Práctica 2: Construcción de un analizador sintáctico para *gcl*

Procesadores de lenguajes

Dpto. de Informática e Ingeniería de Sistemas,
Grado de Ingeniería Informática
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

1. Objetivos

Los objetivos para esta práctica son:

- Desarrollar un analizador sintáctico descendente recursivo para *gcl*
- Integrar el análisis léxico con el análisis sintáctico
- Entender los requisitos para desarrollar un analizador descendente predictivo

2. Un ejemplo de analizador sintáctico con *javacc*

En moodle, en la sección correspondiente a la práctica 2, podéis encontrar un ejemplo de analizador sintáctico para una calculadora de enteros, que permite declarar constantes y variables. Se encuentra en el fichero `calc_enteros_sint.zip`. Para entender la forma en que se han de describir las producciones en *javacc*, deberéis leer documentación relacionada con la herramienta. El objetivo es que sirva de ejemplo para facilitar el inicio del trabajo a desarrollar, que se detalla a continuación.

3. Descripción del trabajo a realizar

En esta práctica se debe desarrollar el analizador sintáctico para *gcl*. Al final del documento se muestra una posible gramática (incompleta) para el lenguaje. Se muestra completa la parte correspondiente a las expresiones, pues, usualmente, es una de las partes más complicadas de una gramática para un lenguaje de programación. Como se

ve, y por simplificar el desarrollo del compilador, la gramática hace uso de la cláusula *LOOKAHEAD* en algunas de las producciones. Esto hace que el analizador sintáctico generado por *javacc* mire más de un símbolo de preanálisis antes de tomar la decisión de qué producción disparar. Es importante recordar que la transformación en una gramática LL(1) es muy sencillo, si así fuera necesario, mediante la adecuada factorización a izquierda. Así, las producciones

```
variable :
  LOOKAHEAD(2)
  <tID> <tACOR> <tCONST_INT> <tCCOR>
| <tID>
```

se podrían transformar en LL(1) como sigue:

```
variable : <tID> mas_variable

mas_variable :
  <tACOR> <tCONST_INT> <tCCOR>
| //corresponde a epsilon
```

Nótese también que estamos usando notación EBNF que, igualmente, puede transformarse en LL(1) si fuera necesario. Por ejemplo:

```
lista_cero_o_mas_exps: ( lista_una_o_mas_exps )?
```

se escribiría como

```
lista_cero_o_mas_exps :
   lista_una_o_mas_exps
| //corresponde a epsilon
```

4. Errores sintácticos

No es necesario aplicar ninguna política de recuperación de errores. En caso de detectar un error sintáctico, es suficiente con el mensaje por defecto que genera javacc, que informa de cuáles son los tokens que esperaban.

5. Entrega de resultados de la práctica

5.1. Lo que hay que entregar

Como resultado de la práctica se deberá entregar:

- El fichero `practica_2.zip`. Este, una vez descomprimido, tendrá la misma estructura que en la práctica 1. El fichero `README.txt`, del directorio `Doc` con la información de uso del compilador.

- El fichero `tests.zip`. Este, una vez descomprimido, deberá contener tres programas de prueba propios en *gcl*. Los fuentes se deben llamar `test_1.al`, `test_2.al` y `test_3.al`. Se valorará que los programas tengan algún interés.

5.2. Método de entrega

La entrega se debe hacer en *lab000.cps.unizar.es* mediante la ejecución del programa `someter`, análogamente a como se hizo en la práctica anterior

5.3. Plazos de entrega

El plazo de entrega es el establecido en moodle, y dependerá del grupo de prácticas.

Gramática parcial para *gcl*

Lo que sigue es una posible gramática (parcial) para *gcl*, con el objetivo de que sirva de ayuda para el desarrolo de la práctica. No es necesario utilizarla. Cada pareja puede desarrollar una correcta completamente diferente, o adaptar esta según sus necesidades. Para enteder qué representa cada token o cada producción, hay que mirar los programas entregados en la batería de test suministrada.

```
1  Programa:
2      <tPROGRAMA>
3      <tID>
4      ( declaracion_variables )?
5      ( declaracion_procs_funcs )?
6      <tPRINCIPIO>
7      instrucciones
8      <tFIN>
9
10 declaracion_variables:
11     ( declaracion_vars )+
12
13 declaracion_vars:
14     tipo_escalar_o_array
15     lista_ids
16     <tPC>
17
18 ...
19
20 instruccion() :
21     inst_leer
22 | inst_saltar_linea
23 | inst_escribir
24 | inst_escribir_linea
25 | inst_asignacion
26 | inst_if
27 ....
28
29
30 expresion:
31     relacion ( (< tAND > | < tOR > ) relacion )*
32
33 relacion:
34     expresion_simple ( operador_relacional expresion_simple )?
35
36 operador_relacional:
37     <tIG> | <tMEN> | <tMAY> | <tMENI> | <tMAYI> | <tDIF>
38
39 expresion_simple:
40     termino ( ( <tMAS> | tMENOS> ) termino )*
41
```

```
42 termino:
43     factor ( operador_multiplicativo factor )*
44
45 operador_multiplicativo:
46     <tPROD> | <tMOD> | <tDIV>
47
48 factor:
49     ( <tNOT> | <tMAS> | <tMENOS> )? ( <tAPAR> expresion <tCPAR> | primario )
50
51 primario:
52     <tENTACAR> <tAPAR> expresion <tCPAR>
53 | <tCARAENT> <tAPAR> expresion <tCPAR>
54 | LOOKAHEAD(2)
55     <tID> <tAPAR> lista_una_o_mas_exps <tCPAR> //invoc a func.
56 | LOOKAHEAD(2)
57     <tID> <tACOR> expresion <tCCOR> //comp. array
58 | <tID>
59 | <tCONST_INT>
60 | <tCONST_CHAR>
61 | <tCONST_STRING>
62 | <tTRUE>
63 ...
```