Name Equivalence

- Two types are name equivalent if they have same name or label

```
typedef int Value
typedef int Total
...
Value var1, var2
Total var3, var4
```

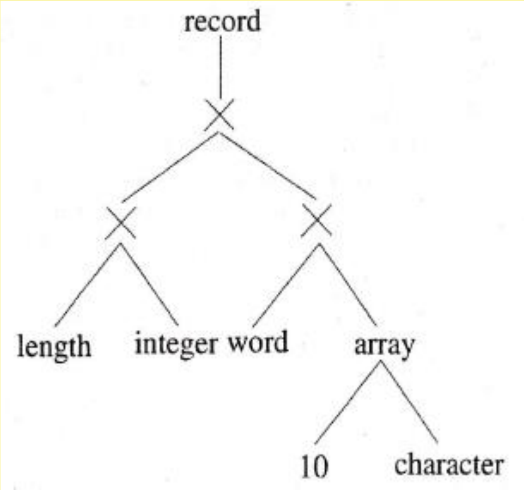
- Variables var1, var2 are name equivalent, so are var3 and var4
- Variables var1 and var4 are not name equivalent, as their type names are different

Structural Equivalence

- Checks the structure of the type
- Determines equivalence by checking whether they have same constructor applied to structurally equivalent types
- Checked recursively
- Types array(I1, T1) and array(I2, T2) are structurally equivalent if I1 and I2 are equal and T1 and T2 are structurally equivalent

Directed Acyclic Graph Representation

- Type expressions can be represented as a DAG or a tree
- “record((length × integer) × (word × array(10, character)))”



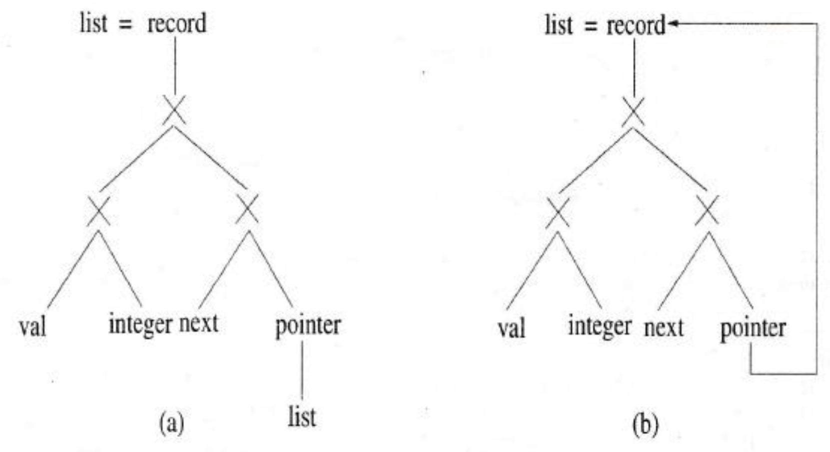
Function dag_equivalence

```
function dag-equivalence(s,t: type-DAGs): boolean
begin
    if s and t represents the same basic type then return true
    if s represents array(I₁, T₁) and t represents array(I₂, T₂) then
        if I₁ = I₂ then return dag-equivalence(T₁, T₂)
        else return false
    if s represents s₁ × s₂ and t represents t₁ × t₂ then
        return dag-equivalence(s₁, t₁) and dag-equivalence(s₂, t₂)
    if s represents pointer(s₁) and t represents pointer(t₁) then
        return dag-equivalence(s₁, t₁)
    if s = s₁ → s₂ and t = t₁ → t₂ then
        return dag-equivalence(s₁, t₁) and dag-equivalence(s₂, t₂)
    return false
end.
```

Cycles in Type Representation

- Some languages allow types to be defined in a cyclical fashion

```
struct list
{
        int val;
        struct list *next;
}
```



- (a) Acyclic representation   (b) Cyclic representation

Cycles in Type Representation

- Most programming languages, including C, uses acyclic one
- Type names are to be declared before using it, excepting pointers
- Name of the structure is also part of the type
- Equivalence test stops when a structure is reached
- At this point, type expressions are equivalent if they point to the same structure name, nonequivalent otherwise

Type Conversion

- Refers to local modification of type for a variable or subexpression
- For example, it may be necessary to add an integer quantity to a real variable, however, the language may require both the operands to be of same type
- Modifying integer variable to real will require more space
- Solution: to treat integer operand as really operand locally and perform the operation
- May be done explicitly or implicitly
- Implicit conversion ➔ type coercion

```
int x;
float y;
...
y = ((float)x)/14.0
```

```
int x;
float y;
...
y = x/14.0
```

Conclusion

- Compilers usually perform static type checking
- Dynamic type checking is costly
- Types are normally represented as type expressions
- Type checking can be performed by syntax directed techniques
- Type graphs may be compared to check type equivalence

FREE ONLINE EDUCATION

swayam

शिक्षित भारत, उन्नत भारत

NPTEL ONLINE CERTIFICATION COURSES

Thank you!

Compiler Design

Symbol Tables

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

- Information in Symbol Table
- Features of Symbol Table
- Simple Symbol Table
- Scoped Symbol Table
- Conclusion

Introduction

- Essential data structure used by compilers to remember information about identifiers in the source program
- Usually lexical analyzer and parser fill up the entries in the table, later phases like code generator and optimizer make use of table information
- Types of symbols stored in the symbol table include variables, procedures, functions, defined constants, labels, structures etc.
- Symbol tables may vary widely from implementation to implementation, even for the same language

Information in Symbol Table

- Name
  - Name of the identifier
  - May be stored directly or as a pointer to another character string in an associated string table – names can be arbitrarily long
- Type
  - Type of the identifier: variable, label, procedure name etc.
  - For variables, its type: basic types, derived types etc.
- Location
  - Offset within the program where the identifier is defined
- Scope
  - Region of the program where the current definition is valid
- Other attributes: array limits, fields of records, parameters, return values etc.

Usage of Symbol Table Information

- Semantic Analysis – check correct semantic usage of language constructs, e.g. types of identifiers
- Code Generation – Types of variables provide their sizes during code generation
- Error Detection – Undefined variables. Recurrence of error messages can be avoided by marking the variable type as undefined in the symbol table
- Optimization – Two or more temporaries can be merged if their types are same

Operations on Symbol Table

- Lookup – Most frequent, whenever an identifier is seen it is needed to check its type, or create a new entry
- Insert – Adding new names to the table, happens mostly in lexical and syntax analysis phases
- Modify – When a name is defined, all information may not be available, may be updated later
- Delete – Not very frequent. Needed sometimes, such as when a procedure body ends

Issues in Symbol Table Design

- Format of entries – Various formats from linear array to tree structured table
- Access methodology – Linear search, Binary search, Tree search, Hashing, etc.
- Location of storage – Primary memory, partial storage in secondary memory
- Scope Issues – In block-structured language, a variable defined in upper blocks must be visible to inner blocks, not the other way

Simple Symbol Table

- Works well for languages with a single scope
- Commonly used techniques are
  - Linear table
  - Ordered list
  - Tree
  - Hash table

Linear Table

- Simple array of records with each record corresponding to an identifier in the program
- Example:

```
int x, y
real z
...
procedure abc
...
L1:...
...
```

Name	Type	Location
x	integer	Offset of x
y	integer	Offset of y
z	real	Offset of z
abc	procedure	Offset of abc
L1	label	Offset of L1

Linear Table

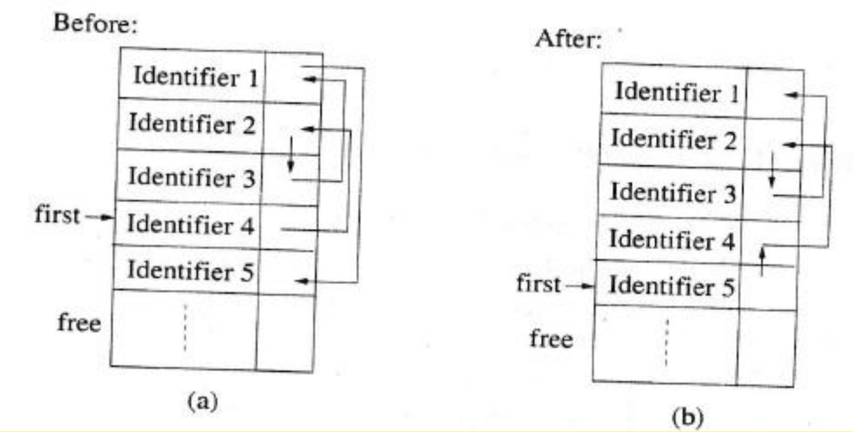
- If there is no restriction in the length of the string for the name of an identifier, string table may be used, with name field holding pointers
- Lookup, insert, modify take O(n) time
- Insertion can be made O(1) by remembering the pointer to the next free index
- Scanning most recent entries first may probably speed up the access – due to program locality – a variable defined just inside a block is expected to be referred to more often than some earlier variables

Ordered List

- Variation of linear tables in which list organization is used
- List is sorted in some fashion , then binary search can be used with O(log n) time
- Insertion needs more time
- A variant – self-organizing list: neighbourhood of entries changed dynamically

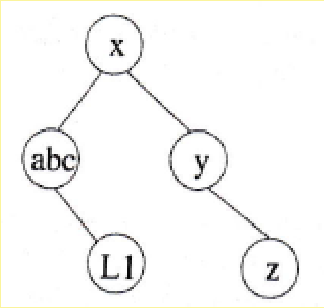
Self-Organizing List

- In Fig (a), Identifier4 is the most recently used symbol, followed by Identifier2, Identifier3 and so on
- In Fig (b), Identifier5 is accessed next, accordingly the order changes
- Due to program locality, it is expected that during compilation, entries near the beginning of the ordered list will be accessed more frequently
- This improves lookup time



Tree

- Each entry represented by a node of the tree
- Based on string comparison of names, entries lesser than a reference node are kept in its left subtree, otherwise in the right subtree
- Average lookup time O(log n)
- Proper height balancing techniques need to be utilized



Hash Table

- Useful to minimize access time
- Most common method for implementing symbol tables in compilers
- Mapping done using Hash function that results in unique location in the table organized as array
- Access time O(1)
- Imperfection of hash function results in several symbols mapped to the same location – collision resolution strategy needed
- To keep collisions reasonable, hash table is chosen to be of size between n and 2n for n keys

Desirable Properties of Hash Functions

- Should depend on the name of the symbol. Equal emphasis be given to each part
- Should be quickly computable
- Should be uniform in mapping names to different parts of the table. Similar names (such as, data1 and data2) should not cluster to the same address
- Computed value must be within the range of table index

Scoped Symbol Table

- Scope of a symbol defines the region of the program in which a particular definition of the symbol is valid – definition is *visible*
- Block structured languages permit different types of scopes for the identifiers – *scope rules* for the language
  - Global scope: visibility throughout the program, global variables
  - File-wide scope: visible only within the file
  - Local scope within a procedure: visible only to the points inside the procedure, local variables
  - Local scope within a block: visible only within the block in which it is defined

Scoping Rules

- Two categories depending on the time at which the scope gets defined
- Static or Lexical Scoping
  - Scope defined by syntactic nesting
  - Can be used efficiently by the compiler to generate correct references
- Dynamic or Runtime Scoping
  - Scoping depends on execution sequence of the program
  - Lot of extra code needed to dynamically decide the definition to be used

Nested Lexical Scoping

- To reach the definition of a symbol, apart from the current block, the blocks that contain this innermost one, also have to be considered
- Current scope is the innermost one
- There exists a number of open scopes – one corresponding to the current scope and others to each of the blocks surrounding it

```
Procedure P1
…
  Procedure P2
  …
  end procedure
Procedure P3
  x =
  …
```

Current scope of x is P3, it has another open scope P1

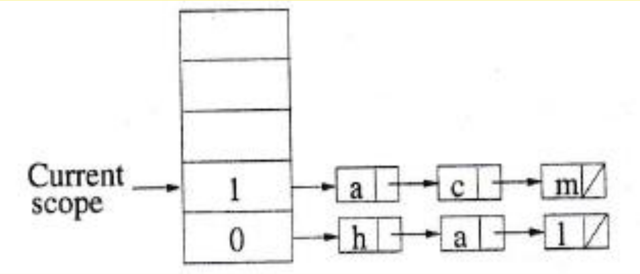
Visibility Rules

- Used to resolve conflicts arising out of same variable being defined more than once
- If a name is defined in more than one scope, the innermost declaration closest to the reference is used to interpret
- When a scope is exited all declared variables in that scope are deleted and the scope is thus *closed*
- Two methods to implement symbol tables with nested scope
  - One table for each scope
  - A single global table

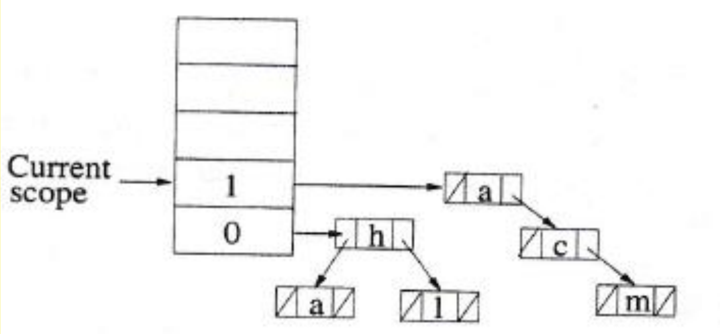
One Table Per Scope

- Maintain a different table for each scope
- A stack is used to remember the scopes of the symbol tables
- Drawbacks:
  - For a single-pass compiler, table can be popped out and destroyed when a scope is closed, not for a multi-pass compiler
  - Search may be expensive if variable is defined much above in the hierarchy
  - Table size allotted to each block is another issue
- Lists, Trees, Hash Tables can be used

One Table Per Scope



Scoped Symbol Table – Lists

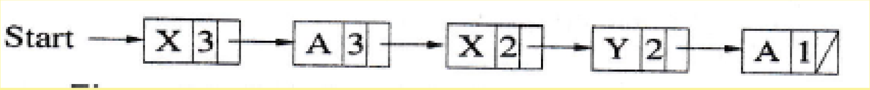


Scoped Symbol Table – Trees

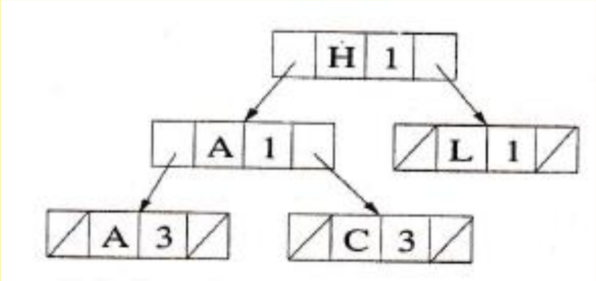
One Table for All Scopes

- All identifiers are stored in a single table
- Each entry in the symbol table has an extra field identifying the scope
- To search for an identifier, start with the highest scope number, then try out the entries having next lesser scope number, and so on
- When a scope gets closed, all identifiers with that scope number are removed from the table
- Suitable particularly for single-pass compilers
- List, Tree and Hash Table can be used

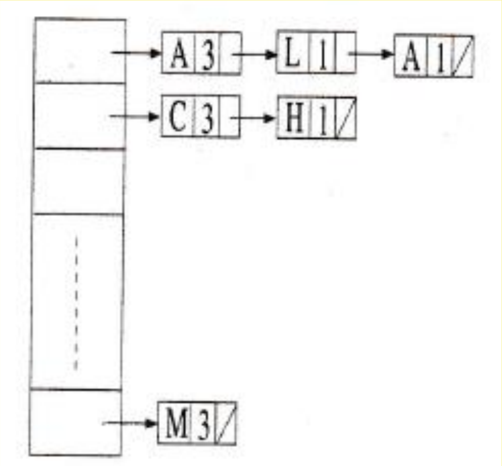
One Table for All Scopes



Single List



Tree



Hash Table

Conclusions

- Symbol table, though not part of code generated by the compiler, helps in the compilation process
- Phases like Lexical Analysis and Syntax Analysis produce the symbol table, while other phases use its content
- Depending upon the scope rules of the language, symbol table needs to be organized in various different manners
- Data structures commonly used for symbol table are linear table, ordered list, tree, hash table, etc.

Thank you

Compiler Design

Runtime Environment Management

Santanu Chattopadhyay
Electronics and Electrical Communication Engineering

- What is Runtime Environment
- Activation Record
- Environment without Local Procedures
- Environment with Local Procedures
- Display
- Conclusion

What is Runtime Environment

- Refers to the program snap-shot during execution
- Three main segments of a program
  - Code
  - Static and global variables
  - Local variables and arguments
- Memory needed for each of these entities
  - Generated code: Text for procedures and programs. Size known at compile time. Space can be allotted statically before execution
  - Data objects:
    - Global variables/constants – space known at compile time
    - Local variables – space known at compile time
    - Dynamically created variables – space (heap) in response to memory allocation requests
  - Stack: To keep track of procedure activations

Logical Address Space of Program

Code	Static	Heap	⟵ Free Memory ⟶	Stack

Low High

- Code occupies the lowest portion
- Global variables are allocated in the static portion
- Remaining portion of the address space, stack and heap are allocated from the opposite ends to have maximum flexibility

Activation Record

- Storage space needed for variables associated with each activation of a procedure – *activation record* or *frame*
- Typical activation record contains
  - Parameters passed to the procedure
  - Bookkeeping information, including return values
  - Space for local variables
  - Space for compiler generated local variables to hold sub-expression values