ĐẠI HỌC BÁCH KHOA HÀ NỘI
TRƯỜNG CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
***** *****

BÀI THỰC HÀNH SỐ 4: QUẢN LÝ PHẠM VI
HỌC PHẦN: THỰC HÀNH XÂY DỰNG CHƯƠNG TRÌNH DỊCH

Mã lớp học: 161629
GVHD: Nguyễn Thị Thu Hương
TA: Đỗ Gia Huy

- Hà Nội, tháng 12 năm 2025 -

Mục lục

1 Nội dung chính về quản lý phạm vi

1.1 Khái niệm về phạm vi

Scope (phạm vi) là vùng trong chương trình mà một identifier (tên biến, hằng, kiểu, hàm, thủ tục) có thể được truy cập. Trong compiler, quản lý phạm vi đóng vai trò quan trọng trong việc đảm bảo tính đúng đắn của chương trình. Cơ chế này cho phép tránh xung đột tên bằng cách cho phép cùng một tên được sử dụng ở các scope khác nhau, đồng thời kiểm tra tính hợp lệ của identifier bằng cách xác định xem identifier có được khai báo trước khi sử dụng hay không. Ngoài ra, hệ thống hỗ trợ scope lồng nhau, cho phép scope con truy cập identifier của scope cha, trong khi scope cha không thể truy cập identifier của scope con.

Ví dụ minh họa: Trong một function, ta có thể khai báo biến cùng tên với biến ở scope ngoài, và biến trong function sẽ che (shadow) biến ngoài, tức là khi sử dụng tên đó trong function, nó sẽ tham chiếu đến biến trong function chứ không phải biến ở scope ngoài.

1.2 Cấu trúc dữ liệu quản lý phạm vi

1.2.1 Cấu trúc Scope

Cấu trúc Scope được định nghĩa như sau:

```
struct Scope_ {
  ObjectNode *objList;
  Object *owner;
  struct Scope_ *outer;
};
```

Trường `objList` chứa tất cả các identifier được khai báo trong scope hiện tại, bao gồm constants, types, variables, functions và procedures. Trường `owner` xác định scope này thuộc về đối tượng nào, ví dụ scope của function F thuộc về object F. Trường `outer` tạo chuỗi scope, cho phép tìm kiếm identifier từ scope hiện tại lên các scope ngoài thông qua cơ chế scope chain.

1.2.2 Cấu trúc Symbol Table (SymTab)

Cấu trúc Symbol Table được định nghĩa như sau:

```
struct SymTab_ {
  Object* program;
  Scope* currentScope;
  ObjectNode *globalObjectList;
};
```

Trường `program` là object chương trình chính, có scope riêng của nó. Trường `currentScope` là scope đang active, được cập nhật động khi vào hoặc ra khỏi một block. Trường `globalObjectList` chứa các hàm built-in như READC, READI, WRITEI, WRITEC, WRITELN, những hàm này có thể được gọi từ bất kỳ đâu trong chương trình.

1.3 Cơ chế quản lý phạm vi

1.3.1 Tạo scope mới khi vào block/procedure/function

Khi compiler xử lý một cấu trúc mới như program, function hoặc procedure, một scope mới sẽ được tạo ra. Quá trình này được thực hiện thông qua hàm `createScope`, nhận vào owner (object sở hữu scope) và outer (scope bên ngoài) để tạo một scope mới và liên kết nó với scope ngoài.

Khi vào Program, compiler tạo program object và vào scope của nó:

```
void compileProgram(void) {
  program = createProgramObject(currentToken->string);
  enterBlock(program->progAttrs->scope);
  // ...
}
```

Khi vào Function, compiler tạo function object, khai báo nó ở scope ngoài, sau đó vào scope của function để xử lý tham số và body:

```
void compileFuncDecl(void) {
  Object* funcObj;
  Type* returnType;
  eat(KW_FUNCTION);
  eat(TK_IDENT);
  checkFreshIdent(currentToken->string);
  funcObj = createFunctionObject(currentToken->string);
  declareObject(funcObj);
  enterBlock(funcObj->funcAttrs->scope);
  compileParams();
  eat(SB_COLON);
  returnType = compileBasicType();
  funcObj->funcAttrs->returnType = returnType;
  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_SEMICOLON);
  exitBlock();
}
```

Khi vào Procedure, quá trình tương tự như function, nhưng không có return type. Cơ chế tạo scope được thực hiện bởi hàm `createScope`:

```
Scope* createScope(Object* owner, Scope* outer) {
  Scope* scope = (Scope*)malloc(sizeof(Scope));
  scope->objList = NULL;
  scope->owner = owner;
  scope->outer = outer;
  return scope;
}
```

1.3.2 Thoát scope khi kết thúc block

Khi kết thúc một block, compiler cần thoát khỏi scope hiện tại và quay về scope ngoài. Điều này được thực hiện bởi hàm `exitBlock`:

```
void exitBlock(void) {
  symtab->currentScope = symtab->currentScope->outer;
}
```

Ví dụ về luồng xử lý: Khi vào Program scope, `currentScope` được đặt bằng `program->scope`. Khi vào Function F scope, `currentScope` được đặt bằng `F->scope` với `outer` trỏ đến `program->scope`. Khi thoát Function F, `currentScope` quay về `program->scope`. Cuối cùng, khi thoát Program, `currentScope` được đặt thành NULL.

1.3.3 Quan hệ scope lồng nhau qua outer

Scope được tổ chức dưới dạng chuỗi (chain) thông qua con trỏ `outer`. Cấu trúc này tạo ra một hệ thống phân cấp, trong đó mỗi scope có thể truy cập các identifier của scope ngoài thông qua con trỏ `outer`. Cấu trúc chuỗi scope có dạng như sau:

```
Global Objects
    ↑
Program Scope (outer = NULL)
    ↑
Function F Scope (outer = Program Scope)
    ↑
Nested Block Scope (outer = Function F Scope)
```

Khi tìm kiếm identifier, quá trình bắt đầu từ scope hiện tại. Nếu không tìm thấy, hệ thống sẽ tìm trong scope ngoài thông qua con trỏ `outer`, và tiếp tục quá trình này cho đến khi tìm thấy identifier hoặc đã duyệt hết tất cả các scope. Cuối cùng, nếu vẫn không tìm thấy, hệ thống sẽ tìm trong `globalObjectList` chứa các hàm built-in.

1.4 Tìm kiếm và kiểm tra identifier

1.4.1 Cách lookup từ scope hiện tại đến scope ngoài

Hàm `lookupObject` thực hiện việc tìm kiếm identifier trong toàn bộ scope chain. Quá trình này bắt đầu từ scope hiện tại và di chuyển dần ra các scope ngoài:

```
Object* lookupObject(char *name) {
  Scope* scope = symtab->currentScope;
  Object* obj;

  while (scope != NULL) {
    obj = findObject(scope->objList, name);
    if (obj != NULL) return obj;
    scope = scope->outer;
  }

  obj = findObject(symtab->globalObjectList, name);
  if (obj != NULL) return obj;

  return NULL;
}
```

Ví dụ minh họa: Trong chương trình có constant C được khai báo ở scope Program với giá trị 10, và trong function F có constant C được khai báo với giá trị 20. Khi lookup "C" trong body của function F, hệ thống sẽ tìm trong Scope F trước và tìm thấy C = 20, do đó trả về object này. Nếu không tìm thấy trong Scope F, hệ thống sẽ tìm trong Scope Program và tìm thấy C = 10.

```
PROGRAM EXAMPLE;
CONST C = 10;            // Scope Program
FUNCTION F: INTEGER;
  CONST C = 20;          // Scope Function F
  BEGIN
  END;
```

1.4.2 Kiểm tra trùng tên (checkFreshIdent)

Hàm `checkFreshIdent` đảm bảo rằng identifier chưa tồn tại trong scope hiện tại khi khai báo. Điều này ngăn chặn việc khai báo trùng tên trong cùng một scope:

```
void checkFreshIdent(char *name) {
  if (findObject(symtab->currentScope->objList, name) != NULL)
    error(ERR_DUPLICATE_IDENT, currentToken->lineNo, currentToken
       ->colNo);
}
```

Hàm này được sử dụng khi khai báo constant (ví dụ `CONST C = 10;`), khi khai báo type (ví dụ `TYPE T = INTEGER;`), khi khai báo variable (ví dụ `VAR X: INTEGER;`), và khi khai báo function hoặc procedure. Lưu ý quan trọng là hàm chỉ kiểm tra trong scope hiện tại, không kiểm tra scope ngoài, điều này cho phép shadowing - tức là identifier trong scope con có thể che identifier cùng tên ở scope ngoài.

1.4.3 Kiểm tra chưa khai báo

Hàm `checkDeclaredIdent` kiểm tra xem identifier đã được khai báo hay chưa, bất kể loại của nó. Hàm này sử dụng `lookupObject` để tìm kiếm trong toàn bộ scope chain:

```

Object* checkDeclaredIdent(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_IDENT, currentToken->lineNo, currentToken
       ->colNo);
  return obj;
}
```

Lưu ý: Hàm `checkDeclaredIdent` tồn tại trong code nhưng không được sử dụng trực tiếp. Thay vào đó, lỗi `ERR_UNDECLARED_IDENT` được xử lý bởi các hàm chuyên biệt như `checkDeclaredLValueIdent` trong các ngữ cảnh cụ thể.

Ngoài hàm tổng quát này, hệ thống còn có các hàm kiểm tra chuyên biệt cho từng loại identifier. Hàm `checkDeclaredConstant` kiểm tra xem identifier có phải là constant không, `checkDeclaredType` kiểm tra xem có phải là type không, `checkDeclaredVariable` kiểm tra xem có phải là variable không, `checkDeclaredProcedure` kiểm tra xem có phải

là procedure không, và `checkDeclaredLValueIdent` kiểm tra xem identifier có thể dùng làm l-value không (chỉ variable, parameter, hoặc function mới có thể là l-value).

1.4.4 Kiểm tra sai loại identifier

Sau khi xác định identifier đã được khai báo, hệ thống cần kiểm tra xem loại của identifier có phù hợp với ngữ cảnh sử dụng hay không. Ví dụ, hàm `checkDeclaredVariable` không chỉ kiểm tra identifier đã được khai báo, mà còn kiểm tra xem nó có phải là variable không:

```
Object* checkDeclaredVariable(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_VARIABLE, ...);
  if (obj->kind != OBJ_VARIABLE)
    error(ERR_INVALID_VARIABLE, ...);
  return obj;
}
```

Ví dụ về các trường hợp lỗi: Khi có constant C và variable X, việc gán `X := C` là hợp lệ vì C là constant có thể dùng trong expression. Tuy nhiên, việc gán `C := 5` sẽ gây lỗi `ERR_INVALID_LVALUE` vì C không phải variable nên không thể là l-value. Tương tự, nếu F là function, việc sử dụng F trong expression như `X := F` là hợp lệ, nhưng nếu cố gắng sử dụng F như một variable trong ngữ cảnh không phù hợp sẽ gây lỗi.

```
CONST C = 10;
VAR X: INTEGER;
BEGIN
  X := C;
  C := 5;
  X := F;
END;
```

1.5 Liên hệ với xử lý lỗi ngữ nghĩa

Các lỗi ngữ nghĩa liên quan đến scope được phát hiện trong quá trình lookup và kiểm tra identifier. Hệ thống phân loại các lỗi này thành ba nhóm chính: lỗi trùng tên, lỗi chưa khai báo, và lỗi sai loại identifier.

1.5.1 ERR_DUPLICATE_IDENT

Lỗi `ERR_DUPLICATE_IDENT` được phát hiện khi hàm `checkFreshIdent` tìm thấy identifier đã tồn tại trong scope hiện tại. Điều này xảy ra khi người lập trình cố gắng khai báo một identifier trùng tên với identifier đã được khai báo trước đó trong cùng một scope. Ví dụ, nếu đã khai báo `VAR X: INTEGER;` và sau đó lại khai báo `VAR X: CHAR;` trong cùng scope, hệ thống sẽ báo lỗi `ERR_DUPLICATE_IDENT`.

```
VAR X: INTEGER;
VAR X: CHAR;
```

Hàm `checkFreshIdent` thực hiện kiểm tra này bằng cách tìm kiếm identifier trong danh sách object của scope hiện tại. Nếu tìm thấy, hàm sẽ gọi `error` với mã lỗi `ERR_DUPLICATE_IDENT` cùng với thông tin vị trí dòng và cột của token hiện tại.

1.5.2 ERR_UNDECLARED_*

Nhóm lỗi `ERR_UNDECLARED_*` bao gồm các lỗi sau:

- `ERR_UNDECLARED_IDENT`: identifier chưa được khai báo
- `ERR_UNDECLARED_CONSTANT`: constant chưa được khai báo
- `ERR_UNDECLARED_TYPE`: type chưa được khai báo
- `ERR_UNDECLARED_VARIABLE`: variable chưa được khai báo
- `ERR_UNDECLARED_PROCEDURE`: procedure chưa được khai báo

Các lỗi này được phát hiện khi hàm `lookupObject` trả về `NULL`, tức là không tìm thấy identifier trong scope chain và global list

Ví dụ, nếu trong một block có câu lệnh `X := 10;` nhưng X chưa được khai báo, hệ thống sẽ báo lỗi `ERR_UNDECLARED_VARIABLE`. Tương tự, nếu có câu lệnh `CALL P;` nhưng P chưa được khai báo, hệ thống sẽ báo lỗi `ERR_UNDECLARED_PROCEDURE`.

```
BEGIN
  X := 10;
  CALL P;
END;
```

Hàm `checkDeclaredVariable` minh họa cách xử lý lỗi này. Sau khi gọi `lookupObject`, nếu kết quả là NULL, hàm sẽ gọi `error` với mã lỗi tương ứng.

1.5.3 ERR_INVALID_*

Nhóm lỗi `ERR_INVALID_*` bao gồm các lỗi sau:

- `ERR_INVALID_CONSTANT`: identifier không phải constant
- `ERR_INVALID_TYPE`: identifier không phải type
- `ERR_INVALID_VARIABLE`: identifier không phải variable
- `ERR_INVALID_PROCEDURE`: identifier không phải procedure
- `ERR_INVALID_FACTOR`: identifier không thể dùng làm factor trong expression
- `ERR_INVALID_LVALUE`: identifier không thể dùng làm l-value
- `ERR_INVALID_RETURN`: gán giá trị return cho function không phải function hiện tại

Các lỗi này được phát hiện khi identifier đã được khai báo nhưng loại của nó không khớp với ngữ cảnh sử dụng, hoặc vi phạm quy tắc sử dụng (như trong trường hợp `ERR_INVALID_RETURN`).

Ví dụ, nếu X là variable và F là function, các câu lệnh `X := 10` và `F := 10` đều hợp lệ vì cả variable và function đều có thể là l-value. Câu lệnh `X := F` cũng hợp lệ vì function có thể dùng trong expression. Tuy nhiên, câu lệnh `CALL X` sẽ gây lỗi `ERR_INVALID_PROCEDURE` vì X là variable, không phải procedure.

```
VAR X: INTEGER;
FUNCTION F: INTEGER;
BEGIN
  X := 10;
  F := 10;
  X := F;
  CALL X;
  X := F();
END;
```

Hàm `checkDeclaredLValueIdent` minh họa cách kiểm tra loại identifier. Sau khi xác định identifier đã được khai báo, hàm kiểm tra xem loại của nó có phải là variable, parameter, hoặc function không. Chỉ những loại này mới có thể là l-value. Nếu không, hàm sẽ báo lỗi `ERR_INVALID_LVALUE`.

1.5.4 Tóm tắt luồng xử lý lỗi

Quá trình xử lý lỗi liên quan đến scope được thực hiện theo một luồng nhất quán. Khi khai báo identifier, hệ thống gọi `checkFreshIdent` để kiểm tra xem identifier có trùng với identifier đã tồn tại trong scope hiện tại không. Nếu trùng, hệ thống báo lỗi `ERR_DUPLICATE_IDENT`.

Khi sử dụng identifier, hệ thống gọi `lookupObject` để tìm kiếm identifier trong scope chain. Nếu kết quả là NULL, hệ thống báo lỗi `ERR_UNDECLARED_*` tương ứng với loại identifier mong đợi. Nếu tìm thấy, hệ thống tiếp tục kiểm tra loại của identifier. Nếu loại không khớp với ngữ cảnh sử dụng, hệ thống báo lỗi `ERR_INVALID_*` tương ứng.

2 Mô tả một hàm điển hình trong semantics.c

2.1 Hàm checkDeclaredLValueIdent

2.1.1 Tên hàm

Hàm có tên là `checkDeclaredLValueIdent`, được định nghĩa trong file `semantics.c` với signature như sau:

```
Object* checkDeclaredLValueIdent(char* name);
```

Hàm nhận vào một tham số là chuỗi ký tự `name` đại diện cho tên identifier cần kiểm tra, và trả về một con trỏ đến `Object` nếu identifier hợp lệ, hoặc báo lỗi nếu identifier không hợp lệ.

2.1.2 Mục đích

Hàm `checkDeclaredLValueIdent` có mục đích kiểm tra xem một identifier có thể được sử dụng như một l-value (left-value) hay không. L-value là một biểu thức có thể xuất hiện ở vế trái của phép gán, tức là có thể nhận giá trị được gán vào. Trong ngôn ngữ KPL, chỉ có ba loại identifier có thể là l-value: variable (biến), parameter (tham số), và function (hàm). Hàm này đảm bảo rằng identifier đã được khai báo và thuộc một trong ba loại trên trước khi cho phép sử dụng nó trong ngữ cảnh l-value.

Cụ thể, hàm thực hiện hai bước kiểm tra. Bước đầu tiên, hàm sử dụng `lookupObject` để tìm kiếm identifier trong toàn bộ scope chain, từ scope hiện tại lên các scope ngoài, và cuối cùng là trong global object list. Nếu không tìm thấy identifier ở bất kỳ đâu, hàm báo lỗi `ERR_UNDECLARED_IDENT`. Bước thứ hai, sau khi đã xác định identifier tồn tại, hàm kiểm tra loại của nó. Chỉ khi identifier là variable (`OBJ_VARIABLE`), parameter (`OBJ_PARAMETER`), hoặc function (`OBJ_FUNCTION`), hàm mới trả về object đó. Nếu identifier thuộc loại khác như constant (`OBJ_CONSTANT`), type (`OBJ_TYPE`), hoặc procedure (`OBJ_PROCEDURE`), hàm sẽ báo lỗi `ERR_INVALID_LVALUE`.

2.1.3 Khi nào được gọi

Hàm `checkDeclaredLValueIdent` được gọi bởi parser trong quá trình xử lý câu lệnh gán (assignment statement). Cụ thể, hàm được gọi từ hàm `compileLValue`, và `compileLValue` lại được gọi từ hàm `compileAssignSt` khi parser gặp một câu lệnh gán.

Khi parser gặp một token identifier ở đầu câu lệnh (không phải sau từ khóa `CALL`), nó nhận biết đây có thể là một câu lệnh gán. Hàm `compileStatement` sẽ gọi `compileAssignSt` trong trường hợp này. Trong `compileAssignSt`, parser gọi `compileLValue` để xử lý vế trái của phép gán. Hàm `compileLValue` sẽ đọc token identifier, sau đó gọi

```
checkDeclaredLValueIdent
```

để kiểm tra xem identifier này có thể là l-value không.

Ví dụ, khi parser gặp câu lệnh `X := 10;`, quá trình xử lý diễn ra như sau: Parser đọc token `X`, nhận biết đây là identifier và gọi `compileAssignSt`. Trong `compileAssignSt`, parser gọi `compileLValue`, và trong `compileLValue`, parser đọc token `X` và gọi

```
checkDeclaredLValueIdent("X")
```

để kiểm tra. Nếu X là variable, parameter, hoặc function, hàm trả về object X và quá trình tiếp tục. Sau đó parser đọc dấu `:=` và xử lý biểu thức bên phải.

Một trường hợp đặc biệt là khi gán giá trị cho function. Trong KPL, function có thể được gán giá trị để trả về kết quả. Ví dụ, trong câu lệnh `F := 1;` trong body của function F, hàm `checkDeclaredLValueIdent` sẽ xác nhận rằng F là function và cho phép gán giá trị cho nó. Sau khi kiểm tra, `compileLValue` sẽ gọi `compileArguments()` để xử lý các tham số nếu có (mặc dù trong trường hợp gán giá trị return, thường không có tham số).

2.1.4 Những lỗi hàm này có thể phát hiện

Hàm `checkDeclaredLValueIdent` có thể phát hiện hai loại lỗi ngữ nghĩa chính liên quan đến việc sử dụng identifier như l-value.

Lỗi đầu tiên là `ERR_UNDECLARED_IDENT`, được phát hiện khi identifier chưa được khai báo trong bất kỳ scope nào. Lỗi này xảy ra khi hàm `lookupObject` trả về NULL, tức là không tìm thấy identifier trong scope chain và global object list. Ví dụ, nếu trong một block có câu lệnh `Y := 10;` nhưng Y chưa được khai báo ở bất kỳ đâu, hàm sẽ báo lỗi `ERR_UNDECLARED_IDENT` tại vị trí của token Y.

Lỗi thứ hai là `ERR_INVALID_LVALUE`, được phát hiện khi identifier đã được khai báo nhưng không thuộc loại có thể là l-value. Cụ thể, nếu identifier là constant, type, hoặc procedure, hàm sẽ báo lỗi này. Ví dụ, nếu có khai báo `CONST C = 10;` và sau đó có câu lệnh `C := 5;`, hàm sẽ báo lỗi `ERR_INVALID_LVALUE` vì constant không thể nhận giá trị mới. Tương tự, nếu có procedure P và câu lệnh `P := 10;`, hàm cũng sẽ báo lỗi vì procedure không thể là l-value (lưu ý rằng đây khác với việc gọi procedure bằng `CALL P`).

Một ví dụ minh họa cụ thể: Trong chương trình có constant C, variable X, và function F. Câu lệnh `X := C;` là hợp lệ vì C là constant có thể dùng trong expression ở vế phải. Tuy nhiên, câu lệnh `C := 5;` sẽ gây lỗi `ERR_INVALID_LVALUE` vì C không thể là l-value. Câu lệnh `F := 10;` là hợp lệ nếu F là function, vì function có thể nhận giá trị để trả về. Nhưng nếu có procedure P và câu lệnh `P := 10;`, sẽ gây lỗi `ERR_INVALID_LVALUE`.

Việc phân biệt giữa các loại identifier và kiểm tra tính hợp lệ của chúng trong ngữ cảnh l-value là rất quan trọng để đảm bảo tính đúng đắn của chương trình. Hàm `checkDeclaredLValueIdent` đóng vai trò then chốt trong việc thực hiện kiểm tra này, giúp phát hiện các lỗi ngữ nghĩa sớm trong quá trình biên dịch và đảm bảo rằng chỉ những identifier phù hợp mới có thể được sử dụng ở vế trái của phép gán.

3 Kết quả thực hiện với example4.kpl

3.1 Chương trình nguồn

Chương trình `example4.kpl` có nội dung như sau:

```
PROGRAM  EXAMPLE4;  (* Example 4 *)
CONST MAX = 10;
TYPE T = INTEGER;
VAR  A : ARRAY(. 10 .) OF T;
     N : INTEGER;
     CH : CHAR;

PROCEDURE INPUT;
VAR I : INTEGER;
    TMP : INTEGER;
BEGIN
  N := READI;
  FOR I := 1 TO N DO
     A(.I.) := READI;
END;

PROCEDURE OUTPUT;
VAR I : INTEGER;
BEGIN
  FOR I := 1 TO N DO
     BEGIN
       CALL WRITEI(A(.I.));
       CALL WRITELN;
     END
END;

FUNCTION SUM : INTEGER;
VAR I: INTEGER;
    S : INTEGER;
BEGIN
    S := 0;
    I := 1;
    WHILE I <= N DO
```

```
34      BEGIN
35        S := S + A(.I.);
36        I := I + 1;
37      END
38  END;
39
40  BEGIN
41     CH := 'y';
42     WHILE CH = 'y' DO
43       BEGIN
44         CALL INPUT;
45         CALL OUTPUT;
46         CALL WRITEI(SUM);
47         CH := READC;
48       END
49  END.  (* Example 4 *)
```

3.2 Kết quả chạy chương trình

Khi chạy compiler với file `example4.kpl`, kết quả in ra cấu trúc Symbol Table như sau:

```
carbon@DESKTOP-C85RJUV:/mnt/c/CODE/Compiler_Construction/Lab04_Scope_Management$ ./semantics ./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

carbon@DESKTOP-C85RJUV:/mnt/c/CODE/Compiler_Construction/Lab04_Scope_Management$
```

Hình 1: Kết quả chạy compiler với example4.kpl

3.3 Giải thích kết quả

Kết quả in ra là cấu trúc Symbol Table sau khi compiler phân tích `example4.kpl`, thể hiện cấu trúc scope và các identifier đã được quản lý.

Cấu trúc scope:

1. **Program scope (EXAMPLE4)** - scope toàn cục:
   - `Const MAX = 10` - constant được khai báo với giá trị 10
   - `Type T = Int` - type alias (T = INTEGER)
   - `Var A : Arr(10,Int)` - mảng 10 phần tử kiểu INTEGER
   - `Var N : Int`, `Var CH : Char` - các biến toàn cục
2. **Procedure INPUT scope** - scope riêng:

- `Var I : Int`, `Var TMP : Int` - các biến cục bộ

3. **Procedure OUTPUT scope** - scope riêng:
   - `Var I : Int` - biến cục bộ (có thể trùng tên với I trong INPUT)
4. **Function SUM scope** - scope riêng:
   - `Var I : Int`, `Var S : Int` - các biến cục bộ

Điểm quan trọng:

- Mỗi procedure/function có scope riêng, thể hiện qua indentation (thụt lề 4 spaces)
- Các biến cùng tên (như `I`) ở các scope khác nhau là các object độc lập
- Type `T` được resolve thành `Int` (INTEGER) khi in ra
- Array được hiển thị dạng `Arr(10,Int)` thay vì `ARRAY(. 10 .) OF INTEGER`

Kết quả cho thấy hệ thống quản lý scope hoạt động đúng: mỗi block có scope riêng, và các identifier được tổ chức theo cấu trúc phân cấp.

4 Các trường hợp gây ra lỗi

4.1 ERR_UNDECLARED_IDENT

File test: `test_undeclared_ident.kpl`

```
PROGRAM TEST_UNDECLARED_IDENT;
BEGIN
  X := 10;  //ERR_UNDECLARED_IDENT
END.
```

Nguyên nhân: Identifier `X` được sử dụng nhưng chưa được khai báo trong bất kỳ scope nào (không có trong scope chain và global object list).

Hàm xử lý: `checkDeclaredLValueIdent()` trong `semantics.c`

```
Object* checkDeclaredLValueIdent(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_IDENT, currentToken->lineNo, currentToken
      ->colNo);
  if (obj->kind != OBJ_VARIABLE && obj->kind != OBJ_PARAMETER &&
     obj->kind != OBJ_FUNCTION)
    error(ERR_INVALID_LVALUE, currentToken->lineNo, currentToken->
      colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp identifier `X` trong assignment `X := 10`

2. Gọi `compileAssignSt()` → `compileLValue()`
3. `compileLValue()` gọi `checkDeclaredLValueIdent("X")`
4. `checkDeclaredLValueIdent` gọi `lookupObject("X")` để tìm trong scope chain
5. `lookupObject` không tìm thấy → trả về NULL
6. `checkDeclaredLValueIdent` phát hiện NULL → gọi `error(ERR_UNDECLARED_IDENT, ...)`

4.2 ERR_UNDECLARED_CONSTANT

File test: `test_undeclared_constant.kpl`

```
PROGRAM TEST;
VAR Y : INTEGER;
BEGIN
  Y := C;  // ERR_UNDECLARED_CONSTANT
END.
```

Nguyên nhân: Constant `C` được sử dụng trong expression (vế phải của assignment) nhưng chưa được khai báo.

Hàm xử lý: `compileFactor()` trong `parser.c`

```
void compileFactor(void) {
  Object* obj;
  // ...
  case TK_IDENT:
    eat(TK_IDENT);
    obj = lookupObject(currentToken->string);
    if (obj == NULL) {
      error(ERR_UNDECLARED_CONSTANT, currentToken->lineNo,
          currentToken->colNo);
      return;
    }
    // ...
}
```

Luồng xử lý:

1. Parser gặp identifier `C` trong expression `Y := C`
2. Gọi `compileExpression()` → `compileTerm()` → `compileFactor()`
3. `compileFactor()` gọi `lookupObject("C")` trực tiếp
4. `lookupObject` không tìm thấy → trả về NULL
5. `compileFactor` phát hiện NULL → gọi `error(ERR_UNDECLARED_CONSTANT, ...)`

Lưu ý: Trong ngữ cảnh expression, identifier chưa khai báo được coi là constant vì constant thường được dùng trong expression.

4.3 ERR_UNDECLARED_TYPE

File test: `test_undeclared_type.kpl`

```
PROGRAM TEST;
VAR X : T;  // ERR_UNDECLARED_TYPE
BEGIN
END.
```

Nguyên nhân: Type `T` được sử dụng trong khai báo variable nhưng chưa được khai báo.

Hàm xử lý: `checkDeclaredType()` trong `semantics.c`

```
Object* checkDeclaredType(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_TYPE, currentToken->lineNo, currentToken
       ->colNo);
  if (obj->kind != OBJ_TYPE)
    error(ERR_INVALID_TYPE, currentToken->lineNo, currentToken->
       colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp identifier `T` trong khai báo type `VAR X : T`
2. Gọi `compileBlock3()` → `compileType()`
3. `compileType()` gọi `checkDeclaredType("T")`
4. `checkDeclaredType` gọi `lookupObject("T")`
5. Không tìm thấy → NULL → báo lỗi `ERR_UNDECLARED_TYPE`

4.4 ERR_UNDECLARED_VARIABLE

File test: `test_undeclared_variable.kpl`

```
PROGRAM TEST;
BEGIN
  FOR Z := 1 TO 5 DO  // ERR_UNDECLARED_VARIABLE
    BEGIN
    END;
END.
```

Nguyên nhân: Variable `Z` được sử dụng trong FOR loop nhưng chưa được khai báo. FOR loop yêu cầu variable cụ thể, không phải identifier tổng quát.

Hàm xử lý: `checkDeclaredVariable()` trong `semantics.c`

```
Object* checkDeclaredVariable(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_VARIABLE, currentToken->lineNo,
       currentToken->colNo);
```

```
  if (obj->kind != OBJ_VARIABLE)
    error(ERR_INVALID_VARIABLE, currentToken->lineNo, currentToken
       ->colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp identifier `Z` trong FOR loop `FOR Z := 1 TO 5 DO`
2. Gọi `compileForSt()`
3. `compileForSt()` gọi `checkDeclaredVariable("Z")`
4. `checkDeclaredVariable` gọi `lookupObject("Z")`
5. Không tìm thấy → NULL → báo lỗi `ERR_UNDECLARED_VARIABLE`

4.5 ERR_UNDECLARED_PROCEDURE

File test: `test_undeclared_procedure.kpl`

```
PROGRAM TEST;
BEGIN
  CALL P;  // ERR_UNDECLARED_PROCEDURE
END.
```

Nguyên nhân: Procedure `P` được gọi bằng `CALL` nhưng chưa được khai báo.
Hàm xử lý: `checkDeclaredProcedure()` trong `semantics.c`

```
Object* checkDeclaredProcedure(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_PROCEDURE, currentToken->lineNo,
       currentToken->colNo);
  if (obj->kind != OBJ_PROCEDURE)
    error(ERR_INVALID_PROCEDURE, currentToken->lineNo,
       currentToken->colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp `CALL P` → gọi `compileCallSt()`
2. `compileCallSt` gọi `checkDeclaredProcedure("P")`
3. `checkDeclaredProcedure` gọi `lookupObject("P")`
4. Không tìm thấy → NULL → báo lỗi `ERR_UNDECLARED_PROCEDURE`

4.6 ERR_INVALID_VARIABLE

File test: `test_invalid_variable.kpl`

```
PROGRAM TEST;
FUNCTION F : INTEGER;
BEGIN
END;
VAR X : INTEGER;
BEGIN
  FOR F := 1 TO 10 DO  // ERR_INVALID_VARIABLE
    X := 1;
END.
```

Nguyên nhân: `F` là function nhưng được sử dụng trong FOR loop, nơi yêu cầu variable cụ thể.

Hàm xử lý: `checkDeclaredVariable()` trong `semantics.c`

```
Object* checkDeclaredVariable(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_VARIABLE, currentToken->lineNo,
       currentToken->colNo);
  if (obj->kind != OBJ_VARIABLE)
    error(ERR_INVALID_VARIABLE, currentToken->lineNo, currentToken
       ->colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp identifier `F` trong FOR loop `FOR F := 1 TO 10 DO`
2. Gọi `compileForSt()`
3. `compileForSt()` gọi `checkDeclaredVariable("F")`
4. `lookupObject("F")` tìm thấy function F
5. Kiểm tra `obj->kind != OBJ_VARIABLE` → TRUE (vì F là OBJ_FUNCTION)
6. Báo lỗi `ERR_INVALID_VARIABLE`

4.7 ERR_INVALID_RETURN

File test: `test_invalid_return.kpl`

```
PROGRAM TEST;
FUNCTION F : INTEGER;
FUNCTION G : INTEGER;
BEGIN
  F := 10;  // ERR_INVALID_RETURN
END;
BEGIN
END.
```

Nguyên nhân: Trong function `G`, cố gắng gán giá trị return cho function `F` khác. Chỉ có thể gán giá trị return cho chính function hiện tại (owner của current scope).

Hàm xử lý: `compileLValue()` trong `parser.c`

```
void compileLValue(void) {
  Object* var;
  eat(TK_IDENT);
  var = checkDeclaredLValueIdent(currentToken->string);
  switch (var->kind) {
  case OBJ_VARIABLE:
    compileIndexes();
    break;
  case OBJ_PARAMETER:
    break;
  case OBJ_FUNCTION:
    if (symtab->currentScope->owner != NULL &&
        symtab->currentScope->owner->kind == OBJ_FUNCTION) {
      if (var != symtab->currentScope->owner) {
        error(ERR_INVALID_RETURN, currentToken->lineNo,
           currentToken->colNo);
      }
    }
    compileArguments();
    break;
  default:
    error(ERR_INVALID_LVALUE, currentToken->lineNo, currentToken->
       colNo);
    break;
  }
}
```

Luồng xử lý:

1. Parser gặp `F := 10` trong function G → gọi `compileAssignSt()` → `compileLValue()`
2. `compileLValue()` phát hiện `F` là function (OBJ_FUNCTION)
3. Kiểm tra: đang trong function scope (`symtab->currentScope->owner->kind == OBJ_FUNCTION`)
4. Kiểm tra: `var != symtab->currentScope->owner` → TRUE (F != G)
5. Báo lỗi `ERR_INVALID_RETURN`

4.8 ERR_DUPLICATE_IDENT

File test: `test_duplicate_ident.kpl`

```
PROGRAM TEST;
VAR X : INTEGER;
VAR X : CHAR;  // ERR_DUPLICATE_IDENT
BEGIN
END.
```

Nguyên nhân: Identifier `X` được khai báo hai lần trong cùng một scope. Mỗi identifier chỉ có thể được khai báo một lần trong một scope.

Hàm xử lý: `checkFreshIdent()` trong `semantics.c`

```
void checkFreshIdent(char *name) {
  if (findObject(symtab->currentScope->objList, name) != NULL)
    error(ERR_DUPLICATE_IDENT, currentToken->lineNo, currentToken
       ->colNo);
}
```

Luồng xử lý:

1. Parser gặp khai báo `VAR X : INTEGER;` → gọi `compileBlock3()`
2. Trước khi khai báo, gọi `checkFreshIdent("X")`
3. `checkFreshIdent` tìm trong `symtab->currentScope->objList` → không tìm thấy → OK
4. Khai báo X thành công
5. Parser gặp khai báo `VAR X : CHAR;` → gọi `checkFreshIdent("X")` lại
6. `checkFreshIdent` tìm trong `symtab->currentScope->objList` → tìm thấy X đã tồn tại
7. Báo lỗi `ERR_DUPLICATE_IDENT`

Lưu ý: `checkFreshIdent()` chỉ kiểm tra trong scope hiện tại (`currentScope->objList`), không kiểm tra scope ngoài, vì cho phép cùng tên ở các scope khác nhau.

4.9 ERR_INVALID_LVALUE

File test: `test_invalid_lvalue.kpl`

```
PROGRAM TEST;
CONST C = 10;
BEGIN
  C := 5;  // ERR_INVALID_LVALUE
END.
```

Nguyên nhân: Constant `C` được sử dụng ở vế trái của assignment (l-value), nhưng constant không thể nhận giá trị mới. Chỉ variable, parameter, hoặc function mới có thể là l-value.

Hàm xử lý: `checkDeclaredLValueIdent()` trong `semantics.c`

```
Object* checkDeclaredLValueIdent(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_IDENT, currentToken->lineNo, currentToken
       ->colNo);
  if (obj->kind != OBJ_VARIABLE && obj->kind != OBJ_PARAMETER &&
     obj->kind != OBJ_FUNCTION)
    error(ERR_INVALID_LVALUE, currentToken->lineNo, currentToken->
       colNo);
```

```
  return obj;
}
```

Luồng xử lý:

1. Parser gặp `C := 5` → gọi `compileAssignSt()` → `compileLValue()`
2. `compileLValue` gọi `checkDeclaredLValueIdent("C")`
3. `lookupObject("C")` tìm thấy constant C
4. Kiểm tra: `C->kind == OBJ_CONSTANT` → không phải VARIABLE, PARAMETER, hay FUNCTION
5. Báo lỗi `ERR_INVALID_LVALUE`

4.10 ERR_INVALID_CONSTANT

File test: `test_invalid_constant.kpl`

```
PROGRAM TEST;
VAR X : INTEGER;
CONST C = X;  // ERR_INVALID_CONSTANT
BEGIN
END.
```

Nguyên nhân: Identifier `X` được sử dụng trong ngữ cảnh yêu cầu constant (khai báo constant) nhưng `X` là variable, không phải constant.

Hàm xử lý: `checkDeclaredConstant()` trong `semantics.c`

```
Object* checkDeclaredConstant(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_CONSTANT, currentToken->lineNo,
        currentToken->colNo);
  if (obj->kind != OBJ_CONSTANT)
    error(ERR_INVALID_CONSTANT, currentToken->lineNo, currentToken
        ->colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp identifier `X` trong khai báo constant `CONST C = X`
2. Gọi `compileBlock()`→ `compileConstant()` → `compileConstant2()` → `checkDeclaredConstan`
3. `checkDeclaredConstant` gọi `lookupObject("X")`
4. `lookupObject("X")` tìm thấy variable X
5. Kiểm tra `obj->kind != OBJ_CONSTANT` → TRUE (vì X là OBJ_VARIABLE)
6. Báo lỗi `ERR_INVALID_CONSTANT`

4.11 ERR_INVALID_TYPE

File test: `test_invalid_type.kpl`

```
PROGRAM TEST;
VAR X : INTEGER;
VAR Y : X;  // ERR_INVALID_TYPE
BEGIN
END.
```

Nguyên nhân: Identifier `X` được sử dụng trong ngữ cảnh yêu cầu type (khai báo type của variable) nhưng `X` là variable, không phải type.

Hàm xử lý: `checkDeclaredType()` trong `semantics.c`

```
Object* checkDeclaredType(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_TYPE, currentToken->lineNo, currentToken
       ->colNo);
  if (obj->kind != OBJ_TYPE)
    error(ERR_INVALID_TYPE, currentToken->lineNo, currentToken->
       colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp identifier `X` trong khai báo type `VAR Y : X`
2. Gọi `compileBlock3()` → `compileType()`
3. `compileType()` gọi `checkDeclaredType("X")`
4. `lookupObject("X")` tìm thấy variable X
5. Kiểm tra `obj->kind != OBJ_TYPE` → TRUE (vì X là OBJ_VARIABLE)
6. Báo lỗi `ERR_INVALID_TYPE`

4.12 ERR_INVALID_FACTOR

File test: `test_invalid_factor.kpl`

```
PROGRAM TEST;
PROCEDURE P;
BEGIN
END;
VAR X : INTEGER;
BEGIN
  X := P;  // ERR_INVALID_FACTOR
END.
```

Nguyên nhân: Procedure `P` được sử dụng trong expression (factor), nhưng procedure không thể trả về giá trị để dùng trong expression. Chỉ constant, variable, parameter, hoặc function mới có thể dùng trong expression.

Hàm xử lý: `compileFactor()` trong `parser.c`

```
void compileFactor(void) {
  Object* obj;
  switch (lookAhead->tokenType) {
  case TK_IDENT:
    eat(TK_IDENT);
    obj = lookupObject(currentToken->string);
    if (obj == NULL) {
      error(ERR_UNDECLARED_CONSTANT, currentToken->lineNo,
          currentToken->colNo);
      return;
    }
    switch (obj->kind) {
    case OBJ_CONSTANT:
      break;
    case OBJ_VARIABLE:
      compileIndexes();
      break;
    case OBJ_PARAMETER:
      break;
    case OBJ_FUNCTION:
      compileArguments();
      break;
    case OBJ_PROCEDURE:
    case OBJ_TYPE:
    case OBJ_PROGRAM:
    default:
      error(ERR_INVALID_FACTOR, currentToken->lineNo, currentToken
          ->colNo);
      return;
    }
    break;
  }
}
```

Luồng xử lý:

1. Parser gặp identifier `P` trong expression `X := P`
2. Gọi `compileExpression()` → `compileFactor()`
3. `compileFactor()` gọi `lookupObject("P")` → tìm thấy procedure P
4. Kiểm tra `obj->kind` → là `OBJ_PROCEDURE`
5. Không khớp với các case hợp lệ (CONSTANT, VARIABLE, PARAMETER, FUNCTION)
6. Vào default case → báo lỗi `ERR_INVALID_FACTOR`

4.13 ERR_INVALID_PROCEDURE

File test: `test_invalid_procedure.kpl`

```
PROGRAM TEST;
VAR X : INTEGER;
BEGIN
  CALL X;  // ERR_INVALID_PROCEDURE
END.
```

Nguyên nhân: Variable `X` được sử dụng sau `CALL` nhưng `CALL` chỉ dùng cho procedure, không phải variable.

Hàm xử lý: `checkDeclaredProcedure()` trong `semantics.c`

```
Object* checkDeclaredProcedure(char* name) {
  Object* obj = lookupObject(name);
  if (obj == NULL)
    error(ERR_UNDECLARED_PROCEDURE, currentToken->lineNo,
       currentToken->colNo);
  if (obj->kind != OBJ_PROCEDURE)
    error(ERR_INVALID_PROCEDURE, currentToken->lineNo,
       currentToken->colNo);
  return obj;
}
```

Luồng xử lý:

1. Parser gặp `CALL X` → gọi `compileCallSt()`
2. `compileCallSt` gọi `checkDeclaredProcedure("X")`
3. `lookupObject("X")` tìm thấy variable X
4. Kiểm tra `obj->kind != OBJ_PROCEDURE` → TRUE (vì X là OBJ_VARIABLE)
5. Báo lỗi `ERR_INVALID_PROCEDURE`

5 Kết luận

Tất cả các lỗi semantic đều được phát hiện thông qua các hàm kiểm tra trong `semantics.c` và `parser.c`. Quá trình kiểm tra luôn bắt đầu bằng `lookupObject()` để tìm identifier trong scope chain, sau đó kiểm tra:

1. **Identifier có tồn tại không?** → Nếu không, báo lỗi `ERR_UNDECLARED_*`
2. **Loại của identifier có phù hợp với ngữ cảnh sử dụng không?** → Nếu không, báo lỗi `ERR_INVALID_*`
3. **Identifier có bị trùng lặp trong scope hiện tại không?** → Nếu có, báo lỗi `ERR_DUPLICATE_IDENT`

Các hàm kiểm tra này đảm bảo tính đúng đắn của chương trình bằng cách phát hiện các lỗi ngữ nghĩa sớm trong quá trình biên dịch.