Jeongbin Son
CS403 Assignment 1
1/27/2024

Question 1)

3. Rewrite the BNF of Example 3.4 to give + precedence over * and force + to be right associative.

EXAMPLE 3.4 An Unambiguous Grammar for Expressions

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

<assign> -> <id> = <expr>
<id> -> A | B | C
<expr> -> <term> + <expr> | <term>
<term> -> <factor> * <term> | <factor>
<factor> -> (<expr>) | <id>

Question 2)

8. Prove that the following grammar is ambiguous:

<S> → <A>

<A> → <A> + <A> | <id>

<id> → a | b | c

String “a+b+c”
Derivation 1:

<S>
<A>
<A>+<A>
<id>+<A>
a+<A>
a+<id>
a+b

Derivation 2:

<S>
<A>
<A>+<A>
<id>+<A>
b+<A>
b+<id>
b+c

“a+b+c” can be derived in different ways with this ambiguous grammar.

Question 3)

20. Write an attribute grammar whose base BNF is that of Example 3.2 and whose type rules are the same as for the assignment statement example of Section 3.4.5.

EXAMPLE 3.2 A Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

EXAMPLE 3.6 An Attribute Grammar for Simple Assignment Statements

1. Syntax rule: <assign> → <var> = <expr>
   Semantic rule: <expr>.expected_type ← <var>.actual_type
2. Syntax rule: <expr> → <var>[2] + <var>[3]
   Semantic rule: <expr>.actual_type ←
   if (<var>[2].actual_type = int) and
   (<var>[3].actual_type = int)
   then int
   else real
   end if
   Predicate: <expr>.actual_type == <expr>.expected_type
3. Syntax rule: <expr> → <var>
   Semantic rule: <expr>.actual_type ← <var>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type
4. Syntax rule: <var> → A | B | C
   Semantic rule: <var>.actual_type ← look-up (<var>.string)

The look-up function looks up a given variable name in the symbol table and returns the variable's type.

1. Syntax rule: <assign> -> <id> = <expr>
   Semantic rule: <expr>. expected_type <- <id>.actual_type

2. Syntax rule: <id> -> A | B | C
   Semantic rule: <id>.actual_type <- look-up(<id>.string)

3. Syntax rule: <expr> -> <id> + <expr>

Semantic rule:

<expr>.actual_type <-
If (<id>.actual_type = int) and (<expr>.actual_type = int)
then int
else real
end if

Predicate: <expr>.actual_type == <expr>.expected_type

4. Syntax rule: <expr> -> <id> * <expr>
   Semantic rule:

<expr>.actual_type <-
If (<id>.actual_type = int) and (<expr>.actual_type = int)
then int
else real
end if

Predicate: <expr>.actual_type == <expr>.expected_type

5. Syntax rule: <expr> -> (<expr>)
   Semantic rule: <expr>.actual_type <- <expr>.actual_type
   Predicate: <expr>.actual_type == <expr>.expected_type

6. Syntax rule: <expr> -> <id>
   Semantic rule: <expr>.actual_type <- <id>.actual_type
   Predicate: <expr>.actual_type == <id>.expected_type

Question 4)

8. Show a complete parse, including the parse stack contents, input string, and action for the string (id + id) * id, using the grammar and parse table in Section 4.5.3.

Figure 4.5

The LR parsing table for an arithmetic expression grammar

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Answer:

Stack	Input	Action

0	(id+id)*id $	Shift 4
0(4	id+id)*id $	Shift 5
0(4id5	+id)*id $	Reduce 6, Goto (4, F)
0(4F3	+id)*id $	Reduce 4, Goto (4, T)
0(4T2	+id)*id $	Reduce 2, Goto (4, E)
0(4E8	+id)*id $	Shift 6
0(4E8+6	id)*id $	Shift 5
0(4E8+6id5	)*id $	Reduce 6, Goto (6, F)
0(4E8+6F3	)*id $	Reduce 4, Goto (6, T)
0(4E8+6T9	)*id $	Reduce 1, Goto (4, E)
0(4E8	)*id $	Shift 11
0(4E8)11	*id $	Reduce 5, Goto (0, F)
0F3	*id $	Reduce 4, Goto (0, T)
0T2	*id $	Shift 7
0T2*7	id $	Shift 5
0T2*7id5	$	Reduce 6, Goto (7, F)
0T2*7F10	$	Reduce 3, Goto (0, T)
0T2	$	Reduce 2, Goto (0, E)
0E1	$	Accept