TABLE OF CONTENTS

Contents ***Page***

GRAMMAR PRODUCTIONS AND SEMANTIC RULES

After parsing the token stream from the lexical analyzer, the programming constructs in the source program are associated with certain information relevant for the next part of compilation which is the semantic analysis. This is done by attaching attributes to grammar symbols. The values in the attributes are computed by means of the ***semantic rules***.

Shown in the table below are the semantic rules associated with a grammar production. Along with the semantic rules are predicate rule(s) that are evaluated after the semantic rule for a production.

PRODUCTION RULES	**SEMANTIC RULES & PREDICATE RULES**
START → VIPER HEAD MAIN IS STMT END FUNCTION TAIL	-
STMT → FN_ID L_PAREN FN_ARG R_PAREN SCOLON STMTS	FN_ID.retval := FN_ARG.val FN_ID.rtype := FN_ARG.type
| IN L_PAREN VAR_ID R_PAREN SCOLON STMTS	-
| OUT L_PAREN OUT_ARG R_PAREN SCOLON STMTS	print(OUT_ARG.val)
| LET VAR_ID COLON DATA_TYPE EQUALS LITERAL SCOLON STMTS	VAR_ID.val := LITERAL.val
| VAR_ID STMT2	// Declaration VAR_ID.type := STMT2.type AddType(VAR_ID.name, STMT2.type); // Assignment STMT2.type := VAR_ID.type
| FOR L_PAREN VAR_ID EQUALS FOR_ARG INC_DEC FOR_ARG_1 R_PAREN L_BRC STMT R_BRC STMTS	FOR_ARG.expected_type := VAR_ID.type VAR_ID.val := FOR_ARG.val FOR.val := if (INC_DEC.val == to) { if (FOR_ARG.val < FOR_ARG_1.val) true false } else if (INC_DEC.val == downto) { if (FOR_ARG.val > FOR_ARG_1.val) true false }
| WHILE L_PAREN EXPR R_PAREN L_BRC STMT R_BRC STMTS	WHILE.val := if (EXPR.val == (true | false)) {

	EXPR.val } else error()
| DO L_BRC STMT R_BRC WHILE L_PAREN EXPR R_PAREN SCOLON STMTS	WHILE.val := if (EXPR.val == (true | false)) { EXPR.val } else error()
| IF L_PAREN EXPR R_PAREN L_BRC STMT R_BRC ELSE_PART STMTS	IF.val := if (EXPR.val == (true | false)) { EXPR.val } else error()
| PLUSPLUS VAR_ID SCOLON STMTS	VAR_ID.val := VAR_ID.val + 1
| SUBTSUBT VAR_ID SCOLON STMTS	VAR_ID.val := VAR_ID.val – 1
FUNCTION → FN_ID L_PAREN PARAM R_PAREN RET FN_RET IS STMT END FUNCTIONS	FN_ID.rtype := FN_RET.type addtype(FN_ID.name, FN_RET.type)
| *NULL*	-
FUNCTIONS → FN_ID L_PAREN PARAM R_PAREN RET FN_RET IS STMT END FUNCTION	FN_ID.rtype := FN_RET.type addtype(FN_ID.name, FN_RET.type)
| *NULL*	-
FN_RET → DATA_TYPE	FN_RET.type := DATA_TYPE.type
| VOID	FN_RET.type := void
PARAM → VAR_ID COLON DATA_TYPE PARAMS	PARAM.type := DATA_TYPE.type VAR_ID.type := DATA_TYPE.type AddType(VAR_ID.name, DATA_TYPE.type)
| *NULL*	-
PARAMS → COMMA PARAM	PARAMS.type := PARAM.type
| *NULL*	-
FOR_ARG → INT_LIT	FOR_ARG.val := INT_LIT.lexval FOR_ARG.type := integer <u>Predicate Rule:</u>

	FOR_ARG.type == FOR_ARG.expected_type
| REAL_LIT	FOR_ARG.val := REAL_LIT.lexval FOR_ARG.type := real Predicate Rule: *FOR_ARG.type == FOR_ARG.expected_type*
| VAR_ID	FOR_ARG.val := VAR_ID.val FOR_ARG.type := VAR_ID.dtype Predicate Rule: *FOR_ARG.type == FOR_ARG.expected_type*
STMTS → STMT	STMTS.val := STMT.val
| *NULL*	-
STMT2 → COLON DATA_TYPE STMT3	STMT2.type := DATA_TYPE.type
| EQUALS STMT4	STMT4.expected_type := STMT2.type
| PLUSPLUS SCOLON STMTS	STMT2.val := STMT2.val + 1
| SUBTSUBT SCOLON STMTS	STMT2.val := STMT2.val – 1
STMT3 → SCOLON STMTS	-
| EQUALS LITERAL SCOLON STMTS	LITERAL.expected_type := STMT3.type
| ARRAY OPT_RANGE L_BRAC INT_LIT R_BRAC SCOLON STMTS	STMT3.val :=
STMT4 → LITERAL SCOLON STMTS	LITERAL.expected_type := STMT4.type
| FN_CALL STMTS	FN_CALL.expected_type := STMT4.type
| EXPR SCOLON STMTS	EXPR.expected_type := STMT4.type
| VAR_ID STMT5	VAR_ID.expected_type := STMT4.type
STMT5 → SCOLON STMTS	-
| REL_OP EXPR SCOLON STMTS	STMT5.optr := REL_OP.optr STMT5.val := EXPR.val
| AND EXPR SCOLON STMTS	STMT5.optr := AND.lexval STMT5.val := EXPR.val
| OR EXPR SCOLON STMTS	STMT5.optr := OR.lexval STMT5.val := EXPR.val
| AR_OP EXPR SCOLON STMTS	STMT5.optr := AR_OP.optr STMT5.val := EXPR.val
DATA_TYPE → INT	DATA_TYPE.type := integer
| CHAR	DATA_TYPE.type := char
| CHARS	DATA_TYPE.type := chars
| REAL	DATA_TYPE.type := real

	BOOL	DATA_TYPE.type := boolean
LITERAL → INT_LIT		LITERAL.type := integer LITERAL.val := INT_LIT.lexval
	STR_LIT	LITERAL.type := chars LITERAL.val := STR_LIT.lexval
	CHAR_LIT	LITERAL.type := char LITERAL.val := CHAR_LIT.lexval
	REAL_LIT	LITERAL.type := real LITERAL.val := REAL_LIT.lexval
	BOOL_LIT	LITERAL.type := boolean LITERAL.val := BOOL_LIT.lexval
ELSE_PART → ELSE L_BRC STMT R_BRC		-
	ELSIF L_PAREN EXPR R_PAREN L_BRC STMT R_BRC ELSIF_PART	ELSE_PART.val := if (EXPR.val == (true | false)) { EXPR.val } else error()
	NULL	-
ELSIF_PART → ELSE_PART		ELSIF_PART.val := ELSE_PART.val
	NULL	-
OPT_RANGE → RANGE		-
	NULL	-
INC_DEC → TO		INC_DEC.val := to
	DOWNTO	INC_DEC.val := downto
OUT_ARG → STR_LIT OUT_ARGS		OUT_ARG.val := STR_LIT.lexval OUT_ARGS.val
	VAR_ID OUT_ARGS	OUT_ARG.val := VAR_ID.val OUT_ARGS.val
	NULL	-
OUT_ARGS → COMMA VAR_ID OUT_ARG		OUT_ARGS.val := VAR_ID.val OUT_ARG.val

	NULL	-
FN_CALL → FN_ID L_PAREN FN_ARG R_PAREN SCOLON	FN_CALL.type := FN_ID.rtype FN_CALL.val := FN_ID.retval
FN_ARG → VAR_ID FN_ARGS	FN_ARG.val := VAR_ID.val FN_ARG.type := VAR_ID.type
	NULL	-
FN_ARGS → COMMA FN_ARG	FN_ARGS.val := FN_ARG.val FN_ARGS.type := FN_ARG.type
	NULL	-
EXPR → VAR_ID EXPR_PART1 // requirement: // 1. Operands must have the same data type. Else produce an incompatibility error used on operator // 2. If the operator is arithmetic or relational, the operands must be of type integer or real, then evaluate the expression. Else produce and error (incompatible operands to operator) // 3. If the operator is logical AND or logical OR, the operands must be boolean, otherwise, produce an error	EXPR.type := if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) { if ((VAR_ID.type == integer) && (EXPR_PART1.type == integer)) { integer } else if ((VAR_ID.type == real) && (EXPR_PART1.type == real)) { real } else error() } else EXPR_PART1.type EXPR.val := if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) { if ((VAR_ID.type == integer) && (EXPR_PART1.type == integer)) || ((VAR_ID.type == real) && (EXPR_PART1.type == real)) { VAR_ID.val EXPR_PART1.optr EXPR_PART1.val } else error() } else if (EXPR_PART1.optr == AND || EXPR_PART1.optr == OR) { if (!(VAR_ID.type == boolean) && !(EXPR_PART1.type == boolean)) { error() } } else error() Predicate rule:

	EXPR.type == EXPR.expected_type
| INT_LIT EXPR_PART1 // requirements: // 1. Since one of the operands is of type integer, then EXPR_PART1 must also have type integer. Else produce and incompatibility error used on operator // 2. Operators must only be arithmetic and/or comparison operators.	EXPR.type := if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) { if ((VAR_ID.type == integer) && (EXPR_PART1.type == integer)) { integer } else error() } EXPR.val := if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) { if ((VAR_ID.type == integer) && (EXPR_PART1.type == integer)) { VAR_ID.val EXPR_PART1.optr EXPR_PART1.val } else error() } else error() <u>Predicate rule:</u> *EXPR.type == EXPR.expected_type*
| REAL_LIT EXPR_PART1 // requirements: // 1. Since one of the operands is of type real, then EXPR_PART1 must also have type real. Else produce and incompatibility error used on operator // 2. Operators must only be arithmetic and/or comparison operators. If the operators are arithmetic/comparison, evaluate the expression	EXPR.type := if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) { if ((VAR_ID.type == real) && (EXPR_PART1.type == real)) { real } else error() } EXPR.val := if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) { if ((VAR_ID.type == real) && (EXPR_PART1.type == real)) { VAR_ID.val EXPR_PART1.optr EXPR_PART1.val } else error() } else error() <u>Predicate rule:</u> *EXPR.type == EXPR.expected_type*

	| L_PAREN EXPR_PART3	EXPR.type := EXPR_PART3.type EXPR.val := EXPR_PART3.val
	| NOT EXPR_PART2	EXPR.val := !(EXPR_PART2.val)
EXPR_PART3 →	EXPR R_PAREN	EXPR_PART3.val := EXPR.val EXPR_PART3.type := EXPR.type
EXPR_PART2 →	VAR_ID	EXPR_PART2.val := VAR_ID.val EXPR_PART2.type := VAR_ID.dtype
	| INT_LIT	EXPR_PART2.val := INT_LIT.lexval EXPR_PART2.type := integer
	| REAL_LIT	EXPR_PART2.val := REAL_LIT.lexval EXPR_PART2.type := real
	| L_PAREN EXPR_PART3	EXPR_PART2.val := EXPR_PART3.val EXPR_PART2.type := EXPR_PART3.type
EXPR_PART1 →	REL_OP EXPR	EXPR_PART1.optr := REL_OP.val EXPR_PART1.val := EXPR.val EXPR_PART1.type := EXPR.type
	| OR EXPR	EXPR_PART1.optr := OR.lexval EXPR_PART1.val := EXPR.val EXPR_PART1.type := EXPR.type
	| AND EXPR	EXPR_PART1.optr := AND.lexval EXPR_PART1.val := EXPR.val EXPR_PART1.type := EXPR.type
	| AR_OP EXPR	EXPR_PART1.optr := AR_OP.val EXPR_PART1.val := EXPR.val EXPR_PART1.type := EXPR.type
	| *NULL*	-
REL_OP →	GRTR_THAN	REL_OP.val := GRTR_THAN.lexval
	| LESS_THAN	REL_OP.val := LESS_THAN.lexval
	| EQUAL_TO	REL_OP.val := EQUAL_TO.lexval
	| NOTEQUAL	REL_OP.val := NOTEQUAL.lexval
	| GRTR_THAN_OR_EQ	REL_OP.val := GRTR_THAN_OR_EQ.lexval
	| LESS_THAN_OR_EQ	REL_OP.val := LESS_THAN_OR_EQ.lexval

AR_OP → PLUS	AR_OP.val := PLUS.lexval
| MINUS	AR_OP.val := MINUS.lexval
| MULTI	AR_OP.val := MULTI.lexval
| INT_DIV	AR_OP.val := INT_DIV.lexval
| REAL_DIV	AR_OP.val := REAL_DIV.lexval
| MOD	AR_OP.val := MOD.lexval
| EXP	AR_OP.val := EXP.lexval

ATTRIBUTES UTILIZED IN THE SEMANTIC RULES:

Attribute	**Definition**
expected_type	inherited attribute associated with the non-terminals VAR_ID, EXPR, FN_CALL, LITERAL, FOR_ARG, STMT4,
Type	synthesized attribute associated with the non-terminals LITERAL, FN_CALL, VAR_ID, EXPR, FN_RET, STMT2, PARAM, PARAMS, FOR_ARG, DATA_TYPE, FN_CALL, FN_ARG, FN_ARGS, EXPR_PART1, EXPR_PART2, EXPR_PART3
Lexval	synthesized attribute whose value is assumed to be supplied by the lexical analyzer; associated with FOR_ARG, INT_LIT, REAL_LIT,

expected_type – inherited attribute associated with the non-terminals VAR_ID, EXPR, FN_CALL, LITERAL

type – synthesized attribute associated with the non-terminals LITERAL, FN_CALL, VAR_ID, EXPR, FN_RET

lexval – synthesized attribute whose value is assumed to be supplied by the lexical analyzer

val – inherited attribute from the value of a LITERAL in the symbol table and computed EXPR value

rtype – inherited attribute associated with the non-terminal FN_ID

OPERATIONAL SEMANTICS

EXPRESSIONS

a. ARITHMETIC

Example:

@var1 + @var2

Operational Semantics:

$$\frac{\Gamma \vdash \text{@var1} : \text{int} \quad \Gamma \vdash \text{@var2} : \text{int} \quad \text{op} \in \{+\}}{\Gamma \vdash (\text{@var1 op @var2}) : \text{int}}$$

The operational semantics of the arithmetic expression tells that a function of the finite set @var1 and @var2 has an integer type. With the given value of operation which is "+", The expression should result a sum of the given values for @var1 and @var2.

b. BOOLEAN EXPRESSION

Example:

@var1 || @var2

Operational Semantics:

$$b \in \{\text{true}, \text{false}\}$$

$$\frac{\Gamma \vdash \text{@var1} : \text{bool} \quad \Gamma \vdash \text{@var1} : \text{bool} \quad \text{op} \in \{ \,||\, \}}{\Gamma \vdash (\text{@var1 op @var2}) : \text{bool}}$$

As said, the logical expression only returns a true of false value. With the logical AND, the values of BOTH @var1 and @var2 will return true if the values are true otherwise it'll return to false.

STATEMENTS

a. INPUT Statement

Example:

```
in(@var1);
```

Operational Semantics:

$$\Gamma \vdash \text{in(@var)} : \text{input}$$

The statement will read a string then assign its value to the given variable which is @var1

b. OUTPUT Statement

Example:

```
out("Hello", @name);
```

Operational Semantics:

$$\Gamma \vdash \text{out("Hello",@name)} : \text{output}$$

The statement will print the word "Hello" and also the value given to the variable @name.

c. CONDITIONAL Statements

if statement

Example:

```
if (@var1 == 5) {
      // Statements
}
```

This is an example of an if statement, a conditional statement. It will ask a condition @var1==5. It if returns a true value, it will execute the statements inside of it.

if-else statement

```
EXAMPLE:
      if(@var1 == 5){
          // Statements
      } else {
          // Statements
      }
```

This is an example of an if-else statement, a conditional statement. It will ask a condition @var1==5. If it returns a true value, it will execute the statement inside of it. If it returns a false value, it will go to the else block to execute the statements inside it.

If-elsif-else

```
EXAMPLE:
if(@var1 == 5){
     // Statements
} elsif(@var2 == 6){
     // Statements
} else {
     // Statements
}
```

This is an example of an if-elsif-else statement, a conditional statement. It will ask a condition @var1==5. If it returns a true value, it will execute the statement inside of it. If it returns a false value, it will go to the elsif statement. In an elsif statement, there is also a condition. If the condition returns a true value it will execute the statements inside of it. If the condition returns a false value it will go to the else block to execute the statements in it.

d. LOOPING/ITERATIVE Statements

for loop

```
EXAMPLE:
      for(@var1 = 5 to 10) {
          // Statements
      }

      for(@var2 = 6 downto 0){
          for(@var1 = 5 to 10) {
                  // Statements
           }
```

```
Operational semantics: @var1=5
      loop: if @var1>10 goto loop2
            //statements
            @var1=@var1+1
```

```
            goto loop

      loop2: @var2=6
            if @var2<0 goto done

      loop3:@var1=5
            if @var1>10 goto loop2
            //statements
            @var1=@var1+1
            goto loop3
            @var2=@var2-1
            goto loop2
      done : //statements
```

This is an example of a for loop statement, it means that @var1 will have a value of 5 then it will check the if condition. If the condition returns a false value it will execute the statements inside the loop, then will increment @var1 by 1 then repeat the process starting with the if condition until it returns a value true. If the condition returns a true value the loop will stop then it will go to the next statement, which is another for loop. The next for loop assigns @var2 a value of 6 then ask the if condition is @var2<0. If the condition returns a false value then it will execute the statements inside the loop, in this case another for loop. The inside for loop will assign @var1 a value 5 then will ask an if condition if @var1>10. If it returns a false value it will execute the statements inside of it then will increment @var1 by 1 then will repeat the process the condition returns a true value. If it returns a true value it will return to the previous for loop, then @var2 will decrement by 1 then it will repeat the process until the condition returns a true value. If the condition returns a true value the for loop will stop then the program will continue the remaining statements then exit.

while loop

```
EXAMPLE:
      while(@var2 != 23){
            while(@counter != 0) @counter --;
      }
      while(@counter != 0){
             // Statements
      }
```

```
Operational semantics:
      loop: if @var1!=23 goto loop3
      loop2:if @counter!=0 goto loop
            @counter=@counter-1
            goto loop2
            goto loop
      loop3:if @counter!=0 goto done
            //statements
```

```
            goto loop3
      done : //statements
```

This is an example of a while statement. There are three while statements used in this example. The first while loop has an if condition that if @var1!=23. If it returns a false value it will run the second while statement, the first while statement will continue to loop until the condition returns a true value. The second while statement has an if condition that is if @counter =0. If it returns a false value, it will decrement the @counter. The second while statement will continue to loop until the condition returns a true value. If it returns a true value, it will return to the first loop to continue the process. If the first while statement condition returns a true value it will end its looping process and will now execute the third while statement which has a condition of @counter!=0. If it returns a false value it will execute all the statements in it until the condition returns a true value. If the condition returns a true value the while statement will end then the program will continue to execute the remaining statements then exit the program.

do-while loop

```
EXAMPLE:
      do {
          // Statements
      } while( @x > 5);
            do {
                  while(@apple != NULL) { // Statements }
            } while( @x > 5);
```

```
Operational semantics:
      loop: //statemets
            goto loop
            if @x>5 goto loop 2
      loop2: if @apple != NULL goto loop2
            goto loop2
            if @x>5 goto done
      done://statements
```

This is an example of a do-while loop statement. Do-while loop is different from for loop and while loop because it first executes the statements then it will ask the condition. The first loop will execute the statements inside of it then will ask the if condition @x>5. If it returns a false value it will continue the loop but if it returns true value, the loop will stop and the program will execute the next statements. Then next do-while loop will execute a statement while loop inside its body. The while loop has an if condition @apple!=NULL. It will continue to loop until it returns a true value. If it returns a true value it will then go back to the do while loop, then ask the if condition @x>5. It will continue its process until it returns a true value that will exit the loop statement then continue to execute other statement then will exit the program.

e. ASSIGNMENT Statement

Example:

```
@my_name = “kiko”;
@var_hold : string;
@var_hold = @my_name;
```

Operational Semantics:

```
Γ   ├─   @my_name = “kiko”
Γ   ├─   @var_hold : string
    ──────────────────────
Γ   ├─   @var_hold = @my_name: string
```

statement 1: @my_name = “kiko”
@my_name have a string value “kiko”

statement2: @var_hold : string;
the identifier @var_hold is a string value

statement3: @var_hold = @my_name;
this indicates that the identifier @my_name will have a string value for @var_hold has a constant value of @my_name.

f. DECLARATION

Examples of VARIABLE DECLARATION:

Statement:

@identifier : integer;

Operational semantics:

<integer,@identifier>->integer(@identifier)

The statement means that the variable id, @identifier, is declared as an integer. The variable number can now receive or store an integer. The operational semantic of the statement means that in the state integer, input a variable id, will go to the state integer(variable_id).

Statement:

@apple3 : char;

Operational semantics:

<char,@apple3>->char(@apple3)

This statement means that the variable id, @apple3, is declared as an character. The variable letter can now receive or store a single character. The operational semantic of the statement means that in the state char, input a variable id, will go to the state char(variable_id).

Statement:

word : chars;

Operational semantics:

<chars,word>->chars(word)

This statement means that the variable id, word, is declared as a string or an array of character. It can now receive or store set of characters. The operational semantic of the statement means that in the state chars, input a variable id, will go to the state chars(variable_id).

Statement:

@grapes : real;

Operational semantics:

<real, @grapes >->real(@grapes)

This statement means that the variable id, @grapes, is declared as a real floating point number. The operational semantic of the statement means that in the state real, input a variable id, will go to the state real(variable_id).

Statement:

@orange : boolean;

Operational semantics:

<boolean,@orange>->boolean(@orange)

This statement means that the variable id, @orange, is declared as a boolean. It can receive or store only the value true or false. The operational semantic of the statement means that in the state bool, input a variable id, will go to the state boolean(variable_id).

Examples of VARIABLE INITIALIZATION:

Statement:

@identifier : integer = 5;

Operational semantics:

```
<integer(@identifier),13>=s(@identifier ->s(13))
```

The statement means that the variable id, @identifier, is declared as an integer and assigned the value 13. The operational semantic of the statement means that in the state integer(@identifier), input a value 13, will assign a value 13 from a state that contains a value of 13.

Statement:

```
letter : char='k';
```

Operational semantics:

```
<char(letter),'k'>=s(letter->s('k'))
```

The statement means that the variable id, letter, is declared as a character and assigned the character 'k'. The operational semantic of the statement means that in the state char(letter), input a character 'k', will assign a character 'k' from a state that contains a character 'k'.

Statement:

```
@apple3 : chars = "apple tree";
```

Operational semantics:

```
<chars(@apple3), "apple tree">=s(@apple3->s("apple tree"))
```

The statement means that the variable id, @apple3, is declared as an array of character and assigned the string "apple tree". The operational semantic of the statement means that in the state chars(@apple3), input a string "apple tree", will assign a string "apple tree" from a state that contains a string "apple tree".

Statement:

```
@orange : boolean = false;
```

Operational semantics:

```
<boolean(@orange), false>=s(@orange ->s(false))
```

The statement means that the variable id, @orange, is declared as a boolean and assigned the value false. The operational semantic of the statement means that in the state boolean(@orange), input a value false, will assign a value false from a state that contains a value false.

Statement:

```
@grapes : real = 12.5;
```

Operational semantics:

```
<real(@grapes), 12.5>=s(@grapes ->s(12.5))
```

The statement means that the variable id, @grapes, is declared as a real floating point number and assigned the value 12.5. The operational semantic of the statement means that in the state real(@grapes), input a value 12.5, will assign a value 12.5 from a state that contains a value 12.5.

Examples of CONSTANTS:

Statement:

```
let @const_var : integer = 5;
```

Operational semantics:

```
<let,<integer(@const_var),5>>->constant(@const_var=5)
```

The statement means that the @const_var, declared as an integer and has a value of 5, is declared as a constant value. The operational semantic of the statement means that the integer @const_var,input a value 5, will be an input to the state let and will go to the state constant(@const_var=5).

Statement:

```
let @my_name : chars = “pransisko”;
```

Operational semantics:

```
<let,<chars (@my_name),”pransisko”>>->constant(@my_name=”pransisko”)
```

The statement means that the @my_name, declared as chars and has a string “pransisko”, is declared as a constant value. The operational semantic of the statement means that the chars @my_name,input a string “pransisko”, will be an input to the state let and will go to the state constant(@my_name=“pransisko”).

Statement:

```
let @grades : real = “1.0”;
```

Operational semantics:

```
<let,<real(@grades),”1.0”>>->constant(@grades=”1.0”)
```

The statement means that the @grades, declared as real and has a real value “1.0”,is declared as a constant value. The operational semantic of the statement means that the real @grades, input a real value “1.0”, will be an input to the state let and will go to the state constant(@grades=“1.0”).

Statement:

```
let @pines : boolean = false;
```

Operational semantics:

<let,<boolean(@pines),false>>->constant(@pines=false)

The statement means that the @pines, declared as a boolean and has value false,is declared as a constant value. The operational semantic of the statement means that the boolean @pines,input a value false, will be an input to the state let and will go to the state constant(@pines=false).

SEMANTIC ERRORS

DATA TYPE MISMATCH

Example:

```
main is
      @prod_price: real;
      @sales_comm: integer;
      @sales_disc: real;

      @prod_price = @sales_comm + @sales_disc; #data type mismatch#
end
```

Semantic Rule:

Operands must have the same data type. Else produce and incompatibility error used on operator.

```
if ((is_arith(EXPR_PART1.optr) || (is_relop(EXPR_PART1.optr)) {
      if ((VAR_ID.type == integer) &&(EXPR_PART1.type == integer)) {
            integer
      }
      else if ((VAR_ID.type == real) && (EXPR_PART1.type == real)) {
            real
      }
      else error()
} else EXPR_PART1.type
```

IDENTIFIER USED BUT NOT DECLARED

Example:

```
main is
      @local: integer;
      @non_local: integer;
      @answer = @local + @non_local; #identifier used but not declared#
end
```

Semantic Rule:

Identifier must be declared either globally or locally. Else identifier is not declared.

IDENTIFIER DECLARED BUT NOT USED

Example:

```
main is
      @unused: chars; //identifier declared but not used
end
```

Semantic Rule:

Identifier must be used when it has been declared either globally or locally. Else identifier is not used and will prompt a warning message.

DOUBLE DECLARATION OF IDENTIFIER

Example:

```
main is
      @prod_price: real;
      @custm_disc: real;

      if (@custm_disc > 10) {
            @prod_price: real; #double declaration#
      }
end
```

Semantic Rule:

Identifier must be declared only once either globally or locally.