Proyecto de Autómatas y Lenguajes

Primeros pasos del compilador final

Objetivos del guión

- Guiarte paso a paso para que partas de tu analizador sintáctico y llegues a un compilador básico que procese adecuadamente un programa básico como

```
main {
       int x;
       x = 8;
       printf x;
}
```

ENSAMBLAR .l Y .y

1. Definir la estructura de información semántica AS: t13

Asignar información semántica a los tokens y a los no terminales

1. Ajustar includes en el .l

```
#include "alfa.h"
#include "y.tab.h"
```

1. Asignar valor semántico a los tokens en el .l

```
{IDENTIFICADOR}     {···
          strcpy(yylval.atributos.lexema,yytext);
                    ···}
{ENTERO}      {···
          yylval.atributos.valor_entero=atoi(yytext);              ···}
```

1. Declarar el valor semántico de los tokens en el .y

```
%{
#include "alfa.h"
…
%}

%union {
        tipo_atributos atributos;
}

…
%token <atributos> TOK_CONSTANTE_ENTERA
%token <atributos> TOK_CONSTANTE_REAL
%token <atributos> TOK_IDENTIFICADOR
```

…

```
%%
```

1. Declarar el valor semántico de los no terminales (no todos son necesarios ahora)

```
%type <atributos> condicional
%type <atributos> comparacion
%type <atributos> elemento_vector
%type <atributos> exp
%type <atributos> constante
%type <atributos> constante_entera
%type <atributos> constante_logica
%type <atributos> identificador
```

Declaración de variables: captura de tipo y clase de la declaración (simular herencia)

1. Variables globales en el .y

```
int
tipo_actual;
int clase_actual;
```

1. Tipos globales en el .h

```
/* CLASES */
#define ESCALAR 1
...
#define VECTOR 3
...

/* TIPOS */
#define INT 1
...
#define BOOLEAN 3
```

1. "Heredar" el tipo y clase de la declaración

```
/*--------------------------------------------------------------------------
---*/
/*  PRODUCCION 5
                          */
/*--------------------------------------------------------------------------
---*/
clase:  clase_escalar
          {
                        clase_actual = ESCALAR;
                        fprintf(output, ";R5:   <clase> ::=
<clase_escalar>\n");
          }
          ;
/*--------------------------------------------------------------------------
---*/
/*  PRODUCCION 7
                          */
```

```
/*------------------------------------------------------------------------
---*/
clase:  clase_vector
          {
                        clase_actual = VECTOR;
                        fprintf(output, ";R7:   <clase> ::=
<clase_vector>\n");
          }
          ;
/*------------------------------------------------------------------------
---*/
/*  PRODUCCION 10
                          */
/*------------------------------------------------------------------------
---*/
tipo:   TOK_INT
      {
                        tipo_actual = INT;
                        fprintf(output, ";R10:  <tipo> ::= int\n");
      }
      ;
/*------------------------------------------------------------------------
---*/
/*  PRODUCCION 11
                          */
/*------------------------------------------------------------------------
---*/
tipo:   TOK_BOOLEAN
      {
                        tipo_actual = BOOLEAN;
                        fprintf(output, ";R11:  <tipo> ::= boolean\n");
      }
      ;
```

Declaración de variables: preparación de la tabla de símbolos

1. Ensambla tu tabla de símbolos en la aplicación del compilador

```
#include "tabla_simbolos.h"   /* O COMO SE LLAME ALLÍ DONDE LA USES */
```

1. Declara tu tabla de símbolos (recuerda que será global) e Inicialízala donde te parezca adecuado

```
tablaSimbolosAmbitos tabla;       /* Tabla de simbolos */

iniciarTablaSimbolosAmbitos(&tabla);
```

1. Amplía tu tabla de símbolos con la información que se te sugiere en los guiones.

```
typedef struct elemento
{
/*
   lexema
   categoría  (variable, parámetro o función)
   clase  (escalar o vector)
   tipo  (entero o booleano)
```

```
   tamaño  (número de elementos de un vector)
   número de variables locales
   posición de variable local
   número de parámetros
   posición de parámetro
*/
...
} elementoTablaSimbolos;
```

1. Añade en la regla en la que se reducirá el identificador en la declaración, el código para insertarlo adecuadamente en la tabla desímbolos (AS:t20)

```
identificador:  TOK_IDENTIFICADOR {
   if ((buscarTablaSimbolosAmbitoActual(tabla, $1.lexema, ...)) == OK)
    {
      /* TRATAR EL ERROR DE QUE YA EXISTE */
    }
    else
    {
      insertarTablaSimbolosAmbitos(&tabla, $1.lexema,...,
                                   tipo_actual, calse_actual,... );

    }
```

Declaración de variables: escritura del principio del asm:

- **tabla de símbolos**
- **...**
- **etiqueta main:**

1. Modifica las reglas necesarias para escribir la tabla de símbolos en un lugar adecuado (también se te sugiere otros lugares en donde escribir diferentes partes del fichero ensamblador)

```
programa: TOK_MAIN'{'declaraciones escritura1 funciones escritura2
sentencias '}'
{
  /* Llamada a funciones para escribir el fin del fichero
fichero_ensamblador */
}
;
escritura1:
{
  /* Llamada a funciones para escribir la sección data con los mensajes
generales, y la
  tabla de símbolos así como todo lo necesario para que lo siguiente que
haya que escribirse
  sea main: */
}
escritura2:
{
  /* Aquí ya se podría llamar a la función que escribe inicio main
}
;
```

Parte de sentencias: distinción de declaración y uso de identificadores

1. Modifica las reglas en las que se usa identificador por TOK_IDENTIFICADOR, al menos las siguientes (AS:t40-42)

```
(43)  asignacion :  TOK_IDENTIFICADOR  '='  exp

(80)  exp  :  TOK_IDENTIFICADOR
```

Enhorabuena, puedes compilar por primera vez el programa

1. Con los siguientes comandos, deberías generar el ensamblador de una versión del programa que sólo escribe declara la variable x pero que al menos compila. No sigas hasta que no consigas eso (estamos suponiendo que el ejecutable se llama alfa, que estás ejecutando en una máquina de 64 bits y que el programa del ejemplo está en el fichero prg1.alf)

```
./alfa prg1.alf prg1.asm
nasm -felf32 prg1.asm
gcc -m32 -o prg1 prg1.o alfalib.o
```

PARA LA SEMANA QUE VIENE: Sentencias: gestión de 8 como expresión

1. Añade las acciones semánticas (AS:t52,54,55,57) y escribe el código (utiliza tus librerías) (GC:t20) para procesar la constante_entera como expresión

```
constante_entera:   TOK_CONSTANTE_ENTERA
{
  $$.tipo = INT;
  $$.es_direccion = 0;
  $$.valor_entero = $1.valor_entero;
  /* escribe código con tu librería para meter en la pila la constante
          push dword $1.valor_entero
  */

}
;

...
constante: constante_entera
{
        $$.tipo = $1.tipo;
        $$.es_direccion = $1.es_direccion;
}

...

exp:   constante
{
  $$.tipo = $1.tipo;
  $$.es_direccion = $1.es_direccion;

}
```

```
;
```

Sentencias: tratamiento de la asignación

1. Añadir tratamiento semántico y generación de código para la asignación

```
asignacion:     TOK_IDENTIFICADOR '=' exp
{
  if (  (buscarTablaSimbolosAmbitos(tabla, $1.lexema, ···) no lo encuentra )
  {
    Salir con ERROR
  }
  if (la categoria es FUNCION )
  {
    Salir con ERROR
  }
  if ( la clase es VECTOR )
  {
    Salir con ERROR
  }
  if ( el tipo encontrado en la tabla de símbolos != $3.tipo)
  {
    Salir con ERROR
  }
  /* Generar el código para
    • desapilar la parte derecha de la asignación:   pop dword eax
    • si la expresión es una dirección ($3.es_direccion == 1) (una variable)
        • Acceder a memoria por el valor adecuado: mov dword eax , [eax]
    • realizar la asignación :    mov dword [$1.lexema], eax
```

Sentencias: tratamiento de la impresión : tratamiento de una expresión cuando es un identificador

1. Modifica las reglas en las que la expresión es un identificador. AS:t40,42,56 GC:16

```
exp: TOK_IDENTIFICADOR
{
  if ((buscarTablaSimbolosAmbitos(tabla, $1.lexema, ···)) no lo encuantra )
  {
    Salir con ERROR
  }
  if (categoria recuperada de la tabla es FUNCION)
  {
    Salir con ERROR
  }
  if (clase recuperada de la tabla es VECTOR)
  {
    Salir con ERROR
  }

  $$.tipo=tipo sacado de la tabla de símbolos;
  $$.es_direccion=1;    /* ES UNA VARIABLE */

  /* Escritura en ensamblador de la introduccion en la pila de la dirección del
identificador: push dword  _$1.lexema */

}
;
```

Sentencias: tratamiento de la escritura

1. Añade la gestión semántica y la generación de código de la regla de escritura de una expresión GC:t52-54

```
escritura:   TOK_PRINTF exp
{
  /* Generar el código para
    ● Si la expresión es un identificador ($2.es_direccion == 1)
        ● Apilar lo que hay en la memoria de la dirección de la cima de la
          pila
                                    pop eax
                                mov eax, [eax]
                                push dword eax
  */

  /* Generar código para llamar a la función de impresión adecuada al tipo de
     la expresión call print_int, call print_boolean para $2.tipo == INT y
     BOOLEAN respectivamente */

  /* Generarl código para limpiar la pila y escribir un salto de línea
                                  add esp, 4
                              call print_endofline
  */

}
```

Comprobación

1. Comprueba que tu compilador hace la tarea adecuadamente

```
./alfa prg1.alf prg1.asm
nasm -felf32 prg1.asm
gcc -m32 -o prg1 prg1.o alfalib.o
./prog1
8
```