Escuela
Politécnica
Superior

Diseño y creación de un compilador para un lenguaje funcional tipo Scheme

Grado en Ingeniería Informática

Trabajo Fin de Grado

Autor:
Jorge Sánchez Pastor
Tutor/es:
Antonio Miguel Corbi Bellot (tutor)

Universitat d'Alacant
Universidad de Alicante

Julio 2024

Diseño y creación de un compilador para un lenguaje funcional tipo Scheme

Autor
Jorge Sánchez Pastor

Tutor/es
Antonio Miguel Corbi Bellot (tutor)
Departamento de Lenguajes y Sistemas Informáticos

Grado en Ingeniería Informática

Escuela
Politécnica
Superior

Universitat d'Alacant
Universidad de Alicante

ALICANTE, Julio 2024

Preámbulo

El presente proyecto surge de la necesidad de profundizar en los conceptos fundamentales de los lenguajes de programación funcionales, tanto desde una perspectiva teórica como práctica. La motivación principal ha sido explorar y comprender mejor la interpretación y compilación de lenguajes. El estudio se centra en la implementación de un subconjunto de Scheme, destacando el uso de macros y la evaluación de expresiones. La finalidad del proyecto es establecer una base sólida sobre la cual se puedan construir características más avanzadas, optimizaciones y, eventualmente, un compilador completo. Además, quiero expresar mi gratitud a mi tutor, cuyo apoyo ha sido decisivo en la ejecución exitosa de este proyecto.

Índice general

Índice de figuras

Índice de tablas

Índice de Códigos

1 Introducción

Este proyecto se centra en la implementación de un núcleo del lenguaje Lisp y en las técnicas de compilación y evaluación para llevarlo a cabo. Explora las estructuras internas y las interacciones de los objetos básicos del lenguaje. Aunque Lisp es conocido por su simplicidad su implementación conlleva dificultad. Inicialmente, el objetivo era crear un compilador completo, pero debido a limitaciones de tiempo, el proyecto se enfocó en un evaluador funcional. Este enfoque permitió una comprensión detallada de los mecanismos fundamentales de Lisp, desde listas y átomos hasta closures y s-expresiones. A lo largo del documento se abordan aspectos esenciales como el análisis léxico y sintáctico, las técnicas de parsing y la importancia de las gramáticas formales. Además, se discuten las decisiones de diseño, como la no implementación de macros higiénicos de Scheme y el uso de estructuras de datos nativas para simplificar el desarrollo. Aunque la implementación completa del compilador quedó fuera del alcance, el trabajo realizado establece una base sólida para futuras extensiones y optimizaciones, destacando la flexibilidad y el poder del lenguaje Lisp en la metaprogramación y la manipulación de código.

2 funcionamiento del Lisp

Una parte importante de este proyecto fue entender en profundidad cuál era el funcionamiento interno de las estructuras de los lenguajes tipo Lisp. Lisp es un lenguaje relativamente simple con una cantidad limitada de objetos que interaccionan entre sí, conocer el comportamiento de los elementos básicos del lenguaje será de gran ayuda para el proyecto.

2.1 Listas

Las listas son el objeto principal de Lisp, estas funcionan como un conjunto de pares anidados, donde cada par está compuesto por dos elementos: el primer elemento es el valor actual (par o átomo), y el segundo elemento es un par o una lista vacía. Este encadenamiento permite construir estructuras de datos complejas y flexibles.

Una lista puede contener cualquier tipo de dato, incluidos otros pares, listas, y funciones. Las listas se representan típicamente utilizando paréntesis, y los elementos de la lista se separan por espacios.

Para crear una lista en Lisp, se puede usar la función `cons`, que toma dos argumentos y devuelve un par. Por ejemplo:

```
(cons 1 (cons 2 '()) ; devuelve (1 2)
```

Algo importante a la hora de entender el funcionamiento de Lisp es comprender que las listas son en realidad un solo par, con una manipulación más similar a una lista encadenada que a un vector. Para acceder a los elementos de una lista, se utilizan las funciones `car` y `cdr`. La función `car` devuelve el primer elemento de la lista o par , mientras que `cdr` devuelve el segundo elemento del par, que sería el resto de pares anidados, devolviendo así el resto de la lista.

```
(setq my-list '(1 2 3))
(car my-list) ; devuelve 1
(cdr my-list) ; devuelve (2 3)
```

Así si expresamos las listas como pares. podemos ver cómo `cdr` se limita a devolver el segundo elemento del par.

```
(setq my-list (1 . (2 . (3 . '()))))
(car my-list) ; devuelve 1
(cdr my-list) ; devuelve (2 . (3 . '()))
```

2.1.1 S-expresiones

Para Lisp una s-expresión (abreviatura de expresión simbólica) no es más que es una lista cuyo primer componente es un procedimiento (o*closure*) y el resto de los componentes son los argumentos que se pasan a ese procedimiento. Las s-expresiones son la forma en que se evalúan las funciones en Lisp.

2.2 Quotes, quasiquotes y unquotes

En Lisp, los mecanismos de quotes, quasiquotes y unquotes son herramientas poderosas para la construcción y manipulación de listas y expresiones simbólicas. Estos mecanismos permiten a los programadores controlar cómo se evalúan las expresiones, facilitando la creación de macros y la metaprogramación.

2.2.1 Quotes

El operador `quote` en Lisp se utiliza para evitar la evaluación de una expresión. Cuando se aplica `quote` a una expresión, Lisp trata la expresión como un dato literal en lugar de evaluarla como código. Esto es útil para crear listas y otros datos de forma literal sin que se interpreten como llamadas a funciones o variables.

El operador `quote` se denota con una comilla simple (`'`) seguida de la expresión que se desea citar. Por ejemplo:

```
'(1 2 3) ; devuelve (1 2 3)
```

En este ejemplo, la lista `(1 2 3)` no se evalúa; simplemente se trata como un literal. Sin el `quote`, Lisp intentaría evaluar `(1 2 3)` como una expresión, lo que resultaría en un error si `1` no es una función.

El uso de `quote` es fundamental en Lisp para trabajar con s-expresiones como datos. Permite a los programadores manipular y construir código Lisp sin que se evalúe inmediatamente.

2.2.2 Quasiquotes

El operador `quasiquote`, denotado por una comilla invertida (`` ` ``), es similar a `quote`, pero con una diferencia importante: permite la evaluación selectiva de partes de la expresión citada. Esto se logra utilizando el operador `unquote` dentro de la expresión quasiquoted.

El `quasiquote` se utiliza para crear plantillas donde algunas partes de la expresión se mantienen literales y otras se evalúan. Por ejemplo:

```
`(1 2 ,(+ 1 2)) ; devuelve (1 2 3)
```

En este ejemplo, la lista `(1 2 ,(+ 1 2))` utiliza `quasiquote` para citar la lista, pero la expresión `(+ 1 2)` precedida por `unquote` (`,`) se evalúa, resultando en `(1 2 3)`.

2.2.3 Unquotes

El operador `unquote`, denotado por una coma (`,`), se utiliza dentro de una expresión quasiquoted para indicar que la expresión debe ser evaluada. El `unquote` permite insertar valores calculados en estructuras de datos quasiquoted.

Por ejemplo:

```
(setq x 10)
`(1 2 ,x) ; devuelve (1 2 10)
```

En este caso, `x` se evalúa a su valor `10` dentro de la lista quasiquoted, resultando en `(1 2 10)`.

Además de `unquote`, Lisp proporciona `unquote-splicing`, denotado por `,@`, que permite insertar múltiples elementos en una lista quasiquoted. Por ejemplo:

```
(setq lst '(3 4))
`(1 2 ,@lst) ; devuelve (1 2 3 4)
```

Aquí, `,@lst` inserta los elementos de la lista `lst` en la posición correspondiente, resultando en `(1 2 3 4)`.

2.2.4 Aplicaciones de Quotes, Quasiquotes y Unquotes

Estos operadores son esenciales para la metaprogramación en Lisp. Facilitan la creación de macros, que son funciones que generan y manipulan código Lisp. Las macros permiten a los programadores extender el lenguaje, creando nuevas construcciones sintácticas y abstracciones.

Por ejemplo, se puede definir una macro que crea una función que siempre devuelve un valor constante:

```
(defmacro always (val)
  `(lambda () ,val))

(setq always-5 (always 5))
(funcall always-5) ; devuelve 5
```

En este ejemplo, la macro `always` utiliza `quasiquote` para construir una expresión lambda que retorna `val`. El uso de `unquote` asegura que el valor de `val` se inserte correctamente en la expresión lambda.

2.2.5 Importancia en la Programación Lisp

La capacidad de citar y manipular código como datos es una de las características más poderosas de Lisp. Permite una gran flexibilidad y expresividad en la programación, haciendo que Lisp sea ideal para aplicaciones que requieren una manipulación sofisticada del códi-

go, como en el desarrollo de lenguajes de dominio específico (DSLs), la implementación de sistemas expertos, y la construcción de herramientas de análisis y transformación de código.

En resumen, los operadores `quote`, `quasiquote` y `unquote` son herramientas fundamentales en Lisp que permiten a los programadores controlar la evaluación de expresiones y facilitar la metaprogramación. Estas características distinguen a Lisp como un lenguaje excepcionalmente poderoso y adaptable.

2.3 Entorno

el Entorno (*environment*) es una estructura de datos que asocia nombres de variables con sus valores. Un *environment* es esencialmente una tabla de símbolos donde cada símbolo está vinculado a un valor. Esta estructura permite que las variables sean accesibles en diferentes contextos y mantiene la información necesaria para la evaluación de expresiones Lisp.

2.4 Closures

En Lisp un*closure* es un objeto que contiene dos cosas, una función y un entorno donde el entrno es aquel entorno en el que fue creada. Esto significa que una función definida dentro de otro contexto puede acceder a las variables locales de dicho contexto incluso después de que el contexto haya finalizado su ejecución.

Un*closure* funciona capturando las variables locales de su entorno en el momento de su creación. Estas variables se mantienen accesibles mientras exista el*closure*. El*closure* retiene una referencia al entorno en el que fue creado, permitiendo que la función acceda y modifique las variables locales de ese entorno incluso cuando se ejecuta fuera de su contexto original.

Los*closures* son una parte importante de Lisp, ya que son el objeto que crearemos al definir funciones.Estas se definene a traves de `lambda` de la siguiente forma.

```
(lambda (a b) (+ a b))
```

Este devolverá un*closure* que esencialmente contendrá la lista de parámetros, el Entorno del momento en que se cree, o al menos un puntero a este, y el cuerpo que acabamos de definir. Si queremos guardar esta función debemos enlazar a un símbolo en nuestro *environment*.

Código 2.1: Definición de un macro

```
(define .map (lambda (f xs)
          (if (eq? xs null) null
              (cons (f (car xs)) (.map f (cdr xs))))))
(define suma1 (lambda (x) (+ 1 x)))
```

Cuando vayamos a aplicar este*closure* en forma de `(.map suma1 '(1 3 4))`, primero se añadirá la información del entorno al actual , luego se encajaran los argumentos con lo parámetros del*closure*, añadiéndolos también al entorno actual, y finalmente simplemente se evaluará el cuerpo con el nuevo entorno que se acaba de construir.

2.5 Átomos

Los átomos representan los elementos indivisibles que componen las expresiones y estructuras de datos. Los átomos son el elemento que pincipalemnte representa los datos: Números , símbolos, cadenas de texto, etc. Los átomos por lo general son contante, incluso los valores de las variables aunque podemos forzar a que esto no sea así.

Algunos de estos átomos son autoevaluables, lo que significa que su valor es el mismo que su representación. Los números y las cadenas de texto son ejemplos de átomos autovaluables. Por lo tanto cuando evaluemos esto elementos ya sea por si solos, durante el funcionamiento o de forma explicita con la función `eval` estos devolverán el mismo objeto.

```
42 -> 42; El número 42 se evalúa a sí mismo
(eval 42) -> 42
```

Los símbolos, por otro lado, no son autovaluables por defecto. Cuando se evalúa un símbolo, Lisp busca su valor en el entorno actual.

```
(define x 10)
x ; Se evalúa a 10, el valor asociado al símbolo x
```

Debido a su simplicidad, los átomos se pueden representar y manipular de manera eficiente en memoria. Los números y símbolos, por ejemplo, se almacenan de manera compacta.

Los símbolos se utilizan como identificadores y pueden tener un valor asociado en el entorno. Un símbolo tiene un nombre, que es una cadena de caracteres, y opcionalmente, un valor, una función y propiedades adicionales.

```
(setq my-symbol 'example) ; Define el símbolo example y le asigna a my-symbol
(symbol-name my-symbol) ; Devuelve "example"
(symbol-value my-symbol) ; Devuelve el valor asociado al símbolo, si lo tiene
```

Los símbolos son fundamentales en la programación Lisp porque permiten la creación de variables y la definición de funciones. Además, los símbolos pueden ser utilizados en macros y otros constructos de metaprogramación para manipular y generar código de manera dinámica.

2.5.1 Importancia de los Átomos en Lisp

Los átomos son cruciales para la simplicidad y flexibilidad de Lisp. Al proporcionar una base de tipos de datos indivisibles, permiten la construcción de estructuras de datos más complejas y la manipulación eficiente de código. La distinción clara entre átomos y listas ayuda a mantener la coherencia y la predictibilidad del comportamiento del lenguaje.

2.6 homoiconicidad

Una de las características mas notables de Lisp es el lenguaje como manipulación de la propia estructura del código, esto se debe a la homoiconicidad. En un lenguaje homoicónico,

el código y los datos comparten la misma estructura subyacente, lo que permite manipular el código como si fuera datos y viceversa. En Lisp tanto el código como los datos se representan mediante listas, estas listas pueden ser modificadas por el propio código. De esta forma el propio lenguaje puede evaluar y transformar las listas que constituyen el propio código.

Este atributo permite dos de las capacidades mas características de Lisp: los macros y la evaluación diferida.

- **Macros** Debido a que el código es representado como listas, es posible escribir macros que transformen el código antes de ser evaluado. Esto permite a los programadores crear nuevas estructuras de control y abstracciones.
- **Evaluación diferida** El control del código como estructura también permite el control de la propia evaluación del código. De forma que podemos crear código que solo se e bajo ciertas circunstancias

2.6.0.1 Macros

Los macros en Lisp permiten la manipulación y generación de código de formas que no son posibles o tan accesibles en otro tipo de lenguajes como C/C++. Esto Se debe a que las macros de Lisp actúan sobre el código como la estructura nativa de datos de Lisp.

Las macros en Lisp funcionan de una manera similar a como se hace en otros lenguajes, se declara un patrón con unos argumentos y una plantilla. El evaluador o compilador, tras generar el AST y antes de evaluar o compilar el código, expandirá los patrones del código a la plantilla declarada.

2.7 Evaluación

El proceso de evaluación en Lisp es un aspecto central del lenguaje, ya que define cómo se ejecutan las expresiones y se obtienen los resultados. La evaluación de una expresión en Lisp sigue un conjunto de reglas bien definidas, las cuales permiten interpretar y ejecutar el código de manera efectiva. A continuación, se presenta una descripción teórica del proceso de evaluación en Lisp, ilustrada con ejemplos de código Lisp.

2.7.1 Proceso de Evaluación

El proceso de evaluación en Lisp puede dividirse en varios pasos:

2.7.1.1 Evaluación de átomos

Si la expresión es un átomo autovaluable su valor es el mismo que su representación, Si es un símbolo, su valor se busca en el entorno actual.

```
42 ; devuelve 42
(setq x 10)
x ; devuelve 10
```

2.7.1.2 Evaluación de listas

Si la expresión es una lista, se evalúa la lista de acuerdo con el tipo de la primera subexpresión (la cabeza de la lista).

- Si la cabeza de la lista es un símbolo que se refiere a una función especial (como `if` o `quote`), se aplica la semántica especial correspondiente.
- Si la cabeza de la lista es un símbolo que se refiere a una función o a un*closure*, se evalúan los argumentos y se aplica la función al resultado de la evaluación de los argumentos.

```
(+ 1 2) ; devuelve 3
(if t 1 2) ; devuelve 1
```

2.7.1.3 Evaluación de macros

Las macros permiten la transformación del código antes de su evaluación. La macro expande en una nueva expresión que luego se evalúa de acuerdo con las reglas estándar.

```
(defmacro unless (cond body)
  `(if (not ,cond) ,body))
(unless nil (+ 1 2)) ; devuelve 3
```

2.7.1.4 Aplicación de Funciones

Cuando se llama a una función, Lisp evalúa primero todos los argumentos de la función. Luego, el cuerpo de la función se evalúa en un entorno extendido en el que los parámetros formales de la función están vinculados a los valores de los argumentos actuales.

```
(defun add (a b)
  (+ a b))
(add 3 4) ; devuelve 7
```

2.7.1.5 Evaluación Paso a Paso

Para ilustrar el proceso de evaluación paso a paso, consideremos la evaluación de una expresión más compleja:

```
(defun factorial (n)
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))

(factorial 3)
```

El proceso de evaluación de `(factorial 3)` sería el siguiente:

1. Evaluación del símbolo `factorial`, busca factorial en el entorno y devuelve un *closure*
2. Evalúa los argumentos , el literal 3 se autoevalúa a si mismo
3. Aplica la función, se evalua el cuerpo de `factorial` que es `(if (<= n 1) 1 (* n (factorial (- n 1))))` con `n = 3` en el entorno
4. Recursión, se realiza el mismo proceso con el código `(factorial 2)` y volvemos a punto 1
5. Fin de la recursión, una vez se llegue a la condición de final de la recursión se devolverá al primer nivel el resultado de `(factorial (- n 1))` siendo este 2 y evaluando finalmente `(if (<= 3 1) 1 (* 3 2))` devolviendo el resultado final `5`

3 Estructura de un compilador

Un compilador es un programa que traduce programas escritos en un lenguaje de alto nivel a un lenguaje de bajo nivel o lenguaje máquina, comprensible por el hardware de la computadora. La estructura de un compilador se compone de varias fases y componentes que trabajan en conjunto para realizar esta traducción de manera eficiente y correcta. A continuación, se describirán las principales fases y componentes de un compilador típico visibles en el esquema 3.1.

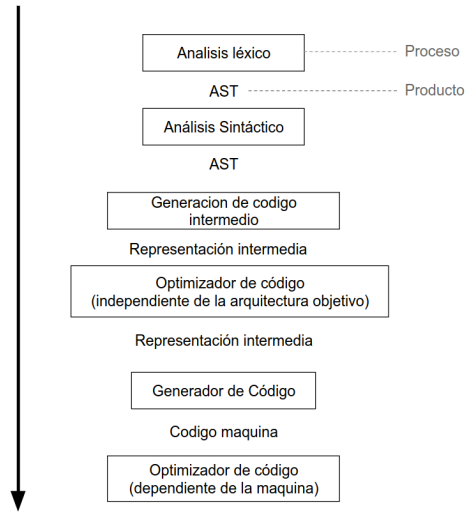

Figura 3.1: Diagrama de las fases de compilación

3.0.1 Análisis Léxico

El análisis léxico es la primera fase de un compilador. También conocido como escaneo, esta fase convierte el código fuente en una secuencia de tokens. Un token objeto a través del cual el compilador asimila cada lexema ya sea una palabra clave, un identificador, un operador, o un delimitador.El componente que realiza esta tarea se llama analizador léxico o lexer. El lexer elimina los espacios en blanco y los comentarios y según el lenguaje o el compilador puede tener mas capacidades

3.0.2 Análisis Sintáctico

La segunda fase es el análisis sintáctico o parsing. En esta etapa, la secuencia de tokens generada por el análisis léxico se organiza en una estructura de datos llamada árbol de sintaxis

abstracta (AST). El AST representa la estructura gramatical del programa de acuerdo con las reglas de la gramática del lenguaje de programación. El analizador sintáctico debe asegurare de que el programa sigue las reglas gramaticales del lenguaje.

3.0.3 Análisis Semántico

En la fase de análisis semántico, el compilador asegura que las operaciones en el código fuente tengan sentido semántico. Por ejemplo, que las variables sean declaradas antes de ser usadas, que los tipos de datos sean compatibles en las operaciones, y que las funciones sean llamadas con los argumentos correctos. El analizador semántico también puede realizar la resolución de nombres y la comprobación de tipos.

3.0.4 Generación de Código Intermedio

En esta fase, el compilador traduce el AST a un código intermedio. El código intermedio es una representación abstracta que es más cercana al lenguaje máquina que el código fuente, pero todavía independiente de la arquitectura específica del hardware. Al ser un lenguaje independiente de la plataforma es el lugar ideal para realizar optimizaciones de código además de facilitar la posterior traducción al código maquina de la correspondiente plataforma.

3.0.5 Optimización de Código

La optimización de código es una fase opcional pero importante, donde el compilador mejora la eficiencia del código intermedio. Las optimizaciones pueden incluir la eliminación de código muerto, la propagación de constantes, la reducción de fuerza, entre otras técnicas. El objetivo es generar un código más eficiente en términos de tiempo de ejecución y uso de recursos.

3.0.6 Generación de Código

En la fase de generación de código, el código intermedio optimizado se traduce a código máquina específico de la arquitectura del hardware destino. Esta fase produce el código objeto, que es un programa ejecutable en el sistema de destino.

3.0.7 Enlazado

Finalmente, el enlazado es el proceso de combinar el código objeto generado con bibliotecas externas y otros módulos para producir un ejecutable completo. El enlazador resuelve referencias a funciones y variables que están definidas en otros módulos o bibliotecas.

4 Análisis léxico

El análisis léxico es la primera fase de un compilador, donde se transforma el texto fuente en una lista de tokens. Un token es un objeto que representa una unidad sintáctica significativa, como identificadores, palabras clave, operadores, delimitadores o literales. Este proceso facilita la comprensión del código fuente de cara al analizador sintáctico, que se encargará de la sintaxis.

Normalmente el analizador léxico, o lexer, realiza este proceso en dos etapas principales: la segmentación y la categorización. Durante la segmentación, el lexer divide la entrada en unidades sintácticas llamadas lexemas. En la categorización, estos lexemas se convierten en tokens asociados a categorías específicas definidas por la gramática léxica del lenguaje.

Sin embargo, además de identificar lexemas el analizador sintáctico también puede llevar a cabo otras tareas, como el preprocesamiento del texto eliminando de los espacios en blanco que no son sintácticamente relevantes o los saltos de línea. Además de poder encargarse por ejemplo de la identificación de comentarios en el código.

A su vez puede ser un primer elemento en un buen sistema de control de error, ya que tiene contacto con el código escrito una práctica habitual es que haga una copia del propio código y/o lleve la cuenta de los saltos de línea para que más adelante, si se produce un error, pueda indicarle al programador en que parte del código ha sucedido. De la misma forma, según el lenguaje, el propio analizador sintáctico puede llevar a cabo la expansión de macros.

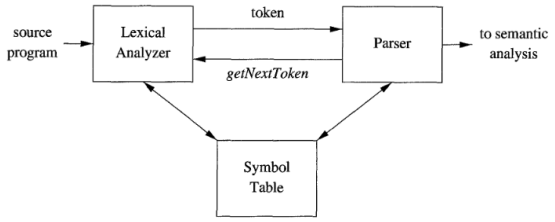

Figura 4.1: Enter Caption

4.1 Tokens

Los Tokens son el principal producto del análisis léxico. Estos pueden ser desde una cadena de texto, un struct con información adicional hasta un objeto con sus propios métodos. Una de las primeras decisiones de diseño al abordar el proyecto de crear un compilador es la estructura de los tokens iniciales en los que vas a dividir tu código. Algunos de los atributos más utilizados son:

- un **id** único para cada uno de los toques como identificador inequívoco

- un **type_id** que identifique el tipo de token (operación, expresión, literal, ...)
- El **numero de linea**, para el control de errores de compilación y favorecer la depuración
- **data** o **value**, un string (o el tipo correspondiente) conteniendo la información del token,

4.2 Técnicas de análisis léxico

4.3 Expresiones regulares

las expresiones regulares no son un método en sí mismo sino más bien una potente herramienta que puede ser utilizada por múltiples métodos además de por el parser.

Una expresión regular (regex o regexp, a veces expresión racional) es una secuencia de caracteres que especifica un patrón de búsqueda dentro de un texto. Generalmente, estos patrones se utilizan en algoritmos de búsqueda de cadenas para operaciones de búsqueda o de reemplazo en cadenas de texto, o para la validación de entradas.

Las expresiones regulares (regex) tienen una serie de normas básicas que permiten construir patrones de búsqueda y manipulación de texto. En el análisis sintáctico nos sirven para describir de forma formal los tokens que debemos analizar, así como para identificarlos dentro del texto. Pos ejemplo si quisiéramos que nuestros nombre de variables permitieran tanto letras como números, pero que no comenzar por un dígito podríamos usar la siguiente cadena:

```
regex_id = [a-zA-Z]{1}([a-zA-Z]|\\d)*
```

Este regex podemos desgranarlo en los siguientes elementos:

- `[a-zA-Z]`: carácter del abecedario tanto mayúsculas como minúsculas
- `{1}`: la regla anterior sólo se aplica a un carácter
- `([a-zA-Z]|\d)` : coincidirá con caracteres tanto de la a a la z como numéricos, d es una forma rápida de escribir `[0-9]`, mientras que el operador `|` (or) nos permite que el carácter coincida con cualquiera de los dos patrones
- `*` : la regla anterior buscará coincidir con todos los caracteres consecutivos que pueda con un mínimo de 0

Además de para identificar nuestros tokens también nos sirve como lenguaje para definirlos de manera formal, es común utilizar regex en forma de árbol para describir la forma de nuestros tokens. Por ejemplo, podríamos definir la estructura del anterior id de la siguiente forma.

```
1  digit -> 0|1|2|3|4|5|6|7|8|9
2  letter -> a|b|c|...|z
3  id -> letter (letter | digit)*
```

a partir de aquí podríamos seguir definiendo lexemas de lenguaje , por ejemplo un lexema `number` que cubra tanto lo enteros, los decimales y la notación científica de la forma 23e-2

```
digits -> digit digit*
number -> digits(. digits)?(E [+-]?digits)?
```

De esta forma podemos ir definiendo de una forma incremental los lexemas de nuestro lenguaje. Entre las principales ventajas de la regex como sistema es el poder de definir complejos patrones de una forma compacta y pueden llegar a ser más eficientes que el resto de los métodos.

4.3.1 Maquina de estados finitos

Una máquina de estados finitos es un modelo matemático que define un autómata con un número finito de estados, uno de los cuales es el estado inicial, y uno o más estados pueden ser estados finales o de aceptación. Las FSM son uno de los principales enfoques en el análisis léxico.

Una FSM para análisis léxico funciona de la siguiente manera:

1. Estados y Transiciones: La máquina consta de un conjunto de estados y transiciones entre estos estados. Cada transición está asociada a un símbolo o un conjunto de símbolos del alfabeto de entrada.

2. Estado Inicial: Es el estado en el que la FSM comienza su procesamiento.

3. Estados de Aceptación: Son los estados en los que, si se llega al final de la cadena de entrada, se considera que la cadena ha sido reconocida por la máquina.

4. Procesamiento de Entrada: La FSM procesa la cadena de entrada símbolo por símbolo, moviéndose de un estado a otro según las transiciones definidas.

Por ejemplo, en la figura 4.2, se muestra un grafo que representa una FSM utilizada para el reconocimiento de identificadores en un lenguaje de programación. Esta máquina empieza en el estado inicial y transita a diferentes estados basándose en los caracteres de entrada hasta llegar a un estado de aceptación, indicando el final de un identificador válido.

Comúnmente las FSM se implementan utilizando tablas de transición y estructuras de datos eficientes para representar los estados y transiciones. Este enfoque permite una rápida determinación de la transición siguiente y una fácil modificación de la máquina de estados, una tabla de estados de nuestra máquina de ejemplo sería 4.1

La máquina de estados finitos ofrece una solución eficiente y estructurada para el reconocimiento de lexemas en nuestro texto además de ser rápidas y simples de implementar, la mayor dificultad de este método reside en el diseño de la máquina de estados

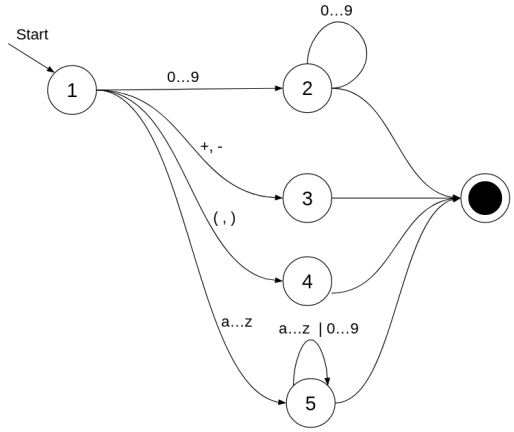

Figura 4.2: Grafo de la máquina de estados finito

	0...9	),(	a..z	+,-
1	2	4	5	3
2	2	-	-	-
3	-	-	-	-
4	-	-	-	-
5	5	-	5	-

Tabla 4.1: Tabla de estados de la FSM

5 Análisis sintáctico

El parser o analizador sintáctico, es uno de los principales componentes del compilador que se encarga de interpretar y estructurar los tokens resultantes de analisis lexico en una estructura jerárquica en forma de árbol (AST, Abstract syntax tree). Esta estructura refleja la gramática del lenguaje y las relaciones sintácticas entre los distintos elementos del código.

El analizador sintáctico permite al compilador tener una, o varias, estructura de datos que contenga la información completa del código para entender la lógica del programa.

En el flujo de trabajo de un compilador, el parser se encuentra entre el análisis léxico y el análisis semántico, actuando como un puente que transforma una secuencia lineal de tokens en una representación estructurada y comprensible del código fuente

5.1 Control de errores

El análisis sintáctico también es el responsable de encontrar los errores gramaticales que pueda haber en el código antes de que estos se propaguen a fases posteriores. El control de errores es un sistema general del programa por lo que dependerá también del control de errores del que disponga el analizador léxico

5.2 Gramaticas formales

Las gramáticas formales son una herramienta principal en la teoría de lenguajes de programación y en la construcción de compiladores e intérpretes. Estas gramáticas proporcionan una forma precisa y matemática de definir la sintaxis de un lenguaje, permitiendo describir de manera formal qué secuencias de símbolos (tokens) constituyen programas válidos en ese lenguaje. La teoría de gramáticas formales fue introducida por Noam Chomsky en la década de 1950 y ha sido una base crucial para el desarrollo de la informática teórica y la ingeniería de software.

Durante el diseño de un nuevo lenguaje de programación, una gramática formal proporciona una descripción clara y precisa de la sintaxis del lenguaje. Esto incluye la definición de la estructura de las declaraciones, expresiones, y otros constructos del lenguaje. La gramática sirve como una especificación que guía la implementación del compilador o intérprete y asegura que los programas escritos en el lenguaje sean sintácticamente correctos.

Una gramática formal se define como un conjunto de reglas de producción que especifican cómo se pueden formar las cadenas en un lenguaje. Formalmente, una gramática se puede definir como $G = (N, \Sigma, P, S)$, donde:

- N es un conjunto finito de símbolos no terminales, que representan categorías sintácticas.
- Σ es un conjunto finito de símbolos terminales, que son los tokens del lenguaje.

- P es un conjunto de reglas de producción, cada una de la forma $\alpha \to \beta$, donde α y β son cadenas de símbolos en $(N \cup \Sigma)^*$.
- S es el símbolo inicial, un no terminal desde el cual se generan todas las cadenas del lenguaje.

Para ilustrar cómo se relaciona esto con la teoría, consideremos un ejemplo sencillo de una gramática para un lenguaje de expresiones aritméticas básicas:

$$G = (\{E, T, F\}, \{+, *, (,), id\}, P, E)$$

donde $N = \{E, T, F\}$ son los no terminales, $\Sigma = \{+, *, (,), id\}$ son los terminales, y $S = E$ es el símbolo inicial. Las reglas de producción P son:

$$\begin{array}{lcl} E & \to & E + T \\ E & \to & T \\ T & \to & T * F \\ T & \to & F \\ F & \to & (E) \\ F & \to & id \end{array}$$

Esta gramática define cómo se pueden construir expresiones aritméticas válidas en este lenguaje. Por ejemplo, la cadena $id + id * id$ puede ser derivada a partir del símbolo inicial E siguiendo las reglas de producción de la siguiente manera:

$$\begin{array}{lcl} E & \to & E + T \to \\ & \to & T + T \to \\ & \to & F + T \to \\ & \to & id + T \to \\ & \to & id + T * F \to \\ & \to & id + F * F \to \\ & \to & id + id * F \to \\ & \to & id + id * id \end{array}$$

Al aplicar estas reglas de producción, podemos construir un árbol de sintaxis abstracta (AST) que representa la estructura gramatical de la expresión. Este árbol puede ser utilizado por el compilador para analizar y generar el código correspondiente.

Las gramáticas formales, como la del ejemplo anterior, son importantes para el diseño y la implementación de lenguajes de programación. Proporcionan un marco teórico que asegura que las construcciones sintácticas del lenguaje sean correctas y coherentes, facilitando así la construcción de herramientas de procesamiento de código como compiladores e intérpretes.

5.3 Jerarquías de gramaticas Chomsky

Las jerarquías de Chomsky son una clasificación de los lenguajes formales basada en su complejidad y poder expresivo. Esta clasificación, introducida por Noam Chomsky en 1956, divide los lenguajes en cuatro tipos: Tipo 0 (Lenguajes Recursivamente Enumerables), Tipo 1 (Lenguajes Sensibles al Contexto), Tipo 2 (Lenguajes Libres de Contexto) y Tipo 3 (Lenguajes

Regulares). Cada nivel de la jerarquía corresponde a un tipo de gramática y a un tipo de autómata que puede reconocer esos lenguajes.

5.3.1 Lenguajes Recursivamente Enumerables (Tipo 0)

Los lenguajes recursivamente enumerables son los más generales en la jerarquía de Chomsky. Cualquier lenguaje que puede ser reconocido por una máquina de Turing pertenece a esta categoría. Una gramática Tipo 0, también conocida como gramática ilimitada, se define formalmente como $G = (N, \Sigma, P, S)$, donde:

- N es un conjunto finito de no terminales.
- Σ es un conjunto finito de terminales.
- P es un conjunto de producciones de la forma $\alpha \to \beta$, donde α y β son cadenas de símbolos en $(N \cup \Sigma)^*$ y $\alpha \neq \epsilon$.
- S es el símbolo inicial.

5.3.2 Lenguajes Sensibles al Contexto (Tipo 1)

Los lenguajes sensibles al contexto son menos generales que los recursivamente enumerables y pueden ser reconocidos por una máquina de Turing linealmente acotada. Una gramática Tipo 1, o gramática sensible al contexto, se define como $G = (N, \Sigma, P, S)$, donde las producciones tienen la forma $\alpha A \beta \to \alpha \gamma \beta$, con $A \in N$ y $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$, y γ no es la cadena vacía ($\gamma \neq \epsilon$).

5.3.3 Lenguajes Libres de Contexto (Tipo 2)

Los lenguajes libres de contexto son reconocidos por autómatas de pila y son muy importantes en la teoría de lenguajes de programación y en el análisis sintáctico. Una gramática Tipo 2, o gramática libre de contexto, se define como $G = (N, \Sigma, P, S)$, donde las producciones son de la forma $A \to \gamma$, con $A \in N$ y $\gamma \in (N \cup \Sigma)^*$.

5.3.4 Lenguajes Regulares (Tipo 3)

Los lenguajes regulares son los más simples y pueden ser reconocidos por autómatas finitos. Una gramática Tipo 3, o gramática regular, se define como $G = (N, \Sigma, P, S)$, donde las producciones son de la forma $A \to aB$ o $A \to a$, con $A, B \in N$ y $a \in \Sigma$.

Cada nivel en la jerarquía de Chomsky es un superconjunto del siguiente: los lenguajes regulares son un subconjunto de los lenguajes libres de contexto, que a su vez son un subconjunto de los lenguajes sensibles al contexto, los cuales son un subconjunto de los lenguajes recursivamente enumerables. Esta jerarquía proporciona una forma estructurada de entender y clasificar los lenguajes formales y sus respectivas gramáticas.

LL(1), FSA, etc

5.4 Tecnicas de parsing

5.4.1 Recursive descent parser

La principal característica del recursive descent parser es que e basa en la reclusión estructurando el parser con una función para cada norma gramatical. De esta forma es el propio código del parser el que estructuralmente refleja las normas gramaticales. En un método rápido y fácil de implementar. Así por ejemplo si tuviésemos una gramática como la siguiente:

```
Expr -> Term (('+' | '-') Term)*
Term -> Factor (('*' | '/') Factor)*
Factor -> '(' Expr ')' | NUMBER
```

En esta gramática tendríamos por ejemplo la función `parse_term`,

Código 5.1: Pseudocódigo de un analizador sintactico basado en recurión descendiente

```
FUNCTION parse_expr(tokens: LIST OF TOKENS) -> (AST, LIST OF TOKENS)
      term, rest_tokens = parse_term(tokens)
      RETURN parse_expr_rest(term, rest_tokens)
END FUNCTION

FUNCTION parse_term(tokens: LIST OF TOKENS) -> (AST, LIST OF TOKENS)
   factor, rest_tokens = parse_factor(tokens)
   RETURN parse_term_rest(factor, rest_tokens)
END FUNCTION

FUNCTION parse_factor(tokens: LIST OF TOKENS) -> (AST, LIST OF TOKENS)
   IF tokens[0] == '(' THEN
      expr, rest_tokens = parse_expr(tokens[1:])
      EXPECT(rest_tokens[0] == ')')
      RETURN expr, rest_tokens[1:]
   ELSE
      RETURN parse_number(tokens)
END FUNCTION
```

A la hora de analizar un `Term` tenemos una función `parse_term` que e encargaría de comprobar y construir de la estructura de la Expresión, y que a la hora de llegar al termino `parse_factor` llamaría a la función `Term-parse` en una llamada recursiva a funciones que va describiendo la gramática hasta que termina en un elemento terminal.

Este enfoque de análisis sintáctico cuenta con las siguiente ventajas

- Claridad: La estructura del código es directa y sigue la gramática del lenguaje.
- Modularidad: Cada regla gramatical tiene una función correspondiente.
- Simplicidad: Fácil de implementar y entender para gramáticas simples.

Sin embargo su principal desventaja es el rendimiento, basar su estructura en la recursión hace que muchos otros métodos la superen en este aspecto.

	(	)	a	+	$
S	2	-	1	-	-
F	-	-	3	-	-

Tabla 5.1: Tabla de transición de estados

5.4.2 LL

Lo parsers de la forma LL tanto LL(1) como LL(n), son analizadores descendentes (Top-Down), este algoritmo lee las entradas de izquierda a derecha y solo puede analizar gramáticas que tengan derivaciones por la izquierda, a las gramáticas que podemos analizar con LL se les llama LL grammars. Este algoritmo utiliza una pila de parsing, que va modificando según una tabla de parsing donde está la infracción de la gramática del lenguaje. Esta tabla contiene la norma gramatical que se debe aplicar según el token encontrado. El analizador funciona así:

1. La pila se inicia con el simbolo inical dela gramatica y un final de cadena
2. Lee el proximo simbolo
3. Si es terminal: Si el símbolo en la parte superior
   de la pila coincide con el símbolo de entrada, ambos se eliminan

   Si no es terminal: Si el símbolo en la parte superior de la pila es un no terminal, el parser consulta la tabla de parsing para determinar la producción a aplicar, reemplazando el no terminal con los símbolos de la producción.
4. Error: Si no hay una regla de produccion valida para el simbolo actual se produce un error
5. Final: El parsing se completa exitosamente cuando tanto la pila de parsing como la entrada se han agotado.

Un analizador LL(n) funciona igual que un LL(1) pero este toma n símbolos de entrada en lugar de 1 para tomar decisiones. De esta forma el algoritmo es el mismo pero en el paso 3 leería n símbolos.

Un ejemplo de gramática LL(1) seria

```
A -> B
B -> (A + A)
B -> a
```

y su tabla de reglas:

5.4.3 RL

Los analizador RL al contrario que los LL siguen una estructura Bottop-Up, es decir analiza el texto de entrada de izquierda a derecha y produce una derivación más a la derecha en orden inverso, realizando un análisis sintáctico ascendente, al igual que con los parsers LL se indica en forma de RL(k) el número k de tone que va a mirar para decidir cómo analizar. La principal ventaja de estos analizadores es su tiempo de ejecución lineal, lo que los hace una gran elección

si lo que queremos es optimizar el tiempo de compilación de nuestro programa. LR requiere un stack y una tabla.

```
1. Se inicializa el Stack a 0
   Se inicializa el InputBuffer con la cadena de entrada
2. Bucle, el bublle continua hasta
3. Obtener estado actual
4. Segun el estado y la cadena de entrada consulta la tabla de accion para decidir
   la operacion a realizar
5. Ejecuta la accion segun cual se haya decidido:
   5.1 shift:
           mueve el simbolo de entrada al stack
           cambia el etado actual al espedificado en la tabla
           el simbolo de entrada se elimina del buffer
   5.2 reduce
           Se aplica una norma gramatical A->b
           Se eliminan del stack tantos simbolos como tenga b
           Se obtiene el nuevo estado de la tabla para A
           Se agrega A y el nuevo estado al stack
   5.3 accept
           Finaliza
   5.4 error
           Finaliza
```

5.4.4 Herramientas de generación de parsers

Debido a la complejidad que puede llevar la sola implementación de un analizador sintáctico, a lo largo de los años han surgido múltiples herramientas para asistir a los programadores. Estas herramientas automatizan gran parte del proceso de creación de parsers, facilitando la implementación y garantizando la corrección de los mismos. A continuación, se describen algunas de las herramientas más populares:

5.4.4.1 ANTLR

ANTLR (Another Tool for Language Recognition) es una poderosa herramienta para la generación de parsers, que permite crear parsers para gramáticas LL(*) de forma automática. ANTLR proporciona un entorno completo para definir gramáticas y generar el código necesario en varios lenguajes de programación. A continuación podemos ver un ejemplo de la descripción de la gramática en ANTLR.

```
1  grammar Expr;
2  expr: expr ('*'|'/') expr
3      | expr ('+'|'-') expr
4      | INT
5      | '(' expr ')';
6  INT: [0-9]+;
```

```
7       WS: [ \t\r\n]+ -> skip;
```

5.4.4.2 Yacc/Bison

Yacc (Yet Another Compiler Compiler) y su versión GNU Bison son herramientas clásicas para la generación de parsers LR(1). Estas herramientas permiten definir gramáticas independientes del contexto y generan el código C necesario para el parser.

Una de sus principales ventajas es la generacion de parsers eficientes para gramaticas LR(1) y la intergración con lexers generados a traves de lex/flex, a conntinuacion podemo ver un ejemplo de la defincion de una gramtica para yacc.

Código 5.2: Definición de una gramatica formal en yacc

```
1       %token NUMBER
2       %%
3       expr: expr '+' term
4           | expr '-' term
5           | term;
6       term: term '*' factor
7           | term '/' factor
8           | factor;
9       factor: '(' expr ')'
10            | NUMBER;
```

5.4.4.3 mpc

MPC (Micro Parser Combinator) es una biblioteca en C que facilita la construcción de analizadores sintácticos mediante el uso de combinadores de parsers. Los combinadores de parsers son una técnica de análisis sintáctico que permite construir parsers complejos a partir de parsers más simples, utilizando combinaciones funcionales. Al igual que el resto de generadores nos permite la declaración de nuestra gramática de forma declarativa, como podemos ver en el siguiente código extraído de uno de los prototipos:

Código 5.3: Definición de una gramatica formal en mpc.c

```
1 int main(int argc, char **argv)
2 {
3    mpc_parser_t *Form = mpc_new("form");
4    mpc_parser_t *Definition = mpc_new("definition");
5    mpc_parser_t *Expression = mpc_new("expression");
6
7    mpc_parser_t *Number = mpc_new("number");
8    mpc_parser_t *Operator = mpc_new("operator");
9    mpc_parser_t *Expr = mpc_new("expr");
10   mpc_parser_t *Program = mpc_new("program");
11   mpc_parser_t *Lispy = mpc_new("Lispy");
12
13   mpca_lang(MPCA_LANG_DEFAULT, "
```

```
14          form : <definicion> | <expression> \
15          definition : <variable_definition>
16                     | <syntax_definition>
17                     | (begin <definition>*)
18          number : /[0-9]+/ ; \
19          operator : '+' | '-' | '*' | '/' | '%' | \"max\" | \"min\" ;\
20          expr : '(' <operator> <expr>+ ')' | <number> ; \
21          Lispy : /^/ <expr> /$/ ; \
22       ",
23            Number, Operator, Expr, Lispy);
```

5.4.4.4 LLVM

LLVM, anteriormente "Low Level Virtual Machine," es un proyecto de infraestructura de compiladores que proporciona múltiples herramientas y librerías para construir compiladores y sistemas de análisis de código. Este proyecto es especialmente relevante por su presencia en muchos proyectos y lenguajes modernos, incluyendo el compilador Clang para C/C++, Rust, Swift.

La mayor ventaja de utilizar LLVM es el conjunto de herramientas que tiene para cada paso de la compilación desde el análisis léxico hasta la optimización y generación de código. Además una de las facilidades de estas herramienta es que te permite definir la gramática del lenguaje de una notación muy similar a la notación general para definir gramáticas formales. Una gramática simple, por ejemplo, se definiría de la siguiente forma:

Código 5.4: Definición de una gramatica formal en LLVM

```
grammar Expr;

prog: expr EOF;

expr: expr ('*'|'/') expr
    | expr ('+'|'-') expr
    | INT
    | '(' expr ')'
    ;

INT: [0-9]+;

WS: [ \t\r\n]+ -> skip;
```

5.4.4.5 conclusiones de porque no elegí hacerlo con un generador de parser

La razón principal que me llevó a no elegir un generador de parser como herramienta para mi TFG fue mi interés en explorar los algoritmos de parseo. Más adelante, me di cuenta de que no necesitaba estos métodos para Lisp debido a su sintaxis simple y su estructura homoicónica. sin embargo este enfoque me sirvió para profundizar en los fundamentos teóricos del análisis sintáctico y comprender mejor las mecánicas internas del lenguaje.

5.5 La gramática de Lisp/Scheme

Lisp plantea una cuestión respecto a la necesidad y utilidad de las gramáticas formales en su definición y análisis. En Lisp, la sintaxis está basada en una estructura muy sencilla: todo es una lista, y las listas están delimitadas por paréntesis. Esta simplicidad, combinada con la homoiconicidad (donde el código y los datos comparten la misma estructura), hace que el análisis sintáctico de Lisp sea más directo en comparación con otros lenguajes de programación.

Podemos encontrar tanto en la especificación oficial de Scheme como en algunas de sus implementacoines las definiciones formales de su gramática como se puede ver en la figura 5.1.

⟨lambda expression⟩ ⟶ `(lambda` ⟨formals⟩ ⟨body⟩`)`
⟨formals⟩ ⟶ `(`⟨identifier⟩`*)` | ⟨identifier⟩
| `(`⟨identifier⟩$^{+}$ `.` ⟨identifier⟩`)`
⟨body⟩ ⟶ ⟨definition⟩* ⟨sequence⟩
⟨sequence⟩ ⟶ ⟨command⟩* ⟨expression⟩
⟨command⟩ ⟶ ⟨expression⟩

⟨conditional⟩ ⟶ `(if` ⟨test⟩ ⟨consequent⟩ ⟨alternate⟩`)`
⟨test⟩ ⟶ ⟨expression⟩
⟨consequent⟩ ⟶ ⟨expression⟩
⟨alternate⟩ ⟶ ⟨expression⟩ | ⟨empty⟩

Figura 5.1: Gramática de `lambda` e `if` extraída de la especificación R7RS

La simplicidad y homoiconicidad de Lisp, donde tanto el código como los datos se representan como listas, hacen que el análisis sintáctico de Lisp sea relativamente sencillo. El propio lenguaje Lisp puede manipular su código como datos, permitiendo construir y transformar programas de manera dinámica. Esta característica reduce la necesidad de gramáticas formales complejas para el análisis sintáctico

Sin embargo, definir una gramática formal para Lisp tiene sentido y es útil en varios contextos: proporciona una especificación clara y precisa de la sintaxis del lenguaje, lo cual es valioso para la documentación y la enseñanza; ayuda a definir e implementar las reglas de parseo de manera sistemática y estructurada, lo que es crucial en la construcción de compiladores e intérpretes; ofrece una base sólida para asegurar que las nuevas construcciones sintácticas sean consistentes y correctas al diseñar nuevas variantes de Lisp o extender el lenguaje; y beneficia herramientas como formateadores de código y analizadores estáticos, permitiéndoles entender y manipular el código Lisp de manera precisa.

6 implementación de un prototipo

6.1 Busqueda del minimo a implementar para hacer Scheme

Una de las primeras incógnitas al comenzar con un prototipo era definir las especificaciones concretas del lenguaje que tenía que compilar. Ya que Lisp/Scheme cuenta con una gran extensibilidad la idea principal era generar un núcleo funcional del lenguaje a partir del cual se pudiese generar, en el propio lenguaje, el resto de este hasta completar algo cercano a la especificación R5RS.

Encontrar un núcleo del lenguaje que implementar brindaba múltiples ventajas. En primer lugar, debido a la expansibilidad de Lisp a través de los macros podía extender la gramática, por lo que conocer un núcleo mínimo del lenguaje amplificaría significativamente la gramática del lenguaje, lo que haría más simple y rápida la implementación.

Esta simplificación era importante, ya que me permitía centrarme en las estructuras fundamentales y garantiza que el compilador sea más manejable y menos propenso a errores en las primeras etapas de desarrollo. Lo que me daba unos buenos cimientos para el resto de estructuras que tuviese que implementar por encima.

6.1.1 Nucleo

Finalmente si con la publicación The Roots of Lisp de Paul Graham 2002 (que hace referencia al trabajo original de John McCarthy que dio lugar al lenguaje Lisp). En esta publicación se parte de un conjunto mínimo de expresiones sobre listas y átomos, para desarrollar todo un lenguaje. Este es el primer núcleo en el que me base para definir cuales serían los elementos mínimos a implementar en mi Scheme. En la publicación se parte de un conjunto de 7 operadores sobre listas y átomos que se puede ver en la 6.1

TRoL	Scheme	decipcion
(quote x)	(quote x)	devuelve x sin evaluarlo
(atom x)	(atom? x)	devuelve #t si x e un atomo #f si no lo es
(eq a b)	(eq? a b))	devuelve #t i a y b son el mismo atomo
(car x)	(car x)	devuelve el primer elemento de la lista, si x = '(a b c) → (car x) = a
(cdr x)	(cdr x)	devuelve la lista x sin el primer elemento, (cdr '(1 2 3)) → (2 3)
(cons a b)	(cons a b)	creación de un par (a . b)
(cond	(cond	evalua cada q de 1 a n, de aquel q
(q_1 e_1)	(q_1 e_1)	que sea ciertos se evalua si e_n correspondiente
...	...	
(q_n, e_n))	(q_2, e_2))	

Tabla 6.1: Lista de los los 7 operadores básicos

A partir de las construcciones del paper adapte el léxico y los tipos de dato a Scheme, por ejemplo, mientras el paper utiliza 't como true y la lista vacía () como false, tanto true como false son tipos de datos booleano que debía implementar en Scheme , por lo que lo traduje directamente a booleanos.

Otra de las principales diferencias es el comportamiento de `(cons a b)`, mientras que en la publicación este operador espera que b sea una lista, sin embargo siguiendo los tipos de datos de Lisp donde una lista no es más que un par que contiene a otro, el comportamiento de `cons` Se establece como el normal de crear un par con a y b, lo que cubre tanto el comportamiento de añadir un elemento a una lista como el de la creación de un par.

Además la publicación establece dos elementos sintacticos `label` al que nosotros llamaremos define, para enlazar un valor a un identificador y `lambda` para crear funciones.

A partir de estas bases decidí añadir algunas extras para acercarme más a una implementación completa de Lisp

A partir de este pequeño núcleo durante se fueron añadiendo nuevas palabras reservadas y operadores según iba probando el lenguaje con código de Scheme.

6.2 Selección del lenguaje de implementación

6.2.1 C/C++

Las primeras aproximaciones hacia la creación del prototipo fueron a través de C, sin embargo esto fue antes de darme cuenta que la simpleza de la gramática de Lisp se reflejaria en la simpleza de mi analizador sintáctico. Esta elección se hizo principalmente pensando en que un lenguaje de bajo nivel me permitiría tener un mayor control sobre la asignación de memoria y la generación de código máquina. Sin embargo debido a la complejidad de los analizadores léxicos y sintácticos la sola implementación de estos y de los elementos básicos conllevaba un gran esfuerzo y tiempo.

Estos primeros acercamientos eran principalmente analizadores basados en máquinas de estados finitos, no siempre enfocadas hacia Lisp, sino más bien prototipos del propio algoritmo que más tarde podría generalizar a otros léxicos y gramaticales más complejas.

A continuación se muestra la función de `move` de una máquina finita de uno de las primeras aproximaciones a un analizador sintáctico.

Código 6.1: Función de control de la máquina de estados de un analizador sintáctico

```
1    // devuelve le resualdo de la tabla de la maquian de TokenTypes finitos
2// |---+-------+----------+---|
3// | | digit | operador | ; |
4// |---+-------+----------+---|
5// | 1 | 2 | - | - |
6// | 2 | 2 | 3 | 4 |
7// | 3 | 2 | - | - |
8// | 4 | - | - | - |
9// |---+-------+----------+---|
10enum TokenType move(enum TokenType s, enum TokenType old_s, char c, char ↩
    ↪ lexema[], TokenList *tokenlist)
11{
```

```
    enum TokenType next_s = VOID;
    Token t;

    next_s = lexer_matrix[s][c];

    if (next_s == VOID)
    {
        if (!strcmp(lexema, ""))
        {
            t = new_token(old_s, lexema);
            push_token(tokenlist, t);
            fseek(file, -1L, SEEK_CUR);
            strcpy(lexema, "");
        }
        return INICIO;
    }
    else
    {
        return next_s;
    }
}
```

Una de las particularidades de estas primeras implementaciones fue que, previendo los problemas que me surgirían al trabajar con árboles y listas encadenadas relacionadas por posiciones de memoria, cree funciones auxiliares que me estructuraran el árbol en la forma de grafo con las posicione de memoria, intentando con esto facilitar la depuración de estructuras tan delicadas, en la figura 6.1 podemos ver el grafo resultante de la relaciones entre los nodos en el AST en una de la ejecuciones.

Código 6.2: Funciones de creación de un grafo para visualizar relaciones entre direcciones de memoria

```
    void aux_draw_adress_node(Node *node, FILE *fp, int depth)
{
    if (node == NULL) {perror("aux_link node is NULL"); return ;}
    fprintf(fp, "%p, %p\n", node, node);
    for (int i = 0; i < node->numChildren; i++)
    {
        aux_draw_adress_node(node->children[i], fp, depth + 1);
    }
}



void aux_draw_adress_link(Node *node, FILE *fp, int depth)
{
    if (node == NULL) {perror("aux_link node is NULL"); return ;}
    for (int i = 0; i < node->numChildren; i++)
    {
        fprintf(fp, "%p, %p, true\n", node, node->children[i]);
        aux_draw_adress_link(node->children[i], fp, depth + 1);
```

```
    }
}


// // generate a gdf file to visualize the tree
void draw_adress(Node *node, int depth)
{
    FILE *fp;
    fp = fopen("adress.gdf", "w");
    printf("adress from head node: %p \n", node);
    fprintf(fp, "nodedef>name VARCHAR,label VARCHAR\n");
    aux_draw_adress_node(node, fp, 0);
    fprintf(fp, "edgedef>node1 VARCHAR,node2 VARCHAR, directed BOOLEAN\n");
    aux_draw_adress_link(node, fp, 0);
    fclose(fp);
}
```

from 2024-05-23 00-02-20.png from 2024-05-23 00-02-20.bb

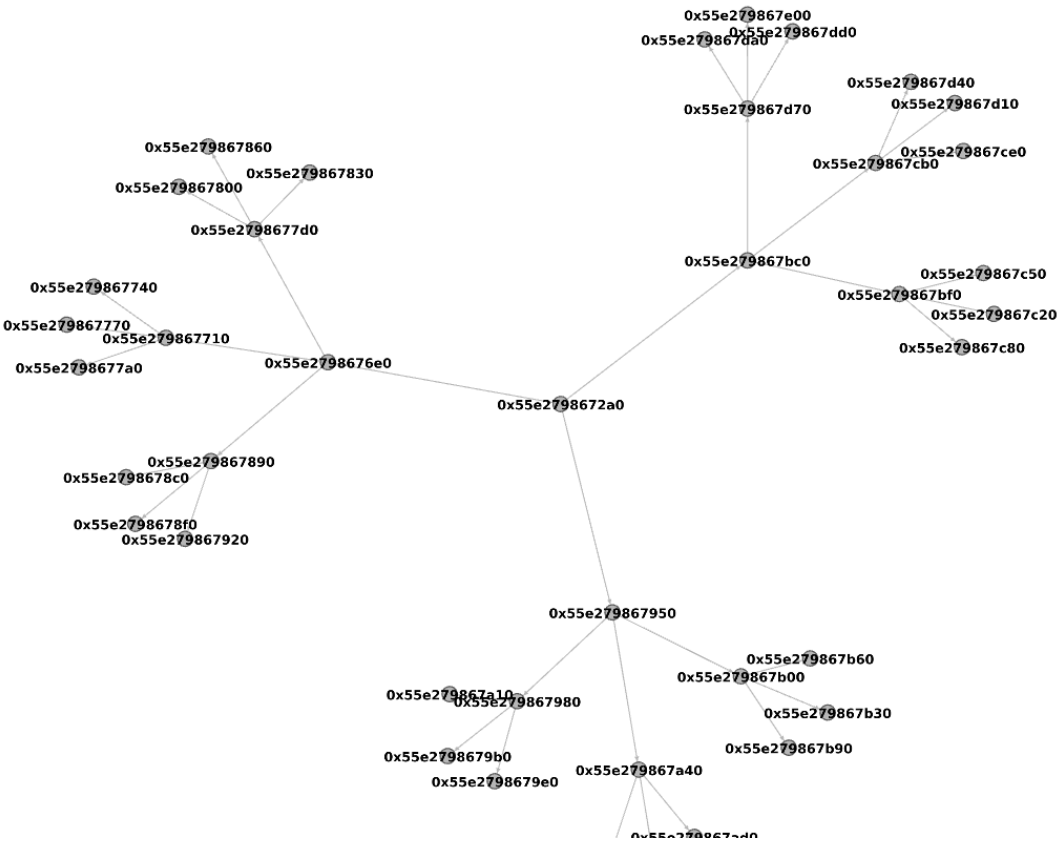

Figura 6.1: Árbol de referencias entre punteros

6.2.2 Scheme / bootstrapping

Un concepto que me parecía atractivo desde el principio era el concepto de bootstrapping, es decir, que la propia implementación del intérprete/compilador estuviese escrita en el mismo lenguaje que compila, es decir, que el lenguaje se compilarse a sí mismo. El camino más rápido para esto habría sido escribir mi código en Scheme, eligiendo cuidadosamente qué elementos del lenguaje utilizar por si alguno no terminaba siendo implementado en mi prototipo.

A continuación muestro algunas de las funciones que formaban parte del analizador léxico creado en Scheme.

Código 6.3: Sección del sistema de análisis lexico basado en recursión implementado en Scheme

```
1
2 ;; guarda las cifras
3 (define (lex-numero lista token tokens)
4   (cond ((null? lista) (cons (mk-token "NUMERO" token) tokens))
5         ((cifra? (car lista)) (lex-numero (cdr lista) (cons (car lista) token) ↩
              ↪ tokens))
6         (#t (lex-general lista'() (cons (mk-token "NUMERO" token) tokens)))))
7
8 ;; si es un signo y el siguiente es una cifra se convierte en numero
9 ;; si a continuacion hay cualquier otra cosa es un operador
10 (define (lex-signo lista token tokens)
11   (cond ((null? lista) (cons (mk-token "OPERADOR" token) tokens))
12         ((cifra? (car lista)) (lex-numero (cdr lista) (cons (car lista) token) ↩
              ↪ tokens))
13         (#t (lex-general lista '() (cons (mk-token "OPERADOR" token) tokens)))))
14
15
16 ;; parse los strings debn estar envueltos en ""
17 (define (lex-string lista token tokens)
18   (cond
19     ((null? lista) (cond ((eq? #\" (car token))
20                           (cons (mk-token "STRING" token) tokens))
21                          (#t
22                           error "Error lexico en string, falta cerrar comillas")))
23     ((eq? #\" (car lista))
24      (lex-general (cdr lista) '() (cons (mk-token "STRING" token) tokens)))
25     (#t (lex-string (cdr lista) (cons (car lista) token) tokens))))
```

6.2.3 Python

Finalmente, se optó por utilizar Python como el lenguaje de implementación para este proyecto debido a varias ventajas, principalmente en el contexto de desarrollo rápido y flexible permitiendo un desarrollo más rápido en comparación con lenguajes de bajo nivel como C o C++. Esta rapidez en la implementación fue crucial.

En el contexto de este proyecto, la capacidad de manipular estructuras de datos y realizar operaciones complejas de manera eficiente fue esencial. Python, con su soporte nativo para listas, tuplas y diccionarios, permitió una implementación más sencilla y directa de las

estructuras de datos de Lisp.

Además, Python soporta programación funcional, lo que se alinea bien con los paradigmas de Lisp, facilitando la implementación de funciones como `map`, `filter` y `reduce`. Esto no solo simplificó el proceso de codificación, sino que también mejoró la legibilidad y mantenibilidad del código.

Otra ventaja de Python es su capacidad para la depuración y el testing rápidos. Con herramientas integradas como el depurador de Python (`pdb`) pude identificar y corregir errores de manera más eficiente. Esta capacidad de iterar rápidamente sobre el código permitió realizar cambios y mejoras de manera ágil, adaptándose a los nuevos requerimientos o descubriendo mejores enfoques durante el desarrollo.

6.2.3.1 Analizador léxico

El analizador léxico desarrollado en este proyecto se caracteriza por su simplicidad, en la versión final, se optó por un enfoque que aprovecha las operaciones de listas de alto nivel en Python para mejorar la legibilidad y el mantenimiento del código. Además no de no requerir construcciones de objetos ya que se utiliza el *string* del lexema como tokens.

Código 6.4: Analizador lexico del prototipo final

```
def tokenize(code : str):
    code = code.replace("'()","null")
    code = code.replace('[',' ( ').replace(']',' ) ')
    code = code.replace('(',' ( ').replace(')',' ) ')
    code = code.replace('\n',' ')
    code = code.replace('\t',' ')
    code = code.replace("'"," ' ")
    code = code.replace(","," , ")
    code = code.replace("`"," ` ")
    tokens = code.split(' ')
    tokens = [t for t in tokens if t != '']
    tokens = [t for t in tokens if not is_blank(t)]
    if DEBUG: print("DEBUG: tokenize return: ",tokens)
    return tokens
```

6.3 Simplificación del análisis sintáctico en Lisp

Uno de los puntos más importantes de este trabajo fue darme cuenta no solo de la simpleza semántica de Lisp sino de la homoiconicidad, siendo que la estructura del código de Lisp es en sí misma una declaración del AST interno. Debido a esto el parser que requiere un lenguaje tipo Lisp es tremendamente simple, ya que solo hay que ir siguiendo la propia estructura.

Los únicos elementos que se salen de esta norma son lo elementos que suelen llamarse Syntactic sugar, o azúcar sintáctico, esto son pequeñas comodidades como acortar el uso de `(quote a)` a simplemente `a'a`, sin embargo, debido a la estructura de pares de Lisp la expansión de estas acotaciones también es simple.

Así todo el trabajo de la creación de un AST a partir de token se puede reducir a la siguiente función recursiva:

Código 6.5: Analizador sintáctico del prototipo final

```
def parse(tokens):
if not tokens:
    return ()

token = tokens.pop(0)
if token not in [OPEN, CLOSE,"'","`",","]:
    return token

# expansion de las quotes
if token == "'":
    new_pair = ("quote",())
    new_pair = append(new_pair, parse(tokens))
    return new_pair

if token == "`":
    new_pair = ("quasiquote",())
    new_pair = append(new_pair, parse(tokens))
    return new_pair

if token == ",":
    new_pair = ("unquote",())
    new_pair = append(new_pair, parse(tokens))
    return new_pair

if token == OPEN:
    if tokens[0] == CLOSE:
        tokens.pop(0)
        return ()
    kar = parse(tokens)
    new_pair = (kar, ())
    while tokens:
        if tokens and tokens[0] == CLOSE:
            tokens.pop(0)
            break
        new_pair = append(new_pair, parse(tokens))
    return new_pair
return ()
```

6.4 Evaluación

A la hora de diseñar la evaluación se ha intentado crear de forma que se puedan usar las propias funciones de evaluación en el propio lenguaje de forma que desde nuestro código podemos llamar a las funciones eval y apply. La evaluación se hace principalmente a través de dos funciones que se llaman mutuamente. La primera es `ieval` que es la encargada de recibir el código en forma de pares anidados y decidir qué hacer con ellos, si son átomos serán evaluados, pero puesto que la mayoría de tipos de datos son autoevaluables estos simplemente

se devuelven a sí mismos.

Las formas especiales tienen en sí mismas un significado sintáctico y no siguen las mismas normas que los *closures* sino que su evaluación depende de cada una. Entre estas formas especiales encontraríamos:

- **define** : se encarga de enlazar un identificador con un valor en el entorno actual, este valor puede ser tanto una lista como un átomo
- **cond** : `cond` es la base de las estructuras de control, a partir de esta podemos utilizar los macros para desarrollar otras como if o switch.
- **lambda** : lambda es la encargada de generar*closures*
- **quote, quasiquote y unquote** : se utiliza para prevenir la evaluación de una expresión, tratándola como un dato literal. quasiquote es similar a quote, pero permite la evaluación selectiva de partes de la expresión utilizando unquote.
- **begin** : `begin`permite agrupar múltiples expresiones para que se ejecuten secuencialmente en el entorno actual.

Puesto que siguen normas especiales de evaluación la lógica de estos operadores debe estar en el propio interprete/compilador. Por ejemplo en el funcionamiento de `cond` solo se evalúa aquel código cuya condición sea cierta. Este control preciso de la evaluación garantiza que las estructuras de control como `cond` o `begin` operen correctamente.

Código 6.6: Implementación de una forma especial del lenguaje

```
def cond(args, env):
if args == ():
    return None
estatements = pair_to_list(args)
for e in estatements:
    q = ieval(car(e), env)
    if e == ():
        return None
    if q:
        return ieval(cadr(e), env)
return None
```

Tras las normas especiales se evalúa el `car` de la lista o expresión que estamos tratando, este `car`, ya sea un átomo o una expresión en sí misma debe devolver un *closure*, en caso contrario el intérprete lanzará error, ya que no podrá aplicar el objeto a los argumentos de la expresión. Una vez establecido que nuestra lista es una *s-expression* con un operador válido podemos llamar a la función `apply`.

Código 6.7: Sistema de evaluación del prototipo final

```

def ieval(code, env):
```

```
    if atom_(code):
        return eval_atom(code, env)

    if car(code) in SPECIAL_FORMS:
        return eval_special_form(code, env)

    kar = car(code)
    kar = ieval(car(code), env)
    if not procedure_(kar):
        merror("not a procedure: "+stringify(kar))

    r = iapply(kar, cdr(code), env)

    if atom_(r) and r != ():
        r = eval_atom(r, env)

    if car(code) == "sumalista":
        pass

    if DEBUG: print("IEVAL: evaluate: ",stringify(code)," = ",stringify(r))
    return r
```

La función `apply` es la encargada de aplicar los procedimientos o *closures* sobre lo argumentos, esta función distingue entre dos tipos importantes de operadores los operadore *built-in*, aquellos que están integrados en el propio código del intérprete, y los procedimientos que hayan sido definido por el usuario. En ambos casos el primer paso es evaluar cada uno de los argumentos.

En caso de que el operador sea *built-in*, gracias a que Python también soporta programación funcional y podemos tratar a las funciones como objetos, podemos simplemente aplicar la función guardada en el propio *closure* de la forma `closure.body(args,env)`.

En caso de que el operador sea un procedimiento definido por el usuario, el proceso es ligeramente diferente. Primero, se crea un nuevo entorno extendido que incluye las variables locales del procedimiento. Luego, se enlazan los parámetros del procedimiento con los argumentos evaluados, y finalmente, se evalúa el cuerpo del procedimiento en el nuevo entorno.

Código 6.8: Funciones para aplicación de procedimientos del prototipo final

```


def iapply(closure, args, env):
    if APPLY_DEBUG: print(f"IAPPLY: aplying: {closure.name} ",stringify(closure↩
        ↪ .body))
    new_env = closure.env + env
    if closure.built_in:
        return iapply_built_in(closure, args, env)

    # bind the parameters
    if APPLY_DEBUG: print("IAPPLY: unevaluated args: ", stringify(args))
    aux_args = ieval_args(args, env)
```

```
    aux_params = closure.params
    while aux_params != () and aux_args != ():
        # dot notation
        if car(aux_params) == ".":
            new_env[0][car(cdr(aux_params))] = aux_args
            aux_args = ()
            aux_params = ()
            break

        new_env[0][car(aux_params)] = car(aux_args)
        aux_params = cdr(aux_params)
        aux_args = cdr(aux_args)

    if aux_params != () or aux_args != ():
        merror("Mismatched number of arguments and parameters MAPPLY")

    r = ieval(closure.body, new_env)
    if APPLY_DEBUG: print("IAPPLY: closure = ",stringify(closure.body), " = ",r↩
        ↪ ,"\n",
                    "\tparams = ",closure.params,"\n",
                    "\targs = ", args)
    return r


def iapply_built_in(closure, args, env):
    e_args = ieval_args(args, env)
    if closure.name == "eval":
        return closure.body(e_args, env)
    if closure.name in ['car','cdr']:
        return closure.body(car(e_args))
    return closure.body(e_args)
```

Este sistema de evaluación y aplicación de funciones permite que el intérprete de Lisp sea flexible y extensible, soportando tanto funciones integradas como definidas por el usuario, y manejando de manera eficiente la evaluación y la aplicación de*closures*.

6.5 Funciones auxiliares relevantes

6.5.1 pair_to_list

Una estrategia muy útil a lo largo del desarrollo fue poder transformar los pares de Lisp en las propias listas de Python, lo que me permitió usar las propias herramientas en Python para su manipulación ni tener que implementar nuevas funciones sobre mi estructura de tuplas. Para realizar esta transformación, implementé una función que convertía los pares anidados de Lisp en listas de Python. Un ejemplo de cómo esto simplifica el código lo podemos ver en la anterior definición de `cond`, donde transformó a la lista de argumento de `cond` en forma de listas anidadas a una lista de Python que me permite iterar entre ellas de forma cómoda.

Código 6.9: Conversión de pares a listas

```
1
2def pair_to_list(pair):
3    if pair == ():
4        return []
5    return [car(pair)] + pair_to_list((cdr(pair)))
6
7
8def list_to_pair(l):
9    if l == []:
10       return ()
11   return (l[0], list_to_pair(l[1:]))
```

6.6 Ventajas de utilizar tipos de datos nativos como estructuras

En uno de los prototipos se decidió que el objeto por el cual se representa las listas de Lisp sería un clase de Python con dos elementos, un `car` y un `cdr`, utilizando en esta implementación una traducción directa entre la lista vacía de Lisp y el objeto `None` de Python. Sin embargo, este enfoque llevó a problemas a la hora de manejar el comportamiento de las listas vacías.

La solución a esto fue representar los pares encadenados de Lisp a través de las tuplas de Python. El hecho de representar el objeto principal del código a través de un tipo de datos nativo del lenguaje de la implementación nos permite en primer lugar tener un abanico de herramientas de Python que nos permite la manipulación de este objeto, y además nos permite trabajar con un tipo de dato más cercano a las listas de Lisp que representa lo que hace más simple la implementación. A continuación puede verse la primera implementación del la clase `Pair`.

Código 6.10: Primera definición de un objeto Par

```
1
2
3class Pair:
4    def __init__(self,car,cdr):
5        self.car = car
6        self.cdr = cdr
7
8    def __str__(self):
9        return "( {} . {} )".format(str(self.car), str(self.cdr))
10
11   def __format__(self,fmt):
12       return self.__str__()
13
14   def add_tail(self,item):
15       if self.cdr == None:
16           self.cdr = Pair(item,None)
17       else:
18           self.cdr.add_tail(item)
```

```
19
20    (...)
```

Debido al cambio de tipo del principal objeto que manipula el lenguaje , se tuvo que reescribir todo el interprete, sin embargo al estar la lógica del intérprete ya estructurada esto no llevo demasiado tiempo.

6.6.1 Macros

Los macros era uno de los elementos más interesantes a la hora de realizar el prototipo ya que proporcionan una gran potencia al lenguaje para ser expandido. Ya se ha comentado como uno de los desafíos importante a la hora de realizar el prototipo fue encontrar un núcleo mínimo a implementar, esto se hizo con el conocimiento de que gracias a la potencia de los macros de Lisp podría ampliar el lenguaje incluso a formas que no siguen la evaluación normal como las estructuras de control o las formas *let*

Cabe destacar que no se ha conseguido implementar al completo todas los atributos de los macros de Lisp, sin embargo si se ha implementado un sistema de macros funcional que añade gran potencia al lenguaje.

A la hora de establecer un macro la lógica del programa es simple, ya que un macro viene dado por un nombre, un patrón y un cuerpo, de forma similar a un *closure*.

Código 6.11: Definición interna de un macro

```
1
2def defmacro(args, env):
3    if length(args) != 3:
4        merror("defmacro: wrong number of arguments")
5
6    if type(car(args)) != str:
7        merror("defmacro: first argument must be a symbol")
8
9    name = car(args)
10   params = car(cdr(args))
11   body = car(cdr(cdr(args)))
12   macros[name] = Macro(params, body, name)
```

A la hora de expandir estos macros sin embargo debemos recorrer todo el código, antes de que sea evaluado, buscando identificadores que apunten a macros para expandirlos. Gracias a los quotes, semiquotes y unquotes podemos estructurar el código por el que queremos hacer la substitución a la hora de escribir el macro, ya que este código será evaluado evaluando aquellas partes que nosotros queramos y dejando como estructura sin evaluar aquellas que no.

Código 6.12: Sistema final de expansión de macros

```
def expand_macro(code, env):
    if car(code) in macros:
        new_env = [{}] + env
        macro = macros[car(code)]
        lparams = pair_to_list(macro.params)
        largs = pair_to_list(cdr(code))
        for i in range(0,len(lparams)):
            new_env[0][lparams[i]] = largs[i]

        new_code = ieval(macro.body, new_env)
        return new_code


def expand_macros(code, env):
    if pair_(code):
        if car(code) in macros:
            return expand_macro(code, env)
        return (expand_macros(car(code), env), expand_macros(cdr(code), env))
    return code
```

6.7 Conclusiones

6.8 Diferencias con una implementación completa

6.8.0.1 Macros higiénicos

Los macros higiénicos en Scheme están diseñados para evitar conflictos de nombres entre las variables locales del macro y las variables en el contexto de la expansión del macro. Aunque esto mejora la seguridad y la predictibilidad del código, también introduce una complejidad significativa en su implementación. Los macros higiénicos requieren un análisis y transformación detallada de las expresiones para asegurar que no haya colisiones de nombres, lo cual puede ser complejo de implementar desde cero.

Durante el desarrollo del proyecto, se tomó la decisión de no implementar los macros higiénicos de Scheme y, en su lugar, se optó por utilizar un estilo de macros más cercano al de Lisp. Esta elección se debió a la complejidad de los macros higiénicos en Scheme.

Además el proyecto no requiere específicamente las características adicionales proporcionadas por los macros higiénicos. Los macros al estilo Lisp eran suficientes para las necesidades de metaprogramación y extensibilidad del lenguaje.

6.8.1 Soporte de caracteres y cadenas

Uno de los elementos básicos y fundamentales de cualquier lenguaje de programación es el soporte de cadenas de caracteres como tipo de dato. Las cadenas permiten la manipulación de texto, una funcionalidad esencial para muchas aplicaciones, desde el scripting hasta el

desarrollo web. Sin embargo, debido a limitaciones de tiempo de desarrollo del proyecto, no se logró integrar completamente el soporte para cadenas de caracteres en el sistema, a pesar de que se había comenzado a definir una clase para manejar estos objetos.

Código 6.13: Objeto para la representación interna de cadenas de texto

```
1
2class LispString:
3    def __init__(self, string):
4        self.string = string
5
6    def __str__(self):
7        return "\""+self.string+"\""
8
9    def copy(self):
10       return LispString(self.string)
```

6.8.2 Sintaxis de macros completa

La sintaxis actual de los macros es simple y efectiva, pero le faltan elementos de la sintaxis completa como los argumentos opcionales o las expansiones de patrones más avanzados. Estos elementos adicionales permitirían una mayor flexibilidad y potencia en la definición de macros, facilitando la creación de construcciones más complejas y específicas dentro del lenguaje.

6.9 Proyecciones de desarrollo futuro

6.9.1 Mejoras rápidas

- Soporte de cadenas como tipo de dato
- Implementación de todas las características de los macros como los parámetros opcionales
- Soporte de caracteres como tipo de dato
- Completitud de operadores sintácticos básicos como `set`

6.9.2 Operadores no implementados

El proyecto se centró en implementar un núcleo del lenguaje a partir del cual se pudiese desarrollar el resto, esto hizo que al ser la implementación completa un objetivo secundario finalmente solo se haya diseñado un subset de Scheme con macros de Lisp.

Ampliar el abanico de recursos de Scheme a partir de este núcleo sería mucho más sencillo, dado que el motor principal ya ha sido implementado. El enfoque inicial del proyecto era extender todo lo que no fuese estrictamente necesario a partir del propio lenguaje, minimizando así la dificultad de implementación de las partes complejas de un compilador. Este enfoque

permitió concentrar los esfuerzos en establecer una base sólida, sobre la cual se pueden añadir funcionalidades adicionales con mayor facilidad y menos riesgo de introducir errores.

6.9.3 Generación de Código y Optimizaciones

El proyecto inicialmente tenía como objetivo la creación de un compilador completo para un lenguaje basado en Lisp. Sin embargo, debido a limitaciones de tiempo, el desarrollo se centró en la implementación de un evaluador en lugar de un compilador completo.

Esto aborda las fases más complejas del desarrollo de un compilador, como la generación de código y las optimizaciones. Estas fases son críticas para transformar el código fuente en un código de máquina eficiente y optimizado, pero requieren un esfuerzo significativo en términos de diseño e implementación.

Implementar la generación de código habría implicado traducir las estructuras del lenguaje Lisp a un lenguaje intermedio o directamente a código de máquina, por otro lado, las optimizaciones son esenciales para mejorar la eficiencia del código generado. Esto incluye técnicas como la eliminación de código muerto, la optimización de bucles y la gestión eficiente de memoria. A pesar de estas limitaciones, el proyecto logró implementar un evaluador funcional, que sirve como una sólida base para futuras extensiones. Un evaluador permite la ejecución inmediata de código Lisp en un entorno controlado, facilitando el desarrollo y la depuración de la futura implementación de un compilador.

Bibliografía

Alfred V. Aho, R. S., Monica S. Lam, y Ullman, J. D. (1985). *Compilers: Principles, techniques, and tools.*

Cooper, K. D. (2003). *Engineering a compiler.*

Ghuloum, A. (2006). *An incremental approach to compiler construction.*

Graham, P. (1984). *On lisp: Advanced techniques for common lisp.*

Graham, P. (2002). *The roots of lisp.*

Grune, C. J. . D. (1998). *Parsing techniques: A practical guide.*

Harold Abelson, J. S., Gerald Jay Sussman. (1984). *Structure and interpretation of computer programs.*

l. Holub, A. (1990). *Compiler design in c.*

Multiple. (2013). *R7rs.*

Nystrom, R. (2021). *Crafting interpreters.*

Palsberg, A. W. . J. (2002). *Modern compiler implementation in java.*

Wilson, P. R. (1996). *An introduction to scheme and its implementation.*