The L_3 project

Advanced Compiler Construction
Michel Schinz – 2013-02-21

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

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

$x^0 = 1$

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

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

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.

Functions

(**fun** (n_1 n_2 …) b_1 … b_k)

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.

(e e_1 e_2 …)

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, …

Conditional expressions

(**if** e_1 e_2 e_3)

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

(**cond** (c_1 e_1) (c_2 e_2) …)

N-ways conditional. If c_1 evaluates to a true value, evaluate e_1; else, if c_2 evaluates to a true value, evaluate e_2; etc. The value of the whole expression is the value of the evaluated branch.

Logical expressions

(**and** e_1 e_2)

Equivalent to (**if** e_1 e_2 **#f**).

(**or** e_1 e_2)

Equivalent to (**let** ((v_1 e_1)) (**if** v_1 v_1 e_2)), where v_1 is a fresh name.

(**not** e)

Equivalent to (**if** e #f #t).

Loops and blocks

(**rec** n ((n$_1$ e$_1$) (n$_2$ e$_2$) ...) b$_1$ b$_2$ ...)

General loop. Equivalent to:

(**letrec** ((n (**fun** (n$_1$ n$_2$ ...) b$_1$ b$_2$ ...)))
(n e$_1$ e$_2$ ...))

(**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 e_1 e_2 …)

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->char`
- 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 ⇒ int

`< <= > >=` : int * int ⇒ bool

`= !=` : ∀α, β. α * β ⇒ bool

`int->char` : int ⇒ char

`char->int` : char ⇒ int

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

`char-read` : ⇒ char

`char-print` : char ⇒ unit

Valid primitive arguments

`block-alloc-n` : int ⇒ block

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

`block-get` : ∀α. block*int ⇒ α

`block-set!` : ∀α. block*int*α ⇒ unit

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.

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.

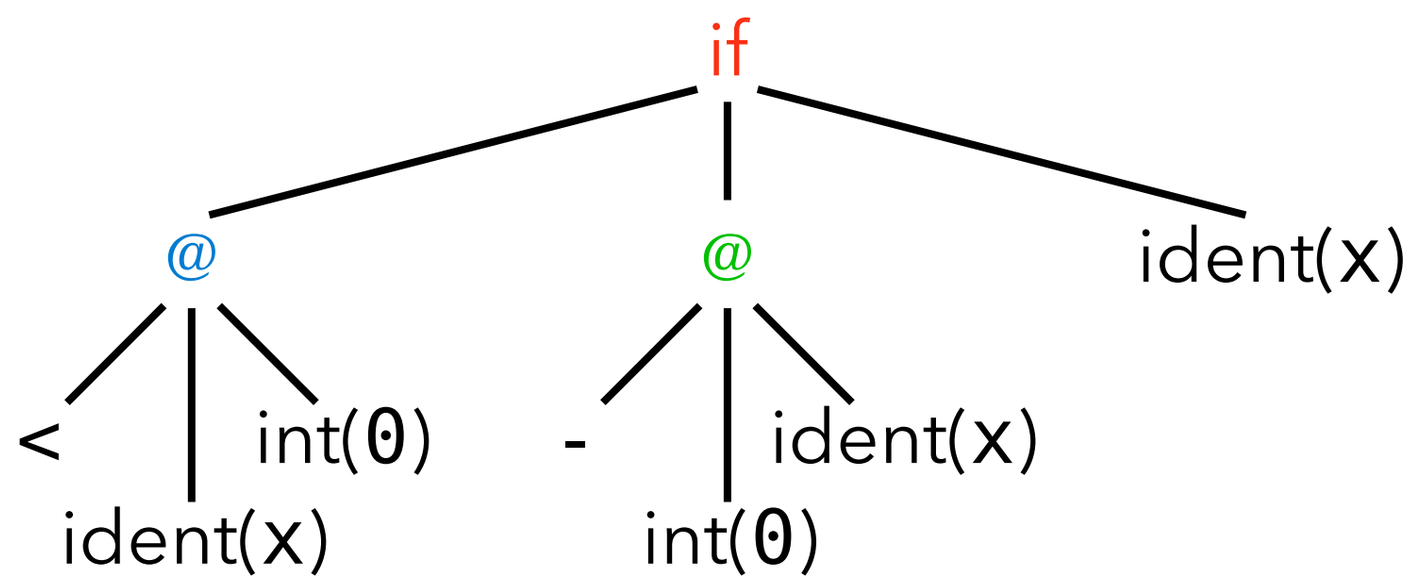
The valid tags range from 0 to 255, inclusive. Tags ≥ 200 are reserved by the compiler, while the others are available for general use. (For example, our L_3 library uses a few tags to represent arrays, lists, etc.)

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.

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



L_3 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`)`

L_3 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 })
```

L_3 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?

0 ≤ n ≤ 255

Exercises

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`:

$$(\textbf{if}\ e_1\ e_2) \Leftrightarrow (\textbf{if}\ e_1\ e_2\ \#u)$$

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 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:

$$(\textbf{if}\ e_1\ e_2) \rightsquigarrow_{ds} (\textbf{if}\ e_1\ e_2\ \textbf{\#u})$$

L_3 desugaring (1)

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

`(program` ... `(def` n e`)` b`)`

  $\rightsquigarrow_{ds}$ `(program` ... `(let ((`n e`))` b`))`

`(program` ... `(defrec` n f`)` b`)`

  $\rightsquigarrow_{ds}$ `(program` ... `(letrec ((`n f`))` b`))`

`(program` ... e_1 e_2`)`

  $\rightsquigarrow_{ds}$ `(program` ... `(begin` e_1 e_2`))`

L_3 desugaring (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 ...))

L_3 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 (#).

(fun (n$_1$ ...) b$_1$ b$_2$ ...)
  $\rightsquigarrow_{ds}$ (letrec ((f (fun$^{\#}$ (n1 ...) b$_1$ b$_2$ ...))) f)

(rec n ((n$_1$ e$_1$) (n$_2$ e$_2$) ...) b$_1$ b$_2$ ...)
  $\rightsquigarrow_{ds}$ (letrec ((n (fun$^{\#}$ (n$_1$ n$_2$ ...) b$_1$ b$_2$ ...))
        (n e$_1$ e$_2$ ...))

(fun$^{\#}$ (n$_1$ ...) b$_1$ b$_2$ ...)
  $\rightsquigarrow_{ds}$ (fun$^{\#}$ (n$_1$ ...) (begin b$_1$ b$_2$ ...))

underlined names are fresh

L_3 desugaring (4)

(begin e)

  $\rightsquigarrow_{ds}$ e

(begin e_1 e_2 ...)

  $\rightsquigarrow_{ds}$ (let ((n e_1)) (begin e_2 ...))

(if c e)

  $\rightsquigarrow_{ds}$ (if c e #u)

(cond (c_1 e_1))

  $\rightsquigarrow_{ds}$ (if c_1 e_1)

(cond (c_1 e_1) (c_2 e_2) ...)

  $\rightsquigarrow_{ds}$ (if c_1 e_1 (cond (c_2 e_2) ...))

L_3 desugaring (5)

(and e_1 e_2)

  $\rightsquigarrow_{ds}$ (if e_1 e_2 #f)

(or e_1 e_2)

  $\rightsquigarrow_{ds}$ (let ((v e_1)) (if v v e_2))

(not e)

  $\rightsquigarrow_{ds}$ (if e #f #t)

L_3 desugaring (6)

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

"$c_1c_2\ldots c_n$"

$\rightsquigarrow_{ds}$
```
(let ((s (@block-alloc-200 n)))
  (@block-set! s 0 'c1')
  (@block-set! s 1 'c2')
  ...
  s)
```

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 □.

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

For example, if C is `(if □ 1 2)`, then C{`(< x y)`} is `(if (< x y) 1 2)`.

Desugaring contexts

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

C_{ds} ::= □

| `(program` C_{ds}`)`

| `(let ((`n_1 e_1`)` ... `(`n_i C_{ds}`)` ... `(`n_k e_k`))` e`)`

| `(let ((`n_1 e_1`)` ... `(`n_k e_k`))` C_{ds}`)`

| `(letrec ((`n_1 f_1`)` ... `(`n_i C_{ds}`)` ... `(`n_k f_k`))` e`)`

| `(letrec ((`n_1 f_1`)` ... `(`n_k f_k`))` C_{ds}`)`

| `(fun#` `(`n_1 ... n_k`)` C_{ds}`)`

| `(if` C_{ds} e_2 e_3`)` | `(if` e_1 C_{ds} e_3`)` | `(if` e_1 e_2 C_{ds}`)`

| `(`C_{ds} e_1 ... e_k`)` | `(`e e_1 ... C_{ds} ... e_k`)`

| `(@` p e_1 ... C_{ds} ... e_k`)`

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:

$C_{ds}\{T\} \Rightarrow_{ds} C_{ds}\{T'\}$ where $T \rightsquigarrow_{ds} T'$

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

L_3 desugaring example

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

$\Rightarrow_{ds}$
```
(program
    (begin (char-print (if #t 'o' 'k'))
           (char-print (if #f 'o' 'k'))))
```

$\Rightarrow_{ds}$
```
(program
    (let ((t (char-print (if #t 'o' 'k'))))
      (char-print (if #f 'o' 'k'))))
```

$\not\Rightarrow_{ds}$ cannot be rewritten further

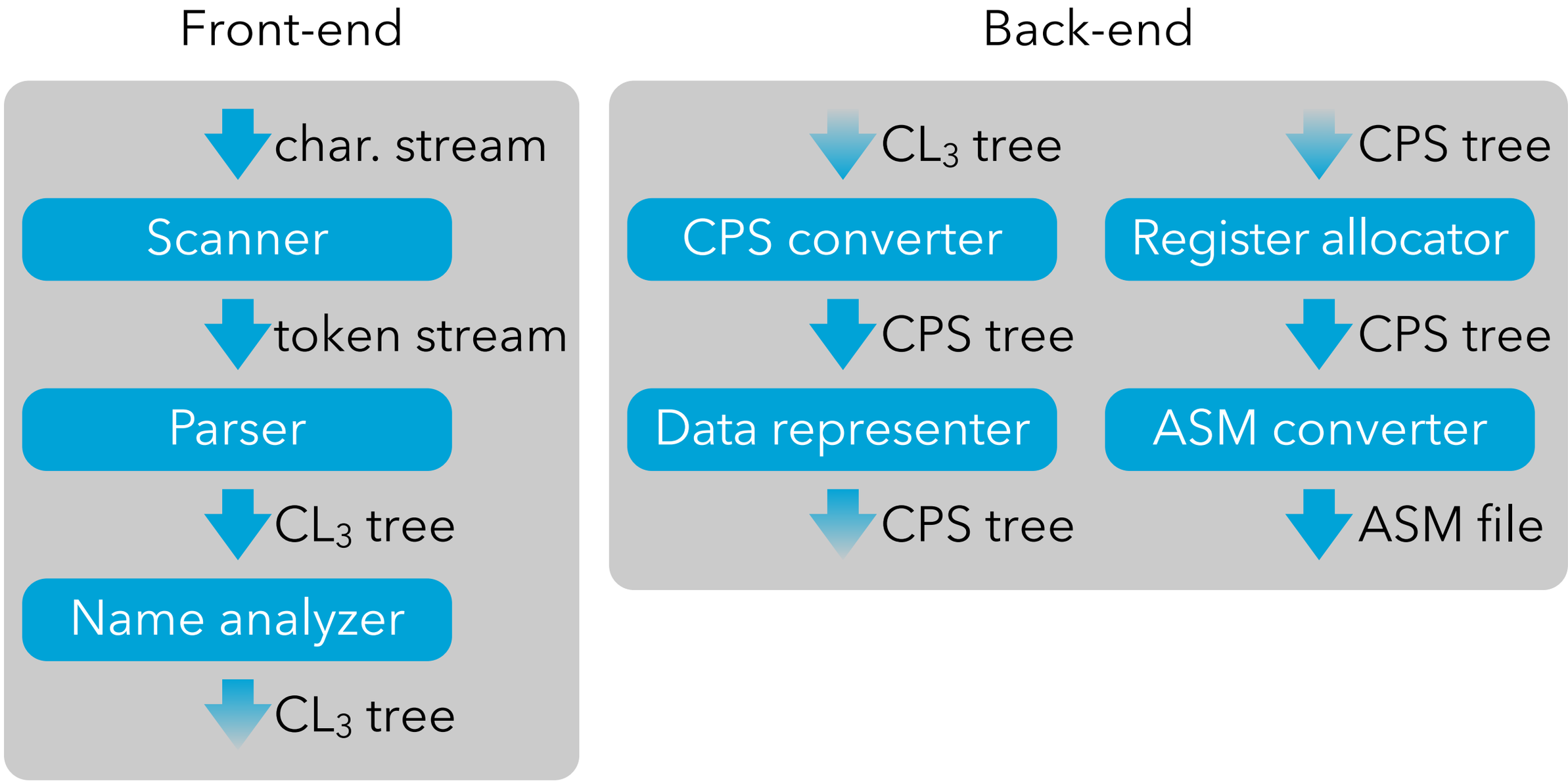
L_3 desugaring exercise

Desugar the following L_3 expression by applying the desugaring relation until you get a term that cannot be rewritten anymore:

```
(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

Intermediate languages

The L_3 compiler manipulates a total of four languages:

1. L_3 is the source language that is parsed,
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.