

THỰC HÀNH XÂY DỰNG CHƯƠNG TRÌNH DỊCH

Báo cáo bài 3: Tạo bảng ký hiệu

I. Sinh viên thực hiện

Vũ Hải Nam – 20225750

II. Kết quả thực hiện

1. Thư mục SymTab_Incompleted

Kết quả thực hiện sửa đổi và bổ sung các câu lệnh có đánh dấu là `//TODO` bên trong hàm `parser.c` đã hoàn tất.

2. Các dòng lệnh đã sửa đổi

Hàm `compileProgram()`:

```
void compileProgram(void) { //ok
    // TODO: create, enter, and exit program block
    Object* obj;
    eat(KW_PROGRAM);
    eat(TK_IDENT);
    obj = createProgramObject(currentToken->string);
    declareObject(obj);
    enterBlock(obj->progAttrs->scope);
    eat(SB_SEMICOLON);
    compileBlock();
    exitBlock();
    eat(SB_PERIOD);
}
```

Hàm `compileBlock()`:

```
void compileBlock(void) { //ok
    // TODO: create and declare constant objects
    Object* obj;
    if (lookAhead->tokenType == KW_CONST) {
        eat(KW_CONST);

        do {
            eat(TK_IDENT);
            obj = createConstantObject(currentToken->string);
            eat(SB_EQ);
            obj->constAttrs->value = compileConstant();
            declareObject(obj);
            eat(SB_SEMICOLON);
        } while (lookAhead->tokenType == TK_IDENT);
        compileBlock2();
    }
    else compileBlock2();
}
```

Hàm compileBlock2():

```
void compileBlock2(void) { //ok
    // TODO: create and declare type objects
    Object* obj;
    if (lookAhead->tokenType == KW_TYPE) {
        eat(KW_TYPE);

        do {
            eat(TK_IDENT);
            obj = createTypeObject(currentToken->string);
            eat(SB_EQ);
            obj->typeAttrs->actualType = compileType();
            declareObject(obj);
            eat(SB_SEMICOLON);
        } while (lookAhead->tokenType == TK_IDENT);

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

Hàm compileBlock3():

```
void compileBlock3(void) { //ok
    // TODO: create and declare variable objects
    Object* obj;
    if (lookAhead->tokenType == KW_VAR) {
        eat(KW_VAR);

        do {
            eat(TK_IDENT);
            obj = createVariableObject(currentToken->string);
            eat(SB_COLON);
            obj->varAttrs->type = compileType();
            declareObject(obj);
            eat(SB_SEMICOLON);
        } while (lookAhead->tokenType == TK_IDENT);

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

Hàm compileFuncDecl():

```
void compileFuncDecl(void) { //ok
    // TODO: create and declare a function object
    Object* obj;
    Type* returnType;
    eat(KW_FUNCTION);
    eat(TK_IDENT);
    obj = createFunctionObject(currentToken->string);
    declareObject(obj);
    enterBlock(obj->funcAttrs->scope);
    compileParams();
    eat(SB_COLON);
    returnType = compileBasicType();
    obj->funcAttrs->returnType = returnType;
    eat(SB_SEMICOLON);
    compileBlock();
    eat(SB_SEMICOLON);
    exitBlock();
}
```

Hàm compileProcDecl():

```
void compileProcDecl(void) { //ok
    // TODO: create and declare a procedure object
    Object* obj;
    eat(KW_PROCEDURE);
    eat(TK_IDENT);
    obj = createProcedureObject(currentToken->string);
    declareObject(obj);
    enterBlock(obj->procAttrs->scope);
    compileParams();
    eat(SB_SEMICOLON);
    compileBlock();
    eat(SB_SEMICOLON);
    exitBlock();
}
```

Hàm compileUnsignedConstant():

```
ConstantValue* compileUnsignedConstant(void) { //ok
    // TODO: create and return an unsigned constant value
    ConstantValue* constValue;
    Object* obj;

    switch (lookAhead->tokenType) {
    case TK_NUMBER:
        eat(TK_NUMBER);
        constValue = makeIntConstant(currentToken->value);
        break;
    case TK_IDENT:
        eat(TK_IDENT);
        obj = lookupObject(currentToken->string);
        if((obj != NULL)&&(obj->kind == OBJ_CONSTANT))
            constValue = duplicateConstantValue(obj->constAttrs->value);
        else
            error(ERR_UNDECLARED_CONSTANT, currentToken->lineNo, currentToken->colNo);
        break;
    case TK_CHAR:
        eat(TK_CHAR);
        constValue = makeCharConstant(currentToken->string[0]);
        break;
    default:
        error(ERR_INVALID_CONSTANT, lookAhead->lineNo, lookAhead->colNo);
        break;
    }
    return constValue;
}
```

Hàm compileConstant():

```
ConstantValue* compileConstant(void) { // ok
    // TODO: create and return a constant
    ConstantValue* constValue;

    switch (lookAhead->tokenType) {
    case SB_PLUS:
        eat(SB_PLUS);
        constValue = compileConstant2();
        if (constValue->type != TP_INT)
            error(ERR_TYPE_INCONSISTENCY, lookAhead->lineNo, lookAhead->colNo);
        break;
    case SB_MINUS:
        eat(SB_MINUS);
        constValue = compileConstant2();
        if (constValue->type != TP_INT)
            error(ERR_TYPE_INCONSISTENCY, lookAhead->lineNo, lookAhead->colNo);
        constValue->intValue = -constValue->intValue;
        break;
    case TK_CHAR:
        eat(TK_CHAR);
        constValue = makeCharConstant(currentToken->string[0]);
        break;
    default:
        constValue = compileConstant2();
        break;
    }
    return constValue;
}
```

Hàm compileConstant2():

```
ConstantValue* compileConstant2(void) { //ok
    // TODO: create and return a constant value
    ConstantValue* constValue;
    Object* obj;
    switch (lookAhead->tokenType) {
    case TK_NUMBER:
        eat(TK_NUMBER);
        constValue = makeIntConstant(currentToken->value);
        break;
    case TK_IDENT:
        eat(TK_IDENT);
        obj = lookupObject(currentToken->string);
        if (obj != NULL && obj->kind == OBJ_CONSTANT)
            constValue = duplicateConstantValue(obj->constAttrs->value);
        else
            error(ERR_UNDECLARED_CONSTANT, lookAhead->lineNo, lookAhead->colNo);
        break;
    default:
        error(ERR_INVALID_CONSTANT, lookAhead->lineNo, lookAhead->colNo);
        break;
    }
    return constValue;
}
```

Hàm compileType():

```
Type* compileType(void) { //ok
    // TODO: create and return a type
    Type* type;
    Object* obj;

    switch (lookAhead->tokenType) {
    case KW_INTEGER:
        eat(KW_INTEGER);
        type = makeIntType();
        break;
    case KW_CHAR:
        eat(KW_CHAR);
        type = makeCharType();
        break;
    case KW_ARRAY:
        eat(KW_ARRAY);
        eat(SB_LSEL);
        eat(TK_NUMBER);
        int arraySize = currentToken->value;
        eat(SB_RSEL);
        eat(KW_OF);
        type = makeArrayType(arraySize, compileType());
        break;
    case TK_IDENT:
        eat(TK_IDENT);
        obj = lookupObject(currentToken->string);
        if(obj!=NULL && obj->kind==OBJ_TYPE)
            type = duplicateType(obj->typeAttrs->actualType);
        else
            error(ERR_INVALID_TYPE, lookAhead->lineNo, lookAhead->colNo);
        break;
    default:
        error(ERR_INVALID_TYPE, lookAhead->lineNo, lookAhead->colNo);
        break;
    }
    return type;
}
```

Hàm compileBasicType():

```
Type* compileBasicType(void) { // ok
    // TODO: create and return a basic type
    Type* type;

    switch (lookAhead->tokenType) {
    case KW_INTEGER:
        eat(KW_INTEGER);
        type = makeIntType();
        break;
    case KW_CHAR:
        eat(KW_CHAR);
        type = makeCharType();
        break;
    default:
        error(ERR_INVALID_BASICTYPE, lookAhead->lineNo, lookAhead->colNo);
        break;
    }
    return type;
}
```

Hàm compileParam():

```
void compileParam(void) { //ok
    // TODO: create and declare a parameter
    Object* obj;
    switch (lookAhead->tokenType) {
    case TK_IDENT:
        eat(TK_IDENT);
        obj = createParameterObject(currentToken->string, PARAM_VALUE, symtab->currentScope->owner);
        declareObject(obj);
        eat(SB_COLON);
        obj->paramAttrs->type=compileBasicType();
        break;
    case KW_VAR:
        eat(KW_VAR);
        eat(TK_IDENT);
        obj = createParameterObject(currentToken->string, PARAM_REFERENCE, symtab->currentScope->owner);
        declareObject(obj);
        eat(SB_COLON);
        obj->paramAttrs->type=compileBasicType();
        break;
    default:
        error(ERR_INVALID_PARAMETER, lookAhead->lineNo, lookAhead->colNo);
        break;
    }
}
```

3. Kết quả chạy chương trình

Biên dịch chương trình:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> gcc main.c parser.c scanner.c reader.c charcode.c token.c error.c symtab.c debug.c -o kpl
```

Kết quả thực hiện chương trình với example1.kpl trong folder tests:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kpl tests/example1.kpl
Program EXAMPLE1
```

Giải thích kết quả:

1.Parser đọc "PROGRAM EXAMPLE1":

-Hàm compileProgram() được gọi, nó đọc từ khóa PROGRAM và định danh EXAMPLE1; tiếp đó sẽ gọi hàm createProgramObject("EXAMPLE1") để tạo ra đối tượng Object với thuộc tính kind = OBJ_PROGRAM và name = "EXAMPLE1", được gán trong symtab->program.

2.Parser đọc phần thân:

-Hàm compileBlock không tạo tên đối tượng con vào danh sách do không gặp các từ khóa như CONST, VAR, TYPE,... cho nên scope của chương trình là rỗng.

3.Kết quả hiển thị:

Hàm printObject(symtab->program,0); sẽ đọc thông tin từ symtab->program; dựa trên thuộc tính của đối tượng đó là kind là Program và name là EXAMPLE1 nên in ra dòng Program EXAMPLE1

Kết quả thực hiện chương trình với example2.kpl trong folder tests:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleated\SymTab_Incompleated> ./kpl tests/example2.kpl
Program EXAMPLE2
  Var N : Int
  Function F : Int
    Param N : Int
```

Giải thích kết quả:

1.Parser đọc từ khóa PROGRAM và EXAMPLE2 và đăng kí đối tượng đó, đối tượng được gán vào symtab->program.

2.Bên trong chương trình có hai khai báo là VAR N:INTEGER -> khai báo biến N và FUNCTION F():INTEGER -> khai báo hàm tên là F; vì vậy hai hàm này được in cùng cấp và thụt 1 cấp so với PROGRAM

3.Trong hàm F có định nghĩa tham số đầu vào (N:INTEGER), vì vậy ở đây ta sẽ in kết quả thụt 1 cấp so với hàm F ở trên

Mặc dù trong chương trình này xuất hiện tận 2 tham số N, tuy nhiên biến N ở Var N lại là biến toàn cục, còn ở Param N là biến cục bộ.

Kết quả thực hiện chương trình với example3.kpl trong folder tests:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kpl tests/example3.kpl
Program EXAMPLE3
  Var I : Int
  Var N : Int
  Var P : Int
  Var Q : Int
  Var C : Char
  Procedure HANOI
    Param N : Int
    Param S : Int
    Param Z : Int
```

Giải thích kết quả:

1. Parser đọc từ khóa PROGRAM và EXAMPLE3 và đăng kí đối tượng đó, đối tượng được gán vào symtab->program. In ra Program EXAMPLE3
2. Các dòng khai báo Var và thủ tục Procedure HANOI do thuộc phạm vi quản lý trực tiếp của chương trình chính cho nên kết quả in ra các dòng thụt lại 1 cấp so với Program EXAMPLE3
3. Trong thủ tục Procedure, có các Param chỉ tồn tại trong thủ tục này, vì vậy các dòng kết quả của Param sẽ lùi 1 cấp so với Procedure.

Ngoài ra các lệnh logic như FOR, IF, CALL WRITELN là các hành động thực thi không phải khai báo biến mới cho nên không xuất hiện trong SymTab và không được in ra.

Kết quả thực hiện chương trình với example4.kpl trong folder tests:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kpl tests/example4.kpl
Program EXAMPLE4
  Const MAX = 10
  Type T = Int
  Var A : Arr(10,Int)
  Var N : Int
  Var CH : Char
  Procedure INPUT
    Var I : Int
    Var TMP : Int

  Procedure OUTPUT
    Var I : Int

  Function SUM : Int
    Var I : Int
    Var S : Int
```

Giải thích kết quả:

1. Parser đọc từ khóa PROGRAM và EXAMPLE4 và đăng kí đối tượng đó, đối tượng được gán vào symtab->program. In ra Program EXAMPLE4
- 2.Với các khai báo toàn cục: Const MAX, Type T, VAR thuộc về chương trình EXAMPLE4 vì vậy sẽ in thụt 1 cấp so với Program
- 3.Trong hàm thủ tục Procedure INPUT có các biến cục bộ I, TMP; thủ tục Procedure OUTPUT có biến cục bộ I; hàm Function SUM có biến cục bộ I,S; vì các biến cục bộ nằm trong thân hàm, thủ tục bên nó sẽ được in ra lùi 2 cấp so với Program

Kết quả thực hiện chương trình với example5.kpl trong folder tests:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kpl 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)
```

Giải thích kết quả:

1. Parser nhận diện Program và tên chương trình là EXAMPLE5. Đây là scope cha chứa tất cả khai báo còn lại
2. Parser tiếp tục ghi nhận các khai báo Const C, Type T, Function F nằm trong phạm vi trực tiếp của chương trình EXAMPLE5, vì vậy kết quả in ra sẽ thụt 1 cấp so với Program EXAMPLE5; chứng tỏ parser hiểu các khai báo đó nằm trong phạm vi của chương trình chính
3. Trong Function F có các khai báo cục bộ Param I, Cost B, Type A; do vậy kết quả in ra sẽ phải thụt 2 cấp so với Program EXAMPLE5
 - Với Const B, trình biên dịch đã thực hiện việc tìm định danh C, thấy C là hằng toàn cục có giá trị 1 do đó gán giá trị trực tiếp cho B, kết quả hiển thị ở đây là Const B = 1
 - Với Param I, I ở đây được xác định là tham số chứ không phải biến thường
 - Với Type A = ARRAY(.5.) OF T; ta có mảng, parser đã tìm định danh T và thấy T là tên gọi được gán kiểu Char, do đó thay T bằng Char; kích thước '5.' được phân tích thành số nguyên 5; và cuối cùng hiển thị kết quả in ra là Arr(5,Char)

Kết quả thực hiện chương trình với example6.kpl trong folder tests:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kpl 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
```

Giải thích kết quả:

1. Parser nhận diện Program và tên chương trình là EXAMPLE6. Đây là scope cha chứa tất cả khai báo còn lại.
2. Trong phạm vi chương trình EXAMPLE6 có các khai báo về Const C1, C2; Type T1; Var V1, V2, Function F; Procedure P; các khai báo này nằm trực tiếp trong phạm vi chương trình nên khi hiển thị kết quả in ra sẽ thuộc 1 cấp so với Program.
3. Trong các phạm vi hàm F và thủ tục P có các thuộc tính nằm trong phạm vi thân hàm, vì vậy có thể thấy kết quả in ra sẽ thuộc 2 cấp so với Program.

Kết quả thực hiện chương trình với example3.kpl nằm cùng cấp với file main.c:

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kp1 example3.kpl
Program EXAMPLE3
  Var I : Int
  Var N : Int
  Var P : Int
  Var Q : Int
  Var C : Char
  Procedure HANOI
    Param N : Int
    Param S : Int
    Param Z : Int
```

Giải thích kết quả:

1. Parser nhận diện Program và tên chương trình là EXAMPLE6. Đây là scope cha chứa tất cả khai báo còn lại.
2. Trong phạm vi chương trình chính có các khai báo các biến Var I,N,P,Q,C và thủ tục procedure nằm trực tiếp trong thân EXAMPLE6; vì vậy kết quả in ra sẽ thụt 1 đầu so với Program.
3. Trong thủ tục Procedure có các tham số cục bộ nằm trong phạm vi của nó vì vậy sẽ in ra kết quả thụt 2 cấp so với Program.

Chương trình KPL ở tiết lý thuyết:

Bài 1:

```
PROGRAM Bai1;
CONST MAX = 50;
TYPE ROW = ARRAY(. 50 .) OF INTEGER;
      MATRIX = ARRAY(. 50 .) OF ROW;

VAR N : INTEGER;
    A : MATRIX;
    I : INTEGER;
    J : INTEGER;
    RESULT : INTEGER;

BEGIN
  CALL ReadI(N);

  FOR I := 1 TO N DO
    FOR J := 1 TO N DO
      CALL ReadI(A(.I.)(.J.));

    RESULT := 1;

  FOR I := 1 TO N DO
    BEGIN
      FOR J := 1 TO N DO
        BEGIN
          IF I > J THEN
            BEGIN
              IF A(.I.)(.J.) != 0 THEN
                RESULT := 0;
            END;
          END;
        END;
      END;
    END;

  CALL WriteI(RESULT);
END.
```

Kết quả chạy chương trình:

```
Program Bai1
Const MAX = 50
Type ROW = Arr(50,Int)
Type MATRIX = Arr(50,Arr(50,Int))
Var N : Int
Var A : Arr(50,Arr(50,Int))
Var I : Int
Var J : Int
Var RESULT : Int
```

Giải thích kết quả:

Kết quả trên là Bảng ký hiệu xác nhận trình biên dịch đã hiểu đúng cấu trúc dữ liệu của chương trình Bai1. Hằng số MAX được ghi nhận là 50, làm cơ sở để định nghĩa kiểu ROW và đặc biệt là kiểu MATRIX được phân giải thành mảng 2 chiều lồng nhau. Biến A nhờ đó mang cấu trúc ma trận 50x50 chính xác, còn các biến N, I, J, RESULT được xác định là số nguyên đơn giản để phục vụ việc tính toán và duyệt vòng lặp.

Bài 2:

```
PROGRAM Sum2Num;  
VAR A : INTEGER;  
    B : INTEGER;  
    S : INTEGER;  
  
BEGIN  
    CALL ReadI(A);  
    CALL ReadI(B);  
    S := A + B;  
    CALL WriteI(S);  
END.
```

```
PS C:\Users\Admin\OneDrive\Máy tính\ctd\SymTab_Incompleted\SymTab_Incompleted> ./kpl Bai2.kpl  
Program Sum2Num  
  Var A : Int  
  Var B : Int  
  Var S : Int
```

Giải thích kết quả:

Kết quả này hiển thị Bảng ký hiệu của chương trình Sum2Num, xác nhận trình biên dịch đã ghi nhận chính xác 3 biến toàn cục cần thiết. Các biến A, B và S đều được định nghĩa thành công với kiểu số nguyên, đảm bảo bộ nhớ đã được cấp phát đúng để thực hiện phép tính cộng trong thân chương trình.