COMPILER (CSE 4120)

(Lecture 9: Semantic Analysis 2)

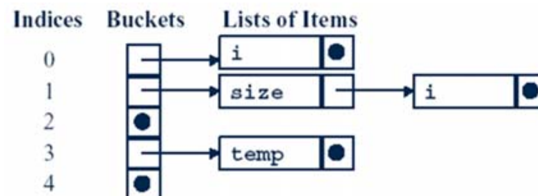
Sungwon Jung, Ph.D.

Mobile Computing and Data Engineering Lab.
Dept. of Computer Science and Engineering
Sogang University
Seoul, Korea
Tel: +82-2-705-8930
Email : jungsung@sogang.ac.kr

Dept. of Computer Science at Sogang University

The Symbol Table

- The structure of the Symbol Table
  - Separate chained hash table showing collision resolution



  - Symbol table operations
    - Insert
    - Lookup
    - Delete

```
typedef struct BucketListRec
   { char * name;
     LineList lines;
     int memloc ; /* memory location */
     struct BucketListRec * next;
   } * BucketList;

/* the hash table */
static BucketList hashTable[SIZE];
```

Hash Function for a Symbol Table

- To repeatedly use a constant number α(SHIFT) as a multiplying factor when adding in the value of the next character
  - c_i = the numeric value of the ith character

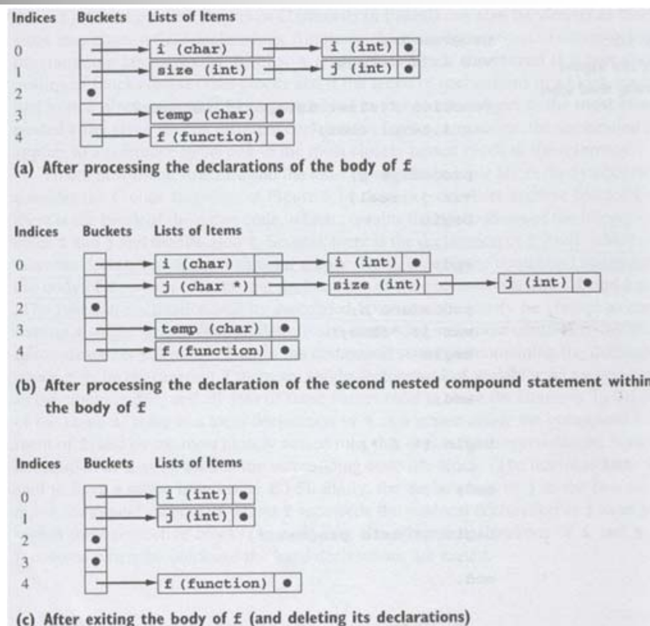
$$h = (\alpha^{n-1}c_1 + \alpha^{n-2}c_2 + \ldots + \alpha^{1}c_n) \bmod size = \left(\sum_{i=1}^{n} \alpha^{n-i}c_i\right) \bmod size$$

```
/* SIZE is the size of the hash table */
#define SIZE 211

/* SHIFT is the power of two used as multiplier
   in hash function  */
#define SHIFT 4

/* the hash function */
static int hash ( char * key )
{ int temp = 0;
  int i = 0;
  while (key[i] != '\0')
  { temp = ((temp << SHIFT) + key[i]) % SIZE;
    ++i;
  }
  return temp;
}
```


Scope Rules and Block Structure

- C code fragment illustrating nested scope and symbol table contents at various places in the code

```
int i, j;
int f(int size)
{ char i, temp;
  ...
   { double j;
     ...
   }
   ...
   { char *j;
     ...
   }
}
```



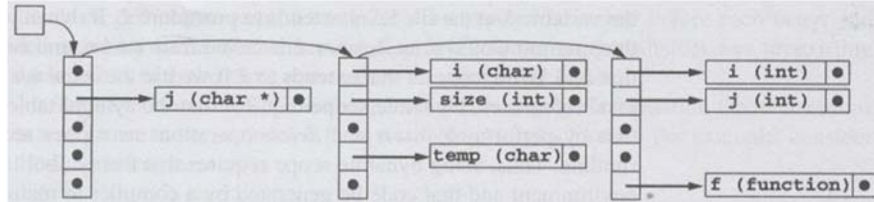
(a) After processing the declarations of the body of f

(b) After processing the declaration of the second nested compound statement within the body of f

(c) After exiting the body of f (and deleting its declarations)

Scope Rules and Block Structure

- Symbol table structure using separate tables for each scope



- Static scope or lexical scope
- Dynamic scope

```
#include <stdio.h>

int i = 1;

void f(void)
{ printf("%d\n",i);}

void main(void)
{ int i = 2;
  f();
  return 0;
}
```


Dept. of Computer Science at Sogang University

Symbol Table Interfaces

- void st_insert (char *name, int lineno, int loc);
  - inserts line numbers and memory locations into the symbol table
  - loc = memory location is inserted only the first time, otherwise ignored
- int st_lookup (char *name);
  - returns the memory location of a variable or -1 if not found
- void printSymTab (FILE *listing);
  - prints a formatted listing of the symbol table contents to the listing file

st_insert

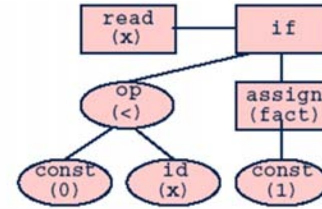
```
void st_insert( char * name, int lineno, int loc )
{ int h = hash(name);
  BucketList l = hashTable[h];
  while ((l != NULL) && (strcmp(name,l->name) != 0))
    l = l->next;
  if (l == NULL) /* variable not yet in table */
  { l = (BucketList) malloc(sizeof(struct BucketListRec));
    l->name = name;
    l->lines = (LineList) malloc(sizeof(struct LineListRec));
    l->lines->lineno = lineno;
    l->memloc = loc;
    l->lines->next = NULL;
    l->next = hashTable[h];
    hashTable[h] = l; }
  else /* found in table, so just add line number */
  { LineList t = l->lines;
    while (t->next != NULL) t = t->next;
    t->next = (LineList) malloc(sizeof(struct LineListRec));
    t->next->lineno = lineno;
    t->next->next = NULL;
  }
} /* st_insert */
```


st_lookup

```
int st_lookup ( char * name )
{ int h = hash(name);
  BucketList l =  hashTable[h];
  while ((l != NULL) && (strcmp(name,l->name) != 0))
    l = l->next;
  if (l == NULL) return -1;
  else return l->memloc;
}
```

Building Symbol Table

- buildSymtab() constructs the symbol table by preorder traversal of the syntax tree



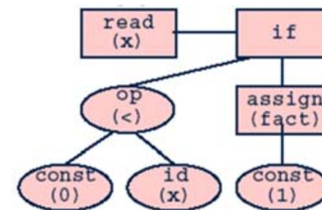

```
void buildSymtab(TreeNode * syntaxTree)
{ traverse(syntaxTree,insertNode,nullProc);
  if (TraceAnalyze)
  { fprintf(listing,"\nSymbol table:\n\n");
    printSymTab(listing);
  }
}
```

traverse()

- generic recursive syntax tree traversal routine:
- it applies preProc in preorder and postProc in postorder to tree pointed to by t

```
static void traverse( TreeNode * t,
               void (* preProc) (TreeNode *),
               void (* postProc) (TreeNode *)
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc,postProc);
    }
    postProc(t);
    traverse(t->sibling,preProc,postProc);
  }
}
```


insertNode

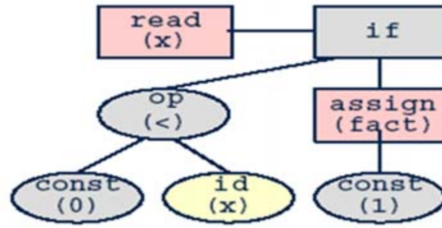



```
static void insertNode( TreeNode * t )
{ switch (t->nodekind)
  { case StmtK:
      switch (t->kind.stmt)
      { case AssignK:
        case ReadK:
          if (st_lookup(t->attr.name) == -1)
          /* not yet in table, so treat as new definition */
            st_insert(t->attr.name,t->lineno,location++);
          else
          /* already in table, so ignore location,
             add line number of use only */
            st_insert(t->attr.name,t->lineno,0);
          break;
        default:
          break;
      }
      break;
```

```
/* Procedure insertNode inserts
 * identifiers stored in t into
 * the symbol table
 */
```

insertNode (Cont'd)

```
case ExpK:
  switch (t->kind.exp)
  { case IdK:
      if (st_lookup(t->attr.name) == -1)
      /* not yet in table, so treat as new definition */
        st_insert(t->attr.name,t->lineno,location++);
      else
      /* already in table, so ignore location,
         add line number of use only */
        st_insert(t->attr.name,t->lineno,0);
      break;
    default:
      break;
  }
  break;
default:
  break;
 }
}
```



Dept. of Computer Science at Sogang University

Example of Symbol Table

```
 1: { Sample program
 2:    in TINY language -
 3:    computes factorial
 4: }
 5: read x;
 6: if 0 < x then { don't compile if x <= 0 }
 7:    fact := 1;
 8:    repeat
 9:       fact := fact * x;
10:       x := x - 1
11:    until x = 0;
12:    write fact { output factorial of x }
13: end
```

Variable Name	Location	Line Numbers
x	0	5 6 9 10 10 11
fact	1	7 9 9 12