Compiler Construction

Cheat Sheet: Code Generation

See also:

- `http://people.cs.uu.nl/atze/SSM/index.html`,
- `http://www.cs.uu.nl/wiki/pub/Cco/MiniProjects/imp-0.0.4.tar.gz`.

Basic Pushing and Popping

Integer and boolean constants are simply pushed onto the stack. For example: the constant 42 results into the instruction ldc 42:

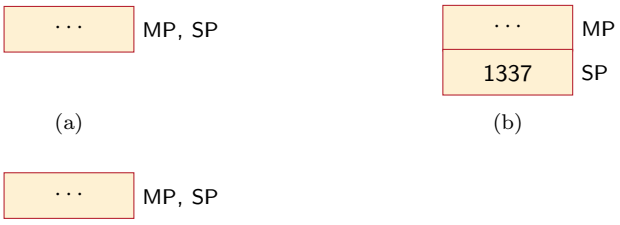
(a) (b).

The boolean constants `false` and `true` are represented by, respectively the integers 0 and 1. When reading a boolean from the stack 0 is interpreted as `false`, while any nonzero integer is interpreted as `true`.

The instruction trap 0 pops the top of the stack and outputs the popped value to the execution environment. For example, the statement **print** 1337; is compiled into

```
ldc  1337
trap 0,
```

which first pushes the constant 1337 onto the stack and then pops the stack, writing 1337 to the environment:



(a) (b)

(c).

In general, a statement **print** e; results in the code for e followed by the pop instruction trap 0.

Arithmetic and Relational Operations

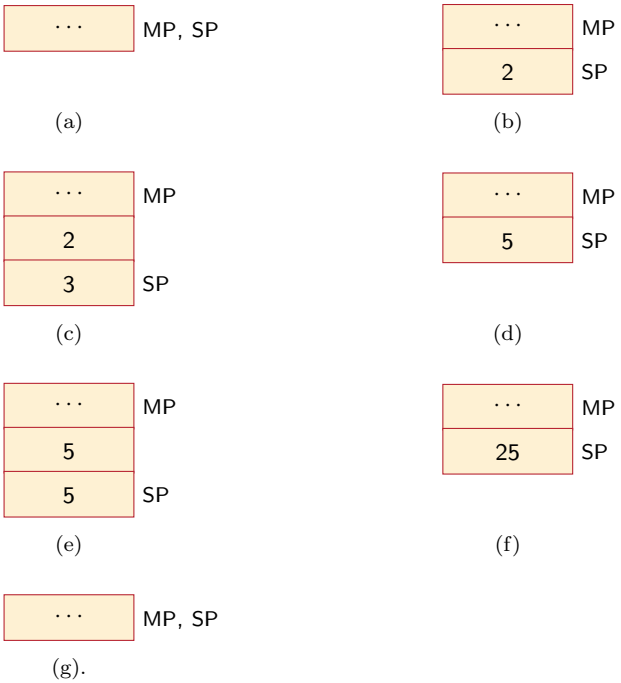
Expressions involving binary operators are compiled into code that pushes the operands onto the stack, followed by the instruction corresponding to the operation. For instance, $2 + 3$ results in

```
ldc  2
ldc  3
add.
```

Since instructions like add pop their operands from the stack and push their results, this scheme nicely works out for nested expressions. Consider for example the statement **print** $(2 + 3) * 5$, compiled into

```
ldc  2
ldc  3
add
ldc  5
mul
trap 0
```

and resulting in the stack sequence



Branching

Conditional statements **if** (e) S_1 **else** S_2 are compiled using unconditional branch instructions (bra) and conditional branch instructions (brf). For example, the statement **if** $(2 \equiv 3)$ **print** 5; **else print** 7; results in

```
        ldc 2
        ldc 3
        eq
        brf ℓ0
        ldc 5
        trap 0
        bra ℓ1
ℓ0 :    ldc 7
        trap 0
ℓ1 :    nop.
```

for some "fresh" labels ℓ_0 and ℓ_1. That is, the result of the guard expression is popped from the stack; if it is zero (i.e., `false`), execution continues at ℓ_0; if it is nonzero (`true`) execution continues, as usual, with the next instruction. Following the code for the `true`-branch, an unconditional branch instruction makes sure that the instructions generated for the `false`-branch are skipped.

Sequencing

Sequences of statements, $\{S_1 \cdots S_n\}$, are compiled by just concatenating the code for the individual statements in order. For instance, the sequence

```
{   print 2;
    print 3;   }
```

results in

```
ldc  2
trap 0
ldc  3
trap 0.
```

Local Variables

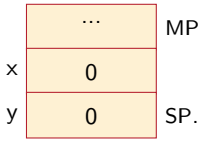
For the local variables declared in a block of statements, we allocate space on the stack that we initially fill with zero-values for the variables. For instance, if a block declares two variables, as in

```
{   var x;
    var y;
    ···      },
```

allocation simply consists of pushing two intial values onto the stack:

```
ldc 0
ldc 0.
```

Executing the allocation code thus results in intial values directly following the stack cell pointed at by the mark pointer:



Note that, in this example, the values of the variables x and y can be found at offsets 1 and 2 respectively from the address pointed at by the mark pointer. So, to assign values to these variables, as in, for example,

$$\begin{array}{ll} \{ & \cdots \\ & y = 5; \\ & \cdots \qquad \}, \end{array}$$

we simply have to push the MP-address on the stack and then use it to write to a location relative to the pushed address:

```
ldc 5
ldr MP
sta 2.
```

The same can be accomplished more directly and, hence, more efficiently, by using the instruction stl:

```
ldc 5
stl 2.
```

Retrieving the value of a variable is done using the instruction ldl, which pushes a value found at a specified offset relative to the MP-address. For example,

$$\begin{array}{ll} \{ & \cdots \\ & \textbf{print } x + y; \\ & \cdots \qquad \qquad \}, \end{array}$$

is compiled into

```
ldl  1
ldl  2
add
trap 0.
```

When compiling blocks of statements involving local variables, we make use of a symbol table that maps from variables to MP-relative offsets.

The code generated for a block is followed by code that pops the local variables, restoring the stack layout from before the execution of the block.

Functions

When a function is called, we update the mark pointer and let it point to the address that is on top of the stack when the execution of the code for the funtion starts; right before the update, the address then pointed at by the mark pointer is saved on the stack:

```
ldr  MP
ldrr MP SP.
```

Directly following the code for updating the MPcomes the code for the body of the function, which typically begins with code for allocating the local variables of a function (in the fashion described above).

The code that is issued for calling a function is responsible for pushing the arguments onto stack, followed by a so-called *static link* for the function (see Global Variables, below) and a return address. (Consult the implementation for details on the code that is generated for a function call.)

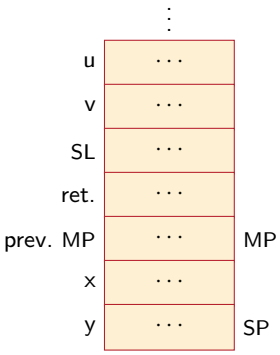
Thus, every function call gives rise to a *stack frame* or *activation record* with a fixed layout:

function arguments
static link
return address
previous MP-address
local arguments.

For instance, a call to a function f,

function $f\ (u, v)$
{ **var** x;
var y;
...
},

establishes the following stack layout:



Note that, from within the body of f the arguments u and v are available at MP-relative offsets -4 and -3; when compiling the body, this information should be stored in the symbol table.

When execution of the body has finished, the result of the function is supposed to be on top of the stack. The complete the function call, we then overwrite the value of the first argument[1] (u at offset -4, in our example) with the result value and we pop all local arguments:

```
stl  -4
ldrr SP MP.
```

Next, we restore the previous MP-address by loading the saved address into the MP-register:

```
ldr MP.
```

[1] If the function has no arguments, the static link is overwritten.

Then, the return address is moved just next to the result value, and the stack popped so that the return address is on top of the stack, after which the actual jump to the return address can be performed, leaving the result value on top of the stack:

```
sts −2
ajs −1
ret.
```

Global Variables

At each point in a program's execution the stack is partioned as a sequence of activation records that describe the dynamic nesting of function calls. These activation records are "linked together" by a chain of saved MP-addresses. Therefore, these saved addresses are sometimes also referred to as *dynamic links*. For example, consider an interleaving of calls to functions f and g,

```
function f (u, v)
{   var x;
    var y;
    ··· g (···) ···
}

function g (k)
{   var m;
    var n;
    ··· f (···, ···) ···
},
```

and the corresponding chain of activation records depicted in Figure 1.

The static link for a function call (available at offset -2 relative to the MP-address) estalished—in contrast to the dynamic link, which connects to the *calling* context—a connection to the context in which the function was declared. For example, consider, once more, an interleaving of calls to functions f and g, but this time in the context of a global variable p declared at the same lexical level as f and g:

```
var p;

function f (u, v)
{   var x;
    var y;
    ··· g (···) ···
}

function g (k)
{   var m;
    var n;
    ··· f (···, ···) ···
}.
```

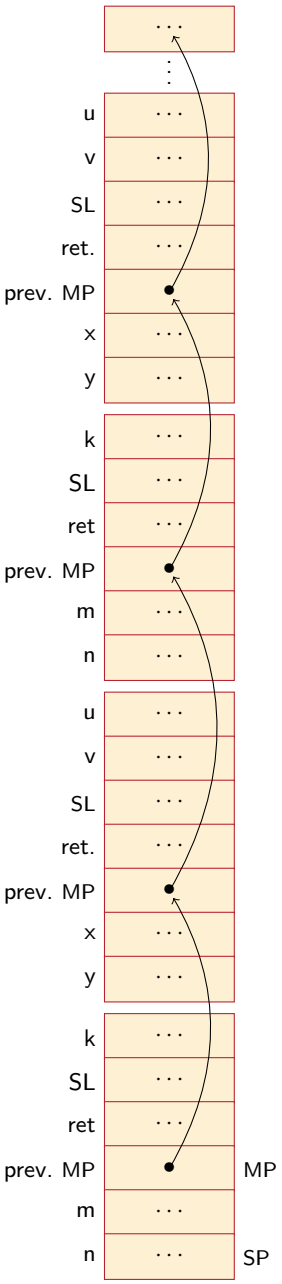


Figure 1: Chain of activation records: dynamic links.

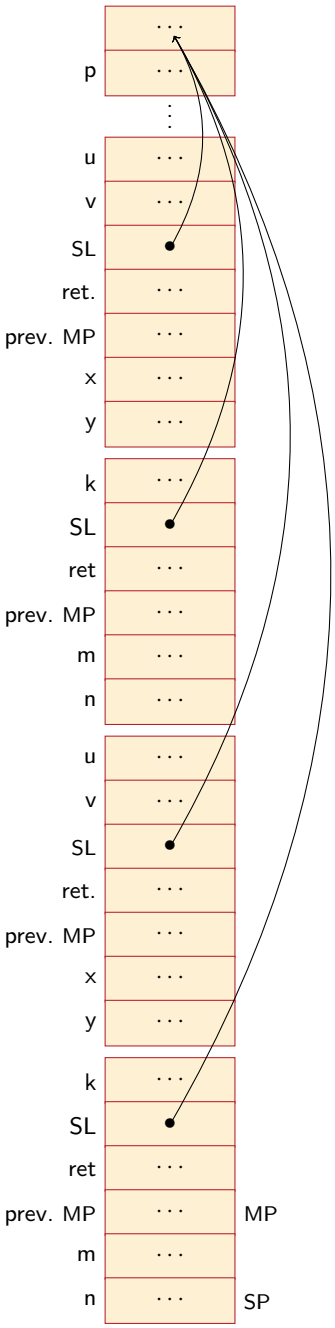


Figure 2: Static links.

The static links for f and g now point to the MP-address relative to which p, f, and g are declared: see Figure 2. Hence, to access the global variable p from within the bodies of f and g, we simply push the address pointed at by the static link and then load from or store into the address at offset 1 relative to this address:

```
ldl  − 2
lda 1.
```

If we have nested function definitions,

```
var p;

function f (u, v)
{   var x;
    var y;
    function g (k)
    {   var m;
        var n;
        ···
        ··· p ···
    }
    ··· g (···) ···
},
```

the static links form a chain (see Figure 3) and accessing a global variable involves traversing this chain. For instance, to access p from the body of g in the example above, we need to follow the static link twice:

```
ldl  − 2
lda  − 2
lda 1.
```

Compiling code for global variables and nested function declarations requires that we make the nesting of declarations explicit in the symbol table.

Interestingly, in order to set up an activation record, the code for a function call needs to perform a similar static-link traversal when pushing the static link for the callee.

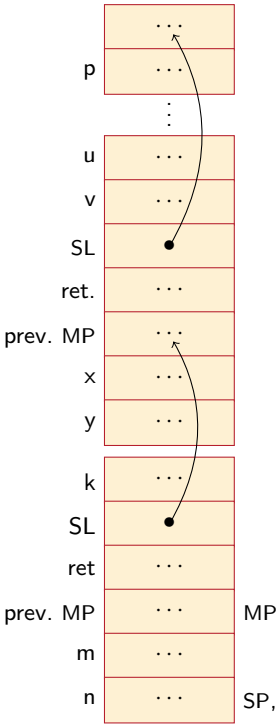


Figure 3: Nested function declarations: chain of static links.