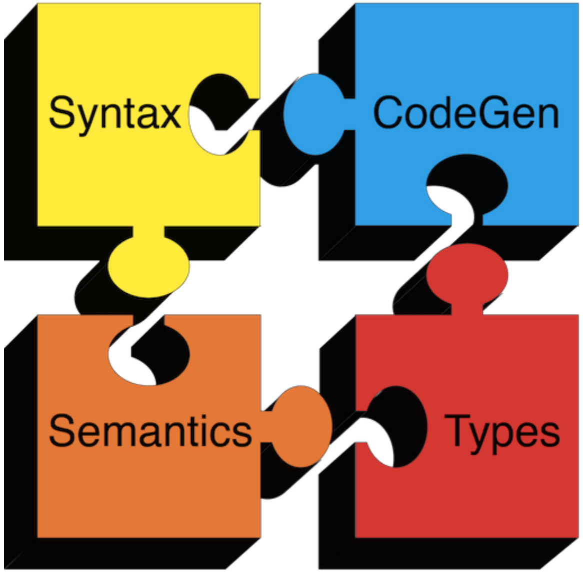


Compilers

Code Generation II

Alex Aiken

A language with integers and integer operations

$$P \to D;\ P \mid D$$

$$D \to \text{def id(ARGS)} = E;$$

$$\text{ARGS} \to \text{id, ARGS} \mid \text{id}$$

$$E \to \text{int} \mid \text{id} \mid \text{if } E_1 = E_2 \text{ then } E_3 \text{ else } E_4$$
$$\mid E_1 + E_2 \mid E_1 - E_2 \mid \text{id}(E_1,\ldots,E_n)$$

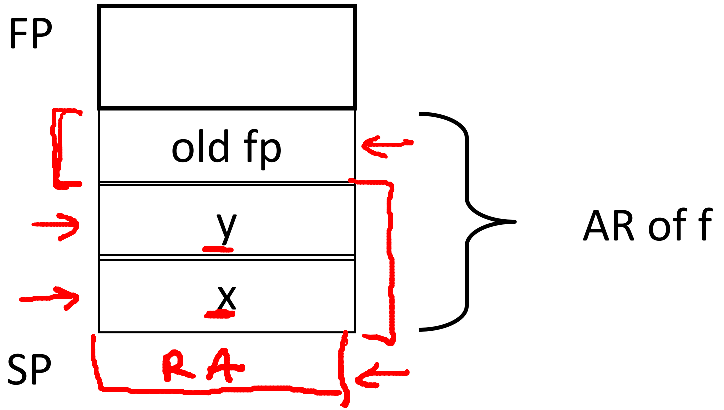
- Code for function calls and function definitions depends on the layout of the AR

- A very simple AR suffices for this language:
  - The result is always in the accumulator
    - No need to store the result in the AR
  - The activation record holds actual parameters
    - For $f(x_1,\ldots,x_n)$ push $x_n,\ldots,x_1$ on the stack
    - These are the only variables in this language

- The stack discipline guarantees that on function exit $sp is the same as it was on function entry
  - No need for a control link
- We need the return address
- A pointer to the current activation is useful
  - This pointer lives in register $fp (frame pointer)

- Summary: For this language, an AR with the caller's frame pointer, the actual parameters, and the return address suffices

- Picture: Consider a call to f(x,y), the AR is:



- The calling sequence is the instructions (of both caller and callee) to set up a function invocation

jal L
add

- New instruction: jal label
  - Jump to label, save address of next instruction in $ra
  - On other architectures the return address is stored on the stack by the "call" instruction

Code Generation II

caller side

$cgen(f(e_1,\ldots,e_n)) =$

```
sw $fp 0($sp)
addiu $sp $sp -4
cgen(e_n)
sw $a0 0($sp)
addiu $sp $sp -4
…
cgen(e_1)
sw $a0 0($sp)
addiu $sp $sp -4
jal f_entry
```

- The caller saves its value of the frame pointer
- Then it saves the actual parameters in reverse order
- Finally the caller saves the return address in register $ra
- The AR so far is 4*n+4 bytes long

- New instruction: jr reg
  - Jump to address in register reg

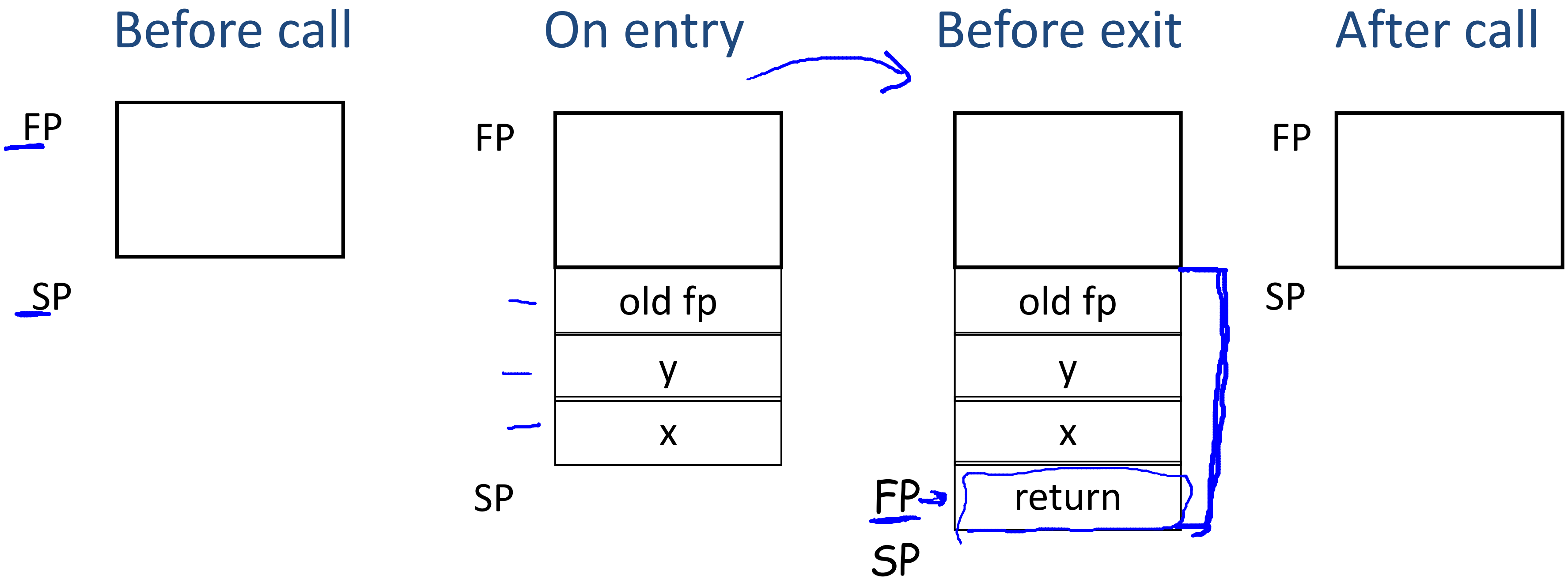
cgen(def f(x_1,…,x_n) = e) =

f_entry:

```
move $fp $sp
sw $ra 0($sp)
addiu $sp $sp -4
cgen(e)
lw $ra 4($sp)
addiu $sp $sp z
lw $fp 0($sp)
jr $ra
```

- Note: The frame pointer points to the top, not bottom of the frame
- The callee pops the return address, the actual arguments and the saved value of the frame pointer
- z = 4*n + 8

return add.
old fp


Before call
FP
SP
On entry
FP
old fp
y
x
SP
Before exit
old fp
y
x
FP
return
SP
After call
FP
SP

- Variable references are the last construct

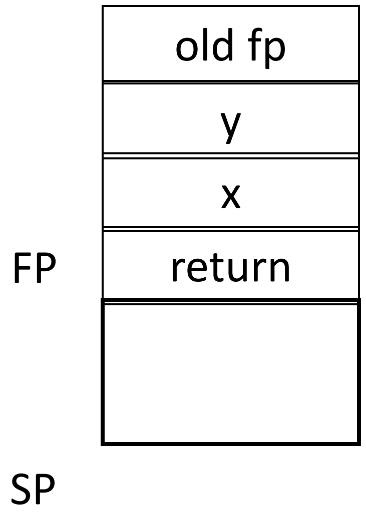
- The "variables" of a function are just its parameters
  - They are all in the AR
  - Pushed by the caller

- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from $sp

- Solution: use a frame pointer
  - Always points to the return address on the stack
  - Since it does not move it can be used to find the variables
- Let x_i be the i^{th} ($i = 1,\dots,n$) formal parameter of the function for which code is being generated

$$\text{cgen}(x_i) = \text{lw \$a0 z(\$fp)} \qquad (\, z = 4*i \,)$$

- Example: For a function def f(x,y) = e the activation and frame pointer are set up as follows:



- X is at fp + 4
- Y is at fp + 8

Code Generation II

For the function definitions at right, which of the following appear in the activation record on a call to f()?

- ☐ x
- ☐ t
- ☐ g
- ☐ z

```
def f(x,y,z) =
    if x
    then g(y)
    else g(z)

def g(t) =
    t + 1
```

- The activation record must be designed together with the code generator

- Code generation can be done by recursive traversal of the AST

- We recommend you use a stack machine for your Cool compiler (it’s simple)

- Production compilers do different things
  - Emphasis is on keeping values in registers
    - Especially the current stack frame
  - Intermediate results are laid out in the AR, not pushed and popped from the stack