Compiladores 2020-1
Facultad de Ciencias UNAM
Proyecto Final

Lourdes del Carmen González Huesca Javier Enríquez Mendoza*

20 de noviembre de 2019
Fecha de entrega: 9 de diciembre de 2019

1. Especificación

Durante el semestre se estuvo construyendo en el laboratorio un compilador de expresiones de nuestro lenguaje fuente `LF` al lenguaje de programación 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 estructura del compilador y siendo el último de estos archivos uno con extensión `.c`.
Los procesos que conforman nuestro compilador completo son:

1. `remove-one-armed-if`
2. `remove-string`
3. `curry-let`
4. `identify-assigments`
5. `un-anonymous`
6. `verify-arity`
7. `verify-vars`
8. `curry`
9. `type-const`
10. `type-infer`
11. `uncurry`
12. `list-to-array`
13. `c`

El compilador consiste en aplicar un proceso tras otro comenzando con un código en lenguaje `LF`, pasando por una serie de lenguajes intermedios definidos durante el semestre y terminando en C.
El compilador tiene una estructura general y que organiza las etapas usuales de compilación. En nuestro caso:

- Front-end que lo conforman los procesos 1 - 7
- Middle-end que lo conforman los procesos 8 - 11
- Back-end que lo conforman los procesos 12 - 13

que serán los resultados intermedios expresados en los archivos auxiliares.

*javierem_94@ciencias.unam.mx

2. Ejercicios

A continuación se describen los ejercicios que pueden ser implementados para el proyecto: los marcados con $*$ son obligatorios y dependiendo del número de integrantes del equipo será el número de ejercicios que se entregarán [1].

1. 🔴 $*$ Extender el lenguaje fuente con el constructor `(define x e)` para definir variables globales. Se debe manejar este constructor en todos los procesos del compilador sin alterar su semántica.

2. 🟡 $*$ Extender el lenguaje fuente con el constructor `(while [e0] e1)` que representa un ciclo while, es decir se va a ejecutar la expresión `e1` mientras `e0` se evalúe a verdadero. Se debe manejar este constructor en todos los procesos del compilador sin alterar su sémantica.

3. 🟡 Extender el lenguaje fuente con el constructor `(for [x e0] e1)` que representa un ciclo for sobre una lista, es decir se va a ejecutar la expresión `e1` por cada uno de los elementos de la lista que resulta de la evaluación de `e0`. Si `e0` no fuera una lista entonces se lanza un error (las cadenas son consideradas como listas de caracteres). Se debe manejar este constructor en todos los procesos del compilador sin alterar su sémantica.

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.

5. 🔴 Implementar en un proceso del compilador la optimización conocida como *Continuation Passing Style* aplicado sobre las funciones. Este proceso puede ser parte del Middle-end o Back-end de nuestro compilador.

6. 🔴 Agregar al lenguaje fuente el constructor `(lambda(x) e)` para definir funciones polimórficas, es decir funciones sin anotaciones explícitas de tipos. En este ejercicio hay que tener especial cuidado pues aunque el añadir el constructor resulta sencillo, la inferencia de tipos y la traducción a C pueden ser complicadas.

7. 🟡 En la traducción a C, que se implementó en la práctica 7, las listas son traducidas a arreglos y no se permite la construcción de una lista vacía ni una lista con expresiones como elementos. Mejorar esta traducción de tal forma que ya no existan estas restricciones y que por ejemplo se puedan tener listas de funciones.

8. 🔴 Agregar las operaciones primitivas de listas: `equal-lst?` (igualdad), `empty?` (recibe una lista y decide si ésta es vacía), `elem?` (recibe un elemento y una lista y decide si este elemento es parte de la lista), `append` (agrega un elemento al final de la lista), `concat` (concatena dos listas) y `cons` (agrega un elemento como cabeza de una lista).

9. 🔴 $*$ Implementar un programa en Racket que compile un archivo con la extensión `.mt` pasando por todos los procesos definidos durante el semestre. Este programa debe generar tres archivos intermedio, uno correspondiente a cada conjunto de procesos descritos arriba. Por ejemplo, si

[1] Equipos de 6 personas realizar un total de 7 ejercicios, equipo de 5 personas un total de 6 ejercicios y de 4 personas o menos 5 ejercicios en total. **Los ejercicios marcados con $*$ son obligatorios**

se compila el archivo `example.mt` se generan los archivos `example.fe` que corresponde al archivo generado a partir de aplicar los procesos correspondientes a la etapa de Front-end, el archivo `example.me` correspondiente a aplicar los procesos que corresponden al Middle-end y el archivo `example.c` que es un programa en C equivalente al programa escrito en el lenguaje fuente.

10. 🟡 Implementar dentro del mismo programa en Racket la traducción a código ensamblador **Hint:** Utilizar un compilador de C a ensamblador.

11. **Ejercicio extra: hasta 1.5 pts** Análisis de los tiempos de ejecución de los procesos en cada etapa además de calcular los tamaños de los archivos obtenidos.

Dificultad Alta
Dificultad Media
Dificultad baja

3. Entrega

- La entrega será en el repositorio de GitHub Classroom.
  El enlace de la actividad es `https://classroom.github.com/g/mRFW5GpB`
- Este proyecto debe realizarse en equipos de máximo 6 personas.
- Además de los archivos que contengan la solución de los ejercicios escogidos, se debe agregar un archivo Readme.pdf en el que se incluya
  - Documentación del compilador. Ésta debe contener:
    - EL manual de uso.
    - Especificación formal del lenguaje fuente.
    - Justificaciones del diseño.
    - Explicación de cada etapa y cada archivo de salida.
  - Los nombres y números de cuenta de los integrantes del equipo.
  - En caso de haber sido necesario, la especificación formal de los nuevos lenguajes definidos, utilizando gramáticas.

  El pdf debe obtenerse a partir de LaTeX. De no incluirse o no cumplir con los requerimientos no se calificará el proyecto.
- Se recibirá el proyecto hasta las 23:59:59 horas del día fijado como fecha de entrega, cualquier proyecto recibido después no será tomado en cuenta.
- **Cualquier código total o parcialmente plagiado, implicará que el proyecto sea calificado automáticamente con cero.**