PRÁCTICA DE PROCESADORES DEL LENGUAJE I - Curso 2012 – 2013 - Entrega de Febrero

May 2, 2015

APELLIDOS Y NOMBRE: **Martínez García, Octavio**
IDENTIFICADOR: **omartinez185**
DNI: **71280002-N**
CENTRO ASOCIADO MATRICULADO: **CA Burgos**
CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: **CA Burgos**
MAIL DE CONTACTO: **omartinez185@alumno.uned.es**
TELÉFONO DE CONTACTO: **690 36 11 55**
GRUPO (A ó B): **A**

1 Analizador Léxico

En el archivo scanner.flex se han definido las especificaciones del analizador léxico del compilador:

- Se han definido los tokens del lenguaje, utilizando macros para definir aquellos con una estructura fija pero cuyas instancias pueden variar, como pueden ser los identificadores, y las constantes literales, aparte de otras macros que no definen tokens propiamente dichos, sino estructuras léxicas que deben ser tratadas de una forma específica por el analizador léxico, como son los comentarios, los espacios blancos y los saltos de línea.

- Se ha establecido un procedimiento, createToken(int sym), para crear nuevos tokens, el cual crea un nuevo token, utilizando las constantes enteras definidas en el analizador sintáctico, las cuales identifican unívocamente un token en la interfaz sym, establece su linea y columna en el programa fuente, así como su lexema de token, para finalmente devolver dicho token.

- Asimismo se ha definido otro procedimiento similar al anterior, createStringToken(int sym, String value), utilizado para crear tokens de literales de cadena de caracteres, cuya única diferencia respecto al anterior es que el lexema de token no se obtiene con en procedimiento interno de jflex, yytext(), sino desde el estado definido en el scanner, <STRING>, almacenando en el parámetro "value" la cadena de caracteres exceptuando las comillas de inicio y final.

- Como se comentaba anteriormente, se ha definido un estado <STRING>, en el cual entra el scanner cuando reconoce un " inicial, y dentro del cual se van añadiendo caracteres al stringbuffer "string" (exceptuando retornos de linea, " y \ ) hasta que se detecta un " final, punto en el cual el scanner retorna al estado inicial <YYINITIAL> devolviendo un nuevo token de String con su valor de lexema calculado correctamente

- Dentro del estado <YYINITIAL> se van reconociendo el resto de patrones de caracteres que corresponden a tokens del lenguaje y emitiendo los tokens correspondientes en cada caso, o ignorando las estructuras léxicas para las cuales no se debe emitir token, señaladas anteriormente.

- Cuando se detecta una estructura léxica no definida anteriormente, no perteneciente al lenguaje especificado, se emite un error léxico lanzando un nuevo LexicalError

- VER ESTADO COMMENT, (ANIDAMIENTO DE COMENTARIOS, DOC DE ESPECS DEL LENGUAJE)

2 Analizador Sintáctico

En el archivo parser.cup se han definido las especificaciones del analizador sintáctico del compilador:

- Se han definido los tokens del lenguaje, que representan símbolos terminales de la gramática del lenguaje, concordantes con los tokens que emite el analizador léxico, para establecer la compatibilidad entre jflex y cup (entre el scanner y el parser)
- Asimismo, se han definido los símbolos no terminales de la gramática que especifica al lenguaje, y que serán usados en las reglas de producción de dicha gramática
- Se han establecido las relaciones de precedencia y asociatividad (izquierda en todos los casos) de los operadores según se muestran en las especificaciones del lenguaje
- Finalmente se ha definido el conjunto de reglas de producción que definen la gramática del lenguaje, comenzando desde la estructura general de un programa correcto en HAda y refinando dichas reglas globales hasta definir el resto de reglas mas específicas. La gramática resultante se muestra de forma esquemática en la sección 4 de la memoria.
- VER SI INCLUIMOS EN LA MEMORIA LAS DECISIONES COMO ARRAY_ACCESS (SIMILAR A INVOC DE PROC DE 1 SOLO ARG) Y OTRAS DECISIONES DE DISEÑO, O NO ES NECESARIO ENTRAR EN DETALLES (COMENTADO EN CÓDIGO CUP)

3 Conclusiones

4 Gramática

```
program ::= axiom;

// estructura general del programa HAda
axiom ::= PROCEDURE IDENTIFIER LPAREN RPAREN IS declarations BEGIN fun_st_block END IDENTIFIER SEMIC
                | PROCEDURE IDENTIFIER LPAREN RPAREN IS declarations BEGIN END IDENTIFIER SEMICOL
                | PROCEDURE IDENTIFIER LPAREN RPAREN IS BEGIN fun_st_block END IDENTIFIER SEMICOL
                | PROCEDURE IDENTIFIER LPAREN RPAREN IS BEGIN END IDENTIFIER SEMICOL
                | error {: syntaxErrorManager.syntaxDebug ("Error en estructura del programa"); :};


// declaraciones iniciales
declarations ::= symbolic_constant_declaration | symbolic_constant_declaration declarations | tvs;

tvs ::= global_types_declaration | global_types_declaration tvs | vs;

vs ::= global_var_declaration | global_var_declaration vs | sp;

sp ::= subprog_declaration | subprog_declaration sp;



// declaracion de constantes simbolicas
symbolic_constant_declaration ::= list_ids COL CONSTANT ASSIGN literal_constant SEMICOL
                                                                | error SEMICOL {: syntaxErrorManage
literal_constant ::= LITERALINT | TRUE | FALSE;


// declaracion de tipos globales, excluimos tipos de params (paso por ref in/out)
global_types_declaration ::= structured_type_decl;

structured_type_decl ::= array_type_decl;

array_type_decl ::= TYPE IDENTIFIER IS ARRAY LPAREN array_indexes RPAREN OF array_values SEMICOL
                                | error {: syntaxErrorManager.syntaxDebug ("Error en declaracion d

array_indexes ::= a_index PNTPNT a_index;

a_index ::= LITERALINT | IDENTIFIER;

array_values ::= INTEGER | BOOLEAN;


// declaracion de vbles globales
global_var_declaration ::= var_decl IDENTIFIER SEMICOL
                                                | var_decl INTEGER SEMICOL
                                                | var_decl BOOLEAN SEMICOL;

var_decl ::= list_ids COL;

list_ids ::= IDENTIFIER COMMA list_ids | IDENTIFIER;


// declaracion de subprogramas
subprog_declaration ::= proced | func;

proced ::= PROCEDURE IDENTIFIER LPAREN ref_params RPAREN IS local_declarations BEGIN st_block END ID
                | PROCEDURE IDENTIFIER LPAREN RPAREN IS local_declarations BEGIN st_block END IDENT
                | PROCEDURE IDENTIFIER LPAREN RPAREN IS BEGIN st_block END IDENTIFIER SEMICOL
                | PROCEDURE IDENTIFIER LPAREN ref_params RPAREN IS BEGIN st_block END IDENTIFIER SE

ref_params ::= param SEMICOL ref_params | param;
param ::= var_decl io INTEGER | var_decl io BOOLEAN | var_decl io IDENTIFIER;
io ::= IN | OUT;

local_declarations ::= tvs;
```

```
st_block ::= sent st_block | sent;

sent ::= sent_assign | sent_if | sent_ctrl | sent_io | func_invoc SEMICOL | array_access SEMICOL
          | error SEMICOL {: syntaxErrorManager.syntaxDebug ("Error en sentencia"); :};

// sentencias de asignacion
sent_assign ::= IDENTIFIER ASSIGN exp SEMICOL
                        | array_access ASSIGN exp SEMICOL;

exp ::= exp op exp | LPAREN exp RPAREN | func_invoc | array_access | IDENTIFIER | LITERALINT | TRUE
         | error {: syntaxErrorManager.syntaxDebug ("Error en expresion"); :};

op ::= LESSER | NOTEQUAL | AND | MINUS;

array_access ::= IDENTIFIER LPAREN exp RPAREN;

func_invoc ::= IDENTIFIER LPAREN act_params RPAREN
                        | IDENTIFIER LPAREN RPAREN;

act_params ::= exp COMMA act_params | exp;


// sentencias if
sent_if ::= IF exp THEN st_block END IF SEMICOL
                 | IF exp THEN st_block ELSE st_block END IF SEMICOL;


// sentencias de control de flujo (solo while en esta espec-A)
sent_ctrl ::= WHILE exp LOOP st_block END LOOP SEMICOL;


// sentencias de entrada/salida (putline solo)
sent_io ::= PUTLINE LPAREN exp RPAREN SEMICOL
                 | PUTLINE LPAREN LITERALSTRING RPAREN SEMICOL;


// declaracion de funciones
func ::= FUNCTION IDENTIFIER LPAREN ref_params RPAREN RETURN INTEGER IS local_declarations BEGIN fun
          | FUNCTION IDENTIFIER LPAREN ref_params RPAREN RETURN INTEGER IS BEGIN fun_st_block END I
          | FUNCTION IDENTIFIER LPAREN RPAREN RETURN INTEGER IS local_declarations BEGIN fun_st_blo
          | FUNCTION IDENTIFIER LPAREN RPAREN RETURN INTEGER IS BEGIN fun_st_block END IDENTIFIER S

fun_st_block ::= f_sent fun_st_block | f_sent;

f_sent ::= sent_assign | sent_if | sent_ctrl | sent_io | sent_return | func_invoc SEMICOL | array_a

sent_return ::= RETURN exp SEMICOL;
```