Introduction to Compiling:

1.1 INTRODUCTION OF LANGUAGE PROCESSING SYSTEM

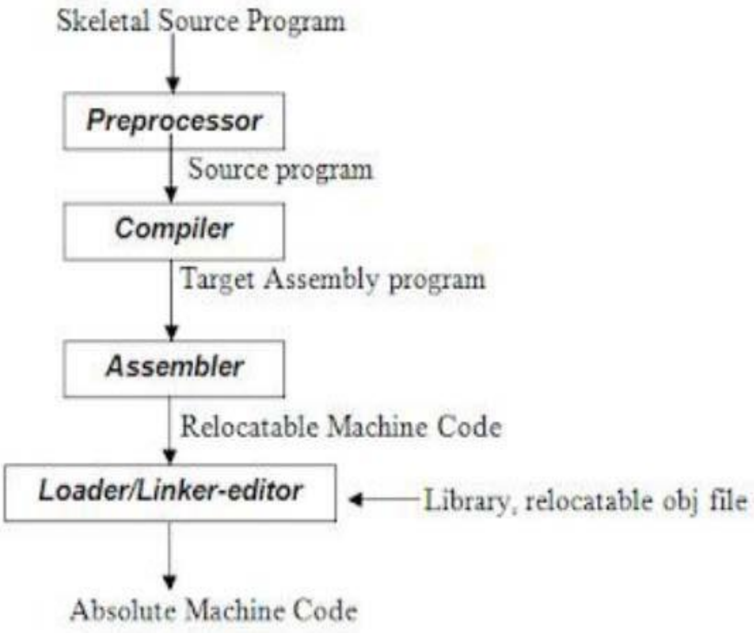


Fig 1.1: Language Processing System

Preprocessor

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

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

COMPILER

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

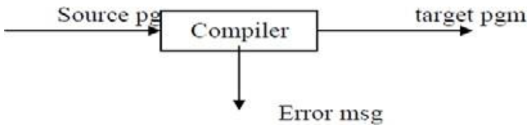


Fig 1.2: Structure of Compiler

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.

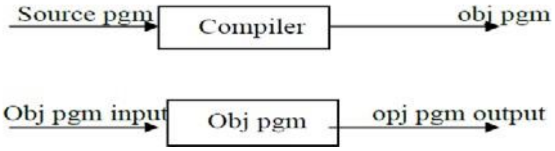


Fig 1.3: Execution process of source program in Compiler

ASSEMBLER

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

INTERPRETER

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

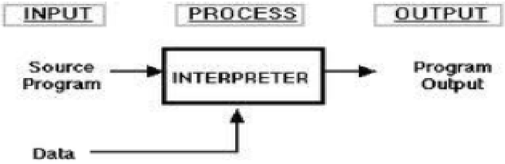


Fig1.4: Execution in Interpreter

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.

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.

1.2 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.

1.3 LIST OF COMPILERS

1. Ada compilers
2 .ALGOL compilers
3 .BASIC compilers
4 .C# compilers
5 .C compilers
6 .C++ compilers
7 .COBOL compilers
8 .Common Lisp compilers
9. ECMAScript interpreters
10. Fortran compilers
11 .Java compilers
12. Pascal compilers
13. PL/I compilers
14. Python compilers
15. Smalltalk compilers

1.4 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 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.**

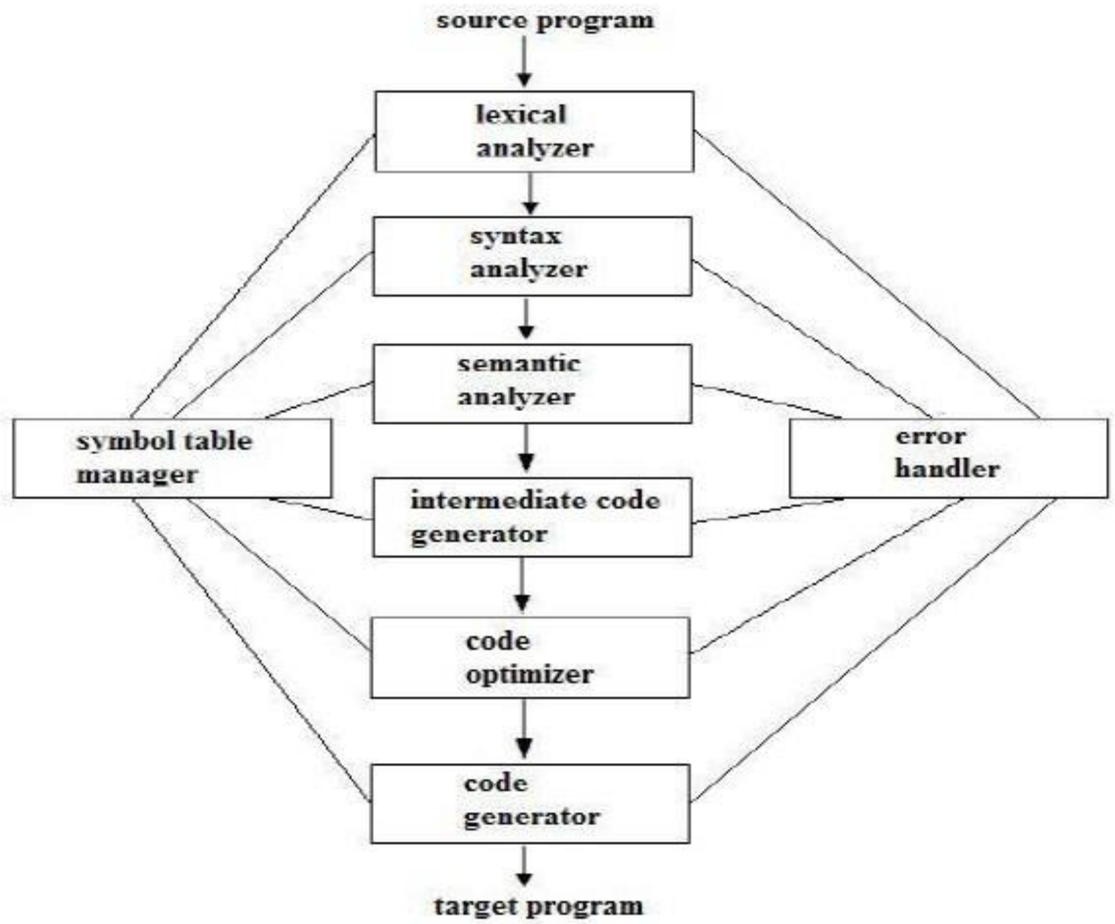


Fig 1.5: Phases of Compiler

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.

a. 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.**

b. 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 :-

Code Generator 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:

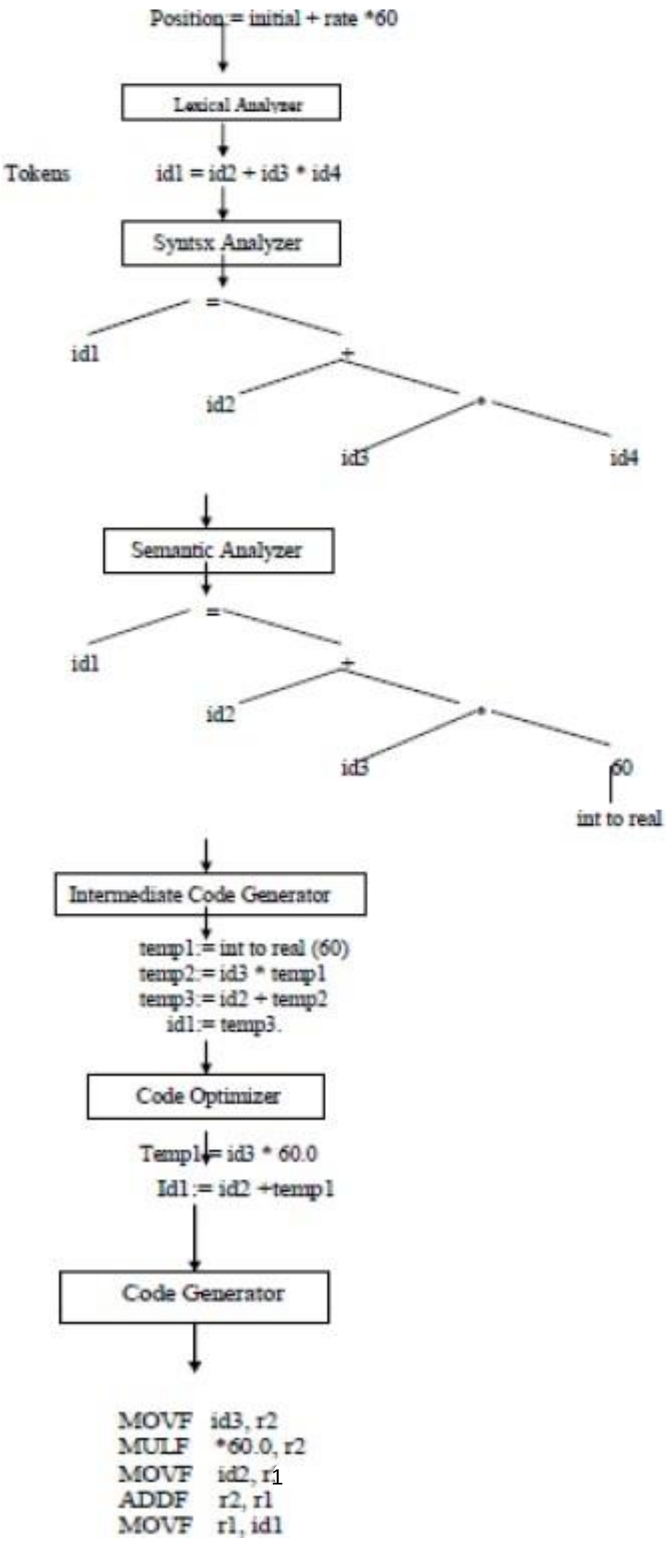


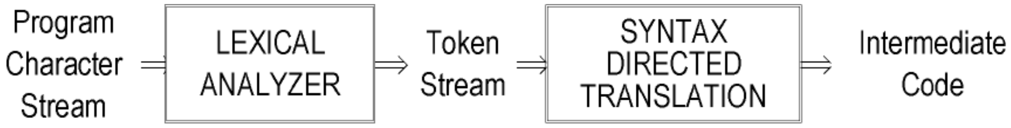
Fig 1.6: Compilation Process of a source code through phases

2. A simple One Pass Compiler:

2.0 INTRODUCTION: In computer programming, a **one-pass compiler** is a compiler that passes through the parts of each compilation unit only once, immediately translating each part into its final machine code. This is in contrast to a **multi-pass compiler** which converts the program into one or more intermediate representations in steps between source code and machine code, and which reprocesses the entire compilation unit in each sequential pass.

2.1 OVERVIEW

- Language Definition
  - Appearance of programming language :
    Vocabulary : Regular expression
    Syntax : Backus-Naur Form(BNF) or Context Free Form(CFG)
  - Semantics : Informal language or some examples



- **Fig 2.1.** Structure of our compiler front end

2.2 SYNTAX DEFINITION

- To specify the syntax of a language : CFG and BNF
  - Example : if-else statement in C has the form of statement →if (expression) statement else statement
- An alphabet of a language is a set of symbols.
  - Examples : {0,1} for a binary number system(language)={0,1,100,101,...}
    {a,b,c} for language={a,b,c, ac,abcc..}
    {if,(,),else ...} for a if statements={if(a==1)goto10, if--}
- A string over an alphabet
  - is a sequence of zero or more symbols from the alphabet.
  - Examples : 0,1,10,00,11,111,0202 ... strings for a alphabet {0,1}
  - Null string is a string which does not have any symbol of alphabet.
- Language
  - Is a subset of all the strings over a given alphabet.
  - Alphabets Ai Languages Li for Ai

Alphabets Ai	Languages Li for Ai
A0={0,1}	L0={0,1,100,101,...}
A1={a,b,c}	L1={a,b,c, ac, abcc..}
A2={all of C tokens}	L2= {all sentences of C program }

- Example 2.1. Grammar for expressions consisting of digits and plus and minus signs.
  - Language of expressions L={9-5+2, 3-1, ...}
  - The productions of grammar for this language L are:

list → *list* + *digit*
list → *list* - *digit*
list → *digit*
digit → *0|1|2|3|4|5|6|7|8|9*

  - ○ list, digit : Grammar variables, Grammar symbols
  - ○ **0,1,2,3,4,5,6,7,8,9,-,+** : Tokens, Terminal symbols

- Convention specifying grammar
  - ○ Terminal symbols : bold face string **if, num, id**
  - ○ Nonterminal symbol, grammar symbol : italicized names, list, digit ,A,B

- Grammar G=(N,T,P,S)
  - ○ N : a set of nonterminal symbols
  - ○ T : a set of terminal symbols, tokens
  - ○ P : a set of production rules
  - ○ S : a start symbol, S∈N

- Grammar G for a language L={9-5+2, 3-1, ...}
  - ○ G=(N,T,P,S)

    N={list,digit}
    T={0,1,2,3,4,5,6,7,8,9,-,+}
    P : *list* -> *list* + *digit*
    list -> *list* - *digit*
    list -> *digit*
    digit -> **0|1|2|3|4|5|6|7|8|9**
    S=*list*

- Some definitions for a language L and its grammar G
  - Derivation :

    A sequence of replacements S⇒α1⇒α2⇒…⇒αn is a derivation of αn.
    Example, A derivation 1+9 from the grammar G
    - left most derivation

      list ⇒ list + digit ⇒ digit + digit ⇒ 1 + digit ⇒ 1 + 9
    - right most derivation

      list ⇒ list + digit ⇒ list + 9 ⇒ digit + 9 ⇒ 1 + 9
  - Language of grammar L(G)

    L(G) is a set of sentences that can be generated from the grammar G.
    L(G)={x| S ⇒* x} where x ∈ a sequence of terminal symbols
  - Example: Consider a grammar G=(N,T,P,S):

    N={S} T={a,b}
    S=S P ={S → aSb | ε }
    - is aabb a sentecne of L(g)? (derivation of string aabb)

      S⇒aSb⇒aaSbb⇒aaεbb⇒aabb(or S⇒* aabb) so, aabbεL(G)
    - there is no derivation for aa, so aa∉L(G)
    - note L(G)={anbn| n≧0} where anbn meas n a's followed by n b's.

- **Parse Tree**

A derivation can be conveniently represented by a derivation tree(parse tree).

- ○ The root is labeled by the start symbol.
- ○ Each leaf is labeled by a token or ε.
- ○ Each interior none is labeled by a nonterminal symbol.
- ○ When a production A→x1… xn is derived, nodes labeled by x1… xn are made as children nodes of node labeled by A.
    - root : the start symbol
    - internal nodes : nonterminal
    - leaf nodes : terminal

○ Example G:

list -> list + digit | list - digit | digit

digit -> 0|1|2|3|4|5|6|7|8|9

- left most derivation for 9-5+2,

  list ⇒ list+digit ⇒ list-digit+digit ⇒ digit-digit+digit ⇒ 9-digit+digit ⇒ 9-5+digit ⇒ 9-5+2

  right most derivation for 9-5+2,

  list ⇒ list+digit ⇒ list+2 ⇒ list-digit+2 ⇒ list-5+2 ⇒ digit-5+2 ⇒ 9-5+2

parse tree for 9-5+2

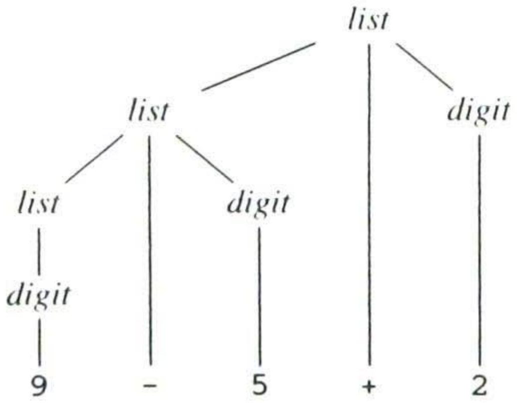


Fig 2.2. Parse tree for 9-5+2 according to the grammar in Example

Ambiguity

- A grammar is said to be ambiguous if the grammar has more than one parse tree for a given string of tokens.
- Example 2.5. Suppose a grammar G that can not distinguish between lists and digits as in Example 2.1.
    - G : string → string + string | string - string |0|1|2|3|4|5|6|7|8|9

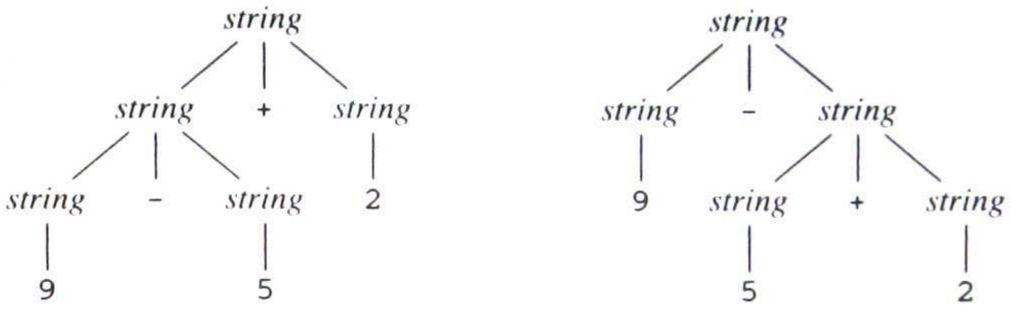


Fig 2.3. Two Parse tree for 9-5+2

- 1-5+2 has 2 parse trees => Grammar G is ambiguous.

Associativity of operator

A operator is said to be left associative if an operand with operators on both sides of it is taken by the operator to its left.

eg) 9+5+2≡(9+5)+2, a=b=c≡a=(b=c)

- Left Associative Grammar :

  list → list + digit | list - digit
  digit →0|1|…|9

- Right Associative Grammar :

  right → letter = right | letter
  letter → a|b|…|z

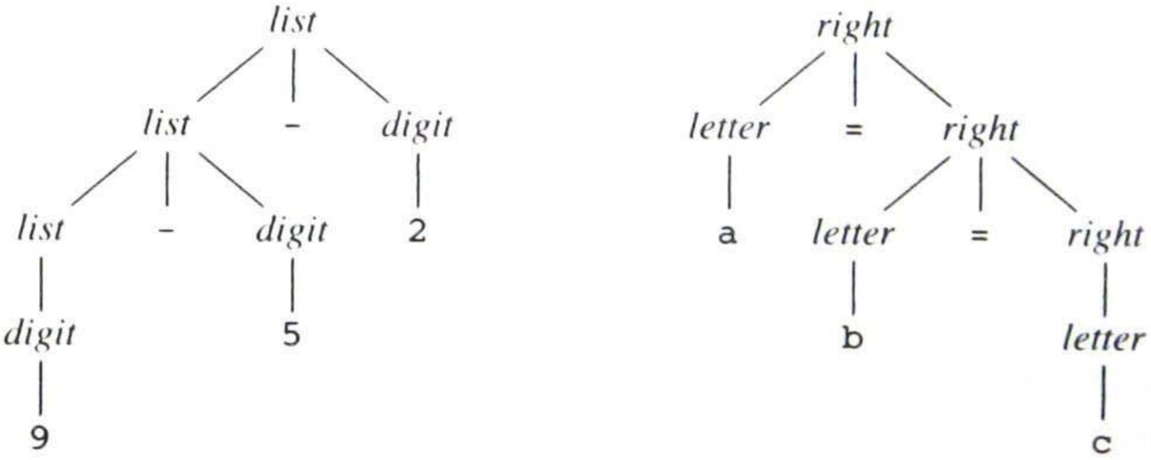


Fig 2.4. Parse tree left- and right-associative operators.

Precedence of operators

We say that a operator(*) has higher precedence than other operator(+) if the operator(*) takes operands before other operator(+) does.

- ex. 9+5*2≡9+(5*2), 9*5+2≡(9*5)+2
- left associative operators : + , - , * , /
- right associative operators : = , **

- Syntax of full expressions

operator	associative	precedence
+ , -	left	1 low
* , /	left	2 heigh

- *expr → expr + term | expr - term | term*
  *term → term * factor | term / factor | factor*
  factor → digit | (expr)
  digit → 0 | 1 | … | 9

- Syntax of statements
  - stmt → id = expr ;
    | if (expr) stmt ;
    | if (expr) stmt else stmt ;
    | while (expr) stmt ;

    expr → expr + term | expr - term | term
    term → term * factor | term / factor | factor
    factor → digit | (expr)
    digit → 0 | 1 | … | 9

2.3 SYNTAX-DIRECTED TRANSLATION(SDT)

A formalism for specifying translations for programming language constructs.
(attributes of a construct: type, string, location, etc)

- Syntax directed definition(SDD) for the translation of constructs
- Syntax directed translation scheme(SDTS) for specifying translation

Postfix notation for an expression E

- If E is a variable or constant, then the postfix nation for E is E itself (E.t≡E).
- if E is an expression of the form E1 op E2 where op is a binary operator
  - E1' is the postfix of E1,
  - E2' is the postfix of E2
  - then E1' E2' op is the postfix for E1 op E2
- if E is (E1), and E1' is a postfix
  then E1' is the postfix for E

eg) 9 - 5 + 2 ⇒ 9 5 - 2 +

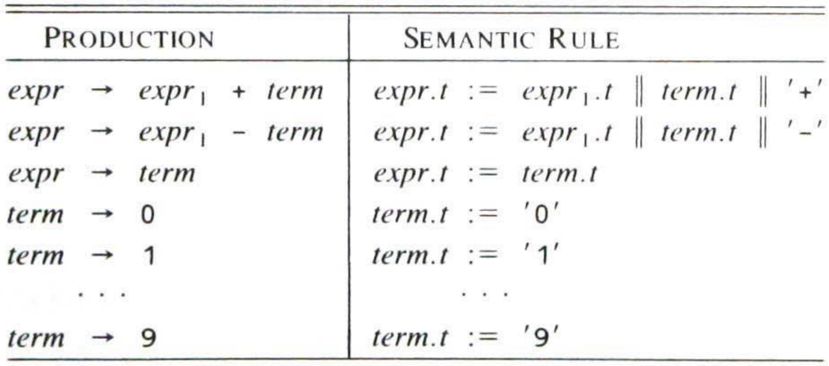
9 - (5 + 2) ⇒ 9 5 2 + -

Syntax-Directed Definition(SDD) for translation

- SDD is a set of semantic rules predefined for each productions respectively for translation.
- A translation is an input-output mapping procedure for translation of an input X,
  - construct a parse tree for X.
  - synthesize attributes over the parse tree.

- Suppose a node n in parse tree is labeled by X and X.a denotes the value of attribute a of X at that node.
- compute X's attributes X.a using the semantic rules associated with X.

Example 2.6. SDD for infix to postfix translation



PRODUCTION	SEMANTIC RULE
$expr \to expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$
$expr \to expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel '-'$
$expr \to term$	$expr.t := term.t$
$term \to 0$	$term.t := '0'$
$term \to 1$	$term.t := '1'$
. . .	. . .
$term \to 9$	$term.t := '9'$

Fig 2.5. Syntax-directed definition for infix to postfix translation.

An example of synthesized attributes for input X=9-5+2

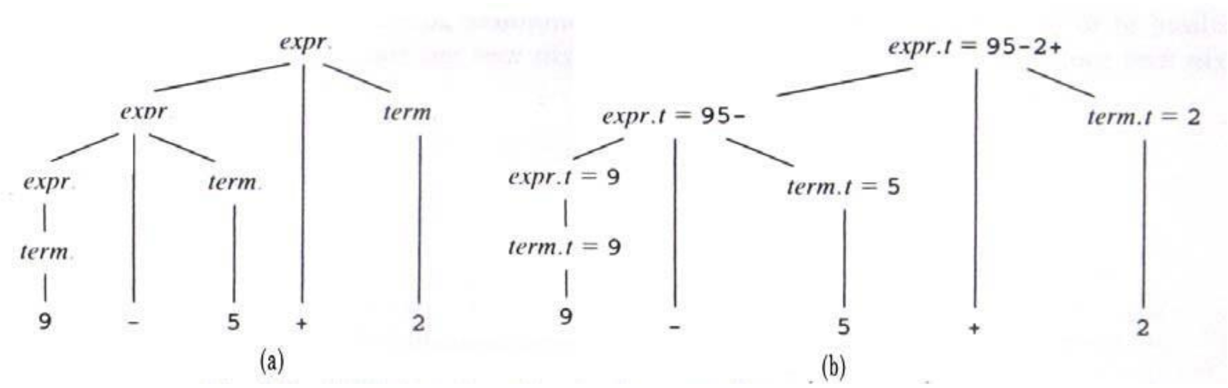


Fig 2.6. Attribute values at nodes in a parse tree.

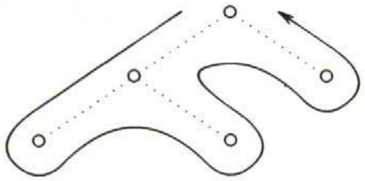
Syntax-directed Translation Schemes(SDTS)

- A translation scheme is a context-free grammar in which program fragments called translation actions are embedded within the right sides of the production.

productions(postfix)	SDD for postfix to infix notation	SDTS
list → list + term list.t = li	t.t || term.t || "+" list → lis	+ term {print("+")}

- {print("+");} : translation(semantic) action.
- SDTS generates an output for each sentence x generated by underlying grammar by executing actions in the order they appear during depth-first traversal of a parse tree for x.

1. Design translation schemes(SDTS) for translation
2. Translate :
   a) parse the input string x and
   b) emit the action result encountered during the depth-first traversal of parse tree.



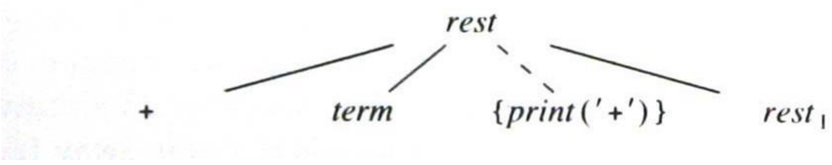


Fig 2.7. Example of a depth-first traversal of a tree. **Fig 2.8.** An extra leaf is constructed for a semantic action.

Example 2.8.

- SDD vs. SDTS for infix to postfix translation.

productions	SDD	SDTS

expr → list + term
expr → list + term
expr → term
term → 0
term → 1
…
term → 9
	expr.t = list.t || term.t || "+"
	expr.t = list.t || term.t || "-"
	expr.t = term.t
	term.t = "0"
	term.t = "1"
	…
	term.t = "9"
		expr → list + term printf{"+")}
		expr → list + term printf{"-")}
		expr → term
		term → 0 printf{"0")}
		term → 1 printf{"1")}
		…
		term → 9 printf{"0")}

- Action translating for input 9-5+2

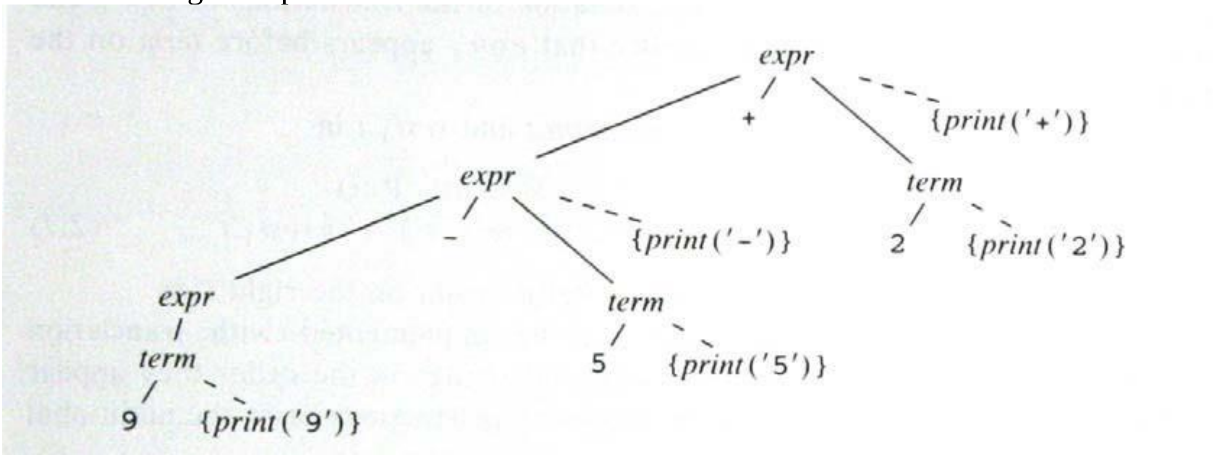


Fig 2.9. Actions translating 9-5+2 into 95-2+.

1) Parse.
2) Translate.
Do we have to maintain the whole parse tree ?
No, Semantic actions are performed during parsing, and we don't need the nodes (whose semantic actions done).

2.4 PARSING

if token string x ∈ L(G), then parse tree
else error message

Top-Down parsing

1. At node n labeled with nonterminal A, select one of the productions whose left part is A and construct children of node n with the symbols on the right side of that production.
2. Find the next node at which a sub-tree is to be constructed.

ex. G: type → simple
|↑id
|array [simple] of type
simple → integer
|char
|num dotdot num

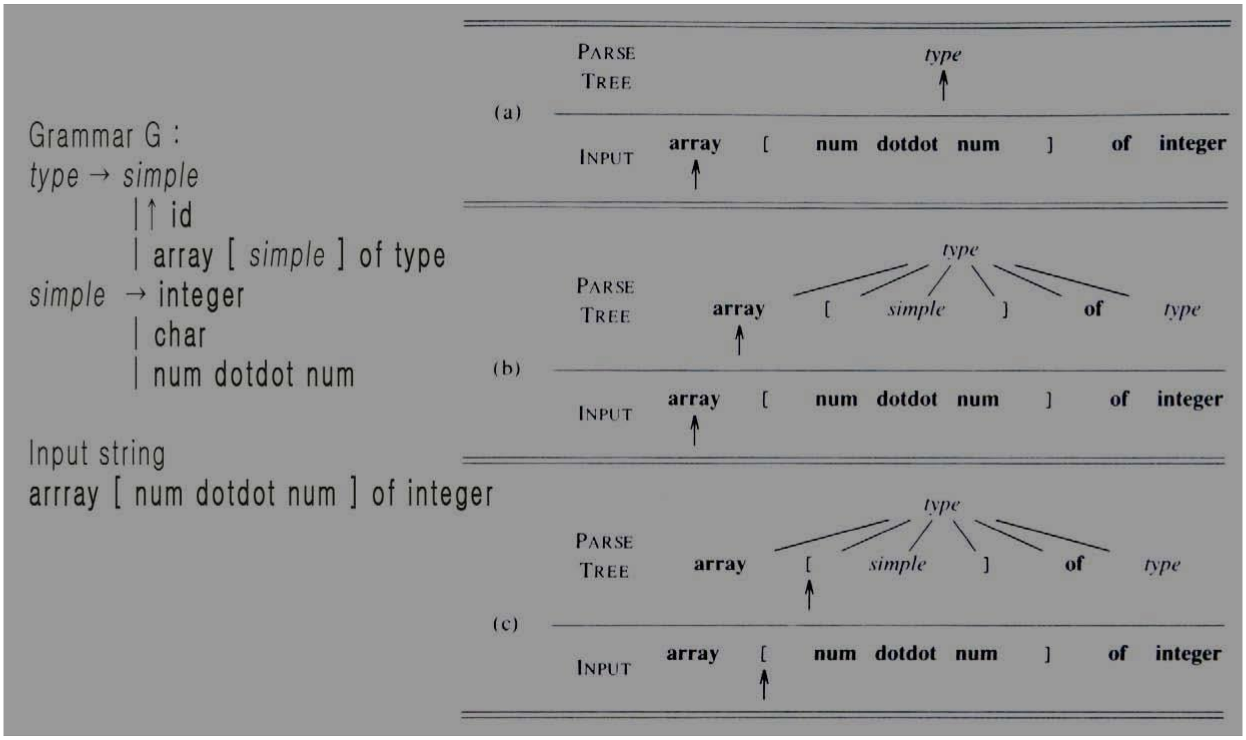


Fig 2.10. Top-down parsing while scanning the input from left to right.

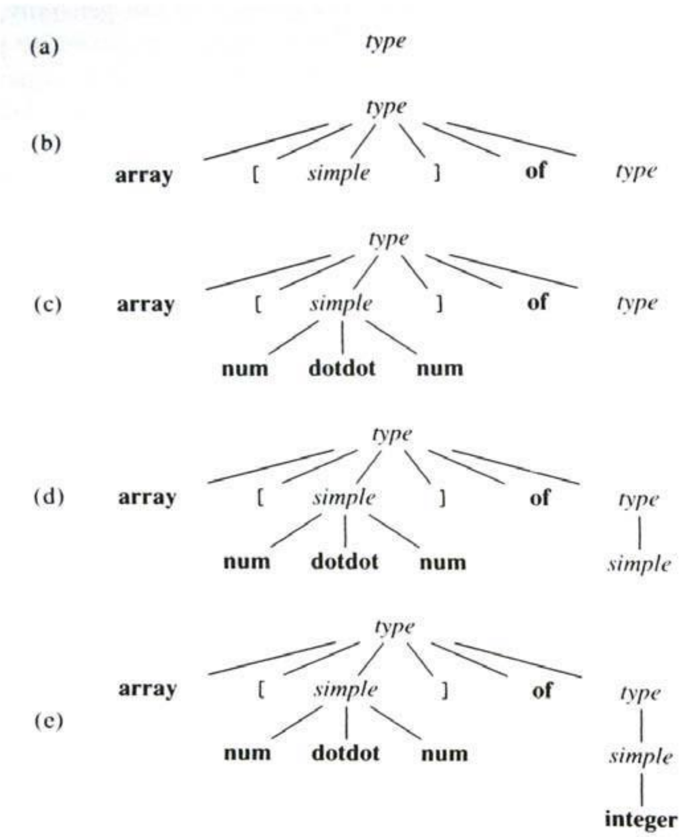


Fig 2.11. Steps in the top-down construction of a parse tree.

- The selection of production for a nonterminal may involve trial-and-error. => backtracking

- G : { S->aSb | c | ab }
  According to topdown parsing procedure, acb , aabb∈L(G)?

- S/acb⇒aSb/acb⇒aSb/acb⇒aaSbb/acb ⇒ X

  (S→aSb) move (S→aSb) backtracking

  ⇒aSb/acb⇒acb/acb⇒acb/acb⇒acb/acb

  (s→c) move move

  so, acb∈ L(G)

  Is is finished in 7 steps including one backtracking.

- S/aabb⇒aSb/aabb⇒aSb/aabb⇒aaSbb/aabb⇒aaSbb/aabb⇒aaaSbbb/aabb ⇒ X

  (S→aSb) move (S→aSb) move (S→aSb) backtracking

  ⇒aaSbb/aabb⇒aacbb/aabb ⇒ X

  (S→c) backtracking

  ⇒aaSbb/aabb⇒aaabbb/aabb⇒ X

  (S→ab) backtracking

  ⇒aaSbb/aabb⇒ X

  backtracking

  ⇒aSb/aabb⇒acb/aabb

  (S→c) bactracking

  ⇒aSb/aabb⇒aabb/aabb⇒aabb/aabb⇒aabb/aabb⇒aaba/aabb

  (S→ab) move move move

  so, aabb∈L(G)

  but process is too difficult. It needs 18 steps including 5 backtrackings.

- procedure of top-down parsing

  let a pointed grammar symbol and pointed input symbol be g, a respectively.

  - if(g ∈ N) select and expand a production whose left part equals to g next to current production.

    else if(g = a) then make g and a be a symbol next to current symbol.

    else if(g ≠a) back tracking

    - let the pointed input symbol a be the symbol that moves back to steps same with the number of current symbols of underlying production
    - eliminate the right side symbols of current production and let the pointed symbol g be the left side symbol of current production.

Predictive parsing (Recursive Decent Parsing,RDP)

- A strategy for the general top-down parsing

  Guess a production, see if it matches, if not, backtrack and try another.

  ⇒
- It may fail to recognize correct string in some grammar G and is tedious in processing.

  ⇒
- **Predictive parsing**
  - is a kind of top-down parsing that predicts a production whose derived terminal symbol is equal to next input symbol while expanding in top-down paring.
  - without backtracking.
  - Procedure decent parser is a kind of predictive parser that is implemented by disjoint recursive procedures one procedure for each nonterminal, the procedures are patterned after the productions.
- procedure of predictive parsing(RDP)

  let a pointed grammar symbol and pointed input symbol be g, a respectively.

  - if(g ∈ N)
    - select next production P whose left symbol equals to g and a set of first terminal symbols of derivation from the right symbols of the production P includes a input symbol a.
    - expand derivation with that production P.
  - else if(g = a) then make g and a be a symbol next to current symbol.
  - else if(g ≠a) error

- G : { S→aSb | c | ab } => G1 : { S->aS' | c S'->Sb | ab }

  According to predictive parsing procedure, acb , aabb∈L(G)?

  - S/acb⇒ confused in { S→aSb, S→ab }
  - so, a predictive parser requires some restriction in grammar, that is, there should be only one production whose left part of productions are A and each first terminal symbol of those productions have unique terminal symbol.
- Requirements for a grammar to be suitable for RDP: For each nonterminal either
  1. A → Bα, or
  2. A → a1α1 | a2α2 | … | anαn
     1) for 1 ≦ i, j ≦ n and i≠ j, ai ≠ aj
     2) A ε may also occur if none of ai can follow A in a derivation and if we have A→ε

- If the grammar is suitable, we can parse efficiently without backtrack.

  General top-down parser with backtracking
  ↓
  Recursive Descent Parser without backtracking
  ↓
  Picture Parsing (a kind of predictive parsing) without backtracking

Left Factoring

- If a grammar contains two productions of form

  S→ aα and S → aβ

  it is not suitable for top down parsing without backtracking. Troubles of this form can sometimes be removed from the grammar by a technique called the left factoring.
- In the left factoring, we replace { S→ aα, S→ aβ } by

  { S → aS', S'→ α, S'→ β } cf. S→ a(α|β)
  (Hopefully α and β start with different symbols)
- left factoring for G { S→aSb | c | ab }

  S→aS' | c   cf. S(=aSb | ab | c = a (Sb | b) | c) → a S' | c
  S'→Sb | b
- A concrete example:

  <stmt> →   IF <boolean> THEN <stmt> |
              IF <boolean> THEN <stmt> ELSE <stmt>

  is transformed into

  <stmt>→   IF <boolean> THEN <stmt> S'
  S' →   ELSE <stmt> | ε

- **Example**,
  - for G1 : { S→aSb | c | ab }

    According to predictive parsing procedure, acb , aabb∈L(G)?
    - S/aabb⇒ unable to choose { S→aSb, S→ab ?}
  - According for the left factored gtrammar G1, acb , aabb∈L(G)?

    G1 : { S→aS'|c S'→Sb|b} <= {S=a(Sb|b) | c }
  - S/acb⇒aS'/acb⇒aS'/acb ⇒ aSb/acb ⇒ acb/acb ⇒ acb/acb⇒ acb/acb
    (S→aS')   move   (S'→Sb⇒aS'b) (S'→c)   move   move

    so, acb∈ L(G)
    It needs only 6 steps whithout any backtracking.
    cf. General top-down parsing needs 7 steps and I backtracking.
  - S/aabb⇒aS'/aabb⇒aS'/aabb⇒aSb/aabb⇒aaS'b/aabb⇒aaS'b/aabb⇒aabb/aabb⇒ ⇒
    (S→aS')   move   (S'→Sb⇒aS'b)   (S'→aS')   move   (S'→b)   move move

    so, aabb∈L(G)
    but, process is finished in 8 steps without any backtracking.
    cf. General top-down parsing needs 18 steps including 5 backtrackings.

Left Recursion

- A grammar is left recursive iff it contains a nonterminal A, such that

  A⇒+ Aα, where is any string.
  - Grammar {S→ Sα | c} is left recursive because of S⇒Sα
  - Grammar {S→ Aα, A→ Sb | c} is also left recursive because of S⇒Aα⇒ Sbα
- If a grammar is left recursive, you cannot build a predictive top down parser for it.

1) If a parser is trying to match S & $S \rightarrow S\alpha$, it has no idea how many times S must be applied
2) Given a left recursive grammar, it is always possible to find another grammar that generates the same language and is not left recursive.
3) The resulting grammar might or might not be suitable for RDP.

- After this, if we need left factoring, it is not suitable for RDP.
- Right recursion: Special care/Harder than left recursion/SDT can handle.

Eliminating Left Recursion

Let G be $S \rightarrow S\ A \mid A$

Note that a top-down parser cannot parse the grammar G, regardless of the order the productions are tried.

⇒ The productions generate strings of form AA⋯A

⇒ They can be replaced by $S \rightarrow A\ S'$ and $S' \rightarrow A\ S' \mid \varepsilon$

Example :

- $A \rightarrow A\alpha \mid \beta$

  =>

  $A \rightarrow \beta R$

  $R \rightarrow \alpha R \mid \varepsilon$

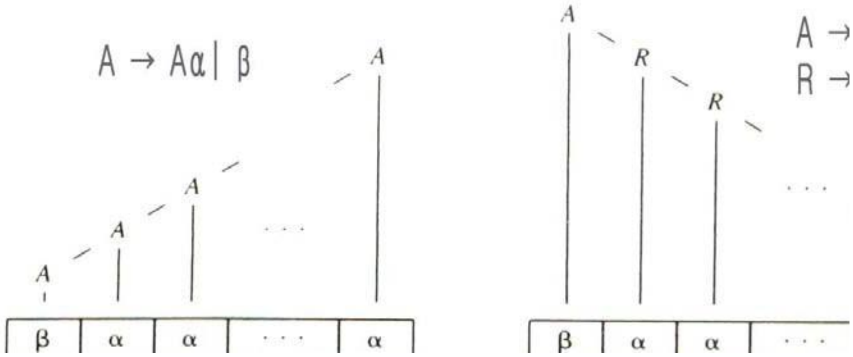


Fig 2.12. Left-and right-recursive ways of generating a string.

- In general, the rule is that
  - If $A \rightarrow A\alpha 1 \mid A\alpha 2 \mid \ldots \mid A\alpha n$ and
    $A \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta m$ (no βi's start with A),
    then, replace by
    $A \rightarrow \beta 1R \mid \beta 2R \mid \ldots \mid \beta mR$ and
    $Z \rightarrow \alpha 1R \mid \alpha 2R \mid \ldots \mid \alpha nR \mid \varepsilon$

Exercise: Remove the left recursion in the following grammar:

expr → expr + term | expr - term

expr → term

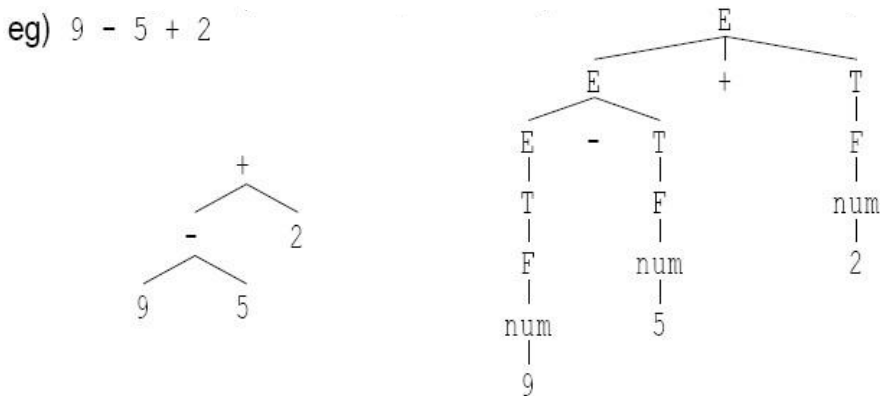
solution:

expr → term rest

rest → + term rest | - term rest | ε

2.5 A TRANSLATOR FOR SIMPLE EXPRESSIONS

- Convert infix into postfix(polish notation) using SDT.
- Abstract syntax (annotated parse tree) tree vs. Concrete syntax tree



- Concrete syntax tree : parse tree.
- Abstract syntax tree: syntax tree
- Concrete syntax : underlying grammar

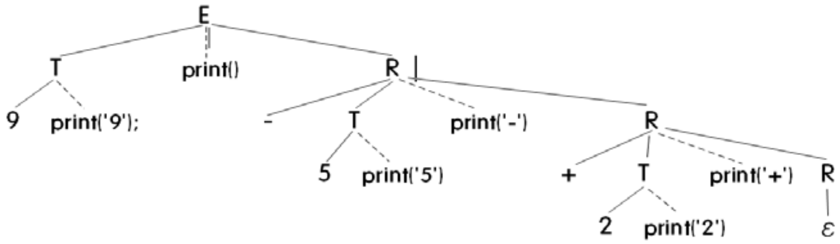
Adapting the Translation Scheme

- Embed the semantic action in the production
- Design a translation scheme
- Left recursion elimination and Left factoring
- Example

3) Design a translate scheme and eliminate left recursion

E→ E + T {'+'}	E→ T {} R
E→ E - T {'-'}	R→ + T{'+'} R
E→ T {}	R→ - T{'-'} R
T→ 0{'0'}|…|9{'9'}	R→ ε
	T→ 0{'0'}…|9{'9'}

4)Translate of a input string 9-5+2 : parsing and SDT



Result: 9 5 – 2 +

Example of translator design and execution

- A translation scheme and with left-recursion.

Initial specification for infix-to-postfix translator	with left recursion eliminated

expr → expr + term {printf{"+")}
expr → expr - term {printf{"-")}
expr → term
term → 0 {printf{"0")}
term → 1 {printf{"1")}
…
term → 9 {printf{"0")}
	expr → term rest
	rest → + term {printf{"+")} rest
	rest → - term {printf{"-")} rest
	rest → ε
	term → 0 {printf{"0")}
	term → 1 {printf{"1")}
	…
	term → 9 {printf{"0")}

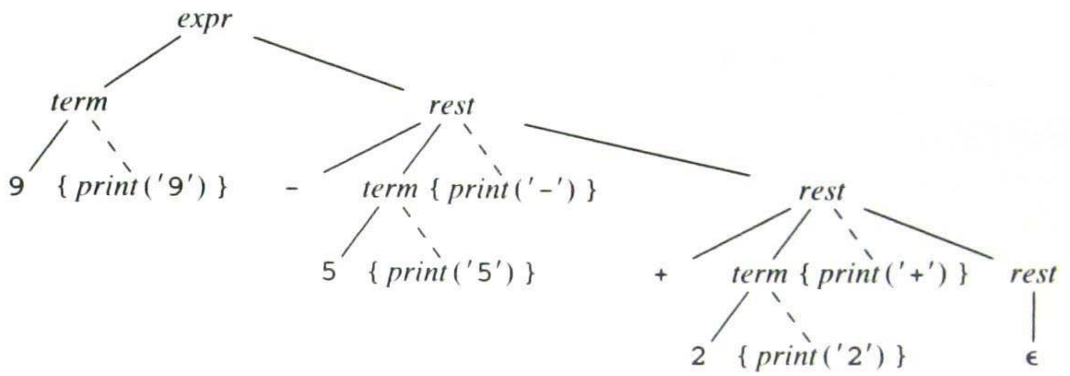
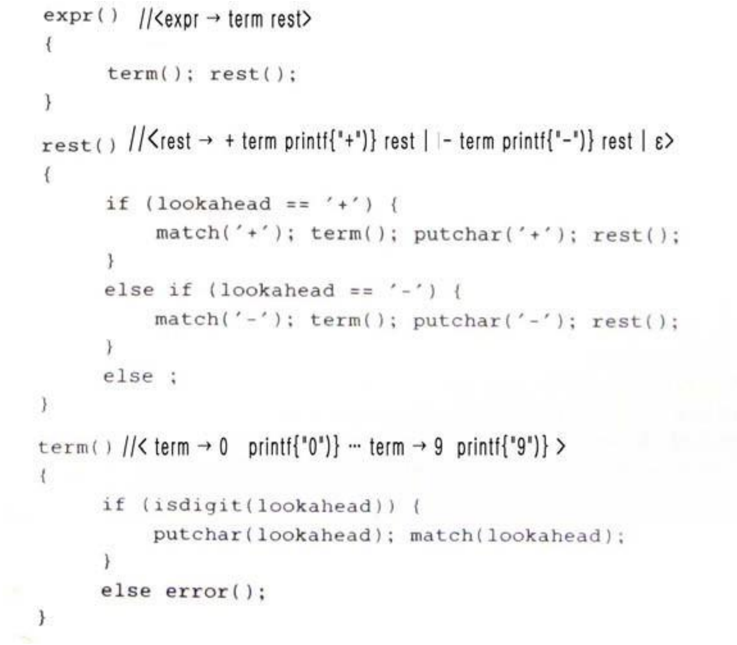


Fig 2.13. Translation of 9 – 5 +2 into 95-2+.

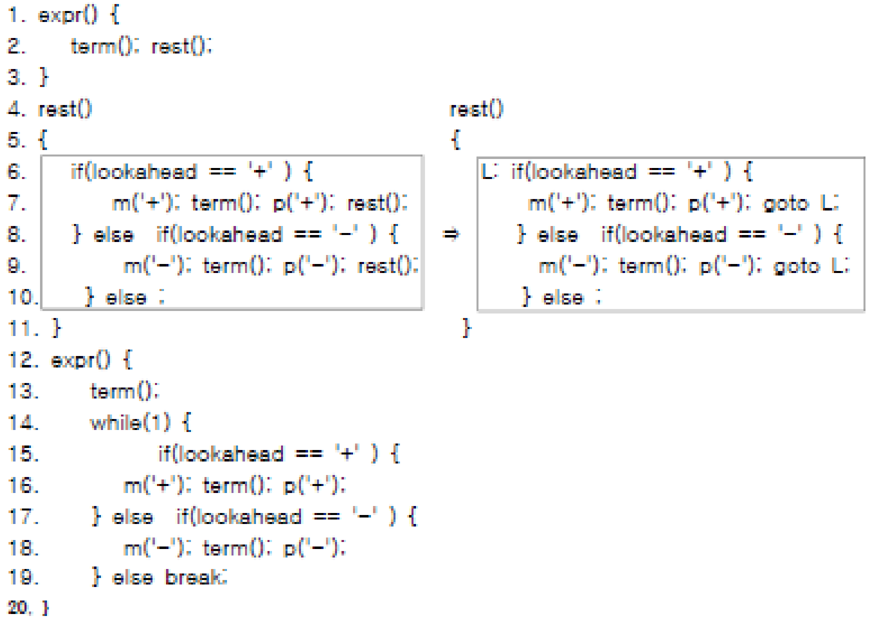
Procedure for the Nonterminal expr, term, and rest



```
expr( )  //<expr → term rest>
{
      term(); rest();
}
rest( ) //<rest →  + term printf{"+")} rest | - term printf{"-")} rest | ε>
{
      if (lookahead == '+') {
          match('+'); term(); putchar('+'); rest();
      }
      else if (lookahead == '-') {
          match('-'); term(); putchar('-'); rest();
      }
      else ;
}
term( ) //< term → 0   printf{"0")} ··· term → 9  printf{"9")} >
{
      if (isdigit(lookahead)) {
          putchar(lookahead); match(lookahead);
      }
      else error();
}
```

Fig 2.14. Function for the nonterminals expr, rest, and term.

Optimizer and Translator



```
expr() {
    term(); rest();
}
rest()                                      rest()
{                                           {
    if(lookahead == '+' ) {                 L: if(lookahead == '+' ) {
        m('+'); term(); p('+'); rest();          m('+'); term(); p('+'); goto L;
    } else  if(lookahead == '-' ) {      ⇒     } else  if(lookahead == '-' ) {
         m('-'); term(); p('-'); rest();          m('-'); term(); p('-'); goto L;
      } else ;                                  } else ;
}                                             }
expr() {
    term();
    while(1) {
            if(lookahead == '+' ) {
        m('+'); term(); p('+');
    } else  if(lookahead == '-' ) {
        m('-'); term(); p('-');
    } else break;
}
```

2.6 LEXICAL ANALYSIS

- reads and converts the input into a stream of tokens to be analyzed by parser.
- lexeme : a sequence of characters which comprises a single token.
- Lexical Analyzer →Lexeme / Token → Parser

Removal of White Space and Comments

- Remove white space(blank, tab, new line etc.) and comments

Contsants

- Constants: For a while, consider only integers
- eg) for input 31 + 28, output(token representation)?

    input : 31 + 28

    output: <num, 31> <+, > <num, 28>

        num + :token

        31 28 : attribute, value(or lexeme) of integer token num

Recognizing

- Identifiers
    - Identifiers are names of variables, arrays, functions...
    - A grammar treats an identifier as a token.
    - eg) input : count = count + increment;

      output : <id,1> <=, > <id,1> <+, > <id, 2>;

      Symbol table

	tokens	attributes(lexeme)
0		
1	id	count
2	id	increment
3		

- Keywords are reserved, i.e., they cannot be used as identifiers.

Then a character string forms an identifier only if it is no a keyword.

- punctuation symbols
  - operators : + - * / := < > …

Interface to lexical analyzer

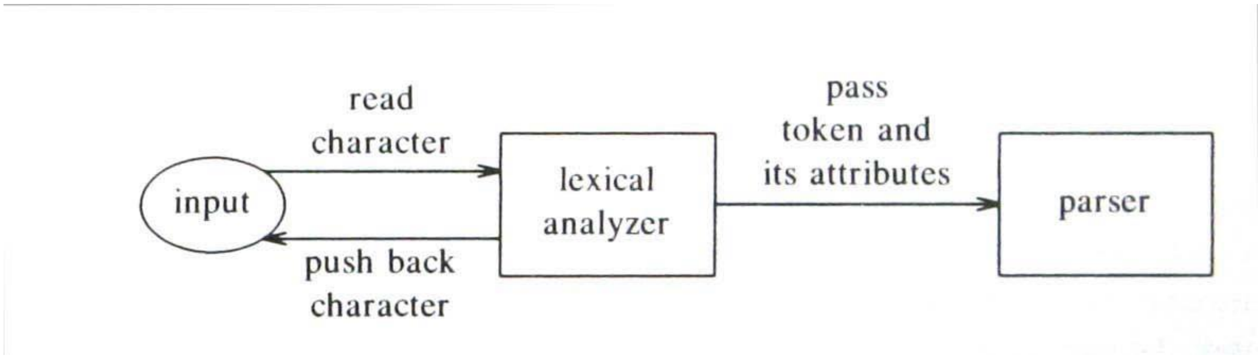


Fig 2.15. Inserting a lexical analyzer between the input and the parser

A Lexical Analyzer

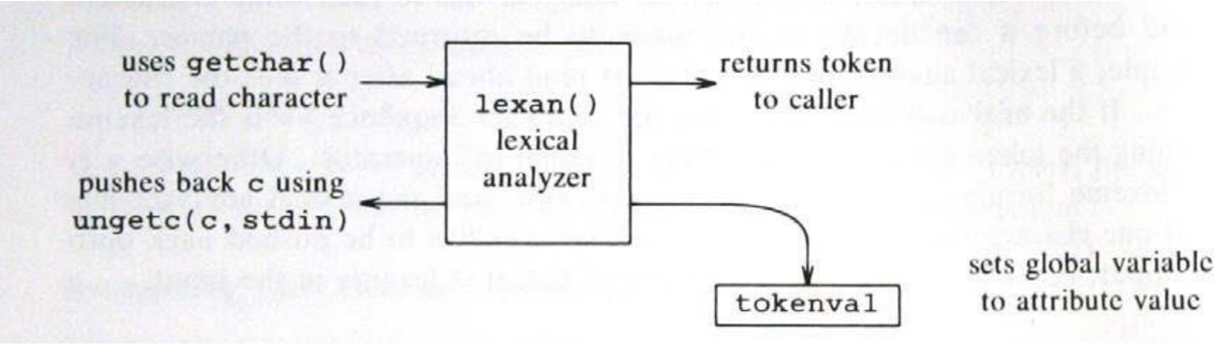


Fig 2.16. Implementing the interactions in Fig. 2.15.

- c=getchcar(); ungetc(c,stdin);
- token representation
  - #define NUM 256
- Function lexan()

  eg) input string 76 + a

  input , output(returned value)

  76 NUM, tokenval=76 (integer)

  + +

  A id , tokeval="a"

- A way that parser handles the token NUM returned by laxan()
  - consider a translation scheme

    factor → (expr)

    | num { print(num.value) }

    #define NUM 256

```
...
factor() {
        if(lookahead == '(' ) {
        match('('); exor(); match(')');
    } else if (lookahead == NUM) {
        printf(" %f ",tokenval); match(NUM);
    } else error();
}
```

- The implementation of function lexan

```
#include <stdio.h>
#include <ctype.h>
int lino = 1;
int tokenval = NONE;
int lexan() {
        int t;
        while(1) {
                t = getchar();
                if ( t==' ' || t=='\t' ) ;
                else if ( t=='\n' ) lineno +=1;
                else if (isdigit(t)) {
                        tokenval = t -'0';
                        t = getchar();
                        while ( isdigit(t)) {
                                        tokenval = tokenval*10 + t - '0';
                                t =getchar();
                               }
                               ungetc(t,stdin);
                               retunr NUM;
                } else {
                        tokenval = NONE;
                        return t;
                       }
               }
        }
```

2.7 INCORPORATION A SYMBOL TABLE

- The symbol table interface, operation, usually called by parser.
    - insert(s,t): input s: lexeme
      t: token
      output index of new entry
    - lookup(s): input s: lexeme
      output index of the entry for string s, or 0 if s is not found in the symbol table.
- Handling reserved keywords
    1. Inserts all keywords in the symbol table in advance.
       ex) insert("div", div)

insert("mod", mod)

2. while parsing

- whenever an identifier s is encountered.
  if (lookup(s)'s token in {keywords}) s is for a keyword; else s is for a identifier;

- example
  - ○ preset
    insert("div",div);
    insert("mod",mod);
  - ○ while parsing
    lookup("count")=>0 insert("count",id);
    lookup("i") =>0 insert("i",id);
    lookup("i") =>4, id
    llokup("div")=>1,div

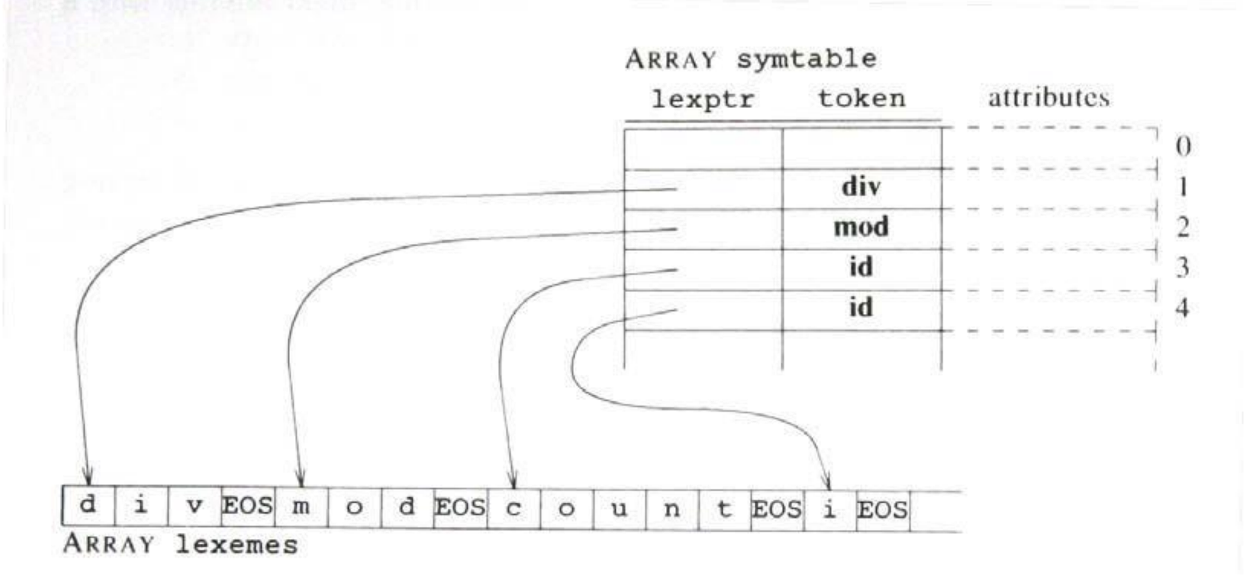
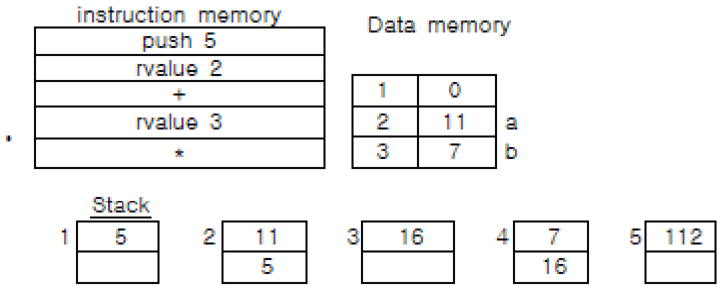


Fig 2.17. Symbol table and array for storing strings.

2.8 ABSTRACT STACK MACHINE

- ○ An abstract machine is for intermediate code generation/execution.
- ○ Instruction classes: arithmetic / stack manipulation / control flow

- 3 components of abstract stack machine
  1) Instruction memory : abstract machine code, intermediate code(instruction)
  2) Stack
  3) Data memory
- An example of stack machine operation.
  - ○ for a input (5+a)*b, intermediate codes : push 5 rvalue 2 ....



L-value and r-value

- l-values a : address of location a
- r-values a : if a is location, then content of location a
  if a is constant, then value a
- eg) a :=5 + b;
  lvalue a⇒2 r value 5 ⇒ 5 r value of b ⇒ 7

Stack Manipulation

- Some instructions for assignment operation
  - ○ push v : push v onto the stack.
  - ○ rvalue a : push the contents of data location a.
  - ○ lvalue a : push the address of data location a.
  - ○ pop : throw away the top element of the stack.
  - ○ := : assignment for the top 2 elements of the stack.
  - ○ copy : push a copy of the top element of the stack.

Translation of Expressions

- Infix expression(IE) ⟶ SDD/SDTS ⟶ Abstact macine codes(ASC) of postfix expression for stack machine evaluation.
  eg)
  - ○ IE: a + b, (⇒PE: a b +) ⇒ IC: rvalue a
    rvalue b
    +
  - ○ day := (1461 * y) div 4 + (153 * m + 2) div 5 + d
    (⇒ day 1462 y * 4 div 153 m * 2 + 5 div + d + :=)
    ⇒1) lvalue day 6) div 11) push 5 16) :=
    2) push 1461 7) push 153 12) div
    3) rvalue y 8) rvalue m 13) +
    4) * 9) push 2 14) rvalue d
    5) push 4 10) + 15) +
- A translation scheme for assignment-statement into abstract astack machine code e can be expressed formally In the form as follows:
  stmt ⟶ id := expr
  { stmt.t := 'lvalue' || id.lexeme || expr.t || ':=' }
  eg) day :=a+b ⇒ lvalue day rvalue a rvalue b + :=

Control Flow

- 3 types of jump instructions :
    - Absolute target location
    - Relative target location(distance :Current ↔Target)
    - Symbolic target location(*i.e.* the machine supports labels)
- Control-flow instructions:
    - **label a**: the jump's target a
    - **goto a**: the next instruction is taken from statement labeled a
    - **gofalse a**: pop the top & if it is 0 then jump to a
    - **gotrue a**: pop the top & if it is nonzero then jump to a
    - **halt** : stop execution

Translation of Statements

- Translation scheme for translation if-statement into abstract machine code.

  stmt → if expr then stmt1
  { out := newlabel1)
  stmt.t := expr.t || 'gofalse' out || stmt1.t || 'label' out }

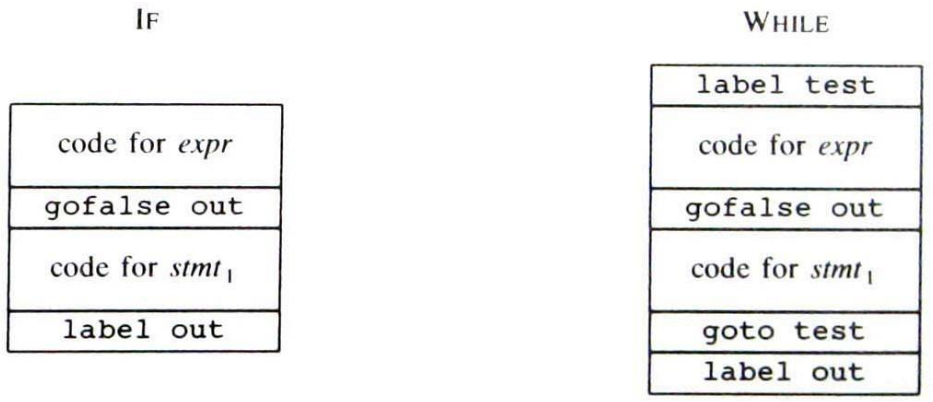


Fig 2.18. Code layout for conditional and while statements.

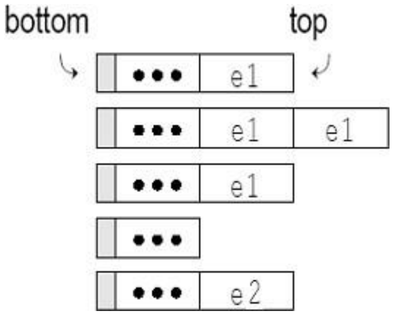
- Translation scheme for while-statement ?

Emitting a Translation

- Semantic Action(Tranaslation Scheme):

1. stmt → if
   expr { out := newlabel; emit('gofalse', out) }
   then
   stmt1 { emit('label', out) }
2. stmt → id { emit('lvalue', id.lexeme) }
   :=
   expr { emit(':=') }
3. stmt → i
   expr { out := newlabel; emit('gofalse', out) }
   then
   stmt1 { emit('label', out) ; out1 := newlabel; emit('goto', out`1); }

```
          else
          stmt2 { emit('label', out1) ; }
          if(expr==false) goto out
          stmt1 goto out1
     out : stmt2
     out1:
```



Implementation

- procedure stmt()
- var test,out:integer;
- begin
    - if lookahead = id then begin
        - emit('lvalue',tokenval); match(id);
          match(':='); expr(); emit(':=');
    - end
    - else if lookahead = 'if' then begin
        - match('if');
        - expr();
        - out := newlabel();
        - emit('gofalse', out);
        - match('then');
        - stmt;
        - emit('label', out)
    - end
    - else error();
- end

Control Flow with Analysis

- if E1 or E2 then S vs if E1 and E2 then S

  E1 or E2 = if E1 then true else E2

  E1 and E2 = if E1 then E2 else false

- **The code for E1 or E2.**
    - Codes for E1 Evaluation result: e1
    - copy
    - gotrue OUT
    - pop
    - Codes for E2 Evaluation result: e2
    - label OUT

- The full code for **if E1 or E2 then S ;**
    - codes for E1
    - copy
    - gotrue OUT1
    - pop
    - codes for E2
    - label OUT1

- ○ gofalse OUT2
- ○ code for S
- ○ label OUT2

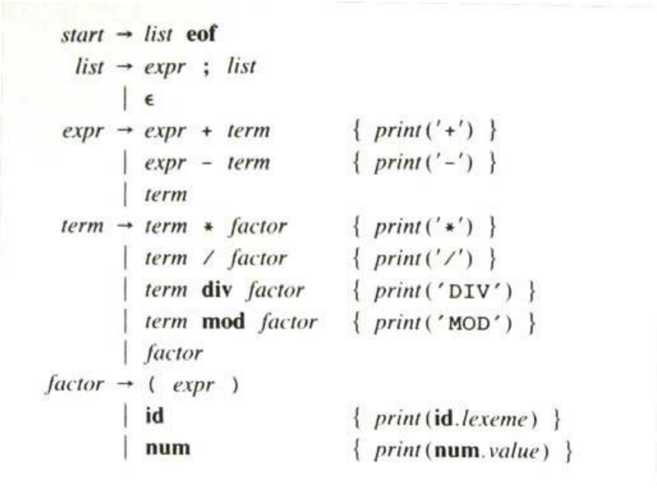
- Exercise: How about if E1 and E2 then S;
  - ○ if E1 and E2 then S1 else S2;

2.9 Putting the techniques together!

- infix expression ⇒ postfix expression
  eg) id+(id-id)*num/id ⇒ id id id - num * id / +

Description of the Translator

- Syntax directed translation scheme (SDTS) to translate the infix expressions into the postfix expressions,



```
   start → list eof
    list → expr ; list
         | ε
    expr → expr + term       { print('+') }
         | expr - term       { print('-') }
         | term
    term → term * factor     { print('*') }
         | term / factor     { print('/') }
         | term div factor   { print('DIV') }
         | term mod factor   { print('MOD') }
         | factor
  factor → ( expr )
         | id                { print(id.lexeme) }
         | num               { print(num.value) }
```

Fig 2.19. Specification for infix-to-postfix translation

Structure of the translator,

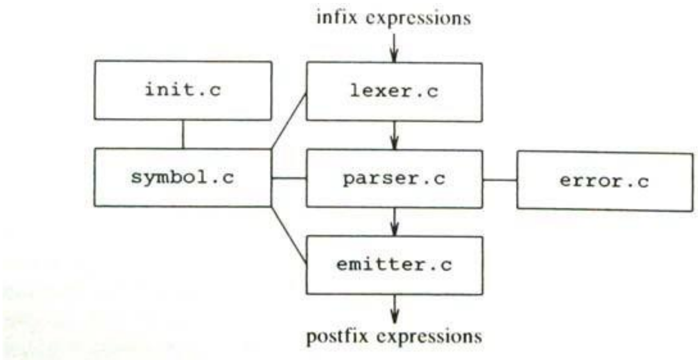
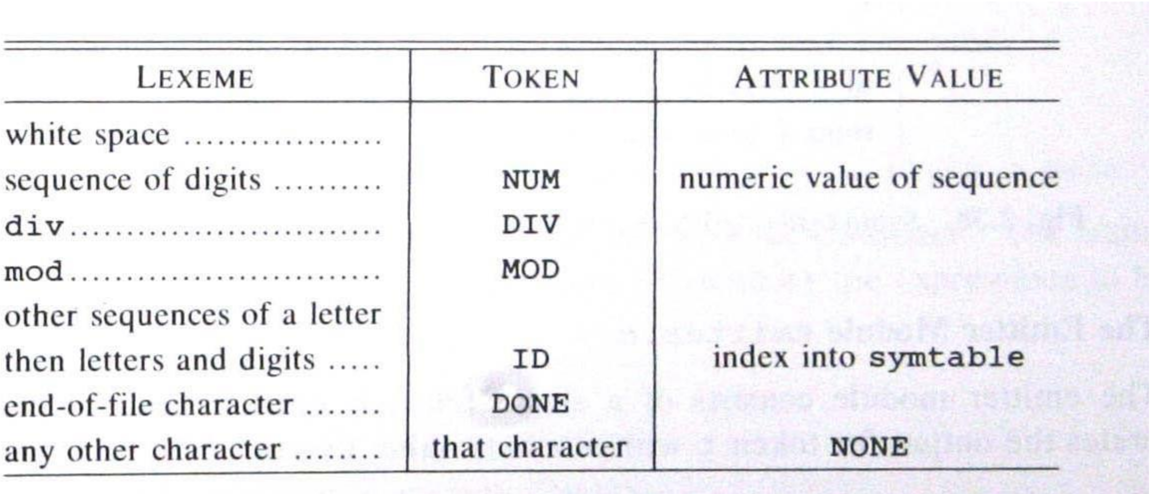


Fig 2.19. Modules of infix to postfix translator.

- ○ global header file "header.h"

The Lexical Analysis Module lexer.c

- ○ Description of tokens
  + - * / DIV MOD ( ) ID NUM DONE

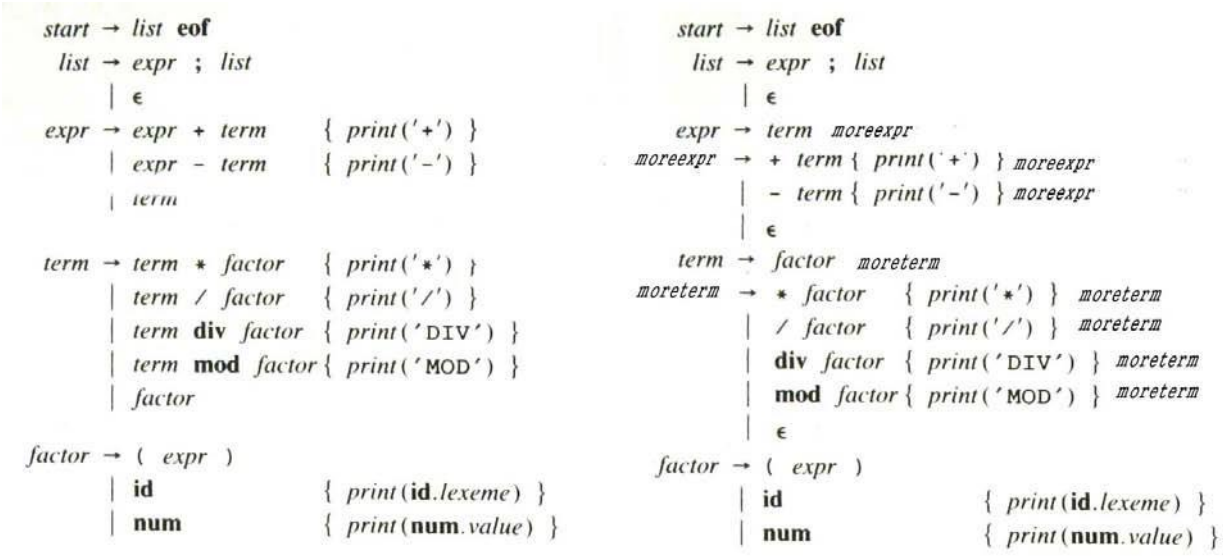


LEXEME	TOKEN	ATTRIBUTE VALUE
white space		
sequence of digits	NUM	numeric value of sequence
div	DIV	
mod	MOD	
other sequences of a letter then letters and digits	ID	index into symtable
end-of-file character	DONE	
any other character	that character	NONE

Fig 2.20. Description of tokens.

The Parser Module parser.c

SDTS
|| ← left recursion elimination
New SDTS



```
start → list eof
 list → expr ; list
      | ϵ
 expr → expr + term       { print('+') }
      | expr - term       { print('-') }
      | term

 term → term * factor     { print('*') }
      | term / factor     { print('/') }
      | term div factor   { print('DIV') }
      | term mod factor   { print('MOD') }
      | factor

factor → ( expr )
      | id                { print(id.lexeme) }
      | num               { print(num.value) }
```

```
    start → list eof
     list → expr ; list
          | ϵ
     expr → term moreexpr
 moreexpr → + term { print('+') } moreexpr
          | - term { print('-') } moreexpr
          | ϵ
     term → factor moreterm
 moreterm → * factor    { print('*') }   moreterm
          | / factor    { print('/') }   moreterm
          | div factor  { print('DIV') } moreterm
          | mod factor  { print('MOD') } moreterm
          | ϵ
   factor → ( expr )
          | id          { print(id.lexeme) }
          | num         { print(num.value) }
```

Fig 2.20. Specification for infix to postfix translator & syntax directed translation scheme after eliminating left-recursion.

The Emitter Module emitter.c

emit (t,tval)

The Symbol-Table Modules symbol.c and init.c

Symbol.c

data structure of symbol table Fig 2.29 p62

insert(s,t)

lookup(s)

The Error Module error.c

Example of execution

```
        input 12 div 5 + 2
        output 12
                5
                div
                2
                +
```

3. Lexical Analysis:

3.1 OVER VIEW OF LEXICAL ANALYSIS

- To identify the tokens we need some method of describing the possible tokens that can appear in the input stream. For this purpose we introduce regular expression, a notation that can be used to describe essentially all the tokens of programming language.
- Secondly , having decided what the tokens are, we need some mechanism to recognize these in the input stream. This is done by the token recognizers, which are designed using transition diagrams and finite automata.

3.2 ROLE OF LEXICAL ANALYZER

The LA is the first phase of a compiler. It main task is to read the input character and produce as output a sequence of tokens that the parser uses for syntax analysis.

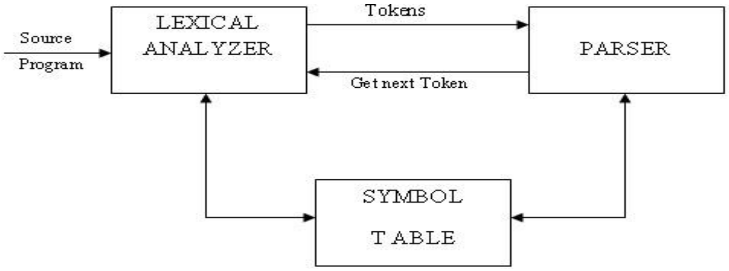


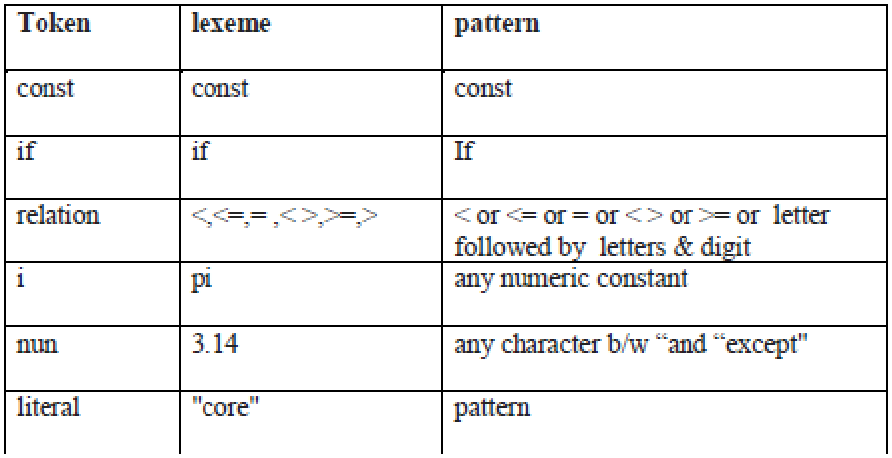
Fig. 3.1: Role of Lexical analyzer

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

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

3.3 TOKEN, LEXEME, PATTERN:

Token: Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are,
1) Identifiers 2) keywords 3) operators 4) special symbols 5)constants
Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.
Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.



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

Fig. 3.2: Example of Token, Lexeme and Pattern

3.4. LEXICAL ERRORS:

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

Error-recovery actions are:

i. Delete one character from the remaining input.

ii. Insert a missing character in to the remaining input.

iii. Replace a character by another character.

iv. Transpose two adjacent characters.

3.5. REGULAR EXPRESSIONS

Regular expression is a formula that describes a possible set of string. Component of regular expression..

X	**the character x**
.	**any character, usually accept a new line**
[x y z]	**any of the characters x, y, z, …..**
R?	**a R or nothing (=optionally as R)**
R*	**zero or more occurrences…..**
R+	**one or more occurrences ……**
R1R2	**an R1 followed by an R2**
R1|R1	**either an R1 or an R2.**

A token is either a single string or one of a collection of strings of a certain type. If we view the set of strings in each token class as an language, we can use the regular-expression notation to describe tokens.

Consider an identifier, which is defined to be a letter followed by zero or more letters or digits. In regular expression notation we would write.

Identifier = letter (letter | digit)*

Here are the rules that define the regular expression over alphabet .

- is a regular expression denoting { € }, that is, the language containing only the empty string.
- For each 'a' in Σ, is a regular expression denoting { a }, the language with only one string consisting of the single symbol 'a' .
- If R and S are regular expressions, then

(R) | (S) means L(r) U L(s)
R.S means L(r).L(s)
R* denotes L(r*)

3.6. REGULAR DEFINITIONS

For notational convenience, we may wish to give names to regular expressions and to define regular expressions using these names as if they were symbols.
Identifiers are the set or string of letters and digits beginning with a letter. The following regular definition provides a precise specification for this class of string.

Example-1,

Ab*|cd? Is equivalent to (a(b*)) | (c(d?))

Pascal identifier

Letter - A | B | ……| Z | a | b |……| z|
Digits - 0 | 1 | 2 | …. | 9
Id - letter (letter / digit)*

Recognition of tokens:

We learn how to express pattern using regular expressions. Now, we must study how to take the patterns for all the needed tokens and build a piece of code that examins the input string and finds a prefix that is a lexeme matching one of the patterns.

```
Stmt →if expr then stmt
         | If expr then else stmt
         | є
Expr →term relop term
          | term
Term →id
           |number
```

For relop ,we use the comparison operations of languages like Pascal or SQL where = is "equals" and < > is "not equals" because it presents an interesting structure of lexemes.
The terminal of grammar, which are if, then , else, relop ,id and numbers are the names of tokens as far as the lexical analyzer is concerned, the patterns for the tokens are described using regular definitions.

digit → [0,9]
digits →digit+
number →digit(.digit)?(e.[+-]?digits)?
letter → [A-Z,a-z]
id →letter(letter/digit)*
if → if
then →then

else →else
relop →< | > |<= | >= | = = | < >

In addition, we assign the lexical analyzer the job stripping out white space, by recognizing the "token" we defined by:

WS → (blank/tab/newline)+

Here, blank, tab and newline are abstract symbols that we use to express the ASCII characters of the same names. Token ws is different from the other tokens in that ,when we recognize it, we do not return it to parser ,but rather restart the lexical analysis from the character that follows the white space . It is the following token that gets returned to the parser.

Lexeme	**Token Name**	**Attribute Value**
Any WS	-	-
if	if	-
then	then	-
else	else	-
Any id	Id	Pointer to table entry
Any number	number	Pointer to table entry
<	relop	LT
<=	relop	LE
==	relop	EQ
<>	relop	NE

3.7. TRANSITION DIAGRAM:

Transition Diagram has a collection of nodes or circles, called states. Each state represents a condition that could occur during the process of scanning the input looking for a lexeme that matches one of several patterns .

Edges are directed from one state of the transition diagram to another. each edge is labeled by a symbol or set of symbols.

If we are in one state s, and the next input symbol is a, we look for an edge out of state s labeled by a. if we find such an edge ,we advance the forward pointer and enter the state of the transition diagram to which that edge leads.

Some important conventions about transition diagrams are

1. Certain states are said to be accepting or final .These states indicates that a lexeme has been found, although the actual lexeme may not consist of all positions b/w the lexeme Begin and forward pointers we always indicate an accepting state by a double circle.
2. In addition, if it is necessary to return the forward pointer one position, then we shall additionally place a * near that accepting state.
3. One state is designed the state ,or initial state ., it is indicated by an edge labeled "start" entering from nowhere .the transition diagram always begins in the state before any input symbols have been used.

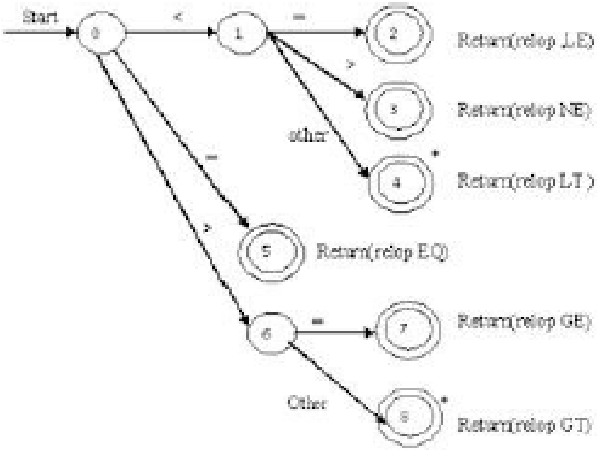


Fig. 3.3: Transition diagram of Relational operators

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

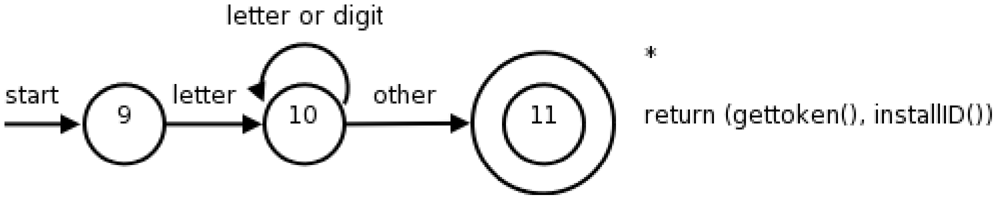


Fig. 3.4: Transition diagram of Identifier

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

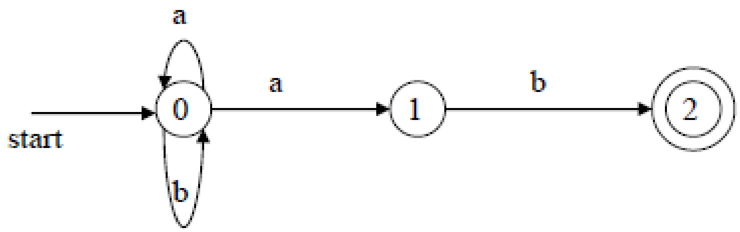
3.8. FINITE AUTOMATON

- A *recognizer* for a language is a program that takes a string x, and answers “yes” if x is a sentence of that language, and “no” otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic (DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.

3.9. Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
    - ○ S - a set of states
    - ○ Σ - a set of input symbols (alphabet)
    - ○ move - a transition function move to map state-symbol pairs to sets of states.
    - ○ s0 - a start (initial) state
    - ○ F- a set of accepting states (final states)
- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

Example:



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

Transition Function:

	a	b
0	{0,1}	{0}
1	{}	{2}
2	{}	{}

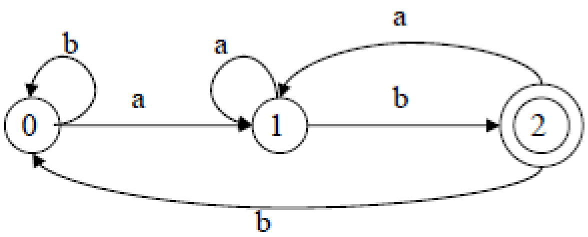
The language recognized by this NFA is (a|b)*ab

3.10. Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
- No state has ε- transition
- For each symbol a and state s, there is at most one labeled edge a leaving s. i.e. transition function is from pair of state-symbol to state (not set of states)

Example:

The DFA to recognize the language (a|b)* ab is as follows.



Transition Graph

0 is the start state s0
{2} is the set of final states F
Σ = {a,b}
S = {0,1,2}

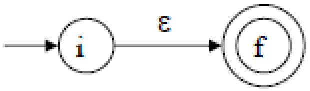
Transition Function:

	a	b
0	1	0
1	1	2
2	1	0

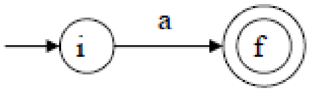
Note that the entries in this function are single value and not set of values (unlike NFA).

3.11. Converting RE to NFA

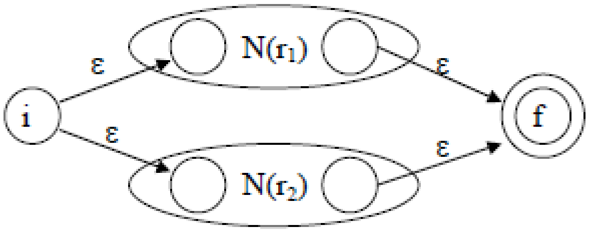
- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
- It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA.
- To recognize an empty string ε:



- To recognize a symbol a in the alphabet Σ :

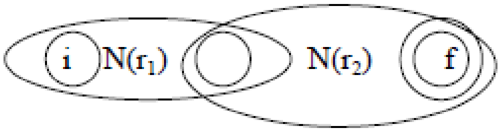


- For regular expression r1 | r2:



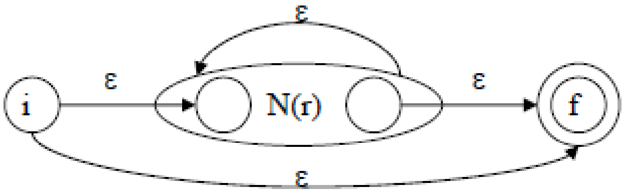
N(r1) and N(r2) are NFAs for regular expressions r1 and r2.

- For regular expression r1 r2

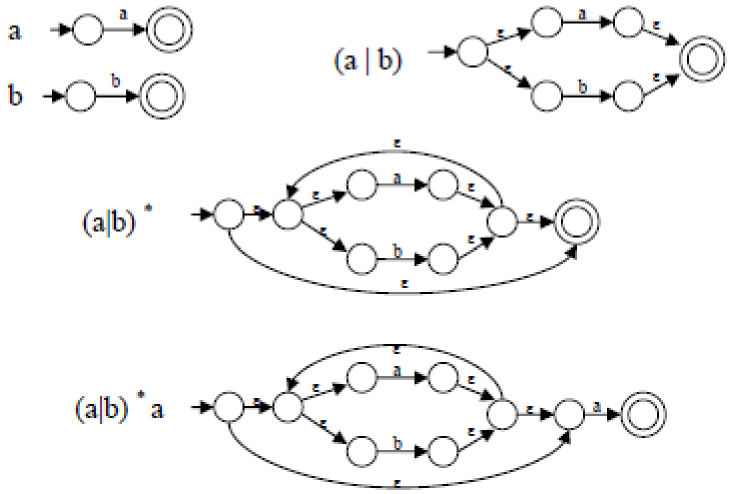


Here, final state of N(r1) becomes the final state of N(r1r2).

- For regular expression r*



Example:
For a RE (a|b) * a, the NFA construction is shown below.



3.12. Converting NFA to DFA (Subset Construction)

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an –transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an -transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- The **-closure** function takes a state and returns the set of states reachable from it based on (one or more) -transitions. Note that this will always include the state itself. We should be able to get from a state to any state in its -closure without consuming any input.
- The function **move** takes a state and a character, and returns the set of states reachable by one transition on this character.

We can generalise both these functions to apply to sets of states by taking the union of the application to individual states.

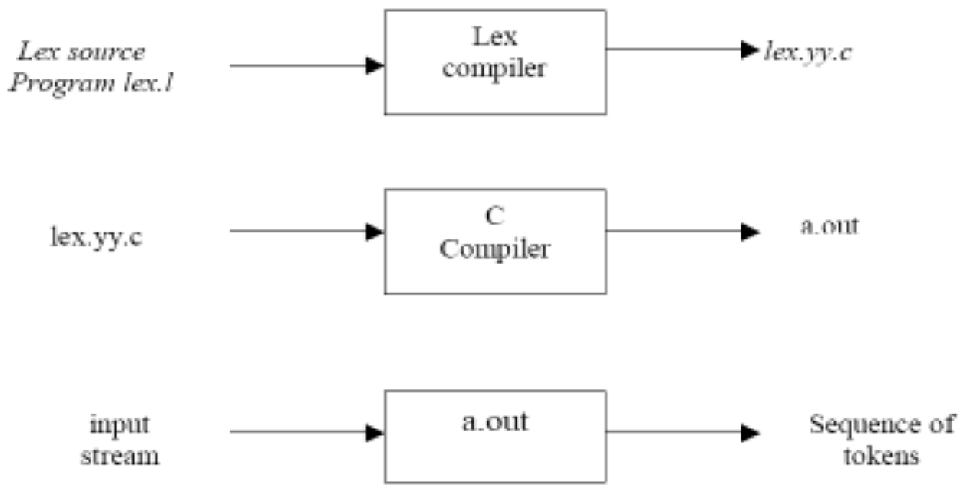
For Example, if A, B and C are states, move({A,B,C},`a') = move(A,`a') move(B,`a') move(C,`a').

The Subset Construction Algorithm is a follows:

```
put ε-closure({s0}) as an unmarked state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
    begin
        mark S1
        for each input symbol a do
            begin
                S2 ← ε-closure(move(S1,a))
                if (S2 is not in DS) then
                add S2 into DS as an unmarked state
                transfunc[S1,a] ← S2
            end
    end
```

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure({s0})

3.13. Lexical Analyzer Generator



3.18. 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.

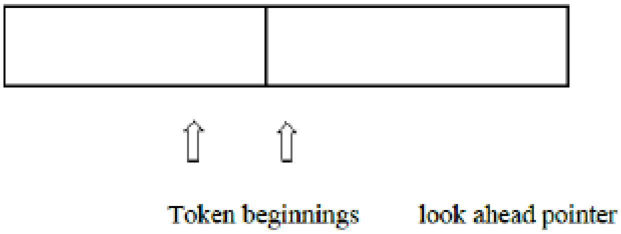
Note: You can refer to a sample lex program given in page no. 109 of chapter 3 of the book: *Compilers: Principles, Techniques, and Tools* by Aho, Sethi & Ullman for more clarity.

3.19. INPUT BUFFERING

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

1. Buffer pairs
2. Sentinels

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



Token beginnings look ahead pointerThe distance which the lookahead pointer may have to travel past the actual token may belarge. For example, in a PL/I program we may see:

DECALRE (ARG1, ARG2… ARG *n*) Without knowing whether DECLARE is a keyword or an array name until we see the character that follows the right parenthesis. In either case, the token itself ends at the second E. If the look ahead pointer travels beyond the buffer half in which it began, the other half must be loaded with the next characters from the source file. Since the buffer shown in above figure is of limited size there is an implied constraint on how much look ahead can be used before the next token is discovered. In the above example, ifthe look ahead traveled to the left half and all the way through the left half to the middle, we could not reload the right half, because we would lose characters that had not yet been groupedinto tokens. While we can make the buffer larger if we chose or use another buffering scheme,we cannot ignore the fact that overhead is limited.

SYNTAX ANALYSIS

4.1 ROLE OF THE PARSER :

Parser for any grammar is program that takes as input string w (obtain set of strings tokens from the lexical analyzer) and produces as output either a parse tree for w , if w is a valid sentences of grammar or error message indicating that w is not a valid sentences of given grammar. The goal of the parser is to determine the syntactic validity of a source string is valid, a tree is built for use by the subsequent phases of the computer. The tree reflects the sequence of derivations or reduction used during the parser. Hence, it is called parse tree. If string is invalid, the parse has to issue diagnostic message identifying the nature and cause of the errors in string. Every elementary subtree in the parse tree corresponds to a production of the grammar.

There are two ways of identifying an elementry sutree:

1. By deriving a string from a non-terminal or
2. By reducing a string of symbol to a non-terminal.

The two types of parsers employed are:

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

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

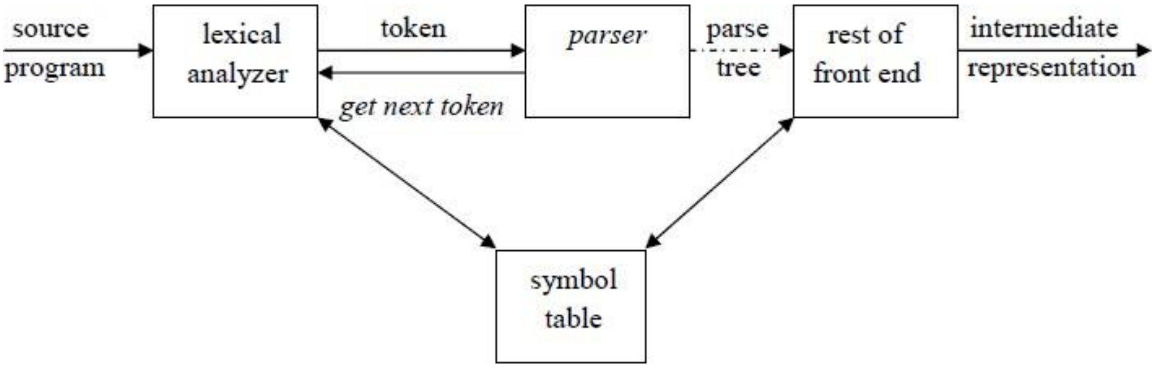


Fig . 4.1: position of parser in compiler model.

4.2 CONTEXT FREE GRAMMARS

Inherently recursive structures of a programming language are defined by a context-free Grammar. In a context-free grammar, we have four triples G(V,T,P,S).

Here , V is finite set of terminals (in our case, this will be the set of tokens)

T is a finite set of non-terminals (syntactic-variables)

P is a finite set of productions rules in the following form

$A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string)

S is a start symbol (one of the non-terminal symbol)

L(G) is the language of G (the language generated by G) which is a set of sentences.

A sentence of L(G) is a string of terminal symbols of G. If S is the start symbol of G then ω is a sentence of L(G) iff $S \Rightarrow \omega$ whereω is a string of terminals of G. If G is a context-free grammar, L(G) is a context-free language. Two grammar G_1 and G_2 are equivalent, if they produce same grammar.

Consider the production of the form $S \Rightarrow \alpha$, If α contains non-terminals, it is called as a sentential form of G. If α does not contain non-terminals, it is called as a sentence of G.

4.2.1 Derivations

In general a derivation step is

$\alpha A \beta \Rightarrow \alpha\gamma\beta$ is sentential form and if there is a production rule $A \rightarrow \gamma$ in our grammar. where α and β are arbitrary strings of terminal and non-terminal symbols $\alpha 1 \Rightarrow \alpha 2 \Rightarrow ... \Rightarrow \alpha n$ (αn derives from $\alpha 1$ or $\alpha 1$ derives αn). There are two types of derivaion

1At each derivation step, we can choose any of the non-terminal in the sentential form of G for the replacement.

2If we always choose the left-most non-terminal in each derivation step, this derivation is called as left-most derivation.

Example:
$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid - E$
$E \rightarrow (E)$
$E \rightarrow id$

Leftmost derivation :

$E \rightarrow E + E$

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

The string is derive from the grammar w= id*id+id, which is consists of all terminal symbols

Rightmost derivation

$E \rightarrow E + E$

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

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

Sentence to be derived : – (id+id)

LEFTMOST DERIVATION	RIGHTMOST DERIVATION
E → - E	E → - E
E → - (E)	E → - (E)
E → - (E+E)	E → - (E+E)
E → - (id+E)	E → - (E+id)
E → - (id+id)	E → - (id+id)

- String that appear in leftmost derivation are called **left sentinel forms.**
- String that appear in rightmost derivation are called **right sentinel forms.**

Sentinels:

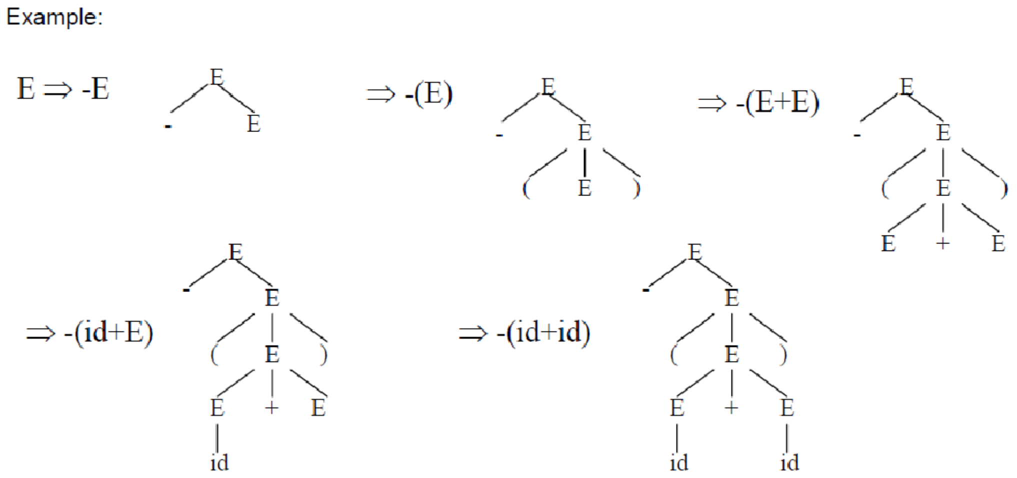
- Given a grammar G with start symbol S, if S → α , where α may contain non-terminals or terminals, then α is called the sentinel form of G.

Yield or frontier of tree:

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

4.2.2 PARSE TREE

- Inner nodes of a parse tree are non-terminal symbols.
- The leaves of a parse tree are terminal symbols.
- A parse tree can be seen as a graphical representation of a derivation.



Ambiguity:

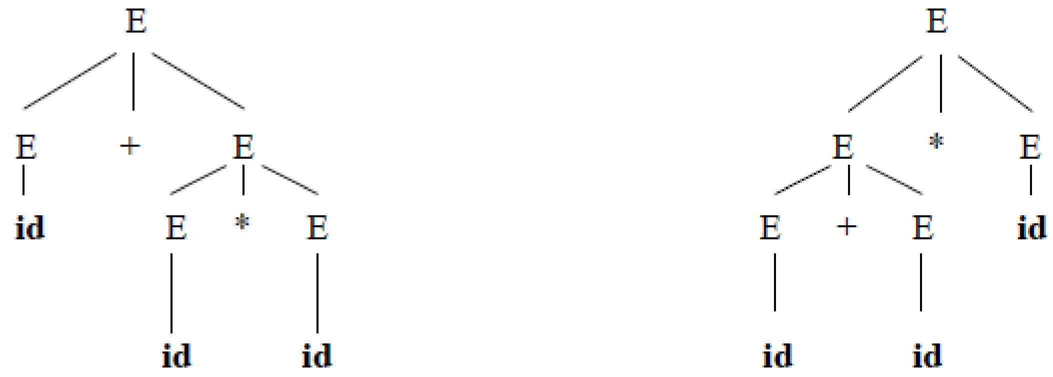
A grammar that produces more than one parse for some sentence is said to be **ambiguous grammar**.

Example : Given grammar G : E → E+E | E*E | (E) | - E | id

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

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

The two corresponding parse trees are :



Example:

To disambiguate the grammar E → E+E | E*E | E^E | id | (E), we can use precedence of operators as follows:

^ (right to left)

/,* (left to right)

-,+ (left to right)

We get the following unambiguous grammar:

E → E+T | T

T → T*F | F

F → G^F | G

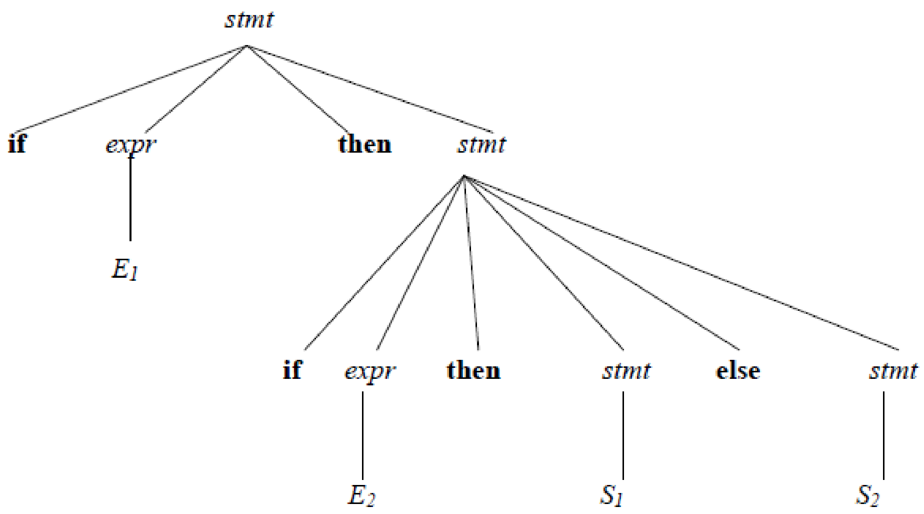
G → id | (E)

Consider this example, G: *stmt* → **if** *expr* **then** *stmt* |**if** *expr* **then** *stmt* **else***stmt* | **other**

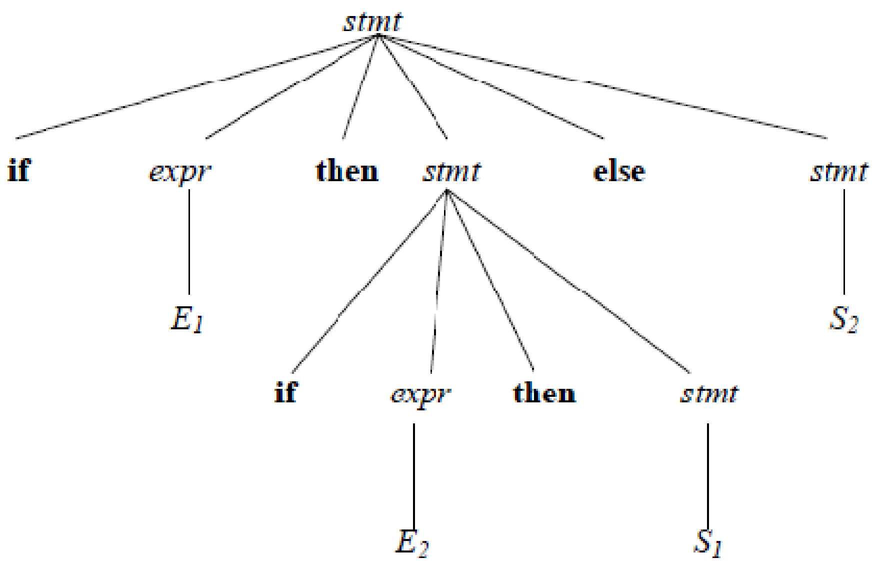
This grammar is ambiguous since the string **if E1 then if E2 then S1 else S2** has the following

Two parse trees for leftmost derivation :

1.



2.



To eliminate ambiguity, the following grammar may be used:

stmt → *matched_stmt* | *unmatched_stmt*

matched_stmt → **if** *expr* **then** *matched_stmt* **else** *matched_stmt* | **other**

unmatched_stmt → **if** *expr* **then** *stmt*| **if** *expr* **then** *matched_stmt* **else** *unmatched_stmt*

Eliminating Left Recursion:

A grammar is said to be *left recursive* if it has a non-terminal *A* such that there is a derivation A=>Aα for some string α. Top-down parsing methods cannot handle left-recursive grammars. Hence, left recursion can be eliminated as follows:

If there is a production A → Aα | β it can be replaced with a sequence of two productions

A → βA’
A’ → αA’ | ε

Without changing the set of strings derivable from A.

Example : Consider the following grammar for arithmetic expressions:

E → E+T | T

T → T*F | F

F → (E) | id

First eliminate the left recursion for E as

E → TE’

E’ → +TE’ |ε

Then eliminate for T as

T → FT’

T’→ *FT’ | ε

Thus the obtained grammar after eliminating left recursion is

E → TE’

E’ → +TE’ |ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

Algorithm to eliminate left recursion:

```
1. Arrange the non-terminals in some order A1, A2 . . . An.
2.for i := 1 to n do begin
        for j := 1 to i-1 do begin
                replace each production of the form Ai → Aj γ
                by the productions Ai → δ1 γ | δ2γ | . . . | δk γ
                where Aj→ δ1 |δ2 | . . . |δk are all the current Aj-productions;
        end
                eliminate the immediate left recursion among the Ai-productions
    end
```

Left factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we can rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

If there is any production A → αβ1 | αβ2 , it can be rewritten as

A → αA'

A' → β1 | β2

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

E → b

Left factored, this grammar becomes

S → iEtSS' | a

S' → eS | ε

E → b

TOP-DOWN PARSING

It can be viewed as an attempt to find a left-most derivation for an input string or an attempt to construct a parse tree for the input starting from the root to the leaves.

Types of top-down parsing :

1. Recursive descent parsing
2. Predictive parsing

1. RECURSIVE DESCENT PARSING

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

Example for backtracking :

Consider the grammar G : S→cAd

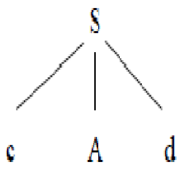
A → ab | a

and the input string w=cad.

The parse tree can be constructed using the following top-down approach :

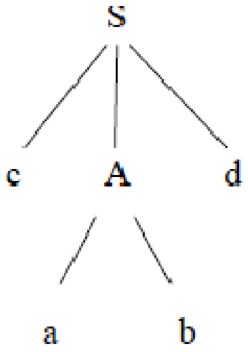
Step1:

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



Step2:

The leftmost leaf ‘c’ matches the first symbol of w, so advance the input pointer to the second symbol of w ‘a’ and consider the next leaf ‘A’. Expand A using the first alternative.



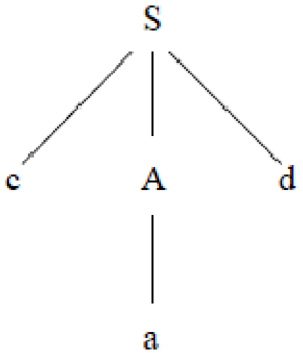
Step3:

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

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

Step4:

Now try the second alternative for A.



Now we can halt and announce the successful completion of parsing.

Example for recursive decent parsing:

A left-recursive grammar can cause a recursive-descent parser to go into an infinite loop.

Hence, **elimination of left-recursion** must be done before parsing.

Consider the grammar for arithmetic expressions

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating the left-recursion the grammar becomes,

E → TE’

E’ → +TE’ |ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

Now we can write the procedure for grammar as follows:

Recursive procedure:

```
Procedure E()
begin
    T( );
    EPRIME( );
End
Procedure EPRIME( )
    begin
        If input_symbol=’+’ then
        ADVANCE( );
        T( );
        EPRIME( );
    end
Procedure T( )
    begin
        F( );
        TPRIME( );
    End
```

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

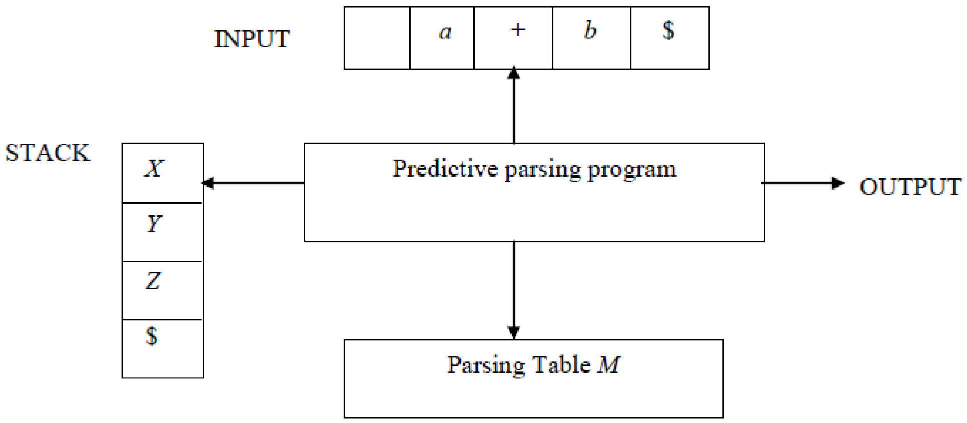
Stack implementation:

PROCEDURE	INPUT STRING
E( )	**id**+id*id
T( )	**id**+id*id
F( )	**id**+id*id
ADVANCE( )	id_id*id
TPRIME( )	id_id*id
EPRIME( )	id_id*id
ADVANCE( )	id+**id***id
T( )	id+**id***id
F( )	id+**id***id
ADVANCE( )	id+id*id
TPRIME( )	id+id*id
ADVANCE( )	id+id*id
F( )	id+id*id
ADVANCE( )	id+id* **id**
TPRIME( )	id+id* **id**

2. PREDICTIVE PARSING

- ✓ Predictive parsing is a special case of recursive descent parsing where no backtracking is required.
- ✓ The key problem of predictive parsing is to determine the production to be applied for a non-terminal in case of alternatives.

Non-recursive predictive parser



The table-driven predictive parser has an input buffer, stack, a parsing table and an output stream.

Input buffer:

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

Stack:

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

Parsing table:

It is a two-dimensional array $M[A, a]$, where ***'A'*** is a non-terminal and ***'a'*** is a terminal.

Predictive parsing program:

The parser is controlled by a program that considers X, the symbol on top of stack, and a, the current input symbol. These two symbols determine the parser action. There are three possibilities:

1. If $X = a = \$$, the parser halts and announces successful completion of parsing.
2. If $X = a \neq \$$, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a non-terminal , the program consults entry $M[X, a]$ of the parsing table M. This entry will either be an X-production of the grammar or an error entry.

If *M*[*X*, *a*] = {*X* → *UVW*},the parser replaces *X* on top of the stack by *UVW*

If *M*[*X*, *a*] =**error**, the parser calls an error recovery routine.

Algorithm for nonrecursive predictive parsing:

Input : A string *w* and a parsing table *M* for grammar *G*.

Output : If *w* is in *L*(*G*), a leftmost derivation of *w*; otherwise, an error indication.

Method : Initially, the parser has $*S* on the stack with *S*, the start symbol of *G* on top, and *w*$ in the input buffer. The program that utilizes the predictive parsing table *M* to produce a parse for the input is as follows:

```
set ip to point to the first symbol of w$;
repeat
    letX be the top stack symbol andathe symbol pointed to by ip;
    if X is a terminal or $then
        if X = a then
            popX from the stack and advance ip
        else error()
    else/* X is a non-terminal */
      if M[X, a] = X →Y1Y2 … Yk then begin
      pop X from the stack;
      push Yk, Yk-1, … ,Y1 onto the stack, with Y1 on top;
      output the production X → Y1 Y2 . . . Yk
    end
    elseerror()
until X = $
```

Predictive parsing table construction:

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

1. FIRST
2. FOLLOW

Rules for first( ):

1. If *X* is terminal, then FIRST(*X*) is {X}.
2. If *X* → ε is a production, then add ε to FIRST(*X*).
3. If *X* is non-terminal and X → *a*α is a production then add *a* to FIRST(X).

4. If X is non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then place *a* in FIRST(*X*) if for some *i*, *a* is in FIRST(*Yi*), and ε is in all of FIRST(*Y1*),…,FIRST(*Yi-1*); that is, *Y1*,….*Yi-1* => ε. If ε is in FIRST(Y_j) for all j=1,2,..,k, then add ε to FIRST(*X*).

Rules for follow( ):

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

2. If there is a production *A* → α*B*β, then everything in FIRST(β) except ε is placed in follow(*B*).
3. If there is a production *A* → α*B*, or a production *A* → α*B*β where FIRST(β) contains ε, then everything in FOLLOW(*A*) is in FOLLOW(*B*).

Algorithm for construction of predictive parsing table:

Input : Grammar*G*

Output : Parsing table *M*

Method :

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

Example:

Consider the following grammar :

E → E+T | T

T→T*F | F

F → (E) | id

After eliminating left-recursion the grammar is

E → TE’

E’ → +TE’ |ε

T → FT’

T’ → *FT’ | ε

F → (E) | id

First( ) :

FIRST(E) = { (, id}

FIRST(E’) ={+ ,ε}

FIRST(T) = { (, id}

FIRST(T’) = {*, ε }

FIRST(F) = { (, id }

Follow( ):

FOLLOW(E) = { $,) }

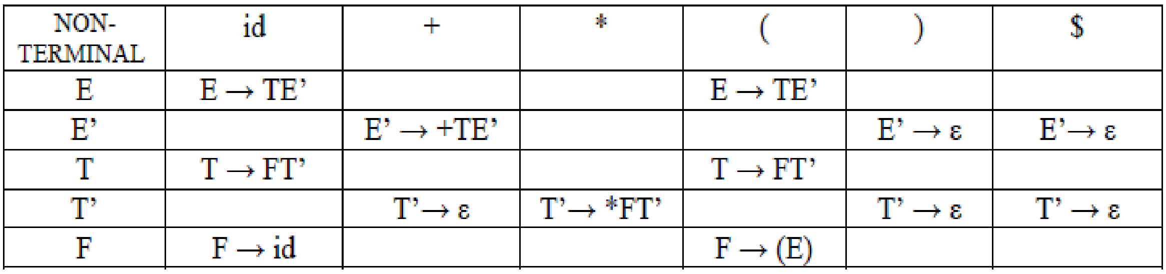
FOLLOW(E’) = { $,) }

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

FOLLOW(T’) = { +, $,) }

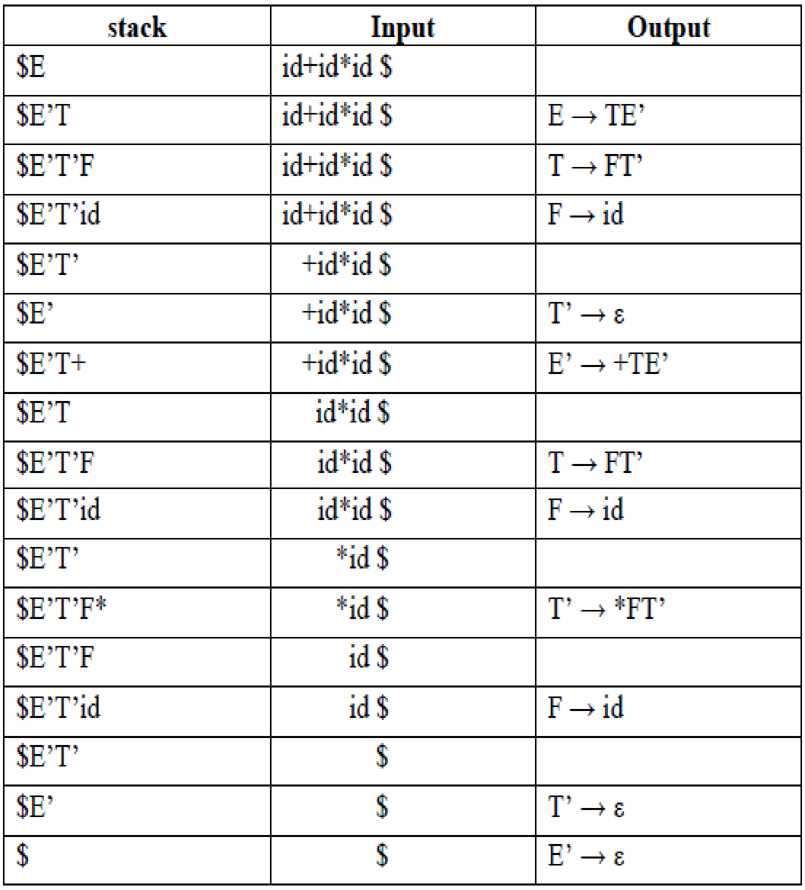
FOLLOW(F) = {+, * , $,) }

Predictive parsing table :



NON-TERMINAL	id	+	*	(	)	$
E	E → TE’			E → TE’		
E’		E’ → +TE’			E’ → ε	E’→ ε
T	T → FT’			T → FT’		
T’		T’→ ε	T’→ *FT’		T’ → ε	T’ → ε
F	F → id			F → (E)		

Stack implementation:



stack	Input	Output
$E	id+id*id $	
$E’T	id+id*id $	E → TE’
$E’T’F	id+id*id $	T → FT’
$E’T’id	id+id*id $	F → id
$E’T’	+id*id $	
$E’	+id*id $	T’ → ε
$E’T+	+id*id $	E’ → +TE’
$E’T	id*id $	
$E’T’F	id*id $	T → FT’
$E’T’id	id*id $	F → id
$E’T’	*id $	
$E’T’F*	*id $	T’ → *FT’
$E’T’F	id $	
$E’T’id	id $	F → id
$E’T’	$	
$E’	$	T’ → ε
$	$	E’ → ε

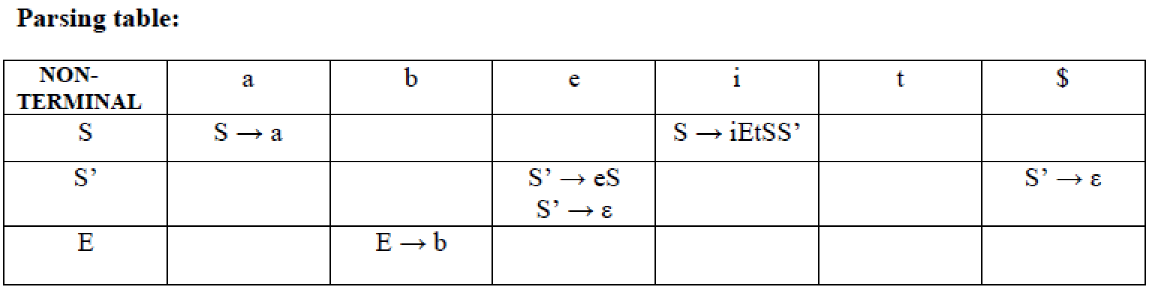
LL(1) grammar:

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

Consider this following grammar:

S → iEtS | iEtSeS | a
E → b

After eliminating left factoring, we have
S→iEtSS’ | a
S’→eS |ε
E→b
To construct a parsing table, we need FIRST() and FOLLOW() for all the non-terminals.
FIRST(S) = { i, a }
FIRST(S’) = {e,ε}
FIRST(E) = { b}
FOLLOW(S) = { $,e }
FOLLOW(S’) = { $,e }
FOLLOW(E) = {t}



Parsing table:

NON-TERMINAL	a	b	e	i	t	$
S	S → a			S → iEtSS’		
S’			S’ → eS S’ → ε			S’ → ε
E		E → b				

Since there are more than one production, the grammar is not LL(1) grammar.
Actions performed in predictive parsing:
1. Shift
2. Reduce
3. Accept
4. Error
Implementation of predictive parser:
1. Elimination of left recursion, left factoring and ambiguous grammar.
2. Construct FIRST() and FOLLOW() for all non-terminals.
3. Construct predictive parsing table.
4. Parse the given input string using stack and parsing table.

BOTTOM-UP PARSING

Constructing a parse tree for an input string beginning at the leaves and going towards the root is called bottom-up parsing.
A general type of bottom-up parser is a **shift-reduce parser**.

SHIFT-REDUCE PARSING

Shift-reduce parsing is a type of bottom-up parsing that attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).
Example:
Consider the grammar:
S → aABe
A → Abc | b
B → d
The sentence to be recognized is **abbcde.**

REDUCTION (LEFTMOST)	RIGHTMOST DERIVATION
a**b**bcde (A → b)	**S**→ aA**B**e
a**Abc**de (A → Abc)	→ a**A**de
aA**d**e (B → d)	→ a**A**bcde
aABe(S → aABe)	→ abbcde
S	

The reductions trace out the right-most derivation in reverse.

Handles:

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

Example:

Consider the grammar:

E → E+E
E → E*E
E → (E)
E → id

And the input string $id_1+id_2*id_3$

The rightmost derivation is :

$$
\begin{aligned}
E &\rightarrow \underline{\mathbf{E}+\mathbf{E}}\\
&\rightarrow E+\underline{\mathbf{E*E}}\\
&\rightarrow E+E\mathbf{*}\underline{\mathbf{id_3}}\\
&\rightarrow E+\underline{\mathbf{id_2}}\mathbf{*}id_3\\
&\rightarrow \underline{\mathbf{id}\ _\mathbf{1}}+id_2\mathbf{*}id_3
\end{aligned}
$$

In the above derivation the underlined substrings are called**handles.**

Handle pruning:

A rightmost derivation in reverse can be obtained by "**handle pruning**".
(i.e.) if*w*is a sentence or string of the grammar at hand, then*w*= y_n, where y_n is then^{th} right-sentinel form of some rightmost derivation.

Stack implementation of shift-reduce parsing :

Stack	Input	Action
$	$id_1+id_2*id_3$ $	shift
$ id_1	$+id_2*id_3$ $	reduce by E→id
$ E	$+id_2*id_3$ $	shift
$ E+	id_2*id_3 $	shift
$ $E+id_2$	$*id_3$ $	reduce by E→id
$ E+E	$*id_3$ $	shift
$ E+E*	id3 $	shift
$ E+E*id3	$	reduce by E→id
$ E+E*E	$	reduce by E→ E *E
$ E+E	$	reduce by E→ E+E
$ E	$	accept

Actions in shift-reduce parser:

- shift – The next input symbol is shifted onto the top of the stack.
- reduce – The parser replaces the handle within a stack with a non-terminal.
- accept – The parser announces successful completion of parsing.
- error – The parser discovers that a syntax error has occurred and calls an error recovery routine.

Conflicts in shift-reduce parsing:

There are two conflicts that occur in shift shift-reduce parsing:

1. Shift-reduce conflict: The parser cannot decide whether to shift or to reduce.

2. Reduce-reduce conflict: The parser cannot decide which of several reductions to make.

1. Shift-reduce conflict:

Example:

Consider the grammar:

E→E+E | E*E | id and input id+id*id

Stack	Input	Action	Stack	Input	Action
$ E+E	*id $	Reduce by E→E+E	$E+E	*id $	Shift
$ E	*id $	Shift	$E+E*	id $	Shift
$ E*	id $	Shift	$E+E*id	$	Reduce by E→id
$ E*id	$	Reduce by E→id	$E+E*E	$	Reduce by E→E*E
$ E*E	$	Reduce by E→E*E	$E+E	$	Reduce by E→E*E
$ E			$E		

2. **Reduce-reduce conflict:**

Consider the grammar:

M → R+R | R+c | R
R → c
and input c+c

Stack	Input	Action	Stack	Input	Action
$	c+c $	Shift	$	c+c $	Shift
$ c	+c $	Reduce by R→c	$ c	+c $	Reduce by R→c
$ R	+c $	Shift	$ R	+c $	Shift
$ R+	c $	Shift	$ R+	c $	Shift
$ R+c	$	Reduce by R→c	$ R+c	$	Reduce by M→R+c
$ R+R	$	Reduce by M→R+R	$ M	$	
$ M	$				

Viable prefixes:

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

OPERATOR-PRECEDENCE PARSING

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

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

Example:

Consider the grammar:

E → EAE | (E) | -E | id
A → + | - | * | / | ↑

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

E → E+E | E-E | E*E | E/E | E↑E | -E | id

Operator precedence relations:

There are three disjoint precedence relations namely

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

The relations give the following meaning:

a<·b – a yields precedence to b
a = b – a has the same precedence as b
a·>b – a takes precedence over b

Rules for binary operations:

1. If operator θ_1 has higher precedence than operator $\theta_{2,}$ then make
$\theta_1 \cdot\!> \theta_2$ and $\theta_2 <\!\cdot\ \theta_1$

2. If operators θ_1 and $\theta_{2,}$ are of equal precedence, then make
$\theta_1 \cdot\!> \theta_2$ and $\theta_2 \cdot\!> \theta_1$ if operators are left associative
$\theta_1 <\!\cdot\ \theta_2$ and $\theta_2 <\!\cdot\ \theta_1$ if right associative

3. Make the following for all operators θ:
θ<· id , id·>θ
θ<·(, (<·θ
)·>θ , θ·>)
θ·>$, $<·θ

Also make
(=) , (< · (,) ·>) , (< · id , id ·>) , $< · id , id ·>$, $< · (,) ·>$

Example:

Operator-precedence relations for the grammar

E → E+E | E-E | E*E | E/E | E↑E | (E) | -E | id is given in the following table assuming

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

Note that the**blanks**in the table denote error entries.

TABLE : Operator-precedence relations

	+	-	*	/	↑	id	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
-	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	.·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

<u>Operator precedence parsing algorithm:</u>

Input : An input string*w*and a table of precedence relations.
Output : If*w*is well formed, a*skeletal*parse tree**,**with a placeholder non-terminal E labeling all interior nodes; otherwise, an error indication.
Method : Initially the stack contains $ and the input buffer the string*w*$. To parse, we execute the following program :

```
(1) Set ip to point to the first symbol of w$;
(2) repeat forever
(3) if $ is on top of the stack and ip points to $ then
(4) return
      else begin
(5)      let a be the topmost terminal symbol on the stack
           and let b be the symbol pointed to by ip;
(6) if a <· b or a = b then begin
(7)         push b onto the stack;
(8)         advance ip to the next input symbol;
         end;
```

```
else if a ·> b then                    /*reduce*/
repeat
        pop the stack
until the top stack terminal is related by <·
        to the terminal most recently popped
else error( )
    end
```

Stack implementation of operator precedence parsing:

Operator precedence parsing uses a stack and precedence relation table for its implementation of above algorithm. It is a shift-reduce parsing containing all four actions shift, reduce, accept and error.

The initial configuration of an operator precedence parsing is

STACK	INPUT
$	w $

where w is the input string to be parsed.

Example:

Consider the grammar E → E+E | E-E | E*E | E/E | E↑E | (E) | id. Input string is**id+id*id**.The implementation is as follows:

STACK	INPUT	COMMENT
$	<· id+id*id $	shift id
$ id	·> +id*id $	pop the top of the stack id
$	<· +id*id $	shift +
$ +	<· id*id $	shift id
$ +id	·> *id $	pop id
$ +	<· *id $	shift *
$ + *	<· id $	shift id
$ + * id	·> $	pop id
$ + *	·> $	pop *
$ +	·> $	pop +
$	$	accept

Advantages of operator precedence parsing:

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

Disadvantages of operator precedence parsing:

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

LR PARSERS

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

Advantages of LR parsing:

- ✓ It recognizes virtually all programming language constructs for which CFG can be written.
- ✓ It is an efficient non-backtracking shift-reduce parsing method.
- ✓ A grammar that can be parsed using LR method is a proper superset of a grammar that can be parsed with predictive parser.
- ✓ It detects a syntactic error as soon as possible.

Drawbacks of LR method:

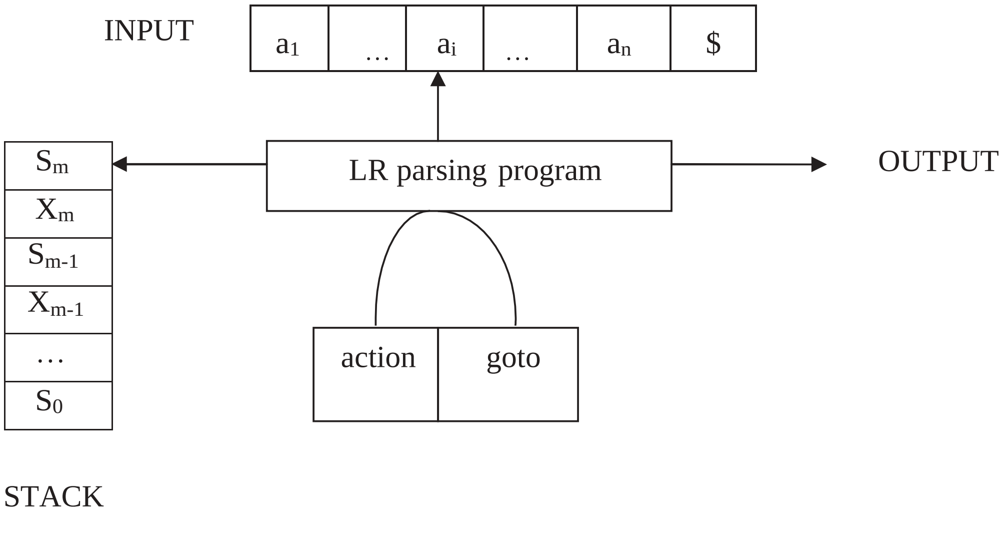
It is too much of work to construct a LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator, is needed. Example: YACC.

Types of LR parsing method:

1. SLR- Simple LR
   - Easiest to implement, least powerful.
2. CLR- Canonical LR
   - Most powerful, most expensive.
3. LALR- Look-Ahead LR
   - Intermediate in size and cost between the other two methods.

The LR parsing algorithm:

The schematic form of an LR parser is as follows:



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

➢The driver program is the same for all LR parser.

➢The parsing program reads characters from an input buffer one at a time.

➢The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2\ldots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a state.

➢The parsing table consists of two parts :*action*and*goto*functions.

Action: The parsing program determines s_m, the state currently on top of stack, and a_i, the current input symbol. It then consults*action*[s_m,a_i] in the action table which can have one of four values :

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

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

LR Parsing algorithm:

Input: An input string*w*and an LR parsing table with functions*action*and*goto*for grammar G.

Output: If*w*is in L(G), a bottom-up-parse for*w*; otherwise, an error indication.

Method: Initially, the parser has s_0 on its stack, where s_0 is the initial state, and*w*$ in the input buffer. The parser then executes the following program :

```
set ip to point to the first input symbol of w$;
repeat forever begin
          let s be the state on top of the stack and
               a the symbol pointed to by ip;
          if action[s,a] = shift s’ then begin
               push a then s’ on top of the stack;
               advance ip to the next input symbol
          end
          else if action[s,a] = reduce A→β then begin
                 pop 2* | β | symbols off the stack;
                 let s’ be the state now on top of the stack;
                 push A then goto[s’, A] on top of the stack;
                 output the production A→ β
          end
          else if action[s,a] = accept then
                 return
          else error( )
end
```

CONSTRUCTING SLR(1) PARSING TABLE:

To perform SLR parsing, take grammar as input and do the following:

1. Find LR(0) items.
2. Completing the closure.
3. Compute*goto*(I,X), where, I is set of items and X is grammar symbol.

LR(O) items:

An*LR(O) item*of a grammar G is a production of G with a dot at some position of the right side. For example, production A → XYZ yields the four items :

A →**.**XYZ
A → X**.**YZ
A → XY**.**Z
A → XYZ**.**

Closure operation:

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

1. Initially, every item in I is added to closure(I).
2. If A → a . Bβ is in closure(I) and B → y is a production, then add the item B → . y to I , if it is not already there. We apply this rule until no more new items can be added to closure(I).

Goto operation:

Goto(I, X) is defined to be the closure of the set of all items [A→ aX . β] such that [A→ a . Xβ] is in I.

Steps to construct SLR parsing table for grammar G are:

1. Augment G and produce G'
2. Construct the canonical collection of set of items C for G'
3. Construct the parsing action function*action*and*goto*using the following algorithm that requires FOLLOW(A) for each non-terminal of grammar.

<u>Algorithm for construction of SLR parsing table:</u>

Input: An augmented grammar G'
Output: The SLR parsing table functions*action*and*goto*for G'
Method:

1. Construct C = {$I_{0,}$ $I_{1,}$ …. I_n}, the collection of sets of LR(0) items for G'.
2. State*i*is constructed from I $_{i.}$. The parsing functions for state*i*are determined as follows:
   (a) If [A→a·*a*β] is in I_i and goto(I_i,*a*) = $I_{j,}$ then set*action*[*i*,*a*] to "shift j". Here*a*must be terminal.
   (b) If [A→a·] is in $I_{i\ ,}$ then set*action*[*i*,*a*] to "reduce A→a" for all*a*in FOLLOW(A).
   (c) If [S'→S.] is in I_i, then set*action*[*i*,$] to "accept".

If any conflicting actions are generated by the above rules, we say grammar is not SLR(1).

3. The*goto*transitions for state*i*are constructed for all non-terminals A using the rule: If*goto*(I_i,A) = I_j, then*goto*[i,A] =*j*.
4. All entries not defined by rules (2) and (3) are made “error”
5. The initial state of the parser is the one constructed from the set of items containing [S’→.S].

Example for SLR parsing:

Construct SLR parsing for the following grammar :

G : E → E + T | T
T → T * F | F
F → (E) | id

The given grammar is :

G : E → E + T ------ (1)
E →T ------ (2)
T → T * F ------ (3)
T → F ------ (4)
F → (E) ------ (5)
F → id ------ (6)

Step 1 : Convert given grammar into augmented grammar.

Augmented grammar :

E’ → E
E → E + T
E → T
T → T * F
T → F
F → (E)
F → id

Step 2 : Find LR (0) items.

I_0 : E’ →**.**E
E →**.**E + T
E →**.**T
T →**.**T * F
T →**.**F
F →**.**(E)
F →**.**id

GOTO (I_0 , E)

I_1 : E’ → E**.**
E → E**.**+ T

GOTO (I_4 , id)

I_5 : F → id**.**

GOTO (I_0 , T)
I_2 : E → T.
T → T.* F

GOTO (I_0 , F)
I_3 : T → F.

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

GOTO (I_0 , id)
I_5 : F → id.

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

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

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

GOTO (I_4 , T)
I_2 : E →T.
T → T.* F

GOTO (I_4 , F)
I_3 : T → F.

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

GOTO (I_6 , F)
I_3 : T → F.

GOTO (I_6 , ( )
I_4 : F → (.E)

GOTO (I_6 , id)
I_5 : F → id.

GOTO (I_7 , F)
I_{10} : T → T * F.

GOTO (I_7 , ( )
I_4 : F → (.E)
E →.E + T
E →.T
T →.T * F
T →.F
F →.(E)
F →.id

GOTO (I_7 , id)
I_5 : F → id.

GOTO (I_8 ,))
I_{11} : F → (E).

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

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

GOTO (I_4 , ( )

I_4 : F → (**.**E)
E →**.**E + T
E →**.**T
T →**.**T * F
T →**.**F
F →**.**(E)
F → id

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

SLR parsing table:

	ACTION						**GOTO**		
	id	**+**	*****	**(**	**)**	**$**	**E**	**T**	**F**
I_O	s5			s4			1	2	3
I_1		s6				ACC			
I_2		r2	s7		r2	r2			
I_3		r4	r4		r4	r4			
I_4	s5			s4			8	2	3
I_5		r6	r6		r6	r6			
I_6	s5			s4				9	3
I_7	s5			s4					10
I_8		s6			s11				
I_9		r1	s7		r1	r1			
I_{1O}		r3	r3		r3	r3			
I_{11}		r5	r5		r5	r5			

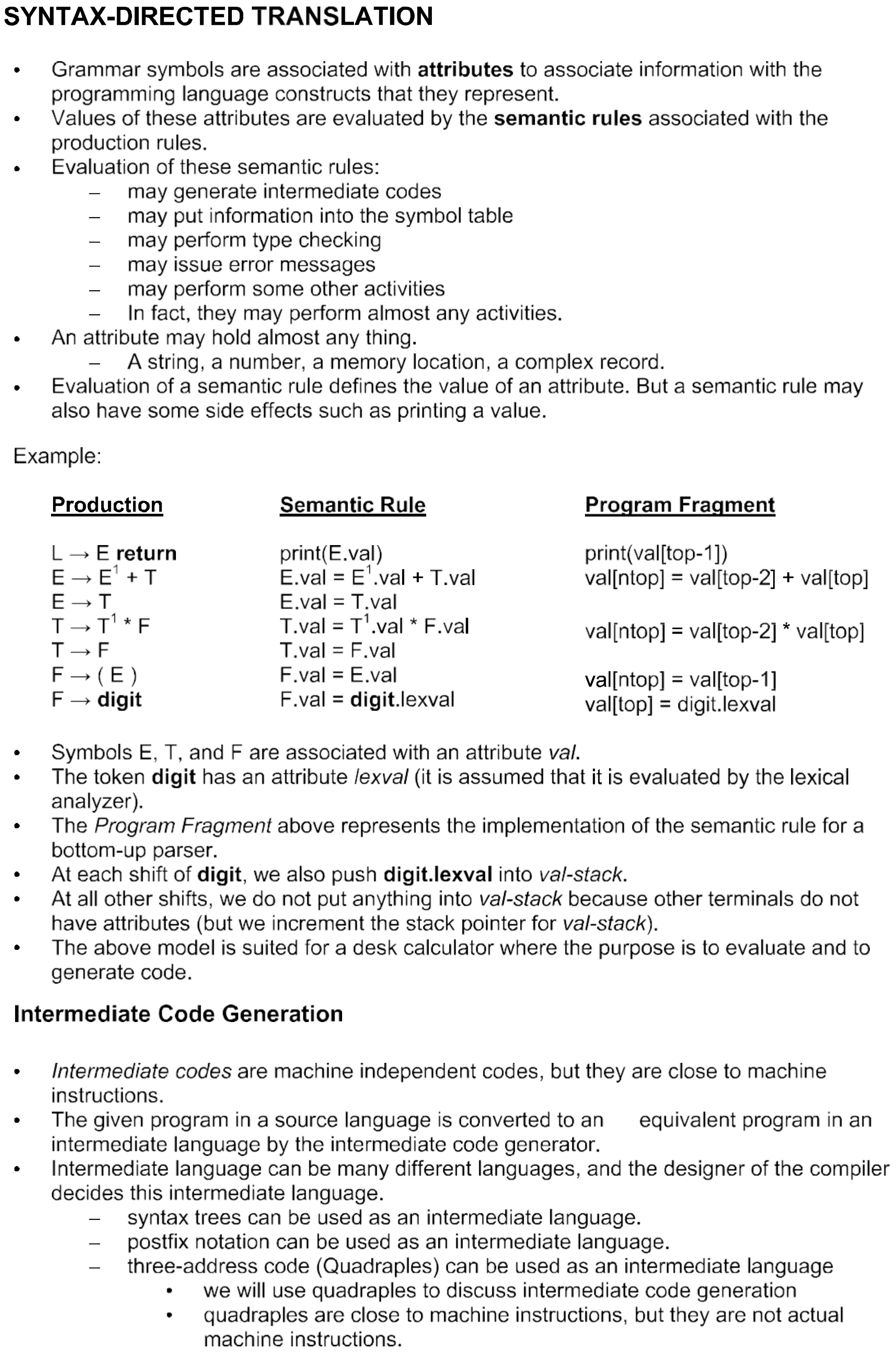
Blank entries are error entries.

Stack implementation:

Check whether the input**id + id * id**is valid or not.

STACK	INPUT	ACTION
0	id + id * id $	GOTO (I_0 , id) = s5 ;**shift**
0 id 5	+ id * id $	GOTO (I_5 , +) = r6 ;**reduce**by F→id
0 F 3	+ id * id $	GOTO (I_0 , F) = 3 GOTO (I_3 , +) = r4 ;**reduce**by T → F
0 T 2	+ id * id $	GOTO (I_0 , T) = 2 GOTO (I_2 , +) = r2 ;**reduce**by E → T
0 E 1	+ id * id $	GOTO (I_0 , E) = 1 GOTO (I_1 , +) = s6 ;**shift**
0 E 1 + 6	id * id $	GOTO (I_6 , id) = s5 ;**shift**
0 E 1 + 6 id 5	* id $	GOTO (I_5 , *) = r6 ;**reduce**by F → id
0 E 1 + 6 F 3	* id $	GOTO (I_6 , F) = 3 GOTO (I_3 , *) = r4 ;**reduce**by T → F
0 E 1 + 6 T 9	* id $	GOTO (I_6 , T) = 9 GOTO (I_9 , *) = s7 ;**shift**
0 E 1 + 6 T 9 * 7	id $	GOTO (I_7 , id) = s5 ;**shift**
0 E 1 + 6 T 9 * 7 id 5	$	GOTO (I_5 , $) = r6 ;**reduce**by F → id
0 E 1 + 6 T 9 * 7 F 10	$	GOTO (I_7 , F) = 10 GOTO (I_{10} , $) = r3 ;**reduce**by T → T * F
0 E 1 + 6 T 9	$	GOTO (I_6 , T) = 9 GOTO (I_9 , $) = r1 ;**reduce**by E → E + T
0 E 1	$	GOTO (I_0 , E) = 1 GOTO (I_1 , $) =**accept**

SYNTAX-DIRECTED TRANSLATION



SYNTAX-DIRECTED TRANSLATION

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - In fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - A string, a number, a memory location, a complex record.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Example:

Production	**Semantic Rule**	**Program Fragment**
L → E **return**	print(E.val)	print(val[top-1])
E → E^1 + T	E.val = E^1.val + T.val	val[ntop] = val[top-2] + val[top]
E → T	E.val = T.val	
T → T^1 * F	T.val = T^1.val * F.val	val[ntop] = val[top-2] * val[top]
T → F	T.val = F.val	
F → (E)	F.val = E.val	val[ntop] = val[top-1]
F → **digit**	F.val = **digit**.lexval	val[top] = digit.lexval

- Symbols E, T, and F are associated with an attribute *val*.
- The token **digit** has an attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
- The *Program Fragment* above represents the implementation of the semantic rule for a bottom-up parser.
- At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
- At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).
- The above model is suited for a desk calculator where the purpose is to evaluate and to generate code.

Intermediate Code Generation

- *Intermediate codes* are machine independent codes, but they are close to machine instructions.
- The given program in a source language is converted to an equivalent program in an intermediate language by the intermediate code generator.
- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.

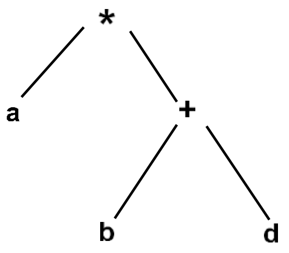
Syntax Tree

Syntax Tree is a variant of the Parse tree, where each leaf represents an operand and each interior node an operator.

Example:

Production	**Semantic Rule**
E → E1 **op** E2	E.val = NODE (op, E1.val, E2.val)
E → (E1)	E.val = E1.val
E → - E1	E.val = UNARY (- , E1.val)
E → **id**	E.val = LEAF (id)

A sentence **a*(b+d)** would have the following syntax tree:



Postfix Notation

Postfix Notation is another useful form of intermediate code if the language is mostly expressions.

Example:

Production	**Semantic Rule**	**Program Fragment**
E → E1 **op** E2	E.code = E1.code || E2.code || op	print op
E → (E1)	E.code = E1.code	
E → **id**	E.code = id	print id

Three Address Code

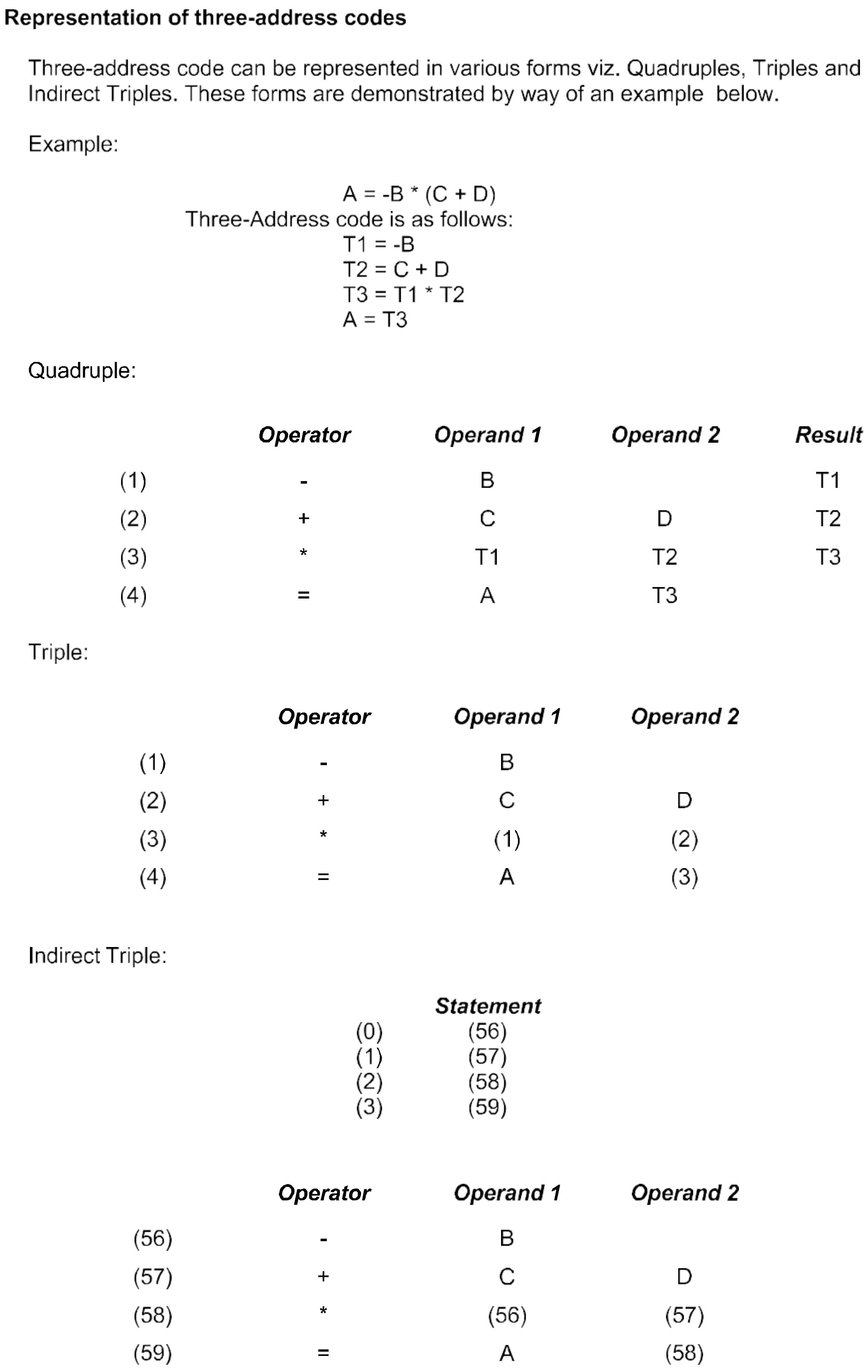
- We use the term "three-address code" because each statement usually contains three addresses (two for operands, one for the result).
- The most general kind of three-address code is:

  x := y ***op*** z

  where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.
- But we may also the following notation for quadraples (much better notation because it looks like a machine code instruction)

  op y,z,x

  apply operator op to y and z, and store the result in x.



Representation of three-address codes

Three-address code can be represented in various forms viz. Quadruples, Triples and Indirect Triples. These forms are demonstrated by way of an example below.

Example:

A = -B * (C + D)
Three-Address code is as follows:
T1 = -B
T2 = C + D
T3 = T1 * T2
A = T3

Quadruple:

	Operator	***Operand 1***	***Operand 2***	***Result***
(1)	-	B		T1
(2)	+	C	D	T2
(3)	*	T1	T2	T3
(4)	=	A	T3	

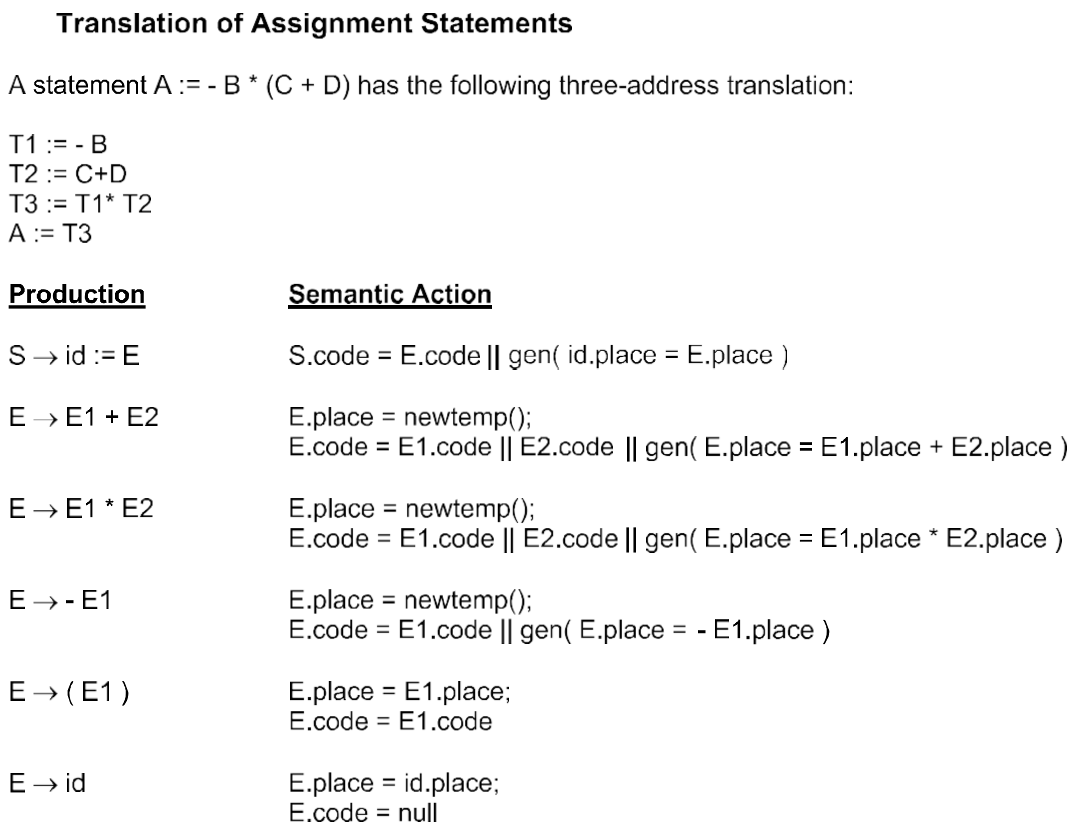
Triple:

	Operator	***Operand 1***	***Operand 2***
(1)	-	B	
(2)	+	C	D
(3)	*	(1)	(2)
(4)	=	A	(3)

Indirect Triple:

	Statement
(0)	(56)
(1)	(57)
(2)	(58)
(3)	(59)

	Operator	***Operand 1***	***Operand 2***
(56)	-	B	
(57)	+	C	D
(58)	*	(56)	(57)
(59)	=	A	(58)



Translation of Assignment Statements

A statement A := - B * (C + D) has the following three-address translation:

T1 := - B
T2 := C+D
T3 := T1* T2
A := T3

Production	**Semantic Action**
S → id := E	S.code = E.code || gen(id.place = E.place)

E → E1 + E2	E.place = newtemp();
	E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)

E → E1 * E2	E.place = newtemp();
	E.code = E1.code || E2.code || gen(E.place = E1.place * E2.place)

E → - E1	E.place = newtemp();
	E.code = E1.code || gen(E.place = - E1.place)

E → (E1)	E.place = E1.place;
	E.code = E1.code

E → id	E.place = id.place;
	E.code = null

INTERMEDIATE CODE GENERATION

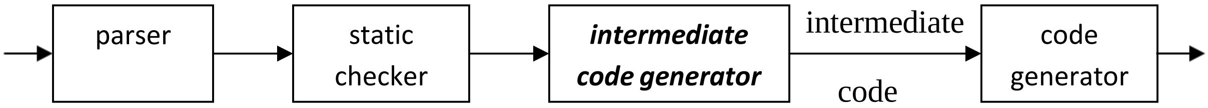
INTRODUCTION

The front end translates a source program into an intermediate representation from which the back end generates target code.

Benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated. That is, a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.

Position of intermediate code generator



INTERMEDIATE LANGUAGES

Three ways of intermediate representation:

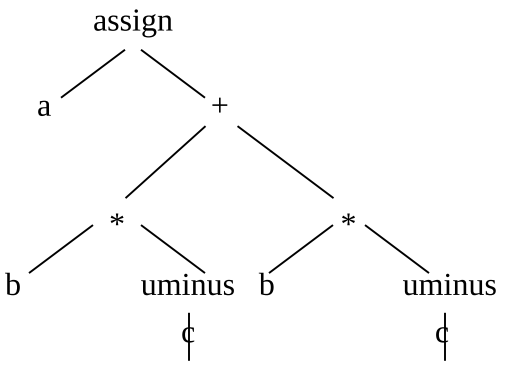
- Syntax tree
- Postfix notation
- Three address code

The semantic rules for generating three-address code from common programming language constructs are similar to those for constructing syntax trees or for generating postfix notation.

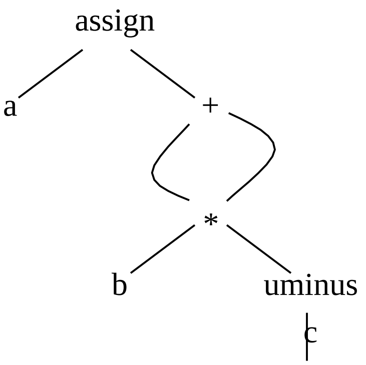
Graphical Representations:

Syntax tree:

A syntax tree depicts the natural hierarchical structure of a source program. A**dag (Directed Acyclic Graph)**gives the same information but in a more compact way because common subexpressions are identified. A syntax tree and dag for the assignment statementa **a : = b * - c + b * - c**are as follows:



(a) Syntax tree



(b) Dag

Postfix notation:

Postfix notation is a linearized representation of a syntax tree; it is a list of the nodes of the tree in which a node appears immediately after its children. The postfix notation for the syntax tree given above is

a b c uminus * b c uminus * + assign

Syntax-directed definition:

Syntax trees for assignment statements are produced by the syntax-directed definition. Non-terminal S generates an assignment statement. The two binary operators + and * are examples of the full operator set in a typical language. Operator associativities and precedences are the usual ones, even though they have not been put into the grammar. This definition constructs the tree from the input a : = b * - c + b* - c.

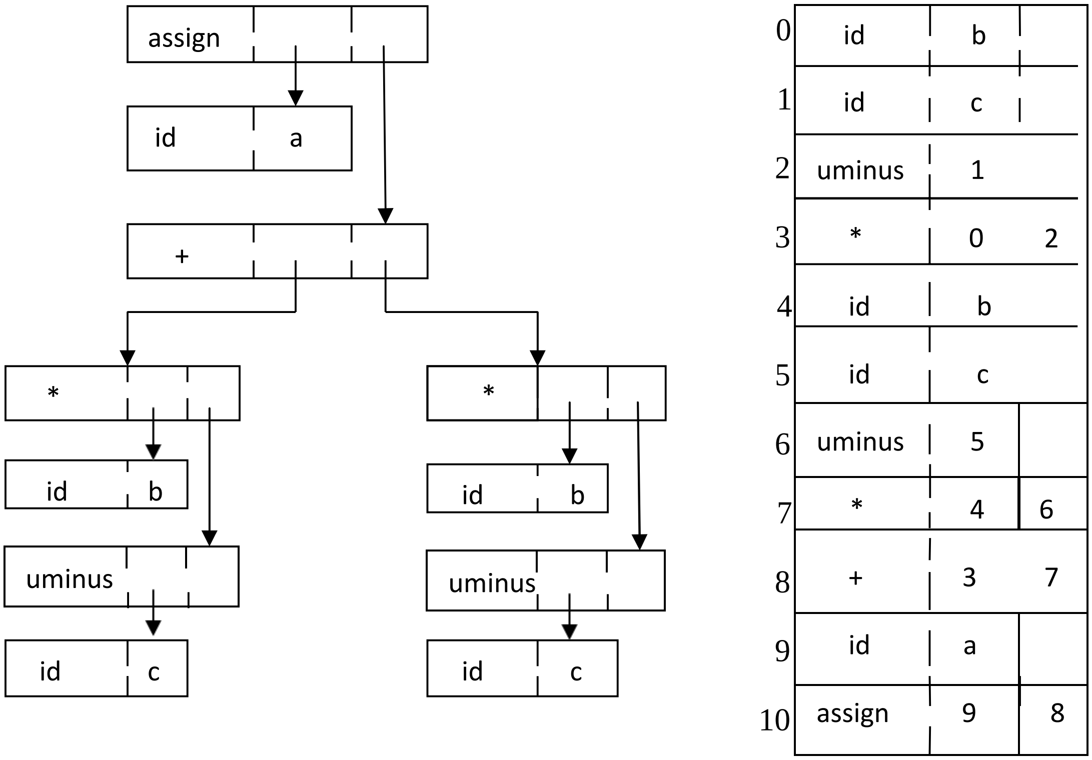
PRODUCTION	SEMANTIC RULE
S→id : = E	S.nptr : = mknode(‘assign’,mkleaf(id, id.place), E.nptr)
E→E_1 + E_2	E.nptr : = mknode(‘+’, E_1.nptr, E_2.nptr)
E→E_1 * E_2	E.nptr : = mknode(‘*’, E_1.nptr, E_2.nptr)
E→-E_1	E.nptr : = mknode(‘uminus’, E_1.nptr)
E→(E_1)	E.nptr : = E_1.nptr
E→id	E.nptr : = mkleaf(id, id.place)

Syntax-directed definition to produce syntax trees for assignment statements

The token**id**has an attribute*place*that points to the symbol-table entry for the identifier. A symbol-table entry can be found from an attribute**id**.*name*, representing the lexeme associated with that occurrence of**id.**If the lexical analyzer holds all lexemes in a single array of characters, then attribute*name*might be the index of the first character of the lexeme.

Two representations of the syntax tree are as follows. In (a) each node is represented as a record with a field for its operator and additional fields for pointers to its children. In (b), nodes are allocated from an array of records and the index or position of the node serves as the pointer to the node. All the nodes in the syntax tree can be visited by following pointers, starting from the root at position 10.

Two representations of the syntax tree



0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8

(a)

(b)

Three-Address Code:

Three-address code is a sequence of statements of the general form

x : = y*op*z

where x, y and z are names, constants, or compiler-generated temporaries;*op*stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on boolean-valued data. Thus a source language expression like x+ y*z might be translated into a sequence

$t_1 := y * z$

$t_2 := x + t_1$

where t_1 and t_2 are compiler-generated temporary names.

Advantages of three-address code:

- The unraveling of complicated arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target code generation and optimization.

- The use of names for the intermediate values computed by a program allows three-address code to be easily rearranged – unlike postfix notation.

Three-address code is a linearized representation of a syntax tree or a dag in which explicit names correspond to the interior nodes of the graph. The syntax tree and dag are represented by the three-address code sequences. Variable names can appear directly in three-address statements.

Three-address code corresponding to the syntax tree and dag given above

$t_1 := -c$	$t_1 := -c$
$t_2 := b * t_1$	$t_2 := b * t_1$
$t_3 := -c$	$t_5 := t_2 + t_2$
$t_4 := b * t_3$	$a := t_5$
$t_5 := t_2 + t_4$	
$a := t_5$	
(a) Code for the syntax tree	**(b) Code for the dag**

The reason for the term “three-address code” is that each statement usually contains three addresses, two for the operands and one for the result.

Types of Three-Address Statements:

The common three-address statements are:

1. Assignment statements of the form**x : = y*op*z**, where***op***is a binary arithmetic or logical operation.

2. Assignment instructions of the form**x : =*op*y**, where***op***is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number.

3.*Copy statements*of the form**x : = y**where the value of*y*is assigned to*x*.

4. The unconditional jump goto L. The three-address statement with label L is the next to be executed.

5. Conditional jumps such as**if*x relop y*goto L**. This instruction applies a relational operator (<, =, >=, etc.) to*x*and*y*, and executes the statement with label L next if*x*stands in relation

relop to y. If not, the three-address statement following if*x relop y*goto L is executed next, as in the usual sequence.

6.*param x*and*call p, n*for procedure calls and*return y*, where y representing a returned value is optional. For example,

param x_1
param x_2
. . .
param x_n
call p,n

generated as part of a call of the procedure p(x_1, x_2, …. ,x_n).

7. Indexed assignments of the form x : = y[i] and x[i] : = y.

8. Address and pointer assignments of the form x : = &y , x : = *y, and *x : = y.

Syntax-Directed Translation into Three-Address Code:

When three-address code is generated, temporary names are made up for the interior nodes of a syntax tree. For example,**id : =*E***consists of code to evaluate***E***into some temporary t, followed by the assignment**id**.*place*: =**t.**

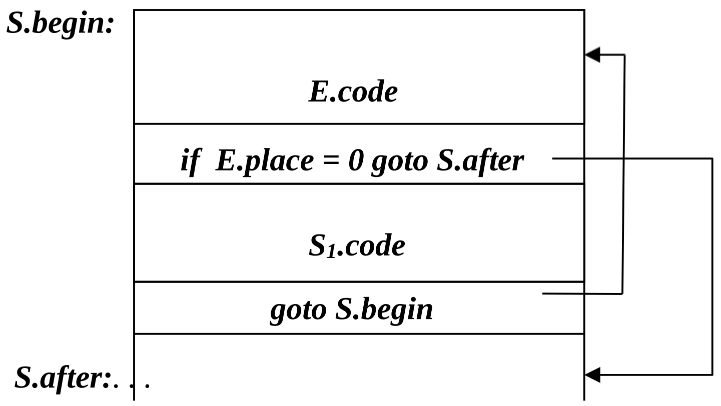
Given input a : = b * - c + b * - c, the three-address code is as shown above. The synthesized attribute*S*.*code*represents the three-address code for the assignment*S.*
The nonterminal*E*has two attributes :
1.*E.place*, the name that will hold the value of*E*, and
2.E. *code*, the sequence of three-address statements evaluating*E*.

Syntax-directed definition to produce three-address code for assignments

PRODUCTION	**SEMANTIC RULES**
⇨	
⇨	
⇨	
⇨	
⇨	
⇨	

S → id : = E	***S.code : = E.code||gen(id.place ‘:=’ E.place)***
E → E_1 + E_2	***E.place := newtemp;*** ***E.code := E_1.code||E $_2$.code||gen(E.place ‘:=’ E $_1$.place ‘+’ E_2.place)***
E → E_1 * E_2	***E.place := newtemp;*** ***E.code := E_1.code || E_2.code || gen(E.place ‘:=’ E_1.place ‘*’ E_2.place)***
E → - E_1	***E.place := newtemp;*** ***E.code := E_1.code || gen(E.place ‘:=’ ‘uminus’ E_1.place)***
E → (E_1)	***E.place : = E_1.place;*** ***E.code : = E_1.code***
E → id	***E.place : = id.place;*** ***E.code : = ‘ ‘***

Semantic rules generating code for a while statement



PRODUCTION	**SEMANTIC RULES**
S→ while*E*do*S* $_1$	***S.begin := newlabel;*** ***S.after := newlabel;*** ***S.code := gen(S.begin ':') ||*** ***E.code ||*** ***gen ('if' E.place '=' '0' 'goto' S.after)||*** ***S₁.code ||*** ***gen ('goto' S.begin) ||*** ***gen (S.after ':')***

- The function*newtemp*returns a sequence of distinct names t $_1$,t$_2$,….. in response to successive calls.
- Notation*gen(x ':=' y '+' z)*is used to represent three-address statement x := y + z. Expressions appearing instead of variables like*x*, *y*and*z*are evaluated when passed to *gen*, and quoted operators or operand, like '+' are taken literally.
- Flow-of–control statements can be added to the language of assignments. The code for***S*** **→while*E*do*S* $_1$** is generated using new attributes*S.begin*and*S.after*to mark the first statement in the code for*E*and the statement following the code for S, respectively.
- The function*newlabel*returns a new label every time it is called.
- We assume that a non-zero expression represents true; that is when the value of***E*** becomes zero, control leaves the while statement.

Implementation of Three-Address Statements:

A three-address statement is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are:

- Quadruples
- Triples
- Indirect triples

Quadruples:

- A quadruple is a record structure with four fields, which are,***op, arg1, arg2***and***result.***
- The*op*field contains an internal code for the operator. The three-address statement**x : = y op z**is represented by placing*y*in*arg1*,*z*in*arg2*and*x*in*result.*
- The contents of fields arg1, arg2 and result are normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

Triples:

- To avoid entering temporary names into the symbol table, we might refer to a temporary value by the position of the statement that computes it.
- If we do so, three-address statements can be represented by records with only three fields: *op, arg1*and*arg2.*
- The fields*arg1*and*arg2*, for the arguments of*op*, are either pointers to the symbol table or pointers into the triple structure (for temporary values).
- Since three fields are used, this intermediate code format is known as*triples*.

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	: =	t_3		a

(a) Quadruples

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

Quadruple and triple representation of three-address statements given above

A ternary operation like x[i] : = y requires two entries in the triple structure as shown as below while x : = y[i] is naturally represented as two operations.

	op	*arg1*	*arg2*
(0)	[ ] =	x	i
(1)	assign	(0)	y

	op	*arg1*	*arg2*
(0)	= [ ]	y	i
(1)	assign	x	(0)

(a) x[i] : = y **(b) x : = y[i]**

Indirect Triples:

- Another implementation of three-address code is that of listing pointers to triples, rather than listing the triples themselves. This implementation is called indirect triples.
- For example, let us use an array statement to list pointers to triples in the desired order. Then the triples shown above might be represented as follows:

	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

DECLARATIONS

As the sequence of declarations in a procedure or block is examined, we can lay out storage for names local to the procedure. For each local name, we create a symbol-table entry with information like the type and the relative address of the storage for the name. The relative address consists of an offset from the base of the static data area or the field for local data in an activation record.

Declarations in a Procedure:

The syntax of languages such as C, Pascal and Fortran, allows all the declarations in a single procedure to be processed as a group. In this case, a global variable, say*offset*, can keep track of the next available relative address.

In the translation scheme shown below:

- Nonterminal P generates a sequence of declarations of the form**id :*T.***
- Before the first declaration is considered,*offset*is set to 0. As each new name is seen , that name is entered in the symbol table with offset equal to the current value of*offset*, and*offset*is incremented by the width of the data object denoted by that name.
- The procedure*enter(name*, *type*, *offset*) creates a symbol-table entry for*name*, gives its type*type*and relative address*offset*in its data area.
- Attribute*type*represents a type expression constructed from the basic types*integer*and *real*by applying the type constructors*pointer*and*array*. If type expressions are represented by graphs, then attribute*type*might be a pointer to the node representing a type expression.
- The width of an array is obtained by multiplying the width of each element by the number of elements in the array. The width of each pointer is assumed to be 4.

Computing the types and relative addresses of declared names

P ⇒D	***{ offset : = 0 }***
D ⇒D ; D	

D ⇒id : T	***{ enter(id.name, T.type, offset);***
	offset : = offset + T.width }

T ⇒integer	***{ T.type : = integer;***
	T.width : = 4 }

T ⇒real	***{ T.type : = real;***
	T.width : = 8 }

T ⇒array [num] of T_1	***{ T.type : = array(num.val,*** T_1***.type);***
	T.width : = num.val X T_1***.width }***

T ⇒↑ T_1	***{ T.type : = pointer (*** T_1***.type);***
	T.width : = 4 }

Keeping Track of Scope Information:

When a nested procedure is seen, processing of declarations in the enclosing procedure is temporarily suspended. This approach will be illustrated by adding semantic rules to the following language:

P □ *D*

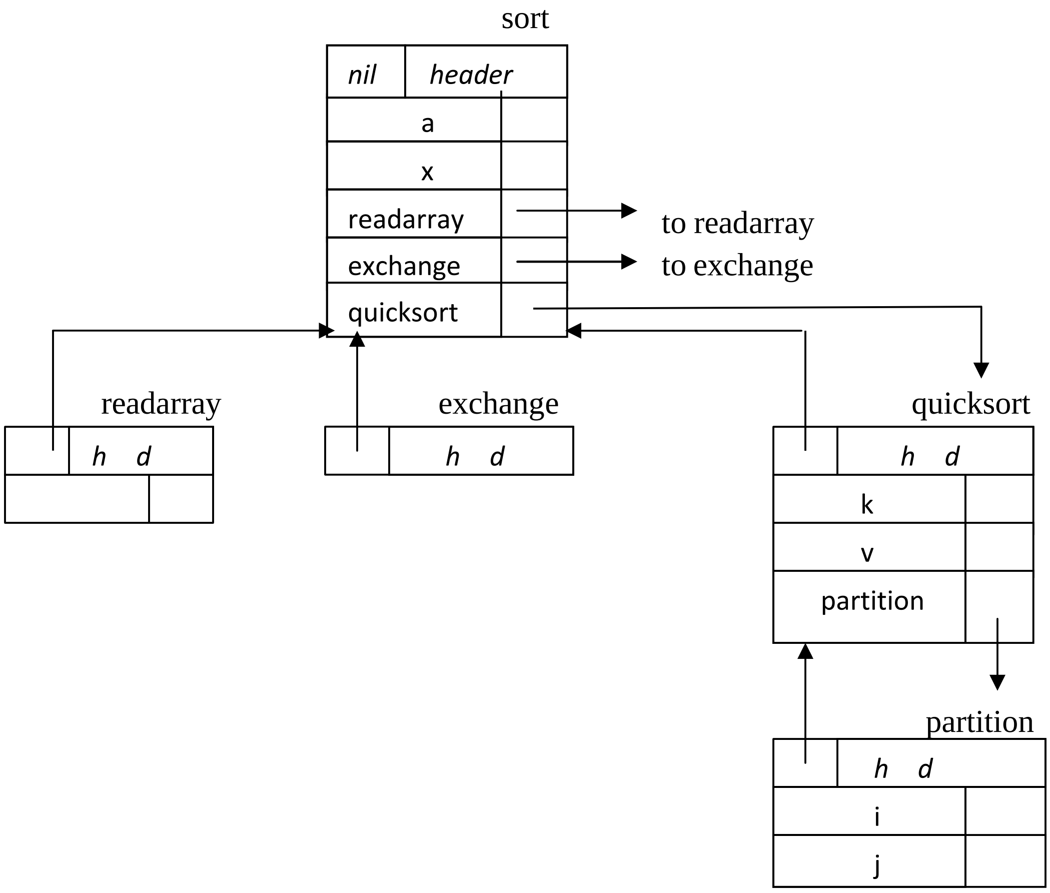
D □ *D ; D* |**id***: T* |**proc id***; D ; S*

One possible implementation of a symbol table is a linked list of entries for names.

A new symbol table is created when a procedure declaration*D* □ ***proc id****D* $_1$*;S*is seen, and entries for the declarations in D_1 are created in the new table. The new table points back to the symbol table of the enclosing procedure; the name represented by id itself is local to the enclosing procedure. The only change from the treatment of variable declarations is that the procedure*enter*is told which symbol table to make an entry in.

For example, consider the symbol tables for procedures*readarray, exchange,* and *quicksort*pointing back to that for the containing procedure*sort*, consisting of the entire program. Since*partition*is declared within*quicksort*, its table points to that of*quicksort*.

Symbol tables for nested procedures



The semantic rules are defined in terms of the following operations:

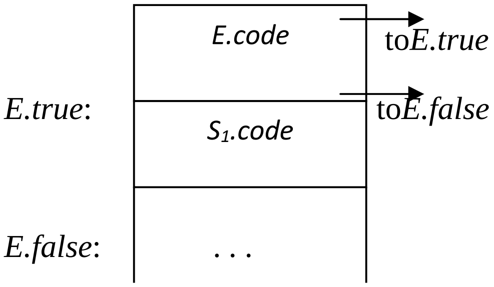
1. *mktable(previous)*creates a new symbol table and returns a pointer to the new table. The argument*previous*points to a previously created symbol table, presumably that for the enclosing procedure.

2. *enter(table, name, type, offset)*creates a new entry for name*name*in the symbol table pointed to by*table*.Again,*enter*places type*type*and relative address*offset*in fields within the entry.

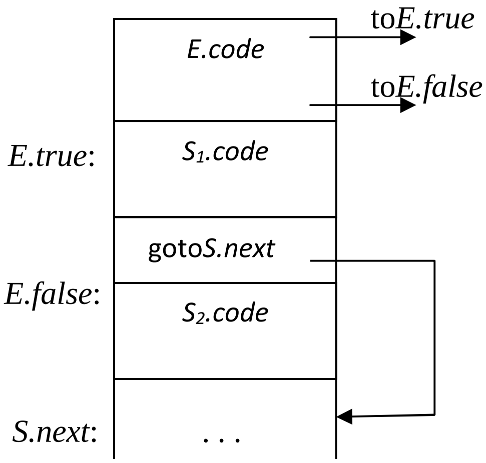
3. *addwidth(table, width)*records the cumulative width of all the entries in table in the header associated with this symbol table.

4. *enterproc(table, name, newtable)*creates a new entry for procedure*name*in the symbol table pointed to by*table*. The argument*newtable*points to the symbol table for this procedure *name*.

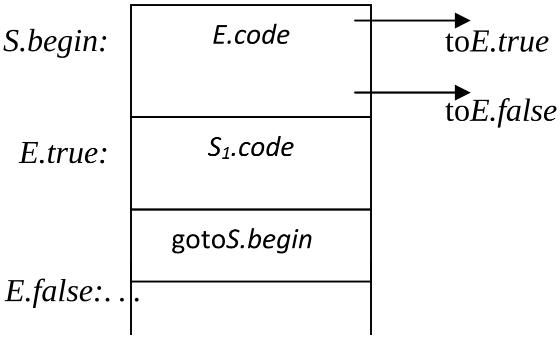
Code for if-then , if-then-else, and while-do statements



(a) if-then



(b) if-then-else



(c) while-do

Syntax-directed definition for flow-of-control statements

PRODUCTION	SEMANTIC RULES

S→**if** *E* **then** S_1	*E.true : = newlabel;*
	E.false : = S.next;
	S_1*.next : = S.next;*
	S.code : = E.code || *gen(E.true ‘:’)* || S_1*.code*

S→**if** *E* **then** S_1 **else** S_2	*E.true : = newlabel;* *E.false : = newlabel;* S_1*.next : = S.next;* S_2*.next : = S.next;* *S.code : = E.code* || *gen(E.true ‘:’)* || S_1*.code* || *gen(‘***goto***’ S.next)* || *gen(E.false ‘:’)* || S_2*.code*

S→**while** *E* **do** S_1	*S.begin : = newlabel;*
	E.true : = newlabel;
	E.false : = S.next;
	S_1*.next : = S.begin;*
	*S.code : = gen(S.begin ‘:’)*|| *E.code* ||
	gen(E.true ‘:’) || S_1*.code* ||
	*gen(‘***goto***’ S.begin)*

Control-Flow Translation of Boolean Expressions:

Syntax-directed definition to produce three-address code for booleans

PRODUCTION	SEMANTIC RULES
E→E_1 **or** E_2	E_1*.true : = E.true;* E_1*.false : = newlabel;* E_2*.true : = E.true;* E_2*.false : = E.false;* *E.code : =* E_1*.code* || *gen(*E_1*.false ‘:’)* || E_2*.code*
E→E_1 **and** E_2	*E.true : = newlabel;* E_1*.false : = E.false;* E_2*.true : = E.true;* E_2*.false : = E.false;* *E.code : =* E_1*.code* || *gen(*E_1*.true ‘:’)* || E_2*.code*

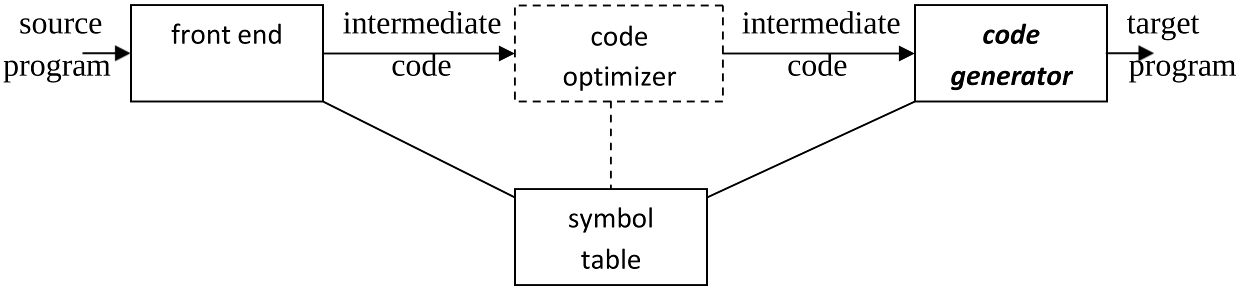
E→**not**E_1	E_1*.true : = E.false;* E_1*.false : = E.true;* *E.code : =* E_1*.code*
E→*(E1)*	E_1*.true : = E.true;*

	E_1*.false : = E.false;* *E.code : =* E_1*.code*
E→**id** $_1$ **relop id**$_2$	*E.code : = gen*(‘**if**’**id** $_1$.*place***relop.***op***id** $_2$.*place* ‘**goto**’*E.true)* || *gen*(‘**goto**’*E.false*)
E→**true**	*E.code : = gen(*‘**goto**’*E.true)*
E→**false**	*E.code : = gen(‘***goto***’ E.false)*

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.

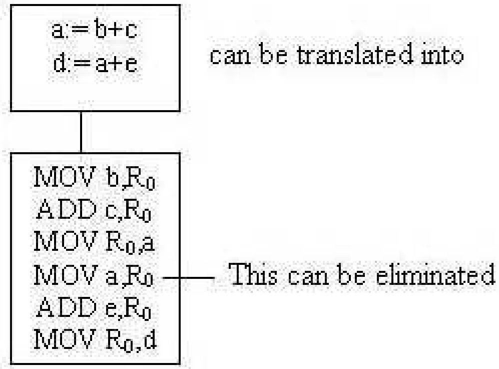
     ➢ if *i*>*j*, the jump is forward. We must store on a list for quadruple *i* the location of the first machine instruction generated for quadruple *j*. When *i* is processed, the machine locations for all instructions that forward jumps to *i* are filled.

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*ADDR***	***ESS*ADDED COST**	
*absolute*M		M	1
*register*R		R	0
indexed	*c*(R)*c*+*con*	*ents*(R)	1
*indirect register**	*contents*(R)	0	
*indirect indexed***c*(R)*contents*(*c*+		*contents*(R))	1

$literal\#c$	$c1$		

- 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, R_0
ADD c, R_0 cost = 6
MOV R_0, 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 $*R_1$, $*R_0$
ADD $*R_2$, $*R_0$ 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 */

GOTOcallee.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)     $t_1$ := 4* i
(4)     $t_2$ := a[$t_1$]          /*compute a[i] */
(5)     $t_3$ := 4* i
(6)     $t_4$ :=  b[$t_3$]         /*compute b[i] */
(7)     $t_5$ := $t_2$*$t_4$
(8)     $t_6$ := prod+$t_5$
(9)     prod := $t_6$
(10)    $t_7$ := i+1
(11)    i := $t_7$
(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:

```
a : = b + c                          a : = b + c
b : = a – d        ──────────→       b : = a - d
c : = b + c                          c : = b + c
d : = a – d                          d : = b
```

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 $\mathbf{t}_1$ and neither **b** nor **c** is $\mathbf{t}_2$.

2. Algebraic transformations:

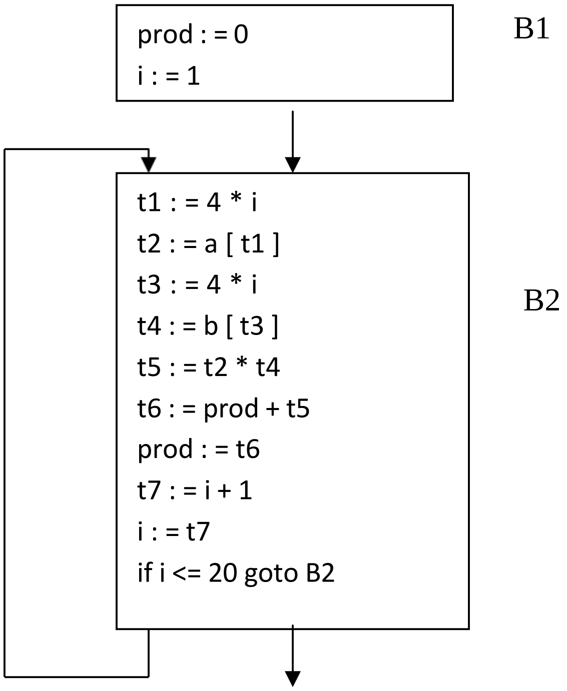
Algebraic transformations can be used to change the set of expressions computed by a basic block into an algebraically equivalent set.
Examples:
i) x : = x + 0 or x : = x * 1 can be eliminated from a basic block without changing the set of

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 : = *pand***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