# Rapport

Made using the LaTeX document provided on the course page.

## PRACTICAL INFORMATION

Certain design decisions are commented in the code. To execute the main file, run the command `stack run -- "-p/-i/" FILEPATH "(`$e_1, e_2 \ldots e_n$`)"`, where each $e_i$ is an expression. The first argument is optional; `"-p"` to parse a file, `"-i"` to interpret it, or omit it to parse and interpret it. There are several program examples under the `programs` folder, most of which have accompanying hand-made abstract syntax trees. To run the provided tests, type the command `stack test`. These test both the parser and interpreter individually, as well as the parser and interpreter combined.

## QUESTION 1

The `return` statement is supposed to halt any further execution of a procedure call, evaluating to the value returned. As such, statements containing sub-statements, notably `while` and sequenced statements, need to somehow propagate the returned value down to the root of a derivation. The current semantics do not account for this, which results in the value of an evaluation of a statement as `return` $e_1$ `;` $e_2$ to the evaluation of $e_2$, and a terminating `while` statement to `null`.

I see two approaches that deal with this problem. First off, we can change the judgement of statements to the form $\langle \pi, \sigma, s \rangle \downarrow \langle v, \sigma, \delta \rangle$, where $\delta \in \{\top, \bot\}$. When a statement evaluates to $\langle v, \sigma, \top \rangle$, we know a return statement has evaluated to v at some point in the derivation. We can tweak the rules accordingly, such as for `while` and sequenced statements:

$$\text{Return}: \frac{\langle \pi, \sigma, e \rangle \downarrow \langle v \rangle}{\langle \pi, \sigma, \texttt{return}\ e \rangle \downarrow \langle v, \sigma, \top \rangle} \qquad \text{Sequence-}\top: \frac{\langle \pi, \sigma, s_1 \rangle \downarrow \langle v, \sigma_1, \top \rangle}{\langle \pi, \sigma, s_1\ \texttt{;}\ s_2 \rangle \downarrow \langle v, \sigma_1, \top \rangle}$$

$$\text{Sequence-}\bot: \frac{\langle \pi, \sigma, s_1 \rangle \downarrow \langle v_1, \sigma_1, \bot \rangle \quad \langle \pi, \sigma_1, s_2 \rangle \downarrow \langle v_2, \sigma_2, \delta \rangle}{\langle \pi, \sigma, s_1\ \texttt{;}\ s_2 \rangle \downarrow \langle v_2, \sigma_2, \delta \rangle}$$

$$\text{While-True-}\top: \frac{\langle \pi, \sigma_1, e \rangle \downarrow \langle \texttt{true} \rangle \quad \langle \pi, \sigma_1, s \rangle \downarrow \langle v, \sigma_2, \top \rangle}{\langle \pi, \sigma_1, \texttt{while}\ e\ s \rangle \downarrow \langle v, \sigma_2, \top \rangle}$$

$$\text{While-True-}\bot: \frac{\langle \pi, \sigma_1, e \rangle \downarrow \langle \texttt{true} \rangle \quad \langle \pi, \sigma_1, s \rangle \downarrow \langle v_1, \sigma_2, \bot \rangle \quad \langle \pi, \sigma_2, \texttt{while}\ e\ s \rangle \downarrow \langle v_2, \sigma_3, \delta \rangle}{\langle \pi, \sigma_1, \texttt{while}\ e\ s \rangle \downarrow \langle v_2, \sigma_3, \delta \rangle}$$

The other rules not containing sub-statements as premises (*Skip*, *Expression*, *AssignNew* and *AssignOld*) evaluates to $\langle v, \sigma, \bot \rangle$, and the other rules, such as for `if`-statements, remain unchanged (aside from propagating $\delta$). To show these rules are sound, we can show that if $\langle \pi, \sigma, \texttt{return}\ e \rangle \downarrow \langle v, \sigma, \top \rangle$ appears on a branch of some statement s's derivation before a *Call* rule is used, then $\langle \pi, \sigma, s \rangle \downarrow \langle v, \sigma, \top \rangle$ (using the *Call* rule may give rise to multiple applications of the *Return* rule, but we want to show that `return` statements in the context of the current procedure call work as intended). This can be done with proof by induction on derivations. We can also prove the completeness, by showing that a procedure body with a reachable `return` statement will eventually apply the *Return*-rule. This can be done by induction on statements.

The other approach, and the one closest to my implementation, leaves the judgement untouched, but involves more rules (I say closest to my implementation, as these semantics were made after making the interpreter, so they don't reflect each other completely). The idea is to let the rightmost derivation imitate a callstack. When evaluating a sequenced statement, we have a rule for each possible case for the first statement which might contain a `return` statement in it's derivation. Underneath are a few examples:

$$\text{Sequence-Return}: \frac{\langle \pi, \sigma, \text{e} \rangle \downarrow \langle \text{v} \rangle}{\langle \pi, \sigma, \texttt{return e ; s} \rangle \downarrow \langle \sigma, \text{v} \rangle}$$

$$\text{Sequence-If-True}: \frac{\langle \pi, \sigma_1, \text{e} \rangle \downarrow \langle \texttt{true} \rangle \quad \langle \pi, \sigma_1, \text{s}_1 \texttt{ ; } \text{s}_3 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}{\langle \pi, \sigma_1, \texttt{if e then } \text{s}_1 \texttt{ else } \text{s}_2 \texttt{ ; } \text{s}_3 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}$$

$$\text{Sequence-If-False}: \frac{\langle \pi, \sigma_1, \text{e} \rangle \downarrow \langle \texttt{false} \rangle \quad \langle \pi, \sigma_1, \text{s}_2 \texttt{ ; } \text{s}_3 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}{\langle \pi, \sigma_1, \texttt{if e then } \text{s}_1 \texttt{ else } \text{s}_2 \texttt{ ; } \text{s}_3 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}$$

$$\text{Sequence-While-True}: \frac{\langle \pi, \sigma, \text{e} \rangle \downarrow \langle \texttt{true} \rangle \quad \langle \pi, \sigma, \text{s}_1 \texttt{ ; while e } \text{s}_1 \texttt{ ; } \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_1 \rangle}{\langle \pi, \sigma, \texttt{while e } \text{s}_1 \texttt{ ; } \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_1 \rangle}$$

$$\text{Sequence-While-False}: \frac{\langle \pi, \sigma, \text{e} \rangle \downarrow \langle \texttt{false} \rangle \quad \langle \pi, \sigma, \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_1 \rangle}{\langle \pi, \sigma, \texttt{while e } \text{s}_1 \texttt{ ; } \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_1 \rangle}$$

By placing each statement on the "callstack", we guarantee that the *Sequence-Return* rule will be used for a statement `return e ; s`. In addition to the new sequence rules, we have a rule for each statement individually behaving in a similar manner, as not all statements are sequenced:

$$\text{If-True}: \frac{\langle \pi, \sigma_1, \text{e} \rangle \downarrow \langle \texttt{true} \rangle \quad \langle \pi, \sigma_1, \text{s}_1 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}{\langle \pi, \sigma_1, \texttt{if e then } \text{s}_1 \texttt{ else } \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}$$

$$\text{If-False}: \frac{\langle \pi, \sigma_1, \text{e} \rangle \downarrow \langle \texttt{false} \rangle \quad \langle \pi, \sigma_1, \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}{\langle \pi, \sigma_1, \texttt{if e then } \text{s}_1 \texttt{ else } \text{s}_2 \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}$$

$$\text{While-True}: \frac{\langle \pi, \sigma_1, \text{e} \rangle \downarrow \langle \texttt{true} \rangle \quad \langle \pi, \sigma_1, \text{s} \texttt{ ; while e s} \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}{\langle \pi, \sigma, \texttt{while e s} \rangle \downarrow \langle \text{v}, \sigma_2 \rangle}$$

$$\text{While-False}: \frac{\langle \pi, \sigma, \text{e} \rangle \downarrow \langle \texttt{false} \rangle}{\langle \pi, \sigma, \texttt{while e } \text{s}_1 \rangle \downarrow \langle \texttt{null}, \sigma \rangle}$$

The *While-True* rule ensures that the *Sequence-Return* rule will be applied if $\text{s} = \texttt{return e}$, as does *Sequence-While-True*. Again, we can use induction to show soundness and completeness.

One thing to note, is that I've decided to keep the implicit return (such as the procedure body `true ; false` evaluating to `false`) intact in the semantics (and interpreter) alongside the new return semantics. It would be clearer for procedures to require a `return` statement to evaluate to some value, and simply let any other statement evaluate to `null` (i.e. procedure calls evaluate to `null` if nothing is explicitly returned. This is what I had in my implementation at first), but I think it is interesting to keep both.

## QUESTION 2

The judgement form for typing expressions can be $\pi, \Gamma \vdash \text{e} : \tau$. Here, $\pi$ are the procedures (or source-code), $\Gamma$ the typing context, a mapping from variables to types, e an expression, and $\tau$ a type.

$\boxed{\pi, \Gamma \vdash \text{e} : \tau}$

$$\text{T-Call}: \frac{\pi, \Gamma \vdash \text{e}_i : \tau_i \quad \pi, [x_i \mapsto \tau_i] \vdash \text{s} : \tau}{\pi[\tau f(\tau_i, x_i)\{\text{s}\}], \Gamma \vdash f\,(\text{e}_i) : \tau} \qquad \text{T-Eq}: \frac{\pi, \Gamma \vdash \text{e}_1 : \tau \quad \pi, \Gamma \vdash \text{e}_2 : \tau}{\pi, \Gamma \vdash \text{e}_1 == \text{e}_2 : \texttt{boolean}}$$

$$\text{T-ArithOp}: \frac{\pi, \Gamma \vdash \text{e}_1 : \texttt{integer} \quad \pi, \Gamma \vdash \text{e}_2 : \texttt{integer}}{\pi, \Gamma \vdash \text{e}_1 \circ \text{e}_2 : \texttt{integer}} (\circ \in \{+, -, *, /\})$$

$$\text{T-NumCompOp}: \frac{\pi, \Gamma \vdash \text{e}_1 : \texttt{integer} \quad \pi, \Gamma \vdash \text{e}_2 : \texttt{integer}}{\pi, \Gamma \vdash \text{e}_1 < \text{e}_2 : \texttt{boolean}}$$

$$\text{T-BoolOp}: \frac{\pi, \Gamma \vdash \text{e}_1 : \texttt{boolean} \quad \pi, \Gamma \vdash \text{e}_2 : \texttt{boolean}}{\pi, \Gamma \vdash \text{e}_1 \circ \text{e}_2 : \texttt{boolean}} (\circ \in \{\&\&, ||\})$$

The rule *T-Call* states the type of a call is the type it is declared with. The premises ensure that each argument types to the type of the associated parameter, and the body of the procedure also types to the

type the procedure is typed with. It is given a fresh context where each variable is typed according to the parameters.

There are four rules for operations; *T-Eq*, *T-ArithOp*, *T-NumCompOp* and *T-BoolOp*. Each rule type the operands differently in the premises. The *T-Eq* rule specifies that == works on any operands, as long as they share the same type. This includes `null`. The result is a `boolean`. *T-ArithOp* collects the typing for any arithmetic operations into one rule. It states that given any two integers, applying one of the operations +, -, *, / will result in another operation (except for dividing by zero). The rule *T-NumCompOp* works in the same way as *T-ArithOp*, only resulting in a boolean, and applying to operations which compare numbers (only $<$ right now, but operations with $\leq$, $\geq$ etc. would also be typed here). *T-BoolOp* is similar, only for boolean operators.

## QUESTION 3

### Assumptions and design

Objective B as presented contains many unaddressed problems, such as precedence, associativity and other ambiguities. As such, I have made several assumptions and/or decisions.

Firstly, precedence of operators is undefined. This leaves expressions such as `1+2*3` and `p == q && s || t` ambiguous. To remedy this, I have defined the following precedence, from loosest to tightest grouping, taking inspiration from the `Boa` language:

1. The negation operator `!`.
2. The boolean operators `&&` and `||`. These are left-associative.
3. The relational operators < and ==. These are, as in `Boa`, non-associative.
4. Additive arithmetic operators (+ and -), which are also left-associative.
5. Multiplicative arithmetic operators (* and /), also left-associative.

Although not an operator in the traditional sense, the sequential separators between statements (;) binds the loosest when viewed as one. These bind to the left, meaning a statement such as `while` e $s_1$ `;` $s_2$ will be parsed as (`while` e $s_1$) `;` $s_2$. More generally, the separators are parsed as one sequence within it's respective braces. For a sub-statement of another to be a sequence, it must be contained within braces. An example is found in the file `gcdsum.objb`, where the statement `if ... then {sum=sum+grcmdv; print(n); print(grcmdv)} else ...` occurs. This statement contains a sequence of three sub-statements in it's `then`-branch. Without these braces, the program won't parse, as the parser tries to parse the `if`-statement as a sequence of three statements `if ... then sum=sum+grcmdv`, `print(n)` and `print(grcmdv) else ...` and fails.

On the topic of braces, I've also implemented local scopes into the interpreter. While the semantics do not account for this, I thought it would be interesting to implement (and the assignment text encouraged new ideas). Every sequence of statements within a brace has it's own "frame". Every scope can mutate and read variables from the previous scopes, but variables declared in the current scope will do so in the current frame, which is lost when finished interpreting the braced sequence. Variables can be redeclared as long as it is not a part of the current frame. Otherwise, redeclaration is illegal. An example can be found in the file `programs/simple/braces/returnbraced.objb` with the following code:

```
integer main () {
  integer i=0;
  {
    integer x = 1;
    {
      integer i = 1;
      x = x + 1;
      return i + x
    }
  };
  return i
}
```

This program returns 3; the first definition of `i` is overwritten by redeclaring the variable in the innermost scope, and `x` is mutated and read even though it is not a part of the current scope. One way scopes could be represented in the semantics, would be to let the environment $\sigma$ be a stack of mappings. The *Variable* rule could be replaced with the two new rules *Variable-1* and *Variable-2*:

$$\text{Variable-1}: \frac{}{\langle \pi, \sigma[x \mapsto v] : \Delta, x \rangle \downarrow \langle v \rangle} \qquad \text{Variable-2}: \frac{\langle \pi, \Delta, x \rangle \downarrow \langle v \rangle}{\langle \pi, \sigma : \Delta, x \rangle \downarrow \langle v \rangle} (x \mapsto \_) \notin \sigma$$

$\Delta$ here would be the stack of mappings. Alongside other rules, such as *AssignNew*/*AssignOld*, needing changes, there would also be a need for new *Brace* rules, which should extend the environment with a new frame, and evaluate the statement inside in the context of the new state. Underneath is an example rule:

$$\text{Sequence-Brace}: \frac{\langle \pi, [\,] : \Delta, \mathrm{s}_1 \rangle \downarrow \langle \mathrm{v}_1, \sigma : \Delta' \rangle \quad \langle \pi, \Delta', \mathrm{s}_2 \rangle \downarrow \langle \mathrm{v}_2, \Delta'' \rangle}{\langle \pi, \Delta, \{\mathrm{s}_1\} \texttt{ ; } \mathrm{s}_2 \rangle \downarrow \langle \mathrm{v}_2, \Delta'' \rangle}$$

Another aspect not formally discussed is the role of whitespaces in Objective B. As such, I treat whitespaces similarly as in Boa; in some cases it serves as a separator and is mandatory, while in other it is not. Specifically, a whitespace is needed after the following keywords: `while`,`if`, `then`, `else` (only in the case the `else`- branch contains a statement other than $\varepsilon$), `return` and any type. Otherwise, whitespace does not serve a role (see the simple example `simplecall1weirdformat.objb`).

Identifiers are the same as Boa; any sequence of strings that begin with a letter and continues with either letters, numbers or underscores.

Whereas the types `integer` and `boolean` are understandable, it is not stated what `void` is. Intuitively, I interpret it as the type of a procedure which returns nothing. I've decided that this "nothing" is `null`, meaning a procedure `void f(...) ...` should return `null`. Since `void` is a type as other types in the semantics, the declaration `void x = null` is legal, albeit counter-intuitive. In fact, this is the only way to bind `null` to some variable, as I've interpreted the types `integer` and `boolean` as number and truth value, both of which are not "nothing". The formal semantics do not require that a variable annotated with some type actually maps to a value of that type, nor a procedure evaluate to a value of the type of the procedure, so that is another, albeit fair, assumption in my implementation.

### Completeness and correctness

I have not done any formal proofs for either the semantics nor the implementation. As such, we cannot state anything about the completeness or correctness with full certainty. I have tried to implement all of the semantics in the interpreter, and do not think there is anything missing. There are 30+ example programs, ranging from simple expression evaluations, to full programs. These are used in over 80 tests, where I compare parsed programs to hand-made abstract syntax trees, compare the output of interpreted hand-made AST's to hand-made output, and compare the output of interpreted parsed programs to hand-made output. While not completely exhaustive, I've tried to make varied test programs which usually test more than a single aspect, such as `programs/simple/branching.if` both testing whether an `if` statement works as expected, but also that a `then` branch can be empty. There are also tests under the `programs/error` folder which should evaluate to errors. Given that all the tests pass, I'm rather confident in my implementation.

### Efficiency

The parser uses `try` from the `Parsec` library extensively. This allows the parser to backtrack it's parsing in case it fails due to statements or expressions sharing a prefix. This usage results in understandable and maintainable code, but also in a loss in efficiency, as there might be several nested branches the parser tries before either failing or succeeding. As far as I understand, one could use left-factorization on the grammar of the language to avoid the usage of `try`'s. This, however, is not trivial, and makes for a less maintainable and understandable code.

The interpreter contains a lot of functions which must linearly go through the whole variable environment, to for example look up variables, or mutate some variable. These are the less efficient parts of the interpreter, and a result of using plain lists to represent the environments. This could be improved by utilizing another data structure, such as a binary tree or hash map. Another improvement could be made if a type checker was implemented. The interpreter then would not need to constantly check types when working with variables or return values.

### Weaknesses

There are some weaknesses with my implementation. First off, the usage of ;, the sequence statement, can be rather unconventional and unclear. The `return` statements `return` e and `return` e `;` $\varepsilon$ are practically equivalent. When taking implicit "return" into account however, procedures ending in e and e `;` $\varepsilon$ will not evaluate to the same value. It is also necessary to have a ; after braces, such as in the statement `while` e $\{s_1\}$ `;` $s_2$ which also breaks with conventions.

The local scopes, which should not have been implemented to better reflect the semantics, can also result in unexpected behaviour. It is possible to declare variables in a braced `while` loop's body. Take `while` e $\{\tau\ x = v\}$ as an example. For each loop, we'd define `x` anew with no problem, as at the end of each loop, we'd exit out of the scope and lose `x`. However, the statement `while` e $\tau\ x = v$ would result in an error if the body is repeated more than once. An `if` statement followed by some other statement like `if` e `then` $\tau_1\ x = v_1$ `else` $\tau_2\ x = v_2$ `;` $x$ executes without error. `if` e `then` $\{\tau_1\ x = v_1\}$ `else` $\{\tau_2\ x = v_2\}$ `;` $x$ however, will always fail, as when exiting from the local scope, we lose `x`.

## CHALLENGE 1

For printed expressions to be reflected in the semantics, we can extend the judgement for both expressions and statements to include a sequence of values $\theta$. I won't formulate a formal definition, but let $\theta \circ v$ be the sequence $\theta$ of values ending with v, and $\theta_1 \circ \theta_2$ the sequence $\theta_1$ extended with $\theta_2$. A sequence $\theta_1 \circ \theta_2 = \theta_1$ if $\theta_2$ is the empty sequence, and vice versa (for some notion of the empty sequence). All rules simply propagate this $\theta$ except for the new rule *Print*. To print a value, we simply evaluate the expression, which yields some value v and output $\theta$, and add v to the end of $\theta$. The value of the `print` expression itself is always `null`.

$\boxed{\langle \pi, \sigma, e \rangle \downarrow \langle v, \theta \rangle}$

$$\text{Print}: \frac{\langle \pi, \sigma, e \rangle \downarrow \langle v, \theta \rangle}{\langle \pi, \sigma, \texttt{print (e)} \rangle \downarrow \langle \texttt{null}, \theta \circ v \rangle}$$

This rule can be seen as a special case of the *Call* rule. The following is an example on how other rules might propagate $\theta$:

$$\text{Operation}: \frac{\langle \pi, \sigma, e_1 \rangle \downarrow \langle v_1, \theta_1 \rangle \quad \langle \pi, \sigma, e_2 \rangle \downarrow \langle v_2, \theta_2 \rangle}{\langle \pi, \sigma, e_1 \oplus e_2 \rangle \downarrow \langle v, \theta_1 \circ \theta_2 \rangle} (v_1 \oplus v_2 = v)$$

Interestingly, these rules introduce a notion of evaluation order that was not present before when evaluating expressions, as whatever occurs first in $\theta$ can be considered to have been evaluated first as well (this order was present for statements though through $\sigma$).