DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 1

6 de julio de 2016 — Teoría de Lenguajes

Grupo n_n

Integrante	LU	Correo electrónico
Gasco, Emilio		gascoe@gmail.com
Gatti, Mathias	477/14	mathigatti@gmail.com
Patané, Federico		fede_river_8e@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

0. Introducción

En el presente trabajo práctico hemos llevado a cabo la construcción por completo de un parser. La problemática planteada por la cátedra ha sido la de poder, a partir de un texto de entrada con líneas de código del lenguaje descripto por el Trabajo Práctico, poder si este era tanto sintáctica como semánticamente válido, darle una tabulación correcta según términos de legibilidad conocidos por los programadores.

Para realizar este trabajo, tuvimos que dividir nuestras tareas en varias etapas. La primer etapa, es lograr conseguir los tokens de nuestra entrada, que es tener una lista con todos los símbolos que vinieron en las líneas. Para poder realizar esto, hizo falta crear un lexer, que es un tokenizador, que a través de reglas sabrá dividir la entrada en los tokens, dandole un significado a cada uno de ellos para que luego podamos parsear a los mismos.

Cabe destacar que para la realización de este lexer, decidimos utilizar la herramienta de ply, del lenguaje de programación Python.

Luego de realizada esta etapa, deberemos crear una gramática que pueda parsear correctamente cualquier entrada esperada de nuestro lenguaje. Muy importante destacar la necesidad de tener una gramática no ambigua, para que en el parseo posterior se genere solo un árbol sintáctico, lo cuál nos dará la posibilidad de utilizar el parser que la herramienta ply nos brinda, ya que trabaja con un parser LALR, que es un tipo de parser ascendente, osea que genera este árbol desde las hojas hacia la raìz.

Una vez parseado esto, y generado el árbol de derivación debemos darle semántica a las producciones, para poder hacer que este lenguaje tenga también significado.

1. La Gramática

```
<programa>      :   <sentencia> <programa'>
                |   <control> <programa'>
                |   COMENTARIO <programa>

<programa'>     :   <sentencia> <programa'>
                |   <control> <programa'>
                |   COMENTARIO <programa'>
                |   <empty>

<sentencia>     :   <var_asig> ';'
                |   <funcion> ';'

<control>       :   <ifelse>
                |   <loop>

<control_cond>  :   <var_asig_l>
                |   <exp_bool>
                |   <comparacion>
                |   <op_ternario>

<loop>          :   'while' '(' <control_cond> ')' <bloque>
                |   'do' <bloque> 'while' '(' <control_cond> ')' ';'
                |   <for>

<for>           :   'for' '(' <for_term> ',' <form_term_2> ',' <for_term> ')' <bloque>


<for_term>      :   <var_asig>
                |   <empty>

<form_term_2>    :   <valores>
                |   <comparacion>

<ifelse>        :   'if' '(' <control_cond> ')' <bloque>
                |   'if' '(' <control_cond> ')' <bloque> else <bloque>

<bloque>        :   COMENTARIO <bloque>
                |   <sentencia>
                |   <control>
                |   '{' <programa'> '}'

<funcion>       :   func_ret
                |   func_void
```

```
<func_void>     :   'print' '(' <valores> ')'

<func_ret>      :   <func_ret_int>
                |   <func_ret_cadena>
                |   <func_ret_bool>
                |   <func_ret_arreglo>

<func_ret_bool>     :   'colineales' '(' <valores> ',' <valores> ')'

<func_ret_cadena>   :   'capitalizar' '(' <valores> ')'

<func_ret_arreglo>  :   'multiplicarEscalar' '(' <valores> ',' <valores> ')'
                    |   'multiplicarEscalar' '(' <valores> ',' <valores> ',' <valores>
<func_ret_int>  :   'length' '(' valores ')'


<valores>       :   <exp_arit>
                |   <exp_bool>
                |   <exp_cadena>
                |   <exp_arreglo>
                |   <registro>
                |   <registro> '.' VARIABLE
                |   <var_asig_l>
                |   <op_ternario>

<exp_arreglo>   :   '[' <list_valores> ']'
                |   '[' <list_valores> ']' <exp_arreglo>
                |   '['']'
                |   func_ret_arreglo

<lista_valores> :  <valores>
                |  <valores> ',' <lista_valores>

<registro>      :   '{' <reg_item> '}'

<reg_item>      :   VARIABLE ':' <valores> ',' <reg_item>
                |   VARIABLE ':' <valores>

<var_asig_l>    :   VARIABLE
                |   RES
                |   VARIABLE <var_member>


<var_member>    :   '[' var_asig_l ']' <var_member>
                |   '[' <exp_arit> ']'  <var_member>
                |   '[' <exp_arit> ']'
```

```
                |   '[' <var_asig_l> ']'
                |   '.' VARIABLE
                |   '.' VARIABLE <var_member>

<var_asig>      :   <var_asig_l> '++'
                |   '++' <var_asig_l>
                |   <var_asig_l> '--'
                |   '--' <var_asig_l>
                |   <var_asig_l> '*=' <valores>
                |   <var_asig_l> '/=' <valores>
                |   <var_asig_l> '+=' <valores>
                |   <var_asig_l> '-=' <valores>
                |   <var_asig_l> '=' <valores>
                |   <var_asig_l> '=' <comparacion>

<op_ternario>   :   <op_ternario_1>
                |   <op_ternario_paren>

<op_ternario_paren>     :   '(' <op_ternario_1> ')'
<op_ternario_1>         :   <valores> '?' <op_ternario_2>
<op_ternario_2>         :   <valores> ':' <op_ternario_3>
<op_ternario_3>         :   <valores>


<var_oper>      :   <var_asig_l>
                |   <op_ternario_paren>
                |   <exp_arreglo>
                |   <registro> '.' VARIABLE


<exp_arit>      :   <exp_arit> '+' <term>
                |   <exp_arit> '+' <var_oper>
                |   <var_oper> '+' <term>
                |   <var_oper> '+' <var_oper>
                |   <exp_arit> '-' <term>
                |   <exp_arit> '-' <var_oper>
                |   <var_oper> '-' <term>
                |   <var_oper> '-' <var_oper>
                |   <term>


<arit_oper_2>   :   '*'
                |   '/'
                |   '%'

<term>          :   <term>  <arti_oper_2> <factor>
```

```
                |   VARIABLE <arit_oper_2> <factor>
                |   <var_oper> <arit_oper_2> <factor>
                |   <term> <arit_oper_2> VARIABLE
                |   <term> <arit_oper_2> <var_oper>
                |   VARIABLE <arit_oper_2> VARIABLE
                |   <var_oper> <arit_oper_2> <var_oper>
                |   <factor>

<factor>        :   <base> '^' <sigexp>
                |   <var_oper> '^' <sigexp>
                |   VARIABLE '^' <sigexp>
                |   '-' <base>
                |   <base> '++'
                |   <base> '--'
                |   '++' <base>
                |   '--'<base>
                |   VARIABLE '++'
                |   '++' VARIABLE
                |   VARIABLE '--'
                |   '--' VARIABLE
                |   <var_oper> '++'
                |   '++' <var_oper>
                |   <var_oper> '--'
                |   '--' <var_oper>
                |   <base>

<base>          :   '(' <exp_arit> ')'
                |   '(' VARIABLE ')'
                |   NUMBER
                |   <func_int>

<sigexp>        :   '-' <exp>
                |   <exp>

<exp>           :   VARIABLE
                |   NUMBER
                |   '(' <exp_arit> ')'

<exp_cadena>    :   <exp_cadena> '+' <term_cadena>
                |   <var_oper> '+' <term_cadena>
                |   <term_cadena> '+' <var_oper>
                |   <term_cadena>

<term_cadena>   :   CADENA
                |   'capitalizar' '(' <valores> ')'
                |   '(' <exp_cadena> ')'
```

```
<comparacion>   :   <exp_cadena> <op_comp>  <exp_cadena>
                |   <var_oper> <op_comp>  <exp_cadena>
                |   <exp_cadena> <op_comp> <var_oper>
                |   <exp_arit>   <op_comp>   <exp_arit>
                |   <var_oper>   <op_comp>   <exp_arit>
                |   <exp_arit>   <op_comp>   <var_oper>
                |   <exp_bool> <op_comp> <exp_bool>
                |   <exp_bool> <op_comp> <var_oper>
                |   <var_oper> <op_comp> <exp_bool>
                |   <var_oper>   <op_comp>  <var_oper>

<exp_bool>      :   <exp_bool> AND <term_bool>
                |   <var_oper> AND <term_bool>
                |   <exp_bool> AND <var_oper>
                |   <var_oper> AND <var_oper>
                |   <term_bool>

<term_bool>     :   <term_bool> OR <factor_bool>
                |   <var_oper> OR <factor_bool>
                |   <term_bool> OR <var_oper>
                |   <var_oper> OR <var_oper>
                |   'not' <factor_bool>
                |   'not' <var_oper>

<factor_bool>   :   BOOL
                |   '(' <exp_bool> ')'
                |   '(' <comparacion> ')'
                |   <func_bool>
                |   <comparacion>

<op_comp>       :   '=='
                |   '>'
                |   '<'
                |   '!='
```

2. El Código

3. Casos de Prueba

A continuación presentaremos una serie de casos correctos e incorrectos respecto a la sintáxis del programa.

3.1. Casos Correctos.

3.1.1. Caso 1:

```
#NADA
z = [a.edad + length("doce"+capitalizar("2")) / 2, 1.0 + (siete ? ((aa OR bb AND (2!=
g*12 %3) , NOT aa ? b / -5 : c]; while(dos);

a = 0;
b = 1;
a = b;
c = 2.5;
d = "Hola";
e = d;

f = [1, 2, 3];
a = f[1];
b = f[2];

g = [1, a, 2, b, 3];
g[a] = b;
g[b] = a;

h = [f[g[a]], g[g[b]]];
h[f[g[a]]] = g[f[g[g[b]]]];
```

3.1.2. Caso 2:

```
for (i=0; i<10; i++) {
while(true) {
for(;true;) {
do {
a=0;
}
while(false);
}
```

```
}
}
```

3.1.3. Caso 3:

```
a = -10;
b = 2 + 3 - 8 + 1;
c = 3 * 5 % 4 / 6;
d = 2.0 ^ 9.0;
e = -3 * (4 / ((3) + 8 ^ ((1))) - -5 + -(7) * 6);

f = -a * (b / ((a) + d ^ ((c))) - -b + -(c) * a);

g = [1.5, 2 * (7) + 5 / 0, a * d, 8 ^ 1.0, (6) / (6), (3 + 3) + 0.0, e - 5, 2 * -f];
h = [1,b,3];

g[2 * 4] = 10 + (3 * 2/1);
g[b ^ b] = (g[7 * (5) + (h[(2)])]);
g[9 - (5 % 3)] = g[b % h[b]];

while (true) {
d = 1;
a = 0;
}

do {
# Co
# men
# tario
b = 2;
c = 3;
} while ((d > 10) AND NOT false);

usuario = {nombre:"Al", edad:50};
print(capitalizar(usuario.nombre));
nacimiento = 2016;
nacimiento -= usuario.edad;
usuarios = [{nombre:"Mr.X", edad:10}, usuario];

suma = 0;
for (i = 0; i < length(usuarios); i++) {
print(usuarios[i].nombre);
suma += usuarios[i].edad;
}
```

3.2. Casos incorrectos.

3.2.1. Caso 1.

```
else
  b=0;
```

3.2.2. Caso 2.

```
while(true){
#comentario sin sentencias
}
```

3.2.3. Caso 3.

```
for(true){}
```

3.2.4. Caso 4.

```
While = 10;
```

3.2.5. Caso 5.

```
a = 5
```

3.2.6. Caso 6.

```
multiplicacionEscalar("1,2,3");
```

3.2.7. Caso 7.

```
multiplicacionEscalar();
```