Semantic Analysis

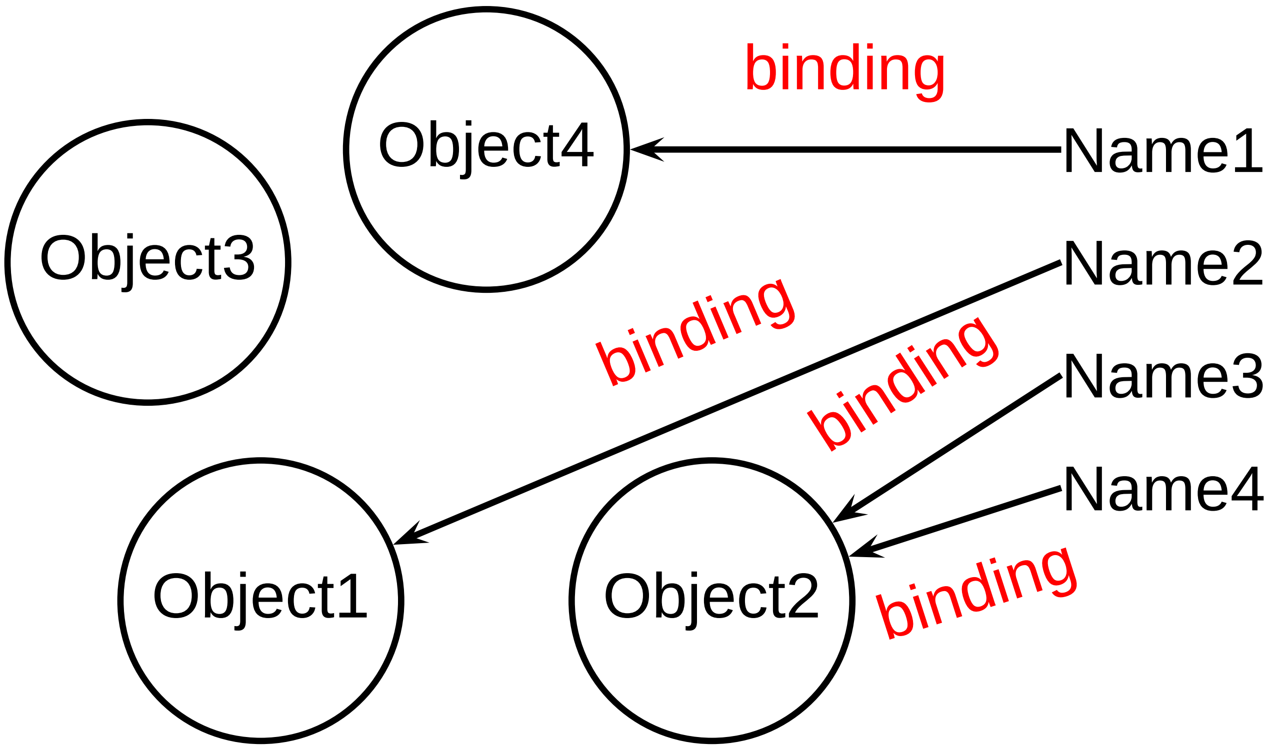
What’s In a Name?

Name: way to refer to something else

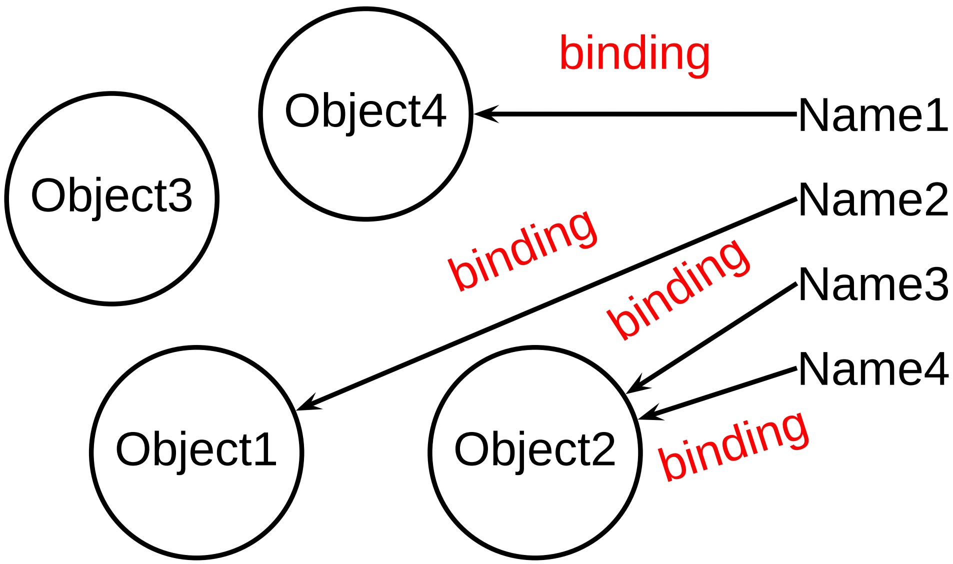
variables, functions, namespaces, objects, types

```
if ( a < 3 ) {
  int bar = baz(a + 2);
  int a = 10;
}
```

Names, Objects, and Bindings



Names, Objects, and Bindings



When are objects created and destroyed?

When are names created and destroyed?

When are bindings created and destroyed?

Object Lifetimes

When are objects created and destroyed?

Object Lifetimes

The objects considered here are regions in memory.

Three principal storage allocation mechanisms:

1. Static

   Objects created when program is compiled, persists throughout run

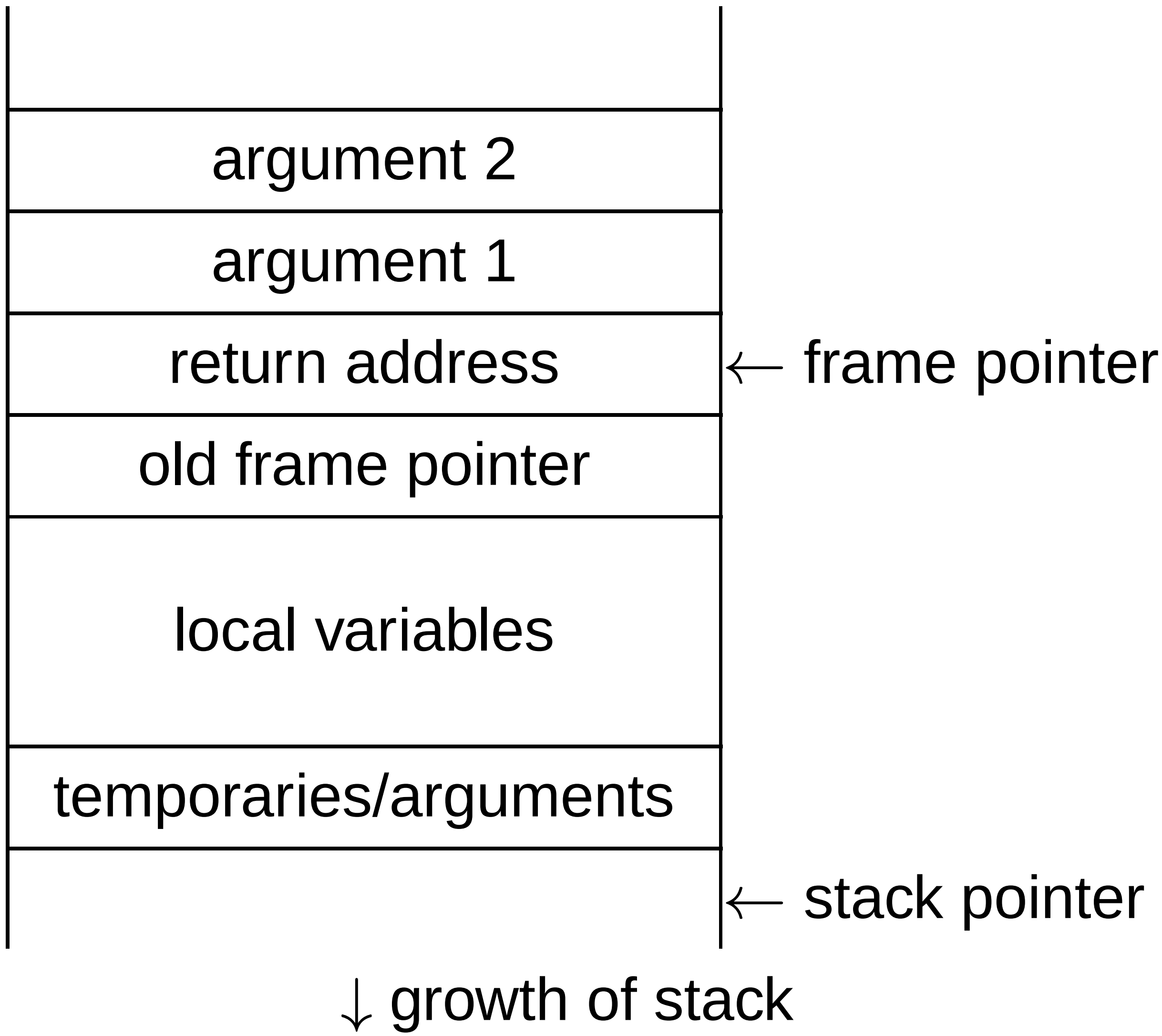
2. Stack

   Objects created/destroyed in last-in, first-out order. Usually associated with function calls.

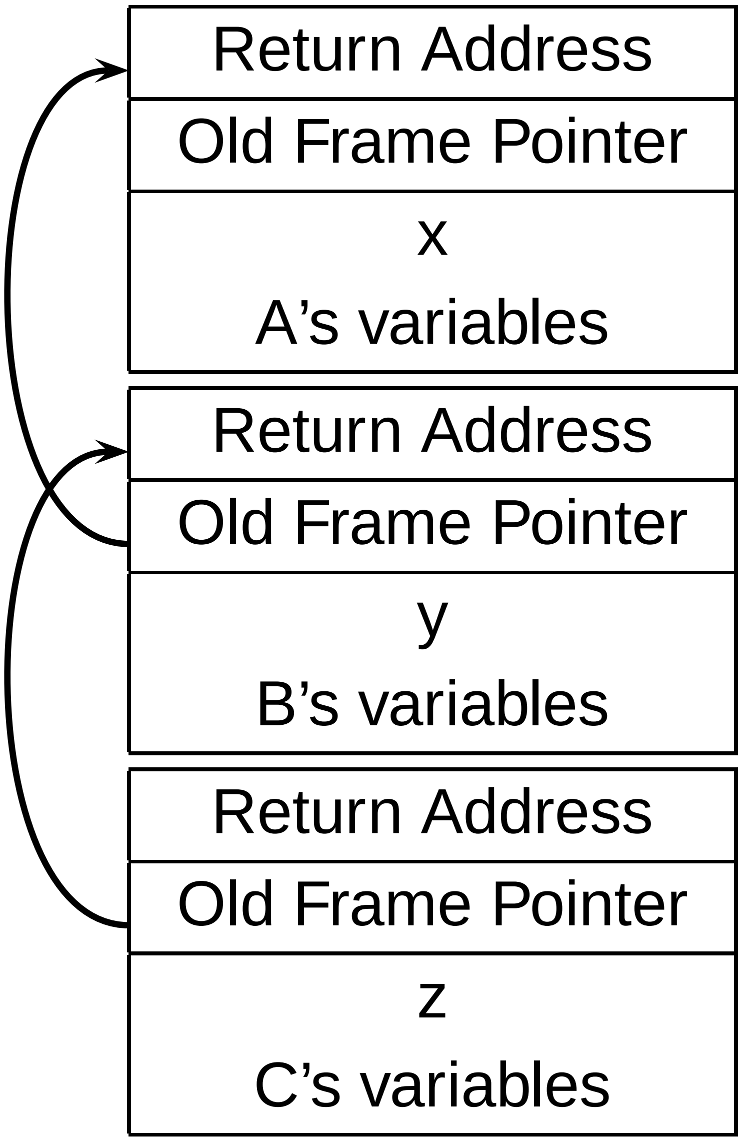
3. Heap

   Objects created/deleted in any order, possibly with automatic garbage collection.

Activation Records



Activation Records




```
int A() {
  int x;
  B();
}

int B() {
  int y;
  C();
}

int C() {
  int z;
}
```

Scope

When are names created, visible, and destroyed?

Scope

The scope of a name is the textual region in the program in which the binding is active.

Static scoping: active names only a function of program text.

Dynamic scoping: active names a function of run-time behavior.

Scope: Why Bother?

Scope is not necessary. Languages such as assembly have exactly one scope: the whole program.

Reason: Information hiding and modularity.

Goal of any language is to make the programmer's job simpler.

One way: keep things isolated.

Make each thing only affect a limited area.

Make it hard to break something far away.

Basic Static Scope

Usually, a name begins life where it is declared and ends at the end of its block.

```
void foo()
{

  int k;
}
```

Hiding a Definition

Nested scopes can hide earlier definitions, giving a hole.

```
void foo()
{
  int x;

   while ( a < 10 ) {
     int x;

  }


}
```

Static Scoping in Java

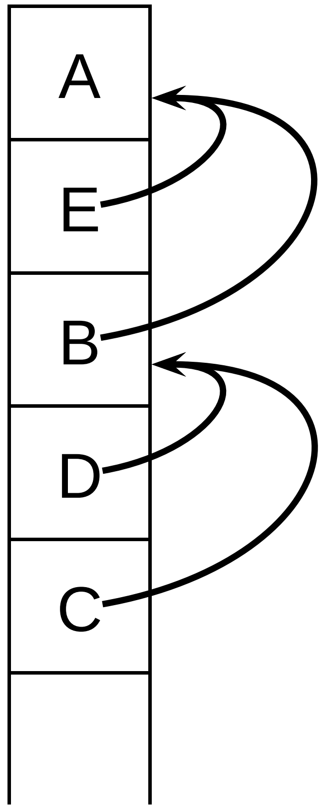
```
public void example() {
  // x, y, z not visible

  int x;
  // x visible

  for ( int y = 1 ; y < 10 ; y++ ) {
    // x, y visible

    int z;
    // x, y, z visible
  }

  // x visible
}
```

Nested Subroutines in Pascal

```
procedure mergesort;
var N : integer;

  procedure split;
  var I : integer;
  begin .. end

  procedure merge;
  var J : integer;
  begin .. end

begin .. end
```

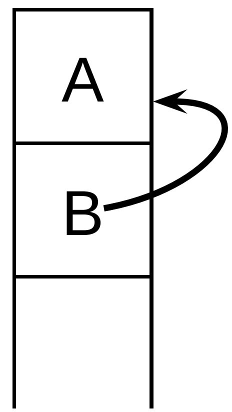
Nested Subroutines in Pascal

```
procedure A;
  procedure B;
    procedure C;
    begin .. end

    procedure D;
    begin C end
  begin D end

  procedure E;
  begin B end
begin E end
```



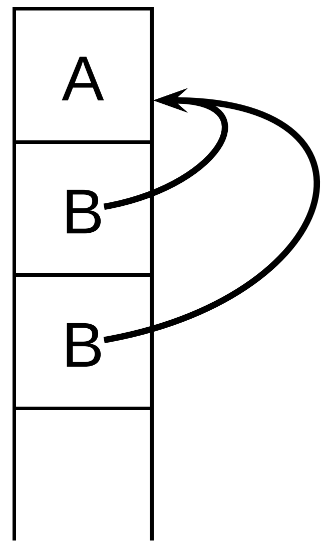
Example

```
procedure A;
  procedure B;
    procedure C;
    begin if (..) C; end
  begin
    if (..) B;
    C;
    D;
  end
  procedure D;
  begin .. end
begin B; end
```



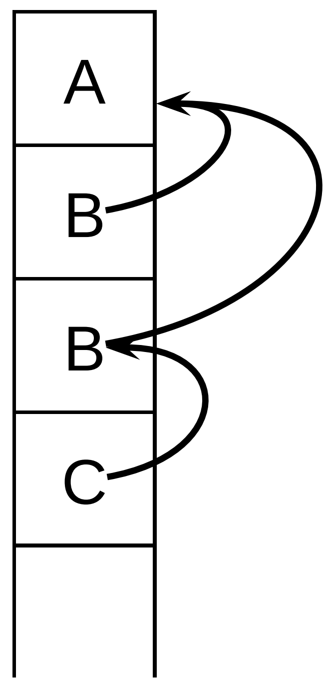
Example

```
procedure A;
  procedure B;
    procedure C;
    begin if (..) C; end
  begin
    if (..) B;
    C;
    D;
  end
  procedure D;
  begin .. end
begin B; end
```



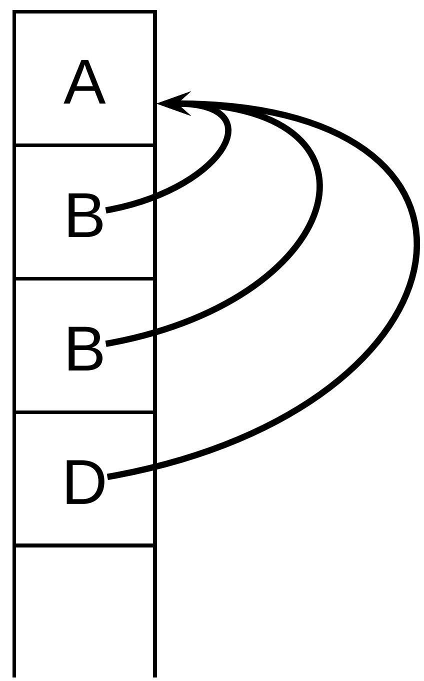
Example

```
procedure A;
  procedure B;
    procedure C;
    begin if (..) C; end
  begin
    if (..) B;
    C;
    D;
  end
  procedure D;
  begin .. end
begin B; end
```



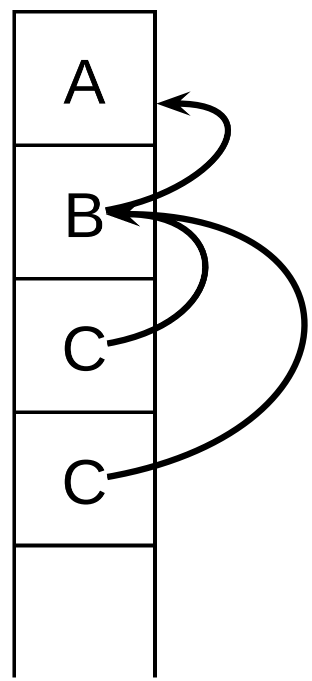
Example

```
procedure A;
  procedure B;
    procedure C;
    begin if (..) C; end
  begin
    if (..) B;
    C;
    D;
  end
  procedure D;
  begin .. end
begin B; end
```



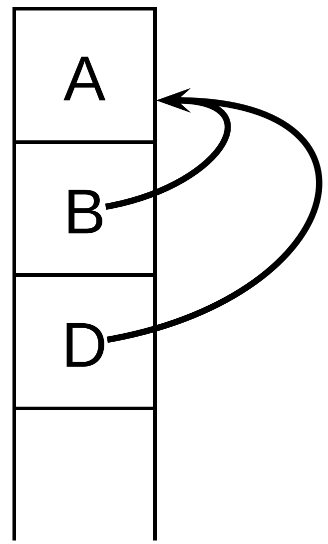
Example

```
procedure A;
  procedure B;
    procedure C;
    begin if (..) C; end
  begin
    if (..) B;
    C;
    D;
  end
  procedure D;
  begin .. end
begin B; end
```



Example

```
procedure A;
  procedure B;
    procedure C;
    begin if (..) C; end
  begin
    if (..) B;
    C;
    D;
  end
  procedure D;
  begin .. end
begin B; end
```



Dynamic Scoping in TeX

```
% \x, \y undefined
{
  % \x, \y undefined
  \def \x 1
  % \x defined, \y undefined

  \ifnum \a < 5
    \def \y 2
  \fi

  % \x defined, \y may be undefined
}
% \x, \y undefined
```

Static vs. Dynamic Scope

```
program example;
var a : integer; (* Outer a *)

  procedure seta;    begin a := 1 end

  procedure locala;
  var a : integer; (* Inner a *)
  begin seta end

begin
  a := 2;
  if (readln() = 'b') locala
  else seta;
  writeln(a)
end
```

Static vs. Dynamic Scope

Most languages now use static scoping.

Easier to understand, harder to break programs.

Advantage of dynamic scoping: ability to change environment.

A way to surreptitiously pass additional parameters.

Symbol Tables

How does a compiler implement scope rules?

Symbol Tables

Basic mechanism for relating symbols to their definitions in a compiler.

Eventually need to know many things about a symbol:

- Whether it is defined in the current scope. “Undefined symbol”
- Whether its defined type matches its use.
  `1 + "hello"`
- Where its object is stored (statically allocated, on stack).

Symbol Tables

Implemented as a collection of dictionaries in which each symbol is placed.

Two operations: insert adds a binding to a table and lookup locates the binding for a name.

Symbol tables are created and filled, but never destroyed.

Implementing Symbol Tables

Many different ways:

- linked-list
- hash table
- binary tree

Hash tables are faster, but linked lists are good enough for simple compilers.

Symbol Table Lookup

Basic operation is to find the entry for a given symbol.

In many implementation, each symbol table is a scope.

Each symbol table has a pointer to its parent scope.

Lookup: if symbol in current table, return it, otherwise look in parent.

Static Semantic Checking

Main application of symbol tables.

A taste of things to come:

Enter each declaration into its symbol table.

Check that each symbol used is actually defined in the symbol table.

Check its type. . . (next time)

Static Semantic Analysis

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This"                  /* valid */
#a1123                         /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

```
for i := 1 to 5 do 1 + break /* valid */
if i 3                       /* invalid */
```

Semantic analysis: Makes sure program is consistent

```
let v := 3 in v + 8 end      /* valid */
let v := "f" in v(3) + v end /* invalid */
```

Name vs. Structural Equivalence

```
let
  type a = { x: int, y: int }
  type b = { x: int, y: int }
  var i : a := a { x = 1, y = 2 }
  var j : b := b { x = 0, y = 0 }
in
  i := j
end
```

Not legal because **a** and **b** are considered distinct types.

Name vs. Structural Equivalence

```
let
  type a = { x: int, y: int }
  type b = a
  var i : a := a { x = 1, y = 2 }
  var j : b := b { x = 0, y = 0 }
in
  i := j
end
```

Legal because **`b`** is an alias for type **`a`**.

`{ x: int, y: int }` creates a new type, not the **`type`** keyword.

Things to Check

Make sure variables and functions are defined.

```
let var i := 10
in  i(10,20) /* Error: i is a variable */
end
```

Verify each expression's types are consistent.

```
let var i := 10
    var j := "Hello"
in  i + j /* Error: i is int, j is string */
end
```

Things to Check

- Used identifiers must be defined
- Function calls must refer to functions
- Identifier references must be to variables
- The types of operands for unary and binary operators must be consistent.
- The first expression in an **`if`** and **`while`** must be a Boolean.
- It must be possible to assign the type on the right side of an assignment to the lvalue on the left.
- . . .

Static Semantic Analysis

Basic paradigm: recursively check AST nodes.

`1 + break`

```
   +
  / \
1   break
```

check(+)

check(1) = int

check(break) = void

FAIL: int $\neq$ void

`1 - 5`

```
  -
 / \
1   5
```

check(**-**)

check(1) = int

check(5) = int

Types match, return int

Ask yourself: at a particular node type, what must be true?

Implementing Static Semantics

Recursive walk over the AST.

Analysis of a node returns its type or signals an error.

Implicit "environment" maintains information about what symbols are currently in scope.

Implementing Static Semantics

```
SymbolTable Table;

TypeCheck(Program(Ltype Lvar Lblock Linst))
{
  Table:=EmptyTable;
  Push(EmptyScope);
  UpdateTable(Ltype);
  CheckCircularTypes();
  UpdateTable(Lvar);
  AddHeaders(Lblock);
  TypeCheck(Lblock);
  TypeCheck(Linst);
  Pop();
}
```

Implementing Static Semantics

```
UpdateTable(TypeDecl(id,t))=
   Table.push(id,⟨TYPE,TypeId(t)⟩)

UpdateTable(VarDecl(id,t))=
   Table.push(id,⟨LOCALVAR,TypeId(t)⟩)

TypeId(t)=
   if basic_type(t) return t;
   else if defined_type(t) return Index(t);
   else /* structured type */
      for each component c of t do
         t.c.type = TypeId(t.c.type);
      return t;
```

Implementing Static Semantics

```
TypeCheck(Procedure(id,Lpar,Ltype,Lvar,Lblock,Linst))
{
  Push(EmptyScope);
  UpdateTable(Lpar);
  UpdateTable(Ltype);
  CheckCircularTypes();
  UpdateTable(Lvar);
  AddHeaders(Lblock);
  TypeCheck(Lblock);
  TypeCheck(Linst);
  Pop();
}
```

Example

```
Program
  Types t1 Array [1..10] Of Int EndTypes
  Vars x t1 EndVars
  Procedure P(Val y t1)
    Types
      t1 Struct
           a Int
           b Real
         EndStruct
    EndTypes
    Vars
      z t1
    EndVars
    x[3]:=z.a+y[4]
  EndProcedure
  P(x)
EndProgram
```

Implementing Static Semantics

Type(e$_1$[e$_2$])	=	**Type(t)** , if **Type(e$_1$)** is **array(i,j,t)** and **Type(e$_2$)=INT**.
	=	**ERROR**, otherwise.
Type(&e)	=	**pointer(Type(e))**, if **Addressable(e)**.
	=	**ERROR** , otherwise.
Type(eˆ)	=	**Type(t)** , if **Type(e)** is **pointer(t)**.
	=	**ERROR** , otherwise.
Type(id)	=	**Type(Table[id].type)**
Type(t)	=	**Type(Table[t.address].type)**, if **t** is a defined type.
Type(t)	=	**t**, if **t** is a basic or structured type.

Implementing Static Semantics

`Addressable(op e)`	=	`False`
`Addressable(`e_1 `op` e_2`)`	=	`False`
`Addressable(&e)`	=	`False`
`Addressable(e^)`	=	`True`
`Addressable(`e_1`[`e_2`])`	=	`Addressable(`e_1`)`
`Addressable(e.id)`	=	`Addressable(e)`
`Addressable(id(L_real_params))`	=	`False`
`Addressable(id ,T)`	=	`False`, if `T[id].type.class` $\in$ $\{$`procedure,function`$\}$
	=	`True`, otherwise.

Implementing Static Semantics

TypeCheck(e$_1$:=e$_2$) = **OK**, if (**Type(e$_1$)=Type(e$_2$)** or
(**Type(e$_1$)=REAL and Type(e$_2$)=INT)**
or (**Type(e$_1$)** is **pointer(t)** and **e$_2$=NULL**))
and **Addressable(e$_1$)**
= **ERROR**, otherwise.

TypeCheck(while e do l) = **OK** , if **Type(e)=BOOL** and
TypeCheck(l)=OK
= **ERROR**, otherwise.

Exercise

Define the function **TypeCheck** for the rest of instructions of CL:

$$\begin{array}{lcl}
\texttt{TypeCheck(if e then l}_1\texttt{ else l}_2\texttt{)} & = & ? \\
\texttt{TypeCheck(id(L_real_params)))} & = & ?
\end{array}$$