Universidad de La Laguna

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

PRACTICA 5: Análisis sintáctico descendente predictivo no recursivo

Factor de ponderación [0-10]: 8

5.1. Objetivos

La práctica consiste en la implementación de un analizador sintáctico descendente predictivo no recursivo para *Pascal-*.

5.2. 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. <FielSelector> ::= **. 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.

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 que encaja con la expresión regular: Digit (Digit)*. Análogamente, devuelve `ID` si reconoce un identificador definido por la expresión regular: Letter (Letter | Digit)*.

El analizador ha de reconocer las sentencias correctas y detectar errores sintácticos.

Una vez transformada la gramática si ello resulta necesario, se calcularán los conjuntos *First*, *Follow* y de *Predicción* para comprobar que la gramática es LL(1).

Se implementará un analizador sintáctico descendente predictivo y no recursivo (guiado por la tabla LL(1)). L@s alumn@s deben decidir cómo representar los símbolos gramaticales (no terminales y tokens) para construir la tabla LL(1), y también cómo representar la tabla en sí. Como siempre, pero incluso más en este caso, merece la pena dedicar tiempo a pensar en la implementación antes de lanzarse a codificar la primera idea que se nos ocurra.

El programa no construirá la tabla LL(1) partiendo de la gramática, sino que la tabla se construirá “a mano” y se codificará (con la representación elegida) en el analizador.

El algoritmo de análisis a utilizar será el estudiado en clase para este tipo de análisis, utilizando la tabla LL(1) y una pila.

El analizador deberá indicar simplemente si el programa es sintácticamente correcto o no. Deberá permitir también la opción de que el analizador escriba en pantalla la producción gramatical que ha utilizado en cada paso, imprimiendo una producción por línea, de forma que se pueda estudiar la derivación que el analizador construye. Esta opción se implementará mediante la utilización en la llamada de un parámetro en línea de comandos que indique si se desea imprimir en pantalla la derivación utilizada o no.

Además, si el programa fuente contiene algún error sintáctico, el analizador debe producir un mensaje de error (lo más explicativo posible) y finalizar la ejecución.

5.3. Notas

Junto con el código correspondiente al analizador sintáctico, los alumnos deben llevar a la corrección de la práctica la gramática en que se han basado, la tabla LL(1) que han diseñado, el cálculo de los conjuntos First, Follow y de Predicción que han utilizado y al menos tres programas con los que verificar el comportamiento del analizador. Se valorará positivamente que los programas de ejemplo tengan algún significado (efectivamente computen algo).