Symbol tables

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.
- 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?
- 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 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

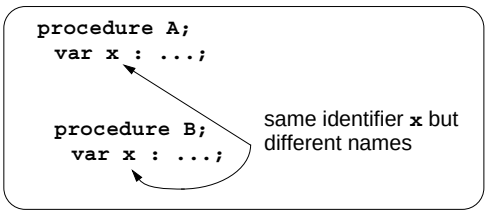
Important concepts

- Identifiers, names
- L-values, *L-values* and r-values, *r-values*
- Environments and bindings
- Operators and various notations
- Lexical- and dynamic- *scope*
- Block structures

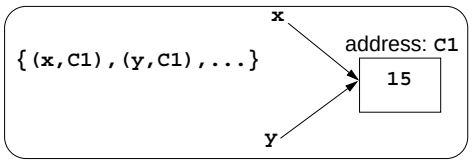
Identifiers — Names

- 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.

Example:



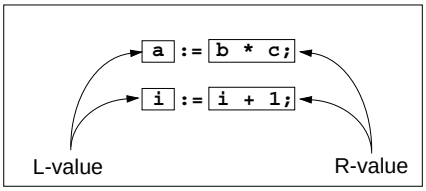
- A name can be denoted by several identifiers, so-called *aliasing*.



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 expression have l- or r-value, while some have both l-value and r-value.

Expression	has l-value	has r-value
`i+1`	n	j
`b^`	j	j
`a`	j	j
`a[i]`	j	j
`2`	n	j

Operators and different notations

- Unary operators have one operand, e.g. `-x`
- Binary operators have two operands, `x + y`
- Ternary operators have three operands,
  `if villkor1 then sats2 else sats3`

Operators are denoted with the help of different notations:

- Prefix notation, `-x, sort(a,b,c)`
- Infix notation, `x + y`
- Postfix notation, `x!`

Binding: *<names, attributes>*

- names

  Come from the lexical analysis.

- attributes

  Come from the syntactic analysis, semantic analysis and code generation phase.

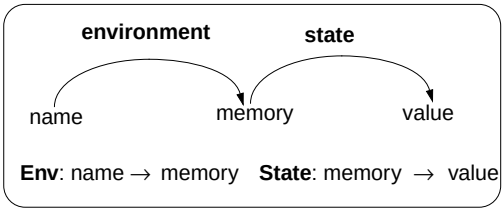
Binding is associating an attribute with a name, e.g.:

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

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

Static and dynamic concepts:

Static concepts	Dynamic counterpart
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.

Lexical- and dynamic- *scope*

How do we find the object which is referenced by non-local names?

Two different methods are used:

1. Lexical- or static- *scope*

   The object is determined by investigating the program text, statically.

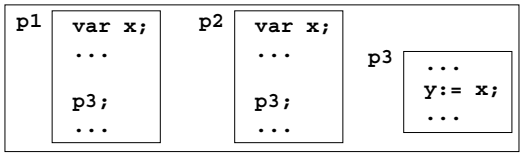
   Is used in the languages Pascal, Algol, C.

2. Dynamic *scope*

   The object is determined during run-time by investigating the current call chain.

   Is used in the languages LISP, APL.

   Example: Dynamic-*scope*



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

Which **x** is referenced in procedure **fie** in the program below if

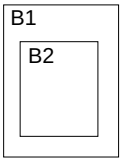
a) static scoping applies?

b) dynamic scoping applies?

```
program foo;
var x ;

   procedure fie(...);
   var y;
   begin
      y:= x;     (* which x ? *)
   end;

   procedure fum(...);
   var x ;
   begin
     x:= 5;
     fie(...);
   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) A name is visible (available) in the block the name is declared in.
  b) 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 constructions

Static characteristics

Characteristics which are determined during compilation.

Example

- 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.*

Example

- The value of a Pascal variable
- Memory assignment for dynamic variables in Pascal (accessible via pointer variables)

Advantages and disadvantages

Static

- Reduced freedom for the programmer

+ Allows type checking during compilation

+ Compilation is easier

+ More efficient execution

Dynamic

- 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 (decision that must be made)

1. Structuring of various types of information (attributes) for each name:
   a) string space for names
   b) information for procedures, variables, arrays, ...
   c) *access* functions (operations) on the symbol table
   d) *scope,* for block-structured languages.

2. Choosing data structures for the symbol table which enable efficient storage and retrieval of information. Three different data structures will be examined:
   a) Linear lists
   b) Trees
   c) Hash tables

Design choices:

- One or more tables
- Direct information or pointers (or indexes)

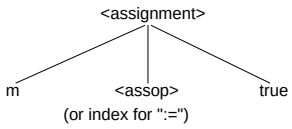
Structuring problems

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

0	...	...	...	...	...
	...	...	...	...	...
	...	...	...	...	...
m	done	id	Boolean	...	...
	...	...	...	...	...
n	...	...	...	...	...

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

```
done := true
```



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

  Here come two different ways.

String space for names

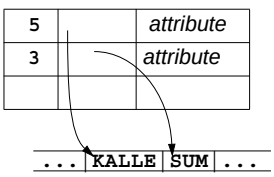
Method 1: Fixed space of max expected characters

FORTRAN: 6 characters, Hedrick Pascal: 10 characters

KALLE	*attributes*
SUM	*attributes*
...	*...*

Method 2: **<length, pointer>** (Sun Pascal: 1024 characters)

length pointer ....



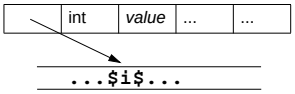
Alt. without specifying length: **...$KALLE$SUM$...** where **$** denotes end of string.

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.

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:

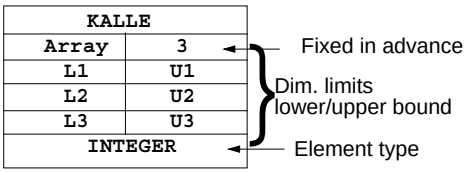


How do you store information about an array in the symbol table?

Information in the symbol table for arrays

• Fixed allocation (BASIC, FORTRAN)

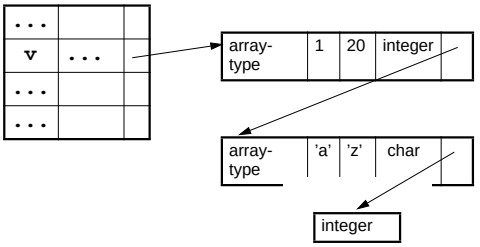
- The number of dimensions is known at compilation.
- FORTRAN4*:* max 3 dimensions, integer index.



• Flexible allocation (Pascal, Simula, ADA)

Pascal: Arbitrary number of dimensions, elements of arbitrary type.

```
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)

Access functions (operations) on symboltab

- `is(x)`
  Determine whether `x` is in the symbol table.
- `enter(x)`
  Insert `x` in the symbol table.

```
lookup(x) = is(x) + enter(x)
```

- `put(x, attr, value)`
  Insert value as the value of the attribute `attr` for the name `x`
- `get(x, attr)`
  Return the value of the attribute `attr` for `x`
- `delete(x)`
  Remove `x` and all information about `x` from the table.

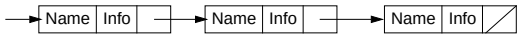
Data structures for symbol tables

- Linear lists
- Trees
- Hash tables

Keep in mind:

- Search for a name
- Insert a name
- *Scoping* (removing info about a scope)

Linear lists



Search
Search linearly from beginning to end. Stop if found.

Adding
Search (does it exist?). Add at beginning if not found.

Effectivity
To insert *n* names and search for *m* names the cost will be *cn(n+m)* comparisons.
Inefficient.

Positive

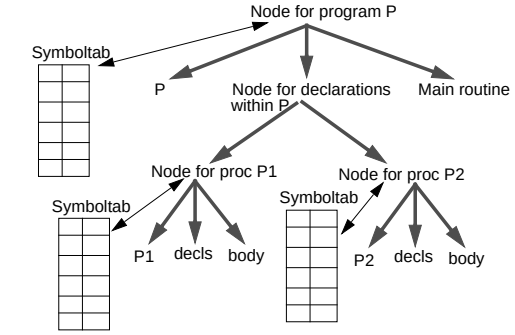
- Easy to implement
- Uses little space
- Easy to represent scoping.

Negative

- Slow for large *n* and *m*.

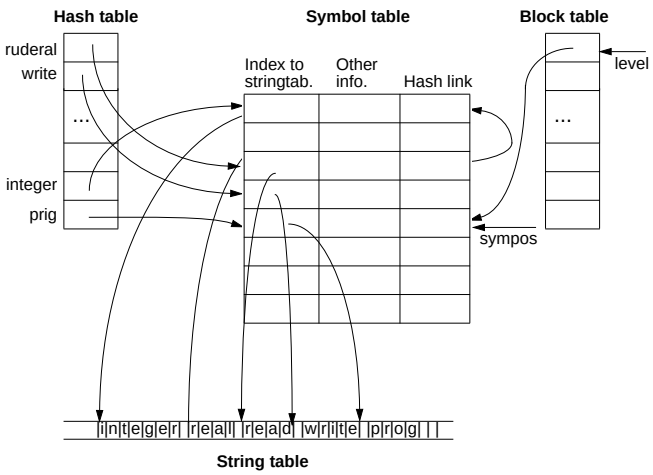
Trees

You can have the symbol table in the form of trees as below:



- Each subprogram has a symbol table associated to its node in the abstract syntax tree.
- The main program has a similar table for globally declared objects.
- Quicker that linear lists.
- Easy to represent scoping.

Hash tables (with chaining)



Hash tables (with chaining)

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.

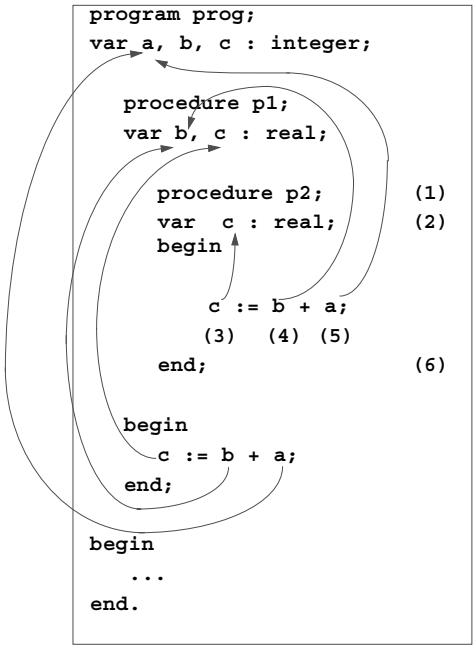
Positive

- Very quick search

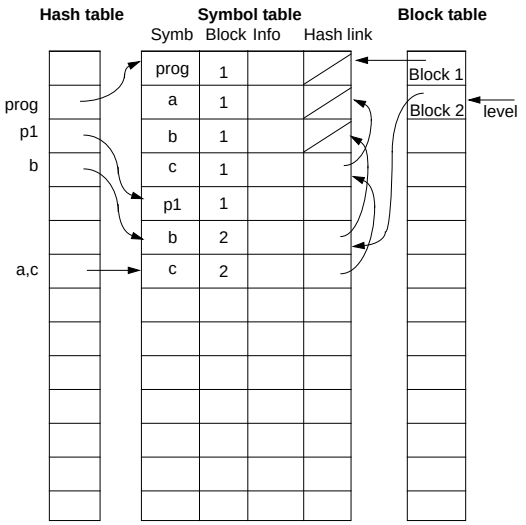
Negative

- Relatively complicated
- Extra space required, *k w*ords for the hash table.
- More difficult to introduce scoping.

Example of symbol table with hashing



1. 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.
2. 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 **undefined**.
3. A new block is started
   - Insert block pointer in **BLOCKTAB**.
4. 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.



NB. a and c have the same hash value