DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

2 de Julio de 2014 — Teoría de lenguajes

Integrante	LU	Correo electrónico
Axel Iglesias	79/10	axeligl@gmail.com
Agustin Martínez Suñé	630/11	agusmartinez.92@gmail.com
Nahuel Lascano	476/11	laski.nahuel@gmail.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

1. Introducción

El objetivo del Trabajo Práctico es crear un lenguaje de programación imperativo que permita definir funciones que serán evaluadas y nos permitirán traficar curvas paramédicas en el plano.

Para esto, tuvimos que decidir e implementar los siguientes puntos:

- La gramática que vamos a usar para la implementación
- El analizador léxico que recibe el código fuente
- El parser que toma los tokens y verifica que coincida con la gramática
- El Abstract Syntax Tree que permite representar las funciones del código fuente

Para la implementación decidimos utilizar BISON y C++ y utilizamos como referencia la documentación de BISON y el blog que fue recomendado.

2. Decisiones

2.1. BISON

Como se nombró en la introducción, para la implementación del parser decidimos usar BISON. Esto se debe a la documentación que encontramos para desarrollar el parser y a la aparente sencillez para usarlo.

2.2. Comentarios

Con respecto al uso de comentarios en el código fuente, para la primer entrega decidimos usar el comando *sed* para eliminarlos. Esta decisión fue más que nada por ignorancia de cómo implementarlo en BISON.

Para esta segunda entrega, el lexer se encarga de detectar los comentarios.

2.3. Curiosidades

Un detalle que no pudimos confirmar su procedencia, es el error de redondeo que causa nuestra implementación.

En un comienzo, creímos que se debía al uso de Int's en vez de Float's, pero finalmente vimos que no era esa la razón. También creímos que podía ser la precisión de PI, pero *bspline* no utiliza PI.

No logramos encontrar una explicación acorde para este problema.

3. Resultados

4. Gramática

El esquelto para hacer la gramática[1] fue tomado de la página citada en la presentación[2]. Sobre este esqueleto armamos nuestra gramática, que tiene varios puntos de diferencia con la que muestran de ejemplo en ese sitio.

[1] implementada en parser.y

[2] http://gnuu.org/2009/09/18/writing-your-own-toy-compiler/

Nuestra gramática parte de un símbolo incial *programa* que a su vez se divide en una lista de $funciones$ y una instrucción de *ploteo*.

Una función consiste en la palabra 'function' (representada por el token terminal TFUNC), un *nombre* (que no es más que un terminal que matchea con strings no empezados por números), una lista de $argumentos$ (lista de *nombre*s separadas por comas) entre paréntesis y un *bloque* de código.

Un *bloque* es una lista (quizás vacía) de sentencias encerradas entre llaves o una única sentencia.

Una *sentencia* puede ser una *asignacion* (un *nombre*, un símbolo de = y una *expresin*), una sentencia $ifthenelse$, un *while* (que a su vez tiene una *condicion* y otro *bloque*) o un $return$ (que lleva a su lado una *expresion*).

El caso del $ifthenelse$ requiere un análisis específico. Un $ifthenelse$ puede tener o no tener 'else', lo cual nos trae una ambiguedad a la hora de parsear cadenas del estilo (intencionalmente sin indentar):

```
if cond1 then
if cond2 then
codigo1
else
codigo2
```

que se refleja en el parser en un conflicto shift/reduce (cuando termina el codigo1 y viene un else puede tanto consumir el else y shiftear como reducir lo que ya tiene). Para resolverlo, nos basamos en el comportamiento de C en este caso (hacer shift, es decir, que el código se lea como

```
if cond1 then
  if cond2 then
    codigo1
  else
    codigo2
```

) y como Bison por defecto elije hacer siempre shift lo dejamos así.

Una *expresion* puede ser un *numero* (entero, double o π), una $llamada_funcion$ (el nombre de una función con más *expresiones* entre paréntesis como argumentos) u operaciones aritméticas entre expresiones. Para salvar las ambiguedades propias de esta parte de la gramática le indicamos a Bison la precedencia de los operadores[3].

Una *condicion* puede ser una comparación entre dos *expresion*es u otras condiciones con operadores lógicos.

Por último, la instrucción *ploteo* consiste en dos $llamada_funcion$ con un *nombre* de variable y tres *expresion*es que reflejan el inicio, el paso y el final.

La gramática se complementa con un archivo[4] con directivas para el tokenizador, que le permite convertir símbolos terminales en los tokens que usamos, ignora saltos de línea y maneja números enteros, floats y cadenas arbitrarias (como los nombres de variables).

Entre los problemas que encontramos hubo uno que no pudimos resolver: no logramos armar la expresión regular para que el tokenizador ignore los comentarios multilínea. Es por esto que para pasar los tests los archivos con comentarios debe ser en primer lugar pasados por un script hecho en bash que los borra.

[3] basándonos en el código de ejemplo de http://www-h.eng.cam.ac.uk/help/tpl/languages/flexbison/

[4] tokens.l

5. Ejemplos de funcionamiento

6. Respuestas

6.1. 1er Pregunta

Para verificar estáticamente que las condiciones de los if y los while sean booleanos se puede utilizar, en la gramática, un no terminal que represente únicamente operaciones booleanas (por ej comparación). De esta forma la gramática se asegura de que las condiciones sean booleanas y al momento de parsear el código se verifica el cumplimiento de ese requisito.

6.2. 2da Pregunta

El uso del ; en la gramática de C++ permite desambiguar código. Veamoslo con un ejemplo para dejarlo más claro:

---

int main()
int x = 0;
-f();
return 0;

Este es el código de un ejemplo básico de C++. Lo interesante es que, a diferencia de nuestro lenguaje, nos permite poner expresiones como si fueran instrucciones. Las llamadas a funciones en C/C++ son sentencias, a diferencia de nuestro lenguaje donde nuestras funciones no tienen efectos secundarios.

Es por esto que si quitamos los ; del código de ejemplo nos queda una ambigüedad. Veamos que sucede cuando los quitamos y notemos la ambigüedad que generaría en el lenguaje.

---

int main()
int x = 0
-f()
return 0

---

int main()
int x = 0 - f()
return 0

---

7. Manual

Modo de uso:

```
make clean && make
cat archivodetest.cod | ./borracomentarios.sed | ./parser | ./graficador.sh
```