Compiler Construction
Lecture 05
A Simple Stack Machine

Lent Term, 2015

Lecturer: Timothy G. Griffin

Computer Laboratory
University of Cambridge

1

Where are we going?

- When we derived the stack machine from the expression evaluator, we really knew where we were going --- to a simple stack machine with a simple compiler for "reverse Polish" notation. (Well, at least I knew that….)
- Let's pause to think about what the stack machine target of our Slang.1 derivation might look like….
- Today, we will consider only the simple case : simple functions with NO nesting.

Caller and Callee

```
fun f (x, y) = e1

…

fun g(w, v) =
    w + f(v, v)
```

For this invocation of the function f, we say that g is the caller while f is the callee

Recursive functions can play both roles at the same time …

A word about "dynamic binding" --- IT IS A VERY BAD IDEA

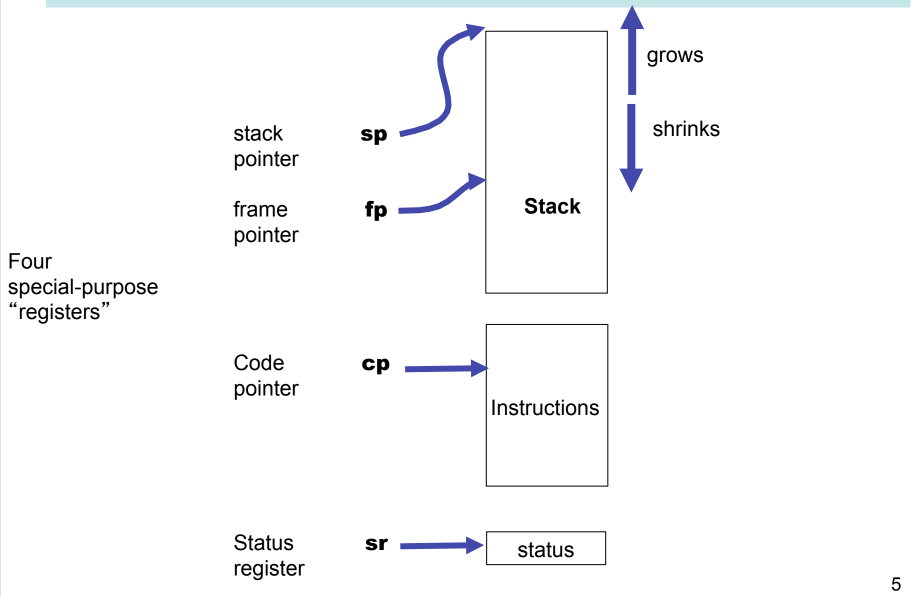
```
let val x = 1
    fun g(y) = x + y
    fun h(x) = g(x) + 1
in
   h(17)
end
```

With good old **static binding** we get 19.

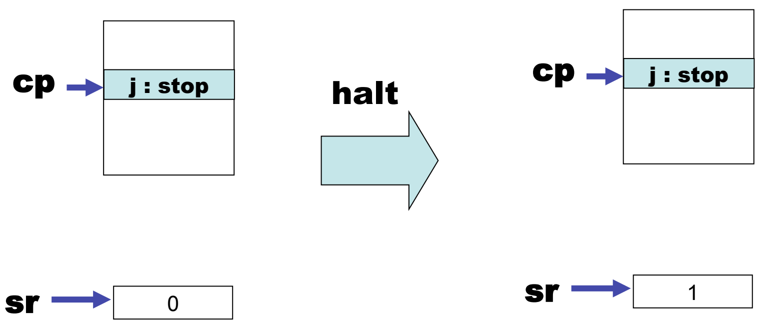
With insane **dynamic binding** we get 35.

But might there be a place for dynamic binding? Is there dynamic binding of some kind behind the raise/handle exception mechanism?

Jargon Virtual Machine

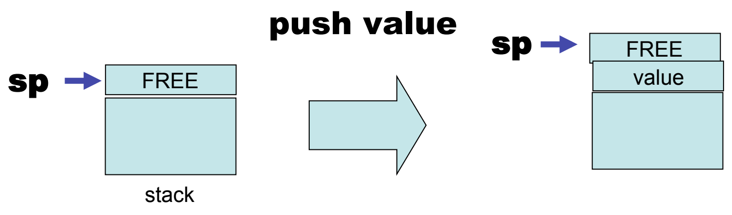


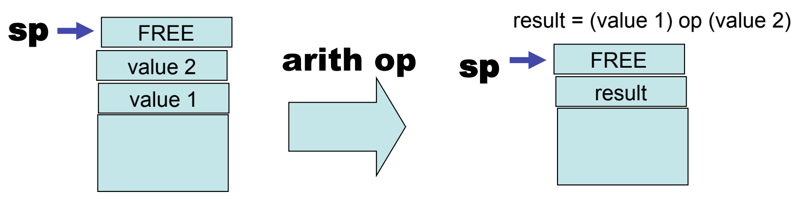
halt



Status codes
- 0 = running
- 1 = stopped (program answer should be on top of stack)
- 2 = store index out of bounds
- 3 = call out of bounds
- 4 = stack overflow
- 5 = fp offset out of bounds
- 6 = return cp out of bounds
- 7 = arith error (div by 0)

Top-of-Stack arithmetic





Op in { + , *, -, / , <, >, <=, >=, =, &&, ||}

7

Translation of expressions

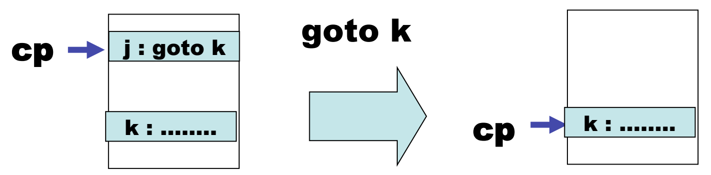
e1 op e2

code for e1
code for e2
arith op

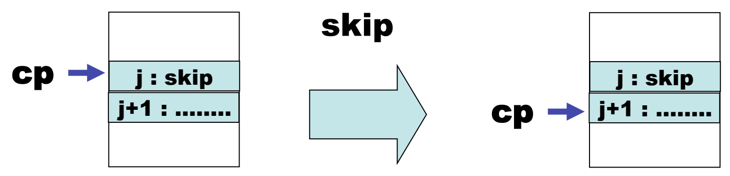
```
3 * ((8 + 17) * (2 - 6))
```

```
0 : push 3
1 : push 8
2 : push 17
3 : arith +
4 : push 2
5 : push 6
6 : arith -
7 : arith *
8 : arith *
```

goto, skip

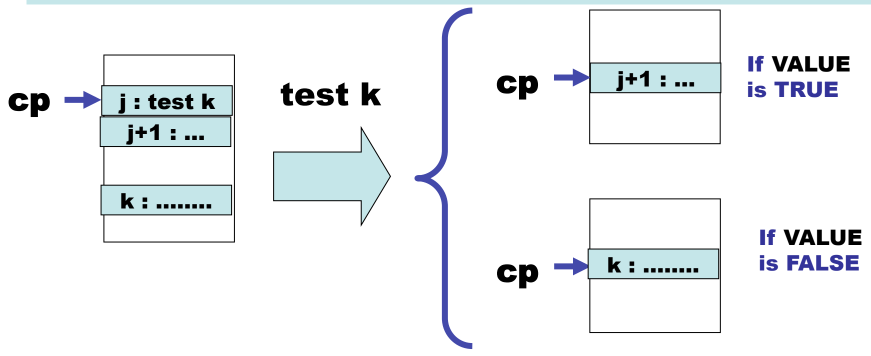


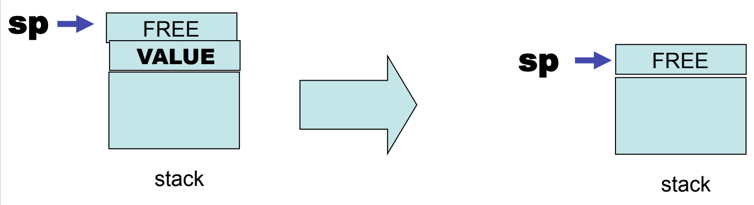
(set status to an error code if k is not in range…)



9

test





10

Conditionals, Loops

If e then c1 else c2

```
code for e
test k
code for c1
goto m
k: code for c2
m:     skip
```

while e { c }

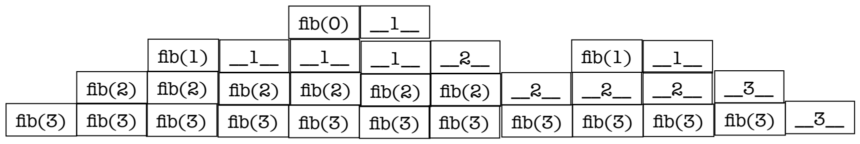
```
m: code for e
test k
code for c
goto m
k:    skip
```

11

How do we organize the call stack?

```
let rec fib m =
   if m = 0
   then 1
   else if m = 1
        then 1
        else fib(m - 1) + fib (m - 2)
```

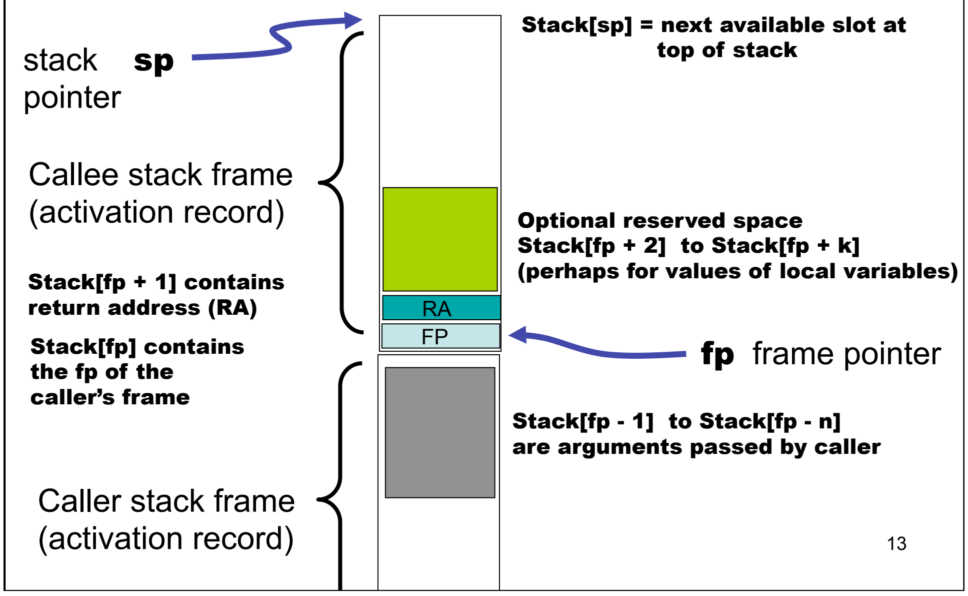
```
List.map fib [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;

  = [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```



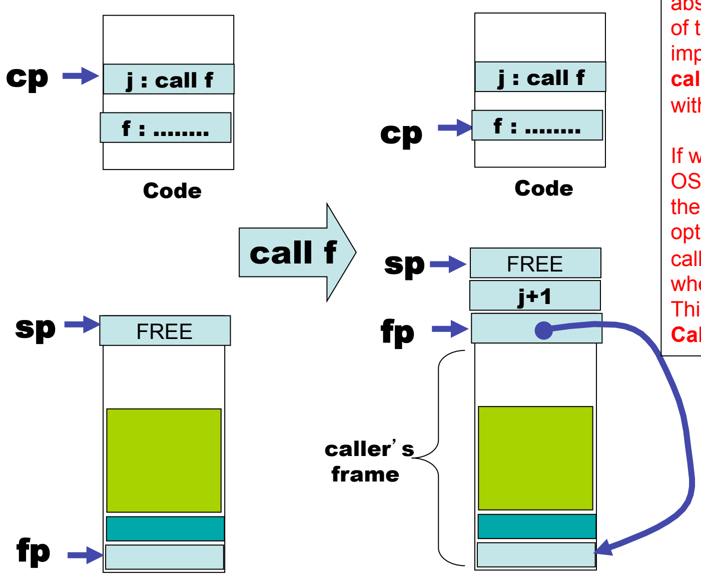
What information does the call stack contain? Does the answer depend on the language implemented? Yes!

12

First : Assume simple functions with NO nesting …


stack pointer
sp
Stack[sp] = next available slot at top of stack
Callee stack frame (activation record)
Optional reserved space Stack[fp + 2] to Stack[fp + k] (perhaps for values of local variables)
Stack[fp + 1] contains return address (RA)
RA
FP
Stack[fp] contains the fp of the caller's frame
fp frame pointer
Stack[fp - 1] to Stack[fp - n] are arguments passed by caller
Caller stack frame (activation record)

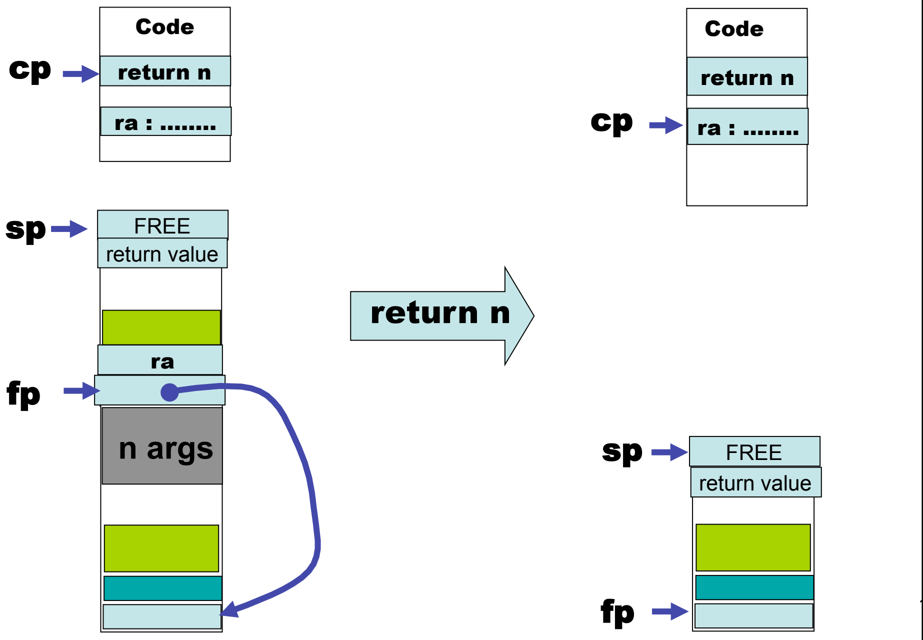
We can now design "high level" VSM commands



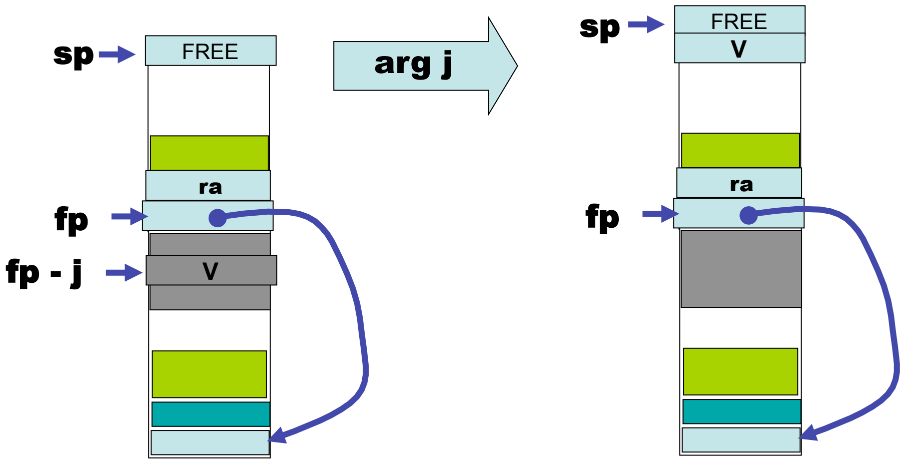
This is a VM-level abstraction. At the level of the program implementing the VM **call** is implemented with many instructions.

If we are targeting an OS/ISA pair (Unix/x86, …), then there are many more options as to who (caller or callee) does what when and where (registers or stack). This is captured in a **Calling Convention**.

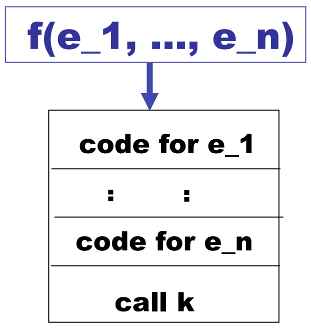
Return



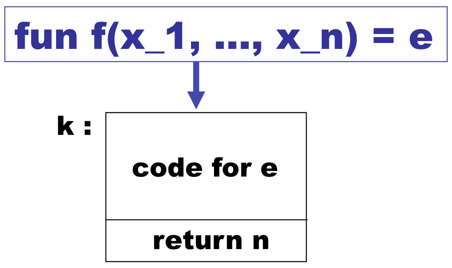
Access to argument values



Translation of (call-by-value) functions



This will leave the values of each arg on the stack, with the value of e_n at the top. Here k is the address for the start of the code for f.

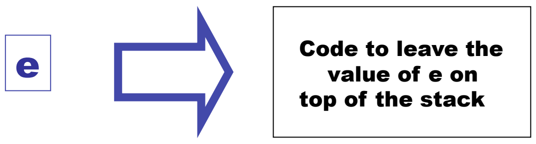


k is a location (address) where code for function f starts.

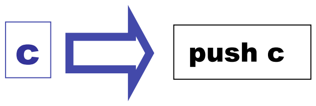
In code for e, access to variable x_i is translated to arg ((n – i) + 1).

17

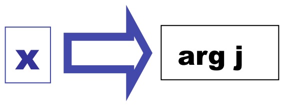
simple expressions



constant

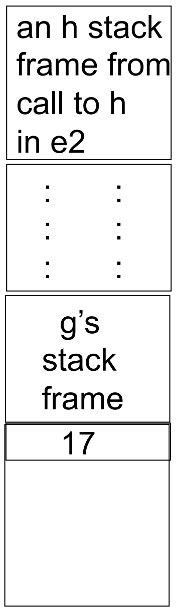


j = (n – i) + 1
where x is the i-th formal parameter (from left to right)



What if we allow nested functions?

```
fun g(x) =
   fun h(y) = e1
   in e2 end
...
g(17)
...
```



How will the code generated from e1 find the value of x?

Approach 1: Lambda Lifting

```
fun g(x) =
   fun h(y) = e1
   in e2 end
...
g(17)
...
```

```
fun h(y, x) = e1

fun g(x) = e3
...
g(17)
...
```

Construct e3 from e2 by replacing each call h(e) with h(e, x)

(+) Keeps our VM simple
(+) Low variable access cost
(-) can duplicate many arg values on the stack