The L_3 project

Advanced Compiler Construction
Michel Schinz – 2021–02–25

Project overview

As the semester progresses, you will get:
- parts of an L_3 compiler written in Scala, and
- parts of a virtual machine, written in C.

You will have to:
- do one non-graded, warm-up exercise,
- complete the compiler,
- complete the virtual machine.

The L_3 language

The L_3 language

L_3 is a **L**isp-**l**ike **l**anguage. 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 is simple but quite powerful.

A taste of L_3

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

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

$x^0 = 1$

$x^{2z} = (x^z)^2$

$x^{z+1} = x(x^z)$

Values

L_3 offers four types of atomic values:

1. unit,
2. booleans,
3. characters, represented by their Unicode code point,
4. integers, 31 bits [!] in two's complement.

and one type of composite value: tagged blocks.

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 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 e_1, … are evaluated in that order, and their values are then bound to names n_1, … in the body b_1, b_2, … The value of the whole expression is the value of the last b_i.

`(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 f_1, … are evaluated and bound to names n_1, … in the body b_1, b_2 … The functions can be mutually recursive.

Conditional expressions

`(if e₁ e₂ e₃)`

Two-ways conditional. If e_1 evaluates to a true value (i.e. anything but `#f`), e_2 is evaluated, otherwise e_3 is evaluated. The value of the whole expression is the value of the evaluated branch.

The else branch, e_3, is optional and defaults to `#u` (unit).

`(cond (c₁ b₁,₁ b₁,₂ …) (c₂ b₂,₁ b₂,₂ …) …)`

N-ways conditional. If c_1 evaluates to a true value, evaluate $b_{1,1}$, $b_{1,2}$ …; else, if c_2 evaluates to a true value, evaluate $b_{2,1}$, $b_{2,2}$ …; 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 e₃ …)`

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.

`(or e₁ e₂ e₃ …)`

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.

`(not e)`

Negation. If e evaluates to a true value, produce the value `#f`. Otherwise, produce the value `#t`.

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 b_1, discarding its value, then b_2, etc. The value of the whole expression is the value of the last b_i.

Functions and primitives

`(fun (n₁ …) b₁ b₂ …)`
Anonymous function with arguments n_1, … and body b_1, b_2, … The return value is the value of the last b_i.

`(e e₁ …)`
Function application. Expressions e, e_1, … are evaluated in order, and then the value of e – which must be a function – is applied to the value of e_1, …
Note: if e is a simple identifier, a special form of name resolution, based on arity, is used – see later.

`(@ p e₁ e₂ …)`
Primitive application. First evaluate expressions e_1, e_2, … in that order, and then apply primitive p to the value of these expressions.

Arity-based name lookup

A special name lookup rule is used when analysing a function application in which the function is a simple name:
`(n e₁ e₂ … eₖ)`
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)
```


Primitives

L_3 offers the following primitives:

- integer: `< <= + - * / %` truncated division/remainder
- integer: `shift-left shift-right and or xor`
- polymorphic: `= id` identity
- type tests: `block? int? char? bool? unit?`
- character: `char->int int->char`
- I/O: `byte-read byte-write`
- tagged blocks: `block-alloc-`*n* $0 \le n \le 255$
  `block-tag block-length block-get block-set!`

Tagged blocks

L_3 offers a single kind of composite values: tagged blocks. They are manipulated with the following primitives:

`(@ block-alloc-`*n* `s)`
Allocates an uninitialised 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.

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 L_3 library uses a few tags to represent arrays, lists, etc.)

Valid primitive arguments

Primitives only work correctly when applied to certain arguments, otherwise their behaviour is undefined.

`+ - * and or xor` : *int* × *int* ⇒ *int*

`shift-left shift-right` : *int* × (*int* ∈ {0, 1, …, 31}) ⇒ *int*

`/ %` : *int* × (*int* ≠ 0) ⇒ *int*

`< <=` : *int* × *int* ⇒ *bool*

`=` : ∀α, β. α × β ⇒ *bool*

`id` : ∀α. α ⇒ α

`int->char` : *int* ∈ { valid Unicode code-points } ⇒ *char*

`char->int` : *char* ⇒ *int*

Valid primitive arguments

`block? int? char? bool? unit?` : ∀α. α ⇒ *bool*

`byte-read` : ⇒ *int* ∈ { -1, 0, 1, …, 255 }

`byte-write` : *int* ∈ { 0, 1, …, 255 } ⇒ ?

arbitrary return value

`block-alloc-`*n* : *int* ⇒ *block*

`block-tag block-length` : *block* ⇒ *int*

`block-get` : ∃α. *block* × *int* ⇒ α

`block-set!` : ∀α. *block* × *int* × α ⇒ ?

Undefined behaviour

The fact that primitives have undefined behaviour when applied to invalid arguments means that they can do *anything* in such a case.

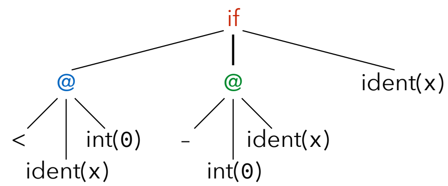
For example, division by zero can produce an error, crash the program, or produce an arbitrary value like 0.

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 parenthesised and tagged, while leaves are unadorned.

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



L_3 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)

L_3 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 })

L_3 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

L_3 EBNF grammar (4)

num ::= num$_2$ | num$_{10}$ | num$_{16}$
num$_2$ ::= `#b` digit$_2$ { digit$_2$ }
num$_{10}$ ::= [`-`] digit$_{10}$ { digit$_{10}$ }
num$_{16}$ ::= `#x` digit$_{16}$ { digit$_{16}$ }
digit$_2$ ::= `0` | `1`
digit$_{10}$ ::= digit$_2$ | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
digit$_{16}$ ::= digit$_{10}$ | `A` | `B` | `C` | `D` | `E` | `F` | `a` | `b` | `c` | `d` | `e` | `f`

Exercise

Write the L_3 version of the factorial function, defined as:
 fact(0) = 1
 fact(n) = n · fact(n - 1) [if n > 0]
What does the following (valid) L_3 program compute?

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


L_3 syntactic sugar

L_3 syntactic sugar

L_3 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, L_3 allows `if` expressions without an else branch, which is implicitly taken to be the unit value `#u`:
`(if` e_1 e_2`)` ⇔ `(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 ⟦·⟧ 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.

L_3 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`)` s_1 s_2 …`)`⟧ =
  `(let ((`n ⟦e⟧`))` ⟦`(program` s_1 s_2 …`)`⟧`)`
⟦`(program (defrec` n e`)` s_1 s_2 …`)`⟧ =
  `(letrec ((`n ⟦e⟧`))` ⟦`(program` s_1 s_2 …`)`⟧`)`
⟦`(program` e s_1 s_2 …`)`⟧ =
  ⟦`(begin` e `(program` s_1 s_2 …`))`⟧
⟦`(program` e`)`⟧ =
  ⟦e⟧

L_3 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` b_1 b_2 b_3 …`)`⟧ =
  `((let ((`<u>t</u> ⟦b_1⟧`))` ⟦`(begin` b_2 b_3 …`)`⟧`)`
⟦`(begin` b`)`⟧ =
  ⟦b⟧

L$_3$ desugaring (3)

⟦(let ((n$_1$ e$_1$) …) b$_1$ b$_2$ …)⟧ =
  (let ((n$_1$ ⟦e$_1$⟧) …) ⟦(begin b$_1$ b$_2$ …)⟧)
⟦(let* ((n$_1$ e$_1$) (n$_2$ e$_2$) …) b$_1$ b$_2$ …)⟧ =
  ⟦(let ((n$_1$ e$_1$)) (let* ((n$_2$ e$_2$) …) b$_1$ b$_2$ …))⟧
⟦(let* () b$_1$ b$_2$ …)⟧ =
  ⟦(begin b$_1$ b$_2$ …)⟧
⟦(letrec ((f$_1$ (fun (n$_{1,1}$ …) b$_{1,1}$ b$_{1,2}$ …)) …) b$_1$ b$_2$ …)⟧ =
  (letrec ((f$_1$ (fun (n$_{1,1}$ …) ⟦(begin b$_{1,1}$ b$_{1,2}$ …)⟧))
        …)
    ⟦(begin b$_1$ b$_2$ …)⟧)

L$_3$ desugaring (4)

⟦(fun (n$_1$ …) b$_1$ b$_2$ …)⟧ =
  (letrec ((f (fun (n$_1$ …) ⟦(begin b$_1$ b$_2$ …)⟧)))
    f)
⟦(rec n ((n$_1$ e$_1$) …) b$_1$ b$_2$ …)⟧ =
  (letrec ((n (fun (n$_1$ …) ⟦(begin b$_1$ b$_2$ …)⟧)))
    (n ⟦e$_1$⟧ …))
⟦(e e$_1$ …)⟧ =
  (⟦e⟧ ⟦e$_1$⟧ …)
⟦(@ p e$_1$ …)⟧ =
  (@ p ⟦e$_1$⟧ …)

L$_3$ desugaring (5)

⟦(if e e$_1$)⟧ =
  ⟦(if e e$_1$ #u)⟧
⟦(if e e$_1$ e$_2$)⟧ =
  (if ⟦e⟧ ⟦e$_1$⟧ ⟦e$_2$⟧)
⟦(cond (e$_1$ b$_{1,1}$ b$_{1,2}$ …) (e$_2$ b$_{2,1}$ b$_{2,2}$ …) …)⟧ =
  ⟦(if e$_1$ (begin b$_{1,1}$ b$_{1,2}$ …) (cond (e$_2$ b$_{2,1}$ b$_{2,2}$ …) …))⟧
⟦(cond ())⟧ =
  #u

L$_3$ desugaring (6)

⟦(and e$_1$ e$_2$ e$_3$ …)⟧ =
  ⟦(if e$_1$ (and e$_2$ e$_3$ …) #f)⟧
⟦(and e)⟧ =
  ⟦e⟧
⟦(or e$_1$ e$_2$ e$_3$ …)⟧ =
  ⟦(let ((v e$_1$)) (if v v (or e$_2$ e$_3$ …)))⟧
⟦(or e)⟧ =
  ⟦e⟧
⟦(not e)⟧ =
  ⟦(if e #f #t)⟧

L_3 desugaring (7)

L_3 does not have a string type. It offers string literals, though, which are desugared to blocks of characters.

⟦"c_1…c_n"⟧ =

```
  ⟦(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

L_3 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)))
```

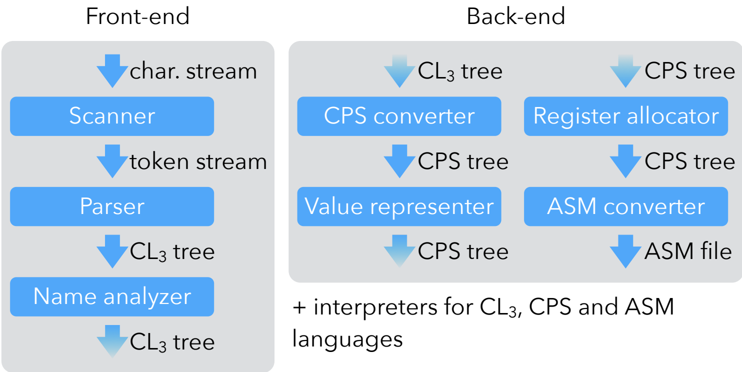

Exercise

Desugar the following L_3 expression :

```
(rec loop ((i 1))
   (int-print i)
   (if (< i 9)
       (loop (+ i 1))))
```


The L_3 compiler

L_3 compiler architecture



+ interpreters for CL_3, CPS and ASM languages

Note: CL_3, CPS and ASM each designate a *family* of very similar languages, with minor differences between them.

Intermediate languages

The L_3 compiler manipulates a total of four (families of) languages:

1. L_3 is the source language that is parsed, but never exists as a tree – it is desugared to CL_3 immediately,
2. CL_3 – a.k.a. $CoreL_3$ – is the desugared version of L_3,
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.