Semantic Analysis with Attribute Grammars Part 5

Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012

NPTEL Course on Principles of Compiler Design

Outline of the Lecture

- Introduction (covered in lecture 1)
- Attribute grammars (covered in lectures 2 and 3)
- Attributed translation grammars (covered in lecture 3)
- Semantic analysis with attributed translation grammars

Symbol Table Data Structure

- A symbol table (in a compiler) stores names of all kinds that occur in a program along with information about them
  - Type of the name (int, float, function, etc.), level at which it has been declared, whether it is a declared parameter of a function or an ordinary variable, etc.
  - In the case of a function, additional information about the list of parameters and their types, local variables and their types, result type, etc., are also stored
- It is used during semantic analysis, optimization, and code generation
- Symbol table must be organized to enable a search based on the level of declaration
- It can be based on:
  - Binary search tree, hash table, array, etc.

A Simple Symbol Table - 1

- A very simple symbol table (quite restricted and not really fast) is presented for use in the semantic analysis of functions
- An array, *func_name_table* stores the function name records, assuming no nested function definitions
- Each function name record has fields: name, result type, parameter list pointer, and variable list pointer
- Parameter and variable names are stored as lists
- Each parameter and variable name record has fields: name, type, parameter-or-variable tag, and level of declaration (1 for parameters, and 2 or more for variables)

A Simple Symbol Table - 2

func_name_table

name	result type	parameter list pointer	local variable list pointer	number of parameters

Parameter/Variable name record

name	type	parameter or variable tag	level of declaration

A Simple Symbol Table - 3

- Two variables in the same function, with the same name but different declaration levels, are treated as different variables (in their respective scopes)
- If a variable (at level > 2) and a parameter have the same name, then the variable name overrides the parameter name (only within the corresponding scope)
- However, a declaration of a variable at level 2, with the same name as a parameter, is flagged as an error
- The above two cases must be checked carefully
- A search in the symbol table for a given name must always consider the names with the declaration levels *l, l-1, ... , 2*, in that order, where *l* is the current level

v3, integer, var, 2

v4, real, var, 2

func_name_table

name	type	param list ptr	local var list ptr	num param
f1	int			2
f2	void	NIL		0

p1, real, param, 1

p2, integer, param, 1

v1, real, var, 2

v2, integer, var, 2

v3, integer, var, 3

v1, real, var, 3

```
int  f1( float p1, int p2){
   float v1; int v2;
...
      { int v3; float v1;
          ....
      }
...
}

void f2 (){
   int v3; float v4;
...
}
```

Variables v1(level 2) and v1(level 3) in function f1 are different variables

A Simple Symbol Table - 5

- The global variable, *active_func_ptr*, stores a pointer to the function name entry in *func_name_table* of the function that is currently being compiled
- The global variable, *level*, stores the current nesting level of a statement block
- The global variable, *call_name_ptr*, stores a pointer to the function name entry in *func_name_table* of the function whose call is being currently processed
- The function $search\_func(n, found, fnptr)$ searches the function name table for the name n and returns *found* as T or F; if found, it returns a pointer to that entry in *fnptr*

A Simple Symbol Table - 6

- The function *search_param*(p, *fnptr*, *found*, *pnptr*) searches the parameter list of the function at *fnptr* for the name p, and returns *found* as T or F; if found, it returns a pointer to that entry in the parameter list, in *pnptr*
- The function *search_var*(v, *fnptr*, l, *found*, *vnptr*) searches the variable list of the function at *fnptr* for the name v at level l or lower, and returns *found* as T or F; if found, it returns a pointer to that entry in the variable list, in *vnptr*. Higher levels are preferred
- The other symbol table routines will be explained during semantic analysis

SATG for Sem. Analysis of Functions and Calls - 1

1. $FUNC\_DECL \rightarrow FUNC\_HEAD \; \{ \; VAR\_DECL \; BODY \; \}$
2. $FUNC\_HEAD \rightarrow RES\_ID \; (\; DECL\_PLIST \;)$
3. $RES\_ID \rightarrow RESULT \;\; id$
4. $RESULT \rightarrow int \mid float \mid void$
5. $DECL\_PLIST \rightarrow DECL\_PL \mid \epsilon$
6. $DECL\_PL \rightarrow DECL\_PL \; , \; DECL\_PARAM \mid DECL\_PARAM$
7. $DECL\_PARAM \rightarrow T \; id$
8. $VAR\_DECL \rightarrow DLIST \mid \epsilon$
9. $DLIST \rightarrow D \mid DLIST \; ; \; D$
10. $D \rightarrow T \; L$
11. $T \rightarrow int \mid float$
12. $L \rightarrow id \mid L \; , \; id$

SATG for Sem. Analysis of Functions and Calls - 2

13. $BODY \rightarrow \{\ VAR\_DECL\ STMT\_LIST\ \}$
14. $STMT\_LIST \rightarrow STMT\_LIST\ ;\ STMT \mid STMT$
15. $STMT \rightarrow BODY \mid FUNC\_CALL \mid ASG \mid$ /* others */
    /* BODY may be regarded as a *compound statement* */
    /* Assignment statement is being singled out */
    /* to show how function calls can be handled */
16. $ASG \rightarrow LHS\ :=\ E$
17. $LHS \rightarrow id$ /* array expression for exercises */
18. $E \rightarrow LHS \mid FUNC\_CALL \mid$/* other expressions */
19. $FUNC\_CALL \rightarrow id\ (\ PARAMLIST\ )$
20. $PARAMLIST \rightarrow PLIST \mid \epsilon$
21. $PLIST \rightarrow PLIST\ ,\ E \mid E$

SATG for Sem. Analysis of Functions and Calls - 3

1. *FUNC_DECL* $\rightarrow$ *FUNC_HEAD* { *VAR_DECL BODY* }
   {delete_var_list(active_func_ptr, level);
   active_func_ptr := NULL; level := 0;}
2. *FUNC_HEAD* $\rightarrow$ *RES_ID* (*DECL_PLIST*) {level := 2}
3. *RES_ID* $\rightarrow$ *RESULT* *id*
   { search_func(id.name, found, namptr);
   if (found) error('function already declared');
   else enter_func(id.name, RESULT.type, namptr);
   active_func_ptr := namptr; level := 1}
4. *RESULT* $\rightarrow$ *int* {**action1**} | *float* {**action2**}
   | *void* {**action3**}
   {action 1:} {RESULT.type := integer}
   {action 2:} {RESULT.type := real}
   {action 3:} {RESULT.type := void}

5. *DECL_PLIST* $\rightarrow$ *DECL_PL* | ϵ
6. *DECL_PL* $\rightarrow$ *DECL_PL* , *DECL_PARAM* | *DECL_PARAM*
7. *DECL_PARAM* $\rightarrow$ *T id*
   {search_param(id.name, active_func_ptr, found, pnptr);
   if (found) {error('parameter already declared')}
   else {enter_param(id.name, T.type, active_func_ptr)}
8. *T* $\rightarrow$ *int* {T.type := integer} | *float* {T.type := real}
9. *VAR_DECL* $\rightarrow$ *DLIST* | ϵ
10. *DLIST* $\rightarrow$ *D* | *DLIST* ; *D*
    /* We show the analysis of simple variable declarations. Arrays can be handled using methods desribed earlier. Extension of the symbol table and SATG to handle arrays is left as an exercise. */

11. $D \rightarrow T\ L$ {patch_var_type(T.type, L.list, level)}
    /* Patch all names on L.list with declaration level, *level*, with T.type */
12. $L \rightarrow id$
    ```
    {search_var(id.name, active_func_ptr, level, found, vn);
     if (found && vn -> level == level)
        {error('variable already declared at the same level');
          L.list := makelist(NULL);}
     else if (level==2)
    {search_param(id.name, active_func_ptr, found, pn);
     if (found) {error('redeclaration of parameter as variable');
                  L.list := makelist(NULL);}
    } /* end of if (level == 2) */
     else {enter_var(id.name, level, active_func_ptr, vnptr);
            L.list := makelist(vnptr);}}
    ```

SATG for Sem. Analysis of Functions and Calls - 6

13. $L_1 \rightarrow L_2$, id
    {search_var(id.name, active_func_ptr, level, found, vn);
    if (found && vn -> level == level)
    {error('variable already declared at the same level');
    L_1.list := L_2.list;}
    else if (level==2)
    {search_param(id.name, active_func_ptr, found, pn);
    if (found) {error('redclaration of parameter as variable');
    L_1.list := L_2.list;}
    } /* end of if (level == 2) */
    else {enter_var(id.name, level, active_func_ptr, vnptr);
    L_1.list := append(L_2.list, vnptr);}}
14. $BODY \rightarrow$ '{'{level++;} VAR_DECL $STMT\_LIST$
    {delete_var_list(active_func_ptr, level); level- -;}'}'
15. $STMT\_LIST \rightarrow STMT\_LIST$; $STMT \mid STMT$
16. $STMT \rightarrow BODY \mid FUNC\_CALL \mid ASG \mid$ /* others */

17. *ASG* $\rightarrow$ *LHS* $:=$ *E*
    {if (LHS.type $\neq$ *errortype* && E.type $\neq$ *errortype*)
      if (LHS.type $\neq$ E.type) error('type mismatch of
          operands in assignment statement')}
18. *LHS* $\rightarrow$ *id*
    {search_var(id.name, active_func_ptr, level, found, vn);
     if ($\sim$found)
       {search_param(id.name, active_func_ptr, found, pn);
         if ($\sim$found){ error('identifier not declared');
                          LHS.type := *errortype*}
         else LHS.type := pn -> type}
     else LHS.type := vn -> type}
19. *E* $\rightarrow$ *LHS* {E.type := LHS.type}
20. *E* $\rightarrow$ *FUNC_CALL* {E.type := FUNC_CALL.type}

21. *FUNC_CALL* $\rightarrow$ *id (PARAMLIST)*

```
{ search_func(id.name, found, fnptr);
  if (∼found) {error(‘function not declared’);
                  call_name_ptr := NULL;
                  FUNC_CALL.type := errortype;}
  else {FUNC_CALL.type := get_result_type(fnptr);
        call_name_ptr := fnptr;
  if (call_name_ptr.numparam ≠ PARAMLIST.pno)
     error(‘mismatch in mumber of parameters
           in declaration and call’);}
```

22. *PARAMLIST* $\rightarrow$ *PLIST* {PARAMLIST.pno := PLIST.pno }
    | ϵ {PARAMLIST.pno := 0 }

23. $PLIST \rightarrow E$ {PLIST.pno := 1;
    check_param_type(call_name_ptr, 1, E.type, ok);
    if (∼ok) error('parameter type mismatch
    in declaration and call');}
24. $PLIST_1 \rightarrow PLIST_2$, E {$PLIST_1$.pno := $PLIST_2$.pno + 1;
    check_param_type(call_name_ptr, $PLIST_2$.pno + 1,
    E.type, ok);
    if (∼ok) error('parameter type mismatch
    in declaration and call');}

Semantic Analysis of Arrays

Multi-dimensional arrays

- length of each dimension must be stored in the symbol table and connected to the array name, while processing declarations
- C allows assignment of array slices. Therefore, size and type of slices must be checked during semantic analysis of assignments
- `int a[10][20], b[20], c[10][10];`
  `a[5] = b; c[7] = a[8];`
  In the above code fragment, the first assignment is valid, but the second one is not
- The above is called *structure equivalence* and it is different from *name equivalance*

Semantic Analysis of Structs

- Names inside structs belong to a higher level
- Equivalance of structs is based on *name equivalance* and not on *structure equivalence*
- ```
  struct {int a,b; float c[10]; char d} x,y;
  struct {char d; float c[10]; int a,b} a,b;
  x = y; a = x;
  ```
- In the code fragment above
  - In the second struct, the fields `a, b` of the struct are different from the struct variables *a* and *b*
  - The assignment `x = y;` is valid but `a = x;` is not valid, even though both structs have the same fields (but permuted)
- For a `struct` variable, an extra pointer pointing to the fields of the struct variable, along with their levels, can be maintained in the symbol table
  ```

Operator Overloading

- Operators such as '+' are usually overloaded in most languages
  - For example, the same symbol '+' is used with integers and reals
  - Programmers can define new functions for the existing operators in C++
  - This is operator overloading
  - Examples are defining '+' on complex numbers, rational numbers, or *time*

```
Complex operator+(const Complex& lhs,
                  const Complex& rhs)
{   Complex temp = lhs;
    temp.real += rhs.real;
    temp.imaginary += rhs.imaginary;
    return temp;
}
```

Function Overloading

- C++ also allows function overloading
- Overloaded functions with the same name (or same operator)
  - return results with different *types*, or
  - have different number of parameters, or
  - differ in parameter types
- The meaning of overloaded operators (in C++) with built-in types as parameters cannot be redefined
  - E.g., '+' on integers cannot be overloaded
  - Further, overloaded '+' must have exactly two operands
- Both operator and function overloading are resolved at compile time
- Either of them is different from *virtual functions* or *function overriding*

Function Overloading Example

```
// area of a square
int area(int s) { return s*s; }

// area of a rectangle
int area(int l, int b) { return l*b; }

// area of a circle
float area(float radius)
{ return 3.1416*radius*radius; }

int main()
{   std::cout << area(10);
    std::cout << area(12, 8);
    std::cout << area(2.5);
}
```

Implementing Operator Overloading

- A list of operator functions along with their parameter types is needed
- This list may be stored in a hash table, with the hash function designed to take the operator and its parameter types into account
- While handling a production such as $E \rightarrow E_1 + E_2$, the above hash table is searched with the signature $+(E_1.type, E_2.type)$
- If there is only one exact match (with the same operand types), then the overloading is resolved in favor of the match
- In case there is more than one exact match, an error is flagged
- The situation gets rather complicated in C++, due to possible conversions of operand types (char to int, int to float, etc.)

Implementing Function Overloading

- The symbol table should store multiple instances of the same function name along with their parameter types (and other information)
- While resolving a function call such as, $test(a, b, c)$, all the overloaded functions with the name *test* are collected and the closest possible match is chosen
  - Suppose the parameters a, b, c are all of `int` type
  - And the available overloaded functions are: `int test(int a, int b, float c)` and `int test(float a, int b, float c)`
  - In this case, we may choose the first one because it entails only one conversion from `int` to `float` (faster)
- If there is no match (or more than one match) even after conversions, an error is flagged

- $FUNC\_DECL \rightarrow FUNC\_HEAD \; \{ \; VAR\_DECL \;\; BODY \}$
  $BODY \rightarrow \{ \; VAR\_DECL \;\; STMT\_LIST \; \}$
  - Variable declarations appear stricty before their use
- $FUNC\_DECL \rightarrow$
  $FUNC\_HEAD \; \{ \; VAR\_DECL \;\; BODY \;\; VAR\_DECL \; \}$
  $BODY \rightarrow \{ \; VAR\_DECL \;\; STMT\_LIST \;\; VAR\_DECL \; \}$
  - permits variable declarations before *and after their use*
- Semantic analysis in this case requires two passes
  - Symbol table is constructed in the 1^{st} pass
  - Declarations are all processed in the 1^{st} pass
  - 1^{st} pass can be integrated with LR-parsing during which a parse tree is built
  - Statements are analyzed in the 2^{nd} pass
  - Sem. errors in statements are reported only in the 2^{nd} pass
  - This effectively presents all the variable declarations before their use
  - 2^{nd} pass can be made over the parse tree

Symbol Table for a 2-pass Semantic Analyzer

block_table (indexed by blk.num)

blk. num	name	result type	param. list ptr	local var. list ptr	num. param	surr. blk. num
1						
2						
3						
4						

Parameter/Variable name record

name	type	parameter or variable tag	level of declaration	blk.num

- The symbol table has to be *persistent*
- Cannot be destroyed after the block/function is processed in pass 1
- Should be stored in a form that can be accessed according to levels in pass 2

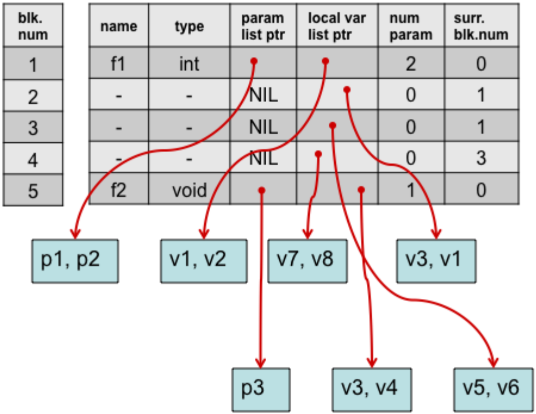
Symbol Table for a 2-pass Semantic Analyzer(contd.)

- The symbol table(ST) is indexed by block number
- In the previous version of the ST, there were no separate entries for blocks
- The surrounder block number (*surr.blk.num*) is the block number of the enclosing block
- All the blocks below a function entry *f* in the ST, upto the next function entry, belong to the function *f*
- To get the name of the parent function for a given block *b*, we go up table using surrounder block numbers until the surrounder block number becomes zero

Symbol Table for a 2-pass Semantic Analyzer(contd.)

```
1  int  f1( float p1, int p2){
       float v1; int v2;
       ...
2      { int v3; float v1;
          ....
       }
       ...
3      { float v5,v6;
          ....
4         { char v7,v8;
             ...
          }
       }
   } /* end of f1 */
5  void f2 (char p3){
       int v3; float v4;
     ...
   }
```

block_table



blk. num	name	type	param list ptr	local var list ptr	num param	surr. blk.num
1	f1	int			2	0
2	-	-	NIL		0	1
3	-	-	NIL		0	1
4	-	-	NIL		0	3
5	f2	void			1	0

- Block numbers begin from 1, and a counter *last_blk_num* generates new block numbers by incrementing itself
- *curr_blk_num* is the currently open block
- While opening a new block, *curr_blk_num* becomes its surrounder block number
- Similarly, while closing a block, its *surr.blk.num* is copied into *curr_blk_num*

Symbol Table for a 2-pass Semantic Analyzer(contd.)

- Apart from *active_func_ptr*, and *call_name_ptr*, we also need an *active_blk_ptr*
- *level* remains the same (nesting level of the current block)
- $search\_func(n, found, fnptr)$ remains the same, except that it searches entries corresponding to functions only (with $surr.blk.num = 0$)
- $search\_param(p, fnptr, found, pnptr)$ remains the same
- $search\_var(v, fnptr, l, found, vnptr)$ is similar to the old one, but the method of searching is now different
  - The variables of each block are stored separately under different block numbers
  - The parameter *level* is now replaced by *active_blk_ptr*
  - The search starts from *active_blk_ptr* and proceeds upwards using surrounder block numbers until the enclosing function is reached (with $surr.blk.num = 0$)