Instituto Politécnico Nacional
Escuela Superior de Cómputo
Ingeniería en Sistemas Computacionales

Compiladores

Práctica 4: Desarrollo de un analizador léxico especificado por medio de expresiones regulares

Profesor: Rafael Norman Saucedo Delgado

Alumno: Ayala Segoviano Donaldo Horacio

Boleta: 2019630415

Introducción

Descripción del proyecto

El objetivo del proyecto es crear un compilador, que reciba lenguaje *mipl* y entregue ensamblador para el procesador *escomips* que está siendo desarrollado en la materia de arquitectura de computadoras.

Actualmente el procesador debe ser programado en un lenguaje de bajo nivel, es decir, ensamblador específico para esa arquitectura de procesador. Lo que este proyecto busca es crear un lenguaje de programación de alto nivel, que facilite el proceso de programación del procesador mencionado.

Dado que el procesador es creado con fines didácticos, puede realizar operaciones limitadas, es por esto que el lenguaje *mipl* no será demasiado complejo.

Arquitectura MIPS

Con el nombre de MIPS (siglas de Microprocessor without Interlocked Pipeline Stages) se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

Este tipo de procesador ha sido utilizado en distintos sistemas, como sistemas embebidos. Esto debido a que este tipo de procesadores son la base de los procesadores superescalares actuales.

Procesadores Superescalares

Los procesadores superescalares son capaces de ejecutar más de una instrucción por ciclo de reloj, ya que hace uso del paralelismo de instrucciones usando técnicas de *pipeline* en etapas: búsqueda de instrucción, decodificación, ejecución y acceso a memoria.

Flex

'flex' es una herramienta para generar "escáneres". Un escáner es un programa que reconoce patrones léxicos en el texto. El programa 'flex' lee el archivos de entrada dados (o de su entrada estándar si es que no se dan nombres de archivo), y busca una descripción de un escáner para generarlo. La descripción está en la forma de pares de expresiones regulares y código C, llamados "reglas". 'flex' genera como salida un archivo fuente en C, 'lex.yy.c' por defecto, que define

una rutina 'yylex ()'. Este archivo se puede compilar y vincular con la biblioteca de tiempo de ejecución flexible para producir un ejecutable. Cuando el ejecutable se ejecuta, analiza las entradas en busca de concordancias con las expresiones. Siempre que encuentra una concordancia, ejecuta el correspondiente C código.

Para el desarrollo del analizador léxico que se implementará en la presente práctica, se usará la versión 2.6.4 de flex.

Desarrollo

1. Ejemplificación del lenguaje

A continuación, se muestra el ejemplo que se tomó de base para la elaboración del analizador léxico. En el ejemplo se muestra la mayoría de la sintaxis que se busca implementar en el lenguaje *mipl.*

```
/*
   will work as comments section
   !!  Functions will be optional (Until knowing it is
      possible to implement in MIPS assembly this
      will be an idea only)

   The syntax to create a function is the following:
   function function_name ( reg parameter , ... )  {

   }
*/

reg min ( reg a, reg b ) {
   if ( a < b ) {
      return a;
   } else {
      return b;
   }
}

main {
   reg register1 = 3;
   reg register2 = 7;
   reg register3 = register1 + register2;
   /*
        Hexadecimal constants must have a
        length of between 1 and 4 charachters
    */
   reg register4 = 0x00F2;

   if ( register1 < register2 ) {
      register2 = register2 - 2;
   } else {
      register2 = register2 + 3;
   }
   reg register5 = min( register1, register2 );

   while ( register2 > 0 ) {
      register2 = register2 - 1;
   }

   register1 = register1 & register2;
   register1 = register1 | register2;
   register1 = register1 ^ register2;
   register1 = ~ register2;
   register1 = register1 >> 2;
   register1 = register1 << 2;
   register1 = register1 * register2;
   register1 = register1 | register2;
   register1 = register1 + register2;
   register1 = register1 - register2;
   register1 = register1 / register2;
   if ( register1 <= register2 ) {
      register1 = ~register1;
   }
   if ( register1 >= register2 ) {
      register1 = ~register1;
   }
   if ( register1 < register2 ) {
      register1 = ~register1;
   }
   if ( register1 > register2 ) {
      register1 = ~register1;
   }
   if ( register1 <= register2 ) {
      register1 = ~register1;
   }
   if ( register1 != 3 ) {
      register1 = register1 + 1;
   }
   mem[50] = 20;
   mem[ registro3 + 3 ] = 0xAF;


}
```

2. Identificación de clases léxicas

Un punto importante durante el desarrollo de la presente práctica es la identificación de clases léxicas. Debido a que, en un futuro, la clasificación de los tokens en categorías más específicas facilita el proceso del desarrollo de un compilador, se ha decidido crear clases léxicas más específicas para poder identificar más fácilmente cada elemento en un futuro. A continuación, se muestran las clases léxicas identificadas.

- Clases léxicas

- main_block
- data_type
- control
- memory_acces
- delimiter
- eoi (end of line)
- separator
- bitwise_operation
- arithmetical_operation
- comparison
- assign
- constant

3. Expresiones regulares para cada clase léxica

Para la simplificación de la escritura de las expresiones regulares, se hará uso de definiciones regulares, mostradas a continuación:

- Definiciones regulares:

- letters [A-Za-z]
- digits [0-9]
- spaces [\t]
- hex_digits [0-9A-Fa-f]
- sign [+-]
- ari_operators [+-/*]
- bit_operators [&~^|]
- delimiters [()\[\]{}]

- separators [,]

Para describir las expresiones regulares se usarán algunos de los operadores mostrados en el manual de flex.

Expresiones regulares	Clase léxica
“main”	main_block
“reg”	data_type
“if”|”else”|”while”|”return”	control
“mem”	memory_acces
{delimiters}	delimiter
“;”	eoi
{bit_operators}|”<<”|”>>”	bitwise_operation
{ari_operators}	aritmetical_operation
“<=”|“>=”|“<”|“>”|“==”|“!=”	comparison
“=”	assign
({sign}?{digits}*)|(0x{hex_digits}{1,4})	constant
{letters}({letters}|{digits}|"_")*	Id

4. Codificar en LEX

Ahora se procederá a codificar el archivo lex, que definirá el analizador léxico para nuestro compilador, usando las clases y expresiones regulares mencionadas anteriormente.

Para la primera sección, se define lo siguiente:

```
%{
    /* C Language Definitions */
    #include <stdio.h>
    static void comment(void);
%}
/*language definitions*/
letters                 [A-Za-z]
digits                  [0-9]
spaces                  [ \t]
hex_digits              [0-9A-Fa-f]
sign                    [+-]
ari_operators           [+-/*]
bit_operators           [&~^|]
delimiters               [()\[\]{}]
separators               [,]
```

Figura 1. Definiciones del lenguaje y definiciones regulares.

En la figura 1 se pueden observar las definiciones del lenguaje, como las cabeceras y las definiciones del lenguaje. En la segunda sección se definen las expresiones regulares con su correspondiente código de C.

```
"main"                          { printf("<main_block>"); }
"/*"                            { comment(); }
"reg"                           { printf("<data_type>"); }
"if"|"else"|"while"|"return"    { printf("<control>"); }
"mem"                           { printf("<memory_access>"); }
{delimiters}                    { printf("<delimiter>"); }
";"                             { printf("<eoi>"); /*End of instruction*/ }
{separators}                    { printf("<separator>"); }
{bit_operators}|"<<"|">>"       { printf("<bitwise_operation>"); }
{ari_operators}                 { printf("<aritmetical_operation>"); }
"<="|">="|"<"|">"|"=="|"!="     { printf("<comparison>"); }
"="                             {printf("<assign>");}
({sign}?{digits}*)|(0x{hex_digits}{1,4})  { printf("<constant>"); }
{letters}({letters}|{digits}|"_")* { printf("<id>"); }
```

Figura 2. Expresiones regulares con su correspondiente código en C

.

Por último, en la última sección del archivo, se definen algunas funciones auxiliares para el analizador léxico (véase figura 3). Una de ellas es la de *comment()* la cuál solo omite las secciones de comentarios definidas de la siguiente manera: */*comentario*/.*

```
/* User Code Section */
static void comment(void) {
    int c;
    while ((c = input()) != 0) { // Until end of file
        if (c == '*') {
            if ((c = input()) == '/') // Found "*/"
                return;
            if (c == 0) // End of file
                break;
        }
    }
}
```

Figura 3. Definición del código de C.

Una vez teniendo el código fuente de nuestro archivo, lo guardaremos, en este caso, con el nombre "*mipl_lex.l*".

5. Pruebas

Una vez teniendo el archivo "*mipl_lex.l*" se procederá a probarlo, para esto, se creó un archivo *Makefile* que nos facilitará la ejecución de los comandos

repetitivos para probar el analizador generado. A continuación se muestra el archivo *Makefile:*

```
lex.yy.c: mipl_lex.l
	flex mipl_lex.l

lex.yy.o: lex.yy.c
	gcc -c lex.yy.c

main.o: main.c
	gcc -c main.c

a.out: main.o lex.yy.o
	gcc main.o lex.yy.o -lfl

clean:
	rm -f main.o lex.yy.o a.out lex.yy.c run.log

run: a.out
	./a.out

test: a.out test.in
	echo "\n--MIPS HIGH LEVEL LANGUAGE--\n" > run.
log && cat test.in >> run.log && echo "\n---TOKENS---\n" >
> run.log && ./a.out < test.in >> run.log && cat run.log |
 less
```

Figura 4. Contenido del archivo Makefile.

En la figura 4, se puede observar el contenido del archivo *Makefile*, y para ejecutar el analizador, primero debemos ejecutar el comando "*make*" que ejecutará los primeros cuatro comandos del archivo *Makefile,* después de esto, se puede ejecutar el comando "*make clear*" que eliminará los archivos que no necesitamos. Y para probar el analizador léxico, tenemos dos formas de hacerlo, la primera de ellas es ejecutar el comando "*make run*" que permitirá introducir desde la entrada estándar texto para el analizador léxico pueda procesarlo.

La segunda forma de ejecutarlo, es mediante el comando "*make test*" el cual, tomará el contenido del archivo "*test.in*" que contiene el código fuente ejemplo de nuestro lenguaje de programación *mipl.* Después de ejecutarlo, nos llevara a la interfaz usada por el comando "*less*" en consola, donde se mostrará el contenido del archivo "*run.log*" que es donde se guardará la salida de la ejecución del analizador léxico una vez que este procese el archivo "*test.in*".

El contenido del archivo *run.log* está dividido en dos secciones: en la primera parte se muestra el código escrito en el lenguaje *mipl* y en la segunda parte se muestra la salida de nuestro analizador léxico cuando le proporcionamos el código mostrado en la primera parte como entrada. Para navegar se usan los botones "*arriba*" y "*abajo*".

Con la salida de nuestro analizador, podemos observar que todo funciona correctamente, el analizador está clasificando el texto correctamente de acuerdo a las expresiones regulares definidas.

Conclusiones

Después de la realización de está práctica, se reforzaron los conocimientos acerca del manejo de flex, y se pudo comprobar el funcionamiento de nuestro analizador.

Se llegó a la conclusión de que el analizador léxico de esta práctica será de ayuda para el posterior desarrollo de nuestro compilador. También se llegó a la conclusión de que flex es una herramienta muy útil para el desarrollo de compiladores.

Referencias

- Aho A., Lam M. (2008) *Compiladores, principios técnicas y herramientas. Segunda edición*. México: Pearson Education.
- Estes W., Paxson V., Millway J. (May 2017) *The flex Manual*.
- Parhami B. (2005) *Computer Architecture: from microprocessors to supercomputers.* USA: Oxford University Press.