Function Calls

A function is a parameterized portion of code within a larger program which performs a specific task.

A function in a programming language is in many ways similar to a mathematical function, but can have side-effects outside of the simple "function return value" .

Function Calls

What exactly happens at function calls? Consider this simple example in a C like language:

```
fun f (x) {
    var t = x + 1;
    return t;
}

...

var i = f(3);
print(i);

...
```

Observations:

- In many programming languages communication to a function is via *formal* and *actual* parameters *by value*.
- Communication from a function is via *return values*.
- Functions have local scope – locally defined variables.
- The formal parameter are declared as local variables.

Function Calls

How do we interpret function definitions and calls?

```
fun f (x) {
    var t = x + 1;
    return t;
}

...

var i = f(3);
print(i);

...
```

```
var f = fun (x) {
    var t = x + 1;
    return t;
}

...

var i = f (x = 3);
print(i);

...
```

Observations:

- Function names act like variables that have been initialized with a *function value*!
- During a function call we simply look up the function stored in the variable, activate it, and pass it the actual parameters.
- The actual parameters act like initial values for the formal parameters during a function call.

Function Parameters

There are two different ways to associate actual with formal parameters:

Positional Correspondence – Here the actual parameters are associated with the formal parameters via their position in the actual and formal parameter lists.

```
fun foo(a,b) ...
... call foo(1,2)...
```

Here actual parameter 1 will be associated with the formal parameter `a` and actual parameter 2 will be associated with formal parameter `b`.

Keyword Correspondence – Here the actual arguments are associated with the formal parameters by an explicit assignment.

```
fun foo(a,b) ...
... call foo(a=1,b=2)...
```

Function Parameters

Our implementation:

- We implement function parameters by call-by-value using keyword correspondence.
- The biggest problem we will be facing are *side effects*; function calls can modify global variables and functions can appear in both arithmetic and boolean expressions! We need to be able to model both of these aspects of functions in imperative languages.

Function Calls

In our language we introduce new syntactic structures to deal with functions. The first one is the *function call*:

```
A ::= call(f,[ PL ])  |  call(f, [ ])
```

here `f` represents function names and `PL` is the non-terminal for actual parameter lists,

```
PL ::= P , PL  |  P
P  ::= assign(x,A)
```

The second structure is the *function definition*:

```
C  ::= fun(f,[ FL ],C,A)  |  fun(f,[ ],C,A)
FL ::= x , FL  |  x
```

Here `FL` is the formal argument list. This is a highly stylized version of a function definition, think of it as

```
fun f(x) is C returns A
```

Function Calls

This will enable us to write programs such as

```
fun inc(x) is skip returns add(x,1) seq
var(q) seq
assign(q,call(inc,[assign(x,q)]))
```

Or consider the factorial program

```
fun fact(i) is
        var(result) seq
        if(eq(i,1),
            assign(result,1),
            assign(result,mult(i,call(fact,[assign(i,sub(i,1))]))
    returns result seq
var(x) seq
assign(x,call(fact,[assign(i,3)]))
```

Function Calls

The semantics of a function declaration is binding a *funval* to a *function variable*,

```
(fun(F,L,C,A),State) -->> OState :-        % function declaration
    declarevar(F,funval(L,C,A),State,OState).
```

The semantics of a function call is returning a value from the execution of the function,

```
(call(F,[ ]),State) -->> Val :-        % function call
    lookup(F,State,funval([ ],C,A)),
    pushenv(State,LocalState),
    (C,LocalState) -->> S1,
    (A,S1) -->> Val,
    popenv(S1,_),!.       % TROUBLE!!!
```

```
(call(F,PList),State) -->> Val :-        % function call
    lookup(F,State,funval(FList,C,A)),
    pushenv(State,LocalState),
    declareparams(FList,LocalState,S1),
    initparams(PList,S1,S2),
    (C,S2) -->> S3,
    (A,S3) -->> Val,
    popenv(S3,_),!.       % TROUBLE!!!
```

Note: We are only dealing with side effect free functions. Why?

Function Calls

In order to include side effects we need to create a new semantic function that interprets arithmetic expressions, including function calls, and carries side effect information with it.

```
(call(F,[ ]),State) -->> (Val,OState) :-        % function call
    lookup(F,State,funval([ ],C,A)),
    pushenv(State,LocalState),
    (C,LocalState) -->> S1,
   (A,S1) -->> (Val,S2),
    popenv(S2,OState),!.

(call(F,PList),State) -->> (Val,OState) :-        % function call
    lookup(F,State,funval(FList,C,A)),
    pushenv(State,LocalState),
    declareparams(FList,LocalState,S1),
    initparams(PList,S1,S2),
    (C,S2) -->> S3,
    (A,S3) -->> (Val,S4),
    popenv(S4,OState),!.
```

Function Calls

Side effects impose an ordering on the evaluation of operands in an expression:

```
(add(A,B),State) -->> (Val,OState) :-      % addition
     (A,State) -->> (ValA,S),
     (B,S) -->> (ValB,OState),
     Val xis ValA + ValB,!.
```

compare this to our side effect free semantics:

```
(add(A,B),State) -->> Val :-       % addition
     (A,State) -->> ValA,
     (B,State) -->> ValB,
     Val xis ValA + ValB,!.
```

Function Calls

In optimizing programming languages function calls with side effects together with boolean expressions are a notorious source of bugs. Consider this C program:

```
#include <stdio.h>

int cnt = 0;
int x = 0;

int f(void) {
  cnt++;
  return x;
}

void main(void) {

  if (f() && f())
    printf("condition is true\n");
  else
    printf("condition is false\n");

  printf("function f was called %d time(s)\n",cnt);
}
```

What kind of output would you expect with `x == 0`? What kind of output would you expect with `x == 1`?

Function Calls

Side effects ripple through the whole language definition. Consider the boolean expressions:

```
(not(A),State) -->> (Val,OState) :-      % not
    (A,State) -->> (ValA,OState),
    Val xis (not ValA),!.

(and(A,B),State) -->> (Val,OState) :-     % and
    (A,State) -->> (ValA,S),
    (B,S) -->> (ValB,OState),
    Val xis (ValA and ValB),!.
```

Consider the statements:

```
(assign(X,A),State) -->> OState :-      % assignment
    lookup(X,State,_),
    (A,State) -->> (ValA,S),
    bindval(X,ValA,S,OState),!.

(put(A),State) -->> OState :-            % writing
    (A,State) -->> (ValA,OState),
    write(A),
    write(' is '),
    writeln(ValA),!.
```