Nhập môn Chương trình dịch

Bài 1: Tổng quan

Nội dung chính

- Sơ lược về môn học
- Các chương trình dịch
  - Chương trình dịch là gì?
  - Tại sao phải biết chúng?
  - Các bộ phận của một chương trình dịch
- Giới thiệu về “Phân tích từ vựng”
  - Từ luồng văn bản đến luồng từ tố (tokens)

Tài liệu

- Phạm Hồng Nguyên, “Nhập môn Chương trình dịch”.
- Phạm Hồng Nguyên, “Giáo trình thực hành Chương trình dịch”.
- Aho, Sethi, Ullman, “Compilers – Principles, Techniques and Tools”.

Mục tiêu

- Ứng dụng thực tế của lý thuyết ngôn ngữ
  → rất đẹp nhưng rất khó ☹
- Phân tích văn bản (parsing)
- Nâng cao hiểu biết về mã nguồn
- Sử dụng các cấu trúc dữ liệu phức tạp
  → rất tốn nơron thần kinh ☹
- Hiểu cách cài đặt các ngôn ngữ bậc cao và cách chuyển đổi chúng về ngôn ngữ máy
- Hiểu ngữ nghĩa của các ngôn ngữ lập trình
- Lập trình giỏi hơn (đặc biệt là trong nhóm)

Chương trình dịch là gì?

- Chương trình chuyển đổi cách thể hiện này của một chương trình sang cách thể hiện khác.
  - Nhận dạng tính hợp lệ hoặc không hợp lệ của các chương trình.
  - Nhằm mục đích tạo ra các đoạn mã đúng, hiệu quả, chính xác.
- Ví dụ: Chuyển mã nguồn viết trong ngôn ngữ bậc cao sang ngôn ngữ máy
- Ví dụ:
  - *.CPP → *.EXE
  - *.JAVA → *.CLASS (bytecode)

Mã nguồn

- Được tối ưu để tạo cảm giác thân thiện, dễ dùng đối với lập trình viên
  - Có cú pháp gần giống ngữ pháp của ngôn ngữ tự nhiên
  - Có nhiều câu lệnh phức tạp hơn ngôn ngữ máy

```
int expr(int n)
{
   int d;
   d = 4 * n * n * (n + 1) * (n + 1);
   return d;
}
```

Mã máy

- Được tối ưu cho phần cứng
  - Giảm tối đa số câu lệnh thừa
  - Mã Assembly ≈ mã máy

```
lda $30,-32($30)
stq $26,0($30)
stq $15,8($30)
bis $30,$30,$15
bis $16,$16,$1
stl $1,16($15)
lds $f1,16($15)
sts $f1,24($15)
ldl $5,24($15)
bis $5,$5,$2
s4addq $2,0,$3
ldl $4,16($15)
mull $4,$3,$2
ldl $3,16($15)
```

```
addq $3,1,$4
mull $2,$4,$2
ldl $3,16($15)
addq $3,1,$4
mull $2,$4,$2
stl $2,20($15)
ldl $0,20($15)
br $31,$33
$33:
bis $15,$15,$30
ldq $26,0($30)
ldq $15,8($30)
addq $30,32,$30
ret $31,($26),1
```

Dịch như thế nào ?

- Mã nguồn và mã máy không giống nhau
- Độ phức tạp của các ngôn ngữ bậc cao cũng khác nhau
- Mục tiêu của các chương trình dịch:
  - Cho phép lập trình viên sử dụng ngôn ngữ nguồn để lập trình
  - Chương trình dịch chuyển sang mã máy với hiệu quả cao nhất có thể
  - Tốc độ dịch cao (< $O(n^3)$)
  - Có thể thay đổi dễ dàng chương trình dịch khi cần thay đổi ngôn ngữ (thêm từ vựng, cú pháp, khái niệm mới, hoặc chuyển sang ngôn ngữ mới)

Ví dụ

Mã máy chưa tối ưu

```
lda $30,-32($30)
stq $26,0($30)
stq $15,8($30)
bis $30,$30,$15
bis $16,$16,$1
stl $1,16($15)
lds $f1,16($15)
sts $f1,24($15)
ldl $5,24($15)
bis $5,$5,$2
s4addq $2,0,$3
ldl $4,16($15)
mull $4,$3,$2
ldl $3,16($15)
addq $3,1,$4
mull $2,$4,$2
ldl $3,16($15)
addq $3,1,$4
mull $2,$4,$2
stl $2,20($15)
ldl $0,20($15)
br $31,$33
$33:
bis $15,$15,$30
ldq $26,0($30)
ldq $15,8($30)
addq $30,32,$30
ret $31,($26),1
```

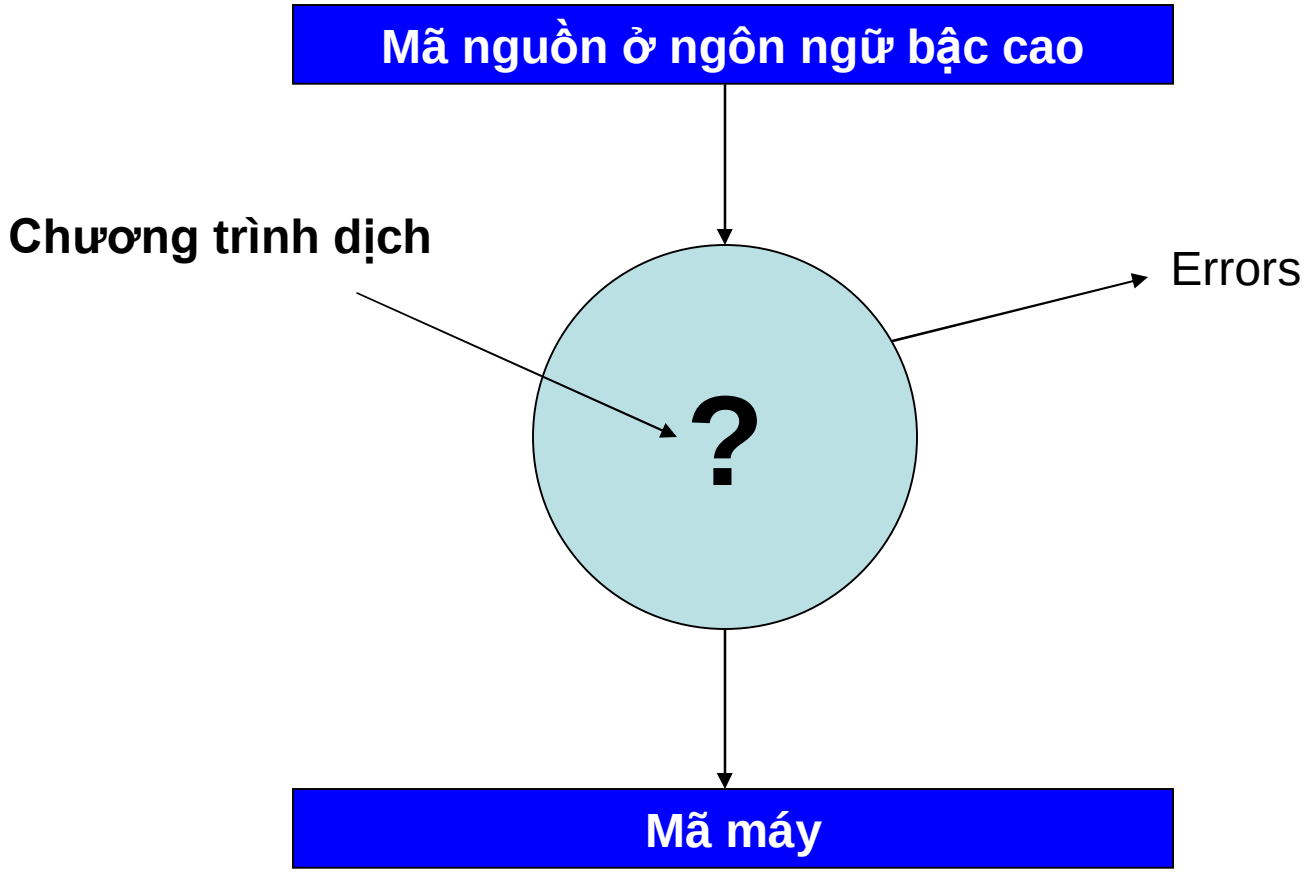
Mã máy sau khi tối ưu

```
s4addq $16,0,$0
mull $16,$0,$0
addq $16,1,$16
mull $0,$16,$0
mull $0,$16,$0
ret $31,($26),1
```

Tính đúng đắn

- Các ngôn ngữ lập trình cho phép mô tả các chương trình một cách chính xác
- Vì thế, việc dịch cũng có thể được mô tả một cách chính xác (nghĩa là, các chương trình dịch có thể được viết đúng)
- Ý nghĩa
  - Khó có thể gỡ rối một chương trình với một chương trình dịch được viết sai
  - Chương trình dịch cũng là 1 chương trình
    → Các khái niệm về chi phí, bảo mật
  - Trong môn học này, ta sẽ nghiên cứu các kỹ thuật để viết một chương trình dịch đúng.

Dịch như thế nào để hiệu quả?



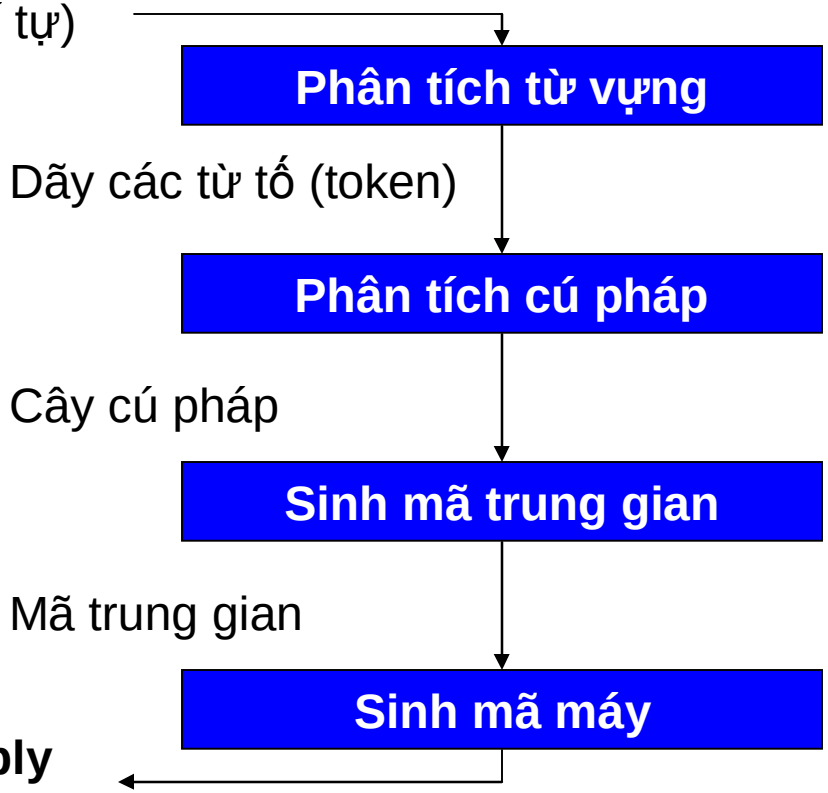
Ý tưởng: dịch từng bước

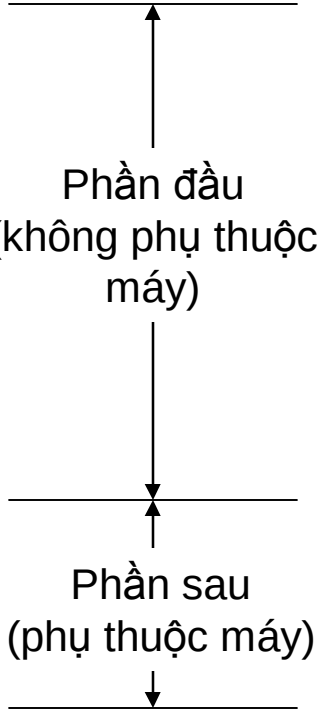
- Chương trình nguồn sẽ chuyển qua một chuỗi các dạng thể hiện khác nhau
- Mỗi dạng thể hiện được tối ưu để thực hiện các thao tác khác nhau trong quá trình dịch (ví dụ: kiểm tra kiểu, tối ưu mã)
- Càng về cuối, các dạng thể hiện càng gần với mã máy hơn.

Cấu trúc của chương trình dịch

Mã nguồn (dãy các kí tự)

```
If (a < 0) min = a;
```

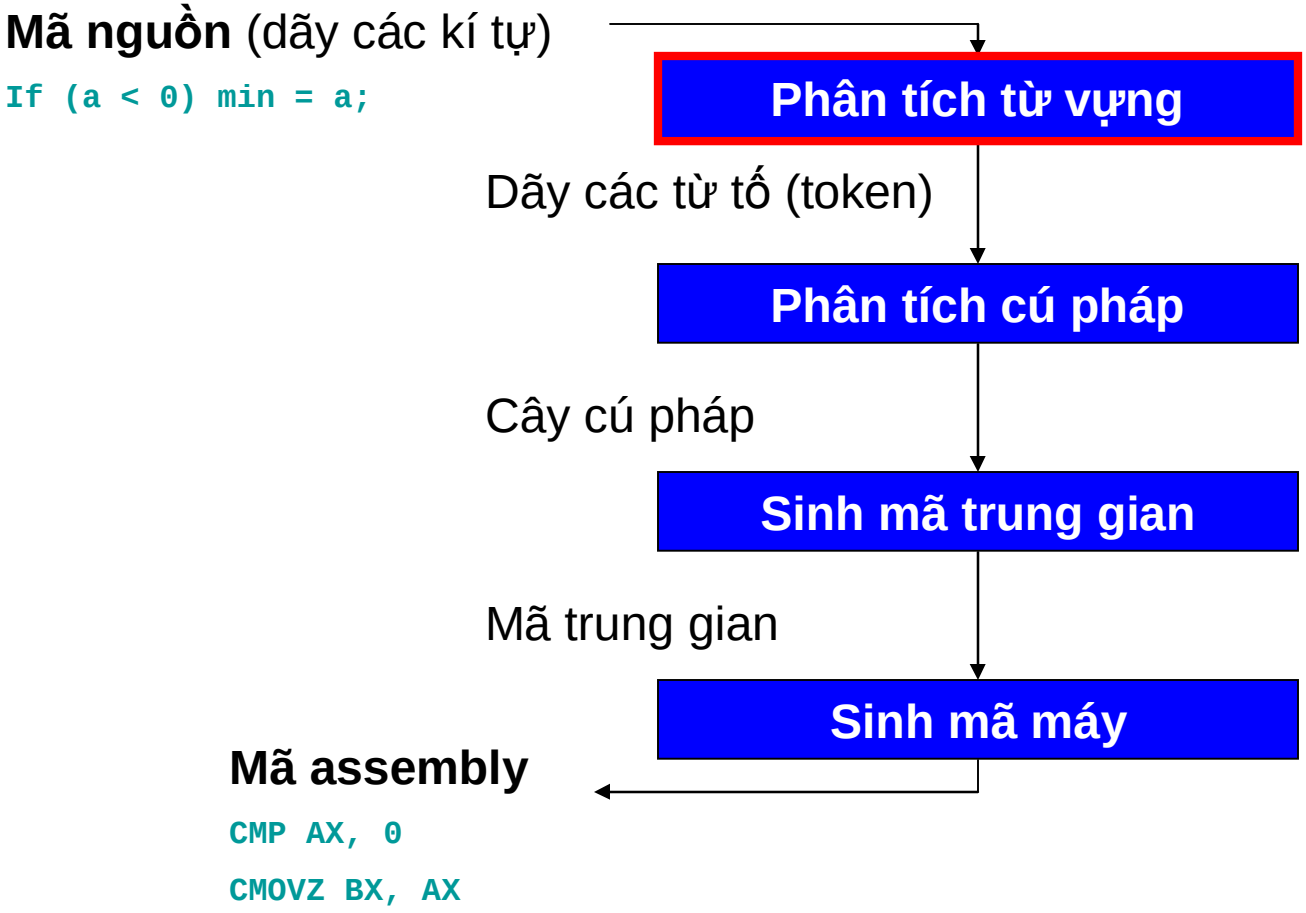




Mã assembly

```
CMP AX, 0
CMOVZ BX, AX
```

Phân tích từ vựng (PTTV)



Nhiệm vụ của PTTV

- Chuyển đổi dãy các kí tự của chương trình nguồn thành dãy các từ tố (token) bao gồm <loại từ tố> và <thuộc tính>
- Các từ tố (token) là các đơn vị cơ bản của cú pháp

```
If (a < b) min = a;
else min = b;
```

I	f		(	a	<	b	)		m	i	n		=		a	;	\n	e	l	s	e
m	i	n		=		b	;	EOF													

If	(	Id:a	<	Id:b	)	Id:min	=	Id:a	;
Else	Id:min	=	Id:b	;					

Dãy từ tố

- Một dạng thể hiện của chương trình nguồn
- Mô tả từ tố: <loại từ tố> và <thuộc tính>
- Ví dụ: <Id, “a”> <Id, “min”> <If,> <Int, 10> <Float, 1.5> <[,> <),>
- Khi cần gỡ lỗi, thông báo lỗi, mô tả từ tố sẽ bao gồm cả vị trí của từ tố: file, số dòng
- Ví dụ: <Id, “min”, “min.cpp”, 15>

Các vấn đề trong PTTV

- Cách mô tả từ tố
  - Các thuộc tính
  - Bảng kí hiệu
- Cài đặt các bộ PTTV
  - Phương pháp AD-HOC
  - Phương pháp nhận dạng biểu thức chính quy
  - NFAs
  - DFAs

Nhập môn Chương trình dịch

Bài 2: Phân tích từ vựng

Nội dung chính

- Mô tả các bước chính của chương trình dịch
- Phân tích từ vựng là gì?
- Viết một chương trình phân tích từ vựng
- Mô tả từ tố - biểu thức chính quy (REs)
- Viết một chương trình sinh ra chương trình phân tích từ vựng
- Chuyển đổi REs – NFAs
- Chuyển đổi NFAs – DFAs
- Bài tập về nhà 1

Mô tả các bước dịch (1)

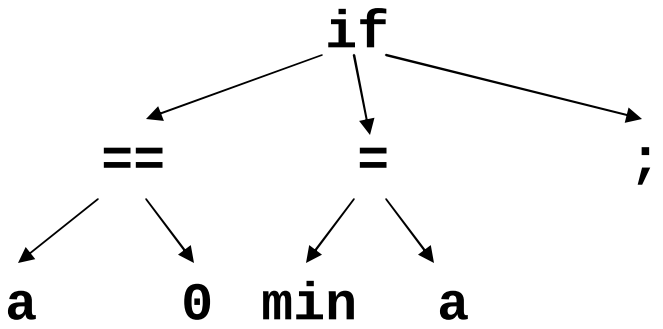
Mã nguồn (dãy các kí tự)

```
If (a == 0) min = a;
```

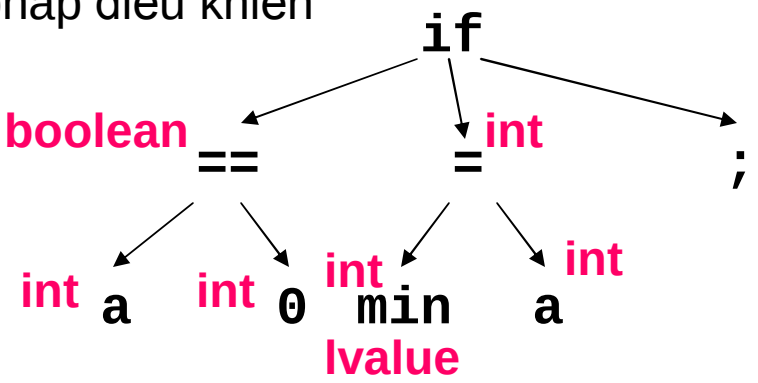
Dãy các từ tố (token)

If	(	Id:a	==	0	)	Id:min	=	Id:a	;

Cây cú pháp



Cây cú pháp điều khiển



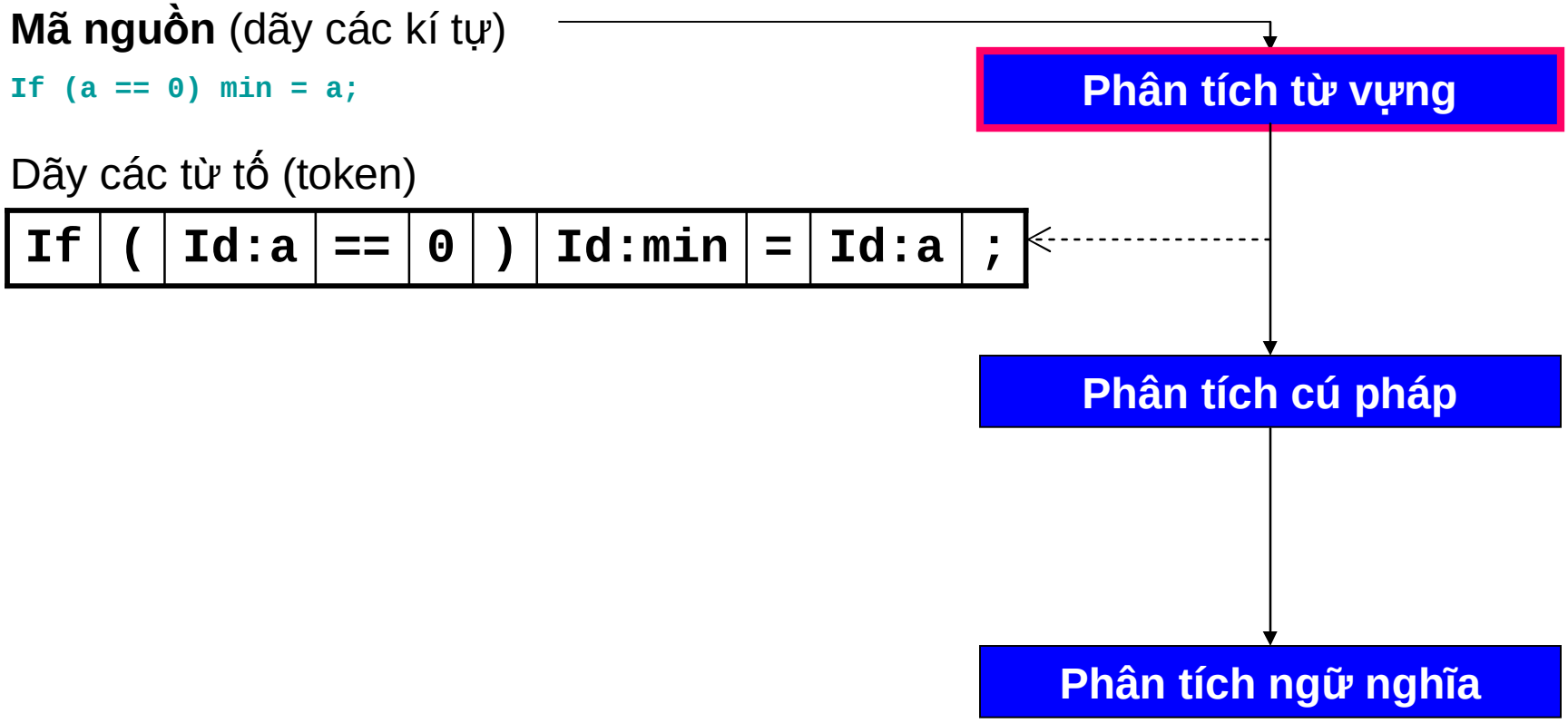
Phân tích từ vựng

Phân tích cú pháp

Phân tích ngữ nghĩa

Phân tích từ vựng

(lexical analysis)



Từ tố (token)

- Tên: x, y11, elsex, _i00
- Từ khóa: if, else, while, break
- Số nguyên: 2, 1000, -500, 5L
- Số thực: 2.0, 0.00020, .02, 1.1e5, 0.e-10
- Kí hiệu: +, *, {, }, ++, <, <<, [,], >=
- Xâu kí tự: “x”, “He said, \“Are you?\””
- Ghi chú: /** don’t change this **/

Phân tích từ vựng kiểu AD-HOC

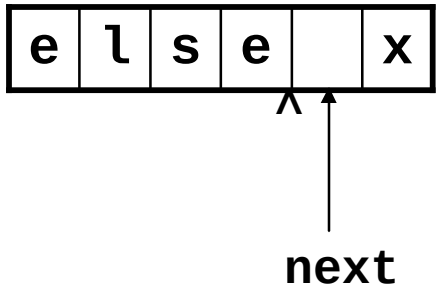
- Tự viết mã lệnh để sinh ra các từ tố
- Ví dụ: Đoạn chương trình nhận dạng từ tố loại “Tên”

```
Token readIdentifier( ) {
  String id = “”;
  while (true) {
      char c = input.read();
      if (!identifierChar(c))
            return new Token(ID, id, lineNumber);
      id = id + String(c);
  }
}
```

Nhìn trước 1 kí tự

- Duyệt chương trình nguồn từng kí tự một
  - Sử dụng kí tự nhìn trước để quyết định loại từ tố và vị trí kết thúc từ tố

```
char next;
…
while (identifierChar(next)) {
  id = id + String(next);
  next = input.read ();
}
```



Vòng lặp chính

- **Hàm nextToken**

```
Token nextToken( ) {
  if (identifierFirstChar(next))
      return readIdentifier();
  if (numericChar(next))
      return readNumber();
  if (next == '\"') return readStringConst();
…
}
```

Các vấn đề của phương pháp AD-HOC

- Trong một số trường hợp, không thể biết loại từ tố qua kí tự đọc vào đầu tiên
  - Nếu kí tự đầu tiên là “i”, đó có thể là tên hoặc từ khóa “if”
  - Nếu kí tự đầu tiên là “2”, đó có thể là số nguyên hoặc số thực
  - Mã lệnh kiểu AD-HOC khó viết đúng, và bảo trì nó còn khó hơn nhiều
- Vì vậy, cần một cách tiếp cận mới, có nguyên tắc hơn: các *chương trình* ***sinh ra*** các chương trình phân tích từ vựng một cách tự động (ví dụ: lex, JLex)

Các vấn đề nảy sinh

- Cần cách mô tả các từ tố không bị nhập nhằng
  - 2.e0 20.e-01 2.0000
  - “” “x” “\\” “\”\”
- Cần một phương pháp để tách chuỗi kí tự thành các từ tố

```
if (x == 0) a = x<<1;
iff (x == 0) a = x<1;
```

- Phương pháp này phải hiệu quả
  - Phân biệt các từ tố có chung đoạn kí tự đầu
  - Chỉ cần nhìn trước 1 kí tự

Mô tả từ tố

- Từ tố trong các ngôn ngữ lập trình có thể mô tả bằng các **biểu thức chính quy** (REs)
- Mỗi biểu thức chính quy R có thể biểu diễn một tập các xâu kí tự L(R)
- L(R) gọi là "ngôn ngữ" định nghĩa bởi R
- Ví dụ
  - L(**abc**) = { **abc** }
  - L(**hello**|**goodbye**) = {**hello**, **goodbye**}
  - L([**1**-**9**][**0**-**9**]*) = tập các hằng số nguyên dương
- Ý tưởng: mô tả mỗi loại từ tố bằng một biểu thức chính quy

Quy ước của REs

a	L(a) = {a} – tập hợp gồm xâu “a”
ε	L(ε) = {ε} – tập hợp gồm xâu rỗng
R|S	L(R|S) = L(R) ∪ L(S) – hợp của L(R) và L(S)
RS	L(RS) = {xy | x ∈ L(R), y ∈ L(S)} – nối 2 xâu bất kì của L(R) và L(S)
R*	L(R*) = L(ε|R|RR|RRR|RRRR …) – nối các xâu của L(R) lại với nhau

Một số quy ước khác

R+	L(R+) = L(R*) \ {ε} – R* loại bỏ xâu rỗng
R?	L(R?) = L(R|ε)
[abcd]	L([abcd]) = L(a|b|c|d)
[a-z]	L([a-z]) = L(a|b|..|z)
[^abc]	L([^abc]) = kí tự bất kì không thuộc L([abc])
[^a-z]	L([^a-z]) = kí tự bất kì không thuộc L([a-z])

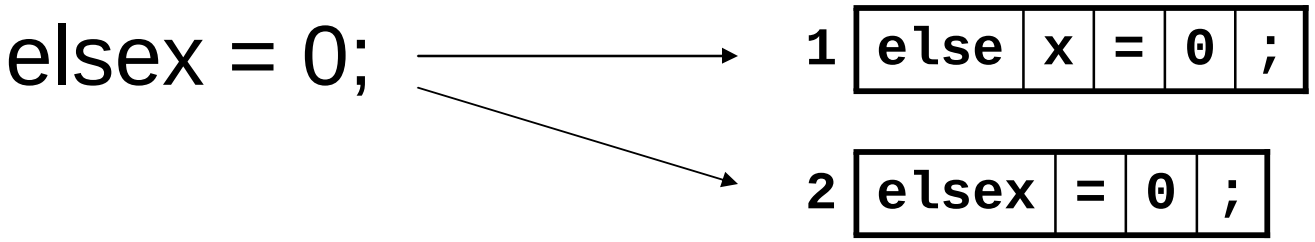
Ví dụ

Biểu thức chính quy (RE)	Xâu thuộc ngôn ngữ
a	“a”
ab	“ab”
a | **b**	“a”, “b”
(**ab**)*	“”, “ab”, “abab” …
(**a** | ε) **b**	“ab”, “b”

Ví dụ

Biểu thức chính quy (RE)	Xâu thuộc ngôn ngữ
digit = [0-9]	“0”, “2”, “9”
posint = digit+	“123”, “5678”
int = -? posint	“123”, “-5678”
real = int (ε | (. posint))	“123”, “-5678.123”
= -?[0-9]+(ε | (. [0-9]+))	
id = [a-zA-Z_][a-zA-Z0-9_]*	“min”, “cal_Max1”

Tách từ tố từ chuỗi kí tự



- REs là chưa đủ để chỉ ra cách tách các từ tố
- Đa số các ngôn ngữ sử dụng luật “dài nhất thắng”: RE cho xâu dài nhất có thể được sẽ được chọn
- Khi RE cho các xâu dài bằng nhau, từ tố được ưu tiên hơn sẽ được chọn
- Đặc tả chương trình PTTV = REs + luật “dài nhất thắng” + mức độ ưu tiên

Đặc tả chương trình PTTV

- Là ***đầu vào*** cho các *chương trình* ***sinh ra*** chương trình phân tích từ vựng
  - Danh sách REs theo thứ tự ưu tiên
  - Hành động gắn liền với mỗi RE khi chương trình PTTV nhận dạng được một từ tố bằng RE đó
- ***Đầu ra*** của các chương trình này là một chương trình PTTV có thể
  - Đọc chương trình nguồn và tách nó ra thành các từ tố bằng cách nhận dạng REs
  - Thông báo lỗi nếu gặp phải kí tự không đúng theo REs

Example: Lex

```
%%
digits = 0|[1-9][0-9]*
letter = [A-Za-z]
identifier = {letter}({letter}|[0-9_])*
whitespace = [\ \t\n\r]+
%%
{whitespace} {/* discard */}
{digits} { return new
  IntegerConstant(Integer.parseInt(yytext()); }
“if” { return new IfToken(); }
“while” { return new WhileToken(); }
…
{identifier} { return new IdentifierToken(yytext()); }
```

Tổng kết

- Chương trình PTTV chuyển chương trình nguồn thành dãy các từ tố (token)
- PTTV kiểu AD-HOC khó viết đúng, khó bảo trì
- Có thể mô tả chính xác các từ tố trong các ngôn ngữ lập trình bằng biểu thức chính quy (RE)
- Đầu vào của chương trình sinh ra chương trình PTTV là đặc tả của chương trình PTTV: REs, mức độ ưu tiên và các hành động tương ứng
- Bài tới: hoạt động của các chương trình sinh ra chương trình PTTV

Nhập môn Chương trình dịch

Bài 3: Tự động sinh bộ PTTV

Nội dung chính

- Mở rộng cú pháp biểu thức chính quy - RE
- Vòng lặp chính
- Ôtômát hữu hạn đơn định - DFAs
- Ôtômát hữu hạn không đơn định - NFAs
- Chuyển đổi RE-NFA
- Chuyển đổi NFA-DFA

Mở rộng cú pháp của RE

- R_1 và R_2 là các RE, các biểu thức sau là RE
  - a — $L(a) = \{a\}$
  - $R_1|R_2$ — $L(R_1|R_2) = L(R_1) \cup L(R_2)$
  - R_1R_2 — Nối 2 xâu thuộc $L(R_1)$ và $L(R_2)$
  - R_1* — Nối 0 hoặc nhiều xâu thuộc $L(R_1)$
  - $R_1?$ — Xâu rỗng hoặc xâu thuộc $L(R_1)$
  - R_1 — Nối 1 hoặc nhiều xâu thuộc $L(R_1)$
  - (R_1)
  - [abc] — L([abc]) = L(a|b|c)
  - [a-e] — L([a-e]) = L(a|b|..|e)
  - [^…] — Kí tự bất kì ngoài các kí tự trong ngoặc

Chương trình sinh ra bộ PTTV

- Đọc danh sách theo thứ tự ưu tiên các RE: $R_1,\ldots R_n$, mỗi biểu thức mô tả một loại từ tố cùng với các hành động tương ứng
- Ví dụ

```
-?[1-9][0-9]* { return new Token(Tokens.IntConst,
Integer.parseInt(yytext()) }
```

- Sinh ra mã lệnh của chương trình PTTV có thể
  1. Kiểm tra tính đúng đắn về từ vựng của chương trình nguồn
  2. Sinh ra một dãy các từ tố tương ứng
- Quan sát: Bài toán 1 tương đương với việc kiểm tra xem chương trình nguồn có thuộc ngôn ngữ của biểu thức chính quy sau

$$(R_1|\ldots|R_n)^*$$

- Bài toán 1: Tìm cách kiểm tra một xâu có thuộc L(R) với R là biểu thức chính quy bất kì

Nhận dạng biểu thức chính quy

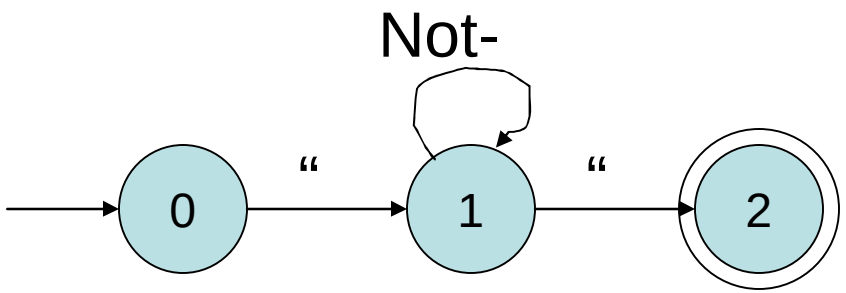
- Ôtômát nhận dạng RE
  - Bắt đầu ở một trạng thái khởi tạo
  - Lần lượt xét các kí tự của xâu
  - Thay đổi trạng thái tùy theo kí tự đọc vào
  - Khi đọc hết xâu, nếu đạt được trạng thái kết thúc thì xâu vào được nhận dạng theo biểu thức chính quy đó
- Với PTTV, ta chỉ cần một số hữu hạn trạng thái: ôtômát hữu hạn (đơn định hoặc không đơn định)
  - NFA & DFA
  - Trạng thái = một số nguyên

Ôtômát hữu hạn (FA)

- Biểu diễn ôtômát hữu hạn
  - Bằng bảng chuyển trạng thái

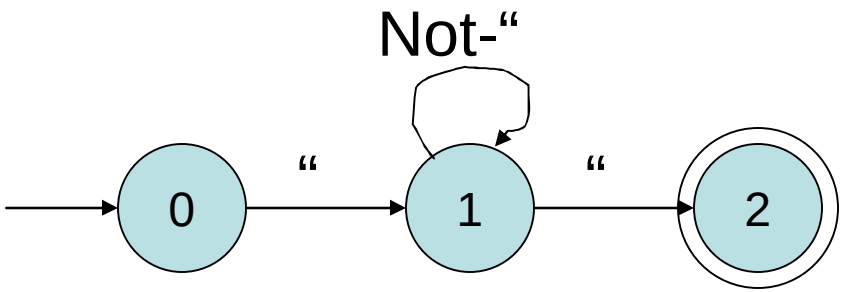
	“	Not-
0	1	Error
1	2	1
2	Error	Error

  - Bằng đồ thị chuyển



Nhận dạng biểu thức chính quy

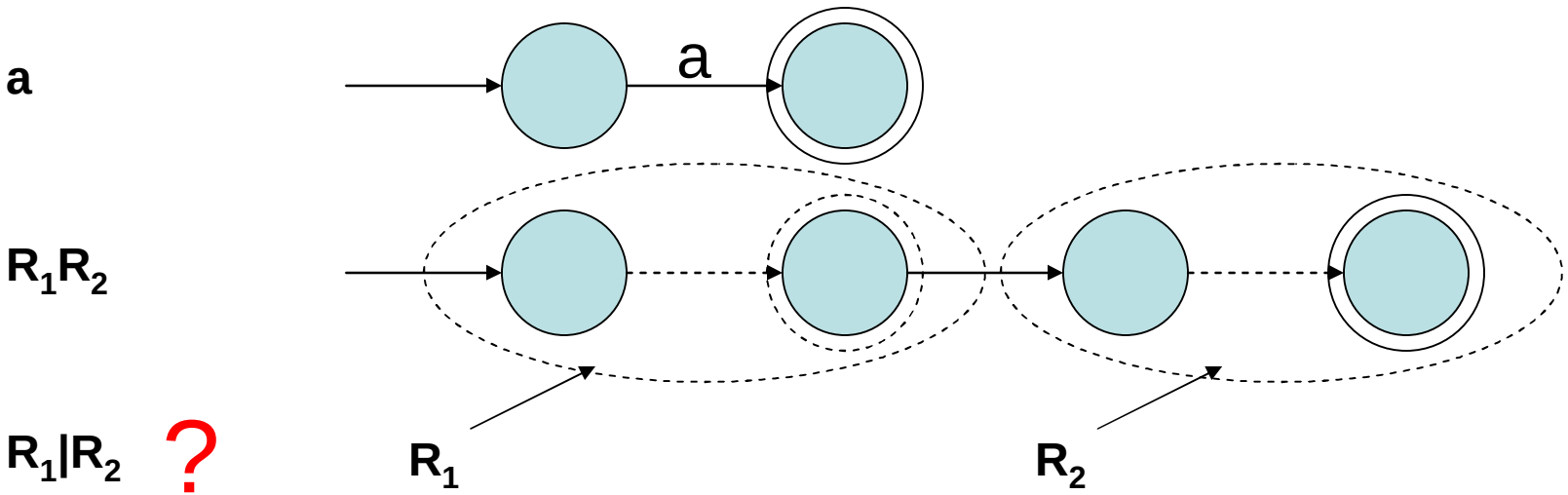
- Vòng lặp chính




```
boolean accept_state[NSTATES] = { … };
int trans_table[NSTATES][NCHARS] = { … };
int state = 0;
while (state != ERROR_STATE) {
   c = input.read();
   if (c < 0) break;
   state = trans_table[state][c];
}
return accept_state[state];
```

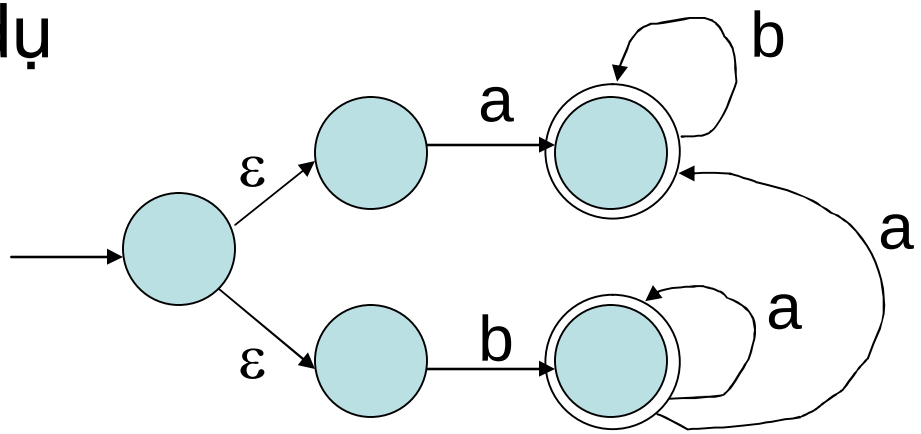
RE → FA

- Có thể chuyển RE bất kì thành FA hay không?
- Phương pháp: sử dụng định nghĩa đệ quy của RE



Ôtômát hữu hạn không đơn định (NFA)

- NFA bao gồm
  - Tập trạng thái, trạng thái bắt đầu, tập trạng thái kết thúc
  - Phép chuyển trạng thái bằng kí tự vào
  - Kí tự rỗng - ε (chuyển trạng thái không cần đọc kí tự của xâu vào)
  - Từ một trạng thái có thể chuyển đến nhiều trạng thái khác bằng cùng một kí tự vào
- Ví dụ



RE ?

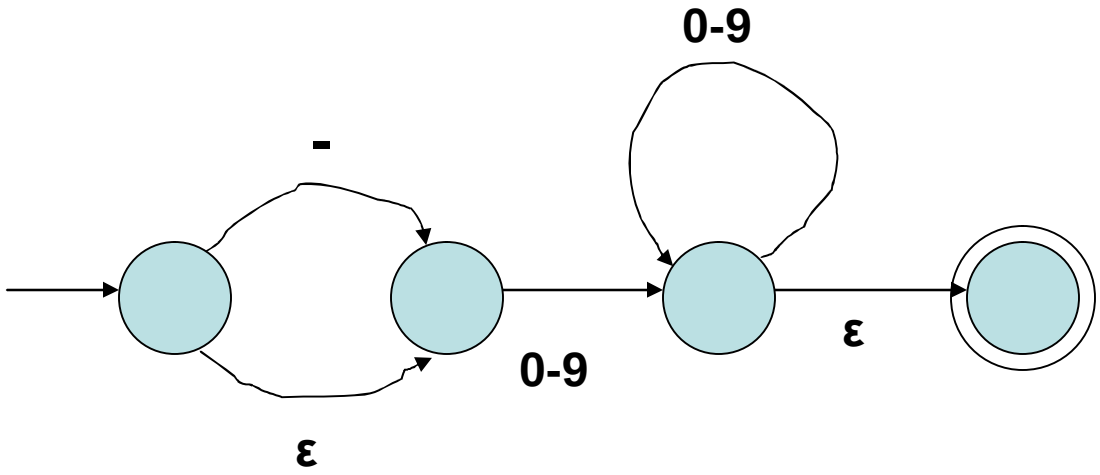
DFA và NFA

- DFA
  - Với mỗi kí tự vào, trạng thái của ôtômát luôn xác định
  - Cài đặt bằng bảng chuyển rất dễ dàng
- NFA
  - Các kí hiệu vào và kí tự rỗng cho phép có nhiều lựa chọn tại mỗi trạng thái
  - NFA chấp nhận xâu vào nếu có một dãy lựa chọn dẫn đến trạng thái kết thúc
  - Cài đặt NFA không dễ như DFA

RE → NFA

- **Ví dụ**

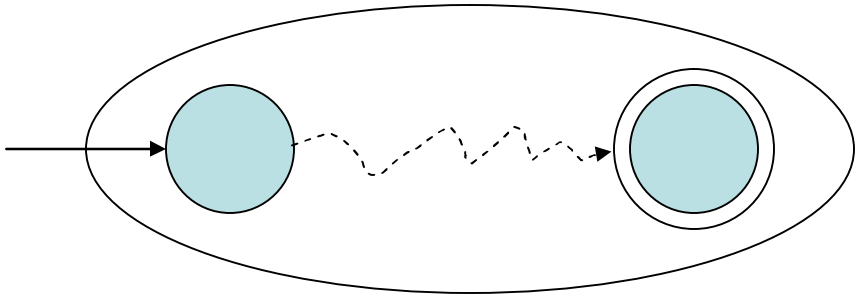
(-?) [0-9]+ **(-|ε) [0-9][0-9]***



RE → NFA

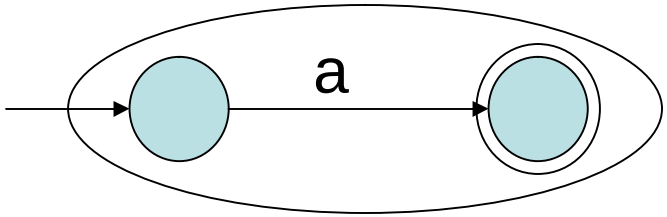
- Nhận xét:

NFA chỉ cần một trạng thái kết thúc ?

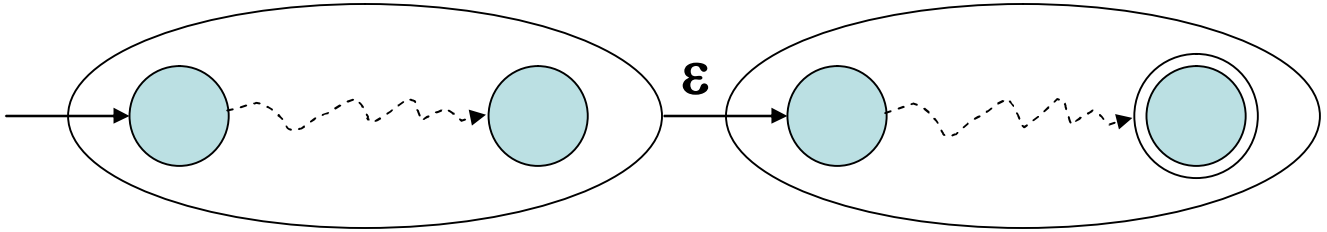


RE → NFA
(phương pháp quy nạp)

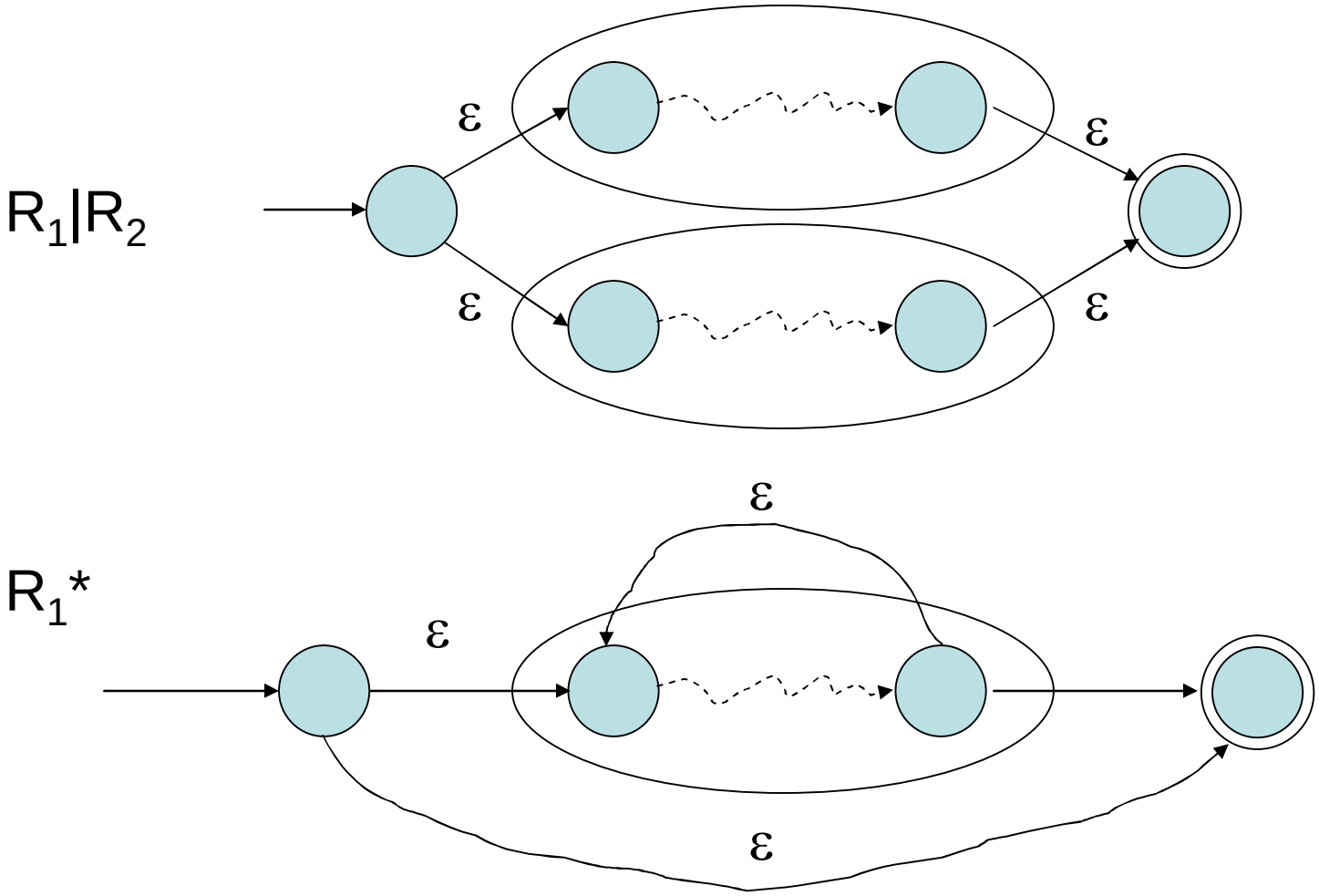
a



R_1R_2



RE → NFA
(phương pháp quy nạp)



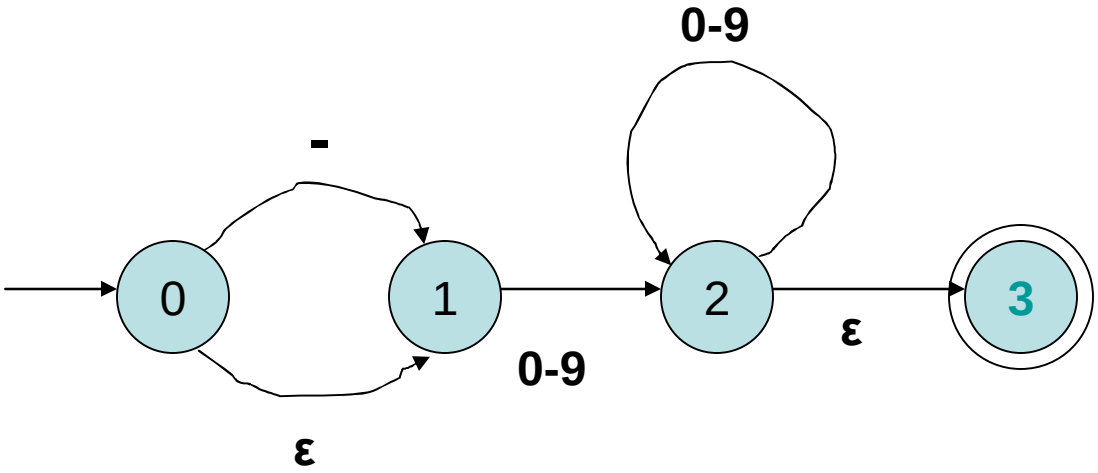
Cài đặt NFA

- Kiểm tra 1 xâu có thuộc ngôn ngữ của NFA
  - Kiểm tra xem có tồn tại 1 đường đi từ trại thái bắt đầu đến trạng thái kết thúc sử dụng các kí tự của xâu vào
  - Mỗi trạng thái có nhiều lựa chọn ?
- Tìm kiếm đồng thời nhiều đường đi
  - Lưu giữ tập trạng thái có thể đạt tới ứng với một đoạn đầu của xâu vào
  - Giống như ta chỉ vào nhiều trạng thái trên đồ thị cùng một lúc

Ví dụ

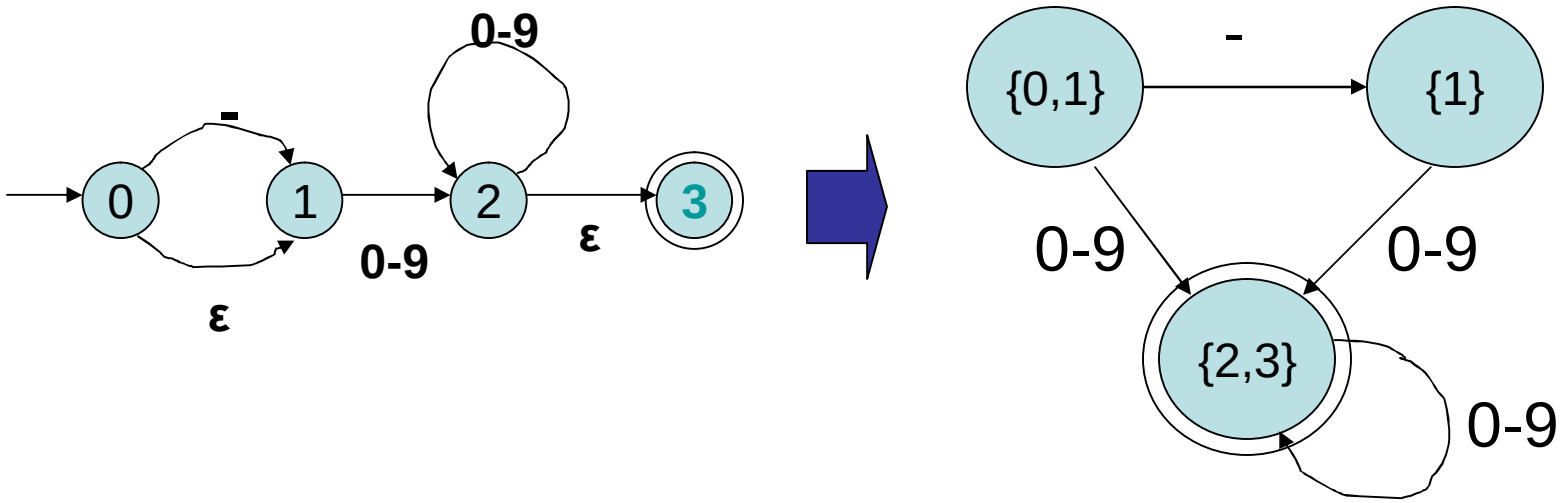
- Xâu vào: -23

	{0,1}
-	{1}
-2	{2, 3}
-23	{2, **3**}



Chuyển đổi NFA - DFA

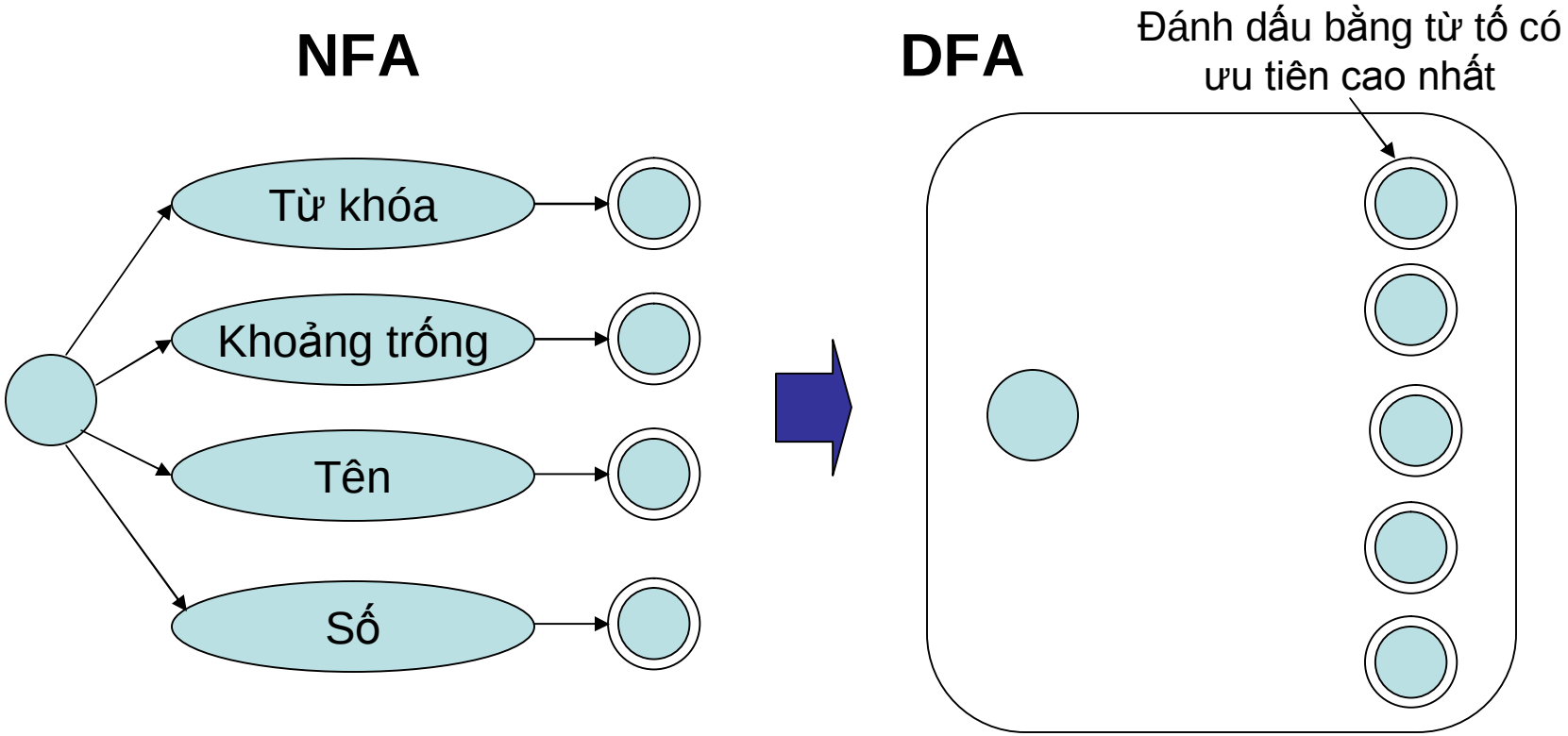
- Có thể chuyển NFA thành DFA bằng phương pháp tương tự
- Lập 1 trạng thái của DFA ứng với mỗi tập trạng thái có thể có của NFA



Tối ưu hóa DFA

- Chuyển đổi NFA sang DFA có thể tạo thành các DFA có số lượng trạng thái rất lớn ($\approx O(2^n)$)
- Các chương trình sinh ra bộ PTTV thường có bước tối ưu hóa DFA tới kích thước nhỏ nhất có thể được (xem tài liệu tham khảo số 3 – Aho, Sethi, Ullman)

Xử lý nhiều REs cùng lúc



- Cài đặt luật "dài nhất thắng" bằng DFA: khi có lỗi, nếu đang ở trạng thái kết thúc thì trả về từ tố tương ứng với trạng thái kết thúc đó

Tổng kết

- Chương trình PTTV chuyển đổi mã nguồn thành một dãy các từ tố
- RE có thể mô tả từ tố một cách chính xác
- RE và thứ tự ưu tiên có thể chuyển thành bộ PTTV qua 2 bước
  1. Chuyển RE → NFA
  2. Chuyển NFA → DFA (nếu có thể, tối ưu hóa DFA)
- Kết quả: Bộ PTTV ngắn gọn và dễ bảo trì
- Chuyển đổi NFA-DFA giải quyết hiệu quả vấn đề các từ tố có chung tiếp đầu ngữ (prefix)
  - Mã lệnh dễ thay đổi và bảo trì khi từ vựng thay đổi
  - Thường hiệu quả hơn là viết bằng tay
- Các chương trình sinh bộ PTTV đã có sẵn và miễn phí

Bài tập

- Bài 1: Xây dựng NFA đoán nhận ngôn ngữ được tạo bởi biểu thức chính quy if|[a-zA-Z_][a-zA-Z_0-9]*. Biến đổi NFA sang DFA.
- Bài 2: Xây dựng Automat đoán nhận ngôn ngữ được tạo bởi biểu thức chính quy biểu diễn chú thích trong ngôn ngữ lập trình C/C++

Nhập môn Chương trình dịch

Bài 4: Phân tích cú pháp

(syntactic analysis)

Nội dung chính

- Văn phạm phi ngữ cảnh (CFG)
- Dẫn xuất
- Cây suy dẫn và cây cú pháp
- Văn phạm nhập nhằng

Phân tích cú pháp

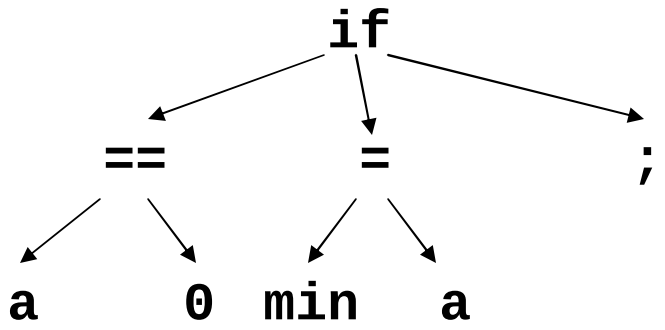
Mã nguồn (dãy các kí tự)

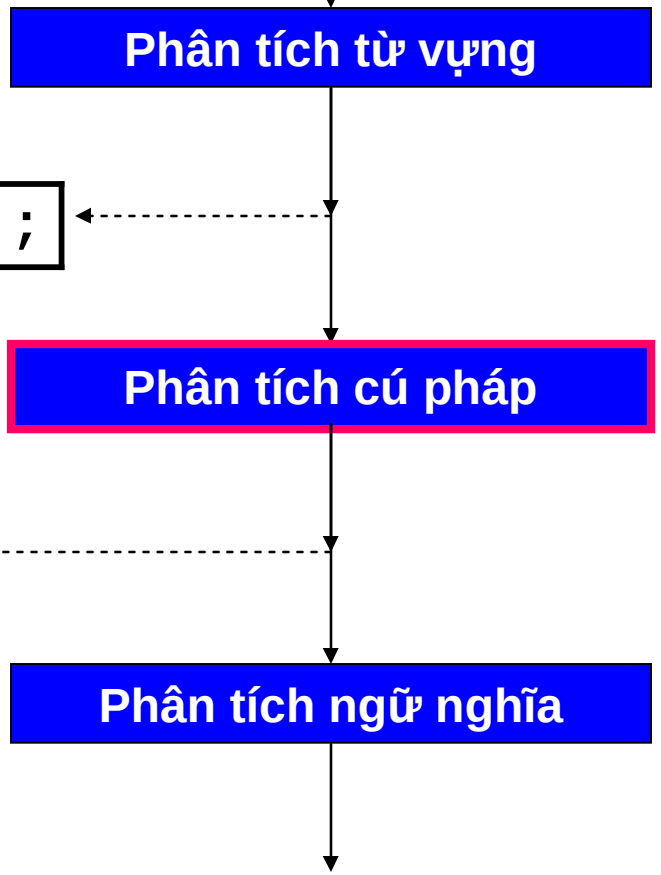
```
If (a == 0) min = a;
```

Dãy các từ tố (token)

If	(	Id:a	==	0	)	Id:min	=	Id:a	;

Cây cú pháp



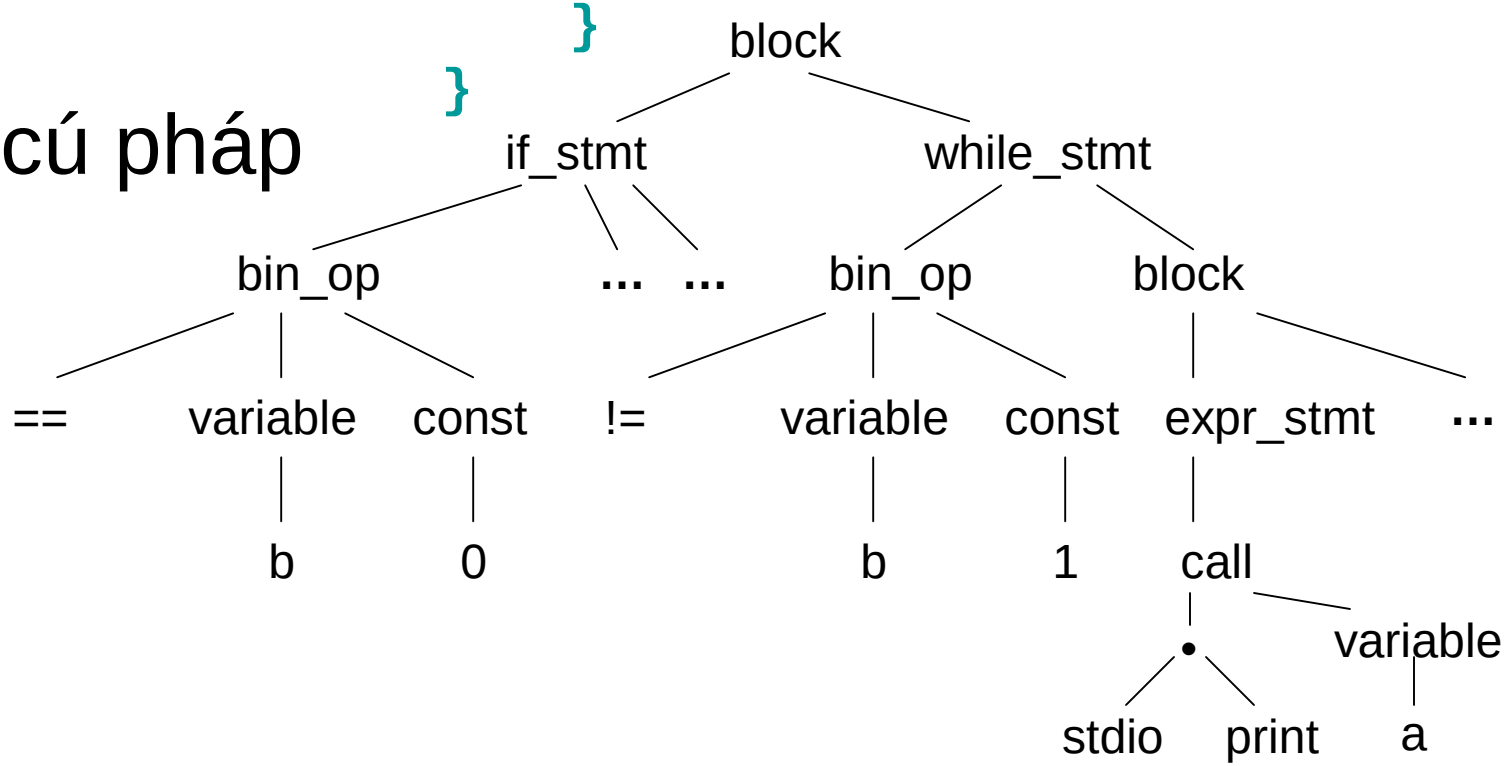


Phân tích cú pháp

- Mã nguồn

```
{
    if (b == (0)) a = b;
    while (a != 1) {
        stdio.print(a);
        a = a - 1;
    }
}
```

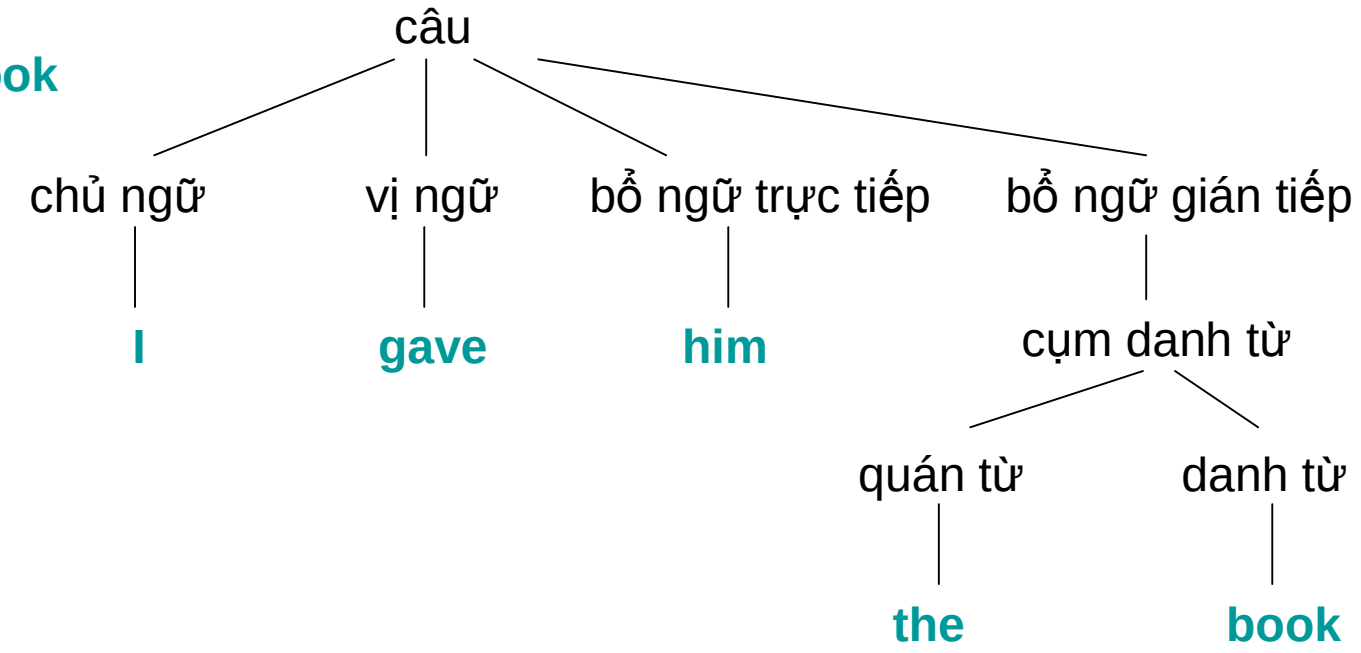
- Cây cú pháp



Phân tích cú pháp

- Kiểm tra tính đúng đắn về cú pháp của chương trình nguồn
- Xác định chức năng của các thành phần trong chương trình nguồn

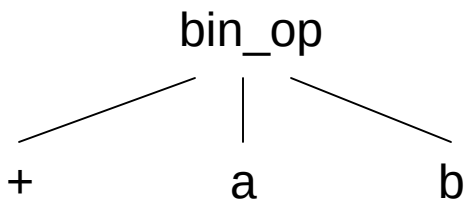
I gave him the book



Phân tích cú pháp

- Input: Dãy các từ tố
- Output: Cây cú pháp
- Cài đặt:
  - Duyệt qua dãy các từ tố
  - Xây dựng cây cú pháp
  - Loại bỏ các cú pháp thừa trong cây cú pháp

VD: a+b ≈ (a)+(b) ≈ ((a)+((b)))



Phân tích cú pháp

- Phân tích cú pháp không làm tất cả mọi công đoạn của chương trình dịch
- Ví dụ: kiểm tra kiểu, khai báo biến, khởi tạo biến
- Để lại cho phần phân tích ngữ nghĩa

Đặc tả cú pháp của ngôn ngữ

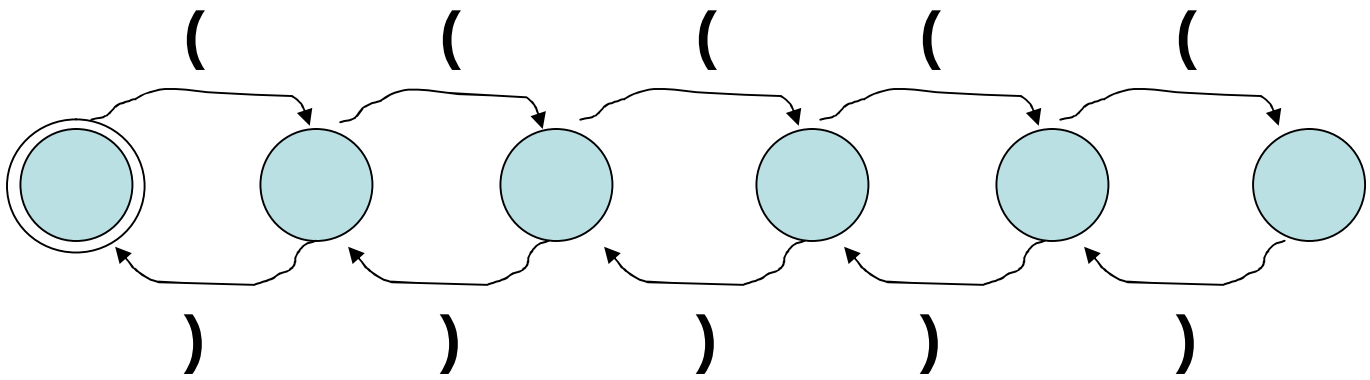
- Vấn đề: Làm thế nào để mô tả chính xác và dễ dàng cú pháp của ngôn ngữ tạo nên mã nguồn?
- Giống như từ tố được mô tả bằng REs
- REs dễ cài đặt (bằng NFA hoặc DFA)
- Có thể dùng REs để mô tả cú pháp của ngôn ngữ lập trình được không?

Giới hạn của REs

- Cú pháp của ngôn ngữ lập trình không thuộc lớp ngôn ngữ chính quy → không thể mô tả bằng REs được
- Ví dụ: L $\subseteq$ { (,) }* sao cho L là tập các cách viết () đúng.
- Nếu dùng RE để biểu diễn L → phải đếm số lượng dấu "(" chưa có dấu ")" tương ứng → số đếm không bị giới hạn

Cần cách mô tả mạnh hơn

- Ta biết: RE $\Leftrightarrow$ DFA
- Số đếm không giới hạn → số trạng thái không giới hạn → mâu thuẫn với cấu trúc của DFA (hữu hạn)



< 6: OK **>=6**

Văn phạm phi ngữ cảnh (CFG)

- Ví dụ: mô tả ngôn ngữ L

  $S \rightarrow (S)S$

  $S \rightarrow \varepsilon$

- CFG sử dụng định nghĩa đệ quy
- CFG trực quan hơn REs

  $S \Rightarrow (S)S \Rightarrow ((S)S)S \Rightarrow ((\varepsilon)S)S \Rightarrow \ldots \Rightarrow (())$

- Một xâu nằm trong ngôn ngữ của CFG nếu có một dãy suy dẫn sử dụng các **sản xuất** của CFG tạo nên xâu đó

Định nghĩa CFG

- Kí hiệu kết thúc: Từ tố hoặc ε
- Kí hiệu không kết thúc: Các biến cú pháp
- Kí hiệu bắt đầu: S
- Các sản xuất: $S \rightarrow (S)S$
  - Chỉ ra cách phát triển các kí hiệu không kết thúc thành các xâu
  - Vế trái: kí hiệu không kết thúc
  - Vế phải: xâu gồm kí hiệu kết thúc và không kết thúc
- Có thể gộp nhiều sản xuất có chung vế trái

$$S \rightarrow (S)S \mid \varepsilon$$

REs là tập con của CFG

- REs không có đệ quy

```
digit  = [0-9]
posint = digit+
int    = -? posint
real   = int (ε | (. posint))
```

- Về trái của REs có thể phát triển đến các kí hiệu vào

```
real   = -?[0-9]+(ε | (. [0-9]+))
```

Ví dụ (1)

S → E + S | E

E → số | (S)

- Xâu (1 + 2 + (3 + 4)) + 5

S → E + S

S → E

E → số

E → (S)

2 kí hiệu không kết thúc: E, S

4 kí hiệu kết thúc: số, (,), +

4 sản xuất

Kí hiệu bắt đầu S

Ví dụ (2)

kí hiệu không kết thúc – vế trái

S → E + S | E

E → số | (S)

về phải của sản xuất

- Xâu (1 + 2 + (3 + 4)) + 5

S ⇒ **E** + S ⇒ (**S**) + S ⇒ (**E** + S) + S

⇒ (1 + **S**) + S ⇒ (1 + **E** + S) + S

⇒ (1 + 2 + **S**) + S ⇒ (1 + 2 + **E**) + S

⇒ (1 + 2 + (**S**)) + S ⇒ (1 + 2 + (**E** + S)) + S

⇒ (1 + 2 + (3 + **S**)) + S ⇒ (1 + 2 + (3 + **E**)) + S

⇒ (1 + 2 + (3 + 4)) + **S** ⇒ (1 + 2 + (3 + 4)) + **E**

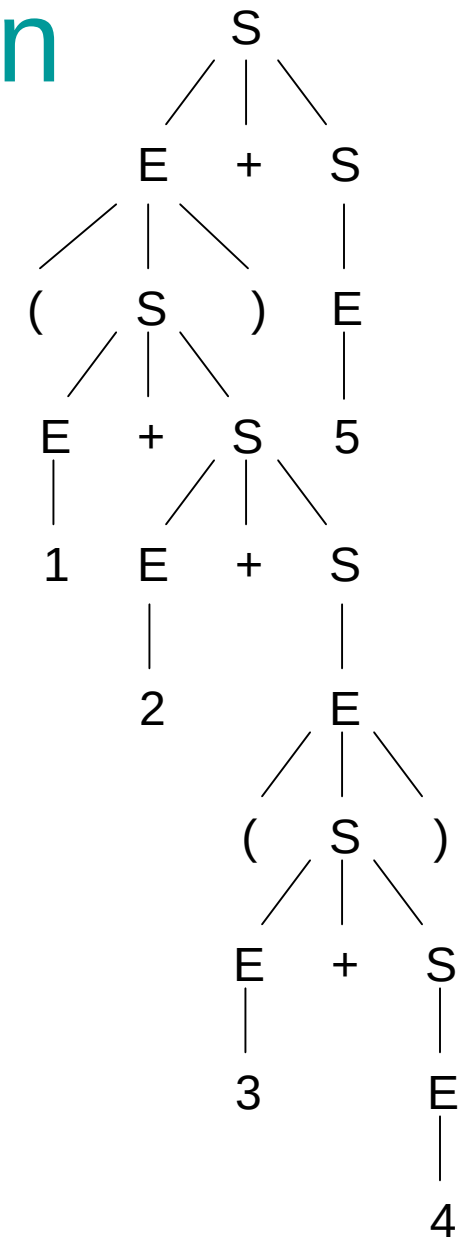
⇒ (1 + 2 + (3 + 4)) + 5

Dẫn xuất (1)

- Bắt đầu từ S
- Sử dụng dẫn xuất để tạo nên dãy các từ tố (kí hiệu kết thúc)
- Với các xâu α, β và γ bất kì và 1 sản xuất $A \rightarrow \beta$
- Một dẫn xuất là $\alpha A \gamma \Rightarrow \alpha\beta\gamma$
  - Thay β vào vị trí của A ở vế trái
- Ví dụ:
  - (**S** + E) + E $\Rightarrow$ (**E + S** + E)+E
  - trong đó (A = S, β = E + S)

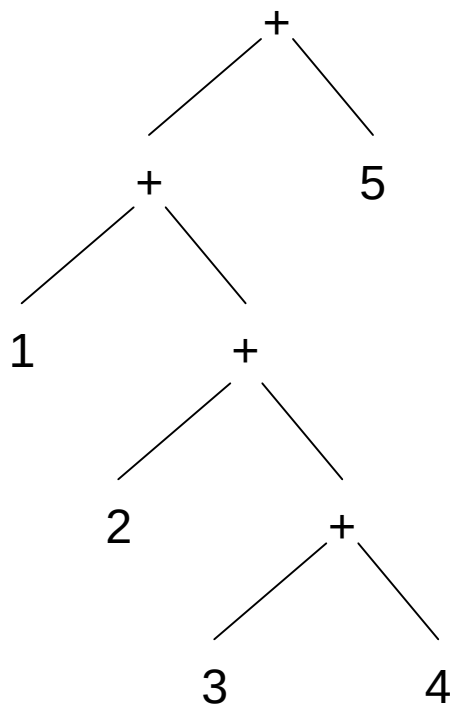
Cây suy dẫn

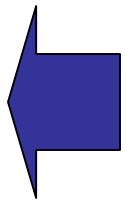
- Một dãy dẫn xuất bắt đầu từ S có thể mô tả dưới dạng cây suy dẫn
  - Lá của cây là kí hiệu kết thúc; theo thứ tự duyệt sẽ tạo thành xâu vào
  - Nút trong của cây là các kí hiệu không kết thúc
  - Cây không chỉ rõ thứ tự của các dẫn xuất

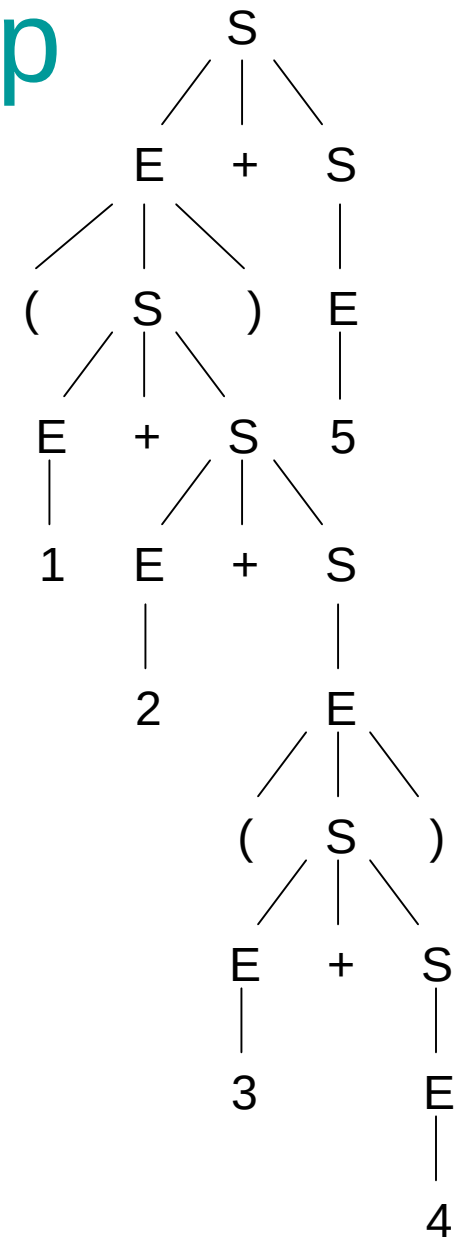


Cây cú pháp

- Giản lược các thông tin thừa khỏi cây suy dẫn







Dẫn xuất (2)

- Thứ tự dẫn xuất tùy ý, có thể chọn bất cứ một kí hiệu không kết thúc nào để áp dụng sản xuất
- Hai thứ tự dẫn xuất thường dùng:
  - Suy dẫn trái: chọn kí hiệu bên trái nhất
  - Suy dẫn phải: chọn kí hiệu bên phải nhất
- Được sử dụng trong nhiều kiểu phân tích cú pháp tự động (automatic parsing)

Ví dụ

- Suy dẫn trái

  S ⇒ ***E***+S ⇒ (***S***) + S ⇒ (***E*** + S)+ S ⇒ (1 + ***S***)+S
  ⇒ (1+***E***+S)+S ⇒ (1+2+***S***)+S ⇒ (1+2+***E***)+S
  ⇒ (1+2+(***S***))+S ⇒ (1+2+(***E***+S))+S ⇒ (1+2+(3+***S***))+S
  ⇒ (1+2+(3+***E***))+S ⇒ (1+2+(3+4))+***S*** ⇒ (1+2+(3+4))+***E***
  ⇒ (1+2+(3+4))+5

- Suy dẫn phải

  S ⇒ E+***S*** ⇒ E+***E*** ⇒ ***E***+5 ⇒ (***S***)+5 ⇒ (E+***S***)+5
  ⇒ (E+E+***S***)+5 ⇒ (E+E+***E***)+5 ⇒ (E+E+(***S***))+5
  ⇒ (E+E+(E+***S***))+5 ⇒ (E+E+(E+***E***))+5
  ⇒ (E+E+(***E***+4))+5 ⇒ (E+***E***+(3+4))+5
  ⇒ (***E***+2+(3+4))+5 ⇒ (1+2+(3+4))+5

- Cùng một cây suy dẫn, cùng sử dụng các dẫn xuất nhưng theo thứ tự khác nhau

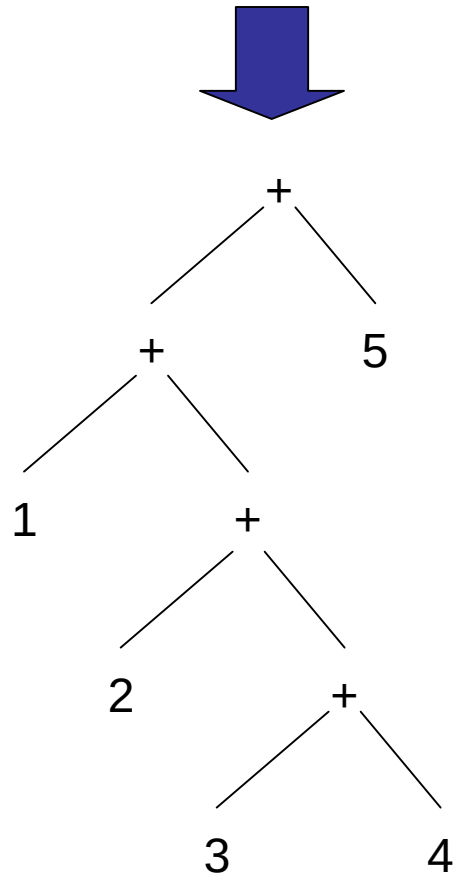
Văn phạm nhập nhằng (1)

- Ở ví dụ trước, cả hai cây suy dẫn trái và phải đều giống nhau
- Lý do: phép cộng (+) có xu hương kết hợp về bên phải bất kể thứ tự dẫn xuất như thế nào
- Lý do: Phép cộng được định nghĩa trong văn phạm

$$S \rightarrow E + S$$

→ Đệ quy phải

(1 + 2 + (3 + 4)) + 5



Văn phạm nhập nhằng (2)

- Xét văn phạm sau

$$S \rightarrow S + S \mid S * S \mid \text{number}$$

- Sử dụng dẫn xuất khác nhau cho ra các cây suy dẫn khác nhau
- Văn phạm nhập nhằng

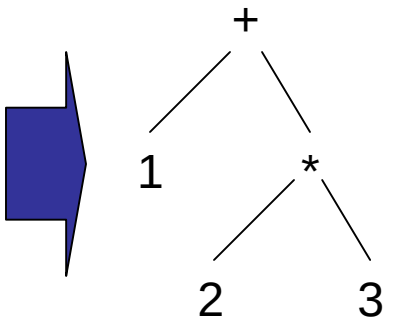
Văn phạm nhập nhằng (3)

$S \rightarrow S + S \mid S * S \mid \text{number}$

- Nếu xâu vào là 1 + 2 * 3
- Suy dẫn 1:

  $S \Rightarrow S + S \Rightarrow 1 + S \Rightarrow 1 + S * S$

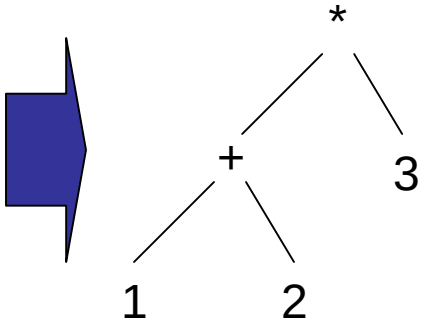
  $\Rightarrow 1 + 2 * S \Rightarrow 1 + 2 * 3$



- Suy dẫn 2:

  $S \Rightarrow S * S \Rightarrow S + S * S \Rightarrow 1 + S * S$

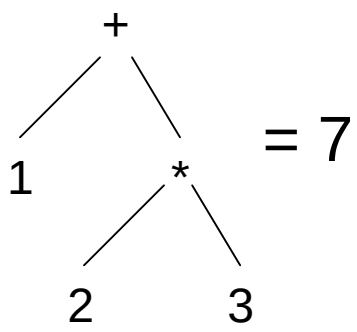
  $\Rightarrow 1 + 2 * S \Rightarrow 1 + 2 * 3$



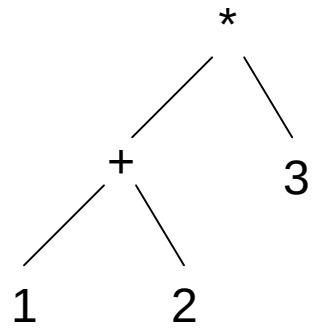
Ý nghĩa

- Cây suy dẫn khác nhau cho kết quả tính toán khác nhau
- Văn phạm nhập nhằng

  → Có nhiều cách hiểu chương trình nguồn



= 7



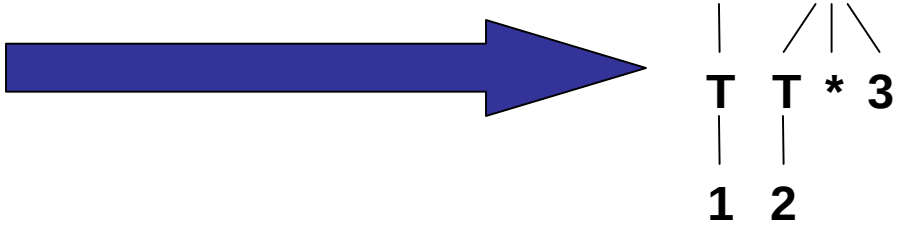
= 9

Loại bỏ nhập nhằng

- Có thể loại bỏ nhập nhằng bằng
  - Thêm vào một số kí hiệu không kết thúc
  - Chỉ cho phép sử dụng đệ quy trái hoặc phải

$S \Rightarrow S + T \mid T$

$T \Rightarrow T * \mathbf{num} \mid \mathbf{num}$



- T: kí hiệu không kết thúc cho phép chỉ ra thứ tự ưu tiên của các phép toán
- Đệ quy trái: các phép toán có xu hướng kết hợp bên trái

Giới hạn của CFG

- Vẫn chưa thể bắt hết các lỗi cú pháp
- Ví dụ: C++

```
HashTable<Key,Value> x;
```

- Cần kiểm tra HashTable là kiểu gì?
- Các kí hiệu “<”, “,” được nạp chồng (overload)
- Ví dụ: C++

```
f[4][5][6] = x;
```

- Khó dùng CFG để mô tả vế trái
- Ý tưởng: cho phép cả 2 vế đều là các biểu thức, kiểm tra vế trái sau (phân tích ngữ nghĩa)

Tổng kết

- CFG cho phép mô tả cú pháp của mã nguồn khá chính xác và ngắn gọn
- CFG cho phép mô tả cách chuyển dãy các từ tố thành cây suy dẫn (cây cú pháp)

Nhập môn Chương trình dịch

Bài 05: Phân tích trên xuống

(Top – down parsing)

Nội dung chính

- Tiếp tục với CFG
- Phân tích trên xuống
- Lớp ngôn ngữ LL(1)
- Chuyển văn phạm về dạng LL(1)
- Phân tích đệ quy xuống (recursive descent)

Phân tích cú pháp

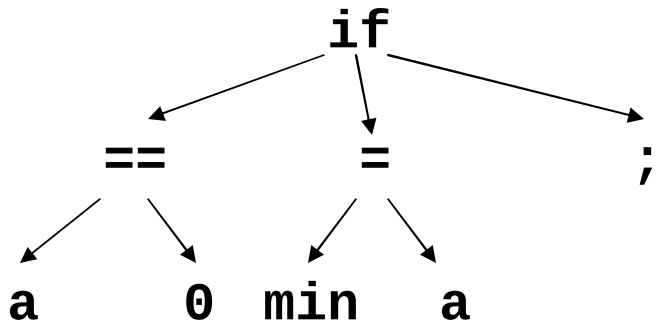
Mã nguồn (dãy các kí tự)

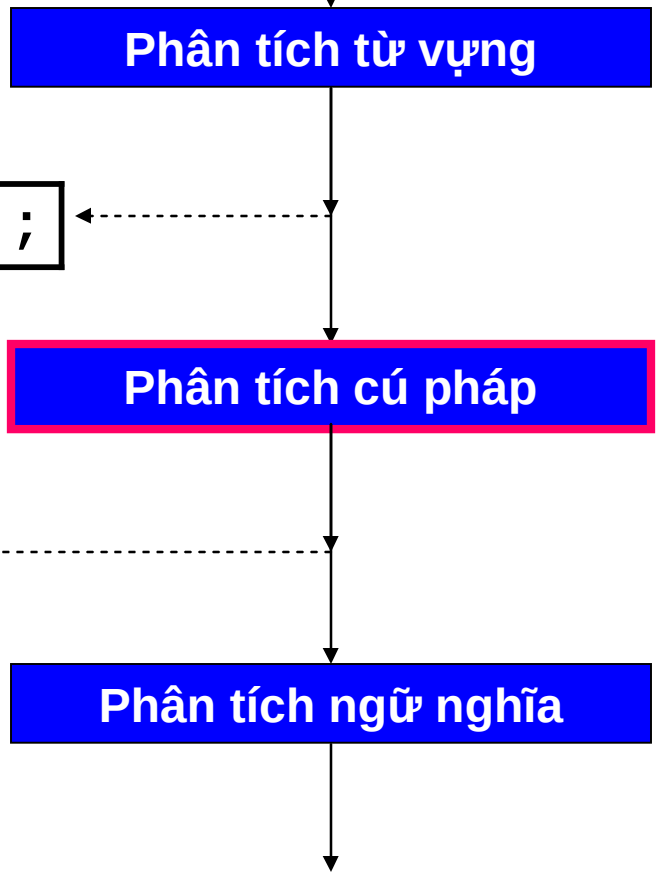
```
If (a == 0) min = a;
```

Dãy các từ tố (token)

If	(	Id:a	==	0	)	Id:min	=	Id:a	;

Cây cú pháp





Văn phạm phi ngữ cảnh (CFG)

- CFG có thể mô tả cú pháp của ngôn ngữ lập trình
- CFG có khả năng diễn tả các cú pháp lồng nhau (VD: dấu ngoặc, các lệnh lồng nhau)
- Một xâu nằm trong ngôn ngữ của CFG nếu có một suy dẫn từ kí hiệu bắt đầu sinh ra xâu đó
- Vấn đề: Văn phạm nhập nhằng

if-then-else

- Văn phạm cho câu lệnh if **?**

  $S \rightarrow$ if (E) S

  $S \rightarrow$ if (E) S else S

  $S \rightarrow X = E$ | if (E) S else S

- Văn phạm có mô tả được câu lệnh if không?

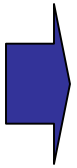
if-then-else

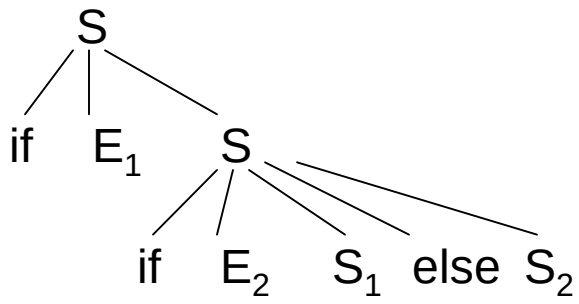
- Phân tích câu sau

if (E_1) if (E_2) S_1 else S_2

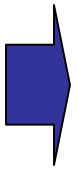
$S \rightarrow$ if (*E*) *S*
$S \rightarrow$ if (*E*) *S* else *S*
$S \rightarrow$ *other*

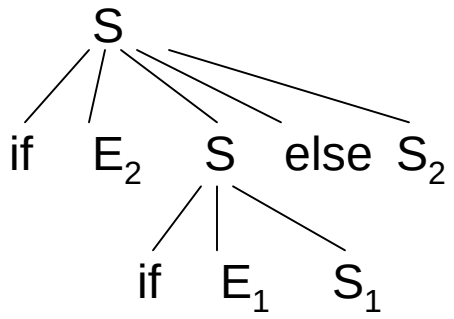
$S \Rightarrow$ if (E_1) **S**
$\Rightarrow$ if (E_1) if (E_2) S_1 else S_2





$S \Rightarrow$ if (E_1) **S** else S_2
$\Rightarrow$ if (E_1) if (E_2) S_1 else S_2



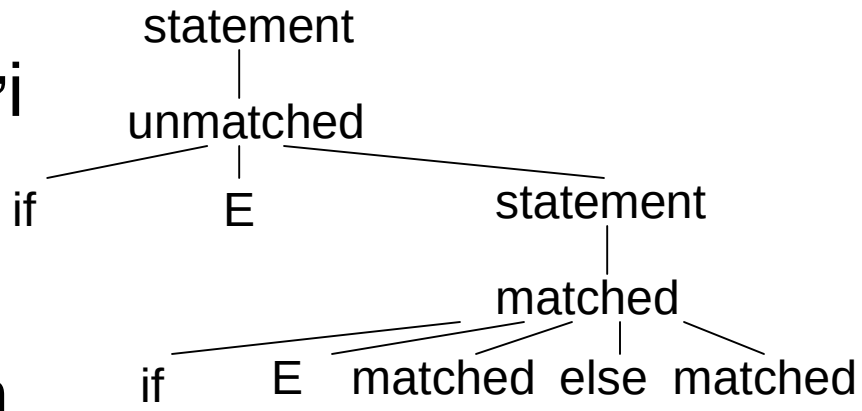


- else đi với if nào?

if-then-else

- Ta không muốn else đi với if đầu tiên

  if (*E*) if (*E*) *S* else *S*



- Vấn đề: Không có gì phân biệt 2 kí hiệu S với nhau
- Sửa lại văn phạm

  statement → *matched* | *unmatched*
  matched → if (*E*) *matched* else *matched* | *other*
  unmatched → if (*E*) *matched* else *unmatched*
  |
  if (*E*) *statement*

Phân tích trên xuống (top-down)

- Văn phạm có thể phân tích trên xuống
- Cài đặt bộ phân tích cú pháp trên xuống (recursive descent parser)
- Xây dựng cây cú pháp

Phân tích trên xuống

S → E + S | E
E → số | (S)

- Mục tiêu: xây dựng cây **suy dẫn trái** trong khi đọc dãy từ tố

Suy dẫn	Từ tố nhìn trước	Dãy từ tố Đã đọc / Chưa đọc
S	(	(1+2+(3+4))+5
E+S	(	(1+2+(3+4))+5
(S)+S	1	(1+2+(3+4))+5
(E+S)+S	1	(1+2+(3+4))+5
(1+S)+S	2	(1+2+(3+4))+5
(1+E+S)+S	2	(1+2+(3+4))+5
(1+2+S)+S	(	(1+2+(3+4))+5
(1+2+E)+S	(	(1+2+(3+4))+5
(1+2+(S))+S	3	(1+2+(3+4))+5

Vấn đề

S → E + S | E
E → số | (S)

- Ta muốn lựa chọn sản xuất dựa vào từ tố nhìn trước

(1) **S** ⇒ **E** ⇒ (**S**) ⇒ (**E**) ⇒ (1)

(1)+2 **S** ⇒ **E** + S ⇒ (**S**) + S ⇒ (**E**) + S

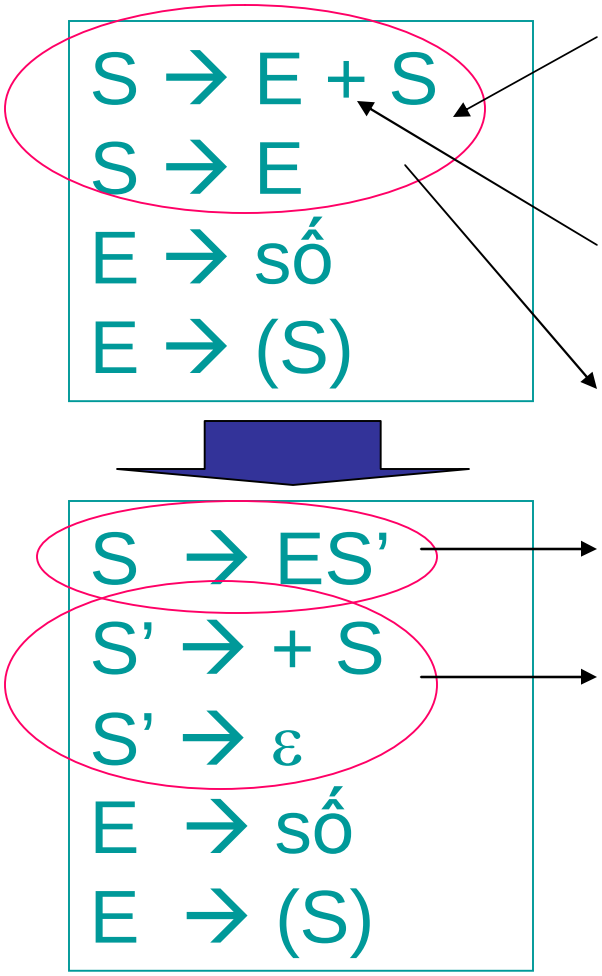
⇒ (1)+**E** ⇒ (1)+2

- Với văn phạm này ta không lựa chọn được

Vấn đề ở văn phạm

- Văn phạm này không thể phân tích trên xuống nếu chỉ nhìn trước 1 kí tự
- Không phải thuộc lớp văn phạm LL(1)
- **L**eft–to–right scanning
  Left–most derivation
  1 token lookahead
- Có thể viết lại văn phạm, cho phép phân tích trên xuống
- Tức là, văn phạm LL(1) cho cùng ngôn ngữ

Viết lại văn phạm - LL(1)



Không lựa chọn được khi
kí hiệu không kết thúc là S

phải nhìn thấy dấu "+"
để quyết định

Nhận xét: $S \Rightarrow E(+S)^*$

Chuyển việc lựa chọn cho S'

$S' \Rightarrow (+S)^*$

➢ Left factoring

Phân tích trên xuống với văn phạm LL(1)

S	(	(1+2+(3+4))+5
ES’	(	(1+2+(3+4))+5
(S)S’	1	(1+2+(3+4))+5
(ES’)S’	1	(1+2+(3+4))+5
(1S’)S’	+	(1+2+(3+4))+5
(1+S)S’	2	(1+2+(3+4))+5
(1+ES’)S’	2	(1+2+(3+4))+5
(1+2S’)S’	+	(1+2+(3+4))+5
(1+2+S)S’	(	(1+2+(3+4))+5
(1+2+ES’)S’	(	(1+2+(3+4))+5
(1+2+(S)S’)S’	3	(1+2+(3+4))+5
(1+2+(ES’)S’)S’	3	(1+2+(3+4))+5
(1+2+(3S’)S’)S’	+	(1+2+(3+4))+5
(1+2+(3+S)S’)S’	4	(1+2+(3+4))+5

Phân tích tất định

- Lớp văn phạm LL(1):
  - Với mỗi kí hiệu không kết thúc, từ tố nhìn trước sẽ xác định sản xuất phải sử dụng
  - Phân tích trên xuống $\Leftrightarrow$ phân tích tất định
  - Cài đặt bằng bảng phân tích

    Kí hiệu không kết thúc x ký hiệu kết thúc $\Rightarrow$ sản xuất

Bảng phân tích

S	(	(1+2+(3+4))+5
ES’	(	(1+2+(3+4))+5
(S)S’	1	(1+2+(3+4))+5
(ES’)S’	1	(1+2+(3+4))+5
(1S’)S’	+	(1+2+(3+4))+5
(1+S)S’	2	(1+2+(3+4))+5
(1+ES’)S’	2	(1+2+(3+4))+5
(1+2S’)S’	+	(1+2+(3+4))+5

	số	+	(	)	$ - EOF
S	→ ES’		→ ES’		
S’		→ +S		→ ε	→ ε
E	→ số		→(S)		

Cài đặt

- Bảng phân tích được dùng trong phân tích đệ quy xuống (recursive descent)

	số	+	(	)	$ - EOF
S	→ ES’		→ ES’		
S’		→ +S		→ ε	→ ε
E	→ số		→(S)		

- Cài đặt 3 thủ tục: parseS, parseS’, parseE

Phân tích đệ quy xuống

```
void parse_S () {
   switch (token) {
       case num: parse_E(); parse_S'(); return;
       case '(': parse_E(); parse_S'(); return;
       default: throw new ParseError();
   }
}
```

từ tố nhìn trước

	số	+	(	)	$ - EOF
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ số		→(S)		

Phân tích đệ quy xuống

```
void parse_S'() {
   switch (token) {
        case '+': token = input.read(); parse_S(); return;
        case ')': return;
        case EOF: return;
        default: throw new ParseError();
   }
}
```

	số	+	(	)	$ - EOF
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ số		→(S)		

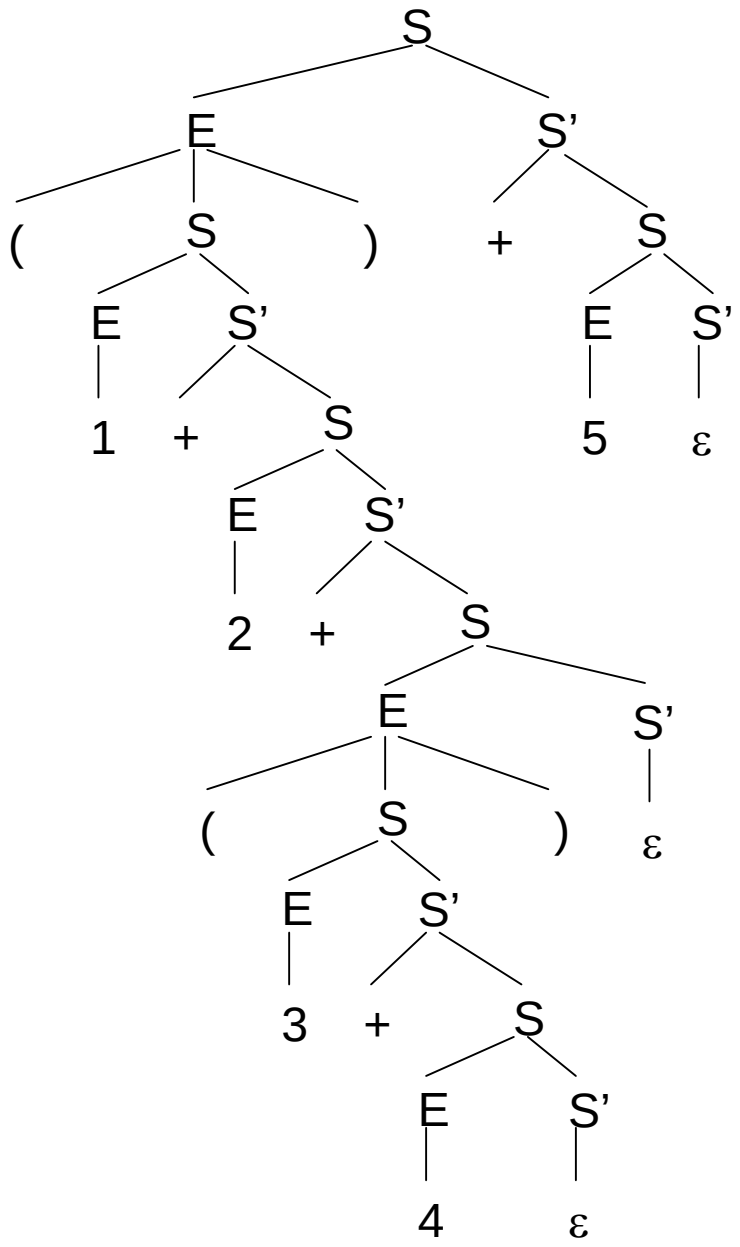
Phân tích đệ quy xuống

```
void parse_E() {
   switch (token) {
       case number: token = input.read(); return;
       case '(':
              token = input.read(); parse_S();
              if (token != ')') throw new ParseError();
              token = input.read(); return;
              default: throw new ParseError();
   }
}
```

	số	+	(	)	$ - EOF
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ số		→(S)		

Cây hàm = Cây suy dẫn

- Thứ tự và cấp bậc các lời gọi hàm trùng với cây suy dẫn

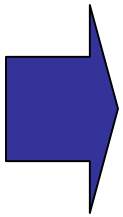


Xây dựng bảng phân tích (1)

- Tự động xây dựng bảng phân tích từ văn phạm cho trước thuộc lớp LL(1)

S → ES’
S’ → + S
S’ → ε
E → số
E → (S)

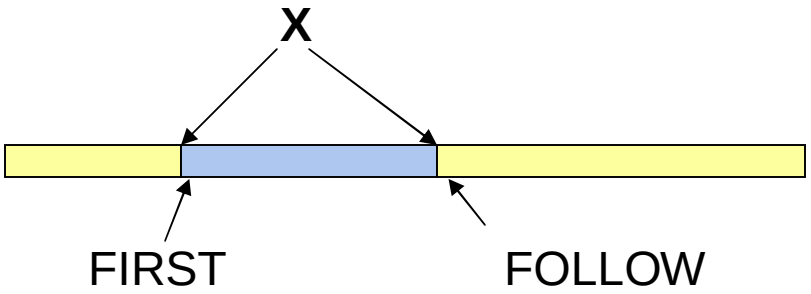
?



	số	+	(	)	$
S	→ ES’		→ ES’		
S’		→ +S		→ ε	→ ε
E	→ số		→(S)		

Xây dựng bảng phân tích (2)

- Phân tích tất định: Với mỗi ký hiệu không kết thúc, từ tố nhìn trước sẽ xác định sản xuất cần sử dụng
- Định nghĩa:
- FIRST(γ) với γ là xâu bất kì gồm các kí hiệu kết thúc và không kết thúc là: tập hợp các ký hiệu có thể bắt đầu xâu suy dẫn được từ γ.
- FOLLOW(X) với X là kí hiệu không kết thúc là : tập hợp các kí hiệu có thể theo sau xâu suy dẫn được từ X trong xâu vào.



Xây dựng bảng phân tích (3)

- Xét sản xuất dạng **X → γ**
- Đặt "**→ γ**" vào dòng **X**, các cột nằm trong FIRST(γ)

	số	+	(	)	$
S	→ ES'		→ ES'		
S'		→ +S		→ ε	→ ε
E	→ số		→(S)		

- Nếu từ **γ** có thể suy dẫn ra ε (triệt tiêu được - nullable) đặt "**→ γ**" vào dòng **X**, các cột nằm trong FOLLOW(X)
- Văn phạm là LL(1) nếu không có ô nào được điền quá 1 lần

Tính các ký hiệu triệt tiêu được

- **$\gamma = X_1 X_2 .. X_n$** triệt tiêu được nếu X_i triệt tiêu được (i = 1,2, .., n)
- Đệ quy:
  - X → ε: X triệt tiêu được
  - X → $Y_1 Y_2 .. Y_n$ triệt tiêu được nếu Y_i triệt tiêu được (i = 1, 2, ..n)
- Thuật toán: sử dụng 2 luật trên liên tục để đánh dấu các ký hiệu triệt tiêu được đến khi không đánh dấu thêm được ký hiệu nào

Tính FIRST(γ)

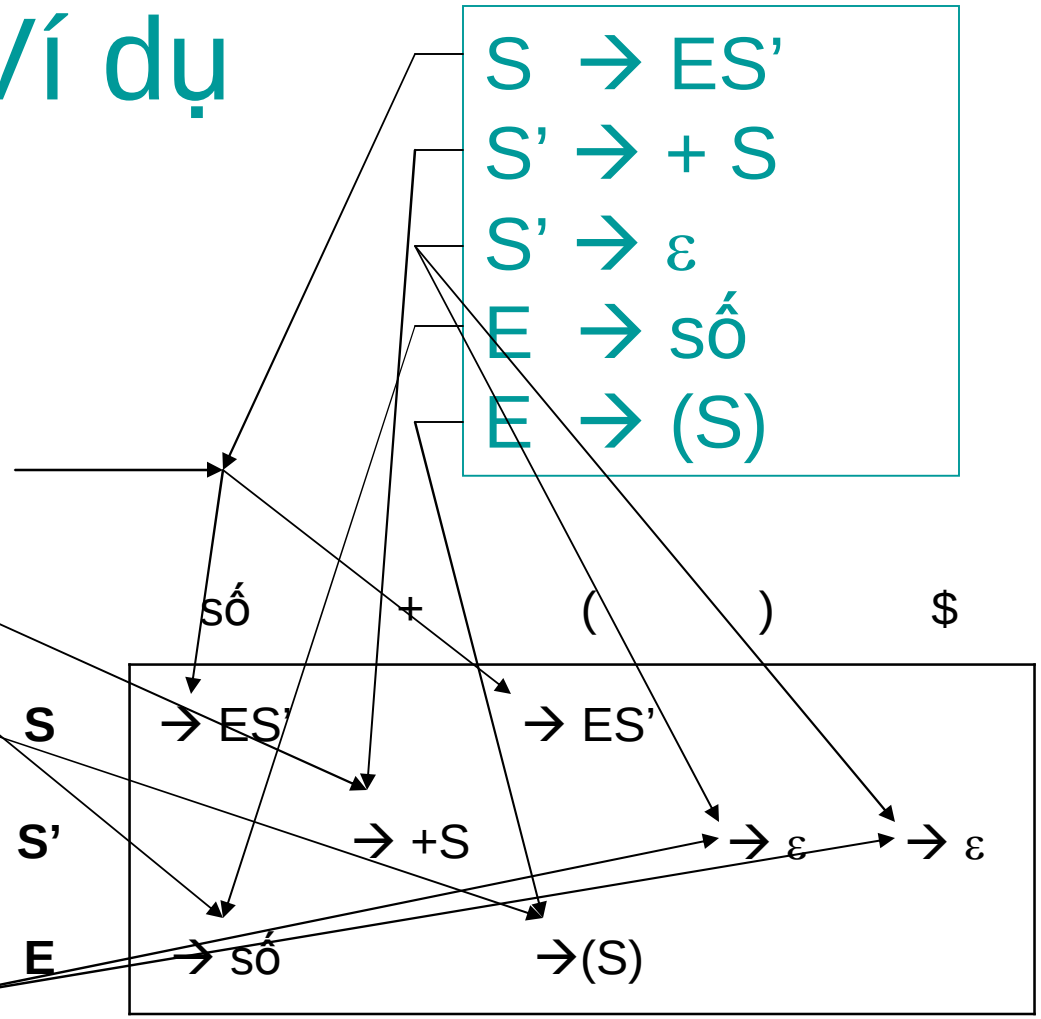
- FIRST(X) $\supseteq$ FIRST(γ) nếu X → γ
- FIRST(aβ) = {a}
- FIRST(Xβ) $\supseteq$ FIRST(X)
- FIRST(Xβ) $\supseteq$ FIRST(β) nếu X triệt tiêu được
- Thuật toán: Giả sử với mọi γ, FIRST(γ) rỗng, áp dụng các luật trên liên tục để xây dựng các tập FIRST.

Tính FOLLOW(X)

- FOLLOW(S) $\supseteq$ {$}
- Nếu X → αYβ
  - FOLLOW(Y) $\supseteq$ FIRST(β)
  - FOLLOW(Y) $\supseteq$ FOLLOW(X) nếu β→ ε
- Thuật toán: Giả sử với mọi X, FOLLOW(X) rỗng, áp dụng các luật trên đến khi không có thay đổi nào

Ví dụ

- Triệt tiêu được
  - Chỉ có S' triệt tiêu được
- FIRST
  - FIRST(E S') = {**số**, **(** }
  - FIRST(+S) = { **+** }
  - FIRST(**số**) = {**số**}
  - FIRST(**(**S**)**) = { **(** }
  - FIRST(S') = { + }
- FOLLOW
  - FOLLOW(S) = { **$**, **)** }
  - FOLLOW(S') = {**$**, **)**}
  - FOLLOW(E) = { **+**, **)**, **$**}



Văn phạm nhập nhằng

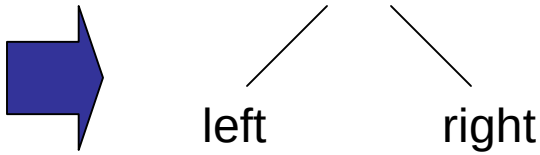
- Bảng phân tích của văn phạm nhập nhằng sẽ có ô được ghi hơn 1 lần (xung đột)
- Ngược lại, văn phạm có ô xung đột có phải là văn phạm nhập nhằng không?

Finish him

- Thủ tục đệ quy xuống đã xây dựng được cây suy dẫn
- Cây suy dẫn vẫn còn nhiều chi tiết thừa. Ví dụ: các dấu “(“, “)”, “ε”
- Bước cuối cùng: xây dựng cây cú pháp

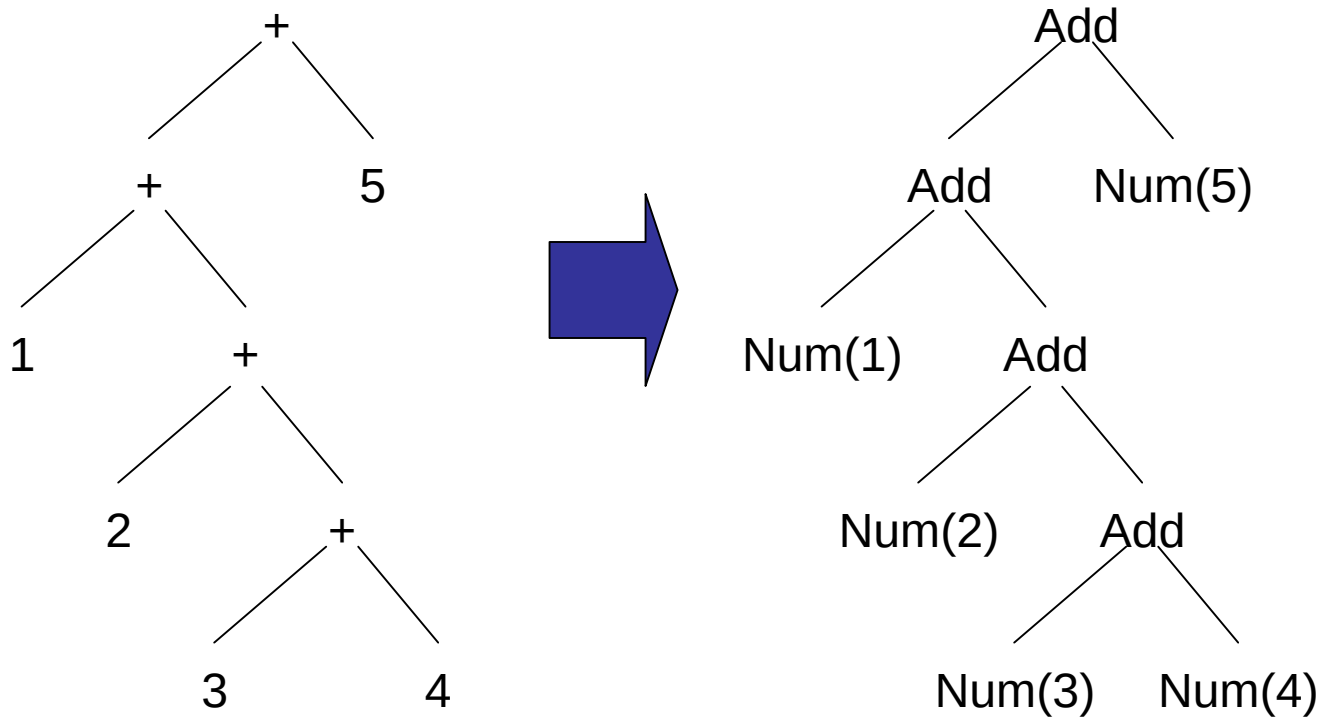
Xây dựng cây cú pháp (1)

```
abstract class Expr { }

class Add extends Expr {
   Expr left, right;
   Add(Expr L, Expr R) {
       left = L; right = R;
   }
}

class Num extends Expr {
   int value;
   Num (int v) {
       value = v;
   }
}
```



Xây dựng cây cú pháp (2)

- (1+2+(3+4))+5



Xây dựng cây cú pháp (3)

- Ý tưởng: cây cú pháp có cùng cấu trúc với cây suy dẫn
- Cài đặt thêm mã lệnh vào các thủ tục đệ quy xuống để xây dựng cây cú pháp
- parse_S, parse_S', parse_E cùng trả về kiểu Expr

```
void parse_E()    ⇒ Expr parse_E()
void parse_S()    ⇒ Expr parse_S()
void parse_S'()   ⇒ Expr parse_S'()
```

Xây dựng cây cú pháp (4)

```
Expr parse_E() {
   switch(token) {
       case num: // E → number
              Expr result = Num (token.value);
              token = input.read(); return result;
       case ‘(‘: // E → ( S )
              token = input.read();
              Expr result = parse_S();
              if (token != ‘)’) throw new ParseError();
              token = input.read(); return result;
       default: throw new ParseError();
   }
}
```

Xây dựng cây cú pháp (5)

```
Expr parse_S() {
  switch (token) {
      case num:
      case ‘(’:
            Expr left = parse_E();
            Expr right = parse_S’();
            if (right == null) return left;
            else return new Add(left, right);
      default: throw new ParseError();
  }
}
```

Xây dựng cây cú pháp (6)

```
Expr parse_S'() {
  switch (token) {
      case '+':
            token = input.read();
            return parse_S();
      case ')': return null;
      case EOF: return null;
      default: throw new ParseError();
  }
}
```

Thông dịch

```
int parse_E() {
   switch(token) {
        case num: // E → number
                int result = token.value;
                token = input.read(); return result;
        case '(': // E → ( S )
                token = input.read();
                int result = parse_S();
                if (token != ')') throw new ParseError();
                token = input.read(); return result;
        default: throw new ParseError();
   }
}
```

```
int parse_S() {
        switch (token) {
                case num:
                case '(':
                    int left = parse_E();
                    int right = parse_S'();
                    if (right == 0) return left;
                    else return left + right;
                default: throw new ParseError();
        }
}
```

```
int parse_S'() {
    switch (token) {
        case '+':
            token = input.read();
            return parse_S();
        case ')': return 0;
        case EOF: return 0;
        default:
            throw new ParseError();
    }
}
```

Tổng kết

- Với lớp văn phạm LL(1), có thể dùng phương pháp phân tích đệ quy xuống
  - Xây dựng bảng phân tích từ các kí hiệu triệt tiêu đươc, các tập FIRST và FOLLOW
  - Chuyển bảng phân tích sang lập trình phân tích đệ quy xuống
  - Thêm mã lệnh để xây dựng cây cú pháp
- Cách tiếp cận có hệ thống, tránh lỗi và phát hiện văn phạm nhập nhằng
- Bài tới: Chuyển văn phạm sang dạng LL(1), phân tích dưới lên (bottom-up parsing)

Nhập môn Chương trình dịch

Học kì II 2006 – 2007

Bài 6: Phân tích dưới lên

(bottom-up parsing)

Nhớ lại

- Văn phạm LL(1) và bảng phân tích
- Phân tích đệ quy xuống
- Xây dựng cây cú pháp trong khi phân tích đệ quy xuống

Ví dụ

- Văn phạm của các biểu thức cộng có ngoặc.
  VD: (1+2+(3+4))+5
- Văn phạm đệ quy phải

  S → E+S | E

  E → số | (S)
- Văn phạm đệ quy trái

  S → S+E | E

  E → số | (S)
- Văn phạm LL(1)

  S → ES’

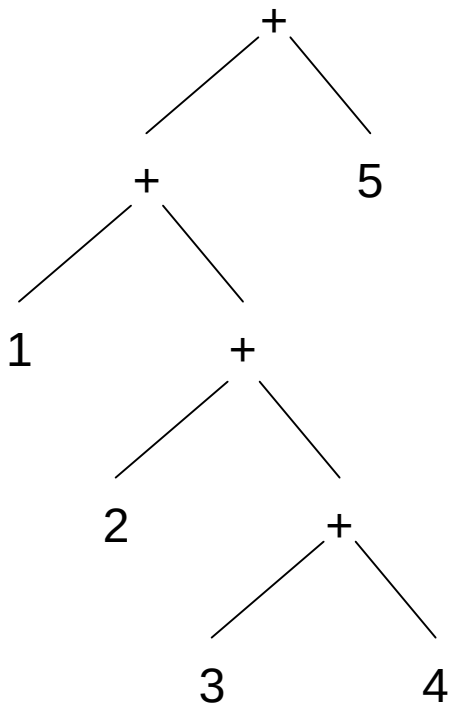
  S’ → +S | ε

  E → số | (S)

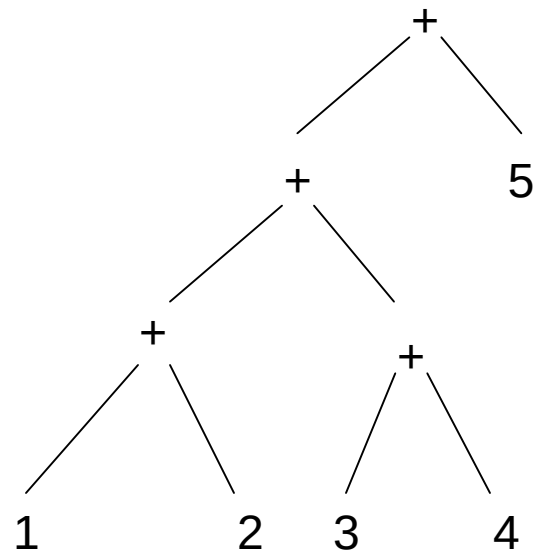
Đệ quy trái và đệ quy phải (1)

(1+2+(3+4))+5

Đệ quy phải – kết hợp bên phải



Đệ quy trái – kết hợp bên trái



Đệ quy trái và đệ quy phải (2)

- Văn phạm đệ quy trái không thể dùng để phân tích từ trên xuống: lặp vô hạn

  S $\Rightarrow$ **S** + E $\Rightarrow$ **S** + E + E $\Rightarrow$ S + E + E + E

- Làm thế nào để phân tích hiệu quả?

Xây dựng văn phạm LL(1)

- Chuyển văn phạm về dạng đệ quy phải

  S → E+S | E

  E → số | (S)

- Dùng kĩ thuật “left – factoring”, sử dụng thêm kí hiệu không kết thúc

  S → ES’

  S’ → +S | ε

  E → số | (S)

Cú pháp BNF mở rộng - EBNF

- Cho phép sử dụng một số cú pháp của biểu thức chính quy: *, +, ?

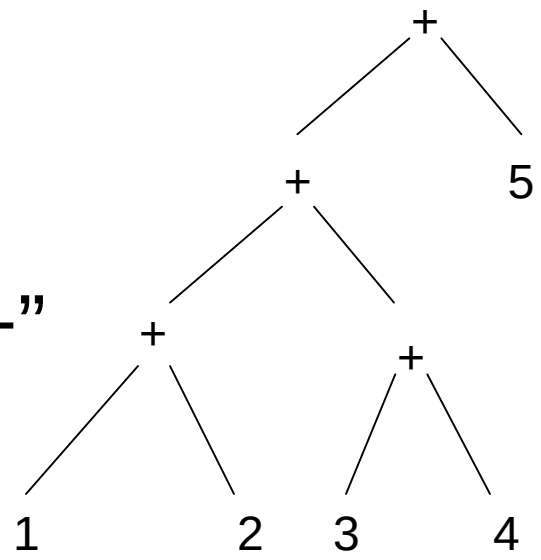
S → ES’

S’ → +S | ε

S → E(+E)*

- EBNF: không còn chỉ rõ thứ tự kết hợp của phép “+”

(1+2+(3+4))+5



Phân tích đệ quy xuống - EBNF

- Chuyển từ EBNF sang phân tích đệ quy xuống

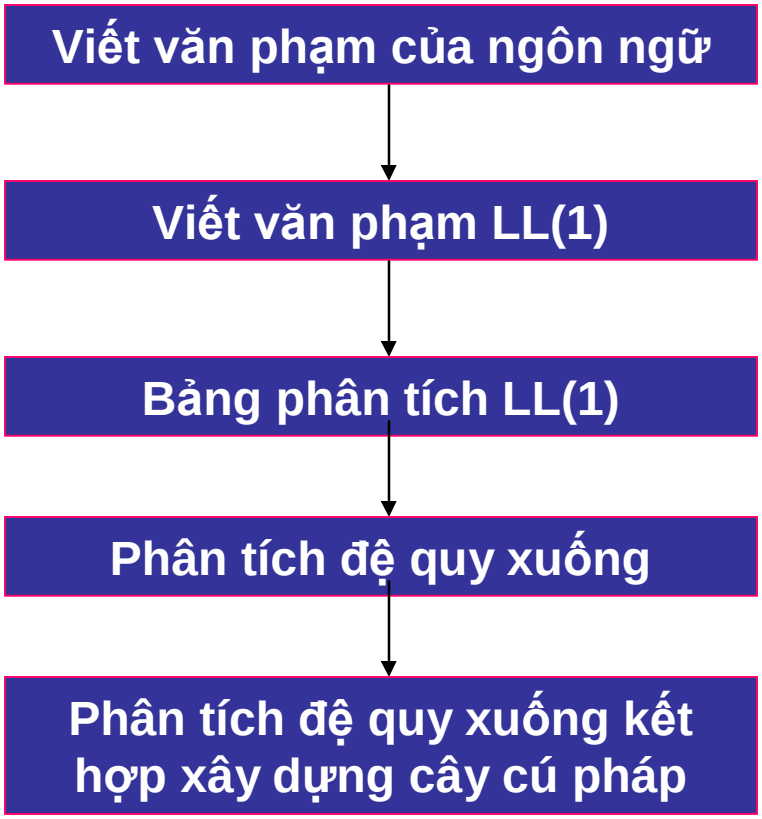
S → E(+E)*

```
void parse_S () { // phân tích dãy E + E + ...
  parse_E ();
  while (true) {
      switch (token) {
      case ‘+’:
            token = input.read(); parse_E();
            break;
      case ‘)’: case EOF: return;
      default: throw new ParseError();
  }
}
```

Kết hợp sinh cây cú pháp - EBNF

```
Expr parse_S() {
   Expr result = parse_E();
   while (true) {
       switch (token) {
             case ‘+’:
               token = input.read();
               result = new Add(result, parse_E());
               break;
       case ‘)’: case EOF: return result;
       default: throw new ParseError();
   }
}
```

Xây dựng bộ PTCP trên xuống

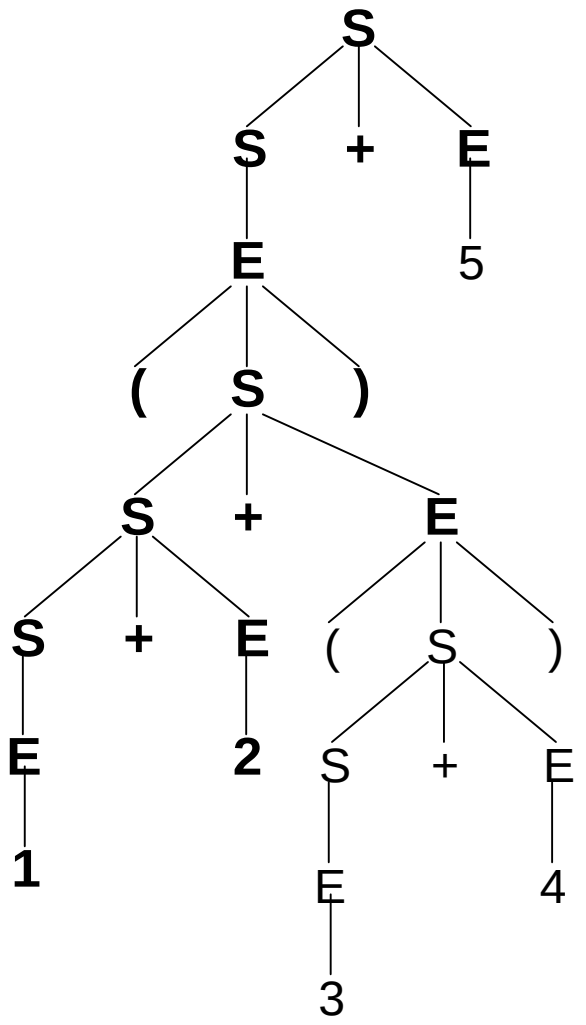


Phân tích từ dưới lên (bottom-up parsing)

- Kỹ thuật phân tích mạnh hơn
- Văn phạm lớp LR có khả năng mô tả mạnh hơn văn phạm lớp LL, có thể mô tả văn phạm đệ quy trái (có trong hầu hết các ngôn ngữ lập trình)
- Dễ dàng mô tả các ngôn ngữ lập trình thông thường
- Bộ phân tích cú pháp ***gạt – thu gọn***
  - Xây dựng cây suy dẫn phải
  - Tự động xây dựng bộ phân tích cú pháp
    VD: yacc, CUP
  - Phát hiện lỗi ngay khi xuất hiện
  - Cho phép phục hồi khi lỗi xảy ra

Phân tích trên xuống

- Suy dẫn trái
- Toàn bộ cây phía trên một kí hiệu được sinh ra
- Phải có khả năng đoán trước được sản xuất



Phân tích dưới lên (1)

S → S+E | E
E → số | (S)

- Suy dẫn phải
- Cây suy dẫn được xây dựng ngược lại
  - Bắt đầu từ kí hiệu kết thúc
  - Kết thúc tại kí hiệu bắt đầu
- Ví dụ

(1+2+(3+4))+5 ⇐ (**E**+2+(3+4))+5 ⇐
(**S**+2+(3+4))+5 ⇐ (S+**E**+(3+4))+5 ⇐
(**S**+(3+4))+5 ⇐ (S+(**E**+4))+5 ⇐ (S+(**S**+4))+5 ⇐
(S+(S+**E**))+5 ⇐ (S+(**S**))+5 ⇐ (S+**E**)+5 ⇐
(**S**)+5 ⇐ **E**+5 ⇐ **S**+5 ⇐ S+**E** ⇐ **S**

Phân tích dưới lên (2)

Suy dẫn phải

(1+2+(3+4))+5 ⇐	**(1+2+(3+4))+5**
(**E**+2+(3+4))+5 ⇐	(1 **+2+(3+4))+5**
(**S**+2+(3+4))+5 ⇐	(1 **+2+(3+4))+5**
(S+**E**+(3+4))+5 ⇐	(1+2 **+(3+4))+5**
(**S**+(3+4))+5 ⇐	(1+2+(3 **+4))+5**
(S+(**E**+4))+5 ⇐	(1+2+(3 **+4))+5**
(S+(**S**+4))+5 ⇐	(1+2+(3 **+4))+5**
(S+(S+**E**))+5 ⇐	(1+2+(3+4 **))+5**
(S+(**S**))+5 ⇐	(1+2+(3+4 **))+5**
(S+**E**)+5 ⇐	(1+2+(3+4) **)+5**
(**S**)+5 ⇐	(1+2+(3+4) **)+5**
E+5 ⇐	(1+2+(3+4)) **+5**
S+**E** ⇐	(1+2+(3+4))+5
S	(1+2+(3+4))+5

Phân tích dưới lên (3)

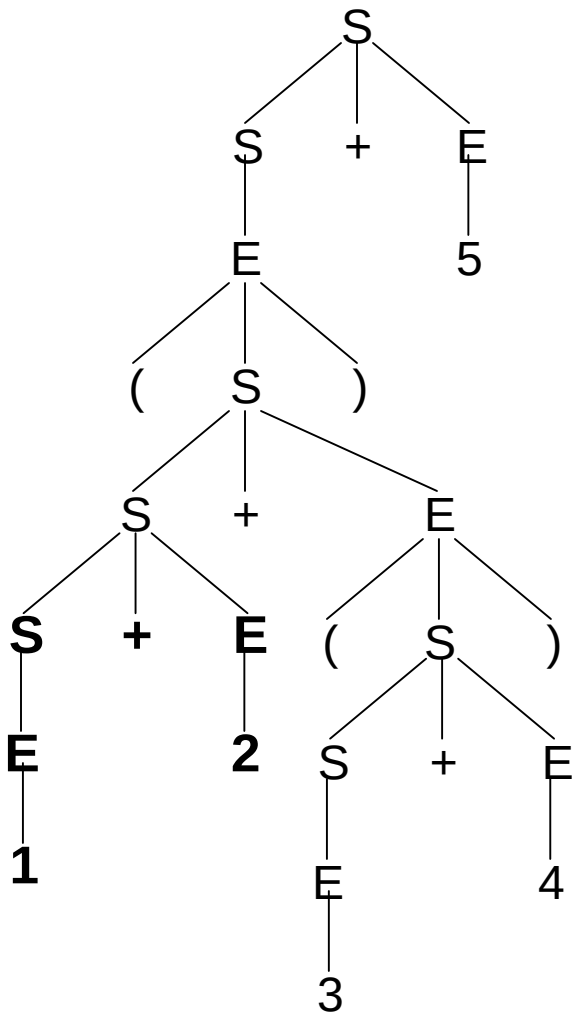
(1+2+(3+4))+5

⇐ (***E***+2+(3+4))+5

⇐ (***S***+2+(3+4))+5

⇐ (S+***E***+(3+4))+5 …

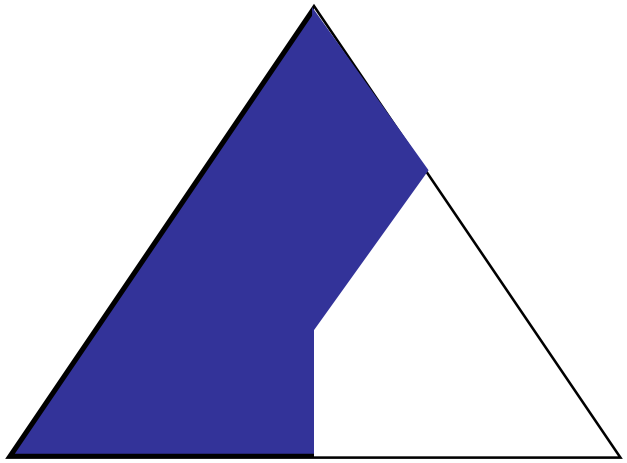
- Phân tích dưới lên có nhiều thông tin hơn khi phân tích



Phân tích dưới lên và phân tích trên xuống

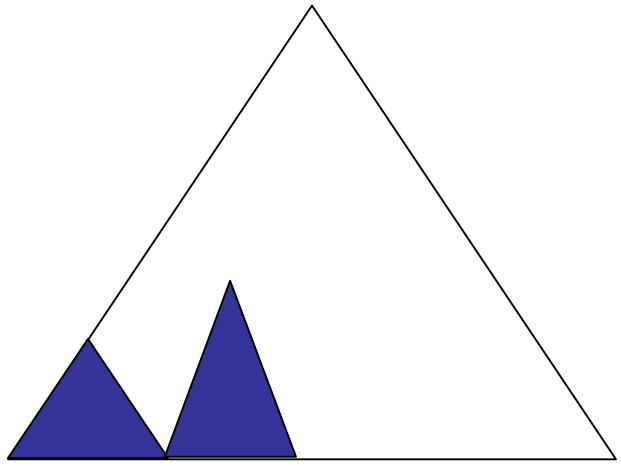
- Phân tích dưới lên không cần sinh ra toàn bộ cây suy dẫn trong quá trình phân tích

Phân tích trên xuống



Đã đọc **Chưa đọc**

Phân tích dưới lên



Đã đọc **Chưa đọc**

Phân tích gạt – thu gọn (1)

- Phân tích bằng một dãy thao tác: ***gạt*** và ***thu gọn***
- Mỗi thời điểm, trạng thái của bộ phân tích là ngăn xếp các kí hiệu kết thúc và không kết thúc
- Cấu hình tại mỗi thời điểm gồm:

ngăn xếp + xâu các kí hiệu chưa đọc

Suy dẫn	Ngăn xếp	Chưa đọc
(1+2+(3+4))+5 ⇐		**(1+2+(3+4))+5**
(E+2+(3+4))+5 ⇐	(E	**+2+(3+4))+5**
(S+2+(3+4))+5 ⇐	(S	**+2+(3+4))+5**
(S+E+(3+4))+5 ⇐	(S+E	**+(3+4))+5**

Phân tích gạt – thu gọn (2)

- **Gạt**: Đọc và đưa một kí hiệu kết thúc của xâu vào stack

Ngăn xếp	Chưa đọc	Thao tác
((1	**1+2+(3+4))+5** **+2+(3+4))+5**	***Gạt* 1**

- **Thu gọn**: Thay thế một xâu γ ở đỉnh của ngăn xếp bằng kí hiệu không kết thúc X với X → γ (pop γ, push X)

Ngăn xếp	Chưa đọc	Thao tác
(S+E (S	**+(3+4))+5** **+(3+4))+5**	***Thu gọn:* S → S+E**

Phân tích gạt – thu gọn (3)

Suy dẫn	Ngăn xếp	Chưa đọc	Thao tác
(1+2+(3+4))+5 ⇐		(1+2+(3+4))+5	*gạt* (
(1+2+(3+4))+5 ⇐	(	1+2+(3+4))+5	*gạt* 1
(1+2+(3+4))+5 ⇐	(1	+2+(3+4))+5	*thu gọn* E→1
(E+2+(3+4))+5 ⇐	(E	+2+(3+4))+5	*thu gọn* S→E
(S+2+(3+4))+5 ⇐	(S	+2+(3+4))+5	*gạt* +
(S+2+(3+4))+5 ⇐	(S+	2+(3+4))+5	*gạt* 2
(S+2+(3+4))+5 ⇐	(S+2	+(3+4))+5	*thu gọn* E→2
(S+E+(3+4))+5 ⇐	(S+E	+(3+4))+5	*thu gọn* S→S+E
(S+(3+4))+5 ⇐	(S	+(3+4))+5	*gạt* +
(S+(3+4))+5 ⇐	(S+	(3+4))+5	*gạt* (
(S+(3+4))+5 ⇐	(S+(	3+4))+5	*gạt* 3
(S+(3+4))+5 ⇐	(S+(3	+4))+5	*thu gọn* E→3
(S+(E+4))+5 ⇐	(S+(E	+4))+5	*thu gọn* S→E
(S+(S+4))+5 ⇐	(S+(S	+4))+5	gạt +
(S+(S+4))+5 ⇐	(S+(S+	4))+5	*gạt* 4
...	...	...	...

Các vấn đề nảy sinh

- Cần xác định khi nào gạt hoặc thu gọn hoặc thu gọn với sản xuất nào?
- Thu gọn sản xuất rỗng

$$X \rightarrow \varepsilon$$

- Có nhiều cách thu gọn

$S \rightarrow E$ hay $S \rightarrow S+E$

Lựa chọn thao tác

- Tại mỗi thời điểm, từ cấu hình

  <S – ngăn xếp, a – từ tố nhìn trước>

- Xác định
  - ***Gạt*** a, ngăn xếp trở thành <Sa>
  - ***Thu gọn*** $X \rightarrow \gamma$, nếu $S = \alpha\gamma$, ngăn xếp trở thành $<\alpha X>$
- Nếu $S = \alpha\gamma$, cần lựa chọn ***gạt*** a hoặc ***thu gọn*** $X \rightarrow \gamma$ dựa vào tiền tố α
  - Với mỗi khả năng thu gọn $X \rightarrow \gamma$ có một α
  - Cần tìm cách đánh dấu các khả năng thu gọn

Trạng thái của bộ phân tích gạt – thu gọn

- Mục tiêu: Xác định khả năng thu gọn hợp lệ tại từng thời điểm
- Ý tưởng: gộp các khả năng có thể có của tiền tố α thành trạng thái của bộ phân tích
- Các vấn đề nảy sinh:
  - Tính toán các trạng thái của bộ phân tích
  - Tính toán các trạng thái kết thúc
  - Phân tích tất định (loại văn phạm nào)
  - Kích cỡ của bộ phân tích (số lượng trạng thái)

Nhập môn Chương trình dịch

Học kì II 2006-2007

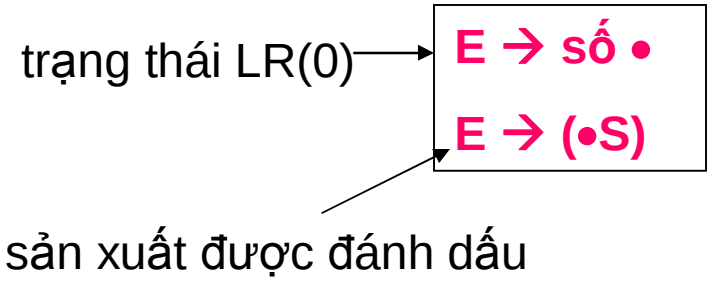
Bài 7: Phân tích LR

Bộ phân tích LR(0)

- Left-to-right scanning
- Right-most derivation
- “zero” lookahead token
- Chưa đủ mạnh để nhận dạng các ngôn ngữ lập trình
- Dùng để hiểu các loại phân tích LR

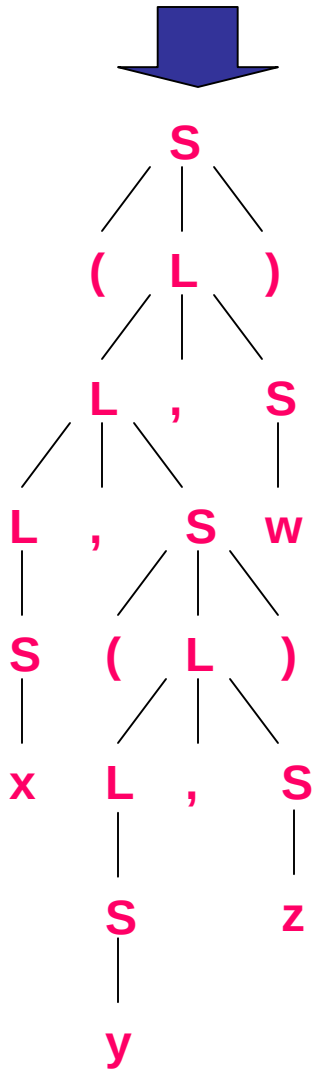
Các trạng thái LR(0)

- Mỗi trạng thái thể hiện và đánh dấu các khả năng có thể xảy ra
- Mỗi trạng thái LR(0) là một tập hợp các sản xuất được đánh dấu bằng các dấu chấm
- Trước dấu • là các kí hiệu đã nằm trên ngăn xếp
- Sau dấu • là các kí hiệu có thể xuất hiện (các từ tố chưa đọc)



Ví dụ: Danh sách

(x, (y,z), w)



- Văn phạm của danh sách

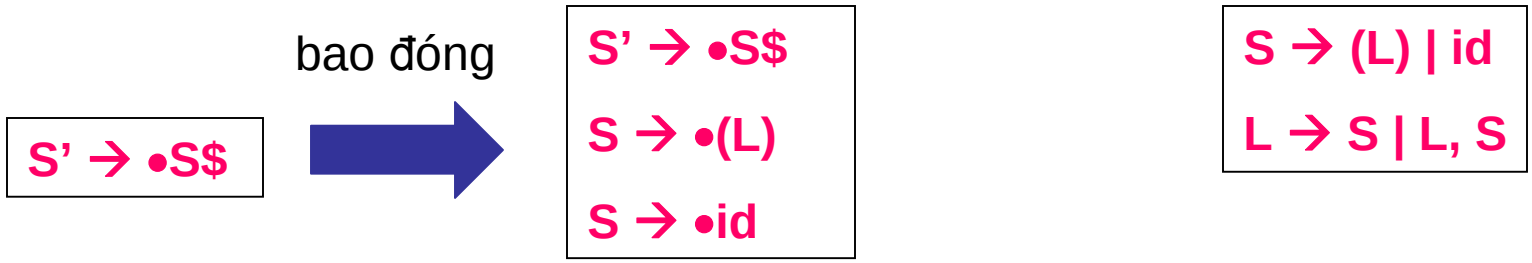
S → (L) | id

L → S | L, S

x (x,(y,z), w)

(((x))) ((x,(y, z)), w)

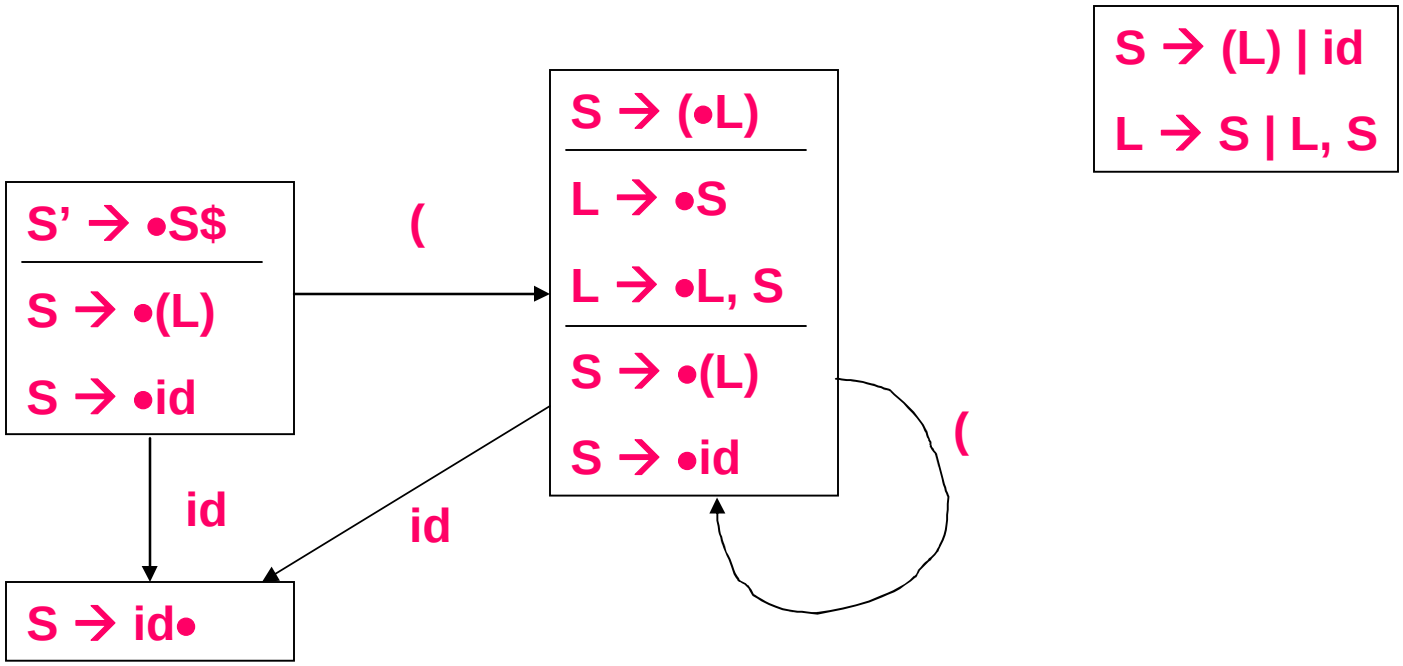
Trạng thái LR(0) và bao đóng



- Thêm sản xuất: S' → S$
- Trạng thái ban đầu của ngăn xếp

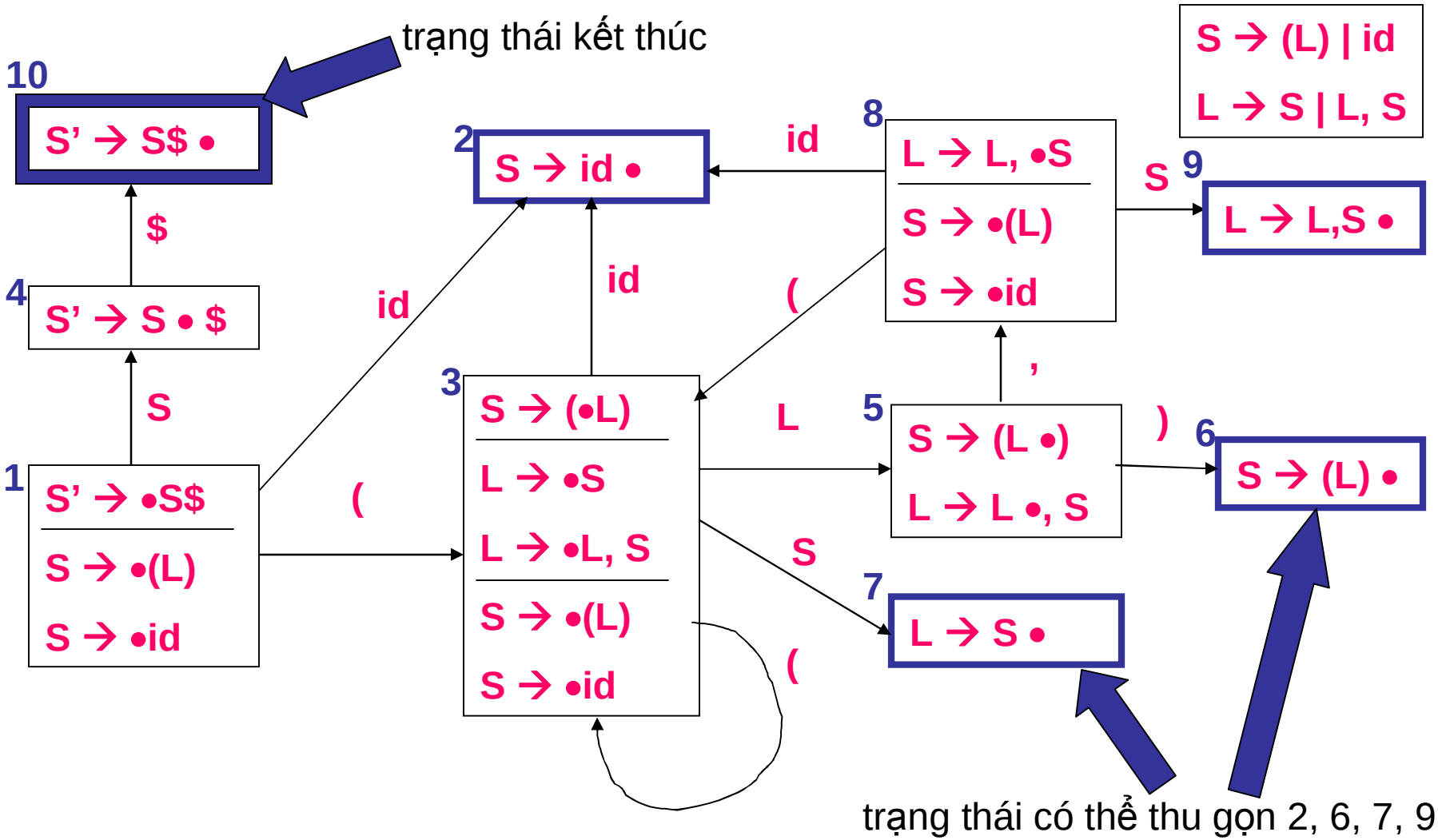
  S' → •S$
- Bao đóng của trạng thái:
  - Với mỗi sản xuất dạng A → α•Bβ, thêm vào trạng thái tất cả các sản xuất dạng B → •γ
  - Ý nghĩa:
    - A → α•Bβ: B chưa xuất hiện (hay có thể xuất hiện ở xâu vào)
    - B → •γ: Chưa kí hiệu nào do B suy dẫn ra xuất hiện (hay γ có thể xuất hiện ở xâu vào)

Chuyển trạng thái – DFA (1)



- Sử dụng các kí hiệu kết thúc và không kết thúc để chuyển trạng thái
- Trạng thái mới bao gồm các sản xuất và bao đóng của chúng

Chuyển trạng thái – DFA (2)



Cài đặt DFA qua PDA (1)

- PDA: push-down automata – ôtômát đẩy xuống
- Khởi tạo ngăn xếp: push trạng thái 1 (bắt đầu)
- Tại mỗi thời điểm, ngăn xếp có dạng

$$(s_1\ x_1\ s_2\ x_2 \ldots s_{n-r}\ x_{n-r} \ldots s_{n-1}\ x_{n-1}\ s_n)$$

- Trong đó:
  - s_i: là trạng thái của DFA
  - x_i: là kí hiệu kết thúc hoặc không kết thúc (kí hiệu trước dấu •)
  - $s_{i+1} = \delta(s_i, x_i)$ – chuyển từ trạng thái s_i sang s_{i+1} nhờ kí hiệu vào x_i

Cài đặt DFA qua PDA (2)

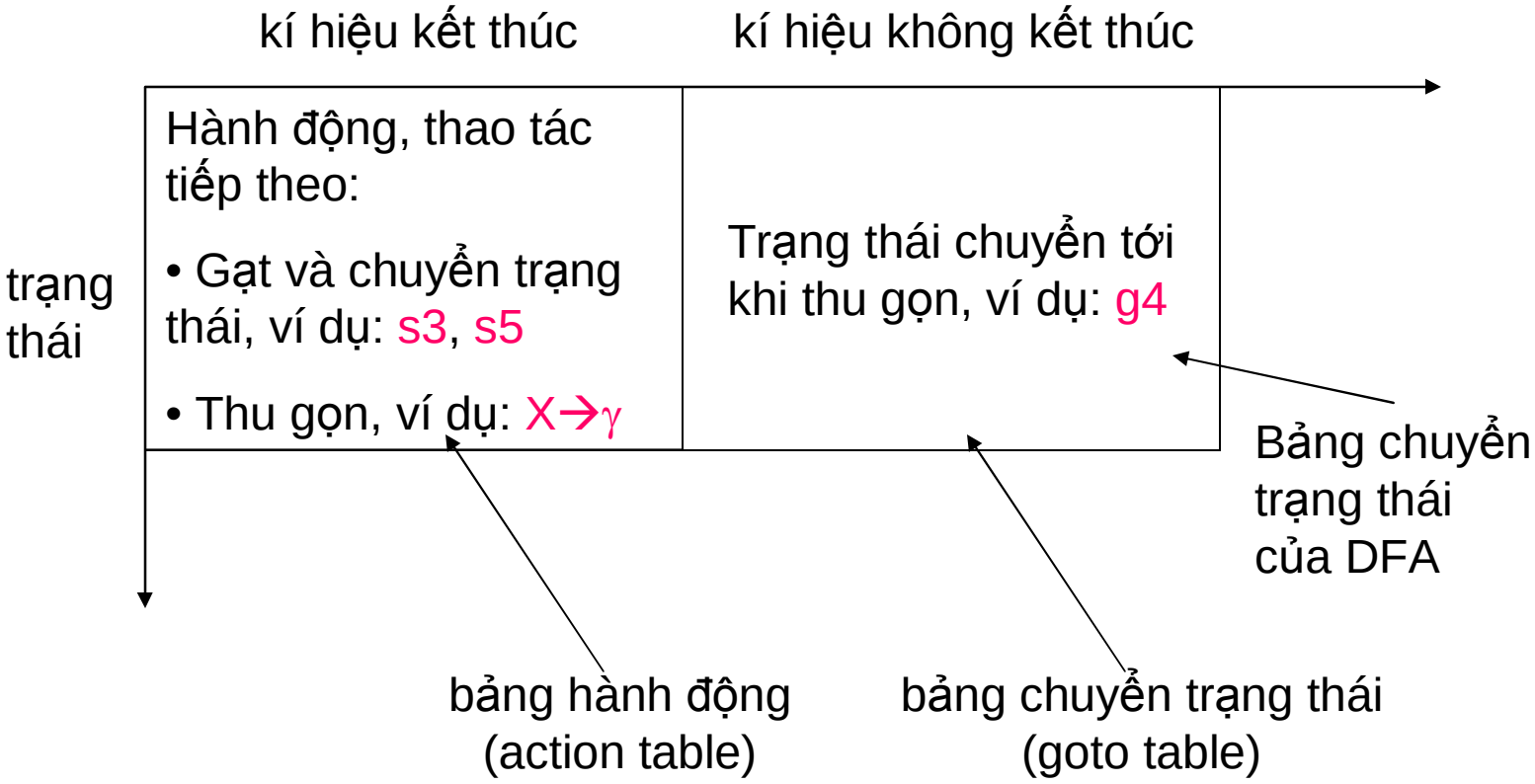
$(s_1\ x_1\ s_2\ x_2 \ldots s_{n-r}\ x_{n-r} \ldots s_{n-1}\ x_{n-1}\ s_n)$

- Hoạt động: với a là kí hiệu vào
  - Gạt a, chuyển tới trạng thái s với $s = \delta(s_n, a)$ = gạt s

    $(s_1\ x_1\ s_2\ x_2 \ldots s_{n-r}\ x_{n-r} \ldots s_{n-1}\ x_{n-1}\ s_n\ a\ s)$
  - Thu gọn $X \rightarrow \gamma$, chuyển tới trạng thái s với
    - $\gamma = x_{n-r} \ldots x_{n-2}\ x_{n-1}$
    - $s = \delta(s_{n-r}, X)$ = goto (s_{n-r}, X)

      $(s_1\ x_1\ s_2\ x_2 \ldots s_{n-r}\ X\ s)$
- DFA + stack = PDA

Cài đặt DFA qua PDA (3)



- Bảng phân tích LR(k)

Ví dụ: bảng LR(0) cho văn phạm danh sách

	(	)	id	,	$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					10		
5		s6		s8			
6	S → (L)	S → (L)	S → (L)	S → (L)	S → (L)		
7	L → S	L → S	L → S	L → S	L → S		
8	s3		s2			g9	
9	L → L, S	L → L, S	L → L, S	L → L, S	L → L, S		

Ví dụ: sử dụng bảng LR(0) – ((x),y)

S → (L) | id

L → S | L, S

suy dẫn phải	ngăn xếp	xâu vào	hành động
((x),y)	1	((x),y)$	*gạt*, chuyển sang 3
((x),y)	1(3	(x),y)$	*gạt*, chuyển sang 3
((x),y)	1(3(3	x),y)$	*gạt*, chuyển sang 2
((x),y)	1(3(3x2	),y)$	*thu gọn* S→id, sang 7
((**S**),y)	1(3(3S7	),y)$	*thu gọn* L→S, sang 5
((**L**),y)	1(3(3L5	),y)$	*gạt*, chuyển sang 6
((L),y)	1(3(3L5)6	,y)$	*thu gọn* S→(L), sang 7
(**S**,y)	1(3S7	,y)$	*thu gọn* L→S, sang 5
(**L**,y)	1(3L5	,y)$	*gạt*, chuyển sang 8
(L,y)	1(3L5,8	y)$	*gạt*, chuyển sang 2
(L,y)	1(3L5,8y2	)$	*thu gọn* S→id, sang 9
(L,**S**)	1(3L5,8S9	)$	*thu gọn* L→L,S, sang 5
(**L**)	1(3L5	)$	*gạt*, chuyển sang 6
(L)	1(3L5)6	$	*thu gọn* S→(L), sang 4
S	4	$	sang 10 – chấp nhận

Bảng phân tích LR(0)

- Thu gọn khi gặp trạng thái có thể thu gọn
- Không cần nhìn trước từ tố tiếp theo khi thu gọn
- Khi có nhiều sản xuất có thể thu gọn ở cùng một trạng thái → xung đột
- Ví dụ: xung đột gạt/thu gọn, thu gọn/thu gọn

không xung đột

L → L, S•

xung đột gạt/thu gọn

E → E • + S
S → E •

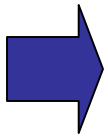
xung đột thu gọn/thu gọn

S → E •
E → (S) •

Ví dụ: Văn phạm biểu thức cộng

- Đệ quy trái

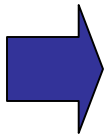
S → S+E | E
E → số | (S)



LR(0) ?

- Đệ quy phải

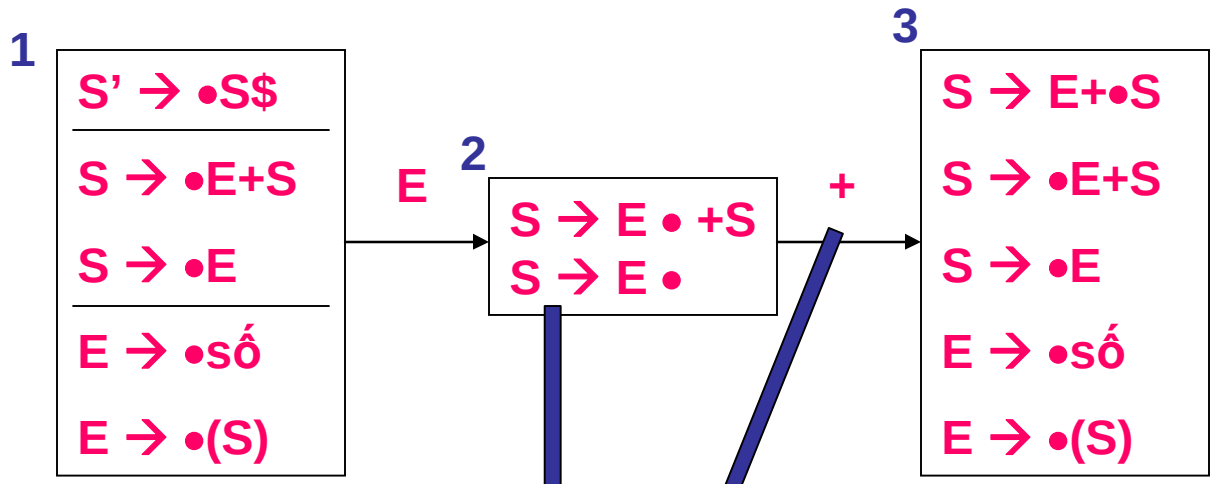
S → E+S | E
E → số | (S)



LR(0) ?

Ví dụ: Văn phạm biểu thức cộng

S → E+S | E
E → số | (S)



	+	$	E
1			g2
2	s3/S→E	S → E	

Bảng phân tích SLR

- Ý tưởng: chỉ thu gọn khi từ tố nhìn trước nằm trong FOLLOW của kí hiệu không kết thúc (vế trái của sản xuất)
- Một số thu gọn trong bảng LR(0) không còn trong bảng SLR
- FOLLOW(S) = {), $ }

	+	$	E
1			g2
2	s3	S → E	

Phân tích LR(1)

- Tận dụng hết khả năng phân tích khi nhìn trước 1 từ tố
- LR(1) = phân tích gạt – thu gọn với 1 từ tố nhìn trước
- Trạng thái LR(1) = trạng thái LR(0) + từ tố có thể theo sau về phải sản xuất

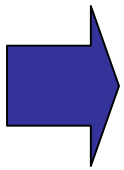
LR(0) | **S → • S+E** |

LR(1) | **S → • S+E +** |

Trạng thái LR(1)

- Các sản xuất giống nhau và có cùng vị trí của dấu chấm (•) được gộp lại với nhau

S → S • + E	+
S → S • + E	$
S → S + • E	số



S → S • + E	+,$
S → S + • E	số

- Bao đóng LR(1): Với sản xuất **A → α•Bβ a**
  - Thêm vào các sản xuất **B → •γ b**
  - b ∈ FIRST(βa) → b ∈ FOLLOW(B)

LR(1)

SLR

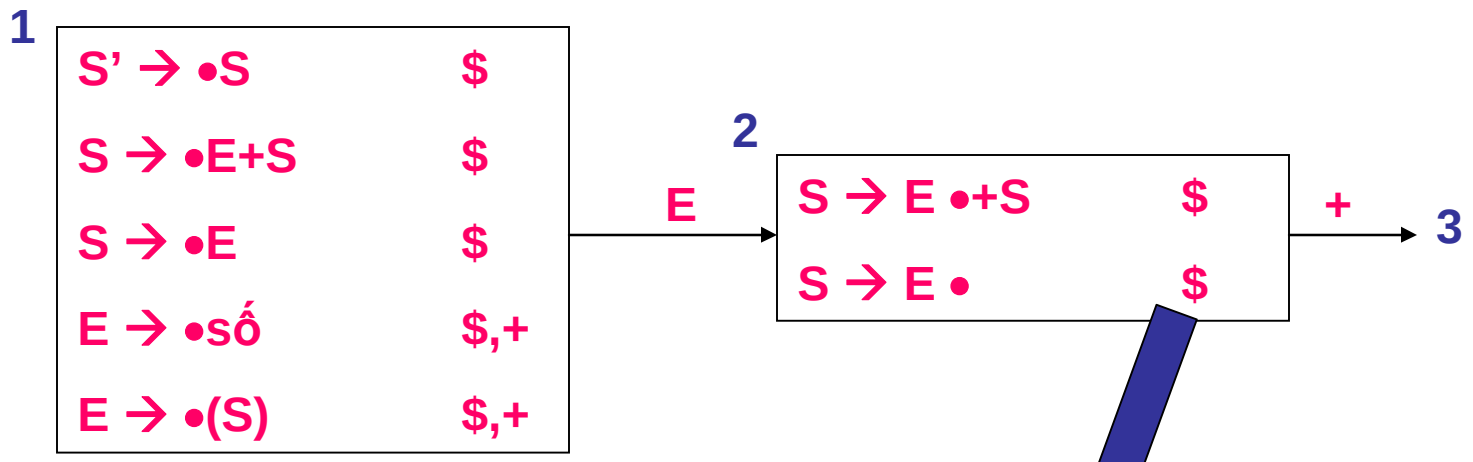
Ví dụ: Trạng thái LR(1)

Bao đóng LR(1): Với sản xuất $A \rightarrow \alpha \bullet B\beta$ **a**

Thêm vào các sản xuất $B \rightarrow \bullet\gamma$ **b**

b $\in$ **FIRST(**β**a)**

S → E+S | E
E → số | (S)

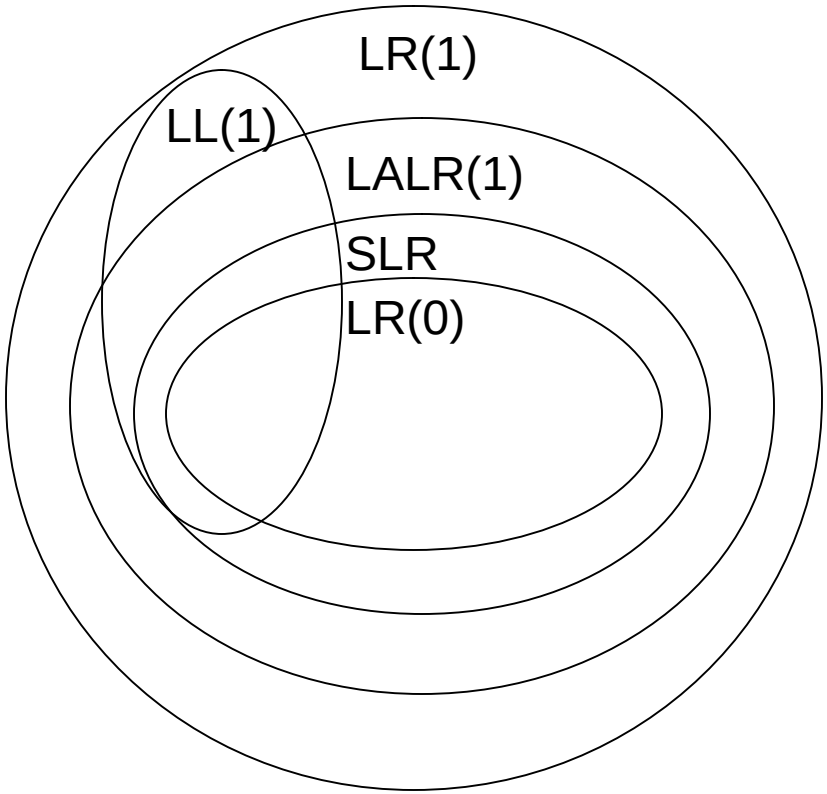


	+	$	E
1			g2
2	s3	S → E	

Phân tích LALR(1)

- LR(1): Có quá nhiều trạng thái
- SLR ≈ 100 trạng thái → LR(1) ≈ 1000 trạng thái
- Kết hợp các trạng thái LR(1) có các sản xuất và vị trí của dấu chấm (•) giống nhau lại thành 1 trạng thái
- LALR(1) = SLR ?
- Có số trạng thái bằng SLR
- Là kỹ thuật phân tích thường dùng trong thực tế

Phân lớp văn phạm



Nhập môn Chương trình dịch

Học kì II 2006-2007

Bài 8: Phân tích LR (tiếp)

Lập trình bộ phân tích cú pháp

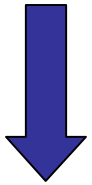
- Trước đây: tự viết bộ phân tích cú pháp
- Hiện nay: sử dụng các trình sinh bộ phân tích cú pháp. VD: yacc, cup, bison
- Ưu điểm:
  - Sử dụng phương pháp phân tích LALR(1)
  - Cho phép khai báo thứ tự ưu tiên, kết hợp của các phép toán
  - Tự động sinh code phân tích cú pháp (kể cả bảng phân tích LALR(1))

Thứ tự kết hợp (1)

- Ví dụ

S → S + E | E

E → số | (S)



E → E + E | (E) | số

- Nếu sử dụng phương pháp LALR(1), ta sẽ được bộ phân tích như thê nào ?

Thứ tự kết hợp (2)

- Xung đột

$$E \rightarrow E + E \mid (E) \mid số$$

$E \rightarrow E\bullet + E$	+
$E \rightarrow E + E\bullet$	+,$

xung đột gạt / thu gọn

1 + 2 + 3

gạt: 1 + (2 + 3)

thu gọn: (1 + 2) + 3

Khai báo cú pháp trong CUP

non terminal E; terminal PLUS, LPAREN...

precedence left PLUS;

Khi gặp dấu cộng, **thu gọn** nếu vế phải có dấu cộng, **gạt** nếu vế phải không có dấu cộng

E ::= E PLUS E

| LPAREN E RPAREN

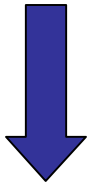
| NUMBER ;

Thứ tự ưu tiên (1)

- Ví dụ

E → E + E | T

T → T x T | số |(E)

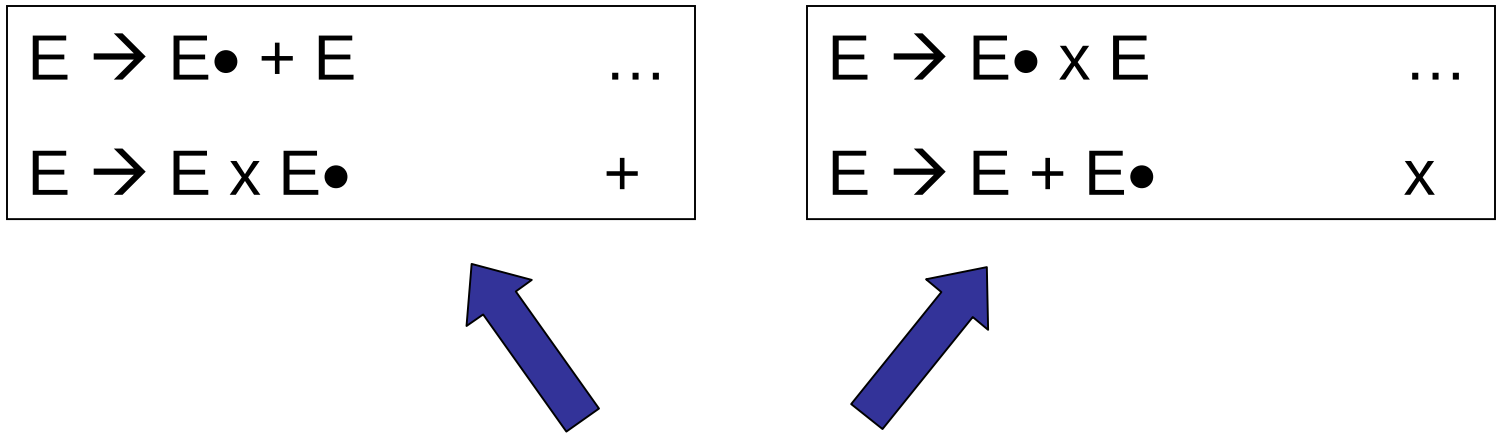


E → E + E | E x E | (E) | số

Thứ tự ưu tiên (2)

- Xung đột

$$E \rightarrow E + E \mid E \times E \mid (E) \mid \text{số}$$

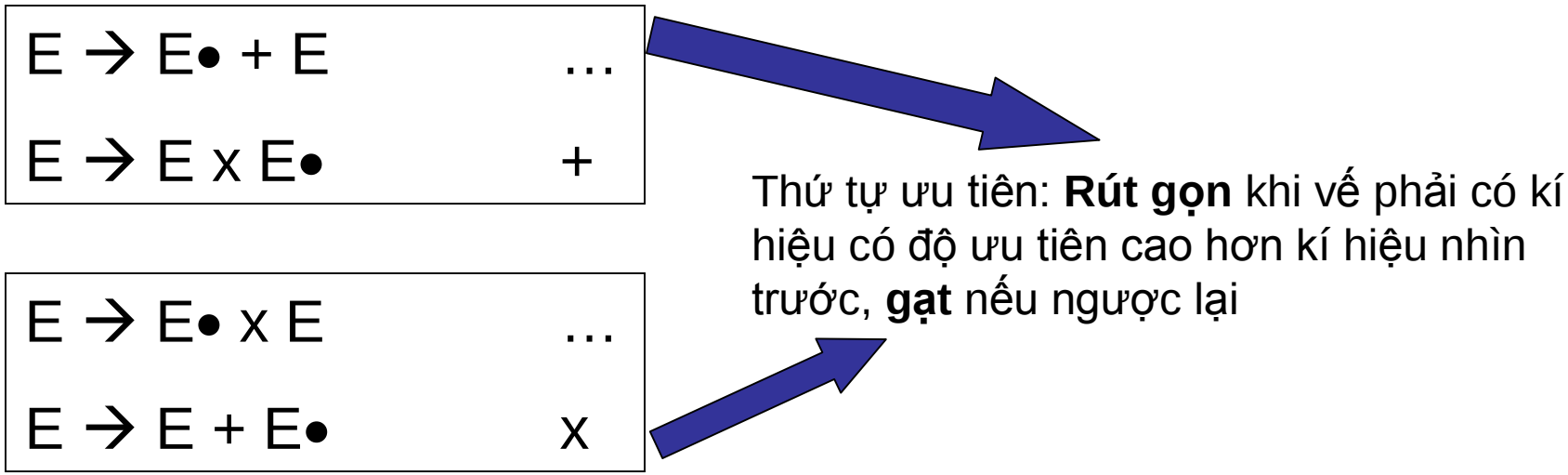


Khai báo cú pháp trong CUP

precedence left PLUS;

precedence left TIMES; // TIMES > PLUS

E ::= E PLUS E | E TIMES E | ...



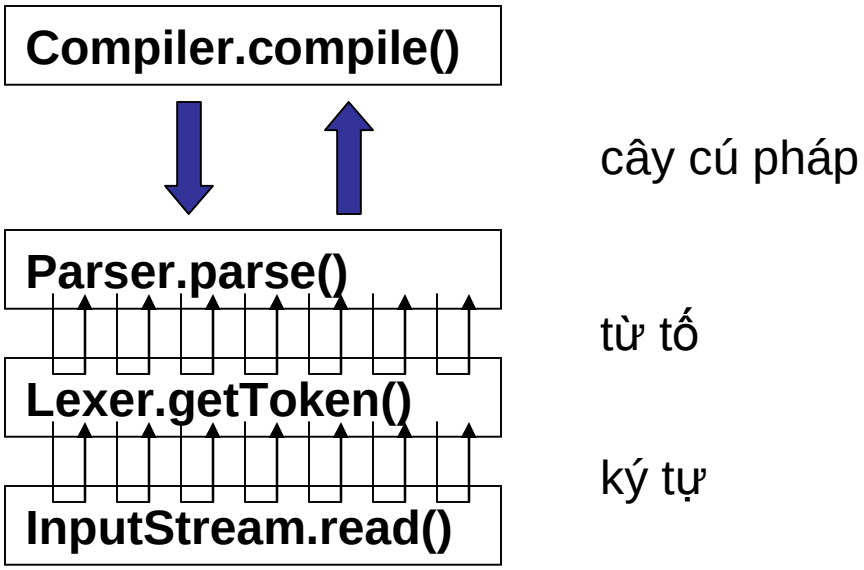
Tổng kết

- Các chương trinh sinh bộ phân tích cú pháp sử dụng phương pháp LALR(1)
- Thứ tự ưu tiên và kết hợp cho phép viết cú pháp ngôn ngữ dễ dàng hơn
- Văn phạm ngôn ngữ gần với cách viết thông thường

Chương trình chính của một chương trình dịch (1)

```
class Compiler {
  void compile() throws CompileError {
     Lexer l = new Lexer(input);
     Parser p = new Parser(l);
     AST tree = p.parse();
     // gọi l.getToken() để lấy dãy từ tố
     if (typeCheck(tree))
        IR = genIntermediateCode(tree);
     IR.emitCode();
  }
}
```

Chương trình chính của một chương trình dịch (2)



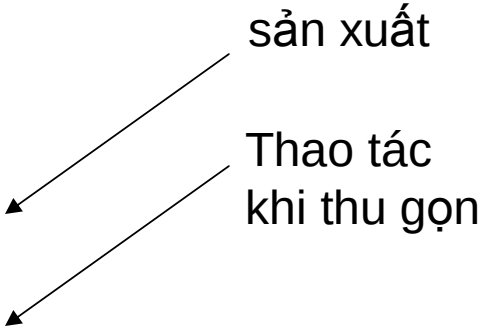
Cây cú pháp

- Là kết quả của bộ phân tích cú pháp
- Là một dạng thể hiện của chương trình nguồn, cho phép
  - phân tích ngữ nghĩa (kiểm tra kiểu)
  - tối ưu một phần chương trình nguồn
  - sinh mã trung gian
- Dịch = duyệt cây cú pháp
- Biểu diễn cây cú pháp bằng ngôn ngữ hướng đối tượng

Xây dựng cây cú pháp

- Các thao tác xây dựng cây cú pháp được gắn vào văn phạm
- Ví dụ: CUP

```
non terminal Expr expr; ...
expr ::= expr:e1 PLUS expr:e2
{: RESULT = new Add(e1,e2); :}
```

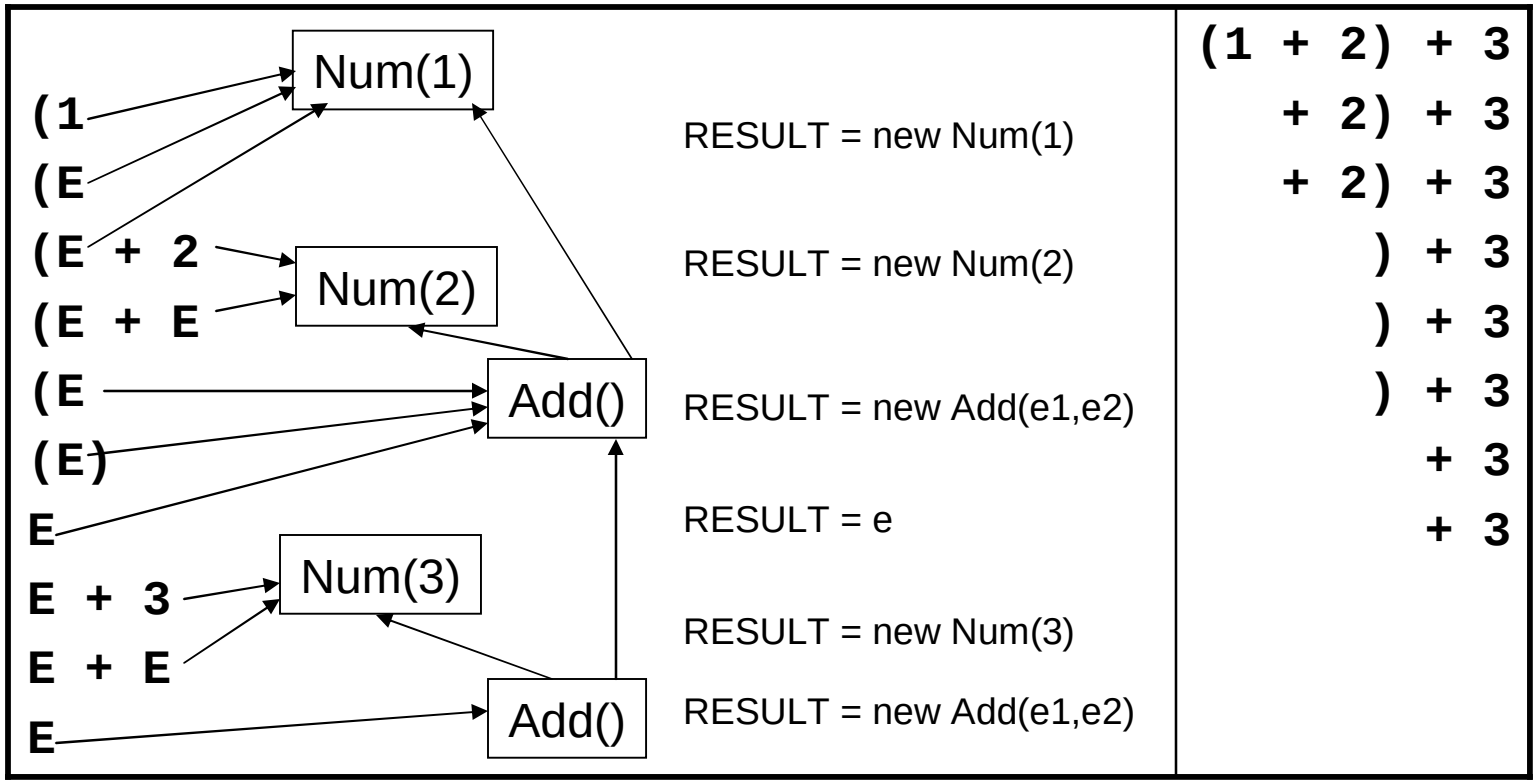


- Thao tác được thực hiện khi **thu gọn** một sản xuất
- Biến RESULT đại diện cho vế trái của sản xuất
- Cây cú pháp được xây dựng từ dưới lên

Ví dụ

```
non terminal Expr expr; ...
expr ::= expr:e1 PLUS expr:e2
{: RESULT = new Add(e1,e2); :}
```

- Ngăn xếp của bộ phân tích cú pháp lưu giá trị RESULT của vế trái



Hướng đối tượng (1)

- Viết lớp trừu tượng (abstract class) cho các kí hiệu không kết thúc
- Với mỗi sản xuất viết một lớp dẫn xuất

E → E + E | E x E | (E) | số

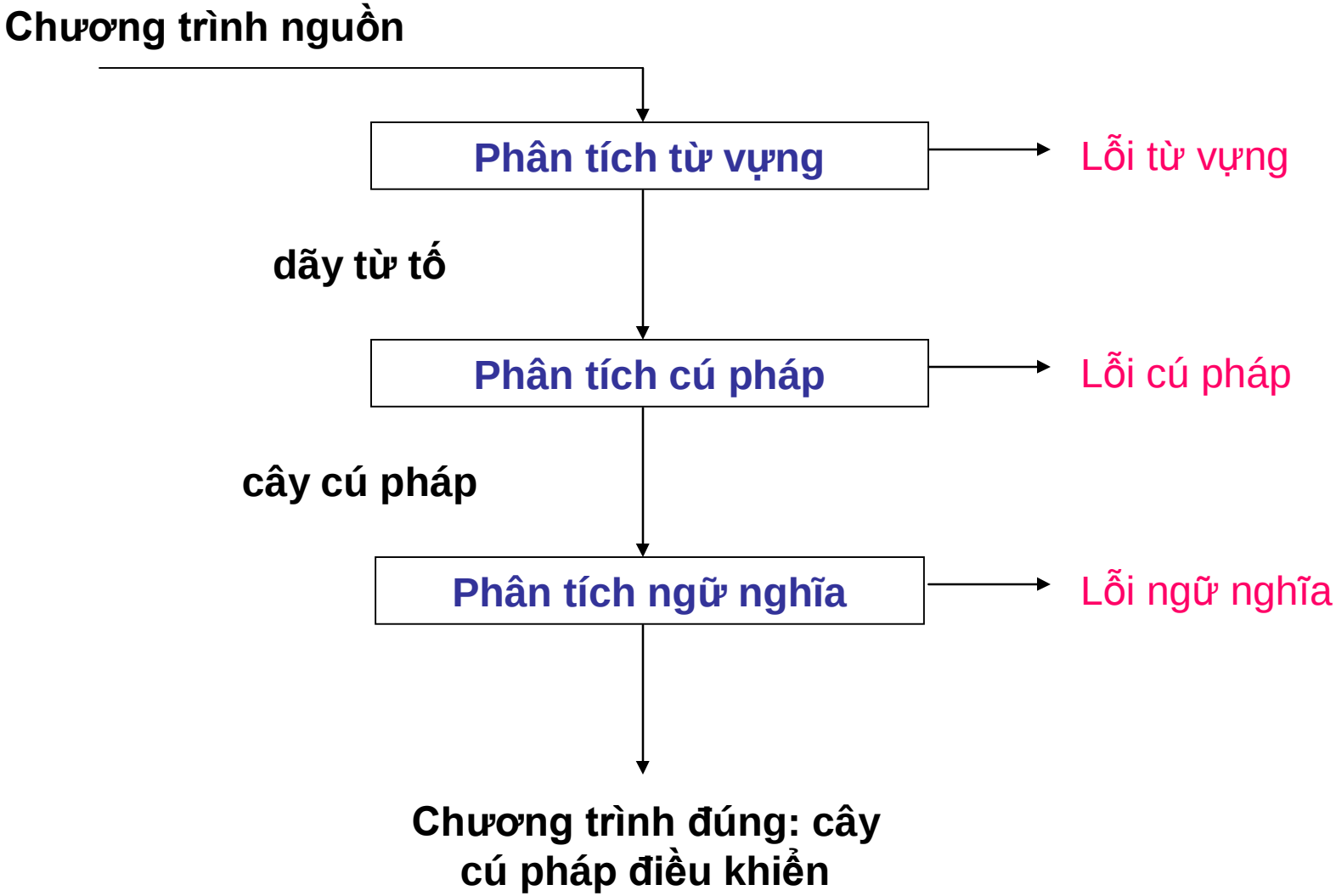
```
abstract class Expr { … } // E
class Add extends Expr { Expr left, right; … }
class Mult extends Expr { Expr left, right; … }
// class BinExpr extends Expr { Oper o; Expr l, r; }
class Negate extends Expr { Expr e; …}
// class UnaryExpr extends Expr { Oper o; Expr e; }
```

Hướng đối tượng (2)

```
non terminal Expr expr; ...
expr ::= expr:e1 PLUS expr:e2
{: RESULT = new BinaryExpr(plus, e1, e2); :}
| expr:e1 TIMES expr:e2
{: RESULT = new BinaryExpr(times, e1, e2); :}
| MINUS expr:e
{: RESULT = new UnaryExpr(negate, e); :}
| LPAREN expr:e RPAREN
{: RESULT = e; :}
```

- plus, times, negate: Oper

Phân tích ngữ nghĩa



Nhập môn Chương trình dịch

Học kì II 2006-2007

Bài 09: Phân tích ngữ nghĩa

Nhắc việc

- Nộp bài tập lập trình số 1: **tuần sau**

Phân tích ngữ nghĩa

- Tìm tất cả các lỗi còn lại của chương trình nguồn
  - Khai báo biến
  - Kiểm tra kiểu (kiểu tĩnh)
- Thiết lập các thông tin cần thiết cho các bước dịch sau đó
  - Kiểu của các biểu thức
  - Bố trí dữ liệu

Kiểm tra ngữ nghĩa – đệ quy

- Ta đã có cây cú pháp
  - Duyệt cây cú pháp bằng phương pháp đệ quy, tới thăm tất cả các nút trong cây
  - Tại mỗi nút, xây dựng các thông tin cần thiết (thông tin về kiểu)

```
class Add extends Expr {
  Expr e1, e2;
  Type typeCheck() throws SemanticError {
    Type t1 = e1.typeCheck(), t2 = e2.typeCheck();
    if (t1 == Int && t2 == Int) return Int;
    else throw new TypeCheckError(“type error +”);
  }
}
```

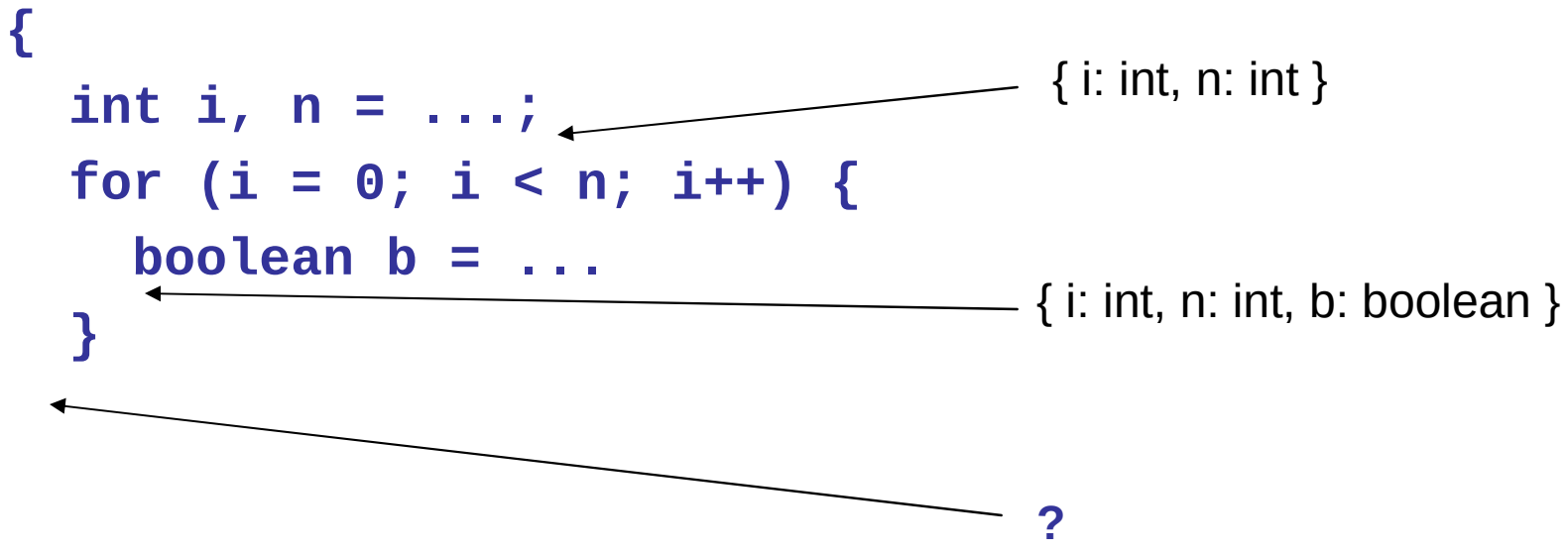
Kiểm tra kiểu của tên

```
class Id extends Expr {
  String name;
  Type typeCheck() {
    return ?
  }
}
```

- Cần lưu giữ thông tin về kiểu của các tên xuất hiện trong phạm vi (scope) của chương trình: ***bảng kí hiệu*** (symbol table)

Bảng kí hiệu

- Là một tập hợp các cặp tên và kiểu của chúng
- VD: { x: int, y: array[string] }



Cài đặt bảng kí hiệu (1)

- Bảng kí hiệu cho phép kiểm tra kiểu của các tên trong chương trình

```
class SymTab {
  Type lookup(String id) ...
  void add(String id, Type binding) ...
}
```

Sử dụng bảng kí hiệu

- Bảng kí hiệu là tham số của tất cả các thủ tục kiểm tra kiểu (typeCheck()) tại các nút

```
class Id extends Expr {
  String name;
  Type typeCheck(SymTab s) {
    try {
      return s.lookup(name);
    }
    catch (NotFound exc) {
      throw new UndefinedIdentifier(this);
    }
  }
}
```

Quản lý phạm vi biến (scope)

```
class Add extends Expr {
  Expr e1, e2;
  Type typeCheck(SymTab s) {
    Type t1 = e1.typeCheck(s),
    t2 = e2.typeCheck(s);
    if (t1 == Int && t2 == Int) return Int;
    else throw new TypeCheckError(“+”);
  }
}
```

- Các tên xuất hiện trong cùng phạm vi phải được lưu ở cùng một bảng kí hiệu
- Vậy khi nào ta thêm một tên vào bảng kí hiệu?

Quản lý phạm vi biến (2)

- Java, C++: các lệnh khai báo biến
- Giả sử dãy các câu lệnh {*stmt*1; *stmt*2; *stmt*3...} được biểu diễn bằng các nút trong cây cú pháp:

```
abstract class Stmt { … }
class Block { Vector/*Stmt*/ stmts; … }
```

- Mỗi khai báo biến là một lệnh:

```
class Decl extends Stmt {
String id; TypeExpr typeExpr; ...
```

Quản lý phạm vi biến (3)

```
class Block {
  Vector stmts;
  Type typeCheck(SymTab s) {
    Type t;
    for (int i = 0; i < stmts.length(); i++) {
      t = stmts[i].typeCheck(s);
      if (stmts[i] instanceof Decl) {
        Decl d = (Decl) stmts[i];
        s.add(d.id, d.typeExpr.interpret());
      }
    }
    return t;
  }
}
```

Quản lý phạm vi biến (4)

- Khôi phục lại bảng kí hiệu của phạm vi trước đó


```
{ int x = 5;
  { int y = 1; }
  x = y;
  // câu lệnh sai bởi y không nằm trong cùng phạm vi!
}
```

Quản lý phạm vi biến (4)

```
class Block {
  Vector stmts;
  Type typeCheck(SymTab s) {
    Type t;
    SymTab s1 = s.clone();
    for (int i = 0; i < stmts.length(); i++) {
      t = stmts[i].typeCheck(s1);
      if (stmts[i] instanceof Decl) {
        Decl d = (Decl) stmts[i];
        s1.add(d.id, d.typeExpr.interpret());
      }
    }
    return t;
  }
}
```

Các biến được thêm vào phạm vi (s1) không ảnh hưởng đến các đoạn mã bên ngoài khối (block)

Cài đặt bảng kí hiệu (2)

- Quá trính kiểm tra kiểu cũng là quá trình xây dựng bảng kí hiệu
  - Bảng kí hiệu còn dùng để lưu trữ các khai báo kiểu, các nhãn nhảy (lệnh break, continue), …
  - Ở cấp cao nhất, bảng kí hiệu lưu trữ các biến toàn cục, các khai báo kiểu và khai báo module
  - Ở các cấp thấp hơn, bảng kí hiệu được mở rộng thêm nhờ vào các lệnh khai báo
- Có thể xây dựng lại bảng kí hiệu tại mỗi nút, nhưng để tránh phải tính toán lại, ta nên lưu bảng kí hiệu tại các nút tương ứng

Cài đặt bảng kí hiệu (3)

```
class SymTab {
  SymTab parent;
  HashMap table;
  Object lookup(String id) {
    if (table.get(id) != null) return table.get(id);
    else return parent.lookup(id); // can cache..
  }

  void add(String id, Object t)
  { table.add(id,t); }

  SymTab(Symtab p)
  { parent = p; } // =clone
}
```

Sử dụng kiểu cài đặt của danh sách liên kết

Thông tin điều khiển: kiểu

- Lưu trữ thông tin về kiểu tại các nút (“trang trí” cây cú pháp)

```
abstract class Expr {
  protected Type type = null;
  public Type typeCheck();
}

class Add extends Expr {
  Type typeCheck() {
    Type t1 = e1.typeCheck(), t2 = e2.typeCheck();
    if (t1 == Int && t2 == Int)
      { type = Int; return type; }
    else throw new TypeCheckError(“+”);
  }
}
```

- Có thể lưu trữ bảng kí hiệu tại các nút

Các bước dịch = duyệt cây cú pháp

- Các bước dịch khác cũng hoạt động giống như việc kiểm tra kiểu
  - Thay thế các hằng số
  - Sinh mã trung gian
  - Tối ưu mã
  - Sinh mã máy
- Các bước dịch này đều duyệt cây cú pháp

→ tính đa hình của lập trình hướng đối tượng

- Nhận xét
  - Mã lệnh cho từng bước dịch phải viết cho từng nút trong cây
  - Cách duyệt cây cú pháp giống nhau (đệ quy xuống)

Tổng kết

- Phân tích ngữ nghĩa = duyệt cây cú pháp đệ quy xuống
- Quản lý phạm vi biến bằng bảng kí hiệu
- Có thể tận dụng tính lặp lại của các bước dịch để viết các đoạn mã duyệt cây cú pháp tổng quát (tính đa hình)

Nhập môn Chương trình dịch

Học kì II 2006 – 2007

Bài 10: Biểu thức kiểu

Nội dung

- Kiểm tra kiểu
- Cài đặt các biểu thức kiểu

Kiểm tra kiểu

- Các phép toán đòi hỏi các toán hạng phải phù hợp kiểu
- Các hàm đòi hỏi tham số phù hợp kiểu
- Lệnh return phải trả về đúng kiểu trả về của hàm
- Lệnh gán đòi hỏi kiểu của vế phải phù hợp với kiểu của vế trái
- Lệnh khai báo kiểu: typedef, class

Hệ thống kiểu (1)

- Mỗi ngôn ngữ lập trình có hệ thống kiểu riêng
- Mỗi kiểu là một giới hạn dữ liệu
- VD: int = $[-2^{31}, 2^{31}]$, char = [-128, 127]
- Các kiểu dữ liệu phức hợp được tạo từ các kiểu đơn giản bởi các biểu thức kiểu (type expressions, type contructors)
- VD: int, string, Array[int], Object

Ví dụ: C++

- Kiểu cơ bản: int, char, …
- Kiểu phức hợp:

  int[100], struct {int a, char b}
- Biểu thức kiểu:

  T là kiểu

  T[ ] là kiểu với mọi T

Hệ thống kiểu (2): định nghĩa kiểu

- Một số ngôn ngữ cho phép người lập trình tự định nghĩa kiểu
- VD: C++

  typedef int int_array[ ];

  class cView { … };
- int_array là một kiểu giống với int[ ]
- Có thể có nhiều định nghĩa kiểu của cùng một kiểu

Biểu thức kiểu: Mảng

- Mỗi ngôn ngữ có một cách định nghĩa mảng
- Mảng không giới hạn:
  - C/C++: T[ ]
- Mảng có giới hạn
  - C/C++/Java: T[L] – L phần tử kiểu T
- Mảng có giới hạn trên, dưới
  - Pascal: T[L, U] – đánh chỉ số từ L đến U
- Mảng nhiều chiều
  - C/C++/Java/Pascal

Biểu thức kiểu: Cấu trúc

- Là biểu thức kiểu khá phức tạp
- Biểu thức kiểu có dạng {id1: T1, id2: T2, …} với id và T là tên và kiểu của các trường
- Ví dụ
  - C/C++: struct { int a; float b; } tương ứng với biểu thức kiểu {a: **int**, b: **float**}
- Các kiểu lớp (Class) là mở rộng của kiểu struct (cho phép thành viên là hàm)

Biểu thức kiểu: Hàm

- Hàm cho phép nhận nhiều tham số và trả về giá trị
- Tham số thứ i có kiểu T_i, kiểu trả là T
- Biểu thức kiểu: T_1 x T_2 x … T_n → T
- Ví dụ: int f(int, char) tương ứng với biểu thức kiểu int x char → int
- Trong C++/Java, cần mở rộng biểu thức kiểu của hàm để có thể trả lại ngoại lệ

Cài đặt kiểu (1)

- Cài đặt lớp trừu tượng Type là lớp cơ sở của tất cả các kiểu

```
abstract class Type {
  abstract boolean operator ==(Type t);
}
```

- Mỗi kiểu được đại diện bằng một lớp dẫn xuất

```
class BaseType extends Type { String name; }
static BaseType Int, Char, Float, …
class IotaClass extends Type { … }
class ArrayType extends Type { Type elemType; }
```

Cài đặt kiểu (2)

- int[ ] được đại diện bởi một đối tượng kiểu, bất kể được định nghĩa kiểu như thế nào
  - typedef int int_array[ ];
  - typedef int int_Array[ ];
- Phân tích ngữ nghĩa
  - Xây dựng đối tượng kiểu từ các biểu thức kiểu
  - Gắn các tên với đối tượng kiểu tương ứng
  - Có thể xây dựng kiểu giống như khi xây dựng cây cú pháp (VD: đưa vào định nghĩa văn phạm trong CUP)
- Kiểm tra kiểu: bằng cách cài đặt toán tử == trong lớp Type

Nhập môn Chương trình dịch

Học kì II 2006 - 2007

Bài 11: Luật ngữ nghĩa

Luật ngữ nghĩa

- Bảng kí hiệu → đánh dấu các tên và các kiểu
- Biểu thức kiểu → mô tả cách tạo thành các kiểu
- Làm thế nào để mô tả việc kiểm tra kiểu

→ Dùng luật ngữ nghĩa

- Luật ngữ nghĩa định nghĩa kiểu của các nút trong cây cú pháp

Luật ngữ nghĩa

- Luật ngữ nghĩa cho phép ***đánh giá kiểu*** của các nút trong cây cú pháp
- Ví dụ:
- E : T nút E có kiểu T
- 2 : int nút 2 có kiểu int
- 2 * (3 + 4) : int
- true : bool
- “hello” : string
- if (b) 2 else 3 : int

Đánh giá kiểu

if (b) 2 else 3 : int

- Làm thế nào để đánh giá biểu thức trên có kiểu **int** ?

b phải có kiểu bool (b: bool)

2 phải có kiểu int (2: int)

3 phải có kiểu int (3: int)

Đánh giá kiểu

- Ta viết: $A \vdash E : T$
  - nghĩa là: trong ngữ cảnh A (bảng kí hiệu) thì biểu thức E có kiểu T
  - Ví dụ:

$b : bool, x : int \vdash b : bool$

$b : bool, x : int \vdash \text{if } (b) \text{ } 2 \text{ else } x : int$

$\vdash 2 + 2 : int$

Đánh giá kiểu

- Để đánh giá

  b : bool, x : int ⊢ if (b) 2 else x : int

- Phải đánh giá được

  b : bool, x : int ⊢ b : bool

  b : bool, x : int ⊢ 2 : int

  b : bool, x : int ⊢ x : int

Luật ngữ nghĩa

- Với mọi ngữ cảnh A, biểu thức E, lệnh S_1 và lệnh S_2 ta có luật ngữ nghĩa

  $A \vdash \text{if (E) } S_1 \text{ else } S_2 : T$

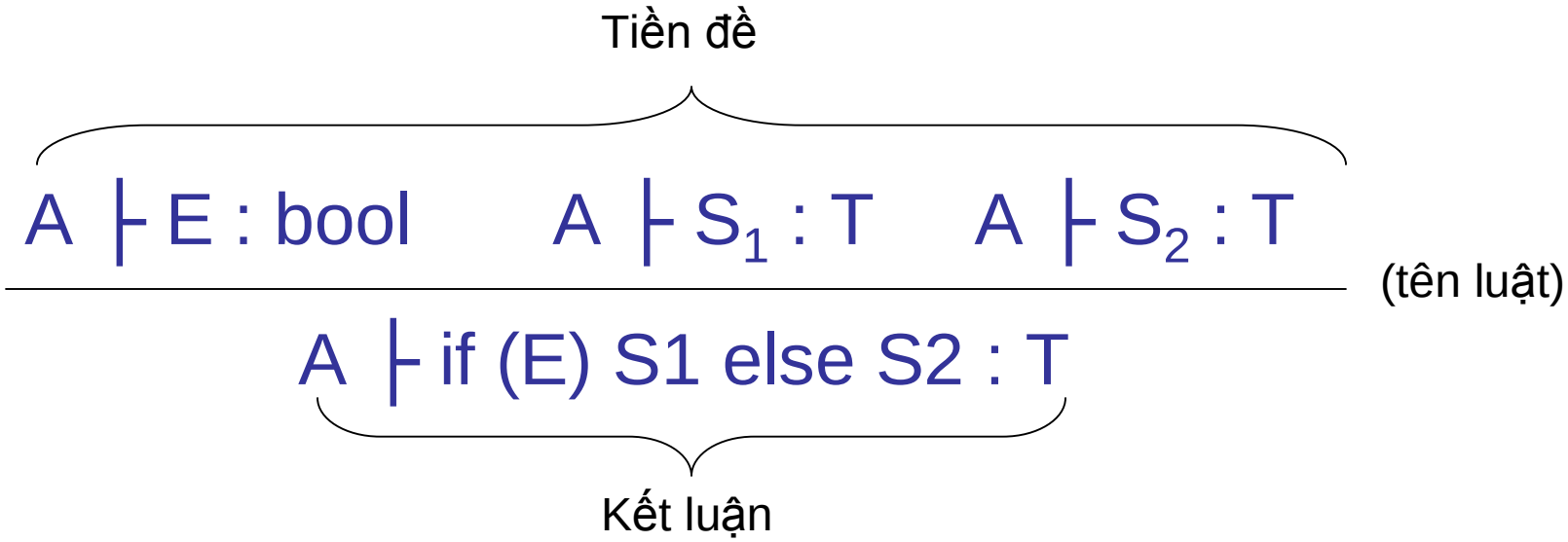
  là đúng nếu

  $A \vdash E : bool$

  $A \vdash S_1 : T$

  $A \vdash S_2 : T$

Viết luật ngữ nghĩa

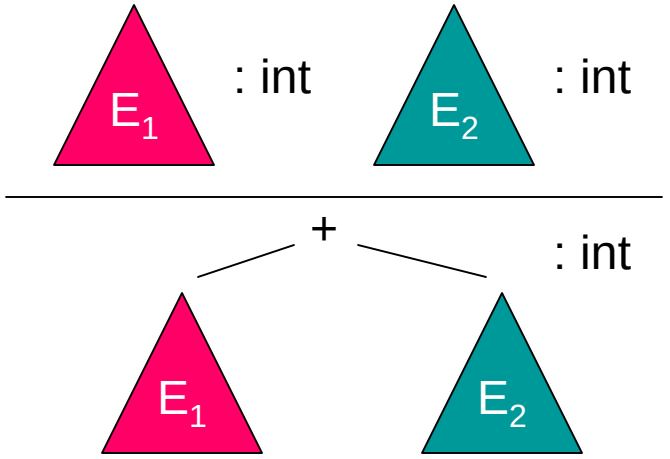


Viết luật ngữ nghĩa

- Cho phép mô tả chính xác, ngắn gọn cách đánh giá kiểu
- Luật ngữ nghĩa được viết cho từng nút của cây hoặc từng sản xuất của văn phạm
- Đánh giá kiểu (kiểm tra kiểu) là quá trình lần ngược cây cú pháp dựa vào các luật

Ví dụ

$$\frac{A \vdash E_1 : \text{int} \qquad A \vdash E_2 : \text{int}}{A \vdash E_1 + E_2 : \text{int}} \; (+)$$



Cài đặt luật ngữ nghĩa

- Cài đặt bằng cách lần ngược lại theo luật

```
class Add extends Expr {
  Expr e1, e2;
  Type typeCheck(SymTab A) {
    Type t1 = e1.typeCheck(A),
    t2 = e2.typeCheck(A);
    if (t1 == Int && t2 == Int) return Int;
    else throw new TypeCheckError("+");
  }
}
```

T = E.typeCheck(A)

$\Leftrightarrow A \vdash E : T$

$$\frac{A \vdash E_1 : \text{int} \qquad A \vdash E_2 : \text{int}}{A \vdash E_1 + E_2 : \text{int}} \; (+)$$

Luật ngữ nghĩa

- Luật ngữ nghĩa đúng với mọi giá trị của các biến trong luật
- Luật ngữ nghĩa không có tiền đề: ***tiên đề***
- Cùng một kết luận có thể có nhiều cách chứng minh (nhiều luật có cùng kết luận)

Luật ngữ nghĩa: lệnh While

- Với các lệnh không có kiểu, ta đưa vào một kiểu giả - **unit** (unit = có kiểu đúng)

$$\frac{A \vdash E : bool \qquad A \vdash S : T}{A \vdash \text{while (E) S} : unit} \text{ (while)}$$

Luật ngữ nghĩa: lệnh If

$$\frac{A \vdash E : \text{bool} \qquad A \vdash S : T}{A \vdash \text{if (E) S} : \text{unit}} \text{(If)}$$

Luật ngữ nghĩa: lệnh gán

$$\frac{A \vdash id : T \qquad A \vdash E : T}{A \vdash id = E : T} \text{ (Assign)}$$

$$\frac{A \vdash E_1 : Array\ [\ T\ ] \qquad A \vdash E_2 : int \qquad A \vdash E_3 : T}{A \vdash E_1[E_2] = E_3 : T} \text{ (Array assign)}$$

Nhập môn Chương trình dịch

Học kì II 2006 – 2007

Bài 12: Luật ngữ nghĩa (tiếp)

Luật ngữ nghĩa: dãy lệnh (block)

- Luật: một dãy lệnh có kiểu đúng nếu lệnh đầu tiên có kiểu đúng và dãy lệnh sau đó cũng có kiểu đúng.

$$\frac{A \vdash S_1 : T_1 \qquad A \vdash (S_2, S_3, \ldots S_n) : T_n}{A \vdash (S_1, S_2, \ldots S_n) : T_n}\ \text{(block)}$$

- Làm thế nào nếu S_1 là lệnh khai báo?

Luật ngữ nghĩa: dãy lệnh (block)

$$\frac{A \vdash T\ id : T_1 \text{ (lệnh khai báo)} \qquad A, id : T \vdash (S_2, S_3, \ldots S_n) : T_n}{A \vdash (T\ id, S_2, \ldots S_n) : T_n}\ \text{(decl. block)}$$

- Luật này mô tả đoạn mã kiểm tra kiểu của dãy lệnh (bài 10)

Cài đặt luật ngữ nghĩa cho dãy lệnh

```
class Block {
  Stmt stmts[];
  Type typeCheck(SymTab s) {
    Type t;
    for (int i = 0; i < stmts.length; i++) {
      t = stmts[i].typeCheck(s);
      if (stmts[i] instanceof Decl) {
        Decl d = (Decl)stmts[i];
        s = s.add(d.id, d.type.interpret());
      }
    }
    return t;
  }
}
```

$$\frac{A \vdash T\ id : T_1 \text{ (lệnh khai báo)} \qquad A, id : T \vdash (S_2, S_3, \ldots S_n) : T_n}{A \vdash (T\ id, S_2, \ldots S_n) : T_n}\ \text{(decl. block)}$$

Luật ngữ nghĩa: lời gọi hàm

- Nếu E là một hàm có kiểu

$$E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$$

- T_i là kiểu của tham số, T_r là kiểu trả về
- Luật ngữ nghĩa cho lời gọi hàm

$$E(E_1, E_2, \ldots E_n)$$

$$\frac{A \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \qquad A \vdash E_i : T_i \ (i = 1, 2, \ldots n)}{A \vdash E(E_1, E_2, \ldots E_n) : T_r} \ \text{(func. call)}$$

Luật ngữ nghĩa: định nghĩa hàm

- C/C++: hàm được viết dưới dạng

```
Tr f(T1 a1, … Tn an)
{ …
  return E;
}
```

- Kiểu của E phải là T_r, nhưng trong ngữ cảnh (bảng kí hiệu) nào?

Luật ngữ nghĩa: định nghĩa hàm

- Giả sử A là ngữ cảnh bao quanh định nghĩa hàm f
- Định nghĩa hàm f có kiểu đúng nếu

$$A, a_1 : T_1, \ldots, a_n : T_n \vdash E : T_r$$

Ví dụ: hàm đệ quy

```
int fact(int x) {
  if (x == 0) return 1;
  else return x * fact(x-1);
}
```

A_1 = {fact: int → int}

A_2 = {fact: int → int, x : int}

$$\frac{A_2 \vdash x : \text{int} \quad A_2 \vdash 0 : \text{int}}{A_2 \vdash x == 0 : \text{bool}}$$

$$A_2 \vdash 1 : \text{int}$$

$$\frac{A_2 \vdash x : \text{int} \quad \dfrac{A_2 \vdash \text{fact} : \text{int} \to \text{int} \quad \dfrac{A_2 \vdash x : \text{int} \quad A_2 \vdash 1 : \text{int}}{A_2 \vdash x - 1 : \text{int}}}{A_2 \vdash \text{fact}(x-1) : \text{int}}}{A_2 \vdash x*\text{fact}(x-1) : \text{int}}$$

Luật ngữ nghĩa: lệnh return

$$\frac{A \vdash E : T}{A \vdash \text{return } E : \text{unit}}$$

- Kiểm tra kiểu của lệnh return: E phải có kiểu là kiểu trả về của hàm (tức là $T = T_r$)
- Lệnh return có kiểu unit (có kiểu đúng) nếu $T = T_r$

Luật ngữ nghĩa: lệnh return

- Thêm một dòng {return: T_r} vào bảng kí hiệu
- Kiểm tra kiểu của return (T_r) có giống kiểu của E không?
- Tóm lại, ta có luật ngữ nghĩa của định nghĩa hàm và lệnh return:

$$A, a_1 : T_1, \ldots, a_n : T_n, \text{return } T_r \vdash E : T_r$$

$$\frac{A_2 \vdash E : T \;,\; A \vdash \text{return} : T}{A_2 \vdash \text{return } E : \text{unit}}$$

Xây dựng bộ luật ngữ nghĩa

- Các luật ngữ nghĩa khác đều viết tương tự như các luật đã học
- Bộ luật ngữ nghĩa cho phép đánh giá một chương trình có kiểu đúng hay không
- Cách viết: theo kiểu quy nạp
  - Viết các luật tiên đề
  - Với mỗi nút (sản xuất) trong cây cú pháp viết luật ngữ nghĩa cho nút đó từ các luật nhỏ hơn
- Như vậy, bộ luật ngữ nghĩa cho phép kiểm tra kiểu của một chương trình viết đúng cú pháp và việc kiểm tra luôn luôn dừng

Nhập môn Chương trình dịch

Học kì II 2006-2007

Bài 13: Sinh mã trung gian

Mô tả các bước dịch (1)

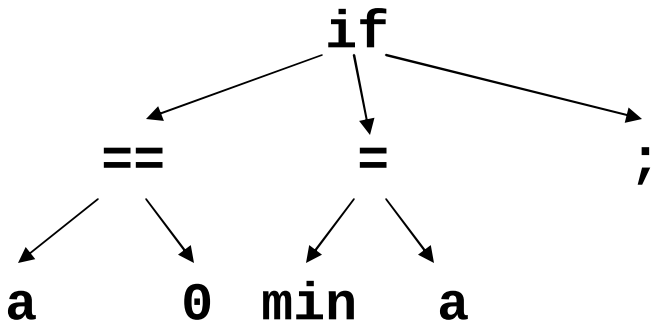
Mã nguồn (dãy các kí tự)

```
If (a == 0) min = a;
```

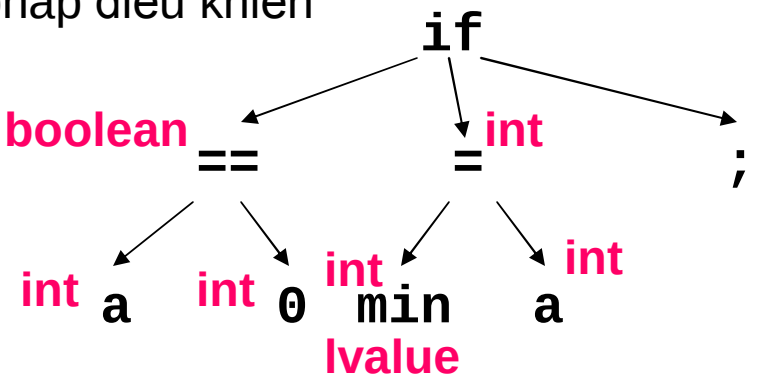
Dãy các từ tố (token)

If	(	Id:a	==	0	)	Id:min	=	Id:a	;

Cây cú pháp



Cây cú pháp điều khiển

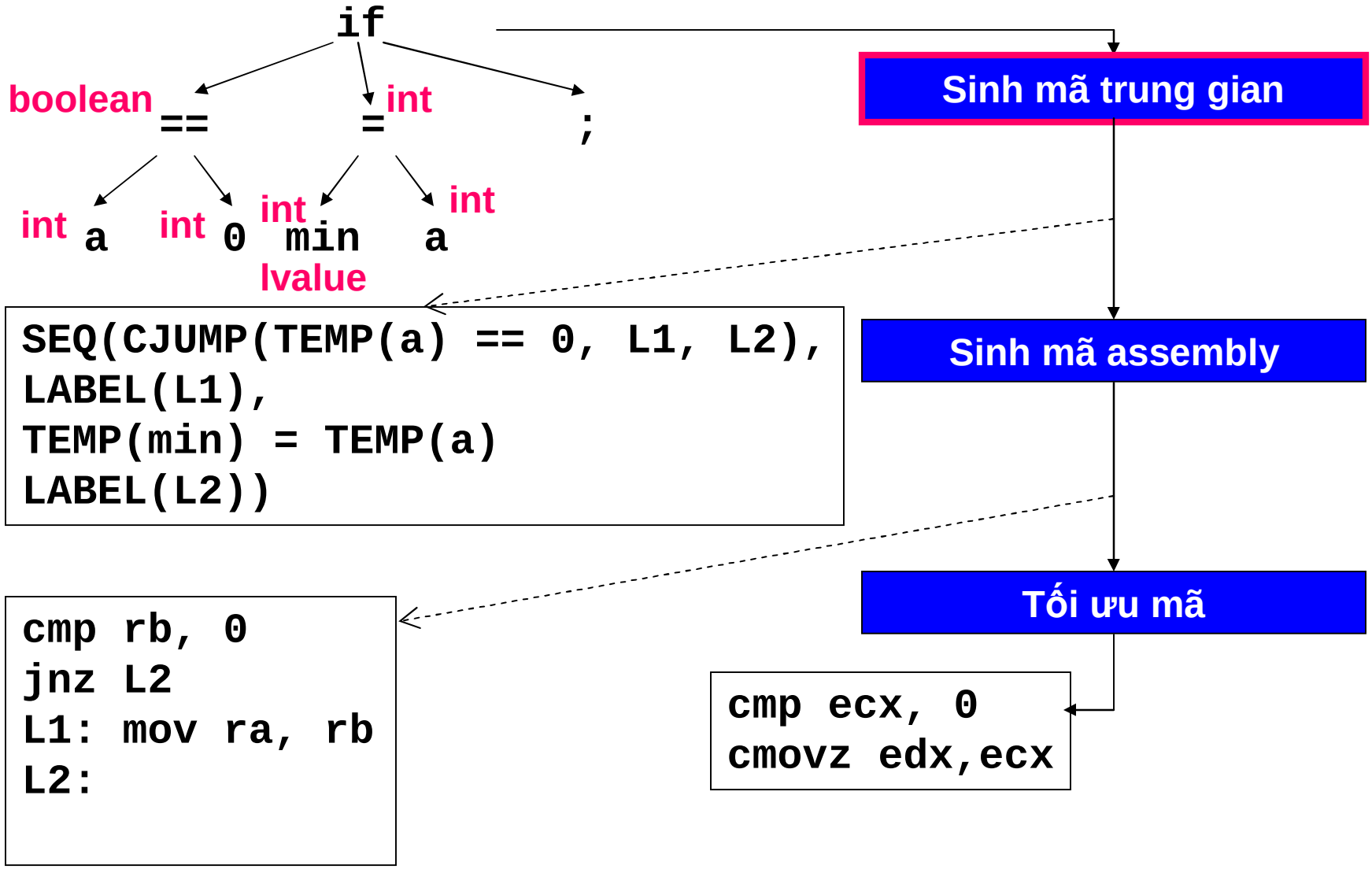


Phân tích từ vựng

Phân tích cú pháp

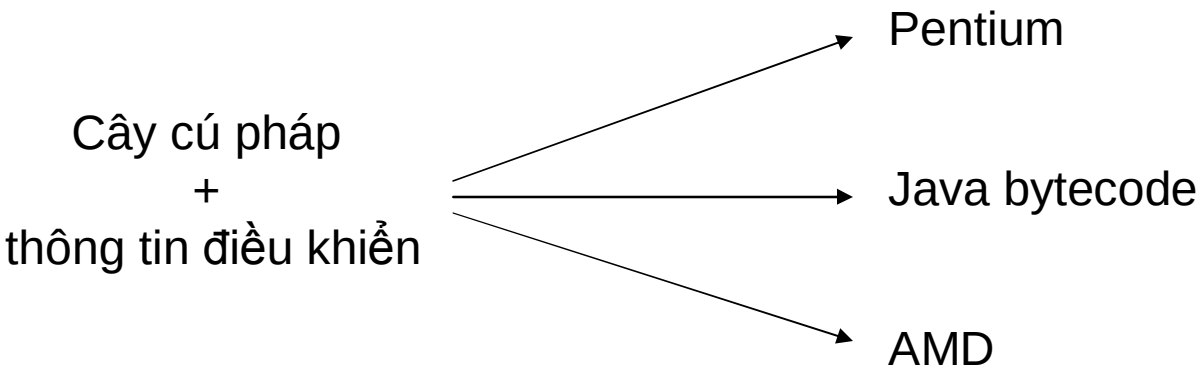
Phân tích ngữ nghĩa

Mô tả các bước dịch (2)



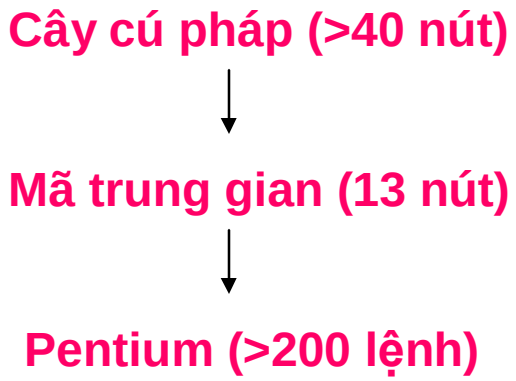
Ngôn ngữ trung gian

- Là ngôn ngữ cho một loại máy *trừu tượng*
- Cho phép sinh mã không phụ thuộc vào máy đích
- Cho phép tối ưu mã trước khi sinh mã máy thật sự



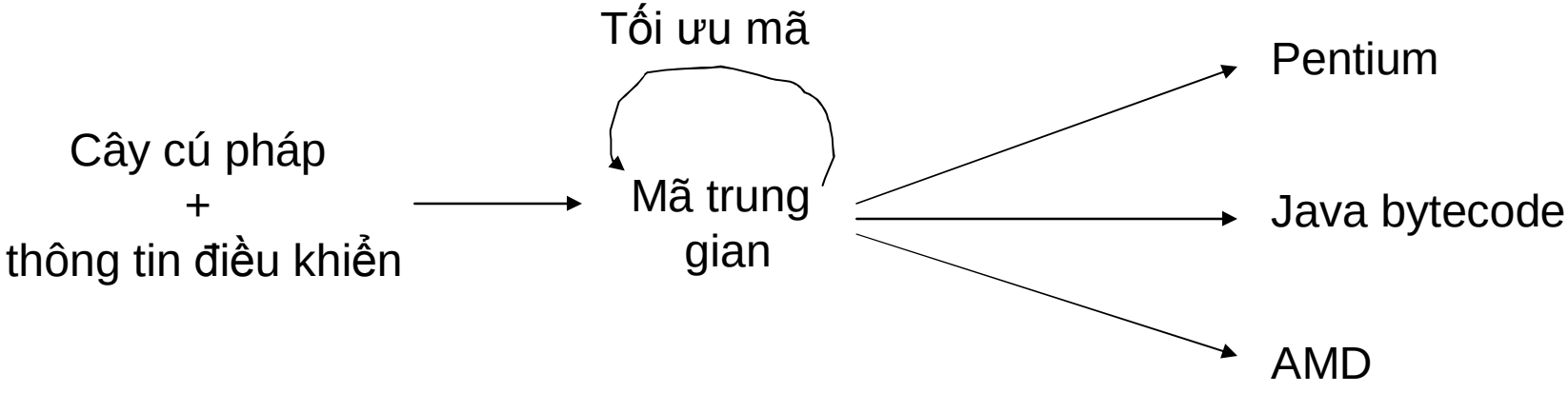
Ngôn ngữ trung gian

- Dễ sinh ra từ cây cú pháp
- Dễ sinh mã máy
- Số lượng lệnh nhỏ, gọn
  - Dễ tối ưu mã
  - Dễ chuyển sang loại mã máy khác



Ngôn ngữ trung gian

- Một dạng thể hiện của chương trình nằm giữa cây cú pháp điều khiển và mã máy
- Sử dụng
  - Lệnh nhảy
  - Thanh ghi
  - Vị trí trên bộ nhớ



Một ngôn ngữ trung gian

- **IR** (**I**ntermediate **R**epresentation) là một cây thể hiện các lệnh của một loại máy trừu tượng
- Nút lệnh không trả lại giá trị, được thực hiện theo thứ tự nhất định
  - Ví dụ: MOVE, SEQ, CJUMP
- Nút biểu thức trả lại giá trị, các nút con có thể thực hiện theo thứ tự bất kì
  - Ví dụ: ADD, SUB
  - Cho phép tối ưu mã

Mô tả các nút biểu thức của IR

- CONST(i): hằng số nguyên i
- TEMP(t): thanh ghi t, máy trừu tượng có vô hạn thanh ghi.
- OP(e_1, e_2): các phép toán
  - Số học: ADD, SUB, MUL, DIV, MOD
  - Logic: AND, OR, XOR, LSHIFT, RSHIFT
  - So sánh: EQ, NEQ, LT, GT, LEQ, GEQ
- MEM(e): giá trị bộ nhớ ở vị trí e
- CALL(f, a_0, a_1, …): giá trị của hàm f với các tham số a_0, a_1, …
- NAME(n): địa chỉ của lệnh hoặc dữ liệu có tên là n
- ESEQ(s, e): giá trị của e sau khi lệnh s được thực hiện

CONST

- Nút CONST đại diện cho hằng số

CONST(i)

- Giá trị của nút là i

TEMP

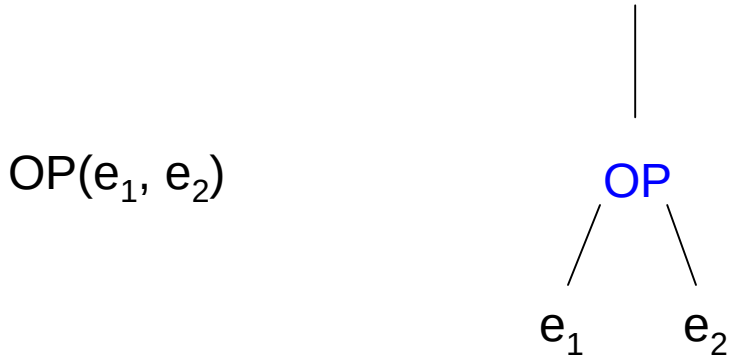
- Nút TEMP đại diện cho một thanh ghi trong số vô hạn các thanh ghi của máy trừu tượng

TEMP(t)

- Các biến cục bộ và các biến tạm
- Để dễ viết, ký hiệu FP = TEMP(FP) là địa chỉ bắt đầu bộ nhớ của hàm
- Giá trị của nút là giá trị của thanh ghi tại thời điểm tính toán

Toán tử

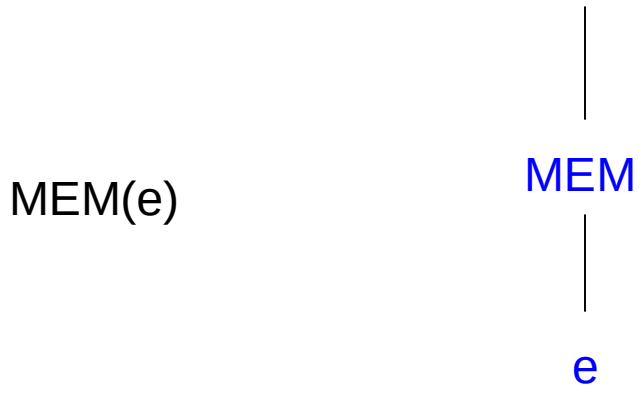
- Máy trừu tượng có nhiều phép toán



- Tính giá trị của e_1 và e_2, sau đó áp dụng phép toán với các giá trị này
- e_1 và e_2 phải là hai nút có giá trị
- Có thể tính giá trị e_1 và e_2 theo thứ tự bất kì

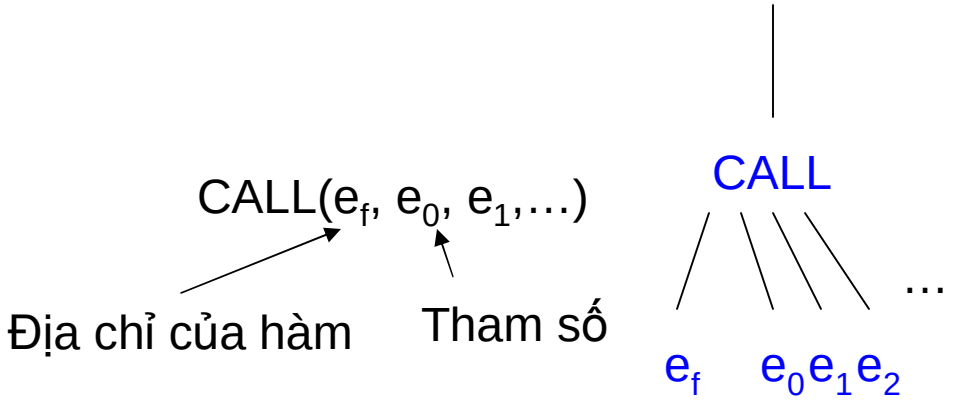
MEM

- Nút MEM đại diện cho một vị trí trong bộ nhớ
- Giá trị của nút là giá trị tại vị trí e trong bộ nhớ



CALL

- Nút CALL đại diện cho một lời gọi hàm



- Không định nghĩa cách cài đặt việc truyền tham số, quản lý ngăn xếp
- Giá trị của nút là giá trị của hàm

NAME

- Nút NAME đại diện cho địa chỉ của một tên trên bộ nhớ
- VD: địa chỉ của một nhãn nhảy

NAME(n)

ESEQ

- Nút ESEQ tính toán giá trị của biểu thức e sau khi thực hiện lệnh s

ESEQ(s, e)

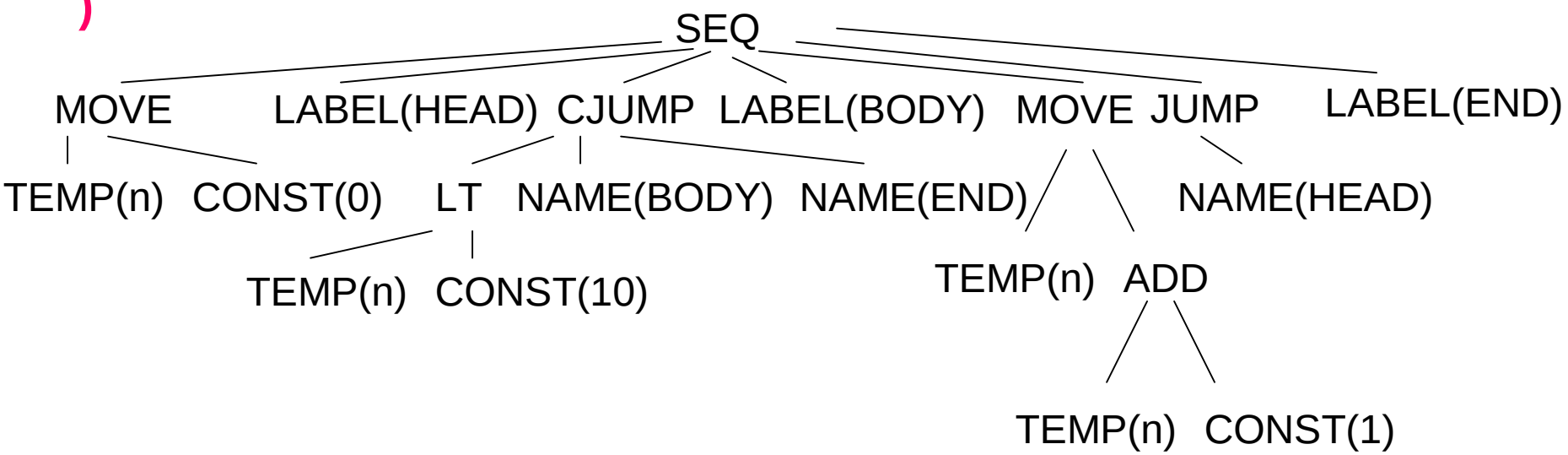
Mô tả các nút lệnh của IR

- MOVE(dest, e): chuyển giá trị của e vào dest
- EXP(e): tính toán giá trị của e, không cần lưu lại kết quả
- SEQ(s_1, s_2, … s_n): thực hiện các lệnh theo thứ tự
- JUMP(e): nhảy đến địa chỉ e
- CJUMP(e, l_1, l_2): nhảy đến l_1 hoặc l_2 tuỳ thuộc vào giá trị của e là true hoặc false
- LABEL(n): tạo ra nhãn có tên n

Ví dụ

```
n = 0;
while (n < 10) {
  n = n + 1;
}
```

```
SEQ(
  MOVE(TEMP(n), CONST(0)),
  LABEL(HEAD),
  CJUMP(LT(TEMP(n), CONST(10)), NAME(BODY), NAME(END)),
  LABEL(BODY),
  MOVE(TEMP(n), ADD(TEMP(n), CONST(1))),
  JUMP(NAME(HEAD)),
  LABEL(END)
)
```



Cấu trúc của IR

- Gốc của cây là một nút lệnh
- Các nút biểu thức nằm dưới nút lệnh
- Chỉ có nút biểu thức ESEQ có nút lệnh nằm dưới
- Có thể duyệt cây IR để chạy chương trình

Sinh cây IR (mã trung gian)

- Kỹ thuật: phương pháp dịch sử dụng cú pháp điều khiển (giống kiểm tra kiểu)
- Chuyển cây cú pháp điều khiển thành cây IR
- Mỗi cây con của cây cú pháp được chuyển thành một cây con dạng IR có cùng giá trị

Sinh cây IR

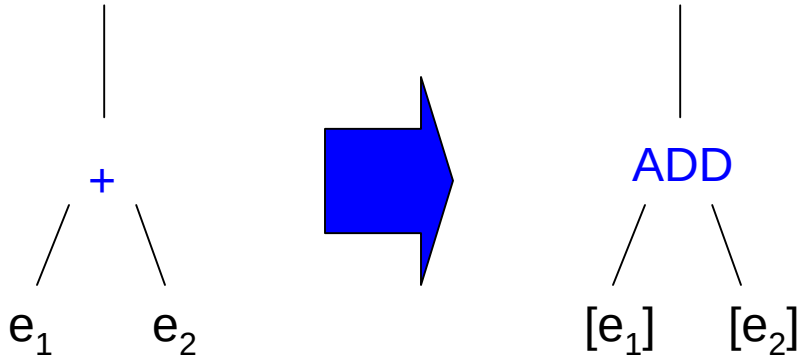
- Giống kiểm tra kiểu: thêm một phương thức vào nút tương ứng trong cây cú pháp

```
abstract class ASTNode {
  IRNode translate(SymTab A) { … }
}
```

- Cài đặt kiểu đệ quy
- Vấn đề: giống như kiểm tra kiểu, cần mô tả chính xác cách viết hàm translate()

Biểu thức

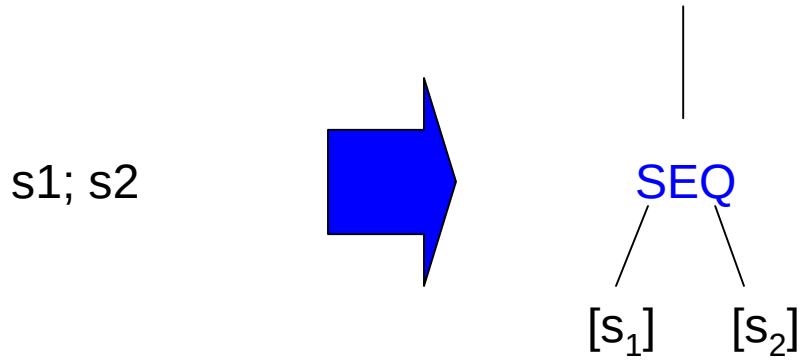
- Các nút của cây cú pháp thể hiện biểu thức được chuyển thành nút IR tương ứng



- Kí hiệu [e] là biểu diễn IR của nút e trong cây cú pháp

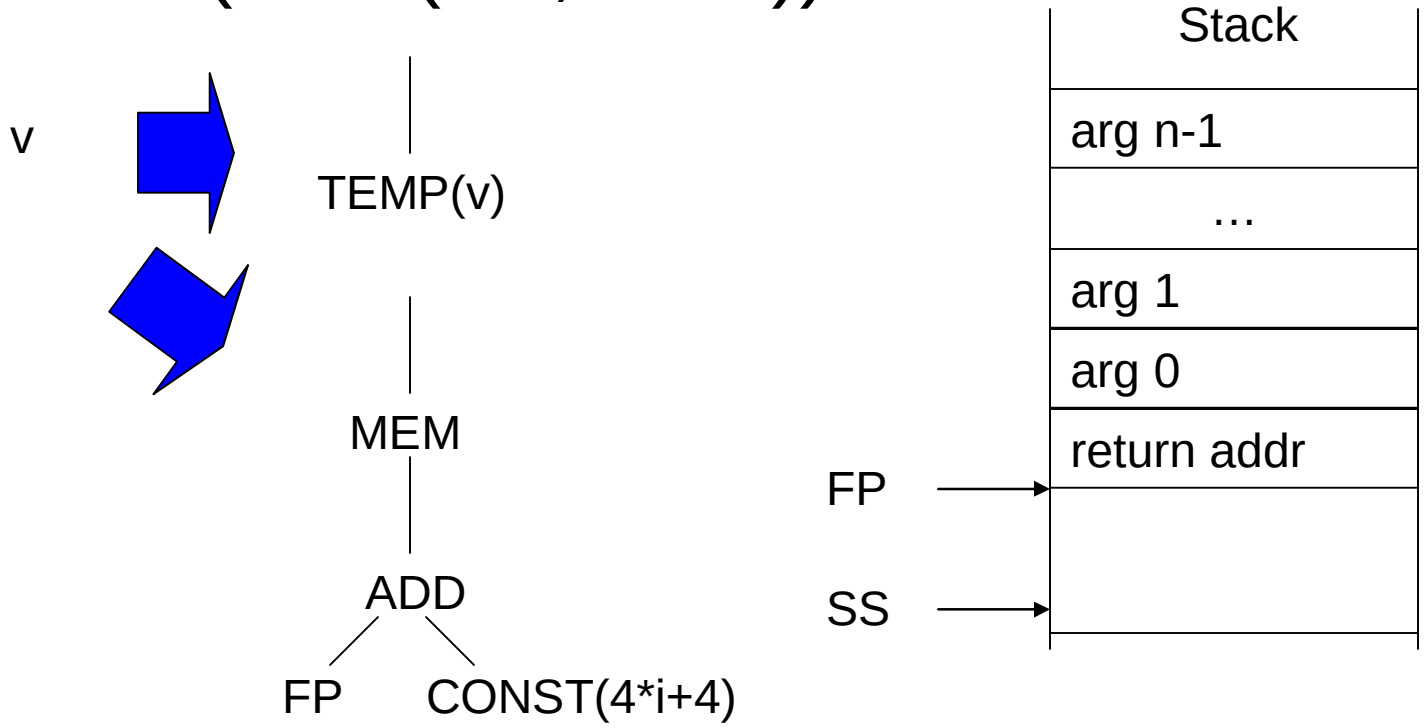
Câu lệnh

- Dãy các lệnh được biểu diễn bằng nút SEQ trong biểu diễn IR
- Nếu $[s_1]$ và $[s_2]$ là biểu diễn IR của nút s_1 và s_2
- thì SEQ($[s_1]$, $[s_2]$) là biểu diễn IR của s_1; s_2



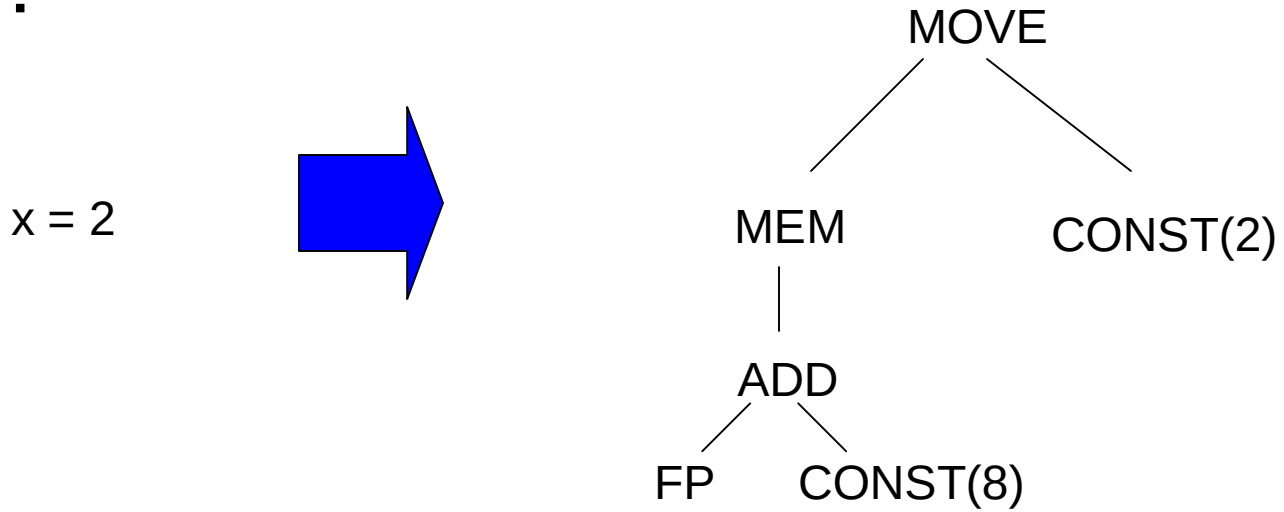
Biến

- Biến cục bộ v chuyển thành nút TEMP(v)
- Tham số thứ i nằm ở vị trí MEM(ADD(FP,4*i+4))



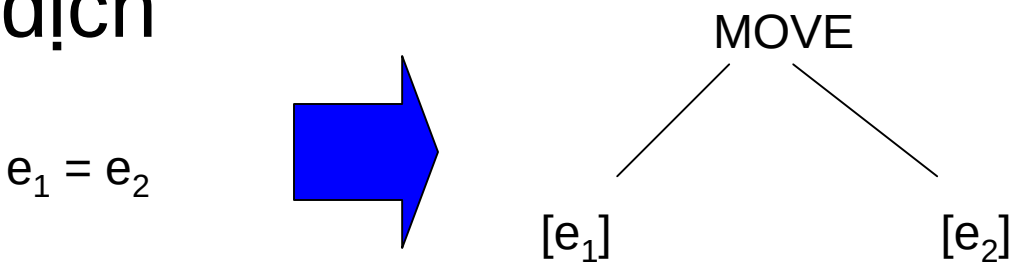
Phép gán

- Phép gán v = e chuyển thành nút MOVE(dest, [e]) với dest là địa chỉ của v, [e] là biểu diễn IR của e
- Ví dụ

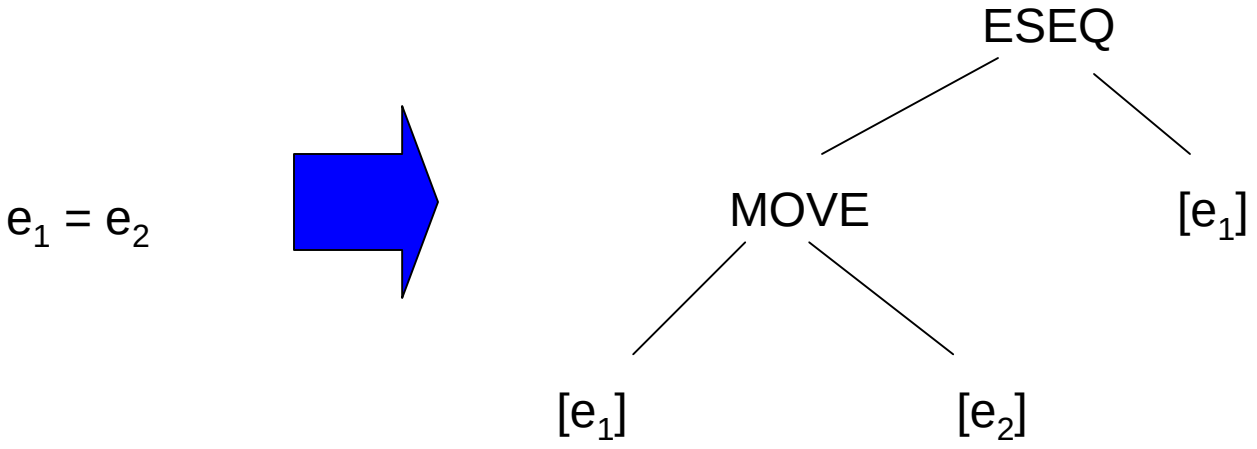


Phép gán

- Cách dịch



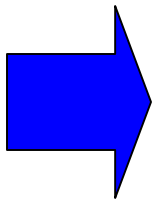
- Vấn đề: nút MOVE không có giá trị, làm thế nào để dịch x = (y = 2)?

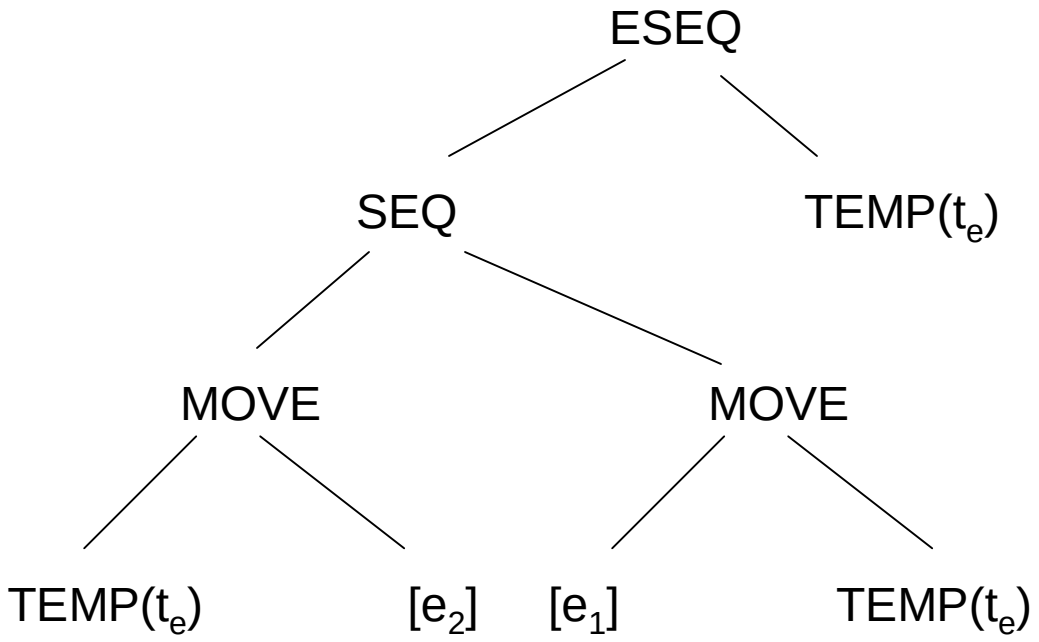


Phép gán

- Như vậy, [e_1] phải chạy 2 lần, cần lưu lại giá trị của [e_1]

$e_1 = e_2$





Nhập môn Chương trình dịch

Học kì II 2006-2007

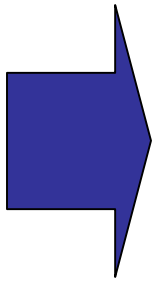
Bài 14: Sinh mã trung gian (tiếp)

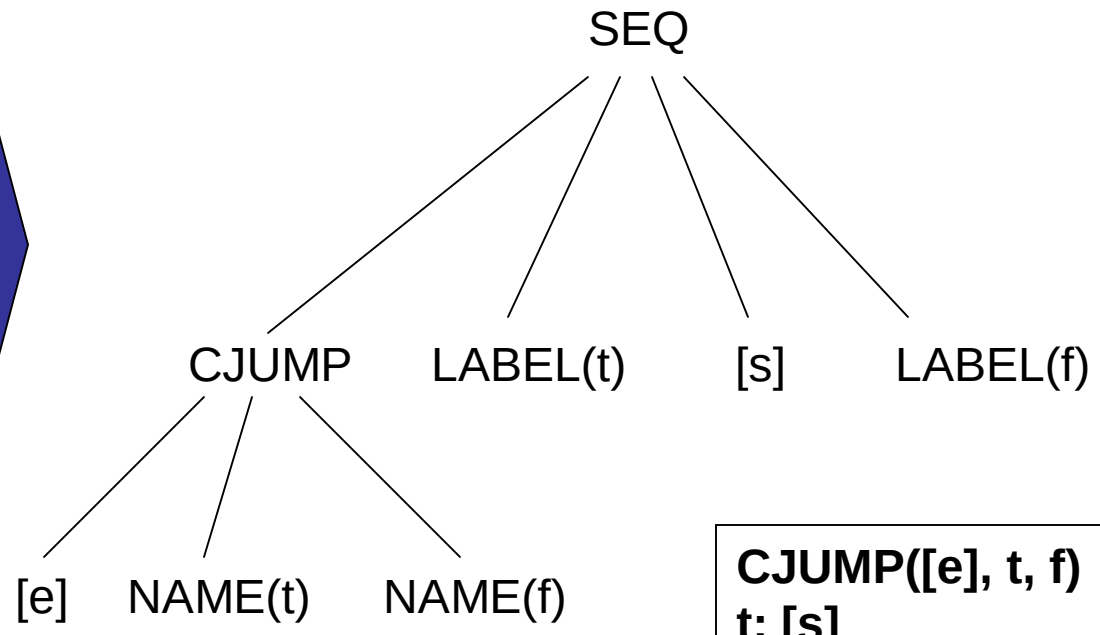
Sinh mã trung gian

- Sử dụng cú pháp điều khiển (giống kiểm tra kiểu)
- Sinh mã các nút biểu thức hoặc nút lệnh dựa vào mã của các nút con
- Cú pháp điều khiển
  - Mô tả chính xác chương trình dịch cần làm gì
  - Có thể cài đặt dễ dàng
  - Có thể chứng minh tính đúng của chương trình dịch

Sinh mã lệnh if

if (e) s

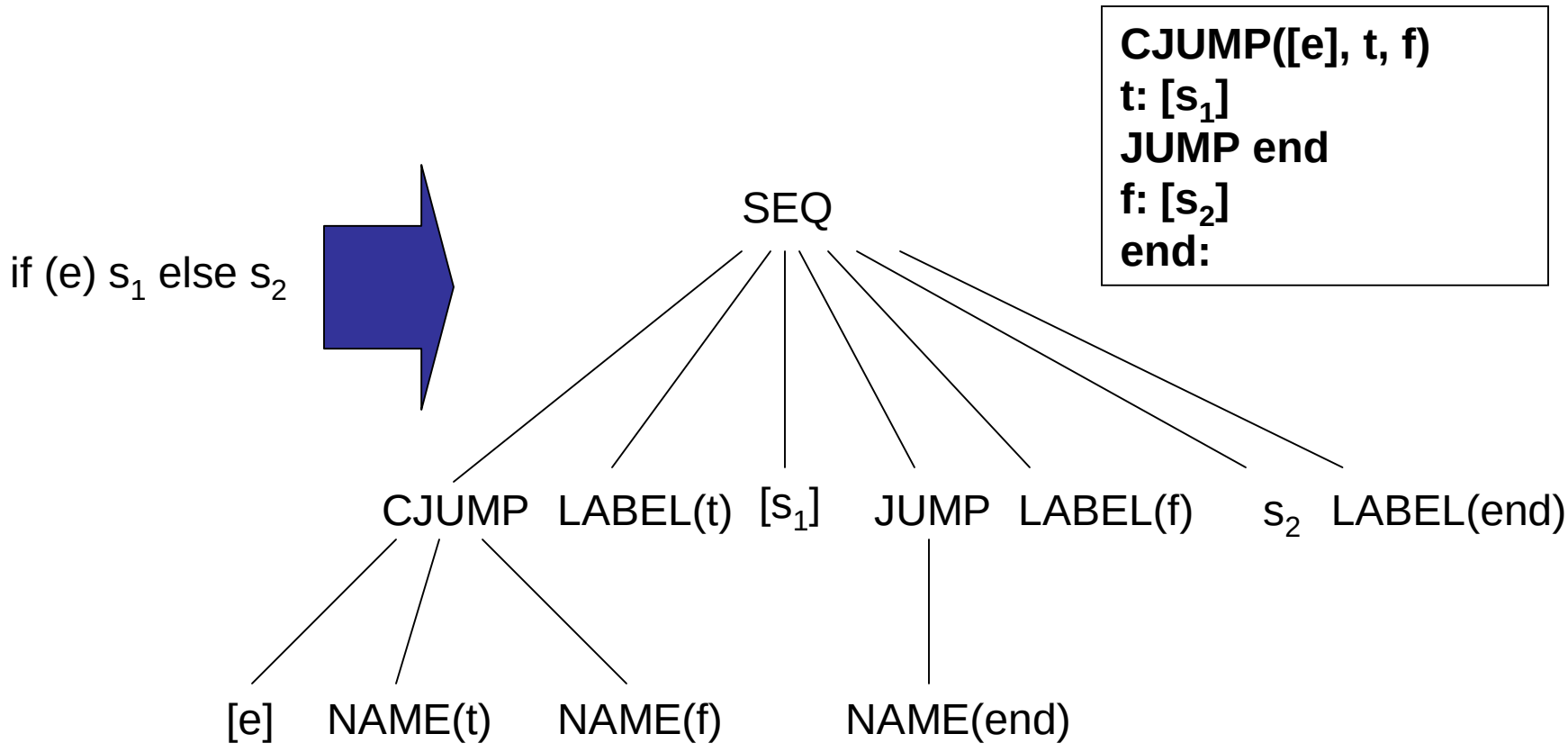




CJUMP([e], t, f)
t: [s]
f:

[if (e) s] = SEQ(CJUMP([e], NAME(t), NAME(f)), LABEL(t), [s], LABEL(f))

Sinh mã lệnh if-else

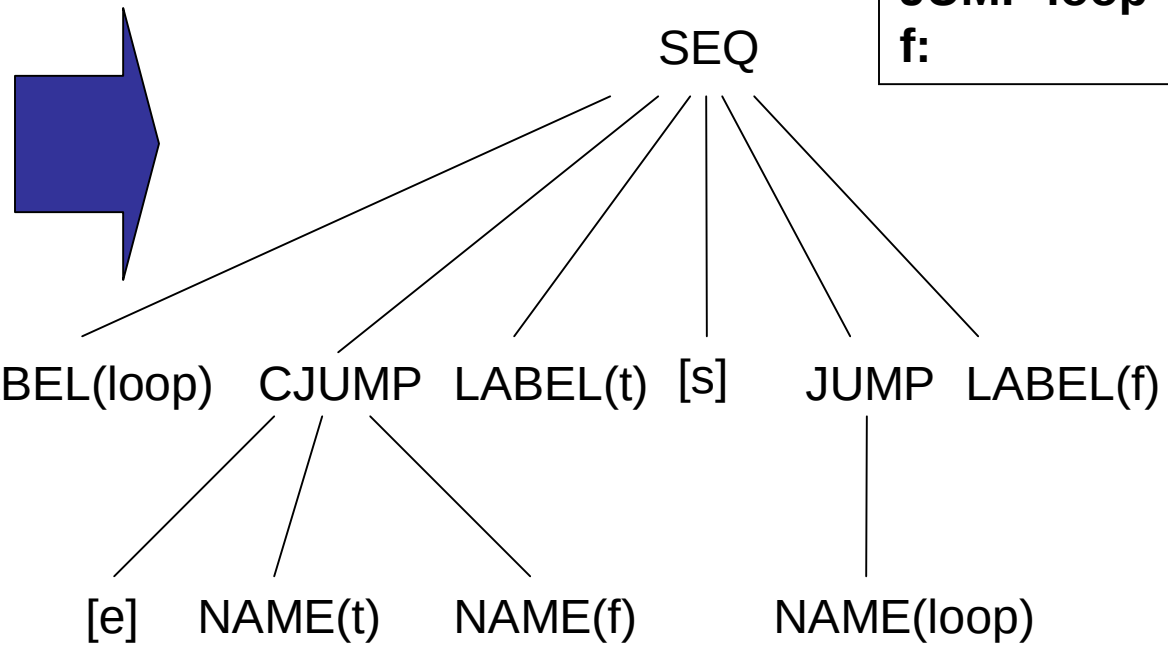


[if (e) s_1 else s_2] = SEQ(CJUMP([e], NAME(t), NAME(f)), LABEL(t), [s_1], JUMP(NAME(end)), LABEL(f), [s_2], LABEL(end))

Sinh mã lệnh while

loop: CJUMP([e], t, f)
t: [s]
JUMP loop
f:

while (e) s



[while (e) s] = SEQ(LABEL(loop), CJUMP([e], NAME(t), NAME(f)), LABEL(t), [s], JUMP(NAME(loop)), LABEL(f))

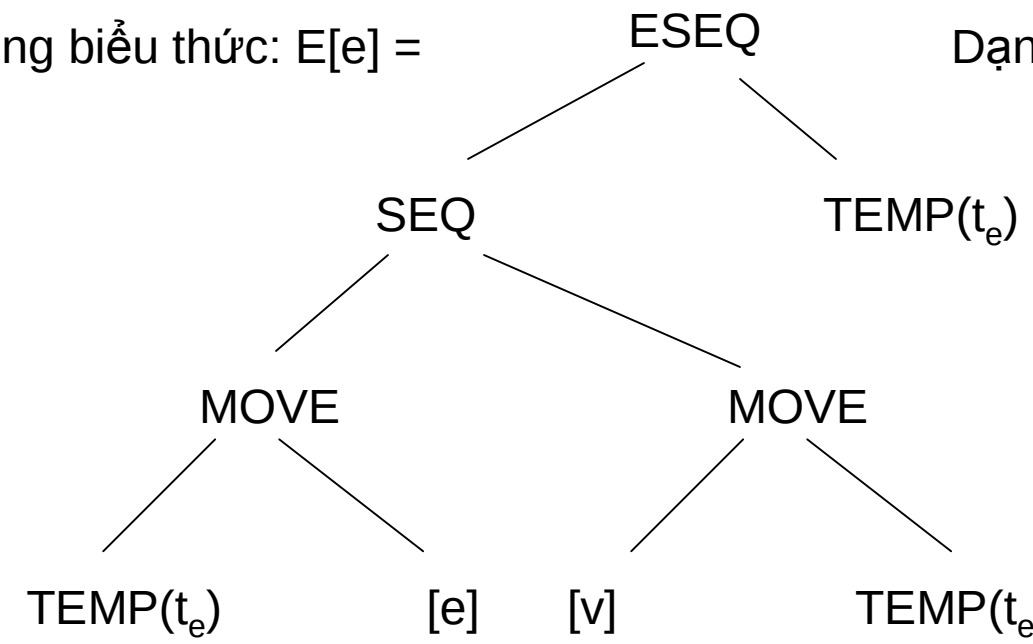
Cài đặt

```
abstract class Node {
  abstract IRnode translate(); …
}

// if (e) s = SEQ(CJUMP(e, t, f), LABEL(t), s, LABEL(f ))

class IfNode extends Node { …
  IRnode translate() {
    SeqNode ret = new SEQ();
    ret.append(new CJUMP(e.translate(), “t”, “f”));
    ret.append(new LABEL(“t”));
    ret.append(s.translate());
    ret.append(new LABEL(“f”));
    return ret;
  } …
}
```

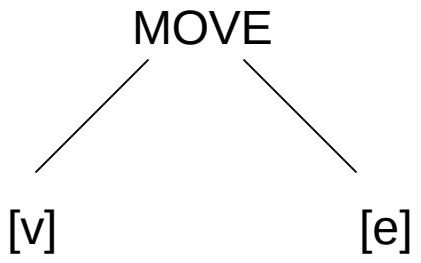
Trường hợp có nhiều cách dịch

v = e

Dạng biểu thức: E[e] =



Dạng câu lệnh: S[e] =



Cài đặt

```
abstract class Node { ...
  abstract IRnode translateE();
  abstract IRnode translateS();
  abstract IRnode translateC(); …
}
class Assignment {
  Expr variable, value;
  IRnode translateS() {
    return new MOVE(variable.translateE(),
                    value.translateE());
  }
  IRnode translateE() {
    TEMP t = freshTemp(); // new TEMP()
    return new ESEQ(new SEQ(new MOVE(t, value.translateE()),
                            new MOVE(variable.translateE(), t)),
                    t);
  }
}
```

Một số kí hiệu

- E[e] : cây IR (biểu thức) trả lại giá trị của biểu thức e
- S[s] : cây IR (câu lệnh) làm các công việc của lệnh s
- C[e, l_1, l_2] với e là biểu thức logic: cây IR nhảy đến nhãn l_1 nếu e đúng (true), nhảy đến nhãn l_2 nếu e sai (false).

Các lệnh đã mô tả

- E[v] = TEMP(v)
- E[e_1 + e_2] = ADD([e_1], [e_2])
- S[v = e] = MOVE([v], [e])
- E[v = e] = ESEQ(SEQ(MOVE(TEMP(t), e), MOVE(v, TEMP(t))), TEMP(t))
- S[if (e) s] = SEQ(…)
- S[if (e) s_1 else s_2] = …
- S[while (e) s] = …

Sinh mã hàm

- Giả sử thân hàm là lệnh s với mã IR là S[s]
- Làm thế nào để sinh mã cho lệnh return?
- Ý tưởng: thêm vào một biến RV (return value) và một nhãn ở cuối hàm
- Hàm có thể được dịch sang mã sau

SEQ(S[s], LABEL(epilogue))

- Lệnh return e có thể được dịch sang mã sau

S[return e] = SEQ(MOVE(TEMP(RV), E[e]), JUMP(NAME(epilogue)))

Biểu thức logic

- Ví dụ: e_1 & e_2
- Có nhiều cách tính
  - Sử dụng toán tử có sẵn:
    $E[e_1 \& e_2] = AND([e_1], [e_2])$
  - Tự tính toán

```
ESEQ(SEQ(MOVE(TEMP(x), 0),
                CJUMP([e1], t1, no_set),
                LABEL(t1),
                CJUMP([e2], t2, no_set)
                LABEL(t2),
                MOVE(TEMP(x), 1),
                LABEL(no_set)),
          TEMP(x)
)
```

Biểu thức logic trong câu lệnh điều kiện

[if (e) s] = SEQ(CJUMP([e], NAME(t), NAME(f)), LABEL(t), [s], LABEL(f))

[if (e_1 & e_2) s] = SEQ(CJUMP([e_1 & e_2], NAME(t), NAME(f)),
LABEL(t), [s], LABEL(f))
= SEQ(CJUMP(ESEQ(SEQ(MOVE(TEMP(x), 0),
CJUMP([e1], t1, no_set),
LABEL(t1),
CJUMP([e2], t2, no_set)
LABEL(t2),
MOVE(TEMP(x), 1),
LABEL(no_set)),
TEMP(x)), NAME(t), NAME(f)),
LABEL(t), [s], LABEL(f))

Mã lệnh tồi !

Mã lệnh tốt hơn !

[if (e1 & e2) s] = SEQ(CJUMP([e_1], t1, f), LABEL(t1),
CJUMP([e_2], t2, f), LABEL(t2), [s], LABEL(f))

Biểu thức logic trong câu lệnh điều kiện

- Ý tưởng: biểu diễn giá trị logic bằng nhãn nhảy thay vì giá trị đúng/sai
- Định nghĩa: C[e, l_1,l_2] là cây IR nhảy đến nhãn l_1 nếu e đúng và nhảy đến nhãn l_2 nếu e sai
- Công thức hồi quy:
  - C[true, l_1, l_2] = JUMP(NAME(l_1))
  - C[false, l_1, l_2] = JUMP(NAME(l_2))
  - C[e_1==e_2, l_1, l_2] = CJUMP(EQ(E[e_1], E[e_2]), l_1, l_2)
  - C[e_1 & e_2, l_1, l_2] = SEQ(C[e_1, t, l_2], LABEL(t), C[e_2, l_1, l_2])
  - và công thức hồi quy cho các phép toán quan hệ, logic khác

Biểu thức logic trong câu lệnh điều kiện

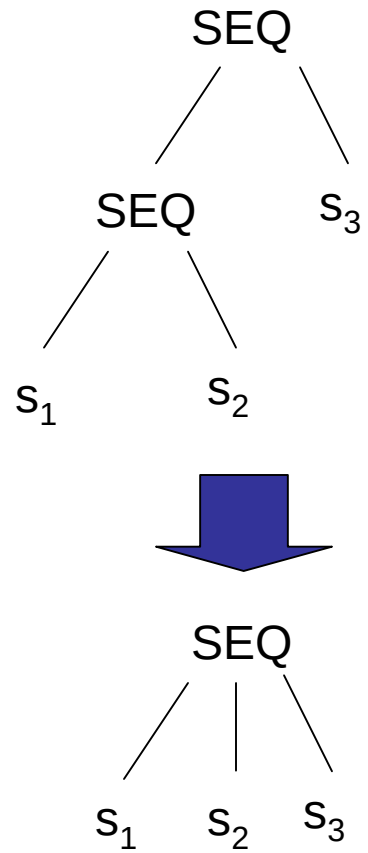
- C[e, l_1,l_2] là cây IR nhảy đến nhãn l_1 nếu e đúng và nhảy đến nhãn l_2 nếu e sai
- Câu lệnh if

S[if (e) s] = SEQ(C[e, t, f], LABEL(t), [s], LABEL(f))

S[if (e_1 & e_2) s] = SEQ(C[e_1 & e_2, t, f], LABEL(t),
[s], LABEL(f))
= SEQ(SEQ(C[e_1, n, f], LABEL(n),
C[e_2, t, f]),
LABEL(t), [s], LABEL(s))

làm phẳng cây IR

= SEQ(C[e_1, n, f], LABEL(n), C[e_2, t, f],
LABEL(t), [s], LABEL(f))



Tổng kết

- Cú pháp điều khiển mô tả cách chuyển cây cú pháp thành cây IR
- IR khá giống với mã máy, tuy nhiên
  - Cây IR có độ sâu tuỳ ý, mã máy là các lệnh kế tiếp nhau
  - Cây IR cho phép thực hiện các biểu thức có các tác dụng phụ (VD: ESEQ, CALL)
  - CJUMP bắt buộc phải có 2 nhãn nhảy
- Buổi sau: làm phẳng cây IR

Nhập môn Chương trình dịch

Học kì II 2006 – 2007

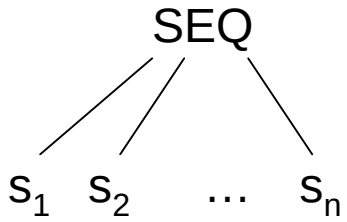
Bài 15: Làm phẳng cây IR

Làm phẳng cây IR

- Cây IR vẫn còn cấu trúc đệ quy của cây cú pháp
- Mã máy là một dãy liên tiếp các lệnh
- Cần làm phẳng cây IR (đưa về cây có độ cao bằng 1) trước khi sinh mã
- Ở dạng phẳng, các lệnh được đưa đến sát gốc của cây

Dạng IR phẳng

- Chỉ có một nút SEQ làm gốc của cây IR



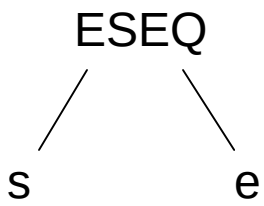
- Một hàm được biểu diễn dưới dạng SEQ(s_1, s_2, … s_n)
- Có thể dịch thành mã mãy bằng cách dịch lần lượt s_1, s_2, …, s_n rồi nối mã lại với nhau.

Dạng IR phẳng

- Ý tưởng: viết lại cây IR nhưng lược bớt các cấu trúc không thích hợp với việc sinh mã máy
  - Các cây con biểu thức
  - Các cây con với gốc là ESEQ hoặc CALL
    → triệt tiêu ESEQ và chuyển CALL về gốc

Ví dụ: không có nút ESEQ

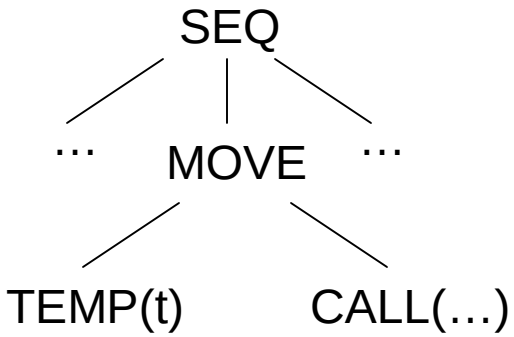
- ESEQ cho phép tính biểu thức sau khi thực hiện lệnh

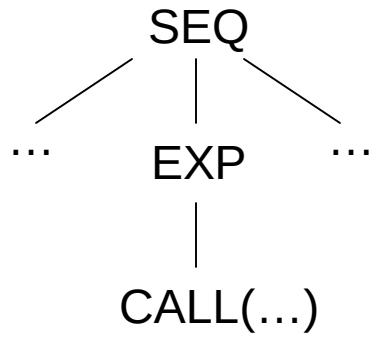


- Ví dụ: S[x = a[i = i + 1];] = ?
- Ở dạng IR phẳng: S[i = i + 1]; S[x = a[i]];

Ví dụ: các lệnh CALL

- Cần chuyển các lệnh CALL về gần gốc của cây IR
- Có hai loại CALL
  - Cần lưu giá trị: MOVE(TEMP(t), CALL(…))
  - Không cần lưu giá trị: EXP(CALL(…))

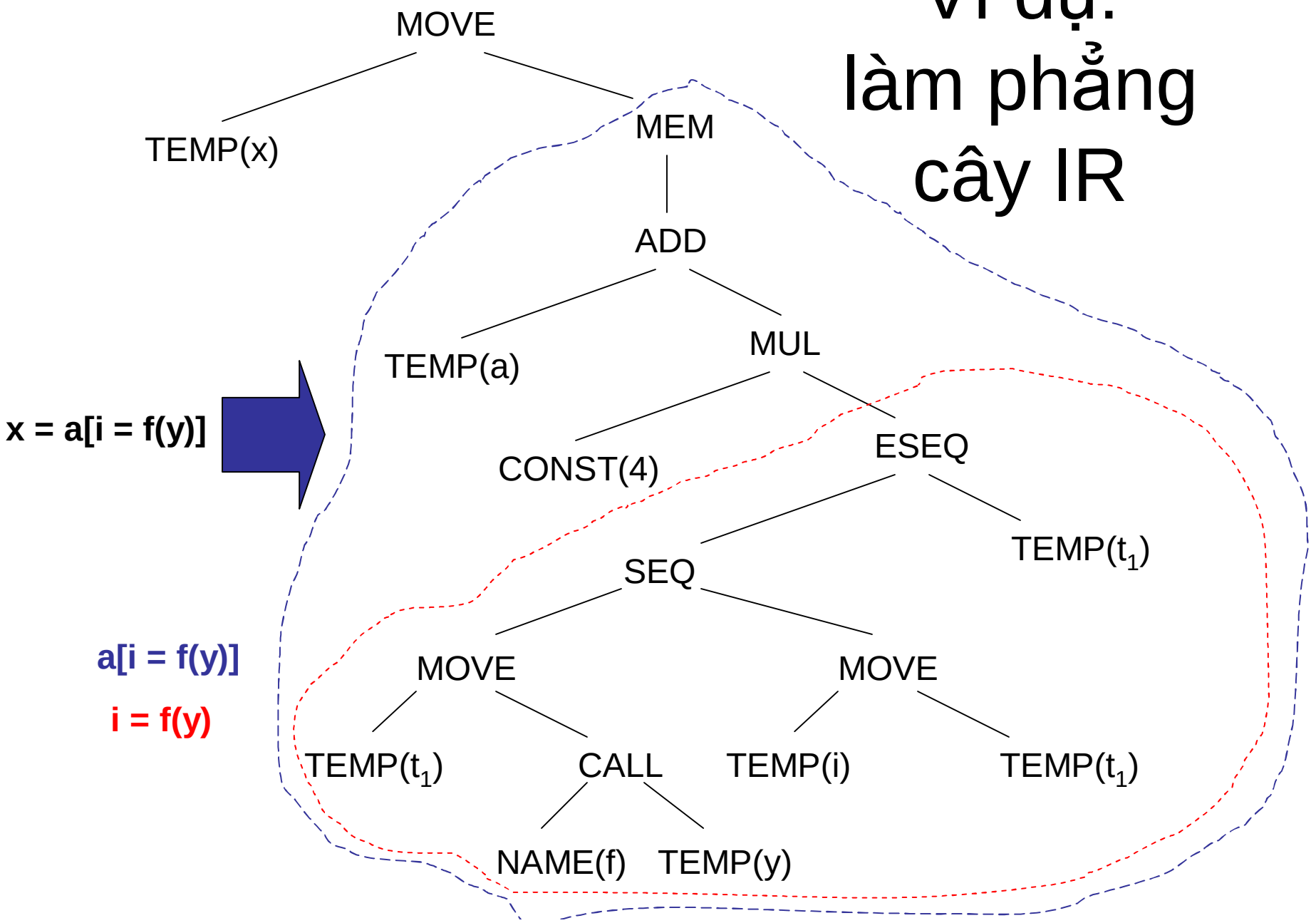




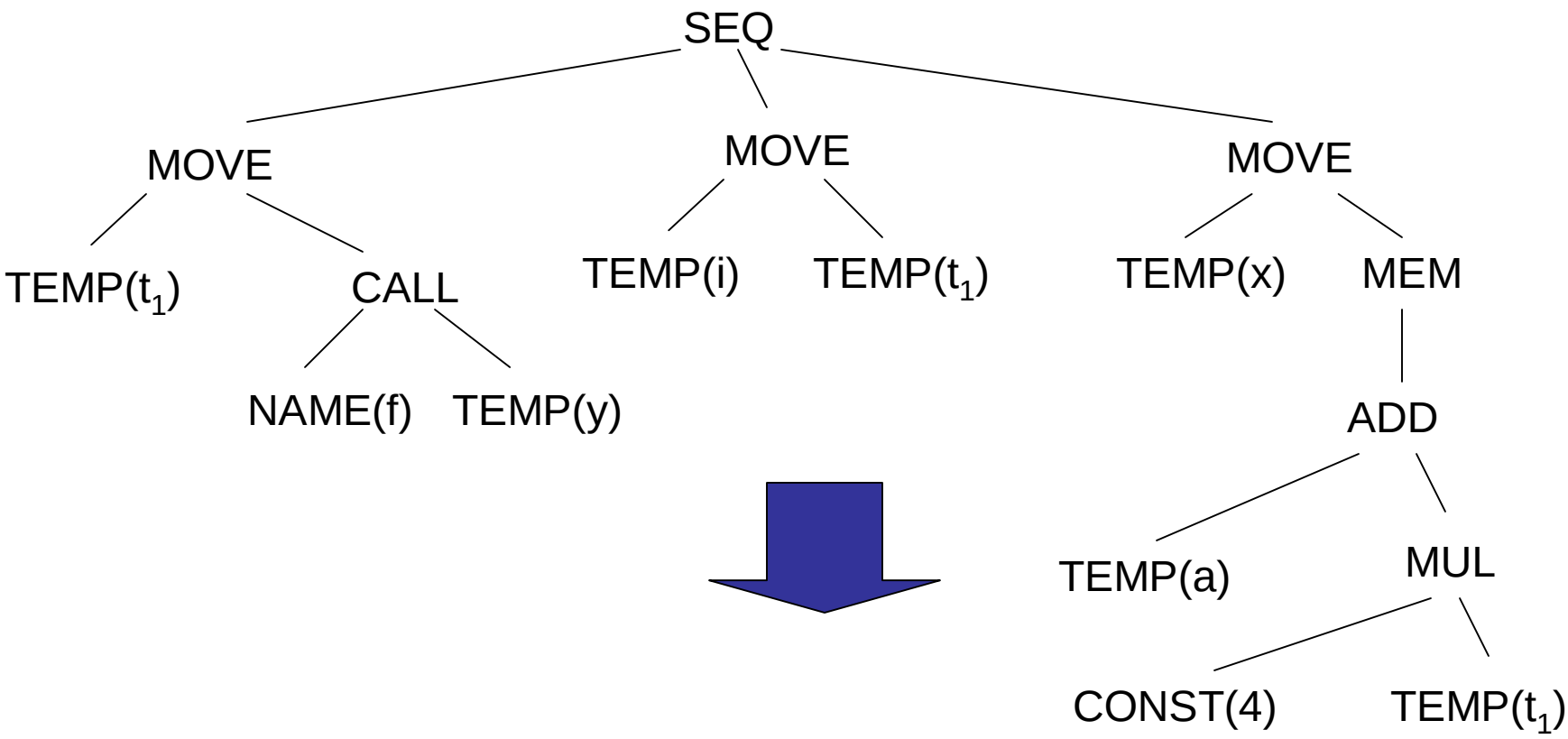
Dạng IR phẳng

- Ở dạng phẳng, các nút con của gốc chỉ có các dạng
  - MOVE(*dest*, *e*)
  - MOVE(TEMP(*t*), CALL(…))
  - EXP(CALL(…))
  - JUMP(*e*)
  - CJUMP(*e*, l_1, l_2)
  - LABEL(*l*)
- Có thể dễ dàng chuyển thành mã máy
- Kí hiệu J[s] là dạng phẳng của cây IR s

Ví dụ: làm phẳng cây IR



Ví dụ: Làm phẳng cây IR




```
push y
call f
move t1, rv
```

```
move i, t1
```

```
move x, [a + i * 4]
```

Ví dụ: Làm phẳng lệnh ESEQ

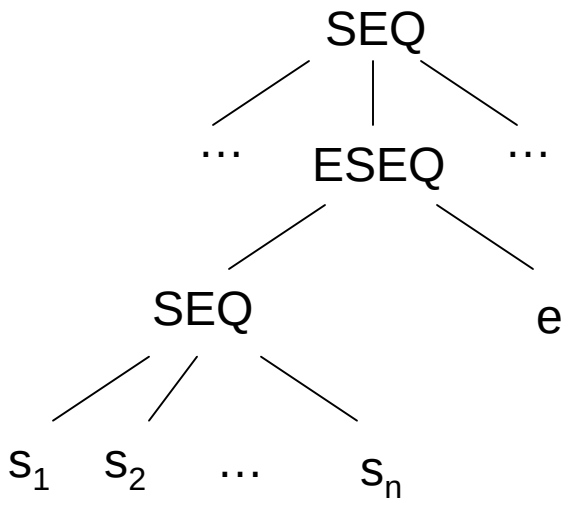
- Chuyển các lệnh ESEQ về gốc để có thể chuyển thành lệnh SEQ
- Ý tưởng: sử dụng cú pháp điều khiển tại các nút của cây IR để đưa ESEQ về gốc

Cú pháp điều khiển: ESEQ

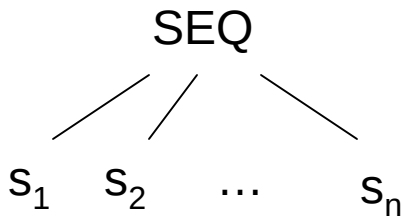
- ESEQ(s_1, ESEQ(s_2, e)) → ESEQ(SEQ(s_1, s_2), e)
- MOVE(ESEQ(s_1, e), dest) → SEQ(s_1, MOVE(e, dest))
- OP(ESEQ(s, e_1), e_2) → ESEQ(s, OP(e_1, e_2))
- OP(e_1, ESEQ(s, e_2)) → ESEQ(s, OP(e_1, e_2))

Làm phẳng IR: lệnh ESEQ

- Sau khi chuyển các lệnh ESEQ đến gốc của cây IR



- Có thể thay cả lệnh ESEQ bằng



tại sao?

Cài đặt

```
class CanonicalExpr {
  IRStmt[ ] pre_stmts;
  IRExpr expr;
}

class CanonicalStmt {
  IRStmt[ ] stmts;
}

abstract class IRExpr { CanonicalExpr simplify(); }
abstract class IRStmt { CanonicalStmt simplify( ); }
```

Cài đặt

Cần cài đặt 2 hàm simplify

- J[e]: trả lại dãy các lệnh (s_1, s_2, …, s_n) và biểu thức e’ (đã phẳng hoá) sao cho thực hiện liên tiếp s_1, … s_n rồi tính e’ tương đương với mã IR tính e (IRExpr.simplify)
- J[s]: trả lại dãy các lệnh (s_1, … s_n) (đã phẳng hoá) sao cho thực hiện liên tiếp s_1, … s_n tương đương với mã IR s (IRStmt.simplify)

Cú pháp điều khiển (phẳng hoá)

- Mục tiêu: định nghĩa cú pháp điều khiển J[e] và J[s] cho tất cả 13 nút của cây IR.
- 4 trường hợp đơn giản:
  - J[CONST(i)] = ( ); CONST(i)
  - J[NAME(n)] = ( ); NAME(n)
  - J[TEMP(t)] = ( ); TEMP(t)
  - J[LABEL(l)] = LABEL(l)
- 4 lệnh trên đã ở dạng phẳng

Cú pháp điều khiển

- JUMP(e), CJUMP(e, l_1, l_2), MEM(e)
  - Cần phẳng hoá e trước
- Viết dưới dạng luật

$$\frac{J[e] = (s_1, s_2, \ldots, s_n); e'}{J[JUMP(e)] = (s_1, s_2, \ldots, s_n, JUMP(e'))}$$

$$\frac{J[e] = (s_1, s_2, \ldots, s_n); e'}{J[CJUMP(e, l_1, l_2)] = (s_1, s_2, \ldots, s_n, CJUMP(e', l_1, l_2))}$$

$$\frac{J[e] = (s_1, s_2, \ldots, s_n); e'}{J[MEM(e)] = (s_1, s_2, \ldots, s_n); MEM(e')}$$

Cú pháp điều khiển: ESEQ

- Làm thế nào để phẳng hoá ESEQ(s, e)

$$\frac{J[e] = (s_1, s_2, \ldots, s_n);\ e'}{J[ESEQ(s, e)] = (s, s_1, s_2, \ldots, s_n);\ e'}$$

- Đã phẳng chưa?

Cú pháp điều khiển: ESEQ

- Cần phẳng hoá cả s
- Luật phẳng hoá ESEQ(s, e)

$$\frac{J[e] = (s_1, s_2, \ldots, s_n); e' \qquad J[s] = (s_1', s_2', \ldots, s_n')}{J[\mathrm{ESEQ}(s, e)] = (s_1', s_2', \ldots, s_n', s_1, s_2, \ldots, s_n); e'}$$

Cú pháp điều khiển: SEQ

- Phẳng hoá SEQ(s_1, s_2)
- Nối các lệnh của s_1 và s_2 lại

$$J[s_1] = (s_1, s_2, \ldots, s_n)$$

$$J[s_2] = (s_1', s_2', \ldots, s_n')$$

$$J[SEQ(s_1, s_2)] = (s_1, s_2, \ldots, s_n, s_1', s_2', \ldots, s_n')$$

Cú pháp điều khiển: EXP

- Nút EXP(e) không lưu giá trị của e
- Luật:

$$\frac{J[e] = (s_1, s_2, \ldots, s_n);\ e'}{J[EXP(e)] = (s_1, s_2, \ldots, s_n)}$$

Cú pháp điều khiển: OP

$$J[e_1] = (s_1, s_2, \ldots, s_n);\ e_1'$$

$$J[e_2] = (s_1', s_2', \ldots, s_n');\ e_2'$$

$$J[OP(e_1, e_2)] = (s_1, s_2, \ldots, s_n, s_1', s_2', \ldots, s_n');\ OP(e_1', e_2')$$

- Luật này đã thể hiện đúng ý đồ của người lập trình chưa?

Cú pháp điều khiển: OP

- Nếu s_i' làm thay đổi e_1 sẽ làm thay đổi ý đồ của người lập trình
- Cần lưu lại giá trị của e_1 trước khi tính s_i'

$$J[e_1] = (s_1, s_2, \ldots, s_n); e_1'$$

$$J[e_2] = (s_1', s_2', \ldots, s_n'); e_2'$$

$$J[OP(e_1, e_2)] = (s_1, s_2, \ldots, s_n, MOVE(TEMP(t), e_1'), s_1', s_2', \ldots, s_n'); OP(TEMP(t), e_2')$$

- Tốt, nhưng:
  - Cần thêm 1 biến t
  - Không cho phép tính OP(e_1', e_2') trong một phép tính

Cú pháp điều khiển: CALL

$J[e_f] = (s_1, s_2, \ldots, s_n); e_f'$

$J[e_1] = (s_1', s_2', \ldots, s_n'); e_1'$

$J[CALL(e_f, e_1)] = (s_1, s_2, \ldots, s_n, s_1', s_2', \ldots, s_n', MOVE(TEMP(t), CALL(e_f', e_1')); TEMP(t)$

Cú pháp điều khiển: MOVE

$J[dest] = (s_1, s_2, \ldots, s_n); d'$

$J[e] = (s_1', s_2', \ldots, s_n'); e'$

$J[MOVE(dest, e)] = (s_1', s_2', \ldots, s_n', MOVE(TEMP(t), e'), s_1, s_2, \ldots, s_n, MOVE(d', TEMP(t)))$

Tổng kết

- Sử dụng cú pháp điều khiển để thiết kế các hàm chuyển cây IR về dạng phẳng
- Cài đặt các hàm IRExpr.simplify và IRStmt.simplify
- Dạng IR phẳng: các lệnh được xếp liên tiếp nhau, sẵn sàng để dịch ra mã máy