Chapter 6
Semantic Analysis – Part 3

한양대학교 컴퓨터공학부

컴파일러

2014년 2학기

Scope Rules and Block Structure

- Declaration before use
  - C and Pascal
  - 파싱 과정에서 symbol table이 생성되고
  - name reference가 있으면 symbol table을 lookup
    - if lookup fails, a violation of declaration

scope rule
1. declaration before use
2. the most closely nested rule

Block Structure

- common property of modern languages
- any construct that can contain declarations
  - C: compilation units, procedure/function declarations,
    code file
    compound statements, struct/union
- a language is **block structured** if nesting of blocks permitted and if the scope of declarations in a block are limited to that block and the other blocks contained in that block.
  - **most closely nested rule**

Figure 6.14 nested scopes

```
int i, j

int f(int size)
{ char i, temp;
   ...
   { double j;
      ...
   }
   ...
   { char *j;
      ...
   }
}
```

five block
1. entire code
2. declaration of f itself
3. compound statement of the body of f
4. compound statement
5. compound statement

Figure 6.15 nest scopes (Pascal)

```
program Ex;
var i, j: integer;

function f(size: integer): integer;
var i, temp: char;

  procedure g;
  var j: real;
  begin
  ...
  end;
  procedure h;
  var j: ^char;
  begin
  ...
  end;
```

<- procedure&functions can be nested

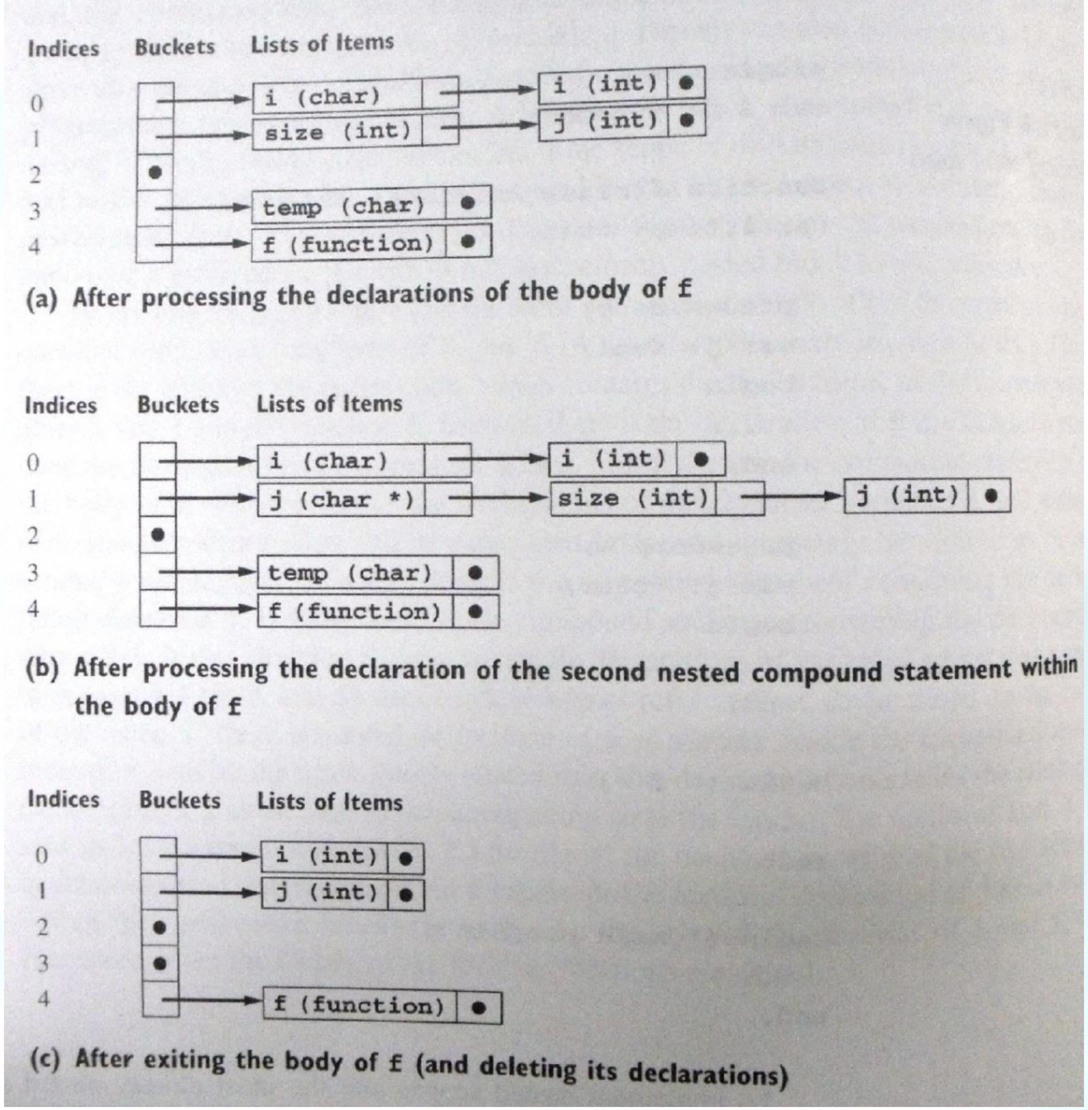
```
begin (* f *)
  ...
end;

begin (* main *)
  ...
end.
```

nested scopes

- symbol table operations
  - *insert*: must **not** overwrite previous declarations
  - *lookup*: find the most recently inserted declaration
  - *delete*: delete only the most recent one

Figure 6.16

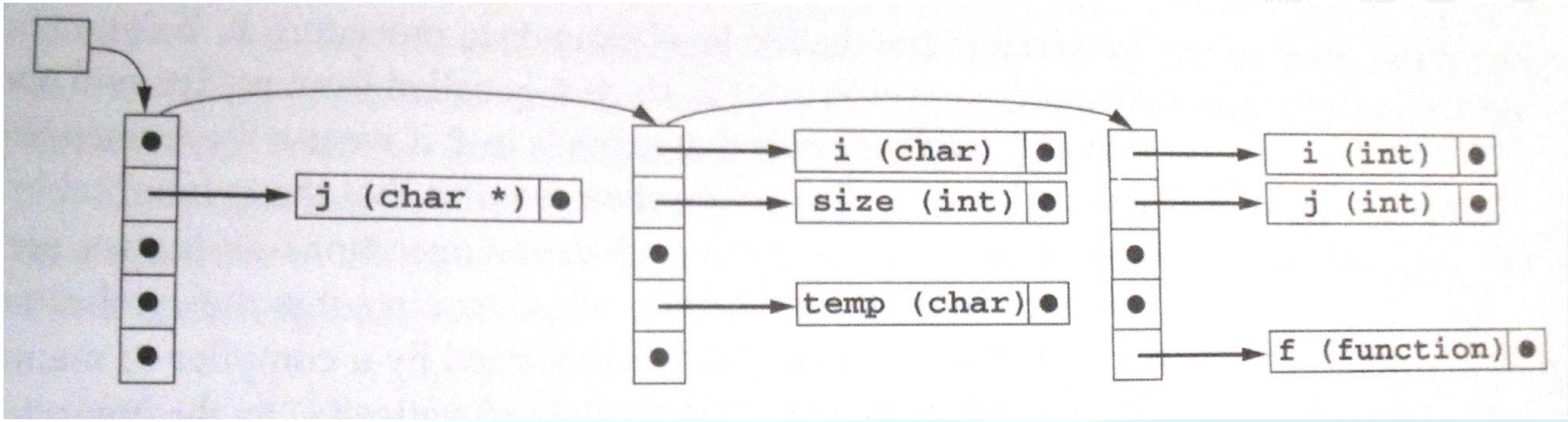


(a) After processing the declarations of the body of f

(b) After processing the declaration of the second nested compound statement within the body of f

(c) After exiting the body of f (and deleting its declarations)

Figure 6.17



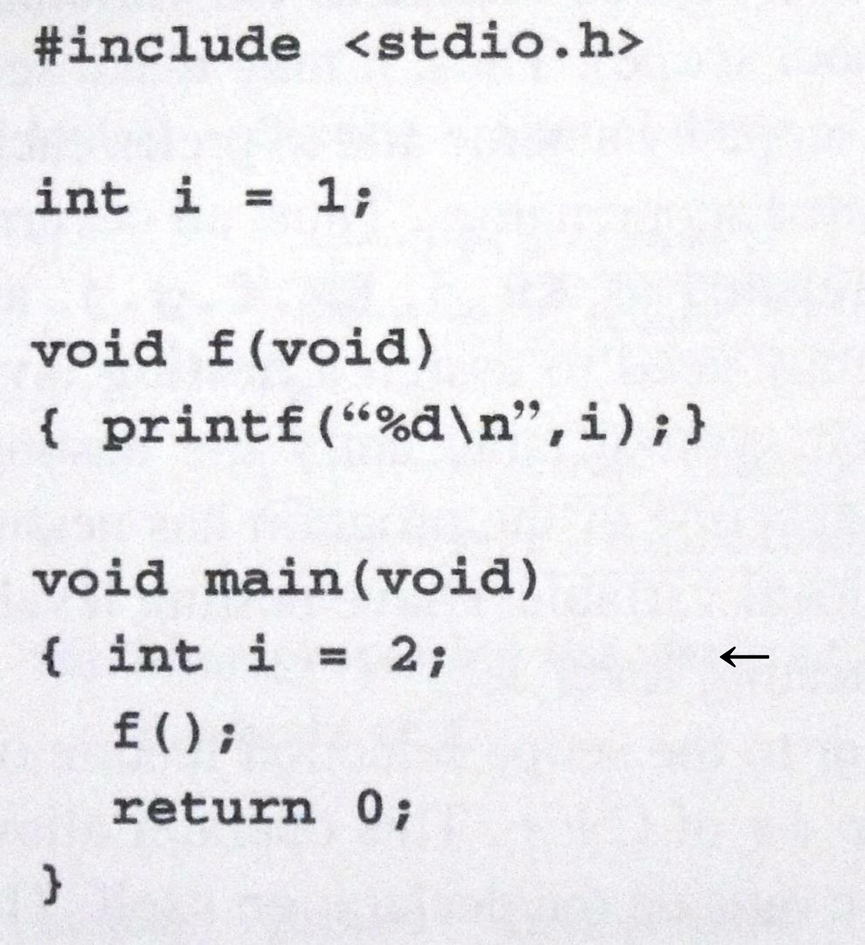
build a new symbol table for each scope and to link the tables from inner to outer scopes together, so that the lookup operation will automatically continue the search with an enclosing table if it fails to find a name in the current table

delete - entire symbol table corresponding to the scope can be released in one step

static (or lexical) vs. dynamic scope

lexical scope or static scope - 1 출력 (c,대부분 language)
 scope rule that follows the textual structure of program
dynamic scope - 2 출력
 scope rule that follows the execution path, rather than the textual layout of the program
 main -> f()

Figure 6.18



```
#include <stdio.h>

int i = 1;

void f(void)
{ printf("%d\n",i);}

void main(void)
{ int i = 2;
  f();
  return 0;
}
```

← what if "double i" ?

Interaction of Same-Level declarations

```
int i = 1;

void f(void)
{ int i = 2, j = i + 1;
   …
}
…
```

- **sequential declaration**: each declaration is added to the symbol table as it is processed j=3 (c에서 지원)
- **collateral declaration**: all declarations processed “simultaneously” and added at once at the end of declaration (in this case, j = 2)
- **recursive declaration**: recursive function calls, mutually recursive functions declarations may refer to themselves or each other
- scope modifier
  - C: function prototype
  - Pascal: **forward** declaration

Extended Example of an Attribute Grammar using a Symbol Table

- Grammar

  S → *exp*
  exp → **(** *exp* **)** | *exp* **+** *exp* | ***id*** | ***num*** | **let** *dec-list* **in** *exp*
  dec-list → *dec-list* **,** *decl* | *decl*
  decl → ***id*** = *exp*

- Examples

```
let x = 2+1, y = 3+4 in x  + y

let x=2, y=x+1 in (let x=x+y, y=x+y in y)

let x = 2, y = 3 in
  ( let x = x + 1, y = (let z = 3 in x+y+z)
     in (x+y)
   )
```

6.4 Data Types and Type Checking

- one of the principal tasks of a compiler
  - **type inference**: computation and maintenance of information on data types
  - **type checking**: to ensure that each part of a program makes sense under the type rules of the language
- data type information
  - static or dynamic, or a mixture of the two
- Pascal, C, Ada → primarily static

data type and type declarations

- data type 연산 +가질 수 있는 값의 범위
  - a set of values with certain operations on those values
  - ex) **integer**: a subset of the math integers, and arithmetic operations, such as + and *
- type or variable declarations
  - explicit
    - type RealArray = array [1..10] of real;
  - implicit
    - const greeting = “Hello!”;

6.4.1 Type Expressions and Type Constructors

- built-in types or simple types
  - int, double, ...
- subrange types and enumerated types
  - type Digit = 0..9;
  - typedef enum {red, green, blue} Color;
- type constructors
  - array, record, struct
  - often called **structured types**

Array

- takes two type parameters
  - index type, component type
    - Pascal: array [*index-type*] of *component-type*
    - index type is limited to ordinal types
- C
  - typedef char Ar[3];
- set of functions $I \longrightarrow C$
  - I : index-type
  - C : component-type
- other issues
  - multidimensioned arrays
  - open-indexed array

Record or Structure

```
struct
{ double r;
  int i;
 }
```

- components of different types
- roughly correspond to the Cartesian product
  - R × I
- implementation: allocate memory sequentially

Union

```
union
{ double r;
  int i;
 }
```

- disjoint union
- implementation: to allocate memory in parallel for each component
- Pascal: discriminant component

```
record case isReal: boolean of
  true: (r: real);
  false: (i: integer);
end;
```

Pointer

- consists of values that are references to values of another type
- frequently thought of as numeric types
    - adding, multiplying
- Pascal: ‘^’ is the pointer type constructor
    - var p: ^integer;
    - p^

Function

- Modula-2 declaration

  VAR f: PROCEDURE (INTEGER) : INTEGER;

  - the set of functions $\{f : I \rightarrow I\}$
  - C: int (*f) (int);
- allocate space according to the address size of the target machine
- may need to be allocated space for a code pointer alone
- or a code pointer and an environment pointer

Class

- used in most object-oriented language
- similar to a record declaration + definition of operations (**methods** or **member functions**)
- class hierarchy
- virtual method table

virtual function - java abstract function과 비슷

Type Equivalence

- Structural Equivalence
  - two types are the same iff they have the same structure
- Name Equivalence

이름이 같으면 타입이 같다고 취급

```
t1 = int;
t2 = int


t3 = record
     x: int;
     t: pointer to t4;
    end;
```

Type Equivalence

- Declaration Equivalence

```
t1 = int;
t2 = int

t1 = array [10] of int;
t2 = array [10] of int;
t3 = t1
```

t1,int - 같은 타입
t2,int - 같은 타입
t1,t2 - 다른 타입
t3,t1 - 같은 타입

- Pascal
- C structures and unions (but structural equivalence for pointers and arrays)

Type Checking

- declarations
  - type of an identifier to be entered into the symbol table
- statements
  - substructures will need to be checked for type correctness
- expressions
  - constant expressions: implicitly defined *integer* and *boolean* types
  - variable names: determined by a *lookup* in the symbol table