Programming Languages

Departamento de Engenharia Informática, Técnico Lisboa

MEIC P4 24.25

Project Statement (Phase 1)

© Luís Caires (luis.caires@tecnico.ulisboa.pt | https://luiscaires.org)

Objective of Phase 1

Implement an interpreter for L1++ functional-imperative language

You will follow the principled approach developed in the course lectures.

- L1++ is essentially the L1 functional-imperative language studied in the course lectures (T2 and T3) extended with an additional datatype for immutable lists, using primitives and semantics proposed in the present project statement. You are expected to:
  - Base your implementation in the big-step environment-based semantics studied in the course.
  - Implement full dynamic type-checking, ensuring that all undefined operations are correctly signalled using exceptions and adequate error messages (NB. in Phase 2, we will address static typing).
- A challenge for the development involves implementing an additional datatype of **lazy lists**.For this you will also be expected to describe the semantics of the lazy list primitives using big step semantics.
- The next slides describe the details on the L1++ language.

L1++ language (abstract syntax)

```
EE ->  | num | true | false
     | nil  | EE::EE |
     | EE + EE | EE - EE
     | EE * EE | EE / EE | -EE |  ( EE )
     | EE == EE | EE > EE | EE >= EE | …
     | EE && EE |  EE || EE |  ~ EE
     | match EE { nil -> EE | id:: id -> EE }
     | let (id = EE)+ in EE
     | box EE | EE := EE | * EE
     | if EE then EE else  EE end
     | while EE do EE end
     | print EE | | println EE | EE ; EE
     | fn ( id, EE )
     | app ( EE, E )
```

Work Plan

Implement an interpreter for L1++ functional-imperative language

We suggest that you guide your self using the following work plan.

Consider the language fragments described in the next pages

L1++ language (semantics)

The semantics of L1++ is an extension of the semantics of the functional imperative-language L1 studied in the course, extended with lists.

We describe here the semantics of the primitives related to lists.

L1++ language (values)

The semantics of L1++ is an extension of the semantics of the functional imperative-language L1 studied in the course, extended with lists.

We describe here the values of L1++.

$$\frac{\text{(integer)}}{\text{value } n} \qquad \frac{}{\text{value false}} \qquad \frac{}{\text{value true}} \qquad \frac{}{\text{value clos}(\mathscr{E}, \lambda x \,.\, M)}$$

$$\frac{}{\text{value nil}} \qquad \frac{\text{value } V \quad \text{value } L}{\text{value cons}(V, L)}$$

L1++ language (expressions)

The semantics of L1++ is an extension of the semantics of the functional imperative-language L1 studied in the course, extended with lists.

Here we specify new program expressions of L1++, extending L1.

M, N *(Terms)* $::=$

$\mid \cdots$ (basic terms of L1)

$\mid$ nil *(empty list)*

$\mid$ cons(M, N) *(list cons node)*

$\mid$ match M $\{$nil $\to$ N $\mid$ $(x :: l) \to R$ $\}$ *(list destructor)*

L1++ language (evaluation)

The semantics of L1++ is an extension of the semantics of the functional imperative-language L1 studied in the course, extended with lists.

Here we specify the big-step environment-store semantics for new L1++ primitives.

$\mathscr{E}; \mathcal{S}; M \Downarrow N; \mathcal{S}'$ (M *evaluates to* N in environment $\mathscr{E}$ and store $\mathcal{S}$)

$$\frac{}{\mathscr{E}; \mathsf{nil} \Downarrow \mathsf{nil}} \qquad \frac{\mathscr{E}; \mathcal{S}; M \Downarrow V; \mathcal{S}' \quad \mathscr{E}; \mathcal{S}'; N \Downarrow L; \mathcal{S}'' \quad U = V :: L}{\mathscr{E}; \mathsf{cons}(M, N) \Downarrow U; \mathcal{S}''}$$

$$\frac{\mathscr{E}; \mathcal{S}; M \Downarrow \mathsf{nil}; \mathcal{S}' \quad \mathscr{E}; \mathcal{S}' N \Downarrow U; \mathcal{S}''}{\mathscr{E}; \mathcal{S}; \mathsf{match}\ M\ \{\mathsf{nil} \to\ N \mid \mathsf{cons}(x, l) \to R\ \} \Downarrow U; \mathcal{S}''}$$

$$\frac{\mathscr{E}; \mathcal{S}; M \Downarrow \mathsf{cons}(V, L); \mathcal{S}' \quad \mathscr{E}[x \to V][l \to L]; \mathcal{S}'; R \Downarrow U; \mathcal{S}''}{\mathscr{E}; \mathcal{S}; \mathsf{match}\ M\ \{\mathsf{nil} \to\ N \mid \mathsf{cons}(x, l) \to R\ \} \Downarrow U; \mathcal{S}''}$$

L1++ language (lazy lists)

We present here the "challenge part" of the handout: add lazy lists to L1++ and define its semantics, from the following informal explanation.

A lazy list is build using the constructors nil and lcons(V,L).

nil represent the empty list also for lazy lists.

lcons(V,M) represent the lazy list obtained from adding value V at the head of the list produced by expression M.

The difference between the **lcons(N,M)** node and the basic **cons(N,M)** node is that **lcons** **does not evaluate** the expressions **N** and **M** when building the new node, just stores their suspended code in for future evaluation.

Only when client code opens a lazy list node lcons(N,M) via the match construct, the expressions N and M at the head and tail are evaluated to some values VN and VM. These values are then stored in the node (which becomes a regular **cons** node), and passed in x and l to the match | **cons**(x,l) branch.

L1++ language (lazy lists)

The code below defines the infinite stream of fibonacci numbers at fibogen.

Then prints its first 100 elements.

Notice the “never message” will never be printed, as the end of the fibogen lazy list will never be reached.

```
let fibo = fn a, b => { lcons(a,fibo(b,a+b)) };
let fibogen = fibo(0,1);
let count = box ( 100 ) ;
let lv = new( fibogen );
while (count* != 0) {
    match lv* {
     | nil -> print “never happens”
     | v :: tail -> println (v); lv := tail
    }
    count := count - 1
};;
```

Submission Instructions

1 - Place your code in a gitlab repo shared with me (luis.caires@tecnico.ulisboa.pt)

2 - Include a small report (pdf) explaining how you have defined the big step evaluation rules specification for the lazy list primitives, and their implementation.

NOTE: the contents of the shared repo **must not be changed** after the deadline.

3- You may use the software available in the gitlab RNL standard installation and javacc (installation requested).

4 - Include a top level script "makeit", that I may use to compile all your source code by typing "\.makeit" at the command line.

5- Include some L1++ code examples to demonstrate your interpreter. You must include original new examples of your own, not just the ones I gave in the lectures.

Appendix Notes

1 - Some startup source code (Java) will be placed in the PL Course Drive. This startup code uses the javacc parser generator, which is quite simple to use.

2 - You may find also there a EBNF LL(1) spec of the L1++ concrete syntax.

2 - This handout statement will be presented in the lab sessions, and the startup source code as well. The startup code only covers a very small fragment of the class, but already hints to the general structure of the interpreter we look for.

2 - You are free to pick another implementation language (Java, C, C++ or Rust), and some other standard parser generator, if you wish, but that will take you some extra time to redo the recode the parser.