Lecture Outline

- Code Generation for a Stack Machine
- a simple language
- activation trees again
- a simple implementation model: the stack machine
- stack machine implementation of the simple language
  - design of activation records
  - code generation

Note: these lecture notes are by Alex Aiken for his compiler class at UC Berkeley with minor modifications made for local use.

A Small Language

- A language with integers and integer operations:

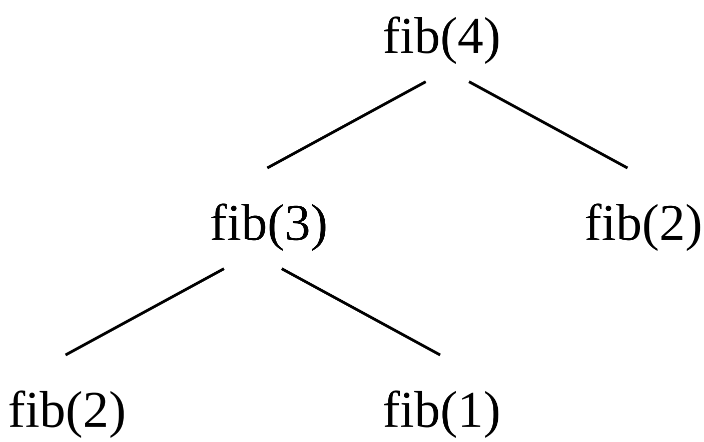
$$
\begin{aligned}
\mathtt{P} &\rightarrow \mathtt{D};\mathtt{P} \,|\, \mathtt{D} \\
\mathtt{D} &\rightarrow \mathtt{def\ id}(\mathtt{ARGS}) = \mathtt{E}; \\
\mathtt{ARGS} &\rightarrow \mathtt{id},\ \mathtt{ARGS} \,|\, \mathtt{id} \\
\mathtt{E} &\rightarrow \mathtt{int} \,|\, \mathtt{id} \,|\, \mathtt{if}\ \mathtt{E}_1 = \mathtt{E}_2\ \mathtt{then}\ \mathtt{E}_3\ \mathtt{else}\ \mathtt{E}_4 \,| \\
&\quad\ \mathtt{E}_1 + \mathtt{E}_2 \,|\, \mathtt{E}_1 - \mathtt{E}_2 \,|\, \mathtt{id}(\mathtt{E}_1, \ldots, \mathtt{E}_\mathtt{n})
\end{aligned}
$$

- The first function definition f is the "main" routine.
- Running the program on input i means compute $f(i)$.
- Computing the ith Fibonacci number:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
                      fib(x-1) + fib(x-2)
```

Review: Activation Trees

- The activation tree for a run of a program is a graph of the function calls.
- For `fib(4)`, the activation tree is:



- Activation records are managed using a runtime stack.
- At any point during execution, the activation stack describes some path starting from the root of the activation tree.

A Stack Machine

- A stack machine evaluates one expression at a time.
- The value of an expression is stored in a distinguished register called the *accumulator* (or *acc*).
- A stack is used to hold intermediate results.
- To evaluate an expression $\mathrm{op}(\mathrm{e}_1, \ldots, \mathrm{e}_\mathrm{n})$:
  1. Evaluate $\mathrm{e}_1, \ldots, \mathrm{e}_\mathrm{n}$, pushing results on the stack.
  2. Set $\mathtt{acc} = \mathrm{op}(\mathrm{e}_1, \ldots, \mathrm{e}_\mathrm{n})$ using values on the stack.
  3. Pop values of $\mathrm{e}_1, \ldots, \mathrm{e}_\mathrm{n}$ off of the stack.
- Note: All expressions evaluate into `acc`; all expressions expect to find the values for other expressions in `acc`.

Example

- Consider the expression e1 + e2.

- At a high level, the stack machine code will be:

```
<code to evaluate e1>
push acc on the stack
<code to evaluate e2>
push acc on the stack
add top two stack elements, store in acc
pop two elements off the stack
```

- Observation: There is no need to push the result of e2 on the stack.

```
<code to evaluate e1>
push acc on the stack
<code to evaluate e2>
add top stack element and acc, store in acc
pop one element off the stack
```

Notes

- The code for $+$ is a template with "holes" for code for e1 and e2.

- Stack machine code generation is recursive.

- Code for `e1 + e2` consists of code for e1 and e2 glued together.

- Code generation—at least for expressions—can be written as a recursive-descent of the AST.

A Bigger Example

- Consider (1 + 2) + 3.
- Let sp be the stack pointer (held in a register).
- Code for an integer i is acc ← i.
- (sp) is the value stored at address sp.
- Pseudo-code for the expression:

```
acc <- 1                | 1's code |      |
push acc (onto stack)              | code |
acc <- 2                | 2's code | for  |
acc <- acc + (sp)                  | 1+2  | code
pop (the stack)                    |      | for
push acc                                  | (1+2)
acc <- 3                | 3's code        | +3
acc <- acc + (sp)                         |
pop                                       |
```

MIPS Assembly

- Next Step: Switch to MIPS assembly language.
- A sample of MIPS instructions:
  - `sw reg1 offset(reg2)`
    store word in `reg1` at `reg2 + offset`
    (contents of `reg2` used as an address)
  - `lw reg1 offset(reg2)`
    load word from `reg2 + offset` into `reg1`
  - `add reg1 reg2 reg3`
    `reg1 := reg2 + reg3`
  - `addiu reg1 reg2 imm`
    add immediate (i.e., constant),
    `u` means overflow not checked
- `$a0` is the accumulator (a MIPS register)
- `$sp` is the stack pointer
  - `$sp` points to the first word beyond stack top
  - on the MIPS, stack grows towards low addresses.
    (we'll show it as growing downward)

MIPS Code Generation for Add

- Define a function `cgen(e)` for each expression `e`.

```
cgen(e1 + e2) =
    cgen(e1)
    sw    $a0 0($sp)  | push the acc on the
    addiu $sp $sp -4  | stack
    cgen(e2)
    lw    $t1 4($sp)  | load result of e1
    add   $a0 $a0 $t1
    addiu $sp $sp 4   | pop the stack
```

- MIPS addresses bytes; to move a pointer by one word, add 4 bytes.

- Question: Why not put `e1` in register `$t1` immediately, instead of pushing it on the stack?

Code Generation for Sub and Constants

- New instruction: `sub reg1 reg2 reg3`
  `reg1 = reg2 - reg3`

```
cgen(e1 - e2) =
     cgen(e1)
     sw    $a0 0($sp)  | push acc on the
     addiu $sp $sp -4  | stack
     cgen(e2)
     lw    $t1 4($sp)  | load result of e1
     sub   $a0 $t1 $a0
     addiu $sp $sp 4   | pop the stack
```

- New instruction: `li reg1 imm`
  Load immediate, `reg1 := imm`

- Code generation for constant `i`:

```
cgen(i) = li $a0 i
```

Code Generation for If

- New instruction: `beq reg1 reg2 label`
  jump to `label` if `reg1 = reg2`

- New instruction: `b label`
  jump to `label`

- Code for conditionals:

```
cgen(if e1 = e2 then e3 else e4) =
      cgen(e1)
      sw    $a0 0($sp) | push e1 on stack
      addiu $sp $sp -4
      cgen(e2)
      lw    $t1 4($sp) | pop stack into t1
      addiu $sp $sp 4
      beq   $a0 $t1 true_branch
false_branch:
      cgen(e4)
      b     end_if
true_branch:
      cgen(e3)
end_if:
```

The Activation Record

- Code for function calls and function definitions depends on the activation record.

- 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 $\mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_\mathtt{n})$, push $x_n, \ldots, x_1$ on the stack.
  - 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.
  - It's handy to have a pointer to the current activation. This pointer lives in register `$fp` (for "frame pointer").

- Reason for frame pointer will be clear shortly ...

The Activation Record (Cont.)

- 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 will be:

- Using the AR, we can describe code generation for function definitions and function calls.

Code Generation for Function Call

- The *calling sequence* is the instructions—of both caller and callee—to set up a function invocation.
- New instruction: `jal label`
  Jump to `label`, save address of next op in `$ra`.

```
cgen(f(e1,...,en)) =
 sw    $fp 0($sp)  | push frame pointer
 addiu $sp $sp -4
 cgen(en)          | evaluate and
 sw    $a0 0($sp)  | push actual parameter #n
 addiu $sp $sp -4  |
 ...
 cgen(e1)          | evaluate and
 sw    $a0 0($sp)  | push actual parameter #1
 addiu $sp $sp -4  |
 jal   f_entry     | jump to function entry
```

- The caller saves the actual parameters in the AR.
- The callee must save the return address.

Code Generation for Function Definition

- New instruction: `jr reg`
  jump to address in register `reg`

```
cgen(def f(x1,...,xn) = e) =
f_entry:
     move  $fp $sp    | set new fp
     sw    $ra 0($sp)  push return address
     addiu $sp $sp -4 |
     cgen(e)
     lw    $ra 4($sp) | reload return address
     addiu $sp $sp z  | z = 4*n + 8 (pop AR)
     lw    $fp 0($sp) | restore old fp
     jr    $ra        | return
```

- Note the frame pointer points to the top, not bottom, of the frame.

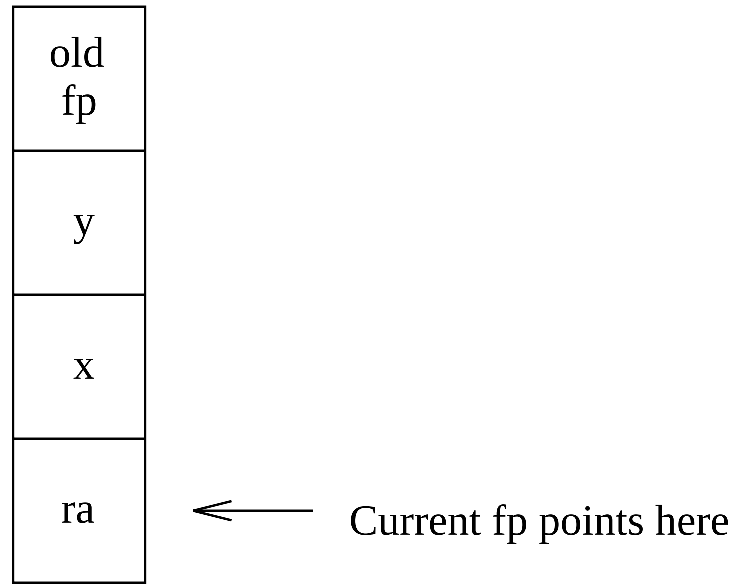
Code Generation for Variables

- Variable references are the last construct.
- The "variables" of a function are just its parameters, which are in the AR.
- Problem: Because the stack grows when intermediate results are saved, the variables are not at a fixed offset from `$sp`.
- Solution: Use the frame pointer.
- Let `xi` be the ith formal parameter of the function for which code is being generated.

```
cgen(xi) = lw  $a0 z($fp)     | z = 4*i
```

Code Generation for Variables (Cont.)

- 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`

Summary and Warnings

- 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 Espresso compiler (it's simple!).
- Production compilers do things differently:
  - Emphasis is on keeping values (esp. current stack frame) in registers.
  - Intermediate results are laid out in the AR, not pushed and popped from the stack.
- See the Web page for a large code generation example.