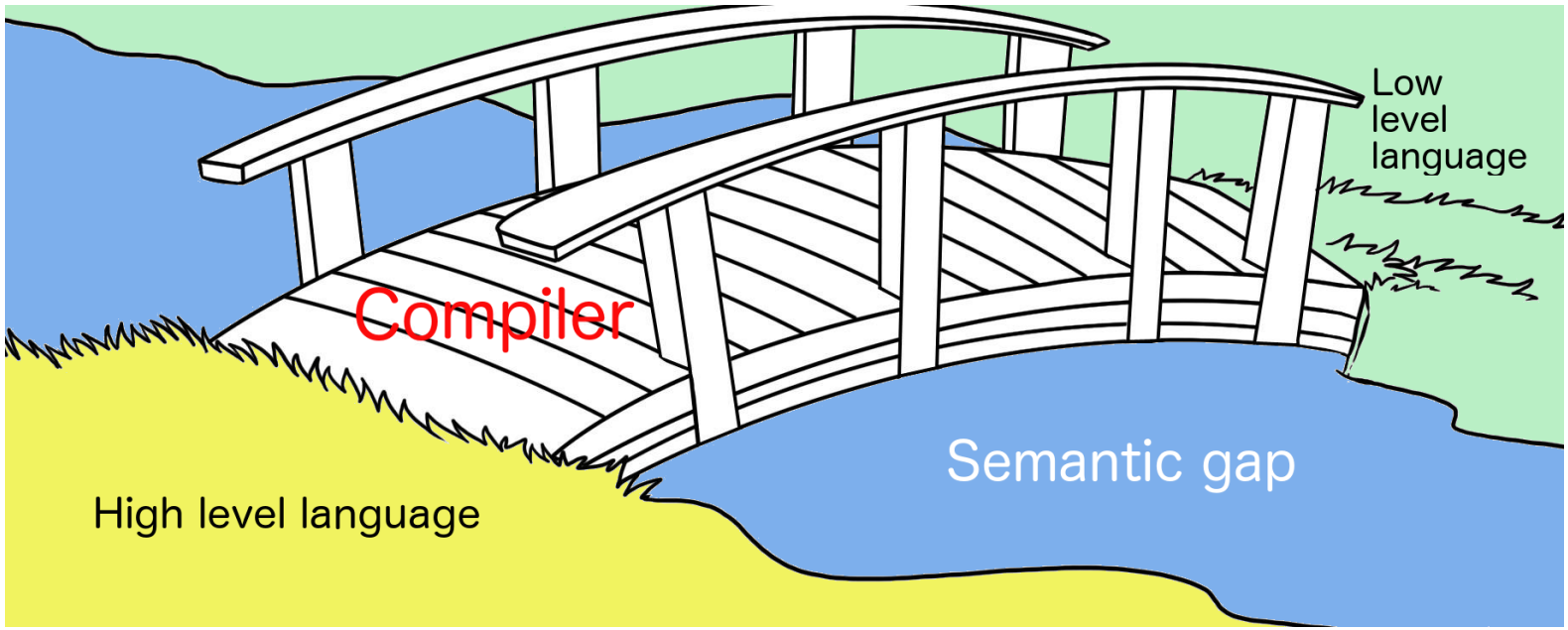
Compilers and computer architecture
Code-generation (2): register-machines

Martin Berger [1]

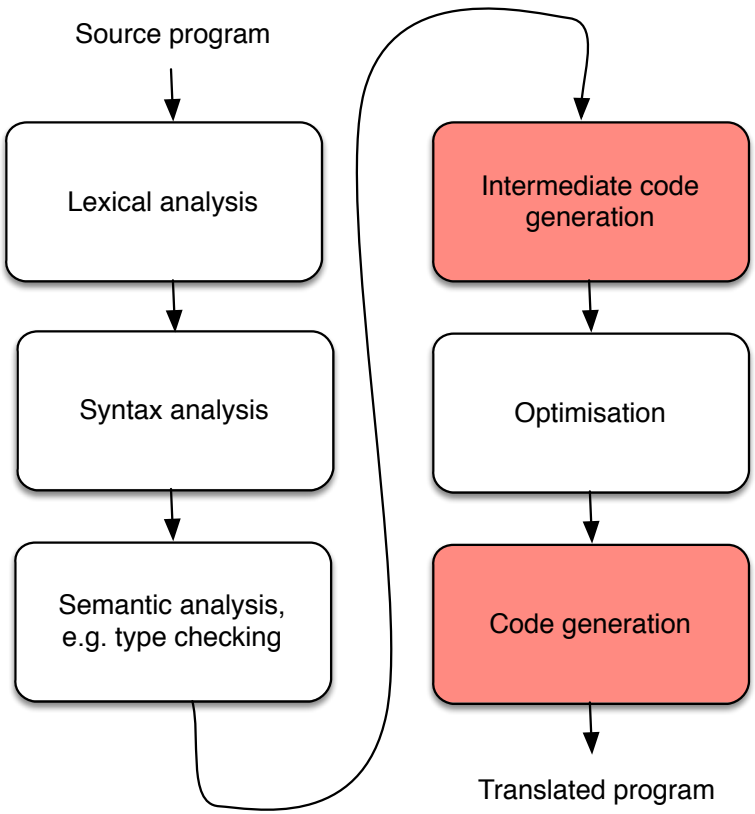
November 2019

[1] Email: M.F.Berger@sussex.ac.uk, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



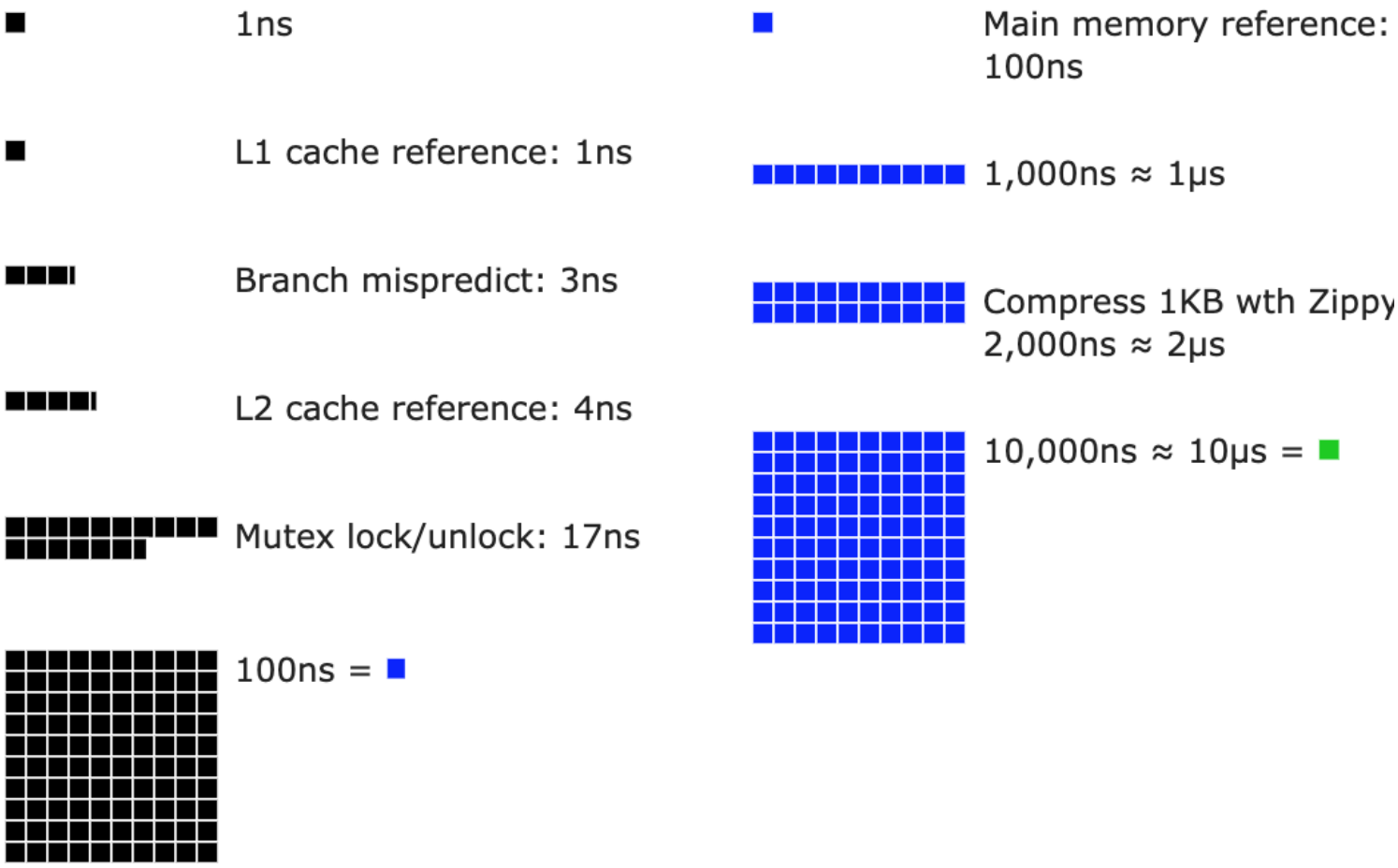
Plan for this week



In the previous section we introduced the stack machine architecture, and then investigated a simple syntax-directed code-generator for this architecture.

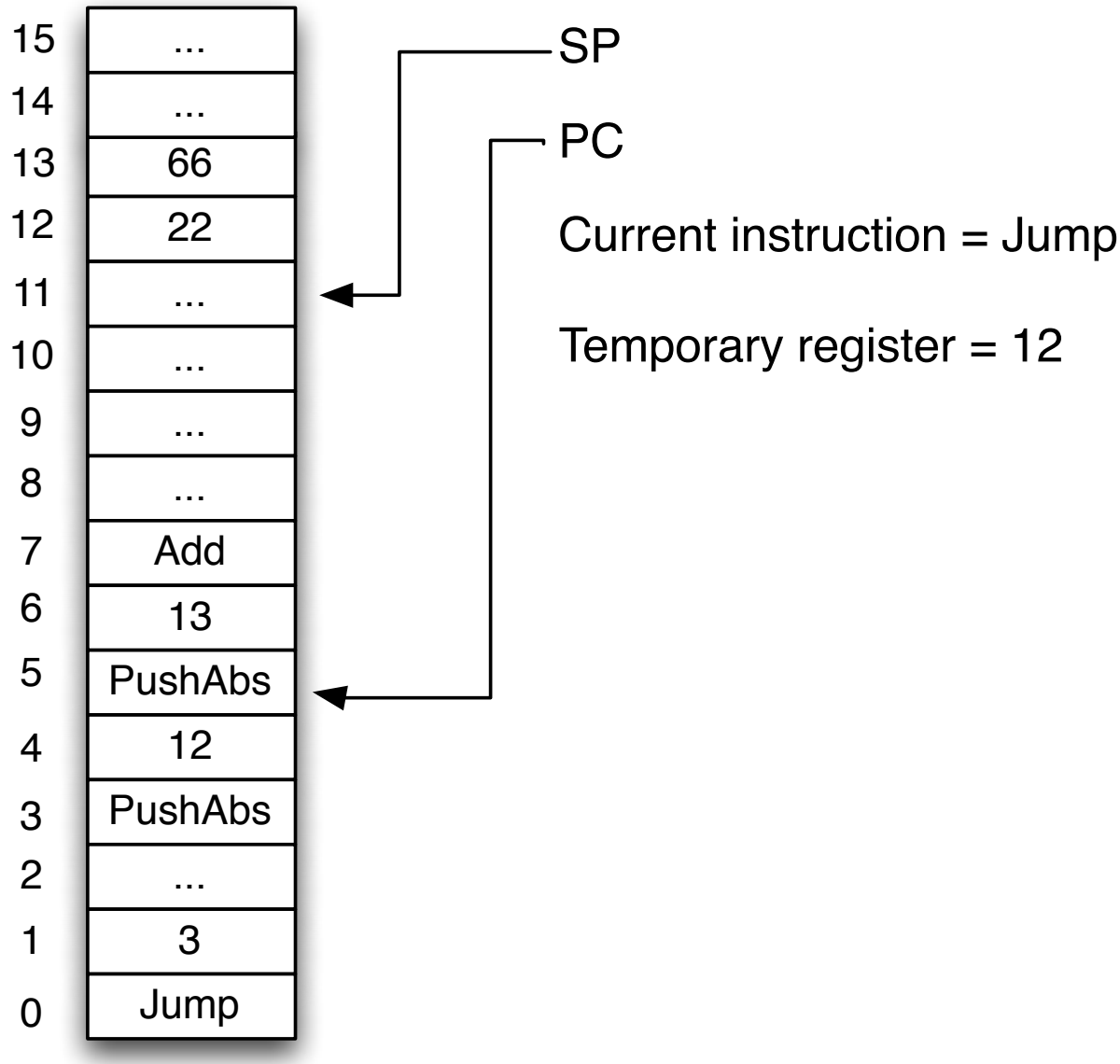
This week we continue looking at code generation, but for register machines, a faster CPU architecture. If time permits, we'll also look at accumulator machines.

Memory wall



From: `https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

Recall: stack machine architecture



Recall: stack machine language

Nop	Pop x
PushAbs x	PushImm n
CompGreaterThan	CompEq
Jump l	JumpTrue l
Plus	Minus
Times	Divide
Negate	

Important: arguments (e.g. to Plus) are always on top of the stack, and are 'removed' (by rearranging the stack pointer (SP)). The result of the command is placed on the top of the stack.

Register machines

The problem with the stack machine is that memory access (on modern CPUs) is **very slow** in comparion with CPU operations, approx. 20-100 times slower.

The stack machine forces us constantly to access memory, even for the simplest operations. It would be nice if the CPU let us store, access and manipulate data directly, rather than only work on the top elements of the stack.

Registers are **fast** (in comparison with memory), temporary, addressable storage in the CPU, that let us do this, whence register machines.

But compilation for register machines is more complicated than compilation for stack machines. Can you guess why?

Compilation for register machines

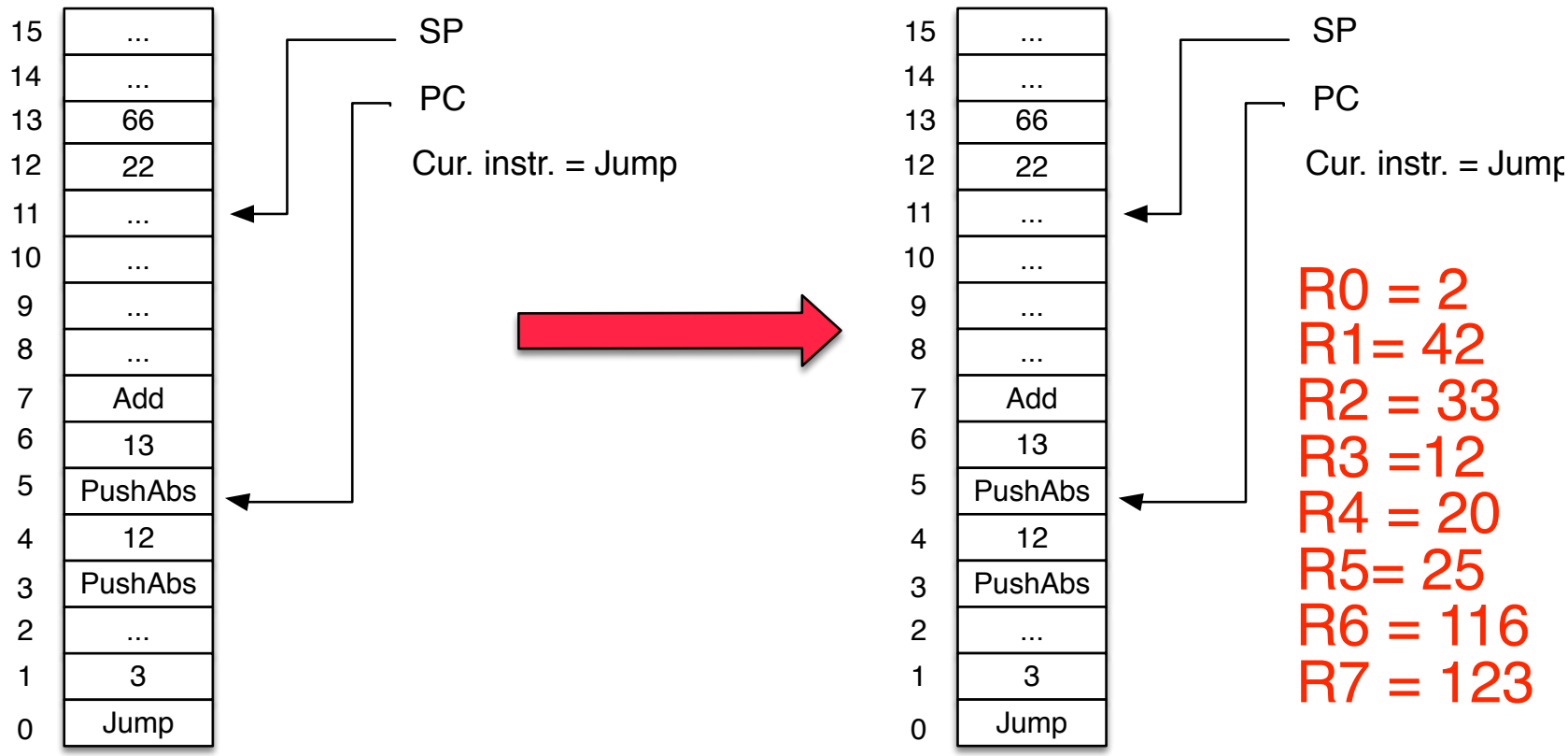
Each CPU has only a small finite number of registers (e.g. 16, 32, 128). That can be a problem. Why?

Because for small expressions, we can fit all the relevant parameters into the registers, but for the execution of larger expressions this is no longer the case. Moreover, what registers are available at each point in the computation depends on what other code is being executed. Hence a compiler must be able to do the following things.

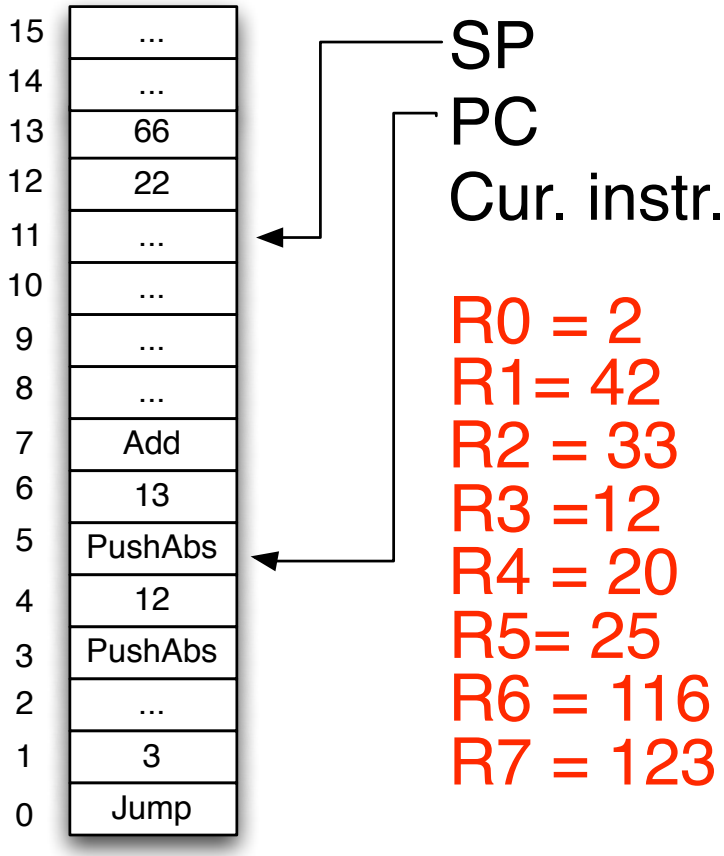
- Generate code that has all parameters in registers.
- Generate code that has some (or most) parameters in main memory.
- Detect which of the above is the case, and be able seamlessly to switch between the two.

All of this makes compilers more difficult. Let's look at register machines.

Goal: add registers



A simple register machine



Quite similar to the stack machine, but we have additional registers.
Important: operations like add, multiply operate on registers, no longer on the top of the stack

How to generate code for register machines?

Dealing with registers by 'divide-and-conquer'

In order to explain the difficult problem of generating efficient code for register machines, we split the problem into three simpler sub-problems.

- Generate code assuming an unlimited supply of registers.
- Modify the translator to evaluate expressions in the order which minimises the number of registers needed, while still generating efficient code.
- Invent a scheme to handle cases where we run out of registers.

Let's start by looking at register machines with an **unlimited** number of registers.

Commands for register machines

We assume an **unlimited** supply of registers R0, R1, R2, ... ranged over by r, r' We call these general purpose registers (as distinct from PC, SP).

Nop	Does nothing
Pop r	removes the top of the stack and stores it in register r
Push r	Pushes the content of the register r on stack
Load r x	Loads the content of memory location x into register r
LoadImm r n	Loads integer n into register r
Store r x	Stores the content of register r in memory location x
CompGreaterThan r r'	Compares the content of register r with the content of register r'. Stores 1 in r if former is bigger than latter, otherwise stores 0

Commands for register machines

CompEq r r’	Compares the content of register r with the content of register r. Stores 1 in r if both are equal, otherwise stores 0
Jump l	Jumps to l
JumpTrue r l	Jumps to address/label l if the content of register r is not 0
JumpFalse r l	Jumps to address/label l if the content of register r is 0
Plus r r’	Adds the content of r and r’, leaving the result in r
...	Remaining arithmetic operations are similar

Some commands have arguments (called operands). They take two (if the command has one operand) or three units of storage, the others only one. These operands need to be specified in the op-code, unlike with the stack machine. (Why?)

Commands for register machines

Question: Why do we bother with stack operations at all?

Important for e.g. procedure/method invocation. We'll talk about that later.

Source language

Our source language is unchanged: a really simple imperative language.

$$
\begin{array}{rcl}
M & ::= & M; M \mid \texttt{for}\ x = E\ \texttt{to}\ E\ \{M\} \mid x := E \\
E & ::= & n \mid x \mid E + E \mid E - E \mid E * E \mid E / E \mid -E
\end{array}
$$

Everything that's difficult to compile, e.g. procedures, objects, is left out. We come to that later.

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
          : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
          : List [ Instruction ]
```

Important convention 1: The result of evaluating the expression is returned in register `target`.

Important convention 2: The code generated by `codegenExpr( e, i )` can use registers Ri, Ri+1, ... **upwards**, but must leave the other registers R0 ... Ri-1 **unchanged**!

One way of thinking about this `target` is that it is used to track where the stack pointer would point.

Similar conventions for `codegen` for statements.

Code generation for constants

```
def codegenExpr ( exp : Expr, target : Register ) = {
    if exp is of shape
        ...
      Const ( n ) then
          List ( I_LoadImm ( target, n ) ) } }
```

Code generation for variables

```
def codegenExpr ( exp : Expr, target : Register ) = {
    if exp is of shape
           ...
    Ident ( x ) then
         List ( I_Load ( target, x ) )
```

Code generation for binary expressions

```
def codegenExpr ( exp : Expr, target : Register ) = {
    if exp is of shape
           ...
    Binop ( lhs, op, rhs ) then {
        codegenExpr ( rhs, target ) ++
        codegenExpr ( lhs, target+1 ) ++
        codegenBinop ( op, target, target+1 ) }
```

where

```
def codegenBinop ( op : Op, r1 : Register,
                            r2 : Register ) = {
    if op is of shape
     Plus then List ( I_Plus ( r1, r2 ) )
     Minus then List ( I_Minus ( r1, r2 ) )
     Times then List ( I_Times ( r1, r2 ) )
     Divide then List ( I_Divide ( r1, r2 ) ) } }
```

Code generation for binary expressions

Note that the call `codegenExpr(lhs, target+1)` in

```
Binop ( lhs, op, rhs ) then {
    codegenExpr ( rhs, target ) ++
    codegenExpr ( lhs, target+1 ) ++
    codegenBinop ( op, target, target+1 ) }
```

leaves the result of the first call `codegenExpr(rhs, target)` in the register `target` unchanged by our assumptions that `codegenExpr` never modifies registers below its second argument.

Please convince yourself that each clause of `codegenExpr` really implements this guarantee!

Example (x*3)+4

Compiling the expression (x*3)+4 (to target register r17, say) gives:

```
LoadImm r17 4
LoadImm r18 3
Load r19 x
Times r18 r19
Plus r17 r18
```

How can this be improved (1)?

Let's use commutativity of addition (a+b = b+a) and compile 4+(x*3)! When we compile it, we obtain:

```
LoadImm r17 3
Load r18 x
Times r17 r18
LoadImm r18 4
Plus r17 r18
```

How is this better?

Side by side

Compilation of (x*3)+4

```
LoadImm r17 4
LoadImm r18 3
Load r19 x
Times r18 r19
Plus r17 r18
```

Compilation of 4+(x*3)

```
LoadImm r17 3
Load r18 x
Times r17 r18
LoadImm r18 4
Plus r17 r18
```

The translation on the left uses 3 registers, while the right only two. We are currently assuming an unbounded number of registers, so who cares ... For realistic CPUs the number of registers is small, so smart translation strategies that save registers are better. More on this later!

Translation of statements

Similar to stack machine, except that arguments and results of expressions are held in registers. We'll see this in detail later.

Question: Does the `codegenStatement` method need to be passed a target register (as opposed to 'hard-coding' one)?
Answer: Yes, because statements may contain expressions, e.g. `x := x*y+3`.

Now we do something more interesting.

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Why is this interesting? We can combine two strategies:

- While > 1 free registers remain, be 'greedy': use the register machine strategy discussed above for compilation.
- When the limit is reached (ie. when there is one register left), revert to the accumulator strategy, using the last register as the accumulator.

The effect is that most expressions get the full benefit of registers, while unusually large expressions are handled correctly.