Compiladores 2020-1
Facultad de Ciencias UNAM
Proyecto Final

Juan Yair Chiu Valderrama 314114910 Ricardo Badillo Macías 314004051
Edith Araceli Reyes López 314212102 Luis Eduardo Martinez Hernandez 314240028
Emmanuel Cruz Hernandez 314272588 Daniela Susana Vega Monroy 314582915

12 de diciembre de 2019

1. Manual de Uso

En esté manual procederemos a explicar usos básicos de nuestro lenguaje, así como generar código y como compilarlo para generar un .py que pueda ser usado en cualquier interprete de python 3.
Nuestro lenguaje esta mas inclinado al paradigma funcional y nos permite hacer trabajar con operaciones aritméticas y lógicas, en la forma más básica y definir funciones. A continuación daremos definiciones para poder definir un programa en nuestro lenguaje.

Para compilar nuestro programa una vez ya escrito (utilizando el lenguaje LF) en un archivo con terminación .mt.
Nuestro compilador se llama çompilador.rkt", para utilizarlo debemos de colocar en una terminal el siguiente comando:
racket compilador.rkt
Una vez hecho esto, damos enter e inmediatamente después escribimos el nombre del archivo y damos enter de nuevo.
Si el archivo es compilado exitosamente el programa, se habran creado tres archivos con terminaciones .fe , .me y .py.
Tambien es posible utilizar DrRacket para compilar el archivo, tomando en cuanta que esta opción es mas lenta.

Estructura -

Mas abajo explicaremos como funciona la definición del lenguaje, consecuentemente si se cumple la definición de está, la estructura será un programa valido para nuestro lenguaje. Debe remarcar que cualquier parentesis que se abre se debe cerrar y que ninguna variable debe repetirse su identificador. Si alguna de estas acciones no se cumple puede que el compilador no detecte correctamente los errores y no funcione el mismo.

Tomemos en cuenta que la entrada es un archivo .mt donde se tendra la expresión en lenguaje fuente. No es necesario agregar el símbolo ' al inicio.

Además cada línea representa un bloque de código independiente. En caso de separa la misma expresión en distintas líneas habrá un error. También es importante denotar que el alcance de variables solo llega a su propio bloque exceptuando las variables declaradas con define.

Errores -

Cuando compilamos algún programa es inevitable que alguna vez cometamos un error ya sea tipificación, síntaxis o semántica, para darle a conocer al usuario algún error que haya cometido le daremos a saber errores, sí estos fueron cometidos al escribir el programa así tenemos:

- "Se esperaban 2 tipos"
- "Se esperaban mas parametros." (remove-primB)
- "Arity mismatch."
- "Free variable" (metodo verify-vars)
- "No puedes regresar un tipo Void" (metodo J)

- ”La Lista debe ser homogenea” (metodo J)
- ”Se esperaba tipo Bool” (metodo J)
- ”No unificable” (metodo J)
- ”Se esperaba tipo Bool en la condición” (metodo J)
- ”Se esperaba tipo List of T” (metodo J)
- ”Aridad Incorrecta” (metodo J)
- ”Se esperaba tipo Int” (metodo J)
- ”El tipo no corresponde con el valor” (metodo J)
- ”Se esperaba tipo función” (metodo J)
- ”Expresion Invalida” (metodo J)
- ”Mala definición de función” (python)
- ”No se definio la función previamente.” (python)
- ”El archivo debe tener terminación .mt” (compilador)

El archivo de entrada
Unos ejemplos de unos programas en nuestro lenguaje serían

ENTRADA ".mt"

```
(for [x (list 1 2 3)] (if #t 6 4))
```

SALIDA ".py"

```
for x in [1, 2, 3]:
  if True:
    6
  else:
    4
```

ENTRADA ".mt"

```
(length (list 1 2 3 4))
(car (list 1 2 3 4))
(cdr (list 1 2 3 4))
```

SALIDA ".py"

```
x1 = [1, 2, 3, 4]
len(x1)

x2 = [1, 2, 3, 4]
x2[0]

x3 = [1, 2, 3, 4]
x3[1:len(x3)]
```

ENTRADA ".mt"

```
(lambda ([x Int]) (+ x 5))
```

SALIDA ".py"

```
def foo0(x):
  x1 = x
  x2 = 5
  return x1 + x2
```

ENTRADA ".mt"

```
(letrec ([x Lambda (lambda ([y Int]) (+ y 6))]) x)
```

SALIDA ".py"

```
def x():
  def foo0(y):
    x1 = y
    x2 = 6
    return x1 + x2
x
```

ENTRADA ".mt"

```
(letrec ([x Int (lambda ([y Int]) (+ y x))]) x)
```

SALIDA ".py"

```
verify-vars: Free variable

(x)
```

ENTRADA ".mt"

```
(if #t 1 2)
(define a #t)
(if a 4 5)
```

SALIDA ".py"

```
if True:
  1
else:
  2
global a
a = True
if a:
  4
else:
  5
```

ENTRADA ".mt"

```
(lambda ([x Int]) (if #t 5 6))
```

SALIDA ".py"

```
def foo0(x):
  if True:
    return 5
  else:
    return 6
```

2. Especificación formal del lenguaje fuente

LF

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

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

<cons> ::=  <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 | length | car | cdr | < | > | ++ | -- | equal?
    | iszero?
```

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

3. Especificación de Nuevos Lenguajes

Se definió un nuevo lenguaje L4PA para poder eliminar las operaciones aritmeticas $++$, $--$ e *iszero*? del lenguaje.
Gramática L4PA

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

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

<cons> ::=  <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>)

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

<primA> ::= + | - | * | / | and | or | length | car | cdr | not | < | > | equal?

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

4. Justificaciones del Diseño

- Variables Globales.

Para poder incluir la expresión (define x e) tuvimos que fijarnos primero de como era su comportamiento, para que está tenga sentido en el lenguaje y pueda tener un efecto sobre el programa, es necesario que se encuentre definida dentro de un (begin e* ... e). Donde no importa donde aparezca siempre y cuando no sea la última expresión.
Lo anterior mencionado sería el caso si solo estuvieramos trabajando en un bloque de código, en el caso que tengamos varios, se puede tener la instrucción sin necesidad de meterla en un begin.
Pero en ambos casos es importante recalcar que solo puede afectar a las expresiones que se encuentren por debajo de está, ya que de utilizar la variable que define en una parte superior provocaría un error.
Para poder agregar define al lenguaje se tuvo que hacer los siguientes cambios:
Fue necesario agregarlo en el lenguaje fuente.
Los procesos que afectan más son los que utilizan un contexto, por ejemplo el proceso que se encarga de aplicar el algoritmo J. Para poder llevar a cabo estos, se modificaron para que al mismo tiempo utilizaran un contexto global (el cual se calcula con su propia función). De esta forma podemos saber cuando una variable esta definida o no.
Al momento de agregar esta expresión al algoritmo J, verificamos que la expresión e no sea del tipo Lambda o función, ya que si fuera de este tipo nos ocasiona errores en la traducción.

- For each. Para incluir un for each element in a list en el lenguaje se decidio de no traducir este for each en un while debido a la imposibilidad de asignar variables en un lenguaje temprano. Esta desicion la hicimos basandonos en el hecho de que para traducir un foreach en un while necesitamos asignar variables, pero en los lenguajes tempranos no podemos utilizar el algoritmo J para obtener un tipo y por lo tanto no podemos utilizar let. De la misma forma no se puede asignar variables con un define por los errores que se tenian a la hora de hacer la traduccion final. Por lo que se tuvo que incluir en el lenguaje.
  Para incluir el for each en el lenguaje se hicieron los siguientes cambios: Se incluyo en el lenguaje fuente, se incluyo en el proceso de Fv para encontrar variables libres que puedan estar en el cuerpo y en la expresion e0 de la asignacion y finalmente se incluyo en el algoritmo J porque toda expresion de nuestro lenguaje debe de tener un tipo. La inferencia sobre el algoritmo J se hizo infiriendo el tipo de e0 y despues verificando que sea de tipo lista para finalmente regresar el tipo del cuerpo.
  Decidimos solo modificar estos procesos porque eran los unicos que afectaba si teniamos un for each.

- While. Para incluir el while en el lenguaje se tuvo que incluir en el lenguaje fuente. Además se tuvo que modificar el proceso de globalVar y la función J. Esta decisión la tomamos porque son los unicos procesos en los cuales tiene sentido incluir a while. Dentro del proceso globalVar se obtienen las variables que pudieran estar en la condicion del while y en el cuerpo del while,es por esta razón que se debe de incluir el while en este proceso.
  Fue necesario incluir el while en la funcion J porque while es un constructor del lenguaje y todos los constructor del lenguaje deben detener un tipo. La modificación al algoritmo J fue ,que para inferir el tipo de un while se tiene que verificar que el tipo de la condición sea un boolean y de ser un boolean se regresa el tipo de la condición.

- Operaciones aritmeticas extendidas. Se incluyeron las operaciones $< > ++ -- \ equal? \ iszero?$, pero notamos que las que se podian eliminar eran el $++$, $--$, iszero?, ya que tanto el $++$ y el $--$, es solo sumar 1 o restar 1.Ademas el $iszero?$ y es solamente utilizar el $equal?$ con 0. Y las sustituimos con las siguientes equivalencias:

  - $++ \ x \rightarrow + \ x \ 1$
  - $-- \ x \rightarrow - \ x \ 1$
  - $iszero? \ x \rightarrow equal? \ x \ 0$

  Como las operaciones que no se pudieron sustituir fueron los $< > equal?$ entonces, tenemos que hacer lo mismo que hicimos en prácticas pasadas, verificar su aridad, verificar en el algoritmo J que lo que nos esten pasando sean numeros. Pero es análogo a lo que hicimos en las prácticas.

5. Explicación por etapa

Antes que nada nuestro compilador verifica que el archivo que nos este pasando sea un archivo con terminación ".mt", en caso de que no sea manda un error. En otro caso empieza con la etapa Front-End.

- Front-End: Es la Fase del compilador en donde a partir del codigo fuente este se traduce a una representación intermedia. En esta fase el compilador nos detecta los errores sintácticos en el programa a compilar. Tenemos los siguientes procesos que conforman a esta fase:

  - make-explicit : Proceso que cambia el cuerpo de lambda, let y letrec por un begin.
  - remove-one-armed-if: Proceso que elimina los if con dos parametros.
  - remove-string: Proceso que elimina las cadenas y los convierte en una lista de chars.
  - remove-primA: Proceso que elimina las operaciones aritmeticas (++, –, iszero?) y las convierte en operaciones equivalentes.
  - remove-logical-operators: Proceso que sustituye las operaciones booleanas (and,or,not) por equivalencias de ifs.
  - curry-prim: Proceso que currifica las expresiones primitivas.
  - eta-expand: Proceso que añade lambdas cada vez que ve una primitiva.
  - quote-const: Proceso que añade un constructor nuevo a las constantes.
  - purify-recursion: Proceso que verifica que las expresiones letrec asignen únicamente expresiones lambda a variables, y si es necesario añade let a letrec.
  - direct-app: Proceso que traduce una aplicaciónn de función a un expresión let
  - curry-let: Proceso que currifica una expresión let o letrec
  - indentify-assigments: Prceso que detecta los let utilizados para definir funciones y se reemplazan por letrec
  - un-anonymous- Proceso encargado de asignarle un identificador a las funciones anónimas.
  - verify-arity: Proceso que verifica la aridad de las operaciones.
  - verify-vars: Proceso que se encarga de ver que no haya variables libres.

  Y tenemos las siguientes funciones auxiliares para la realización de algunos procesos en la fase:

  - union: Función que hace la unión de dos listas.
  - globalVar: Función encargada de encontrar las variables globales
  - Fv: Función encargada de ligar las variables libres cuando estas son definidas, es decir las elimina de la lista de variables libres.

  Una vez pasado por todas las funciones y procesos de esta fase sin que hubiera errores de sintaxis,el compilador con el proceso **compilador** genera un archivo con terminación **.fe** que tendra lo que obtuvimos en esta fase, y se lo pasaremos como entrada a la fase de Middle-End.

- Middle-End: En esta fase es donde ocurre todas las optimizaciones, el compilador nos avisará si hemos cometido errores semánticos. Tenemos los siguientes procesos que conforman a esta fase:

  - curry: Proceso que se encarga de currificar las expresiones lambda y la aplicacion.
  - type-const: Proceso que currifica una expresión let o letrec.
  - type-infer: Proceso que de ser necesario se encarga de quitar las anotaciones de tipo Lambda y de Tipo List.
  - uncurry: Proceso que se encarga de descurrificar las expresiones lambda de nuestro lenguaje.

  Y tenemos las siguientes funciones auxiliares para la realización de algunos procesos en la fase:

  - buscar: Función que busca el tipo de una variable en el contexto.
  - tipoLista: Funcion que verifica que sea tipo List of T.
  - tipoFuncion: Función que verifica que sea tipo Función.
  - getTipo: Función para obtener el tipo que debe de regresar una primitiva.
  - union-hash: Función que une dos diccionarios.
  - agregar: Función que hace la unión de los diccionarios que generan expresiones en una lista.

- verif: Función que verifica que no se regresa un tipo Void
- getCtx: Función que obtiene el contexto.
- getTable: Función que genera la tabla de símbolos de una expresión del lenguaje.
- J: Función que implementa el algoritmo J.

Una vez pasado por todas las funciones y procesos sin ningun error de semantica , el compilador con el proceso **compilador** genera un archivo con terminación **.me** que tendra lo que obtuvimos en esta fase y se lo pasaremos como entrada a la fase Back-End que sera la encargada de pasarlo a Python.

- Back-End: Esta fase son las ultimas optimizaciones, es aqui donde se genera el código que sera ejecutado, una vez que se realizaron las fases anteriores. Tenemos los siguientes procesos que conforman a esta fase:

  - list-to-array: Proceso encargado de traducir las listas del lenguaje en arreglos.
  - python: Proceso que se encarga de traducir las expresiones de nuestro lenguaje en expresiones del lenguaje de programación Python.

  Y tenemos las siguientes funciones auxiliares para la realización de algunos procesos en la fase:

  - elimElem: Función que elimina los últimos f elementos de la expresión.
  - genElem: Función para generar una lista de elementos.
  - genBody: Función que genera una cadena traducida con una lista de expresiones.
  - regresa: Función que genera el cuerpo de una función.
  - genFuncion: Función que genera los parametros que recibe la función como el cuerpo de esta.
  - constante: Función que regresa el valor de una constante, en caso de que sea un valor booleano revisa si es true o false y lo regresa.
  - aux-python: Función que realiza la traducción a Python.

  Una vez pasado por todos los procesos y funciones de esta fase, el compilador con el proceso **compilador** genera un archivo con terminación **.py**, el cual sera el programa que este listo para ejecutarse.