Compilador Luac

Andrés, Isaac & Arturo

LuaC

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.

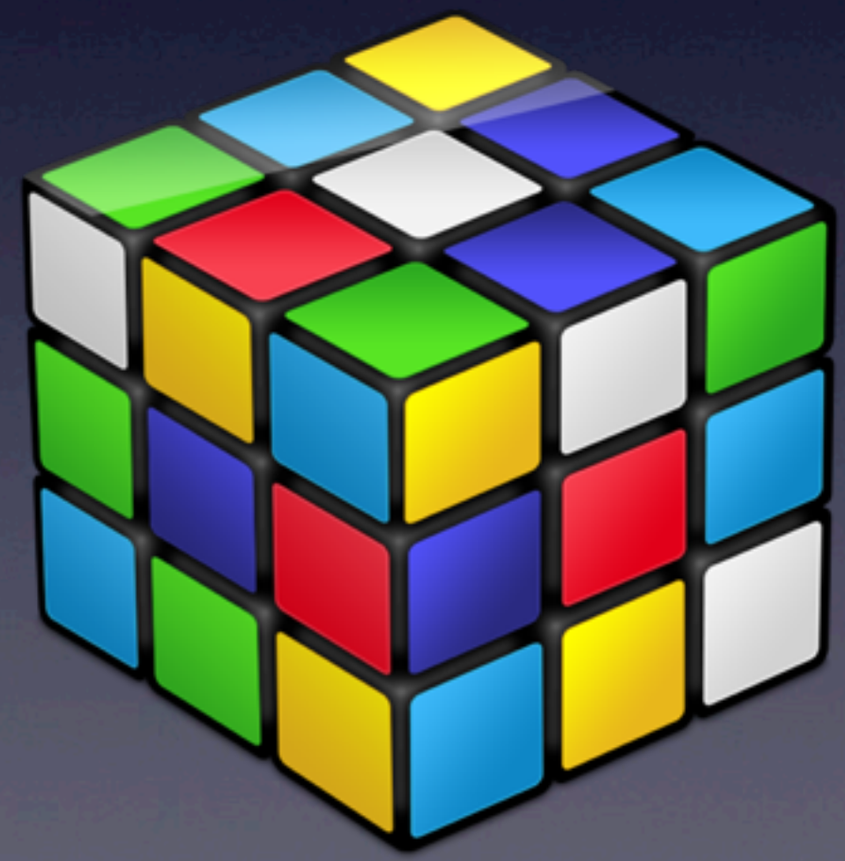
Lua está implementado como una biblioteca escrita en C.

Ejemplo

```
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

- Declaración e inicialización de variables.
- Operaciones algebraicas
- Operaciones lógicas
- Condicionales (if, else)
- Ciclos (while)



Lexico

- 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.

Análisis Sintáctico

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

Siguientes y Primeros

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, id	;, binop,), while, do, end, if, else, elseif, then, return, local, ID, $
T		SI	(, true, false, numer, string, id	;, binop,), while, do, end, if, else, elseif, then, return, local, ID, $
F		SI	(, true, false, numer, string, id	;, 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, id	end
CONDLIST			(, true, false, numer, string, id	end, else, elseif
COND			(, true, false, numer, string, id	end, else, elseif
LASTSTAT		SI	return	;, end, else, elseif, $
BINDING		SI	local, ID	;, while, end, if, else, elseif, return, local, ID, $

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

- Matriz de estados
- Matriz de Indices
- Matriz de Reducciones

Matriz de Estados

```
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,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,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,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},
    {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},
    {d,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},

    ...
};
```

Reducciones

Semántico

```
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);
```

AST

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

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.

```
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);
}
```

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.

Prosceso

- 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

Pruebas

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

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

```
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	√

```
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	√

```
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	√

Como funciona

```
andres@ubuntu:~/De                    ac/Anal    $ gcc analex.c analex.h anasin.c anas
in.h List.h Stack.c Stack.h TreeStack.h TreeStack.c arbol.h arbol.c -std=c99 -w
-o compilador.c
andres@ubuntu:~/                      luac/Anal    $ ./compilador.c
------- ANALEX -------i id
=       =
0       numer
while   while
(       (
false   false
)       )
do      do
i       id
=       =
i       id
+       binop
```

Compilación y directivas

Ejecución del compilador

Datos

- 87.4 % C
- 11.1 % C++
- 1.5 % Ensamblador

¡Gracias!