Universidad de La Laguna

E. T. S. DE INGENIERÍA INFORMÁTICA
Departamento de E.I.O. y Computación
Compiladores

PRACTICA 2: Diseño de un Analizador Sintáctico descendente recursivo predictivo para *Pascal-*

Factor de ponderación [0-10]: 7

2.1. Objetivos

La práctica consiste en la implementación en Pascal de un analizador sintáctico descendente recursivo predictivo para *Pascal-*.

Dicho analizador ha de reconocer las sentencias correctas y detectar errores sintácticos. Ante un error sintáctico, el analizador debería notificar el error y finalizar su ejecución.

2.2. Descripción

La gramática de *Pascal-* en notación EBNF es la siguiente:

1. <Program> ::= **program program_name ;** <Block_body> **.**
2. <Block_body> ::= [<Constant_definition_part>] [<Type_definition_part>] [<Variable_definition_part>] {<Procedure_definition>}<Compound_statement>
3. <Constant_definition_part> ::= **const** <Constant_definition> {<Constant_definition>}
4. <Constant_definition> ::= **constant_name** = <Constant> **;**
5. <Type_definition_part> ::= **type** <Type_definition> {<Type_definition>}
6. <Type_definition> ::= **type_name =** <New_type> **;**
7. <New_type> ::= <New_array_type> | <New_record_type>
8. <New_array_type> ::= **array** "**[**"<Index_range> "**]**"**of type_name**
9. <Index_range> ::= <Constant> **..** <Constant>
10. <New_record_type> ::= **record** <Field_list> **end**
11. <Field_list> ::= <Record_section> {**;** <Record_section>}
12. <Record_section> ::= **field_name** {**, field_name**} : **type_name**
13. <Variable_definition_part> ::= **var** <Variable_definition> {<Variable_definition>}
14. <Variable_definition> ::= <Variable_group> **;**
15. <Variable_group> ::= **variable_name** {**, variable_name**} **: type_name**
16. <Procedure_definition> ::= **procedure procedure_name** <Procedure_block> **;**

17. <Procedure_block> ::= [**(** <Formal_parameter_list> **)**] **;** <Block_body>
18. <Formal_parameter_list> ::= <Parameter_definition> {**;** <Parameter_definition>}
19. <Parameter_definition> ::= [**var**] <Variable_group>
20. <Statement> ::= <Assignment_statement> | <Procedure_statement> | <If_statement> | <While_statement> | <Compound_statement> |ϵ
21. <Assignment_statement> ::= <Variable_access> **:=** <Expression>
22. <Procedure_statement> ::= **procedure_name** [**(** <Actual_parameter_list> **)**]
23. <Actual_parameter_list> ::= <Actual_parameter> { **,** <Actual_parameter>}
24. <Actual_parameter> ::= <Expression> | <Variable_access>
25. <If_statement> ::= **if** <Expression> **then** <Statement> [**else** <Statement>]
26. <While_statement> ::= **while** <Expression> **do** <Statement>
27. <Compound_statement> ::= **begin** <Statement> {**;** <Statement>} **end**
28. <Expression> ::= <Simple_expression> [<Relational_operator> <Simple_expression>]
29. <Relational_operator> ::= < | = | > | <= | <> | >=
30. <Simple_expression> ::= [<Sign_operator>] <Term> {<Additive_operator> <Term>}
31. <Sign_operator> ::= **+** | **-**
32. <Additive_operator> ::= **+** | **-** | **or**
33. <Term> ::= <Factor> {<Multiplying_operator> <Factor>}
34. <Multiplying_operator> ::= ***** | **div** | **mod** | **and**
35. <Factor> ::= <Constant> | <Variable_access> | **(** <Expression> **)** | **not** <Factor>
36. <Variable_access> ::= **variable_name** {<Selector>}
37. <Selector> ::= <Index_selector> | <Field_selector>
38. <Index_selector> ::= "["<Expression> "]"
39. <Field_Selector> ::= **. field_name**
40. <Constant> ::= **Numeral** | **constant_name**

Los siguientes elementos léxicos (tokens) **program_name**, **constant_name**, **type_name**, **field_name**, **variable_name** y **procedure_name** corresponden todos con el token `ID` del analizador léxico (se les ha puesto un nombre específico, para facilitar su comprensión).

Los símbolos de corchete abierto y cerrado ([y]) indican opcionalidad (como es habitual en EBNF) salvo cuando aparecen entre comillas dobles ("[", "]") cuando hacen referencia a los tokens LEFTBRACKET y RIGHTBRACKET.

El analizador léxico devuelve el token `NUMERAL` si reconoce en el programa fuente un lexema representado por la expresión regular: `digito+`. Análogamente, devuelve `ID` si reconoce un identificador representado por la expresión regular: `letra(letra | digito)*`.

La función que realiza el análisis léxico de la entrada es:

```
token yylex(argtype *argvalue);
```

Algunos de los tokens tienen asociado un atributo. En nuestro caso:

`ID` ptr

`NUMERAL` Value

El token `ID` tiene asociado un atributo en el que se almacena un puntero al lexema que da lugar al identificador. El token `NUMERAL` tiene asociado un argumento de tipo entero en el que se devuelve el valor del entero que se ha reconocido.

El token devuelto en cada momento por el analizador léxico se almacenará en la variable global *lookahead* (símbolo de preanálisis). Si es un token que tiene asociado un atributo, su valor se almacenará según el caso en el campo correspondiente de `argvalue` (para el token `NUMERAL` en `argvalue.value` o en `argvalue.ptr` para el token `ID`).

```
token lookahead;
argtype argvalue;
```

Se usan también conjuntos de tokens para clasificarlos y escribir los conjuntos *FIRST* de las diferentes cadenas.

Se usará el token `ENDTEXT` para indicar el final del programa fuente y el análisis sintáctico debe detenerse cuando el analizador léxico lo devuelva.

En primer lugar se han de calcular los conjuntos *FIRST*, *FOLLOW* para todos los símbolos gramaticales y el conjunto de *PREDICCIÓN* para cada una de las reglas de producción de la gramática.

Antes de proceder a escribir el analizador sintáctico, se debe comprobar que la gramática es apta para realizar con ella un análisis sintáctico descendente recursivo.

Estudiar si se cumplen las condiciones para ello:

1. Considérense todas las producciones de la forma:
   $N ::= \alpha|\beta$
   utilizando el símbolo de preanálisis (*lookahead*) se ha de decidir si se utiliza α o β; por tanto se ha de verificar que las dos alternativas de la producción comienzan con símbolos diferentes, esto es
   $PRED(\alpha) \cap PRED(\beta) = \emptyset$

Realizar los cambios pertinentes en la gramática para que se cumpla la condición anterior.

El analizador sintáctico se puede construir directamente de la gramática siguiendo las siguientes reglas:

Regla 2.2.1 *Para cada regla de producción de la gramática: $N \rightarrow \alpha$ el analizador sintáctico define un procedimiento con el mismo nombre:*

```
procedure N;
begin
  A(alpha)
end;
```

El procedimiento define un algoritmo de análisis A(α). Dicho algoritmo A examina uno o más símbolos y determina si forman una frase descrita por la expresión sintáctica α. Si ello es cierto, el algoritmo lee toda la frase y el primer símbolo que la siga; en otro caso genera un error sintáctico después de leer un número indeterminado de símbolos.

Regla 2.2.2 *Una lista de sentencias de la forma $S_1 S_2 \ldots S_n$ se reconoce mediante el análisis individual de cada una de ellas en el mismo orden en el que están escritas. Esto es, $A(S_1 S_2 \ldots S_n) = A(S_1); A(S_2); \ldots; A(S_n);$*

Regla 2.2.3 *Cuando el analizador sintáctico espera un único token tok, utiliza el siguiente algoritmo:*

```
A(tok):

if (lookahead == tok) then
  lookahead := yylex(argvalue)
else
  syntax_error();
```

Regla 2.2.4 *Para reconocer un lado derecho en el que aparece un no terminal N, el analizador sintáctico invoca al correspondiente procedimiento llamado N.*
$A(N) = N();$

Regla 2.2.5 *Para reconocer una producción descrita con una expresión de la forma $[\alpha]$ se utiliza el siguiente algoritmo:*

```
A([alpha]):

if lookahead in FIRST(alpha) then
  A(alpha)
```

Regla 2.2.6 *Para reconocer una producción descrita con una expresión de la forma α usar el siguiente algoritmo:*

```
A({alpha}):

while lookahead in FIRST(alpha) do
  A(alpha)
```

Regla 2.2.7 *Si todas las cadenas de la forma $\alpha_1, \alpha_2, \ldots, \alpha_n$ son distintas de ϵ el siguiente algoritmo reconoce una producción de la forma $\alpha_1|\alpha_2|\ldots,|\alpha_n$*

```
A(alpha1 | alpha2 | ... | alphan) :

  if lookahead in FIRST(alpha1) then A(alpha1)
  else if lookahead in FIRST(alpha2) then A(alpha2)
  ...
  else if lookahead in FIRST(alphan) then A(alphan)
  else Syntax_error();
```

Si cualquiera de las alternativas α_i puede derivar en la cadena ϵ el mensaje de error del final se omite.

El programa principal después de inicializar las variables, lee el primer token e invoca a la función $Program()$ (asociada al símbolo de arranque de la gramática) para analizar la sintaxis del programa de entrada:

```
begin
...
Initialize();
lookahead = yylex(argvalue);
Program();
end.
```

Notas: Junto con el código correspondiente al analizador sintáctico, los alumnos deben llevar escrito en papel el estudio realizado respecto a las condiciones que debe verificar la gramática (cálculo de First y Follow y la verificación de las condiciones) así como la gramática con las modificaciones que se le hayan realizado.