Documentación del Compilador: Lambdish

Diseño de Compiladores

Erick Francisco González Martínez

A01039859

Carlos Daniel Estrada Guerra

A01039919

3 de junio del 2020
Monterrey, Nuevo León México

Tabla de Contenidos

1) Descripción del Proyecto

a) Propósito

Lambdish es un lenguaje funcional pequeño que tiene como objetivo la simplicidad, el rendimiento y la programación estricta. Este lenguaje tiende a ser un enfoque más simple para los lenguajes modernos de programación funcional como Haskell, Racket, etc. Este nuevo enfoque tiene la intención de aplanar la curva de aprendizaje de la programación funcional al proporcionar una manera fácil de usar las características que hacen que la programación funcional sea útil e interesante permitiendo que la propia comunidad muestra apoyo a los nuevos usuarios que deseen contribuir con el código.

b) Objetivo

El objetivo principal de Lambdish se basa en el hecho de que el esquema del lenguaje permite a los usuarios encontrar su camino más fácilmente que cualquier otro lenguaje funcional, es decir, proporcionar un sistema de módulos útil y dividir su implementación en varias áreas. El objetivo principal se beneficia del deseo del usuario de implementar nuevas áreas de conocimiento en la nueva plataforma que proporciona este lenguaje. El área principal del lenguaje se basa en el paradigma mismo, lo que significa que el área de desarrollo es funcional, es decir, evaluaciones matemáticas.

c) Alcance del Proyecto

El sistema final sería un compilador de un lenguaje completamente funcional al igual que una máquina virtual que ejecute el programa compilado. El objetivo principal es proporcionar una manera más simple de representar los lenguajes funcionales de una manera más dinámica y atractiva y con esta mismo, mejorar la adquisición de aprendizaje para los nuevos usuarios de los lenguajes funcionales. El compilador tiene las funciones principales desde análisis léxico, sintáctico, semántico, generación de código intermedio y generación de código para obtener un archivo que contenga todo el código para que una máquina virtual lo ejecute. Por otra parte, la máquina virtual tendrá su propia administración de memoria y recibe como entrada el archivo generado por el compilador. El alcance del proyecto es entregar un compilador con casos de prueba que comprueben la robustez del mismo al igual que casos de prueba para la máquina virtual para que sea más fácil la verificación de su utilidad. Ambas partes serán capaces de administrar recursos básicos, una verificando cualquier tipo de error tratando de especificarlo y señalar cual es el funcionamiento correcto, por otra parte la administración de recursos, manejo de entradas o llamadas de las funciones y salida o resultado del archivo que contenga el código generado por el compilador.

d) Análisis de Requerimientos y Casos de Uso Generales

Lista de Requerimientos Funcionales

ID	Requerimiento	Descripción
RF0001	Declaración de Funciones	Dado el esquema funcional, la extensión de declaración de funciones conlleva determinar el contexto que se pueden utilizar. Se pueden declarar múltiples funciones, no se permite polimorfismo de funciones. La función permite recibir la cantidad deseada de parámetros como variables locales y siempre debe de tener un tipo de dato para el retorno de la función, En caso de que la función sea declarada como parámetro, solo debe de decir el tipo de parámetros que recibe, tipo de dato de retorno y un identificador.
RF0002	Declaración de Funciones Lambda	El sistema debe de permitir la declaración de funciones lambda con el carácter especial #, parámetros con identificador y el cuerpo de la función declarado entre los paréntesis.
RF0003	Declaración de Variables	El sistema debe permitir la declaración de variables como parámetro de tipo num, char, bool, list y funciones paramétricas.
RF0004	Llamada de Funciones	El sistema debe de permitir hacer llamadas a funciones previamente declaradas.
RF0005	Declaración de Listas	El sistema debe de permitir la declaración de listas de cualquier tipo de dato y de ser requerido con un nivel de anidación definido por el usuario.
RF0006	Declaración de Constantes	El sistema debe de permitir al usuario crear constantes dentro de las declaraciones y llamadas de las funciones. Permite la declaración de números, caracteres, booleanos, listas de cualquier tipo de dato con anidación definida por el usuario considerando.
RF0007	Inicialización de Listas Vacías	El sistema debe de permitir al usuario inicializar las listas vacías con el tipo de dato en el cuerpo de la función.
RF0008	Uso de las Funciones Aritméticas	El sistema debe de permitir al usuario realizar operaciones aritméticas con los operadores básicos: a) + b) - c) * d) / e) mod
RF0009	Uso de las	El sistema debe de permitir al usuario realizar operaciones

	Funciones Lógicas	lógicas con los siguientes nombres: a) And b) Or c) ! que es equivalente al not
RF0010	Uso de las Funciones Relacionales	El sistema debe de permitir al usuario realizar operaciones relacionales con los símbolos o nombres: a) < b) > c) Equal
RF0011	Uso de las Funciones del Sistema	El sistema debe de permitir al usuario realizar operaciones sobre los tipos de datos con los siguientes nombres: : a) Head: Recibe una lista y regresa el primer elemento de la lista. b) Tail: Recibe una lista y regresa la lista sin el primer elemento. c) Insert: Recibe un elemento y una lista y regresa la lista con el elemento al inicio. d) Append: Recibe 2 listas y regresa una sola lista con los elementos de las dos listas. e) Empty: Recibe una lista y determina si está vacía.
RF0012	Uso de las Funciones Condicionales	El sistema debe de permitir al usuario llamar a la función if que recibe un parámetro para evaluar la condición y dependiendo del valor se ejecuta el flujo deseado por el usuario. Dado que tipo de paradigma, en caso de tener un valor de retorno, regresa el valor del tipo de cualquier tipo de dato.
RF0013	Uso de los Comentarios	El sistema debe de permitir al usuario hacer comentarios sobre la misma línea en el código con el conjunto de símbolos “//”.

Casos de Usos Generales

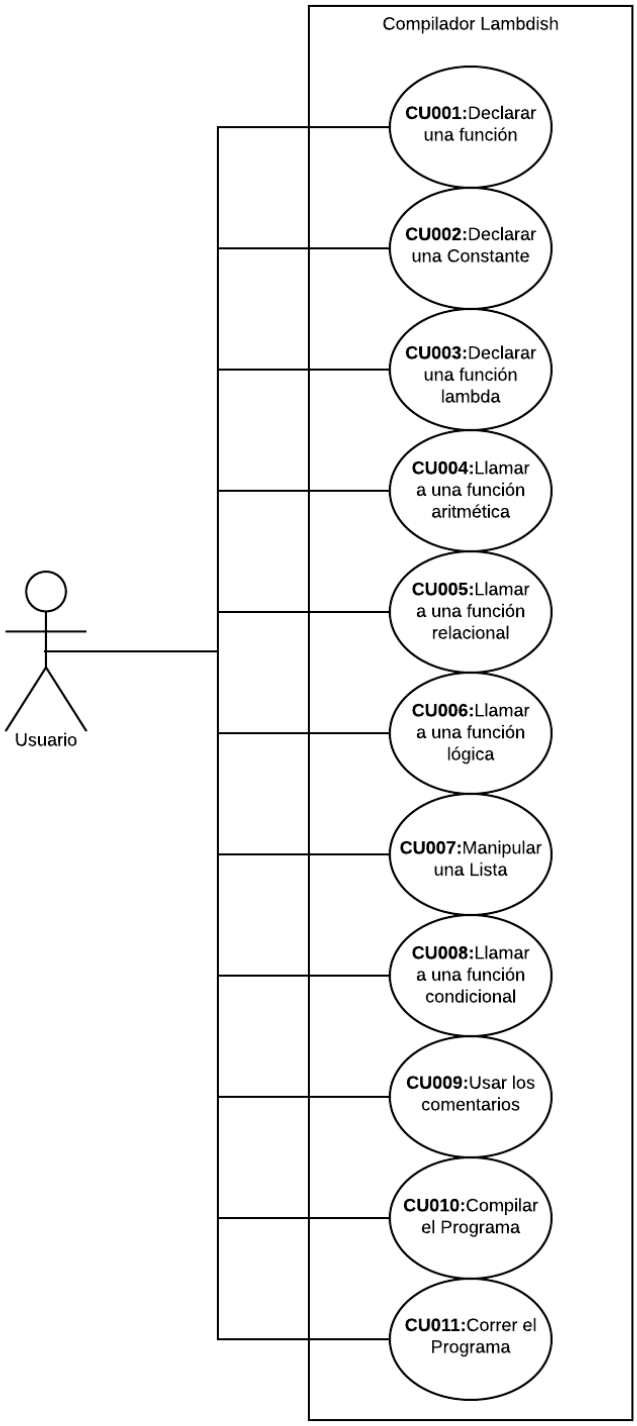


Figura 1 Diagrama de casos de uso

Mapeo de Casos de Usos

Nombre: Declarar una Función	**ID:** CU001
Descripción: El usuario declara una función con sus parámetros y el valor del tipo de retorno	
Criterios de aceptación: 1. La función permite recibir la cantidad deseada de parámetros. 2. Tener un tipo de dato para el retorno de la función. 3. Si es una función paramétrica solo debe de decir el tipo de parámetros que recibe, tipo de dato de retorno y un identificador.	

Nombre: Declarar una Constante	**ID:** CU002
Descripción: El usuario declara una constante con un tipo de dato	
Criterios de aceptación: 1. Se declara dentro de las declaraciones y llamadas de las funciones. 2. Permite la declaración de números, caracteres, booleanos, listas de cualquier tipo de dato con anidación definida por el usuario considerando.	

Nombre: Declarar una función lambda	**ID:** CU003
Descripción: El usuario declara una función lambda dado que no se permiten hacer llamadas a estas funciones.	
Criterios de aceptación: 1. Inicia con el carácter especial # 2. Los parámetros tienen identificador 3. El cuerpo de la función está declarado entre los paréntesis.	

Nombre: Llamar a una función aritmética	**ID:** CU004
Descripción: El usuario llama a una función aritmética básicas proveídas por el usuario.	
Criterios de aceptación: 1. El usuario realiza operaciones aritméticas con los operadores básicos.	

Nombre: Llamar a una función relacional	**ID:** CU005
Descripción: El usuario llama a una función relacional con los tipos de datos adecuados.	
Criterios de aceptación: 1. El usuario realiza operaciones relacionales con los símbolos o nombres provenientes por el sistema.	

Nombre: Llamar a una función lógica	**ID:** CU006
Descripción: El usuario llama a una función de orden lógico con los tipos de datos adecuados.	
Criterios de aceptación: 1. El usuario realizar operaciones lógicas con los nombres del sistema.	

Nombre: Manipular una Lista	**ID:** CU007
Descripción: El usuario decide manipular una lista con las funciones del sistema.	
Criterios de aceptación: 1. Realizar operaciones sobre los tipos de datos con los nombres de funciones del sistema. 2. Si es **Head**: Recibir una lista y regresa el primer elemento de la lista. 3. Si es **Tail**: Recibe una lista y regresa la lista sin el primer elemento. 4. Si es **Insert**: Recibe un elemento y una lista y regresa la lista con el elemento al inicio. 5. Si es **Append**: Recibe 2 listas y regresa una sola lista con los elementos de las dos listas. 6. Si es **Empty**: Recibe una lista y determina si está vacía.	

Nombre: Llamar a una función condicional	**ID:** CU008
Descripción: El usuario llama a la función condicional y provee 3 tipos de parámetros, la cual consiste de una función relacional o lógicas y otras dos estatutos.	
Criterios de aceptación: 1. Recibe un parámetro para evaluar la condición 2. Se ejecuta el flujo deseado por el usuario. 3. Regresa el valor del tipo de cualquier tipo de dato.	

Nombre: Usar los comentarios	**ID:** CU009
Descripción: El usuario utiliza los comentarios en el código para darle legibilidad el código.	
Criterios de aceptación: 1. Los comentarios están sobre la misma línea en el código 2. El conjunto de símbolos “//” son puestos en la misma línea al inicio.	

Nombre: Compilar el Programa	**ID:** CU010
Descripción: El usuario corre el comando **clamb** con el archivo **.lsh** que contiene el código del programa.	
Criterios de aceptación: 4. El sistema contiene análisis léxico. 5. El sistema contiene análisis sintáctico. 6. El sistema contiene semántica básica de variables. a. Directorio de Funciones b. Directorio de Variables 7. El sistema contiene semántica básica de expresiones a. Cubo Semántico 8. El sistema contiene la generación de código a. Estatutos Condicionales b. Funciones i. Aritméticas ii. Modulares iii. Parámetro iv. Lambda v. Pre instaladas 9. El sistema genera un archivo con el contenido del código	

Nombre: Correr el programa	**ID:** CU011
Descripción: El usuario corre el comando **rlamb** con el archivo **.obj** que contiene la información ejecutable del programa.	
Criterios de aceptación: 1. La máquina virtual contiene un mapa de memoria a. Segmento Númerico b. Segmento Booleano c. Segmento Char	

d. Segmento de Funciones
e. Segmento de Listas

2. La máquina virtual ejecuta
    a. Funciones
        i. Aritméticas
        ii. Modulares
        iii. Parámetro
        iv. Lambda
        v. Pre instaladas
    b. Estatutos condicionales

e) Descripción de los principales Test Case

Bajo el folder de "***Examples***", se desarrollaron múltiples casos de prueba para verificar la extensión de nuestro compilador. Dentro de las siguientes pruebas se puede determinar las siguientes características en cada uno de los archivos.

1. **Test_Algoritmos.lsh:** Se comprueba
    a. Funciones del sistema
        i. Head
        ii. Insert
        iii. Tail
        iv. Equal
    b. Funciones
        i. Aritméticas
        ii. Lógicas
        iii. Relacionales
    c. Funciones programadas por el usuario utilizando sus variables de contexto
    d. Funciones por parámetro
    e. Uso de listas
        i. Constantes
        ii. Vacías
    f. Recursividad de Funciones
2. **Test_Lambdas.lsh:** Se comprueban las siguientes características
    a. Funciones lambdas
        i. Simples
        ii. Anidadas
        iii. Parámetro
3. **Test_Matrix.lsh**
    a. Uso de listas
        i. Anidación Múltiple (matrices)
    b. Recursividad

c. Construcción de Listas
d. Modularidad de funciones
e. Comentarios

4. **Test_Error.lsh**
    a. División por cero
    b. Inicialización de Listas
        i. Vacías
        ii. Contantes

f) Bitácora Semana

Avance #1: 13 de abril del 2020

¿Funciona?

Sí.

¿Faltó una parte?

Se tuvieron que hacer algunos cambios en la idea inicial de la gramática para poder minimizar los errores en la gramática y hacer el lenguaje lo más robusto posible. Primero que todo, se tomó la decisión de hacer que todo en el lenguaje sea una función, esto implica que las operaciones aritméticas básicas también serían funciones, así también como las operaciones relacionales y lógicas. Además, para las funciones anónimas se cambió el carácter reservado \ por el carácter #.

¿Corrieron suficientes test-cases?

Se ejecutaron 4 archivos de prueba dónde pudimos identificar que se recorrieran todos los estados generados por la gramática.

¿Tienen todavía ambigüedades en la gramática?

No existen conflictos de shift-reduce ni reduce-reduce en la gramática de acuerdo al generador de analizador de sintaxis escogido.

¿Pueden marcar errores particulares o solo uno genérico (syntax error)?,

El parser generado identifica el siguiente token que se espera recibir. Dado que la resolución de conflictos se resuelve de la misma manera que C. Utiliza el shifting y parsing de la producción más larga posible (maximal-munch).

Avance #2: 20 de abril del 2020

Para esta entrega se crearon dos nuevos packages dir y types. En el package de dir se definieron las estructuras para manejar el directorio de funciones y los directorios de variables. En types se definió una estructura para representar los tipos de datos disponibles. En dir se creó la estructura FuncDirectory que sirve para guardar las definiciones de las funciones. FuncDirectory es una tabla de estructuras FuncEntry.

FuncEntry representa la definición de una función, por lo tanto, contiene el id de la función, el tipo de retorno, los parámetros y VarDirectory. Para poder acceder a un FuncEntry de manera constante en la tabla FuncDirectory, se ejecutó una estrategia para representar FuncEntry como un string único y utilizar este string como llave. VarDirectory contiene las variables declaradas dentro del alcance de la función.

VarEntry representa la definición de una variable, por lo tanto, contiene el id de la variable y el tipo de la variable. Para poder acceder a una VarEntry de manera constante en la tabla de VarDirectory, se ejecutó una estrategia para representar VarEntry como un string único y utilizar este string como llave.

A continuación se muestra la estructura actual de los archivos del programa.

```
.
├── ast
├── dir
├── gocc
│   ├── errors
│   ├── lexer
│   ├── parser
│   ├── token
│   └── util
├── grammar
│   └── tests
├── sem
└── types
```

En cuanto a la implementación de las acciones semánticas en la gramática. La herramienta Gocc, no permite estados globales mientras se ejeDEcuta el análisis sintáctico, por lo que no se puede tener una tabla global o algo por el estilo para ir agregando funciones. Por lo tanto, primero se tiene que construir el árbol abstracto de sintaxis (AST), y después se recorre para construir todas las estructuras. Dado esto, no hemos alcanzado a implementar la construcción del AST por lo que aún no implementamos las estructuras de FuncDirectory y VarDirectory. Sin embargo, las pruebas unitarias para ambas estructuras nos indican que ambas tienen el funcionamiento deseado.

En cuanto a la metodología de trabajo y versión de controles se hacen a través de un solo repositorio y bajo un solo autor debido a que se usa una herramienta de trabajo en el cual se puede trabajar simultáneamente en una computadora del host, en este caso el autor principal del Github. VsCode Live Share es recomendable para trabajo colaborativo pero no trackea los cambios hechos por un autor en específico.

Cambios

1. Cambio de la gramática para definir las funciones antes de hacer las llamadas
    a. Se tomó la decisión de que en el cuerpo del programa solo pueden haber declaraciones de funciones y solamente al final puede haber una sola llamada a una función.
2. Cambio de estructura de proyecto
    a. Se movió
        i. La gramática dentro de /gocc.
    b. Se agregó
        i. Dir
            1. Se agregó el directorio de funciones y los directorios de variables.
            2. Tests (Ok)
        ii. Ast
            1. Estructura del árbol abstracto de sintaxis.(Por definir)
        iii. Sem
            1. Estructura del análisis semántico.(Por definir)
3. Metodología de trabajo

Avance #3: 27 de abril del 2020

Para esta semana se desarrollaron 4 principales funcionalidades de la semántica. La primera funcionalidad verifica que ninguna función no es declarada dos veces ni que el nombre de los parámetros se repita dentro de la llamada de la función. La segunda funcionalidad verifica el alcance de las variables y funciones que se crean en sus respectivos directorios. La tercera funcionalidad verifica la cohesión de los tipos. La cuarta funcionalidad es la implementación de el cubo semántico.

Para poder incluir la primera funcionalidad se crearon 3 archivos bajo el paquete sem:

- Funcheck: Se encarga de registrar las funciones y sus respectivas variables para determinar el alcance.
- Funcutil: Incluye funcionalidades de utilidad miscelánea, en este caso para verificar nombre de funciones con palabras reservadas como lo son los operadores lógicos.
- Funccheck_test: Prueba la funcionalidad implementada para el directorio de funciones.

Para poder incluir la segunda funcionalidad se agregaron 3 archivos bajo el paquete sem:

- Scopecheck: Se encarga de recorrer el árbol AST para identificar el scope de las funciones. Se crea un stack de entradas de funciones para determinar cuál es el alcance de las variables. Debido a que también se aceptan llamadas de función como parámetros, utilizar un stack para resolver la recursividad de estas llamadas es conveniente para la optimización de número de recorridos en el árbol.

- Scopecheckutil: Incluye funcionalidades de utilidad miscelánea, como verificar si un identificador ya existe en el stack implementado o si ya existe la función declarada en el directorio de funciones.
- Scopecheck_test: Prueba la funcionalidad implementada para definir el alcance correcto de las variables y funcionalidades.

Para poder incluir la tercera funcionalidad se agregaron 2 archivos bajo el paquete sem:
- Typecheck: Se utiliza el stack y el cubo semántico de funciones reservadas o implementadas por el sistema previamente.Verifica la cohesión de tipos
- Typeutil: Incluye funcionalidades de utilidad miscelánea, verifica las funciones reservadas y valida que los tipos de datos que se le llamen a este tipo de funciones sean los correctos. Funciones como:
    - If: La longitud de los argumentos no puede ser mayor a 3, se va a utilizar para el flujo de decisiones.
    - Append: La longitud de la lista de argumentos debe de ser 2.
    - Empty:La longitud de la lista de argumentos debe de ser 1 y debe de ser tipo lista.
    - Head: La longitud de la lista de argumentos debe de ser 1 y debe de ser tipo lista.
    - Tail: La longitud de la lista de argumentos debe de ser 1 y debe de ser tipo lista.
    - Insert: La longitud de la lista de argumentos debe de ser 2, el primero debe de ser un tipo básico y el segundo debe de ser tipo lista.

Para poder incluir la cuarta funcionalidad se agregó 1 archivo bajo el paquete sem:
- Semanticcube: Dentro de este tipo se define la estructura del cubo semántico, el cual contiene el id de la función generada y el tipo de retorno de esa función. También identifica los errores específicos de cohesión de tipos al momento de que se ejecuta el Typeutil. Debido a que existen una cantidad variable de funciones a generar como parámetros es necesario hacer una verificación de tipos en lugar de incluirlas en el cubo semántico cómo válidas.

Cambios

1. Se cambió la gramática inicial para clasificar las llamadas lambda a la declaración de llamadas lambda, haciendo más fácil la detección de errores al momento de ser registrada en el directorio de funciones anónimas.
2. En la gramática se cambió la regla function call para que se puedan hacer llamadas a funciones no solo partiendo de ids sino que también de otros tipos de datos.
3. Como cambio más importante, se modificó tanto la gramática como nuestras estructuras internas para poder aceptar funciones como tipos de datos. Esto para poder pasar funciones como argumentos a otras funciones y también para poder regresar funciones, por ejemplo las funciones lambda.

Avance #4: 4 de mayo del 2020

Para esta semana se modificó el código intermedio y se generaron 2 nuevos paquetes de golang llamados “ic” que representa la generación del código intermedio y “mem” que maneja la memoria virtual. Para esto se crearon nuevas estructuras principales que permiten la generación de cuadruplos al momento de encontrarse. También se definieron controles de flujo sintáctico para el estatuto “if”. A continuación, se explicará a detalle los cambios importantes en cada uno de los archivos generados.

Para la principal funcionalidad se hicieron 2 archivos ic y ic_test. El archivo ic.go genera 2 partes importantes. Primero, inicializa la llamada a todas las entradas del directorio de variables con un manejador de memoria.

Manejador de Memoria: La memoria está dividida en 5 grandes partes como lo son globales, locales, temporales, constantes y fuera del contexto junto con un contador ubicado al inicio de cada partición. Se les dedicó 1,000 registros a cada tipo de dato en cada una de las particiones. Los tipos de dato son num, char, bool, functions, lists. Debido a que es un lenguaje funcional se aceptan funciones declaradas como parámetros por lo que se le tiene que asignar un espacio.

Generación de Código Intermedio: Las 2 partes importantes son la inicialización de direcciones de las variables en las funciones y la generación de código del programa una vez inicializado el atributo de dirección en el directorio de variables. Se estableció la estructura cuádruplos que contiene la operación, las direcciones de los argumentos y el resultado.

Para la generación de direcciones del programa a ejecutar se creó la primera parte de inicialización. Se recorre el árbol AST para poder determinar si existen valores a los cuales se les pueden determinar direcciones como variables locales o constantes.

Por otro lado, la generación de código es más amplia. En esta parte, con los valores ya inicializados se genera una estructura que contiene apuntadores a la memoria virtual, el cubo semántico, la estructura “Generator” y el directorio de funciones. La estructura “Generator” contiene 2 stacks, contadores generales: uno para parámetros al momento de generar el código intermedio de declaración de funciones y el otro es para el contador general de la memoria.

1. Generación de operaciones aritméticas: se generó el código para las operaciones aritméticas básicas y operaciones relacionales y lógicas
2. Generación de estatutos no lineales: Se generó el código para los estatutos if-else. Dado que el lenguaje no tiene ciclos, este fue el único estatuto que se tuvo que generar para esta categoría

3. Generación de código de funciones: Además se generó el código para llamar funciones y para declarar funciones. Parte del reto de esto estaba en generar el código para las funciones lambda. Al final se logró generar el código correspondiente de manera adecuada.

Recorriendo el árbol, se generan los cuàdruplos con sus respectivas direcciones al principio del programa. La principal es main que contiene todo el archivo de ".lsh." Para generar funciones se utiliza el stack y así recursivamente resuelve los estatutos de las funciones declaradas ya sean declaraciones normales o funciones lambda. Para el caso de las funciones reservadas se generan con el mismo nombre ya sean operaciones aritméticas, relacionales, operadores lógicos, estatuto de control de flujo y funciones incorporadas en el lenguaje como empty, head,tail, append o insert.

Cambios

1. Se reestructuró la gramática para que aceptara el símbolo ! como operador lógico.
2. Se agregó el atributo de address a la tabla de variables para poder llevar un rastreo de la dirección de cada variable
3. Se agregó el atributo de location a la tabla de funciones para poder ubicar el comienzo de todas las variables globales y las lambda

Avance #5: 11 de mayo del 2020

Para esta semana se modificó el código intermedio "ic" para incluir la validación de funciones y además algunas correcciones que surgieron al momento de realizar algunas pruebas. Se creó otro nuevo paquete llamado "vm" para empezar a manejar la memoria de la máquina virtual.

1. Generación de Código Intermedio Para Funciones: Se generó el código de implementación de funciones al momento de llamarlas. En el caso de las llamadas a funciones previamente declaradas y la llamada de funciones lambda. La estructura de generación de contexto nos permite verificar a través de un generador todas aquellas direcciones pendientes de los valores de retorno y de los valores de las direcciones virtuales de las llamadas de las funciones. Se corrigió el paquete de manejo de memoria virtual para los cuádruplos para casos especiales en las variables extraordinarias como los son las funciones como parámetros y las listas declaradas. Se delegó esta tarea al momento de ejecución para que la validación semántica dinámica, es decir que la máquina virtual tendrá su propia estructura para manejar las listas.

2. Máquina Virtual: Se crearon 2 estructuras que servirán como la representación de la memoria de la máquina virtual por lo que se implementó un tipo de segmentación por contexto y dentro del contexto los diferentes tipos de datos. Se creó un comando dentro del paquete "cmd" que registra a la llamada "clamb"

que toma como argumento el nombre del archivo para compilarlo y generar un .obj. Este archivo contiene todos los cuádruplos generados al momento de ser creado.

3. Integración: Para la revisión se añadió una generación de pruebas para mostrar el avance de la junta que se tuvo el 12 de mayo del 2020 a las 09:40.

Cambios

1. Ahora se acepta las listas anidadas y se añaden los operandos Lst,GeLst, PaLst para atender las listas y poder guardarlas en un ambiente extraordinario.
2. Se corrigió la gramática para que acepte números negativos, que es algo que no habíamos considerado dentro del EBNF.
3. Se corrigió la gramática para que acepte los valores “true” y “false” como valores constantes.
4. Se determinó la prioridad del ERA dentro de las llamadas de las funciones, ya que no es necesario saber la cantidad de variables temporales que se necesitan porque se le añadió un límite para todas las variables temporales en la máquina virtual.

Avance #6: 19 de mayo del 2020

Para esta semana se modificó el código intermedio “vm” la cual implementa todas las operaciones aritméticas, lógicas, relacionales y funciones que se crean por parte del sistema. En este avance el proceso de desarrollo queda como terminado y pasa a la fase de pruebas de integración la cual incluye desde el análisis léxico hasta la generación de código.

Para el paquete “vm” los cambios más importantes fueron el manejo de listas y las operaciones sobre estas listas.

1. Máquina Virtual:
    a. Para el manejo de listas se creó una nueva estructura que maneja 5 tipos de listas con la cual se puede representar en la memoria para guardar los valores.
    b. También un conflicto que se tenía con la gramática se pudo resolver a partir de este manejador de listas, ahora se puede aceptar las listas vacías y en la declaración de listas constantes de char se puede manejar como un conjunto a partir de el uso de comillas dobles.
    c. Dentro de las operaciones se pudieron implementar las funciones como head, tail, insert, append y empty sobre las listas.
    d. Para el manejo de memoria de las variables en la máquina virtual se pudo implementar lo que son segmentos de memoria con los cuáles se pudieron acceder a los 5 segmentos antes mencionados de todos los tipos de datos.

e. Para el manejo de llamada a las funciones se implementó la estructura de activation records, esta estructura en un stack nos puede indicar el principio de la llamada de la función.
f. Los estatutos condicionales, así como las llamadas a funciones, con todo lo que esto involucra, también se pudieron implementar en la máquina virtual para este avance.

Cambios

1. Se decidió cambiar el funcionamiento del lenguaje para que dentro de una declaración lambda, solo se pueda hacer uso de las variables estrictamente locales a esa declaración, esto para mantener el lenguaje simple, que es su propósito principal.
2. Se agregó una nueva forma de declarar una lista de caracteres utilizando la sintaxis de C de comillas dobles para representar un string. Esto se decidió implementar para que sea más fácil para el usuario el trabajar con listas de caracteres ya que es una estructura muy común en la programación.

   Dentro del paquete “vm” se agregó un folder de pruebas dónde se tradujeron ciertos programas de nuestra clase de lenguajes de programación en el cual implementaremos la solución a las actividades y todos los tests que se aplicaron pudieron mostrar la respuesta correcta.

Avance #7: 26 de mayo del 2020

Para esta entrega se terminó la versión final del compilador, la cual se encuentra en una fase estable que puede manejar todos los casos de prueba posible. Después de terminar la versión final se construyeron diferentes casos de prueba para probar casos extremos o inusuales. Con estos casos se descubrieron algunos errores en la máquina virtual y en la generación de código intermedio. Se corrigieron los errores encontrados hasta que no se pudieron encontrar más errores.

Pruebas de Integración: Se agregaron los siguientes casos de prueba
1. Pruebas con listas y matrices
2. Pruebas con recursión
3. Pruebas con funciones lambda

Todas las pruebas ejecutadas pasaron exitosamente.
Cambios
1. Se agregaron pruebas de integración de los cambios más recientes en la gramática.
2. Se agregaron comentarios de línea de tipo C, con los caracteres especiales “//”. Estas líneas de código son ignoradas al momento de compilación.

Avance #8: 31 de mayo del 2020

Para este avance se modificó el paquete “**vm**” debido a que en la fase de pruebas se identificó un caso de prueba que no cumplía con los requerimientos que habíamos establecido. Se diseñó un caso de prueba donde se identificó que no se puede declarar un if dentro del argumento de la llamada a una función. Esto debería de funcionar porque en el lenguaje el if debe de funcionar como una función, por lo que debería de poder ser llamado desde cualquier parte donde puede haber una expresión, y esto incluye en los argumentos de las funciones.

Además, se incluyó la documentación parcial del proyecto en otro anexo. Este anexo contiene el avance de la documentación del proyecto que se tiene.

El avance real del compilador es **terminado** y **funcional**. SIn embargo, durante los siguientes días se seguirá probando haciendo nuevas pruebas del compilador.
Cambios

- Se modificó el paquete "ic" para que la generación de código no genere un cuádruplo RET por default, sino que genere un nuevo cuádruplo ASSIGN
- Se modificó el paquete "quad" para agregar un nuevo cuádruplo llamado ASSIGN. Este cuadruplo asigna el primer operando al operando de resultado.
- Se modificó el paquete "vm" para agregar el soporte del nuevo cuádruplo ASSIGN

Con estos cambios en pie, ahora se puede utilizar la función if en cualquier parte del código.

g) Reflexiones

Erick González

Gracias a este proyecto se aprendieron muchas cosas, adquirí nuevos conocimientos y durante el desarrollo del compilador. Mucho de los conflictos con los cuáles nos enfrentamos es cómo solucionar las principales características del lenguaje porque existen diferentes metodologías y patrones de diseño para resolver todos los problemas que tuvimos pero teníamos que elegir una de las soluciones con lo que veíamos en clase o investigar por nuestra propia cuenta en foros de desarrollo. A pesar de que haya sido mucha ayuda leer investigaciones y el mismo libro del dragón, mucho del conocimiento fue a través de la discusión y probar nuevas soluciones con mi compañero. Unas de mis principales áreas de conocimiento fueron el autoaprendizaje y la organización. Estos son los más importantes para la vida profesional, ya que siempre se necesita la voluntad y la motivación para seguir aprendiendo por su cuenta y mucho más en esta carrera. Un proyecto de esta magnitud no solo se cumplió con motivación sino que la comunicación efectiva fue otro factor crucial para cumplir con el propósito inicial de este proyecto.

Carlos Estrada

Dicen que crear un compilador representa el pináculo de la esencia del estudio de las ciencias computacionales. Después de desarrollar este proyecto, creo que esto es cierto. Este proyecto tuvo la capacidad de retar cada una de las áreas de las ciencias computacionales que he estudiado hasta el momento. A lo largo del desarrollo de este proyecto puede aprender, no solo sobre la teoría de los lenguajes y los compiladores, sino también sobre la administración de un proyecto con un nivel de reto intelectual bastante elevado. Desde que comencé mi carrera de tecnologías computacionales, la magia del compilador siempre había sido un misterio para mí, una caja negra incomprensible. Siempre había tenido el sueño de comprender los detalles intrínsecos detrás de las cortinas. El compilador de Lambdish me ha ayudado a alcanzar este sueño.

2) Descripción del Lenguaje

a) Nombre

Lambdish

b) Descripción Genérica de las características principales

La siguiente lista describe las principales características del lenguaje.

i) El lenguaje es un lenguaje tipado estáticamente por lo que los tipos de datos de las variables tienen que ser explícitamente declarados
ii) Dentro del lenguaje no hay variables globales ya que el paradigma de programación funcional no permite el estado de las variables globales.
iii) Los parámetros en funciones solo existen en el contexto de la función.
iv) Siguiendo el paradigma de programación funcional, una función solo puede contener una expresión, que es el valor que devuelve.
v) Las operaciones aritméticas solo se pueden realizar con el tipo de datos number (num). Las operaciones lógicas sólo se pueden realizar con el tipo de datos booleano (bool).
vi) No hay acceso constante a los elementos de una lista, solo se puede acceder al primer elemento y al último.
vii) Para definir el punto de inicio del programa, no hay una función principal para iniciar el programa. Sino que siempre se ejecuta la única llamada a una función que está en el contexto global, y está llamada siempre se encuentra al final del programa.
viii) Cuando se ejecuta un programa, siempre se imprime el valor de retorno de la única llama global a una función que está al final del programa (punto de entrada)

ix) Además de las funciones con nombre en el contexto global, existen las funciones lambda que se declaran en cualquier parte del lenguaje donde cabe una expresión.

x) Las funciones lambda solo pueden accesar sus propios parámetros y no los de las funciones de donde se declaran.

xi) Como todo lenguaje funcional, la función es un tipo de dato que puede ser pasado como parámetro o devuelto en una función

xii) El tipo de dato de función depende de los tipos de los parámetros, su orden, y el valor de retorno.

c) Listado de errores que pueden suceder

Léxico

- Análsis
    - Token
    - Valor esperado
    - Posición
        - Offset
        - Línea
        - Columna

Sintáctico

- Análisis
    - Token
    - Valor esperado
    - Posición
        - Offset
        - Línea
        - Columna
- AST
    - Error en la función de la creación del AST, es decir de los argumentos en las funciones para la generación del árbol

Semántico

- Funciones
    - Redeclaración de una función
    - Redeclaración de un parámetro dentro de una función
- Scope
    - Uso de ids no declarados en el scope dado
    - Llamada a funciones no declaradas
- Tipos
    - Retorno de una función no coincide con el tipo de retorno de la función
    - Tipo de dato del argumento no coincide con el parámetro de la función

        - Discordancia de tipos de datos en el cubo semántico
    - Funciones Built in
        - Error en tipos de datos en llamadas a funciones de listas (head, tail, insert, append, empty)
        - Error en el uso de la función if
            - La función if debe tomar exactamente 3 parámetros. El primero debe ser de tipo booleano y el segundo debe de ser del tipo que se espere que se regrese de la función if

Runtime

- Aritmético
    - División entre 0
    - Modulo de 0
- Listas
    - Head en lista vacía
    - Tail en lista vacía
- Memoria
    - Acceso a una dirección de memoria inválido (sólo puede ocurrir si el usuario modifica directamente el código objeto)

3) Descripción del Compilador

a) Equipo de cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto

Equipo de Cómputo:

1. macOS / MacBook Pro 2.7 GHz Dual-Core Intel Core i5 8 GB 1867 MHz DDR3
2. Custom Built
    a. Procesador: Intel Core i7-7700
    b. RAM: 16 GB
    c. OS: Windows 10

Lenguaje: Go es un lenguaje de programación de código abierto que facilita la creación de software simple, confiable y eficiente.

Utilerías:

1. **GOCC**: Generación de analizador sintáctico y léxico a partir de una gramática diseñada para EBNF. Dado que era un lenguaje dirigido por la sintaxis, se podía llamar funciones con los argumentos en la gramática y así fue posible generar un AST.
2. **ErrUtil**: Paquete de librerías genéricas. Debido a que go, no tiene estructuras genéricas, esta librería ofrece un módulo de errores que permiten darle formato dependiendo de la gravedad del problema aumentando la legibilidad de código y facilitando la depuración de código al igual que el mantenimiento del mismo.

3. El resto de las librerías que se utilizaron son las que vienen por default con el compilador de Go que se pueden encontrar en la siguiente página: https://golang.org/pkg/

b) Descripción del Análisis Léxico

A continuación se describen las expresiones regulares para los tokens que se requieren para el lenguaje:

Tokens y sus expresiones regulares

1. `func → func`
2. `id → [a-z] | [A-Z] *([0-9] | [a-z] | [A-Z])`
3. `dcolon → ::`
4. `arrow → =>`
5. `lparen → (`
6. `rparen → )`
7. `comma → ,`
8. `lbracket → [`
9. `rbracket →  ]`
10. `pound → #`
11. `num → num`
12. `bool → bool`
13. `char → char`
14. `operations → + | - | * | / | %;`
15. `relop → < | > | !;`
16. `boolean → true | false`
17. `number → ?(-) ([0-9] *[0-9]) | ([0-9] *[0-9] . [0-9] *[0-9])`
18. `charac → ' *([a-z] | [A-Z] | [0-9] | ' ') '`
19. `string → " *([0-9] | [a-z] | [A-Z] | ' ') "`

c) Descripción del Análisis de Sintaxis

A continuación se describe la gramática libre de contexto que se utiliza para el análisis de sintaxis del lenguaje. Para fines expositivos, se adoptaron las siguientes convenciones.

- Todo no-terminal en la gramática comienza con letra mayúscula
- Todo terminal comienza con letra minúscula y está escrito en negritas
- El vacío se representa por el carácter ε.
- Program es el no-terminal raíz del programa

Reglas de sintaxis

Program → Functions Statement

Functions → Function Functions | Function | ε

Function → ***func id dcolon*** Params ***arrow*** Type ***lparen*** Statement ***rparen***

Params → Type ***id comma*** Params | Type ***id*** | ε

Type → BasicType | ***lparen*** FuncTypes ***arrow*** Type ***rparen*** | ***lbracket*** Type ***rbracket***

BasicType → ***num*** | ***bool*** | ***char***

FuncTypes → Type ***comma*** FuncTypes | Type | ε

Statement → ***id*** | Constant | Lambda | FunctionCall

FunctionCall → Statement ***lparen*** Args ***rparen***
| ***operations lparen*** Args ***rparen*** | ***relop lparen*** Args ***rparen***

Lambda → ***lparen pound*** Params ***arrow*** Type ***lparen*** Statement ***rparen rparen***

Args → Statement ***comma*** Args | Statement | ε

Constant → ***boolean*** | ***number*** | ***charac*** | ***string*** | ***lbracket*** ConstantArgs ***rbracket*** | ***lbracket*** Type ***rbracket***

ConstantArgs → Statement ***comma*** Args | Statement

d) Descripción de Generación de Código Intermedio y Análisis Semántico

Para realizar este compilador, se ha tomado un enfoque diferente al visto en clase. En vez de realizar acciones de análisis semántico y generación de código conforme se recorre la gramática, primero se construye un árbol abstracto de sintaxis (AST por sus siglas en inglés). Este AST entonces se recorre las veces que sea necesario para realizar todo tipo de análisis semántico y generación de código. En la sección de los diagramas de sintaxis mostramos los puntos neurálgicos para crear el AST con los diferentes nodos.

Una vez que tenemos el árbol construido, se recorre un total de 5 veces para hacer los chequeos semánticos y construcción de cuádruplos. A continuación se listan los recorridos que se realizan.

1. Construcción de la tabla de funciones y tablas de variables
2. Chequeo de uso de variables en sus respectivos scopes
3. Chequeo del uso correcto de tipos
4. Asignación de direcciones de la memoria virtual a las variables
5. Generación de cuádruplos correspondientes

El hecho de tener un AST nos da la flexibilidad de recorrer la sintaxis las veces que sea necesario con el propósito de realizar los chequeos semánticos y de generación de código de una forma modular.

A continuación se explican los cuádruplos que se generan por el código. Después de esto, se hace un análisis más profundo en la construcción de AST.

Código de operación y direcciones virtuales asociadas a los elementos del código.

A continuación se enlista el nombre de las operaciones del código intermedio con su descripción respectiva y el tipo de parámetros que recibe. Los cuádruplos siguen la siguiente estructura:

```
<Nombre> <lop> <rop> <resultado>
```

Para saber más información de las direcciones reales verificar la *Figura 4 Mapa de la Memoria* .

1. **Add**: Operación aritmética que sirve para sumar 2 números.
   Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal NUM.
2. **Sub**: Operación aritmética que sirve para restar 2 números.
   Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal NUM.
3. **Mult**: Operación aritmética que sirve para multiplicación 2 números.
   Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal NUM.
4. **Div**: Operación aritmética que sirve para división 2 números.
   Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal NUM.
5. **Mod**: Operación aritmética que sirve para obtener el módulo 2 números.
   Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal NUM.
6. **Lt**: Operación relacional que sirve para determinar si el parámetro 1 es menor que el 2.

Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal BOOL.

7. **Gt**: Operación relacional que sirve para determinar si el parámetro 1 es mayor que el 2.

   Tiene operandos de tipo NUM en el rango de local, temporal, constante y guarda el resultado en un temporal BOOL.

8. **Equal**: Operación relacional que sirve para determinar si el parámetro 1 es igual que el 2.

   Tiene operandos de tipo NUM, CHAR, BOOL en el rango de local, temporal, constante y guarda el resultado en un temporal BOOL.

9. **And**: Operación lógica que sirve para la operación **and** en ambos parámetros.

   Tiene operandos de tipo BOOL en el rango de local, temporal, constante y guarda el resultado en un temporal BOOL.

10. **Or**: Operación lógica que sirve para la operación **or** en ambos parámetros.

    Tiene operandos de tipo BOOL en el rango de local, temporal, constante y guarda el resultado en un temporal BOOL.

11. **Not**: Operación lógica que sirve para la operación **not** en ambos parámetros.

    Tiene operandos de tipo BOOL en el rango de local, temporal, constante y guarda el resultado en un temporal BOOL.

12. **GotoT**: Operación que indica al IP dónde seguir el flujo de ejecución en caso de ser verdadero el operando, se puede dejar pendiente para verificar con el stack a dónde ir.

    Tiene operandos de tipo BOOL en el rango de local, temporal, constante y guarda la posición del IP a dónde debe de ir.

13. **GotoF**: Operación que indica al IP dónde seguir el flujo de ejecución en caso de ser falso el operando, se puede dejar pendiente para verificar con el stack a dónde ir.

    Tiene operando de tipo BOOL en el rango de local, temporal, constante y guarda la posición del IP a dónde debe de ir.

14. **Goto**: Operación que indica al IP dónde seguir el flujo de ejecución, se puede dejar pendiente para verificar con el stack a dónde ir.

    Tiene la posición del IP a dónde debe de ir.

15. **Ret**: Operación obtiene el valor de retorno y resetea las variables locales, copia a la memoria los valores temporales y obtiene la dirección de dónde guardar el valor de retorno.

    Tiene operando de tipo NUM, CHAR, BOOL, FUNC, LISTS en el rango de local, temporal, constante y guarda la posición del IP a dónde debe de ir.

16. **Era**: Crea el récord de activación para la función.

    No tiene ningún operando significativo

17. **Param**: Obtiene los los parámetros y asigna al récord de activación el parámetro que recibió.

    Tiene operando de tipo NUM, CHAR, BOOL, FUNC, LISTS en el rango de local, temporal, constante y guarda la posición del IP a dónde debe de ir.

18. **Call**: Obtiene el salto que tiene que hacer el IP para localizar la función a la que se está llamando.

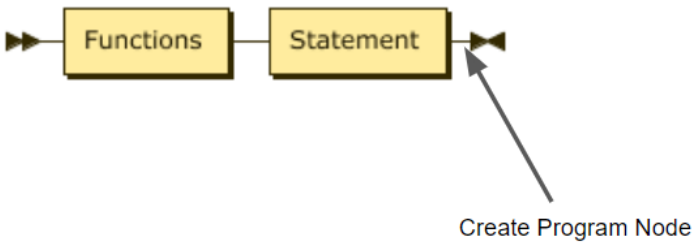
    Tiene un operando que es la posición de dónde se encuentra la función.

19. **Emp**: Operación del sistema que determina si una lista está vacía.
    Tiene un operando de tipo BOOL en el rango de temporal.
20. **Head**: Operación que regresa el primer elemento de una lista.
    Tiene operandos de tipo LIST en el rango de local, temporal, constante y otro de tipo NUM, CHAR, BOOL, LISTS en el rango de temporal para regresar el elemento .
21. **Tail**: Operación que regresa una lista sin el primer elemento
    Tiene operandos de tipo LIST en el rango de local, temporal, constante y otro de tipo NUM, CHAR, BOOL, LISTS en el rango de temporal para regresar el elemento .
22. **Ins**: Operación que inserta un elemento a una lista sin el primer elemento
    Tiene operandos de tipo NUM, CHAR, BOOL, LISTS en el rango de temporal para regresar el elemento y otro de tipo LIST en el rango de local, temporal, constante y la dirección del resultado de la lista.
23. **App**: Operación que une dos listas del mismo tipo.
    Tiene operandos tipo LIST en el rango de local, temporal, constante y la dirección del resultado de la lista.
24. **Lst**: Operación del sistema que genera un nuevo manejador de listas.
    Tiene un operando que determina el tipo de dato y otro para determinar la longitud de la cantidad de elementos que se generan con el PaLst.
25. **GeLst**: Operación del sistema que obtiene la lista terminada con todos los elementos de la misma.
    Tiene un operando de tipo LIST en el rango de local y temporal.
26. **PaLst**: Operación del sistema que obtiene el elemento de la lista y adjunta a la lista que se está creando.
    Tiene operandos de tipo NUM, CHAR, BOOL, LIST en el rango de local, temporal, constante y otro para identificar el número de elemento.
27. **Print**: Operación del sistema que imprime el resultado final de la llamada de la función.
    Tiene operandos de tipo NUM, CHAR, BOOL, LIST en el rango de local, temporal y constante.
28. **Assign**: Operación para asignar un valor a una dirección en caso de un estatuto condicional.
    Tiene operandos de tipo NUM, CHAR, BOOL, LIST en el rango de local, temporal, constante y otro para identificar la dirección del elemento local o temporal al cual se le va a asignar.
29. **Invalid**: Genera un error de cualquier tipo por lo cual se debe de abortar la ejecución del programa y no genera ninguna dirección.

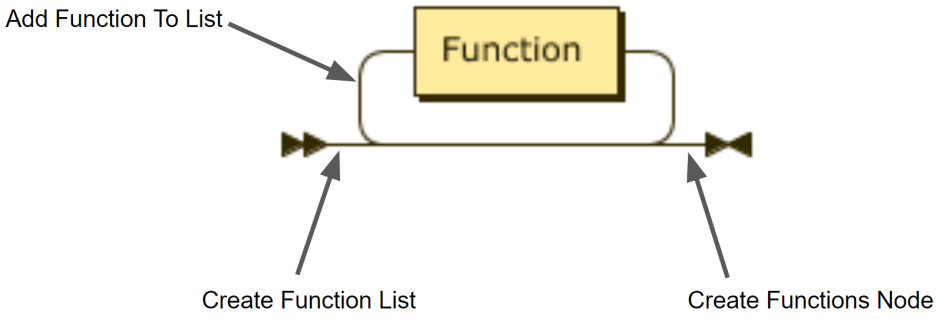
Diagramas de Sintaxis con acciones semánticas

A continuación se presentan los diagramas de sintaxis con las acciones semánticas que se realizaron. Cabe recalcar que las acciones que se presentan en el diagrama son solamente para crear el árbol abstracto de sintaxis, y después se usa este árbol para hacer todo tipo de análisis semántico.

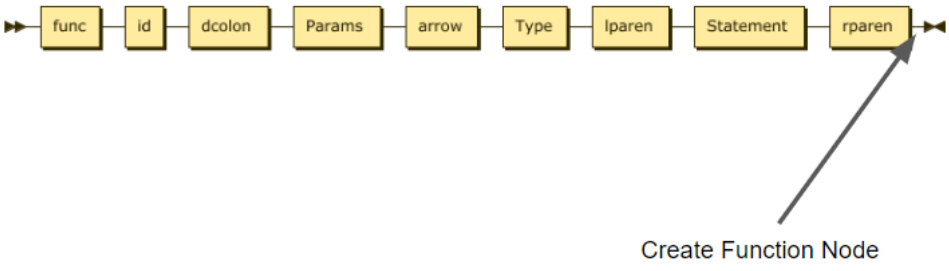
Program:



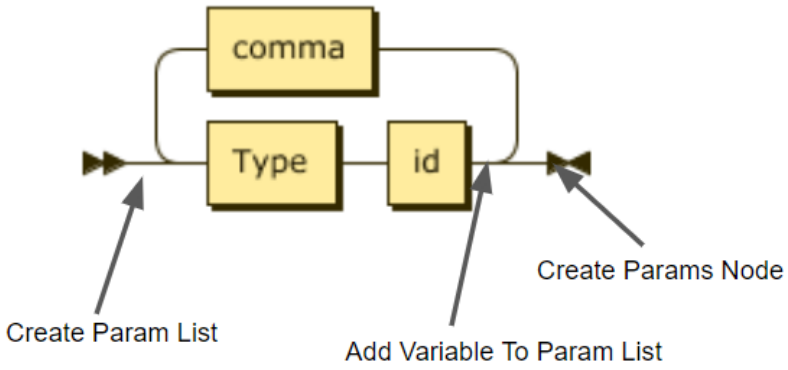
Functions:



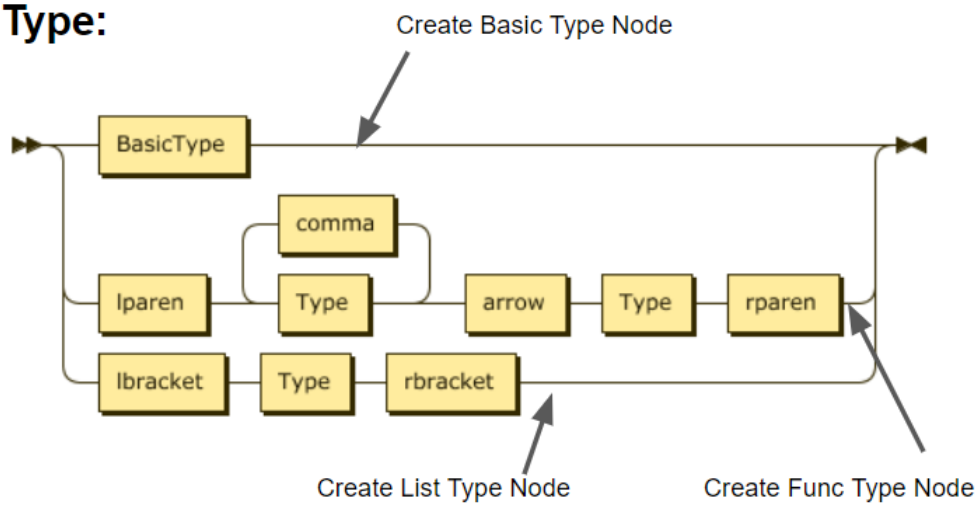
Function:



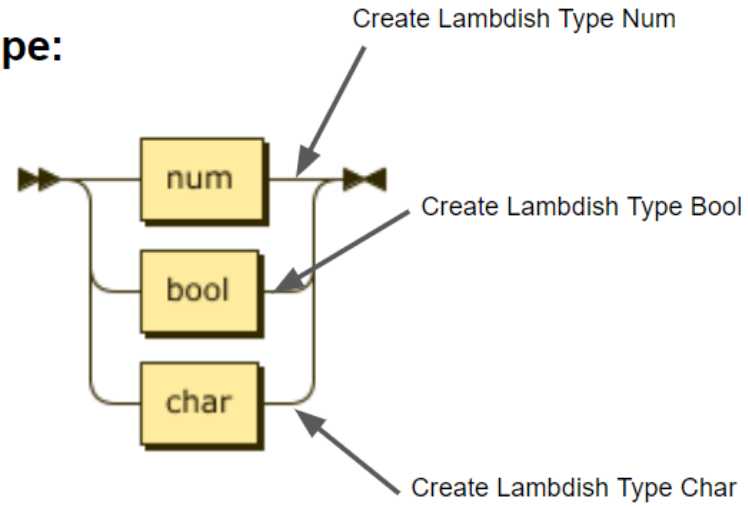
Params:



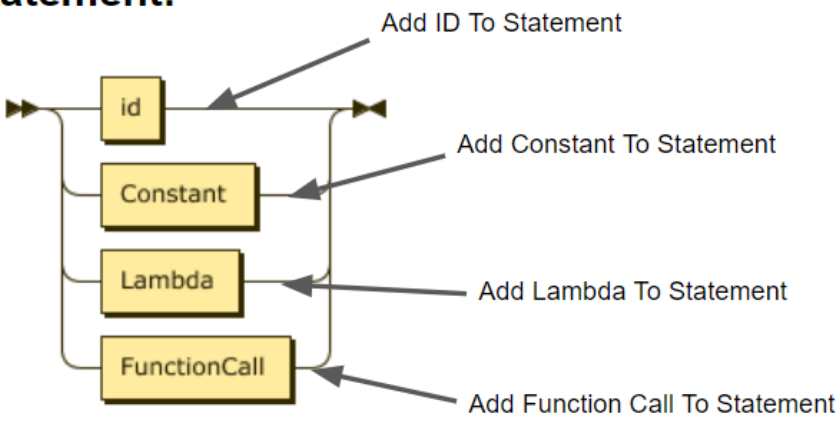
Type:



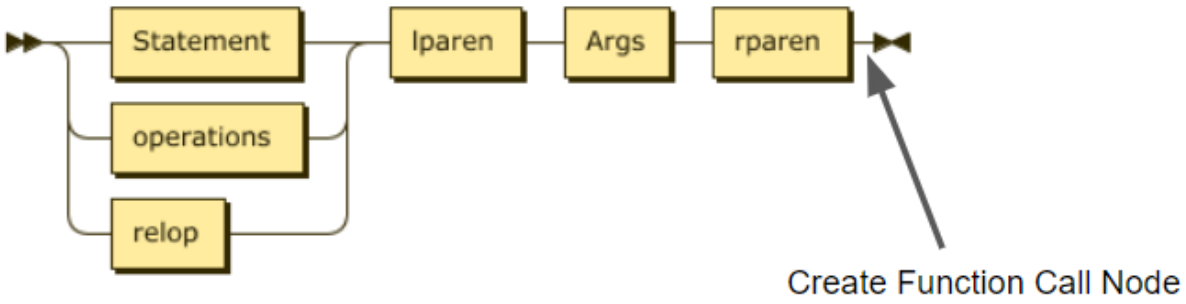
BasicType:



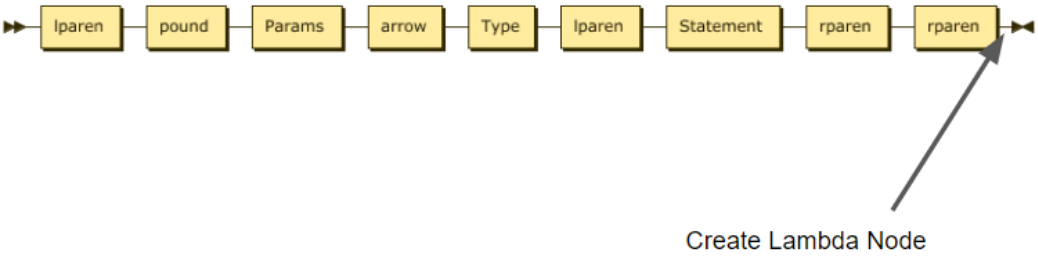
Statement:



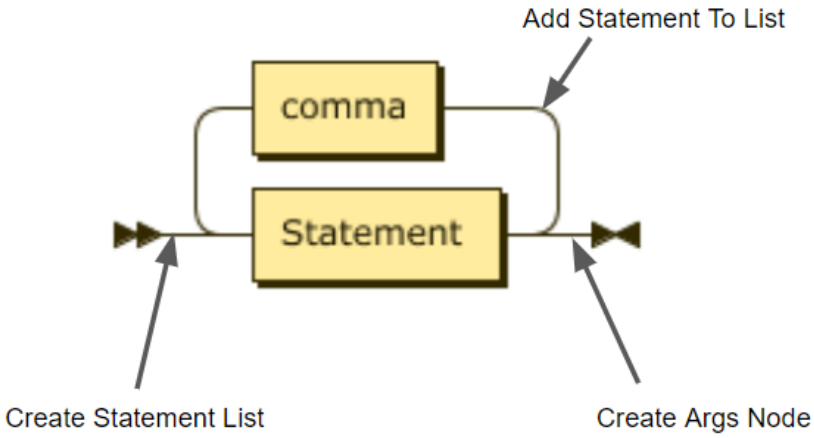
FunctionCall:



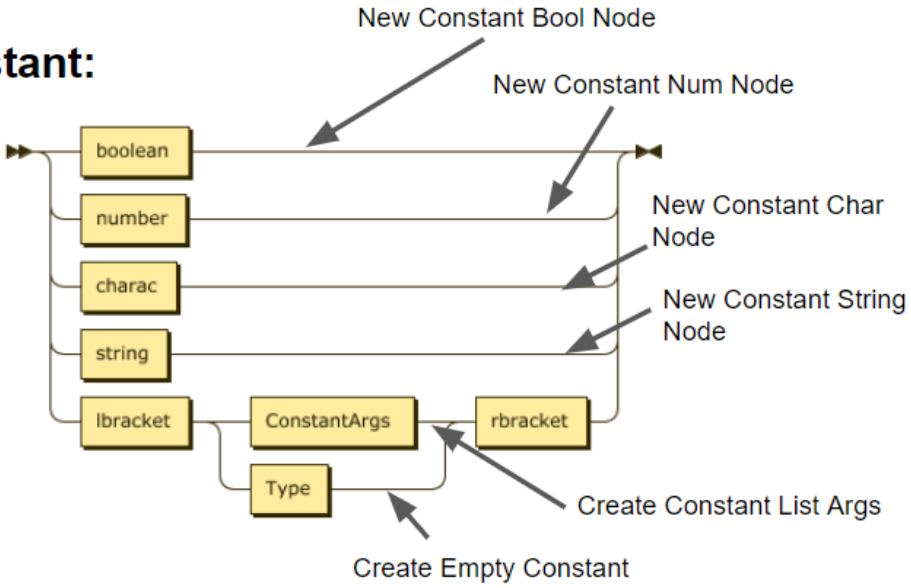
Lambda:



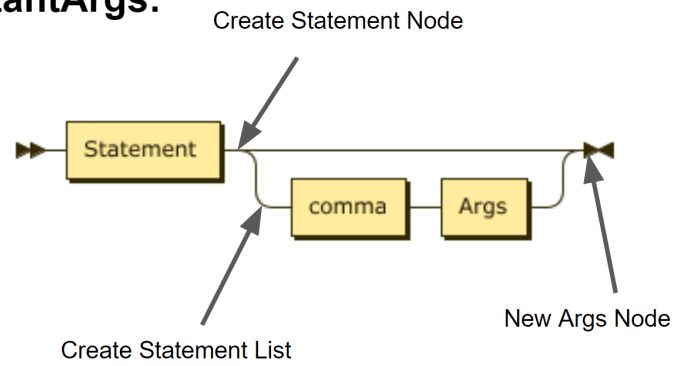
Args:



Constant:



ConstantArgs:



Acciones Semánticas y el AST

Estructura del AST

Las acciones semánticas en la gramática nos permiten construir nuestro árbol abstracto de sintaxis (AST). Para entender que hacer cada acción semántica primero hay que explicar la estructura de nuestra AST y cómo se construye para que sea de utilidad en en análisis semántico y la generación de código intermedio.

El AST es un árbol dirigido que representa los elementos de un programa Lambdish de una forma jerárquica. A continuación se enumeran los diferentes nodos del árbol. Para fines explicativos, los nodos que no son hoja se representan en negritas, mientras que los nodos hoja se representan con letra normal.

1. **Program**: El nodo raíz de todo programa
    a. Hijos:
        i. **Function**[ ]: Todas las funciones del programa
        ii. **FunctionCall:** La llamada principal que funciona como punto de entrada
2. **Function**: Representa una función en el programa
    a. Hijos:
        i. id: string: Nombre de la función
        ii. key: string: Llave en la tabla de funciones
        iii. type: LambdishType: Tipo de retorno
        iv. token: Token: Apuntador a la posición en el código
        v. **VarEntry[ ]:** Lista de parámetros de la función
        vi. **Statement:** Cuerpo de la función
3. **Statement**: Interface para representar alguno de los tipos de statement
    a. Puede tomar la forma de
        i. **Id**
        ii. **ConstantValue**
        iii. **ConstantList**
        iv. **Lambda**
        v. **FunctionCall**
4. **Id:** Representa un statement que solo es una variable
    a. Hijos
        i. id: string: Nombre de la variable
        ii. token: Token: Apuntador a la posición en el código
5. **FunctionCall:** Representa una llamada a una función
    a. Hijos
        i. **Statement:** Función a llamar
        ii. **Statement[ ]:** Lista de argumentos para llamar a la función
6. **Lambda:** Representa la declaración de una función lambda
    a. Hijos:

        i. returntype: LambdishType: tipo de retorno de la lambda
        ii. id: string: nombre interno de identificación para la lambda
        iii. token: Token: Apuntador a la posición en el código
        iv. VarEntry[ ]: Lista de parámetros de la función
        v. Statement: Cuerpo de la función
7. **ConstantValue:** Valor constante atómico, ya sea num, bool, o char.
    a. Hijos
        i. type: LambdishType: Tipo de dato de la constante
        ii. value: string: Representación literal del valor en forma de string
        iii. token: Token: Apuntador a la posición en el código
8. **ConstantList:** Representa una lista constante de cualquier tipo
    a. Hijos
        i. type: LambdishType: tipo de dato de la lista
        ii. token: Token: Apuntador a la posición en el código
        iii. Statement[ ]: Lista de elementos que se incluyen en la lista
9. **VarEntry:** Descripción de un parámetro dentro de una función
    a. Hijos
        i. id: string: nombre del parámetro
        ii. type: LambdishType: tipo del parámetro
        iii. token: Token: Apuntador a la posición en el código
        iv. addr: Address: dirección en la memoria virtual del parámetro

Con los nodos presentados la lista anterior se puede representar cualquier programa Lambdish. Por ejemplo, a continuación vamos a representar el siguiente pequeño programa en su AST equivalente.

```
func hola :: num x, num y => num (
     +(x, y)
)

func adios :: bool b => char (
     if (b, 'a', 'z')
)

hola(3)
```

Este pequeño fragmento de código genera el siguiente AST:

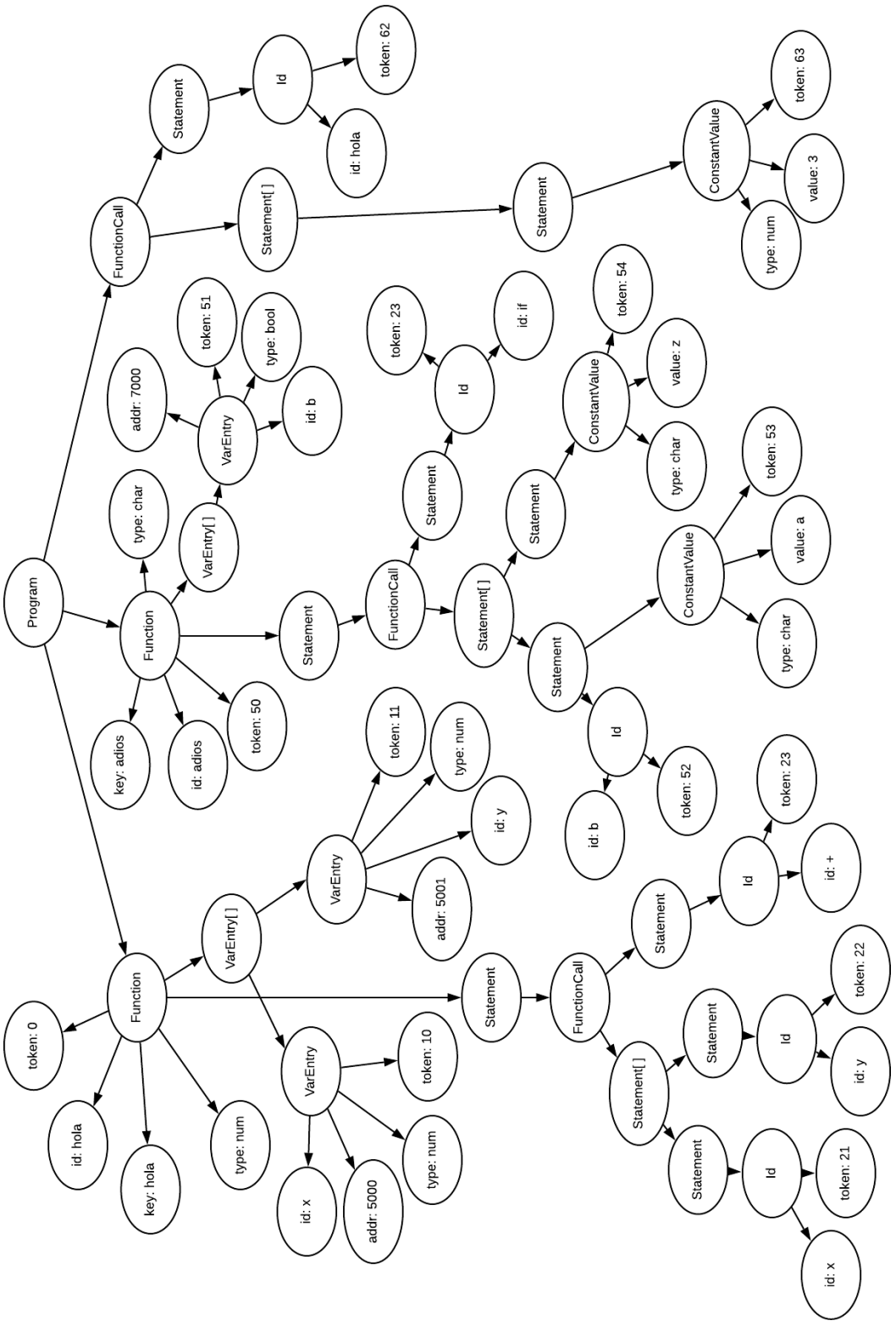


Figura 2. AST para el programa ejemplo

Si se requiere analizar el AST con más detalle, se puede visitar la siguiente liga donde se puede visualizar el ejemplo del árbol con mayor detalle:

https://app.lucidchart.com/invitations/accept/38f56b23-f3ae-4d47-843f-34c15f6e73ae

Con esto en mente, es más diferente comprender cómo las acciones semánticas introducidas en el diagrama de sintaxis ayudan a crear el AST. A continuación se presenta una pequeña descripción de cada acción semántica que se presentó en los diagramas de sintaxis.

- Program
  - Create Program Node: Crea el nodo raíz del árbol **Program**. Está acción se hace al final debido al bottom up parsing.
- Functions
  - Create Function List: Crea el arreglo de funciones
  - Add Function to List: Agrega la función que se acaba de parsear a la lista de funciones.
  - Create Functions Node: Crea el nodo **Function[]** que contiene el arreglo de todas las funciones.
- Function
  - Create Function node: Crea el nodo **Function** individual con todos los atributos de una función
- Params
  - Create Param List: Crea la lista de **VarEntry[]** para guardar los parámetros
  - Add Variable to Param List: Agrega un nuevo parámetro a la lista **VarEntry[]**
  - Create Params Node: Crea el nodo **VarEntry[]** para representar todos los parámetros.
- Type:
  - Create Basic Type Node: Crea un nodo Lambdish type de estilo básico
  - Create List Type Node: Crea un nodo Lambdish type de estilo lista
  - Create Func Type Node: crea un nodo Lambdish type de estilo función
- BasicType
  - Create Lambdish Type Num: Crea un lambdish type de tipo num
  - Create Lambdish Type Bool: Crea un lambdish type de tipo bool
  - Create Lambdish Type Char: Crea un lambdish type de tipo char
- Statement
  - Add ID To Statement: Crea un nodo **Statement** de tipo **ID**
  - Add Constant To Statement: Crea un nodo **Statement** de tipo **ConstantValue** o **ConstantList**
  - Add Lambda To Statement: Crea un nodo **Statement** de tipo **Lambda**
  - Add Function Call to Statement: Crea un nodo **Statement** de tipo **FunctionCall**
- FunctionCall
  - Create Function Call Node: Crea un nodo de tipo **FunctionCall**
- Lambda:

    - Create lambda Node: Crea un nuevo nodo de tipo **Lambda**
- Args:
    - Create Statement List: Crea una nueva lista de **Statement**
    - Add statement To List: Agrega el **Statement** generado a la lista de statements
    - Create Args Node: Crea el nodo de tipo **Statement[]** con los elementos agregados
- Constant:
    - New Constant Bool Node: Crea nuevo nodo **ConstantValue** de tipo bool
    - New Constant Num Node: Crea nuevo nodo **ConstantValue** de tipo num
    - New Constant Char Node: Crea nuevo nodo **ConstantValue** de tipo char
    - New Constant String Node: Crea nuevo nodo **ConstantValue** de tipo arreglo de char
    - Create Constant List Args: Crea nuevo nodo **ConstantList** con los elementos dados
    - Create Empty Constant: Crea nuevo nodo **ConstantList** vacío del tipo dado
- ConstantArgs:
    - Create Statement Node: Crea el nodo de **Statement**
    - Create Statement List: Crea una nueva lista de **Statement[]**
    - New Args Node: Crea un nuevo nodo de **Statement[]**

Hay que tomar en cuenta que el parser utilizado utiliza un enfoque bottom up. Por lo tanto, comienza construyendo primero las hojas del árbol y luego usa esa información para construir los nodos que van más arriba en el árbol. Por lo tanto, cuando se tiene que construir uno de los árboles posicionados más arriba cómo **Function**, este ya tiene toda la información de los nodos por debajo de él para construir su propio nodo.

Recorridos del AST

Una vez que tenemos construido nuestro AST, lo podemos recorrer las veces que sean necesarias para ejecutar todo tipo de chequeos semánticos y acciones de generación de código intermedio. Como se explicó al comienzo de esta sección, el AST que se genera se recorre un total de 5 veces con el propósito de hacer todo tipo de chequeos. A continuación se describe cada uno de los recorridos.

Construcción de Tablas de Funciones y de Variables

Lo primero que se hace es recorrer el árbol para generar la tabla de funciones globales, y las tablas de variables para cada una de las entradas. Esto se hace siguiendo los siguientes pasos.

1. Inicializar la tabla FuncDir vacía
2. Para cada nodo Function como hijo del nodo Program, crear una nueva entrada y agregarla a la tabla.
3. Si el id de la función ya está en la tabla genrar un ERROR de id repetido.

4. Para cada nodo Function, obtener los nodos VarEntry, y agregarlos a la Tabla de Variables de esa función.
5. Inicializar la tabla FuncDir para cada nodo Function
6. Para cada nodo Function, checar si tiene hijos de tipo lambda. Si si, crea una nueva entrada de FuncDir y agregarlo a la FuncDir de Function.
7. Para cada nodo Lambda obtener los nodos VarEntry, y agregarlos a la Tabla de Variables de esa lambda
8. Entrar al nodo de cada Lambda y crear una nueva FunDir
9. Para cada nodo lambda, checar si tiene hijos de tipo lambda. Si sí, crea una nueva entrada de FuncDir y agregarlo a la FuncDir de Lambda.
10. Repetir hasta que ya no haya más lambdas

Al terminar este proceso se debe de tener una tabla de tipo FuncDirectory, que debe de tener referencia a todas las funciones.

Chequeo de uso de variables en sus respectivos scopes

En este recorrido se checa que todos los ids accesados en el programa son válidos en el scope en el que se usan. Esto se hace siguiendo los siguientes pasos

1. Comenzar en la raíz Program
2. Para cada nodo Function, checar el nodo statement
3. Si el statement es:
    a. **ID**: Checa que el ID esté dado de alta en la tabla de variables de la función en la que se encuentra, si no, checar si hace referencia a una función global
    b. **ConstantValue**: no se realiza chequeo
    c. **ConstantList**: Repetir el paso 3 para cada statement de la lista
    d. **FunctionCall**: Repetir el paso 3 para el statement de la llamada y para los statements de los argumentos.
4. Para el nodo FunctionCall de program: Repetir el paso 3 para el statement de la llamada y para los statements de los argumentos

Si se genera cualquier tipo de error en el chequeo, este causa que el chequeo se aborte y el error generado es reportado al usuario.

Chequeo del uso correcto de tipos

En este recorrido se checa que el uso de los tipos de datos sea el adecuado. Este recorrido genera errores de discordancia de tipos. Para hacer este chequeo se siguen los siguientes pasos.

1. Comenzar en la raíz del programa
2. Para cada nodo Function checar el tipo de dato del nodo statement
3. Si no coincide,marca un error de discordancia de valor de retorno
4. Si el statement es
    a. **FunctionCall**: Si la función es
        i. Llamada **ID**: Si el id es

1. Función reservada (aritmética, lógica, relacional, builtin): Llamar el cubo semántico con los argumentos de la función y regresar error si se encuentra uno
2. Función no reservada: buscar función en los parámetros y en la tabla de funciones global. Comprar los tipos de datos de los argumentos con los que se requieren, si no coinciden regresa error.

ii. **Lambda**: comparar los argumentos con los parámetros de la lambda. Si no coinciden regresar error

5. Aplicar la regla 4a al nodo FunctionCall que está en el nodo Program.

Asignación de direcciones de la memoria virtual a las variables

En este recorrido se les asigna las direcciones de la memoria virtual dependiendo del tipo de dato que se esté recibiendo. Para hacer esta asignación antes de generar el código intermedio se siguen los siguientes pasos.

1. Comenzar en la raíz del programa
2. Para cada función en el directorio de funciones checar la tabla de variables
3. Para cada variable en la tabla de variables: generar una nueva dirección tomando en cuenta el tipo de dato.
4. Recursivamente checar los nodos **Statement** de cada función hasta encontrar las declaraciones constantes. Para cada constante diferente generar una nueva dirección en el segmento de las constantes.

Generación de cuádruplos correspondientes

En este recorrido se generan los cuádruplos de todo el código intermedio una vez asignado direcciones a la tabla de variables y al directorio de funciones. Para generar los cuádruplos correspondientes se siguen los siguientes pasos.

1. Comenzar en la raíz del programa.
2. Generar el cuádruplo GOTO “main” dónde indica en dónde empieza el IP.
3. Para cada nodo **Function** dentro del programa se reinician las variables temporales, se añade al stack de **FuncEntry** para generar el código para cada nodo **Statement.** Mover el IP y darle un valor de retorno a la función.
4. Para cada nodo **Statement,** si es:
    a. **ID:** Si la dirección
        i. **Función Paramétrica**: Se añade al stack pendiente la dirección de la llamada.
        ii. **Función Global**: Se busca dentro del directorio global, se agrega al stack de funciones con dirección pendiente y además se agrega la dirección pendiente por resolver.
        iii. En caso de no ser encontrada se muestra el error de que no se encontró en un contexto local ni global

b. **FunctionCall**: Si la función es
    i. Llamada **ID**: Si el id es
        1. Función reservada (aritmética, lógica, relacional, builtin): verificar el nombre de la función reservada, obtener el nombre del cuádruplo, resolver los parámetros y añadir las direcciones de los argumentos haciendo la búsqueda en el stack de la función para resolver las direcciones pendientes. Se genera el cuádruplo **CALL**.
        2. Función no reservada: Generar **ERA** de la función, resolver los parámetros de la función, verificar si una función ya está en el directorio global generar la llamada, sino se verifica si existe en un contexto local, se generan los temporales y el cuádruplo **CALL.**
    ii. **Lambda**: El arreglo de las funciones lambda se genera dependiendo del id o de la cantidad de lambdas que haya. Se define una lambda y se genera un **GOTO** para prevenir que se ejecute explícitamente. Se generaría el cuádruplo para **Statement,** dado que es la misma situación que una función. Una vez que se genera se llenan todas las direcciones pendientes del stack y se completa el **GOTO.**
    iii. **FunctionCall:** Puede ser posible que exista una llamada a otra FunctionCall, por lo que se dejaría pendiente las direcciones y seguiría resolviendo recursivamente. Se generaría el cuádruplo para **Statement.**
c. **Lambda:** Se define una lambda y se genera un **GOTO** para prevenir que se ejecute explícitamente. Se generaría el cuádruplo para **Statement,** dado que es la misma situación que una función. Una vez que se genera se llenan todas las direcciones pendientes del stack y se completa el **GOTO.**
d. **Constant List:** Se determina el tipo de la lista constante, se hace una lista, se obtienen los elementos de la lista, se genera el cuádruplo **LST.** Para obtener los contenidos de la lista, se verifica el tipo de cada elemento y se obtiene la dirección para generar **PaLst.** Una vez terminados los elementos se genera la dirección de la lista y se obtiene con **GeLst.**
e. **Constant Value:** Se determina el tipo de constantes, se verifica en el mapa de constantes en la memoria y se deja genera la dirección al stack de pendientes para llenar otros cuádrplos.

Tabla de consideraciones semánticas

Para poder identificar las consideraciones semánticas del compilador se utilizó una estructura de mapa de un string al tipo de dato de retorno para la operación deseada. Se desarrolló una nueva estrategia para generar una nueva llave para el mapa.

Para generar la llave se seguía el siguiente esquema:

Llave: **<NombreFunción>@<Parámetros>**

La llave está compuesta por el símbolo o el nombre de la función, el delimitador “@”, seguido de la representación del tipo de datos del sistema siendo 1 el **Num**, el 2 **Bool** y el 3 **Char**. La cantidad de parámetros se determina por la longitud después del delimitador.

```
 1. "+@11":          types.Num
 2. "-@11":          types.Num
 3. "/@11":          types.Num
 4. "*@11":          types.Num
 5. "%@11":          types.Num
 6. "<@11":          types.Bool
 7. ">@11":          types.Bool
 8. "Equal@11":      types.Bool
 9. "Equal@22":      types.Bool
10. "Equal@33":      types.Bool
11. "And@33":        types.Bool
12. "Or@33":         types.Bool
13. "!@3":           types.Bool
```

Dentro de las identificadores del cubo semántico se identifica si la función ya está preinstalada por lo que se hace de acuerdo al tipo de argumentos que se recibe. Por lo que existen otras funciones para identificar si vienen dentro de funciones reservadas o nombre reservados para las funciones.

e) Descripción del Proceso de Administración de Memoria

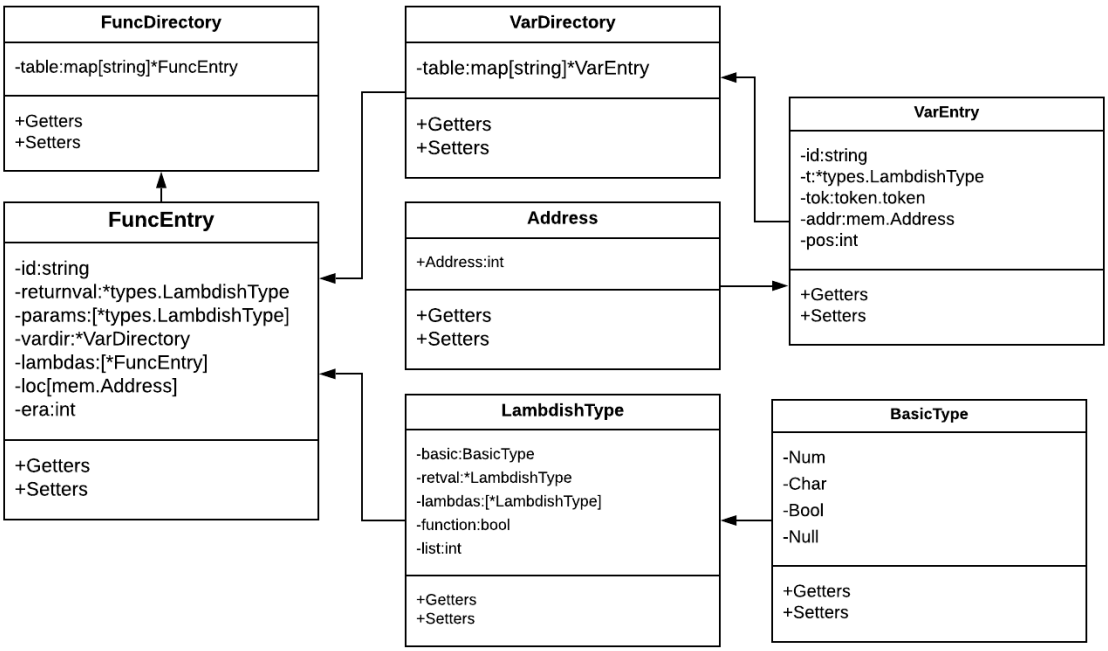


Figura 3 Diagrama de clases de directorio de funciones

Este diagrama muestra el diagrama de clases de las estructuras que se utilizaron para la generación del directorio de funciones. El directorio de funciones está representado con la estructura FuncDirectory. Esta tiene un mapa de la estructura FuncEntry. Esta última se utiliza para representar cada función en el programa. Cada FuncEntry entonces tiene un directorio de variables que se representa con la estructura VarDirectory. Las entradas de VarDirectory representan cada uno de los parámetros de la función, ya que en nuestro contexto no existen las variables locales mas que los parámetros.

Además, creamos la estructura LambdishType. Ésta nos sirve para representar un tipo de dato en el lenguaje Lambdish. Varias estructuras usan LambdishType para representar sus variables internas como VarEntry y FuncEntry. LambdishType se puede volver bastante complejo ya que puede representar datos atómicos, listas de datos, listas de listas, o hasta funciones como tipos de datos, especificando los tipos de datos de los parámetros y el tipo de retorno. Para representar un tipo de dato atómico, se utiliza la estructura BasicType. Por lo tanto LambdishType contiene un BasicType cuando se trata de un tipo de dato atómico.

FuncEntry, además de tener un directorio de variables (VarDirectory), también tiene un arreglo de FuncEntry. Este arreglo representa las funciones lambdas que están declaradas justo dentro de la función representada por FuncEntry. Esta recursividad nos permite declarar

lambdas adentro de funciones y además, nos permite declarar más lambdas dentro de lambdas y así sucesivamente.

Mapa de memoria

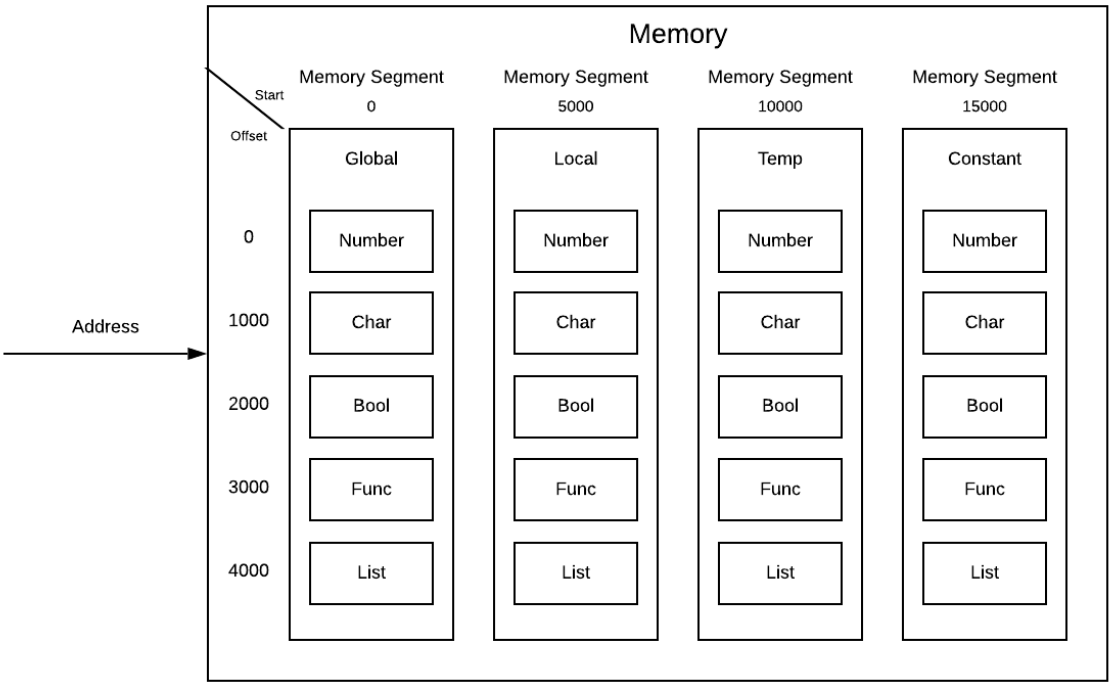


Figura 4 Mapa de la Memoria

En este diagrama se muestra el manejo de memoria al momento de compilar. Como se puede observar existen 4 segmentos en la memoria: el global, local, temporal y constante. Cada segmento comienza en un número diferente que se muestra en la parte superior de cada segmento.

Además cada segmento de memoria se divide por tipo de dato. Después de realizar un análisis profundo se descubrió que solo se requieren 5 divisiones para los tipos de datos: una para los números, para los chars, para los booleanos, para las funciones, y para las listas. Cada división tiene lo que se le conoce como un offset que se muestra a la izquierda del diagrama.

Asignar una nueva dirección

Cuando queremos asignar una nueva variable a memoria. Primero tenemos que identificar a qué segmento va a pertenecer, y después qué tipo de dato es. Por ejemplo, si queremos guardar un nuevo temporal booleano que acabamos de generar, para encontrar su

dirección primero agarramos el start de los temporales, que en este caso es 100000. Ahora vemos el offset del tipo de dato, que en este caso es 2000, por lo que ahora tenemos la dirección 12000. Sin embargo, hay que primero verificar si hay otros booleanos asignados, por lo que la fórmula sería sumar a la dirección que ya tenemos la cantidad de booleanos en este segmento ya asignados. Vamos a asumir que ya hay 4 en este segmento. Por lo tanto nuestra dirección final sería 12000 + 4 = 12004.

4) Descripción de la Máquina Virtual

La máquina virtual es un programa bastante simple que ejecuta los cuádruplos dados de manera secuencial sin ningún otro tipo de inteligencia. Al comienzo de su ejecución, la máquina virtual lee un archivo .obj que contiene 2 cosas: los cuádruplos a generar, y las constantes que se necesitan. Entonces, primero lee todos los cuádruplos y los guarda en memoria, después lee cada una de las constantes y las guarda en la memoria virtual (en la siguiente sección explicaremos cómo funciona esta memoria virtual).

Después de cargar todos los datos, comienza a ejecutar cada cuádruplo, comenzando con la primera posición. Una vez que llegue al final de los cuádruplos, la máquina se detiene y el programa como tal se termina, mostrando al usuario el resultado del programa en la consola.

1. Descripción del proceso de Administración de Memoria en ejecución

a. Manejo de Segmentos en la Memoria Virtual

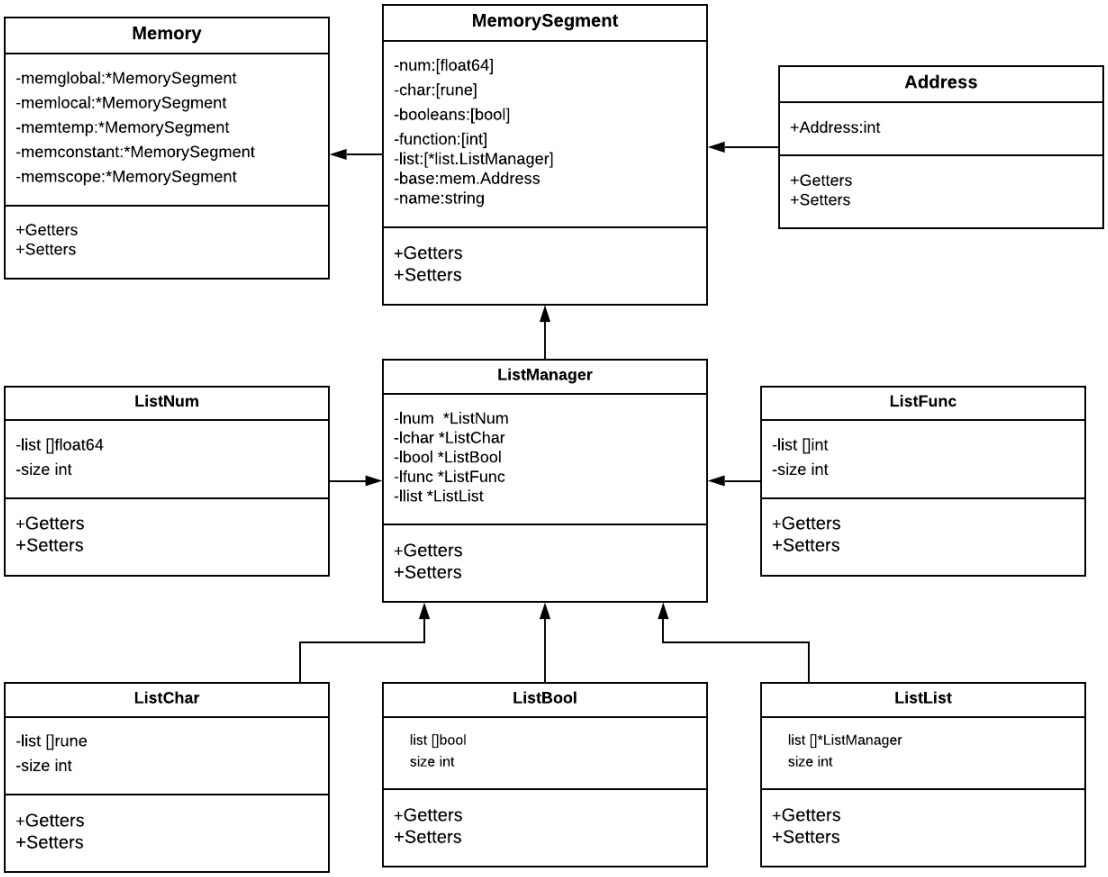


Figura 5 Diagrama de clases de la memoria virtual

Este diagrama nos muestra las estructuras utilizadas para representar la memoria en la máquina virtual. La estructura más importante aquí es Memory, la cual contiene el resto de las estructuras. Memory tiene 5 estructuras de tipo MemorySegment. Esta última es la representación de la fragmentación de memoria. MemorySegment tiene entonces 5 arreglos. Estos 5 arreglos representa cada uno de los tipos de datos que se guardan en un segmento.

Por ejemplo, para guardar los datos de tipo num (flotantes) se tiene un arreglo del tipo de dato float64. Es importante recalcar que el arreglo no es de tamaño 1000 inicialmente (que es el tamaño máximo de elementos que se pueden guardar) sino que comienza como un arreglo de longitud 0 y va incrementando su tamaño conforme se asignan más valores. Por lo tanto, en la máquina virtual, solo se asigna la memoria que realmente se necesita.

Cómo se guardan los valores

Como acabamos de mencionar, para guardar los datos de tipo num, se utiliza un arreglo de float64, que en Go nos da una precisión de lo que sería un double en C. Para guardar datos de tipo char se usa un arreglo de rune, el cual es el equivalente de char en go, y para los booleanos se usa un arreglo de tipo bool. Lo interesante aquí es cómo se guardan las funciones. Después de un duro análisis del diseño, se decidió que las funciones de guardarán como tipo entero. Este número representa una posición en los cuádruplos, y ese cuádruplo al que apunta ese número es entonces el inicio de tal función.

Cómo se manejan las listas

En el caso de las listas, se tomó un enfoque diferente. En el lenguaje Lambdish, las listas no tienen un tamaño fijo, sino que reducen o aumentan su tamaño a petición del usuario, por lo que se requería un poco más de complejidad para representarlas en memoria. La decisión que se tomó fue guardar las listas como un objeto ListManager.

ListManager es una estructura que se encarga de representar una lista en la máquina virtual. Una lista puede ser de diferentes tipos pero siempre homogénea. Por lo tanto ListManager tiene 5 objetos, uno para cada tipo de lista: ListNum, ListChar, ListBool, ListFunc y ListList. Como una lista solo es de un tipo, solo uno de estos objetos puede estar instanciado, pero esto nos permite representar cualquier tipo de lista con un mismo objeto ListManager. ListNum tiene un arreglo de float64 para representar sus elementos, ListChar tiene un arreglo de rune, ListBool tiene un arreglo de bool, y ListFunc tiene un arreglo de enteros (por qué las funciones se guardan como enteros se explica al inicio de la sección).

El último objeto de ListManager es ListList. Esta última estructura contiene un arreglo de objetos ListManager, por lo tanto nos sirve para representar listas de listas. Este tipo de recursividad nos permite tener listas de listas al nivel de anidación que el usuario quiera sin ninguna restricción del sistema.

b. Asociación hecha entre las direcciones virtuales y las reales .

Identificar una dirección

Aquí vamos a ver a ver cómo podemos ubicar una dirección que nos dan, por ejemplo, 6534, en la estructura de memoria de ejecución. Para esta explicación vamos a hacer referencia a la figura 3, el mapa de memoria, ya que seguimos la misma estructura tanto en compilación para generar las direcciones, como en ejecución para guardar los valores. Primero que nada, la dirección que nos interesa, 6534, está entre 5000 y 10000 por lo que, si consultamos la figura, sabemos que se trata de segmento local, por lo que está en memlocal de la estructura Memory.

Para ver qué tipo de dato representa, podemos entonces restarle el start del segmento local, en este caso es 5000, por lo que tendríamos 6534 - 5000 = 1534. Ahora vemos el offset y vemos que está entre 1000 y 2000, por lo que se debe de tratar de un char. Entonces le restamos el offset y nos queda 1534 - 1000 = 534.

Ahora sabemos que se trata de un char en nuestro segmento de locales, por lo que debe de estar en el arreglo de []rune dentro de nuestro MemorySegment. El número que nos queda, 534, nos dice que está en la posición 534 de este arreglo. Por lo que podemos ir a consultar este valor directamente. Además, esto también nos dice que existen otros 534 chars guardados en la memoria local porque el manejo de la memoria tanto en compilación al momento de generarla, como en runtime es eficiente, por lo que solo asigna números conforme se van ocupando más variables. El número 534 nos dice que antes de esta variable otros 534 chars pidieron ser almacenados en memoria y ocupan esas casillas empezando en la 0.

Toda esta complejidad la manera una función especial en la máquina virtual que se llama

GetValue(mem.Address)

Esta función se encarga de pasar por todo el proceso que acabamos de explicar, primero, encuentra qué MemorySegment es, luego identifica el tipo de dato, y luego consulta el arreglo correspondiente. Sí la dirección está fuera de rango, o no existe la dirección dada, este regresa un error y causa un runtime error.

Cabe recalcar que este runtime error no puede suceder al menos que el usuario modifique directamente el código objeto generado por el compilador. Ya que el orden de acceso en los cuádruplos, garantizan que siempre va a haber un valor en la casilla que se está accediendo.

5) Pruebas del Funcionamiento del Lenguaje

Test_Algorithms.lsh

```
//sortParity - Test 01
func filter :: [num] l, (num => bool) f => [num] (
        if (empty(l),
        [num],
        if (f(head(l)),
        insert(
                head(l),
                filter(tail(l), f)
        ),
        filter(tail(l), f)
        )
        )
)

func sortParity :: [num] l => [num] (
        append(
        filter(l, (# num x => bool (
        equal(%(x, 2), 0)
        ))),
        filter(l, (# num x => bool (
        equal(%(x, 2), 1)
        ))),
        )
)
//Merge Sort - Test 02
func length :: [num] l => num (
        if (empty(l),
        0,
        +(1, length(tail(l)))
        )
)
func take :: num n, [num] l => [num] (
        if (empty(l),
        l,
        if (or(equal(n, 0), <(n, 0)),
        [num],
        insert(head(l), take(-(n, 1), tail(l)))
        )
        )
)
func drop :: num n, [num] l => [num] (
        if (or(equal(n, 0), <(n, 0)),
        l,
        drop(-(n, 1), tail(l))
        )
)

func mergeSortAux :: [num] xl, [num] yl => [num] (
        if (empty(yl),
        xl,
        if (empty(xl),
        yl,
        if (<(head(xl), head(yl)),
                insert(
                head(xl),
                mergeSortAux(tail(xl), yl)
                ),
                insert(
                head(yl),
                mergeSortAux(xl, tail(yl))
                )
        )
        )
        )
)

func mergeSort :: [num] l => [num] (
        if (empty(l),
        [num],
        if (equal(length(l), 1),
        [head(l)],
        mergeSortAux(
                mergeSort(
                take(/(length(l), 2), l)
                ),
                mergeSort(
                drop(/(length(l), 2), l)
                )
        )
        )
        )
)

//Reduce - Test 03
func reduce :: [num] l, num x, (num, num => num) f =>
num (
        if (empty(l),
        x,
        f(
        head(l),
        reduce(tail(l), x, f)
        )
        )
)

//fibonacci - Test 04
func fibonacci :: num n => num (
    if(<(n, 0),
        0,
        if(<(n, 2),
        n,
                 +(
                fibonacci(-(n,1)),
                fibonacci(-(n,2))
        )
        )
        )
)

sortParity([1,2,3,4,5,6,7,8,9,0])
//sortParity([3,1,2,4])
//mergeSort([5,3,2,3,1])
// reduce(
//       [1,2,3,4,5], 0,
//       (# num x, num y => num (
//       +(x, y)
```

```
//        )
// ))
```

Test_Algorithms.obj

```
226
Goto -1 -1 210
Emp 9000 -1 12000
GotoF 12000 -1 7
Lst 1 -1 0
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 28
Era -1 -1 -1
Head 9000 -1 10000
Param 10000 -1 0
Call 8000 -1 12001
GotoF 12001 -1 21
Head 9000 -1 10001
Era -1 -1 -1
Tail 9000 -1 14003
Param 14003 -1 0
Param 8000 -1 1
Call 1 -1 14004
Ins 10001 14004 14005
Assign 14005 -1 14002
Goto 14005 -1 27
Era -1 -1 -1
Tail 9000 -1 14006
Param 14006 -1 0
Param 8000 -1 1
Call 1 -1 14007
Assign 14007 -1 14002
Assign 14002 -1 14000
Ret 14000 -1 -1
Era -1 -1 -1
Param 9000 -1 0
Goto -1 -1 35
% 5000 15000 10000
Equal 10000 15001 12000
Ret 12000 -1 -1
Param 32 -1 1
Call 1 -1 14000
Era -1 -1 -1
Param 9000 -1 0
Goto -1 -1 43
% 5000 15000 10001
Equal 10001 15002 12001
Ret 12001 -1 -1
Param 40 -1 1
Call 1 -1 14001
App 14000 14001 14002
Ret 14002 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 51
Assign 15001 -1 10000
Goto 15001 -1 57
Era -1 -1 -1
Tail 9000 -1 14000
Param 14000 -1 0
Call 47 -1 10001
+ 15002 10001 10002
Assign 10002 -1 10000
Ret 10000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 62
Assign 9000 -1 14000
Goto 9000 -1 80
Equal 5000 15001 12001
< 5000 15001 12002
Or 12001 12002 12003
GotoF 12003 -1 70
Lst 1 -1 0
GeLst -1 -1 14002
Assign 14002 -1 14001
Goto 14002 -1 79
Head 9000 -1 10000
Era -1 -1 -1
- 5000 15002 10001
Param 10001 -1 0
Tail 9000 -1 14003
Param 14003 -1 1
Call 58 -1 14004
Ins 10000 14004 14005
Assign 14005 -1 14001
Assign 14001 -1 14000
Ret 14000 -1 -1
Equal 5000 15001 12000
< 5000 15001 12001
Or 12000 12001 12002
GotoF 12002 -1 87
Assign 9000 -1 14000
Goto 9000 -1 94
Era -1 -1 -1
- 5000 15002 10000
Param 10000 -1 0
Tail 9000 -1 14001
Param 14001 -1 1
Call 81 -1 14002
Assign 14002 -1 14000
Ret 14000 -1 -1
Emp 9001 -1 12000
GotoF 12000 -1 99
Assign 9000 -1 14000
Goto 9000 -1 126
Emp 9000 -1 12001
GotoF 12001 -1 103
Assign 9001 -1 14001
Goto 9001 -1 125
Head 9000 -1 10000
Head 9001 -1 10001
< 10000 10001 12002
GotoF 12002 -1 116
Head 9000 -1 10002
Era -1 -1 -1
Tail 9000 -1 14003
Param 14003 -1 0
Param 9001 -1 1
Call 95 -1 14004
Ins 10002 14004 14005
Assign 14005 -1 14002
Goto 14005 -1 124
Head 9001 -1 10003
Era -1 -1 -1
Param 9000 -1 0
Tail 9001 -1 14006
Param 14006 -1 1
Call 95 -1 14007
Ins 10003 14007 14008
Assign 14008 -1 14002
Assign 14002 -1 14001
Assign 14001 -1 14000
Ret 14000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 133
Lst 1 -1 0
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 172
Era -1 -1 -1
Param 9000 -1 0
Call 47 -1 10000
Equal 10000 15002 12001
GotoF 12001 -1 144
Lst 1 -1 1
Head 9000 -1 10001
PaLst 10001 -1 0
GeLst -1 -1 14003
Assign 14003 -1 14002
Goto 14003 -1 171
Era -1 -1 -1
Era -1 -1 -1
Era -1 -1 -1
Era -1 -1 -1
Param 9000 -1 0
Call 47 -1 10002
/ 10002 15000 10003
Param 10003 -1 0
Param 9000 -1 1
Call 58 -1 14004
Param 14004 -1 0
Call 127 -1 14005
Param 14005 -1 0
Era -1 -1 -1
Era -1 -1 -1
Era -1 -1 -1
Param 9000 -1 0
Call 47 -1 10004
/ 10004 15000 10005
Param 10005 -1 0
Param 9000 -1 1
Call 81 -1 14006
Param 14006 -1 0
Call 127 -1 14007
Param 14007 -1 1
Call 95 -1 14008
Assign 14008 -1 14002
Assign 14002 -1 14000
Ret 14000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 177
Assign 5000 -1 10000
Goto 5000 -1 189
Era -1 -1 -1
Head 9000 -1 10001
Param 10001 -1 0
Era -1 -1 -1
Tail 9000 -1 14000
Param 14000 -1 0
Param 5000 -1 1
Param 8000 -1 2
```

```
Call 173 -1 10002
Param 10002 -1 1
Call 8000 -1 10003
Assign 10003 -1 10000
Ret 10000 -1 -1
< 5000 15001 12000
GotoF 12000 -1 194
Assign 15001 -1 10000
Goto 15001 -1 209
< 5000 15000 12001
GotoF 12001 -1 198
Assign 5000 -1 10001
Goto 5000 -1 208
Era -1 -1 -1
- 5000 15002 10002
Param 10002 -1 0
Call 190 -1 10003
Era -1 -1 -1
- 5000 15000 10004
Param 10004 -1 0
Call 190 -1 10005
+ 10003 10005 10006
Assign 10006 -1 10001
Assign 10001 -1 10000
Ret 10000 -1 -1
Era -1 -1 -1
Lst 1 -1 10
```

```
PaLst 15002 -1 0
PaLst 15000 -1 1
PaLst 15003 -1 2
PaLst 15004 -1 3
PaLst 15005 -1 4
PaLst 15006 -1 5
PaLst 15007 -1 6
PaLst 15008 -1 7
PaLst 15009 -1 8
PaLst 15001 -1 9
GeLst -1 -1 14000
Param 14000 -1 0
Call 29 -1 14001
Print -1 -1 14001
10
1 15002
3 15003
4 15004
5 15005
6 15006
7 15007
2 15000
0 15001
8 15008
9 15009
```

Test_Algorithms Output

```
vsapiens@Usuarios-MacBook-Pro:~/Documents/lambdish-compiler/examples$ rlamb test_algorithms.obj
[2, 4, 6, 8, 0, 1, 3, 5, 7, 9]
```

Test_Error.lsh

```
func toggleDivisionByZero :: bool m => num (
        if(m,
        /(3,0),
        /(3,1)
        )
)

func toggleEmptyList :: bool m => num (
        if(m,
        head([num]),
        head([1,2,3,4,5,6,8])
        )
)

func map :: [num] l, (num => num) f => [num] (
        if (empty(l),
        [0],
        insert(f(head(l)), map(tail(l), f))
        )
)

toggleEmptyList(false)
//toggleEmptyList(true)
//toggleDivisionByZero(true)
//toggleDivisionByZero(true)
//map([1, -2, 3.234, 4], (# num x => num (
//       +(x, 1)
//)))
```

Test_Error.obj

```
49
Goto -1 -1 45
```

```
GotoF 7000 -1 5
/ 15000 15001 10001
Assign 10001 -1 10000
Goto 10001 -1 7
/ 15000 15002 10002
Assign 10002 -1 10000
Ret 10000 -1 -1
GotoF 7000 -1 14
Lst 1 -1 0
GeLst -1 -1 14000
Head 14000 -1 10001
Assign 10001 -1 10000
Goto 10001 -1 25
Lst 1 -1 7
PaLst 15002 -1 0
PaLst 15003 -1 1
PaLst 15000 -1 2
PaLst 15004 -1 3
PaLst 15005 -1 4
PaLst 15006 -1 5
PaLst 15007 -1 6
GeLst -1 -1 14001
Head 14001 -1 10002
Assign 10002 -1 10000
Ret 10000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 33
Lst 1 -1 1
PaLst 15001 -1 0
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 44
Era -1 -1 -1
Head 9000 -1 10000
Param 10000 -1 0
Call 8000 -1 10001
Era -1 -1 -1
Tail 9000 -1 14002
Param 14002 -1 0
Param 8000 -1 1
Call 26 -1 14003
Ins 10001 14003 14004
Assign 14004 -1 14000
Ret 14000 -1 -1
Era -1 -1 -1
Param 17000 -1 0
Call 8 -1 10002
Print -1 -1 10002
9
```

```
2 15003
5 15005
6 15006
8 15007
false 17000
3 15000
```

```
0 15001
1 15002
4 15004
```

Test_Error Output

```
vsapiens@Usuarios-MacBook-Pro:~/Documents/lambdish-compiler/examples$ rlamb test_error.obj
1
```

Test_Lambdas.lsh

```
//Filter Test-01
func filter :: [num] l, (num => bool) f => [num] (
        if (empty(l),
        [num],
        if (f(head(l)),
        insert(
                head(l),
                filter(tail(l), f)
        ),
        filter(tail(l), f)
        )
        )
)
//Map Test-02
func map :: [num] l, (num => num) f => [num] (
        if (empty(l),
        [num],
        insert(
        f(head(l)),
        map(tail(l), f)
        )
        )
)
//GetOperation Test-03
func getOperation :: num x, num y => ( => (num, num
=> num)) (
        if (<(x, y),
        sum,
        sub
        )
)

func sum :: => (num, num => num) (
        (# num x, num y => num (
        +(x, y)
        ))
)

func sub :: => (num, num => num) (
        (# num x, num y => num (
        -(x, y)
        ))
)

func operate :: num x, num y => num (
        getOperation(x, y)()(x, y)
)

map([1,2,3,4], (# num x => num (*(*(x, x), x))))
//filter([1,2,3,4,5,6], (# num x => bool (<(x, 4))))
//operate(3, 4)
```

Test_Lambda.obj

```
87
Goto -1 -1 72
Emp 9000 -1 12000
GotoF 12000 -1 7
Lst 1 -1 0
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 28
Era -1 -1 -1
Head 9000 -1 10000
Param 10000 -1 0
Call 8000 -1 12001
GotoF 12001 -1 21
Head 9000 -1 10001
Era -1 -1 -1
Tail 9000 -1 14003
Param 14003 -1 0
Param 8000 -1 1
Call 1 -1 14004
Ins 10001 14004 14005
Assign 14005 -1 14002
Goto 14005 -1 27
Era -1 -1 -1
Tail 9000 -1 14006
Param 14006 -1 0
Param 8000 -1 1
Call 1 -1 14007
Assign 14007 -1 14002
Assign 14002 -1 14000
Ret 14000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 35
Lst 1 -1 0
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 46
Era -1 -1 -1
Head 9000 -1 10000
Param 10000 -1 0
Call 8000 -1 10001
Era -1 -1 -1
Tail 9000 -1 14002
Param 14002 -1 0
Param 8000 -1 1
Call 29 -1 14003
Ins 10001 14003 14004
Assign 14004 -1 14000
Ret 14000 -1 -1
< 5000 5001 12000
GotoF 12000 -1 51
Assign 53 -1 13000
Goto -1 -1 52
Assign 57 -1 13000
Ret 13000 -1 -1
Goto -1 -1 56
+ 5000 5001 10000
Ret 10000 -1 -1
Ret 54 -1 -1
Goto -1 -1 60
- 5000 5001 10000
Ret 10000 -1 -1
Ret 58 -1 -1
```

```
Era -1 -1 -1
Param 5000 -1 0
Param 5001 -1 1
Call 47 -1 13000
Era 0 -1 -1
Call 13000 -1 13001
Era 0 -1 -1
Param 5000 -1 0
Param 5001 -1 1
Call 13001 -1 10000
Ret 10000 -1 -1
Era -1 -1 -1
Lst 1 -1 4
PaLst 15000 -1 0
PaLst 15001 -1 1
PaLst 15002 -1 2
PaLst 15003 -1 3
GeLst -1 -1 14000
Param 14000 -1 0
Goto -1 -1 84
* 5000 5000 10001
* 10001 5000 10002
Ret 10002 -1 -1
Param 81 -1 1
Call 29 -1 14001
Print -1 -1 14001
4
3 15002
4 15003
1 15000
2 15001
```

Test_Lambda Output

```
vsapiens@Usuarios-MacBook-Pro:~/Documents/lambdish-compiler/examples$ rlamb test_lambdas.obj
[1, 8, 27, 64]
```

Test_Matrix.lsh

```
func flatten:: [[num]] x => [num] (
        if(empty(x),
        [num],
        if(empty(head(x)),
        flatten(tail(x)),
        insert(
                head(head(x)),
                flatten(
                insert(
                        tail(head(x)),
                        tail(x)
                )
                )
        )
        )
        )
)

func flattenMat :: [[num]] x => [num] (
        if(empty(x),
        [num],
        if( empty(head(x)),
                flatten(tail(x)),
                insert(
                head(head(x)),
                flatten(insert(tail(head(x))
,tail(x)))
                )
        )
        )
)
func sumAux :: [num] x => num (
        if(empty(x),
        0,
        +(head(x),sumAux(tail(x)))
        )
)

func sumMat :: [[num]] x => num (
        if(empty(x),
        0,
        +(sumAux(head(x)),sumMat(tail(x)))
        )
)

sumMat([
        [1,2,3],
        [4,5,6],
        [7,8,9]
])

// flattenMat([
//      [1,2,3],
//      [4,5,6],
//      [7,8,9]
// ])
```

Test_Matrix.obj

```
108
Goto -1 -1 84
Emp 9000 -1 12000
GotoF 12000 -1 7
Lst 1 -1 0
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 28
Head 9000 -1 14003
Emp 14003 -1 12001
GotoF 12001 -1 16
Era -1 -1 -1
Tail 9000 -1 14004
Param 14004 -1 0
Call 1 -1 14005
Assign 14005 -1 14002
Goto 14005 -1 27
Head 9000 -1 14006
Head 14006 -1 10000
Era -1 -1 -1
Head 9000 -1 14007
Tail 14007 -1 14008
Tail 9000 -1 14009
Ins 14008 14009 14010
Param 14010 -1 0
Call 1 -1 14011
Ins 10000 14011 14012
Assign 14012 -1 14002
Assign 14002 -1 14000
Ret 14000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 35
Lst 1 -1 0
```

```
GeLst -1 -1 14001
Assign 14001 -1 14000
Goto 14001 -1 56
Head 9000 -1 14003
Emp 14003 -1 12001
GotoF 12001 -1 44
Era -1 -1 -1
Tail 9000 -1 14004
Param 14004 -1 0
Call 1 -1 14005
Assign 14005 -1 14002
Goto 14005 -1 55
Head 9000 -1 14006
Head 14006 -1 10000
Era -1 -1 -1
Head 9000 -1 14007
Tail 14007 -1 14008
Tail 9000 -1 14009
Ins 14008 14009 14010
Param 14010 -1 0
Call 1 -1 14011
Ins 10000 14011 14012
Assign 14012 -1 14002
Assign 14002 -1 14000
Ret 14000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 61
Assign 15000 -1 10000
Goto 15000 -1 68
Head 9000 -1 10001
Era -1 -1 -1
Tail 9000 -1 14000
Param 14000 -1 0
Call 57 -1 10002
+ 10001 10002 10003
Assign 10003 -1 10000
Ret 10000 -1 -1
Emp 9000 -1 12000
GotoF 12000 -1 73
Assign 15000 -1 10000
Goto 15000 -1 83
Era -1 -1 -1
Head 9000 -1 14000
Param 14000 -1 0
Call 57 -1 10001
Era -1 -1 -1
Tail 9000 -1 14001
Param 14001 -1 0
Call 69 -1 10002
+ 10001 10002 10003
Assign 10003 -1 10000
Ret 10000 -1 -1
Era -1 -1 -1
Lst 5 -1 3
Lst 1 -1 3
PaLst 15001 -1 0
PaLst 15002 -1 1
PaLst 15003 -1 2
GeLst -1 -1 14003
PaLst 14003 -1 0
Lst 1 -1 3
PaLst 15004 -1 0
PaLst 15005 -1 1
PaLst 15006 -1 2
GeLst -1 -1 14004
PaLst 14004 -1 1
Lst 1 -1 3
PaLst 15007 -1 0
PaLst 15008 -1 1
PaLst 15009 -1 2
GeLst -1 -1 14005
PaLst 14005 -1 2
GeLst -1 -1 14002
Param 14002 -1 0
Call 69 -1 10004
Print -1 -1 10004
10
6 15006
7 15007
0 15000
1 15001
2 15002
3 15003
4 15004
5 15005
8 15008
9 15009
```

Test_Matrix Output

```
vsapiens@Usuarios-MacBook-Pro:~/Documents/lambdish-compiler/examples$ rlamb test_matrix.obj
45
```

6) Listados del Proyecto

Dado el tipo de lenguaje, el patrón de diseño para el lenguaje de GO es la generación de paquetes o módulos que contienen cierta cantidad de archivos. A continuación se listan las carpetas por orden alfabético del proyecto, dentro de esta descripción se podrá encontrar detalles de cada una de las carpetas o módulos generados, dado el patrón del diseño.

Ast

Los siguientes fragmentos de código se utilizan para generar los nodos del AST para el programa, las funciones, el cuerpo de las funciones, llamada de las funciones y las funciones tipo lambda.

```
// NewProgram creates a new Program node which acts as
    the root of the tree
func    NewProgram(functions,    call    interface{})
    (*Program, error) {
    fs, ok := functions.([]*Function)
    if !ok {
            return  nil,  errutil.NewNoPosf("Invalid
    type for functions. Expected []*Function")
    }

    c, ok := call.(*FunctionCall)
    if !ok {
            return  nil,  errutil.NewNoPosf("Invalid
    type for function call. Expected *FunctionCall")
    }
    return &Program{fs, c}, nil
}
// NewFunction creates a new Function node which acts
    as the children of the tree
func    NewFunction(id,    params,    typ,    statement
    interface{}) (*Function, error) {
    i, ok := id.(*token.Token)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    id. Expected token")
    }

    d := string(i.Lit)

    p, ok := params.([]*dir.VarEntry)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    params. Expected []*dir.VarEntry")
    }

    t, ok := typ.(*types.LambdishType)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    typ. Expected *types.LambdishType")
    }

    s, ok := statement.(Statement)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    statement. Expected Statement")
    }

    f := &Function{d, "", p, t, s, i}
    f.CreateKey()

    return f, nil
}
// NewStatement creates a new Statement node which
    acts as the children of the function, which is
    the body of the function
func   NewStatement(value   interface{})   (Statement,
    error) {
    // Check if the value is an id and cast it fist to
    a token
    if t, ok := value.(*token.Token); ok {
      return &Id{string(t.Lit), t}, nil
      // If not, cast the value to a statement
    interface
    } else if s, ok := value.(Statement); ok {
      return s, nil
    }

      return nil, errutil.NewNoPosf("Invalid type for
    statement. Expected Statement interface got %v",
    value)
}
// NewFunctionCall creates a new FunctionCall node
    which acts as the children of the program or
    function, which is the function call
func     NewFunctionCall(id,     args     interface{})
    (*FunctionCall, error) {
    i, ok := id.(Statement)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    id. Expected statement")
    }

    a, ok := args.([]Statement)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    args. Expected []Statement, got %v", args)
   }
    return &FunctionCall{i, a}, nil
}
// NewLambda creates a new NewLambda node which acts
    as a child of a Function Node, which is the
    lambda declaration
func   NewLambda(tok,   params,   retval,   statement
    interface{}) (*Lambda, error) {
    tk, ok := tok.(*token.Token)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    paratkms. Expected *token.Token")
    }

    p, ok := params.([]*dir.VarEntry)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    params. Expected []*dir.VarEntry]")
    }

    s, ok := statement.(Statement)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    params. Expected Statement")
    }

    t, ok := retval.(*types.LambdishType)
    if !ok {
      return nil, errutil.NewNoPosf("Invalid type for
    type. Expected *types.LambdishType")
    }

    return &Lambda{p, s, t, "", tk}, nil
```

Cmd

Para este paquete se tienen 2 submódulos que representan los comandos de instalación para compilar el programa y ejecutarlo. Las subcarpetas contienen la función principal para hacer la compilación que llama a otros submódulos que se explican más adelante.

Clamb, Rlamb

-----Clamb.go

```
//usage The correct use of the clamb command
func usage() {
     fmt.Printf("Usage: clamb <lambdish source
    file>\n")
}

//readFile Reads the argument which is the file
    and maps it to a buffer to read the .lsh
    file
func readFile(path string) ([]byte, error) {
    file, err := os.Open(path)
    if err != nil {
     return nil, err
    }

    defer file.Close()

    fileinfo, err := file.Stat()
    if err != nil {
     return nil, err
    }

    filesize := fileinfo.Size()
    buffer := make([]byte, filesize)

    _, err = file.Read(buffer)
    if err != nil {
     return nil, err
    }

    return buffer, nil
}

//compile Function that compiles the program
    calling the new parser, lexer, ast,
    semantic check, code gen and generate the
    .obj file
func compile(file string) error {
    p := parser.NewParser()
    input, err := readFile(file)

    if err != nil {
     return err
    }

    s := lexer.NewLexer(input)
    pro, err := p.Parse(s)

    if err != nil {
     return err
    }

    program, ok := pro.(*ast.Program)
    if !ok {
       return   errutil.NewNoPos("Cannot   cast
    program")
    }

    funcdir, err := sem.SemanticCheck(program)
    if err != nil {
     return err
    }

                   gen,     vm,     err     :=
    ic.GenerateIntermediateCode(program,
    funcdir)
    if err != nil {
     return err
    }

    fileroot := strings.Split(file, ".")

    return gen.CreateFile(fileroot[0], vm)
```

-----Rlamb.go

```
//usage The correct use of the rlamb command
func usage() {
      fmt.Printf("Usage:  rlamb  <lambdish  object
file>\n")
}

//run  Function  that  initializes  the  new  Virtual
Machine and loads the program file and prints the
output of the program
func run(file string) error {
    machine := vm.NewVirtualMachine()

    err := machine.LoadProgram(file)
    if err != nil {
         return err
    }

    err = machine.Run()
    if err != nil {
         return err
    }

    return nil
}
```

Dir

Dentro de este paquete se tienen las inicializaciones de las estructuras principales como lo son el stack de entradas de funciones, directorio de funciones y directorio de variables.

-----FuncDir.go

```
// NewFuncEntry creates a new FuncEntry struct

    func    NewFuncEntry(id    string,    returnval
        *types.LambdishType,                params
        []*types.LambdishType,              vardir
        *VarDirectory) *FuncEntry {
          return &FuncEntry{id, returnval, params,
        vardir,       make([]*FuncEntry,       0),
        mem.Address(-1), 0}
    }

    //NewFuncDirectory  initializes  a  new  empty
        function directory

    func NewFuncDirectory() *FuncDirectory {
                                              return
        &FuncDirectory{make(map[string]*FuncEntry)
        }
    }

    //MainFuncEntry Initialization of the function
        directory  with  the  initial  parameters  of
        the main program

    func MainFuncEntry() *FuncEntry {
                    return    &FuncEntry{"main",
        types.NewDataLambdishType(types.Num,    0),
        make([]*types.LambdishType,             0),
        NewVarDirectory(),  make([]*FuncEntry,  0),
        mem.Address(-1), 0}
    }

    //  Add  function adds a new funcentry to the
        table. If the addition is successful
    //   the   function   returns   true   and   false
        otherwise.

    func (fd *FuncDirectory) Add(e *FuncEntry) bool
        {

        _, ok := fd.table[e.Key()]
        if !ok {
         fd.table[e.Key()] = e
        }
        return !ok
    }
```

-----Vardir.go

```
    //NewVarDirectory  New  directory  of  variables
        that stores the var entry
func NewVarDirectory() *VarDirectory {
                                              return
    &VarDirectory{make(map[string]*VarEntry)}
}

//NewVarEntry  Initialization  of  one  entry  of
    the variable with its attributes
func       NewVarEntry(id       string,       t
    *types.LambdishType, tok *token.Token, pos
    int) *VarEntry {
    return &VarEntry{id, t, tok, 0, pos}
}

//Add Add a varentry to the directory variables
    using the toString function as key
func (vd *VarDirectory) Add(e *VarEntry) bool {

    _, ok := vd.table[e.String()]
    if !ok {
     vd.table[e.String()] = e
    }
    return !ok
}
```

-----Festack.go

```
// node is a data container
type node struct {
    val  *FuncEntry
    next *node
}

//  FuncEntryStack  implements  a  stack  for  the
    FuncEntry data type
type FuncEntryStack struct {
    head *node
}
//  Push  adds  an  element  to  the  top  of  the
    container
func (s *FuncEntryStack) Push(val *FuncEntry) {
    newHead := &node{val, s.head}
    s.head = newHead
}

//NewFuncEntryStack  Creation  of  the  func  entry
    stack
func NewFuncEntryStack() *FuncEntryStack {
    return &FuncEntryStack{nil}
}
```

Grammar

En este módulo se explica la gramática y las acciones de la sintaxis para construir el AST, además se incluye un módulo de prueba para visualizar el árbol.

-----Lambdish.ebnf

```
_digit : '0'-'9' ;
_alpha : 'a'-'z' | 'A'-'Z';
_id : _alpha {(_digit | _alpha)};
_integer : _digit {_digit};
_float : _digit {_digit} '.' _digit {_digit};
_string : '"' {(_digit | _alpha | ' ')} '"';
_true : 't' 'r' 'u' 'e';
_false : 'f' 'a' 'l' 's' 'e';

!comment : '/' '/' { . } '\n';

!ws : ' ' | '\t' | '\v' | '\f' | '\r' | '\n' ;

operations : '+' | '-' | '*' | '/' | '%';

relop : '<' | '>' | '!';

number : ['-'] (_integer | _float);
charac : '\'' (_alpha | _digit | ' ') '\'';
string : _string;
boolean : _true | _false;
id : _id;

<< import (
            "github.com/Loptt/lambdish-compiler/ast"
            "github.com/Loptt/lambdish-compiler/dir"
            "github.com/Loptt/lambdish-compiler/types") >>

Program
            : Functions Statement       << ast.NewProgram($0, $1) >>
            ;

Functions
            : Function Functions    << ast.AppendFunctionList($0, $1) >>
            | Function               << ast.NewFunctionList($0) >>
            | empty                  << make([]*ast.Function, 0), nil >>
            ;

Function
            : "func" id "::" Params "=>" Type "(" Statement ")"  <<
ast.NewFunction($1, $3, $5, $7) >>
            ;

Params :
            Type id "," Params    << ast.AppendParamsList($0, $1, $3) >>
            | Type id                 << ast.NewParamsList($0, $1) >>
            | empty                  << make([]*dir.VarEntry, 0), nil >>
            ;

Type
            : BasicType                         << $0, nil >>
            | "(" FuncTypes "=>" Type ")<< ast.NewFunctionType($1, $3)>>
            | "[" Type "]"                      << ast.AppendType($1) >>
            ;

BasicType
            : "num"                             << ast.NewType($0) >>
            | "bool"                            << ast.NewType($0) >>
            | "char"                            << ast.NewType($0) >>
            ;

FuncTypes
            : Type "," FuncTypes    << ast.AppendFuncTypeList($0, $2) >>
            | Type                    << ast.NewFuncTypeList($0) >>
            | empty           << make([]*types.LambdishType, 0 ), nil >>
            ;

Statement
            : id                        << ast.NewStatement($0) >>
            | Constant                   << ast.NewStatement($0) >>
            | Lambda                     << ast.NewStatement($0) >>
            | FunctionCall               << ast.NewStatement($0) >>
            ;

FunctionCall
            : Statement "(" Args ")"  << ast.NewFunctionCall($0, $2) >>
| operations "(" Args ")"<< ast.NewFunctionReservedCall($0, $2) >>
            | relop "("Args ")"<< ast.NewFunctionReservedCall($0, $2) >>
            ;

Lambda
            : "("  "#"  Params  "=>"  Type  "("  Statement  ")"  ")"  <<
ast.NewLambda($1,$2,$4,$6) >>
            ;

Args
            : Statement "," Args    << ast.AppendStatementList($0,$2) >>
            | Statement              << ast.NewStatementList($0) >>
            | empty                  << make([]ast.Statement, 0), nil >>
            ;

Constant
            : boolean                 << ast.NewConstantBool($0) >>
            | number                   << ast.NewConstantNum($0) >>
            | charac                   << ast.NewConstantChar($0) >>
            | string                  << ast.AppendStringConstant($0) >>
            | "[" ConstantArgs "]"      << ast.AppendConstant($0, $1) >>
            | "[" Type "]"         << ast.AppendEmptyConstant($0, $1) >>
            ;

ConstantArgs
            : Statement "," Args    << ast.AppendStatementList($0,$2) >>
            |  Statement              << ast.NewStatementList($0) >>
            ;
```

-----Grammar_test.go

```
//TestGrammar Function that tests the grammar from the
lexer and parser
func TestGrammar(t *testing.T) {
    p := parser.NewParser()
    tests := []string{
           "tests/test6.lsh",
    }

    for _, test := range tests {
           input, err := readFile(test)

           if err != nil {
                      t.Fatalf("Error   reading   file   %s",
test)
           }

           s := lexer.NewLexer(input)
           program, errtest := p.Parse(s)

           spew.Dump(program)

           if errtest != nil {
                     t.Errorf("%s: %v", test, errtest)
           }
    }
}
```

IC

Este módulo se encarga de generar los cuádruplos correspondientes al programa dado. Realiza un escaneo del código a partir del AST y genera el código intermedio con la ayuda de un contador y dos stacks, uno para las direcciones y otro para los saltos.

-----generator.go

```
// Generator ...
type Generator struct {
        jumpStack       *AddressStack
        addrStack       *AddressStack
        icounter        int
        pcounter        int
        quads           []*quad.Quadruple
        pendingFuncAddr map[int]string
        pendingEraSize  map[int]string
}
// Generate creates a new quadruple with the given parameters
func (g *Generator) Generate(op quad.Operation, a1, a2, r mem.Address) {
        g.quads          =          append(g.quads, quad.NewQuadruple(op, a1, a2, r))
        g.icounter++
}

//PushToAddrStack adds an address to the address stack
func (g *Generator) PushToAddrStack(a mem.Address) {
        g.addrStack.Push(a)
}

//GetFromAddrStack pops and returns the top address of the stack
func (g *Generator) GetFromAddrStack() mem.Address {
        val := g.addrStack.Top()
        g.addrStack.Pop()
        return val
}
```

-----generatecode.go

```
// generateCodeProgram starts the code generation for the whole program
func generateCodeProgram(program *ast.Program, ctx *GenerationContext) error {

        // This statement adds the initial goto to the pending jumps

ctx.gen.AddPendingFuncAddr(ctx.gen.ICounter(), "main")

        // The first instruction we generate jumps the execution
        ctx.gen.Generate(quad.Goto, mem.Address(-1), mem.Address(-1), mem.Address(-1))

        for      _,      function      :=      range program.Functions() {
                if              err             := generateCodeFunction(function, ctx); err != nil {
                        return err
                }
        }

        fes := dir.NewFuncEntryStack()
        fes.Push(ctx.funcdir.Get("main"))


ctx.funcdir.Get("main").SetLocation(ctx.gen.ICounter())

        if                  err                 := generateCodeFunctionCall(program.Call(), fes, ctx); err != nil {
                return err
        }

        ctx.gen.FillPendingFuncAddr(ctx.funcdir)
        ctx.gen.Generate(quad.Print, mem.Address(-1),                   mem.Address(-1), ctx.gen.GetFromAddrStack())

        return nil
}

// generateCodeFunction takes a function node and generate its corresponding quadruples.
// Function nodes always end with a RET quad
func  generateCodeFunction(function  *ast.Function, ctx *GenerationContext) error {

        ctx.vm.ResetTemp()

        fe := ctx.funcdir.Get(function.Key())

        fes := dir.NewFuncEntryStack()
        fes.Push(fe)

        fe.SetLocation(ctx.gen.ICounter())

        if                  err                 := generateCodeStatement(function.Statement(),    fes, ctx); err != nil {
                return err
        }

        addr := ctx.gen.GetFromAddrStack()

        ctx.gen.Generate(quad.Ret,            addr, mem.Address(-1), mem.Address(-1))

        return nil
}

// generateCodeStatemet checks the type of the statement and calls the specific function to generate the code
```

```
func generateCodeStatement(statement ast.Statement,
fes  *dir.FuncEntryStack,  ctx  *GenerationContext)
error {
        if id, ok := statement.(*ast.Id); ok {
                return   generateCodeID(id,   fes,
ctx)
        }     else     if     fcall,     ok     :=
statement.(*ast.FunctionCall); ok {
                return
generateCodeFunctionCall(fcall, fes, ctx)
        }     else     if     lambda,     ok     :=
statement.(*ast.Lambda); ok {
                return  generateCodeLambda(lambda,
fes, ctx)
        }      else      if      cl,      ok      :=
statement.(*ast.ConstantList); ok {
                return
generateCodeConstantList(cl, fes, ctx)
        }      else      if      cv,      ok      :=
statement.(*ast.ConstantValue); ok {
                return
generateCodeConstantValue(cv, fes, ctx)
        }

        return errutil.NewNoPosf("Statement cannot
be casted to any valid form")
}
```

Integration

Este módulo hace la integración desde el análisis de léxico hasta la generación de código intermedio, donde se visualiza todo el contenido a través de la terminal. Este sería considerado como un hito importante del proyecto ya que determina el fin de la fase de compilación sin generar el archivo de los objetos y sin considerar la memoria virtual.

```
        //TestGenerateIntermediateCode     Function
that compiles the program and integrates it
        func         TestGenerateIntermediateCode(t
*testing.T) {
            p := parser.NewParser()
            tests := []string{
                 "test1.lsh",
            }

            for _, test := range tests {
                 input, err := readFile(test)

                 if err != nil {
                          t.Fatalf("Error  reading
file %s", test)
                 }

                 s := lexer.NewLexer(input)
                 pro, err := p.Parse(s)
                 if err != nil {
                          t.Fatalf("%s: %v", test,
err)
                 }

                 program, ok := pro.(*ast.Program)
                 if !ok {
                          t.Fatalf("Cannot cast to
Program")
                 }

                                fmt.Printf("======PROGRAM
AST=======\n")
                 spew.Dump(program)

                       funcdir,       err       :=
sem.SemanticCheck(program)
                 if err != nil {
                           t.Fatalf("Error   from
semantic: %v", err)
                 }

                      fmt.Printf("\n======FUNCTION
DIRECTORY=======\n")
                 spew.Dump(funcdir)

                     gen,      _,      err      :=
ic.GenerateIntermediateCode(program, funcdir)
                 if err != nil {
                           t.Fatalf("Error   from
generate code: %v", err)
                 }

                  fmt.Printf("\n======INTERMEDIATE
CODE=======\n")
                 fmt.Printf("%s\n", gen)
             }
```

Mem

En este módulo se manejo lo que es la memoria virtual, se dividió en segmentos y se crearon marcadores dependiendo del contexto de la memoria. Se dividió en 2 archivos que contienen la representación de la memoria.

```
//Type of the address of the memory is an int
type Address int

//Constants that mark the start of the context
const Globalstart = 0
const Localstart = 5000
const Tempstart = 10000
const Constantstart = 15000
const Scopestart = 20000

//Constants that set the offset of the segment
const NumOffset = 0
const CharOffset = 1000
const BoolOffset = 2000
const FunctionOffset = 3000
const ListOffset = 4000

//Constant that defines de size of the segment
const segmentsize = 1000

//String  If  the  address  is  not  in  range  or  not
useful it must be set to -1 to be declared invalid
func (a Address) String() string {
    if a < 0 {
          return "-1"
    }

    return fmt.Sprintf("%d", a)
}
//Struct  that  defines  what  the  Virtual  memory
containts
type VirtualMemory struct {
    globalnumcount       int
    globalcharcount      int
    globalboolcount      int
    globalfunctioncount int
```

```
    globallistcount      int

    localnumcount        int
    localcharcount       int
    localboolcount       int
    localfunctioncount int
    locallistcount       int

    tempnumcount         int
    tempcharcount        int
    tempboolcount        int
    tempfunctioncount int
    templistcount        int

    constantnumcount     int
    constantcharcount    int
    constantboolcount    int
    constantfunctioncount int
    constantlistcount    int

    scopenumcount        int
    scopecharcount       int
    scopeboolcount       int
    scopefunctioncount int
    scopelistcount       int

    constantmap map[string]int
}
//GetNextLocal Receives the type and determines the next local variable available in the scope to assign it
func (vm *VirtualMemory) GetNextLocal(t *types.LambdishType) (Address, error) {
    switch t.String() {
    // Num
    case "1":
         if vm.localnumcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: local variables for numbers exceeded.")
         }
          result := vm.localnumcount + NumOffset + Localstart
         vm.localnumcount++
         return Address(result), nil
    // Char
    case "2":
         if vm.localcharcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: local variables for chars exceeded.")
         }
          result := vm.localcharcount + CharOffset + Localstart
         vm.localcharcount++
         return Address(result), nil
    // Bool
    case "3":
         if vm.localboolcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: local variables for bools exceeded.")
         }
          result := vm.localboolcount + BoolOffset + Localstart
         vm.localboolcount++
         return Address(result), nil
    }

    if t.List() > 0 {
         if vm.locallistcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: local variables for lists exceeded.")
         }
          result := vm.locallistcount + ListOffset + Localstart
         vm.locallistcount++
         return Address(result), nil
    }

    if t.Function() {
         if vm.localfunctioncount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: local variables for function exceeded.")
         }
           result := vm.localfunctioncount + FunctionOffset + Localstart
         vm.localfunctioncount++
         return Address(result), nil
    }

     return Address(-1), errutil.NewNoPosf("Error: variable type not identified.")
}
//GetNextTemp Receives the type and determines the next temp variable available in the scope to assign it
func (vm *VirtualMemory) GetNextTemp(t *types.LambdishType) (Address, error) {
    switch t.String() {
    // Num
    case "1":
         if vm.tempnumcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: temp variables for numbers exceeded.")
         }
          result := vm.tempnumcount + NumOffset + Tempstart
         vm.tempnumcount++
         return Address(result), nil
    // Char
    case "2":
         if vm.tempcharcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: temp variables for chars exceeded.")
         }
         result := vm.tempcharcount + CharOffset + Tempstart
         vm.tempcharcount++
         return Address(result), nil
    // Bool
    case "3":
         if vm.tempboolcount >= segmentsize {
                        return       Address(-1), errutil.NewNoPosf("Error: temp variables for bools exceeded.")
         }
```

```
         result := vm.tempboolcount + BoolOffset +
Tempstart
         vm.tempboolcount++
         return Address(result), nil
    }

    if t.List() > 0 {
         if vm.templistcount >= segmentsize {
                        return        Address(-1),
errutil.NewNoPosf("Error: temp variables for lists
exceeded.")
         }
         result := vm.templistcount + ListOffset +
Tempstart
         vm.templistcount++
         return Address(result), nil
    }

    if t.Function() {
         if vm.tempfunctioncount >= segmentsize {
                        return        Address(-1),
errutil.NewNoPosf("Error:   temp   variables   for
functions exceeded.")
         }
           result   :=   vm.tempfunctioncount   +
FunctionOffset + Tempstart
         vm.tempfunctioncount++
         return Address(result), nil
    }

     return Address(-1), errutil.NewNoPosf("Error:
variable type not identified.")
}
//getNextConstant Receives the type and determines
the next constant variable available in the scope to
assign it
    func   (vm   *VirtualMemory)   getNextConstant(t
        *types.LambdishType) (Address, error) {
       switch t.String() {
       // Num
       case "1":
           if vm.constantnumcount >= segmentsize {
                              return  Address(-1),
        errutil.NewNoPosf("Error:         constant
        variables for numbers exceeded.")
           }
                 result  :=  vm.constantnumcount  +
        NumOffset + Constantstart
           vm.constantnumcount++
           return Address(result), nil
       // Char
       case "2":
           if vm.constantcharcount >= segmentsize {
                              return  Address(-1),
        errutil.NewNoPosf("Error:         constant
        variables for chars exceeded.")
           }
                 result  :=  vm.constantcharcount  +
        CharOffset + Constantstart
           vm.constantcharcount++
           return Address(result), nil
       // Bool
       case "3":
           if vm.constantboolcount >= segmentsize {
                              return  Address(-1),
        errutil.NewNoPosf("Error:         constant
        variables for bools exceeded.")
           }
                 result  :=  vm.constantboolcount  +
        BoolOffset + Constantstart
           vm.constantboolcount++
           return Address(result), nil
       }

       if t.List() > 0 {
           if vm.constantlistcount >= segmentsize {
                              return  Address(-1),
        errutil.NewNoPosf("Error:         constant
        variables for lists exceeded.")
           }
                 result  :=  vm.constantlistcount  +
        ListOffset + Constantstart
           vm.constantlistcount++
           return Address(result), nil
       }

       if t.Function() {
                  if  vm.constantfunctioncount  >=
        segmentsize {
                              return  Address(-1),
        errutil.NewNoPosf("Error:         constant
        variables for functions exceeded.")
           }
              result := vm.constantfunctioncount +
        FunctionOffset + Constantstart
           vm.constantfunctioncount++
           return Address(result), nil
       }

                           return       Address(-1),
        errutil.NewNoPosf("Error:   variable   type
        not identified.")
    }

    //GetConstantAddress gets the address of the
        constant from the map of constants
    func  (vm  *VirtualMemory)  GetConstantAddress(c
        string) Address {
        a, ok := vm.constantmap[c]
        if !ok {
         return Address(-1)
        }

        return Address(a)
    }
```

Quad

En este módulo se definen los módulos que permiten definir las operaciones del sistema que se generar al momento de generar el código. La cantidad de cuádruplos se mapean a un string para poder ser manejados al momento de obtener el código. Se hizo una estructura de cuádruplo que contenía el nombre, los 2 operandos y el último era para guardar el resultado.

```
// Operation The operation or instruction of the
system is a number
type Operation int

func (o Operation) String() string {
    switch o {
    case Add:
          return "+"
    case Sub:
          return "-"
    case Mult:
          return "*"
    case Div:
          return "/"
    case Mod:
          return "%"
    case Lt:
          return "<"
    case Gt:
          return ">"
    case Equal:
          return "Equal"
    case And:
          return "And"
    case Or:
          return "Or"
    case Not:
          return "!"
    case GotoT:
          return "GotoT"
    case GotoF:
          return "GotoF"
    case Goto:
          return "Goto"
    case Ret:
          return "Ret"
    case Call:
          return "Call"
    case Era:
          return "Era"
    case Param:
          return "Param"
    case Emp:
          return "Emp"
    case Head:
          return "Head"
    case Tail:
          return "Tail"
    case Ins:
          return "Ins"
    case App:
          return "App"
    case GeLst:
          return "GeLst"
    case Lst:
          return "Lst"
    case PaLst:
          return "PaLst"
    case Print:
          return "Print"
    case Assign:
          return "Assign"
    }

    return ""
}
/*
     Structure: Defines the main structure of code
generation
      It contains the operation, 2 operands that
holds the addresses and the third one
    is usually the result of the operation.
 */
type Quadruple struct {
    op Operation
    a1 mem.Address
    a2 mem.Address
    r  mem.Address
}
//String is used in the creation of the file object
that has the quadruples
func (q Quadruple) String() string {
     return fmt.Sprintf("%s %s %s %s", q.op, q.a1,
q.a2, q.r)
}

//NewQuadruple  Creates  new  quadruple  with  the
addresses given and the number of operation
func   NewQuadruple(op   Operation,   a1,   a2,   r
mem.Address) *Quadruple {
    return &Quadruple{op, a1, a2, r}
}
```

Sem

```
-----sem.go
// SemanticCheck calls the 3 main functions that perform the semantic analysis and
// reports any errors
func SemanticCheck(program *ast.Program) (*dir.FuncDirectory, error) {
	funcdir := dir.NewFuncDirectory()
	semcube := NewSemanticCube()

	// Build the function directory and their corresponding Var directiories
	// Errors to check:
	//  * If a function is declared twice
	//    * If two parameters in the same function have the same id
	//
	if err := buildFuncDirProgram(program, funcdir); err != nil {
		return nil, err
	}

	// Check the scope of function calls and variable uses.
	// Errors to check:
	//   * If a function is called that does not exist
	//  * If a variable is used and it has not been declared in the parameters
	//
	if err := scopeCheckProgram(program, funcdir, semcube); err != nil {
		return funcdir, err
	}

	// Check type cohesion
	// Errors to check:
	//    * If a function is called and no function in the funcdir match the argument types
	//   * If the value in the statement of a function does not match its return value
	//    * Illegal use of use of built-in functions
	//        - Arithmetic operators: +, -, *, /,%
	//         - Relational operators: < , >, equal
	//        - Logical operatios: (only to be used with bools) and, or, !
	//  * If statements:
	//       - Check that first argument is of type bool
	//         - Check that second and third arguments are of the same type
	//       - The type of the second and third argument will define the type of the if statement
	//   * To check whether a combination of params for an operator is valid, the semantic cube must be consulted
	//
	if err := typeCheckProgram(program, funcdir, semcube); err != nil {
		return funcdir, err
	}
	return funcdir, nil
}
```

-----funccheck.go

```
//buildFuncDirProgram receives the program and the function directory to start building the funcdir
func buildFuncDirProgram(program *ast.Program, funcdir *dir.FuncDirectory) error {
	for _, f := range program.Functions() {
		if err := buildFuncDirFunction(f, funcdir); err != nil {
			return err
		}
	}

	if err := buildFuncDirCall(program.Call(), funcdir); err != nil {
		return err
	}

	return nil
}

//buildFuncDirFunction creates a new FuncEntry for the function and adds it to the directory
func buildFuncDirFunction(function *ast.Function, funcdir *dir.FuncDirectory) error {
	id := function.Id()
	t := function.Type()
	vardir := dir.NewVarDirectory()
	params := make([]*types.LambdishType, 0)

	if funcdir.Exists(id) {
		return errutil.NewNoPosf("%+v: Redeclaration of function %s", function.Token(), id)
	}

	if idIsReserved(id) {
		return errutil.NewNoPosf("%+v: Cannot declare a function with reserved keyword %s", function.Token(), id)
	}

	for _, p := range function.Params() {
		params = append(params, p.Type())
		if err := buildVarDirFunction(p, vardir); err != nil {
			return err
		}
	}

	if kw, ok := checkVarDirReserved(vardir); !ok {
		return errutil.NewNoPosf("%+v: Cannot declare variable with reserved keyword %s", function.Token(), kw)
	}

	fe := dir.NewFuncEntry(id, t, params, vardir)

	if ok := funcdir.Add(fe); !ok {
```

```
                return     errutil.NewNoPosf("%+v:
Invalid Function. This Function already exists.",
function.Token())
        }

        if                  err                 :=
buildFuncDirStatement(function.Statement(),     fe);
err != nil {
                return err
        }

        return nil
}
```

-----scopecheck.go

```
//scopeCheckProgram starts the scope checking for
the whole program
func     scopeCheckProgram(program     *ast.Program,
funcdir  *dir.FuncDirectory,  semcube  *SemanticCube)
error {
        for _, f := range program.Functions() {
                if  err  :=  scopeCheckFunction(f,
funcdir, semcube); err != nil {
                        return err
                }
        }

        fes := dir.NewFuncEntryStack()
        fes.Push(funcdir.Get("main"))
        if                  err                 :=
scopeCheckFunctionCall(program.Call(),          fes,
funcdir, semcube); err != nil {
                return err
        }

        return nil
}

//scopeCheckFunction verifies the function is added
to the func directory and the checks its statement
func   scopeCheckFunction(function   *ast.Function,
funcdir  *dir.FuncDirectory,  semcube  *SemanticCube)
error {
        fe := funcdir.Get(function.Key())
        if fe == nil {
                return     errutil.NewNoPosf("%+v:
Function  entry  %+v  not  found  in  FuncDirectory",
function.Token(), fe)
        }

        fes := dir.NewFuncEntryStack()
        fes.Push(fe)

        if                  err                 :=
scopeCheckStatement(function.Statement(),       fes,
funcdir, semcube); err != nil {
                return err
        }

        fes.Pop()

        return nil
}
```

```
//scopeCheckStatement   calls   the   corresponding
function to check the statement depending on the
type
func  scopeCheckStatement(statement  ast.Statement,
fes           *dir.FuncEntryStack,           funcdir
*dir.FuncDirectory, semcube *SemanticCube) error {
        if id, ok := statement.(*ast.Id); ok {
                return    scopeCheckID(id,    fes,
funcdir)
        }     else     if     fcall,     ok     :=
statement.(*ast.FunctionCall); ok {
                return
scopeCheckFunctionCall(fcall,     fes,     funcdir,
semcube)
        }     else     if     lambda,     ok     :=
statement.(*ast.Lambda); ok {
                return    scopeCheckLambda(lambda,
fes, funcdir, semcube)
        }      else      if      cl,      ok      :=
statement.(*ast.ConstantList); ok {
                return  scopeCheckConstantList(cl,
fes, funcdir, semcube)
        }      else      if      _,      ok      :=
statement.(*ast.ConstantValue); ok {
                return nil
        }

        return errutil.NewNoPosf("Statement cannot
be casted to any valid form")
}
```

-----scopeutil.go

```
//  idExistsInFuncStack  checks  if  the  given  id
exists at any parameter declaration in the stack of
// FuncEntry. If it exist, it means that the id has
been declared and is in scope
func     idExistsInFuncStack(id     *ast.Id,     fes
*dir.FuncEntryStack) bool {
        fescpy := *fes
        for !fescpy.Empty() {
                fe := fescpy.Top()
                if fe.VarDir().Exists(id.String())
{
                        return true
                }

                fescpy.Pop()
        }

        return false
}

// idExistsInFuncDir checks if the given id is the
name of a function declared in the
// function directory.
func    idExistsInFuncDir(id    *ast.Id,    funcdir
*dir.FuncDirectory) bool {
        return funcdir.Exists(id.String())
}
```

-----typecheck.go

```
//typeCheckProgram starts the type checking in the
whole program
func typeCheckProgram(program *ast.Program, funcdir
*dir.FuncDirectory, semcube *SemanticCube) error {
```

```
        for _, f := range program.Functions() {
                if err := typeCheckFunction(f, funcdir, semcube); err != nil {
                        return err
                }
        }

        fes := dir.NewFuncEntryStack()
        fes.Push(funcdir.Get("main"))
        if err := typeCheckFunctionCall(program.Call(), fes, funcdir, semcube); err != nil {
                return err
        }

        return nil
}

//typeCheckFunction verifies its statement is of the same type of the return type of the function
func typeCheckFunction(function *ast.Function, funcdir *dir.FuncDirectory, semcube *SemanticCube) error {

        fe := funcdir.Get(function.Key())
        if fe == nil {
                return errutil.NewNoPosf("%+v: Cannot get function %s from Func Directory", function.Token(), function.Id())
        }

        rv := fe.ReturnVal()
        fes := dir.NewFuncEntryStack()
        fes.Push(fe)

        statementType, err := GetTypeStatement(function.Statement(), fes, funcdir, semcube)
        if err != nil {
                return err
        }

        if !rv.Equal(statementType) {
                return errutil.NewNoPosf("%+v: Statement type does not match return type in function %s", function.Token(), function.Id())
        }

        if err := typeCheckStatement(function.Statement(), fes, funcdir, semcube); err != nil {
                return err
        }

        fes.Pop()
        return nil
}

//typeCheckStatement performs the type checking depending on the type of the statement
func typeCheckStatement(statement ast.Statement, fes *dir.FuncEntryStack, funcdir *dir.FuncDirectory, semcube *SemanticCube) error {
        if _, ok := statement.(*ast.Id); ok {
                return nil
        } else if fcall, ok := statement.(*ast.FunctionCall); ok {
                return typeCheckFunctionCall(fcall, fes, funcdir, semcube)
        } else if lambda, ok := statement.(*ast.Lambda); ok {
                return typeCheckLambda(lambda, fes, funcdir, semcube)
        } else if cl, ok := statement.(*ast.ConstantList); ok {
                return typeCheckConstantList(cl, fes, funcdir, semcube)
        } else if _, ok := statement.(*ast.ConstantValue); ok {
                return nil
        }

        return errutil.NewNoPosf("Statement cannot be casted to any valid form")
}
```

-----typeutil.go

```
// GetTypeStatement takes a statement interface and tries to find its type by any means possible
func GetTypeStatement(statement ast.Statement, fes *dir.FuncEntryStack, funcdir *dir.FuncDirectory, semcube *SemanticCube) (*types.LambdishType, error) {
        if id, ok := statement.(*ast.Id); ok {
                if t, err := getIDTypeFromFuncStack(id, fes); err == nil {
                        return t, nil
                } else if fe := funcdir.Get(id.String()); fe != nil {
                        return convertFuncEntryToLambdishType(fe), nil
                }
                return nil, errutil.NewNoPosf("%+v: Id %s not declared in local or global scope", id.Token(), id.String())
        } else if fcall, ok := statement.(*ast.FunctionCall); ok {
                return getTypeFunctionCall(fcall, fes, funcdir, semcube)
        } else if cv, ok := statement.(*ast.ConstantValue); ok {
                return cv.Type(), nil
        } else if cl, ok := statement.(*ast.ConstantList); ok {
                return GetTypeConstantList(cl, fes, funcdir, semcube)
        } else if l, ok := statement.(*ast.Lambda); ok {
                return types.NewFuncLambdishType(l.Retval(), l.Params(), 0), nil
        }
        return nil, errutil.NewNoPosf("Statement cannot be casted to any valid form")
}

// argumentsMatchParameters takes a list of arguments and a list of parameters and verifies if they match in type, position, and length
```

```
func                argumentsMatchParameters(fcall
*ast.FunctionCall,    args    []*types.LambdishType,
params          []*types.LambdishType,          fes
*dir.FuncEntryStack,   funcdir   *dir.FuncDirectory,
semcube *SemanticCube) error {
        if len(args) != len(params) {
                return     errutil.NewNoPosf("%+v:
function   expects   %d   arguments,   got   %d",
fcall.Token(), len(params), len(args))
        }

        for i, p := range params {
                if !(p.Equal(args[i])) {
                        return
errutil.NewNoPosf("%+v: Function call arguments do
not match its parameters", fcall.Token())
                }
        }

        return nil
}

// GetTypesFromArgs takes a list of arguments and
returns a list of LambdishType structs
func  GetTypesFromArgs(args  []ast.Statement,  fes
*dir.FuncEntryStack,   funcdir   *dir.FuncDirectory,
semcube    *SemanticCube)    ([]*types.LambdishType,
error) {
        ts := make([]*types.LambdishType, 0)

        for _, arg := range args {
                if t, err := GetTypeStatement(arg,
fes, funcdir, semcube); err == nil {
                        ts = append(ts, t)
                } else {
                        return nil, err
                }
        }

        return ts, nil
}
```

-----semanitccube.go

```
// SemanticCube represents the semantic cube as a
map of a key to its result type
type SemanticCube struct {
        operations map[string]types.BasicType
}

//  NewSemanticCube  creates  a  new  semantic  cube
struct
func NewSemanticCube() *SemanticCube {

        return &SemanticCube{
                map[string]types.BasicType{
                        //Arithmetical Operators
                        "+@11": types.Num,
                        "-@11": types.Num,
                        "/@11": types.Num,
                        "*@11": types.Num,
                        "%@11": types.Num,
                        //Relational Operators
                        "<@11":     types.Bool,
                        ">@11":     types.Bool,
                        "Equal@11": types.Bool,
                        "Equal@22": types.Bool,
                        "Equal@33": types.Bool,
                        //Logical Operators
                        "And@33": types.Bool,
                        "Or@33":  types.Bool,
                        "!@3":    types.Bool,
                },
        }
}

// Get  takes  a  key  a  checks  if  it  exists  in  the
semantic  cube.  If  it  does,  it  returns  the  result
type
func    (c    *SemanticCube)    Get(key    string)
(types.BasicType, bool) {
        typ, ok := c.operations[key]
        if !ok {
                return types.Null, false
        }
        return typ, true
}
```

Types

Este módulo define las estructuras que se utilizan en todo el programa para representar los tipos de dato de Lambdish como lo son num,char,bool,list y func.

```
// BasicType indicates the three elemental types in the Lambdish language
//
//  Num: any integer or floating point number, either positive or negative
// Char: any character representable as an ascii value
// Bool: a boolean value that can only be true or false
// Null: It is used in the LambdishType struct to indicate that that type is a function
// type and thus the BasicType should not be used
type BasicType int

// LambdishType represents any type on the lambdish language.
// A type in the language can be either a basic type (num, bool, char), or a function type.
//
//  -  In  the  case  of  the  function  type,  the following should be set
//        -function: true
//         -params:  non  null  (might  be  empty anyways)
//        -basic: NULL
//
// - In the case of a basic type, the following should be set
//        -function: false
//        -params: null
//        -basic: non null, the corresponing type
//
// Additional to this, the type might represent a list. If that is the case, all rules set above will
// remain true, and list will be set to a non-zero value, indicating the levels of nesting of the type
// in the list.
//
// For example, a value of list = 1 for a basic type num will consists of a list as following
//        - [num]
//
// And a value of list = 3 for a function type could then look like this
//        - [[[(num, num => bool)]]]
//
// Nontheless external users of this package should only construct Lambdishtype structs using the
// predefined constructors provided below. When a function type is needed, the NewFuncLambdishType
//     function     should     be     called,     and NewDataLambdishType    with    the    basic    type accordingly.
// This will ensure that the values are initialized correctly according to the rules set above.
type LambdishType struct {
    basic        BasicType
    retval   *LambdishType
    params   []*LambdishType
    function bool
    list         int
}
//  String  converts  the  type  to  its  string representation which is used only in the dirfunc package
// to build the composite key of an entry
func (l LambdishType) String() string {
    var builder strings.Builder

    for i := 0; i < l.list; i++ {
         builder.WriteRune('[')
    }

    if l.function {
         builder.WriteRune('(')

         for _, t := range l.params {
                 builder.WriteString(t.String())
         }

         builder.WriteString("=>")
         builder.WriteString(l.retval.String())
         builder.WriteRune(')')
    } else {
         builder.WriteRune(l.basic.convert())
    }

    for i := 0; i < l.list; i++ {
         builder.WriteRune(']')
    }

    return builder.String()
}
//   NewDataLambdishType   Declares   a   new,basic Lambdish type
func  NewDataLambdishType(b  BasicType,  list  int) *LambdishType {
    return &LambdishType{b, nil, nil, false, list}
}

// NewDataLambdishType Declares a new lambdish type as a function
func   NewFuncLambdishType(retval   *LambdishType, params []*LambdishType, list int) *LambdishType {
       return  &LambdishType{Null,  retval,  params, true, list}
}
```

Vm

Este módulo contiene las funciones de la máquina virtual que le permite cargar el programa y ejecutar cada una de las instrucciones definidas. Además contiene las funciones que permiten un manejo adecuado y eficiente de la memoria virtual.

-----vm.go

```
// VirtualMachine contains the necessary attributes
of a virtual machine to execute sequential code,
// manage memory, and call submodules
type VirtualMachine struct {
        ip           int
        quads        []*quad.Quadruple
        mm           *Memory
        ar           *ar.ArStack
        pendingcalls *ar.ArStack
        pendinglists *list.ListStack
        output       interface{}
}


// LoadProgram takes a path to a file and parses
its contents to quadruples and constants
func (vm *VirtualMachine) LoadProgram(path string)
error {
        input, err := readFile(path)
        if err != nil {
                return err
        }

        lines    :=    strings.Split(string(input),
"\n")

        // Get the Amount of instructions from the
top of the file
        iamount, err := strconv.Atoi(lines[0])
        if err != nil {
                return err
        }

        err = vm.loadInstructions(lines[1:(iamount
+ 1)])
        if err != nil {
                return err
        }

        // We get the constant amount at the end
of the instructions
        camount,              err              :=
strconv.Atoi(lines[iamount+1])
        if err != nil {
                return err
        }

        cstart := iamount + 2

        err                                      =
vm.loadConstants(lines[cstart:(cstart + camount)])
        if err != nil {
                return err
        }

        return nil
}


//   executeNextInstruction   indexes   the   next
quadruple with the instruction pointer and
// proceeds to execute that instruction
func (vm *VirtualMachine) executeNextInstruction()
error {
        q := vm.quads[vm.ip]

        //fmt.Printf("%d: Operation: %s\n", vm.ip,
q.Op().String())

        switch q.Op() {
        case quad.Add:
                if err := vm.operationAdd(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Sub:
                if err := vm.operationSub(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Mult:
                if             err             :=
vm.operationMult(q.Lop(),  q.Rop(),  q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Div:
                if err := vm.operationDiv(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Mod:
                if err := vm.operationMod(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.And:
                if err := vm.operationAnd(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Or:
                if  err  := vm.operationOr(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
```

```
        case quad.Not:
                if err := vm.operationNot(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Gt:
                if err := vm.operationGt(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Lt:
                if err := vm.operationLt(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Equal:
                if              err             :=
vm.operationEqual(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Head:
                if              err             :=
vm.operationHead(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Tail:
                if              err             :=
vm.operationTail(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Ins:
                if err := vm.operationIns(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.App:
                if err := vm.operationApp(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Emp:
                if err := vm.operationEmp(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Lst:
                if err := vm.operationLst(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.PaLst:
                if              err             :=
vm.operationPalst(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.GeLst:
                if              err             :=
vm.operationGelst(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Print:
                if              err             :=
vm.operationPrint(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Era:
                if err := vm.operationEra(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
                vm.ip++
        case quad.Param:
                if              err             :=
vm.operationParam(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Call:
                if              err             :=
vm.operationCall(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
        case quad.Ret:
                if err := vm.operationRet(q.Lop(),
q.Rop(), q.R()); err != nil {
                        return err
                }
        case quad.Assign:
                if              err             :=
vm.operationAssign(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
                vm.ip++
        case quad.Goto:
                if              err             :=
vm.operationGoto(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
        case quad.GotoT:
                if              err             :=
vm.operationGotoT(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
        case quad.GotoF:
```

```
                if              err             :=
vm.operationGotoF(q.Lop(), q.Rop(), q.R()); err !=
nil {
                        return err
                }
        }

        return nil
}

// Run starts executing the instructions in the
virtual machine
func (vm *VirtualMachine) Run() error {
        if len(vm.quads) < 1 {
                return       errutil.NewNoPosf("No
instructions to execute")
        }

        // Push the main activation record
        vm.ar.Push(ar.NewActivationRecord())

        for vm.ip < len(vm.quads) {
                if              err             :=
vm.executeNextInstruction(); err != nil {
                        return err
                }
        }

        vm.printOutput()

        return nil
}
```

-----mem.go

```
//MemorySegment represent a single segment in the
Memory struct
type MemorySegment struct {
        num      []float64
        char     []rune
        booleans []bool
        function []int
        list     []*list.ListManager
        base     mem.Address
        name     string
}

// SetValue takes a value and an address and finds
the corresponding position to that address and
// tries to save the given value
func (ms *MemorySegment) SetValue(v interface{},
addr mem.Address) error {
        baseaddr := addr - ms.base

        switch {
        case baseaddr < mem.NumOffset: // Error
                return  errutil.NewNoPosf("Address
out of scope")
        case baseaddr < mem.CharOffset: // Number
                if n, ok := v.(float64); ok {
                        typebaseaddr            :=
int(baseaddr - mem.NumOffset)
                        //   If   the   specified
address is bigger than the current size of the
array
                        // we need to grow the
array to that size
                        if     len(ms.num)     <=
typebaseaddr {
                                //    First    we
create a new slice with the extra cells we need
                                newslice       :=
make([]float64, typebaseaddr-len(ms.num)+1)
                                ms.num          =
append(ms.num, newslice...)
                                // Now we set the
value to the specified address

ms.num[typebaseaddr] = n
                        } else {

ms.num[typebaseaddr] = n
                        }
                        return nil
                }
                return   errutil.NewNoPosf("Cannot
set non-number in number address range")
        case   baseaddr   <   mem.BoolOffset:   //
Character
                if c, ok := v.(rune); ok {
                        typebaseaddr            :=
int(baseaddr - mem.CharOffset)
                        //   If   the   specified
address is bigger than the current size of the
array
                        // we need to grow the
array to that size
                        if     len(ms.char)     <=
typebaseaddr {
                                //    First    we
create a new slice with the extra cells we need
                                newslice       :=
make([]rune, typebaseaddr-len(ms.char)+1)
                                ms.char         =
append(ms.char, newslice...)
                                // Now we set the
value to the specified address

ms.char[typebaseaddr] = c
                        } else {

ms.char[typebaseaddr] = c
                        }
                        return nil
                }
                return   errutil.NewNoPosf("Cannot
set non-char in char address range")
        case  baseaddr  <  mem.FunctionOffset:  //
Boolean
                if b, ok := v.(bool); ok {
                        typebaseaddr            :=
int(baseaddr - mem.BoolOffset)
                        //   If   the   specified
address is bigger than the current size of the
array
                        // we need to grow the
array to that size
                        if   len(ms.booleans)   <=
typebaseaddr {
                                //    First    we
create a new slice with the extra cells we need
```

```
                                newslice       :=
make([]bool, typebaseaddr-len(ms.booleans)+1)
                                ms.booleans     =
append(ms.booleans, newslice...)
                                // Now we set the
value to the specified address

ms.booleans[typebaseaddr] = b
                        } else {

ms.booleans[typebaseaddr] = b
                        }
                        return nil
                }
                return   errutil.NewNoPosf("Cannot
set non-boolean in boolean address range")
        case baseaddr < mem.ListOffset: //Function
                if a, ok := v.(int); ok {
                        typebaseaddr           :=
int(baseaddr - mem.FunctionOffset)
                        //   If   the   specified
address is bigger than the current size of the
array
                        // we need to grow the
array to that size
                        if   len(ms.function)   <=
typebaseaddr {
                                //    First    we
create a new slice with the extra cells we need
                                newslice       :=
make([]int, typebaseaddr-len(ms.function)+1)
                                ms.function     =
append(ms.function, newslice...)
                                // Now we set the
value to the specified address

ms.function[typebaseaddr] = a
                        } else {

ms.function[typebaseaddr] = a
                        }
                        return nil
                }
                return   errutil.NewNoPosf("Cannot
set non-function in function address range %+v, %T
base: %d", v, v, baseaddr)
        case   baseaddr   <   mem.ListOffset+1000:
//List
                if a, ok := v.(*list.ListManager);
ok {
                        typebaseaddr           :=
int(baseaddr - mem.ListOffset)
                        //   If   the   specified
address is bigger than the current size of the
array
                        // we need to grow the
array to that size
                        if     len(ms.list)     <=
typebaseaddr {
                                //    First    we
create a new slice with the extra cells we need
                                newslice       :=
make([]*list.ListManager,
typebaseaddr-len(ms.list)+1)
                                ms.list         =
append(ms.list, newslice...)
                                // Now we set the
value to the specified address

ms.list[typebaseaddr] = a
                        } else {

ms.list[typebaseaddr] = a
                        }
                        return nil
                }
                return   errutil.NewNoPosf("Cannot
set non-list in list address range")
        default: // Error
                return  errutil.NewNoPosf("Address
out of scope")
        }
}
// GetValue takes and address and tries to retrieve
the corresponding value in that position
func (ms *MemorySegment) GetValue(addr mem.Address)
(interface{}, error) {
        baseaddr := addr - ms.base

        switch {
        case baseaddr < mem.NumOffset: // Error
                return                          nil,
errutil.NewNoPosf("Address out of scope")
        case baseaddr < mem.CharOffset: // Number
                typebaseaddr   :=   int(baseaddr   -
mem.NumOffset)
                if len(ms.num) <= typebaseaddr {
                        return                nil,
errutil.NewNoPosf("%s:  Referencing  address  %d  out
of   scope   of   %d",   ms.name,   typebaseaddr,
len(ms.num))
                }
                return ms.num[typebaseaddr], nil
        case   baseaddr   <   mem.BoolOffset:   //
Character
                typebaseaddr   :=   int(baseaddr   -
mem.CharOffset)
                if len(ms.char) <= typebaseaddr {
                        return                nil,
errutil.NewNoPosf("Referencing   address   out   of
scope")
                }
                return ms.char[typebaseaddr], nil
        case  baseaddr  <  mem.FunctionOffset:  //
Boolean
                typebaseaddr   :=   int(baseaddr   -
mem.BoolOffset)
                if       len(ms.booleans)       <=
typebaseaddr {
                        return                nil,
errutil.NewNoPosf("Referencing   address   out   of
scope")
                }
                return  ms.booleans[typebaseaddr],
nil
        case baseaddr < mem.ListOffset: //Function
                typebaseaddr   :=   int(baseaddr   -
mem.FunctionOffset)
                if       len(ms.function)       <=
typebaseaddr {
```

```
                        return              nil,
errutil.NewNoPosf("Referencing   address   out   of
scope")
                }
                return  ms.function[typebaseaddr],
nil
        case   baseaddr   <   mem.ListOffset+1000:
//List
                typebaseaddr  :=  int(baseaddr  -
mem.ListOffset)
                if len(ms.list) <= typebaseaddr {
                        return              nil,
errutil.NewNoPosf("Referencing   address   out   of
scope")
                }
                return ms.list[typebaseaddr], nil
        default: // Error
                return                      nil,
errutil.NewNoPosf("Address out of scope")
        }

}
//  Memory  represents  the  virtual  memory  for  the
virtual machine
// it contains one MemorySegment for each segment
type Memory struct {
        memglobal   *MemorySegment
        memlocal    *MemorySegment
        memtemp     *MemorySegment
        memconstant *MemorySegment
        memscope    *MemorySegment

func NewMemory() *Memory {
        return &Memory{
                NewMemorySegment(mem.Globalstart,
"Global"),
                NewMemorySegment(mem.Localstart,
"Local"),
                NewMemorySegment(mem.Tempstart,
"Temp"),

NewMemorySegment(mem.Constantstart, "Constant"),
                NewMemorySegment(mem.Scopestart,
"Scope"),
        }
}
// Get value takes an address and tries to retrieve
the saved value by consulting its memory segments
func   (m   *Memory)   GetValue(addr   mem.Address)
(interface{}, error) {
        switch {
        case addr < mem.Globalstart: // Error
                return                     false,
errutil.NewNoPosf("Address out of scope")
        case addr < mem.Localstart: // Global
                return int(addr), nil
        case addr < mem.Tempstart: // Local
                v,             err             :=
m.memlocal.GetValue(addr)
                if err != nil {
                        return nil, err
                }
                return v, nil
        case addr < mem.Constantstart: // Temp
                v, err := m.memtemp.GetValue(addr)
                if err != nil {
                        return nil, err
                }
                return v, nil
        case addr < mem.Scopestart: // Constant
                v,             err             :=
m.memconstant.GetValue(addr)
                if err != nil {
                        return nil, err
                }
                return v, nil
        case addr < mem.Scopestart+5000: // Scope
                v,             err             :=
m.memscope.GetValue(addr)
                if err != nil {
                        return nil, err
                }
                return v, nil
        default: // Error
                return                     nil,
errutil.NewNoPosf("Address out of scope")
        }
}
// SetValue takes a value and an address and then
consults its segments to
// try to save the value
func  (m  *Memory)  SetValue(v  interface{},  addr
mem.Address) error {
        switch {
        case addr < mem.Globalstart: // Error
                return  errutil.NewNoPosf("Address
out of scope")
        case addr < mem.Localstart: // Global
                if  err  :=  m.memglobal.SetValue(v,
addr); err != nil {
                        return err
                }
                return nil
        case addr < mem.Tempstart: // Local
                if  err  :=  m.memlocal.SetValue(v,
addr); err != nil {
                        return err
                }
                return nil
        case addr < mem.Constantstart: // Temp
                if  err  :=  m.memtemp.SetValue(v,
addr); err != nil {
                        return err
                }
                return nil
        case addr < mem.Scopestart: // Constant
                if             err             :=
m.memconstant.SetValue(v, addr); err != nil {
                        return err
                }
                return nil
        case addr < mem.Scopestart+5000: // Scope
                if  err  :=  m.memscope.SetValue(v,
addr); err != nil {
                        return err
                }
                return nil
        default: // Error
                return  errutil.NewNoPosf("Address
out of scope")
        }
}
```

AR

Este submódulo contiene la estructura de los activation record y la implementación para modificar sus atributos.

-----activationrecord.go

```
type ActivationRecord struct {
    retip    int
    numparams  []NumParam
    charparams []CharParam
    boolparams []BoolParam
    funcparams []FuncParam
    listparams []ListParam
    numtemps   []NumParam
    chartemps  []CharParam
    booltemps  []BoolParam
    functemps  []FuncParam
    listtemps  []ListParam
    numcount   int
    charcount  int
    boolcount  int
    funccount  int
    listcount  int
}
func (a *ActivationRecord) AddNumParam(num
    float64) {
        addr := mem.Address(a.numcount +
    mem.Localstart + mem.NumOffset)
        a.numparams = append(a.numparams,
    NumParam{num, addr})
    a.numcount++
}
func (a *ActivationRecord) AddCharParam(char
    rune) {
        addr := mem.Address(a.charcount +
    mem.Localstart + mem.CharOffset)
        a.charparams = append(a.charparams,
    CharParam{char, addr})
    a.charcount++
}
func (a *ActivationRecord) AddBoolParam(b bool)
    {
        addr := mem.Address(a.boolcount +
    mem.Localstart + mem.BoolOffset)
        a.boolparams = append(a.boolparams,
    BoolParam{b, addr})
    a.boolcount++
}
func (a *ActivationRecord) AddFuncParam(f int)
    {
        addr := mem.Address(a.funccount +
    mem.Localstart + mem.FunctionOffset)
        a.funcparams = append(a.funcparams,
    FuncParam{f, addr})
    a.funccount++
}
func (a *ActivationRecord) AddListParam(l
    *list.ListManager) {
        addr := mem.Address(a.listcount +
    mem.Localstart + mem.ListOffset)
        a.listparams = append(a.listparams,
    ListParam{l, addr})
        a.listcount++
    }
    func (a *ActivationRecord) AddNumTemp(num
        float64, addr mem.Address) {
            a.numtemps = append(a.numtemps,
        NumParam{num, addr})
    }
    func (a *ActivationRecord) AddCharTemp(char
        rune, addr mem.Address) {
            a.chartemps = append(a.chartemps,
        CharParam{char, addr})
    }
    func (a *ActivationRecord) AddBoolTemp(b bool,
        addr mem.Address) {
            a.booltemps = append(a.booltemps,
        BoolParam{b, addr})
    }
    func (a *ActivationRecord) AddFuncTemp(f int,
        addr mem.Address) {
            a.functemps = append(a.functemps,
        FuncParam{f, addr})
    }
    func (a *ActivationRecord) AddListTemp(l
        *list.ListManager, addr mem.Address) {
            a.listtemps = append(a.listtemps,
        ListParam{l, addr})
    }
```

-----arstack.go

```
    // Empty returns true if ArStack is empty
    func (s *ArStack) Empty() bool {
        return s.head == nil
    }
    // Pop removes the first element in the
container
    func (s *ArStack) Pop() {
        if s.Empty() {
            return
        }
        s.head = s.head.next
    }
    // Top returns the first element in the
container
    func (s *ArStack) Top() *ActivationRecord
{
        if s.Empty() {
            return nil
        }
        return s.head.val
    }
    // Push adds an element to the top of the
container
    func (s *ArStack) Push(val
*ActivationRecord) {
        newHead := &node{val, s.head}
        s.head = newHead
    }
```

List

Este submódulo contiene la estructura de la lista y su implementación en la memoria virtual. Se enlista como se manejan las funciones del sistema para poder manejar las listas. Y se incluye lo que sería la representación de las funciones de las listas en el tipo de dato num.

-----list.go

```
type ListManager struct {
    lnum  *ListNum
    lchar *ListChar
    lbool *ListBool
    lfunc *ListFunc
    llist *ListList
}
func (l *ListNum) List() []float64 {
    return l.list
}

func (l *ListNum) Size() int {
    return l.size
}

func (l *ListNum) Insert(n float64) {
    l.list = append([]float64{n}, l.list...)
    l.size++
}

func (l *ListNum) Head() (float64, error) {
    if len(l.list) < 1 {
      return  0,  errutil.NewNoPosf("Attempting
     to call head on empty list")
    }

    return l.list[0], nil
}

func (l *ListNum) Tail() ([]float64, error) {
    if len(l.list) < 1 {
                  return              l.list,
     errutil.NewNoPosf("Attempting  to call tail
     on empty list")
    }

    return l.list[1:], nil
}

func (l *ListNum) Copy() *ListNum {
    if l == nil {
     return nil
    }
    return &ListNum{l.list, l.size}
}
//String  List  Manager  representation  in  the
    virtual machine output
func (lm *ListManager) String() string {
    var builder strings.Builder

    builder.WriteString("[")

    if lm.lnum != nil {
     for i, n := range lm.lnum.list {
             if i != len(lm.lnum.list)-1 {
                        if       n       ==
float64(int64(n)) {

builder.WriteString(fmt.Sprintf("%d,     ",
int64(n)))
                } else {

builder.WriteString(fmt.Sprintf("%f,     ",
n))
                }
         } else {
                        if       n       ==
float64(int64(n)) {

builder.WriteString(fmt.Sprintf("%d",
int64(n)))
                } else {

builder.WriteString(fmt.Sprintf("%f", n))
                }
         }
 }
} else if lm.lchar != nil {
 for i, n := range lm.lchar.list {
         if i != len(lm.lchar.list)-1 {

builder.WriteString(fmt.Sprintf("%c,     ",
n))
         } else {

builder.WriteString(fmt.Sprintf("%c", n))
         }
 }
} else if lm.lbool != nil {
 for i, n := range lm.lbool.list {
         if i != len(lm.lbool.list)-1 {

builder.WriteString(fmt.Sprintf("%t,     ",
n))
         } else {

builder.WriteString(fmt.Sprintf("%t", n))
         }
 }
} else if lm.lfunc != nil {
 for i, n := range lm.lfunc.list {
         if i != len(lm.lfunc.list)-1 {

builder.WriteString(fmt.Sprintf("%d,     ",
n))
         } else {

builder.WriteString(fmt.Sprintf("%d", n))
         }
 }
} else if lm.llist != nil {
```

```
     for i, n := range lm.llist.list {
             if i != len(lm.llist.list)-1 {

    builder.WriteString(fmt.Sprintf("%s,     ",
    n))
             } else {

    builder.WriteString(fmt.Sprintf("%s", n))
             }
     }
    }

    builder.WriteString("]")

    return builder.String()
}
func NewListManager(t int) *ListManager {
    switch t {
    case 1:
                                        return
    &ListManager{&ListNum{make([]float64,   0),
    0}, nil, nil, nil, nil}
    case 2:
             return          &ListManager{nil,
    &ListChar{make([]rune,  0),  0},  nil,  nil,
    nil}
    case 3:
         return     &ListManager{nil,     nil,
    &ListBool{make([]bool, 0), 0}, nil, nil}
    case 4:
       return   &ListManager{nil,   nil,   nil,
    &ListFunc{make([]int, 0), 0}, nil}
    case 5:
      return  &ListManager{nil,  nil,  nil,  nil,
    &ListList{make([]*ListManager, 0), 0}}
    }

    return nil
}
//Add  Implementation  of  add  as  a  built  in
    function
func (lm *ListManager) Add(n interface{}) error
    {
    if f, ok := n.(float64); ok {
     if lm.lnum == nil {
             return  errutil.NewNoPosf("Cannot
    set num in non-num list")
     }
     lm.lnum.list = append(lm.lnum.list, f)
     return nil
    } else if c, ok := n.(rune); ok {
     if lm.lchar == nil {
             return  errutil.NewNoPosf("Cannot
    set num in non-char list")
     }
     lm.lchar.list = append(lm.lchar.list, c)
     return nil
    } else if b, ok := n.(bool); ok {
     if lm.lbool == nil {
             return  errutil.NewNoPosf("Cannot
    set num in non-bool list")
     }

     lm.lbool.list = append(lm.lbool.list, b)
     return nil
    } else if f, ok := n.(int); ok {
     if lm.lfunc == nil {
             return  errutil.NewNoPosf("Cannot
    set num in non-func list")
     }
     lm.lfunc.list = append(lm.lfunc.list, f)
     return nil
    } else if l, ok := n.(*ListManager); ok {
     if lm.llist == nil {
             return  errutil.NewNoPosf("Cannot
    set num in non-list list")
     }
     lm.llist.list = append(lm.llist.list, l)
     return nil
    }

        return  errutil.NewNoPosf("Cannot  cast
    element to valid form to add to list")
}
```

-----liststack.go

```
// Pop removes the first element in the
container
func (s *ListStack) Pop() {
    if s.Empty() {
         return
    }

    s.head = s.head.next
}
// Top returns the first element in the
container
func (s *ListStack) Top() *ListManager {
    if s.Empty() {
         return nil
    }
    return s.head.val
}
// Push adds an element to the top of the
container
func (s *ListStack) Push(val *ListManager)
{
    newHead := &node{val, s.head}
    s.head = newHead
}
//NewListStack Implementation of the lists
stack
func NewListStack() *ListStack {
    return &ListStack{nil}
}
```

7) Anexos

a) Bibliografía

1. Racket Documentation. (2020). Retrieved 29 March 2020, from https://docs.racket-lang.org/
2. Documentation. (2020). Retrieved 29 March 2020, from https://www.haskell.org/documentation/
3. Understanding Lambda Expressions. (2015). Retrieved 29 March 2020, from https://medium.com/@luijar/understanding-lambda-expressions-4fb7ed216bc5

b) Listado de Herramientas utilizadas

1. GOCC: https://github.com/goccmack/gocc
2. Package ErrorUtil: https://github.com/mewkiz/pkg/tree/master/errutil

```
// Package errutil implements some error utility functions.
package errutil

import (
    "errors"
    "fmt"
    "path"
    "runtime"

    "github.com/mewkiz/pkg/term"
)
// UseColor indicates if error messages should use colors.
var UseColor = true
// ErrInfo is en error containing position information.
type ErrInfo struct {
    // err is the original error message.
    Err error
      // pos refers to the position of the original error message. A nil value
      // indicates that no position information should be displayed with the error
    // message.
    pos *position
}
// position includes information about file name, line number and callee.
type position struct {
    // base file name.
    file string
    // line number.
    line int
    // callee function name.
    callee string
}
func (pos *position) String() string {
    if pos == nil {
           return "<no position>"
    }
    filePos := fmt.Sprintf("(%s:%d):", pos.file, pos.line)
    if UseColor {
           // Use colors.
           filePosColor := term.WhiteBold(filePos)
           if pos.callee == "" {
                  return filePosColor
           }
                  return        fmt.Sprintf("%s        %s", term.MagentaBold(pos.callee), filePosColor)
    }
    // No colors.
    if pos.callee == "" {
           return filePos
    }
    return fmt.Sprintf("%s %s", pos.callee, filePos)
}
// New returns an error which contains position information from the callee.
func New(text string) (err error) {
    return backendErr(errors.New(text))
}

// Newf returns a formatted error which contains position information from the
// callee.
func Newf(format string, a ...interface{}) (err error) {
    return backendErr(fmt.Errorf(format, a...))
}

// NewNoPos returns an error which explicitly contains no position information.
//  Further calls to Err will not embed any position information.
func NewNoPos(text string) (err error) {
    return &ErrInfo{Err: errors.New(text)}
}
//  NewNoPosf returns a formatted error which explicitly contains no position information.
//  Further calls to Err will not embed any position information.
func NewNoPosf(format string, a ...interface{}) (err error) {
    return &ErrInfo{Err: fmt.Errorf(format, a...)}
}
// ErrNoPos return an error which explicitly contains no position information.
func ErrNoPos(e error) (err error) {
    return &ErrInfo{Err: e}
}
// Err returns an error which contains position information from the callee. The
//  original position information is left unaltered if available.
func Err(e error) (err error) {
    return backendErr(e)
}
func backendErr(e error) (err error) {
    _, ok := e.(*ErrInfo)
    if ok {
           return e
    }
    pc, file, line, ok := runtime.Caller(2)
    if !ok {
           return e
    }
    var callee string
    f := runtime.FuncForPC(pc)
    if f != nil {
           callee = f.Name()
    }
    err = &ErrInfo{
           Err: e,
           pos: &position{
                  file:   path.Base(file),
                  line:   line,
                  callee: callee,
           },
    }
    return err
}
// Error returns an error string with position information.
//
// The error format is as follows:
//        pkg.func (file:line): error: text
func (e *ErrInfo) Error() string {
    text := "<nil>"
    if e.Err != nil {
           text = e.Err.Error()
    }

    if UseColor {
           // Use colors.
           prefix := term.RedBold("error:")
           if e.pos == nil {
                  return  fmt.Sprintf("%s  %s",  prefix, text)
           }
           return  fmt.Sprintf("%s  %s  %s",  e.pos,  prefix, text)
    }

    // No colors.
    if e.pos == nil {
           return text
    }
    return fmt.Sprintf("%s %s", e.pos, text)
```