**Chapter 3**

*Problem Set:*

2a. <class\_head> → {<modifier>} **class** <id> [**extends** class\_name]

[**implements** <interface\_name> {, <interface\_name>}]

<modifier> → **public** | **abstract** | **final**

2c. <switch\_stmt> → **switch** ( <expr> ) {**case** <literal> : <stmt\_list>

{**case** <literal> : <stmt\_list> } [**default** : <stmt\_list>] }

3. <assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> \* <term>

| <term>

<term> → <factor> + <term>

| <factor>

<factor> → ( <expr> )

| <id>

6.

(a) <assign> => <id> = <expr>

=> A = <expr>

=> A = <id> \* <expr>

=> A = A \* <expr>

=> A = A \* ( <expr> )

=> A = A \* ( <id> + <expr> )

=> A = A \* ( B + <expr> )

=> A = A \* ( B + ( <expr> ) )

=> A = A \* ( B + ( <id> \* <expr> ) )

=> A = A \* ( B + ( C \* <expr> ) )

=> A = A \* ( B + ( C \* <id> ) )

=> A = A \* ( B + ( C \* A ) )



7.

(a) <assign> => <id> = <expr>

=> A = <expr>

=> A = <term>

=> A = <factor> \* <term>

=> A = ( <expr> ) \* <term>

=> A = ( <expr> + <term> ) \* <term>

=> A = ( <term> + <term> ) \* <term>

=> A = ( <factor> + <term> ) \* <term>

=> A = ( <id> + <term> ) \* <term>

=> A = ( A + <term> ) \* <term>

=> A = ( A + <factor> ) \* <term>

=> A = ( A + <id> ) \* <term>

=> A = ( A + B ) \* <term>

=> A = ( A + B ) \* <factor>

=> A = ( A + B ) \* <id>

=> A = ( A + B ) \* C



8. The following two distinct parse tree for the same string prove that the grammar is ambiguous.



9. Assume that the unary operators can precede any operand. Replace the rule

<factor> → <id>

with

<factor> → + <id>

| - <id>

10. One or more a's followed by one or more b's followed by one or more c's.

13. S → a S b | a b

14.



16. <assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> (+ | -) <expr>

| (<expr>)

| <id>

18. The value of an intrinsic attribute is supplied from outside the attribute evaluation process, usually from the lexical analyzer. A value of a synthesized attribute is computed by an attribute evaluation function.

19. Replace the second semantic rule with:

<var>[2].env ← <expr>.env

<var>[3].env ← <expr>.env

<expr>.actual\_type ← <var>[2].actual\_type

predicate: <var>[2].actual\_type = <var>[3].actual\_type

21.

(a) (Java **do-while**) We assume that the logic expression is a single relational expression.

loop: (do body)

if <relational\_expression> goto out

goto loop

out: ...

(b) (Ada **for**) **for** I **in** first .. last **loop**

I = first

loop: if I < last goto out

...

I = I + 1

goto loop

out: ...

(c) (Fortran Do)

K = start

loop: if K > end goto out

...

K = K + step

goto loop

out: ...

(e) (C **for**) **for** (expr1; expr2; expr3) ...

evaluate(expr1)

loop: control = evaluate(expr2)

if control == 0 goto out

...

evaluate(expr3)

goto loop

out: ...

22a. Mpf(for var in init\_expr .. final\_expr loop L end loop, s) 

if VARMAP(i, s) = **undef** for var or some i in init\_expr or final\_expr

then **error**

else if Me(init\_expr, s) > Me(final\_expr, s)

then s

else Ml(while init\_expr - 1 <= final\_expr do L, Ma(var := init\_expr + 1, s))

22b. Mr(repeat L until B) 

if Mb(B, s) = **undef**

then **error**

else if Msl(L, s) = **error**

then **error**

else if Mb(B, s) = **true**

then Msl(L, s)

else Mr(repeat L until B), Msl(L, s))

22c. Mb(B, s)  if VARMAP(i, s) = **undef** for some i in B

then **error**

else B', where B' is the result of

evaluating B after setting each

variable i in B to VARMAP(i, s)

22d. Mcf(for (expr1; expr2; expr3) L, s) 

if VARMAP (i, s) = **undef** for some i in expr1, expr2, expr3, or L

then **error**

else if Me (expr2, Me (expr1, s)) = 0

then s

else Mhelp (expr2, expr3, L, s)

Mhelp (expr2, expr3, L, s) 

if VARMAP (i, s) = **undef** for some i in expr2, expr3, or L

then **error**

else

if Msl (L, s) = **error**

then s

else Mhelp (expr2, expr3, L, Msl (L, Me (expr3, s))

23.

(a) a = 2 \* (b - 1) - 1 {a > 0}

2 \* (b - 1) - 1 > 0

2 \* b - 2 - 1 > 0

2 \* b > 3

b > 3 / 2

(b) b = (c + 10) / 3 {b > 6}

(c + 10) / 3 > 6

c + 10 > 18

c > 8

(c) a = a + 2 \* b - 1 {a > 1}

a + 2 \* b - 1 > 1

2 \* b > 2 - a

b > 1 - a / 2

(d) x = 2 \* y + x - 1 {x > 11}

2 \* y + x - 1 > 11

2 \* y + x > 12

24.

(a) a = 2 \* b + 1

b = a - 3 {b < 0}

a - 3 < 0

a < 3

Now, we have:

a = 2 \* b + 1 {a < 3}

2 \* b + 1 < 3

2 \* b + 1 < 3

2 \* b < 2

b < 1

(b) a = 3 \* (2 \* b + a);

b = 2 \* a - 1 {b > 5}

2 \* a - 1 > 5

2 \* a > 6

a > 3

Now we have:

a = 3 \* (2 \* b + a) {a > 3}

3 \* (2 \* b + a) > 3

6 \* b + 3 \* a > 3

2 \* b + a > 1

n > (1 - a) / 2

**Chapter 4**

*Problem Set:*

1.

(a) FIRST(aB) = {a}, FIRST(b) = {b}, FIRST(cBB) = {c}, Passes the test

(b) FIRST(aB) = {a}, FIRST(bA) = {b}, FIRST(aBb) = {a}, Fails the test

(c) FIRST(aaA) = {a}, FIRST(b) = {b}, FIRST(caB) = {c}, Passes the test

3. a + b \* c

Call lex /\* returns a \*/

Enter <expr>

Enter <term>

Enter <factor>

Call lex /\* returns + \*/

Exit <factor>

Exit <term>

Call lex /\* returns b \*/

Enter <term>

Enter <factor>

Call lex /\* returns \* \*/

Exit <factor>

Call lex /\* returns c \*/

Enter <factor>

Call lex /\* returns end-of-input \*/

Exit <factor>

Exit <term>

Exit <expr>

5.

(a) aaAbb

S

a A b

a A B

b

Phrases: aaAbb, aAb, b

Simple phrases: b

Handle: b

(b) bBab S

b B A

a b

Phrases: bBab, ab

Simple phrases: ab

Handle: ab

7. *Stack Input Action*

0 id \* (id + id) $ Shift 5

0id5 \* (id + id) $ Reduce 6 (Use GOTO[0, F])

0F3 \* (id + id) $ Reduce 4 (Use GOTO[0, T])

0T2 \* (id + id) $ Reduce 2 (Use GOTO[0, E])

0T2\*7 (id + id) $ Shift 7

0T2\*7(4 id + id ) $ Shift 4

0T2\*7(4id5 + id ) $ Shift 5

0T2\*7(4F3 + id ) $ Reduce 6 (Use GOTO[4, F])

0T2\*7(4T2 + id ) $ Reduce 4 (Use GOTO[4, T])

0T2\*7(4E8 + id ) $ Reduce 2 (Use GOTO[4, E])

0T2\*7(4E8+6 id ) $ Shift 6

0T2\*7(4E8+6id5 ) $ Shift 5

0T2\*7(4E8+6F3 ) $ Reduce 6 (Use GOTO[6, F])

0T2\*7(4E8+6T9 ) $ Reduce 4 (Use GOTO[6, T])

0T2\*7(4E8 ) $ Reduce 1 (Use GOTO[4, E])

0T2\*7(4E8)11 $ Shift 11

0T2\*7F10 $ Reduce 5 (Use GOTO[7, F])

0T2 $ Reduce 5 (Use GOTO[0, T])

0E1 $ Reduce 2 (Use GOTO[0, E])

--ACCEPT--

*Programming Exercises:*

1. Every arc in this graph is assumed to have addChar attached. Assume we get here only if charClass is SLASH.

other

/ \* \* /

start slash com end return COMMENT

other

return SLASH\_CODE

3. int getComment() {

getChar();

/\* The slash state \*/

if (charClass != AST)

return SLASH\_CODE;

else {

/\* The com state-end state loop \*/

do {

getChar();

/\* The com state loop \*/

while (charClass != AST)

getChar();

} while (charClass != SLASH);

return COMMENT;

}