Báo cáo Thực hành Xây dựng chương trình dịch
Bài 3: Phân tích ngữ nghĩa

Họ và tên: Lê Hải Yến
MSSV: 20225780

I. Phân tích kiến trúc Symbol Table

1. Kiến trúc tổng quan của hệ thống Symbol Table

Hệ thống bảng ký hiệu được thiết kế theo mô hình phân cấp ba tầng, bao gồm:

- SymTab (Bảng ký hiệu chính): Quản lý toàn bộ thông tin về chương trình
- Scope (Phạm vi): Biểu diễn các khối khai báo lồng nhau
- Object (Đối tượng): Lưu trữ thông tin chi tiết của từng định danh

Mối quan hệ giữa các thành phần:

- SymTab duy trì con trỏ đến đối tượng chương trình (program) và scope đang hoạt động (currentScope)
- Mỗi Scope chứa danh sách các Object (objList), biết chủ sở hữu của nó (owner) và liên kết với scope bên ngoài (outer)
- Object lưu tên, loại (kind) và các thuộc tính đặc trưng tùy theo loại

2. Cấu trúc dữ liệu Object

a. Các loại Object (ObjectKind)

Hệ thống hỗ trợ 7 loại đối tượng:

- OBJ_CONSTANT: Biểu diễn hằng số - chứa giá trị (int/char)
- OBJ_TYPE: Định nghĩa kiểu dữ liệu - lưu kiểu thực tế (actualType)
- OBJ_VARIABLE: Biến - ghi nhớ kiểu dữ liệu và scope khai báo
- OBJ_FUNCTION: Hàm - quản lý tham số, kiểu trả về và scope nội bộ
- OBJ_PROCEDURE: Thủ tục - lưu danh sách tham số và scope riêng
- OBJ_PARAMETER: Tham số hình thức - chứa thông tin về kiểu và cách truyền
- OBJ_PROGRAM: Chương trình gốc - có scope toàn cục

b. Cơ chế lưu trữ thuộc tính

Để tối ưu bộ nhớ, mỗi Object sử dụng union chứa con trỏ đến attributes tương ứng với kind. Ví dụ:

- Function chứa funcAttrs với paramList, returnType, scope
- Variable chỉ cần varAttrs với type và scope

3. Hệ thống quản lý Type

a. Phân loại kiểu dữ liệu

Ngôn ngữ KPL hỗ trợ ba loại kiểu:

- TP_INT: Kiểu số nguyên
- TP_CHAR: Kiểu ký tự
- TP_ARRAY: Kiểu mảng - chứa kích thước (arraySize) và kiểu phần tử (elementType)

b. Các thao tác trên Type

Tạo kiểu cơ bản:

- makeIntType(): Cấp phát và khởi tạo Type với typeClass = TP_INT
- makeCharType(): Tương tự cho TP_CHAR
- makeArrayType(): Tạo kiểu mảng với kích thước và kiểu phần tử cho trước

Sao chép kiểu:

- duplicateType(): Thực hiện deep copy, đặc biệt xử lý đệ quy cho mảng nhiều chiều

So sánh kiểu:

- compareType(): Kiểm tra tính tương đồng bằng cách so sánh typeClass, với mảng thì so sánh thêm arraySize và elementType một cách đệ quy

4. Quản lý giá trị hằng (ConstantValue)

Cấu trúc ConstantValue sử dụng union để tiết kiệm bộ nhớ, lưu intValue hoặc charValue cùng với trường type để phân biệt.

Các hàm hỗ trợ:

- makeIntConstant(int i): Tạo hằng số nguyên
- makeCharConstant(char ch): Tạo hằng ký tự
- duplicateConstantValue(): Sao chép giá trị hằng

5. Cơ chế tạo và quản lý Object

a. Nguyên tắc chung

Mỗi loại Object có hàm tạo chuyên biệt (createXxxObject), thực hiện:

- Cấp phát bộ nhớ cho Object
- Sao chép tên vào trường name
- Gán giá trị cho trường kind
- Cấp phát và khởi tạo attributes tương ứng

b. Xử lý đặc biệt

- createProgramObject(): Tạo scope gốc và gán vào symtab->program

- createFunctionObject() và createProcedureObject(): Tự động tạo scope riêng với owner trỏ về chính nó
- createParameterObject(): Yêu cầu truyền vào owner (function/procedure chứa nó)
- createVariableObject(): Lưu con trỏ đến currentScope để biết nơi khai báo

6. Quản lý danh sách Object (ObjectNode)

a. Cấu trúc linked list

ObjectNode là nút của danh sách liên kết đơn:

```
struct ObjectNode_ {
  Object *object;
  struct ObjectNode_ *next;
};
```

b. Thao tác trên danh sách

addObject(): Thêm Object vào cuối danh sách

- Nếu danh sách rỗng: gán node mới làm đầu
- Ngược lại: duyệt đến cuối rồi nối node mới

findObject(): Tìm Object theo tên

- Duyệt tuần tự danh sách
- So sánh chuỗi với `strcmp()`
- Trả về NULL nếu không tìm thấy

7. Cơ chế Scope và phạm vi khai báo

a. Ý nghĩa của Scope

Mỗi Scope đại diện cho một phạm vi khai báo, chứa:

- `objList`: Danh sách các Object được khai báo trong phạm vi này
- `owner`: Object sở hữu scope (program/function/procedure)
- `outer`: Con trỏ đến scope bao ngoài

b. Cấu trúc phân cấp

Các Scope tạo thành cây phân cấp:

- Scope toàn cục (program) có `outer = NULL`
- Scope của function/procedure có `outer` trỏ đến scope nơi khai báo
- Chuỗi liên kết qua `outer` cho phép tra cứu từ trong ra ngoài

8. Khởi tạo Symbol Table

a. Hàm initSymTab()

Thực hiện các công việc:

- Cấp phát bộ nhớ cho SymTab
- Khởi tạo `globalObjectList = NULL`

- Tạo các hàm/thủ tục built-in của ngôn ngữ

b. Các hàm built-in
   - READC: Hàm đọc ký tự, không tham số, trả về CHAR
   - READI: Hàm đọc số nguyên, không tham số, trả về INT
   - WRITEI: Thủ tục xuất số nguyên, nhận tham số INT (PARAM_VALUE)
   - WRITEC: Thủ tục xuất ký tự, nhận tham số CHAR (PARAM_VALUE)
   - WRITELN: Thủ tục xuống dòng, không tham số

c. Tạo kiểu toàn cục
   Hai biến toàn cục `intType` và `charType` được tạo một lần duy nhất để tái sử dụng trong toàn bộ chương trình, tránh cấp phát lặp lại.

9. Quản lý Scope động

a. enterBlock(Scope* scope)
   Chuyển `currentScope` sang scope mới khi vào một khối code. Các khai báo tiếp theo sẽ thuộc scope này.

b. exitBlock()
   Di chuyển `currentScope` về scope bên ngoài (`outer`) khi kết thúc khối. Đảm bảo quản lý phạm vi đúng theo cấu trúc lồng nhau.
   Luồng hoạt động:
   - Khi gặp function/procedure: gọi `enterBlock()` với scope của nó
   - Mọi khai báo trong thân hàm thuộc scope này
   - Kết thúc hàm: gọi `exitBlock()` để quay lại scope cha

10. Tìm kiếm Object (Lookup)

a. Hàm lookupObject(char *name)
   Tìm kiếm Object theo tên với cơ chế phạm vi lồng nhau:
   Thuật toán:
   - Bắt đầu từ `currentScope`
   - Gọi `findObject()` tìm trong `objList` của scope hiện tại
   - Nếu tìm thấy: trả về ngay
   - Không tìm thấy: chuyển sang `outer` scope
   - Lặp lại cho đến `outer = NULL`
   - Cuối cùng tìm trong `globalObjectList`
   - Không tìm thấy ở đâu: trả về NULL

b. Nguyên tắc shadowing
   Khai báo ở scope trong sẽ che khuất khai báo cùng tên ở scope ngoài, vì tìm kiếm ưu tiên scope trong trước.

11. Khai báo Object vào Symbol Table

a. Hàm declareObject(Object* obj)

Thêm Object vào vị trí thích hợp trong symbol table.

Xử lý đặc biệt cho Parameter:

- Kiểm tra `obj->kind == OBJ_PARAMETER`
- Lấy `owner` từ `currentScope->owner`
- Tùy theo `owner->kind`, thêm vào `funcAttrs->paramList` hoặc `procAttrs->paramList`
- Sau đó thêm vào `objList` của scope

Các loại khác:

- Thêm trực tiếp vào `currentScope->objList`

Lý do: Function và Procedure cần danh sách tham số riêng để kiểm tra khi gọi hàm, nên Parameter được lưu ở hai nơi.

II. <u>Tích hợp Symtab vào Parser</u>

1. Khởi tạo và giải phóng trong compile()

Luồng hoạt động:

Hàm `compile()` trong parser.c thực hiện theo trình tự:

openInputStream() → Mở file nguồn

↓

Khởi tạo currentToken và lookAhead

↓

initSymTab() → Tạo symbol table với built-in

↓

compileProgram() → Phân tích và điền thông tin

↓

printObject() → In symbol table ra màn hình

↓

cleanSymTab() → Giải phóng bộ nhớ

↓

Giải phóng token và đóng file

2. Biên dịch PROGRAM và thiết lập scope chính

Hàm compileProgram()

```
void compileProgram(void) {
  // Create, enter, and exit program block
  Object* programObject;

  eat(KW_PROGRAM);
  eat(TK_IDENT);

  programObject = createProgramObject(currentToken->string);

  enterBlock(programObject->progAttrs->scope);

  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_PERIOD);

  exitBlock();
}
```

Giải thích:

- Sau khi nhận diện cú pháp PROGRAM và lấy tên chương trình
- createProgramObject() tạo đối tượng chương trình với scope riêng, gán vào symtab->program
- enterBlock() thiết lập context để các khai báo toàn cục thuộc scope này
- Sau khi biên dịch xong block: exitBlock() quay về scope NULL

3. **Biên dịch khai báo CONST**

Hàm compileBlock()

```
void compileBlock(void) {
  // Create and declare constant objects
  Object* constantObject;
  ConstantValue* value;

  if (lookAhead->tokenType == KW_CONST) {
    eat(KW_CONST);

    do {
      eat(TK_IDENT);

      checkFreshIdent(currentToken->string);
      constantObject = createConstantObject(currentToken->string);

      eat(SB_EQ);
      value = compileConstant();

      constantObject->constAttrs->value = value;
      declareObject(constantObject);

      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);

    compileBlock2();
  }
  else compileBlock2();
}
```

Quy trình xử lý từng hằng:

- Lấy tên hằng từ TK_IDENT
- Kiểm tra tên chưa bị trùng trong scope bằng checkFreshIdent()
- Tạo Object với createConstantObject()
- Phân tích giá trị bằng compileConstant()
- Gán value vào constAttrs
- Đăng ký vào symbol table bằng declareObject()

4. Biên dịch khai báo TYPE

a. Hàm compileBlock2()

Cấu trúc tương tự compileBlock() nhưng xử lý TYPE:

```
void compileBlock2(void) {
  // Create and declare type objects
  Object* typeObject;
  Type* typeDefinition;

  if (lookAhead->tokenType == KW_TYPE) {
    eat(KW_TYPE);

    do {
      eat(TK_IDENT);

      checkFreshIdent(currentToken->string);
      typeObject = createTypeObject(currentToken->string);

      eat(SB_EQ);
      typeDefinition = compileType();

      typeObject->typeAttrs->actualType = typeDefinition;
      declareObject(typeObject);

      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);

    compileBlock3();
  }
  else compileBlock3();
}
```

b. Hàm compileType()

Xử lý các loại kiểu:

- INTEGER: Gọi makeIntType()
- CHAR: Gọi makeCharType()
- ARRAY: Gọi makeArrayType() với kích thước và kiểu phần tử
- TK_IDENT: Tra cứu kiểu đã khai báo bằng checkDeclaredType(), sau đó duplicateType()

5. Biên dịch khai báo VAR

a. Hàm compileBlock3()

```
void compileBlock3(void) {
  // Create and declare variable objects
  Object* variableObject;
  Type* variableType;

  if (lookAhead->tokenType == KW_VAR) {
    eat(KW_VAR);

    do {
      eat(TK_IDENT);

      checkFreshIdent(currentToken->string);
      variableObject = createVariableObject(currentToken->string);

      eat(SB_COLON);
      variableType = compileType();

      variableObject->varAttrs->type = variableType;
      declareObject(variableObject);

      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);

    compileBlock4();
  }
  else compileBlock4();
}
```

createVariableObject() tự động lưu currentScope vào varAttrs->scope, giúp biến nhớ nơi khai báo của mình.

6. **Biên dịch khai báo FUNCTION**

   Hàm compileFuncDecl()

```
void compileFuncDecl(void) {
  // Create and declare a function object
  Object* functionObject;
  Type* functionReturnType;

  eat(KW_FUNCTION);
  eat(TK_IDENT);

  checkFreshIdent(currentToken->string);
  functionObject = createFunctionObject(currentToken->string);

  declareObject(functionObject);
  enterBlock(functionObject->funcAttrs->scope);

  compileParams();

  eat(SB_COLON);
  functionReturnType = compileBasicType();
  functionObject->funcAttrs->returnType = functionReturnType;

  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_SEMICOLON);

  exitBlock();
}
```

Các giai đoạn:

- Tạo Object: Nhận diện cú pháp và tạo function object
- Đăng ký: declareObject() thêm vào scope cha
- Vào scope con: enterBlock() để xử lý tham số và khai báo local
- Xử lý tham số: compileParams() thêm các tham số vào scope
- Kiểu trả về: compileBasicType() và gán vào returnType
- Thân hàm: compileBlock() xử lý khai báo local và lệnh
- Thoát scope: exitBlock() quay về scope cha

7. **Biên dịch khai báo PROCEDURE**

   Hàm compileProcDecl()

   Tương tự compileFuncDecl() với các điểm khác:

   - Dùng createProcedureObject() thay vì createFunctionObject()
   - Không có phần xử lý kiểu trả về
   - Các bước khác giống hệt

8. **Biên dịch Parameters**

   Hàm compileParam()

```
void compileParam(void) {
  // Create and declare a parameter
  Object* parameterObject;
  Type* parameterType;
  enum ParamKind kind;

  switch (lookAhead->tokenType) {
  case TK_IDENT:
    kind = PARAM_VALUE;
    eat(TK_IDENT);

    checkFreshIdent(currentToken->string);
    parameterObject = createParameterObject(currentToken->string, kind, symtab->currentScope->owner);

    eat(SB_COLON);
    parameterType = compileBasicType();
    parameterObject->paramAttrs->type = parameterType;

    declareObject(parameterObject);
    break;
```

```
  case KW_VAR:
    kind = PARAM_REFERENCE;
    eat(KW_VAR);
    eat(TK_IDENT);

    checkFreshIdent(currentToken->string);
    parameterObject = createParameterObject(currentToken->string, kind, symtab->currentScope->owner);

    eat(SB_COLON);
    parameterType = compileBasicType();
    parameterObject->paramAttrs->type = parameterType;

    declareObject(parameterObject);
    break;
  default:
    error(ERR_INVALID_PARAMETER, lookAhead->lineNo, lookAhead->colNo);
    break;
  }
}
```

Hai loại tham số:

- PARAM_VALUE: Bắt đầu bằng TK_IDENT, truyền theo giá trị
- PARAM_REFERENCE: Bắt đầu bằng VAR, truyền theo tham chiếu

Xử lý trong declareObject():

- Parameter được thêm vào cả objList của scope (để tra cứu tên)
- Và paramList của owner function/procedure (để duy trì thứ tự)

9. Kiểm tra semantic cơ bản

a. checkFreshIdent(char *name)

Kiểm tra tên chưa bị trùng trong scope hiện tại:

- Dùng findObject() tìm trong currentScope->objList (chỉ scope hiện tại)
- Nếu tìm thấy: báo lỗi ERR_DUPLICATE_IDENT

b. checkDeclaredIdent(char *name)

Kiểm tra identifier đã được khai báo:

- Dùng lookupObject() tìm trong toàn bộ chuỗi scope
- Nếu không tìm thấy: báo lỗi ERR_UNDECLARED_IDENT

c. Các hàm kiểm tra chuyên biệt

- checkDeclaredConstant(): Kiểm tra kind = OBJ_CONSTANT
- checkDeclaredType(): Kiểm tra kind = OBJ_TYPE
- checkDeclaredVariable(): Kiểm tra kind = OBJ_VARIABLE
- checkDeclaredFunction(): Kiểm tra kind = OBJ_FUNCTION
- checkDeclaredProcedure(): Kiểm tra kind = OBJ_PROCEDURE

10. Xử lý hằng số trong biểu thức

a. compileConstant2()

Xử lý hằng không dấu:

- TK_NUMBER: Gọi makeIntConstant() với value từ token
- TK_IDENT: Gọi checkDeclaredConstant() để lấy Object, sau đó duplicateConstantValue() để sao chép giá trị

b. compileConstant()

Xử lý hằng có dấu:

- Với dấu trừ: Gọi compileConstant2(), kiểm tra type = TP_INT, rồi đảo dấu
- Với TP_CHAR: Báo lỗi ERR_TYPE_INCONSISTENCY

11. Xử lý trong compileFactor()

Trường hợp TK_IDENT

Parser phân tích context sử dụng dựa vào lookAhead:

- Trường hợp 1 - SB_LPAR (Function call):

```
case SB_LPAR:
  // Must be a function - call specific check
  obj = checkDeclaredFunction(currentToken->string);
  compileArguments();
  break;
```

  + Kiểm tra ngay là function bằng checkDeclaredFunction()
  + Nếu không phải: báo lỗi ERR_INVALID_FUNCTION

- Trường hợp 2 - SB_LSEL (Array indexing):

```
case SB_LSEL:
  // Must be a variable (array) - can be variable or parameter
  obj = checkDeclaredIdent(currentToken->string);
  if (obj->kind != OBJ_VARIABLE && obj->kind != OBJ_PARAMETER)
    error(ERR_INVALID_VARIABLE, currentToken->lineNo, currentToken->colNo);
  compileIndexes();
  break;
```

+ Kiểm tra phải là biến hoặc tham số
+ Chỉ hai loại này mới có thể index như mảng

- Trường hợp 3 - Default (Simple identifier):

```
default:
  // Can be variable, parameter, constant, or function (for return value)
  obj = checkDeclaredIdent(currentToken->string);
  if (obj->kind != OBJ_VARIABLE && obj->kind != OBJ_PARAMETER &&
      obj->kind != OBJ_CONSTANT && obj->kind != OBJ_FUNCTION)
    error(ERR_INVALID_FACTOR, currentToken->lineNo, currentToken->colNo);
  break;
}
```

+ Cho phép 4 loại: biến, tham số, hằng, tên hàm (cho giá trị trả về)

12. Biên dịch lệnh CALL

compileCallSt()

```
void compileCallSt(void) {
  eat(KW_CALL);
  eat(TK_IDENT);

  // Check if it's a declared procedure
  checkDeclaredProcedure(currentToken->string);

  compileArguments();
}
```

Cơ chế checkDeclaredProcedure():

- Kiểm tra tồn tại: lookupObject() tìm trong symbol table
- Kiểm tra kind: Phải là OBJ_PROCEDURE, nếu không báo ERR_INVALID_PROCEDURE

13. Biên dịch lệnh gán (Assignment)

Hàm compileLValue():

Xử lý vế trái của phép gán với kiểm tra đầy đủ.

Quy trình bổ sung:

- eat TK_IDENT lấy tên identifier
- Gọi checkDeclaredIdent() để tìm Object trong symbol table

- Kiểm tra kind phải là một trong ba loại hợp lệ cho lvalue:
  + OBJ_VARIABLE: Biến thông thường có thể gán
  + OBJ_FUNCTION: Tên function trong thân function (để gán giá trị trả về)
  + OBJ_PARAMETER: Tham số có thể được gán giá trị mới
- Nếu kind không thuộc ba loại này, báo lỗi ERR_INVALID_LVALUE
- compileIndexes() xử lý các chỉ số mảng nếu có

Luồng hoạt động: eat TK_IDENT → checkDeclaredIdent() → kiểm tra kind (3 loại) → compileIndexes()

14. Biên dịch vòng lặp FOR

Hàm compileForSt():

- Xử lý vòng lặp FOR với kiểm tra biến điều khiển.

Quy trình đã được bổ sung:

- eat keyword FOR
- eat TK_IDENT lấy tên biến điều khiển
- Gọi checkDeclaredVariable() để xác minh biến điều khiển (đây là bổ sung mới)
- eat SB_ASSIGN
- compileExpression() cho giá trị bắt đầu
- eat keyword TO
- compileExpression() cho giá trị kết thúc
- eat keyword DO
- compileStatement() cho thân vòng lặp

Cơ chế checkDeclaredVariable():

- Thực hiện kiểm tra hai tầng tương tự checkDeclaredProcedure()

III. <u>Kết quả thực hiện</u>

1. Kết quả với các ví dụ trong thư mục tests

a. example1.kpl

parser.c example1.kpl

tests > example1.kpl

```
1  PROGRAM EXAMPLE1;
2  BEGIN
3  END. (* Example 1 *)
```

PROBLEMS 220 OUTPUT DEBUG CONSOLE TERMINAL PORTS

TERMINAL

```
(base) PS C:\Users\lehai\Documents\Github\IT3323_BT_Week3> ./symtab .\tests\example1.kpl
Program EXAMPLE1
(base) PS C:\Users\lehai\Documents\Github\IT3323_BT_Week3>
```

Phân tích: Chương trình đơn giản nhất, không có khai báo nào. Symbol table chỉ chứa đối tượng program.

b. example2.kpl

parser.c example2.kpl

tests > example2.kpl

```
1  PROGRAM EXAMPLE2; (* Factorial *)
2
3  VAR N : INTEGER;
4
5  FUNCTION F(N : INTEGER) : INTEGER;
6    BEGIN
7      IF N = 0 THEN F := 1 ELSE F := N * F (N - 1);
8    END;
9
10 BEGIN
11   FOR N := 1 TO 7 DO
12     BEGIN
13       CALL WRITELN;
14       CALL WRITEI(F(N));
15     END;
16 END. (* Factorial *)
```

PROBLEMS 220 OUTPUT DEBUG CONSOLE TERMINAL PORTS

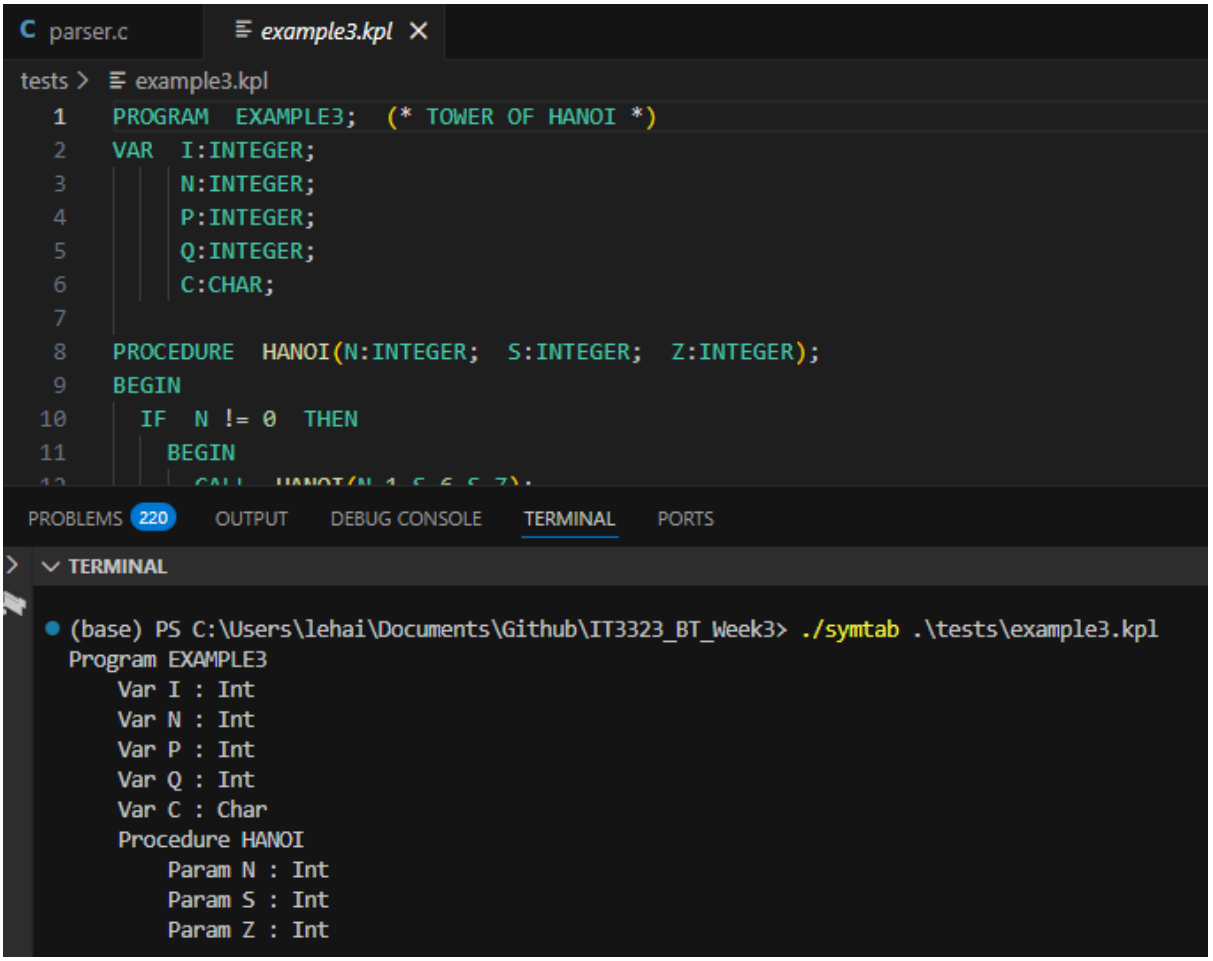
TERMINAL

```
(base) PS C:\Users\lehai\Documents\Github\IT3323_BT_Week3> ./symtab .\tests\example2.kpl
Program EXAMPLE2
    Var N : Int
    Function F : Int
        Param N : Int
```

Phân tích:

- Biến toàn cục N được khai báo
- Function F có tham số N (PARAM_VALUE) và trả về INT
- Tham số N trong F che khuất biến toàn cục N (shadowing)

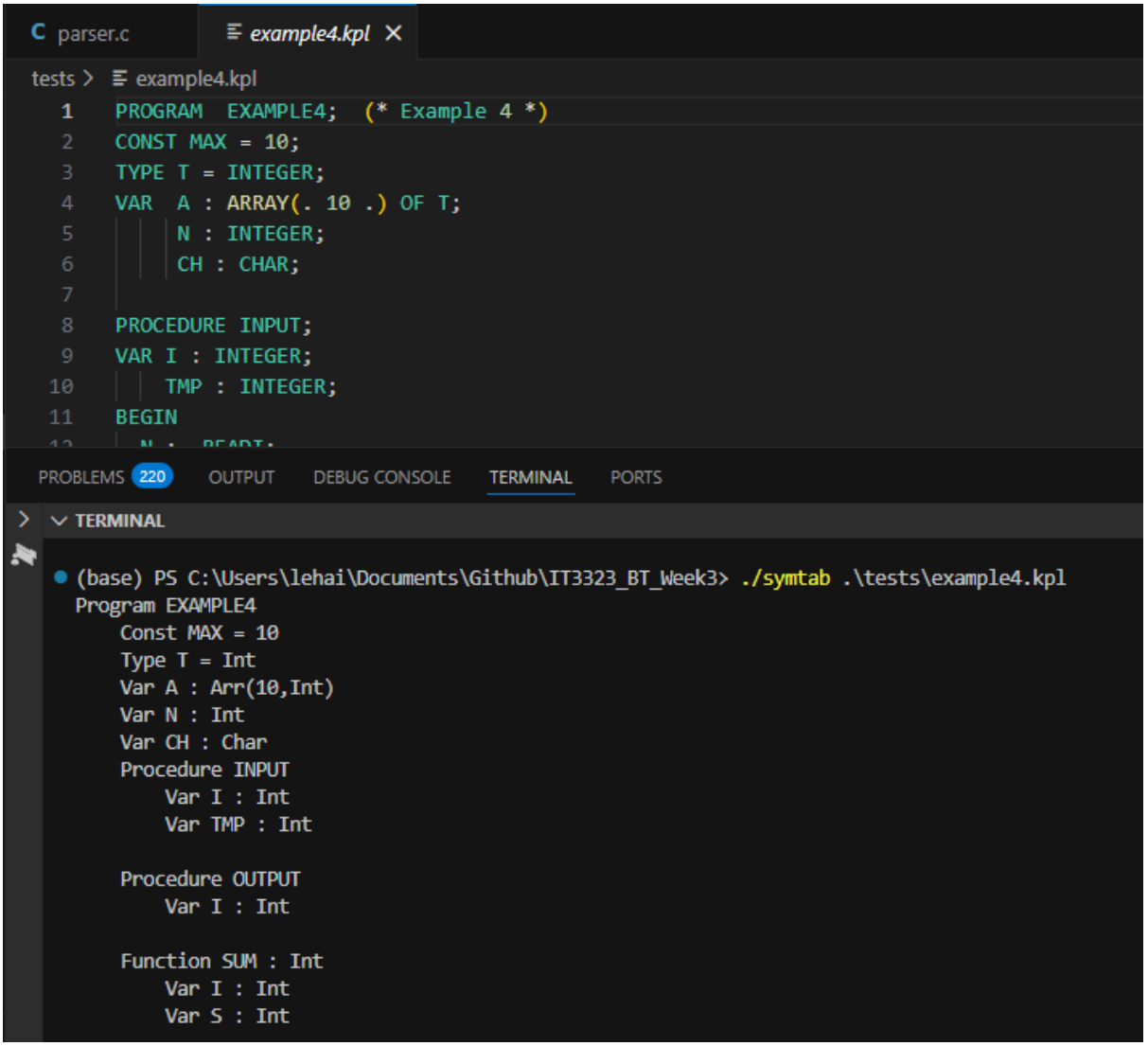
c. example3.kpl



Phân tích:

- Nhiều biến toàn cục với kiểu INT và CHAR
- Procedure HANOI nhận 3 tham số INT (PARAM_VALUE)
- Tham số N che khuất biến toàn cục N

d. example4.kpl



Phân tích:

- Hằng MAX = 10
- Kiểu T là alias của INTEGER
- Biến mảng A với kích thước 10
- Ba procedure/function với biến local riêng
- Biến I xuất hiện trong nhiều scope khác nhau (không xung đột)

e. example5.kpl

parser.c | example5.kpl

tests > example5.kpl

```
1   PROGRAM EXAMPLE5; (* Example 5 *)
2   CONST C = 1;
3   TYPE T = CHAR;
4   FUNCTION F(I : INTEGER):CHAR;
5   CONST B = C;
6   TYPE A = ARRAY(.5.) OF T;
7   BEGIN
8   END;
9   BEGIN
10  END. (* Example 5 *)
```

PROBLEMS 220 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

TERMINAL

```
(base) PS C:\Users\lehai\Documents\Github\IT3323_BT_Week3> ./symtab .\tests\example5.kpl
Program EXAMPLE5
    Const C = 1
    Type T = Char
    Function F : Char
        Param I : Int
        Const B = 1
        Type A = Arr(5,Char)
```

Phân tích:

- Hằng và kiểu toàn cục
- Function F có khai báo local (hằng B và kiểu A)
- Kiểu A là mảng 5 phần tử CHAR

f. example6.kpl

parser.c example6.kpl

tests > example6.kpl

```
1   PROGRAM EXAMPLE6;
2      CONST C1 = 10;
3          C2 = 'a';
4      TYPE T1 = ARRAY(. 10 .) OF INTEGER;
5      VAR V1 : INTEGER;
6          V2 : ARRAY(. 10 .) OF T1;
7
8      FUNCTION F(P1 : INTEGER; VAR P2 : CHAR) : INTEGER;
9        BEGIN
```

PROBLEMS 220 OUTPUT DEBUG CONSOLE TERMINAL PORTS

TERMINAL

```
(base) PS C:\Users\lehai\Documents\Github\IT3323_BT_Week3> ./symtab .\tests\example6.kpl
Program EXAMPLE6
    Const C1 = 10
    Const C2 = 'a'
    Type T1 = Arr(10,Int)
    Var V1 : Int
    Var V2 : Arr(10,Arr(10,Int))
    Function F : Int
        Param P1 : Int
        Param VAR P2 : Char

    Procedure P
        Param V1 : Int
        Const C1 = 'a'
        Const C3 = 10
        Type T1 = Int
        Type T2 = Arr(10,Int)
        Var V2 : Arr(10,Int)
        Var V3 : Char
```

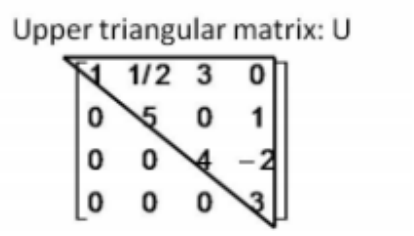
Phân tích:

- Mảng 2 chiều V2: `ARRAY [10] OF ARRAY [10] OF INTEGER`
- Function F có tham số P2 truyền tham chiếu (VAR)
- Procedure P có shadowing: C1, T1, V1, V2 che khuất khai báo toàn cục
- Thể hiện rõ cơ chế scope lồng nhau

2. Kiểm tra với bài tập lớp lý thuyết

Bài 1. Một ma trận vuông là ma trận tam giác trên nếu mọi phần tử nằm dưới đường chéo chính là bằng 0. Viết chương trình trên ngôn ngữ KPL để nhập một ma trận vuông kích thước nxn, n nhập từ bàn phím. In ra 1 nếu ma trận là tam giác trên , 0 nếu ngược lại.

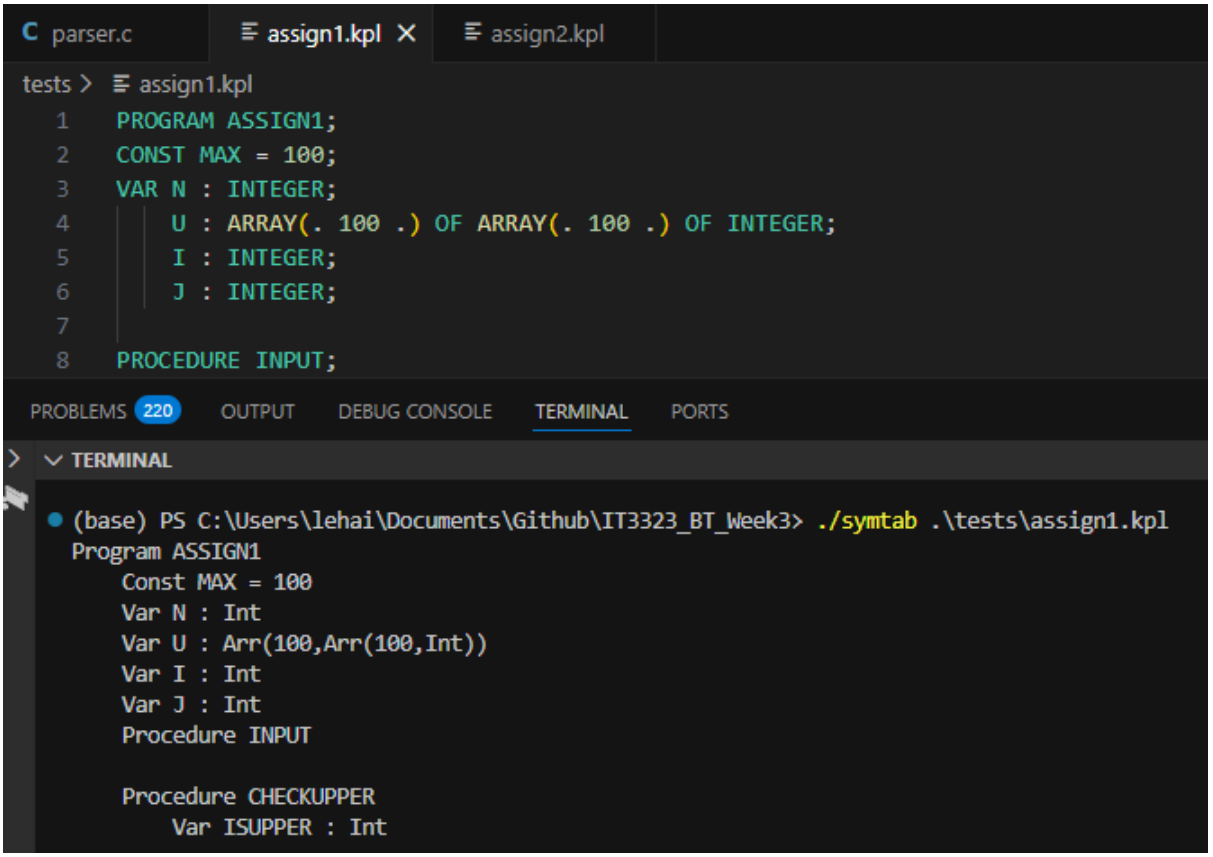
Ví dụ một ma trận tam giác trên:



parser.c | assign1.kpl | assign2.kpl

tests > assign1.kpl

```
PROGRAM ASSIGN1;
CONST MAX = 100;
VAR N : INTEGER;
    U : ARRAY(. 100 .) OF ARRAY(. 100 .) OF INTEGER;
    I : INTEGER;
    J : INTEGER;

PROCEDURE INPUT;
BEGIN
  CALL WRITEC('N');
  CALL WRITEC('=');
  N := READI;

  FOR I := 1 TO N DO
    FOR J := 1 TO N DO
      BEGIN
        U(.I.)(.J.) := READI;
      END
END;

PROCEDURE CHECKUPPER;
VAR ISUPPER : INTEGER;
BEGIN
  ISUPPER := 1;

  FOR I := 1 TO N DO
    FOR J := 1 TO N DO
      IF I > J THEN
        IF U(.I.)(.J.) != 0 THEN
          ISUPPER := 0;

  IF ISUPPER = 1 THEN
    CALL WRITEI(1)
  ELSE
    CALL WRITEI(0);
  CALL WRITELN
END;

BEGIN
  CALL INPUT;
  CALL CHECKUPPER
END.
```



Phân tích:

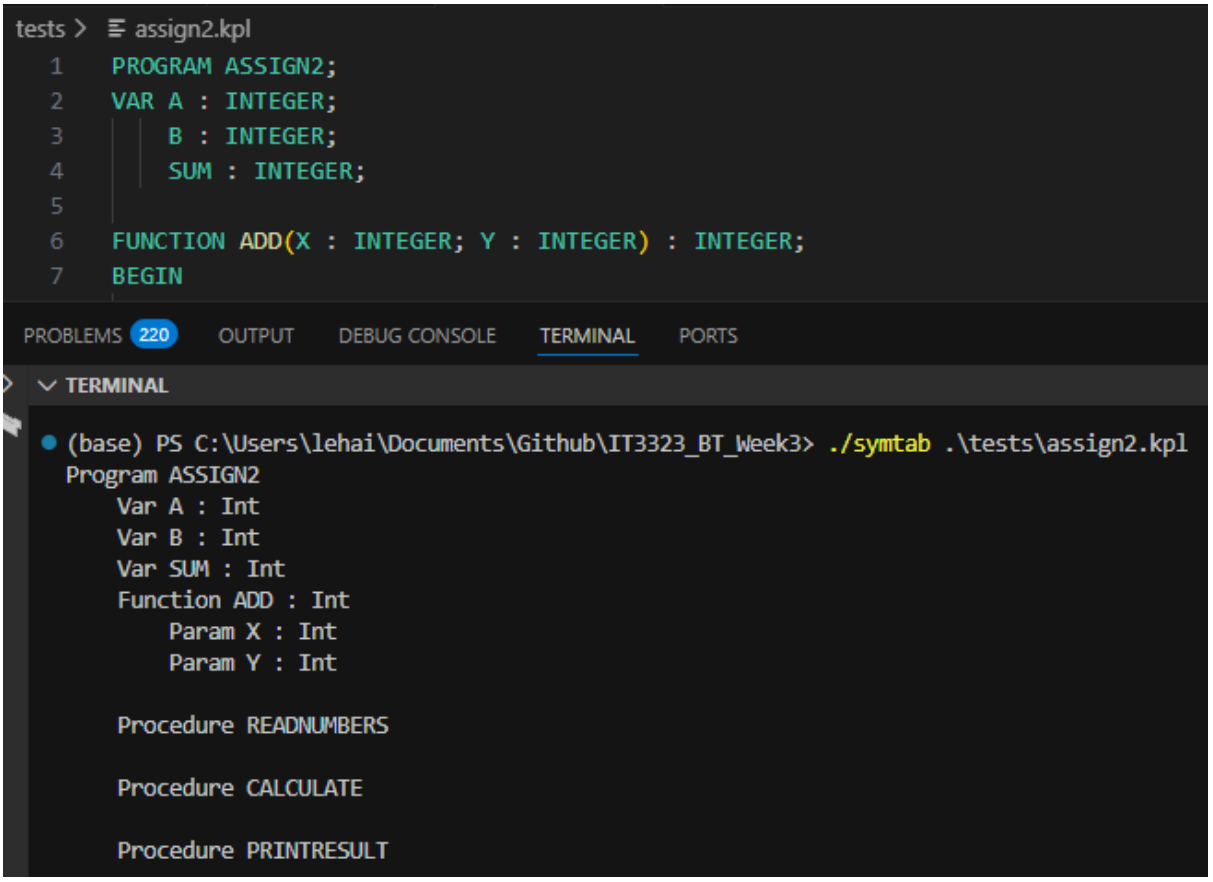
- Các đối tượng toàn cục (MAX, N, U, I, J) được đăng ký trong phạm vi chương trình chính
- 2 thủ tục (INPUT, CHECKUPPER) được khai báo với phạm vi riêng
- Biến cục bộ ISUPPER chỉ tồn tại trong phạm vi thủ tục CHECKUPPER
- Cấu trúc phân cấp thể hiện đúng mối quan hệ scope (phạm vi) giữa các đối tượng

Bài 2

Viết chương trình tính và in ra tổng 2 số bằng ngôn ngữ KPL. Chỉ ra phân tích trái của chương trình (dãy số hiệu sản xuất được dùng trong suy dẫn trái)

assign2.kpl

```
PROGRAM ASSIGN2;
VAR A : INTEGER;
    B : INTEGER;
    SUM : INTEGER;

FUNCTION ADD(X : INTEGER; Y : INTEGER) : INTEGER;
BEGIN
  ADD := X + Y
END;

PROCEDURE READNUMBERS;
BEGIN
  CALL WRITEC('A');
  CALL WRITEC('=');
  A := READI;
  CALL WRITELN;

  CALL WRITEC('B');
  CALL WRITEC('=');
  B := READI;
  CALL WRITELN
END;

PROCEDURE CALCULATE;
BEGIN
  SUM := ADD(A, B)
END;

PROCEDURE PRINTRESULT;
BEGIN
  CALL WRITEC('S');
  CALL WRITEC('U');
  CALL WRITEC('M');
  CALL WRITEC('=');
  CALL WRITEI(SUM);
  CALL WRITELN
END;

BEGIN
  CALL READNUMBERS;
  CALL CALCULATE;
  CALL PRINTRESULT
END.
```



Phân tích:

- 3 biến toàn cục (A, B, SUM) được khai báo ở mức chương trình, có thể truy cập từ mọi nơi
- Hàm ADD có phạm vi riêng với 2 tham số X, Y (tham trị), nhận vào 2 số nguyên và trả về tổng của chúng
- 3 thủ tục (READNUMBERS, CALCULATE, PRINTRESULT) được khai báo nhưng không có tham số và biến cục bộ, chúng sử dụng biến toàn cục