Analizador Léxico

Universidad Tecnológica Nacional - Facultad regional Delta

Materia: Sintaxis y semántica de los lenguajes

Alumno: KUCHARUK, Manuel

Docentes: SANTOS, Juan, MIRANDA, Hernán

Fecha: 22/06/2015

Índice

Gramática

De todas las producciones dadas en la consigna del TP solamente hago las que corresponden a un lenguaje regular, ya que son las únicas que pueden ser programadas con autómatas de estado finito.

Simbolos terminales = { (,), {, }, +, *, -, O, Y, NO, A..Z, a..z, 0..9, VERDADERO, FALSO, Para, desde, hasta, Si, entonces, =, ==, <, >, >=, <=, ;, entero, real, logica, return}

Tipo de variable

Tipo → entero / real / logica

Nombre de variable

NombreVariable→ a NombreVariableCont / .. / z NombreVariableCont

NombreVariableCont → a NombreVariableCont / .. / z NombreVariableCont / l

Nombre de función

NombreFuncion → A NombreFuncionCont / .. / Z NombreFuncionCont / l

NombreFuncionCont → a NombreFuncionCont / .. / z NombreFuncionCont / l

Constante entera

ConstEntera → 1 ConstEnteraCont / .. / 9 ConstEnteraCont / 0

ConstEnteraCont → 0 ConstEnteraCont / .. / 9 ConstEnteraCont / l

Constante lógica

ConstLogica → VERDADERO / FALSO

Operadores

Operador→ == / > / < / >= / <=

Si una expresión satisface las condiciones de la producción se escribe el identificador entre los símbolos “<” y “>”. Por ejemplo:

- var → <NombreVariable>
- F → <NombreFuncion>
- 100 → <ContanteEntera >

Las palabras reservadas (símbolos terminales) que no coinciden con las producciones se escriben tal cual entre “<” y “>”. Por ejemplo:

- Para → <Para>
-) → <)>

Código del programa (lexico.l)

```
%{
#include <string.h>
#include <stdlib.h>
char* limpiarUltimo(char* str){
  int longStr=strlen(str);
  char* copia=(char*) malloc((longStr+1)*sizeof(char));
  strncpy(copia,str,longStr-1);
  copia[longStr-1]='\0';
  return copia;
}
%}
delimID [-+{}()><=;,* \t\n]
%%
%{/*Palabras reservadas y simbolos que pasan tal cual estan escritos*/
%}
("Para"|"desde"|"hasta"|"Si"|"entonces"|"return"|"NO"|"Y"|"O"){delimID}
{printf("<%s>",limpiarUltimo(yytext)); unput(yytext[yyleng-1]);}
[-+{}()=;,*] printf("<%s>",yytext);

%{/*Palabras reservadas de tipo*/
%}
("entero"|"real"|"logica")[ \t] printf("<Tipo>");

%{/*Constantes*/
%}
([1-9][0-9]*|0){delimID} {printf("<ConstEntera>"); unput(yytext[yyleng-1]);}
("VERDADERO"|"FALSO"){delimID} {printf("<ConstLogica>"); unput(yytext[yyleng-
1]);}

%{/*Identificadores*/
%}
[a-z]+{delimID} {printf("<NombreVariable>",yytext); unput(yytext[yyleng-1]);}
[A-Z][a-z]*{delimID} {printf("<NombreFuncion>"); unput(yytext[yyleng-1]);}

%{/*Operadores*/
%}
"=="|">="|"<="|">"|"<" printf("<Operador>");

%{/*Ignorar espacios, tabulaciones y saltos de linea*/
%}
[ \t\n]

%{/*Otros caracteres dan errores y se termina la ejecucion*/
%}
. {printf("ERROR"); exit(0);}
```

```
%%

int yywrap(void){return 1;}

int main(int argc, char *argv[]) {
 yyin = fopen("programa.txt", "r");
 freopen("salida.txt", "w", stdout);
 yylex();
 fclose(yyin);
 }
```

Explicación, observaciones y comentarios

Para hacer el analizador léxico usé el lenguaje LEX, que traduce mediante el compilador flex el código a un programa en lenguaje C, que luego debe ser compilado (por ejemplo con gcc) y ejecutado.
LEX brinda la posibilidad de escribir mediante expresiones regulares los requerimientos del lenguaje, también cuenta con un escáner para leer el archivo de entrada que es transparente al programador.
El código generado (lex.yy.c) cuenta con un autómata de estado finito (implementado con *switch..case*) y todo lo necesario para leer el archivo, coincidir las expresiones regulares y escribir en el archivo de salida.

Explicación del código:

```
%{
#include <string.h>
#include <stdlib.h>
char* limpiarUltimo(char* str){
  int longStr=strlen(str);
  char* copia=(char*) malloc((longStr+1)*sizeof(char));
  strncpy(copia,str,longStr-1);
  copia[longStr-1]='\0';
  return copia;
}
%}
```

El código que está entre `%{` y `}%` se transfiere al archivo lex.yy.c tal como está. En este caso es una función que, dada una cadena devuelve una copia sin el último caracter.

```
delimID [-+{}()><=;,* \t\n]
```

En este analizador intento cumplir con las producciones dadas por la gramática, pero en caso de dudas, intento que la gramática aceptada se parezca lo más posible al lenguaje C.
Por ejemplo, las cadenas `a = b ;` o `a=b;` generarán la misma salida (como en C). Para esto utilizo una expresión regular que coincida con los limitadores “aceptados”: `delimID`.
Definición de `delimID`: En vez de escribir la expresión de la derecha en las reglas de sustitución (se explican a continuación), es posible escribir `{delimID}`, simplificando la lectura.
En particular, al escribir `{delimID}` en las reglas de sustitución, una cadena satisface la expresión regular cuando coincide con uno (cualquiera) de los caracteres dentro de los corchetes.

%%
División de bloque, de aquí hasta la próxima división serán reglas de sustitución
Todas las reglas de sustitución tienen la forma `regla acción`, siendo regla una expresión regular y `acción` una porción de código C que se ejecuta en caso de que una cadena cumpla con `regla`

```
("Para"|"desde"|"hasta"|"Si"|"entonces"|"return"|"NO"|"Y"|"O"){delimID}
{
printf("<%s>",limpiarUltimo(yytext));
unput(yytext[yyleng-1]);
}
```

La expresión regular se cumple cuando la cadena de entrada tiene alguna palabra reservada (Para, desde, etc.) seguida de un delimitador.
La acción que se ejecuta:
`printf("<%s>",limpiarUltimo(yytext));`
Imprime la cadena leída (`yytext`) sin el último carácter (el delimitador) dentro de "<" y">".

`unput(yytext[yyleng-1]);`
Devuelve a la entrada el último caracter leído (el delimitador) para que pueda ser leído nuevamente por el escáner.

`[-+{}()=;,*] printf("<%s>",yytext);`
Al encontrar cualquiera de los caracteres entre corchetes, lo imprime entre "<" y ">"

`("entero"|"real"|"logica")[ \t] printf("<Tipo>");`
Al encontrar un nombre de tipo seguido por un espacio o tabulación, imprime `<Tipo>`

`([1-9][0-9]*|0){delimID} {printf("<ConstEntera>"); unput(yytext[yyleng-1]);}`
La expresión regular se satisface al encontrar una cadena que empieza por un número del 1 al 9 (`[1-9]`) y luego tiene una cantidad cualquiera (0 o más) de números del 0 al 9 (`[0-9]*`) seguidos de un delimitador.
También se satisface cuando hay un único 0 seguido de un delimitador.
Esta última expresión (la del 0) cumple con la producción **ConstEntera** $\rightarrow$ 0. Pero no permite escribir cualquier cantidad de 0. Por ejemplo, la cadena 0000 daría un error.
Al encontrar la expresión regular, imprime `<ConstEntera>` y devuelve el delimitador a la entrada.

`("VERDADERO"|"FALSO"){delimID} {printf("<ConstLogica>"); unput(yytext[yyleng-1]);}`
Si la cadena es `VERDADERO` o `FALSO` seguido de un delimitador. Imprime `<ConstLogica>` y devuelve el delimitador a la entrada.

`[a-z]+{delimID} {printf("<NombreVariable>",yytext); unput(yytext[yyleng-1]);}`
La expresión regular se satisface al encontrar una cadena con una o más letras minúsculas de la "a" a la "z", seguida de un delimitador. Imprime `<NombreVariable>` y devuelve el delimitador a la entrada.

`[A-Z][a-z]*{delimID} {printf("<NombreFuncion>"); unput(yytext[yyleng-1]);}`
La expresión regular se satisface al encontrar una cadena con que empieza con una letra mayúscula `[A-Z]` y luego tiene 0 o más letras minúsculas `[a-z]*`, seguida de un delimitador.
Imprime `<NombreFuncion>` y devuelve el delimitador a la entrada.

```
"=="|">="|"<="|">"|"<" printf("<Operador>");
```
Al encontrar cualquier cadena de la izquierda imprime `<Operador>`.
En esta regla se ve la importancia de devolver el último caracter leído a la entrada, ya que por ejemplo el símbolo = funciona como delimitador y como primer símbolo de un operador.
Si no se hubiera devuelto, la expresión `a==b` hubiera sido traducida a `<NombreVariable><=><NombreVariable>` (asignación) en vez de a `<NombreVariable><Operador><NombreVariable>` (comparación).

```
[ \t\n]
```
Al encontrar un espacio, tabulación o salto de línea no se ejecuta ninguna acción, solo se sigue leyendo la entrada.

```
. {printf("ERROR"); exit(0);}
```
Al encontrar cualquier otra cadena que no coincida con las expresiones regulares anteriores, se imprime la leyenda ERROR y se corta la ejecución del programa.
LEX prioriza (entre otras cosas) por orden de declaración de la regla. Esta última regla debe ir al final, ya que en coincide con cualquier carácter (excepto salto de línea) y, si se encuentra en otro lado, podría abortar la ejecución del programa innecesariamente.

```
%%
```
División de bloque, de aquí hasta el final se transcribe exactamente al archivo lex.yy.c

```
int yywrap(void){return 1;}
```
Se usa cuando hay múltiples archivos de entrada. En este caso con devolver 1 el programa compila sin errores.

```
int main(int argc, char *argv[]) {
 yyin = fopen("programa.txt", "r");
 freopen("salida.txt", "w", stdout);
 yylex();
 fclose(yyin);
 }
```
Programa principal
abre archivos de entrada
abre archivo de salida, asocia salida estándar
ejecuta el analizador léxico
cierra archivo entrada

Ejemplo de ejecución

entrada.txt

```
entero sum;
entero i;
logica log;

logica Soniguales(entero a, real b){
  logica iguales;
  iguales=FALSO;
  Si (a-b)==0 O (b-a)==0 entonces{
    iguales=VERDADERO;
  }
  return iguales;
}

sum=0;
Para i desde 0 hasta 15{
  Si i>=5 entonces{
    sum=sum+1;
  }
}

Mostrar(Soniguales(3,4));
```

salida.txt (saltos de línea agregados)

```
<Tipo><NombreVariable><;>
<Tipo><NombreVariable><;>
<Tipo><NombreVariable><;>

<Tipo><NombreFuncion><(><Tipo><NombreVariable><,><Tipo><NombreVariable><)><{>
  <Tipo><NombreVariable><;>
  <NombreVariable><=><ConstLogica><;>
  <Si>
     <(><NombreVariable><-><NombreVariable><)><Operador><ConstEntera>
     <O>
     <(><NombreVariable><-><NombreVariable><)><Operador><ConstEntera>
     <entonces><{>
       <NombreVariable><=><ConstLogica><;>
  <}>
  <return><NombreVariable><;>
<}>

<NombreVariable><=><ConstEntera><;>
<Para><NombreVariable><desde><ConstEntera><hasta><ConstEntera><{>
  <Si><NombreVariable><Operador><ConstEntera><entonces><{>
    <NombreVariable><=><NombreVariable><+><ConstEntera><;>
  <}>
<}>

<NombreFuncion><(><NombreFuncion><(><ConstEntera><,><ConstEntera><)><)><;>
```

Conclusión

Intenté hacer el analizador léxico con un programa en C que lea línea por línea, pero son tantas las excepciones que luego intenté hacer uno que lea carácter por carácter usando switch..case para simular el funcionamiento de un AFD.

Es muy complicado, hasta no saber el final de cada palabra no se puede saber de qué tipo es (por ejemplo la palabra reservada `Para` y la función `Parametro` no se puede diferenciar hasta encontrar la letra `m`). También es difícil implementar correctamente los delimitadores (por ejemplo, para que `a=b;` sea lo mismo que `a = b ;`).

LEX proporciona las herramientas necesarias para hacer esto (lector del archivo de entrada, coincidencia de expresiones regulares, etc.) de una manera más simple y más fácil de leer para el programador.

El resultado del análisis léxico debe luego ser enviado a un analizador sintáctico para poder compilar el programa.