UNIT-1
LANGUAGE TRANSLATION

1. OVERVIEW OF LANGUAGE PROCESSING SYSTEM

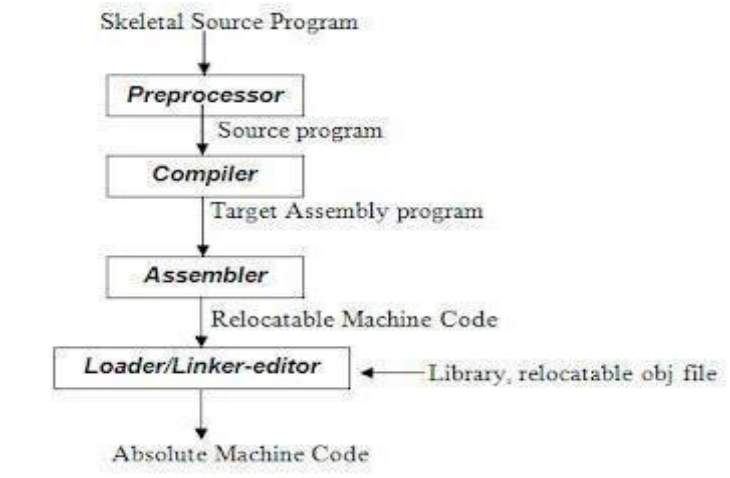


Fig 1.1 Language –processing System

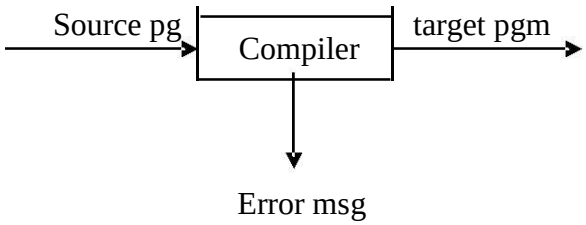
Preprocessor

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

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

Compiler

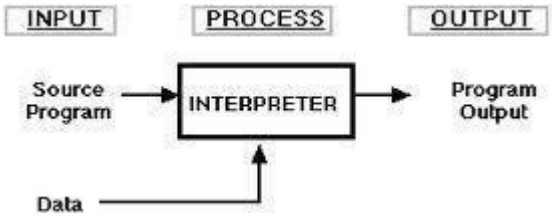
Compiler is a translator program that translates a program written in (HLL) the source program and translates it into an equivalent program in (MLL) the target program. As an important part of a compiler is error showing to the programmer.



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

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

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



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

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

Advantages:

- Modification of user program can be easily made and implemented as execution proceeds.
- Type of object that denotes various may change dynamically.
- Debugging a program and finding errors is simplified task for a program used for interpretation.
- The interpreter for the language makes it machine independent.

Disadvantages:

- The execution of theprogramis slower.
- Memory consumption is more.

Loader and Link-editor:

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

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

TRANSLATOR

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

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

TYPE OF TRANSLATORS:-

- Interpreter
- Compiler
- preprocessor

LIST OF COMPILERS

1. Ada compilers
2. ALGOL compilers
3. BASIC compilers
4. C# compilers
5. C compilers
6. C++ compilers
7. COBOL compilers
8. Java compilers

2. PHASES OF A COMPILER:

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

There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)

b. Synthesis (Machine Dependent/Language independent) Compilation process is partitioned into no-of-sub processes called **'phases'**.

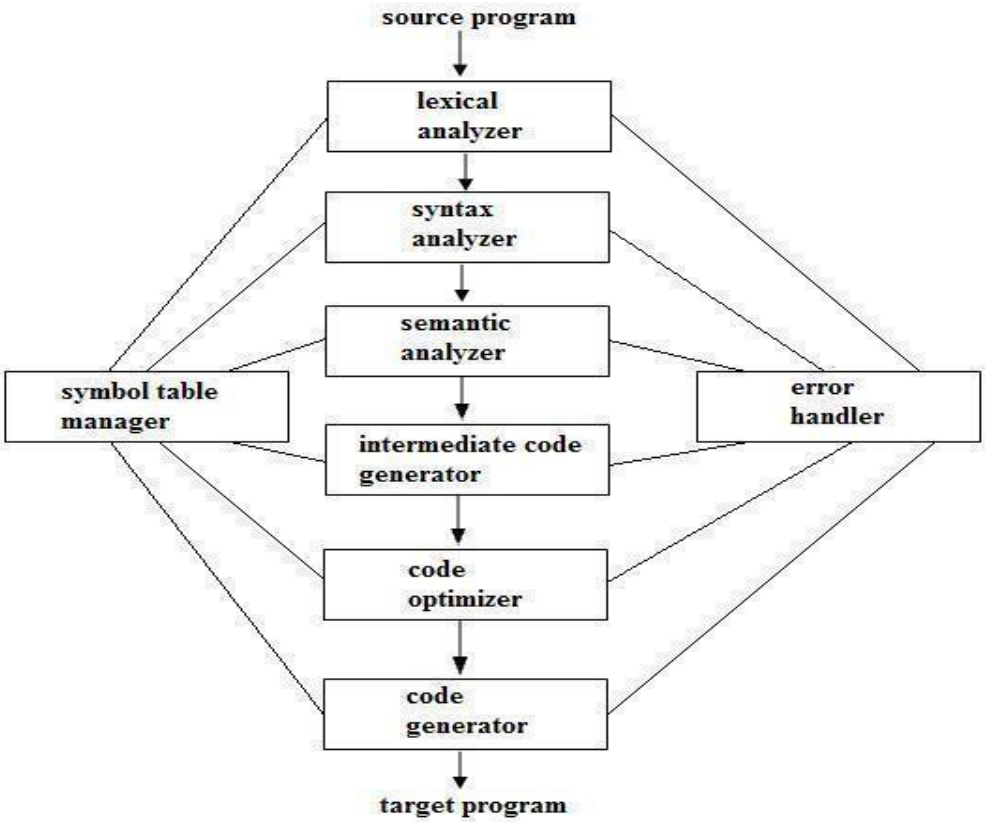


Fig 1.5 Phases of a compiler

Lexical Analysis:-

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

Syntax Analysis:-

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

Intermediate Code Generations:-

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

Code Optimization:-

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

Code Generation:-

The last phase of translation is code generation. A number of optimizations to

Reduce the length of machine language program are carried out during this phase. The output of the code generator is the machine language program of the specified computer.

Table Management (or) Book-keeping:-

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

Error Handlers:-

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

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

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

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

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

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

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

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

Intermediate Code Generation:-

The intermediate code generation uses the structure produced by the syntax analyzer to create a stream of simple instructions. Many styles of intermediate code are

possible. One common style uses instruction with one operator and a small number of operands.The output of the syntax analyzer is some representation of a parse tree. The intermediate code generation phase transforms this parse tree into an intermediate language representation of the source program.

Code Optimization:-

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

/* 1, Local Optimization:-

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

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

Goto **L3 L2 :**

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

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

A := B + C + D

E := B + C + F

Might be evaluated as

T1 := B + C

A := T1 + D

E := T1 + F

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

Loop Optimization:-

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

Code generator :-

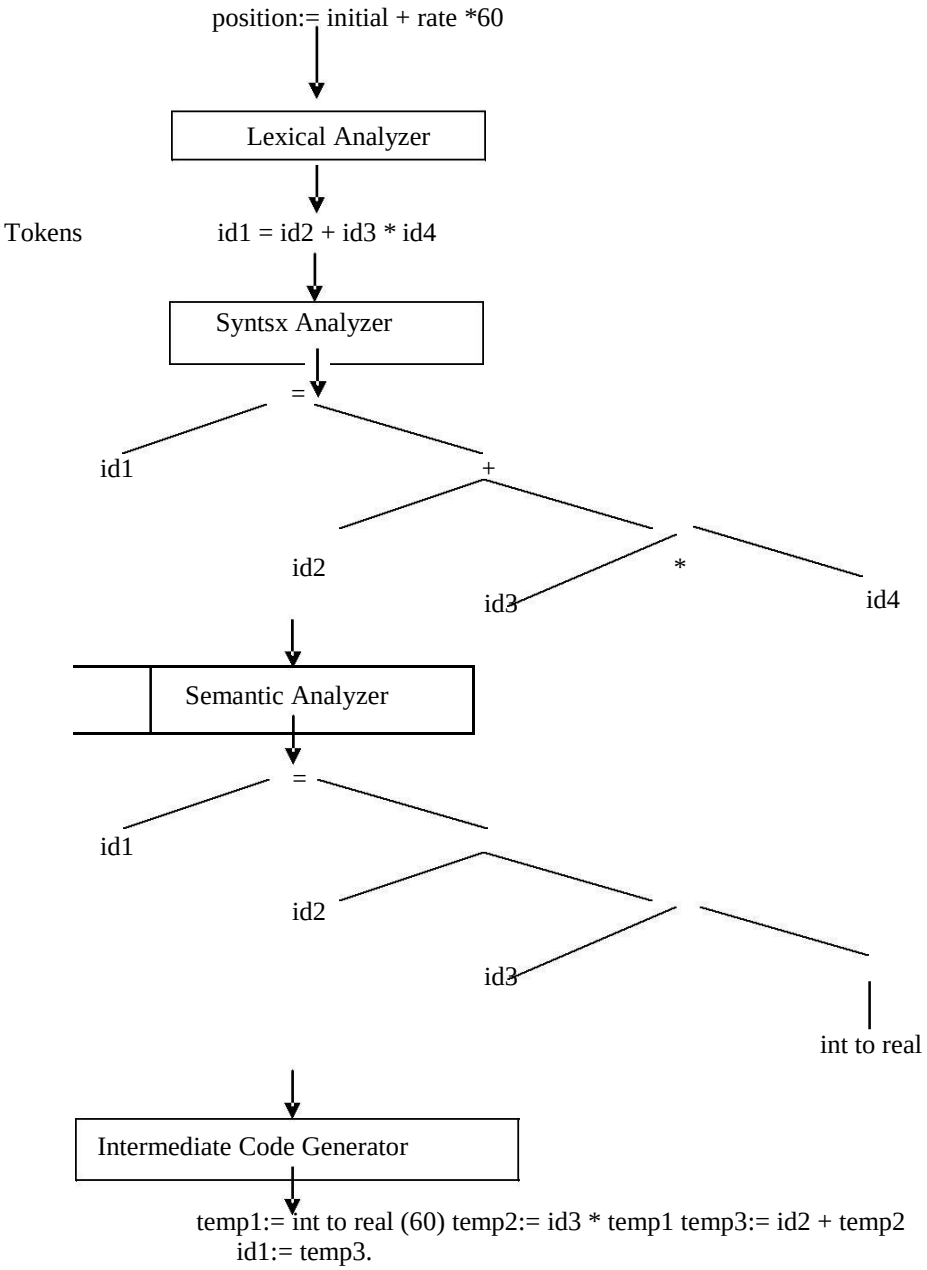
C produces the object code by deciding on the memory locations for data, selecting code to access each data and selecting the registers in which each computation is to be done. Many computers have only a few high speed registers in which computations can be performed quickly. A good code generator would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

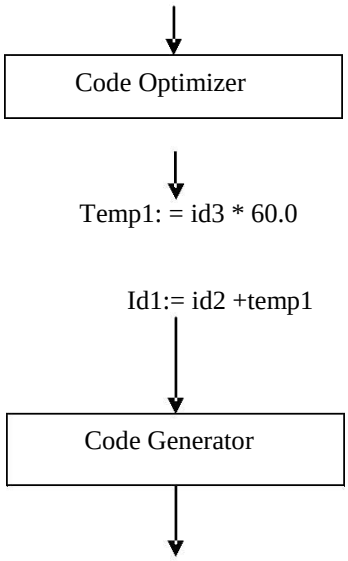
Error Handing :-

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

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

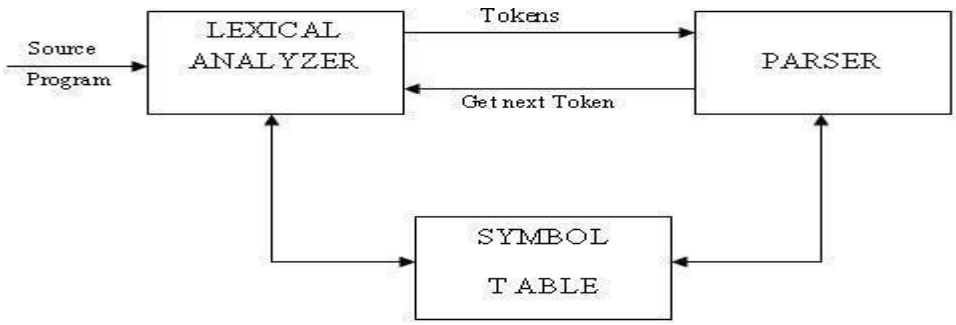
Example:





Lexical Analyzer:

The LA is the first phase of a compiler. Lexical analysis is called as linear analysis or scanning. In this phase the stream of characters making up the source program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.



Upon receiving a 'get next token' command form the parser, the lexical analyzer

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

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

Lexical Analysis Vs Parsing:

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

Token, Lexeme, Pattern:

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

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

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

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

Example:

Description of token

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

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

Lexical Errors:

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

Error-recovery actions are:

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

Difference Between Compiler And Interpreter:

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

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

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

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

5. The compiler produce object code whereas interpreter does not produce object code.

6. In the process of compilation the program is analyzed only once and then the code is generated Whereas source program is interpreted every time it is to be executed and every time the source program is analyzed. Hence interpreter is less efficient than compiler.

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

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

3. REGULAR EXPRESSIONS:

SPECIFICATION OF TOKENS

There are 3 specifications of tokens:

1) Strings
2) Language
3) Regular expression

Strings and Languages

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

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

countable set of strings over some fixed alphabet.

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

Operations on strings
The following string-related terms are commonly used:

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of strings.
   For example, ban is a prefix of banana.

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

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

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

5. A subsequence of s is any string formed by deleting zero or more not necessarily consecutive positions of s

For example, baan is a subsequence of banana.

Operations on languages:
The following are the operations that can be applied to languages:
1. Union
2. Concatenation
3. Kleene closure
4.Positive closure

The following example shows the operations on strings:

Let L={0,1} and S={a,b,c}
Union : L U S={0,1,a,b,c}
Concatenation : L.S={0a,1a,0b,1b,0c,1c}
Kleene closure : L^*={ ε,0,1,00….}
Positive closure : L^+={0,1,00….}

Regular Expressions:

Each regular expression r denotes a language L(r).
Here are the rules that define the regular expressions over some alphabet Σ and the languages that those expressions denote:

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

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

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

   - (r)|(s) is a regular expression denoting the language L(r) U L(s). ○ (r)(s) is a regular expression denoting the language L(r)L(s).
   - (r)* is a regular expression denoting (L(r))*.
   - (r) is a regular expression denoting L(r).

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

5. Concatenation has second highest precedence and is left associative. has lowest precedence and is left associative.

REGULAR DEFINITIONS:

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

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

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?)) Pascal identifier
Letter - A | B | ……| Z | a | b |……| z| Digits - 0 | 1 | 2 | …. | 9
Id - letter (letter / digit)*

Shorthand's

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

1. One or more instances (+):

- The unary postfix operator + means " one or more instances of" .
- If r is a regular expression that denotes the language L(r), then $(r)^+$ is a regular expression that denotes the language $(L (r))^+$
- Thus the regular expression a^+ denotes the set of all strings of one or more a's.
- The operator $^+$ has the same precedence and associativity as the operator *.

2. Zero or one instance (?):

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

- The notation r? is a shorthand for r | ε.

- If 'r' is a regular expression, then (r)? is a regular expression that denotes the language L(r) U { ε }.

3. Character Classes:

- The notation [abc] where a, b and c are alphabet symbols denotes the regular expression a | b | c.
- Character class such as [a – z] denotes the regular expression a | b | c | d | ….|z.
- We can describe identifiers as being strings generated by the regular expression, [A–Za–z][A–Za–z0–9]*

Non-regular Set

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

RECOGNITION OF TOKENS:

Consider the following grammar fragment: stmt → if expr then stmt
|if expr then stmt else stmt |ε

expr → term relop term |term term → id |num

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

If → if
then → then
else → else
relop → <|<=|=|<>|>|>=
id → letter(letter|digit)*

num → digit+ (.digit+)?(E(+|-)?digit+)?

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

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

TRANSITION DIAGRAM:

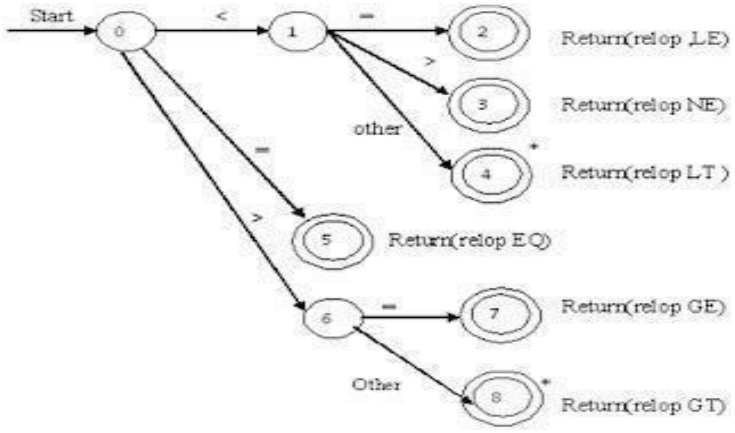
Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

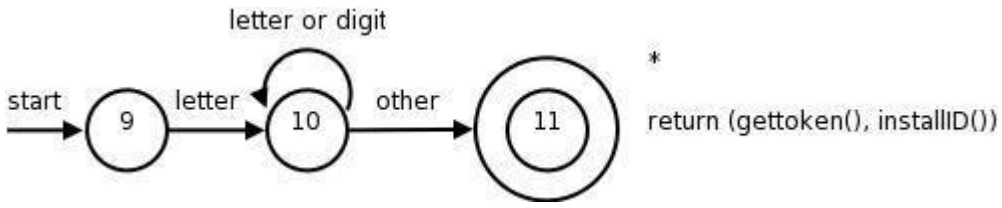
1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.

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

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



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



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

Automata:

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

1. An automation in which the output depends only on the input is **called automation without memory.**
2. An automation in which the output depends on the input and state also is **called as automation with memory**.
3. An automation in which the output depends only on the state of the machine is **called a Moore machine**.
4. An automation in which the output depends on the state and input at any instant of time is **called a mealy machine**.

DESCRIPTION OF AUTOMATA

1. An automata has a mechanism to read input from input tape,
2. Any language is recognized by some automation, Hence these automation are basically language 'acceptors' or 'language recognizers'.

Types of Finite Automata

- Deterministic Automata
- Non-Deterministic Automata.

Deterministic Automata:

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

1. it has no transitions on input € ,
2. Each input symbol has at most one transition from any state.

DFA formally defined by 5 tuple notation M = (Q, ∑, δ, qo, F), where Q is a finite 'set of states', which is non empty.

∑ is 'input alphabets', indicates input set.

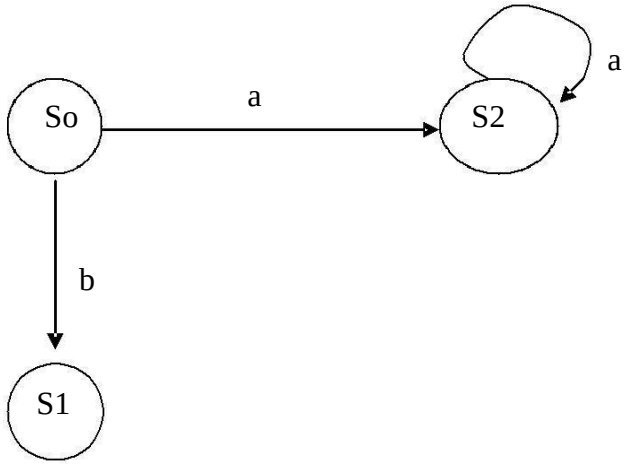
qo is an 'initial state' and qo is in Q ie, qo, ∑, Q F is a set of 'Final states',

δ is a 'transmission function' or mapping function, using this function the next state can be determined.

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

Regular expression → NFA → DFA → Minimized DFA

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



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

Nondeterministic Automata:

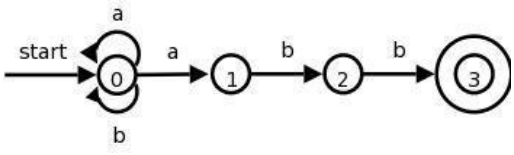
A NFA ia A mathematical model consists of

- A set of states S.
- A set of input symbols ∑.
- A transition is a move from one state to another.
- A state so that is distinguished as the start (or initial) state
- A set of states F distinguished as accepting (or final)state

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

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

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



5. Bootstrapping:

When a computer is first turned on or restarted, a special type of absolute loader, called as bootstrap loader is executed. This bootstrap loads the first program to be run by the computer usually an operating system. The bootstrap itself begins at address O in the memory of the machine. It loads the operating system (or some other program) starting at address 80. After all of the object code from device has been loaded, the bootstrap program jumps to address 80, which begins the execution of the program that was loaded.

Such loaders can be used to run stand-alone programs independent of the operating system or the system loader. They can also be used to load the operating system or the loader itself into memory.

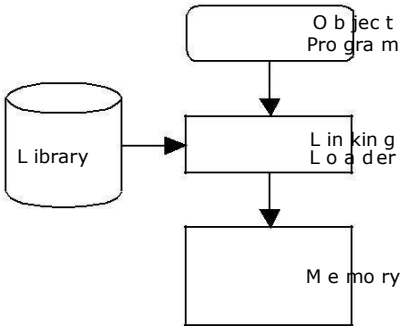
Loaders are of two types:

- Linking loader.
- Linkage editor.

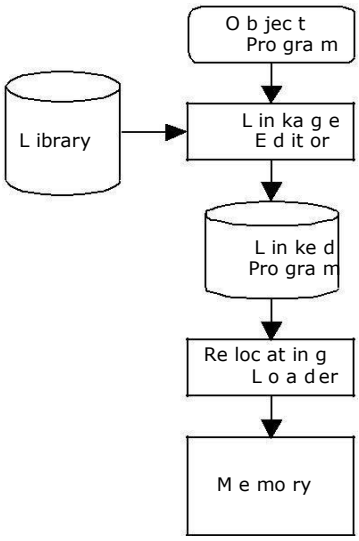
Linkage loaders, perform all linking and relocation at load time.

Linkage editors, perform linking prior to load time and dynamic linking, in which the linking function is performed at execution time.

A linkage editor performs linking and some relocation; however, the linkaged program is written to a file or library instead of being immediately loaded into memory. This approach reduces the overhead when the program is executed. All that is required at load time is a very simple form of relocation.



Linking Loader



Linkage Editor

PASS AND PHASES OF TRANSLATION:

Phases: (Phases are collected into a front end and back end)

Frontend:

The front end consists of those phases, or parts of phase, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code.

A certain amount of code optimization can be done by front end as well. the front end also includes the error handling tha goes along with each of these phases.

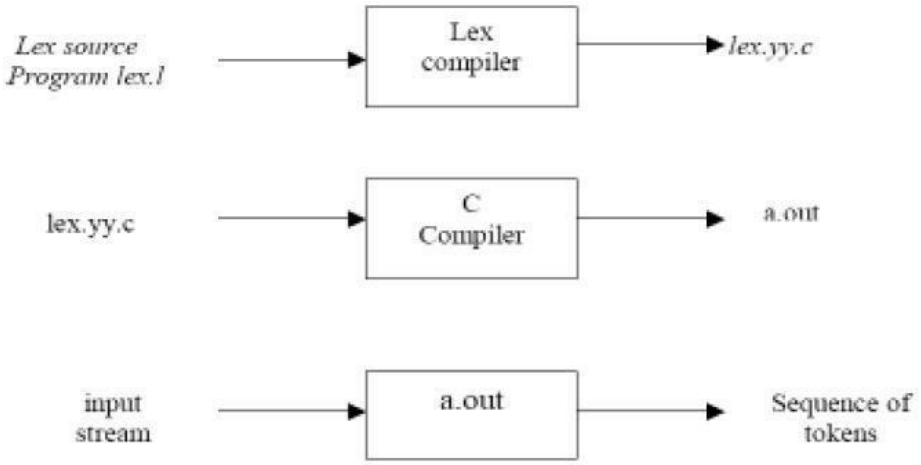
Back end:

The back end includes those portions of the compiler that depend on the target machine and generally, these portions do not depend on the source language .

6. Lexical Analyzer Generator:

7.1Creating a lexical analyzer with Lex:

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



Lex Specification

A Lex program consists of three parts:

{ definitions }
%%
{ rules }
%%
{ user subroutines }

- **Definitions** include declarations of variables, constants, and regular definitions
- **Rules** are statements of the form p1 {action1}p2 {action2} … pn {action}
- where pi is regular expression and actioni describes what action the lexical analyzer should take when pattern pi matches a lexeme. Actions are written in C code.
- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

7. INPUT BUFFERING

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

Buffering techniques:

1. Buffer pairs
2. Sentinels

The lexical analyzer scans the characters of the source program one a t a time to discover tokens.

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

Token beginnings look ahead pointer, The distance which the look ahead pointer may have to travel past the actual token may be large.

For example, in a PL/I program we may see: DECALRE (ARG1, ARG2… ARG *n*) without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis.

UNIT – II

TOPDOWN PARSING

1. Context-free Grammars: Definition:

Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where:

1. V is a finite set of variables (or nonterminals). These describe sets of "related" strings.
2. T is a finite set of terminals (i.e., tokens).
3. P is a finite set of productions, each of the form

$A \rightarrow \alpha$

where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals. $S \in V$ is the start symbol.

Example of CFG:

E ==>EAE | (E) | -E | id A==> + | - | * | / |

Where E, A are the non-terminals while id, +, *, -, /,(,) are the terminals. 2. Syntax analysis:

In syntax analysis phase the source program is analyzed to check whether if conforms to the source language's syntax, and to determine its phase structure. This phase is often separated into two phases:

- Lexical analysis: which produces a stream of tokens?
- Parser: which determines the phrase structure of the program based on the context-free grammar for the language?

PARSING:

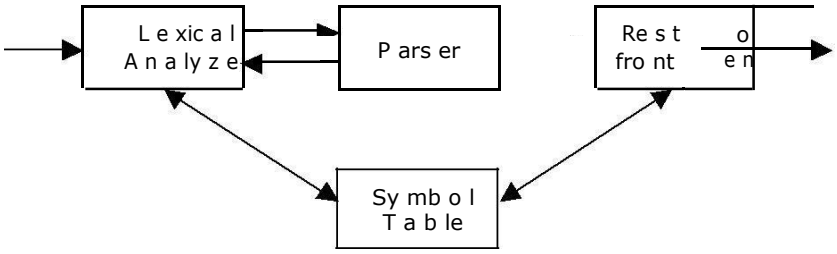
Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they build the parse trees:

- Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
- Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

Parse Tree:



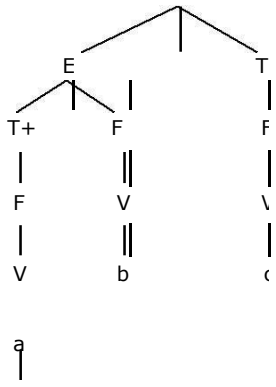
A parse tree is the graphical representation of the structure of a sentence according to its grammar.

Example:
Let the production P is:

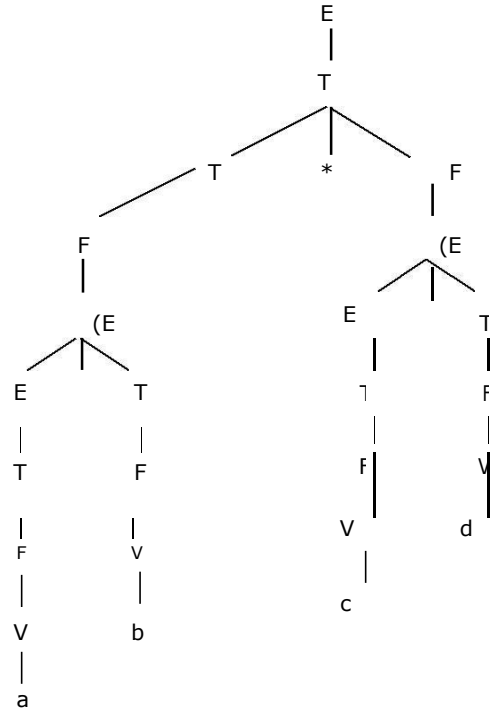
$E \rightarrow T \mid E+T$
$T \rightarrow F \mid T*F$
$F \rightarrow V \mid (E)$
$V \rightarrow a \mid b \mid c \mid d$

The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.

Parse tree for a * b + c



Parse tree for (a * b) * (c + d)

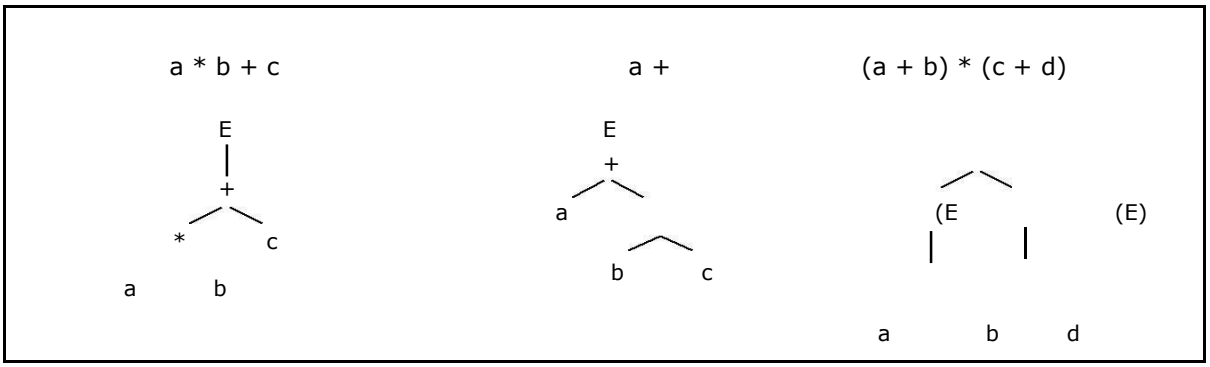


SYNTAX TREES:

Parse tree can be presented in a simplified form with only the relevant structure information by:

- Leaving out chains of derivations (whose sole purpose is to give operators difference precedence).
- Labeling the nodes with the operators in question rather than a non-terminal.

The simplified Parse tree is sometimes called as structural tree or syntax tree.



Synt a x T re e s

Syntax Error Handling:

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.The programs contain errors at many different levels.
For example, errors can be:

1) Lexical – such as misspelling an identifier, keyword or operator
2) Syntactic – such as an arithmetic expression with un-balanced parentheses.
3) Semantic – such as an operator applied to an incompatible operand.
4) Logical – such as an infinitely recursive call.

Much of error detection and recovery in a compiler is centered around the syntax analysis phase. The goals of error handler in a parser are:

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

Ambiguity:

Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parses for an ambiguous grammar has 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.

Fixing some simple ambiguities in a grammar:

	Ambiguous	language	unambiguous
(i)	$A \rightarrow B \mid AA$	Lists of one or more B's $C \rightarrow A \mid E$	$A \rightarrow BC$
(ii)	$A \rightarrow B \mid A;A$	Lists of one or more B's with punctuation $C \rightarrow ;A \mid E$	$A \rightarrow BC$
(iii)	$A \rightarrow B \mid AA \mid E$	lists of zero or more B's	$A \rightarrow BA \mid E$

Any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.

Left Recursion:

If there is any non terminal A, such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α, then

grammar is left recursive.
Algorithm for eliminating left Recursion:

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 is any string of terminals and non terminals that does not begin with A.

1. 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 non terminal.

Top down parsers cannot handle left recursive grammars.

If our expression grammar is left recursive:

- This can lead to non termination in a top-down parser.
- for a top-down parser, any recursion must be right recursion.
- we would like to convert the left recursion to right recursion.

Example 1:
Remove the left recursion from the production: $A \rightarrow A\ \alpha \mid \beta$

Left Recursive.
Eliminate

Applying the transformation yields:

$A \rightarrow \beta\ A^I$
$A^I \rightarrow \alpha\ A^I \mid \in$
↑
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:

$E \rightarrow T\ E^I$ $\qquad$ $T \rightarrow F\ T^I$

$E^I \rightarrow T\ E^I \mid \in$ $\qquad$ $T^I \rightarrow * F\ T^I \mid \in$

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:

$E \rightarrow T\ E^I$ $\qquad$ $T \rightarrow F\ T^I$

$E \rightarrow + T\ E^I \mid - T\ E^I \mid \in$ $\qquad$ $T^I \rightarrow * F\ T^I \mid /F\ T^I \mid \in$

1. The non terminal S is left recursive because $S \rightarrow A\ a \rightarrow S\ d\ a$ But it is not immediate left recursive.
2. Substitute S-productions in $A \rightarrow S\ d$ to obtain:
   $A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \in$
3. Eliminating the immediate left recursion:

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 of A.

If $\alpha \neq \in$ then replace all of the A productions, $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid - - - \mid \alpha\ \beta_n \mid r$
With

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

$A^I \rightarrow \beta_1 \mid \beta_2 \mid - - - \mid \beta_n \mid \in$

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:

$V \rightarrow \alpha\ \beta \mid \alpha\ r$ Change to:

$V \rightarrow \alpha\ V^I$ $V^I \rightarrow \beta \mid r$

Example 1:

Eliminate Left factoring in the grammar: $S \rightarrow V := int$

$V \rightarrow$ alpha '[' int ']' | alpha

Becomes:

$S \rightarrow V := int$

$V \rightarrow alpha\ V^I$

$V^I \rightarrow$ '[' int '] | $\in$

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. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.
For example, let us consider the grammar to see how top-down parser works:

S → if E then S else S | while E do S | print
E → true | False | id

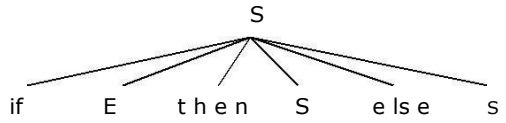
The input token string is: If id then while true do print else print.

1. Tree:

S

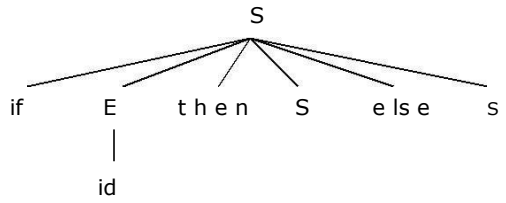
Input: if id then while true do print else print.
Action: Guess for S.

2. Tree:



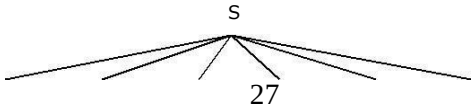
Input: if id then while true do print else print.
Action: if matches; guess for E.

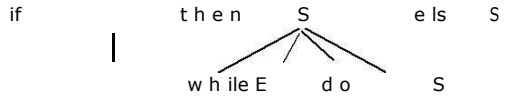
3. Tree:



Input: id then while true do print else print. Action: id matches; then matches; guess for S.

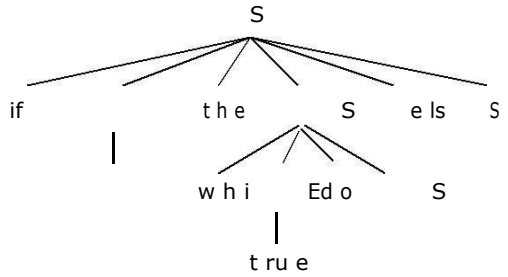
4. Tree:





Input: while true do print else print.
Action: while matches; guess for E.

5. Tree:



Input: true do print else print
Action:true matches; do matches; guess S.

Recursive Descent Parsing:

Top-down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewd as a 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 scans of the input. 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 of Fig(a).

A

c

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 ecpand A without Having consumed any input.

Predictive Parsing:

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

$A \rightarrow \alpha_1 | \alpha_2 | - - - | \alpha_n$.

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.

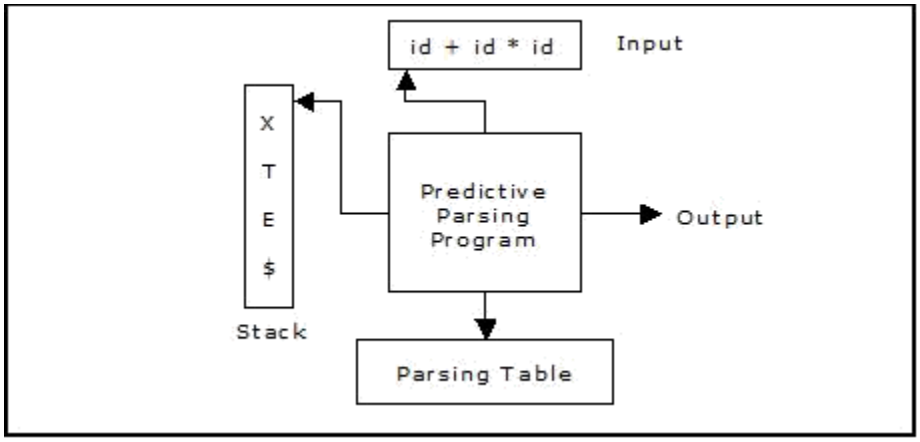
For example:

stmt $\rightarrow$ **if** exp **then** stmt **else** stmt | **while** expr **do** stmt

| **begin** stmt-list **end**.

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

The model of predictive parser is as follows:



A predictive parser has:

- Stack
- Input
- Parsing Table
- Output

The input buffer consists 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 $S on the stack with S, the start symbol of G on top, and w$ in the 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 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 → $Y_1 Y_2 . . . . . Y_k$ **then begin**

pop X from the stack;

push $Y_k, Y_{k-1}, ... Y_1$ onto the stack, with Y_1 on top; output the

end

production X $Y_1 Y_2 . . . . . Y_k$

else error()

until X = $ /*stack is empty*/

FIRST and FOLLOW:

The construction of a predictive parser is aided by two functions with a grammar G. these functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the **FOLLOW** function can also be used as synchronizing tokens during pannic-mode error recovery.

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α=>€,then € is also in FIRST(α).

Define FOLLOW (A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist 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 € can be added to any FIRST set.

- If X is terminal, then FIRST(X) is {X}.
- If X→€ is production, then add € to FIRST(X).
- If X is nonterminal and $X→Y_1 Y_2……Y_k$ is a production, then place a in FIRST(X) if for some i,a is in FIRST(Y_i),and € is in all of FIRST(Y_i),and € is in all of FIRST(Y_1),….. FIRST(Y_{i-1});that is $Y_1……. Y_{i-1}$=>€.if € is in FIRST(Y_j), for all j=,2,3…….k, then add € to FIRST(X).for example, everything in FIRST(Y_1) is surely in FIRST(X).if Y_1 does not derive €,then we add nothing more to FIRST(X),but if Y_1=>€,then we add FIRST(Y_2) and so on.

FIRST (A) = FIRST (α_1) U FIRST (α_2) U - - - U FIRST (α_n) Where, A → α_1 | α_2 | - - - |α_n, are all the productions for A. FIRST (Aα) = if $\in \notin$ FIRST (A) then FIRST (A)
else (FIRST (A) - {$\in$}) 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 in FIRST(β) except for € is placed in FOLLOW(B).
- If there is production A→αB, or a production A→αBβ where FIRST (β) contains € (i.e.,β→€),then everything in FOLLOW(A)is in FOLLOW(B).

Example:

Construct the FIRST and FOLLOW for the grammar:

$A \rightarrow BC \mid EFGH \mid H$
$B \rightarrow b$
$C \rightarrow c \mid \in$
$E \rightarrow e \mid \in$
$F \rightarrow CE$
$G \rightarrow g$
$H \rightarrow h \mid \in$

Solution:

1. Finding first () set:
   1. first (H) = first (h) ∪ first (ε) = {h, ε}
   2. first (G) = first (g) = {g}
   3. first (C) = first (c) ∪ first (ε) = c, ε}
   4. first (E) = first (e) ∪ first (ε) = {e, ε}
   5. first (F) = first (CE) = (first (c) - {ε}) ∪ first (E)
      = (c, ε} {ε}) ∪ {e, ε} = {c, e, ε}
   6. first (B) = first (b)={b}
   7. first (A) = first (BC) ∪ first (EFGH) ∪ first (H)
      = first (B) ∪ (first (E) – { ε}) ∪ first (FGH) ∪ {h, ε}
      = {b, h, ε} ∪ {e} ∪ (first (F) – {ε}) ∪ first (GH)
      = {b, e, h, ε} ∪ {C, e} ∪ first (G)
      = {b, c, e, h, ε} ∪ {g} = {b, c, e, g, h, ε}

2. Finding follow() sets:
   1. follow(A) = {$}

2. follow(B) = first(C) – {ε} ∪ follow(A) = {C, $}

3. follow(G) = first(H) – {ε} ∪ follow(A)

   ={h, ε} – {ε} ∪ {$} = {h, $}

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

5. follow(F) = first(GH) – {ε} = {g}

6. follow(E) = first(FGH) m- {ε} ∪ follow(F)

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

   = {c, e} ∪ {g} ∪ {g} = {c, e, g}

7. follow(C) = follow(A) ∪ first (E) – {ε} ∪ follow (F)

   ={$} ∪ {e, ε} ∪ {g} = {e, g, $}

Example 1:

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$ $F \rightarrow (E) \mid id$

Step 1:

Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left Recursive grammar (as it goes to infinite loop).

$E \rightarrow T\ E^I$

$E^I \rightarrow + T\ E^I \mid \in$ $T^I \rightarrow F\ T^I$

$T^I \rightarrow * F\ T^I \mid \in$ $F \rightarrow (E) \mid id$

Step 2:

Find the FIRST(X) and FOLLOW(X) for all the variables.

The variables are: {E, E^I, T, T^I, F}

Terminals are: {+, *, (,), id} and $

Computation of FIRST() sets:

FIRST (F) = FIRST ((E)) U FIRST (id) = {(, id}
FIRST (T^I) = FIRST (*FT^I) U FIRST (∈) = {*, ∈}
FIRST (T) = FIRST (FT^I) = FIRST (F) = {(, id}
FIRST (E^I) = FIRST (+TE^I) U FIRST (∈) = {+, ∈}
FIRST (E) = FIRST (TE^I) = FIRST (T) = {(, id}

Computation of FOLLOW () sets:

	Relevant production
FOLLOW (E) = {$} U FIRST ( )) = {$,)}	F → (E)

FOLLOW (E^I) = FOLLOW (E) = {$,)} $E \rightarrow TE^I$

FOLLOW (T) = (FIRST (E^I) - {$\in$}) U FOLLOW (E) U FOLLOW (E^I) $E \rightarrow TE^I$

= {+, $E^I \rightarrow +TE^I$

FOLLOW (T^I) = FOLLOW (T) = {+,), $} $T \rightarrow FT^I$

FOLLOW (F) = (FIRST (T^I) - {$\in$}) U FOLLOW (T) U FOLLOW (T^I) $T \rightarrow T^I$

= {*, +,

Step 3:
Construction of parsing table:

Termina / Variables	+		(	)	id	$
E			$E \rightarrow TE$		$E \rightarrow TE^I$	
E^I	$E^I \rightarrow +TE^I$			$E^I \rightarrow \in$		$E^I \rightarrow \in$
T			$T \rightarrow FT$		$T \rightarrow FT^I$	
T^I	$T^I \rightarrow \in$	$T^I \rightarrow *F$		$T^I \rightarrow \in$		$T^I \rightarrow \in$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Table 3.1. Parsing Table

Fill the table with the production on the basis of the FIRST(α). If the input symbol is an $\in$ in FIRST(α), then goto FOLLOW(α) and fill $\alpha \rightarrow \in$, in all those input symbols.

Let us start with the non-terminal E, FIRST(E) = {(, id}. So, place the production $E \rightarrow TE^I$ at (and id.

For the non-terminal E^I, FIRST (E^I) = {+, $\in$}.

So, place the production $E^I \rightarrow +TE^I$ at + and also as there is a $\in$ in FIRST(E^I), see FOLLOW(E^I) = {$,)}. So write the production $E^I \rightarrow \in$ at the place $ and).

Similarly:

For the non-terminal T, FIRST(T) = {(, id}. So place the production $T \rightarrow FT^I$ at (and id.

For the non-terminal T^I, FIRST (T^I) = {*, $\in$}

So place the production $T^I \rightarrow *FT^I$ at * and also as there is a $\in$ in FIRST (T^I), see FOLLOW (T^I) = {+, $,)}, so write the production $T^I \rightarrow \in$ at +, $ and).

For the non-terminal F, FIRST (F) = {(, id}.

So place the production $F \rightarrow$ id at id location and $F \rightarrow$ (E) at (as it has two productions.

Finally, make all undefined entries as error.

As these were no multiple entries in the table, hence the given grammar is LL(1).

Step 4:

Moves made by predictive parser on the input id + id * id is:

STACK	INPUT	REMARKS
$ **E**	**id** + id * id $	E and id are not identical; so see E on id in parse table, the production is $E \rightarrow TE^I$; pop E, push E^I and T i.e., move in reverse order.
$ E^I **T**	**id** + id * id $	See T on id the production is $T \rightarrow F\ T^I$; Pop T, push T^I and F; Proceed until both are identical.
$ $E^I\ T^I$ **F**	**id** + id * id $	$F \rightarrow$ id
$ $E^I\ T^I$ **id**	**id** + id * id $	Identical; pop id and remove id from input symbol.
$ E^I **T^I**	+ id *	See T^I on +; $T^I \rightarrow \in$ so, pop T^I
$ **E^I**	+ id *	See E^I on +; $E^I \rightarrow +T\ E^I$; push E^I , + and T
$ E^I T +	+ id *	Identical; pop + and remove + from input symbol.
$ E^I **T**	**id** *	
$ $E^I\ T^I$ **F**	**id** *	$T \rightarrow F\ T^I$
$ $E^I\ T^I$ **id**	**id** *	$F \rightarrow$ id
$ E^I **T^I**	*	
$ $E^I\ T^I$ F *	*	$T^I \rightarrow * F\ T^I$
$ $E^I\ T^I$ **F**		
$ $E^I\ T^I$ **id**		$F \rightarrow$ id
$ E^I **T^I**		$T^I \rightarrow \in$
$ **E^I**		$E^I \rightarrow \in$
$		Accept.

Table 3.2 Moves made by the parser on input id + id * id

Predictive parser accepts the given input string. We can notice that $ in input and stuck, i.e., both are empty, hence accepted.

LL (1) Grammar:

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.

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminal / Variables	+	*	(	)	id	$
E	error	error	$E \rightarrow TE$	synch	$E \rightarrow TE^I$	synch
E^I	$E^I \rightarrow +TE^I$	error	error	$E^I \rightarrow \in$	error	$E^I \rightarrow \in$
T	synch	error	$T \rightarrow FT$	synch	$T \rightarrow FT^I$	synch
T^I	$T^I \rightarrow \in$	$T^I \rightarrow *F$	error	$T^I \rightarrow \in$	error	$T^I \rightarrow \in$
F	synch	synch	$F \rightarrow (E)$	synch	$F \rightarrow id$	synch

Table3.3 :Synchronizing tokens added to parsing table for table 3.1.

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

STACK	IN	REMARKS
$ **E**	**)** id * +	Error, skip)
$ **E**	**id** * +	
$ E^I **T**	**id** * +	
$ E^I T^I **F**	**id** * +	
$ E^I T^I **id**	**id** * +	
$ E^I $\mathbf{T^I}$	***** +	
$ E^I T^I F *	***** +	

$ E^{I} T^{I} **F**	+	Error; F on + is synch; F has been popped.
$ E^{I} **T**^{I}	+	
$ **E**^{I}	+	
$ E^{I} T +	+	
$ E^{I} **T**		
$ E^{I} T^{I} **F**		
$ E^{I} T^{I} **id**		
$ E^{I} **T**^{I}		
$ **E**^{I}		
$		Accept.

Example 2:

Table 3.4. Parsing and error recovery moves made by predictive parser

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$S \rightarrow iEtSS^{I} \mid a$
$S^{I} \rightarrow eS \mid \in$
$E \rightarrow b$

Solution:

1. Computation of First () set:

   1. First (E) = first (b) = {b}
   2. First (S^{I}) = first (eS) $\cup$ first (ε) = {e, ε}
   3. first (S) = first ($iEtSS^{I}$) $\cup$ first (a) = {i, a}

2. Computation of follow() set:

   1. follow (S) = {$} $\cup$ first (S^{I}) – {ε} $\cup$ follow (S) $\cup$ follow (S^{I})
      = {$} $\cup$ {e} = {e, $}
   2. follow (S^{I}) = follow (S) = {e, $}
   3. follow (E) = first (tSS^{I}) = {t}

3. The parsing table for this grammar is:

	a	b	e	i	t	
S	$S \rightarrow a$			$S \rightarrow iEtSS^{I}$		
S^{I}			$S^{I} \rightarrow eS$ $S^{I} \rightarrow \in$			$S^{I} \rightarrow \in$
E		$E \rightarrow b$				

∈

As the table multiply defined entry. The given grammar is not LL(1).

Example 3:

Construct the FIRST and FOLLOW and predictive parse table for the grammar:

$S \rightarrow AC\$$
$C \rightarrow c \mid \in$
$A \rightarrow aBCd \mid BQ \mid \in$
$B \rightarrow bB \mid d$
$Q \rightarrow q$

Solution:

1. Finding the first () sets: First (Q) = {q}

   First (B) = {b, d}

   First (C) = {c, ε}

   First (A) = First (aBCd) ∪ First (BQ) ∪ First (ε)

   = {a} ∪ First (B) ∪ First (d) ∪{ε}

   = {a} ∪ First (bB) ∪ First (d) ∪ {ε}

   = {a} ∪ {b} ∪ {d} ∪ {ε}

   = {a, b, d, ε} First (S) = First (AC$)

   = (First (A) – {ε}) ∪ (First (C) – {ε}) ∪ First (ε)

   = ({a, b, d, ε} – {ε}) ∪ ({c, ε} – {ε}) ∪ {ε}

   = {a, b, d, c, ε}

2. Finding Follow () sets: Follow (S) = {#}

   Follow (A) = (First (C) – {ε}) ∪ First ($) = ({c, ε} – {ε}) ∪ {$} Follow (A) = {c, $}

   Follow (B) = (First (C) – {ε}) ∪ First (d) ∪ First (Q)

   = {c} ∪ {d} ∪ {q} = {c, d, q} Follow (C) = (First ($) ∪ First (d) = {d, $}

Follow (Q) = (First (A) = {c, $}

3. The parsing table for this grammar is:

	a	b	c	D	q	$	
S	S → AC$	S → AC $	S → AC $	S → AC $		S → AC $	
A	A → aBCd	A → BQ	A → ε	A → BQ		A → ε	
B		B → bB		B → d			
C			C → c	C → ε		C → ε	
Q					Q → q		

4. Moves made by predictive parser on the input abdcdc$ is:

Stack symbol	Input	Remarks
#S	abdcdc$#	S → AC$
#$CA	abdcdc$#	A → aBCd
#$CdCBa	abdcdc$#	Pop a
#$CdCB	bdcdc$#	B → bB
#$CdCBb	bdcdc$#	Pop b
#$CdCB	dcdc$#	B → d
#$CdCd	dcdc$#	Pop d
#$CdC	cdc$#	C → c
#$Cdc	cdc$#	Pop C
#$Cd	dc$#	Pop d
#$C	c$#	C → c
#$c	c$#	Pop c
#$	$#	Pop $
#	#	Accepted

BOTTOM UP PARSING

1. BOTTOM UP PARSING:

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow - - - \Rightarrow r_{n-1} \Rightarrow r_n \Rightarrow$ sentence

$\longleftarrow$

Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce r_i r_{i-1} match some RHS β against r_i then replace β with its corresponding LHS, A.

In terms of the parse tree, this is working from leaves to root.

Example – 1:

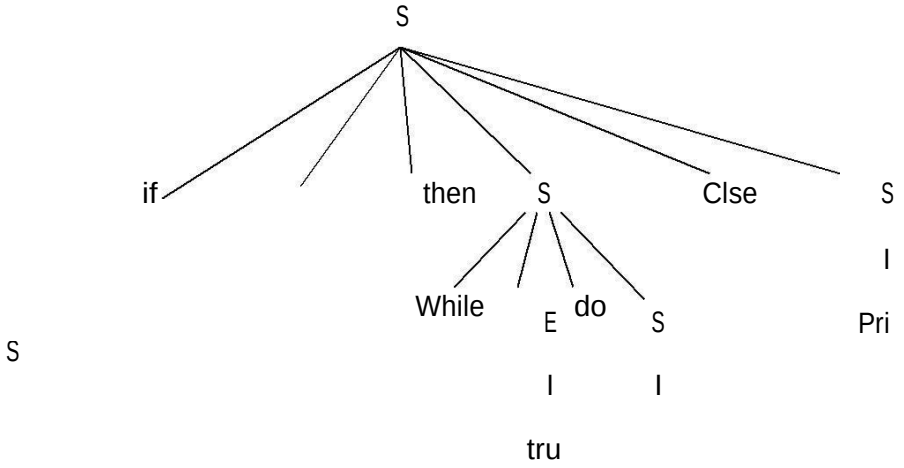
S→if E then S else S/while E do S/ print

E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basic idea: Given input string a, "reduce" it to the goal (start) symbol, by looking for substring that match production RHS.



$\Rightarrow_{lm}$ if E then S else S

$\Rightarrow_{lm}$ if id then S else S

$\Rightarrow_{lm}$ if id then while E do S else S

$\Rightarrow_{lm}$ if id then while true do S else S

$\Rightarrow_{lm}$ if id then while true do print else S

$\Rightarrow_{lm}$ if id then while true do print else print

$\Leftarrow_{rm}$ if E then while true do print else print

$\Leftarrow_{rm}$ if E then while E do print else print

$\Leftarrow_{rm}$ if E then while E do S else print

$\Leftarrow_{rm}$ if E then S else print

$\Leftarrow_{rm}$ if E then S else S

$\Leftarrow_{rm}$ S

Topdown Vs Bottom-up parsing:

Top-down	Bottom-up
1. Construct tree from root to leaves 2. "Guers" which RHS to substitute for nonterminal 3. Produces left-most derivation 4. Recursive descent, LL parsers 5. Recursive descent, LL parsers 6. Easy for humans	1. Construct tree from leaves to root 2. "Guers" which rule to "reduce" terminals 3. Produces reverse right-most derivation. 4. Shift-reduce, LR, LALR, etc. 5. "Harder" for humans.

→ Bottom-up can parse a larger set of languages than topdown.

→ Both work for most (but not all) features of most computer languages.

Example – 2:

S→aAcBe

A→Ab/b

B→d

llp: abbcde/

Right-most derivation

S→aAcBe

→ aAcde

→aAbcde

→ abbcde

Bottom-up approach

"Right sentential form"	Reduction
abbcde	
aAbcde	A→b
Aacde	A→Ab
AacBe	B→d
S	S→aAcBe

Steps correspond to a right-most derivation in reverse.

(must choose RHS wisely)

Example – 3:

S→aABe

A→Abc/b

B→d

1/p: abbcde

Right most derivation:

aABe

aAde Since ( ) B→d

aAbcde Since ( ) A→Abc

abbcde Since ( ) A→b

Parsing using Bottom-up approach:

Input	Production used
abbcde	
aAbcde	A→b
AAde	A→Abc
AABe	B→d

S parsing is completed as we got a start symbol

Hence the 1/p string is acceptable.

Example – 4

$E \rightarrow E+E$

$E \rightarrow E*E$

$E \rightarrow (E)$

$E \rightarrow id$

1/p: $id_1+id_2+id_3$

Right most derivation

$E \rightarrow E+E$

$\rightarrow E+E*E$

$\rightarrow E+E*id_3 \rightarrow E+id_2*id_3$

$\rightarrow id_1+id_2*id_3$

Parsing using Bottom-up approach:

Go from left to right

$id_1+id_2*id_3$	
$E+id_2*id_3$	$E \rightarrow id$
$E+E*id_3$	$E \rightarrow id$
$E*id_3$	$E \rightarrow E+E$
$E*E$	$E \rightarrow id$
E	

= start symbol, Hence acceptable.

2. HANDLES:

Always making progress by replacing a substring with LHS of a matching production will not lead to the goal/start symbol.

For example:

abbcde

aAbcde $A \rightarrow b$

aAAcde $A \rightarrow b$

struck

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

If the grammar is unambiguous, every right sentential form has exactly one handle.

More formally, A handle is a production $A \rightarrow \beta$ and a position in the current right-sentential form $\alpha\beta\omega$ such that:

$S \Rightarrow \alpha A\omega \Rightarrow \alpha/\beta\omega$

For example grammar, if current right-sentential form is

a/Abcde

Then the handle is $A \rightarrow Ab$ at the marked position. 'a' never contains non-terminals.

HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

$E \rightarrow E+E/E*E/(E)/id$

Right-sentential form	Handle	Reducing production
a+b*c	a	$E \rightarrow id$
E+b*c	b	$E \rightarrow id$

E+E*C	C	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us.

3. SHIFT- REDUCE PARSING:

Shift Reduce Parsing uses a stuck to hold grammar symbols and input buffer to hold string to be parsed, because handles always appear at the top of the stack i.e., there's no need to look deeper into the state.

A shift-reduce parser has just four actions:

1. Shift-next word is shifted onto the stack (input symbols) until a handle is formed.
2. Reduce – right end of handle is at top of stack, locate left end of handle within the stack. Pop handle off stack and push appropriate LHS.
3. Accept – stop parsing on successful completion of parse and report success.
4. Error – call an error reporting/recovery routine.

Possible Conflicts:

Ambiguous grammars lead to parsing conflicts.

1. **Shift-reduce:** Both a shift action and a reduce action are possible in the same state (should we shift or reduce)

   Example: dangling-else problem

2. **Reduce-reduce:** Two or more distinct reduce actions are possible in the same state. (Which production should we reduce with 2).

Example:

Stmt →id (param) (a(i) is procedure call)

Param→ id

Expr → id (expr) /id (a(i) is array subscript)

Stack	input buffer	action
$…aa (i) ….$	Reduce by ?	

Should we reduce to param or to expr? Need to know the type of a: is it an array or a function. This information must flow from declaration of a to this use, typically via a symbol table.

Shift – reduce parsing example: (Stack implementation)

Grammar: E→E+E/E*E/(E)/id Input: $id_1+id_2+id_3$

One Scheme to implement a handle-pruning, bottom-up parser is called a shift-reduce parser. Shift reduce parsers use stack and an input buffer.

The sequence of steps is as follows:

1. initialize stack with $.
2. Repeat until the top of the stack is the goal symbol and the input token is "end of life". **a. Find the handle**

If we don't have a handle on top of stack, shift an input symbol onto the stack**.**

b. Prune the handle

if we have a handle (A→β) on the stack, reduce

(i) pop /β/ symbols off the stack (ii)push A onto the stack.

Stack	input	Action
$	$id_1+id_2*id_3$$	Shift
$ id_1	$+id_2*id_3$$	Reduce by E→id
$E	$+id_2*id_3$$	Shift
$E+	$id_2*id_3$$	Shift
$E+ id_2	$*id_3$$	Reduce by E→id

$E+E	*id3$	Shift
$E+E*	id3$	Shift
$E+E* id3	$	Reduce by E→id
$E+E*E	$	Reduce by E→E*E
$E+E	$	Reduce by E→E+E
$E	$	Accept

Example 2:

Goal → Expr

Expr → Expr + term | Expr – Term | Term

Term → Tem & Factor | Term | factor | Factor

Factor → number | id | (Expr)

The expression grammar : x – z * y

Stack	Input	Action
$	Id - num * id	Shift
$ id	- num * id	Reduce factor → id
$ Factor	- num * id	Reduce Term → Factor
$ Term	- num * id	Reduce Expr → Term
$ Expr	- num * id	Shift
$ Expr -	num * id	Shift
$ Expr – num	* id	Reduce Factor → num
$ Expr – Factor	* id	Reduce Term → Factor
$ Expr – Term	* id	Shift
$ Expr – Term *	id	Shift

$ Expr – Term * id		Reduce Factor → id
$ Expr – Term & Factor		Reduce Term → Term * Factor
$ Expr – Term		Reduce Expr → Expr – Term
$ Expr		Reduce Goal → Expr
$ Goal		Accept

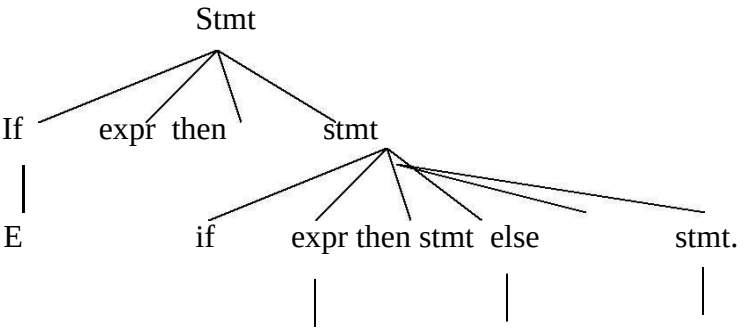
1. shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce.

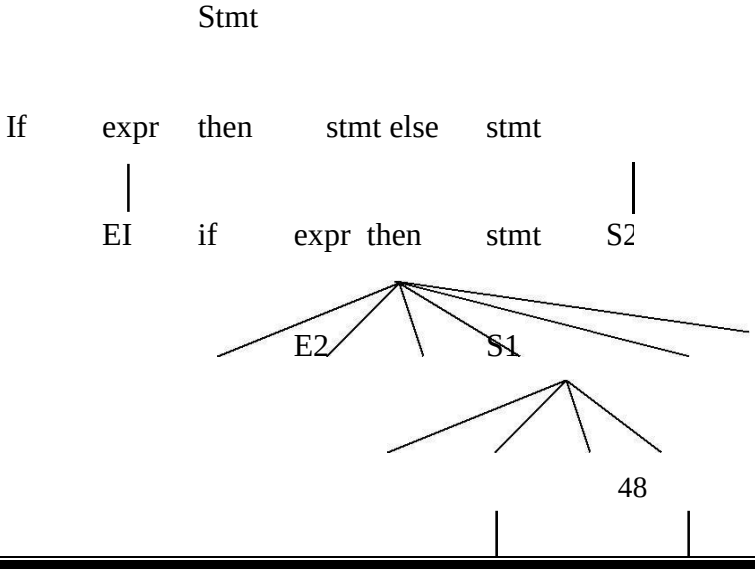
Procedure:

1. Shift until top of stack is the right end of a handle.
2. Find the left end of the handle and reduce.

* Dangling-else problem:

stmt→if expr then stmt/if expr then stmt/other then example string is: if E_1 then if E_2 then S_1 else S_2 has two parse trees (ambiguity) and so this grammar is not of LR(k) type.





3. OPERATOR – PRECEDENCE PARSING:

Precedence/ Operator grammar: The grammars having the property:

1. **No production right side is should contain ∈.**
2. **No production sight side should contain two adjacent non-terminals.**

Is called an **operator grammar.**

Operator – precedence parsing has three disjoint precedence relations, <.,=and .> between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
a<.b	'a' yields precedence to 'b'.
a=b	'a' has the same precedence 'b'
a.>b	'a' takes precedence over 'b'.

Operator precedence parsing has a number of disadvantages:

1. It is hard to handle tokens like the minus sign, which has two different precedences.
2. Only a small class of grammars can be parsed.
3. The relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.

Disadvantages:

1. **L(G) ≠L(parser)**
2. **error detection**
3. **usage is limited**
4. **They are easy to analyse manually Example:**

Grammar: E→EAE|(E)|-E/id

A→+|-|*|/|↑

Input string: id+id*id

The operator – precedence relations are:

	Id	+	*	$
Id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Solution: This is not operator grammar, so first reduce it to operator grammar form, by eliminating adjacent non-terminals.

Operator grammar is:

$E \rightarrow E+E|E-E|E*E|E/E|E\uparrow E|(E)|-E|id$

The input string with precedence relations interested is:

$<.id.> + <.id.> * <.id.> $

Scan the string the from left end until first .> is encounted.

$<.id.>+<.id.>*<.id.<$

This occurs between the first id and +.

Scan backwards (to the left) over any =’s until a <. Is encounted. We scan backwards to $.

$<.id.>+<.id.>*<.id.>$

↑ ↑

Everything to the left of the first .> and to the right of <. Is called handle. Here, the handle is the first id.

Then reduce id to E. At this point we have: E+id*id

By repeating the process and proceding in the same way: $+<.id.>*<.id.>$

substitute $E \rightarrow id$,

After reducing the other id to E by the same process, we obtain the right-sentential form

E+E*E

Now, the 1/p string afte detecting the non-terminals sis:

$\Rightarrow$ $+*$

Inserting the precedence relations, we get: $<.+<.*.>$

↑ ↑

The left end of the handle lies between + and * and the right end between * and $. It indicates that, in the right sentential form E+E*E, the handle is E*E.

Reducing by E→E*E, we get:

E+E

Now the input string is: $<.+$

Again inserting the precedence relations, we get:

⇒$<.+.>$

↑ ↑

reducing by E→E+E, we get,

$ $

and finally we are left with:

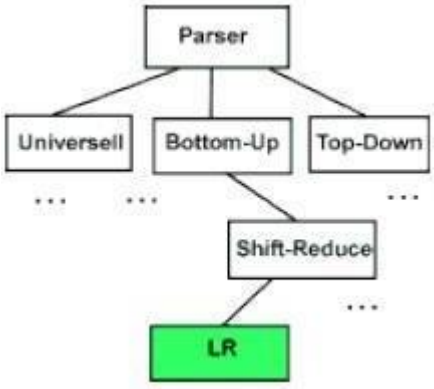
E

Hence accepted.

Input string	Precedence relations inserted	Action
id+id*id	$<.id.>+<.id.>*<.id.>$	
E+id*id	$+<.id.>*<.id.>$	E→id
E+E*id	$+*<.id.>$	E→id
E+E*E	$+*$	
E+E*E	$<.+<.*.>$	E→E*E
E+E	$<.+$	
E+E	$<.+.>$	E→E+E
E	$$	Accepted

5. LR PARSING INTRODUCTION:

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



WHY LR PARSING:

1. LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.

2. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
3. 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.
4. 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 constuct 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.

LR PARSERS:

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

'L' stands for "Left-to-right" scan of input.

'R' stands for "Rightmost derivation (in reverse)".

'K' stands for number of input symbols of look-a-head that are used in making parsing decisions.

When (K) is omitted, 'K' is assumed to be 1.

LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar. A grammar is LR(1) if, given a right-most derivation

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 - - - r_{n-1} \Rightarrow r_n \Rightarrow$ sentence.

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol are left.

LR(0) item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex– 1: $S^I \rightarrow .S\$$ $S \rightarrow .x$ $S \rightarrow .(L)$

Ex-2: $A \rightarrow XYZ$ generates 4LR(0) items –

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

The '.' Indicates how much of an item we have seen at a given state in the parse.

$A \rightarrow .XYZ$ indicates that the parser is looking for a string that can be derived from XYZ.

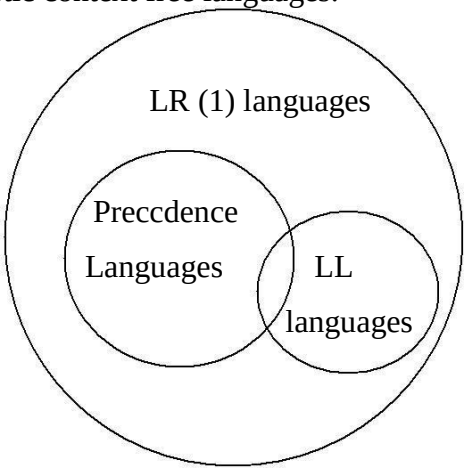
A→XY.Z indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

→ LR(0) items play a key role in the SLR(1) table construction algorithm.

→ LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms. LR parsers have more information available than LL parsers when choosing a production:

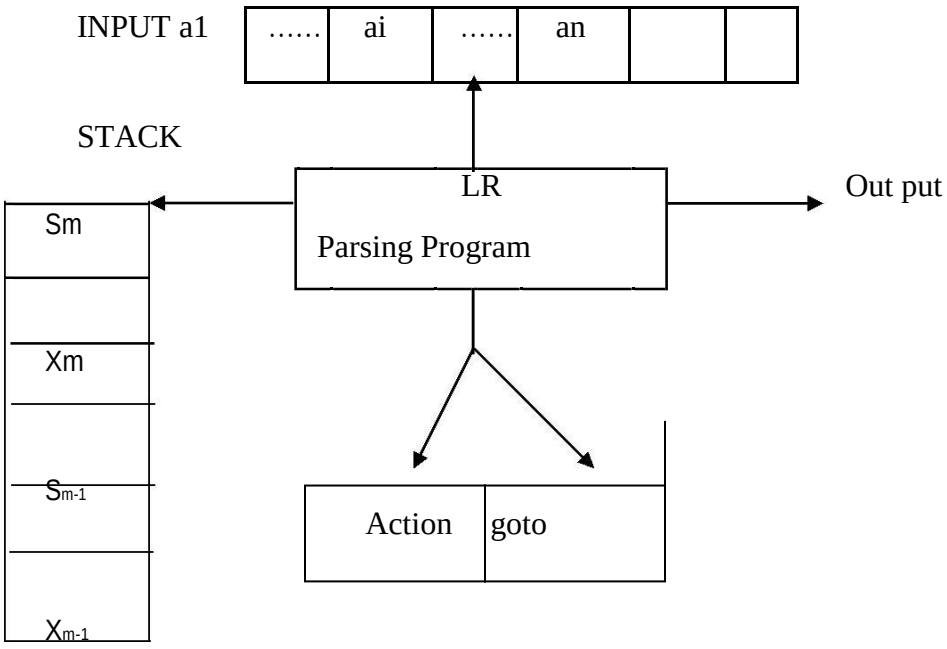
* **LR knows everything derived from RHS plus 'K' lookahead symbols.**
* **LL just knows 'K' lookahead symbols into what's derived from RHS.**

Deterministic context free languages:



LR PARSING ALGORITHM:

The schematic form of an LR parser is shown below:



It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts: action and goto.

The LR parser program determines Sm, the current state on the top of the stack, and a_i, the current input symbol. It then consults action [Sm, a_i], which can have one of four values:

1. **Shift S, where S is a state.**
2. **reduce by a grammar production A→β**
3. **accept and**
4. **error**

The function goes to takes a state and grammar symbol as arguments and produces a state. The goto function of a parsing table constructed from a grammar G using the SLR, canonical LR or LALR method is the transition function of DFA that recognizes the viable prefixes of G. (Viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the right-most handle)

5.6 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 $S^I \rightarrow S$.

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 $S^I \rightarrow S$.

CONSTRUCTION OF SLR PARSING TABLE:

Example:

The given grammar is:

1. **E→E+T**
2. **E→ T**
3. **T →T*F**
4. **T→F**
5. **F→(E)**
6. **F→id Step I: The Augmented grammar is:**

$E^I \rightarrow E$

$E \rightarrow E+T$

$E \rightarrow T$

$T \rightarrow T*F$

$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

Step II: The collection of LR (0) items are:

I_0: $E^I \rightarrow .E$

$E \rightarrow .E+T$

$E \rightarrow .T$

$T \rightarrow .T*F$

$T \rightarrow .F$

$F \rightarrow .(E)$

$F \rightarrow .id$

Start with start symbol after since ( ) there is E, start writing all productions of E.

Start writing 'T' productions

Start writing F productions

Goto (I_0,E): States have successor states formed by advancing the marker over the symbol it preceeds. For state 1 there are successor states reached by advancing the masks over the

$E^I \rightarrow E.$ - symbols E,T,F,C or id. Consider, first, the

$E \rightarrow E.+T$

Goto (I_0,T):

I_2: $E \rightarrow T.$ - reduced Item (RI)

T→T.*F

Goto (I_0,F):

I_2: E→T. - reduced item (RI)

T→T.*F

Goto (I_0,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

If '.' Precedes non-terminal start writing its corresponding production. Here first E then T after that F.

Start writing F productions.

Goto (I_0,id):

I_5: F →id. - reduced item.

E successor (I, state), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non-terminal). The state I_2 is thus:

Goto (I_1,+):

I_6: E→E+.T start writing T productions

T→.T*F

T→.F start writing F productions

F→.(E)

F→.id

Goto (I_2,*):

I_7: T→T*.F start writing F productions

F→.(E)

F→.id

Goto (I_4,E):

I_8: F→(E.)

E→E.+T

Goto (I_4,T):

I_2: E→T. these are same as I_2.

T→T.*F

Goto (I_4,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

goto (I_4,id):

I_5: F→id. - reduced item

Goto (I_6,T):

I_9: E→E+T. - reduced item

T→T.*F

Goto (I_6,F):

I_3: T→F. - reduced item Goto (I_6,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_6,id):

I_5: F→id. reduced item.

Goto (I_7,F):

I_{10}: T→T*F reduced item

Goto (I_7,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_7,id):

I_5: F→id. - reduced item

Goto (I_8,)):

I_{11}: F→(E). reduced item

Goto (I_8,+):

I_{11}: F→(E). reduced item

Goto (I_8,+):

I_6: E→E+.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_9,+):

I_7: T→T*.f

F→.(E)

F→.id

Step IV: Construction of Parse table:

Construction must proceed according to the algorithm 4.8

S→shift items

R→reduce items

Initially E^I→E. is in I_1 so, I = 1.
Set action [I, $] to accept i.e., action [1, $] to Acc

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		R4	R4			
4	S5			S4					3
5		R 6	R 6		R6	R6			
6	S5			S4					3
7	S5			S4					10
8		S 6			S11				
9		R1	S7		r1	r1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

As there are no multiply defined entries, the grammar is SLR®.

STEP – III Finding FOLLOW ( ) set for all non-terminals.

Relevant production

FOLLOW (E) = {$} U FIRST (+T) U FIRST ( )) $E \rightarrow {}^{E}/_{B} + {}^{T}/_{B}$

= {+,), $} $F \rightarrow (E)$

$B\beta$

FOLLOW (T) = FOLLOW (E) U $E \rightarrow T$

FIRST (*F) U $T \rightarrow T*F$

FOLLOW (E) $E \rightarrow E+T$

B

= {+,*,),$}

FOLLOW (F) = FOLLOW (T)

= {*,*,),$}

Step – V:

1. Consider I_0:

1. The item $F \rightarrow .(E)$ gives rise to goto (I_0,C) = I_4, then action [0,C] = shift 4
2. The item $F \rightarrow .id$ gies rise goto (I_0,id) = I_4, then action [0,id] = shift 5

the other items in I_0 yield no actions. Goto (I_0,E) = I_1 then goto [0,E] = 1

Goto (I_0,T) = I_2 then goto [0,T] = 2

Goto (I_0,F) = I_3 then goto [0,F] = 3

2. Consider I_1:

    1. The item $E^I \rightarrow E.$ is the reduced item, so I = 1 This gives rise to

       action [1,$] to accept.

    2. The item $E \rightarrow E.+T$ gives rise to

       goto (I_1,+)=I_6, then action [1,+] = shift 6.

    3. Consider I_2:

        1. The item $E \rightarrow T.$ is the reduced item, so take FOLLOW (E),

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

The first item +, makes action [Z,+] = reduce $E \rightarrow T$. $E \rightarrow T$ is production rule no.2. So action [Z,+] = reduce 2.

The second item, makes action [Z,)] = reduce 2 The third item $, makes action [Z,$] = reduce 2

2. The item $T \rightarrow T.*F$ gives rise to

goto [I_2,*]=I_7, then action [Z,*] = shift 7.

4. Consider I_3:

    1. $T \rightarrow F.$ is the reduced item, so take FOLLOW (T).

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

So, make action [3,+] = reduce 4

Action [3,*] = reduce 4

Action [3,)] = reduce 4

Action [3,$] = reduce 4

In forming item sets a closure operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, say E, then initial items must be included in the set for all productions with E on the left hand side.

The first item set is formed by taking initial item for the start state and then performing the closure operation, giving the item set;

We construct the action and goto as follows:

1. If there is a transition from state I to state J under the terminal symbol K, then set action [I,k] to S_J.

2. If there is a transition under a non-terminal symbol a, say from state 'i' to state 'J', set goto [I,A] to S_J.

3. If state I contains a transition under $ set action [I,$] to accept.

4. If there is a reduce transition #p from state I, set action [I,k] to reduce #p for all terminals k belonging to FOLLOW (A) where A is the subject to production #P.

If any entry is multiply defined then the grammar is not SLR(1). Blank entries are represented by dash (-).

5. Consider I_4 items:

The item F→id gives rise to goto [I_4,id] = I_5 so,

Action (4,id) → shift 5

The item F→.E action (4,c)→ shift 4

The item goto (I_4,F) → I_3, so goto [4,F] = 3

The item goto (I_4,T) → I_2, so goto [4,F] = 2

The item goto (I_4,E) → I_8, so goto [4,F] = 8

6. Consider I_5 items:

F→id. Is the reduced item, so take FOLLOW (F).

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

F→id is rule no.6 so reduce 6

Action (5,+) = reduce 6

Action (5,*) = reduce 6

Action (5,)) = reduce 6

Action (5,)) = reduce 6

Action (5,$) = reduce 6

7. Consider I_6 items:

goto (I_6,T) = I_9, then goto [6,T] = 9 goto (I_6,F) = I_3, then goto [6,F] = 3 goto (I_6,C) = I_4, then goto [6,C] = 4 goto (I_6,id) = I_5, then goto [6,id] = 5

8. Consider I_7 items:

1. goto (I_7,F) = I_{10}, then goto [7,F] = 10
2. goto (I_7,C) = I_4, then action [7,C] = shift 4
3. goto (I_7,id) = I_5, then goto [7,id] = shift 5

9. Consider I_8 items:

1. goto (I_8,)) = I_{11}, then action [8,)] = shift 11
2. goto (I_8,+) = I_6, then action [8,+] = shift 6

10. Consider I_9 items:

1. E→E+T. is the reduced item, so take FOLLOW (E).

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

E→E+T is the production no.1., so

Action [9,+] = reduce 1

Action [9,)] = reduce 1

Action [9,$] = reduce 1

2. goto [I_5,*] = I_7, then acgtion [9,*] = shift 7.

11. Consider I_{10} items:

1. T→T*F. is the reduced item, so take

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

T→T*F is production no.3., so

Action [10,+] = reduce 3

Action [10,*] = reduce 3

Action [10,)] = reduce 3

Action [10,$] = reduce 3

12. Consider I_{11} items:

1. F→(E). is the reduced item, so take

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

F→(E) is production no.5., so

Action [11,+] = reduce 5

Action [11,*] = reduce 5

Action [11,)] = reduce 5

Action [11,$] = reduce 5

VI MOVES OF LR PARSER ON id*id+id:

	STACK	INPUT	ACTION
1.	0	id*id+id$	shift by S5
2.	0id5	*id+id$	sec 5 on *
			reduce by F→id
			If 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, Secn 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
0E1+6F3	$	reduce by r4, i.e.,
		T →F
		Pop2 symbols,
		Push T, see 6 on T
		It is 9, push 9.
0E1+6T9	$	reduce by r1, i.e.,
		E →E+T
		Pop 6 symbols, push E
		See 0 on E, it is 1
		Push 1.
0E1	$	Accept

Procedure for Step-V

The parsing algorithm used for all LR methods uses a stack that contains alternatively state numbers and symbols from the grammar and a list of input terminal symbols terminated by $. For example:

AAbBcCdDeEf/uvwxyz$

Where, a. . .. f are state numbers

A . . .. E are grammar symbols (either terminal or non-terminals) u . . . z are the terminal symbols of the text still to be parsed. The parsing algorithm starts in state I_0 with the configuration –

0 / whole program upto $.

Repeatedly apply the following rules until either a syntactic error is found or the parse is complete.

(i) If action [f,4] = S_i then transform aAbBcCdDeEf / uvwxyz$

to aAbBcCdDeEfui / vwxyz$ This is called a SHIFT transition

(ii) If action [f,4] = #P and production # P is of length 3, say, then it will be of the form P → CDE where CDE exactly matches the top three symbols on the stack, and P is some non-terminal, then assuming goto [C,P] = g

aAbBcCdDEfui / vwxyz$ will transform to

aAbBcPg / vwxyz$

The symbols in the stack corresponding to the right hand side of the production have been replaced by the subject of the production and a new state chosen using the goto table. This is called a REDUCE transition.

(iii) If action [f,u] = accept. Parsing is completed

(iv) If action [f,u] = - then the text parsed is syntactically in-correct.

Canonical LR(O) collection for a grammar can be constructed by augmented grammar and two functions, closure and goto.

The closure operation:

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

i) initially, every item in I is added to closure (I).

CANONICAL LR PARSING*:*

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:

S^I →.S,$ begin with look-a-head (LAH) as $.

We match the item [S^I →.S,$] with the term [A →α.Bβ,a]
In the procedure closure, i.e.,

A = S^I

α = ∈

B = S

β = ∈ 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 ∈ 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→α.Bβ,a] we have, A=S, α=∈, B=C and a=$. FIRST (C$) = FIRST © FIRST© = {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_2 : 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 (I_2,C) I_6

	C→c.C,$	
	C→.cC,$	
	C→.d,$	
Goto (I_2,d)	I_7	
	C→d.,$	→ reduced item.
Goto (I_3,C)	I_8	
	C→cC.,c/d	→ reduced item.
Goto (I_3,C)	I_3	
	C→c.C, c/d	
	C→.cC,c/d	
	C→.d,c/d	
Goto (I_3,d)	I_4	
	C→d.,c/d.	→ reduced item.
Goto (I_6,C)	I_9	
	C→cC.,$	→ reduced item.
Goto (I_6,C)	I_6	
	C→c.C,$	
	C→,cC,$	
	C→.d,$	
Goto (I_6,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 neet 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		

1. **Consider I_0 items:**

The item S→.S.$ gives rise to goto [I_0,S] = I_1 so goto [0,s] = 1.

The item S→.CC, $ gives rise to goto [I_0,C] = I_2 so goto [0,C] = 2.

The item C→.cC, c/d gives rise to goto [I_0,C] = I_3 so goto [0,C] = shift 3

The item C→.d, c/d gives rise to goto [I_0,d] = I_4 so goto [0,d] = shift 4

2. **Consider I_0 items:**

The item S^I→S.,$ is in I_1, then set action [1,$] = accept

3. **Consider I_2 items:**

The item S→C.C,$ gives rise to goto [I_2,C] = I_5. so goto [2,C] = 5

The item C→.cC, $ gives rise to goto [I_2,C] = I_6. so action [0,C] = shift The item C→.d,$ gives rise to goto [I_2,d] = I_7. so action [2,d] = shift 7

4. **Consider I_3 items:**

The item C→.cC, c/d gives rise to goto [I_3,C] = I_8. so goto [3,C] = 8

The item C→.cC, c/d gives rise to goto [I_3,C] = I_3. so action [3,C] = shift 3. The item C→.d, c/d gives rise to goto [I_3,d] = I_4. so action [3,d] = shift 4.

5. **Consider I_4 items:**

The item C→.d, c/d is the reduced item, it is in I_4 so set action [4,c/d] to reduce c→d. (production rule no.3)

6. **Consider I_5 items:**

The item S→CC.,$ is the reduced item, it is in I_5 so set action [5,$] to S→CC (production rule no.1)

7. **Consider I_6 items:**

The item C→c.C,$ gives rise to goto [I_6 ,C] = I_9. so goto [6,C] = 9

The item C→.cC,$ gives rise to goto [I_6 ,C] = I_6. so action [6,C] = shift 6

The item C→.d,$ gives rise to goto [I_6 ,d] = I_7. so action [6,d] = shift 7

8. **Consider I_7 items:**

   The item C→d., $ is the reduced item, it is in I_7.

   So set action [7,$] to reduce C→d (production no.3)

9. **Consider I_8 items:**

   The item C→CC.c/d in the reduced item, It is in Is, so set action[8,c/d] to reduce C→cd (production rale no .2)

10. **Consider I9 items:**

    The item C →cC, $ is the reduced item, It is in I9, so set action [9,$] to reduce C→cC (Production rale no.2)

    If the Parsing action table has no multiply –defined entries, then the given grammar is called as LR(1) grammar

LALR PARSING:

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# (clus them into a single term)

I_0 →same as previous

I_1 → "

I_2 → "

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} →C→d,c/d/$

I_{89} →C→cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d			C
I_o	S_{36}	S_{47}			2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		

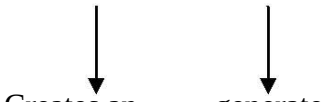
UNIT-III
SEMANTIC ANALYSIS

Intermediate Code Generation

1. Intermediate code forms:

An intermediate code form of source program is an internal form of a program created by the compiler while translating the program created by the compiler while translating the program from a high –level language to assembly code(or)object code(machine code).an intermediate source form represents a more attractive form of target code than does assembly. An optimizing Compiler performs optimizations on the intermediate source form and produces an object module.

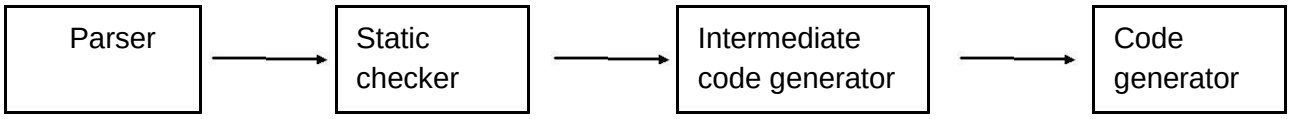
Analysis + syntheses=translation



Creates an generate

targe code Intermediate code

In the analysis –synthesis model of a compiler, the front-end translates a source program into an intermediate representation from which the back-end generates target code, in many compilers the source code is translated into a language which is intermediate in complexity between a HLL and machine code .the usual intermediate code introduces symbols to stand for various temporary quantities.



position of intermediate code generator

We assume that the source program has already been parsed and statically checked.. the various intermediate code forms are:

a) Polish notation
b) Abstract syntax trees(or)syntax trees
c) Quadruples
d) Triples } three address code
e) Indirect triples
f) Abstract machine code(or)pseudocopde **a. postfix notation:**

The ordinary (infix) way of writing the sum of a and b is with the operator in the middle: a+b. the postfix (or postfix polish)notation for the same expression places the operator at the right end, as ab+.

In general, if e1 and e2 are any postfix expressions, and Ø to the values denoted by e1 and e2 is indicated in postfix notation nby e1e2Ø.no parentheses are needed in postfix notation because the position and priority (number of arguments) of the operators permits only one way to decode a postfix expression.

Example:

1. (a+b)*c in postfix notation is ab+c*,since ab+ represents the infix expression(a+b).

2. a*(b+c)is abc+* in postfix.

3. (a+b)*(c+d) is ab+cd+* in postfix.

Postfix notation can be generalized to k-ary operators for any k>=1.if k-ary operator Ø is applied to postfix expression e1,e2,…….ek, then the result is denoted by e1e2…….ek Ø. if we know the priority of each operator then we can uniquely decipher any postfix expression by scanning it from either end.

Example:

Consider the postfix string ab+c*.

The right hand * says that there are two arguments to its left. since the next –to-rightmost symbol is c, simple operand, we know c must be the second operand of *.continuing to the left, we encounter the operator +.we know the sub expression ending in + makes up the first operand of *.continuing in this way ,we deduce that ab+c* is “parsed” as (((a,b)+),c)*.

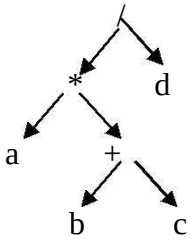
b. syntax tree:

The parse tree itself is a useful intermediate-language representation for a source program, especially in optimizing compilers where the intermediate code needs to extensively restructure.

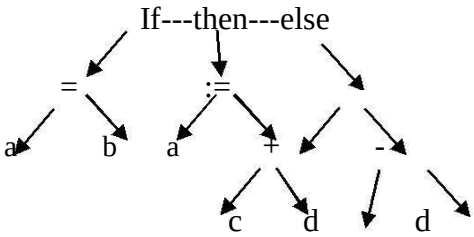
A parse tree, however, often contains redundant information which can be eliminated, Thus producing a more economical representation of the source program. One such variant of a parse tree is what is called an (abstract) syntax tree, a tree in which each leaf represents an operand and each interior node an operator.

Exmples:

1) Syntax tree for the expression a*(b+c)/d



2) syntax tree for **if a=b then a:=c+d else b:=c-d**



Three-Address Code:

• In three-address code, there is at most one operator on the right side of aninstruction; that is, no built-up arithmetic expressions are permitted.

x+y*z t1 = y * z t2 = x + t1

• **Example**

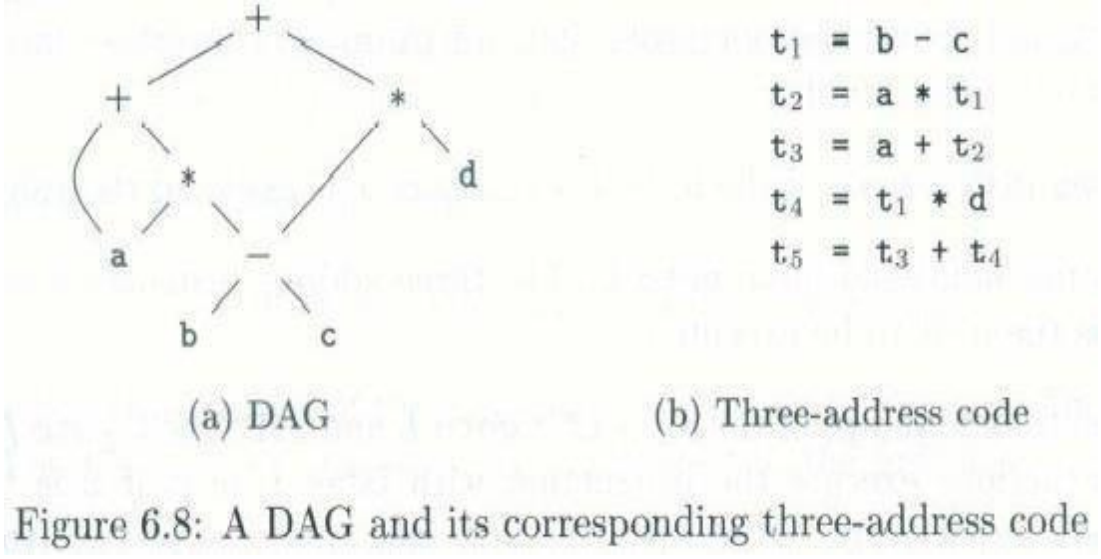


Figure 6.8: A DAG and its corresponding three-address code

Problems:

Write the 3-address code for the following expression

1. if(x + y * z > x * y +z) a=0;
2. (2 + a * (b – c / d)) / e
3. A :=b * -c + b * -c

Address and Instructions

•

• **Example** Three-address code is built from two concepts: addresses and instructions.

• An address can be one of the following:

– A name: A source name is replaced by a pointer to its symbol table entry.

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

– A constant

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

– A compiler-generated temporary

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

A list of common three-address instruction forms: Assignment statements

– x= y op z, where op is a binary operation

– x= op y, where op is a unary operation

– Copy statement: x=y

– Indexed assignments: x=y[i] and x[i]=y

– Pointer assignments: x=&y, *x=y and x=*y

Control flow statements

– Unconditional jump: goto L

– Conditional jump: if x relop y goto L ; if x goto L; if False x goto L

– Procedure calls: call procedure p with n parameters and **return y**, is Optional

param x1 param x2

…

param xn call p, n

– **do i = i +1; while (a[i]<v);**

```
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
```

(a) Symbolic labels. (b) Position numbers.

The multiplication i * 8 is appropriate for an array of elements that each take 8 units of space.

C. quadruples:

• Three-address instructions can be implemented as objects or as record with fields for the operator and operands.

• Three such representations

– Quadruple, triples, and indirect triples

• A quadruple (or quad) has four fields: op, arg1, arg2, and result.

Example D. Triples

• A triple has only three fields: op, arg1, and arg2

• Using triples, we refer to the result of an operation x op y by its position, rather by an explicit temporary name.

Example

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

(a) Three-address code

	op	arg_1	arg_2	*result*
0	minus	c		t_1
1	*	b	t_1	t_2
2	minus	c		t_3
3	*	b	t_3	t_4
4	+	t_2	t_4	t_5
5	=	t_5		a
		...		

(b) Quadruples

d. Triples:

• A triple has only three fields: op, arg1, and arg2

• Using triples, we refer to the result of an operation x op y by its position, rather by an explicit temporary name.

Example

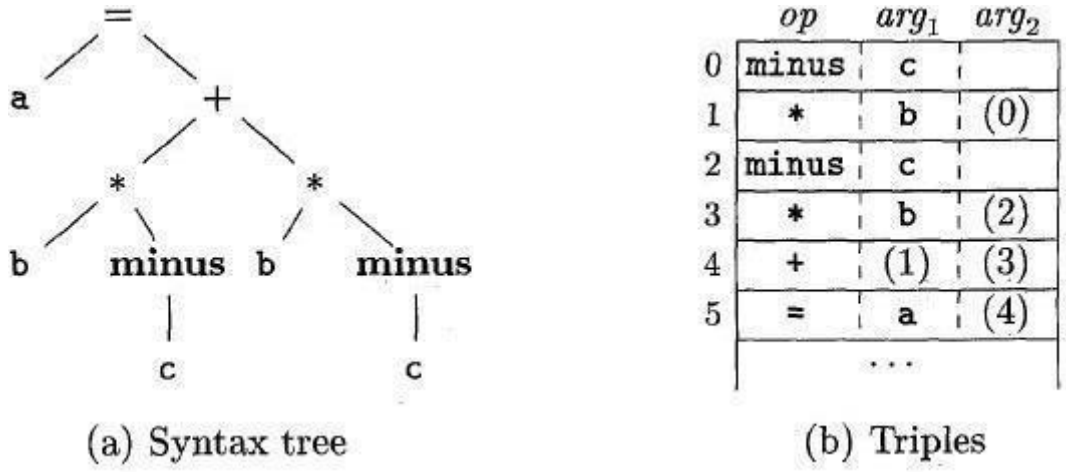


Fig: Representations of a = b * - c + **b** * - c

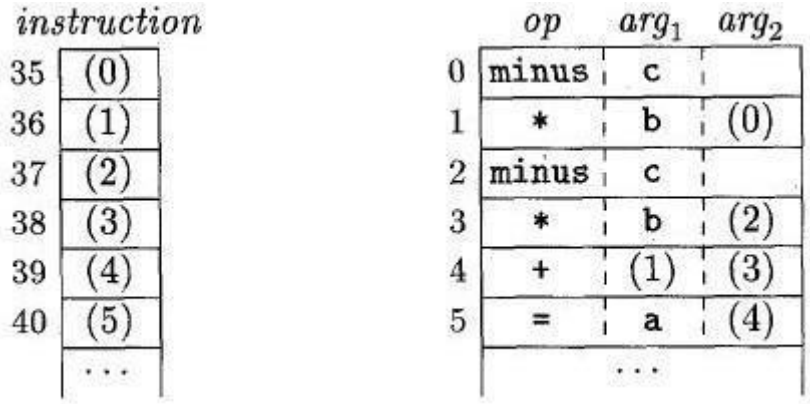


Fig: Indirect triples representation of 3-address code

-> The benefit of **Quadruples** over **Triples** can be seen in an optimizing compiler, where instructions are often moved around.

->With **quadruples**, if we move an instruction that computes a temporary **t**, then the instructions that use **t** require no change. With **triples**, the result of an operation is referred to by its position, so moving an instruction may require changing all references to that result. **This problem does not occur with indirect triples**.

Single-Assignment Static Form

Static single assignment form (SSA) is an intermediate representation that facilitates certain code optimization.

• Two distinct aspects distinguish SSA from three –address code.

– All assignments in SSA are to variables with distinct names; hence the term static single-assignment.

```
p = a + b          p1 = a + b
q = p - c          q1 = p1 - c
p = q * d          p2 = q1 * d
p = e - p          p3 = e - p2
q = p + q          q2 = p3 + q1
```

(a) Three-address code. (b) Static single-assignment form.

Figure 6.13: Intermediate program in three-address code and SSA

```
if (flag) x = -1; else x = 1;
y = x * ;a
if (flag) x1 = -1; else x2 = 1;
X3 = ϕ(x1, x2)
```

2. Type Checking:

•A compiler has to do semantic checks in addition to syntactic checks. •Semantic Checks

–Static –done during compilation

–Dynamic –done during run-time

•**Type checking** is one of these static checking operations.

–we may not do all type checking at compile-time.

–Some systems also use dynamic type checking too.

•A **type system** is a collection of rules for assigning type expressions to the parts of a program.

•A **type checker** implements a type system.

•A **sound type** system eliminates run-time type checking for type errors.

•A programming language is strongly-typed, if every program its compiler accepts will execute without type errors.

In practice, some of type checking operations is done at run-time (so, most of the programming languages are not strongly yped).

Type Expression:

The type of a language construct is denoted by a type expression.

A type expression can be:

A basic type a primitive data type such as integer, real, char, Boolean, …

type-error to signal a type error void: no type

A type name a name can be used to denote a type expression.

A type constructor applies to other type expressions.

•**arrays**: If T is a type expression, then array (I,T)is a type expression where I denotes index range. Ex: array (0..99,int)

•**products**: If T1and T2 are type expressions, then their Cartesian product T1 x T2 is a type expression. Ex: int x int

•**pointers**: If T is a type expression, then pointer (T) is a type expression. Ex: pointer (int)

•**functions**: We may treat functions in a programming language as mapping from a domain type D to a range type R. So, the type of a function can be denoted by the type expression D→R where D are R type expressions. Ex: int→int represents the type of a function which takes an int value as parameter, and its return type is also int.

Type Checking of Statements:

S ->d= E { if (id.type=E.type then S.type=void

else S.type=type-error }

S ->if E then S1 { if (E.type=boolean then S.type=S1.type

else S.type=type-error }

S->while E do S1 { if (E.type=boolean then S.type=S1.type

else S.type=type-error }

Type Checking of Functions:

E->E1(E2) {

else E.type=type-error }

Ex: int f(double x, char y) { ... }

f: double x char->int

argume types return type

Structural Equivalence of Type Expressions:

•How do we know that two type expressions are equal?

•As long as type expressions are built from basic types (no type names), we may use structural equivalence between two type expressions

Structural Equivalence Algorithm (sequin):

if (s and t are same basic types) then return true

else if (s=array(s1,s2) and t=array(t1,t2)) then return (sequiv(s1,t1) and sequiv(s2,t2)) else if (s = s1 x

s2and t = t1 x t2) then return (sequiv(s1,t1) and sequiv(s2,t2))

else if (s=pointer(s1) and t=pointer(t1)) then return (sequiv(s1,t1))

else if (s = s1 else return false

Names for Type Expressions:

•In some programming languages, we give a name to a type expression, and we use that name as a type expression afterwards.

type link = ↑cell; ? p,q,r,s have same types ? var p,q : link;

var r,s : ↑cell

•How do we treat type names?

–Get equivalent type expression for a type name (then use structural equivalence), or

–Treat a type name as a basic type

3.Syntax Directed Translation:

- A formalist called as syntax directed definition is used fort specifying translations for programming language constructs.
- A syntax directed definition is a generalization of a context free grammar in which each grammar symbol has associated set of attributes and each and each productions is associated with a set of semantic rules

Definition of (syntax Directed definition) SDD :

SDD is a generalization of CFG in which each grammar productions X->α is associated with it a set of semantic rules of the form

a: = f(b1,b2…..bk)

Where a is an attributes obtained from the function f.

• A syntax-directed definition is a generalization of a context-free grammar in which:

– Each grammar symbol is associated with a set of attributes.

– This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.

– Each production rule is associated with a set of semantic rules.

• Semantic rules set up dependencies between attributes which can be represented by a dependency graph.

• This dependency graph determines the evaluation order of these semantic rules.

• Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

The two attributes for non terminal are :

1) Synthesized attribute (S-attribute) : (↑)

An attribute is said to be synthesized attribute if its value at a parse tree node is determined from attribute values at the children of the node

2) Inherited attribute: (↑,→)

An inherited attribute is one whose value at parse tree node is determined in terms of attributes at the parent and | or siblings of that node.

- The attribute can be string, a number, a type, a, memory location or anything else.
- The parse tree showing the value of attributes at each node is called an annotated parse tree.

The process of computing the attribute values at the node is called annotating or decorating the parse tree.**Terminals** can have synthesized attributes, but not inherited attributes.

Annotated Parse Tree

• A parse tree showing the values of attributes at each node is called an **Annotated parse tree**.

• The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.

• Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Ex1:1) Synthesized Attributes : Ex: Consider the CFG :

S→ EN E→ E+T E→E-T E→ T T→ T*F T→T/F T→F F→ (E) F→digit N→;

Solution: The syntax directed definition can be written for the above grammar by using semantic actions for each production.

Production rule	Semantic actions
S →EN	S.val=E.val
E →E1+T	E.val =E1.val + T.val
E →E1-T	E.val = E1.val – T.val
E →T	E.val =T.val
T →T*F	T.val = T.val * F.val
T →T|F	T.val =T.val | F.val
F → (E)	F.val =E.val
T →F	T.val =F.val
F →digit	F.val =digit.lexval
N →;	can be ignored by lexical Analyzer as; I is terminating symbol

For the Non-terminals E,T and F the values can be obtained using the attribute “Val”.

The taken digit has synthesized attribute “lexval”.

In S→EN, symbol S is the start symbol. This rule is to print the final answer of expressed.

Following steps are followed to Compute S attributed definition

1. Write the SDD using the appropriate semantic actions for corresponding production rule of the given Grammar.

2. The annotated parse tree is generated and attribute values are computed. The Computation is done in bottom up manner.

3. The value obtained at the node is supposed to be final output.

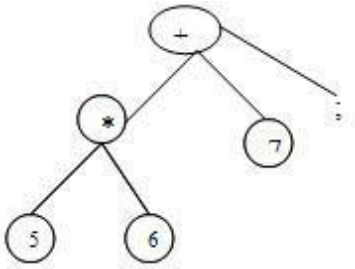
PROBLEM 1:

Consider the string 5*6+7; Construct Syntax tree, parse tree and annotated tree.

Solution:

The corresponding annotated parse tree is shown below for the string 5*6+7;

Syntax tree:



Annotated parse tree :

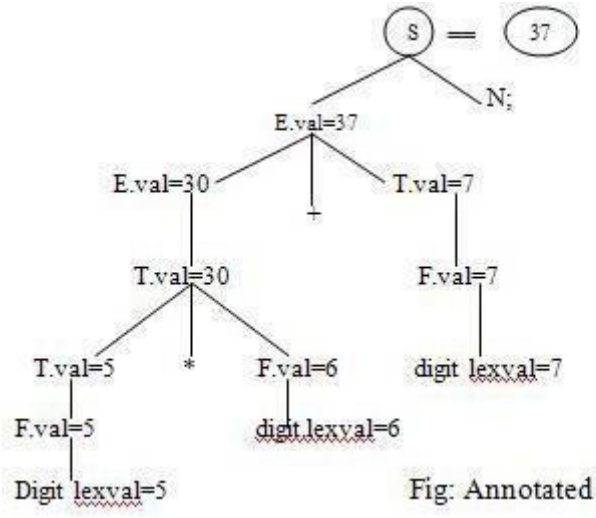


Fig: Annotated parse tree

Advantages: SDDs are more readable and hence useful for specifications

Disadvantages: not very efficient.

Ex2:

PROBLEM : Consider the grammar that is used for Simple desk calculator. Obtain the Semantic action and also the annotated parse tree for the string

3*5+4n. L→En E→E1+T

E→T

T→T1*F

T→F

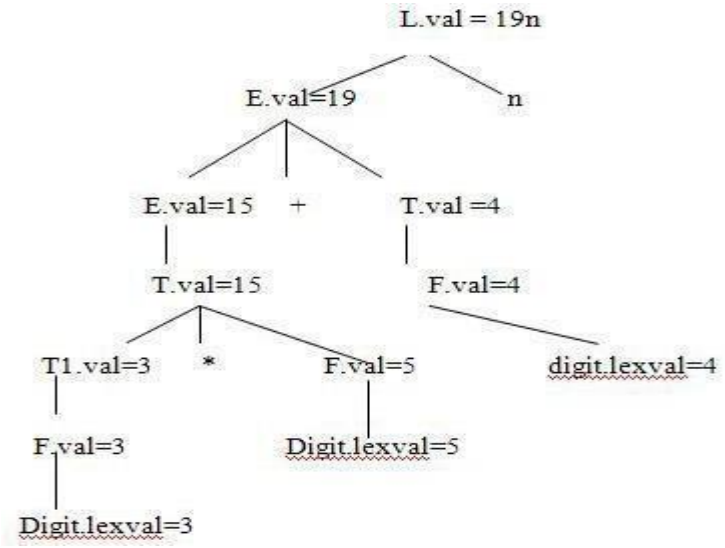
F→ (E)

F→digit

Solution :

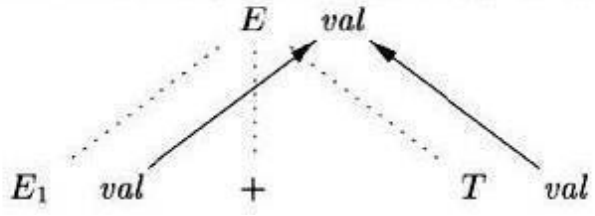
Production rule	**Semantic actions**
L→En	L.val=E.val
E→E1+T	E.val=E1.val + T.val
E→T	E.val=T.val
T→T1*F	T.val=T1.val*F.val
T→F	T.val=F.val
F→(E)	F.val=E.val
F→digit	F.val=digit.lexval

The corresponding annotated parse tree U shown below, for the string 3*5+4n.



Dependency Graphs:

Figure 5.6. *E.val* is synthesized from E_1.*val* and E_2.*val*



Dependency graph and topological sort:

- For each parse-tree node, say a node labeled by grammar symbol X, the dependency graph has a node for each attribute associated with X.
- If a semantic rule associated with a production p defines the value of synthesized attribute A.b in terms of the value of X.c. Then the dependency graph has an edge from X.c to A.b
- If a semantic rule associated with a production p defines the value of inherited attribute B.c in terms of the value X.a. Then , the dependency graph has an edge from X.a to B.c.

Applications of Syntax-Directed Translation

• Construction of syntax Trees

– The nodes of the syntax tree are represented by objects with a suitable number of fields.

– Each object will have an ***op*** field that is the label of the node.

– The objects will have additional fields as follows

• If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function ***Leaf (op, val)*** creates a leaf object.

• If nodes are viewed as records, the Leaf returns a pointer to a new record for a leaf.

• If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function

Node takes two or more arguments:

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

Syntax-Directed Translation Schemes

A SDT scheme is a context-free grammar with program fragments embedded within production bodies .The program fragments are called semantic actions and can appear at any position within the production body.

Any SDT can be implemented by first building a parse tree and then pre-forming the actions in a left-to-right depth first order. i.e during preorder traversal.

The use of SDT's to implement two important classes of SDD's

1. If the grammar is LR parsable, then SDD is S-attributed.
2. If the grammar is LL parsable, then SDD is L-attributed.

Postfix Translation Schemes

The postfix SDT implements the desk calculator SDD with one change: the action for the first production prints the value. As the grammar is LR, and the SDD is S-attributed.

L →E n {print(E.val);}
E → E1 + T { E.val = E1.val + T.val }
E → E1 - T { E.val = E1.val - T.val }
E → T { E.val = T.val }
T → T1 * F { T.val = T1.val * F.val } T → F { T.val = F.val }
F → (E) { F.val = E.val }
F → digit { F.val = digit.lexval }

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

Symbol table:

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

- 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 Interface:

The basic operations defined on a symbol table include:

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

  For example, a **delete** operation removes a name previously inserted Some identifiers

  become invisible (out of scope) after exiting a block

- This interface provides an abstract view of a symbol table.
- Supports the simultaneous existence of multiple tables
- Implementation can vary without modifying the interface

Basic Implementation Techniques:

First consideration is how to **insert** and **lookup** names

Variety of implementation techniques

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

Ordered List

If an array is sorted, it can be searched using binary search – $O(\log_2 n)$

Insertion into a sorted array is expensive – $O(n)$ on average

Useful when set of names is known in advance – table of reserved words

Binary Search Tree

Can grow dynamically

Insertion and lookup are $O(\log 2\ n)$ on average

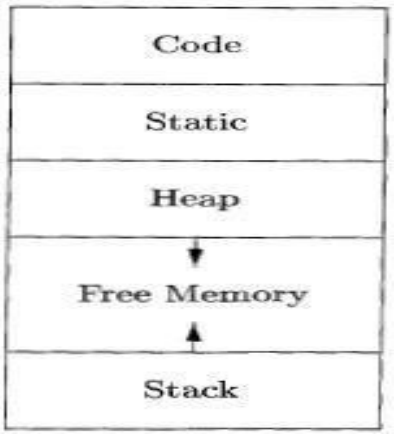
Hash Tables and Hash Functions:

- 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.

Storage Allocation:

- Compiler must do the storage allocation and provide access to variables and data
- Memory management
  - Stack allocation
  - Heap management
  - Garbage collection

<u>Storage Organization:</u>



• Assumes a logical address space

- Operating system will later map it to physical addresses, decide how touse cache memory, etc.

• Memory typically divided into areas for

- Program code
- Other static data storage, including global constants and compilergenerated data
- Stack to support call/return policy for procedures
- Heap to store data that can outlive a call to a procedure

Static vs. Dynamic Allocation:

- Static: Compile time, Dynamic: Runtime allocation
- Many compilers use some combination of following
- Stack storage: for local variables, parameters and so on
- Heap storage: Data that may outlive the call to the procedure that created it

Stack allocation is a valid allocation for procedures since procedure calls are nest

Example:

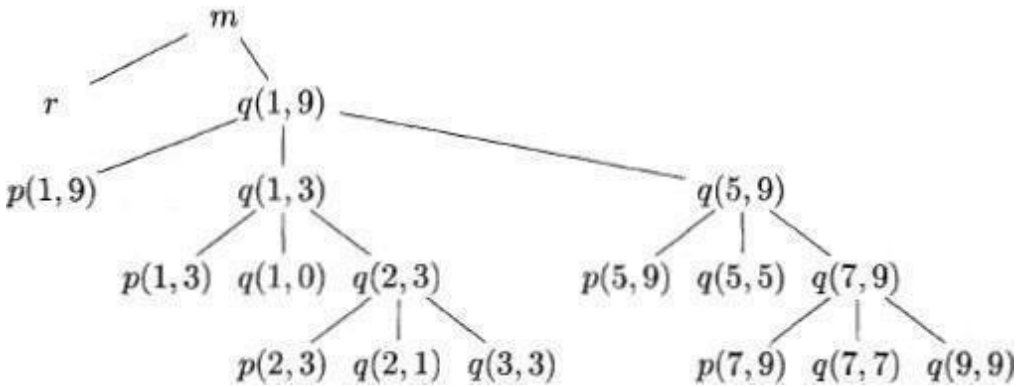
Consider the quick sort program

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

Activation for Quicksort:

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

Activation tree representing calls during an execution of quicksort:



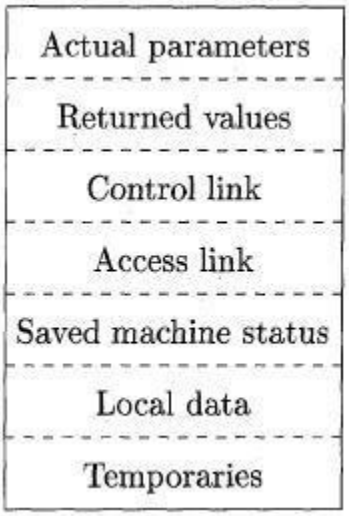
UNIT - IV

RUN TIME STORAGE ORGANIZATION

Activation records

- Procedure calls and returns are usually managed by a run-time stack called the control stack.
- Each live activation has an activation record (sometimes called a frame)
- The root of activation tree is at the bottom of the stack
- The current execution path specifies the content of the stack with the last
- Activation has record in the top of the stack.

A General Activation Record



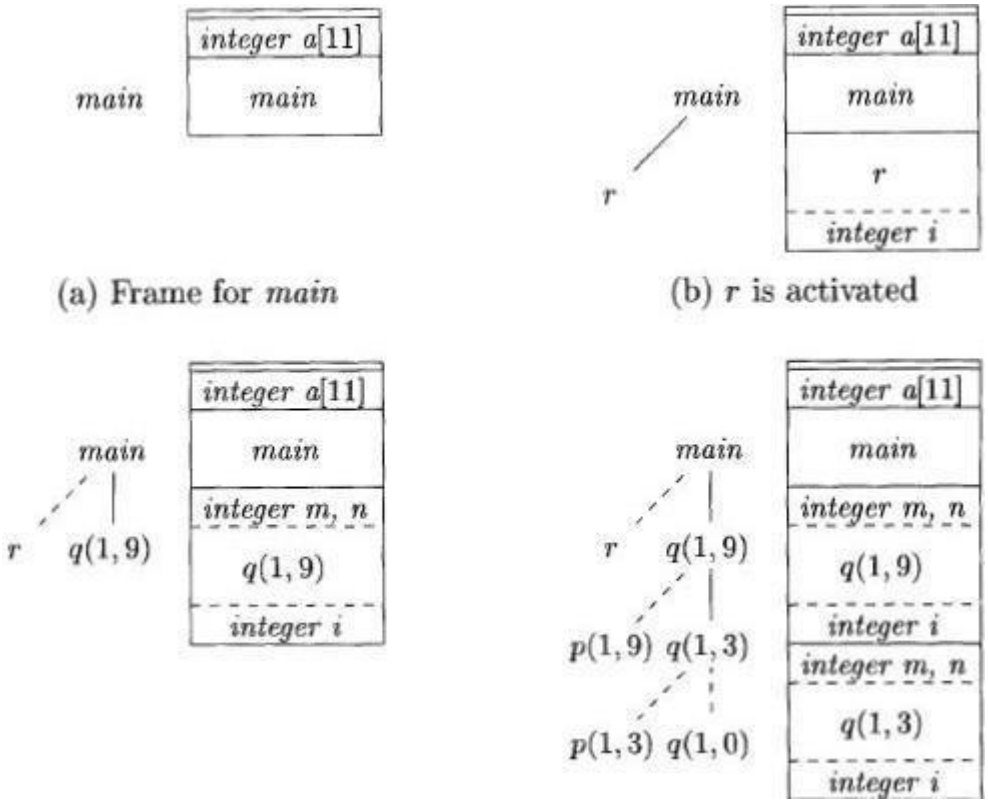
Activation Record

- Temporary values
- Local data
- A saved machine status
- An "access link"
- A control link
- Space for the return value of the called function
- The actual parameters used by the calling procedure

- Elements in the activation record:

- Temporary values that could not fit into registers.
- Local variables of the procedure.
- Saved machine status for point at which this procedure called. Includes return address and contents of registers to be restored.
- Access link to activation record of previous block or procedure in lexical scope chain.
- Control link pointing to the activation record of the caller.
- Space for the return value of the function, if any.
- actual parameters (or they may be placed in registers, if possible)

Downward-growing stack of activation records:

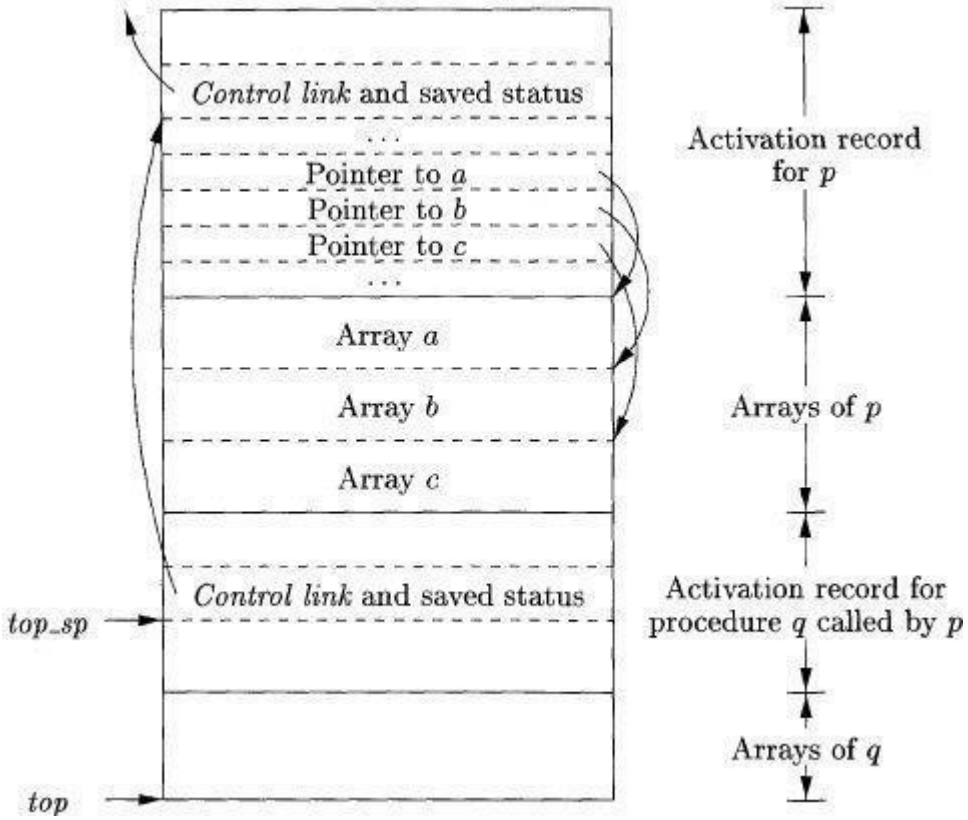


Designing Calling Sequences:

- Values communicated between caller and callee are generally placed at the beginning of callee's activation record
- Fixed-length items: are generally placed at the middle
- Items whose size may not be known early enough: are placed at the end of activation record

- We must locate the top-of-stack pointer judiciously: a common approach is to have it point to the end of fixed length fields

Access to dynamically allocated arrays:



ML:

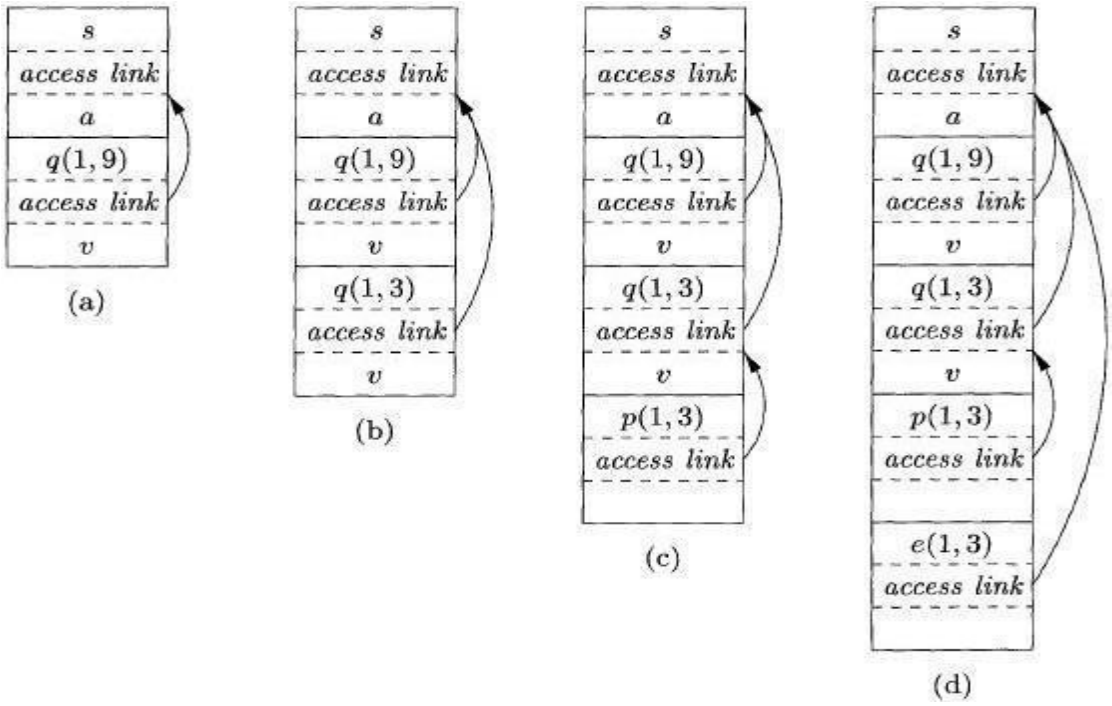
- ML is a functional language
- Variables are defined, and have their unchangeable values initialized, by a statementof the form:
  val (name) = (expression)
- Functions are defined using the syntax:

  fun (name) ((arguments)) = (body)
- For function bodies we shall use let-statements of the form: let
  (list of definitions) in (statements) end

A version of quick sort, in ML style, using nested functions:

```
1) fun sort(inputFile, outputFile) =
       let
2)         val a = array(11,0);
3)         fun readArray(inputFile) = ··· ;
4)               ··· a ··· ;
5)         fun exchange(i,j) =
6)               ··· a ··· ;
7)         fun quicksort(m,n) =
               let
8)                 val v = ··· ;
9)                 fun partition(y,z) =
10)                      ··· a ··· v ··· exchange ···
               in
11)                ··· a ··· v ··· partition ··· quicksort
               end
       in
12)        ··· a ··· readArray ··· quicksort ···
       end;
```

Access links for finding nonlocal data:

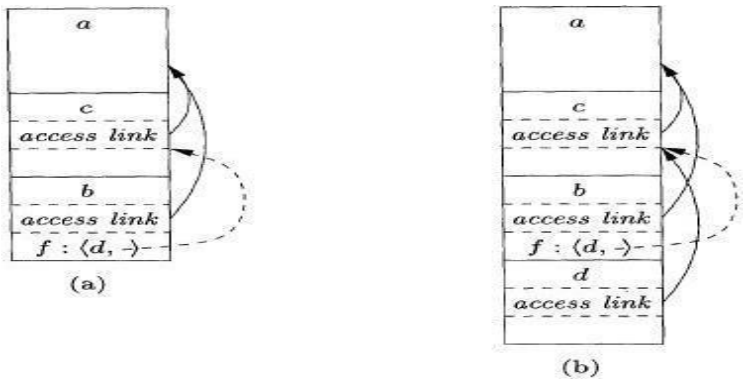


Sketch of ML program that uses function-parameters:

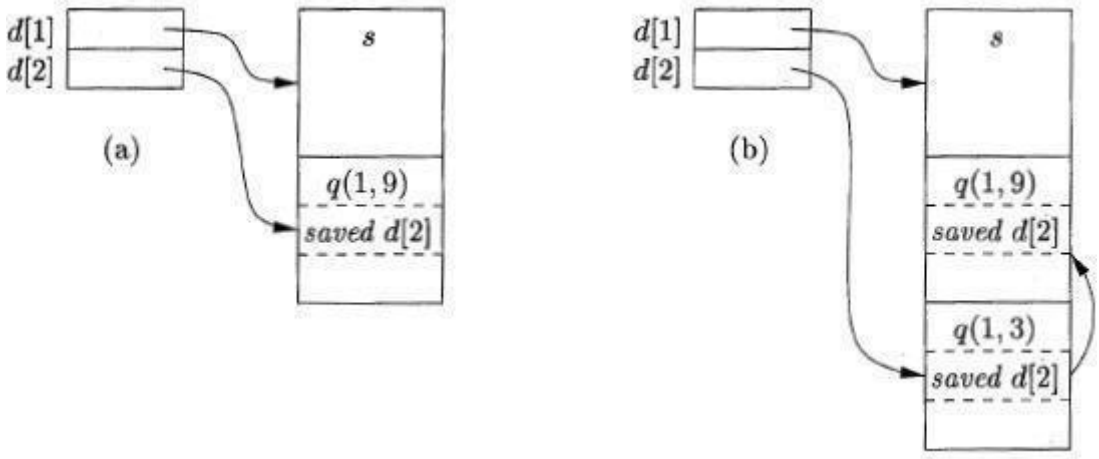
```
fun a(x) =
    let
        fun b(f) =
            ··· f ··· ;
        fun c(y) =
            let
                fun d(z) = ···
            in
                ··· b(d) ···
            end
    in
        ··· c(1) ···
    end;
```

\

Actual parameters carry their access link with them:



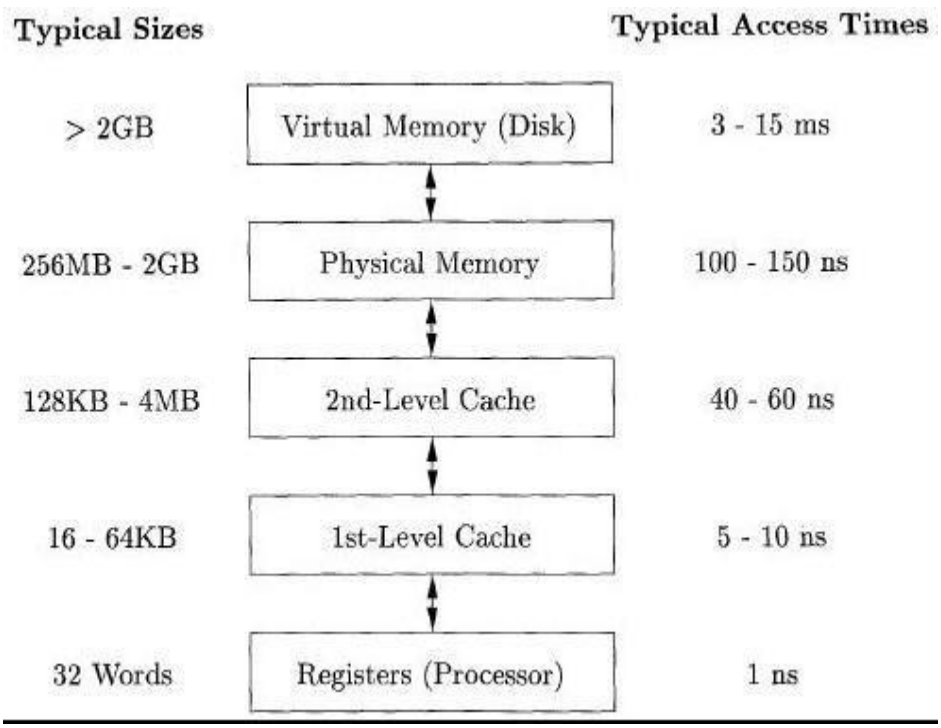
Maintaining the Display:



Memory Manager:

- Two basic functions:
  - Allocation
  - Deallocation
- Properties of memory managers:
  - Space efficiency
  - Program efficiency
  - Low overhead

Typical Memory Hierarchy Configurations:



Locality in Programs:

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

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

CODE OPTIMIZATION

1. INTRODUCTION

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

- Optimizations are classified into two categories. They are

- Machine independent optimizations:

- Machine dependant optimizations:

Machine independent optimizations:

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

Machine dependant optimizations:

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

The criteria for code improvement transformations:

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

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

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

- The transformation must be worth the effort. It does not make sense for a compiler writer

to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. "Peephole" transformations of this kind are simple enough and beneficial enough to be included in any compiler.

- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flow analysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
- A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

- Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function it computes.

- The transformations

    - Common sub expression elimination, o Copypropagation,
    - Dead-code elimination, and
    - Constant folding, are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.
- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

Common Sub expressions elimination:

- An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

- For example

  t1: =4*i t2: =a [t1] t3: =4*j t4:=4*i t5: =n
  t 6: =b [t 4] +t 5

The above code can be optimized using the common sub-expression elimination as t1:

=4*i t2: =a [t1] t3: =4*j t5: =n
t6: =b [t1] +t5

The common sub expression t 4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

Copy Propagation:

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

For example: x=Pi;

……
A=x*r*
r;

The optimization using copy propagation can be done as follows:

A=Pi*r*r;

Here the variable x is eliminated

Dead-Code Eliminations:

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

Exampl
e: i=0;
if(i=1)
{
a=b+5;
}

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

Constant folding:

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

- One advantage of copy propagation is that it often turns the copy statement into dead code.

For example,
a=3.14157/2 can be replaced by
a=1.570 there by eliminating a division operation.

Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

- Three techniques are important for loop optimization:
  - code motion, which moves code outside a loop;
  - Induction -variable elimination, which we apply to replace variables from inner loop.
  - Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

Code Motion:

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

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

Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called induction variables.

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t4 completely; t4 is used in B3 and j in B4.
- However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

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

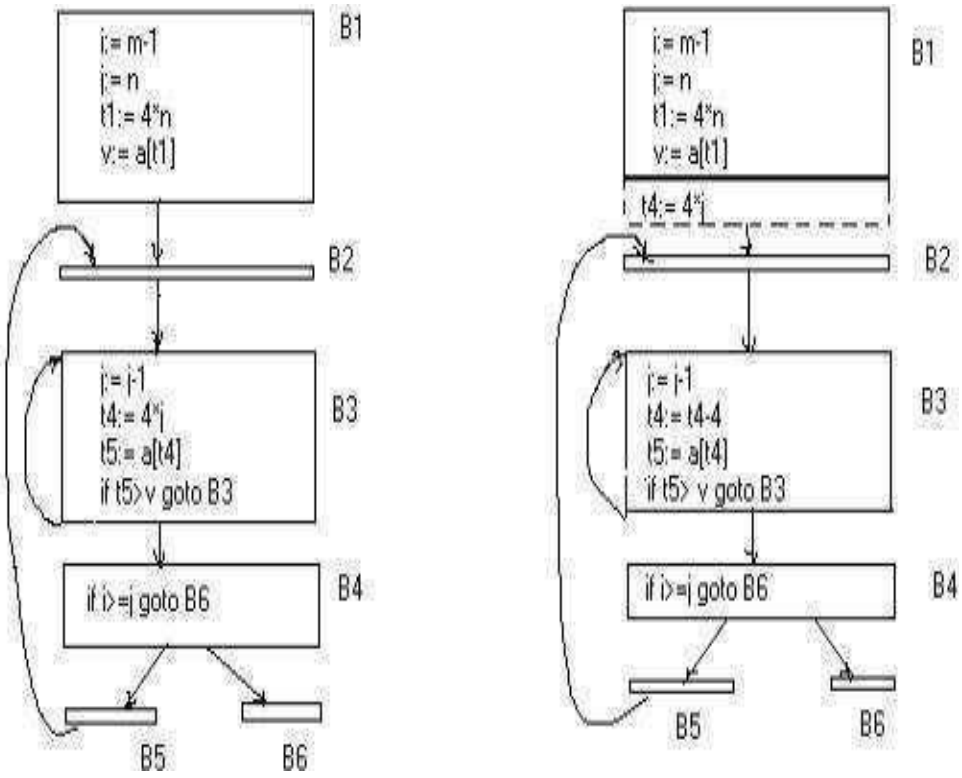
The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

Reduction in Strength:

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

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

3. OPTIMIZATION OF BASIC BLOCKS



There are two types of basic block optimizations. They

are : Structure -Preserving Transformations
Algebraic Transformations

Structure- Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

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

Common sub-expression elimination:

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

Example:

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

The 2nd and 4th statements compute the same expression: b+c

and a-d Basic block can be transformed to

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

Dead code elimination:

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

Renaming of temporary variables:

- A statement t:=b+c where t is a temporary name can be changed to u:=b+c where u is

another temporary name, and change all uses of t to u.

- In this we can transform a basic block to its equivalent block called normal-form block.

Interchange of two independent adjacent statements:

Two statements

t1:

=b

+c

t2:

=x

+y

can be interchanged or reordered in its computation in the basic block when value of t1 does not affect the value of t2.

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.

- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.

- The relational operators <=, >=, <, >, + and = sometimes generate unexpected common sub expressions.

- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

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

the following intermediate code may be generated:

a :=b+c t :=c+d

e :=t+b

Example: x:=x+0 can be removed
x:=y2 can be replaced by a cheaper statement x:=y*y**

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

UNIT – V
Control flow &Data flow Analysis

Flow graph

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

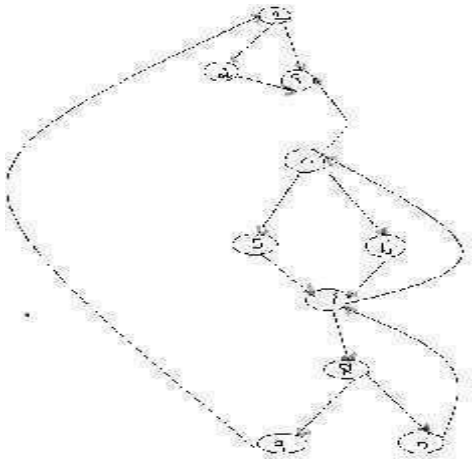
Dominators:

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

Example:
*In the flow graph below,
*Initial node,node1 dominates every node. *node 2 dominates itself *node 3 dominates all but 1 and
2. *node 4 dominates all but 1,2 and 3.

*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10.

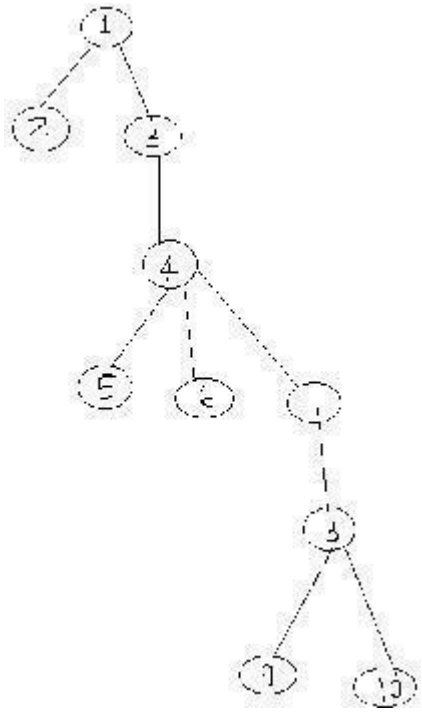
*node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.

- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendents in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n.

- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d dom m.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

D(4)={1,3,4}

D(5)={1,3,4,5}

D(6)={1,3,4,6}

D(7)={1,3,4,7}

D(8)={1,3,4,7,8}

D(9)={1,3,4,7,8,9}

D(10)={1,3,4,7,8,10}

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are
    - A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop, or it would not be the sole entry to the loop.
    - There must be at least one wayto iterate the loop(i.e.)at least one path back to the header.
- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If a→b is an edge, b is the head and a is the tail. These types of edges are called as back edges.

Example:

In the above graph,

→ 4 4 DOM 7

→7 7 DOM 10

→ 3

→ 3

9 →1

- The above edges will form loop in flow graph.
- Given a back edge n → d, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.
Input: A flow graph G and a back edge n→d

Output: The set loop consisting of all nodes in the natural loop n→d.

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

Procedure insert(m);
if m is not in *loop* **then begin** *loop* := *loop* U {m}; push *m* onto *stack*
end;
stack : =empty; *loop* : ={*d*}; *insert*(*n*);

while *stack* is not empty **do begin**

pop *m*, the first element of *stack*, off *stack*; **for** each predecessor *p* of *m* **do** *insert(p)*

end Inner

LOOP:

- If we use the natural loops as "the loops", then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.

- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

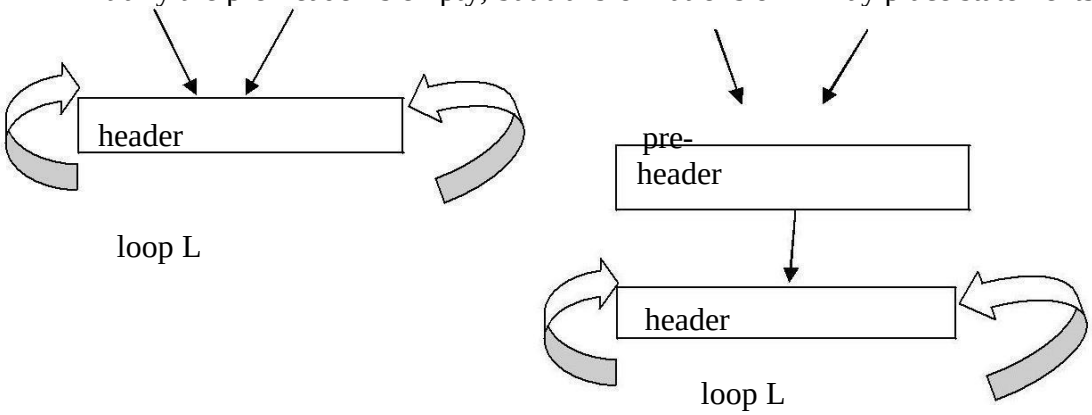
Pre-Headers:

- Several transformations require us to move statements "before the header". Therefore begin treatment of a loop L by creating a new block, called the preheater.

- The pre -header has only the header as successor, and all edges which formerly entered the header of Lfrom outside L instead enter the pre-header.

- Edges from inside loop L to the header are not changed.

- Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before **(b) After**

Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.

- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible. The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.

- ***Definition***:

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

- The forward edges from an acyclic graph in which every node can be reached from initial node of G.

- The back edges consist only of edges where heads dominate theirs tails.

- Example: The above flow graph is reducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.

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

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the target program.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions by a shorter or faster sequence, whenever possible.
- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
- We shall give the following examples of program transformations that are characteristic of peephole optimizations:
  - Redundant-instructions elimination
  - Flow-of-control optimizations
  - Algebraic simplifications
  - Use of machine idioms
  - Unreachable Code

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R0,a
(2) MOV a,R0

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

Unreachable Code:

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

```
#define debug
0 ….
If ( debug ) {

Print debugging information

}
```

In the intermediate representations the if-statement may be translated as:

```
debug =1 goto L2

goto L2

L1: print debugging information

L2:……(a)
```

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

```
If debug ≠1 goto L2

Print debugging information

L2:……(b)
```

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

```
If debug ≠0 goto L2

Print debugging information

L2:                    ……(c)
```

- As the argument of the first statement of (c) evaluates to a constant true, it can be replaced by goto L2. Then all the statement that print debugging aids are manifestly

  unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

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

  goto L1
  ….
  L1: gotoL2 by the
  sequence goto L2
  ….
  L1: goto L2

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

  if a < b goto L1
  ….
  L1: goto L2
  can be replaced by Ifa < b goto L2
  ….
  L1: goto L2

- Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto.
  Then the sequence

  goto L1

  ……..
  L1: if a <b goto L2

  L3:……

  …..(1)

- Maybe replaced by Ifa<b goto L2
  goto L3
  …….

OBJECT CODE GENERATION:

code generation is the process by which a compiler's **code generator** converts some intermediate representation of source code into a form (e.g., machine code) that can be readily executed by a machine.

Code generation can be considered as the final phase of compilation. Through post code generation, optimization process can be applied on the code, but that can be seen as a part of code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

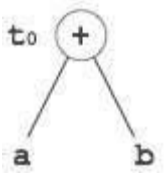
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

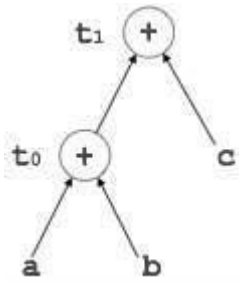
$t_0 = a + b$

$t_1 = t_0 + c$

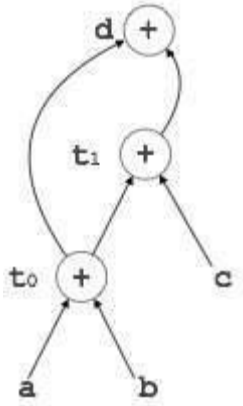
$d = t_0 + t_1$



$[t_0 = a + b]$



$[t_1 = t_0 + c]$



$[d = t_0 + t_1]$

REGISTER ALLOCATION:

Register allocation In the IR, we assumed an unlimited number of registers (to ease IR code generation) This is obviously not the case on a physical machine (typically, from 5 to 10 general-purpose registers) Registers can be accessed quickly and operations can be performed on them directly Using registers intelligently is therefore a critical step in any compiler (can make a di↵erence in orders of magnitude) Register allocation is the process of assigning variables to registers and managing data transfer in and out of the registers

GENERIC CODE GENERATION ALGORITHM:

Assume that for each operator in the statement, there is a corresponding target language operator
The computed results can be left in registers as long as possible, storing them only if the register is needed for another computation or just before a procedure call, jump or labeled statement

Register and Address Descriptors

These are the primary data structures used by the code generator. They keep track of what values are in each registers, as well as where a given value resides

- Each register has a register descriptor containing the list of variables currently stored in this register. At the start of the basic block, all register descriptors are empty. It keeps track of recent/current variable in each register. It is constructed whenever a new register is needed
- Each variable has an address descriptor containing the list of locations where this variable is currently stored. Possibilities are its memory location and one or more registers
- The memory location might be in the static area, the stack, r presumably the heap. The register descriptors can be computed from the address descriptors. For each name of the block, an address descriptors is maintained that keep track of location where current value of name is found at runtime

There are basically three aspects to be considered in code generation:

Choosing registers
Generating instructions
Managing descriptors

Register allocation is done in a function getReg(Instruction). The instruction generation algorithms

uses getReg() and the descriptors

The generation of machine instruction is done as follows:

Given a TAC, OP x, Y(i.e., x=x OP y), generation of machine instructions proceeds as follows:

Step 1: Call getReg(OP, x, y) to get R_x and R_y, the registers to be used for x and y respectively. GetReg merely selects the registers, it does not gurantee that the desired values are present in these registers.

Step 2: Check the register descriptor for R_y. If y is not present in Ry, check the address descriptor for y and issue LD R_y, y

Step 3: Similar treatment is done for R_x

Step 4: Generate the instruction OP R_x, R_y

When the TAC is x = y, step 1 and step 2 are same (getReg() will set Rx =Ry). Step 3 is empty and step 4 is omitted. If 'y' was already in a register before the copy instruction, no code is generated at this point. Since the value of 'y' is not in its memory location, we may need to store this value back into 'y' at block exit.

All variables needed by (dynamically) subsequent blocks (i.e., that live-on-exit) have their current values in their memory locations. Such live variables are identified as follows:

- Temporaries never live beyond a basic block. Hence, they are ignored
- Variables dead on exit are also ignored
- All live on exit variables need to be stored in their memory location on exit from the block. So, check the address descriptor for each live on exit variable. If its own memory location is not listed, generate ST X, R, where R is a register listed in the address descriptor

The management of register and address descriptor is performed as follows:

- For a register R, let Desc(R) be is register descriptor. For a program variable x, let Desc(x) be its address descriptor. The management of descriptor for load, store, operation and copy are given below

Load: LD R, x

Desc(R) = x(removing everything else from Desc(R))

Add R to Desc(x) (leaving alone everything else in Desc(x))

Remove R from Desc(w) for all $w \neq x$

Store: ST x, R

Add the memory location of x to Desc(x)

Operation: OP Rx, Ry implementing the quas OP x, y

Desc(R_x) = x

Desc(x) = R_x

After operation R_x has R_x OP R_y

Copy: for x = y after processing the load

Add x to Desc(R_y) (note y not x)

Desc(x) = R_y

Minimize the number of registers used:

- When a register holds a temporary value and there are no subsequent uses for this value, we reuse that register
- When a register holds the value of a program variable and there are no subsequent uses of this value, we reuse that register providing this value also in the memory location for the variable
- When a register holds the values of a program variable and all subsequent uses of this value are preceded by a redefinition, we could reuse this register. But to know about all subsequent uses, one

may require live/dead-on-exit knowledge

Assume a, b, c and d are program variables and t, u, v are compiler generated temporaries. These are represented as t1,t1,t2 and t$3. The code generated for different TACs is given below:

```
t = a –b
        LD R1, a
        LD R2, b
        SUB R1, R2
U = a – c
        LD R3, a
        LD R2, c
        SUB R3, R2
v = t + u
        ADD R1, R3
a = d
        LD R2, d
        ST a, R2
d = v + u
        ADD R1, R3
        ST d, R1
Exit
```

DAG for Register Allocation:

Code generation from DAG is much simpler than the linear sequence of three address code

With the help of DAG one can rearrange sequence of instructions and generate and efficient code

There exist various algorithms which are used for generating code from DAG. They are:

Code Generation from DAG:-

- Rearranging Order
- Heuristic Ordering
- Labeling Algorithm

Rearranging Order

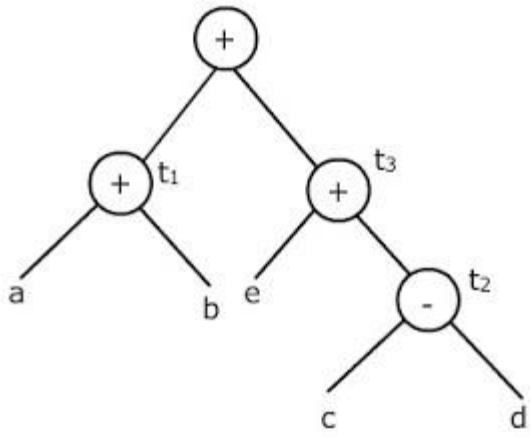
These address code's order affects the cost of the object code which is being generated

Object code with minimum cost can be achieved by changing the order of computations

Example:

```
t1:= a + b
t2:= c – d
t3:= e + t2
t4:= t1 + t3
```

For the expression (a+b) + (e+(c-d)), a DAG can be constructed for the above sequence as shown below



DAG for *(a + b) + (e + (c - d))*

The code is thus generated by translating the three address code line by line

```
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, t        t1:= a+b
MOV e, R0        R1 has c-d
ADD R0, R1        /* R1 contains e + (c – d)*/
MOV t1, R0       /R0 contains a + b*/
ADD R1, R0
MOV R0, t4
```

Now, if the ordering sequence of the three address code is changed

```
t2:= c - d
t3:= e + t2
t1:= a + b
t4:= t1 + t3
```

Then, an improved code is obtained as:

```
MOV c, R0
SUB D, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, t4
```

Heuristic Ordering

The algorithm displayed below is for heuristic ordering. It lists the nodes of a DAG such that the node's reverse listing results in the computation order.