

The L₃ project

Advanced Compiler Construction
Michel Schinz - 2014-02-20

1

Project overview

What you will get (as the semester progresses):

- parts of an L₃ compiler written in Scala, and
- parts of a virtual machine, written in C.

What you will have to do:

- one non-graded exercise to warm you up,
- complete the compiler,
- complete the virtual machine.

2

The L₃ language

3

The L₃ language

L₃ is a Lisp-like language. Its main characteristics are:

- it is "dynamically typed",
- it is functional:
 - functions are first-class values, and can be nested,
 - there are few side-effects (exceptions: mutable blocks and I/O),
- it automatically frees memory,
- it has six kinds of values: unit, booleans, characters, integers, blocks and functions,
- it is simple but quite powerful.

4

A taste of L₃

An L₃ function to compute x^y for $x \in \mathbb{Z}, y \in \mathbb{N}$:

```
(defrec pow
  (fun (x y)
    (cond ((= 0 y) 1)
          ((even? y)
           (let ((t (pow x (/ y 2))))
             (* t t)))
          (#t
           (* x (pow x (- y 1)))))))
```

$x^0 = 1$

$x^{2z} = (x^z)^2$

$x^{z+1} = x(x^z)$

5

Top-level definitions

(def n e)

Top-level non-recursive definition. The expression e is evaluated and its value is bound to name n in the rest of the program. The name n is *not* visible in expression e .

(defrec n f)

Top-level recursive *function* definition. The function expression f is evaluated and its value is bound to name n in the rest of the program. The function can be recursive, i.e. the name n is visible in the function expression f .

6

Local definitions

(let ((n₁ e₁) (n₂ e₂) ...) b₁ ... b_k)

Parallel local value definition. The expressions e_1, e_2, \dots are evaluated in that order, and their values are then bound to names n_1, \dots in the body b_1, \dots, b_k . The value of the whole expression is the value of b_k .

(let* ((n₁ e₁) (n₂ e₂) ...) b₁ ... b_k)

Sequential local value definition. Equivalent to a nested sequence of **let**: **(let ((n₁ e₁)) (let ((n₂ e₂)) ...))**

(letrec ((n₁ f₁) (n₂ f₂) ...) b₁ ... b_k)

Recursive local function definition. The function expressions f_1, f_2, \dots are evaluated and bound to names n_1, n_2, \dots in the body b_1, \dots, b_k . The functions can be mutually recursive.

7

Functions

(fun (n₁ n₂ ...) b₁ ... b_k)

Anonymous function with arguments n_1, n_2, \dots and body b_1, \dots, b_k . The return value of the function is the value of b_k .

(e e₁ e₂ ...)

Function application. Expressions e, e_1, e_2, \dots are evaluated in that order, and then the value of e - which must be a function - is applied to the value of e_1, e_2, \dots

8

Conditional expressions

(if e_1 e_2 e_3)

Two-ways conditional. If e_1 evaluates to a true value (i.e. anything but **#f**), e_2 is evaluated, else e_3 is evaluated. The value of the whole expression is the value of the evaluated branch.

The else branch is optional and defaults to **#u** (unit).

(cond (c_1 e_1) (c_2 e_2) ...)

N-ways conditional. If c_1 evaluates to a true value, evaluate e_1 ; else, if c_2 evaluates to a true value, evaluate e_2 ; etc. The value of the whole expression is the value of the evaluated branch or **#u** if none of the conditions are true.

9

Logical expressions

(and e_1 e_2)

Equivalent to **(if** e_1 e_2 **#f**).

(or e_1 e_2)

Equivalent to **(let** ((\underline{v} e_1)) **(if** v v e_2)), where v is a fresh name.

(not e)

Equivalent to **(if** e **#f** **#t**).

10

Loops and blocks

(rec n ((n_1 e_1) (n_2 e_2) ...) b_1 b_2 ...)

General loop. Equivalent to:

(letrec ((n (**fun** (n_1 n_2 ...) b_1 b_2 ...)))
(n e_1 e_2 ...))

(begin b_1 b_2 ... b_k)

Sequential evaluation. First evaluate expression b_1 , discarding its value, then b_2 , etc. Finally evaluate b_k , whose value is the value of the whole expression.

11

Literal values

" $c_1c_2\dots c_n$ "

String literal (translated to a block expression, see later).

' c '

Character literal.

... -2 -1 0 1 2 3 ...

Integer literals.

#t **#f**

Boolean literals (true and false, respectively).

#u

Unit literal.

12

Primitives

(@ p e₁ e₂ ...)

Primitive application. First evaluate expressions e₁, e₂, ... in that order, and then apply primitive p to the value of these expressions.

L₃ offers the following primitives:

- integer: + - * / % < <= > >= int->char
- bit vectors (integers): << >> & | ^
- polymorphic comparison: = !=
- type tests: block? int? char? bool? unit?
- character: char-read char-print char->int
- tagged blocks (see later): block-alloc-n
block-tag block-length
block-get block-set!

13

Valid primitive arguments

Primitives only work correctly when applied to certain types of arguments, otherwise their behavior is undefined.

+ - * / % << >> & | ^ : int * int ⇒ int

< <= > >= : int * int ⇒ bool

= != : ∀α, β. α * β ⇒ bool

int->char : int ⇒ char

char->int : char ⇒ int

block? int? char? bool? unit? : ∀α. α ⇒ bool

char-read : ⇒ char

char-print : char ⇒ unit

14

Valid primitive arguments

block-alloc-n : int ⇒ block

block-tag block-length : block ⇒ int

block-get : ∀α. block * int ⇒ α

block-set! : ∀α. block * int * α ⇒ unit

15

Tagged blocks

L₃ offers a single structured datatype: tagged blocks. They are manipulated with the following primitives:

(@ block-alloc-n s)

Allocates an uninitialized block with tag n and length s.

(@ block-tag b)

Returns the tag of block b (as an integer).

(@ block-length b)

Returns the length of block b.

(@ block-get b n)

Returns the nth element (0-based) of block b.

(@ block-set! b n v)

Sets the nth element (0-based) of block b to v.

16

Using tagged blocks

Tagged blocks are a low-level data structure. They are not meant to be used directly in programs, but rather as a means to implement more sophisticated data structures like strings, arrays, lists, etc.

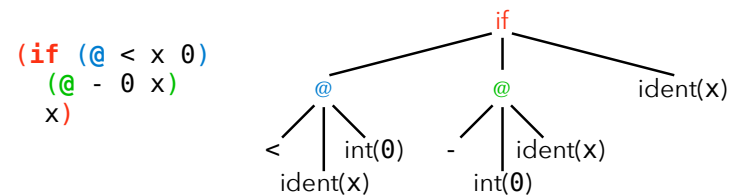
The valid tags range from 0 to 255, inclusive. Tags ≥ 200 are reserved by the compiler, while the others are available for general use. (For example, our L₃ library uses a few tags to represent arrays, lists, etc.)

17

Grasping the syntax

Like all Lisp-like languages, L₃ "has no syntax", in that its concrete syntax is very close to its abstract syntax.

For example, the L₃ expression on the left is almost a direct transcription of a pre-order traversal of its AST on the right, in which nodes are parenthesized and tagged, while leaves are unadorned.



18

L₃ EBNF grammar (1)

```
program ::= { def | defrec | expr }
def ::= (def ident expr)
defrec ::= (defrec ident fun)
expr ::= fun | let | let* | letrec | rec | begin | if | cond | and | or
        | not | app | prim | ident | num | str | chr | bool | unit
exprs ::= expr { expr }
fun ::= (fun ({ ident }) exprs)
let ::= (let ({ (ident expr) }) exprs)
let* ::= (let* ({ (ident expr) }) exprs)
letrec ::= (letrec ({ (ident fun) }) exprs)
rec ::= (rec ident ({ (ident expr) }) exprs)
begin ::= (begin exprs)
```

19

L₃ EBNF grammar (2)

```
if ::= (if expr expr [ expr ])
cond ::= (cond (expr expr) {(expr expr)})
and ::= (and expr expr)
or ::= (or expr expr)
not ::= (not expr)
app ::= (expr { expr })
prim ::= (@ prim-name { expr })
```

20

L₃ EBNF grammar (3)

```
str ::= "{any character except newline}"
chr ::= 'any character'
num ::= [-] digit { digit }
bool ::= #t | #f
unit ::= #u
ident ::= identstart { identstart | digit }
identstart ::= a | ... | z | A | ... | Z | | ! | % | & | * | + | -
| . | / | : | < | = | > | ? | ^ | _ | ~
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
prim-name ::= block-tag | block-alloc-n
| etc.
```

0 ≤ n ≤ 255

21

Exercises

Write the L₃ version of the factorial function, defined as:

fact(0) = 1

fact(n) = n · fact(n - 1) [if n > 0]

What does the following (valid) L₃ program compute?

```
((fun (f x) (f x))
 (fun (x) (@+ x 1))
 20)
```

22

L₃ syntactic sugar

L₃ has a substantial amount of **syntactic sugar**: constructs that can be syntactically translated to other existing constructs. Syntactic sugar does not offer additional expressive power to the programmer, but some syntactical convenience.

For example, L₃ allows **if** expressions without an else branch, which is implicitly taken to be the unit value **#u**:

(if e₁ e₂) ⇔ (if e₁ e₂ #u)

23

L₃ syntactic sugar

24

Desugaring

Syntactic sugar is typically removed very early in the compilation process - e.g. during parsing - to simplify the language that the compiler has to handle.

This process is known as **desugaring**.

Desugaring can be specified using rewriting rules that rewrite a sugared term into a (partially) desugared one.

For example, **if** expressions without an else branch can be desugared using the following rewriting rule:

$(\mathbf{if} \ e_1 \ e_2) \rightsquigarrow_{\text{ds}} (\mathbf{if} \ e_1 \ e_2 \ \#\mathbf{u})$

25

L₃ desugaring (1)

To simplify the rewriting rules for whole programs, we assume that all top-level expressions are wrapped sequentially in a **(program ...)** expression.

$(\mathbf{program} \ \dots \ (\mathbf{def} \ n \ e) \ b)$

$\rightsquigarrow_{\text{ds}} (\mathbf{program} \ \dots \ (\mathbf{let} \ ((n \ e)) \ b))$

$(\mathbf{program} \ \dots \ (\mathbf{defrec} \ n \ f) \ b)$

$\rightsquigarrow_{\text{ds}} (\mathbf{program} \ \dots \ (\mathbf{letrec} \ ((n \ f)) \ b))$

$(\mathbf{program} \ \dots \ e_1 \ e_2)$

$\rightsquigarrow_{\text{ds}} (\mathbf{program} \ \dots \ (\mathbf{begin} \ e_1 \ e_2))$

26

L₃ desugaring (2)

$(\mathbf{let}^* \ ((n_1 \ e_1)) \ b)$

$\rightsquigarrow_{\text{ds}} (\mathbf{let} \ ((n_1 \ e_1)) \ b)$

$(\mathbf{let}^* \ ((n_1 \ e_1) \ (n_2 \ e_2) \ \dots) \ b)$

$\rightsquigarrow_{\text{ds}} (\mathbf{let} \ ((n_1 \ e_1)) \ (\mathbf{let}^* \ ((n_2 \ e_2) \ \dots) \ b))$

$(\mathbf{let}^* \ ((n_1 \ e_1) \ \dots) \ b_1 \ \dots)$

$\rightsquigarrow_{\text{ds}} (\mathbf{let}^* \ ((n_1 \ e_1) \ \dots) \ (\mathbf{begin} \ b_1 \ \dots))$

$(\mathbf{let} \ ((n_1 \ e_1) \ \dots) \ b_1 \ \dots)$

$\rightsquigarrow_{\text{ds}} (\mathbf{let} \ ((n_1 \ e_1) \ \dots) \ (\mathbf{begin} \ b_1 \ \dots))$

$(\mathbf{letrec} \ ((n_1 \ f_1) \ \dots) \ b_1 \ \dots)$

$\rightsquigarrow_{\text{ds}} (\mathbf{letrec} \ ((n_1 \ f_1) \ \dots) \ (\mathbf{begin} \ b_1 \ \dots))$

27

L₃ desugaring (3)

To avoid non-termination of the desugaring process, we suppose that functions bound by a **defrec** or **letrec** are tagged - e.g. during parsing - with a hash sign (#).

$(\mathbf{fun} \ (n_1 \ \dots) \ b_1 \ b_2 \ \dots)$ **underlined names are fresh**

$\rightsquigarrow_{\text{ds}} (\mathbf{letrec} \ ((\underline{\mathbf{f}} \ (\mathbf{fun}^\# \ (n_1 \ \dots) \ b_1 \ b_2 \ \dots))) \ \mathbf{f})$

$(\mathbf{rec} \ n \ ((n_1 \ e_1) \ (n_2 \ e_2) \ \dots) \ b_1 \ b_2 \ \dots)$

$\rightsquigarrow_{\text{ds}} (\mathbf{letrec} \ ((n \ (\mathbf{fun}^\# \ (n_1 \ n_2 \ \dots) \ b_1 \ b_2 \ \dots)))$

$(n \ e_1 \ e_2 \ \dots))$

$(\mathbf{fun}^\# \ (n_1 \ \dots) \ b_1 \ b_2 \ \dots)$

$\rightsquigarrow_{\text{ds}} (\mathbf{fun}^\# \ (n_1 \ \dots) \ (\mathbf{begin} \ b_1 \ b_2 \ \dots))$

28

L₃ desugaring (4)

```
(begin e)
  ↦ds e
(begin e1 e2 ...)
  ↦ds (let ((n e1)) (begin e2 ...))
(if c e)
  ↦ds (if c e #u)
(cond (c1 e1))
  ↦ds (if c1 e1)
(cond (c1 e1) (c2 e2) ...)
  ↦ds (if c1 e1 (cond (c2 e2) ...))
```

29

L₃ desugaring (5)

```
(and e1 e2)
  ↦ds (if e1 e2 #f)
(or e1 e2)
  ↦ds (let ((v e1)) (if v v e2))
(not e)
  ↦ds (if e #f #t)
```

30

L₃ desugaring (6)

L₃ does not have a string type. It offers string literals, though, which are desugared to blocks containing characters.

```
"c1c2...cn"
  ↦ds (let ((s (@block-alloc-200 n)))
        (@block-set! s 0 'c1')
        (@block-set! s 1 'c2')
        ...
        s)
```

the (reserved)
tag 200 is used for
strings

31

Desugaring contexts

Desugaring rules cannot be applied anywhere, but only in specific locations. For example, it would be incorrect to try to desugar the parameter list of a function.

This constraint can be captured by specifying all the **contexts** in which it is valid to perform a rewrite, where a context is a term with a single **hole** denoted by \square .

The hole of a context C can be plugged with a term T, an operation written as C{T}.

For example, if C is (if \square 1 2), then C{(< x y)} is (if (< x y) 1 2).

32

Desugaring contexts

All the contexts C_{ds} in which it is legal to apply the desugaring rewrite rule \rightsquigarrow_{ds} are generated by the following grammar:

```
 $C_{ds} ::= \square$   
| (program  $C_{ds}$ )  
| (let (( $n_1 e_1$ ) ... ( $n_i C_{ds}$ ) ... ( $n_k e_k$ ))  $e$ )  
| (let (( $n_1 e_1$ ) ... ( $n_k e_k$ ))  $C_{ds}$ )  
| (letrec (( $n_1 f_1$ ) ... ( $n_i C_{ds}$ ) ... ( $n_k f_k$ ))  $e$ )  
| (letrec (( $n_1 f_1$ ) ... ( $n_k f_k$ ))  $C_{ds}$ )  
| (fun# ( $n_1$  ...  $n_k$ )  $C_{ds}$ )  
| (if  $C_{ds} e_2 e_3$ ) | (if  $e_1 C_{ds} e_3$ ) | (if  $e_1 e_2 C_{ds}$ )  
| ( $C_{ds} e_1$  ...  $e_k$ ) | ( $e e_1$  ...  $C_{ds}$  ...  $e_k$ )  
| (@p  $e_1$  ...  $C_{ds}$  ...  $e_k$ )
```

33

Desugaring relation

Having defined the desugaring rewrite rules and the valid desugaring contexts, it is now easy to specify the desugaring relation that maps a sugared program to a (partially) desugared program:

$C_{ds}\{T\} \Rightarrow_{ds} C_{ds}\{T'\}$ where $T \rightsquigarrow_{ds} T'$

Completely desugaring a program amounts to reducing it using the desugaring relation until it cannot be reduced further.

34

L₃ desugaring example

```
(program (char-print (if #t 'o' 'k'))  
         (char-print (if #f 'o' 'k'))))
```

```
 $\Rightarrow_{ds}$  (program  
      (begin (char-print (if #t 'o' 'k'))  
             (char-print (if #f 'o' 'k')))))
```

```
 $\Rightarrow_{ds}$  (program  
      (let ((t (char-print (if #t 'o' 'k'))))  
            (begin (char-print (if #f 'o' 'k')))))
```

```
 $\Rightarrow_{ds}$  (program  
      (let ((t (char-print (if #t 'o' 'k'))))  
            (char-print (if #f 'o' 'k')))))
```

$\not\Rightarrow_{ds}$ can't be rewritten further

35

L₃ desugaring exercise

Desugar the following L₃ expression by applying the desugaring relation until you get a term that cannot be rewritten anymore:

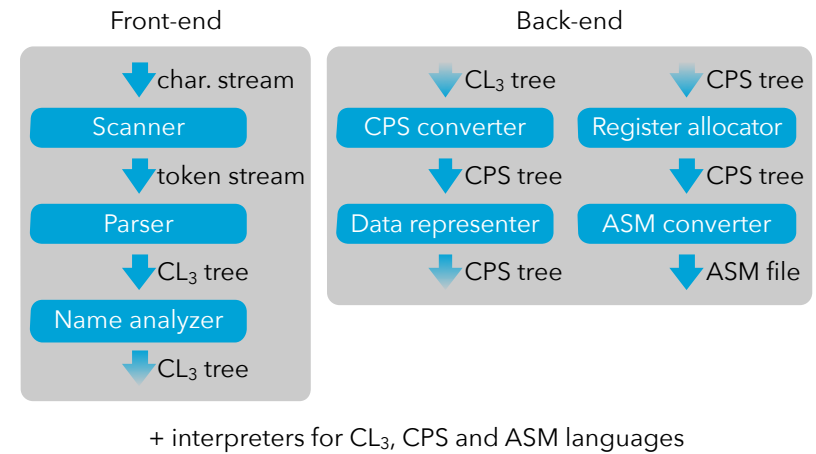
```
(rec loop ((i 1))  
        (int-print i)  
        (if (< i 9)  
            (loop (+ i 1))))
```

36

The L₃ compiler

37

L₃ compiler architecture



38

Intermediate languages

The L₃ compiler manipulates a total of four languages:

1. L₃ is the source language that is parsed, but never exists as a tree (it is desugared to CL₃ immediately),
2. CL₃ (a.k.a. CoreL₃) is the desugared version of L₃,
3. CPS is the main intermediate language, on which optimizations are performed,
4. ASM is the assembly language of the target (virtual) machine.

The compiler contains interpreters for the last three languages, which is useful to check that a program behaves in the same way as it undergoes transformation.

These interpreters also serve as semantics for their language.

39