Chapter 6
Semantic Analysis
– Part 2 –

한양대학교 컴퓨터공학부

컴파일러

2014년 2학기

Attributes as Parameters and Returned Values

- If many of attribute values are the same or are only used temporarily to compute other attribute values
- Pass the inherited attribute values as parameters to recursive calls on children and receive synthesized attribute values as returned values

Example 6.15

- **Pseudocode**

function *EvalWithBase* (*T*: *treenode; base*: *integer*): *integer*;
var *temp*, *temp2*: *integer*;
begin
  case *nodekind of T* of
  based-num:
    temp:= ***EvalWithBase***(*right child of T, base*);   return base, other return val
    return *EvalWithBase*(*left child of T, temp*);
  num:
    temp := *EvalWithBase(left child of T, base*);
    if *right child of T is not nil* then
      temp2 := *EvalWithBase* (*right child of T*, *base*);
      if *temp* ≠ *error* and *temp2* ≠ *error* then
        return *base***temp* + *temp2*
      else return *error*;
    else return *temp*;
  basechar:
    if *child of T* = o then return 8
    else return 10;
  digit:
    if *base* = 8 and *child of T* = 8 or 9 then return *error*
    else return *numval* (*child of T*);
  end case;
end *EvalWithBase*;

Notes

- The *base* attribute and the *val* attribute have the same *integer* type
- Initial invocation
  - EvalWithBase (rootnode, 0);

not regular : based-num의 맨 처음 EvalWithBase에서 base parameter는 아직 존재하지 않는데 값을 집어 넣었다,
dummy base
ignore 된다.

- Modification

(더 일반적으로 처리하기위해)

  - To distinguish three cases
    - based_num
    - basechar
    - num and digit case

Example 6.15

- **Pseudocode**

```
function EvalBasedNum (T: treenode ): integer;
(* only called on root node *)
begin
  return EvalNum(left child of T, EvalBase(right child of T));
end EvalBasedNum;


function EvalBase (T: treenode ): integer;
(* only called on basechar node *)
begin
  if child of T = o then return 8
  else return 10;
end EvalBase;
```

Example 6.15

- **Pseudocode**

```
function EvalNum (T: treenode; base: integer ): integer;
var temp, temp2: integer;
begin
  case nodekind of T of
  num:
    temp := EvalWithBase(left child of T, base );
    if right child of T is not nil then
       temp2 := EvalWithBase (right child of T, base );
       if temp ≠ error and temp2 ≠ error then
          return base*temp + temp2
       else return error;
     else return temp;
   digit:
     if base = 8 and child of T = 8 or 9 then return error
     else return numval (child of T);
   end case;
end EvalNum;
```

The Use of External Data Structures - Example 6.16

when attribute values do not lead themselves easily to the method of parameters and return values

- Pseudocode
  - use a nonlocal variable to store '*base*'

    가능한 이유 : once base is set, it is fixed for the duration of the value computation 반면 fixed 되지 않음녀 이 방법은 위험하다.

```
function EvalWithBase (T: treenode ): integer;
var temp, temp2: integer;
begin
 case nodekind of T of
 based-num:
    SetBase(right child of T );
    return EvalWithBase(left child of T );
 num:
    temp := EvalWithBase(left child of T );
    if right child of T is not nil then
       temp2 := EvalWithBase (right child of T );
       if temp ≠ error and temp2 ≠ error then
          return base*temp + temp2
       else return error;
     else return temp;
  digit:
     if base = 8 and child of T = 8 or 9 then return error
     else return numval (child of T);
  end case;
end EvalWithBase;
```

```
procedure SetBase (T: treenode ):
begin
   if child of T = o then base := 8
   else base := 10;
end SetBase;
```

Example 6.16

attribute equation 변경시킴 (to reflect the use of the nonlocal base variable)

Grammar Rule	Semantic Rules
based-num → *num basechar*	*based-num.val* = *num.val*
basechar → **o**	*base* := 8
basechar → **d**	*base* := 10
$num_1 \rightarrow num_2 digit$	$num_1.val$ = **if** *digit.val* = *error* **or** $num_2.val$ = *error* **then** *error* **else** $num_2.val * base + digit.val$
etc.	etc.

Notes

- one of the prime examples of a data struct external to the syntax tree is
  - the **symbol table** store attributes associated to declared constants,variables, and procedures in a program
- A symbol table
  - dictionary data structure with operations such as
    - insert, lookup, and delete

Example 6.17

- information in declarations is inserted into a symbol table

procedure *insert* (*name* : *string* ; *dtype* : *typekind*);

Grammar Rule	Semantic Rules
$decl \rightarrow type\ var\text{-}list$	
$type \rightarrow \textbf{int}$	$dtype = integer$
$type \rightarrow \textbf{float}$	$dtype = real$
$var\text{-}list_1 \rightarrow \textbf{id}\ \textbf{,}\ var\text{-}list_2$	$insert(\textbf{id}.name, dtype)$
$var\text{-}list \rightarrow \textbf{id}$	$insert(\textbf{id}.name, dtype)$

dtype이 한번 set되면 계산 과정에서 fixexd 되기 때문에 nonlocal variable로 처리 가능

Example 6.17

- **Pseudocode**

```
procedure EvalType (T: treenode );
begin
   case nodekind of T of
   decl:
         EvalType( type child of T );
         EvalType( var-list child of T );
   type:
         if child of T = int then dtype := integer
         else dtype := real;
   var-list:
         insert(name of first child of T, dtype)
         if third child of T is not nil then
            EvalType ( third child of T );
   end case;
end EvalType;
```

The Computation of Attributes During Parsing

- To what extent attributes can be computed at the same time as the parsing stage?
  - Possibility of computing all attributes during the parse?
- Which attributes can be successfully computed during a parse depends on the power and properties of the parsing method employed
- LL and LR parsing techniques → left-to-right traversal (process the input program from left to right)
  - This is not a restriction for synthesized attributes
  - For inherited attributes, this means that there may be no "backward" dependencies in the dependency graph → **L-attributed** dependencies pointing from right to left in parse tree

L-attributed

- Definition
  - An attribute grammar for attributes $a_1, \ldots, a_k$ is **L-attributed** if, for each inherited attribute a_j and each grammar rule

    $$X_0 \rightarrow X_1 X_2 \ldots X_n$$

    the associated equations for a_j are all of the form

$$X_i.a_j = f_{ij}(X_0.a_1,\ldots,X_0.a_k,X_1.a_0,\ldots,X_1.a_k,\ldots,X_{i-1}.a_1,\ldots,X_{i-1}.a_k)$$

  That is, the value of a_j at X_i can only depend on attributes of the symbol $X_0, \ldots, X_{i-1}$ that occur to the left of X_i in the grammar rule. Additionally, only inherited attribute of X_0 can appear in the f_{ij}

Notes

- As a special case, S-attributed grammar is L-attributed
- Unfortunately, LR parsers, such as an LALR(1) parser, are suited to handling primarily synthesized attributes
  - put off deciding which grammar rule to use in a derivation until the right-had side of a grammar rule is fully formed
  - This makes it difficult for inherited attributes to be made available unless their property remain fixed for all posiible right-hand side choices

The Computation of Attributes During Parsing

- actions on the value stack
  - a **value stack** is used to store synthesized attributes

Value Stack	Semantic Action
\$ 12 + 5	$E_1.val = E_2.val + E_3.val$

pop t3	{ get E_3.val from the value stack }
pop	{ discard the + token }
pop t2	{ get E_2.val from the value stack }
t1 = t2 + t3	{ add }
push t1	{ push the result back onto the value stack }

Table 6.8 LR parsing

	Parsing Stack	Input	Parsing Action	Value Stack	Semantic Action
1	\$	3*4+5 \$	shift	\$	
2	\$ $\boldsymbol{n}$	*4+5 \$	reduce $E \rightarrow \boldsymbol{n}$	\$ $\boldsymbol{n}$	$E.val = \boldsymbol{n}.val$
3	\$ E	*4+5 \$	shift	\$ 3	
4	\$ E *	4+5 \$	shift	\$ 3 *	
5	\$ E * $\boldsymbol{n}$	+5 \$	reduce $E \rightarrow \boldsymbol{n}$	\$ 3 * $\boldsymbol{n}$	$E.val = \boldsymbol{n}.val$
6	\$ E * E	+5 \$	reduce $E \rightarrow E * E$	\$ 3 * 4	$E_1.val = E_2.val * E_3.val$
7	\$ E	+5 \$	shift	\$ 12	
8	\$ E +	5 \$	shift	\$ 12 +	
9	\$ E + $\boldsymbol{n}$	\$	reduce $E \rightarrow \boldsymbol{n}$	\$ 12 + $\boldsymbol{n}$	$E.val = \boldsymbol{n}.val$
10	\$ E + E	\$	reduce $E \rightarrow E + E$	\$ 12 + 5	$E_1.val = E_2.val + E_3.val$
11	\$ E	\$		\$ 17	

Inheriting a Previously Computed Synthesized Attribute

- Suppose
  - $A \rightarrow B\ C$
  - C has an inherited attribute i that depends on the synthesized attribute s of B
    - i.e. $C.i = f(B.s)$

Grammar Rule	**Semantic Rules**
$A \rightarrow B\ D\ C$	
$B \rightarrow \ldots$	{ compute $B.s$ }
$D \rightarrow \varepsilon$	*saved_i* = *f*(*valstack*[*top*])
$C \rightarrow \ldots$	{ now *saved_i* is available }

Inheriting a Previously Computed Synthesized Attribute

- Problems
  - requires the programmer to directly access the value stack during a parse → risky In yacc,there is no value stack. so programmer implement such a schema in yacc
  - only works if the position of the previously computed attribute is predictable from the grammar
    - what if *var-list* is right recursive arbitary number of id's could be on the stack, and the postion of dtype in the tack would be unkown
  - ε-productions can add parsing conflicts

L-attributed grammar

- $X_0 \rightarrow X_1X_2\ldots X_n$
- The value of a_j at X_i can only depend on attributes of the symbols X_0, …, X_{i-1}.
- Additionally, only inherited attributes of X_0 can appear in the function.

Using two copy rules

Grammar Rule	Semantic Rules
decl → *type var-list*	*var-list.dtype* = *type.dtype*
type → **int**	*type.dtype* = *integer*
type → **float**	*type.dtype* = *real*
$var\text{-}list_1$ → $var\text{-}list_2$ **, id**	*insert*(**id**.name, $var\text{-}list_1$, *dtype*)
	$var\text{-}list_2.dtype$ = $var\text{-}list_1.dtype$
var-list → **id**	*insert*(**id**.name, *var-list.dtype*)

- *dtype* can be computed before the first *var-list* is recognized (*var-list.dtype* = *type.dtype*)
- When *var-list* → *var-list*, ***id*** is reduced, *dtype* is three positions below the top of the stack.

Eliminating the two copy rules

Grammar Rule	Semantic Rules
$decl \rightarrow type\ var\text{-}list$	
$type \rightarrow$ **int**	$type.dtype = integer$
$type \rightarrow$ **float**	$type.dtype = real$
$var\text{-}list_1 \rightarrow var\text{-}list_2$ **, *id***	$insert(\textbf{id}.name, valstack[top\text{-}3])$
$var\text{-}list \rightarrow$ ***id***	$insert(\textbf{id}.name, valstack[top\text{-}1])$

can directly access value stack

when the position of a previously computed synthesized attribute in the value stack can always be predicted

Example 6.18
– can make the computation of attributes simpler -

- **Grammar**

$$decl \rightarrow type\ var\text{-}list$$

$$type \rightarrow \textbf{int} \mid \textbf{float}$$

dtype inherited attribute

$$var\text{-}list \rightarrow \textbf{id}, var\text{-}list \mid \textbf{id}$$

- **Rewrite Grammar** – the dtype attribute becomes synthesized

$$decl \rightarrow var\text{-}list\ \ \textbf{id}$$

$$var\text{-}list \rightarrow var\text{-}list\ \ \textbf{id}\ , \mid type$$

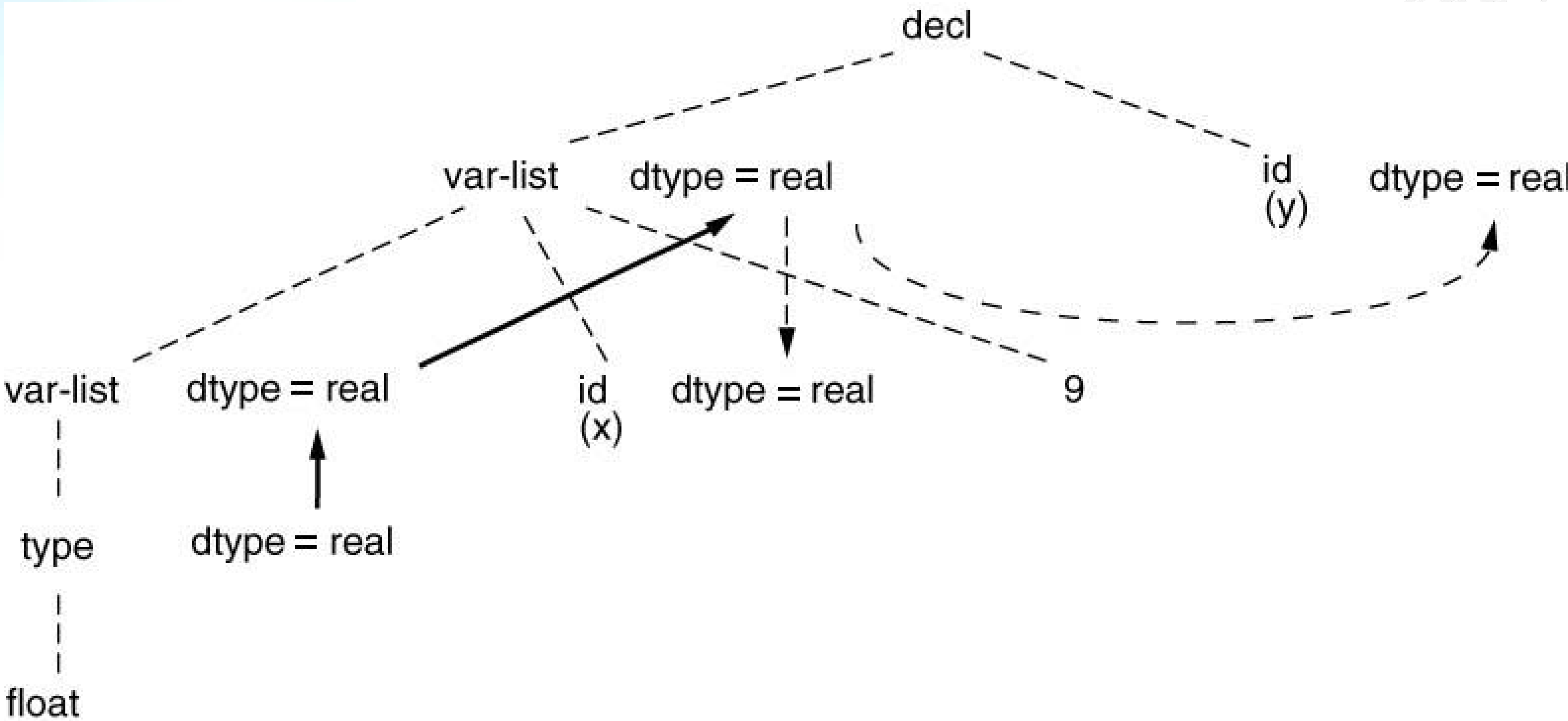
$$type \rightarrow \textbf{int} \mid \textbf{float}$$

given an attribute grammar, all inherited attributes can be changed into synthesized attributes by suitable modification of the grammar, without chaning the language of the grammar.
(make the grammar and semantic rules much more complex and difficult to understand)

Example 6.18

Grammar Rule	Semantic Rules
$decl \rightarrow var\text{-}list\ \mathbf{id}$	$\mathbf{id}.dtype = var\text{-}list.dtype$
$var\text{-}list_1 \rightarrow var\text{-}list_2\ \mathbf{id}\ \mathbf{,}$	$var\text{-}list_1.dtype = var\text{-}list_2.dtype$ $\mathbf{id}.dtype = var\text{-}list_1.dtype$
$var\text{-}list \rightarrow type$	$var\text{-}list.dtype = type.dtype$
$type \rightarrow \mathbf{int}$	$type.dtype = integer$
$type \rightarrow \mathbf{float}$	$type.dtype = real$

Figure 6.11



In case of **id**, these dependencies are always to leaves in the parse tree
And may be achieved by operations at the appropriate parent nodes.

Symbol Table

- principal sym table operations
  - *insert*, *lookup*, and *delete*
- typical dictionary data structure
  - linear lists
  - tree structures
  - hash tables

data structures

- linear lists
  - constant-time *insert* operation
  - *lookup* and *delete* operations
    - linear time in the size of the list
  - slow compilation time
- tree structures
  - less useful
  - not provide best case efficiency
  - complex *delete* operation
- hash table
  - best choice

hash table

- buckets
- hash function
- collisions
- collision-resolution
  - open addressing
    - Resolves collisions by inserting new items in successive buckets
    - delete?
  - separate chaining
    - **mod** function
    - how to handle an identifier (i.e. char string)?
      - Ex) temp1, temp2, ...

declarations

http://usecurity.hanyang.ac.kr

- four basic kinds of declarations
  - constant declarations
    - C: `const`
  - type declarations
    - Pascal: `type Table = array [1..size] of Entry;`
    - C: `struct, union, typedef`
  - variable declarations
    - FORTRAN: `integer a,b(100)`
    - C: `integer a,b[100];`
    - declaration by use
  - procedure declarations
    - explicit (C, …) vs. implicit (FORTRAN, BASIC)

explicit - 명시적으로 표시
implicit - 표시안해줘도 처리하는데,
linking에서 함수 확인

- different symbol tables for different kinds of declarations

declarations

- constant declarations
  - also called **value bindings**
  - static or dynamic
    - Dynamic: only computable during execution
- type declarations
  - bind names to newly constructed types
  - type checking
- variable declarations
  - bind names to data types
  - **scope** of a declaration
    - `automatic`
    - `static`
    - `extern`