École Polytechnique

CSC_52064 : Compilation

examen 2025 (X2022)

Jean-Christophe Filliâtre

17 mars 2025 — 14h00–17h00

The test lasts 3 hours. Handwritten or printed course notes are the only documents allowed. Most questions are independent, in the sense that it is not necessary to have answered the previous questions in order to deal with a question. On the other hand, questions can call on definitions or results introduced in previous questions. Unless explicitly stated otherwise, all answers must be justified.

Feel free to answer in French or English.

Figures 1–3 are grouped together at the end of the subject on page 12.

Suggestion: detach the last sheet.

Throughout this subject, we consider a small imperative language called WHILE, whose abstract syntax is given in figure 1. A program is limited to a single statement. A statement (noted s) is an assignment, a display with `print`, a conditional, a `while` loop or a block (a sequence of statements, possibly empty). A variable (noted x) contains an integer and assignments are limited to nonnegative constants and subtractions of variables. Here is an example:

```
{ a=0  b=1
  while n<>0 do { t=0  t=t-b  b=a-t  a=b-a  t=1  n=n-t }
  print a }
```

Semantics. We equip WHILE with a small-step operational semantics, taking the form of a binary relation noted $\to$ between two configurations. A configuration, noted $\langle E, s\rangle$, is an environment E and a statement s. An environment is a partial function from variables to integers. Its domain, that is the set of variables for which E is defined, is noted $\mathrm{dom}(E)$. The relation $\langle E, s\rangle \to \langle E', s'\rangle$ reads as follows: In the environment E, the statement s successfully executes one step of computation and reaches environment E' and statement s'.

Figure 2 defines the relation $\to$ with a set of inference rules. In these rules, notation $E \oplus \{x \mapsto n\}$ stands for the function E' defined by

$$\begin{array}{rcl} E'(x) & = & n, \\ E'(y) & = & E(y) \ \text{ for } y \in \mathrm{dom}(E) \text{ and } y \neq x. \end{array}$$

In particular, we have $\mathrm{dom}(E') = \mathrm{dom}(E) \cup \{x\}$. Note carefully how the rules defining $\to$ manipulate environments. If `fib` stands for the example program above, we have the following execution,

$$\langle \{\mathtt{n} \mapsto 10\}, \mathtt{fib}\rangle \to^\star \langle \{\mathtt{a} \mapsto 55, \mathtt{n} \mapsto 0, \mathtt{b} \mapsto 89, \mathtt{t} \mapsto 1, \}, \mathtt{\{\ \}}\rangle$$

(and the program printed 55).

A configuration C is said to be irreducible if there is no configuration C' such that $C \to C'$. We say that the execution of a configuration *blocks* if it reaches, after some execution steps, an irreducible configuration $\langle E, s\rangle$ with $s \neq \mathtt{\{\ \}}$. A trivial example of blocking configuration is $\langle \emptyset, \mathtt{print}\ x\rangle$ since the variable x is not defined in the empty environment. An execution that does not block terminates on the statement `{ }` or never terminates.

Question 1 Give the execution steps of the program

```
{ x=1 if x=0 then y=1 else z=2 print y }
```

in an empty initial environment.

Correction :

```
                {}, { x=1 if x=0 then y=1 else z=2 print y }
->       {x->1}, { { } if x=0 then y=1 else z=2 print y }
->       {x->1}, {     if x=0 then y=1 else z=2 print y }
->       {x->1}, { z=2 print y }
-> {x->1, z->2}, { { } print y }
-> {x->1, z->2}, { print y }
```

and the program blocks.

Question 2 Propose a program that, in an environment E defining at least the variables a and b, with $E(b) \geq 0$, prints the value of $E(a) \times E(b)$ and terminates. How can you relax the hypothesis $E(b) \geq 0$? (Do not give the code this time, but only the idea.)

Correction : We loop over the value of b, adding a at each step to some accumulator c.

```
c = 0      // accumulator
a = c - a  // -a rather than a
o = 1
while b <> 0 do {
  c = c - a
  b = b - o
}
print c
```

To relax the hypothesis $b \geq 0$, one solution consists in, when $b < 0$, replacing a with $-a$ and b with $-b$. To determine the sign of b, we can make a loop that successively explores all the natural integers and, for each one, compares it with b and $-b$ (a subtraction then a `if`). At the first equality, the loop is exited (by setting to 0 the variable tested by the loop).

Parsing. We wish to implement a parser for the language WHILE.

Question 3 Discuss the problem of the constant `0` which appears in the syntax of the constructions `if` and `while` but also in an assignment of the form x `= 0`. Show precisely how this can be taken into account in both lexical and syntactic analysis. Try to suggest two different solutions.

Correction : At least two solutions:

1. During lexical analysis, we construct two different tokens, one for `0` and another for literal constants other than `0`. (The priority rule in `lex`-like tools makes this easy, by recognizing the first before the second.) Next, we write two grammar rules for assignment in the parser:

   ```
   stmt:
   | IDENT EQUAL CONST { ... }
   | IDENT EQUAL ZERO  { ... }
   ...
   ```

2. During lexical analysis, we make no distinction, with a single token `CONST` for integer literals. In the parser, we then introduce a non terminal

   ```
   zero:
   | n=CONST { if n <> 0 then raise Parsing.Parse_error }
   ;
   ```

   whose purpose is to recognize only `0` and we use it in the grammar rules for `if` and `while`.

Question 4 Give a grammar for the language WHILE in the syntax of either CUP or Menhir (your choice). It must be able to parse a file containing a single statement. Declare the tokens and the start symbol. Declare associativity rules and priorities, if needed. *Do not include the semantic actions.*

Correction : We opt for the first solution in the previous question.

```
%token <string> IDENT
%token <int> CONST
%token IF THEN ELSE PRINT DO WHILE
%token ZERO EQUAL DIFF MINUS LBRA RBRA
%token EOF

%start file
%type <Ast.file> file

file:
| stmt EOF                                { ... }
;
stmt:
| IDENT EQUAL CONST                       { ... }
| IDENT EQUAL ZERO                        { ... }
```

```
| IDENT EQUAL IDENT MINUS IDENT            { ... }
| PRINT IDENT                              { ... }
| IF IDENT EQUAL ZERO THEN stmt ELSE stmt  { ... }
| WHILE IDENT DIFF ZERO DO stmt            { ... }
| LBRA stmt* RBRA                          { ... }
;
```

No need for priority or associativity rule here.

Static Analysis of Definitions and Uses. For a program not to block, it is sufficient that any variable used by a statement (to the right of an assignment or as an argument of a `print`, `if` or `while` statement) must be present in the environment at the time of execution. This variable may have been present in the environment since the start of execution, or it may have been introduced into the environment by a previously executed assignment. For instance, the program

```
{ if x = 0 then y = 1 else y = 2
  print y }
```

successfully executes in an initial environment that defines only `x`.

We propose to statically determine, for a statement s, a set of variables noted $use(s)$ sufficient for its correct execution, in the following sense: for any environment E such that $use(s) \subseteq \mathrm{dom}(E)$, the execution of $\langle E, s\rangle$ does not block. Of course, it would be sufficient to take for $use(s)$ all the variables that appear in s, but this would be a very crude solution. For the program above, for example, the set $\{\texttt{x}\}$ is sufficient.

In order to perform a more detailed analysis of $use(s)$, we are going to simultaneously compute a second set of variables, noted $def(s)$, corresponding to variables necessarily defined by the execution of s, in the following sense: for any execution $\langle E, s\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$, then we have $def(s) \subseteq \mathrm{dom}(E')$. For the program above, for example, we have $def(s) = \{\texttt{y}\}$ because every execution gives a value to the variable `y`.

We propose to compute the sets $def(s)$ and $use(s)$ by induction on the structure of s. Here is how to perform the calculation for the first three statements:

	$def(s)$	$use(s)$
x = n	$\{x\}$	$\emptyset$
x_1 = x_2-x_3	$\{x_1\}$	$\{x_2, x_3\}$
`print` x	$\emptyset$	$\{x\}$

Question 5 Complete the table above by proposing a computation of $def(s)$ and $use(s)$ for the remaining three statements of WHILE: conditional, loop, and block.

Correction :

	$def(s)$	$use(s)$
`if` x `= 0 then` s_1 `else` s_2	$def(s_1) \cap def(s_2)$	$\{x\} \cup use(s_1) \cup use(s_2)$
`while` x `<> 0 do` s	$\emptyset$	$\{x\} \cup use(s)$
`{ }`	$\emptyset$	$\emptyset$
`{` s_1 $s_2 \ldots$ `}`	$def(s_1) \cup def(\texttt{\{}\ s_2 \ldots\ \texttt{\}})$	$use(s_1) \cup (use(\texttt{\{}\ s_2 \ldots\ \texttt{\}}) \setminus def(s_1))$

Note that a `while` loop defines no variable, as we can't know, in the general case, if its body is going to be executed at least once.

Question 6 Show the correctness of your definition of $use(s)$. Make sure you clearly state and prove all the intermediate properties you need.

Correction : We start with a monotony lemma: if $\langle E, s\rangle \to^\star \langle E', s'\rangle$ then $\mathrm{dom}(E) \subseteq \mathrm{dom}(E')$. (The environment only grows.) Trivial induction over the number of steps.

We continue with a context lemma: we have $\langle E, s_1\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$ if and only if $\langle E, \texttt{\{}\, s_1\ s_2 \ldots\, \texttt{\}}\rangle \to^\star \langle E', \texttt{\{\{ \}}\ s_2 \ldots\, \texttt{\}}\rangle$. Trivial induction over the number of steps.

Last, we show by structural induction over evaluation the following property: If $use(s) \subseteq \mathrm{dom}(E)$ then the execution of $\langle E, s\rangle$ does not block, and for each execution $\langle E, s\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$ we have $def(s) \subseteq \mathrm{dom}(E')$.

- x `=` n : immediate
- x_1 `=` x_2`-`x_3 : immediate
- `print` x : immediate
- `if` x `= 0 then` s_1 `else` s_2 : We have $x \in use(s)$ and thus we can make a step, to $\langle E, s_1\rangle$ or $\langle E, s_2\rangle$. In both cases, the execution does not block by IH since $use(s_1), use(s_2) \subseteq use(s) \subseteq E$. If $E(x) = 0$ and $\langle E, s_1\rangle \to^\star \langle E', \texttt{\{ \}}\rangle$ then $def(s_1) \subseteq \mathrm{dom}(E')$ by IH, and $def(s) \subseteq def(s_1)$. Similarly if $E(x) \neq 0$.
- `while` x `<> 0 do` s_1 : We have $x \in use(s)$ and thus we can make a step. If $E(x) = 0$, this is immediate. Otherwise, the execution does not block by IH since $use(s_1) \subseteq use(s) \subseteq E$. Besides, $def(s) = \emptyset$ so there is nothing to prove for the second part.
- `{ }`: immediate
- `{` s_1 $s_2 \ldots$ `}` : We have $use(s_1) \subseteq use(s)$ and thus the execution of $\langle E, s_1\rangle$ does not block by IH. If it does not terminate, then so does the execution of s and it does not block. Otherwise, the context lemma tells us that $\langle E, \texttt{\{}\, s_1\ s_2 \ldots\, \texttt{\}}\rangle \to^\star \langle E_1, \texttt{\{\{ \}}\ s_2 \ldots\, \texttt{\}}\rangle \to \langle E_1, \texttt{\{}\, s_2 \ldots\, \texttt{\}}\rangle$ for some E_1. By IH we have $def(s_1) \subseteq \mathrm{dom}(E_1)$ and thus $use(\texttt{\{}\, s_2 \ldots\, \texttt{\}}) \subseteq \mathrm{dom}(E_1)$ using the definition of use and the monotony lemma. Therefore, the execution of `{` $s_2 \ldots$ `}` does not block. If it terminates, then $def(\texttt{\{}\, s_2 \ldots\, \texttt{\}}) \subseteq \mathrm{dom}(E')$ but also $def(s_1) \subseteq \mathrm{dom}(E')$ by monotony, hence the result.

Question 7 Is an algorithm (yours or any other) able to compute sets $def(s)$ and $use(s)$ that are minimal for inclusion? If yes, justify. If no, explain why.

Correction : Computing minimal sets would be equivalent to determine when a variable gets the value 0, which is not decidable in full generality. Our sets $def(s)$ and $use(s)$ are thus doomed to be approximations, as in the following trivial example

```
{ if x = 0 then x = 1 else {}
  while x <> 0 do { y = 2 x = 0 } }
```

where we do not determine that `y` is always defined. Of course, we could do a better job on this example, by propagating the value of `x` (or rather its nullity), but any computation ending on a non-null value for `x` can be used instead.

In a more formal way, computing a minimal *use* set for the program

```
{ ...p... print x }
```

where the program `p` does not use variable `x` amounts to solve the halting problem for `p`.

RTL Language. We are now considering an RTL-type language (for Register Transfer Language) whose abstract syntax is given in figure 3. An RTL program is a finite set of basic blocks. A basic block b is given by a label L, a sequence of instructions, and a jump. An instruction i is an assignment or display, as in the WHILE language. A jump j is either the termination of the program (`halt`), a conditional jump (`ifz` x L_1 L_2) or an unconditional jump (`goto` L). An RTL program is well-formed if

- any label mentioned by `ifz` or `goto` corresponds to a program block;
- there is no jump `goto` L to a block with only one predecessor (*i.e.* to a block other than the first block, whose only reference is this `goto` L jump).

Here is an example of a well-formed RTL program with four basic blocks:

```
L0: n=10  r=0  one=1  one=r-one  goto L1
L1: t=r-n  ifz t L2 L3
L2: halt
L3: r=r-one  print r  goto L1
```
(R_0)

Note that block `L1` has two predecessors.

An RTL program runs in an E environment (of the same type as for WHILE) and starts at its first block (`L0` in the example above). The execution of a block is the execution of its instructions, in sequence, with the same semantics as for WHILE. If the execution of an instruction blocks, the RTL program also blocks. Otherwise, we get a new environment E' and we make the jump. The `halt` jump terminates the RTL program. The conditional jump `ifz` x L_1 L_2 continues execution on block L_1 if $E'(x) = 0$ and on block L_2 otherwise. The unconditional jump `goto` L continues execution on block L. In the case of a jump, execution continues with the environment E'. For instance, the above program prints the integers from 1 to 10 and then terminates.

Question 8 What is the effect of the following program

```
L0: zero=0  one=1  one=zero-one  u=1  r=0  goto L1
L1: t=r-n  ifz t L6 L2
L2: v=u-zero  s=0  goto L3
L3: t=s-r  ifz t L5 L4
L4: t=zero-v  u=u-t  s=s-one  goto L3
L5: r=r-one  goto L1
L6: print u  halt
```

in an environment that defines a variable `n` with a nonnegative value? Provide a detailed justification.

Correction : This program computes and prints the factorial of `n`. Indeed, it corresponds to the following nested loops:

```
u=1
for r=0 to n-1 do  // invariant u = fact(r)
  v = u
  for s=0 to r-1 do  // invariant u = (s+1) * fact(r)
    u = u+v
print u  // thus with u = fact(n)
```

Note: this is the program from *Checking a Large Routine* (Turing, 1949).

Question 9 Propose an abstract syntax (in Java or OCaml) for RTL programs. Propose data structures and methods/functions (in Java or OCaml) to execute an RTL program from an empty environment. *You do not have to implement these methods/functions.*

Correction : Solution in OCaml. Here are types for the abstract syntax:

```
type var = string
type label = string
module M = Map.Make(String)

type instr =
| Iprint of var
| Icst of var * int
| Isub of var * var * var

type branch =
| Halt
| Ifz of var * label * label
| Goto of label

type block = { code: instr list; branch: branch; }
type rlt = { entry: label; blocks: block M.t; }
```

We have chosen to represent all the blocks by a dictionary, to facilitate execution. (Another solution would be to use integers $0, \ldots, n-1$ for the labels and an array directly for the array for all blocks).

For execution, the simplest solution is to use a hash table (module `H` below) to represent the environment:

```
let exec {entry; blocks} =
  let vars = H.create 16 in
  let var x = try H.find vars x
```

```
  with Not_found -> eprintf "unbound variable %s@." x; exit 1 in
let rec instr = function
  | Iprint x          -> printf "  %d@." (var x)
  | Icst (x, n)       -> H.replace vars x n
  | Isub (x1, x2, x3) -> H.replace vars x1 (var x2 - var x3) in
let rec exec l =
  let b = M.find l blocks in
  List.iter instr b.code;
  match b.branch with
  | Halt            -> ()
  | Goto l          -> exec l
  | Ifz (x, l1, l2) -> exec (if var x = 0 then l1 else l2) in
exec entry
```

Question 10 We wish to compute the set *use* of an RTL program, with the same meaning as for a WHILE program (*i.e.*, execution does not block on an environment that defines at least the variables of the set *use*). Propose an algorithm to compute this set.

Correction : We note that we can compute the sets *def* and *use* of a basic block once and for all, independently of the other blocks. This is done in exactly the same way as for the WHILE language.

Next, we can compute the live input and output variables of each block with a fixed point calculation, just as we did in class, with the equations

$$\begin{cases} in(\ell) &= use(\ell) \cup (out(\ell) \backslash def(\ell)) \\ out(\ell) &= \bigcup_{s \in succ(\ell)} in(s) \end{cases}$$

and for instance Kildall's algorithm. The set we look for is thus in for the first block.

Question 11 Propose an algorithm to compile the WHILE language to the RTL language. Make sure that the resulting RTL program is well-formed.

Correction : Let's write the compilation in the form of a function $C(s, j)$ which takes as parameters an instruction s from the WHILE language and a branch (to be performed at the end), and returns the label of the block corresponding to this calculation. To answer the question, we simply compute $C(s, \texttt{halt})$ and we consider the output as the first block.

We give ourselves an auxiliary function $B(i_1 \dots i_n\ j)$ which constructs the block $L : i_1 \dots i_n\ j$ for a fresh label L and returns it, when $n > 0$ or $j \neq \texttt{goto}$, and returns L directly when $n = 0$ and $j = \texttt{goto}\ L$.

To compute $C(s, j)$, we start by collecting the sequence $i_1 \dots i_n$ of all elementary instructions at the beginning of s, up to the first `if` or `while`. If there are none, then s is of the form

$$\{\, i_1 \dots i_n \,\}$$

and we simply return the block $B(i_1 \dots i_n\ j)$.

If s is of the form

$$\texttt{\{}\ i_1 \dots i_n\ \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ s_3 \dots\ \texttt{\}}$$

we set $L = C(\texttt{\{}\ s_3 \dots\ \texttt{\}}, j)$, then $L_1 = C(s_1, \texttt{goto}\ L)$ and $L_2 = C(s_2, \texttt{goto}\ L)$, and we return $B(i_1 \dots i_n, \texttt{ifz}\ x\ L_1\ L_2)$. (We have two `goto`s to L.)

Last, if s is of the form

$$\texttt{\{}\ i_1 \dots i_n\ \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s_1\ s_2 \dots\ \texttt{\}}$$

we set $L = C(\texttt{\{}\ s_2 \dots\ \texttt{\}}, j)$, then we choose a fresh label L_t and we set $L_b = C(s_1, \texttt{goto}\ L_t)$, we declare the block L_t : $\texttt{ifz}\ x\ L\ L_b$, and finally we return $B(i_1 \dots i_n, \texttt{goto}\ L_t)$. (We have two `goto`s to L_t.)

Compiling to x86-64 Assembly. We propose to compile the RTL language to x86-64 assembly. (A cheat sheet is given in the appendix). We assume that the integers in our language are limited to signed 64-bit integers. We assume that an assembly function `print` is given, obeying the calling conventions (its argument is in `%rdi` and it potentially overwrites any *caller-saved* register) but not requiring stack alignment when called.

As our RTL language is very simple, we can perform register allocation by graph coloring on an interference graph built directly from the RTL code. For the next two questions, we take the following RTL program as an example:

```
A: zero=0  s=0  one=1  n=10  goto B
B: t=zero-n  s=s-t  n=n-one  ifz n H B                    (R1)
H: print s  halt
```

Question 12 Give the interference graph for program (R_1). Try to suggest one or more preference edges (with justification).

Correction : We have five variables (`n`, `s`, `zero`, `one` and `t`) that all interfere in pairs and therefore form a clique. It's a good idea to add a preference edge between `s` and `%rdi` (`print`'s parameter).

Question 13 Propose an assembly code for program (R_1).

Correction : The simplest solution is to use *caller-saved* registers, since instruction `print` is used only at the very end of the program.

```
##    s %rdi (because we print it)
##  one %rsi
## zero %rdx
##    n %rcx
##    t %r8
```

```
main:   mov $10, %rcx
        mov $0, %rdi
        mov $1, %rsi
        mov $0, %rdx
B:      mov %rdx, %r8
        subq %rcx, %r8
        subq %r8, %rdi
        subq %rsi, %rcx
        jnz  B
H:      call print
        xorq %rax, %rax
        ret
```

Question 14 Propose an assembly code for program (R_0) on page 6.

Correction : This time, on the contrary, it's better to use *callee-saved* registers for the variables `n`, `r` and `one`. The variable `t`, on the other hand, can remain in a *caller-saved* register.

```
        ##    n %rbx
        ##    r %r12
        ##  one %r13
        ##    t %rcx
main:   pushq %rbx
        pushq %r12
        pushq %r13
        mov $10, %rbx
        mov $0, %r12
        mov $1, %r13
        subq %r12, %r13         # one = r - one
        negq %r13               # see later
L1:     mov %r12, %r8
        subq %rbx, %r8
        jz  L2
L3:     subq %r13, %r12
        movq %r12, %rdi
        call print
        jmp L1
L2:     popq %r13
        popq %r12
        popq %rbx
        xorq %rax, %rax
        ret
```

(Of course, this could be simplified a bit, but we compile this program in a mechanical way, following the answer of question 16 below.)

Question 15 Generally speaking, what are the preference edges we can derive from an RTL program?

Correction : There are two places where a preference can be expressed:

- in an $x = y - z$ instruction, between the variables x and y. Indeed, the subtraction $x = x - z$ can then be compiled directly by an assembly instruction `sub`.
- in an `print` x instruction, between the variable x and the register `%rdi` (as we did above).

Question 16 Propose a general compilation scheme for the RTL language, *i.e.*, explain how each of its constructs (instruction or jump) can be compiled to x86-64 assembly.

Correction : We perform a linearization as seen in class. A register is reserved as temporary (e.g. `%r10`). At the start of the program, we save the *callee-saved* registers to be used.

$x = n$ **:** This is a `mov`.

$x = y - z$ **:** Since x86-64 subtraction has two operands, we make three cases:

- $x = x - z$ **:** This is the best case, which translates to a single `sub`.
- $x = y - z$ **with** $y \neq x$ **and** $z \neq x$**:** We copy y to x then we subtract z.
- $x = y - x$ **:** We subtract y from x, then we do a `neg` on x. (This is the `one = r - one` in the answer to the question 14.)

Of course, in all cases, you have to take into account the possibility that several variables are spilled into memory. In this case, a temporary must be used.

`print` x **:** Copy x into `%rdi`, if necessary, and then do `call print`.

`ifz` x L_1 L_2 **:** If the jump is immediately preceded by (in the same basic block) a subtraction $x = y - z$, then we can directly make a conditional jump (`jz` or `jnz` as desired). Otherwise, you must test x explicitly (with `test` x, x) before the jump.

`goto` L**:** This is a direct `jmp` instruction.

`halt:` Restore the *callee-saved*, if necessary, set `%rax` to zero then end the program with `ret`.

$$
\begin{array}{rcll}
s & ::= & x\texttt{=}n & \textit{constant } n \in \mathbb{N} \\
& | & x\texttt{=}x\texttt{-}x & \textit{subtraction} \\
& | & \texttt{print}\ x & \textit{display} \\
& | & \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s\ \texttt{else}\ s & \textit{conditional} \\
& | & \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s & \textit{loop} \\
& | & \texttt{\{}\, s \ldots s\, \texttt{\}} & \textit{block}
\end{array}
$$

Figure 1: Abstract Syntax of WHILE.

$$
\frac{}{\langle E, x\texttt{=}n\rangle \to \langle E \oplus \{x \mapsto n\}, \texttt{\{ \}}\rangle}
\qquad
\frac{x_2 \in \mathrm{dom}(E) \quad x_3 \in \mathrm{dom}(E)}{\langle E, x_1\texttt{=}x_2\texttt{-}x_3\rangle \to \langle E \oplus \{x_1 \mapsto E(x_2) - E(x_3)\}, \texttt{\{ \}}\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E)}{\langle E, \texttt{print}\ x\rangle \to \langle E, \texttt{\{ \}}\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E) \quad E(x) = 0}{\langle E, \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \to \langle E, s_1\rangle}
\qquad
\frac{x \in \mathrm{dom}(E) \quad E(x) \neq 0}{\langle E, \texttt{if}\ x\ \texttt{=}\ \texttt{0}\ \texttt{then}\ s_1\ \texttt{else}\ s_2\rangle \to \langle E, s_2\rangle}
$$

$$
\frac{x \in \mathrm{dom}(E) \quad E(x) = 0}{\langle E, \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\rangle \to \langle E, \texttt{\{ \}}\rangle}
\qquad
\frac{x \in \mathrm{dom}(E) \quad E(x) \neq 0}{\langle E, \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\rangle \to \langle E, \texttt{\{}\, s\ \texttt{while}\ x\ \texttt{<>}\ \texttt{0}\ \texttt{do}\ s\, \texttt{\}}\rangle}
$$

$$
\frac{}{\langle E, \texttt{\{\{ \}}\ s \ldots \texttt{\}}\rangle \to \langle E, \texttt{\{}\, s \ldots \texttt{\}}\rangle}
\qquad
\frac{\langle E, s_1\rangle \to \langle E_1, s_1'\rangle}{\langle E, \texttt{\{}\, s_1\ s_2 \ldots \texttt{\}}\rangle \to \langle E_1, \texttt{\{}\, s_1'\ s_2 \ldots \texttt{\}}\rangle}
$$

Figure 2: Operational Semantics of WHILE.

$$
\begin{array}{rcll}
p & ::= & b \ldots b & \textit{RTL program} \\
b & ::= & L : i \ldots i\ j & \textit{basic block} \\
i & ::= & x\texttt{=}n & \textit{constant } n \in \mathbb{N} \\
& | & x\texttt{=}x\texttt{-}x & \textit{subtraction} \\
& | & \texttt{print}\ x & \textit{display} \\
j & ::= & \texttt{halt} & \textit{end of program} \\
& | & \texttt{ifz}\ x\ L\ L & \textit{conditional jump} \\
& | & \texttt{goto}\ L & \textit{unconditional jump}
\end{array}
$$

Figure 3: RTL Language.

Appendix: x86-64 cheat sheet

A fragment of the x86-64 instruction set is given here. You are free to use any other part of the x86-64 assembler. In the following, r_i designates a register, n an integer constant and L a label.

`mov` r_2`,` r_1	copies register r_2 into register r_1
`mov` `$`n`,` r_1	loads constant n into register r_1
`mov` `$`L`,` r_1	loads the address of label L into register r_1
`sub` r_2`,` r_1	computes $r_1 - r_2$ and stores it into r_1
`neg` r_1	computes $-r_1$ and stores it into r_1
`mov` n`(`r_2`),` r_1	loads r_1 with the value contained in memory at address $r_2 + n$
`mov` r_1`,` n`(`r_2`)`	writes in memory at address $r_2 + n$ the value of r_1
`push` r_1	pushes the value of r_1 on the stack
`pop` r_1	pops a value from the stack and stores it into register r_1
`test` r_2`,` r_1	sets the flags according to the value of r_1 AND r_2
`jz` L	jumps to address L if flags signal a zero value
`jmp` L	jumps to address L
`call` L	pushes the return address to the stack and jumps to address L
`ret`	pops an address from the stack and jumps there

Calling conventions:

- up to six arguments are passed via registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, `%r9`;
- other arguments are passed on the stack, if any;
- the returned value is put in `%rax`;
- registers `%rsp`, `%rbp`, `%rbx`, `%r12`, `%r13`, `%14` and `%r15` are *callee-saved*: they won't be clobbered by a `call`;
- the other registers are *caller-saved*: they may be clobbered by a `call`;
- `%rsp` is the stack pointer, `%rbp` the *frame pointer*.