The L₃ project

Advanced Compiler Construction
Michel Schinz - 2014-02-20
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 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 $L_3$

An $L_3$ function to compute $x^y$ for $x \in \mathbb{Z}$, $y \in \mathbb{N}$:

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

- $x^0 = 1$
- $x^{2z} = (x^z)^2$
- $x^{z+1} = x(x^z)$
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 ((n_1 e_1) (n_2 e_2) ...) b_1 ... b_k)
Parallel local value definition. The expressions e_1, e_2, ... are evaluated in that order, and their values are then bound to names n_1, ... in the body b_1, ..., b_k. The value of the whole expression is the value of b_k.

(let* ((n_1 e_1) (n_2 e_2) ...) b_1 ... b_k)
Sequential local value definition. Equivalent to a nested sequence of let: (let ((n_1 e_1)) (let ((n_2 e_2)) ...))

(letrec ((n_1 f_1) (n_2 f_2) ...) b_1 ... b_k)
Recursive local function definition. The function expressions f_1, f_2, ... are evaluated and bound to names n_1, n_2, ... in the body b_1, ..., b_k. The functions can be mutually recursive.
Anonymous function with arguments $n_1$, $n_2$ ... and body $b_1$, ..., $b_k$. The return value of the function is the value of $b_k$.

Function application. Expressions $e$, $e_1$, $e_2$, ... are evaluated in that order, and then the value of $e$ - which must be a function - is applied to the value of $e_1$, $e_2$, ...
(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.
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 \ e_1 \ e_2)
Equivalent to (if \ e_1 \ e_2 \ #f).

(or \ e_1 \ e_2)
Equivalent to (let \ ((v \ e_1)) \ (if \ v \ v \ e_2)), 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.
Literal values

"c_1c_2...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.
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!
Valid primitive arguments

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

- `+ - * / % << >> & | ^` : `int * int` \(\Rightarrow\) `int`
- `< <= > >=` : `int * int` \(\Rightarrow\) `bool`
- `=` `!=` : `\forall \alpha, \beta. \alpha * \beta` \(\Rightarrow\) `bool`
- `int->char` : `int` \(\Rightarrow\) `char`
- `char->int` : `char` \(\Rightarrow\) `int`
- `block? int? char? bool? unit?` : `\forall \alpha. \alpha` \(\Rightarrow\) `bool`
- `char-read` : `\Rightarrow` `char`
- `char-print` : `char` \(\Rightarrow\) `unit`
Valid primitive arguments

\[
\begin{align*}
\text{block-alloc-n} & : \text{int} \Rightarrow \text{block} \\
\text{block-tag block-length} & : \text{block} \Rightarrow \text{int} \\
\text{block-get} & : \forall \alpha. \text{block*int} \Rightarrow \alpha \\
\text{block-set!} & : \forall \alpha. \text{block*int*alpha} \Rightarrow \text{unit}
\end{align*}
\]
Tagged blocks

$L_3$ 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 $n^{th}$ element (0-based) of block $b$.

**(@ block-set! b n v)**

Sets the $n^{th}$ 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.)
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.

\[
\text{(if}\ (@ < x \ 0)\ (@ - 0 x)\ x)\]

\[
\text{if}\ @ \text{ident}(x)\ \text{ident}(x)\ \text{int}(0)\ \text{-}\ \text{int}(0)\ \text{ident}(x)\ \text{ident}(x)\ \text{int}(0)\ x)
\]
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)
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 })
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.
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`:

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

\( (\text{if } e_1 e_2) \rightarrow_{ds} (\text{if } e_1 e_2 \#u) \)
To simplify the rewriting rules for whole programs, we assume that all top-level expressions are wrapped sequentially in a \((\text{program} \ldots )\) expression.

\[(\text{program} \ldots (\text{def } n e b)) \Rightarrow_{ds} (\text{program} \ldots (\text{let } ((n e)) b))\]

\[(\text{program} \ldots (\text{defrec } n f b)) \Rightarrow_{ds} (\text{program} \ldots (\text{letrec } ((n f)) b))\]

\[(\text{program} \ldots e_1 e_2) \Rightarrow_{ds} (\text{program} \ldots (\text{begin } e_1 e_2))\]
(let* ((n₁ e₁)) b)
⇝ds (let ((n₁ e₁)) b)

(let* ((n₁ e₁) (n₂ e₂) ...) b)
⇝ds (let ((n₁ e₁)) (let* ((n₂ e₂) ...) b)
(let* ((n₁ e₁) ...) b₁ ...)
⇝ds (let* ((n₁ e₁) ...) (begin b₁ ...))

(let ((n₁ e₁) ...) b₁ ...)
⇝ds (let ((n₁ e₁) ...) (begin b₁ ...))

(letrec ((n₁ f₁) ...) b₁ ...)
⇝ds (letrec ((n₁ f₁) ...) (begin b₁ ...))
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 (#).

\[
\begin{align*}
\text{(fun } (n_1 \ldots) b_1 b_2 \ldots) & \overset{\text{underlined names are fresh}}{\mapsto}_{ds} \text{(letrec } ((f \ (\text{fun# } (n_1 \ldots) b_1 b_2 \ldots))) \ f) \\
\text{(rec } n \ ((n_1 \ e_1) \ (n_2 \ e_2) \ldots) b_1 b_2 \ldots) & \overset{\text{ds}}{\mapsto} \text{(letrec } ((n \ (\text{fun# } (n_1 n_2 \ldots) b_1 b_2 \ldots))) \\
& \quad (n \ e_1 \ e_2 \ldots)) \\
\text{(fun# } (n_1 \ldots) b_1 b_2 \ldots) & \overset{\text{ds}}{\mapsto} \text{(fun# } (n_1 \ldots) \ (\text{begin } b_1 b_2 \ldots))
\end{align*}
\]
(begin e)
    \(\Rightarrow_{ds} e\)

(begin e₁ e₂ ...)
    \(\Rightarrow_{ds} (let ((n e₁)) (begin e₂ ...))\)

(if c e)
    \(\Rightarrow_{ds} (if c e \#u)\)

(cond (c₁ e₁))
    \(\Rightarrow_{ds} (if c₁ e₁)\)

(cond (c₁ e₁) (c₂ e₂) ...)
    \(\Rightarrow_{ds} (if c₁ e₁ (cond (c₂ e₂) ...))\)
L₃ desugaring (5)

(\text{and}\ e_1\ e_2)
\quad \mapsto_{ds} (\text{if}\ e_1\ e_2\ \#f)

(\text{or}\ e_1\ e_2)
\quad \mapsto_{ds} (\text{let}\ ((v\ e_1))\ (\text{if}\ v\ v\ e_2))

(\text{not}\ e)
\quad \mapsto_{ds} (\text{if}\ e\ \#f\ \#t)
L₃ desugaring (6)

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

"c₁c₂…cₙ"

$\mapsto_{\text{ds}} \begin{array}{l}
\text{(let ((s (@block-alloc-200 n)))}
\text{(@block-set! s 0 'c₁')}
\text{(@block-set! s 1 'c₂')}
\text{...}
\text{s)}
\end{array}$

the (reserved) tag 200 is used for strings
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 $(\text{if } \square \ 1 \ 2)$, then $C\{(\text{< } x \ y)\}$ is $(\text{if } (\text{< } x \ y) \ 1 \ 2)$. 
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:

$$C_{ds} ::= \square$$

$$| (\text{program } C_{ds})$$

$$| (\text{let } ((n_1 \ e_1) \ldots (n_i C_{ds}) \ldots (n_k e_k)) \ e)$$

$$| (\text{let } ((n_1 \ e_1) \ldots (n_k e_k)) C_{ds})$$

$$| (\text{letrec } ((n_1 f_1) \ldots (n_i C_{ds}) \ldots (n_k f_k)) \ e)$$

$$| (\text{letrec } ((n_1 f_1) \ldots (n_k f_k)) C_{ds})$$

$$| (\text{fun} \# (n_1 \ldots n_k) C_{ds})$$

$$| (\text{if } C_{ds} e_2 e_3)$$

$$| (\text{if } e_1 C_{ds} e_3)$$

$$| (\text{if } e_1 e_2 C_{ds})$$

$$| (C_{ds} e_1 \ldots e_k)$$

$$| (e e_1 \ldots C_{ds} \ldots e_k)$$

$$| (@ p \ e_1 \ldots C_{ds} \ldots e_k)$$
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:

\[ C_{ds}\{T\} \Rightarrow_{ds} C_{ds}\{T'\} \text{ where } T \rightsquigarrow_{ds} T' \]

Completely desugaring a program amounts to reducing it using the desugaring relation until it cannot be reduced further.
L₃ desugaring example

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

⇒₃ds (program  
(begin (char-print (if #t 'o' 'k')))  
(char-print (if #f 'o' 'k')))  

⇒₃ds (program  
(let ((t (char-print (if #t 'o' 'k'))))  
(begin (char-print (if #f 'o' 'k')))))  

⇒₃ds (program  
(let ((t (char-print (if #t 'o' 'k'))))  
(char-print (if #f 'o' 'k'))))  

⇒₃ds (program  
(let ((t (char-print (if #t 'o' 'k'))))  
(char-print (if #f 'o' 'k'))))  

can't 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:

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

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

Back-end
- CPS converter
  - CL₃ tree
- Data representer
  - CPS tree
- Register allocator
  - CPS tree
- ASM converter
  - CPS tree
- ASM file

+ 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.