Code Generation 1

A higher-level language

```
P --> D; P | D

D --> def id(ARGS) = E
ARGS --> id, ARGS | id
E --> int | id | if E1 = E2 then E3 else E4 | E1 + E2 | E1 - E2 | id(E1,...,En)
```

The first function definition `f` is the entry point

- The "main" routine

Program for computing the Fibonacci numbers:

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

For each expression `e` we generate MIPS code that:

- Computes the value of `e` in `$a0`
- Preserves `$sp` and the contents of the stack

We define a code-generation function `cgen(e)` whose result is the code generated for `e`

- The code to evaluate a constant simply copies it into the accumulator:

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

  - This preserves the stack, as required
  - Color key:
    - RED: compile time
    - BLUE: run time

- ```
  cgen(e1+e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp-4
  cgen(e2)
  lw $t1 r($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  ```

  - Establish the two invariants: the accumulator has the value of this expression, and the value of the stack is unchanged.

  Another (more precise) way to write this:

  ```
  cgen(e1+e2) =
  cgen(e1)
  print "sw $a0 0($sp)"
  print "addiu $sp $sp-4"
  cgen(e2)
  print "lw $t1 r($sp)"
  print "add $a0 $t1 $a0"
  print "addiu $sp $sp 4"
  ```

  This latter syntax is a bit more verbose, so we will largely stick to the short-hand notation.
- "Optimization": Put the result of `e1` directly in `$t1`

  ```
  cgen(e1+e2) =
  cgen(e1)
  move $t1 $a0
  cgen(e2)
  add $a0 $t1 $a0
  ```

  Is this code correct? Why not? Consider if `e2=e3+e4`
- The code for + is a template with "holes" for code for evaluating `e1` and `e2`.
- Stack machine code generation is recursive
  - Code for `e1+e2` is code for `e1` and `e2` glued together
- Code generation can be written as a recursive descent of the AST
  - At least for expressions

- ```
  cgen(e1-e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp-4
  cgen(e2)
  lw $t1 r($sp)
  sub $a0 $t1 $a0
  addiu $sp $sp 4
  ```

Dealing with if-then-else constructs:

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

- ```
  cgen(if e1=e2 then e3 else e4) =
  cgen(e1)
  ```
  ```

```
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
addiu $sp $sp 4
beq $a0 $t1 true_branch
...
false_branch:
cgen(e4)
b end_if
true_branch:
cgen(e3)
end_if:
```

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 the actual parameters
  - For `f(x1,...,xn)` push `x1,...,xn` on the stack
  - These are the only variables in this language
- The stack discipline guarantees that on function exit `$sp` is the same as it was on function entry
  - No need for a control link (which is usually needed to find another activation).
- We need the return address
- A pointer to the current activation is useful
  - This pointer lives in the register `$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:

```
FP:
      old fp  |
      y       } AR of f
      x       |
SP-> RA
```

- 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 instruction in `$ra`
    - Show an example of what the "address of next instruction" means
  - On x86, the return address is stored on the stack by the "call" instruction
- New instruction: `jr reg`
  - Jump to the address in register reg
- Code for the "caller side":

```
  cgen(f(e1,e2,...,en)) =
```

```
sw $fp 0($sp)
addiu $sp $sp -4
cgen(en)
sw $a0 0($sp)
addiu $sp $sp - 4
...
cgen(e1)
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+8` bytes long
- Code for the "callee side":

```
  cgen(def f(x1,x2,...,xn) = 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
    - This is just one example strategy. Another strategy could be that the caller pops these elements off the stack, and restores the frame pointer. Both are valid strategies.
  - *z* = 4*n+8 (size of the activation record)
- Before call:

```
FP -->

SP -->
```

- On entry:

```
FP -->

     old fp
     y
     x
SP--> (this is where the return address will go)
```

- Before exit:

```
FP -->

     old fp
     y
     x
FP--> ra
SP-->
```

- After exit:

```
FP -->

SP -->
```

- Function calls have preserved the invariant that the stack would be exactly the same after the call, as it was at entry to the call

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`
- One 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 `xi` be the ith (i=1..n) formal parameter of the function for which code is being generated

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

  e.g., `x` at `$fp+4` and `y` at `$fp+8`
- Another solution: keep track of the current `$sp` vis-a-vis the value of `$sp` at function entry, and thus appropriately adjust the addresses of all variables. Pro: one less register (no frame pointer needed now)! Con: much harder to read and debug the code now. Also not possible with VLAs (variable length arrays).

Summary

- The activation record must be designed together with the code generator
  - Can be a *convention* (allows compatibility) or can be decided at compile-time on a function-to-function basis, but any decision should account for all possible callers/callees for any callee/caller.
- Code generation can be done by recursive traversal of the AST
- While we discussed code generation for a stack machine, production compilers often do different things
  - Emphasis is on keeping values in registers (optimization)
    - Especially the current stack frame
  - Intermediate results are laid out in the AR (at fixed offsets for direct access). Not pushed and popped from the stack (can be more expensive as it does not allow value re-use)

Example

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

- ```
  sumto_entry:
  move $fp $sp
  sw $ra 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 0
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true1

  false1:
  lw $a0 4($fp)
  sw $a0 0($sp)
  addiu $sp $sp -4
  sw $fp 0($sp)
  addiu $sp $sp -4
  lw $a0 4($fp)              # x
  sw $a0 0($sp)
  addiu $sp $sp -4
  li $a0 1
  lw $t1 4($sp)
  sub $a0 $t1 $a0           # x - 1
  addiu $sp $sp 4
  sw $a0 0($sp)
  addiu $sp $sp -4
  jal sumto_entry
  lw $t1 4($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  b endif1

  true1:
  li $a0 0

  endif1:
  lw $r1 4(esp)
  addiu $sp $sp 12
  lw $fp 0($sp)
  jr $ra
  ```

  - Quite inefficient: pushing and popping on the stack all the time (also called *stack traffic*)
  - Will discuss more efficient code generation techniques and optimizations later
  ```

Temporaries

Let's discuss a better way for compilers to manage temporary values. Idea: keep temporaries in the AR. The code generator must assign a fixed location in the AR for each temporary. (We will later discuss keeping temporaries in registers, which is even more efficient).

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

If we first analyze the code to see how many temporaries we need, we can allocate space for that, instead of pushing and popping it from the stack each time.

- One temporary for expression `x=1`
- One temporary for expression `x=2`
- One temporary each for the two recursive calls to `fib`
- One temporary for the sum of the two values returned by the recursive calls
- One temporary for `x-1`
- One temporary for `x-2`

But many of these temporaries are only needed once and not needed after that. So some of the space can be re-used for the subsequent temporaries. In fact, we can evaluate the whole expression using only two temporaries.

Let `NT(e)` be the number of temps needed to evaluate `e`

```
NT(e1+e2)
```

- Needs at least as many temporaries as `NT(e1)`
- Needs at least as many temporaries as `NT(e2) + 1`
  - Need one extra temporary to hold on to value of `e1`
- Space used for temporaries in `e1` can be reused for temporaries in `e2`
- `NT(e1+e2) = max(NT(e1), 1+NT(e2))`

Generalizing, we get a system of equations:

```
NT(e1+e2)       = max(NT(e1), 1+NT(e2))
NT(e1-e2)       = max(NT(e1), 1+NT(e2))
NT(if e1=e2 then e3 else e4) = max(NT(e1),1+NT(e2), NT(e3), NT(e4)) #once the branch is decided we do not need to hang on to e1/e2
NT(id(e1,...,en)) = max(NT(e1),...,NT(en))  #the space for the result for ei is saved in the new activation record that we are building a
NT(int) = 0
NT(id) = 0
```

Looking at our fib example

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

Show that the entire expression evaluates to NT=2

Once we know the number of temporaries required, we can add that much space for the AR

- For a function definition f(x1...xn) = e the AR has 2 + n + NT(e) elements
  - The return address
  - The frame pointer
  - n arguments
  - NT(e) locations for intermediate results

AR layout

```
Old FP
xn
..
x1
RA
Temp NT(e)
Temp NT(e) - 1
..
Temp 1
```

Code generation must know how many temporaries are in use at each point

- Add a new argument to code generation
  - the position of the next available temporary
- The temporary area is used like a small, fixed-size stack
  - Instead of pushing and popping at run time, we simply do all that in the compiler (at compile time)
- Previous code

  ```
  cgen(e1+e2) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  add $a0 $t1 $a0
  addiu $sp $sp 4
  ```

- New code (avoid stack manipulation instructions)

  ```
  cgen(e1+e2, nt) =
  cgen(e1, nt)
  sw $a0 nt($sp)
  cgen(e2, nt+4)
  lw $t1 nt($fp)
  add $a0 $t1 $a0
  ```