The L₃ project

Advanced Compiler Construction
Michel Schinz – 2016-02-25
What you will get (as the semester progresses):
- parts of an L_3 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
The $L_3$ language

$L_3$ 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 $x \in \mathbb{Z}$, $y \in \mathbb{N}$:

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

"c_1…c_n"
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 bᵢ.

(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.
(\textbf{fun} (n_1 \ldots) b_1 \ b_2 \ldots)
Anonymous function with arguments $n_1$, $\ldots$ and body $b_1, b_2, \ldots$ The return value of the function is the value of the last $b_i$.

(e e_1 \ldots)
Function application. Expressions $e$, $e_1$, $\ldots$ are evaluated in that order, and then the value of $e$ – which must be a function – is applied to the value of $e_1$, $\ldots$

Note: if $e$ is a simple identifier, a special form of name resolution, based on arity, is used – see later.
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 b₁₁ b₁₂ ... ) (c₂ b₂₁ b₂₂ ... ) ... )
N-ways conditional. If c₁ evaluates to a true value, evaluate b₁₁, b₁₂ ... ; else, if c₂ evaluates to a true value, evaluate b₂₁, b₂₂ ... ; etc. The value of the whole expression is the value of the evaluated branch or #u if none of the conditions are true.
Logical expressions

\((\text{and } e_1 e_2 e_3 \ldots)\)
Short-cutting conjunction. If \(e_1\) evaluates to a true value, proceed with the evaluation of \(e_2\), and so on. The value of the whole expression is that of the last evaluated \(e_i\).

\((\text{or } e_1 e_2 e_3 \ldots)\)
Short-cutting disjunction. If \(e_1\) evaluates to a true value, produce that value. Otherwise, proceed with the evaluation of \(e_2\), and so on.

\((\text{not } e)\)
Negation. If \(e\) evaluates to a true value, produce the value \(#\text{f}\). Otherwise, produce the value \(#\text{t}\).
Loops and blocks

\[(\text{rec } n \ ((n_1 \ e_1) \ \ldots ) \ b_1 \ b_2 \ \ldots )\]

General loop. Equivalent to:

\[(\text{letrec } ((n \ (\text{fun } (n_1 \ \ldots ) \ b_1 \ b_2 \ \ldots ))) )\]

\[(\text{letrec } (n \ e_1 \ \ldots ))\]

\[(\text{begin } b_1 \ b_2 \ \ldots )\]

Sequential evaluation. First evaluate expression \(b_1\), discarding its value, then \(b_2\), etc. The value of the whole expression is the value of the last \(b_i\).
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. (list-make 1) (invokes list-make@1),
2. (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:

(map list-make l)
L₃ integers are represented using 31 (!) bits, in two's complement, and therefore range from $-2^{31}$ to $2^{31} - 1$. L₃ 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 `int->char` and `char->int`.)
Primitives

(\texttt{@ p e_1 e_2 ...})

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

\(L_3\) offers the following primitives:

- integer: < <= > >= + - * / %
- bit vectors (integers): << >> & | ^
- polymorphic: = != id
- type tests: block? int? char? bool? unit?
- character: char-\rightarrow\text{int} \text{int-}\rightarrow\text{char}
- I/O: byte-read byte-write
- tagged blocks: block-alloc\(-n\) block-tag block-length block-get block-set!

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:

(@ 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.
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 $\geq 200$ are reserved by the compiler, while the others are available for general use. (For example, our L$_3$ library uses a few tags to represent arrays, lists, etc.)
Valid primitive arguments

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

\[
+ \quad - \quad \ast \quad \ll\quad \gg\quad \&\quad |\quad \wedge : \text{int} \times \text{int} \Rightarrow \text{int}
\]

\[
\lor \quad \% : \text{int} \times \text{non-zero int} \Rightarrow \text{int}
\]

\[
< \quad \leq \quad > \quad \geq : \text{int} \times \text{int} \Rightarrow \text{bool}
\]

\[
= \quad \neq : \forall \alpha, \beta. \alpha \times \beta \Rightarrow \text{bool}
\]

\[
\text{id} : \forall \alpha. \alpha \Rightarrow \alpha
\]

\[
\text{int} \rightarrow \text{char} : \text{int (valid Unicode code-point)} \Rightarrow \text{char}
\]

\[
\text{char} \rightarrow \text{int} : \text{char} \Rightarrow \text{int}
\]

\[
\text{block?} \quad \text{int?} \quad \text{char?} \quad \text{bool?} \quad \text{unit?:} \forall \alpha. \alpha \Rightarrow \text{bool}
\]
Valid primitive arguments

byte-read : ⇒ int between 0 and 255, or -1
byte-write : ∃α. int between 0 and 255 ⇒ α
block-alloc-n : int ⇒ block
block-tag block-length : block ⇒ int
block-get : ∀α. block × int ⇒ α
block-set! : ∀α ∃β. block × int × α ⇒ β
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)
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 })
L₃ EBNF grammar (3)

str ::= "\{any character except newline\}"
chr ::= 'any character'
bool ::= #t | #f
unit ::= #u
ident ::= identstart { identstart | digit } [@ digit { digit }]
identstart ::= a | ... | z | A | ... | Z | | | | % | & | * | + | – | . | / | : | < | = | > | ? | ^ | _ | ~
prim-name ::= block-tag | block-alloc-n | etc.

0 ≤ n < 200
L₃ EBNF grammar (4)

num ::= num₂ | num₁₀ | num₁₆
num₂ ::= #b [-] digit₂ { digit₂ }
num₁₀ ::= [-] digit₁₀ { digit₁₀ }
num₁₆ ::= #x [-] digit₁₆ { digit₁₆ }
digit₂ ::= 0 | 1
digit₁₀ ::= digit₂ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit₁₆ ::= digit₁₀ | A | B | C | D | E | F | a | b | c | d | e | f
Write the L₃ version of the factorial function, defined as:

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

What does the following (valid) L₃ program compute?

\[
((\text{fun } (f \ x) (f \ x)) \\
(\text{fun } (x) (@+ \ x \ 1)) \\
20)
\]
$L_3$ syntactic sugar
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:

\[(\text{if } e_1 e_2) \leftrightarrow (\text{if } e_1 e_2 \#u)\]
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 $[\cdot]$ taking an $L_3$ term and producing a desugared $CL_3$ term ($CL_3$ is Core $L_3$, the desugared version of $L_3$). To clarify the presentation, $L_3$ terms appear in orange, $CL_3$ terms in green, and meta-terms in black.
To simplify the specification of desugaring for whole programs, we assume that all top-level expressions are wrapped sequentially in a single \((\text{program } \ldots)\) expression.

\[
[(\text{program } (\text{def } n \ e) \ s_1 \ s_2 \ \ldots)] = \\
(\text{let } ((n \ [e])) [(\text{program } s_1 \ s_2 \ \ldots)])
\]

\[
[(\text{program } (\text{defrec } n \ e) \ s_1 \ s_2 \ \ldots)] = \\
(\text{letrec } ((n \ [e])) [(\text{program } s_1 \ s_2 \ \ldots)])
\]

\[
[(\text{program } e \ s_1 \ s_2 \ \ldots)] = \\
[(\text{begin } e \ (\text{program } s_1 \ s_2 \ \ldots))]
\]

\[
[(\text{program } e)] = \\
[e]
\]
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.

\[
\begin{align*}
(\text{begin } b_1 b_2 b_3 ...) & = \\
(\text{let } (t [b_1]) (\text{begin } b_2 b_3 ...)) & = \\
(\text{begin } b) & = \\
[b] & =
\end{align*}
\]
\[\begin{align*}
\llbracket (\text{let } ((n_1 e_1) \ldots) b_1 b_2 \ldots) \rrbracket &= \\
&= (\text{let } ((n_1 [e_1]) \ldots) \llbracket (\text{begin } b_1 b_2 \ldots) \rrbracket) \\
\llbracket (\text{let* } ((n_1 e_1) (n_2 e_2) \ldots) b_1 b_2 \ldots) \rrbracket &= \\
&= \llbracket (\text{let } ((n_1 e_1)) (\text{let* } ((n_2 e_2) \ldots) b_1 b_2 \ldots)) \rrbracket \\
\llbracket (\text{let* } () b_1 b_2 \ldots) \rrbracket &= \\
&= \llbracket (\text{begin } b_1 b_2 \ldots) \rrbracket \\
\llbracket (\text{letrec } ((f_1 (\text{fun } (n_{1,1} \ldots) b_{1,1} b_{1,2} \ldots)) \ldots) b_1 b_2 \ldots) \rrbracket &= \\
&= (\text{letrec } ((f_1 (\text{fun } (n_{1,1} \ldots) \llbracket (\text{begin } b_{1,1} b_{1,2} \ldots) \rrbracket)) \ldots) \\
&\quad \llbracket (\text{begin } b_1 b_2 \ldots) \rrbracket)
\end{align*}\]
L₃ desugaring (4)

\[
\begin{align*}
[(\text{fun } (n₁ \ldots) b₁ b₂ \ldots)] &= \\
&\ \ (\text{letrec } ((f (\text{fun } (n₁ \ldots) [(\text{begin } b₁ b₂ \ldots)])) f) \\
[(\text{rec } n ((n₁ e₁) \ldots) b₁ b₂ \ldots)] &= \\
&\ \ (\text{letrec } ((n (\text{fun } (n₁ \ldots) [(\text{begin } b₁ b₂ \ldots)])) (n [e₁] \ldots))
\end{align*}
\]

\[
\begin{align*}
[(e e₁ \ldots)] &= \\
&\ \ ([e] [e₁] \ldots)
\end{align*}
\]

\[
\begin{align*}
[(\text{@ p } e₁ \ldots)] &= \\
&\ \ (@ p [e₁] \ldots)
\end{align*}
\]
\[
\begin{align*}
\text{L}_3 \text{ desugaring (5)}
\end{align*}
\]

\[
[\text{(if } e \text{ e}_1)] = \\
[\text{(if } e \text{ e}_1 \#u)]
\]

\[
[\text{(if } e \text{ e}_1 \text{ e}_2)] = \\
(\text{if } [e] [e_1] [e_2])
\]

\[
[\text{(cond } (e_1 \text{ b}_{1,1} \text{ b}_{1,2} \ldots) (e_2 \text{ b}_{2,1} \text{ b}_{2,2} \ldots) \ldots)] = \\
[\text{(if } e_1 (\text{begin } \text{ b}_{1,1} \text{ b}_{1,2}) (\text{cond } (e_2 \text{ b}_{2,1} \text{ b}_{2,2}) \ldots))]
\]

\[
[\text{(cond } ())] = \\
\#u
\]
L₃ desugaring (6)

\[
\left[(\text{and } e₁ e₂ e₃ \ldots)\right] = \\
\left[(\text{if } e₁ (\text{and } e₂ e₃ \ldots) \text{ #f})\right]
\]

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

\[
\left[(\text{or } e₁ e₂ e₃ \ldots)\right] = \\
\left[(\text{let } ((v \ e₁)) (\text{if } v v (\text{or } e₂ e₃ \ldots)))\right]
\]

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

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

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

```scheme
⟦"c₁…cₙ"⟧ =
  (let ((s (@block-alloc-200 n)))
    (@block-set! s 0 'c₁')
    ...
    s)

⟦l⟧ = if l is a (non-string) literal
    l

⟦n⟧ = if n is a name
    n
```

the (reserved) tag 200 is used for strings
L₃ desugaring example

⟦(program (@byte-write (if #t 79 75))
  (@byte-write (if #f 79 75))))⟧
= [((begin (@byte-write (if #t 79 75))
    (program
     (@byte-write (if #f 79 75')))))]
= (let ((t [(@byte-write (if #t 79 75'))])
    [(begin
      (program
       (@byte-write (if #f 79 75'))))])
= (let ((t (@byte-write (if #t 79 75)))
    (@byte-write (if #f 79 75)))
    (@byte-write (if #f 79 75)))
Exercise

Desugar the following L₃ expression:

\[
\text{(rec loop ((i 1))}
\text{(int-print i)}
\text{(if (< i 9)}
\text{(loop (+ i 1))))}
\]
The $L_3$ compiler
L₃ compiler architecture

Front-end

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

Back-end

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

+ 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.