DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Teoría de lenguajes

Trabajo práctico

Parser

Resumen

Este trabajo consiste en parser estilo c++

Integrante	LU	Correo electrónico
Acosta, Javier Sebastian	338/11	acostajavier.ajs@gmail.com
Mastropasqua Nicolas Ezequiel	828/13	mastropasqua.nicolas@gmail.com
Negri, Franco	893/13	franconegri2004@hotmail.com

Palabras claves:

TP

Índice

1. Introducción

El objetivo del presente trabajo práctico es la realización de un analizador léxico y sintáctico para un lenguaje de scripting, mediante el uso de una gramática LALR y la herramienta para generar analizadores sintácticos **PLY**.

El analizador tendrá como objetivo aceptar cadenas que corresponden a programas estilo c++.

Ademas dado un lenguaje de entrada, que el analizador sintáctico lo acepta, se retorna una salida correspondiente con la correcta indentación del código.

2. Descripción del lexer:

3. Descripción de la gramática

3.1. Gramática inicial

Comenzamos el armado de la gramática definiendo los tipos básicos que debíamos aceptar en nuestro lenguaje.
Así definimos los tipos Bool, String, Float, Bool, Int, Vector, Registro. De allí el siguiente paso fue ver que operaciones se le podía realizar a cada uno de ellos.
Mientras realizábamos esto empezamos a notar que necesitaríamos determinar la precedencia de estas operaciones y la asociatividad. Para esto nos basamos en la precedencia de c++ (http://en.cppreference.com/w/cpp/language/operator_precedence) dado lo parecido que es al lenguaje de entrada.

Una vez que todas las expresiones del lenguaje estaban completas, lo pasamos a código Python usando la herramienta *PLY*. Ya terminado el código, nos encontramos de que la gramática tenía muchos mas conflictos que los que se podía ver a simple vista y por lo tanto era ambigua.

3.2. Gramática ambigua

Luego de resolver algunos conflictos, nos dimos cuenta de que era un proceso muy complejo pues cada conflicto implicaba añadir mas reglas a la gramática haciendo que sea mas compleja y difícil de entender.

Después de leer la documentación de ply (http://www.dabeaz.com/ply/ply.html y observar que ply puede encargarse de los conflictos de una gramática ambigua usando reglas de asociatividad y precedencia, decidimos hacer una gramática ambigua sencilla y aprovechar esta característica de ply.

3.3. Gramática no ambigua

Si bien esto solucionó todos los conflictos de una manera sencilla, por desgracia, nos comunicaron de que lo que se pedía era una gramática **no** ambigua. Por lo tanto decidimos tirar la gramática y comenzar de nuevo.

Con el tiempo contado y sin ideas mejores nos dimos cuenta que la gramática inicial no estaba tan mal, tenía toda la precedencia definida de las expresiones y que una solución rápida era hacer la gramática menos restrictiva, es decir, que acepte mas cadenas de lo que se pida y rechazar las que no pertenecen al lenguaje mediante atributos. La tarea de los atributos (además de armar el

código indentado) es chequear los tipos de las expresiones para que, si una operación acepta unos tipos determinados y se le pasa una expresión de otro tipo, lance una excepción de error de tipos.

3.4. Valores

Los valores son los cualquier expresión que denote un valor (por ejemplo 3 * 4 + 3). La primer idea era definir valores como **cualquier** expresión (para poder usarla como valor en asignaciones, en funciones como parámetro, etc.):

$Valores \rightarrow ExpresionMatematica|ExpresionBooleana|ExpresionString|ExpresionVector|$
$ExpresionRegistro|OperacionVariables$

Donde cada producción que empieza con *Expresion* denota una expresión de un tipo determinado que también puede generar los valores primitivos (bool, string, float, bool, int, vec, reg), variables (id) y además funciones que retornen valores a su tipo correspondiente (por ejemplo *capitalizar* en *Expresionstring*). $OperacionVariables$ genera las expresiones que usan operaciones de variables (asignación, incremento, decremento, etc).

El problema con esto es que **id** se puede generar en todas las expresiones salvo en la última. Si se tiene una producción $A \rightarrow Valores$, entonces se puede llegar de 5 maneras distintas a **id** y, por lo tanto, hay conflictos (la gramática es ambigua).

Por lo tanto decidimos sacar a **id** de cada producción. El problema es que se tiene que seguir produciendo expresiones complejas con **id** (por ejemplo *id and id*).

Para solucionar esto, decidimos sacar todos los valores primitivos de cada expresión y colocarlo en $Valores$. Además hay que observar que una función de un determinado tipo puede devolver una variable de ese tipo, por lo que decidimos sacar también a esas funciones. Lo mismo pasa con los operadores ternarios. De esta forma, cada producción *Expresion* genera expresiones con al menos un operador de su tipo:

$Valores \rightarrow ExpresionMatematica|ExpresionBooleana|ExpresionString|ExpresionVector|$
$ExpresionRegistro|OperacionVariables|bool|string|float|bool|int|vec|reg|id|FuncReturn|Ternario$

$FuncReturn \rightarrow FuncInt|FuncString|FuncBool|FuncVector$

$Ternario- > Ternariomat|Ternariobool|Ternariostring|Ternariovector|Ternarioregistro$

3.5. Expresiones Matemáticas

Para las expresiones matemáticas consideramos la siguiente tabla de precedencia:

Tipo	Operador	Asociatividad
Binario	+,-	izquierda
Binario	*,/, %	izquierda
Binario	^	izquierda
Unario	+,-	
Unario	()	

Cuadro 1: Tabla de menor a mayor precedencia

La producción que se encarga de realizar estas expresiones es eMat.

La primera aproximación fue hacer la gramática para poder respetar la precedencia y asociatividad de los operadores.

Este proceso se logra teniendo en cuenta de que se puede fijar la precedencia (por ejemplo entre + y $*$), con las reglas de la gramática. Como bien se sabe, el operador + tiene menos precedencia

que el $*$. Una gramática que respeta esto es la siguiente:

$A \rightarrow A + B$
$B \rightarrow B * C$
$C \rightarrow int|float$

Esto se debe a que antes de calcular la suma entre A y B, hay que parsear la producción B (pues es un parser ascendente).

Esto también respeta la asociatividad de ambos operados (que es a izquierda) por la misma razón que antes: La única forma de colocar mas de un operador + es usando la producción A y, por lo tanto, se tiene que parsear A antes que calcular la suma.

Este proceso se realiza con todos los operadores, de manera que si 2 operadores tienen la misma precedencia, entonces parten de un mismo no terminal.

A esta gramática le falta la posibilidad de agregar paréntesis para poder cambiar la precedencia. Una forma de hacerlos es agregarlos a C para que estos tengan mas prioridad que todas las operaciones.

Además también falta poder agregar variables. Para esto agregamos un no terminal a C llamado id que las representa. La gramática resultante es esta:

$A \rightarrow A + B$
$B \rightarrow B * C$
$C \rightarrow (A)|int|float|id$

Para evitar problemas de ambigüedad, decidimos hacer que eMat devuelva expresiones con al menos un operador matemático (ver Tabla de precedencia). Esto es por que en la producción valores se tienen las variables (correspondientes al token ID) y estas también se generarían en las expresiones de los otros tipos.

3.6. Problema del Dangling else

Otro problema que encontramos ya avanzados en el proceso de desarrollo de la gramática fue el problema del Dangling else (https://en.wikipedia.org/wiki/Dangling_else) que consiste en que los else opcionales en un if hacen que la gramática sea ambigua.
Por ejemplo:

3.7. Gramática Final

A continuación se define la gramática utilizada para construir el parser. La misma es **no ambigua** y **LALR**. La garantía de esto es que ply acepta gramáticas de este tipo, nuestra implementación no arroja conflictos **shift/reduce** o **reduce/reduce**

G—$< V_t, V_{nt}, g, P >$

V_t es el conjunto de símbolos terminales dado por los símbolos en **mayúsculas** que aparecen en las producciones.

V_{nt} es el conjunto de símbolos no-terminales dado los literales(operadores) y los símbolos en **minúsculas** que aparecen en las producciones.

P es el conjunto de producciones dadas a continuación|

Sentencias y estructura general

g → linea g | COMMENT g | empty

linea → lAbierta | lCerrada

Linea Abierta: Hay por lo menos un IF que no matchea con un else

lAbierta → IF (cosaBooleana) linea |

| IF (cosasBooleana) g ELSE lAbierta

| IF (cosasBooleana) g ELSE lAbierta

| IF (cosasBooleana) g

| loop Labierta

Las siguientes son las variantes de tener bloques cerrados else bloques cerrados.

Un bloque cerrado puede ser una sentencia única o un bloque entre llaves.

En cada uno de estos casos puede haber, o no, comentarios. De ahí todas estas combinaciones.

lCerrada → sentencia

| COMMENT com | lambda

| IF (cosaBooleana) g ELSE g

| IF (cosaBooleana) lCerrada ELSE g

| IF (cosaBooleana) COMMENT com lCerrada ELSE g

| IF (cosaBooleana) g ELSE lCerrada

| IF (cosaBooleana) lCerrada ELSE lCerrada

| IF (cosaBooleana) COMMENT com lCerrada ELSE lCerrada

| IF (cosaBooleana) lCerrada ELSE COMMENT com lCerrada

| IF (cosaBooleana) COMMENT com lCerrada ELSE COMMENT com lCerrada

| loop g

| loop lCerrada

| loop COMMENT com lCerrada

| DO g WHILE (valores) ;

| DO lCerrada WHILE (valores) ;

| DO COMMENT com lCerrada WHILE (valores) ;

Sentencias básicas:

sentencia → varsOps | func ; | varAsig ; | RETURN; | ;

Bucles:

loop → WHILE (valores) | FOR (primerParam ; valores ; tercerParam)

primerParam → varAsig | lambda

tercarParam → varsOps | varAsig | func | lambda

cosaBooleana → expBool | valoresBool

Funciones:

func → FuncReturn | FuncVoid

funcReturn → FuncInt | FuncString | FuncBool

funcInt → MULTIESCALAR(valores, valores param)

funcInt → LENGTH(valores)

funcString → CAPIALIZAR(valores)

FuncBool → colineales(valores,valores)

FuncVoid → print(Valores)

param → valores | lambda

Vectores y variables:

vec → [elem]

elem → valores,elem | valores

vecval → id [expresion] | vec [expresion] | vecVal [expresion] | ID[INT]

expresion → eMat | expBool | funcReturn | reg | FLOAT | |STRING | | RES
| BOOL | varYVals | varsOps | vec | atributos | ternario

valores → varYVals | varsOps | eMat | expBool | funcReturn | reg | INT | FLOAT
| STRING | BOOL | ternario | atributos | vec | RES

atributos → ID.valoresCampos | reg.valoresCampos

valoresCampos → varYVals | END | BEGIN

Operadores ternarios:

ternario → ternarioMat | ternarioBool | (ternarioBool) | (ternarioMat)
| ternarioVars | (ternarioVars)

ternarioVars → valoresBool ? valoresTernarioVars : valoresTernarioVars
| valoresBool ? valoresTernarioVars : valoresTernarioMat
| valoresBool ? valoresTernarioMat : valoresTernarioVars
| valoresBool ? valoresTernarioVars : valoresTernarioBool
| valoresBool ? valoresTernarioBool : valoresTernarioVars
| expBool ? valoresTernarioVars : valoresTernarioVars

| expBool ? valoresTernarioVars : valoresTernarioMat

| expBool ? valoresTernarioMat : valoresTernarioVars

| expBool ? valoresTernarioVars : valoresTernarioBool

| expBool ? valoresTernarioBool : valoresTernarioVars

valoresTernaioVars → reg | vec | ternarioVars | (ternarioVars) | atributos

| varsOps | varYVals | RES

valoresTernarioMat → valoresBool ? valoresTernarioMat : valoresTernarioMat

| expBool ? valoresTernarioMat : valoresTernarioMat

valoresTernarioMat → INT | FLOAT | funcInt | STRING | eMAt

| ternarioMat | (ternarioMat)

ternarioBool → valoresBool ? valoresTernarioBool : valoresTernarioBool

| expBool ? valoresTernarioBool : valoresTernarioBool

valoresTernarioBool → BOOL | funcBool | ternarioBool | (ternarioBool)| expBool

varYVals:

varYVals → ID | vecVal | vecVal.varYVals

Registros:

reg → campos

campos → ID:valores, campos | ID:valores

Operadores de variables:

varsOps→ MENOSMENOS varYVals | MASMAS varYVals

| varYVals MASMAS | varYVals MENOSMENOS

Asignaciones:

varAsig → variable MULEQ valores | variable DIVEQ valores | variable MASEQ valores

| variable MENOSEQ valores | variable = valores| ID . ID = valores

variable → ID | vecVal | vecVal.varYVals

Operaciones binarias enteras:

valoresMat → INT | FLOAT | funcInt | atributos | funcString

| STRING | varYVals | varsOps | (ternarioMat)

eMat → eMat + p | valoresMat + p | eMat + valoresMat | valoresMat + valoresMat

| eMat - p | valoresMat - p | eMat - valoresMat | valoresMat - valoresMat | p

p → p * exp | p / exp | p | valoresMat * exp | valoresMat / exp | valoresMat

| p * valoresMat | p / valoresMat | p % valoresMat | valoresMat * valoresMat

| valoresMat valoresMat | valoresMat % valoresMat | exp

exp → exp îSing | valoresMat îSing | exp v̂aloresMat
| valoresMat v̂aloresMat | iSing
iSing → - paren | + paren | - valoresMat | + valoresMat | paren
paren → (eMat) | (valoresMat)

Expresiones booleanas:

valoresBool → BOOL | funcBoo l| varYVals | varsOps | (ternarioBool)
expBool → expBool OR and | valoresBool OR and |
| expBool OR valoresBool | valoresBool OR valoresBool | and
and → and AND eq | valoresBool AND eq | and AND valoresBool
| valoresBool AND valoresBool | eq
eq → eq EQEQ mayor | eq DISTINTO mayor | tCompareEQ EQEQ mayor
| tCompareEQ DISTINTO mayor | eq EQEQ tCompareEQ
| eq DISTINTO tCompareEQ tCompareEQ EQEQ tCompareEQ
| tCompareEQ DISTINTO tCompareEQ | mayor
tCompareEQ → BOOL | funcBool | varYVals | varsOps | INT
| FLOAT | funcInt | eMat | (ternarioBool) | (ternarioMat)
tCompare → eMat | varsOps | varYVals | INT | funcInt | FLOAT | (ternarioMat)
mayor → tCompare > tCompare | menor
menor → tCompare < tCompare | not
not → NOT not | NOT valoresBool | parenBool
parenBool → (expBool)

4. Descripción de la implementación

5. Conclusiones