**Bold: terminals**

*Non-bold: non-terminals*

TYPE CHECKING - (#) denotes position, no (#) means it’s at the end/obvious, id.type means getType(id.lex)

SCOPE CHECKING

MEMORY CHECKING

PROCEDURE CHECKING

RETURN

INHERITED

1.1 *program* --> **program id** (1)(2) **(**idlst**);** VOID VOID

*program’*

(1) {offset := 0}

(2) checkAddGreenNode(id.lex, PGNAME) set GN pointer

1.2.1 *program’* --> *subdeclarations* VOID VOID

*compound\_statement*

**.**

1.2.2 *program’* --> *compound\_statement*

**.**

1.2.3 *program’* --> *declarations*

*program’’*

1.3.1 *program’’ --> subdeclarations* VOID VOID

*compound\_statement*

**.**

1.3.2 *program’’ --> compound\_statement*

**.**

2.1 *idlst* --> **id** (1)*idlst’* VOID VOID

(1) checkAddBlueNode(id.lex, PGPARAM)

2.2.1 *idlst’* --> **, id** (1)*idlst’*

(1) checkAddBlueNode(id.lex, PGPARAM) VOID VOID

2.2.2 idlst’ --> ϵ

3.1 *declarations* --> **var** **id:** *type*(1) (2)**;** *declarations’* VOID VOID

(1) checkAddBlueNode(id.lex, type.type)

(2) offset += type.width

3.2.1 *declarations’* --> **var** **id:** *type* (1) (2)**;** *declarations’* VOID VOID

(1) checkAddBlueNode(id.lex, type.type)

(2) offset += type.width

3.2.2 *declarations’* --> ϵ

4.1 *type* --> *standard\_type* VOID TYPE/WIDTH

{type.type := standard\_type.type}

{width = standard\_type.width}

4.2 *type* --> **array** **[num** **..** **num]** **of** *standard\_type*

{type.type := “a” + standard\_type.type}

Check if num and num are ints and check if the second one is bigger than the first

{width = standard\_type.width \* (num2-num1+1)}

5.1 *standard*\_*type* --> **integer** VOID TYPE/WIDTH

{standard\_type.type := int}

{width := 4}

5.2 *standard\_type*  --> **real**

{standard\_type.type := real}

{width := 8}

6.1 *subdeclarations* --> *subdeclaration* **;** (1)*subdeclarations’* VOID VOID

(1) pop stack

6.2.1 *subdeclarations’* --> *subdeclaration* **;** (1) *subdeclarations’* VOID VOID

(1) pop stack

6.2.2 *subdeclarations’* --> ϵ

7.1 *subdeclaration* --> *subprogram\_head* VOID VOID

*subdeclaration’*

7.2.1 *subdeclaration’* --> *subdeclarations* VOID VOID

*compound\_statement*

7.2.2 *subdeclaration’* --> *compound\_statement*

7.2.3 *subdeclaration’* --> *declarations*

*subdeclaration’’*

7.3.1 *subdeclaration’’ --> subdeclarations* VOID VOID

*compound\_statement*

7.3.2 *subdeclaration’’ --> compound\_statement*

8.1 *subprogram\_head* --> **procedure** **id** (1) *subprogram\_head’* VOID VOID

(1) checkAddGreenNode(id.lex, PPNAME) set GN pointer

{offset := 0}

8.2.1 *subprogram\_head’* --> **;** VOID VOID

8.2.2 *subprogram\_head’* --> *arguments* **;**

9.1 *arguments* --> **(***parameter\_list***)**VOID VOID

10.1 *parameter\_list* --> **id** **:** *type* (1) *parameter\_list’* VOID VOID

(1) checkAddBlueNode(id.lex, PPARAM, type.type)

10.2.1 *parameter\_list’*  --> **;** ***id*** **:** *type* (1) *parameter\_list’* VOID VOID

(1) checkAddBlueNode(id.lex, PPARAM, type.type)

10.2.2 *parameter\_list’*  --> ϵ

11.1 *compound\_statement* --> **begin** *compound\_statement’* VOID VOID

11.2.1 *compound\_statement’* --> *optional\_statements*

**end**

11.2.2 *compound\_statement’* --> **end** VOID VOID

12.1 *optional\_statements* --> *statement\_list* VOID VOID

13.1 *statement\_list --> statement statement\_list’* VOID VOID

13.2.1 *statement\_list’ -->* **;** *statement statement\_list’* VOID VOID

13.2.2 *statement\_list’ -->* ϵ

14.1 *statement* --> *variable* **assignop** *expression* VOID VOID

{if variable.type == expression.type or either variable.type or expression.type are ERR: Do Nothing

Else: ERR\*}

14.2 *statement* --> *procedure\_statement*

14.3 *statement* --> *compound\_statement*

14.4 *statement* --> **while** *expression* **do** *statement*

{if expression != Bool: ERR\*}

14.5 *statement* --> **if** *expression* () **then** *statement* *statement’*

{if expression != Bool: ERR\*}

14.6.1 *statement’* --> **else** *statement* VOID VOID

14.6.2 *statement’* --> ϵ

15.1 *variable* --> **id** (1)*variable’* (2) VOID TYPE

(1){variable’.i := id.type}

(2){variable.type := variable’.type}

15.2.1 *variable’* --> **[***expression***]** TYPE TYPE

|  |  |  |
| --- | --- | --- |
| variable’.type | expression.type | variable’.i |
| INT | INT | AINT |
| REAL | INT | AREAL |
| ERR\* | Anything except ERR | Anything except ERR |
| ERR | Anything | ERR |
| ERR | ERR | Anything |

15.2.2 *variable’* --> ϵ

{variable’.type := variable’.i}

16.1 *procedure\_statement* --> **call** **id** *procedure\_statement’* VOID VOID

procedurePointer := checkProcedure(id.lex) (print SYM ERR if not)

(global or procedure\_statement’.i := procedurePointer)

16.2.1 *procedure\_statement’* --> **(***expression\_list***)** POINTER VOID

16.2.2 *procedure\_statement’* --> ϵ

Check if procedure had originally defined inputs

17.1 *expression\_list*  --> *expression expression\_list’* POINTER VOID

#If pointer is null do nothing

checkParam(expression.type)

advance parameter (expression\_list.i)

17.2.1 *expression\_list* ‘ --> **,** *expression expression\_list’* POINTER VOID

#If pointer is null do nothing

checkParam(expression.type)

advance parameter (expression\_list.i)

17.2.2 *expression\_list* ‘ --> ϵ

CheckNoMoreParams()

18.1 *expression* --> *simple\_expression* (1) *expression’* (2) VOID TYPE

(1) {expression’.i := simple\_expression’.type}

(2) {expression.type := expression’type}

18.2.1 *expression’* --> **relop** *simple\_expression* TYPE TYPE

|  |  |  |  |
| --- | --- | --- | --- |
| expression’.type (out) | expression’.i (in) | relop.op (in) | se.type(in) |
| BOOL | REAL/INT | all relop ops | REAL/INT |
| ERR\* | non equivalent | all relop ops | non equivalent |
| ERR | Anything | all relop ops | ERR |
| ERR | ERR | all relop ops | anything |

18.2.2 *expression’* --> ϵ

{expression’.type := expression’.i}

19.1 *simple\_expression* --> *term* (1) *simple\_expression’* (2)TYPE TYPE

(1) {simple\_expression’.i := term.type}

(2) {simple\_expression.type := simple\_expression’.type}

19.2 *simple\_expression* --> *sign term* (1) *simple\_expression’* (2)

(1) {simple\_expression’.i := term.type}

(2) {simple\_expression.type := simple\_expression’.type}

19.3.1 *simple\_expression’* --> **addop** *term* (1) *simple\_expression’* (2)TYPE TYPE

(1)

|  |  |  |  |
| --- | --- | --- | --- |
| se2’.i (out) | term.type (in) | addop.op (in) | se1’.i (in) |
| INT | INT | + - | INT |
| ERR\* | INT BOOL | + - | REAL BOOL |
| ERR\* | REAL BOOL | + - | INT BOOL |
| REAL | REAL | + - | REAL |
| ERR | ERR | + - or | Anything |
| ERR | Anything | + - or | ERR |
| BOOL | BOOL | or | BOOL |
| ERR\* | BOOL INT REAL | or | INT REAL |
| ERR\* | INT REAL | or | BOOL INT REAL |

(2) {simple\_expression’1.type := simple\_expression’2.type}

19.3.2 *simple\_expression’* --> ϵ

{simple\_expression’.type := simple\_expression.i}

20.1 *term* --> *factor* (1) *term’* (2) VOID TYPE

(1) {term’.i := factor.type}

(2) {term.type := term’.type}

20.2.1 *term’* --> **mulop** *factor* (1) *term’* (2) TYPE TYPE

(1)

|  |  |  |  |
| --- | --- | --- | --- |
| term2’.i (out) | factor.type (in) | mulop.op (in) | term1’.i (in) |
| INT | INT | \* / mod div | INT |
| ERR\* | REAL BOOL | \* / mod div | INT BOOL |
| ERR\* | INT BOOL | \* / mod div | REAL BOOL |
| REAL | REAL | \* / mod div | REAL |
| ERR | Anything | \* / mod div and | ERR |
| ERR | ERR | \* / mod div and | Anything |
| BOOL | BOOL | and | BOOL |
| ERR\* | BOOL | and | REAL INT |
| ERR\* | REAL INT | and | BOOL |

NOTE: If mulop.op is / must check for divide by zero

(2) {term.type := term’.type}

20.2.2 *term’* --> ϵ

{term’.type := term’.i}

21.1 *factor* --> **id** (1)*factor’* (2) VOID TYPE

(1){factor’.i := id.type}

(2){factor.type := factor’.type}

21.2 *factor* --> **num**

{factor.type := check num for either sreal, lreal or int}

21.3 *factor* --> **(** *expression* **)**

{factor.type := expression.type}

21.4 *factor* --> **not** *factor (1)(2)*

*(1) {if factor.type == BOOL: we good; else; SYM ERR}*

*(2) {factor1.type := factor2.type}*

21.5.1 *factor’* --> ***[****expression****]*** TYPE TYPE

|  |  |  |
| --- | --- | --- |
| factor’.type | expression.type | factor’.i |
| INT | INT | AINT |
| REAL | INT | AREAL |
| ERR\* | Anything except ERR | Anything except ERR |
| ERR | Anything | ERR |
| ERR | ERR | Anything |

21.5.2 *factor’* --> ϵ

{factor’.type := factor’.i}

22.1 *sign* --> **+** VOID VOID

22.2 *sign* --> **-**