



UNIVERSIDAD SIMÓN BOLÍVAR  
DECANATO DE ESTUDIOS PROFESIONALES  
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN

**Implementación de un compilador nativo para GCL**

Por:  
Joel Orlando Araujo Dos Santos  
José Luis Jiménez Betancourt

PROYECTO DE GRADO  
Presentado ante la Ilustre Universidad Simón Bolívar  
como requisito parcial para optar al título de  
Ingeniero en Computación

**Sartenejas, diciembre de 2015**



**UNIVERSIDAD SIMÓN BOLÍVAR  
DECANATO DE ESTUDIOS PROFESIONALES  
COORDINACIÓN DE INGENIERÍA DE LA COMPUTACIÓN**

**Implementación de un compilador nativo para GCL**

Por:  
Joel Orlando Araujo Dos Santos  
José Luis Jiménez Betancourt

Realizado con la asesoría de:  
Ernesto Hernández-Novich

**PROYECTO DE GRADO**  
Presentado ante la Ilustre Universidad Simón Bolívar  
como requisito parcial para optar al título de  
Ingeniero en Computación

**Sartenejas, diciembre de 2015**

  
UNIVERSIDAD SIMÓN BOLÍVAR  
VICERRECTORADO ACADÉMICO  
DECANATO DE ESTUDIOS PROFESIONALES  
**Coordinación de Computación**

**ACTA DE EVALUACIÓN DE PROYECTO DE GRADO**

CÓDIGO DE LA ASIGNATURA: EP-3308 FECHA: 26-01-2016

TÍTULO DEL TRABAJO: IMPLEMENTACIÓN DE UN COMPILADOR NATIVO PARA GCL

NOMBRE DEL ESTUDIANTE: JOEL ARAUJO CARNÉ: 10-10797

TUTOR: PROF. Ernesto Hernández

JURADOS: \_\_\_\_\_

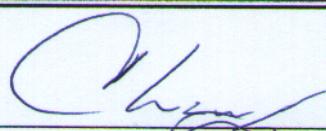
APROBADO:

REPROBADO:

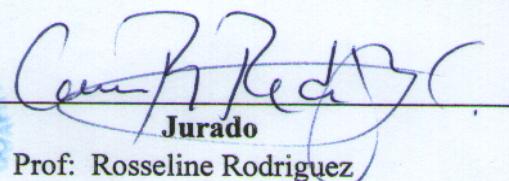
OBSERVACIONES: \_\_\_\_\_  
\_\_\_\_\_

El jurado considera **por unanimidad** que el trabajo es EXCEPCIONALMENTE BUENO:

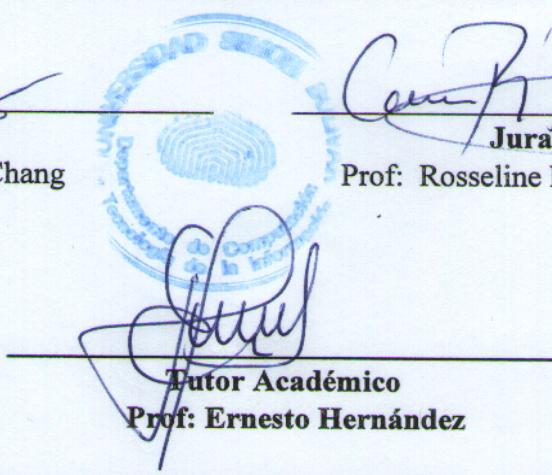
SI:      NO:      En caso positivo, justificar razonadamente: \_\_\_\_\_  
\_\_\_\_\_

  
\_\_\_\_\_  
Jurado

Prof: Carolina Chang

  
\_\_\_\_\_  
Jurado

Prof: Rosseline Rodriguez



Tutor Académico  
Prof: Ernesto Hernández

**Notas:** Colocar los sellos de los respectivos Departamentos Académicos. Para jurados externos usar el sello de la Coordinación. Este documento debe ir sin enmiendas



**UNIVERSIDAD SIMÓN BOLÍVAR  
VICERRECTORADO ACADÉMICO  
DECANATO DE ESTUDIOS PROFESIONALES**

## **Coordinación de Computación**

## **ACTA DE EVALUACIÓN DE PROYECTO DE GRADO**

CÓDIGO DE LA ASIGNATURA: **EP-3308** FECHA: 26 -01-2016

FECHA: 26-01-2016

## TÍTULO DEL TRABAJO: IMPLEMENTACIÓN DE UN COMPILADOR NATIVO PARA GCL

NOMBRE DEL ESTUDIANTE : JOSÉ JIMÉNEZ CARNÉ: 10-10839

**TUTOR: PROF. Ernesto Hernández**

**JURADOS:** \_\_\_\_\_

APROBADO: 

**REPROBADO:**

## OBSERVACIONES:

El jurado considera **por unanimidad** que el trabajo es EXCEPCIONALMENTE BUENO.

SI:      NO:      En caso positivo, justificar razonadamente:

20

Prof: Carolina Chang

  
Jurado  
Prof: Rosseline Rodriguez

Tutor Académico  
Prof: Ernesto Hernández

**Notas:** Colocar los sellos de los respectivos Departamentos Académicos. Para jurados externos usar el sello de la Coordinación. Este documento debe ir sin enmiendas.

## Resumen

En la Universidad Simón Bolívar los estudiantes de Ingeniería de Computación diseñan sus primeros programas en el curso teórico de Algoritmos y Estructuras I usando el lenguaje de programación GCL, diseñado por Edsger Dijkstra. Al mismo tiempo, en el curso de Laboratorio de Algoritmos y Estructuras I usan un lenguaje de programación escogido por los profesores encargados de la materia para practicar lo aprendido en la teoría. Un problema es que no existe una buena relación entre ambas, debido a que los lenguajes usados en la práctica no refuerzan ni promueven las técnicas aprendidas gracias al uso de GCL en el curso teórico. Esto obliga a los estudiantes, que en su mayoría no tienen experiencia previa programando, a transformar sus programas en GCL al lenguaje particular para poder usarlos en la práctica. Este Proyecto de Grado nace de la necesidad de crear un lenguaje de programación basado en GCL con su respectivo compilador para ser usado en el curso de laboratorio.

El desarrollo fue llevado a cabo en tres etapas. La primera consistió en diseñar la gramática del lenguaje y la toma de decisiones de diseño del compilador. Para definir la gramática se partió de las instrucciones del lenguaje GCL y se añadieron otros mecanismos con el fin de obtener un lenguaje completo. Adicionalmente, se implantó una calculadora sencilla con el objetivo de obtener mayor familiaridad con las construcciones que ofrece el lenguaje escogido para la implementación, Haskell, y posteriormente se realizó el analizador lexicográfico del lenguaje.

En la segunda etapa continuó la implementación del *Front-End* del compilador por lo que se completó el analizador sintáctico para generar el AST correspondiente al código fuente, así como el diseño de la tabla de símbolos. Posteriormente, con base en el AST generado se completó la verificación de tipos, coherente con las reglas del sistema de tipos del lenguaje.

Por último, la tercera etapa consistió en la culminación del *Front-End* para el compilador y en la conexión del mismo con el *Back-End* provisto por el *Low Level Virtual Machine* (LLVM). Para ello, se transformó el AST proveniente de la verificación de tipos en una estructura de código intermedio LLVM que finalmente fue suministrada al *Back-End* LLVM para la generación de código de máquina correspondiente a la plataforma destino.

**Palabras clave:** GCL, compilador, Haskell, LLVM.

# Agradecimientos

A nuestros familiares, Carmen Betancourt, José Gregorio Jimenez, Luis Miguel Betancourt, Maria F. Dos Santos, Carlos Ventura, Adriana Araujo, y Oscar Araujo, por su apoyo incondicional en todo momento, en especial a la Sra. Carmen Betancourt por la deliciosa comida que nos preparaba.

A nuestro profesor y tutor Ernesto Hernández-Novich, por toda la ayuda y explicaciones que nos brindó en el desarrollo y por su calidad en todas las clases de la cadena de Lenguajes de Programación y de Programación Funcional Avanzada, ya que sin ellas no podríamos haber hecho este Proyecto de Grado.

Al profesor Ricardo Monascal, ya que sin su ayuda y cooperación no hubiera sido posible realizar este Proyecto de Grado.

A nuestra coordinadora Marlene Goncalves, por todos los consejos y ayuda que nos brindó a la hora de realizar este Proyecto de Grado.

A Samantha Campisi por su apoyo incondicional y ayuda invaluable en la redacción de este libro.

A nuestros compañeros y amigos: Ramón Márquez, José Noel Mendoza, Andrea Salcedo, Reinaldo Verdugo, Emmanuel Piñate, Marcos Campos, Aileen Moreno, Esteban Oliveros, Arturo Voltattorni, Gabriela Limonta y Gabriela Pérez por compartir tantos momentos de risa y soporte, capaces de aliviar cualquier tensión.

A nuestras *Playstation 4* por todas las secciones de los juegos *FIFA* y *Rocket League* que disfrutamos para descansar un poco del Proyecto de Grado.

A todas las bandas y conciertos de música que escuchamos mientras desarrollábamos el código, por hacernos el trabajo más ameno.

# Índice general

<b>Índice general</b>	<b>vii</b>
<b>Introducción</b>	<b>1</b>
<b>1. Planteamiento del problema</b>	<b>4</b>
<b>2. Marco Teórico</b>	<b>6</b>
2.1. Semántica Axiomática y Lógica de Hoare . . . . .	6
2.2. <i>Guarded Command Language (GCL)</i> . . . . .	7
2.3. Implementaciones existentes basadas de GCL . . . . .	13
2.4. Gramática Libre de Contexto . . . . .	14
2.5. Análisis de programas . . . . .	16
<b>3. Marco tecnológico</b>	<b>17</b>
3.1. Haskell . . . . .	17
3.2. <i>Low Level Virtual Machine (LLVM)</i> . . . . .	21
<b>4. Desarrollo</b>	<b>22</b>
4.1. Primera Etapa . . . . .	22
4.2. Segunda Etapa . . . . .	25
4.2.1. Tabla de Símbolos . . . . .	25
4.2.2. Árbol de Sintaxis Abstracta . . . . .	27
4.2.3. Verificación de Tipos . . . . .	27
4.2.4. Cuantificadores . . . . .	28
4.3. Tercera Etapa . . . . .	29
4.3.1. Arreglos . . . . .	33
4.3.2. Entrada y Salida . . . . .	34

<b>5. Resultados</b>	<b>35</b>
5.1. Diseño del lenguaje y detalles del compilador . . . . .	35
5.1.1. Tipos . . . . .	36
5.1.2. Operadores . . . . .	38
5.1.3. Aserciones . . . . .	38
5.1.4. Instrucciones . . . . .	40
5.1.5. Funciones . . . . .	44
5.1.6. Procedimientos . . . . .	45
5.1.7. Variables y Constantes . . . . .	47
5.1.8. Lectura de variables . . . . .	50
5.1.9. UTF-8 . . . . .	51
5.1.10. Cuantificadores . . . . .	51
5.1.11. Funciones Predefinidas . . . . .	53
5.1.12. Errores de compilación . . . . .	53
5.1.13. Errores de ejecución . . . . .	54
<b>6. Conclusiones y recomendaciones</b>	<b>56</b>
<b>Bibliografía</b>	<b>59</b>
A. Sistema de pruebas basado en Tripletas de Hoare	62
B. Teoría de tipos	65
C. Tabla de operadores	67
D. Flujo de datos del compilador	69
E. Funciones Predefinidas	71
F. Gramática del lenguaje	73
G. Instrucciones de instalación	82
H. Manual de Usuario y Programas Ejemplos	83
I. Archivos del Compilador	84

<b>J. Instrucciones de instalación</b>	<b>85</b>
<b>K. Glosario de términos</b>	<b>86</b>
K.1. Acrónimos . . . . .	86
K.2. Definiciones . . . . .	87
K.3. Librerías . . . . .	91

## Introducción

Existen numerosos lenguajes de programación disponibles en la actualidad, cada uno diseñado con la intención de asistir en la resolución de problemas particulares. Sin embargo, es difícil encontrar un candidato idóneo cuando se quiere seleccionar un lenguaje para la enseñanza. La mayoría de los lenguajes de programación populares hacen énfasis en ofrecer herramientas que soporten técnicas de programación que requieren disciplina y experiencia, sacrificando formalidad, e introduciendo complejidad asociada con el conjunto de problemas que ayudan a resolver. Esto hace complicado enseñar a programar apoyándose en esos lenguajes.

El curso de Algoritmos y Estructuras I es el primero donde los estudiantes de Ingeniería de Computación de la Universidad Simón Bolívar aprenden técnicas para la resolución de problemas básicos utilizando el lenguaje GCL (*Guarded Command Language*), propuesto por Edsger Dijkstra. De igual manera, se imparten los conceptos y principios fundamentales, tales como las estructuras básicas de control, las reglas de alcance de identificadores y el paso de parámetros.

Al mismo tiempo, los estudiantes realizan sus primeros programas en el curso de Laboratorio de Algoritmos y Estructuras I, en el cual los profesores que imparten la materia eligen un lenguaje de programación particular para asignar a los estudiantes tareas, pruebas y proyectos enfocados en reforzar las técnicas aprendidas en el curso teórico.

Anteriormente, se utilizaba el lenguaje GaCeLa desarrollado en la USB y basado en GCL. Su compilador se encargaba de traducir el código fuente del estudiante en código en el lenguaje Java, para luego ser ejecutado en la maquina virtual de Java, JVM. Gracias a GaCeLa, existía coherencia entre lo enseñado en la teoría y lo practicado en el laboratorio. Sin embargo, los estudiantes presentaban dificultades al utilizar el compilador debido a la falta de soporte para su mantenimiento. Lo que resultaba en errores de compilación en programas

correctos, y traducciones al lenguaje Java que podían ser incorrectas, por lo que JVM producía errores que los estudiantes no comprendían. Estos errores eran encontrados frecuentemente por los estudiantes y profesores en los cursos donde el lenguaje era utilizado, por lo tanto, los profesores abandonaron su uso y comenzaron a valerse de otros lenguajes ya existentes no enfocados en la enseñanza.

Desde entonces, utilizar en el laboratorio un lenguaje particular que no está basado en GCL, genera que los estudiantes deban hacer trabajo adicional para comprender la sintaxis y las características propias del lenguaje. Por otro lado, no se benefician de las restricciones necesarias para desarrollar la disciplina requerida para escribir programar correctos, las cuales comprenden las verificaciones de tipos fuerte y estática, de la precondición y postcondición, y de los invariantes y función de cota en los ciclos. Evitar estas reglas ocasiona que los estudiantes queden expuestos a técnicas propias del lenguaje en el razonamiento sobre los algoritmos.

Por todo lo anterior, el presente Proyecto de Grado se enfoca en crear un lenguaje de programación orientado al apoyo de la enseñanza y el buen uso de las técnicas básicas de programación necesarias para formar programadores capaces de resolver problemas, independientemente del lenguaje de programación que deban utilizar. Como base para la creación de este lenguaje se tomaron en cuenta las especificaciones de GCL, con el fin de reducir el impacto de la traducción de los programas aprendidos en el curso de teoría al lenguaje utilizado en el laboratorio, así como incentivar al desarrollo de una solución organizada manteniendo un conocimiento a fondo del problema a resolver.

El presente informe final para el Proyecto de Grado se encuentra estructurado en seis capítulos, además de la introducción. El primer capítulo está enfocado en el planteamiento del problema, el cual radica en la falta de un lenguaje de programación dedicado a la enseñanza de algoritmos. El segundo y tercer capítulo comprenden los marcos teórico y tecnológico,

respectivamente, en los cuales se fundamenta el trabajo llevado a cabo en este Proyecto de Grado. El cuarto capítulo describe las etapas del proceso seguidas por el equipo de trabajo para crear el lenguaje bautizado con el nombre de GraCieLa e implantar su compilador. En el quinto capítulo se presentan los resultados, lo que incluye la especificación del lenguaje y el conjunto de reglas para la compilación de los programas. Por último, en el sexto capítulo se dan a conocer las conclusiones y recomendaciones para aquellos dispuestos a continuar el trabajo iniciado en este proyecto.

# Capítulo 1

## Planteamiento del problema

En los cursos de Laboratorio de Algoritmos I y II de la Universidad Simón Bolívar fue utilizado durante varios años el lenguaje de programación GaCeLa basado en GCL. Su compilador no generaba código nativo para la plataforma destino, sino que transformaba el programa escrito en GaCeLa a código en el lenguaje de programación Java para después ser ejecutado usando su máquina virtual, llamada JVM (*Java Virtual Machine*). Sin embargo, en algunas ocasiones la traducción a Java presentaba errores que eran reportados por JVM, obligando a estudiantes que en su mayoría no tenían experiencia previa en programación a manipular código de alto nivel escrito en el lenguaje Java, para lo que no estaban preparados. Además, el compilador no contaba con suficiente personal capacitado para hacer el mantenimiento y actualizaciones necesarias. En consecuencia, GaCeLa fue reemplazado por otros lenguajes; entre ellos Pascal, Modula 2, JML y Python.

En la actualidad, GCL es aún el lenguaje de programación modelo empleado en los cursos de teoría de la USB de modo que es un problema para los estudiantes tener que aplicar un lenguaje disímil en el Laboratorio. Por un lado, se ven en la necesidad de completar una cantidad importante de modificaciones relativas al lenguaje particular, lo que puede llegar a ser una tarea muy complicada. Por otro lado, pierden los elementos de apoyo en el lenguaje

para el razonamiento axiomático sobre sus programas, que es precisamente lo que se quiere enseñar.

Asimismo, el uso de lenguajes de programación de propósito industrial (Python, Java o C) para la enseñanza de algoritmos a programadores aprendices no es recomendable, puesto que fueron especialmente diseñados para favorecer estilos de programación particulares y facilitar el trabajo para programadores experimentados, pero se prestan a confusión en programadores novatos que no tienen la preparación suficiente para adaptarse rápidamente al lenguaje.

Por el contrario, una gran ventaja que trae la enseñanza de algoritmos utilizando un método formal como la programación por contratos apoyada en GCL, es inculcar en los programadores novatos un método para el desarrollo de algoritmos que garantice su correctitud. Así, se forman programadores capaces de desarrollar programas de mejor calidad sin necesidad de realizar su demostración formal, pues el sólo hecho de diseñar el programa tomando en cuenta las técnicas aprendidas garantiza la calidad del mismo. Esto no implica que el código escrito por un programador que aprendió con base en un método formal esté libre de errores. Sin embargo, sí garantiza una menor cantidad de iteraciones en el desarrollo para la resolución efectiva de los problemas.

Por todos los motivos listados anteriormente, es necesaria la creación de un lenguaje general de enseñanza basado en GCL, sencillo de aprender y que no permita las particularidades que promueven los lenguajes de alto nivel. Este Proyecto de Grado tiene como objetivo el desarrollo de un lenguaje basado en GCL con su respectivo compilador, el cual sea diseñado e implantado utilizando herramientas que garanticen la calidad del producto obtenido, y así evitar el uso de otros lenguajes de programación que no tienen relación con el contenido visto en el curso de Teoría de Algoritmos. La intención de desarrollar este lenguaje no es sustituir a aquellos lenguajes usados en la práctica, sino crear uno que promueva el aprendizaje utilizando un método formal como la programación por contratos, apoyada en GCL.

# Capítulo 2

## Marco Teórico

### 2.1. Semántica Axiomática y Lógica de Hoare

Según lo planteado por [25, 26, 29], una semántica formal de un lenguaje de programación es un modelo matemático cuyo propósito es servir de base para la compresión y razonamiento sobre el comportamiento de los programas. En específico, una **Semántica Axiomática** es aquella en la cual las propiedades de las construcciones de un lenguaje son expresadas mediante proposiciones lógicas. Un sistema de pruebas sobre el cual apoyar las propiedades de una semántica axiomática es la **Lógica de Hoare**, el cual es un sistema de proposiciones lógicas sobre las cuales es posible demostrar la correctitud parcial de programas. Las fórmulas en la Lógica de Hoare son llamadas **Tripletas de Hoare** y mantienen la siguiente estructura:

$$\{P\} \ S \ \{Q\}$$

En la fórmula anterior,  $S$  es una instrucción,  $P$  y  $Q$  son llamadas la **Precondición** y la **Postcondición**, respectivamente. Entonces, si la precondición  $P$  es válida en el estado inicial de las variables y la ejecución de  $S$  termina al iniciar en ese estado, entonces la postcondición  $Q$  es válida sobre el estado de las variables al finalizar  $S$ . Esta aproximación es suficiente para

la correctitud parcial de los programas, ya que las postcondición sólo se cumple en el caso que el programa termine; en cambio, en un programa totalmente correcto, si la precondición se cumple, entonces está garantizada su terminación y el cumplimiento de la postcondición. En resumen:

$$\text{correctitud total} = \text{correctitud parcial} + \text{terminación del programa}$$

En el [apéndice A](#) se encuentra un sistema de pruebas basado en tripletas de Hoare para un lenguaje genérico.

## 2.2. *Guarded Command Language (GCL)*

De acuerdo con [22] [24], GCL es un lenguaje de programación diseñado por Edsger Dijkstra, que permite escribir programas junto con su especificación como un teorema para luego demostrar su validez razonando sobre la forma en que las instrucciones modifican el **Estado de las Variables**, el cual determina el conjunto de los valores de las variables definidas dentro del programa durante su ejecución. Por ejemplo, en un programa con dos variables del tipo enteras en total son  $|\mathbb{Z}| \times |\mathbb{Z}|$  posibles estados.

Sobre el estado de las variables se definen **Aserciones** que son expresiones lógicas que permiten acotar el conjunto de estados posibles. Usando el ejemplo anterior, sean X e Y el nombre de dos variables del tipo entero, la aserción  $\{X > 0 \wedge Y < 0\}$  es válida sobre el conjunto de estados y representa todos aquellos estados para los cuales la variable X es mayor que cero y la variable Y es menor que cero.

Cada programa escrito en GCL cuenta con una **Precondición** y una **Postcondición**. Una precondición es una aserción sobre el estado de un programa antes del comienzo de su ejecución. Una postcondición es una aserción sobre el estado del programa después de su ejecución. Demostrar la validez del programa garantiza que, para cualquier estado defini-

do por la precondición, la ejecución del programa terminará en un estado que satisface la postcondición.

La **semántica** del lenguaje es expresada en términos de la **Precondición más Débil** (abreviada *wp*, *weakest precondition*) del programa, que representa a *todos* los estados para los cuales la ejecución de alguna instrucción *S* termina en un estado que satisface una postcondición *Q*. En resumen:

```
{P} S {Q} si y sólo si [P == wp(S, Q)]  
y S termina en un estado que satisface Q
```

Las acciones o comandos permiten modificar el estado de las variables, las cuales se listan a continuación:

### Skip

Es el comando vacío. Es útil en condiciones para las cuales el estado de las variables es posible y la ejecución del programa debe continuar. Se expresa de la siguiente forma:

```
{P} skip {Q}
```

Y la precondición más débil del comando **skip** es la siguiente:

```
[wp('skip', R) == R] para cualquier R.
```

Es decir, la precondición más débil es la misma postcondición.

### Abort

Se utiliza para terminar un programa en cualquier momento durante su ejecución. Si la ejecución alcanza un punto en el cual la instrucción **abort** debe ser ejecutada, entonces las variables definidas en el programa alcanzaron un estado en el cual es imposible continuar con la ejecución.

La precondición más débil del comando **abort** es la siguiente:

[wp('abort', R) == False] para cualquier postcondición  
R.

Es decir, no existe una precondición válida para la ejecución del comando `abort`

## Asignación

La asignación es la única acción que modifica directamente el estado del programa. Permite sustituir el valor de una variable por otro en el estado actual del programa. Se escribe de la siguiente forma:

{P} x := E {Q}

Así, la instrucción `asignación` tiene como efecto sustituir en el estado descrito por  $P$  el valor de la variable  $x$  por el valor de la expresión  $E$ .

La precondición más débil de la asignación es:

[wp('x := E', R) == R[E/x]]

La cual se interpreta de la siguiente forma: La ejecución de la `asignación` terminará en un estado que satisface  $R$ , si  $R$  con el valor de  $x$  sustituido por  $E$  es un estado posible.

## Concatenación

La concatenación permite indicar una secuencia de acciones para ser ejecutadas en orden. De este modo es posible crear acciones compuestas, elaboradas a partir de otras más simples. Así, siendo  $S$  y  $T$  acciones, se puede escribir:

{P} S ; T {Q}

Es equivalente a decir que existe un estado  $R$ , tal que:

{P} S {R} y {R} T {Q}

Es decir, existe un estado  $R$  posible el cual sirve como postcondición de  $S$  y a su vez como precondición de  $T$ . El símbolo  $;$ , se usa como separador de acciones.

La definición de la precondición más débil para la concatenación es la siguiente:

$$[\text{wp}('S; T', Q) == \text{wp}('S', \text{wp}('T', Q))]$$

La precondición más débil de  $T$  con la postcondición  $Q$ , sirve de postcondición para la precondición más débil de  $S$ .

## Selección

La selección permite la elección de una acción a partir de un conjunto de acciones dependiendo del estado de las variables del programa. Es representado de la siguiente forma:

```
{P}
if B0 → S0
:
[] Bi → Si
:
[] Bn → Sn
fi
{Q}
```

A los  $B_i \rightarrow S_i$  se les llama comandos con guardia (*guarded commands*). Los  $S_i$  son acciones del lenguaje y los  $B_i$  son expresiones lógicas conocidas como guardias, las cuales funcionan como condiciones sobre el conjunto de estados. Estas pueden dar valor de cierto o falso dependiendo si el estado cumple con la definición de la guardia.

Durante la ejecución de esta instrucción, se selecciona al de forma no-determinista cualquiera de aquellas guardias  $B_i$  cuyo valor sea cierto y se continuará la ejecución con

la instrucción  $S_i$  asociada. Si ninguna guardia evalúa a cierto, entonces será ejecutada la instrucción **abort**, obligando al programador a incluir todos los escenarios en los cuales es posible ejecutar una instrucción. Definir la selección de esta forma tiene las siguientes ventajas:

- En el caso que un compilador sea implantado para el lenguaje, entonces el programador deja a juicio del diseñador del compilador cuál guardia ejecutar. Un criterio posible es escoger la primera guardia que evalúa a cierto y así evitar la ejecución de código para evaluar el resto de las guardias. Para los estados los cuales más de una guardia es cierta, no importa cuál acción se ejecute; cualquiera debe terminar en un estado posible para la continuación del programa.
- Al ejecutar el comando **abort** cuando ninguna guardia es cierta se obtiene legalidad del código, ya que al obligar al programador a escribir todos los posibles escenarios, cualquier persona que lea el programa podrá entender bajo qué casos se deben ejecutar las instrucciones definidas en el comando con guardia. Incluso, puede haber condiciones para las cuales ninguna acción deba ser ejecutada y para ello existe la acción **skip**, definida anteriormente. Además facilita el aprendizaje, puesto que para los programadores novatos es buena práctica analizar todos los estados para los cuales algún comando guardado debe ser ejecutado. El programador está forzado así a entender el problema que desea solucionar y a desarrollar todas las alternativas de ejecución del programa.

La precondición más débil de la selección es la siguiente:

$$\begin{aligned} [\text{wp(IF, Q)} == (\exists i : 1 \leq i \leq n : B_i) \wedge \\ (\forall i : 1 \leq i \leq n : B_i \rightarrow \text{wp}(S_i, Q))] \end{aligned}$$

La cual puede interpretarse de la siguiente forma: Al momento de la ejecución, alguna guardia debe ser cierta. Esto está asegurado con el primer término de la conjunción. El segundo término garantiza que para todas las guardias, el conjunto de estados definido por  $B_i$  es subconjunto del conjunto de estados que define la precondición más débil de  $S_i$  con la postcondición de la selección. Es decir, cualquier guardia escogida terminará en un estado que satisface la postcondición.

### Repetición

Esta acción permite ejecutar un conjunto de comandos con guardia de manera continua hasta que ninguna guardia se cumpla. Tiene la siguiente sintaxis:

```
{P}
{inv Q}
{bound t}
do  $B_0 \rightarrow S_0$ 
:
[]  $B_i \rightarrow S_i$ 
:
od  $B_n \rightarrow S_n$ 
{Q}
```

Durante la ejecución de esta instrucción, se selecciona de forma no-determinista una guardia  $B_i$  cuyo valor sea cierto y continuará la ejecución con la instrucción  $S_i$  asociada. Posteriormente, se repite el proceso de elección de guardia y ejecución de acción asociada hasta que ninguna evalúe a cierto, en cuyo caso se ejecuta la instrucción **skip**.

Además, para poder demostrar la validez de la instrucción, debe proveerse un **Invariante** y una **Función de Cota**. El invariante es una aserción que debe ser verdadera

antes y después de la ejecución del comando guardado seleccionado, es decir, la ejecución de éste no destruye la validez del invariante. Por otro lado, la función de cota es una expresión numérica, la cual debe ser no negativa y decreciente en cada ciclo de la repetición para garantizar su terminación.

La precondición más débil de la instrucción **repetición** es la siguiente:

$$\begin{aligned} [\text{wp(DO, Q)} & == (\exists k : 0 \leq k : H_k(Q))] \\ H_k(Q) & = H_0(Q) \vee \text{wp(IF, } H_{k-1}(Q)) \text{ para } k > 0 \\ H_0(Q) & = !(\exists i : 1 \leq i \leq n : Bi) \wedge Q \end{aligned}$$

Para  $k = 0$ , la postcondición debe mantenerse y ningún comando guardado debe ser seleccionado para ejecución.

Para  $k > 0$ , existen dos casos. Aquél en que no existe ninguna guardia cierta para la ejecución, en cuyo caso el primer término se cumple, y aquél en que al menos uno se cumple, debiendo asegurar que la ejecución del comando con guardia seleccionado termina en un estado que permite la ejecución de  $k - 1$  repeticiones más, en cuyo caso el segundo término se cumple.

## 2.3. Implementaciones existentes basadas de GCL

El lenguaje GCL diseñado por Edsger Dijkstra posee algunas implementaciones de interpretadores y compiladores basadas en sus características principales.

La primera, el lenguaje llamado **GCL 1.2** [15] trata de un pequeño interpretador escrito en Prolog (solo el Analizador Sintáctico esta implementado en el Lenguaje C) que fue realizado por el profesor Pedro Calabar en el Departamento de Ciencias de Computación de la Universidad de La Coruña (UDC) de España. GCL 1.2, solo permite ejecutar programas muy simples de GCL dado que posee varias restricciones que no permiten realizar programas mas

complejos. Las restricciones son las siguientes: no posee arreglos de múltiples dimensiones, aserciones y constantes, además de no permitir la declaración de funciones y procedimientos. Por otro lado, se encuentra en desuso y su ultima actualización fue en el año 2003.

La segunda, el lenguaje **GaCeLa** [14], trata de un traductor escrito en Java que fue implementado por un grupo de profesores, estudiantes de doctorado y maestría de la Universidad Simón Bolívar (USB) de Venezuela. GaCeLa, verifica la corrección de los programas con respecto a su especificación con el objetivo de inculcar a los estudiantes del Laboratorio de Algoritmos y Estructuras I la disciplina necesaria para desarrollar algoritmos a partir de su especificación. Permite la ejecución de programas complejos dado que su diseño contempla la declaración de funciones, procedimientos, tipos abstractos, y tipos algebraicos. Por último, no se encuentra en uso desde el año 2010 por falta de soporte y su ultimas mejoras ocurrieron en el año 2004.

Por ultimo, existe un modulo para el lenguaje Perl llamado **Commands::Guarded** [12] que implementa una variante de GCL para el lenguaje. El modulo permite efectuar verificaciones para un paso específico, por lo que se comportan como una aserción.

## 2.4. Gramática Libre de Contexto

Basado en lo expuesto por [20, 23, 28], una gramática libre de contexto puede ser definida como un cuádruple  $(V, \Sigma, P, S)$ , en el cual  $V$  es un conjunto finito de variables o símbolos no-terminales,  $\Sigma$  es un conjunto finito de símbolos terminales,  $P$  es un conjunto finito de producciones y  $S$  es un elemento de  $V$  y es llamado el símbolo inicial.

Una producción consiste de una variable perteneciente al conjunto  $V$  la cual está siendo definida por la producción, llamada la cabeza de la producción. Además de una cadena de cero o más variables y símbolos terminales y no-terminales, llamada el cuerpo de la producción, que representan la forma en la que son construidas las palabras pertenecientes a la producción.

Un lenguaje libre de contexto es aquel generado a partir de una gramática libre de contexto  $G$ . Es definido como el conjunto de palabras que pueden ser derivadas usando las producciones de  $G$  a partir de la variable inicial.

Derivar es el proceso de producir palabras con una gramática libre de contexto, construyendo una **Derivación**. Una derivación comienza por el símbolo inicial  $S$ , expandiéndolo con alguna de sus reglas asociadas y, de aparecer nuevos símbolos no-terminales, repite expansiones sucesivamente hasta terminar en una cadena de símbolos terminales. Es importante recalcar que una variable puede ser cabeza de más de una regla, en este caso puede existir más de una derivación para la palabra. Una **Derivación más Izquierda**, es aquella en la cual los símbolos no-terminales son sustituidos de izquierda a derecha, mientras que en una **Derivación más Derecha** son sustituidos de derecha a izquierda. Una gramática libre de contexto es ambigua si es posible derivar una palabra perteneciente al lenguaje de la gramática usando más de una derivación más izquierda, o más de una derivación más derecha.

Un **Reconocedor** es un algoritmo encargado de verificar si una palabra pertenece a un lenguaje libre de contexto buscando una derivación a partir de la gramática correspondiente. Sea  $w$  una palabra a reconocer, existen dos estrategias que pueden ser usadas para reconocerla. La primera es llamada **Reconocedor Descendente** y comienza desde el símbolo inicial  $S$  e intenta encontrar una derivación más izquierda para  $w$ . A partir de esta estrategia es posible definir un programa llamado *Reconocedor Recursivo Descendente* que consiste de un conjunto de procedimientos (uno por cada no-terminal) los cuales emulan cada expansión de una derivación mediante llamadas a procedimientos. La segunda es llamada **Reconocedor Ascendente** e intenta alcanzar al símbolo inicial  $S$  a partir de la palabra  $w$ .

El proceso de reconocimiento es no-determinista porque al momento de transformar una variable cualquier regla con la variable de cabeza puede ser expandida. Pero, existen dos familias de gramáticas que posibilitan un reconocimiento determinista:  $\text{LL}(K)$  y  $\text{LR}(K)$ .

La primera usa un reconocedor descendente y es llamada así porque explora la palabra de izquierda a derecha y el reconocedor genera la derivación más izquierda. Mientras que la segunda utiliza un autómata finito y su nombre se debe a que también explora la palabra de izquierda a derecha, pero produce la derivación más derecha de la palabra. Ambas necesitan  $K$  símbolos terminales por cada símbolos no-terminal (también llamados conjuntos *lookahead*) para decidir cual regla usar en un momento determinado del reconocimiento.

## 2.5. Análisis de programas

A partir de lo propuesto por [27], los lenguajes de programación pueden ser clasificados dependiendo del modo en que se hace el análisis de sus programas. Más información sobre este apartado puede encontrarse en el [apéndice b](#).

# **Capítulo 3**

## **Marco tecnológico**

En el presente capítulo se encuentra la información referente a las herramientas de relevancia que fueron utilizadas durante todo el desarrollo de este Proyecto de Grado, a saber: Haskell y LLVM.

### **3.1. Haskell**

Haskell [17] es un lenguaje de programación funcional puro, de tipos estáticos y polimórficos y con evaluación de expresiones perezosa, en el cual los programas son funciones [4]. La pureza del lenguaje implica que las funciones siempre evaluarán al mismo resultado ante la misma entrada y que además no existe un estado de variables modificable, a diferencia de los lenguajes imperativos. Además, al ser un lenguaje con evaluación de expresiones perezoso, asegura que toda expresión será evaluada en la medida que se necesite, ya que escribir y evaluar una expresión son dos acciones diferentes. Esta característica permite, por ejemplo, la capacidad de declarar listas infinitas, ya que ésta nunca se guarda completamente en memoria sino que se mantienen los valores que haya sido necesario calcular. Como último rasgo resaltante, Haskell es un lenguaje estático y de verificación de tipos fuerte, por lo que toda

expresión tiene un tipo asociado a tiempo de compilación y no permite violaciones al sistema de tipos, razón por la cual todas las conversiones entre tipos deben realizarse de forma explícita.

En vista de que Haskell es un lenguaje puro, necesita de estructuras adicionales para realizar cálculos con efectos de borde. Esta funcionalidad, entre otras, es ofrecida por los *monads* [19]. Los *monads* son estructuras que permiten establecer un contexto para expresar evaluaciones en secuencia, para combinar cálculos puros simples y construir cálculos de complejidad arbitraria; algunos contextos monádicos permiten representar cambios de estado, gestión de excepciones, o entrada y salida.

La comunidad de Haskell ofrece una gran variedad de librerías [13] que permiten el manejo de estructuras complejas, tales como cadenas de caracteres de gran tamaño, diccionarios, árboles n-arios, código intermedio LLVM y *monads* para diversos fines, entre los cuales resaltan reconocedores recursivos descendentes para gramáticas y mantenimiento de estados durante la ejecución de un algoritmo. Además, existe una plataforma diseñada para simplificar la instalación y el uso de Haskell, llamada *Haskell Platform* [16]. Incluye el compilador **ghc**, un programa para la descarga y construcción de librerías y paquetes llamado **cabal**, y las librerías de mayor uso para el desarrollo de aplicaciones en Haskell.

A continuación se presentan las librerías pertenecientes a *Haskell Platform* más relevantes para este Proyecto de Grado.

### 1. `Text.Parsec`

Este módulo facilita la construcción de reconocedores recursivos descendentes. Entre las principales características que ofrece se encuentra la posibilidad de combinar varios reconocedores simples en secuencia para reconocer paso a paso producciones complejas a partir de sus componentes básicos. Esto es posible gracias a que el tipo de datos ofrecido por la librería es monádico y, por lo tanto, su definición contiene operadores

que permiten secuenciación.

Además, se ofrece la posibilidad de seleccionar entre reconocedores con el operador  $p < | > q$ , donde  $p$  y  $q$  son reconocedores. El operador ejecuta el primer reconocedor,  $p$ , y en caso de fallar sin consumir parte de la entrada, ejecuta  $q$ . De igual manera, permite efectuar *backtracking* con la función *try*, la cual recibe un reconocedor y, si al ejecutarlo éste falla después de consumir parte de la entrada, reconstruye dicha entrada a su estado original, es decir, antes de aplicar la operación. Por último, la librería mantiene el nombre del archivo, así como la actualización de fila y columna a medida que se ejecuta el reconocimiento.

## 2. Data.Text [8]

Es un módulo que permite representar cadenas de caracteres. Es más eficiente que la librería por defecto para tal fin, (*Data.String*), puesto que cuenta con un diseño óptimo en términos de tiempo y espacio. Utiliza el estándar Unicode versión 5.2 para la representación de caracteres.

## 3. Data.Map.Strict [6]

Es un módulo que ofrece una implementación del tipo de datos diccionario usando árboles binarios balanceados, junto con todas las funciones necesarias para inserción y búsqueda de datos. La variante **Strict** implica que la librería no evalúa sus expresiones de forma perezosa, por lo tanto, todos los valores almacenados en el diccionario se calculan tan pronto se introducen, previniendo el consumo excesivo de memoria.

## 4. Data.Tree [9]

Es un modulo que ofrece un tipo de datos para árboles polimórficos en los que se permite una cantidad indefinida de ramas por nodo. A este tipo de árbol se le conoce como *Rose Trees*.

La definición de este tipo de datos en Haskell es la siguiente:

```
data Tree a    = Node a (Forest a)
type Forest a = [Tree a]
```

### 5. Control.Monad.RWS.Strict [5]

Este módulo ofrece el *Monad RWS*, diseñado para combinar las cualidades de los *Monads Reader*, *Writer* y *State*. El *Monad Reader* permite tener un ambiente sólo para lectura de la estructura de datos usada por el programador en su declaración. Por su parte, el *Monad Writer* es el registro acumulador de operaciones sólo para agregar modificaciones sobre una estructura de datos. Por ultimo, el *Monad State* es un estado mutable, es decir, permite tanto la lectura de estructuras de datos como también su modificación.

Además, la variante **Strict** garantiza que todas las operaciones sobre el *Monad* evaluarán sus operandos y el resultado correspondiente apenas se ejecute la operación, es decir, opera con evaluación ambiciosa en lugar de la habitual evaluación perezosa del lenguaje. Por lo tanto la estructura se mantiene constantemente evaluada, previniendo así el consumo excesivo de memoria y el diferimiento del trabajo.

Por último se muestran aquellas librerías usadas en este Proyecto de Grado que no están incluidas en *Haskell Platform*, pero han sido desarrolladas por la comunidad de Haskell.

### 1. LLVM.General.Pure [11]

Es un módulo que brinda un tipo de datos con todos los constructores necesarios para representar estructuras de código intermedio LLVM en programas escritos en Haskell.

### 2. LLVM.General [10]

Este módulo facilita a Haskell conectarse con la librerías LLVM del sistema y permitir la traducción de estructuras de código intermedio LLVM que fueron creadas usando la librería `LLVM.General.Pure` a código de máquina.

### 3. Data.Range.Range [7]

Es un módulo que ofrece la posibilidad de crear rangos de valores. Además, brinda un conjunto de operaciones que permiten combinar dos o más rangos, tales como intersección, unión y diferencia.

## 3.2. *Low Level Virtual Machine (LLVM)*

El proyecto LLVM [18] (*Low Level Virtual Machine*) es una colección de tecnologías modulares que permiten reducir el tiempo y costo en la construcción y desarrollo de un compilador. Dentro de las características de LLVM, son relevantes para este Proyecto de Grado el lenguaje de representación intermedia (IR) [2] y el *Back-End* [1] encargado de transformar IR LLVM en código de máquina. Estas herramientas permiten completar el desarrollo de un compilador a partir del *Front-End* implementado para un lenguaje, ya que basta con que éste convierta los programas a IR LLVM y luego el *Back-End* LLVM se encargará de generar código ejecutable nativo para la plataforma destino.

Adicionalmente, el *Back-End* LLVM soporta la generación de código para los CPU's más populares (Intel, AMD, Sparc y MIPS), provee un optimizador de código que incluye una gran cantidad de algoritmos para mejorar el rendimiento de los programas y proporciona herramientas para hacer *Debugging*, *Profiling* y recolección de basura.

# **Capítulo 4**

## **Desarrollo**

El diseño del lenguaje GraCieLa y el desarrollo de su respectivo compilador se llevaron a cabo en tres etapas, desde enero hasta noviembre del año 2015. En la primera etapa, se tuvo como objetivo diseñar el lenguaje, decidir la técnica de análisis sintáctico y entrenar al equipo de trabajo en el lenguaje de programación [Haskell](#) y las librerías necesarias para el desarrollo del proyecto. En la segunda, se elaboró el reconocedor sintáctico del lenguaje para luego efectuar la verificación de tipos. Por último, en la tercera etapa se buscó generar el código intermedio [LLVM](#) y código de máquina correspondientes a la máquina utilizada para compilar.

### **4.1. Primera Etapa**

Esta etapa comprendió el período entre Enero y Marzo de 2015. Su objetivo principal fue completar el diseño de un lenguaje basado en GCL. Fue necesario analizar, discutir y decidir las técnicas para el análisis sintáctico y librerías de apoyo para implantar el compilador, además de mejorar destrezas con el uso de [Haskell](#) y las librerías seleccionadas.

El primer paso consistió en diseñar la gramática del lenguaje. Para esto se realizó una

investigación que incluyó revisar el lenguaje GaCeLa, diseñado e implementado en la USB, y el lenguaje [GCL](#) diseñado por Dijkstra. Después de realizar la investigación, se concluyó que el lenguaje GaCeLa no ofrecía ciertos elementos que pudieran ser tomados por el equipo de trabajo y se decidió tomar el lenguaje GCL y agregar aquellas herramientas necesarias para ofrecer un lenguaje de programación básico. Tales herramientas incluyen entrada y salida de datos, declaración de variables dentro de bloques y declaración de funciones puras.

Para decidir la técnica de análisis sintáctico a utilizar, primero se consideró si el lenguaje podía ser reconocido por una gramática LL, ya que este tipo de gramáticas permite escribir un reconocedor recursivo descendiente eficiente. Por lo tanto, en primer lugar fue diseñada una gramática LR a la cual se le hicieron las transformaciones necesarias para obtener una gramática LL.

Posteriormente, se consideraron las herramientas ofrecidas por Haskell para implementar reconocedores LL y LR. La técnica LR fue rápidamente descartada, debido a que la herramienta diseñada en Haskell para tal fin ([Happy](#)), aunque brinda un rendimiento aceptable para la recuperación de errores, ofrece funcionalidades de un nivel muy básico, por lo que no resulta viable para tener una recuperación robusta en tan corto tiempo. Este apartado es de suma importancia, ya que una de las intenciones del lenguaje es simplificar su aprendizaje y, entre otros aspectos, eso requiere reportar errores con claridad y precisión contextual, para lo cual son necesarios mecanismos robustos y flexibles de recuperación de errores en el compilador.

Para escribir un reconocedor LL, la comunidad Haskell dispone de los módulos de [attoparsec](#) y [parsec](#). Ambas son librerías que proveen combinadores para la construcción de reconocedores recursivos descendentes, incluyendo *backtracking*. Luego de evaluar ambas librerías, se prescindió de [attoparsec](#), pues a pesar de ofrecer todos los combinadores necesarios para el reconocedor y la recuperación de errores, fue diseñada de forma tal que no

es posible combinarla con cómputos estructurados utilizando *monads*, necesarios en etapas posteriores del desarrollo. Por el contrario, `parsec` sí permite el uso de transformadores de *monads* por lo que fue la herramienta escogida para implementar el reconocedor sintáctico.

Una vez decidida la herramienta de análisis sintáctico, se inició el desarrollo de una calculadora para adquirir familiaridad con el uso de Haskell y la librería seleccionada (`parsec`). El desarrollo fue exitoso, obteniendo como resultado una calculadora funcional con operaciones de suma, multiplicación, resta, división y potencia. Además, la calculadora estaba en la capacidad de reportar y recuperarse de errores de sintaxis.

Posteriormente, se decidió separar el reconocimiento de los *tokens* de la calculadora del reconocimiento sintáctico para mejorar su diseño, por lo que se desarrolló un reconocedor lexicográfico. Para realizar esta tarea, primero se consideró incluir la posibilidad de añadir caracteres Unicode codificados en [UTF-8](#) necesarios para permitir el uso de las letras del idioma español y símbolos especiales, con el propósito de mejorar la legibilidad del código escrito en el lenguaje. Por último, fueron analizadas dos librerías para operar cadenas de caracteres en Haskell: `Data.ByteString` y `Data.Text`. Ambas están optimizadas para manejar cadenas de gran tamaño y ofrecen un rendimiento superior a la librería `Data.String`, la cual ofrece Haskell por defecto para tal fin. A pesar de esto, se prescindió la utilización de la primera debido a que no posee compatibilidad con los caracteres Unicode codificados en UTF-8, mientras que la segunda sí, por lo que fue la librería seleccionada. Además, la calculadora ofrecía recuperación de errores en caso de encontrar un *token* inesperado. La recuperación de errores de tipos no fue necesaria, puesto que todos los valores manejados por la calculadora eran del tipo entero.

Se empleó la técnica de recuperación de errores conocida como *Panic Mode* [21]. Cuando se encuentra un *token* inesperado durante el análisis sintáctico, se descartan *tokens* hasta encontrar alguno en el conjunto de resincronización. Ya que se trata de un reconocedor

LL, basta esperar hasta algún *token* en el conjunto *FOLLOW* de la regla gramatical. Esto es, aquellos símbolos que puedan aparecer en la entrada, después de haber terminado de reconocer la regla en cuestión.

## 4.2. Segunda Etapa

Esta etapa transcurrió durante el período entre Abril y Julio de 2015. Los objetivos para esta etapa incluyeron implantar el analizador sintáctico que construye el AST correspondiente al programa reconocido, agregar la tabla de símbolos y desarrollar la verificación de tipos.

El analizador lexicográfico [21] del ensayo de la calculadora fue refinado para reconocer todos los *tokens* que posee el lenguaje GraCieLa. Además, retorna un error cuando se encuentra un elemento lexicográfico que no pertenece a aquellos permitidos por el lenguaje.

Con respecto al analizador sintáctico [21], fueron incluidas las reglas suficientes para reconocer el lenguaje GraCieLa. En primer lugar, el analizador de la calculadora se amplió para reconocer aquellos operadores y valores pertenecientes al lenguaje no incluidos en la misma. Luego, se añadió el resto de las reglas necesarias para reconocer el lenguaje y el tipo de retorno fue cambiado para obtener un AST.

Uno de los conceptos estudiados en CI-4251 (Programación Funcional Avanzada) es el de Cómputo Mónadico como técnica para secuenciar y combinar cómputos. En particular, se estudió la construcción de cómputos que modifican estado (*Monad State*) y cómo combinarlos con otros *Monads* (*Monad State Transformer*). Esto sirve de apoyo para la tabla de símbolos, la construcción del AST y también en las etapas posteriores de compilación.

### 4.2.1. Tabla de Símbolos

El *Monad State* anteriormente mencionado facilitó la combinación del reconocedor sintáctico con una representación del estado para manejar la **Tabla de Símbolos**, que es

una estructura encargada de mantener la información necesaria (tipo, línea y columna, entre otros) de todos los símbolos (variables, funciones y procedimientos) del programa. Con los objetivos de asignar fácilmente el tipo de cualquier variable encontrada durante la compilación, reportar errores por el uso de variables no declaradas, el uso de funciones y procedimientos no definidos e intentos de modificar una constante. La tabla está implementada utilizando un árbol, en el cual cada nodo representa un alcance dentro del programa y un arco entre dos nodos representa la relación de alcance entre bloques de código del programa original. En la raíz del árbol se mantienen aquellos símbolos que son alcanzables desde cualquier punto del programa, como es el caso de las funciones y procedimientos creados por el programador.

Las restricciones del lenguaje permitieron un diseño de tabla de símbolos sencillo. En una primera instancia se utilizó un tipo de datos recursivo para modelar los arcos del árbol. No obstante, posteriormente fue considerado el uso de librería [Data.Tree](#) para mantener los arcos del árbol, dado que ofrece mayor eficiencia y simplicidad. Mientras tanto, para los nodos fue utilizada la librería [Data.Map.Strict](#), que tiene como función la representación de diccionarios, ya que era necesario mantener por cada símbolo perteneciente a un alcance toda la información necesaria para continuar con el proceso de compilación. Además, se utilizó la versión estricta de la librería para impedir la evaluación de expresiones de manera perezosa. Por lo tanto, el lenguaje no espera a que los datos insertados en la estructura se necesiten por otra operación para evaluarlos, por el contrario, son evaluados inmediatamente al ser ejecutada la función encargada de modificar la estructura. Esto es beneficioso, porque se traduce en un compilador que hace mejor uso de la memoria RAM asignada ya que las expresiones a evaluar no se mantienen almacenadas.

### 4.2.2. Árbol de Sintaxis Abstracta

Para la implementación del **AST** fue creado un tipo de datos recursivo, en el cual cada constructor representa un elemento sintáctico del lenguaje. El tipo de datos es recursivo ya que cada nodo puede tener apuntadores a uno o a varios nodos del mismo AST. Además, es polimórfico, ya que el mismo está parametrizado con un tipo genérico. En la etapa posterior de verificación de tipos, se apreció el beneficio de transformar la estructura en una polimórfica, ya que dependiendo del momento de la compilación en que se encuentre, permite guardar la información necesaria para ser aprovechada por el compilador.

Una vez terminado el reconocimiento sintáctico se entrega a la verificación de tipos un árbol con información parcial de los tipos de cada constructor del árbol. Específicamente, cada constructor del árbol incluye la información de tipos que pueda ser obtenida sin necesidad de explorar los sub-árboles correspondientes. Este grupo incluye a los valores constantes y las variables, las cuales pueden ser consultadas en la tabla de símbolos para obtener el tipo bajo el que fueron declaradas y reportar un error en caso de no ser encontradas. En caso de no poder decidir el tipo del constructor, se le asigna el tipo vacío para que esta información sea completada durante la verificación de tipos.

### 4.2.3. Verificación de Tipos

La función encargada de la verificación de tipos es recursiva y revisa el árbol completo, examinando cada construcción para verificar que los tipos de cada sub-construcción coincidan con aquellos definidos en la especificación del lenguaje. En caso que los tipos sean correctos, se asigna el tipo correspondiente al constructor. En caso contrario, se reporta un error de tipos y se le asigna un tipo especial para estos casos, llamado **error**. Por otro lado, si alguno de los sub-árboles posee el tipo error, el constructor que está siendo verificado tendrá el tipo error automáticamente, repitiendo este comportamiento hasta propagarlo a la raíz del árbol.

Para realizar la verificación de tipos, la función se apoya en el *Monad RWS*, el cual incluye los *Monads Reader, Writer, y State* en uno solo. Sin embargo, para este proyecto solo fueron necesarios los *Monads Reader y Writer*. El *Monad Writer* fue utilizado para mantener una lista con todos los errores encontrados durante la verificación de tipos, mientras que el *Monad Reader* se empleó para consultar la tabla de símbolos cada vez que fuera necesario. El *Monad State* no fue utilizado para ninguna función específica.

Para modelar los tipos del lenguaje en Haskell, se implantó un tipo de datos recursivo con constructores que representan a cada uno de ellos y almacenan la información necesaria. Es recursivo porque las funciones necesitan el tipo de retorno y los arreglos su tipo almacenado. Además, fue definido el operador de igualdad para representar las reglas de verificación de tipos del lenguaje. Es importante recalcar que el proceso de verificación de tipos no modifica la estructura del árbol correspondiente al programa, solo asigna al constructor el tipo correspondiente especificado en la documentación del lenguaje.

#### 4.2.4. Cuantificadores

Los cuantificadores se componen de una variable de cuantificación, un operador binario (asociativo y commutativo), un predicado llamado el **rango** del cuantificador, y una expresión definida como el **cuerpo** del cuantificador. El cuerpo del cuantificador se aplica sobre cada elemento perteneciente a la secuencia de valores definida por el rango, para obtener el resultado a partir de aplicaciones sucesivas del operador sobre el rango modificado. Nos permitimos la libertad de extender la noción clásica de cuantificación, que incluye únicamente universales y existenciales, con funciones de agregación generales que cumplan con las características antes mencionadas.

Primero, se implementó un procedimiento recursivo para revisar que la variable de cuantificación perteneciera a cada sub-expresión correspondiente al rango (sólo se permite una

variable de cuantificación), en caso que esto no fuera cierto, se emite un error a tiempo de compilación. Posteriormente, es necesario transformar el predicado definido por el rango en un conjunto de valores ordenados sobre los que se debe ejecutar el cuerpo del cuantificador. Primero, se incluyó una limitación en el lenguaje para que los tipos permitidos en un cuantificador solo sean algunos que sabemos son contables, es decir, el tipo `int`, `char`, y `bool` los cuales representan a los enteros, caracteres y valores lógicos respectivamente.

Es importante señalar que el rango de un cuantificador no necesariamente se puede calcular a tiempo de compilación, ya que se permiten expresiones cuyo valor no es conocido durante esta etapa, como lo son variables y llamadas a funciones. En estos casos, el cálculo del rango se deja a tiempo de ejecución del programa. Aquellos rangos que sí son calculables durante la compilación están conformados por constantes (ya sean literales o por nombre), y para su representación y cálculo se usó la librería `Data.Range`, que permite representar rangos de manera sencilla. Ofrece operaciones entre rangos, tales como la intersección y unión, que facilitan la obtención del rango final.

### 4.3. Tercera Etapa

Esta etapa se desarrolló durante el período de Julio a Noviembre del año 2015, con los objetivos de generar código intermedio LLVM y código de máquina correspondientes a la máquina utilizada para compilar.

LLVM provee una representación intermedia de máquina virtual, útil durante el desarrollo y depuración de herramientas. Para realizar la conexión entre el compilador de GraCieLa y las librerías del LLVM, surgieron dos modulos de Haskell: `LLVM.General.Pure` y `LLVM.General`. El primero se encarga de ofrecer una estructura de datos recursiva en Haskell, cuyos constructores permiten representar a todas las construcciones provistas por el lenguaje intermedio LLVM. El segundo módulo se encarga de transformar la estructura anteriormente descrita en

código intermedio LLVM (el cual puede ser interpretado usando la herramienta `lli`, accesible desde la línea de comandos) o de máquina, dependiendo de las necesidades del programador.

Para el proceso de generación de código fue utilizado un *Monad State* que almacena una lista de definiciones, la cual es necesaria para los programas escritos en código intermedio LLVM, ya que cada módulo cuenta con, al menos, un nombre y una lista de definiciones. De las definiciones provistas por el LLVM, únicamente son relevantes para este Proyecto de Grado las definiciones globales, utilizadas para representar las funciones, procedimientos y variables globales generadas para uso interno del compilador.

Las variables y las funciones son las únicas definiciones globales empleadas dentro del compilador. Las funciones de LLVM son utilizadas para representar las funciones, los procedimientos y el código principal del programa escrito en GraCieLa. En cualquiera de los casos se necesita una lista de bloques básicos que también es actualizada mediante modificaciones al *Monad State* utilizado para esta etapa del desarrollo. Mientras tanto, las variables se emplean para mantener un apuntador a los archivos usados para realizar la entrada de datos.

Cuando la función encargada de la transformación del AST en código intermedio termina de transformar una función o procedimiento, la lista de bloques básicos se guarda en la definición LLVM incluyendo el nombre, los parámetros y tipo en caso de las funciones (para los procedimientos el tipo es `void` provisto por el lenguaje intermedio LLVM). Luego, la lista es sustituida por una nueva lista vacía para ser llenada por la siguiente función o procedimiento a ser transformado, mientras que la definición recién creada se guarda en la lista de definiciones.

Un bloque básico LLVM es definido por un nombre, una lista de instrucciones, y un terminador.

El nombre es un identificador único, por lo que se optó por utilizar números enteros contando desde el cero. Así, cada bloque tiene un número que le identifica únicamente. Los nombres son necesarios, ya que las instrucciones de cambio de flujo de programa se valen de

ellos para identificar el bloque básico a saltar.

La lista de instrucciones se genera a medida que las instrucciones del lenguaje GraCieLa son transformadas a código intermedio. Las instrucciones en LLVM pueden ser de dos tipos: con nombre o sin nombre. Para este lenguaje solo son tomadas en cuenta instrucciones con nombre, debido a que en muchos casos el nombre es necesario para utilizarlo como dirección de destino de un cálculo. En el caso que un nombre se asigne a una instrucción para la cual no es necesario, la instrucción en cuestión se encargará de ignorarlo. Por ejemplo, una instrucción que necesita nombre es la encargada de sumar el contenido de dos registros, que emplea el nombre dado para representar el resultado de la suma. Por otro lado, una instrucción que no necesita nombre es la encargada de realizar un salto hacia otro fragmento de código intermedio.

Los terminadores son la última instrucción ejecutada antes de cerrar un bloque básico. Son especiales porque implican una ruptura en el hilo de ejecución del programa, ya que la siguiente instrucción a un terminador no necesariamente es la siguiente instrucción a ejecutar. El lenguaje de código intermedio LLVM ofrece varios terminadores, pero para el lenguaje GraCiela solo son relevantes los correspondientes a salto incondicional, salto condicional, retorno, y **switch**.

El terminador de salto incondicional se utiliza en aquellos casos en los que es necesario ejecutar un salto en el código sin evaluar una expresión lógica. Por ejemplo, el compilador genera este tipo de salto para transformar la instrucción de repetición en el lenguaje GraCieLa, ya que después de ejecutada la guardia es necesario volver a la primera para repetir la evaluación de las mismas.

El terminador de salto condicional se utiliza para transformar las guardias correspondientes a las instrucciones de repetición y condición del lenguaje GraCieLa, ya que requieren la evaluación de una condición lógica para decidir la dirección de salto a tomar.

En el LLVM los saltos se hacen desde bloques básicos hacia otros bloques básicos dentro de la misma definición. Específicamente, es necesario generar saltos para las instrucciones condicional y repetición del lenguaje GraCieLa. Para ambos casos, es necesaria la generación de al menos el doble de saltos que el número de guardias que poseen, ya que cada guardia requiere de al menos dos bloques básicos. Por un lado, necesita uno para el código encargado de evaluar la guardia. Por otro lado, necesita de al menos uno para el código ejecutado en caso que la guardia sea cierta, ya que la instrucción correspondiente puede generar otros bloques básicos durante su transformación.

Por otro lado, el terminador `switch` es necesario para aquellos casos en los que es requerida la evaluación de varias condiciones lógicas, con la respectiva ejecución del código correspondiente a alguna evaluación con valor `true`. No es necesario conocer cual código fue ejecutado, pero sí el valor producto de dicha ejecución. Específicamente, esta instrucción es usada para la transformación de las funciones del lenguaje GraCieLa, las cuales pueden ser definidas usando un condicional cuyas guardias están asociadas con expresiones. Por lo tanto, cualquier guardia seleccionada para ejecución tendrá un resultado asociado. Sin embargo, la instrucción `switch` no es utilizada para la instrucciones de selección o repetición del lenguaje GraCieLa dado que no retornan ningún valor.

Por último, el terminador de retorno implica un cambio en el flujo de ejecución, debido a que se alcanzó el fin de una función. Puede incluir un valor en el caso que sea necesario. El compilador lo utiliza para terminar la ejecución tanto de funciones como de procedimientos en el lenguaje. Para el caso de las funciones el valor de retorno es aquel evaluado por la función, mientras que en el de los procedimientos es el valor `void`, que representa la falta de valor.

### 4.3.1. Arreglos

Para la representación de arreglos se consideró el uso del tipo `array` o `vector` provistos por el LLVM pero fue preferida la reserva de memoria contigua usando la instrucción `alloca` de LLVM. Por cada declaración de un arreglo de dimensión  $n$ , en el programa a compilar se reserva la siguiente cantidad de *bytes*:

$$tamarr_i = cantelem_i \times tamarr_{i+1}, \quad 1 \leq i \leq n$$

$$tamarr_n = cantelem_n \times tamtipo$$

Donde  $tamarr_i$  es el tamaño de la dimensión  $i$  del arreglo,  $cantelem_i$  es la cantidad de elementos en la dimensión  $i$ , y  $tamtipo$  es el tamaño del tipo contenido en la última dimensión del arreglo en *bytes*. Nótese que la última dimensión siempre contendrá algún tipo básico del lenguaje debido a que el único tipo compuesto de este es el arreglo. Esta fórmula garantiza la cantidad suficiente de espacio para almacenar la totalidad de los elementos del arreglo.

Para acceder a una posición del arreglo es necesario resolver la siguiente ecuación:

$$\text{dirbase} + \sum_{i=1}^n cantdesp_i \times tamarr_i$$

Donde  $dirbase$  es la dirección de la posición del primer elemento del arreglo,  $cantdesp_i$  es la posición del elemento en el sub-arreglo  $i$  al que se desea acceder y  $tamarr_i$  es el tamaño de los elementos alocados en el sub-arreglo  $i$ . Es importante aclarar que la fórmula está diseñada para albergar el caso de arreglos de más de una dimensión, y que cada sub-arreglo  $i$  representa cada una de las dimensiones que podría tener un arreglo. Esta fórmula corresponde al método de organización por filas de un arreglo, ya que cada elemento de los sub-arreglos  $i$  están contiguos en memoria.

### 4.3.2. Entrada y Salida

Para la entrada y salida de datos, se escribió una librería en el lenguaje C con el objetivo de efectuar las llamadas al sistema necesarias para este fin. La librería debe ser compilada como compartida y almacenada en una carpeta específica del sistema para poder ser encontrada al momento de ejecutar un programa. En el diseño del lenguaje, se decidió que no fuera necesario que el programador se encargara de abrir y cerrar los archivos a los que deseaba aplicar operaciones de lectura, por lo que el programa debe hacerlo internamente.

Por lo tanto, durante la etapa de reconocimiento sintáctico, todos los nombres de archivos se guardan en el estado usado para esa etapa de compilación. Esta información se transfiere a la función encargada de generar código intermedio, la cual declara una variable global por cada archivo utilizado que servirá para almacenar un apuntador al archivo. Al iniciar la transformación del código principal del programa a código intermedio, el compilador genera código para llamar al procedimiento perteneciente a la librería externa al programa encargado de realizar la apertura del archivo, el cual retorna un apuntador al mismo que será almacenado en la variable global correspondiente.

Por cada lectura efectuada a un archivo, se pasa el apuntador correspondiente al procedimiento destinado a tal fin, se realiza la lectura correspondiente y el valor leído es retornado junto al apuntador actualizado en la posición en la que se realizará la próxima lectura. Después de transformar todo el código perteneciente al programa principal, se genera código para cerrar todos los archivos abiertos.

# Capítulo 5

## Resultados

Como resultado del presente Proyecto de Grado, se diseñó el lenguaje GraCieLa, basado en GCL y se implementó su respectivo [compilador](#) gacela. En general, fueron tomados de GCL el conjunto de [instrucciones](#), y la necesidad de escribir la precondición y la postcondición en la definición de procedimientos, además de los tipos básicos y la sintaxis para la declaración de variables.

Además, se incorporaron facilidades para la entrada y salida, posiblemente usando archivos, además de construcciones para expresar funciones sin efectos de borde.

### 5.1. Diseño del lenguaje y detalles del compilador

El enfoque del lenguaje GraCieLa siempre se centró en apegarse a las especificaciones principales de GCL. Todas las extensiones provistas fueron incorporadas de la manera más simple y consistente posible. En ambos casos, recordando que los usuarios del lenguaje serían estudiantes de los primeros cursos de algorítmica de la USB.

Todo programa escrito en el lenguaje GraCieLa debe contar con la siguiente estructura:

```
program <nombre del programa>
```

```

begin
    <lista de definiciones>
    <instrucción de bloque de código>
end

```

La lista de definiciones solo puede contar con funciones y procedimientos que podrán ser utilizados en el bloque de código principal, el cual está compuesto por una lista de declaraciones de variables y una secuencia de acciones a ejecutar por el programa.

### 5.1.1. Tipos

En el lenguaje se proveen cuatro tipos de datos básicos y un tipo compuesto. Los tipos de datos básicos son: `boolean`, `int`, `double`, y `char`. El tipo compuesto es: `array`.

1. `boolean`: Modela los valores lógicos pertenecientes al Álgebra de Boole. En consecuencia, sus valores posibles son `true` y `false`. Se implantan como 1 bit de memoria.
2. `int`: Modela a los números enteros que pueden tomar valores tanto positivos como negativos. Sus valores posibles son comprendidos dentro del rango  $[-2^{31} \dots 2^{31} - 1]$ . Cada uno se implanta como 32 bits de memoria, usando complemento a dos.
3. `double`: Modela los números en punto flotante según IEEE-754 [3]. Sus valores posibles corresponden al rango de números comprendido en  $[-10^{308,25} \dots 10^{308,25}]$ . Se implantan usando 64 bits de memoria <sup>1</sup>.
4. `char`: Modela los caracteres ASCII imprimibles. Ocupan 8 bits de memoria.
5. `array`: El lenguaje permite la declaración y utilización de arreglos con base cero los cuales son colecciones continuas de elementos accesibles, cada uno identificado con un

---

<sup>1</sup><http://llvm.org/docs/LangRef.html#floating-point-types>

número entero. En la declaración, el programador debe proveer el tipo contenido en el arreglo y su tamaño. Dicho tamaño solo puede ser un número o una variable de tipo entero, por lo que no necesariamente se conoce su tamaño a tiempo de compilación. Por lo tanto, son elaborados a tiempo de ejecución por el programa. Su ocupación en memoria corresponde a la cantidad de *bytes* ocupados por el tipo del arreglo multiplicado por su tamaño.

La declaración de arreglos se realiza utilizando la palabra reservada **array** como en el siguiente ejemplo:

```
var arr : array [10] of array [x] of int;
```

Como se puede apreciar, la variable **arr** es del tipo **array** cuyo tamaño es diez y su tipo contenido es **array** de tamaño **x**. Es decir, es un arreglo de tipo entero con diez celdas y cada una de estas es a su vez un arreglo de tamaño igual al valor almacenado en la variable **x**.

Por otra parte, el lenguaje propuesto en este proyecto es de verificación de tipos fuerte. Por lo tanto, no está permitido ejecutar operaciones que violen el sistema de tipos del lenguaje y toda variable dentro del programa posee un tipo asociado a tiempo de compilación. La única manera de modificar el tipo de una variable, es a través de la realización de una conversión explícita utilizando una de las funciones provistas por el lenguaje para este propósito. Por otro lado, si los tipos usados en la llamada a una operación definida en el lenguaje no coinciden con los definidos para dicha operación el compilador emite un error a tiempo de compilación. Además, se permite declarar más de una variable en una sola declaración.

### 5.1.2. Operadores

En el lenguaje se definen tres tipos de operadores: aritméticos, relacionales, y lógicos. Como el lenguaje es de tipos fuertes las operaciones solo podrán aplicarse a operandos para los cuales la operación está definida, en caso contrario, ocurrirá un error a tiempo de compilación.

Los operadores aritméticos son aquellos definidos para realizar las operaciones aritméticas más comunes. En caso que el resultado de la operación no pueda ser representado, es decir, está fuera del rango definido para el tipo, será emitido un error a tiempo de ejecución y finalizará la ejecución del programa. Están definidos para los tipos `int` y `double`, además, ambos operandos y el resultado de la operación son del mismo tipo.

Los operadores relationales son aquellas operaciones matemáticas para determinar orden entre los valores pertenecientes a un tipo. Están definidos para los tipos `int` y `double` y ambos operandos deben ser del mismo tipo. El valor de retorno es del tipo `boolean`.

Los operadores lógicos son aquellos definidos en el álgebra de Boole para realizar operaciones sobre valores del tipo `boolean`. En el cuadro 5.1 se muestran todos los operadores con su respectiva precedencia.

Es importante señalar que la evaluación de todas las operaciones en el mismo nivel de precedencia se hacen desde el último operando hasta el primero, es decir, de derecha a izquierda. La tabla con todos los operadores y sus precedencias se encuentra en el [apéndice C](#).

### 5.1.3. Aserciones

El lenguaje GraCieLa posee cinco formas de escribir condiciones sobre el estado de las variables, las cuales, dependiendo de la definición de la aserción, pueden interrumpir la ejecución del programa. Estas cinco formas se presentan a continuación:

1. **Precondición:** Es la condición que deben cumplir los procedimientos definidos por

el programador antes de ejecutar su bloque de código. En caso que el estado de las variables no cumpla la precondición, se emite una advertencia al programador pero no termina la ejecución del programa. Las precondiciones se representan con el *token* `pre`.

2. **Postcondición:** Es aquel predicado que debe cumplir el estado de las variables después de terminar la ejecución del bloque de código perteneciente a un procedimiento. En caso que la precondición sea válida pero la postcondición no, será emitido un mensaje de error al programador y terminará la ejecución del programa. En caso que la precondición no sea válida, la postcondición no será verificada. Las postcondiciones se representan con el *token* `post`.
3. **Instrucción con aserción:** En cualquier momento en la ejecución del programa, el programador puede escribir un predicado para acompañar a una instrucción. En caso de ser una aserción válida la instrucción será ejecutada y el programa continuará su ejecución. En caso contrario, se emite un mensaje de error al programador y terminará la ejecución del programa. La aserción de una instrucción se representan con el *token* `a`.
4. **Invariante:** Cada instrucción de repetición debe contar con un invariante que debe cumplirse al inicio de cada iteración (incluyendo el caso en el que ninguna guardia se cumple). En caso que se viole la condición del invariante se emite un mensaje de error y el programa termina su ejecución. Los invariantes se representan con el *token* `inv`.
5. **Función de cota:** De igual forma que el invariante, cada instrucción de repetición debe contar con una función de cota, que es una expresión del tipo entero encargada de garantizar la terminación del ciclo. Por cada una de las iteraciones se verificará que el valor sea menor al valor obtenido en la iteración anterior y que además sea mayor o igual a cero. En caso de que esta condición no se cumpla se dará un mensaje de

error al programador y terminará la ejecución del programa. Las funciones de cota se representan con el *token* bound.

#### 5.1.4. Instrucciones

EL lenguaje GraCieLa cuenta con un conjunto de instrucciones basadas en las acciones del lenguaje del GCL. Por un lado, todas las instrucciones se separan utilizando el *token* ';' . Por otro lado, en caso de ser una única instrucción o la ultima de ellas no posee el *token* ';' . Las instrucciones se describen como sigue:

##### 1. Skip

La instrucción skip no realiza acción alguna y es útil para expresar aquellos casos en los que el estado de las variables es correcto y la ejecución del programa puede continuar sin problemas.

##### 2. Abort

La instrucción abort se ocupa de la terminación inmediata de la ejecución del programa. Puede ser utilizada por el programador cuando el estado de las variables no permite la continuación del algoritmo y la ejecución debe terminar. Ante ciertos eventos en el programa, esta instrucción se ejecuta implícitamente, por lo que ofrecerla al programador no altera el comportamiento del compilador. Es posible utilizarla de manera explícita para que el programador tenga la posibilidad de expresar programas en los cuales todos los caminos de cómputo estén claramente definidos.

##### 3. Asignación

La instrucción de asignación es la encargada de modificar directamente el estado actual de la ejecución del programa. No está permitida la asignación de una expresión cuyo tipo no coincida con el tipo de la variable, por lo que será emitido un error a tiempo

de compilación. No está permitida la asignación de un valor a una variable cuyos tipos no coincidan y en ese caso será emitido un error a tiempo de compilación.

Además, es permitida la asignación de más de una variable en la misma instrucción. Por último, también está permitida la asignación a una posición de un arreglo, en cuyo caso debe especificarse la posición a modificar entre corchetes.

A continuación se muestra un ejemplo de asignación válida:

```
x, y, arreglo[6] := 5, 10, 25;
```

#### 4. Selección

La instrucción de selección está compuesta por una lista de **Instrucciones con Guardia**, los cuales son condiciones (expresiones lógicas) sobre el estado actual de las variables, cada una de ellas acompañada por una instrucción. De esta lista será elegido una instrucción con guardia cuya condición sobre el estado de las variables sea posible para ejecutar la instrucción correspondiente.

Una selección válida es la siguiente:

```
if x < 0 -> skip
[] x == 0 -> abort
[] x > 0 -> x := x + 1
fi
```

Es importante recalcar que el *token* ' [] ' es usado como separador de las instrucciones guardadas. Por otra parte, a diferencia del lenguaje GCL en el que al haber más de una condición valida se escoge una instrucción guardada de forma no determinista, en GraCieLa será ejecutada la primera en el orden sintáctico. En caso que ninguna

sea válida la ejecución del programa será abortada emitiendo un mensaje de error al programador.

## 5. Repetición

De forma análoga a la instrucción selección, la repetición se encarga de elegir una instrucción guardada, cuya condición sobre el estado de las variables es posible, de una lista de instrucciones guardadas. Luego, ejecuta la instrucción asociada y repite el proceso descrito anteriormente hasta que ninguna guardia se cumpla, en cuyo caso la instrucción **skip** es ejecutada. Además, debe ser provisto un invariante y una función de cota para garantizar la validez y culminación de la instrucción.

Un ejemplo de una repetición válida es la siguiente:

```
{inv    i < x}
{bound x - 1}
do i < 25 -> skip
[] i < 25 -> i := i + 1
[] i == 50 -> x, i := x * 2, i + 1
od
```

En caso de que haya más de una condición válida sobre el estado de las variables será ejecutada la primera en orden sintáctico. Esta decisión se tomó para evitar la verificación del resto de las condiciones cuando ya existe una válida y así optimizar el rendimiento del programa. La misma justificación aplica para la instrucción de selección.

## 6. Llamada a procedimiento

Esta instrucción implica la ejecución del conjunto de instrucciones definido por el procedimiento a llamar, bajo el estado de las variables definidas por los parámetros y

variables locales al [procedimiento](#). En vista de que el lenguaje es de verificación de tipos fuerte, los tipos de las variables usadas para la llamada deben coincidir con aquellos en la definición de los parámetros del procedimiento. En caso contrario, será emitido un error a tiempo de compilación.

El siguiente ejemplo ilustra una llamada a procedimiento sintácticamente válida:

```
sumar(x, y)
```

El *token* `,` se emplea como separador de parámetros. Además, solamente pueden utilizarse expresiones compuestas en el caso que el parámetro correspondiente sea definido como `in`, estos modos de pasaje de parámetros serán explicados más adelante. Para el resto de los parámetros solo pueden usarse variables previamente declaradas, ya que es necesaria su dirección de memoria para almacenar el valor de retorno.

## 7. Escritura

Se definen dos instrucciones para escribir por [salida estándar](#): `write` y `writeln`. La primera recibe como parámetro una expresión de cualquier tipo o un valor literal que también puede ser una cadena de caracteres y luego muestra su valor correspondiente por salida estándar. La segunda se comporta de la misma manera adicionando un salto de linea después de escribir por salida estándar.

Un ejemplo válido de ambas instrucciones es el siguiente:

```
write("Este ejemplo incluye una cadena de caracteres")
```

```
writeln(x)
```

Es importante recalcar que una cadena de caracteres es una lista de caracteres que incluye el carácter " al principio y al final de la misma. Únicamente pueden ser utilizados para la escritura, mediante las instrucciones `write` y `writeln`.

## 8. Bloque

La instrucción de bloque sirve para definir nuevos alcances. Está compuesta por una lista opcional de declaraciones y por una lista de instrucciones. Ambas listas usan el carácter ';' como separador.

Este ejemplo contiene un bloque con declaración de variables y otro sin declaración:

```
| [
    var    x : int;
    const y := 45 : int;

    write("Hello world!");
    write(x)
]
|
```

### 5.1.5. Funciones

Las [funciones](#) no tienen efectos de borde, por lo tanto, no pueden modificar el estado definido por los parámetros de entrada. Sólo es posible emplear expresiones del mismo tipo de retorno especificado en la declaración de la función.

Por ejemplo, el siguiente fragmento de código pertenece a una definición de función válida:

```
func foo : (bar : int, qux : int, baz : int) -> int
begin
```

```
(qux * baz) + bar
end
```

Además, en las funciones es posible escribir un selector, utilizando la misma sintaxis de la instrucción equivalente. La diferencia entre ambas radica en los comandos guardados, pues mientras que en la instrucción cada uno de estos contiene una instrucción, en las funciones incluyen una expresión. Ya que el lenguaje es de verificación de tipos fuerte, cada una de las expresiones pertenecientes a los comandos guardados debe ser del mismo tipo de la función.

Este ejemplo contiene una función con un selector de expresiones:

```
func fibonacci : (n: int) -> int
begin
    if (n == 0) -> 0
    [] (n == 1) -> 1
    [] (n >= 2) -> fibonacci(n-1) + fibonacci(n-2)
    fi
end
```

Debido a que las funciones son puras los parámetros no pueden ser alterados y, por lo tanto, son considerados de tipo lectura.

### 5.1.6. Procedimientos

Los procedimientos sí cuentan con la posibilidad de modificar el estado definido por las variables declaradas y los parámetros usados en su definición. Además, requieren de una precondición y una postcondición, las cuales definen condiciones sobre el estado de las variables al comenzar y finalizar la ejecución de las instrucciones pertenecientes al procedimiento.

El siguiente ejemplo, es un fragmento de código pertenece a una definición de un procedimiento válido:

```

proc suma : (in bar : int, in qux : int, inout baz : int)
begin
{pre bar > 0 /\ qux <= 0 /\ baz > 0 }
| [
    baz := bar + qux + baz
] |
{post baz >= bar + qux + baz }
end

```

Los parámetros pueden tener distinto comportamiento dependiendo de la forma en que fueron declarados. En total, pueden declararse cuatro tipos de parámetros distintos:

1. **in**: El valor con el que cuenta la variable correspondiente al momento de la llamada es copiado al parámetro, el cual puede ser modificado durante la ejecución del procedimiento sin afectar la variable usada para la llamada.
2. **out**: Son parámetros cuyo valor al momento de la llamada del procedimiento no es tomado en cuenta, por lo que solo importa la dirección de memoria de la variable asociada a la llamada. Son inicializados antes de ejecutar el cuerpo del procedimiento con el valor por defecto correspondiente al tipo del parámetro. Por ultimo, su último valor asociado será copiado a la variable de llamada correspondiente.
3. **inout**: Poseen el valor de la variable de llamada correspondiente al inicio del procedimiento y copian su último valor asociado al terminar la ejecución del mismo.
4. **ref**: Este tipo de parámetros es pasado por referencia, por lo tanto, todo valor que se le asigne dentro del procedimiento también afectará inmediatamente al valor de la variable que fue utilizada en la llamada del procedimiento.

Es importante destacar que los arreglos únicamente pueden ser pasados como parámetro `ref`, es decir, el compilador emitirá un error a tiempo de compilación cuando se intente pasar un arreglo por los métodos `in`, `inout` o `out`. Además, el tamaño del arreglo debe ser uno de los parámetros anteriores al arreglo en la definición del procedimiento o función. Por otro lado, los parámetros de tipo `out` e `inout` solo permiten pasar variables por lo que no está permitido el uso de otras expresiones en la llamada; es decir, solo pueden usarse direcciones a las cuales sea posible reemplazar su valor.

### 5.1.7. Variables y Constantes

Las variables son alias de direcciones de memoria sobre los cuales es posible almacenar, modificar, y consultar valores. Las constantes poseen el mismo comportamiento con la diferencia que su valor es asignado a tiempo de compilación y no puede ser modificado. Si mediante una asignación se intenta modificar el valor de una constante se producirá un error a tiempo de compilación. Por otro lado, la declaración puede estar acompañada de una asignación inicial, esto es obligatorio en el caso que sea declarada una constante, no así en el caso de las variables. Las palabras reservadas `var` y `const` son usadas para declarar variables y constantes, respectivamente.

El compilador graciela no permite la utilización de variables globales para evitar que los programadores novatos desarrolleen vicios al hacer mal uso de ellas, debido a que es muy común que durante su aprendizaje utilicen únicamente variables globales por la facilidad y sencillez que les brinda. Sin embargo, es un mal hábito de programación que no les permite comprender el funcionamiento de alcance y anidamiento de las variables. Además, genera confusión en el programador por lo que solo se permite la definición de variables locales en los procedimientos y en los bloques de código.

Los parámetros de los procedimientos se encuentran en el mismo alcance de las variables

locales. Cada bloque de código define un nuevo alcance, por lo que todas las variables definidas solo serán visibles por aquellos bloques e instrucciones escritos en su interior. Por lo tanto, no se podrá acceder a las variables definidas en los bloques internos.

El siguiente ejemplo ilustra el alcance dentro de bloques:

```
[ [ // Primer bloque de código
    var foo, bar : int;
    var qux : boolean;

    [ [ // Segundo bloque de código
        const baz := 79.7, bee := 5.5 : double;
        var aux := 49 : int;

        foo := aux;
    ] ]
]
```

Las variables declaradas dentro del primer bloque tienen alcance en el segundo; por lo tanto, pueden ser utilizadas. No obstante, las variables del segundo bloque no se pueden utilizar en el primero. En el supuesto caso en el que existiese un tercer bloque de código dentro del segundo, sí se podrían utilizar las del segundo (y así sucesivamente).

Toda variable es inicializada con un valor por defecto dependiendo de su tipo. Los valores por defecto son:

1. **int**: El entero 0.
2. **char**: El símbolo ASCII 0, es decir, el carácter nulo.
3. **bool**: El valor lógico **false**.

4. **double**: El valor punto flotante 0,0.

En el caso de los arreglos al momento de compilación graciela reserva el espacio en memoria necesario para almacenar todos los valores del arreglo, pero no se hacen modificaciones sobre él ni se verifica que los contenidos del arreglo estén inicializados por el programador antes de usarlo, por lo que queda a riesgo del programador utilizar un arreglo sin inicializar.

La redefinición de variables no está permitida en graciela. Si una variable fue declarada previamente ya sea en el mismo alcance o en un alcance superior, no se podrá declarar nuevamente otra con el mismo nombre, sin importar su tipo. Esta decisión fue tomada para minimizar la cantidad de errores que pueda cometer un programador inexperto en el proceso de aprendizaje, puesto que no cuentan con la práctica suficiente sobre el alcance de las variables y creen estar usando una versión de una variable cuando en realidad están utilizando su redefinición.

El siguiente programa ejemplifica una redefinición de variables.

```
| [
```

```
var foo, bar : int;
var qux : boolean;
```

```
| [
```

```
const baz := 79.7, qux := 5.5 : double;
var bar := 49 : int;
```

```
foo := bar;
```

```
]|
```

```
||
```

En el ejemplo se declaró la variable `bar` en ambos bloques de código por lo que el código daría un error a tiempo de compilación.

### 5.1.8. Lectura de variables

Todas las lecturas de datos, tanto de un archivo como a través de `entrada estándar`, únicamente se pueden realizar antes de escribir la precondición y después de la declaración de variables de un procedimiento. No se provee ninguna otra forma de entrada de datos. Para hacer una lectura se emplea la palabra reservada `read` seguida de una lista de variables entre paréntesis, en cuyo caso la lectura será por entrada estándar. Para hacer la lectura de un archivo el programador debe usar la palabra reservada `with` acompañada por una cadena de caracteres representando el nombre del archivo, posterior a la lista de variables a ser utilizadas.

Estos métodos de entrada de datos no están presentes en GCL, sin embargo, fueron incluidos en el diseño de GraCieLa para poder ofrecerlo en un contexto didáctico. Su diseño persiguió el objetivo de simplificar este proceso para programadores novatos, además de asegurar que todos los valores leídos puedan ser verificados en la precondición del procedimiento. Por lo tanto, el no-determinismo de hacer lectura de variables se mantiene controlado. Las variables utilizadas pueden estar declaradas como variables locales del procedimiento o pueden ser parámetros de entrada. Por otro lado, es necesario resaltar que el comportamiento del programa en caso de intentar leer e interpretar una línea de texto que no coincide con el tipo de la variable usada para la lectura, es indefinido y se deja a riesgo del programador.

El siguiente ejemplo ilustra cómo se da la lectura de variables:

```
proc foo : (in bar : boolean, out qux : int)
begin
    var text : char;
```

```

var i      : int;
var a, b : double;

read(a, text, qux) with "nombre_archivo.txt";
{pre a > 0.5 /\ b < 1.2 /\ i >= 0 }

...
end

```

### 5.1.9. UTF-8

El compilador gacela permite que el programador además de usar el formato de codificación de caracteres ASCII, pueda utilizar los caracteres Unicode codificados en UTF-8 para poder escribir los símbolos matemáticos, lógicos, y los operadores de cuantificación, permitiendo que el código sea más legible y elegante.

A continuación se presenta un cuantificador en formato ASCII y en formato UTF-8:

```

ASCII: (% sigma k : int | 0 <= k /\ k < 10 | 2*k %)
UTF-8: (% Σ k : int | 0 ≤ k ∧ k < 10 | 2×k %)

```

### 5.1.10. Cuantificadores

Todo cuantificador posee exclusivamente una variable cuantificada que necesariamente debe ser de tipo `int`, `char`, o `boolean`. La variable debe ocurrir en el rango del cuantificador, es decir, debe estar presente en todas las sub-expresiones pertenecientes al rango. En caso contrario, un error a tiempo de compilación será emitido. En el caso que se desee cuantificar sobre más de una variable, es posible escribir un cuantificador dentro del cuerpo de otro

externo. Dejar vacía la expresión dentro del rango se traduce en un rango desde el mínimo hasta el máximo valor perteneciente al tipo de la variable de cuantificación.

Además de los cuantificadores pertenecientes a la lógica clásica (universal y existencial), el lenguaje ofrece los cuantificadores aritméticos sumatoria, producto, máximo y mínimo a conveniencia del programador. A continuación, se detallan los distintos tipos de cuantificadores presentes en el lenguaje:

1. **forall**: Corresponde al cuantificador universal y su función consiste en verificar que el cuerpo del cuantificador se cumple con todos los elementos que componen el rango. Si el rango es vacío, se dará una advertencia al programador y el resultado será **true**.
2. **exist**: El cuantificador existencial es utilizado para verificar que al menos uno los elementos que componen el rango cumplen el cuerpo del cuantificador. Si el rango es vacío, se dará una advertencia al programador y el resultado será **false**.
3. **sigma**: Se encarga de sumar todos los valores dados al evaluar el cuerpo del cuantificador en cada uno de los elementos pertenecientes al rango. El cuerpo del cuantificador debe ser del tipo **int** o **double**. Si el rango es vacío, el resultado será 0.
4. **pi**: Mantiene un funcionamiento análogo al del cuantificador **sigma**, pero el operador es el de multiplicación. Si el rango es vacío, el resultado será 1.
5. **max**: La función del máximo es obtener el mayor elemento entre aquellos obtenidos después de la evaluación del cuerpo del cuantificador, cuyo tipo debe ser **int** o **double**. Si el rango es vacío, se emitirá un error a tiempo de ejecución.
6. **min**: Se comporta de forma análoga a la del cuantificador **max**, pero en lugar del máximo se obtiene el valor mínimo. Si el rango es vacío, se emitirá un error a tiempo de ejecución.

Todo cuantificador comienza con el *token* reservado (% y termina con %), como se puede apreciar en la siguiente ejemplo de su sintaxis:

```
(%<Tipo de Cuantificador> <variable> : <Tipo> | <Rango> | <Expresión> %)
```

### 5.1.11. Funciones Predefinidas

Son funciones definidas por defecto en el lenguaje para facilitar el trabajo del programador y están compuestas por las conversiones entre tipos y las operaciones matemáticas de valor absoluto y raíz cuadrada. Las funciones predefinidas del lenguaje se enumeran en el [apéndice E](#).

### 5.1.12. Errores de compilación

EL compilador graciela desarrollado en este proyecto de grado es de verificación estática por lo tanto mostrará al programador todos los errores posibles a tiempo de compilación y se dejarán a tiempo de ejecución aquellas verificaciones que por sus características no sea posible verificar a tiempo de compilación.

Existen tres clases de errores que el compilador puede reportar, estos son: lexicográficos, sintácticos, y de contexto, que incluye a los errores de tipos.

1. **Los errores lexicográficos:** Son aquellos pertenecientes a la primera etapa de compilación y se dan cuando el programa a compilar contiene un carácter que no puede ser asociado con ninguna regla de reconocimiento lexicográfico, por lo que no puede producir un *token* para ser aprovechado por el reconocedor sintáctico. En este caso, el compilador emite un error al programador y se termina la compilación.
2. **Los errores sintácticos:** Se producen en la etapa de reconocimiento sintáctico del compilador y resultan cuando el reconocedor recibe un *token* que no pertenece al con-

junto de *tokens* esperados; por lo que no podrá continuar. En este caso, se emite un error al programador y el reconocedor intenta recuperarse del error utilizando la técnica de *Panic Mode*, la cual descarta *tokens* hasta que encuentra uno en el cual puede continuar el reconocimiento. Esta técnica no garantiza una recuperación exitosa, por lo que es posible que el compilador reporte uno o más falsos positivos, o no reconozca otros errores en la entrada.

3. **Los errores de contexto:** Son aquellos ocurridos por violaciones a las reglas de declaración de variables, uso de una variable no definida, mal uso de una función, entre otros. El compilador es capaz de reportar todos los errores de contexto encontrados sobre operaciones independientes. Esto significa que si es encontrado un error en una sub-operación perteneciente a una operación más compleja, entonces toda la operación será considerada como errónea. Por otro lado, los errores de tipos ocurren durante la etapa de verificación de tipos y acontecen cuando el programador intenta ejecutar una operación sobre la cual los tipos de los operandos no coinciden con aquellos definidos para la operación.

### 5.1.13. Errores de ejecución

Existen tres clases de errores para los cuales graciela genera código para ser verificados a tiempo de ejecución, causando la terminación del programa: *overflow*, división por cero, y la verificación de aserciones.

1. ***Overflow:*** Ocurren cuando una operación aritmética supera el valor máximo o mínimo representable por graciela, causando un desbordamiento aritmético.
2. **División por cero:** Cuando se encuentra el valor cero en un operando derecho de una división se emite un mensaje de error.

3. **Aserciones:** Ocurren cuando el estado de las variables del programa no cumple con la verificación de una aserción.

# Capítulo 6

## Conclusiones y recomendaciones

En el presente Proyecto de Grado se diseñó e implementó un lenguaje de programación fundamentado en las características del lenguaje GCL desarrollado por Edsger Dijkstra, el cual fue bautizado con el nombre de GraCieLa. Tiene un propósito didáctico, enfocado en apoyar prácticas disciplinadas de programación y sustentar el uso de estrategias formales para el desarrollo de algoritmos, en particular aquellas discutidas en los cursos de Algoritmos y Estructuras de la USB.

Para implementar el compilador de este lenguaje se emplearon dos herramientas fundamentales. La primera fue el lenguaje de programación Haskell y un conjunto de librerías adecuadas para la implantación del *Front-End* del compilador. El uso de Haskell por sobre otros lenguajes produjo grandes beneficios. En primer lugar es un lenguaje estático de verificación de tipos fuerte, garantizando un producto sin errores de tipos ni errores por transformaciones hechas por el lenguaje. Además, la librería **Parsec** y la capacidad de combinar cÓmputos monádicos en Haskell, simplificó notablemente el desarrollo de los reconocedores lexicográfico y sintáctico. Por último, los tipos algebraicos polimórficos provistos por Haskell facilitaron en gran medida la creación de aquellas estructuras de datos necesarias para las distintas etapas del compilador.

La segunda fue el *Back-End* provisto por el proyecto LLVM, el cual se acopló al *Front-End* para la culminación del compilador. Su principal característica aprovechada por el compilador, es la generación de código ejecutable nativo para un gran número de plataformas, simplificando así esta etapa de la compilación. Además, el *Back-End* LLVM cuenta con un gran número de optimizaciones que pueden ser aplicadas al código nativo generado que no fueron consideradas para el compilador, por lo que para futuros proyectos se recomienda analizar el conjunto de optimizaciones disponibles y aplicar aquellas convenientes.

El producto final del presente Proyecto de Grado es un lenguaje enfocado en la enseñanza, con su respectivo compilador, para ser usado en el curso de Laboratorio de Algoritmos I de la USB. El lenguaje es estático y de verificación de tipos fuerte e incluye las siguientes herramientas:

- Procedimientos y funciones puras.
- Pase de parámetros *in*, *in-out*, *out* y por referencia.
- Verificación de aserciones, precondición, postcondición, invariante y función de cota.
- Conjunto de instrucciones basada en GCL.
- Entrada y Salida de datos simple.
- Arreglos de más de una dimensión con base cero.

Para futuros proyectos enfocados en mejorar y ampliar el lenguaje GraCieLa, se presentan las siguientes recomendaciones:

- Proveer la capacidad de creación de tipos de datos enumerados definidos por el usuario, que puedan ser integrados al sistema de operadores aritméticos y relacionales del lenguaje. Estos podrán ser utilizados como cualquier otro tipo primitivo del lenguaje GraCieLa.

- Incluir la manipulación de apuntadores por parte del programador. Para ello el sistema de tipos del lenguaje GraCieLa debe ser ampliado con el fin de introducir el tipo **apuntador**, el cual debe ser recursivo y permitir la representación de direcciones de memoria hacia algún otro valor u otras direcciones de memoria. Esta característica sería aprovechada principalmente por los estudiantes del curso Laboratorio de Algoritmos II, ya que el tema de apuntadores es de gran importancia en el curso.
- Ofrecer al programador la posibilidad de crear estructuras de datos propias, mediante la implementación de Tipos Abstractos de Datos (TAD's). De este modo, se podrían manejar de forma simple, segura y eficiente datos que posean diversos atributos de tipos diferentes.

Todas estas recomendaciones brindan la posibilidad de aumentar el conjunto de herramientas que se encuentran a la disposición del usuario para la representación de datos complejos y el desarrollo de nuevos algoritmos. Además de ser de utilidad para los cursos de Laboratorio de Algoritmos II y III, implementar estas características no acarrea mayor dificultad si se toma en consideración la siguiente secuencia de acciones: primero debe ampliarse el reconocedor lexicográfico con aquellas palabras reservadas necesarias para la nueva característica del lenguaje. Luego, debe modificarse el analizador sintáctico para incluir todas las reglas necesarias para su reconocimiento, además de ampliar la definición del AST (en caso de ser necesario, proveyendo el constructor correspondiente). Posteriormente, debe ser realizada una investigación sobre código intermedio LLVM para analizar aquellas construcciones que permitan la implantación de la característica, para después incorporarla al generador de código intermedio LLVM. Por último, ya que el *Back-End* del compilador está provisto por LLVM, no se requiere la modificación del mismo.

# Bibliografía

- [1] Documentación del generador de código de máquina LLVM. <http://llvm.org/docs/CodeGenerator.html>. Consultado en Noviembre de 2015.
- [2] Documentación del lenguaje de código intermedio LLVM. <http://llvm.org/docs/LangRef.html>. Consultado en Noviembre de 2015.
- [3] Estándar IEEE-754. <http://babbage.cs.qc.edu/courses/cs341/IEEE-754references.html>. Consultado en Noviembre de 2015.
- [4] Introducción a Haskell. <https://wiki.haskell.org/Introduction>. Consultado en Noviembre de 2015.
- [5] Librería Control.Monad.RWS.Strict. <http://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-RWS-Strict.html>. Consultado en Noviembre de 2015.
- [6] Librería Data.Map. <http://hackage.haskell.org/package/containers-0.5.6.3/docs/Data-Map-Strict.html>. Consultado en Noviembre de 2015.
- [7] Librería Data.Range.Range. <https://hackage.haskell.org/package/range-0.1.2.0/docs/Data-Range-Range.html>. Consultado en Noviembre de 2015.

- [8] Librería Data.Text. <https://hackage.haskell.org/package/text>. Consultado en Noviembre de 2015.
- [9] Librería Data.Tree. <http://hackage.haskell.org/package/containers-0.5.6.3/docs/Data-Tree.html>. Consultado en Noviembre de 2015.
- [10] Librería LLVM.General. <https://hackage.haskell.org/package/llvm-general>. Consultado en Noviembre de 2015.
- [11] Librería LLVM.General.Pure. <https://hackage.haskell.org/package/llvm-general-pure>. Consultado en Noviembre de 2015.
- [12] Modulo Commands::Guarded de Perl. <https://metacpan.org/pod/Commands::Guarded>. Consultado en Enero de 2016.
- [13] Paquetes ofrecidos por la comunidad de Haskell. <https://www.haskell.org/platform/>. Consultado en Noviembre de 2015.
- [14] Página web del lenguaje GaCeLa. <http://web.archive.org/web/20100405235417/http://gacela.labf.usb.ve/GacelaWiki/>. Consultado en Enero de 2016.
- [15] Página web del lenguaje GCL 1.2. <http://www.dc.fi.udc.es/~cabalar/gcl/>. Consultado en Enero de 2016.
- [16] Página web oficial de Haskell *Platform*. <https://www.haskell.org/platform/>. Consultado en Noviembre de 2015.
- [17] Página web oficial del lenguaje Haskell. <https://www.haskell.org/>. Consultado en Noviembre de 2015.
- [18] Página web oficial del proyecto LLVM. <http://llvm.org>. Consultado en Noviembre de 2015.

- [19] *Monads en Haskell*. <https://wiki.haskell.org/Monad>. Consultado en Noviembre de 2015.
- [20] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 2nd edition, 2007.
- [21] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, 1st edition, 1986.
- [22] David Gries. *The Science Of Programming*. Springer-Verlag, 1st edition, 1981.
- [23] John E. Hopcroft, Rajeev Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.
- [24] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. Prentice Hall, 1st edition, 1990.
- [25] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: An Appetizer*. Springer, 1st edition, 2007.
- [26] Tobias Nipkow and Gerwin Klein. *Concrete Semantics: with Isabelle/HOL*. Springer, 1st edition, 2015. Disponible en Internet: <http://www.concrete-semantics.org/concrete-semantics.pdf>, consultado en Octubre de 2015.
- [27] Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 3rd edition, 2009.
- [28] Thomas A. Sudkamp. *Languages and Machines*. Addison-Wesley, 2nd edition, 1997.
- [29] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1st edition, 1993.

# Apéndice A

## Sistema de pruebas basado en Tripletas de Hoare

A continuación, será mostrado un sistema de pruebas compuesto por un conjunto de reglas basadas en Tripletas de Hoare para un lenguaje de programación genérico:

### 1. Skip

$$\frac{P \rightarrow Q}{\{P\} \text{ SKIP } \{Q\}}$$

*SKIP* es la instrucción que no tiene efecto alguno sobre el estado de las variables del programa. Si el predicado  $P$  implica a  $Q$ , entonces la instrucción *SKIP* con la precondición  $\{P\}$  y postcondición  $\{Q\}$  es válida.

### 2. Asignación

$$\frac{P \rightarrow Q[a/x]}{\{P\} \text{ } x ::= a \text{ } \{Q\}}$$

La instrucción asignación es la encargada de sustituir el valor de una variable. Si el predicado  $P$  implica al predicado  $Q$  con todas las ocurrencias de  $x$  sustituidas por el valor  $a$ , entonces la instrucción  $x ::= a$  con precondición  $\{P\}$  y postcondición  $\{Q\}$  es válida.

### 3. Concatenación

$$\frac{\{P_1\} c_1 \{P_2\} \quad \{P_2\} c_2 \{P_3\}}{\{P_1\} c_1; c_2 \{P_3\}}$$

La instrucción concatenación permite la secuenciación de instrucciones. Si la instrucción  $c_1$  con precondición  $\{P_1\}$  y postcondición  $\{P_2\}$  es válida y la instrucción  $c_2$  con precondición  $\{P_2\}$  y postcondición  $\{P_3\}$  también es válida, entonces la instrucción  $c_1 ; c_2$  es válida. El predicado  $P_2$  sirve para describir aquel conjunto de estados necesario para que la concatenación de las instrucciones  $c_1$  y  $c_2$  sea válida.

### 4. Condicional

$$\frac{\{P \wedge b\} c_1 \{Q\} \quad \{P \wedge \neg b\} c_2 \{Q\}}{\{P\} IF b THEN c_1 ELSE c_2 \{Q\}}$$

La instrucción condicional permite la elección entre las instrucciones  $c_1$  y  $c_2$ , dependiendo de la validez del predicado  $b$  sobre el estado de las variables. En el caso que  $b$  evalúe a cierto, entonces se ejecuta  $c_1$  y, en caso contrario, se ejecuta  $c_2$ . La conjunción de los predicados  $P$  y  $b$  en la precondición de  $c_1$  y la conjunción de los predicados  $P$  y la negación de  $b$  en la precondición de  $c_2$ , ambos con la postcondición  $\{Q\}$ , son válidas; entonces, la instrucción  $\{P\} IF b THEN c_1 ELSE c_2 \{Q\}$  es válida. Es decir, cada instrucción  $c_i$  perteneciente a las ramas del condicional debe validar a  $\{P\}$ , acompañado con  $b$  o  $\neg b$  dependiendo del caso, con  $\{Q\}$ .

### 5. Repetición

$$\frac{\{P \wedge b\} c \{P\}}{\{P\} WHILE b DO c \{P \wedge \neg b\}}$$

La instrucción repetición facilita la ejecución de una instrucción mientras que el predicado  $b$  evalúe a cierto sobre el estado de las variables. Si la instrucción  $c$  con la precondición  $\{P \wedge b\}$  y la postcondición  $\{P\}$  es válida, entonces la instrucción repetición es válida, con precondición  $\{P\}$  y postcondición  $\{P \wedge \neg b\}$ . El predicado  $P$  es llamado el invariante de la repetición, ya que debe ser cierto en la precondición y en la postcondición de la instrucción. Además, el negado de  $b$  es parte de la postcondición porque la instrucción terminó y la instrucción  $c$  no debe seguir ejecutándose.

## 6. Consecuente

$$\frac{[P \rightarrow P'] \quad \{P\} c \{Q\} \quad [Q \rightarrow Q']}{\{P'\} c \{Q'\}}$$

La regla del consecuente no involucra ninguna construcción del lenguaje y su propósito es ajustar la precondición y postcondición de la premisa  $\{P\} c \{Q\}$ . Este proceso es conocido como fortalecer la precondición y debilitar la postcondición; es decir, si la implicación  $A \rightarrow B$  es cierta, entonces el conjunto de estados representado por  $\{A\}$  es más fuerte que aquel representado por  $\{B\}$ .

# Apéndice B

## Teoría de tipos

Una forma de dividir los lenguajes es por su forma de realizar las verificaciones de tipos. Los lenguajes de tipos fuertes son lenguajes que no permiten violaciones al sistema de tipos, por lo que todas las operaciones deben ser realizadas sobre operandos cuyos tipos coincidan con aquellos en la definición de la operación. Mientras que los lenguajes de tipos débiles no cumplen con esta característica, ya que en el caso de encontrar una operación sobre la cual los tipos de sus operandos no son compatibles, aplicará las transformaciones necesarias para hacerlos coincidir con la definición de la operación.

Otra es dependiendo del momento en que se realiza la verificación de tipos y existen dos clasificaciones. La primera es **verificación de tipos estática**, la cual se realiza durante la compilación del programa, mientras que la segunda es **verificación de tipos dinámica** y se realiza a tiempo de ejecución.

Las clasificaciones son ortogonales, por lo que un lenguaje estático de tipos fuertes realiza la verificación de tipos a tiempo de compilación y no permite operaciones sobre tipos no compatibles. Mientras que un lenguaje dinámico de tipos débiles hace la verificación a tiempo de ejecución y convierte operandos en el caso que sea necesario. Una ventaja de la estrategia estática de tipos fuerte es que un compilador para el lenguaje puede reportar todos los

errores de tipos al programador a tiempo de compilación, lo que no sucede en un lenguaje dinámico. Esto garantiza que la gran mayoría de errores debidos a inconsistencia de tipos serán detectados antes de ejecutar, siendo necesarias menos pruebas de regresión y cobertura sobre los programas. Otra ventaja es que se reduce el tiempo de correcciones en aquellos programas que no funcionan correctamente debido a una transformación automática hecha por el lenguaje, por lo que el programador debe encargarse de escribir aquellas que sean necesarias.

# **Apéndice C**

## **Tabla de operadores**

La siguiente tabla contiene todos los operadores permitidos en el lenguaje GraCieLa como también la precedencia existente entre ellos.

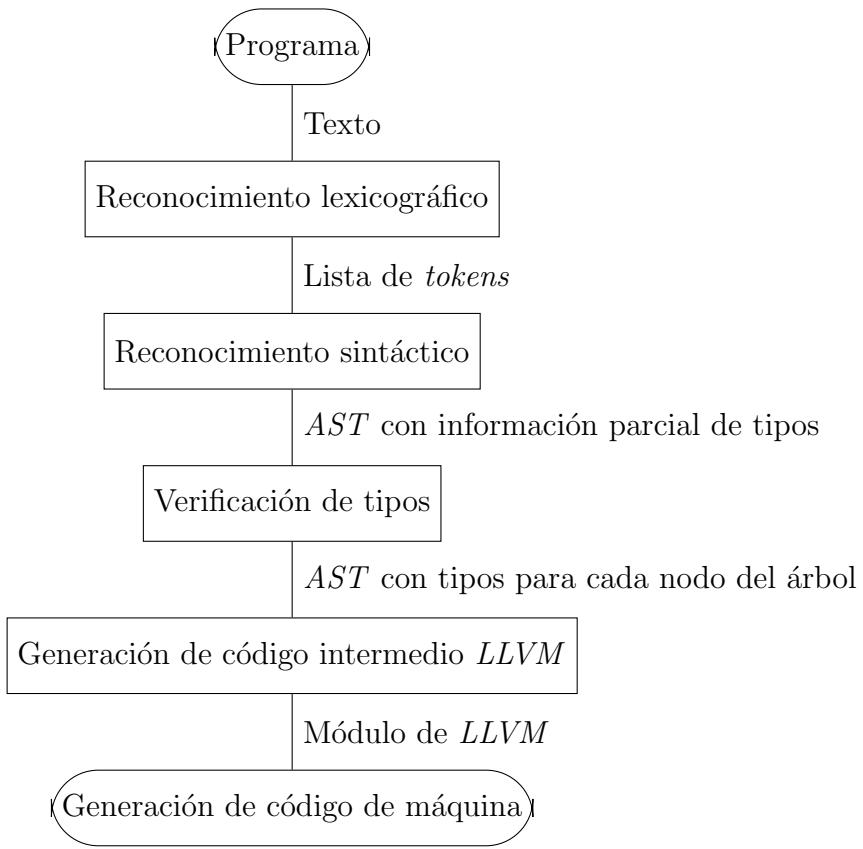
Precedencia	Operador	Operación	Operandos
1	!	Negación	Unario
	-	Menos	
2	*	Multiplicación	Binario
	/	División	
	$\wedge$	Potencia	
	mod	Módulo	
	max	Máximo	
	min	Mínimo	
3	+	Suma	Binario
	-	Resta	
4	<	Menor	Binario
	$\leq$	Menor igual	
	>	Mayor	
	$\geq$	Mayor igual	
5	$\equiv$	Equivalencia	Binario
	$\neq$	Inequivalencia	
6	$\wedge$	Disjunción	Binario
	$\vee$	Conjunción	
7	$\Rightarrow$	Implicación	Binario
	$\Leftarrow$	Consecuencia	

Cuadro C.1: Tabla de operadores

## **Apéndice D**

### **Flujo de datos del compilador**

El diagrama de flujo a continuación muestra cinco etapas de transformación del programa a compilar. Es importante señalar que no hace falta la transformación de código intermedio LLVM a código de máquina porque existe una librería en el sistema encargada de esa tarea.



# Apéndice E

## Funciones Predefinidas

1. **toInt:** Es la función para convertir al tipo `int` y está definida para los tipos `double` y `char`. Para el primero, sustrae la parte decimal y retorna la parte entera del número real. Para el segundo, retorna el código decimal correspondiente de la tabla ASCII.
2. **toDouble:** Función definida para los tipos `int` y `char` que se encarga de convertir valores al tipo `double`. Para el tipo `int` se retorna un número real con la parte entera igual al número entero el cual se desea convertir y la parte decimal igual a 0. En el caso del tipo `char` es equivalente a ejecutar la función `toInt` y posteriormente la función `toDouble`, es decir, retorna el código decimal correspondiente de la tabla ASCII.
3. **toChar:** Función de conversión al tipo `char`, definida para los tipos `int` y `double`. Para valores del tipo `int`, retorna el carácter representante del número en la tabla ASCII y para el tipo `double`, es equivalente a la concatenación de las funciones `toInt` y `toChar`.
4. **abs:** Está definida sobre los tipos `int` o `double` y retorna el valor absoluto del número. Mantiene el tipo del valor original.
5. **sqrt:** Calcula la raíz cuadrada de un valor del tipo `int` o `double`. El resultado es del

tipo `double`.

Es importante recalcar que las funciones descritas no efectúan verificaciones sobre los datos a utilizar al momento de ejecución, por lo que en caso de usar valores para los cuales la función no está definida, su comportamiento es indefinido y se deja a riesgo del programador.

# Apéndice F

## Gramática del lenguaje

La gramática del lenguaje GraCieLa es la siguiente:

### - Descripción de un programa

$$\langle \text{program} \rangle \models \text{program } \langle \text{tk-id} \rangle \text{ begin } \langle \text{defs} \rangle \langle \text{actions-list} \rangle \mid \lambda$$

### - Lista de definiciones

$$\langle \text{defs} \rangle \models \langle \text{def} \rangle \langle \text{defs} \rangle \mid \lambda$$

### - Definiciones posibles

$$\langle \text{def} \rangle \models \langle \text{function} \rangle \mid \langle \text{procedure} \rangle$$

### - Resto de argumentos de funciones

$$\langle \text{rest-arg-func} \rangle \models , \langle \text{arg} \rangle \langle \text{rest-arg-func} \rangle \mid \lambda$$

### - Identificador del Argumento

$$\langle \text{arg} \rangle \models \langle \text{tk-id} \rangle : \langle \text{type} \rangle$$

- **Tipos**

$$\langle \text{type} \rangle \models \text{int} \mid \text{char} \mid \text{double} \mid \text{boolean}$$

$$\langle \text{type} \rangle \models \text{array } \langle \text{bracket-list} \rangle \text{ of } \langle \text{type} \rangle$$

- **Primer índice del arreglo**

$$\langle \text{bracket-list} \rangle \models [ \langle \text{expr-bl} \rangle ] \langle \text{rest-bracket-list} \rangle$$

- **Numero del índice del arreglo**

$$\langle \text{expr-bl} \rangle \models \langle \text{tk-id} \rangle \mid \langle \text{tkint} \rangle$$

- **Resto de índices de los arreglos**

$$\langle \text{rest-bracket-list} \rangle \models [ \langle \text{expr-bl} \rangle ] \langle \text{rest-bracket-list} \rangle \mid \lambda$$

- **Cuerpo de las funciones**

$$\langle \text{func-body} \rangle \models \text{begin } \langle \text{func-expr} \rangle \text{ end}$$

- **Expresión del cuerpo de la función**

$$\langle \text{func-expr} \rangle \models \text{if } \langle \text{guardexprs} \rangle \text{ fi } \mid \langle \text{expr} \rangle$$

- **Lista de guardias con expresiones**

$$\langle \text{guard-list-exprs} \rangle \models \langle \text{guard-expr} \rangle \langle \text{rest-guard-expr} \rangle$$

- **Guardia para el condicional de funciones**

$$\langle \text{guard-expr} \rangle \models \langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle$$

- **Resto de guardias con expresiones**

$$\langle \text{rest-guard-expr} \rangle \models [] \langle \text{guard-expr} \rangle \langle \text{rest-guard-expr} \rangle \mid \lambda$$

- **Expresiones**

$$\langle \text{expr} \rangle \models \langle \text{id-rules} \rangle \mid \langle \text{const} \rangle \mid \langle \text{literals} \rangle \mid \langle \text{conv} \rangle$$

$$\langle \text{expr} \rangle \models \langle \text{quantification} \rangle \mid \langle \text{operations} \rangle \mid \langle \text{def-fun} \rangle$$

### - Operaciones

$$\langle \text{operations} \rangle \models ( \langle \text{expr} \rangle ) \mid \langle \text{expr} \rangle \langle \text{opbin} \rangle \langle \text{expr} \rangle$$

$$\langle \text{operations} \rangle \models - \langle \text{expr} \rangle \mid ! \langle \text{expr} \rangle$$

### - Constantes

$$\langle \text{const} \rangle \models \text{MIN\_INT} \mid \text{MAX\_INT}$$

$$\langle \text{const} \rangle \models \text{MIN\_DOUBLE} \mid \text{MAX\_DOUBLE}$$

### - Literales

$$\langle \text{literals} \rangle \models \langle \text{tk-bool} \rangle \mid \langle \text{tk-int} \rangle \mid \langle \text{tk-char} \rangle$$

$$\langle \text{literals} \rangle \models \langle \text{tk-double} \rangle \mid \langle \text{tk-string} \rangle$$

### - Conversiones

$$\langle \text{conv} \rangle \models \text{toInt} ( \langle \text{expr} \rangle ) \mid \text{toDouble} ( \langle \text{expr} \rangle )$$

$$\langle \text{conv} \rangle \models \text{toChar} ( \langle \text{expr} \rangle )$$

## - Funciones por defecto del lenguaje

$$\langle \text{def-fun} \rangle \models \text{abs} ( \langle \text{expr} \rangle ) \mid \text{sqrt} ( \langle \text{expr} \rangle )$$

### - Expresiones con identificadores

$$\langle \text{id-rules} \rangle \models \langle \text{tk-id} \rangle \mid \langle \text{array-access} \rangle \mid \langle \text{function-call} \rangle$$

### - Acceso a arreglos

$$\langle \text{array-access} \rangle \models \langle \text{tk-id} \rangle \langle \text{bracket-list} \rangle$$

### - Llamadas a funciones

$$\langle \text{function-call} \rangle \models \langle \text{tk-id} \rangle ( \langle \text{args-call} \rangle )$$

### - Argumentos de las llamadas

$$\langle \text{args-call} \rangle \models \langle \text{expr-list} \rangle \mid \lambda$$

### - Procedimientos

$\langle \text{procedure} \rangle \models \text{proc } \langle \text{tk-id} \rangle ( \langle \text{argsp} \rangle ) \text{ begin } \langle \text{decs} \rangle \langle \text{pre} \rangle$

$\langle \text{procedure} \rangle \models \langle \text{action} \rangle \langle \text{post} \rangle \text{ end}$

### - Argumentos de los procedimientos

$\langle \text{argsp} \rangle \models \langle \text{proc-arg} \rangle \langle \text{rest-args-proc} \rangle \mid \lambda$

### - Resto de argumentos de los procedimientos

$\langle \text{rest-args-proc} \rangle \models , \langle \text{arg} \rangle \langle \text{rest-arg-proc} \rangle \mid \lambda$

#### - Tipos de argumentos

$\langle \text{proc-arg} \rangle \models \langle \text{argin} \rangle \mid \langle \text{arginout} \rangle \mid \langle \text{argout} \rangle \mid \langle \text{argref} \rangle$

#### - Argumento del tipo ref

$\langle \text{argref} \rangle \models \text{ref } \langle \text{arg} \rangle$

#### - Argumento del tipo out

$\langle \text{argout} \rangle \models \text{out } \langle \text{arg} \rangle$

#### - Argumento del tipo in-out

$\langle \text{arginout} \rangle \models \text{inout } \langle \text{arg} \rangle$

#### - Argumento del tipo in

$\langle \text{argin} \rangle \models \text{in } \langle \text{arg} \rangle$

#### - Bloque de declaraciones

$\langle \text{decs} \rangle \models \langle \text{dec-list} \rangle \langle \text{reading} \rangle \mid \lambda$

#### - Primera declaración de variable

$\langle \text{dec-list} \rangle \models \langle \text{declaration} \rangle ; \langle \text{rest-dec-list} \rangle$

#### - Resto de declaraciones de variables

$\langle \text{rest-dec-list} \rangle \models \langle \text{declaration} \rangle ; \langle \text{rest-dec-list} \rangle \mid \lambda$

**- Tipos de declaraciones**

$$\langle \text{declaration} \rangle \models \langle \text{const-assign} \rangle \mid \langle \text{var-assign} \rangle$$

$$\langle \text{declaration} \rangle \models \langle \text{var-dec} \rangle$$

**- Declaración de variables sin inicialización**

$$\langle \text{var-dec} \rangle \models \text{var } \langle \text{dec-id-list} \rangle : \langle \text{type} \rangle$$

**- Declaración de variables con inicialización**

$$\langle \text{var-assign} \rangle \models \text{var } \langle \text{dec-assign-list} \rangle : \langle \text{type} \rangle$$

**- Declaración de constantes**

$$\langle \text{const-assign} \rangle \models \text{const } \langle \text{dec-assign-list} \rangle : \langle \text{type} \rangle$$

**- Identificadores para declaraciones**

$$\langle \text{dec-id-list} \rangle \models \langle \text{tk-id} \rangle \langle \text{rest-decid-list} \rangle$$

**- Resto de identificadores de las declaraciones**

$$\langle \text{rest-decid-list} \rangle \models , \langle \text{tk-id} \rangle \langle \text{rest-decid-list} \rangle \mid \lambda$$

**- Lista de identificadores**

$$\langle \text{id-list} \rangle \models \langle \text{tk-id} \rangle \langle \text{m-array-access} \rangle \langle \text{rest-id-list} \rangle$$

**- Resto de identificadores**

$$\langle \text{rest-id-list} \rangle \models , \langle \text{tk-id} \rangle \langle \text{m-array-access} \rangle \langle \text{rest-id-list} \rangle \mid \lambda$$

**- Posible acceso a arreglo**

$$\langle \text{m-array-access} \rangle \models \langle \text{array-access} \rangle \mid \lambda$$

**- Asignación de las declaraciones**

$$\langle \text{dec-assign-list} \rangle \models \langle \text{dec-id-list} \rangle := \langle \text{expr-list} \rangle$$

**- Asignación**

$$\langle \text{assign-list} \rangle \models \langle \text{id-list} \rangle := \langle \text{expr-list} \rangle$$

**- Lista de expresiones**

$$\langle \text{expr-list} \rangle \models \langle \text{expr} \rangle \langle \text{rest-expr-list} \rangle$$

**- Resto de la lista de expresiones**

$$\langle \text{rest-expr-list} \rangle \models , \langle \text{expr} \rangle \langle \text{rest-expr-list} \rangle \mid \lambda$$

**- Entrada de valores**

$$\langle \text{reading} \rangle \models \text{read} ( \langle \text{id-list} \rangle ) \langle \text{maybe-with} \rangle ; \mid \lambda$$

**- Nombre de archivo**

$$\langle \text{maybe-with} \rangle \models \text{with} \langle \text{tk-string} \rangle \mid \lambda$$

**- Lista de aserción con acciones**

$$\langle \text{actions-list} \rangle \models \langle \text{assertion} \rangle \langle \text{rest-actions-list} \rangle$$

**- Resto de aserciones con acciones**

$$\langle \text{rest-actions-list} \rangle \models ; \langle \text{assertion} \rangle \langle \text{rest-actions-list} \rangle \mid \lambda$$

**- Aserción con acción**

$$\langle \text{assertion} \rangle \models \{ \text{a} \langle \text{assertion-expr} \rangle \text{a} \} \langle \text{action} \rangle \mid \langle \text{action} \rangle$$

**- Acciones**

$$\langle \text{action} \rangle \models \langle \text{assign} \rangle \mid \text{skip} \mid \langle \text{cond} \rangle \mid \langle \text{rep} \rangle$$

$$\langle \text{action} \rangle \models \langle \text{block} \rangle \mid \text{abort} \mid \langle \text{random-acc} \rangle$$

$$\langle \text{action} \rangle \models \langle \text{write-acc} \rangle \mid \langle \text{function-call} \rangle$$

- Cálculo de valores aleatorios

$$\langle \text{random-acc} \rangle \models \text{random } \langle \text{tk-id} \rangle$$

- Asignación de variables

$$\langle \text{assign} \rangle \models \langle \text{id-list} \rangle := \langle \text{expr-list} \rangle$$

- Regla para el condicional

$$\langle \text{cond} \rangle \models \text{if } \langle \text{guard-list} \rangle \text{ fi}$$

- Regla para la repetición

$$\langle \text{rep} \rangle \models \langle \text{invar} \rangle \langle \text{bound-f} \rangle \text{ do } \langle \text{guard-list} \rangle \text{ od }$$

- Regla para el bloque

$$\langle \text{block} \rangle \models |[ \langle \text{dec-list} \rangle \langle \text{actions-list} \rangle ]|$$

- Acciones para la escritura

$$\langle \text{write-acc} \rangle \models \text{write} ( \langle \text{expr-list} \rangle )$$

$$\langle \text{write-acc} \rangle \models \text{writeln} ( \langle \text{expr-list} \rangle )$$

- Regla para el invariante

$$\langle \text{invar} \rangle \models \{\text{inv } \langle \text{assertion-expr} \rangle\}$$

- Regla para la función de cota

$$\langle \text{bound-f} \rangle \models \{\text{bound } \langle \text{expr} \rangle\}$$

- Lista de guardias con acciones

$$\langle \text{guard-list} \rangle \models \langle \text{guard-action} \rangle \langle \text{rest-guard-list} \rangle$$

- Resto de las guardias

$$\langle \text{rest-guard-list} \rangle \models [] \langle \text{guard-action} \rangle \langle \text{rest-guard-list} \rangle$$

$$\langle \text{rest-guard-list} \rangle \models \lambda$$

**- Guardias del condicional usado como acción**

$$\langle \text{guard-action} \rangle \models \langle \text{expr} \rangle \rightarrow \langle \text{action} \rangle$$

**- Cuantificadores**

$$\langle \text{quantification} \rangle \models (\langle \text{op-quant} \rangle \langle \text{tk-id} \rangle : \langle \text{type} \rangle | \langle \text{expr} \rangle | \langle \text{expr} \rangle)$$

**- Precondición**

$$\langle \text{precondition} \rangle \models \{\text{pre } \langle \text{assertion-espr} \rangle\}$$

**- Postcondición**

$$\langle \text{postcondition} \rangle \models \{\text{post } \langle \text{assertion-espr} \rangle\}$$

**- Operadores binarios**

$$\langle \text{opbin} \rangle \models \langle \text{op-ari} \rangle | \langle \text{op-bool} \rangle | \langle \text{op-rel} \rangle$$

**- Operadores aritméticos**

$$\langle \text{op-ari} \rangle \models + | - | * | / | \text{min}$$

$$\langle \text{op-ari} \rangle \models \text{max} | \text{mod} | \wedge$$

**- Operadores lógicos**

$$\langle \text{op-bool} \rangle \models \wedge | \vee | ==> | <=$$

**- Operadores relacionales**

$$\langle \text{op-rel} \rangle \models == | != | > | < | <= | >=$$

**- Expresión regular de los identificadores**

$$\langle \text{tk-id} \rangle \models ([a - zA - Z])([a - zA - Z0 - 9_?])^*$$

**- Regla para los valores lógicos**

$$\langle \text{tk-bool} \rangle \models \text{true} | \text{false}$$

-Expresión regular para números enteros

$$\langle \text{tk-int} \rangle \models [0 - 9]([0 - 9])^*$$

- Expresión regular para números flotantes

$$\langle \text{tk-double} \rangle \models [0 - 9]([0 - 9])^*[.][([0 - 9])^*$$

- Expresión regular para los caracteres

$$\langle \text{tk-char} \rangle \models '(.)'$$

- Expresión regular para las cadenas de caracteres

$$\langle \text{tk-string} \rangle \models ""(.)^""$$

# Apéndice G

## Instrucciones de instalación

En el siguiente apéndice se presenta el *link* donde se encuentra disponible *online* las instrucciones a seguir para instalar adecuadamente el compilador del lenguaje GraCieLa, tanto en maquinas con distribuciones *GNU/Linux*, como para las que poseen el sistema operativo *Windows* o OS X.

Disponible en: <https://github.com/GraCieLa-USB/Archivos/tree/master/Instalaci%C3%B3n>

## **Apéndice H**

### **Manual de Usuario y Programas**

### **Ejemplos**

En el siguiente apéndice se presenta el *link* donde se encuentra disponible *online* el manual de usuario del lenguaje GraCieLA, como también programas ejemplos extraídos del libro [24] escritos en GraCieLa.

Disponible en: <https://github.com/GraCieLa-USB/Archivos/>

# Apéndice I

## Archivos del Compilador

En el siguiente *link* se encuentra disponible todos los archivos fuentes que componen el compilador del lenguaje GraCieLa escrito en Haskell.

Disponible en: <https://github.com/GraCieLa-USB/Archivos/tree/master/ArchivosCompilador>

## Apéndice J

### Instrucciones de instalación

En el siguiente apéndice se presenta el *link* donde se encuentra disponible *online* las instrucciones a seguir para instalar adecuadamente el compilador del lenguaje GraCieLa, tanto en maquinas con distribuciones *GNU/Linux*, como para las que poseen el sistema operativo *Windows* o OS X.

Disponible en: <https://github.com/GraCieLa-USB/Archivos/tree/master/Instalaci%C3%B3n>

# Apéndice K

## Glosario de términos

En el presente apéndice figuran los acrónimos y definiciones de mayor importancia para facilitar la comprensión del contenido de este Proyecto de Grado.

### K.1. Acrónimos

- **ASCII:** *American Standard Code for Information Interchange.*
- **GCL:** *Guarded Command Language.*
- **LL:** *Left to right, Leftmost derivation.*
- **LR:** *Left to right, Rightmost derivation.*
- **LLVM:** *Low Level Virtual Machine.*
- **USB:** Universidad Simón Bolívar.
- **UTF-8:** *8-bit Unicode Transformation Format.*

## K.2. Definiciones

- **Analizador lexicográfico** : El principal objetivo del analizador lexicográfico es leer el texto de entrada y reconocer todas las palabras y símbolos permitidos por el lenguaje para producir los *tokens* (elementos reservados del lenguaje) que se le suministraran al analizador sintáctico, en caso de que exista un símbolo que no pertenezca al conjunto permitido, terminara la ejecución con un error. Además, se encarga de ignorar todos los espacios en blanco entre cada símbolo y los comentarios del lenguaje escritor por el programador.
- **Analizador sintáctico** : Definimos el analizador sintáctico también llamado *parser* como la parte del compilador que se encarga de revisar que el conjunto de *tokens* recibidos del *lexer* posea la estructura correcta en base a una gramática dada. En caso de que no haya ningún error sintáctico el *parser* construye el árbol de derivación asociado al programa de entrada. En caso de que si ocurra un error, no se generara el árbol de derivación y terminara la ejecución del programa.
- **Árbol de Sintaxis Abstracta** : Un árbol de sintaxis abstracta (*Abstract Syntax tree*) es la estructura más eficiente de manejar la información semántica del código fuente, cada uno de los nodo del árbol denota una construcción que ocurre en el código fuente sin alterar la relación jerárquica entre los símbolos de la gramática y a la vez omitiendo información que no sea relevante luego del análisis léxico y sintáctico, simplificando (abstrayendo) la semántica del lenguaje. Esta estructura permite realizar la verificación de tipos y realizar optimizaciones de manera sencilla y efectiva con el objetivo de luego convertirlo en código intermedio.
- **ASCII<sup>1</sup>** : Es un esquema de codificación de caracteres compuesto por códigos de 7 bits,

---

<sup>1</sup><http://www.asciitable.com/>

para un total de 128 caracteres representables. Está dividido en caracteres de control y caracteres imprimibles. Existe una ampliación llamada **Extended ASCII**, la cual agrega un bit a la codificación permitiendo representar hasta 255 caracteres.

- **Back-End** : Gracias a la división entre el *Front-End* y el *Back-End* este ultimo puede generar a partir del código intermedio proporcionado por el *Front-End* el código de maquina para la plataforma que se desee, como también permite que el *Back-End* sea utilizado para compilar otros lenguajes de programación.
- **Backtracking** [20]: En el contexto de reconocimiento sintáctico, un reconocedor con *backtracking* es aquel capaz de rehacer la entrada e intentar otra producción en caso de fallar intentando una producción y consumiendo parte de la entrada.
- **Código de máquina** : Es el código resultante del compilador que es reconocible e interpretable por el microprocesador de la maquina destino, este código es distinto entre las diferentes arquitecturas existentes pero suele tener instrucciones similares. Éste código está compuesto por un conjunto de instrucciones que son ejecutadas en secuencia y determinan las acciones de la maquina, también existen cambios de flujo que suelen ser causados por eventos externos o por el mismo programa. Pero, en el fondo todas estas acciones se traducen a señales eléctricas simbolizada con el número 1 y a la ausencia de ellas mismas simbolizada con el número 0.
- **Código intermedio** : Se puede considerar como un código abstracto independiente, debido a que no es ni un lenguaje de alto nivel ni tampoco uno de bajo nivel. Sus principales propiedades deben, una fácil producción del mismo y ser sencillo de traducir al código de maquina destino, esto permite que el código intermedio sea muy eficiente gracias a las facilidades de optimizaciones que posee y también permite al código ser re-dirigible dado a que el *Front-end* no se altera y no es dependiente de la maquina destino,

permitiendo tener distintos *Back-ends* para generar el código máquina específico de cada una de las plataformas, sin tener que realizar todo el compilador nuevamente.

- **Compilador** : Podemos definir un compilador como un software encargado de traducir un texto o código fuente escrito en un lenguaje de programación de alto nivel a otro lenguaje, el cual suele ser uno de bajo nivel, también llamado lenguaje de máquina. Las tareas que abarca un compilador se pueden agrupar en dos fases *Front-End* y *Back-End*, esta división se genera de la necesidad de separar los procesos que son independientes de la plataforma destino y los que no lo son.
- **Debugging** : Proceso por el cual se identifica los errores de un programa con el fin de utilizar herramientas y técnicas para detectarlos y corregirlos.
- **Entrada Estándar** : Mecanismo por el cual un usuario le indica al programa la información que éste debe procesar. Por defecto, el teclado es la entrada estándar.
- **Expresión**: Es una secuencia de uno o más operandos (constantes, variables o funciones) que junto con sus respectivos operadores dan como resultado un solo valor único.
- **Front-End** : Se encarga de analizar el código fuente, verificar que no existen errores lexicográficos ni sintácticos para poder crear el árbol de derivación asociado al programa junto a su tabla de símbolos correspondiente. Por último, se genera el código intermedio que recibirá el *Back-End*.
- **Función** : Podemos definir una función como una sección del programa que recibe valores de entrada para calcular un valor resultante, por lo tanto las funciones solo se pueden utilizar en las expresiones de un programa.
- **Instrucción** : Una instrucción es la única operación que se encuentra definida en un procesador por lo tanto, son las acciones que generan cambios de estados dentro del

programa.

- **L-values** : Son aquellos valores que dependen de una dirección en memoria y se les asigna un nuevo valor asociado.
- **Panic Mode** [20]: Es un método de recuperación de errores para un reconocedor sintáctico. Al momento de descubrir un error, el reconocedor descarta símbolos de la entrada hasta encontrar uno que pertenezca al *conjunto de sincronización*, el cual está compuesto por *tokens* seleccionados por el diseñador del reconocedor.
- **Procedimiento** : Porción de código o programa que se encuentra dentro de un programa más grande, el procedimiento ejerce una tarea específica y es relativamente independiente del resto del programa, se utiliza como una instrucción más del programa.
- **Profiling** : Es el análisis de rendimiento que se le realiza a un programa para reunir información del tiempo dedicado a la ejecución de cada una de sus partes, con el fin de optimizar los puntos críticos.
- **Recuperación de errores** : Llamamos recuperación de errores al proceso por el cual un compilador reconoce un error léxico, sintáctico o semántico cometido por el programador, permitiendo informar con claridad y exactitud las razones por las que el error fue probablemente cometido y también corregir.<sup>el</sup> el error mediante herramientas y técnicas heurísticas implantadas dentro del compilador, con el fin de reducir la cantidad de errores derivados por el error inicial.
- **R-values** : Son aquellos valores que no dependen de una dirección en memoria, por lo que puede ser una expresión cualquiera.
- **Salida estándar** : Mecanismo por el cual el programa muestra la información procesada al usuario. Por defecto, la pantalla (terminal) es la salida estándar.

- **Semántica** : Se refiere a las reglas que determinan el significado de los programas sin importar cual sea su sintaxis.
- **Token** [20]: Un *token* es una estructura representante de un **lexema** que indica de forma explícita su categoría sintáctica. Un lexema es una cadena de caracteres con un significado dado por la categoría sintáctica a la que pertenece. Por ejemplo, el *token* llamado **ID** puede representar a los identificadores de variables en un lenguaje de programación, y un lexema válido pudiera ser la cadena "foo".
- **Variable Global** : Variable accesible en todo el código del programa. Es decir, se puede hacer referencia a su dirección de memoria en cualquier parte del programa.
- **UTF-8**<sup>2</sup>: Es una codificación de caracteres que puede ser tan compacta como ASCII (en un archivo son solo letras del alfabeto Inglés), pero también puede contener cualquier carácter unicode (con un incremento en el tamaño del archivo).
- **Monad State**<sup>3</sup>: Es un *Monad* definido en Haskell que permite simular un estado arbitrariamente complejo y modificable durante la ejecución de una función.
- **Monad State Transformer**<sup>4</sup>: Análogo al *Monad State* pero añadiendo la posibilidad de combinar su funcionamiento con otro *Monad* interno.

### K.3. Librerías

- **Data.Attoparsec**<sup>5</sup> : Librería ofrecida por la comunidad Haskell para la definición de reconocedores sintácticos, enfocada en tratar de manera eficiente con protocolos de red y archivos de texto complicados.

---

<sup>2</sup><http://www.fileformat.info/info/unicode/utf8.htm>

<sup>3</sup><http://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-State.html>

<sup>4</sup><http://hackage.haskell.org/package/mtl-1.1.0.2/docs/Control-Monad-State.html>

<sup>5</sup><https://hackage.haskell.org/package/atoparsec>

- `Data.ByteString`<sup>6</sup> : Librería en la cual se implementan vectores de *bytes* eficientes en tiempo y espacio. Adecuados para usos que requieren alto rendimiento, tanto en términos de grandes cantidades de datos como de alta velocidad.
- `Data.String`<sup>7</sup> : Librería para el tipo *String* y sus operaciones asociadas. Un *String* en Haskell no es más que una lista de caracteres y se define de la siguiente forma:

```
data String = [Char]
```

- Happy<sup>8</sup>

Es un generador de reconocedores sintácticos para Haskell. Dada la especificación de una gramática, Happy genera un módulo con un reconocedor correspondiente. Es flexible, por lo que es posible tener más de un reconocedor en el mismo programa y más de un no-terminal de entrada para una gramática.

---

<sup>6</sup><https://hackage.haskell.org/package/bytestring-0.10.6.0/docs/Data-ByteString.html>

<sup>7</sup><https://hackage.haskell.org/package/base-4.8.1.0/docs/Data-String.html>

<sup>8</sup><https://hackage.haskell.org/package/happy>