



Bài 10

Phân tích ngữ nghĩa



Nội dung

- **Những vấn đề ngữ nghĩa**

- **Bảng ký hiệu**

- ☐ Luật về phạm vi ảnh hưởng của biến
- ☐ Các sơ đồ dịch để xây dựng bảng ký hiệu

- **Kiểm tra kiểu (Type checking)**

- ☐ Hệ thống kiểu trong ngôn ngữ lập trình
- ☐ Đặc tả một bộ kiểm tra kiểu
- ☐ Chuyển đổi kiểu



Phân tích ngữ nghĩa

- Ngữ nghĩa: Ý nghĩa thực sự của các cấu trúc ngôn ngữ
- Tìm ra các lỗi sau giai đoạn phân tích cú pháp, ví dụ:
 - Không tương thích về kiểu (vế trái và vế phải lệnh gán)
 - Không tương ứng giữa việc sử dụng hàm, biến với khai báo của chúng
 - Một biến (hàm) được sử dụng ngoài phạm vi ảnh hưởng trong chương trình
- Phân tích ngữ nghĩa sử dụng cây phân tích cú pháp



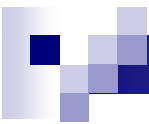
Bảng ký hiệu

- Định danh đại diện cho hàm, biến, hằng...
- Phần lớn các ngôn ngữ lập trình yêu cầu khai báo trước khi sử dụng
- Mỗi định danh có thể được sử dụng nhiều lần trong chương trình
- Cần thiết lập bảng ký hiệu lưu trữ thông tin về các định danh
- Thông tin mỗi định danh gồm: tên, kiểu, phạm vi và kích cỡ bộ nhớ cần phân phối
- Bổ sung dữ liệu vào bảng ký hiệu khi phân tích khai báo
- Tìm kiếm trong bảng ký hiệu khi sử dụng



Cấu trúc dữ liệu cho bảng ký hiệu

- Thích hợp với các cấu trúc dữ liệu động
 - Thường dùng danh sách tuyến tính và bảng băm
 - Mỗi lối vào có dạng bản ghi với một trường cho mỗi loại thông tin
 - Mỗi kiểu đối tượng cần lưu trữ những loại thông tin khác nhau



Các trường cần lưu trữ với các kiểu đối tượng khác nhau

// 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 một số đối tượng

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

```
struct VariableAttributes_ {  
    Type *type;  
    // Phạm vi của biến  
    struct Scope_ *scope;  
};
```

```
struct TypeAttributes_ {  
    Type *actualType;  
};
```

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



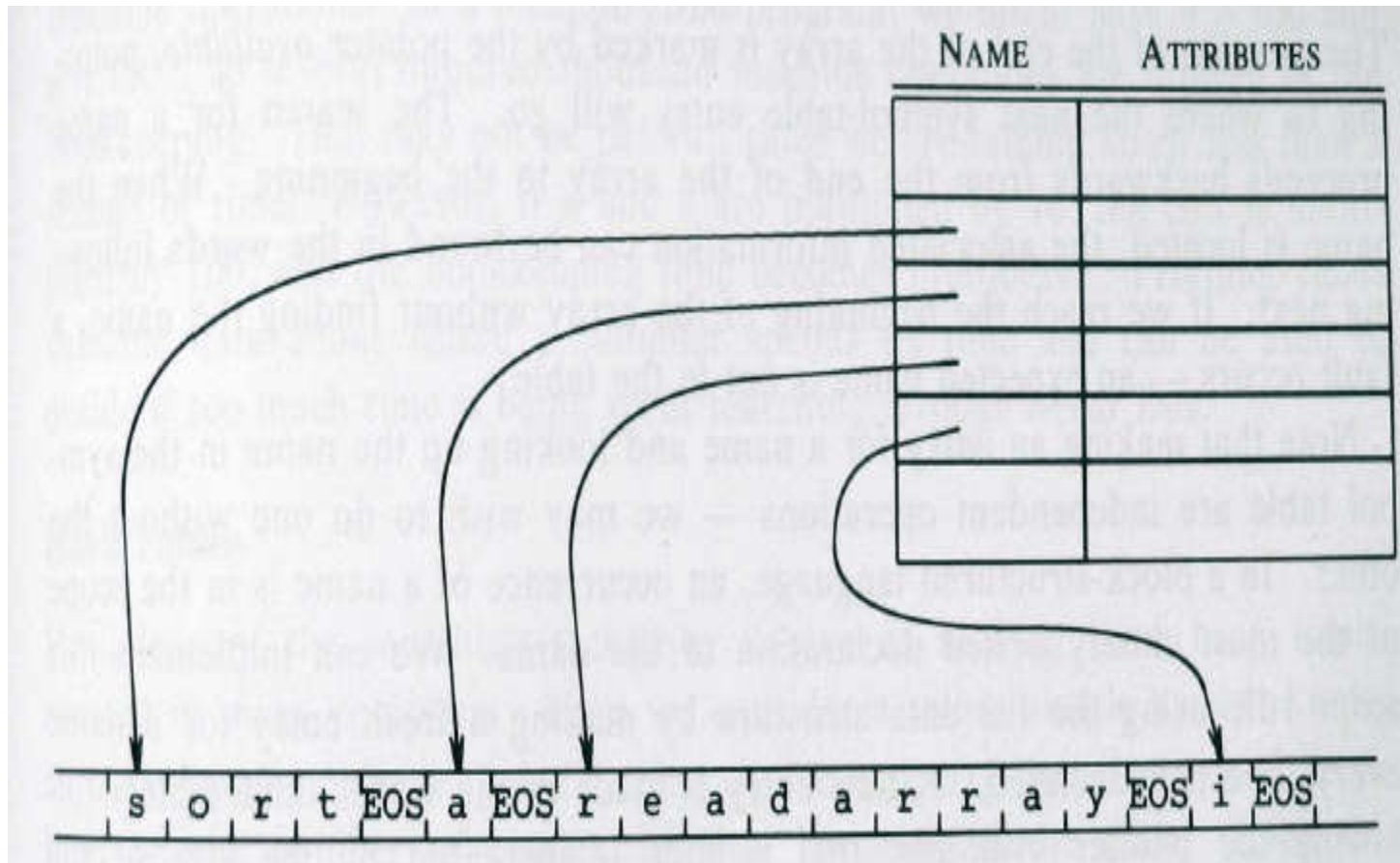
Thuộc tính của một số đối tượng (tiếp)

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

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

```
struct ProgramAttributes_ {  
    struct Scope_ *scope;
```


Liên kết khai báo – sử dụng



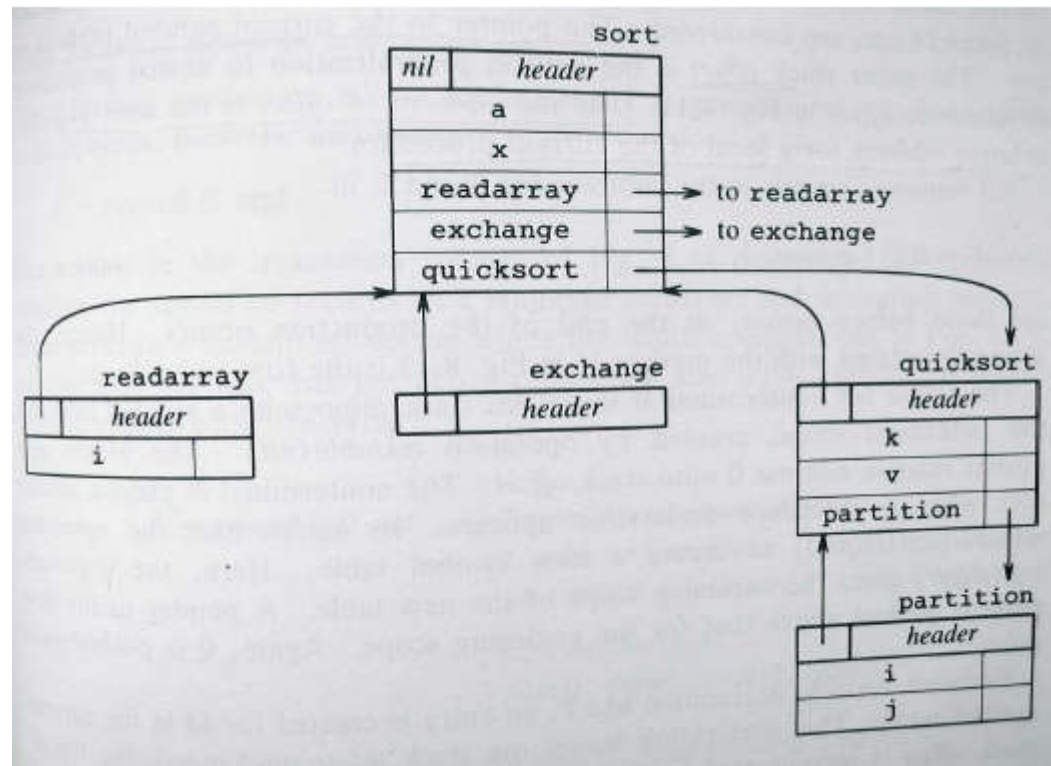


Bảng ký hiệu và các luật về phạm vi

- Khối trong ngôn ngữ lập trình là tập các cấu trúc ngôn ngữ có chứa khai báo
- Một ngôn ngữ là có cấu trúc khối nếu
 - Các khối được lồng bên trong những khối khác
 - Phạm vi của khai báo trong mỗi khối là chính khối đó và các khối chứa trong nó
- Luật lồng nhau gần nhất
 - Cho nhiều khai báo của cùng một tên. Khai báo có hiệu lực là khai báo nằm trong khối gần nhất

Giải pháp nhiều bảng ký hiệu

- Các bảng cần được kết nối từ phạm vi trong ra phạm vi ngoài và ngược lại

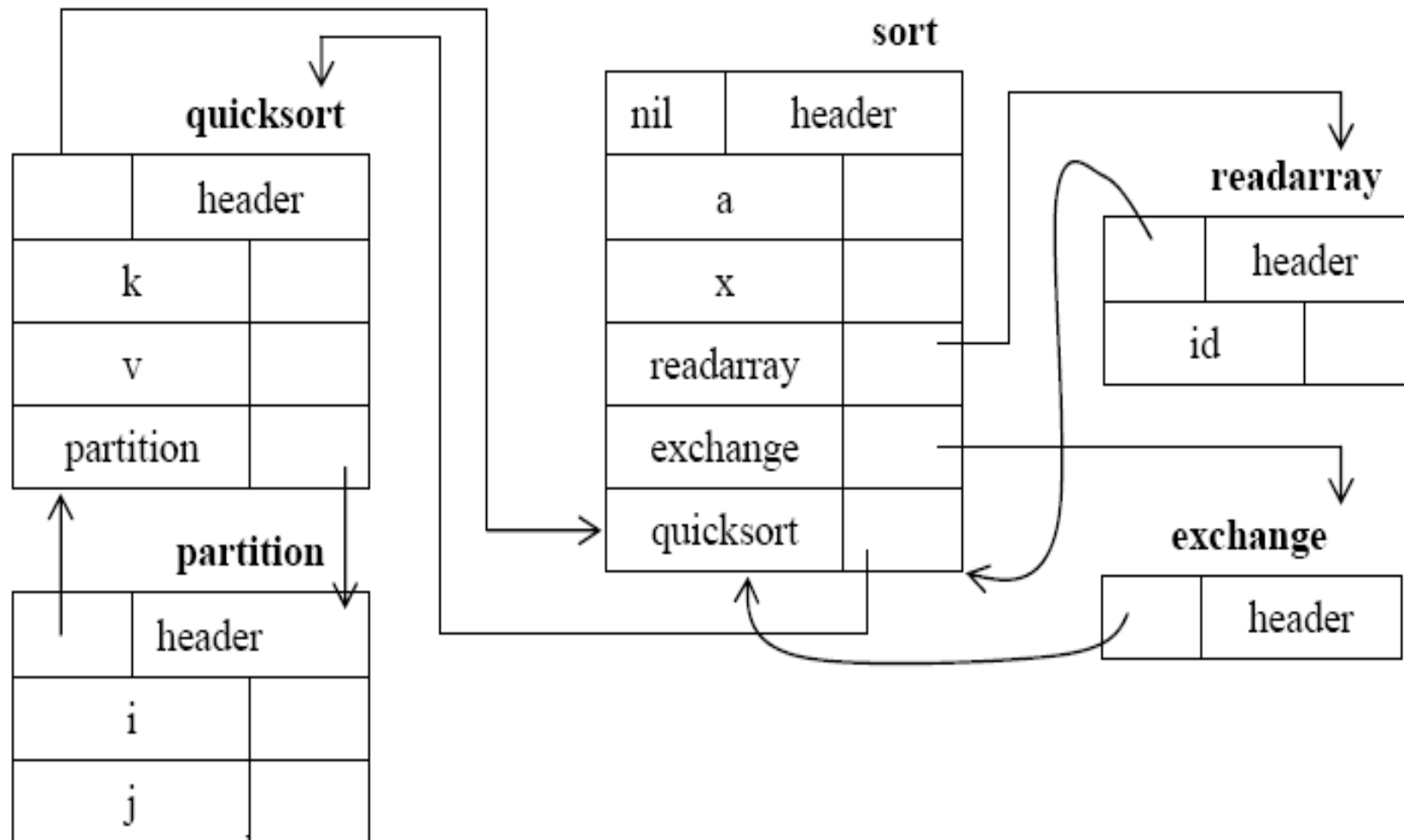




Ví dụ chương trình con lồng nhau

- 1) Program sort;
- 2) *Var a: array[0..10] of integer;*
- 3) *x: integer;*
- 4) Procedure readarray;
- 5) *Var i: integer;*
- 6) *Begin end {readarray};*
- 7) Procedure exchange(i, j: integer);
- 8) *Begin end {exchange};*
- 9) Procedure quicksort(m, n: integer);
- 10) *Var k, v: integer;*
- 11) Function partition(y,z: integer): integer;
- 12) *Beginexchange(i,j) end; {partition}*
- 13) *Begin ... end; {quicksort}*
- 14) *Begin ... end; {sort}*

Năm bảng kí hiệu của Sort

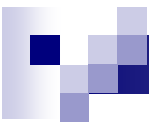




● Các thành phần của bảng ký hiệu

```
// Bảng ký hiệu của chương trình
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 khối
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 trong giai đoạn phân tích cú pháp

- Chỉ có thể bắt đầu nhập thông tin vào bảng ký hiệu từ khi phân tích từ vựng nếu ngôn ngữ lập trình không cho khai báo tên trùng nhau.
- Nếu cho phép dùng tên trùng nhau trong các phạm vi khác nhau, bộ phân tích từ vựng chỉ trả ra tên của định danh cùng với loại token
- Định danh được thêm vào bảng ký hiệu khi vai trò cú pháp của định danh được phát hiện

Tính địa chỉ logic của đối tượng

$P \rightarrow \{\text{offset} := 0\} D$

$D \rightarrow D; D$

$D \rightarrow \text{id} : T \quad \{ \text{enter}(\text{id.name}, T.\text{type}, \text{offset});$
 $\quad \text{offset} := \text{offset} + T.\text{width} \}$

$T \rightarrow \text{int} \quad \{ T.\text{type} := \text{int}; T.\text{width} := 4 \}$

$T \rightarrow \text{real} \quad \{ T.\text{type} := \text{real}; T.\text{width} := 8 \}$

$T \rightarrow \text{array}[\text{num}] \text{ of } T_1 \quad \{ T.\text{type} := \text{array}(\text{num.val}, T_1.\text{type});$
 $\quad T.\text{width} := \text{num.val} * T_1.\text{width} \}$

$T \rightarrow \uparrow T_1 \quad \{ T.\text{type} := \text{pointer}(T_1.\text{type}); T.\text{width} := 4 \}$



Biến offset

- Biến toàn cục offset lưu vết của địa chỉ còn rồi tiếp theo
- Trước khai báo đầu tiên, offset được gán giá trị 0.
- Mỗi khi tìm thấy một định danh, offset được tăng lên



Lưu trữ thông tin về các phạm vi lồng nhau

- Xét các thủ tục lồng nhau: Khi một thủ tục nằm trong thủ tục khác được gọi, các khai báo của thủ tục bên ngoài tạm dừng hoạt động
- Dùng stack để lưu trữ dấu vết của các thủ tục lồng nhau
- Tạo bảng ký hiệu mới cho mỗi thủ tục
 - Khi thêm một định danh mới vào bảng ký hiệu, cần chỉ rõ bảng ký hiệu cần thêm

Xử lý các khai báo trong thủ tục lồng nhau

$P \rightarrow \mathbf{M} D$	$\{addwidth(top(tblptr), top(offset));$ $pop(tblptr); pop(offset)\}$
$\mathbf{M} \rightarrow \epsilon$	$\{t := mktable(nil);$ $push(t, tblptr); push(0, offset)\}$
$D \rightarrow D; D$	
$D \rightarrow \text{proc } id; \mathbf{N} D_1; S$	$\{t := top(tblptr); addwidth(t, top(offset));$ $pop(tblptr); pop(offset);$ $enterproc(top(tblptr), id.name, t)\}$
$D \rightarrow id : T$	$\{enter(top(tblptr), id.name, T.type, top(offset));$ $top(offset) := top(offset) + T.width\}$
$\mathbf{N} \rightarrow \epsilon$	$\{t := mktable(top(tblptr));$ $push(t, tblptr); push(0, offset)\}$



Lưu trữ dấu vết của các phạm vi

- *mktable(previous)* Tạo một bảng ký hiệu mới và trả lại con trỏ của bảng ký hiệu đó. Tham số *previous* là con trỏ tới thủ tục chứa nó.
- Stack *tblptr* chứa con trỏ tới các bảng ký hiệu và các thủ tục chứa nó.
- Stack *offset* chứa dấu vết các địa chỉ tương ứng ở mức lồng nhau nào đó
- *enter(table,name,type,offset)* tạo một lối vào mới cho định danh *name* trong bảng ký hiệu mà *table* trỏ tới, đồng thời chỉ ra các thuộc tính *type* và *offset*.
- *addwidth(table,width)* ghi giá trị *width* của mỗi lối vào trong *table* vào *header* của bảng ký hiệu
- *enterproc(table,name,newtable)* Tạo một lối vào mới cho thủ tục *name* trong bảng ký hiệu chỉ bởi *table*. Tham số *newtable* chỉ tới bảng ký hiệu cho thủ tục *name*.



Các luật về phạm vi lồng nhau

- Toán tử insert (dùng khi phân tích cú pháp khai báo) vào bảng ký hiệu không được ghi đè những khai báo trước
- Toán tử lookup (dùng khi phân tích việc sử dụng các định danh trong bảng ký hiệu) vào bảng ký hiệu luôn luôn tham chiếu luật phạm vi gần nhất
- Toán tử delete (dùng khi giải phóng bộ nhớ) chỉ được xóa những dòng của định danh được khai báo gần nhất



Thêm một đối tượng vào bảngKH

```
void declareObject(Object* obj) {
    if (obj->kind == OBJ_PARAMETER) {
        Object* owner = symtab->currentScope->owner;
        switch (owner->kind) {
            case OBJ_FUNCTION:
                addObject(&(owner->funcAttrs->paramList), obj);
                break;
            case OBJ_PROCEDURE:
                addObject(&(owner->procAttrs->paramList), obj);
                break;
            default:
                break;
        }
    }

    addObject(&(symtab->currentScope->objList), obj);
}
```



Ứng dụng luật phạm vi gần nhất

```
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;
}

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

    while (scope != NULL) {
        obj = findObject(scope->objList, name); //Tìm trong phạm vi hiện hành
        if (obj != NULL) return obj;
        scope = scope->outer;
    }
    obj = findObject(symtab->globalObjectList, name);
    if (obj != NULL) return obj;
    return NULL;
}
```



Khái niệm kiểm tra kiểu

- Kiểm tra xem chương trình có tuân theo các luật về kiểu của ngôn ngữ không
- Trình biên dịch quản lý thông tin về kiểu
- Việc kiểm tra kiểu được thực hiện bởi bộ kiểm tra kiểu (type checker), một bộ phận của trình biên dịch



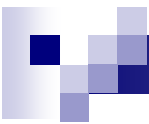
Ví dụ về kiểm tra kiểu

- Toán tử % của C chỉ thực hiện khi các toán hạng là số nguyên
- Chỉ có mảng mới có chỉ số và kiểu của chỉ số phải đếm được (nguyên, ký tự)
- Một hàm phải có một số lượng tham số nhất định và các tham số phải đúng kiểu



Kiểm tra kiểu

- Có hai phương pháp tĩnh và động
- Phương pháp áp dụng trong thời gian dịch là tĩnh
- Trong các ngôn ngữ như C hay Pascal, kiểm tra kiểu là tĩnh và được dùng để kiểm tra tính đúng đắn của chương trình trước khi nó được thực hiện
- Kiểm tra kiểu tĩnh cũng được sử dụng khi xác định dung lượng bộ nhớ cần thiết cho các biến
- Bộ kiểm tra kiểu được xây dựng dựa trên
 - Các biểu thức kiểu của ngôn ngữ
 - Bộ luật để định kiểu cho các cấu trúc



Biểu thức kiểu (Type Expression)

Biểu diễn kiểu của một cấu trúc ngôn ngữ

Một biểu thức kiểu là một kiểu dữ liệu chuẩn hoặc được xây dựng từ các kiểu dữ liệu khác bởi cấu trúc kiểu (*Type Constructor*)

1. Kiểu dữ liệu chuẩn (int, real, boolean, char) là biểu thức kiểu
2. Biểu thức kiểu có thể liên hệ với một tên. Tên kiểu là biểu thức
3. Cấu trúc kiểu được ứng dụng vào các biểu thức kiểu tạo ra biểu thức kiểu



Cấu trúc kiểu

(a) Mảng (*Array*). Nếu T là biểu thức kiểu thì $array(I, T)$ là biểu thức kiểu biểu diễn một mảng với các phần tử kiểu T và chỉ số trong miền I

Ví dụ : `array [10] of integer` có kiểu `array(1..10,int)`;

(b) Tích Descarter. Nếu T_1 và T_2 là các biểu thức kiểu thì tích Descarter $T_1 \times T_2$ là biểu thức kiểu

(c) Bản ghi (*Record*). Tương tự như tích Descarter nhưng chứa các tên khác nhau cho các kiểu khác nhau,

Ví dụ

```
struct
{
  double r;
  int i;
}
Có kiểu ((r x double) x (i x int))
```




Cấu trúc kiểu (tiếp)

- (d) *Con trỏ*: Nếu T là biểu thức kiểu thì $\text{pointer}(T)$ là biểu thức kiểu
- (e) *Hàm* Nếu D là miền xác định và R là miền giá trị của hàm thì kiểu của nó được biểu diễn là
biểu_thức : $D : R$.

Ví dụ hàm của C

int f(char a, b)

Có kiểu: $\text{char} \times \text{char} : \text{int}$.



Hệ thống kiểu (Type System)

- Tập các luật để xây dựng các biểu thức kiểu trong những phần khác nhau của chương trình
- Được định nghĩa thông qua định nghĩa tựa cú pháp
- Bộ kiểm tra kiểu thực hiện một hệ thống kiểu
- Ngôn ngữ định kiểu mạnh: Chương trình dịch kiểm soát được hết các lỗi về kiểu



Thuộc tính

- Thuộc tính là khái niệm trừu tượng biểu diễn một đại lượng bất kỳ , chẳng hạn một số, một xâu, một vị trí trong bộ nhớ
- Thuộc tính được gọi là **tổng hợp** nếu giá trị của nó tại một nút trong cây được xác định từ giá trị của các nút con của nó.
- Thuộc tính **kế thừa** là thuộc tính tại một nút mà giá trị của nó được định nghĩa dựa vào giá trị nút cha và/hoặc các nút anh em của nó.

Định nghĩa tựa cú pháp (syntax directed definition)

Định nghĩa tựa cú pháp là dạng tổng quát của văn phạm phi ngữ cảnh trong đó:

- Mỗi ký hiệu của văn phạm liên kết với một tập thuộc tính ,
- Mỗi sản xuất $A \rightarrow \alpha$ liên hệ với một tập các quy tắc ngữ nghĩa để tính giá trị thuộc tính liên kết với những ký hiệu xuất hiện trong sản xuất. Tập các quy tắc ngữ nghĩa có dạng

$$b = f(c_1, c_2, \dots, c_n)$$

f là một hàm và b thoả một trong hai yêu cầu sau:

- b là một thuộc tính tổng hợp của A và c_1, \dots, c_n là các thuộc tính liên kết với các ký hiệu trong vế phải sản xuất $A \rightarrow \alpha$
- b là một thuộc tính thừa kế một trong những ký hiệu xuất hiện trong α , và c_1, \dots, c_n là thuộc tính của các ký hiệu trong vế phải sản xuất $A \rightarrow \alpha$

Ví dụ

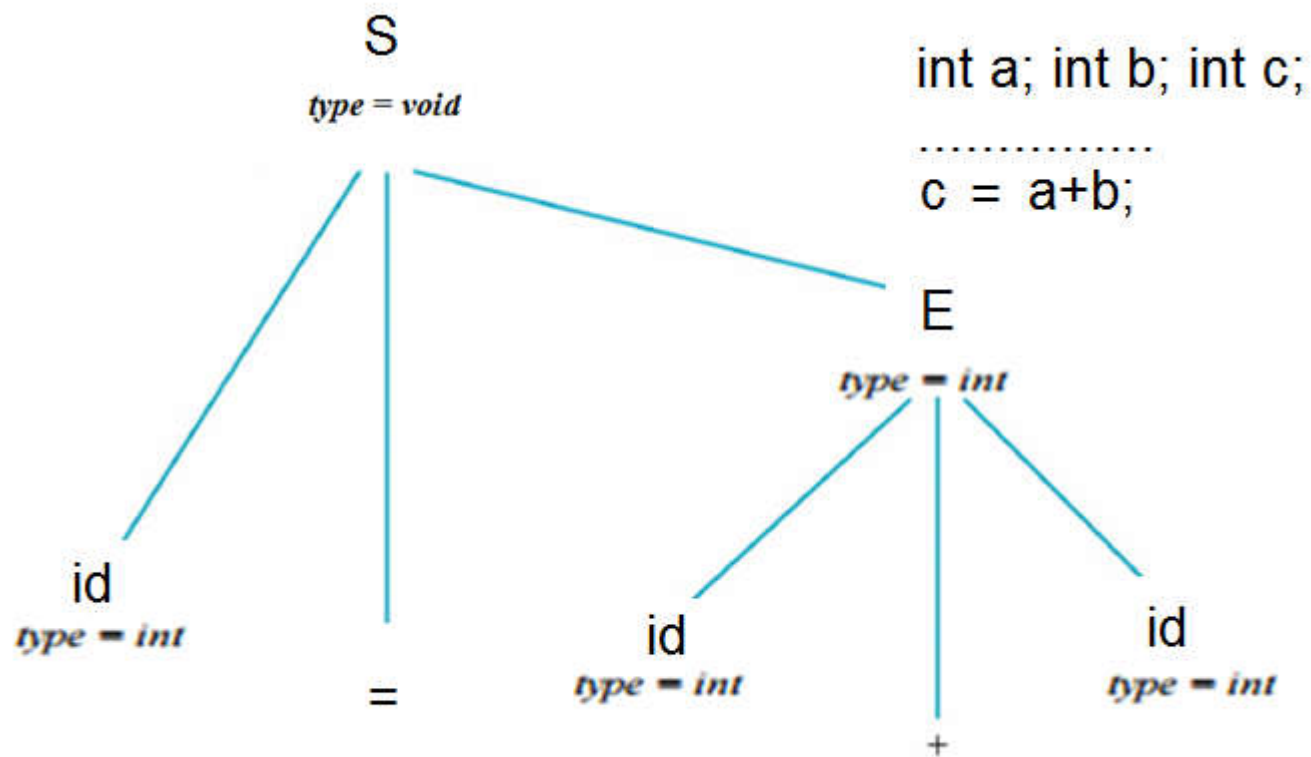
Sản xuất	Quy tắc ngữ nghĩa
$L \rightarrow E \text{ return}$	Print (E.val)
$E \rightarrow E1 + T$	$E.val = E1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T1 * F$	$T.val = T1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{num}$	$F.val = \text{num.Lexval}$

- Các ký hiệu E , T , F liên hệ với thuộc tính tổng hợp val
- Từ tố *digit* có thuộc tính tổng hợp lexval (Được bộ phân tích từ vựng đưa ra)

Lệnh gán: Thuộc tính kiểu là tổng hợp

SẢN XUẤT	QUY TẮC NGŨ NGHĨA
$S \rightarrow \text{id} := E$	$S.type := \text{if id.type} = E.type \text{ then void}$ $\quad \text{else type_error}$
$E \rightarrow \text{literal}$	$E.type := \text{char}$
$E \rightarrow \text{num}$	$E.type := \text{int}$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 + E_2$	$E.type := \text{if } E_1.type = \text{int and } E_2.type = \text{int}$ $\quad \text{then int}$ $\quad \text{else type_error}$

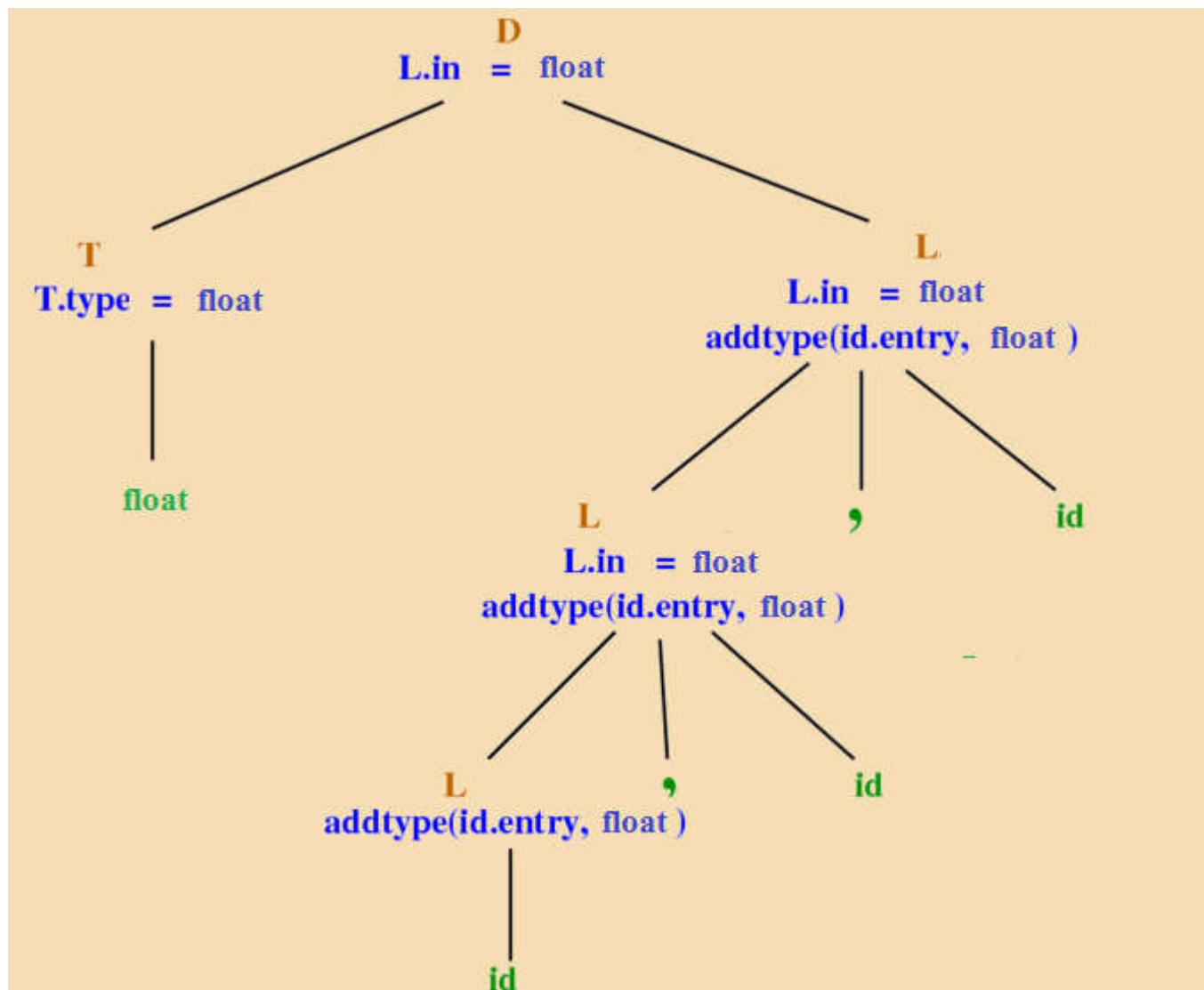
Lệnh gán: Thuộc tính kiểu là tổng hợp



Khai báo: Thuộc tính kiểu là kế thừa

Sản xuất	Quy tắc ngữ nghĩa
$D \rightarrow TL$	<code>L.in = T.type</code>
$T \rightarrow \mathbf{int}$	<code>T.type = INT</code>
$T \rightarrow \mathbf{float}$	<code>T.type = FLOAT</code>
$L \rightarrow L_1, \mathbf{id}$	<code>L1.in = L.in</code> <code>addtype(L.in, id.entry)</code>
$L \rightarrow \mathbf{id}$	<code>addtype(L.in, id.entry)</code>

Khai báo: Thuộc tính kiểu là kế thừa



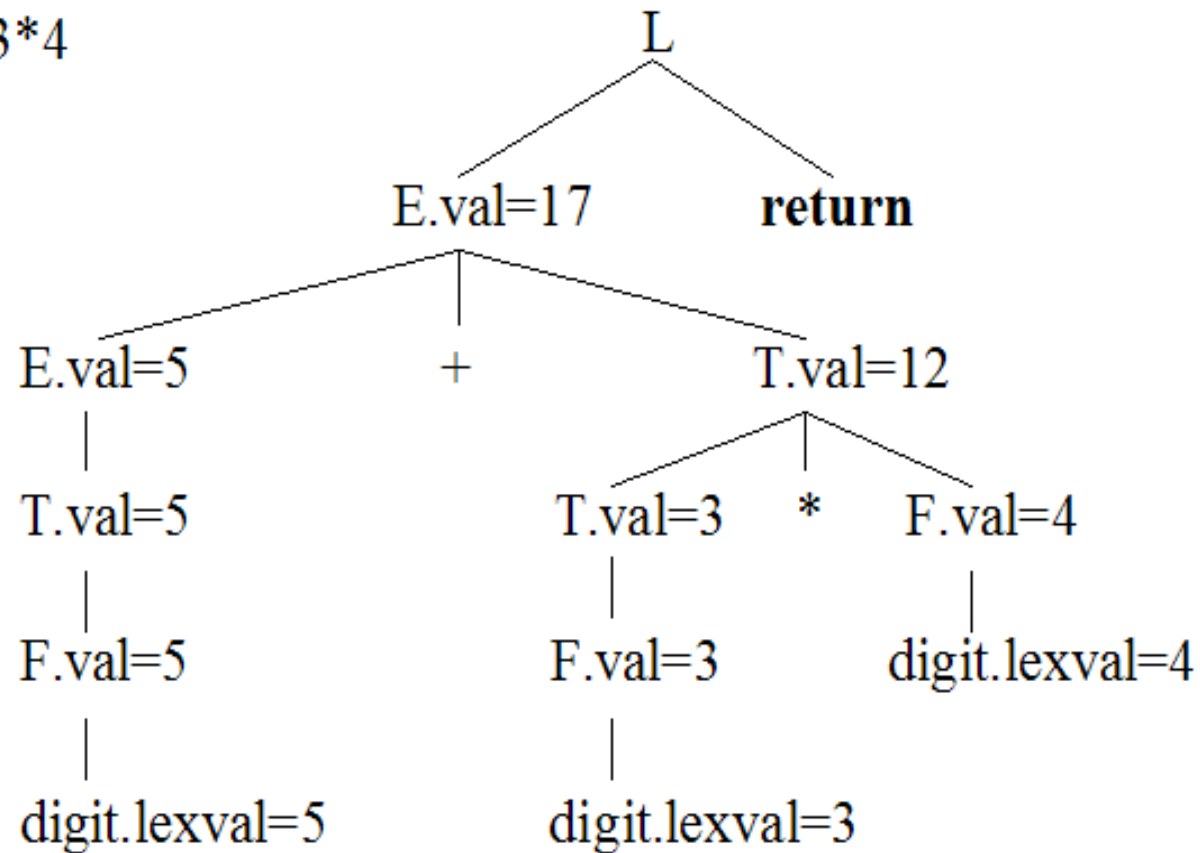


Cây phân tích cú pháp có chú giải

- Cây cú pháp có chỉ ra giá trị các thuộc tính tại mỗi nút được gọi là cây cú pháp có chú giải.

Ví dụ

Input: $5+3*4$



Bộ kiểm tra kiểu của định danh

SẢN XUẤT	QUY TẮC NGŨ NGHĨA
$D \rightarrow \text{id} : T$	$\text{addtype}(\text{id.entry}, T.\text{type})$
$T \rightarrow \text{char}$	$T.\text{type} := \text{char}$
$T \rightarrow \text{int}$	$T.\text{type} := \text{int}$
$T \rightarrow \uparrow T_1$	$T.\text{type} := \text{pointer}(T_1.\text{type})$
$T \rightarrow \text{array}[\text{num}] \text{ of } T_1$	$T.\text{type} := \text{array}(1..\text{num.val}, T_1.\text{type})$

Thực hiện addtype(id.entry,T.type) cho KPL

```
void compileBlock3(void) {
    Object* varObj;
    Type* varType;
    if (lookAhead->tokenType == KW_VAR) {
        eat(KW_VAR);
        do {
            eat(TK_IDENT);
            checkFreshIdent(currentToken->string);
            varObj = createVariableObject(currentToken->string);
            eat(SB_COLON);
            varType = compileType();
            varObj->varAttrs->type = varType;
            declareObject(varObj);
            eat(SB_SEMICOLON);
        } while (lookAhead->tokenType == TK_IDENT);
        compileBlock4();
    }
    else compileBlock4();
}
```

CÁC LUẬT CÚ PHÁP

```
6) Block3 ::= KW_VAR VarDecl
              VarDecls Block4
7) Block3 ::= Block4
16) VarDecls ::= VarDecl VarDecls
17) VarDecls ::= ε
18) VarDecl ::= Ident SB_COLON
              Type SB_SEMICOLON
```

Hàm **compileType** trả về một kiểu dữ liệu, không phải kiểu void như trong bộ parser.

Bộ kiểm tra kiểu của biểu thức

SẢN XUẤT	QUY TẮC NGŨ NGHĨA
$E \rightarrow \text{literal}$	$E.type := \text{char}$
$E \rightarrow \text{num}$	$E.type := \text{int}$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ mod } E_2$	$E.type := \text{if } E_1.type = \text{int} \text{ and } E_2.type = \text{int}$ then int else type_error
$E \rightarrow E_1[E_2]$	$E.type := \text{if } E_2.type = \text{int} \text{ and } E_1.type = \text{array}(s,t)$ then t else type_error
$E \rightarrow E_1 \uparrow$	$E.type := \text{if } E_1.type = \text{pointer}(t) \text{ then } t$ else type_error

Kiểm tra kiểu của nhân tử

```
Type* compileFactor(void) {
    Type* type;
    Object* obj;
    switch (lookAhead->tokenType) {
    case TK_NUMBER:
        eat(TK_NUMBER);
        type = intType;
        break;
    case TK_CHAR:
        eat(TK_CHAR);
        type = charType;
        break;
    case TK_IDENT:
        eat(TK_IDENT);
        obj = checkDeclaredIdent(currentToken->string);
        switch (obj->kind) {
        case OBJ_CONSTANT:
            switch (obj->constAttrs->value->type) {
            case TP_INT:
                type = intType;
                break;
            case TP_CHAR:
                type = charType; break;
            default: break;
            } break;
        case OBJ_VARIABLE:
            if (obj->varAttrs->type->typeClass == TP_ARRAY)
                type = compileIndexes(obj->varAttrs->type);
            else
                type = obj->varAttrs->type;
            break;
    }
```

```
        case OBJ_PARAMETER:
            type = obj->paramAttrs->type;
            break;
        case OBJ_FUNCTION:
            compileArguments(obj->funcAttrs->paramList);
            type = obj->funcAttrs->returnType;
            break;
        default:
            error(ERR_INVALID_FACTOR, currentToken->lineNo,
currentToken->colNo);
            break;
    }
    break;
default:
    error(ERR_INVALID_FACTOR, lookAhead->lineNo, lookAhead-
>colNo);
}

return type;
}
```

LUẬT CÚ PHÁP TƯƠNG ỨNG

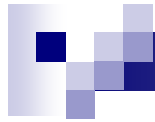
- 86) Factor ::= UnsignedConstant
- 87) Factor ::= **Variable**
- 88) Factor ::= FunctionApplication
- 89) Factor ::= SB_LPAR Expression SB_RPAR
- 90) **Variable** ::= VariableIdent Indexes
- 91) FunctionApplication ::= FunctionIdent Arguments
- 92) Indexes ::= SB_LSEL Expression SB_RSEL Indexes
- 93) Indexes ::= ϵ

Bộ kiểm tra kiểu của lệnh

SẢN XUẤT	QUY TẮC NGŨ NGHĨA
$S \rightarrow \text{id} := E$	$S.type := \text{if id.type} = E.type \text{ then void}$ else type_error
$S \rightarrow \text{if } E \text{ then } S_1$	$S.type := \text{if } E.type = \text{boolean} \text{ then } S_1.type$ else type_error
$S \rightarrow \text{while } E \text{ do } S_1$	$S.type := \text{if } E.type = \text{boolean} \text{ then } S_1.type$ else type_error
$S \rightarrow S_1; S_2$	$S.type := \text{if } S_1.type = \text{void} \text{ and } S_2.type = \text{void}$ then void else type_error

Bộ kiểm tra kiểu của hàm

SẢN XUẤT	QUY TẮC NGŨ NGHĨA
$D \rightarrow \text{id} : T$	$\text{addtype}(\text{id.entry}, T.\text{type}); D.\text{type} := T.\text{type}$
$D \rightarrow D_1; D_2$	$D.\text{type} := D_1.\text{type} \times D_2.\text{type}$
$\text{Fun} \rightarrow \text{fun id}(D) : T; B$	$\text{addtype}(\text{id.entry}, D.\text{type}:T.\text{type})$
$B \rightarrow \{S\}$	
$S \rightarrow \text{id}(EList)$	$E.\text{type} := \text{if } \text{lookup}(\text{id.entry})=t_1 : t_2 \text{ and } EList.\text{type}=t_1$ then t_2 else type_error
$EList \rightarrow E$	$EList.\text{type} := E.\text{type}$
$EList \rightarrow EList, E$	$EList.\text{type} := EList_1.\text{type} \times E.\text{type}$



Hàm kiểm tra biểu thức kiểu tương đương

```
function sequiv(s, t): boolean;  
begin  
    if s và t là cùng kiểu dữ liệu chuẩn then  
        return true;  
    else if s = array(s1, s2) and t = array(t1, t2) then  
        return sequiv(s1, t1) and sequiv(s2, t2)  
    else if s = s1 x s2 and t = t1 x t2 then  
        return sequiv(s1, t1) and sequiv(s2, t2)  
    else if s = pointer(s1) and t = pointer(t1) then  
        return sequiv(s1, t1)  
    else if s = s1 → s2 and t = t1 → t2 then  
        return sequiv(s1, t1) and sequiv(s2, t2)  
    else  
        return false;  
end;
```



Kiểm tra kiểu tương đương trong KPL

```
int compareType(Type* type1, Type* type2) //module symtab
{
    if (type1->typeClass == type2->typeClass) {
        if (type1->typeClass == TP_ARRAY) {
            if (type1->arraySize == type2->arraySize)
                return compareType(type1->elementType, type2-
>elementType);
            else return 0;
        } else return 1;
    } else return 0;
}
```



Chuyển đổi kiểu

- Kiểu của $x+i$ với
 - x kiểu real
 - i kiểu int

Khi dịch sang lệnh máy, phép cộng với kiểu real và kiểu int có mã lệnh khác nhau

- Tùy ngôn ngữ và bộ luật chuyển đổi sẽ quy đổi các toán hạng về một trong hai kiểu

Tự động chuyển đổi kiểu trong biểu thức

SẢN XUẤT	QUY TẮC NGŨ NGHĨA
$E \rightarrow \text{num}$	$E.type := int$
$E \rightarrow \text{num.num}$	$E.type := real$
$E \rightarrow \text{id}$	$E.type := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ op } E_2$	$E.type :=$ if $E_1.type = int$ and $E_2.type = int$ then int else if $E_1.type = int$ and $E_2.type = real$ then $real$ else if $E_1.type = real$ and $E_2.type = int$ then $real$ else if $E_1.type = real$ and $E_2.type = real$ then $real$ else $type_error$