Contents

1 La gramática

Esta es la gramática que proponemos para el lenguaje de las caritas:
$G = \langle \{\text{Carita, LuegoOjos, NarizYBoca, SubCarita}\}, \{\text{Ojos, Nariz, Boca, "o"}\}, \text{P}, \text{Carita} \rangle$
Donde P es:

- Carita → Ojos LuegoOjos
- LuegoOjos → SubCarita SubCarita NarizYBoca
- LuegoOjos → NarizYBoca
- NarizYBoca → Nariz Boca
- NarizYBoca → Boca
- SubCarita → Carita
- SubCarita → "o"

Decidimos agrupar los símbolos que representan Ojos, Nariz y Boca en tres terminales. Luego, el analizador sintáctico solo tendrá que entender de estos tres tokens, y accederá a sus nombres en un atributo.
Los valores posibles son:
Ojos = {: ; 8}
Nariz = {-}
Boca = {) ( ] [S P}

1.1 El lenguaje que genera

Para ver que genera el lenguaje pedido (sin cadenas de más o de menos), la vamos a construir paso por paso.

1. Partimos de una versión bastante declarativa que, creemos, muestra con claridad el lenguaje generado:

   Carita → Ojos Boca

   Carita → Ojos Nariz Boca

   Carita → Ojos o o Boca

   Carita → Ojos o o Nariz Boca

   Carita → Ojos o Carita Boca

   Carita → Ojos o Carita Nariz Boca

   Carita → Ojos Carita o Boca

   Carita → Ojos Carita o Nariz Boca

   Carita → Ojos Carita Carita Boca

   Carita → Ojos Carita Carita Nariz Boca

2. Sacando factor común "Ojos" es equivalente a:

   Carita → Ojos (Boca | Nariz Boca | o o Boca | o o Nariz Boca | o Carita Boca | o Carita Nariz Boca | Carita o Boca | Carita o Nariz Boca | Carita Carita Boca | Carita Carita Nariz Boca)

3. Remplazando la construcción disyuntiva por un nuevo símbolo LuegoOjos es equivalente a:

   Carita → Ojos LuegoOjos

   LuegoOjos → Boca | Nariz Boca | o o Boca | o o Nariz Boca | o Carita Boca | o Carita Nariz Boca | Carita o Boca | Carita o Nariz Boca | Carita Carita Boca | Carita Carita Nariz Boca

4. Desdoblando la producción "LuegoOjos" en dos, según haya subcaritas o no, queda:

   Carita → Ojos LuegoOjos

   LuegoOjos → o o Boca | o o Nariz Boca | o Carita Boca | o Carita Nariz Boca | Carita o Boca | Carita o Nariz Boca | Carita Carita Boca | Carita Carita Nariz Boca

   LuegoOjos → Boca | Nariz Boca

5. Sacando los cuatro factores comunes "o o", "o Carita", "Carita o" y "Carita Carita", es equivalente a:

   Carita → Ojos LuegoOjos

LuegoOjos → o o (Boca | Nariz Boca) | o Carita (Boca | Nariz Boca) | Carita o (Boca | Nariz Boca) | Carita Carita (Boca | Nariz Boca)

LuegoOjos → Boca | Nariz Boca

6. Y sacando factor común (Boca | Nariz Boca) es equivalente a:

   Carita → Ojos LuegoOjos

   LuegoOjos → (o o | o Carita | Carita o | Carita Carita) (Boca | Nariz Boca)

   LuegoOjos → Boca | Nariz Boca

7. Y luego reemplazando (Boca | Nariz Boca) por un nuevo no terminal "NarizYBoca" nos queda:

   Carita → Ojos LuegoOjos

   LuegoOjos → (o o | o Carita | Carita o | Carita Carita) NarizYBoca

   LuegoOjos → NarizYBoca

   NarizYBoca → Boca | Nariz Boca

8. Y luego sacando los dos factores comunes "o" y "Carita" nos queda:

   Carita → Ojos LuegoOjos

   LuegoOjos → (o (o | Carita) | Carita (o | Carita)) NarizYBoca

   LuegoOjos → NarizYBoca

   NarizYBoca → Boca | Nariz Boca

9. Y luego sacando otra vez factor común (o | Carita) nos queda:

   Carita → Ojos LuegoOjos

   LuegoOjos → (o | Carita) (o | Carita) NarizYBoca

   LuegoOjos → NarizYBoca

   NarizYBoca → Boca | Nariz Boca

10. Y creando un nuevo no terminal "SubCarita" en la gramática, para reemplazar los (o | Carita), nos queda:

    Carita → Ojos LuegoOjos

    LuegoOjos → SubCarita SubCarita NarizYBoca

    LuegoOjos → NarizYBoca

    NarizYBoca → Boca | Nariz Boca

    SubCarita → o | Carita

Y esta es exactamente nuestra gramática original.

1.2 La gramática es LL(1)

Calculamos los primeros de cada símbolo no terminal:

Primeros(Carita) = Primeros(Ojos LuegoOjos) = {Ojos}

Primeros(SubCarita) = Primeros(Carita) ∪ Primeros(o) = {Ojos, o}

Primeros(NarizYBoca) = Primeros(Nariz Boca) ∪ Primeros(Boca) = {Nariz, Boca}

Primeros(LuegoOjos) = Primeros(SubCarita SubCarita NarizYBoca) ∪ Primeros(NarizYBoca) = Primeros(SubCarita) ∪ {Nariz, Boca} = {Ojos, o, Nariz, Boca}

Como ningún símbolo es anulable, ningún lado derecho de producción lo es. Entonces, no necesitamos calcular Siguientes: los símbolos directrices (SD) de cada producción coinciden con los Primeros de cada lado derecho.

SD(Carita → Ojos LuegoOjos) = Primeros(Ojos LuegoOjos) = {Ojos}

SD(LuegoOjos → SubCarita SubCarita NarizYBoca) = Primeros(SubCarita SubCarita NarizYBoca) = Primeros(SubCarita) = {Ojos, o}

SD(LuegoOjos → NarizYBoca) = Primeros(NarizYBoca) = {Nariz, Boca}

SD(NarizYBoca → Nariz Boca) = Primeros(Nariz Boca) = {Nariz}

SD(NarizYBoca → Boca) = Primeros(Boca) = {Boca}

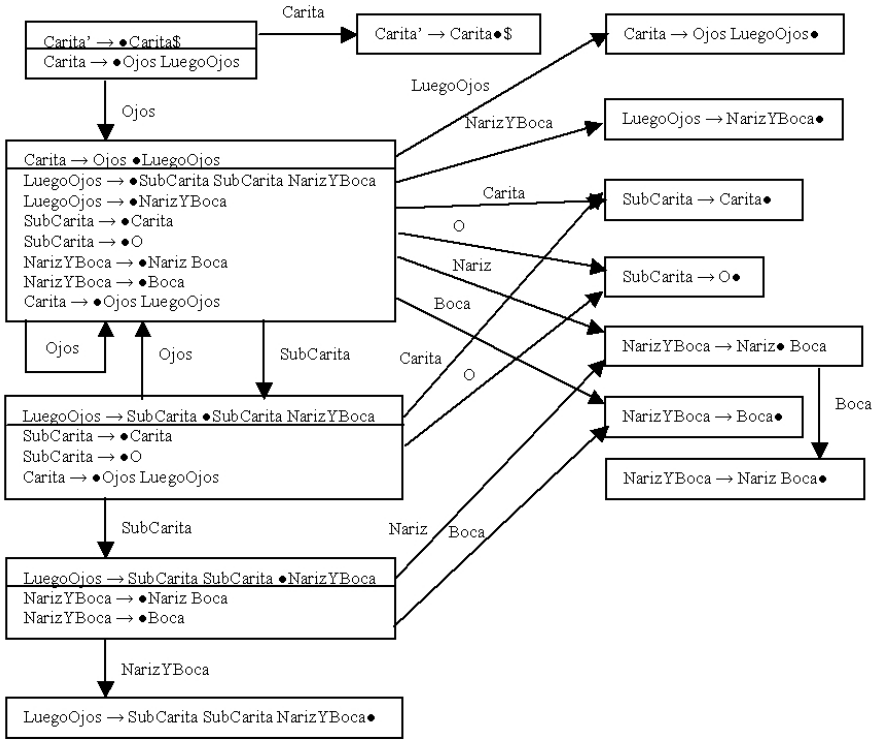
SD(SubCarita → Carita) = Primeros(Ojos LuegoOjos) = {Ojos}

SD(SubCarita → o) = {o}

Los símbolos directrices de cada par de producciones con un mismo no terminal a la izquierda son disjuntos. Entonces, la gramática es LL(1), y por ende, no es ambigua.

1.3 La gramática es LR(0)

En el siguiente autómata, todos los items completos quedan solos en sus estados, por lo que no puede haber conflictos Reduce/Reduce ni Shift/Reduce. Por lo tanto, la gramática es LR(0). No creemos que sea necesario construir la tabla.



2 Cómo se implementó la solución

Trabajamos en C, con el Flex para el análisis léxico y el Bison para el análisis sintáctico. Siendo la gramática LL(1) y LR(0), un analizador LALR como el Bison alcanza y sobra.

2.1 Flex

La entrada del lex (archivo "lex.lex") es la siguiente:

```
%{
#include "global.h"
#include "_y.h"

#include <stdlib.h>

extern int lineNumber;
```

```
%}

white [ \t]+

ojos [:;8]
nariz   [-]
boca [)(|\]\[PS]

%%

{white}  {
/* Los blancos se ignoran  */
}

{ojos} {
yylval = yytext[0];
return(OJOS);
}

{nariz} {
yylval = yytext[0];
return(NARIZ);
}


{boca} {
yylval = yytext[0];
return(BOCA);
}


"o" {
  return (CIRCULO);
}

"\n" {
  return(EOL);
}

. { // Cualquier otro caracter es un error léxico
printf("%d: caracter no válido '%c'\n", lineNumber, yytext[0]);
}
```

Y el contenido del archivo "global.h" es el siguiente:

```
#define YYSTYPE char
extern YYSTYPE yylval;
```

Como se puede ver, utilizamos patterns para agrupar los no terminales según fueran representaciones de bocas, narices u ojos.

El tipo que elegimos para el yylval es char. Cuando el Flex encuentra, por ejemplo, el caracter "P", devuelve un token "BOCA", pero almacena su valor "P" en yylval.

El pattern "white" corresponde a uno o más espacios blancos. El programa admite e ignora los blancos.

El "." atrapa cualquier caracter inválido. Esto es un error léxico y se maneja desde acá.

2.2 Bison

El archivo de entrada de Bison ("y.y") tiene cuatro secciones:

```
%{
  Prologue
%}
  Bison declarations
%%
  Grammar rules
%%
  Epilogue
```

En nuestro caso es un poco largo, por lo que lo vamos a mostrar por secciones

2.2.1 Prologue

En el Prologue implementamos una especie de pila para almacenar las representaciones de caritas. La codificación elegida es la siguiente:

1. Un código de OJOS es 1000 + el valor ascii del caracter.
2. Un código de NARIZ es 2000 + (el valor ascii del caracter, ó cero si no hay nariz)
3. Un código de BOCA es 3000 + el valor ascii del caracter.

La función "writePS" escribe un archivo postscript, agregando la pila al código en template_header.ps.

```
%{

#include "global.h"
#include <stdio.h>
```

```
#include <stdlib.h>
#include <math.h>

#define MAXSTACK 100
#define BASE_OJOS  1000
#define BASE_NARIZ 2000
#define BASE_BOCA  3000

int lineNumber = 0;
int stack[MAXSTACK];
int top = 0;

int push(int i) {
    // Apila el entero i
    stack[top] = i;
    top++;

}
int pushOJOS(char c) {
    // Apila el código de OJOS para el char c
    push( BASE_OJOS + (int) c);
}

int pushNARIZ(char c) {
    // Apila el código de NARIZ para el char c
    push ( BASE_NARIZ + (int) c);
}

int pushBOCA(char c) {
    // Apila el código de BOCA para el char c
    push ( BASE_BOCA + (int) c) ;
}

int reset() {
    // Vacía la pila
    top = 0;
}
int view() {
    // Muestra la pila
    int i;
    printf("\nSTACK = [ ");
    for(i=0; i<top;i++) {
        printf("%d ", stack[i]);
    }
    printf("] \n");
```

```
}

int writePS(int number) {
    /*
    Escribe la salida number-ésima con la pila actual
    */
    char *buffer;
    int lSize;
    FILE* outFile;
    int i;
    FILE* pFile;
    char filename[8] = "_000.ps";

    // Toma la parte fija de postscript de un template
    pFile = fopen("template_header.ps", "r");
    fseek (pFile , 0 , SEEK_END);
    lSize = ftell (pFile);
    if (lSize == -1){
        printf("No existe el archivo template_header.ps");
        exit(0);
    }
    rewind (pFile);

    buffer = (char*) malloc (lSize);
    if (buffer == NULL) {
        printf("Error");
        exit(0);
    }
    fread (buffer,1, lSize, pFile);


    if (number < 10) {
        filename[1] = '0';
        filename[2] = '0';
        filename[3] = (char) (48 + number);
    }
    else {
        printf("ERROR: máximo 9 pruebas\n");
        exit(0);
    }



    // Salida
    outFile = fopen(filename, "w");
```

```
    // Header
    fwrite(buffer, 1, lSize, outFile);

    // Stack
    for(i=top-1;i>=0;i--) {
        fprintf(outFile, "%d\n", stack[i]);
    }

    // Footer
    fprintf(outFile, "smile\n");
    fprintf(outFile, "end\n");
    fprintf(outFile, "showpage\n");


    fclose(outFile);
    fclose (pFile);
    free (buffer);

}


%}
```

2.2.2 Bison declarations

En Bison declarations tenemos los tokens. El símbolo distinguido es "Input".

```
%}

%token  OJOS
%token  NARIZ
%token  BOCA
%token  CIRCULO
%token  EOL


%start Input
%%
```

2.2.3 Grammar rules

En Grammar rules está la gramática descripta, pero formando parte de una gramática extendida. Decidimos soportar entradas sucesivas, con una expresión de carita por línea. Entonces, la gramática extendida comienza con "Input", que produce una secuencia de "Lines". Para cada "Line" que produce una "Carita" y un EOL, llamamos a writePS. Para las "Line" que producen un error (sintáctico) y un EOL, le informamos al usuario.

En las producciones de la gramática original, lo que hacemos es una TDS. Traduce el lenguaje de caritas a nuestra codificación numérica. Un tag "$k" representa el yylval del k-ésimo símbolo de una parte derecha. Cuando hay no terminales, es relevante en qué momento se apilan los valores.

Lo último que nos falta explicar sobre nuestra codificación son el 0 y el 1. Debe haber siempre dos de ellos en cada Carita de cualquier nivel. El primero se aplica al ojo izquierdo y el segundo al derecho. Un cero aquí significa que en ese ojo no hay subcarita, y un 1 indica lo contrario. Entonces, en esta codificación, un cero representa tanto a un ojo vacío explícito (una "o" en el lenguaje de las caritas) o implícito (en cualquier carita sin subcaritas).

```
%%

Input:
      /* Línea vacía */
    | Input Line
    ;

Line:
      EOL
    | Carita EOL            {
                        lineNumber++;
                        writePS(lineNumber);
                        view();
                        reset();
                        printf("%d: OK\n", lineNumber);
                    }
    | error EOL                     {
                        lineNumber++;
                        view();
                        reset();
                        printf("%d: ERROR SINTACTICO\n", lineNumber);
                        yyerrok;
                    }
    ;

Carita:
      OJOS {pushOJOS($1);} LuegoOjos
```

```
    ;

LuegoOjos:
      SubCarita SubCarita NarizYBoca
    | {push(0);push(0);} NarizYBoca
    ;

NarizYBoca:
      NARIZ BOCA                { pushNARIZ($1); pushBOCA($2);}
    | BOCA                     { pushNARIZ((char)0); pushBOCA($1);}
    ;

SubCarita:
      { push(1);} Carita
    | CIRCULO                 { push(0);}
    ;



%%
```

2.2.4 Epilogue

En Epilogue está el main para el código generado, que llama a yyparse, del Bison. Ésta última llama a yylex, del Flex.

```
%%

int yyerror(char *s) {
  printf("%s\n",s);
}

int main(void) {
    yyparse();
    printf("FIN\n");
}
```

2.3 Postscript

Los archivos de postscript generados se llaman _00N.ps, donde N es el número de línea de la entrada.

Tienen tres partes:

-Una parte fija, donde está definida la función "smile". Esto se lee de "template_header.ps"

-La pila de números que representa las caritas con nuestra codificación.
-Las instrucciones smile, end y showpage.

Hay una única función (smile), que es una extensión del ejemplo 2 del enunciado. No hace ninguna validación: confía en que la entrada en la pila es válida. Lo que hace es lo siguiente:

1. Dibuja un círculo
2. switch(OJOS):
   case X (para cada tipo de ojos X)
      Dibuja las particularidades del tipo de ojos X.
      Si en la pila hay un 1, cambia la escala y se llama recursivamente.
      Si no, dibuja un círculo vacío.
   ...
3. switch(NARIZ):
   case Y (para cada tipo de nariz Y)
      Dibuja el tipo de nariz Y
   ...
4. switch(BOCA):
   case Z (para cada tipo de boca Z)
      Dibuja el tipo de boca Z
5. Fin

Estos switches los implementamos anidando ifelses de este modo:

```
dup      // Antes de preguntar si la opción es Z1, duplicamos el tope por si no lo es
Z1 eq {
    pop      // Era Z1. Borramos la duplicación
    Dibujar      lo que indica Z1
}{
    // No era Z1. Hacemos lo mismo para ver si es Z2, y así sucesivamente.
    dup
    Z2 eq {
        pop
        Dibujar lo que indica Z2

        ...
        Zn eq {
            pop
            Dibujar lo que indica Zn
        }{
            Opción "default"
        } ifelse
        ...
    }{
    } ifelse
```

} ifelse

El contenido de "template_header.ps" es el siguiente:

```
%!PS-Adobe-3.0 EPSF-3.0
%%BoundingBox: 63 166 533 656
5 dict begin
/smile { gsave
    .25 setlinewidth
    0 0 4.5 0 360 arc closepath stroke % circulo exterior

    %%%%%%%%%%%%%%%% Ojos
    dup
    1058 eq {
        % case ":" = 1000 + ascii(":")
        pop
        gsave % ojo izquierdo
            -1.5 1.6 translate
            .2 .2 scale
            1 eq {
                smile
            }{
                0 0 4.5 0 360 arc closepath stroke
            } ifelse
        grestore
        gsave % ojo derecho
            1.5 1.6 translate
            .2 .2 scale
            1 eq {
                smile
            }{
                0 0 4.5 0 360 arc closepath stroke
            } ifelse
        grestore
    }{
        dup
        1059 eq {
            % case ";" = 1000 + ascii(";")
            pop
            gsave % ojo izquierdo
                -1.5 1.6 translate
                .2 .1 scale
                1 eq {
                    smile
```

```
            }{
                0 0 4.5 0 360 arc closepath stroke
            } ifelse
        grestore
        gsave % ojo derecho
            1.5 1.6 translate
            .2 .2 scale
            1 eq {
                smile
            }{
                0 0 4.5 0 360 arc closepath stroke
            } ifelse
        grestore
    }{
        % case "8" = "default". Lo que recibe actualmente es 1056
        pop
        gsave % ojo izquierdo
            -1.5 1.6 translate
            .2 .2 scale
            1 eq {
                smile
            }{
                0 0 4.5 0 360 arc closepath stroke
            } ifelse
        grestore
        gsave % ojo derecho
            1.5 1.6 translate
            .2 .2 scale
            1 eq {
                smile
            }{
                0 0 4.5 0 360 arc closepath stroke
            } ifelse
        grestore
        gsave
            .15 setlinewidth
            -0.6 1.6 moveto 1.2 0 rlineto stroke % nariz
        grestore

    } ifelse
} ifelse


.05 setlinewidth
%%%%%%%%%%%%%%%%%% Nariz
2045 eq {
```

```
    0 -0.5 moveto 0 1 rlineto stroke % nariz
}{
    % Sin nariz = 2000
} ifelse

%%%%%%%%%%%%%%%% Bocas
dup
3041 eq {
    0 0 2.75 210 330 arc stroke % boca )
    pop
}{
    dup
    3040 eq {
        0 -4 2.75 30 160 arc stroke % boca (
        pop
    }{
        dup
        3124 eq {
            pop
            -2 -2 moveto 4 0 rlineto stroke % boca |
        }{
            dup
            3091 eq { % boca [
                pop
                -2 -2 moveto 4 0 rlineto stroke
                -2 -2 moveto 0 -1 rlineto stroke
                2 -2 moveto 0 -1 rlineto stroke
            }{
                dup
                3093 eq { % boca [
                    pop
                    -2 -2 moveto 4 0 rlineto stroke
                    -2 -2 moveto 0 1 rlineto stroke
                    2 -2 moveto 0 1 rlineto stroke
                }{
                    dup
                    3080 eq { % boca P
                        pop
                        -2 -2 moveto 4 0 rlineto stroke
                        1 -2 1 180 360 arc closepath stroke

                    }{
                        dup
                        3083 eq { % boca S
                            pop
                            gsave
```

```
                                1 .7 scale
                                1 -3 1 180 10 arc stroke
                                -1 -3 1 0 190 arc stroke
                                grestore
                            }{
                                % sin boca
                            } ifelse
                        } ifelse
                    } ifelse
                } ifelse
            } ifelse
        } ifelse
    } ifelse

grestore } def

298 421 translate
50 50 scale
% Fin parte comun. Comienzo codigo generado
```

2.4 Ejemplos

2.4.1 Errores sintácticos

Ejemplos de errores sintácticos se pueden encontrar corriendo:

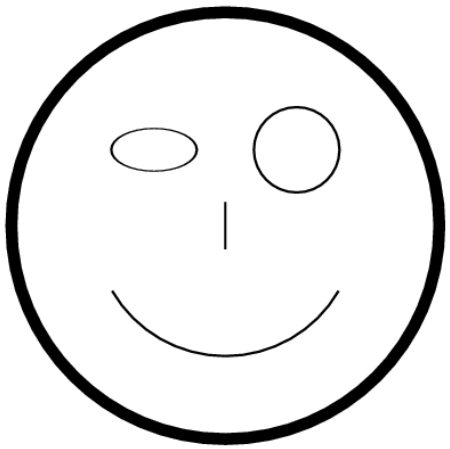
```
#./tp < ErroresSintacticos.txt
```

Entrada	Problema
:))	Doble boca
: : :-)]	Subcarita del ojo izquierdo incompleta
(:)	Una carita no puede empezar por la boca
; - - (	Doble nariz

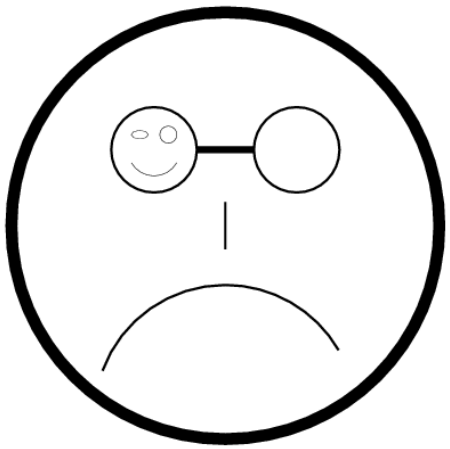
2.4.2 Entradas válidas

Estos ejemplos se puede reproducir corriendo:

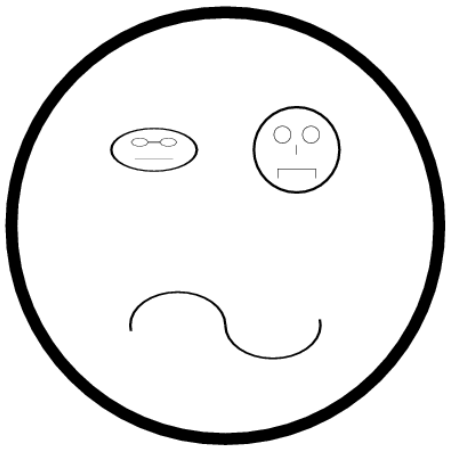
```
./tp < Validas.txt
```



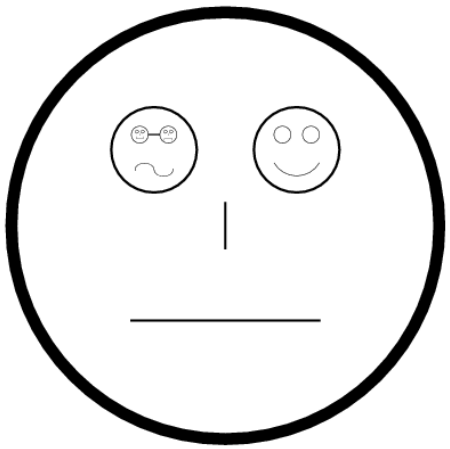
; -)



8 ;) o - (



; 8| :-[S



:-[S :) - |

3 Conclusiones

Es claro que la herramienta LALR sobra para parsear esta gramática. No habiendo requerimientos de performance, nos pareció más interesante aprender a utilizar el Bison que implementar un parser ad-hoc.

De esta forma, además, es muy sencillo agregar tipos de ojos, bocas y narices: basta con agregar los caracteres en lex.lex y los respectivos casos en template_header.ps.