The L₃ project
Advanced Compiler Construction
Michel Schinz – 2013-02-21

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 xⁿ for x ∈ ℤ, y ∈ ℕ:

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

x⁰ = 1

x² = (x¹)²

x²⁺¹ = x(x²)

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.

Functions

(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, else e₃ is evaluated. The value of the whole expression is the value of the evaluated branch.

(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.

Logical expressions

(and e₁ e₂)
Equivalent to (if e₁ e₂ #f).
(or e₁ e₂)
Equivalent to (let ((v₁ e₁)) (if v₁ v₁ e₂)), where v₁ is a fresh name.
(not e)
Equivalent to (if e #f #t).

Loops and blocks

(rec n ((n₁ e₁) (n₂ e₂) ...) b₁ b₂ ...)
General loop. Equivalent to:
(letrec ((n (fun (n₁ n₂ ...) b₁ b₂ ...) ))
(n e₁ e₂ ...) )

(begin b₁ b₂ ... bₖ)
Sequential evaluation. First evaluate expression b₁, discarding its value, then b₂, etc. Finally evaluate bₖ, whose value is the value of the whole expression.

Literal values

"c₁c₂...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.
Primitives

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

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

$L_3$ offers the following primitives:
- integer: $+$ $-$ $*$ $/$ $\%$ $<=$ $>$ $=>$ int $\rightarrow$ char
- polymorphic comparison: $\neq$
- type tests: $\text{block}?$ $\text{int}?$ $\text{char}?$ $\text{bool}?$ $\text{unit}?$
- character: $\text{char-read}$ $\text{char-print}$ $\text{char} \rightarrow$ int
- tagged blocks (see later): $\text{block-alloc-n}$ $\text{block-tag}$ $\text{block-length}$ $\text{block-get}$ $\text{block-set!}$

Valid primitive arguments

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

- $+$ $-$ $*$ $/$ $\%$ : int $\times$ int $\Rightarrow$ int
- $<=$ $>$ $=>$ : int $\times$ int $\Rightarrow$ bool
- $\neq$ : $\forall \alpha, \beta. \alpha \times \beta \Rightarrow$ bool
- int $\rightarrow$ char : int $\Rightarrow$ char
- char $\rightarrow$ int : char $\Rightarrow$ int
- block? int char bool unit? : $\forall \alpha. \alpha \Rightarrow$ bool
- char-read : $\Rightarrow$ char
- char-print : char $\Rightarrow$ unit

Tagged blocks

$L_3$ offers a single structured datatype: tagged blocks. They are manipulated with the following primitives:

- $\text{block-alloc-n}$ s : int $\Rightarrow$ block
- $\text{block-tag}$ block-length : block $\Rightarrow$ int
- block-get : $\forall \alpha. \text{block} \times \text{int} \Rightarrow \alpha$
- block-set! : $\forall \alpha. \text{block} \times \text{int} \times \alpha \Rightarrow$ unit

- Allocates an uninitialized block with tag $n$ and length $s$.
- Returns the tag of block $b$ (as an integer).
- Returns the length of block $b$.
- Returns the $n^{\text{th}}$ element (0-based) of block $b$.
- Sets the $n^{\text{th}}$ element (0-based) of block $b$ to $v$. 
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.)

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.

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)
```

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 ::= function? | block-alloc-n | block?
            | block-tag | block-length | block-get | block-set!
            | int? | + | - | * | / | % | < | <= | = | != | >= | >
            | char-read | char-print | bool? | not | unit?

Exercises

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

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

What does the following (valid) L₃ program compute?

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

L₃ syntactic sugar

L₃ 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:

\[
(\text{if } e₁ \ e₂) \leftrightarrow (\text{if } e₁ \ e₂ \ #u)
\]
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 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:

\[
\text{if } e_1 \text{ } e_2 \Rightarrow ds \text{ (if } e_1 \text{ } e_2 \#u)
\]

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

\[
\text{program ... (def } n \text{ e) b} \\
\Rightarrow ds \text{ (program ... (let } ((n e)) \text{ b))}
\]

\[
\text{program ... (defrec } n \text{ f) b} \\
\Rightarrow ds \text{ (program ... (letrec } ((n f)) \text{ b))}
\]

\[
\text{program ... } e_1 \text{ } e_2 \\
\Rightarrow ds \text{ (program ... (begin } e_1 \text{ } e_2))
\]

\[
\text{let* } ((n_1 \text{ e}_1)) \text{ b) } \\
\Rightarrow ds \text{ (let } ((n_1 \text{ e}_1)) \text{ b)}
\]

\[
\text{let* } ((n_1 \text{ e}_1) \text{ (n}_2 \text{ e}_2) \text{ ... b)} \\
\Rightarrow ds \text{ (let } ((n_1 \text{ e}_1)) \text{ (let* } ((n_2 \text{ e}_2) \text{ ... b)}
\]

\[
\text{let* } ((n_1 \text{ e}_1) \text{ ... b}_1 \text{ ...)} \\
\Rightarrow ds \text{ (let* } ((n_1 \text{ e}_1) \text{ ... (begin b}_1 \text{ ...}))
\]

\[
\text{let } ((n_1 \text{ e}_1) \text{ ... b}_1 \text{ ...)} \\
\Rightarrow ds \text{ (let } ((n_1 \text{ e}_1) \text{ ... (begin b}_1 \text{ ...))}
\]

\[
\text{letrec } ((\text{n}_1 \text{ f}_1) \text{ ... b}_1 \text{ ...)} \\
\Rightarrow ds \text{ (letrec } ((\text{n}_1 \text{ f}_1) \text{ ... (begin b}_1 \text{ ...))}
\]

\[
\text{L3 desugaring (1)}
\]

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

\[
\text{program ... (def } n \text{ e) b} \\
\Rightarrow ds \text{ (program ... (let } ((n e)) \text{ b))}
\]

\[
\text{program ... (defrec } n \text{ f) b} \\
\Rightarrow ds \text{ (program ... (letrec } ((n f)) \text{ b))}
\]

\[
\text{program ... } e_1 \text{ } e_2 \\
\Rightarrow ds \text{ (program ... (begin } e_1 \text{ } e_2))
\]

\[
\text{L3 desugaring (2)}
\]

\[
\text{let* } ((n_1 \text{ e}_1)) \text{ b) } \\
\Rightarrow ds \text{ (let } ((n_1 \text{ e}_1)) \text{ b)}
\]

\[
\text{let* } ((n_1 \text{ e}_1) \text{ (n}_2 \text{ e}_2) \text{ ... b)} \\
\Rightarrow ds \text{ (let } ((n_1 \text{ e}_1)) \text{ (let* } ((n_2 \text{ e}_2) \text{ ... b)}
\]

\[
\text{let* } ((n_1 \text{ e}_1) \text{ ... b}_1 \text{ ...)} \\
\Rightarrow ds \text{ (let* } ((n_1 \text{ e}_1) \text{ ... (begin b}_1 \text{ ...}))
\]

\[
\text{let } ((n_1 \text{ e}_1) \text{ ... b}_1 \text{ ...)} \\
\Rightarrow ds \text{ (let } ((n_1 \text{ e}_1) \text{ ... (begin b}_1 \text{ ...))}
\]

\[
\text{letrec } ((\text{n}_1 \text{ f}_1) \text{ ... b}_1 \text{ ...)} \\
\Rightarrow ds \text{ (letrec } ((\text{n}_1 \text{ f}_1) \text{ ... (begin b}_1 \text{ ...))}
\]

\[
\text{L3 desugaring (3)}
\]

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

\[
\text{fun } ((n_1 \text{ ...}) \text{ b}_1 \text{ b}_2 \text{ ...)} \\
\Rightarrow ds \text{ (letrec } ((\text{f } \text{ fun# } ((n_1 \text{ ...}) \text{ b}_1 \text{ b}_2 \text{ ...})) \text{ f)}
\]

\[
\text{rec } n (((n_1 \text{ e}_1) \text{ (n}_2 \text{ e}_2) \text{ ... b}_1 \text{ b}_2 \text{ ...)} \\
\Rightarrow ds \text{ (letrec } (((n \text{ fun# } ((n_1 \text{ n}_2 \text{ ...}) \text{ b}_1 \text{ b}_2 \text{ ...))) \text{ (n e}_1 \text{ e}_2 \text{ ...))}
\]

\[
\text{fun# } ((n_1 \text{ ...}) \text{ b}_1 \text{ b}_2 \text{ ...)} \\
\Rightarrow ds \text{ (fun# } ((n_1 \text{ ...}) \text{ (begin b}_1 \text{ b}_2 \text{ ...))}
\]
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 \( \_ \).

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

For example, if \( C \) is \( (\text{if} \ _ \ _ \ 1 \ 2) \), then \( C\{(\lt \ x \ y)\} \) is \( (\text{if} \ (\lt \ x \ y) \ 1 \ 2) \).
Desugaring contexts

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

\[
\begin{align*}
C_{ds} & ::= \square \\
& | \text{program } C_{ds} \\
& | (\text{let } ((n_1 e_1) \ldots (n_l e_l)) \rightarrow (n_k e_k)) \rightarrow C_{ds} \\
& | (\text{letrec } ((n_1 f_1) \ldots (n_l e_l)) \rightarrow C_{ds} \\
& | (\text{letrec } ((n_1 f_1) \ldots (n_k f_k)) \rightarrow C_{ds} \\
& | (\text{fun}\#(n_1 \ldots n_k) C_{ds} \\
& | (\text{if } C_{ds} e_1 e_2 e_3) \\
& | (\text{if } e_1 C_{ds} e_2 e_3) \\
& | (\text{if } e_1 e_2 C_{ds} \\
& | (C_{ds} e_1 \ldots e_l) \\
& | (e e_1 \ldots C_{ds} \ldots e_k) \\
& | (@p e_1 \ldots C_{ds} \ldots e_k)
\end{align*}
\]

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:

\[
\begin{align*}
C_{ds}(T) & \Rightarrow_{ds} C_{ds}(T') \text{ where } T \Rightarrow_{ds} T'
\end{align*}
\]

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

L₃ desugaring example

\[
\begin{align*}
\text{(program } (\text{char-print (if #t 'o' 'k')) }) \\
\text{(program } (\text{char-print (if #f 'o' 'k')) })
\end{align*}
\]

\[
\begin{align*}
\Rightarrow_{ds}(\text{program}} \\
\text{(begin (char-print (if #t 'o' 'k'))}) \\
\text{(char-print (if #f 'o' 'k'))})
\end{align*}
\]

\[
\begin{align*}
\Rightarrow_{ds}(\text{program}} \\
\text{(let ((t (char-print (if #t 'o' 'k'))))}) \\
\text{(char-print (if #f 'o' 'k'))})
\end{align*}
\]

\[
\Rightarrow_{ds}
\] cannot be rewritten further

L₃ desugaring exercise

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

\[
\begin{align*}
\text{(rec loop ((i 1))} \\
\text{(int-print i)} \\
\text{(if (< i 9) } \\
\text{(loop (+ i 1))))}
\end{align*}
\]
The L₃ compiler

Intermediate languages

The L₃ compiler manipulates a total of four languages:
1. L₃ is the source language that is parsed,
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.