Erick González A01039859
Carlos Estrada A01039919

Octavo Avance: Lambdish

Resumen del Avance al 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.

Séptimo Avance: Lambdish

Resumen del Avance al 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.

Sexto Avance: Lambdish

Resumen del Avance al 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.

Quinto Avance: Lambdish

Resumen del Avance al 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.

Cuarto Avance: Lambdish

Resumen del Avance al 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

Tercer Avance: Lambdish

Resumen del Avance al 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.

Segundo Avance: Lambdish

Resumen del Avance al 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 ejecuta 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

Primer Avance: Lambdish

Resumen del Avance al 13 de abril del 2020

1. **¿Funciona?**
   Sí.
2. **¿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 #.
3. **¿Corrieron suficientes test-cases?**
   Se ejecutaron 4 archivos de prueba dónde pudimos identificar que se recorrieran todos los estados generados por la gramática.
4. **¿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.
5. **¿Pueden marcar errores particulares o solo uno genérico (syntax error)?, etc**
   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).