Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Proyecto final

Integrantes del Equipo:
Ángeles Martínez Ángela Janín **314201009**
Bernal Martínez Fernando **313352304**
García Landa Valeria **314033008**
Hernández Chávez Jorge Argenis **312169206**
Martínez Monroy Emily **314212391**
Luna Vázquez Felipe Alberto **313203079**
Rebollar Pérez Ailyn **314322164**

Profesor y ayudantes:
Lourdes del Carmen González Huesca
Alejandra Krystel Coloapa Díaz
Uriel Agustín Ochoa Gónzalez
Javier Enríquez Mendoza

Documentación de Proyecto

1. Manual de Uso

Archivo de entrada (características)

Se recomienda que cada expresión este escrita en una sola línea

Ejemplo.

Linea 1: '(+ 3 5)

Linea 2: '(and #t #f)

Linea 3: '(or #t #f)

Linea 4:'(< 1 2)

El archivo debe tener terminación **.mt**

Ejecución

Si se desea compilar un archivo, debe tener extensión **.mt**

1. Ejecutar el compilador, *racket compiler.rtk*
2. En seguida se solicita el nombre del archivo a compilar, se debe ingresar el nombre del archivo **completo y con extensión**
3. El proceso tarda un momento y genera dos archivos, con extensión **.fe, .me**

2. Lenguaje Fuente

Gramática

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

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

Lenguaje

```
(define-language LF
  (terminals
    (variable (x))
    (primitive (pr))
    (constant (c))
    (list (l))
    (string (s))
    (type (t)))
  (Expr (e body)
    x
    c
    l
    s
    pr
    (define x e)
    (begin e* ... e)
    (while [e0] e1)
    (for [x e0] e1)
    (if e0 e1)
    (if e0 e1 e2)
    (lambda ([x* t*] ...) body* ... body)
    (let ([x* t* e*] ...) body* ... body)
    (letrec [x* t* e*] ...) body* ... body)
    (list e* ... )
    (e0 e1 ...)))
```

3. Justificación de Diseño

Ejercicio 4: Extender el conjunto de operaciones aritméticas primitivas del lenguaje con ¡(menor que), ¿(mayor que), equal? (igualdad), iszero? (recibe un número y dice si este es 0), ++ (incrementa en 1) y (decrementa en 1). Todas estas primitivas se deben tratar en los procesos del compilador y de ser posible eliminarse con alguna equivalencia en una etapa temprana como se hizo con las primitivas lógicas.
Se extendió el conjunto de operaciones aritméticas en las siguientes líneas:

```
12  (define (primitive? x) (memq x '(+ - * / length car cdr and or not equal? < > isZero? ++ --)))
13
14  (define (primitiva? x) (memq x '(+ - * / length car cdr < > equal?)))
```

Como podemos notar se implementó otro predicado llamado primitiva donde ya no se incluye ++, –, isZero? que se utilizará para eliminarse con una equivalecia.
Poe ejemplo en el porceso llamado change-languaje se elimina ++ y –, el operador ++ es equivalente a sumar 1 y el operador – es equivalente a restar 1.

```
274  ;; Proceso que cambia del lenguaje L2 a LIF
275  (define-pass change-language : L2 (ir) -> LIF ()
276    (Expr : Expr (ir) -> Expr ()
277          [,pr (cond
278                 [(equal? 'not pr) `(not)]
279                 [(equal? 'and pr) `(and)]
280                 [(equal? 'or pr) `(or)]
281                 [else `,pr])]
282          [(,e0 ,[e1] ...) (cond
283                            [(and (primitive? e0)(equal? 'not e0)) `(not ,(car e1))]
284                            [(and (primitive? e0)(equal? 'and e0)) `(and ,e1 ...)]
285                            [(and (primitive? e0)(equal? 'or e0)) `(or ,e1 ...)]
286                            [(and (primitive? e0) (equal? '++ e0)) `(+ ,(car e1) 1)]
287                            [(and (primitive? e0) (equal? '-- e0)) `(- ,(car e1) 1)]
288                            [(and (primitive? e0) (equal? 'isZero? e0)) `(isZero? ,(car e1))]
289                            [else `(,e0 ,e1 ...)])]))
```

En el proceso llamado remove-logical-operators nos encargamos de eliminar las primitivas and, or, not y isZero?, por ejemplo, isZero? es equivalente a un if donde se pregunta si el parámetro es 0, si esto sucede regresamos verdadero y falso en otro caso.

```
291  ;; Proceso que elimina las primitivas and, or y not
292  (define-pass remove-logical-operators : LIF (ir) -> L4 ()
293    (Expr : Expr (ir) -> Expr()
294          [(not ,[e]) `(if ,e #f #t)]
295          [(or ,[e0] ,[e1]) `(if ,e0 #t ,e1)]
296          [(and ,[e0] ,[e1]) `(if ,e0 ,e1 #f)]
297          [(isZero? ,[e0]) `(if (equal? ,e0 0) ,#t #f)]))
```

Para las operaciones restantes $<$, $>$ y equal? además de reemplazarse con una equivalencia se debe tener cuidado con el tipo, por lo que en el proceso arit-bool se identifican estos operadores para regresar el tipo correcto.

```
43  ;; Verifica si es una primitiva que regresa un booleano
44  (define (arit-bool? x) (memq x '(< > equal?)))
```

4. Etapas

1. Front-End

   a) remove-one-armed-if: Proceso del compilador encargado de unificar if con una condicional

   b) remove-string: Proceso del compilador encargado de unificar if con una condicional

   c) curry-let: Este proceso encargado de currificar las expresiones let y letrec.

   d) identify-assigments: Este proceso se detecten los let utilizados para definir funciones y se remplazan por letrec.

   e) un-anonymous: Este proceso encargado de asignarle un identificador a las funciones anónimas (lambda).

   f) verify-arity: Proceso que funciona como verificador de la sintaxis de las expresiones.

   g) verify-vars: Proceso que consiste en verificar que la expresión no tenga variables libres. De existir variables libres, regresa un error; si no hay varibales libres, la salida es la misma expresión.

2. Middle-End

   a) curry: Proceso que se encarga de currificar las expresiones lambda así como las aplicaciones de función.

b) type-const: Proceso que se encarga de colocar las anotaciones de tipos correspondientes a las constandes de nuestro lenguaje.

c) type-infer: Proceso que se encarga de quitar la anotación de tipo Lambda y sustituirlas por el tipo $(T \to T)$ que corresponda a la definición de la función. Sustituye las anotaciones de tipo List por el tipo $(ListofT)$

d) uncurry: Proceso uncurry encargado de descurrificar las expresiones lambda de nuestro lenguaje.