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
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 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

<table>
<thead>
<tr>
<th>name</th>
<th>result type</th>
<th>parameter list pointer</th>
<th>local variable list pointer</th>
<th>number of parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Parameter/Variable name record

<table>
<thead>
<tr>
<th>name</th>
<th>type</th>
<th>parameter or variable tag</th>
<th>level of declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
A Simple Symbol Table - 4

```
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
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*. 
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.
1. `FUNC_DECL → FUNC_HEAD { VAR_DECL BODY }`
2. `FUNC_HEAD → RES_ID ( DECL_PLIST )`
3. `RES_ID → RESULT id`
4. `RESULT → int | float | void`
5. `DECL_PLIST → DECL_PL | ϵ`
6. `DECL_PL → DECL_PL , DECL_PARAM | DECL_PARAM`
7. `DECL_PARAM → T id`
8. `VAR_DECL → DLIST | ϵ`
9. `DLIST → D | DLIST ; D`
10. `D → T L`
11. `T → int | float`
12. `L → id | L , id`
\[ BODY \rightarrow \{ \text{VAR
decl} \ STMT\_LIST \} \]
\[ STMT\_LIST \rightarrow STMT\_LIST ; \ STMT \mid STMT \]
\[ STMT \rightarrow BODY \mid \text{FUNC\_CALL} \mid \text{ASG} \mid /* \text{others} */ \]
\[ /* \text{BODY may be regarded as a compound statement} */ \]
\[ /* \text{Assignment statement is being singled out} */ \]
\[ /* \text{to show how function calls can be handled} */ \]
\[ ASG \rightarrow LHS := E \]
\[ LHS \rightarrow id /* \text{array expression for exercises} */ \]
\[ E \rightarrow LHS \mid \text{FUNC\_CALL} \mid /* \text{other expressions} */ \]
\[ \text{FUNC\_CALL} \rightarrow id \ ( \text{PARAMLIST} ) \]
\[ \text{PARAMLIST} \rightarrow \text{PLIST} \mid \epsilon \]
\[ \text{PLIST} \rightarrow \text{PLIST} , E \mid E \]
1  \textit{FUNC\_DECL} \rightarrow \textit{FUNC\_HEAD} \{ \textit{VAR\_DECLL BODY} \} \\
\{\text{delete\_var\_list}(active\_func\_ptr, level); \\
\quad active\_func\_ptr := \texttt{NULL}; level := 0;\}

2  \textit{FUNC\_HEAD} \rightarrow \textit{RES\_ID} ( \textit{DECL\_PLIST} ) \{\text{level} := 2\}

3  \textit{RES\_ID} \rightarrow \textit{RESULT} \ {id} \\
\{ \text{search\_func}(id.\text{name}, \text{found}, \text{namptr}); \\
\quad \text{if (found) error(‘function already declared’)}; \\
\quad \text{else enter\_func}(id.\text{name}, \text{RESULT.type}, \text{namptr}); \\
\quad active\_func\_ptr := \text{namptr}; level := 1\}

4  \textit{RESULT} \rightarrow \textit{int} \ {\text{action1}} \mid \textit{float} \ {\text{action2}} \\
\quad \mid \textit{void} \ {\text{action3}} \\
\{\text{action 1:} \} \{\text{RESULT.type} := \text{integer}\} \\
\{\text{action 2:} \} \{\text{RESULT.type} := \text{real}\} \\
\{\text{action 3:} \} \{\text{RESULT.type} := \text{void}\}
5. \( \text{DECL\_PLIST} \rightarrow \text{DECL\_PL} \mid \epsilon \)

6. \( \text{DECL\_PL} \rightarrow \text{DECL\_PL} \ , \ \text{DECL\_PARAM} \mid \text{DECL\_PARAM} \)

7. \( \text{DECL\_PARAM} \rightarrow T \ id \\
\{\text{search\_param(id.name, active\_func\_ptr, found, pnptr);} \\
\quad \text{if (found) \{error('parameter already declared')}}\} \\
\quad \text{else \{enter\_param(id.name, T.type, active\_func\_ptr)}\}

8. \( T \rightarrow \text{int} \{T\.type := \text{integer}\} \mid \text{float} \{T\.type := \text{real}\} \)

9. \( \text{VAR\_DECL} \rightarrow \text{DLIST} \mid \epsilon \)

10. \( \text{DLIST} \rightarrow D \mid \text{DLIST} ; D \)

/* We show the analysis of simple variable declarations. Arrays can be handled using methods described earlier. Extension of the symbol table and SATG to handle arrays is left as an exercise. */
\[ D \rightarrow T L \ \{\text{patch\_var\_type}(T.\text{type}, L.\text{list}, \text{level})\}\]

/* Patch all names on L.list with declaration level, \text{level}, with T.type */

\[ L \rightarrow id \]

\{\text{search\_var}(id.\text{name}, \text{active\_func\_ptr}, \text{level}, \text{found}, \text{vn});
  \text{if} (\text{found} && \text{vn} \rightarrow \text{level} == \text{level})
  \{\text{error('variable already declared at the same level')};
    \text{L.list} := \text{makelist}(\text{NULL});\}
  \text{else if} (\text{level}==2)
  \{\text{search\_param}(id.\text{name}, \text{active\_func\_ptr}, \text{found}, \text{pn});
    \text{if} (\text{found}) \{\text{error('redeclaration of parameter as variable')};
      \text{L.list} := \text{makelist}(\text{NULL});\}
  \} /* end of if (level == 2) */
  \text{else} \{\text{enter\_var}(id.\text{name}, \text{level}, \text{active\_func\_ptr}, \text{vnptr});
    \text{L.list} := \text{makelist}(\text{vnptr});\}\}
L₁ → L₂, id
{search_var(id.name, active_func_ptr, level, found, vn);
  if (found && vn -> level == level)
    {error('variable already declared at the same level');
      L₁.list := L₂.list;}
  else if (level==2)
    {search_param(id.name, active_func_ptr, found, pn);
      if (found) {error('redclaration of parameter as variable');
        L₁.list := L₂.list;}
    } /* end of if (level == 2) */
  else {enter_var(id.name, level, active_func_ptr, vnptr);
    L₁.list := append(L₂.list, vnptr);}]

BODY → '{' {level++;} VAR_DECL STMT_LIST
  {delete_var_list(active_func_ptr, level); level- -;}'}'

STMT_LIST → STMT_LIST ; STMT | STMT

STMT → BODY | FUNC_CALL | ASG | /* others */
17 \[ \text{ASG} \rightarrow \text{LHS} := E \]
{if (LHS.type \neq \text{errortype} \&\& E.type \neq \text{errortype})
  if (LHS.type \neq E.type) \text{error(‘type mismatch of operands in assignment statement’)}
}

18 \[ \text{LHS} \rightarrow id \]
{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){ \text{error(‘identifier not declared’)};
        \text{LHS.type := errortype}
      }
      else LHS.type := pn -> type
    }
  else LHS.type := vn -> type
}

19 \[ E \rightarrow \text{LHS} \{E.type := \text{LHS.type}\} \]

20 \[ E \rightarrow \text{FUNC\_CALL} \{E.type := \text{FUNC\_CALL.type}\} \]
\[ \text{FUNC_CALL} \rightarrow \text{id} (\text{PARAMLIST}) \]
\[
\{ \text{search_func(id.name, found, fnptr);} \\
\quad \text{if (\sim\text{found}) \{\text{error(‘function not declared’);}} \\
\quad \quad \text{call_name_ptr := NULL;} \\
\quad \quad \text{FUNC_CALL.type := errortype;}} \\
\quad \text{else \{FUNC_CALL.type := get_result_type(fnptr);} \\
\quad \quad \text{call_name_ptr := fnptr;} \\
\quad \quad \text{if (call_name_ptr.numparam} \neq \text{PARAMLIST.pno}) \\
\quad \quad \quad \text{error(‘mismatch in number of parameters in declaration and call’);} \\
\}
\]

\[ \text{PARAMLIST} \rightarrow \text{PLIST} \{\text{PARAMLIST.pno := PLIST.pno}\} \]
\[
\mid \epsilon \{\text{PARAMLIST.pno := 0}\} \]
$23 \quad PLIST \rightarrow E \{PLIST.pno := 1;
   \text{check}_\text{param}_\text{type}(\text{call}_\text{name}_\text{ptr}, 1, E.\text{type}, \text{ok});
   \text{if} (\neg \text{ok}) \text{error}(\text{‘parameter type mismatch in declaration and call’});\}

$24 \quad PLIST_1 \rightarrow PLIST_2 , E \{PLIST_1.pno := PLIST_2.pno + 1;
   \text{check}_\text{param}_\text{type}(\text{call}_\text{name}_\text{ptr}, PLIST_2.pno + 1, E.\text{type}, \text{ok});
   \text{if} (\neg \text{ok}) \text{error}(\text{‘parameter type mismatch in declaration and call’});\}$
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

```c
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 equivalence*
Semantic Analysis of Structs

- Names inside structs belong to a higher level
- Equivalence of structs is based on *name equivalence* and not on *structure equivalence*

```c
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
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;
}
```
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);
}

Y.N. Srikant  Semantic Analysis
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, \texttt{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 \texttt{int} type.
  - And the available overloaded functions are:
    - \texttt{int test(int a, int b, float c)} and
    - \texttt{int test(float a, int b, float c)}
  - In this case, we may choose the first one because it entails only one conversion from \texttt{int} to \texttt{float} (faster).
- If there is no match (or more than one match) even after conversions, an error is flagged.
SATG for 2-pass Sem. Analysis of Func. and Calls

- **FUNC_DECL** → **FUNC_HEAD** { **VAR_DECL** **BODY** }
  - **BODY** → { **VAR_DECL** **STMT_LIST** }
    - Variable declarations appear strictly before their use

- **FUNC_DECL** →
  - **FUNC_HEAD** { **VAR_DECL** **BODY** **VAR_DECL** }
  - **BODY** → { **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\textsuperscript{st} pass
- Declarations are all processed in the 1\textsuperscript{st} pass
- 1\textsuperscript{st} pass can be integrated with LR-parsing during which a parse tree is built
- Statements are analyzed in the 2\textsuperscript{nd} pass
- Sem. errors in statements are reported only in the 2\textsuperscript{nd} pass
- This effectively presents all the variable declarations before their use
- 2\textsuperscript{nd} pass can be made over the parse tree
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
The symbol table (ST) is indexed by block number.

In the previous version of the ST, there were no separate entries for blocks.

The surround block number (surr.blk.num) is the block number of the enclosing block.

All the blocks below a function entry $f$ in the ST, up to 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 the table using surround block numbers until the surround block number becomes zero.
```c
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 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`
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`).