Compiler Construction: Principles and Practice
by Kenneth C. Louden

Chapter 6 Exercise Answers

Exercise 6.2

GRAMMAR RULE	SEMANTIC RULES
$dnum \rightarrow num_1 \,.\, num_2$	$dnum.val =$ $num_1.val + num_2.val / 10^{num_2.count}$
$num_1 \rightarrow num_2\ digit$	$num_1.val = num_2.val * 10 + digit.val$ $num_1.count = num_2.count + 1$
$num \rightarrow digit$	$num.val = digit.val$ $num.count = 1$
$digit \rightarrow$ `0`	$digit.val = 0$
$digit \rightarrow$ `1`	$digit.val = 1$
$digit \rightarrow$ `2`	$digit.val = 2$
$digit \rightarrow$ `3`	$digit.val = 3$
$digit \rightarrow$ `4`	$digit.val = 4$
$digit \rightarrow$ `5`	$digit.val = 5$
$digit \rightarrow$ `6`	$digit.val = 6$
$digit \rightarrow$ `7`	$digit.val = 7$
$digit \rightarrow$ `8`	$digit.val = 8$
$digit \rightarrow$ `9`	$digit.val = 9$

Exercise 6.4

GRAMMAR RULE	SEMANTIC RULES
$exp \rightarrow term\ exp'$	$exp'.inval = term.val$ $exp.val = exp'.outval$
$exp_1' \rightarrow$ `+` $term\ exp_2'$	$exp_2'.inval = exp_1'.inval + term.val$ $exp_1'.outval = exp_2'.outval$
$exp_1' \rightarrow$ `-` $term\ exp_2'$	$exp_2'.inval = exp_1'.inval - term.val$ $exp_1'.outval = exp_2'.outval$
$exp' \rightarrow \varepsilon$	$exp'.outval = exp'.inval$
$term \rightarrow factor\ term'$	$term'.inval = factor.val$ $term.val = term'.outval$
$term_1' \rightarrow$ `*` $factor\ term_2'$	$term_2'.inval = term_1'.inval * factor.val$ $term_1'.outval = term_2'.outval$
$term' \rightarrow \varepsilon$	$term'.outval = term'.inval$
$factor \rightarrow$ `(` exp `)`	$factor.val = exp.val$
$factor \rightarrow$ **`number`**	$factor.val =$ **`number`** $.val$

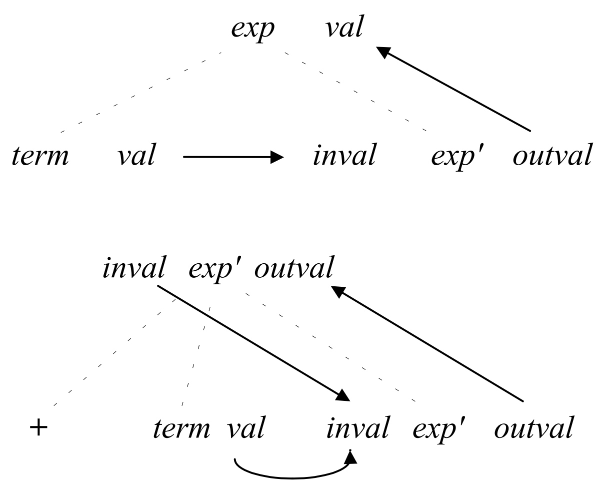
<u>Exercise 6.7</u>

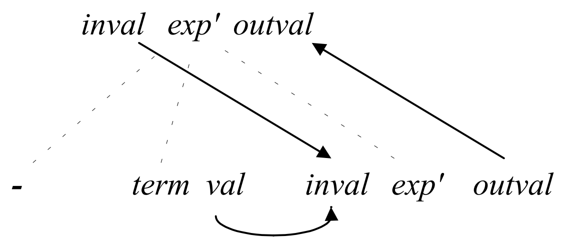
GRAMMAR RULE	SEMANTIC RULES
decl → *var-list* : *type*	*var-list.dtype* = *type.dtype*
$\textit{var-list}_1$ → $\textit{var-list}_2$ **`, id`**	**`id`** *.dtype* = $\textit{var-list}_1$*.dtype* $\textit{var-list}_2$*.dtype* = $\textit{var-list}_1$*.dtype*
var-list → **`id`**	**`id`** *.dtype* = *var-list.dtype*
type → **`integer`**	*type.dtype* = *integer*
type → **`real`**	*type.dtype* = *real*

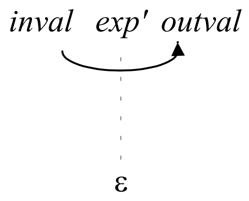
<u>Exercise 6.8</u>

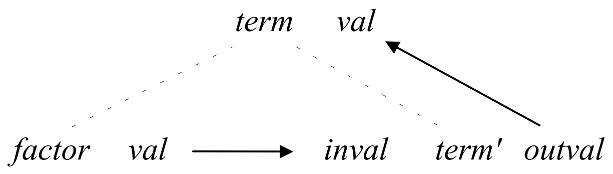
GRAMMAR RULE	SEMANTIC RULES
decl → **`id`** *rest*	**`id`** *.dtype* = *rest.dtype*
$\textit{rest}_1$ → **`, id`** $\textit{rest}_2$	**`id`** *.dtype* = $\textit{rest}_2$*.dtype* $\textit{rest}_1$*.dtype* = $\textit{rest}_2$*.dtype*
rest → : *type*	*rest.dtype* = *type.dtype*
type → **`integer`**	*type.dtype* = *integer*
type → **`real`**	*type.dtype* = *real*

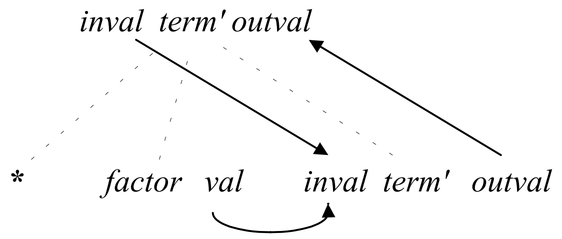
<u>Exercise 6.11</u>

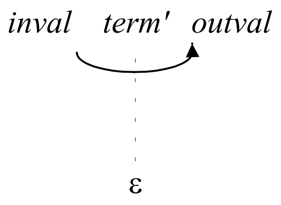


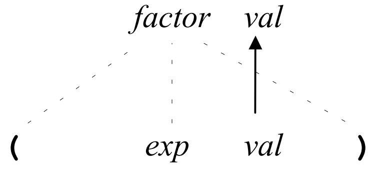

inval exp' outval
-
term val
inval exp' outval

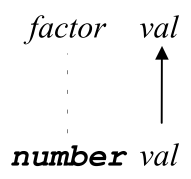

inval exp' outval
ε

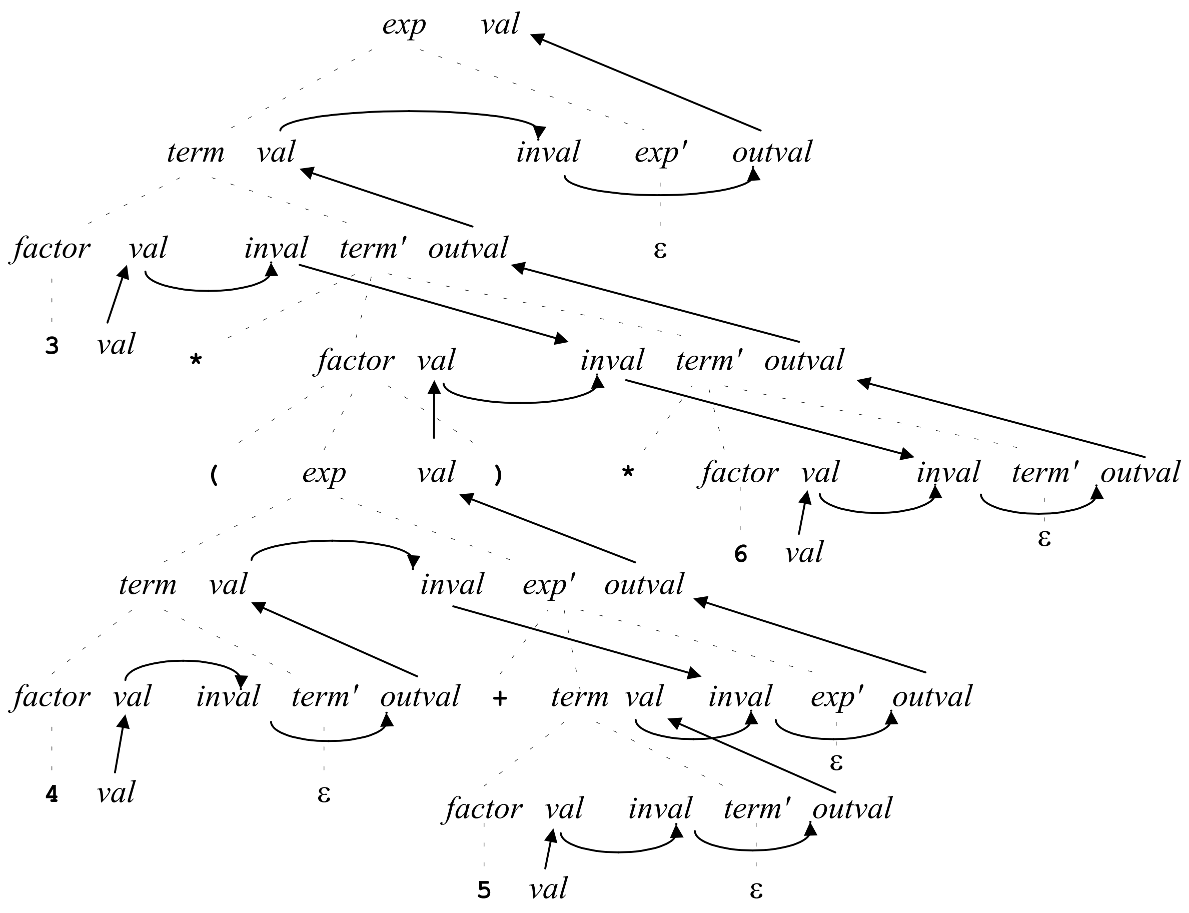

term val
factor val
inval term' outval


inval term' outval
*
factor val
inval term' outval


inval term' outval
ε


factor val
(
exp val
)





Exercise 6.14

Consider the two input strings **`int x`** and **`int x,y`**. Using the conventions of Section 6.2.5 (pages 291-293), and using *T* for *type*, *D* for *decl*, and *V* for *var-list*, we have the following actions during an LR parse of each of these input strings:

	Parsing stack	Input	Parsing Action	Value stack	Semantic Action
1	$	**`int x`** $	shift	$	
2	$ **`int`**	**`x`** $	reduce *T* → **`int`**	$ **`int`**	*T.dtype* = *integer*
3	$ *T*	**`x`** $	shift	$ *integer*	
4	$ *T* ***`id`***	$	reduce *V* → ***`id`***	$ *integer* ***`id`***	***`id`****.dtype* = *V.dtype*

	Parsing stack	Input	Parsing Action	Value stack	Semantic Action
1	\$	`int x,y` \$	shift	\$	
2	\$ `int`	`x,y` \$	reduce $T \to$ `int`	\$ `int`	*T.dtype* = *integer*
3	\$ *T*	`x,y` \$	shift	\$ *integer*	
4	\$ *T* `id`	`,y` \$	shift	\$ *integer* `id`	
5	\$ *T* `id ,`	`y` \$	shift	\$ *integer* `id ,`	
6	\$ *T* `id , id`	\$	reduce $V \to$ `id`	\$ *integer* `id , id`	`id`*.dtype* = *V.dtype*

In the first parse, when the reduction in the last line occcurs, the *dtype* value is found in the value stack at position top–1, while in the second parse, when the reduction in the last line occurs the *dtype* value is found at position top-3.

Exercise 6.18

In the following attribute grammar, we designate the new value attribute *val*. We also change the *lookup* function from Table 6.9, page 311 (which returns the nesting level of a name) to *lookupLevel*. We also add a *lookupVal* function, since the symbol table must now store the numeric value of a name, as well as the nesting level. Also, the *insert* procedure must have the value as an additional parameter. Finally, we include an **error** value as a potential value and discard the *err* attribute and the *isin* function (the *lookupVal* function returns **error** if the name is not found in the symbol table).

GRAMMAR RULE	SEMANTIC RULES
$S \to exp$	*exp.symtab* = *emptytable* *exp.nestlevel* = 0
$exp_1 \to exp_2$ `+` exp_3	exp_2*.symtab* = exp_1*.symtab* exp_3*.symtab* = exp_1*.symtab* exp_2*.nestlevel* = exp_1*.nestlevel* exp_3*.nestlevel* = exp_1*.nestlevel* exp_1*.val* = **if** (exp_2*.val* = **error**) **or** (exp_3*.val* = **error**) **then error** **else** exp_2*.val* + exp_3*.val*
$exp_1 \to$ `(` exp_2 `)`	exp_2*.symtab* = exp_1*.symtab* exp_2*.nestlevel* = exp_1*.nestlevel* exp_1*.val* = exp_2*.val*
$exp \to$ `id`	*exp.val* = *lookupVal*(*exp.symtab*,`id`*.name*)

exp → **`num`**	*exp.val* = **`num`** *.val*
exp_1 → `let` *dec-list* `in` exp_2	*dec-list.intab* = exp_1 *.symtab* *dec-list.nestlevel* = exp_1 *.nestlevel* + 1 exp_2 *.symtab* = *dec-list.outtab* exp_2 *.nestlevel* = *dec-list.nestlevel* exp_1 *.val* = **if** (*decl-list.outtab* = *errtab*) **then error** **else** exp_2 *.val*
$dec\text{-}list_1$ → $dec\text{-}list_2$ `,` *decl*	$dec\text{-}list_2$ *.intab* = $dec\text{-}list_1$ *.intab* $dec\text{-}list_2$ *.nestlevel* = $dec\text{-}list_1$ *.nestlevel* *decl.intab* = $dec\text{-}list_2$ *.outtab* *decl.nestlevel* = $dec\text{-}list_2$ *.nestlevel* $dec\text{-}list_1$ *.outtab* = *decl.outtab*
dec-list → *decl*	*decl.intab* = *dec-list.intab* *decl.nestlevel* = *dec-list.nestlevel* *dec-list.outtab* = *decl.outtab*

decl → **`id`** `=` *exp*	*exp.symtab* = *decl.intab*
	exp.nestlevel = *decl.nestlevel*
	decl.outtab =
	  if (*decl.intab* = *errtab*)
	  then *errtab*
	  else if
	    lookupLevel(*decl.intab*,**`id`** *.name*) = *decl.nestlevel*)
	  then *errtab*
	  else
	    insert(*decl.intab*,**`id`** *.name*, *decl.nestlevel*, *exp.val*)

<u>Exercise 6.22</u>

(a) If we want the parser to distinguish between cast expresssions and regular arithmetic expressions, then when an identifier such as A is reached, the parser must be able to lookup A in the symbol table to determine whether it was declared as a type name or a variable name. It can do this if, during the parse, the parser also enters each name into the symbol table as it is declared, with an indication of whether it is declared as a type or variable (i.e. in a typedef or not). Note that further type checking/semantic analysis is not required. Then the parser would construct a syntax tree node representing a type expression for the expression (A), rather than an arithmetic expression. Based on this tree, the parser would then determine whether the following minus sign is unary or binary.

(b) If we want the scanner to disambiguate the use of A, then the scanner must be able to look A up in the symbol table. This also requires that the *parser* has been inserting the names of variables and types as they are declared into the symbol table, since the scanner does not see enough of the input at one time to determine whether an identifier is declared in a typedef or not. Once the scanner has looked up A in the table and found it to be a type name, it can return a *different token* to indicate the status of A: instead of an ID, it can generate a TypeID token. The parser then doesn't need to perform a lookup, but based on the token returned by the scanner, it can generate the appropriate syntax tree.