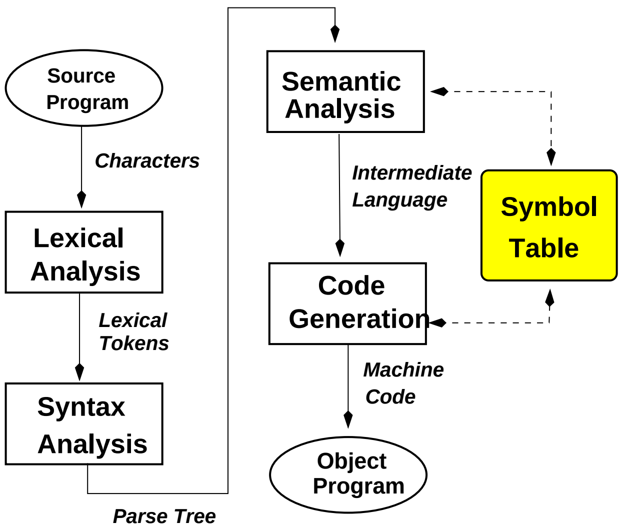
CSC 488S/CSC 2107S Lecture Notes

Reading Assignment

Fischer, Cytron and LeBlanc

Chapter 8

Omit 8.5, 8.6, 8.7, 8.8, 8.9



Compiler Tables

- The *Symbol Table* contains information about declared things (usually identifiers).
- A *Type Table* is used to record information about builtin and user declared types. Necessary for languages that allow arbitrary user declared types.
- Efficient table management is often a major performance issue in the design of a compiler. Table search is the dominant operation.
- Table lookup must follow the languages scope rules.
- Table usage
  - Declaration processing - add entries for identifiers and types being declared
  - Semantic analysis - look up identifiers to determine their attributes.
    Look up types to vaildate program usage.
  - Code generation - look up identifiers to determine their attributes.

- Table organization
  - One global table for entire program.
  - Separate linked tables for each major scope
- Table storage
  - Fixed size memory resident, limits program size
  - Partially memory resident, remainder on disk.
    LRU caching works well.
  - Statically or dynamically allocated.

Generic Symbol Table Entries

- A typical symbol table entry might contain:
  - Name of the item
  - Kind of item, e.g. constant, variable, type, procedure, function, etc.
  - Type of the item (index into type table)
  - Attributes or properties associated with the item (usually language specific)
  - Size of the item (or derive from type)
  - Run time address for the item, e.g. base register and offset.
  - Value for the item, e.g. value of constants, initial value for variables (might be index into a table of constants)
  - Links to related symbols, e.g. parameter lists, enumerated constants, fields in a struct or union.

  These are the attributes of symbols that might be different for each symbol.
- Usually one symbol table entry per declared symbol, low hundreds for student, toy compilers, in the thousands for production compilers.

Symbol Table Operations

The following operations are typical of a symbol table class/module

- Create Symbol Table
- Enter new scope
- Exit Scope
- Lookup symbol in current scope
- Lookup symbol using scope rule
- Enter new symbol in current scope
- Enter new symbol in designated scope
- Delete symbol from table
- Retain symbol in symbol table
- For all fields in the symbol table
  - Set value of field for symbol
  - Get value of field for symbol
  - Modify value of field for symbol
- Language specific operations

Generic Type Table Entry

- A typical type table entry might contain:
  - Name of the item (often omitted)
  - Kind of type e.g. typedef, enum, scalar, array, struct, union, etc.
    Language specific.
  - Attributes or properties associated with the type.
    e.g packed, read only, etc.
  - Size and memory alignment for objects of this type.
  - Actual definition of the type, usually involves links to embedded components.
  - Links to related definitions.
  - For many modern languages, the type table is a directed acyclic graph.

These are the attributes that might be different for each distinct type.

- Example: array type
  - Number of dimensions, Size and allignment information
  - Link to list of array bounds, Link to element type

array	dim=3	size=16000	align=4		

bounds element

- Example: record/structure
  - Number of fields, Size/alignment information
  - Link to symbol table entry for first field
    Fields form a linked list in the symbol table

record	nFields=23	size=1440	align=8	

fields

- Example: function or procedure
  - Number of parameters, entry point
  - Link to return type of function, NULL for procedure
  - Link to symbol table entry for first parameter
    Parameters form linked list in symbol table.

funcproc	nParams=3	entryAt 0X00640		

return type parameters

Symbol and Type Table Example

typedef
struct str {
int a ;
char * b ;
} T ;

int i, ia[20] ;
const int c = 23 ;

T A, B[100], * TP ;

name	kind	value	type
i	var		
ia	var		
c	const	23	
str	struct		
a	field		
b	field		
T	type		
A	var		
B	var		
TP	var		

type	data / links	
int	builtin	
char	builtin	
array		
range	0	19
struct		
pointer		
array		
range	0	99
pointer		

Handling Related Entries

- There are cases where the compiler needs to keep track of related symbol and/or type table entries.
  Examples: fields in a struct, the parameters of a procedure or function, the names in an enumerated type.
- One approach is to assign consecutive table entries to related items so that the index of the first item and an offset is sufficient to locate any item.
  This approach may be a considerable pain to implement if embedding in the language requires that table entries be build from the inside out, e.g. struct definition containing an embedded struct definition.
- Another approach is to use explicit links (pointers or array indices) in the table structure. The related items are kept as a linked list usually in the symbol table. This costs table space, but is easier to build during declaration processing.
  This is the preferred approach for modern compilers.

Handling Values

- The compiler needs to be able to represent in its tables any kind of *value*, i.e. constant, that could occur in a program.
- To conserve table space, some form of union data structure is often used:

```
struct constantDesc {
     short  constantKind ;
     union  {
            int    intValue ;
            float  floatValue ;
            char   charValue ;
            char  *stringValue ;
            void  *bigValue ;
     } constantValue ;
  }
```

- Large constant values, e.g. initialization for arrays or structs often require special storage.

Scopes and Declarations

- Assume that a program consists of some number of nested *scopes* of declarations (e.g. main program, **begin** - **end** blocks, functions and procedures).
- Identifiers are declared in one or more scopes.
- The *scope rule* for the programming language determines the visibility of of symbols declared in one scope in other scopes.
- The most common scope rule is the *Algol-60 scope rule* which allows identifiers declared in a scope to be automatically visible in all contained (properly nested) scopes.
- Languages with modules, objects, packages or classes often have different visibility rules for identifier declared within those constructs.
(e.g. C++ classes allow declared identifiers to be **public**, **protected**, or **private**)

Scope Rules and Tables

- In most programming languages a program is partitioned into some (possibly overlapping) scopes of declaration.
- The scope rule for a language specifies the algorithm that must be used to search compiler tables.
- To implement most scope rules, each scope of declaration is *logically* a separate set of symbol table entries.
It is often useful to handle major scopes (main program, classes, routines) differently from minor scopes (bodies of **if** and **for** statements).
- For the most common (Algol-60) scope rule, scopes are properly nested and the symbol table behaves in a strictly stack-like fashion.
- Most compilers distinguish symbol *visibility* from symbol *storage*. The scope rule determines which symbols are visible at any point in the program. Symbols may remain in the symbol table even if they are not visible (e.g. entries for the parameters of a function).

Handling Major and Minor Scopes Efficiently

Major Scope A major scope corresponds to a construct with special significance in the programming language.
Examples: main program, body of a class, body of a routine.
A major scope has significance beyond symbol table lookup, it is also usually a unit for storage allocation.

Minor scope A minor scope is a small scope of less significance that occurs within a major scope.
Example: scopes created within statements using { and } .

Although minor scopes could be handled the same way as major scopes, this is not necessary and costs a lot of unnecessary scope manipulation effort. It is simpler to merge the symbol table entries for minor scopes into the symbol table for the nearest enclosing major scope. Visibility of minor scopes is handled by managing the symbol lookup algorithm.

This merging up of minor scopes also makes allocation of storage for variables much easier.

Condensed Scopes in Running Example Program

```
/* Turing program to find prime numbers <= 10,000 */
const n := 10000
var sieve, primes : array 2 .. n of boolean
var next : int := 2
var __k1 , __k2 : int
for __k1 := 2 .. n /* initialize */
    sieve[ __k1 ] := true primes[ __k1 ] := false
end for
loop
    exit when next > n
    primes[ next ] := true
    for __k2 : next .. n by next
        sieve[ __k2 ] := false
    end for
    loop /* find next prime */
exit when next > n or sieve[ next ]
next += 1      end loop
end loop
/* prime[i] is true only for prime numbers */
```

Symbol Table for Running Example Program (See Slide 172)

Name	Kind	Type	Scope
n	const	int	1
sieve	var	array,boolean	1
primes	var	array,boolean	1
next	var	int	1
__k1	loopvar	int	1a
__k2	loopvar	int	1b

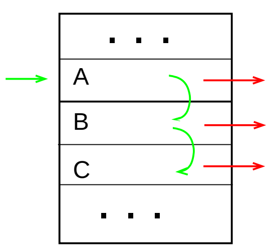
Minor scopes have been moved up to enclosing major scope.

Retained Symbols

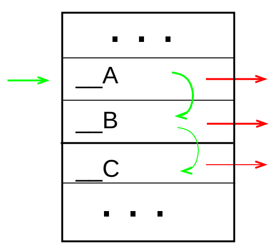
- The formal parameters of functions and procedures need special symbol table handling.
- The symbol table entries are created when the prototype or actual definition of the routine is processed.
- The entries are used during processing the body of the routine.
- After the routine has been processed, the symbol table entries for the formal parameters of the routine need to be retained in the symbol table so they can be used to process *calls* of the routine.
- The usual technique is to leave the formal parameter entries in the symbol table linked to the routine entry, but with the parameters marked so they won't be seen by ordinary symbol table lookup.

Retained Symbols Example

procedure doIt(**integer** A , **float** B , **boolean** C)



processing
body of doIt



processing
calls on doIt

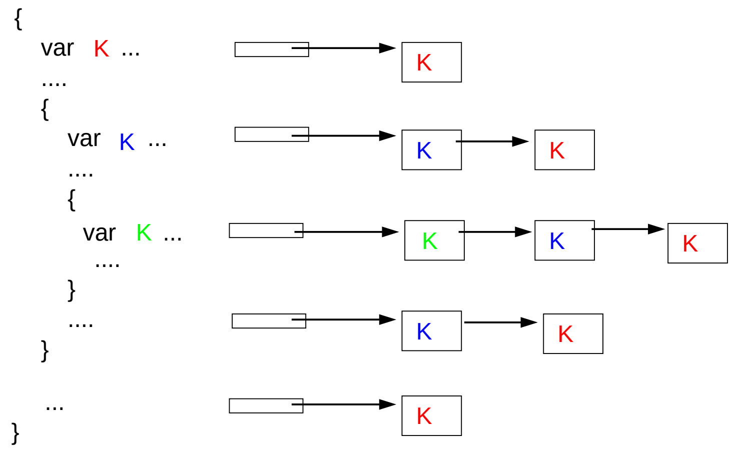
Table Search Strategies

- Most symbol table operations are table lookup. A typical symbol gets entered once when it is declared, looked up at least twice (semantic analysis and code generation) every time it is used. Deletion is usually done wholesale by discarding all the entries for a scope.
- For searching small scopes or for toy and student compilers *linear search* is a good alternative. Search top of symbol stack to bottom automatically implements Algol-60 scope rule.
- Use binary search for static tables, i.e. lists of reserved words, lists of builtin functions.
- Hash tables are the preferred search method for production compilers. O(1) search if the table is large enough.
- Indexing search is a good choice if the scanner already maps identifiers to small integers.

Table Search Strategies

- Indexing Search
  - Scanner maps all identifiers into small integers.
    Cost one lookup per occurrence. Call this integer the symbol index.
  - A vector (symbol vector) indexed by symbol index points at the most recent symbol table entry for the identifier.
  - On declaration, save current index for identifier in new symbol table entry, point symbol vector entry at new symbol table entry.
  - On scope exit, restore previous symbol vector entry for each symbol in the scope.
  - Symbol lookup through the symbol vector is always O(1).
- Builtin symbols (library routines) are often handled by wrapping a meta-scope around the entire program and building symbol table entries for the builtin symbols there. This permits automatic redeclaration of builtins.

Indexing Search Example



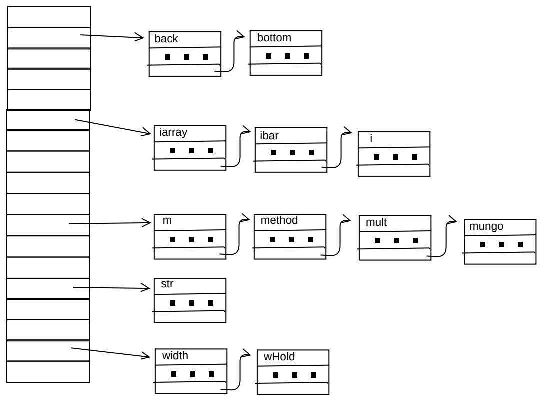
Hash Tables

- Preferred Organization for Symbol Tables
- Design issues
  - Hash function
  - Collision resolution
  - Interaction with scopes
  - Efficiency of search, insertion, deletion
- Hash function
  - Should be *simple* to compute and provide good dispersion.
  - Typical hash functions are some arithmetic combination of letters from the identifier and/or the length of the identifier.
  - Spending a *lot* of time optimizing the hash function probably isn't worthwhile, simple functions like product of first and last letters , or sum (xor) of letters is good enough.
  - Pro Compiler: have scanner compute hash function *once* for each identifier and package that value with the internal representation of identifiers.

- Hash function is a mapping from identifiers to hash table indices. Since this (must be) a many-to-one mapping, collisions *will* occur when two different identifiers map to the same index.
- The preferred method for dealing with collisions is resolution by chaining since it supports symbol table scope operations well.
- Each hash table entry is the head of a linked list of identifier nodes. The hash function is used once to locate one table entry, then the linked list is searched serially to find a given identifier.
- Advantages of using chaining to resolve collisions are
  - It allows arbitrarily large symbol tables since table storage is allocated incrementally for each identifier.
  - It facilitates removing identifiers from the table at the end of a scope.
- Indexing search (Slide 177) is a form of hashing with optimized collision resolution by chaining that uses a one-to-one hash function.

Hash Table Example

Hash Table with collision resolution by chaining.



Fisher/Lablanc Figure 8.2

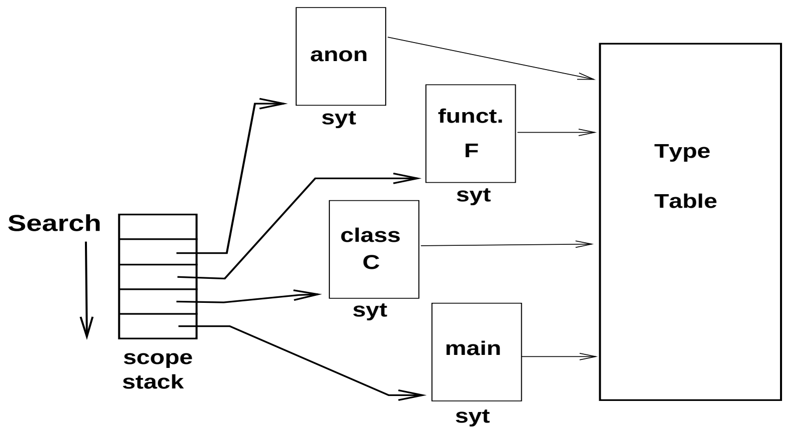
Minimizing Table Space

- The symbol table for most compilers is designed to hold a *large* number of entries. Thousands of symbol table entries would not be unusual in compiling a large program.
- It is therefore good design practice to try and keep symbol table entries as small as possible. One approach to minimizing the size of symbol table entries is to use indirect reference via pointers for any large things that would otherwise be in the symbol table.
- Many compilers use a separate table for storing the actual names of identifiers since otherwise a lot of (potentially wasted) space would be used in each symbol table entry. This optimization is particularly useful if the language allows very long identifiers, e.g. 32 characters in PL/I, 50 characters in Turing
- The semantic analysis and code generation operations performed by a compiler generally *do not need* the actual name of each identifier as long as there is some index value that maps one-to-one to identifier names.

Managing Scope Information

- There are several alternatives for designing symbol tables to implement scoping in programming languages.
- Separate symbol table per (major) scope.
  Allocate on scope entry, free on scope exit. Compiler keeps stack of pointers to currently active scopes.
  This table could be a stack or separately allocated chunks of memory.
- Global symbol table.
  Allocate at start of compilation. Add symbols as required.
  At the end of a scope (logically) delete all identifiers declared in the scope from the symbol table.
  Using hash table with chained entries, insert new symbols at head of chain, delete symbols from head of each chain until symbol from previous scope is uncovered or chain is empty.

Scope Stack Symbol Table Search



Symbol Table Design Issues

- Statements modifying visibility. Constructs like **with** in Pascal and **bind** in Turing change the visibility of symbols in the middle of a scope. These changes can be supported by copying symbol table entries or by using flags in the symbol table to render symbols invisible to the search process.
- Micro Scopes Many languages allow very small scopes (e.g. { } scopes in C)
  It is usually more efficient to store symbols for micro scopes in the symbol table of the enclosing major scope (procedure or function) and control their visibility with symbol table flags.
- Record fields. The names of record fields should only be visible inside the record. Such field names should be stored in a way that makes them invisible to normal search.

- Retained names/entries. For some types of scopes, names and/or symbol table entries must be retained in the symbol table after the scope has been processed. See Slide 174.
- Information Hiding. Constructs like **opaque** in Turing allow the programmer to hide information about the actual data representation of a type while still allowing the type to be used to create variables.
- Importation. If a module or similar construct imports some names from its environment, these names need to be made available in the modules local symbol table.
- Name Overloading Some languages (Ada, Java, C++) allow multiple definitions for the *same* name in cases where there is some language specific rule for disambiguating uses of the name.
  For example, Ada allows multiple definitions for the same function as long as the definitions have distinct formal parameter lists.

- Implicit Declarations In some languages the *occurrence* of an item constitutes a declaration for the item. Examples: statement labels, previously undeclared functions in C and Fortran, **for** loop indices in Turing.
  To handle implicit declaration the compiler must be prepared to create new symbol table entries (and possibly a new micro scope) for any undeclared identifier that it encounters.
  There may be scope related interactions for some kinds of symbols .e.g.
  for i := 1 **..** i+3 **...** **end for** in Turing.
- Supporting separate compilation. Some compilers support separate compilation at the file or module level by saving the symbol table for names exported from the file or module on disk in a compressed form. Any compilation that needs this symbol information can efficiently read it from disk. In contrast, almost all C compilers process `#include` directives by reading and processing a source file.

- forward references. Many languages allow implicit or explicit pre-declaration of names.
  Examples: label appearing in a **go to** statement, the explicit **forward** declaration in Turing, the use of function headers to pre-declare functions in C.
  This usage can be implemented by creating symbol/type table entries that are marked with an *IamForward* flag. If the corresponding real declaration is later encountered, it is validated against the forward declaration and the flag is turned off. At the end of a scope the symbol table entries for the scope need to be checked for unsatisfied forward declarations. Example:

```
L:      ...
   begin
       ...
       go to L
       ...
   L:  ...
   end
```

Pascal Tables Example - Base Definitions

```
 /* Sizes of various things */
#define maxString 500    /* size of string constant table */
#define maxSymbols 600    /*size of symbol table */
#define maxTypes 400    /* size of type table */
#define maxName 8    /* max length of names */
#define nullIndex 0    /* null table index */

/* Kinds of Symbols */
enum symKind { unknownSym, constSym, typeSym, varSym, procSym,
   funcSym, labelSym } ;

/* Kinds of Types */
enum typeKind {unknownType, intType, charType, realType, boolType,
   enumType, stringType, subrangeType, pointerType,
   forwardType, arrayType, recordType, setType,
   fileType, vrField } ;

/* Kinds of Constants*/
enum constKind { intConst , charConst , realConst , boolConst ,
   enumConst , stringConst } ;

typedef short symIndex ;    /* symbol table index */
typedef short typeIndex ;    /* type table index */
typedef short stringIndex ;   /* string table index */
typedef char [ maxName ] name ;   /* symbol name */
```


Pascal Tables Example - Descriptors

```
 typedef struct mAddrStruct{ /* machine addresses */
       unsigned base ;
       integer offset ;
       } mAddress ;

typedef struct lAddrStruct{    /* logical addresses */
       unsigned lexLevel ;
       unsigned ordNumber ;
       } lAddress ;

typedef struct constStruct {   /* description of constant's values */
       constKind cType ; /* type of constant */
       union {
          integer iValue ; /* integer */
          char chValue ; /* char */
          float rValue ; /* real */
          unsigned char bValue ; /* Boolean */
          integer eValue ; /* enum */
          stringIndex sValue ; /* string */
          }
       } constDesc ;

typedef struct pfdStruct {    /* procedure/function descriptor */
       mAddress pfAddress ; /* code address */
       unsigned pfCount ; /* parameter count */
       symIndex pfParml ; /* syt 1st parameter */
       } pfDesc ;
```


Pascal Example - Type Table Entry

```
 /* definition of type table entry */
typedef struct typeStruct {
       /* data common to all types */
       unsigned tySize ; /* size of this type */
       /* fields that depend on the kind of type */
       typeKind tyKind ; /* type kind for this type */

       union { /* information depending on tyKind */
          struct { /* enumerated type */
             short eCount ; /* number of values */
             symIndex eValue1 ; /* ptr to 1st value constant */
          } enumInfo ;

          struct {  /* sub range type */
             typeIndex srBase ; /* ptr to base type */
             constDesc srFirst, srLast ; /* bounds */
          } subRangeInfo ;
          unsigned strMaxLength ; /* string */

          struct { /* array */
             boolean aPacked ; /* packed array */
             unsigned aCount ; /* element count */
             typeIndex aIndex ; /* ptr to index type */
             typeIndex aComponent ; /* ptr to component type */
          } arrayInfo ;

          typeIndex pObject ; /* pointer type */
          symIndex fwdName ; /* forward type */
```



```
     /* typeEntry continued */
       struct { /* record type */
          boolean rPacked ; /* packed record */
          unsigned rCount ; /* number of record fields */
          symIndex rField1 ; /* ptr 1st field */
          boolean hasVariant ; /* contains variant record */
          symIndex vrTag ; /* var record tag identifier */
          typeIndex vrType ; /* variant record tag type */
          typeIndex vrVariant1 ; /* ptr to first variant */
       } recdInfo ;

       struct { /* pseudo type entry for variant record fields */
          constDesc vrLabel ; /* value of label */
          typeIndex vrNextVariant ; /* ptr to next variant */
          symIndex vrFields ; /* fields for this variant */
       } vrRecdInfo ;

       struct {  /* set type */
          boolean stPacked ; /* packed set */
          unsigned stCount ; /* number of elements */
          typeIndex stBase ; /* ptr to base type of set */
          constDesc stFirst, stLast ; /* set bounds */
       } setInfo ;
       struct { /* file type */
          boolean fPacked ; /* packed file */
          typeIndex fComponent ; /* file object type */
       } ;
    } tyInfo ; /* end tyKind union */
} typeEntry ;
```

Pascal Example - Symbol Table Entry, Tables

```
  /* definition of symbol table entry */
typedef struct symStruct {
       /* data common to all symbols */
       name syName ;        /* symbol's name */
       symKind syKind ;     /* kind of symbol */
       typeIndex syType ;     /* ptr to type table */
       symIndex syNext ;     /* link to related symbol */
       lAddress syLAddr ;     /* logical address */
       /* fields dependent on kind of symbol */
       union {
          constDesc :  syCValue ;     /* constant's value */
          struct {                /* variable */
             mAddress vAddress ; /* run time address */
             unsigned vSize ; /* size */
          } syVarInfo ;
          pfDesc pfInfo ;     /* procedure/function */
          mAddress lblAddr ;     /* statement label */
       } syInfo ;
    } symEntry ;

 /* Table Definitions */
char * stringTable[ maxString ] ;     /* String Table */
symEntry syt[ maxSymbols ] ;        /* Symbol Table */
typeEntry tyt[ maxTypes ] ;        /* Type Table */
```


Pascal Symbol and Type Table Example[a]

type

angle = 0 .. 360 ;

shape = (triangle , circle , rectangle) ;

R = **record**

    x , y : **real** ;

    area : **real** ;

    case s : shape **of**

    triangle: (side : **real** ;

        incline, angle1, angle2 : angle);

    rectangle: (side1, side2 : **real** ;

        skew, angle3 : angle)

    circle : (diameter : **real**

end ;

[a]Pascal User Manual and Report, pg. 141