The L₃ project

Advanced Compiler Construction
Michel Schinz – 2016-02-25

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.

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.
A taste of L₃

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

\[
\text{(defrec pow} (\text{fun} (x y) \\
\quad (\text{cond} ((= 0 y) 1) \\
\quad \quad ((\text{even?} y) \\
\quad \quad \quad (\text{let} (((t (\text{pow} x (/ y 2)))) (* t t))) \\
\quad \quad \quad (#t (* x (\text{pow} x (- y 1)))))))))
\]

\[ x^0 = 1 \]
\[ x^{2z} = (x^2)^z \]
\[ x^{z+1} = x(x^z) \]

Literal values

"c₁...cₙ"
String literal (translated to a block expression, see later).
'c'
Character literal.
... -2 -1 0 1 2 3 ...
Integer literals (also available in base 16 with #x prefix, or in base 2 with #b prefix).
#t #f
Boolean literals (true and false, respectively).
#u
Unit literal.

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.

Local definitions

(let ((n₁ e₁) ...) b₁ b₂ ...)
Parallel local value definition. The expressions e₁, ... are evaluated in that order, and their values are then bound to names n₁, ... in the body b₁, b₂, ... The value of the whole expression is the value of the last bₙ.

(let* ((n₁ e₁) ...) b₁ b₂ ...)
Sequential local value definition. Equivalent to a nested sequence of let: (let ((n₁ e₁)) (let (...) ...))

(letrec ((n₁ f₁) ...) b₁ b₂ ...)
Recursive local function definition. The function expressions f₁, ... are evaluated and bound to names n₁, ... in the body b₁, b₂ ... The functions can be mutually recursive.
Functions

(fun (n1 ...) b1 b2 ...)
Anonymous function with arguments n1, ... and body b1, b2, ... The return value of the function is the value of the last b.
(e e1 ...)
Function application. Expressions e, e1, ... are evaluated in that order, and then the value of e – which must be a function – is applied to the value of e1, ...
Note: if e is a simple identifier, a special form of name resolution, based on arity, is used – see later.

Logical expressions

(and e1 e2 e3 ...)
Short-cutting conjunction. If e1 evaluates to a true value, proceed with the evaluation of e2, and so on. The value of the whole expression is that of the last evaluated e.
(or e1 e2 e3 ...)
Short-cutting disjunction. If e1 evaluates to a true value, produce that value. Otherwise, proceed with the evaluation of e2, and so on.
(not e)
Negation. If e evaluates to a true value, produce the value #f. Otherwise, produce the value #t.

Conditional expressions

(if e1 e2 e3)
Two-ways conditional. If e1 evaluates to a true value (i.e. anything but #f), e2 is evaluated, otherwise e3 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 (c1 b1,1 b1,2 ...) (c2 b2,1 b2,2 ...) ...)
N-ways conditional. If c1 evaluates to a true value, evaluate b1,1, b1,2 ...; else, if c2 evaluates to a true value, evaluate b2,1, b2,2 ...; etc. The value of the whole expression is the value of the evaluated branch or #u if none of the conditions are true.

Loops and blocks

(rec n ((n1 e1) ...) b1 b2 ...)
General loop. Equivalent to:
(letrec ((n (fun (n1 ...) b1 b2 ...)))
  (n e1 ...))
(begin b1 b2 ...)
Sequential evaluation. First evaluate expression b1, discarding its value, then b2, etc. The value of the whole expression is the value of the last b.
Arity-based name lookup

A special name lookup rule is used when analyzing a function application in which the function is a simple name:

\((n \ e_1 \ e_2 \ldots \ e_k)\)

In such a case, the name \(n\@k\) (i.e. the name itself, followed by \(@\), followed by the arity in base 10) is first looked up, and used instead of \(n\) instead if it exists. Otherwise, name analysis proceeds as usual.

This allows a kind of overloading based on arity (although it is not overloading per se).

Arity-based name lookup

Arity-based name lookup can for example be used to define several functions to create lists of different lengths:

```
(def list-make@1 (fun (e1) ...))
(def list-make@2 (fun (e1 e2) ...))
and so on for list-make@3, list-make@4, etc.
```

With these definitions, the following two function applications are both valid:

1. \((\text{list-make } 1)\) (invokes list-make@1),
2. \((\text{list-make } 1 (+ 2 3))\) (invokes list-make@2).

However, the following one is not valid, unless a definition for the bare name list-make also appears in scope:

\((\text{map list-make } l)\)

Values

\(\mathbb{L}_3\) integers are represented using 31 (!) bits, in two’s complement, and therefore range from \(-2^{31}\) to \(2^{31} - 1\).

\(\mathbb{L}_3\) characters represent Unicode code points, i.e. 21 bits positive integers in one of the following ranges:
- from 0000\(_{16}\) to D7FF\(_{16}\), or
- from E000\(_{16}\) to 10FFFF\(_{16}\).

(Notice that characters and integers are different types, and conversion between the two must be done using the primitives \text{int->char} and \text{char->int}.)

Primitives

\((@ \ p \ e_1 \ e_2 \ldots)\)

Primitive application. First evaluate expressions \(e_1, e_2, \ldots\) in that order, and then apply primitive \(p\) to the value of these expressions.

\(\mathbb{L}_3\) offers the following primitives:
- integer: \(< \ <= \ >\ >= \ + \ - \ * \ / \ %\)
- bit vectors (integers): \(<< >> \ & \ | \ ^\)
- polymorphic: \(= != \ id\)
- type tests: \text{block? int? char? bool? unit?}
- character: \text{char->int int->char}
- I/O: \text{byte-read byte-write}
- tagged blocks: \text{block-alloc-n}

Floored integer division, e.g. \((/ -5 2) \Rightarrow -3\)
Tagged blocks

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

(\texttt{@ block-alloc-n s})

Allocates an uninitialized block with tag n and length s.

(\texttt{@ block-tag b})

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

(\texttt{@ block-length b})

Returns the length of block b.

(\texttt{@ block-get b n})

Returns the n\textsuperscript{th} element (0-based) of block b.

(\texttt{@ block-set! b n v})

Sets the n\textsuperscript{th} element (0-based) of block b to v.

Valid primitive arguments

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

+ - * \& \mid ^ \div \mod \ll \gg : \text{int} \times \text{int} \Rightarrow \text{int}

\texttt{byte-read} : \Rightarrow \text{int \ between \ 0 \ and \ 255, \ or \ -1}

\texttt{byte-write} : \exists \alpha. \text{int \ between \ 0 \ and \ 255} \Rightarrow \alpha

\texttt{block-alloc-n} : \text{int} \Rightarrow \text{block}

\texttt{block-tag} \ block-length : \text{block} \Rightarrow \text{int}

\texttt{block-get} : \forall \alpha. \text{block} \times \text{int} \Rightarrow \alpha

\texttt{block-set!} : \forall \alpha \exists \beta. \text{block} \times \text{int} \times \alpha \Rightarrow \beta

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.)
Undefined behavior

The fact that primitives have undefined behavior when applied to invalid arguments means that they can do anything in such a case. For example, division by zero can produce an error, crash the program, or produce an arbitrary value.

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.

```
(if (@ < x 0)
  (@ - 0 x)
  x)
```

L3 EBNF grammar (1)

```
program ::= { def | defrec | expr } 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)
```

L3 EBNF grammar (2)

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

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

\[
\text{fact}(0) = 1 \\
\text{fact}(n) = n \cdot \text{fact}(n - 1) \quad \text{[if } n > 0\text{]}
\]

What does the following (valid) L₃ program compute?

```l3
(((fun (f x) (f x))
  (fun (x) (@+ x 1))
20)
```
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)
```

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 as a function denoted by $⟦·⟧$ taking an L₃ term and producing a desugared CL₃ term (CL₃ is Core L₃, the desugared version of L₃). To clarify the presentation, L₃ terms appear in orange, CL₃ terms in green, and meta-terms in black.

L₃ desugaring (1)

To simplify the specification of desugaring for whole programs, we assume that all top-level expressions are wrapped sequentially in a single (program ...) expression.

```
⟦(program (def n e) s₁ s₂ ...)⟧ =
  (let ((n (e))) (program s₁ s₂ ...))

⟦(program (defrec n e) s₁ s₂ ...)⟧ =
  (letrec ((n (e))) (program s₁ s₂ ...))

⟦(program e s₁ s₂ ...)⟧ =
  (begin e (program s₁ s₂ ...))

⟦(program e)⟧ =
  [e]
```

L₃ desugaring (2)

Desugaring sometimes requires the creation of **fresh names**, i.e. names that do not appear anywhere else in the program. Their binding occurrence is underlined in the rules, as illustrated by the one below.

```
⟦(begin b₁ b₂ b₃ ...)⟧ =
  (let ((t (b₁))) (begin b₂ b₃ ...))

⟦(begin b)⟧ =
  [b]
```
L₃ desugaring (3)

\[
(\text{let } ((n_1 \ e_1) \ldots) b_1 b_2 \ldots) = \\
(\text{let } ((n_1 \ e_1) \ldots) ((\text{begin } b_1 b_2 \ldots)))
\]

L₃ desugaring (4)

\[
(\text{fun } (n_1 \ldots) b_1 b_2 \ldots) = \\
(\text{letrec } ((f \ (\text{fun } (n_1 \ldots) ((\text{begin } b_1 b_2 \ldots)))))) f)
\]

L₃ desugaring (5)

\[
(\text{if } e_1 e_2 e_3 \ldots) = \\
(\text{if } e_1 (\text{and } e_2 e_3 \ldots) \#f)
\]

L₃ desugaring (6)

\[
(\text{and } e_1 e_2 e_3 \ldots) = \\
(\text{if } e_1 (\text{and } e_2 e_3 \ldots) \#f)
\]

\[
(\text{and } e) = \\
[e]
\]

\[
(\text{or } e_1 e_2 e_3 \ldots) = \\
((\text{let } ((v \ e_1)) (\text{if } v v (\text{or } e_2 e_3 \ldots))))
\]

\[
(\text{or } e) = \\
[e]
\]

\[
(\text{not } e) = \\
(\text{if } e \#f \#t)
\]
L₃ desugaring (7)

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

\[ "c_1...c_n" = \]
\[
(\text{let } ((s (\text{block-alloc-200 } n)))
  (\text{block-set! } s 0 'c_1'))
...
]

\[ [] = \text{if } l \text{ is a (non-string) literal} \]
\[ [n] = \text{if } n \text{ is a name} \]

Exercise

Desugar the following L₃ expression:

\[
\text{rec } \text{loop } ((i 1))
  \text{int-print } i
  \text{if } (< i 9)
    \text{loop } (+ i 1)))
\]

L₃ desugaring example

\[
((\text{program } (\text{byte-write (if #t 79 75))})
  (\text{byte-write (if #f 79 75)})))
= ((\text{(begin } (\text{byte-write (if #t 79 75)})
  (\text{program }
    (\text{byte-write (if #f 79 75])))\text{)})
= ((\text{let } ((t [((\text{byte-write (if #t 79 75)})]))
  ((\text{begin }
    (\text{program }
      (\text{byte-write (if #f 79 75)})))\text{)})
= (\text{let } ((t (\text{byte-write (if #t 79 75)})))
  (\text{byte-write (if #f 79 75)})))

The L₃ compiler
L₃ compiler architecture

Front-end

- char. stream
- Scanner
- token stream
- Parser
- CL₃ tree
- Name analyzer
- CL₃ tree

Back-end

- CL₃ tree
- CPS converter
- Value representer
- Register allocator
- CPS tree
- CPS tree
- CPS tree
- ASM converter
- ASM file

+ interpreters for CL₃, CPS and ASM languages

Note: CL₃, CPS and ASM each designate a family of very similar languages, with minor differences between them.

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 is undergoes transformation. These interpreters also serve as semantics for their language.