Diseño de Compiladores 1

Trabajo Práctico

Autores:
Roman Luna
romanluna1234@gmail.com

Savo Gabriel Koprivica
skoprivica@alumnos.exa.unicen.edu.ar

Grupo:
20

Cátedra:
Diseño de compiladores 1

Profesores:
Jose A. Fernandez Leon

Introducción

El presente informe exhibe el desarrollo de la creación de un compilador basado en el lenguaje Java. En este informe se verá la creación del analizador Léxico(El cual se encarga de detectar los tokens del código, así como también los errores léxicos que pueda contener) y el analizador Sintáctico(Encargado de la parte sintáctica del código, detectando las estructuras sintácticas y errores) de la Generación de código intermedio (Árbol Sintactico y Comprobaciones sintácticas) y Generación de salida (Assembler y Salida de Assembre).. Las decisiones tomadas y cómo resolvimos diversos problemas o cuestiones que nos fueron surgiendo.

Temas Particulares

6. Enteros largos (32 bits): Constantes enteras con valores entre -2^{31} y $2^{31} - 1$. Estas constantes llevarán el sufijo "**_l**".

Enteros sin signo (16 bits): Constantes con valores entre 0 y $2^{16} - 1$. Estas constantes llevarán el sufijo "**_ui**". Se deben incorporar a la lista de palabras reservadas las palabras **LONG** y **UINT**.

7. Punto Flotante de 32 bits: Números reales con signo y parte exponencial. La parte exponencial puede estar ausente. Si está presente, el exponente comienza con la letra "**e**" (mayúscula o minúscula) y el signo del exponente es obligatorio.

Puede estar ausente la parte entera o la parte decimal, pero no ambas. El „." es obligatorio. Ejemplos válidos:

1. .6 -1.2 3.e–5 2.E+34 2.5E-1 15. 0. 1.2e+10

Considerar el rango 1.17549435E-38 < x < 3.40282347E+38 U

-3.40282347E+38 < x < -1.17549435E-38 U 0.0

10. En los lugares donde un identificador puede utilizarse como operando (expresiones aritméticas o comparaciones), considerar el uso del operador '--' luego del identificador. Por ejemplo: a = b-- * 7_i,

z = a---b--,

IF (a-- > 2_l) …

13 .WHILE (<condicion>) **DO** <Bloque_de_sentencias_ejecutables>

<condicion> tendrá la misma definición que la condición de las sentencias de selección.

<Bloque_de_sentencias_ejecutables> podrá contener una sentencia, o un grupo de sentencias ejecutables delimitadas por llaves.

18. Herencia por Composición - Uso con nombre

19. Declaración de métodos concentrados con control de niveles de herencia:

Los métodos se declaran dentro de la declaración de la clase a la que pertenecen. Además, el compilador debe chequear que no haya más de 3 niveles de herencia. Por ejemplo, si la clase A hereda de la clase B, y B hereda de la clase C, no se debe permitir que la clase C herede de ninguna otra clase. El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

21. Forward declaration

23. Sobreescritura de métodos: Una clase que hereda de otra puede sobreescribir métodos declarados en la clase de la que hereda, pero no puede sobreescribir atributos. El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

26. El compilador debe chequear e informar, para cada variable declarada, si no aparece en el lado izquierdo de al menos una asignación en el ámbito donde se declaró.

29. Conversiones Explícitas: **TOF**

- Se debe considerar que cuando se indique la conversión TOF(expresión), el compilador deberá generar código para efectuar una conversión del tipo del argumento al tipo indicado por la palabra reservada. Dado que los otros tipos asignados al grupo son enteros (1-2-3-4-5-6), el argumento de una conversión, debe ser de alguno de esos dos tipos.
- Sólo se podrán efectuar operaciones entre dos operadores de distinto tipo (uno entero y otro flotante), si se convierte el operando de tipo entero (1-2-3-4-5-6) al tipo de punto flotante mediante la conversión explícita que corresponda. En otro caso, se debe informar error.
- En el caso de asignaciones, si el lado izquierdo es de uno de los tipos enteros asignados (1-2-3-4-5-6), y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos.

33. Comentarios multilínea: Comentarios que comiencen con “*{” y terminen con “}*” (estos comentarios pueden ocupar más de una línea).

35. Cadenas multilínea: Cadenas de caracteres que comiencen y terminen con “ % ” . Estas cadenas pueden ocupar más de una línea. (En la Tabla de símbolos se guardará la cadena sin los saltos de línea). Ejemplo: % ¡Hola

mundo! %

Controles en Tiempo de Ejecución

Incorporar los chequeos en tiempo de ejecución que correspondan al tema particular asignado al grupo. Cada grupo deberá efectuar los chequeos indicados para situaciones de error que pueden producirse en tiempo de ejecución. El código generado por el compilador deberá, cada vez que se produzca la situación de error correspondiente, emitir un mensaje de error, y finalizar la ejecución.

División por cero para datos enteros y de punto flotante: El código Assembler deberá chequear que el divisor sea diferente de cero antes de efectuar una división. Este chequeo deberá efectuarse para los dos tipos de datos asignados al grupo.

Overflow en sumas de datos de punto flotante: El código Assembler deberá controlar el resultado de la operación indicada, para el tipo de datos de punto flotante asignado al

grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.

Overflow en productos de enteros: El código Assembler deberá controlar el resultado de la operación indicada, para los tipos de datos enteros asignados al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.

Decisiones de Diseño

Para la implementación del compilador, decidimos usar el lenguaje java. Creamos una tabla de símbolos que se encarga de almacenar todos los símbolos de identificadores, constantes y cadenas distintas que haya en el código que se está compilando. Los símbolos en esta etapa solo tienen el lema y el identificador correspondiente. A su vez también tenemos 2 estructuras más encargadas de almacenar el resto de símbolos posibles, la tabla de palabras reservadas y la tabla de caracteres ASCII. La decisión de qué tabla utilizar dependiendo el token es de las acciones semánticas. Dependiendo del estado actual y el carácter actual es la acción semántica que se ejecuta. Las acciones semánticas son las encargadas de ir armando los tokens de manera correcta, así como también detectar los errores y almacenar los símbolos detectados. En lo que corresponde a la parte sintáctica, implementamos la gramática siguiendo las indicaciones propuestas por la cátedra.

El compilador comienza en el main. Se le pasa el código a compilar como un objeto de tipo reader. El mismo se encuentra en el analizador léxico. Se ejecuta la función run() del parser. El parser es el que le solicita los tokens al léxico y arma las sentencias que se van detectando. Una vez finalizada la compilación, se vuelve al main y se imprimen por pantalla todos los errores sintácticos encontrados. Luego se imprime la lista de tokens detectados, así como los errores léxicos. Todas las sentencias detectadas en el sintáctico así como los errores léxicos y sintácticos muestran la línea en la que ocurrieron utilizando una variable que se encuentra en el Analizador Léxico, la cual se actualiza cada vez que se encuentra un nuevo salto de línea.

Las estructuras antes mencionadas son creadas a partir de una clase que se encarga de leer los archivos txt y formarlas. Todo lo antes mencionado es detallado a continuación en el informe.

Clases Generales

Tabla de símbolos

Esta clase es la encargada de almacenar los diferentes símbolos que van surgiendo en el código que estemos compilando. Los símbolos se guardan en un Hashtable<string, símbolo> en el cual la llave string corresponde al lexema. Antes de agregar un nuevo símbolo corrobora que no exista previamente en la tabla. Estos símbolos pueden corresponder a identificadores, constantes o cadenas.

Símbolo

Un símbolo es una clase que utilizamos para la representación de identificadores, constantes, cadenas, variables, etc. Su lexema, un entero como id, strings para representar tipo, uso, herencia y un símbolo como representación de parámetro en caso de ser tipo “Función” y tenerlo. Salvo el lexema todos estos campos se pueden inicializar como vacío.

LectorArchivo

Esta clase la decidimos implementar para mantener un mejor orden en el compilador. Dicha clase se encarga de crear las tablas o matrices correspondientes a partir de los archivos txt mencionados anteriormente Como por ejemplo la Matriz de transición estados, matriz de acciones semánticas, caracteres ascii, etc.

Constantes

Como Lector Archivo, esta clase la decidimos crear para mejorar la legibilidad y el orden del código. En esta se encuentran constantes que utilizamos en varias partes del programa y a su vez las tablas/matrices que vamos a utilizar, como la Matriz de acciones semánticas o las palabras reservadas, etc.

Main

El main es el encargado de realizar la ejecución del parser, el cual utiliza el léxico. Primero seteamos el código a compilar en el Analizador Léxico y luego se ejecuta la función run() del parser. Mientras el parser se ejecuta, se van consumiendo los tokens encontrados y se va imprimiendo por pantalla las sentencias/bloques que se van detectando con el parser. Los errores sintácticos y léxicos se van almacenando para luego ser mostrados al final de la ejecución del compilador. Una vez finalizada la ejecución de la función parser.run(). El main imprime todos los errores sintácticos encontrados así como también la lista de tokens detectados como los errores léxicos.

Analizador Léxico

Acciones Semánticas

Tenemos los siguientes paquetes:

- **Acciones Semánticas:** En este paquete se encuentran todas las acciones semánticas utilizadas para poder construir los token de manera correcta. Para implementarla decidimos utilizar una interfaz, la cual implementamos en cada AS individual.

  Las acciones semánticas son:

- **AS0:** Esta Acción Semántica se encarga de leer los espacios en blanco, tabulaciones y saltos de línea, sin agregarlos al token actual. En caso de ser un salto de línea también aumenta uno la variable linea_actual del analizador léxico.
- **AS1:** Esta Acción Semántica se encarga solamente de leer el próximo carácter del código y concatenarlo con el token actual. Si el carácter es un salto de línea le suma 1 a la variable linea_actual del Analizador Léxico. Esta acción retorna 0 lo que significa que que el token todavía no está listo y que tiene que seguir leyendo.
- **AS2:** Esta Acción Semántica se encarga de leer los caracteres especiales: '+' , '/' , ';' , ',' , '{' , '}' , '(' , ')'. Los busca en el mapa de caracteres ASCII y retorna el identificador del mismo.
- **AS3:** Esta Acción Semántica se encarga de verificar que el token encontrado corresponda a un identificador válido. Se encarga de verificar que su longitud no supere los 20 caracteres, en cuyo caso lo trunca eliminando los caracteres excedentes del final y agrega el Warning a la lista de errores del Analizador Léxico, para luego informarle sobre el mismo al usuario y en que linea ocurrió. Luego de verificar su longitud, lo agrega a la tabla de símbolos en caso de no estar y retorna el ID correspondiente.
- **AS4:** Esta Acción Semántica se encarga de verificar si el token obtenido pertenece a una palabra reservada(del tipo ==, >=, etc) o si pertenece a un carácter ASCII(=, >, <, etc). Luego obtiene el símbolo y retorna el identificador del mismo.
- **AS5:** Esta Acción Semántica se encarga de verificar que tipo de constante entera se encontró, si un entero largo o un entero sin signo. Luego verifica que el rango no supere el máximo valor permitido(A este se le suma uno más por el rango de los negativos). Luego se verifica el rango en la Gramática, dependiendo de si es un número positivo o negativo. Almacena el valor en la tabla de símbolos y entrega el token.
- **AS6:** Esta Acción Semántica se encarga de verificar si la constante flotante excede o no el rango permitido, en caso de no hacerlo lo agrega a la tabla de símbolos y retorna el identificador.
- **AS7:** Esta Acción Semántica se encarga de leer el siguiente carácter, concatenado con el token actual ya que forma parte del mismo y luego retornando el identificador del mismo.
- **AS8:** Esta Acción Semántica Se encarga de buscar si el token encontrado pertenece a una palabra reservada válida. Si es así, retorna el identificador de la misma.
- **AS9:** Esta acción Semántica se encarga de leer los comentarios. Una vez finalizado el token de comentario, esta acción se encarga de eliminarlo y retirar 0,(que significa que siga leyendo). Ya que los comentarios no deben ser almacenados como tokens en la tabla de símbolos.
- **AS10:** Esta acción Semántica se encarga de almacenar las cadenas en la tabla de símbolo y retorna el id del mismo.
- **ASE:** Acción semántica encargada de retornar error. En caso de que ocurra un error, por ejemplo, detecte un carácter no válido. Esta acción retorna -1 y detiene la ejecución de la compilación.

- **archivosTxt**: En este paquete se encuentran los archivos txt con los cuales generamos las diferentes tablas/matrices/mapas que vamos a necesitar para el funcionamiento de nuestro analizador léxico:
  - Tabla de palabras reservadas.
  - Tabla de caracteres ASCII.
  - Matriz de acciones semánticas.
  - Matriz de transición de estados.
  - Gramática(Utilizada para generar el parser).

- **Compilador**: Este paquete contiene todas las clases que se utilizan para el funcionamiento del Compilador. Las cuales detallaremos posteriormente en el informe
- **Testeos**: Contiene todos los códigos de prueba utilizados para verificar que el compilador funciona de manera correcta.

Clase Analizador Léxico

Esta es la clase principal del analizador léxico, la cual se encarga de generar los tokens y enviárselos al parser. La función principal dentro de la clase es proximoEstado. La misma al ser invocada se ejecuta, avanzando por diferentes estados hasta generar un token completo. Si el token no incumple ninguna regla ni contiene ningún error, el mismo es almacenado en la tabla de símbolos y su id es retornada para poder ser analizada por el analizador sintáctico. El analizador léxico es el encargado de utilizar las acciones semánticas dependiendo de lo indicado en la tabla de acciones semánticas y en la tabla de transición de estados.

Matriz de Acciones Semánticas:

Matriz de transición de estados:

Autómata De transición de Estados

Salida del Analizador Léxico

Para mostrar la salida del analizador léxico, tanto los tokens obtenidos como los errores, decidimos hacerlo al final de la ejecución del compilador. Mostramos la lista de tokens encontrados así como también los errores.

Ejemplo de salida del Léxico para el siguiente código:

```
{
  FLOAT _numero4;_hola,
  FLOAT _numero5,

  LONG _numero6,

  _numero6 = _numero4,
  _numero6 = _numero4 + _hola,

  VOID _fun1 (FLOAT _numero4){
        LONG _dentro,
        FLOAT _alto,
        _alto = _dentro,
        RETURN,
  },
}
```

La salida resultante es:

```
TOKENS DETECTADOS POR EL LEXICO:
[ FLOAT , 269 ]
[ _numero6 , 257 ]
[ ( , 40 ]
[ _numero5 , 257 ]
[ ) , 41 ]
[ _numero4 , 257 ]
[ + , 43 ]
[ , , 44 ]
[ _alto , 257 ]
[ _hola , 257 ]
[ RETURN , 272 ]
[ _dentro , 257 ]
[ { , 123 ]
[ ; , 59 ]
[ _fun1 , 257 ]
[ VOID , 266 ]
[ = , 61 ]
[ } , 125 ]
[ LONG , 267 ]
Errores Lexicos:
No hubo ningun error lexico
```

Analizador Sintactico

El objetivo de esta parte del trabajo era construir un Parser (Analizador Sintáctico) que utilice al Analizador Léxico, y que reconozca un lenguaje con las siguientes características:

Programa constituido por un conjunto de sentencias, que pueden ser declarativas o ejecutables. Las sentencias declarativas pueden aparecer en cualquier lugar del código fuente, exceptuando los bloques de las sentencias de control.

Los elementos declarados sólo serán visibles a partir de su declaración (esto será chequeado en etapas posteriores).

El programa estará delimitado por llaves ‘{‘ y ‘}’. Cada sentencia debe terminar con coma ",".

Decisiones e implementación:

Para realizar el parser tuvimos que implementar la gramática siguiendo las pautas que nos brinda el TP2. Tratamos de llevar un orden en la gramática para facilitar su entendimiento y a su vez utilizar nombres claros para cada regla. También agregamos reglas con errores, ejemplo, una regla sin una coma al final. Si se detecta un error en el código, se almacena un mensaje de error que es mostrado al final de la ejecución del parser. Al momento de realizar la gramática no nos surgieron grandes inconvenientes y todos fueron solucionados probando y modificando la gramática hasta obtener un funcionamiento óptimo.

Lista de no terminales de la Gramática:

- **bloque_sentencia_programa**: El bloque que va a contener las sentencias ya sean declarativas o ejecutables
- **sentencia_programa**: Puede ser una sentencia ejecutable o declarativa
- **sentencias_declarativas**: Son las sentencias de declaración, puede ser por ejemplo, una variable, función, etc.
- **sentecias_ejecutables**: Son las sentencias que pueden ejecutarse y realizar cambios en el programa, por ejemplo, asignaciones, mensajes de salida, etc.
- **declaracion_variables**: Se declara una variable, el nombre y de qué tipo de variable se trata
- **declaracion_funciones**: Declara la función. El formato es VOID luego nombre de función y por último el bloque de la función
- **tipo**: Se utiliza para ver de qué tipo es la variable, si entero largo, entero sin signo o flotante
- **list_de_variables**: Utilizada para poder declarar más de una variable del mismo tipo sin la necesidad de estar aclarando el tipo a cada variable por separado, ejemplo, FLOAT _a;_b;_c,
- **cuerpo_de_funcion**: En esta regla se pueden definir dentro todas las sentencias que va a poseer la función.
- **sentencia_declarativa_especifica**: Son las sentencias declarativas que pueden ser utilizadas dentro de una función, ya que no son las mismas que se permiten en el cuerpo del programa.
- **sentencias_retorno**: es la que se utiliza al final de una función
- **declaracion_clases**: Sirve para declarar las clases, la forma que tiene que tener la **declaracion** de la clase.
- **cuerpo_clase**: el cuerpo de clase es las sentencias que se permiten dentro de la declaracion de una clase.
- **list_objts_clase**: Permite crear como en el caso de las variables una declaración del objeto de la clase sin la necesidad de colocarle el tipo a cada una por separado.
- **sentencia_ejecucion_funcion**: Son las sentencias ejecutables que se permiten dentro de una función.
- **sentencia_asignacion**: Sentencia que le asigna el valor a una variable, ya sea una constante o el retorno de una función.
- **valor_asignacion**: Puede ser una expresión aritmética
- **invocacion_funcion**: es posible invocar una función en cualquier parte del código.
- **sentencia_IF**: Es la sentencia ejecutable del bloque if, la forma que tiene que tener.
- **condicion_if_while**: Son las condiciones que se pueden colocar en un if o un while, ejemplo, mayor, menor, menor o igual, etc.
- **bloque_sentencias_ejecutables**
- **sentencias_salida**: Es el print que muestra por pantalla, en nuestro caso solo puede ser una cadena por lo especificado en el enunciado
- **sentencias_control**: es el bloque WHILE
- **sentencias_while_do**: Determina cómo debe declararse el bloque while, la forma que tiene que tener.
- **expr_aritmetic**: puede ser suma, resta o un término
- **término**: puede ser una multiplicacion, division, factor o invocacion_funcion
- **factor**: constante o un identificador
- **const**: puede ser positiva o negativa.

Salida el analizador Sintáctico

La salida del analizador sintáctico se realiza mientras se está ejecutando el parser. Cada vez que se detecta una regla nueva se imprime por pantalla la línea en la que se encontró, así como también un mensaje descriptivo de lo que se detectó. En caso de ser un error, como mencionamos anteriormente, este se almacena en una lista para ser mostrado una vez finalizado el parser.

Ejemplo de salidas del analizador sintáctico:

```
{
	CLASS class_1,

	CLASS class_2 {

		LONG long1;long2,
		FLOAT float1,

		VOID m1 (LONG a){
			LONG aux,

			IF (a < 5_l){
				aux = 10_l,
			}END_IF,

			RETURN,

		},

		class_1,

	},

	CLASS class_1 {
		LONG long3,
	},

	class_2 obj_1,

	obj_1.long1 = 10_l,
	obj_1.float1 = 8.,

	obj_1.class_1.long3 = 21_l,

}
```

```
Linea: 2. Se reconocio una DECLARACION de CLASE A POSTERIOR
Linea: 2. Se reconocio sentencia DECLARATIVA
Linea: 6. Se reconocio un tipo LONG
```

```
Linea: 7. Se reconocio un tipo FLOAT
Linea: 9. Se reconocio un tipo LONG
Linea: 10. Se reconocio un tipo LONG
Linea: 12. Se reconocio un IDENTIFICADOR
Linea: 12. Se reconocio una CONSTANTE POSITIVA
Linea: 12. Se reconocio una COMPARACION POR MENOR
Linea: 13. Se reconocio una CONSTANTE POSITIVA
Linea: 13. Se reconocio una ASIGNACION
Linea: 14. Se reconocio un IF
Linea: 16. Se reconocio RETURN
Linea: 18. Se reconocio una DELCARACION de FUNCION CON PARAMETRO
Linea: 22. Se reconocio una DECLARACION de CLASE
Linea: 22. Se reconocio sentencia DECLARATIVA
Linea: 25. Se reconocio un tipo LONG
Linea: 26. Se reconocio una DECLARACION de CLASE
Linea: 26. Se reconocio sentencia DECLARATIVA
Linea: 28. Se reconocio una DECLARACION de OBJETO DE CLASE
Linea: 28. Se reconocio sentencia DECLARATIVA
Linea: 30. Se reconocio una CONSTANTE POSITIVA
Linea: 30. Se reconocio una ASIGNACION
Linea: 30. Se reconocio sentencia EJECUTABLE
Linea: 31. Se reconocio una CONSTANTE POSITIVA
Linea: 31. Se reconocio una ASIGNACION
Linea: 31. Se reconocio sentencia EJECUTABLE
Linea: 33. Se reconocio una CONSTANTE POSITIVA
Linea: 33. Se reconocio una ASIGNACION
Linea: 33. Se reconocio sentencia EJECUTABLE
Linea 35, Se reconocio el programa
ARRANCO EL PROGRAMA

Errores Sintacticos:
NO HUBO ERRORES SINTACTICOS
```

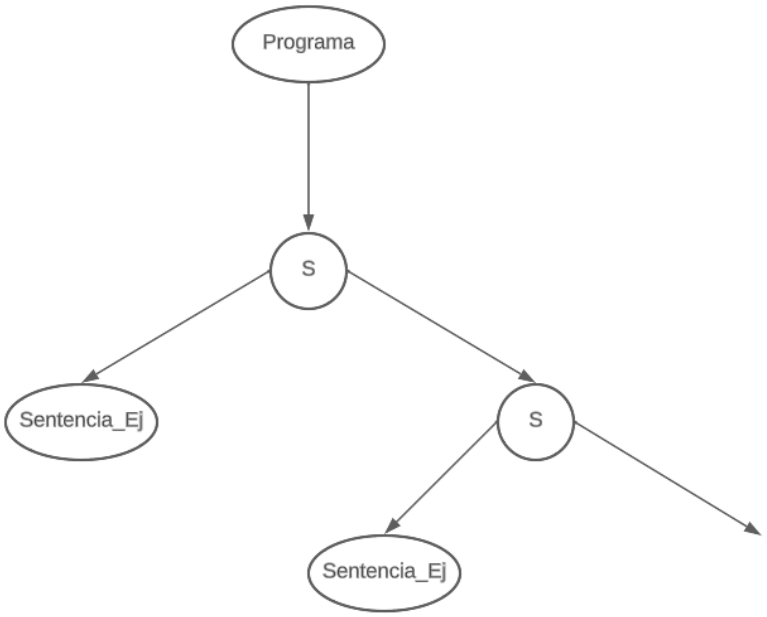
Generación de Código Intermedio

Árbol Sintáctico

El Árbol Sintactico es una estructura de datos que representa la estructura jerárquica y la semántica de un programa después de haber sido analizado sintácticamente. Es una representación intermedia que se utiliza comúnmente en la fase de análisis semántico de un compilador. Se compone por: nodos y hojas, donde los nodos del árbol representan operaciones o construcciones, mientras que las hojas representan operaciones o valores.

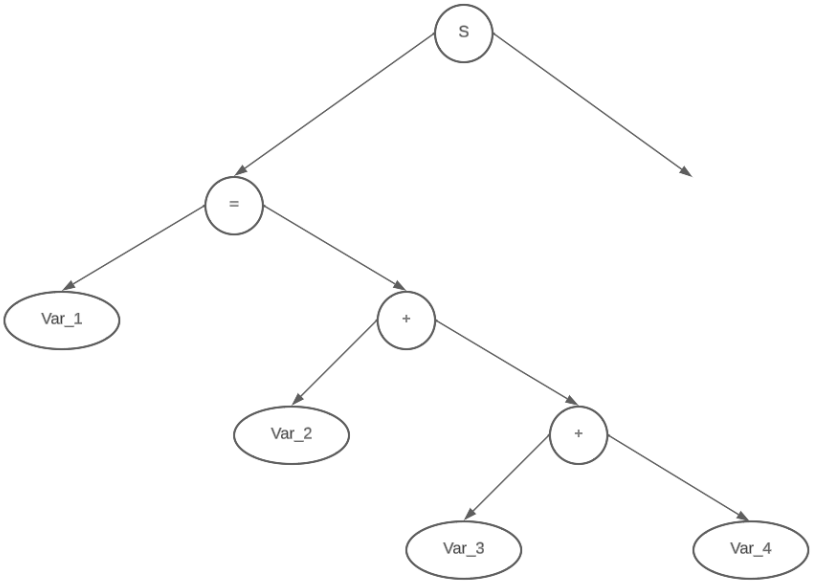
Cada nodo puede tener cero o más nodos hijos, dependiendo de la construcción gramatical que representa.

Estructura general del Árbol



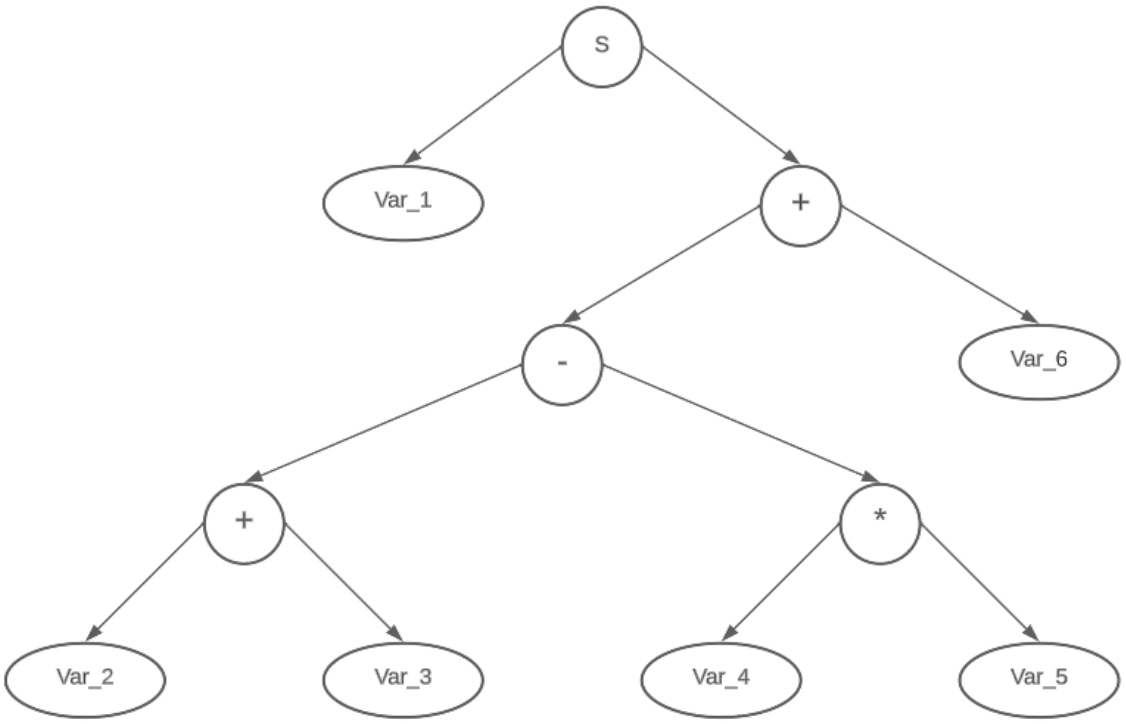
La forma en la que se genera el árbol es a partir de un nodo Control llamado “programa”, se le crea un hijo llamado sentencia. La manera en la que se crea el árbol es:
1°) Género un nodo Sentencia
2°) En su hijo izquierdo agregó una sentencia ejecutable, ya sea una asignación, llamado Función, etc.
3°) Al generar una nueva rama que corresponde a una sentencia ejecutable, si el hijo izquierdo del nodo “sentencia” es null, lo agrego en ese lugar. Sino Repito desde el paso uno.
El hijo derecho siempre va a corresponder a un nodo “sentencia”, y el hijo izquierdo a una subrama generada por una sentencia ejecutable.

Generación de una rama asignación:

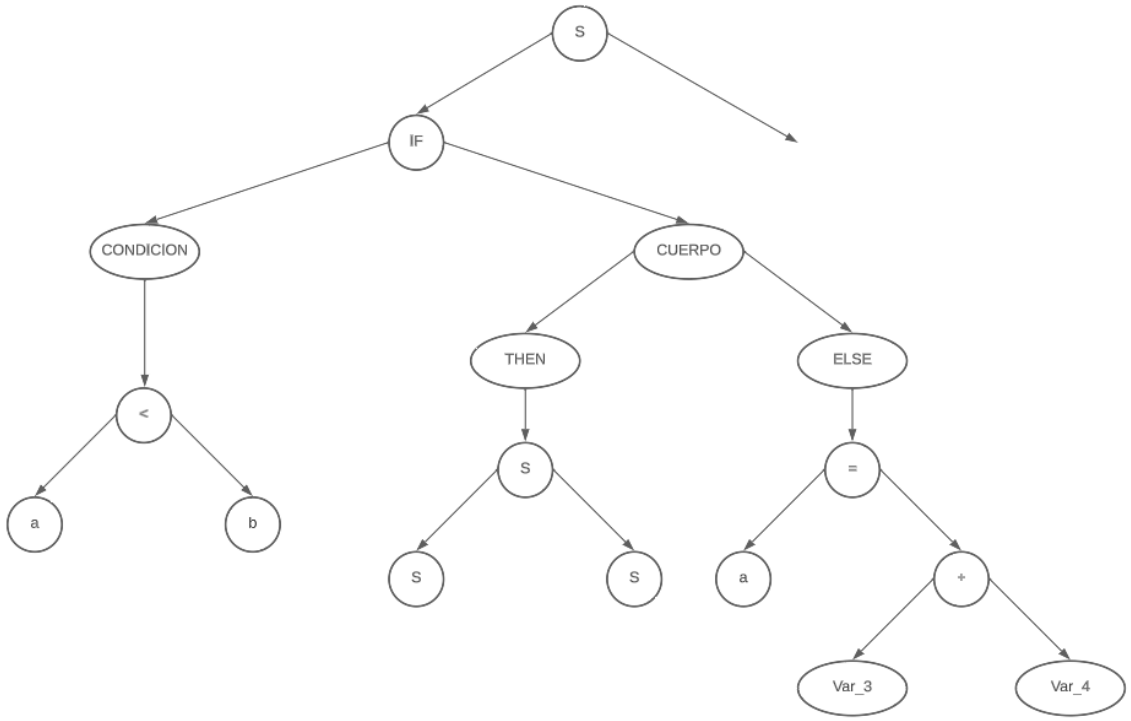


Una asignación se genera de las hojas a la raíz, que en este caso es “=”.
1°) se crean los nodos hojas correspondientes a var3 y var4
2°) se crea un nodo padre “+” y se le agregan los dos nodos hoja creados
3°) se crea el nodo hoja de var2. luego se crea otro nodo “+” y se le agregan como hijos var2 y el subárbol generado anteriormente.
4°) Se repite el mismo proceso hasta generar todo el lado derecho de la asignación
5°) se crea el nodo var1. se crea el nodo “=” y se le asigna el subárbol del lado derecho de la asignación y el lado izquierdo.
Para realizar esta asignación se realizan los chequeos semánticos necesarios para verificar que no hay errores de tipo por ejemplo.

El proceso de generación de los subÁrboles asignación respeta la procedencia de las operaciones, como se ve en el siguiente ejemplo:

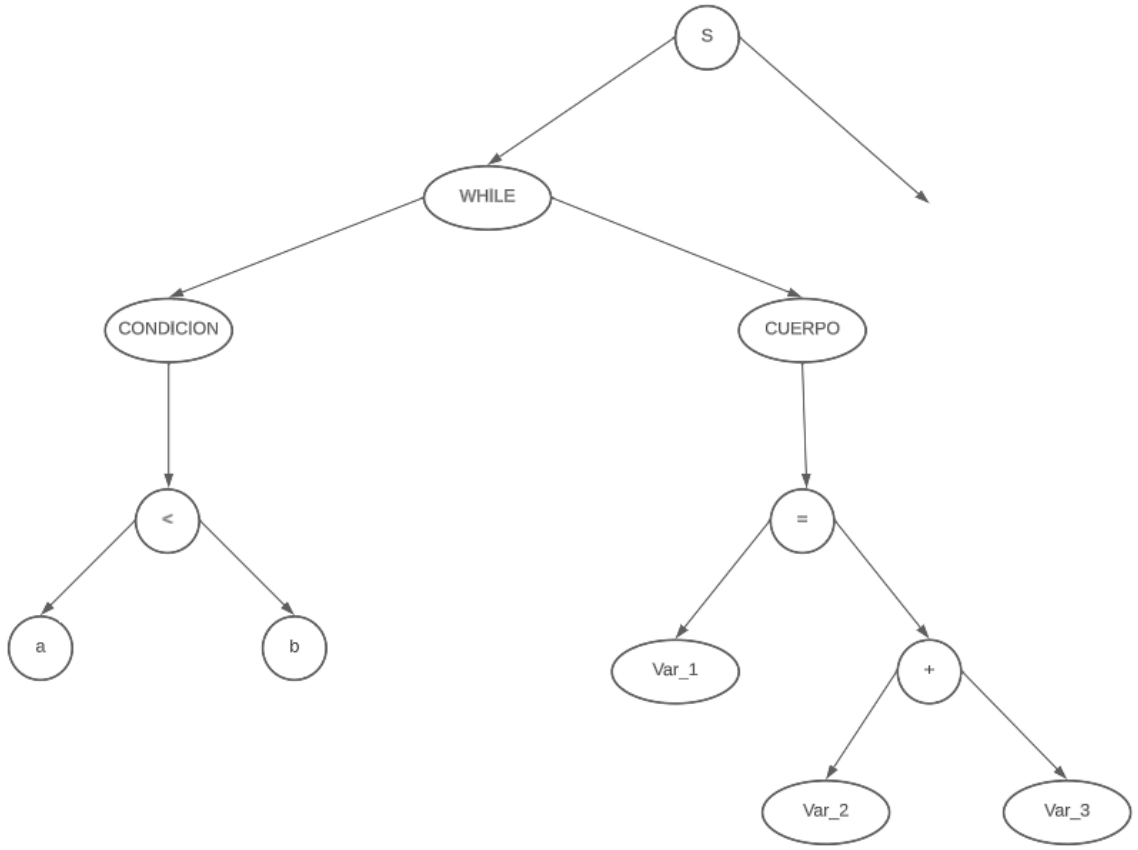


Generación de subárbol de sentencia IF:



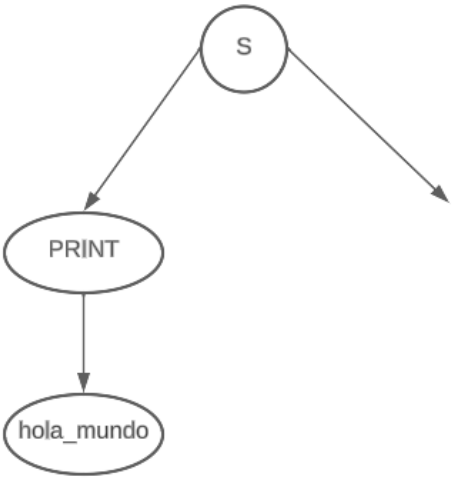
Para la generación de un subÁrbol IF es necesario utilizar nodos de control para las bifurcaciones, tanto para la condición y el cuerpo, como para el bloque then y el se del mismo.
Los pasos son:
1°) Género el subÁrbol de la condición
2°) Conecto ese subÁrbol a un nodo de control “condición”
3°) Creó el cuerpo del IF, genero el subárbol para la rama del then y el subÁrbol para la rama del else.
4°) Uno cada subÁrbol a su respectivo nodo de control, then o else, los cuales serán utilizados luego para la generación de código assembler
5°) Una vez generado ambas ramas del cuerpo, se conectan con un nodo control “cuerpo”
6°) Por último, la subRama “condición” y la subRama “cuerpo” se conectan al nodo de control “IF”.
7°) Conectar el nodo “IF” al hijo izquierdo de un nodo sentencias del árbol general.

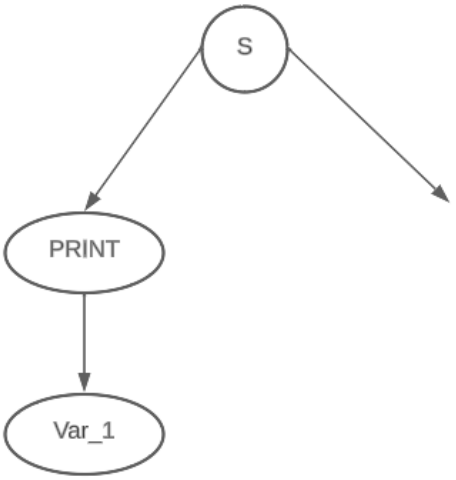
Generación de SubÁrbol de sentencia While:



Para la generación de un subÁrbol IF es necesario utilizar nodos de control para las bifurcaciones, tanto para la condición como para el cuerpo.
Los pasos para su generación son:
1°) Género el subÁrbol de la condición
2°) Conecto ese subÁrbol a un nodo de control “condición”
3°) Género el subÁrbol del cuerpo
4°) Conectó ese subÁrbol a un nodo de control “cuerpo”
5°) Generar un nodo de control “WHILE” y conectar los subÁrboles generados al mismo.
6°) Conectar el nodo “WHILE” a un nodo sentencia del programa

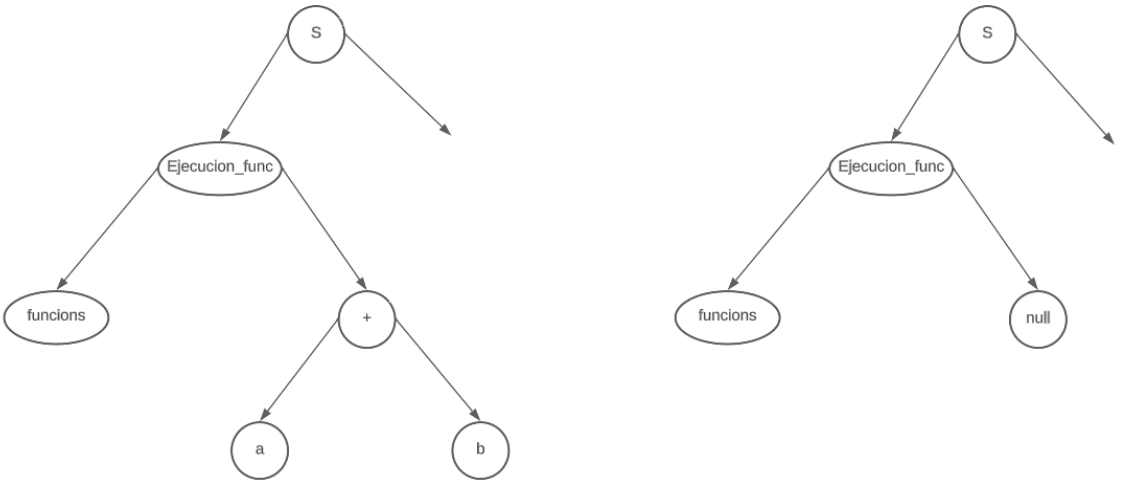
Generación de subÁrbol de sentencia PRINT:





Los pasos para su generación son:
1°) Género el nodo con la cadena
2°) Lo conecto a un nodo de control “PRINT”, el cual será utilizado luego para generar el código assembler
3°) Conectar el nodo PRINT creado a un nodo “sentencia” del programa

Generación de subÁrbol de sentencia ejecución Funcion:



Para la ejecución de las funciones decidimos crearlo de la siguiente manera:
1°) Generamos el subÁrbol del parámetro en caso de tenerlo y el subÁrbol del nombre de la función
2°) Se conectan a un nodo de control “ejecución func”
3°) Se conectan a un nodo “sentencia” del programa

Notación posicional de Yacc

La notación posicional Yacc, la utilizamos para:

- Generar los nodos del árbol
- Verificar que las sentencias detectadas no contienen ningún error sintáctico o semántico
- Para asignarles el identificador a los nodos hoja o para realizar modificaciones en la tabla de símbolos.

Con $n nos posicionamos en el token que nos interesa de la sentencia para obtener su valor, por ejemplo, para la regla:

```
-sentencias_ejecucion_funcion: ID '(' expr_aritmetic ')'
```

Si quiero obtener el valor de ID, utilizó la regla $1, que hace referencia a su primer valor.

En nuestro caso $n podia tomar el valor de un String, de un nodo de ÁrbolSintáctico o de un Símbolo, dependiendo de los valores retornados por las sentencias anteriores.

Con $$ la regla retorna el valor que le asignemos, ya sea el valor de un token haciendo $$=$1 por ejemplo, $$ = simbolo1 o $$ = nodoArbol. Retornamos esos valores para luego ser utilizados en reglas posteriores.

Consideración y decisiones tomadas con respecto a los errores

En esta etapa del compilador detectamos errores semánticos, tales como tipos de variables incorrectas, variables no declaradas, etc. Dichos errores son detectados en el procesamiento de las reglas. En caso de encontrar un error semántico o sintáctico,

decidimos utilizar un “nodoError” constante. El mismo es retornado en caso de ocurrido un error y agregado al árbol. Esta acción no genera que se creen nodos innecesarios en el árbol ya que el mismo es constante y siempre que se detecta un error, se hace referencia al mismo nodo. El compilador sigue ejecutando aunque encuentre un error y se almacena al igual que los errores sintácticos en una lista de Strings. Si se detectó un error de una etapa anterior, por ejemplo, sintáctico o léxico, es probable que no se detecten de manera correcta los errores semánticos detectados en esta etapa.

Los errores semántico son mostrados al final de la ejecución del compilador. También a los símbolos se les agregó el atributo de “Usada”, el cual es utilizado una vez finalizada la compilación, para mostrar qué variables fueron usadas en el lado derecho de una asignación o en una comparación.

Comprobaciones Semánticas

Las comprobaciones semánticas las fuimos realizando a la misma vez que las comprobaciones sintácticas. En el procesamiento de cada regla, se verifican todos los errores semánticos propuestos por la cátedra y además incluimos el error semántico de las variables no inicializadas.

Salida Generacion de Codigo Intermedio

La salida para el código intermedio es El árbol sintáctico y una lista de errores semánticos. para el siguiente código:

```
{
        LONG num1,
       LONG var1;var2,
       UINT _var3,
       FLOAT _var4;var5,

       var1 = 10_l,
       var2 = 25_l,
       _var3 = 5_ui,
       _var4 = 10.52,
       var5 = 2.0e+5,

       VOID f1(){
              var1 = 5_l+var2,
              IF(var1 !! 100_l){
                     var1 = var1 - 10_l,
              }ELSE{
                     var1 = var1 * 2_l,
              }END_IF,
              RETURN,
       },

       PRINT %el valor de _var3 es%,

       *{esto es un comentario}*
```

```
        WHILE(_var3 <= 1000_ui)DO{
                _var3 = _var3 - 1_ui,
}
```

La salida generada es:

```
ARBOL:
-Lexema: Programa
-   Lexama Nodo: Sentencia
-   Hijo Izquierdo:
-      Lexama Nodo: =
-      Hijo Izquierdo:
-         Lexema Nodo Hoja: var1#global
-      Hijo Derecho:
-         Lexema Nodo Hoja: 10_l
-   Hijo Derecho:
-      Lexama Nodo: Sentencia
-      Hijo Izquierdo:
-         Lexama Nodo: =
-         Hijo Izquierdo:
-            Lexema Nodo Hoja: var2#global
-         Hijo Derecho:
-            Lexema Nodo Hoja: 25_l
-      Hijo Derecho:
-         Lexama Nodo: Sentencia
-         Hijo Izquierdo:
-            Lexama Nodo: =
-            Hijo Izquierdo:
-               Lexema Nodo Hoja: _var3#global
-            Hijo Derecho:
-               Lexema Nodo Hoja: 5_ui
-         Hijo Derecho:
-            Lexama Nodo: Sentencia
-            Hijo Izquierdo:
-               Lexama Nodo: =
-               Hijo Izquierdo:
-                  Lexema Nodo Hoja: _var4#global
-               Hijo Derecho:
-                  Lexema Nodo Hoja: 10.52
-            Hijo Derecho:
-               Lexama Nodo: Sentencia
-               Hijo Izquierdo:
-                  Lexama Nodo: =
-                  Hijo Izquierdo:
-                     Lexema Nodo Hoja: var5#global
-                  Hijo Derecho:
-                     Lexema Nodo Hoja: 2.0e+5
-               Hijo Derecho:
-                  Lexama Nodo: Sentencia
-                  Hijo Izquierdo:
-                     Lexema: PRINT
```

```
-                       Lexema Nodo Hoja: el valor de _var3 es
-                   Hijo Derecho:
-                     Lexama Nodo: Sentencia
-                     Hijo Izquierdo:
-                       Lexama Nodo: WHILE
-                       Hijo Izquierdo:
-                         Lexema: condicion_while
-                           Lexama Nodo: <=
-                           Hijo Izquierdo:
-                             Lexema Nodo Hoja: _var3#global
-                           Hijo Derecho:
-                             Lexema Nodo Hoja: 1000_ui
-                       Hijo Derecho:
-                         Lexema: cuerpo_while
-                           Lexama Nodo: =
-                           Hijo Izquierdo:
-                             Lexema Nodo Hoja: _var3#global
-                           Hijo Derecho:
-                             Lexama Nodo: -
-                             Hijo Izquierdo:
-                               Lexema Nodo Hoja: _var3#global
-                             Hijo Derecho:
-                               Lexema Nodo Hoja: 1_ui

WARNINGS:
Variable num1#global no usada
Variable var5#global no usada
Variable _var4#global no usada


Errores Semanticos:
No hubo ningún error Semántico
```

Generación de la Salida

Generacion de Codigo Assembler

La generación de código assembler es la última etapa del compilador. Esta se encarga de convertir la información obtenida en la tabla de símbolos, así como el árbol que generamos para el assembler.

El código assembler lo generamos a partir del árbol y la tabla de símbolos una vez finalizada su creación. Si no ocurre ningún error léxico, sintáctico y semántico, generamos el código assembler. Para generarlo, creamos una función “getAssembler” en los nodos del árbol, la cual se encarga a partir del lexema, tipo y uso que contiene el nodo, elegir qué comandos assembler agregar a la salida. A su vez, creamos una clase “GeneradorAssembler” la cual se encarga de armar la estructura de la salida del compilador,

convirtiendo los datos de la tabla de símbolos en formato assembler, agregando el encabezado y la parte final.

Operaciones Aritméticas

Tanto las operaciones aritméticas como las comparaciones las fuimos generando a partir de if anidados, los cuales dependiendo de las características del nodo y de sus hijos agregaba unos comandos u otros a la salida Assembler. Por ejemplo, si el nodo detectado es una suma en la cual ambos hijos son identificadores o constantes, el código assembler generado será:

MOV EAX , “variable”
ADD EAX , “variable2”
MOV @aux , EAX

Los cambios realizados a los identificadores de las variables para un correcto funcionamiento del assembler fueron las siguientes:
-Identificadores, constantes y cadenas:Se le agregó al principio de cada identificador el signo “$”, para diferenciarlo de las variables auxiliares. Reemplazamos “#” por “$”, ya que el lenguaje ensamblador no reconoce “#” como un carácter válido. También reemplazamos “.” por “_” ya qué punto es utilizado para otras funciones dentro del assembler. Por último si el identificador contiene un “+” o un “-” se reemplaza por “$”, para resolver conflictos.

Generación de las etiquetas destino y variables auxiliares

Para generar variables auxiliares, utilizamos una pila, la cual lleva los auxiliares utilizados. En el tope de la pila se encuentra la última variable asignada y la cual vamos a usar probablemente en la próxima instrucción. Las variables las armamos concatenando el string “@aux+el indiceAux” que lleva el índice del último auxiliar asignado.

Para el manejo de las etiquetas, también utilizamos una pila de labels, la cual se maneja de la misma manera que la pila de auxiliares. Las etiquetas las usamos para saltos de comparaciones y llamados a funciones.

Salida Assembler

La salida que proporciona el compilador al terminar su ejecución es la siguiente:
Para el código:

```
{
	LONG var1,

	FLOAT var2;var3,

	var1 = 20_l,

	var2 = 10.,

	var3 = TOF(var1) + var2,
```

```
	IF(var3 == 30.0){
		PRINT %var3 es igual a 30.0%,
	}ELSE{
		PRINT %var3 es distinto a 30.0%,
	}END_IF,

	var1 = var1 + 1_l,

	IF(var1 == 21_l){
		PRINT %var1 es igual a 21%,
	}ELSE{
		PRINT %var1 es distinto de 21%,
	}END_IF,
}
```

La salida generada es:

```
.386
.model flat, stdcall
option casemap :none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
.data
$errorDivisionPorCero db " Error Assembler:No se puede realizar la division por
cero"
$errorOverflowMultEntero db " Error Assembler: overflow en producto de enteros "
$errorOverflowSumaFlotantes db " Error Assembler: overflow en la suma de flotantes
"
$30_0 dd 30.0
$var3$global dd ?
$var1$es$distinto$de$21 db "var1 es distinto de 21" , 0
$var1$es$igual$a$21 db "var1 es igual a 21" , 0
$var2$global dd ?
$var3$es$igual$a$30_0 db "var3 es igual a 30.0" , 0
@aux2 dd ?
@aux1 dd ?
$20_l dd 20
$var1$global dd ?
$21_l dd 21
$1_l dd 1
$10_ dd 10.
$var3$es$distinto$a$30_0 db "var3 es distinto a 30.0" , 0
.code
main:
MOV EAX , $20_l
MOV $var1$global, EAX
FLD $10_
FST $var2$global
FILD $var1$global
FLD @aux1
FST $var3$global
```

```
FLD $var3$global
FCOM $30_0
FSTSW AX
SAHF
JNE label1
invoke MessageBox, NULL, addr $var3$es$igual$a$30_0, addr $var3$es$igual$a$30_0,
MB_OK
JMP label2
label1:
invoke MessageBox, NULL, addr $var3$es$distinto$a$30_0, addr
$var3$es$distinto$a$30_0, MB_OK
label2:
MOV EAX , $var1$global
ADD EAX , $1_l
MOV @aux2 , EAX
MOV EAX , @aux2
MOV $var1$global , EAX
MOV EAX , $var1$global
CMP EAX , $21_l
JNE label3
invoke MessageBox, NULL, addr $var1$es$igual$a$21, addr $var1$es$igual$a$21, MB_OK
JMP label4
label3:
invoke MessageBox, NULL, addr $var1$es$distinto$de$21, addr
$var1$es$distinto$de$21, MB_OK
label4:
invoke ExitProcess, 0
errorDivisionPorCero:
invoke MessageBox, NULL, addr $errorDivisionPorCero , addr $errorDivisionPorCero ,
MB_OK
invoke ExitProcess, 0
errorOverflowMultEntero:
invoke MessageBox, NULL, addr $errorOverflowMultEntero , addr
$errorOverflowMultEntero , MB_OK
invoke ExitProcess, 0
errorOverflowSumaFlotantes:
invoke MessageBox, NULL, addr $errorOverflowSumaFlotantes , addr
$errorOverflowSumaFlotantes , MB_OK
invoke ExitProcess, 0
end main
```

Consideración y decisiones tomadas con respecto a los errores

Léxico

Los errores Léxicos se manejan en esta etapa, exceptuando el rango de las constantes negativas, las cuales son tratadas en el análisis sintáctico ya que es imposible detectar si una constante es positiva o negativa en esta parte.

Los errores tratados son:

- Identificadores con una longitud mayor a la permitida: Los identificadores tienen un máximo de longitud de 20 caracteres. En caso de superarse esta cantidad, el token se trunca y se eliminan los últimos caracteres sobrantes. Esto es tomado como un warning ya que el código puede seguir compilando y es posible su ejecución pero es probable que no realice la funcionalidad que estaba planeada en un principio. Ya que al recortar el nombre, este puede quedar idéntico a otro Identificador y eso significa que estamos trabajando sobre una variable que no corresponde. También es posible que el código ejecute de manera correcta aunque se recorte el nombre, por esa razón se lo considera una advertencia y se le permite terminar de compilarla para luego ejecutarse y no terminar la compilación por error.
- Errores de rango en las constantes: Las constantes ya sean enteros largos, enteros sin signo o flotantes, tienen un rango tanto superior como inferior. Este mismo no puede ser superado. Si alguna constante tiene un rango mayor o menor, se considera un error y se añade a la lista de errores léxicos. Ya que no es posible reemplazar el valor como en los identificadores porque cambiar el valor de una constante si puede afectar de manera significativa el funcionamiento del programa, sea cual sea el mismo.
- Si se encuentra con la siguiente cadena “FLOATidentificador”, no se cuenta como error. El Analizador lo toma como confusión del programador y separa los tokens.

Cualquier error léxico es almacenado en la Lista de errores Léxicos que se encuentra en la clase Analizador Léxico. Al detectarse cualquiera de estos errores, el compilador debe seguir su curso, por lo tanto la sentencia en la que es detectado el error léxico se descarta, es decir que el AnalizadorLexico avanza caracteres hasta encontrarse con una ‘,’ la cual utilizamos como valor de restablecimiento. Todo lo anterior se descarta y se sigue compilando a partir de ese punto. Esto puede ocasionar que en la etapa siguiente, es decir el analizadorSintactico, no se detecten correctamente las estructuras posteriores a la cual tuvo el error porque al descartar caracteres hasta encontrar una coma es probable que se haya descartado una parte de código que no contenía error. Es por ello que si se detecta un error Léxico, el analizador sintáctico pierde algunas sentencias posteriores al error y no las detecta correctamente.

Sintáctico

- Dentro de las funciones solo es posible realizar sentencias declarativas ya que el enunciado no menciona nada de sentencias ejecutables.
- Los identificadores no pueden contener letras mayúsculas
- Es necesario que los archivos txt sean UNIX(LF).
  Código Intermedio

Generación Código intermedio

Las comprobaciones semánticas las fuimos realizando a la misma vez que las comprobaciones sintácticas. En el procesamiento de cada regla, se verifican todos los errores semánticos propuestos por la cátedra y además incluimos el error semántico de las variables no inicializadas.

Modificaciones a las etapas anteriores

Los cambios realizados en las etapas anteriores son:

- La Matriz de transicion de estados y la matriz de acciones semanticas se modificaron para cumplir con los requerimientos de la catedra. Para ello realizamos

cambios en los estados por los que pasa un token hasta formarse y modificamos el funcionamiento de ciertas acciones semanticas.
- Realizamos cambios en la gramatica, agregando y modificando reglas para resolver la generacion de codigo intermedio de manera correcta.

Implementación de los temas particulares

- Sobreescritura de métodos: Para permitir la sobrescritura de métodos, verificamos que el ámbito de los métodos sea distinto si tienen el mismo nombre.
- Para realizar el chequeo de variable Usada, agregamos al objeto símbolo el atributo usado, el cual al encontrarse en el lado derecho de una asignación o comparación ya considero como usada la variable. Luego al finalizar la ejecución del parser recorro la lista imprimiendo las variables cuyo atributo se encuentre en false
- Forward declaration: Para resolverlo, agregamos una regla que permita esto y luego al querer utilizar un método o variable de dicha clase, verificamos si la misma ya fue declarada o su declaración se realizará con posterioridad.
- Para permitir la herencia por composición, agregamos a las sentencias declarativas de clases, la posibilidad de declarar una clase de la forma “nombre clase,” lo cual permite que la clase herede todos los métodos y variables de la clase que hereda. Esta herencia puede tener hasta tres niveles, en caso de superarse se notifica el error.

Implementación Controles en Tiempo de Ejecución

Los controles son los siguientes:

- División por 0 en enteros y flotantes: para realizar este punto la lógica utilizada fue, para los enteros Largos y enteros Sin Signo, comparamos su valor con cero, en caso de ser igual saltó a la etiqueta error y terminó la ejecución del programa. Para el caso de los flotantes, primero realizó la resta del número consigo mismo y si el resultado es cero, significa que el divisor es cero y saltó a la etiqueta de error.
- Overflow en suma de datos flotantes: Primero realizó la suma, luego muevo los flags al registro AH. Verificar si hubo overflow y saltar a la etiqueta de error en caso de que se cumpla el error.
- Overflow en producto de entero: Luego de realizar la multiplicación, verificó con la instrucción JO que no hubo overflow, en caso de haberlo salto a la etiqueta del error y culminar la ejecución.
- Para realizar la conversión explícita, al detectar un nodo TOF, utiliza la instrucción FILD para pasar el número de 16 a 32 bits y así poder realizar la operación requerida.

Ejecución del archivo Jar

El archivo .jar fue creado para 2 versiones distintas de jdk, para la 17.0.8 y para la 11. Ambos archivos están subidos al drive para que sea utilizado el correspondiente.

VERSIÓN DEL JDK:

JAVA_RUNTIME_VERSION="17.0.8+9-LTS-211"
JAVA_VERSION="17.0.8"
JAVA_VERSION_DATE="2023-07-18"

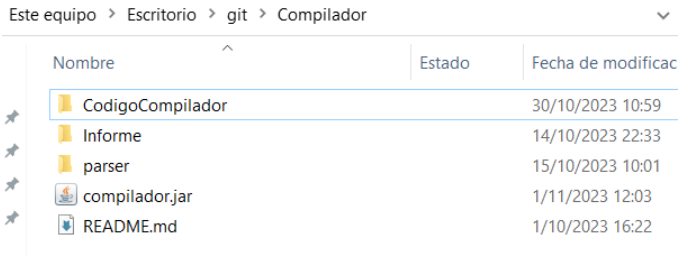
VERSIÓN DEL ECLIPSE:

Version: 2023-09 (4.29.0)

Para para poder ejecutar el jar es necesario seguir los siguientes pasos:

1°) abrir una consola de comandos de windows(cmd)

2°) pararte en la carpeta en la que se encuentra el archivo compilador.jar utilizando el comando cd “direccion de la carpeta” (en este caso es en la carpeta …/compilador)



3°) Una vez parado en la carpeta ejecutamos el siguiente comando:

“java -jar compilador.jar”

4°)En caso de que no ocurra ningún error tendría que aparecer un mensaje en la consola de la siguiente forma:

```
La carpeta a partir de la cual el compilador toma el valor ingresado es \CodigoCompilador\src\Testeos\
Ejemplo de dato a ingresar: [CPP_general.txt]   o     Sintactico\CPP_general.txt

Ingrese la direccion del archivo a compilar:
```

La manera de ingresar la dirección de un código para ser compilado es, por ejemplo, si el archivo txt se encuentra en “C:\Users\rolus\OneDrive\Escritorio\git\Compilador\CodigoCompilador\src\Testeos\lexico. LX_pruebaConErrores.txt”, El dato que debo ingresar al compilador es: lexico\LX_pruebaConErrores.txt.

Para probar cada código es necesario volver a ejecutar los pasos anteriores.

Conclusiones

Durante el desarrollo del trabajo práctico pudimos comprender el funcionamiento del léxico y el sintáctico de un compilador. Como un compilador analiza un código y a partir de este genera tokens e identifica sentencias que luego van a ser usadas en etapas posteriores. Como son detectados los errores léxicos y sintácticos y una idea general de cómo funciona un compilador hasta este punto. En las etapas 3 y 4 pudimos ver e implementar como un compilador a partir de la gramática y los tokens generados por las etapas anteriores genera las estructuras necesarias para luego poder generar el código assembler que se ejecutará en el procesador para obtener los resultados deseados con el código que le proporcionamos. Pudimos ver los errores semánticos y en tiempo de

ejecución y cómo los detectamos. También nos encontramos con dificultades las cuales pudimos resolver pensando las cosas de diferente manera. Cómo a partir de un código el compilador se encarga de analizar cada palabra y verificar que todo sea correcto, llegando a ejecutar y realizar las operaciones proporcionadas en el código de prueba.