***Name: Nguyen Huu Hoang Hai Anh***

***Student ID: 20226010***

**LAB 05. TYPE CHECKING No.2**

Table of Contents

[1. a call to a procedure contains fewer arguments than it is declared. 2](#_Toc198535656)

[2. a call to a function contains more arguments than it is declared. 3](#_Toc198535657)

[3. a call to a non\_parametric procedure contains some arguments. 4](#_Toc198535658)

[4. a call with empty list of arguments to a function declared with parameters 5](#_Toc198535659)

[5. type inconsistency in indexes of an array. 6](#_Toc198535660)

[6. a variable declared as an array, but it is used as a single variable 6](#_Toc198535661)

[7. an array is used with fewer dimensions than declared. 6](#_Toc198535662)

[8. an array is used with more dimensions than declared. 7](#_Toc198535663)

[9. use an expression as an argument  for  a call-by-reference  parameter. 7](#_Toc198535664)

[10. the type of an argument does not match the type of its corresponding parameter 7](#_Toc198535665)

[11. A(.4.)(.2.) is used in some statement, but array A was declared with 2 rows and 4 columns. 8](#_Toc198535666)

# ***a call to a procedure contains fewer arguments than it is declared.***

|  |
| --- |
| PROGRAM error1;  PROCEDURE P(a: integer; b: integer);  BEGIN  END;  BEGIN  CALL P(10);  END. |

***A screenshot of a computer

AI-generated content may be incorrect.***

***Analysis***

Parsing the procedure declaration

* When the parser sees PROCEDURE P(a: integer; b: integer), it calls compileProcDecl().
* Inside compileProcDecl(), it registers a new procedure P, enters its own scope, and calls compileParams() to read its parameter list.
* compileParams() creates two parameter objects (a and b) of type integer and links them into procAttrs->paramList.

Parsing the CALL statement

* In the program’s main block, the parser encounters CALL P(10); and invokes compileCallSt().
* compileCallSt() does three things:
  1. Reads the CALL keyword.
  2. Reads the identifier P and checks that P was declared.
  3. Calls compileArguments(proc->procAttrs->paramList) with the list of expected parameters.

Matching arguments to parameters

* compileArguments(ObjectNode\* paramList) walks through the parameter list (a, then b) while reading the actual arguments inside the parentheses:
* After reading a single argument 10, node moves to the second parameter (b) but finds no more commas/arguments.
* Since node != NULL (there’s still parameter b without a matching argument), it calls error(ERR\_PARAMETERS\_ARGUMENTS\_INCONSISTENCY, lineNo, colNo);

# ***a call to a function contains more arguments than it is declared.***

|  |
| --- |
| PROGRAM error2;  FUNCTION F(x: integer): integer;  BEGIN  F := x  END;  BEGIN  WRITEI(F(1, 2));  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

Declaring the function F

* On seeing FUNCTION F(x: integer): integer;, the parser calls compileFuncDecl() to register F. Inside it:
  + It creates the function object F and enters its own scope.
  + It invokes compileParams(), which reads the single parameter x: integer and links it into funcObj->funcAttrs->paramList.

Parsing the WRITEI(F(1, 2)); statement

* In the main block’s statements, compileStatement() sees WRITEI and dispatches to the write routine (e.g. compileWriteSt()), which in turn calls compileExpression() to evaluate its argument.
* Within compileExpression(), it eventually calls compileFactor(). When lookAhead is TK\_IDENT (F), compileFactor() does:
* it recognizes F as a function and immediately calls compileArguments() with the one-element parameter list for x.

Mismatch in argument count

* compileArguments(ObjectNode\* paramList) expects exactly one argument but sees two
* After consuming the first argument (1), node becomes NULL (no second parameter). Encountering the comma before the second argument (2), it finds node == NULL and calls

error(ERR\_PARAMETERS\_ARGUMENTS\_INCONSISTENCY, currentToken->lineNo, currentToken->colNo);

# ***a call to a non\_parametric procedure contains some arguments.***

|  |
| --- |
| PROGRAM error3;  PROCEDURE Q;  BEGIN  END;  BEGIN  CALL Q(1);  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

*Similar explanation to the example section for error number 1 (error1.kpl)*

compileArguments() detects the mismatch

eat(SB\_LPAR); // sees and consumes '('

if (node == NULL) {

// No parameters expected, but '(' indicates at least one argument

error(ERR\_PARAMETERS\_ARGUMENTS\_INCONSISTENCY,

currentToken->lineNo, currentToken->colNo);

}

* Because node is NULL (procedure Q expects zero parameters) yet the code finds '(', the very first check if (node == NULL) fires, calling error(ERR\_PARAMETERS\_ARGUMENTS\_INCONSISTENCY, lineNo, colNo);

# ***a call with empty list of arguments to a function declared with parameters***

|  |
| --- |
| PROGRAM error4;  FUNCTION G(a: integer; b: integer): integer;  BEGIN  G := 4;  END;  BEGIN  CALL WRITEI(G);  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

*Similar explanation to the example section for error number 2 (error2.kpl)*

Declaring G

* On seeing FUNCTION G(a: integer; b: integer): integer;, the parser calls compileFuncDecl() to register G, enters its own scope, and invokes compileParams(), which builds a two-element parameter list (a, b) linked in funcObj->funcAttrs->paramList.

Parsing WRITEI(G)

* In the main block, compileStatement() spots WRITEI and routes to the write routine. That routine calls compileExpression(), which in turn calls compileFactor().
* In compileFactor(), the TK\_IDENT case for a function does

eat(TK\_IDENT);

obj = checkDeclaredIdent("G");

/\* ... \*/

case OBJ\_FUNCTION:

compileArguments(obj->funcAttrs->paramList);

* It immediately invokes compileArguments() with the two-parameter list for G

Detecting “no parentheses” mismatch

* Inside compileArguments(ObjectNode\* paramList), the lookahead token is not (,
* Because paramList is non-NULL (two parameters expected) yet no argument list was supplied, the very first check in this branch fires and reports ERR\_PARAMETERS\_ARGUMENTS\_INCONSISTENCY at the current position.

if (paramList != NULL) {

error(ERR\_PARAMETERS\_ARGUMENTS\_INCONSISTENCY, …);

}

# ***type inconsistency in indexes of an array.***

|  |
| --- |
| PROGRAM error5;  VAR A: ARRAY(. 5 .) OF INTEGER;  BEGIN  A(. 'c' .) := 10;  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

compileLValue() sees A, confirms it’s a variable of array type, and calls compileIndexes() to parse the (. ’c’ .) selector

Inside compileIndexes(Type\* arrayType):

while (lookAhead->tokenType == SB\_LSEL) { // sees '(.'

eat(SB\_LSEL);

Type\* idxType = compileExpression(); // parses 'c'

Type check on the index

* compileExpression() on 'c' returns charType.
* checkIntType(charType) immediately fails, because only TP\_INT is allowed for an array index. That function is implemented as

void checkIntType(Type\* type) {

if (type != NULL && type->typeClass == TP\_INT) return;

else error(ERR\_TYPE\_INCONSISTENCY, currentToken->lineNo, currentToken->colNo);

}

and so reports ERR\_TYPE\_INCONSISTENCY at the position of the 'c' token.

# ***a variable declared as an array, but it is used as a single variable***

|  |
| --- |
| PROGRAM error6;  VAR B: ARRAY(. 3 .) OF integer;  BEGIN  B := 5;  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

Parsing the assignment statement

* In the main block, the parser sees B := 5; and calls compileAssignSt():
* Here, compileAssignSt() will (1) parse the left‐hand side, (2) parse the right‐hand side, then (3) invoke checkTypeEquality to make sure their types match

Determining the left‐hand side type (compileLValue)

* compileLValue() sees the identifier B, looks it up as a variable of array type, and then—because there is no index selector ((. … .))—immediately does a basic‐type check on the array itself:

Type\* compileLValue(void) {

…

eat(TK\_IDENT); // consumes 'B'

Object\* var = checkDeclaredLValueIdent(currentToken->string);

if (var->varAttrs->type->typeClass == TP\_ARRAY)

varType = compileIndexes(var->varAttrs->type);

else

varType = var->varAttrs->type;

return varType;

}

* Since lookAhead is SB\_ASSIGN (not SB\_LSEL), compileIndexes immediately calls checkBasicType on the full array type (which fails to satisfy “integer or char”) but nonetheless returns the array type as varType

Parsing the right‐hand side expression

* compileExpression() sees the numeric literal 5 and returns the built-in integer type (TP\_INT).

Type‐equality check and error

* Finally, compileAssignSt calls: checkTypeEquality(varType, expType);
* Since type1 is an array type and type2 is an integer type, compareType(...) != 0, so the compiler raises ERR\_TYPE\_INCONSISTENCY at the := token

# ***an array is used with fewer dimensions than declared.***

|  |
| --- |
| PROGRAM error7;  VAR C: ARRAY(. 2 .) OF ARRAY(. 3 .) OF integer;  BEGIN  C(.1.) := 1;  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

Parsing the assignment

* The statement C(.1.) := 1; invokes compileAssignSt():

Type\* varType = compileLValue();

eat(SB\_ASSIGN);

Type\* expType = compileExpression();

checkTypeEquality(varType, expType);

* It first parses the left-value (compileLValue()), then the right-value (compileExpression()), and finally checks that their types are equal

Parsing the left-value and its index

* In compileLValue(), seeing C as a variable of array type, it calls compileIndexes() on C’s declared type (ARRAY(.2.) OF ARRAY(.3.) OF INTEGER) .
* Inside compileIndexes(Type\* arrayType)

while (lookAhead->tokenType == SB\_LSEL) {

eat(SB\_LSEL);

Type\* idxType = compileExpression(); // parses '1' → intType

checkIntType(idxType); // OK for integer index

checkArrayType(arrayType); // OK for outer array

arrayType = arrayType->elementType; // unwrap one dimension

eat(SB\_RSEL);

}

checkBasicType(arrayType);

return arrayType;

* After consuming the single index (.1.), arrayType becomes ARRAY(.3.) OF INTEGER (the inner array type).
* Exiting the loop, it calls checkBasicType(arrayType) to ensure the indexed result is a basic type (integer or char). But here arrayType->typeClass is TP\_ARRAY, not TP\_INT or TP\_CHAR, so checkBasicType fails and reports ERR\_TYPE\_INCONSISTENCY

# ***an array is used with more dimensions than declared.***

|  |
| --- |
| PROGRAM error8;  VAR D: ARRAY(. 4 .) OF integer; (\* D chỉ 1 chiều \*)  BEGIN  D(.1.)(.2.);  END. |

A screen shot of a computer

AI-generated content may be incorrect.

***Analysis***

* First index handling  
  When the parser sees D(.1.)(.2.), it calls compileLValue(), recognizes D as a one-dimensional array, and enters compileIndexes(). On the first (.1.), it confirms 1 is an integer, checks that the array is valid, and steps into its element type, which is now integer.
* Error on the second index  
  Still inside the compileIndexes() loop, it encounters the second selector (.2.). At this point, arrayType is already integer, not an array. The call to checkArrayType(integer) fails because only TP\_ARRAY is accepted, triggering ERR\_TYPE\_INCONSISTENCY at the start of the second selector.

# ***use an expression as an argument  for  a call-by-reference  parameter.***

|  |
| --- |
| PROGRAM error9;  VAR n: INTEGER;  m: INTEGER;  PROCEDURE R(var x: integer);  BEGIN  x := x+1;  END;  BEGIN  n := 1;  m := 2;  CALL R(n+m);  END. |

A screen shot of a black background

AI-generated content may be incorrect.

Choosing between L-value and expression

* Because R’s parameter x is declared with VAR, compileArgument() takes the reference branch and calls compileLValue(). This consumes only the identifier n (recognizing it as a variable), then returns its type (integer).

Detecting leftover tokens

* When compileLValue() returns, the next token is the + of n+m. The check

if (lookAhead->tokenType != SB\_COMMA && lookAhead->tokenType != SB\_RPAR)

error(ERR\_TYPE\_INCONSISTENCY, …);

sees that + is neither a comma nor a closing parenthesis, so it raises ERR\_TYPE\_INCONSISTENCY. This catches the fact that you didn’t pass a *pure* variable (l-value) but an expression.

# ***the type of an argument does not match the type of its corresponding parameter***

|  |
| --- |
| PROGRAM error10;  PROCEDURE S(a: integer; VAR b: char);  BEGIN  END;  BEGIN  CALL S('c', 10);  END. |

A screenshot of a computer

AI-generated content may be incorrect.

***Analysis***

Procedure declaration

compileProcDecl() is called, enters S’s scope, and invokes compileParams(). This builds a two‐node parameter list:

* a of kind PARAM\_VALUE and type integer,
* b of kind PARAM\_REFERENCE and type char

Argument matching and first‐parameter type check  
In the main block, CALL S('c', 10); triggers compileCallSt(), which then calls compileArguments(proc->procAttrs->paramList);

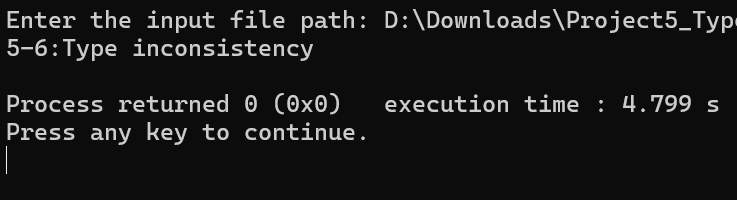
Inside compileArguments(), for the first node (a): since it’s a value parameter, compileArgument() invokes compileExpression() on 'c', yielding charType, then calls

checkTypeEquality(param->paramAttrs->type, type);

Here param->paramAttrs->type is integer, so comparing integer vs. char fails and immediately raises ERR\_TYPE\_INCONSISTENCY at the location of 'c'

# ***A(.4.)(.2.) is used in some statement, but array A was declared with 2 rows and 4 columns.***

|  |
| --- |
| PROGRAM error11;  VAR A: ARRAY(.2.) OF ARRAY(.4.) OF INTEGER; (\* 2 hàng, 4 cột \*)  BEGIN  A(.4.)(.2.) := 1;  END. |



Parsing the assignment and entering compileLValue  
The statement A(.4.)(.2.) := 1; invokes compileAssignSt(), which first calls compileLValue() to determine the left‐hand side type .

First selector and bounds check  
Inside compileLValue(), because A is a two‐dimensional array, it enters the compileIndexes(Type\* arrayType) loop. For the first selector (.4.):

* compileExpression() reads 4 (an integer).
* checkIntType(type) passes since it’s an integer.
* checkArrayType(arrayType) passes since arrayType is TP\_ARRAY.
* New bound check: indexValue = 4 is compared against arrayType->arraySize == 2 (the number of rows). Because 4 > 2, the code calls

error(ERR\_TYPE\_INCONSISTENCY, currentToken->lineNo, currentToken->colNo);

if (currentToken->tokenType == TK\_NUMBER) {

int indexValue = currentToken->value;

if (indexValue > arrayType->arraySize || indexValue < 1) {

error(ERR\_TYPE\_INCONSISTENCY, currentToken->lineNo, currentToken->colNo);

}

}