Thực hành
CHƯƠNG TRÌNH DỊCH
Bài 4: Phân tích ngữ nghĩa

Phạm Đăng Hải
haipd@soict.hut.edu.vn

Ví dụ 1

Cho văn phạm G = (Σ, Δ, P, S)

P: { <Câu> → <Chủ ngữ> <Vị ngữ>

<Chủ ngữ> → <Danh ngữ>|<Danh từ>

<Chủ ngữ> → <Danh ngữ>|<Danh từ>

<Danh ngữ>→ <Danh từ> <Tính từ>

<Vị ngữ> →<Động từ>|<Động từ><Bổ ngữ>

<Bổ ngữ> → <Danh ngữ>

<Danh từ> → « Bò »| « Cỏ »|

<Tính từ> →« Vàng »| « Non »

<Động từ> → « gặm» }

Ví dụ 1

L(G) =

« Bò vàng gặm cỏ non »

« Bò vàng gặm cỏ vàng »

« Bò non gặm cỏ non »

« Bò vàng gặm bò non »

« Cỏ non gặm bò vàng »

…..

Các câu đều đúng ngữ pháp, nhưng không phải câu nào cũng đúng ngữ nghĩa (có ý nghĩa)

Ví dụ 2

```
Program Toto;
Const N = 0;
Begin
      N :=10;
End.
```

<Statement>

⇒ <Variable> := <Expression>

⇒ <Variableidentifier>:=<Expression>

⇒ N:= <Expression>

⇒ N:=<Term>

⇒ N:=<Factor>

⇒ N:=<Unsignedconstant>

⇒ N:=<unsignedinteger>

⇒ N:=10

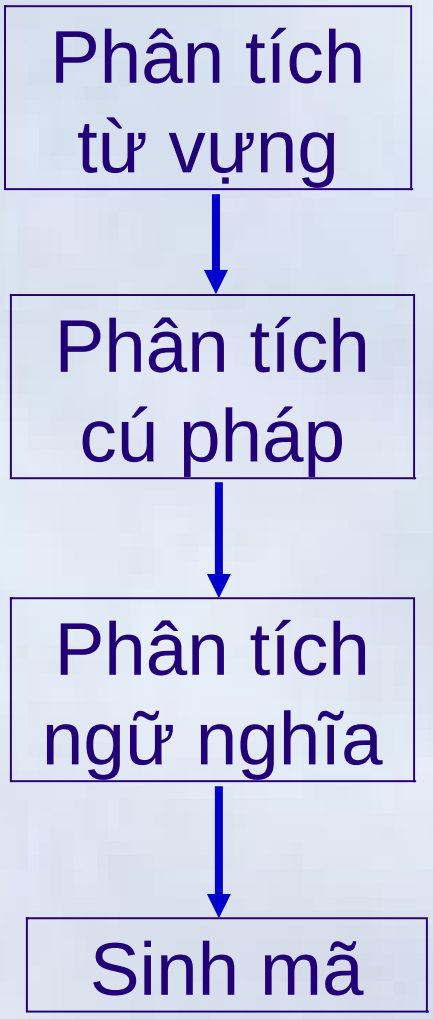
Hoàn toàn đúng cú pháp của KPL

Sử dụng sai ý nghĩa ban đầu (Hằng số)

Nhận xét

- Không phải mọi câu văn (**NNLT**: câu lệnh) đúng ngữ pháp (**NNLT**: cú pháp) đều có giá trị sử dụng (**NNLT**: thực hiện được)
- Bộ phân tích ngữ nghĩa nhằm mục đích kiểm tra tính đúng đắn về mặt ngữ nghĩa của câu văn (**NNLT:** câu lệnh)

Vị trí của bộ phân tích ngữ nghĩa



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

Nhiệm vụ của bộ phân tích ngữ nghĩa

- Quản lý thông tin về các định danh (tên)
  - Hằng, biến, kiểu tự định nghĩa, chương trình con
- Kiểm tra việc sử dụng các định danh
  - Phải được khai báo trước khi dùng
  - Phải được sự dụng đúng mục đích
    - Gán giá trị cho hằng, tính toán trên kiểu, thủ tục…
  - Đảm bảo tính nhất quán
    - Tên được khai báo chỉ một lần trong phạm vi
    - Các phần tử trong kiểu liệt kê (*enum*) là duy nhất

Bảng ký hiệu

Nhiệm vụ của bộ phân tích ngữ nghĩa

- Kiểm tra kiểu dữ liệu cho toán tử
  - Toán tử % của C đòi hỏi toán hạng kiểu nguyên
  - Có thể yêu cầu chuyển kiểu bắt buộc (*int2real*)
  - Chỉ số của mảng phải nguyên
- Kiểm tra sự tương ứng giữa tham số thực sự và hình thức
  - Số lượng tham số, tương ứng kiểu
- Kiểm tra kiểu trả về của hàm..

Các biểu thức kiểu của ngôn ngữ
Bộ luật để định kiểu cho các cấu trúc

Bảng ký hiệu

- 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 khai báo địa phương}
  - Thủ tục: {tên, các tham số hình thức, các khai báo địa phương)
  - Tham số hình thức: {tên, kiểu, tham biến/tham trị}

Bảng ký hiệu

Khi gặp một tên trong chương trình

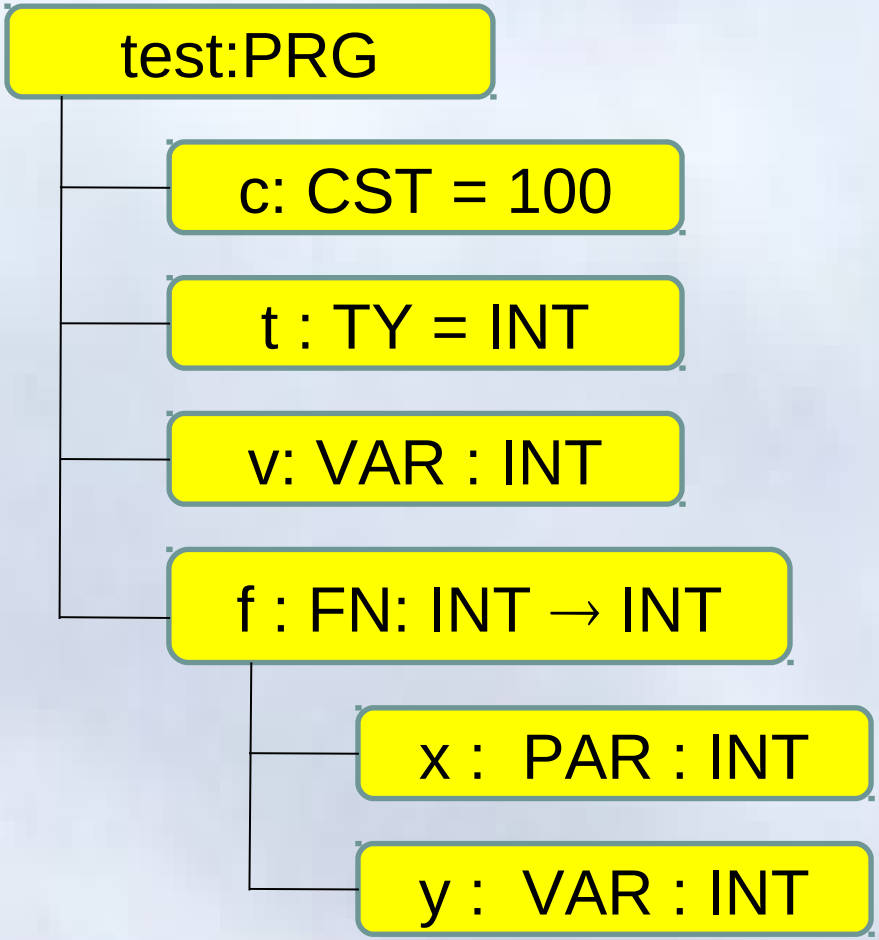
- Gặp trong giai đoạn khai báo
  - Đưa tên và các thông tin tương ứng vào bảng
  - Ví dụ: **Const Max = 10;**
    - Đưa **Max** vào bảng, với kiểu là **constant**, giá trị là **10**;
- Gặp trong câu lệnh
  - Đọc thông tin ra để sử dụng
    - Phân tích ngữ nghĩa: Sử dụng đúng mục đích không?
      - Ví dụ: Max := 20; ← Sai mục đích
    - Sinh mã: Kích thước bộ nhớ cấp phát cho tên
      - Ví dụ: int →2 bytes, float → 4 byte

Bảng ký hiệu trong KPL

Trong chương trình dịch KPL, bảng ký hiệu được biểu diễn theo cấu trúc phân cấp

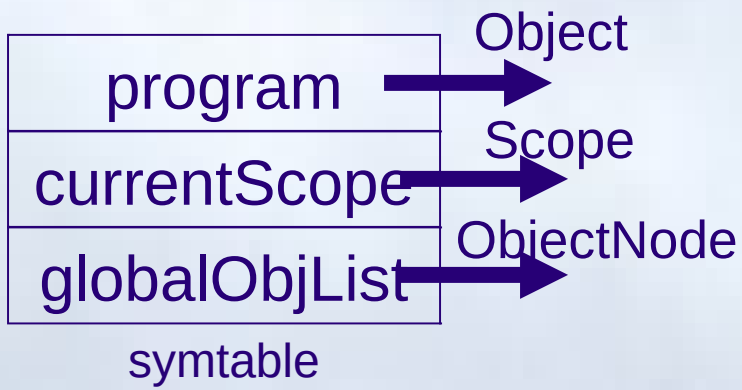
```
PROGRAM test;
CONST c = 100;
TYPE t = Integer;
VAR v : t;
FUNCTION f(x : t) : t;
VAR y : t;
BEGIN
  y := x + 1;
  f := y;
END;

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



Xây dựng bảng ký hiệu→Các thành phần

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

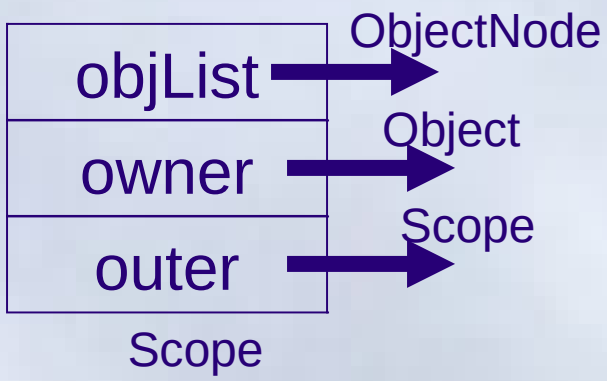


Xây dựng bảng ký hiệu→Các thành phần

```
// 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;
};
```



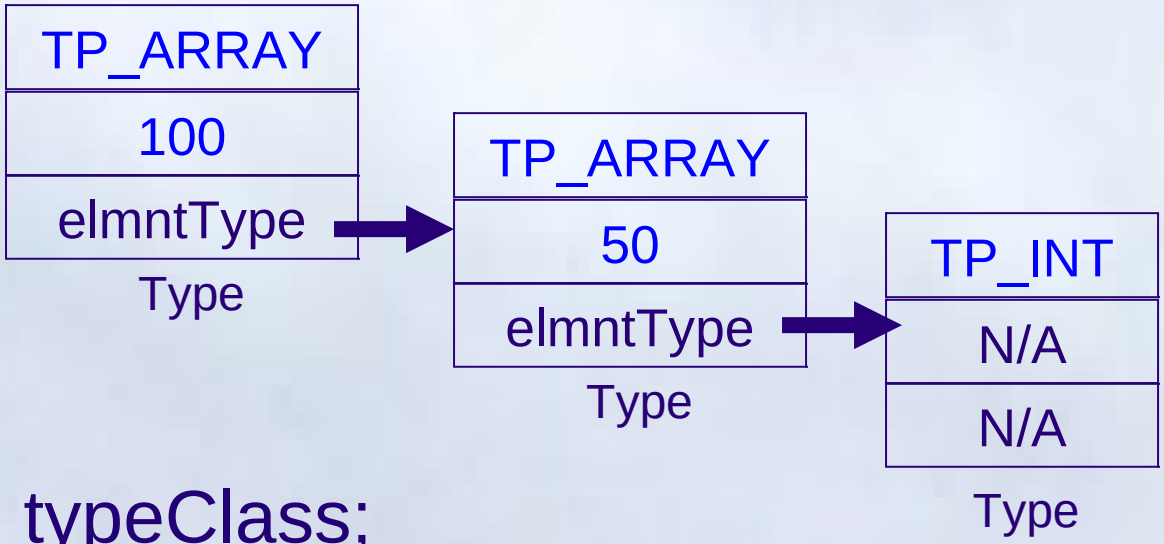
Xây dựng bảng ký hiệu

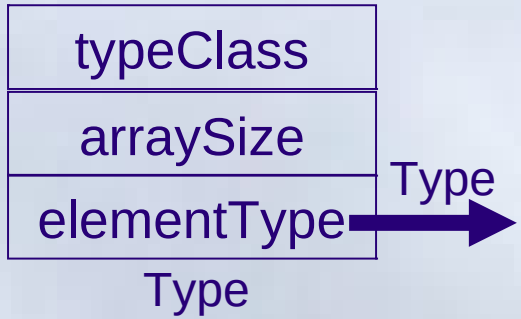
- Bảng ký hiệu ghi nhớ block hiện đang duyệt trong biến currentScope
- Mỗi khi dịch một hàm hay thủ tục, phải cập nhật giá trị của currentScope
- void enterBlock(Scope* scope);
- Mỗi khi kết thúc duyệt một hàm hay thủ tục phải chuyển lại currentScope ra block bên ngoài
- void exitBlock(void);
- Đăng ký một đối tượng vào block hiện tại
- void declareObject(Object* obj);

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;
};
```

ARRAY[100] OF ARRAY[50] OF Integer





Hằng số

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

Hằng số 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

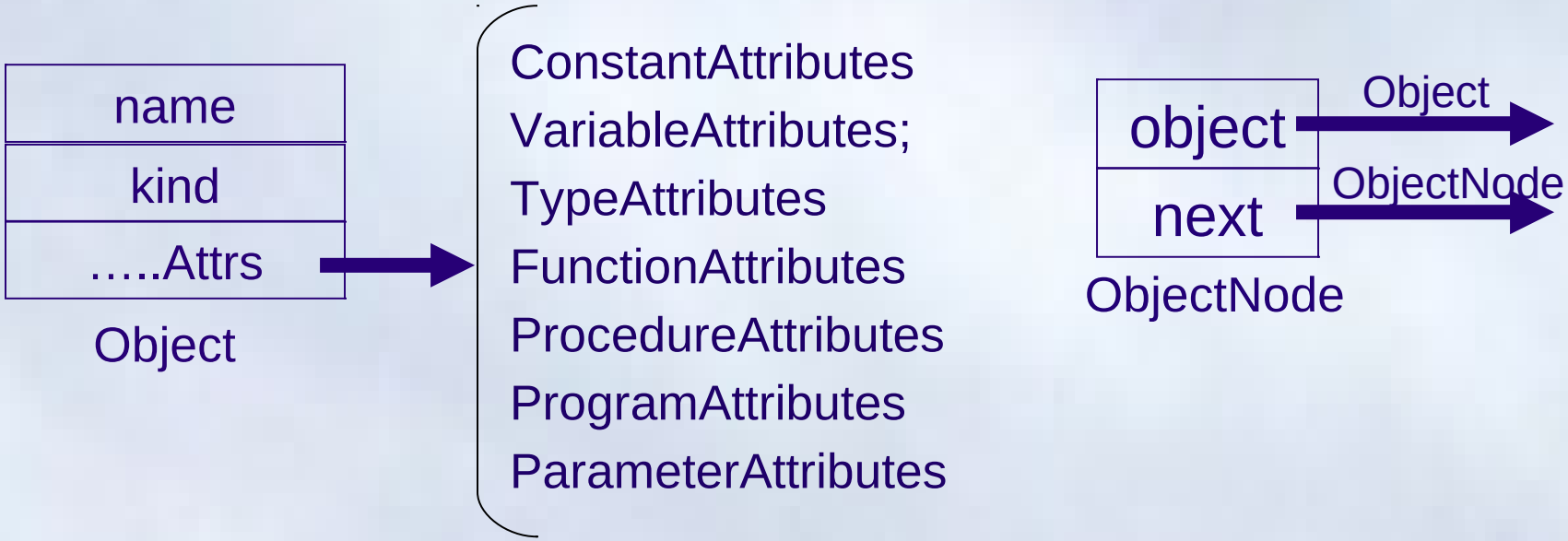
```
ConstantValue* makeIntConstant(int i);
ConstantValue* makeCharConstant(char ch);
ConstantValue* duplicateConstantValue(ConstantValue* v);
```

Đối tượng

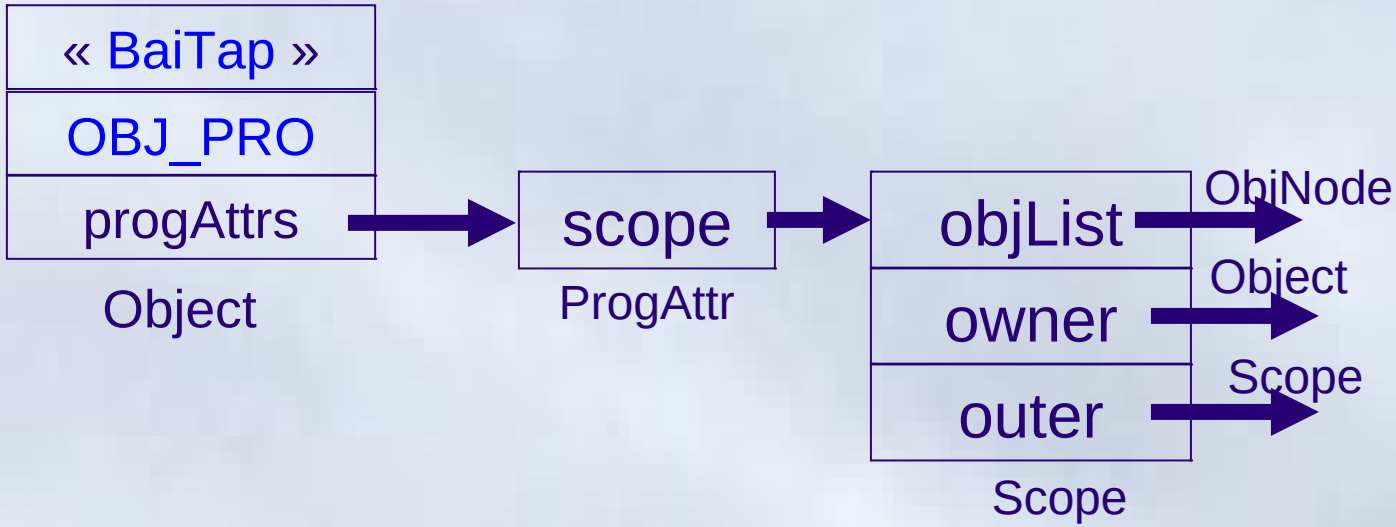
```
// Phân loại ký hiệu
enum ObjectKind {
  OBJ_CONSTANT,
  OBJ_VARIABLE,
  OBJ_TYPE,
  OBJ_FUNCTION,
  OBJ_PROCEDURE,
  OBJ_PARAMETER,
  OBJ_PROGRAM
};
```

```
//Thuộc tính của đối tượng trong bảng
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;
  };
};
```

Đối tượng (tiếp)

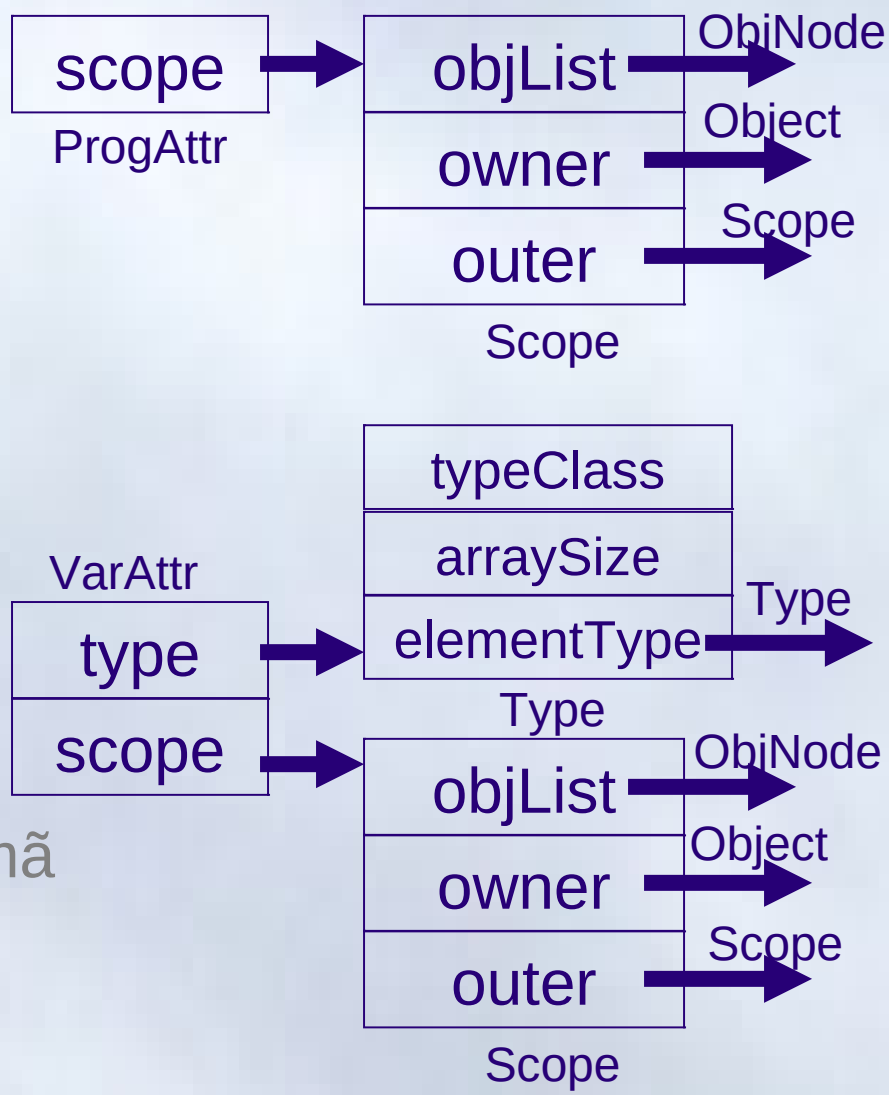


Ví du: Đối tượng: **program BaiTap**



Thuộc tính của đối tượng

```
struct ProgramAttributes_ {
  struct Scope_ *scope;
};
struct ConstantAttributes_ {
  ConstantValue* value;
};
struct VariableAttributes_ {
  Type *type;
   struct Scope_ *scope;
}; // Phạm vi; sử dụng cho sinh mã
struct TypeAttributes_ {
  Type *actualType;
};
```

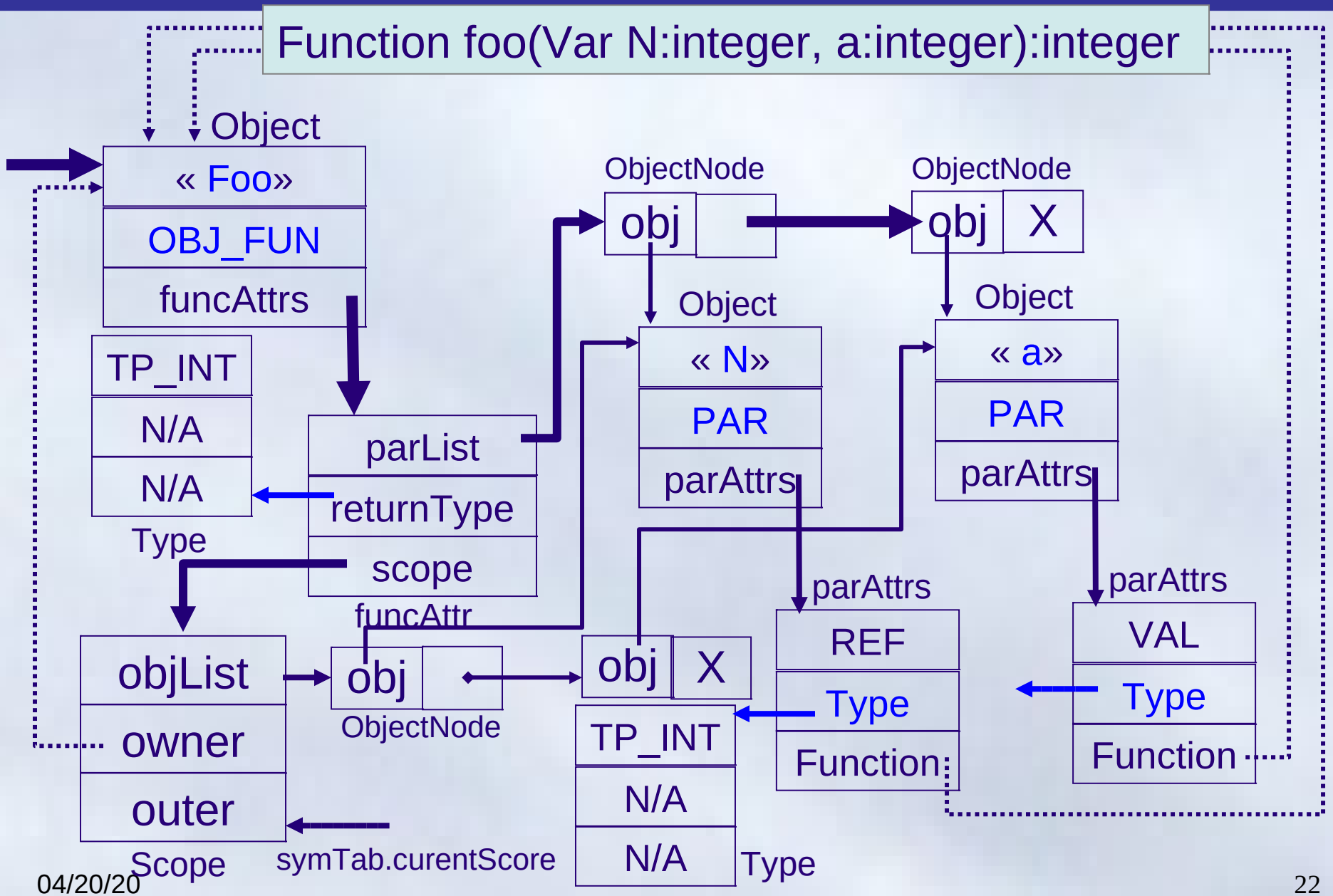


Thuộc tính của đối tượng

```
struct ParameterAttributes_ {
   enum ParamKind kind; // Tham biến hoặc tham trị
  Type* type;
  struct Object_ *function;
};
struct ProcedureAttributes_ {
  struct ObjectNode_ *paramList;
  struct Scope_* scope;
};
struct FunctionAttributes_ {
  struct ObjectNode_ *paramList;
  Type* returnType;
  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)

Ví dụ: Đối tượng hàm



Đối tượng→Các hàm liên quan

Tạo một đối tượng hằng số

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

Tạo một đối tượng kiểu

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

Tạo một đối tượng biến

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

Tạo một đối tượng tham số hình thức

```
Object* createParameterObject(char *name
                enum ParamKind kind;Object* owner);
```

Tạo một đối tượng hàm

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

Tạo một đối tượng thủ tục

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

Tạo một đối tượng chương trình

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

Giải phóng bộ nhớ

Giải phóng kiểu

```
void freeType(Type* type);
```

Giải phóng đối tượng

```
void freeObject(Object* obj)
```

Giải phóng danh sách đối tượng

```
void freeObjectList(ObjectNode* objList)
void freeReferenceList(ObjectNode* objList)
```

Giải phóng block

```
void freeScope(Scope* scope)
```

Hỗ trợ gỡ rối

In thông tin kiểu

```
void printType(Type* type);
```

In thông tin đối tượng

```
void printObject(Object* obj, int indent)
```

In danh sách danh sách đối tượng

```
void printObjectList(ObjectNode* objList, int indent)
```

In block

```
void printScope(Scope* scope, int indent)
```

Indent: khoảng cách in ra so với cột 1 (lề)

Nhiệm vụ ngày thứ nhất

- Cài đặt bảng ký hiệu
- Các tệp mã nguồn
  - Makefile
  - symtab.h, symtab.c
  - debug.h, debug.c
  - main.c
- Hoàn thiện nội dung cho những hàm được đánh dấu TODO trong tệp symtab.c

Phân tích ngữ nghĩa

- Ngày 1:
  - Cài đặt bảng ký hiệu
- Ngày 2:
  - Xây dựng nội dung cho bảng ký hiệu
    - Trong khai báo các đối tượng
- Ngày 3:
  - Kiểm tra trong khai báo
- Ngày 4:
  - Kiểm tra tính nhất quán khi sử dụng

Xây dựng bảng ký hiệu trong KPL

- Khởi tạo và giải phóng
- Khai báo hằng
- Khai báo kiểu
- Khai báo biến
- Khai báo hàm, thủ tục
- Khai báo tham số hình thức

Khởi tạo và giải phóng bảng ký hiệu

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

Khởi tạo chương trình

- Chương trình được khởi tạo tại hàm

  void compileProgram(void);

- Tạo một đối tượng chương trình

  program = createProgramObject(currentToken->string);

- Sau khi khởi tạo chương trình phải chuyển vào block chính bằng hàm *enterBlock()*

  enterBlock(program->progAttrs->scope);

- Dịch một block

  compileBlock();

- Sau khi duyệt xong toàn bộ chương trình, ra khỏi khối bằng hàm *exitBlock()*

  – exitBlock();

Khai báo hằng

- Các đối tượng hằng số được tạo ra và khai báo ở hàm compileBlock()
- Tạo một đối tượng hằng

```
constObj = createConstantObject(currentToken->string);
```

- Giá trị của hằng số được lấy từ quá trình duyệt giá trị hằng qua hàm

```
ConstantValue* compileConstant(void)
```

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

Khai báo kiểu tự định nghĩa

- Các đối tượng kiểu được tạo ra và khai báo ở hàm compileBlock2()
- Tạo một đối tượng kiểu

  typeObj = createTypeObject(currentToken->string);
- 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: lookupObject(currentToken->string)
- Sau khi duyệt xong một kiểu người dùng định nghĩa, phải đăng ký vào block hiện tại bằng hàm declareObject(typeObj)

Khai báo biến

- Các đối tượng biến được tạo ra và khai báo ở hàm compileBlock3()
- Tạo một đối tượng biến

```
varObj = createVariableObject(currentToken->string);
```

- 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(varObj)

Khai báo hàm

- 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 (declareObject)và chuyển block hiện tại sang block của hàm (*enterBlock*) trước khi duyệt tiếp các đối tượng cục bộ. Khi duyệt xong → ra khỏi khối con

Khai báo thủ tục

- Các đối tượng thủ tục được tạo ra và khai báo ở hàm compileProcDecl()
- Tạo đối tượng thủ tục

```
procObj = createProcedureObject(currentToken->string);
```

- Các thuộc tính của đối tượng thủ tục sẽ được cập nhật 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

Khai báo tham số hình thức

- Các đối tượng tham số hình thức được tạo ra và khai báo ở hàm compileParam()
- Tạo đối tượng tham số

  param = createParameterObject()
- Thuộc tính của đối tượng tham số hình thức gồm:
  - Kiểu dữ liệu cơ bản
  - Tham biến (PARAM_REFERENCE) hoặc tham trị (PARAM_VALUE)
- **Lưu ý:** đối tượng tham số hình thức được đăng ký vào đồng thời vào
  - Thuộc tính paramList của hàm/thủ tục hiện tại,
  - Danh sách đối tượng trong phạm vi hiện tại: declareObject()

Nhiệm vụ ngày thứ hai

- 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

Kết quả ví dụ

```
Program PRG
    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

Press any key to continue . . . _
```

Phân tích ngữ nghĩa

- Ngày 1:
  - Cài đặt bảng ký hiệu
- Ngày 2:
  - Xây dựng nội dung cho bảng ký hiệu
    - Trong khai báo các đối tượng
- Ngày 3:
  - Kiểm tra trong khai báo
- Ngày 4:
  - Kiểm tra tính nhất quán khi sử dụng

Kiểm tra trong khai báo

1. Kiểm tra sự trùng lặp khi khai báo đối tượng
2. Kiểm tra tham chiếu tới các đối tượng

Kiểm tra tên hợp lệ

- Tên là hợp lệ nếu như chưa từng được khai báo trong phạm vi hiện tại.
- Để kiểm tra tên hợp lệ, sử dụng hàm

  void checkFreshIdent(char *name)

- Kiểm tra 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 hằng số 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 ⇒ 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
  - Không chia sẻ → duplicateConstantValue

Kiểm tra kiểu đã khai báo

- Được thực hiện khi có tham chiếu tới kiểu đó
  - Khi duyệt kiểu: compileType
- Lưu ý phạm vi của kiểu:
  - Nếu kiểu không được định nghĩa trong phạm vi hiện tại ⇒ 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
  - Không chia sẻ → duplicateType

Kiểm tra biến đã khai báo (1/2)

- 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 ⇒ tìm kiếm ở những phạm vi rộng hơn

Kiểm tra biến đã khai báo (2/2)

- Một định danh xuất hiện bên trái của biểu thức gán hoặc trong factor, có thể là
  - Tên hàm hiện tại
  - Một biến đã khai báo
    - Nếu biến 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 hàm đã khai báo

- Được thực hiện khi có tham chiếu tới hàm
  - Vế trái của lệnh gán (hàm hiện tại)
  - Trong một factor
    - 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 ⇒ tìm kiếm ở những phạm vi rộng hơn
- Một số hàm toàn cục: READC, READI

Kiểm tra thủ tục đã khai báo

- Được thực hiện khi có tham chiếu tới thủ tục
  - Lệnh gọi
- Lưu ý tới phạm vi của thủ tục:
  - Nếu thủ tục không được định nghĩa trong phạm vi hiện tại ⇒ tìm kiếm ở những phạm vi rộng hơn
- Một số thủ tục toàn cục:

  WRITEI, WRITEC, WRITELN

Các mã lỗi

- ERR_UNDECLARED_IDENT
- ERR_UNDECLARED_CONSTANT
- ERR_UNDECLARED_TYPE
- ERR_UNDECLARED_VARIABLE
- ERR_UNDECLARED_FUNCTION
- ERR_UNDECLARED_PROCEDURE
- ERR_DUPLICATE_IDENT

Nhiệm vụ ngày thứ ba

- Lập trình các hàm sau trong tệp **semantics.c**
  - checkFreshIdent()
  - checkDeclaredIdent()
  - checkDeclaredConstant()
  - checkDeclaredType()
  - checkDeclaredVariable()
  - checkDeclaredProcedure()
  - checkDeclaredFunction()
  - checkDeclaredLValueIdent()
    - Variable / Parameter/Function (cùng phạm vi)
- Biên dịch và thử nghiệm với các ví dụ mẫu

Phân tích ngữ nghĩa

- Ngày 1:
  - Cài đặt bảng ký hiệu
- Ngày 2:
  - Xây dựng nội dung cho bảng ký hiệu
    - Trong khai báo các đối tượng
- Ngày 3:
  - Kiểm tra trong khai báo
- Ngày 4:
  - Kiểm tra tính nhất quán của ký hiệu

Kiểm tra tính nhất quán

- Kiểm tra tính nhất quán về kiểu trong các cấu trúc chương trình
  - Nhất quán trong các câu lệnh gán
  - Định nghĩa biến mảng và sử dụng biến mảng
  - Trong định nghĩa hàm và sử dụng hàm
  - Trong định nghĩa thủ tục và lời gọi thủ tục
  - Trong việc sử dụng tham biến

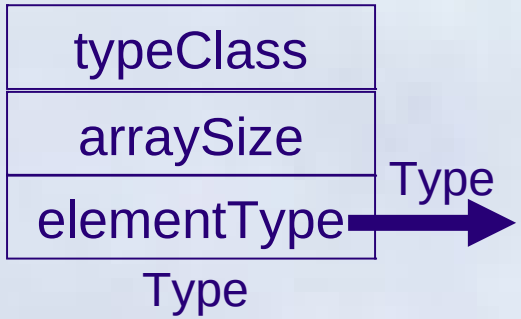
Các hàm so sánh kiểu

Cần xây dựng các hàm kiểm tra kiểu

```
enum TypeClass{
  TP_INT,
  TP_CHAR,
  TP_ARRAY
};
```

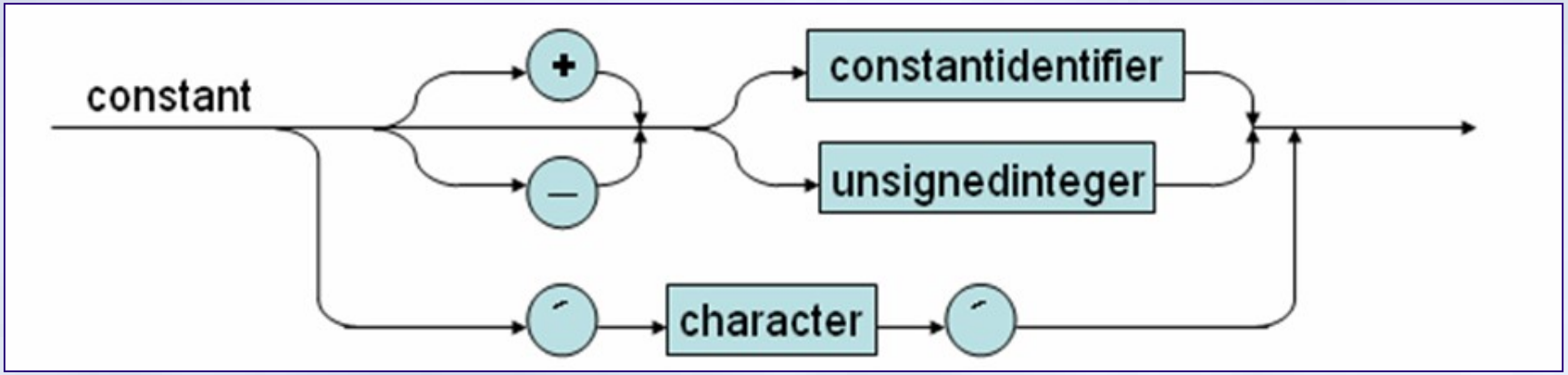
- checkIntType(Type * t)

```
if( (t != NULL) &&(t->typeClase ==TP_INT) )
   return;
else
   Error("Not Integer type")
```



- checkCharType(Type * t)
- checkArrayType(Type * t)
- checkBasicType(Type * t)
  - Kiểu tham số, kiểu hàm phải là TP_INT/TP_CHAR
- checkTypeEquality(Type *t1, Type * t2)

Duyệt hằng



if (Token == [SB_PLUS, SB_MINUS])

Eat(SB_PLUS)/ Eat(SB_MINUS)

if(Token == Ident)

- Kiểm tra Ident đã được khai báo

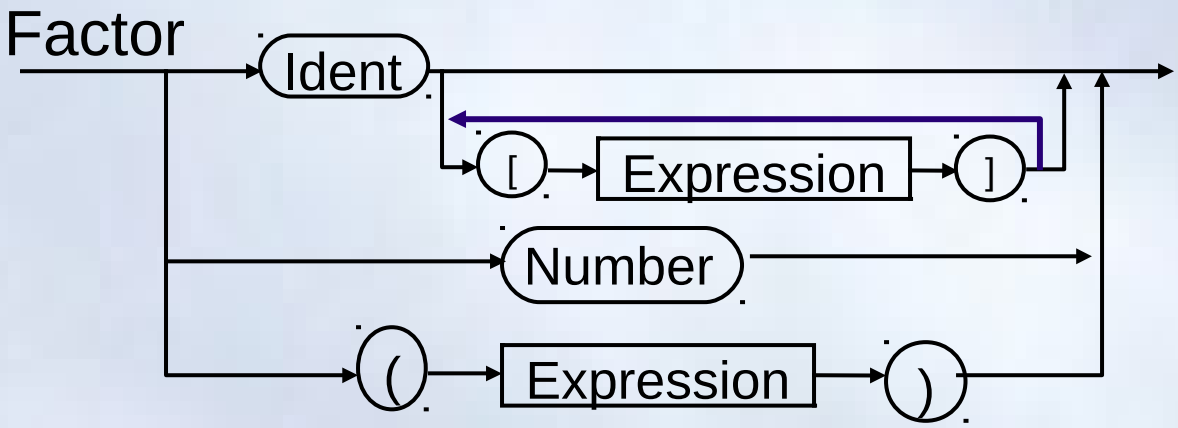
  obj = checkDeclaredConstant(currentToken->string);

- Nếu đã khai báo, phải có kiểu nguyên

  obj->constAttrs->value->type == TP_INT

```
CONST
    MAX = 100;
    MIN = -MAX;
```

Kiểu của nhân tố



Type * compileFactor()

Nếu Token == NUMBER ⇒ return intType

Nếu Token == CHAR ⇒ return charType

Nếu Token == IDENT

Ident đã khai báo?→ obj = checkDeclaredIdent(….)

obj->kind = OBJ_CONST return kiểu của hằng

obj->kind = OBJ_VAR

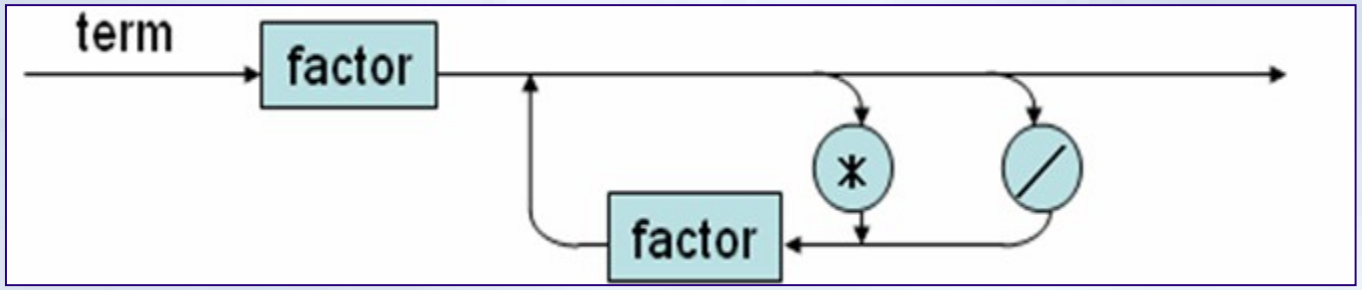
Biến mảng: compileIndexs()

return Kiểu của biến

obj->kind = OBJ_FUNCTION return Kiểu trả về của hàm

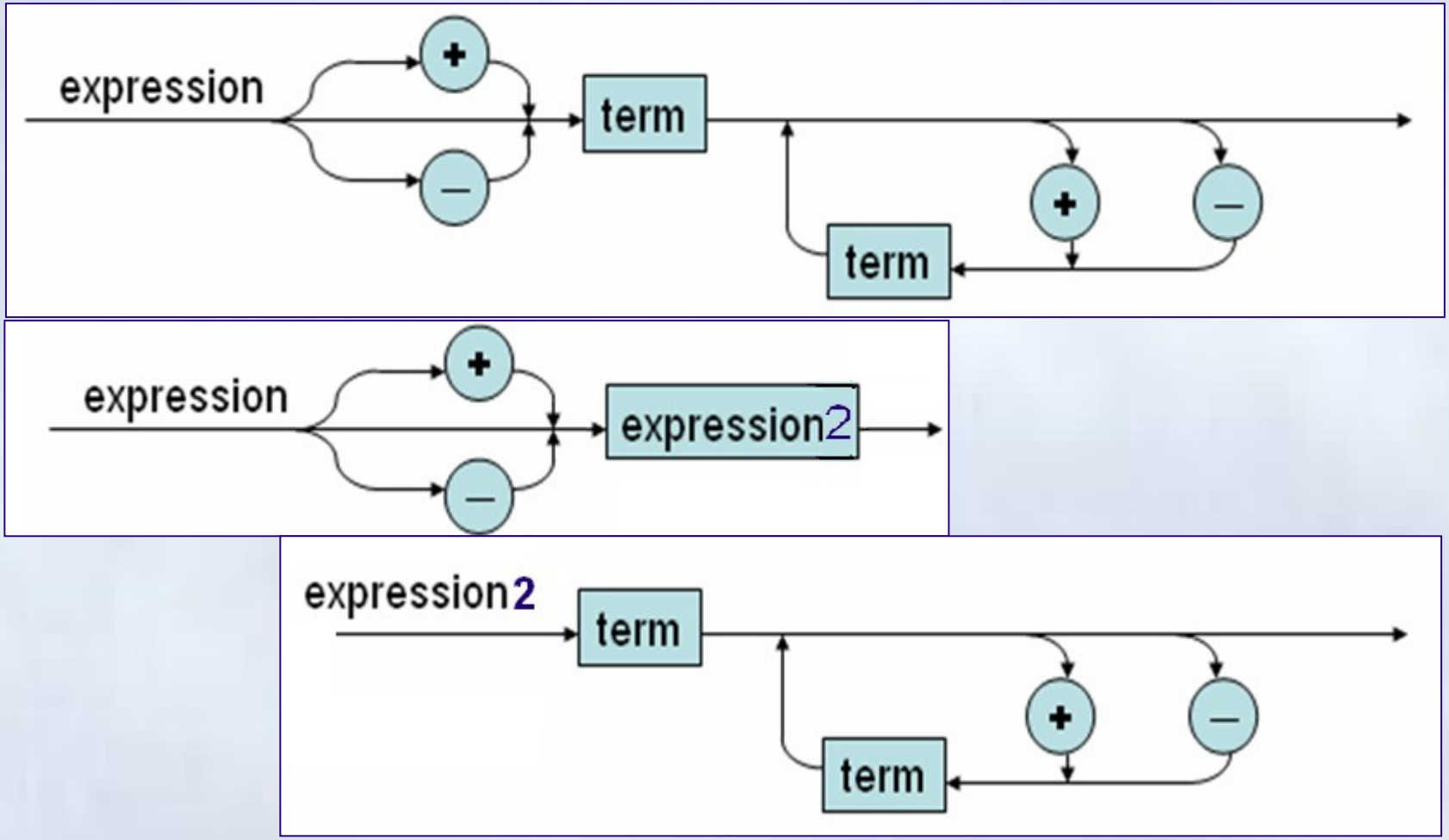
obj->kind = OBJ_PARAM return Kiểu của hàm

Kiểu toán hạng




```
Type * compileTerm()
    Type * t1 = compileFactor()
    while (Token == [SB_TIMES, SB_SLASH] )
    EAT (SB_TIMES) / EAT(SB_SLASH)
    Type * t2 = compileFactor()//
    Kiểm tra t1, t2 cùng kiểu nguyên //checkIntType()
    return t1
```

Duyệt Biểu thức

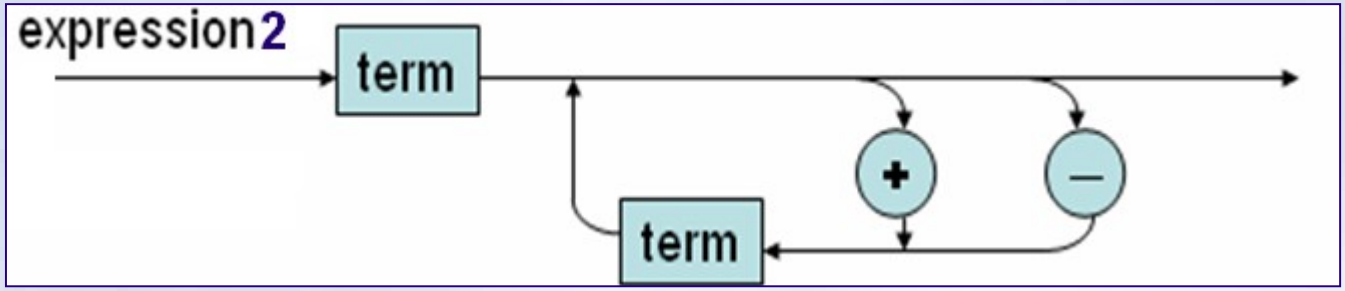


if (Token == [SB_PLUS, SB_MINUS])

Type * t = compileExpression2()

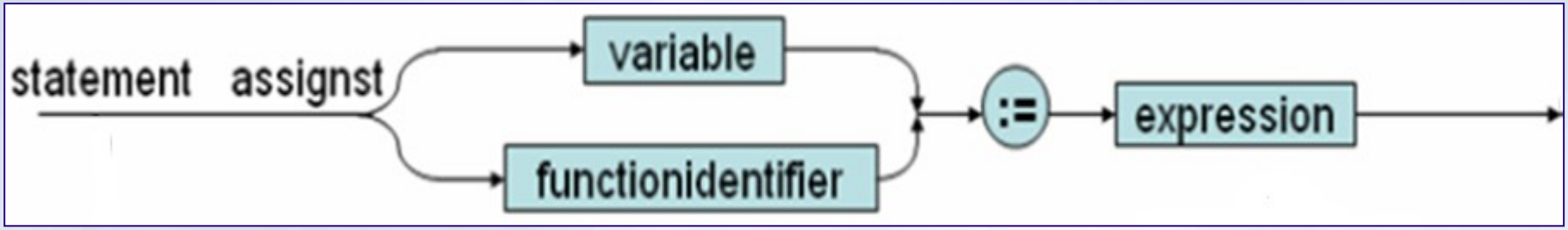
Kiểu của t phải là nguyên: checkIntType(t)

Kiểu biểu thức




```
Type * compileExpression2()
   Type * t1 = compileTerm()
   while (Token == [SB_PLUS, SB_MINUS] )
   eat(SB_PLUS)/Eat(SB_MINUS)
    t2 phải là kiểu nguyên
   Type t2 =compileTerm()
   t2 phải là kiểu nguyên
    return t1
```

Câu lệnh gán



- Bên phải và bên trái của câu lệnh gán phải có cùng kiểu cơ bản
  - Ghi nhận kiểu của vế trái phép gán
    - Type* t1 = compileLValue(void) ←Biến, tham số, hàm
  - Eat(SB_ASSIGN)// Đọc ký hiệu gán
  - Ghi nhận kiểu của Expression
    - Type * t2 = compileExpression();
  - So sánh kiểu tương đương
    - checkTypeEquality(t1, t2)

LValue

Object* **checkDeclaredLValueIdent**(char* name)

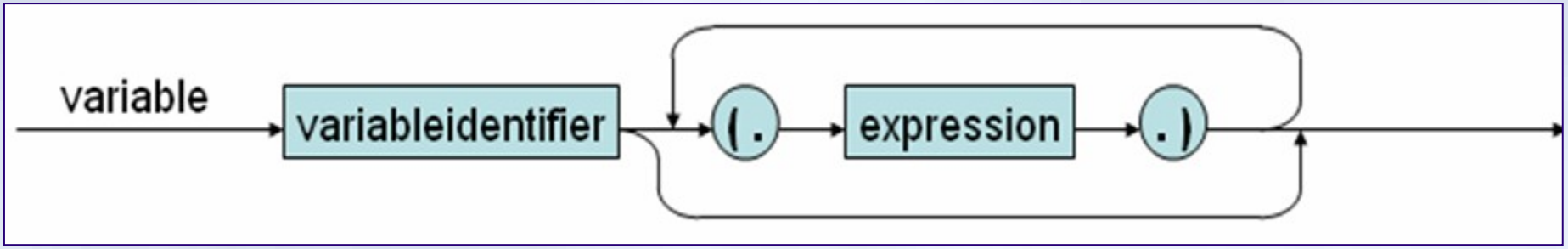
- Name đã được khai báo?

  → Object * lookupObject(Name)
- Kiểu của tên phải là: Biến, Tham số, Hàm.
  - Thuộc tính kind của đối tượng Object
- Nếu là hàm, phải trong pham vi hiện thời
  - Đối tượng của phạm vi hiện tại: symtab->currentScope->owner

Type * **compileLValue**()

- Kiểm tra là LValueIdent → checkDeclaredLValueIdent()
- Nếu là Biến
  - Biến mảng: Kiểu phần tử của mảng → compileIndexs()
  - Không phải mảng: Kiểu của biến: varAttrs->type
- Nếu là Tham số: Kiểu của tham số: paramAttrs->type
- Nếu là Hàm số: Kiểu của hàm số: …

Chỉ số mảng




```
Type* compileIndexes(Type* arrayType)
   while (Token==SB_LSEL)
   eat(SB_LSEL)
   Type * t = compileExpression();
   checkIntType(t) //kiểu của biểu thức là nguyên
   Nếu mảng nhiều chiều, giảm số chiều đi
            arrayType = arrayType->elementType;
   Eat(SB_RSEL)
    return arrayType;//Kiểu phần tử của mảng
```

Câu lệnh For

For <var> := <exp1> To <exp2> Do <stmt>

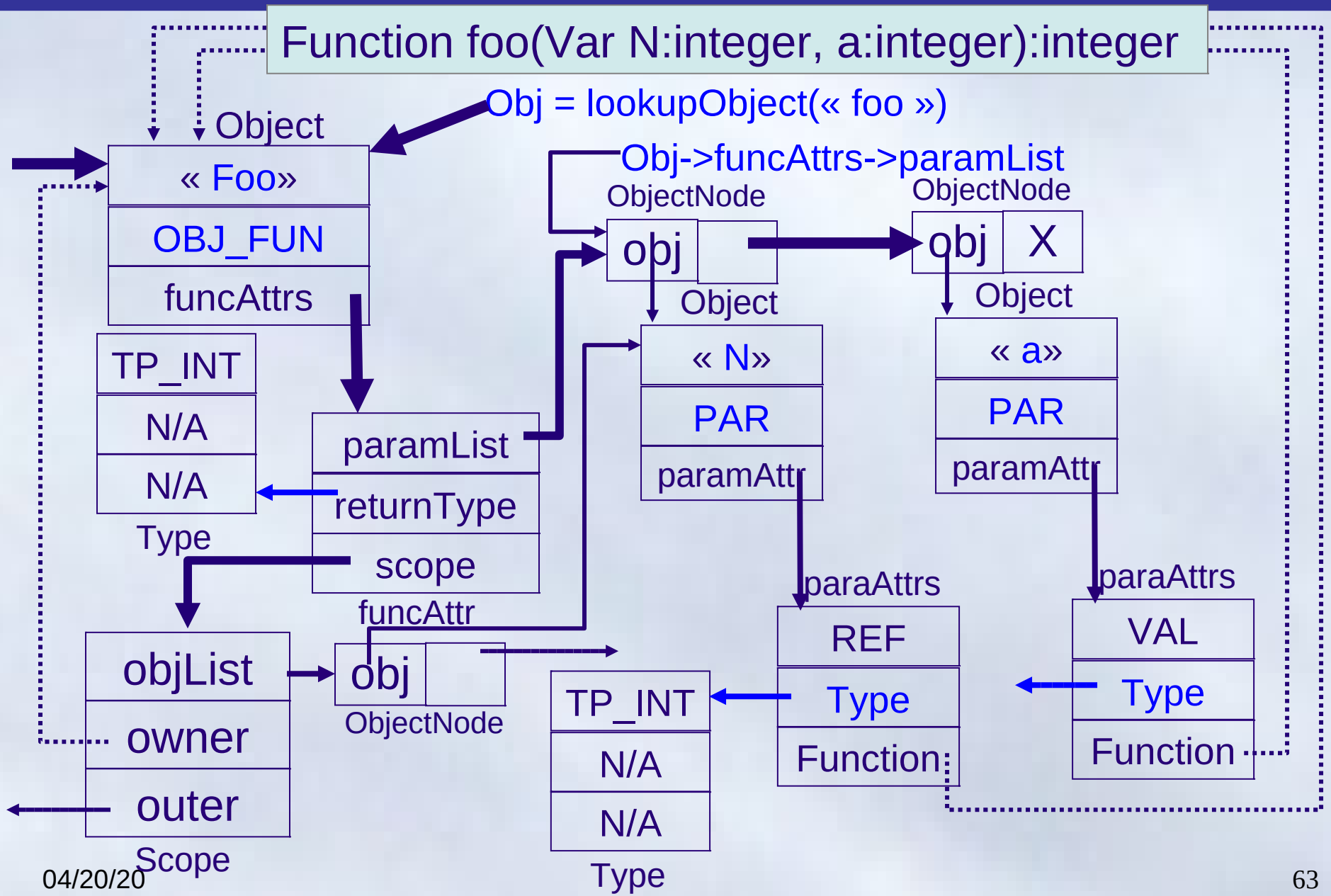
<var> ,<exp1> ,<exp2> phải cùng kiểu cơ bản

- Eat(KW_FOR)
- Eat(Ident)
- Ident là một biến? → Ghi nhận kiểu của Ident
- Eat(SB_ASSIGN)
- Ghi nhận kiểu của Expression() (Type * compileExpression())
- So sánh kiểu tương đương (checkTypeEquality())
- Eat(KW_TO)
- Ghi nhận kiểu của Expression() (Type * compileExpression())
- So sánh kiểu tương đương (checkTypeEquality())
- Eat(KW_DO)

Gọi thủ tục, hàm

- Gọi thủ tục trong câu lệnh Call
  - If (lookAhead->TokenType == KW_CALL)
    - Eat(KW_CALL); Eat(Ident);
    - Ident đã khai báo là thủ tục?
      - Proc=CheckDeclaredProcedure(curentTokent->String)
    - compileArguments(Proc->procAttrs->paramList)
- Gọi hàm ra sử dụng trong Factor
  - If (lookAhead->TokenType == IDENT)
    - Eat(IDENT)
    - Ident đã khai báo ?→ obj = checkDeclaredIdent()
    - Nếu Ident là hàm ? obj->kind == OBJ_FUNCTION
      - compileArguments(obj->funcAttrs->paramList)
      - type = obj - >funcAttrs-> returnType// Kiểu của Factor

Duyệt tham số hàm/thủ tục



Duyệt tham số hàm/thủ tục

- Tham số hình thức và t/số thực sự phải trùng kiểu
  - void compileArguments(ObjectNode* paramList)
  - void compileArgument(Object* param)
- Tham số hình thức: param->paramAttrs->type
- Tham số thực sự: Type * compileExpression()
- Nếu tham số hình thức là tham biến thì tham số thực tế phải là một biến (LValue)
  - Tham biến
    - param->paramAttrs->kind ==PARAM_REFERENCE
  - Kiểu của tham biến param->paramAttrs->type
  - Tham số truyền vào phải là LValue:
    - Sử dụng hàm Type * compileLValue() để kiểm tra;

Duyệt Điều kiện

<Exp1> Op <Exp2>

- Exp1 và Exp2 có cùng kiểu cơ bản
  - Lấy kiểu của biểu thức <Exp1>
    - Type * t1 = compileExpression();
  - Kiểu tra t1 là kiểu cơ bản
    - checkBasicType(t1)
  - Đọc toán tử quan hệ
    - Eat(Op)
  - Lấy kiểu của biểu thức <Exp2>
    - Type * t2 = compileExpression();
  - Kiểm tra tương thích kiểu

Nhiệm vụ ngày thứ tư

- Lập trình cho các hàm trong semantics.c
  - void checkIntType(Type* type);
  - void checkCharType(Type* type);
  - void checkArrayType(Type* type);
  - void checkBasicType(Type* type);
  - void checkTypeEquality(Type* t1, Type* t2);
- Bổ sung các đoạn mã kiểm tra kiểu trong bộ parser tương ứng với các luật kiểm tra trên
- Biên dịch và thử nghiệm với các ví dụ mẫu