Traductor de un lenguaje procedural mediante JFlex y Java CUP

Ingeniería Técnica En Informática de Sistemas

Escuela Técnica Superior de Ingeniería de Sistemas Informáticos

Alumno: Jesús Velayos Pastrana
Tutor: Jesús López Sánchez
2017

A papá ya que esto sólo tiene sentido por y para ti.

A Vane, Daniel y Eneko porque el sentido de todo lo demás lo fijáis vosotros.

ÍNDICE

Parte I - Introducción

1. Introducción

El presente documento representa la documentación del Proyecto Fin de Carrera, en adelante PFC, “TRADUCTOR DE UN LENGUAJE PROCEDURAL MEDIANTE JFLEX Y JAVA CUP” que fue desarrollado entre los años 2005 y 2007 a pesar de que su entrega se realiza en 2017.

Este PFC constituye la implementación de un traductor de un lenguaje procedural mediante un lenguaje orientado a objetos, utilizando para ello herramientas generadoras de analizadores léxicos y sintácticos. Es de importancia destacar que el presente proyecto se basa en la implementación del lenguaje spascal que sirvió como estructura para el desarrollo de la asignatura “Compiladores e Intérpretes” de la Ingeniería Técnica de Informática de Sistemas junto a su traducción al lenguaje intermedio mpascal.

La implementación de este trabajo surge de la casi inexistencia de documentación, compiladores, o cualquier otro tipo de medio sobre la implementación de traductores/compiladores en el paradigma de la programación orientada a objetos y concretamente en Java. Cabe entonces preguntarse la razón por la cual no hace hincapié esta tecnología en la implementación de compiladores. Por esta razón y el interés de estudiar nuevas herramientas nace el desarrollo del presente proyecto.

Este proyecto tiene un carácter principalmente práctico y desarrolla la implementación en Java de un traductor para un subconjunto de Pascal a un código intermedio. Al subconjunto de Pascal se le ha denominado JSPascal (Java Subconjunto de Pascal) y su especificación la podemos encontrar en el capítulo 2 del presente documento. El lenguaje intermedio recibe el nombre de JMPascal (Java Máquina Pascal).

El proyecto tiene dos características básicas que le destacan y diferencian de otros proyectos ya existentes:

- El estudio e implementación de un traductor e intérprete mediante un lenguaje orientado a objetos como es Java.
- El uso de las herramientas libres y gratuitas para la generación de analizadores léxicos y sintácticos en Java: Jflex y Java CUP.

Sobre la primera característica se verá que no es sencillo realizar una implementación totalmente orientada a objetos y que si se realiza es a costa de un código no totalmente eficaz que se basa en la captura de múltiples excepciones.

Sobre la segunda se expondrá el uso de dos herramientas muy potentes que generan código Java.

La primera de ellas es Jflex basada en el programa no libre JLex basado a su vez en el antiguo programa lex de Unix. Este programa genera el código Java de un analizador léxico, scanner, a partir de un fichero de especificación.

La segunda es Java CUP que se basa en el programa para Unix yacc. Esta herramienta genera el código Java necesario para la implementación de un analizador sintáctico, parser, a partir de un fichero de especificación.

Ambos programas admiten la inclusión de acciones que permiten ante todo la inclusión de comprobaciones semánticas en el análisis sintáctico.

La implementación del traductor requiere de análisis léxico, sintáctico y semántico, los cuales se realizan mediante los analizadores que se generan con las herramientas citadas.

Cabe destacar que la finalidad de este proyecto es fundamentalmente:

- académica para que pueda ser usado por otros alumnos en el estudio de asignaturas relacionados con el mundo de los traductores, compiladores e intérpretes
- poder validar el uso de lenguajes de alto nivel orientados a objetos como es Java en el desarrollo de compiladores, traductores e intérpretes

2. Abstract

This Final Project is an implementation of a translator of procedural language developed by an Object Oriented language and tools to generate lexical and syntactic analyzers. This project embraces the language spascal, language that was the base for the subject “Compiladores e Intérpretes” in Ingeniería Técnica de Informática de Sistemas.

This project was born because there are not documentation, compilers or translators developed under Object Oriented model and specifically with Java, so this presented a challenge.

This project is mainly practical and develops the implementation in Java of a translator for a subset of Pascal to an intermediate language. The subset of Pascal is called JSPascal (Java Subset of Pascal) and the intermediate language is called JMPascal (Java Machine Pascal).

This project has two features and it make itself different from other projects:

- To study and develop a translator with Java
- To study and use the free and open source tools Jflex and Java CUP

Jflex is based in privative JLex software that is based in old lex software from Unix. It generates Java code for a lexical analyzer, scanner, from a file with an Jflex specification.

Java CUP is based in the old software for Unix yacc. This tool generates Java code for a Java syntactic analyzer from a Java CUP specification file.

Both of them allow to include code to do semantic analysis or other actions.

It’s important to highlight that this project has two purposes:

- educational to subjects related with translators, compilers and interpreters
- to validate the use of oriented object languages as Java in translators and compilers development

3. Especificación del lenguaje JSPascal

El lenguaje JSPascal implementado es un subconjunto de Pascal, que permite escribir programas en lenguaje Pascal con ciertas limitaciones léxicas, sintácticas y semánticas.

El lenguaje JSPascal tiene una serie de características generales que definen las posibilidades que tendrá el usuario final para programar en JSPascal utilizando como compilador e intérprete el programa implementado en el presente proyecto. Dichas características se listan a continuación:

- Definición de constantes nominativas de tipo entero y real.
- Definición de literales nominativos.
- Definición y uso de tablas de una dimensión mediante la sentencia de declaración **ARRAY**.
- Definición y uso de registros con al menos un campo mediante la sentencia de declaración **RECORD**.
- Implementación de los tipos primitivos entero, lógico y real representados por los identificadores predefinidos **BOOLEAN**, **INTEGER** y **REAL**.
- Implementación de los procedimientos predefinidos **READ** y **WRITE** para los tipos predefinidos implementados.
- Implementación del procedimiento predefinido **WRITE** para los literales.
- Implementación de procedimientos con parámetros por valor y por referencia.
- Implementación de la sentencia condicional **IF-ELSE**.

- Implementación de los bucles **WHILE** y **REPEAT**.
- Implementación de los operadores condicionales: **>**, **<**, **<=**, **>=**, **=** y **<>**.
- Implementación de los operadores aritméticos: **+**, **-**, *****, **DIV** y **MOD**.
- Implementación de los operadores lógicos: **AND**, **OR** y **NOT**.
- Se permite el uso de la recursividad.
- Se permite el anidamiento de subprogramas.

La implementación sin embargo tiene las siguientes limitaciones: no se implementan funciones, no se admiten registros con variantes, no se implementa la sentencia de selección múltiple case, no se implementan conjuntos, no se implementan las sentencias with y forward, no se implementan punteros y no se implementan ficheros.

3.1. Especificación Léxica

A continuación se muestran las especificaciones léxicas del lenguaje JSPascal:

- El lenguaje es de notación libre, es decir, no existen limitaciones en cuanto a columnas, espacios o líneas en blanco se refiere.
- Los espacios en blanco, tabuladores y saltos de línea hacen las veces de separador.
- Existen un total de 26 palabras reservadas cuyo listado se puede ver a continuación:

- AND
- ARRAY
- BEGIN
- CONST
- DIV
- DO
- DOWNTO
- ELSE
- END
- FOR
- FUNCTION
- IF
- MOD
- NOT
- OF
- OR
- PROCEDURE
- PROGRAM
- RECORD
- REPEAT
- THEN
- TO
- TYPE
- UNTIL
- VAR
- WHILE

- Los identificadores de todo el lenguaje deben comenzar por una letra del alfabeto inglés seguida de cero o más dígitos o letras del mismo alfabeto.
- El alfabeto del lenguaje además incluye los siguientes caracteres:

- .
- :
- ,
- ;
- [
-]
- (
-)
- <
- >
- =
- +
- -
- *

- Los constantes enteras se situarán en el rango definido por las constantes –65535 y 65535, que son los valores que se representan con dos bytes. Este valor está representado por la constante predefinida **MAXINT.**
- Las constantes reales sólo admiten la notación decimal y no la exponencial. Por limitación de la implementación la parte entera admite números que se encuentren en el rango de los enteros. La parte decimal sin embargo se limita por poder definirse con un máximo de 6 dígitos.
- No se distinguen mayúsculas y minúsculas tanto en las palabras reservadas como en los identificadores.
- Se pueden definir comentarios mediante los símbolos (* *) o { }. Dichos comentarios no pueden estar anidados y pueden contener cualquier carácter del alfabeto que se esté utilizando. Un comentario hace las veces de separador.

Se observa que en el conjunto de palabras reservadas se relacionan palabras de funcionalidades no recogidas en el lenguaje JSPascal como es el ejemplo de la palabra reservada FUNCTION. Su integración en la especificación léxica es debido a que en la implementación sí se contemplarán dando únicamente con el fin de notificar al usuario que el uso de dicha funcionalidad no está soportado.

3.2 Especificación sintáctica

La especificación sintáctica queda determinada por las características generales del lenguaje JSPascal descritas en el capítulo 2 del presente documento que definen las posibles instrucciones que se pueden utilizar.

La sintaxis queda determinada por la gramática sintáctica siguiente, donde los terminales son piezas sintácticas:

```
<program> ::= prPROGRAM pID sPUNTOCOMA <bloque> sPUNTO
<bloque> ::= <declaracion> <grupoSent>
<declaracion> ::= <defCte> <defTipo> <defVar> <defSubpr>
<defCte> ::= prCONST <constantes>
| ε
<constantes> ::= <constantes> <constante> sPUNTOCOMA
| <constante> sPUNTOCOMA
<constante> ::= pID sIGUAL <expCte>
<expCte> ::= <expLiteral>
| <expresionCte>
<expLiteral> ::= <cadena>
<expresionCte> ::= <expresionCte> <opRel> <exprSimpleCte>
| <exprSimpleCte>
<exprSimpleCte> ::= <exprSimpleCte> <opAdt> <terminoCte>
| <signo> <terminoCte>
| <terminoCte>
<terminoCte> ::= <terminoCte> <opMult> <factorCte>
| <factorCte>
```

<factorCte> ::= **pCteEntera**

| **pCteReal**

| **pID**

| **sPARAPER** <expresionCte> **sPARACIE**

<defTipo> ::= **prTYPE** <tipos>

| **ε**

<tipos> ::= <tipos> <tipo> **sPUNTOCOMA**

| <tipo> **sPUNTOCOMA**

<tipo> ::= **pID sIGUAL** <espTipo>

<espTipo> ::= <tipoEnum>

| <tipoTab>

| <tipoReg>

<tipoEnum> ::= **sPARAPER** <listaID> **sPARCIE**

<tipoTab> ::= **prARRAY sCORAPER** <defSubrango> **sCORCIE prOF pID**

<defSubrango> ::= <exprSimpleCte> **sSUBRANGO** <exprSimpleCte>

| **pID**

<tipoReg> ::= **prRECORD** <listaCampos> **prEND**

<listaCampos> ::= <listaCampos> **sPUNTOCOMA** <listaVar>

| <listaVar>

<defVar> ::= **prVAR** <listasVar>

| **ε**

<listasVar> ::= <listasVar> <listaVar> **sPUNTOCOMA**

| <listaVar> **sPUNTOCOMA**

<defSubpr> ::= <defSubpr> <defProc> **sPUNTOCOMA**

| **ε**

<defProc> ::= **prPROCEDURE pID** <parametros> **sPUNTOCOMA** <bloque>

<parametros> ::= **sPARAPER** <listasParam> **sPARCIE**

| **ε**

<listasParam> ::= <listasParam> **sPUNTOCOMA** <listaParam>

| <listaParam>

<listaParam> ::= **prVAR** <listaVar>

| <listaVar>

<listaVar> ::= <listaID> **sDOSPUNTOS pID**

<listaID> ::= <listaID> **sCOMA pID**

| **pID**

<grupoSent> ::= **prBEGIN** <sentencias> **prEND**

<sentencias> ::= <sentencias> **sPUNTOCOMA** <sentencia>

| <sentencia>

<sentencia> ::= <condicional>

| <asignLlamada>

| <bucle>

| <grupoSent>

| **ε**

<condicional> ::= **prIF** <expresion> **prTHEN** <sentencia> <restoIF>

<restoIF> ::= **prELSE** <sentencia>

| **ε**

<asignLlamada> ::= **pID** <restoAsign>

<restoAsign> ::= <restoNombre> **sASIGNACION** <expresion>

| **sCORAPER** <paramLlamada> **sCORCIE**

| **ε**

<paramLlamada> ::= <paramLlamada> **sCOMA** <expresion>

| <expresion>

<bucle> ::= <bucleWHILE>

| <bucleREPEAT>

<bucleWHILE> ::= **prWHILE** <expresion> **prDO** <sentencia>

<bucleREPEAT> ::= **prREPEAT** <sentencias> **prUNTIL** <expresion>

<expresion> ::= <expresion> <opRel> <exprSimple>

| <exprSimple>

<exprSimple> ::= <exprSimple> <opAdt> <termino>

| <signo> <termino>

| <termino>

<termino> ::= <termino> <opMult> <factor>

| <factor>

<factor> ::= <pCteEntera>

| <pCteReal>

| **pID** <restoNombre>

| **prNOT** <factor>

| **sCORAPER** <expresion> **sCORCIE**

<opRel> ::= **sMAYOR**

| **sMAYORIGU**

| **sMENOR**

| **sMENORIGU**

| **sIGUAL**

| **sDISTINTO**

<opAdt> ::= **sMAS**

| **sMENOS**

| **prOR**

<opMult> ::= **sPOR**

| **prMOD**

| **prDIV**

| **prAND**

```
<signo> ::= sMAS
| sMENOS
<restoNombre> ::= <selector>
| ε
selector ::= sCORAPER <expresion> sCORCIE
| sPUNTO pID
```

3.3 Especificación semántica

La especificación semántica es la más compleja en lo que a implementación se refiere. Las características que la definen son:

- Los tipos predefinidos se declaran mediante identificadores predefinidos y no mediante palabras reservadas. Se permite así redefinir dichos identificadores.
- Existen dos procedimientos predefinidos, read y write, que sirven para leer y escribir en la entrada/salida estándar respectivamente. Dichos procedimientos deben tener un único parámetro. El parámetro ha de ser una variable en el caso del procedimiento read y una expresión o identificador de literal en el caso del procedimiento write. Los identificadores de dichos procedimientos se almacenan como identificadores predefinidos por lo que se pueden redefinir.
- MAXINT es una constante predefinida que marca el máximo y mínimo valor que se puede utilizar en constantes, expresiones o variables.

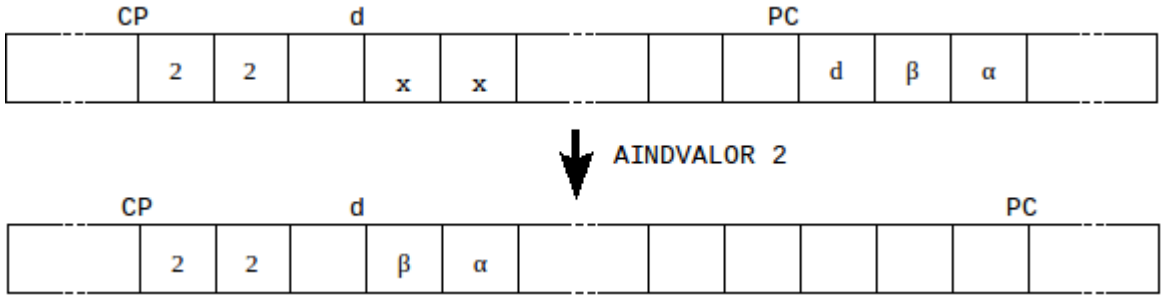
- Los valores reales de constantes, variables o expresiones tendrán en la parte entera los valores que se puede adoptar un valor de tipo entero y en la parte decimal tendrá cualquier valor que se pueda especificar con 6 decimales.
- Las asignaciones deben realizarse entre variables/campos de tipos primitivos no pudiendo realizarse asignación por ejemplo entre variables de tipo array o registros.
- Respecto a los tipos se destacan las siguientes normas:
    - En las asignaciones debe existir igualdad de tipos entre la variable a la que se asigna el valor y el valor excepto si se asigna un valor entero a una variable de tipo real.
    - En las expresiones aritméticas que se mezclen valores enteros y reales se obtendrán siempre un valor real.
    - En la declaración de literales no se pueden realizar concatenaciones lo que también queda determinado por la gramática sintáctica anteriormente expuesta.
- Los campos de los registros deben ser siempre tipos primitivos.
- Los elementos de los arrays deben ser siempre de tipos primitivos.

4. Especificación del lenguaje JMPascal

A continuación se expone el detalle de cada instrucción del lenguaje intermedio JMPascal implementado.

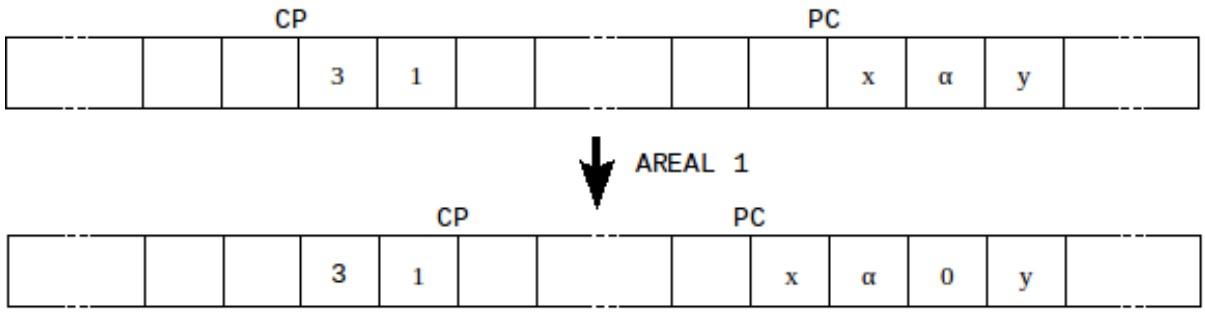
AINDVALOR w

Esta instrucción copia en la dirección que se encuentra en la cima de la pila de ejecución tantos elementos de la pila de ejecución como indica el parámetro w.



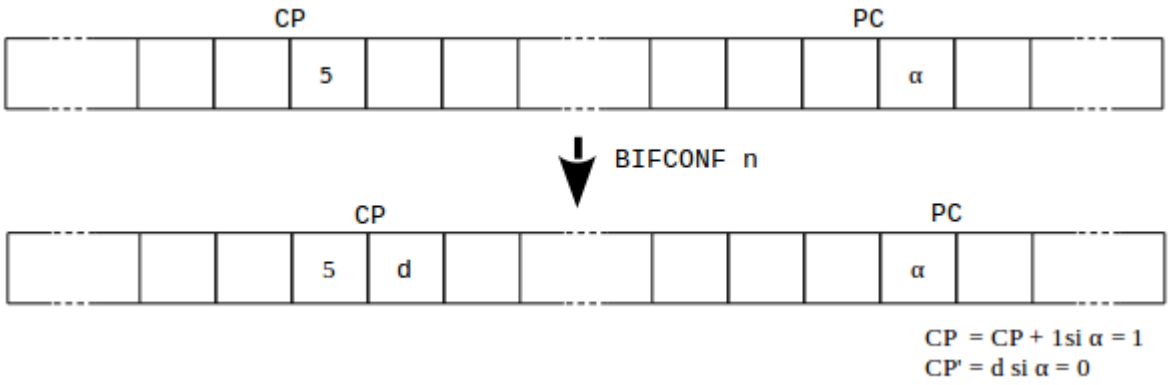
AREAL w

Con esta instrucción conseguimos convertir un operando de tipo entero que se encuentra en la pila en un valor real. El parámetro indica el número de posiciones de memoria que hay en la pila encima del operando.



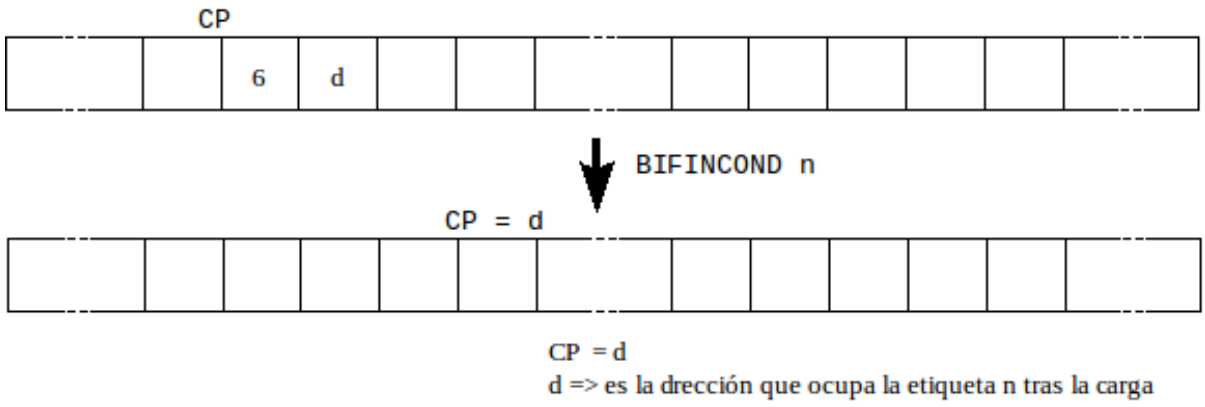
BIFCOND n

Comprueba si el valor de la cima de la pila es verdadero, es decir, 1. Si es así continúa la ejecución en la primera instrucción tras la etiqueta condicional n. En caso contrario continúa con la siguiente instrucción.



BIFINCOND n

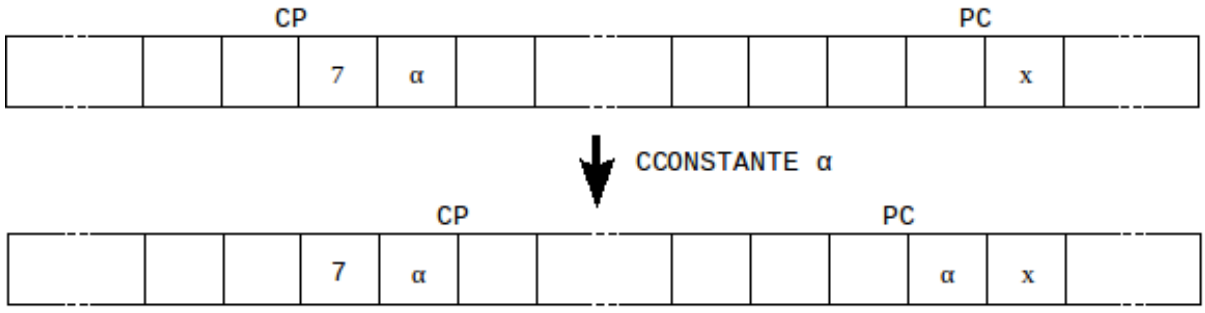
Esta instrucción hace que el contador de programa, CP, se modifique apuntando a la dirección de memoria que almacena la siguiente dirección de memoria. El parámetro representa la etiqueta de salto del programa, la cual tras en la carga del programa será traducida por una dirección de la memoria de la máquina virtual que representamos como d.



CP = d
d => es la drección que ocupa la etiqueta n tras la carga

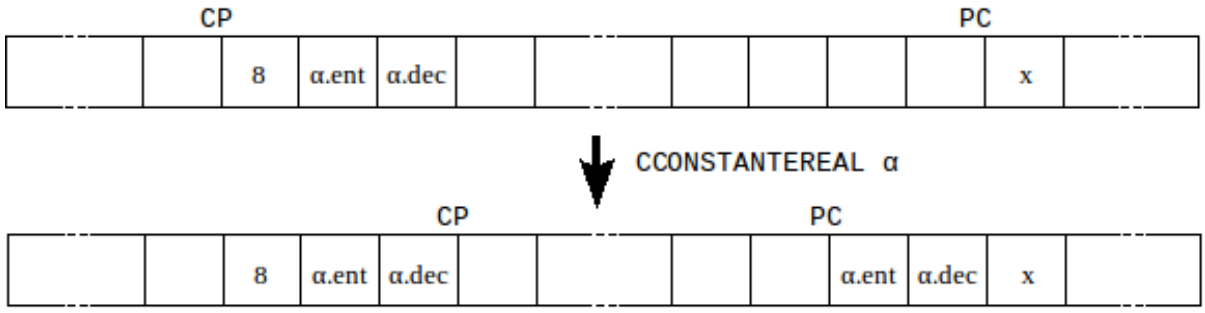
CCONSTANTE n

Esta instrucción carga en la cima de la pila la constante entera n.



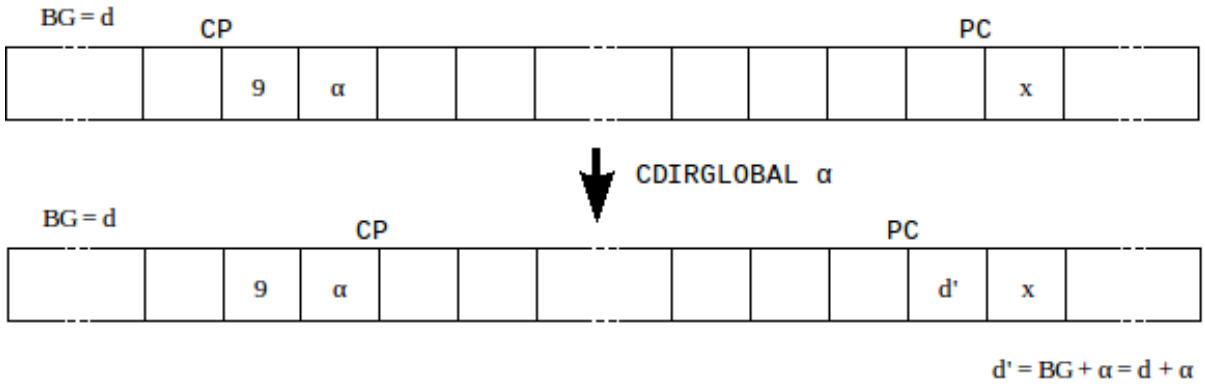
CCONSTANTEREAL n

Carga en la cima de la pila la constante real n.



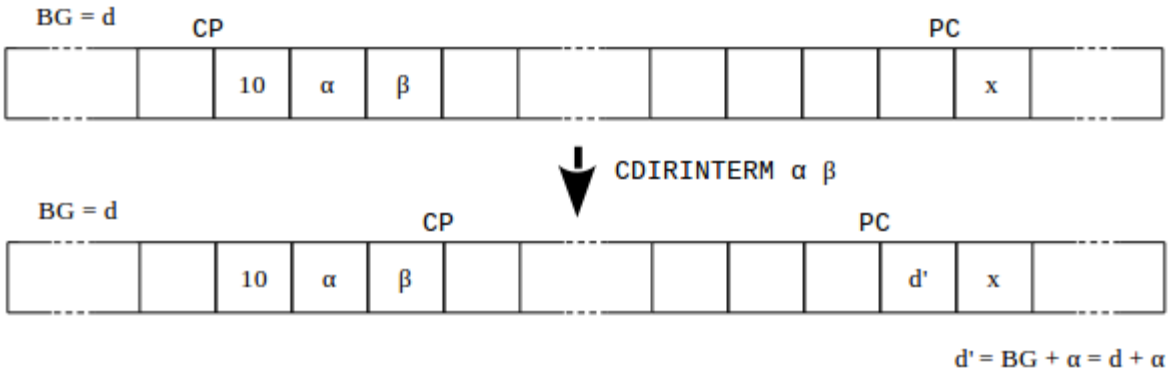
CDIRGLOBAL d

Con esta instrucción cargamos en la cima de la pila la dirección relativa a la base global determinada por el registro BG de la máquina virtual.



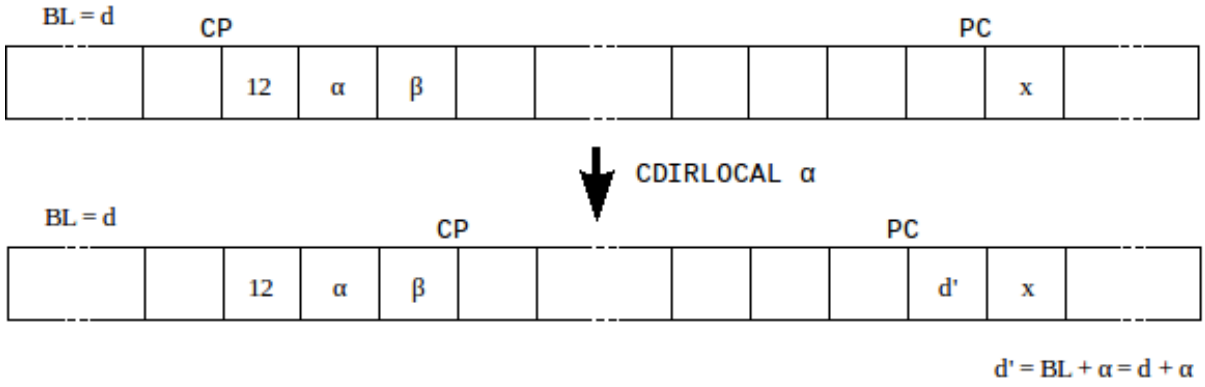
CDIRINTERM n m

Carga la dirección m relativa al área intermedia n en la cima de la pila.



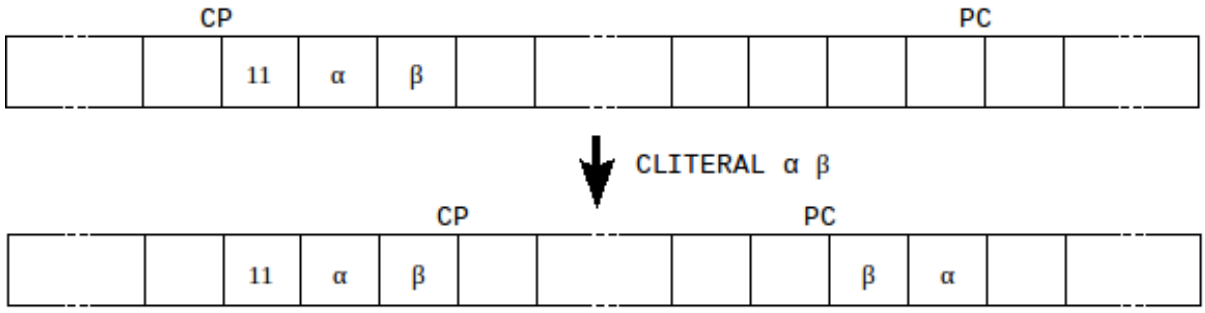
CDIRLOCAL n

Carga la dirección relativa n a la base local en la cima de la pila.



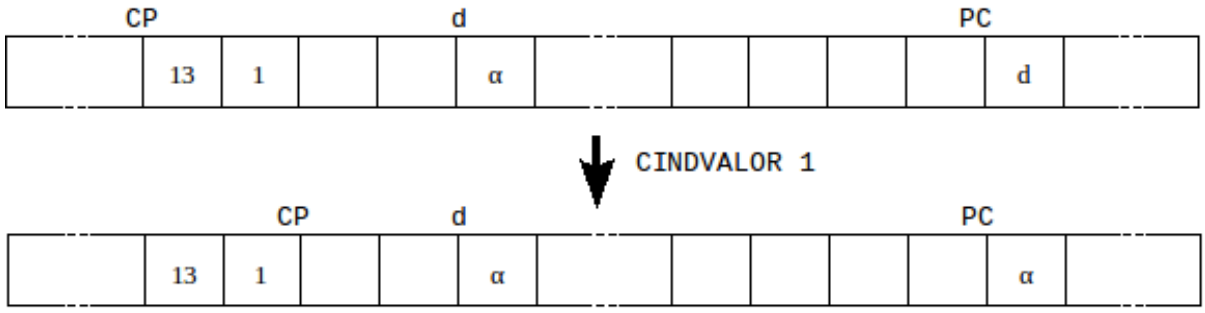
CLITERAL w n

En la implementación actual esta instrucción sólo es usada para utilizar a continuación la instrucción GRABACIONLITERAL. Su misión es cargar en la cima de la pila la dirección y el tamaño de un literal para que posteriormente pueda ser usado por GRABACIONLITERAL. En futuras implementaciones se podrá utilizar esta instrucción para nuevas funcionalidades como pueden ser operaciones con cadenas. El parámetro w indica el tamaño del literal y el parámetro n indica el número de literal que en el proceso de carga es traducido por la dirección del área de literales en la que está almacenado dicho literal.



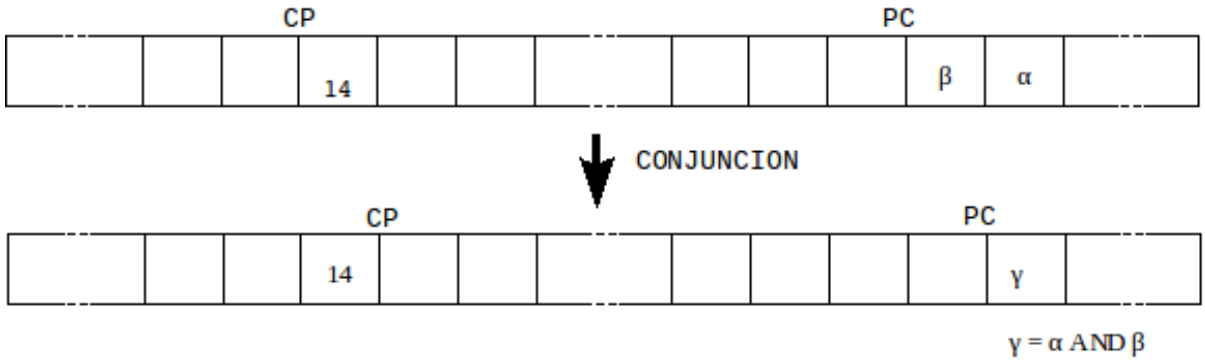
CINDVALOR w

Copia tantas posiciones de memoria como indica el parámetro w desde la dirección que se encuentra en la cima de la pila hacia la pila de ejecución.



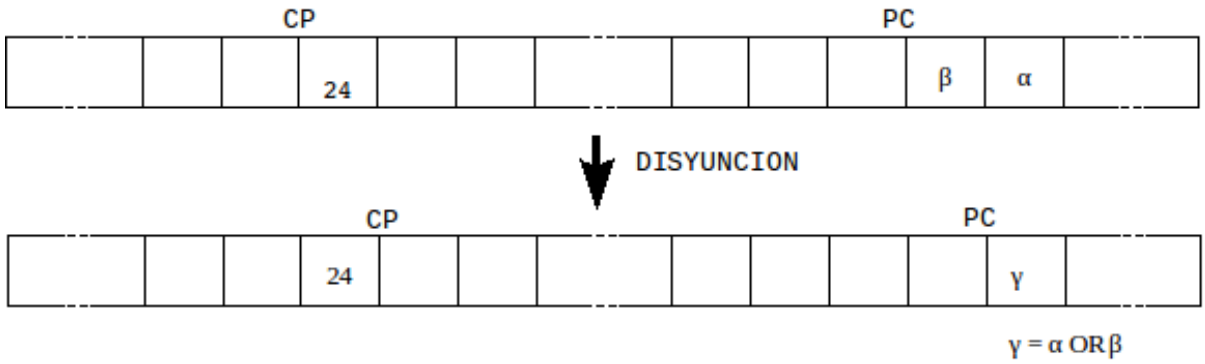
CONJUNCION

Realiza la operación lógica AND de los dos operandos que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



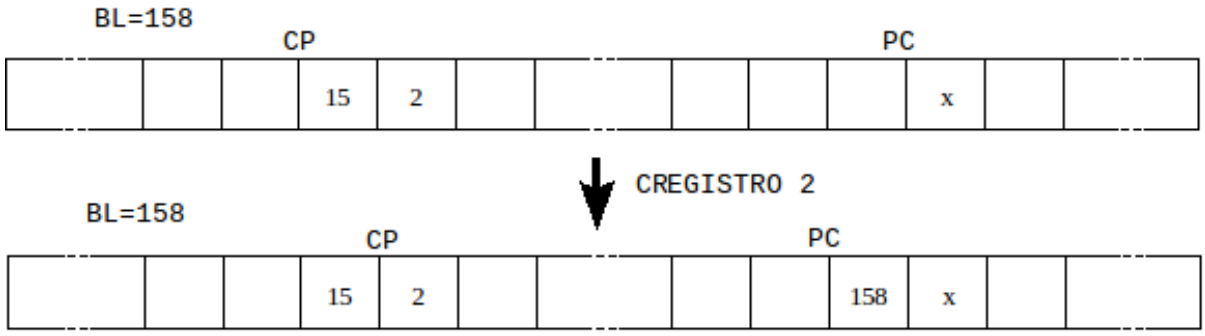
DISYUNCION

Realiza la operación lógica OR de los dos operandos que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



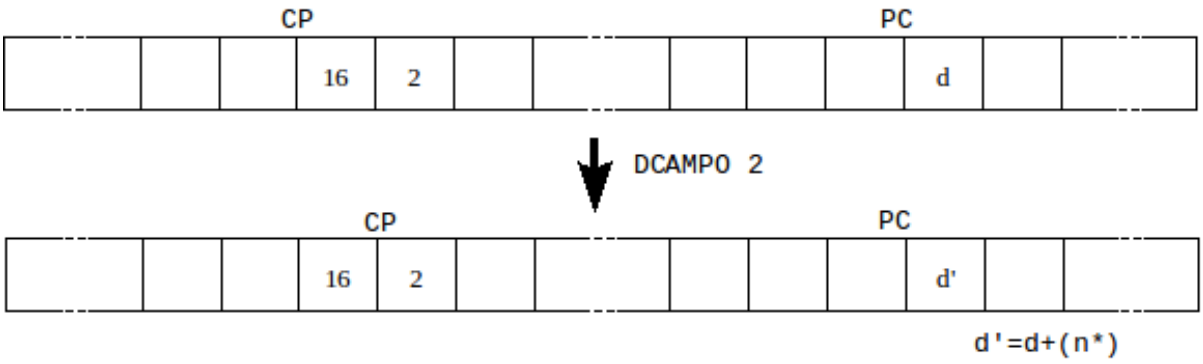
CREGISTRO

Carga en la cima de la pila unos de los registros internos de la máquina virtual JMPascalVM. Cada registro se reconoce por un valor entero.



DCAMPO n

Se suma a la dirección almacenada en la cima de la pila el valor n multiplicado por el número de posiciones que ocupa una dirección de memoria en la memoria que en la implementación actual es una posición.

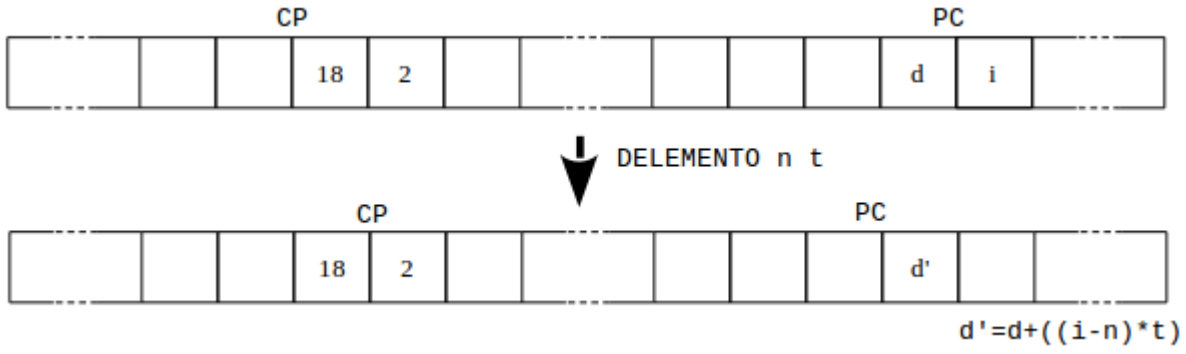


#DEFLITERAL n literal

Pseudo instrucción que carga en el área de literales, (cima de la pila mientras se realiza la carga del programa) los literales.

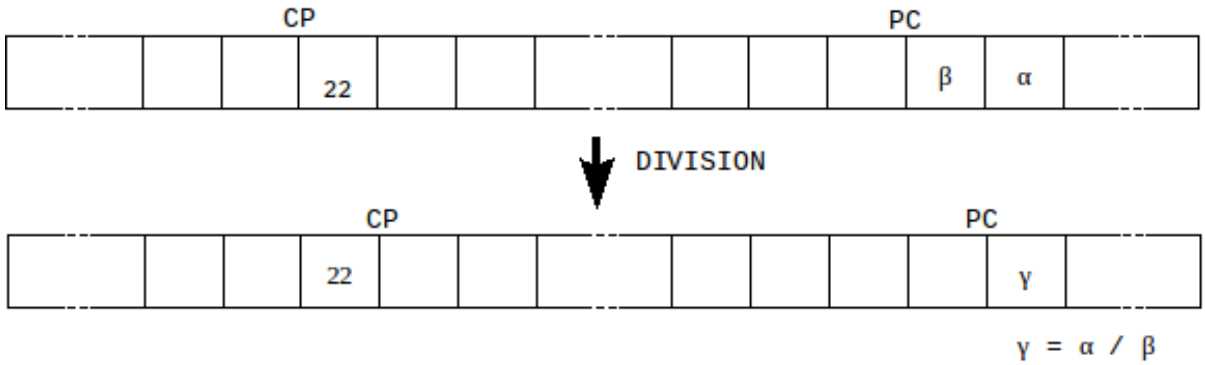
DELEMENTO n d

Incrementa la direccioń que se encuentra en la subcima de la pila con i elementos teniendo en cuenta que el primer índice es el parámetro inicial y que cada elemento tiene un tamaño t.



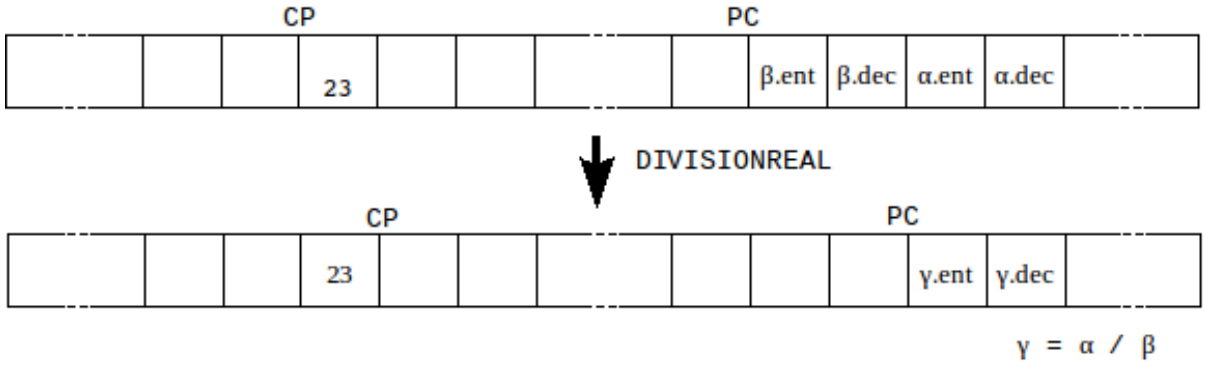
DIVISION

Realiza la división entera de los dos operandos enteros que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



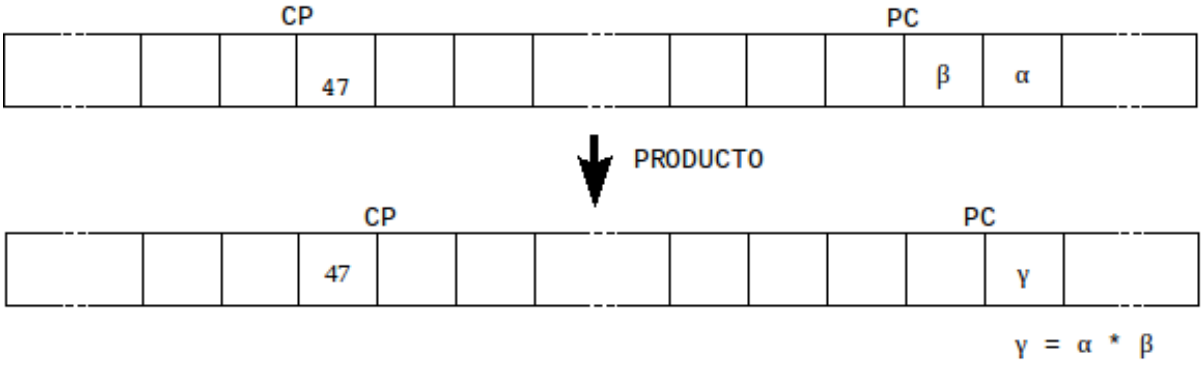
DIVISIONREAL

Realiza la división real de los dos operandos reales que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



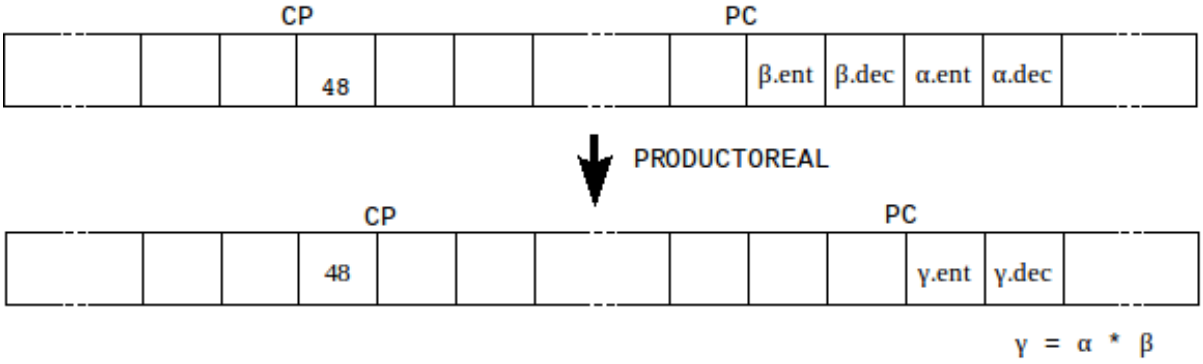
PRODUCTO

Realiza la multiplicación entera de los dos operandos enteros que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



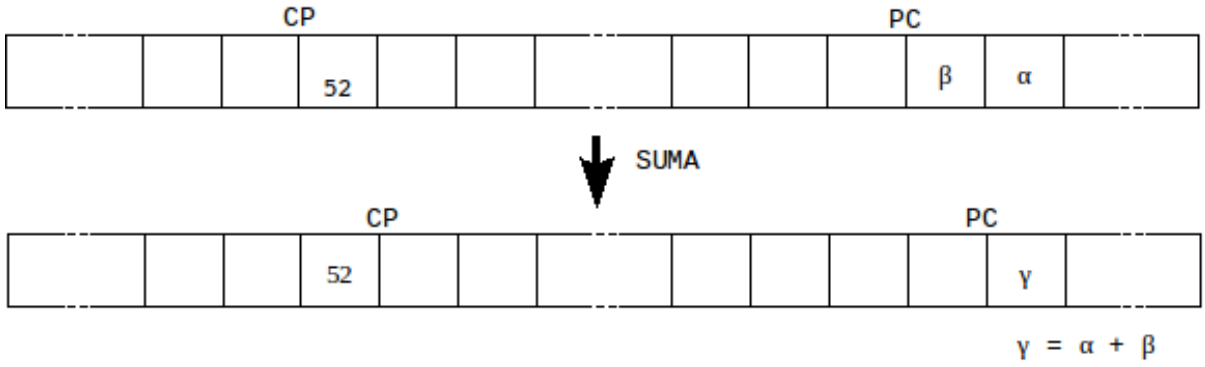
PRODUCTOREAL

Realiza la multiplicación real de los dos operandos reales que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



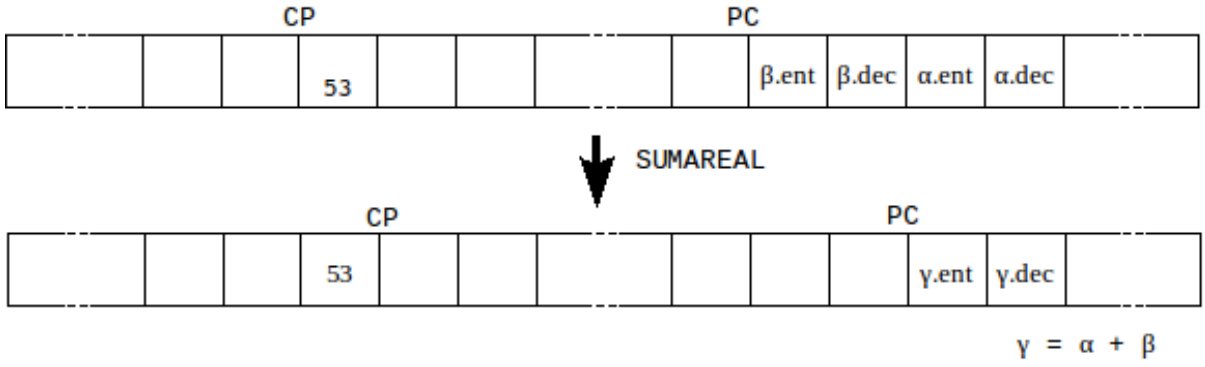
SUMA

Realiza la suma entera de los dos operandos enteros que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



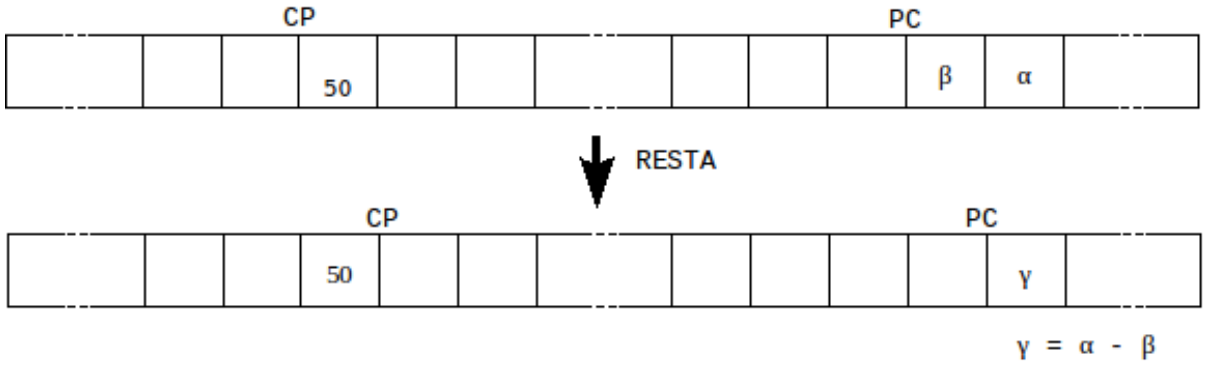
SUMAREAL

Realiza la suma real de los dos operandos reales que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



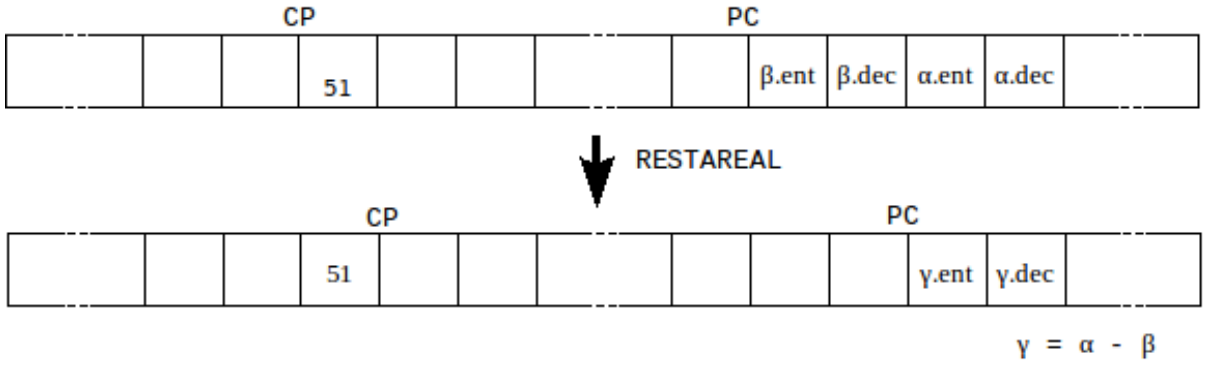
RESTA

Realiza la resta entera de los dos operandos enteros que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



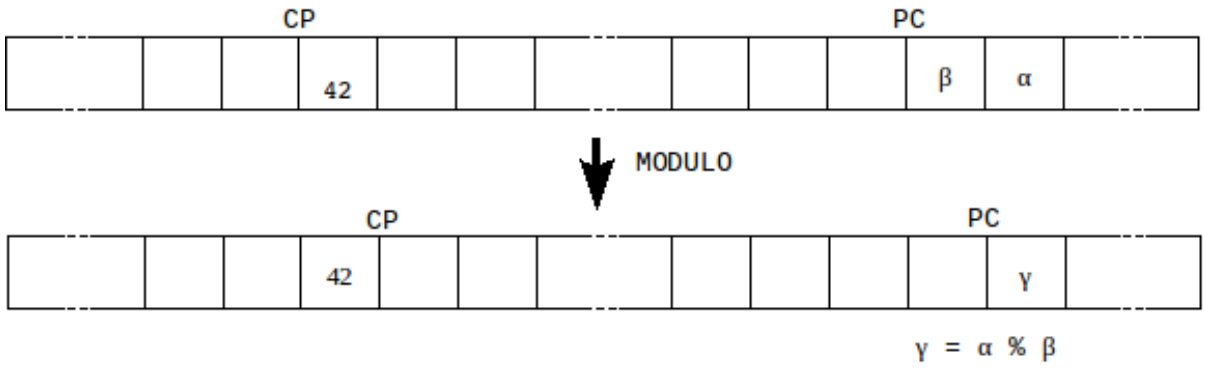
RESTAREAL

Realiza la resta real de los dos operandos reales que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



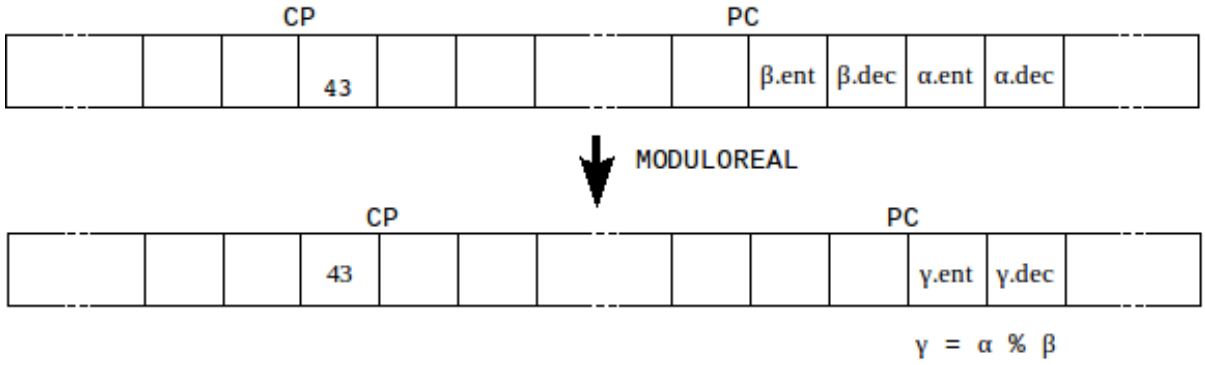
MODULO

Realiza el módulo de los dos operandos enteros que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



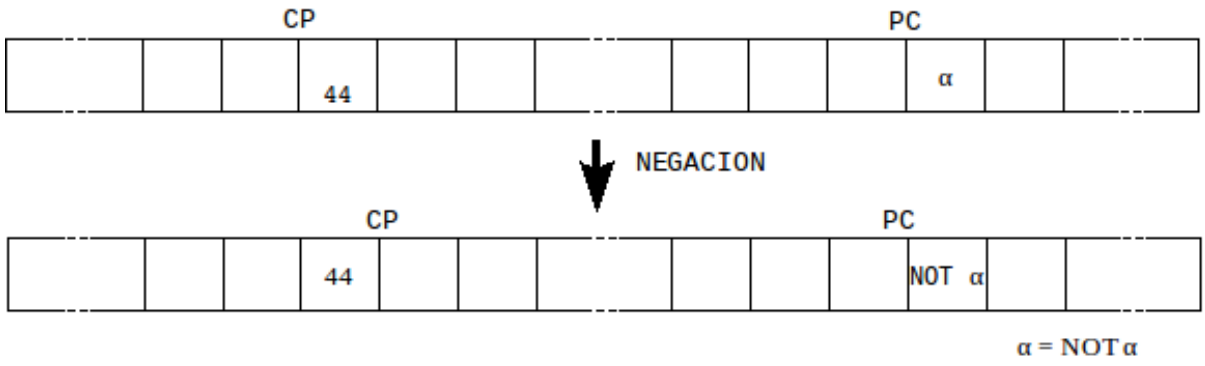
MODULOREAL

Realiza el módulo real de los dos operandos reales que se encuentran en la cima de la pila, dejando el resultado también en la cima de la pila.



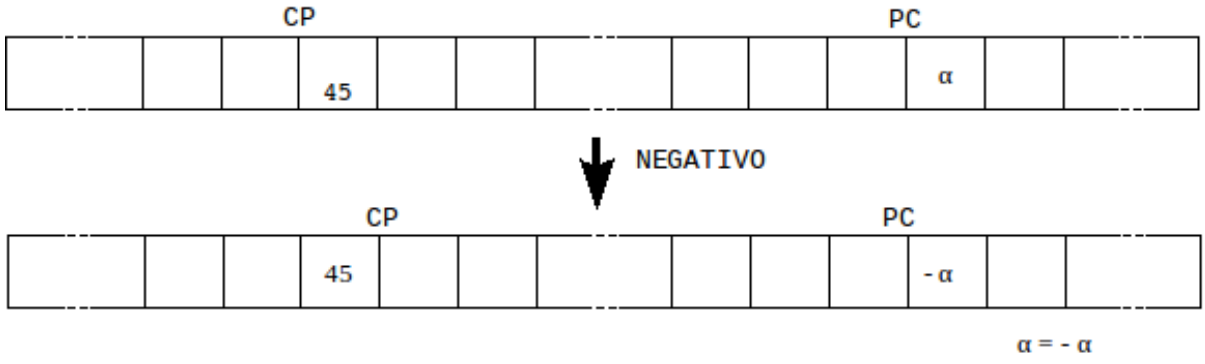
NEGACION

Niega lógicamente el operador que se encuentra en la cima de la pila, dejando el resultado también en la cima de la pila.



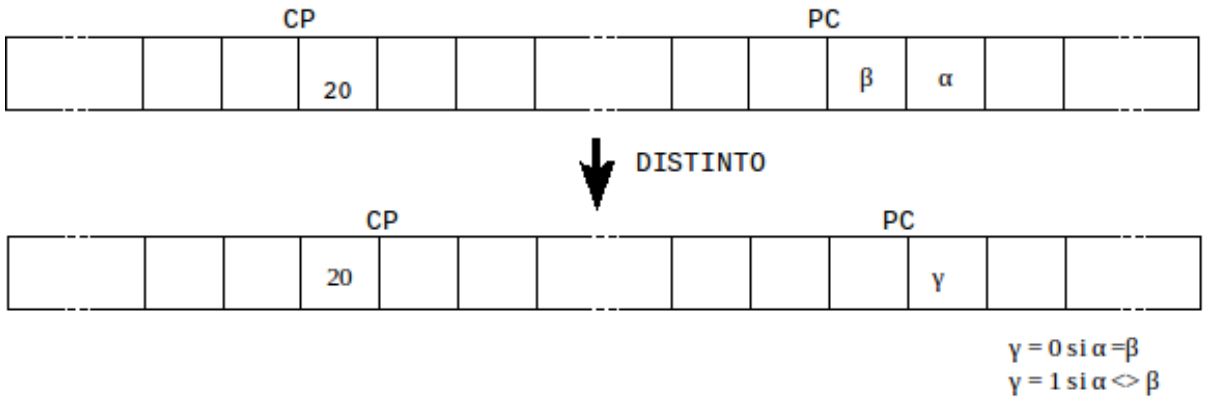
NEGATIVO

Niega aritméticamente el operador que se encuentra en la cima de la pila, dejando el resultado en la cima de la pila.



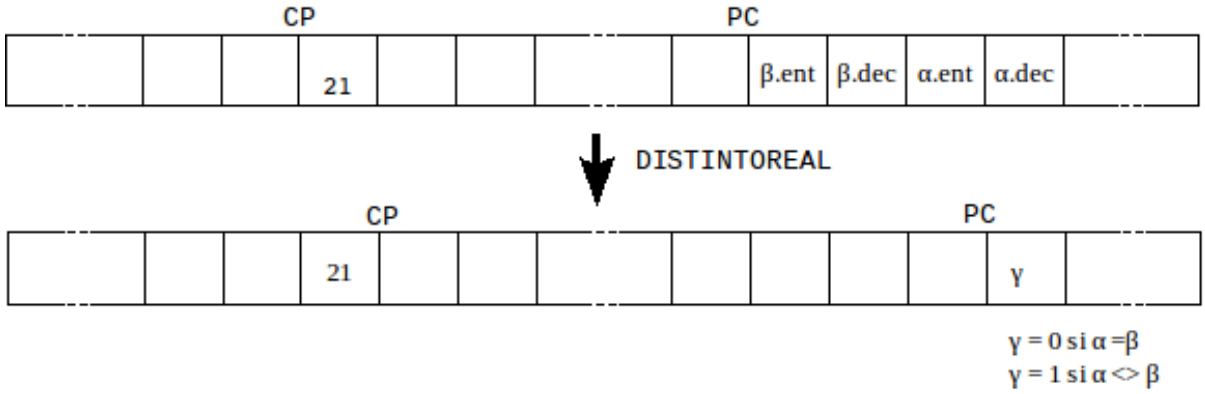
DISTINTO

Compara los dos operandos enteros de la cima de la pila dejando en la cima de la pila un 0 si ambos son iguales y 1 si son distintos.



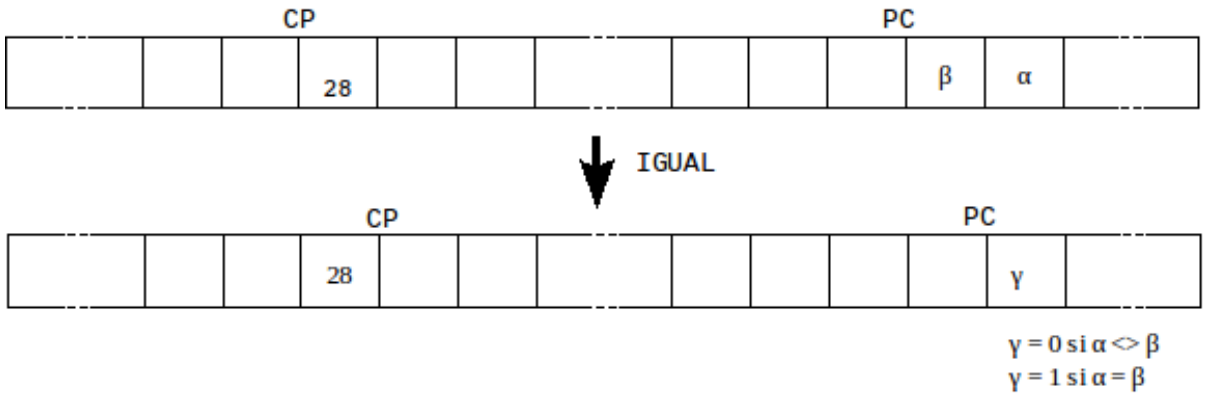
DISTINTOREAL

Compara los dos operandos reales de la cima de la pila dejando en la cima de la pila un 0 si ambos son iguales y 1 si son distintos.



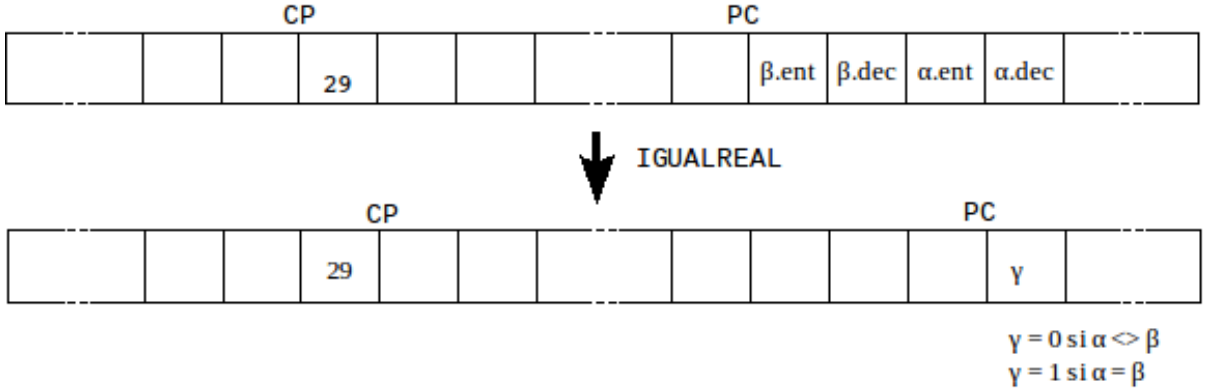
IGUAL

Compara los dos operandos enteros de la cima de la pila dejando en la cima de la pila un 1 si ambos son iguales y 0 si son distintos.



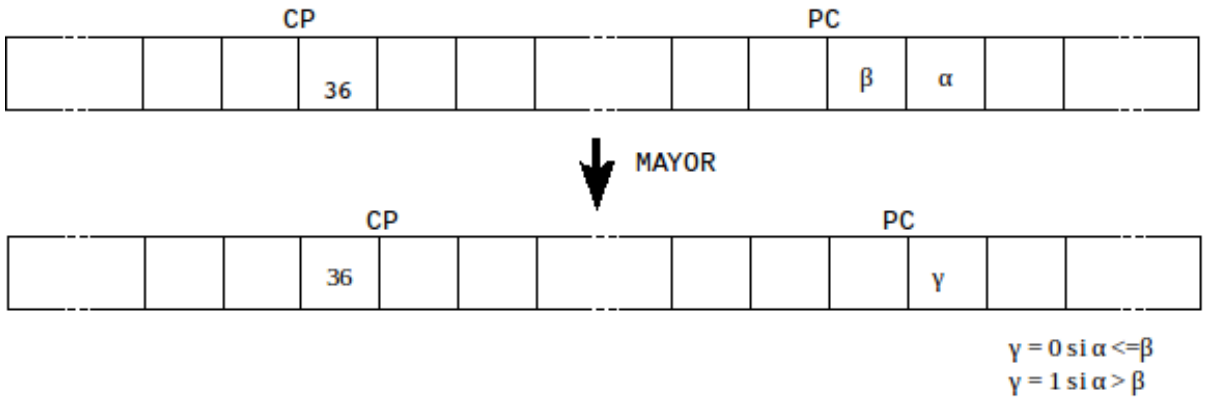
IGUALREAL

Compara los dos operandos reales de la cima de la pila dejando en la cima de la pila un 1 si ambos son iguales y 0 si son distintos.



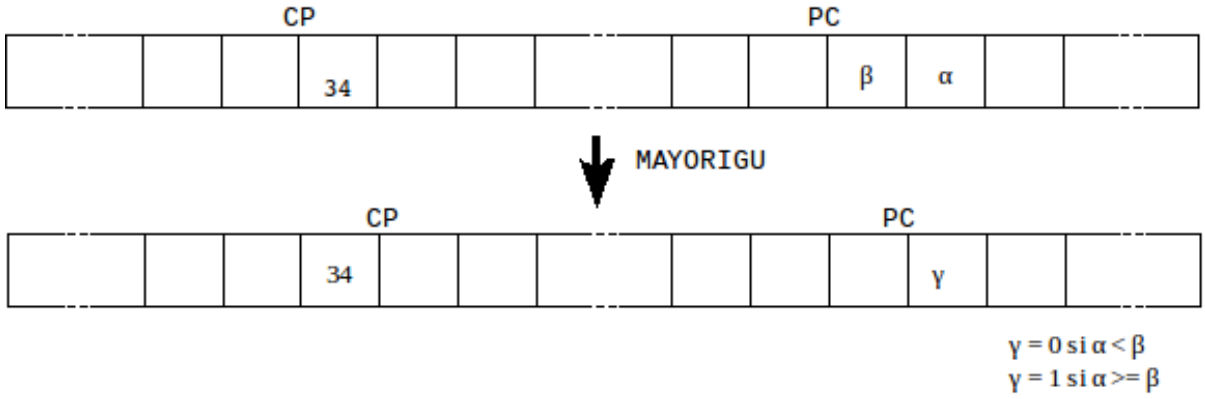
MAYOR

Compara los dos operandos enteros de la cima de la pila. Si el primero (dirección mayor de la memoria) es mayor que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



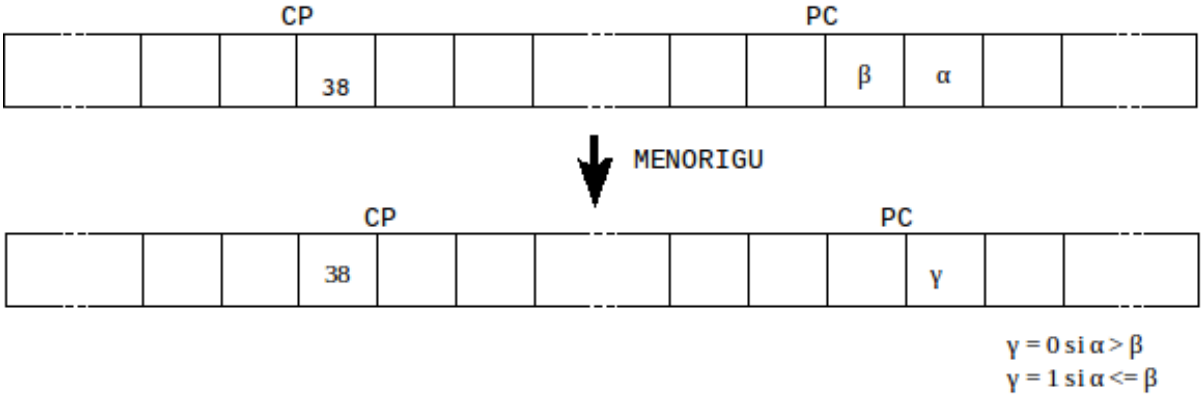
MAYORIGU

Compara los dos operandos enteros de la cima de la pila. Si el primero (dirección mayor de la memoria) es mayor que el segundo o igual se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



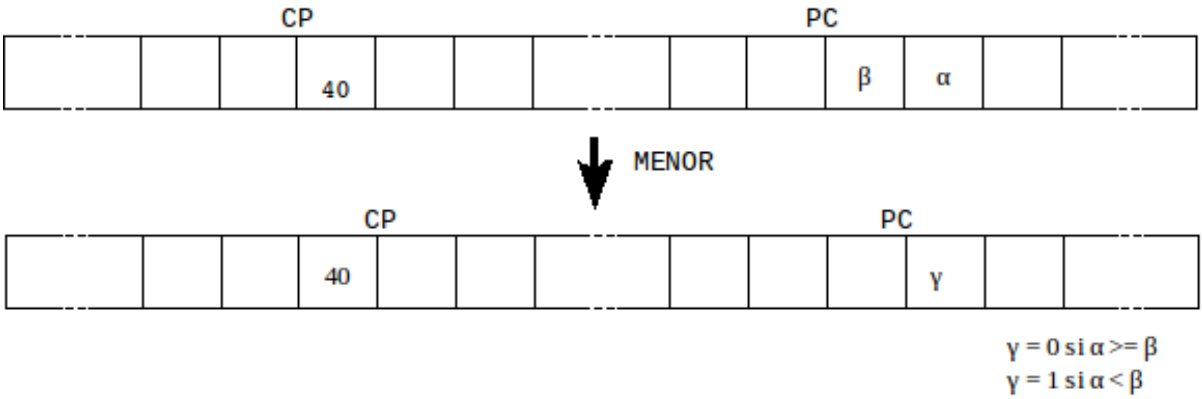
MENORIGU

Compara los dos operandos enteros de la cima de la pila. Si el primero (dirección mayor de la memoria) es menor o igual que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



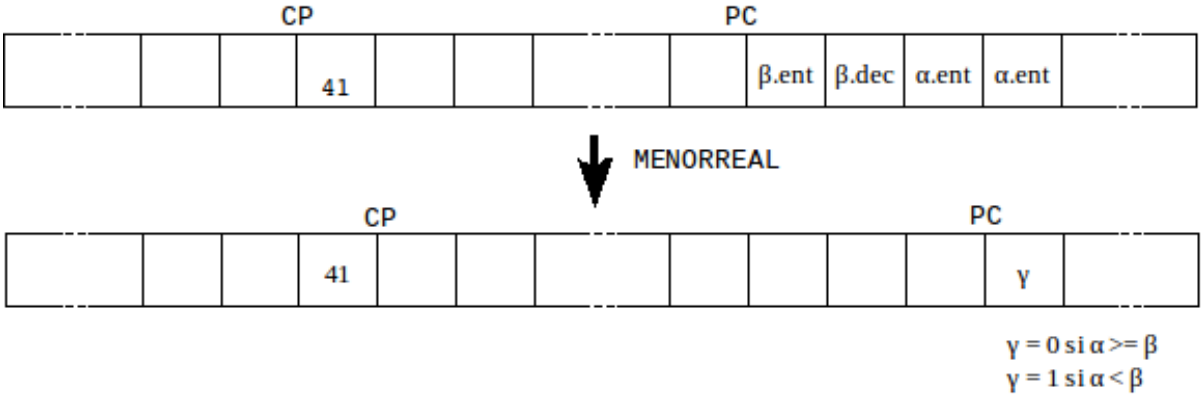
MENOR

Compara los dos operandos enteros de la cima de la pila. Si el primero (dirección mayor de la memoria) es menor que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



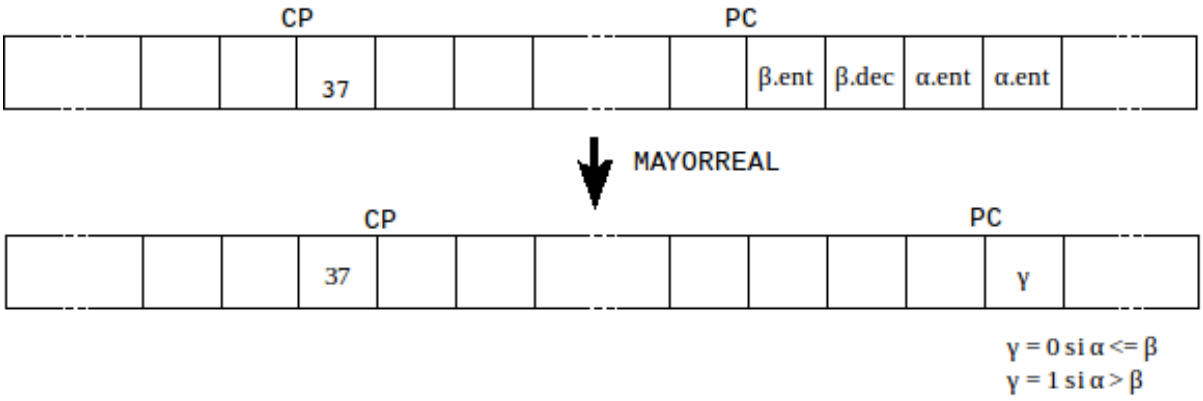
MENORREAL

Compara los dos operandos reales de la cima de la pila. Si el primero (dirección mayor de la memoria) es menor que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



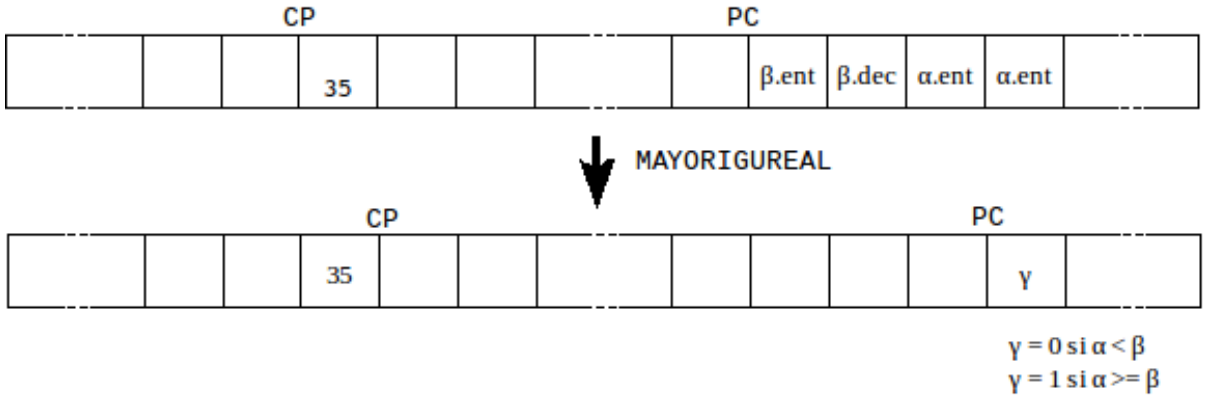
MAYORREAL

Compara los dos operandos reales de la cima de la pila. Si el primero (dirección mayor de la memoria) es mayor que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



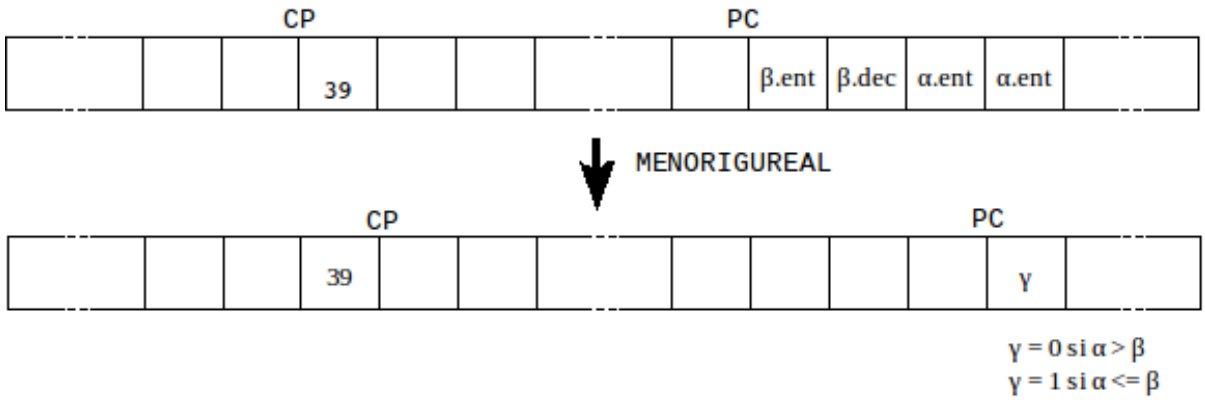
MAYORIGUREAL

Compara los dos operandos reales de la cima de la pila. Si el primero (dirección mayor de la memoria) es mayor o igual que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



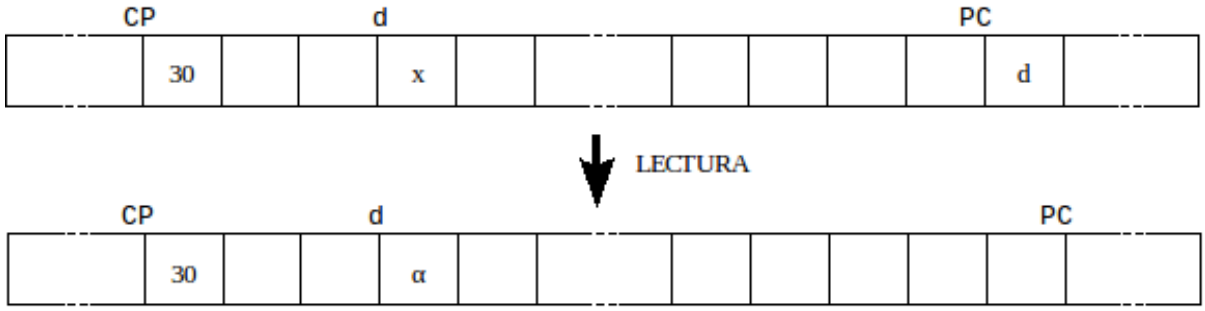
MENORIGUREAL

Compara los dos operandos reales de la cima de la pila. Si el primero (dirección mayor de la memoria) es menor o igual que el segundo se almacena un 1 en la cima de la pila, en caso contrario se almacena un 0.



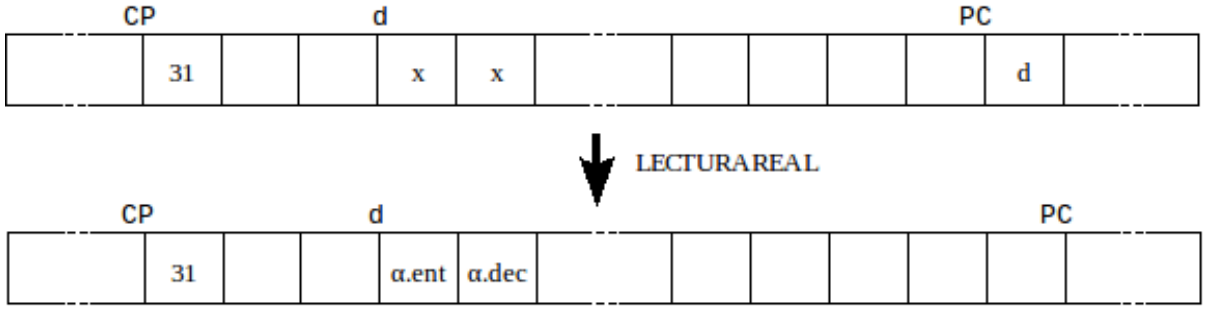
LECTURA

Esta instrucción lee de teclado una constante entera y la carga en la dirección que indica la cima de la de la pila.



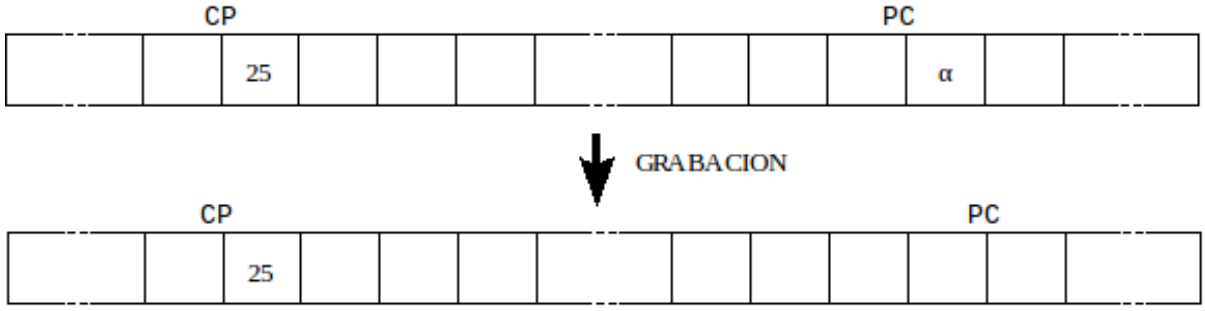
LECTURAREAL

Esta instrucción lee de teclado una constante real y la carga en la dirección que indica la cima de la pila.



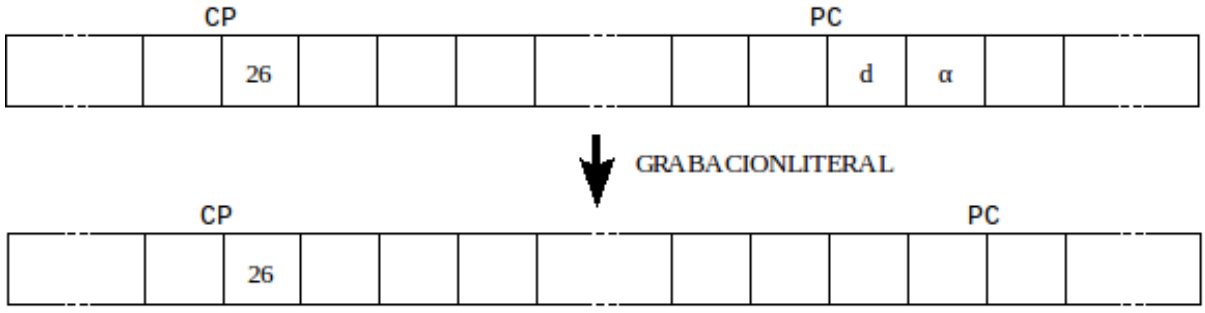
GRABACION

Escribe por la salida estándar la constante entera que se encuentra en la cima de la pila.



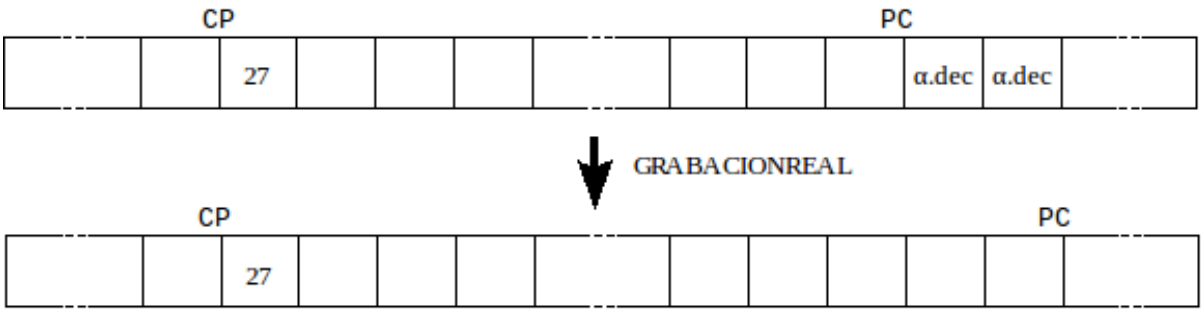
GRABACIONLITERAL

Esta instrucción escribe por la salida estándar el literal que indican los dos valores de la cima de la pila. El primer valor indica la dirección de comienzo del literal y el segundo el tamaño del mismo. Se debe tener en cuenta que los literales se almacenan de direcciones altas hacia direcciones bajas.



GRABACIONREAL

Escribe por la salida estándar la constante real que se encuentra en la cima de la pila.


CP
PC
27
α.dec
α.dec
GRABACIONREAL
CP
PC
27

5. Máquina Virtual JMPascalVM

Para ejecutar los programas traducidos de JSPascal a JMPascal, es necesario utilizar una máquina virtual diseñada e implementada para tal efecto. Esta máquina ha sido denominada JMPascalVM.

Aunque la implementación del intérprete y por tanto de la máquina virtual para ejecutar el código JMPascal no es ámbito de implementación del presente PFC (se planteará como ampliación en el capítulo *Conclusiones y Ampliaciones*), se considera necesario establecer, y conocer en detalle la definición de la máquina virtual que ejecutará el código JMPascal de cara a la correcta implementación del traductor.

En esta máquina virtual se pueden ejecutar programas cuyas instrucciones estén formadas por el conjunto de instrucciones que componen el lenguaje JMPascal descrito anteriormente.

Cada una de estas instrucciones tiene asociado un código numérico. Este código es el mismo que CUP genera al compilar la especificación sintáctica del lenguaje JMPascal para los terminales correspondientes a las instrucciones. Veamos a continuación la relación de valores:

- AINDVALOR (2)
- AREAL (3)
- ARESFUNC (4)
- BIFCOND (5)

- BIFINCOND (6)
- CCONSTANTE (7)
- CCONSTANTEREAL (8)
- CDIRGLOBAL (9)
- CDIRINTERM (10)
- CLITERAL (11)
- CDIRLOCAL (12)
- CINDVALOR (13)
- CONJUNCION (14)
- CREGISTRO (15)
- DCAMPO (16)
- DEFLITERAL (17)
- DELEMENTO (18)
- DEVOLUCION (19)
- DISTINTO (20)
- DISTINTOREAL (21)
- DIVISION (22)
- DIVISIONREAL (23)
- DISYUNCION (24)
- GRABACION (25)
- GRABACIONLITERAL (26)
- GRABACIONREAL (27)
- IGUAL (28)
- IGUALREAL (29)
- LECTURA (30)
- LECTURAREAL (31)
- LIMITACION (32)
- LLAMADA (33)
- MAYORIGU (34)
- MAYORIGUREAL (35)
- MAYOR (36)
- MAYORREAL (37)
- MENORIGU (38)
- MENORIGUREAL (39)
- MENOR (40)
- MENORREAL (41)
- MODULO (42)
- MODULOREAL (43)
- NEGACION (44)
- NEGATIVO (45)
- OCUPAESP (46)
- PRODUCTO (47)
- PRODUCTOREAL (48)
- RESERVAESP (49)
- RESTA (50)
- RESTAREAL (51)

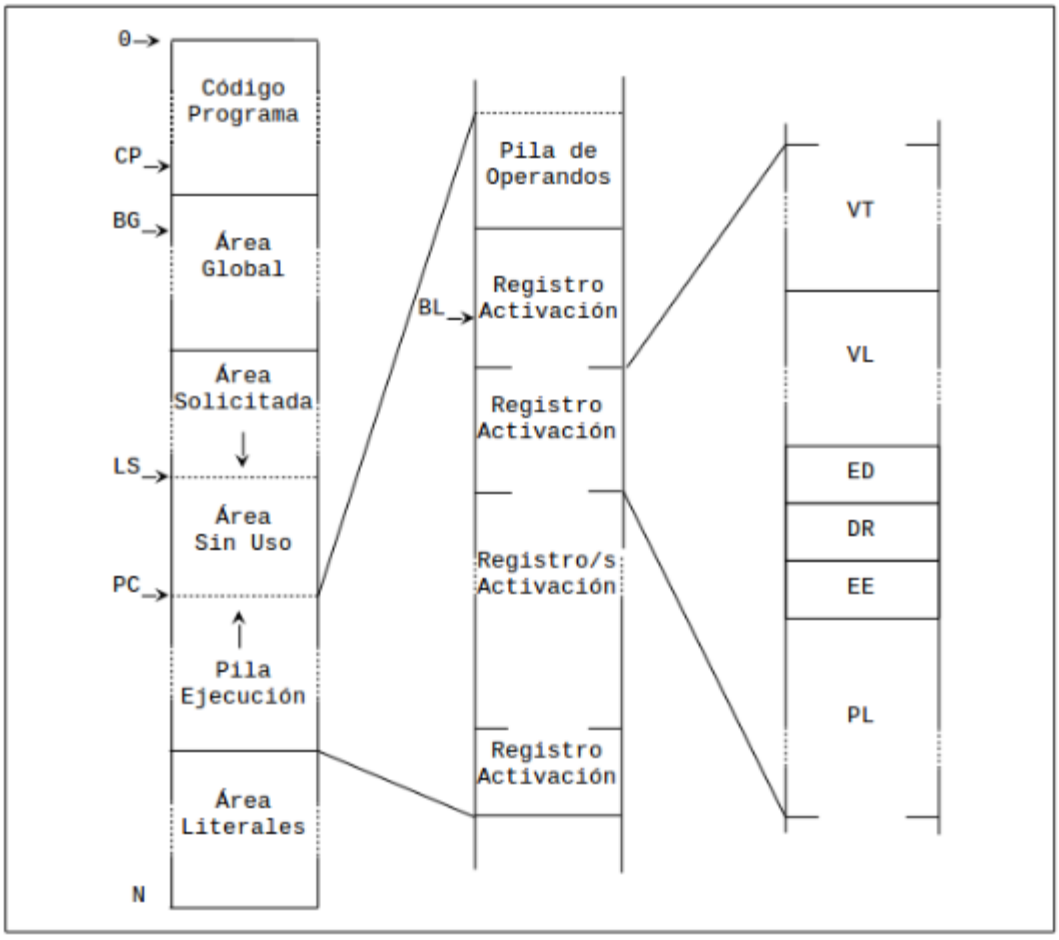
- SUMA (52)
- SUMAREAL (53)
- TAMFUNCION (54)
- TAMGLOBAL (55)
- TERMINACION (56)

5.1. Memoria de la máquina virtual JMPascalVM

La máquina virtual para la interpretación del código JMPascal se basa en un array de enteros que simule la memoria física de la máquina. Cada posición de la memoria admitirá una instrucción del lenguaje que es representada por el número correspondiente del listado anterior. Además de instrucciones las posiciones de la memoria pueden contener valores enteros que representan números enteros, número reales, direcciones de la memoria o caracteres pertenecientes a los literales. Los números reales tienen una manera especial de ser introducidos en la memoria que será explicada en secciones posteriores.

El tamaño de la memoria es fijo y está delimitado por el tamaño del array que la implementa. Este array se definirá en base a una constante, de nombre TAMMEMORIA, que se especificará en la clase que represente la máquina virtual cuyo valor en la implementación inicialmente puede ser de 16384.

Se pueden distinguir claramente 5 zonas o áreas que se separan de manera lógica y no de manera física. En la figura siguiente podemos ver una descripción gráfica de la distribución de la memoria de la máquina virtual.



Memoria de la máquina virtual JMPascalVM

La primera de ellas es el **Área de Código** que empieza en la dirección 0 de la memoria y que se extiende ascendentemente todo lo que sea necesario para almacenar el número de instrucciones y parámetros del programa JMPascal a ejecutar.

La siguiente zona representa el **Área Global**, que es la zona destinada a almacenar las variables globales del programa a interpretar. El comienzo de este área se delimita por el registro de la máquina virtual **BG** (base global) y se extiende ascendentemente tantas posiciones de memoria como sea necesario para albergar las variables globales del programa en ejecución.

A continuación se encuentra el **Área Solicitada** cuya función es proveer memoria dinámica al programa en ejecución, tal y como se puede realizar con el uso de punteros en Pascal. La definición actual de JSPascal no incluye punteros, sin embargo, se ha tenido en cuenta para futuras implementaciones o versiones del sPascalator. Comienza en la posición delimitada por el registro **LS** (límite solicitado) de la máquina virtual.

El último área es el **Área de Literales** que almacena los literales definidos en el programa mediante la instrucción DEFLITERAL. Comienza en la última posición de la memoria y se va rellenando de manera descendente. Cada literal se almacena mediante un recorrido lineal por los caracteres del literal, insertando cada uno de ellos en la cima de la pila. Este espacio está delimitado por una constante, **MAXLITERALES,** que se definirá en la clase que implemente la máquina virtual JMPascalVM.

Entre el Área Solicitado y el Área de Literales se encuentra la **Pila de Ejecución** que es el espacio de memoria destinado a albergar los operandos así como todos los parámetros necesarios para poder llevar a cabo la ejecución definidos estos por los espacios de direccionamiento destinados a la ejecución de cada uno de los subprogramas de un programa, **los registros de activación** que se explicarán en el apartado 5.3 de esta documentación. Este área comienza en la dirección anterior a la última utilizada por el Área de Literales y crece hacia las direcciones más bajas de la memoria.

Ya que en esta máquina virtual la pila de ejecución crece de direcciones altas hacia direcciones bajas y el bloque formado por el código de programa, el área global y el área solicitado lo hace de manera inversa podría darse el caso de que ambos se junten lo que ocasionará un desbordamiento de pila, que debe ser controlado en la implementación como un error de ejecución.

5.2. Control de la máquina virtual JMPascalVM

Cualquier CPU de la actualidad tiene registros para almacenar valores temporales, direcciones, etc. necesarios para poder llevar a cabo la ejecución de programas de una manera controlada. Además hoy en día encontramos un sistema operativo en todos los ordenadores que facilita las operaciones de entrada/salida normalmente mediante semáforos en el propio sistema operativo. De la misma

manera la máquina virtual JMPascalVM dispone de varios registros y semáforos. Todos los registros almacenan direcciones mientras que los semáforos tienen dos funcionalidades diferenciadas.

La JMPascalVM dispone de un total de 5 registros:

- **CP: Contador de programa**, se usa en la carga del programa para saber la dirección siguiente en la que se debe almacenar la próxima instrucción o parámetro y durante la ejecución para indicar la dirección siguiente desde la que leer para continuar con la ejecución.
- **BG: Base Global**, marca la dirección en la que comienza el espacio de la memoria destinado a almacenar las variables globales, es decir, el área global.
- **BL: Base Local**, indica la dirección en la que se encuentra la base del nivel de ejecución correspondiente al subprograma que se está ejecutando en cada momento.
- **PC: Puntero de Cima**, indica la dirección a partir de la cual, de manera descendente, se puede incluir un nuevo dato en la pila de ejecución.
- **LS: Límite Solicitado**, indica la dirección a partir de la cual se puede asignar memoria de manera dinámica. Mientras que la definición del lenguaje JSPascal no incluya punteros este valor siempre será igual al registro BG más el número de direcciones ocupadas por las variables globales debido a que no se habrá asignación dinámica de memoria.

La máquina virtual además debe disponer de dos semáforos para controlar los dos sistemas con los que puede interactuar el usuario final.

Dado el carácter educativo del PFC el primer semáforo tiene como fin controlar la ejecución cuando se activa el debug en la máquina virtual, permitiendo realizar la ejecución paso a paso. Este semáforo debe ser por tanto binario para bloquear la ejecución tras cada instrucción hasta que el usuario fuerza el avance siempre y cuando el debug haya sido activado.

El otro semáforo tiene como misión controlar el sistema de entrada/salida que utilizan las instrucciones LECTURA y LECTURAREAL. Para ello se utiliza un semáforo que al contrario del anterior no debe ser binario y que bloquea la ejecución cuando se realiza una petición de lectura sin que el usuario haya introducido dato alguno. La máquina virtual dispondrá de un buffer que almacene los datos introducidos por el usuario que se consumen a medida que se ejecuten instrucciones de lectura, cada vez que el usuario introduce un dato en este buffer se producirá un incremento del contador de este semáforo.

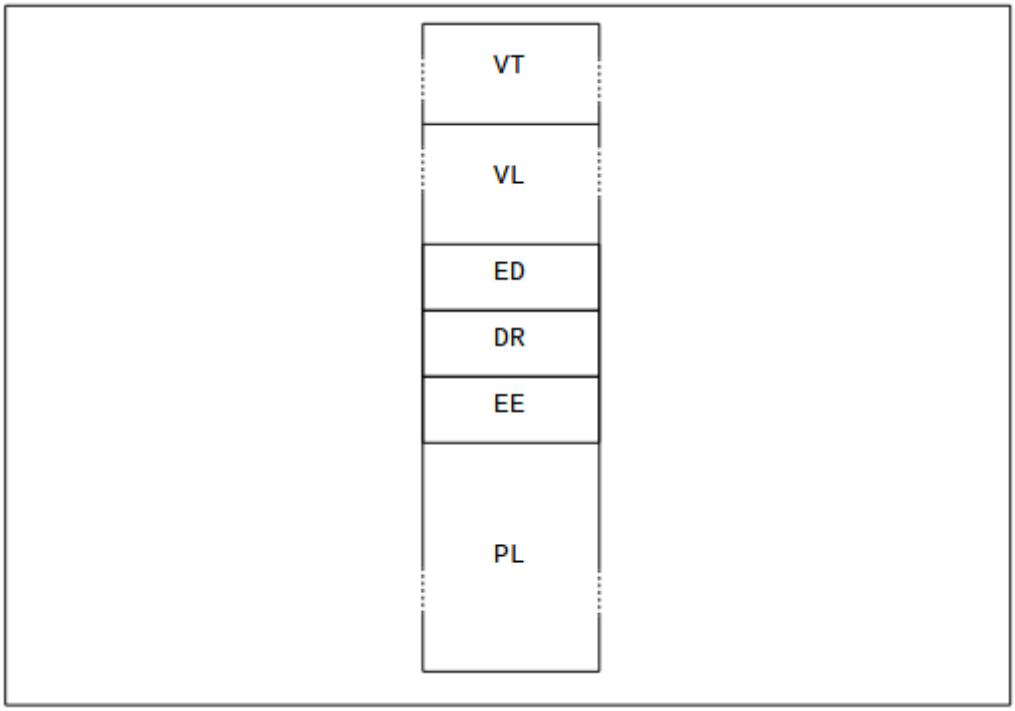
5.3. El registro de activación

Los registros de activación son la representación en memoria de los subprogramas. Cada vez que se realiza una llamada a un subprograma se crea un

registro de activación, el cual contiene toda la información necesaria para la ejecución del subprograma excepto su código que se encuentra en el área de código de la memoria.

Estos registros se apilan y desapilan de la pila de ejecución tal y como se mostraba en la figura anterior.

La composición de un registro de activación es la que se muestra en la siguiente figura.



Registro de Activación

Como se puede ver el registro de activación tiene diferentes elementos establecidos en un orden que debe ser inalterable a no ser que modifique totalmente

la especificación de la máquina JMPascalVM y del lenguaje JMPascal. Los componentes que forman el registro son:

- **PL**: Se almacenan en este espacio de memoria los parámetros con los que se llama al procedimiento, que lógicamente pueden no existir. En el caso de que el parámetro sea por referencia se copiará a este espacio la dirección de la variable en cuestión mientras que si se trata de un parámetro por valor se copiará directamente el contenido de la variable.
- **EE**: El **enlace estático** sirve para enlazar los registros de activación que son accesibles desde el registro actual. Funciona según las reglas de ámbitos de un lenguaje de bloques como Pascal. Este elemento sólo existirá si existen espacios de otros subprogramas accesibles desde el subprograma en ejecución.
- **DR**: La **dirección de retorno** almacena el valor del contador de programa en el momento de realizar la llamada al procedimiento para que tras su ejecución se pueda restaurar.
- **ED**: El enlace dinámico enlaza el subprograma llamado con el llamante. Este elemento es considerado como base para los desplazamientos locales en el registro de activación, por lo tanto, cuando el registro de activación está activo el valor del registro BL coincide con la dirección en la que se ha almacenado este elemento.
- **VL**: Espacio para las variables locales al subprograma representado por el registro de activación.
- **VT**: Variables temporales que el compilador irá apilando para realizar las operaciones que necesita para ejecutar las distintas instrucciones.

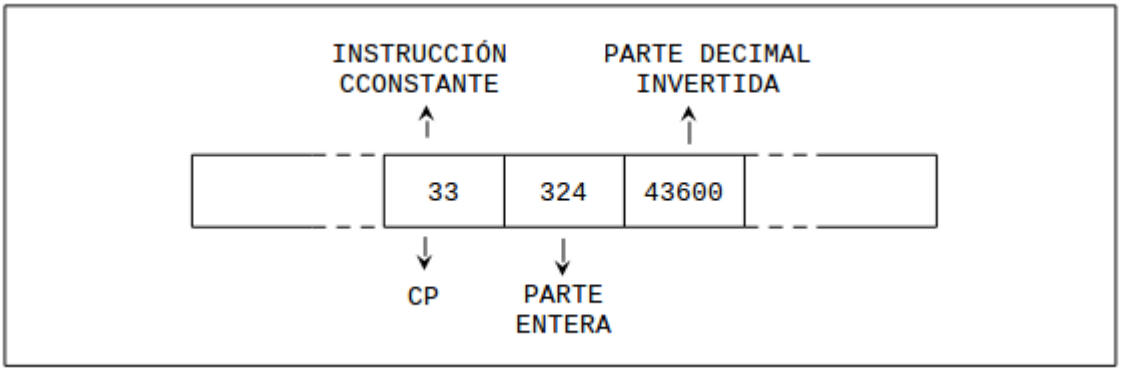
El direccionamiento de las variables locales en el registro de activación se realiza teniendo en cuenta que como la base es elemento del enlace estático, los parámetros se situarán en direcciones locales con desplazamiento positivos mientras que las variables locales tendrán un desplazamiento negativo.

5.4. Representación de los números reales en la JMPascalVM

Los números reales en la máquina JMPascalVM se representan mediante dos posiciones consecutivas de la memoria. La primera de ellas (la más baja) guarda la parte entera del número mientras que la parte decimal se guarda en la dirección más alta.

Para no perder el valor real de la parte decimal este se guarda tras una transformación que consiste en trasladar la última cifra a la primera, la penúltima a la segunda etc. De esta manera se evita perder las cifras decimales más altas en el caso de que sean ceros.

En el siguiente ejemplo podemos ver como quedaría la instrucción para cargar una constante real, 324.00634, en la memoria virtual tras la carga de un programa JMPascal:


INSTRUCCIÓN
CCONSTANTE
PARTE DECIMAL
INVERTIDA
33
324
43600
CP
PARTE
ENTERA

Parte II - Implementación

6. Compiladores y objetos ¿compatibles?

Tal y como se exponía en la introducción del presente documento es evidente que el mundo actual la programación orientada a objetos es clave en casi cualquier ámbito habiéndose introducido incluso en ambientes tan complejos como los sistemas empotrados, lenguajes directamente interpretados como PHP, o lenguajes veteranos como Cobol.

Sin embargo, parece que el paradigma de la programación orientada a objetos no incide en el desarrollo de compiladores, traductores o intérpretes.

Cabe entonces preguntarse si es posible implementar un traductor completo utilizando programación orientada a objetos con todo lo que ello conlleva: modularidad, herencia, jerarquía y encapsulación. Inicialmente la respuesta debe ser que sí, ya que según los principios de la programación cualquier programa que se puede implementar con programación procedural puede ser implementado también mediante programación orientada a objetos.

Si la implementación es posible, debemos pensar en posibles métodos de implementación para seguir comprobando si la implementación será válida o no. De esta manera se intuyen dos posibles implementaciones:

- Una de ellas basada en una única clase que representa todas las piezas sintácticas del lenguaje, con multitud de atributos, propiedades y métodos de entre los cuales sólo serán usado unos pocos según el tipo de token que se esté analizando.
- La otra consiste en crear una jerarquía de clases. Cada una de estas clases representa un tipo de token y tiene únicamente las propiedades y métodos que ese tipo de token necesita para ser analizado. Aplicando herencia y redefiniendo métodos sobre esta jerarquía se puede conseguir personalizar más y más los tokens genéricos en partes concretas del programa en la que adoptan alguna característica especial.

Es intuitivo que la primera de las versiones consiste en la migración de una implementación tradicional en C o Pascal mediante uniones o registros con variantes respectivamente a una clase. Podemos intuir inicialmente que este tipo de implementación no obtiene ninguna ventaja de las propiedades de la programación orientada a objetos y que difícilmente se pueda obtener una implementación más eficaz que la que pueden proporcionar las uniones de C.

Por otra parte si analizamos la segunda opción comprobamos que se adapta perfectamente al concepto de herencia de la programación orientada a objetos y se aprovechan al máximo las posibilidades que ofrece esta tecnología, sin embargo, adelantándonos en la implementación cabe preguntarse “si el tipo de token viene definido por la clase que lo representa en el análisis ¿cómo se comprueba de qué

tipo de token hablamos?”. Sabemos que la programación orientada a objetos tiene cuatro grandes características: *abstracción*, *modularidad*, *jerarquización* y por último *encapsulación* que entre otras cosas indica que no se debe preguntar nunca por la clase de un objeto, entonces ¿cómo sabremos qué tipo de objeto estamos analizando? Nuevamente existen dos posibles soluciones:

- Saltarse las reglas básicas de la programación orientada a objetos y comprobar de qué clase es un objeto, para lo cual en cualquier clase Java (todas heredan de la clase Object) es posible utilizar el método getClass() que devuelve un String con el nombre de la clase.
- La otra posibilidad consiste en no preguntar nunca a qué clase pertenece los objetos y utilizar el sistema de prueba y fallo, es decir, intentar ejecutar presuponiendo una clase y esperar a que sea válida la suposición. Si no es así se provocará un error y lo intentaremos con otra suposición, al final daremos con la interpretación adecuada.

Debido a que lo que se pretende es conseguir una implementación los más orientada a objetos posible, en este proyecto se ha optado por la segunda manera de trabajar, aprovechando además que en Java se puede realizar la captura de excepciones permitiendo el lenguaje realizar la implementación basada en prueba y fallo mediante la concesión de objetos por medio de operaciones “*cast*” ya que si la suposición es buena el casting entre clases será correcto y si no saltará una excepción de tipo ClassCastException indicando que nos hemos equivocado y que se debe probar con la siguiente de las posibilidades. Esta solución no es muy

elegante como podrá apreciar cualquier programador experimentado y deja además en el aire la duda de lo eficaz que será una implementación concretamente en Java basada en que la lógica de la ejecución de la traducción se rija por la consecución continua de excepciones.

Con la implementación elegida además se consigue también y en gran medida cumplir las reglas de modularidad y jerarquía ya que al dividir todo en varias clases que forman una jerarquía de objetos se permite que cada módulo sea independiente de los demás y cambiarlo sin más con el único requisito de mantener la API que le envuelve.

Inicialmente no es posible mantener de manera completa la abstracción ya que el hecho de utilizar el sistema de prueba y fallo anula esta posibilidad haciendo al programador ser consciente de cada uno de estos casos y realizar la implementación en base a ello.

Las expectativas por lo tanto no son muy alentadoras para llevar a cabo implementaciones de traductores o compiladores basándose en programación orientada a objetos, sin embargo por otro lado el simple hecho de comprobar los resultados que se pueden obtener dan pie a realizar las pruebas y comprobar si la teoría es correcta o se equivoca, cosa que sólo es posible comprobar tras llevar a la práctica una implementación más o menos completa de un traductor.

Otra característica importante que influye en la posibilidad de implementar traductores con lenguajes orientados a objetos como una realidad es la aparición de herramientas que se basan en estos lenguajes, fundamentalmente Java, y que facilitan la implementación del traductor en gran medida:

- **JLex**: Herramienta generadora de analizadores léxicos basado en el programa lex de Unix.
- **Jflex**: Otra herramienta generadora de analizadores léxicos basada en la anterior con la particularidad de ser software libre.
- **Byacc**: Modificación del programa yacc, generador de analizadores sintácticos en C, que añade la funcionalidad de generar clases Java como resultado.
- **Java CUP**: Proyecto de software libre que provee al programador de un generador de analizadores sintácticos en Java rápido y eficiente.
- **JavaCC**: Herramienta que genera de manera global parsers completos para lenguajes, es decir, une en una herramienta el generador de analizadores léxicos y de analizadores sintácticos.

El uso de cada uno de ellos está condicionado fundamentalmente a las preferencias del usuario final así como sus preferencias por el uso de software libre o propietario. Una persona que conozca bien las herramientas lex y yacc o sus versiones libres flex y bison seguramente prefiera utilizar un tándem formado por la unión de Jlex o Jflex junto con BYacc o Java CUP, sin embargo una persona que nunca haya utilizado este tipo de herramientas posiblemente preferirá utilizar

JavaCC debido a que únicamente debe aprender a utilizar una herramienta. En principio se supone según la documentación existente comprobada de cada uno de ellos que la dificultad es similar siendo quizás un poco más complejas las especificaciones de las gramáticas JavaCC debido a que confluyen en la misma gramática el análisis léxico y sintáctico.

Como ya se ha comentado anteriormente en este proyecto se ha optado por las herramientas de software libre Java CUP y Jflex sobre las cuales se podrán realizar una valoraciones finales esta vez en base a datos prácticos reales en las conclusiones del presente documento.

7. Especificación de requisitos

En el presente capítulo se presenta un estudio de los requisitos de la aplicación desde un punto de vista técnico y de análisis, es decir, se omiten las alusiones a consideraciones propias de los lenguajes a implementar que ya han sido estudiadas en la parte I de la presente documentación.

7.1. Visión global

El desarrollo del presente proyecto está formado por dos partes claramente diferenciadas en lo que a programación se refiere, la implementación del traductor y la creación de un interfaz gráfico.

Se mencionaba en la introducción el fin de obtener un carácter didáctico en la aplicación a desarrollar debido a la procedencia del mismo (la extinta asignatura de Compiladores e Intérpretes de la Escuela Universitaria de Informática de Universidad Politécnica de Madrid) y es por ello que no se debe dejar de contemplar en los requisitos esta propiedad para poder ser utilizado con fines de estudio y análisis.

Se desea también que la aplicación sea multiplataforma e independiente del sistema operativo que se utilice, para evitar así, que los estudiantes o personal académico que desee utilizar la actual herramienta se vea obligada al uso de plataformas privativas y de caros costes. Por ello y por ser uno de las características del programa el ser implementado con un lenguaje orientado a objetos se utilizará Java como lenguaje en todo el desarrollo.

Los requisitos de implementación deben ser estudiados de manera independientes para cada una de las partes ya mencionadas ya que estas tienen una entidad y complejidad suficiente como para agruparlos todos.

7.2. Interfaz gráfico

Una de las ventajas al realizar el programa en Java es que podemos utilizar la funcionalidad que los sistemas gráficos proporcionan para implementar un entorno amigable e intuitivo similar a cualquiera de los programas actuales basado en sistemas de escritorio.

La implementación del interfaz gráfico de este tipo es además lo que posibilita que el programa final tenga un carácter didáctico ya que a través de él se puede mostrar un seguimiento de la traducción o de los análisis realizados de

manera rápida y sencilla. Igualmente en futuras ampliaciones se podrá mostrar cualquier característica de la máquina virtual.

Se pretende que el interfaz permita realizar todas las operaciones necesarias para escribir y traducir JSPascal sobre la misma pantalla consiguiendo así un sistema sencillo y eficaz que no consuma demasiados recursos.

Las funcionalidades que se requieren en el interfaz gráfico son las siguientes:

- Integrar un editor de texto sencillo con las funciones básicas como copiar, pegar, cortar, guardar.... que ofrezca ayudas a la programación.
- Creación de un sistema de log visible que permite al usuario en tiempo real comprobar que hace el programa mientras se realizan las operaciones de traducción o en una futuro ampliación de interpretación.

7.3. Traductor

Los requisitos aquí son simples ya que se pretende obtener un traductor capaz de traducir programas escritos en lenguaje JSPascal definido anteriormente al lenguaje máquina JMPascal ya definido.

Desde un punto de vista de análisis el traductor debe cumplir los siguientes requisitos:

- Tener un desarrollo basado en la teoría de la programación orientada a objetos.
- Realizar el análisis y la traducción de una manera rápida y eficaz.
- Permitir escalabilidad y ampliación del mismo con nuevas funcionalidades.

Funcionalmente el traductor debe ajustarse al siguiente funcionamiento:

- El traductor debe leer un fichero de entrada escrito en lenguaje JSPascal (identificado por la extensión pas) que será el fichero abierto y seleccionado en el editor del interfaz gráfico para analizarlo y traducirlo.
- Si el análisis es correcto debe generar un fichero de salida con el código JMPascal generado. Este fichero se llamará igual que el fichero de entrada pero con la extensión mps.
- Proveer de un sistema que permita seguir y analizar las acciones que se realizan en el análisis en tiempo real.
- No se considera la recuperación de errores por lo que la traducción será abortada en el momento en que se encuentra un error léxico, sintáctico o semántico.

8. Implementación

Antes de pasar a realizar el estudio como tal se deben definir las tecnologías a utilizar, las cuales han sido obviadas en el apartado de requisitos. Como ya se ha indicado desde la introducción el lenguaje a utilizar será Java, haciendo uso para ello de las herramientas libres Jflex y Java CUP.

Se utilizará en el desarrollo el entorno de Java de Sun en su versión 1.4.1 usando para la programación de los interfaces gráficos el framework Swing de Java.

Para desarrollar las dos partes diferenciadas en el capítulo de requisitos es necesario realizar el desarrollo de tres paquetes Java:

- **compilador**: Paquete que contiene las clases necesarias para el desarrollo del traductor del lenguaje JSPascal al JMPascal
- **user**: En este paquete se implementan todas las clases del interfaz gráfico que serán a su vez las clases con las que interactuará el usuario final de la aplicación.
- **utils**: Las clases que componen este paquete son clases que proveen utilidades como manejo de cadenas, semáforos, etc de las cuales algunas no han sido implementadas por el autor del presente proyecto. Se puede encontrar referencias a estas clases y a su implementación en la bibliografía de esta documentación.

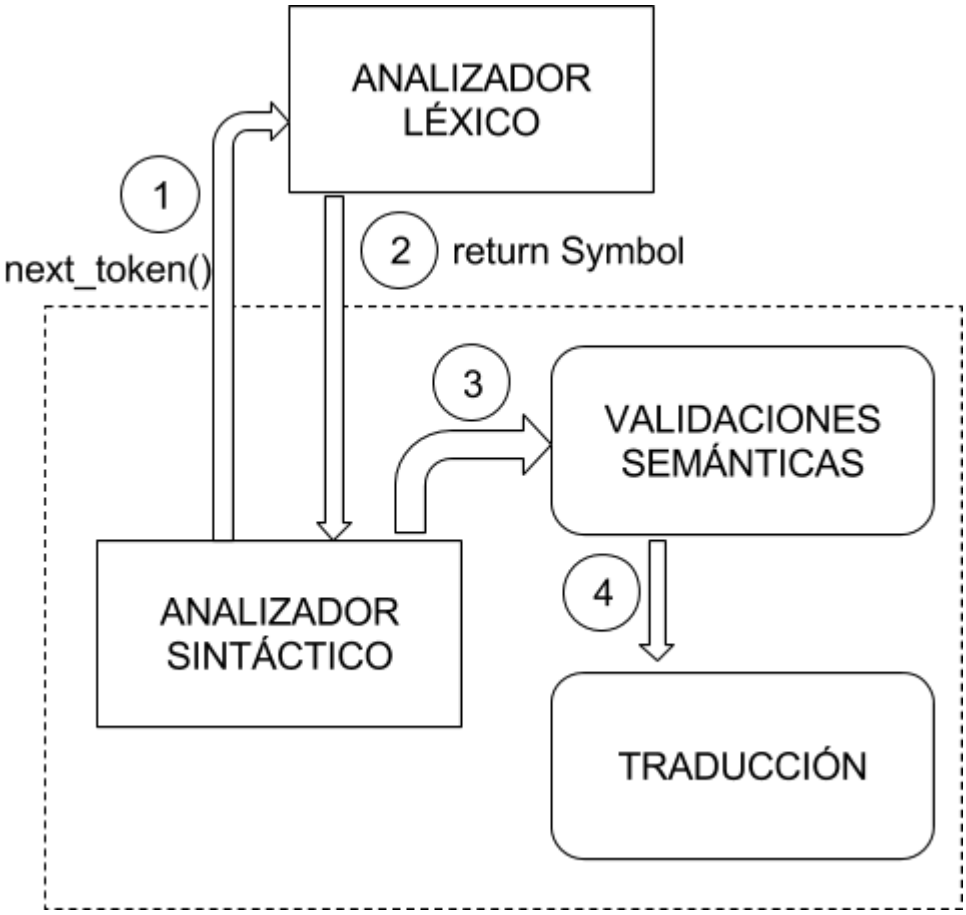
Además de los paquetes anteriores el desarrollo añade también los paquetes necesarios para que los analizadores generados por Jflex y Java CUP funcionen, estos paquetes contienen las librerías de ambas aplicaciones así como el entorno de ejecución de Java CUP.

8.1. Implementación del traductor

Para llevar a cabo la implementación del traductor es necesario tener una idea global del funcionamiento del mismo por lo que inicialmente procederemos a realizar un análisis.

8.1.1. Análisis global del traductor

A partir del siguiente gráfico se puede entender el funcionamiento global del traductor:



Tal y como se aprecia en el gráfico el analizador sintáctico es la pieza central del traductor. Será éste el que dirija el resto del análisis y la traducción y además integrará los análisis semánticos (exceptuando algunos básicos que se delegan directamente al analizador léxico por sencillez) y finalmente genera la traducción.

8.1.2. Implementación del analizador léxico

El funcionamiento del analizador léxico del presente proyecto funcionará de una manera habitual. Dada la ruta de un fichero se procederá a la lectura secuencial de la misma formando bloques que coincidan con alguna de las piezas sintácticas del lenguaje. En la implementación completa del traductor el análisis es dirigido por el analizador sintáctico por lo que las peticiones de lectura de la siguiente pieza sintáctica serán solicitadas por el analizador sintáctico.

El uso de Jflex con Java CUP facilita esta operación ya que el analizador generado proporciona el método next_token() que leerá el siguiente token de la entrada y se lo devolverá directamente al analizador sintáctico.

Siempre que se reconoce una pieza sintáctica y ésta es válida o útil (los comentarios y los separadores no se consideran útiles para el analizador sintáctico) se devolverá un objeto de la clase Symbol que tendrá siempre como parámetros el identificador que representa dicho terminal en la clase Sym generada por Java CUP, el número de línea y de columna del fichero de entrada en el que se ha encontrado dicho token.

Algunas piezas sintácticas o tokens necesitarán algún dato más para que posteriormente el analizador sintáctico pueda rellenar o consultar la tabla de símbolos. Por ejemplo para los identificadores necesitaremos el nombre del propio identificador para insertarlo en la tabla de símbolos cuando sea definido comprobando antes si es un identificador repetido en el contexto actual. Para

almacenar estos datos al objeto de la clase Symbol se le puede pasar un objeto de la clase Object en el que se almacene toda la información que se pueda necesitar posteriormente.

La implementación del analizador léxico del lenguaje JSPascal pasa por la definición de la gramática Jflex para el lenguaje, especificado en el capítulo 3. En los siguientes apartados se estudiará la implementación de dicha gramática.

8.1.2.1. Declaraciones Java

En este apartado simplemente se define el paquete al que pertenece la clase generada, *compilador*, y los import necesarios en el código implementado en el resto de la gramática.

8.1.2.2. Declaraciones Jflex

En la implementación se han utilizado las siguientes declaraciones (se puede ver una definición mayor de las posibles declaraciones Jflex en el anexo A.1.2 del presente documento):

- **%class Lexer**: Indica que la clase generada por Jflex se llama Lexer y será generada en el fichero Lexer.java

- **%unicode**: Especifica el juego de caracteres que se utiliza en los ficheros de entrada al analizador generado.
- **%ignorecase**: Hace que las expresiones regulares que se definan posteriormente en la gramática no distinguen entre mayúsculas y minúsculas. Esto facilita mucho las expresiones regulares.
- **%cupsym Sym**: Indica el nombre de la clase generada por Java CUP que contiene las constantes enteras con la definición de los terminales y no terminales.
- **%cup**: Activa la compatibilidad con Java CUP, es decir, genera un analizador compatible con el runtime de Java CUP mediante la inclusión del método next_token().
- **%int**: Con esta opción se consigue que el método yylex devuelve valores numéricos de tipo entero.
- **%line**: Convierte en pública la variable que almacena el número de línea que se está analizando.
- **%column**: Hace pública la variable que almacena el número de columna que se está analizando.
- **%char**: Mediante esta declaración se consigue que la variable que almacena el número de caracteres leídos desde el comienzo del fichero tenga carácter público.
- **%public**: Añadiendo esta declaración se consigue que la clase que se genera sea pública. Necesario si se desea interactuar con Java CUP.

Las declaraciones anteriores facilitan por lo tanto una de las especificaciones puramente léxicas como es la insensibilidad a mayúsculas y minúsculas del lenguaje.

8.1.2.3. Código Java

Se definen una serie de métodos en la parte destinada a tal efecto que se utilizarán posteriormente en el código de las producciones o desde otros métodos o clases.

En primer lugar se implementan tres métodos públicos que permitirán a otras clases solicitar el número de línea, columna o carácter:

- **public int getLine()**: Devuelve el número de línea del fichero de entrada que se está analizando.
- **public int getColumn()**: Devuelve la columna en la que está analizando el scanner en cada momento.
- **public int getCharacter()**: Devuelve un valor entero que indica el número de caracteres leídos desde la entrada.

Se definen además dos métodos privados que validan si las constantes enteras y reales que se detecten en la entrada son válidas:

- **private boolean esEnteraValida(String cte)**: Analiza las constantes enteras que se detectan en la entrada, comprobando que el valor absoluto de la misma es menor o igual que 65535, el máximo número permitido para los números enteros, representado por la constante JMPascalVM.MAXINT. Devuelve true si la constante es válida y false en caso contrario.
- **private boolean esRealValida(String cteReal)**: Comprueba cada una de las constantes reales que se detectan en la entrada. Se comprueba en este método que la parte entera está en el rango de los enteros y que la parte decimal tiene 6 o menos decimales. En caso de que la constante no sea válida se devuelve false y true si ésta es válida.

Por último se define un método que recibe como parámetro una cadena y devuelve otra cadena que será igual que la cadena dada como parámetro excepto si contiene dos comillas simples seguidas que serán sobrescritas como una sola comilla. Dicho método es también privado y se define como:

- **private String traduceCadena(String cadena)**: Devuelve una cadena suprimiendo las comillas simples que se utilizan en el lenguaje JSPascal para identificar las cadena de caracteres.

8.1.2.4. Comentarios

Gracias al operador ~ que ha incorporado Jflex la definición de las expresiones regulares para definir los dos tipos de comentarios que existen en el lenguaje son muy sencillos quedando éstas declaradas como se muestra a continuación:

- **"(*"~"*)"** : Comentarios del estilo *(* Esto es un comentario *)*
- **"{"~"}"** : Comentarios del estilo *{ Esto es un comentario }*

Cuando se reconoce un comentario no se hace nada, es decir, simplemente se ignora la entrada, ya que para el analizador sintáctico el comentario no tiene valor alguno.

8.1.2.5. Literales

Los literales en JSPascal se identifican de la misma manera que en Pascal, es decir, comienzan y terminan por el carácter de la comilla simple **'**. Además para que el propio literal puede contener una comilla simple se incluirán dos comillas simples, es decir, para representar el literal *Esto es un literal de 'JSPascal'* se debe definir como *'Esto es un literal de ''JSPascal'''*.

Debido a la sencillez del resto del léxico la expresión regular que identifica los literales es sin duda la más compleja de toda la gramática, la cual queda definida como **\'([^\']|\'\')*\'**

Los literales se representan mediante la constante pCADENA, y necesitan además del valor numérico que representa dicho token, el número de línea, el número de columna y la propia cadena que forma dicho literal. El literal no se pasa tal cual ha sido leído en el fichero de entrada sino que se realiza un tratamiento del mismo para quitarle la comilla inicial y final así como para dejar una sola comilla cuando el propio literal contiene dicho carácter, es decir, siguiendo con el ejemplo anterior cuando el analizador léxico detecte la entrada *'Esto es un literal de ''JSPascal'''* realmente pasará como parámetro al objeto Symbol un String formado por la cadena *Esto es un literal de 'JSPascal'*. Dicha transformación se realiza con la ayuda del método traduceCadena ya comentado anteriormente.

8.1.2.6. Palabras Reservadas

Para reconocer las palabras reservadas simplemente se definen éstas como expresiones regulares utilizando la sintaxis de Jflex. Como no es necesario indicar en las expresiones regulares la posibilidad de definirlas indistintamente en mayúsculas o minúsculas se opta por definir todas ellas en mayúsculas. La expresión regular Jflex que representa a cada una de las palabras reservadas es por tanto la propia palabra reservada en mayúsculas, por supuesto se podrían haber definido en minúsculas o incluso con alternancia de mayúsculas y minúsculas.

Todas las palabras reservadas están definidas en el fichero de símbolos mediante el prefijo “pr” seguido de la propia palabra reservada en mayúsculas, es decir, para la palabra reservada AND se utiliza la constante prAND.

Las palabras reservadas no necesitan ningún tipo de información adicional por lo que no se pasa ningún parámetro adicional al objeto Symbol creado.

8.1.2.7. Identificadores

Los identificadores del lenguaje deben comenzar por letras y estar formado por letras y dígitos. Para definir la expresión regular que reconoce los identificadores se hace uso de las macros de Jflex destinadas a reconocer el conjunto de letras válidas **[:jletter:]** y el conjunto que reconoce todas las letras y números válidos **[:jletterdigit:]**. El reconocimiento de los identificadores queda por tanto reducido a la expresión regular Jflex **[:jletter:][:jletterdigit:]***

Es intuitivo que de los identificadores necesitamos además de la posición del fichero en la que se han detectado y la constante que identifica al token, pID, el propio identificador almacenado accesible mediante el método *yylex()* y que será almacenado en el objeto Symbol devuelto mediante un objeto de la clase String.

8.1.2.8. Constantes enteras y reales

Existen dos tipos de constantes numéricas en el lenguaje JSPascal como ya se ha comentado anteriormente:

- **Constantes enteras:** formadas por la consecución de números enteros cuyo valor debe estar contenido en el rango que va desde -65535 hasta 65535
- **Constantes reales:** Formadas por la consecución de números reales cuya parte entera igualmente estará contenida en el rango que va desde -65535 hasta 65535

8.1.2.9. Separadores

Los separadores al igual que los comentarios no tienen ningún valor sintáctico en el lenguaje JSPascal por lo que cuando se detectan simplemente se consume la entrada sin realizar ninguna acción.

Para detectar el conjunto de separadores del lenguaje se utiliza la siguiente expresión regular: **[\ \t\n\r]+**

8.1.2.10. Símbolos del lenguaje

Los símbolos del lenguaje se representan por el propio símbolo encerrado entre comillas para evitar que Jflex los considere metacaracteres. Se definen primero aquellos que están formados por dos caracteres y posteriormente los que simplemente constan de un carácter.

Al contrario de lo que pasa en otras implementaciones con herramientas como lex o incluso Jflex se ha decidido definir una expresión regular por cada uno de los símbolos aunque estos estén formados por un solo carácter. De esta manera se consigue que los errores de caracteres inválidos se detecten directamente en el analizador léxico. Además se simplifica la creación del objeto Symbol para estas piezas sintácticas ya que únicamente es necesario identificarlas por la constante que identifica al terminal en cuestión mientras que si se realiza mediante un único token es necesario además pasar como parámetro un objeto de la clase String o Character que almacene el texto de dicho token, obligando al analizador sintáctico a comprobar nuevamente el contenido de dicho objeto.

8.1.2.11. Detección de errores léxicos

La detección de errores léxicos nombrada anteriormente se realiza en la última producción de la gramática Jflex declarada mediante la expresión regular ****. que se cumplirá siempre que exista un carácter en el fichero de entrada que no

pertenece al lenguaje y que además se encuentre fuera de un bloque de comentarios.

Siempre que el analizador entre por esta producción se lanzará una excepción de la clase ErrorLexicoException que abortará la traducción del programa.

8.1.2.12. Gramática Jflex

Juntando todos las puntos vistos obtenemos la gramática Jflex que se puede ver como parte del código fuente.

Con la gramática generada podemos además hacer una relación de las constantes necesarias para definir los diferentes tokens, entre paréntesis se muestra el valor entero que representa la constante:

- EOF (0)
- error (1)
- prAND (2)
- prARRAY (3)
- prBEGIN (4)
- prCONST (5)
- prDIV (6)
- prDO (7)
- prDOWNTO (8)
- prELSE (9)
- prEND (10)
- prFOR (11)
- prFUNCTION (12)
- prIF (13)
- prMOD (14)
- prNOT (15)

- prOF (16)
- prOR (17)
- prPROCEDURE (18)
- prPROGRAM (26)
- prRECORD (27)
- prREPEAT (28)
- prTHEN (29)
- prTO (30)
- prTYPE (31)
- prUNTIL (32)
- prVAR (33)
- prWHILE (34)
- sSUBRANGO (35)
- sASIGNACION (36)
- sCORAPER (37)
- sCORCIE (38)
- sCOMA (39)
- sDISTINTO (40)
- sDOSPUNTOS (41)
- sIGUAL (42)
- sMAS (43)
- sMAYOR (44)
- sMAYORIGU (19)
- sMENOR (20)
- sMENORIGU (21)
- sMENOS (22)
- sPARAPER (23)
- sPARCIE (24)
- sPOR (25)
- sPUNTO (45)
- sPUNTOCOMA (46)
- pID (47)
- pCADENA (48)
- pCteEntera (49)
- pCteReal (50)

Todas estas constantes son definidas como public static final int en la clase Sym del paquete compilador. Esta clase es generada de manera automática por Java CUP por lo que si en versiones posteriores del presente proyecto se desea

mantener el valor de las constantes se deberá realizar manualmente la edición de dicha clase.

A partir de la gramática JFlex se generará el código Java que implementa nuestro analizador léxico integrable con Java CUP o con cualquier otro código Java mediante la clase principal generada de nombre Lexer.

8.1.3. Implementación del analizador sintáctico, analizador semántico y traductor

Se estudiará en este apartado la implementación del analizador sintáctico que es el núcleo del traductor ya que es el que dirige todo el análisis pero que además integra la gran mayoría del análisis semántico y dirige la generación de código.

8.1.3.1. Análisis inicial

Anteriormente se han comentado dos aspectos importantes de la implementación del analizador sintáctico:

- El analizador sintáctico dirige el proceso de traducción: esto implica que es el encargado, además de solicitar las piezas sintácticas al analizador léxico, de

controlar el proceso del análisis semántico y de la traducción al lenguaje JMPascal.

- Se utilizará la herramienta Java CUP para generar el analizador sintáctico: al utilizar una herramienta que generará la clase encargada de realizar el analizador sintáctico debemos empotrar en éste el analizador semántico también (ya que será el que tenga acceso a la tabla de símbolos en tiempo real) y el control del generador de código.

Con estas ideas se puede pensar en implementar el analizador sintáctico en dos fases:

- Fase 1: Creación de la gramática Java CUP que realizará el análisis sintáctico y el poblado de la tabla de símbolos
- Fase 2: Inclusión del análisis semántico y de la generación de código sobre la gramática anterior.

Para desarrollar la fase 1 deben estar muy claras las especificaciones del lenguaje descritas anteriormente en el capítulo 2. Teniendo clara dichas especificaciones se debe desarrollar la gramática Java CUP necesaria.

El desarrollo de la fase 2 consiste en la modificación de la gramática obtenida anteriormente incluyendo para ello acciones (código Java) en las producciones que se necesite que realicen el análisis semántico y en su caso la traducción del código. Es importante tener en cuenta que el tipo de análisis que realiza Java CUP es

ascendente, concretamente LALR, lo que significa que las formaciones sintácticas no se obtienen de la misma manera en la que están escritas sino que se van formando a la vez que el árbol de análisis comenzando por las hojas del árbol y terminando por la raíz.

8.1.3.2. Implementación de la tabla de símbolos

Dado que el análisis sintáctico integrará el análisis semántico la traducción es necesario conocer la implementación que se realizará de la tabla de símbolos en este punto de cara a poder utilizar correctamente durante dicho análisis.

La tabla de símbolos implementada permite la gestión de todos los tipos de símbolos necesarios y la representación necesaria de niveles que define el lenguaje y la máquina JMPascaVM anteriormente definida. De esta manera su implementación se basa principalmente en tres clases del paquete compilador. Estas clases son “Objeto”, TablaSimbolos y NivelTablaSimbolos.

La **clase Objeto** representa un objeto de la tabla de símbolos y como ya se ha mencionado anteriormente los diferentes tipos de objetos del lenguaje se definirán con clases distintas que serán clases que hereden de esta clase principal o incluso de otras clase objeto, por eso, cada clase tendrá unos métodos u otros para según la necesidad del tipo de objeto. Las clases de objeto definidas se

corresponden con cada uno de los objetos que definen el lenguaje en lo que a símbolos se refiere y concretamente:

- **ObjetoCte**: clase base para la definición de constantes.
- **ObjetoCteEntera**: clase que hereda de la clase ObjetoCte y que define los objetos para las constantes enteras permitiendo almacenar el valor de dicha constante.
- **ObjetoCteLiteral**: clase que hereda de la clase ObjetoCte y que define los objetos para las constantes de tipo literal permitiendo almacenar el valor de dicha constante así como la etiqueta asignada a la misma en la tabla de símbolos.
- **ObjetoCteLogica**: clase que hereda de la clase ObjetoCte y que define los objetos para las constantes lógicas permitiendo almacenar el valor de dicha constante.
- **ObjetoCteReal**: clase que hereda de la clase ObjetoCte y que define los objetos para las constantes reales permitiendo almacenar el valor de dicha constante.
- **ObjetoEtiquetaEnumerado**: extiende la clase ObjetoCteEntera y se usa para los tipo enumerados.
- **ObjetoPrograma**: clase que hereda directamente la clase Objeto para almacenar los objetos de tipo programa y poder consultar el identificador de subprograma.

- **ObjetoProc**: clase que extiende la clase ObjetoPrograma para almacenar los procedimientos predefinidos permitiendo además gestionar el nivel de anidamiento y los parámetros del mismo.
- **ObjetoProcPredefinido**: clase que extiende la clase ObjetoProc para almacenar los procedimientos predefinidos.
- **ObjetoTipo**: clase base que extiende de la clase Objeto para definir los tipos del lenguaje.
- **ObjetoTipoEnumerado**: extensión de la clase ObjetoTipo para los tipos enumerados permitiendo gestionar sus elementos y el tipo de los mismos.
- **ObjetoTipoPredefinido**: extensión de la clase ObjetoTipo para los tipos predefinidos.
- **ObjetoTipoRegistro**: extensión de la clase ObjetoTipo para la definición de tipos registro permitiendo gestionar sus campos.
- **ObjetoTipoTabla**: extensión de la clase ObjetoTipo para la definición de tipos matriz, permitiendo gestionar su tipo de elemento y rango.
- **ObjetoVariable**: clase base que hereda de Objeto para la definición de variables. Permite gestionar lo relativo a la dirección de la variable y el nivel en el que se encuentra.
- **ObjetoCampo**: extiende la clase ObjetoVariable para definir los campos de una variable de tipo registro.
- **ObjetoParametro**: extiende la clase ObjetoVariable para almacenar los objetos de tipo parámetro permitiendo además conocer si el parámetro es por valor o por referencia.

Para cada tipo de objeto, según la necesidad de los datos a almacenar sobre los mismos se han utilizado diferentes estructuras estáticas o dinámicas que mejor se adaptasen del lenguaje Java consiguiendo que la implementación de los objetos no sea especialmente rígida de cara a futuras ampliaciones del lenguaje.

La **clase NivelTablaSimbolos** almacena la tabla de símbolos de un nivel estático. Para almacenar la tabla se ha optado por utilizar la clase HashMap que implementa una tabla Hash. La clave de esta tabla hash serán los propios identificadores de los símbolos a insertar. Los objetos que se insertan con cada clave son objetos de la clase “Objeto” o de clases que heredan de la misma. Esta clase además tiene dos objetos NivelTablaSimbolos que son referencias a el objeto del nivel anterior y el objeto del nivel siguiente. De esta manera podemos mantener una estructura de pila que representa la estructura de los niveles de ejecución. Las instancias de esta clase se realizan desde la clase TablaSimbolos cada vez que se crea un nivel nuevo.

Los principales métodos implementados en esta clase que permiten entender su funcionamiento son:

- **public NivelTablaSimbolos(NivelTablaSimbolos anterior, int n):** constructor de la clase. Recibe como parámetro una referencia al nivel anterior y el nivel que va a almacenar.

- **public int getNivel()**: devuelve el nivel que almacena el objeto

- **public void setNivel(int n)**: fija el nivel sobre el que almacena la info el objeto

- **public NivelTablaSimbolos getNivelAnt()**: devuelve el objeto que almacena la información del nivel anterior

- **public void setNivelAnt(NivelTablasimbolos anterior)**: fija el objeto que almacena la información del nivel anterior

- **public NivelTablaSimbolos getNivelSig()**: devuelve el objeto que almacena la información del nivel siguiente

- **public void setNivelSig(NivelTablasimbolos siguiente)**: fija el objeto que almacena la información del nivel siguiente

- **public Object getObjeto(String nombre)**: devuelve el objeto asociado al identificador nombre o null si no existe

- **public void insertaObjeto(String nombre, Object objeto)**: inserta un objeto en la tabla de símbolos con la clave nombre. Eleva la excepción

NullPointerException si nombre u objeto son null. Eleva la excepción ObjetoExisteException si ya existe un objeto con esa clave en la tabla.

La **clase TablaSimbolos** es instanciada en el parser generado por cup y es única. Su cometido es guardar el conjunto global de la tabla de símbolos. Para ello ésta va creando objetos de la clase NivelTablaSimbolos y los utiliza a modo de pila. Se guarda siempre una referencia al nivel inicial (0 o predefinido) y al nivel activo que será aquel que se esté analizando.

Los métodos que esta clase ofrece son los necesarios para crear la tabla de símbolos y ejecutar las operaciones que sobre ella son necesarias: buscar, insertar y borrar:

- **public TablaSimbolos()**: constructor de la clase. Crea el primer nivel que será el nivel de objetos predefinidos y del identificador del programa principal. Marca como activo este primer nivel.

- **public Object buscarLocal(String nombre)**: devuelve el objeto que se corresponde con el identificador nombre si éste se encuentra en el nivel actual. En caso contrario devuelve null.

- **public Object buscarGlobal(String nombre)**: busca en el nivel actual el objeto que se corresponde con el nivel actual y devuelve el objeto. Si no lo

encuentra recorre la pila de niveles y busca el objeto en los niveles anteriores. Si tampoco lo encuentra devuelve null.

- **public void insertar(String nombre, Object objeto)**: inserta el objeto con identificador nombre en el nivel actual. Puede elevar las excepciones que eleva el método insertaObjeto de la clase NivelTablaSimbolos.

- **public void creaNivel()**: crea un nuevo nivel, realiza la ordenación de la pila y fija el nivel activo.

- **public void suprimirNivel()**: elimina el nivel actual reordenador la pila y fijan como nivel activo el nivel anterior al activo.

- **public void insertaPredefinido(String nombre, Object objeto)**: inserta el objeto predefinido y eleva las mismas excepciones que el método insertar. Además eleva la excepción InsercionIlegalException si el nivel actual no es el nivel predefinido.

8.1.3.3. Generación de código

Para generar el código traducido en JMPascal se ha implementado una clase de nombre *Generador* dentro del paquete compiler que básicamente tiene las siguientes características principales:

- dispone de un constructor que instancia la clase, al cual se le indica el nombre del fichero origen (fichero .pas) que se está traduciendo ya que lo usará como base para generar el fichero destino (fichero .mps).

- dispone de un atributo de tipo *String* y de nombre *codigo* que almacena en memoria el código generado.

- tiene definido un método por cada instrucción del lenguaje JMPascal cuyos nombres son del tipo g<instruccion> .

- dispone de un método de nombre *termina* al que se llama desde el parser cuando se finaliza la traducción y que provoca la escritura a disco del código generado en un fichero con path y nombre idéntico al fichero origen pero con la extensión .mps.

Por tanto para generar código el parser debe instanciar o tener acceso a un objeto, que será único por cada análisis realizado, de la clase Generador e ir llamando a los diferentes métodos g<instruccion> a medida que sea necesario, llamando en último caso al método termina del mismo objeto.

8.1.3.4. Especificación Java CUP

Como podemos ver con detalle en el anexo dedicado al *Estudio de Java CUP* la definición realizada en Java CUP y dado que usamos JFlex junto a Java CUP, la especificación de Java CUP que se puede encontrar junto al código fuente del presente proyecto tiene las características destacables que se exponen a continuación.

El bloque **action code** que encontramos tras la definición de paquetes incluye una serie de métodos auxiliares que permiten realizar operaciones como la inicialización del proceso de análisis, así como facilitar la carga de objetos en la tabla de símbolos tras su reconocimiento o la generación de código en aquellos casos que es necesario realizar cálculo de direcciones.

El bloque **parser code**:

- define el constructor de la clase Parser de Java CUP que es el que será invocado desde fuera para instanciar el analizador y que en nuestro caso hemos modificado para poder pasarle un objeto de la clase TablaSimbolos indicada anteriormente, un objeto de la clase Log que se comentará más adelante y el path al fichero que queremos analizar

- redefine otros métodos de la clase Parser (syntax_error(, unrecovered_syntax_error, report_fatal_error y report_error) de cara a mejorar la información proporcionada por el parser ante errores detectados en el análisis

Así mismo se define **el bloque de terminales y no terminales**, que dará pie a la generación de la clase Sym.java indicada anteriormente en el análisis léxico y que por tanto debe tenerse en cuenta para evitar que una modificación aquí genere errores en el analizador léxico, siendo recomendable gestionar esta clase de manera manual una vez se llega a un punto consolidado del desarrollo.

Tras el bloque de terminales se definen reglas de precedencia para dichos terminales y no terminales lo que nos permite definir de manera sencilla reglas que serían mucho más complejas de cualquier otra manera como la precedencia de los operadores aritméticos.

Por último se define la gramática indicando primero el no terminal inicial de dicha gramática con la cláusula *start with* que en nuestro caso quedará como *start with program* ya que es nuestro no terminal inicial. La gramática por sí sola ya generaría un parser que permite la validación sintáctica y que sería el paso correspondiente a la fase 1 indicada anteriormente.

La implementación realizada integra además en la gramática código Java por cada una de las reglas de la misma. Este código Java será el encargado de:

- rellenar la tabla de símbolos cuando proceda, como por ejemplo sucede en la regla inicial que almacena el objeto programa del programa global peor que antes inicializa los objetos predefinidos (para simplificar la gestión todos los identificadores de objetos de la tabla de símbolos se almacenarán en minúsculas)

- realizar las validaciones semánticas que correspondan en cada regla, siendo prácticamente el 100% de validaciones realizada en este código (recordemos que alguna validación básica se había delegado al analizador léxico) en base a los datos de la tabla de síḿbolos obtenidos en el mismo momento o en momentos anteriores del proceso de análisis

- realizar la generación de código cada vez que sea necesaria, para lo cual se apoya en la clase Generador comentada anteriormente

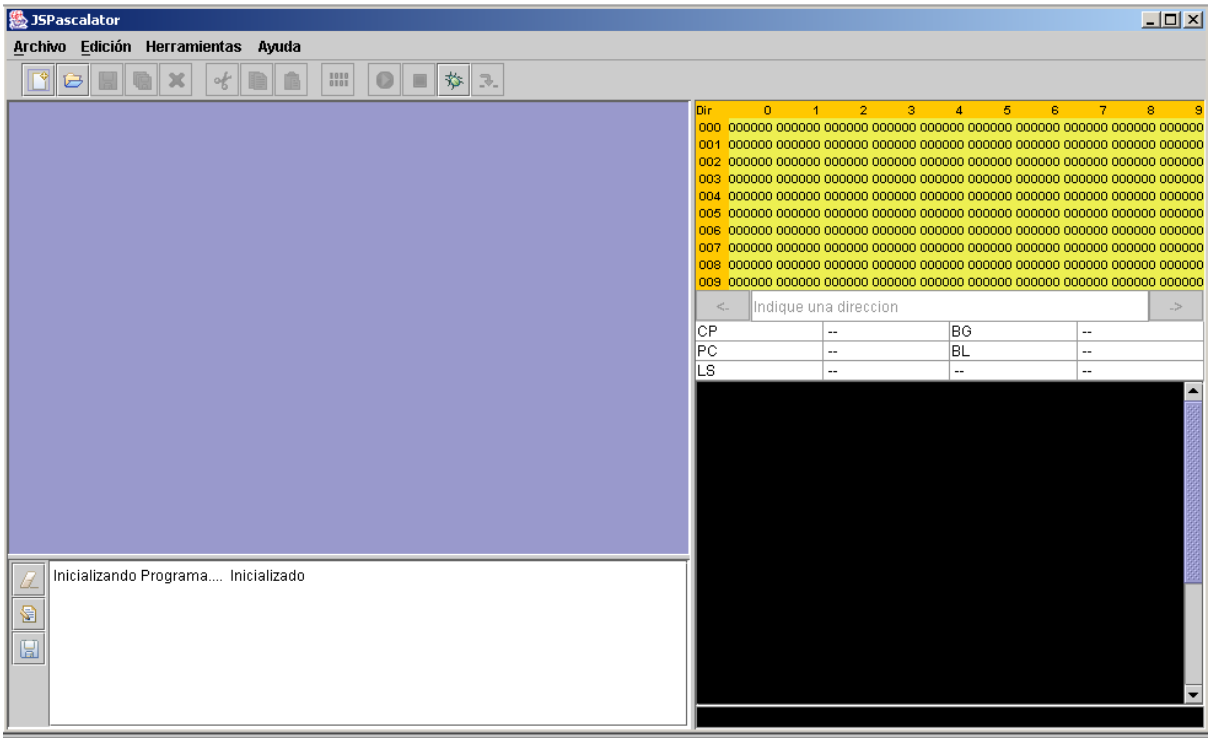
A partir de la especificación realizada basada en las indicaciones expuestas se genera mediante Java CUP el código Java del analizador sintáctico que además incluye el analizador semántico y las llamadas para poder generar código JMPascal. El código generado incluye multitud de clases siendo la principal la clase Parser que

puede instanciarse de una manera que permite su integración en el conjunto de cualquier aplicación Java a través de su método principal *parser*.

8.1.4. Implementación del interfaz gráfico

Era requisito de la aplicación la implementación de un interfaz gráfico que facilitase el uso del traductor y éste se ha realizado como una aplicacioń de escritorio con Java Swing.

El interfaz gráfico implementado es como el siguiente:



Se aprecian claramente varias zonas en el interfaz:

- en la parte superior existe una barra de menú que permite acceder a las distintas funcionalidades del software

- justo debajo encontramos una barra de herramientas que da acceso a las funcionalidades más usuales de la aplicación mediante botones gráficos de manera más accesible que el menú superior

- en la parte central izquierda se aprecia el espacio destinado el editor que incluye numeración de líneas y operaciones básicas de copiar, pegar y cortar. Por supuesto permite abrir ficheros, editarlos, guardarlos y se puede trabajar con varios a la vez.

- en la parte inferior izquierda vemos un cuadro de log en el que se mostrarán los procesos que se realicen de análisis y traducción o en el futuro de ejecución. Igualmente permite copiar el contenido, borrarlo o incluso guardarlo a disco

- en la parte derecha vemos una parte del interfaz destinada a una posible ampliación del software para la ejecución de programas en lenguaje JMPascal sobre la máquina JMPascalVM con visualización de la memoria de la máquina virtual, sus registros y la posibilidad de usar la entrada y salida estándar, pero que en el presente proyecto carece de uso aunque dada la

finalidad educativa del proyecto se ha incluido para dar visibilidad de su finalidad.

La integración del interfaz gráfico implementado con el analizador generado básicamente consta de:

- integración de la opción de compilación tanto en el botón de la barra de herramientas como en la opción del menú cuyo evento instancia la clase *Traductor* creada para la integración y que al elegir esta opción instancia la clase *Parser* y lanza el método *parser* de la misma que inicia todo el proceso de análisis y traducción

- dado que la finalidad es educativa se ha integrado una opción en el menú para poder realizar un análisis léxico cuyo evento instancia la clase *Traductor* creada para la integración y que al elegir esta opción instancia la clase *Lexer* y ejecuta el anaĺisis léxico mostrando por la ventan de log todos los símbolos reconocidos

Parte III - Pruebas y conclusiones

9. Pruebas

Se proporciona en este apartado una batería de pruebas de pruebas que permiten validar el buen funcionamiento del software diseñado e implementado para una gran variedad de casos y en todas las funcionalidades implementadas.

Las pruebas que se presentan consisten en ficheros fuentes cuyo contenido se puede encontrar junto al código fuente en el directorio programas:

- Ficheros fuentes JSPascal (*.pas)
- Ficheros traducidos a JMPascal (*.mps)

9.1. Batería de pruebas del traductor

La validación del software debe ser progresivo y por ello se deben realizar por partes las pruebas.

9.1.1. Prueba del analizador léxico del traductor

En primer lugar se procederá a probar que el analizador léxico del lenguaje JSPascal funciona de manera correcta. Para ello se hace uso del botón incorporado en el interfaz gráfico mediante el menú (Herramientas -> Análisis Léxico) que

permite realizar un análisis léxico independiente del resto del análisis. Se recuerda que este botón leerá un fichero fuente y proporcionará un listado con cada una de las piezas sintácticas encontradas indicando el texto que la define, la línea y columna en la que se encuentra dicho texto y la constante que define dicho token.

Para proceder a realizar esta prueba se abre el fichero de pruebas (*Archivo => Abrir*) y se hace clic sobre *Herramientas => Análisis Léxico* obteniendo en la consola de acciones el listado anteriormente señalado.

Para llevar a cabo esta prueba se hace uso del fichero **prueba1.pas**. Este fichero no tiene sentido desde el punto de vista sintáctico ni semántico por lo que no se debe prestar más atención que al léxico. En él aparecen diferentes palabras reservadas, identificadores, comentarios constantes, literales, etc. Al final del mismo encontramos un identificador no válido, niño, que provocará un error del análisis léxico y cancelará el resto del análisis reconociendo como último token válido un identificador mediante el texto “ni” y posteriormente el carácter no válido “ñ”.

Además existe otro fichero de pruebas destinado a probar el analizador léxico, **prueba2.pas**. En este fichero se encuentran en primer lugar una serie de constantes enteras y reales correctas las cuales no deben producir ningún error en el análisis. Posteriormente aparecen otras constantes entre comentarios que deben producir errores. Para realizar la prueba de errores se puede descomentar y

comentar las diferentes constantes erróneas para comprobar los errores que provocan.

Tras la prueba anterior se puede comprobar que el analizador léxico funciona aparentemente de manera correcta incluyendo además alguna comprobación semántica como puede ser el desbordamiento de las constantes enteras y reales.

9.1.2. Prueba del traductor

Se ha indicado anteriormente la razón por la cual no se ha realizado independiente del analizador sintáctico que permitiría en este punto realizar las pruebas del mismo por separado. Por esta razón es evidente que las siguientes pruebas son las correspondientes al traductor completo, lo que implica probar el analizador sintáctico y semántico así como el generador de código.

La razón por la cual se debe probar todos estos bloques del analizador es sencilla, como ya se ha explicado anteriormente el analizador sintáctico es el núcleo del traductor y es el que dirige el proceso de análisis y traducción por lo que no tiene sentido y no es posible la ejecución de los módulos de análisis semántico y traducción sin el uso del analizador sintáctico. Sin embargo si sería posible ejecutar por separado el analizador sintáctico pero no se va a realizar ya que obligaría bien a duplicar el código del analizador para separar el analizador semántico (recordemos que su código está integrado con el del analizador sintáctico) y eliminar las llamadas

al generador de código; bien a añadir una infinidad de sentencias if en el código en base a una variable que identifique si el análisis es total o simplemente es un análisis sintáctico consiguiendo en ese caso un código muy sucio y nada eficiente.

Para realizar las pruebas del analizador se han utilizado una gran cantidad de programas de prueba que permiten comprobar de manera progresiva el funcionamiento del traductor.

La primera prueba se realiza con el fichero **prueba3.pas**. Este sencillo programa comprueba que la declaración de un programa y la declaración de constantes se realiza de manera correcta. Para ello se realiza la declaración de dos constantes enteras, la segunda a partir de la multiplicación de la primera por sí misma, dos reales siendo obtenida la segunda de ellas a partir de la primera y la suma de una de las constantes enteras, además se declara una constante de tipo lógico a la que se le asigna la constante predefinida TRUE. Por último se define un literal con el valor ‘Hola caracola’.

Al realizar la traducción de este programa comprobamos que el análisis es correcto y comprobamos que la traducción se realiza de manera correcta aunque en este caso se comprueba que el programa generado no realiza ninguna acción ya que en el código fuente no existen instrucciones de código. Sin embargo sí que aparece la etiqueta correspondiente al programa principal *0 y la instrucción de terminación en el final que se generan a partir de las instrucciones begin y end. del

programa fuente. Cabe destacar que aunque en la declaración de las constantes existen operaciones aritméticas estas no generan instrucciones en la traducción ya que al tratarse de constantes los valores se calculan en el análisis y se almacenan en la tabla de símbolos no debiendo realizarse en el programa.

Se comprueba además cómo se genera la directiva #DEFLITERAL para la definición del literal definido.

Se generan además las dos directivas del compilador #TAMGLOBAL y #TAMFUNCION ambas con un 0 como parámetro.

La directiva #TAMFUNCION que se generará siempre no tiene interés ya que en esta implementación no se han incluido las funciones por lo que siempre será generada con el valor 0.

La directiva #TAMGLOBAL en este caso también tiene el parámetro 0 debido a que no hay declarada ninguna variable por lo que no es necesario reservar espacio en el área global.

En el siguiente programa de pruebas se comprueba que la declaración de variables se realiza de manera correcta. En el programa **prueba4.pas** podemos encontrar un programa que tampoco tiene instrucciones pero en el que se declaran dos variables enteras, una variable real y una variable lógica.

Al traducir el código vemos que en este caso tampoco se genera ninguna instrucción debido a que en el código fuente no existen instrucciones.

Lo que sí se comprueba es que la reserva de la memoria del espacio global para las variables se realiza correctamente ya que el espacio reservado es correcto si se tiene en cuenta que las variables de tipo entero y lógico ocupan una dirección de memoria y las reales ocupan dos sumando un total de 5.

El fichero **prueba5.pas** une las declaraciones de los dos programas anteriores y añade la asignación de valores constantes a las variables tanto desde constantes declaradas como de constantes directamente especificadas en el código fuente.

Tras pulsar sobre el botón de compilación se genera el código traducido que esta vez sí contiene instrucciones con la carga en la pila de las constantes y su asignación a direcciones de memoria. Podemos ver que la generalidad para una asignación viene dado por una carga de la constante en la pila de la memoria seguida de la carga de una dirección del área global en la pila de la cima y por último de la asignación que tiene como parámetro el número de posiciones de memoria que ocupa el tipo de la variable . Por ejemplo para la asignación de la línea 20 se genera el siguiente código:

```
08    CCONSTANTE  5
09    CDIRGLOBAL  1
10    AINDVALOR   1
```

```

```

En este código también se identifica como las constantes numéricas negativas definidas directamente en el código, generan una instrucción aritmética NEGATIVO ya que todas las constantes se reconocen como valores positivos. Se genera entonces el siguiente código para la línea 22 del código fuente:

```
14    CCONSTANTEREAL    45.87
15    NEGATIVO
16    CDIRGLOBAL  2
17    AINDVALOR   2
```

Cabe destacar también la asignación de la variable tercera en la línea 23 del fichero prueba5.pas lo cual genera el siguiente código:

```
18    CCONSTANTE 6
19    AREAL 0
20    CDIRGLOBAL 2
21    AINDVALOR 2
```

En el código anterior se identifica otra alteración de la estructura en la línea 19 del código generado provocado por la instrucción AREAL que permite la conversión automática entre el tipo entero y el real para realizar asignaciones, operaciones aritméticas, comparaciones convirtiendo valores enteros en memoria en valores reales para lo que se ocupan dos direcciones de memoria en vez de una añadiendo 0 en la parte decimal de dicho valor.

El siguiente fichero de prueba, **prueba6.pas**, permite comprobar la definición de tipos complejos: matrices y registros. El programa declara los tipos tipoMatriz y tipoRegistro. El primero de ellos es una matriz de una dimensión con índices de 1 a 10 y el segundo un registro formado por un campo entero uno real y otro lógico. Además se ha declarado una variable entera, una del tipo tipoMatriz y otra tipoRegistro. Al generar el código se comprueba que no se generan errores en el análisis/traducción y que la reserva de espacio se realiza adecuadamente ya que se genera la instrucción #TAMGLOBAL 15 que se corresponde con el tamaño de las variables declaradas en el código fuente.

El programa **prueba7.pas** amplia el programa anterior y realiza asignaciones en las variables declaradas. Permite comprobar que la manera en la que se obtienen la dirección de memoria varía ya que es necesario ejecutar una instrucción más para realizar el desplazamiento desde la dirección en la que se ubica la variable hasta la dirección del elemento o campo al que se realiza la asignación. Además en las asignaciones de la variables array se incluye también la instrucción que comprueba que el valor del índice dado es correcto.

En el código generado por la línea 17 del código fuente:

```
06    CCONSTANTE  2
07    CCONSTANTE  67
08    LIMITACION  1 10 1
09    CDIRGLOBAL  1
10    DELEMENTO   1 1
11    AINDVALOR   1
```

se comprueba cómo en primer lugar se carga la constante que indica el valor del índice. Posteriormente se carga la constante entera que se va a asignar en la cima de la pila y se comprueba que el índice especificado en la asignación es correcto mediante la instrucción LIMITACION cuyos parámetros son índice mínimo (1), índice máximo (10) y número de direcciones de memoria (1) usadas en la pila por encima del índice del elemento a utilizar para poder reordenar la pila tras realizar la comprobación de los límites. A continuación se carga en la pila la dirección global dónde comienza la variable matriz mediante la instrucción CDIRGLOBAL para posteriormente modificarla mediante la instrucción DELEMENTO que tendrá como parámetro el índice inicial de la matriz (1) y el tamaño en memoria de cada elemento de la matriz (1). Por último se realiza la asignación mediante la escritura en memoria.

En la línea 21 se realiza una asignación al campo real de la variable registro que genera el siguiente código:

```
22    CCONSTANTEREAL 29.87
23    CDIRGLOBAL 11
24    DCAMPO 1
25    AINDVALOR 2
```

Aquí la estructura se asemeja más a la de una asignación simple dónde podemos ver que la mayor diferencia es que tras realizar la carga de la dirección en

la cima de la pila se ha de modificar dicha dirección para lo que se genera la instrucción DCAMPO que tiene como parámetro el número de direcciones que hay que sumar a la dirección ubicada en la cima de la pila para posteriormente realizar la asignación.

La siguiente prueba permite comprobar que la generación de código de las operaciones aritméticas que se deben realizar en tiempo de ejecución son analizadas y traducidas correctamente. El fichero de prueba que se utiliza en este caso es **prueba8.pas** que realiza asignaciones a variables de la misma manera que se hizo en el fichero prueba5.pas pero con operaciones aritméticas involucradas.

Al visualizar el código generado comprobamos nuevamente que se crea una estructura de instrucciones que se repite en cada asignación. En primer lugar se van cargando los operandos (en este caso son todo constantes) a medida que se van leyendo y a la vez se generan las instrucciones aritméticas según las reglas de precedencia del lenguaje. Una vez realizadas todas las operaciones aritméticas se produce la carga de la dirección de la memoria y se realiza la asignación.
Realizando un análisis del código generado se comprueba que la generación de código ha sido correcta.

De todo el código generado merece la pena comentar la diferencia en el código generado por las líneas 17 y 18 del código fuente que son iguales con una modificación de prioridad mediante el uso de paréntesis:

```
08    CCONSTANTE 3
09    CCONSTANTE 4
10    CCONSTANTE 7
11    PRODUCTO
12    SUMA
13    CDIRGLOBAL 0
14    AINDVALOR 1
15    CCONSTANTE 3
16    CCONSTANTE 4
17    SUMA
18    CCONSTANTE 7
19    PRODUCTO
20    CDIRGLOBAL 0
21    AINDVALOR 1
```

la línea 17 genera las líneas 8 a 14 mientras que la la línea 18 genera las líneas 15 a 21. Se comprueba que en el primer caso al no existir paréntesis el producto tiene prioridad por lo que se realiza el producto y posteriormente la suma para finalmente realizar la asignación. Sin embargo en las líneas 15 a 21 se realiza primero la suma de las constantes 3 y 4 debido al uso de paréntesis y posteriormente se apila el 7 se realiza el producto para terminar realizando la asignación.

El resto del código es similar con las diferencias que suponen el uso de valores reales por lo que sólo interesa comentar cómo se convierten las constantes enteras en reales cuando es necesario mediante la instrucción AREAL. En la línea 25 del código fuente se realiza la asignación de una operación aritmética que mezcla constantes enteras y reales a una variable real. Por ello en el código generado cuando se va a realizar una operación entre una constante entera y una real se realiza una conversión de la constante entera de la cima de la cima de la pila a constante real. Sin embargo cuando la operación se realiza entre operandos

enteros no se realiza la conversión aunque estemos en el ámbito de una asignación a variable de tipo real:

```
64    CCONSTANTE 3
65    CCONSTANTEREAL 45.67
66    AREAL 2
67    PRODUCTOREAL
68    CCONSTANTE 8
69    CCONSTANTE 6
70    DIVISION
71    AREAL 0
72    RESTAREAL
73    CDIRGLOBAL 1
74    AINDVALOR 2
```

Para continuar con las pruebas del traductor se implementa en el fichero **prueba9.pas** un programa similar al anterior pero con la inclusión de variables como operadores en la parte derecha de las asignaciones o en el uso de índices. Este hecho provoca que antes de realizar operaciones sea necesario cargar la dirección de memoria en la que se ubica la variable a leer, posteriormente cargar el valor almacenado en dicha dirección y por fin utilizarla.

Por ejemplo la asignación de la línea 28 genera el siguiente código:

```
08    CDIRGLOBAL 0
09    CINDVALOR 1
10    CDIRGLOBAL 1
11    AINDVALOR 1
```

Podemos comprobar también en el código generado por la línea 40 como para leer el valor que se almacena en los campos de las variables de tipo registro se

utiliza la instrucción DELEMENTO. También se comprueba en este código la conversión automática del tipo entero a real aunque este proceda de carga de variables:

```
65    CDIRGLOBAL 21
66    DCAMPO 0
67    CINDVALOR 1
68    AREAL 0
69    CDIRGLOBAL 21
70    DCAMPO 1
71    AINDVALOR 2
```

La línea 42 del código fuente permite comprobar como el procedimiento para utilizar una variable como índice de otra variable de tipo matriz es el mismo que para utilizarla en una asignación y que se debe de hacer uso de la instrucción DELEMENTO para calcular la dirección correcta dónde se almacena el valor a leer y también de la instrucción LIMITACION para comprobar que el elemento a leer está dentro del rango de la matriz en cuestión:

```
78    CDIRGLOBAL 21
79    DCAMPO 0
80    CINDVALOR 1
81    CCONSTANTE 1
82    LIMITACION 1 10 0
83    CDIRGLOBAL 11
84    DELEMENTO 1 1
85    CINDVALOR 1
86    CDIRGLOBAL 0
87    CINDVALOR 1
88    SUMA
89    CCONSTANTE 4
90    SUMA
91    LIMITACION 1 10 1
92    CDIRGLOBAL 11
93    DELEMENTO 1 1
94    AINDVALOR 1
```

Cómo última prueba se ha realizado la asignación completa de una variable matriz a otra del mismo tipo y de una variable registro a otra del mismo tipo. En este caso se observa que en ningún caso se realiza modificación sobre la dirección cargada ya que se copian todas las posiciones de memoria que ocupan las variables:

```
95    CDIRGLOBAL 35
96    CINDVALOR 4
97    CDIRGLOBAL 31
98    AINDVALOR 4
99    CDIRGLOBAL 11
100   CINDVALOR 10
101   CDIRGLOBAL 21
102   AINDVALOR 10
```

El siguiente programa de prueba, **prueba10.pas**, añade al programa prueba9.pas una serie de instrucciones comentadas con errores sintácticos y semánticos comentados que se explican en el propio código. Se puede comentar y descomentar los errores para comprobar que se detectan de manera de correcta.

Con el programa **prueba11.pas** se continúan las pruebas del traductor y en este caso se prueban las operaciones lógicas y su asignación a variables. El código que se genera tiene la misma estructura que en los programas anteriores con operaciones aritméticas dónde lógicamente cambian las instrucciones generadas por las correspondientes para realizar las operaciones lógicas. Como ejemplo podemos analizar el código generado por la línea 23 del código fuente:

```
14    CCONSTANTE 4
15    CDIRGLOBAL 0
16    CINDVALOR 1
17    MENOR
18    CCONSTANTE 5
19    CCONSTANTE 4
20    MAYOR
21    CONJUNCION
22    CDIRGLOBAL 3
23    AINDVALOR 1
```

Se observa fácilmente que el código generado es correcto y que funciona de la misma manera que en los casos anteriores. Se cargan los operandos (ya sean constantes o variables) se realizan las operaciones con los operandos en la cima de la pila y al final se almacena el valor obtenido en la variable en cuestión.

Una vez que se ha probado la correcta generación de las operaciones aritméticas podemos comenzar a probar las instrucciones en las que estas operaciones tienen sentido: las estructuras de control.

En el programa **prueba12.pas** se realizan las pruebas de la estructura if-else. En este programa se comprueba como la traducción que realiza el software de dicha estructura es correcta. El caso más sencillo, líneas 9 y 10, representado por una sentencia if cuya condición es una constante lógica y sin parte else genera el siguiente código:

```
03    CCONSTANTE  1
04    BIFCOND     0
05    CCONSTANTE  4
```

```
06    CDIRGLOBAL  0
07    AINDVALOR   1
08    BIFINCOND   1
09    e0
10    e1
```

Analizando el código comprobamos que en primer lugar se calcula el valor de la condición que en este caso simplemente es una constante pero podría ser una serie de operaciones aritméticas y lógicas que finalmente almacene un valor lógico en la cima de la pila. A continuación se genera la instrucción BIFCOND que realiza la función if que recibe como parámetro la etiqueta condicional a la que debe saltar en caso de que en la cima de la pila exista un 0. A continuación se generan las instrucciones de la parte if y por último aunque en este caso no existe parte else se genera la instrucción BIFINCOND que evitaría ejecutar la parte else en caso de existir y haber entrado por la parte if. Esta instrucción fuerza un salto hasta la instrucción siguiente a la etiqueta condicional que tiene la instrucción como parámetro.

El resto de ejemplos funcionan de la misma manera. El caso más complejo del fichero es el generado por las líneas 26 a 35 del código fuente que genera el siguiente código:

```
49    CDIRGLOBAL  1
50    CINDVALOR   1
51    CCONSTANTE  5
56    CCONSTANTE  6
57    DISTINTO
58    CONJUNCION
59    BIFCOND     8
60    CCONSTANTE  6
```

```
61    CDIRGLOBAL  0
62    AINDVALOR   1
63    CCONSTANTE  0
64    CDIRGLOBAL  1
65    AINDVALOR   1
66    BIFINCOND   9
67    e8
68    CDIRGLOBAL  1
69    CINDVALOR   1
70    BIFCOND     10
71    CDIRGLOBAL  0
72    CINDVALOR   1
73    CCONSTANTE  2
74    DIVISION
75    CDIRGLOBAL  0
76    AINDVALOR   1
77    CCONSTANTE  0
78    CDIRGLOBAL  2
79    AINDVALOR   1
80    BIFINCOND   11
81    e10
82    e11
83    e9
```

Las líneas 49 a 58 calculan el valor de la condición del primer if. A continuación se genera la instrucción BIFCOND que saltará a la instrucción siguiente de la etiqueta e8, línea 68, si la condición resulta en un 0 en la cima de la pila. En las líneas 60 a 66 se generan las instrucciones correspondientes a las asignaciones de las líneas 27 y 28 del código fuente y por último una instrucción de salto incondicional para evitar ejecutar toda la parte else que como se puede observar saltará hasta la etiqueta e9 que es la última del código generado. En la línea 68 comienza la parte else y nuevamente lo primero que se genera es el cálculo de la condición del segundo if repitiendo nuevamente la estructura de la parte if anterior.

Analizando el código completo se aprecia claramente como las etiquetas condicionales se invierten en la generación de código, es decir, las etiquetas mayores se generan antes que las menores lo que permite que el curso de la posterior ejecución del código generado sea correcto.

La siguiente estructura a probar son los bucles implementados en este desarrollo: while y repeat. El programa **prueba13.pas** está implementado para ello con la inclusión de un bucle while y un bucle repeat. En primer lugar si se observa el código entero se puede comprobar como las etiquetas condicionales en el caso de los bucles se generan en orden ascendente en contra de lo que pasaba con las sentencias if-else.

El código que genera el bucle while, líneas 12 a 14 del código fuente, es el siguiente:

```
06    e0
07    CDIRGLOBAL 0
08    CINDVALOR 1
09    CCONSTANTE 4
10    MENOR
11    BIFCOND 1
12    CDIRGLOBAL 0
13    CINDVALOR 1
14    CCONSTANTE 1
15    SUMA
16    CDIRGLOBAL 0
17    AINDVALOR 1
18    BIFINCOND 0
19    e1
```

Se observa que lo primero que se genera es una etiqueta condicional. Esta etiqueta es la que permite volver al principio del bucle en caso necesario. A continuación se calcula el valor de la condición del bucle (líneas 7 a 10 del código generado) y posteriormente se genera una instrucción de bifurcación condicional que saltará a la etiqueta e1 en caso necesario provocando el fin del bucle. Las líneas 12 a 17 del código generado se corresponden con la asignación que se realiza en el interior del bucle en la línea 12 del código fuente. Por último la línea 18 tras ejecutar una vuelta del bucle fuerza un salto a la etiqueta e0 que se generó inicialmente para realizar nuevamente la comprobación de la condición y ejecutar o no nuevamente las instrucciones del bucle.

El código generado por el bucle repeat definido en las líneas 16 a 18 es el siguiente:

```
20    e2
21    CDIRGLOBAL 0
22    CINDVALOR 1
23    CCONSTANTE 1
24    RESTA
25    CDIRGLOBAL 0
26    AINDVALOR 1
27    CDIRGLOBAL 0
28    CINDVALOR 1
29    CCONSTANTE 0
30    IGUAL
31    BIFCOND 2
```

Nuevamente en primer lugar se ejecuta la etiqueta condicional que indica el principio del bucle, sin embargo en este caso no se comprueba inicialmente la condición sino que se generan primero las instrucciones del bucle, líneas 21 a 26

que se corresponden con la línea 16 del código fuente. A continuación en las líneas 27 a 30 se realiza el cálculo de la condición del bucle y en caso necesario se realiza un salto al principio del bucle mediante la generación de la etiqueta BIFCOND que apunta a la etiqueta generada al principio del bucle.

El programa **prueba14.pas** es un programa con todo tipo de operaciones y estructuras de control con errores sintácticos y semánticos comentados que se pueden descomentar de uno en uno para comprobar que se detectan los errores al realizar la compilación. El programa y sus errores están explicados en el propio código mediante los comentarios que contiene.

Hasta ahora se han realizado pruebas con programas que únicamente tienen instrucciones en el bloque principal del programa. En el siguiente bloque de pruebas se realizarán pruebas con subprogramas (ya sean predefinidos o no) y llamadas a los mismos.

El fichero **prueba15.pas** se prueba la generación de código de los procedimientos predefinidos read y write. Para ello se han definido una serie de constantes y variables y se han realizado llamadas a estos procedimientos con dichas variables y constantes.

Por ejemplo el código que genera un lectura como la de la línea 17 es el siguiente:

```
06    CDIRGLOBAL  1
07    LECTURAREAL
```

El código generado lo que hace es cargar la dirección de la variable y posteriormente llamar a la instrucción de lectura que en este caso es LECTURAREAL por tratarse de una variable de tipo real. Además las llamadas a read se pueden dar con variables enteras en cuyo caso es código que se genera es igual con la función LECTURA tal y como se puede ver en las líneas 4 y 5 del código generado.

Las escrituras generan el código de manera que en primer lugar calculan el valor sobre el que se va a hacer el write de la manera necesaria dependiendo de si es una expresión aritmética, un literal, una variable o una constante y posteriormente generan la función que realiza la escritura. Por ejemplo para el write de un literal como en la línea 20 del código se genera el siguiente código:

```
14    CLITERAL 18 0
15    GRABACIONLITERAL
```

La instrucción CLITERAL tiene dos parámetros enteros que se cargan en la cima de la pila y que indican el tamaño del literal y el número de literal. Posteriormente la escritura en la salida estándar la realiza el comando

GRABACIONLITERAL usando para ello los dos parámetros que encuentra en la cima de la pila.

El fichero **prueba16.pas** define procedimientos sin instrucciones con diferentes parámetros con el fin de analizar la carga de parámetros, la llamada, la reserva y la devolución de memoria de los procedimientos definidos en el código fuente.

En este programa de prueba en primer lugar se define un procedimiento “uno” sin parámetros y con variables locales. Posteriormente un procedimiento “dos” con 2 parámetros enteros, uno real y uno de tipo lógico, todos ellos por valor. A continuación está declarado el procedimiento “tres” que tiene un parámetro por referencia de un tipo matriz declarado anteriormente otro parámetro del mismo tipo por valor y dos parámetros enteros que se pasan por valor. En el cuerpo del programa principal se realizan llamadas a los tres procedimientos.

Al intentar traducir el programa se observa que no se generan errores aunque las variables locales de los procedimientos tienen el mismo nombre que las variables globales lo que muestra indicios de que la redefinición de identificadores en los diferentes niveles de anidamiento funciona correctamente.

En el código que se genera se comprueba que los procedimientos al menos han sido correctamente reconocidos ya que se han generado las etiquetas de

subprograma *1, *2 y *3 además de la que se generaba en los programas anteriores *0 correspondiente al programa principal:

```
00    *1
01    RESERVAESP 3
02    DEVOLUCION 0
03    *2
04    RESERVAESP 13
05    DEVOLUCION 5
06    *3
07    RESERVAESP 10
08    DEVOLUCION 13
09    *0
```

Si se analiza el código podemos ver que en el procedimiento uno traducido mediante la etiqueta *1 se reservan 3 direcciones de memoria que corresponden con el tamaño de las variables locales (una entera y una real). La instrucción DEVOLUCION que siempre es la última instrucción que se ejecutará en un subprograma devuelve el control al subprograma que realizó la llamada y se encarga de vaciar el espacio de memoria que ocupan los parámetros. En este caso al no haber parámetros el parámetro que recibe es 0. El segundo procedimiento traducido como *2 reserva 13 direcciones de memoria que se corresponde con las variables locales (matriz de 10 enteros, variable entera y variable real). En la instrucción DEVOLUCION se genera como parámetro un 5 que se corresponden con el tamaño de los parámetros. En el procedimiento tres se reserva mediante RESERVAESP el tamaño de la matriz definida localmente. La instrucción DEVOLUCION tiene como parámetro generado un 13, ya que el primer parámetro aunque se trata de una matriz es pasada por referencia lo que supone que sólo se

pasa la dirección del mismo y cómo las direcciones ocupan una dirección de memoria el código generado es correcto.

Si se analiza el código generado por el cuerpo del programa principal y en concreto por la línea 57 del código fuente se puede ver el código que genera una llamada a un procedimiento con parámetros por valor:

```
44    CDIRGLOBAL 0
45    CINDVALOR 1
46    CCONSTANTE 4
47    CDIRGLOBAL 2
48    CINDVALOR 2
49    CCONSTANTE 1
50    LLAMADA 2
```

Al tratarse de parámetros por valor estos pueden ser cualquier tipo de expresión que coincida en tipo con el del parámetro. El código generado realiza la carga de los valores de las expresiones que se usarán como parámetros mediante las instrucciones necesarias (carga de dirección y extracción de valor en variables, carga de constantes, o resolución de expresiones aritméticas o lógicas en su caso) para por último y cuando se encuentran todos los parámetros cargados en memoria se realiza la llamada al procedimiento mediante la instrucción LLAMADA con el número de subprograma al que se desea ceder el control como parámetro, en este caso 2.

Es interesante también analizar el código generado por la llamada de la línea 61 para comprobar la diferencia de la carga de una llamada con parámetros por referencia:

```
72    CDIRGLOBAL 4
73    CDIRGLOBAL 14
74    CINDVALOR 10
75    CCONSTANTE 27
76    CCONSTANTE 3
77    CCONSTANTE 1
78    SUMA
79    LIMITACION 1 10 0
80    CDIRGLOBAL 14
81    DELEMENTO 1 1
82    CINDVALOR 1
83    LLAMADA 3
```

Se comprueba cómo en este caso, línea 72 del código generado el parámetro por referencia sólo genera la carga de la dirección de la variable (sin extracción de valor) y el resto es igual que en el caso anterior con la generación de las operaciones necesarias para calcular los valores de los parámetros.

El fichero **prueba17.pas** es un programa con un único procedimiento que contiene una serie de instrucciones. Es interesante estudiar en el código generado a partir de este fichero como se generan las cargas de los valores de las variables tanto globales como locales. Se comprueba que el acceso a las variables globales se genera de la misma manera que en el cuerpo del programa principal mientras que las variables locales se carga mediante el uso de la instrucción destinada a cargas de direcciones locales CDIRLOCAL. Además se comprueba como las

direcciones locales son relativas a la base local ya que pueden ser positivas (parámetros) y negativas (variables locales). Por ejemplo el código generado por la línea 25 del código fuente es:

```
07    CCONSTANTE 4
08    CCONSTANTE 6
09    LIMITACION 1 10 0
10    CDIRLOCAL 4
11    DELEMENTO 1 1
12    CINDVALOR 1
13    CDIRGLOBAL 1
14    CINDVALOR 1
15    CDIRLOCAL 3
16    CINDVALOR 1
17    CINDVALOR 1
18    PRODUCTO
19    SUMA
20    LIMITACION 1 10 1
21    CDIRLOCAL -10
22    DELEMENTO 1 1
23    AINDVALOR 1
```

En las líneas 8 a 12 se realiza la carga del elemento 6 de la matriz local que se pasa como parámetro dónde se comprueba que la dirección se realiza mediante CDIRLOCAL como un parámetro positivo.

En las líneas 13 y 14 se realiza la carga de la variable i que al ser local se realiza con la instrucción CDIRGLOBAL.

En las líneas 7, 20, 21 y 22 se realiza la carga de la dirección del elemento 4 de la matriz tablaAux definida localmente por lo que se genera la instrucción CDIRLOCAL y en este caso al no tratarse de un parámetro se genera como parámetro de la instrucción una constante negativa.

También en este programa como el procedimiento uno tiene declarada una llamada así mismo, llamada recursiva, podemos observar como dentro del código generado, por el procedimiento se encuentra la instrucción LLAMADA con el número 1 como parámetro que se corresponde con la etiqueta de subprograma del propio procedimiento.

Respecto al subconjunto de Pascal implementado y en lo referente a subprogramas queda comprobar el anidamiento de los mismos. Esto se realiza en el programa **prueba18.pas**. Este programa tiene tres procedimientos anidados (declarados uno dentro de otro) y con llamadas desde el programa principal hacia el procedimiento, uno, en el cual hay una llamada al procedimiento declarado en éste, dos, que a su vez tiene una llamada al procedimiento más interno, tres.

El fichero **prueba19.pas** es un fichero con errores sintácticos y semánticos comentados en lo que a la declaración y llamada de procedimientos se refiere. Al igual que en los anteriores programas de este tipo se puede descomentar cada error para comprobar cómo el traductor reconoce y avisa del error.

Los ficheros **prueba20.pas** y **prueba21.pas** son programas completos con la funcionalidad de una calculadora (son idénticos excepto que el primero usa operadores enteros y el segundo operadores reales), que hacen uso de bastantes partes del lenguaje y cuya utilidad sería más la de comprobar el funcionamiento del

intérprete que se expone como ampliación del proyecto en el capítulo de *Conclusiones y Ampliaciones*. Por esta razón aquí simplemente cabe mencionar que el análisis y la traducción del mismo se realiza aparentemente de manera correcta.

10. Conclusiones y ampliaciones

Tras la implementación del traductor y la realización de todas las pruebas y el desarrollo de la presente documentación se puede realizar una valoración de los objetivos propuestos así como las posibilidades de ampliación del proyecto.

10.1. Conclusiones referentes a la implementación con lenguaje orientado a objetos

Uno de los objetivos principales del proyecto era comprobar la posibilidad de implementar con teoría orientada a objetos de un traductor. En este sentido queda claro que es totalmente viable y que las herramientas existentes permiten realizar este tipo de desarrollos de manera ágil simplificando su desarrollo.

El código necesario para la implementación, tanto en lo referente a las definiciones JFlex y Java CUP como al resto del software Java:

- el uso de herencia permite sacar mucho partido en partes de la implementación y reutilizar código como ha quedado demostrado por ejemplo en la implementación de los tipos de objetos que usa la tabla de símbolos

- la generación de las especificaciones JFlex y Java CUP es relativamente amigable destacando como parte mal que al tener que embeber bastante código en la misma si éste no se realiza bien puede complicar su mantenimiento, por eso, es totalmente necesario usar clases auxiliares externas y embeber código relativamente sencillo que haga llamadas a dichas clases

- el uso de un lenguaje Java con soporte nativo para realizar interfaces gráficos permite integrar muy bien un interfaz amigable y que permita el uso de la herramienta de manera sencilla

Por tanto, y con las limitaciones explicadas a lo largo de este documento se comprueba la total viabilidad para implementar este tipo de software con lenguajes orientados a objetos y concretamente con Java.

10.2. Conclusiones referentes a la finalidad didáctica

Dado que este PFC surge a partir de la extinta asignatura *Compiladores e Intérpretes* de la Ingeniería Técnica de Informática de Sistemas es de interés dotarle de un carácter didáctico desde el principio.

Para completar este fin:

- el desarrollo implementado está estructurado pensando en que pueda ser analizado por estudiantes de manera evolutiva permitiendo por ejemplo probar por separado el analizador léxico o sintáctico

- presenta una base para la implementación con lenguajes orientados a objetos de componentes básicos en el mundo de los traductores, compiladores e intérpretes como por ejemplo la tabla de símbolos respetando las directrices del paradigma de objetos

- todo el código fuente así como este documento estarán disponibles en Github con acceso libre para su estudio

- el interfaz gráfico y algunas funcionalidades implementadas tienen como finalidad única facilitar la realización de pruebas y comprender el funcionamiento tal y como es por ejemplo la funcionalidad de log con el debug que este genera en ciertas operaciones y la posibilidad de gestionarlo para poder analizarlo

10.3. Ampliaciones del PFC

Como se ha indicado en varios puntos de la presente documentación existe una ampliación evidente del proyecto que consiste en la implementación del intérprete y por tanto de la máquina virtual para permitir analizar y ejecutar programas escritos en JMPascal o programas JSPascal traducidos con el traductor implementado en este PFC.

Esta ampliación debería contemplar los siguientes objetivos:

- implementación de los analizadores léxico, sintáctico y semántico del lenguaje JMPascal, lo cual, puede ser realizado igualmente con las herramientas JFlex y Java CUP

- implementación de la máquina virtual que permita realizar la ejecución de los programas, si la implementación se realiza con Java CUP, el uso de la máquina virtual quedará dirigido por el analizador sintáctico al igual que en el traductor del presente PFC se ha realizado la generación de código

- integración de las opciones de ejecución en el interfaz gráfico, sin olvidar la finalidad formativa del PFC por lo que se debería incluir la posibilidad de ver y analizar el estado de los diferentes componentes de la máquina virtual

- permitir la ejecución paso a paso para la depuración de programas, incluyendo para ello las opciones correspondientes en el interfaz gráfico

Como nota, indicar que en el software implementado y aunque no es parte de este proyecto se ha realizado esta ampliación y el código fuente del mismo incluye el paquete intérprete.

Bibliografía

11. Bibliografía

“Java a tope: Compiladores.Traductores y Compiladores con Lex/Yacc, Jflex/cup y JavaCC” (2005)

Sergio Gálvez Rojas y Miguel Ángel Mora Mata.

Universidad de Málaga http://www.lcc.uma.es/~galvez/compila dores.html

“Jflex User’s Manual” (Versión 1.4.1 / Noviembre de 2004)

Gerwin Klein

http://Jflex.de/manual.html

“Cup User’s Manual” (Versión 0.10j)

Scott E. Hudson

Modificado por Frank Flannery, C. Scott Ananian, Dan Wang con la supervisión de Andrew W. Appel

http://www.cs.princeton.edu/~appel/mo dern/java/CUP/manual.html

“Implementación de un Subconjunto de Pascal”

José Gabriel Pérez Díez

Publicaciones Escuela Universitaria de Informática (UPM)

“Programming Language Processors In Java. Compilers And Interpreters” (2000)

David A. Watt y Derick F. Brown.

Editorial Prentice Hall.

“Java2 Versión 1.4” (2003)

Israel Pastrana Vicente

Editorial Anaya

“Java 2”

Herbert Schildt

Editorial Osborne-McGraw Hill

“J2SE”

Sun Microsystems

“J2SE v 1.4.2 API Specification”

Sun Microsystems

http://www.java.sun.com/j2se/1.4.2/doc s/api/index.html

Eclipse

“La web del programador”

http://www.lawebdelprogramador .com

“javaHispano. Tu lenguaje, tu comunidad.”

http://www.javahispano.org

Anexos

Anexo A. Estudio de JFlex

Jflex es una herramienta desarrollada por Gerwin Klein cuya función principal es la de generar analizadores léxicos también en Java.

Está basada en el programa JLex de Ellit Berk (Universidad de Princeton) que a su vez utilizó las ideas de los famosos programas lex y flex, escritos en C/C++, todos ellos programas generadores de analizadores léxicos a partir de un fichero de especificaciones.

El uso de esta herramienta capacita al programador de una potente herramienta para comenzar con el desarrollo de un traductor, compilador o intérprete en el paradigma de la programación orientada a objetos, ya que facilita una de las tareas más sencillas pero a su vez más pesadas como es el reconocimiento de la piezas sintácticas de un lenguaje.

La manera de trabajar con Jflex es sencilla ya que simplemente se debe definir un fichero de especificaciones y tratarlo con el programa para obtener una clase java que implementa los autómatas necesarios de manera óptima. La manera de interactuar con la clase generada es también sencilla ya que dicha clase implementa un método público llamado *next_token*, si se ejecuta con compatibilidad Java CUP, el cual como su nombre indica devuelve la siguiente pieza sintáctica encontrada en el fichero que se esté analizando.

A.1. Fichero de Especificación Jflex

El fichero de especificación está formado por varias secciones que se detallan a continuación:

```
declaraciones java
%%
declaraciones Jflex
%{
código Java
%}
declaración de macros
%%
reglas y acciones
```

A.1.1. Sección 1: Declaraciones Java

La primera sección sirve para especificar código Java que se debe generar antes de cualquier otro código. Se utiliza para especificar el paquete al que pertenece la clase generada así como para declarar los import necesarios en el código Java que se escribirá en la última sección.

A.1.2. Sección 2: Declaraciones Jflex

La segunda sección está destinada a definir las declaraciones propias de Jflex que definen su manera de funcionar. En la tabla que se muestra a continuación se puede ver una descripción de las mismas. Cada directiva debe declararse en una línea nueva y además debe especificarse en la primera columna de la línea.

DIRECTIVA	DESCRIPCIÓN
%class MiClase	Esta directiva permite definir el nombre que tendrá la clase generada por Jflex, así como el nombre del fichero generado.
%public	Con esta directiva se consigue que la clase generada sea pública. Algo necesario si se quiere interactuar con otras clases o paquetes.
%abstract	Genera una clase abstracta.
%final	Genera una clase final.
%apiprivate	Hace que todos los métodos y propiedades del scanner generador sean privados exceptuando el constructor, el código especificado por el usuario y si está definida la compatibilidad con CUP el método next_token.

%implements interfaces	Permite especificar una lista de interfaces que implementará la clase generada.
%extends clase	Esta que el scanner generador herede de otra clase.
%init{ .... %init}	El código que se escribe entre estas dos directivas será copiado íntegramente al constructor de la clase generada.
%initthrow excepciones	Permite especificar una lista de clases de excepción que debe elevar el constructor en caso de producirse.
%scanerror excepciones	Permite definir la clase de excepción que se debe elevar cuando se produzca un error interno en el scanner. Por defecto se lanza java.lang.Error. Esta excepción no será lanzada nunca en condiciones normales.
%buffer tamaño	Indica el tamaño inicial del buffer dedicado al scanner. Por defecto vale 16384.
%include fichero	Permite realizar la inclusión de un fichero reemplazando esta directiva por el texto que contenga dicho fichero. Esta directiva está en versión experimental por lo que puede dar errores inesperados.

%function name	Indica el nombre que tendrá el método escaneador. Si no se define esta directiva dicho método es *yylex*. Si se usa la directiva %cup el nombre por defecto de este método es next_token por lo que si se sobreescribe la clase generada será abstracta obligatoriamente y se debe realizar una pseudo implementación de next_token posteriormente.
%int %integer	Hace que el método yylex (o su equivalente si definimos %function) devuelve valores de tipo int. En este caso el final de fichero estará identificado por la declaración de la YYEOF que es una variable de la clase generada declarada como public static final int.
%intwrap	En este caso el método yylex devolverá objetos de la clase Integer. El final de fichero se identifica entonces como null.
%type nombre	Mediante el uso de esta directivo se puede especificar el tipo o clase que devolverá el método yylex o su equivalente mediante esta directiva. En este caso el valor para identificar el final de fichero será null si el tipo hereda de java.lang.Object, en otros casos se debe definir el valor que identifique el final de fichero mediante las directivas %eofval{ ... %eofval}.

	El uso de las directivas %cup y %type son incompatibles teniendo preferencia la segunda.
%yylexthrow excepciones	Permite definir las excepciones que deberá elevar el método yylex.
%debug	Genera una clase main que espera como parámetro un nombre de fichero. Esta clase ejecuta el scanner sobre el fichero y muestra un listado para cada token encontrado con la línea (si está definido %line), columna (si está definido %column), el texto, y la acción realizada (indicando para ello el número de línea en la especificación).
%standalone	Crea un método main que espera como parámetro un nombre de fichero y muestra por pantalla todas las partes de texto que no es capaz de reconocer.
%7bit	Indica que el scanner generado sólo admitirá los caracteres que se definen con 7 bits, es decir, los correspondientes a los caracteres 0 a 127. Si se detecta en la entrada un carácter por encima del 127 el scanner lanzará la excepciónArrayIndexOutOfBoundsException.
%full %8bit	Ambas directivas hacen que el juego de caracteres sea de 8 bits. Si se detecta en la entrada un carácter por encima del 255 el scanner lanzará la excepción ArrayIndexOutOfBoundsException.

%unicode %16bit	Indica que el juego de caracteres de los ficheros de entrada al analizador generado será el juego completo de caracteres Unicode.
%ignorecase %caseless	Esta directiva especifica que el lenguaje fuente no distingue entre mayúsculas y minúsculas. De esta forma se simplifica mucho la especificación de las expresiones regulares que definen las piezas sintácticas.
%cupsym Sym	Indica el nombre de la clase que define las constantes de cada una de la piezas sintácticas. Esta clase será generada por Java CUP de ahí su nombre. Esta directiva debe ser declarada siempre antes de %cup.

%cup	Hace que el analizador generado sea compatible con Java CUP, para ello implementa la interfaz que Java CUP necesita en el analizador léxico. Declarar esta directiva es equivalente a realizar las siguientes declaraciones:
	`%implements java_cup.runtime.Scanner`
	`%function next_token`
	`%type java_cup.runtime.Symbol`
	`%eofval{`
	`return new java_cup.runtime.Symbol(<CUPSYM>.EOF);`
	`%eofval}`
	`%eofclose`

%cupdebug	Dota de una función main al analizador generado que recibe como parámetro un nombre de fichero y ejecuta el scanner con ese fichero como entrada e imprime una lista con la línea, columna, texto englobado y el nombre del símbolo CUP para token encontrado.

%byacc	Hace que el scanner generador sea compatible con baycc (otro generador de analizadores sintácticos en Java). Es equivalente a la declaración de las directivas:
	`%integer`
	`%eofval{`
	`return 0;`
	`%eofval}`
	`%eofclose`

%switch	Genera un analizador basado en el autómata finito determinado se implementa mediante una sentencia switch. El analizador generado de esta manera está bastante comprimido pero no se recomienda para autómatas con más de 200 estados.
%table	Con esta directiva el autómata se dirige por una tabla implementada mediante un array. Esta manera de implementar el autómata es la que menos comprime y por lo tanto no se recomienda su uso

%pack	Hace que la implementación del autómata se comprima como uno o más literales. Esta opción es la más interesante para analizadores con muchos estados. En tiempo de ejecución funciona igual que con la directiva %table ya que lo que hace es descomprimir los literales creados en tablas.
%line	Activa la variable que almacena el número de línea, *yyline*, que se está analizando del fichero de entrada al analizador sea pública. Esta variable se inicializa a 0.
%column	Activa la variable que almacena el número de columna, *yycolumn*, que se está analizando en la línea actual del fichero de entrada al analizador sea pública. La primera columna será considerada como 0.
%char	Activa la variable que almacena el número de caracteres leídos desde el principio del fichero de entrada. La cuenta comienza en 0.

Tabla A.1.2.1-1. Directivas Jflex

A.1.3. Sección 3: Código Java

Respecto a la tercera sección se debe indicar inicialmente que realmente no es un sección sino que es otra directiva de Jflex especificada por %{ y %}. Esta

directiva o sección permite especificar código java que será copiado íntegramente a la clase generada. Se utiliza para definir métodos y variables que se necesiten en la posterior implementación de las acciones.

A.1.4. Sección 4: Definición de macros

La cuarta sección permite crear macros que luego serán utilizadas en la definición de reglas. Si por ejemplo se utiliza muchas veces una expresión regular podemos definirla como macro y luego utilizar el nombre de la macro. Una macro se crea de la siguiente forma:

```
identificador = expresion_regular
```

El identificador debe estar formado por una letra seguida de más letras, números o guiones bajos. El espacio entre el identificador y el igual es opcional así como el espacio entre el símbolo '=' y la expresión regular. La expresión regular se define siguiendo las mismas normas que se siguen para definir las reglas con la excepción de que no se pueden utilizar los operadores ^, $ y /.

A.1.5. Sección 5: Reglas y acciones

En esta sección se definen las reglas que identifican las piezas sintácticas del lenguaje fuente a implementar. Estas reglas son en realidad expresiones regulares.

Cada regla puede tener asociada una acción a realizar. Las acciones se implementan como código Java que será copiado íntegramente a la clase generada. Cada vez que el analizador asocie parte del texto de entrada con una pieza sintáctica se ejecutará el código definido en la acción.

La sintaxis de una regla con su acción es así:

```
regla(ER) {accion(código JAVA)}
```

La definición de reglas debe seguir la sintaxis y normas de Jflex. Cuando se genera la clase final el programa Jflex analiza la sintaxis propia del fichero de definición pero no la del código Java.

A.1.5.1. Definición de reglas

Para definir las expresiones regulares que determinan las reglas se utilizan las convenciones siguientes:

- Un carácter se representa a sí mismo, excepto si es un metacaracter. Por ejemplo el carácter 'a' representa al patrón de entrada a.
- Los metacaracteres pierden su valor especial en la siguientes casos:
    - Cuando van entre comillas, excepto el carácter '\'.
    - Cuando están dentro de la definición de una clase, excepto '^' cuando va al comienzo de la clase y '\' siempre.
    - Cuando van precedidos del meta carácter '\'.
- Se pueden definir clases de caracteres. Una clase de caracteres será patrón válido para una entrada si cualquier carácter de la clase es el de la entrada. La sintaxis para definir una clase es '[{Caracter|Caracter'-'Caracter}*]'. De esta manera podemos englobar caracteres sueltos o rangos. Por ejemplo una definición de clase válida sería [ac5-8D] que sería validado para cualquier entrada de los caracteres 'a', 'c', '5', '6', '7', '8' y 'D'.
- En las clases se puede utilizar el operador ^ al principio de la clase para negar la clase. De esta manera si se define [^ac5-8D] se creará un patrón que será válido para cualquier carácter de entrada excepto los caracteres 'a', 'c', '5', '6', '7', '8' y 'D'.
- Se puede utilizar la expresión regular definida anteriormente en una macro mediante la sintaxis '{'identificador'}' siendo identificador el nombre que asignamos a la macro anteriormente.
- Existen una serie de clases predefinidas en Jflex. La primera de ellas está disponible siempre, el resto sólo están disponibles si utilizamos la directiva

Jflex %unicode y representan el equivalente a ejecutar el método indicado de la clase java.lang.Character:

  - **.**: Contiene cualquier carácter excepto el carácter de salto de línea, '\n'.
  - **[:jletter:]**: isJavaIdentifierStart()
  - **[:jletterdigit:]:** isJavaIdentifierPart()
  - **[:letter:]**: isLetter()
  - **[:digit:]**: isDigit()
  - **[:uppercase:]**: isUpperCase()
  - **[:lowercase:]**: isLowerCase()

- Además existen operadores para las expresiones regulares. Se describen a continuación con la ayuda de dos expresiones regulares, *a* y *b*:
  - **Union (*a*|*b*)**: Se representa por el metacaracter '|' y equivale a la **operación lógica O**. Es decir si la expresión es *a* | *b* cualquier entrada será válida para esta expresión regular si cumple la expresión regular de *a* o si cumple la expresión regular de *b*.
  - **Concatenación (*ab*)**: Une dos expresiones regulares. *ab* equivale a una entrada que cumple la expresión regular de *a* seguida de un patrón que cumple la expresión regular de *b*.
  - **Operación * (*a**)**: Implica que la expresión a la que se aplica dicho operador se ha de dar **0 o más veces**. Por ejemplo *a** representa entradas que se identifican con la expresión regular *a* concatenadas 0 o más veces.

- **Operación + (*a*+)**: Implica que la expresión a la que se aplica dicho operador se ha de dar **1 o más veces**. Por ejemplo *a** representa entradas que se identifican con la expresión regular *a* concatenadas 1 o más veces. Equivale a la expresión regular *aa**.
- **Opcional (*a*?)**: Se representa mediante el metacaracter '?'. *a*? define una expresión regular que será válida para la entrada vacía o para una entrada que cumpla la expresión regular *a*.
- **Negación (!*a*)**: Para utilizarlo se hace uso del metacaracter '!'. La expresión !*a* representa cualquier patrón que no coincida con la expresión regular *a*.
- **Hasta (~*a*)**: Se representa mediante el metacaracter '~'. Esta expresión recoge toda la entrada que se encuentre hasta que se cumpla la expresión regular que viene después. Por ejemplo *a*~*b* representa cualquier entrada que empiece por el patrón de la expresión regular *a* y que acabe por el patrón de la expresión regular *b* pudiendo haber entre medias cualquier cosa. Esta operación es muy útil para definir comentarios, por ejemplo los comentarios multilínea de C se definirían como: “/*” ~ “*/”
- **Repetición (*a*{n})**: Se representa con los caracteres {n} donde n es un número que representa las veces que ha de repetirse la expresión regular. Por ejemplo *a*{3} equivale a la expresión regular *aaa*
- **Repetición Especial (*a*{n,m})**: Se representa con los caracteres {n,m} donde n y m son números que representan el mínimo y máximo de

veces que ha de repetirse la expresión regular. Por ejemplo *a*{3,5} equivale a la expresión regular *aaa*|*aaaa*|*aaaaa*

- **Comienzo Línea (^*a*)**: Indica una expresión regular que se debe dar al comienzo de la línea.
- **Fin Línea (*a*$)**: Indica una expresión regular que se debe dar al final de la línea.

- También existen una serie de caracteres especiales que se representan con la ayuda del meta carácter '\':
  - Tabulador: '\t'
  - Fin línea: '\r'
  - Salto de línea: '\n'
  - Espacio: '\ '
- Se debe tener en cuenta que el analizador generado siempre cogerá la mayor cantidad de texto de la entrada, es decir, siempre entrará por la regla que más caracteres utilice. Si en algún momento hay varias expresiones que cumplen la entrada con el mismo número de caracteres se utilizará la que primero se haya definido. Con esto se deduce que el orden en el que se definen las reglas es importante.

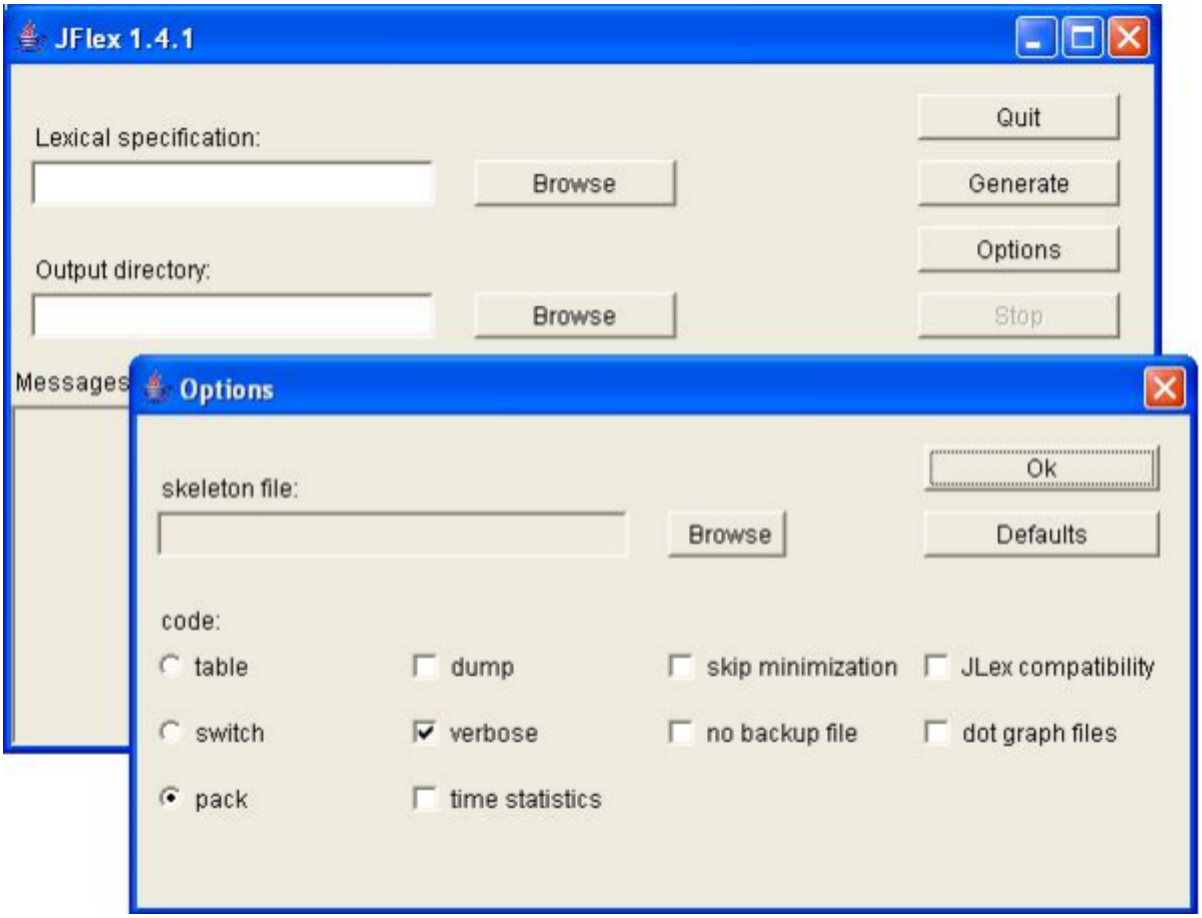
A.2. Uso y apariencia de Jflex

Para utilizar el programa Jflex necesitamos únicamente tener instalada la máquina virtual de Java. Al estar implementado en java puede utilizarse en plataformas Linux/Unix y en plataformas Windows.

Para utilizarlo simplemente debemos descargar el paquete comprimido disponible en formato zip y tar en la url http://Jflex.de/download.html y descomprimirlo. Esto crea cuatro directorios:

- bin: Contiene dos scripts para arrancar el programa de manera sencilla. Uno, Jflex, es un script de la shell bash para arrancar el programa en plataformas Linux/Unix. El otro, Jflex.bat, es un programa batch de MS-DOS que arranca el programa en entornos Windows. Este script deben ser modificados para especificar los directorios del sistema en el que se ejecuta el programa.
- doc: Contiene el manual oficial de Jflex en formato html y ps.
- examples: Varios ejemplos completos.
- lib: Contiene un jar con todo el entorno necesario para realizar desarrollos con las clases generadas por Jflex; este jar debe ser incluido en el classpath de las aplicaciones que utilicen la clase generada.
- src: Contiene los fuentes de Jflex así como la parte de los fuentes del entorno de Java CUP que necesita para poder implementar la compatibilidad con dicho programa.

Para utilizar el programa simplemente se debe arrancar el programa mediante el script correspondiente del directorio bin con lo que se lanzará un interfaz como el siguiente:



donde se debe indicar el fichero de entrada (fichero de especificación Jflex), el de salida (clase generada) y darle al botón generar, *Generate*. Antes de generar se puede definir una serie de opciones mediante el botón *Options* como realizar la generación de modo verboso para ver lo que se está realizando mientras se genera la clase del scanner, activar tiempos de estadísticas, indicar el tipo de scanner a generar o hacer que se generen también unos ficheros de definición de gráficos dot

basados en el proyecto Graphviz, http://www.graphviz.org, que representan los autómatas que implementan el scanner generado. Esta funcionalidad todavía no está muy conseguida ya que en cuanto Jflex crea un autómata complejo el gráfico que genera el software Graphviz no es visible.

B. Estudio de Java CUP

Java CUP es un generador de parsers LALR para Java desarrollado en Java por Scott E. Hudson del instituto de tecnología de Georgia y posteriormente modificado por Frank Flannery, C. Scott Ananian y Dan Wang.

Esta herramienta está basada en los famosos programas yacc y bison ambos generadores de parsers en lenguaje C desarrollados por y para los entornos Unix y Linux respectivamente.

Con esta herramienta se facilita en gran medida la realización de los análisis sintácticos y semánticos que se deseen realizar en un programa Java. Para ello simplemente se debe definir un fichero de especificación Java CUP y el programa generará la clase que implementa el parser.

Java CUP define un entorno de trabajo, runtime, que debe utilizar el resto de clases o programas que deseen trabajar en concordancia con él. Este runtime define:

- Una **interfaz java, Scanner.java**, que especifica lo que debe implementar un scanner para ser compatible con Java CUP.
- La **clase Symbol.java** que implementa un clase para instanciar como objetos las piezas sintácticas que se pasan desde el scanner y que permite

almacenar los datos del token que pueden ser necesarios en el futuro análisis, como la línea, columna, valor, etc.

Java CUP es totalmente compatible con el scanner que genera Jflex (siempre y cuando se activan las opciones necesarias en el fichero de especificación Jflex). Para que funcionen juntos se debe instanciar un scanner de la clase generada por Jflex que se pasa como parámetro al instanciar la clase generada por Java CUP. Para comenzar el análisis se llama al método parse de la clase generada por Java CUP el cual realiza todo el análisis, llamando para ello a medida que lo necesita al método next_token del scanner para solicitar la lectura de la siguiente pieza sintáctica de la entrada.

Sin embargo debe quedar constancia de que no es obligatorio el uso de Jflex para utilizar Java CUP, ya que se podría utilizar cualquier programa que implemente el interfaz del runtime de Java CUP.

B.1. Fichero de especificación Java CUP

Los ficheros de especificación de Java CUP al igual que los de Jflex se dividen en secciones como podemos ver a continuación:

```
declaraciones java
código de usuario
listas de símbolos
declaraciones de precedencia y asociación
gramática (producciones y acciones)
```

B.1.1. Sección 1: Declaraciones Java

En esta sección se pueden realizar declaraciones Java que deben ir al principio del todo en el fichero Java que generará Java CUP. Se utiliza para declarar el paquete al que pertenece la clase generada así como los import de Java de las clases que posteriormente necesitaremos en el código de las acciones o de la segunda sección.

B.1.2. Sección 2: Código de usuario

Esta sección permite declarar código Java que se incluirá en diferentes puntos del fichero Java generado. Para entender dónde se colocarán estos código debemos saber que Java CUP en el fichero generado define dos clases:

- una clase pública, **Parser**, que hereda de la clase lr_parser del runtime de Java CUP que es la que implementa el Parser y

- una clase privada, **CUP$parser$actions**, que se encuentra embebida en la clase Parser que sirve para almacenar todo el código de las acciones definidas por el usuario en la sección de reglas y acciones.

Ambas clases se conectan mediante la declaración de un objeto de la clase CUP$parser$actions en la clase Parser. Este objeto siempre es identificado como **action_obj.**

Sabiendo esto podemos decir que se puede definir código de usuario en hasta 4 zonas diferentes del fichero generado. Cada uno de estos códigos se define utilizando unas etiquetas reservadas de Java CUP. Todas estas etiquetas son opcionales, es decir, se pueden definir o no sin seguir ningún orden en concreto pero siempre delante la lista de símbolos.

La primera de las etiquetas se define como:

```
action code {:
      <codigo_Java>
:};
```

dónde <codigo_Java> es el código del usuario en Java que será transcrito íntegramente a la clase CUP$parser$actions. Podemos definir en él métodos y variables que se utilizan en las acciones. Se utiliza comúnmente para declarar métodos y variables relacionados con la tabla de símbolos y su manejo.

La siguiente etiqueta permite definir código que será copiado íntegramente en la clase Parser, para ello se utiliza las siguiente sintaxis:

```
parser code {:
      <codigo_Java>
:};
```

Este normalmente se utiliza para definir métodos y variables que deben ser accesibles por el restos de clases que haya en nuestro desarrollo, por ejemplo se utiliza para redefinir el constructor de la clase y poder así pasarle parámetros a la clase generada.

La siguiente declaración permite definir código que se ejecutará justo antes de que el parser solicite el primer token al scanner. Su uso suele ser el de inicializar estructuras antes de comenzar con la ejecución de las acciones. La sintaxis para definirlo es la siguiente:

```
init with {:
      <codigo_Java>
:};
```

Por último se puede definir la siguiente declaración:

```
scan with {:
      <codigo_Java>
:};
```

Esta parte del código es copiada a un método concreto de la clase Parser que sólo se genera si existe la declaración scan with en el fichero Java CUP. Este método debe definir cómo se accede al próximo token y debe por tanto devolver un objeto de la clase Symbol para que sea compatible con el parser generado como ya se explicó en el apartado 3. Esta declaración nunca será necesaria si se utiliza Java CUP junto a Jflex (u otro generador de analizadores sintácticos como Jlex) que ya implementan un scanner completo devolviendo objetos de la clase Symbol del runtime de Java CUP.

B.1.3. Sección 3: Lista de símbolos

Esta es la primera sección obligatoria de los ficheros de especificación Java CUP. Aquí se deben definir todas las piezas sintácticas, terminales y no terminales, que se utilicen en la gramática posterior.

Como ya se ha comentado anteriormente las piezas sintácticas se representan mediante objetos de la clase Symbol del runtime. En los terminales este objeto lo devuelve el scanner con la lectura de la entrada. El objeto de la clase Symbol de los no terminales se genera automáticamente por el parser sustituyendo uno o varios objetos de la pila que coinciden con la parte derecha de una producción

de la gramática del parser por el correspondiente no terminal, es decir, cuando se produce una reducción.

Tanto los terminales como los no terminales pueden tener un tipo, es decir, pueden almacenar un valor (variable value de la clase Symbol) y este será de una clase concreta. Los terminales y los no terminales se deben definir por separado y también deben separarse los que sean de diferentes tipos siguiendo la siguiente gramática, dónde suponemos que pID es un identificador típico de C, Java o similares y pIDClaseJava es un identificador de una clase java que puede contener jerarquía de paquetes.

```
<listaTerminales> ::= <listaTerminales> <definicion>
                        | <definicion>
<definicion> ::= <tipoDeclaracion> <nombreClase> <listaID> ;
<tipoDeclaracion> ::= terminal
                     ::= non terminal
<nombreClase>     ::= pIDClaseJava
                        | ε
<listaID>         ::= <listaID> , pID
                        | pID
```

De una manera más amigable podemos decir que la declaración se hace como sigue:

```
terminal clase1 terminal1, terminal2, terminal3, .... ;
non terminal paquete1.clase2 non_terminal1, non_terminal2, ... ;
terminal terminal1, terminal2, ... ;
terminal paquete2.clase3 terminal1, terminal2, terminal2, ... ;
non terminal terminal1, terminal2, ... ;
```

Cabe destacar como nota que desde la versión 0.10j de Java CUP también se pueden utilizar nonterminal en vez de non terminal.

Los nombres de los terminales y los no terminales como es de suponer no pueden coincidir con las palabra reservadas de Java CUP: "**code**", "**action**", "**parser**", "**terminal**", "**non**", "**nonterminal**", "**init**", "**scan**", "**with**", "**start**", "**precedence**", "**left**", "**right**", "**nonassoc**", "**import**" y "**package**".

B.1.4 Sección 4: Declaraciones de precedencia y asociación

Esta sección también es opcional y permite definir la prioridad y la asociatividad de la gramática. Esto es útil sobre todo para gramáticas ambiguas.

Hay tres tipos de declaraciones de precedencia que se definen como se muestra a continuación:

```
<precedencia>     ::= precendence <tipo> listaTerminales
<tipo>      ::= left
                        | right
                        | nonassoc
<listaTerminales> ::= <listaTerminales> , pID
                        | pID
```

dónde pID es un identificador de terminal de los definidos en la sección anterior.

Todos los terminales definidos en la misma declaración precendece tienen la misma asociatividad. Además los terminales tendrán una prioridad equivalente a la prioridad en que se ha declarado, teniendo mayor prioridad la última declaración, es decir, si tenemos la siguiente declaración:

```
precedence left SUMA, RESTA;
precedence left PRODUCTO, DIVISION;
```

querrá decir que PRODUCTO y DIVISIÓN tienen mayor prioridad que SUMA y RESTA. Además todos ellos tienen asociatividad por la izquierda.

Los terminales que no se encuentren en ninguna declaración de precedencia tienen la menor prioridad posible. Además cabe destacar que Java CUP asigna a cada producción una precedencia que es equivalente a la precedencia del último terminal. Si una producción no tiene terminales ésta tendrá la menor precedencia. De esta manera se solventan muchos conflictos shit-reduce en los que las producciones que forman el conflicto tienen distinta precedencia.

La asociatividad puede ser de tres tipos: **left, right o nonassoc.** Como sus nombres en inglés indican definen asociatividad por la izquierda, por la derecha o sin asociatividad. Con esta característica podemos también resolver conflictos shift-reduce provocados por producciones de igual precedencia. Debemos saber que si definimos precedencias sin asociatividad y encontramos en la entrada dos ocurrencias consecutivas de un terminal con la misma precedencia y sin

asociatividad el parser generará un error, es decir, si tenemos como entrada 3 = 4 = 5 = 6 y hemos declarado el operador =, suponemos que su terminal se representa como pIgual, como precendence nonassoc pIGUAL el parser generará un error sintáctico.

B.1.5. Sección 5: Gramática (producciones y acciones)

Esta es la sección más importante ya que define la gramática del analizador sintáctico así como las acciones que se deben realizar cada vez que se realiza una reducción.

La gramática opcionalmente puede comenzar por la declaración start:

```
start with non-terminal
```

Esta declaración permite indicar por qué producción comienza la gramática. Si ésta no es definida la gramática comenzará por la primera producción definida.

A continuación se define la gramática propiamente dicha. La gramática está formada por producciones, que se definen mediante un no terminal que se pone en la parte izquierda seguido del símbolo '**::=**' y seguido por una o más opciones separadas por el símbolo '|' y que acaba por el símbolo '**;**' . Cada una de estas

opciones está formada por una serie de cero o más acciones, símbolos terminales y símbolos no terminales. Las acciones se puede definir por tanto al principio, en medio al final de la producción, para ello se utilizan las etiquetas '**{:**' y '**:}**' que indican el principio y el fin de la acción respectivamente. Entre ambas etiquetas se pone código Java que se copia íntegramente a la clase generada y que se ejecutará cuando la porción de la derecha de la acción de la producción haya sido reconocida.

Además al lado de cada símbolo terminal o no terminal se puede definir una etiqueta precedida del símbolo '**:**'. Esta etiqueta sirve para poder utilizar el objeto Symbol asociado al terminal/no terminal, con el fin de poder obtener el número de línea, columna o valor en el que se ha detectado el terminal. Para ello automáticamente en el código generado se crea un objeto, que será de la clase definida a dicho terminal/no terminal en la lista de símbolos, con el nombre de la etiqueta que contiene el objeto de la propiedad value de la clase Symbol si es un terminal. Si es un no terminal contendrá el objeto que devolvió la producción que generó el no terminal mediante una reducción en la gramática. Además por cada etiqueta se crean otras dos variables accesibles en los códigos de las acciones sucesivas de la producción, éstas se identifican por el nombre de la etiqueta concatenada con “left” y “right”. La primera de ellas representa el número de fila en la que se ha detectado el terminal/no terminal y la segunda el número de columna.

B.2. Uso y apariencia de Java CUP

Java CUP es una herramienta java que funcione en consola mediante parámetros. Al estar implementada con Java es multiplataforma.

Para generar un analizador sintáctico a partir de un fichero Java CUP debemos de lanzar el programa Java CUP, el cual está empaquetado como un jar, y hacer uso de los parámetros que se detallan en la siguiente tabla:

PARÁMETRO	**ACCIÓN**
-package nombre	Permite indicar el nombre del paquete al que pertenecerá la clase generada.
-parser nombre	Permite indicar el nombre de la clase que se va a generar y por lo tanto el del fichero de salida. Por defecto la clase se llama parser.
-symbols nombre	Hace que además de generarse la clase del parser se genere un segunda clase con constantes numéricas que representan a cada uno de los terminales. Por defecto esta clase se llama sym.

-interface	Con esta opción conseguimos que la el fichero generado para representar los símbolos sea un interfaz en vez de una clase.
-nonterms	Con esta opción conseguimos que en la clase sym generada se generen además identificadores para representar los no terminales.
-expect number	Java CUP aborta su ejecución al detectar una gramática ambigua. Esta opción permite definir cuantas ambigüedades esperamos que tenga nuestra gramática. Java CUP no abortará siempre y cuando el número de ambigüedades sea menor que el número esperado. Esto puede ser útil para gramáticas cuya ambigüedad se soluciona con una opción por defecto como sucede con el conocido caso de la sentencias if – else.
-compact_red	Si usamos esta opción conseguimos que se realice una compactación de las tablas de reducción, sin embargo, es muy posible que se detecten errores en el parser generado si usamos esta opción.
-nowarm	Con esto conseguimos que Java CUP no nos muestre los warning que pueda generar al analizar una gramática.

-nosummary	Cuando Java CUP termina de realizar una generación muestra un resumen en pantalla con información como el número de terminales, no terminales, producciones, etc. Con esta opción se consigue que no se muestre dicho resumen.
-progress	Si indicamos esta opción en la línea de comandos obtendremos en pantalla indicaciones de lo que está haciendo Java CUP en la generación.
-dump_grammar *-dump_states* *-dum_tables* *-dump*	Estas opciones hacen que se genere fichero comprensibles que contienen la gramática, los estados del parser y las tablas de reducción. La opción -dump hace que se generen todos.
-time	Si activamos esta opción obtendremos además un resumen con las estadísticas de los tiempos invertidos en cada fase que realiza Java CUP para generar las clases correspondientes.
-nopositions	Con esta opción se consigue que no se propague los valores de columna, y fila (left y right) de los terminales a los no terminales. Esto provoca un ahorro importante de computación en el runtime de Java CUP y se debe utilizar siempre que no vayamos a utilizar dichos valores.

-version	Esta opción muestra la versión que estamos utilizando de Java CUP.

B.3 Personalización de la clase generada por Java CUP

A partir de las posibilidades que nos permite Java CUP para introducir código en diferentes partes del parser generado, ver apartado 3.1.2, y conociendo algunos métodos de la clase generada o de la clase padre, lr_parser, podemos conseguir personalizar algunos aspectos del comportamiento del parser generado. Para ello podemos implementar o redefinir cada uno de estos métodos:

- **public void user_init()**: Este método se llama antes de realizar la primera petición al scanner, por lo tanto contiene el código de la directiva *init with {: ... :}* explicada posteriormente
- **public java_cup.runtime.Symbol scan():** Este método encapsula el scanner y es llamado por el parser cada vez que se requiere un nuevo terminal.
- **public java_cup.runtime.Scanner getScanner()**: Esta función devuelve el scanner que está utilizando el parser para solicitar los nuevos terminales.
- **public void setScanner(java_cup.runtime.Scanner s):** Fija el scanner que utilizará el parser para leer los terminales a partir del método scan.

- **public void report_error(String message, Object info):** Este método originalmente imprime mediante System.err el mensaje pasado mediante el parámetro message. El parámetro info contiene el objeto que se estaba analizando pero en la implementación por defecto es ignorado. Se puede redefinir para conseguir un sistema de reporte de errores más complejo.
- **public void report_fatal_error(String message, Object info):** Este método es informa al usuario cuando un error irrecuperable sucede. Originalmente este método llama a report_error, posteriormente ejecuta el método done_parsing() que finaliza la ejecución del parser y lanza una excepción.
- **public void syntax_error(Symbol cur_token)public void:** Este método es llamado cada vez que un error sintáctico sucede pero éste es recuperable. La implementación original simplemente llama al método report_error(“Syntax error”, null).
- **unrecovered_syntax_error(Symbol cur_token):** Éste método es llamado cuando se encuentra un error sintáctico irrecuperable. Su implementación original hace una llamada a report_fatal_error("Couldn't repair and continue parse", null);

C. Código Fuente y Ficheros de Prueba

Es posible descargar el código fuente, las especificaciones de JFlex y Java CUP así como los ficheros de pruebas y release del software en el repositorio público de github siguiente: https://github.com/txetxuvel/sPascalator