



Universidad Nacional Autónoma de México

FACULTAD DE CIENCIAS

COMPILADORES

Proyecto Final:

Compilador

Autores:

Escamilla Soto Cristopher Alejandro - 314309253 - cristopher@ciencias.unam.mx

Montiel Manriquez Ricardo - 314332662 - moma92@ciencias.unam.mx

Villegas Salvador Kevin Ricardo - 314173739 - kevin.ricardo@ciencias.unam.mx

18 de Junio de 2022

Breve Resumen

El propósito de este proyecto fue unir los procesos que permiten construir un compilador de expresiones de nuestro lenguaje fuente LF al lenguaje de programación C. Así como extenderlo para tomar un archivo que contenga código en LF y obtener resultados intermedios expresados en diferentes archivos auxiliares.

Los lenguajes abarcados para este proyecto van desde LF, L1 hasta L9 de los lenguajes que estuvimos utilizando en las practicas, lo único diferente que tienen es que los renombramos por comodidad para que podamos revisar el paso de procesos desde FRONT-END a BACK-END.

Abarcamos desde el lenguaje LF a L5 en la parte de FRONT-END, de L6 a L7 en la parte de MIDDLE-END y de L8 a L9 en la parte de BACK-END.

En cada una de los archivos generamos una función auxiliar, la cual generaba la compilación de su respectiva clase, en este caso para FRONT-END tenemos la siguiente:

```
(define (front expr)
  (curry
    (verify-vars
      (verify-arity
        (un-anonymous
          (identify-assigments
            (curry-let
              (remove-string
                (remove-one-armed-if expr))))))))))
```

Para MIDDLE-END tenemos la siguiente y podemos observar que estamos mandando a llamar la compilación de FRONT-END que es la cadena de compilación que se forma desde FRONT-END hasta MIDDLE-END:

```
(define (middle expr)
  (uncurry
    (type-infer
      (type-const
        (front expr))))))
```

Y finalmente para BACK-END podemos observar que estamos mandando a llamar la compilación de MIDDLE-END que es la cadena de compilación que se forma desde MIDDLE-END hasta BACK-END:

```
(define (back expr)
  (assignment (middle expr)))
```

Pero podemos observar que todavía falta un proceso **list-to-array**, dado que el proceso anterior **assignment** regresa dos valores que es la expresión y es la tabla hash, primero generamos la compilación hasta el proceso **assignment**, después obtenemos sus valores por aparte y aplicamos el ultimo proceso **list-to-array**

Finalmente tenemos la clase compilador en la cual vamos a abarcar las 3 fases (FRONT-END, MIDDLE-END y BACK-END). Para poder generar esto hacemos uso de tres funciones auxiliares **lee** y **escribe**, las cuales nos apoyaran para leer nuestros archivos ejemplo y para poder generar los archivos **.fe**, **.me** y **.c**. Cabe mencionar que al momento de compilar el archivo **compilador.rkt** también se mostraran en la terminal pruebas acerca de cada uno de los procesos que se generaron a lo largo del curso.

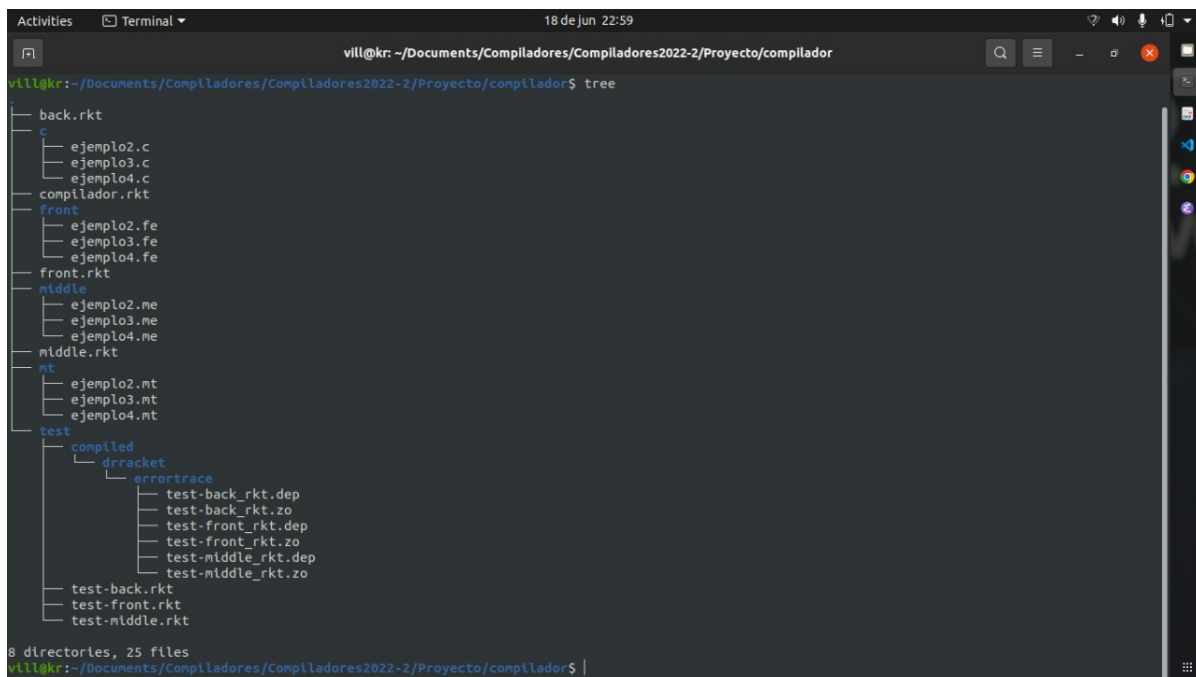
Estructura

Tenemos la siguiente estructura donde tenemos la siguiente carpeta **c**, en donde se almacenaran los archivos **.c** generados de la compilación.

Ahora tenemos la carpeta **front**, en donde se almacenaran los archivos **.fe** generados de la compilación.

Después tenemos la carpeta **middle**, en donde se almacenaran los archivos **.me** generados de la compilación.

Finalmente tenemos dos carpetas mas **mt** y **test**, en donde tenemos los archivos ejemplo y las pruebas de cada una de las fases



```
Activities Terminal 18 de jun 22:59
vill@kr: ~/Documents/Compiladores/Compiladores2022-2/Proyecto/compilador
vill@kr:~/Documents/Compiladores/Compiladores2022-2/Proyecto/compilador$ tree
.
├── back.rkt
├── c
│   ├── ejemplo2.c
│   ├── ejemplo3.c
│   └── ejemplo4.c
├── compilador.rkt
├── front
│   ├── ejemplo2.fe
│   ├── ejemplo3.fe
│   └── ejemplo4.fe
├── front.rkt
├── middle
│   ├── ejemplo2.me
│   ├── ejemplo3.me
│   └── ejemplo4.me
├── middle.rkt
├── mt
│   ├── ejemplo2.mt
│   ├── ejemplo3.mt
│   └── ejemplo4.mt
└── test
    ├── compiled
    │   ├── drracket
    │   │   └── errortrace
    │   │       ├── test-back_rkt.dep
    │   │       ├── test-back_rkt.zo
    │   │       ├── test-front_rkt.dep
    │   │       ├── test-front_rkt.zo
    │   │       ├── test-middle_rkt.dep
    │   │       └── test-middle_rkt.zo
    │   ├── test-back.rkt
    │   ├── test-front.rkt
    │   └── test-middle.rkt
    └── test-back.rkt
        ├── test-front.rkt
        └── test-middle.rkt

8 directories, 25 files
vill@kr:~/Documents/Compiladores/Compiladores2022-2/Proyecto/compilador$
```

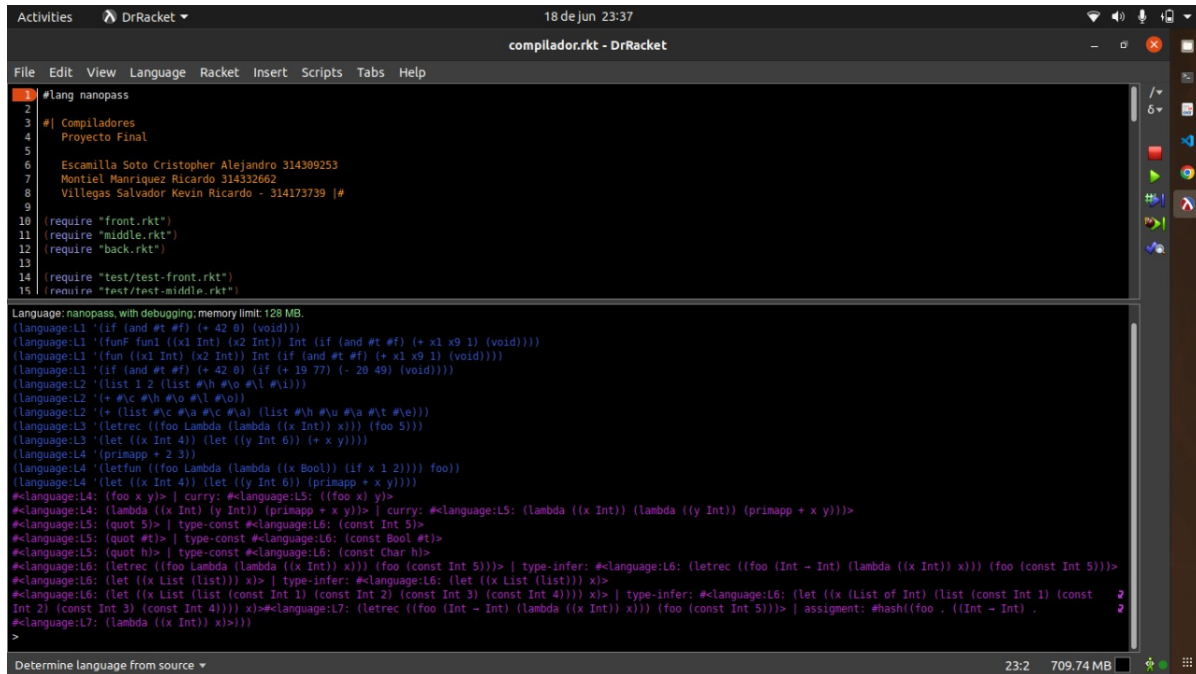
Ejercicios

Los ejercicios realizados fueron

- La extensión de while (ejercicio 2).
- La extensión de for (ejercicio 3).
- La extensión de expresión aritméticas primitivas (ejercicio 4).
- La definición del proceso c (ejercicio 5).
- Y finalmente la compilación de un archivo .mt (ejercicio 7).

Ejecución del Proyecto

Para probar que el proyecto funciona abriremos la clase **compilador.rkt** y la compilaremos dando clic en **Run** o bien apretando **Ctrl + R**.



```
Activities DrRacket 18 de jun 23:37
compilador.rkt - DrRacket
File Edit View Language Racket Insert Scripts Tabs Help
1 #lang nanopass
2
3 #| Compiladores
4 Proyecto Final
5
6 Escamilla Soto Cristopher Alejandro 314309253
7 Montiel Manriquez Ricardo 314332662
8 Villegas Salvador Kevin Ricardo - 314173739 |#
9
10 (require "front.rkt")
11 (require "middle.rkt")
12 (require "back.rkt")
13
14 (require "test/test-front.rkt")
15 (require "test/test-middle.rkt")
16
Language: nanopass, with debugging; memory limit: 128 MB.
(Language:L1 (if (and #t #f) (+ 42 0) (void)))
(Language:L1 (fun (f) (fun (x1 Int) (x2 Int)) Int (if (and #t #f) (+ x1 x2 1) (void))))
(Language:L1 (fun (x1 Int) (x2 Int)) Int (if (and #t #f) (+ x1 x2 1) (void)))
(Language:L1 (if (and #t #f) (+ 42 0) (if (+ 19 77) (- 20 40) (void))))
(Language:L2 (list 1 2 (list #h #o #l #l)))
(Language:L2 (+ #c #h #o #l #l))
(Language:L2 (+ (list #c #a #c #a) (list #h #u #a #t #e)))
(Language:L3 (letrec ((foo (lambda (x Int) (lambda (y Int) (x))) (foo 5)))
(Language:L3 (let ((x Int 4)) (let ((y Int 6)) (+ x y))))
(Language:L4 (primapp + 2 3))
(Language:L4 (letfun ((foo (lambda (x Bool) (if x 1 2))) foo))
(Language:L4 (let ((x Int 4)) (let ((y Int 6)) (primapp + x y))))
#<Language:L4: (foo x y) | curry: #<Language:L5: ((foo x) y)>
#<Language:L4: (lambda ((x Int) (y Int)) (primapp + x y)) | curry: #<Language:L5: (lambda ((x Int) (lambda ((y Int) (primapp + x y))))>
#<Language:L5: (quot 5)> | type-const #<Language:L6: (const Int 5)>
#<Language:L5: (quot #t)> | type-const #<Language:L6: (const Bool #t)>
#<Language:L5: (quot h)> | type-const #<Language:L6: (const Char h)>
#<Language:L6: (letrec ((foo (lambda (x Int) (lambda (y Int) (x))) (foo (const Int 5)))) | type-infer: #<Language:L6: (letrec ((foo (Int - Int) (lambda ((x Int) x))) (foo (const Int 5))))>
#<Language:L6: (let ((x List (list))) x) | type-infer: #<Language:L6: (let ((x List (list))) x)>
#<Language:L6: (let ((x List (list (const Int 1) (const Int 2) (const Int 3) (const Int 4)))) x) | type-infer: #<Language:L6: (let ((x (List of Int) (list (const Int 1) (const
Int 2) (const Int 3) (const Int 4)))) x)>#<Language:L7: (letrec ((foo (Int - Int) (lambda ((x Int) x))) (foo (const Int 5)))) | assignment: #hash((foo . ((Int - Int) .
#<Language:L7: (lambda ((x Int) x))))>
>
Determine language from source 23:2 709.74 MB
```

Podemos verificar en consola que se corren las pruebas que realizamos y para compilar algún archivo **.mt** por medio de la terminal escribiremos (**compila “Archivo”**) donde archivo es el nombre del archivo que deseamos compilar sin extensión **.mt**

Por defecto hemos agregado 3 archivos, ejemplo1, ejemplo2 y ejemplo 3, por lo que podemos probarlos escribiendo (**compila “ejemplo1”**), (**compila “ejemplo2”**) o bien (**compila “ejemplo3”**). Una vez hecho esto se nos generaran sus respectivos archivos dentro de las carpetas antes mencionadas (**front**, **middle** y **c**).

