Universidad Nacional de Educación a Distancia

Escuela Técnica Superior de Ingeniería Informática

Proyecto de fin de Grado en Ingeniería Informática

Compilación y Programación Funcional

Pablo Cumpián Díaz

Dirigido por: Fernando López Ostenero

Curso 2022/2023, convocatoria junio

Compilación y Programación Funcional

Proyecto de fin de Grado en Ingeniería Informática

de modalidad específica

Realizado por: Pablo Cumpián Díaz

Dirigido por: Fernando López Ostenero

Fecha de lectura y defensa: 14 de julio de 2023

Agradecimientos

A mis padres y abuelos, que lo hicieron posible.
A Joaquín y Alejandro, que siempre me inspiraron.
A Carlos Caballero y mis profesores, que me pusieron en el camino.
A Tamara, por su apoyo todos estos meses.

Resumen

El presente proyecto expone el desarrollo de un compilador bajo el paradigma de programación funcional. El beneficio de este enfoque, como se podrá comprobar, reside principalmente en la claridad conceptual que aporta la arquitectura empleada, acercando lo máximo posible el código a la especficación del lenguaje.

El objetivo del proyecto no es sino el de proponer una alternativa sólida y bien fundamentada a las recurrentes aproximaciones imperativas en la docencia de este campo, como pretexto para dar cabida a conceptos avanzados de este paradigma. Se da pie por tanto a distintas construcciones y patrones funcionales especialmente útiles para emprender un desarrollo de estas dimensiones.

El resultado práctico de este trabajo es un compilador, esto es, la implementación de un lenguaje de programación. El alcance de este proyecto no reside, sin embargo, en la potencia del lenguaje en sí, sino en el recorrido que esto supone, explorando desde la traducción del código fuente hasta la ejecución directa sobre una máquina, permitiéndonos exponer en amplitud los retos que supone cada fase de este proceso.

En cuanto a los objetivos de aprendizaje, el proyecto viene justificado por el deseo de adquirir mayor destreza con la programación funcional, así como profundizar en el conocimiento de la compilación y su terminología, técnicas y arquitectura. En este sentido, el trabajo pretende dar pie al bagaje teórico de las asignaturas de Procesadores del Lenguaje del Grado en Ingeniería Informática, de forma que el resultado práctico del mismo esté acompañado con breves apuntes que describan los problemas y soluciones adoptadas.

Abstract

This project presents the development of a compiler under the functional programming paradigm. The benefit of this approach, as will be seen, lies mainly in the conceptual clarity provided by the architecture used, bringing the code as close as possible to the specification of the language.

The aim of the project is simply to propose a solid and well-founded alternative to the recurrent imperative approaches in the teaching of this field, as a pretext to accommodate advanced concepts of this paradigm. Therefore, different constructions and functional patterns especially useful to undertake a development of these dimensions are given rise to.

The practical result of this work is a compiler, that is, the implementation of a programming language. The scope of this project does not lie, however, in the power of the language itself, but in the journey that this entails, exploring from the translation of the source code to direct execution on a machine, allowing us to expose in breadth the challenges involved in each phase of this process.

As for the learning objectives, the project is justified by the desire to become proficient with functional programming, as well as to deepen the knowledge of compilation and its terminology, techniques and architecture. In this sense, the work is intended to give rise to the theoretical background of the Language Processor subjects of the CS degree, so that the practical result of the work is accompanied by brief notes describing the problems and solutions adopted.

Palabras clave

- **Compilador**: programa informático encargado de traducir código de un lenguaje origen a otro de destino mediante el encadenamiento de una fase de análisis y otra de síntesis.
- **Análisis**: proceso de descomposición de un objeto en las partes que lo constituyen. En el caso de la compilación es la primera fase que se encarga de dar estructura a las distintas partes de un programa dado.
- **Síntesis**: proceso de composición de las partes en un todo. Dentro de la compilación se refiere a la segunda fase en la cual se genera el programa a ejecutar a partir de las partes analizadas anteriormente.
- **Frontend**: primera parte del proceso de compilación en la cual se llevan a cabo el análisis léxico, sintáctico y semántico.
- **Backend**: segunda parte del proceso de compilación en la que tiene lugar la generación de código ejecutable (síntesis).
- **Análisis léxico**: primera fase del frontend encargada del reconocimiento y recolección de cada lexema.
- **Análisis sintáctico**: segunda fase del frontend encargada de verificar la correcta codificación de un programa respecto a la sintáxis del lenguaje empleado, así como de producir un árbol sintáctico que refleje la construcción del mismo.
- **Análisis Semántico**: tercera del fase frontend encargada de comprobar que las restricciones semánticas se cumplen en un determinado programa. Ello implica comprobar que exista coherencia entre las construcciones tipadas (símbolos y expresiones), unicidad de los símbolos, etc.
- **Traducción**: cada una de las fases backend que se encargan de transformar sucesivamente una determinada representación del lenguaje para dar lugar a otra.
- **Lexema**: cada uno de los símbolos y palabras que componen el léxico de un determinado lenguaje.
- **Tipo**: construcción del lenguaje nativa o descrita por el usuario que define un dominio de operaciones desde el punto de vista del frontend (análisis). Para el backend representa una construcción que expresa el tamaño en palabras de memoria que ocupa cada uno de los valores de ese dominio. Ej: un entero es un tipo numérico que puede ser usado para las operaciones aritméticas y de comparación, y es a su vez representado en código objeto mediante 4 bytes.
- **Ámbito**: espacio de ejecución dentro de un programa en el cual unos determinados símbolos están disponibles para su uso y referencia.
- **Código fuente**: Programa o librería escrito en un determinado lenguaje de programación.
- **Árbol Sintáctico (Abastract Syntax Tree)**: estructura de datos en árbol que representa un programa para realizar un análisis semántico sobre él.

- **IR**: Representación intermedia utilizada internamente por un compilador para representar un programa de forma independiente a la plataforma de ejecución.
- **Código objeto**: código nativo de una plataforma concreta que, previo ensamblado, permite ser ejecutado en la misma.
- **Programación funcional**: paradigma de los lenguajes de programación en los que se estructura los programas mediante la composición de funciones, resultando especialmente indicada para el cálculo.
- **Parser Combinator**: Subprograma que realiza el análisis léxico y sintáctico del código fuente.
- **Mónada**: Abstracción funcional que permite realizar cálculos parciales de forma que estos puedan componerse entre sí para realizar cálculos más complejos. Usualmente es un tipo de datos que provee de programación con efectos al proveer de una forma de estado interno.

Keywords

- **Compiler**: Computer program responsible for translating code from a source language to a target language through the combination of an analysis phase and a synthesis phase.
- **Analysis**: Process of breaking down an object into its constituent parts. In the case of compilation, it is the first phase that structures the different parts of a given program.
- **Synthesis**: Process of combining the parts into a whole. Within compilation, it refers to the second phase in which the executable program is generated from the previously analyzed parts.
- **Frontend**: First part of the compilation process where lexical, syntactic, and semantic analysis takes place.
- **Backend**: Second part of the compilation process where the generation of executable code (synthesis) occurs.
- **Lexical analysis**: First phase of the frontend responsible for recognizing and collecting each lexeme of the source code.
- **Syntactic analysis**: Second phase of the frontend that verifies the correct encoding of a program with respect to the syntax of the employed language, and produces a syntax tree that reflects its construction.
- **Semantic analysis**: Third phase of the frontend that checks whether the semantic constraints are satisfied in a given program. This involves checking for coherence among typed constructions (symbols and expressions), uniqueness of symbols, etc.
- **Translation**: Each of the backend phases that successively transform a particular representation of the language into another.
- **Lexeme**: Each symbol and word that make up the lexicon of a specific language.
- **Type**: Native language construction or user-defined one that defines a domain of operations from the frontend perspective (analysis). For the backend, it represents a construction that expresses the memory size occupied by each value in that domain. Example: an integer is a numeric type that can be used for arithmetic and comparison operations, and is represented in object code by 4 bytes.
- **Scope**: Execution space within a program in which certain symbols are available for use and reference.
- **Source code**: Program or library written in a specific programming language.
- **Abstract Syntax Tree (AST)**: Tree-like data structure that represents a program for performing semantic analysis on it.
- **IR (Intermediate Representation)**: Intermediate representation used internally by a compiler to represent a program independently of the execution platform.

- **Object code**: Native code of a specific platform that, after assembly, can be executed on that platform.
- **Functional programming**: Programming paradigm in which programs are structured through the composition of functions.
- **Parser Combinator**: Subroutine that performs lexical and syntactic analysis of the source code.
- **Monad**: Functional abstraction that allows partial calculations to be composed together to perform more complex calculations. It is usually a data type that provides programming with effects by providing a way to manage internal state.

Índice

Índice de figuras

Capítulo 1

Introducción

La investigación y el desarrollo de lenguajes de programación se encuentran entre las áreas más fructíferas de la Ingeniería Informática. Si bien el estudio de este campo queda a veces restringido al ámbito académico, uno puede observar como en algo más de medio siglo no han cesado de publicarse lenguajes de programación nuevos[2]. En la figura 1.1 se muestra el número de nuevos lenguajes de programación notables que han aparecido cada década desde los años 50.

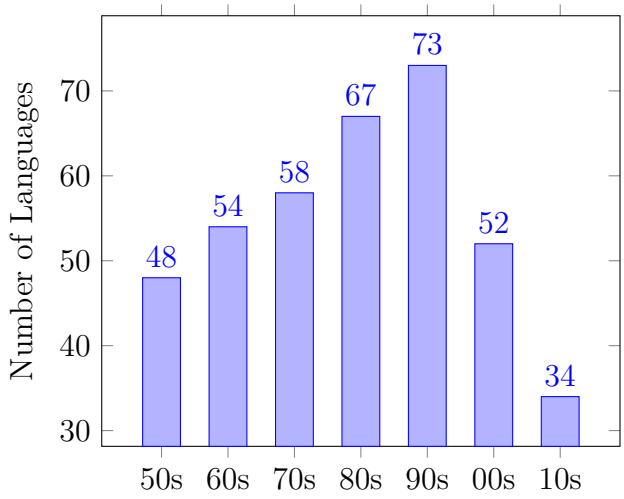


Figura 1.1: Número de lenguajes de programación notables por década. Datos de Wikipedia

Conforme la materia ha logrado conformar su propio dominio, la docencia y aproximaciones adoptadas se han encaminado mayoritariamente hacia paradigma imperativo si uno observa los compiladores implementados para buena parte de los lenguajes más usados[1].

Con todo, no ha dejado de ser una disciplina compleja, en la que bien para conseguir allanar su curva de aprendizaje o simplificar su implementación, muchos de los proyectos cuentan con dos facilidades que abstraen al estudiante de parte del funcionamiento de algunos componentes:

- O bien emplean frameworks especialmente diseñados para cubrir casi por completo alguna de las fases, y/o
- por la propia arquitectura del proyecto se asumen menos fases que las descritas. En este

[1]En buena parte esto es debido a que es un hito para medir la madurez de un lenguaje el poder escribir en él un compilador para sí mismo, lo que se conoce como *self-hosted compiler*. Dado que entre los lenguajes más usados los imperativos son mayoría, estos tienden también a contar por ello con compiladores escritos a la manera imperativa.

sentido podemos encontrar, por ejemplo, compiladores JIT o intérpretes[2]. La razón de ello es abstraer de la complejidad de adaptar el código destino según la especificación de una máquina u otra.

1.1. Estado del arte

Con lo expuesto hasta aquí, destaca la ausencia de compiladores para lenguajes de propósito general que por un lado aborden todas las fases canónicas de un compilador, y además utilicen el paradigma funcional. Dentro de este reducido número podemos encontrar proyectos de compiladores escritos en Haskell, una búsqueda en un sitio como GitHub [3] arroja resultados entre los que cabe destacar:

- PureScript: lenguaje funcional con una sintaxis parecida a Haskell que compila a JavaScript.
- Write you a Haskell[3]: libro-tutorial que enseña a escribir un compilador de Haskell en Haskell.
- Kaleidoscope: libro-tutorial que muestra cómo escribir un compilador usando LLVM.
- GHCJS: compilador de Haskell a JavaScript.

Si bien muchos de ellos muestran parcialmente alguna estructura parecida a la de este proyecto, no cubren integralmente todos las partes:

- No implementan un parser propio, sino que utilizan generadores de parsers a partir de la gramática del lenguaje.
- Algunos traducen directamente a JavaScript.
- Otros utilizan como backend LLVM.

Otro tipo de lenguajes frecuentes son los DSLs (*Domain Specific Languages*). Sólo con Haskell se pueden encontrar en el mismo sitio ejemplos tan diversos como los siguientes:

- Hspec: para la especificación de suites de testing en Haskell.
- Shake: alternativa a make.
- Yesod: para definir aplicaciones web.
- Clay: preprocesador de CSS en Haskell.

[2] Un intérprete provee a la implementación de un lenguaje de programación de una plataforma de ejecución, evitando la traducción completa hasta llegar al código máquina específico de cada arquitectura.

[3] `https://github.com/`

- Liquid Haskell: Haskell con tipos más expresivos

Sin embargo, este tipo de lenguajes no necesitan tampoco generar código ensamblador, ni tienen por qué cubrir las construcciones clásicas (funciones, control de flujo, bucles ...). De hecho, para el dominio que abordan no suelen necesitar ni siquiera ser Turing-completo.

En definitiva, no se encuentran con facilidad compiladores escritos en el paradigma funcional que muestren todas las fases corrientes hasta generar código máquina por sí mismos. Su realización representa, por tanto, no solamente un proyecto original, sino también una meta didáctica, que aglutina un buen número de materias de la Ingeniería Informática y permite aunarlas para un propósito común.

1.2. Objetivo

La meta de este proyecto es proponer y fundamentar un enfoque funcional como alternativa al paradigma imperativo en la realización de un compilador. Como se ha expuesto anteriormente, aunque no es una temática ajena a este paradigma, las introducciones a esta materia suelen estar copadas por aproximaciones imperativas debido principalmente a la familiaridad que ofrecen los lenguajes de esa familia como forma de hacer más accesible un proyecto complejo. El presente trabajo aventura la idea de que el abordaje de un compilador desde el paradigma funcional resulta más comprensible y sencillo de realizar que su versión imperativa.

1.3. Exposición. Estructura de los capítulos

Una vez definidos los objetivos del proyecto es posible dotar de antemano de un orden expositivo, así como de cierta estructura en la retórica de lo que se pretende defender.

Los capítulos que desarrollan la programación del compilador siguen las fases que hemos definido anteriormente como canónicas:

- Metodología: capítulo 2
- Diseño del lenguaje: capítulo 3
- Análisis sintáctico: capítulo 4
- Análisis semántico: capítulo 5
- Generación de IR: capítulo 6
- Generación de código ensamblador: capítulo 7

Cada uno de los capítulos de la implementación cuentan con una introducción en donde se expone qué cometido desempeña cada fase, así como la estructura de datos que produce en una

breve especificación. Le sigue un apartado de traducción entre ambas fases, en donde se aportan breves extractos de código para ilustrar el aporte teórico presentado. Finalmente la conclusión sintetiza lo expuesto en el capítulo, así como ofrece una pequeña comparativa entre la aproximación imperativa y un extracto de la codificación funcional propuesta, cumpliendo el objetivo de resaltar la claridad de la última. En este sentido, además, se recogen algunas carencias y se propone un sencillo bosquejo para poder abarcarlas, poniendo en valor la ampliabilidad que este enfoque nos brinda.

Capítulo 2

Metodología

En este capítulo se dirimen algunas decisiones técnicas tomadas en diversos ámbitos del proyecto, así como su justificación de cara a dotarlo de la coherencia suficiente en cuanto a la arquitectura y diseño.

2.1. Arquitectura

En este apartado se expone la estructura del proyecto desde la propia implementación, dotándola de un esqueleto que articule todas las fases respecto a unos patrones de software que brillan por su simplicidad y su capacidad de cohesión interna.

Para introducir la arquitectura de este tipo de proyectos es necesario acogerse a ciertos conceptos generales que otorguen ortogonalidad entre dominio y software, de forma que estos conceptos queden relacionados con las partes de la arquitectura.

2.1.1. Análisis y síntesis. *Frontend* y *backend*

The dragon book[4] comienza exponiendo la compilación como un proceso basado en dos etapas: análisis y síntesis. En la primera el programa se descompone en partes para poder analizarlo sintáctica y semánticamente, y en la segunda se lleva a cabo la traducción propiamente dicha al lenguaje objeto para poder ser ejecutado.

Es recurrente encontrar en los manuales de la materia referencias a una arquitectura de *frontend* y *backend*[5, p.14], que permite separar una primera fase de proceso del *stream* de caracteres de entrada —con la posibilidad de retorno de errores y *feedback*— y otra que le sigue encargada de producir el código ensamblador que constituiría la salida de todo el compilador.

En nuestra arquitectura, *frontend* y *backend* quedan estrechamente relacionados con análisis y síntesis, respectivamente. De esta manera queda dispuesta una cadena de procesamiento con cada etapa perfectamente definida en su cometido.

Estas etapas quedan además ligadas por las estructuras de datos que producen:

- El *frontend* construye a partir del código fuente un **AST** (*Abstract Syntax Tree*) que ayuda a dar sentido a los lexemas que recibe el compilador. Esta estructura en árbol permite ubicar cada construcción, nodo, dentro de un contexto mayor que la alberga y frente al que comprobar -analizar- determinadas coerciones (semántica).

- En contraste, el *backend* produce -sintetiza- a partir de la descomposición de dicho árbol una **estructura lineal**, ídónea para representar el código ensamblador y que además obedece a la naturaleza secuencial de la ejecución del procesador.

En la figura 2.1 se puede ver una representación gráfica de este proceso.

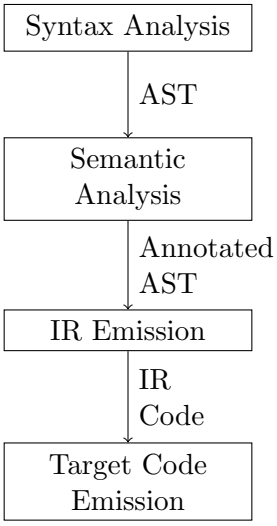


Figura 2.1: Arquitectura del compilador

Conforme a esto cada etapa queda a su vez subdividida en dos más:

- El análisis primero tiene una fase de producción del árbol para luego poder analizarlo realmente (análisis sintáctico y semántico).
- La síntesis recoge este árbol y emite a partir de él una primera representación intermedia (IR), para después traducirla al conjunto de instrucciones deseado: x86, ARM ...

Esta división va estructurar tanto los distintos módulos del proyecto como los capítulos de este escrito. Los compiladores cuentan en los casos más relevantes con muchas otras fases y subdivisiones interesantes de barajar una vez se abarque esta aproximación inicial.

2.2. Enfoque, herramientas e implementación

Respecto al diseño, se ha optado por otorgar un papel protagonista a la fase de compilación, dejando el ensamblado y enlazado del código objeto para su ejecución manual por el usuario (expuestas en el anexo A). Esto se debe a que el ensamblado (la sustitución de los nemotécnicos por el binario) es una traducción obvia y carente de interés teórico para este trabajo. El enlazado constituye en sí mismo un ámbito completamente distinto y muy dependiente de la plataforma de ejecución (ubicación de las librerías, sistema operativo, etc.), aunque, por otro lado, el lenguaje objeto, ARMv7, es lo suficientemente ubicuo como para poder obtener las herramientas adecuadas para realizar esta fase sin demasiado esfuerzo, y con ello se evita distraer al lector del propósito principal del trabajo. Como compensación a esto se adjunta un apéndice donde se describe poder ensamblar, enlazar y ejecutar este código en un entorno GNU/Linux.

2.2.1. Enfoque y paradigma

La elección del enfoque funcional para la construcción de un proyecto de estas características es idónea por las siguientes razones:

- Al ser, como ha quedado expuesto, un ejercicio íntegramente de cálculo, casa perfectamente con este paradigma.
- Asegura un comportamiento determinista.
- Facilita el desarrollo de pruebas.
- Claridad: como quedará reflejado, la codificación en este tipo de lenguajes suele ser clara e intuitiva, facilitando, además del proceso de traducción, la definición de las estructuras de datos clave.

2.2.2. Lenguaje

Por su alcance, versatilidad y librerías se ha escogido Haskell como lenguaje huésped en el que implementar el lenguaje objetivo[1]. No sólo se le ha tenido en cuenta debido a su madurez, sino que también ha pesado en la decisión el bagaje con el que cuenta el lenguaje en algunas publicaciones académicas. Además, este desarrollo supone una buena excusa para exponer las capacidades de este tipo de lenguajes y particularmente de Haskell:

- Composición de funciones puras y totales.
- Un sistema de tipado fuerte.
- Abstracciones potentes para representar las relaciones conceptuales del dominio.

Cada uno de estos epígrafes está relacionado con una construcción interna del compilador. En este sentido el compilador nos servirá como pretexto para el desarrollo de los anteriores puntos, a la vez que ilustramos las diferentes partes de las que se compone el proceso. No obstante, a día de la publicación de este trabajo no se han encontrado referencias completas de todo el proceso que hagan uso de él: si bien la etapa del parser está bien estudiada y cuenta con bastante literatura y librerías, no se han encontrado ni desarrollos ni escritos que muestren como desarrollar el análisis semántico ni el backend en Haskell, por lo que a partir de esa fase, el sustento del proyecto es sólo teórico o reflejo e inspiración de distintos escritos en otros lenguajes.

Entre los otros lenguajes barajados cabe destacar la presencia de Idris[2], un lenguaje con una sintaxis parecida a Haskell, que cuenta además como característica estrella los *dependent types*, lo que le hace especialmente atractivo para proyectos come el presente debido a que permite,

[1] Los ejemplos de código incluidos obedecen únicamente a un propósito expositivo, por lo que pueden verse alterados respecto del código incluido en el repositorio.

[2] https://www.idris-lang.org/

a través de un avanzado sistema de tipos dotar a estos de relaciones más fuertes entre ellos, obteniendo como resultado un compilador más robusto y seguro. Sin embargo, la adopción, fuentes y el número y madurez de librerías disponibles juegan a favor para Haskell, así como también la necesidad de utilizar, dentro del paradigma, un lenguaje lo más *mainstream* posible dentro de la comunidad de desarrolladores para el impacto de esta propuesta.

2.2.3. Herramientas de desarrollo

Junto con Haskell, el tooling para la construcción y gestión de dependencias se delega en Stack. En el primer anexo A se realiza un pequeño recorrido por las opciones y facilidades que nos brinda.

2.2.3.1. Control de versiones

La gestión del avance del proyecto se lleva a cabo con git, y para el hospedaje del mismo se ha recurrido a GitHub, como forma también de abrir el proyecto al reporte comunitario. De hecho, las propuestas de mejora señaladas en las conclusiones de cada capítulo tienen un *issue* relacionado en el panel del repositorio, para una vez presentada la implementación inicial poder constatar su avance.

2.2.4. Componentes como mónadas

Dado que, como se verá en los próximos capítulos, cada fase de la compilación requiere almacenar un estado interno para hacer posible la traducción de cada fase, la manera más ergonómica de responder a este requerimiento es el diseño de componentes mediante mónadas. En este sentido, cada componente será un traductor interno a cada etapa de la compilación.

Por comparación, la programación imperativa, propiamente la orientada a objetos, hubiese prescrito para esta necesidad el modelado de clases. La ventaja que ofrecen las mónadas a este respecto es que su comportamiento ya viene definido en los distintos tipos genéricos, de forma que el desarrollador únicamente debe indicar el tipo del estado a almacenar.

En lo que respecta a la arquitectura, cada tipo de mónada viene a hacer el papel de un patrón de diseño homólogo. Por ejemplo, la *state monad* [6] vendría a representar el mismo rol que el patrón de diseño repositorio [7].

2.2.4.1. Breve introducción a las mónadas

Pese a su ubicuidad en el mundo de la programación funcional, las mónadas constituye para el recién llegado a ella un reto en su adopción. Aunque no es el objetivo principal de este texto el dotar al lector de una sólida comprensión de las mónadas y su uso, se provee aquí -por su peso dentro del proyecto- de un pequeño tutorial que describa su funcionamiento. Además de su formalización más elemental, existen innumerables fuentes a las que acudir para consultar

definiciones y formas de uso. En lo sucesivo el texto se limita a dar una descripción simple para permitir continuar con la lectura del proyecto.

Una mónada es una clase genérica dentro de los lenguajes funcionales dedicada sobretodo a la computación con efectos colaterales. Es posible expresar su constitución con una comparación con la máquina de Von Neumann, lo que se muestra en la figura 2.2:

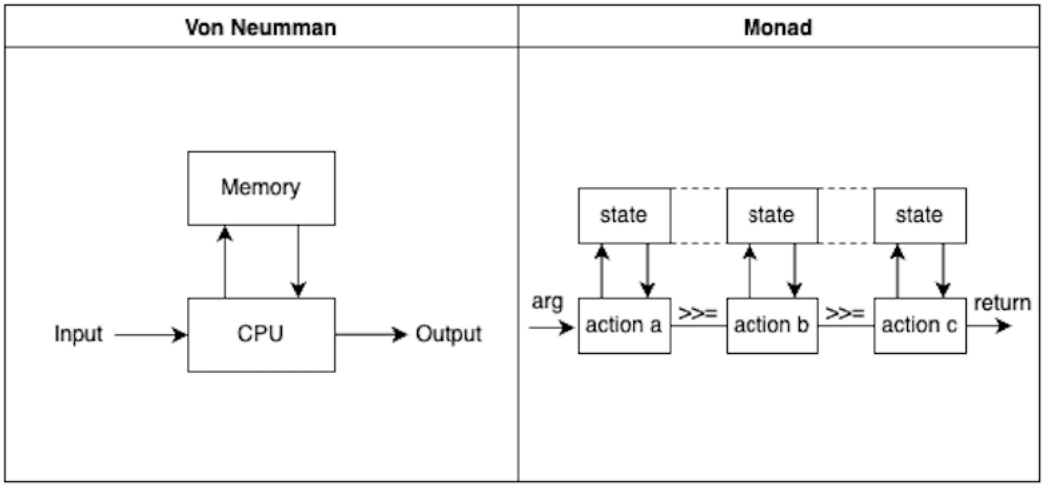


Figura 2.2: Arquitectura de Von Neumann comparada con el uso de varias mónadas enlazadas

Si bien esta arquitectura de computadores es capaz de calcular datos a partir de una entrada, de forma análoga a una función, su característica principal es la capacidad de albergar un *estado* interno en el que alterar una configuración dada inicialmente, lo que constituye el almacenamiento.

De forma similar, una mónada consigue realizar cómputos mediante la conexión de una función, su entrada y su salida, y un estado enlazado a ella. La potencia de esta abstracción radica en su capacidad de enlazar unas con otras, con el requerimiento de que la estructura del estado sea la misma, para poderlas conectar. Con ello, de igual forma que una computadora es capaz realizar cálculos complejos compuestos por otros más simples, varias mónadas con la misma signatura (mismo estado) serán capaces de realizar un computo complejo. La clase de la mónada en Haskell es la siguiente:

```
class Monad m where
  (>>=)  :: m a -> (  a -> m b) -> m b
  return ::   a                 -> m a
```

Durante los distintos capítulos se utilizarán diferentes tipos de mónadas, cada una adecuada para el tipo de problema o el tipo de operaciones que realizar sobre el estado almacenado.

2.2.5. Fases como compiladores

El lema que vertebra el diseño general y su implementación funcional propone que cada fase del compilador es, en sí misma, un compilador. Esto queda reflejado en Haskell mediante el uso de *type clases*:

```
class Compiler source target where
    compile :: source -> target
```

A través de esta clase, cada fase queda instanciada como un compilador, y esto vale para homologarlas entre sí junto al compilador mismo. Sobre este punto se volverá en las conclusiones del proyecto (ver capítulo 8) cuando haya que mostrar el ensamblaje de todas las partes.

Capítulo 3

Diseño del lenguaje

Con objeto de que el lector sepa reconocer adecuadamente las construcciones del lenguaje a implementar, tanto sintáctica como semánticamente, se expone en este capítulo una breve pero somera descripción del mismo. Así, resulta imprescindible este primer momento de definición para poder emprender la fase de análisis.

Siguiendo con la tradición no escrita de que el primer programa que se ha de teclear al aprender un nuevo lenguaje ha de escribir "Hola Mundo", en la figura 3.1 se muestra el código de dicho programa en nuestro lenguaje.

```
fun main() {
    print("hello world")
}
```

Figura 3.1: Ejemplo de *Hola mundo* en el lenguaje

3.1. Descripción general y características del lenguaje. Sintaxis

El lenguaje a implementar cuenta con una sintaxis muy similar a la de C y otros próximos a él, en aras de que resulte familiar al grado de no ser necesario aprenderlo como tal. Por otro lado, es un lenguaje básico, dado que el foco principal de interés es el desarrollo del compilador y no la potencia del lenguaje que modela.

Un ejemplo básico de nuestro lenguaje, con buena parte de sus construcciones expuestas, sería el mostrado en la figura 3.2. Sus características principales son su paradigma imperativo y que está fuertemente tipado y articulado en funciones, *statements* y expresiones.

```
fn max(a int, b int) -> int {
    if (a > b) {
        return a
    }
    return b
}

fn main() {
    if (max(1, 2) == 2) {
        print("2 es el numero mayor")
    } else {
        print("imposible")
    }
}
```

Figura 3.2: Ejemplo de un programa utilizando varias funciones y construcciones

Respecto de la entrada del compilador, esta será un único archivo fuente con codificación UTF-8. La única forma de salida será un archivo indicado donde volcar el código ensamblador generado, también en la misma codificación. Como puede observarse, el punto de entrada es la función *main*, la cual siempre ha de estar presente en cualquier programa. Los recursos con los que cuenta son los siguientes:

- Tipos enteros, booleanos y *strings* nativos
- Operaciones aritméticas (suma, resta y multiplicación), lógicas y de comparación de enteros.
- Control de flujo: `If`/`Else`.
- Bucle `While`.
- Funciones con retorno.
- Capacidad de imprimir cadenas de texto por terminal.

Existen otras construcciones que derivan de las ya expuestas aquí, tales como las construcciones de bucles `for` o `do/while`, que no se incluyen porque no añaden ninguna forma de traducción genuina como tal. Aunque interesantes, es posible derivar estas ampliaciones a partir de lo expuesto aquí, por lo que quedan fuera en aras de la brevedad.

3.1.1. Sintáxis del lenguaje en notación BNF

Al contar con una sintaxis tan reducida es posible dar una especificación breve pero clara y precisa de las combinaciones posibles de las construcciones mencionadas, como se muestra en la figura 3.3.

```
<program> := <function>*

<function> := fun <symbol> (<param>*) -> type? {<statement>*}

<param> := type <symbol>

<type> := int | bool | string

statement := <if> | <ifelse> | <while> | <assignment>
          | <var> | <return>

if := if (<expr>) {statement*}

ifelse := if (<expr>) {statement*} else {statement*}

while := while (expr) {statement*}

var := type <symbol> = <expr>

assignment := <symbol> = <expr>

return := return <expr>
```

Figura 3.3: Notación BNF de la sintaxis del lenguaje

3.2. Semántica: tipos y operaciones

Conforme a las características descritas, se pueden comenzar a añadir al código Haskell de nuestro compilador algunas descripciones que serán usadas por las distintas fases posteriormente. Pueden definirse, en primer lugar, las operaciones, aritméticas, lógicas y de comparación, que nuestro lenguaje soporta. Lo más sencillo para cerrar las combinaciones es definirlas como *sum types*:

```
data ArithOperation = Add | Sub | Mul
data LogicOperation = And | Or
data CompOperation  = Eq | Ne | Gt | Lt | Ge | Le
```

Al ser un lenguaje tipado, han de definirse también los tipos de las expresiones, estableciendo además las clases que describen, respectivamente, el tamaño de una expresión y las construcciones que sintetizan un tipo, junto a las operaciones que permiten obtener dichos valores. Esto puede hacerse así:

```
data TypeValue = IntType | BoolType | StringType

class Sized a where
    size :: a -> Int

class Typed a where
```

```
typeOf :: a -> TypeValue
```

3.3. Conclusiones

Como puede observarse, el lenguaje cuenta con lo básico entre los lenguajes de este tipo, a falta de paliar las siguientes deficiencias:

- Imposibilidad de codificación modular de programas: el compilador no es capaz de compilar varios archivos con referencias entre ellos.
- Sólo se cuenta con tipos nativos, el usuario no puede describir tipos derivados tales como *structs* o *arrays*.
- Falta de abstracciones de mayor nivel: no existen clases ni objetos ni posibilidad de crear estructuras de datos.
- El tipo *string* tan sólo habilita a su uso con valores estáticos y constantes y no es posible realizar operaciones con él más allá de la impresión por terminal.

Sin embargo, se considera que lo abarcado en este trabajo supone un punto de partida suficiente para completar estas faltas. En los siguientes capítulos se aludirá a algunas de estas deficiencias para poner ejemplos de cómo ampliar el lenguaje.

Una vez el lenguaje está bien definido y acotado en sus posibilidades es posible enfrentar la primera etapa de análisis sintáctico del compilador.

Capítulo 4

Análisis Sintático. Parsing

La primera etapa de la compilación requiere de la obtención de un código fuente en forma de texto y de su procesamiento para representarlo en una estructura que permita la interpretación sintáctica del mismo.

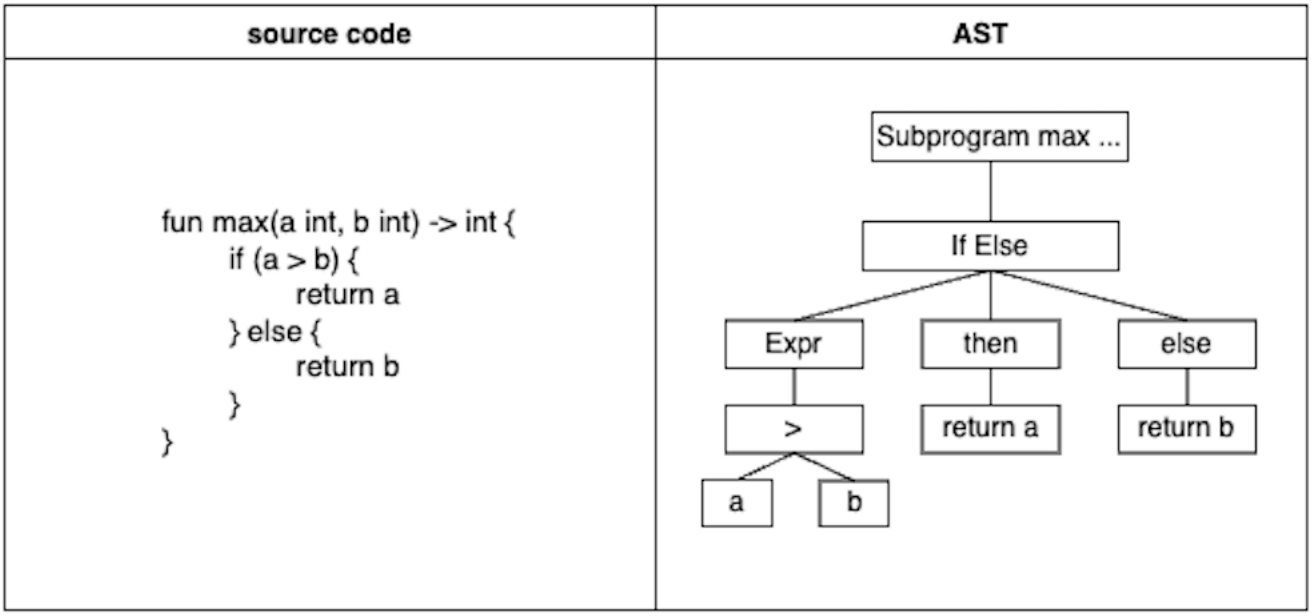


Figura 4.1: Transformación de código fuente en un AST

Para este cometido se usa un parser de cadenas de texto. La función de un parser es la de consumir el código fuente introducido y producir, en caso de que la entrada sea correcta sintácticamente, un AST. Un AST (*Abstract Syntax Tree*) es una estructura de datos en forma de árbol que permite representar un programa válido y procesarlo semánticamente a posteriori. Por ello, la otra característica principal del parser es la de poder reportar errores bien ubicados en el código fuente.

Habitualmente esta fase viene inserta entre otras dos más que aquí se omiten por no ser necesarias ni pertinentes:

- Análisis léxico (scanner): consumo del código y divisón en tokens para que el parser pueda ubicarlos sintácticamente.
- Preprocesado: modificación del código fuente bien de forma estática o dinámica mediante directivas incluidas en el código fuente.

Como se expondrá, el parser implementado ya recoge la funcionalidad del análisis léxico en su

recorrido, no siendo necesario dedicar una fase propia. Por otro lado, el lenguaje propuesto no cuenta con directivas ni macros que requieran de una fase de preprocesado.

4.1. Especificación

En este apartado se expone el esquema de la estructura de datos resultante de esta fase, el árbol de sintáxis, con el objetivo de presentar al lector un acercamiento a la primera transformación que ocurre dentro de la compilación.

La sintáxis del código fuente ha sido descrita en notación BNF en el capítulo de diseño del lenguaje. Es imprescindible un acercamiento a ella para poder reconocer las estructuras descritas en el árbol.

4.1.1. AST

Conforme a esta definición, nuestro tipo AST estaría compuesto mediante el uso de sum types en Haskell, lo que nos permite expresar estructuras de datos de tipo árbol a través de la recursión:

```
type AST = [Definition]

data ReturnType = Maybe TypeValue

data Definition = Subprogram String [Param] ReturnType [Statement]

data Literal = IntVal Int | BoolVal Bool | StringVal String

type Param = (String, TypeValue)

data TypeValue = IntType | BoolType | StringType

data Statement = If Expr [Statement]
               | IfElse Expr [Statement] [Statement]
               | While Expr [Statement]
               | Var String TypeValue Expr
               | Assign String Expr
               | Print Expr
               | CallStmt String [Expr]
               | Return (Maybe Expr)

data Expr = Add Expr Expr
          | Mul Expr Expr
          | Sub Expr Expr
          | Negate Expr
          | Lit Literal
          | SymbolRef String
          | Call String [Expr]
```

```
| Not Expr
| And Expr Expr
| Or Expr Expr
| Eq Expr Expr
| Ne Expr Expr
| Lt Expr Expr
| Gt Expr Expr
| Le Expr Expr
| Ge Expr Expr
```

4.2. Traducción

El proceso de traducción requiere en primera instancia consumir la cadena de texto introducida por el usuario en formato de archivo. Las dos opciones para ello son principalmente dos:

- Utilizar las bien conocidas expresiones regulares y un generador de parsers a partir de ellas (como por ejemplo sería el caso de la librería Alex [8])
- Implementar un parser combinator

En comparación, y desde el punto de vista funcional, un parser combinator permite la construcción de un parser que reconozca construcciones de nuestro lenguaje a partir de la combinación y composición de otros que reconozcan construcciones más pequeñas. Así, por ejemplo, un parser combinator de statements puede ser utilizado para reconocer todo el cuerpo de una función de nuestro lenguaje.

Esta idea ha sido expuesta principalemente por Graham Hutton en su manual sobre Haskell *Programming in Haskell* [9], fuente principal de lo que se expondrá a continuación. También Stephen Diehl ha cubierto con mayor amplitud esta misma aproximación en [3].

Los *parser combinators* son un conjunto de funciones encargadas de procesar la cadena de texto haciendo uso de mónadas. De esta forma un parser combinator nos garantiza:

- Expresividad: representar la sintaxis del lenguaje con ADTs (Abstract Data Types), de forma que de un vistazo podemos saber perfectamente como formular programas válidos (facilidad de interpretación)
- Funcionalidad: expresar a través de mónadas distintas coerciones, a saber: cuantificadores, valores alternativos, consumo de palabras reservadas, etc.
- Composición: a nivel de arquitectura, los parsers combinators nos permiten reflejar cada una de estas funcionalidades de forma aislada para componerlas luego.

Debido a que es un tipo de datos genérico, nos permite definir un tipo de parser para cada construcción del lenguaje. De esta manera reutilizamos código para expresiones, statements, palabras reservadas, literales, funciones, etc.

Existen varias librerías bien conocidas para llevar a cabo esta tarea, pero con objeto de discutir el funcionamiento de esta utilidad prescindiremos de su uso por el momento e implementaremos el tipo y sus propiedades desde cero.

4.2.1. Definición del parser

El tipo Parser es un tipo genérico: esto es, puede aceptar parsear cualquier tipo de los apuntados anteriormente (Expr, Statement, Param ...). Al instanciar el tipo estaremos diciendo cual de esas estructuras necesitamos obtener a partir de un texto (un String).

```
newtype Parser a = Parser { parse :: String -> [(a, String)] }
```

En base a esto la función raíz que se encargará de manejar el comportamiento general de nuestro parser es otra función genérica:

```
runParser :: Parser a -> String -> a
runParser m s =
  case parse m s of
    [(res, [])] -> res
    [(_, _)]    -> error "Parser cannot process the whole source code"
    _           -> error "Undefined error."
```

Como puede observarse, esta función es la que contiene la lógica para manejar el consumo incompleto de una cadena, lo que quiere decir que ha habido un error. En caso de que el parser haya conseguido consumir la cadena por completo, devolverá un valor del tipo adecuado. Así, por ejemplo, un `Parser Expr` devolverá un `Expr`, lo cual se puede leer como que un parser de expresiones devolverá una expresión.

4.2.2. Consumo del código fuente

Al ser representado como texto, el código fuente se consumirá mediante las siguientes funciones:

```
item :: Parser Char
item = Parser $ \s ->
  case s of
    []     -> []
    (c:cs) -> [(c,cs)]

satisfy :: (Char -> Bool) -> Parser Char
satisfy predicate = item `bind` \char ->
  if predicate char
  then return char
  else empty
```

- La función `item` avanza por el recorrido de una cadena situando el primer carácter en la parte a identificar, y el resto en la parte a consumir posteriormente. Este es un procedimiento análogo al de los buffers en los parsers imperativos.

- La función `satisfy` nos sirve para comprobar si el anterior carácter cumple con el predicado que indiquemos, y en tal caso devolver un parser que lo incluya.

De forma intuitiva, pues, se observa como un parser que reconoce una cadena no es más que la composición varios parsers que reconocen distintos caracteres adyacentes. Con esta premisa, es sencillo articular el reconocimiento de construcciones gramaticales, por lo que podemos crear constructores de Parsers a partir de cadenas, como en los siguientes ejemplos:

```
char :: Char -> Parser Char
char c = satisfy (c ==)

oneOf :: String -> Parser Char
oneOf s = satisfy (`elem` s)
```

La función `char` nos crea un Parser para reconocer un carácter determinado, que se le pasa como parámetro, mientras que la función `oneOf` nos crea un Parser para reconocer alguno de los caracteres presentes en su cadena de entrada.

Además de cuantificadores podemos definir a partir de aquí los propios Parsers que consuman literales:

```
intLiteral :: Parser Int
intLiteral = read <$> many1 digit
```

que es una función que lee un literal que representa un valor entero, reconociendo uno a uno sus dígitos (gracias a la función `many1`).

4.2.3. Parser como mónada

Para que un parser pueda resultar de la composición de otros tenemos que poder definirlo como una mónada, por lo que previamente hemos de definirlo como *Functor* y como *Applicative*:

```
instance Functor Parser where
  fmap f (Parser cs) = Parser (\s -> [(f a, b) | (a, b) <- cs s])

instance Applicative Parser where
  pure a = Parser (\s -> [(a, s)])
  (Parser cs1) <*> (Parser cs2) =
    Parser (\s -> [(f a, s2) | (f, s1) <- cs1 s, (a, s2) <- cs2 s1])
```

La funcionalidad lograda mediante estas declaraciones es la siguiente:

- Como *Functor*, el parser es capaz de aplicar una función a un determinado `Parser a` que reconoce una estructura de tipo a, para producir un `Parser b`, capaz de reconocer estructuras de otro tipo b. De esta forma podemos, por ejemplo, modificar una cadena a la vez que consumimos parte de la misma:

```
runParser (toUpper <*> item) "haskell"
-> [("H", "askell")]
```

- Como *Applicative*, es capaz de procesar cadenas secuencialmente sin importar la longitud de la secuencia.

```
drop2 :: Parser String
drop2 = pure (\a b -> [a, b]) <*> item <*> item

runParser drop2 "haskell"
-> [("ha", "skell")]
```

Una vez definido como aplicativo, podemos definir el parser como mónada de la siguiente forma:

```
instance Monad Parser where
  return = pure
  (>>=)  = Parser $ \s -> concatMap (\(a, s')
         -> parse (f a) s') $ parse p s
```

También es importante poder expresar un parser como la alternativa entre varios otros, para lo cual es preciso que implemente la clase `Alternative`:

```
instance Alternative Parser where
  empty = Parser (const [])
  (<|>) = option

option :: Parser a -> Parser a -> Parser a
option p q = Parser $ \s ->
  case parse p s of
    []     -> parse q s
    res    -> res
```

Mediante estas últimas instancias obtenemos las siguientes funcionalidades:

- Como *Monad* nos permite ampliar el cálculo secuencial que nos permitía como aplicativo, para realizar un cálculo a partir de las distintas partes que se vayan consumiendo. Esto nos permite, no solamente consumir un número de caracteres deseado, como se muestra en el ejemplo del *applicative*, sino permite componer los parsers para establecer reglas fácilmente reconocibles con la sintáxis *do*:

```
parseEmail :: Parser (String, String)
```

```
parseEmail = do
    user <- string
    arroba
    domain <- string
    dotcom
    return (user, domain)
```

- Como *Alternative* conseguimos la capacidad de declarar que una estructura del AST puede tener distintas vías válidas. Así por ejemplo el consumo de una definición en el lenguaje vendría dado por:

```
expr :: Parser Expr
expr = addExpr <|> minusExpr <|> multExpr <|> ...
```

4.2.4. Gramática respecto del lenguaje

La estructura de nuestro parser nos permite definir de forma aislada las palabras reservadas del lenguaje, instanciando el tipo como `Parser String`:

```
token :: Parser a -> Parser a
token p = do
    a <- p
    char " " <|> char "\n"
    return a

reserved :: String -> Parser String
reserved s = token (string s)

returnWord = reserved "return"
printWord = reserved "print"
...
```

De esta forma, combinando distintos tipos de parser llegamos a crear Parsers que reconocen construcciones de primer nivel de nuestro lenguaje:

```
printStmt :: Parser Statement
printStmt = do
    printWord
    expr <- parens expr
    return $ Print expr

statement :: Parser Statement
statement = ifElse <|> ifThen <|> while <|> printStmt
          <|> call <|> assign <|> var <|> returnStmt

stmtBlock :: Parser [Statement]
stmtBlock = braces $ many statement
```

```
function :: Parser Function
function = do
    reserved "fun"
    name   <- identifier
    params <- params
    returnType <- (reserved "->" >> retSig) <|> noRetSig
    stmts  <- stmtBlock

    return $ Function name params returnType stmts offset
```

El código anterior reproduce la gramática de las funciones descrita en el capítulo Diseño del lenguaje (ver figura 3.3). Según esto, cada función está compuesta de su cabecera, que empieza por la palabra reservada *fun*, su nombre, los parámetros un tipo de retorno opcional y el cuerpo de statements, (control de flujo, bucles, declaraciones y asignaciones, etc.) Se reproduce por tanto la gramática anotando el orden en el que se consumen los valores esperados.

4.2.5. Expresiones

En este apartado vamos a tratar sobre los Parsers que permiten que el lenguaje reconozca expresiones.

4.2.5.1. Precedencia

El esquema de precedencia deriva del uso de los Parsers como clase Alternative. Así por ejemplo, en el caso de los statements se elegirá entre la siguiente cadena de opciones:

```
statement :: Parser Statement
statement = ifElse <|> ifThen <|> while <|> printStmt
          <|> call <|> assign <|> var <|> returnStmt
```

que lleva implícita una jerarquía de precedencia, basada en el orden en el que se codifican las opciones.

De la misma forma, la precedencia de los operadores en nuestro lenguaje puede formularse definiendo la precedencia entre varios niveles:

```
expr :: Parser Expr
expr = equality

equality :: Parser Expr
equality = chainl1 comparison (infixOp "==" Eq <|> infixOp "!=" Neq)

comparison :: Parser Expr
comparison = chainl1 term (infixOp "<"  Lt <|> infixOp ">" Gt
                       <|> infixOp "<=" Le <|> infixOp ">=" Ge)
```

```
term :: Parser Expr
term = chainl1 factor (infixOp "-" Sub <|> infixOp "+" Add)

factor :: Parser Expr
factor = chainl1 unary (infixOp "*" Mul)

unary :: Parser Expr
unary = Negate <$> (char '-' *> primary) <|> primary

primary :: Parser Expr
primary = parseCallExpr <|> parens expr <|> intExpr <|>
          bool <|> stringExpr <|> parseIdentExpr
```

El código anterior describe cómo está establecida esta precedencia entre los distintos constructores: los operadores con mayor precedencia en la jerarquía son los de igualdad, seguidos de los de comparación, etc. *infixOp* es una parser que reconoce operadores infijos, para lograr extraer los operadores que lo rodean. Por su parte, *chainl1* permite construir parsers que reconocen el encadenamiento de expresiones unarias y binarias.

4.2.6. Tratamiento de errores durante el análisis

Visto cómo se compone un parser, añadir el reporte de errores es algo más complicado de acoplar a la estructura. Por ello se propone que, una vez vistos los fundamentos, se utilice a partir de aquí una librería para cubrir esta necesidad. En este caso se hará uso de megaparsec[1], que es una librería que sigue los preceptos expuestos a lo largo de este capítulo, por lo que es sencillo adaptar el código anteriormente visto. De esta forma, la librería exporta algunas de las definiciones discutidas aquí, dejado al usuario la responsabilidad de describir el reconocimiento léxico:

```
import           Text.Megaparsec
import           Text.Megaparsec.Char
import qualified Text.Megaparsec.Char.Lexer as L

-- a lexeme is every word followed by a space
lexeme :: Parser a -> Parser a
lexeme = L.lexeme blank

-- spaces and comments to skip
blank :: Parser ()
blank = L.space space1 lineCmnt blockCmnt
  where lineCmnt  = L.skipLineComment "//"
        blockCmnt = L.skipBlockComment "/*" "*/"

-- example: a parser that parses integers
natural :: Parser Integer
```

[1] https://hackage.haskell.org/package/megaparsec

```
natural = lexeme L.decimal
```

Debido a que el análisis semántico debe también avisar de errores, es necesario que se anoten en el AST las posiciones de los elementos parseados, para así poder situar el reporte de errores en las posiciones correctas. Por ello el proyecto se acoge a la misma estructura de la librería para reflejar el punto en el que se ha detectado el error. Basta con utilizar el constructor *Parsec* para que se incluya directamente el reporte de errores:

```
type Parser = Parsec Void String
```

4.2.6.1. Ejemplo

A continuación se muestra como ejemplo el caso de un error como conflicto entre el nombre de un identificador y una palabra reservada. Como puede observarse el propósito de este formato de mensaje es el de ubicar adecuadamente al usuario en la parte de código que contiene el error descrito:

```
4:19:
  |
4 |          int string = 2
  |                    ^
keyword "string" cannot be an identifier
```

Para que el parser emita errores definidos por el usuario se puede emplear la funciÓn *fail*, perteneciente a la API de la librería. Así, para capturar el caso de error anterior, hemos de añadir la lógica necesaria para el parser de identificadores:

```
identifier :: Parser String
identifier = do
    ...
    if x `elem` reservedWords
                then fail $ "keyword " ++ show x ++ " cannot be an
                   identifier"
                else return x
```

4.3. Conclusión

En este capítulo se ha realizado un primer acercamiento a la traducción mediante mónadas y su composición para realizar cálculos complejos. También se ha puesto de manifiesto lo simple de estructurar mediante el uso de *alternatives* un traductor que reconozca la sintaxis dada, siendo sencillo de ubicar qué mónada se encarga de procesar qué construcción del lenguaje.

4.3.1. Comparación con el paradigma imperativo

Frente a esta aproximación la programación imperativa suele ofrecer un codificación en primera instancia más intuitiva, por la naturaleza lineal y secuencial del escaneo de caracteres, pero sobre la que es difícil razonar posteriormente. Puede por ejemplo compararse lo desarrollado aquí con un ejemplo imperativo:

```
function parseProgram() {
    program = newProgramASTNode()
    advanceTokens()

    for (currentToken() != EOF_TOKEN) {
        statement = null
        if (currentToken() == LET_TOKEN) {
            statement = parseLetStatement()
        } else if (currentToken() == RETURN_TOKEN) {
            statement = parseReturnStatement()
        } else if (currentToken() == IF_TOKEN) {
            statement = parseIfStatement()
        }

        ...

        if (statement != null) {
            program.Statements.push(statement)
        }
       advanceTokens()
    }
    return program
}
```

Figura 4.2: Psudocódigo de un parser para statements propuesto en [1, p. 36]

Como puede apreciarse, el código imperativo resulta mucho menos conciso y descriptivo, produciendo anidación y acoplamiento de control y lógica propia del dominio. Sin embargo la variante funcional para esta misma función resulta tan breve como explicativa:

```
program :: Parser Program
program = many function
```

4.3.2. Posibles mejoras

Otro de los beneficios de los parsers funcionales es lo sencillo que resulta ampliar las construcciones del lenguaje. De hecho, una de las deficiencias señaladas en la especificación, la ausencia del bucle *for*, es sencilla de cubrir simplemente adjuntando otro parser más, dado que esta construcción deriva de directamente del `while`:

```
for :: Parser [UntypedStatement]
for = do
  reserved "for"

  -- (int i = 0; i < 3; i++)
  pre   <- reserved "(" >> statement
  cond  <- reserved ";" >> expr
  pos   <- reserved ";" >> statement
  stmts <- reserved ")" >> stmtBlock -- { .. }

  return $ pre : [While cond (stmts ++ [pos])]
```

Con una intervención mínima hemos añadido al lenguaje algo de *sintactic sugar* para completarlo. Este Parser traduce un bucle expresado como:

```
for (inicializacion;condicion;incremento)
  <bloque de sentencias>
```

en este bucle equivalente utilizando `while`:

```
inicializacion
while condicion
{
  <bloque de sentencias>
  incremento
}
```

4.3.3. Epílogo

El árbol obtenido en esta primera fase supone la estructura de datos central en el frontend del compilador, ya que como se verá en el siguiente episodio, gracias a ella y a los tipos y operaciones definidos en el lenguaje podemos añadir información semántica adicional para obtener un imagen exacta de nuestro programa.

Capítulo 5

Análisis semántico

Una vez finalizada la fase de parseo del código fuente, se procede al análisis de lo reflejado en el AST. La misión de este análisis es, de forma general, cubrir los siguientes aspectos semánticos:

- Los tipos han de corresponder entre las expresiones y los statements (ej: que una condición en un if corresponde con una expresión booleana).
- Correcto uso de las variables (ej: no declarar una variable más de una vez).
- Uso correcto de las invocaciones de funciones

Como puede observarse, gran parte del ámbito de estas comprobaciones se refiere a los tipos. Por lo demás, el analizador semántico que aquí se presente permite ver en acción y situar conceptos como la tabla de símbolos y los ámbitos (*scopes*), como formas pertinentes para estructurar el análisis.

5.1. Especificación

Tras esta fase, el analizador devolverá el mismo AST, pero con los tipos anotados en las expresiones. En la figura 5.1 podemos ver la comparación entre un AST sin anotaciones y otro con anotaciones de tipo.

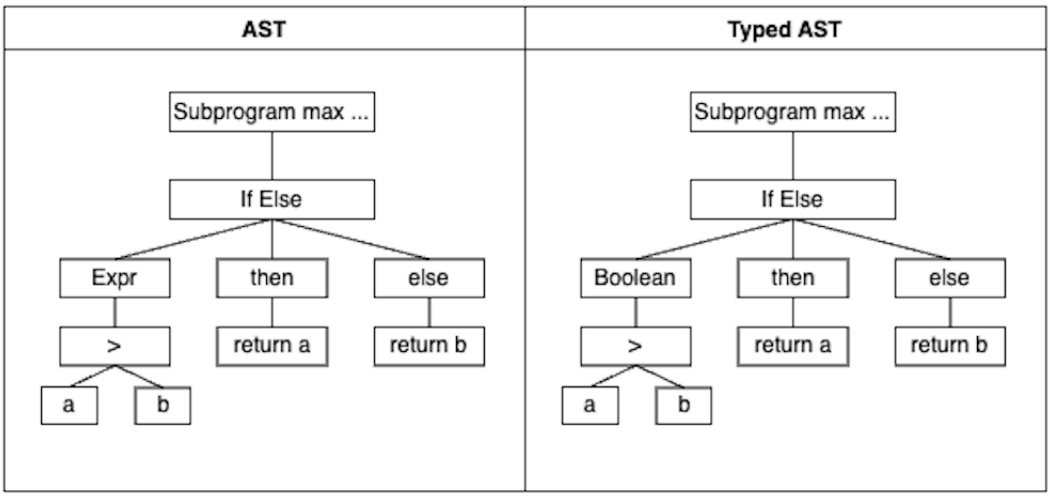


Figura 5.1: Anotación de tipos en el AST

Debido a ello, resulta idóneo refactorizar el AST presentado en la sección 4.1.1 de la fase anterior para permitirle albergar ambos casos: expresiones anotadas y sin anotar. La forma de hacerlo es generalizar la anterior estructura a través de un tipo variable:

```
data Statement expr = If expr [Statement expr]
                    | IfElse expr [Statement expr] [Statement expr]
                    | While expr [Statement expr]
                    | Var SymbolName TypeValue expr -- Declare: int a = 1
                    | Assign String expr            -- Assign:  a = 1
                    | Print expr
                    | CallStmt SymbolName [expr]
                    | Return (Maybe expr)

data TypedExpr = Arith ArithOperation TypedExpr TypedExpr
               | Logic LogicOperation TypedExpr TypedExpr
               | Comp CompOperation TypedExpr TypedExpr
               | Not TypedExpr
               | Negate TypedExpr
               | Lit Literal
               | SymbolRef SymbolName TypeValue
               | Call SymbolName [TypedExpr] TypeValue

type TypedStatement = Statement TypedExpr
```

Existen, de esta manera, dos tipos para expresiones, las que no contienen el tipo anotado, propia de la fase anterior, y las que sí anotan su tipo, reflejada en el código expuesto. Se opta por separarlas para representar mejor, usando el mismo AST, la información semántica que esta fase añade. En adelante el AST se reconstruye sobre las expresiones anotadas, reformulándose de la siguiente manera:

```
type GenericAST expr = [Function expr]

data Function expr = Function{
    funcName :: String,
    params :: [Param],
    ret :: ReturnSignature,
    stmts :: [Statement expr]
}
```

5.2. Elementos del análisis

El análisis semántico requiere de una serie de definiciones y estructuras que garanticen la corrección semántica de los programas. Antes de exponer la creación del analizador semántico, es imprescindible desarrollar todos los elementos necesarios para gestionar la información requerida a la hora de llevar a cabo el análisis semántico propiamente dicho.

5.2.1. Sistema de tipos básicos y operaciones

Como se expuso en la introducción, los tipos básicos en nuestro lenguaje quedan reducidos a tres: enteros, booleanos y cadenas. Cada uno de ellos, salvo los *strings*, cuenta con unas operaciones asociadas

- Los operadores binarios '`+`', '`-`' y '`*`' pertenecen al ámbito de las operaciones aritméticas, y por lo tanto quedan restringidos a los enteros. Así también ocurre lo mismo con la operacion unaria '`-`', para cambiar el signo de una expresión de enteros.

- Los operadores binarios '`&&`' y '`||`' pertenecen al ámbito de las expresiones booleanas, respecto de las operaciones AND y OR, así como el operador unario NOT (!) invierte el valor de una expresión booleana.

5.2.2. Tabla de símbolos

La tabla de símbolos es una estructura de datos muy común en los análisis semánticos de muchos lenguajes de programación. Su objetivo es el de mantener un registro de variables y funciones (símbolos) a medida que avanza el análisis.

El objetivo de este registro en el análisis es el de poder comprobar conforme se avanza en el AST que las referencias concuerdan semánticamente, como se expondrá más adelante.

Su implementación en los lenguajes imperativos que actúan de huesped suele ser una tabla hash, en donde se utiliza el nombre de símbolo como *key*. En Haskell la estructura que mejor responde a este cometido es `Data.Map`, la cual permite, por ser un tipo genérico, describir la estructura con brevedad:

```
data Symbol = Scope SymbolTable
            | Function (Maybe TypeValue) [GAST.Param]
            | Variable TypeValue
            deriving (Show, Eq)

type SymbolTable = Map String Symbol
```

Así pues, una tabla de símbolos establece la relación entre un símbolo, representado mediante una cadena de caracteres, y una estructura `Symbol`, que puede representar una variable con su tipo o una función con el tipo que devuelve (en caso de existir) y sus parámetros.

Pero también se permite un uso recursivo para incluir *scopes* (ámbitos de símbolos) como veremos a continuación.

5.2.3. *Scopes*

Los *scopes* son los espacios en los cuáles permanecen referenciables las variables declaradas. Estos quedan sintácticamente comprendidos entre las definiciones que albergan llaves ({}). De esta manera, forman ámbitos las declaraciones de funciones (dentro de sus cuerpos) y dentro de los cuerpos de los statements *if, if/else* y *while*. El comportamiento de esta estructura es análogo al de una pila de tablas de símbolos:

```
push :: SymbolTable -> SymbolTable
push current = Map.singleton parentKey (Scope current)

pop :: SymbolTable -> SymbolTable
pop current = case fromJust $ lookup parent current of (Scope x) -> x
```

Los ámbitos son estructuras que heredan las variables de los ámbitos superiores. Para reflejar esta herencia, y dado que Haskell no puede referenciar estructuras mediante punteros, cada tabla de símbolos podrá acceder a las variables haciendo una búsqueda recursiva a sus tablas ascendentes, como se refleja en las operaciones para buscar una variable:

```
search :: String -> Analizer (Maybe Symbol)
search name = do
    current <- gets symbolTable

    case Map.lookup name current of
        Nothing -> case Map.lookup parentKey current of
            Just (Scope parent) -> do
                modify (\state -> state {symbolTable = parent})
                var <- search name
                modify (\state -> state {symbolTable = current})
                return var
            _ -> return Nothing
        symbol -> return symbol
```

Ello es posible porque cada *scope* se apila sobre el anterior cada vez que se procede a analizar uno nuevo, y se desapila cuando este se abandona. El modelado de este comportamiento mediante mónadas resulta lo suficientemente descriptivo en la implementación:

```
openScope :: AnalysisState ()
openScope = do
    current <- gets symbolTable
    modify (\state -> state{ symbolTable = Scope.push current })

closeScope :: AnalysisState ()
closeScope = do
    current <- gets symbolTable
    modify (\state -> state{ symbolTable = Scope.pop current })
```

De esta forma, cada apertura de un *scope* nuevo supone el apilamiento de una tabla de símbolos

nueva que hace referencia a la anterior, y el cierre del mismo se realiza mediante el la sustitución de la tabla hija por la del padre. El código anterior ilustra este comportamiento almacenando estas tablas en un *State Monad*, que conforma el estado del análisis semántico general.

5.2.4. Estado en el análisis

El desarrollo de los ámbitos y las tablas de símbolos es uno más de otros efectos paralelos que pueden tener lugar durante el análisis. Es por ello que es necesario, a su vez, mantener una estructura que nos permita ir construyendo las tablas y la pila de *scopes* mediante efectos. La forma más ergonómica de llevar esto a cabo es mediante el uso de la *State Monad*, cuyo tipo se expone aquí:

```
type AnalysisState = State RecordState

data RecordState = RecordState {
    code        :: String,
    returnType  :: Maybe TypeValue,
    symbolTable :: SymbolTable
}
```

El *State Monad*[1] nos permite almacenar un estado y recuperarlo en las funciones basadas en ella. Ya hemos visto algunas de sus primitivas, por ejemplo con *modify* para el apilado y desapilado de *scopes* y en obtención de tabla de símbolos actual con el uso de *gets*.

Los campos a guardar son los relacionados con el análisis que nos ocupa. En primer lugar, es necesario guardar el código fuente de la fase anterior para poder anotar en los errores devueltos la localización, con el objeto de que los errores semánticos tengan el mismo visionado que los sintácticos. Además, es necesario en cada función saber el tipo del retorno, para así poder confrontarlo respecto de las expresiones de retorno que encontremos en la misma. Por último, ubicaremos las tabla de símbolos raíz a partir de la cual partirán las demás.

5.2.5. Errores semánticos

Dentro del dominio del análisis se pueden reconocer, entre otros, algunos de los siguientes casos de error:

- Variables no declaradas anteriormente
- Símbolos ausentes en el ámbito
- Ausencia de *return* en el cuerpo de una función
- Presencia de *return* en el cuerpo de un procedimiento

[1] https://hackage.haskell.org/package/mtl-2.3.1/docs/Control-Monad-State-Lazy.html

La forma de modelar esto es mediante un *sum type* que defina el citado dominio:

```
data SemanticError =  NotEmptyParams
                    | NoMain
                    | TypedMain
                    | NotEmpTyMainReturn
                    | ReturnNeeded
                    | ReturnTypeMissmatch
                    | TooManyArguments
                    | TooFewArguments
                    | InvalidArgType SymbolName
                    | UnknownFunction SymbolName
                    | TypeMissmatch
                    | SymbolAlreadyDefined
                    | SymbolNotFound SymbolName
                    | SymbolNotVariable SymbolName
                    | SymbolNotFunction SymbolName
```

A continuación se explica en qué consiste cada uno de esos errores:

- `NotEmptyParams`: la función main tiene parámetros.
- `NoMain`: no se ha incluído la función main en el programa.
- `TypedMain`: la función main tiene un tipo de retorno.
- `NotEmptyMainReturn`: la función main tiene un contiene un return en su cuerpo.
- `ReturnNeeded`: una función que devuelve un tipo no tiene un return.
- `ReturnTypeMissmatch`: se devuelve un tipo incorrecto en el return de una función.
- `TooManyArguments`: se incluyen más parámetros de los esperados en la llamada a una función.
- `TooFewArguments`: se incluyen menos parámetros de los esperados en la llamada a una función.
- `InvalidArgType`: se usa un tipo incorrecto en uno de los parámetros durante la llamada a una función.
- `UnknownFunction`: se intenta llamar a una función desconocida.
- `TypeMissmatch`: hay un error de uso de tipos (por ejemplo se intentan sumar valores booleanos).
- `SymbolAlreadyDefined`: se intenta definir un símbolo previamente definido.
- `SymbolNotFound`: se intenta utilizar un símbolo que no se ha definido con anterioridad.
- `SymbolNotVariable`: se intenta utilizar como variable un símbolo que no se ha definido como variable.
- `SymbolNotFunction`: se intenta utilizar como función un símbolo que no se ha definido como función.

5.2.5.1. Reporte de errores

Como se ha advertido anteriormente, el reporte de errores sigue el mismo uso que proviene de la fase anterior, y que a su vez se acopla a la API de la librería *megaparsec*. Se ha adoptado así para mantener la cohesión visual entre ambas fases. Felizmente, la librería provee de los tipos necesarios para esta tarea y basta con adaptar el reporte a lo descrito en su documentación [2].

```
type SemanticReport = ParseErrorBundle String SemanticError
```

El constructor ParseError[3] permite en nuestro caso de uso definir las posibilidades de error a partir del tipo SemanticError, representado como String cuando se le muestre al usuario. Este tipo nos brinda la posibilidad de crear reportes para describir un error y su ubicación en el código fuente:

```
newReport :: String -> Int -> Error -> SemanticReport
newReport src pos e = ParseErrorBundle {
    bundleErrors = fromList [FancyError pos (Set.singleton $
       ErrorCustom e)]
  , bundlePosState = PosState { pstateInput = src }
  }
```

Ello nos permite ubicar los casos de error sin ser demasiado invasivos en el resto del código:

```
2:12:
  |
2 |          a = 2 + true
  |                  ^
types doesn't match
```

5.3. Proceso de Análisis

Una vez dispuestos los elementos que conforman el dominio del análisis vamos a proceder a relatar el desarrollo del mismo.

El análisis semántico se va a dividir en dos partes: el que concierne a la función *main* y el que concierte al resto de las funciones definidas por el usuario. Ello se debe a que *main* contiene una semántica *ad hoc* para garantizar la correcta ejecución esperada por el entorno de ejecución.

[2] https://hackage.haskell.org/package/megaparsec-9.3.0/docs/Text-Megaparsec-Error.html
[3] https://hackage.haskell.org/package/megaparsec-9.3.1/docs/Text-Megaparsec-Error.html#t:ParseError

5.3.1. Función principal

Para la función *main* se comprueba que esté presente (si no es así se reporta un error `NoMain`), que no tenga parámetros declarados (lo que provocaría un error `NotEmptyParams`) y que no tenga anotado ningún tipo de dato para retornar (que sería un error `TypedMain`). El código que realiza esto es el siguiente:

```
checkMain :: UntypedAST -> Analizer ()
checkMain ast = case find (funcName >>> (== "main")) ast of
    Nothing -> reportAt 0 NoMain
    Just main -> do
      unless (null (params main)) $ reportAt (parsePos (head (params
         main))) NotEmptyParams
      when (isJust ((ret >>> typeSignature) main)) $ reportAt ((ret
         >>> pos) main) TypedMain
```

5.3.2. Análisis de subprogramas

El análisis del resto de funciones obedece a la naturaleza recursiva del AST. De esta forma, en el análisis de cada función está contenido el análisis los *statements* que forman su cuerpo, y dentro de cada uno de ellos el correspondiente al de sus expresiones.

El análisis de un subprograma conlleva registrar el tipo del retorno en primer lugar. Como vamos a analizar el cuerpo de la función hemos de abrir un *scope*, añadirle los parámetros para que los tenga en cuenta cuando analice las variables locales y proceder al análisis de los statements que conforman el cuerpo:

```
checkSubprogram :: UntypedSubprogram -> Analizer TypedSubprogram
checkSubprogram (Subprogram funcName params returnType stmts) = do
    lift $ setReturn (typeSignature returnType)

    lift openScope

    insert params

    typedStmts <- traverse checkStatement stmts

    lift closeScope

    return $ Subprogram funcName params returnT typedStmts
```

Como cada *statement* está formado por unos campos en concreto, es necesario describir el análisis de cada uno por separado. Se muestran aquí sólo los más representativos. Para las variables se divide su análisis entre las de declaraciones, que insertan la variable en la tabla de símbolos, y las asignaciones, que se encargan únicamente de comprobar que se respete el tipo de la asignación con el de la variable declarada previamente. En el caso de bucles y control de

flujo se comprueba que la condición corresponda al tipo *bool* y se procede a realizar una análisis recursivo por el cuerpo.

```
checkStatement :: UntypedStatement -> Analizer TypedStatement
checkStatement stmt = case stmt of
    (Var name type_ expr pos) -> do
        typedExpr <- typeExpr expr
        match (typedExpr, type_)
        insert name (Symbol.Variable type_)

        return $ Var name type_ typedExpr pos

    (Assign name expr pos) -> do
        found <- search name
        case found of
            Just (Symbol.Variable type_) -> do
                typedExpr <- typeExpr expr
                match (typedExpr, type_)

                return $ Assign name e pos
            Just _  -> report (SymbolNotVariable name)
            Nothing -> report (SymbolNotFound name)

    (While cond then_ pos) -> do
        typedExpr <- typeExpr cond
        match (typedExpr, BoolType)
        stmts <- checkScope then_

        return $ While typedExpr stmts pos

    ...
```

Dentro del *while* hemos visto una llamada a una función que se ocupa de inspeccionar el cuerpo del mismo: *checkScope*. Esta función se encarga además de abrir y cerrar el *scope*. También se utiliza dentro del análisis de los *if* e *if/else*:

```
checkScope :: [UntypedStatement] -> Analizer [TypedStatement]
checkScope stmts = do
    lift openScope
    typedStmts <- traverse checkStatement stmts
    lift closeScope
    return typedStmts
```

5.3.3. Análisis de tipos

Con el mismo patrón se recorre cada una de las expresiones a analizar. Tanto en esta función como en la anterior se usa la función **traverse** para iterar una mónada a través de una colección

de entradas. El fin de esta función es anotar en una expresión su tipo para poder, como se ha visto anteriormente, confrontarlo cuando se hace referencia a él, bien sea para un paso de argumentos en una llamada, una asignación, etc.

```
typeExpr :: UntypedExpr -> Analizer TypedAST.TypedExpr
typeExpr (Arith op _ left right) = do
    typedLeft <- typeExpr left
    match (typedLeft, IntType)

    typedRight <- typeExpr right
    match (typedRight, IntType)

    return $ TypedAST.Arith op typedLeft typedRight

...
```

5.4. Conclusión

En este capítulo, además de ver el uso de mónadas para otro cometido distinto al del parseo, se ha hecho uso de la recursión para el recorrido del AST. Al ser Haskell un lenguaje funcional, este provee de formas de definir estructuras recursivas que luego resultan sencillas de manejar gracias al *pattern matching*, como se ha visto en la función *checkStatement*.

5.4.1. Posibles mejoras

Una de las ampliaciones que podrían realizarse es la inclusión de tipos definidos por el usuario, especialmente los *structs*. Para ello podemos utilizar una estructura análoga a la tabla de símbolos, la **tabla de tipos**, adjunta a un *scope* particular, para reflejar los tipos definidos por el usuario. En el caso del tipo *struct* estaremos valiéndonos de la recursión para poder albergar las mismas posibilidades anidadas, de forma que el *struct* sea, en resumidas cuentas, otra tabla de tipos.

```
data DefinedType = Alias DefinedType -- type rename
                 | Struct TypesTable -- struct type (struct{bool, int}
                 | Basic TypeValue   -- basic types (bool, int ...)
                 | Array DefinedType -- array type ([]int)
                 | Parent TypesTable -- type table of the parent scope

type TypesTable = Map String DefinedType
```

Con este simple bosquejo damos cabida a que el usuario pueda declarar tipos en su programa. De nuevo la ergonomía de *Data* en Haskell y el uso de la recursividad nos permite representar construcciones complejas para el lenguaje propuesto. Por otro lado, el uso de esta nueva tabla es casi idéntica a la expuesta para los símbolos, por lo que este añadido al lenguaje apenas

representa mayor dificultad.

5.4.2. Comparación con el paradigma imperativo

El recorrido recursivo del que se ha hecho uso en este capítulo es posible aplicarlo en un lenguaje imperativo también. Sin embargo, para la identificación de cada nodo, sin contar con *pattern matching*, nos vemos abocados a emplear opciones que hacen el código demasiado descriptivo:

- **Aplicando condiciones**, por ejemplo mediante instrucciones *switch*:

```
function traverseAST(node)  {
    ...

    switch(node) {
        case IfElse {
            ...
        }

        case Assign {
            ...
        }

        ...
    }
}
```

- Utilizando **herencia** o **polimorfismo**: haciendo uso del paradigma orientado a objetos se puede modelar el posible tipo de cada nodo:

```
class AST {}

class Assign extends Node {}

class WhileLoop extends Node {}

function traverseAST(node) {
  if (node instanceof Assign) {
    ...
  } else if (node instanceof WhileLoop) {
    ...
  } ...
}
```

Frente a ello, el código que hemos mostrado en este capítulo permite describir fácilmente la estructura en árbol sin necesidad de dividir el concepto en entidades dispersas (esto es, las clases y sus relaciones). Además, el *pattern matching* nos ahorrar el describir el reconocimiento de cada tipo de nodo, dejando a Haskell reproducirlo internamente.

5.4.3. Epílogo

Una vez el AST esté anotado con los tipos de cada expresión, haremos uso de él para transformar esta estructura en árbol en una estructura de datos lineal que permita ser ejecutada unívocamente de forma secuencial. En este contexto será necesario el uso de mónadas más avanzadas para poder ir reflejando esta transformación a la vez que seguimos realizando cálculos parciales.

Capítulo 6

Síntesis interna. Generación de código IR

Una vez el frontend concluya la fase de análisis, se entrega al backend el AST con anotaciones semánticas para que este lleve a cabo la fase de síntesis. Previamente a la elaboración del código ensamblador se usa una forma de código intermedio conocida como *Intermediate Representation* (IR). En la figura 6.1 se puede ver la transformación que se consigue en esta fase.

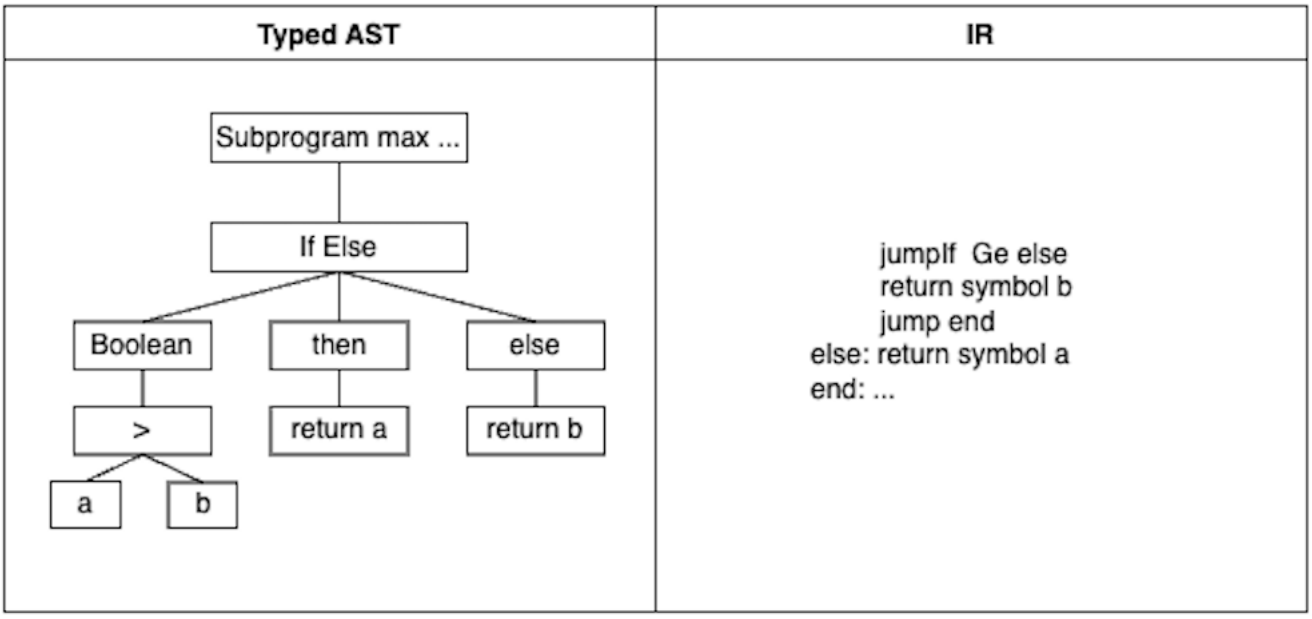


Figura 6.1: Transformación del AST en código IR

6.1. Especificación

La representación intermedia es una traducción del AST que permite al backend trabajar en los siguientes objetivos:

- Sintetizar el programa fuente de una forma en árbol a una lineal
- Optimización (traducir de IR a IR reducido)
- Obtener una única representación para luego traducir a cada código objeto: ARM, x86 ...

6.1.1. Operadores

El diseño de los operadores del IR debe abarcar el conjunto de posibilidades de los tipos definidos en el lenguaje, por ello se discierne entre lo que son operadores de símbolo (variables), enteros, cadenas de texto y temporales donde ubicar los cálculos parciales que se realizan.

```
data Operator = Symbol String TypeValue
              | Number Int
              | ConstString String
              | Temp Temporal
```

Los valores booleanos se codifican como enteros utilizando el anterior constructor Number. Esto es debido a que, contraintuitivamente, las arquitecturas de procesadores no realizan operaciones a nivel de bit (salvo para desplazamientos), sino a nivel de palabra. De hecho, un lenguaje clásico como C no tiene el tipo *bool* como nativo, sino que para ello media una librería dependiente de cada sistema operativo (stdbool.h), que los codifica en ensamblador a través de enteros.

6.1.2. Instrucciones

El criterio a la hora de diseñar las operaciones en el IR es poder reproducir el comportamiento secuencial de los procesadores. Es en esta fase donde la estructura en árbol se transforma en una estructura lineal. Esto tiene su manifestación más fundamental en que el compilador se deshace de los bucles y condicionales como estructuras, y las descompone en código con saltos y etiquetas.

A continuación se pueden ver las instrucciones de nuestro lenguaje IR:

```
data Operation = Var String TypeValue
               | Copy Operator Operator
               | Arithmetic ArithOperation Operator Operator Operator
               | Compare Operator Operator
               | JumpIf CompOperation Label
               | Not Operator Operator
               | Negate Operator Operator
               | Jump Label
               | Call [Operator] String
               | CallRet Operator [Operator] String
               | Return (Maybe Operator)
               | Print Operator
               | None
```

De todas los tipos de operaciones, es conveniente comentar algunas:

- Las operaciones *Arithmetic* pueden ser, como se exponía en la especificación del lenguaje, *Add*, *Sub* y *Mul*, y se indican tres operadores: el primero para guardar el resultado y otros dos para los operandos.

- *Compare* compara dos operandos
- *Jump* y *JumpIF* implica un salto incondicional y condicionales, dependiendo del resultado anterior (el resultado de estas operaciones de comparación se registra internamente en el procesador en la ejecución, no siendo necesario en esta fase por ello indicar ningún temporal para almacenarlo). *Call* y *CallRet* realizan llamadas, con la diferencia de que la segunda espera retornar un valor, que se guarda en el primer *Operator*

6.2. Traducción a IR

La traducción a IR se realiza mediante el recorrido del AST en preorden, generando en paralelo las instrucciones. Para esto último se emplea una *Monad Writer*. Esta mónada cuenta únicamente con la función *tell*, que se encarga de añadir internamente el código como un listado de instrucciones.

6.2.1. Estado en el traductor

Como en el caso del análisis, los traductores necesitan almacenar cierta información conforme avanza la traducción. Por ello nuestra mónada Translator va a incluir también un *Monad State* dentro de un *Monad Transformer*. Este tipo de mónada permite introducir la funcionalidad de una mónada dentro de otra. En este caso, como se requiere incluir un estado dentro de un *Monad Writer*, emplearemos el *WriterT*:

```
type Translator = WriterT [Instruction] TranslatorState
```

La información del estado está relacionada con los dos recursos principales de esta fase: los temporales y las *labels*. Las primitivas para la generación de unos y otras están soportadas en el uso de los contadores para evitar la colisión de nombres entre ellas:

```
data Counters = Counters {
    labels :: Int,
    temporals :: Int,
}

type TranslatorState = State Counters
```

6.2.1.1. Temporales

Los temporales son referencias al resultado de las expresiones. Sirven para conectar unas expresiones con otras y para conocer, de forma temporalmente virtual, la localización de un determinado resultado. Para acceder al estado interno del *Monad Writer* hacemos uso de la función *lift*:

```
newTemp :: TranslatorState Temporal
newTemp = do
    lift $ modify (\s -> s { temporals = temporals s + 1 })

    ti <- lift $ gets temporals

    return $ Temporal ti
```

6.2.1.2. *Labels*

De forma también virtual, las etiquetas sirven para apuntar a direcciones de código referenciables para las instrucciones de salto, haciendo posible la traducción de bucles y condicionales. De hecho, este recurso provee del formato para representar las instrucciones:

```
data Instruction = Instruction {
    label :: Label,
    op :: Operation
}
```

El formato de las *Labels* producida en este proyecto se adhiere al del Clang de ARMv7, que ha sido adoptado por familiaridad. El primer índice marca el de las funciones (global) y el segundo el interno a cada una (local para bucles y condicionales).

```
prefix :: String
prefix = ".LBB" -- e.g.: .LBB0_1
```

La función para obtenerla sólo requiere de la modificación de los contadores antes mencionados:

```
newtype Label = Label String deriving (Eq)

getLabel :: TranslatorState Label
getLabel = do
    lift $ modify (\s -> s { labels = labels s + 1 })

    funcIndex <- lift $ gets funcs
    labelIndex <- lift $ gets labels

    return $ newLabel funcIndex labelIndex
```

Estas etiquetas serán posteriormente transformadas en direcciones de memoria por el loader en el entorno de ejecución, lo que hace posible el funcionamiento de saltos entre instrucciones, al estar estas almacenadas en memoria.

6.2.2. Traducción de statements

Lo primero a destacar de la especificación de la representación interna es que se desecha cualquier estructura anidada, por lo que no tiene sentido hablar a partir de aquí de contextos tales como los statements y las expresiones. Es en esta frontera donde la estructura lineal, la lista de instrucciones, pierde toda información de las construcciones propias del lenguaje fuente.

La traducción de bucles y condicionales requiere precisamente del almacenado de las etiquetas, por lo que algunas de las operaciones del traductor son:

```
setLabels :: Label -> Label -> Translator ()
setLabels thenL elseL = lift $ modify (\ctx -> ctx{condLabels = (thenL
   , elseL)})

thenLabel :: Translator Label
thenLabel = lift $ gets (condLabels >>> fst)

elseLabel :: Translator Label
elseLabel = lift $ gets (condLabels >>> snd)
```

La traducción requiere descomponer los condicionales y bucles a su forma secuencial con saltos.

En el siguiente código podemos ver la traducción de un bloque if/then, en lo primero es generar las etiquetas para poder saltar al bloque del then, o si no se cumple la condición saltaremos directamente al final de este bloque para continuar la ejecución. Estás *labels* se guardan para poder ser usadas por el traductor en el código que comprueba las condiciones en el siguiente paso. Una vez se crea este código de comprobación se se crea el punto del *then*, luego el traduce el propio bloque *then*, y después se establece el punto del *else*.

```
translate :: TypedStatement -> Compiler ()
translate (If cond then_ _) = do
    thenL <- newLabel
    endL  <- newLabel

    saveLabels thenL endL

    check cond                  -- [checks condition code]
    set thenL                   -- then:
    traverse_ translate then_   --      [then code]
    set endL                    -- end:
```

En la figura 6.2 se puede ver el proceso sufrido por un bloque if/else desde su código fuente hasta la obtención del código intermedio.

El código a continuación muestra la traducción de un bucle While, para el cual se generan dos etiquetas: una para identificar el comienzo del código IR del cuerpo del bucle y otra para identificar el comienzo del código IR que evalúa la expresión del bucle.

```
translate (While cond then_ _) = do
```

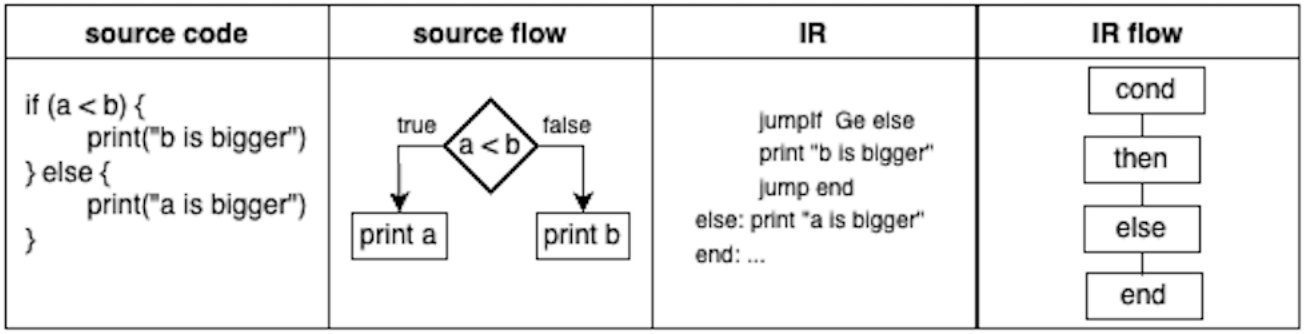


Figura 6.2: Traducción de un bloque if/else

```
test_ <- newLabel
loopL <- newLabel
endL  <- newLabel

saveLabels loopL endL

jumpTo test_                -- jump to test
set loopL                   -- loop:

traverse_ translate then_       -- [loop code]

set test_                   -- test:
check cond                  --      [checks condition code]
jumpTo loopL                -- jump to loop
set endL                    -- end:
```

En la figura 6.3 se puede ver todo el proceso por el que pasa un bucle While desde su código fuente hasta la obtención del código intermedio.

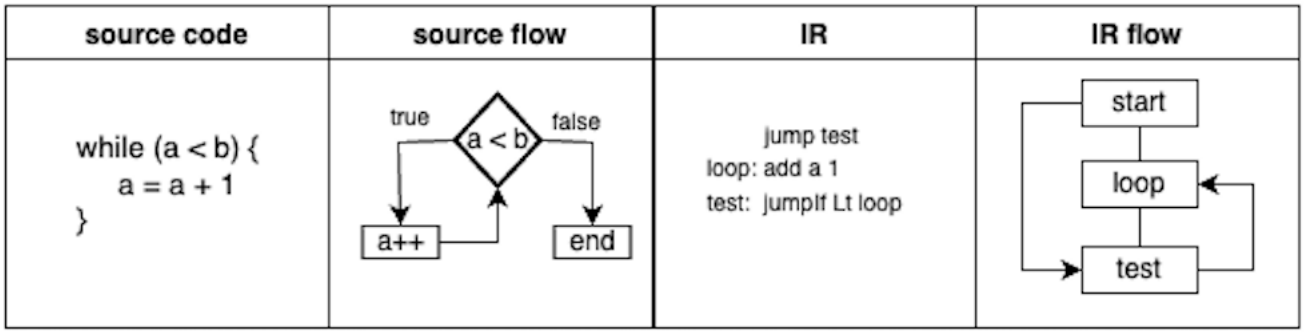


Figura 6.3: Traducción de un bloque While

6.2.3. Traducción de expresiones

Las expresiones en su traducción se evalúan con la mediación de temporales, apuntando los temporales resultantes de la evaluación de los operandos y la creación de un tercero para el

actual:

```
eval :: TypedExpr -> Translator Operator
eval (Arith op expr1 expr2) = do
    temp1 <- eval expr1
    temp2 <- eval expr2

    resultTemp <- newTemp

    unlabeled $ IR.Arithmetic op resultTemp temp1 temp2

    return resultTemp
```

En este sentido, una expresión tal como 2 + 2, emite un temporal para ubicar el resultado y poder enlazarlo con otras expresiones o como forma de asignación a una variable. De esta forma, el temporal podrá ser utilizado para enlazar esta operación con otra, pudiendo con ello representar expresiones más complejas:

```
Add temp1 2     2
Sub temp2 temp1 1
```

6.2.3.1. Expresiones booleanas

La traducción de las expresiones booleanas merece una mención especial debido a que esta no se realiza siguiendo un descenso en preorden como se venía haciendo hasta ahora en el AST, sino que la traducción pasa a realizarse dentro del contexto de un statement de forma secuencial. Esto radica en que el comportamiento con las expresiones booleanas no se realiza mediante operaciones que devuelven resultados del mismo tipo, como ocurre por ejemplo con las operaciones aritméticas, sino que en la arquitectura de procesadores siempre se realiza mediante etiquetas y saltos [5, p.414].

En adelante se expone la traducción de este tipo de expresiones partiendo de la función *check* anteriormente vista en la traducción de los statements. Se distingue en esta traducción entre dos tipos de predicados que envuelven en un contexto particular la traducción como se verá más adelante:

- **Conjunciones**, en las cuales todos los elementos de la misma han de ser resueltos como verdaderos
- **Disyunciones**, en las que al menos uno de los términos ha de ser verdadero.

Según estos predicados la traducción que *check* va a llevar a cabo va a disponer las etiquetas y saltos de forma distinta, o incluso va a producir cortocircuitos *adhoc* para determinadas situaciones. Esto no se produce con un criterio de optimización, sino que la propia transformación a estructura secuencial requiere tratar este tipo de expresiones no como una estructura anidada,

sino como un grafo (cíclico en el caso de los bucles) que sólo dispone del salto como método de enlace entre bloques.

Así pues para traducir estas expresiones hemos de partir de contextos basados en el álgebra de Boole para la operación **and** o la operación **or**:

```
check :: TypedExpr -> Translator ()
check (Logic And l r) = conjuntion l >> conjuntion r -- l and r
check (Logic Or  l r) = disjuntion l >> conjuntion r -- l or r
check cond            = conjuntion cond
```

Como puede observarse, en la operación *or* se permite, por su naturaleza disyuntiva, que el primer término pueda resultar negativo para en tal caso disponer la comprobación del segundo como si tuviese que ser necesariamente verdadero.

Dentro de una conjunción cada término ha de evaluarse en el entorno de ejecución como verdadero. Con arreglo a esto los literales establecen un comportamiento en el cuál, de estar presentes, un *True* simplemente da paso a las siguientes comprobación, y un *False* cortocircuita directamente al final del statement, eludiendo el cuerpo del condicional (ya sea este un *if*, *while*, etc.) La negación requiere, para su correcto funcionamiento, invertir las etiquetas y la operación, para evaluar por el contexto contrario y derivar el control al bloque de código contrario al que se disponía. Respecto de las comparaciones, se hacen respecto del valor de la expresión a evaluar, si esta no es verdadera (es cierta su inversa) el código salta elude el bloque *then*:

```
conjuntion, disjuntion :: TypedExpr -> Translator () -- and, or
conjuntion (Lit (BoolVal True))  = return () -- static true: skip
   check
conjuntion (Lit (BoolVal False)) = jumpTo =<< elseLabel -- static
   false: goto else
conjuntion (Not expr) = swapLabels >> disjuntion expr >> swapLabels
conjuntion (Comp op left right)  = do
    e1 <- eval left
    e2 <- eval right

    elseL <- elseLabel

    Compare e1 e2
    JumpIf (inverse op) elseL

conjuntion expr = do -- for any other case, handle it as a if
   statement
    tmp   <- eval expr
    elseL <- elseLabel

    Compare tmp true
    JumpIf Ne elseL
```

Con objeto de ilustrar el funcionamiento de la fórmula anterior se propone el siguiente ejemplo. Ante el siguiente código código de control de flujo:

```
if (2 > 1) {
    print("a es mayor")
}
```

el código generado en IR sería el siguiente, de forma simplificada:

```
    Comp 2 1
    JumpIf Le end
    Print "a es mayor"
end: ...
```

Nótese cómo la forma de abrir la secuencia a la ejecución condicional pasa por invertir la condición en la comparación para en caso de cumplirse salte y evite el código del bloque mediante el salto a la etiqueta final. Esta forma de hacerlo permite preservar el recorrido en preorden anteriormente citado, para desplegar secuencialmente el código y, además, hacer reconocible este código respecto de su representación anterior.

La operación **or** tiene, en tanto que es una disyunción, un comportamiento inverso al expuesto en el caso **and**:

```
disjuntion (Lit (BoolVal True))  = jumpTo =<< thenLabel -- static true
   : goto then
disjuntion (Lit (BoolVal False)) = return ()            -- static
   false: skip check
disjuntion (Not expr)            = swapLabels >> conjuntion expr >>
   swapLabels
disjuntion (Comp op left right)  = do
    e1 <- eval left
    e2 <- eval right

    then_ <- thenLabel

    Compare e1 e2
    JumpIf op then_

disjuntion expr = do
    tmp   <- eval expr
    then_ <- thenLabel

    Compare tmp true
    JumpIf Eq then_
```

En el caso de literales, el valor estático que cortociruita es *True*, que directamente desemboca en el bloque *then*. La negación realiza el mismo proceso que en el caso de la conjunción, y la comparación no requiere invertir la operación y realiza el salto al bloque *then* si esta es verdadera.

Podemos ver el comportamiento del cortocircuitaje descrito con el siguiente ejemplo:

```
if (... || True || (a > b)) {
    print("cierto")
}
```

En este caso, al ser una disyunción *or*, al encontrar un True en medio de la expresión, a partir de ese momento esta queda evaluada estáticamente como verdadera, desechando el examen del resto de la condición, y por tanto dando paso directamente al bloque *then*:

```
      ...
      Jump then
then: Print "a es mayor"
end:  ...
```

Como puede observarse, tanto el contexto *and* como el *or* pueden ser vistos como predicados estrictos (todos los valores tienen que ser *true*) o permisivos ("se permite"que el primer operando sea falso para evaluar el siguiente), respectivamente.

Por otra parte, la traducción de las asignaciones a una variable funciona de la misma forma, salvo que aquí sí se hace uso de temporales para trasladar el valor booleano:

```
tmp <- eval expr
Copy (Symbol name (typeOf expr)) tmp
```

6.3. Conclusión

La traducción a IR permite desacoplar la sintaxis del lenguaje de su semántica, independizándolo tanto de la gramática del mismo como de su traducción a ensamblador. De esta forma, no sólo el compilador puede escoger entre varios ensambladores destino a partir de una misma representación interna, sino que, por efecto de ello, este lenguaje suele estar regido por razones de diseño propias.

Es por ello que no solamente uno puede encontrar propuestas que supongan una convergencia en el diseño de esta fase, como es *SSA Form*[1], sino que existen especificaciones de terceros para poder derivar el backend completamente a partir de ella, como es el caso del IR de LLVM[2].

6.3.1. Posibles mejoras

La primera mejora que salta a la vista en cuanto a la arquitectura es añadir una fase más de optimización para el IR. A grandes rasgos, esto supone escribir las optimizaciones como

[1] SSA Form (*Static Single-Assignment Form* es una característica de una IR en la que cada asignación de una variable representa una versión distinta de la misma, lo que supone una ayuda para la optimización del código)

[2] https://llvm.org/docs/LangRef.html

heurísticas que reconociesen determinados patrones de código y se sustituyese por un target de mejora.

En cuanto al propio lenguaje, como se expuso en el capítulo de sintaxis, este carece de mucha variedad respecto a los bucles, no solamente respecto a su declaración sino al control de los mismos también. No obstante, construcciones como *break* puede modelarse fácilmente en esta fase mediante con la traducción propuesta:

```
translate :: TypedStatement -> Translator ()
translate Break = jumpTo =<< elseLabel
...
```

Una vez más, vemos como modelar la traducción mediante mónadas permite combinar tener un contexto y accionar efectos secundarios, en este caso, leer el estado de las etiquetas y emitir un salto.

6.3.2. Comparativa con el enfoque imperativo

Por contra, la solución imperativa a esto conllevaría varios niveles de abstracción:

```
class IrTransmitter {

    translate(statement) {
        switch(statement):
        ...

        case Break {
            Label end = this.LabelsVault.getEndLabel();

            Instr goto = this.Translator.NewGoto(end);

            translator.write(goto);
        }
    }
}
```

La implementación imperativa necesita de varias entidades implicadas:

- una para emitir código
- otra para la especificación IR
- y otra generar y almacenar etiquetas

La forma de evitar acoplamiento de dominio entre ellas estaría dado por interfaces que simplemente describiesen las acciones. En el paradigma funcional, por el contrario, la relación entre

todas estas acciones se establece por la entidad que media entre ellas, la etiqueta. No solamente nos permite un código más limpio y descriptivo sino que además permite aislar de forma natural los distintos dominios y unirlos en acciones (mónadas).

6.3.3. Epílogo

En la primera fase de traducción se pone de manifiesto la necesidad de usar una forma de mónada más avanzada para poder transformar una estructura de una naturaleza recursiva en una lineal, teniendo que almacenar un estado y hacer cálculos parciales a la vez que se va acumulando internamente las partes de la estructura lineal (el código IR). Es en este punto donde al acercarnos a la forma más imperativa de código que existe (el código ensamblador) la utilidad para traducir ha de modelarse de forma más imperativa, obedeciendo completamente ya al modelo de Von Neumann.

Capítulo 7

Generación de código ensamblador

La producción de código intermedio facilita la traducción a código ensamblador al proveer de construcciones cuya traducción es ya casi directa. El código ensamblador utilizado aquí es el de la arquitectura ARMv7, tal y como se muestra en la figura 7.1, pero la traducción aquí propuesta es lo suficientemente representativa para aplicarse a otro conjunto de instrucciones como x86 o MIPS. Aunque la estructura del mismo sea casi idéntica a la expuesta en la fase anterior, se aprovecha para poner de relieve algunos aspectos comcretos de ARM, así como problemas generales a todos los compiladores en esta fase.

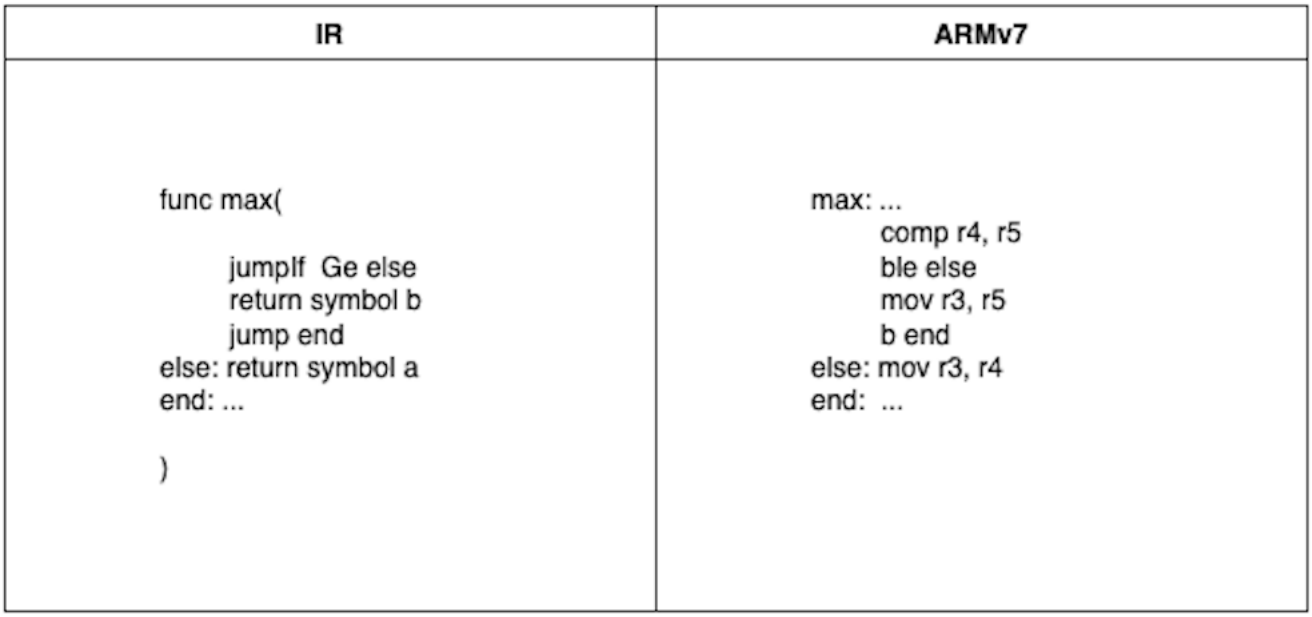


Figura 7.1: Traducción del código IR en código ARMv7

Como en el capítulo anterior, la traducción la va a llevar a cabo un traductor implementado sobre una MonadTransformer Writer (WriterT). Más adelante se expondrá el estado interno de la misma así como las primitivas que a partir de él se desarrollan para expresar construcciones semánticas del lenguaje.

```
type Translator = WriterT [Instruction] TranslatorState

asm :: Instruction -> Translator ()
asm op = tell [op]
```

7.1. Especificación

Para la correcta comprensión de la traducción que se va a llevar a cabo se expone brevemente la especificación del lenguaje ARM que se emplea.

7.1.1. Almacenamiento

La naturaleza secuencial de la ejecución del código ensamblador obliga a definir una estrategia de guardado de datos que tenga presente el carácter volátil de las espacios de almacenamiento entre contextos, como son los temporales dentro de las expresiones, o las variables albergadas en las funciones. En este sentido, se procede a describir los recursos de almacenamiento primero para luego definir la propuesta de guardado en cada contexto.

Ha de tenerse en cuenta también el entorno de ejecución en esta fase final de la compilación. Los tipos se traducen a tamaños expresados en números de palabras de máquina. Una palabra de máquina es un conjunto de bytes que son capaces tanto de almacenar un dato como de direccionar una posición en memoria. Nuestro entorno de ejecución será una máquina ARM con un ancho de palabra de 32 bits (8 x 4 = 32 bits).

7.1.1.1. Registros

Con el objeto de realizar operaciones internas, los procesadores cuentan con registros, pequeños espacios de memoria del tamaño del ancho de palabra de la arquitectura, los cuales facilitan el manejo de datos entre operaciones sin tener que mediar acceso a memoria RAM. Los registros utilizados aquí son los siguiente:

```
data Register = R0 | R1 | R2 | R3 | R4 | R5 | R6
              | R7 | R8 | R9 | R10 | FP | SP | LR
```

Dependiendo de su propósito pueden a su vez dividirse en:

- R0-10 (salvo R7): propósito general.
- R7: Alberga el número de llamada al sistema.
- R11: Frame Pointer (FP), marca el inicio del frame actual en memoria.
- R12: Intra-Procedure Call, no utilizado aquí.
- R13: Stack Pointer (SP), marca el final del fram actual, y el final de la pila.
- R14: Link Register: almacena la localización a la que regresar cuando la actual función retorne.
- R15: Program Counter (PC): guarda el contador del programa. En nuestra traducción no es necesario.

Los registros son el recurso de almacenamiento que ofrece mayor velocidad, a expensas de su limitado número y capacidad. Debido a ello una tarea fundamental en la traducción es su correcto reparto entre variables locales y temporales.

La asignación de registros es un problema NP-completo, bien estudiado y afrontado comúnmente o bien mediante el coloreado de grafos [10] o mediante un algoritmo voraz. La primera opción posibilita el encontrar repartos óptimos bajo determinadas heurísticas, pero un desarrollo de este algoritmo queda fuera del marco de este proyecto por su complejidad. Como alternativa se adopta un algoritmo voraz que realiza, en primer lugar, un particionado del conjunto de registros:

- Registros para el **cálculo de expresiones**: R0, R1, R2. Preasignamos tres registros para esto dado que los operadores binarios tienen 2 operadores + 1 para almacenar el resultado. Una posible mejora es sobrescribir el resultado en uno de los registros usados para operadores (por ejemplo: add R0 R0 R1).
- Registro para **retorno**: (R3)
- Registros para el **paso de argumentos** como parámetros: R4, R5, R6, R8. El protocolo para el paso de argumentos entre funciones suele hacer uso de los cuatro primeros registros, pero aquí para que no se produzcan colisiones entre registros con el espacio dedicado a cálculo de expresiones, optamos por situar los argumentos en otro segmento.

Para el almacenamiento de variables se empleará siempre acceso a memoria. Esta es la opción menos óptima, debido a que en condiciones de transitividad entre operaciones es posible destinar un registro disponible a almacenar el contenido de una variable. Para ello habría que desarrollar un reparto de registros por función con el algoritmo de coloreado de grafos antes mencionado. En esta implementación optamos por una exposición más sencilla en donde el conjunto de variables siempre va destinado a memoria, a expensas de una menor velocidad de ejecución que para el objetivo de este trabajo no resulta clave.

7.1.1.2. Pila

La gestión de memoria más clásica en compiladores suele distinguir entre dos áreas: la pila y el montículo. Para la gestión de datos del tipo que nos ocupa (cuyo tamaño es conocido en tiempo de compilación) no es necesario emplear la segunda opción, así como la exploración del recolección de basura como parte ineludible de su manejo queda fuera del ámbito de este trabajo. Por ello, nos ceñiremos al diseño de la pila para todo almacenamiento en memoria.

El mecanismo de apilamiento de frames en cada función tiene una gran tradición y los espacios de memoria definidos en su interior están siempre presentes en la inmensa mayoría lenguajes de programación, entre los cuales cabe distinguir:

- **parametros**: en las primeras posiciones del frame se ubican los parámetros de la función
- **return value**: espacio para almacenar el valor de retorno de la función

- **control**: frame pointer de la función llamante para devolverle el control una vez finalice la llamada
- **access**: para acceso a variables globales. No se utiliza en nuestro lenguaje
- **state (lr)**: stack pointer de la función llamante
- espacio para **variables locales**

En la figura 7.2 se puede ver la estructura de los frames que se ha seguido en nuestro caso.

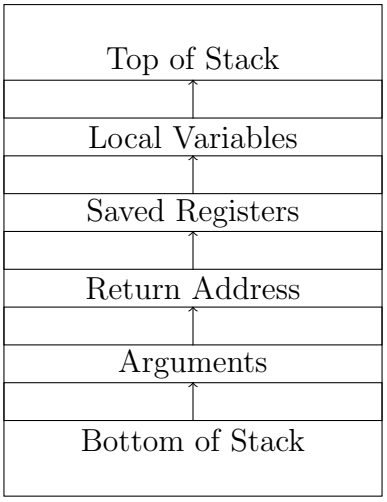


Figura 7.2: Estructura de los frames en la pila

7.1.2. Branching

Al igual que otras arquitecturas, ARM cuenta con instrucciones de salto condicionales e incondicionales. la diferencia con x86 en este caso reside en el nombre, llamándose *branch* en vez de *jump*. Para su uso se han definido algunas primitivas como las siguientes:

```
branchTo :: String -> Translator ()
branchTo target = asm $ branch target

branchIf :: CompOperation -> Label -> Translator ()
branchIf op label = asm $ asmJump op label
```

Además de ello, para la llamada a funciones se utilizan las instrucciones BL y BX, que además de ejecutar un salto hacia la localización indicada, almacenan la próxima dirección en el Link Register para luego ser recuperada cuando la función retorne:

```
grantControl :: String -> Translator ()
grantControl target = asm $ BL target

returnControl :: Translator ()
returnControl = asm $ BX LR
```

7.1.3. Instrucciones

Para soportar las operaciones de nuestro lenguaje tan sólo se requiere de algunas de las muchas instrucciones que ARM posee:

```
data Instruction = Nop Label
                 | B   String
                 | BL  String
                 | BX  Register
                 | BEQ String
                 | BNE String
                 | BGT String
                 | BLT String
                 | BGE String
                 | BLE String
                 | Mov Register Operand  Label
                 | Mvn Register Register Label
                 | Store Register MemSource Label
                 | Load  Register MemSource Label
                 | LoadByte Register MemSource Label
                 | Arith ArithOp Register Register Operand Label
                 | Cmp Register Operand Label
                 | Push [Register]
                 | Pop  [Register]
                 | SWI Int -- signal
```

Estas instrucciones pueden dividirse en dos grandes grupos: de salto y de operación:

- *B*: salto incondicional a la etiqueta indicada
- *BL*: salto incondicional, guardando la dirección de la siguiente línea en el registro LR, para cuando se regrese se continúe por esa dirección. Se utiliza para llamada a funciones
- *BX*: en nuestra traducción emitimos la instrucción *bx lr* para retornar desde una función llamada.
- Saltos condicionales: estos se realizan en caso de que se cumpla el resultado de comparación que señala el mnemotécnico. Este resultado se refleja en el *CPSR: Current Program State Register/Flags*[1]
- *Mov*: instrucción para volcar en el primer operando el valor del segundo. *Mvn* realiza lo mismo pero realizando un NOT sobre el valor, lo que se utiliza dentro del complemento a dos[2] para calcular un número negativo. Tanto aquí como en las siguientes instrucciones se permite apuntar un *Label* para poder
- *Store* y *Load*: Instrucciones para el almacenaje y la recuperación de datos en memoria.

[1] https://azeria-labs.com/arm-data-types-and-registers-part-2/
[2] https://es.wikipedia.org/wiki/Complemento_a_dos

- *LoadByte*[3], carga el valor contenido en la posición de memoria contenida dentro del segundo registro, almacenando dicho valor en el primero. Esta instrucción puede utilizarse, como es obvio, para operar con punteros, sin embargo en el presente proyecto queda restringida a la carga de caracteres desde memoria para la función *print*.
- Operaciones aritméticas y de comparación
- *Push* y *Pop* se usan aquí para salvaguardar el contenido de los registros de R0 a R3 en la llamada a funciones, utilizados para temporales, para poder mantener consistentes operaciones como la siguiente:

```
int x = cos(y) + sin(x)
```

- Por último *SWI* realiza llamadas al kernel del sistema operativo, las cuáles son necesarias para la impresión de cadenas y notificar el final del programa.

Merecen especial atención las instrucciones dedicadas a acceso a memoria (Store, Load y LoadBinary), las cuales pueden hacer referencia a contenido en memoria de diversas formas (indexada, acceso a la región de datos estática, o a través de registros). Además de esto existen operandos flexibles, que permitirían referenciarse no sólo por el contenido en registros, sino también por referencias a memoria o con valores inmediatos.

```
data MemSource = Index Int
               | DataPointer String
               | Parameter String -- \=
               | Direct Register -- [r1]

data Operand = Reg Register
             | ImmNum Int
             | Mem Index
```

7.1.3.1. Interrupciones

La instrucción *SWI* realiza interrupciones software, necesarias tanto para imprimir por pantalla cadenas de texto como para finalizar nuestro programa. Además, es necesario seleccionar el tipo de interrupción, cargando en el R0 el entero correspondiente (1 para terminar el programa, 4 para imprimir por pantalla).

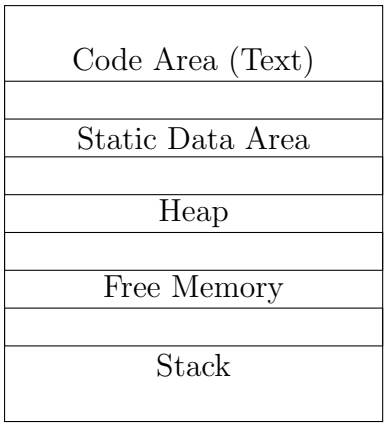
```
interruption :: Instruction
interruption = SWI 0

exitOption :: Instruction
exitOption = Mov syscallRegister (ImmNum 1) nolabel
```

[3] https://developer.arm.com/documentation/ddi0406/cb/Application-Level-Architecture/Instruction-Details/Alphabetical-list-of-instructions/LDRB--register-

```
printOption :: Instruction
printOption = Mov syscallRegister (ImmNum 4) nolabel
```

7.1.4. Secciones

El código emitido en ARM contiene por un lado una sección de código y otra de datos estáticos, que son almacenados en memoria cuando el sistema operativo carga el programa para ser ejecutado. La direcciones posteriores serán las que se ubiquen dinámicamente los datos en memoria, tal y como se muestra en la figura 7.3.




```
.section .text
.global _start

_start:
    mov r0, #1
    mov r1, #0
    svc 0x0

.section .data
my_string:
    .asciz "Hello, world!"
```

Figura 7.3: Secciones del código ensamblador y su ubicación en memoria

En nuestro caso sólo se hará uso de la zona de datos estáticos para almacenar las cadenas de texto constantes.

```
data Program = Program{
    textsec :: TextSection,
    datasec :: DataSection
}

newtype TextSection = Insts [ARM.Instruction]

newtype DataSection = DataSection {strings :: [AsciiText]}
```

7.2. Traducción a ARM

Al igual que en la traducción a código IR, la traducción a código ARM se realiza mediante un *Monad Writer*, con la diferencia de que el recorrido por la estructura de entrada es más simple, al ser esta una lista de instrucciones IR.

7.2.1. Estado interno del traductor

```
data TranslatorCtx = TranslatorCtx {
    program       :: ProgramState,
    registers     :: RegistersVault,
    datasection   :: DataSection,
    counter       :: Int,
    ...
}
```

Al igual que el traductor a código intermedio, el compilador a código ARM necesita mantener un estado compuesto por los siguientes campos dentro de un *Monad State*:

- La sección de datos estáticos que a medida que se realice la traducción almacene las strings que encuentre:

```
storeStr :: String -> Translator String
storeStr str = do
    index <- lift useCounter
    dat <- gets datasection

    let ascii = newConstStr str index

    let strs = strings dat ++ [ascii]

    modify (\ctx -> ctx{datasection =
            DataSection { strings = strs } })

    return (strLabel ascii)
```

- Un segmento de registros para utilizar en expresiones, bien sean operandos o el resultado. En el primer caso se indica que no vamos a usar más ese registro, en el segundo bloqueamos el uso de ese registro para que no pueda ser sobreescrito:

```
result :: Temporal -> Translator Register
result temp = lift $ getRegOf temp -- get a new register

operand :: Temporal -> Translator Register
operand temp = do -- get and release
    reg <- lift $ getRegOf temp
    lift $ leave temp
    return reg
```

- Un mapa de todas las cabeceras de las funciones definidas por el usuario para poder llamar y ubicar los argumentos a través del frame de la función:

```
getFrameFrom :: String -> ContextState Frame
```

```
getFrameFrom name = do
    program <- gets program
    let func = fromJust (Map.lookup name program)

    return $ frame func
```

- un contador global para generar nombres de etiquetas que no colisionen

7.2.2. Operaciones

Como se ha visto en el apartado de instrucciones, el código ensamblador proporciona una traducción directa de las operaciones aritméticas y de comparación:

```
arithmetic :: Label -> ArithOperation -> IR.Temporal
           -> IR.Temporal -> IR.Temporal -> Translator ()
arithmetic label op t0 t1 t2 = do
    r1 <- operand t1
    r2 <- operand t2
    r0 <- result t0

    asm $ Arith (ir2asm op) r0 r1 (Reg r2) label

comparison :: Label -> IR.Temporal -> IR.Temporal -> Translator ()
comparion label t1 t2 = do
    r0 <- operand t1
    r1 <- operand t2

    asm $ Cmp r0 (Reg r1) label
```

Un caso de traducción no directamente intuitiva es el de la negación de enteros, para el que se usa el algoritmo de complemento a dos:

```
neg :: IR.Temporal -> IR.Temporal -> Translator ()
neg temp1 temp2 = do
    r2 <- operand temp1
    r1 <- result temp2

    asm $ MVN r1 r2
    asm $ Arith ARM.Add r1 r1 (ImmNum 1) nolabel
```

Este algoritmo consiste en complementar el número (se cambian los 0's por 1's y viceversa), de lo que se encarga la instrucción MVN y luego se le suma 1. De esta forma se obtiene el mismo número con signo contrario.

7.2.3. Invocación de funciones

Tanto dentro de los statements como de las expresiones pueden realizarse llamadas a funciones. Este proceso conlleva establecer un protocolo de llamada para:

- conservar el estado actual para poder reanudarlo tras la invocación
- para transferir datos a la función llamada
- devolver el control a la función llamante

```
call :: String -> [Operand] -> Translator ()
call name args = do
    pushArgs name args
    grantControlTo name

callWithReturn :: String -> [Operand] -> Temporal -> Translator ()
callWithReturn name args temp = do
    saveRegisters

    pushArgs name args

    res <- operand temp

    grantControlTo name
    restoreRegisters

    asm $ ARM.Mov res (ARM.Reg returnRegister) nolabel
```

7.2.3.1. Árbol de activación. Prologo y epílogo

El árbol de activación refleja cómo a partir de la función main se realizan las llamadas de unas funciones a otras.

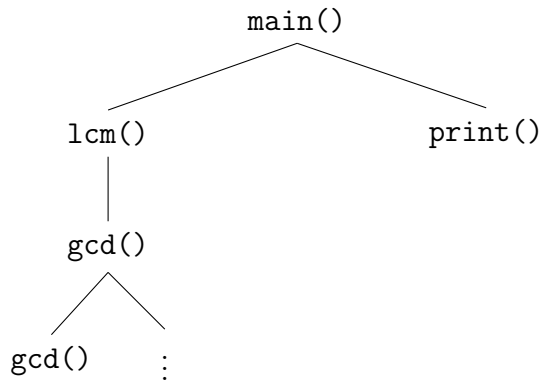


Figura 7.4: Ejemplo de árbol de llamadas

En función de esta estructura clasificamos cada función que hemos obtenido de la fase de IR en si son o no nodos hoja, teniendo en cuenta de si realizan llamadas a otras funciones.

```
data FunctionType = Leaf | NoLeaf deriving (Eq, Show)

leafFunc :: [IR.Instruction] -> FunctionType
leafFunc instrs
  | null instrs = Leaf
  | any isNonLeaf instrs = NoLeaf
  | otherwise = Leaf
  where
    isNonLeaf IR.Instruction {IR.op = Call {}} = True
    isNonLeaf IR.Instruction {IR.op = CallRet {}} = True
    isNonLeaf IR.Instruction {IR.op = Print {}} = True
    isNonLeaf _ = False
```

De esta clasificación deriva el uso de los diferentes entradas y salidas de las invocaciones. Así, se distinguen para cada tipo de función un prólogo y un epílogo distintos [11]:

- **Prologo**: se encarga de establecer el contexto de una función. Si es una función **nodo** (llamante), el prólogo almacena el *frame pointer* y el *link register*, para saber a qué punto restaurar el stack pointer al retornar, y conocer a qué dirección saltar para reanudar la ejecución. Si es una función **nodo** y no realiza llamadas, tan sólo se necesita almacenar el *frame pointer*. Después, reserva espacio en la pila para datos locales a la función.
- **Epílogo**: restablece el contexto de la función anterior. Restaura el *stack pointer* a la dirección del *frame pointer*, esto es, retira el espacio reservado para variables locales; restaura el *frame pointer* anterior, y realiza un salto al *link register* almacenador en la pila, en el caso de que sea una función hoja, o ubica el *link register* en el *program counter* si es un función nodo.

7.2.3.2. Protocolo para paso de parámetros

El paso de parámetros en ARM sigue un protocolo basado en ubicar los cuatro primeros argumentos en los primeros cuatro registros (R0-R3). Como se ha explicado anteriormente, en la presente implementación esos registros pasan a ser R4, R5, R6 y R8 para evitar colisiones con los temporales. A partir de ahí, los argumentos han de ubicarse por la función llamante en el *stack frame* virtual de la función llamada.

```
allocParams :: [Param] -> (Parameters, Int)
allocParams params = do
    -- ARM: the first 4 args use to be passed through registers
    let regParams = zip (map name params)
                        [Reg R4, Reg R5, Reg R6, Reg R8]

    -- if there's more args use memory
    if length params <= 4 then (Map.fromList regParams, 0) else do
```

```
        let restNumber = length params - length regParams

        let tail = drop restNumber params

        let (i, memParams) = mapAccumL
        (\idx param -> (idx + size (type_ param), Pos idx)) 0 tail

        (Map.fromList $ regParams ++ regParams, i)
```

Una vez dentro de la función llamada, se ha optado por reubicar los argumentos en el espacio de las variables.

```
loadParam :: Param -> Translator ()
loadParam param = do -- move param to var location
    paramLocation <- lift $ getParamPos (name param)
    varLocation   <- lift $ getSymbolPos (name param)

    case paramLocation of
      Reg reg -> tell [ -- register to mem
            Store reg (Index varLocation) nolabel
      ]
      -- load the param from memory and store it
      -- in the var location in memory
      Pos pos -> tell [
            Load R0 (Index pos) nolabel,
            Store R0 (Index varLocation) nolabel
      ]
```

Aunque se requiere de más espacio, al tener que duplicar espacio necesario para ello, se ha escogido esta opción por dos motivos:

- Los registros utilizados para argumentos son volátiles y se utilizan para otros fines, por lo que sólo deberían servir para traspasar los datos de los argumentos entre funciones.
- Hace más sencilla la implementación, al realizar todas las lecturas de variables sobre el mismo espacio del stack, que para su traducción funciona como un mapeo con nombres.

7.2.3.3. Retorno del control a la función llamante

El retorno a la función llamante se realiza mediante un salto a la etiqueta formada por el nombre añadiéndole "_end":

```
ret :: Maybe Operand -> Translator ()
ret src = do
    case src of
        Nothing -> tell []
        Just val -> case val of
```

```
            Number num -> asm $ returnMov (ARM.ImmNum num) nolabel
            Symbol name _ -> do
                pos <- lift $ getSymbolPos name
                asm $ ARM.Load returnRegister (Index pos) nolabel
            Temp temp -> do
                reg <- operand temp
                asm $ returnMov (ARM.Reg reg) nolabel
            ConstString str -> do
                strdata <- storeStr str

                asm $ loadFromData R0 strdata nolabel
                asm $ returnMov (ARM.Reg R0) nolabel

    gotoEnd

gotoEnd :: Translator ()
gotoEnd = do
    name <- lift getName

    branchTo (name ++ "_end")
```

7.2.4. Comienzo y fin del programa

En ARM, el comienzo de la ejecución arranca, por defecto, a partir de la instrucción con la etiqueta **_start**. En nuestra traducción en ella se da paso a la función **main**, cuando esta retorna, se procede a finalizar el programa:

```
start :: Translator ()
start = do
    setCheckpoint "_start"
    grantControlTo "main"
    exit
```

Para indicar al sistema operativo que el programa ha finalizado la traducción selecciona la opción y emitimos una interrupción software:

```
setExit :: Translator ()
setExit = asm $ Mov syscallRegister (ImmNum 1)

exit :: Translator ()
exit = setExit >> interrupt
```

7.3. Conclusión

En este capítulo se ha descrito el proceso de traducción a código ensamblador, tarea facilitada por la linearidad del código IR. Se ha querido respetar lo máximo posible la estructura del archivo ARM con el objetivo no sólo de exponer adecuadamente su sintaxis sino también como forma de explicar su funcionamiento en la ejecución.

7.3.1. Posibles mejoras

La gran carencia referente a esta etapa, tanto en lo que concierne a la traducción como a las propias capacidades del lenguaje, es la ausencia de gestión dinámica de la memoria. Por brevedad y alcance de la materia se ha preferido no introducir la temática aquí. Sin embargo, para tener en cuenta en incrementos posteriores, existen dos principales posibilidades bien estudiadas para implementar: o bien gestión de memoria manual, lo que implicaría desarrollar una extensión para comunicarse con el sistema operativo, o bien dotar al lenguaje de un *runtime* para que pueda albergar un recolector de basura. Una tercera vía sería explorar el terreno de la gestión de memoria declarativa como realiza Rust[4].

Además, dado que el código IR permite escribir una traducción específica para cada arquitectura, una ampliación que ponga en valor esto sería implementar un traductor a una arquitectura alternativa como x86. Es posible también añadir una fase de optimización con la misma estructura que la propuesta en el este capítulo.

7.3.2. Comparación con la versión imperativa

Dado que este capítulo desarrolla una traducción de forma similar a la vista en el anterior, apenas pueden añadirse matices a la comparativa con el paradigma imperativo. Sin duda un aspecto en el que recabar es, desde el punto de vista del desempeño, lo inadecuado de la API de *Monad Writer* para adjuntar cada parte del código al todo. Esto está basado en las instancias de la listas como semigrupo y monoide:

```
instance Semigroup [a] where
  (<>) = (++)

instance Monoid [a] where
  mempty = []
```

Mientras que los lenguajes imperativos es más óptima esta acumulación, debido a que es realizada mediante punteros, Haskell necesita recorrer ambas listas, lo que representa un crecimiento cuadrático del problema. Para paliar esto existe la posibilidad de utilizar *differentiable lists*[5] con composición de funciones para unir listas, en vez de *[String]* como se ha utilizado anteriormente.

[4] https://www.rust-lang.org/
[5] https://matthew.brecknell.net/posts/difference-lists/

7.3.3. Epílogo

Una vez finalizada la última fase de la compilación, hemos de enlazar todas la partes que la componen. Esta parte final del proyecto se centrará en la usabilidad del mismo, dejando atrás los detalles de construcción para centrarse en el cometido al que sirve.

Capítulo 8

Integración de las etapas

En esta última parte del proyecto se van a integrar todas las partes desarrolladas en los anteriores capítulos, haciendo un repaso de la arquitectura del compilador y mostrando cómo conectan las partes entre sí.

8.1. Ensamblaje de *frontend* y *backend*

El ensamblaje de cada etapa de la arquitectura se articula mediante la instancia de la clase *compiler*:

```
class Compiler source target where
    compile :: source -> target
```

Por ejemplo, el *frontend* puede fomularse como un *Except Monad*, representando la dualidad del reporte de errores y la creación del AST.

```
type Frontend = ExceptT String (Reader String)

instance Compiler String (Either String TypedAST) where
    compile = runReader (runExceptT analysis)
```

Podemos retomar el enfoque conceptual expuesto en la introducción, uniendo tanto el análisis sintáctico y semántico bajo la forma de acciones del *Frontend*. Es importante señalar aquí que se hace uso de un *Monad Reader* para albergar el código fuente, necesario en ambas partes para poder ubicar los errores:

```
analysis :: Frontend TypedAST
analysis = syntax >>= semantics

syntax :: Frontend UntypedAST
syntax = do
    code <- ask
    case parse program "" code of
        Left err -> throwError err
        Right ast -> return ast

semantics :: UntypedAST -> Frontend TypedAST
semantics ast = do
```

```
    code <- ask
    case analize code ast of
        Left err -> throwError err
        Right ast -> return ast
```

Así, con el anterior código estamos expresando que el análisis está conformado por una fase de análisis sintáctico y otra de análisis semántico. Ambas fases por dentro son casi idénticas: ambas necesitan contar con el código fuente, y a partir de ahí o bien emiten un error o producen un AST.

De una forma más simple, pero con el mismo sentido, puede formularse el ***backend*** respecto de la *síntesis*:

```
instance Compiler TypedAST Program  where
    compile :: TypedAST -> Program
    compile = IR.compile >>> Arm.compile
```

Cada fase, de traducción a IR y ARM, implementan la *typeclass* **Compiler**, encadenándose y formando, a su vez, otra instancia de la misma clase: la que traduce desde el AST tipado a la formulación de un programa en código ensamblador.

8.2. Función main. Entrada y salida

Una vez ensambladas las partes a nivel de arquitectura sólo resta codificar la ejecución del compilador mediante una función:

```
runCompiler :: String -> IO String
runCompiler src = do
    let front :: (Either String TypedAST) = Frontend.compile src
    case front of
      Left err -> putStr err
      Right subs -> do
        let program = Backend.compile subs
        return $ show program
```

Se ha de tener en cuenta que esta función necesita devolver una **mónada IO** para poder cubrir tanto la funcionalidad de impresión de errores tanto como para devolver el código ensamblador resultado de todo el proceso. Ello nos da la oportunidad, además, de componer la llamada al compilador directamente en la función *main*, obteniendo una composición perfecta:

```
main :: IO()
main = do
  (source:dest:_) <- getArgs
  code <- readFile source
  asm  <- runCompiler code
  writeFile dest asm
```

De esta forma, la función *main* se compondría de las siguientes fases: extraer, de los argumentos indicados por terminal, el nombre del archivo donde reside el código fuente, y aquel en el que vamos a volcar el ensamblador generado; leer el contenido del primero y ejecutar el compilador con él; y por último escribir en el segundo el ensamblador obtenido. Nos ha sido posible integrar *runCompiler* dentro del main precisamente porque esta implementa la **mónada IO**, lo que nos está habilitando a contar tanto con efectos (lectura y escritura de ficheros) como con reporte de errores de cada una de las partes anteriormente numeradas.

Capítulo 9

Conclusiones

Finalmente se ofrece en este capítulo una breve retrospectiva de lo realizado aquí, exponiendo las dificultades y dando cuenta de los objetivos propuestos cumplidos.

9.1. Dificultades

El principal problema encontrado durante el desarrollo ha sido el número y complejidad de los temas tratados. Para abordar cada uno de ellos ha sido necesario tener claro desde el principio un nivel de profundidad que no sobrepasar para no perder el enfoque del objeto de estudio. Así, por ejemplo, se han adoptados soluciones subóptimas o alternativas *naif*, como por ejemplo el problema de la asignación de registros a variables y temporales en el la traducción al código final. El criterio para desechar soluciones de mayor complejidad ha sido no sobrepasar la extensión media de los epígrafes sin por ello eludir la mención a otras soluciones mejores.

Por otro lado, la elección de un lenguaje como Haskell, que se acoge al paradigma funcional desde sus propios fundamentos, representa un reto al tener que diseñar los componentes con una mentalidad distinta. Esto en conjunción con el punto anterior ha conllevado la constante reformulación de algunas de las decisiones tomadas en cuanto a diseño.

Por último, el proyecto ha supuesto un desafío en forma y tiempo, dado que al no tener experiencia ni en el lenguaje ni con la temática desde esta perspectiva (un compilador completo) el tiempo dedicado a cada parte excedía con mucho lo esperado por el autor en un principio. En este sentido, el presente trabajo comporta también el aprendizaje de saber anticipar las dimensiones de tiempo y esfuerzo a dedicar.

9.2. Objetivos alcanzados

De lo comentado en la introducción, la propuesta de una alternativa al paradigma imperativo para la acometida de un compilador ha quedado firmemente expuesta y con resultados no solamente accesibles en su comprensión sino también poniendo de relieve lo fácilmente escalable y ampliable que resulta en términos de desarrollo. Así pues, podemos decir que el objetivo marcado al comienzo de esta memoria ha sido alcanzado.

Además, se ha ensanchado el bagaje tanto con el lenguaje como con el paradigma. Aún habiendo eludido explicaciones más teóricas acerca de los recursos utilizados, como por ejemplo todo lo referente a teoría de categorías, el presente trabajo logra aprovechar la versatilidad de unas pocas *typeclasses* para cubrir buena parte de las necesidades.

Para acabar, el trabajo representa tanto un punto y final al recorrido que ha conllevado el estudio de la ingeniería por las materias más señeras que la conforman, así como también el aunarlas y darles un propósito práctico conlleva el abrir un nuevo punto de partida para profundizar en ellas, esperando así poder darle otra vida al proyecto más allá de lo reflejado en estas líneas.

9.3. Trabajos futuros

La ampliación de este proyecto tiene dos posibles vías complementarias:

- En la línea de lo sugerido en las conclusiones de cada unos de los capítulos es posible, como se ha demostrado, dar cabida a extender el lenguaje a nuevas construcciones tanto sintácticas como semánticas. Además de las ya citadas, un paso lógico (en consecuencia con su naturaleza imperativa) es dotar al lenguaje de abstracciones de composición propias del paradigma orientado a objetos, tales como **clases, herencia, interfaces**, etc. Y junto a ello también la posibilidad de incluir **espacios de nombres** y la importación de **múltiplos archivos**. Esta línea vendría, por tanto, a extender al lenguaje en cuanto a sus características.
- Otra forma de hacer avanzar el proyecto es, en relación al propio objetivo de este escrito, perfeccionar la implementación de forma que consiga reflejar de forma más precisa y legible el comportamiento y fines de cada fase y componente de su arquitectura. Con ello se lograría que, como se ha dado pie en estas páginas, aquellos conceptos propios de la compilación tengan una traducción unívoca en lenguaje funcional.

Ambas metas necesitan la una de la otra: en tanto que la ampliación del proyecto conlleva una fase de refactorización que permita al mismo continuar siendo mantenible, esta última requiere, en la medida en que reformula el funcionamiento de sus partes, ser puesta a prueba y contar con múltiples características de las que extraer y abstraer patrones que son comunes a todas ellas.

Esta perspectiva no es sino la expresión última de la naturaleza ingenieril que porta un proyecto de esta complejidad, lo que conlleva no limitarse a garantizar el buen funcionamiento respecto de los resultados esperados, sino de proveerlo a través de plasmar, mediante código, el diseño de un sistema de información que le es propio.

Bibliografía

[1] T. Ball, Writing an Interpreter in Go. Thorsten Ball, 2018.

[2] VV.AA., "Timeline of programming languages," https://en.wikipedia.org/wiki/Timeline_of_programming_languages, [Online; accessed 7-Mayo-2023].

[3] S. Diehl, "Write you a Haskell," https://smunix.github.io/dev.stephendiehl.com/fun/index.html, 2015, [Online; accessed 7-Mayo-2023].

[4] R. Sethi, Compiladores: Principios, técnicas y herramientas. Pearson Addison-Wesley, 2008.

[5] R. Nystrom, Crafting Interpreters. genever benning, 2021.

[6] VV.AA., "mtl state monad," https://hackage.haskell.org/package/mtl-2.3.1/docs/Control-Monad-State-Lazy.html, [Online; accessed 7-Mayo-2023].

[7] M. Fowler, "Repository," https://martinfowler.com/eaaCatalog/repository.html, [Online; accessed 7-Mayo-2023].

[8] VV.AA., "Alex," https://hackage.haskell.org/package/alex, [Online; accessed 7-Mayo-2023].

[9] G. Hutton, Programming in Haskell. 2nd Edition. Cambridge University Press, 2007.

[10] G. J. Chaitin, "Register allocation & spilling via graph coloring," 1982.

[11] VV.AA., "Writing ARM Assembly," https://azeria-labs.com/writing-arm-assembly-part-7/, [Online; accessed 7-Mayo-2023].

[12] L. T. Keith D. Cooper, Engineering A Compiler. 2nd Edition. Morgan Kaufmann, 2012.

[13] M. Karpov, Megaparsec tutorial, 2021.

[14] T. Ball, Writing a Compiler in Go. Thorsten Ball, 2018.

[15] VV.AA., "Hspec. User's Manual," https://hspec.github.io/, [Online; accessed 7-Mayo-2023].

Apéndice A

Uso y entorno de ejecución

Con el objetivo de poder cerrar un ciclo de desarrollo del lenguaje y poder comprobar el resultado de los programas escritos se expone en este anexo tanto el uso del compilador como la ejecución de los programas en ensamblador obtenidos. Se proporciona, por tanto, una primera forma de de comprobar el correcto funcionamiento de lo expuesto hasta aquí.

ARMv7	Machine code
max: ... comp r4, r5 ble else mov r3, r5 b end else: mov r3, r4 end: ...	01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101 01010101

A.1. Invocación del compilador

El compilador es un programa que se invoca a través de la lína de comandos. Para obtenerlo es necesario compilar todo el código que se ha venido desarrollando hasta ahora. Para la fácil obtención del proyecto es necesario acudir al siguiente enlace en GitHub https://github.com/pabloos/chio y descargar, bien en formato zip o utilizando la herramienta de control de versiones Git, el código del proyecto. Se ha elegido este hospedaje para poder mantener vivo y abierto el proyecto, de forma que sea sencillo tanto ampliarlo como documentarlo y poder abrirlo a la comunidad libre.

A.1.1. Entorno de desarrollo

El proyecto utiliza Stack como herramienta de desarrollo, cubriéndose con ella la construcción del mismo y la gestión de dependencias. Para obtener esta herramienta se recomienda seguir el tutorial de instalación en su página oficial: https://docs.haskellstack.org/en/stable/.

Dentro del repositorio se encuentran distintos archivos de configuración relativos a ello que asegurar este comportamiento. Ello supone, como facilidad para el desarrollador, que una vez dentro del repositorio, Stack se encargará de obtener todas las librerías involucradas en el proyecto. Tan sólo hemos de ordenarle la construcción de los binarios y la utilidad se encargará de recopilar las dependencias necesarias:

```
$ stack build
chio-0.1.0.0: unregistering (local file changes: src/Compiler.hs)
chio> configure (lib + exe)
Configuring chio-0.1.0.0...
chio> build (lib + exe)
Preprocessing library for chio-0.1.0.0..
Building library for chio-0.1.0.0..
[59 of 63] Compiling Compiler
...
```

Tras ello, es posible utilizar el compilador dentro del mismo entorno. Tal y como se vio en el *main* 8, el comando requiere que se le indique un primer archivo para leer el programa y otro segundo donde guardar el ensamblador generado:

```
$ stack exec sources/hello.chio targets/hello.s
```

Los dos pasos anteriores pueden realizarse en uno sólo mediante la utilidad *stack run*:

```
$ stack run sources/hello.chio targets/hello.s
```

Con esto es posible realizar un testeo manual del compilador, observando el código ARM producido. Sin embargo, resulta idóneo poder comprobar directamente los resultados de estos programas.

A.2. Ejecución de los programas

En este apartado se muestra cómo ejecutar los programas obtenidos al compilar en un entorno GNU/Linux. Aunque las herramientas puedan variar entre las distintas arquitecturas y sistemas operativos, este tipo de plataforma resulta lo bastante ubicua y además da buen ejemplo del proceso de forma general.

A.2.1. Entorno ARM

Para ejecutar el código ensamblador es necesario contar con un entorno perteneciente a la arquitectura ARM. Existen dos opciones:

- Contar con una máquina con esta arquitectura, como por ejemplo una microcomputadora

como la Raspberry Pi.

- Recurrir a la emulación para obtener un sistema ARM en una máquina que no lo es.

Para cubrir el más restringido de los casos, se provee a continuación de un pequeño tutorial para llevar a cabo la emulación a través de *QEMU* [1]

A.2.1.1. Instalación de *QEMU*

En este caso se describe una instalación sencilla de *QEMU* en un sistema GNU/Linux, en su distribución Ubuntu. Para ello es posible recurrir al sistema de paquetes apt:

```
$ sudo apt update
$ sudo apt install qemu-user
```

Dentro del paquete instalado se encuentra la utilidad **qemu-arm**, que brinda la capacidad de ejecutar un binario en una máquina virtual ARM.

A.2.2. Ensamblado, enlace y carga

En primer lugar, resulta imprescindible discutir las distintas fases y formatos de archivo implicados en la confección del programa final para ser ejecutado. Estas son ensamblado, enlace y carga:

- En el **ensamblado** el código ensamblador se traduce a código máquina o *código objeto*. Esto conlleva en buena medida la conversión de cada mnemotécnico en su equivalente en binario.
- En el **enlace (*linking*)** distintos archivos de código objeto se unen para generar un ejecutable. En nuestro caso el formato de ejecutable utilizado es elf (*eXecutable and linkable format*), con la extensión *.elf*.
- Una vez el .elf es mandado a ejecutar a través de la llamada al kernel *execv*, tiene lugar una fase de **carga (*load*)** en memoria y una modificación de contador de programa a la dirección correspondiente para que la CPU ejecute el código ubicado en memoria.

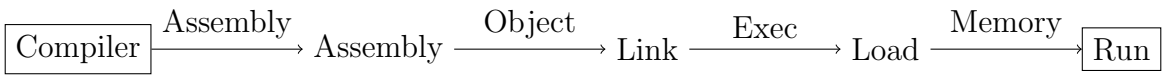


Figura A.1: Pipeline después de la compilación

Es importante resaltar que estas herramientas están destinadas a producir resultados para una plataforma específica. Así por ejemplo, si fuésemos a utilizar un entorno ARM nos valdría con utilizar las herramientas propias para ensamblaje (*as*) y linkado (*ld*):

[1] *QEMU* es un software libre para la emulación y virtualización. para más información se recomienda acudir a la página del proyecto: https://www.qemu.org/

```
$ as hello.s -o hello.o
$ ld hello.o -o hello.elf
```

No obstante los siguientes ejemplos no se prestan a la ambigüedad de esa presunción y utilizan las herramientas específicas para dicha plataforma. Con ellas, podemos estar en un entorno x86 y ensamblar y linkar código para ARM, ejecutnado después el emulador *QEMU* para ejecutar el binario resultante.

A.2.2.1. Instalación de las herramientas

Para el ensamblado y linkado es necesaria la instalación del paquete *binutils* en su versión para ARM con la ABI de GNU [2].

```
$ apt install binutils-arm-linux-gnueabi
```

Como alternativa, en la mayor parte de las distribuciones el paquete GCC[3] está disponible y cuenta con las herramientas homónimas para nuestro propósito.

```
$ apt install gcc-9-arm-linux-gnueabi
```

A.2.3. Ejecución del código ensamblador

Una vez obtenidas las herramientas necesarias podemos empezar a procesar el código. Tal y como podemos comprobar, el código ensamblador que genera el compilador es tan solo un archivo de texto:

```
$ file hello.s
hello.s: ASCII text
```

Para obtener el código objeto es necesario invocar el comando *as*:

```
$ arm-linux-gnueabi-as hello.s -o hello.o
hello.o: ELF 32-bit LSB relocatable, ARM,
EABI5 version 1 (SYSV), not stripped
```

Una vez hecho esto utilizamos el código objeto para obtener el ejecutable mediante el linkado con *ld*:

[2]La ABI de GNU para ARM permite linkar código para esta arquitectura en concreto

[3]GNU Compiler Collection (GCC) es el toolchain de utilidades para la compilación de diversos lenguajes, incluyendo tanto frontends, backends y optimizadores como otras herramientas para el manejo de binarios, listados en https://gcc.gnu.org/.

```
$ arm-linux-gnueabi-gcc-9 hello.o -o hello.elf -nostdlib
hello.elf: ELF 32-bit LSB executable, ARM,
EABI5 version 1 (SYSV), statically linked
```

Para ejecutar el binario resultante tan sólo hay que invocar el emulador de ARM indicándole el programa que queremos ejecutar:

```
$ qemu-arm hello.elf
hello world
```

A.3. Conclusión

Finaliza con este anexo un ciclo de desarrollo del proyecto, ubicados en él todas las etapas de uso del lenguaje propuesto. Como puede observarse, esta última parte resulta poco ergonómica y muy dependiente de las distintas plataformas, obligando al usuario a conocer más herramientas que el propio lenguaje. En este sentido, para hacer al compilador autosuficiente, una posible mejora sería la anexión de todo estas herramientas al proceso de compilado para ganar en facilidad de uso. El camino para realizar esto implica reproducir toda la lógica de los distintos conceptos que se han visto en este capítulos: ensamblaje del código ensamblador a código objeto, linkado de varios objetos, producción de binarios en formato ELF, etc.

Vista toda la complejidad que esto representa parece adecuado, como se discutía en la introducción del proyecto, evitar toda dependencia con la plataforma y desarrollar un *runtime* para el lenguaje que pueda abstraer al usuario de todo lo expuesto aquí y de lo que, con todo, no se ha expuesto por no querer sacrificar brevedad y concisión. Sin embargo, se ha considerado de interés exponer al lenguaje a esta necesidad por motivos didácticos, ya que gracias a ello es posible adentrarse en esta otra cara de la compilación que son los sistemas de destino.

Apéndice B

Entorno de test

Siendo los compiladores una herramienta fundamental en el desarrollo de software, resulta indiscutible la necesidad de garantizar su correcto funcionamiento. El testing de un compilador es una actividad ineludible en su desarrollo, dadas las dimensiones del proyecto y el compromiso adquirido en su buen desempeño. Por el alcance de este requisito se hace necesario enfrentar la tarea de forma estructurada.

El desarrollo exhaustivo de pruebas para un proyecto de estas características, aunque fundamental, escapa al dominio del presente proyecto. Por ello, en lo sucesivo se expone un pequeño corpus de características para una posible ampliación así como algunos ejemplos de implementación de tests con el objeto de ilustrar su acometida en Haskell.

Por la naturaleza modular del compilador, dividido en fases, el testing del mismo se acopla intuitivamente a la división entre suites de tests unitarios, que recogen la funcionalidad parcial en cada etapa; y otra de integración, que comprende el compilador completo.

- Por un lado los **tests unitarios** sirven para obtener un feedback recurrente a medida que avanza el desarrollo, de forma que seamos capaces de corroborar su correcto avance al obtener los resultados esperados
- Por otro, los **tests de integración** aseguran que la arquitectura del compilador es correcta y responde adecuadamente a los objetivos de la misma.

Además, dado que los compiladores constituyen un software de uso recurrente, es pertinente considerar el entorno de pruebas como garante de dos características del proyecto: su mantenibilidad y su capacidad de ser expandido.

- Por una parte los tests permiten dar buena cuenta del estado del proyecto y de la verificación de determinadas características y construcciones del lenguaje. De esta forma, sirven tanto para obtener fallos del sistema inesperados como así también para dejar constancia de su resolución una vez **documentada la incidencia**.
- Por otra, el conjunto de tests provistos por el proyecto puede ser utilizados como **tests regresivos** para que, en caso de añadir más funcionalidad o nuevas construcciones al lenguaje, asegurarnos que no se han alterado las anteriormente añadidas.

Un último beneficio del testing es su valor como documentación del avance del proyecto, donde cada característica e incidencia resuelta queda reflejada en un conjunto de tests.

B.1. Codificación de tests en Haskell

B.1.1. Tests unitarios

Para poner algunos ejemplos de tests en Haskell aplicados a nuestro compilador vamos a valernos del framework de testing *Hspec*[15]. *Hspec* permite definir tests de forma semántica y estructurada, dividiendo cada suite en subapartados:

```
spec :: Spec
spec =
    describe "parser tests" $ do
        describe "expressions parsing" $ do
            it "parses a number" $ do
                let src = "2"
                let expected = Right (IntVal 2)

                parse expr src `shouldBe` expected

            it "parses an addition" $ do
                let src = "2 + 3"
                let expected = Right (Add (IntVal 2) (IntVal 3))

                parse expr src `shouldBe` expected

            it "parses a call" $ do
                let src = "func()"
                let expected = Right (Call "func" [])

                parse expr src `shouldBe` expected
```

B.1.2. Tests de integración

La codificación de tests de integración para compiladores sin embargo representa un problema por el tamaño de la entrada y la salida de los tests. Así, siguiendo con la estructura anterior, una suite de tests de integración de nuestro compilador podría ser de la siguiente manera:

```
spec :: Spec
spec =
    describe "compiler testing" $ do
        describe "complete compilation process" $ do
            it "parses a hello world" $ do
                let src = "fun main () {\n print(\"hello world\n\")\n
                   }"
                let expected = Right [
                    Program{
```

```
                DataSection: ...
                TextSection: {
                    Function "main" ...
                }
            }
        ]

        compile src `shouldBe` expected
```

Como puede observarse, pese a que se haya truncado la codificación, la salida esperada tiene un tamaño tal que hace casi imposible dos tareas fundamentales:

- **Mantener cada *test-case*** con comprobaciones de gran tamaño se torna complicado y propenso a errores.
- **Anulación de las regresiones**: por efecto de lo anterior, se hace imposible acumular una serie de tests que en cada aumento nos aseguren que no hemos desecho los aumentos anteriores. Por ejemplo, si se añade una fase de optimización del código objeto, este se verá alterado, aunque semánticamente el programa emitido responda de la misma forma (**transparencia semántica**).

B.2. Verificación de compiladores

Frente al problema anteriormente expuesto existe actualmente un área de investigación que propone, como complemento a la evaluación usando pruebas, la verificación formal del comportamiento de los compiladores, mediante la prueba de preservación semántica. Este campo pretende, en este sentido, garantizar la transparencia semántica entre el programa fuente y el código emitido, y que por lo tanto las distintas versiones del código dan los mismos resultados en concordancia con lo expresado en el código fuente[1].

B.3. Conclusión

Como colofón a este anexo, podemos afirmar que la codificación de tests para la comprobación de un compilador presenta los suficientes inconvenientes en su realización y mantenimiento como para sugerir como alternativa una aproximación de generación automática y estructurada de las mismas pruebas. En este sentido, pienso que la apuesta y esfuerzos para cubrir estas deficiencias debe de volcarse no tanto en el desarrollo de pruebas extensivas, sino más bien en el uso y generación de pruebas intensivas, menores en número y extensión, pero basadas en manifiestos de características estructuradas y estandarizadas formalmente.

[1] Un ejemplo de ello es CompCert: https://compcert.org/, un compilador que asegura la transparencia semántica para programas escritos en C