{ and \}

Fortran, Algol \begin{center} \texttt{begin .. end}. \end{center}

**Dynamic scope:**

It refers to the fact of anything to be known only during execution. A use of name \( x \) refers to the declaration of \( x \) in the most recently called procedure containing the declaration of \( x \).
1) Macro expansion in the C preprocessor.
2) Method resolution in C++, Java.

For example, the macro,

```c
#define ADD_A(x) x + a
```

will expand to add a to the passed variable, with this identifier only later resolved by the compiler based on where the macro ADD_A is "called" (properly, expanded), is in dynamic scope, and is independent of where the macro is defined.

```c
#define ADD_A(x) x + a
void add_one(int *x) {
    const int a = 1;
    *x = ADD_A(*x);  
}
void add_two(int *x) {
    const int a = 2;
    *x = ADD_A(*x);
}
```

Parameter passing Mechanisms:

The procedure can be called in different ways as,
1) Call by value
2) Call by reference

**Call by value:**

Call-by-value evaluation is the most common evaluation strategy, used in languages. In call-by-value, the argument expression is evaluated, and the resulting value is bound to the corresponding variable in the function (frequently by copying the value into a new memory region). If the function or procedure is able to assign values to its parameters, only its local copy is assigned— that is, anything passed into a function call is unchanged in the caller's scope when the function returns.

**Call by Reference**

In call-by-reference evaluation (also referred to as pass-by-reference), a function receives an implicit reference to a variable used as argument, rather than a copy of its value. This typically means that the function can modify (i.e. assign to) the variable used as argument—something that will be seen by its caller.

Call-by-reference can therefore be used to provide an additional channel of communication between the called function and the calling function. A call-by-reference language makes it more difficult for a programmer to track the effects of a function call, and may introduce subtle bugs.

**Aliasing:** Here, two formal parameters can refer to the same location, where are aliases of one another. Thus any two variables which appear to take their values from two distinct formal parameters can be the aliases of each other.
UNIT II

LEXICAL ANALYSIS

Need and Role of Lexical Analyzer-Lexical Errors-Expressing Tokens by Regular Expressions- Converting Regular Expression to DFA- Minimization of DFA-Language for Specifying Lexical Analyzers-LEX -Design of Lexical Analyzer for a sample Language.

2.1 NEED AND THE ROLE OF THE LEXICAL ANALYZER

Lexical analysis is the process of converting a sequence of characters into a sequence of tokens. A program or function which performs lexical analysis is called a lexical analyzer or scanner. A lex often exists as a single function which is called by a parser or another function.

- The lexical analyzer is the first phase of a compiler.
- Its main task is to read the input characters and produce as output a sequence of tokens that the parser uses for syntax analysis.

Upon receiving a “get next token” command from the parser, the lexical analyzer reads input characters until it can identify the next token.

ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:
- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

2.1.1 TOKENS

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

A token can look like anything that is useful for processing an input text stream or text file. Consider this expression in the C programming language: sum=3+2;

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token type</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>Identifier</td>
</tr>
<tr>
<td>=</td>
<td>Assignment operator</td>
</tr>
<tr>
<td>3</td>
<td>Number</td>
</tr>
<tr>
<td>+</td>
<td>Addition operator</td>
</tr>
<tr>
<td>2</td>
<td>Number</td>
</tr>
<tr>
<td>;</td>
<td>End of statement</td>
</tr>
</tbody>
</table>
LEXEME:
Collection or group of characters forming tokens is called Lexeme.

PATTERN:
A pattern is a description of the form that the lexemes of a token may take.

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

2.1.2 Attributes for Tokens

Some tokens have attributes that can be passed back to the parser. The lexical analyzer collects information about tokens into their associated attributes. The attributes influence the translation of tokens.

i) Constant: value of the constant
ii) Identifiers: pointer to the corresponding symbol table entry.

2.2 Lexical Errors:
The following are the error-recovery actions in lexical analysis:

1) Deleting an extraneous character.

2) Inserting a missing character.

3) Replacing an incorrect character by a correct character.

4) Transforming two adjacent characters.

5) Panic mode recovery: Deletion of successive characters from the token until error is resolved.

2.2.1 INPUT BUFFERING
We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:

```
   A = B + C
```

We introduce a two-buffer scheme that handles large look aheads safely. We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.
BUFFER PAIRS

- A buffer is divided into two N-character halves, as shown below

```
::E::=::M:*  C::*::*:2.eof
```

lexeme_beginning

forward

- Each buffer is of the same size N, and N is usually the number of characters on one disk block. E.g., 1024 or 4096 bytes.
- Using one system read command we can read N characters into a buffer.
- If fewer than N characters remain in the input file, then a special character, represented by `eof`, marks the end of the source file.
- Two pointers to the input are maintained:
  1. Pointer `lexeme_beginning`, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  2. Pointer `forward` scans ahead until a pattern match is found.
    Once the next lexeme is determined, `forward` is set to the character at its right end.
- The string of characters between the two pointers is the current lexeme.
  After the lexeme is recorded as an attribute value of a token returned to the parser, `lexeme_beginning` is set to the character immediately after the lexeme just found.

2.2.2 Advancing forward pointer:

Advancing forward pointer requires that we first test whether we have reached the end of one of the buffers, and if so, we must reload the other buffer from the input, and move forward to the beginning of the newly loaded buffer. If the end of second buffer is reached, we must again reload the first buffer with input and the pointer wraps to the beginning of the buffer.

Code to advance forward pointer:

```plaintext
if forward at end of first half then
  begin reload second half;
  forward := forward + 1
end
else if forward at end of second half then
  begin reload second half;
  move forward to beginning of first half
end
else forward := forward + 1;
```

2.2.3 SENTINELS

- For each character read, we make two tests: one for the end of the buffer, and one to determine what character is read. We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character `eof`. 
The sentinel arrangement is as shown below:

```
:: E :: = :: M :: * :: eof  C :: * :: 2 :: eof :: eof
```

Note that `eof` retains its use as a marker for the end of the entire input. Any `eof` that appears other than at the end of a buffer means that the input is at an end.

**Code to advance forward pointer:**

```
forward := forward + 1;
if forward ↑ = eof then begin
  if forward at end of first half then
    begin reload second half;
    forward := forward + 1
  end
  else if forward at end of second half then
    begin reload first half;
    move forward to beginning of
    first half end
  else /* `eof` within a buffer signifying end of input */
    terminate lexical analysis
end
```

### 2.3 SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

1. **Strings**
2. **Language**
3. **Regular expression**

#### Strings and Languages

An *alphabet* or character class is a finite set of symbols.

A *string* over an alphabet is a finite sequence of symbols drawn from that alphabet.

A *language* is any countable set of strings over some fixed alphabet.

In language theory, the terms "sentence" and "word" are often used as synonyms for "string." The length of a string $s$, usually written $|s|$, is the number of occurrences of symbols in $s$. For example, banana is a string of length six. The empty string, denoted $\epsilon$, is the string of length zero.

#### Operations on strings

The following string-related terms are commonly used:

1. A *prefix* of string $s$ is any string obtained by removing zero or more symbols from the end of string $s$. 

M.I.E.T./CSE/III Yr/Compiler Design
For example, ban is a prefix of banana.

2. A **suffix** of string $s$ is any string obtained by removing zero or more symbols from the beginning of $s$.
   For example, nana is a suffix of banana.

3. A **substring** of $s$ is obtained by deleting any prefix and any suffix from $s$.
   For example, nan is a substring of banana.

4. The **proper prefixes, suffixes, and substrings** of a string $s$ are those prefixes, suffixes, and substrings, respectively of $s$ that are not $\varepsilon$ or not equal to $s$ itself.

5. A subsequence of $s$ is any string formed by deleting zero or more not necessarily consecutive positions of $s$.
   For example, baan is a subsequence of banana.

**Operations on languages:**

The following are the operations that can be applied to languages:

1. Union
2. Concatenation
3. Kleene closure
4. Positive closure

The following example shows the operations on strings:

Let $L = \{0,1\}$ and $S = \{a,b,c\}$

1. Union : $L \cup S = \{0,1,a,b,c\}$
2. Concatenation : $L \cdot S = \{0a,1a,0b,1b,0c,1c\}$
3. Kleene closure : $L^* = \{\varepsilon,0,1,00,\ldots\}$
4. Positive closure : $L^+ = \{0,1,00,\ldots\}$

**2.3.1 Regular Expressions**

Each regular expression $r$ denotes a language $L(r)$.

Here are the rules that define the regular expressions over some alphabet $\Sigma$ and the languages that those expressions denote:

1. $\varepsilon$ is a regular expression, and $L(\varepsilon) = \{\varepsilon\}$, that is, the language whose sole member is the empty string.

2. If ‘$a$’ is a symbol in $\Sigma$, then ‘$a$’ is a regular expression, and $L(a) = \{a\}$, that is, the language with one string, of length one, with ‘$a$’ in its one position.

3. Suppose $r$ and $s$ are regular expressions denoting the languages $L(r)$ and $L(s)$. Then,

   a) $(r)\mid(s)$ is a regular expression denoting the language $L(r) \cup L(s)$.
   b) $(r)(s)$ is a regular expression denoting the language $L(r)L(s)$.
   c) $(r)^*$ is a regular expression denoting $(L(r))^*$.
   d) $(r)$ is a regular expression denoting $L(r)$.

4. The unary operator * has highest precedence and is left associative.
5. Concatenation has second highest precedence and is left associative.

6. | has lowest precedence and is left associative.

**Regular set**

A language that can be defined by a regular expression is called a regular set.

If two regular expressions r and s denote the same regular set, we say they are equivalent and write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.

For instance, r|s = s|r is commutative; r|(s|t)=(r| s) |t is associative.

**Regular Definitions**

Giving names to regular expressions is referred to as a Regular definition. If \( \Sigma \) is an alphabet of basic symbols, then a regular definition is a sequence of definitions of the form

\[
d_1 \rightarrow r_1 \\
d_2 \rightarrow r_2 \\
\ldots \\
d_n \rightarrow r_n
\]

1. Each \( d_i \) is a distinct name.
2. Each \( r_i \) is a regular expression over the alphabet \( \Sigma \cup \{ d_1, d_2, \ldots, d_i \} \).

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:

\[
\text{letter} \rightarrow A | B | \ldots | Z | a | b | \ldots | z \\
\text{digit} \rightarrow 0 | 1 | \ldots | 9 \\
\text{id} \rightarrow \text{letter} (\text{letter} | \text{digit})^* 
\]

**2.3.2 Shorthands**

Certain constructs occur so frequently in regular expressions that it is convenient to introduce notational shorthands for them.

1. **One or more instances (+):**

   - The unary postfix operator + means “one or more instances of”.

   - If \( r \) is a regular expression that denotes the language \( L(r) \), then \( (r)^+ \) is a regular expression that denotes the language \( (L(r))^+ \)

   - Thus the regular expression \( a^+ \) denotes the set of all strings of one or more a’s.

   - The operator \( ^+ \) has the same precedence and associativity as the operator \( ^* \).

2. **Zero or one instance ( ?):**

   - The unary postfix operator ? means “zero or one instance of”.

   - The notation \( r? \) is a shorthand for \( r | \varepsilon \).

   - If ‘r’ is a regular expression, then \( (r)? \) is a regular expression that denotes the language \( L( r ) \cup \{ \varepsilon \} \).

3. **Character Classes:**
- The notation \([abc]\) where \(a, b\) and \(c\) are alphabet symbols denotes the regular expression \(a | b | c\).

- Character class such as \([a–z]\) denotes the regular expression \(a | b | c | d | … | z\).

- We can describe identifiers as being strings generated by the regular expression, 
  \([A–Za–z][A–Za–z0–9]^{*}\)

**Non-regular Set**

A language which cannot be described by any regular expression is a non-regular set. Example: The set of all strings of balanced parentheses and repeating strings cannot be described by a regular expression. This set can be specified by a context-free grammar.

### 2.3.3 RECOGNITION OF TOKENS

Consider the following grammar fragment:

\[
\begin{align*}
\text{stmt} &\rightarrow \text{if expr then stmt} \\
&\quad | \text{if expr then stmt else stmt} \\
&\quad | \epsilon \\
\text{expr} &\rightarrow \text{term relop term} \\
&\quad | \text{term} \\
\text{term} &\rightarrow \text{id} \\
&\quad | \text{num}
\end{align*}
\]

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

\[
\begin{align*}
\text{if} &\rightarrow \text{if} \\
\text{then} &\rightarrow \text{then} \\
\text{else} &\rightarrow \text{else} \\
\text{relop} &\rightarrow \text{id} \rightarrow \text{num} \\
\text{num} &\rightarrow \langle |\leq|=|\geq|\rangle \text{=} \\
&\quad \text{letter(letter|digit)*} \\
&\quad \text{digit*}(.\text{digit*})?(E(\text{|-})?\text{digit*})?
\end{align*}
\]

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

**Transition diagrams**

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

Finite Automata is one of the mathematical models that consist of a number of states and edges. It is a transition diagram that recognizes a regular expression or grammar.

Types of Finite Automata

There are two types of Finite Automata:

- Non-deterministic Finite Automata (NFA)
- Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by $M = \{Q_n, \Sigma, \delta, q_0, f_n\}$

- $Q_n$ – finite set of states
- $\Sigma$ – finite set of input symbols
- $\delta$ – transition function that maps state-symbol pairs to set of states
- $q_0$ – starting state $f_n$ – final state

Deterministic Finite Automata

DFA is a special case of a NFA in which:

- no state has an $\epsilon$-transition.
- there is at most one transition from each state on any input.
DFA has five tuples denoted by
\( M = \{Q_d, \Sigma, \delta, q_0, f_d\}\)

\( Q_d \) – finite set of states
\( \Sigma \) – finite set of input symbols
\( \delta \) – transition function that maps state-symbol pairs to set of states
\( q_0 \) – starting state
\( f_d \) – final state

2.4 Construction of DFA from regular expression

The following steps are involved in the construction of DFA from regular expression:
i) Convert RE to NFA using Thomson’s rules
ii) Convert NFA to DFA
iii) Construct minimized DFA

2.4.1 Constructing an NFA from a RE : Thompson’s Construction Algorithm

- Input: A regular expression \( r \) over an alphabet
- Output: An NFA \( N \) accepting \( L(r) \)
  - First \( r \) is parsed into its constituent sub-expressions
  - Then NFA is constructed for each of the basic symbols
  - If same symbol occurs repeatedly, separate NFA is constructed for each occurrence

1. For \( \epsilon \) in the regular expression, construct NFA

2. For \( a \in \Sigma \) in the regular expression, construct NFA

3(a) If \( s, t \) are regular expressions, \( N(s), N(t) \) their NFAs \( s | t \) has NFA:

where \( i \) and \( f \) are new start/final states, and \( \epsilon \)-moves are introduced from \( i \) to the old start states of \( N(s) \) and \( N(t) \) as well as from all of their final states to \( f \).
3.(b) If $s$, $t$ are regular expressions, $N(s)$, $N(t)$ their NFAs
st (concatenation) has NFA:

\[
\begin{array}{c}
\text{start} \\
\text{i} \quad \text{N(s)} \\
\text{f} \\
\text{L(s) L(t)}
\end{array}
\]

Alternative:

\[
\begin{array}{c}
\text{start} \\
\text{i} \quad \epsilon \quad \text{N(s)} \\
\text{f} \\
\end{array}
\]

where $i$ is the start state of $N(s)$ (or new under the
alternative) and $f$ is the final state of $N(t)$ (or new).
Overlap maps final states of $N(s)$ to start state of $N(t)$.

Example - $ab^*c | a(b|c^*)$

Parse Tree for this regular expression:

\[
\begin{array}{c}
\text{r}_{13} \\
\text{r}_{12} \\
\text{r}_{11} \\
\text{r}_{10} \\
\text{r}_{9} \\
\text{r}_{8} \\
\text{r}_{7} \\
\text{r}_{6} \\
\text{r}_{5} \\
\text{r}_{4} \\
\text{r}_{3} \\
\text{r}_{2} \\
\text{r}_{1} \\
\text{r}_{0}
\end{array}
\]

\[
\begin{align*}
\text{r}_{0}: & \quad \text{b} \\
\text{r}_{3}: & \quad \text{a} \\
\text{r}_{2}: & \quad \text{c} \\
\text{r}_{4}: & \quad \text{r}_{1} \quad \text{r}_{2} \\
\text{r}_{5}: & \quad \text{r}_{3} \quad \text{r}_{4}
\end{align*}
\]
2.4.2 Conversion of NFA to DFA

1. Algorithm Constructs a Transition Table for DFA from NFA
2. Each state in DFA corresponds to a set of states of the NFA
3. DFA state keeps track of set of all states NFA can be in after reading each input symbol
4. After reading $a_1a_2...a_n$, DFA is in a state that represents a subset $T$ of NFA states reachable from start states
   a. $\epsilon$ moves
   b. non-determinism
   c. Handling Ambiguity

Input: NFA $N$; Output: DFA $D$
initialize $D$states to unmarked $\epsilon$-closure($s_0$)
while there is an unmarked state $T$ in $D$states
    mark $T$
    for each input symbol $a$
      $U := \epsilon$-closure(move($T$, $a$))
      if $U$ is not in $D$states
          add $U$ as unmarked state to $D$states
      $D$tran[$T$, $a$] := $U$
2.5 DFA Minimization

- Dead state for undefined transitions
- String w distinguishes state s from state t if
  - by starting from s and giving w – accepting state results;
  - by starting from t and giving w – non-accepting state results;
- Find all groups of states that can be distinguished by some input string
- Merge all states that cannot be distinguished
- Partition states into two groups:
  - Acceptance states and others
  - From a given group transitions must be to same group – if not partition
  - Continue until no further partitions are possible

1. Construct initial partition $\Pi$ of S with two groups: accepting/ non-accepting.
2. (Construct $\Pi_{\text{new}}$ ) for each group G of $\Pi$ do begin
   - Partition G into subgroups such that two states s, t of G are in the same subgroup iff for all symbols a state’s s, t have transitions on a to states of the same group of $\Pi$.
   - Replace G in $\Pi_{\text{new}}$ by the set of all these subgroups.
3. Compare $\Pi_{\text{new}}$ and $\Pi$. If equal, $\Pi_{\text{final}} := \Pi$ then proceed to 4, else set $\Pi := \Pi_{\text{new}}$ and goto 2.
4. Aggregate states belonging in the groups of $\Pi_{\text{final}}$
5. If $M'$ has a dead state d from $M'$. Also remove any states not reachable from the start state. Any transitions to d from other states become undefined.
2.5.1 Converting Regular Expression to DFA

- A regular expression can be converted into a DFA (without creating a NFA first).
- First the given regular expression is augmented by concatenating it with a special symbol 
  #.
  - $r \Rightarrow (r)^#$
  - augmented regular expression
- Then a syntax tree is created for this augmented regular expression.
- In this syntax tree, all alphabet symbols, #, and the empty string in the augmented regular
  expression will be on the leaves, and
- All inner nodes will be the operators
- Then each alphabet symbol and # will be numbered (position numbers).
Types of Interior nodes

- Cat-node
- Star-node
- Or-node

Regular Expression $\rightarrow$ DFA

$$(a|b)^* \ a \rightarrow (a|b)^* \ a \ #$$

**augmented regular expression**

Syntax tree of $(a|b)^* \ a \ #$

- each symbol is numbered
- each symbol is at a leaf
- inner nodes are operators

Followpos:

Followpos is defined for the positions (positions assigned to leaves).

- $\text{followpos}(i)$ is the set of positions which can follow the position $i$ in the strings generated by the augmented regular expression.

For example,

- $(a|b)^* \ a \ #$
- $1 \ 2 \ 3 \ 4$
- $\text{followpos}(1) = \{1,2,3\}$
- $\text{followpos}(2) = \{1,2,3\}$
- $\text{followpos}(3) = \{4\}$
- $\text{followpos}(4) = \{\}$

**firstpos, lastpos, nullable**

To evaluate followpos, three more functions are to be defined for the nodes (not just for leaves) of the syntax tree.

- $\text{firstpos}(n)$ -- the set of the positions of the first symbols of strings generated by the sub-expression rooted at $n$.
- $\text{lastpos}(n)$ -- the set of the positions of the last symbols of strings generated by the sub-expression rooted at $n$.
- $\text{nullable}(n)$ -- true if the empty string is a member of strings generated by the sub-expression rooted by $n$ false otherwise
2.5.2 Evaluation of firstpos, lastpos, nullable

<table>
<thead>
<tr>
<th>N</th>
<th>nullable(n)</th>
<th>firstpos(n)</th>
<th>lastpos(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>leaf labeled $\epsilon$</td>
<td>true</td>
<td>$\phi$</td>
<td>$\phi$</td>
</tr>
<tr>
<td>leaf labeled with position i</td>
<td>false</td>
<td>{i}</td>
<td>{i}</td>
</tr>
<tr>
<td>$c_1 \ c_2$</td>
<td>nullable($c_1$) or nullable($c_2$)</td>
<td>firstpos($c_1$) $\cup$ firstpos($c_2$)</td>
<td>lastpos($c_1$) $\cup$ lastpos($c_2$)</td>
</tr>
<tr>
<td>$\cdot$ $c_1 \ c_2$</td>
<td>nullable($c_1$) and nullable($c_2$)</td>
<td>if (nullable($c_1$)) firstpos($c_1$) $\cup$ firstpos($c_2$) else firstpos($c_1$)</td>
<td>if (nullable($c_2$)) lastpos($c_1$) $\cup$ lastpos($c_2$) else lastpos($c_2$)</td>
</tr>
<tr>
<td>$\ast$ $c_1$</td>
<td>true</td>
<td>firstpos($c_1$)</td>
<td>lastpos($c_1$)</td>
</tr>
</tbody>
</table>

Evaluation of followpos

Two-rules define the function followpos:
1. If $n$ is concatenation-node with left child $c_1$ and right child $c_2$, and $i$ is a position in lastpos($c_1$), then all positions in firstpos($c_2$) are in followpos($i$).
2. If $n$ is a star-node, and $i$ is a position in lastpos($n$), then all positions in firstpos($n$) are in followpos($i$).

If firstpos and lastpos have been computed for each node, followpos of each position can be computed by making one depth-first traversal of the syntax tree.

For cat node,

For star node,
Example -- ( a | b )* a #

Algorithm (RE \( \Rightarrow \) DFA)

- Create the syntax tree of (r) #
- Calculate the functions: followpos, firstpos, lastpos, nullable
- Put firstpos(root) into the states of DFA as an unmarked state.
- while (there is an unmarked state S in the states of DFA) do
  - mark S
  - for each input symbol a do
    - let \( s_1, ..., s_n \) are positions in S and symbols in those positions are a
    - \( S' \leftarrow \text{followpos}(s_1) \cup ... \cup \text{followpos}(s_n) \)
    - move(S,a) \( \leftarrow S' \)
    - if (\( S' \) is not empty and not in the states of DFA)
      - put \( S' \) into the states of DFA as an unmarked state.
- the start state of DFA is firstpos(root)
- the accepting states of DFA are all states containing the position of #

Example -- ( a | b )* a #
2.6 A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER:

There is a wide range of tools for constructing lexical analyzers.

- Lex
- YACC

2.6.1 LEX

Lex is a computer program that generates lexical analyzers. Lex is commonly used with the yacc parser generator.

Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.
- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.

2.6.2 Design of Lexical Analyzer for a sample language

Lex Specification

A Lex program consists of three parts:

```plaintext
{ definitions }
%%
{ rules }
%%
{ user subroutines }

➢ Definitions include declarations of variables, constants, and regular definitions
➢ Rules are statements of the form

\[ p_1 \ \{ \text{action}_1 \} \]
\[ p_2 \ \{ \text{action}_2 \} \]
\[ \ldots \]
\[ p_n \ \{ \text{action}_n \} \]

where \( p_i \) is regular expression and \( \text{action}_i \) describes what action the lexical analyzer should take when pattern \( p_i \) matches a lexeme. Actions are written in C code.
User subroutines are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

Here is a list of Lex operators with examples:

<table>
<thead>
<tr>
<th>Operator notation</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>* (astersk)</td>
<td>a*</td>
<td>Set of all strings of zero or more a's, i.e. (empty a, aa, aaa ... )</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l (or)</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>+</td>
<td>a+</td>
<td>One or more instances of a i.e. a, aa, aaa etc.</td>
</tr>
<tr>
<td>?</td>
<td>a?</td>
<td>Zero one of instance of a</td>
</tr>
<tr>
<td>[]</td>
<td>[a b c]</td>
<td>denotes the regular expression a</td>
</tr>
</tbody>
</table>

Here are few more examples of Lex operators:

- (a | b) (a | h) denotes (aa, ab, ba, bb), the set of an strings of a's and b's of length two. Another regular expression for this same set is aa | ab | ba | bb

(a b)* denotes the set of all strings containing zero or more instances of an a or b, that is the set of all strings of a's and b's.

X | X'YZ denotes the set containing the string X and all strings consisting of zero or more X's followed by YZ.

Given a familiarity with Lex and C, a lexical analyzer for almost any programming language can be written in a very short time. Lex does not have application solely in the generator of lexical analysers. It can also be used to assist in the implementation of almost any text pattern matching application, such as text editing, code conversion and so on.
UNIT III SYNTAX ANALYSIS
Need and Role of the Parser-Context Free Grammars-Top Down Parsing-General Strategies-Recursive Descent Parser-Predictive Parser-LL(1) Parser-Shift Reduce Parser-LR Parser-LR (0) Item-Construction of SLR Parsing Table-Introduction to LALR Parser-
Error Handling and Recovery in Syntax Analyzer-YACC-Design of a syntax Analyzer for a Sample Language.

SYNTAX ANALYSIS

Syntax analysis is the second phase of the compiler. It gets the input from the tokens and generates a syntax tree or parse tree.

Advantages of grammar for syntactic specification:

1. A grammar gives a precise and easy-to-understand syntactic specification of a programming language.
2. An efficient parser can be constructed automatically from a properly designed grammar.
3. A grammar imparts a structure to a source program that is useful for its translation into object code and for the detection of errors.
4. New constructs can be added to a language more easily when there is a grammatical description of the language.

3.1 NEED AND THE ROLE OF PARSER

The parser or syntactic analyzer obtains a string of tokens from the lexical analyzer and verifies that the string can be generated by the grammar for the source language. It reports any syntax errors in the program. It also recovers from commonly occurring errors so that it can continue processing its input.

Functions of the parser:

1. It verifies the structure generated by the tokens based on the grammar.
2. It constructs the parse tree.
3. It reports the errors.
4. It performs error recovery.

Issues:

Parser cannot detect errors such as:
1. Variable re-declaration
2. Variable initialization before use.
3. Data type mismatch for an operation.

The above issues are handled by Semantic Analysis phase.

**Syntax error handling:**

Programs can contain errors at many different levels. For example:
1. Lexical, such as misspelling a keyword.
2. Syntactic, such as an arithmetic expression with unbalanced parentheses.
3. Semantic, such as an operator applied to an incompatible operand.
4. Logical, such as an infinitely recursive call.

Functions of error handler:

1. It should report the presence of errors clearly and accurately.
2. It should recover from each error quickly enough to be able to detect subsequent errors.
3. It should not significantly slow down the processing of correct programs.

**Error recovery strategies:**

The different strategies that a parse uses to recover from a syntactic error are:

1. Panic mode
2. Phrase level
3. Error productions
4. Global correction

**Panic mode recovery:**

On discovering an error, the parser discards input symbols one at a time until a synchronizing token is found. The synchronizing tokens are usually delimiters, such as semicolon or **end**. It has the advantage of simplicity and does not go into an infinite loop. When multiple errors in the same statement are rare, this method is quite useful.

**Phrase level recovery:**

On discovering an error, the parser performs local correction on the remaining input that allows it to continue. Example: Insert a missing semicolon or delete an extraneous semicolon etc.

**Error productions:**

The parser is constructed using augmented grammar with error productions. If an error production is used by the parser, appropriate error diagnostics can be generated to indicate the erroneous constructs recognized by the input.

**Global correction:**

Given an incorrect input string x and grammar G, certain algorithms can be used to find a
parse tree for a string y, such that the number of insertions, deletions and changes of tokens is as small as possible. However, these methods are in general too costly in terms of time and space.

3.2 CONTEXT-FREE GRAMMARS

A Context-Free Grammar is a quadruple that consists of terminals, non-terminals, start symbol and productions.

**Terminals**: These are the basic symbols from which strings are formed.

**Non-Terminals**: These are the syntactic variables that denote a set of strings. These help to define the language generated by the grammar.

**Start Symbol**: One non-terminal in the grammar is denoted as the “Start-symbol” and the set of strings it denotes is the language defined by the grammar.

**Productions**: It specifies the manner in which terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal, followed by an arrow, followed by a string of non-terminals and terminals.

**Example of context-free grammar**: The following grammar defines simple arithmetic expressions:

```plaintext
expr → expr op expr
expr → (expr)
expr → - expr
expr → id
   op → +
   op → -
   op → *
   op → /
   op → ↑
```

In this grammar,

- `id + - * / ↑ ( )` are terminals.
- `expr , op` are non-terminals.
- `expr` is the start symbol.
- Each line is a production.

**Derivations**:

Two basic requirements for a grammar are:

1. To generate a valid string.
2. To recognize a valid string.
Derivation is a process that generates a valid string with the help of grammar by replacing the non-terminals on the left with the string on the right side of the production.

Example: Consider the following grammar for arithmetic expressions:

\[ E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id \]

To generate a valid string - (id+id) from the grammar the steps are

1. \[ E \rightarrow -E \]
2. \[ E \rightarrow -(E) \]
3. \[ E \rightarrow -(E+E) \]
4. \[ E \rightarrow -(id+E) \]
5. \[ E \rightarrow -(id+id) \]

In the above derivation,

- \[ E \] is the start symbol.
- \[ -(id+id) \] is the required sentence (only terminals).
- Strings such as \[ E, -E, -(E), \ldots \] are called sentinel forms.

Types of derivations:

The two types of derivation are:

1. Left most derivation
2. Right most derivation.

- In leftmost derivations, the leftmost non-terminal in each sentinel is always chosen first for replacement.
- In rightmost derivations, the rightmost non-terminal in each sentinel is always chosen first for replacement.

Example:

Given grammar \( G : E \rightarrow E+E \mid E*E \mid (E) \mid -E \mid id \)

Sentence to be derived: \(- (id+id)\)

<table>
<thead>
<tr>
<th>LEFTMOST DERIVATION</th>
<th>RIGHTMOST DERIVATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ E \rightarrow -E ]</td>
<td>[ E \rightarrow -E ]</td>
</tr>
<tr>
<td>[ E \rightarrow -(E) ]</td>
<td>[ E \rightarrow -(E) ]</td>
</tr>
<tr>
<td>[ E \rightarrow -(E+E) ]</td>
<td>[ E \rightarrow -(E+E) ]</td>
</tr>
<tr>
<td>[ E \rightarrow -(id+E) ]</td>
<td>[ E \rightarrow -(id+E) ]</td>
</tr>
<tr>
<td>[ E \rightarrow -(id+id) ]</td>
<td>[ E \rightarrow -(id+id) ]</td>
</tr>
</tbody>
</table>
Sentinels:

Given a grammar $G$ with start symbol $S$, if $S \rightarrow \alpha$, where $\alpha$ may contain non-terminals or terminals, then $\alpha$ is called the sentinel form of $G$.

Yield or frontier of tree:

Each interior node of a parse tree is a non-terminal. The children of node can be a terminal or non-terminal of the sentinel forms that are read from left to right. The sentinel form in the parse tree is called yield or frontier of the tree.

Ambiguity:

A grammar that produces more than one parse for some sentence is said to be ambiguous grammar.

Example: Given grammar $G : E \rightarrow E + E \mid E^*E \mid (E) \mid -E \mid id$

The sentence $id+id*id$ has the following two distinct leftmost derivations:

- $E \rightarrow E + E$
- $E \rightarrow id + E$
- $E \rightarrow id + E * E$
- $E \rightarrow id + id * E$
- $E \rightarrow id + id * id$

The two corresponding parse trees are:

```
     E
    /|
   /|
  E + E
 /|
id E * E
 /|
 id id
```

```
     E
    /|
   /|
  E * E
 /|
 id id
```

```
     E
    /|
   /|
  E + E
 /|
 id id
```

```
     E
    /|
   /|
  E + E
 /|
 id id
```
WRITING A GRAMMAR

There are four categories in writing a grammar:

1. Regular Expression Vs Context Free Grammar
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.

3.2.2 Regular Expressions vs. Context-Free Grammars:

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

- The lexical rules of a language are simple and RE is used to describe them.
- Regular expressions provide a more concise and easier to understand notation for tokens than grammars.
- Efficient lexical analyzers can be constructed automatically from RE than from grammars.
- 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.

3.2.3 Eliminating ambiguity:

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 $E_1$ then if $E_2$ then $S_1$ else $S_2$ has the following two parse trees for leftmost derivation:
To eliminate ambiguity, the following grammar may be used:

\[
\text{stmt} \rightarrow \text{matched_stmt} | \text{unmatched_stmt}
\]

\[
\text{matched_stmt} \rightarrow \text{if expr then matched_stmt else matched_stmt} | \text{other}
\]

\[
\text{unmatched_stmt} \rightarrow \text{if expr then stmt} | \text{if expr then matched_stmt else unmatched_stmt}
\]

3.2.4 Eliminating Left Recursion:

A grammar is said to be left recursive if it has a non-terminal A such that there is a derivation \(A \Rightarrow A\alpha\) for some string \(\alpha\). Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production \(A \rightarrow A\alpha | \beta\) it can be replaced with a sequence of two productions

\[
A \rightarrow \beta A' \\
A' \rightarrow \alpha A' \mid \epsilon
\]

without changing the set of strings derivable from A.

**Example** : Consider the following grammar for arithmetic expressions: \(E\)

\[
\rightarrow E + T | T \\
T \rightarrow T * F | F F \\
\rightarrow (E) | \text{id}
\]

First eliminate the left recursion for \(E\) as

\(E \rightarrow TE'\)
E’ → +TE’ | ε

Then eliminate for T as

T → FT’ T’ →

*FT’ | ε

Thus the obtained grammar after eliminating left recursion is

E → TE’

E’ → +TE’ | ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

Algorithm to eliminate left recursion:

1. Arrange the non-terminals in some order A_1, A_2, ... A_n.
2. for i := 1 to n do begin
   for j := 1 to i-1 do begin
      replace each production of the form A_i → A_j γ
      by the productions A_i → δ_1 γ | δ_2 γ | ... | δ_k γ
      where A_j → δ_1 | δ_2 | ... | δ_k are all the current A_j-productions;
   end
end

eliminate the immediate left recursion among the A_i-productions

3.2.5 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.

If there is any production A → αβ_1 | αβ_2 , it can be rewritten as

A → αA’

A’ → β_1 | β_2

Consider the grammar , G : S → iEtS | iEtSeS | a
    E → b

Left factored, this grammar becomes
3.3 PARSING

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

Parse tree:

Graphical representation of a derivation or deduction is called a parse tree. Each interior node of the parse tree is a non-terminal; the children of the node can be terminals or non-terminals.

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.

3.3.1 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

3.3.1.1. RECURSIVE DESCENT PARSING

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

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:

**Step 1**
Initially create a tree with a single node labeled S. An input pointer points to ‘c’, the first symbol of w. Expand the tree with the production of S.

```
S
```

```
c A d
```

**Step 2**
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.

```
S
```

```
c A d
```

```
a b
```

**Step 3**
The second symbol ‘a’ of w also matches with the second leaf of the tree. So advance the input pointer to the third symbol of w ‘d’. But the third leaf of the tree is b which does not match with the input symbol d.

Hence discard the chosen production and reset the pointer to the second position. This is called **backtracking**.

**Step 4:**
Now try the second alternative for A.

```
S
```

```
  / \
/c  A \d
```

```
a
```

Now we can halt and announce the successful completion of parsing.
Example for recursive decent parsing:

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

Consider the grammar for arithmetic expressions

\[
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T * F \mid F \\
F & \rightarrow (E) \mid \text{id}
\end{align*}
\]

After eliminating the left-recursion the grammar becomes,

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \mid \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' \mid \varepsilon \\
F & \rightarrow (E) \mid \text{id}
\end{align*}
\]

Now we can write the procedure for grammar as follows:

**Recursive procedure:**

\[
\begin{align*}
\text{E()} & \\
\text{begin} & \\
\quad & T(); \\
\quad & \text{EPRIME();}
\text{end}
\end{align*}
\]
Procedure EPRIME()
begin
  If input_symbol='+' then
    ADVANCE();
    T();
    EPRIME();
end

Procedure T()
begin
  F();
  TPRIME();
end

Procedure TPRIME()
begin
  If input_symbol='*' then
    ADVANCE();
    F();
    TPRIME();
end

Procedure F()
begin
  If input-symbol='id' then
    ADVANCE();
  else if input-symbol='(' then
    ADVANCE();
    E();
  else if input-symbol=')' then
    ADVANCE();
  else
    ERROR();
end
else
  ERROR();

Stack implementation:
To recognize input id+id*id:
### 3.3.1.2. PREDICTIVE PARSING

- Predictive parsing is a special case of recursive descent parsing where no backtracking is required.

- The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.
The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

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

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

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

**Predictive parsing program:**
The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. 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 non-terminal, the program consults entry M[X, a] of the parsing table M. This entry will either be an X-production of the grammar or an error entry.
   - If M[X, a] = {X → UVW}, the parser replaces X on top of the stack by WVU.
   - If M[X, a] = error, the parser calls an error recovery routine.
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 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 as follows:

1. set ip to point to the first symbol of \( w \$ \);
2. repeat
   1. let \( X \) be the top stack symbol and \( a \) the symbol pointed to by ip;
   2. if \( X \) is a terminal or \( \$ \) then
      1. if \( X = a \) then
         1. pop \( X \) from the stack and advance ip
      2. else error()
   3. else /* \( X \) is a non-terminal */
      1. if \( M[X, a] = X \Rightarrow Y_2 \ldots Y_k \) then begin
         1. pop \( X \) from the stack;
         2. push \( Y_k, Y_{k-1}, \ldots, Y_1 \) onto the stack, with \( Y_1 \) on top;
         3. output the production \( X \Rightarrow Y_1 Y_2 \ldots Y_k \)
      end
   4. else error()
3. until \( X = \$ \) /* stack is empty */

3.3.1.2 Predictive parsing table construction:

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

1. **FIRST**
2. **FOLLOW**

**Rules for first()**: 

1. If \( X \) is terminal, then \( \text{FIRST}(X) \) is \( \{X\} \).
2. If \( X \rightarrow \epsilon \) is a production, then add \( \epsilon \) to \( \text{FIRST}(X) \).
3. If \( X \) is non-terminal and \( X \rightarrow a\alpha \) is a production then add \( a \) to \( \text{FIRST}(X) \).
4. If \( X \) is non-terminal and \( X \rightarrow Y_1 Y_2 \ldots Y_k \) is a production, then place \( a \) in \( \text{FIRST}(X) \) if for some \( i \), \( a \) is in \( \text{FIRST}(Y_i) \), and \( \epsilon \) is in all of \( \text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1}) \); that is, \( Y_1, \ldots, Y_{i-1} \Rightarrow \epsilon \). If \( \epsilon \) is
in FIRST(Y_j) for all j=1,2,...,k, then add $ \epsilon $ to FIRST(X).

**Rules for follow( ):**

1. If S is a start symbol, then FOLLOW(S) contains $ \$. 

2. If there is a production $ A \rightarrow \alpha B \beta $, then everything in FIRST(\beta) except $ \epsilon $ is placed in follow(B).

3. If there is a production $ A \rightarrow \alpha B $, or a production $ A \rightarrow \alpha B \beta $ where FIRST(\beta) contains $ \epsilon $, 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 \rightarrow \alpha $ of the grammar, do steps 2 and 3.

2. For each terminal a in FIRST(\alpha), add $ A \rightarrow \alpha $ to M[A, a].

3. If $ \epsilon $ is in FIRST(\alpha), add $ A \rightarrow \alpha $ to M[A, b] for each terminal b in FOLLOW(A). If $ \epsilon $ is in FIRST(\alpha) and $ \$ $ is in FOLLOW(A), add $ A \rightarrow \alpha $ to M[A, $ \$ $].

4. Make each undefined entry of M be error.

Example

Consider the following grammar :

$$
\begin{align*}
E & \rightarrow E + T \mid T \\
T & \rightarrow T \ast F \mid F \\
F & \rightarrow (E) \mid \text{id}
\end{align*}
$$

After eliminating left-recursion the grammar is

$$
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' \mid \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow \ast FT' \mid \epsilon \\
F & \rightarrow (E) \mid \text{id}
\end{align*}
$$
First( ):
FIRST(E) = { ( , id}
FIRST(E')={+ , }$
FIRST(T) = { ( , id}
FIRST(T') = {*, }$
FIRST(F) = { ( , id }

Follow( ):
FOLLOW(E) =  { $, ) }
FOLLOW(E') = { $, ) }
FOLLOW(T) = { +, $, ) }
FOLLOW(T') = { +, $, ) }
FOLLOW(F) = {+, *, $, ) }

Predictive parsing table :

<table>
<thead>
<tr>
<th>NON-TERMINAL</th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>E → TE’</td>
<td>E → TE’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E’</td>
<td>E’ → +TE’</td>
<td>E’ → ε</td>
<td>E’ → ε</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T → FT’</td>
<td>T → FT’</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T’</td>
<td>T’ → ε</td>
<td>T’ → *FT’</td>
<td>T’ → ε</td>
<td>T’ → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F → id</td>
<td>F → (E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stack implementation:

<table>
<thead>
<tr>
<th>stack</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id*id $</td>
<td></td>
</tr>
<tr>
<td>$E'T$</td>
<td>id+id*id $</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id+id*id $</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id+id*id $</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>+id*id $</td>
<td>$T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E'$</td>
<td>+id*id $</td>
<td>$E' \rightarrow +TE'$</td>
</tr>
<tr>
<td>$E'T$</td>
<td>id*id $</td>
<td></td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id*id $</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id*id $</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>*id $</td>
<td></td>
</tr>
<tr>
<td>$E'T'F*$</td>
<td>*id $</td>
<td>$T' \rightarrow *FT'$</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id $</td>
<td></td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id $</td>
<td>$F \rightarrow id$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$</td>
<td>$T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$$</td>
<td>$</td>
<td>$E' \rightarrow \epsilon$</td>
</tr>
</tbody>
</table>

3.4 LL(1) grammar:

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

Consider this following grammar:

\[
S \rightarrow iEtS \mid iEtSeS \mid a \\
E \rightarrow b
\]

After eliminating left factoring, we have

\[
S \rightarrow iEtSS' \mid a \\
S' \rightarrow eS \mid \epsilon \\
E \rightarrow b
\]

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

FIRST(S) = \{ i, a \}  \\
FIRST(S') = \{ e, \epsilon \}  \\
FIRST(E) = \{ b \}  \\
FOLLOW(S) = \{ $, \epsilon \}
FOLLOW(S’) = { $, e }

FOLLOW(E) = { t }

Parsing table:

<table>
<thead>
<tr>
<th>NON-TERMINAL</th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>i</th>
<th>t</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>S → iEtSS’</td>
</tr>
<tr>
<td>S’</td>
<td></td>
<td></td>
<td>S’ → eS</td>
<td></td>
<td></td>
<td>S’ → ε</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td>E → b</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Actions performed in predictive parsing:
1. Shift
2. Reduce
3. Accept
4. Error

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.

3.5 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.

3.5.1 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 → aABe
A → Abc | b
B → d
The sentence to be recognized is abbcde.
REDUCTION (LEFTMOST)  RIGHTMOST DERIVATION

abcde  (A → b)  S → aABe
aAbcde  (A → Abc)  → aAde
aAde  (B → d)  → aAbcde
aABe  (S → aABe)  → 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*E
  → E+E*id₃
  → E+id₂*id₃
  → id₁+id₂*id₃

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-sentinel form of some rightmost derivation.
**Stack implementation of shift-reduce parsing:**

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>id_1+id_2*id_3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ id_1</td>
<td>+id_2*id_3 $</td>
<td>reduce by E→id</td>
</tr>
<tr>
<td>$ E</td>
<td>+id_2*id_3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E+</td>
<td>id_2*id_3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E+id_2</td>
<td>*id_3 $</td>
<td>reduce by E→id</td>
</tr>
<tr>
<td>$ E+E</td>
<td>*id_3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E+E*</td>
<td>id_3 $</td>
<td>shift</td>
</tr>
<tr>
<td>$ E+E*id_3</td>
<td>$</td>
<td>reduce by E→id</td>
</tr>
<tr>
<td>$ E+E*E</td>
<td>$</td>
<td>reduce by E→E *E</td>
</tr>
<tr>
<td>$ E+E</td>
<td>$</td>
<td>reduce by E→E+E</td>
</tr>
<tr>
<td>$ E</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

**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 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.

**1. Shift-reduce conflict:**

**Example:**
Consider the grammar:
E→E+E | E*E | id and input id+id*id
### 2. Reduce-reduce conflict:

Consider the grammar:

\[
M \rightarrow R+R \mid R+c \mid R \\
R \rightarrow c
\]

and input c+c

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
<th>Stack</th>
<th>Input</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>c+c</td>
<td>Shift</td>
<td>$</td>
<td>c+c</td>
<td>Shift</td>
</tr>
<tr>
<td>$ c</td>
<td>+c</td>
<td>Reduce by R→c</td>
<td>$ c</td>
<td>+c</td>
<td>Reduce by R→c</td>
</tr>
<tr>
<td>$ R</td>
<td>+c</td>
<td>Shift</td>
<td>$ R</td>
<td>+c</td>
<td>Shift</td>
</tr>
<tr>
<td>$ R+</td>
<td>c</td>
<td>Shift</td>
<td>$ R+</td>
<td>c</td>
<td>Shift</td>
</tr>
<tr>
<td>$ R+c</td>
<td>$</td>
<td>Reduce by R→c</td>
<td>$ R+c</td>
<td>$</td>
<td>Reduce by M→R+c</td>
</tr>
<tr>
<td>$ R+R</td>
<td>$</td>
<td>Reduce by M→R+R</td>
<td>$ M</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>$ M</td>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Viable prefixes:

- $\alpha$ is a viable prefix of the grammar if there is $w$ such that $\alpha w$ is a right sentinel form.
- The set of prefixes of right sentinel forms that can appear on the stack of a shift-reduce parser are called viable prefixes.
- The set of viable prefixes is a regular language.

3.5.2 OPERATOR-PRECEDENCE PARSING

An efficient way of constructing shift-reduce parser is called operator-precedence parsing.

Operator precedence parser can be constructed from a grammar called Operator-grammar. These grammars have the property that no production on right side is $\varepsilon$ or has two adjacent non-terminals.

Example:

Consider the grammar:

$E \rightarrow EA \, E \mid (E) \mid -E \mid \text{id}$

$A \rightarrow + \mid - \mid \ast \mid / \mid \uparrow$

Since the right side EA has three consecutive non-terminals, the grammar can be written as follows:

$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid -E \mid \text{id}$

Operator precedence relations:

There are three disjoint precedence relations namely

- $<$ - less than
- $=$ - equal to
- $>$ - greater than

The relations give the following meaning:

- $a < b$ -- $a$ yields precedence to $b$
- $a = b$ -- $a$ has the same precedence as $b$
- $a > b$ -- $a$ takes precedence over $b$

Rules for binary operations:

1. If operator $\theta_1$ has higher precedence than operator $\theta_2$, then make
   $\theta_1 \, > \, \theta_2$ and $\theta_2 \, < \, \theta_1$

2. If operators $\theta_1$ and $\theta_2$ are of equal precedence, then make
   $\theta_1 \, > \, \theta_2$ and $\theta_2 \, > \, \theta_1$ if operators are left associative
   $\theta_1 \, < \, \theta_2$ and $\theta_2 \, < \, \theta_1$ if right associative

3. Make the following for all operators $\theta$:

   $\theta \, < \, \ast \, \text{id} \, , \, \text{id} \, > \, \theta$
   $\theta \, < \, (\, , (\, < \, \ast \, \theta$
   $\, > \, \theta \, , \, \theta \, > \, )$
   $\theta \, > \, \$, $\, < \, \ast \, \theta$
Also make

\( ( = ) , ( < \cdot ( , ) \cdot > ) , ( < \cdot \text{id} , \text{id} \cdot > ) , $ < \cdot \text{id} , \text{id} \cdot > $ , $ < \cdot ( , ) \cdot > $ \)

**Example:**

Operator-precedence relations for the grammar

\( E \rightarrow E+E | E-E | E^*E | E/E | E \uparrow E | (E) | -E | \text{id} \)

is given in the following table assuming

1. \( \uparrow \) is of highest precedence and right-associative
2. * and / are of next higher precedence and left-associative, and
3. + and - are of lowest precedence and left-associative

Note that the blanks in the table denote error entries.

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>( \uparrow )</th>
<th>id</th>
<th>(</th>
<th>)</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>*</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>/</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>( \uparrow )</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>id</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
<td>&gt;</td>
</tr>
<tr>
<td>(</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>=</td>
</tr>
<tr>
<td>)</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
<tr>
<td>$</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
<td>&lt;</td>
</tr>
</tbody>
</table>

**Operator precedence parsing algorithm:**

**Input** : An input string \( w \) and a table of precedence relations.

**Output** : If \( w \) is well formed, a skeletal parse tree, with a placeholder non-terminal \( E \) labeling all interior nodes; otherwise, an error indication.

**Method** : Initially the stack contains \$\$ and the input buffer the string \( w \$\). To parse, we execute the following program:

1. Set \( \text{ip} \) to point to the first symbol of \( w\$\);
2. **repeat forever**
3. **if** \$\$ is on top of the stack and \( \text{ip} \) points to \$** then**
4. **return else**
   **begin**
   5. let \( a \) be the topmost terminal symbol on the stack and let \( b \) be the symbol pointed to by \( \text{ip} \);
   6. **if** \( a < b \) **then begin**
   7. push \( b \) onto the stack;
Stack implementation of operator precedence parsing:
Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.
The initial configuration of an operator precedence parsing is

\[
\text{STACK} \quad \text{INPUT} \\
\$ \quad \wedge w \$
\]

where \( w \) is the input string to be parsed.

Example:
Consider the grammar \( E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E\uparrow E \mid (E) \mid \text{id} \). Input string is \( \text{id}+\text{id}*\text{id} \). The implementation is as follows:

<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>\wedge id+id*id $</td>
<td>shift id</td>
</tr>
<tr>
<td>$ id</td>
<td>\rightarrow +id*id $</td>
<td>pop the top of the stack id</td>
</tr>
<tr>
<td>$</td>
<td>\wedge +id*id $</td>
<td>shift +</td>
</tr>
<tr>
<td>$ +</td>
<td>\wedge id*id $</td>
<td>shift id</td>
</tr>
<tr>
<td>$ + id</td>
<td>\rightarrow *id $</td>
<td>pop id</td>
</tr>
<tr>
<td>$ +</td>
<td>\wedge *id $</td>
<td>shift *</td>
</tr>
<tr>
<td>$ + *</td>
<td>\wedge id $</td>
<td>shift id</td>
</tr>
<tr>
<td>$ + * id</td>
<td>\rightarrow $</td>
<td>pop id</td>
</tr>
<tr>
<td>$ + *</td>
<td>\rightarrow $</td>
<td>pop *</td>
</tr>
<tr>
<td>$</td>
<td>\rightarrow $</td>
<td>pop +</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>accept</td>
</tr>
</tbody>
</table>

Advantages of operator precedence parsing:
1. It is easy to implement.
2. Once an operator precedence relation is made between all pairs of terminals of a grammar, the grammar can be ignored. The grammar is not referred anymore during implementation.

Disadvantages of operator precedence parsing:
1. It is hard to handle tokens like the minus sign (-) which has two different precedence.
2. Only a small class of grammar can be parsed using operator-precedence parser.
3.5.3 LR PARSERS

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

Advantages of LR parsing:
✓ It recognizes virtually all programming language constructs for which CFG can be written.
✓ It is an efficient non-backtracking shift-reduce parsing method.
✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
✓ 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:
It consists of: an input, an output, a stack, a driver program, and a parsing table that has two parts (action and goto).

- The driver program is the same for all LR parsers.
- The parsing program reads characters from an input buffer one at a time.
- The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where $s_m$ is on top. Each $X_i$ is a grammar symbol and each $s_i$ is a state.
- The parsing table consists of two parts: action and goto functions.

**Action**: The parsing program determines $s_m$, the state currently on top of stack, and $a_i$, the current input symbol. It then consults action[$s_m,a_i$] 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 $A \rightarrow \beta$,
3. accept, and
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 $s_0$ on its stack, where $s_0$ is the initial state, and $w\$$ in the input buffer. The parser then executes the following program:

```plaintext
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 $A \rightarrow \beta$ then begin
        pop 2* | $\beta$ | 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 $A \rightarrow \beta$
    end
    else if action[s, a] = accept then
        return
    else error( )
end
```
3.6 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 $A \rightarrow XYZ$ yields the four items:

- $A \rightarrow . \ XYZ$
- $A \rightarrow X . \ YZ$
- $\rightarrow XY . \ Z A$
- $\rightarrow XYZ .$

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

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

Steps to construct SLR parsing table for grammar G are:
1. Augment G and produce G’
2. Construct the canonical collection of sets 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 = {$I_0, I_1, \ldots, I_n$}, the collection of sets of LR(0) items for G’.
2. State i is constructed from $I_i$. The parsing functions for state i are determined as follows:
   (a) If $[A \rightarrow \alpha \cdot a \beta]$ is in $I_i$ and goto($I_i, a$) = $I_j$, then set action[i,a] to “shift j”. Here a must be terminal.
   (b) If $[A \rightarrow \alpha \cdot]$ is in $I_i$, then set action[i,a] to “reduce $A \rightarrow \alpha$” for all a in FOLLOW(A).
   (c) If $[S' \rightarrow S .]$ is in $I_i$, then set action[i,$\$] to “accept”.

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).
3. The goto transitions for state \( i \) are constructed for all non-terminals \( A \) using the rule:
   If \( \text{goto}(I_i, A) = I_j \), then \( \text{goto}[i, A] = j \).
4. All entries not defined by rules (2) and (3) are made “error”.
5. The initial state of the parser is the one constructed from the set of items containing \( [S' \rightarrow \cdot S] \).

**Example for SLR parsing:**
Construct SLR parsing for the following grammar.

\[
G : E \rightarrow E + T | T \\
T \rightarrow T * F | F \\
F \rightarrow (E) | id
\]

The given grammar is:

\[
G : E \rightarrow E + T \quad ----- \quad (1) \\
E \rightarrow T \quad ----- \quad (2) \\
T \rightarrow T * F \quad ----- \quad (3) \\
T \rightarrow F \quad ----- \quad (4) \\
F \rightarrow (E) \quad ----- \quad (5) \\
F \rightarrow id \quad ----- \quad (6)
\]

**Step 1:** Convert given grammar into augmented grammar.

**Augmented grammar:**

\[
E' \rightarrow E \\
E \rightarrow E + T \\
E \rightarrow T \\
T \rightarrow T * F \\
T \rightarrow F \\
F \rightarrow (E) \\
F \rightarrow id
\]

**Step 2:** Find LR (0) items.

\[
I_0 : E' \rightarrow \cdot E \\
E \rightarrow \cdot E + T \\
E \rightarrow \cdot T \\
T \rightarrow \cdot T * F \\
T \rightarrow \cdot F \\
F \rightarrow \cdot (E) \\
F \rightarrow \cdot id
\]

\[
\text{GOTO} ( I_0, E ) \quad \text{GOTO} ( I_1, id ) \\
I_1 : E' \rightarrow \cdot E \\
E \rightarrow \cdot E + T
\]

\[
I_5 : F \rightarrow \cdot id
\]

**M.I.E.T./CSE/III Yr/Compiler Design**
GOTO (I_0, T)
I_2: E → T.
    T → T · * F

GOTO (I_0, F)
I_3: T → F.

GOTO (I_0, )
I_4: F → (E)
    E → . E + T
    E → . T
    T → . T · F
    T → . F
    F → . (E)
    F → . id

GOTO (I_0, id)
I_5: F → id.

GOTO (I_1, +)
I_6: E → E + . T
    T → . T · F
    T → . F
    F → . (E)
    F → . id

GOTO (I_2, *)
I_7: T → T · * F
    F → . (E)
    F → . id

GOTO (I_4, E)
I_8: F → (E.)
    E → E . + T

GOTO (I_6, T)
I_9: E → E + T.
    T → T · * F

GOTO (I_6, F)
I_3: T → F.

GOTO (I_6, )
I_4: F → (E.)
    E → . E + T
    E → . T
    T → . T · F
    T → . F
    F → . (E)
    F → . id

GOTO (I_7, id)
I_5: F → id.

GOTO (I_8, )
I_11: F → (E).

GOTO (I_8, +)
I_6: E → E + . T
    T → . T · F
    T → . F
    F → . (E)
    F → . id

GOTO (I_9, *)
I_7: T → T · * F
    F → . (E)
    F → . id
GOTO ( I₄ , ( )
I₄ : F → (. E)
   E → . E + T
   E → . T
   T → . T * F
   T → . F
   F → . (E)
   F → id

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

SLR parsing table:

<table>
<thead>
<tr>
<th></th>
<th>ACTION</th>
<th>GOTO</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>id</td>
<td>+</td>
</tr>
<tr>
<td>I₀</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>I₁</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>I₂</td>
<td>r2</td>
<td>s7</td>
</tr>
<tr>
<td>I₃</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>I₄</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>I₅</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>I₆</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>I₇</td>
<td>s5</td>
<td>s4</td>
</tr>
<tr>
<td>I₈</td>
<td>s6</td>
<td></td>
</tr>
<tr>
<td>I₉</td>
<td>r1</td>
<td>s7</td>
</tr>
<tr>
<td>I₁₀</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>I₁₁</td>
<td>r5</td>
<td>r5</td>
</tr>
</tbody>
</table>

Blank entries are error entries.

Stack implementation:

Check whether the input id + id * id is valid or not.
<table>
<thead>
<tr>
<th>STACK</th>
<th>INPUT</th>
<th>ACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>id + id * id $</td>
<td>GOTO ( I₀, id ) = s5 ; shift</td>
</tr>
<tr>
<td>0 id 5</td>
<td>+ id * id $</td>
<td>GOTO ( I₅, + ) = r6 ; reduce by F→id</td>
</tr>
<tr>
<td>0 F 3</td>
<td>+ id * id $</td>
<td>GOTO ( I₀, F ) = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₃, + ) = r4 ; reduce by T→F</td>
</tr>
<tr>
<td>0 T 2</td>
<td>+ id * id $</td>
<td>GOTO ( I₀, T ) = 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₂, + ) = r2 ; reduce by E→T</td>
</tr>
<tr>
<td>0 E 1</td>
<td>+ id * id $</td>
<td>GOTO ( I₀, E ) = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₁, + ) = s6 ; shift</td>
</tr>
<tr>
<td>0 E 1 + 6</td>
<td>id * id $</td>
<td>GOTO ( I₆, id ) = s5 ; shift</td>
</tr>
<tr>
<td>0 E 1 + 6 id 5</td>
<td>* id $</td>
<td>GOTO ( I₅, * ) = r6 ; reduce by F→id</td>
</tr>
<tr>
<td>0 E 1 + 6 F 3</td>
<td>* id $</td>
<td>GOTO ( I₆, F ) = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₃, * ) = r4 ; reduce by T→F</td>
</tr>
<tr>
<td>0 E 1 + 6 T 9</td>
<td>* id $</td>
<td>GOTO ( I₀, T ) = 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₀, * ) = s7 ; shift</td>
</tr>
<tr>
<td>0 E 1 + 6 T 9 * 7</td>
<td>id $</td>
<td>GOTO ( I₇, id ) = s5 ; shift</td>
</tr>
<tr>
<td>0 E 1 + 6 T 9 * 7 id 5</td>
<td>$</td>
<td>GOTO ( I₅, $ ) = r6 ; reduce by F→id</td>
</tr>
<tr>
<td>0 E 1 + 6 T 9 * 7 F 10</td>
<td>$</td>
<td>GOTO ( I₇, F ) = 10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₁₀, $ ) = r3 ; reduce by T→T * F</td>
</tr>
<tr>
<td>0 E 1 + 6 T 9</td>
<td>$</td>
<td>GOTO ( I₆, T ) = 9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₀, $ ) = r1 ; reduce by E→E + T</td>
</tr>
<tr>
<td>0 E 1</td>
<td>$</td>
<td>GOTO ( I₀, E ) = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GOTO ( I₁, $ ) = accept</td>
</tr>
</tbody>
</table>
Constructing Canonical LR(1) Parsing Tables

- In SLR method, the state i makes a reduction by $A \rightarrow \alpha$ when the current token is a:
  - if the $A \rightarrow \alpha$. in the $I_i$ and a is FOLLOW($A$)
  - In some situations, $\beta A$ cannot be followed by the terminal a in a right-sentential form when $\beta \alpha$ and the state i are on the top of stack. This means that making reduction in this case is not correct.

- In the grammar $S \rightarrow L = R,.. I_2$ has $R \rightarrow L$ and FOLLOW($R$) has ‘=‘. So in [2,=] reduce action is carried out. But there is no right sentential form with $R=$

LR(1) Item

- To avoid some of invalid reductions, the states need to carry more information.

- Extra information is put into a state by including a terminal symbol as a second component in an item.

- A LR(1) item is:
  
  $A \rightarrow \alpha.\beta,a$  
  where $a$ is the look-head of the LR(1) item
  ($a$ is a terminal or end-marker.)

- When $\beta$ (in the LR(1) item $A \rightarrow \alpha.\beta,a$) is not empty, the look-head does not have any affect.

- When $\beta$ is empty ($A \rightarrow \alpha.,a$), we do the reduction by $A \rightarrow \alpha$ only if the next input symbol is a (not for any terminal in FOLLOW($A$)).

- A state will contain $A \rightarrow \alpha.,a_1$ where $\{a_1,.,a_n\} \subseteq$ FOLLOW($A$)

  ...  

  $A \rightarrow \alpha.,a_n$

Canonical Collection of Sets of LR(1) Items

- The construction of the canonical collection of the sets of LR(1) items are similar to the construction of the canonical collection of the sets of LR(0) items, except that closure and goto operations work a little bit different.

$\text{closure}(I)$ is: (where I is a set of LR(1) items)

- every LR(1) item in I is in closure(I)

- if $A \rightarrow \alpha.B\beta,a$ in closure(I) and $B \rightarrow \gamma$ is a production rule of G; then $B \rightarrow \gamma,b$ will be in the closure(I) for each terminal b in FIRST($\beta a$).
goto operation
- If I is a set of LR(1) items and X is a grammar symbol (terminal or non-terminal), then goto(I,X) is defined as follows:
  - If \( A \rightarrow \alpha .X\beta .a \) in I
    then every item in closure({\( A \rightarrow \alpha X\beta .a \)}) will be in goto(I,X).

Construction of The Canonical LR(1) Collection
- Algorithm:

C is \( \{ \text{ closure} \{ S' \rightarrow S.\$ \} \} \)
repeat the followings until no more set of LR(1) items can be added to C.
for each I in C and each grammar symbol X
if goto(I,X) is not empty and not in C
add goto(I,X) to C
• goto function is a DFA on the sets in C.

A Short Notation for The Sets of LR(1) Items
- A set of LR(1) items containing the following items

\[ A \rightarrow \alpha .\beta .a_1 \]
\[ \ldots \]
\[ A \rightarrow \alpha .\beta .a_n \]
can be written as \( A \rightarrow \alpha .\beta .a_1/a_2/\ldots/a_n \)

Canonical LR(1) Collection – Example

<table>
<thead>
<tr>
<th>S \rightarrow AaAb</th>
<th>I_0:</th>
<th>S' \rightarrow S.$</th>
<th>I_1:</th>
<th>S' \rightarrow S.$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S \rightarrow BbBa</td>
<td></td>
<td>S \rightarrow AaAb.$</td>
<td></td>
<td>S \rightarrow AaAb.$</td>
</tr>
<tr>
<td>A \rightarrow \epsilon</td>
<td></td>
<td>S \rightarrow BbBa.$</td>
<td></td>
<td>S \rightarrow BbBa.$</td>
</tr>
<tr>
<td>B \rightarrow \epsilon</td>
<td></td>
<td>A \rightarrow \ldots.a</td>
<td></td>
<td>B \rightarrow \ldots.b</td>
</tr>
</tbody>
</table>

| I_2: | S \rightarrow AaAb.\$ | I_6: | S \rightarrow AaA.b.\$ | I_9: | S \rightarrow AaAb.\$ |
| A \rightarrow \ldots.b |     | I_7: | S \rightarrow BbB.a.\$ |     | I_9: | S \rightarrow BbBa.\$ |
| I_7: | S \rightarrow BbBa.\$ |     | B \rightarrow \ldots.a |     | I_6: | S \rightarrow BbBa.\$ |

Canonical LR(1) Collection – Example2

Construction of LR(1) Parsing Tables
1. Construct the canonical collection of sets of LR(1) items for G’. \( C \leftarrow \{ I_0, \ldots, I_n \} \)
2. Create the parsing action table as follows

- If a is a terminal, \( A \rightarrow \alpha .a\beta .b \) in \( I_i \) and goto(I_i,a)=I_j then action[i,a] is \text{ shift } j.
- If \( A \rightarrow \alpha .a \) is in \( I_i \), then action[i,a] is \text{ reduce } A \rightarrow \alpha \) where \( A \neq S' \).
Subject code/Name: CS6660 Compiler Design

- If $S' \rightarrow S.,$ is in $I_i$, then action[$i,S.$] is `accept`.
- If any conflicting actions generated by these rules, the grammar is not LR(1).

3. Create the parsing goto table
   - for all non-terminals $A$, if goto($i,A$)=$I_j$ then goto[$i,A$]=$j$

4. All entries not defined by (2) and (3) are errors.

5. Initial state of the parser contains $S' \rightarrow S.,$

<table>
<thead>
<tr>
<th>id</th>
<th>=</th>
<th>S</th>
<th>S</th>
<th>L</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>$\text{acc}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>s6</td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>r2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
<td>s4</td>
<td></td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>r4</td>
<td>r4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>r3</td>
<td>r3</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>r5</td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td>r1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>r5</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>s12</td>
<td>s11</td>
<td></td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>r4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td>r3</td>
<td></td>
</tr>
</tbody>
</table>

3.7 LALR Parsing Tables
- LALR stands for LookAhead LR.

- LALR parsers are often used in practice because LALR parsing tables are smaller than LR(1) parsing tables.
- The number of states in SLR and LALR parsing tables for a grammar $G$ are equal.
- But LALR parsers recognize more grammars than SLR parsers.
- yacc creates a LALR parser for the given grammar.
- A state of LALR parser will be again a set of LR(1) items.

Creating LALR Parsing Tables
Canonical LR(1) Parser $\Rightarrow$ LALR Parser
shrink # of states
- This shrink process may introduce a reduce/reduce conflict in the resulting LALR parser (so the grammar is NOT LALR)
- But, this shrink process does not produce a shift/reduce conflict.

The Core of A Set of LR(1) Items
The core of a set of LR(1) items is the set of its first component.

Ex: \( S \rightarrow L. = R, $ \rightarrow S \rightarrow L. = R \)
\( R \rightarrow L., $ \rightarrow R \rightarrow L. \)

• We will find the states (sets of LR(1) items) in a canonical LR(1) parser with same cores. Then we will merge them as a single state.

\[ I_1: L \rightarrow \text{id.}, = \rightarrow I_{12}: L \rightarrow \text{id.}, = \]
\( L \rightarrow \text{id.}, $ \)

\( I_2: L \rightarrow \text{id.}, $ \)

We will do this for all states of a canonical LR(1) parser to get the states of the LALR parser.

In fact, the number of the states of the LALR parser for a grammar will be equal to the number of states of the SLR parser for that grammar.

Creation of LALR Parsing Tables

• Create the canonical LR(1) collection of the sets of LR(1) items for the given grammar.

• Find each core; find all sets having that same core; replace those sets having same cores with a single set which is their union.

\[ C = \{I_0, ..., I_n\} \rightarrow C' = \{J_1, ..., J_m\} \text{ where } m \leq n \]

• Create the parsing tables (action and goto tables) same as the construction of the parsing tables of LR(1) parser.

  – Note that: If \( J=I_1 \cup ... \cup I_k \) since \( I_1, ..., I_k \) have same cores

  \[ \Rightarrow \text{cores of goto}(I_1, X), ..., \text{goto}(I_2, X) \text{ must be same.} \]

  – So, \( \text{goto}(J, X) = K \) where \( K \) is the union of all sets of items having same cores as \( \text{goto}(I_1, X) \).

If no conflict is introduced, the grammar is LALR(1) grammar. (We may only introduce reduce/reduce conflicts; we cannot introduce a shift/reduce conflict)

Shift/Reduce Conflict

• We say that we cannot introduce a shift/reduce conflict during the shrink process for the creation of the states of a LALR parser.

• Assume that we can introduce a shift/reduce conflict. In this case, a state of LALR parser must have:

\[ A \rightarrow \alpha., a \quad \text{and} \quad B \rightarrow \beta.a, \gamma.b \]

• This means that a state of the canonical LR(1) parser must have:

\[ A \rightarrow \alpha., a \quad \text{and} \quad B \rightarrow \beta.a, \gamma.c \]

But, this state has also a shift/reduce conflict. i.e. The original canonical LR(1) parser has a conflict.

(Reason for this, the shift operation does not depend on lookaheads)
Reduce/Reduce Conflict

- But, we may introduce a reduce/reduce conflict during the shrink process for the creation of the states of a LALR parser.

\[
\begin{align*}
I_1 & : A \to \alpha.a \\
B & \to \beta.b \\
I_2 & : A \to \alpha.b \\
B & \to \beta.c \\
\downarrow \\
I_{12} & : A \to \alpha.a/b \\
B & \to \beta.b/c \quad \text{reduce/reduce conflict}
\end{align*}
\]

3.8 Error Recovery in LR Parsing

- An LR parser will detect an error when it consults the parsing action table and finds an error entry. All empty entries in the action table are error entries.

- Errors are never detected by consulting the goto table.

- An LR parser will announce error as soon as there is no valid continuation for the scanned portion of the input.

- A canonical LR parser (LR(1) parser) will never make even a single reduction before announcing an error.

- The SLR and LALR parsers may make several reductions before announcing an error.

- But, all LR parsers (LR(1), LALR and SLR parsers) will never shift an erroneous input symbol onto the stack.

Panic Mode Error Recovery in LR Parsing

- Scan down the stack until a state \( s \) with a goto on a particular nonterminal \( A \) is found. (Get rid of everything from the stack before this state \( s \)).

- Discard zero or more input symbols until a symbol \( a \) is found that can legitimately follow \( A \).
  - The symbol \( a \) is simply in FOLLOW(\( A \)), but this may not work for all situations.

- The parser stacks the nonterminal \( A \) and the state goto[s,A], and it resumes the normal parsing.

- This nonterminal \( A \) is normally is a basic programming block (there can be more than one choice for \( A \)).
  - stmt, expr, block, ...

Phrase-Level Error Recovery in LR Parsing

- Each empty entry in the action table is marked with a specific error routine.

- An error routine reflects the error that the user most likely will make in that case.

- An error routine inserts the symbols into the stack or the input (or it deletes the symbols from the stack and the input, or it can do both insertion and deletion).
Error Routines

<table>
<thead>
<tr>
<th>Action</th>
<th>Goto</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>e1</td>
</tr>
<tr>
<td>0</td>
<td>s3</td>
</tr>
<tr>
<td>1</td>
<td>e3</td>
</tr>
<tr>
<td>2</td>
<td>s3</td>
</tr>
<tr>
<td>3</td>
<td>r4</td>
</tr>
<tr>
<td>4</td>
<td>s3</td>
</tr>
<tr>
<td>5</td>
<td>s3</td>
</tr>
<tr>
<td>6</td>
<td>e3</td>
</tr>
<tr>
<td>7</td>
<td>r1</td>
</tr>
<tr>
<td>8</td>
<td>r2</td>
</tr>
<tr>
<td>9</td>
<td>r3</td>
</tr>
</tbody>
</table>

3.9 YACC - YET ANOTHER COMPILER-COMPILER

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

First a file say Parse.y containing a Yacc specification for an expression is prepared. The UNIX system command Yacc parse.y transforms the file parse.y into a C program called Y.tab.c which is a representation of parser written in C language along with other C programs that the user may have prepared. Y.tab.c is run through C compiler and produces object program a.out that performs the translation specified by the original Yacc program. Yacc source program has also 3 parts as Lex. This can be expressed in Yacc specification as:

declaration

```%

```

translation rules
% %

C - Programs

Example:
To illustrate how to prepare a Yacc source program, let us construct a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value. We shall build the desk calculator staffing with the following grammar for arithmetic expressions:

```
Expr    --  expr + term | term
term    --  term * factor | factor
factor  --  (expr) digit
```

The token digit is a single digit ranging from 0 to 9. A Yacc desk calculator program derived from this grammar is shown in figure 13.

**The declarations part.**

There are two optional sections in the declarations part of a Yacc program. In the first section, we write ordinary C declarations, delimited by %( and %). Here we place declarations of any temporaries used by the translation rules or procedures of the second and third sections. In figure 13, this section contains only the include-statement #include < ctype.h > that causes the C pre-processor to include the standard header file < ctype.h > that contains the predicate is digit.

The Yacc specification of a simple desk calculator

Also in the declarations part are declarations of grammar tokens. In figure 13 the statement

```
%token DIGIT
```

declares DIGIT (pre-defined) to be a token. Tokens declared in this section can then be used in the second and third parts of the Yacc specification.

**The translation rules part.**
In the part of the Yacc specification after the first `%%` pair, put the translation rules. Each rule consists of a grammar production and the associated semantic action. A set of productions that we have been writing

\[
<\text{left side}> - <\text{alt 1}> | <\text{alt 2}> ... | <\text{alt n}>
\]

would be written in Yacc as

\[
- | ...
\]

would be written in Yacc as

\[
: (\text{semantic action I})
\]

\[
: (\text{semantic action 2})
\]

\[
: (\text{semantic action n})
\]

In a Yacc production, a quoted single character 'c' is taken to be the terminal symbol c, a unquoted strings of letters and digits not declared to be tokens are taken to be nonterminals. Alternative right sides can be separated by a vertical bar, and a semicolon follows each left side is taken to be the start symbol.

A Yacc semantic action is a sequence of C statements. In a semantic action, the symbol $$ refers to the attribute value associated with the nonterminal on the left, while $i refers to value associated with ith grammar symbol (terminal or nonterminal) on the right. The semantic action is performed whenever we reduce by die associated production, so normally the semantic action computes a value for $$ in terms of the $i's. In the Yacc specification, we have written the two production rules for expressions (expr).

Expr : expr + term 1 term

and their associated semantic actions as

expr : expr '+' term {$$ = $1 + $3;}

: term

Note that the nonterminal term in the fast production is the third grammar symbol on the right while '+' is the second. The semantic action associated with the first production adds the value of the expr and the term on the right and assigns the result as the value for the non-terminal expr on the left. We have omitted the semantic action for the second production altogether, since copying the value is the default action for productions with a single grammar symbol on the right. In general, { $$ = $1; } is the default semantic action.

Notice that we have added a new starting production

Line : expr 'n' { printf("%d\n");}

to the Yacc specification. This production says that an input to the desk calculator is to be an expression. The semantic action associated with this production prints the decimal value of the expression.

The supporting C-routines part The third part of a Yacc specification consists of supporting C-routines. A lexical analyzer by the name lex( ) must be provided. Other procedures such as error recovery routines may be added as necessary.

The lexical analyser lex( ) produces pairs consisting of a token and its associated attribute value. If a token such as DIGIT is returned, the token must be declared in the first section of the Yacc specification. The attribute value associated with a token is communicated to the parser through a Yacc defined variable Ival.
UNIT IV SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT


RUN-TIME ENVIRONMENT: Source Language Issues—Storage Organization—Storage Allocation—Parameter Passing—Symbol Tables—Dynamic Storage Allocation—Storage Allocation in FORTRAN.

4.1 Syntax-directed translation (SDT)

It refers to a method of compiler implementation where the source language translation is completely driven by the parser, i.e., based on the syntax of the language. The parsing process and parse trees are used to direct semantic analysis and the translation of the source program. Almost all modern compilers are syntax-directed.

SDT can be a separate phase of a compiler or we can augment our conventional grammar with information to control the semantic analysis and translation. Such grammars are called attribute grammars.

We augment a grammar by associating attributes with each grammar symbol that describes its properties. With each production in a grammar, we give semantic rules/actions, which describe how to compute the attribute values associated with each grammar symbol in a production.

The general approach to Syntax-Directed Translation is to construct a parse tree or syntax tree and compute the values of attributes at the nodes of the tree by visiting them in some order. In many cases, translation can be done during parsing without building an explicit tree.

A class of syntax-directed translations called "L-attributed translations" (L for left-to-right) includes almost all translations that can be performed during parsing. Similarly, "S-attributed translations" (S for synthesized) can be performed easily in connection with a bottom-up parse.

There are two ways to represent the semantic rules associated with grammar symbols.

- Syntax-Directed Definitions (SDD)
- Syntax-Directed Translation Schemes (SDT)

Syntax-Directed Definitions

A syntax-directed definition (SDD) is a context-free grammar together with attributes and rules. Attributes are associated with grammar symbols and rules are associated with productions.

An attribute has a name and an associated value: a string, a number, a type, a memory location, an assigned register, strings. The strings may even be long sequences of code, say code in the intermediate language used by a compiler. If X is a symbol and a is one of its attributes, then we
write X.a to denote the value of a at a particular parse-tree node labeled X. If we implement the
nodes of the parse tree by records or objects, then the attributes of X can be implemented by
data fields in the records that represent the nodes for X. The attributes are evaluated by the
semantic rules attached to the productions.

Example: PRODUCTION SEMANTIC RULE

E → E1 + T  E.code = E1.code || T.code || ‘+’

SDDs are highly readable and give high-level specifications for translations. But they hide many
implementation details. For example, they do not specify order of evaluation of semantic actions.

4.1.2 Syntax-Directed Translation Schemes (SDT)

SDT embeds program fragments called semantic actions within production bodies. The
position of semantic action in a production body determines the order in which the action is
executed.

Example: In the rule E → E1 + T { print ‘+’ }, the action is positioned after the body of the
production.

SDTs are more efficient than SDDs as they indicate the order of evaluation of semantic actions
associated with a production rule. This also gives some information about implementation
details.

Inherited and Synthesized Attributes

Terminals can have synthesized attributes, which are given to it by the lexer (not the parser).
There are no rules in an SDD giving values to attributes for terminals. Terminals do not have
inherited attributes.

A nonterminal A can have both inherited and synthesized attributes. The difference is how they
are computed by rules associated with a production at a node N of the parse tree.
• A synthesized attribute for a nonterminal A at a parse-tree node N is defined by
  a semantic rule associated with the production at N. Note that the production must have A as its
  head.
A synthesized attribute at node N is defined only in terms of attribute values at the children of N
and at N itself.
• An inherited attribute for a nonterminal B at a parse-tree node N is defined by a semantic
  rule associated with the production at the parent of N. Note that the production must have B as a
  symbol in its body.
An inherited attribute at node N is defined only in terms of attribute values at N's parent, N
itself, and N's siblings.

An inherited attribute at node N cannot be defined in terms of attribute values at the children of
node N. However, a synthesized attribute at node N can be defined in terms of inherited
attribute values at node N itself.
Syntax-Directed Translation

- A syntax-directed definition (or attribute grammar) binds a set of semantic rules to productions.
- Terminals and nonterminals have attributes holding values set by the semantic rules.
- A depth-first traversal algorithm traverses the parse tree thereby executing semantic rules to assign attribute values.
- After the traversal is complete the attributes contain the translated form of the input.

Example Attribute Grammar

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E$</td>
<td>$L.val = E.val$</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>$E.val = E_1.val + T.val$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E.val = T.val$</td>
</tr>
<tr>
<td>$T \rightarrow T_1 \ast F$</td>
<td>$T.val = T_1.val \ast F.val$</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>$T.val = F.val$</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>$F.val = E.val$</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>$F.val = \text{digit.lexval}$</td>
</tr>
</tbody>
</table>

Note: all attributes in this example are of the synthesized type.
- It is a CFG grammar augmented with:
  - “Attributes” (assigned to each grammar symbol).
  - Semantic Rules (associated to each production involving the Attributes of the symbols in the production).
  - Attributes can be synthesized or inherited.
- Semantic Rules for a production $A \rightarrow \alpha$ have the form:
  - $b = f (c_1, \ldots, c_n)$ where
  - (b is synthesized) $b$ is an attribute of $A$ and $c_1 \ldots c_n$ are attributes of symbols in $\alpha$.
  - (b is inherited) $b$ is an attribute of some symbol in $\alpha$ and $c_1 \ldots c_n$ are attributes of symbols in $A, \alpha$.
- Terminals have only synthesized attributes whose values are provided by the lexical analyzer.
- The start non-terminal typically has no inherited attributes.
- We may allow function calls as semantic-rules also; these are “Side-effects”...
- Attribute values may represent...
- Numbers (literal constants)
- Strings (literal constants)
- Memory locations, such as a frame index of a local variable or function argument
- A data type for type checking of expressions
- Scoping information for local declarations
- Intermediate program representations

**Annotated Parse-Trees**
- Parse-tree that also shows the values of the attributes at each node.
- Values of Attributes in nodes of annotated parse-tree are either,
  - initialized to constant values or by the lexical analyzer.
  - determined by the semantic-rules

**Evaluating Attributes**
- If a syntax-directed definition employs only Synthesized attributes the evaluation of all attributes can be done in a bottom-up fashion.
- Inherited attributes would require more arbitrary “traversals” of the annotated parse-tree.

A dependency graph suggests possible evaluation orders for an annotated parse-tree

**Example of a Syntax-Directed Definition**
Grammar symbols: L, E, T, F, n, +, *, (, ) , digit
Non-terminals E, T, F have an attribute called val
Terminal digit has an attribute called lexval
The value for lexval is provided by the lexical analyzer.

**PRODUCTION**  **SEMANTIC RULE**

L \rightarrow E \ n \quad \text{print}(E.\text{val})
E \rightarrow E_1 + T \quad E.\text{val} = E_1.\text{val} + T.\text{val}
E \rightarrow T \quad E.\text{val} = T.\text{val}
T \rightarrow T_1 \ast F \quad T.\text{val} = T_1.\text{val} \ast F.\text{val}
T \rightarrow F \quad T.\text{val} = F.\text{val}
F \rightarrow (E) \quad F.\text{val} = E.\text{val}
F \rightarrow \text{digit} \quad F.\text{val} = \text{digit}.\text{lexval}

**Example: 3*5+4n**
Example with Inherited Attributes
- Even though inherited can be simulated by synthesized it is more natural to write Syntax-Directed Definitions using inherited.
- ...below in is an inherited attribute of L

**PRODUCTION**  **SEMANTIC RULE**

D → T L  \( L.\text{in} = T.\text{type} \)
T → int  \( T.\text{type} = \text{integer} \)
T → real  \( T.\text{type} = \text{real} \)
L → L₁, id  \( L₁.\text{in} = L.\text{in} \)
\( \text{addtype(id.entry, L.\text{in})} \)
L → id  \( \text{addtype(id.entry, L.\text{in})} \)

Example real \( id₁, id₂, id₃ \)

![Dependency Graph](image)

### 4.1.3 Dependency Graph
- Directed Graph
- Shows interdependencies between attributes.
- Construction:
  - Put each semantic rule into the form \( b = f(c₁, \ldots, cₖ) \) by introducing dummy synthesized attribute \( b \) for every semantic rule that consists of a procedure call.
  - E.g.,
    - \( L \rightarrow E n \rightarrow \text{print(E.val)} \)
    - Becomes: \( \text{dummy} = \text{print(E.val)} \)
    - Etc.

**Dependency Graph Construction**

for each node \( n \) in the parse tree do
  for each attribute \( a \) of the grammar symbol at \( n \) do
    construct a node in the dependency graph for \( a \)
  for each node \( n \) in the parse tree do
    for each semantic rule \( b = f(c₁, \ldots, cₙ) \)
Evaluating Attributes

- Notion: Directed Acyclic Graph
- Dependency graph should be a DAG (why?)
- Any topological sort of the directed acyclic graph can be used as a “guide” for attribute evaluation.

Example Annotated Parse Tree with Dependency Graph

Evaluation Methods

- Parse-tree methods determine an evaluation order from a topological sort of the dependence graph constructed from the parse tree for each input
- Rule-base methods the evaluation order is pre-determined from the semantic rules
- Oblivious methods the evaluation order is fixed and semantic rules must be (re)written to support the evaluation order (for example S-attributed definitions)

Applications of SDT:

4.2 Construction of Syntax Trees

- Decoupling Translation from Parsing-Trees.
- Syntax-Tree: an intermediate representation of the compiler’s input.
Example Procedures:
- `mknode`, `mkleaf` (create a labeled node – return pointer)
- Employment of the synthesized attribute `nptr` (type: pointer)

### Production Semantic Rule

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>E → E_1 + T</code></td>
<td><code>E.nptr = mknode(&quot;+&quot;, E_1.nptr, T.nptr)</code></td>
</tr>
<tr>
<td><code>E → E_1 - T</code></td>
<td><code>E.nptr = mknode(&quot;-&quot;, E_1.nptr, T.nptr)</code></td>
</tr>
<tr>
<td><code>E → T</code></td>
<td><code>E.nptr = T.nptr</code></td>
</tr>
<tr>
<td><code>T → (E)</code></td>
<td><code>T.nptr = E.nptr</code></td>
</tr>
<tr>
<td><code>T → id</code></td>
<td><code>T.nptr = mkleaf(id, id.lexval)</code></td>
</tr>
<tr>
<td><code>T → num</code></td>
<td><code>T.nptr = mkleaf(num, num.val)</code></td>
</tr>
</tbody>
</table>

or

### Steps in the construction of the syntax tree for `a-4+c`:

If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.
4.3 Bottom Up Evaluation of S-Attributed Definitions

- A syntax-directed definition is S-Attributed if all attributes are synthesized.
- Dependency graph is a (directed) tree “pointing upwards”
- Can be evaluated in bottom up fashion.
- => Consistent with (bottom-up) Shift/Reduce parsing.
- Possible to do the evaluation of the attributes using the stack at the time of “reduce” operations!

Using the Stack to compute the attributes

- Suppose the stack of a general Shift/Reduce parser is equal to $...XYZ$
- Suppose the grammar symbols A, X, Y, Z have the attributes a,x,y,z respectively.
- The augmented stack with the attributes is as follows:
- $...[X,X.x=v_1][Y,Y.y=v_2][Z, Z.z=v_3]$
- If we reduce by the production $A \rightarrow XYZ$ that has the semantic action $A.a = f(X.x, Y.y, Z.z)$ we will modify the stack as follows:
- $...[A,A.a=f(v_1,v_2,v_3)]$

Recall the S-Attributed Definition

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E \text{\ print}$</td>
<td>print(E.val)</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>E.val = E_1.val + T.val</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>E.val = T.val</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>T.val = T_1.val * F.val</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>T.val = F.val</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>F.val = E.val</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>F.val = digit .lexval</td>
</tr>
</tbody>
</table>

Semantic Rules Using the Stack of the S/R parser

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L \rightarrow E \text{\ print(val[top])}$</td>
<td>print(val[top])</td>
</tr>
<tr>
<td>$E \rightarrow E_1 + T$</td>
<td>val[ntop]=val[top-2]+val[top]</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>val[ntop]=val[top]</td>
</tr>
<tr>
<td>$T \rightarrow T_1 * F$</td>
<td>val[ntop]=val[top-2]*val[top]</td>
</tr>
<tr>
<td>$T \rightarrow F$</td>
<td>val[ntop]=val[top]</td>
</tr>
<tr>
<td>$F \rightarrow (E)$</td>
<td>val[ntop]=val[top-1]</td>
</tr>
<tr>
<td>$F \rightarrow \text{digit}$</td>
<td>val[ntop]=val[top]</td>
</tr>
</tbody>
</table>

top = top of the stack
ntop = top of the stack after popping right hand side of production.
val[...] = attribute value of stack contents
A trace of a Shift Reduce Parser with Attribute Evaluation

Bottom-up Evaluation of S-Attributed Definitions in Yacc

<table>
<thead>
<tr>
<th>Stack</th>
<th>val</th>
<th>Input</th>
<th>Action</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 3</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = $1</td>
</tr>
<tr>
<td>$F</td>
<td>3</td>
<td>$5+4n$</td>
<td>reduce T → F</td>
<td>$S = $1</td>
</tr>
<tr>
<td>$T</td>
<td>3</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = $1</td>
</tr>
<tr>
<td>$T * 3_5</td>
<td>$5+4n$</td>
<td>shift</td>
<td>$S = $1 * $3</td>
<td></td>
</tr>
<tr>
<td>$T * F 3_5</td>
<td>$5+4n$</td>
<td>reduce T → T * F</td>
<td>$S = $1</td>
<td></td>
</tr>
<tr>
<td>$T</td>
<td>15</td>
<td>$5+4n$</td>
<td>reduce E → T</td>
<td>$S = $1</td>
</tr>
<tr>
<td>$E</td>
<td>15</td>
<td>$4n$</td>
<td>shift</td>
<td>$S = $1</td>
</tr>
<tr>
<td>$E + 15_4</td>
<td>4n$</td>
<td>reduce F → digit</td>
<td>$S = $1</td>
<td></td>
</tr>
<tr>
<td>$E + F 15_4</td>
<td>4n$</td>
<td>reduce T → F</td>
<td>$S = $1</td>
<td></td>
</tr>
<tr>
<td>$E + T 15_4</td>
<td>4n$</td>
<td>reduce E → E +</td>
<td>$S = $1 + $3</td>
<td></td>
</tr>
<tr>
<td>$E n 19</td>
<td>nS</td>
<td>shift</td>
<td>print $1</td>
<td>$S = $1</td>
</tr>
<tr>
<td>$L</td>
<td>19</td>
<td>$S</td>
<td>accept</td>
<td>$S = $1</td>
</tr>
</tbody>
</table>

4.4 Design of a Predictive Translator:

Input: translation scheme based on a grammar suitable for predictive parsing
Output: Code for a syntax-directed translator
Method:
1. For each nonterminal A, construct a function with
   Input parameters: one for each inherited attribute of A;
   Return value: synthesized attributes of A;
   Local variables: one for each attribute of each grammar
   symbol that appears in a production for A.
2. Code for non-terminal A decides what production to use based on the current input symbol
   (switch statement). Code for each production forms one case of a switch statement.
3. In the code for a production, tokens, nonterminals, actions in the RHS are considered left to
   right.
   (i) For token X: save X.s in the variable created for X; generate a call to match X and advance
       input.
   (ii) For nonterminal B: generate an assignment c=B(b1, b2, ..., bk);
       where:
       b1, b2, ... are variables corresponding to inherited attributes of B,
       c is the variable for synthesized attribute of B,
       B is the function created for B.
   (iii) For an action, copy the code into the function, replacing each reference to an attribute by
       the variable created for that attribute.

4.5.1 TYPE CHECKING

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

Some examples of static checks:

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

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

   ![Position of type checker diagram](image)

   - A type checker verifies that the type of a construct matches that expected by its context. For example: arithmetic operator mod in Pascal requires integer operands, so a type checker verifies that the operands of mod have type integer.

   - Type information gathered by a type checker may be needed when code is generated.

### 4.5.2 TYPE SYSTEMS

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

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

**Type Expressions**

- The type of a language construct will be denoted by a “type expression.”

- A type expression is either a basic type or is formed by applying an operator called a **type constructor** to other type expressions.

- The sets of basic types and constructors depend on the language to be checked. The following are the definitions of type expressions:

1. Basic types such as boolean, char, integer, real are type expressions.

   A special basic type, `type_error`, will signal an error during type checking; `void` denoting “the absence of a value” allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.

Constructors include:
Arrays: If T is a type expression then array (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products: If T₁ and T₂ are type expressions, then their Cartesian product T₁ X T₂ is a type expression.

Records: The difference between a record and a product is that the fields of a record have names. The record type constructor will be applied to a tuple formed from field names and field types.

For example:

```
type row = record
  address: integer;
  lexeme: array[1..15] of char
end;
```

```
var table: array[1...101] of row;
```
declares the type name row representing the type expression record((address X integer) X (lexeme X array(1..15,char))) and the variable table to be an array of records of this type.

Pointers: If T is a type expression, then pointer(T) is a type expression denoting the type “pointer to an object of type T”.

For example, `var p: ↑ row` declares variable p to have type pointer(row).

Functions: A function in programming languages maps a domain type D to a range type R. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

```
Tree representation for char x char → pointer (integer)
```

```
   → 

   x     pointer
   char   char   integer
```

Type systems

- A type system is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checking done by a compiler is said to be static, while checking done when the target
program runs is termed dynamic.

- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

**Sound type system**

A sound type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than type_error to a program part, then type errors cannot occur when the target code for the program part is run.

**Strongly typed language**

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

**Error Recovery**

- Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.
- Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

**4.6 SPECIFICATION OF A SIMPLE TYPE CHECKER**

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

**A Simple Language**

Consider the following grammar:

\[ P \rightarrow D : E \]
\[ D \rightarrow D ; D | id : T \]
\[ T \rightarrow char | integer | array [ num ] of T | \uparrow T \]
\[ E \rightarrow literal | num | id | E mod E | E [ E ] | E \uparrow \]

**Translation scheme:**

\[ P \rightarrow D : E \]
\[ D \rightarrow D ; D \]
\[ D \rightarrow id : T \quad \{ \text{addtype (id.entry, T.type) } \} \]
\[ T \rightarrow char \quad \{ \text{T.type := char } \} \]
\[ T \rightarrow integer \quad \{ \text{T.type := integer } \} \]
\[ T \rightarrow \uparrow T1 \quad \{ \text{T.type := pointer(T1.type) } \} \]
\[ T \rightarrow array [ num ] of T1 \quad \{ \text{T.type := array (1… num.val, T1.type) } \} \]

In the above language,
There are two basic types: char and integer;
type_error is used to signal errors;
the prefix operator ↑ builds a pointer type. Example, ↑ integer leads to the type expression
pointer ( integer )

4.6.1 Type checking of expressions

In the following rules, the attribute type for E gives the type expression assigned to the
expression generated by E.

1. E → literal         { E.type = char }
   E → num              { E.type = integer }
   Here, constants represented by the tokens literal and num have type char and integer.

2. E → id               { E.type = lookup ( id.entry ) }
   lookup ( e ) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E₁ mod E₂       { E.type = if E₁.type = integer and
   E₂.type = integer then integer
   else type_error }  
   The expression formed by applying the mod operator to two subexpressions of type integer has
type integer; otherwise, its type is type_error.

4. E → E₁ [ E₂ ]        { E.type = if E₂.type = integer and
   E₁.type = array(s,t) then t
   else type_error }  
   In an array reference E₁ [ E₂ ] , the index expression E₂ must have type integer. The result is
   the element type t obtained from the type array(s,t) of E₁.

5. E → E₁ ↑             { E.type = if E₁.type = pointer (t) then t
   else type_error }  
   The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type t
   of the object pointed to by the pointer E.

4.6.2 Type checking of statements

Statements do not have values; hence the basic type void can be assigned to them. If an error is
detected within a statement, then type_error is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:
   S → id : = E           { S.type = if id.type = E.type then void
                              else type_error }  

2. Conditional statement:
   S → if E then S₁       { S.type = if E.type = boolean then S₁.type
                              else type_error }
3. While statement:
   \[ S \rightarrow \text{while} \ E \ \text{do} \ S_1 \{ S\text{.type} = \text{if} \ E\text{.type} = \text{boolean} \ \text{then} \ S_1\text{.type} \text{else type_error} \} \]

4. Sequence of statements:
   \[ S \rightarrow S_1 ; S_2 \{ S\text{.type} = \text{if} \ S_1\text{.type} = \text{void} \ \text{and} \ S_1\text{.type} = \text{void} \ \text{then} \ \text{void} \text{else type_error} \} \]

4.6.3 Type checking of functions

The rule for checking the type of a function application is:
\[ E \rightarrow E_1 ( E_2) \{ E\text{.type} = \text{if} \ E_2\text{.type} = s \ \text{and} \ E_1\text{.type} = s \rightarrow t \ \text{then} \ t \text{else type_error} \} \]

4.7 SOURCE LANGUAGE

ISSUES Procedures:
A procedure definition is a declaration that associates an identifier with a statement. The identifier is the procedure name, and the statement is the procedure body.
For example, the following is the definition of procedure named readarray:

```
procedure readarray;
var i: integer;
begin
  for i := 1 to 9 do read(a[i])
end;
```

When a procedure name appears within an executable statement, the procedure is said to be called at that point.

Activation trees:
An activation tree is used to depict the way control enters and leaves activations. In an activation tree,
1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for a is the parent of the node for b if and only if control flows from activation a to b.
4. The node for a is to the left of the node for b if and only if the lifetime of a occurs before the lifetime of b.

Control stack:
- A control stack is used to keep track of live procedure activations. The idea is to push the node for an activation onto the control stack as the activation begins and to pop the node when the activation ends.
The contents of the control stack are related to paths to the root of the activation tree. When node $n$ is at the top of control stack, the stack contains the nodes along the path from $n$ to the root.

**The Scope of a Declaration:**
A declaration is a syntactic construct that associates information with a name. Declarations may be explicit, such as:

```plaintext
var i : integer ;
```
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the program to which a declaration applies is called the **scope** of that declaration.

**Binding of names:**
Even if each name is declared once in a program, the same name may denote different data objects at run time. “Data object” corresponds to a storage location that holds values.

The term environment refers to a function that maps a name to a storage location. The term state refers to a function that maps a storage location to the value held there.

When an environment associates storage location $s$ with a name $x$, we say that $x$ is bound to $s$. This association is referred to as a binding of $x$.

### 4.8 STORAGE ORGANISATION

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

**Typical subdivision of run-time memory:**

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Data</td>
</tr>
<tr>
<td>Stack</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Heap</td>
</tr>
</tbody>
</table>
Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the control stack.
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.

- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.

<table>
<thead>
<tr>
<th>Actual Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returned values</td>
</tr>
<tr>
<td>Control link</td>
</tr>
<tr>
<td>Access link</td>
</tr>
<tr>
<td>Saved machine status</td>
</tr>
<tr>
<td>Local data</td>
</tr>
<tr>
<td>Temporaries</td>
</tr>
</tbody>
</table>
Temporary values such as those arising from the evaluation of expressions.
  • Local data belonging to the procedure whose activation record this is.
  • A saved machine status, with information about the state of the machine just before the call to procedures.
  • An access link may be needed to locate data needed by the called procedure but found elsewhere.
    A control link pointing to the activation record of the caller.
  Space for the return value of the called functions, if any. Again, not all called procedures
  return a value, and if one does, we may prefer to place that value in a register for efficiency.
  • The actual parameters used by the calling procedure. These are not placed in activation
    record but rather in registers, when possible, for greater efficiency.

4.9 STORAGE ALLOCATION STRATEGIES
The different storage allocation strategies are:
1. **Static allocation** – lays out storage for all data objects at compile time.
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area
    known as heap.

4.9.1 STATIC ALLOCATION
  • In static allocation, names are bound to storage as the program is compiled, so there is no
    need for a run-time support package.
  • Since the bindings do not change at run-time, everytime a procedure is activated, its
    names are bound to the same storage locations.
  • Therefore values of local names are retained across activations of a procedure. That is,
    when control returns to a procedure the values of the locals are the same as they were
    when control left the last time.
  • From the type of a name, the compiler decides the amount of storage for the name and
    decides where the activation records go. At compile time, we can fill in the addresses at
    which the target code can find the data it operates on.

4.9.2 STACK ALLOCATION OF SPACE
  • All compilers for languages that use procedures, functions or methods as units of user-
    defined actions manage at least part of their run-time memory as a stack.
  • Each time a procedure is called, space for its local variables is pushed onto a stack, and
    when the procedure terminates, that space is popped off the stack.

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

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

When designing calling sequences and the layout of activation records, the following principles are helpful:

- Values communicated between caller and callee are generally placed at the beginning of the callee’s activation record, so they are as close as possible to the caller’s activation record.
- Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- Items whose size may not be known early enough are placed at the end of the activation record. The most common example is a dynamically sized array, where the value of one of the callee’s parameters determines the length of the array.
- We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.

The calling sequence and its division between caller and callee are as follows.

- The caller evaluates the actual parameters.
- The caller stores a return address and the old value of top_sp into the callee’s activation record. The caller then increments the top_sp to the respective positions.
- The callee saves the register values and other status information.
- The callee initializes its local data and begins execution.

A suitable, corresponding return sequence is:

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

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.
4.9.3 HEAP ALLOCATION

Stack allocation strategy cannot be used if either of the following is possible:

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.
The record for an activation of procedure r is retained when the activation ends.

- Therefore, the record for the new activation q(1, 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.

4.10 Parameter passing:

The method of communication between procedure is through parameters of the called procedure.

Various parameter passing methods:
- call by value: passing r-values
- call by reference: passing l-values
- call by copy-restore: hybrid between by-value and by-reference;
- call by name/ Macro expansion : passing via name substitution;

4.10.1 Call by Value :

Each actual argument is evaluated before call. On entry, the resulting value is copied and bound to the formal parameter; which behaves just like a local variable.

- Advantages:
  - Simple; easy to understand!
- Formal parameters can be used as local variables, Updating them doesn’t affect actuals in calling procedure:

```c
double hyp( double a, double b ) {
    a = a * a;
    b = b * b;
```
return sqrt( a + b ); // use built-in sqrt()
}

Issues:
1. It can be inefficient if value is large
2. It cannot affect calling environment directly
3. It can return result only through (the single) return value.

4.10.2 Call by Reference:

Implemented by passing address of actual parameter:
• On entry, the formal is bound to the address, providing a reference to actual parameter from within the subroutine
• If actual argument doesn’t have an l-value (e.g., “2 + 3”), then either:
  – Forbid in language, i.e. treat as an error; compiler catches this
  - evaluate it into a temporary location and pass its address; but what will be the modified value of actual after the call?
• Advantages:
  – No more large copying
  – Actual parameter can be updated
    - Now swap, etc., work fine!
Accesses are slower: the formal parameter is an address that must be dereferenced to get at the value
• Opportunity for aliasing problems,
  e.g., procedure MatrixMult( a, b, c: matrix )
      ... sets c := a * b;
      MatrixMult( a, b, a ) // oops!

4.10.3 Call by Copy-Restore:

• Each actual argument is evaluated to a value before call
• On entry, value is bound to formal parameter just like a local
• Updating formal parameters doesn’t affect actuals in calling procedure during execution
• Upon exit, the final contents of formals are copied into the actuals
• Thus, behaves like call by reference in most “normal” situations, but may give different results when concurrency or aliasing are involved:
type t is record a, b:integer; end record;
r : t;
procedure foo( s : in out t ) is
begin
  r.a := 2; s.a := s.a + 3;
end foo;
r.a := 1;
foo( r );
print( r.a ); -- what’s the value of r.a?
4.10.4 Call by Name:

- First introduced in Algol-60, first implemented via “thunk” by Ingerman
- Idea derived from substitution or macro-expansion. In C and some other languages, macros are used to define simple functions. The user interface of a macro resembles that of a function.

```c
#define max(a, b) ((a)>(b) ? (a) : (b))
```

```c
v = max(x, y);
```

- When a macro is invoked, each formal parameter is literally replaced by corresponding actual parameter; string substitution:

```
max(x+1, y-2) => ((x+1) > (y-2) ? (x+1) : (y-2))
```

- However, macros and functions are fundamentally different:
  - Macros are invoked at compile-time or pre-compile-time, functions are called at runtime
  - Macros can not contain recursion
  - Call by name is actually “substitution with renaming where necessary”
  - Flexible, but potentially very confusing, and inefficient to implement; requires small “.dll” or piece of code to evaluate actual parameter
  - If language has no updatable variables (as in “pure” functional languages), call by name renders beautifully simple semantics for Function calls.

4.11 Symbol Table

- Stores the symbol of the source program as the compiler encounters them.
- Each entry contains the symbol name plus a number of parameters describing what is known about the symbol. Reserved words (if, then, else, etc.) maybe stored in the symbol table as well
- As a minimum we must be able to
  - INSERT a new symbol into the table
  - RETRIEVE a symbol so that its parameters may be retrieved and/or modified,
  - Query to find out if a symbol is already in the table.
- Each entry can be implemented as a record. Records can have different formats (Variant records in Pascal).

Storing characters

- Method 1: A fixed size space within each entry large enough to hold the largest possible name. Most names will be much shorter than this so there will be a lot of wasted storage
- Method 2: Store all symbols in one large separate array. Each symbol is terminated with an end of symbol mark (EOS). Each symbol table record contains a pointer to the first character of the symbol.
- Method n: modern languages (e.g. Java, C++ std components) has efficient DS, e.g. string or vector

Symbol Table Data Structure

- One Linear list:
Linear list of records is the easiest way to implement a symbol table. The new names are added to the table in the order that they arrive. Whenever a new name is to be added to the table, the table
is first searched linearly or sequentially to check whether or not the name is already present in the table. If the name is not present, then the record for new name is created and added to the list at a position specified by the available pointer

- Easy to implement
- search time will be very long if source has many symbols.

**Hash table:**
- A hash table is a list in which each member is accessed through a key.
- The key is used to determine where to store the value in the table.
- The function that produces a location from the key is called the hash function.
- For example, if it were a hash table of strings, the hash function might compute the sum of the ASCII values of the first 5 characters of the string, modulo the size of the table.

The numerical value of the hashed key gives the location of the member
- Thus, there is no need to search for the member; the hashed key tells where it is located.
- For example, if the string were "return", then the key would be \((114 + 101 + 116 + 117 + 114) \mod 100 = 62\).
- Thus, "return" would be located in position 62 of the hash table.
- Run the symbol name through a hash function to create an index in a table.
- If some other symbol has already claimed the space then rehash with another hash function to get another index, etc.
- **Hash Table must be large enough to accommodate largest number of symbols**

4.12 Dynamic storage allocation techniques:

The techniques needed to implement dynamic storage allocation techniques depends on how the space is deallocated.
ie, implicitly or explicitly

- Explicit allocation of fixed size block
- Explicit allocation of variable size block
- Implicit deallocation

Explicit allocation of fixed size block

- The simplest form dynamic storage allocation.
- The blocks linked together in a list and the allocation and deallocation can done quickly with less or no storage overhead
A pointer available points to the first block in the list of available blocks

Explicit allocation of variable size block

When blocks are allocated & deallocated storage can become fragmented ie, heap may consist alternate blocks that are free & in use
In variable size allocation it will be a problem bcoz we could not allocate a block larger than any free blocks, even though the space is available
The first fit method can be used to allocate variable sized block

When a block of size is allocated it search for the first free block size f>=s. This block is then subdivided in to a used block of size s & a free block of size f-s. Its time consuming;
when a block is deallocated, it check to see if it is next to a free block. If possible, the deallocated is combined with a free block next to it to create larger free block. It helps to avoid fragmentation

Implicit deallocation

Implicit deallocation requires the cooperation between user program & runtime packages. this is implemented by fixing the format of storage blocks

Optional block size
Optional reference count
Optional mark
Pointers to blocks
User information

The first problem is to recognize the block boundaries, for fixed size it is easy.
In variable size block the size of block is kept in an inaccessible storage attached to the block.
The second problem is of recognizing the if a block is in use. Used block can be referred by the user program using pointers. The pointers are kept in a fixed position in the block for the easiness of checking the reference.
Two approaches can be used for implicit deallocation.

1. Reference counts
2. Marking techniques

Reference counts
- We keep track of the number of reference to the present block. If it ever drops to 0, the block is deallocated.
- Maintaining reference counts can be costly in time (the pointer assignment \( p := q \) leads to changes in the reference counts of the blocks pointed by both \( p \) and \( q \)).
- Reference counts are best if there is no cyclical reference occurs.

Marking techniques
- Here the user program suspends temporarily and uses the frozen pointers to determine the used blocks. This approach requires all the pointers to the heap to be known. (Conceptually, it's like pouring paint to the heap through the pointers.)
  First we go through the heap and mark all the blocks unused. Then we follow the pointers and mark all the reachable blocks as used. Then sequential scan of heap collects all the blocks still marked unused.
UNIT V  
CODE OPTIMIZATION AND CODE GENERATION


INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

- Optimizations are classified into two categories. They are
  - Machine independent optimizations:
  - Machine dependant optimizations:

Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependant optimizations:

- Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- Simply stated, the best program transformations are those that yield the most benefit for the least effort.

- The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.

- A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.

- The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler

Organization for an Optimizing Compiler:
Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
  - control flow graph
  - Call graph
- Data flow analysis (DFA) is the process of ascerting and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

**PRINCIPAL SOURCES OF OPTIMISATION**

- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

**Function-Preserving Transformations**

- There are a number of ways in which a compiler can improve a program without changing the function it computes.
- The transformations
  - ✓ Common sub expression elimination,
  - ✓ Copy propagation,
  - ✓ Dead-code elimination, and
  - ✓ Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

**Common Sub expressions elimination:**

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.
- For example
  - \( t_1: = 4*i \)
  - \( t_2: = a \{ t_1 \} \)
  - \( t_3: = 4*j \)
  - \( t_4: = 4*i \)
  - \( t_5: = n \)
  - \( t_6: = b \{ t_4 \} + t_5 \)
The above code can be optimized using the common sub-expression elimination as
\[ t_1 = 4 \times i \]
\[ t_2 = a \[ t_1 \] \]
\[ t_3 = 4 \times j \]
\[ t_5 = n \]
\[ t_6 = b \[ t_1 \] + t_5 \]

The common sub expression \( t_4 = 4 \times i \) is eliminated as its computation is already in \( t_1 \). And value of \( i \) is not been changed from definition to use.

### Copy Propagation:

- Assignments of the form \( f := g \) called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use \( g \) for \( f \), whenever possible after the copy statement \( f := g \). Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate \( x \).
- For example:

\[
x = P_i;
\]
\[
\ldots
\]
\[
A = x \times r \times r;
\]

The optimization using copy propagation can be done as follows:

\[
A = P_i \times r \times r;
\]

Here the variable \( x \) is eliminated.

### Dead-Code Eliminations:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute
values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```c
i=0;
if(i=1)
{
    a=b+5;
}
```

Here, ‘if’ statement is dead code because this condition will never get satisfied.

- **Constant folding:**
  - We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
  - One advantage of copy propagation is that it often turns the copy statement into dead code.
  - For example, 
    a=3.14157/2 can be replaced by 
    a=1.570 there by eliminating a division operation.

- **Loop Optimizations:**
  - We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
  - Three techniques are important for loop optimization:
    - code motion, which moves code outside a loop;
    - Induction-variable elimination, which we apply to replace variables from inner loop.
    - Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

- **Code Motion:**
  - An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```c
while (i <= limit-2) /* statement does not change limit*/
```
Code motion will result in the equivalent of
t= limit-2;
while (i<=t) /* statement does not change limit or t */

- **Induction Variables**:
  - Loops are usually processed inside out. For example consider the loop around B3.
  - Note that the values of j and t₄ remain in lock-step; every time the value of j decreases by 1, that of t₄ decreases by 4 because 4*j is assigned to t₄. Such identifiers are called induction variables.
  - When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t₄ completely; t₄ is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

**Example:**
As the relationship t₄:=4*j surely holds after such an assignment to t₄ in Fig. and t₄ is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t₄:= 4*j-4 must hold. We may therefore replace the assignment t₄:= 4*j by t₄:= t₄-4. The only problem is that t₄ does not have a value when we enter block B3 for the first time. Since we must maintain the relationship t₄=4*j on entry to the block B3, we place an initializations of t₄ at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig.

The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

- **Reduction In Strength:**
  - Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
  - For example, x² is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.
5.2 THE DAG REPRESENTATION FOR BASIC BLOCKS

- A DAG for a basic block is a **directed acyclic graph** with the following labels on nodes:
  1. Leaves are labeled by unique identifiers, either variable names or constants.
  2. Interior nodes are labeled by an operator symbol.
  3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.
- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub-expressions

**Algorithm for construction of DAG**

**Input:** A basic block
Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values. Case (i) \( x := y \text{ OP } z \)

Case (ii) \( x := \text{ OP } y \)

Case (iii) \( x := y \)

Method:

Step 1: If \( y \) is undefined then create node(y).

If \( z \) is undefined, create node(z) for case(i).

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let \( n \) be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node \( n \) will be node(y).

Step 3: Delete \( x \) from the list of identifiers for node(x). Append \( x \) to the list of attached identifiers for the node \( n \) found in step 2 and set node(x) to \( n \).

Example: Consider the block of three-address statements:

1. \( t_1 := 4* i \)
2. \( t_2 := a[t_1] \)
3. \( t_3 := 4* i \)
4. \( t_4 := b[t_3] \)
5. \( t_5 := t_2*t_4 \)
6. \( t_6 := \text{prod}+t_5 \)
7. \( \text{prod} := t_6 \)
8. \( t_7 := i+1 \)
9. \( i := t_7 \)
10. if \( i \leq 20 \) goto (1)

Stages in DAG Construction
(a) t1
4
I0

Statement (1)

(b) t2

Statement (2)

(c) t2

node for 4*I0 exist already, hence attach identifier t3 to the existing node for Statement (3)

(d) t2
4
I0

Statement (4)
Application of DAGs

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.
Rearranging the order
The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:
\[
\begin{align*}
t_1 & : = a + b \\
t_2 & : = c + d \\
t_3 & : = e - t_2 \\
t_4 & : = t_1 - t_3 \\
\end{align*}
\]

Generated code sequence for basic block:

\[
\begin{align*}
& \text{MOV } a, R_0 \\
& \text{ADD } b, R_0 \\
& \text{MOV } c, R_1 \\
& \text{ADD } d, R_1 \\
& \text{MOV } R_0, t_1 \\
& \text{MOV } e, R_0 \\
& \text{SUB } R_1, R_0 \\
& \text{MOV } t_1, R_1 \\
& \text{SUB } R_0, R_1 \\
& \text{MOV } R_1, t_4 \\
\end{align*}
\]

Rearranged basic block:
Now \( t_1 \) occurs immediately before \( t_4 \).

\[
\begin{align*}
t_2 & : = c + d \\
t_3 & : = e - t_2 \\
t_1 & : = a + b \\
t_4 & : = t_1 - t_3 \\
\end{align*}
\]

Revised code sequence:

\[
\begin{align*}
& \text{MOV } c, R_0 \\
& \text{ADD } d, R_0 \\
& \text{MOV } a, R_0 \\
& \text{SUB } R_0, R_1 \\
& \text{MOV } a, R_0 \\
& \text{ADD } b, R_0 \\
& \text{SUB } R_1, R_0 \\
& \text{MOV } R_0, t_4 \\
\end{align*}
\]

In this order, two instructions \text{MOV } R_0, t_1 \text{ and } \text{MOV } t_1, R_1 have been saved.

A Heuristic ordering for Dags

The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.
The algorithm shown below produces the ordering in reverse.

Algorithm:

1) while unlisted interior nodes remain do begin
2) select an unlisted node n, all of whose parents have been listed;
3) list n;
4) while the leftmost child m of n has no unlisted parents and is not a leaf do
   begin
      5) list m;
      6) n := m
   end
end

Example: Consider the DAG shown below:

Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3).

Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6).

Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus we select anew n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that.

The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

\[ t_8 := d + e \]
\[ t_6 := a + b \]
\[ t_5 := t_6 \]
\[ t_4 := t_5 * t_8 \]
t3 := t4 - e  
= t6 + t4 t1 := t2  
* t3

This will yield an optimal code for the DAG on machine whatever be the number of registers.

5.3 OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are:

- Structure-Preserving Transformations
- Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- Common sub-expression elimination
- Dead code elimination
- Renaming of temporary variables
- Interchange of two independent adjacent statements.

➢ Common sub-expression elimination:

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it’s referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

a: = b + c  
b: = a - d  
c: = b + c  
d: = a - d

The 2\textsuperscript{nd} and 4\textsuperscript{th} statements compute the same expression: b+c and a-d

Basic block can be transformed to

a: = b + c  
b: = a - d  
c: = a  
d: = b

➢ Dead code elimination:

It’s possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

➢ Renaming of temporary variables:
• A statement $t:=b+c$ where $t$ is a temporary name can be changed to $u:=b+c$ where $u$ is another temporary name, and change all uses of $t$ to $u$.
• In this we can transform a basic block to its equivalent block called normal-form block.

➢ Interchange of two independent adjacent statements:

• Two statements

$$t_1:=b+c$$
$$t_2:=x+y$$

can be interchanged or reordered in its computation in the basic block when value of $t_1$ does not affect the value of $t_2$.

Algebraic Transformations:

• Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
• Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression $2*3.14$ would be replaced by $6.28$.
• The relational operators $\leq$, $\geq$, $<$, $>$, $+$ and $=$ sometimes generate unexpected common sub expressions.
• Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

$$a:=b+c$$
$$e:=c+d+b$$

the following intermediate code may be generated:

$$a:=b+c \ t:=c+d \ e:=t+b$$

• Example:

$$x:=x+0 \text{ can be removed}$$
$$x:=y**2 \text{ can be replaced by a cheaper statement } x:=y*y$$

The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate $x*y-x*z$ as $x*(y-z)$ but it may not evaluate $a+(b-c)$ as $(a+b)-c$

LOOPS IN FLOW GRAPH

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.
**Dominator s:**

In a flow graph, a node \( d \) dominates node \( n \), if every path from initial node of the flow graph to \( n \) goes through \( d \). This will be denoted by \( d \) dom \( n \). Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

**Example:**

*In the flow graph below,
*Initial node, node1 dominates every node.
*node 2 dominates itself
*node 3 dominates all but 1 and 2.
*node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10.
*node 8 dominates 8,9 and 10.
*node 9 and 10 dominat es only themselves.

The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node \( d \) dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of \( n \) on any path from the initial node to \( n \).
- In terms of the dom relation, the immediate dominator \( m \) has the property is \( d \neq !n \) and \( d \) dom \( n \), then \( d \) dom \( m \).

**Natural Loop:**

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.

- The properties of loops are
  - A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
  - There must be at least one way to iterate the loop(i.e.)at least one path back to the header.

- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If \( a \rightarrow b \) is an edge, \( b \) is the head and \( a \) is the tail. These types of edges are called as back edges.

  - Example:

    In the above graph,
    
    \[ 7 \rightarrow 4 \quad 4 \text{ DOM } 7 \]
The above edges will form loop in flow graph.

- Given a back edge \( n \to d \), we define the natural loop of the edge to be \( d \) plus the set of nodes that can reach \( n \) without going through \( d \). Node \( d \) is the header of the loop.

**Algorithm:** Constructing the natural loop of a back edge.

**Input:** A flow graph \( G \) and a back edge \( n \to d \).

**Output:** The set \( \text{loop} \) consisting of all nodes in the natural loop \( n \to d \).

**Method:** Beginning with node \( n \), we consider each node \( m \) that we know is in loop, to make sure that \( m \)'s predecessors are also placed in loop. Each node in loop, except for \( d \), is placed once on stack, so its predecessors will be examined. Note that because \( d \) is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach \( n \) without going through \( d \).

**Procedure** `insert(m);`

if `m` is not in loop **then begin**

- `loop := loop U \{m\};`
- `push m` onto stack

**end;**

`stack := empty;`
loop := \{d\};
insert(n);
while stack is not empty do begin
    pop m, the first element of stack, off stack;
    for each predecessor p of m do insert(p)
end

Inner loop:

- If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

- Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheater.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.

Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

**Definition:**

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, forward edges and back edges, with the following properties.

- The forward edges from an acyclic graph in which every node can be reached from initial node of G.
The back edges consist only of edges where heads dominate their tails.

Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.
- The remaining edges are forward edges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

**PEEPHOLE OPTIMIZATION**

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.

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

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

- We shall give the following examples of program transformations that are characteristic of peephole optimizations:
  - Redundant-instructions elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms
  - Unreachable Code

**Redundant Loads And Stores:**

If we see the instructions sequence

1. MOV R₀, a
2. MOV a, R₀

we can delete instructions (2) because whenever (2) is executed, (1) will ensure that the value
of a is already in register R₀. If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

**Unreachable Code:**

- Another opportunity for peephole optimizations is the removal of unreachable instructions.
  An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable `debug` is 1. In C, the source code might look like:

```c
#define debug 0
...
If ( debug ) {
    Print debugging information
}
```

- In the intermediate representations the if-statement may be translated as: If debug =1 goto L2
goto L2
L1: print debugging information
L2: .................................(a)

- One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of `debug`, (a) can be replaced by:

```c
If debug ≠1 goto L2
    Print debugging information
L2: .................................(b)
```

- As the argument of the statement of (b) evaluates to a constant `true` it can be replaced by

```c
If debug ≠0 goto L2
    Print debugging information
L2: .................................(c)
```
- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

**Flows-Of-Control Optimizations:**

- The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

  ```
  goto
  L1
  ...
  ...
  L1: goto L2
  ```

  by the sequence

  ```
  goto L2
  ...
  ...
  L1: goto L2
  ```

- If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump. Similarly, the sequence

  ```
  if a < b goto L1
  ...
  ...
  L1: goto L2
  ```

  can be replaced by

  ```
  If a < b goto L2
  ...
  ...
  L1: goto L2
  ```
Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

L1: if a < b goto L2

L3: .........................................(1)

May be replaced by

If a < b
goto L2
goto L3

L3

........

L3: .................................(2)

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1). Thus (2) is superior to (1) in execution time

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

\[ x := x + 0 \]

Or

\[ x := x \times 1 \]

Are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, \( x^2 \) is invariably cheaper to implement as \( x \times x \) than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

\[ X^2 \rightarrow X \times X \]
Use of Machine Idioms:

- The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value.
- The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like \( i := i + 1 \):
  \[
  i := i + 1 \rightarrow i++
  \]
  \[
  i := i - 1 \rightarrow i--
  \]

5.4 INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation, compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions”, such as knowing where a variable like debug was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form:

\[
\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])
\]

This equation can be read as “the information at the end of a statement is either generated within the statement, or enters at the beginning and is not killed as control flows through the statement.”

- The details of how data-flow equations are set and solved depend on three factors.

  ✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining \( \text{out}[s] \) in terms of \( \text{in}[s] \), we need to proceed backwards and define \( \text{in}[s] \) in terms of \( \text{out}[s] \).

  ✓ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write \( \text{out}[s] \) we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

  ✓ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:
Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

Now let us take a global view and consider all the points in all the blocks. A path from \( p_1 \) to \( p_n \) is a sequence of points \( p_1, p_2, \ldots, p_n \) such that for each \( i \) between 1 and \( n-1 \), either

1. \( p_i \) is the point immediately preceding a statement and \( p_{i+1} \) is the point immediately following that statement in the same block, or
2. \( p_i \) is the end of some block and \( p_{i+1} \) is the beginning of a successor block.

Reaching definitions:

- A definition of variable \( x \) is a statement that assigns, or may assign, a value to \( x \). The most common forms of definition are assignments to \( x \) and statements that read a value from an i/o device and store it in \( x \).

- These statements certainly define a value for \( x \), and they are referred to as **unambiguous** definitions of \( x \). There are certain kinds of statements that may define a value for \( x \); they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of \( x \) are:
  - A call of a procedure with \( x \) as a parameter or a procedure that can access \( x \) because \( x \) is in the scope of the procedure.
  - An assignment through a pointer that could refer to \( x \). For example, the assignment \(*q := y*\) is a definition of \( x \) if it is possible that \( q \) points to \( x \). We must assume that an assignment through a pointer is a definition of every variable.

- We say a definition \( d \) reaches a point \( p \) if there is a path from the point immediately following \( d \) to \( p \), such that \( d \) is not “killed” along that path.

Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

  \[
  S \quad \text{id:} = E | S ; S | \text{if } E \text{ then } S \text{ else } S | \text{do } S \text{ while } E E \quad \text{id + id} | \text{id}
  \]

- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.
ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase:

1. Input to code generator
2. Target program
3. Memory management
4. Instruction selection
5. Register allocation
6. Evaluation order

1. Input to code generator:
   - The input to the code generation consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table to determine run-time addresses of the data objects denoted by the names in the intermediate representation.
   
   - Intermediate representation can be:
     a. Linear representation such as postfix notation
     b. Three address representation such as quadruples
     c. Virtual machine representation such as stack machine code
     d. Graphical representations such as syntax trees and dag
   
   - Prior to code generation, the front end must be scanned, parsed and translated into intermediate representation along with necessary type checking. Therefore, input to code generation is assumed to be error-free.

2. Target program:
   - The output of the code generator is the target program. The output may be:
     a. Absolute machine language
        - It can be placed in a fixed memory location and can be executed immediately.
     b. Relocatable machine language
        - It allows subprograms to be compiled separately.
     c. Assembly language
        - Code generation is made easier.

4. Memory management:
   - Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
   
   - It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
   
   - Labels in three-address statements have to be converted to addresses of instructions. For example, 
     \[ j : \texttt{goto} i \] generates jump instruction as follows:
     - if \( i < j \), a backward jump instruction with target address equal to location of code for quadruple \( i \) is generated.
if $i > j$, the jump is forward. We must store on a list for quadruple $i$ the location of the first machine instruction generated for quadruple $j$. When $i$ is processed, the machine locations for all instructions that forward jumps to $i$ are filled.

5. Instruction selection:

- The instructions of target machine should be complete and uniform.

- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.

- The quality of the generated code is determined by its speed and size.

- The former statement can be translated into the latter statement as shown below:

\[
\begin{align*}
a &= b + c \\
d &= a + c
\end{align*}
\]

\[
\begin{align*}
\text{MOV } b, R_0 \\
\text{ADD } c, R_0 \\
\text{MOV } R_0, a \\
\text{MOV } a, R_0 \\
\text{ADD } e, R_0 \\
\text{MOV } R_0, d
\end{align*}
\]

This can be eliminated