Tecnológico de Monterrey Campus Querétaro

Compiladores EM 2013
Profesor Dr Carlos Ramirez

Fernando Isaac Mendoza Quiñones A01200330
Arturo Jamaica García A00888285
Andrés Camargo Bravo A00888584

LuaC : Compilador de Lua a Ensamblador de 8051

Introducción

Lua es un lenguaje de programación extensible diseñado para una programación procedimental general con utilidades para la descripción de datos. También ofrece un buen soporte para la programación orientada a objetos, programación funcional y programación orientada a datos. Se pretende que Lua sea usado como un lenguaje de script potente y ligero para cualquier programa que lo necesite. Lua está implementado como una biblioteca escrita en C.

Las palabras reservadas en Lua pueden ser cualquier tira de caracteres sólo con letras, dígitos y _, no comenzando por un dígito. Estas son :

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until     while
```

Los nombres que comienzan por un guión bajo seguido por letras en mayúsculas están reservados para uso como variables globales internas de Lua.

Los **strings** pueden ser delimitados por comillas simples (apóstrofes) o dobles, y pueden contener las secuencias de *escape* de C. Es decir \n, \t, \r, \f, \b. Los strings pueden definirse usando un formato largo, encerrados en corchetes. Definimos un corchete largo de abrir de nivel *n* como un corchete de abrir ([) seguido de *n* signos igual (=) seguidos de otro corchete de abrir. Por ejemplo :

```
a = [[alo
     123"]]
```

y tambien :

```
a = [==[
     alo
     123"]==]
```

Los números pueden contener una parte decimal opcional y también un exponente opcional. Lua también acepta constantes enteras hexadecimales, escritas anteponiendo el prefijo 0x. Algunos ejemplos de constantes numéricas válidas son :

```
3         3.0         0x4344443
```

Lua es un lenguaje dinámicamente tipado. Esto quiere decir que aunque los valores tengan su tipo específico, las variables no lo tienen. De esta forma, es posible que una variable referencie a un valor de cualquier tipo.

Gramatica Oficial

A continuación se listan todas las derivaciones de la gramática oficial de Lua. A partir de esta gramática se seleccionarán las derivaciones que cumplen con el alcance y acotación del proyecto y se ajustarán o eliminarán aquellas que excedan estos límites.

```
%fallback  OPEN '(' .

chunk      ::= block .

semi       ::= ';' .
semi       ::= .

block      ::= scope statlist .
block      ::= scope statlist laststat semi .
ublock     ::= block 'until' exp .

scope      ::= .
scope      ::= scope statlist binding semi.

statlist   ::= .
statlist   ::= statlist stat semi .

stat       ::= 'do' block 'end' .
stat       ::= 'while' exp 'do' block 'end' .
stat       ::= repetition 'do' block 'end' .
stat       ::= 'repeat' ublock .
stat       ::= 'if' conds 'end' .
stat       ::= 'function' funcname funcbody .
stat       ::= setlist '=' explist1 .
stat       ::= functioncall .

repetition ::= 'for' NAME '=' explist23 .
repetition ::= 'for' namelist 'in' explist1 .

conds      ::= condlist .
conds      ::= condlist 'else' block .
condlist   ::= cond .
condlist   ::= condlist 'elseif' cond .
cond       ::= exp 'then' block .

laststat   ::= 'break' .
laststat   ::= 'return' .
laststat   ::= 'return' explist1 .

binding    ::= 'local' namelist .
binding    ::= 'local' namelist '=' explist1 .
binding    ::= 'local' 'function' NAME funcbody .

funcname   ::= dottedname .
funcname   ::= dottedname ':' NAME .

dottedname ::= NAME .
dottedname ::= dottedname '.' NAME .
```

```
namelist   ::= NAME .
namelist   ::= namelist ',' NAME .

explist1   ::= exp .
explist1   ::= explist1 ',' exp .
explist23  ::= exp ',' exp .
explist23  ::= exp ',' exp ',' exp .

%left      'or' .
%left      'and' .
%left      '<' '<=' '>' '>=' '==' '~=' .
%right     '..' .
%left      '+' '-' .
%left      '*' '/' '%' .
%right     'not' '#' .
%right     '^' .

exp        ::= 'nil'|'true'|'false'|NUMBER|STRING|'...' .
exp        ::= function .
exp        ::= prefixexp .
exp        ::= tableconstructor .
exp        ::= 'not'|'#'|'-' exp .         ['not']
exp        ::= exp 'or' exp .
exp        ::= exp 'and' exp .
exp        ::= exp '<'|'<='|'>'|'>='|'=='|'~=' exp .
exp        ::= exp '..' exp .
exp        ::= exp '+'|'-' exp .
exp        ::= exp '*'|'/'|'%' exp .
exp        ::= exp '^' exp .

setlist    ::= var .
setlist    ::= setlist ',' var .

var        ::= NAME .
var        ::= prefixexp '[' exp ']' .
var        ::= prefixexp '.' NAME .

prefixexp  ::= var .
prefixexp  ::= functioncall .
prefixexp  ::= OPEN exp ')' .

functioncall ::= prefixexp args .
functioncall ::= prefixexp ':' NAME args .

args       ::= '(' ')' .
args       ::= '(' explist1 ')' .
args       ::= tableconstructor .
args       ::= STRING .

function   ::= 'function' funcbody .
```

```
funcbody    ::= params block 'end' .

params      ::= '(' parlist ')' .

parlist     ::= .
parlist     ::= namelist .
parlist     ::= '...' .
parlist     ::= namelist ',' '...' .

tableconstructor ::= '{' '}' .
tableconstructor ::= '{' fieldlist '}' .
tableconstructor ::= '{' fieldlist ',|';' '}' .

fieldlist   ::= field .
fieldlist   ::= fieldlist ',|';' field .

field       ::= exp .
field       ::= NAME '=' exp .
field       ::= '[' exp ']' '=' exp .
```

Ejemplos de Código Lua

El siguiente fragmento de código incluye declaración y llamado de funciones con argumentos, operaciones lógicas, operaciones aritméticas, condicionales, declaración de variables locales, comentarios y manejo de errores

```
function add_event (op1, op2)
      local o1, o2 = tonumber(op1), tonumber(op2)
      if o1 and o2 then  -- both operands are numeric?
        return o1 + o2   -- '+' here is the primitive 'add'
      else  -- at least one of the operands is not numeric
        local h = getbinhandler(op1, op2, "__add")
        if h then
          -- call the handler with both operands
          return (h(op1, op2))
        else  -- no handler available: default behavior
          error(···)
        end
      end
    end
```

Alcance

Debido a las limitantes de tiempo y recursos, y con el objetivo de cumplir con los requisitos del curso, el desarrollo del compilador de Lua estará sujeto a las siguientes restricciones:

- Solo se implementarán funciones relevantes al paradigma procedimental.
- Las funcionalidades que deben existir en el proyecto son:
    - Declaración e inicialización de variables.
    - Operaciones algebraicas
    - Operaciones lógicas
    - Condicionales (if, else)

    - Ciclos (while)
- Se pretende utilizar el proyecto como guía en el estudio del proceso completo de un compilador, desde el análisis léxico hasta la ejecución del código, sin embargo, no se abordará el tema de optimización de manera exhaustiva.

Desarrollo del proyecto

Analisis Léxico

Tomando como base el analizador léxico desarrollado como parte de la materia de Teoría de la Computación se elaboró un analizador léxico específico para el reconocimiento de los tokens de Lua. Algunas de las estructuras que el analizador léxico reconoce son :

[==[	Apertura // Cierre de Agrupador
function	Palabra reservada
uniqueid_some_event	Identificador
(	Operador de Agrupación
e	Identificador
)	Operador de Agrupación
if	Palabra reservada
(	Operador de Agrupación
e	Identificador
:	Signo de puntuación
HasString	Identificador
(	Operador de Agrupación
"ignore string1"	Cadena de caracteres
)	Operador de Agrupación
)	Operador de Agrupación
then	Palabra reservada
a	Identificador
-	Operador Aritmético
4	Numero Natural
-- do something	Comentario
end	Palabra reservada
if	Palabra reservada
(	Operador de Agrupación
e	Identificador
:	Signo de puntuación

HasString	Identificador
(	Operador de Agrupación
"ignore string2"	Cadena de caracteres
)	Operador de Agrupación
)	Operador de Agrupación
then	Palabra reservada
-- do something	Comentario
end	Palabra reservada
end	Palabra reservada
some	Identificador
invalid	Identificador
closing	Identificador
comment	Identificador
tags	Identificador
:	Signo de puntuación
]==]	Apertura // Cierre de Agrupador
]====]	Apertura // Cierre de Agrupador
]===]	Apertura // Cierre de Agrupador
function	Palabra reservada
uniqueid_some_event	Identificador
(	Operador de Agrupación
e	Identificador
)	Operador de Agrupación
if	Palabra reservada
(	Operador de Agrupación
e	Identificador
:	Signo de puntuación
HasString	Identificador
(	Operador de Agrupación
"string1"	Cadena de caracteres
)	Operador de Agrupación
)	Operador de Agrupación
then	Palabra reservada
-- do something	Comentario
end	Palabra reservada
if	Palabra reservada
(	Operador de Agrupación
e	Identificador
:	Signo de puntuación

HasString	Identificador
(	Operador de Agrupación
"string2"	Cadena de caracteres
)	Operador de Agrupación
)	Operador de Agrupación
then	Palabra reservada
-- do something	Comentario
end	Palabra reservada
end	Palabra reservada
if	Palabra reservada
(	Operador de Agrupación
e	Identificador
:	Signo de puntuación
HasString	Identificador
(	Operador de Agrupación
"outside function..."	Cadena de caracteres
)	Operador de Agrupación
)	Operador de Agrupación
then	Palabra reservada

El analizador léxico es mas poderoso de lo que realmente ocupamos en la gramática y funcionaría perfecto no solo para nuestra propuesta sino para la propuesta original. En esta etapa rellenamos la **tabla de símbolos** para despues manipularla y saber el tamaño de memoria requerida en el código ensamblador.

```
Nombre | Tipo de Dato | Alcance
P1 id –1
y id –1
i id –1
```

Análisis Sintáctico

A partir del alcance y las acotaciones del proyecto se elaboró la siguiente gramática, a partir de la cual se implementarán las fases siguientes del proceso de compilación. Es básicamente un resumen de la gramática oficial.

```
# numer = [1-9]*
# binop = [+ / - + or and < > <= >? == ~=]*
# string = [a-zA-Z]

CHUNK -> BLOCK .

BLOCK -> SCOPE STATLIST .
BLOCK -> SCOPE STATLIST LASTSTAT SEMI .

SEMI -> ; .
```

SEMI-> .

SCOPE -> SCOPE STATLIST BINDING SEMI .
SCOPE -> .

EXP -> EXP binop T .
EXP -> T .
T -> F .
F -> (EXP) .
F -> true .
F -> false .
F -> numer .
F -> string .
F -> id .

STATLIST -> STATLIST STAT SEMI .
STATLIST -> .

STAT -> while EXP do BLOCK end .
STAT -> if CONDS end .

CONDS -> CONDLIST .
CONDS -> CONDLIST else BLOCK .

CONDLIST -> COND .
CONDLIST -> CONDLIST elseif COND .

COND -> EXP then BLOCK .

LASTSTAT -> return EXP .

BINDING -> local ID . # Se pone tipo de dato ID a tipo local
BINDING -> local ID = EXP . # Se pone tipo de dato ID a tipo local
BINDING -> ID = EXP .

Esta gramática contiene los siguientes no terminales con sus primeros y siguientes :

Simbolo	Es nullo	Es Final	Primeros	Siguientes
CHUNK	SI	SI	while, if, return, local, ID	$
BLOCK	SI	SI	while, if, return, local, ID	end, else, elseif, $
SEMI	SI	SI	;	while, end, if, else, elseif, return, local, ID, $
SCOPE	SI	SI	while, if, local, ID	while, end, if, else, elseif, return, local, ID, $
EXP		SI	(, true, false, numer, string, i	;, binop,), while, do, end, if, else, elseif, then, return, local, ID
T		SI	(, true, false, numer, string, i	;, binop,), while, do, end, if, else, elseif, then, return, local, ID
F		SI	(, true, false, numer, string, i	;, binop,), while, do, end, if, else, elseif, then, return, local, ID
STATLIST	SI	SI	while, if	while, end, if, else, elseif, return, local, ID, $
STAT		SI	while, if	;, while, end, if, else, elseif, return, local, ID, $
CONDS			(, true, false, numer, string, i	end
CONDLIST			(, true, false, numer, string, i	end, else, elseif
COND			(, true, false, numer, string, i	end, else, elseif
LASTSTAT		SI	return	;, end, else, elseif, $
BINDING		SI	local, ID	;, while, end, if, else, elseif, return, local, ID, $

A partir de esto generamos todos los estados LALR(1) (Hacer referencia a la Imagen Estados.svg) A partir de estos estados generamos un archivo de Excel con la tabla LALR(1). La tabla se encuentra anexa en el CD de este programa.

Se utilizo el programa Excel para poder exportar la tabla generada por la herramienta a un formato que pueda ser manipulado más fácilmente. Después se realizaron expresiones regulares que buscaban patrones dentro del archivo de Excel para sustituirlos con valores que puedan ser utilizados en el código fuente del compilador:

r\([0-9A-Za-z;=\s*\+\<\-\~>]*\)-> r

d\([\d]*\) -> d

\d+ -> i

Esta tabla representa cada uno de los estados y sus acciones (reducción y desplazamiento) de esta manera el compilador puede saber que tipo de funciones llamar, también el realizar las expresiones regulares hace mucho más sencillo el trabajo en dado caso de que existan futuros cambios a la gramática. Después de manera manual se creo una segunda tabla que representa la acción en especifico (numero de reducción o numero de desplazamiento) que tiene que realizar el compilador. Con estas dos tablas se codificaron todas las reducciones y desplazamientos necesarios de la gramática y con esto poder terminar el analizador sintáctico. Por ultimo se desarrollo la tabla de símbolos para el lenguaje la cual se compone de Lexema, tipo, clase, valor y posición la cual ser utilizada durante todo el proceso de análisis por los diferentes componentes del compilador.

La tabla se transforma en 3 estructuras en el analizador sintáctico.

La matriz de estados es la matriz que manipula el trabajo del analizador léxico y el que lleva el orden en cada caso. Es un apuntador de funciones donde cada función es una acción.

int f0(); // Función de prueba
int d(); // Función de desplazamiento
int e(); // Función de error
int ok(); // Función de Aceptar
int r(); // Función de Reducción

```
int (*matriz[ESTADOS][TERMINALESYNOTERMINALES])()={
    {e,e,e,e,e,e,e,e,r,r,e,e,r,e,e,e,r,r,e,r,i,i,i,e,e,e,e,e,e,e,e,e,e,e},
    {e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,ok,e,e,e,e,e,e,e,e,e,e,e,e},
    {e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,e,r,e,e,e,e,e,e,e,e,e,e,e,e},
    {e,e,e,e,e,e,e,e,r,r,e,r,r,r,r,e,r,r,e,r,e,e,e,i,e,e,e,e,e,e,e,e,e,e},
    {e,e,e,e,e,e,e,e,d,d,e,r,d,r,r,e,d,d,e,r,e,e,e,e,i,e,i,e,e,e,i,e,e,e},
    {d,e,e,e,e,e,e,e,e,e,e,e,r,e,r,r,e,e,e,e,r,e,e,e,e,e,i,e,e,e,e,e,e,e},
    {d,e,e,e,e,e,e,e,e,r,r,e,r,r,r,r,e,r,r,e,r,e,e,e,e,e,i,e,e,e,e,e,e,e},

    ...
};
```

Esta es la matríz que relaciona el tipo de desplazamiento y el tipo de reducción para las funciones de arriba. Por ejemplo R11 o d27

```
int matrizvalores[ESTADOS][TERMINALESYNOTERMINALES]={
    {0,0,0,0,0,0,0,0,6,6,0,0,6,0,0,0,6,6,0,6,1,2,3,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,-1,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,17,17,0,17,17,17,17,0,17,17,0,17,0,0,0,4,0,0,0,0,0,0,0,0,0,0},
```

```
    {0,0,0,0,0,0,0,0,10,11,0,1,12,1,1,0,8,9,0,1,0,0,0,0,5,0,6,0,0,0,7,0,0},
    {14,0,0,0,0,0,0,0,0,0,0,4,0,4,4,0,0,0,0,4,0,0,0,0,0,13,0,0,0,0,0,0,0,0},
    {14,0,0,0,0,0,0,0,4,4,0,4,4,4,4,0,4,4,0,4,0,0,0,0,0,0,15,0,0,0,0,0,0,0,0},
    {14,0,0,0,0,0,0,0,4,4,0,4,4,4,4,0,4,4,0,4,0,0,0,0,0,0,16,0,0,0,0,0,0,0,0},
    {0,0,20,0,21,22,23,24,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,17,18,19,0,0,0,0},
    {0,0,0,0,0,0,0,0,26,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,27,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},

     ...
};
```

Esta última matriz sirve para que la función genérica de reducciones haga la reducción cada elemento corresponde a un Terminal o No terminal de la gramática.

```
int reduccciones[32][8] = {

    {20, 21, -1, -1, -1, -1, -1, -1},
    {21, 22, 23, -1, -1, -1, -1, -1},
    {21, 22, 23, 24, 25, -1, -1, -1},
    {25, 0, -1, -1, -1, -1, -1, -1},
    {25, -1, -1, -1, -1, -1, -1, -1},
    {22, 22, 23, 26, 25, -1, -1, -1},
    {22, -1, -1, -1, -1, -1, -1, -1},
    ...

};
```

Específicamente para esta matríz de reducción programamos a mano cada reducción de los estados que se requerían utilizando condicionales. Por ejemplo en esta se observa la reducción del while en la acción semántica 18. En la parte de abajo vemos la creación del arbol AST al mismo tiempo que construimos el sintáctico.

```
case 18: // STAT -> while EXP do BLOCK end
            // Arbol sintactico
            printf("Accion semantica 18\n");
            creaHoja(nuevoArbol, 1, "STAT");
            hojaAux = (NodoArbol**) malloc(sizeof(NodoArbol*));
            creaHoja(hojaAux, 0, "end");
            agregaHijoExistente(nuevoArbol, hojaAux);
            arbolDePila = popT(&nodosHuerfanos);
            agregaHijoExistente(nuevoArbol, &arbolDePila);
            hojaAux = (NodoArbol**) malloc(sizeof(NodoArbol*));
            creaHoja(hojaAux, 0, "do");
            agregaHijoExistente(nuevoArbol, hojaAux);
            arbolDePila = popT(&nodosHuerfanos);
            agregaHijoExistente(nuevoArbol, &arbolDePila);
            hojaAux = (NodoArbol**) malloc(sizeof(NodoArbol*));
            creaHoja(hojaAux, 0, "while");
            agregaHijoExistente(nuevoArbol, hojaAux);
            pushT(*nuevoArbol, &nodosHuerfanos);

            // Arbol AST
            creaHoja(ASTArbol,1,"while");
            arbolDePila = popT(&ASTstack);
            agregaHijoExistente(ASTArbol, &arbolDePila);
            arbolDePila = popT(&ASTstack);
            agregaHijoExistente(ASTArbol, &arbolDePila);
            pushT(*ASTArbol,&ASTstack);
            break;
```

Uno de los principales objetivos para esta implementacion fue el reconstruir las estructuras de datos que dan soporte al compilador agregando funciones que nos permitan manipular y almacenar datos en una pila de arboles para la construcción del **analizador sintactico**, el **arbol abstracto de sintáxis** y el **codigo en 3 direcciones**.

Generación de Código y Cuadruples

Para la generación del codigo en 3 direcciones (cuadruples) se utilizo el arbol de AST en el cual se realiza un recorrido de todo el árbol y en cada operando encontrado se compone el cuadruple, colocando el operador, sus operandos y la variable donde se almacenará el resultado de la operación y todos estos son desplegados en la consola para poder verificar que sean correctas. Ejemplo:

```
>> (E2,,,)
>> (ADDx,T1,i,1)
>> (MOVx,i,T1,,)
>> (LTx,B2,i,9)
>> (BRT,B2,,E4)
>> (JUMP,E5,,)
>> (E4,,,)
>> (MOVx,i,0,,)
>> (JUMP,E6,,)
```

La implementación se hizo para cada cuadruple de manera independiente. Utilizamos una pila y una función recursiva para la generación de estos. En cada iteración se agregan cosas a una pila de cuadrupes por que en varias ocasiones hay ciclos anidados y es importante saber el orden de las etiquetas y cuando se les hace referencia. Este es el ejemplo de la generación del cuadruple del while :

```
if(!strcmp(AST->valor,"while"))
    {
        char endLbl[10];
        //Genero etiqueta de condicion
        idxRef++;
        sprintf(cuadruples,"%s\n(E%d,,,)",cuadruples,idxRef);
        //Introduzco la etiqueta en la pila
        sprintf(valorTemp,"E%d",idxRef);
        push(stackCuadruples,0,(char)0,valorTemp);

        //Genero los cuadruples de la condicion
        sprintf(cuadruples,"%s\n%s",cuadruples,generaCuadruples(AST-
>hijos[1]));
        //Saco el booleano de la condicion
        pop(stackCuadruples,&intBasura,&charBasura,valorTemp);
        sprintf(cuadruples,"%s\n(BRT,%s",cuadruples,valorTemp);
        //Genero la etiqueta de accion y termino el salto condicional
        idxRef++;

        sprintf(cuadruples,"%s,,E%d)",cuadruples,idxRef);
        //Genero el salto de terminacion
        idxRef++;
        sprintf(cuadruples,"%s\n(JUMP,E%d,,)",cuadruples,idxRef);
```

```
        //Introduzco la etiqueta de terminacion al stack
        sprintf(valorTemp,"E%d",idxRef);
        push(stackCuadruples,0,(char)0,valorTemp);
        //Imprimo la etiqueta de accion
        sprintf(cuadruples,"%s\n(E%d,,,)",cuadruples,idxRef-1);
        //Genero los cuadruples de accion
        sprintf(cuadruples,"%s\n%s",cuadruples,generaCuadruples(AST-
>hijos[0]));
        //Extraigo la etiqueta de terminacion
        pop(stackCuadruples,&intBasura,&charBasura,endLbl);
        pop(stackCuadruples,&intBasura,&charBasura,valorTemp);
        //imprimo el salto de condicion y la etiqueta de terminacion
        sprintf(cuadruples,"%s\n(JUMP,%s,,)
\n(%s,,,)",cuadruples,valorTemp,endLbl);
    }
```

Esto genera una lista en un archivo cuadruples.txt que se puede utilizar para la transformación a ensamblador.

Generación código ensamblador

Este es un método que se ejecuta después de terminado el ciclo inicial de la compilación. Nosotros lo vemos como un programa aparte aunque funciona en el mismo proceso de la compilación. Tomamos el archivo de cuadruples.txt y leemos linea por linea. El proceso que se realiza es el siguiente :

- Obtenemos operando y operadores. Algunas instrucciones no tienen operandos.
- Verificamos si los operando son números o otra cosa.
- Obtener la dirección de memoria del operando
- En base al operador se genera el código necesario

La generación toma en consideración la arquitectura de del micro 8051. Traducimos linea por linea de manera independiente ya que la lógica, los saltos y las etiquetas ya se encuentran definidas en el cuadruple. Este es el ejemplo de un ADD de cuádruple a ensamblador. Donde *operandos* contiene los fragmentos de cada cuádruple.

```
if(!strcmp(operandos[0],"ADDx\0")){
            fprintf(eout,"MOV A, %s \n",operandos[3]);
            fprintf(eout,"ADD A, %s \n",operandos[2]);
            fprintf(eout,"MOV %s, A \n",operandos[1]);
    }
```

Al final el código se almacena en un archivo *ensamblador.asm*

Experimentos y Metodología de pruebas

Metodología

1. Seleccionamos un código que requiera usar la mayor parte de las proyecciones de la gramática.
2. Ingresamos el valor esperado como output
3. Evaluamos el código ensamblador
4. Comparamos resultados

	Aprobado
Proceso Léxico	
Proceso Sintáctico	
Generación de Cuadruples	
Generación de Ensamblador	

Prueba 1 : Muchas estructuras mezcladas

```
i = 0
while ( i < 10 ) do
    i = i + 1
    if ( i > 9 ) then
        i = 0
    end
    if ( i == 0 ) then
        P1 = 192
    elseif ( i == 1 ) then
        P1 = 249
    elseif ( i == 2 ) then
        P1 = 164
    elseif ( i == 3 ) then
        P1 = 176
    elseif ( i == 4 ) then
        P1 = 153
    elseif ( i == 5 ) then
        P1 = 146
    elseif ( i == 6 ) then
        P1 = 130
    elseif ( i == 7 ) then
        P1 = 248
    elseif ( i == 8 ) then
        P1 = 128
    else
        P1 = 144
    end
end
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	√
Generación de Cuadruples	√
Generación de Ensamblador	√

Prueba 2 : Operandos

```
i = 0
i = 4 / 4 + 4
i = ( 4 * 4 ) + 4
```

```
i = ( 4 / 4 ) * ( 4 / 3 )
i = i + 2 + 3 + 4 + 5
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	√
Generación de Cuadruples	√
Generación de Ensamblador	√

Prueba 3 : Expresiones y orden

```
i + 4 = 5
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	x
Generación de Cuadruples	
Generación de Ensamblador	

Prueba 4 : While sencillo

```
while true do
    g = 4 + 1
end
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	√
Generación de Cuadruples	√
Generación de Ensamblador	√

Prueba 5 : While Anidado

```
while true do
    while true do
        while true do
            while true do
```

```
                j = j + 1
            end
        end
    end
end
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	√
Generación de Cuadruples	√
Generación de Ensamblador	√

Prueba 5 : if usando una expresión no booleana

```
if i = 0 then
    t = 1
end
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	x
Generación de Cuadruples	
Generación de Ensamblador	

Prueba 6 : if usando una expresión booleana

```
if i == 0 then
    t = 1
end
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	√
Generación de Cuadruples	√
Generación de Ensamblador	√

Prueba 6 : Prueba más común de Lua (simplificada) (https://code.google.com/p/compilador-lua/source/browse/trunk/+compilador-

lua+--username+djcribeiro/Lua/src/exemplos/arquivoTeste.txt?spec=svn6&r=6)

```
if true then m = 0 end
if n > 0 then m = 0 end
if k > 90 then m = 0 end
if vljfl == true then m = 0 end
if ( true ) then m = 0 end
if ( n > 0 ) then m = 0 end
if ( k > 90 ) then m = 0 end
if ( dkljfl == true ) then m = 0 elseif ( k > 90 ) then m = 9 end
if dkljfl == true then m = 0 elseif k > 90 then m = 9 end
if ( dkljfl == true ) then m = 0 else m = 9 end
while ( x < 90 ) do x = x + 1 end
while x < 89 do x = x + 3 end
```

	Aprobado
Proceso Léxico	√
Proceso Sintáctico	√
Generación de Cuadruples	√
Generación de Ensamblador	√

Interpretación de resultados

Las pruebas fueron satisfactorias. Logramos una compilación ágil y versátil con los conceptos mas comunes de lenguajes de programación. Creemos también que podemos mejorar los siguientes aspectos:

- Split sencillo de lexemas para evitar separlos manualmente.
- Mejorar el nombre de las variables
- Agregar Tablas y For
- Agregar Funciones
- Hacer mas modular cada una de las partes del compilador.

Conclusiones

Creemos que el desarrollo de un compilador maduro lleva mucho mas tiempo del que nos llevo. Si bien nuestro compilador es sencillo, es bastante potente puesto que logra el objetivo inicial. Por otro lado nos pareció bastante satisfactorio el resultado puesto que logramos la meta sin problemas y probamos distintos métodos de resolución de conflictos utilizando un lenguaje como C plano.

Creemos que este tipo de proyectos realmente motivan a los estudiantes a aprender mucho de lo que ocurre después de ejecutar algo como GCC. También te ayuda a

optimizar código ya que entiendes a detalle que ocurre en las capas inferiores del proceso de compilación.

Bibliografía

Data Structures in C, Adam Drozdek, Donald L. Simon, PWS 1995

Estructura de datos con C y C++, Yedidyah Langsam PHH, 1997

Aho, A.V., Sethi, R., Ullmann, J.D. & Suárez, P.F. (1998). Compiladores: principios, técnicas y herramientas. Addison Wesley Longman de México

Manual de Usuario

Interfaz

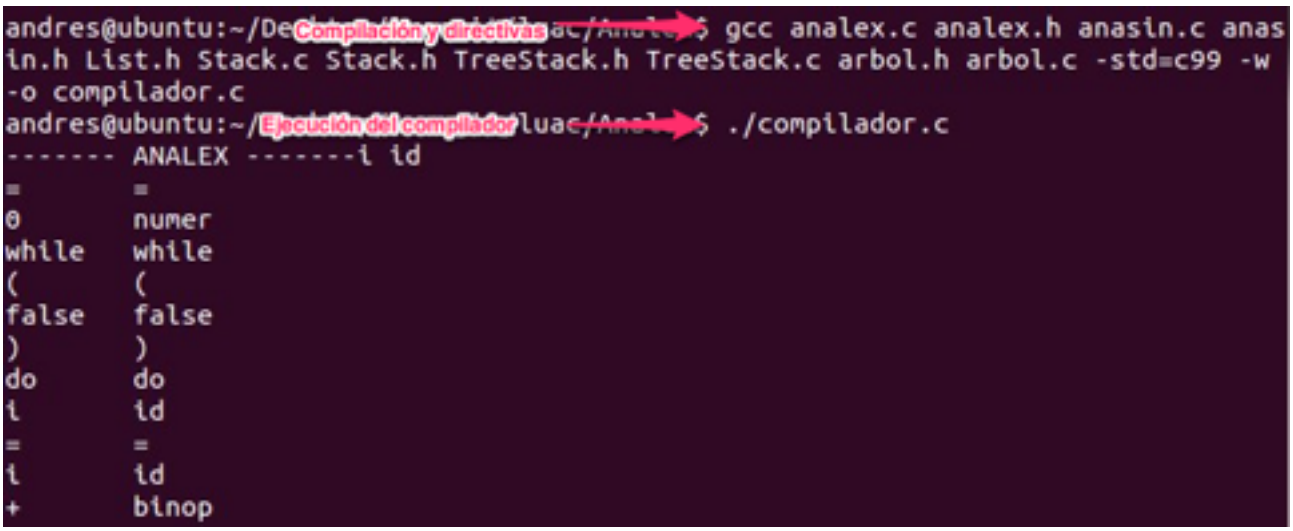
El compilador funciona sobre linea de comandos de Unix, Linux y Windows. Se ejecutan 2 comandos :

Para Compilar :

```
gcc analex.c analex.h anasin.c anasin.h List.h Stack.c Stack.h TreeStack.h TreeStack.c arbol.h arbol.c -std=c99 -w -o compilador.c
```

Para Ejecutar :

```
$ ./compilador.c
el codigo debe estar en codigo.txt
y genera el archivo cuadruples.txt y ensamblador.asm
```



Las variables P1, P2, P3 y P4 se usan como entradas y salidas del chip 8051 y así podemos visualizar correctamente el funcionamiento. El código tiene que estar en el archivo *codigo.txt*

Archivos

- analex.h : Incluye toda la lógica y los autómatas del Analex
- anasin.h : Es el archivo mas importante del proyecto que tiene toda la lógica de compilación
- arbol.h : Una estructura de árbol que guarda Valor, Tipo e Hijos
- codigo.txt : Donde va el código a compilar
- cuadruples.txt : El archivo generado de cuadruples
- List.h : Una estructura completa de lista
- Stack.h : Una estructura de datos de Stack
- tablasim.h : Una lista que se usa para la tabla de Simbolos
- TreeStack.h : Una estructura que guarda Nodos de Arbol para el árbol
- FINALFINAL.xls : Es el LALR (1) de nuestra gramática
- Automatas.svg : Son los estados y autómatas de nuestro lenguaje
- GramaticaFInal.txt : Es la gramática Final de nuestro lexema.

Ejemplos

En Codigo.txt

```
i = 0
while ( false ) do
    i = i + 1
    if ( i > 9 ) then
        i = 0
    end
    if ( i == 0 ) then
        P1 = 192
    elseif ( i == 1 ) then
        P1 = 249
    elseif ( i == 2 ) then
        P1 = 164
    elseif ( i == 3 ) then
        P1 = 176
```

```
    elseif ( i == 4 ) then
        P1 = 153
    elseif ( i == 5 ) then
        P1 = 146
    elseif ( i == 6 ) then
        P1 = 130
    elseif ( i == 7 ) then
        P1 = 248
    elseif ( i == 8 ) then
        P1 = 128
    else
        P1 = 144
    end
end
```

Ejecutamos :

```
$ ./compilador.c
```

Esto Genera :

Cuadruples.txt

```
(MOVx,i,0,,)
(E1,,,)
(MOVx,0,B1,,)
(BRT,B1,,E2)
(JUMP,E3,,)
(E2,,,)
(ADDx,T1,i,1)
(MOVx,i,T1,,)
(LTx,B2,i,9)
(BRT,B2,,E4)
(JUMP,E5,,)
(E4,,,)
(MOVx,i,0,,)
(JUMP,E6,,)
(E5,,,)
(E6,,,)
(EQx,B3,i,0)
(BRT,B3,,E7)
(JUMP,E8,,)
(E7,,,)
(MOVx,P1,192,,)
(JUMP,E9,,)
(E8,,,)
(EQx,B4,i,1)
(BRT,B4,,E10)
(JUMP,E11,,)
(E10,,,)
(MOVx,P1,249,,)
(JUMP,E12,,)
(E11,,,)
(EQx,B5,i,2)
(BRT,B5,,E13)
(JUMP,E14,,)
(E13,,,)
(MOVx,P1,164,,)
(JUMP,E15,,)
(E14,,,)
(EQx,B6,i,3)
(BRT,B6,,E16)
(JUMP,E17,,)
(E16,,,)
```

```
(MOVx,P1,176,,)
(JUMP,E18,,)
(E17,,,)
(EQx,B7,i,4)
(BRT,B7,,E19)
(JUMP,E20,,)
(E19,,,)
(MOVx,P1,153,,)
(JUMP,E21,,)
(E20,,,)
(EQx,B8,i,5)
(BRT,B8,,E22)
(JUMP,E23,,)
(E22,,,)
(MOVx,P1,146,,)
(JUMP,E24,,)
(E23,,,)
(EQx,B9,i,6)
(BRT,B9,,E25)
(JUMP,E26,,)
(E25,,,)
(MOVx,P1,130,,)
(JUMP,E27,,)
(E26,,,)
(EQx,B10,i,7)
(BRT,B10,,E28)
(JUMP,E29,,)
(E28,,,)
(MOVx,P1,248,,)
(JUMP,E30,,)
(E29,,,)
(EQx,B11,i,8)
(BRT,B11,,E31)
(JUMP,E32,,)
(E31,,,)
(MOVx,P1,128,,)
(JUMP,E33,,)
(E32,,,)
(MOVx,P1,144,,)
(E33,,,)
(E30,,,)
(E27,,,)
(E24,,,)
(E21,,,)
(E18,,,)
(E15,,,)
(E12,,,)
(E9,,,)
(JUMP,E1,,)
(E3,,,)
```

Ensamblador.asm

```
MOV #0, 09
E1:
MOV 044, #0
MOV A, 044
CJNE A, #0, E2
JMP E3
E2:
MOV A, #1
ADD A, 09
MOV 011, A
MOV 011, 09
MOV A, 09
SUBB A, #9
```

```
MOV A, #0
RLC A
MOV 045, A
MOV A, 045
CJNE A, #0, E4
JMP E5
E4:
MOV #0, 09
JMP E6
E5:
E6:
MOV A, #0
SUBB A, 09
MOV 046, A
MOV A, 046
JZ E7
JMP E8
E7:
MOV #192, P1
JMP E9
E8:
MOV A, #1
SUBB A, 09
MOV 047, A
MOV A, 047
JZ E10
JMP E11
E10:
MOV #249, P1
JMP E12
E11:
MOV A, #2
SUBB A, 09
MOV 048, A
MOV A, 048
JZ E13
JMP E14
E13:
MOV #164, P1
JMP E15
E14:
MOV A, #3
SUBB A, 09
MOV 049, A
MOV A, 049
JZ E16
JMP E17
E16:
MOV #176, P1
JMP E18
E17:
MOV A, #4
SUBB A, 09
MOV 050, A
MOV A, 050
JZ E19
JMP E20
E19:
MOV #153, P1
JMP E21
E20:
MOV A, #5
SUBB A, 09
MOV 051, A
MOV A, 051
JZ E22
```

```
JMP E23
E22:
MOV #146, P1
JMP E24
E23:
MOV A, #6
SUBB A, 09
MOV 052, A
MOV A, 052
JZ E25
JMP E26
E25:
MOV #130, P1
JMP E27
E26:
MOV A, #7
SUBB A, 09
MOV 053, A
MOV A, 053
JZ E28
JMP E29
E28:
MOV #248, P1
JMP E30
E29:
MOV A, #8
SUBB A, 09
MOV 054, A
MOV A, 054
JZ E31
JMP E32
E31:
MOV #128, P1
JMP E33
E32:
MOV #144, P1
E33:
E30:
E27:
E24:
E21:
E18:
E15:
E12:
E9:
JMP E1
E3:
END
```