Compiler Construction: Principles and Practice
by Kenneth C. Louden

Chapter 4 Exercise Answers

Exercise 4.4

```
function lexp : integer ;
var temp : integer ;
    optemp : TokenType ;
begin
  if token = number then
    temp := value(token) ;
    match(number) ;
  else if token = ( then
    match( ( ) ;
    optemp := op ;
    temp := lexpSeq(optemp) ;
    match( ) ) ;
  end if ;
  return temp ;
end lexp ;

function op : TokenType ;
var optemp : TokenType ;
begin
  optemp := token ;
  case token of
      + , - , * : match (token) ;
      else error ;
  end case ;
  return optemp ;
end op ;

function lexpSeq( op : TokenType ) ;
var temp : integer ;
begin
  temp := lexp ;
  while token = number or token = ( do
    case op of
      + : temp := temp + lexp ;
      - : temp := temp – lexp ;
      * : temp := temp * lexp ;
    end case ;
  end while ;
  return temp ;
end lexpSeq ;
```

New features of the above pseudocode are (1) the *op* parameter to *lexpSeq*, which is necessary to allow it compute its value based on the current operator (a more complex alternative would be for *lexpSeq* to return some kind of list of integer values, and let *lexp* handle the computation); and (2) the use of implicit token information about *lexp* (First(*lexp*) -- see Exercise 4.8) to provide a test for the loop inside *lexpSeq* (the presence of a number or left parenthesis indicates that an *lexp* is about to begin).

Exercise 4.6

(a)

	Parsing stack	Input	Action
1	$ *S*	(( )) ( ) $	*S* → (*S*) *S*
2	$ *S*) *S* (	(( )) ( ) $	match
3	$ *S*) *S*	( )) ( ) $	*S* → (*S*) *S*
4	$ *S*) *S*) *S* (	( )) ( ) $	match
5	$ *S*) *S*) *S*	)) ( ) $	*S* → ε
6	$ *S*) *S*)	)) ( ) $	match
7	$ *S*) *S*	) ( ) $	*S* → ε
8	$ *S*)	) ( ) $	match
9	$ *S*	( ) $	*S* → (*S*) *S*
10	$ *S*) *S* (	( ) $	match
11	$ *S*) *S*	) $	*S* → ε
12	$ *S*)	) $	match
13	$ *S*	$	*S* → ε
14	$	$	accept

Exercise 4.8

(a) The only rule that changes is that of *lexp-seq*:

lexp-seq → *lexp lexp-seq′*
lexp-seq′ → *lexp lexp-seq′* | ε

(b) First(*lexp*) = { **`number`**, **`identifier`**, **`(`** }
First(*atom*) = { **`number`**, **`identifier`** }
First(*list*) = { **`(`** }
First(*lexp-seq*) = { **`number`**, **`identifier`**, **`(`** }
First(*lexp-seq′*) = { **`number`**, **`identifier`**, **`(`** , ε }

Follow(*lexp*) = { $, **`)`** , **`number`**, **`identifier`**, **`(`** }
Follow(*atom*) = { $, **`)`** , **`number`**, **`identifier`**, **`(`** }
Follow(*list*) = { $, **`)`** , **`number`**, **`identifier`**, **`(`** }
Follow(*lexp-seq*) = { **`)`** }

Follow(*lexp-seq′*) = { **)** }

(c) One can either use the definition (page 155) and the result of part (d) below, or the Theorem (page 178) and the fact that First(*atom*) ∩ First(*list*) = ϕ and First(*lexp lexp-seq′*) ∩ First(ε) = First(*lexp*) ∩ First(ε) = ϕ, as well as First(*lexp-seq′*) ∩ Follow(*lexp-seq′*) = ϕ.

(d) In the following table we abbreviate ***number*** to ***num*** and ***identifier*** to ***id***.

M[*N*,*T*]	***num***	***id***	**(**	**)**	$
lexp	*lexp* → *atom*	*lexp* → *atom*	*lexp* → *list*		
atom	*atom* → ***num***	*atom* → ***id***			
list			*list* → **(** *lexp-seq* **)**		
lexp-seq	*lexp-seq* → *lexp lexp-seq′*	*lexp-seq* → *lexp lexp-seq′*	*lexp-seq* → *lexp lexp-seq′*		
lexp-seq′	*lexp-seq′* → *lexp lexp-seq′*	*lexp-seq′* → *lexp lexp-seq′*	*lexp-seq′* → *lexp lexp-seq′*	*lexp-seq′* → ε	

(e) In the following table we abbreviate ***number*** to ***num***, ***identifier*** to ***id***, *lexp* to *E*, *atom* to *A*, *list* to *L*, *lexp-seq* to *S*, and *lexp-seq′* to *S′*.

Parsing stack	Input	Action
$ *E*	**(a(b(2))(c))**$	*E* → *L*
$ *L*	**(a(b(2))(c))**$	*L* → **(** *S* **)**
$ **)** *S* **(**	**(a(b(2))(c))**$	match
$ **)** *S*	**a(b(2))(c))**$	*S* → *E S′*
$ **)** *S′ E*	**a(b(2))(c))**$	*E* → *A*
$ **)** *S′ A*	**a(b(2))(c))**$	*A* → ***id***
$ **)** *S′* ***id***	**a(b(2))(c))**$	match
$ **)** *S′*	**(b(2))(c))**$	*S′* → *E S′*
$ **)** *S′ E*	**(b(2))(c))**$	*E* → *L*
$ **)** *S′ L*	**(b(2))(c))**$	*L* → **(** *S* **)**
$ **)** *S′* **)** *S* **(**	**(b(2))(c))**$	match
$ **)** *S′* **)** *S*	**b(2))(c))**$	*S* → *E S′*
$ **)** *S′* **)** *S′ E*	**b(2))(c))**$	*E* → *A*
$ **)** *S′* **)** *S′ A*	**b(2))(c))**$	*A* → ***id***
$ **)** *S′* **)** *S′* ***id***	**b(2))(c))**$	match
$ **)** *S′* **)** *S′*	**(2))(c))**$	*S′* → *E S′*
$ **)** *S′* **)** *S′ E*	**(2))(c))**$	*E* → *L*
$ **)** *S′* **)** *S′ L*	**(2))(c))**$	*L* → **(** *S* **)**
$ **)** *S′* **)** *S′* **)** *S* **(**	**(2))(c))**$	match
$ **)** *S′* **)** *S′* **)** *S*	**2))(c))**$	*S* → *E S′*
$ **)** *S′* **)** *S′* **)** *S′ E*	**2))(c))**$	*E* → *A*
$ **)** *S′* **)** *S′* **)** *S′ A*	**2))(c))**$	*A* → ***num***

$) S') S' **num**	2)) (c)) $	match
$) S') S'	)) (c)) $	$S' \to \varepsilon$
$) S') S')	)) (c)) $	match
$) S') S'	) (c)) $	$S' \to \varepsilon$
$) S')	) (c)) $	match
$) S'	(c)) $	$S' \to E\ S'$
$) S' E	(c)) $	$E \to L$
$) S' L	(c)) $	$L \to$ (S)
$) S') S (	(c)) $	match
$) S') S	c)) $	$S \to E\ S'$
$) S') S' E	c)) $	$E \to A$
$) S') S' A	c)) $	$A \to$ **id**
$) S') S' **id**	c)) $	match
$) S') S'	)) $	$S' \to \varepsilon$
$) S')	)) $	match
$) S'	) $	$S' \to \varepsilon$
$)	) $	match
$	$	accept

Exercise 4.12

(a) An LL(1) grammar cannot be ambiguous, since the LL(1) algorithm constructs a unique leftmost derivation.

(b) An ambiguous grammar cannot be LL(1), for the same reason as (a): if it were LL(1), then a unique leftmost derivation would exist for every legal string, and the grammar could not be ambiguous.

(c) If the grammar is not ambiguous, it may still fail to be LL(1). Indeed, any unambiguous left recursive grammar is an example of such a grammar. (Of course, given an unambiguous grammar, there may be an equivalent grammar that *is* LL(1), but this is not always the case either.)

Exercise 4.16

(a) Given a grammar G, define a nonterminal A to be **useful** if there exists a derivation $S \Rightarrow^* \alpha A \beta \Rightarrow^* w$, where S is the start symbol and w is a string of terminals (so w is in $L(G)$). Then define a nonterminal A to be **useless** if it is not useful.

(b) Useless nonterminals contribute nothing to the language defined by the grammar, so they are unlikely to appear in a programming language grammar. Indeed, the existence of a useless nonterminal in a programming language grammar is a design error.

(c) Here is a simple grammar that demonstrates this phenomenon:

$S \rightarrow$ **(** S **)** | ε
$B \rightarrow B$ **(** | S

In this grammar, B is a useless nonterminal, and this grammar generates the same language as the grammar $S \rightarrow$ **(** S **)** | ε (the strings **(((...)))**). But First(S) = { **(** , ε } and Follow(S) = { $, **)** , **(** } (since **(** is in Follow(B) and Follow(A) contains Follow(B)). Thus First(S) ∩ Follow(S) is not empty, and yet there is no parsing conflict, because there is no derivation $S \Rightarrow^* \alpha\ S$ **(** β, so given lookahead **(** the rule $S \rightarrow$ ε should never be chosen.

Exercise 4.18

Suppose a grammar is LL(1), and suppose a is in First(α_1) ∩ First(α_2), for productions $A \rightarrow \alpha_1$ and $A \rightarrow \alpha_2$. Then there must be derivations $\alpha_1 \Rightarrow^* a\ \gamma_1$ and $\alpha_2 \Rightarrow^* a\ \gamma_2$, so both productions get added to $M[A, a]$ by rule 1 on page 154, contrary to the LL(1) requirement. Suppose now there is an a in First(A) ∩ Follow(A) for some A such that First(A) contains ε. By the theorem on page 169, there exists a grammar rule $A \rightarrow \alpha$ and a derivation $\alpha \Rightarrow^* \varepsilon$. Also, by the definition of Follow sets, there must exist a derivation $B \Rightarrow^* \beta\ A\ a\ \gamma$ for some nonterminal B. Since B is not useless, there also exists a derivation $S \Rightarrow^* \delta\ B\ \eta$, so $S\ \$ \Rightarrow^* \delta\ \beta\ A\ a\ \gamma\ \eta\ \$$, which means by condition 2 on page 154 that $A \rightarrow \alpha$ is added to $M[A, a]$. But since a is also in First(A), there must be a grammar rule $A \rightarrow \nu$ with a in First(ν), so this rule also gets added to $M[A, a]$, again contradicting the LL(1) property of the grammar. The proof is complete.

Exercise 4.20

(a) Since *exp* and *exp-list* have no grammar rules given for them, we may consider them to be terminals; however, they play no role in the First or Follow sets of the other nonterminals, so we will ignore them in this exercise. The First sets of *statement* and *statement′* are

First(*statement*) = { **identifier**, **other** }
First(*statement′*) = { **:=**, **(** }

and Follow(*statement*) = Follow(*statement′*) = { $ }. (Note that the Follow sets will not be needed to construct the LL(1) parsing table, since there are no ε-productions.

(b) In the following table we abbreviate **identifier** to **id**.

M[*N*,*T*]	**id**	**other**	**:=**	**(**	**)**	$
statement	*statement* → **id** *statement′*	*statement* → **other**				
statement′			*statement′* → **:=** *exp*	*statement′* → **(** *exp-list* **)**		

Exercise 4.22

program → *stmt-sequence*
stmt-sequence → *statement* { **;** *statement* }
statement → *if-stmt* | *repeat-stmt* | *assign-stmt* | *read-stmt* | *write-stmt*
if-stmt → **if** *bexp* **then** *stmt-sequence* [**else** *stmt-sequence*] **end**
repeat-stmt → **repeat** *stmt-sequence* **until** *bexp*
assign-stmt → ***identifier*** **:=** *aexp*
read-stmt → **read** ***identifier***
write-stmt → **write** *aexp*
bexp → *aexp comparison-op aexp*
comparison-op → **<** | **=**
aexp → *term* { *addop term* }
addop → **+** | **-**
term → *factor* { *mulop factor* }
mulop → ***** | **/**
factor → **(** *aexp* **)** | ***number*** | ***identifier***

Exercise 4.26

In the following table of actions we use the same conventions as in Table 4.10, page 187, and in addition we abbreviate ***number*** to ***n*** , *addop* to *A*, *mulop* to *M*, and *factor* to *F*.

Parsing stack	Input	Action
$ *E*	(2+-3)*4-+5 $	*E* → *T E′*
$ *E′ T*	(2+-3)*4-+5 $	*T* → *F T′*
$ *E′ T′ F*	(2+-3)*4-+5 $	*F* → **(** *E* **)**
$ *E′ T′* **)** *E* **(**	(2+-3)*4-+5 $	match
$ *E′ T′* **)** *E*	2+-3)*4-+5 $	*E* → *T E′*
$ *E′ T′* **)** *E′ T*	2+-3)*4-+5 $	*T* → *F T′*
$ *E′ T′* **)** *E′ T′ F*	2+-3)*4-+5 $	*F* → ***n***
$ *E′ T′* **)** *E′ T′* ***n***	2+-3)*4-+5 $	match
$ *E′ T′* **)** *E′ T′*	+-3)*4-+5 $	*T′* → ε
$ *E′ T′* **)** *E′*	+-3)*4-+5 $	*E′* → *A T E′*
$ *E′ T′* **)** *E′ T A*	+-3)*4-+5 $	*A* → **+**
$ *E′ T′* **)** *E′ T* **+**	+-3)*4-+5 $	match
$ *E′ T′* **)** *E′ T*	-3)*4-+5 $	pop (error)
$ *E′ T′* **)** *E′*	-3)*4-+5 $	*E′* → *A T E′*
$ *E′ T′* **)** *E′ T A*	-3)*4-+5 $	*A* → **-**
$ *E′ T′* **)** *E′ T* **-**	-3)*4-+5 $	match
$ *E′ T′* **)** *E′ T*	3)*4-+5 $	*T* → *F T′*
$ *E′ T′* **)** *E′ T′ F*	3)*4-+5 $	*F* → ***n***
$ *E′ T′* **)** *E′ T′* ***n***	3)*4-+5 $	match

$ *E′ T′* **)** *E′ T′*	**)*4-+5** $	$T' \to \varepsilon$
$ *E′ T′* **)** *E′*	**)*4-+5** $	$E' \to \varepsilon$
$ *E′ T′* **)**	**)*4-+5** $	match
$ *E′ T′*	***4-+5** $	$T' \to M\ F\ T'$
$ *E′ T′ F M*	***4-+5** $	$M \to$ *
$ *E′ T′ F* *	***4-+5** $	match
$ *E′ T′ F*	**4-+5** $	$F \to$ *n*
$ *E′ T′* ***n***	**4-+5** $	match
$ *E′ T′*	**-+5** $	$T' \to \varepsilon$
$ *E′*	**-+5** $	$E' \to A\ T\ E'$
$ *E′ T A*	**-+5** $	$A \to$ -
$ *E′ T* -	**-+5** $	match
$ *E′ T*	**+5** $	pop (error)
$ *E′*	**+5** $	$E' \to A\ T\ E'$
$ *E′ T A*	**+5** $	$A \to$ +
$ *E′ T* +	**+5** $	match
$ *E′ T*	**5** $	$T \to F\ T'$
$ *E′ T′ F*	**5** $	$F \to$ *n*
$ *E′ T′* ***n***	**5** $	match
$ *E′ T′*	$	$T' \to \varepsilon$
$ *E′*	$	$E' \to \varepsilon$
$	$	accept

Exercise 4.28

(a) In the code for **stmt_sequence** on page 189, the first call to **statement** matches the assignment statement **x:=2**. Then, since the next token is an identifier, the while-loop test succeeds, and a call to **match(SEMI)** is made. Since the next token is not a semicolon, an error message is generated, but no token is consumed. Then the second call to **statement** is made (inside the loop), which matches the assignment **y:=x+2**. Thus, the correct syntax tree is constructed, and the overall effect is that of "inserting" the missing semicolon.

(b) The answer to this question is found by examining the code in Appendix B (or by running the TINY compiler). As in part (a), a call is first made to **statement**, and this results in a call to **assign_stmt** (page 517, line 966). Within **assign_stmt**, first an identifier and then the **ASSIGN** token (which is missing) is matched (page 518, lines 1004-1005). As before, the call **match(ASSIGN)** generates an error message, but does not consume a token. Thus, the correct syntax tree would be constructed for this example as well (assuming of course that we really did want **x 2** to be an assignment).

(c) In both cases, the new code would still generate a tree for the assignment **x:=2**. After that, however, the test for the while-loop would fail and the statement **y:=x+2** would never be parsed (resulting in the error "Code ends before file").