Chapter 1

Code generation

1.1 Stack machines

A stack machine uses only a stack as storage. When executing an instruction $\mathsf{r} = \mathsf{F}(\mathsf{a}_1, \ldots, \mathsf{a}_\mathsf{n})$, 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 add $r1, r_2, r_3$, 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 a n-register stack machine. As the name reveals, the top n positions of the pure stack machine's stack is held in registers. It turns out the even one single register can provide considerable performance improvement, which is the case of 1-register stack machine. The resgister 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$\leftarrow$acc + top_of_stack. In general, consider an arbitrary expression $\mathsf{op}(\mathsf{e}_1, \ldots, \mathsf{e}_\mathsf{n})$. 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>