Facultad
de
Ciencias

Construcción de un compilador para el lenguaje BeGone

Construction of a Compiler for the BeGone Language

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Pedro Castro Cutillas

Director: Domingo Gómez Pérez

Septiembre - 2024

Acknowledgements

Gracias al apoyo de mi familia y de mis amigos. Gracias a mi tutor, Domingo, que me ha servido de guía en todo este camino.

Resumen

palabras clave: Compilador, Lexer, Parser, Python3, Algoritmo de Compresión

En este trabajo de fin de grado se crea un intérprete para el lenguaje Begone, un lenguaje de programación propuesto por David M. Beazley. El objetivo es que esta implementación se utilice en la asignatura «Lenguajes de programación».

El alumno lleva esta implementación en Python, realizando las siguientes tareas:

- Define especificaciones de los tokens e implementación mediante expresiones regulares
- Define una gramática libre de contexto para el lenguaje Begone, sin ambigüedad
- Utiliza librerías para que el compilador genere código nativo

Adicionalmente, este lenguaje tiene funciones nativas para comprimir ficheros, basado en registros de retroalimentación, que es lo más eficiente posible. Estas funciones:

- Permiten la compresión y descompresión de ficheros de diferentes formatos (por ejemplo, texto, binario).
- Utilizan registros de retroalimentación para optimizar la compresión en función de patrones de uso y tipos de datos más comunes.
- Proporcionan una interfaz sencilla para que los usuarios puedan integrar fácilmente estas funciones en sus programas escritos en Begone.

Construction of a Compiler for the BeGone Language

Abstract

keywords: Compiler, Lexer, Parser, Python3, Compression algorithm

In this final degree project, an interpreter for the Begone language, a programming language proposed by David M. Beazley, is created. The goal is for this implementation to be used in the course "Programming Languages."

The student carries out this implementation in Python, performing the following tasks:

- Defines specifications of the tokens and implementation using regular expressions
- Defines an unambiguous context-free grammar for the Begone language
- Uses libraries to generate native code with the compiler

Additionally, this language has native functions for file compression, based on feedback registers, which are as efficient as possible. These functions:

- Allow the compression and decompression of files in different formats (e.g., text, binary).
- Use feedback registers to optimize compression based on usage patterns and common data types.
- Provide a simple interface for users to easily integrate these functions into their programs written in Begone.

Índice general

1 Introduction

1.1. Motivación

Los compiladores son un pilar fundamental en la tecnología contemporánea. Han conseguido facilitar la programación construyendo capas de abstracción entre la máquina y el programador. Antes se programaba directamente en código ensamblador que después se traducía al código máquina para generar un ejecutable. A día de hoy, lo más común es programar en lenguajes de alto nivel como *Python* o *JavaScript*, donde hay miles de librerías y herramientas que ofrecen múltiples funcionalidades que facilitan el desarrollo de software de cualquier tipo.

Se debe hacer una clara distinción entre compilador e interprete. El compilador traduce todo el código fuente del lenguaje original a código máquina, lo cual, evidentemente, le hace en la mayoría de casos más rápido. Sin embargo, el interprete convierte cada línea, una a una a código máquina teniendo en cuenta los resultados de líneas anteriores con técnicas como el conteo de referencias, garbage collection, etc...

En la primera parte de este trabajo hablaremos del desarrollo de un compilador. He escogido escribir un compilador porque siempre me ha interesado conocer lo que realmente ocurre en la computadora cuando tras escribir tu programa genera un ejecutable porque creo que conocer detalles de programación de bajo nivel son de ayuda en el día a día de un programador. Además, conocer todas las etapas por las que pasa el código fuente hasta convertirse en código máquina, te da una perspectiva global sobre la naturaleza de un lenguaje de programación de alto nivel.

En cuanto a la segunda parte del trabajo, los algoritmos de compresión siempre habían llamado mi atención. Sin embargo, nunca me había tomado el tiempo para analizar ninguno de ellos. Con este trabajo he tenido la oportunidad de profundizar en conceptos como los registros de desplazamiento o la minimización de formulas lógicas que antes desconocía.

En conclusión, la motivación principal de este trabajo era conocer en mayor profundidad los compiladores y los algoritmos de compresión que, en cierta manera, se entremezclan con algunos algoritmos de cifrado.

1.2. Objetivos

El primer objetivo de este proyecto era crear un compilador funcional del lenguaje BeGone, siguiendo las lecciones del curso de David Beazley. Este contiene lecciones básicas para guiarte y después escribir tu propia implementación, algunos tests, y un compilador medio implementado para que compruebes si la salida de tu compilador coincide con la de este. Este compilador de ayuda deja de ser útil a partir de la implementación básica del módulo de generación de código llvm. A partir de ese punto, aunque existen tests, la guía se vuelve mucho más vaga, por lo que las siguientes implementaciones presentan grandes retos. Esta parte posterior del desarrollo abarca la implementación de booleanos, las comparaciones, el control del flujo (sentencias *if*, *ifelse* y *while*) y las funciones, que son las secciones que más tiempo y más complejas son de implementar.

Después del desarrollo completo del compilador, mi tutor me ofreció un artículo Limniotis et al. [2007] donde había un algoritmo que explicaba como transformar cualquier secuencia binaria en un segmento del comienzo de esa secuencia y un polinomio, con estas dos partes puedes decodificar de vuelta la secuencia original. Después de estudiar este algoritmo, se planteó cual sería la mejor forma de guardar la salida de este en un fichero, lo que derivo a desarrollar un formato de archivo especial para este algoritmo. Y con este formato, implemente en el compilador original dos funciones nativas para codificar y decodificar un archivo. Evidentemente, esto provoco cambios en todas las etapas del compilador. Al hacer esto el compilador original paso de llamarse BeGone a GoneFSR.

El último de los objetivos fue desarrollar una memoria técnica donde todo fuera claro para el lector a la vez que didáctico. Para ello, tome como inspiración para la estructura de la sección del desarrollo del compilador, el documento intérprete Lox ofrecido por mi tutor. Para la segunda parte del proyecto, los conceptos se explican desde los más sencillos, hasta los más complejos, para que el lector perciba como se entrelazan las distintas secciones.

1.3. Desarrollo software

Para la escritura del compilador, el modelo de desarrollo software utilizado ha sido el modelo incremental iterativo, tal y como se recomienda en el curso de BeGone. Este consiste en dividir el proyecto en etapas, y para cada una aplicar las distintas fases de desarrollo para producir un software sin errores. Estas etapas se desarrollan secuencialmente, aplicando pequeños incrementos en la complejidad del proyecto después de haberse asegurado que la etapa anterior estaba bien desarrollada. En cada incremento el objetivo es añadir o extender una funcionalidad en el proyecto global, a cada incremento se lo podría asociar con un hito si se conoce el *framework SCRUM*.

Al desarrollar el proyecto de esta manera puedes aplicar tests en cada etapa, haciendo así que el feedback dado por los tests, unitarios sea más fácil de entender y aplicar las correcciones necesarias implique menos esfuerzo.

En la práctica he creado un repositorio git en local y he trabajado sobre dos ramas una *dev* y otra *prod*. Haciendo *commit* en *dev* por cada pequeño avance, y haciendo *merge* sobre *prod* cada vez que completaba una etapa del desarrollo. Podéis consultar el código del compilador en el repositorio. Debo mencionar que no está el historial de logs pertinente. Ya que el compilador fue desarrollado en mayo, y no subí el repositorio original a mi github.

Por último, respecto a la segunda parte del trabajo que se centra en la investigación de los registros de desplazamiento. He dividido cada desarrollo en distintos proyectos en las respectivas carpetas dentro del repositorio siendo estas: minimal fsr (implementación del algoritmo de Limniotis et al. [2007]), file coders (desarrollo del formato nlfsr de la sección 3.4), list product (desarrollo de la implementación en C del producto de dos listas formato nlfsr de la sección 3.5.3), logicformula solver (desarrollo de la enumeración explicíta de ESOP 3.5.2), pruebas de código (donde hay algunos algoritmos cuya compresión fue necesaria para la investigación) y miscelánea (otros programas usados durante la investigación).

1.4. Herramientas de Software

Para el desarrollo del compilador se han usado la librería SLY (Sly Lex Yacc), que tiene como antecesores dos herramientas muy conocidas llamadas lex y yacc (Yet Another Compiler Compiler). Esta librería me ha permitido no tener que implementar el autómata finito del *Lexer*. Además de ahorrarnos el desarrollo de la estructura del *Parser* permitiendo centrarnos en la definición de la

gramática libre de contexto.

También se ha utilizado LLVM (*Low Level Virtual Machine*) el cual es un conjunto de proyectos o herramientas relacionados con la compilación. Además de por ser la recomendación del curso de David Beazley, porque el código llvm es un código muy portable que muchos backends de compilación pueden interpretar para generar código máquina. Se puede argumentar también que LLVM incluye opciones de optimización avanzada para su código intermedio como vectorización, unrolling, entre otras técnicas.

Para transformar el código *llvm* a código máquina se utiliza CLANG, que a su vez utiliza el backend de LLVM. Se utiliza CLANG por recomendación del curso pero se podría utilizar otras herraminetas como *llc* (*LLVM static compiler*) capaces de compilar código *llvm*.

2 Compilador GoneFSR

El compilador está basado en el curso de David Beazley, en el que explica como desarrollar un compilador llamado *BeGone*. Sin embargo debido a algunas extensiones de este compilador que se han realizado además de la completa implementación de las funcionalidadas requeridas por el curso, he decidido llamarlo GoneFSR.

En las siguientes subsecciones analizaremos cada parte que componen este compilador. Cuando compilas código en GoneFSR, el compilador pasa por varias etapas: análisis léxico, sintáctico, semántico, generación de código intermedio y generación de código *llvm* (Low Level Virtual Machine). Ese código *llvm* se transformará en un ejecutable usando CLANG, como último paso.

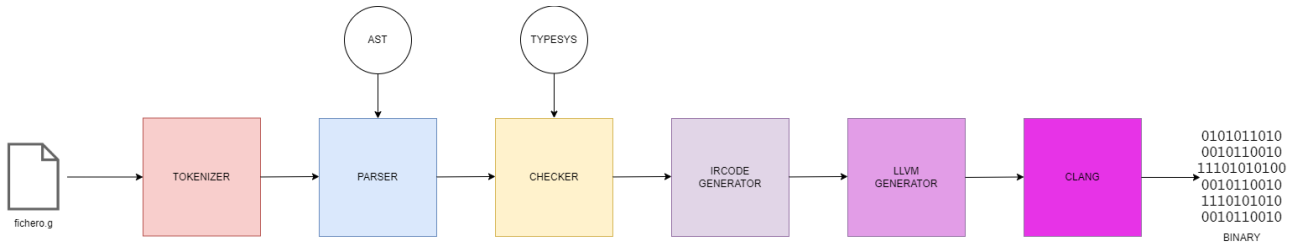

Figura 1: Estructura de GoneFSR

2.1. Análisis léxico

En un lenguaje cualquiera, el léxico se entiende como el conjunto de palabras que pertenecen a ésta. Estas unidades mínimas con significado propio del lenguaje son lo que se denominan *tokens*. Al módulo del compilador, encargado del *tokenizado* o análisis léxico se le suele llamar *Lexer*.

2.1.1. Lexer

La clase Lexer divide la cadena de caracteres de entrada (contenido del archivo ".g") en *tokens*. Por *token* en este lenguaje se entienden todo tipo de palabra clave (*keywords*), identificadores, literales, delimitadores como los paréntesis o el punto y coma. Para encontrar cada token recorre la cadena implementando un autómata finito que transiciona de estado en función del carácter que está recorriendo. Además, dentro del *Lexer*, es necesario especificar que se ignoren algunos caracteres como el espacio, las tabulaciones o el retorno de carro que harán que el *Lexer* avance hacia el siguiente carácter sin generar un token para estos últimos.

En GoneFSR puede haber cadenas de caracteres que no generen ni un solo token, esto se hace a través de los comentarios. Existen dos tipos de comentarios: de una sola línea y multilínea. La sintaxis de ambos es idéntica a la del lenguaje C, "// texto " y "/* texto */ ".

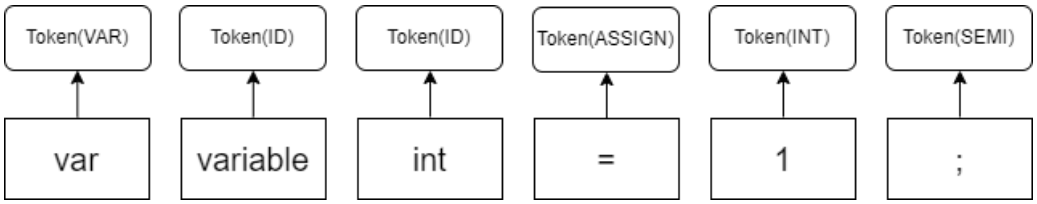

Figura 2: Ejemplo de tokenización

2.1.2. Implementación de un Lexer con SLY

La clase *GoneLexer* de este proyecto hereda de la clase *Lexer* de SLY. Esto nos facilita el desarrollo del *Lexer* de nuestro lenguaje, ya que automatiza toda la implementación del autómata finito. Y nos permite centrarnos en la definición de *tokens*, manejo de los caracteres ignorados, accionas asociadas a carácteres o patrones como los comentarios, y el manejo de errores. En el código 2.1 se incluye un ejemplo de la definición de un analizador léxico simple con la ayuda de SLY.

Código 2.1: Ejemplo de un analizador léxico comentado

```
from sly import Lexer

class SimpleLexer(Lexer):
    # Define tipos de tokens
    tokens = { NUMBER, PLUS, MINUS }
    # Define caracteres ignorados
    ignore = ' \t'
    # Define los tokens mediante Regex
    NUMBER  = r'\d+'
    PLUS    = r'\+'
    MINUS   = r'-'
    # Manejo de errores
    def error(self, t):
        print(f"Illegal character '{t.value[0]}'")
        self.index += 1 # Este es el atributo que maneja el iterador de la
            cadena de caracteres
```

Para abstraernos del código, explicaré lo esencial de la implementación de nuestro *Lexer*. La clave está en las expresiones regulares, más conocidas como REGEX, permitiendo especificar que *tokens* queremos encontrar sobre la entrada.

El orden en el que se definen los tokens tiene importancia, definir un token antes que otro en los que los dos comparten el comienzo del patrón, significa que se le dará prioridad al definido previamente, es por esto que todos los tokens del lenguaje se definen antes que los identificadores, ya que la REGEX de los identificadores engloba a palabras clave, como “const” o “return”.

Los nombres de tipos de dato como *int*, *string* o *char* son clasificados como tokens de tipo *IDENTIFIER*, y más tarde durante la fase de análisis semántico serán validados, para que no se puedan definir variables con tipos arbitrarios.

Dominar las expresiones regulares es complejo, hay muchas reglas que debes conoces, muchas caracteres que dependiendo del contexto significan una cosa u otra: grupos de captura, contenedores, “lookahead” con condiciones, caracteres comodín, etc...

Describir el funcionamiento de estos llevaría probablemente demasiado espacio así que, se pone un ejemplo sobre cómo distinguimos los enteros.

Código 2.2: Definición de un token

```
1 INTEGER = r'0x[a-f0-9]+|0o[0-7]+|0b[10]+|d+'
```

Para aquellos entendidos en expresiones regulares, sabrán que hay un detalle que me he saltado sobre los enteros. Y es que un entero puede escribirse sobre diferentes bases, se puede escribir, en decimal, en binario, en octal y en decimal. En la expresión regular se puede distinguir que se define como un entero la cadena '0x' seguida de al menos un (símbolo +) carácter 'a', 'b', 'c', 'd', 'e' y 'f' o de un dígito del cero hasta el nueve. Con esta configuración no se aceptaran como caracteres hexadecimales aquellos escritos con letras mayúscula, esto es una decisión arbitraria, ya que sería fácil admitirlos. Con el carácter '|' especificamos un "or" en la expresión es decir que también pueden admitirse escritos en octal (comenzando con '0o' y seguidos de al menos un dígito del cero al siete), escritos en binario (comenzando con '0b' y seguidos de al menos un dígito del cero al uno), o escritos en decimal que es lo que quiere decir la subcadena 'd+', es decir al menos un dígito del cero al nueve.

Para cada uno de los *tokens* del lenguaje hay una expresión regular que lo describe, para consultar todas las expresiones regulares, se encuentran en el apéndice A.

2.2. Análisis sintáctico (*parsing*)

La gramática de un lenguaje estudia la manera en la que se organizan y se combinan los elementos del mismo. En nuestro caso los elementos del lenguaje serán los tokens, y en esta fase se reordenaran en una estructura en árbol, que les dará contexto sintáctico.

2.2.1. Tipos de datos

En GoneFSR, hay cinco tipos distintos de datos.

- Booleanos: true o false
- Números enteros (*int*) de 32bits
- números decimales (*float*): este tipo de dato normalmente se asocia a números de coma flotante 32 bits, pero en este lenguaje en la fase final de compilación se mapean a double (64 bits) en las instrucciones llvm
- carácter (*char*): es simplemente un único carácter estando permitidos cualquier letra o número, además de caracteres especiales como \, \n, \", \xhh (siendo hh cualquier valor hexadecimal lo que permite representar cualquier carácter ASCII). Los calores *char* en este lenguaje siempre tendrán que estar recogidos sobre comillas simples.
- cadenas de caracteres (*strings*): son conjuntos de caracteres empezando y acabando por comillas dobles, pudiendo contener todo tipo de caracteres, escapados o no. Es importante decir que no se puede iterar sobre ellas, ni se puede realizar ningun tipo de operación con este tipo de dato.

2.2.2. Reglas básicas de la gramática

Todos las sentencias en GoneFSR, deben acabar en ";", como por ejemplo en lenguajes como C, o Java. En este lenguaje existen variables mutables e inmutables, para las inmutables o constantes no es necesario especificar un tipo de dato ya que se infiere del propio dato.

La sintaxis para constantes sería

```
const ID = value;
```

En el caso de las variables mutables si se requiere que el usuario especifique un tipo de dato en concreto de los cuatro disponibles. La sintaxis sería

```
var ID datatype = value;
```

La sentencia condicional *if*, se define con una sintaxis ampliamente extendida entre otros lenguajes.

```
if (condition) {
    statements
}
```

Para escribir un "if else".

```
if (condition) {
    statements
}
else {
    statements
}
```

En caso de querer definir un bucle, GoneFSR, tan solo cuenta con el bucle "while"por simplicidad, que se escribiría con la sintaxis.

```
while (condition) {
    statements
}
```

Con estas estructuras definidas nuestro lenguaje ya es *Turing completo* lo que quiere decir que se puede escribir cualquier algoritmo.
Ya solo nos queda por ver como se declaran las funciones. En GoneFSR las funciones se escriben tal que así.

```
func function_name (*arguments) return_datatype {
    statements
}
```

En GoneFSR, hay tres funciones *built-in*, es decir las puedes llamar sin necesidad de importar nada, "print","coder" y "decoder". Estas funciones tienen una sintaxis especial, ya que no usan paréntesis para recoger sus argumentos. Por ejemplo, para usar la función "coder".

```
coder "<path-inputfile>" "<path-outputfile">;
```

2.2.3. Tipos de operaciones

Las operaciones binarias disponibles en el compilador tanto para enteros como para decimales son: suma, resta, multiplicación y división. Siguiendo el estándar cada una de ellos se asocia con los símbolos +, -, *, /, respectivamente.

El compilador cuenta con las siguientes operaciones condicionales para después con las sentencias condicionales ("*conditional statements*") poder bifurcar el código, menor que ($<$), mayor que ($>$), menor o igual que ($\leq$), mayor o igual que ($\geq$), igual que ($==$) y no igual que ($!=$), todas estos operadores funcionan tanto para int como para float. Las comparaciones entre booleanos son las siguientes: igual que ($==$), no igual que ($!=$), "y" (& &) y "o" ($||$).

El símbolo + y -, también actúan como operadores unarios para int y float, para especificar cuando un número es positivo o negativo. Para lo booleanos existe el operador unario !, el cual sirve para negar el valor de una variable booleana.

2.2.4. Reglas avanzadas de la gramática

- Ninguna variable ya sea mutable o inmutable puede tener como identificador el nombre de uno de los tipos del lenguaje.
- No se pueden hacer cadenas con operadores de comparación como por ejemplo $0 <= j <= 5$, si no que la manera correcta de escribir lo mismo sería $0 <= j$ & & $j <= 5$.

2.2.5. Parsing

En el *Parser* establecemos las reglas de nuestra gramática. Nos basaremos en la notación BNF (Backus Naur Formalism) para definir nuestra gramática libre de contexto. Una vez definida la gramática, el *Parser* creará un AST (Abstract Syntax Tree) partiendo de los tokens generados por el *Lexer*, esta es una estructura jerárquica en forma de árbol, el cual representa las relaciones entre sentencias, expresiones y operadores como nodos.

Una gramática libre de contexto puede ser definida como una tupla (V, Σ, R, S) donde V es un conjunto no terminales, Σ es un conjunto de símbolos terminales, R es un conjunto de producciones o reglas que asocian cada no terminal a una cadena s donde $s \in (V \cup \Sigma)*$. Y S es un símbolo terminal

perteneciente a V, que es el símbolo sobre el que se comienzan aplicar las reglas. En nuestra gramática libre de contexto Σ son los tokens generados por el *Lexer*. Los símbolos no terminales V y las producciones son las que definirán nuestro lenguaje.

Representados en BNF los tres símbolos no terminales y sus reglas de producción asociadas de los que derivan todos los demás son (siendo *program*, S):

program → statements | λ
statements → statements statement | statements
statement → const_declaration | var_declaration | assign_statement | if_statement
| while_statement | return_statement | print_statement | coder_statement | decoder_statement

Podéis consultar la gramática totalmente desarrollada en el apéndice B. Esta gramática describe las reglas gramáticas básicas descritas previamente en esta sección. Es decir, partiendo de S se va rellenando el AST hasta que se consigue una estructura que representa la estructura gramática del programa, esta técnica se le llama *parsing* recursivo descendente.

En la práctica se definen funciones para cada regla gramatical y cada función llama a las reglas inferiores (funciones) hasta que se llega al nivel más bajo. En ese nivel se comparan los tipos de tokens si coincide, se acepta como el *parsing* correcto. Si no, se retrocede en el nivel de reglas hasta que coincide con alguno de los tipos. Si el token no coincide con ninguno de los tipos, se lanza un error.

Esta etapa se implementa en dos clases, una de ellas es la clase parser.py que hereda directamente de la clase *Parser* de SLY, la otra clase, es la encargada de definir las clases de cada nodo del AST, ast.py.

2.2.6. Precedencia

"A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be ambiguous." de Aho et al. [2006].

Es necesario definir una precedencia en el *Parser* para que no haya ambigüedades a la hora de crear el AST. Ya que se pueden generar dos AST distintos pero igual de válidos para nuestra gramática (en la figura 3 se trata todo como un *string*, esto es porque en el libro se habla de un lenguaje distinto, pero hace su función ilustrando el concepto):

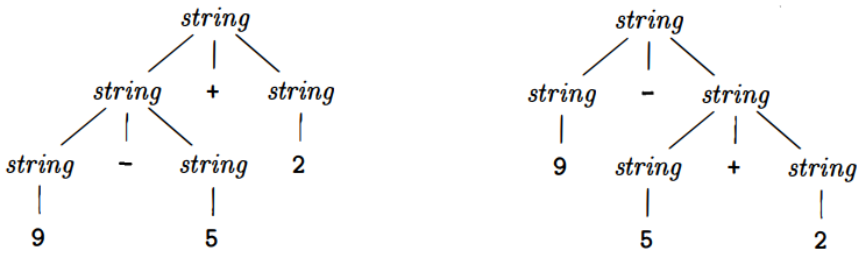

Figura 3: Ejemplo para 9 - 5 + 2 de Aho et al. [2006]

Para solucionar este problema definiremos la precedencia (que símbolo es evaluado primero) como una tabla en la que las filas están ordenadas de menor a mayor prioridad. En esta tabla habrá una tercera columna que será la asociatividad, que determina que operador es evaluado primero dentro del mismo rango de operadores.

Nombre	Operadores	Asociatividad
Lógicos	$\|\|$, &&	izquierda
Comparación	$>, <$	no asociativos
Comparación 2	$\leq, \geq, ==, \neq$	izquierda
Terms	$+, -$	izquierda
Factores	$*, /$	izquierda
Negación	!	derecha

Cuadro 1: Tabla de precedencia GoneFSR

2.2.7. Clases nodos del AST

Cada nodo del AST se representa por una clase, cada una de estas hereda de otra clase superior que clasifica el nodo entre cuatro posibles categorías: *Statement* (sentencia), *Expression*, *Datatype* y *Location*, este último se refiere a valores guardados en memoria como variables, constantes o funciones.

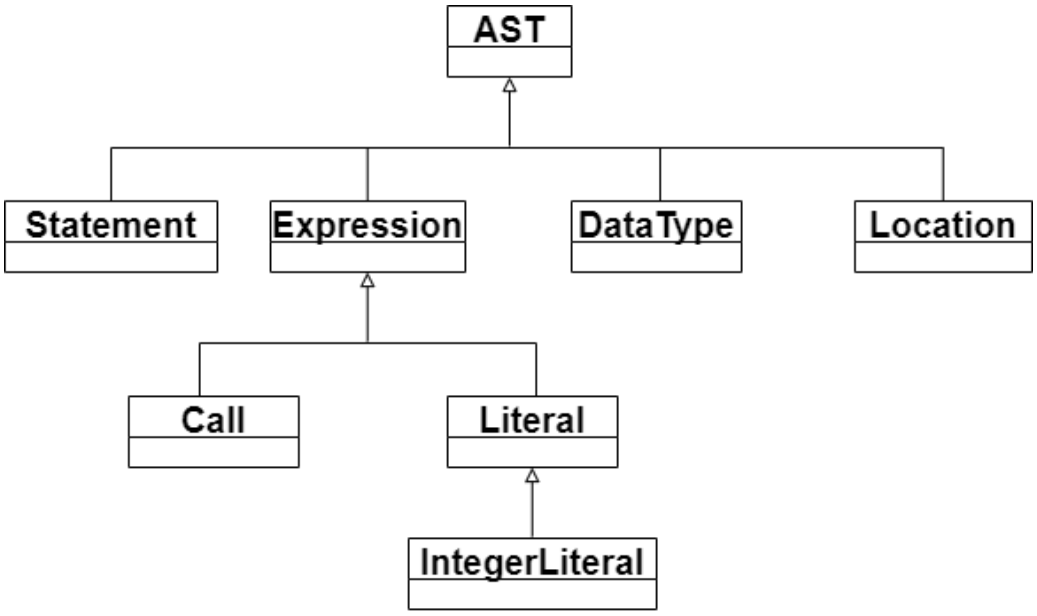

Figura 4: Diagrama UML parcial de la herencia entre las clases de los nodos del AST

A la hora de definir los nodos lo importante es representar las asociaciones entre nodos con otros nodos. Por ejemplo el nodo "ConstDeclaration" tiene como atributos un nombre de tipo "string" y un valor de tipo "Expression", o el nodo "BinOp" (Binary Operation) que tiene como atributos el tipo de operación, el operando izquierdo, que será una instancia de la clase "Expresssion" y el operando derecho, otra instancia de esa misma clase.

Todo esto se entiende mucho mejor con el código de 2.3 de un programa en GoneFSR. Es un ejemplo muy sencillo, pero expresa bien el proceso de conversión de código fuente a AST.

Código 2.3: Ejemplo de programa en GoneFSR

```
1 func main () int {
2     var b int = 1;
3     if 1 < 3 {
4         b = 7;
5         print b;
6     }
7     return 0;
8 }
```

En la figura 5 está el AST asociado generado por el *Parser* en base a ese código. En la imagen no está representado pero el nodo BinOp tiene otro atributo llamado "op" que almacena el tipo de operación binaria a realizar en este caso sería igual a "<", esto es útil para distinguir entre operaciones binarias.

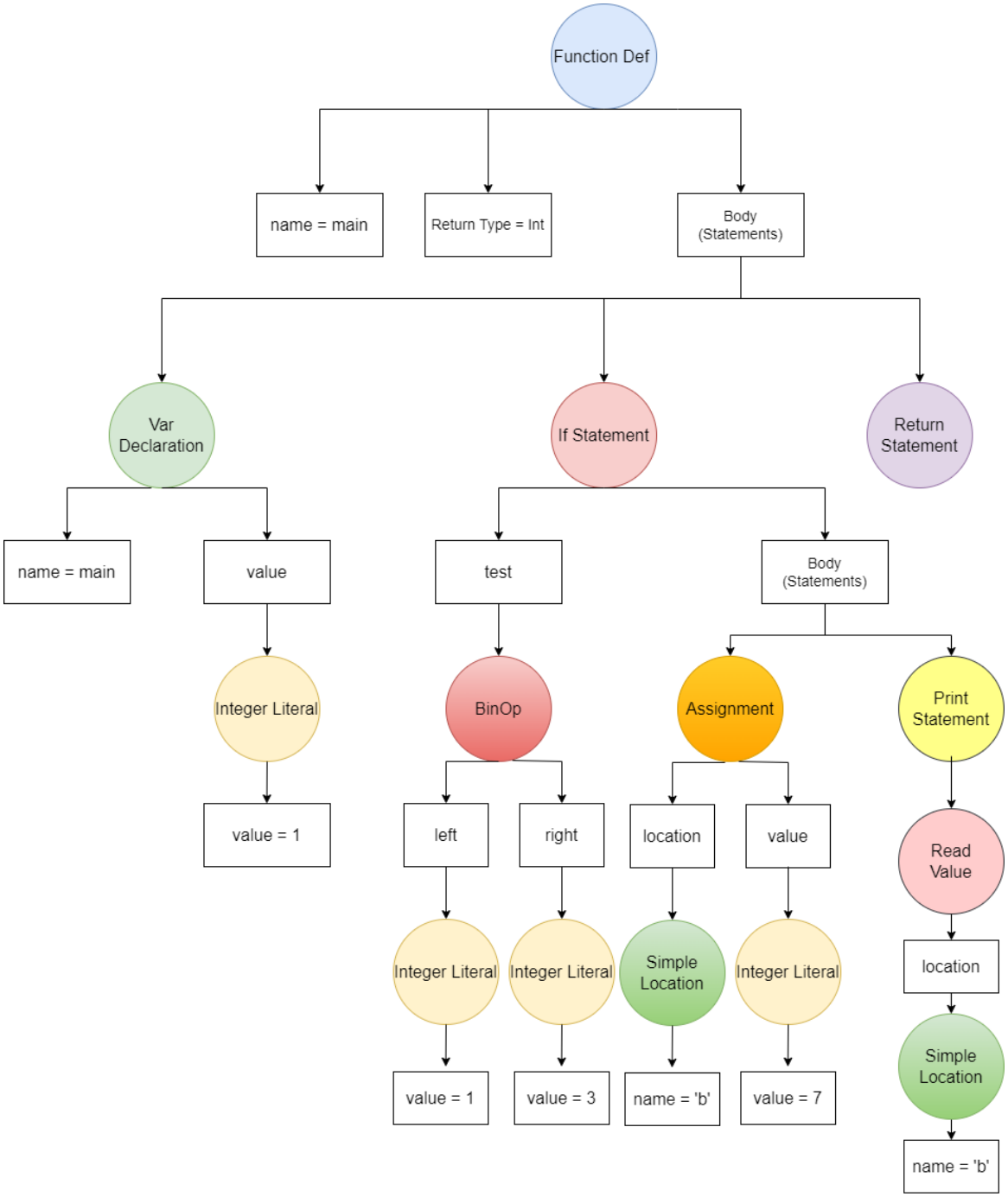

Figura 5: AST del código

2.3. Análisis semántico

El análisis semántico se asegura de que el árbol sintáctico cumpla las condiciones suficientes para pasar a la etapa de código intermedio. Esta etapa debe dar feedback al programador de las cosas que debe corregir. Pues si el analizador semántico detecta que alguna expresión, declaración, inicialización o sentencia sinsentido, lanzará un error y no generará código.

2.3.1. Recorriendo el AST

Las clases encargadas del análisis semántico y de la generación de código intermedio necesitan recorrer el árbol, para ser capaces de realizar este recorrido, heredan de una clase *NodeVisitor*, que está basada en una clase muy parecida del modulo de python *ast.NodeVisitor*. Esta clase permite definir métodos asociados a cada tipo de nodo del árbol, los cuales a su vez contendrán llamadas a los métodos de los hijos de ese nodo. Esto provocará que de manera recursiva se recorran todos los nodos.

En el caso del *Checker*, clase encargada del análisis semántico, cada método define las características que se quieren validar de ese nodo. Por ejemplo, que en una comparación se comparen dos datos del mismo tipo.

2.3.2. Funcionamiento *Checker*

La clase *Checker* cuenta con unas tablas de símbolos durante el análisis que le permiten recordar que símbolos han sido declarados, en mi implementación divido esta tabla de símbolos en tres subtablas: una para símbolo globales, otra para símbolos locales y una última para las funciones.

A la hora de definir el método de un nodo, es importante el orden en el que visitas los nodos, ya que hay casos en los que se deben visitar primero los hijos, para después poder analizar los tipos de estos. Como en el caso de las operaciones binarias, que toman dos operandos como parámetros.

En el caso de las localizaciones (“locations”) es necesario definir un atributo especial antes de visitar el nodo *Location* para saber si se refieren al símbolo en modo escritura o lectura.

Cada vez que se visita un nodo de definición de una función el *Checker* debe asegurarse de que la tabla de símbolos locales se vacíe para que los *scopes* se respeten.

Para realizar todas estas comprobaciones la clase *Checker*, se apoya en un módulo typesys.py donde estan definidos los tipos de datos incluidos en GoneFSR y las operaciones permitidas para cada tipo tanto unarias como binarias.

2.3.3. Validaciones del análisis semántico

El *Checker* se encarga de que no se incumplan ninguna de estas restricciones:

- No se permite definir funciones anidadas dentro de otras funciones.
- Todo código escrito fuera de una función se considera global y se ejecutará en una función especial “*premain*” (inicialización de variables globales).
- Cada función tiene su propio *scope*, por lo que cada función tiene su propia tabla de símbolos.
- Las funciones en GoneFSR siempre tienen que retornar un tipo de dato, ya que no existe el tipo de dato void como en C.
- Todos los flujos de una función deben retornar.
- No se permite redefinir funciones ya definidas.

- Todo código escrito después de un *return* se notificará como un error, ya que es código deprecado.
- los símbolos estén definidos antes de que se les haga referencia, ya sea de lectura o de escritura, en el caso de variables mutables.
- Las valores asignados a variables mutables deben coincidir con el tipo declarado para estas.
- Las comparaciones siempre deben de ser entre valores del mismo tipo de dato, ya que en este lenguaje no existen los *castings*.
- En las estructuras de control de flujo, las expresiones siempre deben ser del tipo booleano.
- Si se intenta cambiar el valor de una constante, se notifica como error.
- No se nombre ningún símbolo con el nombre de un tipo de dato.
- Toda comparación debe resultar en un valor de tipo booleano.
- Se respeten los tipos de los parámetros de una función, ya que estos deben coincidir con los argumentos de una posible llamada a esta función
- Se respete la aridad (número de parámetros) de las funciones.

En esta fase se realizan más validaciones, pero las listadas son las más relevantes.

2.3.4. Validación de retorno de flujos en una función

Me gustaría remarcar el sistema para controlar que todos los flujos (*flows*) de una función retornen, y en caso de que alguno de ellos no retorne lanzar un error / excepción. En una función se pueden dar dos casos, el primer caso es que la función cuente con un *return* incondicional al final de la misma, por lo que el fallo estaría solventado. Y el segundo caso, en el que la función no cuenta con un *return* al final de la misma, por lo que hay que verificar que cada uno de los flujos de la función terminan con el *return* correspondiente. ¿Cómo podemos asegurarnos de que todos los flujos terminan?

En GoneFSR, el problema esta solucionado de la siguiente manera. Primero se recorren uno a uno los *statements* del *body* de la función y en caso de que haya un *return*, lo guardas como verdadero en una variable booleana; en caso de que no haya un *return* en el *main flow* de la función, ocurrirá lo siguiente, durante el recorrido de cada *statement* de la función cada vez que se ha pasado por una bifurcación del flujo ya sea un *if*, un *if else* o un *while*, se ha añadido un true o un false en una lista global del *Checker*. Al terminar de recorrer la función, te aseguras de que todos los booleanos de esa lista sean verdaderos, en caso contrario si además no se ha encontrado un *return* en el flujo principal salta el error. Evidentemente, cada vez que se visita el nodo de la definición de una función se vacía la lista global de booleanos junto a la tabla de símbolos locales.

2.4. Generación de código intermedio

En Python con el módulo *dis* puedes conseguir el código intermedio de cualquier bloque de código. Estudiando este código puedes conseguir un entendimiento de como funciona realmente Python, entendiendo cosas como que Python debe evaluar un símbolo cada vez que se hace uso de este. Entender detalles como este de nivel más bajo, puede ayudar a la hora de escribir optimizaciones para programas de alto nivel.

Este código intermedio tiene muchas similitudes y en ocasiones durante el desarrollo de esta parte del proyecto, me he fijado en como se traducen algunos bloques de código a código intermedio en Python para luego imitar el comportamiento en mi compilador.

En este paso se suelen hacer cambios para que el código funcione en cualquier máquina. Un buen ejemplo es que a pesar de que en GoneFSR existen los booleanos, en este paso los booleanos se convierten en enteros ya que hay máquinas que pueden no contar con este tipo de dato.

En GoneFSR, nuestro código intermedio se parecerá al código intermedio de Python tan solo en forma, ya que nosotros trabajaremos con menos instrucciones y no haremos tantas optimización en tiempo de compilación. Además de que las instrucciones intermedias de Python (también se les llama bytecode) se interpretan directamente por la PVM (Python Virtual Machine), ya que en Python no se hace uso de llvm por cuestiones técnicas.

2.4.1. Transformación de AST en código intermedio

La clase *GenerateIRCode* hereda de *NodeVisitor*, así que en cada método se define como traducir ese tipo de nodo a código intermedio. Una vez definidos todos los métodos, se recorre el AST traduciendo cada nodo. Como resultado final, esta fase del compilador creará un diccionario que tiene como llaves el nombre de las funciones (realmente, la llave es una tupla con más valores como el tipo de los parámetros de la función) y como valor la lista del código intermedio de la función. Ese diccionario se le pasará al generador de código *llvm* que será el encargado de dar el último paso para generar el ejecutable.

Para definir el código intermedio de nuestro compilador (al que también nos referiremos como IR para no confundirlo con el código *llvm*), se ha escogido un estilo de código intermedio llamado *Single Static Assignment* (SSA) este paradigma se define por usar cada variable una sola vez. En la práctica, esto implica que cada vez que se utiliza un registro, se suma uno a un contador que a su vez es el índice del siguiente registro. Una instrucción de código IR puede tener los siguientes posibles argumentos: registros como operandos (“R” + número, p.ej, “R2”), caracteres (p. ej. para distinguir entre el tipo de comparación), *strings* y por último números enteros o flotantes. Ahora veamos la estructura de nuestras instrucciones de código intermedio.

```
mnemónico param1, param2, …, paramN
```

Esta estructura se cumple para todas las instrucciones. Aunque no todas las instrucciones tienen un *mapeo* directo con un mnemónico. Por ejemplo para negar un valor booleano, esto se traduce en dos instrucciones mover un uno a un registro y hacer un *XOR* entre el valor booleano y ese registro. Otro ejemplo es el calculo de un número negativo, en cuyo caso se traduce como el resultado de 0 menos ese número en positivo.

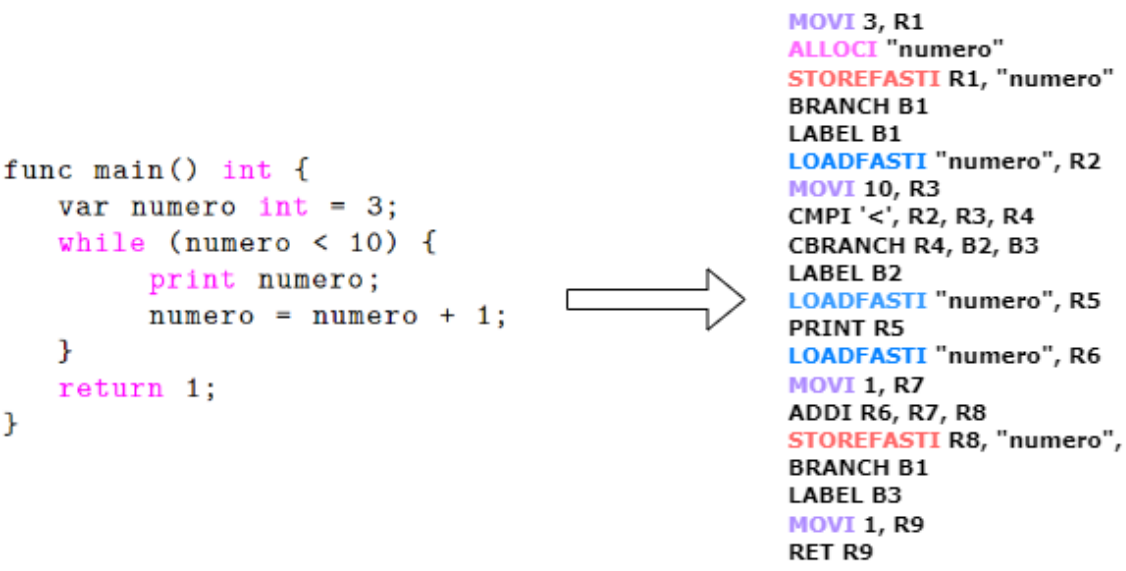

Figura 6: Ejemplo traducción a código intermedio

2.4.2. Instrucciones de código intermedio

Existen instrucciones de código intermedio relacionadas con el manejo de variables. En concreto existen cuatro instrucciones: `MOV`, `VAR`, `ALLOC`, `STORE` y `LOAD`. Estas instrucciones existen para los 4 tipos de datos de GoneFSR. Se distingue para que tipo de dato se aplica la instrucción por el último carácter del mnemónico que puede variar entre: I (entero), F (flotante), B (Byte o Char) y S (string).

- `MOV`: coloca un literal sobre una variable temporal o registro.
- `VAR`: declara una variable global del tipo pertinente
- `ALLOC`: declara una variable local del tipo pertinente
- `LOAD`: carga el valor de una variable global en una variable temporal, en su variante `LOADFAST` hace lo mismo pero en una variable local
- `STORE`: guarda un valor en una variable global, en su variante `STOREFAST` hace lo mismo pero en una variable local

La distinción entre instrucciones de `LOAD` y `STORE` con sus respectivas variantes para variables locales se debe a que en la etapa de generación de código *llvm*, las variables locales y las globales se guardan en tablas de símbolos distintas.

Para los enteros y los flotantes es necesario definir instrucciones de código intermedio aritméticas, estas se aplican para los distintos tipos en función del sufijo del mnemónico, pudiendo ser I (entero) o F (flotante).

- `ADD`: suma el valor de dos registro y guarda el resultado en un tercero
- `SUB`: calcula la diferencia entre el valor de dos registro y guarda el resultado en un tercero
- `MUL`: multiplica el valor de dos registro y guarda el resultado en un tercero
- `DIV`: divide el valor de un primer registro entre el valor de un segundo registro y guarda el resultado en un tercero

- `CMP`: toma como argumento el tipo de comparación, un primer registro, un segundo registro, y un tercer registro donde guardar el resultado booleano de la comparación

También existen instrucciones para los operadores booleanos: `AND`, `OR` y `XOR`, todas ellas toman dos variables temporales como parámetro y un tercer parámetro que se refiere al registro donde guardan el resultado.

Por otro lado, las instrucciones intermedias de control de flujo son:

- `LABEL`: tomo como argumento un nombre, marca un punto en el flujo de las instrucciones de código IR al que después se puede retornar con instrucciones de salto.
- `BRANCH`: toma como argumento una label a la que salta incondicionalmente.
- `CBRANCH`: toma como argumento un valor booleano, una label a la que salta si el valor booleano es verdadero y otra label a la que salta si el valor booleano es falso.
- `CALL`: toma como argumento el nombre de una funcion y tantos argumentos como parámetros tenga esa función.
- `RET`: toma como argumento una variable temporal, la cual define el valor retornado por la función.

2.5. Generación de código LLVM

LLVM es un conjunto de herramientas para el desarrollo tanto de frontend como de backend de compilación. Se define como frontend del compilador, el software encargado del análisis sintáctico o *scanning*, el análisis semántico y de al menos generar una representación intermedia del código (en nuestro caso, la generación de código intermedio se hace en dos etapas). Por otro lado, el backend se define como la parte que optimiza y genera el código máquina optimizando y puliendo las instrucciones para que puedan ejecutarse en ese entorno en concreto. La ventaja más evidente de separar el proceso de compilado en estas dos etapas es que, en lugar de hacer un compilador personalizado para cada lenguaje y para cada arquitectura, programas un compilador frontend para cada lenguaje y después un backend para cada arquitectura. Hay algunos compiladores que realizan las dos etapas como por ejemplo *gcc*.

2.5.1. Ejemplos llvm

Veamos un ejemplo básico de uso del modulo en el código 2.4 de *llvm* para escribir código intermedio llvm. Siendo el código de python.

Código 2.4: Ejemplo de generación un pequeño bloque

```
1  from llvmlite.ir import Module, Function, FunctionType, IntType, IRBuilder,
       Constant, DoubleType, GlobalVariable, VoidType
2  import llvmlite.binding as llvm
3
4  # Inicializacion clases llvm
5  mod = Module('ejemplo')
6  main = Function(mod, FunctionType(IntType(32), []), name="main")
7  block = main.append_basic_block('entry')
8  builder = IRBuilder(block)
9
10 # Inicializacion variable global
11 global_operando = GlobalVariable(mod, IntType(32), 'globalop')
12 global_operando.initializer = Constant(IntType(32), 10)
```

```
valor_global_operando = builder.load(global_operando)
# Inicializacion variable local
local_operando = builder.alloca(IntType(32), name="localop")
builder.store(Constant(IntType(32), 5), local_operando)
valor_local_operando = builder.load(local_operando)
# calculo de resultado
resultado = builder.add(valor_local_operando, valor_global_operando,
    "resultado")
builder.ret_void()
print(str(mod))
```

Y el código llvm que obtenemos una vez ejecutado está en el Código 2.5.

Código 2.5: Salida del código anterior

```
; ModuleID = "ejemplo"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define i32 @"main"()
{
entry:
  %".2" = load i32, i32* @"globalop"
  %"localop" = alloca i32
  store i32 5, i32* %"localop"
  %".4" = load i32, i32* %"localop"
  %"resultado" = add i32 %".4", %".2"
  ret void
}

@"globalop" = global i32 10
```

El código llvm es bastante ofuscado e incómodo de leer, pero es portable. Esto deriva de las definiciones anteriores de frontend y backend en un compilador. Ahora que tenemos el código intermedio cualquier backend ya sea *llc* (el backend más común para llvm), CLANG, o cualquier otra herramienta que compile *llvm*, podría satisfacer nuestro objetivo de llegar hasta código máquina.

Vamos a analizar, línea a línea, que está pasando en ese código *llvm*. En primer lugar, el sitio donde definimos el código se le llama módulo y se podría entender como un "fichero". Dentro del módulo se definen funciones, y dentro de estas funciones se definen bloques. Estos bloques se utilizan a la hora de implementar sentencias condicionales, ya que nuestro código deberá saber a que bloque debe "saltar".

Ya tenemos nuestro módulo, nuestra función y nuestro bloque donde escribir código. Bien, la siguientes dos instrucciones en el *script* de Python, inicializan una constante. Si ahora nos fijamos en el código llvm, podemos ver que la constante inicializada se encuentra fuera del *scope* de la función *main*. La siguiente línea carga el valor de la constante en una variable temporal (%".2"). Después se declara una variable local como *int*, a esta variable se le da un valor de 5. Ahora i32 tiene un asterisco (*) detrás, esto quiere decir que se trata de un puntero, , es decir estamos guardando un entero de 32 bits en la posición de memoria de "localop", cargamos el valor de "localop" en otra variable temporal, para luego sumar ambas variables temporales guardando la suma en una variable local con nombre resultado".

Habiendo visto un ejemplo simple, veamos un ejemplo más complejo en concreto el código *llvm* de la función *main* del ejemplo de la sección 2.2.7.

Código 2.6: Traducción *llvm* del ejemplo de la sección 2.2.7

```
define i32 @"main"()
```

```
{
entry:
  call void @"premain"()
  %"b" = alloca double
  store double              0x0, double* %"b"
  store double 0x3ff0000000000000, double* %"b"
  %"R2" = load double, double* %"b"
  %"R4" = fcmp olt double %"R2", 0x4008000000000000
  br i1 %"R4", label %"B1", label %"B2"
B1:
  %"R7" = fadd double 0x401c000000000000, 0x4020000000000000
  store double %"R7", double* %"b"
  %"R8" = load double, double* %"b"
  call void @"_print_float"(double %"R8")
  br label %"B2"
B2:
  ret i32 0
}
```

La llamada a *premain* es la primera instrucción porque, como ya comenté todo código escrito fuera de *main* se ejecuta en *premain*, en este caso no hay código fuera de *main*, así que la función está vacía (el código llvm es en realidad, un poco más largo, donde se puede ver que *premain* está vacío).

Los detalles distintos respecto al ejemplo simple que me quedan por explicar son los saltos y bloques. La traducción del *if*, esto se hace a través de la instrucción "br" que utiliza un tipo "i1", un *integer* de un solo bit, que es lo mismo que un booleano, para ejecutar un salto condicional. Los otros dos términos de la instrucción son instrucciones *label* con dos etiquetas distintas "B1" y "B2". Pues bien, si ese entero de un solo bit es 1, se saltará al bloque "B1" y en caso contrario a "B2". El valor de ese booleano se determino en la operación de comparación que se guarda en el registro temporal R4.

2.5.2. De código intermedio a código LLVM

La clase GenerateLLVM es la encargada de traducir el código intermedio a código LLVM. Esta clase no hereda de *NodeVisitor* ya que no hace falta recorrer el AST, si no que ahora la tarea se resume en recorrer las instrucciones intermedias de cada función y generar el código *llvm* apropiado en cada una de estas funciones.

Para poder generar el código *llvm*, nada más se instancia la clase *GenerateLLVM* se inicializa un módulo *llvm* en el objeto. Después se recorren todas las funciones retornadas por el módulo de generación de código intermedio (las llaves del diccionario retornado), y por cada funcion se llama a un método *generate_function* de la clase *GenerateLLVM* para escribir la definición de las funciones en el módulo *llvm*. Después de inicializar todas las funciones en el módulo *llvm*, se pasa a generar el código de cada una de estas funciones, llamando al método *generate_code*. Es importante que estos pasos se hagan orden pues dentro de las funciones puede llamarse a otras funciones previamente definidas.

El método *generate_code* internamente lo que hace es llamar a métodos de la propia instancia del objeto con nombre *emit_MNEMO*, donde *MNEMO* se sustituye por cada mnemónico del código intermedio, como por ejemplo *emit_ADDF* o *emit_VARI*. Pasando los argumentos de la instrucción de código IR a su método correspondiente podemos generar dentro del módulo llvm el código asociado. Por ejemplo *emit_VARI*.

```
def emit_VARI(self, name):
        var = GlobalVariable(self.module, int_type, name=name)
        var.initializer = Constant(int_type, 0)
        self.vars[name] = var
```

El código se traduce completamente a código *llvm* sobre el módulo de la clase *GenerateLLVM* cuando se ha finalizado este proceso para todas las instrucciones IR.

2.6. Código llvm a código máquina

CLANG actúa como backend de compilación en nuestro proyecto, ya que no se han implementado las etapas de backend: generar el código máquina en función de la arquitectura o la etapa de enlazado. Hay que resaltar que CLANG emplea las librerías internas de LLVM para poder realizar el procecso de backend, ya que el propósito de CLANG es actuar de frontend, pero en este caso esa utilidad no nos hace falta.

En la práctica, tan solo tendremos que ejecutar CLANG y pasarle como argumentos tanto el archivo con el código intermedio llvm, como el archivo con las funciones "built-in", gonert.c. Con esto, CLANG se encargará de la generación de los ficheros objeto y de la fase de enlazado. Por último, nos escribirá un ejecutable en el directorio donde hayamos ejecutado el comando que será, en efecto nuestro programa.

2.7. Implementación función fsr

Para implementar las funciones de codificación y decodificación del formato de archivo de la sección 3.4. Ha sido necesario hacer las siguientes modificaciones en el compilador.

1. **Añadir token *string***: GoneFSR no contaba con datos *string* así que hubo que añadir en el *Lexer* un token de tipo *string*.
2. **Añadir keywords *coder* y *decoder***: para identificar las nuevas funciones fue necesario añadir estas dos *keywords* al *Lexer*.
3. **Nuevas clases de nodo del AST**: escribir una nueva clases para las sentencias *coder* y *decoder*, ambas heredaran de la clase *Statement*. También tendrán ambas como atributos tanto el input path como el output path de los archivos que quieran usar como argumentos de tipo *string*.
4. **Cambios en el *Parser***: introducir nuevas reglas gramaticales para estas dos nuevas funciones.
5. **Análisis semántico**: en la fase de análisis semántico obligar a que los parametros de las dos funciones sean del tipo *string*.
6. **Código intermedio**: crear mnemónicos en el conjunto del código IR para las dos nuevas funciones, además de las instrucciones de código intermedio para trabajar con *strings* (LOAD, STORE y MOV).
7. **Compilación e instalación de los ejecutables *coder* y *decoder***: En primer lugar, con la herramienta *pyinstaller* se han compilado las funciones escritas originalmente en Python para obtener dos ejecutables uno para el codificador y otra para el decodificador. Para que puedan usarse dentro de GoneFSR, se deben instalar estos dos ejecutables en el sistema, la ruta correcta para instalarlos es *"/usr/local/bin"*.
8. **Escritura de las funciones en la librería nativa *gonert.c***: Al igual que las funciones para impirimir caracteres, enteros y flotantes. También será necesario crear tanto una función para imprimir strings como una función para el *coder* que llame al ejecutable del sistema con los parámetros adecuados y una función igual para el *decoder*.
9. **Generación código llvm**: investigar y analizar el tratamiento de strings en llvm, y el paso de strings como argumento a funciones. Por último, añadir las nuevas funciones "built-in" al módulo *llvm* al igual que sus correspondientes métodos para generar el código *llvm* correspondiente.

Si bien no es la mejor de las maneras para implementar una función nativa en un lenguaje ya que llamar a un ejecutable desde la propia función no es lo más común (aunque esto se intenta aliviar con la función *execlp* de la librería estándar *unistd.h* que sustituye el proceso original). Es una implementación funcional en un proyecto didáctico.

La mejor manera de implementar esto hubiera sido transpilas el código de las funciones de decodificación y codificación de Python a C. Esto se puede conseguir con herramientas como *Cython*, ha habido intentos de transpilar el código pero requería un afinamiento manual que traspasaba la escala del proyecto.

En el repositorio una utilidad que instala los dos ejecutables en *"/usr/local/bin"* para que no haya ningún problema de dependencias en el uso de GoneFSR, utilidad.sh.

2.8. Pruebas unitarias del compilador

Utilizando el modelo incremental iterativo, se han hecho pruebas para cada paso en el desarrollo del compilador. Y se han desarrollado en orden cada una de las fases para poder corregir fases anteriores si se produjesen errores.

Procedimiento

Para cada etapa del compilador, el curso contaba con programas complejos como un programa que genera el fractal de mandelbrot, para probar el funcionamiento del compilador que se encuentran en programas, y tests unitarios para probar la funcionalidad más básica de cada una de estas etapas, podéis consultar estos en tests. Sin embargo, esta batería de tests no cubría todos los casos en los que el compilador debía de ser probado así que desarrolle mi propio conjunto de tests, el cual se encuentra en el directorio *custom tests*.

Si unimos el conjunto de *tests* del curso y los *custom tests*, abarcamos los siguientes tipos de pruebas, pruebas unitarias, para probar el funcionamiento de cada etapa de manera aislada, pruebas de integración que evalúan si la comunicación entre los distintos módulos es adecuada y pruebas de sistema, ya que hay *tests* que comprueban si ese programa en concreto se compila como es debido y genera el ejecutable esperado. Otra manera de clasificar estas pruebas sería dividirlas entre pruebas de caja negra (probando únicamente la funcionalidad, sin tener en cuenta la implementación) y pruebas de caja blanca (teniendola en cuenta).

En caso de no encontrar los errores en el código causados por los tests, se ha hecho uso del módulo de python *pdb* para configurar *breakpoints* y analizar el código hasta dar con las causas.

Automatización de pruebas

En el directorio automatización de testing podéis encontrar algunos *scripts* escritos en *bash* para crear dos archivos distintos, uno con la salida del compilador dado como ejemplo en el curso, el cual también tenéis en el directorio compilador ejemplo, y otro archivo en el cual escribía la salida de mi compilador. Utilizaba la comparación de estos dos archivos para saber si mi compilador estaba obteniendo los resultados correctos.

3 Investigación sobre registros de desplazamiento (FSR)

3.1. Linear Feedback Shift Register (LFSR)

El *Linear Feedback Shift Register* es un registro de desplazamiento en el que la entrada se calcula a través de una función lineal. Uno de sus usos es cifrar una secuencia binaria (a traves de la suma de un *keystream* a los datos en claro), o para generar una secuencia aleatoria de dígitos a partir de una semilla o estado inicial. A la función que describe el calculo de la siguiente entrada del registro también se la conoce como polinomio de *feedback*. Para ser más precisos teniendo una cadena binaria b.

$$b = b_0, b_1, ..., b_n$$

Siendo n el numero de bits del estado inicial del LFSR, se aplicará sobre esta cadena un xor sobre ciertas posiciones de tal manera que nos de como output el bit en la posicion 0, del nuevo estado de la cadena/secuencia. A las posiciones sobre las que se aplica el xor se les llama según la literatura científica *taps*.

Así que teniendo la secuencia b. Y por ejemplo, *taps* en las posiciones 0, 1 y 4. Siendo $b >= 4$. El siguiente estado de la secuencia se computará de esta manera

$$output = b_0 \oplus b_1 \oplus b_4$$

$$b = output, b_0, b_1, b_2, b_3, b_4, ..., b_{n-1}$$

Visualmente el proceso se aprecia más fácilmente la naturaleza del LFSR como proceso recursivo. Pongamos como ejemplo la secuencia "0100", y pongamos como *taps* la posición 0 y la posición 1.

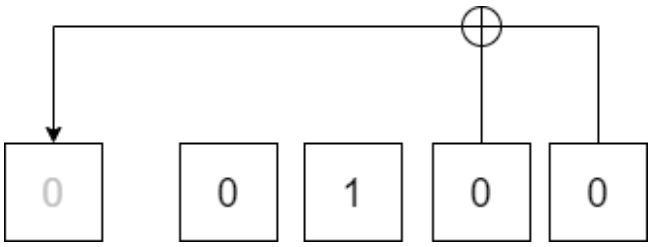

Figura 7: paso 0 ejemplo LFSR

En la figura, se aprecia como el XOR entre la posición 0 de la cadena binaria, y la posición 1, resulta en el bit que ocupará la posición 3 de la cadena. Todos los bits del estado anterior se desplazan una posición hacia la derecha eliminando el bit en posicion 0 del estado anterior. En la figura 8 se aprecia el resultado tras otro *step*.

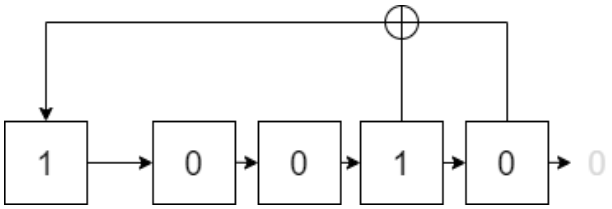

Figura 8: paso 1 ejemplo LFSR

El polinomio del LFSR del ejemplo, es un polinomio primmitivo ya que para que el registro vuelva a tener el valor de la semilla es necesario hacer $2^n - 1$ iteraciones, ese menos uno se debe a que evidentemente el registro no puede valer 0 ya que en ese caso sin importar el número de iteraciones siempre permanecería en 0.

Para concluir esta sección y esclarecer del todo este tema de los registro de desplazamiento, en la siguiente figura tenéis un pseudocódigo naive para implementar cualquier LFSR, aquí podéis consultar una implementación en python basiclfsr.py.

Algoritmo 1: Algoritmo lfsr(s, t, k)

Data: s = secuencia binaria, t = lista con las taps positions, k = numero de steps/iteraciones
Result: s después de k iteraciones
for $i = 0; i < k; i++$ **do**
  $output \leftarrow s[taps[0]]$;
  for $j = 0; j < len(t); j++$ **do**
    $output \leftarrow output \oplus s[taps[j]]$;
  end
  $s = output, s[len(s) - 1]$;
end

Cabe destacar, lo increíblemente simple que es, para lo complejos que pueden llegar a ser los output a los que da salida este algoritmo. Aunque existen técnicas para decodificar estas secuencias y obtener tanto el polinomio como el estado inicial.

3.1.1. Fibonacci y Galois

Hay dos tipos principales de registros de desplazamiento lineal. El que hemos explicado hasta ahora se le denomina LFSR de Fibonacci, aunque este no fuera el autor se le llama así por su relación con las recurrencias. Este tipo de LFSR puede ser de período máximo, es decir si está constituido por un polinomio primitivo, o puede describirlo otro polinomio que no sea primitivo, y entonces tener un período menor a $2^n - 1$.

El otro tipo es más difícil de explicar y cómo en las siguientes secciones no tiene especial impacto, tan sólo lo mencionaré, se trata del LFSR de Galois (sí, el del duelo y las cartas), en este registro de desplazamiento lineal algunos de los bits se desplazan y otros son el resultado de operaciones entre algunas de las posiciones del propio registro.

3.2. Algoritmo de Berlekamp-Massey

Antes de pasar al algoritmo sobre el que se central de traducción de una secuencia binaria a su FSR mínimo. Debemos explicar el procedimiento que toma como referencia, este es el algoritmo de *Berlekamp Massey*, que fue propuesto por dos investigadores de forma independiente. La propuesta

de Berlekamp [1966] fue ideada para decodificar códigos BCH (Bose-Chaudhuri-Hocquenghem). Estos códigos se utilizan para codificar información de manera redundante por si hubiera algún fallo en la transmisión o en el almacenado de los datos poder corregirlos a través de Forward Error Correction (FEC).

Por su parte, Massey [1969] propuso un algoritmo equivalente, que lo denominó *LFSR Synthesis Algorithm (Berlekamp iterative algorithm)* en el articulo original , el cual es capaz de encontrar la recurrencia lineal más corta para cualquier secuencia numérica finita. Esto tiene mucho que ver con la modificación de este algoritmo que utilizaremos en la siguiente sección para calcular la complejidad no lineal de un secuencia binaria. Además también es el culpable de que el cifrado a través de LFSR sea inseguro, ya que si conoces una parte del texto claro de tamaño suficiente puedes recuperar el *keystream* con el que se cifró.

Para explicar *Berlekamp Massey*, es necesario definir una recurrencia lineal. Dada una secuencia $s = \{s_0, s_1, \dots\}$ diremos que existe una secuencia de recurrencia lineal $c = \{c_1, c_2, \dots, c_n\}$, las cuales cumplen.

$$s_i = \Sigma_{j=1}^{n} c_j s_{i-j} \quad \text{para } i \geq n$$

Para $i \geq n$ tan solo quiere decir que es para todos los números fuera de los casos base ya que los casos base de la recurrencia lineal serán desde s_0 hasta s_n. Un ejemplo de recurrencia lineal muy típico es la sucesión de Fibonacci o su primo hermana la sucesión de Lucas.

El algoritmo Berlekamp Massey se basa en lo siguiente. Teniendo una secuencia s cualquiera y una lista con los coeficientes de la recurrencia lineal vacío, ir rellenando la lista de coeficientes según se recorre la secuencia s y en caso de que ocurra un fallo, aplicar una función correctora d. También hay que guardar copias de c y tener un índice de un fallo pasado, en el ejemplo veremos para que y por qué.

Veamos esto con un ejemplo sencillo, dada la secuencia $s = \{1, 3, 9, 15, 9, -81\}$ y la la lista $c = \{\ \}$, asumiremos que si c está vacío, c será igual a 0. Definiremos como pc (*past c*) al conjunto que nos servirá como copia de c, el cual comienza vacío como c. Y la variable donde guardaremos el índice al fallo pasado como f.

En la primera iteración:
$i = 0 \quad s_0 = 1$
$c(0) = 0 \neq s_0$
$f = i = 0$
Como da igual que valor añadamos como primer coeficiente a c ya que s_0 será caso base, seguiré el mismo proceder que en mi implementación, se añaden a c tantos ceros como iteraciones hayan pasado hasta el primer fallo en este caso la primera así que $c = \{0\}$.

En la segunda iteración:
$i = 1 \quad s_1 = 3$
$c(1) = c_1 * s_{1-1} = 0 \neq s_1$

Al ser el segundo fallo, ya hay que aplicar la función correctora d, el cálculo de esta siempre es igual. Primero hacer que d sea igual a $-1 * pc$, insertar un uno en la posición 0 de d, multiplicar d por $\frac{\Delta}{d(f+1)}$, siendo Δ igual a $s_i - c(i)$, finalmente debes insertar $i - f - 1$ ceros a la izquierda de la lista c. En esta iteración d sería igual a.

$d = \{1\}$
$d = \{1\} * \frac{\Delta}{d(f+1)} \quad \text{siendo} \quad d(i) = \Sigma_{j=1}^{n} d_j s_{i-j}$
$d(0+1) = 1 \quad \Delta = 3 \quad \frac{\Delta}{d(f+1)} = 3$

$d = \{1\} * \frac{\Delta}{d(f+1)} = 3$
Finalmente, $d = \{3\}$ ya que, $i - f - 1 = 0$, no se insertan ceros a la izquierda

Ahora sumamos esta función correctora d a c para obtener nuestra c, $c = c + d = \{3\}$. Hay que decir que ni pc ni f han cambiado su valor en esta iteración. Pasaremos directamente a explicar la cuarta iteración (ya que en la tercera no hay fallo) y no repetir los pasos, solamente explicaré el por qué en este fallo sí se actualizan pc y f.

En la cuarta iteración:
$i = 3 \quad s_3 = 15$
$c(3) = c_1 * s_{3-1} = 18 \neq s_3$

Tras ejecutar todos los pasos para el cálculo de d, queda $d = \{0, 0, -12\}$. Y ahora sí que actualizamos tanto f como pc.

$f = i = 3$
$pc = c$ c es la versión de c de $i = 2$, es decir, antes de sumar d

Esta actualización se debe a que esas variables deben cambiar cuando se cumple $i - len(c) > f - len(pc)$. Intuitivamente, diremos que pc guardará el c con menos coeficientes respecto a donde fallo.

Aquí está el pseudocódigo, podéis encontrar una implementación en Python escrita por mí en simplebm.py basado en uno de los códigos del artículo berlekamp massey.

Algoritmo 2: Algoritmo Berlekamp-Massey

Data: s secuencia númerica
Result: La recurrencia lineal mínima
$c \leftarrow [\,]$;
$pc \leftarrow [\,]$;
$f \leftarrow -1$;
for $i = 0; i < len(s); i++$ **do**
  $delta \leftarrow s[i] - \sum_{j=1}^{len(c)+1} c_{j-1} * s_{i-j}$;
  if $f == -1$ **then**
    $c \leftarrow resize(c, i+1)$;
    $f \leftarrow i$; `/* Para la primera vez */`
  else
    $d \leftarrow -pc$;
    $d.insert(0, 1)$;
    $df1 \leftarrow \sum_{j=1}^{len(d)} d_{j-1} * s[f+1-j]$;
    $coeff = \frac{delta}{df1}$;
    $d \leftarrow d * coeff$;
    $d \leftarrow$ `addzerostoleft(i-f-1, d);` `/* suma i-f-1 ceros a la izq */`
    $copyc \leftarrow c$;
    if $len(c) > len(d)$ **then**
      $c \leftarrow$ `addzerostoright(len(d) - len(c), d)`
    end
    $c \leftarrow c + d$;
    if $i - len(copyc) > f - len(pc)$ **then**
      $pc \leftarrow temp$;
      $f \leftarrow i$;
    end
  end
end

3.3. Non lineal Feedback Shift Register (NLFSR / FSR)

Uno de los objetivos de este proyecto ha sido intentar simplificar la función h resultante del algoritmo de Limniotis et al. [2007]. El paso previo ha sido analizar y dividir por partes el algoritmo hasta entenderlo completamente, esto ha sido de las cosas que más tiempo han consumido.

En esta sección, se tratará de explicar este algoritmo recursivo como a mi me hubiera gustado que me fuese explicado. Este procedimiento toma como base Berlekamp-Massey, el cual resuelve eficientemente el cálculo del LFSR mínimo, y lo adapta para conseguir un registro de desplazamiento no lineal (NLFSR) con un polinomio mínimo para una secuencia binaria dada. Se dice que un registro de desplazamiento es no lineal, si en su polinomio se utiliza tanto la operación XOR (suma) como la operación AND (multiplicacion).

Antes de describir el algoritmo, debo hacer un breve resumen sobre la notación que utilizaré durante esta sección para describir conceptos como una secuencia binaria cualquiera. Esta notación está basada en la del artículo original.

Para definir un *minterm* usaremos la notación, $\underline{x_b} = x_1^{b1}, ..., x_n^{bn}$, donde $b = (b_1, ..., b_n) \in \mathbb{F}_2^n$, siendo $x_i^0 = x_i'$ y $x_i^1 = x_i$, por ejemplo, el *minterm* de la secuencia 011 es $x_1' x_2 x_3$. Para las cadenas o secuencias, usaré la siguiente notación, diremos que una secuencia binaria de N elementos se escribe y^N. Si queremos describir un segmento de esa secuencia, escribiremos y_i^j siendo $i \leq j \leq N-1$. Esto es así porque trataremos las posiciones de las secuencias como si fuesen *arrays* o listas en algunos lenguajes de programación, es decir, siendo la posición inicial el 0. Definimos la complejidad no lineal de una cadena como $c(y^N)$. He de decir también que las siglas FSR se utilizaran para referirse a los NLFSR (*Non Lineal Feedback Shift Register*).

3.3.1. Base teórica del algoritmo

Para explicar el algoritmo hay que mencionar los teoremas y proposiciones de Limniotis et al. [2007] que sirven como base.

Proposition 1. *Siendo L la longitud de la subcadena más larga de y^n que aparece al menos dos veces con distintos sucesores. Entonces, $c(y^n) = L+1$, esto se cumple siempre que el termino constante del polinomio pueda ser uno o cero.*

Explicación: teniendo estas dos ocurrencias de la misma subcadena distintos sucesores la complejidad no lineal tiene que aumentar en uno para poder así generar estas cadenas a través del nuevo polinomio FSR.

Proposition 2. *Siendo $c(y^{n-1}) = m$ y asumiendo que el FSR mínimo para y^{n-1} no genera y^n. Entonces $c(y^n) = m$ si y sólo si y_{n-m-1}^{n-2} no ha aparecido en y^{n-1}.*

Explicación: Si y_{n-m-1}^{n-2} hubiera aparecido previamente en y^{n-1} y $c(y^n) = m = c(y^{n-1})$, el polinomio FSR de y^{n-1} estaría obligado a generar y^n sin aumentar m, pues ambas apariciones de y_{n-m-1}^{n-2} tendrían los mismos sucesores.

Corollary 1. *Siendo $c(y^{n-1}) = m$ y asumiendo que el FSR mínimo para y^{n-1} no genera y^n. Entonces $c(y^n) > m$ si y sólo si existe un entero $0 < i < n-m-1$ que cumpla $y_i^{i+m-1} = y_{n-m-1}^{n-2}$ y $y_{i+m} \neq y_{n-1}$.*

Theorem 1. *Siendo $c(y^{n-1}) = m < c(y^n)$. Dado un $i \leq n-m-1$ que cumple $y_j^{j+m-1} = y_i^{i+m-1}$ para $0 \leq j < i$. Entonces*

$$c(y^n) = c(y^{n-1}) + (n-m-i) = m + (n-m-i) = n-i \tag{1}$$

Del teorema 1 se puede derivar que si el salto entre complejidades lineales es $k = c(y^n) - c(y^{n-1})$. Entonces

$$k = (n - i) - m = n - m - i \tag{2}$$

En términos de la periodicidad de la secuencia binaria.

$$k = n - m - (t_0^{(n-1)} + T^{(n-1)}) \tag{3}$$

Siendo $t_0^{(n-1)}$, el preperiodo de la secuencia y^{n-1} y $T^{(n-1)}$ el periodo de y^{n-1}. Además $t_0^{(i-1)} = j$ y $T^{(i-1)} = i - j$ respecto a i y j descritos en el teorema 1.

$$y_x = y_{x+T} \text{ para todo } x \geq t_0, \text{siendo } t_0 \geq 0, T > 0 \tag{4}$$

Esto viene a decir que, T es la longitud del periodo y t_0 el índice en el cual la periodicidad comienza, y a estas dos variables se les llaman *preperiodo* y *periodo*.

Theorem 2. *Siendo $c(y^{n-1}) = m$ y asumiendo que el FSR mínimo para y^{n-1} no genera y^n.*

$$c(y^n) = max\{c(y^{n-1}), n - k(y^{n-1})\} \tag{5}$$

donde $k(y^{n-1})$ es el eigenvalue de y^{n-1}. Además el eigenvalue de cualquier secuencia y^n es igual a la suma del preperiodo y del periodo, $t_0^{(N)} + T^{(N)}$, por lo que $k(y^{n-1})$ tambien es igual al i del teorema 1.

Theorem 3. *Siendo s_n la longitud de la subcadena más larga $y_{n-s_n}^{n-1}$ de y^n, teniendo $y_{n-s_n}^{n-1}$ más de una aparición en y^N.*

$$k(y^n) = n - s_n \tag{6}$$

El teorema 2 es cierto porque si $k(y^{n-1}) < n-m$. La subcadena y_{n-m-1}^{n-2} aparece dos veces con distintos sucesores en y^{n-1}. Luego, debido al corolario 1, se cumple $c(y^n) > c(y^{n-1})$. Concluyendo que $c(y^n)$ debe ser igual a $n-k(y^{n-1})$. Por otro lado, si $k(y^{n-1}) \geq n-m$ entonces la subcadena y_{n-m-1}^{n-2} es única en y^{n-1}, esto provoca que independientemente del valor de y_{n-1}, la complejidad lineal no aumente.

3.3.2. Complejidad no lineal y su relación con la complejidad de *Lempel-Ziv*

La complejidad de *Lempel-Ziv* es igual al número de palabras producidas por un procedimiento de *parsing* concreto. Este procedimiento resulta en el origen del algoritmo de compresión LZ77, Ziv y Lempel [1977]. Y un año más tarde los mismos investigadores crearían LZ78, Ziv y Lempel [1978]. Ambos algoritmos son algoritmos de compresión sin pérdidas, y ambos funcionan reemplazando una cadena que aparece repetida en el archivo por una referencia a esa misma cadena en una aparición anterior. Todo esto está ligado a los conceptos como *Shannon Information* o la entropía de la *teoría de la información*.

Antes de establecer la conexión entre la complejidad no lineal y la complejidad de *Lempel-Ziv* debemos nombrar algunas definiciones básicas.

Se dice que una secuencia y^n es reproducible por su prefijo y^i si y_{i+1}^n es una subcadena de y^i. Se describe *reproducibilidad* como $y^i \to y^n$. Además se dice que una secuencia y^n es producible por su prefijo y^i si y^{n-1} es reproducible por y^i. La *producibilidad* se describe como $y^i \Rightarrow y^n$.

Un historial $S(y^n)$ de una secuencia y^n es cualquier división de subcadenas

$$S(y^n) = y_0^{h_0} y_{h_0+1}^{h_1} y_{h_1+1}^{h_2} \cdots y_{h_{s-1}+1}^{h_s} \tag{7}$$

Donde $h_0 = 0$ y $h_s = n - 1$, $h_{i-1} < h_i$. Se cumple tambien que $y_0^{h_{i-1}} \Rightarrow y_0^{h_i}$, para $1 \leq i \leq s$. A $h_0, \ldots, h_s$ se les denomina como los puntos del historial y una subsecuencia $y_{h_i}^{h_{i-1}+1}$ se dice que es una

palabra del historial ($h_{-1} = 1$ por convención). Una de estas palabras es exhaustiva si $y_0^{h_{i-1}} \nrightarrow y_0^{h_i}$. Si todas las palabras de un historial son exhaustivas (se permite también que la última no lo sea), se le dice a ese historial que es un historial exhaustivo $Se(y^n)$. Cualquier secuencia y^n solo tiene un historial exhaustivo, es decir, es único. De entre todos los historiales posibles generados, el historial exhaustivo es el que menos palabras tiene. El número de palabras del historial exhaustivo es igual a la complejidad de *Lempel-Ziv*.

Pongamos como ejemplo la secuencia $y^{11} = 10001011101$. Esta secuencia tiene como historial exhaustivo:

$$Se(y^{11}) = 1 \cdot 0 \cdot 00 \cdot 10 \cdot 11 \cdot 101 \cdot$$

La complejidad de *Lempel-Ziv* de y^{11} es $LZ(y^{11}) = 6$ y los puntos de $Se(y^1 1)$ son 0, 1, 3, 5, 7, 10. Para poder entender la conexión con la complejidad no lineal $c(y^n)$, primero es necesario explicar qué es el *eigenvalue* de una secuencia y^n.

Al leer las definiciones de *eigenvalue* y *eigenwords* en Limniotis et al. [2007], no termine de entender a que se referían, así que fui a la fuente original, el artículo Lempel y Ziv [1976]. Esto me ayudo a aclarar los conceptos de vocabulario, prefijos *propios*, *eigenvalues* y *eigenwords* de una cadena y^n, junto con otros conceptos como historial, historial exhaustivo, reproducibilidad y producibilidad.

Vamos a definir brevemente cada una de estas cosas, para entender bien, como se calcula este *eigenvalue*. El vocabulario de una secuencia y^N son los subconjuntos de esa misma secuencia formado por todas las *subcadenas* desde el término i al j y_i^j siendo $0 < i < j$ y $j < N$. Por ejemplo, de la secuencia $y^5 = 01001$, su vocabulario sería.

$$v(y^5) = \{0, 1, 01, 10, 00, 01, 010, 100, 001, 0100, 1001, 01001\}$$

El siguiente paso es encontrar las *eigenwords* (*palabras propias*). Una palabra del vocabulario, es una palabra *propia* si no pertenece a ninguno de los vocabularios de los prefijos propios (*proper* prefixes) de la cadena original (y^N). Un prefijo *propio* es un segmento desde i hasta j, donde $0 < i < j < N-2$, es decir que la longitud del prefijo propio y_i^j debe ser menor que la longitud de la cadena original y^N, $l(y_i^j) < l(y^N)$. El conjunto de palabras *propias* se le denomina vocabulario *propio* $(e)(eigenvocabulary)$. El vocabulario propio del ejemplo anterior sería.

$$e(y^5) = \{001, 1001, 01001\}$$

Y ahora ya tenemos la forma de calcular el *eigenvalue* $k(y^N)$, pues el cardinal del vocabulario *propio* es igual al *eigenvalue*, $|e(y^N)| = k(y^N)$, para el ejemplo $k(y^5) = 3$. Toda este proceso tan largo, no es necesario para el calculo del *eigenvalue*, hay una manera mucho más sencilla y óptima, la desarrollaremos en la sección 3.3.5. Podéis consultar una implementación *naive* de este proceso en el programa naiveEigenvalue.py.

Se dice que el *eigenvalue profile* de una secuencia y^n es la secuencia de enteros resultante de $k(y^i)$, para $1 \leq i \leq N$. Para el ejemplo anterior el *eigenvalue profile* sería $1, 2, 2, 3, 3$.

Theorem 4. *Una palabra* $y_{h_i}^{h_{i-1}+1}$ *es exhaustiva si y sólo si* h_i *es el entero más pequeño que cumple* $k(y^{h_i+1}) > h_{i-1} + 1$

Theorem 5. *Si el eigenvalue profile de dos secuencias es igual, entonces necesariamente tienen que tener la misma complejidad no lineal.*

Dado el teorema 4 y el teorema 5 para cualquier secuencia y con un historial exhaustivo dado, su *eigenvalue profile* no puede ser arbitrario pues este es determinado por el historial exhaustivo. Es por esto que, podemos establecer una conexión entre la complejidad de *Lempel-Ziv* y la complejidad no lineal, debido a las propiedades mostradas del *eigenvalue profile*.

3.3.3. Algoritmo NLFSR

Ahora desglosaremos paso por paso el algoritmo, implementación en Python del mismo, nlfsr.py.

Algoritmo 3: Algoritmo FSR mínimo

```
Data: s = cadena binaria
Result: El non linear FSR mínimo
k ← 0;
m ← 0;
h ← y_0;
for i ← 1, ..., N − 1 do
    d ← y_i − h(y_{i−1}, ..., y_{i−m});
    if d ≠ 0 then
        if m = 0 then
            k ← i;
            m ← i;
        else if k ≤ 0 then
            t ← Eigenvalue(y^i);
            if t < i + 1 − m then
                k ← i + 1 − t − m;        /* n − m − k(y^n − 1) */
                m ← i + 1 − t;            /* n − k(y^n − 1) */
            end
        end
        f ← (x_1 + y'_{i+1})...(x_m + y'_{i−m});
        h ← h + f;
    end
    k ← k − 1;
end
```

Definamos las variables utilizadas por el algoritmo, la variable m guardará la complejidad no lineal de la secuencia, la variable k contendrá el valor del salto, y h es la función de feedback que acabara generando la secuencia original a partir de un segmento inicial de esa propia secuencia, el núcleo del algoritmo es entender como y por qué se actualiza esta función.

La variable d se recalcula en cada iteración y nos referiremos a esta como *discrepancia*. El cálculo de la *discrepancia* en la iteración i consiste en computar si el resultado de la función de feedback (h) de esa iteración difiere de la posición i en la secuencia. En caso de que difiera, se sumará el minterm que hace que esa posición se calcule como corresponde a la función de feedback, actualizando esta última. No porque haya una *discrepancia* tiene que aumentar la complejidad no lineal de la secuencia.

Existen varios casos si existe una *discrepancia*, el primero de ellos es que aun no se haya inicializado la complejidad no lineal de y^n, es decir, si m es igual a 0. En ese caso se les asigna tanto al salto, como a m el valor de n, es decir la posición de la primera *discrepancia*. El ejemplo de la sección 3.3.4 ayudará a la comprensión de todos estos pasos, se entenderá mucho mejor. El segundo caso, en el que m no es 0 y $k \leq 0$, se calculará el *eigenvalue* de la secuencia desde la posicion 0 hasta la posicion n.

Ahora expliquemos la comparación $t < n + 1 - m$. En caso de que la condición, se evalúe como cierta, esto quiere decir que siendo $m = c(y^{n-1})$, $k(y^n) < n - m$. Luego la subcadena de tamaño m, y_{n-m-1}^{n-2} se repite dos veces con diferentes sucesores en y^{n-1}, esto provoca que se tenga que sumar un *minterm* con una variable más al polinomio, o dicho de otra manera, corregir la función de *feedback*, línea del pseudocódigo *20*. Sin embargo, si la condición se evalúa como falsa, $k(y^{n-1}) >= n - m$, la subcadena y_{n-m-1}^{n-2} ahora es única por lo que a pesar de que se tenga que añadir un nuevo *minterm* a

la función de *feedback* el numero de parámetros del polinomio no aumentará, por lo que $c(y^n)$ tampoco.

Hay un punto que hemos pasado por alto, se trata de la función de la variable k (*salto*) en el pseudocódigo. La función principal de esta variable en el algoritmo es ahorrarnos el cálculo del *eigenvalue* en momentos en los que no es necesario calcularlo ya que sabemos que no han pasado suficientes iteraciones como para que pueda haber otro **salto** en la complejidad no lineal.

Esta variable solamente condiciona el flujo del programa en el *if* $k <= 0$, y como se puede observar se resta 1 a esta variable por iteración, cuando es menor o igual a 0, se le asigna la diferencia entre $c(y^n)$ y $c(y^{n-1})$. Esto en realidad, se hace para que tengan que pasar $c(y^n) - c(y^{n-1})$ iteraciones, hasta que pueda darse un nuevo salto en la complejidad ya que a partir de esas iteraciones es cuando **podrían** aparecer 2 subcadenas de longitud m con diferentes sucesores, es en ese momento donde el cálculo del *eigenvalue* cobra relevancia.

3.3.4. Ejemplo algoritmo FSR mínimo

Con todo este conocimiento previo, vamos a analizar un ejemplo en el que iremos explicando paso a paso, que hace el algoritmo. Tomando la secuencia binaria $y^{10} = 1101010011$, antes de entrar al bucle k y m serán 0. Y $h(x) = y_0 = 1$, téngase en cuenta que al contador/iterador lo llamaremos i por comodidad.

En la primera iteración del bucle:
$i = 1 \quad y_1 = 1$
Calculo de la discrepancia de $y^1 = 11$:
$h(1) = 1$, en este momento, h(x) es una función constante
¿es $h(1)$ igual a y_1?, sí, por lo que no hay discrepancia
$k = k - 1$, y seguimos adelante

En la segunda iteración del bucle:
$i = 2 \quad y_2 = 0$
Calculo de la discrepancia de $y^2 = 110$:
$h(1) = 1$, h(x) sigue siendo una función constante
¿es $h(1)$ igual a y_2?, No, así que tenemos nuestra primera **discrepancia**
Al ser la primera discrepancia, $m = 0$, así que
$m = i \quad k = i$
Importante, toca corregir nuestra función *fsr*, añadiendo el *minterm*
$\underline{x_{\{11\}}} = x_0 * x_1$
$\overline{h(x)} = h(x) + \underline{x_{\{11\}}} = 1 + x_0 * x_1$
Y como en todas las iteraciones,
$k = k - 1$

Antes pasar a la siguiente iteración, quería hacer un apunte, parece una tontería pero es que gracias al tener h la forma de una suma exclusiva de productos (3.5). Es muy simple e intuitivo que al añadir el *minterm* de esa posición, la función se corrija. Ya que si antes te daba 1, con el *minterm* (pues el minterm equivale a 1, en esa posicion de la cadena) cambiará a 0, y viceversa.

En la tercera iteración del bucle:
$i = 3 \quad y_3 = 1$
Calculo de la discrepancia de $y^3 = 1101$:
La subcadena que servirá como input hay que darle la vuelta para que coincida con el orden de las variables.
$y_{3-m}^{3-1} = y_1^2 = 10$

`reverse(`y_1^2`) = 01`
$h(x_0 = 0, x_1 = 1) = 1 + 0 * 1 = 1$
¿es $h(01)$ igual a y_3?, Sí, no se realizan cambios en $h(x)$
$k = k - 1$

En la cuarta iteración ocurre una *discrepancia* que no actualiza la complejidad no lineal $m = c(y^n)$, tan solo añade a la función h el *minterm*, $x_0 * x_1'$. En la quinta y sexta no hay discrepancias. Sin embargo, en la séptima iteración ocurre lo siguiente

$i = 7 \quad y_7 = 0$
Calculo de la discrepancia de $y^7 = 11010100$:
$y_{7-m}^{7-1} = y_5^6 = 10$
`reverse(`y_5^6`) = 01`
$h(x_0 = 0, x_1 = 1) = 1 + 0 * 1 + 0 * (1 + 1) = 1$
¿es $h(01)$ igual a y_7?, No

Ya que hay más de una subcadena igual y_{i-m-1}^{i-2} con distintos sucesores, hay que aumentar la complejidad no lineal m, en concreto y_5^6 se repite 2 veces previamente pero con el mismo sucesor. En esta iteración es la tercera vez que se repite la subcadena pero se da una discrepancia.

$$10 \quad \rightarrow \quad 10 \quad \rightarrow \quad 10 \quad \rightarrow \quad 0$$

Al darse este caso, el *eigenvalue* será menor que $n + 1 - m$. Por lo que se actualizará el k y m.
$k = n + 1 - t - m$
$m = n + 1 - t$
El siguiente paso es sumar el *minterm* con m ya actualizada, por lo que
$\underline{x_{\{01010\}}} = x_0' * x_1 * x_2' * x_3 * x_4'$
$h(x) = h(x) + \underline{x_{\{01010\}}} = 1 + x_0 * x_1 + x_0 * x_1' + x_0' * x_1 * x_2' * x_3 * x_4'$
Y al final de la iteracion,
$k = k - 1$

En la octava iteración no se dan discrepancias. Y por último, en la novena se da una discrepancia que no aumenta m, sumando el minterm $x_0 * x_1' * x_2' * x_3 * x_4'$. Aquí termina el ejemplo, habiendo recorrido toda la cadena, y quedando una función final de feedback tal que así

$$h(x) = 1 + x_0 * x_1 + x_0 * x_1' + x_0' * x_1 * x_2' * x_3 * x_4' + x_0 * x_1' * x_2' * x_3 * x_4'$$

Con esta función podríamos a partir de los primeros 5 digitos de la secuencia recuperar la secuencia entera, aplicando la función de feedback, añadiendo el resultado a la derecha y recalculando. Este proceso debe repetirse, siendo w, el numero de parámetros de la función de feedback y r, la longitud original de la cadena. Se aplicará el proceso $r - w$ veces, para conseguir la secuencia original. Podéis consultar también el decodificador, que he escrito en Python en verificamfsr.py.

3.3.5. Como encontrar el *eigenvalue* de manera óptima

Este valor es esencial en nuestro algoritmo previo, y ya hemos descrito anteriormente como calcularlo de la manera más sencilla o "*naive*". Sin embargo, en el articulo original Limniotis et al. [2007] en la sección que describe el procedimiento recursivo, indican que la manera más eficiente de calcular el *eigenvalue*, es mediante el algoritmo de *Knuth Morris Pratt*. Este algoritmo nos permite buscar con una complejidad temporal $O(m + n)$, un patrón en una cadena.

La idea general del algoritmo es que siendo n la longitud del patrón y p la secuencia que describe el patrón, en primer lugar se almacenan en una lista la longitud de las subcadenas que son a la vez

prefjio y sufijo de p_0^i para $i <= n$. Esta lista será de longitud n, para la subcadena p_0^0 no existe una cadena que cumpla las condiciones así que convenimos que sea igual a 0. En la segunda etapa del algoritmo utilizaremos está lista para no tener que reiniciar la búsqueda cada vez que el patrón no coincide completamente con el texto.

El algoritmo consta de dos partes, la primera, la llamaremos *preprocesado*, la cual tiene una complejidad temporal $O(m)$, siendo m la longitud del patrón. En esta sección, se calcula una tabla a la que normalmente se le llama *longest proper boundary table*, nos referiremos a esta con la misma notación que a las cadenas binarias de la sección 3.3, lpb^m. Esta tabla, en términos prácticos, será una lista en la que cada posición indica el *longest proper boundary* de la secuencia hasta ese carácter o dígito dependiendo del tipo de cadena a la que le apliquemos el algoritmo. Este valor, representa el tamaño de la subcadena más larga que es a la vez prefijo y sufijo de la cadena hasta el índice de esa posición del patrón.

En la segunda parte, recorremos el texto, ese bucle tendrá complejidad $O(n)$, siendo n la longitud del texto. Y el procedimiento será el siguiente, se recorre el texto y el patrón a la vez con dos índices distintos, i para el texto y j para el patrón, mientras los caracteres del texto y del patrón coincidan, ambos avanzan a la misma velocidad. Pero, cuando no coinciden y el patrón ha coincidido parcialmente con la parte del texto que se itera en ese momento, entonces es cuando es útil la lista lpb^m. Ya que, se le asignará al índice el valor de lpb_{j-1}, y no aumentará i en esa iteración. Retrocediendo el índice del patrón hasta lpb_{j-1}, $j = lpb_{j-1}$, se consigue es ahorrar en iteraciones respecto a una implementación *naive* pues a pesar de que no haya coincido en ese carácter puede que coincida más atrás del patrón. En el caso, de que el patrón coincida completamente en el texto, el índice j se le asigna lpb_{m-1}, ya que solo necesitas que coincidan las siguiente posiciones $m - lpb_{m-1}$ posiciones del texto con la subcadena del patrón $p_{lpb_{m-1}}^{m-1}$.

El algoritmo 4 da el pseudocodigo de una implementación general, que devuelve si el patrón se encuentra o no en el texto, no es difícil imaginar que también puedes devolver el numero de coincidencias del patrón en el texto, que es justo lo que nos interesa para el cálculo del *eigenvalue*.

Algoritmo 4: Algoritmo KMP (Knuth Morris Pratt)

```
Data: t = texto, p = patrón
Result: True si encuentra el patrón False si no lo encuentra
n ← len(t);
m ← len(p);
/* Preprocesado, calculo de lpb                                    */
lpb ← (0)_m;
for i = 0; i < m; i++ do
    j ← lpb_{i-1};
    while j > 0 & p_j ≠ p_i do
        j ← lpb_{j-1};
        if p_j == p_i then
            lpb_i ← j + 1;
        else
            lpb_i ← j;
        end
    end
end
/* Busqueda de p en t                                              */
isFound ← false;
j ← 0;
for i = 0; i < n; i++ do
    while j > 0 & t_i ≠ p_j do
        j ← lpb_{j-1};
    end
    if t_i == p_j then
        j ← j + 1;
    end
    if j == m then
        isFound ← true;
        break;
        /* Si no se para despues de la primera coincidencia completa */
        /* j ← lpb_{j-1}                                           */
    end
end
return isFound;
```

El cuadro 2 se ha tomado de Cormen et al. [2022] donde se explica en detalle y claridad todos estos algoritmos de búsqueda en cadenas. Lo destacable de esta tabla es la notoria diferencia entre *Knuth Morris Pratt* y el *Naive*, este *Naive* recorre la cadena buscando el patrón y cada vez que un carácter del patrón no le coincide, reinicia la búsqueda desde el siguiente carácter es por esto por lo que nos da una complejidad de $O((n-m+1)m)$.

Algortimo	Tiempo de Preprocesado	Tiempo de ejecución
Naive	0	$O((n-m+1)m)$
Rabin-Karp	$O(m)$	$O((n-m+1)m)$
Automáta finito	$O(m\|\Sigma\|)$	$O(n)$
Knuth Morris Pratt	$O(m)$	$O(n)$

Cuadro 2: Algoritmos de búsqueda de cadenas de texto

Nosotros aplicaremos este algoritmo para encontrar el *eigenvalue* de la siguiente manera en la función *Eigenvalue* del pseudocódigo. Guardaremos en una variable p (patrón), la cadena `reverse`(y_0^{i-1}), y en otra variable t (texto), la cadena p_1^{i-1}. Y teniendo una función *knuthmp* implementada con *Knuth Morris Pratt*, que podéis consultar en kmp.py, que lo que hace es devolver el numero de posiciones que se encuentran del patrón en el texto. Realmente, lo que está calculando *Knuth Morris Pratt* en este caso es la subcadena s_n del teorema 3. Con lo que el cálculo del *eigenvalue* resulta en

$$Eigenvalue(y^i) = i - knuthmp(t, p)$$

Esta diferencia puede dar como resultado un numero en el intervalo $[1, i]$ en función del numero de coincidencias del patrón en el texto.

3.4. Formato de archivo nlfsr

El formato de archivo nlfsr (Non Linear Feedback Shift Register) es un formato el cual codifica la secuencia inicial y el polinomio FSR. Teniendo como objetivo hacer que este tipo de archivo ocupe el menor espacio posible, para ello he propuesto que la estructura del formato fuera la representada en la figura 9.

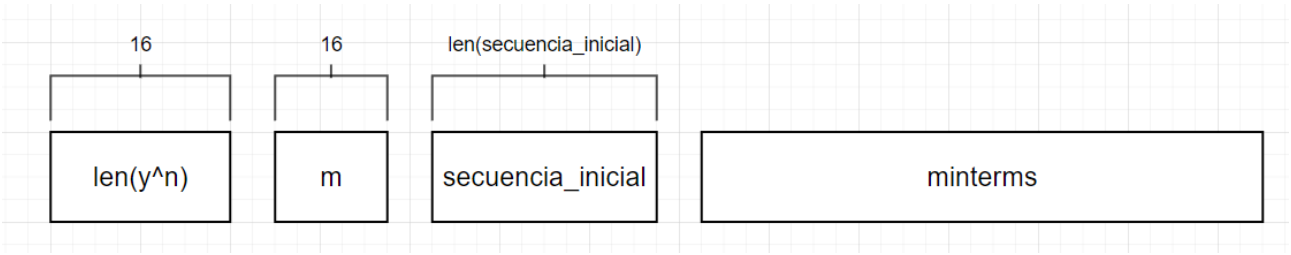

Figura 9: Formato .nlfsr

El primer segmento del archivo es la longitud de la cadena original (y^n), contenido en 16 bits. Estos 16 bits limitan el tamaño de la secuencia a comprimir en este formato, a una secuencia de longitud máxima 65536. Este límite es arbitrario, pero una vez se haya leído toda la explicación del formato se puede entender que no es difícil adaptarlo para cadenas más largas. El segundo segmento es la complejidad no lineal m de y^n contenida también en 16 bits (o 2 Bytes). El tercer segmento es la secuencia inicial que mide m bits. Y el cuarto segmento es una lista de *minterms*, se aprecia esta estructura en la figura 10.

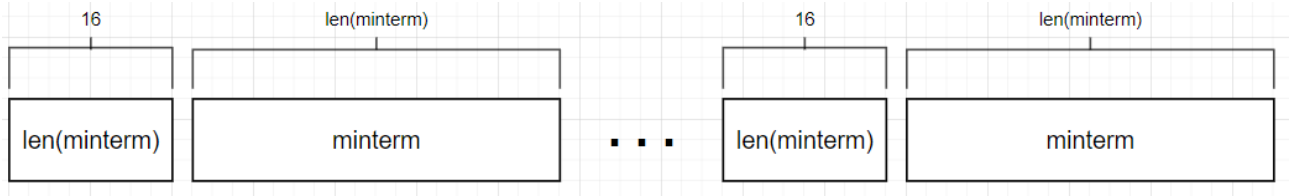

Figura 10: Lista de minterms en formato .nlfsr

Cada *minterm* consta de un primer segmento en el que se indica el numero de variables de ese producto de literales lógicos, o dicho de otra manera la longitud. Cada bit del *minterm* representara el estado de ese literal en el producto, si su valor es 1 eso significa que el literal se encuentra negado, por otro lado, si es 0, significará que es positivo. De esta manera, conseguimos almacenar toda la información consumiendo el mínimo numero de bytes posibles.

La longitud del propio *minterm* actuando como delimitador entre *minterms*, nos permite saber hasta

donde tenemos que leer cuando haya que decodificar el archivo .nlfsr. Es importante aclarar, que no se pueden utilizar delimitadores fijos en este caso, como sí se en algunos protocolos de red, ya que toda secuencia podría estar dentro del minterm, y eso haría que se mezclasen los datos con las delimitaciones, lo cual provocaría errores en el momento de la lectura.

La diferencia entre guardar un nlfsr de una secuencia con la misma estructura del formato .nlfsr pero usando caracteres ASCII (1 Byte por cada uno), y utilizar el formato .nlfsr, evidentemente, es más que notoria. En la figura 11 se encuentra la comparación sin que se haya utilizado ningún algoritmo de compresión después.

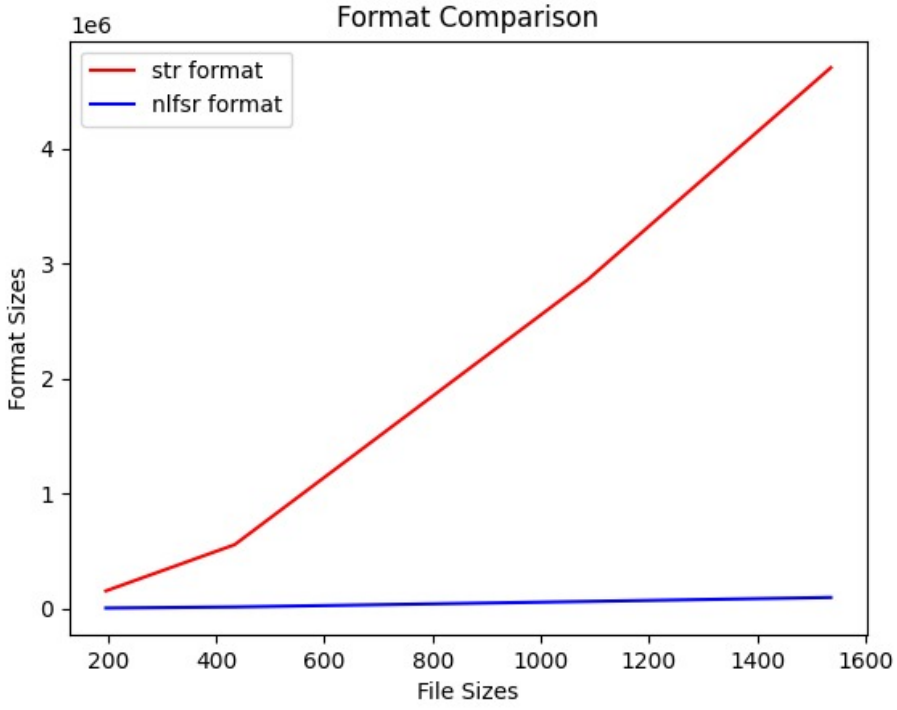

Figura 11: Comparación de tamaño de ambos formatos

3.5. Minimización de ESOP

La minimización de expresiones de operaciones AND-OR cuenta con muchos algoritmos que hacen un buen trabajo, sin embargo la minimización de suma exclusiva de productos (ESOP por sus siglas en inglés[1]) resulta ser mucho más compleja o al menos, no se han desarrollado algoritmos del mismo calibre. Pero primero, definamos que es todo esto.

A la forma de la función FSR de la sección 3.3, se la conoce como ESOP. Una formula lógica con esta forma, tan solo puede tener dos operaciones: XOR (suma) y AND (multiplicación). Minimizar una fórmula ESOP, significa encontrar una formula de menor tamaño que tenga la misma tabla de verdad, o el mismo mapa de Karnaugh; en definitiva, que sean equivalentes. En la figura 12 se describe un ejemplo de un ESOP y su minimización.

X_2 \ X_1X_0	00	01	11	10
0	0	1	0	0
1	0	1	1	1

$$f(X_0, X_1, X_2) = X_0X_1 \oplus X_1X_2 \oplus X_2$$

Figura 12: Ejemplo ESOP 01

En la figura 13 se encuentra una equivalencia entre $X_1X_2 \oplus X_2$ y un cubo formado por estos literales.

X_2 \ X_1	0	1
0	0	1
1	0	0

$$f'(X_1, X_2) = X_1X_2 \oplus X_2$$
$$z(X_1, X_2) = X_1'X_2$$
$$f' = z$$

Figura 13: Ejemplo ESOP 2

Y sabiendo esto, sustituimos en la formula original y nos queda un ESOP minimizado

$$f_{min} = X_0X_1 \oplus X_1'X_2$$

3.5.1. Investigación previa

La *Suma Exclusiva de Productos* se ha estudiado desde 1927 por I.I. Zhegalkin, desde año se han escrito numerosos artículos realizando avances en el campo. En esta investigación se ha tomado como punto de partida la publicación Mishchenko y Perkowski [2001]. En este describen un algoritmo en el

[1] *Exclusive Sum Of Products*

que mediante transformaciones de cubos, expresiones *pseudokronecker*, reescritura de términos Brand y Sasao [1993], y estructuras de datos como BDD (Binary Decision Diagram) (en ESOPtoBDD.py está escrito un algoritmo para pasar un ESOP a BDD). Nuestra aproximación al problema en este proyecto será mucho más sencilla pero haber leído todos estos artículos sobre el tema, me ha hecho entender más en profundidad lo complejo que es.

En esta sección tomaremos prestada la teoría que elaboran Mishchenko y Perkowski [2001] sobre los cubos que forman los ESOP. Definiremos como "cubo" a cada producto del ESOP. Definiremos como distancia entre cubos, el numero de variables que aparecen de distinta "forma" respecto al otro cubo. Una variable puede aparecer en un cubo de tres maneras: en positivo, negada, o no aparecer. Por ejemplo, un cubo ab tiene distancia 1 respecto a un cubo ab'. Considero importannte remarcar estas dos proposiciones del artículo:

Proposición 1:
Si añades el mismo cubo dos veces a cualquier ESOP, la función no cambiará.
Proposición 2:
La suma de dos cubos con distancia 1 se puede representar en un solo cubo.

3.5.2. Enumeración explicita para encontrar el ESOP de N variables de menor longitud

¿Y si en lugar de encontrar la minimización de una fórmula en concreto, encontramos el mínimo de símbolos que debe de tener una fórmula para todos los resultados posibles dada una formula (o ESOP) de n variables de entrada? Pues bien, esto es justo lo que hace nuestro pequeño algoritmo. En lo que sigue veremos paso a paso en que consiste.

El primer paso que debemos entender es como conseguimos representar todas las formulas posibles de n variables con m simbolos, pudiendo ser cada símbolo, o bien, un XOR, o bien, un AND. La estructura que utilizamos para lograrlo es muy sencilla, es una lista de enteros. Pero tiene un truco, y es que el 0 representa el XOR o la suma, el 1 representa la operación AND o la multiplicación, y el 2 representa la negación. Cualquier entero $i > 2$ representa una variable de entrada X_{i-3}. Con este sistema de representación en el que cualquier fórmula puede ser representada por una lista o cadena de enteros en la que cada posición tiene un entero i, $0 <= i < n+3$ de longitud, $m+(m+1)$, siendo n el numero de variables de la fórmula y m, el numero de símbolos(XOR, AND) que puede tener la fórmula.

Para entender mejor el sistema, pondré los siguientes ejemplos. Teniendo una formula $X_0 * X_2 * X_3 + X_0$ y otra formula $X_0' + X_1 * X_3 + X_4$, estas son sus representaciones con este sistema.

$$X_0 * X_2 * X_3 + X_0 = [0, 1, 3, 1, 5, 6, 3]$$
$$X_0' + X_1 * X_3 + X_4 = [1, 0, 2, 3, 4, 0, 6, 7]$$

El cálculo del numero de posibles resultados para una formula de n variables de entrada es

$$k = 2^{2^n}$$

Ahora debemos encontrar para cada posible resultado del conjunto de tamaño k, una formula que satisfaga ese resultado. Para ello lo que haremos, será generar todas las combinaciones con repetición del conjunto $\{0, 1, ..., n+3-1\}$ de longitud p, definimos al conjunto de todas las combinaciones como $g^{(n+3)^p}$, cada elemento de g será una posible combinación de tamaño p. Es importante decir que, p determinará el numero de posibles símbolos que pueden haber en la fórmula.

Es cierto que habrá algunas combinaciones que no tendrán sentido sintáctico en esta gramática que hemos creado. Por ejemplo $[2, 2, 2, 2, 2, 2, 2]$, no representa ninguna fórmula. Para validar las listas que tienen sentido de las que no, hemos escrito una función recursiva que crea una cadena de texto o *string* en base a la lista. Aquí tenéis el pseudocódigo, también podéis leer el código real escrito en Python en logicformulasolver.py.

Algoritmo 5: Algoritmo intListToStringFormula(f)

```
Data: f = lista de enteros
Result: Función lógica en formato string
if isNotEmpty(f) then
    val ← f.pop();                                   // pops last int from f
    if val == 0 then
        return stringFormula(f) + '+' + stringFormula(f);
    end
    if val == 1 then
        return stringFormula(f) + '*' + stringFormula(f);
    end
    if val == 2 then
        return 'not' + stringFormula(f);
    end
    return 'x' + (val - 3);
end
return 'S';
```

El siguiente paso, es escribir una función para calcular cuantos posibles resultados se generan para un ESOP con un numero de variables dado y una longitud de la cadena de lista de enteros dada. Esta válida cada lista de enteros generada. Para después, probar para cada una de estas fórmulas lógicas todas las posibles combinaciones de las variables de entrada de la función, es decir si por ejemplo, es una función de 3 variables de entrada probara las 2^3 combinaciones. Recopilará todos los resultados y hará una lista con ellos. La intentará sumar a un set, la cual se sumará en función del hash calculado a esa lista internamente por Python.

Una vez se han agotado todas las combinaciones de listas de enteros ("formulas lógicas"), se comparará la longitud del set con el número de posibles resultados k; si estos coinciden es que hemos encontrado el tamaño mínimo de la lista de enteros para generar todas las formulas necesarias que coinciden con todos los resultados posibles.

En las figuras 14, 15 y 16 en el eje X la primera fila es el tiempo que ha tardado en ejecutar el algoritmo, y la segunda fila es el tamaño de la lista de enteros. El eje Y representa el numero de posibles resultados retornados por el algoritmo.

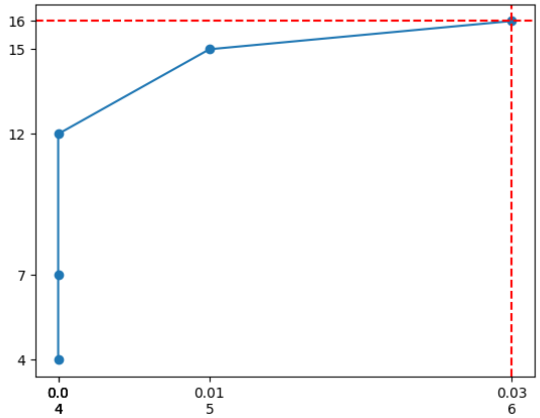

Figura 14: ESOP de 2 variables de entrada

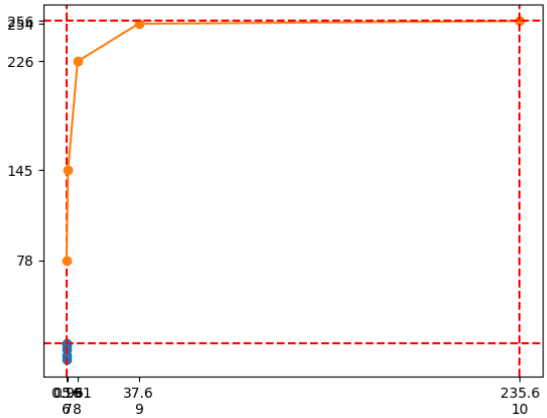

Figura 15: ESOP de 3 variables de entrada

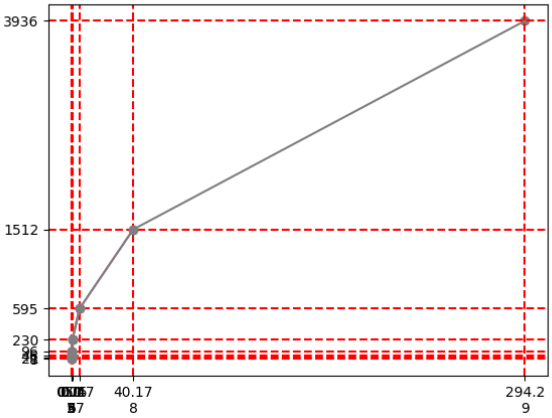

Figura 16: ESOP de 4 variables de entrada

Como vemos en las gráficas, el algoritmo converge en ambos casos es decir, encuentra todas las listas de enteros que satisfacen los k posibles resultados para ESOP de n variables. Sin embargo, a partir de ESOPs de 4 variables debido a la complejidad temporal del algoritmo, se tarda mucho en que el algoritmo converja y empeora conforme el número de variables aumenta, ya que la complejidad temporal de nuestro algoritmo es de $O((n+3)^p * 2^n)$. Con mi procesador, en 10 minutos consigue calcular la función hasta $p = 9$.

3.5.3. Implementación propia y paralelización en C

Para calcular las combinaciones en el código original, hemos usado la librería *itertools*, y en concreto la función *product*, por recomendación de mi tutor. Este modulo de Python genera sus funciones, utilizando Cython para después crear módulos utilizables desde Python, pero escritos en C. Sentí cierta curiosidad, por como sería hacer un programa en C que calcule todas las combinaciones tal y como lo hace *itertools*.

Así que escribí dos versiones una secuencial secuencial.c. y otra paralelizada con *OpenMP* paralelo.c. Si habéis leído el código sabréis que lo hago todo sobre un solo *array*, lo definí así por la localidad de los valores almacenados en memoria. En mi *hardware*, el código secuencial tuvo un rendimiento casi idéntico al de *itertools*, sin embargo, el código paralelizado sí me dio mejores resultados, incluso superando a la función del módulo de Python. Si bien, es cierto que Python está çapado"por el GIL (Global Interpreter Locker), ya que a pesar de contar con módulos que ofrecen funciones de "multithreading"no se consigue en ningún momento una paralelización real, considero que los datos siguen siendo interesantes, aquí tenéis una gráfica.

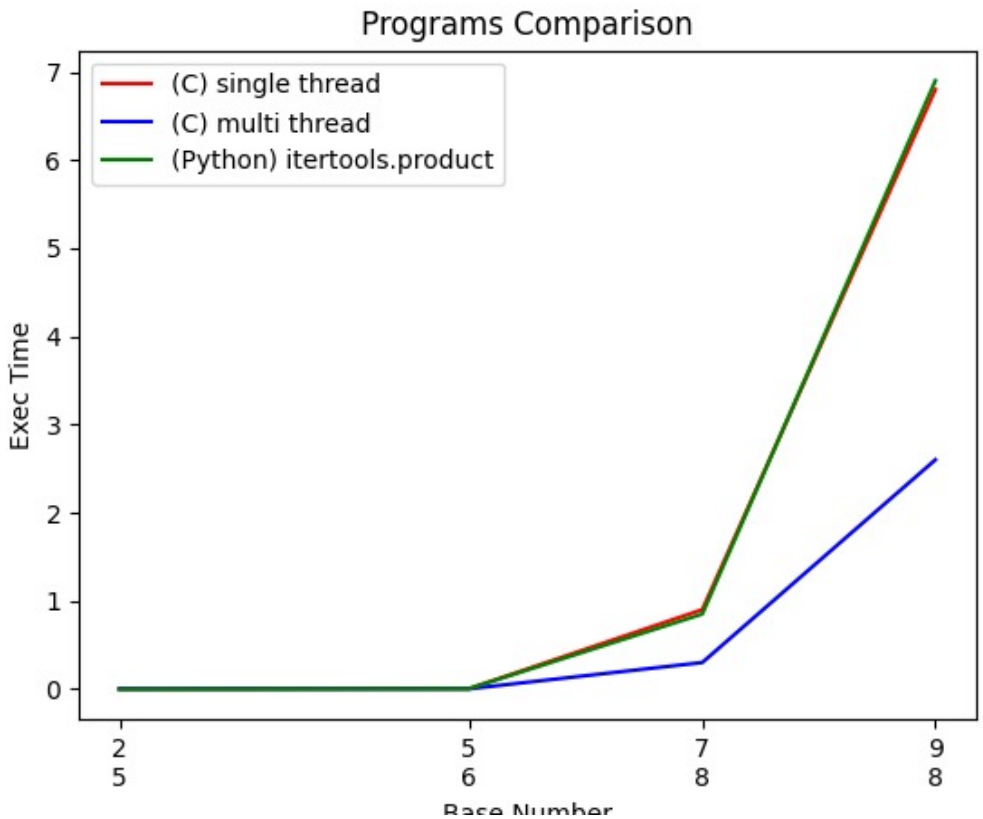

Figura 17: Comparación de rendimiento

El eje Y es el tiempo de ejecución en segundos, y el eje X tiene 2 listas de valores, la lista más cercana al eje es el cardinal del conjunto sobre el que se calculan todas las posibles combinaciones. y el tamaño de la lista es el número de debajo. Cabe destacar que para un conjunto de 10 elementos y un tamaño de lista 8, mi sistema operativo (Debian *bookworm*) mataba el proceso de Python. Sin embargo, para los mismos parámetros mis programas escritos en C, tanto el secuencial como el paralelo, registraron 16.8 y 5.8 segundos respectivamente. Evidentemente con parámetros más altos tenía problemas de memoria debido a mi *hardware* limitado.

4 Conclusiones y posibles avances futuros

4.1. Avances futuros

En este trabajo se tocan muchos temas distintos. Cada uno de ellos puede expandirse mucho más de lo que se ha hecho aquí. Se me ocurren muchas posibilidades para expandir GoneFSR y la investigación de la minimización de ESOP (*Exclusive Sum of Products*).

Para el compilador GoneFSR se podría trabajar en implementar operadores lógicos *bitwise*, implementar instrucciones de control de flujo como *break* o *continue*. O en funcionalidades más complejas, como implementar paradigmas como OOP (*Object Oriented Programming*) con sus respectivas clases, atributos y demás. Podría parecer complicado, pero creo que no lo es tanto ya que las clases no dejan de ser funciones, las cuales tienen otras funciones definidas dentro de su *scope*. En un futuro, también me gustaría programar algún sistema de importaciones para poder trabajar sobre varios ficheros, teniendo en cuenta las importaciones circulares y otras casuísticas. Otra futura implementación podría ser la mejora en la administración de memoria con sistemas como la *garbage collection* de Python que a partir de técnicas como el conteo de referencias, o el *cycle detection* para evitar ciclos de referencias, gestionan la liberación de la memoria de las variables a las que no se hace referencia. Y por qué no, puestos a imaginar un gestor de módulos como *pip* o *npm*.

Por otro lado, se podría desarrollar un backend personalizado con optimizaciones de código usando vectorización con instrucciones SIMD (*Single Instruction Multiple Data*), *unrolling* para descomponer bucles y aumentar la paralelización del código, reordenación de instrucciones para minimizar tiempos de espera. Estas son sólo unas pocas líneas de investigación que se pueden seguir en el desarrollo de un backend de compilación.

En cuanto a la segunda parte del proyecto hay muchas cosas que he estado estudiando antes de escribir esta memoria que no han podido ser reflejadas en estas líneas ya que a pesar de que puede que tengan que ver, quedaban fuera de los límites del proyecto. Como por ejemplo conceptos como la síntesis de programación, la cual trata de generar programas que cumplan ciertos requisitos, estudiando algoritmos como la enumeración explicita e implícita, si queréis saber más podéis consultar estos artículos *programming synthesis*. O novedosos avances del estado del arte de la inteligencia artificial, como los sistemas de agentes LLM (*Large Language Model*) (p.ej Islam et al. [2024]) para generar programas con tan sólo una vaga descripción de lo requerido.

Donde más potencial de investigación en lo referido a las funciones de registro de desplazamiento estudiadas creo que existe es en la minimización de ESOP, destiné algún tiempo a estudiar el artículo Mishchenko y Perkowski [2001] y a estudiar código que encontre de su proyecto *EXORCISM* para minimizar estas funciones lógicas. Me estaba llevando mucho tiempo y no conseguí llegar a nada en concreto, pero creo que con el suficiente tiempo se puede indagar en ese campo para buscar nuevas optimizaciones. Creo que otra aproximación adecuada para este problema de minimización podrían ser los SMT (*Satisfacibility Modulo Theories*) como *Z3*, y los SAT solvers, que hace relativamente poco consiguieron resolver el teorema de las tripletas booleanas pitagóricas en 4 años de CPU tal y como dice el matemático Terrence Tao en la conferencia.

4.2. Conclusión

Este proyecto ha llevado mucho trabajo, ha habido momentos muy frustrantes tanto en el desarrollo del compilador, como en la investigación de los algoritmos de registro de desplazamiento. Pero no me arrepiento en absoluto de haberme embarcado en esta empresa.

La implementación de las sentencias condicionales, trabajar con strings en llvm, añadir la capacidad de escribir funciones, parece poco pero son muchos problemas distintos a los que enfrentarse. Aunque es cierto, que una vez entiendes la estructura central del compilador cada extensión no es tan complicada pues al final son modificaciones de esa propia estructura.

Entender como funciona el algoritmo de la sección 3.3 me llevo mucho tiempo, entender por qué se suma los minterms como se suman, el por qué del cálculo del *eigenvalue*, conceptos como la complejidad de *Lempel-Ziv* u otros de los que nunca había oído hablar. Intentar hacer un formato que sea óptimo para guardar la información de ese algoritmo. Todo esto fue como hacer un *puzzle* en el que te dan las piezas pero no sabes cómo es la imagen final que deben de formar.

A modo de conclusión final, me gustaría reafirmar que este proceso aunque haya tenido sus malos ratos y sus momentos de satisfacción cuando todo cuadraba (o cuando parecía que todo cuadraba y al final no), creo que ha merecido la pena, y acabo el proyecto siendo un mejor ingeniero informático.

Bibliografía

A. V. Aho, M. S. Lam, R. Sethi, y J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, 2nd edition, 2006.

E. R. Berlekamp. Non-binary bch decoding. Technical report, North Carolina State University. Dept. of Statistics, 1966.

D. Brand y T. Sasao. Minimization of and-exor expressions using rewrite rules. *IEEE Transactions on Computers*, 42(5):568–576, 1993.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, y C. Stein. *Introduction to Algorithms*. MIT Press, 4th edition, 2022.

M. A. Islam, M. E. Ali y M. R. Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

A. Lempel y J. Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1):75–81, 1976.

K. Limniotis, N. Kolokotronis y N. Kalouptsidis. On the nonlinear complexity and lempel–ziv complexity of finite length sequences. *IEEE Transactions on Information Theory*, 53(11):4293–4302, 2007.

J. Massey. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, 15 (1):122–127, 1969.

A. Mishchenko y M. Perkowski. Fast heuristic minimization of exclusive-sums-of-products. 2001.

J. Ziv y A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.

J. Ziv y A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, Sep 1978.

A Apéndice 1: Regex Tokens

```
FLOAT = r'\d+\.\d+e[+-]?\d+|\d+\.\d*|\.\d+'
INTEGER = r'0x[a-f0-9]+|0o[0-7]+|0b[10]+|\d+'
CHAR = r"'(?:\\(?:\\|n|x[0-9a-f]{2}|')|[^'\\])'"
STRING = r"\".*\""
BOOL = r"\btrue\b|\bfalse\b"
ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
ONE_LINE_COMMENT = r'//.*'
MULTI_LINE_COMMENT = r'/\*[\s\S]*?*/'
```

Los comentarios no son tokens, simplemente hacen que el automata finito avance sin devolver ningun tipo de token.

Las *keywords* no se incluyen en este listado. Para encontrar todos los tipos de token de GoneFSR, y otros detalles, consultad el código fuente en tokenizer.py.

B Apéndice 2: Gramática completa

```
program : statements
        | empty

statements :  statements statement
           |  statement

statement :  const_declaration
          |  var_declaration
          |  assign_statement
          |  print_statement
          |  if_statement
          |  while_statement
          |  return_statement

const_declaration : CONST ID = expression ;

var_declaration : VAR ID datatype ;
                | VAR ID datatype = expression ;

assign_statement : location = expression ;

print_statement : PRINT expression ;

coder_statement: CODER expression , expression ;

decoder_statement: DECODER expression , expression ;


if_statement : IF expression { statements }
             | IF expression { statements } ELSE { statements }

while_statement: WHILE expression { statements }

function_definition : FUNC ID LPAREN arguments RPAREN datatype { statements }

parameters: parameter
          | parameters COMMA parameter
          | expression
          | empty

function_calling: function_location LPAREN parameters RPAREN SEMI

arg_declaration: ID datatype

arguments: argument
          | arguments COMMA argument
          | arg_declaration
          | empty
```

```
49
50 expression : + expression
51            | - expression
52            | ! expression
53            | expression && expression
54            | expression || expression
55            | expression == expression
56            | expression != expression
57            | expression < expression
58            | expression <= expression
59            | expression > expression
60            | expression >= expression
61            | expression + expression
62            | expression - expression
63            | expression * expression
64            | expression / expression
65            | ( expression )
66            | location
67            | literal
68            | function_call
69
70
71 literal : INTEGER
72         | FLOAT
73         | CHAR
74         | BOOL
75
76 function_location: ID
77
78 location : ID
79          ;
80
81 datatype : ID
82          ;
83
84 empty    :
85 }
```