PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

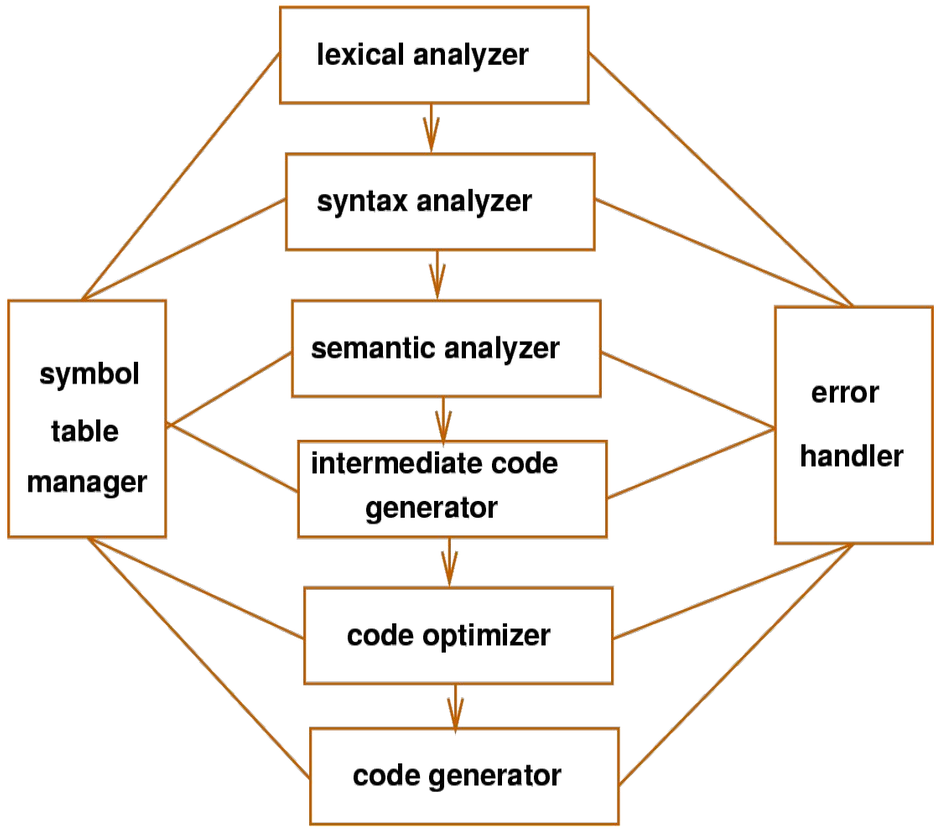
Building a Mini Compiler - Generating Symbol Table

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Introduction



- A symbol table is an important data structure used in compilers to store the information of various entities(variable names, function names, objects, classes, etc). The symbol table is created and maintained by the compiler and is used by both the analysis and synthesis parts of the compiler.

Uses

- Store the names of all entities in a structured form in one place.
- Verify if a variable has been declared.
- Determine the scope of an entity.
- Implement type checking by verifying if assignments and expressions in the source code are semantically correct.

Possible Implementation Techniques

- Linear Lists : Simple and straightforward.
- Binary Search Tree : Efficient in organizing symbol tables.
- Hash Table : Searching is faster and easier.

Compiler Design

Importance of $$

- $$, $1, $2 etc are the semantic values for the symbols and tokens used in the rule in the order they appear.
- The value type is defined by YYSTYPE.
- #define YYSTYPE char* shows that $$, $1 etc will hold string values
- Example: exp: a ‘+’ b {$$=$1+$3}
- Here $1 holds the value for a, $2 is for ‘+’, $3 is for b.
- $$ holds the result value for the grammar rule, which is exp here.

Compiler Design

Implementation tasks

- Creation of a symbol table which contains name of variable, type,storage required, line number, value and scope.
- Insertion and updation of value, line number and scope will be taken care of in the next lab

Expected Results:

Input:

```
int main() {
    int a;
    float b;
    double c;
    char d;
}
```

Sample Output

```
Valid syntax
Name    size    type    lineno  scope   value
a       2       2       3       1       ~
b       4       3       4       1       ~
c       8       4       5       1       ~
d       1       1       6       1       ~
```

(where 1=char, 2=int, 3=float, 4=double)

Symbol table implementation:header file

- Create a header file to define your symbol table structure and its functions.
- Taking the linked list implementation as an example.
  - Define the node structure and for our case the data items will be the data to be displayed on the symbol table.
  - Define the head structure to point to the first node in the list.

- Functions can change based on your implementation and your needs but the most important functions you would need would be
  - Insertion of an entry
  - Updating entries
  - Allocation of space for the head and node
  - Displaying the symbol table

Compiler Design

Mini-Compiler

```
//Structure for a single entry in the list
typedef struct (struct_name)//replace with meaningful name. eg:item,symbol,node,entry
{
    /*
        define the variables to be displayed in the symbol table
        name, value, type, scope, size, line number

    */
    struct (struct_name)* next; //link to the next entry on the list

}(s_name);//typedef will make this "name" a new user defined data type
        //name can be anything meaningful

//structure that keeps track of the start of the list
typedef struct (table_name) //replace with meaningful name. eg:table,list
{
    (s_name)* head;  //points to the first entry
}(t_name);
```

Symbol table implementation:code file

- Create the .c file to implement the functions defined.
- Include the header file created.
- Create the respective functions defined in your header file.
- Important points for implementation:
  - Make sure to allocate space for a node during insertion, this includes the data items in the nodes containing pointers.(Linked list implementation)
  - Check whether if the entries already exist.
- Tips:
  - Default values could be assigned for variables in the node that will get updated later(eg:value) and for checking purpose.

Mini-Compiler


```
(t_name)* allocate_space_for_table()
{
    /*
        allocate space for table pointer structure eg (t_name)* t
        initialise head variable eg t->head
        return structure
    */
}
```

```
(s_name)* allocate_space_for_table_entry(s_name variables)
{
    /*
        allocate space for entry pointer structure eg (s_name)* s
        initialise all struct variables(name, value, type, scope, length, line number)
        return structure
    */
}
```

Mini-Compiler


```
insert_into_table(arguments)/*
 arguments can be the structure s_name already allocated before this function call
 or the variables to be sent to allocate_space_for_table_entry for initialisation
*/
{
    /*
        check if table is empty or not using the struct table pointer
        else traverse to the end of the table and insert the entry
    */
}

display_symbol_table()
{
    /*
        traverse through table and print every entry
        with its struct variables
    */
}
```

Mini-Compiler


```
insert_value_to_name(name,value)
{
    /*
        if value is default value return back
        check if table is empty
        else traverse the table and find the name
        insert value into the entry structure
    */
}
```

Symbol table implementation: yacc file

- Include the symbol table file.
- Declare variables to keep track of scope or store values for your structure before insertion.
- Insertion into the table can be done when declaring the variable.

Compiler Design

Mini-Compiler

```
DECLR : TYPE LISTVAR
LISTVAR : LISTVAR ',' VAR
        | VAR
        ;
VAR: T_ID '=' EXPR  {
                    /*
                    Explained in lab 3
                    */
        }
   | T_ID           {
                    /*
                    check if symbol is in table
                    if it is then error for redeclared variable
                    else make entry and insert into table
                    revert variables to default values:type
                    */
        }
```


```
//assign type here to be returned to the declaration grammar
TYPE : T_INT
       | T_FLOAT
       | T_DOUBLE
       | T_CHAR
```

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu