Universidad Nacional
Autónoma de México

Facultad de Ciencias

Compiladores

Proyecto Final

Alumnos:
Alma Rosa Páes Alcalá
Alejandro T. Valderrama Silva
Francisco Emmanuel Anaya González Erick Enrique Castro Espinosa

Profesor:
Lourdes del Carmen González Huesca

Ayudantes:
Javier Enríquez Mendoza
Alejandra K. Coloapa Díaz

12 de diciembre de 2019

Esta página fue dejada intencionalmente en blanco.

Índice

Introducción

Un compilador es un tipo de traductor que transforma un programa entero de un lenguaje de programación a otro. Usualmente el lenguaje objetivo es código máquina, aunque también puede ser traducido a un código intermedio o a texto.

A lo largo del curso de compiladores se ha estudiado el diseño y desarrollo de éstos; con el objetivo de crear una implementación basada en los conceptos vistos en clase. Para esto se creó un lenguaje fuente, *LF*, que recibiera nuestro compilador; cuya gramática es la siguiente:

```
<programa> ::= <expr>

<expr> ::= <const>
           | <list>
           | <var>
           | <string>
           | (<prim> <const> <const>*)
           | (begin <expr> <expr> *)
           | (if <expr> <expr>)
           | (if (expr) <expr> <expr>)
           | (define <var> <type> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | ( cons <const> <list>)
```

```
<string> :: = "" | "<char> <string>"

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | and | or | not | length | car | cdr |
   ↪ < | > | equal? | iszero? | ++ | -- | equal-lst? | empty? |
   ↪ elem? | append | concat | cons

<type> :: = Bool | Int | Char | List | String
```

Objetivo

A lo largo del curso se ha desarrollado un compilador que transforma, a través de cada uno de los procesos del compilador, el lenguaje fuente en un lenguaje distinto, en muchos aspectos, pero que mantiene la misma semántica; con el objetivo de facilitar su traducción a *C*.

El propósito de este proyecto es extender dicho compilador para tomar un archivo que contiene código en *LF* (con extensión .mt) y obtener tres archivos, uno por cada etapa del compilador.

La implementación consta de once fases, las cuales son representadas por los lenguajes y un grupo de procesos que modifican la sintaxis de las expresiones. Con esto se busca complementar y culminar con el desarrollo de un compilador totalmente funcional para la gramática y lenguaje propuesto. Como bien sabemos el compilador fue hecho en *nanopass*, que es un dialecto de *Racket*, que a su vez, es un lenguaje funcional de la familia LISP.

Finalmente la traducción no se logró, posiblemente debido a falta de tiempo y a falta de más procesos que la simplificaran, por lo que quedó en nosotros modificar la forma en la que se traduciría este proyecto o cambiar el lenguaje de programación objetivo.
Con ese fin nosotros escogimos cambiar el lenguaje objetivo a **Python**.

Manuel de Usuario

En esta sección proporcionaremos explicaciones fundamentales para poder usar el compilador desarrollado. En éstas incluiremos una breve explicación de como ejecutar nuestro compilador, así como también los errores.

Como podemos notar en la definición de la gramática, descrita en la introducción, nuestro lenguaje fuente está orientado a la programación funcional, en donde podemos hacer todas las operaciones definidas por *prim*, las cuales involucran operaciones aritméticas y con listas. También podemos definir y trabajar con funciones.

Uso

Para poder usar nuestro compilador basta con ejecutar nuestro compilador, **proyecto.rkt**, con el interprete de racket y luego resta ejecutar nuestro compilador con el archivo **.mt** que contenga el código de acuerdo a nuestra gramática fuente. De la siguiente manera:

```
racket -i -e '(enter! "proyecto.rkt")'
```

Y luego:

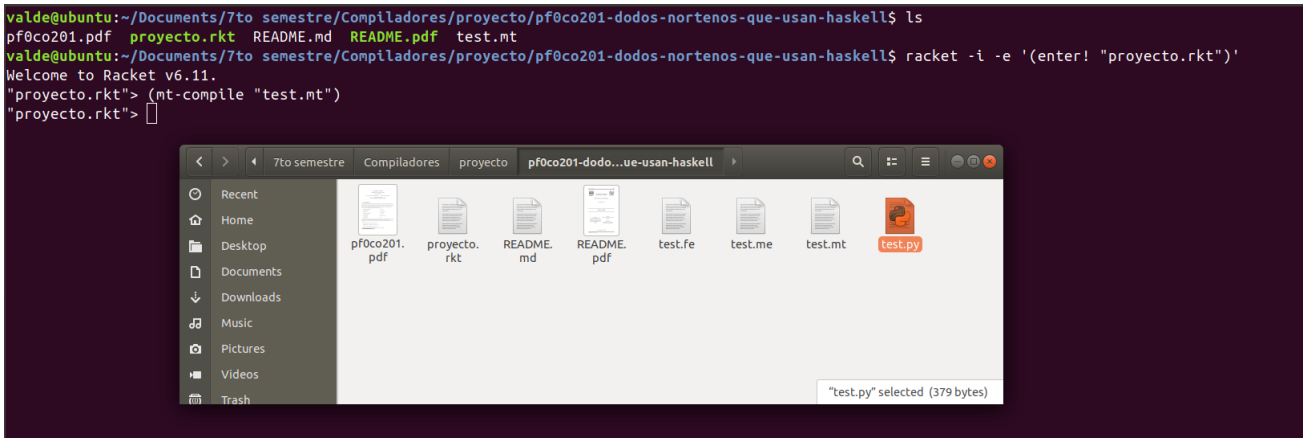
```
(mt-compile <nombre de archivo>)
```

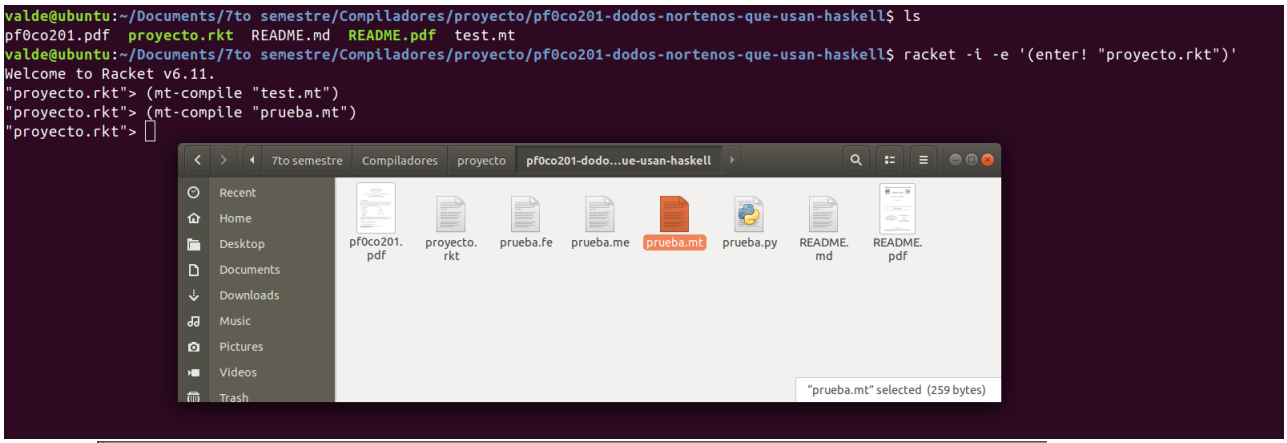
Esto generará, si el archivo es compilado **exitosamente**, tres archivos:

- nombre de archivo.**fe**: Que tendrá las expresiones correspondientes al *front-end* del compilador.
- temp.**me**: Que tendrá las expresiones correspondientes al *middle-end* del compilador.
- temp .**py**: Que tendrá las expresiones correspondientes al *back-end* del compilador. Este es el archivo objetivo, el cual el usuario lo puede ejecutar de su manera favorita.

Ejemplos

A continuación mostraremos un par de ejemplos de como compilar y un ejemplo de un archivo fuente válidos:






```
(+ 2 4)
(equal-lst? (list #\h #\o #\l #\a) (list #\a #\l #\o #\h))
(iszero? 6)
(if (iszero? 3) "cond1" "cond2")
(if #f "adios")
(let ([y Int 5] [z Int (+ y 2)]) (< y z))
(begin (+ 1 3) (list 4 5 6 7) (empty? (list 1)))
(let ([x Int 0]) (++ x))
(list #\c #\o)
```

Errores

Los errores que pueden surgir al compilar nuestro archivo **.mt** son los siguientes:

- "No pueden haber variables libres"
- .Aridad incorrecta"
- "Se esperan 2 tipos para unificar"
- "La variable no está en el contexto"
- "Los operadores binarios deben tener parámetros de tipo Integer"
- .El tipo no concuerda con el operador de listas"
- "Diferentes tipos en cláusulas del if"
- .El tipo no corresponde con el valor"
- "Las listas degen ser homogéneas"
- "No se puede inferir el tipo. No se puede unificar"
- "No se pueden inferir los tipos"

Lenguajes intermedios del compilador

En el proceso de transformación del código fuente, escrito en LF, a nuestro lenguaje objetivo, que es *lenguaje*, establecimos distintas fases en forma de lenguajes intermedios, con el fin de optimizar el código y facilitar su análisis y traducción al lenguaje objetivo.

Fueron nombrados serialmente, y cada uno contiene una transformación distinta. Las transformaciones realizadas para la transición entre lenguajes están especificadas y detalladas en la sección de Procesos.

Lenguaje L1

Remueve la expresión *if* que sólo ejecuta una serie de acciones si la condición dada se cumple, y no tiene instricciones definidas en caso contrario. La gramática del lenguaje L1 se diferencía de L0 en únicamente quitar esta regla de las expresiones permitidas.

```
<programa> ::= <expr>

<expr> ::= <const>
           | <list>
           | <var>
           | <string>
           | (<prim> <const> <const>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>
```

```
<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<string> :: = "" | "<char> <string>"

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | and | or | not | length | car | cdr |
   ↪ < | > | equal? | iszero? | ++ | -- | equal-lst? | empty? |
   ↪ elem? | append | concat | cons

<type> :: = Bool | Int | Char | List | String
```

Lenguaje L2

Remueve las cadenas (*strings*) del lenguaje, ya que éstas pueden interpretarse como listas de caracteres.

```
<programa> ::= <expr>

<expr> ::= <const>
           | <list>
           | <var>
           | (<prim> <const> <const>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z
```

```
<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | and | or | not | length | car | cdr |
   ↪ < | > | equal? | iszero? | ++ | -- | equal-lst? | empty? |
   ↪ elem? | append | concat | cons

<type> :: = Bool | Int | Char | List
```

Lenguaje L-NLO

En la búsqueda por remover los operadores lógicos, operadores aritméticos y de listas, es más fácil trabajar para su sustitución si los vemos como expresiones del lenguaje, y no sólo como operaciones permitidas de éste.

Por lo tanto, creamos un lenguaje llamado *L-NLO*, que pasa los operadores a eliminar a expresiones del lenguaje, y los elimina del conjunto de operadores permitidos.

```
<programa> ::= <expr>

<expr> ::= <const>
           | <list>
           | <var>
           | (<prim> <const> <const>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (and <expr>*)
           | (or <expr>*)
           | (not <expr>)
           | (iszero? <expr>)
           | (++ <expr>)
           | (-- <expr>)
           | (empty? <expr>)
           | (cons <expr> <expr>)
           | (append <expr> <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
```

```
            | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪ equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List
```

Lenguaje L3

No contiene las expresiones del lenguaje que funcionaban como expresiones, logrando así no considerar operadores no útiles.

```
<programa> ::= <expr>

<expr> ::= <const>
           | <list>
           | <var>
           | (<prim> <const> <const>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f
```

```
<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪ equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List
```

Lenguaje L4

Envolvemos las primitivas (operaciones) en una expresión *primapp*.

```
<programa> ::= <expr>

<expr> ::= <const>
           | <list>
           | <var>
           | (primapp <prim> <const> <const>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>
```

```
<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪ equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List
```

Lenguaje L5

Envolvemos las constantes de nuestro lenguaje con el constructor *cuote*.

```
<programa> ::= <expr>

<expr> ::= <list>
           | <var>
           | (cuote <const>)
           | (primapp <prim> <expr> <expr>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]*) <expr>)
           | (letrec ([<var> <type> <expr>]*) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)
```

```
<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪ equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List
```

Lenguaje L6

La optimización de este lenguaje consiste en currificar las expresiones *let* y *letrec*.

```
<programa> ::= <expr>

<expr> ::= <list>
          | <var>
          | (cuote <const>)
          | (primapp <prim> <expr> <expr>*)
          | (begin <expr> <expr> *)
          | (if (expr) <expr> <expr>)
          | (while [<expr>] <expr>)
          | (lambda ([<var> <type>]*) <expr>)
          | (let ([<var> <type> <expr>]) <expr>)
          | (letrec ([<var> <type> <expr>]) <expr>)
          | (<expr> <expr>*)

<const> :: = <boolean>
          | <integer>
          | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪ equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List
```

Lenguaje L7

Hasta ahorita, las funciones se manejaban únicamente a través de expresiones *lambda*, por lo que eran anónimas. La diferencia implementada para este lenguaje consiste en asignarles un nombre, usando el constructor *letfun*, donde el tipo es *Lambda*, la expresión que define al nombre es la lambda, y el cuerpo es el mismo nombre de la función, para no cambiar la semántica de la función.

```
<programa> ::= <expr>

<expr> ::= <list>
           | <var>
           | (cuote <const>)
           | (primapp <prim> <expr> <expr>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]) <expr>)
           | (letrec ([<var> <type> <expr>]) <expr>)
           | (letfun ([<var> <type> <expr>]) <expr>)
           | (<expr> <expr>*)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪  equal-lst? | elem? | concat
```

```
<type> :: = Bool | Int | Char | List | Lambda
```

Lenguaje L8

Con el fin de inferir tipos para validar que las expresiones del lenguaje estén correctamente tipificadas, realizamos algunas modificaciones al lenguaje para que estos procedimientos nos sean más sencillos en un futuro.

En esta ocasión, currificamos las expresiones *lambda* y las aplicaciones de funciones, haciendo que cada una sólo reciba un parámetro, y el resto se encuentren en lambdas o aplicaciones anidadas, respectivamente.

```
<programa> ::= <expr>

<expr> ::= <list>
           | <var>
           | (cuote <const>)
           | (primapp <prim> <expr> <expr>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]) <expr>)
           | (let ([<var> <type> <expr>]) <expr>)
           | (letrec ([<var> <type> <expr>]) <expr>)
           | (letfun ([<var> <type> <expr>]) <expr>)
           | (<expr> <expr>)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
```

```
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪ equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List | Lambda
```

Lenguaje L9

Dado que nuestro objetivo es verificar los tipos, requerimos que incluso las constantes tengan un tipo explícito, por lo que cambiarán de constructor a una expresión *const*, donde se aprecia claramente el tipo de la constante.

Por otro lado, agregamos nuevos tipos a nuestro lenguaje, tales como

- $(< type > \rightarrow < type >)$, para representar a las funciones -donde el lado izquierdo representa el tipo del parámetro de entrada, y el lado derecho representa el tipo del valor que devuelve. Justamente por esto necesitábamos a nuestras expresiones *lambda* currificadas
- $(Listof < type >)$, utilizada para hacer explícito el tipo de los elementos de la lista.

```
<programa> ::= <expr>

<expr> ::= <list>
           | <var>
           | (const <type> <const>)
           | (primapp <prim> <expr> <expr>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]) <expr>)
           | (let ([<var> <type> <expr>]) <expr>)
           | (letrec ([<var> <type> <expr>]) <expr>)
           | (letfun ([<var> <type> <expr>]) <expr>)
           | (<expr> <expr>)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>

<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

```
<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪  equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List | Lambda | (<type> -> <type
   ↪ >) | (List of <type>)
```

Lenguaje L10

Recordemos que nuestro lenguaje objetivo es Python, y en este lenguaje, están permitidas las funciones multi paramétricas. Éstas son modeladas por nuestro lenguaje por la expresión *letfun*, que utiliza una *lambda* para definir su comportamiento y los valores de entrada.

Como ya hicimos la inferencia de tipos, que era el objetivo de nuestra currificación de lambdas, el siguiente paso es descurrificar las lambdas.

```
<programa> ::= <expr>

<expr> ::= <list>
           | <var>
           | (const <type> <const>)
           | (primapp <prim> <expr> <expr>*)
           | (begin <expr> <expr> *)
           | (if (expr) <expr> <expr>)
           | (while [<expr>] <expr>)
           | (lambda ([<var> <type>]*) <expr>)
           | (let ([<var> <type> <expr>]) <expr>)
           | (letrec ([<var> <type> <expr>]) <expr>)
           | (letfun ([<var> <type> <expr>]) <expr>)
           | (<expr> <expr>)

<const> :: = <boolean>
           | <integer>
           | <char>

<boolean> :: = #t | #f

<integer> :: = <digit> | <digit><integer>
```

```
<digit> :: = 0| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<var> :: = <car> | <car><var> | <car><digit> | <car><digit><var>

<car> :: = a | b | c | ... | z

<list> :: = empty | (list <const> <list>)

<char> :: = a | b | c | ... | z | ... | @ | # | \$ | \% | & | ...
<prim> :: = + | - | * | / | length | car | cdr | < | > | equal? |
   ↪  equal-lst? | elem? | concat

<type> :: = Bool | Int | Char | List | Lambda | (<type> -> <type
   ↪ >) | (List of <type>)
```

Procesos

Como ya mencionamos anteriormente nuestro compilador constara de once procesos, los cuales los discutiremos más detalladamente a en esta sección. El compilador consiste en aplicar cada uno de estos proceso, comenzando con una expresión de nuestro lenguaje LF. Es decir, la salida de cada proceso será la entrada del siguiente (en el orden en el que se presentan). Para cada uno de los procesos mostraremos el lenguaje que lo define.

make-explict

Este proceso cambia el cuerpo de las expresiones lambda, let y letrec por un begin, facilitando así futuros análisis. Éste toma una expresión del Lenguaje Fuente (LF) y nos regresa una expresión en el lenguaje $L0$.
Proceso:

```
(define-pass make-explicit : LF (ir) -> L0 ()
  (Expr : Expr (ir) -> Expr ()
    [,c `,c]
    [(lambda ([,x* ,t*] ...) ,[body*] ... ,[body])
     `(lambda ([,x* ,t*] ...) (begin ,body* ... ,body))]
    [(let ([,x* ,t* ,[e*]] ...) ,[body*] ... ,[body])
     `(let ([,x* ,t* ,e*] ...) (begin ,body* ... ,body))]
    [(letrec ([,x* ,t* ,[e*]] ...) ,[body*] ... ,[body])
     `(letrec ([,x* ,t* ,e*] ...) (begin ,body* ... ,body))]))
```

Donde $L0$ es:

```
(define-language L0
```

```
  (extends LF)
  (Expr (e body)
        (- (lambda ([x* t*] ...) body* ... body)
           (let ([x* t* e*] ...) body* ... body)
           (letrec ([x* t* e*] ...) body* ... body))
        (+ (lambda ([x* t*] ...) body)
           (let ([x* t* e*] ...) body)
           (letrec ([x* t* e*] ...) body))))
```

remove-one-armed-if

Es un proceso del compilador que se encarga de eliminar la expresión *if* sin que no tiene una expresión que corresponda al caso de *else*; de esta manera logramos unificar las dos producciones que existen para nuestras expresiones *if*. Es decir, después de este punto ya sólo lidiaremos con expresiones *if* que involucren tres expresiones. Éste recibe una expresión del lenguaje *L0* y regresa una expresión del lenguaje *L1*.
Proceso:

```
(define-pass remove-one-armed-if : L0 (ir) -> L1 ()
  (Expr : Expr (ir) -> Expr ()
  [(if ,[e1] ,[e2]) `(if ,e1 ,e2 (void))]))
```

Donde *L1* es:

```
(define-language L1
  (extends L0)
  (terminals
   (+ (void(v))))
  (Expr (e body)
        (- (if e0 e1))
        (+ v)))
```

remove-string

Es un proceso del compilador para eliminar las cadenas como elementos terminales del lenguaje. En lugar de ser cadenas serán listas de caracteres, similar a lo que se sucede en *C* o *Haskell*. Esto porque nuestro objetivo principal era traducir a *C*.
Éste recibe un expresión de *L1* y regresa otra de *L2*.
Proceso:

```
(define-pass remove-string : L1 (ir) -> L2 ()
  (Expr : Expr (ir) -> Expr ()
        [,s (parser-L2 (append '(list) (string->list s)))]))
```

Donde *L2* es:

```
(define-language L2
  (extends L1)
    (terminals
      (- (string (s))))
    (Expr (e body)
          (- s)))
```

L2-to-NLO

Este es un proceso auxiliar, el cual nos permite cambiar de lenguaje, pasando de *L2* a *L-NLO*. En realidad este es un proceso "dummie", no afecta a las estructura de las expresiones. *Proceso:*

```
(define-pass L2-to-NLO : L2 (ir) -> L-NLO ()
  (Expr : Expr (ir) -> Expr ()
        [else (parser-L-NLO (unparse-L2 ir))]))
```

Donde *L-NLO* es:

```
(define-language L-NLO
  (extends L2)
  (terminals
   (- (primitive (pr)))
   (+ (primitiva (pr))))
  (Expr (e body)
        (+ (and e* ...)
           (or e* ...)
           (not e)
           (iszero? e0)
           (++ e0)
           (-- e0)
           (empty? e0)
           (cons e0 e1)
           (append e0 e1))))
```

remove-stupid-operators

Este proceso se encarga de varias cosas.

- Quita los operadores lógicos *and* y *or* y se sustituyen por su equivalente en una expresión *if*.
- Quita las operaciones aritméticas que podemos definir con base en otras ya existentes en el lenguaje.
- Quita las operaciones con listas que podemos definir con base en otras ya existentes en el lenguaje.

Éste recibe una expresión del lenguaje *L-NLO* y regresa otra del lenguaje *L3*.
Proceso:

```
(define-pass remove-stupid-operators : L-NLO (ir) -> L3 ()
  (Expr : Expr (ir) -> Expr ()
        [(and) `(#t)]
        [(and ,[e1]) `(#t)]
        [(and ,[e1] ,[e2]) `(if ,e1 ,e2 #f)]
        [(or) `(#t)]
        [(or ,[e1]) `(#t)]
        [(or ,[e1] ,[e2]) `(if ,e1 #t ,e2)]
        [(iszero? ,[e0]) `(equal? e0 0)]
        [(++ ,[e0]) `(+ ,e0 1)]
        [(-- ,[e0]) `(- ,e0 1)]
        [(empty? ,[e0]) `(equal-lst? ,e0 (list))]
        [(append ,[e0] ,[e1]) `(concat ,e1 (list ,e0))]
        [(cons ,[e0] ,[e1]) `(concat (list ,e0) ,e1)]
        [(not ,[e1]) `(if ,e1 #f #t)]))
```

Donde *L3* es:

```
(define-language L3
  (extends L-NLO)
  (Expr (e body)
        (- (and e* ...)
           (or e* ...)
           (not e)
           (iszero? e0)
           (++ e0)
           (-- e0)
           (empty? e0)
           (cons e0 e1)
           (append e0 e1))))
```

eta-expand:

Este proceso se basa en las η-reducciones del cálculo lambda, lo que hace es reconocer una función que toma un conjunto de argumentos y los pasa directamente a otra función. Es decir, se van a eliminar las primitivas como expresiones del lenguaje y se sustituyen por *lambdas* en caso de encontrarse solas o por *primapp* si se esta haciendo la aplicación de la primitiva.
Este proceso recibe una expresión del lenguaje *L3* y regresa otra del lenguaje *L4*.
Proceso:

```
(define-pass eta-expand : L3 (ir) -> L4 ()
  (definitions
    (define var1 (new-var))
```

```
    (define var2 (new-var))
    (define (elem-type e0)
      (cond
         [(integer? e0) 'Int]
         [(char? e0) 'Char]
         [(boolean? e0) 'Bool])))
  (Expr : Expr (ir) -> Expr ()
        [(,pr) (if (equal? `,(hash-ref tipos `,pr) 'Int)
                   `(lambda ([,var1 ,(hash-ref tipos `,pr)] [,var2 ,(
                      ↪ hash-ref tipos `,pr)]) (begin (primapp ,pr ,
                      ↪ var1 ,var2)))
                   `(lambda ([,var1 ,(hash-ref tipos `,pr)]) (begin (
                      ↪ primapp ,pr ,var1))))]
        [(,pr ,[e0]) `((lambda ([,var1 ,(hash-ref tipos `,pr)]) (
           ↪ begin (primapp ,pr ,var1))) ,e0)]
        [(,pr ,[e0] ,[e1]) (if (equal? pr 'elem?)
                       `((lambda ([,var1 ,(elem-type `,e0)] [,var2 ,(
                          ↪ hash-ref tipos `,pr)]) (begin (primapp ,
                          ↪ pr ,var1 ,var2))) ,e0 ,e1)
                       `((lambda ([,var1 ,(hash-ref tipos `,pr)] [,
                          ↪ var2 ,(hash-ref tipos `,pr)]) (begin (
                          ↪ primapp ,pr ,var1 ,var2))) ,e0 ,e1))]))
```

Donde *L4* es:

```
(define-language L4
  (extends L3)
  (Expr (e body)
        (- pr)
        (+ (primapp pr e* ... e0))))
```

quote-const

Este proceso se encarga de "quotear" las constantes de nuestra expresión, facilitando así futuro análisis. Es decir, a partir de este momento las constantes, por si solas, ya no serán expresiones válidas.
Éste proceso recibe una expresión de *L4* y regresa otra de *L5*.
Proceso:

```
(define-pass quote-const : L4 (ir) -> L5 ()
  (Expr : Expr (ir) -> Expr ()
        [,c `(cuote ,c)]))
```

Donde *L5* es:

```
(define-language L5
  (extends L4)
  (Expr (e body)
```

```
        (- c)
        (+ (cuote c))))
```

curry-let

Este proceso es el encargado de currificar las expresiones let y letrec. La currificación es la técnica en la que se transforma una función de múltiples parámetros en una secuencia de funciones de un único parámetro. Esto nos serviría para la traducción de código en *C* para transformar cada *let* en una asignación y cada *letrec* en una definición de función.
Éste recibe una expresión de *L5* y regresa otra de *L6*.
Proceso:

```
(define-pass curry-let : L5 (ir) -> L6 ()
  (definitions
    (define (curry vars types values body op)
      (cond
        [(empty? vars) (unparse-L6 body)]
        [else `(,op ([,(car vars) ,(car types) ,(unparse-L6 `,(car
           ↪ values))]) ,(curry (cdr vars) (cdr types) (cdr values)
           ↪ body op))])))
  (Expr : Expr (ir) -> Expr ()
        [(let ([,x* ,t* ,[e*]] ...) ,[body]) (parser-L6 (curry x* t*
           ↪ e* body 'let))]
        [(letrec ([,x* ,t* ,[e*]] ...) ,[body]) (parser-L6 (curry x*
           ↪ t* e* body 'letrec))]))
```

Donde *L6* es:

```
(define-language L6
  (extends L5)
  (Expr (e body)
        (- (let ([x* t* e*] ...) body)
           (letrec ([x* t* e*] ...) body))
        (+ (letrec ([x t e]) body)
           (let ([x t e]) body))))
```

identify-assigments

Este proceso es el encargado de detectar los *let* utilizados para definir funciones y se remplazarlos por *letrec*. Esto lo que nos permite es separar las asignaciones de funciones de los demás tipos de datos, el objetivo de esto es que la traducción a C fuera más sencilla.
Éste proceso no genera un cambio de lenguaje.
proceso:

```
(define-pass identify-assigments : L6 (ir) -> L6 ()
  (Expr : Expr (ir) -> Expr ()
        [(let ([,x ,t ,[e]]) ,body)
         (if (equal? t 'Lambda) `(letrec ([,x ,t ,e]) ,body) ir)]))
```

un-anonymous

Este proceso es el encargado de asignarle un identificador a las funciones anónimas, es decir a nuestras *lambdas*. Esto porque en *C* toda función debe llevar un nombre.
Éste recibe una expresión del lenguaje *L6* y regresa otra del lenguaje *L7*.
proceso:

```
(define-pass un-anonymous : L6 (ir) -> L7 ()
  (definitions
    (define (new-name)
      (begin
        (define str (string-append "foo" (number->string contador)))
        (set! contador (+ 1 contador))
        (string->symbol str))))
  (Expr : Expr (ir) -> Expr ()
        ([lambda ([,x ,t] ...) ,[body]]
         (let ([name (new-name)]) `(letfun ([,name Lambda ,(parser-L8
            ↪ (unparse-L7 ir))]) ,name)))))
```

Donde *L7* es:

```
(define-language L7
  (extends L6)
  (Expr (e body)
        (+ (letfun ([x t e]) body))))
```

verify-arity:

Éste proceso verifica que la aridad de las operaciones, tanto aritmética como de listas, tenga la aridad correcta. Es necesario para no permitir operaciones aritméticas o de listas incorrectas.
Éste proceso no genera un cambio de lenguaje.
proceso:

```
(define-pass verify-arity : L7 (ir) -> L7 ()
  (Expr : Expr (ir) -> Expr ()
        [(primapp ,pr ,[e0]) (if (lst1? pr) ir (error "Arity mismatch
           ↪ "))]
        [(primapp ,pr ,[e0] ,[e1]) (if (or (lst2? pr) (arit? pr)) ir
           ↪ (error "Arity mismatch"))]))
```

verify-vars:

Este proceso verifica que no haya variables libres en nuestras expresiones, pues en caso de que haya, la expresión carecería de sentido; es decir, es necesario para detectar expresiones semánticamente incorrectas. Básicamente todo el trabajo lo hace nuestra función auxiliar FV, lo incluimos ambas a continuación
Éste proceso no genera un cambio de lenguaje.

```
(define (free-var exp)
  (nanopass-case (L7 Expr) exp
        [,x `(,x)]
        [(cuote ,c) '()]
        [(begin ,e* ... ,e) (append (free-var e) (foldr append '() (
           ↪ map free-var e*)))]
        [(primapp ,pr ,e* ... ,e0) (append (free-var e0) (foldr
           ↪ append '() (map free-var e*)))]
        [(if ,e0 ,e1 ,e2) (append (free-var e0) (free-var e1) (
           ↪ free-var e2))]
        [(lambda ([,x* ,t*] ...) ,body) (remq x* (free-var body))]
        [(let ([,x ,t ,e]) ,body) (remq x (append (free-var body) (
           ↪ free-var e)))]
        [(letrec ([,x ,t ,e]) ,body) (remq x (append (free-var body)
           ↪ (free-var e)))]
        [(define ,x ,t ,e) (remq x (free-var e))]
        [(while [,[e0]] ,[e1]) (append (free-var e0) (free-var e1))]
        [(letfun ([,x ,t ,e]) ,body) (remq x (append (free-var body)
           ↪ (free-var e)))]
        [(list ,e* ...) (foldr append '() (map free-var e*))]
        [(,e0 ,e1 ...) (append (free-var e0) (foldr append '() (map
           ↪ free-var e1)))]
        [else '()]))


(define-pass verify-vars : L7 (ir) -> L7 ()
  (Expr : Expr (ir) -> Expr ()
       [else (if (empty? (free-var ir)) ir (error "Free var"))]))
```

curry

Este proceso se encarga de currificar las expresiones lambda así como las aplicaciones de función. Esto puede parecer extraño pues la información ya la tenemos de igual forma, pero esto nos facilitará la implementación del algoritmo J, necesario para la inferencia de tipos.
Éste recibe una expresión del lenguaje $L7$ y regresa otra del lenguaje $L8$.
proceso:

```
(define-pass curry : L7 (ir) -> L8 ()
  (definitions
    (define (curry-lambda vars types body)
      (cond
        [(empty? vars) (unparse-L8 body)]
        [else `(lambda ([,(car vars) ,(car types)]) ,(curry-lambda (
           ↪ cdr vars) (cdr types) body))]))
    (define (curry-app primero resto)
      (cond
        [(empty? resto) primero]
        [else (curry-app `(,primero ,(car resto)) (cdr resto))])))
  (Expr : Expr (ir) -> Expr ()
        [(lambda ([,x* ,t*] ...) ,[body]) (parser-L8 (curry-lambda x*
           ↪  t* body))]
        [(,e0 ,e1 ...) (parser-L8 (curry-app (unparse-L7 e0) (map
           ↪ unparse-L7 e1)))]))
```

Donde *L8* es:

```
(define-language L8
  (extends L7)
  (Expr (e body)
        (- (lambda ([x* t*] ...) body)
           (e0 e1 ...))
        (+ (lambda ([x t]) body)
           (e0 e1))))
```

type-const

Este proceso se encarga de colocar las anotaciones de tipos correspondientes a las constantes de nuestro lenguaje. De igual manera que el proceso anterior, conocer explícitamente el tipo de cada una de las constantes nos facilita el trabajo para el algoritmo J.
Este preproceso recibe una expresión del lenguaje *L8* y regresa otra del lenguaje *L9*.
Preproceso:

```
(define-pass type-const : L8 (ir) -> L9 ()
  (Expr : Expr (ir) -> Expr ()
        [(cuote ,c)
         (cond
           [(integer? c) `(const Int ,c)]
           [(char? c) `(const Char ,c)]
           [(boolean? c) `(const Bool ,c)])]))
```

Donde *L9* es:

```
(define-language L9
```

```
  (extends L8)
  (Expr (e body)
        (- (cuote c))
        (+ (const t c))))
```

type-infer

Este proceso se encarga de de quitar las anotaciones de tipo Lambda y sustituirla por el tipo: $'(T \to T)$, así como las listas por tipo $List\ of < Type >$, sólo en caso de ser necesario. Esto lo queremos hacer porque al traducir a C todo tiene que tener un tipo bien definido, entonces aquí nos aseguramos que todas las expresiones ya los tengan.
Este proceso recibe expreciones del lenguaje *L9* y otra exprecion del mismo lenguaje.

```
(define-pass type-infer : L9 (ir) -> L9 ()
  (definitions)
  (Expr : Expr (ir) -> Expr()
    [(lambda ([,x ,t]) ,[body]) (if (or (equal? t 'List) (equal? t '
       ↪ Lambda))
                                  `(lambda ([,x ,(J body (getCtx ir))
                                     ↪ ]) ,body)
                                   ir)]
    [(let ([,x ,t ,[e]]) ,[body]) (if (equal? t 'List)
                                  `(let ([,x ,(J e (getCtx ir)) ,e]) ,
                                     ↪ body)
                                  ir)]
    [(letrec ([,x ,t ,[e]]) ,[body]) (if (or (equal? t 'List) (equal?
       ↪  t 'Lambda))
                                  `(letrec ([,x ,(J e (getCtx ir)) ,e])
                                     ↪ ,body)
                                     ir)]
    [(letfun ([,x ,t ,[e]]) ,[body]) (if (or (equal? t 'List) (equal?
       ↪  t 'Lambda))
                                  `(letfun ([,x ,(J e (getCtx ir)) ,e])
                                     ↪ ,body)
                                  ir)]))
```

uncurry

Este proceso es el encargado de descurrificar las expresiones lambda de nuestro lenguaje. Puede resultar extraño que busquemos descurrificar cuando hace unos procesos currificamos, pero el objetivo de es esto es que será mucho más fácil traducir a C si tenemos todos los parámetros juntos.
Éste recibe una expresión del lenguaje *L9* y regresa otra del lenguaje *L10*.

```
(define-pass uncurry : L9 (ir) -> L10 ()
```

```
  (definitions
    (define (des-curry espl lst)
      (nanopass-case (L9 Expr) espl
      [(lambda ([,x ,t]) ,body) (des-curry body (append lst (list (
         ↪ cons x t))))]
      [else (values espl lst)])))
  (Expr : Expr (ir) -> Expr ()
        [(lambda ([,x ,t]) ,[body]) (let-values ([(e elem) (des-curry
           ↪  ir '())])
                                    (let ([x* (map car elem)]
                                          [e* (map cdr elem)])
                                      `(lambda ([,x* ,e*] ...) ,(
                                         ↪ parser-L10 (unparse-L9 e))
                                         ↪ )))])))
```

Donde *L10* es:

```
(define-language L10
  (extends L9)
  (Expr (e body)
    (- (lambda ([x t]) body))
    (+ (lambda ([x* t*] ...) body))))
```

Otros

Los proceso que acabamos de discutir son los base, es decir los que se fueron construyendo a lo largo del semestre. Además de lo que ya se había hecho, se modificaron los procesos necesarios para lograr hacer los ejercicios de este proyecto[1] pero además se incluyeron algunos procesos y funciones no mencionadas en la parte anterior. Esto sólo se hizo para el ejercicio 9 y para la traducción a Python, que son los siguientes:

Ejercicio 9: El objetivo de este ejercicio fue generar un archivo por cada etapa del compilador, para lo cual se ocuparon los siguientes procesos:

```
(define-pass front-end : LF (ir) -> L7 ()
  (Expr : Expr (ir) -> Expr ()
        [else (verify-vars
            (verify-arity
             (un-anonymous
              (identify-assigments
               (curry-let
                (direct-app
                 (quote-const
                  (eta-expand
```

[1]Los cuales fueron: 2, 4, 8, 9 y la traducción a **Python**.

```
                  (remove-stupid-operators
                   (L2-to-NL0
                    (remove-string
                     (remove-one-armed-if
                      (make-explicit ir))))))))))))))]))

(define-pass middle-end : L7 (ir) -> L10 ()
  (Expr : Expr (ir) -> Expr()
        [else (uncurry
               (type-infer
                (type-const
                 (curry ir))))]))
```

Y se ocuparon otras funciones auxiliares para lograr escribir esas expresiones en los archivos correspondientes.

Traducción:

```
(define (tabs n) (make-string n #\tab))
```

Consideraciones

Por último nos gustaría hablar de algunas de las consideraciones que hay que tener para este proyecto.

Primero es importante que el usuario sepa que si ya hay archivos, correspondientes a las etapas del compilador, ya construidos con cierto nombre y se quiere compilar un archivo con el mismo nombre, resultará en error.

En el código fuente, cada línea representa un bloque de código, es decir una expresión.

Como bien sabemos, y discutimos más arriba, el objetivo de convertir las cadenas a listas de caracteres fue facilitar la traducción a *C*, pero como el objetivo cambió y tradujimos a *Python* eso ya no fue necesario. Aún así lo dejamos así porque es uno de los procesos obligatorios y no tenía mucho sentido pasar listas de caracteres siempre a cadenas.

Como la traducción que hicimos fu distinta al objetivo, decidimos dejar algunos ejemplos de la función que lo hace. Por si quieren probarse.
Por último, tenemos un pequeño comentario con respecto a la gramática, específicamente sobre el diseño del lenguaje y es que nos parece raro que las *lambda* sean multivariables pero la aplicación de funciones solo tenga un parámetro. Como si no pudiéramos aplicar funciones multivariadas. Por lo mismo nuestra traducción a Python no traduce bien las funciones *lambda*, pues *curry* la convierte en una función de dos parámetros.