Module II
Scanning

Scanning

- Scanning is the process of recognizing the lexical components in a source string.
- The lexical features of a language can be specified using Type 3 or Regular grammars.
- This facilitates automatic generation of efficient recognizers for the lexical features of the language
- LEX : Scanner generator which generates such recognizers from the string specifications input to it.

Finite State Automata

- An **automaton** is an abstract self-propelled computing device which follows a pre-determined sequence of operations automatically.
- **Finite automaton or Finite State Machine**: An automaton with finite number of states.
- **Deterministic Finite automaton** : In DFA, for each input symbol, one can determine the state to which the machine will move. Hence it is called Deterministic Automaton. As it has a finite number of states- DFA
- **Definition**: A finite state automaton is a triple (S, s_{init}, T) where

  S : a finite set of states, one of which is the initial state and one or more of which are the final states

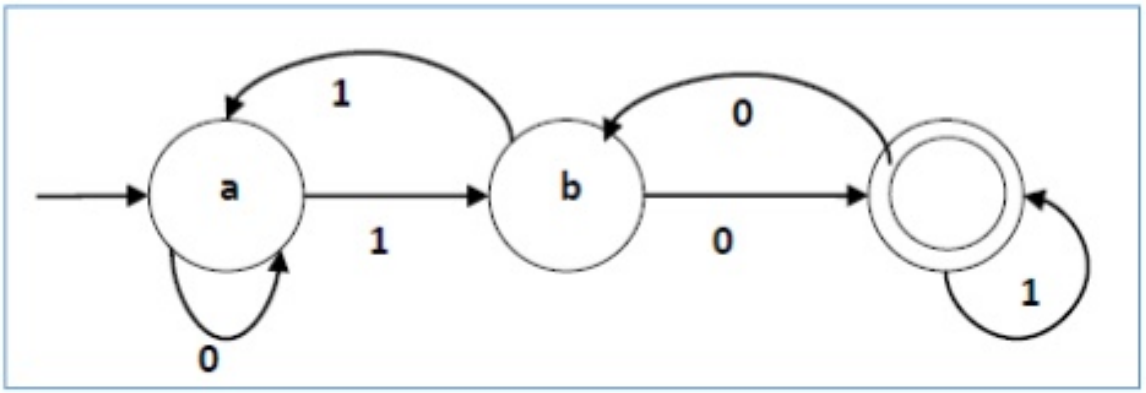
  Σ : alphabet of source symbols

  T : a finite set of state transitions defining transitions

- The transitions in an FSA can be represented in the form of a state transition table (STT) which has one row for each state $s \in S$ and one column for each symbol $\in \Sigma$

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

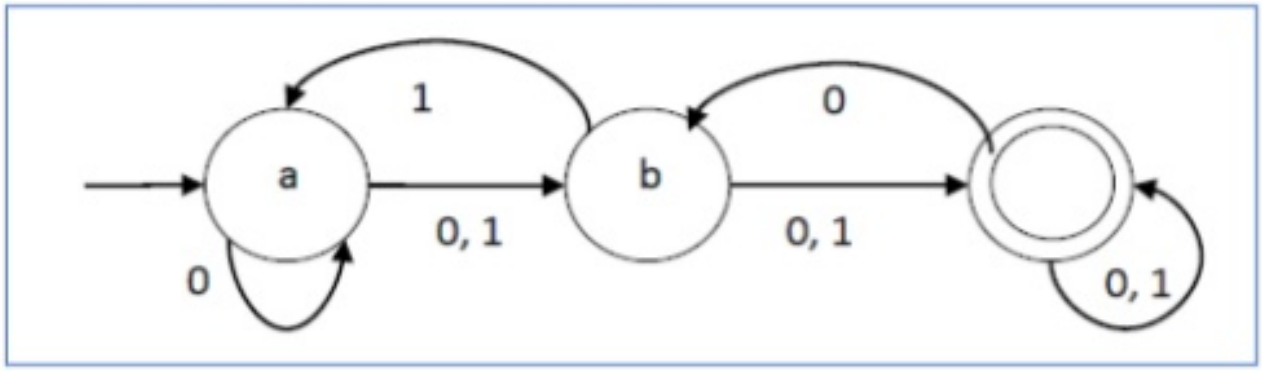
Its graphical representation would be as follows:



- **Non –deterministic Finite Automaton :** In an NDFA for a particular input symbol, the machine can move to any combination of the states in the machine. In other words the exact state to which the machine moves

Present State	Next State for Input 0	Next State for Input 1
a	a, b	b
b	c	a, c
c	b, c	c

Its graphical representation would be as follows:



Overview of Scanning

- The function of a scanner (called also lexical analyzer) is to:
  - Read characters from the source file
  - Group input characters into meaningful units, called **tokens**
- The scanner takes care of other things as well:
  - Removal of comments and white space
  - Keeping track of current line number
    - Required for reporting error messages
  - Case conversions of identifiers and keywords
    - Simplifies searching if the language is not case-sensitive
  - Interpretation of compiler directives
    - Flags are internally set to direct code generation
  - Communication with the symbol or literal table
    - Identifiers can be entered in the symbol table
    - String literals can be entered in the literal table

Tokens and Lexemes

- Consider the following statement:

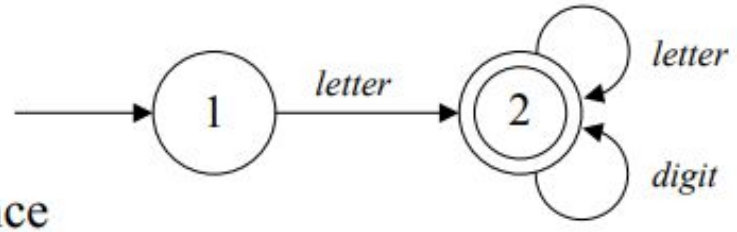
  if distance >= rate * (time1 – time0) **then** distance := maxdist ;

  - Contains 5 identifiers: distance, rate, time1, time0, maxdist
- For parsing purposes, all identifiers are alike
  - It is enough to tell the parser that the next token is an identifier
  - However, the code generator needs the name of the identifier
- Similarly, for parsing purposes all relational operators are alike
  - The syntactic structure would not change if >= were changed to > or <=
  - However, the code generator needs to know exactly what operator is used
- We make the following distinction:
  - A **token** is a logical entity described as part of the syntax of a language
  - A **lexeme** is a special instance of the token, which is the **string value**
- For the above statement, the scanner should return the following tokens:

  if id relop id * (id – id) then id := id ;

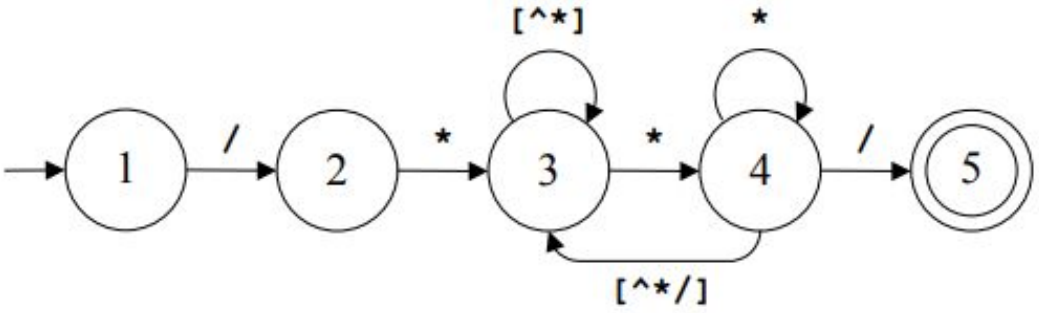
Finite Automata

- Used to recognize the tokens specified by a regular expression
- Can be converted to an algorithm for matching input strings
- A Finite Automaton (FA) consists of:
  - A finite set of **states**
  - A set of **transitions** (or moves) between states
    - The transitions are labeled by characters form the alphabet
  - A special **start state**
  - A set of **final** or **accepting states**
- A finite automaton for `letter(letter|digit)*` is shown below
- We may label a transition with more than one character for convenience
- We start at the start state
- We make a transition if next input character matches label on transition
- If no move is possible, we stop
- If we end in an accepting state then
  - input sequence of characters is valid
- Otherwise, we do not have a valid sequence



Deterministic Finite Automata (DFA)

- Has a **unique** transition for every state and input character
- Can be represented by a **transition table T**
  - Table **T** is indexed by state *s* and input character *c*
  - **T**[*s*][*c*] is the next state to visit from state *s* if the input character is *c*
  - **T** can also be described as a **transition function**
  - **T**: $S \times \Sigma \rightarrow S$ maps the pair (*s*, *c*) to *next_s*
- DFA and transition table for a C comment are show below
  - Blank entries in the table represent an **error state**
  - A full transition table will contain one column for each character (may waste space)
  - Characters are combined into **character classes** when treated identically in a DFA

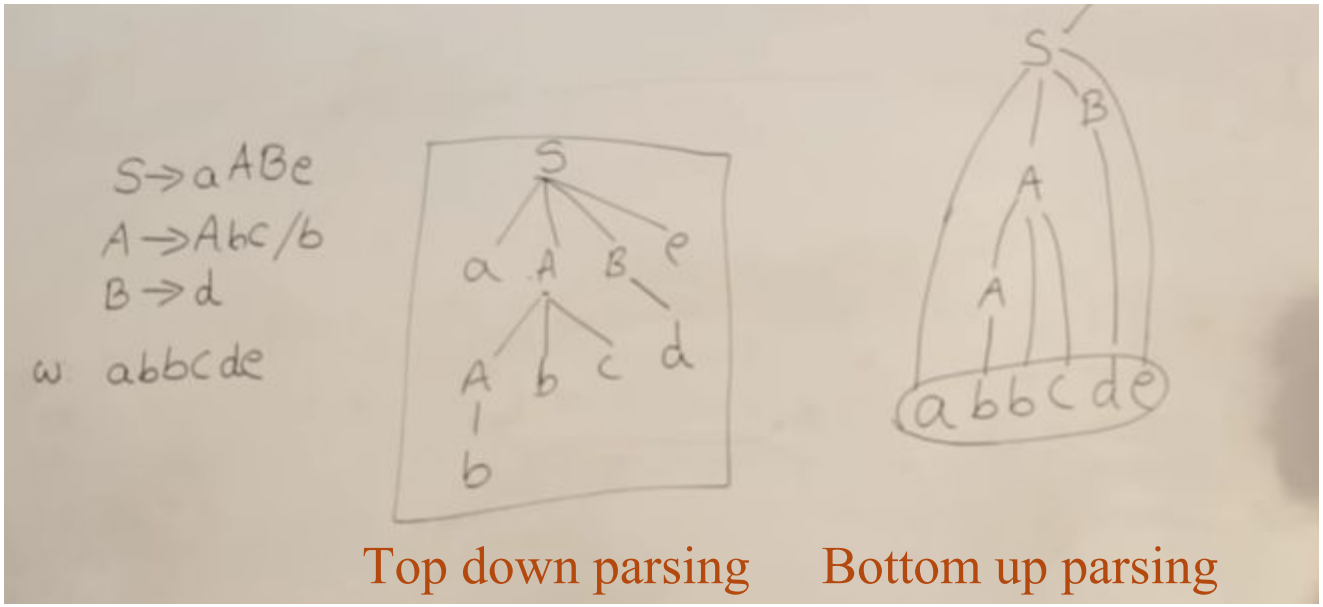


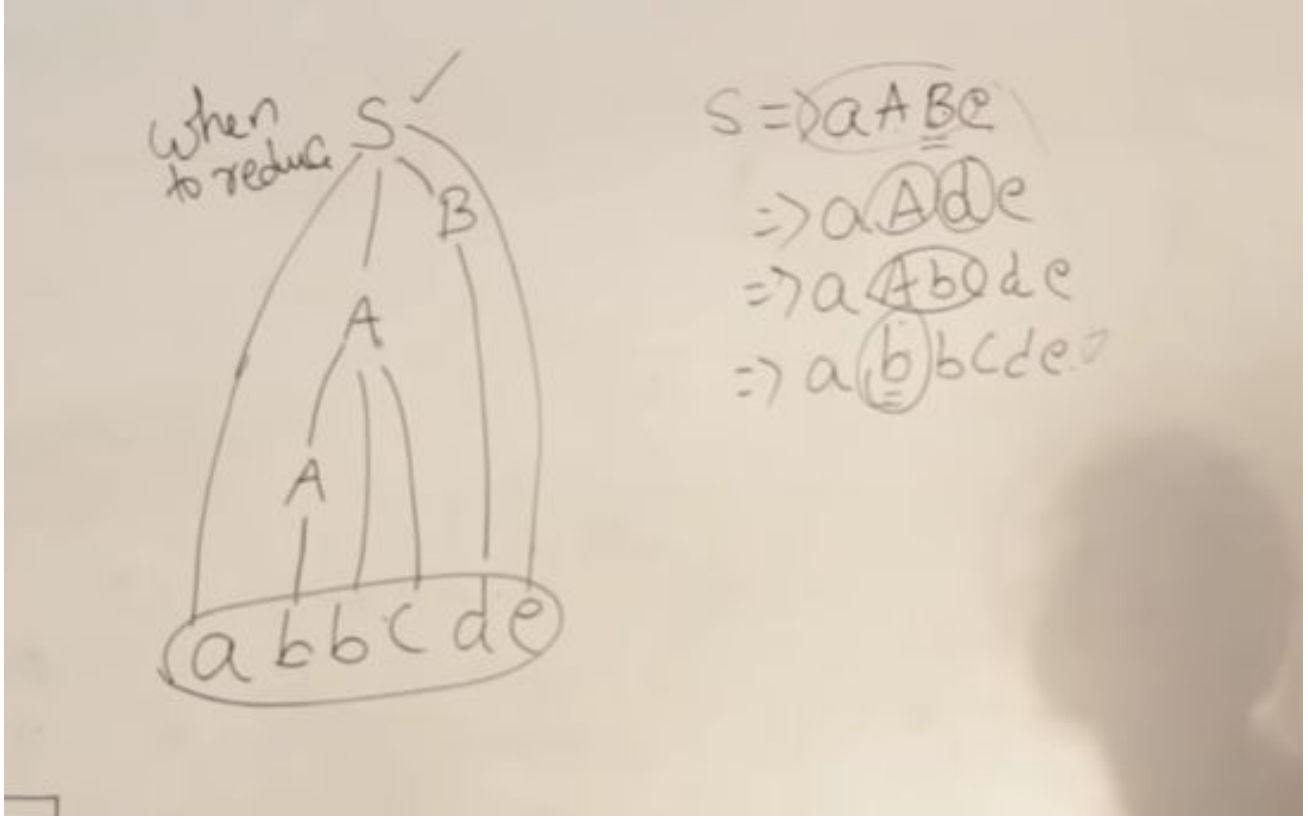
State	/	*	other
1	2		
2		3	
3	3	4	3
4	5	4	3
5			

Parsing

- Parsing is the process of turning a stream of tokens into a parse tree, according to the rules of some grammar. This is the second, and more significant, part of syntax analysis (after scanning).
- **Syntax Analyzer (Parser):** Syntax analyzer creates the syntactic structure of the given source program. This syntactic structure is mostly a parse tree. The syntax of a programming is described by a **Context-Free Grammar** (CFG).
- The syntax analyzer (parser) checks whether a given source program satisfies the rules implied by a context-free grammar or not. If it satisfies, the parser creates the parse tree of that program. Otherwise the parser gives the error messages.
- Important tasks perform by the parser:
  - Helps you to detect all types of Syntax errors
  - Find the position at which error has occurred
  - Clear & accurate description of the error.

- We categorise the parser into two groups
  1. Top-down parser (starts from the root).
  2. Bottom-up parser (starts from the leaf).
- Example: Consider the grammar





- Top down parsing uses **left most derivation** to derive the string and uses **substitutions** during derivation process.
- Bottom up parsing uses **reverse of right most derivation** to verify the string and uses **reductions** during the process.

Consider another grammar E:= E+T | E-T | T

T:= T*F | T/F | F

F:= num | id

For the input string**: id(x) + num(2) * id(y)**

Analysis of the top-down parsing:

E => E + T

=> E + T * F

=> T + T * F

=> T + F * F

=> T + num * F

=> F + num * F

=> id + num * F

=> id + num * id

Analysis of the Bottom-up parsing:

id(x) + num(2) * id(y)

=> id(x) + num(2) * F

=> id(x) + F * F

=> id(x) + T * F

=> id(x) + T

=> F + T

=> T + T

=> E + T

=> E

Top Down Parsing

Top down parsing according to a grammar G attempts to derive a string matching a source string through a sequence of derivations starting with the distinguished symbol of G. For a valid source string α, a top down parse thus determines a derivation sequence

$$S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow \alpha.$$

We shall identify the important issues in top down parsing by trying to develop a naive algorithm for it.

Algorithm 3.1 (Naive top down parsing)

1. *Current sentential form* (CSF) := 'S';
2. Let CSF be of the form $\beta A \pi$, such that β is a string of Ts (note that β may be null), and A is the leftmost NT in CSF. Exit with success if CSF = α.
3. Make a derivation $A \Rightarrow \beta_1 B \delta$ according to a production $A ::= \beta_1 B \delta$ of G such that β_1 is a string of Ts (again, β_1 may be null). This makes CSF = $\beta\beta_1 B \delta\pi$.
4. Go to Step 2.

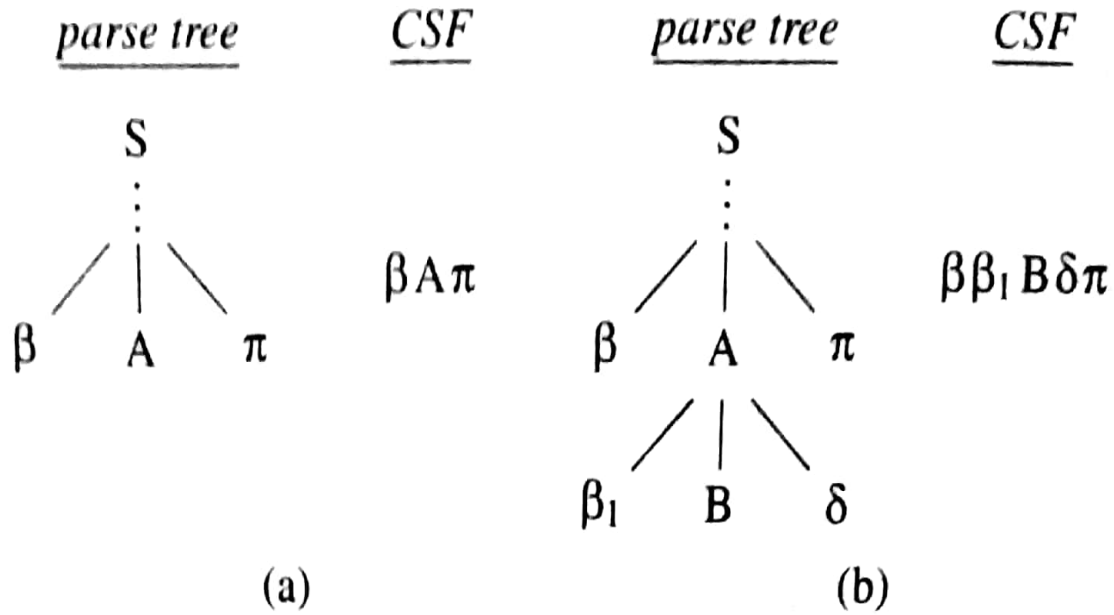


Fig. 3.5 Derivation $A \Rightarrow \beta_1 B\delta$ in top down parsing

Figure 3.5 depicts a step in top down parsing according to Algorithm 3.1. Since we make a derivation for the leftmost NT at any stage, top down parsing is also known as *left-to-left parsing* (LL Parsing).

mplementing top down parsing

The following features are needed to implement top down parsing:

1. *Source string marker* (SSM): SSM points to the first unmatched symbol in the source string.
2. *Prediction making mechanism:* This mechanism systematically selects the RHS alternatives of a production during prediction making. It must ensure that any string in L_G can be derived from S.
3. *Matching and backtracking mechanism:* This mechanism matches every terminal symbol generated during a derivation with the source symbol pointed to by SSM. (This implements the incremental continuation check.) Backtracking is performed if the match fails. This involves resetting CSF and SSM to earlier values.

Continuation check and backtracking is performed in Step 3 of Algorithm 3.1. A complete algorithm incorporating these features can be found in (Dhamdhere, 1983).

Example 3.7 Lexically analysed version of the source string `a+b*c`, viz. $< id > + < id > * < id >$ is to be parsed according to the grammar

$$\begin{array}{lll} E & ::= & T + E \mid T \\ T & ::= & V * T \mid V \\ V & ::= & < id > \end{array} \qquad (3.1)$$

The prediction making mechanism selects the RHS alternatives of a production in a left-to-right manner. First few steps in the parse are:

1. SSM := 1; CSF := E;
2. Make the prediction E ⇒ T + E. Now, CSF = T + E.
3. Make the prediction T ⇒ V * T. CSF = V * T + E.
4. Make the prediction V ⇒ < *id* >. CSF = < *id* > * T + E. < *id* > matches with the first symbol of the source string. Hence, SSM := SSM + 1;
5. Match the second symbol of the prediction in Step 3, viz. '*'. This match fails, hence reject the prediction T ⇒ V * T. SSM and CSF are reset to 1 and T + E, respectively. (The situation now resembles that at the end of Step 2.)
6. Make a new prediction for T, viz. T ⇒ V. CSF = V + E.
7. Make the prediction V ⇒ < *id* >. CSF = < *id* > + E. < *id* > matches with the first symbol of the source string. Hence, SSM := SSM + 1;
8. Match the second symbol of the prediction in Step 2, viz. '+'. Match succeeds. SSM := SSM + 1;
9. Make the prediction E ⇒ T + E. CSF = < *id* > + T + E.
10. Make the prediction T ⇒ V * T. CSF = < *id* > + V * T + E.
11. ...

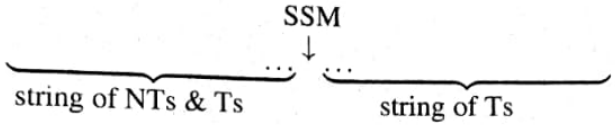
The predictions surviving at the end of the parse are listed in Table 3.3.

Table 3.3 Predictions in top sown parsing

Prediction	*Predicted Sentential Form*
E ⇒ T + E	T + E
T ⇒ V	V + E
V ⇒ < *id* >	< *id* > + E
E ⇒ T	< *id* > + T
T ⇒ V * T	< *id* > + V * T
V ⇒ < *id* >	< *id* > + < *id* > * T
T ⇒ V	< *id* > + < *id* > * V
V ⇒ < *id* >	< *id* > + < *id* > * < *id* >

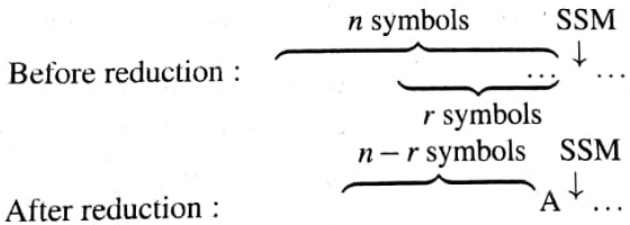
Bottom Up Parsing

A bottom up parser constructs a parse tree for a source string through a sequence of reductions. The source string is valid if it can be reduced to S, the distinguished symbol of G. If not, an error is to be detected and reported during the process of reduction. Bottom up parsing proceeds in a left-to-right manner, i.e. attempts at reduction start with the first symbol(s) in the string and proceed to the right. A typical stage during bottom up parsing is depicted as follows:



Reductions have been applied to the string on the left of SSM. Hence this string is composed of NTs and Ts. Remainder of the string, yet to be processed by the parser, consists of Ts alone.

We try the following naive approach to bottom up parsing: Let there be n symbols to the left of SSM in the current string form. We try to reduce some part of the string to the immediate left of SSM. Let this part have r symbols in it, and let it be reduced to the NT A. This reduction can be depicted as follows:



Since we do not know the value of r, we try the values $r = n$, $r = n - 1$, ... $r = 1$. This approach is summarized in Algorithm 3.2.

Algorithm 3.2 (Naive bottom up parsing)

1. SSM := 1; n := 0;
2. $r := n$;
3. Compare the string of r symbols to the left of SSM with *all* RHS alternatives in G which have a length of r symbols.

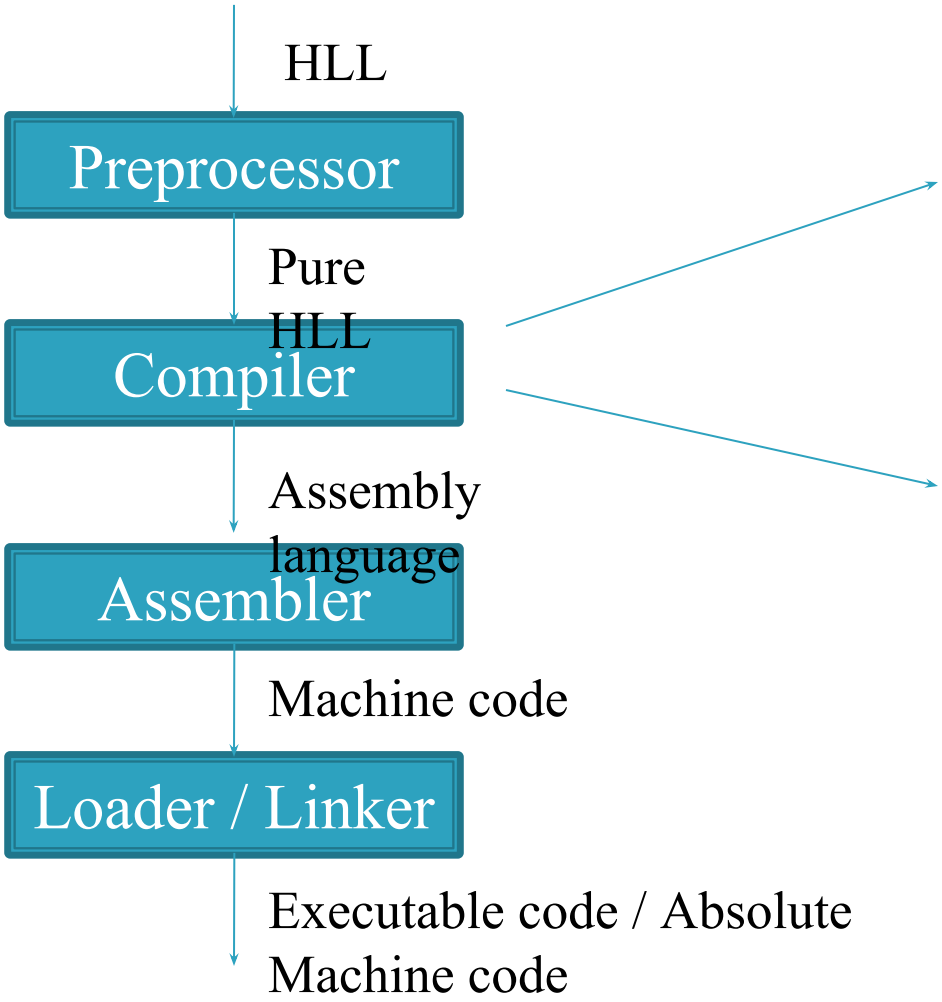
4. If a match is found with a production A ::= α, then
   reduce the string of r symbols to the NT A.
   $n := n - r + 1$;
   Go to Step 2;
5. $r := r - 1$;
   If $r > 0$, go to Step 3;
6. If no more symbols exist to the right of SSM then
   if current string form = 'S' then
   exit with success
   else report error and exit with failure
7. SSM := SSM + 1;
   $n := n + 1$;
   Go to Step 2;

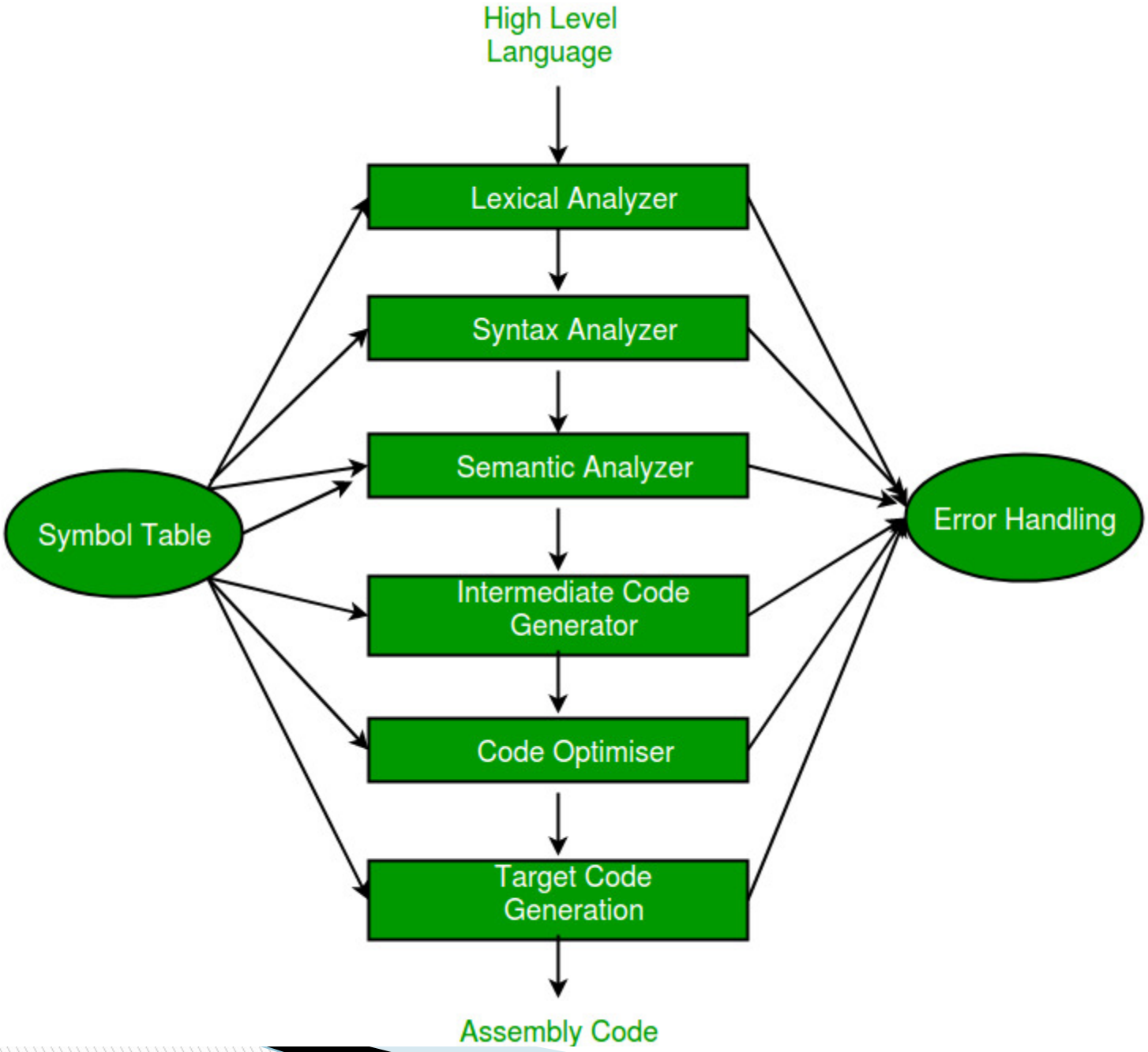
Thus the parser makes as many reductions as possible at the current position of SSM (see Step 5). When no reductions are possible at the current position, SSM is incremented by one symbol (see Step 7). This is called a *shift* action. The parsing process thus consists of shift and reduce actions applied in a left-to-right manner. Hence bottom up parsing is also known as *LR parsing* or *shift-reduce parsing*.

25/08/2020

Phases of Compiler

To convert high level language to a low level language




High Level Language
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Intermediate Code Generator
Code Optimiser
Target Code Generation
Assembly Code
Symbol Table
Error Handling

Phases of Compiler (Eg)

`x = a + b * c ; /* statement */`

⇓

Lexical analyser

⇓

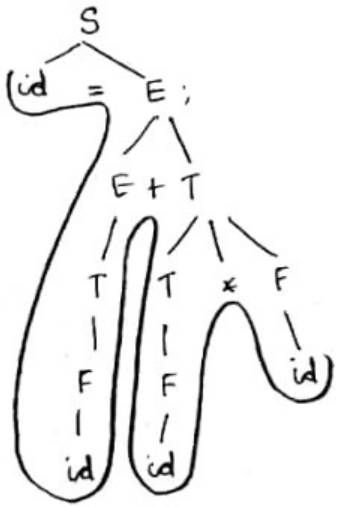
`id = id + id * id`

Stream of tokens

⇓

Syntax analyser

⇓



⇓

Semantic analyser

⇓

(Parse tree semantically verified) — meaningful type checking

`l(l+d)*` ← Pattern

Pattern should be known to lexical analyser before

$S \rightarrow id = E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

← Grammar. (Rules of PL)

FRONT END

⇓

Intermediate Code generator

⇓

```
t1 = b * c ;
t2 = a + t1 ;
x  = t2 ;
```

Three address code

⇓

Code Optimizer

⇓

```
t1 = b * c ;
x = a + t1 ;
```

⇓

Target Code Generator

⇓

```
MUL R1, R2
ADD R0, R1
MOVER X, R0
```

FRONT END

BACK END

- Lexical Analysis: The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

  <token-name, attribute-value>

- Syntax Analysis: The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.
- Semantic Analysis: Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.
- Intermediate Code Generation: After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.
- Code Optimization: The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).
- Code Generation: In t[illegible]phase, the code generator takes the optimized representation of the intermediate code and maps it[illegible]rget machine language. The code generator translates

Aspects of Compilation

27/08/20

CHAPTER 6

Compilers and Interpreters

6.1 ASPECTS OF COMPILATION

A compiler bridges the semantic gap between a PL domain and an execution domain. Two aspects of compilation are:

1. Generate code to implement meaning of a source program in the execution domain.
2. Provide diagnostics for violations of PL semantics in a source program.

To understand the issues involved in implementing these aspects, we briefly discuss PL features which contribute to the semantic gap between a PL domain and an execution domain. These are:

1. Data types
2. Data structures
3. Scope rules
4. Control structure.

Data types

Definition 6.1 (Data type) A data type *is the specification of* (*i*) *legal values for variables of the type, and* (*ii*) *legal operations on the legal values of the type.*

Legal operations of a type typically include an assignment operation and a set of data manipulation operations. Semantics of a data type require a compiler to ensure that variables of a type are assigned or manipulated only through legal operations. The following tasks are involved in ensuring this:

operations of a type.

Example 6.1 Consider the Pascal program segment

```
var
   x, y : real;
   i, j : integer;
begin
   y := 10;
   x := y + i;
```

While compiling the first assignment statement, the compiler must note that y is a real variable, hence every value stored in y must be a real number. Therefore it must generate code to convert the value '10' to the floating point representation. In the second assignment statement, the addition cannot be performed on the values of y and i straightaway as they belong to different types. Hence the compiler must first generate code to convert the value of i to the floating point representation and then generate code to perform the addition as a floating point operation.

Having checked the legality of each operation and determined the need for type conversion operations, the compiler must generate *type specific code* to implement an operation. In a type specific code the value of a variable of type $type_i$ is always manipulated through instructions which know how values of $type_i$ are represented.

Example 6.2 In example 1.12, we have seen how the instructions

```
CONV_R      AREG, I
ADD_R       AREG, B
MOVEM       AREG, A
```

are generated for the program segment

Data structures

A PL permits the declaration and use of data structures like arrays, stacks, records, lists, etc. To compile a reference to an element of a data structure, the compiler must develop a memory mapping to access the memory word(s) allocated to the element. A record, which is a heterogeneous data structure, leads to complex memory mappings. A user defined type requires mappings of a different kind—those that map the values of the type into their representations in a computer, and vice versa.

Example 6.3 Consider the Pascal program segment

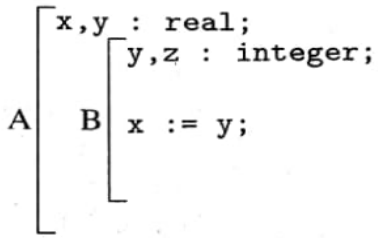
```
program example (input, output);
   type
      employee = record
         name : array [1..10] of character;
         sex : character;
         id : integer
      end;
      weekday = (mon, tue, wed, thu, fri);
   var
      info : array [1..500] of employee;
      today : weekday;
      i,j : integer;
   begin { Main program }
         today := mon;
         info[i].id := j;
         if today = tue then ..
end.
```

Here, `info` is an array of records. The reference `info[i].id` involves use of two different kinds of mappings. The first one is the homogeneous mapping of an array reference. This is used to access `info[i]`. The second mapping is used to access the field `id` within an element of `info`, which is a data item of type `employee`. `weekday` is a user defined data type. The compiler must first decide how to represent different values of the type, and then develop an appropriate mapping between the values `mon .. fri` and their representations. A popular technique is to map these values into a subrange of integers. For example, `mon .. fri` can be mapped into the subrange 1..5. This does not lead to confusion between these values and integers because the legality check is applied to each operation. Thus `tue+10` is an illegal expression even if `tue` is represented by the value '2', because '+' is not a legal operation of type `weekday`.

Scope rules

Scope rules determine the accessibility of variables declared in different blocks of a program. The *scope* of a program entity (e.g. a data item) is that part of a program where the entity is accessible. In most languages the scope of a data item is restricted to the program block in which the data item is declared. It extends to an enclosed block unless the enclosed block declares a variable with an identical name.

Example 6.4



Variable x of block A is accessible in block A and in the enclosed block B. However, variable y of block A is not accessible in block B since y is redeclared in block B. Thus, the statement x := y uses y of block B.

The compiler performs operations called *scope analysis* and *name resolution* to determine the data item designated by the use of a name in the source program. The generated code simply implements the results of the analysis. Section 6.2.2 contains a detailed discussion of the implementation of scope rules.

Control structure

The *control structure* of a language is the collection of language features for altering the flow of control during the execution of a program. This includes conditional transfer of control, conditional execution, iteration control and procedure calls. The compiler must ensure that a source program does not violate the semantics of control structures.

Example 6.5 In the Pascal program segment

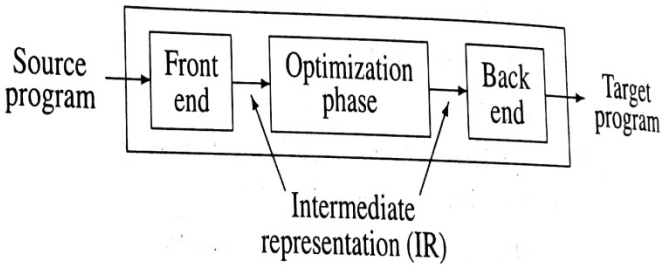
```
for i := 1 to 100 do
begin
   lab1: if i = 10 then ..
end;
```

a control transfer to the statement bearing label `lab1` from outside the loop is forbidden. Some languages (though not Pascal !) also forbid assignments to the control variable of a **for** loop within the body of the loop.

Code Optimization

The compiler was found to consume 40 percent extra compilation time due to optimization.
The optimized program occupied 25 percent less storage and executed three times as fast as the unoptimized program

- To improve the execution efficiency of a pr
- This is achieved in two ways:
  i. Redundancies in a program are eliminated.
  ii. Computations in a program are rearranged or rewritten to make it execute efficiently
- Code optimization must not change the meaning of a program
- Optimizatic
  the PL and



6.28 Schematic of an optimising compiler

- The front end generates an IR which could consists of triples, quadruples or ASTs. The optimization phase transforms this to achieve optimization. The transformed IR is input to the back end.

Optimizing Transformations

- An optimizing transformation is a rule for rewriting a segment of a program to improve its execution efficiency without affecting its meaning.
- They are classified into:
  1. Local Transformations - applied over small segments of a program consisting of a few source statements
  2. Global Transformations - applied over larger segments consisting of loops or function bodies

A few optimizing transformations commonly used in compilers are discussed below:

1.Compile time evaluation

- Execution efficiency can be improved by performing certain actions specified in a program during compilation itself.
- This eliminates the need to perform them during the execution of the program, thereby reducing the execution time of the program.
- Example : Constant folding

When all operands in an operation are constants, the operation can be performed at compile time.

a := 3.14157 / 2 can be replaced by a :=1.570785

2. Elimination of common sub expression

- Common expressions are occurrences of expressions yielding the same value. Also called equivalent expressions

ination of common subexpressions

Common subexpressions are occurrences of expressions yielding the same value. (Such expressions are called *equivalent expressions*.) Let CS_i designate a set of common subexpressions. It is possible to eliminate an occurrence $e_j \in CS_i$ if, no matter how the evaluation of e_j is reached during the execution of the program, the value of some $e_k \in CS_i$ would have been already computed. Provision is made to save this value and use it at the place of occurrence of e_j.

Example 6.31

```
a := b*c                t := b*c;
- - -          ⇒       a := t;
x := b*c+5.2;           - - -
                        x := t+5.2;
```

Here CS_i contains the two occurrences of `b*c`. The second occurrence of `b*c` can be eliminated because the first occurrence of `b*c` is always evaluated before the second occurrence is reached during execution of the program. The value computed at the first occurrence is saved in `t`. This value is used in the assignment to `x`.

This optimization is implemented as follows: First, expressions which yield the same value are identified. Many compilers simplify this task by restricting its scope to congruent, i.e. identical, subexpressions. These can be easily identified using triples or quadruples (see indirect triples, Section 6.3.2). Their equivalence is determined by considering whether their operands have the same values in all occurrences. Occurrences of the subexpression which satisfy the criterion mentioned earlier for expressions e_j can be eliminated. Some compilers also use rules of algebraic equivalence in common subexpression elimination. In the following program:

```
..:= b*c ..
d := b;
..:= d*c..
```

`d*c` is a common subexpression since `d` has the same value as `b`. Use of algebraic equivalence improves the effectiveness of optimization. However, it also increases the cost of optimization.

3. Dead code elimination

- Code which can be omitted from a program without affecting its results is called dead code.
- Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program.
- Example: An assignment x := <exp> constitutes dead code if the value assigned to x is not used in the program

4. Frequency reduction

- Execution time of a program can be reduced by moving code from a part of a program which is executed very frequently to another part of the program which is executed fewer times.
- Ex: the transformation of loop optimization moves loop invariant code out of a loop and places it prior to loop entry.

Example 6.33

```
for i := 1 to 100 do
begin
   z := i;
   x := 25*a;
   y := x+z;
end;
```

⇒

```
x := 25*a;
for i := 1 to 100 do
begin
   z := i;
   y := x+z;
end;
```

Here `x := 25*a;` is loop invariant. Hence in the optimized program it is computed only once before entering the **for** loop. `y := x+z;` is not loop invariant. Hence it cannot be subjected to frequency reduction.

5. Strength reduction

- The strength reduction optimization replaces the occurrence of a time consuming operation (a high strength operation) by an occurrence of a faster operation (a low strength operation).
- Ex: replacement o

Example 6.34 In Fig. 6.29, the 'high strength' operator '*' in `i*5` occurring inside the loop is replaced by a low strength operator '+' in `itemp+5`.

```
for i := 1 to 10 do
begin
  - - -
  k := i*5;
  - - -
end;
```

⇒

```
itemp := 5;
for i := 1 to 10 do
begin
  - - -
  k := itemp;
  - - -
  itemp := itemp+5;
end;
```

6.29 Strength reduction

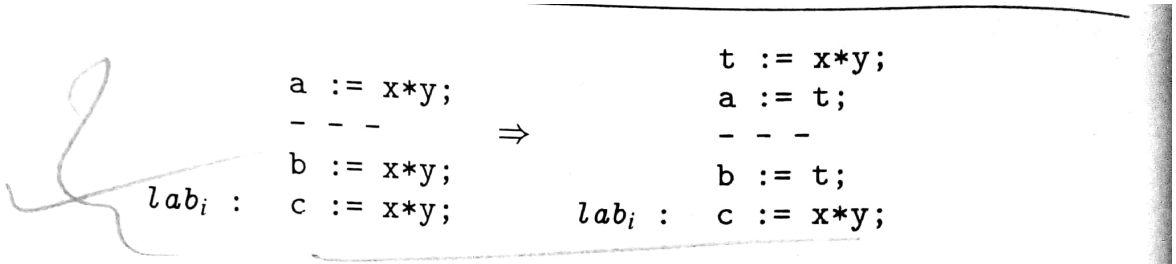
LOCAL OPTIMIZATION

- It provides limited benefits at a low cost.
- The scope of local optimization is a basic block which is an ***essentially sequential***

  segment in the source program.
- The cost of local optimization is low because the sequential nature of the basic block $S_1, S_2, \ldots\ldots S_n$

  simplifies the analysis needed for optimization.
- **Basic block :** A basic block is a sequence of program statements

  such that only S_n can be a transfer of control statement and only s1 can be the destination of a transfer of control statement.



where lab_i is a label. Local optimization identifies two basic blocks in the program. The first block extends up to the statement `b := x*y;`. Its optimization leads to elimination of the second occurrence of `x*y`. The third occurrence is not eliminated because it belongs to a different basic block. If the label lab_i did not exist, the entire program segment would constitute a single basic block and the third occurrence `x*y` can also be eliminated during local optimization.

Value Numbers

- To determine if two occurrences of an expression in a basic block are equivalent.
- A value number is associated with variable alpha.

Value numbers

Value numbers provide a simple means to determine if two occurrences of an expression in a basic block are equivalent. The value numbering technique is applied on the fly while identifying basic blocks in a source program. A value number vn_{alpha} is associated with variable `alpha`. It identifies the last assignment to `alpha` processed so far. Thus, the value number of variable `alpha` changes on processing an assignment `alpha := ...`. Now two expressions e_i and e_j are equivalent if they are congruent and their operands have the same value numbers.

For simplicity all statements of a basic block are numbered in some convenient manner. If statement n, the current statement being processed, is an assignment to `alpha`, we set vn_{alpha} to n. A new field is added to each symbol table entry to hold the value number of a variable. The IC for a basic block is a list of quadruples stored in a tabular form. Each operand field in a quadruple holds the pair (*operand, value number*). A boolean flag *save* is associated with each quadruple to indicate whether its value should be saved for use elsewhere in the program. The flag is initialized to *false* in every new quadruple entered in the table.

Let expression e be a subexpression of e', the expression being compiled. While forming a quadruple for e, the value numbers of its operands are copied from the symbol table. The new quadruple is now compared with all existing quadruples in the IC. Existence of a matching quadruple q_i in the IC implies that the current occurrence of expression e has the same value as a previous occurrence represented by quadruple q_i. If a match is found, the newly generated quadruple is not entered in the IC. Instead, the result name of q_i is used as an operand of e'. In effect, this occurrence of e is eliminated from the program. The result name of q_i should now become a compiler generated temporary variable. This requirement is noted by setting the *save* flag of q_i to *true*. During code generation, this flag is checked to see if the value of q_i needs to be saved in a temporary location.

y	0
x	15
g	14
z	0
d	5
w	0

Quadruples table

	Operator	Operand 1		Operand 2		Result name	Use flag
		Operand	Value no.	Operand	Value no.		
20	:=	g	–	25.2	–	t_{20}	*f*
21	+	z	0	2	–	t_{21}	*f*
22	:=	x	0	t_{21}	–	t_{22}	*f*
23	*	x	15	y	0	t_{23}	*ft*
24	+	t_{23}	–	d	5	t_{24}	*f*
	⋮						
57	:=	w	0	t_{23}	–	t_{57}	*f*

Fig. 6.30 Local optimization using value numbering

stmt no.	*statement*
14	`g := 25.2;`
15	`x := z+2;`
16	`h := x*y+d;`
..	...
34	`w := x*y;`

Local optimization proceeds as follows: All variables are assumed to have the value numbers '0' to start with. Processing of Statements 14 and 15 leads to generation of quadruples numbered 20–22 shown in the table. Value numbers of g, x and y at this stage are 14, 15 and 0, respectively (see the symbol table). Quadruple for `x*y` is generated next, and the value numbers of x and y are copied from the symbol table. This quadruple is assigned the result name t_{23}, and its *save* flag is set to *false*. This quadruple is entered in entry number 23 of the quadruple table. When the statement `w := x*y` (i.e., Statement 34) is processed, the quadruple for `x*y` is formed using the value numbers found in the symbol table entries of x and y. Since these are still 15 and 0, the new quadruple is identical with quadruple 23 in the table. Hence it is not entered in the table. Instead, the *save* flag of quadruple 23 is set to *true*. The only quadruple generated for this statement is therefore the assignment to w (quadruple number 57). While generating code for quadruple 23, its save flag indicates that the

value of expression `x*y` needs to be saved in a temporary location for later use. t_{23} can itself become the name of this temporary location.

This schematic can be easily extended to implement *constant propagation*, which is the substitution of a variable *var* occurring in a statement by a constant *const*, and constant folding. When an assignment of the form *var* := *const* is encountered, we enter *const* into a table of constants, say in entry *n*, and associate the value number '–*n*' with *var*. Constant propagation and folding is implemented while generating a quadruple if each operand is either a constant or has a negative value number.

Example 6.37 In the following program, variable `a` is given a negative value number, say '–10', on processing the first statement.

```
a := 27.3;          a := 27.3;
- - -        ⇒     - - -
b := a*3.0;         b := 81.9;
```

This leads to the possibility of constant propagation and folding in the third statement. The value of `a`, viz. '27.3', is obtained from the 10^{th} entry of constants table. Its multiplication with 3.0 yields 81.9. This value is treated as a new constant and a quadruple for `b := 81.9` is now generated.

6.5.3 Global Optimization

Compared to local optimization, global optimization requires more analysis effort to establish the feasibility of an optimization. Consider global common subexpression elimination. If some expression `x*y` occurs in a set of basic blocks SB of program P, its occurrence in a block $b_j \in$ SB can be eliminated if the following two conditions are satisfied for every execution of P:

1. Basic block b_j is executed only after some block $b_k \in$ SB has been executed one or more times. (6.7)
2. No assignments to `x` or `y` have been executed after the last (or only) evaluation of `x*y` in block b_k.

Condition 1 ensures that `x*y` is evaluated before execution reaches block b_j, while condition 2 ensures that the evaluated value is equivalent to the value of `x*y` in block b_j. The optimization is realized by saving the value of `x*y` in a temporary location in all blocks b_k which satisfy condition 1.

To ensure that *every possible execution* of program P satisfies conditions 1 and 2 of (6.7), the program is analysed using the techniques of *control flow analysis* and *data flow analysis*. Note the emphasis on the words 'every possible execution'. This requirement is introduced to ensure that the meaning of the program is unaffected by the optimization. In this section we use the word 'always' to imply 'in every possible evaluation'.

6.5.3.1 Program Representation

A program is represented in the form of a *program flow graph.*

Definition 6.6 (Program flow graph (PFG))

A program flow graph *for a program P is a directed graph* $G_P = (N, E, n_0)$ *where*

N	:	*set of basic blocks in P*
E	:	*set of directed edges* (b_i, b_j) *indicating the possibility of control flow from the last statement of* b_i *(the* source node*) to the first statement of* b_j *(the* destination node*)*
n_0	:	*start node of P.*

A basic block, which is a sequence of statements $s_1, s_2, \ldots s_n$, is visualized as a sequence of *program points* $p_1, p_2, \ldots p_n$ such that a statement s_i is said to exist at program point p_i. This notion is used to differentiate between different occurrences of identical statements. For example, an occurrence of e at program point p_i is distinct from the occurrence of e at program point p_j.

6.5.3.2 Control and Data Flow Analysis

The techniques of control and data flow analysis are together used to determine whether the Conditions governing an optimizing transformation are satisfied in a program, e.g. conditions 1 and 2 of (6.7).

Control flow analysis

Control flow analysis analyses a program to collect information concerning its structure, e.g. presence and nesting of loops in the program. Information concerning program structure is used to answer specific questions of interest, e.g. condition 1 of (6.7). The control flow concepts of interest are:

1. *Predecessors and successors:* If $(b_i, b_j) \in E$, b_i is a *predecessor* of b_j and b_j is a *successor* of b_i.
2. *Paths:* A path is a sequence of edges such that the destination node of one edge is the source node of the following edge.
3. *Ancestors and descendants:* If a path exists from b_i to b_j, b_i is an ancestor of b_j and b_j is a descendant of b_i.
4. *Dominators and post-dominators:* Block b_i is a dominator of block b_j if every path from n_0 to b_j passes through b_i. b_i is a post-dominator of b_j if every path from b_j to an exit node passes through b_i.

Table 6.6 Data flow concepts

Data flow concept	*Optimization in which used*
Available expression	Common subexpression elimination
Live variable	Dead code elimination
Reaching definition	Constant and variable propagation

Control flow concetps can be used to answer certain questions in a straightforward manner, e.g. the question posed by condition 1 of (6.7). However, this incurs the overheads of control flow analysis for every expression in the program. It may sometimes be possible to reduce the overheads by restricting the scope of optimization. For example, only those expressions may be considered which occur in a block b_j and a dominator block b_k of b_j. Now condition 1 of (6.7) is automatically satisfied.

Data flow analysis

Data flow analysis techniques analyse the use of data in a program to collect information for the purpose of optimization. This information, called *data flow information*, is computed at the entry and exit of each basic block in G_P. It is used to decide whether an optimizing transformation (see Section 6.5.1) can be applied to a segment of code in the program.

Design of the global optimization phase begins with the identification of an appropriate *data flow concept* to support the application of each optimizing transformation. The data flow information concerning a program entity—for example a variable or an expression—is now a boolean value indicating whether the data flow concept is applicable to that entity. Table 6.6 contains a summary of important data flow concepts used in optimization. The first two data flow concepts are discussed in detail.

Available expressions

The transformation of global common subexpression elimination can be defined as follows: Consider a subexpression `x*y` occurring at program point p_i in basic block b_i. This occurrence can be eliminated if

1. Conditions 1 and 2 of (6.7) are satisfied at entry to b_i.
2. No assignments to `x` or `y` precede the occurrence of `x*y` in b_i.

The data flow concept of *available expressions* is used to implement common subexpression elimination. An expression e is *available* at program point p_i if a value equivalent to its value is always computed before program execution reaches p_i. Thus, the concept of available expressions captures the essence of Conditions 1 and 2 of (6.7). The availability of an expression at entry or exit of basic block b_i is computed using the following rules:

1. Expression e is available at the exit of b_i if

   (i) b_i contains an evaluation of e which is not followed by assignments to any operands of e, or
   (ii) the value of e is available at the entry to b_i and b_i does not contain assignments to any operands of e. (6.8)

2. Expression e is available at entry to b_i if it is available at the exit of each predecessor of b_i in G_P.

Available expressions is termed a *forward* data flow concept because availability at the exit of a node determines availability at the entry of its successor(s). It is an *all paths* concept because availability at entry of a basic block requires availability at the exit of all predecessors.

We use the boolean variables $Avail\_in_i$ and $Avail\_out_i$ to represent the availability of expression e at entry and exit of basic block b_i, respectively. Further, we associate the following boolean properties with block b_i to summarize the effect of computations situated in it:

$Eval_i$	:	'true' only if expression e is evaluated in b_i and none of its operands are modified following the evaluation
$Modify_i$	:	'true' only if some operand of e is modified in b_i.

$Eval_i$ and $Modify_i$ are determined solely by the computations situated in b_i. Hence they are called local properties of block b_i. $Avail\_in_i$ and $Avail\_out_i$ are 'global' properties which are computed using the following equations (see Eq. (6.8)):

$$Avail\_in_i = \Pi_{b_j \in pred(b_i)} Avail\_out_j \quad (6.9)$$

$$Avail\_out_i = Eval_i + Avail\_in_i \,.\, \neg Modify_i \quad (6.10)$$

where $\Pi_{b_j \in pred(b_i)}$ is the boolean 'and' operation over all predecessors of b_i. This operation ensures that $Avail\_in_i$ is true only if $Avail\_out$ is true for all predecessors of b_i. Equations (6.9) and (6.10) are called *data flow equations*.

It is to be noted that every basic block in G_P has a pair of equations analogous to (6.9)–(6.10). Thus, we need to solve a system of simultaneous equations to obtain the values of $Avail\_in$ and $Avail\_out$ for all basic blocks in G_P. Data flow analysis is the process of solving these equations. Iterative data flow analysis is a simple method which assigns some initial values to $Avail\_in$ and $Avail\_out$, and iteratively recomputes them for all blocks according to Eqs. (6.9)–(6.10) until they converge onto consistent values.

The initial values are:

$$Avail\_in_i = \begin{cases} true & \text{if } b_i \in N - \{n_o\} \\ false & \text{if } b_i = n_o \end{cases}$$

$$Avail\_out_i = true \quad \forall b_i$$

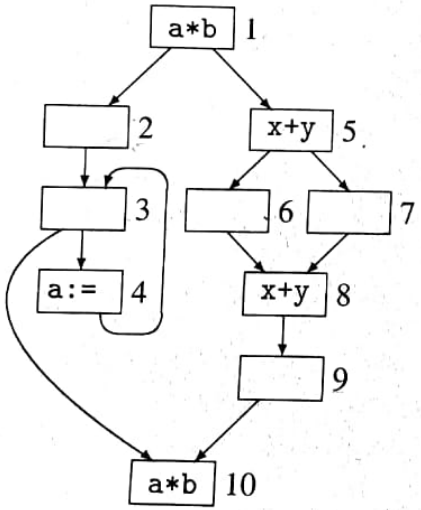
After solving the system of equations (6.9)–(6.10) for all blocks, an evaluatio of expression e can be eliminated from a block b_i if

1. $Avail\_in_i = true$, and
2. The evaluation of e in b_i is not preceded by an assignment to any of its operand

Example 6.38 Available expression analysis for the PFG of Fig. 6.31 gives the following results:

`a*b` : *Avail_in* = *true* for blocks 2, 5, 6, 7, 8, 9
Avail_out = *true* for blocks 1, 2, 5, 6, 7, 8, 9, 10

`x+y` : *Avail_in* = *true* for blocks 6, 7, 8, 9
Avail_out = *true* for blocks 5, 6, 7, 8, 9



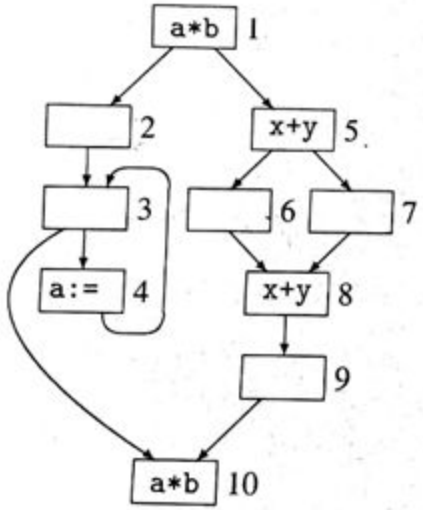
A PFG

The values of *Avail_in* and *Avail_out* for `a*b` can be explained as follows: The assignment `a := ..` in block 4 makes $Avail\_out_4 = false$ (eq. (6.10)). This makes $Avail\_in_3 = false$ (eq. (6.9)), which makes $Avail\_out_3 = Avail\_in_{10} = false$. For `x*y`, $Avail\_in_1 = false$ makes $Avail\_out_1 = false$, which makes *Avail_in* = *Avail_out* = *false* for blocks 2, 3, 4 and 10.

Live variables

A variable *var* is said to be *live* at a program point p_i in basic block b_i if the value contained in it at p_i is likely to be used during subsequent execution of the program. If *var* is not live at the program point which contains a definition *var* := ..., the value assigned to *var* by this definition is redundant in the program. Such a definition constitutes dead code which can be eliminated from the program without changing its meaning.

The liveness property of a variable can be determined as follows:



1. Variable v is live at the entry of b_i if

   (i) b_i contains a use of e which is not preceded by assignment(s) to v, or
   (ii) v is live at the exit of b_i and b_i does not contain assignment(s) to v.

2. v is live at exit of b_i if it is live at the entry of some successor of b_i in G_P.

Data flow information concerning live variables can be collected as follows:

$Live\_in_i$	:	*var* is live at entry of b_i
$Live\_out_i$	:	*var* is live at exit of b_i
Ref_i	:	*var* is referenced in b_i and no assignment of *var* precedes the reference
Def_i	:	An assignment to *var* exists in b_i

$$Live\_in_i = Ref_i + Live\_out_i \,.\, \neg Def_i \tag{6.11}$$

$$Live\_out_i = \Sigma_{b_j \in succ(b_i)} Live\_in_j \tag{6.12}$$

where $\Sigma_{b_j \in succ(b_i)}$ is the boolean 'or' operation over all successors of b_i.

Live variables is termed a *backward* data flow concept because availability at the entry of a block determines availability at the exit of its predecessor(s). It is an ***any path*** concept because liveness at the entry of one successor is sufficient to ensure liveness at the exit of a block. The data flow problem can be solved by iterative data flow analysis using the initializations $Live\_in_i = Live\_out_i = false \; \forall b_i$.

Example 6.39 In the PFG of Fig. 6.31, variable b is live at the entry of all blocks, a is live at the entry of all blocks excepting block 4 and variables x, y are live at the entry of blocks 1, 5, 6, 7 and 8.