Universidad Complutense de Madrid

Facultad de Informática

Compilador

Trabajo realizado por

Ana María Martínez Gómez
Víctor Adolfo Gallego Alcalá

2015

Procesadores de Lenguajes

Índice general

Introducción

A lo largo de este documento se explican las características del lenguaje utilizado y del proceso de compilación de programas en este lenguaje a código de la máquina-P. Nuestro lenguaje está basado en C++, aunque no se disponen de clases, paquetes ni memoria dinámica. Tampoco se puede hacer uso del tipo char ni de palabras reservadas como short y long, debido a la imposibilidad de implementarlas en la máquina-P. Se han introducido otras diferencias que consideramos mejoras de C++ (funciones anidadas, un solo return en las funciones, la función principal no tiene porque ser el main,etc.) y que se detallan la siguiente sección. También se presentan una sección con las reglas de la gramática y una sección con ejemplos.

Especificaciones

Especificaciones básicas

Se han implementado todas los requisitos mínimos pedidos en el enunciado de la práctica:

1. Declaración de variables simples y de matrices con número arbitrario de dimensiones (con límite inferior 0 como en C++, aunque podría generalizar fácilmente si se quisiese). Bloques anidados mediante { } o mediante las instrucciones de selección (*if*) o iteración (*while*) que introducen un nuevo ámbito.

2. Tipos predefinidos enteros (int) y booleanos (bool). Declaración explicita del tipo y tipos sin nombre (por ejemplo *struct*). Operadores infijos y algoritmo ascendente de comprobación de tipos con equivalencia estructural de tipos.

3. Instrucción de asignación incluyendo elementos de array (deben asignarse de uno para facilitar la comprensión de código), ifs con una o dos ramas (else opcional) y whiles. Expresiones formadas por constantes, identificadores (con subíndices en el caso de las matrices o structs, cada uno con su sintaxis) y operadores infijos.

4. Se informa del tipo de error así como de la fila y la columna para toda clase de errores (léxicos, sintácticos y semánticos). Para poder proporcionar la columna hicimos uso de JFlex, versión mejorada de JLex.

Especificaciones avanzadas

1. **Registros** Se disponen de structs, que permiten asignar un tipo anónimo no básico a las variables. Un struct está formado por un conjunto no vacío de campos de tipos primitivos, matrices u otros structs (structs anidados). Se ha decidido que al igual que en Ada los tipos anónimos se consideran distintos aunque estructuralmente sean iguales.

2. **Procedimientos y funciones** Se permiten usar tanto procedimientos (funciones de tipo void) como funciones que devuelven algún valor del tipo de la función. Por seguridad en las funciones se fuerza que la última instrucción sea un return, no dando la posibilidad al programador de que se deje ramas de una instrucción de selección sin devolver un valor o que escriba código que nunca va a ser ejecutado.

3. **Llamadas a procedimientos y funciones** Se pueden pasar un número arbitrario de parámetros de tipo primitivo (o una componente de un struct o matriz), aunque por seguridad no se permite la sobrecarga.

4. **Gestión de errores** Se le da al usuario la posibilidad de elegir si desea detener la compilación en el momento en que se detecta el primer error, o si por el contrario quiere que se trate de buscar todos los errores posibles. De esta forma, el programador podrá corregir varios errores tras tratar de compilar una sola vez. Esto se ha realizado con los errores léxicos, sintácticos y semánticos, aunque algunos errores sintácticos se consideran irrecuperables. Para poder recuperarse de los errores sintácticas añadimos una derivación error en la gramática en todas las reglas en la que era posible. Los errores semánticos fueron tratados mediante distinción de caso y con excepciones o mensajes.

Especificaciones avanzadas no especificadas en el enunciado

1. **Variables globales** En la sección globales se declaran las variables globales (pueden ser usadas en cualquier función).

2. **Inicialización de variables** Se permite la inicialización tanto de variables de tipo básico (int, bool) como de arrays en su declaración.

3. **Modificador const** Con el fin de hacer los programas más seguros y tolerantes a cambios, se permite declarar variables constantes que no pueden ser asignadas (su valor es siempre el de su declaración).

4. **Comprobación de índices en las matrices** Se comprueba en tiempo de ejecución que los índices de una matriz sean correctos.

5. **Selección de función de comienzo** Se permite elegir cualquier función como punto de inicio que por tanto no tiene porque ser la función main.

6. **Funciones anidadas** Como funcionalidad extra no permitida en C++, se permite la declaración de funciones en cualquier ámbito. Además a diferencia de otros lenguajes como Pascal permitimos la declaración de funciones intercaladas con la declaración de variables, por tanto en el código de la función solo se tiene acceso a las variables de ese ámbito declaradas antes que la función.

7. **Operadores prefijos y postfijos** Soportamos operadores prefijos (**!**) y postfijos (**++** y **−−**). Estos últimos se explican en el siguiente punto. Se permite la utilización de estos operadores en las expresiones.

8. **Operadores mejorados** Con el fin de hacer más fácil la escritura y comprensión del código se han introducido los operadores **+ =**,**− =**,**∗ =**,**/ =**,**% =**,**++** y **−−**.

9. **Implementación de operaciones no soportadas por la maquina-P** Con el fin de hacer más fácil la tarea de escribir programas muy comunes (como el de dar la imagen especular de un número, ver ejemplo1) se ha implementado el operador módulo, operación no disponible en la maquina-P.

10. **Generación de código con etiquetas** Para una mejor comprensión y una depuración más sencilla del código generado, se ofrece al usuario la posibilidad de obtener una versión que utilice etiquetas para indicar los saltos y los procedimientos. Podríamos generar el código también en este caso quitando las instrucciones que generan los errores, pero aunque la implementación es muy sencilla hemos preferido no hacerlo porque el código obtenido seguramente no sea coherente y podría confundir al programador que podría equivocarse de versión.

11. **Árbol sintáctico** Se genera un .txt con el árbol sintáctico. No se imprimen todos los nodos para hacer más sencilla su visualización.

Gramática

La gramática en formato .cup puede consultarse en el archivo *Gramatica.cup*, pero dado que la sintaxis y los atributos hacen dificil la lectura de la misma, se presenta a continuación una versión solo con las reglas más legible:

1. programa $\longrightarrow$ **globales** listaDeclaracionVariables **codigo** listaDeclaracionFunciones

2. listaDeclaracionVariables ⟶ listaDeclaracionVariables tipo listaVariables **;** | ϵ

3. listaDeclaracionVarFun ⟶ listaDeclaracionVariables tipo listaVariables **;** | listaDeclaracionVariables declaracionFuncion | ϵ

4. listaVariables ⟶ listaVariables**,** variable | variable

5. variable ⟶ **ID** | **ID =** expresionSimple | **ID** listaDimensiones | **ID** listaDimensiones = expresionArray

6. listaDimensiones ⟶ listaDimensiones **ID[CONSTANTE-ENTERO]** | **ID[CONSTANTE-ENTERO]**

7. tipo ⟶ **const** tipoSimple | tipoSimple

8. tipoSimple ⟶ **int** | **bool** | **struct** { listaStruct }

9. listaStruct ⟶ listaStruct tipo **ID ;** | tipo **ID ;** | listaStruct tipo **ID** listaDimensiones **;** | tipo **ID** listaDimensiones**;**

10. listaDeclaracionFunciones ⟶ listaDeclaracionFunciones declaracionFuncion | declaracionFuncion

11. declaracionFuncion ⟶ **fun** tipoSimple **ID (** parametros **)**{ listaDeclaracionVarFun listaInst instRetorno} | **fun void ID (** parametros **)** instCompuesta

12. parametros ⟶ listaParametros | ϵ

13. listaParametros ⟶ listaParametros **,** parametro | parametro

14. parametro ⟶ tipo **ID** | tipo **ID** listaDimensiones

15. instruccion ⟶ instExpresion | instCompuesta | instSeleccion | instIteracion | instNada

16. instCompuesta ⟶ { listaDeclaracionVarFun listaInst }

17. listaInst ⟶ listaInst instruccion | ϵ

18. instExpresion ⟶ expresion **;**

19. instSeleccion ⟶ **if (** expresionSimple **)** instruccion **end** | **if (** expresionSimple **)** instruccion **else** instruccion **end**

20. instIteracion ⟶ **while (** expresionSimple **)** instruccion **end** | **for (** listaDeclaracionVariables expresionRelacional **;** expresion) instruccion **end**

21. instRetorno ⟶ **return** expresionSimple **;**

22. instNada ⟶ **skip ;**

23. expresion ⟶ modificable **=** expresionSimple | modificable **+ =** expresionAritmetica | modificable **− =** expresionAritmetica | modificable **∗ =** expresionAritmetica | modificable **/ =** expresionAritmetica | modificable **% =** expresionAritmetica | modificable **++** | modificable **−−**

24. expresionArray ⟶ { listaExpresionesArray }

25. listaExpresionesArray ⟶ listaExpresionesArray**,** expresionSimple | expresionSimple

26. expresionSimple ⟶ expresionSimple **||** expresionConj | expresionConj

27. expresionConj ⟶ expresionConj **&&** expresionUnariaORelacional | expresionUnariaORelacional

28. expresionUnariaORelacional ⟶ **!** expresionUnariaORelacional | expresionRelacional

29. expresionRelacional ⟶ expresionAritmetica *opRelacional* expresionAritmetica | expresionAritmetica

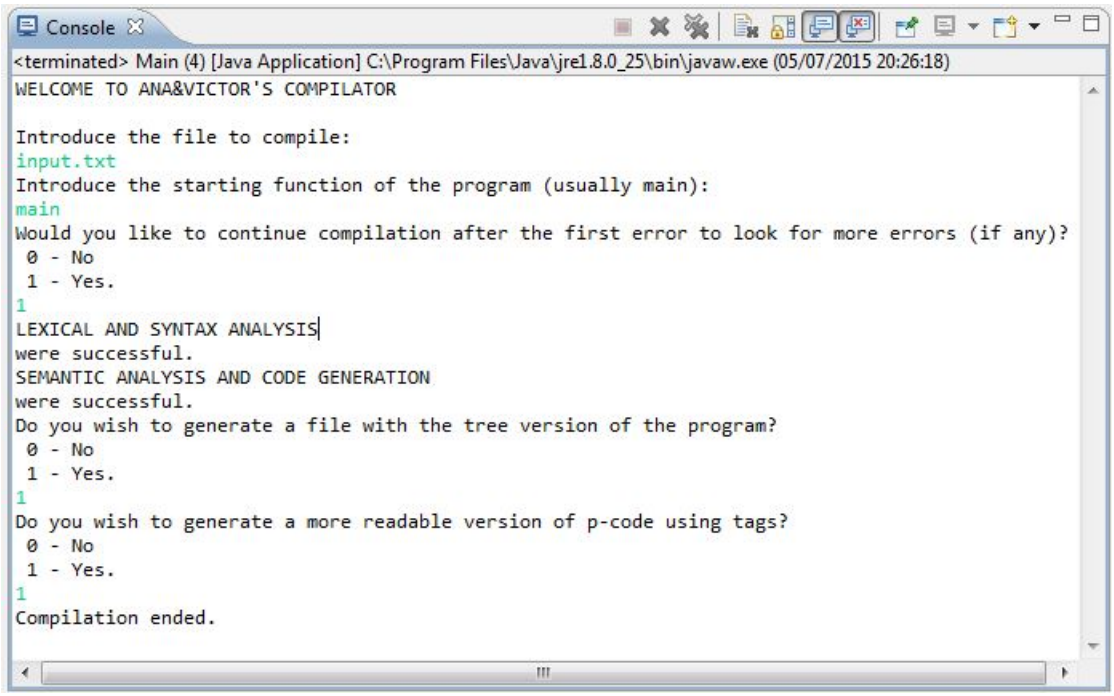
30. *opRelacional* ⟶ **<=** | **<** | **>** | **>=** | **==** | **! =**

31. expresionAritmetica ⟶ expresionAritmetica *opSum* termino | termino

32. *opSum* ⟶ **+** | **−**
33. termino ⟶ termino *opMul* factor | factor
34. *opMul* ⟶ ***** | **/** | **%**
35. factor ⟶ inmodificable | modificable
36. modificable ⟶ **ID** | **ID** listaModificable | **ID − >** modificable | **ID** listaModificable **− >** modificable
37. inmodificable ⟶ **(** expresionSimple **)** | llamada | constante
38. llamada ⟶ **ID (** listaArgumentos **)**
39. listaArgumentos ⟶ listaArgumentos **,** expresionSimple | expresionSimple | ϵ
40. listaModificable ⟶ listaModificable [expresionAritmetica] | [expresionAritmetica]
41. constante ⟶ **CONSTANTE-ENTERO** | **CONSTANTE-CHAR** | **CONSTANTE-STRING** | **CONSTANTE-BOOL**

Uso del compilador

Al ejecutar el compilador se muestra por consola un menú con opciones. En este menú se pide el fichero de entrada con el código a compilar, la función de comienzo (que normalmente es main). Además da la opción de tratar de recuperarse de errores, de imprimir el árbol sintáctico del programa (se guardará en *tree.txt*) y de generar una versión del código con etiquetas (*code-tags.p*). Como salida siempre se obtiene el código compilado para su uso en la máquina-P en *code.p*.



Ejemplos

En la carpeta ejemplos se encuentran todos los ficheros a los que se hace referencia en esta sección, cada uno de ellos en su carpeta ejemploX, donde X es el número del ejemplo. Para todos los que no producen un error en *tree.txt* se encuentra el árbol sintáctico, en *code-tags.p* la versión del código con etiquetas y en *code.p* el código para la máquina-P.

Ejemplo1 - Imagen especular de un número

En *input1.txt* se tiene un código que calcula la imagen especular de un número. En el código se incluyen funciones anidadas, el operador especial %, el operador mejorado / =, llamadas a funciones anidadas e inicialización de variables en su declaración. Al ejecutarlo con la maquina-P se obtiene el siguiente resultado (673 la imagen especular de 376):

```
*Main> :main code.p
Run successfully completed
Final P-machine state:
PC = 77   MP = 0   SP = 5   EP = 54   NP = 100   code size = 78
STORE =
(0,Null)
(1,Null)
(2,Null)
(3,Null)
(4,Null)
(5,Int 673)
(6,Int 0)
(7,Int 0)
(8,Int 54)
(9,Int 77)
(10,Int 376)
(11,Int 673)
(12,Int 673)
(13,Int 673)
(14,Int 5)
```

Ejemplo2 - Structs y matrices

En *input2.txt* se tiene un código con accesos y asignaciones a structs y matrices.

Ejemplo3 - Error léxico y sintáctico

En *input3.txt* se tiene un código con variables globales y donde se asigna una variable constante y donde se asigna un booleano a un entero. El error que se muestra por consola es el siguiente (se le ha indicado que busque todos los errores):

```
LEXICAL AND SYNTAX ANALYSIS
Lexical error at line 5, column 7 : Illegal character <ñ>
instead expected token classes are []
Syntax error for input symbol "=" spanning from unknown:5/9(55) to unknown:5/9(56)
Couldn't repair and continue parse for input symbol "EOF" spanning from unknown:9/1(95) to unknown:9/1(96)
java.lang.Exception: Can't recover from previous error(s)
Compilation ended.
```

Ejemplo4 - Errores semánticos

En *input4.txt* se tiene un código con variables globales y donde se asigna una variable constante, se tiene una condición no booleana en un if, se asigna un booleano a un entero y se utiliza una variable no declarada. Los errores que se muestran por consola son los siguientes (se le ha indicado que busque todos los errores):

```
LEXICAL AND SYNTAX ANALYSIS
were successful.
SEMANTIC ANALYSIS AND CODE GENERATION
Compilation ended. Error:Assigment of the constant variable b in line 7, column 3.
Error: If condition is not a boolean expression.
Error: Assignment of an incompatible typed value in line 9, column 5
Error: Identificator f not declared beginning at line 0, column 0.
```