Chapter 1

Code generation

1.1 Stack machines

A stack machine uses only a stack as storage. When executing an instruction $\mathtt{r} = \mathtt{F}(\mathtt{a_1}, \ldots, \mathtt{a_n})$, it pops n operands from the stack, computes the operation F using the operands, and pushes the result r back on the stack. For example, when computing 7 + 5, the stack will change from s-7-5 to s-12.

Consider two instructions: push i (push integer i on the stack) and add (add two integers). Then we have the program to computer 7 + 5:

push 7
push 5
add

An important property of stack machines is that location of the operands/result is not explicitly stated because they are always at the top of the stack. This is different from register machine in which the locations have to be specified. We have `add` instead of $\mathtt{add}\ \mathtt{r_1}, \mathtt{r_2}, \mathtt{r_3}$, which produces more compact programs. This is one of the reasons why JAVA bytecodes uses stack evaluation.

Stack machine produces compact programs, but register machine executes faster. There is an intermediate point between the two kinds of machines called an **n-register stack machine**. As the name reveals, the top n positions of the pure stack machine's stack are held in registers. It turns out that even one single register can provide considerable performance improvement, which is the case of **1-register stack machine**. The register is called the **accumulator**.

In a pure stack machine, an add does 3 memory operations: two reads and one write. But in a 1-register stack machine, what add does is acc←acc + top_of_stack. In general, consider an arbitrary expression $\mathtt{op}(\mathtt{e_1}, \ldots, \mathtt{e_n})$. For each subexpression $e_i (1 \leq i \leq n-1)$, we will compute e_i, store the result in acc and then push the result on the stack. For e_n, we will have its result remain in acc. Then we will pop n-1 values from the stack to compute op, and store the result in acc. If we follow this procedure, obviously after evaluating an

expression e, acc holds the value of e, and the stack is unchanged. In other words, **expression evaluation preserves the stack.**

Consider the calculation of 3 + (7 + 5). We will have the following process:

Code	Acc	Stack
acc←3	3	<init>
push acc	3	3,<init>
acc←7	7	3,<init>
push acc	7	7,3,<init>
acc←5	5	7,3,<init>
acc←acc + top_of_stack	12	7,3,<init>
pop	12	3,<init>
acc←acc + top_of_stack	15	3,<init>
pop	15	<init>

1.2 Basic MIPS instructions

In our discussion of code generation, we will focus on generating code for a stack machine with accumulator. The resulting code should be able to run on an MIPS processor (or simulator). Thus we have to simulate stack machine instructions using MIPS instructions and registers.

We choose to keep the accumulator in MIPS register $a0. The stack is kept in memory and grows towards lower addresses, which is a standard convention in MIPS. The address of the next location on the stack is kept in MIPS register $sp (which stands for stack pointer), and the top of the stack is at address $sp + 4.

MIPS is an old structure with a relatively simple instruction set (prototypical reduced instruction set computer, or RISC). Most MIPS operations use registers for operands and results. Load and store instructions are used to move values to and from memory. There are 32 general purpose registers (32 bits each) in MIPS, and we will use only $a0, $sp and $t1 (temp register used for operations that take two arguments).

Here are the first set of MIPS instructions that we introduce.

lw reg1 offset(reg2) : Load 32-bit word from address `reg2 + offset` into `reg1`.

sw reg1 offset(reg2) : Store 32-bit word in `reg1` at address `reg2 + offset`.

add reg1 reg2 reg3 : `reg1` ← `reg2 + reg3`

addiu reg1 reg2 imm : `reg1` ← `reg2 + imm`. `u` means that overflow is not checked.

li reg imm : `reg` ← `imm`.

move reg1 reg2 reg1 ← reg2.

The stack machine code for 7 + 5 is:

```
acc ← 7                      li $a0 7
push acc                     sw $a0 0($sp)
                             addiu $sp $sp -4
acc ← 5                      li $a0 5
acc ← acc + top_of_stack     lw $t1 4($sp)
                             add $a0 $a0 $t1
pop                          addiu $sp $sp 4
```

1.3 Code generation for a simple language

In this section we will take a look at code generation for higher level languages rather than a simple stack machine in the previous section.

Consider a language for integer operations. Its grammar depicts a list of function definitions:

$P \rightarrow D;\ P \mid D$

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

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

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

The first function definition `f` is the entry point, i.e. the `main` routine. This language is enough to write a program that computes the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
             fib(x - 1) + fib(x - 2)
```
(1.1)

For each expression `e`, we want to generate MIPS code that **computers the value of `e` in $a0** and **preserves $sp and the content of the stack**. We will define a function `cgen(e)` whose return value if the code generated for `e`.

1.3.1 Constants

For constants, we need to simply load it into the accumulator:

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

1.3.2 Addition

For addition:

```
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
```

Here we use different colors to emphasize the fact that MIPS code is **generated** at compile time and **executed** at run time. Code in red color is what happens at compile time: the generation, while code in black is what happens at run time: the execution. More precisely, we should write `sw $a0 0($sp)` as something like `print sw $a0 0($sp)`, which indicates that the MIPS code is generated at compile time and saved somewhere, maybe in an intermediate file, and does not get executed until run time.

It seems that the piece of code above could be optimized: why don't we save the value of e1 directly in $t1, rather than saving it in the memory and then retrieve it into $t1? The code will look like:

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

Unfortunately, this neat piece of code is wrong. We can convince ourselves by simply considering the code generated for 1 + (2 + 3). In short, the value of $t1 will be modified during `cgen(e2)`, and is no longer the value of e1 when the execution comes to `add $a0 $t1 $a0`.

The simple example of code generation for addition demonstrates a few universal properties of code generation. The code generated for `e1 + e2` is a template with "holes" for code generated to evaluate e1 and e2. Stack machine code generation is actually recursive. The code generation process can be written as a recursive descent of the AST, at least for expressions.

1.3.3 Subtraction

By introducing another MIPS instruction sub:

sub reg1 reg2 reg3 reg1←reg2 - reg3

we can generate the code for `e1 - e2`:

```
cgen(e1 - e2) =
        cgen(e1)
```

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

1.3.4 If-then-else

In order to generate code for `if-then-else` expressions, we need to introduce a couple of new instructions:

beq reg1 reg2 label branch to label if reg1 = reg2

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 $t1 $a0 true_branch
    false_branch:
      cgen(e4)
      b end_if
    true_branch:
      cgen(e3)
    end_if:
```

1.3.5 Function calls & definitions, variable references

Code for function calls and function definitions depends on the layout of the activation record, thus they need to be designed together. For this simple language, a very simple AR is sufficient.

- Since the result is always in the accumulator, there is no need to store the result in AR.
- The AR should hold the actual parameters, thus for $f(x_1, \ldots, x_n)$, we need to push $x_n, \ldots, x_1$ on the stack. These are the only variables in this language.

- The stack discipline guarantees that $sp is preserved after a function call. Thus there is no need for a control link: the previous activation can be found directly; and we do not need to look at another activation during the function call because there is no non-local variable.
- We need the return address.
- A pointer to the **current** activation is useful. It lives in register $fp (frame pointer).

To summarize, for this simple language, an AR will contain the caller's frame pointer, the actual parameters and the return address. The caller's frame pointer should be contained because $fp will be probably overwritten during the execution of the function (by other functions called during the execution). Consider a call to $f(x, y)$. Before its body gets executed, its AR looks like Figure 1.1. The **calling sequence** is the instructions (of both the caller and the

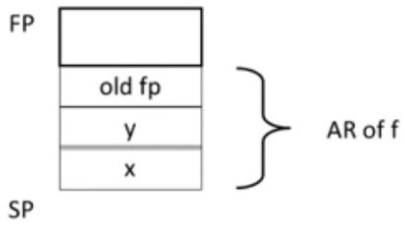

Figure 1.1: AR of f(x,y)

callee) to set up a function invocation. Here we introduce a new instruction

jal label Jump to label, and save the address of the next instruction in $ra (meaning return address). **jal** means jump and link.

Now we can actually generate the code for function call expressions.

```
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
```

For `f(e1,e2,...,en)`, we first save the frame pointer of the caller, then save the arguments one by one (from `en` to `e1`). Up to now we've completed the calling sequence on the caller side. Next we can use `jal` to jump to the entry point of function `f`. The return address is now in $ra. The AR so far is `4*n+4` bytes long.

We introduce another instruction

jr reg Jump to the address in register reg.

Now we can discuss the callee side of the calling sequence.

```
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 4n + 8
      lw $fp 0($sp)
      jr $ra
```

First we set up the frame pointer by saving the current stack pointer into $fp. Then the return address is saved in memory. Now we can generate code for the function body. The stack pointer will be reserved, thus we can load the return address back into $ra. Next we can pop the return address, all arguments and the old fp out of the stack (totally 4*n+8 bytes). We restore the value of the old fp, and finally jump back to the return address.

Variables in this language are just the function arguments. They are all pushed into the AR by the caller. But since the stack grows when intermediate results are saved, the variables are not at fixed offsets from $sp. That's when $fp should be used. The offset of `xi` from $fp is 4*i. Thus we can generate the code for variable reference.

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

1.3.6 Summary

To summarize, the AR must be designed together with the code generator. Code generation can be completed by a recursive traversal of the AST. Such an approach using a stack machine is a wise choice to implement the COOL code generator.

Production compilers are for sure different from the example here. They emphasize keeping values in registers, especially the current stack frame, for the

code to run faster. Also, intermediate results are laid out in the AR rather than pushed and popped from the stack.

As an example, let's generate the code for the following program:

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

```
sumto_entry:
  move $fp $sp          //set up frame pointer
  sw $ra $sp            //return address
  addiu $sp $sp -4
  lw $a0 4($fp)         //start generation for if-then-else. load x from AR.
  sw $a0 0($sp)         //save value of x (1st arg of comparison)
  addiu $sp $sp -4
  li $a0 0              //immediately load 0
  lw $t1 4($sp)         //load x (1st arg of comparison) into $t1
  addiu $sp $sp 4       //pop x
  beq $t1 $a0 true1     //compare and branch
false1:
  lw $a0 4($fp)         //load x from AR
  sw $a0 0($sp)         //save value of x (1st arg of addition)
  addiu $sp $sp -4
  sw $fp 0($sp)         //save frame pointer of caller
  addiu $sp $sp -4
  lw $a0 4($fp)         //load x from AR
  sw $a0 0($sp)         //save value of x (1st arg of subtraction)
  addiu $sp $sp -4
  li $a0 1              //immediately load 1
  lw $t1 4($sp)         //load x (1st arg of subtraction into $t1)
  sub $a0 $t1 $a0       //subtraction
  addiu $sp $sp 4       //pop x
  sw $a0 0($sp)         //save value of x-1 (arg of sumto(x-1))
  addiu $sp $sp -4
  jal sumto_entry       //jump and link
  lw $t1 4($sp)         //load x (1st arf of addition) into $t1
  add $a0 $t1 $a0       //addition (x+sumto(x-1))
  addiu $sp $sp 4       //pop x. up to now finished x+sumto(x-1)
  b end_if1
true1:
```

```
  li $a0 0            //immediately load 0
end_if1               //up to now finished if-then-else
  lw $ra 4($sp)       //load return address
  addiu $sp $sp 12    //pop ra, argument(x) and old fp
  lw $fp 0($sp)       //restore old fp
  jr $ra
```

1.4 Temporaries

One of the advantages of production compilers over the simple one we introduced is that temporaries are kept in the AR. In order to generate more efficient code, the code generator must assign a fixed location in the AR for each temporary. If we use NT(e) to represent the number of temporaries to evaluate expression e. We will have

$$\begin{aligned}
NT(e_1 + e_2) &= \max(NT(e_1), 1 + NT(e_2)) \\
NT(e_1 - e_2) &= \max(NT(e_1), 1 + NT(e_2)) \\
NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) &= \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4)) \\
NT(id(e_1, \ldots, e_n)) &= \max(NT(e_1), \ldots, NT(e_n)) \\
NT(int) &= 0 \\
NT(id) &= 0
\end{aligned}$$

The rule for $id(e_1, \ldots, e_n)$ is correct because the temps to store $e_1, \ldots, e_{i-1}$ are stored in the new AR, not the current AR.

Consider the Fibonacci function (1.1). With the rules above applied, 2 temps are enough to evaluate it.

For a function definition `def f(x1,...,xn) = e`, the AR has `2 + n + NT(e)` elements: return address, frame pointer, n arguments and NT(e) locations for intermediate results. The layout of the AR is shown in Figure 1.2.

Now that we have knowledge of how many temps are needed to evaluate a function and where they are going to be stored in the AR, what's left is to keep track of how many temps are in use at each point in the program. To achieve this, we will add a new argument to code generation: **the position of the next available temp**. The temp area will be used like a small, fixed-size stack.

Here is the old code generation for `e1 + e2`:

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
```

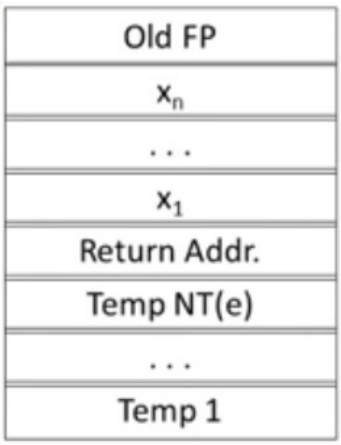

Figure 1.2: Layout of AR

```
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Under the new scheme, code generation will take a new argument:

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

1.5 Object layout

In this section we will focus on code generation for a more advanced feature: objects.

In OO programming, if B is a subclass of A, the an object of class B can be used whenever an object of class A is used, which means that code generated for class A has to work without any modification for class B. To give a complete description of the code generation for objects, two questions need to be answered:

- How are objects represented in memory?
- How is dynamic dispatch implemented?

We will use the following COOL code to illustrate the code generation for objects.

```
1  class A {
2    a:Int <- 0;
3    d:Int <- 1;
4    f():Int { a <- a + d };
5  };
6  class B inherits A {
7    b:Int <- 2;
8    f():Int { a };
9    g():Int { a <- a - b };
10 };
11 class C inherits A {
12   c:Int <- 3;
13   h():Int { a <- a * c };
14 }
```

`A` is the base class while `B` and `C` inherits from it. Note that attributes `a` and `d` are inherited by `B` and `C`, and that `a` is used in all methods of the 3 classes. For these methods to work correctly, attribute `a` must be in the same "place" in each object. To ensure this, 2 conditions have to be satisfied:

- Objects are laid out in contiguous memory.
- Each attribute is stored at a fixed offset in the object.

Let's take a look at the layout of an object in COOL, as shown in Figure 1.3. The first 3 words are always header information. **Class tag** is an integer that

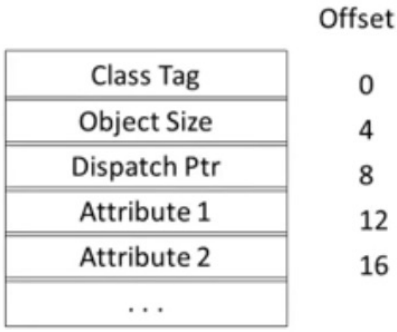

Figure 1.3: Layout of COOL object

identifies the class of the object. **Object size** is an integer that specifies the size of the object in words. **Dispatch pointer** is a pointer to a table of methods,

which will be explained in detail later. They are followed by attributes of the class in subsequent slots. All these are laid out in contiguous memory.

Given the layout of class A, the layout of its subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B. Thus the layout of A is left unchanged.

In our example, the layout of the 3 classes are shown in Table 1.1. In general,

Table 1.1: Layout of classes `A,B,C`

class \ offset	0	4	8	12	16	20
A	Atag	5	*	a	d	
B	Btag	6	*	a	d	b
C	Ctag	6	*	a	d	c

if B is a subclass of A, then an A object is "nested" inside an B object.

Every class has a fixed set of methods, including inherited methods. A **dispatch table** indexes these methods. It is an array of method entry points. A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses. The dispatch tables of classes `A,B,C` are shown in Table 1.2. The

Table 1.2: Dispatch tables of classes `A,B,C`

class \ offset	0	4
A	fA	
B	fB	g
C	fA	h

dispatch pointer in an object of class X points to the dispatch table of class X. The reason for which we use a pointer to the dispatch table rather than putting all the methods in each object directly is that each object can have its own copy of the attributes, but the methods are always the same.

Every method f of class X is assigned an offset O_f in the dispatch table at compile time. To implement a dynamic dispatch `e.f()`, we will evaluate `e` to get an object `x`, and then call $D[O_f]$, in which D is the dispatch table for `x`. In the call, `self` is bound to `x`.

1.6 Semantics