Compiler (컴파일러)
Semantic Analysis
Abstract Syntax Tree, Syntax Directed Definitions, Symbol Tables

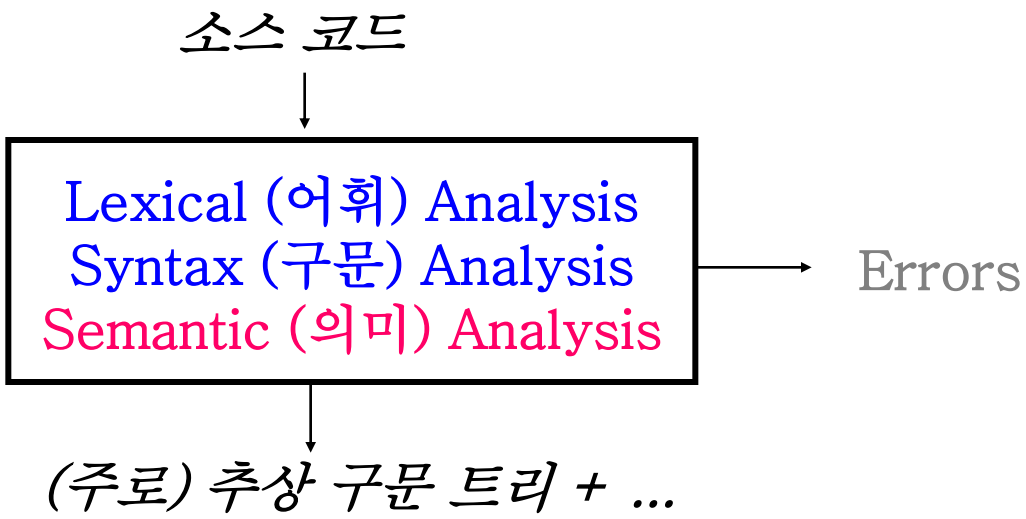
2015년 2학기

충남대학교 컴퓨터공학과

조은선

Semantic Analysis

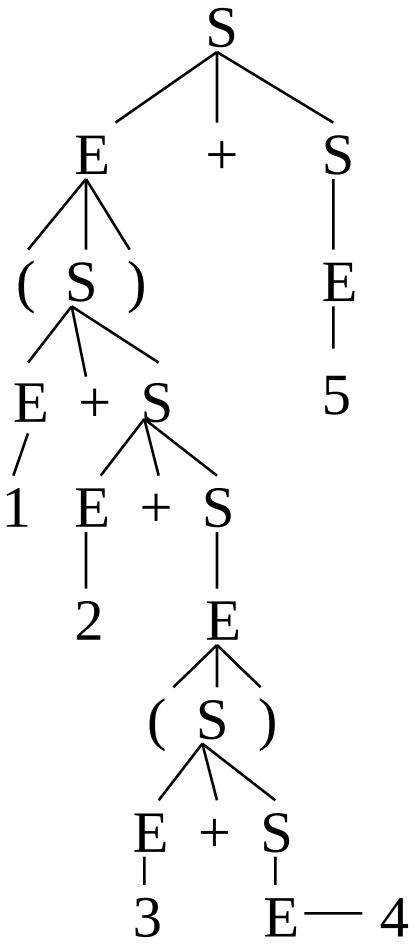
- Semantic Analysis = Syntax Analysis + *a*



- 먼저 두 가지 도구
  - Abstract Syntax Tree (AST)
  - Syntax-Directed Definition/Translation (SDT)

Abstract Syntax Tree

Parse Tree



- 파스트리 (parse tree)
  - 유도과정을 트리로 나타냄
  - terminal 이 leaf 노드
  - non-terminal이 중간노드
  - 유도 순서(left/right) 는 표현 안됨

Motivating Problem

How can I hide parens in ANTLR4?

1

For example, input = '(1+2)*3' .

tree is like that '(expr (expr ((expr (expr 1) + (expr 2))))*(expr 3))'

And then, I would like to hide or delete the '(' and ')' in the tree , they are no needed any more. I try to make it , but didn't.

1

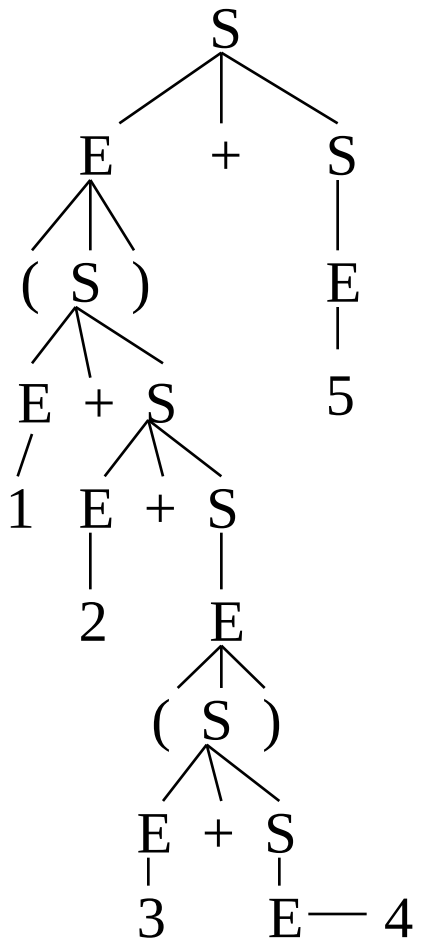
```
expr : ID LPAREN exprList? RPAREN
     | '-' expr
     | '!' expr
     | expr op=('*'|'/') expr
     | expr op=('+'|'-') expr
     | ID
     | INT
     | LPAREN expr RPAREN //### Parens Here ####
     ;
 LPAREN : '(' ;
 RPAREN : ')' ;
```

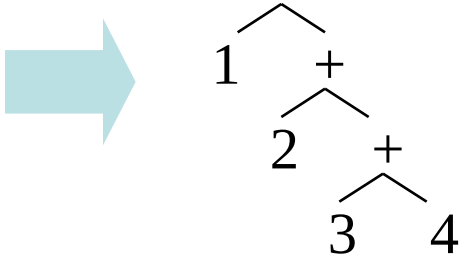
괄호는
파싱하고나면
불필요

stackoverflow

http://stackoverflow.com/questions/29455634/how-can-i-hide-parens-in-antlr4/29456792#29456792

Abstract Syntax Tree (AST)

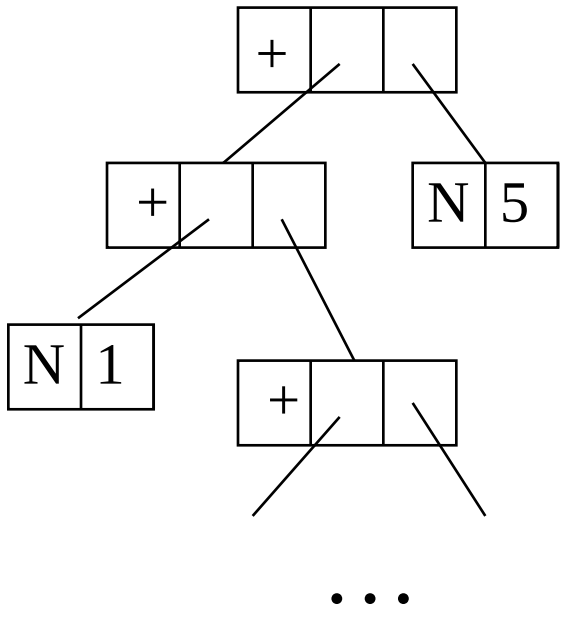




- AST
  - 파스트리에서 불필요한 정보를 제거한 형태

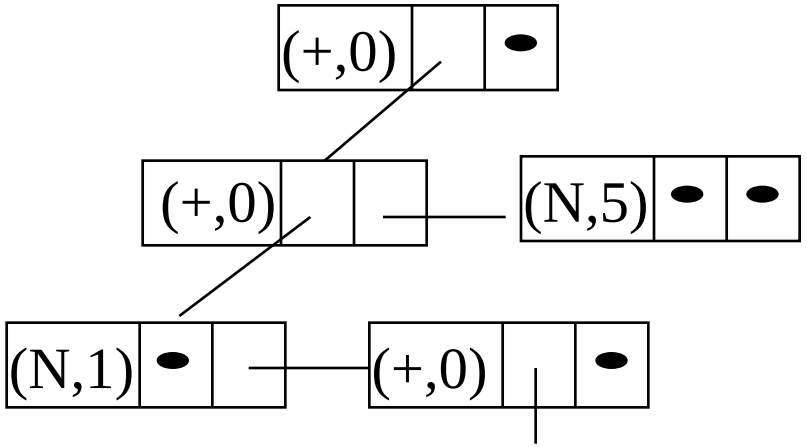
AST 자료구조 – Java 예

```
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;}
}
```



참고 : Visitor Pattern

AST 자료구조 – C 예




```
struct tokenType {
   int tokenNumber;
   char * tokenValue;
}

typedef struct nodeType{
   struct tokenType token; // 의미 있는 token 만 다룸
   struct nodeType * son;
   struct nodeType * brother;
}
// n-ary tree를 binary tree로 나타냄…. 자료구조책 참고
```

S→ aAb
A→ aS | b

파싱 단계에서 AST 만들기 – LL

Recursive descent parser에서non terminal 프로시져를 변형

```
void pS() {
     if (nextSymbol == ta) {
               pa(); pA(); pb();
     }
}
```

```
Node_S pS() {
    if (nextSymbol == ta) {
               Node_a x1 = pa();
               Node_A x2 = pA();
               Node_b x3 = pb();
               return new Node_S(x1,x2,x3);
    } else return null;
}
```

파싱 단계에서 AST 만들기 – LR

- Shift a : 의미있는 terminal일 경우 단말 노드를 만듬
- Reduce $A \rightarrow X_1X_2X_3$ ... :
  - 의미 있는 생성 규칙일 때 : 지금까지 만들었던 노드들을 한데 묶어 subtree를 구성
    - in C
      - 형제 노드를 연결하고 ($X_1X_2X_3$ ...)
      - 새 노드를 만들어 형제 노드를 자식 노드로 삼는다.
  - 의미 있는 생성 규칙이 아닐 때 : skip
    - in C (과제할 때): 새 노드를 만들지 않고 형제 노드만 연결한다.

노드 만들기-예 (LR)

$	1 + 2 + 3$	shift num
$1	+2 + 3$	reduce E→num
$E	+2 + 3 $	shift +
$E+	2+3 $	shift num
$E+2	+3$	reduce E→num
$E+E	+3$	shift +
$E+E+	3$	shift num
$E+E+3	$	reduce E→num
$E+E+E	$	reduce S→E
$E+E+S	$	reduce S→E+S
$E+S	$	reduce S→E+S
$S	$	acc

num node 만들기

num node 만들기

num node 만들기

2+3 node 만들기

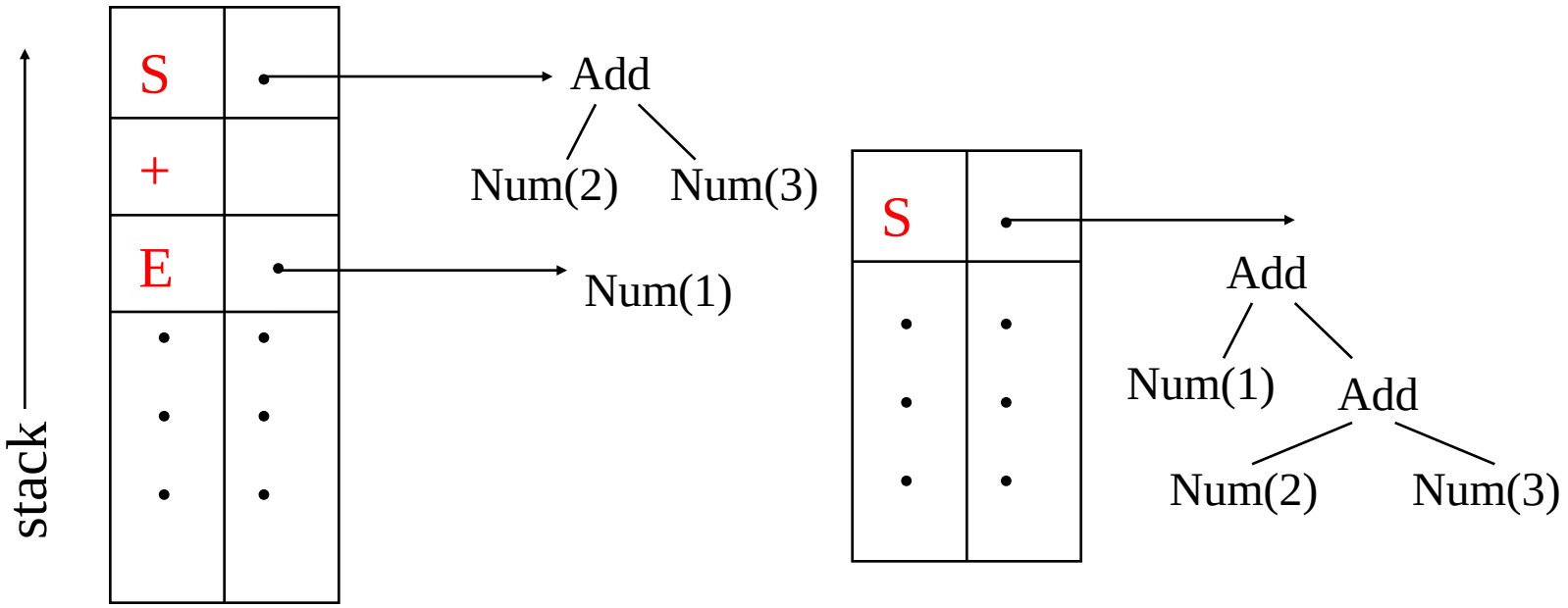
1+(2+3) node 만들기

Note: reduce S→E는 노드를 만들지 않는다

파싱 단계에서 AST 만들기 – LR (예)

S → E + S | E
E → num | (S)

input string: "1 + 2 + 3"



Before reduction: S → E + S

After reduction: S → E + S

in Java

Syntax Directed Definition

Syntax-Directed Definition (SDD)

- **AST**
  - 파싱 단계에서 의미 있는 일을 함
    - LL : production rule derivation 할 때
    - LR : reduce 할 때
- **Syntax-Directed Definition**
  - AST처럼 파싱 단계에서 의미 있는 일을 하는데..
  - 여러 다양한 일을 할 수 있도록, 이걸 좀 체계적으로 확장된 것으로 볼 수 있음
    - production 규칙마다 '할 일(!)'을 적어 놓음
      - 할일 = an associated semantic action (code)
    - 해당 production rule이 reduce 될 때 (LR 경우), derive 될 때 (LL 경우) '할 일'이 수행됨
  - 파서 생성기에서 많이 사용
    - 생성된 파서에 삽입될 semantic action을 사용자가 정의가능

Semantic Actions

- Action은 파서의 스택을 접근하는 코드
  - 스택 entry로, terminal, nonterminal, 파서 상태 이런 것들 말고도 좀 추가해야함.
  - 앞서 LR에서 AST 만들 때 스택에 트리 넣은 작업도 이것의 일종

- E → E + E 규칙에 대한 action
  - 세 E 가 구별이 되야 action code를 쓰므로...
  - Yacc/Bison에서는 $$, $1, $2, ...

    eg. `expr : expr PLUS expr {$$ = $1 + $3;}`
  - ANTLR 에서는 $<*name*>

    eg. `expr : unary_expr {$expr = $unary_expr;}`

- **SDD 예1) AST를 Yacc의 SDD 스타일로 만들려면**

```
%union{
   Expr node;
   int val;
}
%token    <val>  NUM
%type     <node> expr
…
expr : NUM          {$$.node = new Num($1.y); }
    | expr PLUS expr
                    {$$.node = new Add($1.node,$3.node); }
    | expr MULT expr
                    {$$.node = new Mul($1.node,$3.node); }
    | LPAR expr RPAR
                    {$$.node = $2.node; }
    ;
```

Class Problem

아래 문법에서 각 단계별로 계산된 값을 구하여 $$.val에 넣도록 밑줄 친 semantic action 부분을 수정하시오. 단 대문자 NUM, PLUS, MULT, LPAR, RPAR은 토큰이다.

Lex에서

```
…
[0-9]+                  { yylval.val = atoi(yytext); return NUM;}
…
```

Yacc에서

```
%union{
    int val;
}
%token <val> NUM
%type <val> expr
…
```

```
expr : NUM                  {_____________________}
expr : expr PLUS expr       {$$.val = $1.val + $3.val;}
expr : expr MULT expr       {_____________________}
expr :  LPAR expr RPAR      {_____________________}
```

- **SDD 예2) AST를 ANTLR의 SDD 스타일로 만들려면**

```
expr returns [Expr node]:
        NUM          {node = new Num($NUM.int);}
      | ex1=expr PLUS ex2=expr
                     {node=new Add(ex1.node, ex2.node);}
      | ex1=expr MULT ex2=expr
                     {node=new Mul(ex1.node, ex2.node);}
      | LPAR ex1=expr RPAR {node=ex1.node);}
    ;
```

Note:

```
file returns [List<List<String>> data]
    @init {data = new ArrayList();}
    :  (row {data.add($row.list);})+ EOF
    ;
```

Listener Style in ANTLR

```
// Generated from MiniC.g4 by ANTLR 4.5.1

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

/**
 * This class provides an empty implementation of {@link MiniC
 * which can be extended to create a listener which only needs
 * of the available methods.
 */
public class MiniCBaseListener implements MiniCListener {
        /**
         * {@inheritDoc}
         *
         * <p>The default implementation does nothing.</p>
         */
        @Override public void enterProgram(MiniCParser.ProgramContext ctx) { }
        /**
         * {@inheritDoc}
         *
         * <p>The default implementation does nothing.</p>
         */
        @Override public void exitProgram(MiniCParser.ProgramContext ctx) { }
        /**
         * {@inheritDoc}
         *
         * <p>The default implementation does nothing.</p>
         */
```

각 메소드를 채우면 해당 노드를 시작할 때 할 일을 명시할 수 있다.

문법 이름이 MiniC면 인터페이스인 MiniCListener와 껍데기 클래스인 MiniCBaseListener가 자동 생성된다.

Nonterminal마다 entry, exit 메소드가 자동 생성

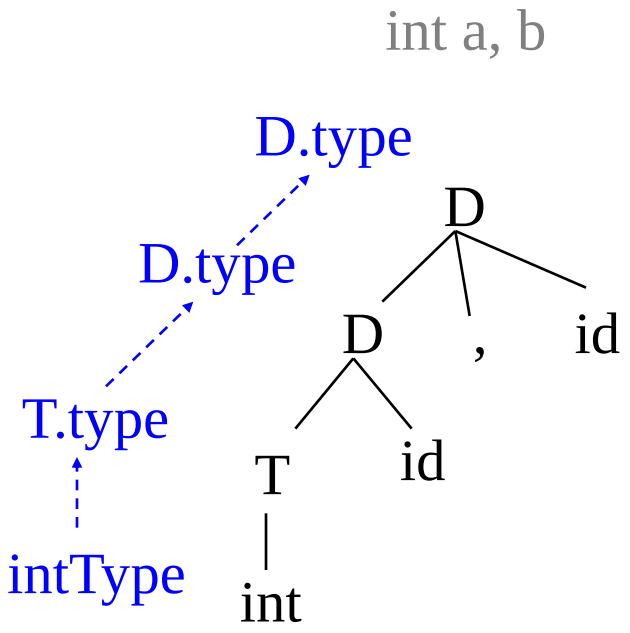
트리를 순회할 때 Nonterminal의 해당 노드 방문 시 entry, exit 메소드가 자동 호출됨

SDD 예3-1) Type Declaration

{AddType($2, $1.type);
$$.type = $1.type; }

in yacc

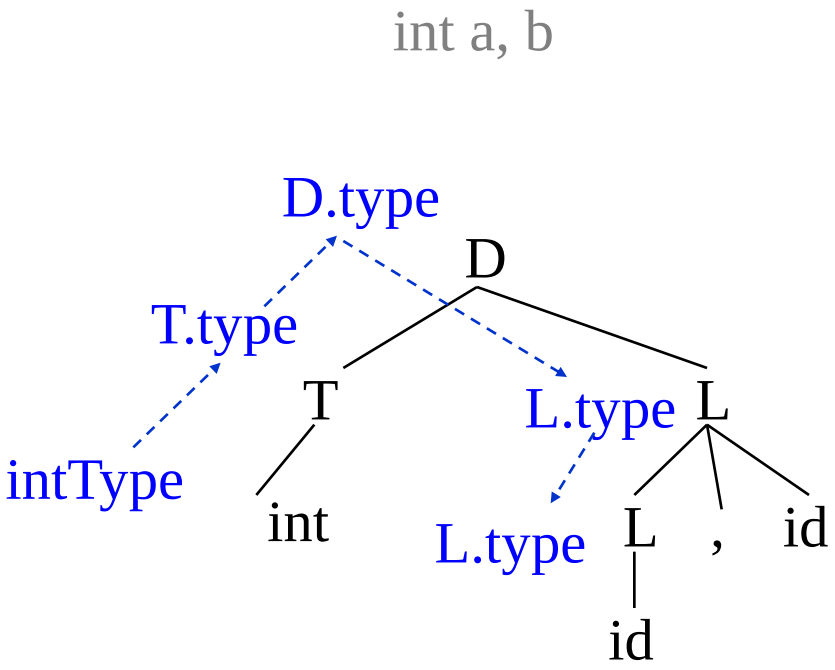
D → T id	{AddType(id, T.type); D.type = T.type; }
D → D1, id	{AddType(id, D1.type); D.type = D1.type; }
T → int	{T.type = intType; }
T → float	{T.type = floatType; }



값이 bottom-up으로 propagation

SDD 예3-2) Type Declaration

D → TL	{AddType(id, T.type); D.type = T.type; L.type = D.type; }
T → int	{T.type = intType; }
T → float	{T.type = floatType; }
L → L1, id	{AddType(id, L1.type); ??? }
L → id	{AddType(id, ???); }



값이 bottom-up/top-down 둘 다로 propagation!!

속성 (Attributes)

- AST vs. SDD
  - AST가 SDD에 의해 정의되는 것으로 보기도 하지만,
  - SDD를 AST의 각 노드에 속성을 붙여 evaluation 해가는 것으로 보기도 함.

- 속성의 종류 (A→XYZ 에서)
  - **synthesized attr**.
    - children에 의해 계산 (bottom-up)
    - A.attr = f(X.attr, Y.attr, Z.attr);
    - terminal은 synthesized attr. 뿐
  - **inherited attr**.
    - parent, sibling에 의해 계산
    - Y.attr = f(A.attr, X.attr, Z.attr);

Attribute Evaluation

- Parse tree method
  - AST 만드는 순서를 속성값 계산 순서대로 조정
    - 속성값 계산 순서 : 속성간에 계산 dependency graph를 만들고 topological sort 를 하면 나옴
  - dependency graph의 cycle 등 있으면 fail위험
- Rules based
  - 각 production 마다 attr. evaluation 순서를 미리 정의
- On-the-fly
  - node 방문 순서대로 그냥 따라 evaluation (e.g., LL 일 때 topdown, LR 일때 bottom-up)
  - 가장 efficient, but restriction이 있음 (다음 경우만...LR 관점)
    - S-attributed SDD : synthesized attr. 만 가지고 있던지,
    - L-attributed SDD : synthesized attr. 만 가지거나, 아니면 값이 왼쪽에서 오른쪽으로 흘러 계산이 이루어지는 경우

Semantic Analysis

- 의미분석 (Semantic analysis)
  - 프로그램 constructs (변수, 객체, 식, 문장...) 의 올바른 사용 여부를 분석, 확인
    - Scope 관련 : 변수가 선언되기 전에 쓰였나? 두 번 정의됐나?
    - 타입 관련 : 변수와 assign되는 값과 서로 타입이 맞는가?
  - 의미분석 ≅"속성 계산하고 값 check"
- Single Pass analysis
  - 기존AST에 첨부된 속성 계산하고 값 check
    - mixed with parser code
  - 장점 – 효율, 단점-AST 바꾸고 싶을 때 checking 도 바꿔야
- Multi Pass analysis
  - (code 생성이 아닌) Semantic checking을 위한 별도의 AST를 만들고, 이 tree를 traverse 하면 check
  - trusted

Class Problem

다음에서 오류가 발생된다면
어휘분석, 구문분석, 의미분석 중 어디쯤 속하는지 생각해보시오.

```
int a;
a = 1.0;
```

```
int a; b
b = a;
```

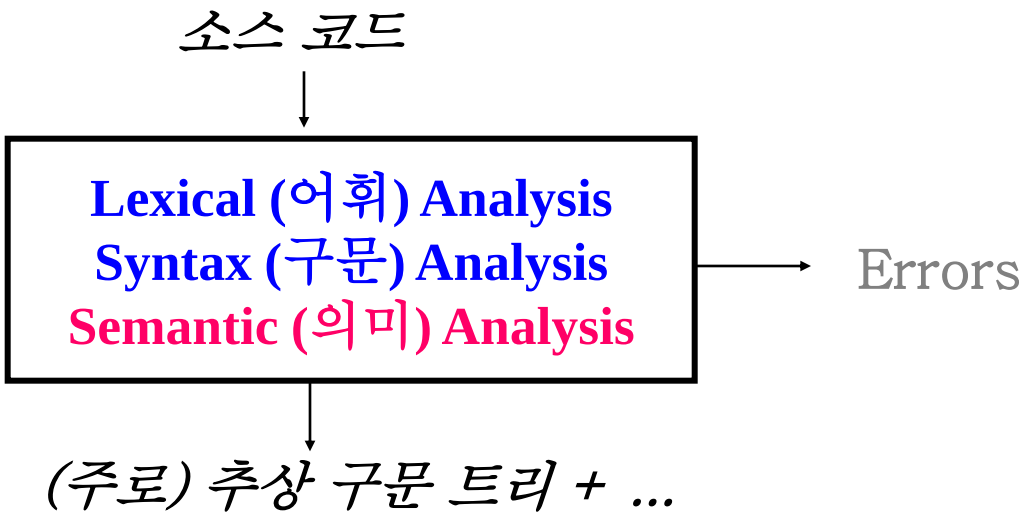
```
{ int a;
   a = 1;
}
{ a = 2;
}
```

```
in a;
a = 1;
```

```
int foo(int a)
{
 foo = 3;
}
```

Semantic Analysis

Semantic Analysis



- 프로그램 constructs (변수, 객체, 식, 문장...) 의 올바른 사용 여부를 분석, 확인
- Scope 관련 : 변수가 선언되기 전에 쓰였나? 두 번 정의됐나?
- Type 관련 : 변수와 assign되는 값과 타입이 맞는가?

Scope

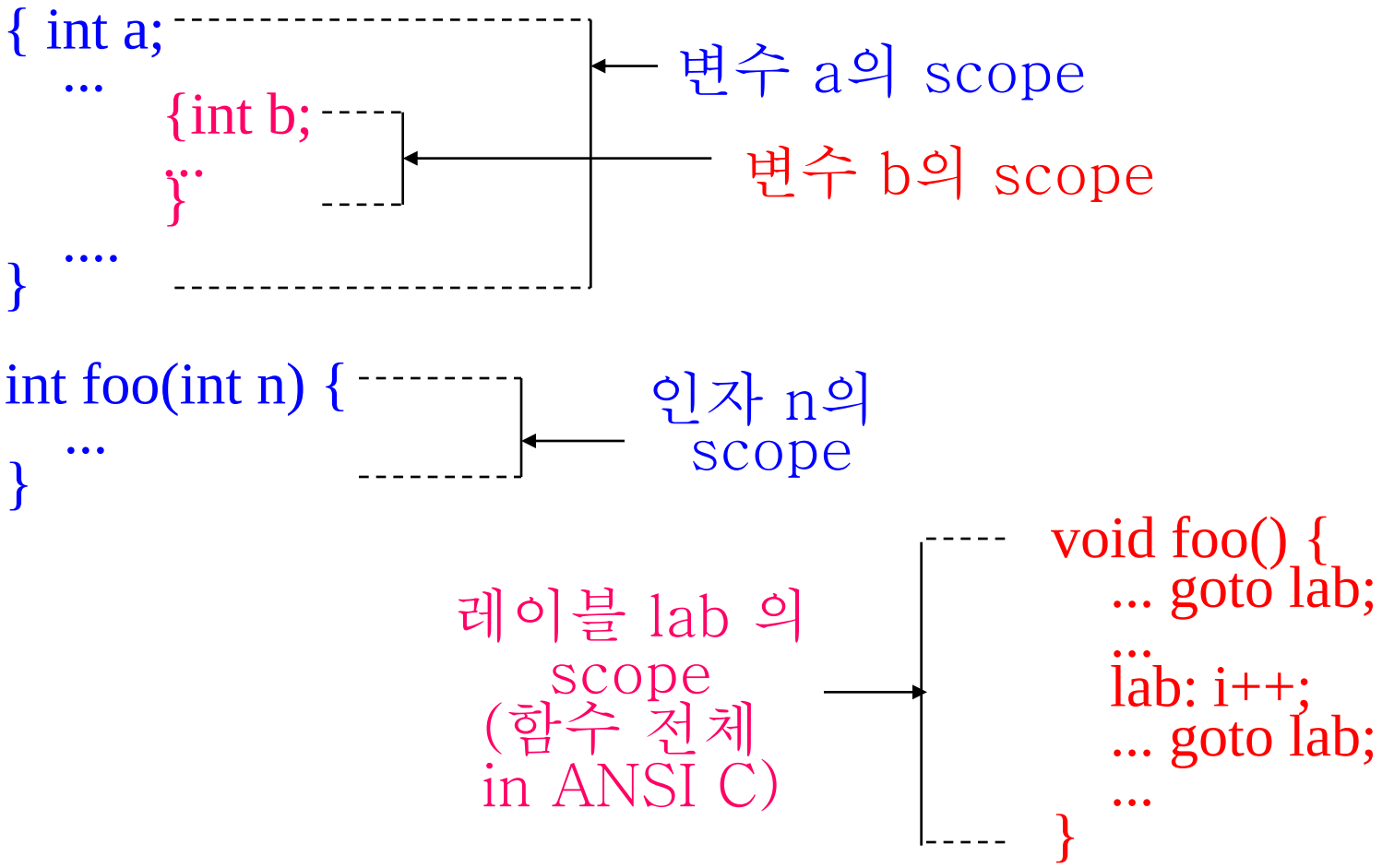
- 식별자 (Identifiers)
  - 변수, 상수, 함수이름, labels ...
- Lexical Scope
  - 프로그램을 문자적으로(textual) 볼 때 특정 범위
    - 문장 block, 형식인자 리스트, 함수 정의문, 소스 파일, 프로그램 전체...
- 식별자의 scope
  - 그 선언이 참조되는 lexical scope

예) 변수의 scope:

문장 block 내 (local 변수), 함수 내 (형식 인자), 소스 파일 (global 변수), 전체 프로그램 (extern 변수)

cf. How about fields? methods?

변수 Scope : PL 복습



Semantic Rules for Scopes

- Main rules:
  - Rule 1: 각 식별자는 자신의 scope 에서 한번만 선언되어야함
  - Rule 2: 같은 종류의 식별자를 한 lexical scope 내에서 두 번이상 선언하면 안됨

```
class X {
    int X;
    void X(int X) {
        X: ...
        goto X;
    }
}
```

```
int X(int X) {
    int X;
    goto X;
    {
        int X;
        X: X = 1;
    }
}
```

문제가 있을까?
있다면,
어디서 발생하나?

Symbol Tables

- Symbol tables
  - 심벌들에 관한 정보를 관리하는데 사용되는 자료구조
  - 의미 분석단계에서 식별자의 성질 (scope, type)을 참조하거나 코드 생성할 때 주로 사용 됨
  - 선언될 때 정보 수집해서 테이블에 삽입(insert)
  - 사용되는 부분에서 테이블 참조 (get)
- 테이블 entry : 식별자 이름 + info
  - 예)

NAME	KIND	TYPE	ATTRIBUTES
foo	func	int,int → int	extern
m	arg	int	
n	arg	int	const
tmp	var	char	const

Scope Information in Symbol Tables

```
int x;
void f(int m) {
    float x, y;
    ...
    {int i, j; ....; }
    {int x; l: ...; }
}

int g(int n) {
    char t;
    ... ;
}
```

- Block structured 언어 특징
  - 각 block (lexical scope) 내에는 local 변수 선언들 존재

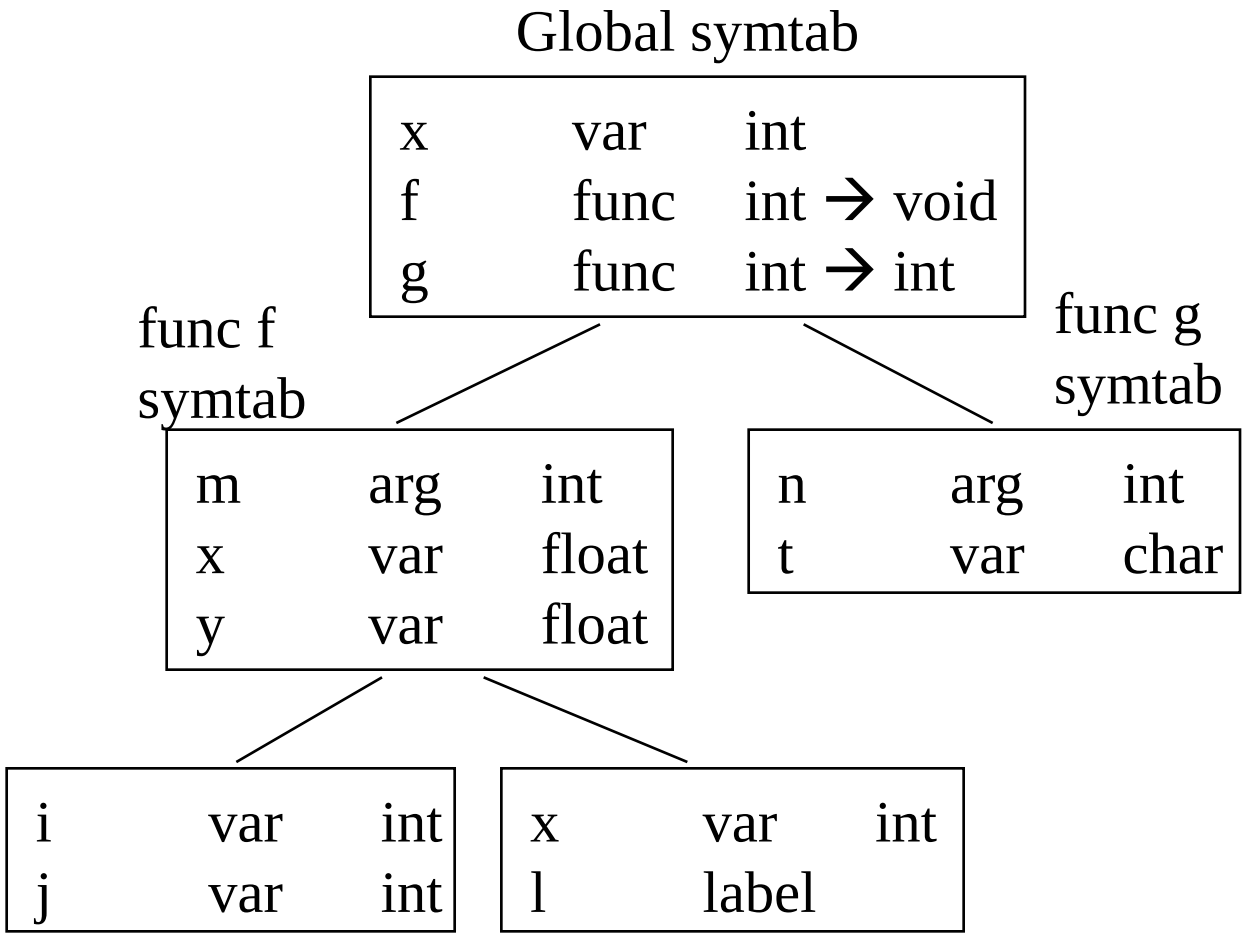
➔ 한 lexical 마다 한 symbol table

  - Hierarchy of scopes :
    - 각 block (lexical scope)은 다른 subblock을 포함가능하고,
    - 자기 block 에서 선언되지 않고, 자신을 포함하는 block 에서 선언된 변수 사용 가능

➔ Hierarchy of symbol tables

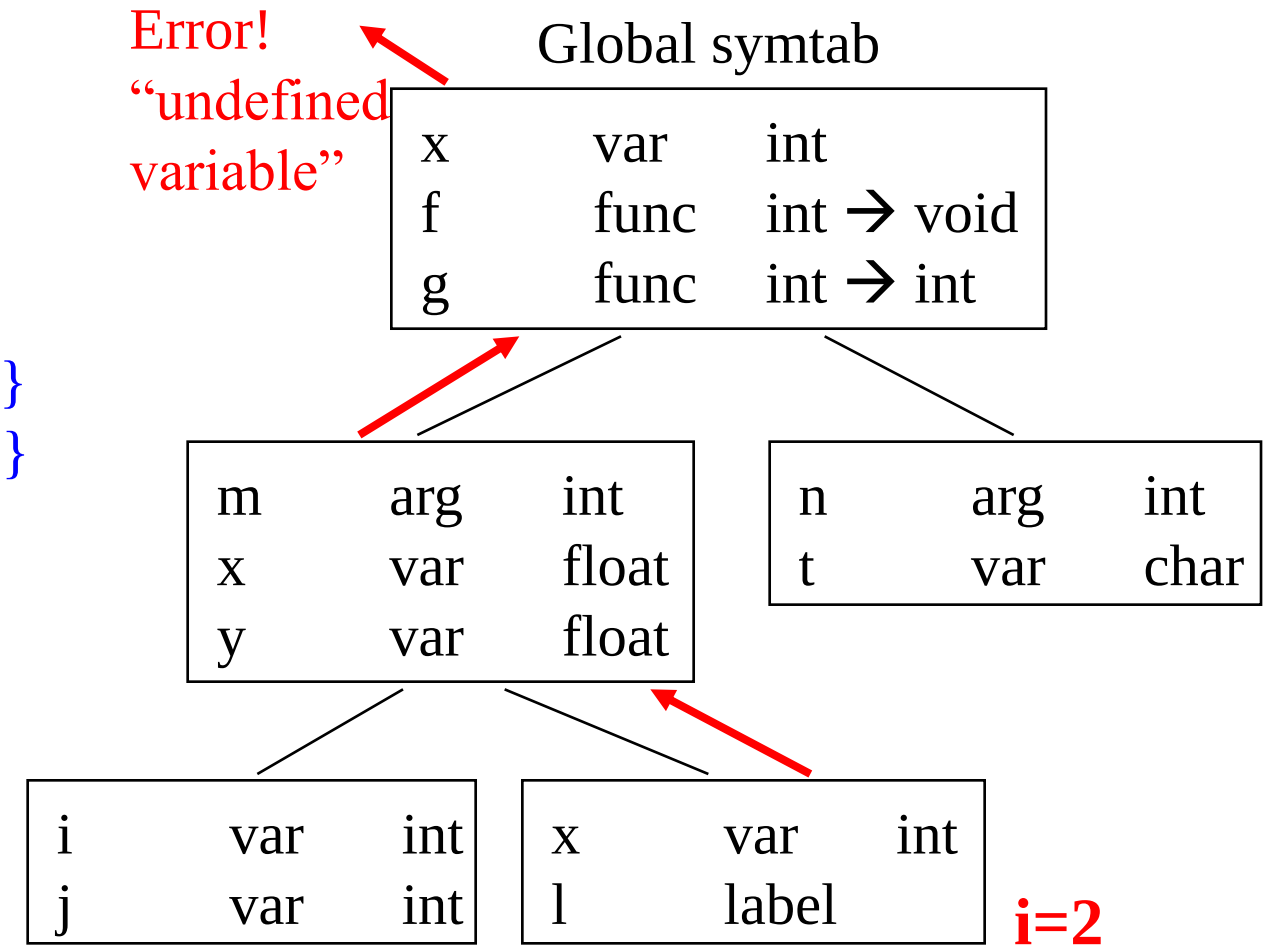
Examples

```
int x;

void f(int m) {
    float x, y;
    ...
    {int i, j; ....; }
    {int x; l: ...; }
}

int g(int n) {
    char t;
    ... ;
}
```



오류 Checking

```
int x;
void f(int m) {
    float x, y;
    ...
    {int i, j; x=1; }
    {int x; l: i=2; }
}

int g(int n) {
    char t;
    x=3;
}
```

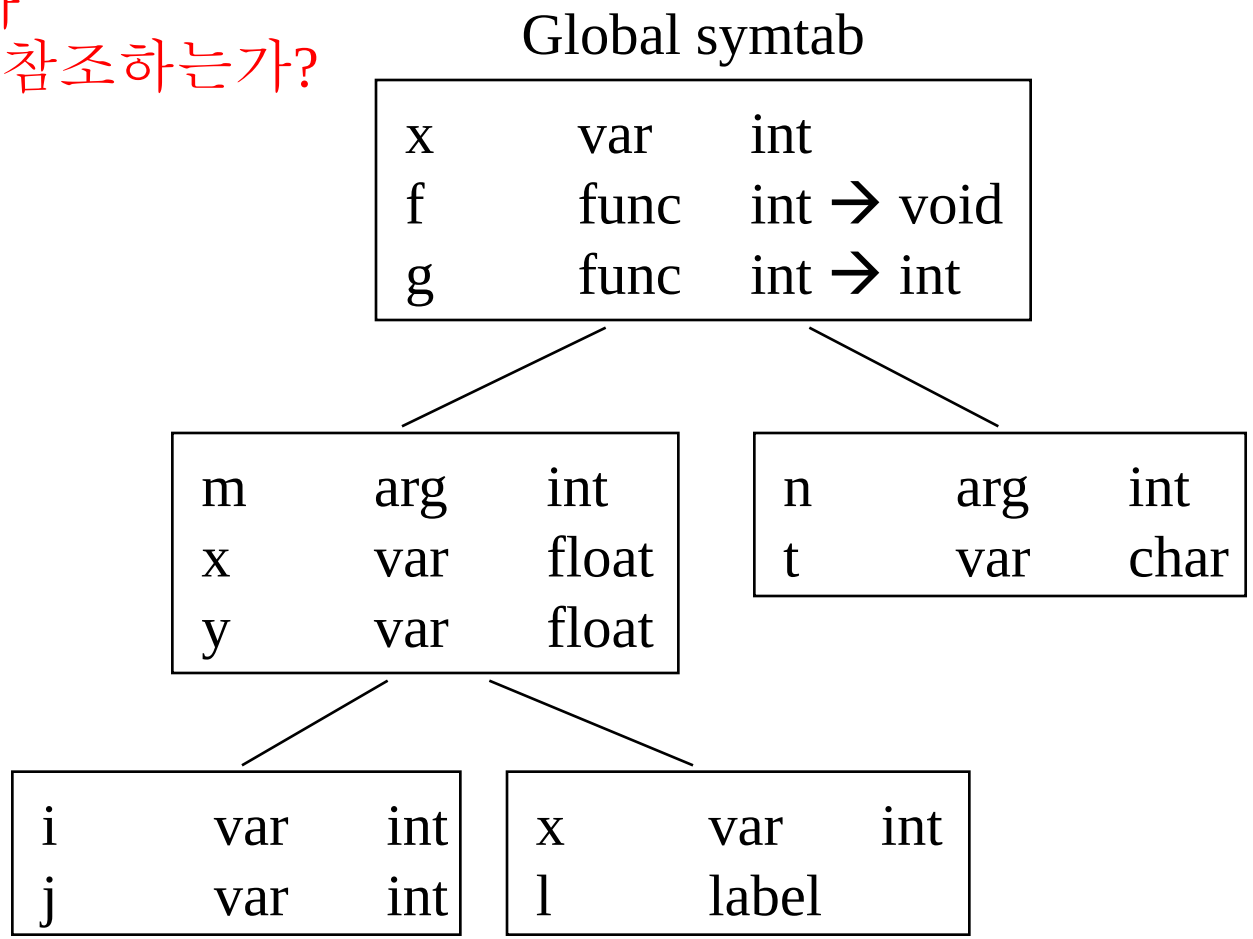


- 현재 scope 에서 시작해서 hierarchy 위쪽으로 올라가면서 찾음
- 끝까지 없으면 에러

Class Problem

각 x 의 assignment 가
어느 symbol table을 참조하는가?

```
int x;
void f(int m) {
    float x, y;
    ...
    {int i, j; x=1; }
    {int x; l: x=2; }
}

int g(int n) {
    char t;
    x=3;
}
```



Symbol Table Implementation

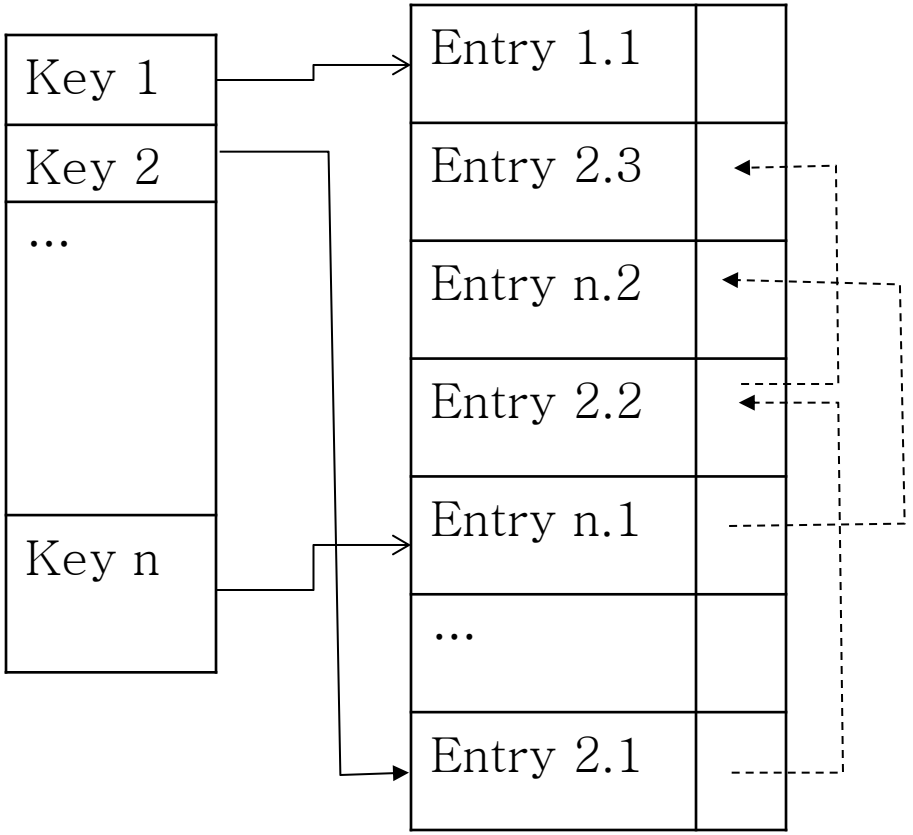
- 필요한 operation
  - 테이블 구성 : AST 까지 만들어져야 가능
  - 삽입 (insertion) : 해당 식별자가 선언될 때
  - 검색 (lookup) : 해당 식별자가 사용될 때 (checking)

cf. forward reference 는?

- Efficiency를 위해
  - 테이블 entry에서 식별자 이름
    - 이름들의 스트링 값을 따로 보관하는 스트링 풀로의 포인터
  - Local 테이블 구조 : hash 테이블 사용
  - Global하게는 N-ary tree 구조 :
    - 트리를 쓰지는 않는다. : 포인터 유지등의 비용 때문
    - 사용의 locality : scope을 빠져 나가면 해당 local table은 필요가 없어짐

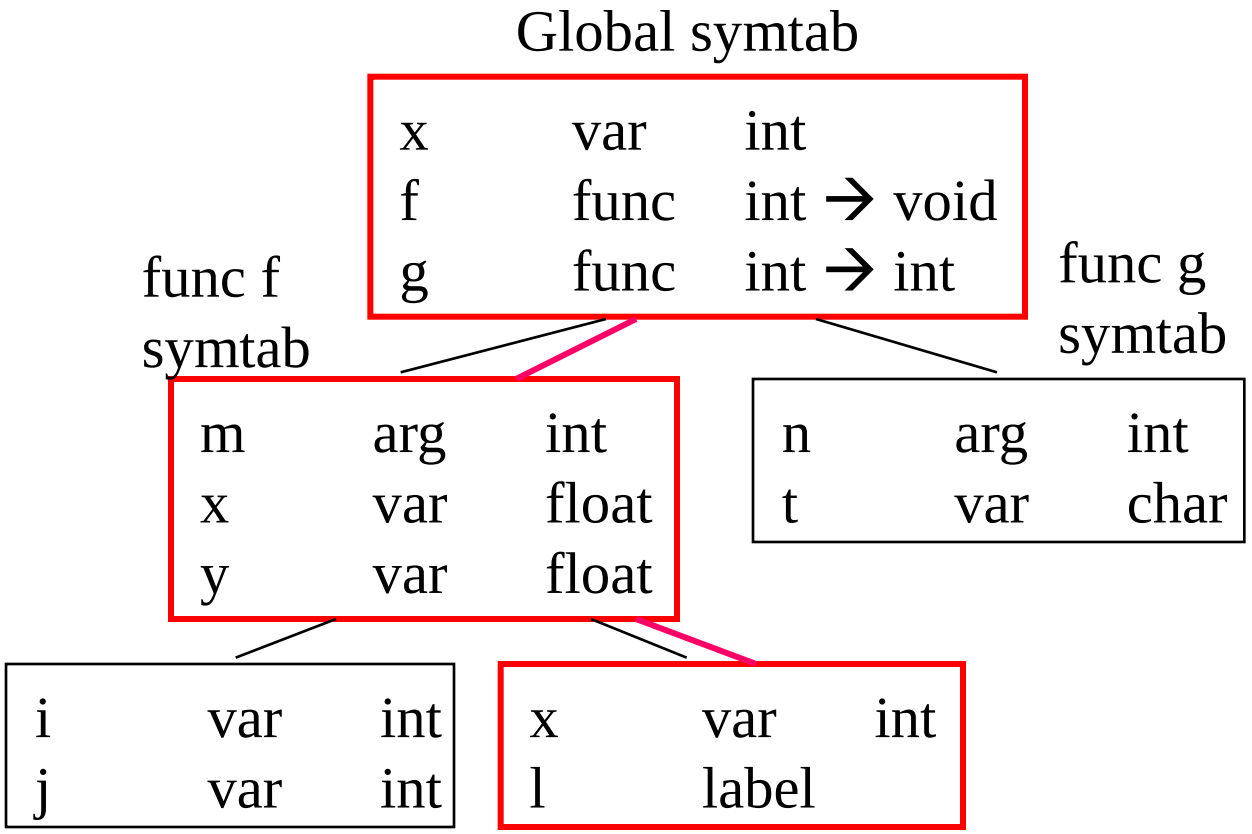
Local Table

- Local Hash Table 만 생각하면...



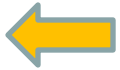
Global Table Hierarchy–스택을 사용

```
int x;

void f(int m) {
    float x, y;
    ...
    {int i, j; ....; }
    {int x; l: ...; }
}

int g(int n) {
    char t;
    ... ;
}
```

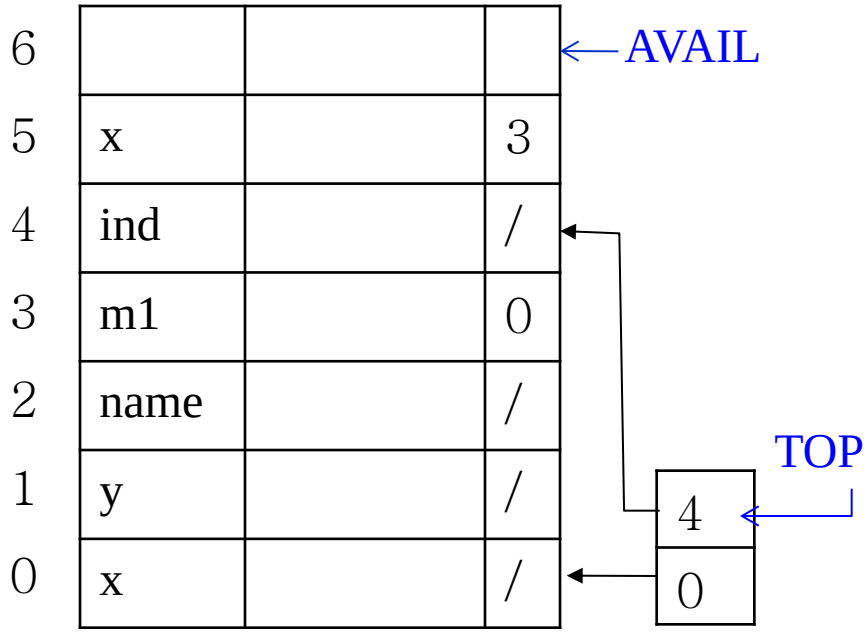


Hierarchies of Local Tables

```
int x,y;
char name;
void m1(int ind) {
    int x;
}
void m2 (int j) {
  {
      int f[j];
      char test;
   }
}
```



0	/
1	5
2	2
3	/
4	4
5	/
6	/
7	1
8	/
9	/
10	/

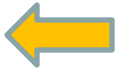


- x, m1은 1
- name은 2
- ind, j는 4
- test는 6
- f, y는 7
- m2는 10

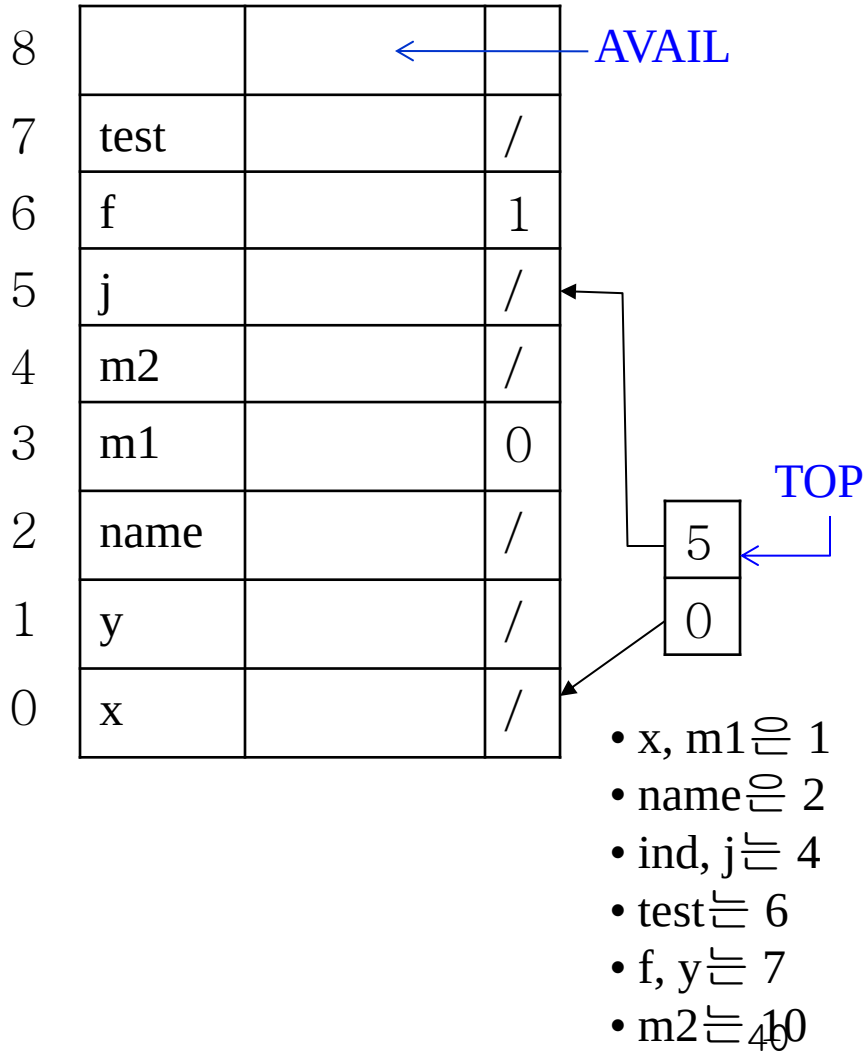
"스택과 해쉬의 공존"

Hierarchies of Local Tables (cont')

```
int x,y;
char name;
void m1(int ind) {
    int x;
}
void m2 (int j) {
  {
     int f[j];
     char test;
   }
}
```



0	/
1	3
2	2
3	/
4	5
5	/
6	7
7	6
8	/
9	/
10	4



Back to Type Checking

- 타입이란?
  - 수행 중 가지게 되는 '값'에 대한 기술
  - Predicate이기도 함.
    (예: C에서 "int x" 는 -2^31 <= x < 2^31 라는 뜻)
- 타입 오류 : 값의 부적절한 사용
- 타입 안전성:
  - "타입 오류가 없음"
  - 보장 방법
    - 타입을 선언하고 (=binding)
      - 명시적방법 (int x) vs. 암묵적 방법 (x=1;)
    - 이를 검사함 (= checking) 으로써 보장
      - 타입 규칙을 세우고 이에 대해 검사함

타입과 관련된 몇 가지 개념들

1. Static vs. dynamic checking
   - 타입 검사 시점 : 컴파일 시 vs. 수행시
2. Static vs. dynamic typing
   - 타입 정의 시점 : 컴파일전 vs. 수행시
3. Strong vs. weak typing
   - 타입 오류 : 엄격히 방지 vs. 조금 허용
4. Sound type systems
   - 모든 타입 오류를 방지 하는 타입 규칙 집합 및 enforce 시스템

타입 표현식 (Type Expressions)

- 언어에서 제공하는 기본 타입
  - int, float, char …
- 타입 표현식 (type expressions)
  - 기본 타입 및 이들로부터 작성되는 의미있는 타입들
  - Array types : T[], T[10], T[1..10], T[2][3]
    - 각 원소 접근, 지정 시 타입 유의
  - Structure types : $\{id_1: T_1, \cdots, id_n: T_n\}$
    - 필드 접근, 지정 시 타입 유의
  - Pointer types : T *
    - 내용 접근, 지정 시 타입 유의
  - Function types : $T_1 \times T_2 \times \cdots \times T_n \rightarrow T_{return}$
    - 호출 시 인자와 리턴 값 타입 유의

타입 규칙 정의

- 타입 규칙들
  - int + float 는 float, float f(int x) 에 인자로 3을 주면 float타입 결과 리턴, … OO는 subtype까지!
- 타입 judgment (= static semantics)
  - 타입 규칙을 formal하게 정의함

E : T *"E는 T 타입을 만족한다."*

- 2 : int
- true : bool
- 2 * (3 + 4) : int
- "Hello" : string
- if (b) 2 else 3 : int
- x == 10 : bool
- y = 2 : int 또는 y=2 : unit // unit 는 well typed 임을 표현

(어휘분석에는 regular expression, 구문 분석에는 CFG, 타입 분석 (의미 분석)에는 타입 judgment)

Class Problem

- 다음 문장의 type 은?

  단 i로 시작하는 식별자는 int 변수거나 int 타입을 원소로 하는 배열이고, f로 시작하는 식별자는 float 변수거나 float 타입을 원소로 하는 배열이라고 가정한다. S1은 타입이 오류가 없는 문장이다.

  - f1 [3]
  - i = i1 [i2]
  - while (i < 10) do S1
  - (i == 0 ? 4.0 : 1.0)

타입 Judgment 유도

- Consider if (b) 2 else 3 : int
- 이것을 위해 알아야할 것들
  - b 는 bool (b : bool)
  - 2 는 int (2 : int)
  - 3 는 int (3 : int)
- 타입 judgment notation
  - A ⊢ E : T "A 상황에서 E는 T타입을 만족한다"
  - 예)

    b: bool, x: int ⊢ b: bool

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

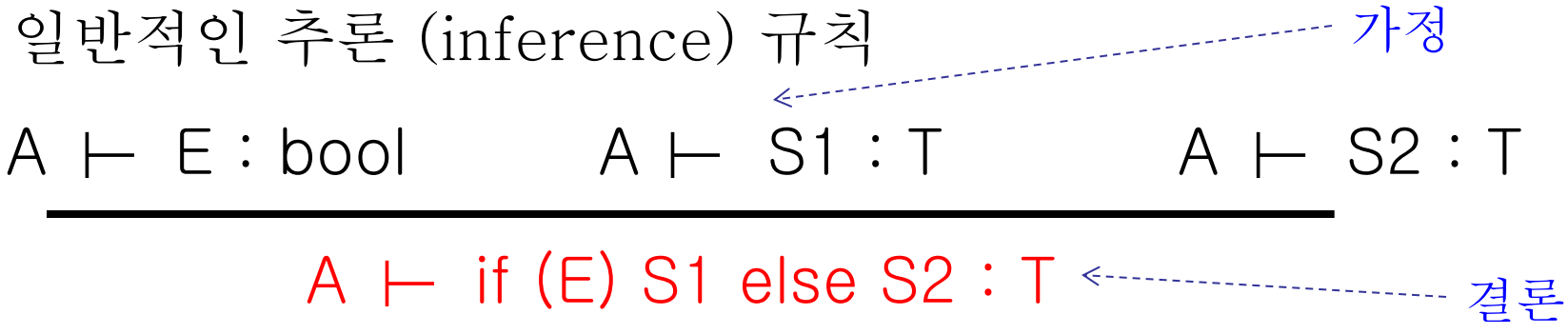
    ⊢ 2 + 2 : int

타입 Judgment 유도

- 이것을 보이기 위해서는
  - b: bool, x: int ⊢ if (b) 2 else x : int

  이것을 먼저 보여야함
  - b: bool, x: int ⊢ b : bool
  - b: bool, x: int ⊢ 2 : int
  - b: bool, x: int ⊢ x : int

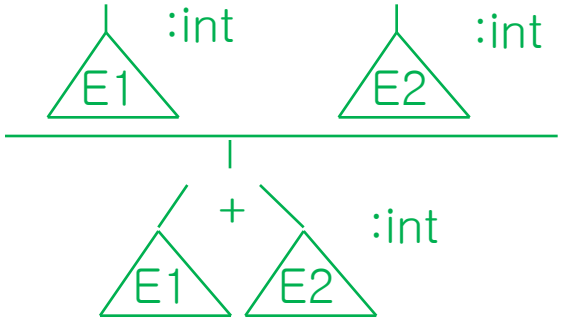
- 일반적인 추론 (inference) 규칙



"모든 E, S1, S2, A, T에 대해 가정이 성립하면 결론이 성립한다."

Proof Tree (타입 유도 트리)

$$\frac{A \vdash E1 : int \quad A \vdash E2 : int}{A \vdash E1 + E2 : int}$$



- 앞서 예에서 S1과 S2가 true 인지도 확인해야함
  - 반복됨, 결국 트리 형태 ➔ proof tree
- 만족하는 proof tree 가 있다 == 타입 오류가 없다
- 예)

$$\frac{\dfrac{A1 \vdash b : bool}{A1 \vdash !b : bool} \quad \dfrac{A1 \vdash 2 : int \qquad A1 \vdash 3 : int}{A1 \vdash 2 + 3 : int} \quad A1 \vdash x : int}{b : bool, x : int \vdash \text{if } (!b)\ 2 + 3 \text{ else } x : int}$$

(단, A1은 b : bool, x : int 를 편의상 짧게 쓴 것)

Class Problem

- 다음과 같은 언어에 대해

  t ::= true
  | false
  | if t then t else t
  | 0
  | succ t
  | pred t
  | iszero t

- 다음 문장의 타입을 트리로 유도하시오.
  (1) if iszero 0 then 0 else pred 0 : int
  (2) pred(succ(iszero(succ(pred(0))))) : int

- 다음과 같은 타입 규칙이 있을 때

  0 : int
  true : bool
  false : bool

  $$\frac{t1: \text{bool} \quad t2: T \quad t3 : T}{\text{if } t1 \text{ then } t2 \text{ else } t3 : T}$$

  $$\frac{t1 : \text{int}}{\text{succ } t1 : \text{int}}$$

  $$\frac{t1 : \text{int}}{\text{pred } t1 : \text{int}}$$

  $$\frac{t1 : \text{int}}{\text{iszero } t1 : \text{bool}}$$

Assignment Statements

$$\frac{id{:}T \in A \qquad A \vdash E : T}{A \vdash id = E : T}$$ (variable-assign)

$$\frac{A \vdash E3 : T \qquad A \vdash E2 : int \qquad A \vdash E1 : array[T]}{A \vdash E1[E2] = E3 : T}$$ (array-assign)

If Statements

- if 는 각 절을 수행한 결과값의 타입

$$\frac{A \vdash E : \text{bool} \quad A \vdash S1 : T \quad A \vdash S2 : T}{A \vdash \text{if (E) S1 else S2} : T} \text{ (if-then-else)}$$

- else 가 없으면 값을 부여하지 않는다

$$\frac{A \vdash E : \text{bool} \quad A \vdash S : T}{A \vdash \text{if (E) S} : \text{unit}} \text{ (if-then)}$$

Class Problem

다음을 위한 추론 규칙을 정의해보시오.

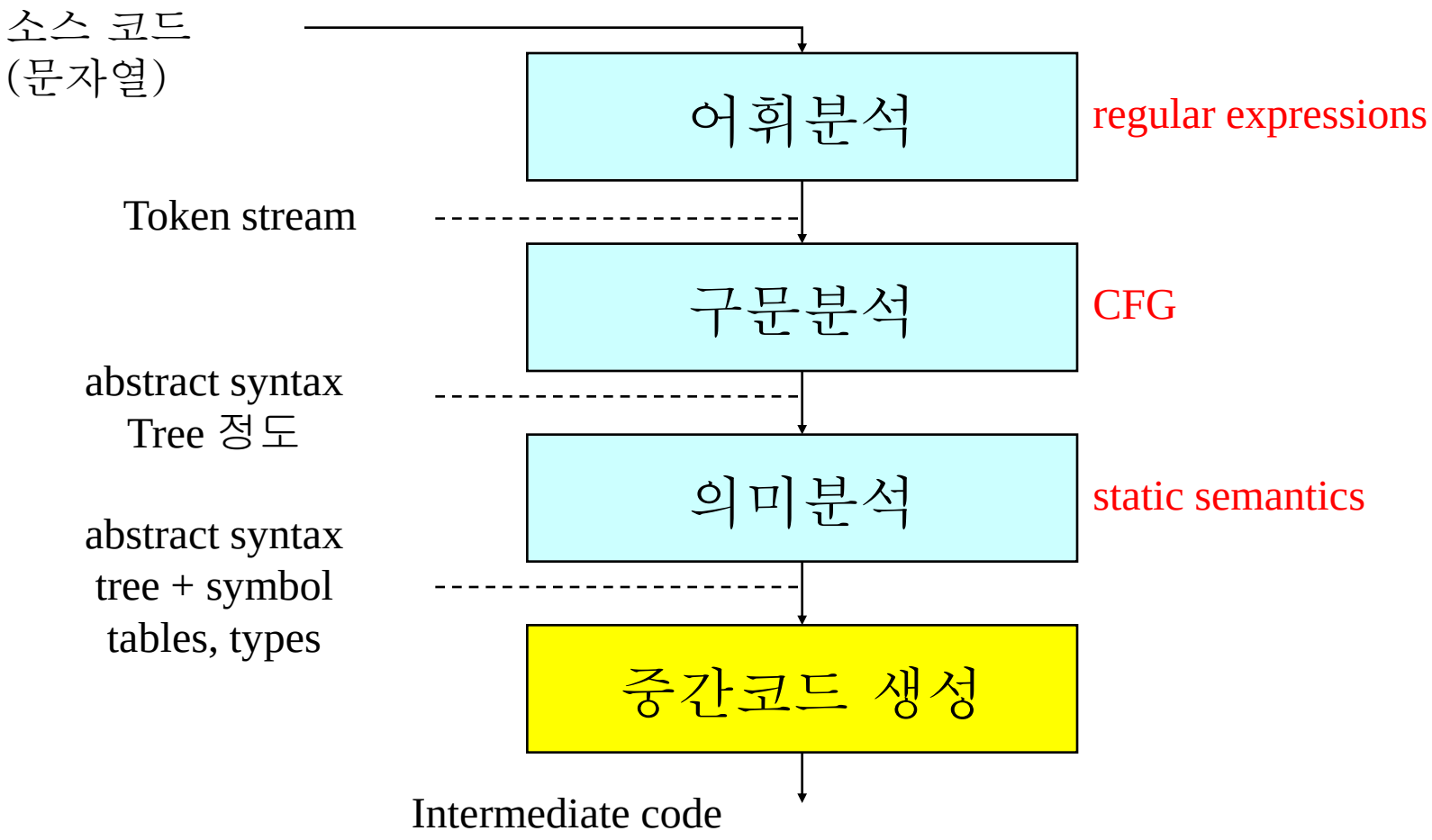
1. while (E) S

2. T id = E

Control Flow 오류

- Scope와 Type check외에 하나 더
- “Control flow errors”
  - break 나 continue 가 while이나, for 속에 들어 있는지.
  - Goto labels 이 해당 함수 안에 있는지 등
  - AST 를 따라가면 어렵지 않게 알 수 있는 간단한 결과들.

Where We Are...


소스 코드
(문자열)
어휘분석
regular expressions
Token stream
구문분석
CFG
abstract syntax
Tree 정도
의미분석
static semantics
abstract syntax
tree + symbol
tables, types
중간코드 생성
Intermediate code