PRÁCTICA DE PROCESADORES DEL LENGUAJE I

Curso 2023 – 2024

Entrega de Febrero | Septiembre

APELLIDOS Y NOMBRE: Osta Supervia Noelia

IDENTIFICADOR: nosta1

DNI: 17768378G

CENTRO ASOCIADO MATRICULADO: Calatayud

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: Calatayud

MAIL DE CONTACTO: nosta1@alumno.uned.es / noeliaosta@gmail.com

TELÉFONO DE CONTACTO: 644417668

1. El analizador léxico.

En el analizador léxico se usa la herramienta JFlex. Se han identificado todos los tokens que se pedían en el enunciado de la práctica: las palabras reservadas, los delimitadores y los operadores.

Para reconocer los tokens y no repetir el código en exceso, en las líneas 35 a 41 se define una función para crear los tokens.

Además de los tokens nombrados anteriormente, se definen también tokens para los identificadores, las cadenas y los números.

- Identificadores. Se detectan mediante la siguiente expresión regular: [a-zA-Z][a-zA-Z0-9]*
- Números: He decidido distinguir dos casos: El número correcto, el cual devuelve un token y en caso de número mal formado, se devuelve un Lexical Error como se puede comprobar en las líneas 99 – 105.
- Strings. Aunque el reconocimiento de las cadenas también se podía hacer mediante una expresión regular, he decido hacerla por estados para aprender su uso. Además, se hace uso del StringBuffer de java para manejar los strings (y sobre todo para eliminar las comillas que no podían pasar al analizador sintáctico). Por ello, en la linea 99 cuando se detecta el lexema "--" comienza el estado comentario, el cual, devuelve el token a no ser que en el comentario se detecten caracteres de control ASCII (identificados mediante la negación de la expresión regular que controla que no existen esos carácteres) o se detecte que el comentario esta malformado.

2. El analizador sintáctico.

A indicar en este apartado, que para evitar las producciones épsilon (puesto que me provocaron muchísimos problemas) se ha optado por hacer las producciones lo más largas posibles y evitando a toda costa los épsilon, si era necesario, repitiendo producciones "similares" (por ejemplo, en el caso del axiom, declarando todas las opciones que se han visto, es decir, que el programa este vacío "linea 183", que el programa no tenga declaraciones pero si lista de sentencias "línea 184", que el programa tenga declaraciones y sentencias "linea 185" y por último, el estado especial de error indicando que falta el *end* final).

No se han tratado todos los posibles errores, he ido tratando los que yo consideraba necesarios para avanzar con el desarrollo de la práctica.

La estructura del fichero parser es:

- Declaración de terminales. Son aquellos que están identificados por los Tokens que han sido aceptados del análisis léxico (Por tanto se excluyen comentarios, *strings* malformadas y números erróneos).
- Declaración de no terminales. Todos los que he necesitado para construir la gramática.
- Declaraciones de precedencia. Se ha puesto la precedencia que marcaba el enunciado de la práctica.

- Axiom. Inicio del programa, comienza con la declaración del procedimiento principal y a partir de producciones se va ampliando la gramática para cumplir todos los objetivos de la práctica.

Se ha intentado seguir más o menos el orden propuesto en la práctica para facilitar su corrección.

3. Conclusiones.

Aunque la parte léxica me resulto "sencilla" la parte sintáctica me costó bastante sobretodo empezarla. A pesar de los videos del ED no sabía muy bien por donde empezar con esta parte, y el no poder *debbugear* el código me ha resultado muy molesto con algunos errores. Además, al principio, tuve muchísimos errores por las producciones épsilon y me costo mucho acostumbrarme a no ponerlas y especificar la gramática lo máximo posible.

Quitando esos detalles, las prácticas son la mejor manera de desenvolverse y aprender acerca de un lenguaje o una asignatura. Si tuviera que volver a hacer la práctica desde luego las horas invertidas no llegarían a la mitad, y considero que personalmente me ha resultado de utilidad.

4. Gramática.

Se copia la gramática resultante (eliminando la declaración de terminales, no terminales y las reglas de precedencia).

```
axiom ::= main_procedure_declaration BEGIN end
    | main_procedure_declaration BEGIN sentences_list end
    | main_procedure_declaration declarations BEGIN end
    | main_procedure_declaration declarations BEGIN sentences_list end //La lista de sentencias y la declaracion de los bloques puede ser vacía.
    | main_procedure_declaration declarations BEGIN sentences_list error{:syntaxErrorManager.syntaxFatalError("ERROR: Error en el end final"); :};

main_procedure_declaration ::= PROCEDURE IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS IS
                    | PROCEDURE error{:syntaxErrorManager.syntaxFatalError("ERROR: Falta el identificador"); :} OPENPARENTHESIS CLOSEPARENTHESIS IS
                    | PROCEDURE IDENTIFIER error{:syntaxErrorManager.syntaxFatalError("ERROR: Falta el parentesis de apertura"); :} CLOSEPARENTHESIS IS
                    | PROCEDURE IDENTIFIER OPENPARENTHESIS error{:syntaxErrorManager.syntaxFatalError("ERROR: Falta el parentesis de cierre"); :} IS;

//Para evitar las proyecciones epsilon, aqui se detallan todas las posibles opciones considerando que todas pueden ser opcionales
declarations ::= const_declarations_list
               | record_declarations
               | var_declarations
               | subprograms_declaration_list
               | const_declarations_list record_declarations
               | const_declarations_list var_declarations
               | const_declarations_list subprograms_declaration_list
               | record_declarations var_declarations
               | record_declarations subprograms_declaration_list
               | var_declarations subprograms_declaration_list
               | record_declarations var_declarations subprograms_declaration_list
               | const_declarations_list record_declarations var_declarations
               | const_declarations_list record_declarations subprograms_declaration_list
               | const_declarations_list var_declarations subprograms_declaration_list
               | const_declarations_list record_declarations var_declarations subprograms_declaration_list;

const_declarations_list ::= const_declaration
                  | const_declarations_list const_declaration;
```

```
214 const_declaration ::= IDENTIFIER COLON CONSTANT ASSIGNMENT const_types SEMICOLON;
215
216 //Sólo pueden ser del tipo entero o lógico.
217 const_types ::= NUMBER
218     | boolean ;
219
220 boolean ::= TRUE
221     | FALSE;
222
223 record_declarations ::= record_declaration
224     | record_declarations record_declaration;
225 record_declaration ::= TYPE IDENTIFIER:id IS RECORD record_list record_end;
226 record_list ::= record_field record_list
227     | record_field;
228 record_field ::= IDENTIFIER COLON primitive_types SEMICOLON
229     | IDENTIFIER COLON IDENTIFIER SEMICOLON;
230 record_end ::= END RECORD SEMICOLON
231     | error {: syntaxErrorManager.syntaxDebug ("Error: Final de registro erroneo"); :};
232
233 end ::= END IDENTIFIER SEMICOLON;
234
235 var_declarations ::= var_declaration_list
236     | var_declarations var_declaration_list ;
237
238 var_declaration_list ::= var_declaration
239     | IDENTIFIER COMMA var_declaration_list;
240
241 var_declaration ::= IDENTIFIER COLON IDENTIFIER:id SEMICOLON
242     | IDENTIFIER COLON primitive_types SEMICOLON;
243
244 subprograms_declaration_list ::= subprogram_declaration
245     | subprograms_declaration_list subprogram_declaration;
246
247 subprogram_declaration ::= function_declaration | procedure_declaration;
248
```

```
249 function_declaration ::= FUNCTION IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS RETURN primitive_types IS BEGIN function_body_list function_end
250                        | FUNCTION IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS RETURN primitive_types IS local_declarations_list BEGIN function_body_list function_end
251                        | FUNCTION IDENTIFIER OPENPARENTHESIS function_parameters CLOSEPARENTHESIS RETURN primitive_types IS BEGIN function_body_list function_end
252     | FUNCTION IDENTIFIER OPENPARENTHESIS function_parameters CLOSEPARENTHESIS RETURN primitive_types IS local_declarations_list BEGIN function_body_list function_end
253     | FUNCTION IDENTIFIER error {: syntaxErrorManager.syntaxFatalError ("Error: Faltan los dos parentesis"); :} RETURN primitive_types IS local_declarations_list BEGIN funct
254 function_parameters ::= parameter_declarations
255     | IDENTIFIER COMMA function_parameters;
256
257 parameter_declarations ::= parameter_declaration
258     | parameter_declaration COMMA parameter_declarations;
259
260 parameter_declaration ::= IDENTIFIER COLON IDENTIFIER
261     | IDENTIFIER COLON primitive_types
262     | IDENTIFIER COLON OUT IDENTIFIER
263     | IDENTIFIER COLON OUT primitive_types;
264
265 function_body_list ::= function_body | function_body_list function_body;
266 function_body ::= assignment_sentence
267     | procedure_declaration
268     | function_call
269     | return_types
270     | if
271     | while;
272
273 function_call ::= single_function_call SEMICOLON
274     | ES_sentences;
275
276 single_function_call ::= IDENTIFIER OPENPARENTHESIS parameter_function_call_list CLOSEPARENTHESIS;
277
278 parameter_function_call_list ::=
279     | parameter_list
280     | parameter_list COMMA parameter_function_call_list;
```

```
parameter_list ::= expression
          | IDENTIFIER
          | STRING
          | NUMBER
          | record_access_list
          | single_function_call;

procedure_declaration ::= PROCEDURE IDENTIFIER OPENPARENTHESIS function_parameters CLOSEPARENTHESIS IS BEGIN procedure_body_list end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS function_parameters CLOSEPARENTHESIS IS BEGIN end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS function_parameters CLOSEPARENTHESIS IS local_declarations_list BEGIN procedure_body_list end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS function_parameters CLOSEPARENTHESIS IS local_declarations_list BEGIN end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS IS BEGIN end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS IS BEGIN procedure_body_list end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS IS local_declarations_list BEGIN end
    | PROCEDURE IDENTIFIER OPENPARENTHESIS CLOSEPARENTHESIS IS BEGIN local_declarations_list procedure_body_list end;

procedure_body_list ::= procedure_body | procedure_body_list procedure_body;
procedure_body ::= assignment_sentence
    | function_call
    | if
    | while
    | RETURN error {: syntaxErrorManager.syntaxFatalError ("Error: En un subprograma procedimiento no hay sentencia return"); :};

ES_sentences ::= PUTLINE OPENPARENTHESIS parameter_putline_call CLOSEPARENTHESIS SEMICOLON;
parameter_putline_call ::= IDENTIFIER
    | STRING
    | NUMBER
    | expression
    | boolean
    | record_access_list
    | error{:syntaxErrorManager.syntaxFatalError("ERROR: Error en la llamada de putline"); :};
```

```
// Declaraciones despues del IS del programa
local_declarations_list ::= local_declarations | local_declarations_list local_declarations;
local_declarations ::= procedure_declaration | local_var;

sentences_list ::= sentences
    | sentences_list sentences;

sentences ::= ES_sentences
    | assignment_sentence
    | if
    | while
    | return_types
    | IDENTIFIER COLON CONSTANT error{:syntaxErrorManager.syntaxFatalError("ERROR: No se permite la declaracion de constantes en este bloque"); :};

assignment_sentence ::= IDENTIFIER ASSIGNMENT expression SEMICOLON
    | IDENTIFIER ASSIGNMENT boolean SEMICOLON
    | IDENTIFIER ASSIGNMENT single_function_call SEMICOLON
    | IDENTIFIER ASSIGNMENT NUMBER SEMICOLON
    | IDENTIFIER ASSIGNMENT record_access_list SEMICOLON
    | record_access_list ASSIGNMENT expression SEMICOLON
    | record_access_list ASSIGNMENT boolean SEMICOLON
    | record_access_list ASSIGNMENT single_function_call SEMICOLON
    | record_access_list ASSIGNMENT NUMBER SEMICOLON
    | record_access_list ASSIGNMENT record_access_list SEMICOLON;

record_access_list ::= IDENTIFIER DOT IDENTIFIER
    | IDENTIFIER DOT record_access_list;

if ::= IF boolean_expression THEN loop_sentences_list END IF SEMICOLON
    | IF boolean_expression THEN loop_sentences_list ELSE loop_sentences_list END IF SEMICOLON;

while ::= WHILE boolean_expression LOOP loop_sentences_list END LOOP SEMICOLON;
```

```
loop_sentences_list ::= loop_sentences | loop_sentences_list loop_sentences;
loop_sentences ::= ES_sentences
    | assignment_sentence
    | if
    | while
    | single_function_call SEMICOLON
    | return_types
    | IDENTIFIER COLON CONSTANT
    | error{:syntaxErrorManager.syntaxFatalError("ERROR: Sentencia no reconocida en el bloque loop"); :};

boolean_expression ::= TRUE
    | FALSE
    | IDENTIFIER
    | single_function_call
    | expression_types logic_expression expression_types
    | OPENPARENTHESIS boolean_expression CLOSEPARENTHESIS
    | boolean_expression AND boolean_expression;

expression ::= expression_types logic_expression expression_plus_function
    | expression_types arithmetic_expression expression_plus_function;

logic_expression ::= GREATER
    | DISTINCT;
arithmetic_expression ::= MINUS
    | MULTIPLY;

expression_types ::= IDENTIFIER | NUMBER | record_access_list;
expression_plus_function ::= IDENTIFIER | NUMBER | record_access_list | single_function_call;
```

```
376 local_var ::= IDENTIFIER COLON primitive_types SEMICOLON
377         | error{:syntaxErrorManager.syntaxFatalError("ERROR: Error en la declaracion de la variable"); :};
378 function_end ::= END IDENTIFIER SEMICOLON;
379
380 return_types ::= RETURN IDENTIFIER SEMICOLON
381     | RETURN expression_types logic_expression expression_types SEMICOLON
382     | RETURN expression_types arithmetic_expression expression_types SEMICOLON
383     | RETURN NUMBER SEMICOLON
384     | RETURN boolean SEMICOLON;
385
386 assignment_types ::= boolean | NUMBER;
387
388 primitive_types ::= INTEGER
389     | BOOLEAN;
```