Compiladores e Intérpretes
Manual del usuario

Diego Marcovecchio (LU: 83815) Leonardo Molas (LU: 82498)

2 de Diciembre de 2010

Índice general

Introducción

Este es el Manual del Usuario del Compilador de Mini-Pascal *pyComp*. Detallaremos el alcance del compilador, el lenguaje y los símbolos utilizados, los tipos de datos cubiertos, la forma de uso del lenguaje, la utilización en general del compilador, y las decisiones de diseño tomadas.

Capítulo 1

Modo de Uso

1.1. Requerimientos

Para ejecutar el compilador es necesario contar con las siguientes librerías (que pueden encontrarse en la carpeta entregada a la cátedra).

- `python27.dll`
- `msvcr90.dll`
- `bz2.pyd`
- `select.pyd`
- `unicodedata.pyd`
- `library.zip` (que contiene las librerías de Python utilizadas)

1.2. Ejecución

```
pyComp IN_FILE [OUT_FILE] [-h] [-d] [-o DISPLAY_FILE]
```

Argumentos

`IN_FILE` El archivo de Pascal de entrada.

Argumentos opcionales

`OUT_FILE` El archivo opcional de salida. En caso de especificarse, en éste archivo se generará el código MEPa correspondiente a `IN_FILE`; en caso de no especificarse, se removerán los últimos tres caracteres del `IN_FILE`, y se reemplazarán por ~~m~~epa~~;~~ creando un nuevo archivo con ese nombre.

`-d` Modo *debug* (utilizado durante el desarrollo, y dejado por la posible utilidad en el futuro).

`-h, --help` Muestra la ayuda por pantalla.

`-o DISPLAY_FILE` Hace que la salida del compilador se muestre en el archivo `DISPLAY_FILE`. Por defecto, `DISPLAY_FILE` es el archivo de salida standard del sistema operativo, por lo que de no especificarse, la salida será realizada por la pantalla.

1.3. Formato de la salida

La salida por pantalla de *pyComp* mostrará un mensaje de éxito si el programa es correcto, o bien un mensaje de error indicando el tipo de error, y el número de línea en el que fue detectado.

Los errores posiblemente devueltos son:

- *Lexical error*: ocurre cuando el compilador se topa con un símbolo desconocido. La lista de símbolos válidos puede ser encontrada en el capítulo correspondiente a **Lenguaje**.
- *Syntactical error*: ocurre cuando el compilador encuentra una sentencia que viola la gramática definida.
- *Semantical error*: ocurre cuando se viola alguna de las reglas semánticas definidas por el compilador; por ejemplo, cuando se referencia a una variable que no fue definida anteriormente, cuando se asignan dos elementos de tipos incompatibles, o cuando se declaran dos variables con el mismo identificador.

Por añadidura, se informará por pantalla en forma de *warning* cuando haya variables, funciones o procedimientos declarados que no hayan sido utilizados.

```
Starting file lexical and syntactical analysis...
The program is syntactically correct.
```

Figura 1.1: Salida por pantalla de un programa sintácticamente correcto

```
Starting file lexical and syntactical analysis...
"bateria\ejemplo10.pas", line 9: Syntactical error found: Expecting "," or ":",
      but ":=" was found
```

Figura 1.2: Salida por pantalla de un programa con un error sintáctico

```
Starting file lexical and syntactical analysis...
"bateria\unusedIdentifier.pas", line 1: WARNING: 'c' has never been initialized
      in program 'unusedidentifier'
The program is syntactically correct.
```

Figura 1.3: Salida por pantalla de un programa correcto con una variable no utilizada

Capítulo 2

Lenguaje

2.1. Símbolos válidos

En esta sección definiremos todos los símbolos válidos en el lenguaje Mini-Pascal. Éstos se dividen en letras, números, y símbolos especiales (dentro de los que se consideran también las palabras reservadas).

```
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
     R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j |
     k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol> ::= + | - | * | = | <> | < | > | <= | >= | ( | ) | [ | ] | { | }
     | := | . | , | ; | : | div | or | and | not | if | then | else | while | do
     | begin | end | const | var | type | array | function | procedure | program
```

Es de notar que cualquier bloque de comentarios (definido por los símbolos { y }, // o bien (* y *)) puede contener cualquier tipo de símbolos, pertenezcan a la gramática o no.

2.2. Identificadores

Un identificador es utilizado para dar nombre a un programa, constante, tipo, variable, función o procedimiento.

```
<identifier> ::= <letter>{<letter or digit>}
```

2.3. Corrección de la gramática

Antes de comenzar con la modificación, se corrigieron los errores marcados por la cátedra.

```
<program> ::= <program heading> <block>.

<program heading> ::= program <identifier>;

<block> ::= <constant definition part><type definition part><variable declaration
     part><procedure and function declaration part><statement part>
```

```
<constant definition part> ::= <empty> | const <constant definition>{;<constant
      definition>};

<constant definition> ::= <identifier>=<constant>

<identifier> ::= <letter>{<letter or digit>}

<letter or digit> ::= <letter> | <digit>

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
      R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j |
      k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<constant> ::= <unsigned number> | <sign><unsigned number> | <constant
      identifier> | <sign><constant identifier> | <char>

<unsigned number> ::= <unsigned integer>

<unsigned integer> ::= <digit sequence>

<digit sequence> ::= <digit>{<digit>}

<sign> ::= + | -

<constant identifier> ::= <identifier>

<type definition part> ::= <empty> | type <type definition>{;<type definition>};

<type definition> ::= <identifier>=<type>

<type> ::= <simple type> | <structured type>

<simple type> ::= <subrange type> | <type identifier>

<subrange type> ::= <constant>..<constant>

<type identifier> ::= <identifier>

<structured type> ::= <unpacked structured type>

<unpacked structured type> ::= <array type>

<array type> ::= array[<index type>] of <component type>

<index type> ::= <simple type>

<component type> ::= <simple type>

<variable definition part> : := <empty> | var<variable declaration>{;<variable
      declaration>};

<variable declaration> ::= <identifier>{,<identifier>} : <type>

<procedure and function declaration part> ::= {<procedure or function declaration
      part>;}

<procedure or function declaration part> ::= <procedure declaration> | <function
      declaration>
```

```
<procedure declaration> ::= <procedure heading><block>

<procedure heading> ::= procedure <identifier>; | procedure <identifier>(<formal
      parameter section>{;<formal parameter section>});

<formal parameter section> ::= <parameter group> | var <parameter group>

<parameter group> ::= <identifier>{,<identifier>}:<type identifier>

<function declaration> ::= <function heading><block>

<function heading> ::= function<identifier>:<result type>; | <function
      identifier>(<formal parameter section>{;<formal parameter
      section>}):<result type>;

<result type> ::= <type identifier>

<statement part> ::= <compound statement>

<compound statement> ::= begin <statement>{;<statement>} end

<statement> ::= <unlabelled statement>

<unlabelled statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <assignment statement> | <procedure statement> | <empty
      statement>

<assignment statement> ::= <variable>:=<expression> | <function
      identifier>:=<expression>

<variable> ::= <entire variable> | <component variable>

<entire variable> ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable> ::= <indexed variable>

<indexed variable> ::= <array variable>[<expression>]

<array variable> ::= <entire variable>

<expression> ::= <simple expression> | <simple expression><relational
      operator><simple expression>

<simple expression> ::= <term> | <simple expression><adding operator><term> |
      <sign><term>

<term>::= <factor> | <term><multiplying operator><factor>

<factor> ::= <variable> | <unsigned constant> | <function designator> |
      (<expression>) | not <factor> | <char>

<char> ::= '<letter>' | '<digit>'

<unsigned constant> ::= <unsigned number> | <constant identifier>

<function designator> ::= <function identifier> | <function identifier>(<actual
      parameter>{,<actual parameter>})
```

```
<function identifier> ::= <identifier>

<actual parameter> ::= <expression> | <variable>

<multiplying operator> ::= * | div | and

<adding operator> ::= + | - | or

<relational operator> ::= = | <> | < | <= | >= | >

<procedure statement> ::= <procedure identifier> | <procedure identifier>(<actual
      parameter>{,<actual parameter>})

<procedure identifier> ::= <identifier>

<empty statement> ::= <empty>

<structured statement> ::= <compound statement> | <conditional statement> |
      <repetitive statement>

<conditional statement> ::= <if statement>

<if statement> ::= if <expression> then <statement> | if <expression> then
      <statement> else <statement>

<repetitive statement> ::= <while statement>

<while statement> ::= while <expression> do <statement>

<special symbol> ::= + | - | * | = | <> | < | > | <= | >= | ( | ) | [ | ] | { | }
      | := | . | , | ; | : | div | or | and | not | if | then | else | while | do
      | begin | end | const | var | type | array | function | procedure | program
```

2.4. Pasaje a notación BNF

Como siguiente paso en la adaptación de la gramática, se reemplazaron los terminales por los tokens que devuelve **LexAn**. Para esto, se adoptó la convención de dejar los no terminales en minúscula, mientras que los tokens (terminales) se encuentran en MAYÚSCULA. Luego, se eliminaron las extensiones propias de la notación EBNF.

```
<program> ::= <program_heading> <block> <END_PROGRAM>

<program_heading> ::= <PROGRAM> <IDENTIFIER> <SEMI_COLON>

<block> ::= <constant_definition_part> <block_cons_rest> | <block_cons_rest>

<block_cons_rest> ::= <type_definition_part> <block_type_rest> |
      <block_type_rest>

<block_type_rest> ::= <variable_definition_part> <block_var_rest> |
      <block_var_rest>

<block_var_rest> ::= <procedure_and_function_declaration_part> <statement_part> |
      <statement_part>
```

```
<constant_definition_part> ::= <CONST> <constant_definition>
      <constant_definition_rest>

<constant_definition_rest> ::= <SEMI_COLON> <constant_definition_rest_rest>

<constant_definition_rest_rest> ::= <constant_definition>
      <constant_definition_rest> | <LAMBDA>

<constant_definition> ::= <IDENTIFIER> <EQUAL> <constant>

<constant> ::= <NUMBER> | <IDENTIFIER> | <CHAR> | <sign> <constant_rest>

<constant_rest> ::= <NUMBER> | <IDENTIFIER>

<sign> ::= <ADD_OP> | <MINUS_OP>

<type_definition_part> ::= <TYPE> <type_definition> <type_definition_rest>

<type_definition_rest> ::= <SEMI_COLON> <type_definition_rest_rest>

<type_definition_rest_rest> ::= <type_definition> <type_definition_rest> |
      <LAMBDA>

<type_definition> ::= <IDENTIFIER> <EQUAL> <type>

<type> ::= <simple_type> | <structured_type>

<simple_type> ::= <NUMBER> <SUBRANGE_SEPARATOR> <constant> | <CHAR>
      <SUBRANGE_SEPARATOR> <constant> | <sign> <subrange_type_rest> |
      <IDENTIFIER> <simple_type_rest>

<simple_type_rest> ::= <SUBRANGE_SEPARATOR> <constant> | <LAMBDA>

<subrange_type_rest> ::= <NUMBER> <SUBRANGE_SEPARATOR> <constant> | <IDENTIFIER>
      <SUBRANGE_SEPARATOR> <constant>

<structured_type> ::= <ARRAY> <OPEN_BRACKET> <simple_type> <CLOSE_BRACKET> <OF>
      <simple_type>

<variable_definition_part> ::= <VAR> <variable_declaration>
      <variable_declaration_part_rest>

<variable_declaration_part_rest> ::= <SEMI_COLON>
      <variable_declaration_rest_rest>

<variable_declaration_rest_rest> ::= <variable_declaration>
      <variable_declaration_part_rest> | <LAMBDA>

<variable_declaration> ::= <IDENTIFIER> <variable_declaration_rest>

<variable_declaration_rest> ::= <COMMA> <IDENTIFIER> <variable_declaration_rest>
      | <TYPE_DECLARATION> <type>

<procedure_and_function_declaration_part> ::=
      <procedure_or_function_declaration_part> <SEMI_COLON>
      <procedure_and_function_declaration_part> | <LAMBDA>

<procedure_or_function_declaration_part> ::= <procedure_declaration> |
      <function_declaration>
```

```
<procedure_declaration> ::= <procedure_heading> <block>

<procedure_heading> ::= <PROCEDURE> <IDENTIFIER> <procedure_heading_rest>

<procedure_heading_rest> ::= <SEMI_COLON> | <OPEN_PARENTHESIS>
      <formal_parameter_section> <formal_parameter_rest>

<formal_parameter_rest> ::= <SEMI_COLON> <formal_parameter_section>
      <formal_parameter_rest> | <CLOSE_PARENTHESIS> <SEMI_COLON>

<formal_parameter_section> ::= <parameter_group> | <VAR> <parameter_group>

<parameter_group> ::= <IDENTIFIER> <parameter_group_rest>

<parameter_group_rest> ::= <COMMA> <IDENTIFIER> <parameter_group_rest> |
      <TYPE_DECLARATION> <IDENTIFIER>

<function_declaration> ::= <function_heading> <block>

<function_heading> ::= <FUNCTION> <IDENTIFIER> <function_heading_rest>

<function_heading_rest> ::= <TYPE_DECLARATION> <IDENTIFIER> <SEMI_COLON> |
      <OPEN_PARENTHESIS> <formal_parameter_section>
      <formal_parameter_function_rest>

<formal_parameter_function_rest> ::= <SEMI_COLON> <formal_parameter_section>
      <formal_parameter_function_rest> | <CLOSE_PARENTHESIS> <TYPE_DECLARATION>
      <IDENTIFIER> <SEMI_COLON>

<statement_part> ::= <compound_statement>

<compound_statement> ::= <BEGIN> <statement> <compound_statement_rest> <END>

<compound_statement_rest> ::= <SEMI_COLON> <statement> <compound_statement_rest>
      | <LAMBDA>

<statement> ::= <simple_statement> | <structured_statement>

<simple_statement> ::= <IDENTIFIER> <simple_statement_rest>

<simple_statement_rest> ::= <ASSIGNMENT> <expression> | <OPEN_BRACKET>
      <expression> <CLOSE_BRACKET> <ASSIGNMENT> <expression> | <OPEN_PARENTHESIS>
      <actual_parameter> <actual_parameter_rest> | <LAMBDA>

<component_variable> ::= <IDENTIFIER> <OPEN_BRACKET> <expression> <CLOSE_BRACKET>

<expression> ::= <simple_expression> <expression_rest>

<expression_rest> ::= <relational_operator> <simple_expression> | <LAMBDA>

<simple_expression> ::= <term> <simple_expression_other> | <sign> <term>
      <simple_expression_other>

<simple_expression_other> ::= <adding_operator> <term> <simple_expression_other>
      | <LAMBDA>

<term> ::= <factor> <term_other>

<term_other> ::= <multiplying_operator> <factor> <term_other> | <LAMBDA>
```

```
<factor> ::= <IDENTIFIER> <factor_rest> | <NUMBER> | <OPEN_PARENTHESIS>
      <expression> <CLOSE_PARENTHESIS> | <NOT_LOGOP> <factor> | <CHAR>

<factor_rest> ::= <OPEN_BRACKET> <expression> <CLOSE_BRACKET> |
      <OPEN_PARENTHESIS> <actual_parameter> <actual_parameter_rest> | <LAMBDA>

<actual_parameter> ::= <expression>

<actual_parameter_rest> ::= <COMMA> <actual_parameter> <actual_parameter_rest> |
      <CLOSE_PARENTHESIS>

<multiplying_operator> ::= <MULTIPLY_OP> | <DIV_OP> | <AND_LOGOP>

<adding_operator> ::= <ADD_OP> | <MINUS_OP> | <OR_LOGOP>

<relational_operator> ::= <LESS_OP> | <LESS_EQUAL_OP> | <GREATER_OP> |
      <GREATER_EQUAL_OP> | <NOT_EQUAL_OP> | <EQUAL>

<procedure_statement> ::= <IDENTIFIER> <procedure_statement_rest>

<procedure_statement_rest> ::= <OPEN_PARENTHESIS> <actual_parameter>
      <actual_parameter_rest> | <LAMBDA>

<structured_statement> ::= <compound_statement> | <conditional_statement> |
      <repetitive_statement>

<conditional_statement> ::= <IF> <expression> <THEN> <statement>
      <conditional_statement_rest>

<conditional_statement_other> ::= <ELSE> <statement> | <LAMBDA>

<repetitive_statement> ::= <WHILE> <expression> <DO> <repetitive_statement_rest>

<repetitive_statement_rest> ::= <statement> | <LAMBDA>
```

2.5. Gramática final

Para llegar a la gramática utilizada para implementar el analizador sintáctico, se realizaron varios pasos:

1. **Eliminar Ambiguedad**: ésta tal vez sea la afirmación más peligrosa, ya que no se puede saber si una gramática es ambigua o no. De cualquier manera, se eliminaron todas las ambiguedades que se encontraron, salvo el caso del `if then else`, del cual se hablará más adelante.
2. **Eliminar Recursión a Izquierda**: Se utilizó el algoritmo explicado en [1, pág. 212].
3. **Factorizar a Izquierda**: Se utilizó el algoritmo explicado en el mismo libro, en la página 214.

Luego de esta serie de pasos, se llegó a la siguiente gramática:

```
<program> ::= <program_heading> <block> <END_PROGRAM> <EOF>

<program_heading> ::= <PROGRAM> <IDENTIFIER> <SEMI_COLON>

<block> ::= <constant_definition_part> <block_cons_rest> | <block_cons_rest>
```

```
<block_cons_rest> ::= <type_definition_part> <block_type_rest> |
      <block_type_rest>

<block_type_rest> ::= <variable_definition_part> <block_var_rest> |
      <block_var_rest>

<block_var_rest> ::= <procedure_and_function_declaration_part> <statement_part> |
      <statement_part>

<constant_definition_part> ::= <CONST> <constant_definition>
      <constant_definition_rest>

<constant_definition_rest> ::= <SEMI_COLON> <constant_definition_rest_rest>

<constant_definition_rest_rest> ::= <constant_definition>
      <constant_definition_rest> | <LAMBDA>

<constant_definition> ::= <IDENTIFIER> <EQUAL> <constant>

<constant> ::= <NUMBER> | <IDENTIFIER> | <CHAR> | <sign> <constant_rest>

<constant_rest> ::= <NUMBER> | <IDENTIFIER>

<sign> ::= <ADD_OP> | <MINUS_OP>

<type_definition_part> ::= <TYPE> <type_definition> <type_definition_rest>

<type_definition_rest> ::= <SEMI_COLON> <type_definition_rest_rest>

<type_definition_rest_rest> ::= <type_definition> <type_definition_rest> |
      <LAMBDA>

<type_definition> ::= <IDENTIFIER> <EQUAL> <type>

<type> ::= <simple_type> | <structured_type>

<simple_type> ::= <NUMBER> <SUBRANGE_SEPARATOR> <constant> | <CHAR>
      <SUBRANGE_SEPARATOR> <constant> | <sign> <subrange_type_rest> |
      <IDENTIFIER> <simple_type_rest>

<simple_type_rest> ::= <SUBRANGE_SEPARATOR> <constant> | <LAMBDA>

<subrange_type_rest> ::= <NUMBER> <SUBRANGE_SEPARATOR> <constant> | <IDENTIFIER>
      <SUBRANGE_SEPARATOR> <constant>

<structured_type> ::= <ARRAY> <OPEN_BRACKET> <simple_type> <CLOSE_BRACKET> <OF>
      <simple_type>

<variable_definition_part> ::= <VAR> <variable_declaration>
      <variable_declaration_part_rest>

<variable_declaration_part_rest> ::= <SEMI_COLON>
      <variable_declaration_rest_rest>

<variable_declaration_rest_rest> ::= <variable_declaration>
      <variable_declaration_part_rest> | <LAMBDA>

<variable_declaration> ::= <IDENTIFIER> <variable_declaration_rest>
```

```
<variable_declaration_rest> ::= <COMMA> <IDENTIFIER> <variable_declaration_rest>
      | <TYPE_DECLARATION> <type>

<procedure_and_function_declaration_part> ::=
      <procedure_or_function_declaration_part> <SEMI_COLON>
      <procedure_and_function_declaration_part> | <LAMBDA>

<procedure_or_function_declaration_part> ::= <procedure_declaration> |
      <function_declaration>

<procedure_declaration> ::= <procedure_heading> <block>

<procedure_heading> ::= <PROCEDURE> <IDENTIFIER> <procedure_heading_rest>

<procedure_heading_rest> ::= <SEMI_COLON> | <OPEN_PARENTHESIS>
      <formal_parameter_section> <formal_parameter_rest>

<formal_parameter_rest> ::= <SEMI_COLON> <formal_parameter_section>
      <formal_parameter_rest> | <CLOSE_PARENTHESIS> <SEMI_COLON>

<formal_parameter_section> ::= <parameter_group> | <VAR> <parameter_group>

<parameter_group> ::= <IDENTIFIER> <parameter_group_rest>

<parameter_group_rest> ::= <COMMA> <IDENTIFIER> <parameter_group_rest> |
      <TYPE_DECLARATION> <IDENTIFIER>

<function_declaration> ::= <function_heading> <block>

<function_heading> ::= <FUNCTION> <IDENTIFIER> <function_heading_rest>

<function_heading_rest> ::= <TYPE_DECLARATION> <IDENTIFIER> <SEMI_COLON> |
      <OPEN_PARENTHESIS> <formal_parameter_section>
      <formal_parameter_function_rest>

<formal_parameter_function_rest> ::= <SEMI_COLON> <formal_parameter_section>
      <formal_parameter_function_rest> | <CLOSE_PARENTHESIS> <TYPE_DECLARATION>
      <IDENTIFIER> <SEMI_COLON>

<statement_part> ::= <compound_statement>

<compound_statement> ::= <BEGIN> <statement> <statement_part_rest> <END>

<statement_part_rest> ::= <SEMI_COLON> <statement> <statement_part_rest> |
      <LAMBDA>

<statement> ::= <simple_statement> | <structured_statement>

<simple_statement> ::= <IDENTIFIER> <simple_statement_rest> | <LAMBDA>

<simple_statement_rest> ::= <ASSIGNMENT> <expression> | <OPEN_BRACKET>
      <expression> <CLOSE_BRACKET> <ASSIGNMENT> <expression> | <OPEN_PARENTHESIS>
      <actual_parameter> <actual_parameter_rest> | <LAMBDA>

<component_variable> ::= <IDENTIFIER> <OPEN_BRACKET> <expression> <CLOSE_BRACKET>

<expression> ::= <simple_expression> <expression_rest>

<expression_rest> ::= <relational_operator> <simple_expression> | <LAMBDA>
```

```
<simple_expression> ::= <term> <simple_expression_other>

<simple_expression_other> ::= <adding_operator> <term> <simple_expression_other>
      | <LAMBDA>

<term> ::= <factor> <term_other>

<term_other> ::= <multiplying_operator> <factor> <term_other> | <LAMBDA>

<factor> ::= <IDENTIFIER> <factor_rest> | <NUMBER> | <OPEN_PARENTHESIS>
      <expression> <CLOSE_PARENTHESIS> | <NOT_LOGOP> <factor> | <CHAR> | <sign>
      <factor>

<factor_rest> ::= <OPEN_BRACKET> <expression> <CLOSE_BRACKET> |
      <OPEN_PARENTHESIS> <actual_parameter> <actual_parameter_rest> | <LAMBDA>

<actual_parameter> ::= <expression>

<actual_parameter_rest> ::= <COMMA> <actual_parameter> <actual_parameter_rest> |
      <CLOSE_PARENTHESIS>

<multiplying_operator> ::= <MULTIPLY_OP> | <DIV_OP> | <AND_LOGOP>

<adding_operator> ::= <ADD_OP> | <MINUS_OP> | <OR_LOGOP>

<relational_operator> ::= <LESS_OP> | <LESS_EQUAL_OP> | <GREATER_OP> |
      <GREATER_EQUAL_OP> | <NOT_EQUAL_OP> | <EQUAL>

<structured_statement> ::= <compound_statement> | <conditional_statement> |
      <repetitive_statement>

<conditional_statement> ::= <IF> <expression> <THEN> <statement>
      <conditional_statement_other>

<conditional_statement_other> ::= <ELSE> <statement> | <LAMBDA>

<repetitive_statement> ::= <WHILE> <expression> <DO> <statement>
```

2.6. ¿Es LL(1)?

La gramática pasó por todos los pasos especificados, en un intento de lograr una gramática para ser utilizada como base de la implementación de un Analizador Sintáctico Descendente Predictivo Recursivo. Cada regla tiene conjuntos disjuntos de la función PRIMERO para cada producción, por lo que el analizador puede saber que producción optar, leyendo un token. Por esto, se podría decir que la gramática obtenida es LL(1).

Sin embargo, como fue dicho en la sección anterior, la gramática contiene por lo menos una ambigüedad: la del `if then else`. La cadena de tokens `<IF> ... <THEN> <IF> ... <THEN> ... <ELSE> ...` (donde los puntos suspensivos representan un grupo de statements válidos) tiene dos árboles de derivación posibles.

Una solución que se puede adoptar es la de utilizar precedencia, donde el `<ELSE>` quedará ligado al `<IF> ... <THEN>` sin `<ELSE>` más cercano. Ésta solución será la que se adopte para la implementación del compilador de MINI-PASCAL, y se llevará a cabo en la etapa de análisis semántico.

Se puede concluir entonces que la gramática no es LL(1), pero de cualquier manera, sirve para realizar el Analizador Sintáctico.

Capítulo 3

Errores detectados

Dado las características del Analizador Sintáctico solicitado por la cátedra, los errores detectados son disparados cuando se encuentra un *token* distinto al que se esperaba. Por esto, el mensaje impreso por pantalla (o en el archivo de salida) muestra el dicho *token*.

Muchas veces, está información no es suficiente para el programador, por lo que, en la mayoría de los posibles errores, también se detalla cuál era el token (o los tokens) esperados, como se muestra en la figura 1.2.

```
Starting file lexical and syntactical analysis...
"bateria\ejemplo5.pas", line 22: Unexpected token: ")" found.
```

Figura 3.1: Error posible: Token inesperado

Bibliografía

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman *Compilers: principles, techniques, and tools*. Addison Wesley 2nd Edition 2007.

[2] Cátedra de Compiladores e Intérpretes, DCIC, UNS *Proyecto Nº1: Compilador de Mini-Pascal-S* 2010

[3] Cátedra de Compiladores e Intérpretes, DCIC, UNS *Consideraciones Generales para la 3er entrega del Proyecto Nº1 - Analizador Sintáctico* 2010