**Analizador Sintáctico y Léxico**

**(Scanner y Parser)**

Scanner:

Mediante JFlex generamos creamos un archivo Lexer.lex, el cual consistió simplemente en escribir las especificaciones del lenguaje (operadores, palabras reservadas, identificadores, definiciones de tipos), las cuales consisten en la descripción del símbolo y su correspondiente creación mediante el comando ‘new Symbol’, cuya implementación se encuentra en el Parser generado por CUP.

Parser:

El Parser, creado a partir de java\_cup, describe básicamente la forma en la que el lenguaje se estructura, mediante la especificación de la gramática del mismo.

En esta etapa nos encontramos la dificultad de tener errores a la hora de generar el código Parser.java, por conflictos del estilo shift-reduce o reduce-reduce, el cual resolvimos mediante una reestructuración modular de la gramática, la cual evita encontrarse con producciones que deriven en lambda (vacío).

**Analizador Semántico**

En esta etapa añadimos lo correspondiente a la tabla de símbolos y la generación del Abstract Semantic Tree (AST) en el código del parser.cup, para luego almacenar los datos necesarios en la tabla y luego hacer los chequeos correctos en el AST al generar el parser.java.

Tabla de Símbolos

Nuestro diseño para la tabla de símbolos se basó en una estructura de árbol, ya que fue la primera opción práctica en la que se pensó.

Dicho árbol consta con un nodo raíz, el cual representa el primer nivel de la tabla, desde el cual se generan los hijos, mediantes los métodos implementados, a la hora de agregar más niveles. El diseño es el siguiente:

*La Clase Simbolo*

Implementa los símbolos (constantes y variables) que el usuario declara al escribir un programa mediante este lenguaje.

Colaboraciones: Type.

Variables:

Type type: especifica el tipo de un símbolo.

String name: especifica el nombre de un símbolo.

String value: especifica el valor de un símbolo (en caso de ser necesario).

Int size: especifica el tamaño de un símbolo (utilizado cuando se trata de arreglos).

Int offset: seria el offset del símbolo en caso de necesitarlo para la creación de VarLocations

Métodos:

Simbolo(Type type, String name, String value, int size,int offset): constructor de la clase. Inicializa las variables mediante los parámetros definidos y crea un objeto Simbolo.

getType(): obtiene el tipo de un objeto Simbolo. Retorna un valor de la clase enumerada Type.

setType(Type type): establece un nuevo tipo para un objeto Simbolo. Retorna void.

getName(): obtiene el nombre de un objeto Simbolo. Retorna string.

setName(String name): establece un nuevo nombre para un objeto Simbolo. Retorna void.

getValue(): obtiene el valor de un objeto Simbolo. Retorna string.

setValue(String value): establece un nuevo valor para un objeto Simbolo. Retorna void.

getSize(): obtiene el tamaño de un objeto Simbolo. Retorna int.

setSize(int size): establece un nuevo tamaño para un objeto Simbolo. Retorna void.

getOffset(): obtiene el offset del objeto Simbolo. Retorna int

setOffset(int o): establece un nuevo offset al objeto actual.

Esta clase también redefine el método toString(), para permitir imprimir por pantalla un objeto Simbolo, y el método equals, el cual permite comparar dos símbolos y determinar si son iguales (nombre, tipo y tamaño).

*La Clase Metodo*

Implementa los métodos y funciones que el usuario declara al escribir un programa mediante este lenguaje.

Colaboraciones: Simbolo, Type.

Variables:

boolean return\_no\_void: variable que indica si el símbolo retorna o no algo como resultado de su ejecución. Sirve para distinguir entre métodos (no retornan) y funciones (retornan).

String name: nombre del método.

Type typeReturn: tipo de retorno del método (puede ser void).

LinkedList<Simbolo> parámetros. Parámetros de un método (representado como lista de símbolos).

Métodos:

Metodo(Type returns. String name): constructor de la clase. Especifica el tipo de retorno y el nombre del método mediante los parámetros pasados, establece si dicho método retorna o no algo y crea un objeto de tipo Metodo.

Metodo(Type returns, String name, LinkedList<Simbolo> param): constructor alternative. A diferencia del anterior, este constructor es utilizado cuando el método posee parámetros.

isReturn\_not\_void(): método que especifica si el objeto Metodo retorna algún valor o no. Retorna boolean.

setReturn\_not\_void(boolean return\_not\_void): establece la condición de retorno del objeto Metodo, es decir, si el objeto retornará o no algún valor. Retorna void.

getName(): obtiene el nombre de un objeto Metodo. Retorna string.

setName(String name): establece un nuevo nombre para un objeto Metodo. Retorna void.

getTypeReturn(): obtiene el tipo de retorno de un objeto Metodo. Retorna un valor de la clase enumerada Type.

setTypeReturn(Type typeReturn): establece un nuevo tipo de retorno para un objeto Metodo. Retorna void.

getParametros(): obtiene los parámetros de un objeto Metodo. Retorna una lista de objetos Simbolo.

setParametros(LinkedLikst <Simbolo> parametros): establece nuevos parámetros para un objeto Metodo. Retorna void.

addParametro(Simbolo s): añade un nuevo parámetro a un objeto Metodo, siempre y cuando éste no haya sido agregado anteriormente (declarado). Retorna void.

Esta clase también redefine el método toString(), el cual permite imprimir por pantalla un objeto Metodo. Ademas cuenta con un metodo llamado mismosParametros que dados dos metodos verifica si ambos tienen los mismo parametros, el cual se utiliza para verificar si un método ya fue declarado, mediante la comparación de nombre y parametros

esParametro(Simbolo s): es un metodo que dado un objeto de tipo Metodo y un Simbolo, devuelve el índice en que se ubica el elemento o -1 en caso de no ser parámetro.

*La Clase Node*

Implementa un nodo para el árbol, el cual implementará la tabla de símbolos del analizador semántico.

Colaboraciones Simbolo, Metodo.

Variables:

LinkedList <Simbolo>variables: lista utilizada para registrar las variables declaradas mediante el lenguaje.

LinkedList<Metodo> metodos: lista utilizada para registrar la información de los métodos declarados mediante el lenguaje

LinkedList<Node> hijos: los hijos de un nodo representan distintos niveles, declarados bajo la misma precedencia.

Node padre: El nivel padre del nodo corriente (nivel corriente, el cual fue declarado por el nivel padre).

Métodos:

Node(): constructor de la clase. Inicializa las variables anteriores y crea un objeto Node. Retorna un objeto Node

Node(Node p): constructor alternativo. Igual que el anterior, con la diferencia de que asigna la variable Node padre con el valor pasado como parámetro, en lugar de null. Retorna un objeto Node

getVariables(): obtiene las variables de un objeto Node. Retorna un objeto Simbolo

getMetodos(): obtiene los métodos definidos en un objeto Node. Retorna un objeto Metodo

getHijos(): obtiene los hijos de un objeto Node. Retorna un objeto Node.

getPadre(): obtiene el padre de un objeto Node. Retorna un objeto Node.

setPadre(Node padre): establece el valor Node pasado como parámetro como nuevo valor para el atributo de tipo Node ‘padre’ de un objeto Node. Retorna void.

AddHijo(Node n): añade un nuevo hijo a un objeto Node. Retorna void.

esHoja(): determina si el objeto Node corriente es o no una hoja (no tiene hijos). Retorna boolean.

esRaiz(): determina si el objeto Node corriente es raíz (no tiene padre). Retorna boolean.

addMetodo(Metodo m): permite agregar un nuevo método a la lista de métodos de un objeto Node. Retorna void.

addVariable(Simbolo s): permite agregar una nueva variable a la lista de variables de un objeto Node.

gerSimbolo(Simbolo s): retorna, en caso de existir, el símbolo indicado como parámetro por dicho método. Retorna un objeto de Simbolo.

getMetodo(Metodo m): retorna, en caso de existir, el método indicado como parámetro por dicho método. Retorna un objeto de Metodo.

getSimboloByName(String id): tomando en cuenta el nombre pasado como parámetro, determina la existencia del símbolo que posea ese nombre, y de ser así lo retorna como resultado. Retorna un objeto Simbolo.

buscarMetodo(Metodo m): tomando en cuenta un metodo verifica si fue declarado en el nivel corriente, y en caso de encontrarlo, lo retorna.

Esta clase también redefine el método toString(), para poder imprimir por pantalla un objeto Node.

*La Clase SymbolTree*

Implementa la estructura de árbol, utilizada después por la tabla de símbolos.

Colaboraciones: Node, Type.

Variables:

Node raiz: representa la raíz del árbol, el nodo inicial de la estructura.

Node corriente: nodo actual en el cual se está trabajando.

Métodos:

SymbolTree(): constructor de la clase. Crea un nuevo objeto Node y se lo asigna al campo raiz, siendo a su vez este nodo raíz el nodo corriente. De esta manera se crea un objeto SymbolTree, un árbol vacío.

getRaiz(): devuelve el nodo raíz del árbol. Retorna un objeto Node.

GetCorriente(): retorna el nivel corriente en que se esta trabajando. Retorna un Node.

agregarNivel(): establece un nuevo nivel para la futura tabla de símbolos. Estos niveles representan tanto declaraciones de nuevos métodos como entrada a estructuras condicionales. Retorna void.

subirNivel(): retrocede a un nivel anterior, ya sea por la finalización de un condicional o de un método.

addSimbolo(Type type, String name, String value, int size): añande un Nuevo símbolo al nodo corriente. Retorna void.

addMetodo(Type returns, String name): agrega un metodo al nivel corriente, siempre que no exista uno que se llame igual y retorne un objeto del mismo tipo.

*La clase symbolTable*

Implementa la tabla de símbolos, la cual se encarga de almacenar las definiciones de métodos, identificadores y variables.

Colaboraciones: Type, Metodo, Simbolo, SymbolTree.

Variables:

SymbolTree TabladeSimbolos: representa la tabla de símbolos.

Métodos:

symbolTable(): inicializa la tabla de símbolos como una tabla vacía.

agregarSimbolo(Type type, String name, String value, int size): añade un nuevo símbolo a la tabla de símbolos. Retorna void.

agregarMetodo(Type returns, String name): añade un nuevo método a la tabla de símbolos. Retorna void.

agregarNivel(): añade un nuevo nivel a la tabla de símbolos, ya sea por la declaración de un nuevo método como por la entrada a una estructura condicional. Retorna void.

SubirNivel(): retrocede a un nivel anterior, por finalización de un método o estructura condicional. Retorna void.

simboloDeclarado(Type type, String name, String value, int size): determina si un símbolo (variable o constante) fue declarado anteriormente para poder usarse. Verifica su existencia en el nivel corriente o en los anteriores. Retorna boolean.

metodoDeclarado(Type result, String name): determina si un método (a o función) fue declarado anteriormente para poder usarse. Verifica su existencia en el nivel corriente o en los anteriores. Retorna boolean.

obtenerMetodo(): devuelve el último método declarado. Retorna un objeto Metodo.

obtenerSimbolo(): devuelve el último símbolo declarado. Retorna un objeto Simbolo.

buscarSimbolo(String id): busca y devuelve el símbolo cuyo nombre coincida con el parámetro especificado. Retorna un objeto Simbolo.

buscarMetodo(String name, LinkedList<Type> paramTypes): busca y devuelve el símbolo cuyo nombre y tipo y orden de parametros coincida con los pasados como parámetros. Retorna un objeto Metodo.

Parser.cup

La gramática esta estructurada para cumplir con las necesidades planteadas al inicio del proyecto.

A causa de un problema al entender como se aplicaba el patrón Visitor planteado, toda verificación de correctitud en el análisis semántico se realiza en esta clase, utilizando las estructuras proporcionadas que se utilizarían con el AST para el registro de datos.

Variables globales:

* symbolTable TS: que es la tabla de símbolos (con estructura de árbol para representar el anidado entre clase y métodos)
* Metodo ult\_metodo: lleva un registro del ultimo método creado para poder trabajar con el (ej: agregarle parámetros).
* boolean mainDeclared: variable booleana que indica si se declaro el método main
* boolean inIfElse: variable booleana que indica que nos encontramos dentro de una sentencia if-then-else
* boolean inLoop: variable booleana que inda que nos encontramos dentro de una estructura cíclica (while, for)
* boolean inMethod: variable booleana que indica que nos encontramos dentro del cuerpo de un método.
* LinkedList<TripletCode> TripletList: lista de códigos de tres direcciones generados.
* LinkedList<String> WhilesStack: lista de labels para whiles.
* LinkedList<String> ForsStack: lista de labels para fors.
* LinkedList<String> IfsStack: lista de labels para ifs.
* Int ultOffsetLeido: ultimo offset leído (al buscar una variable por ejemplo)
* String ultVarLeida: nombre de la ultima variable leída (utilizada en una operación de asignación o binaria por ejemplo).

Métodos:

* boolean validOperators(VarLocation s1, VarLocation s2): metodo que dadas dos VarLocations verifica si son validas para realizar una operación binaria aritmética
* boolean logicValidOperators(VarLocation r1,VarLocation r2): metodo que dadas dos VarLocations verifica si son validas para realizar una operación lógica entre ambas (conjunción, disyunción, comparaciones <, <=, >, >=, ==, etc).

Clase enumerada TripletOperator

Esta clase contiene las operaciones básicas necesarias para la generación de código intermedio de tres direcciones.

Ademas contiene un metodo toString() que retorna una cadena para cada valor del tipo enumerado.

Clase TripletCode:

Esta clase se utiliza para representar código de tres direcciones.

Contiene tres direcciones y un operador de tipo TripletOperator.

protected Object firstDir, secondDir: primer y segunda dirección.

protected Object result: dirección de retorno

protected TripletOperator operator: operador

TripletCode(TripletOperator op, Object fD, Object sD, Object res): constructor de clase

Object getFirstDir(): retorna un objeto que corresponde a la 1era dirección del código de tres direcciones actual

Object getSecondDir(): retorna un objeto que corresponde a la 2da dirección del código de tres direcciones actual

Object getResult(): retorna un objeto que corresponde a la 3era dirección del código de tres direcciones actual

TripletOperator getOperator(): retorna un TripletOperator del código de tres direcciones actual.

Ademas cuenta con un metodo toString() que retorna una cadena para representar el código de tres direcciones actual.

Clase Labels:

esta clase se utiliza para generar un numero incremental que se utiliza solo para numerar las labels creadas para los códigos de tres direcciones, siempre que sea necesario uno.

static int l: variable estática de tipo Int que se aumenta cuando se utiliza el metodo getLabel() el cual retorna una cadena conteniéndolo y lo aumenta a su vez.

Clase AssemblyCodeGenerator:

esta clase se encarga de la generación de código Assembler a partir del código de tres direcciones.

Variables:

* int OrAndlabelInt: nro de label que se utilizara en la generación de Or y And.
* String result: variable de tipo cadena la cual contendrá el código Assembler generado.

Metodos:

* AssemblyCodeGenerator(): constructor de clase. Es un constructor vacío.
* void incLabelOrAnd(): incrementa el nro de label para Or y And en 1.
* void resetLabelOrAnd(): resetea el nro de label para Or y And a 0.
* int evaluateExpression(String expr): metodo que dada una operación aritmética en forma de cadena devuelve su evaluación como entero.
* void ASM\_Add(TripletCode t): metodo que genera el código Assembler para una suma a partir de un código de tres direcciones.
* void ASM\_Sub(TripletCode t): metodo que genera el código Assembler para una resta a partir de un código de tres direcciones.
* void ASM\_Imul(TripletCode t): metodo que genera el código Assembler para una multiplicación a partir de un código de tres direcciones.
* void ASM\_Idiv(TripletCode t): metodo que genera el código Assembler para una división a partir de un código de tres direcciones.
* void ASM\_logic(TripletCode t, TripletOperator logic): metodo que genera el código Assembler para las operaciones lógicas <, >, >=, <=, == y !=, a partir de un código de tres direcciones.
* void ASM\_Non(TripletCode t): metodo que genera el código Assembler para una negación lógica a partir de un código de tres direcciones.
* void ASM\_Unary\_Minus(TripletCode t): metodo que genera el código Assembler para un menos unario, a partir del código de tres direcciones correspondiente.
* void ASM\_Mod(TripletCode t): metodo que genera el código Assembler para una operación de Modulo a partir del código de tres direcciones.
* void ASM\_Assing(TripletCode t): metodo que genera el código Assembler para una asignación.
* void ASM\_Label(TripletCode t): metodo que genera el código assembler para la declaración de un Label.
* void ASM\_Cmp(TripletCode t): metodo que genera el código Assembler para una comparación.
* void ASM\_Method\_Decl(TripletCode t): metodo que genera el código Assembler para la declaración de un metodo.
* void ASM\_Or(TripletCode t): metodo que genera el código Assembler para una disyunción.
* void ASM\_And(TripletCode t): metodo que genera el código Assembler para una conjunción.
* void ASM\_Params(TripletCode t): metodo que genera el código Assembler para el registro de parámetros para las llamadas a métodos.
* void ASM\_Increment(TripletCode t): metodo que genera el código Assembler para un incremento.
* void ASM\_Decrement(TripletCode t): metodo que genera el código Assembler para un decremento.
* String generate(LinkedList<TripletCode> list): metodo que dada la lista de códigos de tres direcciones proveniente del parser, genera el código assembler correspondiente.