Gramáticas Atributadas y AST

Evaluando expresiones

Recordemos la gramática LL de expresiones

```
E -> T X
X -> + T X | - T X | epsilon
T -> F Y
Y -> * F Y | / F Y | epsilon
F -> ( E ) | i
```

¿Cómo evaluar una expresión cualquiera?

Ej: `i{3} * ( i{2} + i{5} )`

Implementando un evaluador de expresiones

```python
def eval(node, val=None):
    if node.type == 'i':
        return float(node.lex)
    ## ...
    if node.type == 'E':
        val = eval(node.children[0])
        return eval(node.children[1], val)
    if node.type == 'X':
        if not node.children:
            return val
        if node.children[0].type == '+':
            val = val + eval(node.children[1])
        elif node.children[0] == '-':
            val = val - eval(node.children[1])
        return eval(node.children[2], tmp)
    ## ...
```

¿Qué hay de malo con este código?

- LL nos obliga a hacer malabares con los valores

- Montón de ramas `if/else` similares
- Para saber que producción se aplicó tengo que mirar los hijos
- Para cualquier gramática LL va a ser algo igualmente horrible
- En general es poco OO y bastante feo...

Parece código escrito por una máquina...

Para cada no-terminal en el árbol de derivación, según la producción aplicada, hay un código específico.

¿Podemos generalizar este proceso, y dejárselo a las máquinas?

Extendamos el concepto de gramática

```
E -> T X      { X.val=T.exp, E.exp=X.exp }
X -> + T X    { X1.val=X0.val + T.exp, X0.exp=X1.exp }
   | - T X    { X1.val=X0.val - T.exp, X0.exp=X1.exp }
   | epsilon  { X0.exp=X0.val }
T -> F Y      { Y.val=F.exp, T.exp=Y.exp }
Y -> * F Y    { Y1.val=Y0.val * F.exp, Y0.exp=Y1.exp }
   | / F Y    { Y1.val=Y0.val / F.exp, Y0.exp=Y1.exp }
   | epsilon  { Y0.exp=Y0.val }
F -> ( E )    { F.exp=E.exp }
   | i        { F.exp=i.exp }
```

Gramáticas atributadas

Una **gramática atributada** es una tupla $< G, A, R >$, donde:

- $G =< S, P, N, T >$ es una gramática libre del contexto,
- A es un conjunto de atributos de la forma $X \cdot a$ donde $X \in N \cup T$ y a es un identificador único entre todos los atributos del mismo símbolo, y
- R es un conjunto de reglas de la forma $< p_i, r_i >$ donde $p_i \in P$ es una producción $X \to Y_1, \ldots, Y_n$, y r_i es una regla de la forma:

  i. $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$, o
  ii. $Y_i \cdot a = f(X \cdot a_0, Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$.

  En el primer caso decimos que a es un **atributo sintetizado**, y en el segundo caso, un **atributo heredado**.

Intuitivamente...

- En cada no-terminal hay atributos, que pueden almacenar algún valor
- En cada terminal hay atributos que pone mágicamente el tokenizer
- En cada producción hay un conjunto de reglas que involucran a algunos de los atributos

Todavía no hemos dicho en qué orden evaluar los atributos

De momento asumiremos que existe una forma de evaluarlos...

Resolviendo dependencias del contexto

Veamos una gramática para el lenguaje $L = a^n b^n c^n$:

```
S -> ABC
A -> aA | epsilon
B -> bB | epsilon
C -> cC | epsilon
```

Resolviendo dependencias del contexto

Veamos las reglas semánticas

```
S -> ABC       {   S.ok = (A.cnt == B.cnt &&
                           B.cnt == C.cnt) }
A -> aA        { A0.cnt = 1 + A1.cnt }
   | epsilon   {  A.cnt = 0 }
B -> bB        { B0.cnt = 1 + B1.cnt }
   | epsilon   {  B.cnt = 0 }
C -> cC        { C0.cnt = 1 + C1.cnt }
   | epsilon   {  C.cnt = 0 }
```

Resolviendo dependencias del contexto

Veamos ahora una gramática más compleja

```
S -> XC
X -> aXb | epsilon
C -> cC  | epsilon
```

Resolviendo dependencias del contexto

Y sus respectivas reglas

```
S -> XC       {   S.ok = (X.cnt == C.cnt) }
X -> aXb      { X0.cnt = 1 + X1.cnt }
   | epsilon  {  X.cnt = 0 }
C -> cC       { C0.cnt = 1 + C1.cnt }
   | epsilon  {  C.cnt = 0 }
```

¿Cuál es mejor?

Gramáticas más simples

- Más fácil de entender
- Más difícil de parsear
- Cambios menos disruptores

Gramáticas más complejas

- Más difícil de entender
- Más eficiente de parsear
- Cambios más disruptores

La legibilidad y la mantenibilidad cuentan !!

Computando el valor de los atributos

Sea la cadena `aabbcc`, tenemos un árbol de derivación

Construyendo un *grafo de dependencia*

Un grafo de dependencia es un grafo dirigido sobre el árbol de derivación, donde:

- Los nodos son los atributos de cada símbolo en cada nodo del árbol de derivación
- Existe una arista dirigida entre un par de nodos $v \to w$ del grafo de dependencia, si los atributos correspondientes participan en una regla semántica definida en la producción asociada al nodo del árbol de derivación donde aparece w, de la forma $w = f(\ldots, v, \ldots)$. Es decir, si el atributo w *depende* del atributo v.

¿Qué nos dice este grafo?

Un grafo de dependencias nos permite:

- saber si la cadena es evaluable, y
- determinar un orden para evaluar los atributos.

Decimos que la gramática es evaluable si y solo si el grafo de dependencia de todo árbol de derivación es un DAG.

Algunas gramáticas particulares son fáciles de evaluar:

Gramáticas s-atributadas:

Una gramática atributada es **s-atributada** si y solo si, para toda regla r_i asociada a una producción $X \to Y_1, \ldots, Y_n$, se cumple que r_i es de la forma $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$.

Gramáticas l-atributadas:

Una gramática atributada es **l-atributada** si y solo si toda regla r_i asociada a una producción $X \to Y_1, \ldots, Y_n$ es de una de las siguientes formas:

1. $X \cdot a = f(Y_1 \cdot a_1, \ldots, Y_n \cdot a_n)$, ó
2. $Y_i \cdot a_i = f(X \cdot a, Y_1 \cdot a_1, \ldots, Y_{i-1} \cdot a_{i-1})$.

Evaluando atributos durante el *parsing* LL

```
E -> T X      { X.val=T.exp, E.exp=X.exp }
X -> + T X    { X1.val=X0.val + T.exp, X0.exp=X1.exp }
   | - T X    { X1.val=X0.val - T.exp, X0.exp=X1.exp }
   | epsilon  { X0.exp=X0.val }
T -> F Y      { Y.val=F.exp, T.exp=Y.exp }
Y -> * F Y    { Y1.val=Y0.val * F.exp, Y0.exp=Y1.exp }
   | / F Y    { Y1.val=Y0.val / F.exp, Y0.exp=Y1.exp }
   | epsilon  { Y0.exp=Y0.val }
F -> ( E )    { F.exp=E.exp }
   | i        { F.exp=i.exp }
```

Retomando el árbol de derivación

¿Qué problemas tenía evaluar esta expresión?

- LL nos obliga a usar atributos heredados.
- La operación a realizar depende de la producción aplicada (tengo que recordarla en la pila).
- Muchas reglas semánticas son solo para "mover" los valores de un lado al otro del árbol.

¿Cuando el lenguaje crezca, cómo evaluaremos...

- invocaciones de funciones?
- declaración y asignación de variables?
- ciclos?

> La representación en árbol de derivación es **sintáctica**. Necesitamos una representación **semántica**.

¿Cómo podemos simplificar este árbol?

- Paso 1: Eliminar los nodos `epsilon`
- Paso 2: Eliminar los nodos hoja sobrantes
- Paso 3: Eliminar `(` y `)`
- Paso 4: Eliminar los nodos con un solo hijo
- Paso 4: Eliminar los nodos con un solo hijo
- Paso 4: Eliminar los nodos con un solo hijo
- Paso 5: Subir las operaciones
- Paso 5: Subir las operaciones
- Finalmente...

¿Qué hemos ganado con esta representación?

> Este árbol contiene **toda** la información necesaria para **evaluar**, *y nada más*.

A este tipo de árbol le llamamos *Árbol de Sintaxis Abstracta (AST)*, o árbol semántico.

NOTA: Por analogía al árbol de derivación se le llama también *árbol de sintaxis concreta*.

Definición de AST

Árbol de Sintaxis Concreta

Define un tipo de nodo por cada tipo de **función sintática** diferente:

- Palabras claves
- Separadores
- Operadores
- Bloques
- Identificadores
- ...

Su forma representa la estructura *sintáctica* del lenguaje.

Se diseña de modo que el lenguaje sea fácil de **parsear** (*leer*).

Definición de AST

Árbol de Sintaxis Abstracta

Define un tipo de nodo por cada tipo de **función semántica** diferente:

- Expresiones
- Ciclos
- Condicionales
- Declaraciones
- Invocaciones
- ...

Su forma representa la estructura *semántica* del lenguaje.

Se diseña de modo que el lenguaje sea fácil de **evaluar** (*entender*).

Un AST es una jerarquía de clases

```
class Exp:
    def eval(self):
        raise NotImplementedError()

class Sum(Exp):
    def __init__(self, left, right):
        self.left = left
        self.right = right
#    # ...

class Num(Exp):
    def __init__(self, value):
        self.value = value
#    # ...
```

Evaluar es muy fácil

```
class Sum(Exp):
#    # ...
    def eval(self):
        return self.left.eval() + self.right.eval()

class Mul(Exp):
#    # ...
    def eval(self):
        return self.left.eval() * self.right.eval()

class Num(Exp):
#    # ...
    def eval(self):
        return self.value
```

¿Cómo construimos el AST?

```
E -> T X    { X.tmp=T.ast, E.ast=X.ast }
X -> + T X  { X1.tmp=Add(X0.tmp,T.ast), X0.ast=X1.ast }
   | - T X  { X1.tmp=Sub(X0.tmp,T.ast), X0.ast=X1.ast }
   | eps    { X0.ast=X0.tmp }
T -> F Y    { Y.tmp=F.ast, T.ast=Y.ast }
Y -> * F Y  { Y1.tmp=Mul(Y0.tmp,F.ast), Y0.ast=Y1.ast }
   | / F Y  { Y1.tmp=Div(Y0.tmp,F.ast), Y0.ast=Y1.ast }
   | eps    { Y0.ast=Y0.tmp }
F -> ( E )  { F.ast=E.ast }
   | i      { F.ast=Num(i.val) }
```

¡¡ Terminado de parsear y el AST ya está listo !!

Podemos definir entonces el proceso de **parsing** como el mecanismo para obtener un **AST** de una cadena.

¿Para qué queremos un AST?

Porque nos hemos liberado de la representación sintáctica, ahora podemos:

- Interpretar directamente el AST
- Verificar la validez de las expresiones
- Detectar tantos errores como sea posible
- Compilar a un lenguaje más eficiente de evaluar
- Optimizar expresiones y transformar código
- Transpilar a otro lenguaje de alto nivel

> Diseñar buenas gramáticas es una **ciencia**, pero diseñar buenos **AST** es un arte.

¿Cuántos AST distintos tengo?

En principio, tantos **AST** como representaciones semánticas distintas tenga mi proceso de compilación. En el compilar de COOL:

- Un primer **AST** para verificar consistencia de tipos.
- Un segundo **AST** en lenguaje intermedio (con posibles optimizaciones).
- Un tercer **AST** en MIPS.

Podemos ver el proceso de compilación como:

```
código -> AST -> ... -> AST -> código
```

Al final es una traducción de un lenguaje a otro, pasando por representaciones intermedias convenientes.