Compiler Construction: Principles and Practice

by Kenneth C. Louden

Chapter 3 Exercise Answers

Exercise 3.4

(a) *rexp* ⇒ *rexp* "*****" ⇒ "**(**" *rexp* "**)**" "*****" ⇒ "**(**" *rexp* "**|**" *rexp* "**)**" "*****"
⇒ "**(**" *rexp rexp* "**|**" *rexp* "**)**" "*****" ⇒ "**(**" ***letter*** *rexp* "**|**" *rexp* "**)**" "*****"
⇒ "**(**" ***letter letter*** "**|**" *rexp* "**)**" "*****"
⇒ "**(**" ***letter letter*** "**|**" ***letter*** "**)**" "*****"

(b) An alternative leftmost derivation for the expression of (a) is

rexp ⇒ *rexp* "*****" ⇒ "**(**" *rexp* "**)**" "*****" ⇒ "**(**" *rexp rexp* "**)**" "*****"
⇒ "**(**" ***letter*** *rexp* "**)**" "*****" ⇒ "**(**" ***letter*** *rexp* "**|**" *rexp* "**)**" "*****"
⇒ "**(**" ***letter letter*** "**|**" *rexp* "**)**" "*****"
⇒ "**(**" ***letter letter*** "**|**" ***letter*** "**)**" "*****"

(c) *rexp* → *rexp* "**|**" *rterm* | *rterm*
rterm → *rterm rfactor* | *rfactor*
rfactor → *rfactor* "*****" | "**(**" *rexp* "**)**" | ***letter***

(d) The grammar of (c) gives the concatenation and choice binary operators *left* associativity. However, this choice is essentially arbitrary, since these are both *associative* operators, and giving them right associativity would not change the semantics of any expression.

Exercise 3.6

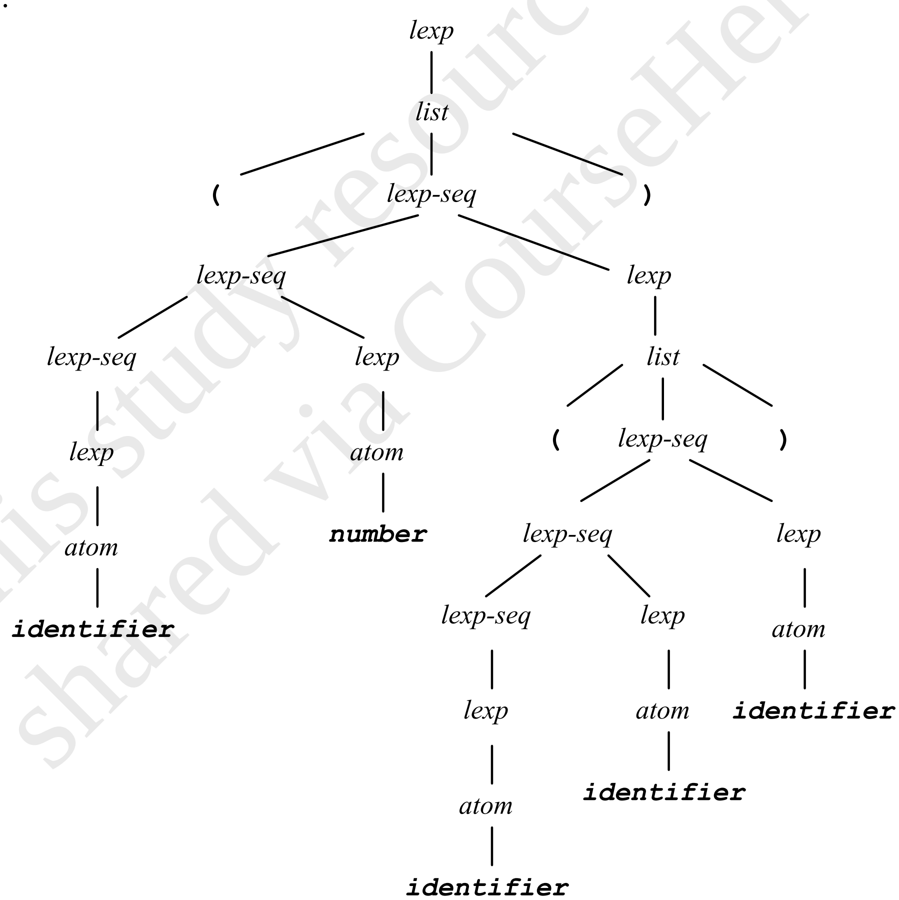
(a) Leftmost derivation:

lexp ⇒ *list* ⇒ **(** *lexp-seq* **)** ⇒ **(** *lexp-seq lexp* **)** ⇒ **(** *lexp-seq lexp lexp* **)**
⇒ **(** *lexp lexp lexp* **)** ⇒ **(** *atom lexp lexp* **)** ⇒ **(** ***identifier*** *lexp lexp* **)**
⇒ **(** ***identifier*** *atom lexp* **)** ⇒ **(** ***identifier number*** *lexp* **)**
⇒ **(** ***identifier number*** *list* **)** ⇒ **(** ***identifier number*** **(** *lexp-seq* **))**
⇒ **(** ***identifier number*** **(** *lexp-seq lexp* **))**
⇒ **(** ***identifier number*** **(** *lexp-seq lexp lexp* **))**
⇒ **(** ***identifier number*** **(** *lexp lexp lexp* **))**
⇒ **(** ***identifier number*** **(** *atom lexp lexp* **))**
⇒ **(** ***identifier number*** **(** ***identifier*** *lexp lexp* **))**
⇒ **(** ***identifier number*** **(** ***identifier*** *atom lexp* **))**
⇒ **(** ***identifier number*** **(** ***identifier identifier*** *lexp* **))**
⇒ **(** ***identifier number*** **(** ***identifier identifier*** *atom* **))**
⇒ **(** ***identifier number*** **(** ***identifier identifier identifier*** **))**

Rightmost derivation:

lexp ⇒ *list* ⇒ **(** *lexp-seq* **)** ⇒ **(** *lexp-seq lexp* **)** ⇒ **(** *lexp-seq list* **)**
⇒ **(** *lexp-seq* **(** *lexp-seq* **))** ⇒ **(** *lexp-seq* **(** *lexp-seq lexp* **))**
⇒ **(** *lexp-seq* **(** *lexp-seq atom* **))** ⇒ **(** *lexp-seq* **(** *lexp-seq* **identifier))**
⇒ **(** *lexp-seq* **(** *lexp-seq lexp* **identifier))**
⇒ **(** *lexp-seq* **(** *lexp-seq atom* **identifier))**
⇒ **(** *lexp-seq* **(** *lexp-seq* **identifier identifier))**
⇒ **(** *lexp-seq* **(** *lexp* **identifier identifier))**
⇒ **(** *lexp-seq* **(** *atom* **identifier identifier))**
⇒ **(** *lexp-seq* **(identifier identifier identifier))**
⇒ **(** *lexp-seq lexp* **(identifier identifier identifier))**
⇒ **(** *lexp-seq atom* **(identifier identifier identifier))**
⇒ **(** *lexp-seq* **number (identifier identifier identifier))**
⇒ **(** *lexp* **number (identifier identifier identifier))**
⇒ **(** *atom* **number (identifier identifier identifier))**
⇒ **(identifier number (identifier identifier identifier))**

(b) Parse tree:



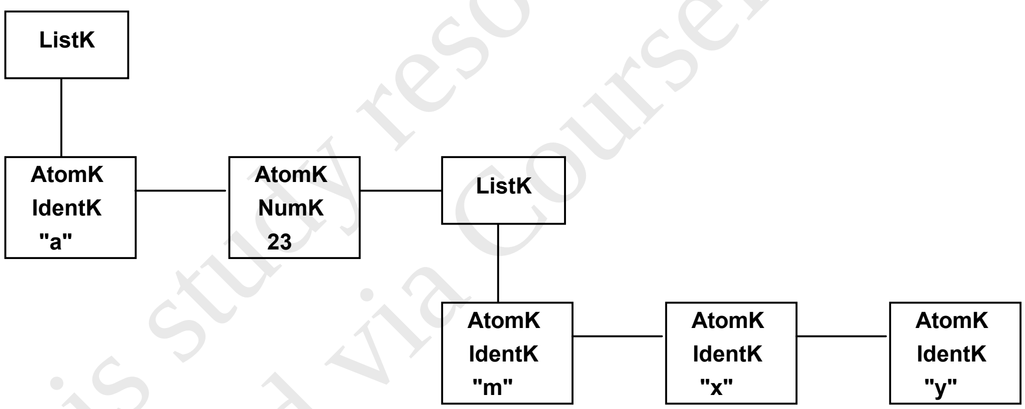
Exercise 3.7

(a) One possibility that uses sibling lists for lists of expressions is as follows:

```
typedef enum {AtomK,ListK} ExpKind;
typedef enum {NumK,IdentK} AtomKind;
typedef struct streenode
      { ExpKind ekind;
        AtomKind akind;
        struct streenode *explist,*sibling;
        int val;
        char* name;
     } STreeNode;
typedef STreeNode *SyntaxTree;
```

In this definition, the **akind** field is only used for atoms, the **val** field only for numbers, and the **name** field only for identifiers. Also, the **explist** pointer points to the sibling list of expressions in the case of a list only (and is null otherwise), while the **sibling** pointer is non-null only for an expression that is not last in a list of expressions. (Part (b) should make these usages clear.)

(b) In the following syntax tree we use rectangles for all nodes, and list the applicable attributes in order of their definition in part (a). Explist pointers are drawn downward, while sibling pointers are drawn to the right.

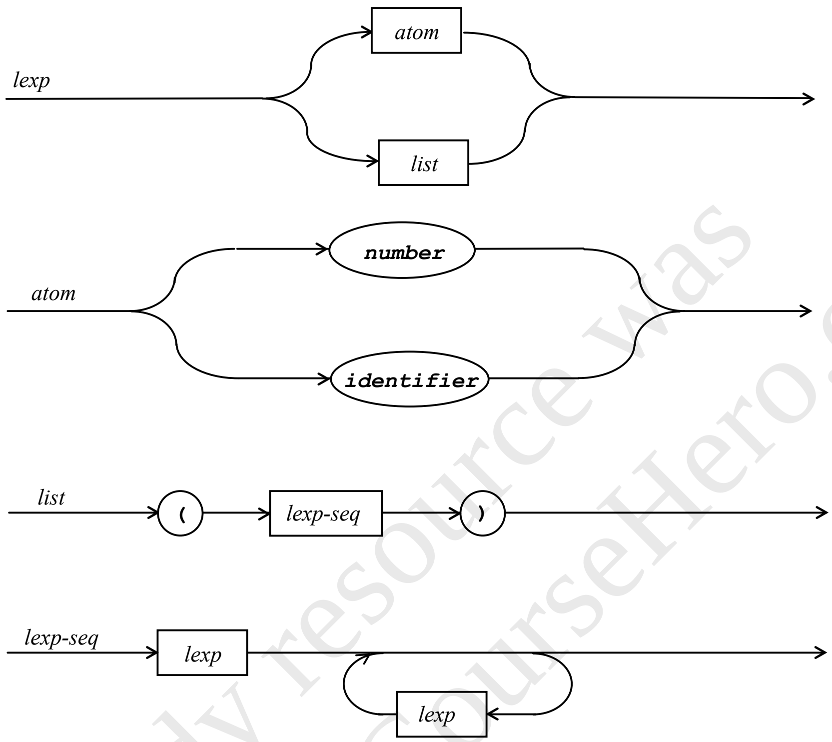


Exercise 3.10

(a) The only grammar rule that changes in EBNF is the last one, which changes to

lexp-seq → *lexp* { *lexp* }

(b)



Exercise 3.12

(a)
exp → *exp addop term* | – *term* | *term*
addop → **+** | **-**
term → *term mulop factor* | *factor*
mulop → *****
factor → **(** *exp* **)** | ***number***

(b)
exp → *exp addop term* | *term*
addop → **+** | **-**
term → *term mulop unary* | *unary*
mulop → *****
unary → – *factor* | *factor*
factor → **(** *exp* **)** | ***number***

(c)
exp → *exp addop term* | *term*
addop → **+** | **-**

$term \rightarrow term\ mulop\ factor \mid factor$
$mulop \rightarrow$ **`*`**
$factor \rightarrow$ **`(`** exp **`)`** | ***number*** | $-$ $factor$

Exercise 3.16

The code on page 111 actually has a small error in that **`OpKind`** is declared twice. We fix this as well as add a union to that code, as follows:

```
typedef enum {Plus,Minus,Times} OpKind;
typedef enum {OpK,ConstK} ExpKind;
typedef struct streenode
     { ExpKind kind;
       union
       { OpKind op;
         int val;
       } attrib;
       struct streenode *lchild,*rchild;
     } STreeNode;
typedef STreeNode *SyntaxTree;
```

Note that we have not put the **`lchild`**/**`rchild`** fields into the union, even though they are only used in the **`OpK`** case (just as the **`op`** field is). There are two reasons for this. First, doing so would require a new **`struct`** declaration within the union (in order to associate the child fields with the **`op`** field). Second, this may not save any space, since the ANSI C standard specifies that enumeration values are to be treated as integers, so a special effort by the compiler is necessary to use less space than that for integers.

Exercise 3.20

(a) **`(a|b)*`** (all strings of *a*'s and *b*'s)

(b) $A \rightarrow B\ C$
$B \rightarrow B\ D \mid \varepsilon$
$D \rightarrow a \mid c \mid ba \mid bc$
$C \rightarrow b \mid \varepsilon$

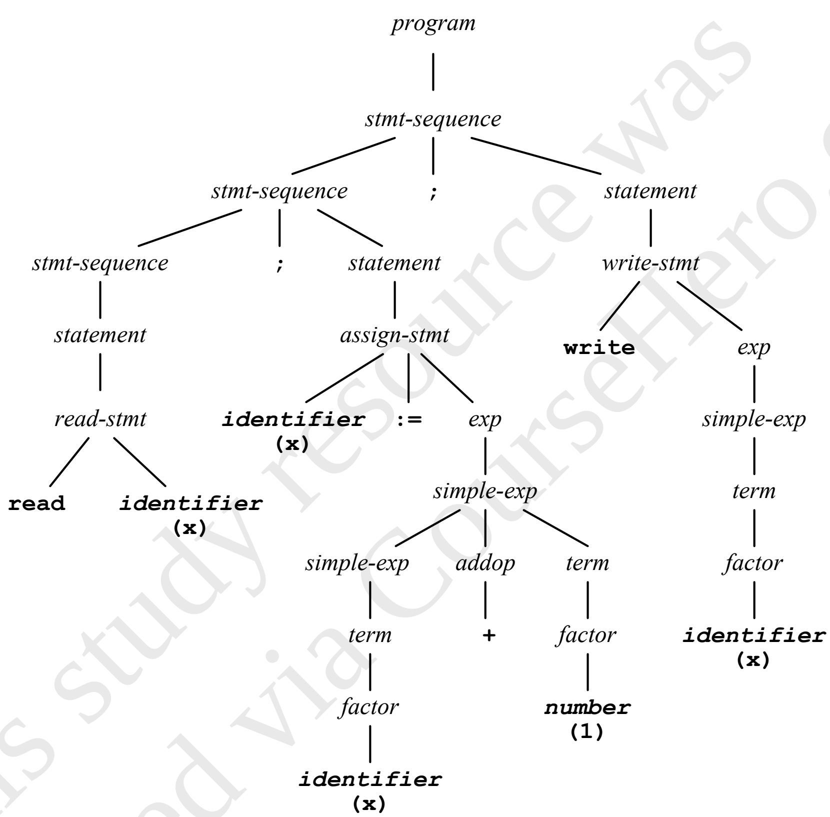
Exercise 3.22

(a) This exercise was not precisely enough stated; in fact, the derivation $A \Rightarrow^* A$ that establishes a grammar as cyclic must be *nontrivial*, that is, it must have at least one derivation step. In addition, the nonterminal A must appear as part of a derivation $S \Rightarrow^* \alpha\ A\ \beta \Rightarrow^* w$, where S is the start symbol and w is a string of terminals (that is, it must not be *useless* in the parlance of computation theory -- see Exercise 4.16, page 191). Given these additional facts, it is easy to see that such a grammar must be ambiguous, since the derivation $A \Rightarrow^* A$ can be inserted an arbitrary number of times in the derivation of w, and this must result in distinct parse trees.

(b) Such a situation is virtually guaranteed not to occur in the grammar of a programming language, since the resulting ambiguity serves absolutely no purpose, and the language generated by eliminating such cycles remains exactly the same. (Furthermore, any parser built using such a grammar must either ignore such cycles or always go into an infinite loop when the symbol A is reached in building a derivation, as we will see in the next chapters.)

Exercise 3.24

(a)



(b)