COMPILER DESIGN
SUBJECT CODE: 303105349

Vishal Singh , Assistant Professor
Computer Science & Engineering

Parul® University

DIGITAL LEARNING CONTENT

CHAPTER-5

Semantic analysis

Contents

- Symbol tables and their data structures.
- Representation of “scope”.
- Semantic analysis of expressions, assignment, and control-flow statements, declarations of variables and using S- and L-attributed SDDs (treatment of arrays and structures included).
- Semantic error recovery

Symbol tables

•An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.

•These attributes may provide information about the **storage** allocated for a name, its **type**, its **scope** (where in the program its value may be used) in the case of procedure -names, such things as the number and types of its arguments, the **method of passing** each argument (for example, by value or by reference), and the **type** returned.

Symbol tables

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.
- The data structure should be designed to allow the compiler to find the record for each name **quickly** and to store or retrieve data from that record **quickly**.
- Lexical analyzer prepare this table. But all the attribute are not entered by lexical analyzer.
- For example: X, Y, Z declared as variable are stored in symbol table by lexical analyzer, and remaining phases enter information about identifier for further use.

Symbol tables

It is used by various phases of compiler as follows :-

- **Lexical Analysis**: Creates new table entries in the table, example like entries about token.
- **Syntax Analysis**: Adds information regarding attribute type, scope, dimension, line of reference, use, etc in the table.
- **Semantic Analysis:** Uses available information in the table to check for semantics i.e. to verify that expressions and assignments are semantically correct(type checking) and update it accordingly.
- **Intermediate Code generation:** Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
- **Code Optimization:** Uses information present in symbol table for machine dependent optimization.
- **Target Code generation:** Generates code by using address information of identifier present in the table.

Symbol tables

•**Symbol Table entries –** Each entry in symbol table is associated with attributes that support compiler in different phases.

Items stored in Symbol table:

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

Symbol tables

- **Information used by compiler from Symbol table:**
  - Data type and name
  - Declaring procedures
  - Offset in storage
  - If structure or record then, pointer to structure table.
  - For parameters, whether parameter passing by value or by reference
  - Number and type of arguments passed to function
  - Base Address

Symbol tables

- **Operations of Symbol table –** The basic operations defined on a symbol table include
  - lookup (name)
  - insert (name)
  - put (name, attribute, value)
  - get (name, attribute)
  - enterscope ()
  - exitscope()

Data structures for Symbol tables

- Following are commonly used data structure for implementing symbol table :-
  - List
  - Linked List
  - Hash Table
  - Binary Search Tree

Data structures for Symbol tables -List

- In this method, an array is used to store names and associated information.
- A pointer **“available”** is maintained at end of all stored records and new names are added in the order as they arrive
- To search for a name we start from beginning of list till available pointer and if not found we get an error **“use of undeclared name”**
- While inserting a new name we must ensure that it is not already present otherwise error occurs i.e. **“Multiple defined name”**
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average
- Advantage is that it takes minimum amount of space.

–

Data structures for Symbol tables - Linked List

- This implementation is using linked list. A link field is added to each record.
- Searching of names is done in order pointed by link of link field.
- A pointer **“First”** is maintained to point to first record of symbol table.
- Insertion is fast O(1), but lookup is slow for large tables – O(n) on average

Data structures for Symbol tables - Hash Table

- In hashing scheme two tables are maintained – a hash table and symbol table and is the most commonly used method to implement symbol tables..
- A hash table is an array with index range: 0 to tablesize – 1.These entries are pointer pointing to names of symbol table.
- To search for a name we use hash function that will result in any integer between 0 to tablesize – 1.
- Insertion and lookup can be made very fast – O(1).
- Advantage is quick search is possible and disadvantage is that hashing is complicated to implement.

Data structures for Symbol tables - Binary Search Tree

- Another approach to implement symbol table is to use binary search tree i.e. we add two link fields i.e. left and right child.
- All names are created as child of root node that always follow the property of binary search tree.
- Insertion and lookup are $O(\log_2 n)$ on average.

Representation of “scope”

- In the source program, every name possesses a region of validity, called the scope of that name.
- A compiler maintains two types of symbol tables: a **global symbol table** which can be accessed by all the procedures and **scope symbol tables** that are created for each scope in the program.
- To determine the scope of a name, symbol tables are arranged in hierarchical structure.

- The rules in a block-structured language are as follows:
- If a name declared within block B then it will be valid only within B.
- If B1 block is nested within B2 then the name that is valid for block B2 is also valid for B1 unless the name's identifier is re-declared in B1.

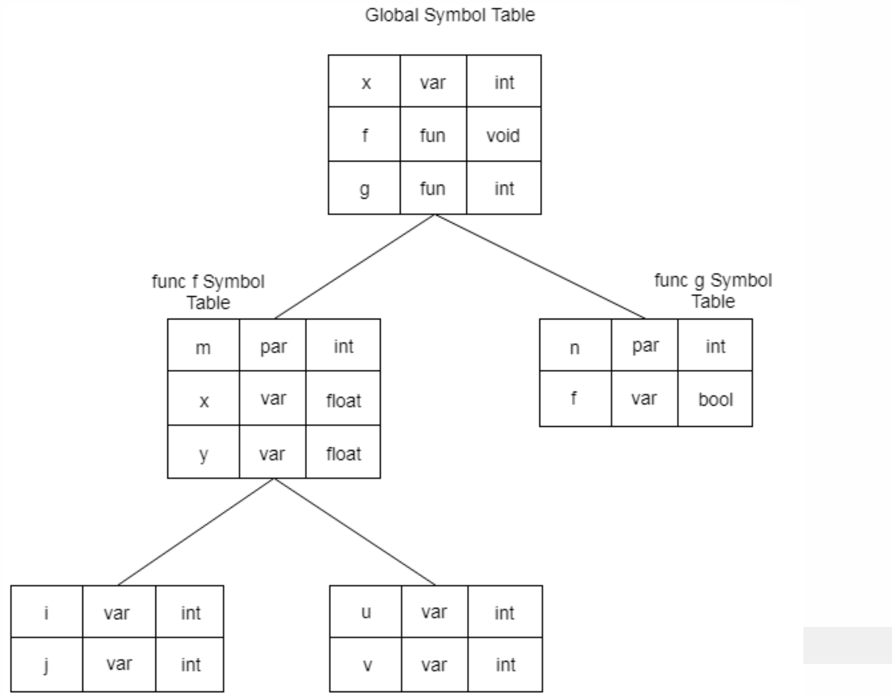
Representation of “scope”

- Tables are organized into stack and each table contains the list of names and their associated attributes.
- Whenever a new block is entered then a new table is entered onto the stack. The new table holds the name that is declared as local to this block.
- When the declaration is compiled then the table is searched for a name.
- If the name is not found in the table then the new name is inserted.
- When the name's reference is translated then each table is searched, starting from the each table on the stack.

Representation of “scope”- example

```
int x;
void f(int m) {
    float x, y;
{
        int i, j;
        int u, v;
}
}
int g (int n)
{
    bool t;
}
```

This program can be represented in a hierarchical structure of symbol tables:



Semantic analysis:

- **S-attributed definitions**- It uses only synthesized attributes, it is called as S-attributed SDT.
- S-attributed SDT can be evaluated in bottom up order of the nodes of the parse tree.
- **L-attributed definitions-** It uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
- So non-terminal can get values from its parent, child, and left sibling nodes.

SDD for expression grammar with synthesized attributes: S-attributed definition

Production	Semantic Rules
1) $L \rightarrow E\ \mathbf{n}$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Expression statement

Assignment statement

- Non-terminal can get values from its child.
- In parse tree the parent node E gets its value from its child node. Synthesized attributes never take values from their parent nodes or any sibling nodes.

SDD for expression grammar with inherited attributes: L-attributed definition

PRODUCTION	SEMANTIC RULE
$D \rightarrow TL$	$L.in := T.type$
$T \rightarrow$ int	$T.type :=$ *integer*
$T \rightarrow$ real	$T.type :=$ *real*
$L \rightarrow L_1$, id	$L_1.in := L.in$; *addtype*(id.*entry*, $L.in$)
$L \rightarrow$ id	*addtype*(id.*entry*, $L.in$)

Declarations of variables

- Synthesized: T.type
- Inherited: L.in

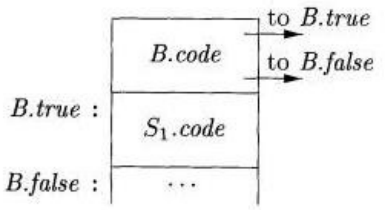
SDD for Control flow statements

- **Flow-of-Control Statements**
  - $S \rightarrow$ if (B) S_1
  - $S \rightarrow$ if (B) S_1 else s_2
  - $S \rightarrow$ while (B) S_1

In these productions, nonterminal *B* represents a Boolean expression and non-terminal S represents a statement.

SDD for Control flow statements

PRODUCTION	*SEMANTIC RULES*
S -> if(B)S1	*B.true = newlabelQ* *B.false = Si.next = S.next* *S.code = B.code || labelQB .true) || Si.code*



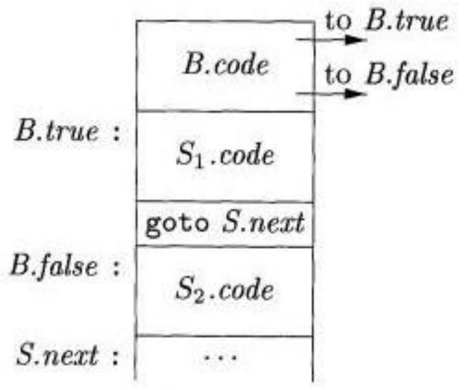
(a) if

- We assume that newlabelQ creates a new label each time it is call.
- Jumps to B.true within the code for B will go to the code for Si.
- Further, by setting B.false to S.next, we ensure that control will skip the code for Si if B evaluates to false.

SDD for Control flow statements

PRODUCTION	*SEMANTIC RULES*
s -> if (B) Si else S2	*B.true = newlabelQ* *B.false = newlabelQ* *Si.next = S2.next = S.next* *S.code = B.code* *|| label(B.true) \\ Si.code* *|| gen('goto' S.next)* *|| label(B.false) \\ S^.code*

SDD for Control flow statements



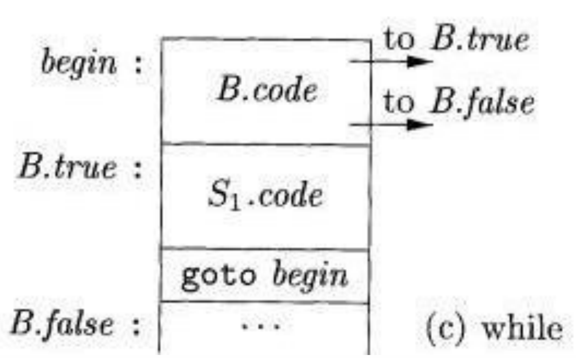
(b) if-else

- The code for the boolean expression *B* has jumps out of it to the first instruction of the code for Si if *B* is true, and to the first instruction of the code for S_2 if *B* is false.
- Further, control flows from both *Si* and S2 to the instruction immediately following the code for *S* — its label is
- given by the inherited attribute *S.next.*
- An explicit **goto** *S.next* appears after the code for Si to skip over the code for S2 .
- No goto is needed after S2 , since *Si.next* is the same as *S.next.*

SDD for Control flow statements

PRODUCTION	*SEMANTIC RULES*
S-> while (B) Si	*begin — newlabelQ* *B.true = newlabelQ* *B.false = S.next* *Si.next — begin* *S.code — label(begin) | | B.code* *| | label(B.true) \\\\ Si.code* *| | geniigoto' &e#m)*

SDD for Control flow statements



- *Si* is formed from *B.code* and *Si.code.*
- We use a local variable *begin* to hold a new label attached to the first instruction for this while-statement, which is also the first instruction for *B.*
- We use a variable rather than an attribute, because *begin* is local to the semantic rules for this production.
- The inherited label *S.next* marks the instruction that control must flow to if *B* is false; hence, *B.false* is set to be *S.next.* A new label *B.true* is attached to the first instruction for Si; the code for *B* generates a jump to this label if *B* is true.
- After the code for Si we place the instruction **goto** *begin,* which causes a jump back to the beginning of the code for the boolean expression.
- Note that *Si.next* is set to this label *begin,* so jumps from within *Si.code* can go directly to *begin.*

Semantic error recovery

- These errors are detected during semantic analysis phase. Typical semantic errors are
  - Incompatible type of operands
  - Undeclared variables
  - Not matching of actual arguments with formal one

```
Example : int a[10], b;
          .......
          .......
          a = b;
```

It generates a semantic error because of an incompatible type of a and b.

Semantic error recovery

- **Error recovery**

- If error **"Undeclared Identifier"** is encountered then, to recover from this a symbol table entry for corresponding identifier is made.
- If data types of two operands are incompatible then, automatic type conversion is done by the compiler.