

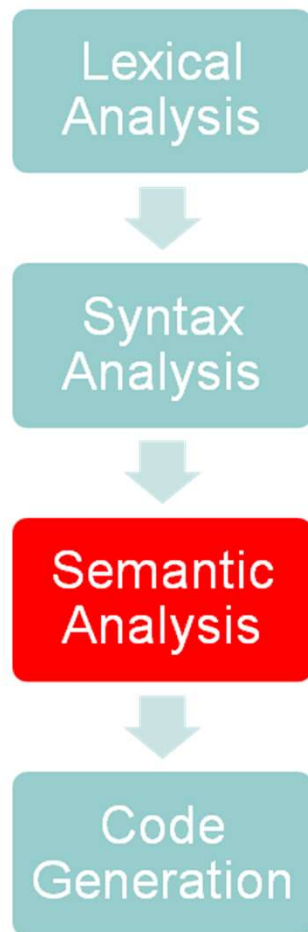
Thực hành  
Xây dựng chương trình dịch

## **Bài 3: Bảng ký hiệu & Quản lý phạm vi**

ONE LOVE. ONE FUTURE.

- 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” là tên một biến hay một hàm?
  - “x” đã được định nghĩa chưa?
  - “x” được định nghĩa ở đâu?
  - Biểu thức “a+b” có kiểu nhất quán không?
  - ...
- Phân tích 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

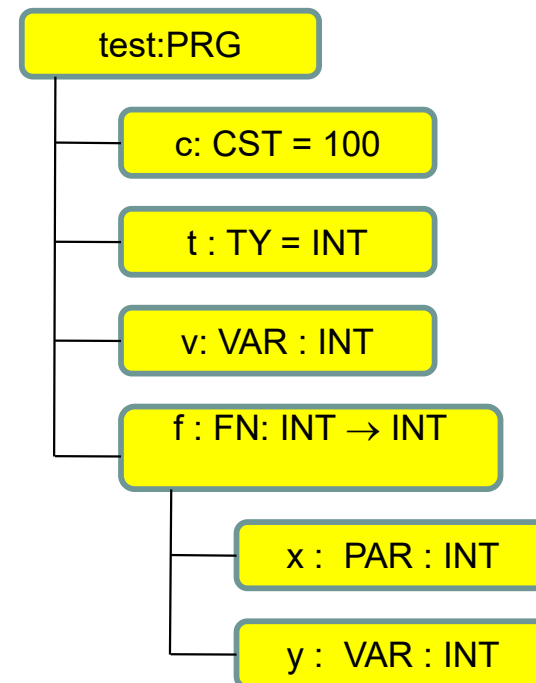
- Quản lý thông tin về các định danh
  - Hằng
  - Biến
  - Kiểu người dùng định nghĩa
  - Chương trình con (hàm, thủ tục)
- Kiểm tra một số luật ngữ nghĩa
  - Phạm vi định danh
  - Nhất quán kiểu

- 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.
```



- 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

- 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;
};
```



- 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) ;
```

# 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;
    };
};
```

- **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);
```

# Lỗi 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;
};
```

## 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();  
    ...  
}
```

- 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()`



- 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 điều 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;
}
```



- 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`

- 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`

- 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: `compileType`
  - 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

- 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

- 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

- Kiểm tra sự trùng lặp khi khai báo đối tượng
- Kiểm tra tham chiếu tới các đối tượng



- Một đối tượng có tên hợp lệ nếu như tên đó chưa từng được sử dụng trong phạm vi hiện tại.
- Sử dụng hàm sau để kiểm tra tên hợp lệ  
`void checkFreshIdent(char *name);`

- Việc kiểm tra một tên hợp lệ được thực hiện khi:
  - Khai báo hằng
  - Khai báo kiểu người dùng định nghĩa
  - Khai báo biến
  - Khai báo tham số hình thức
  - Khai báo hàm
  - Khai báo thủ tục

- Kiểm tra một hằng đã khai báo được thực hiện khi có tham chiếu tới hằng đó
  - Khi duyệt một hằng không dấu
  - Khi duyệt một hằng số
- Lưu ý tới phạm vi của hằng số: nếu hằng không được định nghĩa trong phạm vi hiện tại thì phải tìm kiếm ở những phạm vi rộng hơn
- Giá trị của hằng số đã khai báo sẽ được sử dụng để tạo ra giá trị của hằng số đang duyệt
  - Chia sẻ giá trị hằng → `duplicateConstantValue`

- Kiểm tra một kiểu đã khai báo được thực hiện khi có tham chiếu tới kiểu đó
  - Khi duyệt kiểu: `compileType`
- Lưu ý tới phạm vi của kiểu: nếu kiểu không được định nghĩa trong phạm vi hiện tại thì phải tìm kiếm ở những phạm vi rộng hơn
- Kiểu thực tế của định danh kiểu được tham chiếu sẽ được sử dụng để tạo ra kiểu đang duyệt
  - Chia sẻ → `duplicateType`

- Kiểm tra một biến đã khai báo được thực hiện khi có tham chiếu tới biến đó
  - Trong câu lệnh gán
  - Trong câu lệnh for
  - Trong khi duyệt factor
- Lưu ý tới phạm vi của biến: nếu biến không được định nghĩa trong phạm vi hiện tại thì phải tìm kiếm ở những phạm vi rộng hơn

- Khi một định danh xuất hiện bên trái của biểu thức gán hoặc trong factor, định danh đó có thể tương ứng
  - tên hàm hiện tại
  - Một biến đã khai báo
    - Nếu biến khai báo có kiểu mảng, theo sau tên biến phải có chỉ số của mảng
- Lưu ý phân biệt biến với tham số và tên hàm hiện tại

- Kiểm tra một hàm đã khai báo được thực hiện khi có tham chiếu tới hàm đó
  - Vế trái của lệnh gán (hàm hiện tại)
  - Trong một factor (khi đó cần có danh sách tham số đi kèm)
- Lưu ý tới phạm vi của hàm: nếu hàm không được định nghĩa trong phạm vi hiện tại thì phải tìm kiếm ở những phạm vi rộng hơn
- Một số hàm toàn cục: READC, READI

## Khi tên hàm xuất hiện ở vế trái của lệnh gán...

```
function f(x:integer):integer
```

```
.....
```

```
    function g(y: integer):integer
```

```
    begin
```

```
        f:= y*y (* lenh gán cho ten ham khac- ko hop le*)
```

```
    end;
```

```
.....
```

Tên hàm hiện hành: tên của đối tượng là owner của phạm vi hiện hành





- Kiểm tra một thủ tục đã khai báo được thực hiện khi có tham chiếu tới thủ tục đó
  - Lệnh gọi
- Lưu ý tới phạm vi của thủ tục: nếu thủ tục không được định nghĩa trong phạm vi hiện tại thì phải tìm kiếm ở những phạm vi rộng hơn
- Một số thủ tục toàn cục: WRITEI, WRITEC, WRITELN

- ERR\_UNDECLARED\_IDENT
- ERR\_UNDECLARED\_CONSTANT
- ERR\_UNDECLARED\_TYPE
- ERR\_UNDECLARED\_VARIABLE
- ERR\_UNDECLARED\_FUNCTION
- ERR\_UNDECLARED\_PROCEDURE
- ERR\_DUPLICATE\_IDENT

# Hàm thực hiện luật phạm vi gần nhất: lookupObject

```
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;
}
```

- Hoàn thiện các hàm đánh dấu TODO trong symtab.c
- Tìm hiểu lại cấu trúc của bộ parser (có thay đổi)
- Bổ sung 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

- Lập trình cho các hàm sau trong tệp semantics.c
  - checkFreshIdent
  - checkDeclaredIdent
  - checkDeclaredVariable
  - checkDeclaredProcedure
  - checkDeclaredLValueIdent
- Bổ sung các đoạn code vào những chỗ có đánh dấu TODO hoặc những chỗ cần thiết để thực hiện các công việc kiểm tra khi đối tượng được khai báo hay sử dụng
- Biên dịch và thử nghiệm với các ví dụ mẫu