Parameter Passing

The semantics of passing arguments to routines...

- **Call-by-value**
- **Call-by-reference**
- **Call-by-name**
- **Call-by-copy-restore**

Inline Expansion

Parameter Passing

"Call-by-value"
"Pass-by-value"

```
procedure foo (a:int)
   ...
endProc

...
foo(x+5);
```

Each formal parameter is a local variable.
- Will get a slot in the activation record
- Can be updated

Caller's code evaluates the argument expression.

The calling sequence moves that value into the callee's variable.
(like initializing a local variable...)

Commonly used:
Java, PCAT, C, C++, Haskell, Smalltalk, ...

Source:

```
foo (x+3, y*7, z-9);
```

IR Code:

```
t5 := ...

t7 := ...

t9 := ...

param 1,t5
param 2,t7
param 3,t9

call foo
```


Source:

```
foo (x+3, y*7, z-9);
```

IR Code:

```
t5 := ...

t7 := ...

t9 := ...

param 1,t5
param 2,t7
param 3,t9

call foo
```

Target Code:

```
...
st   %l0,[fp+offset_t5]
...
st   %l0,[fp+offset_t7]
...
st   %l0,[fp+offset_t9]
ld   [%fp+offset_t5],%o0
ld   [%fp+offset_t7],%o1
ld   [%fp+offset_t9],%o2
call foo
nop
```

Source:

```
procedure foo (p1,p2,p3: integer)
    ... end;
```

IR Code:

```
procEntry  foo,lexlevel=4
formal  1,p1
formal  2,p2
formal  3,p3
```


Source:

```
procedure foo (p1,p2,p3: integer)
    ... end;
```

IR Code:

```
procEntry  foo,lexlevel=4
formal  1,p1
formal  2,p2
formal  3,p3
```

Target Code:

```
        save ...
        <set up display register>
        st   %i0,[fp+offset_p1]
        st   %i1,[fp+offset_p2]
        st   %i2,[fp+offset_p3]
        ...
```

Parameter Passing

"Call-by-reference"
"Pass-by-reference"

The argument in the call...

- **A variable**

  The address of the variable is passed

Within the routine...

All accesses to the formal parameter are compiled using indirection

Parameter Passing

"Call-by-reference"
"Pass-by-reference"

The argument in the call...

- **A variable**

  (i.e., an L-Value)
  The address of the variable is passed
- **An expression**

  (i.e., an R-Value)
  The expression is evaluated and put into a temporary
  The address of the temporary is passed

Within the routine...

All accesses to the formal parameter are compiled using indirection

Parameter Passing

"Call-by-reference"
"Pass-by-reference"

```
foo(x, y+5);
```

```
procedure foo (a,b: int)
  ...
    a  :=  b  + 1;

  ...
endProc
```

The argument in the call...

- **A variable**
  - (i.e., an L-Value)
  - The address of the variable is passed
- **An expression**
  - (i.e., an R-Value)
  - The expression is evaluated and put into a temporary
  - The address of the temporary is passed

Within the routine...

All accesses to the formal parameter are compiled using indirection

Parameter Passing

"Call-by-reference"
"Pass-by-reference"

```
~~foo(x, y+5);~~
t := y+5;
foo(&x,&t);
```

```
procedure foo (a,b: int)
  ...
    a  :=  b  + 1;

  ...
endProc
```

The argument in the call...

- **A variable**
  - (i.e., an L-Value)
  - The address of the variable is passed
- **An expression**
  - (i.e., an R-Value)
  - The expression is evaluated and put into a temporary
  - The address of the temporary is passed

Within the routine...

All accesses to the formal parameter are compiled using indirection

Parameter Passing

"Call-by-reference"
"Pass-by-reference"

```
foo(x, y+5);
t := y+5;
foo(&x,&t);

procedure foo (a,b:*int)
   ...
   a := b + 1;
   (*a) := (*b) + 1;
   ...
endProc
```

The argument in the call...

- A variable
  (i.e., an L-Value)
  The address of the variable is passed
- An expression
  (i.e., an R-Value)
  The expression is evaluated and put into a temporary
  The address of the temporary is passed

Within the routine...

All accesses to the formal parameter are compiled using indirection

Does "C" Have Call-By-Reference?

No (not directly)

All arguments in C are passed by value.

Does "C" Have Call-By-Reference?

No (not directly)

All arguments in C are passed by value.

However...

The programmer can work with pointers.

Got a large struct?

Option 1: Pass it directly.
Call by value involves copying... Ugh!

Option 2: Pass a pointer to it.
The pointer is passed by value.

Pass by reference can be achieved when desired!

Does "C" Have Call-By-Reference?

No (not directly)

All arguments in C are passed by value.

However...

The programmer can work with pointers.

Got a large struct?

Option 1: Pass it directly.
Call by value involves copying... Ugh!

Option 2: Pass a pointer to it.
The pointer is passed by value.

Pass by reference can be achieved when desired!

Arrays

Arrays are interchangeable with pointers

Arrays are passed by reference

```
char a[100];
...
foo (a);
foo (&a[0]);
...
void foo (char p[]) {...
void foo (char * p) {...
```

Does "Java" Have Call-By-Reference?

No

All arguments are passed by value in Java.

Java works with...

Objects
Pointers to objects

When an object is passed to a method...

```
BinaryOp p;
...
x.foo(p);
```

Objects are never passed directly.
Instead...
Pointers to objects are passed (by value).

In some sense, objects are always passed by reference.

Copy-Restore Semantics

"Call-by-Copy-Restore"
"Call-by-Value-Result"

Upon Calling...

- Look at the actual arguments
- Determine their L-Values (i.e., get their addresses)
  (If there is an expression, evaluate it into a temporary first)
- Save these addresses.
- Parameters are local variables in the routine.
- Copy values into the parameter variables

Within the Routine...

- Okay to read and update the parameters.

Upon Return...

- Copy the values from the parameters
  using the saved addresses.
- Arguments that were expressions?
  Copy result into temp, then discard the temp.

Example:

```
procedure foo (in a,
               in out b,
               out c: real)
  ...
endProc
```

"Inline Expansion"

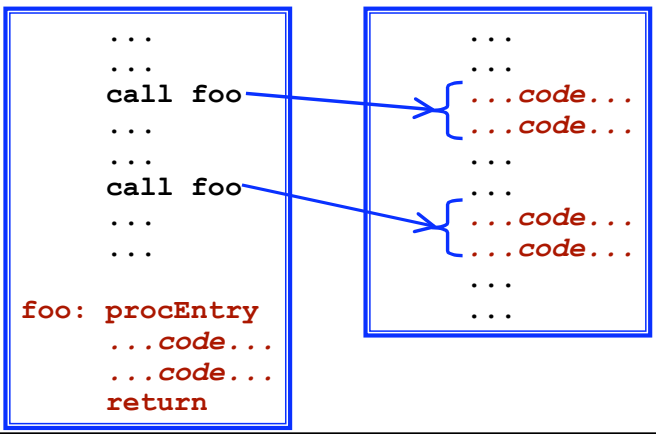
Got a routine called from several places?
Copy the routine body straight into the code.
    Replace the "call" with the body of the routine.
Why?
    Execution efficiency!!!
    (Eliminate the overhead of the call / return sequences)

```
      ...
      ...
      call foo
      ...
      ...
      call foo
      ...
      ...

foo:  procEntry
      ...code...
      ...code...
      return
```

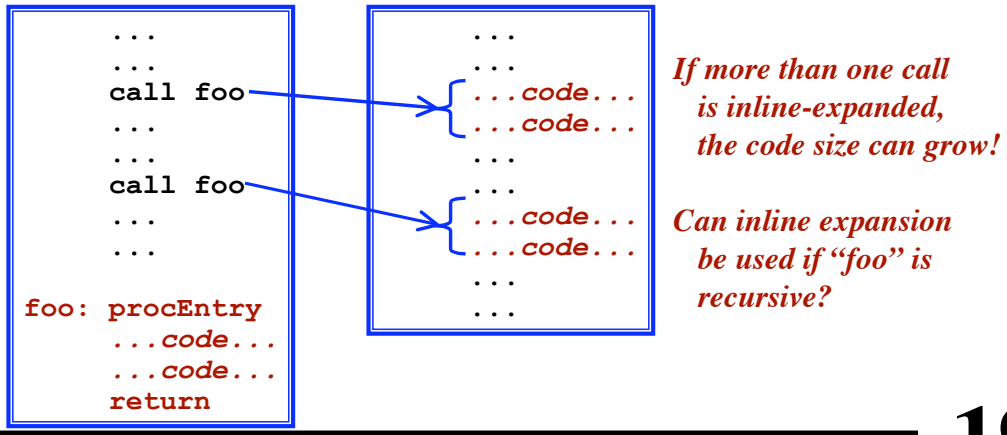

"Inline Expansion"

Got a routine called from several places?
Copy the routine body straight into the code.
    Replace the "call" with the body of the routine.
Why?
    Execution efficiency!!!
    (Eliminate the overhead of the call / return sequences)



"Inline Expansion"

Got a routine called from several places?
Copy the routine body straight into the code.
Replace the "call" with the body of the routine.
Why?
Execution efficiency!!!
(Eliminate the overhead of the call / return sequences)



If more than one call is inline-expanded, the code size can grow!

Can inline expansion be used if "foo" is recursive?

"Inline Expansion"

```
procedure foo (x, y : int) : int is
  var a, b : int
    ...<code containing x, y, a, b >...
    return ...<ret-expr>...;
    ...<code containing x, y, a, b >...
endProc
```

```
...
...
i := foo (expr1,expr2);
...
...
j := foo (expr3,expr4);
...
...
```

GOAL:
Substitute this body of code into each call site.

"Inline Expansion"

```
procedure foo (x, y : int) : int is
  var a, b : int
    ...<code containing x, y, a, b >...
    return ...<ret-expr>...;
    ...<code containing x, y, a, b >...
endProc
```

```
  ...
  ...
  i := foo (expr1,expr2);
  ...
  ...
  j := foo (expr3,expr4);
  ...
  ...
```

GOAL:

Substitute this body of code into each call site.

You can do this at home, too!

"Inline Expansion"

```
procedure foo (x, y : int) : int is
  var a, b : int
    ...<code containing x, y, a, b >...
    return ...<ret-expr>...;
    ...<code containing x, y, a, b >...
endProc
```

```
  ...
  ...
  i := foo (expr1,expr2);
  ...
  ...
  j := foo (expr3,expr4);
  ...
  ...
```

Look at first call site

Need a new temp for each parameter and each local:

```
x ⇒ t1
y ⇒ t2
a ⇒ t3
b ⇒ t4
```

"Inline Expansion"

```
procedure foo (t1,t2: int) : int is
  var t3,t4: int
    ...<code containing t1,t2,t3,t4>...
    return ...<ret-expr>...;
    ...<code containing t1,t2,t3,t4>...
endProc
```

```
  ...
  ...
  i := foo (expr1,expr2);
  ...
  ...
  j := foo (expr3,expr4);
  ...
  ...
```

Need a new temp for each parameter and each local:

x ⇒ t1
y ⇒ t2
a ⇒ t3
b ⇒ t4

"Inline Expansion"

```
procedure foo (t1,t2: int) : int is
  var t3,t4: int
    ...<code containing t1,t2,t3,t4>...
    return ...<ret-expr>...;
    ...<code containing t1,t2,t3,t4>...
endProc
```

```
  ...
  ...
  t1:= expr1;
  t2:= expr2;
  ...<code containing t1,t2,t3,t4>...
  t5 := ...<return-expr>...;
  goto label_54;
  ...<code containing t1,t2,t3,t4>...
Label_54:
  i := t5;
  ...
  ...
  j := foo (expr3,expr4);
  ...
  ...
```

Need a new temp for each parameter and each local:

x ⇒ t1
y ⇒ t2
a ⇒ t3
b ⇒ t4

"Inline Expansion"

```
procedure foo (x, y : int) : int is
  var a, b : int
    ...<code containing x, y, a, b >...
    return ...<ret-expr>...;
    ...<code containing x, y, a, b >...
endProc
```

```
  ...
  ...
  t1:= expr1;
  ...
  i := t5;
  ...
  ...
  j := foo (expr3,expr4);
  ...
  ...
```

Now, look at next call site

"Inline Expansion"

```
procedure foo (t6,t7: int) : int is
  var t8,t9: int
    ...<code containing t6,t7,t8,t9>...
    return ...<ret-expr>...;
    ...<code containing t6,t7,t8,t9>...
endProc
```

```
  ...
  ...
  t1:= expr1;
  ...
  i := t5;
  ...
  ...
  j := foo (expr3,expr4);
  ...
  ...
```

Need a new temp for each parameter and each local:

x ⇒ t6
y ⇒ t7
a ⇒ t8
b ⇒ t9

"Inline Expansion"

```
procedure foo (t6,t7: int) : int is
  var t8,t9: int
    ...<code containing t6,t7,t8,t9>...
    return ...<ret-expr>...;
    ...<code containing t6,t7,t8,t9>...
endProc
```

```
  ...
  ...
  t1:= expr1;
  ...
  i := t5;
  ...
  ...
  t6:= expr3;
  t7:= expr4;
  ...<code containing t6,t7,t8,t9>...
  t10 := ...<return-expr>...;
  goto label_73;
  ...<code containing t6,t7,t8,t9>...
Label_73:
  j := t10;
  ...
  ...
```

Need a new temp for each parameter and each local:

x ⇒ t6
y ⇒ t7
a ⇒ t8
b ⇒ t9

"Inline Expansion"

```
procedure foo (t6,t7: int) : int is
  var t8,t9: int
    ...<code containing t6,t7,t8,t9>...
    return ...<ret-expr>...;
    ...<code containing t6,t7,t8,t9>...
endProc
```

```
  ...
  ...
  t1:= expr1;
  ...
  i := t5;
  ...
  ...
  t6:= expr3;
  t7:= expr4;
  ...<code containing t6,t7,t8,t9>...
  t10 := ...<return-expr>...;
  goto label_73;
  ...<code containing t6,t7,t8,t9>...
Label_73:
  j := t10;
  ...
  ...
```

NOTE:
This is "call-by-value" parameter passing semantics

Need a new temp for each parameter and each local:

x ⇒ t6
y ⇒ t7
a ⇒ t8
b ⇒ t9

Parameter Passing

"Call-by-Name"
"Pass-by-Name"

The original semantic definition of argument passing...

Systematically rename all of the local variables and formals in the program so that every variable has a unique name. (Every "use" is clearly tied to only one definition.)

Treat each routine as a macro. That is...

- *In every call expression, add a set of parentheses around each argument expression.*
- *Substitute the entire body of the routine into the point where the routine is called, replacing the calling expression.*
- *Substitute the actual argument expressions for every occurrence of a formal in the routine's body.*

Example of Call-by-Name

```
procedure foo (x, y: int) : int is
  begin
    return (x * y);
  end
...
x := foo (y, z+3);
...
```

Example of Call-by-Name

```
procedure foo (x, y: int) : int is
  begin
    return (x * y);
  end
...
x := foo (y, z+3);
...
```

```
procedure foo (x1, y1: int) : int is
  begin
    return (x1 * y1);
  end
```


Example of Call-by-Name

```
procedure foo (x, y: int) : int is
  begin
    return (x * y);
  end
...
x := foo (y, z+3);
...
```

```
procedure foo (x1, y1: int) : int is
  begin
    return (x1 * y1);
  end
```

```
x := foo ((y), (z+3));
```

Example of Call-by-Name

```
procedure foo (x, y: int) : int is
  begin
    return (x * y);
  end
...
x := foo (y, z+3);
...
```

```
procedure foo (x1, y1: int) : int is
  begin
    return (x1 * y1);
  end
```

```
x := foo ((y), (z+3));
```

```
x := ((y) * (z+3));
```


Problems with Call-By-Name

```
procedure swap (x, y: int) is
  var t: int
  begin
    t := x;
    x := y;
    y := t;
  end;
```

Problems with Call-By-Name

```
procedure swap (x, y: int) is
  var t: int
  begin
    t := x;
    x := y;
    y := t;
  end;
```

```
...
swap (i,a[i]);
...
```


Problems with Call-By-Name

```
procedure swap (x, y: int) is
  var t: int
  begin
    t := x;
    x := y;
    y := t;
  end;
```

```
...
swap (i,a[i]);
...
```

```
...
t := i;
i := a[i];
a[i] := t;
...
```

Problems with Call-By-Name

```
procedure swap (x, y: int) is
  var t: int
  begin
    t := x;
    x := y;
    y := t;
  end;
```

```
...
swap (i,a[i]);
...
```

```
...
t := i;
i := a[i];
a[i] := t;
...
```

Apparently, swap cannot be written using call-by-name semantics!!!