CSE 505

Lecture 2

August 29, 2012

PL Specification

Syntax
- context-free grammars, Backus-Naur Form, syntax charts

Semantics
- operational → execution
  axiomatic → verification
  denotational → definition

Operational Semantics

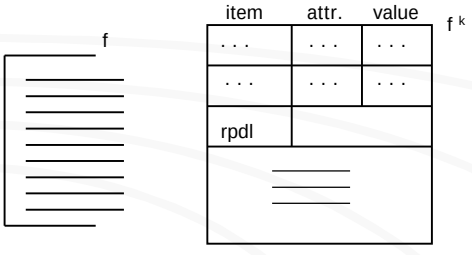
- High-level specification of execution
- "Abstract Machine"
- Textual vs Graphical (or Visual)
- Visual Operational Semantics

Program Unit and Contour

A program unit is typically a procedure-level unit, e.g., a procedure, function, coroutine, task, predicate.

A contour (frame) captures the execution state of the invocation of a program unit.

Program Unit vs Contour



rpdl = return-point + dynamic-link

Contents of a Contour

Data – information about variables, e.g. name, type, and value

Procedure – information about inner procedures

Linkage – information needed to continue execution after the contour terminates

Executable Code – in the case where a contour models the executable state of procedures, functions, coroutines, etc.

Important Note

The contour diagram does not model the details of expression evaluation and control structures such as for and while loops.

A contour diagram focuses on procedure-level issues, including recursive procedures, parameter-passing modes, non-local variable references, scope rules, etc.

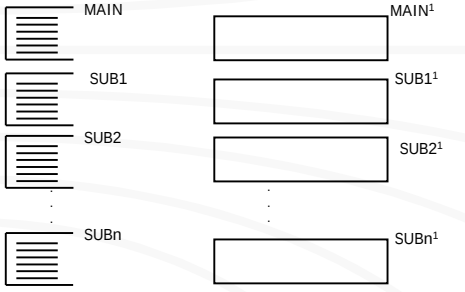
Contour Creation & Deletion

Two approaches to contour creation:

1. Static Allocation of Contours (FORTRAN)
   - one contour per program unit
   - no new contours created during execution
   - contour deletion when program terminates

2. Dynamic Allocation of Contours (Algol/Pascal/C)
   - contour creation when procedure is called
   - contour deletion when procedure ends

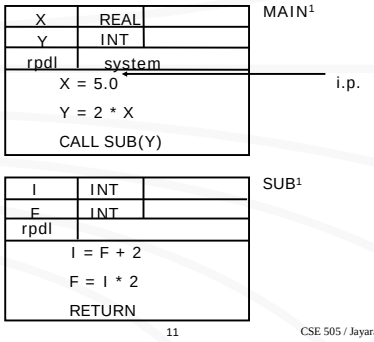
Static Allocation of Contours (Fortran)



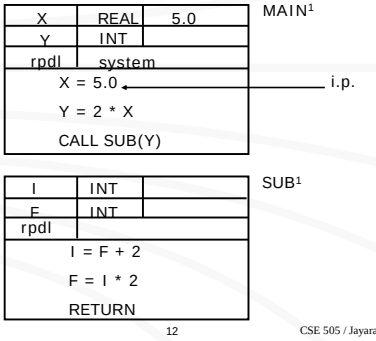

```
   PROGRAM MAIN
10 REAL X
20 INTEGER Y
30 X = 5.0
40 Y = 2 * X
50 CALL SUB(Y)
60 …
   END
   SUBROUTINE SUB(F)
10 INTEGER I,F
20 I = F + 2
30 F = I * 2
40 RETURN
END
```

Simple Fortran Program

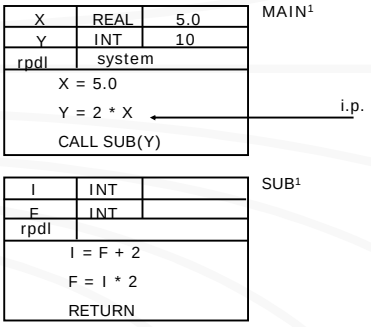
Fortran program execution



FORTRAN program execution

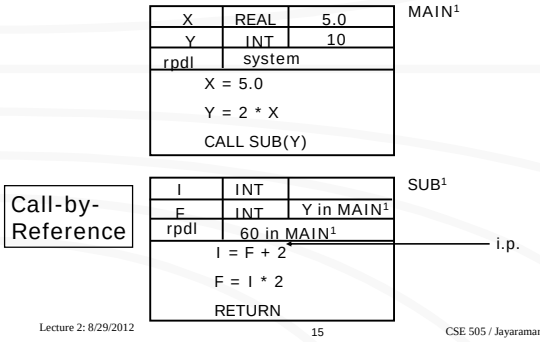


FORTRAN program execution

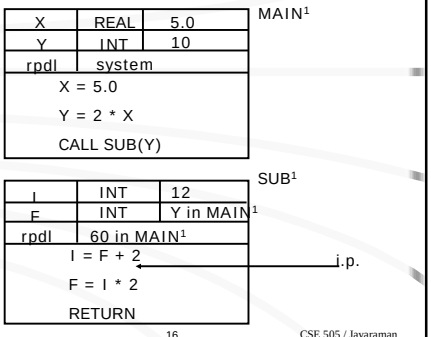



```
   PROGRAM MAIN
10 REAL X
20 INTEGER Y
30 X = 5.0
40 Y = 2 * X
50 CALL SUB(Y)
60 …
   END
   SUBROUTINE SUB(F)
10 INTEGER I,F
20 I = F + 2
30 F = I * 2
40 RETURN
END
```

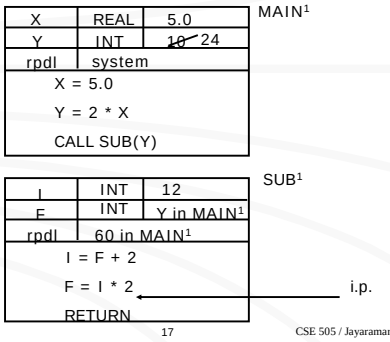

FORTRAN program execution



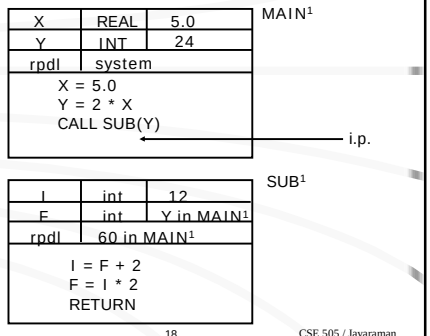
FORTRAN program execution



FORTRAN program execution



FORTRAN program execution



Aliasing

The modification of a non-local variable (e.g. Y in MAIN[1]) via a formal parameter (e.g. F) is referred to as a side-effect.

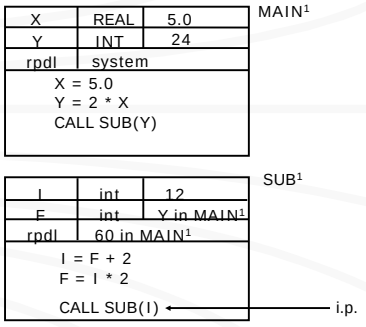
In general, the use of side-effects leads to subtle programming bugs.

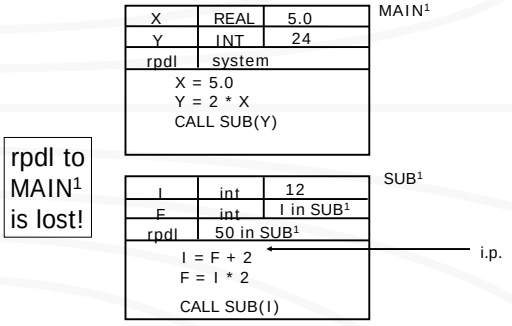
The previous program illustrates the semantics of reference parameters, and is not intended to illustrate good programming style.

Discussion

What if SUB calls itself recursively? Example:

```
SUBROUTINE SUB(F)
10 INTEGER I,F
20 I = F + 2
30 F = I * 2
40 CALL SUB(I)
50
   RETURN
END
```



Moral

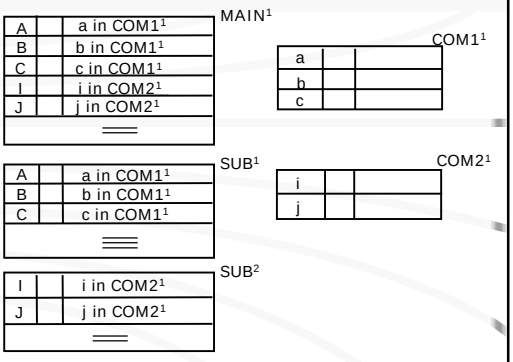
Recursion cannot be correctly executed using static allocation of contours!

History Sensitive Behavior*

- A Fortran subroutine F retains the values of its local variables from one call to the next.
- This allows F to exhibit "history-sensitive" behavior, i.e., the result for a call on F can use the values of the local variables of F at the end of the previous call.
- This property is useful for computing numerical series.
- This behavior can be simulated using static variables in other languages.

```
PROGRAM MAIN
COMMON /COM1/ A, B, C
COMMON /COM2/ I, J
…
END
SUBROUTINE SUB1
COMMON /COM1/ A, B, C
…
END
SUBROUTINE SUB2
COMMON /COM2/ I, J
…
END
```

Data Sharing via COMMON blocks



Dynamic Allocation of Contours

- Create contour when procedure is called.
- Delete contour when procedure terminates.
- Static/Lexical vs Dynamic Scoping.

Scope Rules

- Static/Lexical Scope Rule:
  An occurrence of an identifier is associated with the textually closest surrounding declaration.
- Dynamic Scope Rule:
  An occurrence of an identifier is associated with the dynamically closest surrounding declaration.


```
void f( ){
    int x;
    void g( ) {
        int x;
        . . . x . . .
    }
    void h( ) {
        . . . x . . .
        }
    }
```

Static Scope Rule

This occurrence of **x** is associated with **x in f**


```
void f( ){
    int x;
    void g( ) {
        int x;
        . . . x . . .
    }
    void h( ) {
        . . . x . . .
        }
    }
```

Dynamic Scope Rule

Calling Sequence
(i) **f→h**
x in f
(ii) **f→g→h**
x in g

Remarks

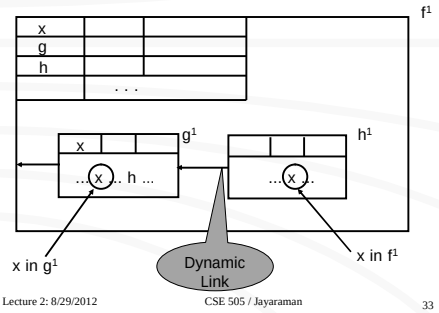
- Static scoping is commonly found in modern PLs, e.g., nested procedures (ANSI C), inner classes (Java), etc.
- Dynamic scoping is not so common. However:

  Which language construct commonly found in Java (and other languages) requires dynamic scoping?

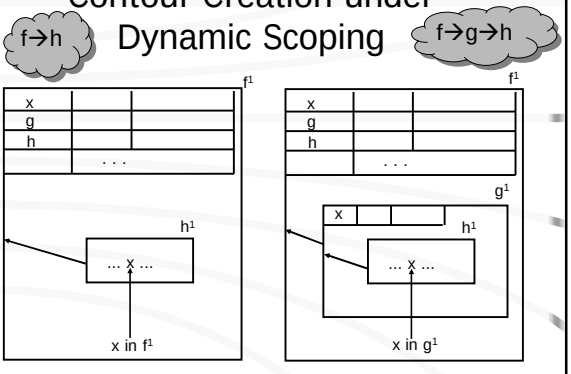
Contour Creation under Static Scoping

The j^{th} call of procedure h declared in contour f^i results in the creation of contour h^j nested inside f^i.

Contour Creation under Static Scoping



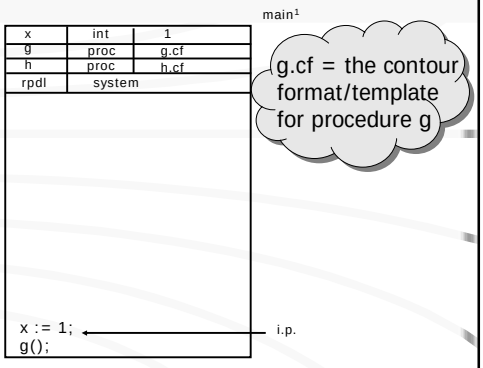
Contour Creation under Dynamic Scoping

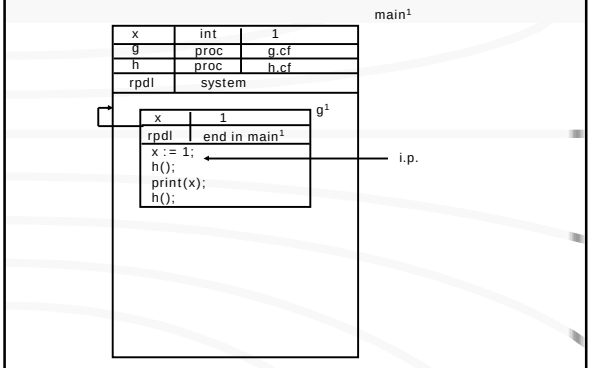


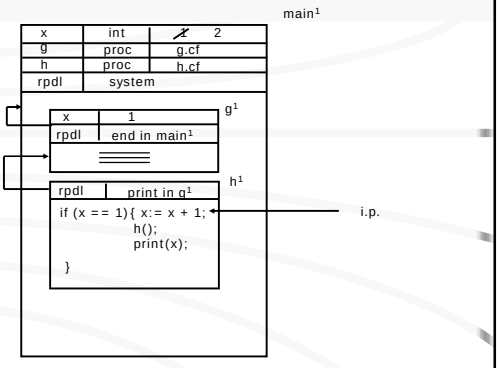
In subsequent discussions, we shall assume that the scope rule is Static Scoping, unless explicitly specified otherwise.

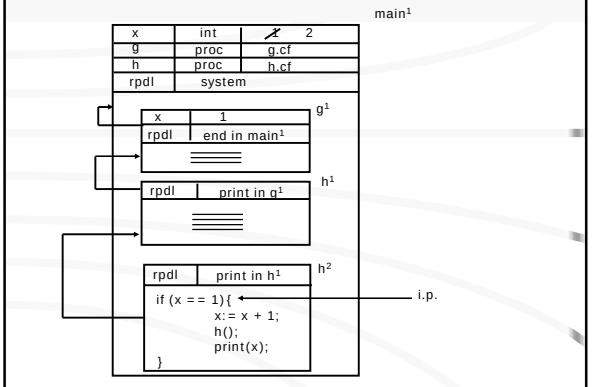

```
void  main() {
    int  x;
    void g() {
      int  x;
      x := 1;  h();  print(x);  h();
      }
    void h() {
      if (x == 1){ x:= x + 1;
                   h(); print(x);
                 }
      }
    x := 1;
    g();
}
```

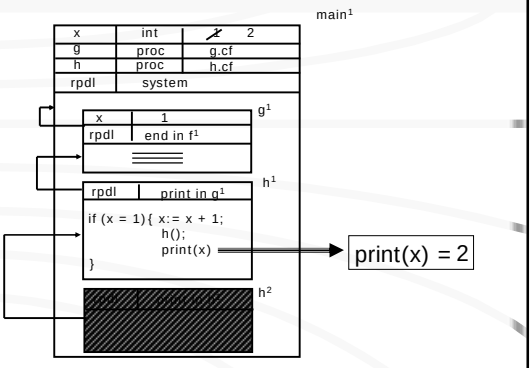
recursive call

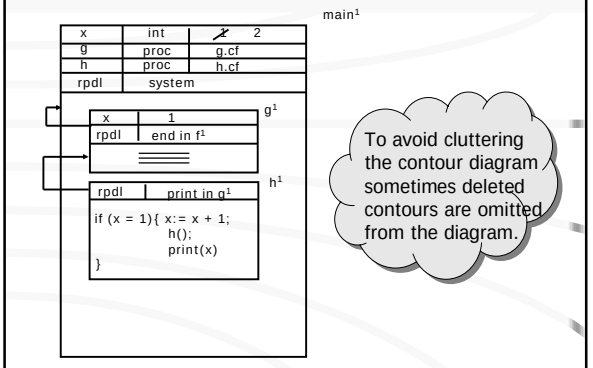

main¹
x int 1
g proc g.cf
h proc h.cf
rpdl system
g.cf = the contour format/template for procedure g
x := 1;
g();
i.p.

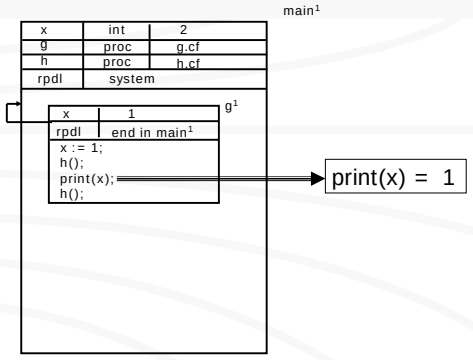

main¹
x int 1
g proc g.cf
h proc h.cf
rpdl system
g¹
x 1
rpdl end in main¹
x := 1;
h();
print(x);
h();
i.p.

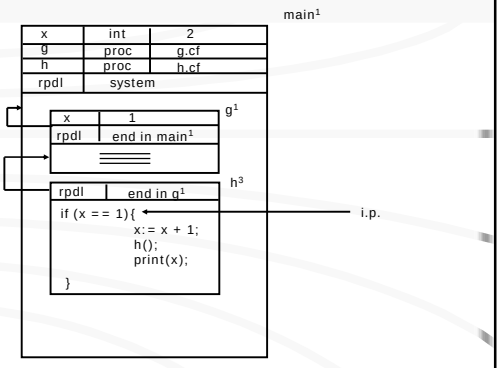

main¹
x int 2
g proc g.cf
h proc h.cf
rpdl system
g¹
x 1
rpdl end in main¹
h¹
rpdl print in g¹
if (x == 1){ x:= x + 1;
h();
print(x);
}
i.p.

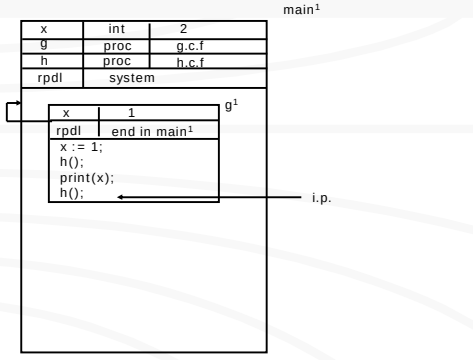

main¹
x int 2
g proc g.cf
h proc h.cf
rpdl system
g¹
x 1
rpdl end in main¹
h¹
rpdl print in g¹
h²
rpdl print in h¹
if (x == 1){
x:= x + 1;
h();
print(x);
}
i.p.

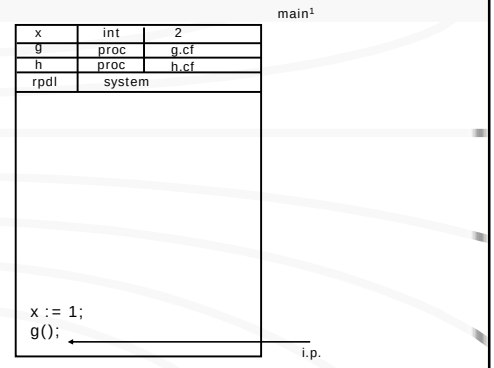

main¹
x int 2
g proc g.cf
h proc h.cf
rpdl system
g¹
x 1
rpdl end in f¹
h¹
rpdl print in g¹
if (x = 1){ x:= x + 1;
h();
print(x)
}
h²
print(x) = 2


main¹
x int 2
g proc g.cf
h proc h.cf
rpdl system
g¹
x 1
rpdl end in f¹
h¹
rpdl print in g¹
if (x = 1){ x:= x + 1;
h();
print(x)
}
To avoid cluttering the contour diagram sometimes deleted contours are omitted from the diagram.









Parameter Passing Modes

- Value (in)
- Result (out)
- Value-Result (inout)
- Reference (var or ref)
- Procedure (proc)
- Name (name)

General Comments

➢ Value parameters are used for sending input values for a function, e.g.
`factorial(in int n)`

➢ Result parameters are used for obtaining resulting values from a function, e.g.
`search(in int key, out int value, out boolean status)`

➢ Value-result parameters combine the capabilities of value and result parameters, e.g.
`normalize(inout int num, den).`