UNIT -1

1.1 OVERVIEW OF LANGUAGE PROCESSING SYSTEM

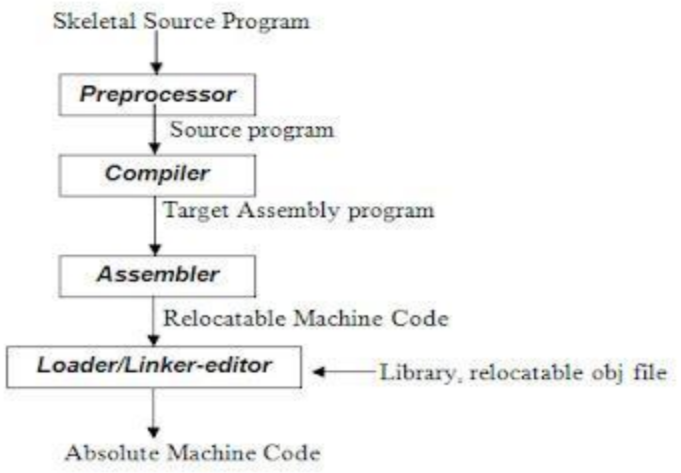


Fig 1.1 Language –processing System

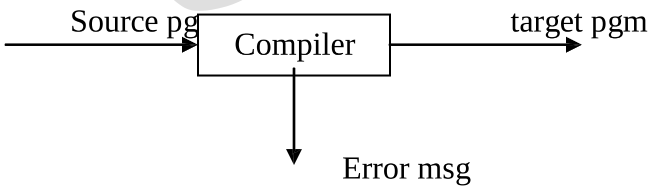
1.2 Preprocessor

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

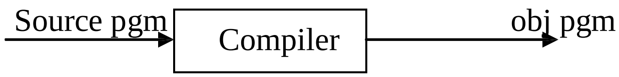
1. *Macro processing:* A preprocessor may allow a user to define macros that are short hands for longer constructs.
2. *File inclusion:* A preprocessor may include header files into the program text.
3. *Rational preprocessor:* these preprocessors augment older languages with more modern flow-of-control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

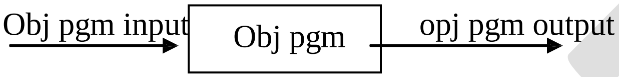
1.3 COMPILER

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



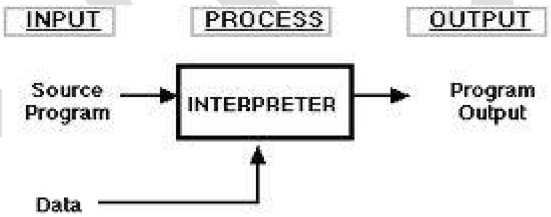
Executing a program written n HLL programming language is basically of two parts. the source program must first be compiled translated into a object program. Then the results object program is loaded into a memory executed.





1.4 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).

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



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

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

Advantages:

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

Disadvantages:

- The execution of the program is *slower*.
- *Memory* consumption is more.

2 ***Loader and Link-editor:***

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

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

1.6 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 the hll.

1.7 TYPE OF TRANSLATORS:-

- INTERPRETOR
- COMPILER
- PREPROSSESSOR

1.9 STRUCTURE OF THE COMPILER DESIGN

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

PHASES OF A COMPILER

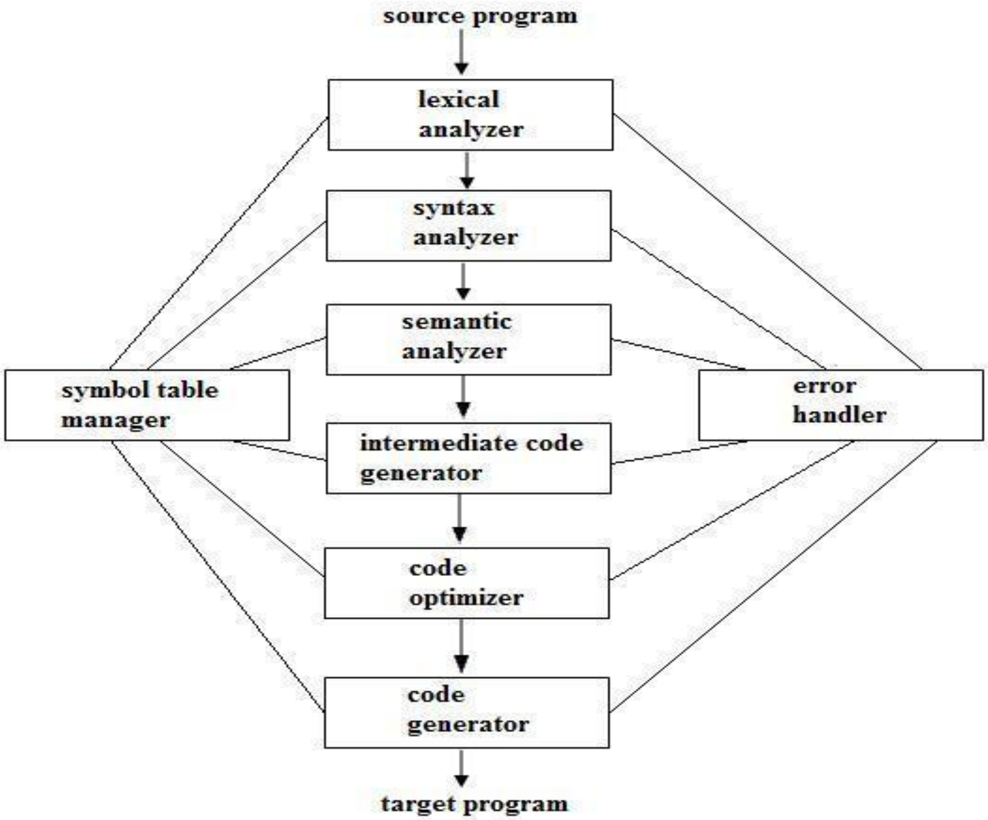


Fig 1.5 Phases of a compiler

No-of-sub processes called **‘phases’**.

Lexical Analysis:-

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

Syntax Analysis:-

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

Intermediate Code Generations:-

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

Code Optimization :-

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

Code Generation:-

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

Table Management (or) Book-keeping:-

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

Error Handlers:-

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

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

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

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

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

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

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

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

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

Intermediate Code Generation:-

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

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

Code Optimization

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

1, Local Optimization:-

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

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

Goto **L3**

L2 :

This can be replaced by a single statement

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

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

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

Might be evaluated as

T1 := B + C

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

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

2, Loop Optimization:-

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

Code generator :-

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

Table Management OR Book-keeping :-

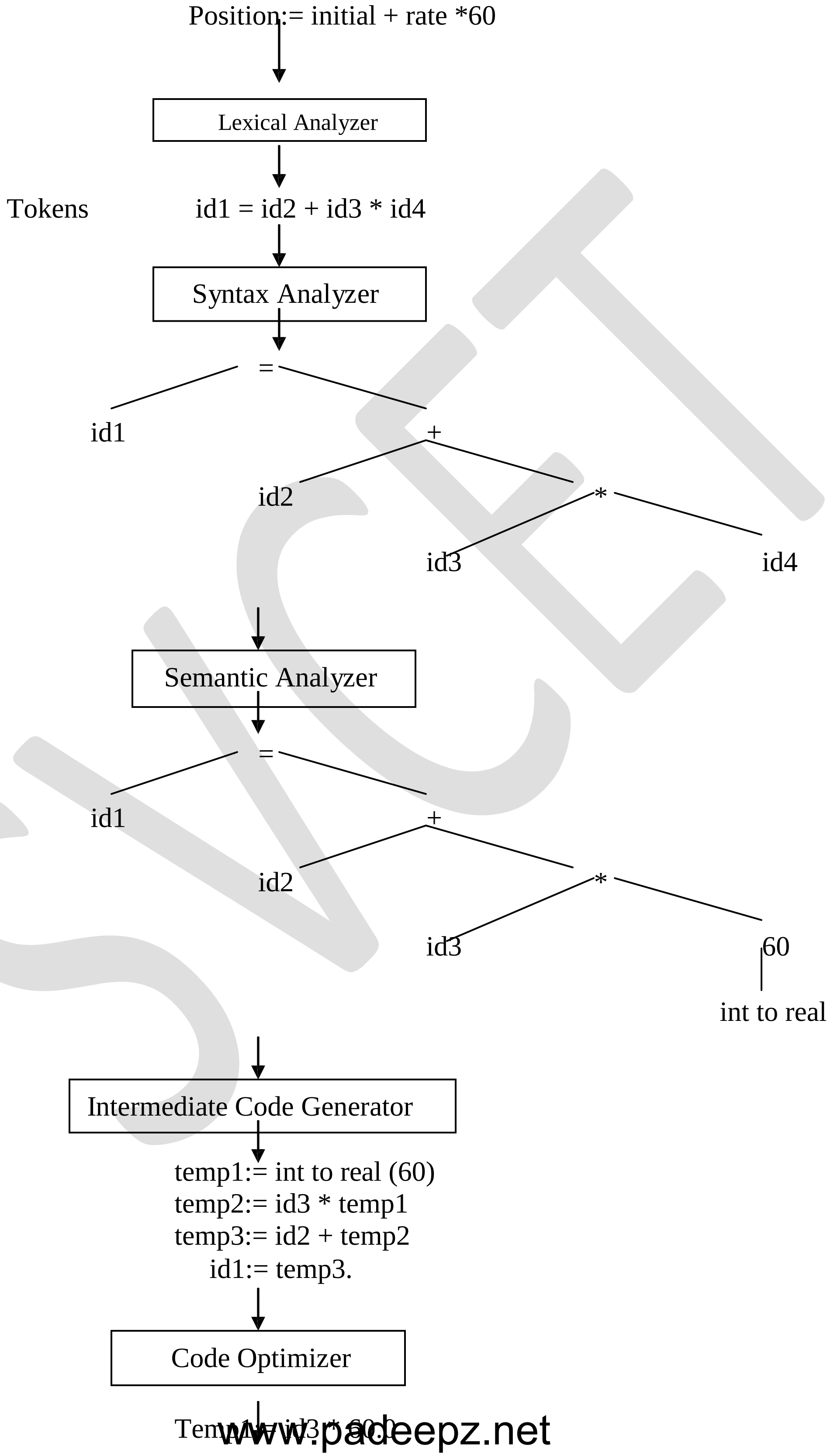
A compiler needs to collect information about all the data objects that appear in the source program. The information about data objects is collected by the early phases of the compiler-lexical and syntactic analyzers. The data structure used to record this information is called as Symbol Table.

Error Handing :-

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

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

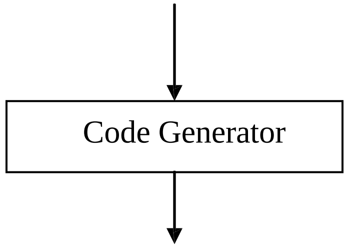
Example:



temp1:= int to real (60)
temp2:= id3 * temp1
temp3:= id2 + temp2
id1:= temp3.

Temp1:= id3 * 60.0

Id1:= id2 +temp1



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

1.10 TOKEN

LA reads the source program one character at a time, carving the source program into a sequence of automatic units called ‘Tokens’.

1, Type of the token.

2, Value of the token.

Type : variable, operator, keyword, constant

Value : N1ame of variable, current variable (or) pointer to symbol table.

If the symbols given in the standard format the LA accepts and produces token as output. Each token is a sub-string of the program that is to be treated as a single unit. Token are two types.

1, Specific strings such as IF (or) semicolon.

2, Classes of string such as identifiers, label, constants.

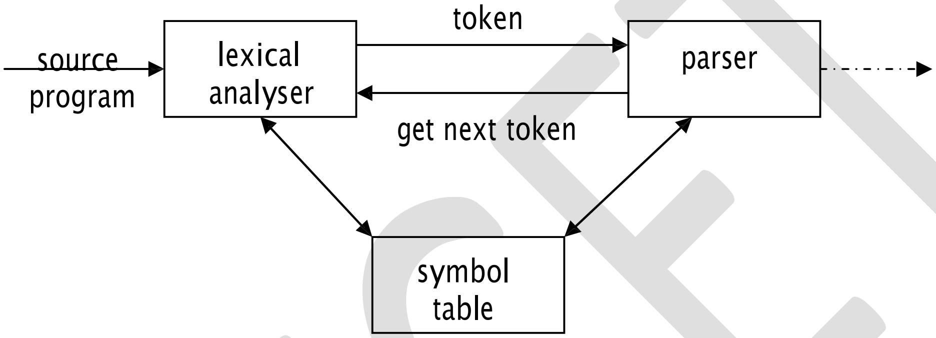
UNIT II- LEXICAL ANALYSIS

LEXICAL ANALYSIS

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

THE ROLE OF THE LEXICAL ANALYZER

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



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

ISSUES OF LEXICAL ANALYZER

There are three issues in lexical analysis:

- To make the design simpler.
- To improve the efficiency of the compiler.
- To enhance the computer portability.

TOKENS

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

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

Lexeme	Token type
sum	Identifier
=	Assignment operator
3	Number
+	Addition operator
2	Number
;	End of statement

LEXEME:

Collection or group of characters forming tokens is called Lexeme.

PATTERN:

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

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

Attributes for Tokens

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

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

ERROR RECOVERY STRATEGIES IN LEXICAL ANALYSIS:

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

1) Deleting an extraneous character.

2) Inserting a missing character.

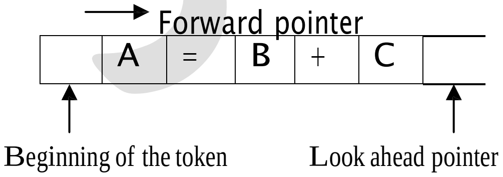
3) Replacing an incorrect character by a correct character.

4) Transforming two adjacent characters.

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

INPUT BUFFERING

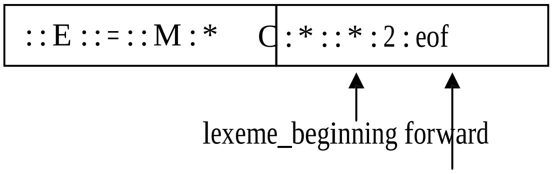
We often have to look one or more characters beyond the next lexeme before we can be sure we have the right lexeme. As characters are read from left to right, each character is stored in the buffer to form a meaningful token as shown below:



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

BUFFER PAIRS

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



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

Advancing forward pointer:

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

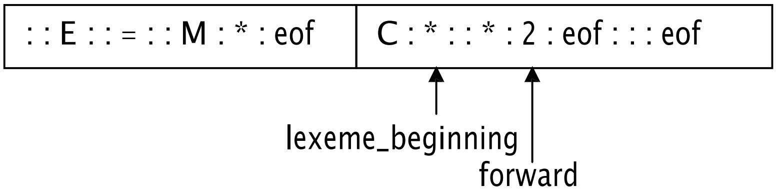
Code to advance forward pointer:

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

SENTINELS

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

- The sentinel arrangement is as shown below:



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

Code to advance forward pointer:

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

<u>SPECIFICATION OF TOKENS</u>

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 string s.

For example, ban is a prefix of banana.

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

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

4. The **proper prefixes, suffixes, and substrings** of a string s are those prefixes, suffixes, and substrings, respectively of s that are not 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}

1. Union : L U S={0,1,a,b,c}
2. Concatenation : L.S={0a,1a,0b,1b,0c,1c}
3. Kleene closure : L^*={ ε,0,1,00….}
4. 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, a) (r)|(s) is a

   regular expression denoting the language L(r) U L(s).
   b) (r)(s) is a regular expression denoting the language L(r)L(s).
   c) (r)* is a regular expression denoting (L(r))*.
   d) (r) is a regular expression denoting L(r).

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

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

6. | has lowest precedence and is left associative.

Regular set

A language that can be defined by a regular expression is called a regular set.
If two regular expressions r and s denote the same regular set, we say they are equivalent and
Write r = s.

There are a number of algebraic laws for regular expressions that can be used to manipulate into equivalent forms.
For instance, r|s = s|r is commutative; r|(s|t)=(r|s)|t is associative.

Regular Definitions

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

$d_1 \rightarrow r_1$
$d_2 \rightarrow r_2$
………
$d_n \rightarrow r_n$

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

Example: Identifiers is the set of strings of letters and digits beginning with a letter. Regular definition for this set:
letter → A | B | …. | Z | a | b | …. | z |
digit → 0 | 1 | …. | 9
id → letter (letter | digit) *

Shorthands

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

1. One or more instances (+):

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

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

- Thus the regular expression a^+ denotes the set of all strings of one or more a's.

- The operator $^+$ has the same precedence and associativity as the operator *.

2. Zero or one instance (?):

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

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

Transition diagrams

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

Transition diagram for relational operators

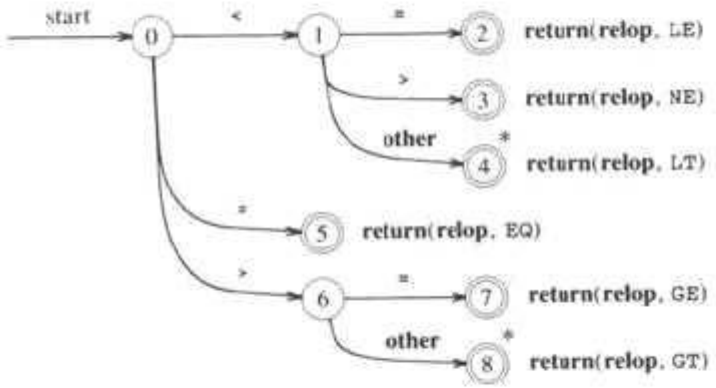
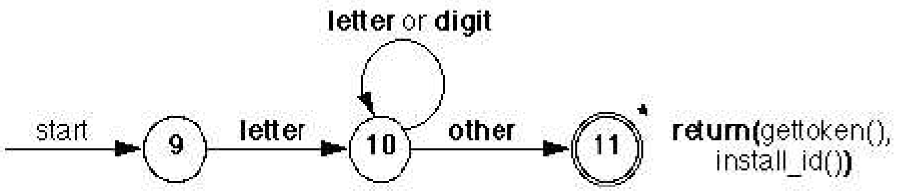


Fig. 3.12. Transition diagram for relational operators.

Transition diagram for identifiers and keywords



A LANGUAGE FOR SPECIFYING LEXICAL ANALYZER

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

- Lex
- YACC

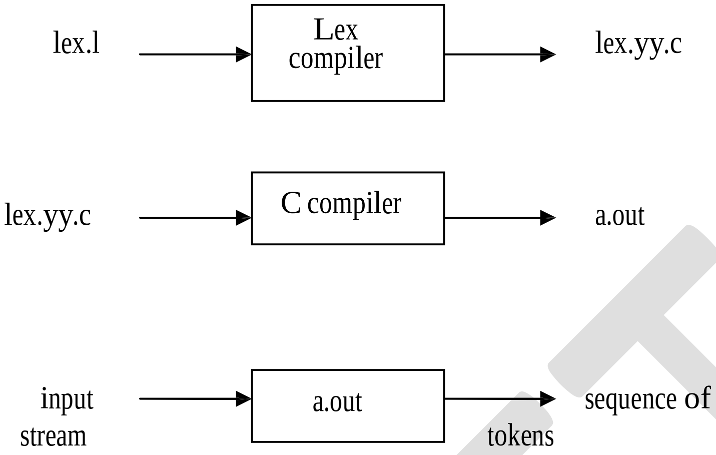
LEX

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

Creating a lexical analyzer

- First, a specification of a lexical analyzer is prepared by creating a program lex.l in the Lex language. Then, lex.l is run through the Lex compiler to produce a C program lex.yy.c.

- Finally, lex.yy.c is run through the C compiler to produce an object program a.out, which is the lexical analyzer that transforms an input stream into a sequence of tokens.



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 p_1
{action$_1$}
p_2 {action$_2$}
…
p_n {action$_n$}
where p_i is regular expression and action$_i$ describes what action the lexical analyzer should take when pattern p_i matches a lexeme. Actions are written in C code.

- **User subroutines** are auxiliary procedures needed by the actions. These can be compiled separately and loaded with the lexical analyzer.

YACC- YET ANOTHER COMPILER-COMPILER

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

FINITE AUTOMATA

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

Types of Finite Automata

There are tow types of Finite Automata:

Non-deterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA)

Non-deterministic Finite Automata

NFA is a mathematical model that consists of five tuples denoted by

M = $\{Q_n, \Sigma, \delta, q_0, f_n\}$

Q_n – finite set of states

Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states

q_0 – starting state

f_n – final state

Deterministic Finite Automata

DFA is a special case of a NFA in which i) no state has an ε-transition.

ii) there is at most one transition from each state on any input.

DFA has five tuples denoted by

M = $\{Q_d, \Sigma, \delta, q_0, f_d\}$

Q_d – finite set of states

Σ – finite set of input symbols

δ – transition function that maps state-symbol pairs to set of states

q_0 – starting state

f_d – final state

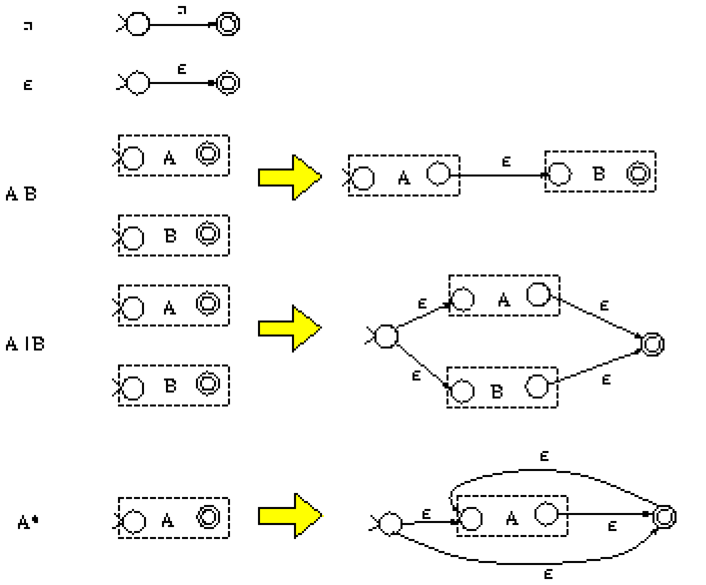
Converting a Regular Expression into a Deterministic Finite Automaton

The task of a scanner generator, such as flex, is to generate the transition tables or to synthesize the scanner program given a scanner specification (in the form of a set of REs). So it needs to convert a RE into a DFA. This is accomplished in two steps: first it converts a RE into a non-deterministic finite automaton (NFA) and then it converts the NFA into a DFA.

A NFA is similar to a DFA but it also permits multiple transitions over the same character and transitions over ε. The first type indicates that, when reading the common character associated with these transitions, we have more than one choice; the NFA succeeds if at least one of these choices succeeds. The εtransition doesn't consume any input characters, so you may jump to another state for free.

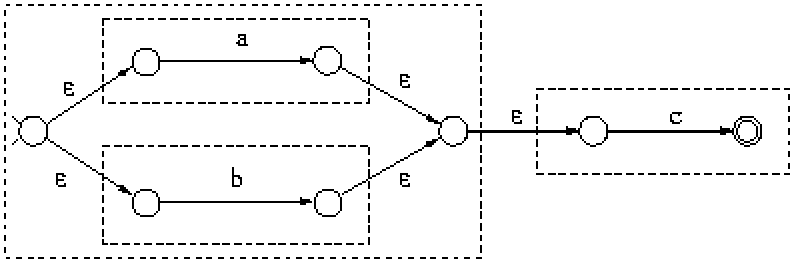
Clearly DFAs are a subset of NFAs. But it turns out that DFAs and NFAs have the same expressive power. The problem is that when converting a NFA to a DFA we may get an exponential blowup in the number of states.

We will first learn how to convert a RE into a NFA. This is the easy part. There are only 5 rules, one for each type of RE:



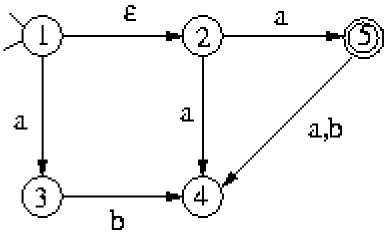
The algorithm constructs NFAs with only one final state. For example, the third rule indicates that, to construct the NFA for the RE *AB*, we construct the NFAs for *A* and *B* which are represented as two boxes with one start and one final state for each box. Then the NFA for *AB* is constructed by connecting the final state of *A* to the start state of *B* using an empty transition.

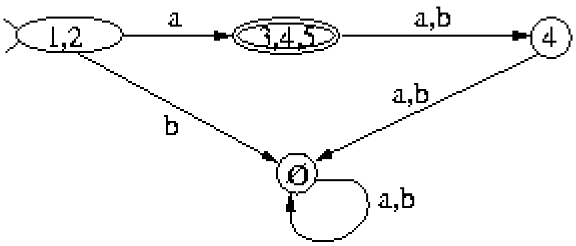
For example, the RE (*a*|b)*c* is mapped to the following NFA:



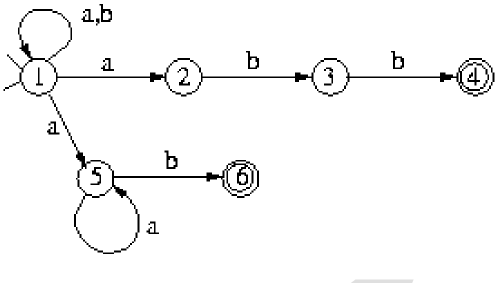
The next step is to convert a NFA to a DFA (called *subset construction*). Suppose that you assign a number to each NFA state. The DFA states generated by subset construction have sets of numbers, instead of just one number. For example, a DFA state may have been assigned the set {5,6,8}. This indicates that arriving to the state labeled {5,6,8} in the DFA is the same as arriving to the state 5, the state 6, or the state 8 in the NFA when parsing the same input. (Recall that a particular input sequence when parsed by a DFA, leads to a unique state, while when parsed by a NFA it may lead to multiple states.)

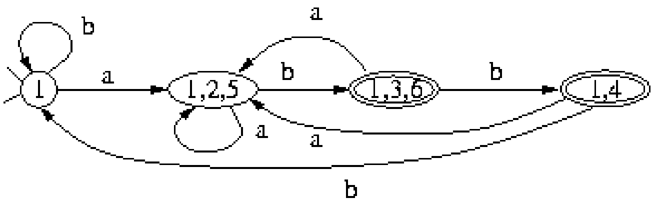
First we need to handle transitions that lead to other states for free (without consuming any input). These are the ε transitions. We define the *closure* of a NFA node as the set of all the nodes reachable by this node using zero, one, or more ε transitions. For example, The closure of node 1 in the left figure below





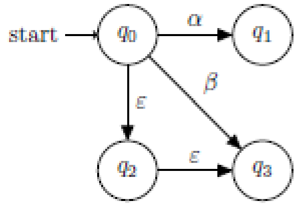
is the set {1,2}. The start state of the constructed DFA is labeled by the closure of the NFA start state. For every DFA state labeled by some set $\{s_1, \ldots, s_n\}$ and for every character *c* in the language alphabet, you find all the states reachable by s_1, s_2, ..., or s_n using *c* arrows and you union together the closures of these nodes. If this set is not the label of any other node in the DFA constructed so far, you create a new DFA node with this label. For example, node {1,2} in the DFA above has an arrow to a {3,4,5} for the character *a* since the NFA node 3 can be reached by 1 on *a* and nodes 4 and 5 can be reached by 2. The *b* arrow for node {1,2} goes to the error node which is associated with an empty set of NFA nodes. The following NFA recognizes $(a|b)^*(abb\,|\,a^+b)$, even though it wasn't constructed with the 5 RE-to-NFA rules. It has the following DFA:

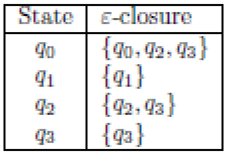




Converting NFAs to DFAs

To convert an NFA to a DFA, we must and a way to remove all "-transitions and to ensure that there is one transition per symbol in each state. We do this by constructing a DFA in which each state corresponds to a set of some states from the NFA. In the DFA, transitions from a state S by some symbol go to the state S that consists of all the possible NFA-states that could be reached by from some NFA state q contained in the present DFA state S. The resulting DFA \simulates" the given NFA in the sense that a single DFA-transition represents many simultaneous NFA-transitions. The first concept we need is the "E-closure pronounced \epsilon closure". The " -closure of an NFA state q is the set containing q along with all states in the automaton that are reachable by any number of " E-transitions from q . In the following automaton, the " E-closures are given in the table to the right:

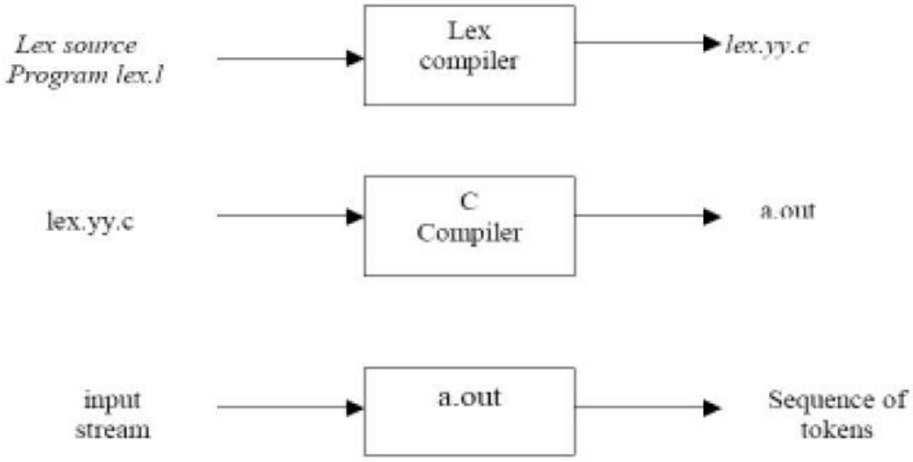




State	ε-closure
q_0	$\{q_0, q_2, q_3\}$
q_1	$\{q_1\}$
q_2	$\{q_2, q_3\}$
q_3	$\{q_3\}$

Likewise, we can done the "-closure of a set of states to be the states reachable by " -transitions from its members. In other words, this is the union of the " -closures of its elements. To convert our NFA to its DFA counterpart, we begin by taking the " –closure of the start state q of our NFA and constructing a new start state S. in our DFA corresponding to that " -closure. Next, for each symbol in our alphabet, we record the set of NFA states that we can reach from S 0on that symbol. For each such set, we make a DFA state corresponding to its "E-closure, taking care to do this only once for each set. In the case two sets are equal, we simply reuse the existing DFA state that we already constructed. This process is then repeated for each of the new DFA states (that is, set of NFA states) until we run out of DFA states to process. Finally, every DFA state whose corresponding set of NFA states contains an accepting state is itself marked as an accepting state.

2.16 Creating a lexical analyzer with Lex



2.17 Lex specifications:

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

declarations
%%
translation rules
%%
auxiliary procedures

1. The *declarations* section includes declarations of variables,manifest constants(A manifest constant is an identifier that is declared to represent a constant e.g. *# define PIE 3.14*), and regular definitions.
2. The *translation rules* of a Lex program are statements of the form :
   p1 *{action 1}*

```
p2          {action 2}
p3          {action 3}
...            ...
...            ...
```

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

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

UNIT -3

SYNTAX ANALYSIS

3.1 ROLE OF THE PARSER

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

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

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

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

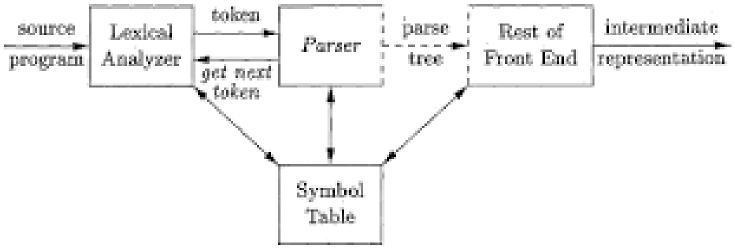


Figure 4.1: Position of parser in compiler model

3.2 TOP-DOWN PARSING

A program that performs syntax analysis is called a parser. A syntax analyzer takes tokens as input and output error message if the program syntax is wrong. The parser uses symbol-look-ahead and an approach called top-down parsing without backtracking. Top-downparsers check to see if a string can be generated by a grammar by creating a parse tree starting from the initial symbol and working down. Bottom-up parsers, however, check to see a string can be generated from a grammar by creating a parse tree from the leaves, and working up. Early parser generators such as YACC creates bottom-up parsers whereas many of Java parser generators such as JavaCC create top-down parsers.

3.3RECURSIVE DESCENT PARSING

Typically, top-down parsers are implemented as a set of recursive functions that descent through a parse tree for a string. This approach is known as recursive descent parsing, also known as LL(k) parsing where the first L stands for left-to-right, the second L stands for leftmost-derivation, and k indicates k-symbol lookahead. Therefore, a parser using the single symbol look-ahead method and top-down parsing without backtracking is called LL(1) parser. In the following sections, we will also use an extended BNF notation in which some regulation expression operators are to be incorporated.

A syntax expression defines sentences of the form , or . A syntax of the form defines sentences that consist of a sentence of the form followed by a sentence of the form followed by a sentence of the form . A syntax of the form defines zero or one occurrence of the form . A syntax of the form defines zero or more occurrences of the form .

A usual implementation of an LL(1) parser is:

- initialize its data structures,
- get the lookahead token by calling scanner routines, and
- call the routine that implements the start symbol.

Here is an example.

```
proc syntaxAnalysis()
begin
initialize(); // initialize global data and structures
nextToken(); // get the lookahead token
program(); // parser routine that implements the start symbol
end;
```

3.4 FIRST AND FOLLOW

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

1. If X is terminal, then FIRST(X) is {X}.

2. If X->e is a production, then add e to FIRST(X).

3. If X is nonterminal and X->Y1Y2...Yk is a production, then place a in FIRST(X) if for some i, a is in FIRST(Yi) and e is in all of FIRST(Y1),...,FIRST(Yi-1) that is,
Y1.......Yi-$_1$=*>e. If e is in FIRST(Yj) for all j=1,2,...,k, then add e to FIRST(X). For example, everything in FIRST(Yj) is surely in FIRST(X). If y1 does not derive e, then we add nothing more to FIRST(X), but if Y1=*>e, then we add FIRST(Y2) and so on.

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

1. Place $ in FOLLOW(S), where S is the start symbol and $ in the input right endmarker.
2. If there is a production A=>aBs where FIRST(s) except e is placed in FOLLOW(B).
3. If there is aproduction A->aB or a production A->aBs where FIRST(s) contains e, then everything in FOLLOW(A) is in FOLLOW(B).

Consider the following example to understand the concept of First and Follow.Find the first and follow of all nonterminals in the Grammar-

E -> TE'
E'-> +TE'|e
T -> FT'
T'-> *FT'|e
F -> (E)|id

Then:

FIRST(E)=FIRST(T)=FIRST(F)={(,id}
FIRST(E')={+,e}
FIRST(T')={*,e}
FOLLOW(E)=FOLLOW(E')={),$}
FOLLOW(T)=FOLLOW(T')={+,),$}
FOLLOW(F)={+,*,),$}

For example, id and left parenthesis are added to FIRST(F) by rule 3 in definition of FIRST with i=1 in each case, since FIRST(id)=(id) and FIRST('(')= {(} by rule 1. Then by rule 3 with i=1, the production T -> FT' implies that id and left parenthesis belong to FIRST(T) also.

To compute FOLLOW,we put $ in FOLLOW(E) by rule 1 for FOLLOW. By rule 2 applied toproduction F-> (E), right parenthesis is also in FOLLOW(E). By rule 3 applied to production E-> TE', $ and right parenthesis are in FOLLOW(E').

3.5 CONSTRUCTION OF PREDICTIVE PARSING TABLES

For any grammar G, the following algorithm can be used to construct the predictive parsing table. The algorithm is

Input : Grammar G

Output : Parsing table M

Method

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

3.6.LL(1) GRAMMAR

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

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

3.7.ERROR RECOVERY IN PREDICTIVE PARSING

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

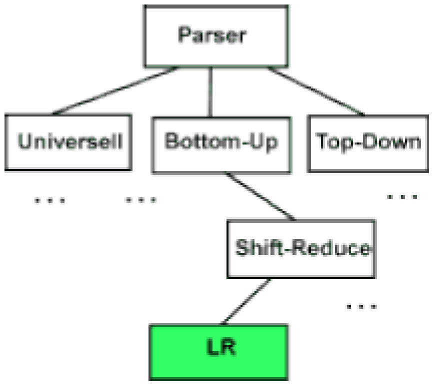
Panic-mode error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appears. Its effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur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

- As a starting point, we can place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.
- It is not enough to use FOLLOW(A) as the synchronizingset for A. Fo example , if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal generating expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchica structure on constructs in a language; e.g., expressions appear within statement, which appear within bblocks,and so on. We can add to the synchronizing set of a lower construct the symbols that begin higher constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generaitn expressions.
- If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

- If a nonterminal can generate the empty string, then the production deriving e can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.
- If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

3.8 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.



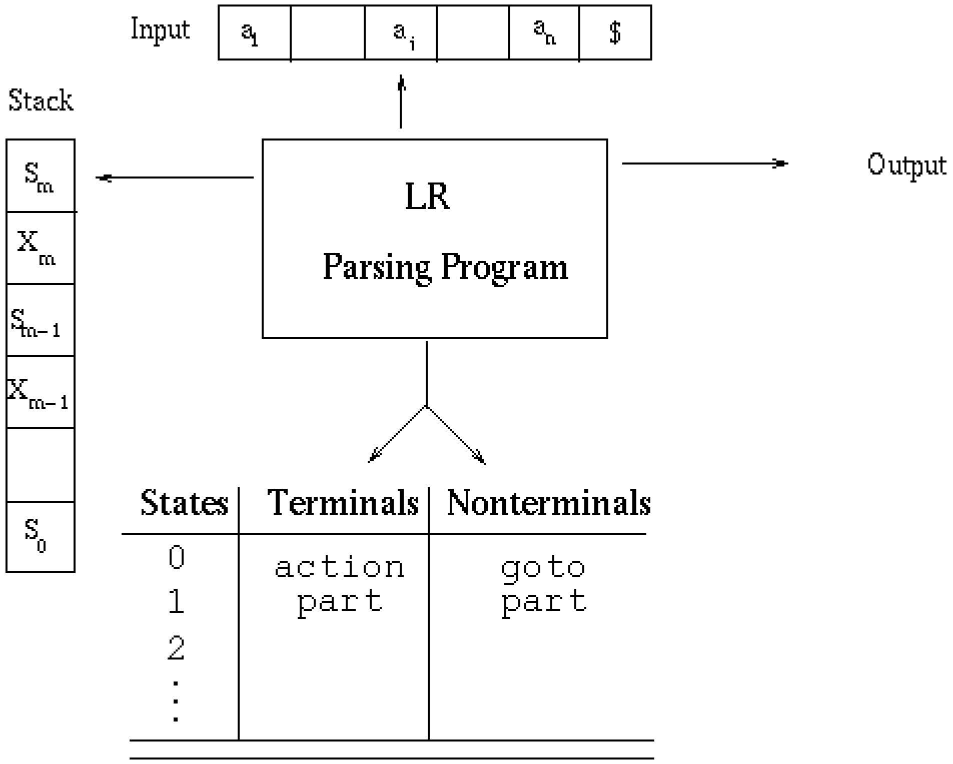
WHY LR PARSING:

- ✓ LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- ✓ The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- ✓ The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.
- ✓ An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to- right scan of the input.

The disadvantage is that it takes too much work to 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.

MODELS OF LR PARSERS

The schematic form of an LR parser is shown below.



LR Parser Table

The program uses a stack to store a string of the form s0X1s1X2...Xmsm where sm is on top. Each Xi is a grammar symbol and each si is a symbol representing a state. Each state symbol summarizes the information contained in the stack below it. The combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shiftreduce parsing decision. The parsing table consists of two parts: a parsing action function action and a goto function goto. The program driving the LR parser behaves as follows: It determines sm the state currently on top of the stack and ai the current input symbol. It then consults action[sm,ai], which can have one of four values:

- shift s, where s is a state
- reduce by a grammar production A -> b

- accept
- error

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

For a parsing table constructed for a grammar G, the goto table is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shiftreduce parser because they do not extend past the rightmost handle.

A configuration of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

(s0 X1 s1 X2 s2... Xm sm, ai ai+1... an$)

This configuration represents the right-sentential form

X1 X1 ... Xm ai ai+1 ...an

in essentially the same way a shift-reduce parser would; only the presence of the states on the stack is new. Recall the sample parse we did (see Example 1: Sample bottom-up parse) in which we assembled the right-sentential form by concatenating the remainder of the input buffer to the top of the stack. The next move of the parser is determined by reading ai and sm, and consulting the parsing action table entry action[sm, ai]. Note that we are just looking at the state here and no symbol below it. We'll see how this actually works later.

The configurations resulting after each of the four types of move are as follows:

If action[sm, ai] = shift s, the parser executes a shift move entering the configuration

(s0 X1 s1 X2 s2... Xm sm ai s, ai+1... an$)

Here the parser has shifted both the current input symbol ai and the next symbol.

If action[sm, ai] = reduce A -> b, then the parser executes a reduce move, entering the configuration,

(s0 X1 s1 X2 s2... Xm-r sm-r A s, ai ai+1... an$)

where s = goto[sm-r, A] and r is the length of b, the right side of the production. The parser first popped 2r symbols off the stack (r state symbols and r grammar symbols), exposing state sm-r. The parser then pushed both A, the left side of the production, and s, the entry for goto[sm-r, A], onto the stack. The current input symbol is not changed in a reduce move.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For example, we might just print out the production reduced.

If action[sm, ai] = accept, parsing is completed.

3.9 SHIFT REDUCE PARSING

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

During the operation of the parser, symbols from the input are shifted onto the stack. If a prefix of the symbols on top of the stack matches the RHS of a grammar rule which is the correct rule to use within the current context, then the parser reduces the RHS of the rule to its LHS,replacing the RHS symbols on top of the stack with the nonterminal occurring on the LHS of the rule. This shift-reduce process continues until the parser terminates, reporting either success or failure. It terminates with success when the input is legal and is accepted by the parser. It terminates with failure if an error is detected in the input. The parser is nothing but a stack automaton which may be in one of several discrete states. A state is usually represented simply as an integer. In reality, the parse stack contains states, rather than grammar symbols. However, since each state corresponds to a unique grammar symbol, the state stack can be mapped onto the grammar symbol stack mentioned earlier.

The operation of the parser is controlled by a couple of tables:

ACTION TABLE

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

Shift s'

Shift state s' onto the parse stack.

Reduce r

Reduce by rule r. This is explained in more detail below.

Accept

Terminate the parse with success, accepting the input.

Error

Signal a parse error

GOTO TABLE

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

The current state of a shift-reduce parser is the state on top of the state stack. The detailed operation of such a parser is as follows:

1. Initialize the parse stack to contain a single state s0, where s0 is the distinguished initial state of the parser.

2. Use the state s on top of the parse stack and the current lookahead t to consult the action table entry action[s][t]:

· If the action table entry is shift s' then push state s' onto the stack and advance the input so that the lookahead is set to the next token.

· If the action table entry is reduce r and rule r has m symbols in its RHS, then pop m symbols off the parse stack. Let s' be the state now revealed on top of the parse stack and N be the LHS nonterminal for rule r. Then consult the goto table and push the state given by goto[s'][N] onto the stack. The lookahead token is not changed by this step.

- If the action table entry is accept, then terminate the parse with success.
- If the action table entry is error, then signal an error.

3. Repeat step (2) until the parser terminates.

For example, consider the following simple grammar

0) $S: stmt <EOF>

1) stmt: ID ':=' expr

2) expr: expr '+' ID

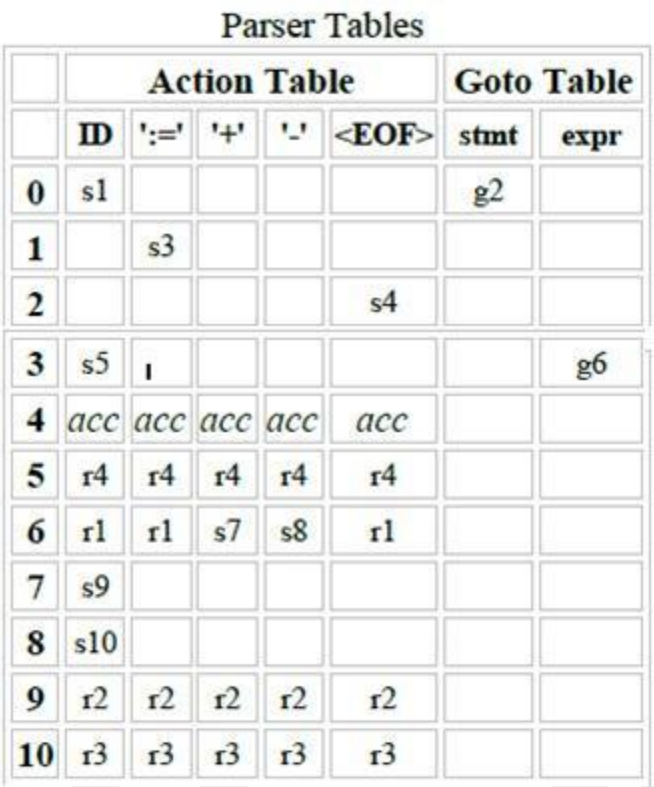
3) expr: expr '-' ID

4) expr: ID

which describes assignment statements like a:= b + c - d. (Rule 0 is a special augmenting production added to the grammar).

One possible set of shift-reduce parsing tables is shown below (sn denotes shift n, rn denotes reduce n, acc denotes accept and blank entries denote error entries):

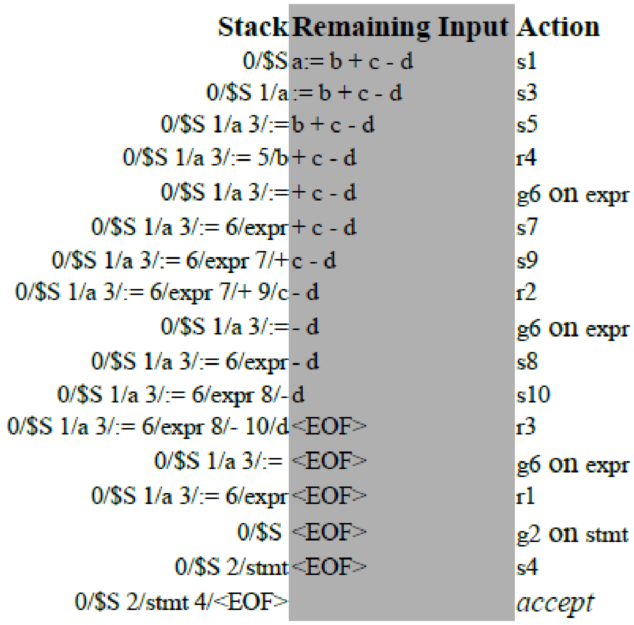
Parser Tables



Parser Tables

	Action Table					Goto Table	
	ID	':='	'+'	'-'	<EOF>	stmt	expr
0	s1					g2	
1		s3					
2					s4		
3	s5						g6
4	*acc*	*acc*	*acc*	*acc*	*acc*		
5	r4	r4	r4	r4	r4		
6	r1	r1	s7	s8	r1		
7	s9						
8	s10						
9	r2	r2	r2	r2	r2		
10	r3	r3	r3	r3	r3		

A trace of the parser on the input a:= b + c - d is shown below:



Stack	Remaining Input	Action
0/$S	a:= b + c - d	s1
0/$S 1/a	:= b + c - d	s3
0/$S 1/a 3/:=	b + c - d	s5
0/$S 1/a 3/:= 5/b	+ c - d	r4
0/$S 1/a 3/:=	+ c - d	g6 on expr
0/$S 1/a 3/:= 6/expr	+ c - d	s7
0/$S 1/a 3/:= 6/expr 7/+	c - d	s9
0/$S 1/a 3/:= 6/expr 7/+ 9/c	- d	r2
0/$S 1/a 3/:=	- d	g6 on expr
0/$S 1/a 3/:= 6/expr	- d	s8
0/$S 1/a 3/:= 6/expr 8/-	d	s10
0/$S 1/a 3/:= 6/expr 8/- 10/d	<EOF>	r3
0/$S 1/a 3/:=	<EOF>	g6 on expr
0/$S 1/a 3/:= 6/expr	<EOF>	r1
0/$S	<EOF>	g2 on stmt
0/$S 2/stmt	<EOF>	s4
0/$S 2/stmt 4/<EOF>		*accept*

Each stack entry is shown as a state number followed by the symbol which caused the transition to that state.

3.10 SLR PARSER

An *LR(0) item* (or just *item*) of a grammar *G* is a production of *G* with a dot at some position of the right side indicating how much of a production we have seen up to a given point.

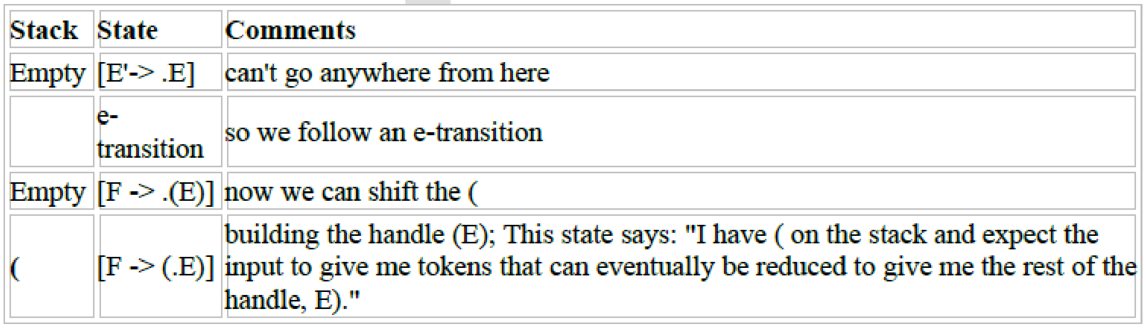
For example, for the production E -> E + T we would have the following items:

[E -> .E + T]

[E -> E. + T]

[E -> E +. T]

[E -> E + T.]



Stack	State	Comments
Empty	[E'-> .E]	can't go anywhere from here
	e-transition	so we follow an e-transition
Empty	[F -> .(E)]	now we can shift the (
(	[F -> (.E)]	building the handle (E); This state says: "I have (on the stack and expect the input to give me tokens that can eventually be reduced to give me the rest of the handle, E)."

CONSTRUCTING THE SLR PARSING TABLE

To construct the parser table we must convert our NFA into a DFA. The states in the LR table will be the e-closures of the states corresponding to the items SO...the process of creating the LR state table parallels the process of constructing an equivalent DFA from a machine with e-transitions. Been there, done that - this is essentially the subset construction algorithm so we are in familiar territory here.

We need two operations: closure()

and goto().

closure()

If I is a set of items for a grammar *G*, then closure(I) is the set of items constructed from I by the two rules: Initially every item in I is added to closure(I)

If *A* -> a.*B*b is in closure(I), and *B* -> g is a production, then add the initial item [*B* -> .g] to I, if it is not already there. Apply this rule until no more new items can be added to closure(I).

From our grammar above, if I is the set of one item {[E'-> .E]}, then closure(I) contains:

I0: E' -> .E

E -> .E + T

E -> .T

T -> .T * F

T -> .F

F -> .(E)

F -> .id

goto()

goto(I, *X*), where I is a set of items and X is a grammar symbol, is defined to be the closure of the set of all items [*A* -> a*X*.b] such that [*A* -> a.*X*b] is in I. The idea here is fairly intuitive: if I is the set of items that are valid for some viable prefix g, then goto(I, *X*) is the set of items that are valid for the viable prefix g*X*.

SETS-OF-ITEMS-CONSTRUCTION

To construct the canonical collection of sets of LR(0) items for

augmented grammar G'.

procedure items(G')

begin

```
C := {closure({[S' -> .S]})};
repeat
for each set of items in C and each grammar symbol X
such that goto(I, X) is not empty and not in C do
add goto(I, X) to C;
until no more sets of items can be added to C
end;
```

ALGORITHM FOR CONSTRUCTING AN SLR PARSING TABLE

Input: augmented grammar G'

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

Method:

Construct C = {I0, I1 , ..., In} the collection of sets of LR(0) items for G'.

State i is constructed from Ii:

if [A -> a.ab] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to "shift j". Here a must be a terminal.

if [A -> a.] is in Ii, then set action[i, a] to "reduce A -> a" for all a in FOLLOW(A). Here A may

not be S'.

if [S' -> S.] is in Ii, then set action[i, $] to "accept"

If any conflicting actions are generated by these rules, the grammar is not SLR(1) and the algorithm fails to produce a parser. The goto transitions for state i are constructed for all *nonterminal*s *A* using the rule: If goto(Ii, A)= Ij, then goto[i, *A*] = j.

All entries not defined by rules 2 and 3 are made "error".

The inital state of the parser is the one constructed from the set of items containing [*S*' -> .S].

Let's work an example to get a feel for what is going on,

An Example

(1) E -> E * B

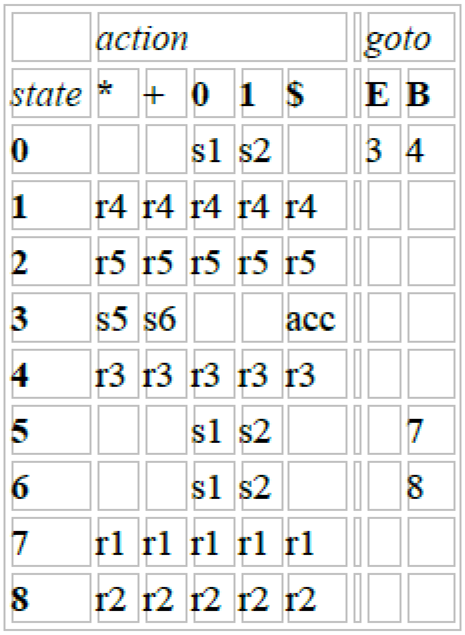
(2) E -> E + B

(3) E -> B

(4) B -> 0

(5) B -> 1

The Action and Goto Table The two LR(0) parsing tables for this grammar look as follows:



	action					*goto*	
state	*	+	**0**	**1**	**$**	**E**	**B**
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

3.11 LALR PARSER:

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

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

ALGORITHM FOR EASY CONSTRUCTION OF AN LALR TABLE

Input: G'

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

Method:

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

Consider the above example,

I3 & I6 can be replaced by their union

I36:C->c.C,c/d/$

C->.Cc,C/D/$

C->.d,c/d/$

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

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

Parsing Table

state	c	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5

36	S36	S47			89
47	R3	R3			
5			R1		
89	R2	R2	R2		

HANDLING ERRORS

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

3.12 LR ERROR RECOVERY

An LR parser will detect an error when it consults the parsing action table and find a blank or error entry. Errors are never detected by consulting the goto table. An LR parser will detect an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing the error. SLR and LALR parsers may make several reductions before detecting an error, but they will never shift an erroneous input symbol onto the stack.

3.12.1 PANIC-MODE ERROR RECOVERY

We can implement panic-mode error recovery by scanning down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow The situation might exist where there is more than one choice for the nonterminal A. Normally these would be nonterminals representing major program pieces, e.g. an expression, a statement, or a block. For example, if A is the nonterminal stmt, a might be semicolon or }, which marks the end of a statement sequence. This method of error recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from A contains an error. Part of that string has already been processed, and the result of this processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow A. By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that if has found an instance of A and resumes normal parsing.

3.12.2 PHRASE-LEVEL RECOVERY

Phrase-level recovery is implemented by examining each error entry in the LR action table and deciding on the basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbol would be modified in a way deemed appropriate for each error entry. In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer.

The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. We must make our choices so that the LR parser will not get into an infinite loop. A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached. Popping a stack state that covers a non terminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

UNIT IV- SYNTAX DIRECTEDTRANSLATION & RUN TIME ENVIRONMENT

SEMANTIC ANALYSIS

- Semantic Analysis computes additional information related to the meaning of the program once the syntactic structure is known.
- In typed languages as C, semantic analysis involves adding information to the symbol table and performing type checking.
- The information to be computed is beyond the capabilities of standard parsing techniques, therefore it is not regarded as syntax.
- As for Lexical and Syntax analysis, also for Semantic Analysis we need both a Representation Formalism and an Implementation Mechanism.
- As representation formalism this lecture illustrates what are called Syntax Directed Translations.

SYNTAX DIRECTED TRANSLATION

- The Principle of Syntax Directed Translation states that the meaning of an input sentence is related to its syntactic structure, i.e., to its Parse-Tree.
- By Syntax Directed Translations we indicate those formalisms for specifying translations for programming language constructs guided by context-free grammars.
  - We associate Attributes to the grammar symbols representing the language constructs.
  - Values for attributes are computed by Semantic Rules associated with grammar productions.
- Evaluation of Semantic Rules may:
  - Generate Code;
  - Insert information into the Symbol Table;
  - Perform Semantic Check;
  - Issue error messages;
  - etc.

There are two notations for attaching semantic rules:

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

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

Syntax Directed Definitions

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

1. Grammar symbols have an associated set of **Attributes**;

2. Productions are associated with **Semantic Rules** for computing the values of attributes.

- Such formalism generates **Annotated Parse-Trees** where each node of the tree is a record with a field for each attribute (e.g.,X.a indicates the attribute a of the grammar symbol X).
- The value of an attribute of a grammar symbol at a given parse-tree node is defined by a semantic rule associated with the production used at that node.

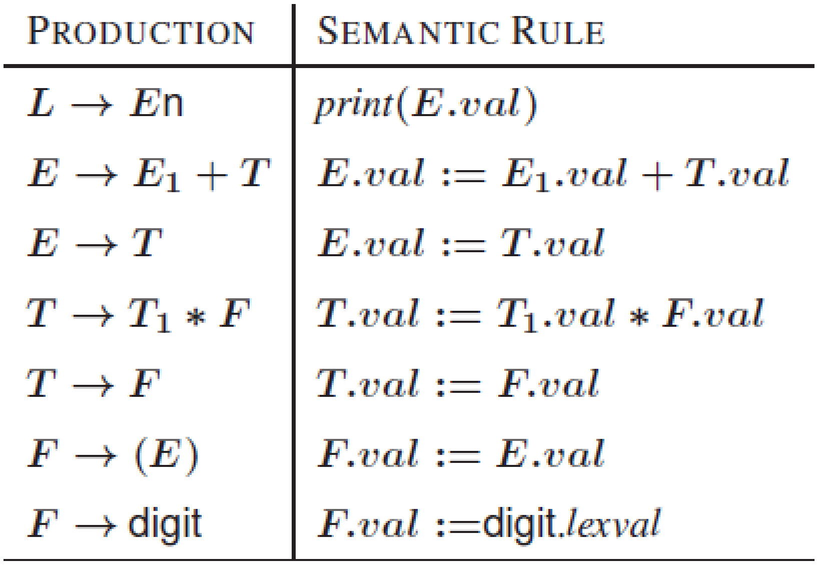
We distinguish between two kinds of attributes:

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

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

Syntax Directed Definitions: An Example

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

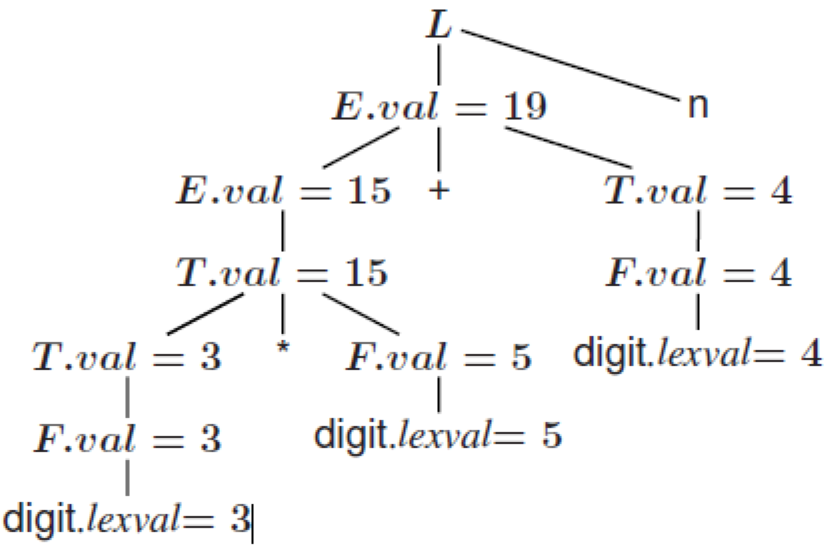


PRODUCTION	SEMANTIC RULE
$L \rightarrow E\mathsf{n}$	$print(E.val)$
$E \rightarrow E_1 + T$	$E.val := E_1.val + T.val$
$E \rightarrow T$	$E.val := T.val$
$T \rightarrow T_1 * F$	$T.val := T_1.val * F.val$
$T \rightarrow F$	$T.val := F.val$
$F \rightarrow (E)$	$F.val := E.val$
$F \rightarrow \mathsf{digit}$	$F.val := \mathsf{digit}.lexval$

S-ATTRIBUTED DEFINITIONS

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

- **Evaluation Order.** Semantic rules in a S-Attributed Definition can be evaluated by a bottom-up, or PostOrder, traversal of the parse-tree.
- **Example.** The above arithmetic grammar is an example of an S-Attributed Definition. The annotated parse-tree for the input 3*5+4n is:



L-attributed definition

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

1. attributes of X1;X2; : : : ;Xi1 (symbols to the left of Xi in the RHS)

2. inherited attributes of A.

Restrictions for translation schemes:

1. Inherited attribute of Xi must be computed by an action before Xi.

2. An action must not refer to synthesized attribute of any symbol to the right of that action.

3. Synthesized attribute for A can only be computed after all attributes it references have been completed (usually at end of RHS).

SYMBOL TABLES

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*(log2 *n*)
- Insertion into a sorted array is expensive – *O*(*n*) on average
- Useful when set of names is known in advance – table of reserved words
- Binary Search Tree
- Can grow dynamically
- Insertion and lookup are *O*(log2 *n*) on average

RUNTIME ENVIRONMENT

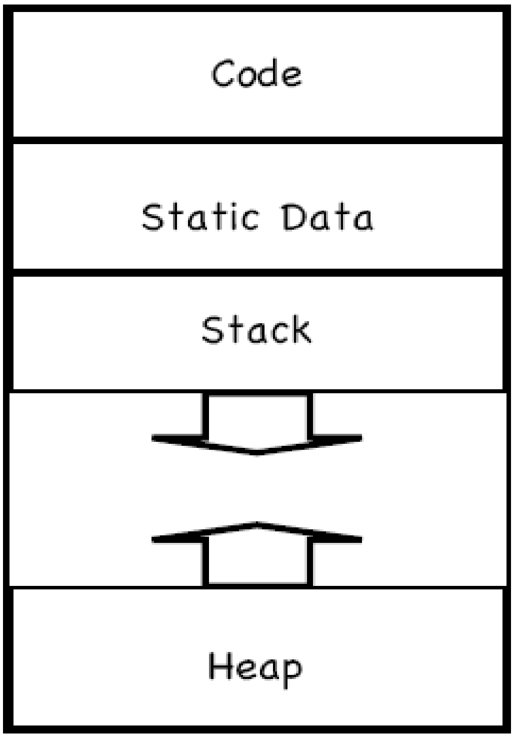
- Runtime organization of different storage locations
- Representation of scopes and extents during program execution.
- Components of executing program reside in blocks of memory (supplied by OS).
- Three kinds of entities that need to be managed at runtime:
    - Generated code for various procedures and programs.
- forms text or code segment of your program: size known at compile time.
    - Data objects:
- Global variables/constants: size known at compile time
- Variables declared within procedures/blocks: size known
- Variables created dynamically: size unknown.
    - Stack to keep track of procedure
- activations. Subdivide memory conceptually into code and data areas:
        - Code:

Program • instructions

        - Stack: Manage activation of procedures at runtime.
        - Heap: holds variables created dynamically

STORAGE ORGANIZATION

1. *Fixed-size objects can be placed in predefined locations.*



2. Run-time stack and heap The STACK is used to store:

- Procedure activations.
- The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
- The HEAP stores data allocated under program control (e.g. by malloc() in C).

Activation records

Any information needed for a single activation of a procedure is stored in the ACTIVATION RECORD (sometimes called the STACK FRAME). Today, we'll assume the stack grows DOWNWARD, as on, e.g., the Intel architecture. The activation record gets pushed for each procedure call and popped for each procedure return.

STATIC ALLOCATION

Statically allocated names are bound to storage at compile time. Storage bindings of statically allocated names never change, so even if a name is local to a procedure, its name is always bound to the same storage. The compiler uses the type of a name (retrieved from the symbol table) to determine storage size required. The required number of bytes (possibly aligned) is set aside for the name.The address of the storage is fixed at compile time.

Limitations:

- The size required must be known at compile time.
- Recursive procedures cannot be implemented as all locals are statically allocated.
- No data structure can be created dynamically as all data's static.

```
float f(int k)
{
float c[10],b;
b = c[k]*3.14;
return b;
}
```

Return value	offset = 0
Parameter k	offset = 4
Local c[10]	offset = 8
Local b	offset = 48

❖Stack-dynamic allocation

- ✓ Storage is organized as a stack.
- ✓ Activation records are pushed and popped.
- ✓ Locals and parameters are contained in the activation records for the call.
- ✓ This means locals are bound to fresh storage on every call.
- ✓ If we have a stack growing downwards, we just need a stack_top pointer.
- ✓ To allocate a new activation record, we just increase stack_top.
- ✓ To deallocate an existing activation record, we just decrease stack_top.

❖Address generation in stack allocation

The position of the activation record on the stack cannot be determined statically. Therefore the compiler must generate addresses RELATIVE to the activation record. If we have a downward-growing stack and a stack_top pointer, we generate addresses of the form stack_top + offset

HEAP ALLOCATION

Some languages do not have tree-structured allocations. In these cases, activations have to be allocated on the heap. This allows strange situations, like callee activations that live longer than their callers' activations. This is not common Heap is used for allocating space for objects created at run timeFor example: nodes of dynamic data structures such as linked lists and trees

Dynamic memory allocation and deallocation based on the requirements of the program*malloc()* and *free()* in C programs

*new()*and *delete()*in C++ programs

*new()*and garbage collection in Java programs

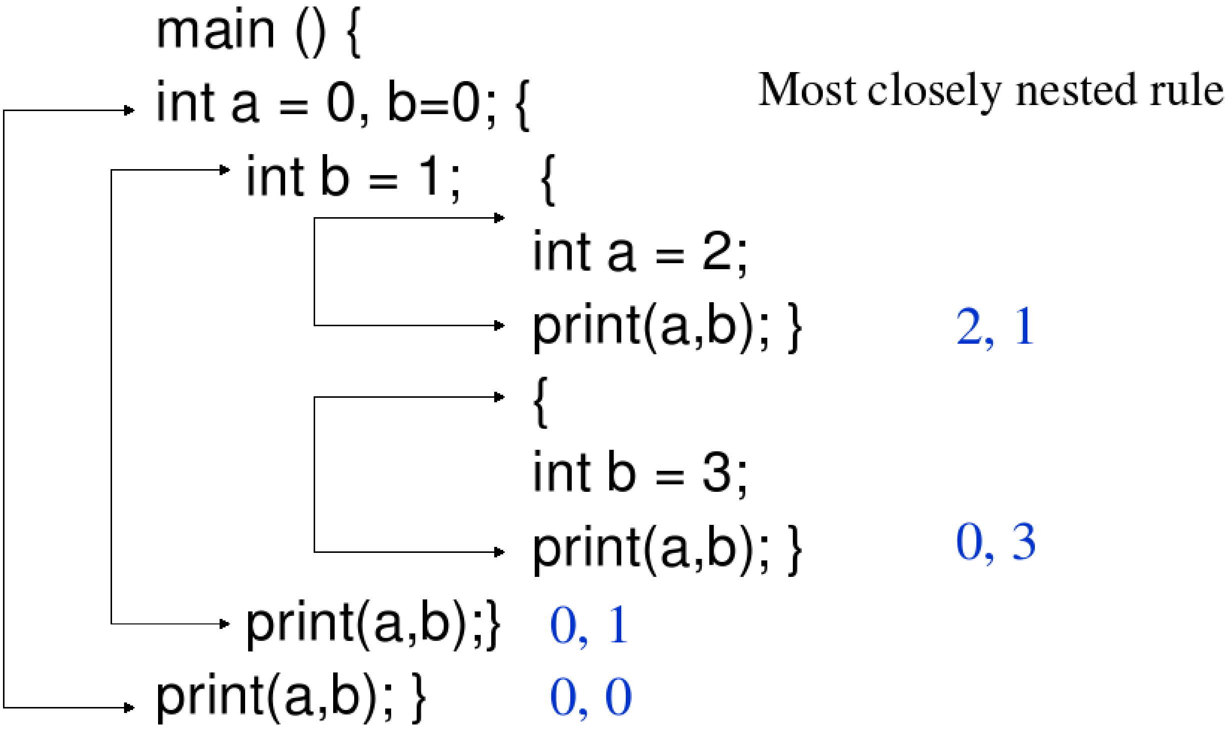
Allocation and deallocation may be *completely manual* (C/C++), *semi-automatic*(Java), or *fully automatic* (Lisp)

PARAMETERS PASSING

A language has first-class functionsif functions can bedeclared within any scope passed as arguments to other functions returned as results of functions.In a language with first-class functions and static scope, a function value is generally represented by a closure. a pair consisting of a pointer to function code a pointer to an activation record.Passing functions as arguments is very useful in structuring of systems using upcalls

An example:

```
main()
{ int
x =
4;
int f
(int
y) {
retur
n
x*y;
}
int g (int →int h){
int x = 7;
return h(3) + x;
}
```



Most closely nested rule

```
main () {
int a = 0, b=0; {
    int b = 1;    {
                  int a = 2;
                  print(a,b); }        2, 1
                  {
                  int b = 3;
                  print(a,b); }        0, 3
    print(a,b);}  0, 1
print(a,b); }     0, 0
```

Call-by-Value

The actual parameters are evaluated and their r-values are passed to the called procedure

A procedure called by value can affect its caller either through nonlocal names or through pointers.

Parameters in C are always passed by value. Array is unusual, what is passed by value is a pointer.

Pascal uses pass by value by default, but var parameters are passed by reference.

Call-by-Reference

Also known as call-by-address or call-by-location. The caller passes to the called procedure the l-value of the parameter.

If the parameter is an expression, then the expression is evaluated in a new location, and the address of the new location is passed.

Parameters in Fortran are passed by reference an old implementation bug in Fortran

```
func(a,b) { a = b};
call func(3,4); print(3);
```

Copy-Restore

A hybrid between call-by-value and call-by reference.

The actual parameters are evaluated and their r-values are passed as in call- by-value. In addition, l values are determined before the call.

When control returns, the current r-values of the formal parameters are copied back into the l-values of the actual parameters.

Call-by-Name

The actual parameters literally substituted for the formals. This is like a macro-expansion or in-line expansion Call-by-name is not used in practice. However, the conceptually related technique of in-line expansion is commonly used. In-lining may be one of the most effective optimization transformations if they are guided by execution profiles.

UNIT V - CODE OPTIMIZATION AND CODE GENERATION

INTRODUCTION

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

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

Machine independent optimizations:

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

Machine dependant optimizations:

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

The criteria for code improvement transformations:

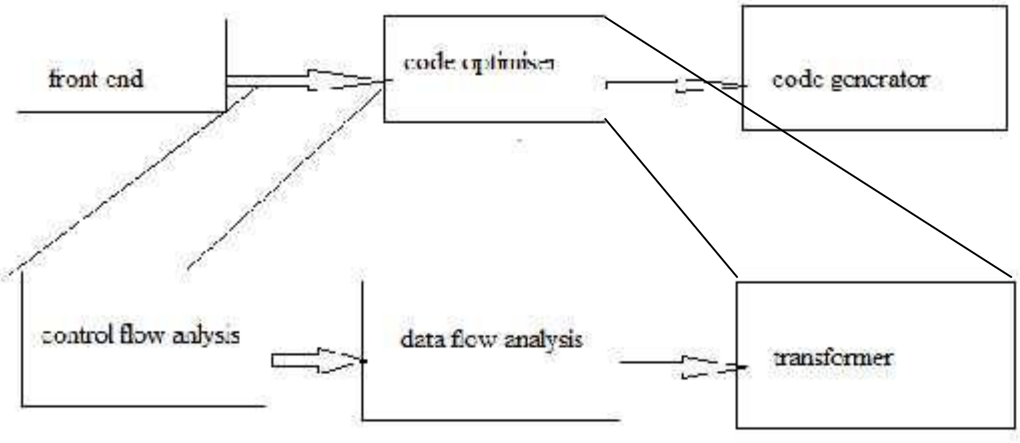
- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.

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

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

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

Organization for an Optimizing Compiler:



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

PRINCIPAL SOURCES OF OPTIMISATION

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

<u>Function-Preserving Transformations</u>

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

  ✓ Common sub expression elimination,
  ✓ Copy propagation,
  ✓ Dead-code elimination, and
  ✓ Constant folding

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

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

➢ Common Sub expressions elimination:

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

t_1: = 4*i
t_2: = a [t1]
t_3: = 4*j
t_4: = 4*i
t_5: = n
t_6: = b [t_4] +t_5

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

t_1: = 4*i
t_2: = a [t_1]
t_3: = 4*j
t_5: = n
t_6: = b [t_1] +t_5

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

➢ Copy Propagation:

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

```
x=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.

Example:

```
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:

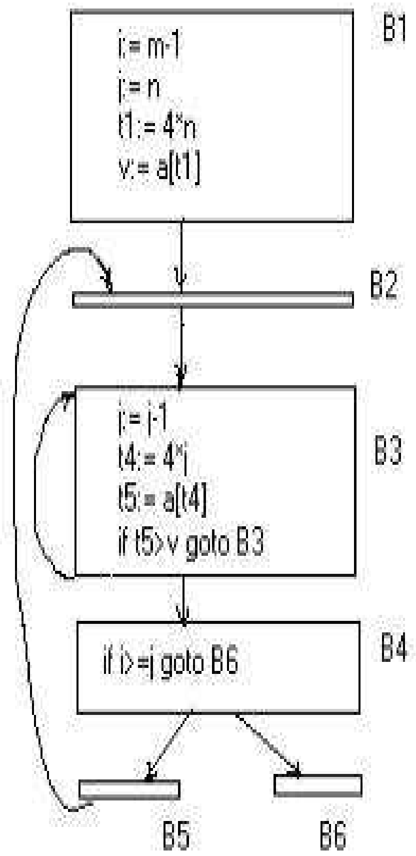
```
while (i <= limit-2)     /* statement does not change limit*/
```

Code motion will result in the equivalent of

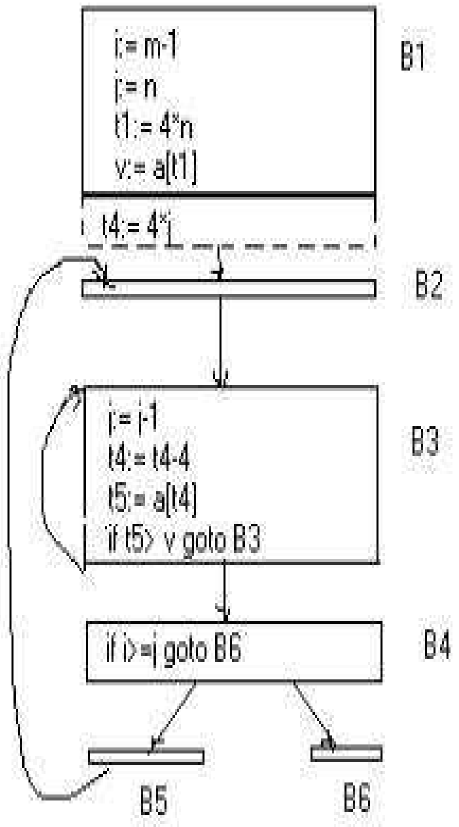
t= limit-2;
while (i<=t) /* statement does not change limit or t */

Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because 4*j is assigned to t_4. 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 t_4 completely; t_4 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 t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement j:=j-1 the relationship t_4:= 4*j-4 must hold. We may therefore replace the assignment t_4:= 4*j by t_4:= t_4-4. 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 t_4=4*j on entry to the block B3, we place an initializations of t_4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in second Fig.



before



after

- 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.

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

t_1:=b+c

t_2:=x+y

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

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression 2*3.14 would be replaced by 6.28.
- The relational operators <=, >=, <, >, + 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:=y**2 can be replaced by a cheaper statement x:=y*y

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

LOOPS IN FLOW GRAPH

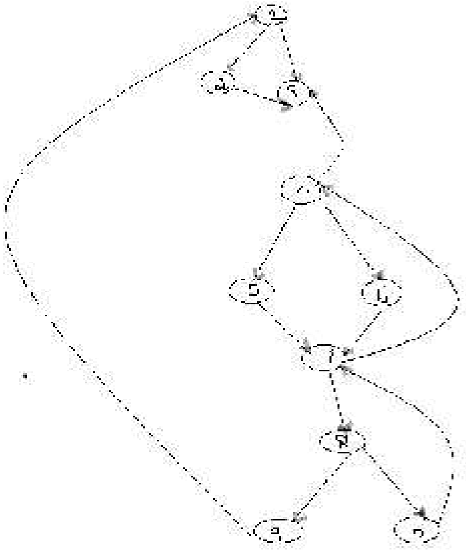
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

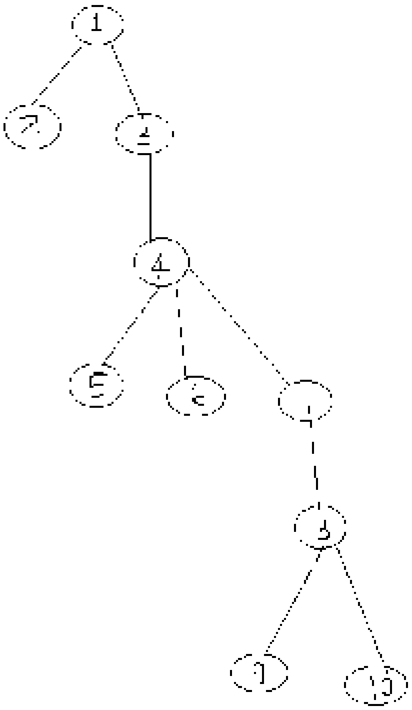
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates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

Example:

*In the flow graph below,
*Initial node,node1 dominates every node.
*node 2 dominates itself
*node 3 dominates all but 1 and 2.
*node 4 dominates all but 1,2 and 3.
*node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through the other.
*node 7 dominates 7,8 ,9 and 10.
*node 8 dominates 8,9 and 10.
*node 9 and 10 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 way to iterate the loop(i.e.)at least one path back to the header.

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

    - ✓ Example:

In the above graph,

7 → 4	4 DOM 7
10 →7	7 DOM 10
4 → 3	
8 → 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;
```

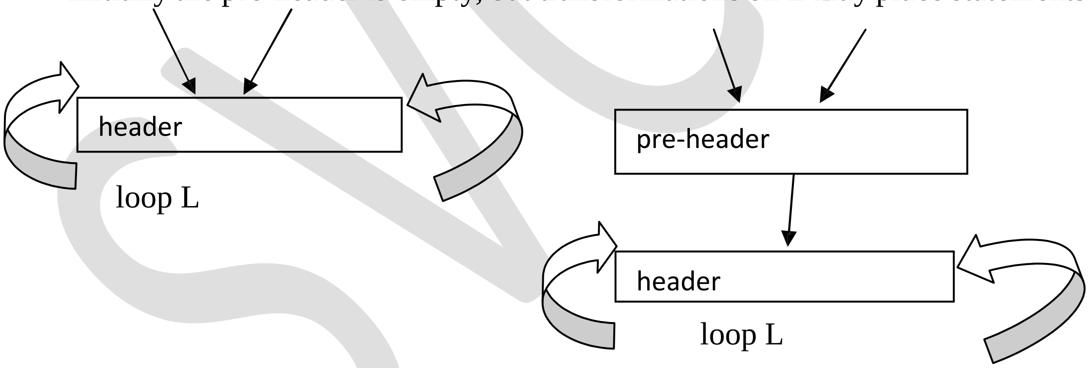
```
loop : = {d};
insert(n);
while stack is not empty do begin
        pop m, the first element of stack, off stack;
        for each predecessor p of m do insert(p)
end
```

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 L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



(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 R_0,a

(2) MOV a,R_0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of **a** is already in register R_0.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:

```
If debug =1 goto L2

  goto L2

L1: print debugging information

L2:                                ………………………….(a)
```

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

```
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

```
     If a < b  goto  L2

     ….

     L1: goto L2
```

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

```
    goto L1

     ……..
     L1: if a < b goto L2

     L3:             …………………………………..(1)
```

- May be replaced by

```
  If a < b goto L2

  goto L3

   …….

   L3:              ………………………………….(2)
```

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

Algebraic Simplification:

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

  x := x+0

  Or

  x := x * 1

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

Reduction in Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

  $X^2 \rightarrow X*X$

Use of Machine Idioms:

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

i:=i+1 → i++
i:=i-1 → i- -

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.

- A compiler could take advantage of "reaching definitions" , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.

- Data-flow information can be collected by setting up and solving systems of equations of the form :

$$out\ [S] = gen\ [S]\ U\ (\ in\ [S] - kill\ [S]\ )$$

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

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

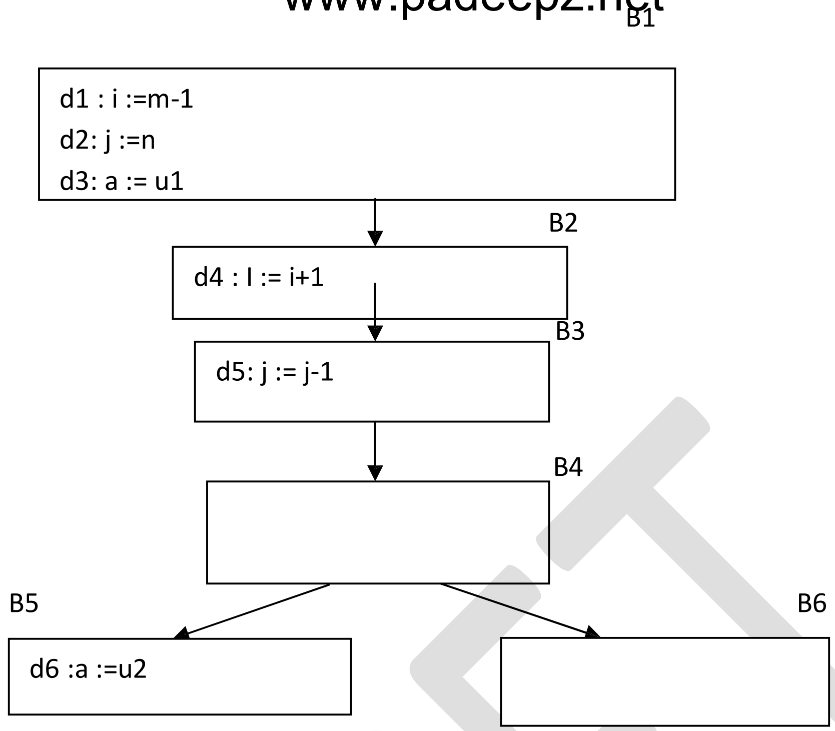
✓ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms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

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

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

Points and Paths:

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



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

✓ P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or

✓ P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

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

- These statements certainly define a value for x, and they are referred to as **unambiguous** definitions of x. There are certain kinds of statements that may define a value for x; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:

✓ A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.

✓ An assignment through a pointer that could refer to x. For example, the assignment *q: = y is a definition of x if it is possible that q points to x. we must assume that an assignment through a pointer is a definition of every variable.

- We say a definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not “killed” along that path. Thus a point can be reached by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

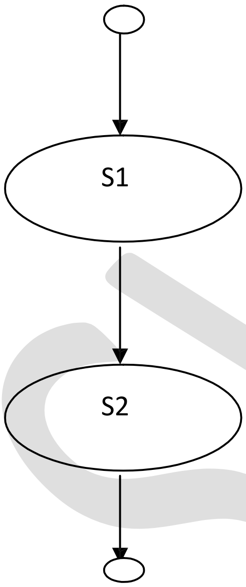
Data-flow analysis of structured programs:

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

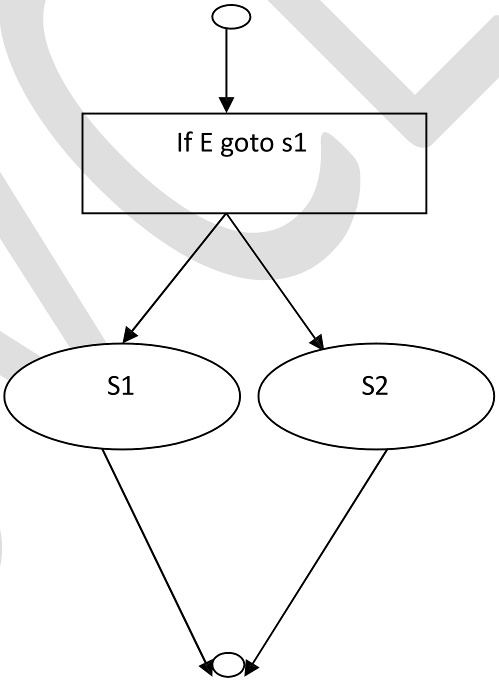
  S → id: = E| S; S | if E then S else S | do S while E

  E → id + id| id

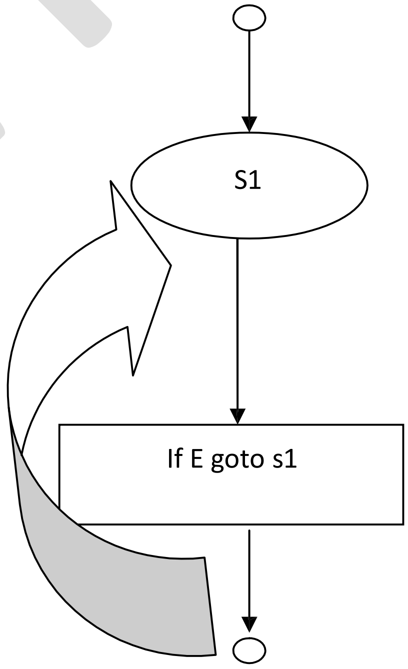
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



S1 ; S2



IF E then S1 else S2

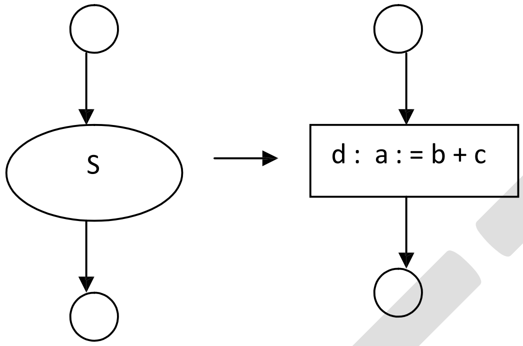


do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

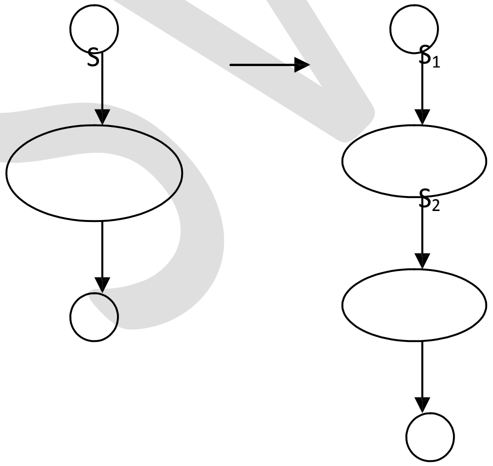
- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets in[S], out[S], gen[S], and kill[S] for all statements S.
- **gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end of S.**
- Consider the following data-flow equations for reaching definitions :

i)



gen [S] = { d }
kill [S] = D_a – { d }
out [S] = gen [S] U (in[S] – kill[S])

- Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. Thus
  Gen[S]={d}
- On the other hand, d "kills" all other definitions of a, so we write
  Kill[S] = D_a – {d}
  Where, D_a is the set of all definitions in the program for variable a. ii)



gen[S]=gen[S_2] U (gen[S_1]-kill[S_2])
Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

in [S_1] = in [S]
in [S_2] = out [S_1]
out [S] = out [S_2]

- Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by S_2, then it is surely generated by S. if d is generated by S_1, it will reach the end of S provided it is not killed by S_2. Thus, we write
  gen[S]=gen[S_2] U (gen[S_1]-kill[S_2]
- Similar reasoning applies to the killing of a definition, so we have
  Kill[S] = kill[S_2] U (kill[S_1] – gen[S_2])

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill. We have made the assumption that the conditional expression E in the if and do statements are “uninterpreted”; that is, there exists inputs to the program that make their branches go either way.

- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.

- When we compare the computed gen with the “true” gen we discover that the true gen is always a subset of the computed gen. on the other hand, the true kill is always a superset of the computed kill.

- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.

- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.

- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill. It can be used, for example, to determine loop-invariant computations.

- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in. we intend that in[S] be the set of definitions reaching the beginning of

S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set out[S] is defined similarly for the end of s. it is important to note the distinction between out[S] and gen[S]. The latter is the set of definitions that reach the end of S without following paths outside S.
- Assuming we know in[S] we compute out by equation, that is

Out[S] = gen[S] U (in[S] - kill[S])

- Considering cascade of two statements S_1; S_2, as in the second case. We start by observing in[S_1]=in[S]. Then, we recursively compute out[S_1], which gives us in[S_2], since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1. Now we can compute out[S_2], and this set is equal to out[S].
- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

In[S_1] = in[S_2] = in[S]

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

Out[S]=out[S_1] U out[S_2]

Representation of sets:

- Sets of definitions, such as gen[S] and kill[S], can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming

languages. The difference A-B of sets A and B can be implemented by taking the complement of B and then using logical and to compute A

.

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.

- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as" use-definition chains" or "ud-chains", which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a, then ud-chain for that use of a is the set of definitions in in[B] that are definitions of a.in addition, if there are ambiguous definitions of a ,then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a.

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.

- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.

- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.

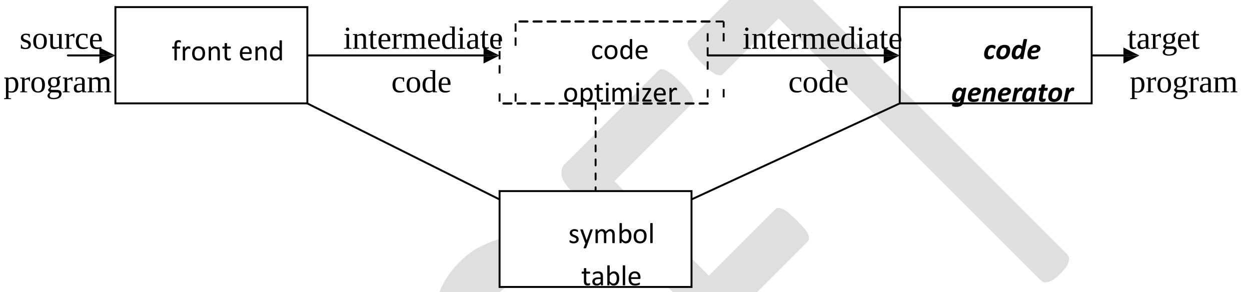
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval -based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.

CODE GENERATION

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

Position of code generator



ISSUES IN THE DESIGN OF A CODE GENERATOR

The following issues arise during the code generation phase :

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

1. Input to code generator:

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

2. Target program:

- The output of the code generator is the target program. The output may be :
  a. Absolute machine language
     - It can be placed in a fixed memory location and can be executed immediately.

b. Relocatable machine language

- It allows subprograms to be compiled separately.

c. Assembly language

- Code generation is made easier.

3. Memory management:

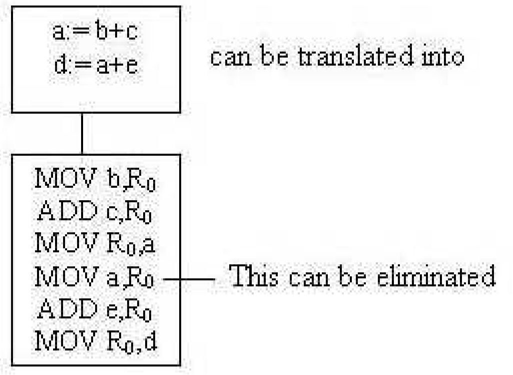
- Names in the source program are mapped to addresses of data objects in run-time memory by the front end and code generator.
- It makes use of symbol table, that is, a name in a three-address statement refers to a symbol-table entry for the name.
- Labels in three-address statements have to be converted to addresses of instructions. For example,

  j : **goto** *i* generates jump instruction as follows :

  - if *i* < *j*, a backward jump instruction with target address equal to location of code for quadruple *i* is generated.
  - if *i* > *j*, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

4. Instruction selection:

- The instructions of target machine should be complete and uniform.
- Instruction speeds and machine idioms are important factors when efficiency of target program is considered.
- The quality of the generated code is determined by its speed and size.
- The former statement can be translated into the latter statement as shown below:



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

```
MOV b,R0
ADD c,R0
MOV R0,a
MOV a,R0
ADD e,R0
MOV R0,d
```

5. Register allocation

- Instructions involving register operands are shorter and faster than those involving operands in memory.
- The use of registers is subdivided into two subproblems :
  - ***Register allocation*** – the set of variables that will reside in registers at a point in the program is selected.

- ***Register assignment*** – the specific register that a variable will reside in is picked

- Certain machine requires even-odd *register pairs* for some operands and results. For example , consider the division instruction of the form :

  D x, y

  where, x – dividend even register in even/odd register pair
  y – divisor
  even register holds the remainder
  odd register holds the quotient

6. Evaluation order

- The order in which the computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others.

TARGET MACHINE

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n* general-purpose registers, R_0, R_1, . . . , R_{n-1}.
- It has two-address instructions of the form:

  op source, destination

  where, *op* is an op-code, and *source* and *destination* are data fields.

- It has the following op-codes :

  MOV (move *source* to *destination*)
  ADD (add *source* to *destination*)
  SUB (subtract *source* from *destination*)

- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.

Address modes with their assembly-language forms

MODE	**FORM**	***ADDRESS***	**ADDED COST**
absolute	M	M	1
register	R	R	0
indexed	*c*(R)	*c*+*contents*(R)	1
indirect register	*R	*contents* (R)	0
indirect indexed	**c*(R)	*contents*(*c*+ *contents*(R))	1
literal	#*c*	*c*	1

- For example : MOV R_0, M stores contents of Register R_0 into memory location M ;
  MOV 4(R_0), M stores the value *contents*(4+*contents*(R_0)) into M.

Instruction costs :

- Instruction cost = 1+cost for source and destination address modes. This cost corresponds to the length of the instruction.
- Address modes involving registers have cost zero.
- Address modes involving memory location or literal have cost one.
- Instruction length should be minimized if space is important. Doing so also minimizes the time taken to fetch and perform the instruction.
  For example : MOV R0, R1 copies the contents of register R0 into R1. It has cost one, since it occupies only one word of memory.
- The three-address statement **a : = b + c** can be implemented by many different instruction sequences :

```
i) MOV b, R0
   ADD c, R0              cost = 6
   MOV R0, a

ii) MOV b, a
    ADD c, a              cost = 6
```

iii) Assuming R_0, R_1 and R_2 contain the addresses of a, b, and c :

```
      MOV *R1, *R0
      ADD *R2, *R0        cost = 2
```

- In order to generate good code for target machine, we must utilize its addressing capabilities efficiently.

RUN-TIME STORAGE MANAGEMENT

- Information needed during an execution of a procedure is kept in a block of storage called an activation record, which includes storage for names local to the procedure.
- The two standard storage allocation strategies are:
  1. Static allocation
  2. Stack allocation
- In static allocation, the position of an activation record in memory is fixed at compile time.
- In stack allocation, a new activation record is pushed onto the stack for each execution of a procedure. The record is popped when the activation ends.
- The following three-address statements are associated with the run-time allocation and deallocation of activation records:
  1. Call,
  2. Return,
  3. Halt, and
  4. Action, a placeholder for other statements.
- We assume that the run-time memory is divided into areas for:
  1. Code
  2. Static data
  3. Stack

Static allocation

Implementation of call statement:

The codes needed to implement static allocation are as follows:

MOV *#here* + 20, *callee.static_area* /*It saves return address*/

GOTO *callee.code_area* /*It transfers control to the target code for the called procedure */

where,
callee.static_area – Address of the activation record
callee.code_area – Address of the first instruction for called procedure
#here + 20 – Literal return address which is the address of the instruction following GOTO.

Implementation of return statement:

A return from procedure *callee* is implemented by :
GOTO **callee.static_area*

This transfers control to the address saved at the beginning of the activation record.

Implementation of action statement:

The instruction ACTION is used to implement action statement.

Implementation of halt statement:

The statement HALT is the final instruction that returns control to the operating system.

Stack allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, the position of activation record is stored in register so words in activation records can be accessed as offsets from the value in this register.

The codes needed to implement stack allocation are as follows:

Initialization of stack:

MOV *#stackstart* , SP /* initializes stack */

Code for the first procedure

HALT /* terminate execution */

Implementation of Call statement:

ADD *#caller.recordsize*, SP /* increment stack pointer */

MOV *#here* + 16, *SP /*Save return address */

GOTO *callee.code_area*

where,
caller.recordsize – size of the activation record
#here + 16 – address of the instruction following the **GOTO**

Implementation of Return statement:

GOTO *0 (SP) /*return to the caller */

SUB *#caller.recordsize,* SP /* decrement SP and restore to previous value */

BASIC BLOCKS AND FLOW GRAPHS

Basic Blocks

- A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching except at the end.
- The following sequence of three-address statements forms a basic block:
  t_1 : = a * a
  t_2 : = a * b
  t_3 : = 2 * t_2
  t_4 : = t_1 + t_3
  t_5 : = b * b
  t_6 : = t_4 + t_5

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block

Method:

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the target of a conditional or unconditional goto is a leader.
   c. Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

- Consider the following source code for dot product of two vectors a and b of length 20

```
begin
        prod :=0;
        i:=1;
        do begin
                prod :=prod+ a[i] * b[i];
                i :=i+1;
        end
        while i <= 20
end
```

- The three-address code for the above source program is given as :

```
(1)     prod := 0
(2)     i := 1
(3)     t1 := 4* i
(4)     t2 := a[t1]         /*compute a[i] */
(5)     t3 := 4* i
(6)     t4 :=  b[t3]        /*compute b[i] */
(7)     t5 := t2*t4
(8)     t6 := prod+t5
(9)     prod := t6
(10)    t7 := i+1
(11)    i := t7
(12)    if i<=20 goto (3)
```

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (12)

Transformations on Basic Blocks:

A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Two important classes of transformation are :

- Structure-preserving transformations
- Algebraic transformations

1. Structure preserving transformations:

a) Common subexpression elimination:

$$\begin{array}{lcl} a : = b + c & & a : = b + c \\ b : = a - d & \longrightarrow & b : = a - d \\ c : = b + c & & c : = b + c \\ d : = a - d & & d : = b \end{array}$$

Since the second and fourth expressions compute the same expression, the basic block can be transformed as above.

b) Dead-code elimination:

Suppose *x* is dead, that is, never subsequently used, at the point where the statement x : = y + z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

c) Renaming temporary variables:

A statement **t : = b + c** (t is a temporary) can be changed to **u : = b + c** (u is a new temporary) and all uses of this instance of **t** can be changed to **u** without changing the value of the basic block.
Such a block is called a *normal-form block.*

d) Interchange of statements:

Suppose a block has the following two adjacent statements:

t1 : = b + c
t2 : = x + y

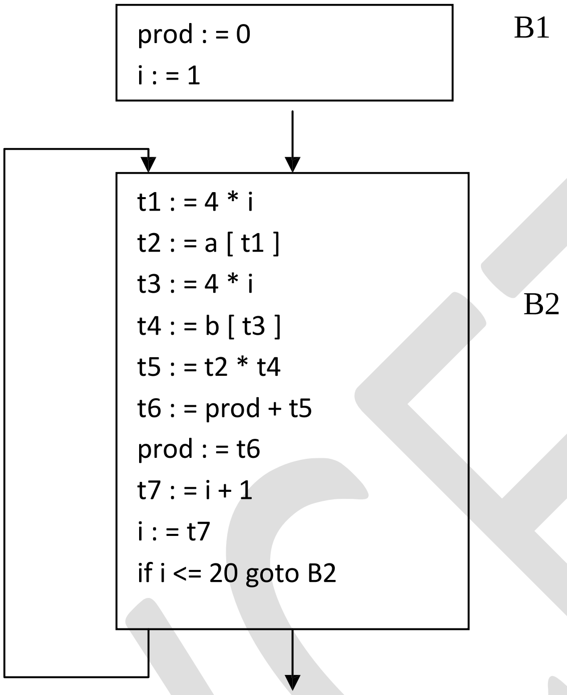
We can interchange the two statements without affecting the value of the block if and only if neither **x** nor **y** is **t_1** and neither **b** nor **c** is **t_2**.

2. Algebraic transformations:

Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.
Examples:
i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of expressions it computes.
ii) The exponential statement x : = y * * 2 can be replaced by x : = y * y.

Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:



```
prod : = 0
i : = 1
```

```
t1 : = 4 * i
t2 : = a [ t1 ]
t3 : = 4 * i
t4 : = b [ t3 ]
t5 : = t2 * t4
t6 : = prod + t5
prod : = t6
t7 : = i + 1
i : = t7
if i <= 20 goto B2
```

- B$_1$ is the *initial* node. B$_2$ immediately follows B$_1$, so there is an edge from B$_1$ to B$_2$. The target of jump from last statement of B$_1$ is the first statement B$_2$, so there is an edge from B$_1$ (last statement) to B$_2$ (first statement).
- B$_1$ is the *predecessor* of B$_2$, and B$_2$ is a *successor* of B$_1$.

Loops

- A loop is a collection of nodes in a flow graph such that
  1. All nodes in the collection are *strongly connected*.
  2. The collection of nodes has a unique *entry*.
- A loop that contains no other loops is called an inner loop.

NEXT-USE INFORMATION

- If the name in a register is no longer needed, then we remove the name from the register and the register can be used to store some other names.

Input: Basic block B of three-address statements

Output: At each statement i: x= y op z, we attach to i the liveliness and next-uses of x, y and z.

Method: We start at the last statement of B and scan backwards.

1. Attach to statement i the information currently found in the symbol table regarding the next-use and liveliness of x, y and z.
2. In the symbol table, set x to "not live" and "no next use".
3. In the symbol table, set y and z to "live", and next-uses of y and z to i.

Symbol Table:

Names	**Liveliness**	**Next-use**
x	not live	no next-use
y	Live	i
z	Live	i

A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

- For example: consider the three-address statement **a := b+c**
It can have the following sequence of codes:

ADD R_j, R_i Cost = 1 // if R_i contains b and R_j contains c

(or)

ADD c, R_i Cost = 2 // if c is in a memory location

(or)

MOV c, R_j Cost = 3 // move c from memory to Rj and add

ADD R_j, R_i

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z**,** perform the following actions:

1. Invoke a function *getreg* to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction **MOV y' , L** to place a copy of y in L.

3. Generate the instruction **OP z' , L** where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z.

Generating Code for Assignment Statements:

- The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:

  t : = a – b
  u : = a – c
  v : = t + u
  d : = v + u

with d live at the end.

Code sequence for the example is:

Statements	**Code Generated**	**Register descriptor**	**Address descriptor**
		Register empty	
t : = a - b	MOV a, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a - c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0
d : = v + u	ADD R_1, R_0 MOV R_0, d	R_0 contains d	d in R_0 d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements
a : = b [i] and **a [i] : = b**

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments
a : = *p and ***p : = a**

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z if x < 0 goto z	MOV y, R_0 ADD z, R_0 MOV R_0,x CJ< z

THE DAG REPRESENTATION FOR BASIC BLOCKS

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

Algorithm for construction of DAG

Input: A basic block

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

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

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

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

Step 2: For the case(i), create a node(OP) whose left child is node(y) and right child is

node(z). (Checking for common sub expression). Let n be this node.

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

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

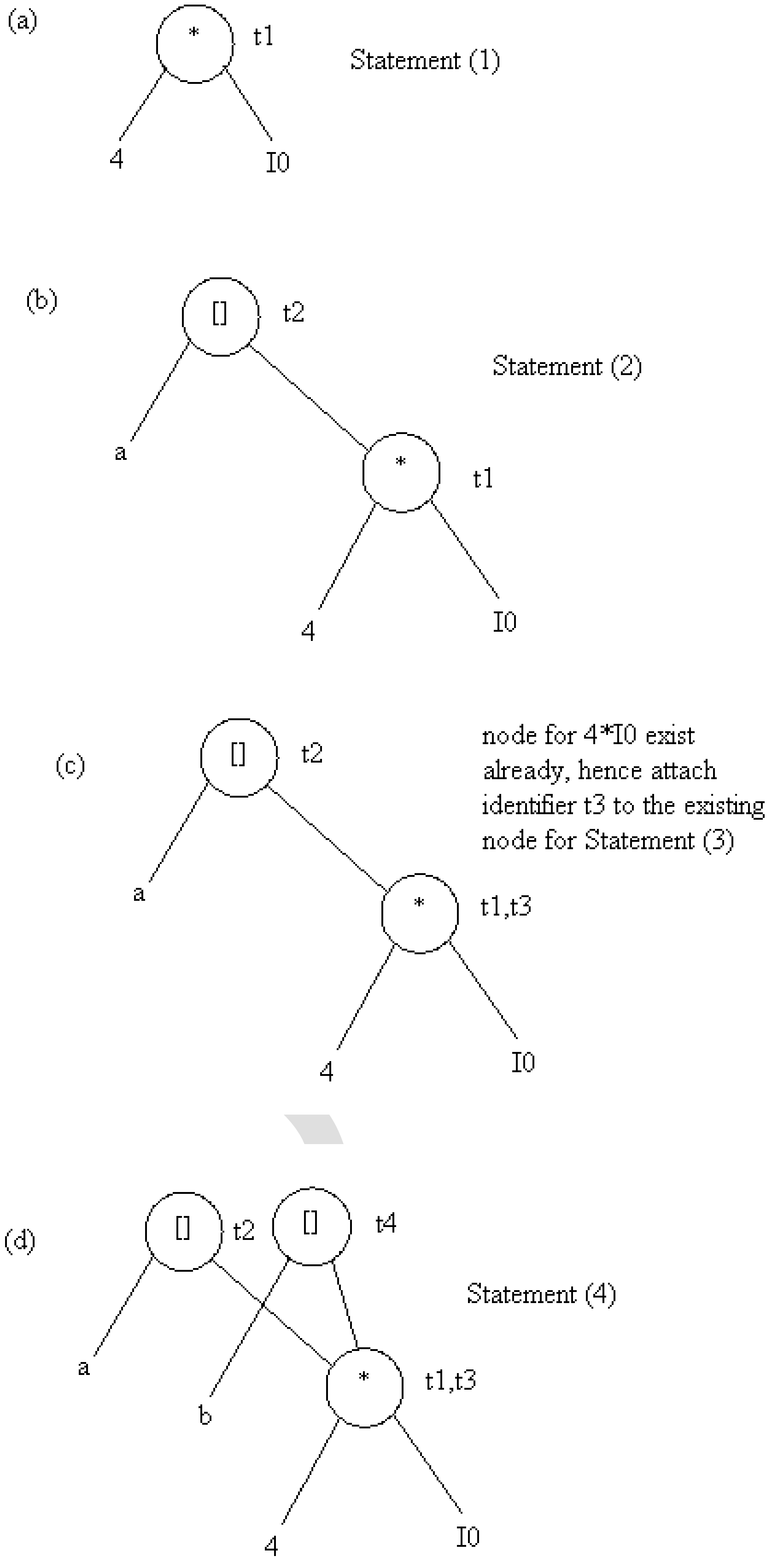
Step 3: Delete x from the list of identifiers for node(x). Append x to the list of attached

identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements:

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

Stages in DAG Construction



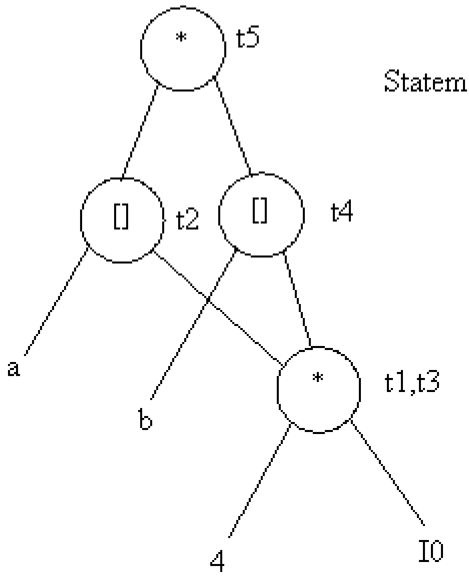
Statement (1)

Statement (2)

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

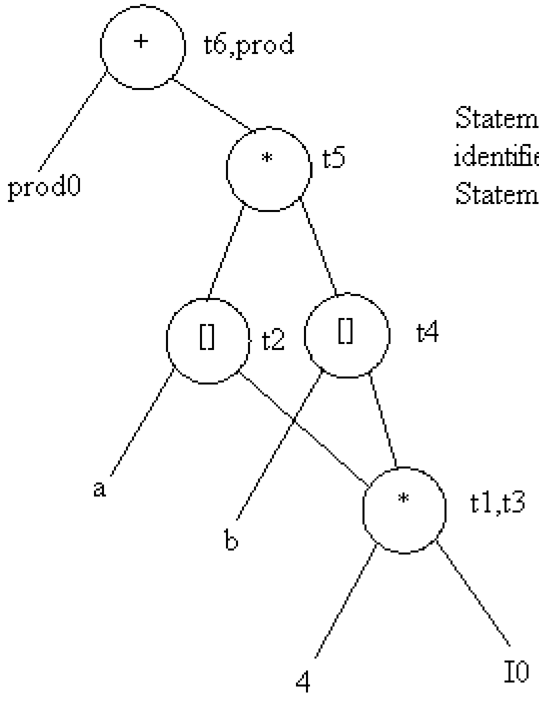
Statement (4)

(e)



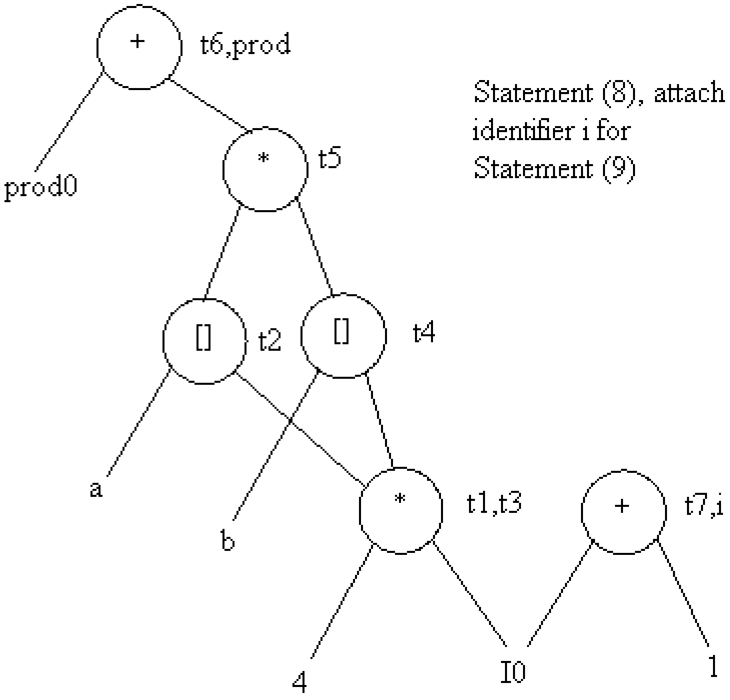
Statement (5)

(f)



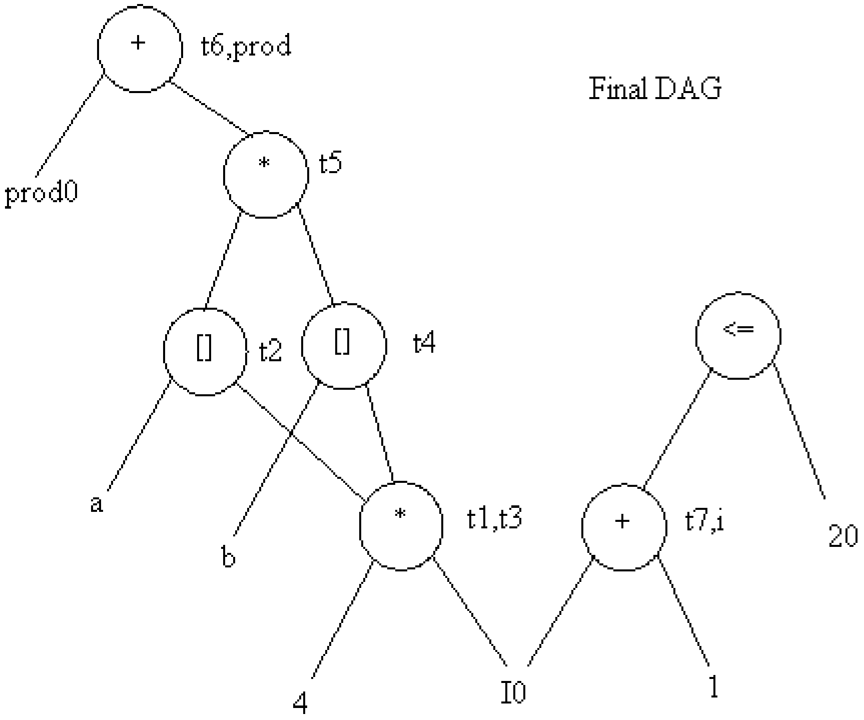
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

Application of DAGs:

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

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions **MOV R_0 , t_1** and **MOV t_1 , R_1** have been saved.

A Heuristic ordering for Dags

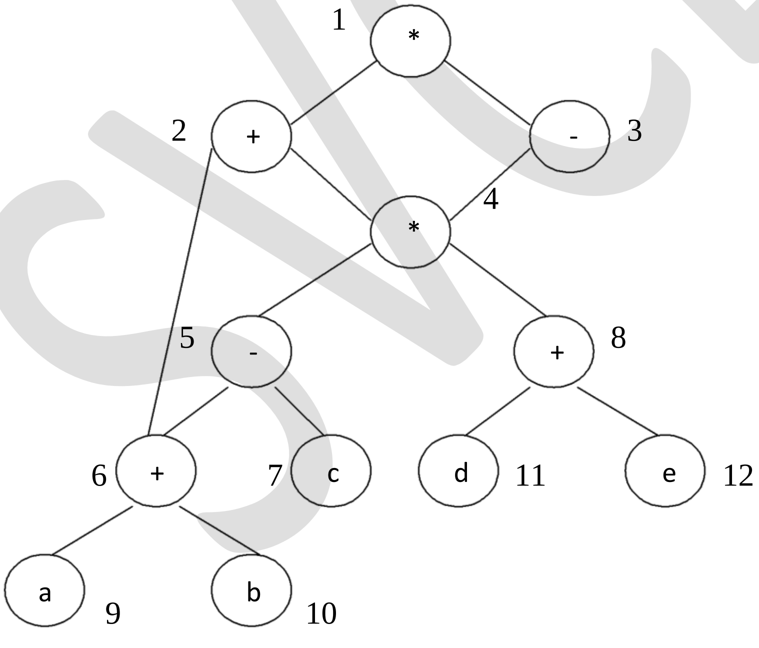
The heuristic ordering algorithm attempts to make the evaluation of a node immediately follow the evaluation of its leftmost argument.

The algorithm shown below produces the ordering in reverse.

Algorithm:

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

Example: Consider the DAG shown below:



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

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

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

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

Code sequence:

t_8 : = d + e
t_6 : = a + b
t_5 : = t_6 – c
t_4 : = t_5 * t_8
t_3 : = t4 – e
t_2 : = t_6 + t_4
t_1 : = t_2 * t_3

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