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

Contents

1 Introducción al problema:

El problema planteado implica el desarrollo de un analizador léxico y sintáctico para un lenguaje de programación dado. Para ello, se buscará una gramática que genere dicho lenguaje y además cumpla con los requisitos necesarios para construir un parser a partir de la misma. Finalmente, además de poder decidir si una cadena de texto pertenece al lenguaje, se formateará la cadena de texto de entrada para cumplir con la indentación adecuada en el caso de que la misma sea válida.

Aclaracion: El informe es provisorio, hay varias secciones sin definir todavía.

2 Descripción del lexer:

3 Descripción de la gramática

3.1 Introducción:

Comenzamos el armado de la gramática definiendo los tipos básicos que debíamos aceptar en nuestro lenguaje.
Así definimos los tipos Bool $'STRING'$,$'FLOAT'$,$'BOOL'$, $'INT'$. De allí el siguiente paso lógico 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 además que necesitaríamos incluirlas dentro de la gramática si queríamos que no terminase resultando ambigua.
Ya avanzados en el proceso de desarrollo de la gramática nos dimos cuenta que esta era demasiado restrictiva. Era imposible realizar una gramática que nos permitiera aceptar exactamente lo que queríamos, por lo que decidimos optar por hacerla mas laxa y restringir lo que quisiéramos utilizando chequeo de tipos.
Llegado ese punto descubrimos que existía una forma de darle a ply una gramática ambigua, darle reglas de precedencia y que ply se encargara de resolver los conflictos. Nos pusimos muy felices porque eso facilitaba mucho el armado de la gramática. Dos días mas tarde descubrimos que esta no era una solución valida y volvimos a trabajar sobre nuestra gramática no ambigua.
Otro problema que encontramos ya avanzados en el proceso de desarrollo de la gramática fue el problema del Dangling else que consiste en que los else opcionales en un if resultan en que la gramática sea ambigua.

En la **sección 3.3** 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**

3.2 Expresiones Matemáticas

Para las expresiones matemáticas consideramos la siguiente tabla de precedencia:

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

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.

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

Table 1: Tabla de menor a mayor precedencia

3.3 Implementación:

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 ŵvaloresMat
| valoresMat ŵvaloresMat | 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 Código

6 Conclusiones