2. A SIMPLE ONE-PASS COMPILER

Infix expression -> Postfix expression

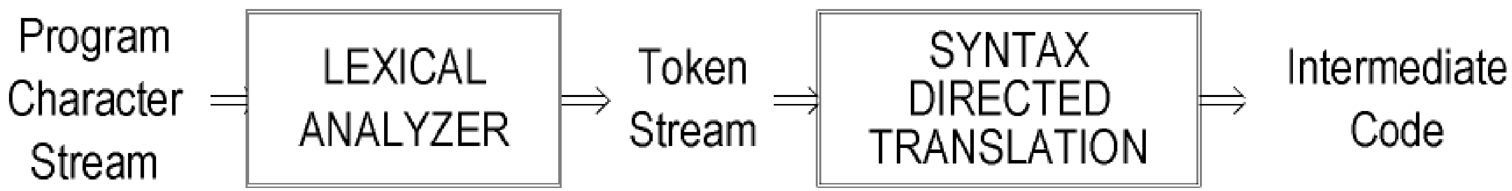
This chapter

- is an introduction to the material in cChapter 3 throught 8.
- concentrates on the front end of compiler, that is, on lexical analysis, parsing, and intermediate code generation.

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
    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 $S \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ is a derivation of α_n.

    Example, A derivation 1+9 from the grammar G
    - left most derivation

      $\underline{list} \Rightarrow \underline{list} + digit \Rightarrow \underline{digit} + digit \Rightarrow 1 + \underline{digit} \Rightarrow 1 + 9$
    - right most derivation

      $\underline{list} \Rightarrow list + \underline{digit} \Rightarrow \underline{list} + 9 \Rightarrow \underline{digit} + 9 \Rightarrow 1 + 9$
  - Language of grammar L(G)

    L(G) is a set of sentences that can be generated from the grammar G.

    $L(G)=\{x \mid S \Rightarrow^* x\}$ where $x \in$ a sequence of terminal symbols
  - Example: Consider a grammar G=(N,T,P,S):

    N={S} T={a,b}

    S=S P ={$S \rightarrow aSb \mid \varepsilon$ }
    - is aabb a sentecne of L(g)? (derivation of string aabb)

      $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aa\varepsilon bb \Rightarrow aabb$(or $S \Rightarrow^* aabb$) so, $aabb \in L(G)$
    - there is no derivation for aa, so $aa \notin L(G)$
    - note $L(G)=\{a_n b_n \mid n \geqq 0\}$ where $a_n b_n$ 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 $A \rightarrow x_1 \cdots x_n$ is derived, nodes labeled by $x_1 \cdots x_n$ 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

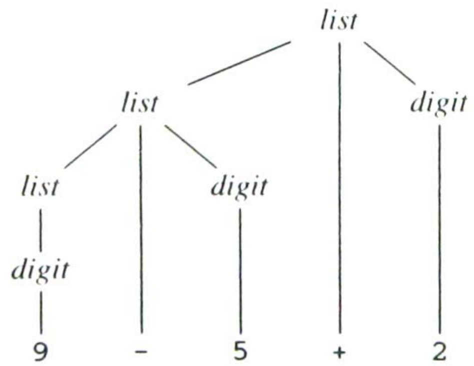


Fig. 2.2. Parse tree for 9-5+2 according to the grammar in Example 2.1.

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.

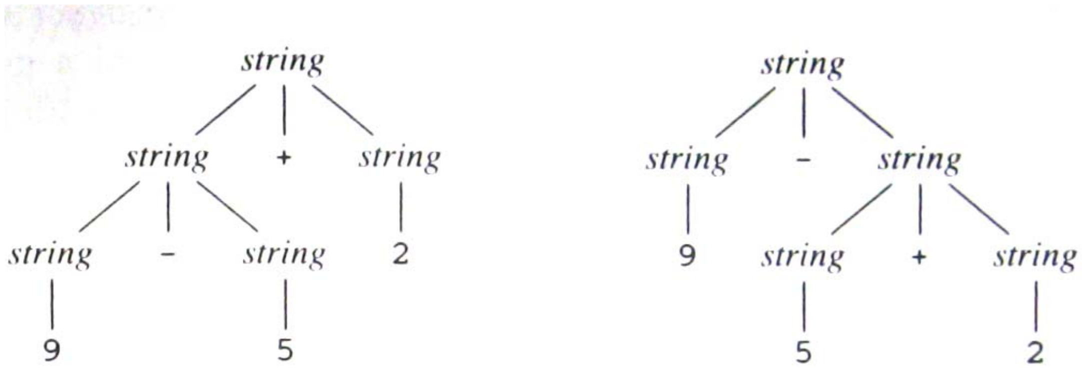
  - G : *string* → *string* + *string* | *string* - *string* |0|1|2|3|4|5|6|7|8|9
  - 

    Fig. 2.3. Two parse trees 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

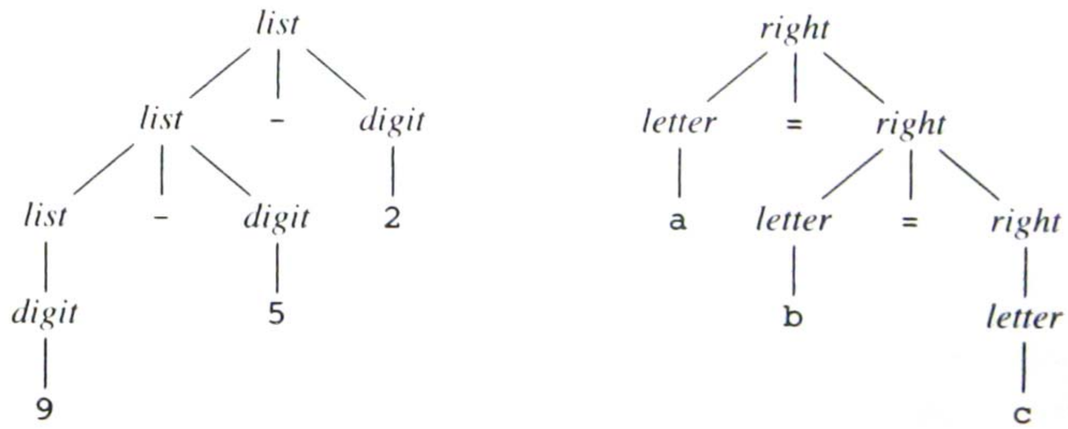


Fig. 2.4. Parse trees for left- and right-associative operators.

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 satements
  - *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 ($E.t \equiv E$).
- if E is an expression of the form E1 op E2 where op is a binary operator
  - E_1' is the postfix of E_1,
  - E_2' is the postfix of E_2
  - then E_1' E_2' op is the postfix for E_1 op E_2
- if E is (E_1), and E_1' is a postfix

  then E_1' is the postfix for E

eg) 9 - 5 + 2 $\Rightarrow$ 9 5 - 2 +

9 - (5 + 2) $\Rightarrow$ 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
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \parallel term.t \parallel '+'$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \parallel term.t \parallel '-'$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow 0$	$term.t := '0'$
$term \rightarrow 1$	$term.t := '1'$
. . .	. . .
$term \rightarrow 9$	$term.t := '9'$

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

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

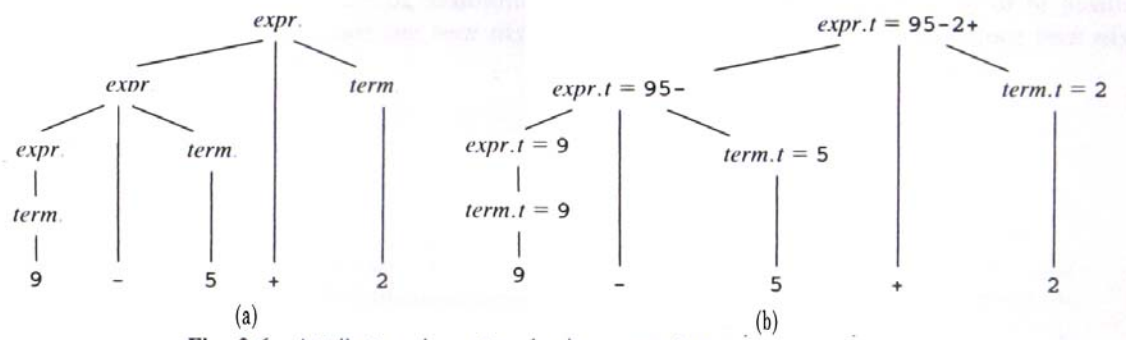


Fig. 2.6. Attribute values at nodes in a parse tree.

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* = *list.t* ‖ *term.t* ‖ "+"	*list* → *list* + *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.

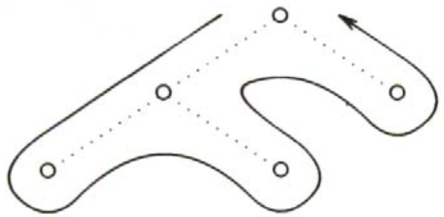


Fig. 2.11. Example of a depth-first traversal of a tree.

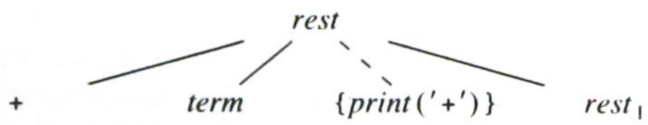


Fig. 2.12. An extra leaf is constructed for a semantic action.

Example 2.8.

- SDD vs. SDTS for infix to postfix translation.

productions	SDD	SDTS
expr → *list* + *term*	*expr.t* = *list.t* ǁ *term.t* ǁ "+"	*expr* → *list* + *term* printf{"+")}
expr → *list* + *term*	*expr.t* = *list.t* ǁ *term.t* ǁ "-"	*expr* → *list* + *term* printf{"-")}
expr → *term*	*expr.t* = *term.t*	*expr* → *term*
term → 0	*term.t* = "0"	*term* → 0 printf{"0")}
term → 1	*term.t* = "1"	*term* → 1 printf{"1")}
…	…	…
term → 9	*term.t* = "9"	*term* → 9 printf{"0")}

- Action translating for input 9-5+2

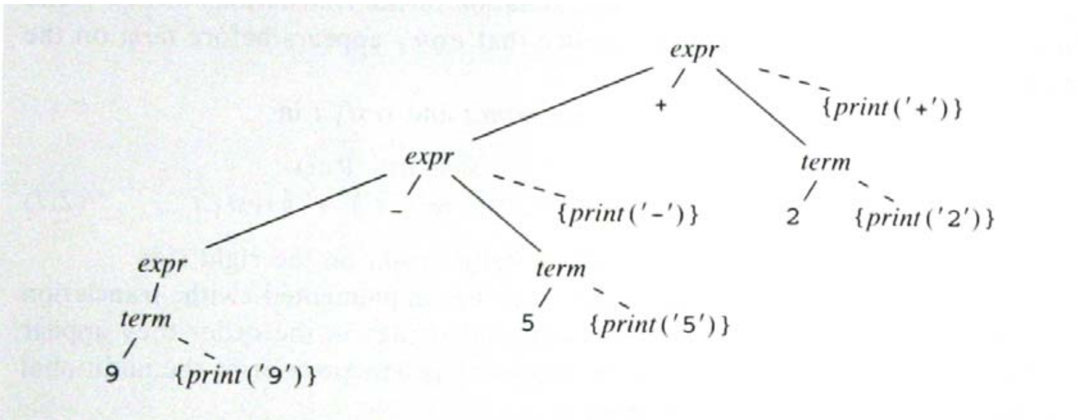


Fig. 2.14. Actions translating 9-5+2 into 95-2+.

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 $x \in L(G)$, 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**

Grammar G :

type → *simple*

| ↑ id

| array [*simple*] of type

simple → integer

| char

| num dotdot num

Input string

arrray [num dotdot num] of integer

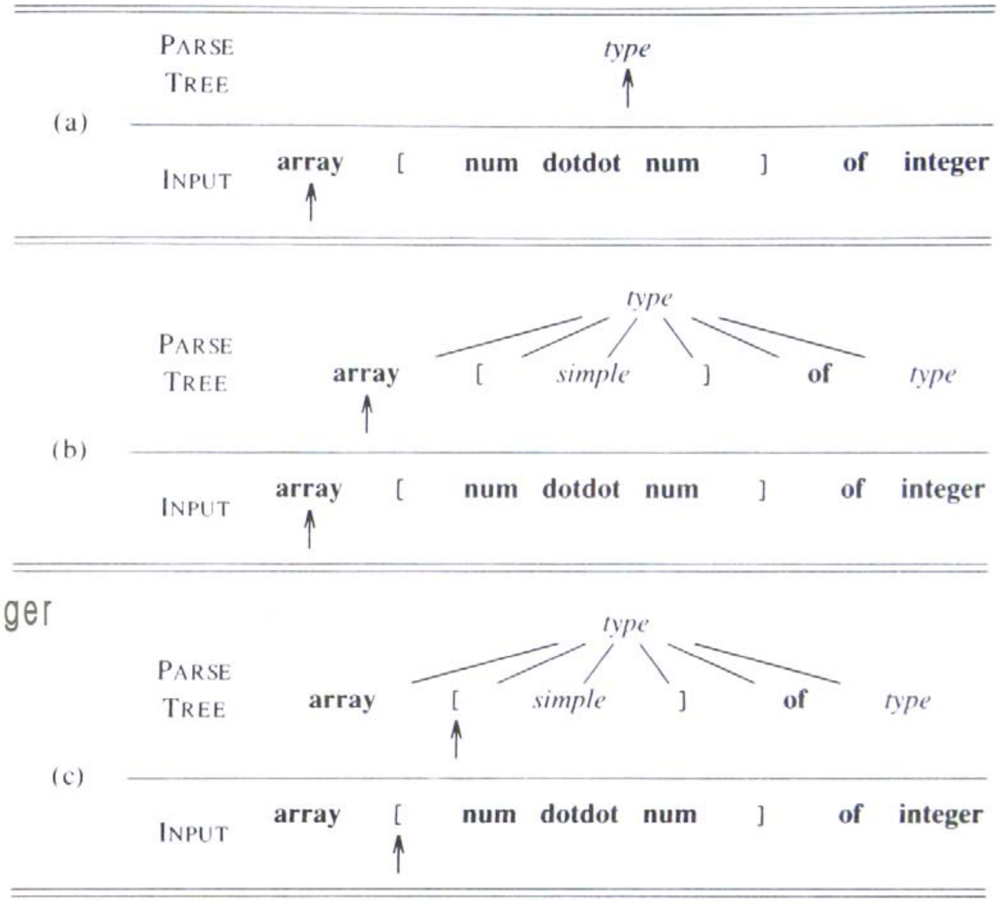


Fig. 2.16. Top-down parsing while scanning the input from left to right.

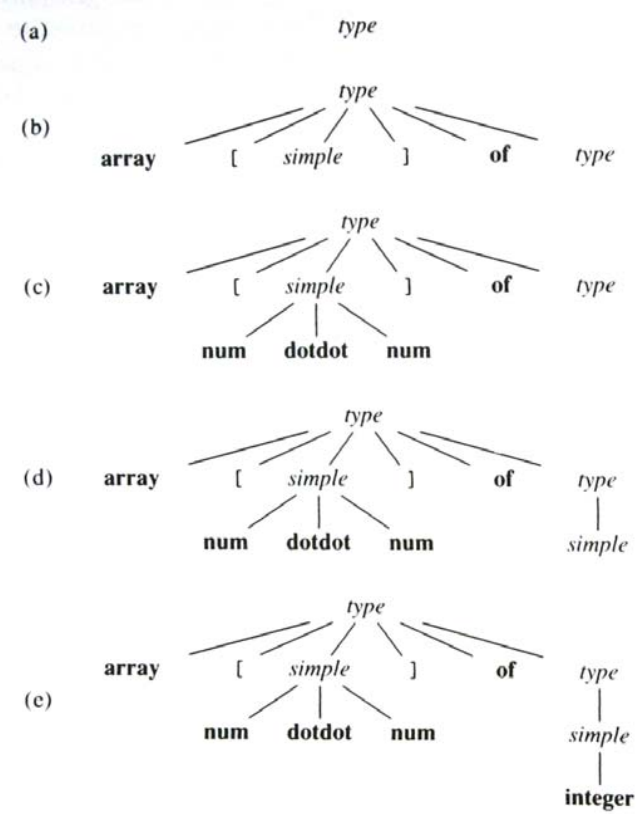


Fig. 2.15. Steps in the top-down construction of a parse tree.

- 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** $\in$ 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** $\neq$ **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.
  $\Rightarrow$
- It may fail to recognize correct string in some grammar G and is tedious in processing.
  $\Rightarrow$

- 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** $\in$ 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** $\neq$ **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. $A \rightarrow B\alpha$, or
  2. $A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid \cdots \mid a_n\alpha_n$
     ① for $1 \leqq i,\ j \leqq n$ and $i \neq j$, $ai \neq aj$
     ② A ε may also occur
        if none of ai can follow A in a derivation and if we have $A \rightarrow \varepsilon$
- 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

  $S \rightarrow a\alpha$ and $S \rightarrow a\beta$

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 { $S \rightarrow a\alpha$, $S \rightarrow a\beta$ } by

  { $S \rightarrow aS'$, $S' \rightarrow \alpha$, $S' \rightarrow \beta$ } cf. $S \rightarrow a(\alpha \mid \beta)$

  (Hopefully α and β start with different symbols)

- left factoring for G { $S \rightarrow aSb \mid c \mid ab$ }

  $S \rightarrow aS' \mid c$ cf. $S(=aSb \mid ab \mid c = a\ (\ Sb \mid b) \mid c\ ) \rightarrow a\ S' \mid c$

  $S' \rightarrow Sb \mid b$

- 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 feft 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
  $A \Rightarrow^{+} A\alpha$, where is any string.
  - Grammar {$S \rightarrow S\alpha$ | c} is left recursive because of $S \Rightarrow S\alpha$
  - Grammar {$S \rightarrow A\alpha$, $A \rightarrow Sb$ | c} is also left recursive because of $S \Rightarrow A\alpha \Rightarrow Sb\alpha$
- 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→ S A | 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→A S' and S'→A S'| ε

Example :

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

  =>

  $A \rightarrow \beta R$

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

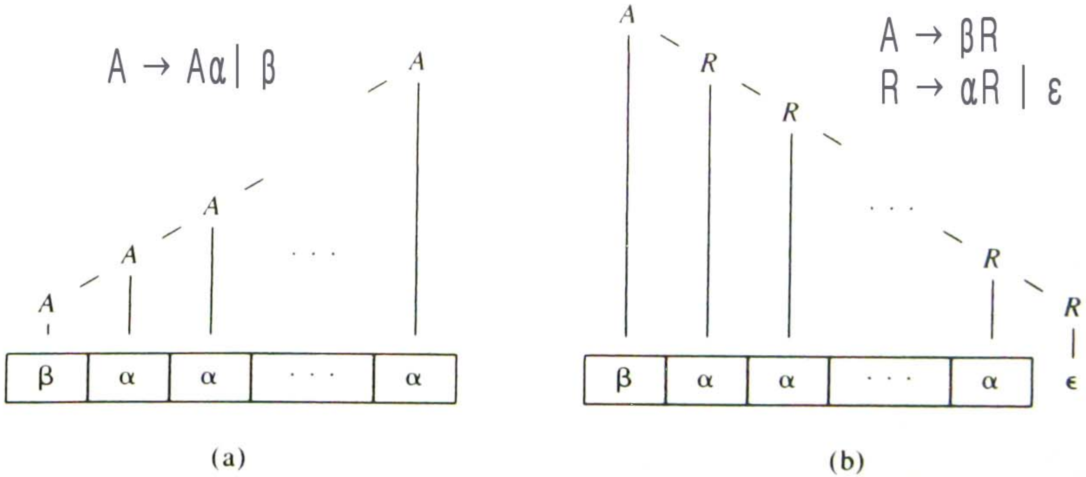


Fig. 2.18. Left- and right-recursive ways of generating a string.

- In general, the rule is that
  - If $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n$ and
    $A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$ (no β_i's start with A),
    then, replace by
    $A \rightarrow \beta_1 R \mid \beta_2 R \mid \cdots \mid \beta_m R$ and
    $Z \rightarrow \alpha_1 R \mid \alpha_2 R \mid \cdots \mid \alpha_n R \mid \varepsilon$

Exercise: Remove the left recursion in the following grammar:

expr → *expr* + *term* | *expr* - *term*
expr → *term*

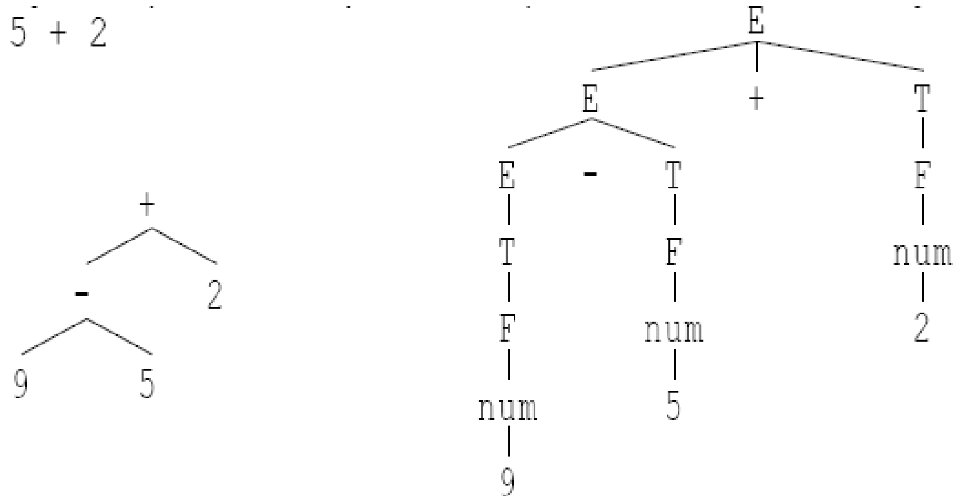
solution:

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

eg) 9 - 5 + 2



- Concrete syntax tree : parse tree Fig 2.2. p28.
- Abstract syntax tree, syntax tree : Fig 2.20.
- Concrete syntax : underlying grammar

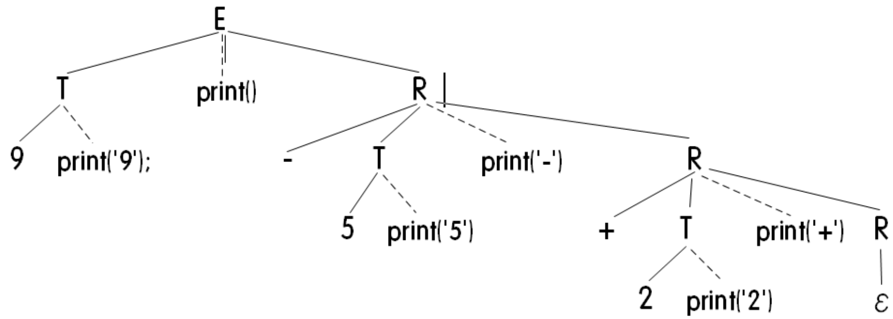
Adapting the Translation Scheme

- 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→ E − T {'−'} E→ T {} T→ 0{'0'}|⋯|9{'9'}	E→ T {} R R→ + T{'+'} R R→ − T{'−'} R 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.

Fig 2.19. 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")}

-

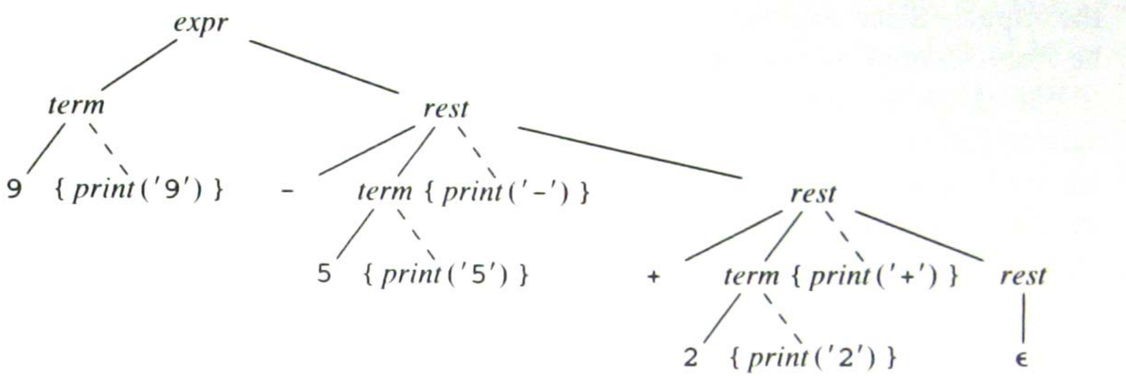


Fig. 2.21. Translation of 9-5+2 into 95-2+.

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. 2.22. Functions for the nonterminals *expr*, *rest*, and *term*.

```
match(t:tkoen)
{
  if(lookhead = t)
        lookahead lexan();
  else error();
}
```

Optimizer and Translator

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

⇒

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

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

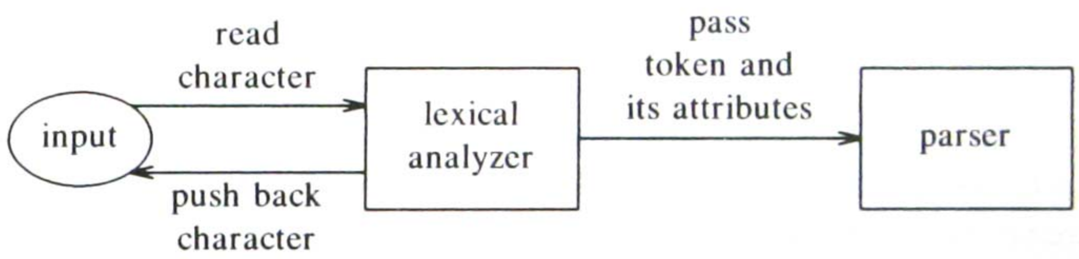


Fig. 2.25. Inserting a lexical analyzer between the input and the parser.

A Lexical Analyzer

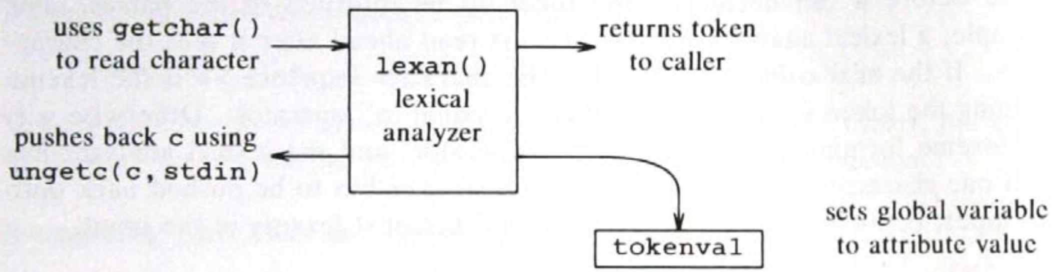


Fig. 2.26. Implementing the interactions in Fig. 2.25.

- 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)
             |     └── token
             └──────── lexeme

    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**

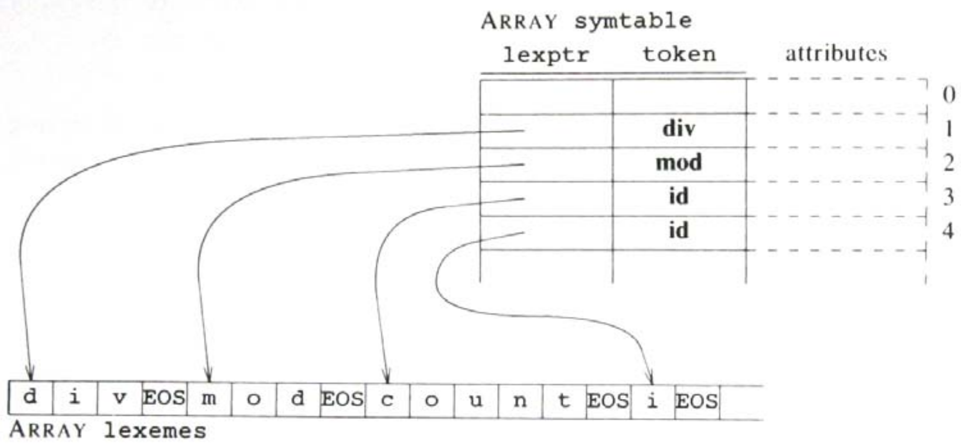


Fig. 2.29. Symbol table and array for storing strings.

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 ....

instruction memory

push 5
rvalue 2
+
rvalue 3
*

Data memory

1	0	
2	11	a
3	7	b

Stack

	1	2	3	4	5
	5	11	16	7	112
		5		16	

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) → $\boxed{\text{SDD/SDTS}}$ → 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 :
  1. Absolute target location
  2. Relative target location(distance :Current ↔Target)
  3. 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** *stmt*$_1$

  { *out* := *newlabel*[1)]

  stmt.t := *expr.t* || 'gofalse' *out* || *stmt*$_1$*.t* || 'label' *out* }

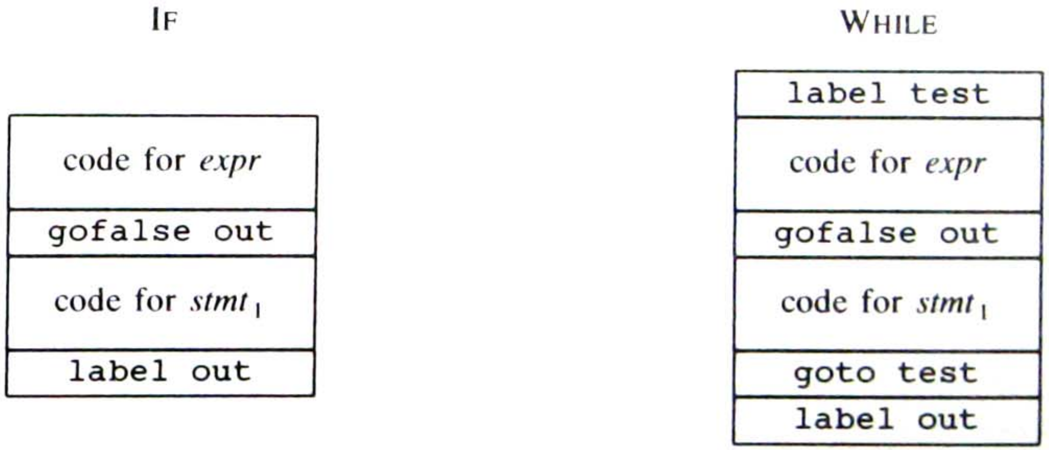


Fig. 2.33. Code layout for conditional and while statements.

- Translation scheme for while-statement ?

---

1) a procedure generates a unique label(eg. zzzzz001, zzzzz002, etc) whenever it is called!

Emitting a Translation

- Semantic Action(Tranaslation Scheme):

① *stmt* → **if**
expr { *out* := *newlabel*; *emit*('gofalse', *out*) }
then
$stmt_1$ { *emit*('label', *out*) }

② *stmt* → **id** { *emit*('lvalue', *id.lexeme*) }
:=
expr { *emit*(':=') }

③ *stmt* → **i**
expr { *out* := *newlabel*; *emit*('gofalse', *out*) }
then
$stmt_1$ { *emit*('label', *out*) ; *out1* := *newlabel*; *emit*('goto', *out 1*); }
else
$stmt_2$ { *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

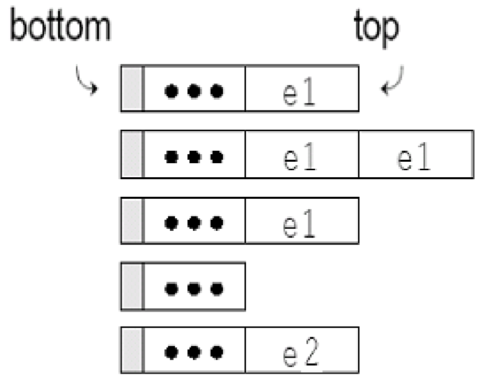
Control Flow with Analysis

- 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, Fig 2.35

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. 2.35. Specification for infix-to-postfix translator.

- Structure of the translator, Fig 2.36

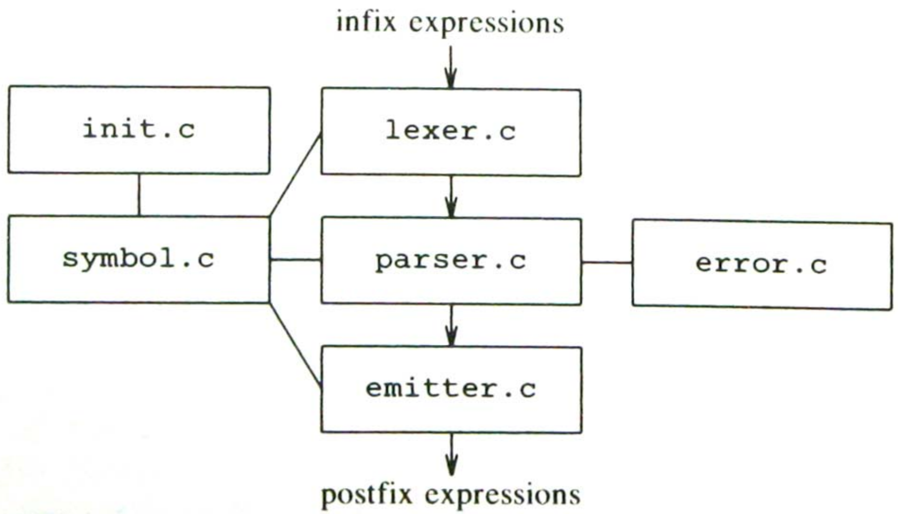


Fig. 2.36. Modules of infix-to-postfix translator.

o global header file "header.h"

The Lexical Analysis Module lexer.c

o 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. 2.37. Description of tokens.

The Parser Module parser.c

SDTS Fig 2.35
⇓ ← left recursion elimination
New SDTS Fig 2.38

```
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. 2.35. Specification for infix-to-postfix translator

```
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. 2.38 Syntax directed ranslation sceme after eliminating left-recursion

```
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. 2.35. Specification for infix-to-postfix translator

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

Fig. 2.38 Syntax directed ranslation sceme 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

+