laSalle
UNIVERSIDAD RAMON LLULL

Curso 2018 - 2019
Lenguajes de Programación

Documentación
Grupo 2

Angel Farre (angel.farre)
Jaime Parra (jaime.parra)
Joan Fitó (joan.fito)
Sergio Rodrigo (sergio.rodrigo)
Victor Garrido (victor.garrido)

laSalle
UNIVERSIDAD RAMON LLULL

Curso 2018 - 2019
Lenguajes de Programación

ÍNDICE

1 INTRODUCCIÓN

El propósito de la práctica presentada en la asignatura de Lenguajes de Programación se centra de aplicar todo el conocimiento adquirido a lo largo del curso en un programa. Este programa, escrito en java, tendrá que cumplir la función de un compilador para un lenguaje de libre contexto. Se le pasara un código en dicho lenguaje el cual el compilador tendrá que leer, analizar y ejecutar. Este lenguaje debe ser ficticio y creado por los mismos integrantes de la práctica. Debe estar compuesto por una serie de palabras o grupo de palabras, conocidos como tokens, una estructura que gobierne las formas en las que estas palabras se puedan o no combinar y una cohesión para que este lenguaje tenga un significado, sentido o interpretación válidos.

Esta se subdivide en varias fases. Antes de poder tan siquiera diseñar cualquier compilador antes se tiene que ingeniar un diccionario de palabras. El analizador lexicográfico de nuestro compilador será el encargado de identificar esas palabras clave, o alertarnos en caso de que haya algún carácter no existente o reconocible en nuestro léxico.

Una vez realizado y testeado el análisis lexicográfico pasamos a analizar la estructura en la que vienen dados estos tokens. Es a partir de nuestro lenguaje que diseñamos su estructura o gramática, representada mediante diagramas y grafos o con una terminología formal como el método BNF, la cual nuestro compilador tendrá que analizar. A esta fase se la conoce como el análisis sintáctico.

Si tanto los tokens analizados como el orden en el que vienen dados son correctos pasamos a la tercera y última parte del análisis, la semántica. Se tienen que establecer e implementar una serie de normas o reglas que den sentido y significado a estos tokens según como vengan dados.

De no haber problemas el compilador deberá pasar por un proceso para convertir este código introducido en un código más masticable que la máquina sea capaz de interpretar y ejecutar, código máquina.

2 LÉXICO

2.1 IDENTIFICADORES

Los identificadores usados para el lenguaje son *case sensitive*, es decir, hay distinción entre mayúsculas y minúsculas. Además, únicamente pueden contener letras, tanto mayúsculas como minúsculas (A-Z, a-z). Su tamaño máximo será de 30 caracteres.

2.2 CONSTANTES

Las constantes usadas para el lenguaje son valores numéricos sin signo, por lo tanto, únicamente serán valores positivos.

2.3 OPERADORES RELACIONALES

Los operadores relacionales definidos para el lenguaje son:

- == : operador relacional utilizado para comprobar si los valores que se encuentran a ambos lados del operador son iguales. Se utiliza tanto dentro de condicionales como de bucles.
- > : operador relacional que comprueba que el valor que se encuentra a la izquierda sea mayor que el de la derecha.
- < : operador relacional que comprueba que el valor que se encuentra a la izquierda sea menor que el de la derecha.

2.4 OPERADORES ARITMÉTICOS

Los operadores aritméticos definidos para el lenguaje son:

- + : operador aritmético para realizar una suma de dos valores, ya sean variables o constantes. Tiene que encontrarse en medio de ambos valores.
- - : operador aritmético para realizar una resta de dos valores, ya sean variables o constantes. Tiene que encontrarse en medio de ambos valores.
- * : operador aritmético para realizar una multiplicación de dos valores, ya sean variables o constantes. tiene que encontrarse en medio de ambos valores.
- / : operador aritmético para realizar una división de dos valores, ya sean variables o constantes. tiene que encontrarse en medio de ambos valores.

2.5 OPERADOR DE ASIGNACIÓN

El único operador de asignación implementado para nuestro lenguaje es el siguiente:

- = : operador para realizar una asignación de una constante a una variable previamente definida. La asignación es de derecha a izquierda, es decir, primero se encontrará la variable, seguido del operador y finalmente el valor constante a asignar.

2.6 TIPOS PREDEFINIDOS

Nuestro lenguaje tiene dos tipos predefinidos: **int** y **boolean**, que representan un valor entero y un booleano (true o false), respectivamente.

2.7 PALABRAS CLAVES

Las diferentes palabras claves que están reservadas son:

- **void** : indica el inicio de la definición de un procedimiento.
- **int** : permite definir una variable o una función del tipo predefinido int.
- **boolean** : permite definir una variable de tipo predefinido *boolean*, que puede adquirir los valores de *true* o *false*.
- **main** : indica que el procedimiento o función que tiene como nombre esta palabra es el principal del programa.
- **while** : sirve para realizar un bucle que ejecutará la sentencia de comandos indicada dentro de éste, mientras se cumpla la condición escrita entre paréntesis.
- **if** : sirve para ejecutar una sentencia de comandos únicamente si se cumple la condición indicada entre paréntesis.

2.8 SEPARADORES

Los espacios en blanco actúan como separador. El número de espacios es indiferente, se trata igual un espacio que múltiples espacios.

2.9 SÍMBOLOS ESPECIALES

Los diferentes símbolos especiales de nuestro lenguaje son:

- **( )** : después del identificador del procedimiento/función siempre tiene que encontrarse ambos símbolos especiales, en éste mismo orden. Puede ir precedido o no de espacios en blanco.
- { : después de los símbolos "()" tiene que encontrarse este carácter para indicar el inicio del bloque.
- } : indica el final de bloque previamente abierto con “}”
- **;** : indica el final de una sentencia

3 SINTÁCTICO

3.1 DIAGRAMAS DE CONWAY

Podemos ver la sintaxis de nuestro lenguaje en los siguientes diagramas de Conway.

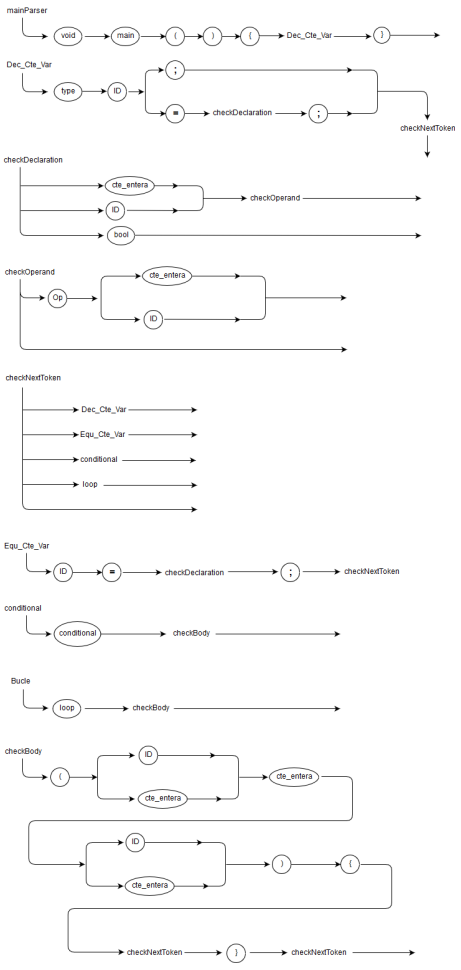

Figure 1: Diagramas de Conway

3.2 REPRESENTACIÓN BNF

Esto diagramas o grafos nos permiten representar de una forma más visual la estructura de la gramática de nuestro lenguaje. Representan una alternativa gráfica para la forma de Backus-Naur o BNF. La representación BNF seria la siguiente:

3.2.1 MAINPARSER

mainParser ⟶ <void> <main> < (> <) > < { > Dec_Cte_Var < } >

3.2.2 DEC_CTE_VAR

Dec_Cte_Var ⟶ <type> <ID> Dec_Cte_Var' checkNextToken

Dec_Cte_Var' ⟶ <;>
⟶ ⟺ checkDeclaration <;>

3.2.3 CHECKDECLARATION

checkDeclaration ⟶ <cte_Entera> checkOperand
⟶ <ID> checkOperand
⟶ <cte_booleana>

3.2.4 CHECKOPERAND

checkOperand ⟶ <Op> checkOperand'
⟶ λ

checkOperand' ⟶ <cte_entera>
⟶ <ID>

3.2.5 CHECKNEXTTOKEN

checkNextToken ⟶ Dec_Cte_Var
⟶ Equ_Cte_Var
⟶ conditional
⟶ loop
⟶ λ

3.2.6 EQU_CTE_VAR

Equu_Cte_Var ⟶ <ID> ⟺ checkDeclaration < ; > checkNextToken

3.2.7 CONDITIONAL

conditional ⟶ <conditional> checkBody

3.2.8 CHECKBODY

checkBody ⟶ < (> checkBody' <cte_entera> checkBody' <) > < { > checkNext-Token < } > checkNextToken

checkBody' ⟶ <ID>
⟶ <cte_entera>

4 SEMÁNTICO

Para la realización de esta fase, se ha creado una nueva clase, Semantic, la cual es responsable de realizar cualquier funcionalidad del semántico.

Las normas establecidas para nuestra semántica son:

1. No declarar variables conflictivas dentro de un mismo scope.
2. El tipo de los elementos usados en una expresión deben coincidir.
3. Declarar variables previo a su uso.

Para esta fase de la practica tambien diseñamos la tabla de símbolos, en la cual guardamos el nombre de la variable, su tipo y si se ha pedido memoria o no para generar el código máquina.

5 GENERACIÓN DE CÓDIGO

5.1 CÓDIGO INTERMEDIO

Para la generación del código intermedio hemos utilizado el lenguaje intermedio *Three Address Code* (TAC).

Éste se define por tener como máximo tres direcciones y es típicamente una combinación de una asignación y una operación.

Estas direcciones pueden ser el nombre de una variable, una constante o una variable temporal creada por el compilador.

Para guardar las instrucciones del *Three Address Code* hemos utilizado estructuras cuádruples, que tienen los campos siguientes:

- **Op** : operador
- **Arg1** : argumento 1
- **Arg2** : argumento 2
- **Res** : resultado

Para nuestro lenguaje, únicamente nos podemos encontrar con operaciones de asignación, de suma, resta, multiplicación y división.

La transformación de una asignación (a = 0;)a TAC se hace de la siguiente forma:

$$a \leftarrow 0$$

Por lo que en nuestra estructura de datos guardaríamos:

- **Op** : =
- **Arg1** : 0
- **Arg2** : *null*
- **Res** : a

La transformación de una operación, sea suma, resta, multiplicación o división, (por ejemplo a = 3 + b;) a TAC es la siguiente:

$$t0 \leftarrow 3 + b$$

$$a \leftarrow t0$$

Por lo que en nuestra estructura de datos guardaríamos:

- **Op** : +
- **Arg1** : 3
- **Arg2** : b
- **Res** : t0

Y luego:

- **Op** : =
- **Arg1** : t0
- **Arg2** : *null*
- **Res** : a

En el caso de los condicionales, hemos implementado únicamente el *if*. Su transformación quedaría de la siguiente forma:

$$
\begin{array}{ll}
 & \text{t1} \leftarrow a == b \\
 & \text{t2} \leftarrow not\ t1 \\
 & \text{if (t2) goto t1} \\
 & \text{a = 2} \\
\text{L1} &
\end{array}
$$

Y su estructura de datos nos quedaría de la siguiente forma:

OP	**Arg1**	**Arg2**	**Result**
==	a	b	t1
not	t1		t2
if	t2		L1
=	2		a
label			L1

Para los bucles, únicamente hemos implementado el *while*. Su transformación y su estructura quedarían de la siguiente forma:

```
L1
        t1← i < 4
        if (t1)  goto   L2
        goto L3
L2
        a = 2
        goto L1
L3
```

OP	Arg1	Arg2	Result
label			L1
<	i	4	t1
if	t1		L2
goto			L3
label			L2
=	2		a
goto			L1
label			L3

5.2 MIPS

Una vez generado el *Three Address Code*, la generación del código máquina mediante MIPS es relativamente sencilla.

Lo primero que hacemos es declarar las variables para que se les asigne memoria. Para esto, nos ayudamos de la tabla de símbolos para saber si una variable ha estado previamente declarada o no. Si no lo ha estado, la declaramos, añadiendo el prefijo *var* a su nombre.

Después pasamos a la generación de código como tal. Tenemos cinco posibles operaciones: asignación, suma, resta, multiplicación y división.

Para la asignación tenemos dos variantes, si la asignación es de una constante, cargamos el entero en un registro temporal y luego lo guardamos en la zona de memoria que hemos declarado para la variable.

Por ejemplo, si estamos realizando a = 0, las instrucciones serán las siguientes:

```
li $t0,0
sw $t0,vara
```

En cambio, si estamos asignando otra variable, deberemos cargar el valor de esta variable en un registro temporal y después guardarlo en la zona de memoria de la variable a la que queremos darle el valor. Por ejemplo, si estamos realizando a = b, las instrucciones serán las siguientes:

```
lw $t0,varb
sw $t0,vara
```

Para la suma, resta, multiplicación y división el procedimiento es análogo, cambiado la instrucción **add** para sumar por **sub** si hay que restar, **mult** si hay que multiplicar, o **div** si hay que dividir.

Igual que en la asignación, si uno de los valores es una contante la cargaremos directamente en un registro temporal, y si es otra variable, deberemos recuperar su valor de memoria para cargarlo en el registro. Después realizaremos la operación, sea **add**, **sub**, **mult** o **div** guardaremos el resultado en un tercer registro temporal y luego lo guardaremos en memoria.

Por ejemplo, si realizamos a = b + 3, las instrucciones serán las siguientes:

```
lw $t0,varb
li $t1,3
add $t2,$t0,$t1
sw $t2,vart2
lw $t0,vart2
lw $t0,vara
```

6 LÍNEAS DE FUTURO

Una vez hemos conseguido desarrollar un compilador funcional para un lenguaje muy básico, el cual admite dos tipo de datos (entero y booleano) y permite asignaciones, sumas, restas, multiplicaciones y divisiones además de condicionales (if), bucles (while) y un procedimiento principal (main), nos gustaría expandirlo hasta los siguientes objetivos:

- Añadir comentarios de código.
- Añadir bucles adicionales (for, do while...)
- Añadir más condicionales (else, switch...)
- Añadir funciones

7 CONCLUSIONES

Como hemos visto en el aparto anterior, hemos conseguido satisfactoriamente desarrollar la versión final del compilador, logrando los objetivos que nos habíamos definido.

Gracias a esto, hemos podido asimilar el funcionamiento de los bloques de un compilador (léxico, sintáctico, semántico, generación de código intermedio y de código máquina), a excepción del optimizador, dado que se ha decidido no implementarlo.

Además, también ha ayudado a entender otros conceptos vistos en clase como la tabla de símbolos o el *three address code*.

Por lo tanto, podemos concluir que la realización de la práctica ha cumplido su objetivo, que no es otro que asentar las bases de los conocimientos teóricos de la asignatura.

8 BIBLIOGRAFÍA

Compiler Design | Lexical Analysis [En línea] [Fecha de consulta:12/03/2019]
Disponible en: https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

Compiler Theory [En línea] [Fecha de consulta: 27/03/2019]
Disponible en: www.cs.um.edu.mt/~sspi3/CompTheory-006.pdf

Intermediate Code Generation [En línea] [Fecha de consulta: 18/04/2019]
Disponible en: https://cs.nyu.edu/courses/spring10/G22.2130-001/lecture9.pdf

Three Adress Code Generation [En línea] [Fecha de consulta: 21/04/2019]
Disponible en: cic.puj.edu.co/wiki/lib/exe/fetch.php?media=materias:compi:comp_sesion21_aux1.pdf

Symbol Table in Compiler [En línea] [Fecha de consulta: 22/04/2019]
Disponible en: https://www.geeksforgeeks.org/symbol-table-compiler/

Three Address Code in Compiler [En línea] [Fecha de consulta: 22/04/2019]
Disponible en: https://www.geeksforgeeks.org/three-address-code-compiler/

MIPS Loops: For, While, Do While [En línea] [Fecha de consulta:05/05/2019]
Disponible en: https://www.assemblylanguagetuts.com/mips-assembly-loops/

MIPS Decision Structures [En línea] [Fecha de consulta:05/05/2019]
Disponible en: http://www.mscs.mu.edu/~rge/cosc2200/lectures/MIPS-jumps.pdf

MIPS Instruction Reference [En línea] [Fecha de consulta: 10/05/2019]
Disponible en: http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html

Repertorio de instrucciones MIPS [En línea] [Fecha de consulta: 10/05/2019]
Disponible en: http://www2.elo.utfsm.cl/~lsb/elo311/aplicaciones/spim/repertorio.pdf