The L₃ project

Advanced Compiler Construction
Michel Schinz – 2015-02-19
Project overview

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

```
(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)))))))
```

- $x^0 = 1$
- $x^{2z} = (x^z)^2$
- $x^{z+1} = x(x^z)$
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) (n₂ e₂) ...) b₁ ... bₖ)
Parallel local value definition. The expressions e₁, e₂, ... are evaluated in that order, and their values are then bound to names n₁, ... in the body b₁, ..., bₖ. The value of the whole expression is the value of bₖ.

(let* ((n₁ e₁) (n₂ e₂) ...) b₁ ... bₖ)
Sequential local value definition. Equivalent to a nested sequence of let: (let ((n₁ e₁)) (let ((n₂ e₂)) ...))

(letrec ((n₁ f₁) (n₂ f₂) ...) b₁ ... bₖ)
Recursive local function definition. The function expressions f₁, f₂, ... are evaluated and bound to names n₁, n₂, ... in the body b₁, ..., bₖ. The functions can be mutually recursive.
(fun (n₁ n₂ ...) b₁ ... bₖ)
Anonymous function with arguments n₁, n₂ ... and body b₁, ..., bₖ. The return value of the function is the value of bₖ.

(e e₁ e₂ ...)
Function application. Expressions e, e₁, e₂, ... are evaluated in that order, and then the value of e – which must be a function – is applied to the value of e₁, e₂, ...
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 e₁) (c₂ e₂) ...)
N-ways conditional. If c₁ evaluates to a true value, evaluate e₁; else, if c₂ evaluates to a true value, evaluate e₂; 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

(and e₁ e₂)
   Equivalent to (if e₁ e₂ #f).
(or e₁ e₂)
   Equivalent to (let ((v e₁)) (if v v e₂)), where v is a fresh name.
(not e)
   Equivalent to (if e #f #t).
Loops and blocks

\[(\text{rec } n ((n_1 e_1) (n_2 e_2) ... ) b_1 b_2 ... )\]

General loop. Equivalent to:

\[(\text{letrec } ((n (\text{fun } (n_1 n_2 ... ) b_1 b_2 ... ))) )\]

\[(n e_1 e_2 ... ))\]

\[(\text{begin } b_1 b_2 ... b_k)\]

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.
"c₁…cₙ"
  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.
(\* 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.

\( L_3 \) offers the following primitives:

- integer: +, -, *, /, %, <, <=, >, >=, \text{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!
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 ≥ 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.)
Valid primitive arguments

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

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

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

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

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

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

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

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

char-read : \( \Rightarrow \) char
char-print : \( \exists \alpha. \) char \( \Rightarrow \) \( \alpha \)
block-alloc-n : int \( \Rightarrow \) block
block-tag block-length : block \( \Rightarrow \) int
block-get : \( \forall \alpha. \) block \( \times \) int \( \Rightarrow \) \( \alpha \)
block-set! : \( \forall \alpha \exists \beta. \) block \( \times \) int \times \( \alpha \Rightarrow \) \( \beta \)

the return value is arbitrary
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, $L_3$ “has no syntax”, in that its concrete syntax is very close to its abstract syntax. For example, the $L_3$ 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.

$$
\texttt{(if } (@ < x \ 0) \\
\quad (@ - 0 x) \\
\quad x) 
$$
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 expr) { (expr expr) })
and ::= (and expr expr)
or ::= (or expr expr)
not ::= (not expr)
app ::= (expr { expr })
prim ::= (@ prim-name { expr })
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
Exercise

Write the L₃ version of the factorial function, defined as:
\[
\text{fact}(0) = 1 \\
\text{fact}(n) = n \cdot \text{fact}(n - 1) \quad [\text{if } n > 0]
\]

What does the following (valid) L₃ program compute?

\[
((\text{fun } (f \ x) (f \ x)) \\
(\text{fun } (x) (@+ \ x \ 1)) \\
20)
\]
L₃ 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:

\[(\text{if } e₁ e₂) \equiv (\text{if } e₁ e₂ #u)\]
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 $\llbracket \cdot \rrbracket$ taking an $L_3$ term and producing a desugared $CL_3$ term.
To simplify the specification of desugaring for whole programs, we assume that all top-level expressions are wrapped sequentially in a single (program ...) expression.

\[
\begin{align*}
\llbracket (\text{program } (\text{def } n \ e) \ s_1 \ s_2 \ldots) \rrbracket &= \\
&= (\text{let } ((n \ \llbracket e \rrbracket)) \llbracket (\text{program } s_1 \ s_2 \ldots) \rrbracket)
\end{align*}
\]

\[
\begin{align*}
\llbracket (\text{program } \text{defrec } n \ e) \ s_1 \ s_2 \ldots) \rrbracket &= \\
&= (\text{letrec } ((n \ \llbracket e \rrbracket)) \llbracket (\text{program } s_1 \ s_2 \ldots) \rrbracket)
\end{align*}
\]

\[
\begin{align*}
\llbracket (\text{program } e \ s_1 \ s_2 \ldots) \rrbracket &= \\
&= \llbracket (\text{begin } e \ (\text{program } s_1 \ s_2 \ldots)) \rrbracket
\end{align*}
\]

\[
\begin{align*}
\llbracket (\text{program } e) \rrbracket &= \\
&= [e]
\end{align*}
\]
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.

\[
\begin{align*}
\&\left[ (\text{begin } b_1 b_2 b_3 \ldots) \right] = \\
&\quad \left( \text{let } ((t \left[ b_1 \right])) \left[ (\text{begin } b_2 b_3 \ldots) \right] \right) \\
\&\left[ (\text{begin } b) \right] = \\
&\quad \left[ b \right]
\end{align*}
\]
\[
\begin{aligned}
\text{L}_3 \text{ desugaring (3)}
\end{aligned}
\]

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

\[
\llbracket (\text{let}\star ((n_1 \; e_1) \; (n_2 \; e_2) \; \ldots) \; b_1 \; b_2 \; \ldots) \rrbracket =
\quad \llbracket (\text{let } ((n_1 \; e_1)) \; (\text{let}\star ((n_2 \; e_2) \; \ldots) \; b_1 \; b_2 \; \ldots)) \rrbracket
\]

\[
\llbracket (\text{letrec } ((f_1 \; (\text{fun } (n_{1,1} \; \ldots) \; b_{1,1} \; b_{1,2} \; \ldots)) \; \ldots) \; b_1 \; b_2 \; \ldots) \rrbracket =
\quad (\text{letrec } ((f_1 \; (\text{fun } (n_{1,1} \; \ldots) \llbracket (\text{begin } b_{1,1} \; b_{1,2} \; \ldots) \rrbracket)) \ldots)
\quad \llbracket (\text{begin } b_1 \; b_2 \; \ldots) \rrbracket
\]

L₃ desugaring (4)

\[
\begin{align*}
\[(\text{fun } (n₁ \ldots) b₁ b₂ \ldots)] &= \\
&\quad (\text{letrec } ((f (\text{fun } (n₁ \ldots) [(\text{begin } b₁ b₂ \ldots)]))) f) \\
\[(\text{rec } n ((n₁ e₁) \ldots) b₁ b₂ \ldots)] &= \\
&\quad (\text{letrec } ((n (\text{fun } (n₁ \ldots) [(\text{begin } b₁ b₂ \ldots)]))) (n [e₁] \ldots)) \\
\[(e e₁ \ldots)] &= \\
&\quad ([e] [e₁] \ldots) \\
\[(@ p e₁ \ldots)] &= \\
&\quad (@ p [e₁] \ldots)
\end{align*}
\]
L₃ desugaring (5)

\[
[(\text{if } e \ e_1)] = \\
[(\text{if } e \ e_1 \ #u)] \\
([(\text{if } e_1 e_2)] = \\
(\text{if } [e] [e_1] [e_2]) \\
([(\text{cond } (e_1 \ b_1) (e_2 \ b_2) (e_3 \ b_3) ...)] = \\
([(\text{if } e_1 \ b_1 (\text{cond } (e_2 \ b_2) (e_3 \ b_3) ...))] \\
([(\text{cond } (e \ b))] = \\
([(\text{if } e \ b)])
\]
\[\text{L}_3 \text{ desugaring (6)}\]

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

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

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

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

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

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

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

```scheme
[[program (char-print (if #t 'o' 'k'))
(char-print (if #f 'o' 'k'))]]
= [[begin (char-print (if #t 'o' 'k'))
(program
 (char-print (if #f 'o' 'k')))]]
= (let ((t [[(char-print (if #t 'o' 'k'))]])
[[begin
 (program
 (char-print (if #f 'o' 'k')))]])
= (let ((t (char-print (if #t 'o' 'k')))
 (char-print (if #f 'o' 'k'))))
```
Exercise

Desugar the following L₃ expression:

```
(rec loop ((i 1))
  (int-print i)
  (if (< i 9)
    (loop (+ i 1)))
```
The $L_3$ compiler
+ interpreters for CL₃, CPS and ASM languages
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.