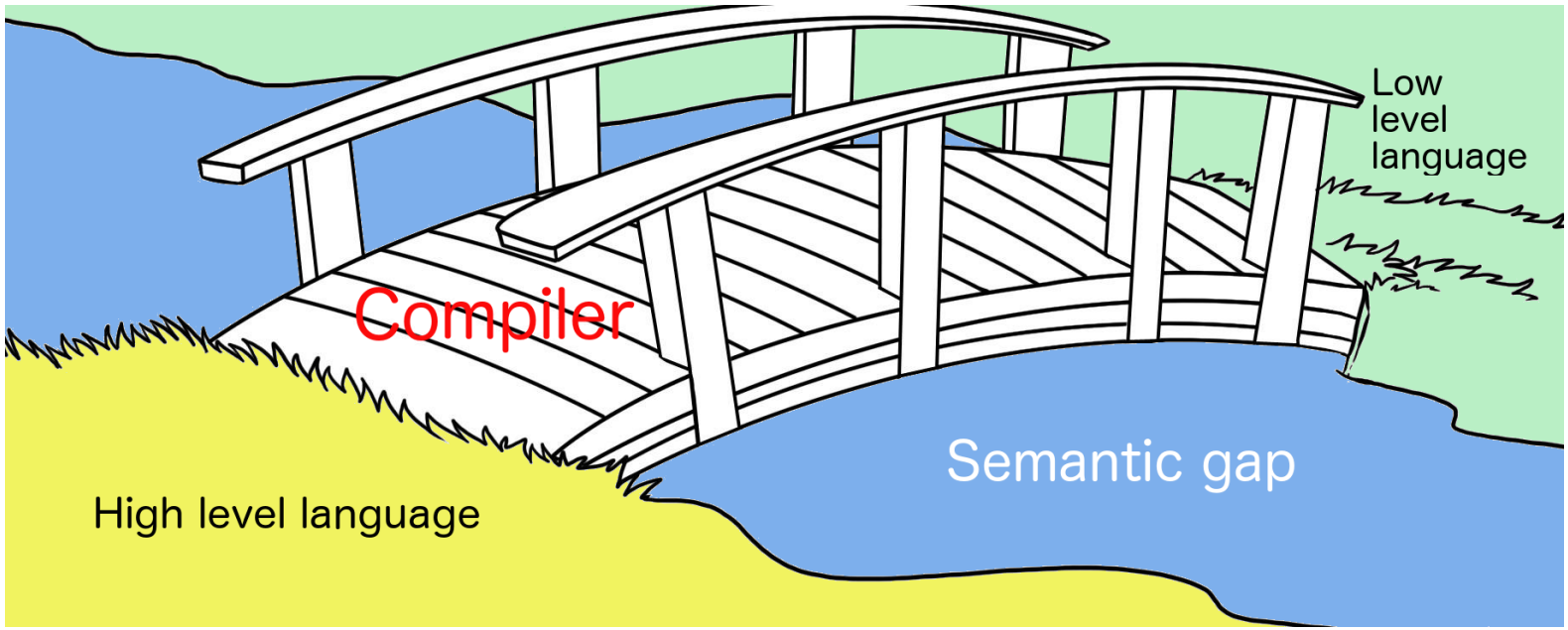
Compilers and computer architecture
Code-generation (1): stack-machines

Martin Berger [1]

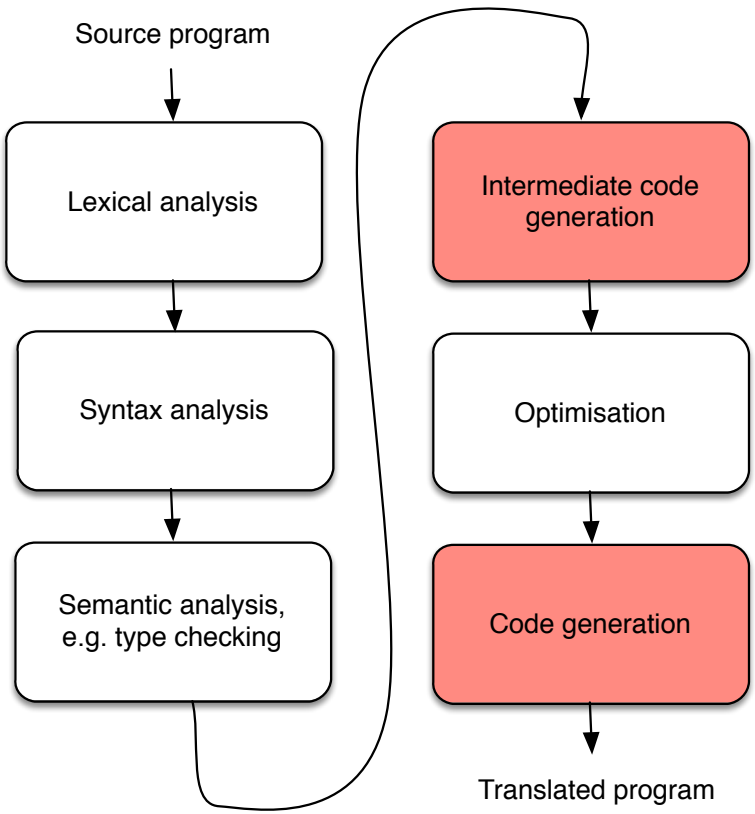
November 2019

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

Recall the function of compilers



Plan for the next two weeks

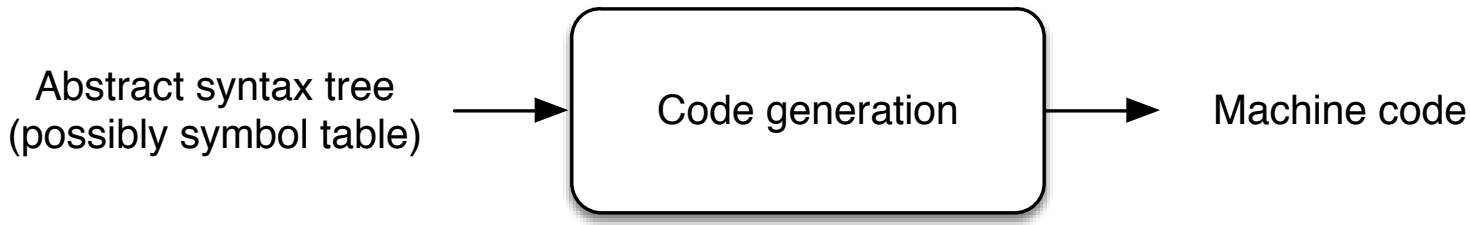


Remember the structure of a compiler?

We now look at code-generation. We start with the stack-machine architecture, because it is arguably the simplest machine architecture, allowing simple code generation. We then consider other, simple architectures such as the register and accumulator machines. This will give us all the tools we need to tackle a real processor (RISC-V).

Code generator input / output

Code generators have a simple structure.



Recall: ASTs are just convenient 'graphical' representations of programs that allow easy (= fast) access to sub-programs. When we see the code generators we'll realise why that is important for fast code generation.

Note that the code generator is completely isolated from the syntacte detail of the source language (e.g `if` vs `IF`).

Source language

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

Example program.

```
x := 0;
for i = 1 to 100 {
    x := x + i;
    x := x + 1 }
```

The code generator

A code generator takes as input an AST representing a program (here of type `AST`) and returns a program in machine code (assembler), here represented as a list of instructions.

```
def codegen ( s : AST ) : List [ Instruction ] = ...
```

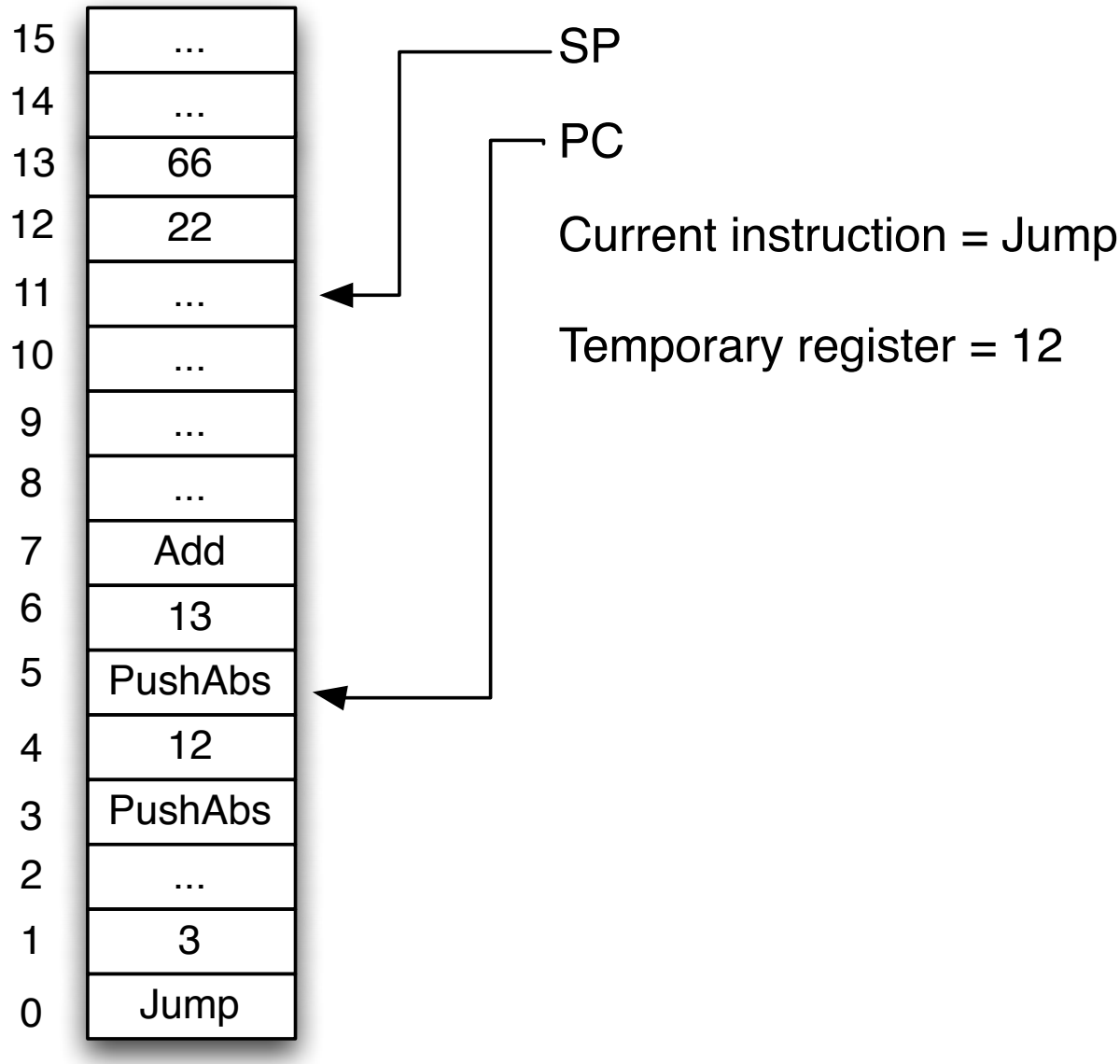
Compilation target: a simple stack machine

The stack machine consisting of the following.

- **Main memory**, addressed from zero up to some limit. Content of each memory cell is an integer. Stores code and data.
- A **program counter** (PC), pointing into the memory, to the command executed **next**.
- A **current instruction register** (IR/CI), holding the instruction currently being executed.
- A **stack-pointer** (SP), pointing into the memory to the **topmost item on** the stack. The stack grows **downwards**. Note: in some other architectures the SP points to the first **free** memory cell above or below the top of the stack.
- A **temporary register**, holding an integer.

This is simple, but can encode all computable programs. Realistic CPUs are more complicated – for speed!

The stack machine as a picture



Commands of the stack machine

Nop	Does nothing
Pop x	removes the top of the stack and stores it in x
PushAbs x	Pushes the content of the variable x on stack
PushImm n	Pushes the number n on stack
CompGreaterThan	Pops the top two elements off the stack. If the first one popped is bigger than the second one, pushes a 1 onto the stack, otherwise pushes a 0. (So 0 means `False`)
CompEq	Pops the top two elements off the stack. If both are equal, pushes a 1 onto the stack, otherwise pushes a 0.
Jump l	Jumps to l (l is an integer)
JumpTrue l	Jumps to address/label l if the top of the stack is not 0 Top element of stack is removed.

Commands of the stack machine

Plus	Adds the top two elements of the stack, and puts result on stack. Both arguments are removed from stack
Minus	Subtracts the top element of the stack from the element just below the top, and pushes the result on stack after popping the top two elements from the stack
Times	Multiplies the top two elements of the stack, and puts result on stack Both arguments are removed from stack
Divide	Divides the second element of the stack by the top element on the stack, and puts result on stack Both arguments are removed from stack
Negate	Negates the top element of the stack (0 is replaced by 1, any non-0 number is replaced by 0).

Commands of the stack machine

Note: PushImm 17 stores 17 on the top of the stack, while PushAbs 17 pushes the content of memory cell 17 on the top of the stack.

Note: Some commands (e.g. Pop) have an argument (called operand). They take up two units of storage. The remaining commands take only one.

Note: Removing something from the stack means only that the SP is rearranged. The old value is not (necessarily) overwritten.

Note: If the stack grows too large, it will overwrite other data, e.g. the program. The stack machine (like many other processor architectures) does not take any precautions to prevent such “stack overflow”.

Note: Jumping means writing the target address to the PC.

Commands of the stack machine in pseudo-code

```
Interface Instruction
   class I_Nop implements Instruction
   class I_Pop implements Instruction
   class I_PushAbs implements Instruction
   class I_PushImm implements Instruction
   class I_CompGreaterThan implements Instruction
   class I_CompEq implements Instruction
   class I_JumpTrue implements Instruction
   class I_Jump implements Instruction
   class I_Plus implements Instruction
   class I_Minus implements Instruction
   class I_Times implements Instruction
   class I_Divide implements Instruction
   class I_Negate implements Instruction

   class I_ConstInt ( n :  Int ) implements Instruction
   class I_DefineLabel ( id :  String )implements Instru
```

A convenient pseudo-command (1)

We need to store integers as second arguments, e.g. for `Pop`. This is the purpose of `I_ConstInt`. It's not a machine command but a **pseudo-code** representation of an integer.

A convenient pseudo-command (2)

We want to jump to addresses, e.g. `JumpTrue 1420`.

But numerical addresses like `1420` are hard to memorise for humans. It's better to have **symbolic** addresses like `JumpTrue Loop_Exit`.

The following pseudo-instruction allows us to do this.

```
abstract class Instruction
    ...

    class I_DefineLabel ( id : String )
            implements Instruction
```

Note that `I_DefineLabel` doesn't correspond to a machine instruction. It's just a convenient way to set up labels (humanly readable forms of addresses). Labels will be removed later (typically by the linker) and replaced by memory addresses (numbers). More on that later.

A typical assembly language program

```
start:
    PushAbs i
    PushImm 1
    Minus
    Pop i
    PushAbs i
    PushImm 0
    CompEq
    Negate
    JumpTrue start
```

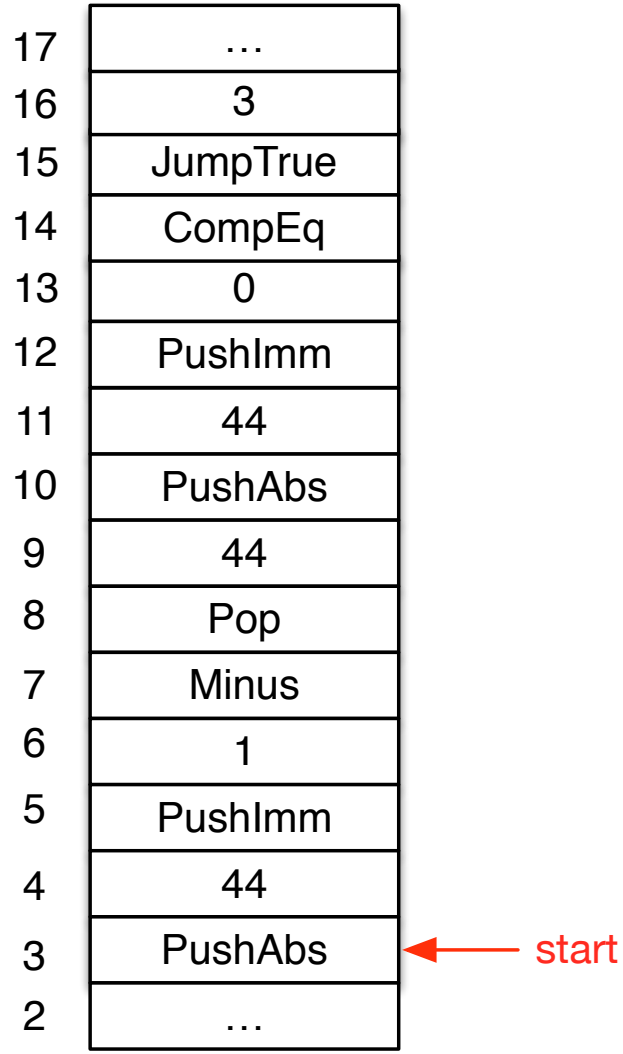
What does it do?

Note once more: labels like `start` appear in the compiler's output stream, even though they don't correspond to instructions. They will be removed later (e.g. by the linker). Can you think of another reason why symbolic addresses are a good idea? To enable running programs at different places in memory (relocation).

A typical assembly language program

```
start:
    PushAbs i
    PushImm 1
    Minus
    Pop i
    PushAbs i
    PushImm 0
    CompEq
    Negate
    JumpTrue start
```

If we were to start the program above at memory location 3, and the variable i was located at 44, then we'd get the memory layout on the right (each command would itself be represented as a number).



Stack machines have several advantages

- **Simplicty**: easy to describe & understand.
- **Simple compilers**: code generation for stack machines is much simpler than for register machines, since e.g. no register allocation is needed (we'll talk about register allocation later).
- **Compact object code**, which saves memory. The reason for this is that machine commands have no, or only one argument, unlike instructions for register machines (which we learn about later).
- **Simple CPUs** (= cheap, easy to manufacture).

Used in e.g. the JVM and WebAssembly.

Stack machines have disadvantages, primarily that they are **slow** (see e.g. the Wikipedia page on stack machines), but for us here simplicity of code generation is key.

The semantics of the stack machine

Before looking at the code generation process, I'd like to discuss the semantics of the stack machine in a slighly different manner, by giving a simple interpreter for stack machine commands. This enables you to implement (simulate) a stack machine.

I recommend that you do this yourself by translating the pseudo code below to a language of your choice.

The semantics of the stack machine (in pseudo-code)

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0 // Stack grows downwards.
                     // Question: why 0, not maxMem-1?
  private var temp = 0

  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      I_Nop then ...
      I_Pop then ...
      I_PushAbs then ...
      I_PushImm then ...
      I_CompGreaterThan then ...
      ... } } }
```

The semantics of the stack machine

The function `opcode` takes an integer (e.g. 7) and returns an instruction (e.g. `I_PushImm`).

Assigning numbers to instructions is by convention, (e.g. we could have associated 19 with `I_PushImm`). But each CPU architecture must make such a choice.

We are now ready to explain each instruction in detail.

Semantics of Nop

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      I_Nop then {} // I_Nop does nothing.
      ...
```

Semantics of Pop

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_Pop then {
        temp = mem ( sp )
        sp = ( sp + 1 ) % maxMem
        val operand = mem ( pc )
        pc = ( pc + 1 ) % maxMem
        mem ( operand ) = temp }
      ...
```

Semantics of PushAbs

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
     ...
      I_PushAbs then {
        val operand = mem ( pc )
        pc = ( pc + 1 ) % maxMem
        sp = ( sp - 1 ) % maxMem
        temp = mem ( operand )
        mem ( sp ) = temp }
      ...
```

Semantics of PushImm

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_PushImm then {
        temp = mem ( pc )
        pc = ( pc + 1 ) % maxMem
        sp = ( sp - 1 ) % maxMem
        mem ( sp ) = temp }
      ...
```

Semantics of CompGreaterThan

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_CompGreaterThan then {
        temp = mem ( sp )
        sp = ( sp + 1 ) % maxMem
        temp = temp - mem ( sp )
        if ( temp > 0 )
           mem ( sp ) != 0 // non-0 means true
        else
           mem ( sp ) = 0 } // 1 means false
```

Semantics of CompEq

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_CompEq then {
        temp = mem ( sp )
        sp = ( sp + 1 ) % maxMem
        temp = temp - mem ( sp )
        if ( temp == 0 )
           mem ( sp ) = 1 // 1 means true
        else
           mem ( sp ) = 0 } // non-1 means false
```

Semantics of Jump

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_Jump then {
        pc = mem ( pc ) }
      ...
```

Semantics of JumpTrue

```
class StackMachine ( maxMem : Int ) {
  ...
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_JumpTrue then {
        temp = mem ( sp )
        sp = ( sp + 1 ) % maxMem
        if ( temp == 1 ) // 1 means true, non-1
                         // means false
           pc = mem ( pc )
        else
           pc = ( pc + 1 ) % maxMem }
      ...
```

Semantics of Plus

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_Plus then {
        temp = mem ( sp )
        sp = ( sp + 1 ) % maxMem
        temp = temp + mem ( sp )
        mem ( sp ) = temp }
      ...
```

Semantics of Minus

```
class StackMachine ( maxMem : Int ) {

  private val mem = Array.fill ( maxMem ) ( 0 )
  private var pc = 0
  private var ir = 0
  private var sp = 0
  private var temp = 0
  while ( true ) {
    ir = mem ( pc )
    pc = ( pc + 1 ) % maxMem
    if opcode ( ir ) is of form
      ...
      I_Plus then {
        temp = mem ( sp )
        sp = ( sp + 1 ) % maxMem
        temp = mem ( sp ) - temp
        mem ( sp ) = temp }
      ...
```

Semantics of remaining commands

Similar to the above.

A naive code generator for the stack machine

We now present a syntax-directed code generator for the simple source language with assignment, sequencing and 'for' loops.

The structure of the translator is derived directed from the AST data type: we deal with each of the alternatives using a separate rule.

In Java a common approach is to use **reflection** (see `instanceof` or `getClass().getName()`). Alternatively, you can use a **visitor pattern** for AST traversal.

Recall syntax of source language

$$M ::= M; M \mid \texttt{for}\ x = E\ \texttt{to}\ E\ \{M\} \mid x := E$$

$$E ::= n \mid x \mid E + E \mid E - E \mid E * E \mid E / E \mid -E$$

A naive code generator for the stack machine

Recall that the signature of our code generator was as follows.

```
def codegen(s : AST) : List [Instruction] = ...
```

So we are looking to write the following pseudo-code:

```
def codegen ( s : AST ) = {
  if s is of form
    Sequence ( lhs, rhs ) then { ... }
    Assign ( x, rhs ) then { ... }
    For ( loopVar, from, to, body ) then { ... }
```

Translation of Sequencing

```
def codegen ( s : AST ) : ... = {
  if s is of form
    Sequence ( lhs, rhs ) then
       codegen ( lhs ) ++ codegen ( rhs )
       ...
```

Note that ++ is list concatenation

So all we are doing is generate the code for the lhs, and then concatenate it with the generated code for the rhs. This works because our assembly language has a sequencing operator (string concatenation), and we can map the sequencing of the source language directly to the sequencing operation of the target language.

In other words: **recursion** does most of the work here.

Translation of Assignment `x := E`

Note that we are assuming here to have a code generator for expressions (given soon) with the following signature.

```
def codegenExpr ( exp : Expr )
        : List [ Instruction ] = { ... }
```

What is the semantics of the code for expressions? The result of executing the **translated** expression **at run-time** is left on the **top of the stack**.

Translation of Assignment `x := E`

With this convention about `codegenExpr` we can now translate assignment as follows.

```
def codegen ( s : AST ) : ... = {
  if s is of form
    Assign ( x, rhs ) then
       codegenExpr ( rhs ) ++
       List ( I_Pop, I_ConstInt ( x ) )
    ...
```

Code in red is generated machine code, it will not be executed by the code generator.

Translation of Assignment

Note that we've been a bit sloppy as the constructor

```
class Assign ( x : String, rhs : Expr )
    implements AST
```

takes a string as first argument (because it's convenient for humans to use strings and give meaningful names to variables), but in

```
Assign ( x, rhs ) then
   codegenExpr ( rhs ) ++
   List ( I_Pop, I_ConstInt ( x ) )
```

we assume `x` is an integer (memory address), because that what the CPU expects. So really you'd have to add a 'mediator' to transform symbolic into numeric addresses (and/or vice versa).

Translation of For-Loops

To be able to translate loops, we need a new construct

```
newLabel ()
```

which, when invoked generates a fresh label every time it is called. For example `newLabel ()` returns the string `"label_1"` on first invocation and the string `"label_2"` on second invocation. So the implementation of `newLabel ()` must use a global counter, or some comparable mechanism.

Translation of For-Loops

```
def codegen ( s : AST ) : ... = {
  if s is of form
    For ( loopVar, from, to, body ) then {
      val loopCondition = newLabel ()
      val loopExit = newLabel ()
      codegenExpr ( from ) ++
      List ( I_Pop, I_ConstInt ( loopVar ),
             I_DefineLabel ( loopCondition ) ) ++
      codegenExpr ( to ) ++
      List ( I_PushAbs, I_ConstInt ( loopVar ),
             I_CompGreaterThan ,
             I_JumpTrue, I_ConstInt ( loopExit ) ) ++
      codegen ( body ) ++
      List ( I_PushAbs, I_ConstInt ( loopVar ),
             I_PushImm, I_ConstInt ( 1 ),
             I_Plus ,
             I_Pop, I_ConstInt ( loopVar ),
             I_Jump, I_ConstInt ( loopCondition ),
             I_DefineLabel ( loopExit ) ) }
  ...
```

Translation of expressions

Remember our convention that the result is **always** left on the top of the stack.

Other conventions are possible, e.g. leave it in the temporary variable.

Recall expressions:

$$E \quad ::= \quad n \mid x \mid E + E' \mid E - E' \mid E * E' \mid E/E' \mid -E$$

Translation of expressions

```
def codegenExpr ( exp : Expr ) : List [ Instruction ] =
  if opcode ( exp ) is of form
     Binop ( lhs, op, rhs ) then
        codegenExpr ( rhs ) ++
        codegenExpr ( lhs ) ++
        codegenBinop ( op )
     Unop ( Minus, e ) then
        List ( I_PushImm, I_ConstInt ( 0 ) ) ++
        codegenExpr ( e ) ++
        List ( I_Minus )
     Ident ( x ) then List(I_PushAbs, I_ConstInt (x))
     Const ( n ) then List(I_PushImm, I_ConstInt (n))
```

Note that we are assuming here to have a code generator for binary and expressions (given soon) with the following signature.

```
def codegenBinop (op : Op) : List[Instruction] = {...}
```

Translation of binary operations

```
def codegenBinop ( op : Op ) : List [ Instruction ] =
  if op is of form
    Plus then List ( I_Plus   )
    Minus then List ( I_Minus   )
    Times then List ( I_Times   )
    Divide then List ( I_Divide   ) } }
```

We are 'lucky' here in that the arithmetic operations of our source language map directly to corresponding instructions in the target language (stack machine commands). For more complex arithmetic operations this cannot be guaranteed, e.g. m^n or $cosin(x)$. The translation of those is more involved.

Example translation

Consider the following program.

```
x := 0;
for i = 1 to 100 {
    x = x + i;
    x = x + 1 }
```

Translation of program from prev. slide

For tersity we write e.g. `I_PushImm (76)` instead of

```
I_PushImm
I_ConstInt (76)
```

and likewise for all other commands with arguments.

Translation of program from prev. slide

```
    I_PushImm ( 0 )          // Begin first command x = 0
    I_Pop ( x )
    I_PushImm ( 1 )          // Initialisation of loop
    I_Pop ( i )
I_DefineLabel ( loopCondition )
    I_PushImm ( 100 )        // test for loop termination
    I_PushAbs ( i )
    I_CompGreaterThan
    I_JumpTrue ( loopExit )
    I_PushAbs ( i )          // Command x = x+i
    I_PushAbs ( x )
    I_Plus
    I_Pop ( x )
    I_PushImm ( 1 )          // Command x = x+1
    I_PushAbs ( x )
    I_Plus
    I_Pop ( x )
    I_PushImm ( 1 )          // Incrementing loop variable
    I_PushAbs ( i )
    I_Plus
    I_Pop ( i )
    I_Jump ( loopCondition )
I_DefineLabel( loopExit )
```

Conclusion

This chapter has shown how a code generator can be written, which takes an AST as input and produces a working assembler program as output.

We divided the problem into two parts: code generation for statements (e.g. assignment, looping, sequencing etc), and code generation for expressions.

For each statement type, the code generator uses a standard "template" heavily based on recusive calls to the code generator; the details of the statement determine how the gaps are filled in.

For expressions we used a simple, stack-based scheme; we will study better, more complicated CPU architecture soon.

We haven’t yet looked at procedures, objects, declarations, records, etc.

The material in the textbooks

- Dragon Book: Chapter 2, introduction to code generation, Chapter 8, especially 8.1 and 8.6.
- Appel, Palsberg: Chapter 7, Chapter 9 (although Appel, Palsberg skip simple code generation and concentrate on finding the best instruction to match the context).
- "Engineering a compiler": Section 4.4: ad-hoc syntax-directed translation, especially Figure 4.14.