Tarea de investigación de Lex y Yacc

Lex es un generador de programas para el análisis léxico de cadenas de caracteres, produce un programa en lenguaje C que reconoce expresiones regulares las cuales son especificadas en el programa Lex, estas mismas son reconocidas en la entrada de datos y particiona las entradas en cadenas que coincides con las expresiones especificadas.

No es un lenguaje completo, es un generador que representa una mejora de idioma que se puede agregar a diferentes lenguajes de programación, llamados “lenguajes receptores”. Puede escribir código en diferentes lenguajes receptores. El lenguaje receptor se usa para el código de salida generado por Lex y también para los fragmentos de programa agregados por el usuario.

El programa Lex genera un llamado ‘Lexer’. Esta es una función que toma una secuencia de caracteres como su entrada, y cada vez que ve un grupo de caracteres que coinciden con una palabra clave, toma una determinada acción.

Yacc proporciona una herramienta general para imponer estructura en la entrada a un programa de computadora. El usuario de Yacc prepara una especificación del proceso de entrada; esto incluye reglas que describen la estructura de entrada, el código que se invocará cuando se reconocen estas reglas, y una rutina de bajo nivel para hacer la entrada básica.

Yacc luego genera una función para controlar el proceso de entrada. El analizador sintáctico llama a la rutina de entrada de bajo nivel suministrada por Lex para recoger los tokens del flujo de entrada, están organizados por reglas de gramática, cuando se ha reconocido una de estas reglas, se invoca el código de usuario proporcionado para esta regla, una acción; las acciones tienen la capacidad de devolver valores y hacer uso de los valores de otras acciones.

El programa Yacc puede analizar flujos de entrada que consisten en tokens con ciertos valores. Esto describe claramente la relación que Yacc tiene con Lex, Yacc no tiene idea de qué son las 'corrientes de entrada', necesita tokens preprocesados que de eso se encarga Lex. Si bien puedes escribir tu propio Tokenizer, lo dejamos por completo a Lex.

En el primer código

La primera sección, entre el par %{ y %}, se incluye directamente es para la librerías que se requieran en el programa de salida.

Las secciones se separan usando '%%', por lo que la primera línea de la segunda sección comienza con el palabras claves ‘entero’, dos gramáticas, dos símbolos y dos palabras especiales.

Usé retornar para alimentar a YACC. Y.tab.h tiene definiciones para estos tokens.

Terminamos la sección de código con '%%' de nuevo.

En el segundo código

En la primera parte, entre el par %{ y %} tenemos la función yyerror ( ) que es invocada por YACC si encuentra un error. Simplemente emitimos el mensaje pasado.

La función yywrap ( ) puede usarse para continuar leyendo desde otro archivo. Se llama en EOF y puede abrir otro archivo y devolver 0. O puede devolver 1, lo que indica que este es realmente el final.

Luego está la función main ( ), que no hace más que poner todo en movimiento.

La última línea simplemente define los tokens que usaremos. Estos se envían usando y.tab.h si se invoca YACC con la opción ‘-d'.

La segunda parte tenemos ‘commands', y que estos comandos consisten en partes individuales de 'command'. Esta regla es recursiva, porque nuevamente contiene la palabra 'commands'. Lo que esto significa es que el programa ahora es capaz de reducir una serie de comandos uno por uno.

La segunda regla define qué es un comando. Solo admitimos dos tipos de comandos, el 'creacion_entero' y el 'asignacion_palabra'. Esto es lo que significa el símbolo | - 'un comando consiste en un creacion_entero o un asignacion_palabra'.

Un creacion_entero consiste en el token ENTERO, que es la palabra 'entero', seguido de una PALABRA que es aceptada por la gramática (que definimos en el archivo Lex) y un SEMICOLON.

Algo más complicado es el asignacion_palabra, que consiste en el token PALABRA, el token IGUAL y seguido de un NUMERO que es aceptado por la gramática (que definimos en el archivo Lex) y un SEMICOLON.

lexyacc.l

```
%{
#include <stdio.h>
#include "y.tab.h"
%}
%%
entero              return ENTERO;
[0-9]+              yylval=atoi(yytext); return NUMERO;
[a-zA-Z][a-zA-Z0-9]*    return PALABRA;
;                   return SEMICOLON;
=                   return IGUAL;
\n                  /* ignore end of line */;
[ \t]+              /* ignore whitespace */;
%%
```

lexyacc.y

```
%{
#include <stdio.h>
#include <string.h>
void yyerror(const char *str) {
    fprintf(stderr,"error: %s\n",str);
}
int yywrap() {
    return 1;
}
int main() {
    yyparse();
}
%}
%token NUMERO PALABRA ENTERO SEMICOLON
IGUAL

%%
commands: /* empty */
    | commands command
    ;
command:
    creacion_entero
    |
    asignacion_palabra
    ;
creacion_entero:
    ENTERO PALABRA SEMICOLON
    {
        printf("\tCreacion de entero!\n");
    }
    ;
asignacion_palabra:
    PALABRA IGUAL NUMERO SEMICOLON
    {
        printf("\tAsignacion de palabra a %d\n", $3);
    }
    ;
%%
```

Bibliografía:

Hubert, B. (septiembre 20, 2004). *Lex & YACC HOWTO*. [en línea] Power DNS.
Disponible en: https://ds9a.nl/lex-yacc/ [Recuperado septiembre 5, 2018].

Lesk, M., Schmidt, E., & Johnson, S. (s.f.). *The LEX & YACC Page*. [en línea] Compiler Tools.
Disponible en: http://dinosaur.compilertools.net/ [Recuperado septiembre 5, 2018].