UNIT-I

Syllabus:
Introduction: The Structure of a Compiler, Phases of Compilation, The Translation Process, Major Data Structures in a Compiler, Bootstrapping and Porting.
Lexical Analysis (Scanner): The Role of the Lexical Analyzer, Input Buffering, Specification of Tokens, Recognition of Tokens, The Lexical Analyzer Generator Lex.
Objective: Basic ideas in concepts of designing and implementing translators forvarious languages and system building tools like LEX is introduced for lexical analysis.
Outcome:

Introduction

In this new era, all are using software application in their daily lives. Software applications are written in some programming languages. These programs must be converted into a form which can be executed by a computer. Software that does this conversion is called as translators.

Types of Translators

1. Interpreter
2. Assembler
3. Compiler

1) Interpreter is one of the translators that translate high level language to low level language. An interpreter reads the source code one instruction or line at a time, converts this line into machine code and executes it. The machine code is then discarded and the next line is read.

2) Assembler is a software or a tool that translates Assembly language to machine code. So, an assembler is a type of a compiler, and the source code is written in Assembly language. Assembly is a human readable language, but it typically has a one to one relationship with the corresponding machine code.

3) Compiler is a program that translates one language(source code) to another language (target code). During this translation, reporting errors to user.

Difference between Compiler and Interpreter

Compiler scans the entire program once and then converts it into machine language which can then be executed by computer's processor. In short compiler translates the entire program in one go and then executes it. Interpreter on the other hand first converts high level language into an intermediate code and then executes it line by line.

The execution of program is faster in compiler than interpreter as in interpreter code is executed line by line.

Compiler generates error report after translation of entire code whereas in case of interpreter once an error is encountered it is notified and no further code is scanned.

Compiler: A compiler is a system software which is used to translate the program written in one programming language into machine understandable language to be executed by a computer. Compiler should make the target code efficient and optimized in terms of time and space.

A compiler reads a program written in one language called as the source language and translate it into an equivalent program in another language called as target language. In this conversion process compiler detects and reports any syntactical errors in the source program.

$$\begin{tikzcd}
 & & \boxed{\text{Input}} \arrow[d] & \\
\boxed{\text{Source Program}} \arrow[r] & \boxed{\text{Compiler}} \arrow[r] & \boxed{\text{Target Program}} \arrow[r] & \boxed{\text{Output}}
\end{tikzcd}$$

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.

First computers of late 1940s were programmed in machine language. They are soon replaced by assembly language instructions and memory locations are specified with symbolic names. An assembler translates the symbolic assemblycode into equivalent machine code. Assembly language is an improved programming language but still it is machine dependent. Later high level languages are introduced, where programs are written in English related statements.

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user. The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Brief History

- The term "compiler" was coined in the early 1950s by Grace Murray Hopper. Translation was then viewed as the "compilation" of a sequence of routines selected from a library
- The first compiler of the high-level language FORTRAN was developed between 1954 and 1957 at IBM by a group led by John Backus
- The study of the scanning and parsing problems was pursued in the 1960s and 1970s and led fairly to a complete solution
- The development of methods for generating efficient target code, known as optimization techniques, is still ongoing research
- Compiler technology was also applied in rather unexpected areas:

Classification of Compilers

Compilers are sometimes classified as single-pass, multi-pass, load-and-go, debugging, or optimizing, depending on how they have been constructed or on what function they are supposed to perform.

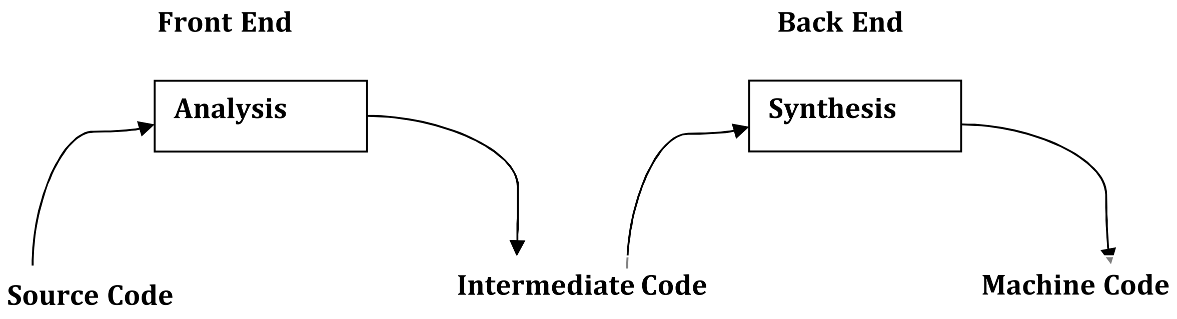
The processes of translation of program to machine understandable code is dividedinto two parts.
1) Analysis part
2) Synthesis part

1) Analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The analysis phase contains the following stages of compiler.

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis

2) Synthesis part constructs the desired target program from the intermediate representation of the source program.

- Intermediate Code Generator
- Code Optimizer
- Code Generator

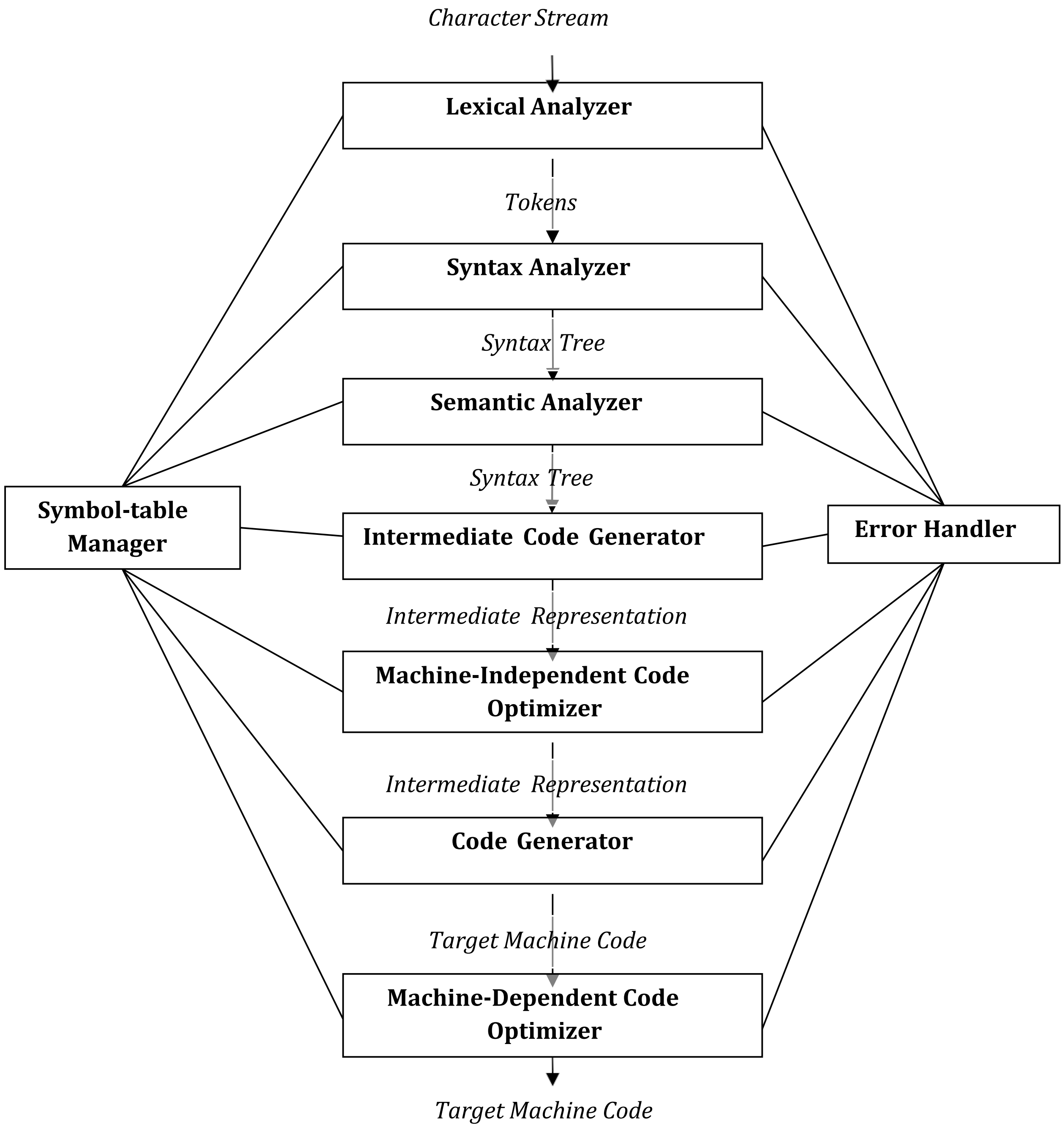


Properties of a good compiler

- Reliability: It must generate correct machine code.
- Faithfulness: Compiler itself must be a bug-free.
- Compilation Speed: Compiler mut run fast(Compilation time must be proportional to program size)
- Diagnostics: It must give good diagnostics and error messages.
- Error handling: must report error to user.
- Implementation and Maintenance: portable and must work well with existingdebuggers.
- Good human interface: Easy to use

The Structure of a Compiler

Compiler operates as a sequence of phases, each phase transforms one representation of the source program to another. A typical decomposition of a compiler into phases is shown in the below figure.



Phases of compiler

The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

Some compilers have a machine-independent optimization phase between the frontend and the back end. The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target code. Since optimization is optional, one or the other of the two optimization phases shown in the above figure may be ignored.

Phases of Compilation

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation
- Symbol Table Management
- Error Handling

Lexical Analysis:
The first phase of a compiler is called lexical analysis or scanning. The lexicalanalyzer reads the stream of characters that make up the source program starting from left to right and groups the characters into meaningful sequences called *lexemes*. For each lexeme, the lexical analyzer produces a *token* of the form

<token-name, attribute-value>

- token-name is an abstract symbol that is used during syntax analysis.
- attribute-value points to an entry in the symbol tableThese

tokens are passed to syntax analyzer.

Token: It represents a logically cohesive sequence of characters such as keywords, operators, identifiers, special symbols etc.

Example: p = i + r * 60

- Here p, =, i, +, r, *, 60 are all separate lexemes.
- <id,1> <=> <id,2> <+> <id,3> <*> <60> are the tokens generated.

Syntax Analysis:
The second phase of the compiler is syntax analysis or parsing. The parser uses the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments(operands) of theoperation.

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60

Semantic Analysis:
It is the third phase of the compiler. The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. It performs type conversion of all the data types into real data types.

An important part of semantic analysis is type checking. Some language specification may permit some type conversions called coercions. Example ***inttofloat.***

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60

Intermediate Code Generation
It is the fourth phase of the compiler. In the process of translating a source program into target code, a compiler may construct one or more intermediaterepresentations.

After semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation of the source program.

- three-address code is one of the intermediate code representations.

three-address code consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60

Three-address code for the above code is

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2id1
= t3
```

Code Optimization:
It is the fifth phase of the compiler. It gets the intermediate code as input and produces optimized intermediate code as output.

The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code is generated. Better means faster,shorter code, or target code that consumes less power.

Example:
For p = i + r * 60, the syntax tree is
<id,1> = <id,2> + <id,3> * 60

Optimized code is:

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

Code Generation:
It is the sixth phase of the compiler. The code generator takes intermediate representation/optimized machine independent representation as input and maps

it into the target language. If the target language is machine code, registers or memory locations are selected for each of the variables used in the program. Then,the intermediate instructions are translated into sequences of machine instructions that perform the same task.

Example:

For p = i + r * 60, the syntax tree is

<id,1> = <id,2> + <id,3> * 60

using registers R1 and R2, the machine code is:LDF R2,

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

Symbol Table Manager:

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

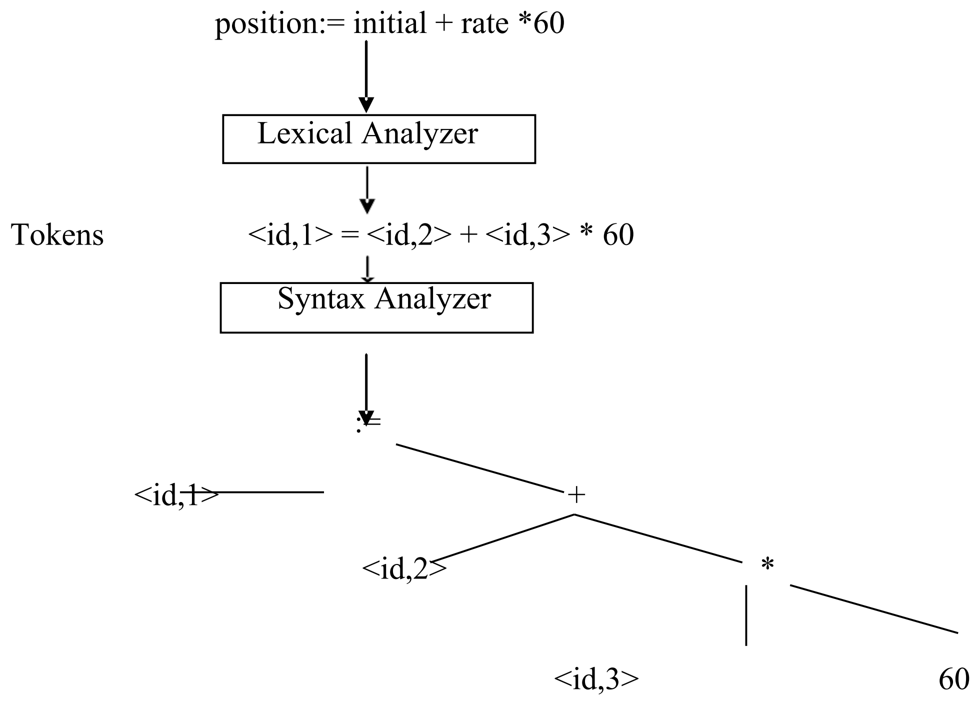
Error Handling:

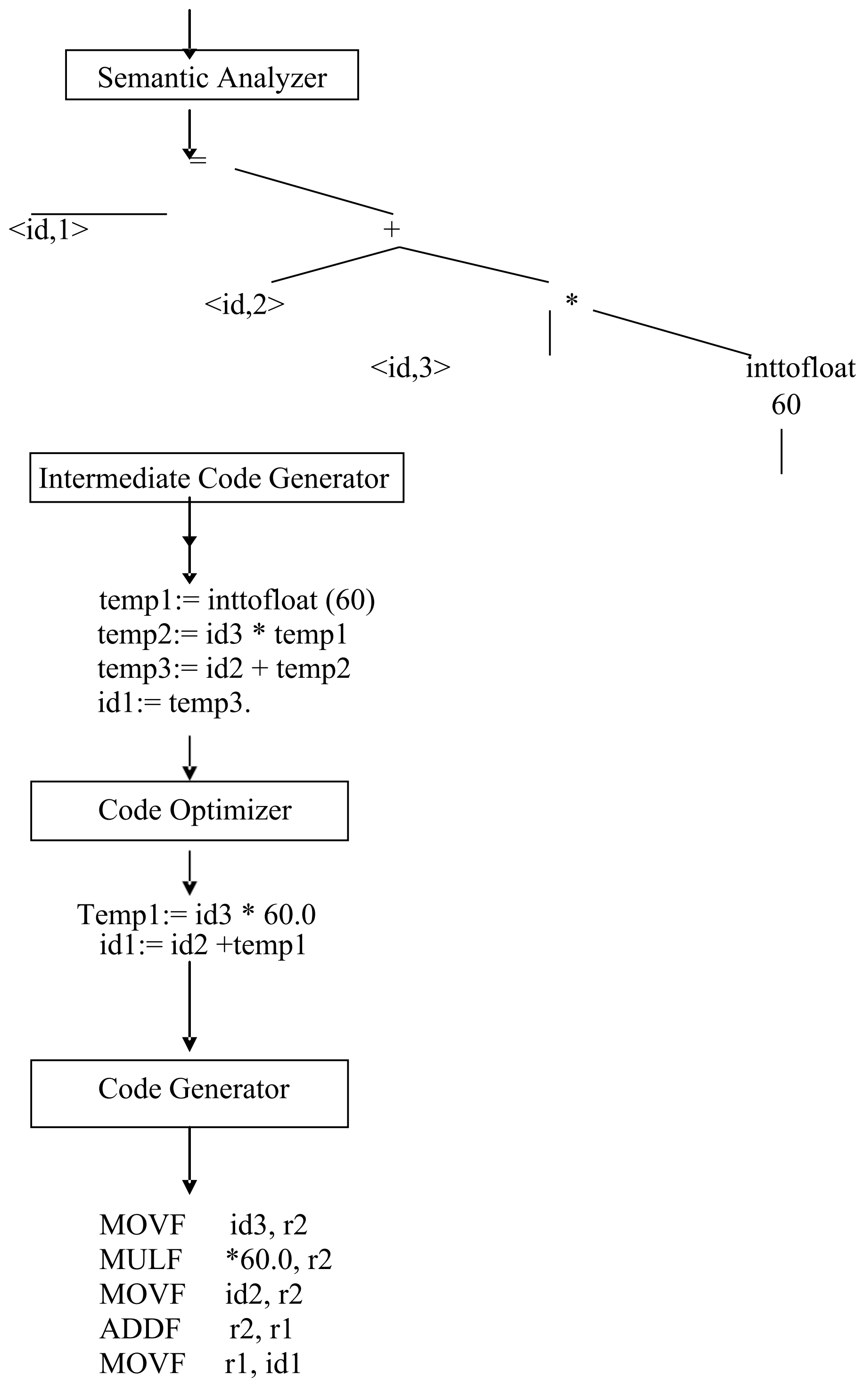
One of the most important functions of a compiler is the detection and reporting of errors in the source program. The error message should allow the programmer to determine exactly where the errors have occurred.

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

The Translation Process

To illustrate the translation of source code through each phase, consider the statement ***position = initial + rate * 60***





```
temp1:= inttofloat (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.
```

```
Temp1:= id3 * 60.0
 id1:= id2 +temp1
```

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

Errors Encountered in different Phases:

Each of the six phases (but mainly the front end) of a compiler detect errors. Ondetecting an error, the compiler must:

- report the error in a helpful way for better understanding,
- correct the error if possible, and
- continue processing (if possible) after the error to look for further errors.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

Types of Error: Two types of error usually encountered in programs.

Syntax errors are errors in the program text. they may be either lexical orgrammatical.

- A lexical error is a mistake in a lexeme, for examples, typing *tehn* instead of *then*, or missing off one of the quotes in a literal.
- A grammatical error is a one that violates the (grammatical) rules of thelanguage, for example if x = 7 y := 4 (missing semicolon).

Syntax errors must be detected by a compiler and at least reported to the user (in ahelpful way). If possible, the compiler should make the appropriate correction(s).

Semantic errors are mistakes concerning the meaning of a program construct;they may be either type errors, logical errors, or run-time errors.

- Type errors occur when an operator is applied to an argument of the wrongtype, or to the wrong number of arguments.
- Logical errors occur when a badly conceived program is executed, forexample:
  while x = y do ...
  when x and y initially have the same value and the body of loop need not change the value of either x or y.
- Run-time errors are errors that can be detected only when the program isexecuted.
  for example: var x : real; readln(x); writeln(1/x)
  which would produce a run time error if the user input 0.

Semantic errors are much harder and sometimes impossible for a computer todetect.

Compiler-Construction Tools

Software development environments contains tools like language editors, debuggers, version managers, profilers, test harnesses, and so on are used to construct compilers. In addition to general software-development tools, other more specialized tools are created to implement various phases of a compiler. They are

1. **Parser generators:** It automatically produces syntax analyzers from a grammatical description of a programming language.
2. **Scanner generators:** It produces lexical analyzers from a regular-expression description of the tokens of a language.
3. **Syntax-directed translation engines**: They produce collections of routines for walking a parse tree and generating intermediate code.
4. **Code-generator:** It produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine languagefor a target machine.
5. **Dataflow analysis engines:** It facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Dataflow analysis is a key part of code optimization.
6. **Compiler-construction toolkits:** They provide an integrated set of routines for constructing various phases of a compiler.

Major Data Structures in a Compiler:

Algorithms used in individual phases of a compiler frequently interacts with data structures for their efficient implementation. so that, a compiler compiles a program in O(n) time irrespective of the size of the program. Few data structures that are used in different phases are:

Tokens: Scanner collects characters/lexeme into a token. It represents the token symbolically as a value of an enumerated data type representing a set of tokens of the source language. Sometimes, it is necessary to preserve the character string itself or other information derived from it. In most languages the scanner needs to generate one token at a time (single symbol lookahead). So, a single global variable can be used to hold the token information. In other cases, an array is required to hold the lexeme or token.

Syntax Tree: Parser generates syntax tree. The syntax tree is constructed as a standard pointer-based structure that is dynamically allocated. Entire tree can be kept as a single variable pointing to the root. Each node is a record. Its fieldsrepresent the information collected by the parser and the semantic analyzer.

Symbol Table: Symbol table is an important data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.

Symbol table is used by both the analysis and the synthesis parts of a compiler.

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

Scanner, parser may enter identifiers into table, semantic analyzer will add data type and other information to identifiers.

Literal Table: The Literal Table Stores constants and strings used in the program.One literal table applies globally to the entire program
Used by the code generator to:

- Assign addresses for literals
- Enter data definitions in the target code file

Avoids the replication of constants and strings. Quick insertion and lookup areessential. Deletion is not allowed.

Intermediate Code:
Depending on the kind of intermediate code, it may be kept in an array of textstrings, a temporary text file, Linked list of structures.

Temporary Files:
Computers did not have enough memory for the entire program to be kept in memory during compilation. This was solved by using temporary files to hold theproducts of intermediate steps.

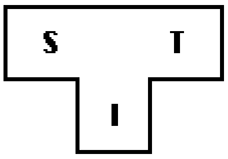
Now a days, memory constrains are not at all a problem. Occasionally, compilersgenerate intermediate files during some of the steps.

Bootstrapping and Porting:

A compiler is characterized by three languages:

- Source Language **(S)**: Language in which programs are written
- Target Language **(T)**: Language understood by the machine. i.e., **S** is to be translated into **T** with the help of **I**
- Implementation Language **(I)**: Language used to write translator/compiler

Compilers are represented in the form of a T-diagram or S_{IT}



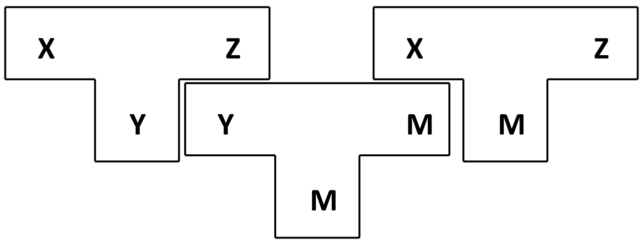
Developing translator for a new language from scratch is a non-trivial exercise (because, we have to use machine language for writing thecompiler).

So, when a translator is required for an old language on a new machine, or a new language on an old machine, use of existing compilers on either machine is the best choice for developing. i.e., write the compiler in another language for which a compiler already exists.

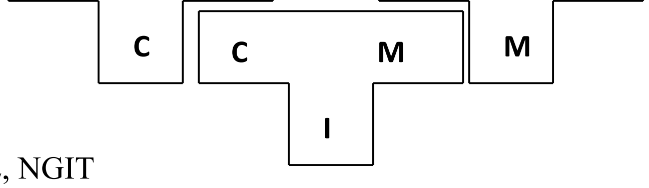
Developing a compiler for a new language by using an existing compiler is called ***bootstrapping.*** Bootstrapping is the process by which a simple language is usedto translate more complicated programs, which intern may handle an even more complicated program.

Bootstrapping is used to create compilers and to move them from one machine to another by modifying the back end.

For example: To write a compiler for new language X and the implementation language of this compiler is say Y and the target code being generated is in language Z. That is, we create XYZ. Now if Y runs on machine M and generates code for M then it is denoted as YMM. Now if we run XYZ using YMM then we get a compiler XMZ. That means a compiler for source language X that generates a target code in language Z and which runs on machine M.



Development of Pascal translator for C++ language is done by bootstrapping Pascal translator for C language with C language translator for M.



Porting:

The process of modifying an existing compiler to work on a new machine is oftenknown as porting the compiler.

To develop a compiler for new hardware machine from an existing compiler, change the synthesis part of the compiler because, synthesis part is machine dependent part. This is called porting.

Quick and dirty Compiler:

Native Compiler: Native compiler are compilers that generates code for the same Platform on which it runs. It converts high language into computer’s native language.

Cross Compiler: A cross compiler is a compiler capable of creating executable code for a platform other than the one on which the compiler is running.
Bootstrap Compiler**.**

Lexical Analysis (Scanner)

In the first phase of compilation process, the source program is divided into characters or sequence of characters called as tokens. Tokens are like character or words of natural language. Each token represents a unit of information in the source program.
In the source program the following elements are considered as tokens.

- Keywords/Reserve words: predefined words of the language (Lexemes)
- Identifiers: use defined strings (ID)
- Special Symbols

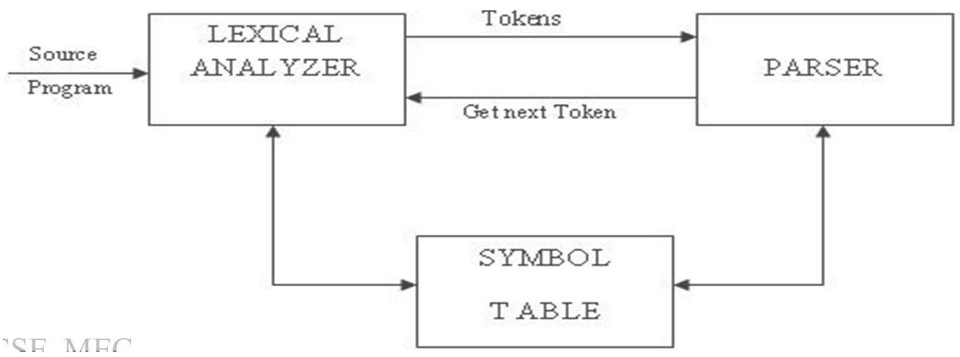
Any value associated to a token is called as an attribute of the token. Attribute maybe a string or numerical.

The ***getNextToken*** command, causes the lexical analyzer to read characters from its input until it can identify the next lexeme and produce for it the next token, which it returns to the parser.

The Role of the Lexical Analyzer

The job of the lexical analyzer (scanner) is to read the source program character by character and form them into logical units called as tokens. These tokens are given as inputs to next phase of compiler. Scanner rarely converts the entire program intotokens at once, but conversion always depends on the parser.

Lexical analyzer interacts with the symbol table when it discovers a lexeme constituting an identifier and enters that lexeme into the symbol table.



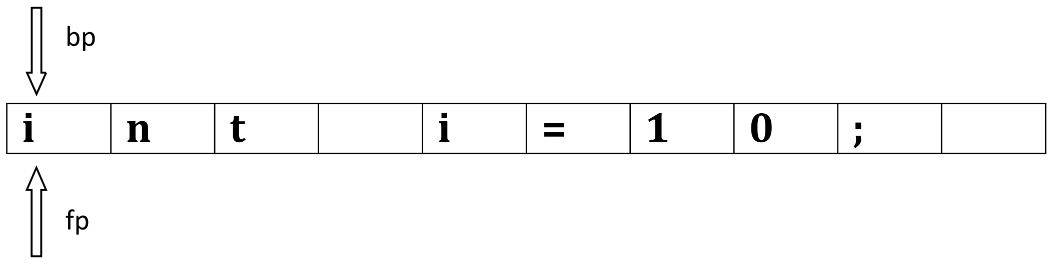
Scanner performs the following tasks:

- identification of lexemes
- stripping out comments and whitespace (blank, newline, tab, and perhapsother characters that are used to separate tokens in the input).
- correlating error messages generated by the compiler with the sourceprogram

Input Buffering:

The lexical analyzer scans the input from left to right one character at a time. Ituses two pointers begin ptr(bp) and forward ptr(fp) to keep track of the pointer of the input scanned.

The forward ptr(fp) moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme. Once a blank space(whitespace) is encountered, lexeme is identified and whitespace is ignored, them both pointers are placed at the next character.



Input character are always read from secondary storage, but this reading is costly. To speed up the scanning process, buffering technique is used. A block of data is first read into a buffer and lexical analysis process id continued on buffer.

Buffering techniques:

1. Buffer (one or two buffers are used)
2. Sentinels

Buffering:

One Buffer Scheme: In this scheme, only one buffer is used to store the input string. The problem with this scheme is that if lexeme is very long then it crosses the buffer boundary, to scan rest of the lexeme the buffer has to be refilled.

Two Buffer Scheme: To overcome the problem of one buffer scheme, two buffers areused to store the input string. The first buffer and second buffer are scanned alternately.

Initially both the ***bp*** and ***fp*** are pointing to the first character of first buffer. Thenthe ***fp*** moves towards right in search of end of lexeme. As soon as blank characteris recognized, the string between bp and ***fp*** is identified as corresponding token. To identify the boundary of first buffer, end of buffer character should be placed at the end first buffer.

Sentinel:
Special character introduced at the end of the buffer is called as Sentinel which isnot a part of program. ***eof*** is the natural choice for sentinel.

Specification of Tokens:

Regular expressions are an important notation for specifying lexeme patterns.

Strings and Languages:

An alphabet is any finite set of symbols. Examples of symbols are letters, digits, and punctuation. The set {0, 1} is the binary alphabet. ASCII and Unicode are important examples of an alphabet. A string over an alphabet is a finite sequence of symbols drawn from that alphabet. In language theory, the terms "sentence" and "word" are often used as synonyms for "string".

A language is any countable set of strings over some fixed alphabet including null set "ø" and set with empty character {ε}.

1. ***Alphabets***: Any finite set of symbols

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

3. ***Special symbols***: A typical high-level language contains the following symbols:

Arithmetic Symbols	Addition(+), Subtraction(-), Multiplication(*), Division(/)
Punctuation	Comma(,), Semicolon(;), Dot(.)
Assignment	=
Special assignment	+=, -=, *=, /=
Comparison	==, !=. <. <=. >, >=
Preprocessor	#

4. ***Language***: A language is considered as a finite set of strings over some finite setof alphabets.

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

6. ***Operations***: The various operations on languages are:

Union of two languages L and M is written as, L U M = {s | s is in L or s is in M} Concatenation of two languages L and M is written as, LM = {st | s is in L and t isin M}

- The Kleene Closure of a language L is written as, L* = Zero or moreoccurrence of language L.

7. ***Notations***: If r and s are regular expressions denoting the languages L(r) andL(s), then

Union : L(r) U L(s) Concatenation
: L(r) L(s)Kleene closure : (L(r))*

8. ***Representing valid tokens of a language in regular expression***: If x is aregular expression, then:

- x* means zero or more occurrence of x.
- x+ means one or more occurrence of x.

9. ***Finite automata***: Finite automata is a state machine that takes a string of symbols as input and changes its state accordingly. If the input string is successfully processed and the automata reaches its final state, it is accepted. The mathematical model of finite automata consists of:

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

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

Recognition of Tokens:

Patterns are created by using regular expression. These patterns are used to builda piece of code that examines the input strings to find a prefix that matches the required lexemes.

Consider the below grammar for branching statements and recognize the tokens from it.

```
stmt -> if expr then stmt
        | If expr then else stmt
        | ϵ
expr -> term relop term
        | term
term -> id
        |number
```

The terminals of the grammar are if, then, else, relop, id, and number, which arethe names of tokens. The patterns for these tokens are described using the following regular definitions.

```
digit       -> [0-9]
digits      -> digit+
number      -> digits (. digits)? ( E [+-]? digits )?
letter      -> [A-Za-z]
id          -> letter ( letter j digit )
if          -> if
```

then	-> then
else	-> else
relop	-> < | > | <= | >= | = | <>

with the help of above definitions, lexical analyzer will recognize, if, then, else as keywords, and relop, id, number as lexemes

Lexical analyzer also strips out white space, by recognizing the "token" with the following regular expression:

ws -> (blank/tab/newline)+

The below table shows, for each lexeme or family of lexemes, which token name is returned to the parser and what is the attribute value of token.

Lexeme	**Token Name**	**Attribute Value**
Any ws	_	_
if	if	_
then	then	_
else	else	_
Any id	id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	ET
< >	relop	NE

The Lexical Analyzer Generator Lex.

We can also produce a lexical analyzer automatically by specifying the lexeme patterns to a lexical-analyzer generator and compiling those patterns into code that functions as a lexical analyzer.

- Modification of lexical analyzer is easy, since we have only to rewrite the affected patterns, not the entire program.
- It also speeds up the process of implementing the lexical analyzer.

A lexical-analyzer generator called Lex (flex) A

Lex input file consists of three parts.

1. A collection of definitions.
2. A collection of rules.
3. A collection of auxiliary routines or user routines.

All the sections are separated by double percent signs. Default layout of a Lex fileis:

{definitions}
%%
{rules}
%%
{auxiliary routines}

Declaration Section: The declarations section includes declarations of variables, identifiers (which are declared to stand for a constant, e.g., the name of a token), and regular definitions. The following syntax is used to include declaration section in LEX specification

```
%{
        Declarations
%}
```

Rules Section: The translation rules each have the form **Rule$_i$\{ Action$_i$ \}.**

Each rule is a regular expression, which may use the regular definitions of the declaration section. The actions are fragments of code, typically written in C. The following syntax is used to include rules section in LEX specification

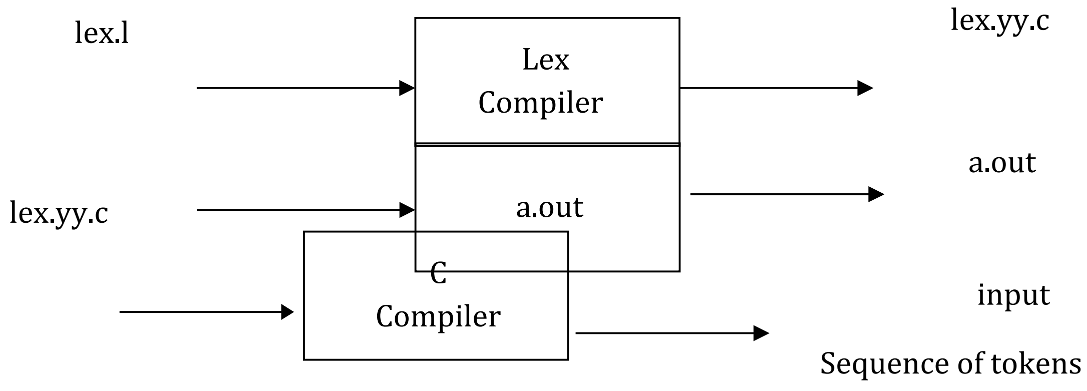
```
%%
        Rule1 { Action1 }
        Rule2 { Action2 }
%%
```

When lexical analyzer starts reading the input character by character. If a character/set of characters is matched with one of the regular expressions, then the corresponding action part will be executed.

Auxiliary Routines: The third section contains additional functions that are required. These functions may be compiled separately and loaded with the lexical analyzer.
Some procedures are required by actions in rule section.
yylex(), yywrap() are the predefined procedures of LEX. Lex

program or Lex files are saved with .l extension(dot l).

The input notation for the Lex tool is referred as the Lex language and the tool itself is the Lex compiler. The Lex compiler transforms ***lex.l*** to a C program, in afile that is always named ***lex.yy.c***. The latter file is compiled by the C compiler intoa file called ***a.out***.



Predefined functions and variables

Lex Predefined functions and Variables	
Name	**Function**
int yylex(void)	call to invoke lexer, returns token
char *yytext	pointer to matched string
yyleng	length of matched string
yylval	value associated with token
int yywrap(void)	wrapup, return 1 if done(input reaches to end), 0 if not done
FILE *yyout	output file
FILE *yyin	input file
INITIAL	initial start condition
BEGIN condition	switch start condition
ECHO	write matched string

The below program appends line number to the lines of the loaded file:

```
                /* Declaration section */
%{
   int yylineno;
%}
                /* Rules section */
%%
^(.*)\n     printf("%4d\t%s", ++yylineno, yytext);
%%
                /* Auxiliary Procedures */

int main(int argc, char **argv)
{

   yyin = fopen(argv[1], "r");
   yylex();
   fclose(yyin);
}
```

Word count without using files

```
%{
#include<stdio.h> int
wc=0,lc=0,cc=0;
%}
word [^ \n\t]+
eol \n
%%
```

```
{word} {wc++;cc+=yyleng;}
{eol} {lc++;}
%%
void main()
{
printf("Enter a line of text:\n");
scanf("%s",yytext);
yylex();
printf("\nno. of characters:%d\nno. of words:%d\nno. of lines:%d\n",cc,wc,lc);
}
int yywrap()
{
return 1;
}
```

UNIT-II

Syllabus:
Syntax Analysis (Parser): The Role of the Parser, Syntax Error Handling and Recovery, Top-Down Parsing, Bottom-Up Parsing, Simple LR Parsing, More Powerful LR Parsing, Using Ambiguous Grammars, Parser Generator YAAC.
Objective: Design top-down and bottom-up parsers
Outcome: For a given parser specification, design top-down and bottom-up parsers.

Syntax Analysis (Parser)
Second phase of compilation process is syntax analysis. Syntax Analyzer checks for the syntax of the program. It took the sequence of tokens from scanner and groups them into a structure called parse tree.

Parsing: determining the syntax or the structure of a program. Parse tree specifies the statements execution sequence.

Syntax (rules to be followed in writing programming constructs in programs) of a programming language is usually represented by CFG (Context-free Grammar) or BNF (Backus-Naur Form).

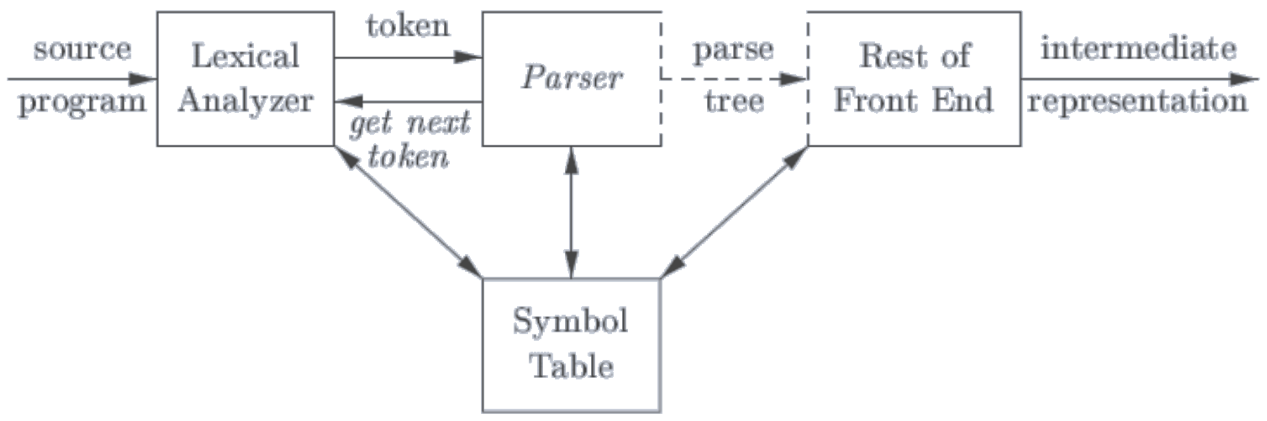
Important steps followed in parsing are:

a. Specification of syntax (CFG)
b. Representation of input after parsing (parse tree)
c. Parsing algorithms (top-down and bottom-up algorithms)

The Role of the Parser:
Parser determines the syntactic structure of a program by taking tokens as inputs and verifies that the string of token names can be generated by the grammar for the source language, then constructs a parse tree/syntax tree.

$$\begin{tikzcd}\textbf{Sequence of tokens} \arrow[r] & \boxed{\textbf{Parser}} \arrow[r] & \textbf{parse tree}\end{tikzcd}$$

Structure of the syntax tree depends on the syntactic structure of the programming language.



There are three general types of parsers for grammars:

- universal
- top-down
- bottom-up

Universal parsing methods can parse any grammar. The following two algorithms are the best examples of universal parsers.

- the Cocke-Younger-Kasami algorithm
- Earley's algorithm

These general methods are, however, too inefficient to use in producing compilers.

Commonly used methods in compilers are either top-down or bottom-up.

As implied by their names, top-down methods build parse trees from the top (root) to the bottom (leaves), while bottom-up methods start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one symbol at a time.

Syntax Error Handling and Recovery:

Most of the programming language specifications never describe about error handling, so it is the responsibility of compiler to handle errors. Compiler must track down all the errors done by the programmer while coding the program and handle them.

Common programming errors are:

Lexical Errors: Misspellings of identifiers, keywords, or operators and missing quotes around text intended as a string.

Syntactic Errors: Misplaced semicolons or extra or missing braces i.e. "{" or "}.", missing or extra else statements.

Semantic Errors: Type mismatches between operators and operands.

Logical Errors: Incorrect reasoning on the part of the programmer and use of assignment operator = instead of the comparison operator ==.

Context Free Grammar (CFG):
Syntax of every programming language construct is specified by using CFG.
Definition: CFG is a collection of four things

G =(V, T, P, S)

- V is a collection of non-terminals
- T is a collection of terminals
- P is a set of production rules of the form P → (V u T)* (LHS → RHS)
- S is a start symbol and S is a subset of V (S belongs to V)

Non-terminal: It is a symbol in a grammar which is further expanded by production rule. Usually represented with capital letters.
Terminal: It is a symbol in a grammar which are not expandable further. Usually represented with small letter (including epsilon).

These are the actual tokens.

Rules used in writing a grammar:

1. Every production rule of the grammar must be in the below form

   LHS → RHS
2. LHS must be a single non-terminal. Every non-terminal symbol must be an expandable(i.e., every non-terminal must produce atleast one terminal).
3. RHS can be a combination of both non-terminal and terminal symbols.
4. Null production rule is specified as ***non-terminal → ε***
5. One of the non-terminal symbols should be the start symbol of the grammar.

Once a grammar is constructed, we can write a language for it.
A language for the grammar is a collection of strings/words that are derived from that grammar.

Simplification of Grammar (CFG):
A grammar can be simplified/reduced by removing useless symbols, useless production rules from the grammar.

1. A symbol can be removed from the grammar, if it is not used is producing any string of the language. (Useless non-terminal and terminal symbols).

S → 0T | 1T | X | 0 | 1

T → 00

In the above grammar X is not derived. So, S → X must be removed.

2. There should not be any production rule of the form

   Non-terminal → Non-terminal (X → Y) called as unit productions.

   A → B **A → a | b**
   B → a | b

   $\}$

3. If Ɛ is not there in the language, epsilon productions can be removed

   Non-terminal → Ɛ

   S → 0S | 1S | Ɛ => S → 0S | 1S | 0 | 1

Ambiguity

A grammar is ambiguous if there is any sentence for which there exists more than one parse tree either in Left Most Derivation or Right Most Derivation.

Any parses for an ambiguous grammar have to choose somehow which tree to return. There are a number of solutions to this; the parser could pick one arbitrarily, or we can provide some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

Left Recursion:

If there is any non-terminal A, such that there is a derivation A ->Aα for some string, then the grammar is left recursive.

Algorithm for eliminating left Recursion:

1. Group all the A productions together like this:

   $A \Rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid - - - \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid - - - \mid \beta_n$

Where, A is the left recursive non-terminal, α is any string of terminals and β_i is any string of terminals and nonterminals that does not begin with A.

2. Replace the above A productions by the following:

   $A \Rightarrow \beta_1 A^I \mid \beta_2 A^I \mid - - - \mid \beta_n A^I$

   $A^I \Rightarrow \alpha_1 A^I \mid \alpha_2 A^I \mid - - - \mid \alpha_m A^I \mid \in$

   Where, A^I is a new nonterminal.

If a grammar contains left recursive, eliminate left recursion by using the below formulae

Remove the left recursion from the production:

$\mathbf{A} \rightarrow \mathbf{A}\ \alpha \mid \beta$

Applying the transformation yields:

$\mathbf{A} \rightarrow \beta\ \mathbf{A}^{\mathbf{I}}$

$\mathbf{A}^{\mathbf{I}} \rightarrow \alpha\ \mathbf{A}^{\mathbf{I}} \mid \in$

Remaining part after A.

Example 1:

Remove the left recursion from the productions:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

Applying the transformation yields:

$E \rightarrow T\ E^{I}$ $\quad$ $T \rightarrow F\ T^{I}$

$E^{I} \rightarrow +T\ E^{I} \mid \in$ $\quad$ $T^{I} \rightarrow *\ F\ T^{I} \mid \in$

Example 2:

Remove the left recursion from the productions:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T/F \mid F$

Applying the transformation yields:

$E \rightarrow T\ E^{I}$ $\quad$ $T \rightarrow F\ T^{I}$

$E \rightarrow +\ T\ E^{I} \mid -\ T\ E^{I} \mid \in$ $\quad$ $T^{I} \rightarrow *\ F\ T^{I} \mid /F\ T^{I} \mid \in$

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 may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

Algorithm:

For all A $\in$ non-terminal, find the longest prefix α that occurs in two or more right-hand sides ofA.

If $\alpha \neq \in$ then replace all of the A productions,

$A \rightarrow \alpha\ \beta_I \mid \alpha\ \beta_2 \mid - - - \mid \alpha\ \beta_n \mid r$

With

$A \rightarrow \alpha\ A^{I} \mid r$

$A^{I} \rightarrow \beta_I \mid \beta_2 \mid - - - \mid \beta_n \mid \in$

Where, A^I is a new element of non-terminal.

Repeat until no common prefixes remain.
It is easy to remove common prefixes by left factoring, creating new non-terminal.
For example, consider:

$V \to \alpha\ \beta \mid \alpha\ r$

Change to:

$V \to \alpha\ V^I$

$V^I \to \beta \mid r \mid \in$

Example:

Eliminate Left factoring in the grammar:

S → V := int

V → alpha '[' int ']' | alpha

Solution:

S → V := int

V → alpha V^I

V^I → '[' int '] | ∈

FIRST and FOLLOW:

Functions, FIRST and FOLLOW, allow us to fill in the entries of a parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during error recovery.

FIRST(α): If α is any string of nonterminal symbols of a grammar, let FIRST (α) be the set of terminals that begin the strings derived from α. If α->ε, then ε is also in FIRST(α).

(OR)

FIRST(A): If A is any nonterminal symbols of a grammar, let FIRST (A) be the set of terminals that begin the strings derived from A. If A->ε, then ε is also in FIRST(A).

FOLLOW (A), for nonterminals A: It is the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exists a derivation of the form S=> αAaβ for some α and β.

- If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

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

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

For example, everything in FIRST(Y1) is surely in FIRST(X). if Y1 does not derive ε, then we add nothing more to FIRST(X), but if Y1-> ε, then we add FIRST(Y2) and so on.

FIRST (A) = FIRST (α1) U FIRST (α2) U - - - U FIRST (αn)

Where, A -> α1 | α2 | - - - | αn, are all the productions for A.

FIRST(Aα) = FIRST (A) if A doesn't derive ε

else FIRST(A) = (FIRST (A) - { ε }) U FIRST (α)

Computation of FOLLOW():

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

- Place $ in FOLLOW(S), where S is the start symbol and $ is input right end marker .
- If there is a production A→αBβ, then everything of FIRST(β) is placed in FOLLOW(B) if β won't derive ε.
- If there is production A→αB, or A→αBβ where FIRST (β) contains ε (i.e.,β→ ε),then everything in FOLLOW(A)is in FOLLOW(B).

Example:

Construct the FIRST and FOLLOW for the grammar:

A -> BC | EFGH | H
B -> b
C -> c | ε
E -> e | ε
F -> CE
G -> g

H -> h | ε

Solution:

1. Finding first () set:

first (H) = first (h) U first (ε) = {h, ε }

first (G) = first (g) = {g}

first (C) = first (c) U first (ε) = {c, ε }

first (E) = first (e) U first (ε) = {e, ε }

first (F) = first (CE) = (first (c) - { ε }) U first (E)

= ({c, ε } { ε }) U {e, ε } = {c, e, ε }

first (B) = first (b)={b}

first (A) = first (BC) U first (EFGH) U first (H)

= first (B) U (first (E) – { ε }) U first (FGH) U {h, ε}

= { b } U { e } U (first (F) – { ε }) U first (GH) U {h, ε}

= { b } U { e } U {c, e} U first (G) U {h, ε}

= { b } U { e } U {c, e} U { g } U {h, ε}

= {b, c, e, g, h, ε }

2. Finding follow() sets:

follow(A) = {$}

follow(B) = first(C) – { ε } U follow(A) = {C, $}

follow(G) = first(H) – { ε } U follow(A)

={h, ε } – { ε } U {$} = {h, $}

follow(H) = follow(A) = {$}

follow(F) = first(GH) = {g}

follow(E) = first(FGH) U follow(F)

= ((first(F) – { ε }) U first(GH)) U follow(F)

= {c, e} U {g} U {g}

= {c, e, g}

follow(C) = follow(A) U (first (E) – { ε }) U follow (F)

= { $ } U { e } U {g}

= {e, g, $}

TOP-DOWN PARSING:
Top down parsing is the construction of a Parse tree by starting at start symbol and "guessing" each derivation until we reach a string that matches input. That is, construct tree from root to leaves.
The advantage of top down parsing in that a parser can directly be written as a program. Table- driven top-down parsers are of minor practical relevance.

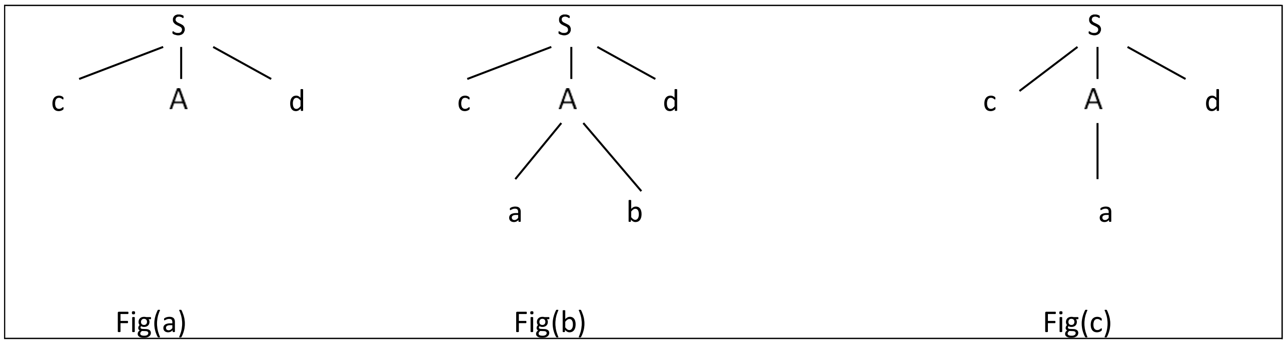
1) **Backtracking**
2) **Recursive descent parser**
3) **Predictive LL(1) parser**

Top-down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.
The special case of recursive –decent parsing, called predictive parsing, where no backtracking is required. The general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input. Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which production to use. Recursive descent parser is a top down parser involving backtracking. It makes a repeated scan of the input.

Backtracking
It will try different production rules to obtain the input string.
Backtracking is powerful, but slower.
Backtracking is not preferred for practical compilers.

Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.
Example: consider the grammar
S→cAd
A→ab|a

And the input string w=cad. To construct a parse tree for this string top-down, we initially create a tree consisting of a single node labeled scan input pointer points to c, the first symbol of w. we then use the first production for S to expand tree and obtain the tree as in the below figures.



Fig(a) Fig(b) Fig(c)

In Fig(a) the left most leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a ,the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree in Fig (b). we now have a match for the second input symbol so we advance the input pointer to d, the third, input symbol, and compare d against the next leaf, labeled b. since b does not match the d ,we report failure and go back to A to see where there is any alternative for Ac that we have not tried but that might produce a match.

In going back to A, we must reset the input pointer to position2,we now try second alternative for A to obtain the tree of Fig(c).The leaf matches second symbol of w and the leaf d matches the third symbol .

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

Predictive Parser: Predictive parser tries to predict the construction of tree using one or more lookahead symbols from the input string.

- **Recursive Descent Parser:**
- **LL(1) Parser:**

Recursive Descent Parser: (no backtracking but recursive)
It is a parser that uses collection of recursive procedures to parse an input string for a given grammar.

- CFG is used to build the procedures for each nonterminal

- RHS of each production rule is directly converted to program code of that procedure.

Rules for constructing RDP (recursive descent parser):

RHS of the production rule is directly converted into program code symbol by symbol.

- If the symbol is a nonterminal, call the corresponding procedure
- If the symbol is a terminal, then it is matched with the input symbol in the input buffer and input reader is moved to next symbol.
- If the production rule contains many alternatives, then all those alternatives must be combined in the same procedure.

Parser must be activated from start symbol.

Example: Consider the grammar.

P->num T

T-> * num T | ε

```
procedure p()
{
        if lookahead == num
        then {
                match ("num");
                T();
                }
        else
                error();
        if lookahead == $
        then {
                declare success;
                }
        else
                error();
}
Note: for start symbol we have to consider $ also.
procedure T()
{
        if lookahead == *
        then {
        match ("*");
                if lookahead == num
                then {
                        match ("num");
                        T();
                        }
                else
                        error();
```

```
                    }
          }

          procedure match(token t)
          {
                    if lookahead == t
                              lookahead = nexttoken;
                    else
                              error();
          }
          procedure error()
          {
                    print("error");
          }
```

Predictive Parser Introduction:

Predictive parsing is top-down parsing without backtracking or lookahead. For many languages, make perfect guesses (avoid backtracking) by using 1-symbol look-a-head. i.e., if:

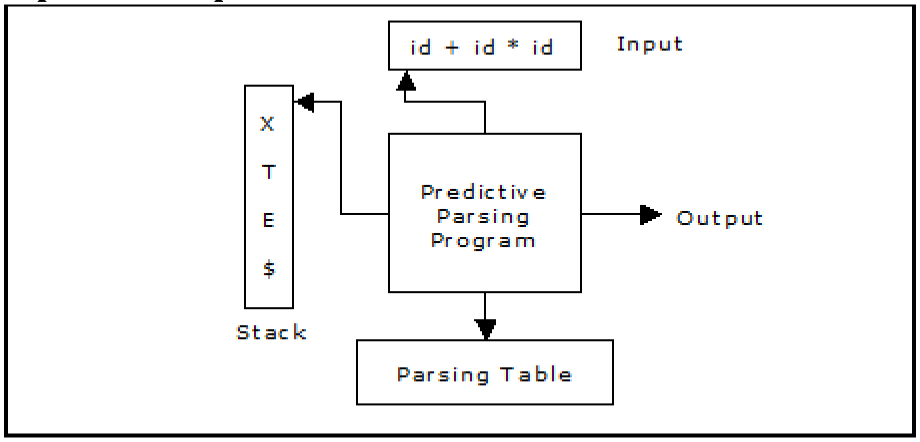
$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \text{- - -} \mid \alpha_n.$$

Choose correct α_i by looking at first symbol it derive. If $\in$ is an alternative, choose it last.

This approach is also called as predictive parsing. There must be at most one production in order to avoid backtracking. If there is no such production then no parse tree exists, and an error is returned.

The crucial property is that the grammar must not be left-recursive. Predictive parsing works well on those fragments of programming languages in which keywords occurs frequently.

The model of predictive parser is as follows:



A predictive parser has:

- Stack
- Input
- Parsing Table
- Output

The input buffer consists of the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.
The stack consists of a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially the stack consists of the start symbol of the grammar on the top of $.
Recursive descent and LL parsers are often called predictive parsers because they operate by predicting the next step in a derivation.

The algorithm for the Predictive Parser Program is as follows:

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 $S on the stack with S, the start symbol of G on top, and w$ inthe input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is:

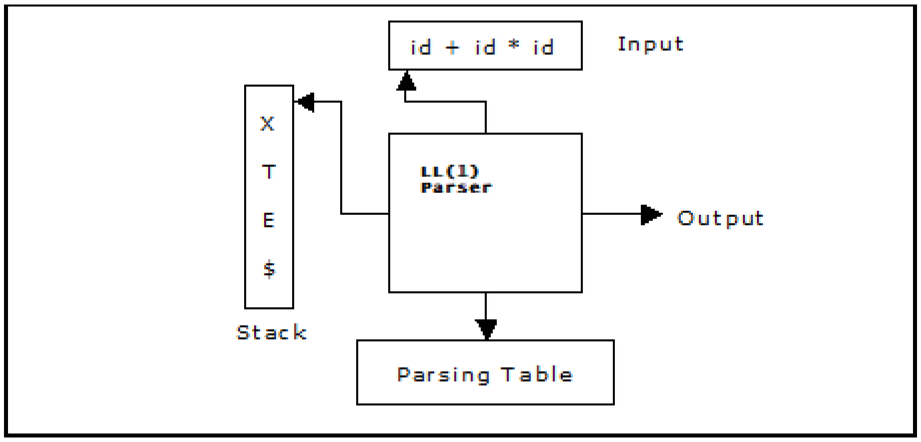
```
Set ip to point to the first symbol of w$;
repeat
let 'x' be the top stack symbol and 'a' is the symbol pointed to by ip;
if X is a terminal or $ then

        if X = a then

                pop X from the stack and advance ip
        else error()
else                              /* X is a non-terminal */
        if M[X, a] = X -> Y1 Y2..............Yk then
        begin
        pop X from the stack;
        push Yk, Yk-1,... ................................Y1 onto the stack, with Y1 on top;
        output the production X -> Y1 Y2..........................................................Yk
        end
else error()

 until X = $ /*stack is empty*/
```

LL(1) Parser:

It is a non-recursive predictive parser and a table driven parser.

The first L stands for “Left-to-right scan of input”. The second L stands for “Left-most derivation”. The ‘1’ stands for “1 token of look ahead”.

- No LL (1) grammar can be ambiguous or left recursive.



The following data structures are used by LL(1) parser:

- Input Buffer: it is an array used to place input string with $ at the end of string.
- Stack: It is used to hold the left sentential for. RHS part of the production rule is pushed into the stack in reverse order(from right to left).
    - Initially stack contains $ on the top of stack.
- Parsing Table: It is a 2D array which contains all nonterminals on rows side and terminals and $ on column side.

Because of using stack, this parser is a non-recursive.

Parser Working principle:

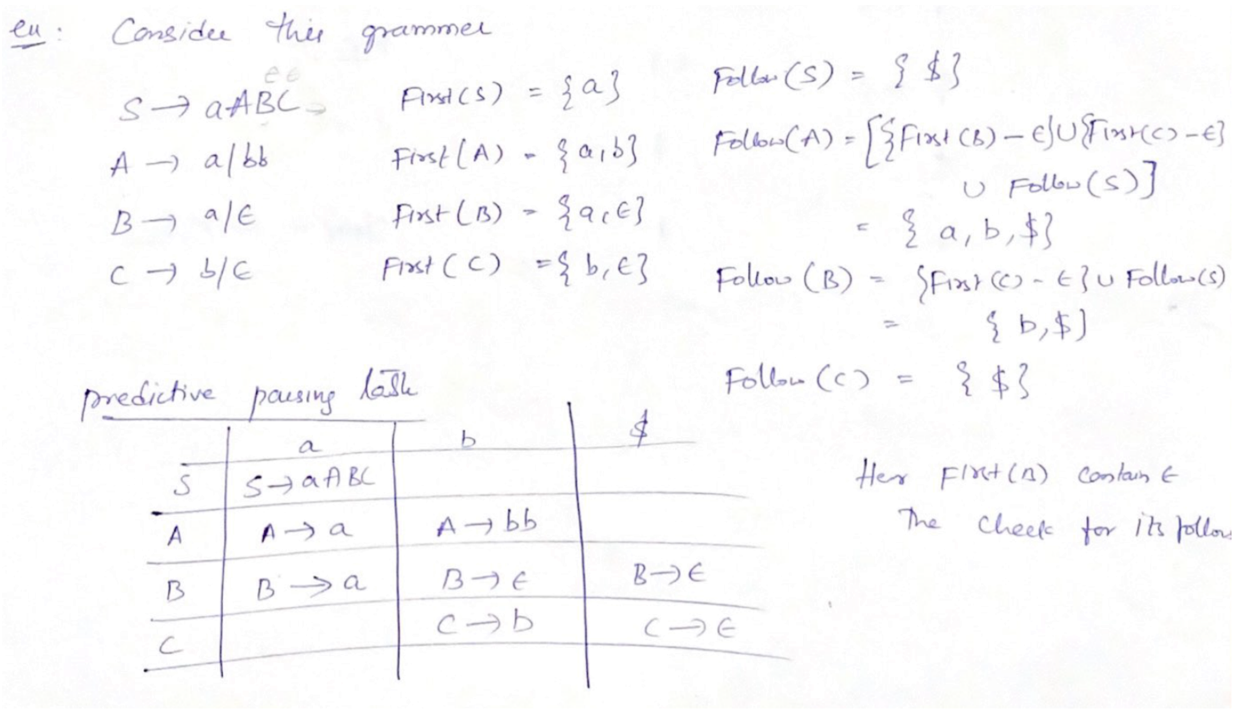
- Initially, start symbol of the grammar is pushed onto the stack
- Parser reads the top of the stack and current input symbol, and corresponding action is determined with the help of parsing table.

Construction of parsing table:

It depends on two important functions called FIRST() and FOLLOW().

1) Compute first() and follow() for all the nonterminals
2) Construct parsing table using first() and follow()
    a. For each nonterminal, fill the columns with production rules that generate its first’s set.
    b. If the first(nonterminal) contains ∈, then the columns with production rule that generates its first’s and follow’s set.

Example for constructing LL(1) parsing table:



ex: Consider this grammar

$S \to aA\overset{\in}{B}\overset{\in}{C}$ First(S) = {a} Follow(S) = {\$}

$A \to a / bb$ First(A) = {a, b} Follow(A) = [{First(B) − ∈} ∪ {First(C) − ∈} ∪ Follow(S)] = {a, b, \$}

$B \to a / \in$ First(B) = {a, ∈} Follow(B) = {First(C) − ∈} ∪ Follow(S) = {b, \$}

$C \to b / \in$ First(C) = {b, ∈} Follow(C) = {\$}

predictive parsing table

	a	b	\$
S	S → aABC		
A	A → a	A → bb	
B	B → a	B → ∈	B → ∈
C		C → b	C → ∈

Here First(B) contains ∈ The check for its follow

If the table doesn't contains multiple entries in any column, we say that the given grammar is LL(1).

Parsing an input string using LL(1) parser and table:

Consider an input string 'abba'. Parsing steps are:

- Initially stack contains \$ on the top.
- Place the input string in input buffer with \$ at the end of string.
- Parsing table is used to decide which production rule is used to parse the string.
- If the stack top is a nonterminal, replace it with its RHS part in reverse.
- If the stack is a terminal and if both top symbol and current input symbol is same, the pop that terminal.

Stack	Input	Parsing Action
\$	abba\$	push S on to the stack
\$S	abba\$	S-> aABC, so replace S with RHS, push RHS in reverse order
\$CBAa	abba\$	a is on stack top and a is current input symbol, so pop a.
\$CBA	bba\$	A->bb, so replace A with RHS in reverse order
\$CBbb	bba\$	pop b
\$CBb	ba\$	pop b
\$CB	a\$	B->a
\$Ca	a\$	pop a
\$C	\$	C->∈, pop C
\$	\$	String is accepted

BOTTOM-UP PARSING:

1) Shift Reduce Parser (SRD)

A shift-reduce parser uses a parse stack which (conceptually) contains grammar symbols.

During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack(handle) matches the RHS of a grammar rule, then the parser reduces the RHS of the rule to its LHS, replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input.

Basic Operation that are performed in SLR parsing is

Shift Operation: Shifts current input symbol on to the stack until a reduction can be applied.

Reduce Operation: If a specific substring(Handle) matching the body of a production(RHS) is appeared on the top of stack, it can be replaced by the nonterminal at the head of the production(LHS).

Accept: if the input string is parsed completely. (i.e., Start symbol is on the top of the stack and current input symbol id $)

Error: If handle is not found on top of stack or input is left in the input buffer, it is called as error. String is not parsed.

The key decisions during bottom-up parsing are about when to reduce and about what production to apply

A reduction is a reverse of a step in a derivation

The goal of a bottom-up parser is to construct a reverse of right most derivation.

Handle: It is string or a substring that matches with any of the RHS part of the production rule in the given grammar.

Handle pruning: Replacing Handle with LHS of the production rule.

A Handle is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation.

Right sentential form	**Handle**	**Reducing Production**
id*id	id	F->id
F*id	F	T->F
T*id	id	F->id
T*F	T*F	T->T*F

T E	T	E->T

Example: E=>T=>T*F=>T*id=>F*id=>id*id

Stack	Input	Action
$	id*id$	shift
$id	*id$	reduce by F->id
$F	*id$	reduce by T->F
$T	*id$	shift
$T*	id$	shift
$T*id	$	reduce by F->id
$T*F	$	reduce by T->T*F
$T	$	reduce by E->T
$E	$	accept

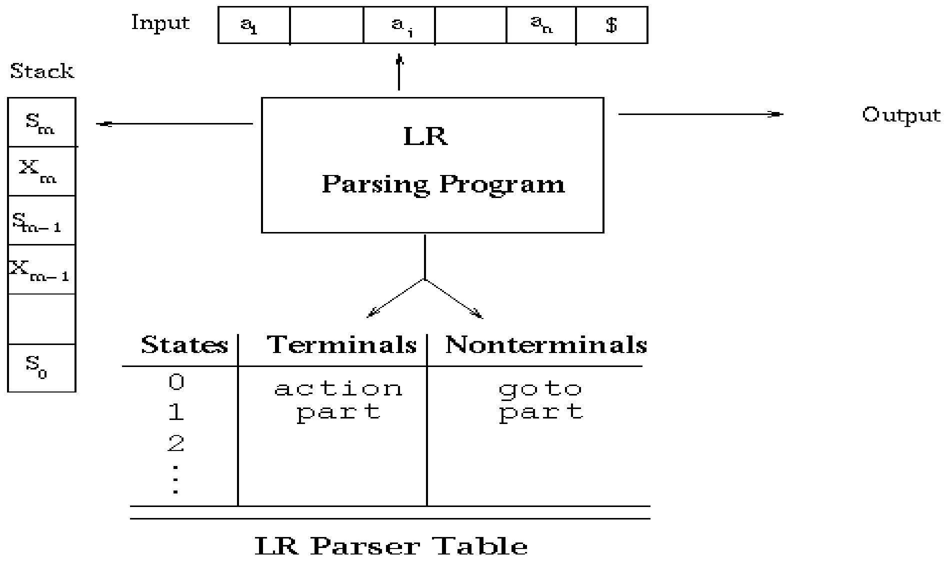
LR PARSERS

In LR parser, "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.

LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written. The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to- right scan of the input.

The disadvantage is that it takes too much work to construct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

The schematic form of an LR parser is as follows:



The parser uses a stack to store states, input symbols and grammar symbols. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift/reduce parsing decision.
The parsing table consists of two parts: action part and goto part.

The action table is a table with rows indexed by states and columns indexed by terminal symbols. When the parser is in some state s and the current lookahead terminal is t, the action taken by the parser depends on the contents of action[s][t], which can contain four different kinds of entries:

- Shift s': Shift state s' onto the parse stack.
- Reduce r: Reduce by rule r. This is explained in more detail below.
- Accept: Terminate the parse with success, accepting the input.
- Error: Signal a parse error

The goto table is a table with rows indexed by states and columns indexed by nonterminal symbols. When the parser is in state 's' immediately after reducing by rule N, then the next state to enter is given by goto[s][N].

Types of LR parsers are:

1) **SLR (Simple LR Parsing)**
2) **CLR**
3) **LALR**

Augmented Grammar:
If G is a grammar with start symbol S, then G^I, the augmented grammar for G with a new start symbol S^I and production S^I ->S.

The purpose of this new start stating production is to indicate to the parser when it should stop parsing and announce acceptance of the input i.e., acceptance occurs when and only when the parser is about to reduce by S^I->S.

LR(k) parsers are most general non-backtracking shift-reduce parsers.
Two cases of interest are k=0 and k=1. LR(1) is of practical relevance.
k' stands for number of look-a-head that are used in making parsing decisions. When (K) is omitted, 'K' is assumed to be 1.
LR(0) item: no lookahead
LR(1) item: one lookahead
LR(0) items play a key role in the SLR parser.
LR(1) items play a key role in the CLR and LALR parsers.

States of an LR parser

States represent set of items. An LR(0) item of G is a production of G with the dot at some position of the body:

For A->XYZ we have following LR(0) items

A->.XYZ

A->X.YZ

A->XY.Z

A->XYZ.

Canonical LR(0) Item sets:

Constructing canonical LR(0) item sets

Augmented grammar:

- G with addition of a production: S'->S

Closure of item sets: (only for nonterminals)

If I is a set of items, closure(I) is a set of items constructed from I by the following rules:

- Add every item in I to closure(I)
- If A->α.Bβ is in closure(I) and B->γ is a production, then add the item B->.γ to clsoure(I).

Goto:

Goto (I_i,X) where I_i is an item set and X is a grammar symbol for [A-> α.Xβ], then create a new item set I_j and add item [A-> αX. β] (here . is moved one symbol a head)

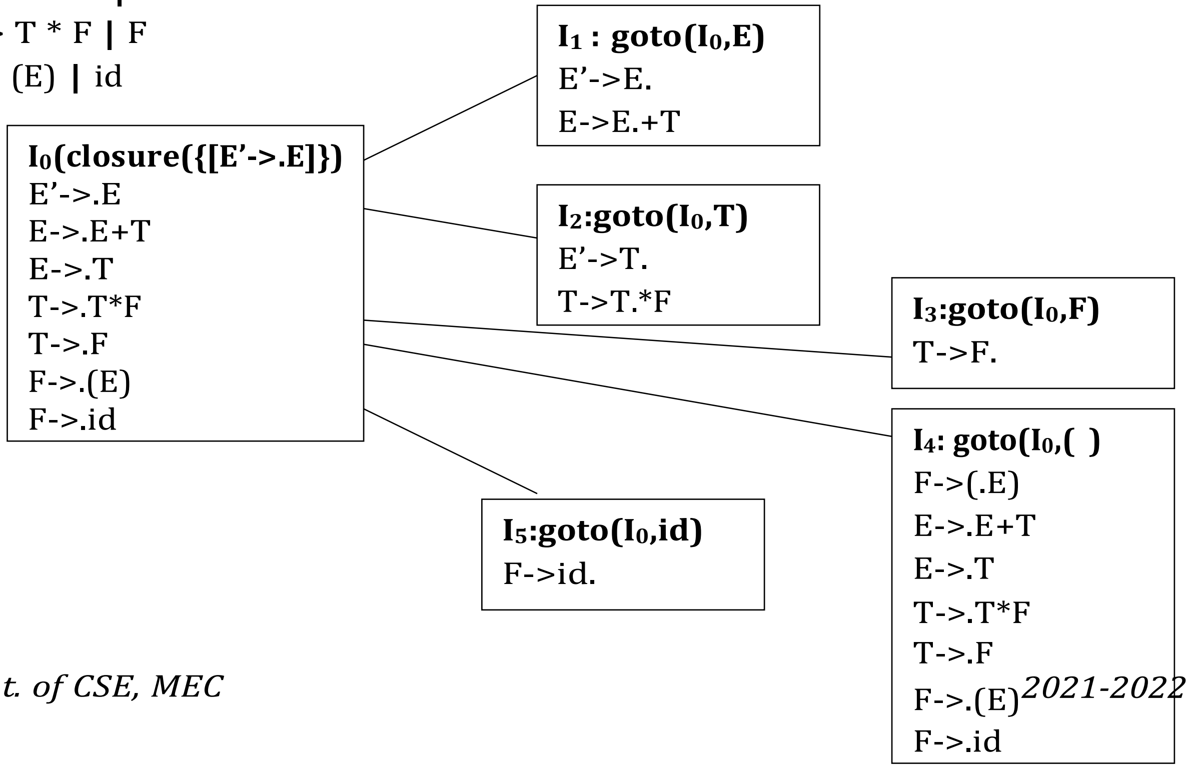
Example: Construction of LR(0) item sets

E'->E (augmented production rule)

E -> E + T | T

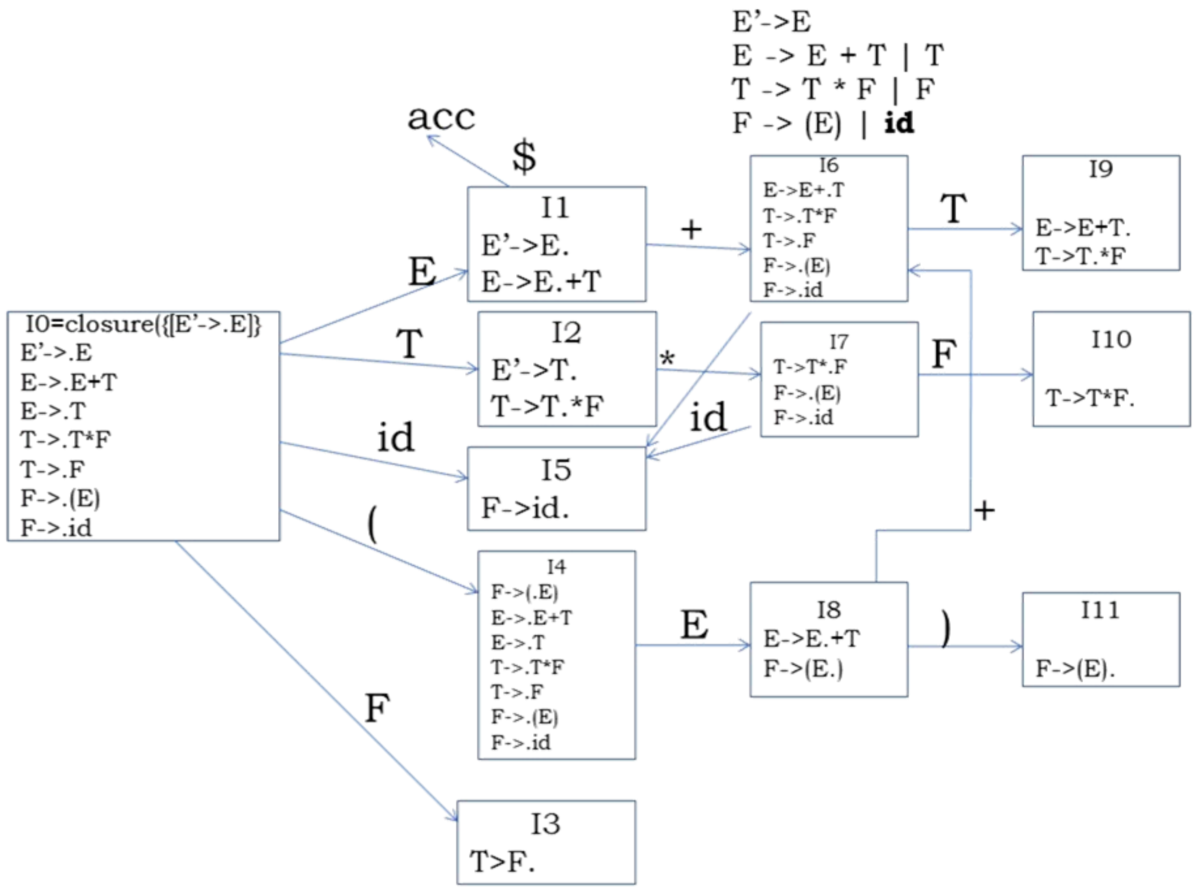
T -> T * F | F

F -> (E) | id



Note: It is just an example, complete item sets are specified in the above example.

1. Simple LR parser(SLR):
Example:



Parsing Table Construction
Steps to be followed to construct parsing table:

	Action						Goto		
State	**Id**	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_0	S_5			S_4			1	2	3
1		S_6				Accept			
2		R_2	S_7		R_2	R_2			
3		R_4	R_4		R_4	R_4			
4	S_5			S_4			8	2	3

5		R_6	R_6		R_6	R_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9		R_1	S_7		r_1	r_1			
10		R_3	R_3		R_3	R_3			
11		R_5	R_5		R_5	R_5			

Parsing an input string
Rules to be followed in parsing are:

	STACK	INPUT	ACTION
1.	0	id*id+id$	shift by S5
2.	0id5	*id+id$	see 5 on * reduce by F→id If A A→β Pop 2*| β| symbols. =2*1=2 symbols. Pop 2 symbols off the stack State 0 is then exposed on F. Since goto of state 0 on F is 3, F and 3 are pushed onto the stack
3.	0F3	*id+id$	reduce by T →F pop 2 symbols push T. Since goto of state 0 on T is 2, T and 2, T and 2 are pushed onto the stack.
4.	0T2	*id+id$	shift by S7
5.	0T2*7	id+id$	shift by S5
6.	0T2*7id5	+id$	reduce by r6 i.e. F →id Pop 2 symbols, Append F, See 7 on F, it is 10
7.	0T2*7F10	+id$	reduce by r3, i.e., T →T*F Pop 6 symbols, push T Sec 0 on T, it is 2 Push 2 on stack.

8. 0T2	+id$	reduce by r2, i.e., E □T Pop two symbols, Push E See 0 on E. It 10 1 Push 1 on stack
9. 0E1	+id$	shift by S6.
10. 0E1+6	id$	shift by S5
11. 0E1+6id5	$	reduce by r6 i.e., F □id Pop 2 symbols, push F, see 6 on F It is 3, push 3
12. 0E1+6F3	$	reduce by r4, i.e., T □F Pop 2 symbols, Push T, see 6 on T It is 9, push 9.
13. 0E1+6T9	$	reduce by r1, i.e., E □E+T Pop 6 symbols, push E See 0 on E, it is 1 Push 1.
14. 0E1	$	Accept

2. CLR Parser

Example:

S ->CC
C ->cC/d.

1.Number the grammar productions:

1. S ->CC
2. C ->cC
3. C ->d

2.The Augmented grammar is:

S^{I} ->S
S ->CC

C ->cC
C ->d.

Constructing the sets of LR(1) items:
We begin with:
SI ->.S,$ begin with look-a-head (LAH) as $.

We match the item [SI ->.S,$] with the term [A ->$\alpha$.B$\beta$, a] In the procedure closure, i.e.,

A = S^I
$\alpha = \in$
B =S
$\beta = \in$
a = $

Function closure tells us to add [B->.r,b] for each production B->r and terminal b in FIRST (βa). Now β->r must be S->CC, and since β is $\in$ and a is $, b may only be $. Thus,

S->.CC,$

We continue to compute the closure by adding all items [C->.r,b] for b in FIRST [C$] i.e., matching [S->.CC,$] against [A ->$\alpha$.B$\beta$, a] we have, A=S, $\alpha = \in$, B=C and a=$.

FIRST (C$) = FIRST (C)
FIRST(C) = {c,d}

We add items:
C->.cC, c
C->cC, d
C->.d, c
C->.d,d
None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial I_0 items are:
I_0:
S^I->.S, $
S->.CC, $
C->.cC, c/d
C->.d, c/d
Now we start computing goto (I_0,X) for various non-terminals i.e.,
Goto (I_0,S):
I_1 : S^I->S.,$ -> reduced item.

Goto (I_0,C): I_2 :

S->C.C, $
C->.cC,$
C->.d,$

Goto (I_0,c) : I_3 :
C->c.C, c/d
C->.cC, c/d
C->.d, c/d

Goto (I_0,d) : I_4 :
C->d., c/d -> reduced item.

Goto (I_2,C) : I_5 :
S->CC., $ -> reduced item.

Goto (I2,C) : I_6
C->c.C, $
C->.cC, $
C->.d, $

Goto (I2,d) : I_7
C->d., $ -> reduced item.

Goto (I3,C) : I_8
C->cC., c/d -> reduced item.

Goto (I3,C) : I_3
C->c.C, c/d
C->.cC, c/d
C->.d, c/d

Goto (I3,d) : I_4
C->d., c/d. -> reduced item.

Goto (I6,C) : I_9
C->cC., $ -> reduced item.

Goto (I6,C) : I_6
C->c.C, $
C->,cC, $

C->.d, $

Goto (I6,d) : I_7
C->d., $ -> reduced item.

All are completely reduced. So now we construct the canonical LR(1) parsing table –
Here there is no need to find FOLLOW ( ) set, as we have already taken look-a-head for each set of productions while constructing the states.

Constructing LR(1) Parsing table:

	Action			goto	
State	C	D	$	S	C
I_0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

3. LALR Parser
Example:

1.Construct C={I0,I1,… ,In} The collection of sets of LR(1) items

2.For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (plus them into a single term)

I_0 ->same as previous
I_1 -> same as previous
I_2 -> same as previous
I_{36} – Clubbing item I3 and I6 into one I36 item.

C ->cC, c/d/$
C->cC, c/d/$
C->d, c/d/$

I_5 ->some as previous

I_{47} - Clubbing item I4 and I7 into one I47 item
C->d, c/d/$

I_{89} - Clubbing item I8 and I9 into one I89 item
C->cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d	$	S	C
0	S_{36}	S_{47}		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3			
5			r1		
89	r2	r2	r2		

Using Ambiguous Grammars:

If the grammar is ambiguous, it creates conflicts, we cannot parse the input string. Ambiguous grammar generates the below to kinds of conflicts while parsing:

- Shift-Reduce Conflict
- Reduce-Reduce Conflict

But for arithmetic expressions, ambiguous grammars are more compact, provides more natural specification.

Ambiguous grammar can add any new production for special constructs. So, ambiguous grammars must be handled carefully to obtain only one parse tree for the specific input (i.e., precedence and associativity for arithmetic expressions)

Parser Generator YAAC:

To automate the process of parsing an input string by parser, certain automation tools for parser generation are available.

YACC (Yet Another Compiler Compiler) is one such automatic tool for parser generation. It is an UNIX based utility tool for LALR parser generator. LEX and YACC work together to analyze the program syntactically, can report conflicts or ambiguities in the form of error messages.

YACC Specification

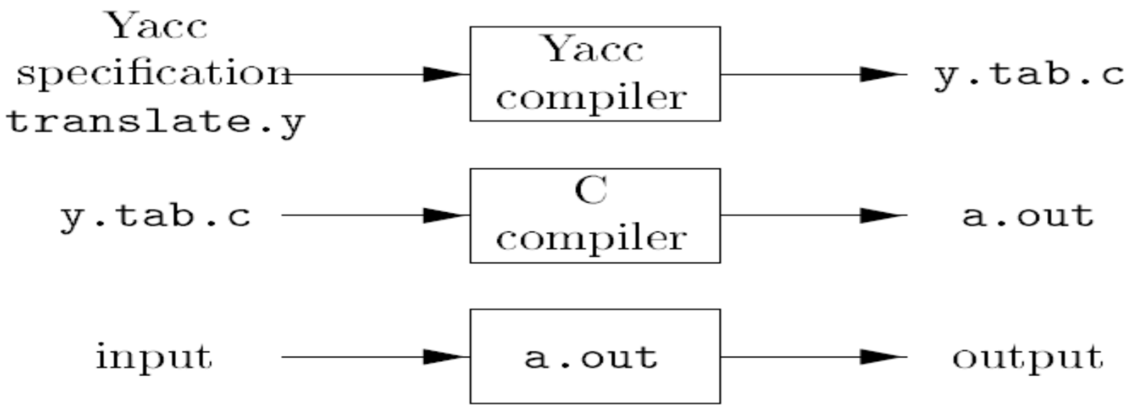
%{ declarations %}	required header files are declared
% token	required tokens are declared
%%	translation rules consist of a grammar production
translation rules	and the associated semantic action.
%%	
supporting C routines	//main() method, yylex() function is called from main.

A translator is constructed using YACC specification. After translator is prepared, it is to be saved with **.y** extension.

Compilation and execution steps:

- The UNIX system command to compile this specification is **yacc filename.y**
- it transforms the specification into **y.tab.c** and **y.tab.h** using LALR algorithm.
- Compile y.tab.c with ly library using CC compiler

  cc y.tab.c -ly
- Compilation produces **a.out** file. execute it the input program to get the desired output.
- To execute use **./a.out** command



A set of productions of the below form

<head> -> <body>1 **|** <body>2 **|** . . . **|** <body>n

would be written in YACC as

```
<head> : <body>1 { <semantic action>1 }
        | <body>2 { <semantic action>2 }
```

```
     . . .
    | <body>n { <semantic action>n }
    ;
```

In a YACC production, unquoted strings of letters and digits not declared as tokens are considered as non-terminals.
A quoted single character, e.g., 'c', is considered as terminal symbol c, as well as the integer code for the token represented by that character (i.e., Lex would return the character code for 'c' to the parser, as an integer).

Second part of YACC specification contains semantic action which are a sequence of C statements.
In this, the symbol $$ refers to the attribute value associated with the nonterminal of the head, while $i refers to the value associated with the ith grammar symbol (terminal or nonterminal) of the body.

In the YACC specification, for the written E -productions
E -> E + T | T
and their associated semantic actions as:

```
expr : expr '+' term { $$ = $1 + $3; } // Note:  + is $2
      | term          { $$ = $1; }     // Note: we can omit this
      ;
```

The third part of a YACC specification consists of supporting C-routines. A lexical analyzer by the name yylex() must be provided.

Example:
YACC source program for a simple desk calculator that reads an arithmetic expression, evaluates it, and then prints its numeric value.
Grammar for arithmetic expressions

E -> E + T | T
T -> T * F | F
F -> (E) | digit

The token digit is a single digit between 0 and 9.

```
%{
#include<stdio.h>
#include<ctype.h>
%}
%token DIGIT
%%
line : expr '\n' { printf("%d\n", $1); }
     ;
expr : expr '+' term { $$ = $1 + $3; }
      | term
```

```
        ;
term : term '*' factor { $$ = $1 * $3; }
        | factor
        ;
factor : '(' expr ')' { $$ = $2; }
          | DIGIT
          ;
%%
void main()
{
yyparse();
}
void yyerror(char *s)
{
fprintf(stderr, "\nerror\n");
}
int yylex() {
int c;
c = getchar();
if (isdigit(c)) {
yylval = c-'0';
return  DIGIT;
}
return c;
}
```

UNIT-III

Syllabus: Syntax-Directed Translation: Syntax-Directed Definitions, Evaluation Orders for SDD's Applications of Syntax-Directed Translation.
Symbol Table: Structure, Operations, Implementation and Management.
Objective:

- Describe semantic analyses using an attribute grammar
- To learn how to build symbol tables

Outcome: Construct an abstract syntax tree for language constructs.

SYNTAX-DIRECTED TRANSLATION:
Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar.

Concepts related to syntax-directed translation are:

1. **Attributes**	**(Attributes are used in SDD)**
2. **Translation Schemes**	**(Written in SDT)**

Attributes: An attribute is any quantity associated with a programming construct.

- Some of the attributes: Data types of expression, number of instructions, location of the instruction, nonterminals, and terminal symbols.

Translation schemes: A translation scheme is a notation for attaching program fragments to the productions of a grammar

If **X** is a grammar 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 labelled as **X**.

Example:

Production	**Semantic Rule**
$E \rightarrow E_1 + T$	$E.code = E_1.code + T.code$ (or) $E.code = E_1.code \| \| T.code \| \| '+'$

- The above production has two non-terminals, E_1 and T. E_1 and T have a string-valued attribute **code**. The semantic rule species that the string E:code is formed by concatenating E1:code, T:code, and the character '+'.

Alternatively, we can also insert the **semantic actions/translation scheme** inside the grammar.

E -> E1 + T { print '+' }

- By convention, semantic actions are enclosed within curly braces.

The position of a semantic action in a production body determines the order inwhich the action is executed. (i.e., semantic actions can be place anywhere in theRHS part of the production rule)

Out of these two notations, syntax-directed definitions (SDD) are more readable, and hence more useful for specifications. However, translation schemes (SDT) can be more efficient, and hence more useful for implementations.

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. If X is a symbol and a is one of its attributes, thenwe 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 attributesof X can be implemented by data fields in the records that represent the nodes forX.

- Attributes may be of any kind: numbers, types, table references, or strings, for instance.
- Rules describe how the attributes are computed at those nodes of the parse tree.

For each **nonterminal** of a production rule, there are two types of attributes.

1. Synthesized Attributes
2. Inherited Attributes

Synthesized Attribute:

- 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 of children of **N** and at **N** itself.

Inherited Attribute:

- 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.

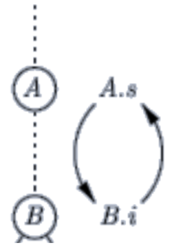
Terminals can have synthesized attributes, but not inherited attributes.

- Attributes for terminals have lexical values that are supplied by the lexical analyzer.
- There are no semantic rules in the SDD itself for computing the value of an attribute for a terminal.

Note:
An inherited attribute at node N cannot be defined in terms of attribute values at the children of node N.
But a synthesized attribute at node N can be defined in terms of inherited attribute values at node N itself.

Example:

PRODUCTION	SEMANTIC RULES
A -> B	**A.s := B.i;**
	B.i := A.s + 1



A and B, have ‘s’ and ‘i’ as attributes.
Here ‘s’ is synthesized and ‘i’ is inherited, because ‘s’ is evaluated from its children and ‘i’ is evaluated from its parent.

EVALUATION ORDERS FOR SDD’S

S-attributed SDD
An SDD that involves only synthesized attributes is called S-attributed SDD.

In an S-attributed SDD, each rule computes an attribute for the nonterminal at the head of a production from attributes taken from the body of the production.

An S-attributed SDD can be implemented naturally in conjunction with an LR parser. An S-attributed definition is suitable for use during bottom-up parsing.

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

To visualize the translation specified by an SDD, annotated parse trees are used.
A parse tree, showing the value(s) of its attribute(s) is called an **annotated parse tree.**

Example of S-Attributed SDD

Production	**Semantic Rules**
L -> E n	L.val = E.val
E -> E_1 + T	E.val = E_1.val + T.val
E -> T	E.val = T.val
T -> T_1 * F	T.val = T_1.val * F.val
T -> F	T.val = F.val
F -> (E)	F.val = E.val
F -> digit	F.val = digit.lexval

In the above SDD, each of the non-terminals has a single synthesized attribute, called **val**.

- Terminal digit has a synthesized attribute **lexval**, which is an integer value returned by the lexical analyzer.
- For production 1: L -> E n, sets L.val to E.val, which is a numerical value of the entire expression (n indicates that).
- For production 2: E -> E_1 + T, also has one rule, which computes the **val** attribute for the head E as the sum of the values at E_1 and T. At any parse tree node N labeled E, the value of **val** for E is the sum of the values of **val**at the children of node N labeled E_1 and T.
- For production 3: E -> T, has a single rule that defines the value of **val** for E
- to be the same as the value of **val** at the child for T.
- For production 4: T -> T_1 * F, also has one rule, which computes the **val** attribute for the head T as the product of the values at T_1 and F. At any parse tree node N labeled T, the value of **val** for T is the product of the values of**val** at the children of node N labeled T_1 and F.
- For productions 5 and 6 copy values at a child, like that for the third production.
- For production 7: It gives F.val the value of a digit, that is, the numerical value of the token digit that the lexical analyzer returned.

Evaluating an SDD at the Nodes of a Parse Tree

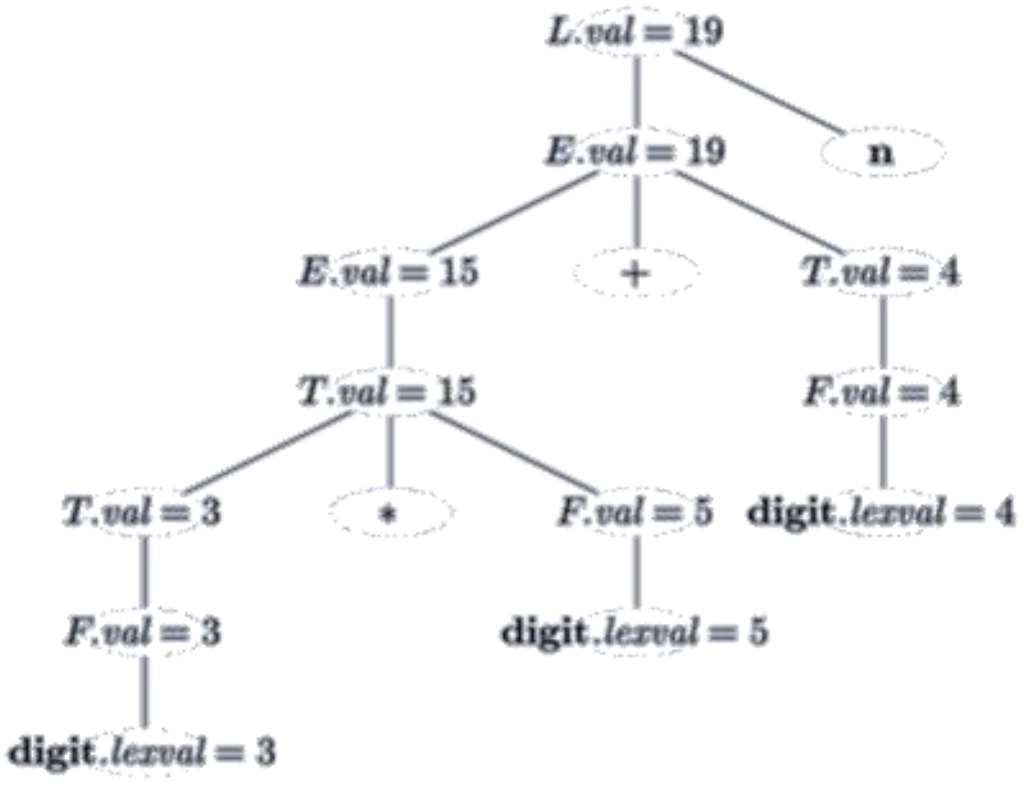
First constructing a parse tree for the grammar and then by using the rules evaluate all of the attributes at each of the nodes of the parse tree. A parse tree, showing the value(s) of its attribute(s) is called an annotated parse tree.

Before evaluating an attribute at a node of a parse tree, we must evaluate all the attributes upon which its value depends.

For example,
If all attributes are synthesized, then we must evaluate the **val** attributes at all of the children of a node before we can evaluate the **val** attribute at the node itself.

With synthesized attributes, we can evaluate attributes in any bottom-up order, such as that of a postorder traversal of the parse tree.

Annotated Parse tree for the above SDD for the expression **3 * 5 + 4 n** is



L-attributed SDD
If an SDD uses both synthesized attributes and inherited attributes with a restriction that inherited attribute can inherit values from left siblings only, it is called as L-attributed SDT.
Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Semantic actions are placed anywhere in RHS.

L-attributed, is suitable for use during top-down parsing.
Example:
A -> XYZ {Y.S = A.S, Y.S = X.S}

is an L-attributed grammar since Y.S = A.S and Y.S = X.S

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

Example:

L-Attributed definitions

Production	**Semantic Rules**
T —> FT'	T'.inh = F.val
T' —> *FT_1'	T'$_1$.inh =T'.inh * F.val

In production 1, the inherited attribute T' is computed from the value of F which is to its left.

In production 2, the inherited attributed T_l' is computed from T'. inh associated with its head and the value of F which appears to its left in the production. i.e., for computing inherited attribute, it must either use from the above or from theleft information of SDD.

Syntax-directed definition with inherited attributes

Production	**Semantic Rules**
D —>TL	L.inh = T.type
T —> int	T.type =integer
T —> float	T.type = float
L —> L_1, id	L_1.inh = L.inh
	addType (id.entry, Linh)
L —> id	addType (id.entry, L.inh)

Evaluation Orders for L-Attributed SDD's

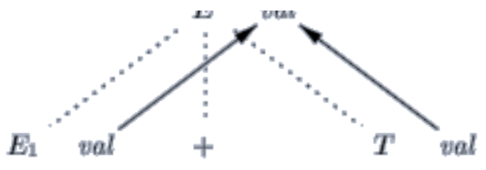
"Dependency graphs" are a useful tool for determining an evaluation order for the attribute instances in a given parse tree. While an annotated parse tree shows the values of attributes, a dependency graph helps us determine how those values can be computed.

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

- For each parse tree node X, the dependency graph has a node for each attribute associated with the node X.

- For a production p, if a semantic rule defines the value of synthesized attribute A.b in terms of the value of X.c then the dependency graph has an edge from X.c to A.b
- For a production p, if a semantic rule defines the value of inherited attribute B.c in terms of the value of X.a then the dependency graph has an edge from X.a to B.c

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



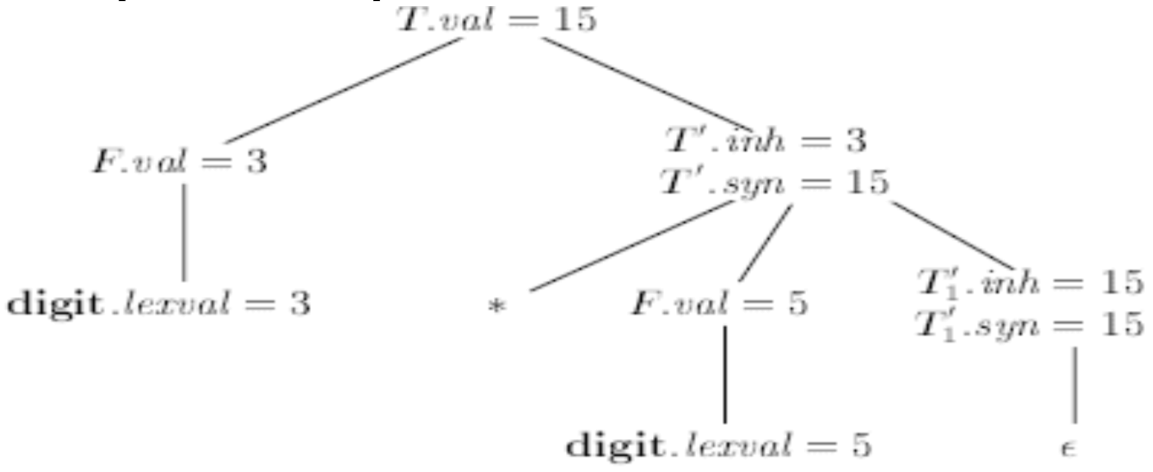
Ordering the evaluation of attributes

- If dependency graph has an edge from M to N then M must be evaluated before the attribute of N
- Thus, the only allowable orders of evaluation are those sequence of nodes $N_1, N_2, \ldots, N_k$ such that if there is an edge from N_i to N_j then $i<j$
- Such an ordering is called a topological sort of a graph.
- If there is any cycle in the graph, then there are no topological sorts; that is, there is no way to evaluate the SDD on this parse tree.

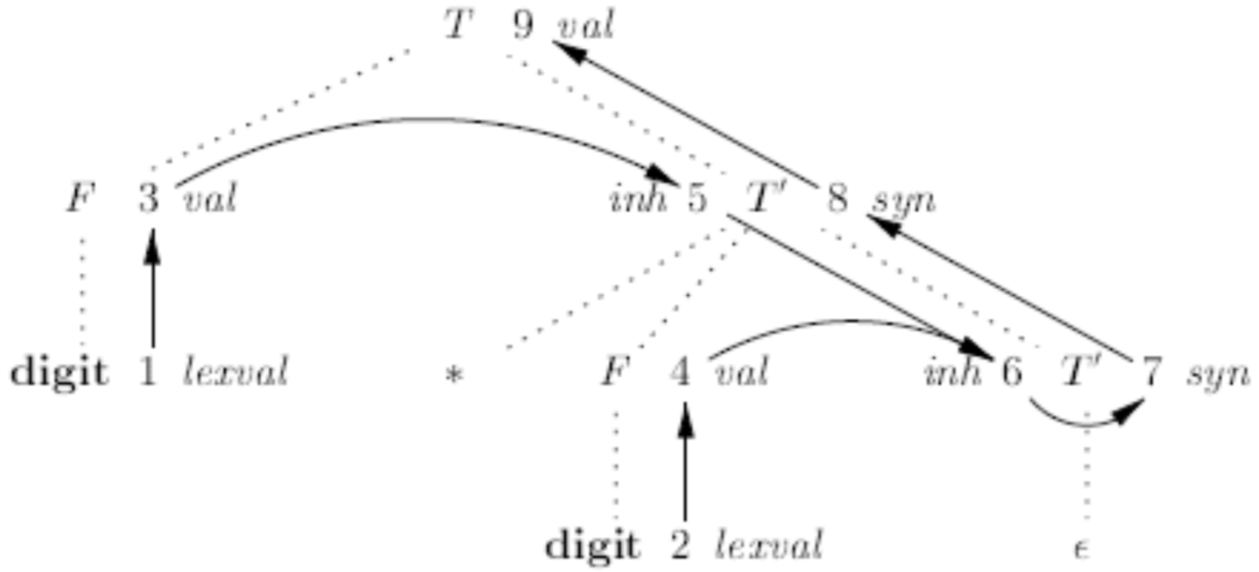
Example: L-Attributed SDD Example

Production	Semantic Rules
T -> FT’	T’.inh = F.val T.val = T’.syn
T’ -> *FT’$_1$	T’1.inh = T’.inh*F.val T’.syn = T’$_1$.syn
T’ -> ε	T’.syn = T’.inh
F -> digit	F.val = F.val = digit.lexval

Annotated parse tree for expression: 3 * 5



Dependency Graph for the expression: 3 * 5



- Nodes 1 and 2 represent the attribute lexval associated with the two leaves labeled digit.
- Nodes 3 and 4 represent the attribute val associated with the two nodes labeled F. The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines F.val in terms of digit.lexval. In fact, F.val equals digit.lexval, but the edge represents dependence, not equality.
- Nodes 5 and 6 represent the inherited attribute T'.inh associated with each of the occurrences of nonterminal T'.
- Nodes 7 and 8 represent the synthesized attribute syn associated with the occurrences of T'.

APPLICATIONS OF SYNTAX-DIRECTED TRANSLATION.

Syntax Directed Translation (SDT)

An SDT is a Context Free grammar with program fragments embedded within production bodies. These program fragments are called semantic actions.

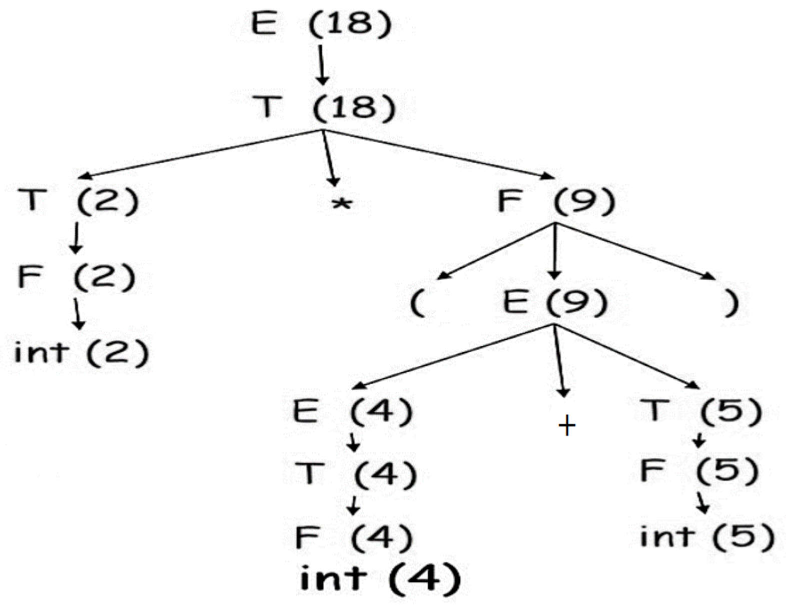
- They can appear at any position within production body, must be placed in **{ }**
- Any SDT can be implemented by first building a parse tree and then performing the actions in a left-to-right depth first order
- Typically, SDT's are implemented during parsing without building a parse tree

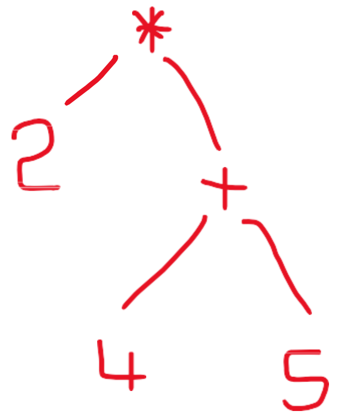
Example of SDT

A common way of syntax-directed translation is translating a string into a sequence of actions by attaching one such action to each grammar rule.

Production	**Semantic Rules**
S → E $	{ print E.VAL }
E → E + E	{ E.VAL := E.VAL + E.VAL }/{ print('+'); }
E → E * E	{ E.VAL := E.VAL * E.VAL }/{ print('*'); }
E → (E)	{ E.VAL := E.VAL }
E → I	{ E.VAL := I.VAL }
I → I digit	{ I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL }

For arithmetic expression 2*(4+5).





Applications of SDT

1. Primary application of SDT is construction of Syntax Trees
   - Since some compilers use the syntax trees as an intermediate representation, a common form of SDD(Syntax Directed Definition) turns its input string into a tree.
2. SDT is used for Executing Arithmetic Expression.
3. In the conversion from infix to postfix expression.
4. In the conversion from infix to prefix expression.
5. It is used for Binary to decimal conversion.
6. In counting number of Reduction.
7. SDT is used to generate intermediate code.
8. In storing information into symbol table.
9. SDT is used for type checking also.

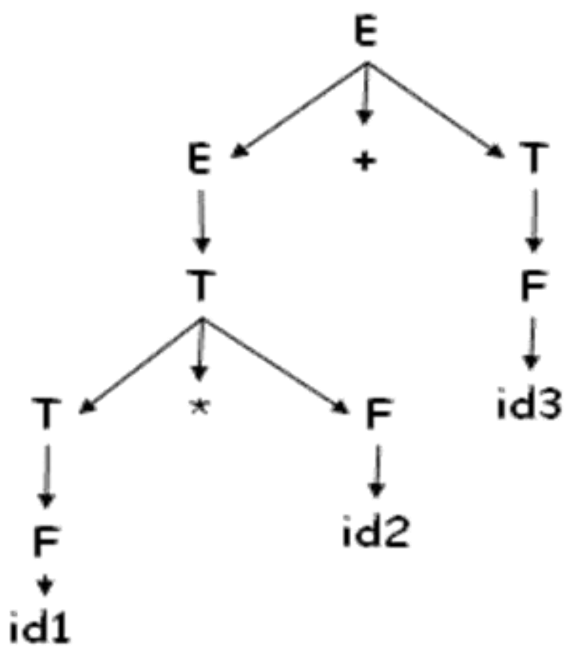
Differences between SDD and SDT

	SDD (Syntax Directed Definition)	**SDT (Syntax Directed Translation)**
1	It is a context-free grammar where attributes and rules are combined and associated with grammar symbols and productions, respectively.	It refers to the translation of a string into an array of actions. This is done by adding an action to a rule of context-free grammar. It is a type of compiler interpretation.
2	Attribute Grammar	Translation Schemes
3	SDD: Specifies the values of attributes by associating semantic rules with the productions	SDT: embeds program fragments (also called semantic actions) within production bodies
4	E -> E + T { E.val := E1.val + T.val }	E -> E + T { print('+'); }
5	Always written at the end of body of production	The position of the action defines the order in which the action is executed (in the middle of production or end)
6	More Readable	More Efficient
7	Used to specify the values of nonterminals	Used to implement S-Attributed SDD and L-Attributed SDD
8	Specifies what calculation is to be done at each production	Specifies what calculation is to be done at each production and at what time they must be done
9	Left to right evaluation	Left to right evaluation
10	Used to know the value of nonterminals	Used to generate Intermediate Code

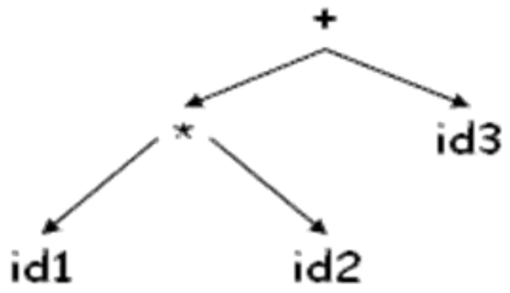
Abstract Syntax Tree:

Abstract Syntax Tree: It is a condensed form of parse trees. Normally operators and keywords appear as leaves in parse tree, but in an abstract syntax tree they are associated with the interior nodes that would be the parent of those leaves in the parse tree.

- Useful for representing language constructs



Parse Tree/Syntax Tree



Abstract Syntax Tree

Chain of single production of parse tree are collapsed into one node with the operators moving up to become the node in abstract syntax tree.

Constructing Abstract Syntax tree for expression

Each node in an abstract syntax tree can be implemented as a record with several fields.

operators: one field for operator, remaining fields ptrs to operands **mknode(op, left, right)**

identifier: one field with label id and another ptr to symbol table **mkleaf(id, entry)**

number: one field with label num and another to keep the value of the number. **mkleaf(num, val)**

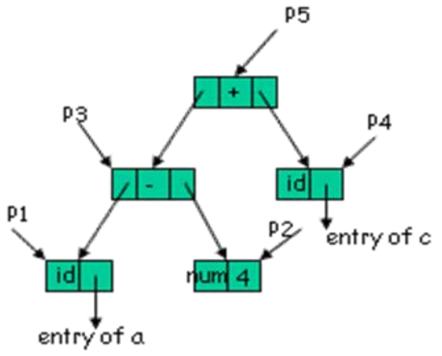
Example of Abstract Syntax Tree

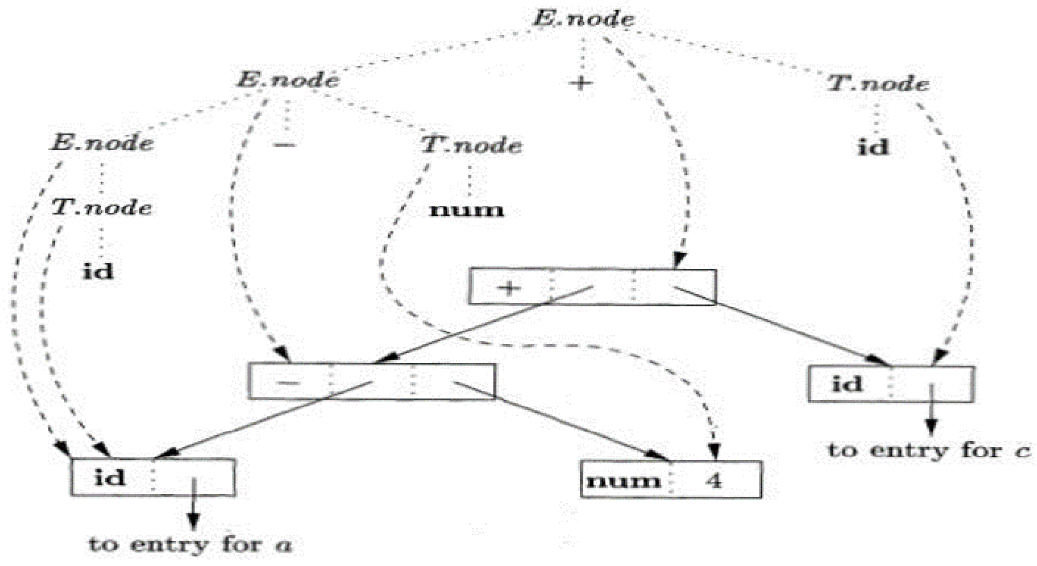
A syntax directed definition for constructing syntax tree

Production	Semantic Rules
E -> E1 + T	E.node=new mknode('+', E1.node,T.node)
E -> E1 - T	E.node=new mknode('-', E1.node,T.node)
E -> T	E.node = T.node
T -> (E)	T.node = E.node
T -> id	T.node = new mkleaf(id,id.entry)
T -> num	T.node = new mkleaf(num,num.val)

the following Sequence of function calls create a syntax tree for **a – 4 + c**

P 1 = mkleaf(id, entry.a)
P 2 = mkleaf(num, 4)
P 3 = mknode(-, P 1 , P 2)
P 4 = mkleaf(id, entry.c)
P 5 = mknode(+, P 3 , P 4)





Syntax tree for $a - 4 + c$

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

SYMBOL TABLE

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

A symbol table is a major data structure used in a compiler:

- It associates attributes with identifiers used in a program. For instance, a type attribute is usually associated with each identifier.

A symbol table is a necessary component. Definition (declaration) of identifiers appears once in a program. Use of identifiers may appear in many places of the program text. Identifiers and attributes are entered by the analysis phases when processing a definition (declaration) of an identifier.

In simple languages with only global variables and implicit declarations:

- The scanner can enter an identifier into a symbol table if it is not already there.

In block-structured languages with scopes and explicit declarations:

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

Symbol table information is used by the analysis and synthesis phases:

- To verify that used identifiers have been defined (declared)
- To verify that expressions and assignments are semantically correct – type checking
- To generate intermediate or target code

Symbol Table Structure:
Assign variables to storage classes that prescribe scope, visibility, and lifetime

  - scope rules prescribe the symbol table structure

- scope: unit of static program structure with one or more variable declarations
  - scope may be nested
- Pascal: procedures are scoping units
- C: blocks, functions, files are scoping units

- Visibility, lifetimes, global variables
    - Common (in Fortran)
- Automatic or stack storage
- Static variables

storage class : A storage class is an extra keyword at the beginning of a declaration which modifies the declaration in some way. Generally, the storage class (if any) is the first word in the declaration, preceding the type name. Ex. static, extern etc.

Scope: The scope of a variable is simply the part of the program where it may be accessed or written. It is the part of the program where the variable's name may be used. If a variable is declared within a function, it is local to that function. Variables of the same name may be declared and used within other functions without any conflicts. For instance,

```
int fun1()
{
   int a;
   int b;
   ....
}

int fun2()
{
   int a;
   int c;
   ....
}
```

Visibility: The visibility of a variable determines how much of the rest of the program can access that variable. You can arrange that a variable is visible only within one part of one function, or in one function, or in one source file, or anywhere in the program.

Local and Global variables: A variable declared within the braces {} of a function is visible only within that function; variables declared within functions are called local variables. On the other hand, a variable declared outside of any function is a global variable , and it is potentially visible anywhere within the program.

Automatic Vs Static duration: How long do variables last? By default, local variables (those declared within a function) have automatic duration : they spring into existence when the function is called, and they (and their values) disappear when the function returns. Global variables, on the other hand, have static duration : they last, and the values stored in them persist, for as long as the program does. (Of course, the values can in general still be overwritten, so they don't necessarily persist forever.) By default, local variables have automatic duration. To give them static duration (so that, instead of coming and going as the function is called, they persist for as long as the function does), you precede their declaration with the static keyword: static int i; By default, a declaration of a global variable (especially if it specifies an initial value) is the defining instance. To make it an external declaration, of a variable which is defined somewhere else, you precede it with the keyword extern: extern int j; Finally, to arrange that a global variable is visible only within its containing source file, you precede it with the static keyword: static int k; Notice that the static keyword can do two different things: it adjusts the duration of a local variable from automatic to static, or it adjusts the visibility of a global variable from truly global to private-to-the-file.

Contents in a symbol table
Possible entries in a symbol table for each identifier/variable:
- Name: a string.
- Attribute:
  - Reserved word
  - Variable name
  - Type name
  - Procedure name
  - Constant name
- Data type.
- Storage allocation, size, . . .
- Scope information: where and when it can be used.

How names are stored
Fixed-length name: allocate a fixed space for each name
If the size allocated is too little: names must be short.
If the size is too much: waste a lot of spaces.

Variable-length name:
- A string of space is used to store all names.
- For each name, store the length and starting index of each name.

Symbol Table Entries
- each entry for a declaration of a name
- format need not be uniform because information depends upon the usage of the name

- each entry is a record consisting of consecutive words
- to keep records uniform some entries may be outside the symbol table
- information is entered into symbol table at various times
    - keywords are entered initially
    - identifier lexemes are entered by lexical analyzer
- symbol table entry may be set up when role of name becomes clear
- attribute values are filled in as information is available

For each declaration of a name, there is an entry in the symbol table. Different entries need to store different information because of the different contexts in which a name can occur. An entry corresponding to a particular name can be inserted into the symbol table at different stages depending on when the role of the name becomes clear. The various attributes that an entry in the symbol table canhave are lexeme, type of name, size of storage and in case of functions - the parameter list etc.

Operations:
The basic operations defined on a symbol table include:

- **allocate** – to allocate a new empty symbol table
- **free** – to remove all entries and free the storage of a symbol table
- **insert** – to insert a name in a symbol table and return a pointer to its entry
- **lookup** – to search for a name and return a pointer to its entry
- **set_attribute** – to associate an attribute with a given entry
- **get_attribute** – to get an attribute associated with a given entry

Other operations can be added depending on requirement

For example, a delete operation removes a name previously inserted

Some identifiers become invisible (out of scope) after exiting a block

Implementation and Management:

Basic Implementation Techniques:

- First consideration is how to insert and lookup names

Variety of implementation techniques

1. Unordered List
2. Ordered List (Array)
3. Binary Search Tree
4. Hash tables

1. **Unordered List**
    - Simplest to implement
    - Implemented as an array or a linked list
    - Linked list can grow dynamically – alleviates problem of a fixed size array Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

2. **Ordered List**
    - If an array is sorted, it can be searched using binary search – O(log2 n)
    - Insertion into a sorted array is expensive – O(n) on average
    - Useful when set of names is known in advance – table of reserved words

3. **Binary Search Tree**
    - Can grow dynamically
    - Insertion and lookup are O(log2 n) on average

4. **Hash Tables and Hash Functions:**
    - A hash table is an array with index range: 0 to TableSize – 1
    - Most commonly used data structure to implement symbol tables
    - Insertion and lookup can be made very fast – O(1)
    - A hash function maps an identifier name into a table index
    - A hash function, h(name), should depend solely on name
    - h(name) should be computed quickly
    - h should be uniform and randomizing in distributing names
    - All table indices should be mapped with equal probability
    - Similar names should not cluster to the same table index.

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

The global symbol table contains names for global variable and procedure names, which should be available to all the child nodes.

To determine the scope of a name, symbol tables are arranged in hierarchical structure. Symbol table data structure hierarchy is stored in the semantic analyzer and whenever a name needs to be searched in a symbol table, it is searched using the following algorithm:

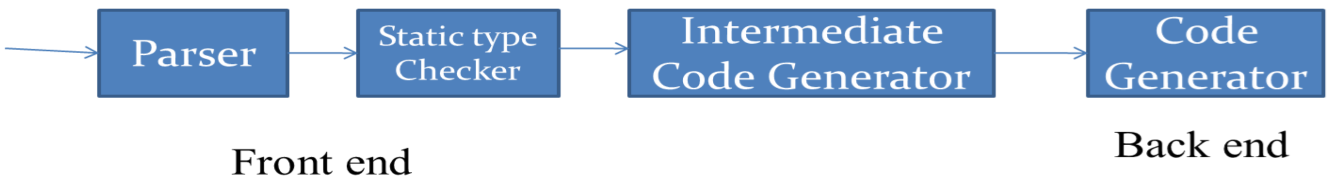
- first a symbol will be searched in the current scope, i.e., current symbol table.
- if a name is found, then search is completed, else it will be searched in the parent symbol table until,
- either the name is found, or global symbol table has been searched for the name.

UNIT-IV

Intermediate Code Generation: Variants of Syntax Trees, Three-Address Code, Types and Declarations, Translation of Expressions, Type Checking, Control Flow, Backpatching, Switch-statements, Intermediate Code for Procedures.
Run-time environment: Storage Organization, Stack Allocation of Space, Access to Nonlocal Data on the Stack, Parameter passing, Heap Management andGarbage Collection.

INTERMEDIATE CODE GENERATION: Task of compiler is to convert source program into machine program.

Converting source program into machine code in one pass is not possible always. So, compiler generates an easy representation of source language (program) called as Intermediate language which leads to an efficient code generation.



Intermediate code is the interface between front end and back end in a compiler

Benefits of Intermediate Code:
There are certain benefits of generating Intermediate Code

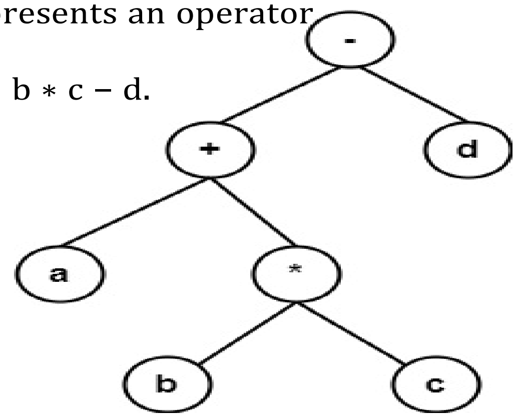
- For different machines, developing a compiler is easy (only back end is to be modified)
- Compiler for different source languages (on same machine) can be created by different front ends.
- A machine independent code optimizer can be applied to intermediate code to optimize the code generation.

VARIANTS OF SYNTAX TREES:
Forms of Intermediate Code
Below three representations are used to represent the intermediate language.

- Abstract syntax tree
- Polish notation
- Three Address code

Abstract Syntax Tree:
It is a condensed form of parse trees. In an abstract syntax tree, each leaf node describes an operand, and each interior node represents an operator.

Example: Abstract Syntax tree for the string : a + b ∗ c − d.



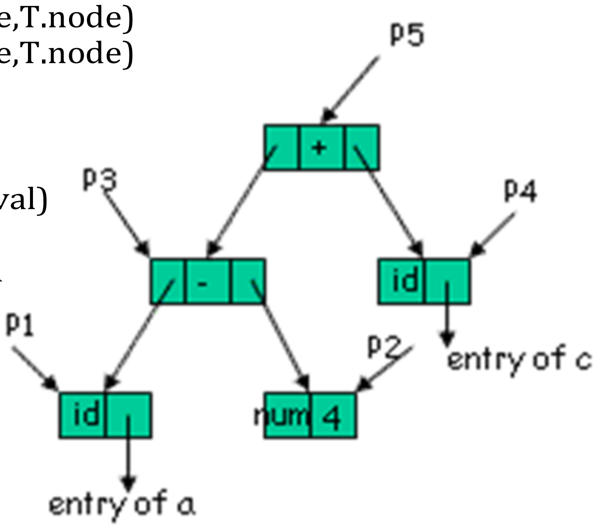
Constructing Abstract Syntax tree for expression

Each node in an abstract syntax tree can be implemented as a record with several fields.

operators : one field for operator, remaining fields points to operands
mknode(op, left, right)
identifier : one field with label id and another ptr to symbol table
mkleaf(id, entry)
number : one field with label num and another to keep the value of the number.
mkleaf(num, val)

Example of Abstract Syntax Tree:

Production	**Semantic Rules**
E -> E1 + T	E.node=new node('+', E1.node,T.node)
E -> E1 - T	E.node=new node('-', E1.node,T.node)
E -> T	E.node = T.node
T -> (E)	T.node = E.node
T -> id	T.node = new Leaf(id,id.entry)
T -> num	T.node = new Leaf(num,num.val)



the following Sequence of function calls create a syntax tree for a – 4 + c

P 1 = mkleaf(id, entry.a)
P 2 = mkleaf(num, 4)
P 3 = mknode(-, P 1 , P 2)
P 4 = mkleaf(id, entry.c)
P 5 = mknode(+, P 3 , P 4)

Polish Notation:

- It is a linearization of abstract syntax tree.
- Most natural way of representing an expression evaluation
- It is also called as prefix expression (operator operand1 operand2)
- In this representation operator can be easily associated with the corresponding operands.

Example:

(a+b)*(c-d) => *(+ab)(-cd)
*+ab-cd

Reverse Polish notation (called as postfix expression)

Example:

(a+b)*(c-d) => (ab+)(cd-)*
ab+cd-*

THREE-ADDRESS CODE:

Three address code is a type of intermediate code. It is built from two concepts called addresses and instructions. Given expression is broken down into several separate instructions. These instructions can easily translate into assembly language.

At most three addresses are used to represent any statement.

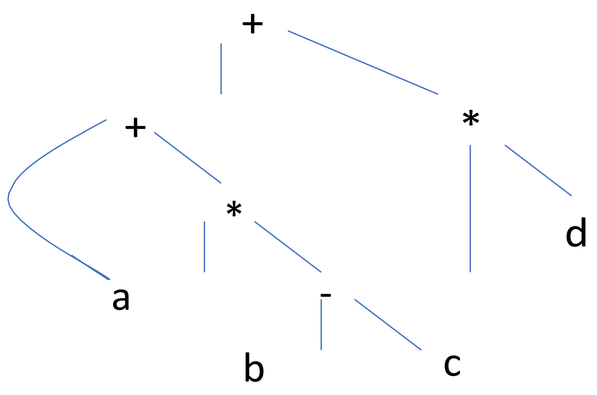
General form is a := b op c

where a, b, c are operands (names, constants, compiler generated temporaries) and op is an operator.

In a three-address code there is at most one operator at the right side of an instruction

Example:

a+a*(b-c)+(b-c)*d	t1 = b – c
a+a*t1+t1*d	t2 = a * t1
a+t2+t1*d	t3 = a + t2
t3+t1*d	t4 = t1 * d
t3+t4	t5 = t3 + t4
t5	



A sample code snippet:

```
do
i = i+1;
while (a[i] < v);
```

Symbolic labels	**Position numbers**
L: t1 = i + 1	100: t1 = i + 1
i = t1	101: i = t1
t2 = i * 8	102: t2 = i * 8
t3 = a[t2]	103: t3 = a[t2]
if t3 < v goto L	104: if t3 < v goto 100

List of the common three-address instruction forms:

- Assignment Instruction with binary operator x = y op z
- Assignment Instruction with unary operator x = op y
- Copy Instruction x = y
- Unconditional Jump goto L
- Conditional Jump if x goto L and if False x goto L
- Conditional Jump with relation operator: if x relop y goto L
- Procedure calls using:
  - param x
  - call p,n
  - y = call p,n
- Indexed copy x = y[i] and x[i] = y
- Address and pointer assignment x = &y and x = *y and *x =y

Data Structures for 3 Address Code:
Three address code is an abstract form of intermediate code. There are three representations used for this.

Quadruples
- It has four fields: op, arg1, arg2 and result
- The op field contains an internal code for the operator

Triples
- A triple has only three fields, which are op, arg1, and arg2.
- Temporaries are avoided by referring the pointer in symbol table (positions)

Indirect triples
- Indirect triples consist of a listing of pointers to triples, rather than a listing of triples themselves.
- So, we use a list of pointers to triples

Example:
Input statement: b * -c + b * -c

Three address code
t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

Quadruples			
op	agr1	agr2	result
minus	c		t1
*	b	t1	t2
minus	c		t3
*	b	t3	t4
+	t2	t4	t5
=	t5		a

	Triples		
	op	agr1	agr2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

35	**0**
36	**1**
37	**2**
38	**3**
39	**4**
40	**5**

	Indirect Triples		
	op	agr1	agr2
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)

Three address code simplifies the task of generating machine code.

TYPES AND DECLARATIONS:

Data type of a name/identifier and type checking of the expression/statement is to be done before Intermediate Code Generation. The applications of types can be grouped under type checking and type translation. Type checking uses logical rules to reason about the behaviour of a program at run time. Specifically, it ensures that the types of the operands match the type expected by an operator.

From the type of a name/identifier,

- A compiler can determine the storage that will be needed for that name/identifier at run time.

Type information is also needed

- to calculate the address denoted by an array reference,
- to insert explicit type conversions, and
- to choose the right version of an arithmetic operator, among other things.

Programming Languages supports two types of data types:

Basic types: integer, character, real, Boolean.
Derived types: Array, structure(record), set and pointer
derived types are built by using basic types.

DECLARATIONS

Types and Declarations are specified by using a simplified grammar that declares just one name at a time.

D -> T id ; D | ∈
T -> B C | record '{' D '}'
B -> int | float
C -> ∈ | [num] C

- Nonterminal D generates a sequence of declarations.
- Nonterminal T generates basic, array, or record types.
- Nonterminal B generates one of the basic types int and float.
- Nonterminal C generates strings of zero or more integers, each integer surrounded by brackets. An array type consists of a basic type specified by B, followed by array components specified by nonterminal C.
- A record type (the second production for T) is a sequence of declarations for the fields of the record, all surrounded by curly braces.

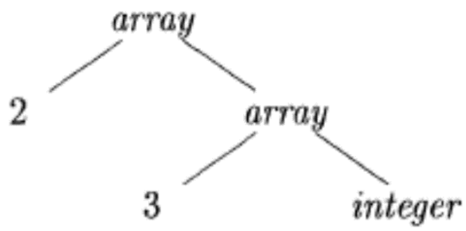
TYPE EXPRESSIONS

Types have structure, represented by using type expressions. A type expression is either a basic type or it is formed by applying an operator called a type constructor to a type expression. Type constructors are derived types (array, structure, pointer, function)

Definitions of type expressions:

- A basic type is a type expression
- A type name is a type expression
- A type expression can be formed by applying the array type constructor to a number and a type expression.
- A record is a data structure with named field.
- A type expression can be formed by using the type constructor **->** for function types
- If s and t are type expressions, then their cartesian product s*t is a type expression
- Type expressions may contain variables whose values are type expressions
- Type expression is represented by using tree or DAG
- Type System: It is a collection of rules for assigning type expression to language constructs. It is implemented by type checker.
- Type checker finds whether the types are compatible to each other or not by using type expression.

Type Expression examples



Array: int array[20] =>array(1,2....19,int)

int[2][3] =>array(2,array(3,integer))

Product: t1 * t2 (* is always left associative)
Struct:

```
struct stud
{
char name[10];
float marks
};
```

struct ((name * array(1,2...9, char)) * (marks float))
struct stud s1[10]; => array(1,2,....9, stud)

Pointer: float *y; => pointer(float)
Function: int sum(int a, int b) => int * int -> int

TYPE EQUIVALENCE

General Form:
"if two type expressions are equal then return a certain type else error."

Ambiguities arise when names are given to type expressions and the names are then used in subsequent type expressions. The key issue is whether a name in a type expression stands for itself or whether it is an abbreviation for another type expression.

When are two type expressions equivalent?

When type expressions are represented by graphs, two types are structurally equivalent if and only if one of the following conditions is true:

- If they are of the same basic type.
- If they are formed by applying the same constructor to structurally equivalent types.
- One is a type name that denotes the other.

If type names are treated as standing for themselves, then the first two conditions in the above definition lead to name equivalence of type expressions.

- Name equivalent expressions are assigned the same value number

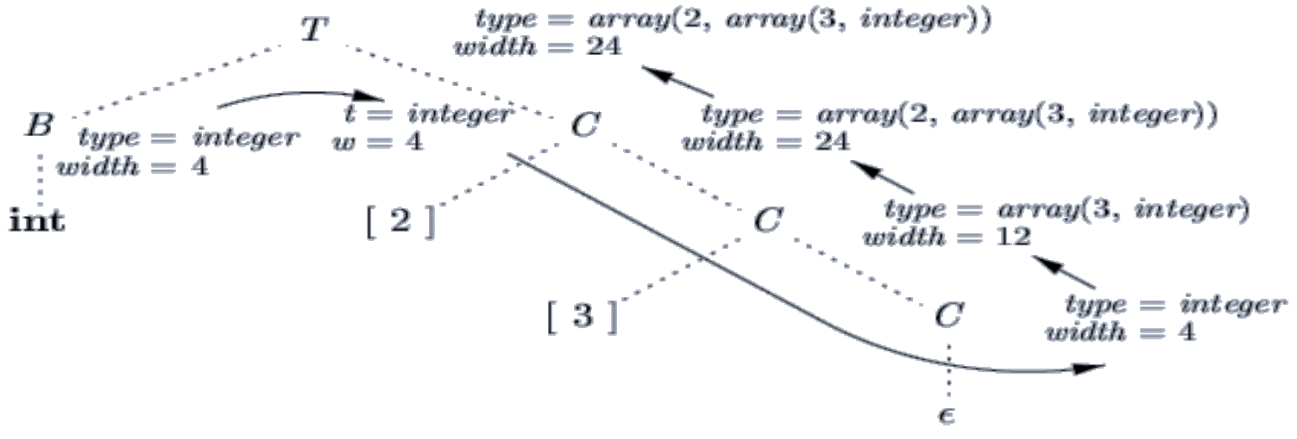
Storage Layout for Local Names

From the type of a name, we can determine the amount of storage that is needed for the name. At compile time, these amounts are used to assign each name a relative address. The type and relative address are saved in the symbol-table entry for the name.

Data of varying length, such as strings, or data whose size cannot be determined until run time, such as dynamic arrays, is handled by reserving a known fixed amount of storage for a pointer to the data.

The width of a type is the number of storage units needed for objects of that type. A basic type requires an integral number of bytes. For easy access, storage for aggregates such as arrays and classes are allocated in one contiguous block of bytes

$$
\begin{array}{lll}
T \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & C & \{\ T.type = C.type;\ T.width = C.width;\ \} \\
B \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
B \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
C \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \} \\
C \rightarrow & [\ \textbf{num}\ ]\ C_1 & \{\ C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & \ \ C.width = \textbf{num}.value \times C_1.width;\ \}
\end{array}
$$



TYPE CHECKING:

To do type checking, a compiler needs to assign a type expression to each component of the source program.

Type checking can take one of the two forms:

- **synthesis**
- **inference**

Type synthesis builds the type of an expression from the types of its subexpressions. It requires names to be declared before they are used.

The type of E1 + E2 is defined in terms of the types of E1 and E2.

A typical rule for type synthesis has the form

if f has type s -> t and x has type s,

then expression f(x) has type t

Here, f and x denote expressions, and s -> t denotes a function from s to t.

This rule is for functions with one argument and to functions with several arguments.

The rule can be adapted for E1 + E2 by viewing it as a function application

add(E1, E2). => E.type = add(E1, E2).

Type inference determines the type of a language construct from the way it is used.

In programming languages, some statements do not have values. So, void type is used.

A typical rule for type inference has the form

if f(x) is an expression,

then for some α and β, f has type α -> β and x has type α

Statements:

The rules for checking statements are similar to those for expressions. For example, we treat the conditional statement "if (E) S;" as if it were the application of a function if to E and S.

S -> if (E) S1 : if E.type = boolean true, S.type = S1.type

Then function if expects to be applied to a boolean and a void; the result of the application is a void.

TYPE CONVERSION

Consider expressions like x + i, where 'x' is of type float, and iI' is of type integer. So, the compiler may need to convert one of the operands of + to ensure that both operands are of the same type when the addition occurs.

For example, the integer 2 is converted to a float in the code for the expression

2 * 3.14:

```
t1 = (float) 2     //explicit
t2 = t1 * 3.14
```

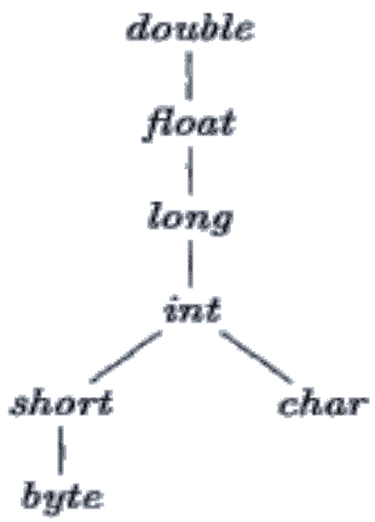
Two types of conversions happen in most of the programming languages

widening conversions, which are intended to preserve information

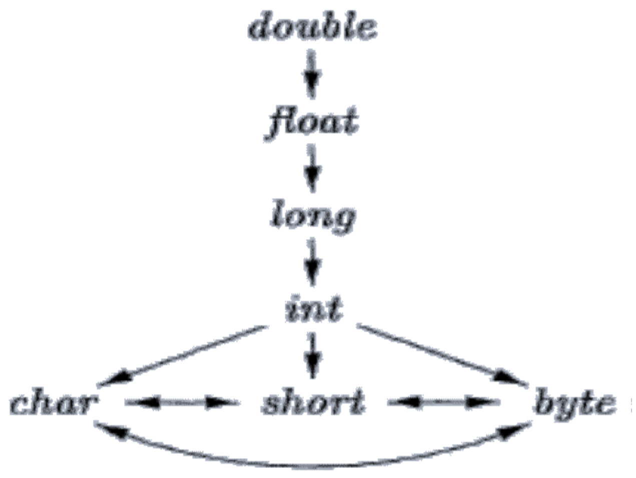
- any type lower in the hierarchy can be widened to a higher type.

narrowing conversions, which can lose information.

- a type 's' can be narrowed to a type t if there is a path from s to t.



(a) Widening conversions



(b) Narrowing conversions

Implicit: Conversion from one type to another is said to be implicit if it is done automatically. Implicit type conversions, also called coercions, are limited in many languages to widening conversions.

Explicit: Conversion is said to be explicit if the programmer must write something to cause the conversion. Explicit conversions are also called casts.

Type checking rules for conversion from integer to float.

```
E -> num          E.type = integer
E -> num.num          E.type = float
E -> id               E.type = look_up(id.entry)
E-> E1 op E2      if ( E1.type = integer and E2.type = integer )
                      E.type = integer;
                  else if ( E1.type = float and E2.type = integer )
                  E.type=float;
                  else if (E1.type = integer and E2.type = float)
                      E.type=float;
                  else if(E1.type = float and E2.type = float)
                      E.type=float;
                  else    type_error
```

TRANSLATION OF EXPRESSIONS:

Translation of expressions and statements into three-address code.

- An expression with more than one operator, like a + b * c, will translate into instructions with at most one operator per instruction.
- An array reference A[i][j] will expand into a sequence of three-address instructions that calculate an address for the reference.

Operation with Expressions

PRODUCTION	SEMANTIC RULES
$S \rightarrow \mathbf{id} = E\ ;$	$S.code = E.code \ \|\|$ $gen(top.get(\mathbf{id}.lexeme)\ '='\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|\ E_2.code \ \|\|$ $gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr)$
$\mid\ -E_1$	$E.addr = \mathbf{new}\ Temp\,()$ $E.code = E_1.code \ \|\|$ $gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr)$
$\mid\ (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$\mid\ \mathbf{id}$	$E.addr = top.get(\mathbf{id}.lexeme)$ $E.code = ''$

top denote the current symbol table. Function **top.get** retrieves the symbol table entry

ARRAYS

Array elements can be accessed quickly if they are stored in a block of consecutive locations (Contiguous). (row major order). In C and Java, array elements are numbered 0, 1, …. n-1, for an array with n elements. If the width of each array element is w, then the i^th element of array A is in location

$$\mathbf{base + i * w}$$

where base is the relative address of the storage allocated for the array. That is, base is the relative address of A[0]
In two-dimensional array, the relative address of A[i1][i2] can be calculated by the formula

$$\mathbf{base} + \mathbf{i_1 * w_1 + i_2 * w_2}$$

for element i2 in row i1, w1 be the width of a row and w2 be the width of an element in a row.
Alternatively, the relative address of an array reference can be calculated in terms of the numbers of elements (n) along dimension (j) of the array and the width (w) is the width of a single element of the array.
In two dimensions, the location for A[i1][i2] is given by

$$\mathbf{base + (i_1 * n + i_2) * w}$$

More generally, array elements need not be numbered from 0 always. In a one-dimensional array, the array elements are numbered low, low + 1….. high and base is the relative address of A[low].
Then the address of **A[i] : base + (i - low) * w.**

The expressions base + i * w and base + (i - low) * w can be rewritten as **i * w + c,** where

the subexpression **c = base - low * w** can be precalculated at compile time. Note that c = base when low is 0.
Once 'c' is precomputed,

For A[i] : i * w + c
for A[i][j] : (i * n + j) * w + c

Compile-time pre-calculation can also be applied to address calculations for elements of multidimensional arrays.
However, if the array's size is dynamic, we cannot use compile-time pre-calculation. If we do not know the values of low and high (or their generalizations in many dimensions) at compile time, then we cannot compute constants such as c.

The above address calculations are based on row-major layout for arrays, which is used in C

Translation of Array References
A grammar for array references: Let nonterminal L generate an array name followed by a sequence of index expressions:

L -> L [E] **|** id [E]

Nonterminal L has three synthesized attributes:

- **L.addr:** L.addr denotes a temporary that is used while computing the offset for the array reference by summing the terms $i_j * w_j$
- **L.array:** L.array is a pointer to the symbol-table entry for the array name. The base address of the array, say, L.array.base is used to determine the actual l-value of an array reference after all the index expressions are analyzed.
- **L.type:** L.type is the type of the subarray generated by L. For any type t, we assume that its width is given by t.width. We use types as attributes, rather than widths since types are needed anyway for type checking.

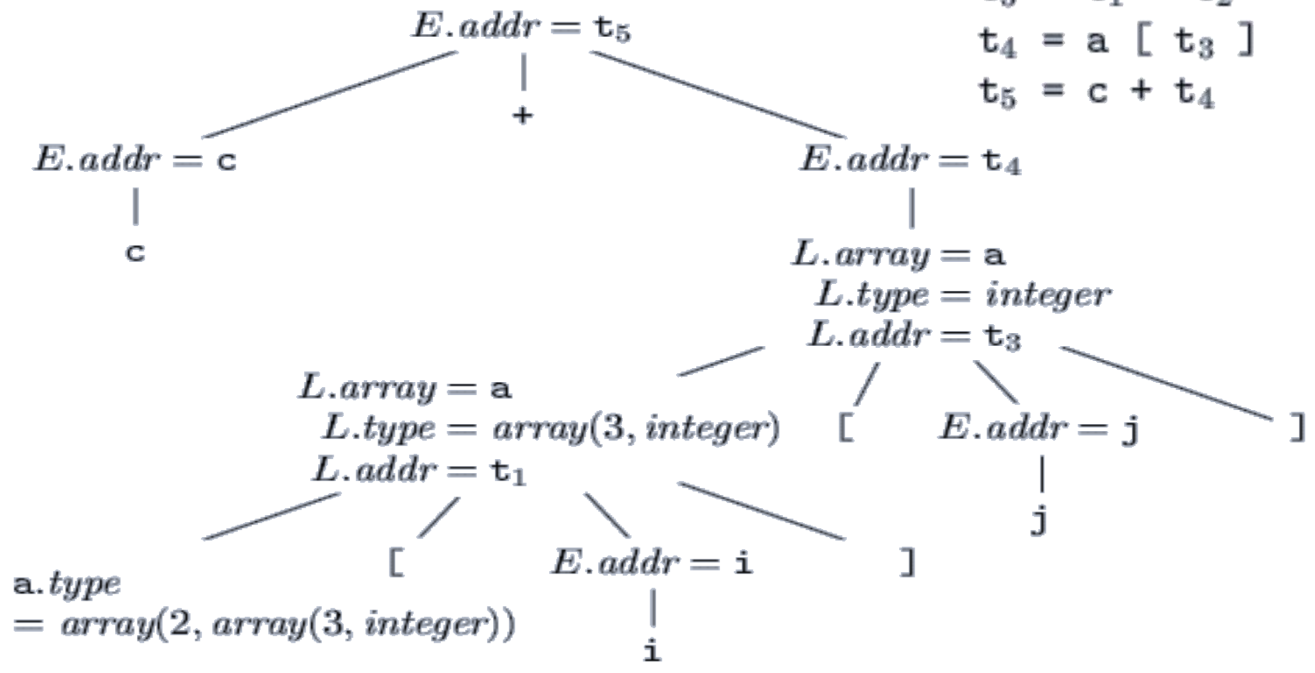
A grammar for array references: Let nonterminal L generate an array name followed by a sequence of index expressions: L -> L [E] **|** id [E]

```
S → id = E ;      { gen( top.get(id.lexeme) '=' E.addr); }

  | L = E ;       { gen(L.addr.base '[' L.addr ']' '=' E.addr); }

E → E1 + E2       { E.addr = new Temp ();
                    gen(E.addr '=' E1.addr '+' E2.addr); }

  | id            { E.addr = top.get(id.lexeme); }

  | L             { E.addr = new Temp ();
                    gen(E.addr '=' L.array.base '[' L.addr ']'); }

L → id [ E ]      { L.array = top.get(id.lexeme);
                    L.type = L.array.type.elem;
                    L.addr = new Temp ();
                    gen(L.addr '=' E.addr '*' L.type.width); }

  | L1 [ E ]      { L.array = L1.array;
                    L.type = L1.type.elem;
                    t = new Temp ();
                    L.addr = new Temp ();
                    gen(t '=' E.addr '*' L.type.width); }
                    gen(L.addr '=' L1.addr '+' t); }
```

Translation of Array References

Annotated parse tree for c + a[i][j]   Three address code

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a [ t3 ]
t5 = c + t4
```



CONTROL FLOW:

The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions.

In programming languages, Boolean expressions are often used to:

Compute logical values.

- A boolean expression can represent true or false as values.
- Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

Alter the flow of control

- Boolean expressions are used as conditional expressions in statements that alter the flow of control.
- The value of such boolean expressions is implicit in a position reached in a program. For example, in if (E) S, the expression E must be true if statement S is reached.

Short Circuit Evaluation

Boolean expressions are composed of the boolean operators (which we denote as &&, ||, and !) (AND, OR, and NOT) applied to elements that are boolean variables or relational expressions.

Relational expressions are of the form E1 rel E2, where E1 and E2 are arithmetic expressions. (<, <=, =, ! =, >, or >= is represented by rel)

Boolean expressions generated by the following grammar:

B -> B || B | B && B | ! B | (B) | E rel E | true | false

For expression B1 || B2, if we determine that B1 is true, then we can conclude that the entire expression is true without having to evaluate B2.

Similarly, given B1 && B2, if B1 is false, then the entire expression is false.

In short-circuit (or jumping) code, the boolean operators &&, ||, and ! translate into jumps. In the translation, new label is created for jumps.

Ex: The statement

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

three address code is

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

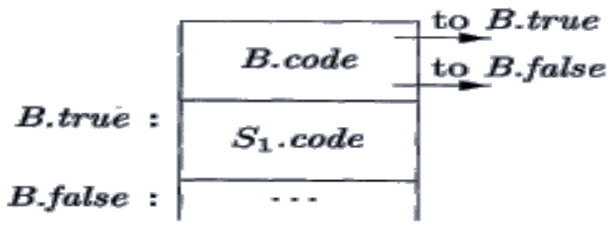
Control-Flow for Boolean statements

PRODUCTION	SEMANTIC RULES
$B \rightarrow B_1 \mathbin{\vert\vert} B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mathbin{\Vert} label(B_1.false) \mathbin{\Vert} B_2.code$
$B \rightarrow B_1 \ \&\& \ B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mathbin{\Vert} label(B_1.true) \mathbin{\Vert} B_2.code$
$B \rightarrow \ ! \ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow E_1 \ \mathbf{rel} \ E_2$	$B.code = E_1.code \mathbin{\Vert} E_2.code$ $\mathbin{\Vert}$ $gen($'if' $E_1.addr$ **rel**.op $E_2.addr$ 'goto' $B.true)$ $\mathbin{\Vert}$ $gen($'goto' $B.false)$
$B \rightarrow$ **true**	$B.code = gen($'goto' $B.true)$
$B \rightarrow$ **false**	$B.code = gen($'goto' $B.false)$

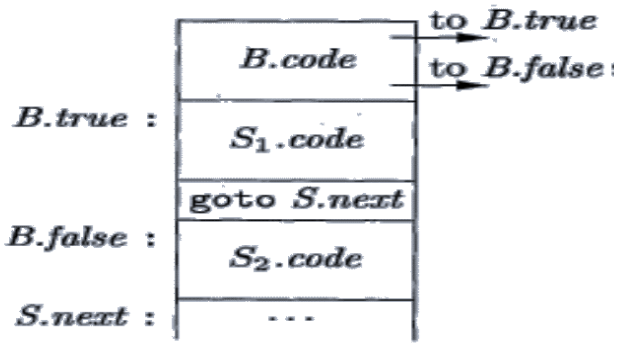
FLOW OF CONTROL STATEMENTS

In the below grammar, nonterminal B represents a boolean expression and non-terminal S represents a statement.

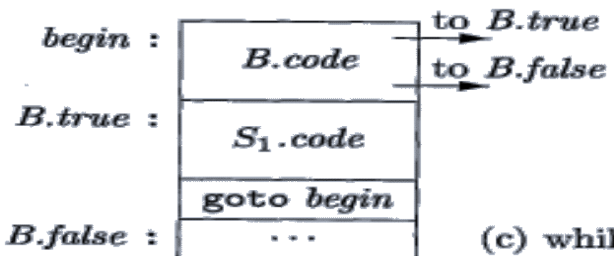
S -> if (B) S1	B.true := newlabel()
| if (B) S1 else S2	B.false := S.next
| while (B) S1	S1.next:= S.next
	S.code:= B.code **||** label(B.true) **||** S1.code



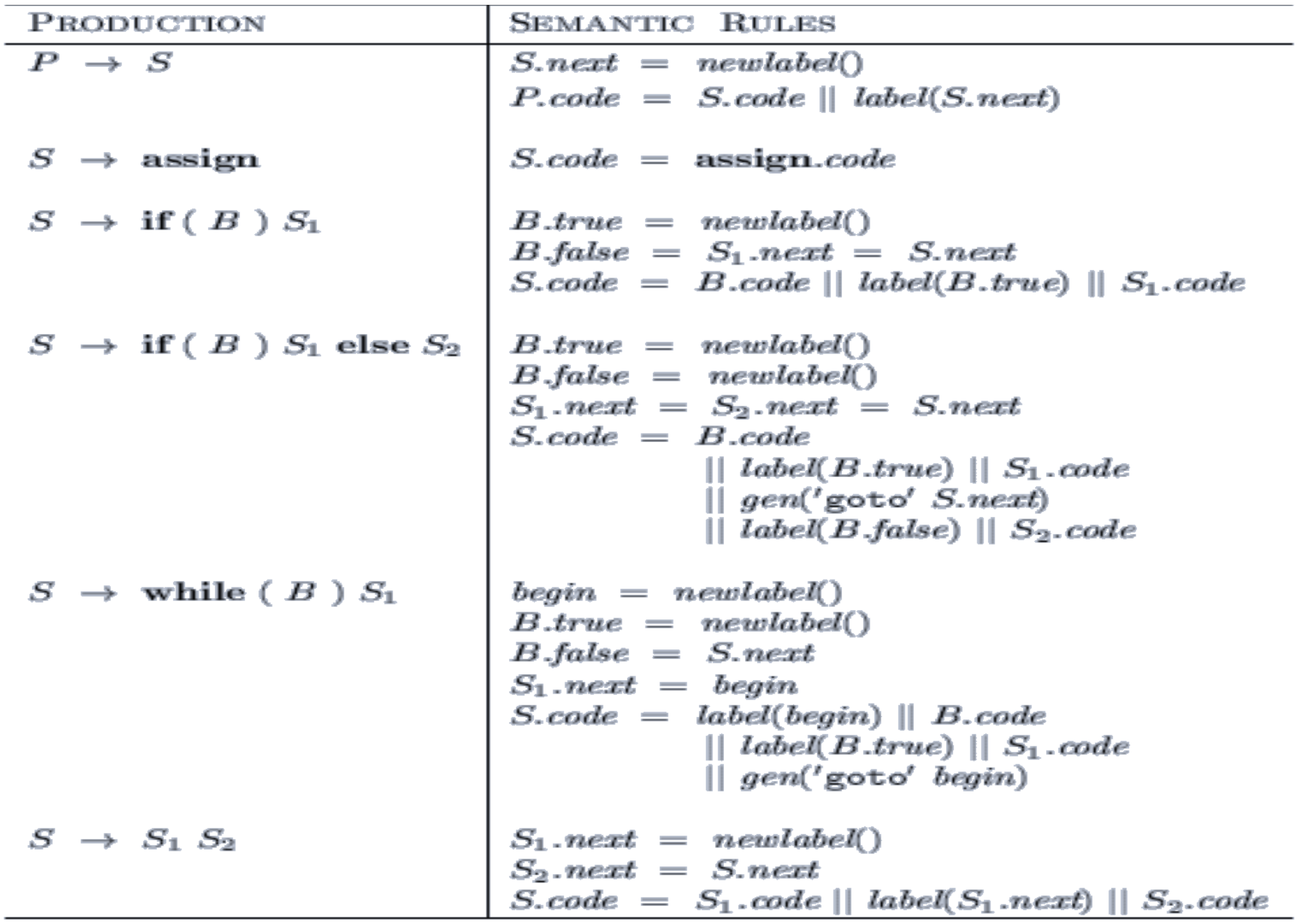
(a) if



(b) if-else



(c) while



PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow \textbf{assign}$	$S.code = \textbf{assign}.code$
$S \rightarrow \textbf{if} \;(\; B \;)\; S_1$	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$
$S \rightarrow \textbf{if} \;(\; B \;)\; S_1 \;\textbf{else}\; S_2$	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}' \; S.next)$ $\|\| \; label(B.false) \;\|\|\; S_2.code$
$S \rightarrow \textbf{while} \;(\; B \;)\; S_1$	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}' \; begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

Three address code for the following control statements:

If (E) S1 else S2

```
    If (E==true) goto L1
    S2
    goto L2
L1: S1
    goto L2
L2: end
```

While (E) S

```
L:if(E==0) goto L1
   S
   goto L
L1:end
```

for(E1; E2; E3) do S

```
     E1
  L:if(E2=true) goto L1
     goto L2
  L1:S
      E3
      goto L
L2:end
```

SWITCH-STATEMENTS:

```
switch(E)
{
case 1: S1
      break;
case 2: S2
      break;
default: S3
}
```

Three address code for switch statement is:

```
          evaluate E value
            goto test
          L1 : S1
            goto Last
          L2 : S2
             goto Last
          L3 : S3
             goto Last
          test: if(E==1) goto L1
         if(E==2) goto L2
       goto L3   //default
Last : end
```

INTERMEDIATE CODE FOR PROCEDURES:
In three-address code, a function call is specified with the below steps

- Evaluation of parameters in preparation for a call (pass by value)
- Function call itself.

Suppose that 'a' is an array of integers, and that 'f' is a function from integers to integers. Then, the assignment

n = f(a[i]);

might translate into the following three-address code:

```
1) t1 = i * 4
2) t2 = a [ t1 ]
3)  param t2
4)  t3 = call f, 1
5)  n = t3
```

Ex: Write three Address code for the following Function Definition

```
int f (int x, int y)
{
return x + y + 1;
 }
Function call : f(2 + 3, 4)
```

1) t1 = x
2) t2 = y
3) param t1
4) param t2
5) t3 = call f, 2 // 2 indicates no. of arguments of function

BACKPATCHING:
Need of Backpatching
For logical OR operation: Example: x = B1 || B2 (B1 & B2 are two Boolean expressions)

In the above statement if B1 produces true, then x is true. (Short circuit evaluation) (Finished in one pass but identifying the next label requires second pass)

if B1 produces false, then we must wait for the completion of B2 to decide the values of x. (Need two passes for completion and label identification)

For logical AND operation: Example: y = B1 && B2 (B1 & B2 are two Boolean expressions)
if B1 produces false, then y is false. (Finished in one pass but identifying the next label requires second pass)

if B1 produces true, then we must evaluate B2 to get the value of y
(Need two passes for completion and label identification)

Backpatching
Generation of three address code in a single pass creates a problem in identifying addresses of the label statements. (Jump statements(goto)) in Boolean and flow-of- control statements.

Definition: Backpatching is the activity of filling up unspecified/missing information of labels using appropriate semantic actions during the code generation phase.

In Boolean expressions we must insert symbolic labels for jumps. So, Boolean expressions needs a separate pass to set them to appropriate addresses for labels.

A technique named backpatching is needed to avoid the need of two passes. Here we save the sequence of instructions into an array and labels will be indices of the array

For nonterminal we use two attributes truelist and falselist.

truelist: it contains the addresses of the true instructions
f**alselist**: it contains the addresses of the false instructions

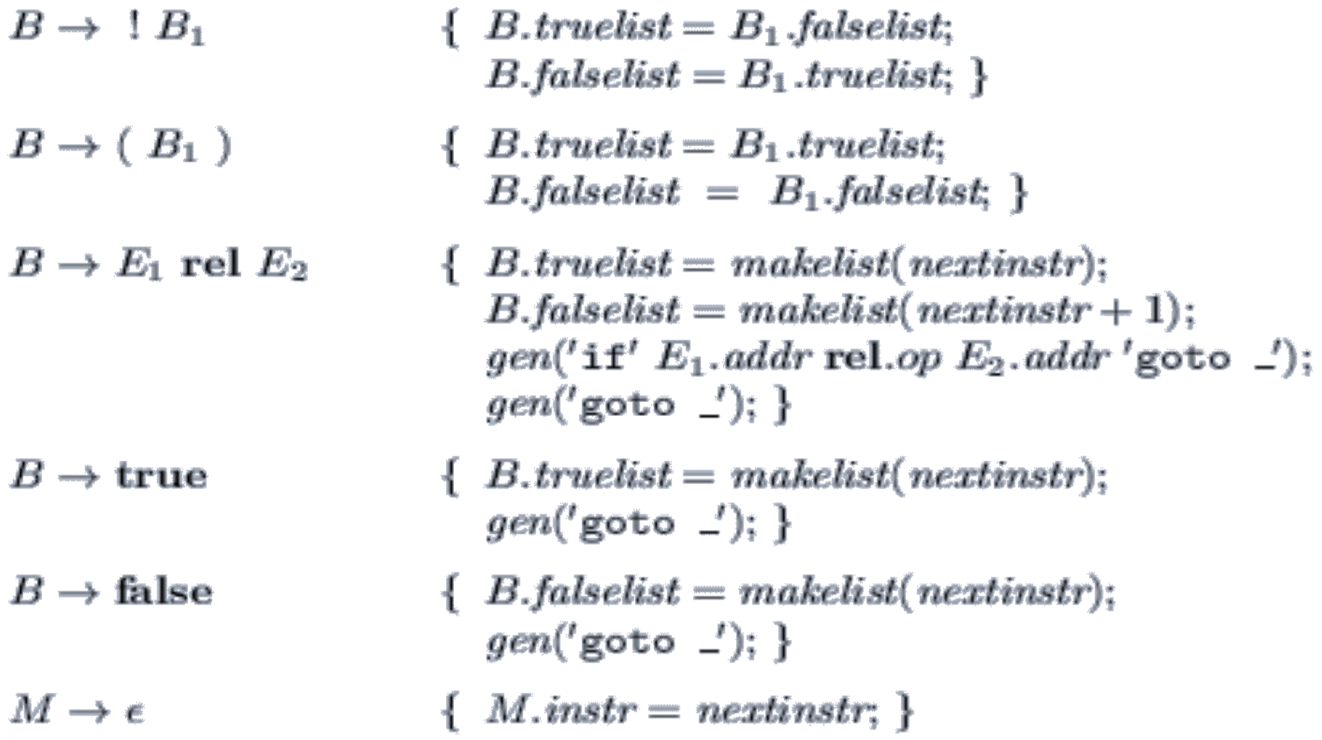
A new non-terminal called M (marker) M -> ϵ, which gives the address of missing/unspecified label

To generate code using backpatching, the following functions are used in semantic

actions of Boolean expressions:
makelist(i): create a new list containing only I, an index into the array of instructions
merge(p1,p2): concatenates the lists pointed by p1 and p2 and returns a pointer to the concatenated list
backpatch(p,i): inserts i as the target label for each of the instruction on the list pointed to by p

Consider this grammar for Boolean and relation expressions
B -> B1 || M B2 | B1 && M B2 | ! B1 | (B1) | E1 rel E2 | true | false
M -> ϵ (here B, B1, B2 are Boolean expressions)

```
B -> B1 || M B2
{
backpatch(B1.falselist, M.instruction)
B.truelist = merge(B1.truelist, B2.truelist) // if B1 is true or B2 is true, or both are
                                                 true, then B is true)
B.falselist = B2.falselist
}

B -> B1 && M B2
{
backpatch(B1.truelist, M.instruction)
B.truelist = B2.truelist    //if B1 is true, the B2 must be true to make B as true
B.falselist = merge(B1.falselist, B2.falselist) // if B1 is false or B2 is false, or both
                                                    are false, then B is false)
}
```



$B \rightarrow\ !\ B_1$	$\{\ B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;\ \}$
$B \rightarrow (\ B_1\ )$	$\{\ B.truelist = B_1.truelist;$ $B.falselist\ =\ B_1.falselist;\ \}$
$B \rightarrow E_1\ \mathbf{rel}\ E_2$	$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $gen(\text{'}\mathtt{if}\text{'}\ E_1.addr\ \mathbf{rel}.op\ E_2.addr\ \text{'}\mathtt{goto}\ \_\text{'});$ $gen(\text{'}\mathtt{goto}\ \_\text{'});\ \}$
$B \rightarrow \mathbf{true}$	$\{\ B.truelist = makelist(nextinstr);$ $gen(\text{'}\mathtt{goto}\ \_\text{'});\ \}$
$B \rightarrow \mathbf{false}$	$\{\ B.falselist = makelist(nextinstr);$ $gen(\text{'}\mathtt{goto}\ \_\text{'});\ \}$
$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

Backpatch for control-flow statements

Normal grammar for control flow statements

> **S -> if (B) S1 | if (B) S1 else S2 | while (B) S | { L } | A ;**
> **L -> L S | S**

Backpatch grammar for control flow statements

> **S -> if (B) M S | if (B) M1 S1 N else M2 S2 | while M1(B) M2 S**
> **| { L } | A ;**
> **L -> L M S | S**

RUN-TIME ENVIRONMENT:

A compiler must accurately implement the abstractions like names, scopes, bindings, data types, operators, procedures, and flow-control statements of source language. Compiler must cooperate with Operating System and other system software's to implement the above abstractions. So, the compiler creates and manages a run-time environment in which it assumes its target programs are being executed.

This environment deals with:

- Layout and storage allocations for objects in source program
- Mechanism of accessing variables by target program
- Linkage between procedures
- Parameter passing mechanism
- Interface to OS, I/O and other programs
- Storage allocation and access to variables and data
- Memory management (stack and heap)
- Garbage collection

Various language features that affect the organization of memory:

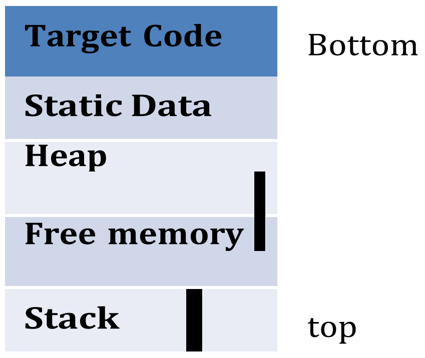
1) Does source language support recursion?
- Several instances of procedures are in active
- Memory allocation is to be done for every instance

2) How are parameters passed to procedures?
- Memory allocation for different parameter passing techniques is different.

3) Does the procedures refer nonlocal names?
- Often procedures have access to its local names. But access to nonlocal name is required.

4) Does the language support allocation and deallocation of memory dynamically?
- Effective utilization of memory is possible.

STORAGE ORGANIZATION:



Compiler demands a block of memory from Operating system to run the compiled program. This block is called as run time storage.

Run-time storage comes in blocks of contiguous bytes. It is divided into parts to hold code and data

- Generated target code
- Data objects
- Information which keeps track of procedure activations

- Size of the generated target code is fixed (so, placed in static space)
- Memory required for data objects is known at compile time (so, placed at static space of memory). These are placed on top of target code.
- Stack is used to manage active procedures
- Heap stores other information
- Size of stack and heap is not fixed.

STORAGE ALLOCATION STRATEGIES:

Based on the division of run time storage, there are three different storage allocation strategies.

Static Allocation:

Allocation is for all the data objects at compile time

Stack Allocation:

Stack is used to manage the run time storage

Names local to a procedure are allocated space on a stack

Heap Allocation:

It is used to manage the dynamic memory allocation

Static Allocation:

If the size of data objects is known at compile time, names are bound to storage at compile time only. So, allocation of objects is done by static allocation.

Memory allocated for objects won't be changes in rum time, hence this allocation is called static. In static allocation, it is easy for the compiler to find the address of the objects in activation record. At compile time, compiler can fill the addresses at which target code can the data it operates on.

- Fortran uses static allocation strategy.

Limitations:

- Knowing the size of data object at compile time is mandatory
- Data structures cannot be created dynamically (cannot manage runtime

allocation of memory)

- Recursive procedures are not supported

Stack Allocation:
Stack is used for storage allocation. This stack is called as control stack. Almost all languages have procedures/functions/methods as units of user-defined actions.

Each time a function is called (activation), space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped out of the stack.

- So, local variables are stored in activation record and bound to it.
- Data structures can be created dynamically.

Limitation:

- Stack allocation would not be feasible if function calls (or activations of function) did not nest in time.
- Memory addressing is slow because of pointers and index registers.

Heap Allocation:
If nonlocal variables must be retained even after the activation record, then heap allocation is required. (Not possible in stack allocation). Heap allocation allocates continuous blocks of memory when required for storage of activation records or for other data objects. It can be deallocated when activation ends, and it can be reused by heap manager. Linked list implementation is suitable for efficient heap management.

STACK ALLOCATION OF SPACE:
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. Information needed for each execution of a procedure is stored in a record called activation record.

Fields of a general activation record are:
Temporaries: Hold temporary values of expressions
Local Data: Belongs to the procedure under execution
Saved Machine Status: Holds the information regarding the status of machine just before the procedure is called. Contains machine registers and program counter.
Control Link: Pointing to the activation record of caller.
Access Link: (optional) refers to nonlocal data in other records.
Return Values: Space for the return value of the called function, if any.
Actual Parameters: Holds the parameters that are

Actual Parameters
Return Values
Control Link (Dynamic)
Access Link (Static)
Saved Machine Status
Local Data
Temporaries

passed to called procedure.

Size of each activation record is determined at the time when a procedure is called.

ACCESS TO NONLOCAL DATA ON THE STACK:
Storage allocation is done for two types of data: Local and Non-Local data

If a procedure refers to variables that are not local to it, then such variables are called non-local variables.

There are languages that don't support nesting of procedures, then non-locals are nothing but global variables (C language). In such languages, global variables are placed in static storage, storage and location is known at compile time. Accessing these variables is easy.

- Static Scope is used to access non-local names

There are languages that support nesting of procedures. In these languages non-local data are handled by using scope information.

There are two types of scope rules,

- Static scope (used by block structured languages)
- Dynamic scope (used by non-block structured languages)

Static Scope/Lexical Scope:
Scope is verified by examining the text of the program. If procedures are nested, nesting depth of the procedure is used. Main procedure nesting depth is 1 and add 1 to every nested procedure.
The lexical scope of nested procedures can be implemented by using Access links and Display.
Access Link: add access link to activation record

- If procedure 'p' is nested in procedure 'q', then access link of procedure 'p' points to access link of most recent activation record of 'q'.
- To access non-local name(x) in a procedure 'p', if 'np' is the nesting depth of 'p' and 'nx' is the nesting depth of 'x', then if nx <= np, the storage for 'x' is foundby following np-nx number of access links.

Code to setup access links

- Procedure 'p' at nesting depth 'np' calls a procedure 'q' at nesting depth 'nq'
- Case1: np < nq, 'q' access link points to 'p' activation
- Case2: np > nq, then follow np-nq+1 links

Display: Faster to access non-local names
It is an array of pointers that points to activation records.

When a new activation record for a procedure at nesting depth 'i' is setup:

- Save the value of d[i] in the new activation record and set d[i] points to the new activation record.
- Before the activation ends d[i] is set to saved value.

Dynamic Scope: used in non-block structured languages (LISP)

- By considering the current activation, it determines the scope of declaration of the names at runtime
- In this type, the scope is verified at runtime

Deep Access: The idea is to keep a stack of active variables. Use control links instead of access links and to find a variable, search the stack from top to bottom

Shallow Access: The idea to keep a central storage and allot one slot for every variable name

Example:

```
#define a (x+1)
int x=2;
void b() { x=1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b();
                c();
               }
output:
2
3
```

PARAMETER PASSING:
If one procedure calls another, communication between them is made by using non-local names and parameters

- **Actual parameters:** variables specified in function call
- **Formal parameters:** variables declared in the function definition

L-value refers to the storage
R-value: refers to value contained in the storage

Parameter Passing Techniques:

1) **call by value:** R-values are passed to formals

example:

- Actual variables are evaluated by caller.

- R-values are passed to formals

```
void swap(int x, int y)
{
int  t;
t = x;
x = y;
y = t;
}

void main()
{
int a = 10, b = 20;
swap(a, b); // call by value
printf("a = %d, b = %d", a, b);
}
```

2) **call by reference:** L-values are passed to formals

Example:

- Actual variables are evaluated by caller.
- L-values are passed to formals

```
void swap(int *x, int *y)
{
int  t;
t = *x;
*x = *y;
*y = t;
}

void main()
{
int a = 10, b = 20;
swap(&a, &b); // call by reference
printf("a = %d, b = %d", a, b);
}
```

3) **copy restore:**

- hybrid (call by value and call by reference)
- Also called as copy-in copy-out
- R-values are passed to formals, before the call L-values are determined
- When control returns to caller, R-value of formal are copied into l-value of actual

4) **call by name**
   - Actual parameters are substituted for the formals
   - Macro expansion/inline expansion

HEAP MANAGEMENT:

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

- Local variables typically become inaccessible when their procedures end.
- C++ and Javas new operator creates objects that may be passed or pointers to them may be passed from procedure to procedure whose existence is not tied with activation record. So, they continue to exist long after the procedure that created them is gone. Such objects are stored on a heap.

Memory manager is a subsystem that allocates and deallocates space within the heap. It serves as an interface between application programs and the operating system. The memory manager always keeps track of the free space in heap storage.

It performs two basic functions: Allocation and Deallocation

Allocation: When a program requests memory for a variable or object, the memory manager produces a chunk of contiguous heap memory of the requested size from the free space of the heap, if no chunk of the needed size is available, it seeks to increasethe heap storage space by getting consecutive bytes of virtual memory from the operating system. If space is exhausted, the memory manager passes that information back to the application program.

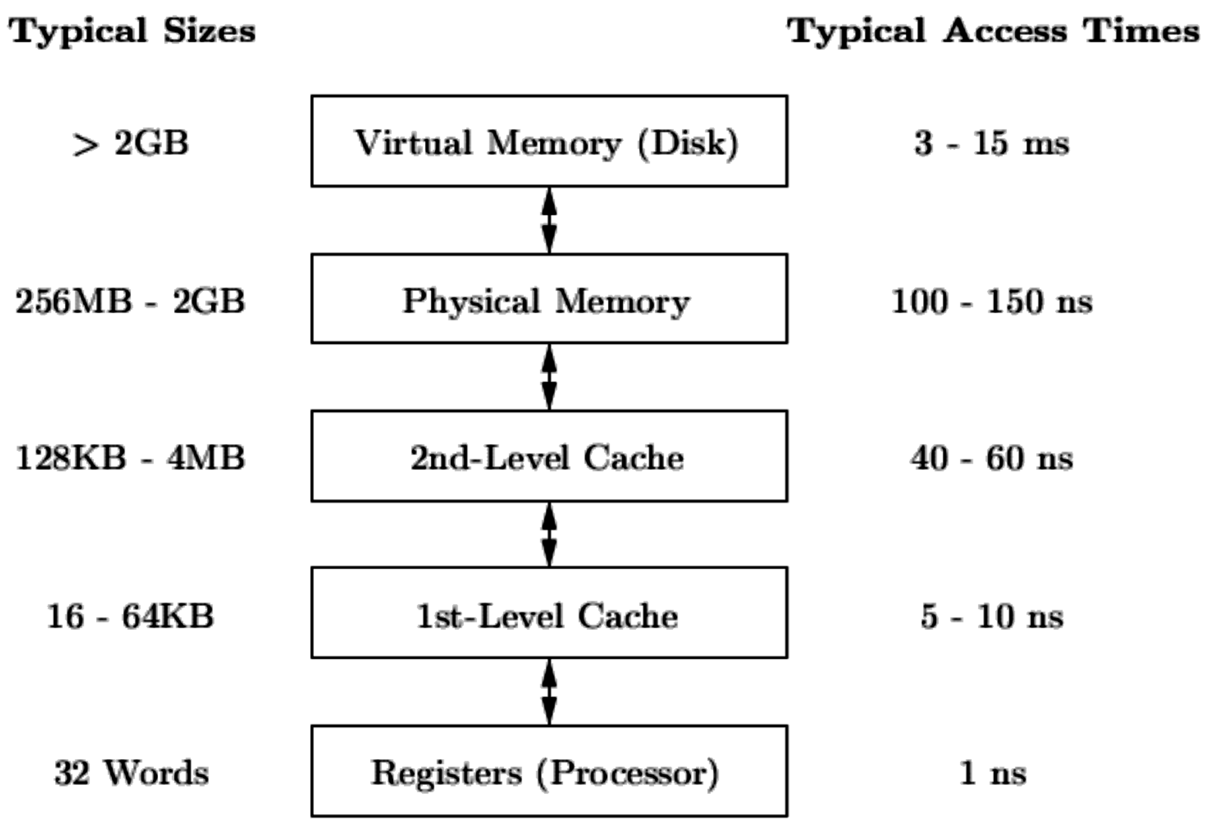
Deallocation: The memory manager returns deallocated space to the pool of free space, so it can reuse the space to satisfy other allocation requests. Memory managers typically do not return memory to the operating system, even if the program's heapusage drops.

Memory management would be simpler if
(a) all allocation requests were for chunks of the same size, and
(b) storage was released predictably, say, first-allocated first-deallocated.

Desirable properties of memory manager are

1) Space Efficiency, 2) Program Efficiency, 3) Low overhead

Memory Hierarchy



GARBAGE COLLECTION:

Data that cannot be referenced is generally known as garbage. Automatic garbage collection is used in many high-level languages.

A user program, which we shall refer to as the mutator, modifies the collection of objects in the heap. The mutator creates objects by acquiring space from the memory manager.

Objects become garbage when the mutator program cannot reach them. The garbage collector finds these unreachable objects and reclaims their space by handing them to the memory manager

Design Goals for automatic Garbage Collectors

Type Safety

- A language in which the type of any data can be determined is said to be type safe.
- Statically typed languages/ Statically type safe (ML at compile time)
- Dynamically typed languages/ Dynamically type safe (JAVA at runtime)
- Neither statically nor dynamically type safe, then it is said to be unsafe. (C & C++)

Performance Metrics

- Garbage collection is often so expensive
- Metrics: Overall Execution Time, Space Usage, Pause Time, Program Locality.

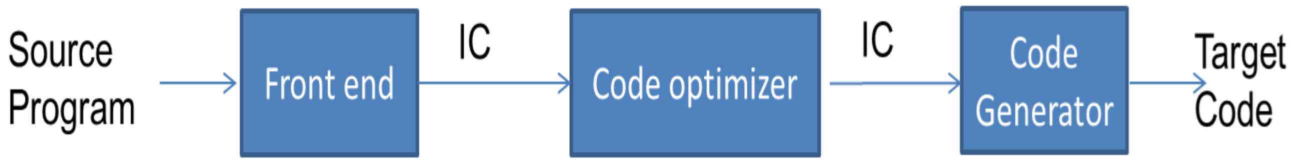
1. Reference counting collectors
2. Trace-based collectors
   - Incremental Garbage Collection
   - Incremental Reachability Analysis
   - Partial-Collection Basics
   - Generational Garbage Collection

UNIT-V

Code Generation: Issues in the Design of a Code Generator, The Target Language, addresses in the Target Code, Basic Blocks and Flow graphs, Optimization of Basic Blocks, Peephole Optimization, Register Allocation andAssignment.
Machine-Independent Optimizations: The Principal Sources of Optimizations, Introduction to Data-Flow Analysis.

CODE GENERATION:

The final phase of a compiler is code generator. Code generation is the process ofcreating machine language
It receives an intermediate representation (IR) from front end of compiler along with supplementary information in symbol table and produces a semantically equivalent target program/object code.



Properties desired by an object code generation phase:

- Correctness
- High Quality
- Efficient use of resource of target machine
- Quick code generation

Output of code generation phase is machine code/object code

Forms of object code

- **Absolute Code:** Machine code that contains references to actual addresses within programs address space. It can be placed directly in the memory and execution starts immediately.
    - Examples: WATFIV and PL/C produces Absolute code
- **Relocatable machine code:** It allows subprograms to be compiled separately. Allows linking and loading of already compiled subroutines.
- **Assembler code:** It makes code generation process somewhat easier. Slower (because, assembling, linking, and loading is required)

ISSUES IN THE DESIGN OF A CODE GENERATOR:

The most important criterion for the target code is
Correct code: The target program must preserve the semantic meaning of the source program and be of high quality

- Correctness
- High-quality
- Efficient use of resources of target code

- Quick code generation

Input to the code generator

- IR + Symbol table

We assume front end produces low-level IR, i.e., values of names in it can be directly manipulated by the machine instructions. Syntactic and semantic errors have been already detected.

The target programs

Common target architectures are RISC, CISC and Stack based machines. We use a very simple RISC-like computer with addition of some CISC-like addressing modes

Code generator main tasks:

- **Instruction selection**

  The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by factors such as
  - The level of the IR
  - The nature of the instruction-set architecture.
  - The desired quality of the generated code.

  Selection of instruction depends on the instruction set of target machine. Speed of instruction and machine idioms are two important factors in selection of instruction.

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

Example:

```
x = y  +  z
LD     R0, y
ADD R0, R0, z
ST     x, R0
```

```
             LD    R0, b
a=b+c        ADD  R0, R0, c
d=a+e        ST    a, R0   ⎫
                           ⎬ Useless
             LD    R0, a   ⎭
             ADD R0, R0, e
             ST    d, R0
```

- **Register allocation and assignment**

  A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values.

  Two subproblems are:

  Register allocation: selecting the set of variables that will reside in registers at each point in the program

  Resister assignment: selecting specific register that a variable resides in.

  Complications imposed by the hardware architecture.

  Example:
  - register pairs for multiplication and division

  The product occupies the entire even/odd register pair.

  After division, the even register holds the remainder and the odd register the quotient.

- **Instruction ordering**

Evaluation order is an important factor in generating an efficient target code. Some orders require a smaller number of registers to hold intermediate resultsthan the other orders.

- Picking up the best order is difficult.
- Picking a best order is another NP- Complete
- Mostly, the order in which the three-address code is generated, same order is followed.

THE TARGET LANGUAGE:

Prior knowledge about the target machine and its instruction set is a prerequisite for designing a good code generator. We use a simple target language assembly code for a simple computer.

A Simple Target Machine Model

- **Load operations**: The instruction LD dst, addr loads the value in location addr into location dst.
  - **LD r, x** (location x into register) and **LD r1, r2** (register-to-register)
- **Store operations**: **ST x, r** (stores the value in register r into the location x).
- **Computation operations**: **OP dst, src1, src2**
  - where OP is an operator like ADD, SUB, MOV and dst, src1, and src2 are locations. ex: **SUB r1, r2, r3** <=> **r1 = r2 - r3**.
- **Unconditional jumps**: **BR L**
  - BR stands for branch. BR L causes the control to move to label L
- **Conditional jumps**: **B*cond* r, L like BLTZ r, L**
  - Where 'r' is a register, L is a label, and ***cond*** stands for any of the common tests on values in the register r.

ADDRESSES IN THE TARGET CODE:

Target machine has a variety of addressing modes.

Addressing Modes	Form	Address	Example	Added Cost
Absolute (location)	M	M	MOV R1, M	1
Register	R	R		0
Indexed address	a(R)	contents(a + contents(R)) (a is a variable)	LD R1, a(R2)	1
Integer Indexed	100(R)	contents(100+contents(R)) (100 is a memory location)	LD R1, 100(R2)	1

Indirect Register	*R	contents(R)		0
Indirect Indexed	*100(R)	contents(contents(100+contnets(R)))	LD R1, *100(R2)	1
Constant/Literal	#100	100	MOV #5, R0	1

Various Address used for three address code are

- **Static Allocation**
  - A statically determined area Code that holds the executable target code. The size of the target code can be determined at compile time.
  - A statically determined data area Static for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.
- **Stack Allocation**
  - A dynamically managed area Stack for holding activation records as they are created and destroyed during procedure calls and returns
- **Dynamic Allocation**
  - A dynamically managed area Heap for holding data objects that are allocated and freed during program execution

Some of the examples of three address codes for statements:

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

```
t1 = z          LD R1, y           // R1 = y
t2 = y          LD R2, z           // R2 = z          LD R1, x
t3 = t1-t2      SUB R1, R1, R2 // R1 = R1 - R2        SUB R1, R1, y
x = t3          ST x, R1          // x = R1           ST x, R1
```

three-address instruction b = a[i]

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

three-address instruction a[j] = c

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

three-address statement x=*p

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

three-address statement *p = y

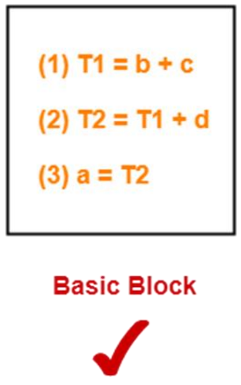
```
LD R1, p          // R1 = p
LD R2, y          // R2 = y
ST 0(R1), R2      // contents(0 + contents(R1)) = R2
```

conditional-jump three-address instruction If x<y goto L

```
LD R1, x                // R1 = x
LD R2, y                // R2 = y
SUB R1, R1, R2          // R1 = R1 - R2
BLTZ R1, M              // if R1 < 0 jump t o M
```

BASIC BLOCKS AND FLOW GRAPHS:

Basic block is a sequence of consecutive statements in which flow of control enters at the beginning of the block and leaves at the end without halt or possibility of branching.



The characteristics of basic blocks are-

- They do not contain any kind of jump statements in them.
- There is no possibility of branching or getting halt in the middle.
- All the statements execute in the same order they appear.
- They do not lose the flow control of the program.

Example:
```
t1 = a * 5
t2 = t1 + 7
t3 = t2 – 5
t4 = t1+ t3
t5 = t2 + b
```
So, the intermediate code is portioned into basic blocks. Each basic blocks become the nodes of a flow graph.

Partitioning Algorithm:
Any given program can be partitioned into blocks using the following two steps

Step-1: First determine the leader statements

Rules for finding leaders

- The first three-address instruction in the intermediate code is a leader.

- Any instruction that is the target of a conditional or unconditional jump (goto) is a leader.
- Any instruction that immediately follows a conditional or unconditional jump(goto) is a leader.

Step-2: Basic block is formed starting at the leader and ends just before the next leader statement.

Example:

```
1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto (3)
10) i = i + 1
11) if i <= 10 goto (2)
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto (13)
```

```
for $i$ from 1 to 10 do
    for $j$ from 1 to 10 do
        $a[i, j] = 0.0;$
for $i$ from 1 to 10 do
    $a[i, i] = 1.0;$
```

- Intermediate code to set a 10*10 matrix to an identity matrix.

As per partition algorithm

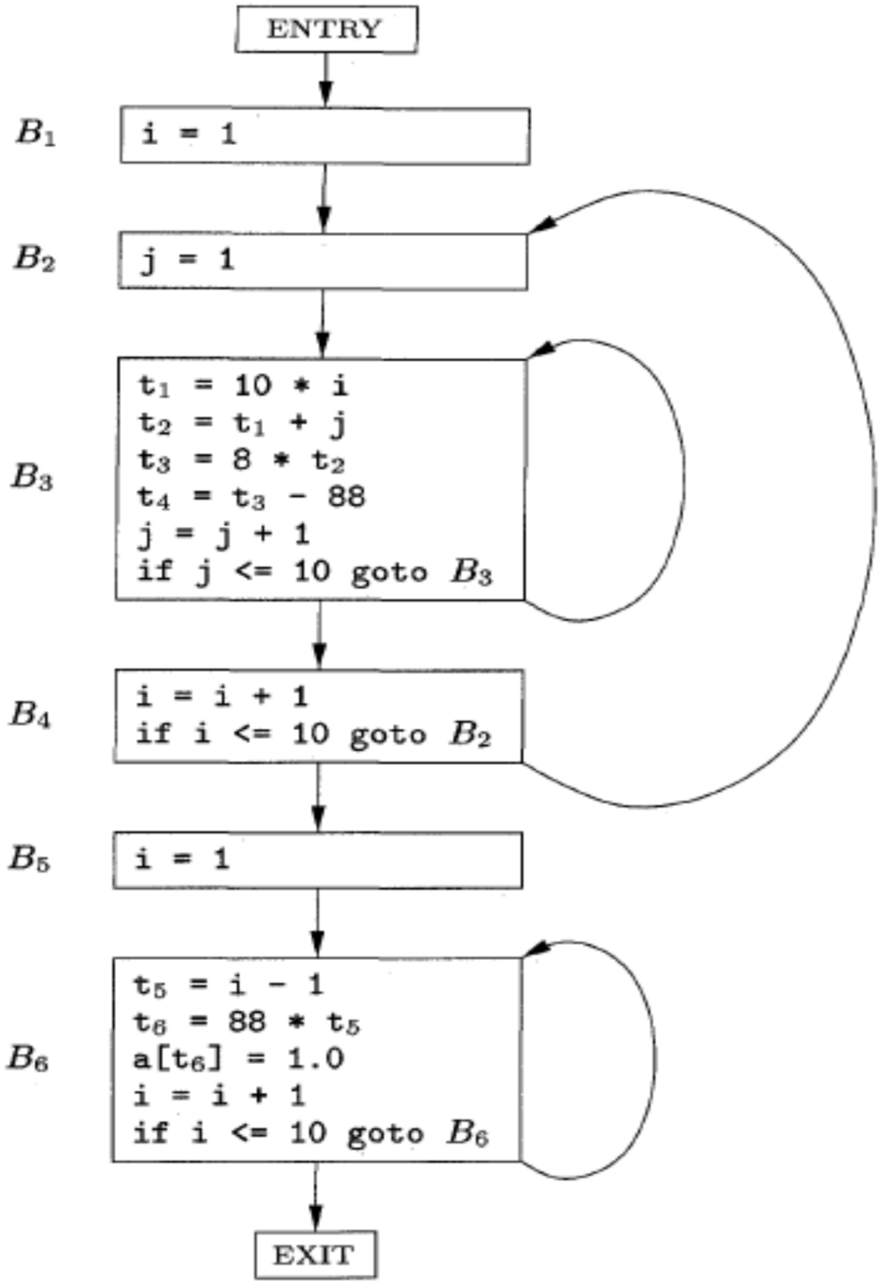
- 1 is a leader by default.
- 9, 11, and 17 are jump statements, their targets are 3, 2, 13. so 3, 2, 13 are leaders.
- 10, 12 are statements that follow jumps.

Leaders: 1, 2, 3, 10, 12, and 13.

Blocks: Each block contains statements from the leader to just before the next leader.

- Usually, we add Entry and Exit blocks also.

Block1: 1st statement
Block2: 2nd statement
Block3: statements from **3 to 9**
Block4: statements **10, 11**
Block5: 12th statement
Block6: 13th statement

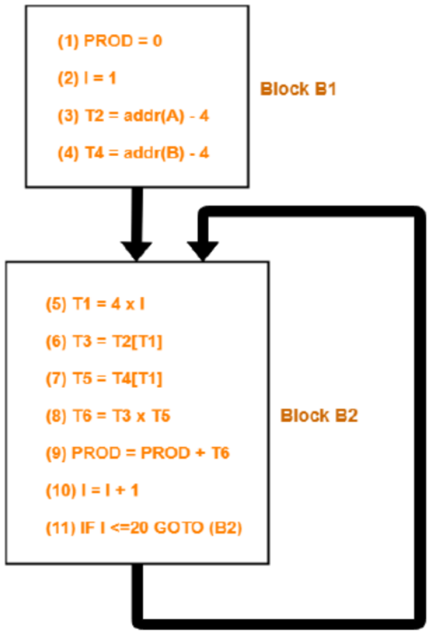


FLOW GRAPH:
Once an intermediate-code program is partitioned into basic blocks, we represent the flow of control between them by a flow graph.

- A flow graph is a directed graph in which the flow control information is added to the basic blocks.
- The nodes of the flow graph are the basic blocks.

There is an edge from block B to block C if the first instruction in block C immediately follow the last instruction in block B.
Example for Glow Graph:

Compute the basic blocks for the given three address statements and draw flow graph
(1) PROD = 0
(2) I = 1
(3) T2 = addr(A) – 4
(4) T4 = addr(B) – 4
(5) T1 = 4 x I
(6) T3 = T2[T1]
(7) T5 = T4[T1]
(8) T6 = T3 x T5
(9) PROD = PROD + T6
(10) I = I + 1
(11) IF I <=20 GOTO (5)



Flow Graph

Solution-
● PROD = 0 is a leader since first statement of the code is a leader.
● T1 = 4 x I is a leader since target of the conditional goto statement is a leader.

OPTIMIZATION OF BASIC BLOCKS:

There are generally two phases of optimization:
Global Optimization: Transformations are applied to large program segments that includes functions, procedures, and loops.

- Machine independent optimization is called as global optimization

Local Optimization: Transformations are applied to small blocks of statements. The local optimization is done prior to global optimization.

- Peephole optimization is the best example for local optimization.

Peephole Optimization: Optimizing the target code of a block.
A simple but effective technique for locally improving the target code is peephole

optimization, which is done by examining a sliding window of target instructions (called the peephole).

Characteristic of peephole optimizations

- Redundant-instruction elimination
    - Eliminating Redundant Loads and Stores
    - Eliminating Unreachable Code
- Flow-of-control optimizations
    - unnecessary jumps can be eliminated

```
goto L1

...

Ll: goto L2
```

Can be replaced by:

```
goto L2

...

Ll: goto L2
```

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

Can be replaced by:

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

    -
- Algebraic simplifications and reduction in strength
    - x = x + 0, x = x * 1, x2 <=> x * x,
- Use of machine idioms
    - Autoincrement and Autodecrement

REGISTER ALLOCATION ANDASSIGNMENT:

Efficient allocation of register is important for generating good code. There are four strategies for deciding what values in a program should reside in a register and which register each value should reside.

1. Global Register Allocation
2. Usage Counts
3. Register Assignment for Outer Loops
4. Register Allocation by Graph Coloring

1. Global Register Allocation:

Simple code generation algorithm does local (block based) register allocation. This resulted that all live variables be stored at the end of block. To save some of these stores and their corresponding loads, we might arrange to assign registers tofrequently used variables and keep these registers consistent across block boundaries (globally)

- Frequently used variables are stored in fixed registers.
- So, assign some fixed registers to hold most active variables in each inner loop.

- Some options are:
  - Keep values of variables used in loops inside registers
  - Use graph coloring approach for more globally allocation

2. Usage Counts:

We can save some cost by keeping a variable x in register for the duration of loop L For the loops we can approximate the saving by register allocation as:

- Sum over all blocks (B) in a loop (L)
- For each use of x before any definition in the block we add one unit of saving
- If x is live on exit from B and is assigned a value in B, then we add 2 units of saving

Usage cost of x:

$$\sum \textbf{blocks B in L use(x, B) + 2 * live(z, B)}$$

use(x,B): number of time x is used in B prior to any definition of x in B.

live(x,B): equals to 1, if x is live on exit of B and assigned a value.
Otherwise equals to 0.

3. Register Allocation for Outer Loops:

If an outer loop L1 contains an inner loop L2, the names allocated registers in L2 need not be allocated registers in L1 - L2.

if we choose to allocate x a register in L2 but not L1, we must load x on entrance to L2 and store x on exit from L2.

4. Register allocation by Graph Coloring:

If all the registers are occupied/in use, if we want a register to store a new variable, one of the register content is to be stored into memory location (spilling). Register allocation by Graph coloring is a simple and systematic technique for register allocation and management.

It is a two-pass method:

1st pass: Target machine instructions are selected with an assumption that there are infinite number of symbolic registers are available

- Names used in three address code become names of symbolic registers
- Three address instructions become machine language instructions

- Once the instruction is selected, 2nd pass assigns physical registers to symbolic registers

2nd pass: For each procedure, a register-inference graph is constructed, in which nodes are symbolic registers, and an edge connects two nodes if one is live at the point where the second is defined.

MACHINE-INDEPENDENT OPTIMIZATIONS:

Code Optimization is an approach to enhance the performance of the code. The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e., CPU, Memory, registers) so that faster-running machine code is generated. Code optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.

The process of code optimization involves-

- Eliminating the unwanted code lines
- Rearranging the statements of the code

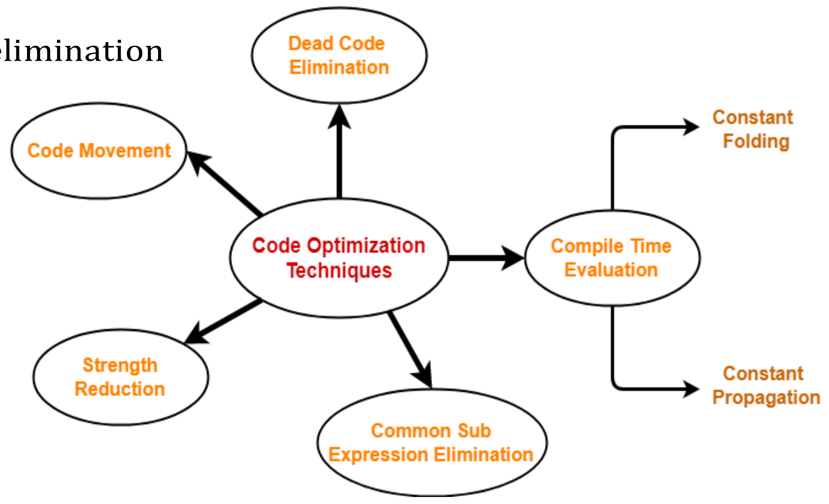
Advantages-

The optimized code has the following advantages-

- Optimized code has faster execution speed.
- Optimized code utilizes the memory efficiently.
- Optimized code gives better performance.

THE PRINCIPAL SOURCES OF OPTIMIZATIONS:

1. Compile Time Evaluation
2. Common sub-expression elimination
3. Dead Code Elimination
4. Code Movement
5. Strength Reduction



1. Compile Time Evaluation-

Two techniques that falls under compile time evaluation are-

A) Constant Folding-

It involves folding the constants.

The expressions that contain the operands having constant values at compile time are evaluated. Those expressions are then replaced with their respective results.

Example-

Circumference of Circle = (22/7) x Diameter

Here,

- This technique evaluates the expression 22/7 at compile time.
- The expression is then replaced with its result 3.14.
- This saves the time at run time.

B) Constant Propagation-

In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

Example-

pi = 3.14
radius = 10
Area of circle = pi x radius x radius

Here,

- This technique substitutes the value of variables 'pi' and 'radius' at compile time.
- It then evaluates the expression 3.14 x 10 x 10.
- The expression is then replaced with its result 314.
- This saves the time at run time.

2. Common Sub-Expression Elimination-

The expression that has been already computed before and appears again in the code for computation is called as Common Sub-Expression.

1. It involves eliminating the common sub expressions.
2. The redundant expressions are eliminated to avoid their re-computation.
3. The already computed result is used in the further program when required.

Example-

Code Before Optimization	Code After Optimization

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i **// Redundant Expression**
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

3. Code Movement-

It involves movement of the code.

- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Example-

Code Before Optimization	Code After Optimization

for (int j = 0 ; j < n ; j ++)
{
x = y + z ;
a[j] = 6 x j;
}
	x = y + z ;
	for (int j = 0 ; j < n ; j ++)
	{
	a[j] = 6 x j;
	}

4. Dead Code Elimination-

It involves eliminating the dead code.

- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

Example-

Code Before Optimization	Code After Optimization

i = 0 ;
if (i == 1)
{
a = x + 5 ;
}
	i = 0 ;

5. Strength Reduction-

It involves reducing the strength of expressions.

- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Example-

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

- The expression "A x 2" is replaced with the expression "A + A".
- This is because the cost of multiplication operator is higher than that of addition operator.

INTRODUCTION TO DATA-FLOW ANALYSIS

Data flow analysis refers to a body of techniques that derive information about the flow of data along program execution paths.

Analysis that determines the information regarding definition and use of data in program. It is useful in global optimization of code.

Each execution of an intermediate-code statement transforms an input state to a new output state. The input state is associated with the program point before the statement and the output state is associated with the program point after the statement.

Basic Terminology:

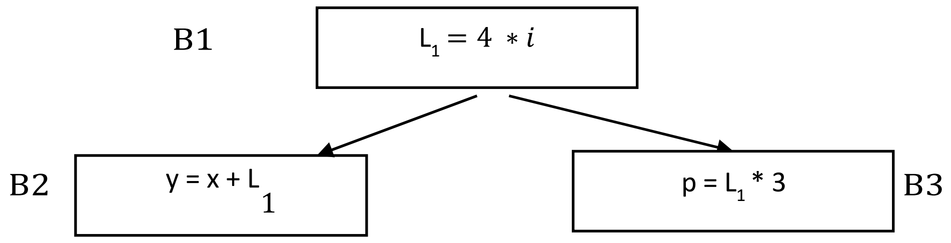
- **Definition Point:** place/line in which a variable is defined.
- **Reference Point:** place/line in which a variable is referred.
- **Evaluation Point:** place/line in which operations (arithmetic) is performed.

Data Flow Properties:

Available Expression: An expression 'a+b' is said to be available at a program point 'x', if none of its operands gets modified before their use.

- It is used to eliminate common sub expressions.

Ex:

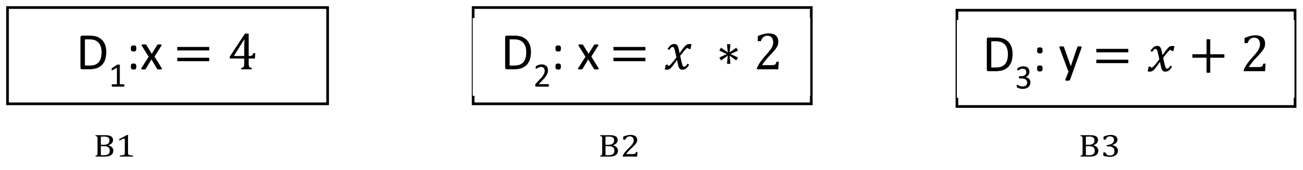


Here 4 * i is available in B2 and B3

Reaching Definition: A definition D is reaching to a point x if D is not killed or redefined before that point.

- It is used in constant/variable propagation.

Ex:



D1 is a reaching definition to B2 not for B3
D2 is a reaching definition to B3