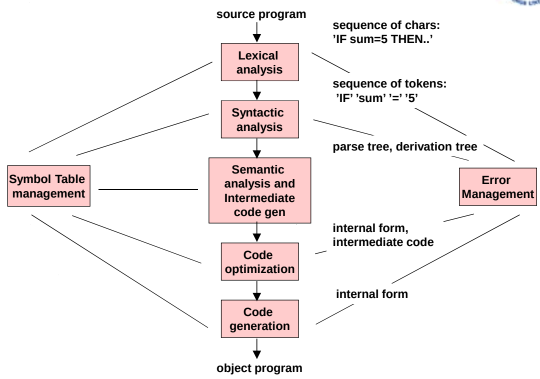
TDDD55 Compilers and interpreters

TDDB44 Compiler Construction

Symbol Tables

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

Symbol Tables in the Compiler



Symbol Table Functionality

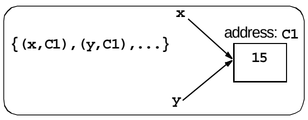
- Function: Gather information about names which are in a program.
- A symbol table is a data structure, where information about program objects is gathered.
  - Is used in both the analysis and synthesis phases.
  - The symbol table is built up during the lexical and syntactic analysis.
- Provides help for other phases during compilation:
  - Semantic analysis: type conflict?
  - Code generation: how much and what type of *run-time* space is to be allocated?
  - Error handling: Has the error message **"Variable A undefined"** already been issued?
- The symbol table phase or symbol table management refer to the symbol table's storage structure, its construction in the analysis phase and its use during the whole compilation.

Requirements and Concepts

- **Requirements for symbol table management**
  - quick insertion of an identifier
  - quick search for an identifier
  - efficient insertion of information (attributes) about an id
  - quick access to information about a certain id
  - Space- and time- efficiency
- **Important concepts**
  - Identifiers, names
  - L-values and r-values
  - Environments and bindings
  - Operators and various notations
  - Lexical- and dynamic- scope
  - Block structures

Identifiers and Names

- **Identifiers — Names**
  - An ***identifier*** is a string, e.g. **ABC**.
  - A ***name*** denotes a space in memory, i.e., it has a value and various attributes, e.g. type, scope.
- A name can be denoted by several identifiers, so-called ***aliasing***.



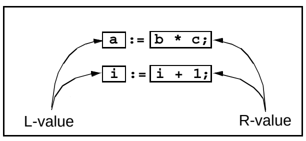
- **Example:**

```
procedure A;
  var x : ...;

  procedure B;
    var x : ...;
```

same identifier x but different names

L-value and R-value

- There is a difference between what is meant by the right and the left side of an assignment.
- Example:



- Certain expressions have either l- or r-value, while some have both l-value and r-value.

Expression	has l-value	has r-value
i+1	no	yes
b->	yes	yes
a	yes	yes
a[i]	yes	yes
2	no	yes

Binding: <*names, attributes*>

- Names
  - Come from the lexical analysis and some additional analysis.
- attributes
  - Come from the syntactic analysis, semantic analysis and code generation phase.
- *Binding* is associating an attribute with a name, e.g.

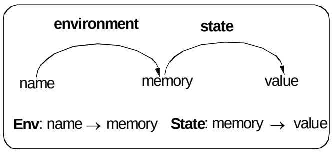
```
procedure foo;
 var k: char;         { Bind k to char }

   procedure fie;
   var k: integer;    { Bind k to integer }
```


Static and Dynamic Language Concepts

Static Concepts	Dynamic Counterparts
Definition of a subprogram	Call by a subprogram
Declaration of a name	Binding of a name
Scope of a declaration	Lifetime of binding

Environments and Bindings

- Different environments are created during execution, e.g. when calling a subprogram
- An **environment** consists of a number of **name bindings**
- Distinguish between environment and state, e.g. the assignment
  A := B;
  changes the current **state**, but not the environment.



- **Example**
  - **Env = {(x,C1),(y,C2),(z,C3),...}**
  - **State = {(C1,3),(C2,5),(C3,9),...}**
- In the environment **Env**, binds **x** to memory cell **C1**,... and memory cell **C1** has the value **3**, ...
- A *name* is bound to a memory cell, *storage location*, which can contain a value.
- A *name* can have several different *bindings* in different environments, e.g. if a procedure calls itself recursively.

Scope
1. Lexical Scope

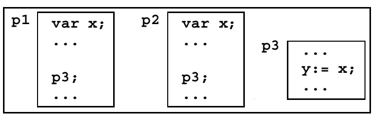
- How do we find the object which is referenced by non-local names?
  - Two different methods are used: *Lexical* and *dynamic* scope

```
program foo;
var x;
          static
procedure fie(...);
var y
begin
  y := x;
end;
 ...
end.
```

- 1. Lexical- or static- *scope*
  - The object is determined by investigating the program text, statically, at compile-time
  - The object with the same name in the nearest enclosing scope according to the text of the program
  - Is used in the languages Pascal, Algol, C, C++, Java, Modelica, etc.

2. Dynamic Scope

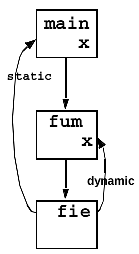
- The object is determined during run-time by investigating the current call chain, to find the most recent in the chain.
- Is used in the languages LISP, APL, Mathematica (has both). Example: Dynamic-scope



- Which **x** is referenced in the assignment statement **p3**? It depends on whether **p3** is called from **p1** or **p2**.

Lexical or Dynamic Scope

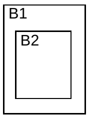
- Which **x** is referenced in procedure **fie** in the program below if
  - lexical/static scoping applies?
  - dynamic scoping applies?




```
program foo;
var x;
         static
procedure fie(...);
var y
begin
   y := x; (* which x? *)
end;
        dynamic
procedure fum(...);
var x;
begin
   x := 5;
   fie(x);
end;

begin
   x:= 10;
   fum(...);
end.
```

Block Structures

- Algol, Pascal, Simula, Ada are typical block-structured languages.
- Blocks can be nested but may not overlap
- Static *scoping* applies for these languages:
  - A name is visible (available) in the block the name is declared in.
  - If block B2 is nested in B1, then a name available in B1 is also available in B2 if the name has not been re-defined in B2.



Static and Dynamic Characteristics in Language Constructs

- **Static characteristics**
  Characteristics which are determined during compilation. Examples:
  - A Pascal-variable type
  - Name of a Pascal procedure
  - Scope of variables in Pascal
  - Dimension of a Pascal-array
  - The value of a Pascal constant
  - Memory assignment for an integer variable in Pascal
- **Dynamic characteristics**
  Characteristics that can not be determined during compilation, but can only be determined during *run-time.*
- *Examples*
  - The value of a Pascal variable
  - Memory assignment for dynamic variables in Pascal (accessible via pointer variables)

Advantages and Disadvantages

- **Static constructs**
  - \- Reduced freedom for the programmer
  - \+ Allows type checking during compilation
  - \+ Compilation is easier
  - \+ More efficient execution
- **Dynamic constructs**
  - \- Less efficient execution because of dynamic type checking
  - \+ Allows more flexible language constructions (e.g. dynamic arrays)
- More about this will be included in the lecture on *memory management*.

Symbol Table Design (decisions that must be made)

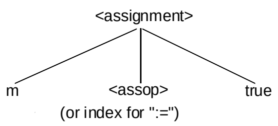
- Structuring of various types of information (attributes) for each name:
  - string space for names
  - information for procedures, variables, arrays, ...
  - *access* functions (operations) on the symbol table
  - *scope,* for block-structured languages.
- Choosing data structures for the symbol table which enable efficient storage and retrieval of information.
  Three different data structures will be examined:
  - **Linear lists**
  - **Trees**
  - **Hash tables**
- Design choices:
  - One or more tables
  - Direct information or pointers (or indexes)

Structuring Problems for Symbol Data

- When a name is declared, the symbol table is filled with various bits of information about the name:

0	...	...	...	...
...	...	...	...	...
...	...	...	...	...
m	...	...	...	...
...	...	...	...	...
n	...	...	...	...

- Normally the symbol table index is used instead of the actual name. For example, the parse tree for the statement



- This is both time- and space-efficient.
- How can the string which represents the name be stored?

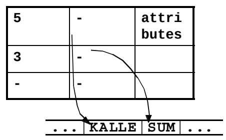
  Next come two different ways.

String Space for Identifiers

- **Method 1**: Fixed space of max expected characters
  FORTRAN4: 6 characters,
  Hedrick Pascal: 10 characters

KALLE	attributes
SUM	attributes
...	

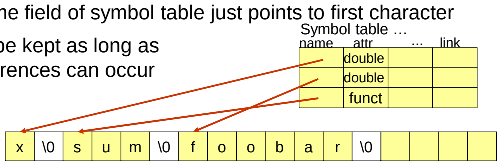
- **Method 2**: **<length, pointer>**
  (e.g. Sun Pascal: 1024 characters



- **Method 3:** without specifying length: **...$KALLE$SUM$...** where $ denotes end of string.
- The name and information must remain in the symbol table as long as a reference can occur.
- For block-structured languages the space can be re-used.

String Space for Identifiers Method 3, cont.

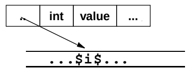
- Identifiers can vary in length
- Must be stored in token table
- Name field of symbol table just points to first character
- To be kept as long as references can occur



- Usually, full names kept only during compilation
  - Exception:
    Added to the program's constant pool in the .data segment
    if symbolic debugging or reflection should be enabled
    (e.g., gcc –g file1.c to prepare for symbolic debugging)

Information in the Symbol Table

- name
- attribute
  - type (integer, boolean, array, procedure, ...)
  - length, precision, packing density
  - address (block, offset)
  - declared or not, used or not

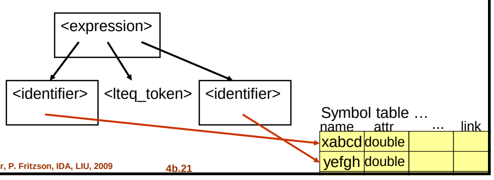


- You can directly allocate space in the symbol table for attributes whose size is known, e.g. type and value of a simple variable

Compiler representation of names

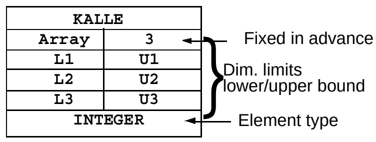
- A unique and compact internal representation for a ***name*** is the ***index*** (address in compiler address space) of its symbol table entry.
- Used instead of full name (string) in the internal representation of a program
- ☺ Time and space efficient

Example: Parse-tree for expression xabcd <= yefgh;



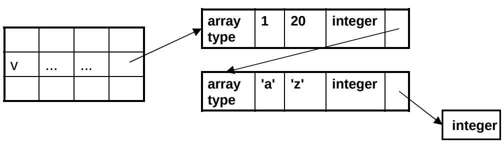
Information in the Symbol Table for Arrays Fixed Allocation

- **Fixed allocation (BASIC, FORTRAN4)**
  - The number of dimensions is known at compilation.
  - FORTRAN4: max 3 dimensions, integer index.



Information in the Symbol Table for Arrays Flexible Allocation

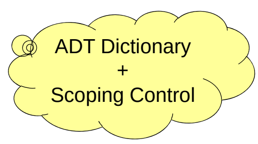
- **Flexible allocation (Pascal, Simula, ADA, Java)**
  - Arbitrary number of dimensions, elements of arbitrary type.
  - Pascal: var v: array[1..20,'a'..'z'] of integer



- You can access an element **v[i,j]** in the above array by calculating its address: **adr = BAS + k*((i-1)*r)+j-1)**
  - where **r**= number of elements/rows,
  - and **k**= number of memory cells/elements (bytes, words)

Symbol Table Data and Operations

- **Set of symbol table items**
  - searchable by name + scope
- **Data** stored for each entry:
  - name
  - attributes
    - type
      (int, bool, array, ptr, function)
    - address
      (block, offset)
    - declared or not,
      used or not
    - ...
- **Operations**
  - lookup (name)
  - insert (name)
  - put (name, attribute, value)
  - get (name, attribute)
  - enterscope ()
  - exitscope()



Data Structures for Symbol Tables

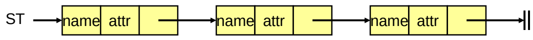
For flat symbol tables:
(one block of scope)

- Linear lists
- Hash tables
- ...
(see data structures for ADT Dictionary)

For nested scopes:

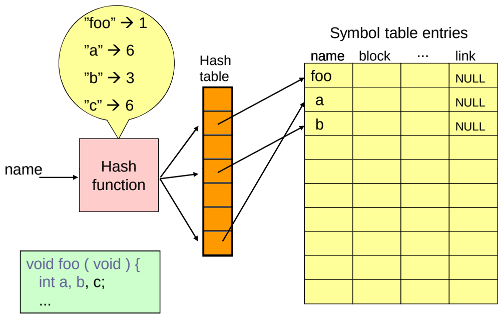
- Trees of flat symbol tables
- Linear lists with scope control
  - Only for 1-pass-compilers
- Hash tables with scope control (see following slides)
  - Only for 1-pass-compilers

Linear lists

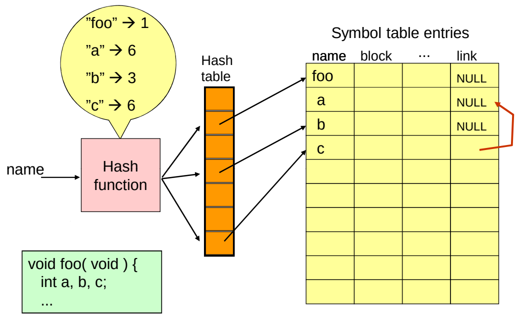


- Unsorted linear lists
  ☺ Easy to implement
  ☺ Space efficient
  ☺ Insertion itself is fast
  but needs lookup to check if the name was already in
  ☹ Lookup is slow
  Inserting *n* identifiers and doing *m* lookups requires $O(n(n+m))$ string comparisons

Hash Table with Chaining (1)



Hash Table with Chaining (2)



☺ Much faster lookup on average

☹ Degenerates towards linear list for bad hash functions

Hash Table with Chaining (3)

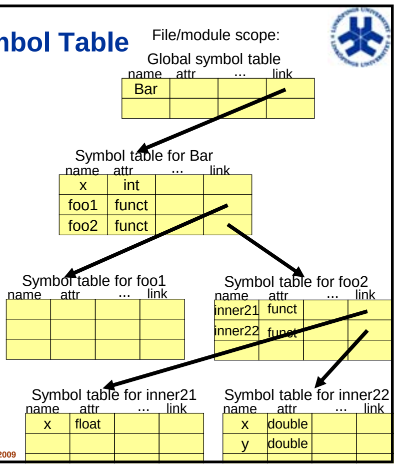
- Search
  - Hash the name in a hash function, *h(symbol)* $\in$ *[0, k-1]*
  - where *k* = table size
  - If the entry is occupied, follow the link field.
- Insertion
  - Search + simple insertion at the end of the symbol table (use the *sympos* pointer).
- Efficiency
  - Search proportional to *n/k* and the number of comparisons is *(m + n) n / k* for *n* insertions and *m* searches.
  - *k* can be chosen arbitrarily large.
- Positive
  - Very quick search
- Negative
  - Relatively complicated
  - Extra space required, *k* words for the hash table.
  - More difficult to introduce scoping.

Hierarchical Symbol Tables

For nested scope blocks

Tree-based Symbol Table

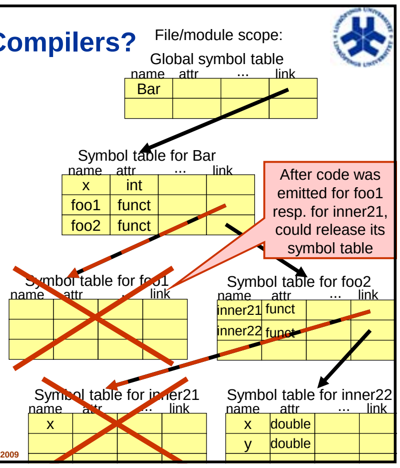
```
class Bar {
   int x;
   void foo1( … ) { … }
   void foo2( … ) {
      int inner21( … ) {
         float x;
         …
      }
      int inner22( … ) {
         double x, y;
         …
         foo1( x );
      }
      …
   }
   …
}
…
```



- enterscope(), exitscope()
- insert(), lookup()

For One-Pass Compilers?

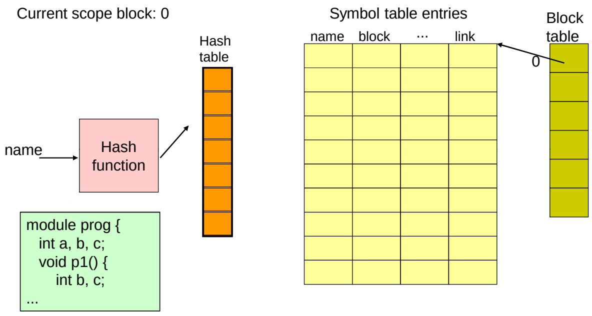
```
class Bar {
   int x;
   void foo1( … ) { … }
   void foo2( … ) {
      int inner21( … ) {
         float x;
         …
      }
      int inner22( … ) {
         double x, y;
         …
         foo1( x );
      }
      …
   }
   …
}
…
```



- enterscope(), exitscope()
- insert(), lookup()

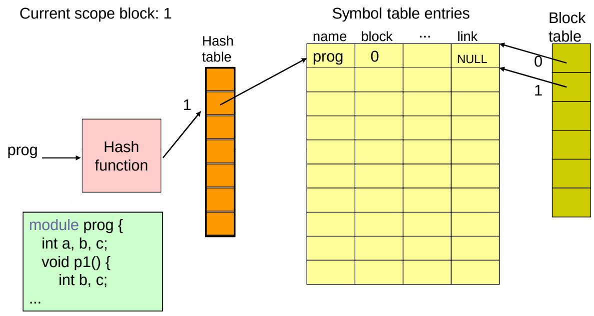
Hash tables with chaining + scoping

(For One-Pass Compilers Only)



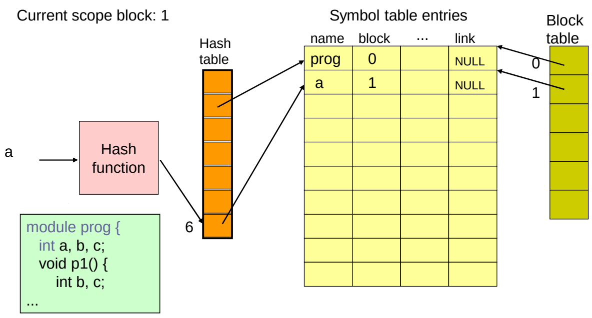
insert p1 and enter a new scope block (2)

Hash tables with chaining + scoping

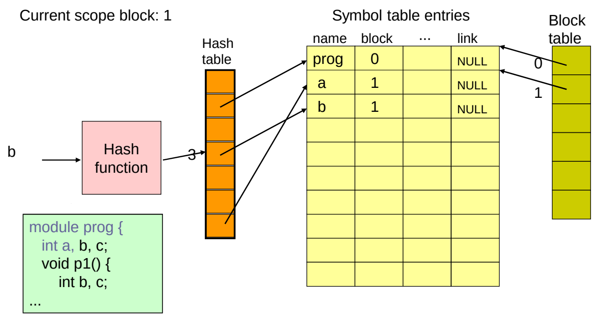


insert prog and enter a new scope block (1)

Hash tables with chaining + scoping

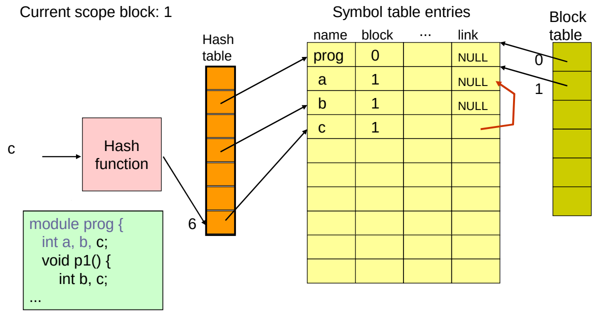


Hash tables with chaining + scoping



Hash tables with chaining + scoping

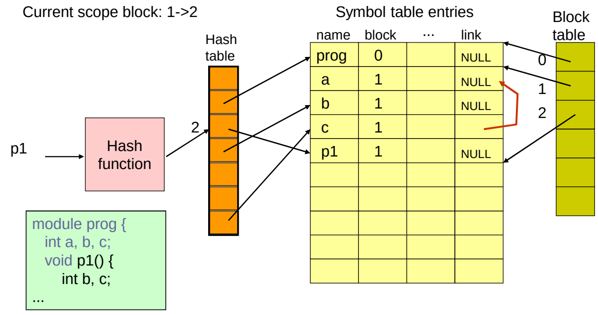
Current scope block: 1



a and c hash to the same hash value (6) – use chaining

Hash tables with chaining + scoping

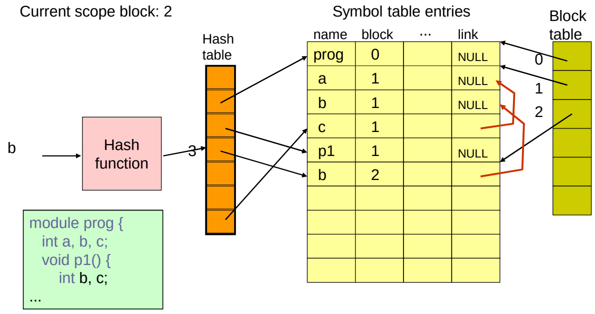
Current scope block: 1->2



insert p1 and enter a new scope block (2)

Hash tables with chaining + scoping

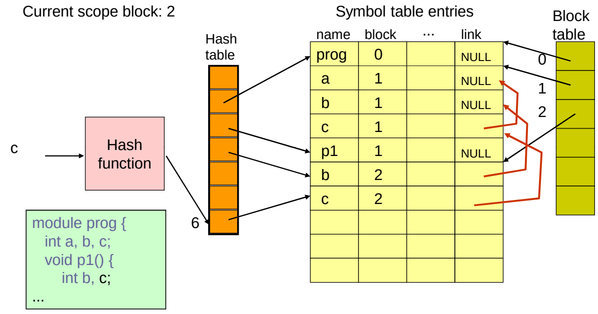
Current scope block: 2



make hash table point to (statically) closest b – will later find this one first in chain

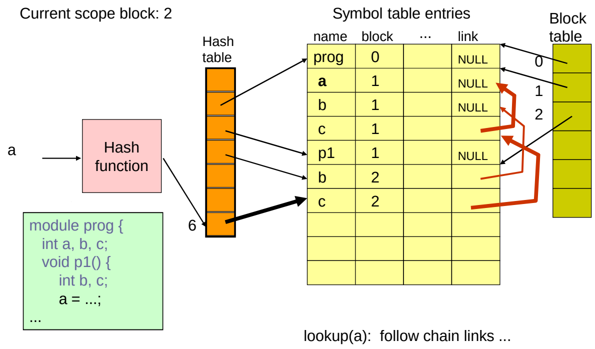
Hash tables with chaining + scoping

Current scope block: 2



Hash tables with chaining + scoping

Current scope block: 2



lookup(a): follow chain links ...

Operations on Hash-Table with Chaining and Scope (Block) Information

- Declaring **x**
  - **Search along the chain for x**'s hash value.
  - When a name (any name) in another block is found, **x** is not **double-defined**.
  - Insert **x** at the beginning of the hash chain.
- Referencing **x**
  - **Search along the chain for x**'s hash value.
  - The first **x** to be found is the right one.
  - If **x** is not found, **x** is **un**defined.
- A new block is started
  - Insert block pointer in **BLOCKTAB**.
- End of the block
  - Move the block down in **BLOCKTAB**.
  - Move the block down in **SYMTAB**.
  - Move the hash pointer to point at the previous block.