Paradigmas de la Programación
Práctico 1: Sintaxis y Gramáticas

Laura Alonso Alemany Ezequiel Orbe

9 de marzo de 2015

Seleccione la(s) respuesta(s) correcta

1. Una gramática describe...
   a) lo que significan las sentencias de un lenguaje de programación.
   b) las sentencias que forman parte de un lenguaje de programación.
   c) cómo se pueden escribir sentencias en un lenguaje de programación.
   d) el alfabeto de un lenguaje de programación.
2. Una gramática independiente de contexto (*context-free grammar*)...
   a) tiene predicados.
   b) es inambigua.
   c) tiene reglas de reescritura.
   d) describe la sintaxis de un lenguaje.
   e) es más expresiva que una gramática dependiente del contexto.
   f) es tan expresiva como una máquina de Turing.
3. ¿Se puede expresar con una gramática libre de contexto la obligación de de clarar una variable antes de que se use?
   a) Verdadero.
   b) Falso.
4. ¿Se puede expresar con una gramática libre de contexto la declaración de v ariables con tipo?
   a) Verdadero.
   b) Falso.
5. ¿Se puede expresar con una gramática libre de contexto la asignación múltiple de variables?
   a) No.
   b) Sí, si se trata de de asignación en secuencia, y no en espejo.
   c) Sí, si se trata de un número finito de variables.
   d) Todas las anteriores.
   e) Ninguna de las anteriores.
6. Una gramática que puede ofrecer más de un árbol de derivación para una misma expresión es una gramática...
   a) Incorrecta.

b) Ambigua.

c) Dependiente de contexto.

d) Independiente de contexto.

7. ¿Dada una gramática EBNF, ¿cómo se puede probar si una secuencia de tokens es válida?

   a) Si existe una secuencia de reglas de la gramática que puedan aplicarse sucesivamente desde el símbolo terminal de forma que su resultado sea la secuencia de tokens.

   b) Si los tokens pertenecen al alfabeto de la gramática.

   c) Si podemos dar un árbol de derivación de la secuencia.

   d) Si la secuencia pertenece al lenguaje de la gramática.

Ejercicios prácticos

1. Escriba una EBNF aumentada con los predicados necesarios para poder expresar:

   - que una variable debe ser declarada antes de ser usada.
   - asignación múltiple de variables, con un número arbitrario de variables.

2. De la misma forma que obtuvimos la derivación de una expresión algebraica usando la gramática de dígitos, use la siguiente gramática:

```
program = 'PROGRAM', espacio, identificador, espacio,
          'BEGIN', espacio,
          { asignacion, ";", espacio },
          'END.' ;
identificador = caracter_alfabetico, { caracter_alfabetico | digito } ;
numero = [ "-" ], digito, { digito } ;
string = '"', { todos_caracteres - '"' }, '"' ;
asignacion = identificador , ":=" , ( numero | identificador | string ) ;
caracter_alfabetico = "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" ;
digito = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
espacio = ? caracteres_espacio ? ;
all caracteres = ? todos_caracteres_visibles ? ;
```

   Para obtener la derivación de la siguiente expresión:

```
PROGRAM DEMO1
BEGIN
  A:=3;
  TEXT:="Hello world!";
END.
```

3. Muestre con un ejemplo que la siguiente gramática es ambigua, y modifíquela para que se convierta en una gramática inambigua. ¿Qué estrategias puede usar para hacerlo?

```
<bloque> ::= { <sentencias> }
<sentencias> ::= <sentencias> <sentencia> | <sentencia>
<sentencia> ::= <asignacion> | <sentenciaIf> | <bloque>
<sentenciaIf> ::= if ( <expresion> ) <sentencia> |
                  if ( <expresion> ) <sentencia> else <sentencia>
<asignacion> ::= <variable> := <expresion>
```

```
<expresion> ::= <variable> | <variable> <operador> <variable>
<operador> ::= + | - | * | / | < | > | 'and' | 'or'
<variable> ::= x | y | z
```

4. Algunos lenguajes de programación, como Python, usan la tabulación para definir el alcance de las variables y los finales de bloque, como en el siguiente ejemplo:

```
a = "mi vieja mula ya no es lo que era"
b = 0
while b <= 3:
    print a
    b = b+1
```

¿Cómo sería el tokenizer para reconocer las expresiones de un lenguaje de programación así? Escriba una gramática EBNF para este lenguaje.

5. Un programa Lisp está compuesto por una sucesión de expresiones simbólicas denominadas S-expressions o sexp. Éstas se definen formalmente como sigue:

```
<s_exp> ::= <atomic_symbol> | <natural> | "(" <s_exp> "." <s_exp> ")"
<atomic_symbol> ::= <letter> <atom_part> | ":" <atom_part>
<natural> ::= {<number>}+
<atom_part> ::= <empty> | <letter> <atom_part> | <number> <atom_part>
<letter> ::= "a" | "b" | ... | "z"
<number> ::= "1" | "2" | ... | "9"
<empty> ::= ""
```

Determine cuáles de los siguientes son programas sintácticamente correctos. Derive, para los que efectivamente sean programas Lisp, el árbol sintáctico correspondiente.

a) `2 .`

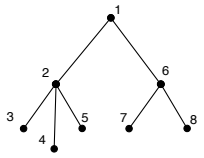
b) `a ( b )`

c) `(3 . 3)`

d) `(3 2 1)`

6. En Lisp un dotted pair es un par escrito con un punto (.) separando ambas partes del mismo, ej: `(3 . 3)`. Por su parte, una lista de Lisp es una concatenación de dotted pairs que además termina en el átomo `nil`. Por ejemplo `(3 . (2 . (1 . nil)))` es una lista con los elementos 3, 2 y 1. Una "syntax sugar" para esta lista es `(3 2 1)`. Modifique la gramática de arriba para que admita esta forma sintáctica.

7. Un árbol es una estructura de datos que está formada por nodos, los cuales tienen asociado un valor y pueden tener cero o más nodos hijos conectados. Es posible implementar dicha estructura utilizando solamente listas, de la siguiente forma:




```
[1 [[2 [[3] [4] [5]]] [6 [[7] [8]]]]]
```

Escriba una EBNF que formalice la estructura de datos descripta.