Part II

Chapter 8
Computer Architecture and Languages

This chapter shows how a small computer architecture, called the Mac-1, and associated languages can be described in Clean. We describe machines by a two level model. At the bottom layer machine components and the access functions to handle them are described. The top layer describes the machine instructions in terms of these access functions. In the first two sections we describe the state and instructions of Mac-1. Next, section 3 shows how this machine specification can be executed. In section 8.4 this machine is extended by memory mapped output.

In this chapter we will use the conventional machine level as the lowest level of abstraction. It is very well possible to give a circuit specification in a functional language, see for example [O'Donnell 95].

Programs for this machine consist of a sequence of bit strings, or numbers. This is fine for a machine, but very troublesome for human beings. In section 8.5 we illustrate the concept of second generation languages by developing an assembly language for Mac-1.

The next section illustrates how the behaviour of the specified machine can be observed by adapting the execution mechanism of the machine. Neither the machine state nor the stored programs need to be changed.

High level languages abstract from the details of a specific machine. Which makes programs written in such a language machine independent. A very small imperative language is introduced in section 8.7. Implementation of a functional language on Mac-1 is too complicated to be treated in this chapter. We show how the introduced high level language can be translated to Mac-1 assembler in section 8.8 and how programs in the high

level language can interpreted in the next section. Finally we discuss the relation between the interpreter and the compiler in section 8.10.

Although it is not a prerequisite, some existing knowledge about imperative programming will make it easier to understand this chapter.

8.1 Computer Architecture

A machine consists of a collection of memory components and hardware to change the machine state stored in these memory components. Each memory element is able to hold some information. In concrete machines, information is always stored in the form of bits. The number of bits stored in a memory element varies from one to several millions. These bits are the concrete representation of everything which is stored in the machine. In more abstract machines more abstract values, like numbers, data types and functions, can be stored directly.

To illustrate the concepts of machine description we need an example architecture. In this chapter we will use a machine called Mac-1 as example. This machine was introduced to illustrate the concept of micro-programming [Tanenbaum 84]. It serves our purpose very well.

Mac-1 has a traditional machine architecture. The set of available operations is at the same level of abstraction as the instruction set of modern processors like the PowerPC™ or the Pentium™. The same description method can be applied at a lower level of abstraction to describe micro-programming, and at higher levels of abstraction to describe high-level architectures [Plasmeijer 93, Koopman 95].

Mac-1 is a small machine with a main memory of 4096 (2^{12}) 16-bit words. These words have consecutive addresses from 0 up to 4095. This memory contains the program to be executed and a stack. The stack grows from high memory addresses to lower ones. Furthermore, there are three one-word memories called registers. The top of the stack is indicated by the register called stack pointer (sp). The current instruction is indicated by the register named program counter (pc). The machine has one register, the accumulator (ac), to store the result of computations. The architecture of Mac-1 is depicted in figure 8.1. Although all memory words contain bit strings we have shown an appropriate interpretation of the bit strings in the figure. The pc and sp are pointers to memory words. The ac and the stack contains usually numbers. The bit string indicated by the pc is interpreted as the current instruction.

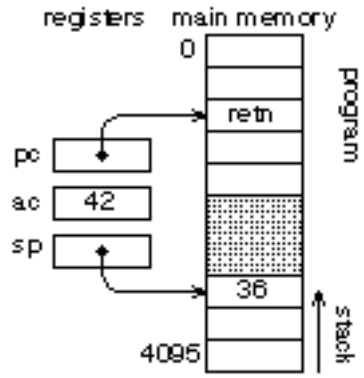

Figure 8.1 The architecture of Mac-1

The state of Mac-1 contains four memory components: the three registers and the main memory. This is represented in Clean as a record with the four obvious fields.

```
:: State =  { pc :: Word    // program counter
            , ac :: Word    // accumulator
            , sp :: Word    // stack pointer
            , mm :: Memory  // main memory
            }
```

The manipulations of words and the memory, as well as their representation in Clean is discussed in the next section.

8.1.1 Memory Components

In this section we will chose a concrete representation for the type Word and introduce the manipulations of words and the main memory needed. We will use knowledge about the chosen representation of words to handle the output of the machine in section 8.4. Also in the conversion of numbers from the assembly language and the high level language to Mac-1 numbers during the generation of Mac-1 code in sections 8.5 and 8.8 respectively, some knowledge about the number representation is used.

The type `Word` is represented by the type `Int` of Clean. When a `Word` is treated as a signed number two's complement notation is used. Whenever the first bit of the binary notation is 1 (word ≥ `word_bound`/2) the number is interpreted in 2's complement as the negative number obtained by subtracting `word_bound` from the ordinary interpretation. This implies that the negative numbers are represented by the integers between 2^{15} and 2^{16}. The value in the Mac-1 world of such a Clean integer can be obtained by subtracting 2^{16} from the integer. Addition, `|+|`, and subtraction, `|-|`, are done modulo 2^{16} in order to keep the numbers within the machine precision.

Numbers larger than 2^{15} are treated as negative numbers in 2's complement notation. The function `neg` tests whether the word represents a negative number. The function `format` transforms a `Word` to an `Int`, this is used in the simulation of the machine.

In the subtraction we add 2^{16} in order to make sure that negative numbers are represented by large integers.

```
:: Word  :== Int
:: Addr  :== Word        // Only 12 bits are significant

word_bound :== 2^16      // The maximum word + 1
sign_bound :== 2^15      // The maximum signed word + 1
addr_bound :== 2^12      // The maximum addres + 1
oper_bound :== 2^12      // The bound on 12 bits operands

(|+|) infixl 6 :: Word Word -> Word
(|+|) n m = (n+m) mod word_bound

(|-|) infixl 6 :: Word Word -> Word
(|-|) n m = (n-m+word_bound) mod word_bound

neg :: Word -> Bool
neg w  = w >= sign_bound     // Large integers are interpreted as negative numbers.

format :: Word -> Int
format n | n > sign_bound = n-word_bound
                          = n
```

The main memory is represented as a list of words. The manipulation functions must take care that the length of this list is always 4096. Using an array of words instead of a list of words for the representation of the memory would make it is easier to meet this requirement. We have chosen to represent the memory as a list of words, since lists are manipulated more easily in Clean than arrays.

There are functions to store an initial program in the memory, to select, `|!|`, and to update, `|:=|`, an individual word.

The function `eval_list` is used to reduce the list representing the memory to normal form (a list of integers instead of an expression yielding a list of integers). The function `update` combines the evaluation and updating of an element. Evaluation is necessary to prevent excessive heap usage.

```
:: Memory   :== [Word]           // Length should always be 2^12 = 4096.
```

```
store :: [Word] -> Memory
store init = eval_list (take addr_bound (init ++ repeat zero))

(|!|) infixl 9 :: Memory Addr -> Word
(|!|) mm a = mm ! (a mod addr_bound)

(|:=|) infix 5 :: Addr Word -> (.Memory -> .Memory)
(|:=|) a w = \ m -> update (a mod addr_bound) w m

update :: Int x [x] -> [x]  // updates addressed element and forces evaluation
update 0 w [x:r] = [w:r]
update a w [x:r] | a>0 = let! r` = update (a-1) w r
                         in [x:r`]

eval_list :: [x] -> [x]     // Forces the evaluation of the list
eval_list [] = []
eval_list [x:xs] = let! x; xs` = eval_list xs
                   in [x:xs`]
```

We assume here that all implementations of Clean works with integers that are longer than 16 bits. Whenever you are working with a 16 bit integer Clean implementation the definitions of the operators must be adapted. The Clean calculation with integers will be modulo 2^{16} so the modulo operations can be removed from the addition and subtraction operation since 2^{16} equals 0 in 16-bit arithmetic. When you happen to work with a Clean implementation with integers of less then 16 bits you need to use the representation developed in exercise 8.1. You can check whether your Clean implementation is able to handle the integer 2^{16} properly or not by evaluating `Start = 2^16`. This should evaluate to `65536.`

8.2 Instructions

Instructions change the state of the machine in a well defined way. We represent the instructions as a function that takes the current machine state as argument and yields the new state. Hence the type of an instruction is

```
:: Instruction :== State -> State
```

Many instructions consists of an identifying part and an operand. For instance, the instruction jump to continue program execution at the specified address consists of an identifying part, the clean function `jump`, and the address. Although we call the function `jump` informally often the jump instruction, the type of the function `jump` is:

```
jump :: Addr State -> State
```

Unfortunately the Clean type system does not allow us to write:

```
jump :: Addr -> Instruction.
```

Like every machine Mac-1 has a limited set of instructions. These instructions can be grouped in the following categories:

Load: load the accumulator with the specified operand;

Store: store the contents of the accumulator at the specified address in the memory;

Add: add the operand to the contents of the accumulator, the sum is stored in the accumulator;

Sub: subtract the operand from the contents of the accumulator, the result is stored in the accumulator;

Jump: set the program counter to the specified address (some jumps are conditional on the contents of the accumulator);

Push: push the specified operand on the stack;

Pop: pop an item from the stack;

Swap: exchange the contents of the accumulator and stack pointer;
Sp handling: *increment* or *decrement* the stack pointer by the 8-bit constant given in the instruction.

Many instructions need an operand. For example the Load instruction needs a value to store in the accumulator. These operands can be specified in different ways. A way to specify where an operand can be found is called an addressing mode.

Four addressing modes are provided in this machine:

immediate: the operand is specified in the instruction;
direct: the instruction contains the address of the operand;
indirect: the address of the operand is in the accumulator;
local: the operand is on the stack, the offset in the stack is given in the instruction.

The addressing mode is indicated by the name of the instruction used. Not all combinations of addressing modes and categories are available. The next table shows the available instructions.

	Load	*Store*	*Add*	*Sub*	*Jump*	*Push*	*Pop*	*Swap*	*Sp*
immediate	loco	–	addd	subd	all jumps	–	–	–	insp desp
direct	lodd	stod	–	–	–	–	–	–	–
indirect	–	–	–	–	–	pshi	popi	–	–
local	lodl	stol	addl	subl	–	–	–	–	–
no operand	–	–	–	–	retn	push	pop	swap	–

An example of an instruction using direct addressing we show the load direct instruction. This instruction takes the address of the word to store in the accumulator as argument:

```
lodd :: Addr State -> State
lodd x s = {s & ac = s.mm|!|x}
```

The instruction push indirect is an example of a more complicated instruction. In this instruction the stack is extended with one word by a decrement of the stack pointer. The memory at the location indicated by the updated stack pointer (the new top of the stack) is updated by the word addressed by the accumulator.

```
pshi :: Instruction
pshi s = { s & sp = sp`
         , mm = (sp` |:=| (s.mm|!|s.ac)) s.mm}
           where sp` = s.sp |-| 1
```

A complete list of Mac-1 instructions is (for uniformity we indicate all elements in the record used to represent the state as s.…):

```
loco x s = {s & ac = x}                                          // load constant
lodd x s = {s & ac = s.mm|!|x}                                   // load direct
lodl x s = {s & ac = s.mm|!|(s.sp |+| x)}                        // load local
stod x s = {s & mm = (x |:=| s.ac) s.mm}                         // store direct
stol x s = {s & mm = ((s.sp |+| x) |:=| s.ac) s.mm}              // store local
addd x s = {s & ac = s.ac |+| s.mm|!|x}                          // add direct
addl x s = {s & ac = s.ac |+| s.mm |!| (s.sp |+| x)}             // add local
subd x s = {s & ac = s.ac |-| s.mm|!|x}                          // subtract direct
subl x s = {s & ac = s.ac |-| s.mm |!| (s.sp |+| x)}             // subtract local
jump x s = {s & pc = x}                                          // jump
jpos x s = if (~(neg s.ac)) {s & pc = x} s                       // jump positive
jzer x s = if (s.ac==0)     {s & pc = x} s                       // jump zero
jneg x s = if (neg s.ac)    {s & pc = x} s                       // jump negative
jnze x s = if (s.ac<>0)     {s & pc = x} s                       // jump not zero
call x s = {s & pc = x, sp = sp`, mm = (sp` |:=| s.pc) s.mm}     //call subroutine
           where sp` = s.sp |-| 1
retn   s = {s & sp = s.sp |+| 1, pc = s.mm |!| s.sp}             // return
push   s = {s & sp = sp`, mm = (sp` |:=| s.ac) s.mm}             // push ac
           where sp` = s.sp |-| 1
pshi   s = {s & sp = sp`, mm = (sp` |:=| (s.mm|!|s.ac)) s.mm}    // push indirect
           where sp` = s.sp |-| 1
pop    s = {s & sp = s.sp |+| 1, ac = s.mm|!|s.sp}               // ac := top; pop
```

```
popi   s = {s & sp = s.sp |+| 1, mm = (s.ac |:=| (s.mm|!|s.sp)) s.mm} // pop indirect
swap   s = {s & ac = s.sp, sp = s.ac}                                  // swap sp, ac
insp y s = {s & sp = s.sp |+| y}                                       // increment sp
desp y s = {s & sp = s.sp |-| y}                                       // decrement sp
```

This is the complete list of Mac-1 instructions. The type of these functions is omitted for reasons of brevity.

8.2.1 Storing instructions in the memory

Like everything else the instruction sequence to be executed, the program, has to be stored in the state of Mac-1. Each instruction is encoded in a single word. More complex machines can have instructions that are encoded in a number of words. Words can be interpreted in several ways in a machine architecture. For Mac-1 we have the following interpretations: number, signed number, address and instruction. In section 8.4 we will add yet another interpretation: characters. It is important to realise that you cannot look at a word and tell how it should be interpreted. In principle each word can be interpreted in all these ways. However, it is possible that a word has no value in some interpretation. For example there is no instruction associated with the word consisting of sixteen 1-bits.

For Mac-1 the encoding of instructions in word is arbitrary. By convention the high order bits of the word, the left-hand part, indicate the instruction encoded in the word: the opcode. When the instruction needs an operand it is stored in the low order bits, the right-hand part. Not that the opcodes operands does not have a fixed size.

Words are decoded to instructions by the function `decode`. This function returns a Boolean indicating whether the decoding was successful, and the instruction. Words are decoded by selecting the interpretation as number of a group of bits. For instance `word/2^12` yields the interpretation as number of the first 4 bits, `word mod 2^4` gives this interpretation of the last four bits. The bits that are checked in some function alternative are printed bold in the comment of that line.

```
decode :: Word -> (Bool, Instruction)
decode word
 = case (word/2^12) of                 // word equals the bit string
    0  -> t (lodd x)                   // 0000 XXXX XXXX XXXX = 0x0XXX
    1  -> t (stod x)                   // 0001 XXXX XXXX XXXX = 0x1XXX
    2  -> t (addd x)                   // 0010 XXXX XXXX XXXX = 0x2XXX
    3  -> t (subd x)                   // 0011 XXXX XXXX XXXX = 0x3XXX
    4  -> t (jpos x)                   // 0100 XXXX XXXX XXXX = 0x4XXX
    5  -> t (jzer x)                   // 0101 XXXX XXXX XXXX = 0x5XXX
    6  -> t (jump x)                   // 0110 XXXX XXXX XXXX = 0x6XXX
    7  -> t (loco x)                   // 0111 XXXX XXXX XXXX = 0x7XXX
    8  -> t (lodl x)                   // 1000 XXXX XXXX XXXX = 0x8XXX
    9  -> t (stol x)                   // 1001 XXXX XXXX XXXX = 0x9XXX
    10 -> t (addl x)                   // 1010 XXXX XXXX XXXX = 0xAXXX
    11 -> t (subl x)                   // 1011 XXXX XXXX XXXX = 0xBXXX
    12 -> t (jneg x)                   // 1100 XXXX XXXX XXXX = 0xCXXX
    13 -> t (jnze x)                   // 1101 XXXX XXXX XXXX = 0xDXXX
    14 -> t (call x)                   // 1110 XXXX XXXX XXXX = 0xEXXX
    15 -> case (x/2^8) of              // 1111 .... .... .... = 0xF...
           0  | y==0 -> t pshi         // 1111 0000 0000 0000 = 0xF000
           2  | y==0 -> t popi         // 1111 0010 0000 0000 = 0xF200
           4  | y==0 -> t push         // 1111 0100 0000 0000 = 0xF400
           6  | y==0 -> t pop          // 1111 0110 0000 0000 = 0xF600
           8  | y==0 -> t retn         // 1111 1000 0000 0000 = 0xF800
           10 | y==0 -> t swap         // 1111 1010 0000 0000 = 0xFA00
           12        -> t (insp y)    // 1111 1100 YYYY YYYY = 0xFCYY
           14        -> t (desp y)    // 1111 1110 YYYY YYYY = 0xFEYY
           n         -> (False,I)     // word is not an instruction
where
    t i = (True,i)
    x = word mod 2^12
    y = word mod 2^8
```

Now we have defined the state of Mac-1, `State`, and the ways to changes this state by specifying the instructions. What remains to be specified is which instruction is executed in a given program state. This is done in the next section.

8.3 Running the Machine

In this section we make the specification complete by specifying which instruction must be applied in a given machine state. The program counter contains the address of the next instruction to be executed. As explained in the previous section, this instruction is encoded in one memory word. This word must be decoded to an instruction. This instruction is applied to the state with an incremented program counter. The program counter is incremented in order to execute the next instruction in the following instruction cycle. The instruction is applied to the state where the program counter is incremented. This implies that all manipulations of the `pc` by an instruction are done on an incremented `pc`. The instruction cycle is interrupted when the word indicated by the program counter does not represent an instruction

```
instruction_cycle :: State -> State
instruction_cycle state=:{pc,mm}
 | is_instr = instruction_cycle (instruction {state & pc = pc|+|1}))
            = state
    where   (is_instr, instruction) = decode (mm|!|pc)
```

Real-world machines usually execute a trap when the word indicated by the program counter cannot be interpreted as an instruction. Execution of a trap merely means that a new program counter is loaded from a fixed address. The current program counter is pushed on the stack. This trap-address contains the address of the trap-handler. The trap-handler is a function that determines how the error should be handled. It is easy to implement this. The current approach is more convenient for simulation.

8.3.1 Booting the Machine

Now we have defined the state, the state transitions and the determination of the instruction to be applied. The only thing that is needed in order to run the specification is an initial state. The initial state is created by the function `boot`. The program to be executed is passed as an argument and stored in the memory. The registers are set to some appropriate predefined values. We have chosen to start program execution from address 0.

```
boot :: [Word] -> State
boot program =  { pc = 0
                , ac = 0
                , sp = init_sp
                , mm = store program
                }

init_sp :== 4092
```

For real machines booting can be a nasty problem. In general a program is needed to load the initial program. This initial loader is usually stored in read only memory (ROM). Since this problem is somewhat similar to the story of the Baron Von Münchhausen who raised himself by his own bootstraps out of the mud [Raspe 1785], the process of loading the initial program into the machine is called bootstrapping.

8.4 Input-Output

A machine that cannot communicate with the world is pretty useless. Mac-1 uses a simple way to communicate with the world: the values written to address 4094 are visible for the world outside Mac-1. This output register is called `out`. The word at address 4095, the output status register, `osr`, is used by Mac-1 to indicate that a new word is written to

the output port. The value of osr is cleared (set to zero) as a side-effect of writing to out. Reading the value of the output port from outside sets the output status register to one. This form of communication with the world is called memory mapped I/O.

In our description of Mac-1 we model the memory mapped output by a new interpretation of the words out and osr. We decided to specify osr and out as two additional registers to the memory. This makes these registers more explicit and is closer to reality.

```
:: Memory = { osr :: Word        // output status register
            , out :: Word        // output port
            , mem :: [Word]      // main memory
            }

addr_osr :== 4095
addr_out :== 4094

free :== 2^15
busy :== 0
```

In the definitions below we show how the memory manipulation functions are changed to reflect the new interpretation of these memory words. Updating the memory at the address of the output register has a side-effect on the output status register. Assigning a value to osr has no effect, its value is determined by the output device. Reading a word from the addresses of on of the new registers yields the value of the register instead of the value in the main memory mem. We show below how this form of output is used.

```
(|!|) infixl 9 :: Memory Addr -> Word
(|!|) mm a | a==addr_osr = mm.osr
           | a==addr_out = mm.out
                         = mm.mem ! (a mod addr_bound)

store :: [Word] -> Memory
store init = { osr=free
             , out=0
             , mem=eval_list (take addr_bound (init ++ repeat zero))}

(|:=|) infix 5 :: Addr Word -> (.Memory -> .Memory)
(|:=|) a w
 | a==addr_out = \ m -> {m & out=w, osr=busy}
               = \ m -> {m & mem=update (a mod addr_bound) w m.mem}
```

The instruction cycle is changed to pass numbers from the output port to the world. We have chosen a very straight forward design here. The word is interpreted as an integer. It becomes part of the result of the instruction cycle in the same cycle as the word in written to the appropriate memory location. For a concrete realisation of Mac-1 doing output can be much slower. In contrast with the current implementation Mac-1 programs on such a realisation may detect a busy output port.

```
instruction_cycle :: State -> [Int]
instruction_cycle state=:{pc,mm}
 | is_instr = output ++ instruction_cycle state`
            = []
   where    (is_instr,instr) = decode (mm|!|pc)
            (output,state`)  = do_io (instr {state & pc = pc|+|1})

do_io :: State -> ([Int],State)
do_io s=:{mm}
 | mm.osr==busy = ([format (mm.out)],{s & mm={mm & osr=free}})
                = ([],s)
```

A similar approach is used to pass input to the machine. The word at address 4092 serves as input register. The input status register, isr, at address 4093 is set to 1 when new input is available at address 4092. Reading the input register clears the input status register.

The original Mac-1 treats the input and output words as ASCII characters instead as plain numbers. See exercise 16.

8.5 Assembly Languages

The first generation of programming languages consists of sequences of numbers to be interpreted as programs by machines. An example is the list of words loaded in Mac-1 by `boot`. It is clear that this a cumbersome and error-prone way to program computers.

Writing symbolic names for the instructions instead of numbers is an significant improvement. These machine specific languages are usually called assembly languages and are the second generation of programming languages. There is a ono-to-one mapping from the assembly program to the program represented as a sequence of numbers. The advantage of assembly programs is that they are much better readable for human beings and that it is easier to change them. Especially using labels, symbolic addresses, instead of direct addresses makes it much easier to change an assembly program. The translation from assembly statements to numbers can be done by a computer program called assembler.

Assembly statements for Mac-1 are represented by the algebraic data type `AStatement`. There exists a statement for each instruction, and there are three additional statements. These additional statements are used to represent numbers (usually data), a stop instruction to terminate program execution, and for the definition of labels. Labels are symbolic names for addresses. The assembler computes the real addresses corresponding to the labels. This makes it much easier to change an existing program and is less error-prone.

```
:: AStatement
 = Lodd Address | Stod Address  | Addd Address  | Subd Address  | Call Address
 | Jpos Address | Jzer Address  | Jump Address  | Jneg Address  | Jnze Address
 | Loco Int     | Lodl Int      | Stol Int      | Addl Int      | Subl Int
 | Pshi         | Popi          | Push          | Pop           | Retn
 | Swap         | Insp Int      | Desp Int      |
 | Const Int    | Stop          | Label String

:: Address  =   L String | C Addr        // Label or Constant address
:: Assembly :== [AStatement]
```

The assembler translates each statement to the corresponding word. No code is generated for label definitions. The assembler replaces applied occurrences of labels by the corresponding addresses. The function `code` assigns a machine `Word` to each other assembler statement.

```
assembler :: Assembly -> [Word]
assembler statements
 = words
   where
   (words,labels) = assemble 0 statements (translate labels) // a cycle in labels !

assemble :: Addr Assembly (Address -> Addr) -> ([Word],String -> Addr)
assemble n [] trans = ([],\s -> abort ("label "+s+" is not defined"))
assemble n [Label l:rest] trans = (words,\s -> if (s==l) n (labs s))
    where   (words,labs) = assemble n rest trans
assemble n [statement:rest] trans = ([word:words],labs)
    where   (words,labs) = assemble (n+1) rest trans
            word = case statement of
                    Lodd a -> c1 0  (trans a)
                    Stod a -> c1 1  (trans a)
                    Addd a -> c1 2  (trans a)
                    Subd a -> c1 3  (trans a)
                    Jpos a -> c1 4  (trans a)
                    Jzer a -> c1 5  (trans a)
```

```
                    Jump a -> c1 6  (trans a)
                    Loco c -> c1 7  (index c 12)
                    Lodl n -> c1 8  (index n 12)
                    Stol n -> c1 9  (index n 12)
                    Addl n -> c1 10 (index n 12)
                    Subl n -> c1 11 (index n 12)
                    Jneg a -> c1 12 (trans a)
                    Jnze a -> c1 13 (trans a)
                    Call a -> c1 14 (trans a)
                    Pshi   -> c2 0  0
                    Popi   -> c2 2  0
                    Push   -> c2 4  0
                    Pop    -> c2 6  0
                    Retn   -> c2 8  0
                    Swap   -> c2 10 0
                    Insp i -> c2 12 (index i 8)
                    Desp i -> c2 14 (index i 8)
                    Stop   -> c2 0  1 // some invalid opcode
                    Const n | 0<=n&&n<word_bound -> n
                            | -2^15<n&&n<0 -> n+word_bound
                                           -> abort ("Bad Const "+toString n)
            c1 opcode x = opcode*2^12 + x
            c2 opcode y = 15*2^12 + opcode*2^8 + y

index :: Int Int -> Int
index n p | 0<=n && n<2^p   = n
                            = abort ("Index "+toString n+" is out of range")
```

The function `translate` turns assembler addresses in the corresponding numerical values. Labels are collected by the function `assemble` in the same pass as the transformation of statements to words. The `assembler` feeds the label information back to `assemble`, and creates a cycle in doing this. The `assembler` assumes that the program is loaded from address 0 in the memory; the first instruction will be at address 0.

```
translate :: (String -> Addr) Address -> Addr
translate addrs (C n)
    | is_Addr n = n
                = abort ("Constant "+toString n+" not an address")
translate addrs (L s) | is_Addr n = n
                                  = abort ("Label "+s+" out of range")
                                    where n = addrs s
```

8.5.1 An Example Assembly Program

The Mac-1 program shown here computes a Fibonacci number in a naïve recursive way. The Fibonacci function defined in Clean is:

```
fib :: Int -> Int
fib n | n<2 = 1
            = fib (n-1) + fib (n-2)
```

The argument and result of this function are passed in the accumulator of Mac-1. The argument of the Fibonacci function is passed as argument to the Clean function containing the assembly as data structure of type `assembly`. Usually a special syntax is used for assembly programs. An assembly program will be represented by a list of characters. The parser translating the list of characters to the data structure `assembly` is straight forward.

Here we omit the special syntax and represent programs by their abstract syntax tree. It is straightforward to construct a parser using the tools developed in chapter II.5.

```
fib_program :: Int -> Assembly
fib_program n
 = [        Loco    n               //   0: Load argument in accumulator.
     ,      Call    (L "fib")       //   1: Call the Fibonacci function.
     ,      Stod    (C addr_out)    //   2: Print result.
     ,      Stop                    //   3: Terminate the execution.
     ,      Label                   //      The Fibonacci function.
    "fib", Subd    (L "two")       //   4: Compute n-2.
```

```
,       Jneg    (L "neg")       //   5: Goto neg if n-2 < 0.
,       Push                    //   6: Push n-2.
,       Addd    (L "one")       //   7: Compute n-1.
,       Call    (L "fib")       //   8: Compute fib (n-1).
,       Push                    //   9: Push fib (n-1).
,       Lodl    1               //  10: Load n-2 in accumulator.
,       Call    (L "fib")       //  11: Compute fib (n-2).
,       Addl    0               //  12: Add fib (n-2) to fib (n-1).
,       Insp    2               //  13: Clear stack.
,       Retn                    //  14: Return to caller.
,       Label                   //      Handle n-2<0.
"neg", Loco     1               //  15: Load accumulator with result.
,       Retn                    //  16: Return to caller.
,       Label
"one", Const    1               //  17: The constant 1.
,       Label
"two", Const    2               //  18: The constant 2.
]
```

Using the assembler (`Start = assembler (fib_program 5)`), the Fibonacci program can be transformed to the Mac-1 representation of this program.

```
[28677,57348,8190,61441,12306,49167,62464,8209,57348,62464,32769,…,63488,1,2]
```

Using hexadecimal numbers instead of decimal numbers enables us to recognise the groups of four bits in each word. This is slightly better readable. For this reason the hexadecimal notation of numbers was very popular in the early days of computer science.

```
[0x7005,0xE004,0x1FFE,0xF001,0x3012,0xC00F,0xF400,0x2011,0xE004,…,0x0001,0x0002]
```

It is evident that the assembly version is much better readable for human beings. This causes also a severely improved maintainability of the program.

The Fibonacci program can be executed by feeding the obtained lists of words into Mac-1: `Start = (instruction_cycle o boot o assembler) (fib_program 5)`. The machine produces the desired output: `[8]`.

8.6 Tracing the Execution of Programs

The machine description developed in the previous sections is able to execute Mac-1 programs. However, it remains unclear in which way the machine reaches the observable output. We can make the behaviour of the machine visible at a step-by-step level by adjusting of the `instruction_cycle`. Apart from producing output and applying the current instruction, the instruction cycle shows important information about the state of Mac-1.

```
trace_cycle state=:{pc,ac,sp,mm}
 | is_instr = [ ( "pc",pc,"ac",ac,format ac,"sp",sp
                , "out",output,"instruction", instr
                , "stack",[format(mm|!|w)\\w<-[sp..init_sp-1]]
                )
              : trace_cycle state`]
            =   []
   where        (is_instr,instr) = decode (mm|!|pc)
                (output,state`)  = do_io (instr {state & pc = pc|+|1})
```

The behaviour of the Fibonacci program can be observed in a step by step fashion by executing `Start = (trace_cycle o boot o assembler) (fib_program 2)`. After some typographical post processing the trace is:

cycle	pc	ac	sp	out	instruction	stack
1	0	0	4092		loco 2	[]
2	1	2	4092		call 4	[]
3	4	2	4091		subd 18	[2]
4	5	0	4091		jneg 15	[2]
5	6	0	4091		push	[2]
6	7	0	4090		addd 17	[0,2]
7	8	1	4090		call 4	[0,2]

8	4	1	4089		subd 18	[9,0,2]
9	5	-1, 65535	4089		jneg 15	[9,0,2]
10	15	-1, 65535	4089		loco 1	[9,0,2]
11	16	1	4089		retn	[9,0,2]
12	9	1	4090		push	[0,2]
13	10	1	4089		lodl 1	[1,0,2]
14	11	0	4088		call 4	[1,0,2]
15	4	0	4088		subd 18	[12,1,0,2]
16	5	-2, 65534	4088		jneg 15	[12,1,0,2]
17	15	-2, 65534	4088		loco 1	[12,1,0,2]
18	16	1	4088		retn	[12,1,0,2]
19	12	1	4089		addl 0	[1,0,2]
20	13	2	4089		insp 2	[1,0,2]
21	14	2	4091		retn	[2]
22	2	2	4092	2	stod 4094	[]
23	3	2	4092		stop	[]

The instruction numbers included in the comments of the fib_program can be used to understand this trace more easily.

8.7 High Level Languages

As shown above the introduction of assembly languages is a serious step forward in computer programming. A further improvement of the programming language can be achieved by abstracting from the details of Mac-1. Instead of specifying how the value of some expression can be computed, the expression itself is listed. For example the statemet `y := 3x+2` can be implemented by the following assembly program (assuming `"x"` and `"x"` are the appropriate addresses):

```
[   Loco    2
,   Addd    "x"
,   Addd    "x"
,   Addd    "x"
,   Stod    "y"
]
```

It is obvious that the statement `y := 3*x+2` is better understandable for human beings.

The computer program that translates high-level language to code that can be executed on a machine is called a compiler. Important advantages of these third generation languages are the higher abstraction level and the independence of a specific machine: one high-level language can be compiled to code for several different machines. A drawback of these high level languages is that it is not always possible to exploit all possibilities of the machine used. The first third-generation language was FORTRAN, many other imperative languages (like COBOL, C, Pascal, Modula-2, …) were introduced afterwards.

To illustrate the concepts of a third generation language and the associated transformations we will introduce a very small language called Tiny. Programs in the language Tinny consists of a list of statements. The language Tiny has only five different statements. The firsts statement is the declaration of a variable: `Declare`. A variable is an abstraction of a memory word. In tiny a variable can be set at an initial value by its declaration, the AST will always contain an initial value. A variable can be set to the value of an expression by an assignment: `Assign`. Usually statements are executed in the order they are listed. Two statements can be used to change this. The conditional, `If`, contains one expression and two sequences of statements. One of these sequences is executed depending on the value of the expression. The next statement, `While`, can be used to execute a sequence of instructions as long as the conditional expression yields true. Finally there is a statement to `Print` the value of an expression. As usual we will represent Tiny programs by their abstract syntax tree. Adding an parser for Tiny programs is straight forward.

```
:: Tiny :== [TStatement]

:: TStatement = Declare Variable    Expression
              | Assign  Variable    Expression
              | If      Expression  [TStatement] [TStatement]
              | While   Expression  [TStatement]
              | Print   Expression
```

Expressions in Tiny are build with a very limited set of constructs:

```
:: Expression = Add     Expression  Expression
              | Sub     Expression  Expression
              | Eq      Expression  Expression
              | Less    Expression  Expression
              | Var     Variable
              | CONST   Int
              | TRUE
              | FALSE

:: Variable :== String
```

Real imperative languages possess a richer palette of constructions for expressions and statements. For example, the concept of a function is missing in Tiny. However, the constructions and expressions of Tiny are sufficient to illustrate the concept of a third generation language.

8.7.1 An Example Program in Tiny

Without having functions, Fibonacci numbers cannot be computed with the algorithm used in the assembler program `fib_program`. The algorithm used in the Tiny program shown below computes *fib n* by adding *n* times the two previous Fibonacci numbers. In the left column we show the abstract syntax tree, in the right column we show a possible corresponding syntax.

```
imp_fib n =
 [Declare "x" (CONST n)
 ,Print (Var "x")
 ,Declare "fibn" (CONST 1)
 ,Declare "fibn-1" (CONST 0)
 ,Declare "fibn-2" (CONST 0)
 ,While (Var "x")                    //  x ≠ 0
     [Assign "fibn-2" (Var "fibn-1")
     ,Assign "fibn-1" (Var "fibn")
     ,Assign "fibn" (Add (Var "fibn-1") (Var "fibn-2"))
     ,Assign "x" (Sub (Var "x") (CONST 1))
     ]
 ,Print (Var "fibn")
 ]
```

```
var x := n;
print(x);
var fibn := 1;
var fibn-1 := 0;
var fibn-2;
WHILE x        // x ≠ 0
DO fibn-2 := fibn-1;
   fibn-1 := fibn;
   fibn := fibn-1 + fibn-2;
   x := x - 1
END;
print(fibn)
```

This program can be executed using the compiler introduced below and the machine description introduced above. Executing `Start = run (imp_fib 5)` produces `[5,8]`.

8.8 Compilation

The task of the compiler developed in this section is to translate a Tiny program to a program that can be executed by Mac-1. Instead of generating a sequence of words we will generate an `Assembly` program. Generating assembly has the advantage that the generated code is better understandable for humans and that it is easier due to the use of symbolic addresses (labels). The assembler is used to convert the assembly program to machine-code.

```
compile :: (Tiny -> [Word])
compile = assembler o compiler
```

The compiler produces a list of assembly statements consisting of code corresponding to the Tiny program, a `Stop` statement, a sequence a statements that can be used by the gen-

erated programs, and the variables and constants introduced. We assume that the Tiny programs to compile are correct: e.g. all variables are properly defined before they are used, no declaration of variables in a conditional or loop, etc.

```
compiler :: Tiny -> Assembly
compiler prog
 =  instructions         // Statements corresponding to the Tiny program
    ++ [Stop]            // The Stop statement
    ++ fixed_code        // The fixed code used by the generated program
    ++ declare decls     // The declarations of the variables and constants used in the Tiny program.
where (instructions,decls,_) = comp prog ["l_"+toString n\\n<-[1..]]
```

The `compiler` uses two additional functions to compile statements. The function `comp_s` generates code for a single statement and `comp` generates assembly code for a list of statements by applying `comp_s` to each Tiny statement. Both functions take a list of strings as additional argument. These strings are used as labels in the generated code. Apart from the assembly code, these functions deliver the global declarations needed and the unused labels.

```
comp :: [TStatement] [String] -> ([AStatement],[(String,Int)],[String])
comp []       labels = ([],[],labels)
comp [s:rest] labels = (s_code++code_rest,decl++decl_rest,labels``)
            where     (s_code,decl,labels`) = comp_s s labels
                      (code_rest,decl_rest,labels``) = comp rest labels`
```

For each declaration `comp_s` yields the name and initial value in the appropriate place in the result. The function `declare` will turn this into the appropriate assembly statements. For an assignment we use `comp_e` to generate code that leaves the value of the expression in the accumulator and a store direct statement to update the variable. A conditional expressions is implemented by code to evaluate the condition and jumps to execute the appropriate sequence of statements. For a `While` loop we generate code to check the condition, a conditional jump to leave the loop, the code corresponding to the body, and a jump back to the start of the loop. A print statement is compiled to code to evaluate the expression and a call to the appropriated function in the `fixed_code`.

```
comp_s :: TStatement [String] -> ([AStatement],[(String,Int)],[String])
comp_s (Declare v (CONST n)) labs = ([],[(v,n)],labs)
comp_s (Declare v TRUE)      labs = ([],[(v,1)],labs)
comp_s (Declare v FALSE)     labs = ([],[(v,0)],labs)
comp_s (Assign v e)          labs = (e_code++[Stod (L (var v))],[],labs`)
                                    where (e_code,labs`) = comp_e e labs
comp_s (If c t e) [l1,l2:labs]
 = (c_code++[Jzer (L l1)]++t_code++[Jump (L l2),Label l1]++e_code++[Label l2]
   ,c_decls++t_decls++e_decls
   ,labs3)
  where (c_code,c_decls,labs1) = comp_e c labs
        (t_code,t_decls,labs2) = comp t labs1
        (e_code,e_decls,labs3) = comp e labs2
comp_s (While c b) [l1,l2:labs]
 = ([Label l1]++c_code++[Jzer (L l2)]++b_code++[Jump (L l1),Label l2]
   ,c_decls++b_decls
   ,labs2)
  where (c_code,c_decls,labs1) = comp_e c labs
        (b_code,b_decls,labs2) = comp b labs1
comp_s (Print e) labs
 = (e_code ++ [Call (L "print")], [], labs1) //  call function in fixed_code
  where (e_code,labs1) = comp_e e labs
```

The function `comp_e` generates assembly code for an expressions. The result of the expression is left in the accumulator. Code is generated by recursive descent of the data structure `Expressions`. Addition and subtracting are done by generating code for the second argument, pushing this value on the stack, performing the operation and clearing the stack. For the comparison operations we generate code that pushes both evaluated arguments on the stack and call the appropriate function in the `fixed_code`. Constants and variables can

be loaded immediately. The value `0` is used as representation of `FALSE`, all other values are treated as representations of `TRUE`.

```
comp_e :: Expression [String] -> ([AStatement],[(String,Int)],[String])
comp_e (Add e1 e2) labs
 = (e2_code++[Push]++e1_code++[Addl 0,Insp 1],e1_decls++e2_decls,labs2)
  where (e1_code,e1_decls,labs1) = comp_e e1 labs
        (e2_code,e2_decls,labs2) = comp_e e2 labs1
comp_e (Sub e1 e2) labs
 = (e2_code++[Push]++e1_code++[Subl 0,Insp 1],e1_decls++e2_decls,labs2)
  where (e1_code,e1_decls,labs1) = comp_e e1 labs
        (e2_code,e2_decls,labs2) = comp_e e2 labs1
comp_e (Eq e1 e2) labs
 = (e2_code++[Push]++e1_code++[Push,Call (L "Eq")],e1_decls++e2_decls,labs2)
  where (e1_code,e1_decls,labs1) = comp_e e1 labs
        (e2_code,e2_decls,labs2) = comp_e e2 labs1
comp_e (Less e1 e2) labs
 = (e2_code++[Push]++e1_code++[Push,Call (L "Less")],e1_decls++e2_decls,labs2)
  where (e1_code,e1_decls,labs1) = comp_e e1 labs
        (e2_code,e2_decls,labs2) = comp_e e2 labs1
comp_e (CONST n) [l:labs]
 | n>=0 &&  n <oper_bound          = ([Loco n]          ,[]     , labs)
 | n>=addr_bound &&  n <word_bound = ([Lodd (L (var l))],[(l,n)], labs)
 | n<0 &&(~n)<signed_bound = ([Lodd (L (var l))],[(l,n+word_bound)],labs)
comp_e TRUE     labs = ([Loco 1          ],[],labs)
comp_e FALSE    labs = ([Loco 0          ],[],labs)
comp_e (Var v)  labs = ([Lodd (L (var v))],[],labs)
```

The function `declare` generates code for the declaration of variables and constants that does not fit in the instruction: a label and a word containing the (initial) value.

```
declare :: [(String,Int)] -> [AStatement]
declare vars = flatten [[Label (var v),Const n]\\(v,n)<-vars]

var :: String -> String
var v = "v_"+v
```

To run a program we use the compiler to generate assembly code corresponding to the program. The assembler transforms this assembly code to machine code. This sequence of words is loaded into the machine by `boot`. The instruction cycle executes the program.

```
run :: (Tiny -> [Int])
run = instruction_cycle o boot o assembler o compiler
```

The fixed code is used to print numbers and for the comparison operators. Using fixed code instead of copying the desired code to every place where it is used limits the size of the generated code and compiler.

```
fixed_code :: Assembly
fixed_code
    =   [               Label                   // Print routine, number in ac.
        "print",        Push                    // Push number to print.
        ,               Label
        "wait",         Lodd    (C addr_osr)    // Load status register.
        ,               Jzer    (L "wait")      // wait until pervious print is finished.
        ,               Pop                     // Load number to print.
        ,               Stod    (C addr_out)    // Store number in output register.
        ,               Retn

        ,               Label                   // Equality, args on stack, result in ac.
        "Eq",           Lodl    1               // ac := arg1
        ,               Subl    2               // ac := arg1-arg2
        ,               Jzer    (L "Ret_TRUE")  // ac = 0 ⇔ arg1 = arg2
        ,               Label                   // ac ≠ 0 ⇔ arg1 ≠ arg2
        "Ret_FALSE",    Pop                     // Return FALSE: ac := return address.
        ,               Insp    2               // Remove args.
        ,               Push                    // Return address on stack.
        ,               Loco    0               // Load FALSE.
        ,               Retn                    //
        ,               Label                   // Comparison, args on stack, result in ac.
```

```
        "Less",         Lodl    1               // ac := arg1
        ,               Subl    2               // ac := arg1-arg2
        ,               Jpos    (L "Ret_FALSE") // ac ≥ 0 ⇔ arg1 ≥ arg2
        ,               Label                   // ac < 0 ⇔ arg1 < arg2
        "Ret_TRUE",     Pop                     // Return TRUE: ac := return address.
        ,               Insp    2               // Remove args.
        ,               Push                    // Return address on stack.
        ,               Loco    1               // Load True.
        ,               Retn                    //
        ]
```

8.8.1 Example of generated assembly code

As example of assembly code generated by the compiler we list the code corresponding to the Fibonacci program from section 8.7.1. We used the obvious function to transform assembly statements to a string.

```
                lodd  v_x           // print  (x) ;
                call  print
    l_1:        lodd  v_x           // WHILE x
                jzer  l_2
                lodd  v_fibn-1      // DO   fibn-2 := fibn-1;
                stod  v_fibn-2
                lodd  v_fibn        //      fibn-1 := fibn;
                stod  v_fibn-1
                lodd  v_fibn-2      //      fibn := fibn-1 + fibn-2;
                push
                lodd  v_fibn-1
                addl  0
                insp  1
                stod  v_fibn
                loco  1             //      x := x - 1
                push
                lodd  v_x
                subl  0
                insp  1
                stod  v_x
                jump  l_1           // END;
    l_2:        lodd  v_fibn        // print (fibn)
                call  print
                Stop                //  End of program.
    print:      ...             //  Fixed code.

    v_x:        5                   // x := 5
    v_fibn:     1                   // fibn := 1
    v_fibn-1:   1                   // fibn-1 := 1
    v_fibn-2:   0                   // fibn-2  obtains the default value.
```

The execution of compiled Tiny programs can be traced in the same way as ordinary Mac-1 programs.

8.8.2 How to prove the correctness of the compiler

The correctness of the Tiny compiler presented here is an interesting point. Correctness appears to have several aspects. A first aspect of correctness is the syntactical correctness of the given Clean program. This can be checked by the Clean compiler. A second point of view to correctness of the compiler is whether or not the compiler assigns a meaning to all valid Tiny programs. The Clean system generates warnings (`function may fail`) when a function is not defined for some combination of its arguments. Although having no partial functions does not imply that the value of all Tiny programs is defined (functions can also fail to terminate), this is a great help.

The most interesting question is whether the meaning assigned to Tiny programs is the correct one or not. The Clean system cannot help us here, it has no knowledge about the semantics of Tiny whatsoever. Even human beings cannot answer this question as the semantics of Tiny programs simply has not been defined. Hence, it is impossible to de-

termine whether the compiler obeys such semantics. In order to state something about the correctness of the compiler for Tiny programs, we need another definition of the meaning of programs in the language Tiny. One of the possibilities to define the semantics of Tiny programs is by defining an interpreter for Tiny programs. Such an interpreter is just another implementation of Tiny. However, when we have two implementations we can reason about their equivalence. In addition we state that whatever the interpretator does *is* what all implementations of Tiny should do. In this way the interpreter *defines* the meaning of Tiny programs: the semantics. The next section contains a specification of the meaning of programs in Tiny by means of an interpreter.

8.9 Interpretation

An interpreter for Tiny programs is developed in this section. The main reason to introduce an interpreter for Tiny is that to show the difference between compilation and interpretation. Furthermore, it can be used to show the correctness of the compiler developed in the previous section.

The interpreter needs an interpreter state, `IState`, to store the declared variables and their current value. The initial state is created by `emptyIState`. A variable can be added to the state by `define`. The value of a variable can be obtained and changed by the functions `read` and `update`.

```
:: IState :== [(Variable,Int)]

emptyIState :: IState
emptyIState = []        // Defines the initial state. This state contains no variables.

read :: Variable IState -> Int
read v [(w,n):r]        // Obtain the value of the given variable from the IState.
 | v == w   = n
            = read v r
read v []   = abort ("read: The variable "+v+" is undefined")

update :: Variable Int IState -> IState
update v n [c=:(w,i):r] // Update the value of the given variable with the given integer in the IState.
 | v == w   = [(w,n):r]
            = let! r` = update v n r // Force evaluation of IState to limit space consumption.
              in [c:r`]
update v n [] = abort ("update: The variable "+v+" is unbound")

define :: Variable Int IState -> IState
define v i state = [(v,i):state]    // Add a variable to the IState.
```

Using this state we can define the actual interpreter. The function `interp` recursively descends the list of instructions. The state of the interpreter is passed as an argument to the recursive calls of the interpretator. Declarations and assignments change the state of the interpreter. A conditional changes the list of statements to execute. The semantics of a `While` loop is expressed in terms of the conditional, `If`.[1]

The result of the interpreter is the list of printed integers. Each `Print` statement produces one `Int`. The interpreter starts with an empty state. Since we assume that all programs in Tiny are correct, all variables will be declared before they are used. In the compiled code all variables are defined and initialized before the first instruction is executed, so it is in principle possible to allow the use of variables before their declaration is executed (see exercises 8.11 – 8.13). We state that all variables in a Tiny program must be de-

[1] This approach cannot be used in the compiler since it would produce an infinite list of statements. The compiler re-uses the code of one `While` loop by a jump back to the begin of the code of the loop.

clared before they are used. So, in this respect there is no difference between the interpreter and the compiler.

The actual interpreter takes the list of state ments and the state as arguments. The first statement is distinguised by pattern matching and the rest of the statements, `r`, is interpreted by a recursive call of the interpreter. The state passed to this recursive call is updated according to the meaning of the instruction. The interpreters yields the list of integers that are printed by the Tiny program. This interpreter is similar to a description of the language semantics, see e.g. [Gordon 79, Nielson 92].

```
interpreter :: Tiny -> [Int]
interpreter prog = interp prog emptyIState
  where interp [Declare v e:r] s = interp r (define v (eval e emptyIState) s)
        interp [Assign v e :r] s = interp r (update v (eval e s) s)
        interp [If c t e   :r] s | eval c s <> 0 = interp (t++r) s
                                 | otherwise     = interp (e++r) s
        interp [While c b  :r] s = interp [If c (b++[While c b:r]) r] s
        interp [Print e    :r] s = [eval e s: interp r s]
        interp []              s = []
```

We use the following evaluator for expressions. This evaluator needs the state as argument to lookup, `read`, the value associated with a variable. The evaluator is a straightforward decomposition of expressions. The semantics of Tiny constructs like addition is expressed in terms of the corresponding construct in Clean.

```
eval :: Expression IState -> Int
eval (Add  n m) s = eval n s +  eval m s
eval (Sub  n m) s = eval n s -  eval m s
eval (Eq   n m) s | eval n s == eval m s = 1
                  | otherwise            = 0
eval (Less n m) s | eval n s <  eval m s = 1
                  | otherwise            = 0
eval (CONST n)  s = n
eval TRUE       s = 1
eval FALSE      s = 0
eval (Var v)    s = read v s
```

8.10 Correctness

At this point we have two separate implementations for programs in the language Tiny. Each implementation assigns a semantics to the language. The first meaning is obtained by compiling a Tiny program to Mac-1 code and executing this code by the Mac-1 machine description. The second meaning of Tiny programs is given by the interpreter. We can prove the correctness of the compiler with respect to the semantics defined by the interpreter. We define correctness as: one language implementation is correct with respect to an other language implementation if both implementations produce the same output for all correct programs when the input is identical. Requiring correct programs guaranties among other things that all variables are defined before they are used.

In order to prove the correctness we want to show that the execution by Mac-1 of the code generated by the compiler yields the same output as the interpreter. Since there are infinite many different correct Tiny programs we cannot check all possible programs. The way to prove this is by induction to the structure of Tiny programs.

For this induction proof we show how the program state represent in Mac-1 can be mapped to the interpreter state of the corresponding program. This `retrieve` function is similar to the inverse of the translation function `t` used in the `assembler`. Its type is:

```
retrieve :: Tiny -> (State -> IState)
```

Now we have to show that this retrieve function maps the initial state of Mac-1, directly after booting the machine, is mapped to the initial state of the interpreter.

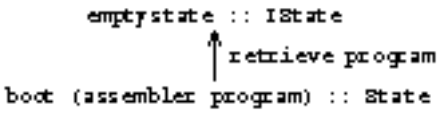

Figure 2. Correspondence of initial states.

Furthermore, we show for each instruction that the state change of Mac-1 caused by execution the sequence of instructions corresponding to a Tiny statement is equivalent to the state change of the interpreter. Moreover, the sequence of instructions executed by Mac-1 should produce exactly the same output as the interpreter executing the Tiny statement. For simulation and interpretation purposes it is most convenient to extract the output from the state as soon as possible. For a correctness proof however, it is more convenient to deliver a new state after each instruction that contains the output. Such an interpreter can be show to be equivalent to the interpreter presented here.

A prerequisite for the equivalence of statements is the equivalence of expressions in Tiny and the associated Mac-1 code. This can only be shown when we assume that all computations are bound to the numbers used in Mac-1. Under this assumption operators like `+`, `|+|` and `-`, `|-|` are equal.

For each `State` and `Statement` we have to show that the state change caused by:

```
interp [statement] (retrieve program state)
```

is equivalent to

```
seq (assembler (compiler [statement])) state
```

The statement and the associated instruction sequence should produce the same output.

The equivalence of a Tiny statement and the associated sequence of Mac-1 instructions can be depicted as:

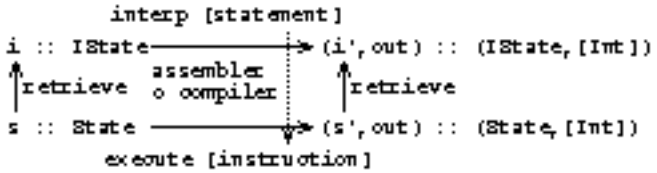

Figure 3. Correspondence of state changes

Finally, we have to show that the entire instruction sequence executed by Mac-1 corresponds to the sequence of statements executed by the interpreter. The correctness of the compiler for all valid programs in Tiny follows by the induction principle.[2]

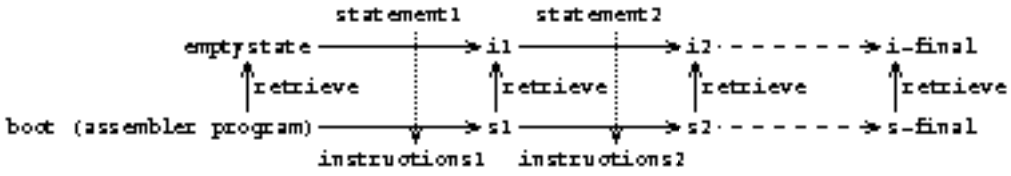

Figure 4. Sketch of the proof by induction of correctness of the compiler.

[2]More advanced compilers does not generate a fixed sequence of instructions for each statement in the source language. These compilers cannot be proven correct by the simple kind of induction proof shown here.

Although the principle of this proof is rather straight forward, the actual proof is rather elaborated and quite complicated. The actual proof is outside the scope of this book.

8.11 Summary

This chapters starts with the description at the conventional machine level of the very small computer architecture for Mac-1. The state of this machine is described by a set of memory components. This state can be changed by executing instructions. In the given description instructions are functions of type `state -> state`.

The conventional machine introduced can be programmed by loading a list of integers in the memory. Although this first generation programming language is fine for a machine, it is cumbersome and error-prone for human beings. The second generation of programming languages solves these problems by using symbolic names for the instructions and strings to label statements. This is illustrated by an assembly language and assembler for Mac-1.

Third generation programming languages are the next abstraction level introduced. These languages abstract from the details of a specific machine. Hence, a program in third generation language can be compiled to code for many different machines. In this chapter the concepts are shown by the introduction of the language Tiny and associated compiler.

To show the differences between a compiler and an interpreter we introduce an interpreter for Tiny programs. As illustrated by a some examples it is very easy to allow semantically differences between the compiled and interpreted versions of the language Tiny. This interpreter can also be used to reason about the correctness of the implementation of Tiny on Mac-1 by the compiler and assembler.

8.12 Exercises

1 Another possible representation of `Word` is an explicit sequence of bits. In the chosen representation the sequence of bits is implicit. Write a new implementation of the type `Word` and the associated manipulations where words are a sequence of elements of type `Bit`.

Redefine the memory as an array of words. Adjust the given three manipulation functions on the memory to the new representation.

2 Redefinition of these instructions using the array based memory developed in exercise 8.2 shows why we have chosen for the list representation of the memory. One should like that it is sufficient to replace the definition of functions manipulating the list-based memory with the equivalents for the array-based memory. Unfortunately, the clean system is not able to derive the required uniqueness information from these definition. Uniqueness of the memory components in the state is a very desirable property. In a real machine there is only one instance of each memory component available.

3 Adapt the definitions of the instructions such that the Clean system can derive the uniqueness information needed to use the array based memory.

4 Discuss the possibilities to add a multiplication instruction to the instruction set. Can this be done similar to the addition? Where should the arguments and the result be? What opcode should be used?

5 List the possible interpretations in Mac-1 of the words corresponding to the following bit strings:

```
0000 0000 0000 0000, 0000 0000 0000 0001, 1000 0000 0000 0000,
1111 0000 0000 0000, 1111 0000 0000 1111, 1111 1100 0000 1111,
0111 1111 1111 1111, 1111 1111 1111 1111, 1111 1111 1111 1110.
```

6 Describe the changes needed in the rest of the machine when the program counter is not incremented in the instruction cycle.

7 Instead of stopping with the execution of instructions, most computers generate an interrupt, or trap, when the current word appears to be not the representation of an instruction. That is, the program counter is set to some predefined value and the execution of instructions is continued from there.

Change the instruction cycle such that program execution is continued from address 1 when the word indicated by the pc is not a valid instruction.

8 Extend Mac-1 with memory mapped input. Reading the memory can have a side effect with this extension. This implies that the type of `|!|` and all its application should be changed.

9 Show how I/O can be modelled in Mac-1 by using the words in the original memory as interface with the world. This implies that the memory remains a list of words, the memory word representing the output status register should be checked instead of the separate `osr`.

10 `Declaration` statements are allowed neither inside conditional expressions, `If c t e`, and `While` loops, nor inside `expressions`. Explain why the compiler still needs to generate a list of labels and initial values for these language constructs in Tiny.

11 Extend the expressions (data type and compiler) of Tiny to the usual set (*, /, mod, >, >=, <=, ≠, AND, OR, NOT). Use distinct types for expressions of type Boolean and Number.

Hint: it is convenient to extend the `fixed_code` with functions similar to `Eq` and `Less` corresponding to the operators to be added.

12 The current compiler allows only expressions consisting of constants as initial value for variables. Extend the compiler such that expressions are allowed as initial value. These expressions should be evaluated when the declaration is "executed".

See the next exercise for the semantical consequences of this language extension.

13 After the language extension of the previous exercise it is not possible to use the initial value of a variable before its declaration is executed. Neither is the interpreter described in the next section able to use variables before their definition is executed. Show how we can force that all declarations are located at the start of a program by changing the data structure `Tiny` and indicated the changes needed in the compiler.

14 An other way to prevent the use of undeclared variables as described in the previous exercise, is to verify for each applied occurrence of a variable whether the corresponding declaration have been executed or not. Change the compiler such that an error message is generated during compilation when a variable will (or can) be used before it is declared. The same machinery can be used to prevent multiple declarations of a variable.

15 Extend Tiny and the compiler with a `Read Variable` statement that assigns the value read from the input port to the variable.

16 Change the I/O of Mac-1 and the associated fixed code of the compiler, such that input and output is treated as a list of ASCII characters instead of numbers.

17 Replace the fixed code in the compiled programs by generating the appropriate instructions at each place where they are needed.

18 (Hard) Extend Tiny by a function concept. In the simplest setting functions do not have local variables. The actual function arguments are evaluated expressions pushed on the stack. A special constructor must added to the abstract syntax tree `expression` as indicator of symbolic function arguments. The compiler should keep track of the position of function arguments on the stack and use a symbolic stack pointer.

19 In the current manipulations of the state of the interpreter, `IState`, the declaration of variables and updating their value are two separate manipulations. Change this such that assign-

ing a value to an undefined variable does not yield an error, but a state extended by the variable.

20 In the current interpreter (and also in the compiler) it is not possible to use the value of previously defined variables in the expression that assigns an initial value to a variable at the declaration. e.g.:

```
[Declare "x" (CONST 1)
,Declare "y" (Var "x")]
```

Change the interpreter to make this possible. Discuss what changes would be needed to allow the same language extension to the compiler.

21 Instead of implementing `IState` as a list of tuples, we can also use a more specification oriented version: `IState :== Variable -> Int`. Instead of using the function `read` to find the value of a variable, we can simply apply the memory to the variable. This kind of memory was used in the assembler to map labels to addresses. It is also used in [Gordon 79, Nielson 92]. Change the interpreter according to this type of `IState` and discuss the consequences. Hint: pay attention to the size of the representation of the `IState` after some updates.

22 Extend the Tiny interpreter to the language extensions described in the exercises 8.11 and 8.12.

23 In contrast with the interpreter, it is in compiled Tiny code possible to use a variable before its declaration has been executed as current instruction. Change the interpreter such that it is possible to use variables before their declaration has occurred in the sequence of instructions to execute. Make it possible to use the value of previously defined variables in the definition of the initial value of a variable.

24 (Hard) Extend the interpreter for Tiny by the extension proposed in the exercise 8.18.