Programming Languages and Computation

Brischeme Prototype Coursework

*This document contains an introduction to, and reference for, the Brischeme language. This language and its interpreter will form part of an assessed coursework in a future version of this unit. In academic year 2025/26, this coursework is **not** assessed, but it is nevertheless an essential component in the teaching of the first part of the unit.*

1 Setup

You will need *git* and a working OCaml development environment. If you are using the lab machines, *git* is available by default and you can obtain a development environment by executing the following command in a terminal:

```
$ module load ocaml
```

If you want to obtain an OCaml development environment on your own machine, follow the all instructions at https://ocaml.org/docs/installing-ocaml. Ensure you complete all three parts of the setup: installing *opam*, initialising *opam* and installing the platform tools.

Choose a convenient directory to work in and clone the git repository containing the Brischeme interpreter:

```
$ git clone https://github.com/uob-coms20007/brischeme.git
$ cd brischeme
```

This will create a new directory *brischeme* containing the OCaml implementation and place you inside it. The tool *dune* is used to manage the development. To build the code, execute:

```
$ dune build
```

To run the interpreter, execute:

```
$ dune exec brischeme
```

This will drop you into a read-eval-print-loop (REPL) in which you can input Brischeme expressions and have them evaluated. For example:

```
Brischeme
> (define x 3)
> (+ x 4)
7
```

However, you will notice that the interpreter does not support line editing. That is, once you have entered some characters you cannot go back and edit them using e.g. backspace or the left arrow key. However, this facility can be easily provided by a utility program such as *socat*. The utility *socat* is already available on the lab machines. If you don't have it on your own machine then you may have to install from your favourite package manager (e.g. *Homebrew*, *apt-get* etc). Once you have it, you can run:

```
$ socat READLINE EXEC:"dune exec brischeme"
```

and you will then find that line editing is available, and this makes interacting with the REPL much easier.

For convenience, there is a script 'brischeme' in the root of the respository which simply encapsulates the above shell command. You may need to chmod +x ./brischeme to be able to execute it.

2 Introduction to Brischeme

Brischeme is a programming language of the Lisp family. It is most closely related to Scheme, which was a hugely influential dialect of Lisp developed by Guy Steele and Gerald Sussman at MIT in the 1970s. It is extremely simple, yet powerful, which derives from its basis in the untyped λ-calculus (pronounced "lambda calculus"): programs can be distilled down to just a few basic forms – variables, primitive constants, λ-abstractions and function applications. However, because this is λ-calculus presented as a Lisp, you must be prepared for a lot of parentheses.

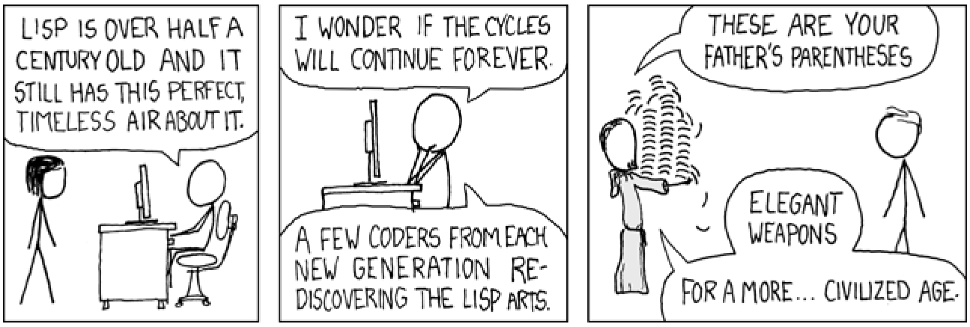

Figure 1: Lisp Cycles (https://xkcd.com/297)

A Brischeme program is essentially a sequence of definitions and expressions to be evaluated. The expressions, called *s-expressions* are either *atoms*, *lambda functions* or parenthesized function calls. Computation of an s-expression involves reducing it to a value and in Brischeme we have four kinds of values, *integers*, *booleans*, *functions* and the degenerate value *nil*. The formal syntax and semantics are given in Appendices A and B.

When you write an expression into the REPL it will be evaluated and the resulting value printed. Non-negative integers are written as you might imagine:

```
> 2025
2025
```

The Boolean value for true is written #t and the value for false as #f.

```
> #f
#f
```

A hallmark of Scheme languages is that all function calls are written prefix and are enclosed in parentheses. This includes calls to the primitive operators (those that are built into the language): $+$, $*$, $-$, $/$, $=$, *and*, *or*, *not* and *if*. This makes them extremely regular. For example, to compute $(3*4)+6$ one writes:

```
> (+ (* 3 4) 6)
18
```

To check if two numbers are equal:

```
> (= 24 42)
#f
```

The primitive *if* can be used to effect control flow. It takes three arguments which are, in order, the guard, the then-branch and the else-branch.

```
> (if #t 1 2)
1
> (if (not #t) 1 2)
2
```

A function value is written using a version of λ-notation. For example, the function that takes in an input x and returns its square $x * x$ can be written as:

```
> (lambda (x) (* x x))
(lambda (x) (* x x))
```

Applying a function to an input is achieved by writing the argument afterwards and enclosing in parentheses:

```
> ((lambda (x) (* x x)) 6)
36
```

It would be tedious to write out the definition of a function every time you wanted to use it, so there is a mechanism to define variable names at the top-level. Afterwards, the name and the expression it abbreviates can be used interchangeably.

```
> (define square (lambda (x) (* x x)))
> (square 6)
36
```

Recursive definitions are supported:

```
> (define fac (lambda (x) (if (< x 1) 1 (* x (fac (- x 1))))))
> (fac 5)
120
```

User-defined functions can take any number of arguments. The following is an implementation of exponentiation x^y:

```
> (define exp (lambda (x y) (if (= y 0) 1 (* x ((exp x) (- y 1))))))
> (exp 2 8)
256
```

3 Coursework Task

*Although this coursework task can be attempted at any time, it is recommended to first complete Week 1, Week 2 and Week 3 problems. This coursework is **not** assessed in 2025/26.*

Consider an extension of Brischeme with a new top-level form (use fn) for dynamically loading files containing Brischeme code. For example, with this extension, the following interaction will be possible.

```
$ cat mycode.bs
(define x 3)
$ rlwrap dune exec brischeme
Brischeme
> x
RUNTIME ERROR: Evaluation undefined for x.
> (use mycode)
> x
3
```

To achieve this, the grammar for forms is extended with a new terminal symbol use and the following additional production rule:

$$\mathit{CForm} ::= \mathsf{use\ ident}$$

Evaluating the top-level form (use fn) reads in the complete text of the file whose (base) name is *fn* and whose extension is *.bs*, parses the text as a program to produce a list of forms and then evaluates and prints the forms.

Task: Implement this extension. Note:

(i) There is no requirement to deal gracefully with errors, such as the non-existence of the file, but you may wish to do so.

(ii) The change required to the parsing functions is relatively straightforward, so you may not need to generate a new parse table. However, if you prefer to do this, then a copy of the LL(1) grammar for the language is given in AppendixC.

A Syntax

A.1 Lexical Structure

The lexical structure is given by the following classes:

ident Identifiers are non-empty finite sequences that start with a lowercase letter of the English alphabet and whose other letters comprise lower and uppcase letters of the English alphabet, digits, underscores, exclamation marks and question marks.

literal Numeric literals are non-empty sequences of digits, representing non-negative integers. Boolean literals are *#t*, representing true, and *#f*, representing false.

primop Primitive operators are: $+$, $-$, $*$, $/$, $=$, $<$, *not*, *and*, *or*, *if*.

Furthermore, the following words are also reserved (cannot be identifiers): *define*, *lambda*.

A.2 Phrase Structure

The phrase structure of the language is given by the following CFG:

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \mathit{Form}^* \\[1ex]
\mathit{Form} & ::= & \mathit{SExpr} \\
 & | & (\ \text{define ident}\ \mathit{SExpr}\ ) \\[1ex]
\mathit{SExpr} & ::= & \text{literal} \\
 & | & \text{ident} \\
 & | & (\ \mathit{SExpr}\ \mathit{SExpr}^*\ ) \\
 & | & (\ \text{primop}\ \mathit{SExpr}^*\ ) \\
 & | & (\ \text{lambda}\ (\ \text{ident}^*\ )\ \mathit{SExpr}\ )
\end{array}
$$

B Semantics

In the following, we use convenient notation:

- P and Q stand for arbitary programs (sequences of s-expressions and definitions).
- S and T stand for arbitrary s-expressions.
- Ss and Ts stand for arbitrary lists of s-expressions, and we write $\langle S; Ss\rangle$ for the list which has head S and tail Ss.
- V and W stand for arbitrary *values*.
- x and y stand for identifiers, and xs for a list of identifiers.
- $\#n$ and $\#m$ stand for the numeric literal whose value is the integer n and m respectively. For, example $\#3$ and $\#(2+1)$ refer to the same numeric literal, whose value is 3.
- $\#b$ for the boolean literals whose semantic value is b. For example, $\#t$ and $\#(t \wedge t)$ are two different ways of writing the boolean literal for true.
- op stands for an arbitrary primitive operator.

Thus n and m always refer to mathematical integers, b will always refer to mathematical truth values. On the other hand $\#n$ is our notation for the integer n written in the syntax of the Brischeme programming language, $\#b$ is our notation for the boolean b written in Brischeme.

Values Values are fully evaluated expressions, and can be described by the following grammar:

$$V ::= \#b \mid \#n \mid (\mathsf{lambda}\ (xs)\ S)$$

Stores A *store* is a mapping from identifiers to values. We use the notation σ and τ to stand for an arbitrary store. We use $\sigma(x)$ as the value assigned to identifier x in the store σ, and $\sigma[x := V]$ to mean the store that can be obtained from σ by mapping the identifier x to the value V.

Operational Semantics We define evaluation in three parts, and so there are three single-step forms, $\Rightarrow_s$, $\Rightarrow_{ss}$ and $\Rightarrow_p$ which have the following shapes:

- Evaluating a single step of an s-expression S with respect to a store σ of top-level definitions, to obtain a new s-expression T:
$$S, \sigma \Rightarrow_s T$$
- Evaluating a single step of a sequence of s-expressions Ss with respect to a store σ of top-level definitions to obtain a new sequence of s-expressions Ts:
$$Ss, \sigma \Rightarrow_{ss} Ts$$
- Evaluating a top-level program P with respect to a store σ, to obtain a new program P' *and a new store* τ
$$P, \sigma \Rightarrow_p Q, \tau$$

These relations are defined by the rules that follow.

B.1 Identifiers

The evaluation of identifiers is the only s-expression evaluation rule that looks into the store:

$$\textsc{Ident}\quad \frac{}{x,\, \sigma \Rightarrow_s \sigma(x)}$$

B.2 Primitive Operator Calls

The if operator is special because its arguments, specifically the arguments that represent the *then* and *else* branches, are *not* all evaluated to values before proceeding.

$$\textsc{IfTrue}\quad \frac{}{(\mathsf{if}\ \#\mathsf{t}\ S\ T),\, \sigma \Rightarrow_s S} \qquad \textsc{IfFalse}\quad \frac{}{(\mathsf{if}\ \#\mathsf{t}\ S\ T),\, \sigma \Rightarrow_s S}$$

For other operators, we evaluate the arguments to values before actioning the operator. This uses the rules for the evaluation of a sequence of s-expressions, which are defined shortly.

$$\textsc{OpArgs}\quad \frac{Ss,\, \sigma \Rightarrow_{ss} Ts}{(op,\, Ss),\, \sigma \Rightarrow_s (op,\, Ts)}\ op \neq \mathsf{if}$$

Once the arguments have been evaluated to values, the behaviour of the rest of the primitive operators is as expected. Note: in the rule (ArithOp) we abuse the coincidence between the syntax of a Brischeme arithmetic operator, like + and the mathematical operator of the same name in order to give a succinct presentation. For example, an instance of this rule is (+ #2 #4) ⇒ #6.

$$\textsc{ArithOp}\quad \frac{}{(op\ \#n\ \#m),\, \sigma \Rightarrow \#(n \otimes m)}\ op \in \{+, -, *, /, \}$$

$$\textsc{AndOp}\quad \frac{}{(\mathsf{and}\ \#b_1\ \#b_2),\, \sigma \Rightarrow \#(b_1 \wedge b_2)} \qquad \textsc{OrOp}\quad \frac{}{(\mathsf{or}\ \#b_1\ \#b_2),\, \sigma \Rightarrow \#(b_1 \vee b_2)} \qquad \textsc{NotOp}\quad \frac{}{(\mathsf{not}\ \#b),\, \sigma \Rightarrow \#(\neg b)}$$

$$\textsc{LessOpT}\quad \frac{}{(<\ \#n\ \#m),\, \sigma \Rightarrow \#t}\ n < m \qquad \textsc{LessOpF}\quad \frac{}{(<\ \#n\ \#m),\, \sigma \Rightarrow \#f}\ n \geq m$$

$$\textsc{EqOpT}\quad \frac{}{(=\ V\ W),\, \sigma \Rightarrow \#t}\ V = W \qquad \textsc{EqOpF}\quad \frac{}{(=\ V\ W),\, \sigma \Rightarrow \#f}\ V \neq W$$

B.3 User-Defined Function Application

To evaluate a function application, we first must evaluate the operator. Once the operator is known to be a user-defined function, we must then evaluate the arguments.

$$\textsc{AppL}\quad \frac{S, \sigma \Rightarrow_s T}{(S\ Ss),\, \sigma \Rightarrow_s (T\ Ss)} \qquad \textsc{AppR}\quad \frac{Ss, \sigma \Rightarrow_{ss} Ts}{((\mathsf{lambda}\ (xs)\ S)\ Ss),\, \sigma \Rightarrow_s ((\mathsf{lambda}\ (xs)\ S)\ Ts)}$$

Only after the operator and the arguments are all values do we action the function call itself, which relies on substitution.

In the following, we use the *substitution* notation $S[V_1/x_1, \ldots, V_n/x_n]$ to stand for the s-expression S but with every occurrence of variable x_1 textually replaced by V_1, x_2 textually replaced by V_2 and so on up to x_n textually replaced by V_n. This is the mechanism by which formal parameters are replaced by actual parameters. For example:

$$(+\ (-\ x\ \#2)\ (*\ y\ \#3))[\#4/x,\ \#8/y] = (+\ (-\ \#4\ \#2)\ (*\ \#8\ \#3))$$

Once the operator and the arguments are evaluated to values, the function call itself can be actioned, which involves replacing formal parameters $x_1, x_2, \ldots, x_n$ in the function body S by the actual parameters $V_1, V_2, \ldots, V_n$ that we supplied at the call site.

$$\textsc{App}\quad \frac{}{((\mathsf{lambda}\ (x_1, x_2, \ldots, x_n)\ S)\ V_1\ V_2\ \cdots V_n),\ \sigma \Rightarrow S[V_1/x_1,\ V_2/x_2, \ldots, V_n/x_n]}$$

Thus, the following is an example of a single step of a function application according to (App):

$$((\mathsf{lambda}\ (x,\ y)\ (+\ (-\ x\ \#2)\ (*\ y\ \#3)))\ \#4\ \#8),\ \sigma \Rightarrow (+\ (-\ \#4\ \#2)\ (*\ \#8\ \#3))$$

B.4 Evaluating a List of SExpr

Evaluating a list of s-expressions proceeds left-to-right:

$$\textsc{StepList}\quad \frac{S,\ \sigma \Rightarrow_s T}{\langle S; Ss\rangle,\ \sigma \Rightarrow_{ss} \langle T; Ss\rangle} \qquad \textsc{StepNext}\quad \frac{Ss,\ \sigma \Rightarrow_{ss} Ts}{\langle V; Ss\rangle,\ \sigma \Rightarrow_{ss} \langle V; Ts\rangle}$$

B.5 Evaluating a Program

Evaluating a program proceeds left to right, but we must also take into account that the store can be updated by the addition of new definitions. Thus, a step of the execution of the program consumes a store (on the left of the arrow) and produces a potentially updated store (on the right of the arrow).

$$\textsc{ProgSExpr}\quad \frac{S,\ \sigma \Rightarrow_s T}{\langle S; P\rangle,\ \sigma \Rightarrow_p \langle T; P\rangle,\ \sigma} \qquad \textsc{ProgNext}\quad \frac{}{\langle V; P\rangle,\ \sigma \Rightarrow_p P,\ \sigma}$$

$$\textsc{ProgDefnBody}\quad \frac{S,\ \sigma \Rightarrow_s T}{\langle(\mathsf{define}\ x\ S); P\rangle,\ \sigma \Rightarrow_p \langle(\mathsf{define}\ x\ T); P\rangle,\ \sigma} \qquad \textsc{ProgDefnBind}\quad \frac{}{\langle(\mathsf{define}\ x\ V); P\rangle,\ \sigma \Rightarrow_p P,\ (\sigma[x := V])}$$

C LL(1) Grammar

The grammar for Brischeme given in Appendix A is *not* LL(1), and so does not form the basis of the parsing functions in *lib/parser.ml*. Instead, these functions are constructed from the equivalent grammar:

$$
\begin{array}{rcl}
\mathit{Prog} & ::= & \mathit{Form}\ \mathit{Prog} \\
\mathit{Prog} & ::= & \mathsf{eof} \\
\\
\mathit{Form} & ::= & \mathit{Atom} \\
\mathit{Form} & ::= & (\ \mathit{CForm}\ ) \\
\\
\mathit{CForm} & ::= & \mathit{Expr} \\
\mathit{CForm} & ::= & \mathsf{define}\ \mathsf{ident}\ \mathit{SExpr} \\
\\
\mathit{Atom} & ::= & \mathsf{literal} \\
\mathit{Atom} & ::= & \mathsf{ident} \\
\\
\mathit{IdentList} & ::= & \mathsf{ident}\ \mathit{IdentList} \\
\mathit{IdentList} & ::= & \epsilon \\
\\
\mathit{SExpr} & ::= & \mathit{Atom} \\
\mathit{SExpr} & ::= & (\ \mathit{Expr}\ ) \\
\\
\mathit{SExprList} & ::= & \mathit{SExpr}\ \mathit{SExprList} \\
\mathit{SExprList} & ::= & \epsilon \\
\\
\mathit{Expr} & ::= & \mathsf{lambda}\ (\ \mathsf{IdentList}\ )\ \mathit{SExpr} \\
\mathit{Expr} & ::= & \mathsf{primop}\ \mathit{SExprList} \\
\mathit{Expr} & ::= & \mathit{SExpr}\ \mathit{SExprList}
\end{array}
$$

For convenience, this grammar is given in a format amenable to parse table generators such as `https://www.cs.princeton.edu/courses/archive/spring20/cos320/LL1/`, but you will need to replace occurrences of ϵ with an empty double single-quote '' to comply with the generator's input syntax.