Theory of Programming Languages

Week 6

VARIABLES

Names, Bindings, Type Checking, Scopes

Contents

- Scoping
- Static and Dynamic Scoping
- Virtues and Problems with Static and Dynamic Scoping
- Assignment# 3
- Scoping and lifetimes
- Referencing Environments

Scope, Visibility and Scoping Rules

The **scope** of a program variable is the range of statements in which the variable is visible.

A variable is **visible** in a statement if it can be referenced in that statement.

The **scope rules** of a language determine how a particular occurrence of a name is associated with a variable.

- Scope rules determine how references to variables declared outside the currently executing subprogram or block are associated with their declarations and attributes.

Local and non-Local Variables

A variable is **local** in a program unit or block if it is declared there.

- For our discussion today we will confine out attention to only the variables declared in the main function or inside subroutines – deferring the discussion of variables declared inside classes or such units.

The **nonlocal** variables of a program unit or block are those that are visible within the program unit or block but are not declared there.

Static Scoping

Scope of a variable can be determined before execution.

- When a reference to a variable is found by a compiler for a static- scoped language, the attributes of the variable are determined by finding the statement in which it is declared.
- The search for a variable declaration is done first inside itself, then inside its **static parent** and then up to its farthest **static ancestor**.

Example Pascal Procedure - I

```
procedure big;
  var x : integer;
  procedure sub1;
    begin  { sub1 }
    ...x...
    end;  { sub1 }
  procedure sub2;
    var x : integer;
    begin  { sub2 }
    ...
    end;  { sub2 }
  begin  { big }
  ...
  end;  { big }
```

The x referenced in sub1, is searched for where?

Example Pascal Procedure - II

In languages that use static scoping, some variable declarations can be hidden from some subprograms.

```
program main;
  var x : integer;
  procedure sub1;
    var x : integer;
    begin  { sub1 }
    ...x...
    end;  { sub1 }
  begin  { main }
  ...
  end.  { main }
```

The 'x' of main, is hidden from sub1

Blocks

Blocks are separate sections of code with their own internal scope.

- Blocks are a means of allowing new static scopes in the midst of executable code.
- Introduced in ALGOL 60, blocks allow a section of code to have its own local variables whose scope is minimized.
- Such variables (inside a block) are typically stack dynamic, so they have their storage allocated when the section is entered and deallocated when the section is exited.

Blocks, examples

```
...
declare TEMP : integer;
  begin
  TEMP := FIRST;
  FIRST := SECOND;
  SECOND := TEMP;
  end;
...
```

An Ada block

```
begin
write('Enter the number of disks: ');
readln(disks);
writeln('Solution:');
Hanoi('A','B','C',disks);
end.
```

A Pascal block

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

C++ blocks

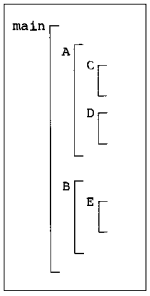
```
int a=2,b=3;
if(a<b){
    ...
    cout<<x<<endl;
    int x=9;
    //scope x begins here
    ...
}
```

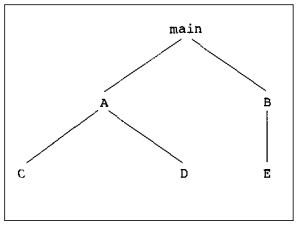
Diversion: How blocks help reduce bugs?

```
;; Language: R5RS Standard Scheme
(let ((empno (ssn-of employee-name)))
  (if (is-manager empno)
    (let ((employees (length (underlings-of empno))))
      (printf "~a has ~a employees working under him:~%" employee-name employees)
      (for-each
        (lambda(empno)
          ;; Within this lambda expression the variable empno refers to the ssn
          ;; of an underling.  The variable empno in the outer expression,
          ;; referring to the manager's ssn, is shadowed.
          (printf "Name: ~a, role: ~a~%"
                  (name-of empno)
                  (role-of empno)))
        (underlings-of empno)))))
```

Variable empno was used at various times, for different meanings, during the design and maintenance of this Fortran function. The scope-management of blocks avoids a potential logical error which could be hard to trace.

Some Issues with Static Scoping

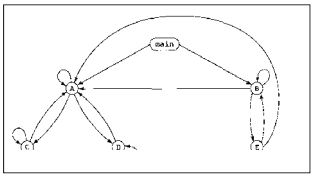




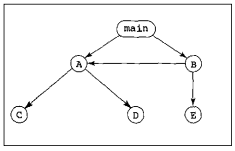
Procedures **main**, **A**, **B** , **C** and **D** are nested as in the tree above.

The required calling access is: main can call **A** and **B**, **A** can call **C** and **D**, and **B** can call **A** and **E**

Permissible access

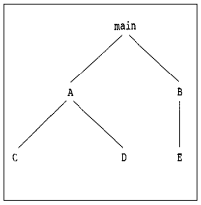


Required access

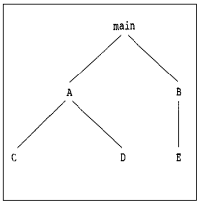


Another problem scenario for Static Scoping

- After a program has been developed and tested, a modification of its specification is required.
- Procedure **E** must now gain access to some variables of the scope of **D**.
- One way to provide that access is to move **E** inside the scope of **D**. But then **E** can no longer access the scope of **B**, which it presumably needs.

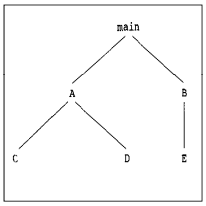


- Another solution is to move the variables defined in d that are needed by **E** into main. This would allow access by all the procedures, which would be more than is needed and thus creates the possibility of incorrect accesses.
- Suppose the variable that is moved to main is named x, and x is needed by **D** and **E**. But suppose that there is a variable named x declared in a. That would hide the correct x from its original owner, **D**.



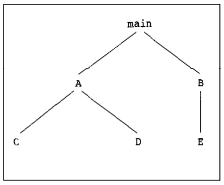
Two more issues arise when we move x to the main:

- This would allow access to x by all the procedures inside the scope of main, thus creating the possibility of incorrect accesses.
- It is harmful to readability to have the declaration of variables so far from their users.



Yet another problem scenario for Static Scoping

- Due to some specification change, procedure **E** needed to call procedure **D**.
- This could only be accomplished by moving **D** to nest directly in main, assuming that it was also needed by either **A** or **C**.
- However, it will then also lose access to the variables defined in **A**.
- This solution, when used repeatedly, results in programs that begin with long lists of low-level utility procedures.



The inflexibility of Static Scoping

- Getting around the restrictions of static scoping can lead to program designs that bear little resemblance to the original.
- Designers are encouraged to use far more globals than are necessary.
- All procedures can end up being nested at the same level, in the main program, using globals instead of deeper levels of nesting.
- The final design maybe awkward and contrived, and it may not reflect the underlying conceptual design.

Dynamic Scoping

Dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other (i.e. the way they are nested in code). Thus the scope can be determined only at run time.

Example of Dynamic Scoping – I

Assume that procedures are called in the following order: **main()** calls **compute()** calls **transform()** calls **initialize()**

Which **x** is accessed in transform?
Which **x** is accessed in initialize?

```
program main
  var y: Real;

  procedure compute()
    var x : Integer;

    procedure initialize()
      var x: Real;
    begin {initialize}
    ...
    end {initialize}

    procedure transform()
      var z: Real;
    begin {transform}
    ...
    end {transform}

  begin {compute}
  ...
  end {compute}
begin {main}
...
end {main}
```

*This is Pascal code, and Pascal uses Static Scoping; but we take it as an example of Dynamic Scoping for the sake of an example only.

Example of Dynamic Scoping – II

big calls sub2 which calls sub1; which **x** does sub1 use?

big calls sub1 directly; which **x** does sub1 use in this case?

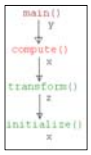
What algorithm is used to determine the dynamic scope of a variable?

```
procedure big;
  var x : integer;
  procedure sub1;
    begin  { sub1 }
    ... x ...
    end;  { sub1 }
  procedure sub2;
    var x : integer;
    begin  { sub2 }
    ...
    end;  { sub2 }
  begin  { big }
  ...
  end;  { big }
```

Algorithm to determine Dynamic Scope

– A directed graph of the order in which procedures are called during the execution of a program is kept. Following is the graph for the example given earlier:

main() calls **compute()** calls **transform()** calls **initialize()**




```
program main
  var y: Real;

  procedure compute()
    var x : Integer;

    procedure initialize()
      var x: Real;
    begin {initialize}
    ...
    end {initialize}

    procedure transform()
      var z: Real;
    begin {transform}
    ...
    end {transform}

  begin {compute}
  ...
  end {compute}
begin {main}
...
end {main}
```

Algorithm to determine the dynamic scope of a variable **v** declared in a procedure **p()** using a directed graph of procedure calls:

– Identify the sub-graph, **G**, of the directed graph that begins with the procedure **p()**.

– In this sub-graph, identify the *first* descendant node (i.e., procedure/function) that re-declares the variable v. Eliminate the entire sub-graph that begins with this node.

– The remaining nodes in the sub-graph **G** are in the dynamic scope of the variable **v** declared in procedure **p()**.

Issues with Dynamic Scoping

- The correct attributes of nonlocal variables visible to a program statement cannot be determined before execution.
- A statement in a subprogram that contains a reference to a nonlocal variable can refer to different nonlocal variables during different executions of the subprogram.
- During the time span beginning when a subprogram begins its execution and ending when that execution ends, the local variables of the subprogram are all visible to any other executing subprogram, regardless of its textual proximity.
  - There is no way to protect local variables from this accessibility. Subprograms are always executed in the immediate environment of the caller; therefore, dynamic scoping results in less reliable programs than static scoping.

– Type checking cannot be done before execution, because the type-binding is very late.

– Dynamic scoping makes programs much more difficult to read, because the calling sequence of subprograms must be known to determine the meaning of references to nonlocal variables. This may even become impossible for a human.

– Accesses to nonlocal variables in dynamic scoped languages take far longer than accesses to nonlocals when static scoping is used (reasons to be discussed later).

Assignment # 3 – Perl coding to test Dynamic Scoping

main() calls **compute()** calls **transform()** calls **initialize()**

```
procedure big;
  var x : integer;
  procedure sub1;
    begin  { sub1 }
    ... x ...
    end;  { sub1 }
  procedure sub2;
    var x : integer;
    begin  { sub2 }
    ...
    end;  { sub2 }
  begin  { big }
  ...
  end;  { big }
```

big calls sub2 which calls sub1; which **x** does sub1 use?

big calls sub1 directly; which **x** does sub1 use in this case?

Assignment # 3—Perl coding to test Dynamic Scoping

Simulate the behavior of the three Dynamic Scoping examples presented in this class, by implementing these functions in Perl and then calling them in the desired sequence.

- Initialize x with a different value on each declaration, and print x whenever it is accessed in a function, and also print "function_name() called" inside each function.

To be submitted on Saturday, March 3, 2012

Submit perl scripts only.

Perl tutorial

Scope and Lifetime

Although the scope and lifetime of a variable are clearly not the same because static scope is a textual, or spatial, concept whereas lifetime is a temporal concept, they at least appear to be related

- For example, the lifetime of a variable declared in a C++ if statement is exactly the same as the time it takes the if block to execute (after the variable declaration).

This apparent relationship between scope and lifetime does not hold in all situations.

- a **static** type variable inside a C++ function in an example.
- Following is another example of the unrelatedness of scope and lifetime: where does the scope of sum end, and where does its lifetime end?

```
void printheader() {
  ...
 }  /* end of printheader */
void compute() {
  int sum;
  ...
  printheader();
 }  /* end of compute */
```

Referencing Environments

The **referencing environment** of a statement is the collection of all names that are visible in the statement.

- The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible.
- In such a language, the referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.

Example from Pascal

What are the referencing environments at points 1, 2, 3 and 4 in the following?

Point	*Referencing Environment*
1	x and y of sub1, a and b of example
2	x of sub3, (x of sub2 is hidden), a and b of example
3	x of sub2, a and b of example
4	a and b of example

What if we allowed sub3 access to the variables of sub1?

```
program example;
  var a, b : integer;
  ...
  procedure sub1;
    var x, y : integer;
    begin  { sub1 }
    ...  <-------1
    end;  { sub1 }
  procedure sub2;
    var x : integer;
    ...
    procedure sub3;
      var x : integer;
      begin  { sub3 }
      ...  <-------2
      end;  { sub3 }
    begin  { sub2 }
    ...  <-------3
    end;  { sub2 }
  begin  { example }
  ...  <-------4
  end.  { example }
```

A subprogram is **active** if its execution has begun but has not yet terminated.

- The referencing environment of a statement in a dynamically scoped language is the locally declared variables, plus the variables of all other subprograms, called in a directed graph before the subprogram, that are currently active.

Reference environments in Dynamic Scoping

`main` calls `sub2`, which calls `sub1`.

Point	*Referencing Environment*
1	a and b of `sub1`, c of `sub2`, d of `main`, , (c of `main` and b of `sub2` are hidden)
2	b and c of `sub2`, d of `main`, (c of `main` is hidden)
3	c and d of `main`

```
void sub1() {
  int a, b;
  ...  <-------- 1
 }  /* end of sub1 */
void sub2() {
  int b, c;
  ...  <-------- 2
  sub1;
 }  /* end of sub2 */
void main() {
  int c, d;
  ...  <-------- 3
  sub2();
 }  /* end of main */
```