Thực hành
Xây dựng chương trình dịch

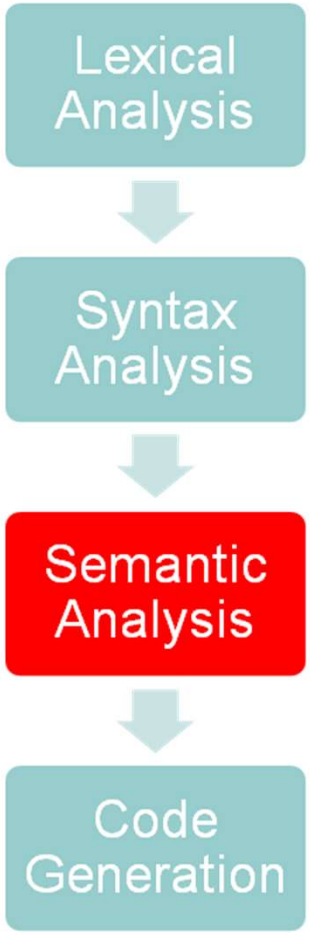
Bài 3: Bảng ký hiệu & Quản lý phạm vi

ONE LOVE. ONE FUTURE.

Nội dung

- Tổng quan về phân tích ngữ nghĩa
- Xây dựng bảng ký hiệu
- Quản lý phạm vi

Phân tích ngữ nghĩa là gì?



- Phân tích cú pháp chỉ kiểm tra cấu trúc ngữ pháp hợp lệ của chương trình
- Những yêu cầu khác ngoài cấu trúc ngữ pháp:
  - "x" đã được khai báo chưa?
  - "x" được khai báo ở đâu?
  - "x" là tên một biến hay một hàm?
  - Biểu thức "a+b" có kiểu nhất quán không?
  - …
- Phân tích ngữ ngữ nghĩa trả lời các câu hỏi đó để làm rõ hơn ngữ nghĩa của chương trình.

Nhiệm vụ của một bộ phân tích ngữ nghĩa

- Quản lý thông tin về các định danh
  - Hằng
  - Biến
  - Kiểu người dùng định nghĩa
  - Chương trình con (hàm, thủ tục)
- Kiểm tra một số luật ngữ nghĩa
  - Phạm vi định danh
  - Nhất quán kiểu

Bảng ký hiệu (1)

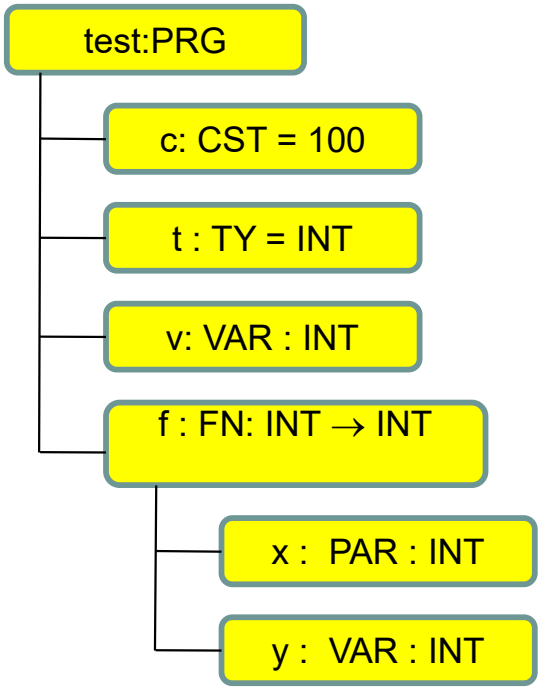
- Lưu trữ thông tin về các định danh trong chương trình và các thuộc tính của chúng
  - Hằng: {tên, kiểu, giá trị}
  - Kiểu người dùng định nghĩa: {tên, kiểu thực tế}
  - Biến: {tên, kiểu}
  - Hàm: {tên, các tham số hình thức, kiểu trả về, các định danh trong khai báo địa phương}
  - Thủ tục: {tên, các tham số hình thức, các định danh trong khai báo địa phương)
  - Tham số hình thức: {tên, kiểu, tham biến/tham trị}

Bảng ký hiệu (2)

- Trong chương trình dịch KPL, bảng ký hiệu được biểu diễn theo cấu trúc phân cấp
- Ví dụ: Lập bảng ký hiệu cho chương trình sau:

```
PROGRAM test;
CONST c = 100;
TYPE t = Integer;
VAR v : t;
FUNCTION f(x : t) : t;
  VAR y : t;
BEGIN
  y := x + 1;
  f := y;
END;

BEGIN
  v := 1;
  WriteI(f(v));
END.
```



Xây dựng bảng ký hiệu cho ngôn ngữ KPL

- Khởi tạo và giải phóng
- Thêm thông tin vào bảng ký hiệu thông qua các khai báo:
  - Khai báo hằng
  - Khai báo kiểu
  - Khai báo biến
  - Khai báo thủ tục và hàm
  - Khai báo tham số

Xây dựng bảng ký hiệu (1)

- Các thành phần của bảng ký hiệu

```
// Bảng ký hiệu
struct SymTab_ {
  // Chương trình chính
  Object* program;
  // Phạm vi hiện tại
  Scope* currentScope;
  // Các đối tượng toàn cục như
  // hàm WRITEI, WRITEC, WRITELN
  // READI, READC
  ObjectNode *globalObjectList;
};
```

```
// Phạm vi của một block
struct Scope_ {
  // Danh sách các đối tượng trong
  // block
  ObjectNode *objList;
  // Hàm, thủ tục, chương trình
  // tương ứng block
  Object *owner;
  // Phạm vi bao ngoài
  struct Scope_ *outer;
};
```

Xây dựng bảng ký hiệu (2)

- Bảng ký hiệu ghi nhớ block hiện đang duyệt trong biến `currentScope`
- Mỗi khi dịch một hàm hay thủ tục, phải cập nhật giá trị của `currentScope`

```
void enterBlock(Scope* scope);
```

- Mỗi khi kết thúc duyệt một hàm hay thủ tục phải chuyển lại `currentScope` ra block bên ngoài

```
void exitBlock(void);
```

- Đăng ký một đối tượng vào block hiện tại

```
void declareObject(Object* obj);
```

Lối vào của bảng ký hiệu: đối tượng

```
// Phân loại ký hiệu

enum ObjectKind {
  OBJ_CONSTANT,
  OBJ_VARIABLE,
  OBJ_TYPE,
  OBJ_FUNCTION,
  OBJ_PROCEDURE,
  OBJ_PARAMETER,
  OBJ_PROGRAM
};
```

```
// Thuộc tính của đối tượng trên bảng ký hiệu

struct Object_ {
  char name[MAX_IDENT_LEN];
  enum ObjectKind kind;
  union {
    ConstantAttributes* constAttrs;
    VariableAttributes* varAttrs;
    TypeAttributes* typeAttrs;
    FunctionAttributes* funcAttrs;
    ProcedureAttributes* procAttrs;
    ProgramAttributes* progAttrs;
    ParameterAttributes* paramAttrs;
  };
};
```

Thuộc tính của từng loại đối tượng (1)

```
struct ConstantAttributes_ {
  ConstantValue* value;
};

struct VariableAttributes_ {
  Type *type;
  // Phạm vi của biến (sử dụng cho pha sinh mã)
  struct Scope_ *scope;
};

struct TypeAttributes_ {
  Type *actualType;
};

struct ParameterAttributes_ {
  // Tham biến hoặc tham trị
  enum ParamKind kind;
  Type* type;
  struct Object_ *function;
};
```

Thông tin đặc trưng của hằng và kiểu

```
// Phân loại kiểu

enum TypeClass {
  TP_INT,
  TP_CHAR,
  TP_ARRAY
};

struct Type_ {
  enum TypeClass typeClass;
  // Chỉ sử dụng cho kiểu mảng
  int arraySize;
  struct Type_ *elementType;
};
```

```
// Hằng

struct ConstantValue_ {
  enum TypeClass type;
  union {
    int intValue;
    char charValue;
  };
};
```

Lưu trữ thông tin đặc trưng của hằng và kiểu

- Các hàm tạo kiểu

```
Type* makeIntType(void);
Type* makeCharType(void);
Type* makeArrayType(int arraySize, Type* elementType);
Type* duplicateType(Type* type)
```

- Các hàm tạo giá trị hằng số

```
ConstantValue* makeIntConstant(int i);
ConstantValue* makeCharConstant(char ch);
ConstantValue*
  duplicateConstantValue(ConstantValue* v);
```

Thuộc tính của từng loại đối tượng (2)

```
struct ProcedureAttributes_ {
  struct ObjectNode_ *paramList;
  struct Scope_* scope;
};

struct FunctionAttributes_ {
  struct ObjectNode_ *paramList;
  Type* returnType;
  struct Scope_ *scope;
};

struct ProgramAttributes_ {
  struct Scope_ *scope;
};

// Lưu ý: các đối tượng tham số hình thức vừa được đăng ký trong danh
   sách tham số (paramList), vừa được đăng ký trong danh sách các đối
   tượng được định nghĩa trong block (scope->objList)
```

Quá trình xây dựng bảng ký hiệu

```
int compile(char *fileName) {
  …
  // Khởi tạo bảng ký hiệu
  initSymTab();
  // Dịch chương trình, điền thông tin vào bảng ký hiệu
  compileProgram();
  // In chương trình để kiểm tra kết quả
  printObject(symtab->program,0);
  // Giải phóng bảng ký hiệu
  cleanSymTab();
  …
}
```

Khởi tạo bảng ký hiệu

- Đối tượng chương trình được khởi tạo ở hàm:

```
void compileProgram(void);
```

- Sau khi khởi tạo cho đối tượng chương trình, vào **khối** chính bằng hàm `enterBlock()`
- Sau khi đọc xong chương trình, ra khỏi khối chính bằng hàm `exitBlock()`

Đưa thông tin về hằng vào bảng ký hiệu

- Các đối tượng hằng số được tạo ra và khai báo ở hàm `compileBlock()`
- Giá trị của hằng số được lấy từ quá trình duyệt giá trị hằng qua hàm

```
ConstantValue* compileConstant(void)

ConstantValue* compileConstant2(void)
```

- Nếu giá trị hằng là một định danh hằng, phải tra bảng ký hiệu để lấy giá trị tương ứng
- Sau khi duyệt xong một hằng số, phải đăng ký vào block hiện tại bằng hàm `declareObject`

Ví dụ về điền thuộc tính cho hàm: Hàm compileConstant2

```
ConstantValue* compileConstant2(void) {
  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) && (obj->constAttrs->value->type == TP_INT))
      constValue = duplicateConstantValue(obj->constAttrs->value);
    else
      error(ERR_UNDECLARED_INT_CONSTANT,currentToken->lineNo, currentToken->colNo);
    break;
  default:
    error(ERR_INVALID_CONSTANT, lookAhead->lineNo, lookAhead->colNo);
    break;
  }
  return constValue;
}
```

Đưa thông tin về kiểu vào bảng ký hiệu

- Các đối tượng kiểu được tạo ra và khai báo ở hàm `compileBlock2()`
- Kiểu thực tế được lấy từ quá trình duyệt kiểu bằng hàm

  `Type* compileType(void)`

  - Nếu gặp định danh kiểu thì phải tra bảng ký hiệu để lấy kiểu tương ứng
- Sau khi duyệt xong một kiểu người dùng định nghĩa, phải đăng ký vào block hiện tại bằng hàm `declareObject`

Đưa thông tin về biến vào bảng ký hiệu

- Các đối tượng biến được tạo ra và khai báo ở hàm `compileBlock3()`
- Kiểu của biến được lấy từ quá trình duyệt kiểu bằng hàm

  `Type* compileType(void)`
- Lưu trữ phạm vi hiện tại vào danh sách thuộc tính của đối tượng biến để phục vụ mục đích sinh mã sau này
- Sau khi duyệt xong một biến, phải đăng ký vào block hiện tại bằng hàm `declareObject`

Đưa thông tin về hàm vào bảng ký hiệu

- Các đối tượng hàm được tạo ra và khai báo ở hàm `compileFuncDecl()`
- Các thuộc tính của đối tượng hàm sẽ được cập nhật bao gồm:
  - danh sách tham số: `compileParams`
  - kiểu dữ liệu trả về: `compileBasicType`
  - phạm vi của hàm
- Lưu ý đăng ký đối tượng hàm vào block hiện tại và chuyển block hiện tại sang block của hàm trước khi duyệt tiếp các đối tượng cục bộ

Đưa thông tin về thủ tục vào bảng ký hiệu

- Các đối tượng thủ tục được tạo ra và khai báo ở hàm `compileProcDecl()`
- Các thuộc tính của đối tượng thủ tục sẽ được cập nhật bao gồm:
  - danh sách tham số: `compileParams`
  - phạm vi của thủ tục
- Lưu ý đăng ký đối tượng thủ tục vào block hiện tại và chuyển block hiện tại sang block của hàm trước khi duyệt tiếp các đối tượng cục bộ

Đưa thông tin về tham số (hình thức) vào bảng ký hiệu

- Các đối tượng tham số hình thức được tạo ra và khai báo ở hàm `compileParam()`
- Các thuộc tính của đối tượng tham số hình thức bao gồm:
  - Kiểu dữ liệu cơ bản
  - Tham biến (PARAM_REFERENCE) hoặc tham trị (PARAM_VALUE)
- Lưu ý: đối tượng tham số hình thức nên được đăng ký vào đồng thời vào cả thuộc tính paramList của hàm/thủ tục hiện tại, cả vào danh sách đối tượng trong phạm vi hiện tại

Nhiệm vụ thực hành

- Tìm hiểu lại cấu trúc của bộ parser (có thay đổi)
- Bổ xung các đoạn code vào những hàm có đánh dấu TODO để thực hiện các công việc đăng ký đối tượng
- Biên dịch và thử nghiệm với các ví dụ mẫu

So sánh với bài 1: Tạo chương trình

```
obj = createProgramObject("PRG");
 enterBlock(obj->progAttrs->scope);
```

```
// TODO: create, enter, and exit program block
  eat(KW_PROGRAM);
  eat(TK_IDENT);
  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_PERIOD);
}
```

So sánh với bài 1: Tạo hằng

```
obj = createConstantObject("c1");
  obj->constAttrs->value = makeIntConstant(10);
  declareObject(obj);
  obj = createConstantObject("c2");
  obj->constAttrs->value = makeCharConstant('a');
  declareObject(obj);
```

```
void compileBlock(void) {
  // TODO: create and declare constant
objects
  if (lookAhead->tokenType == KW_CONST) {
    eat(KW_CONST);
    do {
      eat(TK_IDENT);
      eat(SB_EQ);
      compileConstant();
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType ==
TK_IDENT);

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

So sánh với bài 1: Tạo kiểu

```
obj = createTypeObject("t1");
  obj->typeAttrs->actualType = makeArrayType(10,makeIntType());
  declareObject(obj);
```

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

    do {
      eat(TK_IDENT);
      eat(SB_EQ);
      compileType();
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);

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

So sánh với bài 1: Tạo biến

```
obj = createVariableObject("v1");
  obj->varAttrs->type = makeIntType();
  declareObject(obj);
obj = createVariableObject("v2");
  obj->varAttrs->type =
makeArrayType(10,makeArrayType(10,makeIntType()));
  declareObject(obj);
```

```
void compileBlock3(void) {
  // TODO: create and declare variable objects
  if (lookAhead->tokenType == KW_VAR) {
    eat(KW_VAR);
    do {
      eat(TK_IDENT);
      eat(SB_COLON);
      compileType();
      eat(SB_SEMICOLON);
    } while (lookAhead->tokenType == TK_IDENT);
    compileBlock4();
  }
  else compileBlock4();
}
```

Sử dụng các hàm duplicate

```
Type* compileType(void) {
 // TODO: create and return a type
 Type* type;
 switch (lookAhead->tokenType) {
 case KW_INTEGER:
  eat(KW_INTEGER);
  break;
 case KW_CHAR:
  eat(KW_CHAR);
  break;
 case KW_ARRAY:
  eat(KW_ARRAY);
  eat(SB_LSEL);
  eat(TK_NUMBER);
  eat(SB_RSEL);
  eat(KW_OF);
  elementType)= compileType();
  type = makeArrayType(arraySize, elementType);
  break;
 case TK_IDENT:
  eat(TK_IDENT);
  break;
 default:
  error(ERR_INVALID_TYPE, lookAhead->lineNo, lookAhead->colNo);
  break;
 }
 return type;
}
```

```
ConstantValue* compileUnsignedConstant(void) {
 // TODO: create and return an unsigned constant value
 ConstantValue* constValue;
 switch (lookAhead->tokenType) {
 case TK_NUMBER:
  eat(TK_NUMBER);
constValue = makeIntConstant(currentToken->value);
  break;
 case TK_IDENT:
  eat(TK_IDENT);
.....
constValue = duplicateConstantValue(obj->constAttrs->value);
  break;
 case TK_CHAR:
  eat(TK_CHAR);
  break;
 default:
  error(ERR_INVALID_CONSTANT, lookAhead->lineNo, lookAhead->colNo);
  break;
 }
 return constValue;
}
```

So sánh với bài 1: Tạo hàm

```
obj = createFunctionObject("f");
  obj->funcAttrs->returnType = makeIntType();
  declareObject(obj);
      enterBlock(obj->funcAttrs->scope);
     obj = createParameterObject("p1", PARAM_VALUE, symtab->currentScope->owner);
    obj->paramAttrs->type = makeIntType();
    declareObject(obj);
    obj = createParameterObject("p2", PARAM_REFERENCE, symtab->currentScope->owner);
    obj->paramAttrs->type = makeCharType();
    declareObject(obj);
    exitBlock();
```

```
void compileFuncDecl(void) {
  // TODO: create and declare a function object
  eat(KW_FUNCTION);
  eat(TK_IDENT);
  compileParams();
  eat(SB_COLON);
  returnType = compileBasicType();
  funcObj->funcAttrs->returnType = returnType;

  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_SEMICOLON);
}
```

So sánh với bài 1: Tạo tham số

```
obj = createParameterObject("p1", PARAM_VALUE, symtab->currentScope-> owner);
    obj->paramAttrs->type = makeIntType();
    declareObject(obj);
    obj = createParameterObject("p2", PARAM_REFERENCE, symtab->currentScope-> owner);
    obj->paramAttrs->type = makeCharType();
    declareObject(obj);
```

```
void compileParam(void) {
  // TODO: create and declare a parameter
  switch (lookAhead->tokenType) {
  case TK_IDENT:
    eat(TK_IDENT);
    eat(SB_COLON);
    compileBasicType();
    break;
  case KW_VAR:
    eat(KW_VAR);
    eat(TK_IDENT);
    eat(SB_COLON);
    compileBasicType();
    break;
  default:
    error(ERR_INVALID_PARAMETER, lookAhead->lineNo, lookAhead->colNo);
    break;
  }
}
```

So sánh với bài 1: Tạo thủ tục

```
obj = createProcedureObject("p");
  declareObject(obj);
      enterBlock(obj->procAttrs->scope);
 exitBlock();
```

```
 void compileProcDecl(void) {
  // TODO: create and declare a procedure object
  eat(KW_PROCEDURE);
  eat(TK_IDENT);
  compileParams();
  eat(SB_SEMICOLON);
  compileBlock();
  eat(SB_SEMICOLON);
}
```