TEC | Tecnológico de Costa Rica

29 DE MAYO DE 2023

PROYECTO 2

ANÁLISIS SEMÁNTICO Y GENERACIÓN DE CÓDIGO INTERMEDIO

JEAN HUNT

COMPILADORES E INTERPRETES

Ingeniería en computación

Tabla de contenidos

Tabla de figuras

1. Manual de usuario

A continuación, se detallan los pasos para la compilación y ejecución del proyecto además de pruebas de su funcionalidad.

1) Descargue o clone el repositorio del proyecto a la maquina en donde desea ejecutarlo. Esto se puede hacer directamente desde la página de github o utilizando el comando “git clone” con el siguiente enlace: https://github.com/JPHuntV/P2-Compiladores-IS-2023
2) Abra una terminal en la carpeta src del proyecto y ejecute los siguientes comandos para la compilación del lexer y del parser (ambos archivos han sido modificados para este proyecto):
    a. Lexer: java -jar "../lib/jflex-full-1.9.0.jar" scanner.jflex
    b. Parser: java -jar "../lib/java-cup-11b.jar" -interface parser.cup
3) El paso anterior generara los archivos “Analizador.java”, “parser.java” y “sym.java”.
4) En el archivo “Main.java” se hace referencia a del archivo que se desea analizar, asegúrese que sea la ruta correspondiente a su equipo:

```
1  package src;
2  public class main {
       Run | Debug
3      public static void main(String[] args) {
4          App app = new App();
5          app.analizar(archivoFuente:"programa/src/archivoFuente.txt");//lexer
6          app.parsear(archivoFuente:"programa/src/archivoFuente.txt");//parser
7      }
8  }
9
```

Ilustración 1:Main.java

5) Ejecute el archivo main.java haciendo uso de java. Esto lo puede hacer directamente desde su editor de preferencia.
6) La ejecución del programa generara **4 archivos**:
    a. **Lexemas.txt:** contiene todos los lexemas encontrados en el archivo fuente.Este archivo tiene una estructura similar a la siguiente:

```
1    Token: 46   valor: int
2    Token: 44   valor: test
3    Token: 6    valor: (
4    Token: 46   valor: int
5    Token: 44   valor: a
6    Token: 15   valor: ,
7    Token: 46   valor: int
8    Token: 44   valor: b
9    Token: 7    valor: )
10   Token: 13   valor: {
11   Token: 49   valor: char
12   Token: 44   valor: arrays
13   Token: 11   valor: [
14   Token: 39   valor: 2
15   Token: 12   valor: ]
```

Ilustración 2:Resultado lexer

b. **Tablasimbolos.txt:** Contiene las tablas de símbolos generadas durante el parseo.

```
3   Tabla: test  Tipo de retorno: int
4       valor: a    tipo: int
5       valor: b    tipo: int
6       valor: c    tipo: int
7       valor: t    tipo: char
8       valor: i    tipo: int
9       valor: arrays    tipo: char
10
11
12  Tabla: miFunc  Tipo de retorno: float
13      valor: dif    tipo: int
14      valor: otra    tipo: char
15      valor: str    tipo: String
16      valor: otrasS    tipo: int
17      valor: var    tipo: int
18      valor: otrase    tipo: int
19      valor: i    tipo: int
20      valor: prueba    tipo: int
21      valor: otras    tipo: int
22      valor: otrasSsd    tipo: int
23
24
25  Tabla: miOtraFun  Tipo de retorno: bool
```

Ilustración 3:Tablas de símbolos

c. **AST.txt:** Contiene el árbol sintáctico del programa escrito.

```
inicio {
  programa {
    declaraFuncion: test {
      parametros {
        int: a
        int: b
      }
      bloque {
        declaraArray: arrays {
          dataType: char
          size: 2
          array {
            init {
              literal_char {
                'a'
              }
            }
          }
        }
      }
```

Ilustración 4:Árbol sintáctico

d. **CodigoIntermedio.txt:** Contiene el código intermedio que ha sido generado en base al programa escrito.

```

test:

  dataArray arrays[2]
  t0 = 'a'
  arrays[0] = t0

  dataArray c[2]
  t1 = 1
  t2 = 2
  c[0] = t1
  c[1] = t2

  dataChar t
  t3 = 'a'
  t = t3
  t4 = 'b'
  arrays[2] = t4

_test_while1_eval:
  t5 = a
  t6 = b
  t7 = t5 > t6
  if t7 goto _test_while1_body
  goto _test_while1_end

```

Ilustración 5:Código intermedio

7) Si el programa encuentra algún error sintáctico o semántico este será reportado en consola y no generará los archivos anteriormente mencionados.

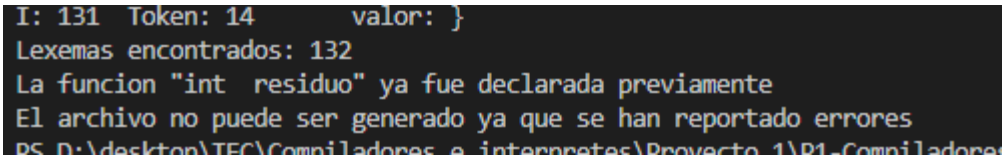

Ilustración 6: Parseo y generación fallidos

2. Descripción del problema

Existe la necesidad de desarrollar un programa informático que permita configurar chips de forma eficiente y eficaz, en una industria que se encuentra en constante crecimiento. Para lograr esto, se requiere de un programa imperativo, lo que significa que el flujo de control del programa se basa en una serie de instrucciones específicas que se ejecutan en orden secuencial.

Además, el programa debe ser ligero y potente, lo que implica que debe tener un tamaño reducido para poder ser ejecutado en sistemas con limitaciones de recursos, como los chips que se van a configurar y debe ser capaz de realizar operaciones básicas de configuración de chips de manera efectiva y eficiente.

En la industria de los chips, la configuración es una tarea crítica, ya que afecta directamente el funcionamiento del dispositivo en el que se utilizará el chip. Por lo tanto, es necesario contar con un programa que permita configurar los chips de manera precisa y confiable.

Para lograr esto, el programa debe contar con una serie de funcionalidades específicas, como la capacidad de leer y escribir datos en el chip, así como de realizar operaciones aritméticas y lógicas básicas. También es importante que el programa cuente con un sistema de gestión de errores para poder detectar y corregir cualquier problema que pueda surgir durante la configuración del chip.

Por lo que el objetivo de este proyecto está centrado en ampliar el alcance del proyecto 1, esto mediante la generación de un árbol sintáctico que será analizado de manera semántica para así generar un código intermedio.

Este código intermedio será utilizado en proyectos futuros para la generación de código objetivo.

3. Diseño del programa

El programa se puede visualizar en 6 grandes partes según el lenguaje o herramienta utilizada para su desarrollo:

1) **Lexer**: Escrito haciendo uso de la herramienta jflex. Este contiene la definición explicita de las terminales que serán utilizadas en el parser. Este documento se puede visualizar como un catalogo de todos los posibles símbolos que serán permitidos en el archivo fuente que se desea analizar, es decir, todos los lexemas.
2) **Parser**: Escrito haciendo uso de cup. Este contiene la gramática BNF que define el lenguaje y hace uso de las terminales definidas en el lexer, además de instrucciones que se deberán ejecutar al encontrar estas producciones para construir el árbol sintáctico.
3) **Analizador semántico**: Escrito en cup. Durante la construcción de las producciones estas serán analizadas para asegurar el sentido semántico del programa, es decir, compatibilidad de tipos, reglas completas, definiciones correctas etc.
4) **Generador de Código intermedio**: Escrito en java. Analizará cada uno de los nodos del árbol sintáctico para así traducir el programa a código intermedio.
5) **Aplicación**: Todos aquellos archivos escritos en lenguaje java que ayudan a la ejecución del programa. Es decir, clases, como “función”, “parámetros”, “listaparametros”, app.java y el main
6) **Resultados**: Aquellos archivos generados por el programa, tales como “lexemas.txt”, “tablasimbolos.txt”, “AST.txt”, “CodigoIntermedio.txt”
7) Además, se adjunta un archivo “test.txt” el cual fue utilizado durante el desarrollo del proyecto para la comprobación del funcionamiento del programa, el código que contiene este archivo está basado en el documento “código prueba 2.txt” el cual fue brindado por el profesor para la revisión del primer proyecto.

Para la creación del árbol sintáctico se creo la clase ASTNode, esta clase almacena el nombre y valor del nodo, así como una lista de nodos hijos de la misma clase. Esta estructura será utilizada para el análisis semántico del programa ya que cada nodo tendrá reglas definidas por la producción para sus nodos hijos, por ejemplo, un nodo “literal_int” no puede tener un nodo hijo de tipo string.

Una vez completado el análisis semántico se hace un recorrido de este árbol en donde se determina el tipo de cada uno de los nodos y su contenido para generar el código intermedio.

4. librerías utilizadas

Para el desarrollo del programa se ha hecho uso de las siguientes librerías:

a) Java-cup-11b-runtime.jar, Java-cup-11b.jar : Escritura del parser en Cup
b) Jflex-full-1.9.0.jar : Escritura del lexer en jflex
c) Java.util.list, Java.util.arraylist: Manipulación de las listas de elementos que iran en las tablas de símbolos.
d) Java.util.Map, Java.util.HashMap: Mapeo de los símbolos que van dentro de dichas listas
e) Java.util.Stack: Manipulación del stack que almacena todas las listas de símbolos en relación a su función
f) BufferedReader, FileReader: Lectura del archivo fuente
g) BufferedWriter, FileWriter: Escritura de los archivos de resultado

5. Análisis de resultados

En la siguiente tabla se muestra una lista de objetivos para los cuales se evaluará si fueron cumplidos o no, en caso de no cumplirse también se indicará la razón del porqué:

Objetivo	***Alcanzado***	***Razón***
Corrección parser y lexer	Si	
Desarrollar analizador semántico	Si	
Generar un reporte de errores si esto existen	Si	
Desarrollar generador de código intermedio	Si	
Extra: Manejo de errores	No	Priorización de otras funcionalidades requeridas
Extra: Manejo de clases	No	

En el siguiente enlace se puede encontrar el repositorio en GitHub que contiene el proyecto: https://github.com/JPHuntV/P2-Compiladores-IS-2023.git