Actividad 5 - Compilador

Compiladores y Lenguajes Formales, GII

Sofía Martínez Parada Nº Expediente: 21926280

Marcos Eladio Somoza Corral Nº Expediente: 21711787

Claudia Jazmín Soria Saavedra Nº Expediente: 21986152

A día 9 de junio de 2022

Índice

Índice de Figuras

Introducción

Este documento expone la explicación de la actividad 5 de la asignatura *Compiladores y lenguajes formales* del cuarto curso del *Grado de Ingeniería Informática*. Se ha desarrollado un compilador empleando *Flex* y *Bison* para una versión simplificada del lenguaje de programación *Ada*, la cual tiene las instrucciones usadas comúnmente como *while* o *if*, y manejo de consola durante la ejecución del programa.

Este compilador generará un código ejecutable en ensamblador de *MIPS* y se validará mediante el emulador *MARS*.

Flex es una herramienta que permite generar analizadores léxicos. Se emplean expresiones regulares para encontrar los patrones en los ficheros de entradas y ejecutar acciones según dichos patrones.

Bison se centra en los analizadores sintácticos de propósito general. Genera un código en C que analiza la gramática.

MIPS, *Microprocessor without Interlocked Pipeline Stages,* es un lenguaje de código máquina interpretado por MARS, *MIPS Assembler and Runtime Simulator*, un programa desarrollado por la Universidad de Missouri.

Para la comprensión visual del proceso que sigue este proyecto para alcanzar el objetivo pedido se ha realizado el siguiente diagrama:

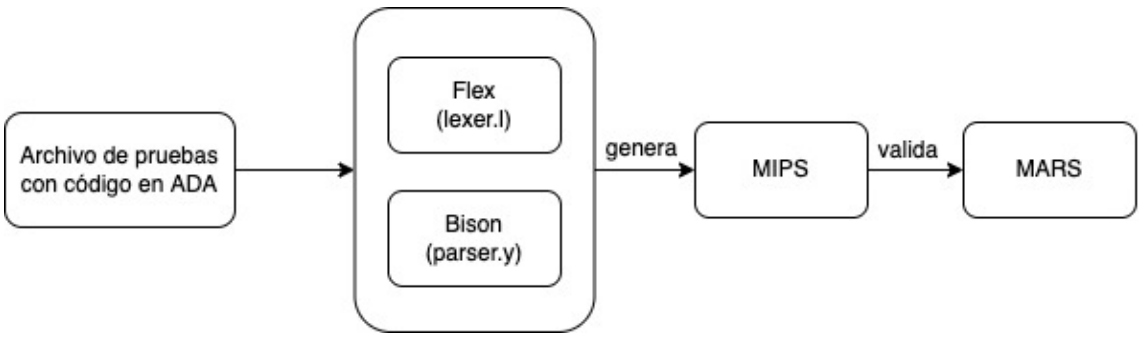

Figura 1. Diagrama del funcionamiento global de un compilador (elaboración propia)

Solución propuesta

Flex

El archivo ***lexer.l*** donde se definen los tokens que utilizará posteriormente el bison, estos tokens corresponderán a las palabras clave empleadas en el lenguaje ADA. El fichero flex se compone de tres fases separadas por los símbolos '%%'

Declaraciones

Se definen los tokens necesarios para la definición de reglas en la siguiente fase.

```
LETRA           [a-zA-Z]
DIGITO          [0-9]
PRINT           [ -~]
IDENT           {LETRA}({LETRA}|{DIGITO})*(_({LETRA}|{DIGITO})*)*
INTCONST        [+-]?{DIGITO}+
FLOATCONST      [-+]?{DIGITO}+(.{DIGITO}+)?((E|e)[-+]?{DIGITO}+)?
STRING          \"{PRINT}*\"
CHARCONST       (\'{PRINT}\')|(\'\\[nftrbv]\')
```

Reglas

Se definen los tokens de palabras reservadas del lenguaje ADA, símbolos y definiciones, basadas en las declaraciones de la fase anterior.

```
"--".*          { printf("Comentario en linea %d\n", lineno); }
"Character"     { return CHAR; }
"String"        { return STR; }
"Integer"       { return INT; }
"Float"         { return FLOAT; }

                                    .
                                    .
                                    .

{CHARCONST}     { yylval.val.cval = yytext[1];    return CHARCONST; }

"\n"            { lineno += 1; }
[ \t\r\f]+          /* eat up whitespace */
.               { yyerror("No reconocido"); }
```

*para ver el código completo, consulte el fichero lexer.l

Código de usuario

Esta fase es opcional y se ha decido que no es necesaria para la implementación del proyecto.

Bison

En el archivo ***parser.y*** se definirá la gramática del compilador, proporcionando las herramientas que permitirán la detección de:

1. Inicio y fin del programa con la estructura: *procedure name …. end name*.
2. Variables: enteros, booleanos y reales. Tanto las declaraciones como las asignaciones.
3. Operaciones aritméticas.
4. Operaciones lógicas.

5. Sentencias *if*, incluyendo *elseif* y *else*.
6. Bucle *while*.
7. Comentarios.

Como en el flex, este archivo también se divide en tres fases distinguidas por los símbolos '%%'.

Declaraciones

Inicialmente se declaran los valores de los tokens:

```
%token<val> CHAR STR INT FLOAT
%token<val> ADDOP MULOP DIVOP INCR OROP ANDOP NOTOP EQUOP RELOP
%token<val> FIRST LAST IMAGE VALUE MIN MAX PRED SUCC
%token<val> PROC IS BEG END
%token<val> SEPARADOR
%token<val> ASSIGN INI LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE SEMI DOT
COMMA CHANGE
%token<val> TYPE RANGE OF ARRAY TWOPOINTS NEW RET
%token<val> IF THEN ELSE ELSIF
%token<val> FOR IN LOOP REVERSE WHILE
%token<val> PUT GET NEW_LINE
%token<symtab_item> IDENT
%token<val> INTCONST
%token<val> FLOATCONST
%token<val> STRING
%token<val> CHARCONST
```

Se determinan las prioridades de los tokens mediante las etiquetas *%left* y *%right*.

```
%left COMMA
%right ASSIGN
%left OROP
%left ANDOP
%left EQUOP
%left RELOP
%left ADDOP
%left MULOP DIVOP
%right NOTOP INCR
%left LPAREN RPAREN LBRACK RBRACK
```

Por último, se declara el símbolo de arranque de la gramática:

```
%start procedure
```

Reglas

En esta segunda fase del código, se determinan las acciones a realizar en código C para cada símbolo no terminal. Además, se genera el árbol de sintaxis abstracta (AST) que se desarrollará en profundidad en un apartado posterior de esta memoria.

```
procedure: PROC IDENT IS declarations BEG statements END IDENT SEMI
    {
        AST_Node_Proc *tempi = (AST_Node_Proc*) $2;
        AST_Node_Proc *tempf = (AST_Node_Proc*) $8;
        if(tempi->val.sval == tempf->val.sval){
            main_decl_tree = $4; ast_traversal($4);
            main_func_tree = $6; ast_traversal($6);
        }else{
            yyerror("No coinciden los nombres del procedimiento");
            exit(0);
        }
    }
;
```

Códigos de usuario

Este apartado incluye funciones empleadas para generar la tabla de símbolos, imprimir por consola el avance del programa, guardar en un archivo la resultante tabla de símbolos y demás métodos necesarios en la función *main* del compilador.

```
int main (int argc, char *argv[]){

    // initialize symbol table
    init_hash_table();

    // parsing
    int flag;
    yyin = fopen(argv[1], "r");
    flag = yyparse();
    fclose(yyin);
    printf("Parsing finished!\n");

    // symbol table dump
    yyout = fopen("symtab_dump.out", "w");
    symtab_dump(yyout);
    fclose(yyout);

    //added on mips
    //generate_code();
    return flag;
}
```

Tabla de símbolos

Es una estructura de datos que guarda cada símbolo del código fuente, su tipo y la línea en donde aparece.

Para generar dicha tabla se ha desarrollado la librería ***symtab*** la cual genera un archivo, cada vez que se ejecuta el compilador, llamado ***symtab_dump.out***.

```
------------ -------------- ------ ------------
Nombre         Tipo           Scope  N linea
------------ -------------- ------ ------------
er           undef            0     8
A            int              0     2    7    7
B            real             0     3
C            int              0     4
Main         undef            0     1
```

Figura 2. Tabla de símbolos guardada en el fichero symtab_dump.out (elaboración propia)

Árbol de Sintaxis Abstracta

Es una estructura de datos que representa la estructura del código de un programa. Un AST es usualmente el resultado del analizador sintáctico en la fase de un compilador.

Para la extracción de este árbol se ha desarrollado la librería ***ast.*** Como todo árbol está formado por nodos, se han declarado diversos tipos de nodos dependiendo del tipo que esté gestionando (declaraciones, constantes, *statements*, expresiones... véase el *enum Node_Type* en la línea 3 de ***ast.h***, donde para cada valor del enum existe uno de estos nodos), pero todos ellos heredan de una estructura de nodo base, tal que:

```c
/* The basic node */
typedef struct AST_Node{
    enum Node_Type type; // node type

    struct AST_Node *left;  // left child
    struct AST_Node *right; // right child
}AST_Node;
```

En el ***ast.c*** se han declarado funciones que, según para cada tipo de nodo, crean una nueva instancia y lo escriben en el árbol. Como por ejemplo esta función, que crea el nodo para una variable constante:

```c
AST_Node *new_ast_const_node(int const_type, Value val){
    // allocate memory
    AST_Node_Const *v = malloc (sizeof (AST_Node_Const));

    // set entries
    v->type = CONST_NODE;
    v->const_type = const_type;
```

```
    v->val = val;

    // return type-casted result
    return (struct AST_Node *) v;
}
```

Además, se han creado dos métodos auxiliares para poder recorrer y representar en consola el contenido de un nodo dado. El primero, *ast_transversal*, se encarga de recorrer recursivamente cada uno de los hijos, tanto izquierdo como derecho del nodo pasado como parámetro, y llamar al método *ast_print_node* (del nodo proporcionado); que se encarga de (en base al tipo de nodo actual) imprimir su información por consola.

MIPS

Finalmente, una vez creado el árbol de sintaxis abstracta, se deberá generar (en base a éste) un fichero de salida que interprete en lenguaje ensamblador (MIPS) el fichero de entrada dado.

Para realizar este cometido, se ha creado una librería propia MIPS, encargada de la lectura del ast y su conversión a código interpretable por MARS en lenguaje MIPS.

Así pues, en la función *main* del ***parser.y*** se llama al método *generate_code* de ***mips.c***, encargado de abrir el fichero de salida final ***out.asm*** y rellenarlo con la estructura correspondiente.

```
// main assembly code generation function
void generate_code(){
    FILE *fp;
    fp = fopen("out.asm", "w");

    generate_data_declarations(fp);

    generate_statements(fp);

    fclose(fp);
}
```

Como se puede ver, para generar el código ensamblador final se llaman a dos métodos. El primero, *generate_data_declarations*, encargado de la creación del apartado de declaración de variables en MIPS. El segundo, *generate_statements*, de la creación del resto del código.

Antes de seguir con esta implementación, se debe entender cómo funciona la estructura básica de MIPS, que se divide en dos partes diferenciadas.

.Data

Como la primera parte del script, centrada en la declaración de variables según sus tipos y valores, siguiendo siempre la siguiente estructura:

$nombreVar : .tipo\ valor$

.Text

En esta segunda parte se encuentran las declaraciones, tales como asignación de variables a determinados registros, operaciones aritméticas entre registros de mismo tipo, movimiento del contenido de un registro a otro, llamadas a funciones, bucles …

Así pues, se han creado dos métodos encargados cada uno de completar el contenido dentro de .data y .text, *generate_data_declarations* y *generate_data_statements* respectivamente.

```
// data declaration assembly code
void generate_data_declarations(FILE *fp){
    // print .data
    fprintf(fp, ".data\n");

    // loop through the symbol table's lists
    fprintf(fp, "# variables\n");
    int i, j;
    for (i = 0; i < SIZE; i++){
        // if hashtable list not empty
        if (hash_table[i] != NULL){
            list_t *l = hash_table[i];
            // loop through list
            while (l != NULL){
                // Simple Variables
                if (l->st_type == INT_TYPE){
                    fprintf(fp, "%s: .word %d\n", l->st_name, l->val.ival);
                }
                else if (l->st_type == REAL_TYPE){
                    fprintf(fp, "%s: .float %f\n", l->st_name, l->val.fval);
                }
                else if (l->st_type == CHAR_TYPE){
                    fprintf(fp, "%s: .byte \'%c\'\n", l->st_name, l-
>val.cval);
                }
                l = l->next;
            }
        }
    }

    // loop through the string messages
    fprintf(fp, "# messages\n");
```

```
    for(i = 0; i < num_of_msg; i++){
        fprintf(fp, "msg%d: .asciiz %s\n", (i + 1), str_messages[i]);
    }
}
```

En primera instancia escribe en el fichero .data, usando *fprintf()*, al igual que con el resto de escrituras a fichero.

Recorre la tabla hash, identificando aquellos elementos que se correspondan con los tipos entero, real, *char* o *char**; donde en cuyo caso escribe su declaración siguiendo la estructura mencionada anteriormente, donde *nombreVar* se recoge con *l-> st_name* y valor con *l-> val.cval*. Véase la línea 16 del fichero ***mips.c*** para más detalle.

```
// statements assembly code
void generate_statements(FILE *fp){
    int i, j;
    // print .text
    fprintf(fp, "\n.text\n");

    // Main Function Register Allocation
    initGraph();
    main_reg_allocation(main_decl_tree);
    main_reg_allocation(main_func_tree);

    // add edges from all the non-temporary variables
    for(i = 0; i < var_count - temp_count; i++){
        for(j = 1; j < var_count; j++){
            if(i < j){
                insertEdge(i, j);
                insertEdge(j, i);
            }
        }
    }
    printVarArray();
    printGraph();

    // Main Function Register Assignment
    int *colors = greedyColoring();

    printf("Colors:\n");
    for(i = 0; i < var_count; i++){
        printf("%s: %d\n", var_name[i], colors[i]);
    }
    printf("\n");

    printf("Registers:\n");
    for(i = 0; i < var_count; i++){
```

```
        printf("%s: %s or %s\n", var_name[i], GetRegisterName(colors[i], 0),
GetRegisterName(colors[i], 1));
    }

    // assign register-color value as reg_name
    list_t *l;
    for(i = 0; i < var_count; i++){
        l = lookup(var_name[i]);
        l->reg_name = colors[i];
    }
    // print main:
    fprintf(fp, "main:\n");
    //actual statement generation
    // reset temporary counter
    temp_count = 0;
    // traverse main function tree
    main_func_traversal(fp, main_func_tree);
}
```

En primera instancia escribe en el fichero .text, usando también *fprintf()*. Seguidamente, instancia un nuevo grafo con *initGraph()* para alocar en él todos los nodos relevantes del ASP que no formen parte del conjunto de declaración de variables (aquel del que se encarga el método anteriormente explicado).

A efectos prácticos, el grafo contiene: una lista dinámica de nodos donde cada nodo tiene un índex y una referencia al siguiente nodo, y el número total de nodos. Seguidamente, se llama al método *main_reg_allocation()* para un árbol (*main_decl_tree*) que contiene las declaraciones principales que haya dentro del *main* del AST, y para un árbol (*main_func_tree*) que contiene todos los *statements* del *main* del AST.

Este método se encarga de detectar el tipo de cada elemento del nodo dado, es decir, de cada árbol que reciba, e insertarlo en el grafo. Una vez lleno el grafo, genera las aristas del mismo con *insertEdge()*.

Seguidamente, se llama al método *greddyColoring()*, encargado de recordar los registros para cada tipo de variable que almacena. Seguidamente se asigna el registro correspondiente a cada uno de los elementos de la lista cuyas acciones impliquen su uso (*lwc1, li, add, move, sub, move...*).

Imprime en el fichero de salida el string *.main:* , a fin de indicar que comienza la parte principal del *.text*.

Finalmente, y gracias a la función *main_func_transversal()* (una similar a la vista anteriormente en AST), se recorren todos los nodos del grafo, detectando de qué Node_Type se trata, a fin de crear un nodo nuevo a partir de él y recorrer sus hijos

izquierdo y derecho recursivamente con *main_func_transversal()*. Además, dependiendo del tipo de nodo que sea, llamará a una u otra función encargada de escribir al fichero las declaraciones correspondientes.

A modo de ejemplo, si entrara un nodo de tipo *ARITHM_NODE*, además de llamar recursivamente a sus hijos (mejor dicho, a los hijos de una copia temporal del nodo original), llamará al método *generate_arithm()*. Este método es el encargado de detectar de qué operación aritmética se trata, y con qué tipos de variables se está trabajando, a fin de escribir la línea correspondiente en el fichero (si fuera una suma de enteros, detectaría *ADD* si se están usando dos variables declaradas o *ADDI* si una de ellas no lo fuera, y colocaría los registros de dichas variables en la escritura al archivo).

Para más detalle sobre cómo funciona, véanse los métodos desarrollados en el fichero ***mips.c***.

Directivas de compilación

Se destaca que este código está pensado para su empleo en máquinas MacOS debido a que todos los integrantes del grupo poseen dichas máquinas.

Instalaciones previas

Se ha empleado la herramienta ***Homebrew*** para la instalación de flex y bison.

```
$ brew install flex
$ brew install bison
```

En caso de no tener instalado la herramienta homebrew, será necesario correr previamente en el terminal de Mac el siguiente comando:

```
$ ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Ejecución

Para la simple ejecución del programa se ha decidido escribir un pequeño script en Shell llamado *comp.sh*. Este fichero está compuesto de las siguientes órdenes.

```
#!/bin/bash
bison -d parser.y
flex lexer.l
gcc parser.tab.c lex.yy.c
rm lex.yy.c parser.tab.c parser.tab.h
./a.out $1
```

La primera línea genera los archivos ***parser.tab.c*** y ***parser.tab.h*** y la línea dos genera el archivo escáner de implementación ***lex.yy.c***.

En cuanto tenemos estos tres archivos generados, los compilamos para generar el ***a.out***.

La instrucción ***rm lex.yy.c parser.tab.c parser.tab.h*** elimina los archivos generados para evitar que se reescriban y así mantener la carpeta más liberada, ya que lo que interesa de este proceso es la obtención del archivo ***.out.***

Por último, se ejecuta dicho archivo pasándole como argumento el nombre del archivo ***.ada***.

Batería de pruebas

Con el objetivo de comprobar el correcto funcionamiento del compilador se han escrito una batería de sencillos programas que se sitúa en la carpeta */pruebas*.

Se ha desarrollado ***semantics***, un fichero que se compone de las funciones *get_result_type* y *type_error*. La primera, que detecta la compatibilidad de operaciones entre variables según los tipos de esta; y la segunda, que escribe por consola el error de incompatibilidad mostrando la línea en la que ocurre dicho error.

Para poder simplificar este apartado y para facilitar la comparativa, se han definido los tipos de variables y los tipos de operaciones como:

```
// token types */
#define UNDEF 0
#define INT_TYPE 1
#define REAL_TYPE 2
#define CHAR_TYPE 3
#define STR_TYPE 4

// operator types */
#define NONE 0      // to check types only - assignment, parameter
#define ARITHM_OP 1 // ADDOP, MULOP, DIVOP (+, -, *, /)
#define INCR_OP 2   // INCR (++, --)
#define BOOL_OP 3   // OROP, ANDOP (||, &&)
#define NOT_OP 4    // NOTOP (!)
#define REL_OP 5    // RELOP (>, <, >=, <=)
#define EQU_OP 6    // EQUOP (==, !=)
```

A continuación, se mostrará el código de comprobación de la instrucción *if*, tanto el código generado correctamente como el que tiene errores, así como el output obtenido.

Test con programas correctos

Código:

```
procedure Main is
   int1: Integer;
   int2: Integer;
begin
   int1:=5;
   int2:=8;
   if int1<int2 then
      int1:= 5 + 5;
   elsif int1 = int2 then
      int1:= 10 + 5;
   else
      int1:= 7;
   end if;
end Main;
```

Output:

```
  Nodo Constante de const-type 1 con valor 5
  Nodo de Asignación de entrada int2
  Asignando:
  Nodo Constante de const-type 1 con valor 8
  Nodo If con 1 elseifs y else
  Condición:
  Nodo de referencia de entrada int1
  Nodo de referencia de entrada int2
  Nodo relacional de operador 1
  If branch:
  Nodo de Statements con 1 statements
  Nodo de Asignación de entrada int1
  Asignando:
  Nodo Constante de const-type 1 con valor 5
  Nodo Constante de const-type 1 con valor 5
  Nodo Aritmético con operador 0 con resultado de tipo 1
  Else if branches:
  Else if branch0:
  Nodo Elsif
  Nodo de referencia de entrada int1
  Nodo de referencia de entrada int2
  Nodo de igualdad de operador 0
  Nodo de Statements con 1 statements
```

Figura 3. Salida de la ejecución de IfNested.ada (elaboración propia)

Test con programas con errores

Código:

```
procedure Main is
   a : Character := 'c';
   b : Float := 2.5;
   c : Float := 7.2;
   begin
   if a = 'c'  && a /= 'b'then
      c := c + b;
   end if;
```

```
end Main;
```

Output:

```
Error  syntax error
Error de sintaxis en linea 6
```

Figura 4. Salida de la ejecución de IfStatementError.ada (elaboración propia)

Conclusiones

Tras haber logrado implementar un compilador extremadamente básico para el lenguaje de programación ADA, hemos podido experimentar lo complejo que puede llegar a ser su desarrollo mínimo indispensable para la compilación de programas simples.
Si ha requerido tanto tiempo de trabajo y diseño planificar cómo se implementarían las diferentes partes del compilador para ser capaces de compilar declaraciones de variables, bucles simples y demás acciones básicas; es impensable el trabajo que requeriría diseñar y desarrollar un compilador completo y funcional, con gestión de errores e incluso intelligence , para un lenguaje como C#, donde además encontraríamos casos de POO, polimorfismo, sobrecarga de operadores, interfaces...

Si bien es cierto que diseñar un compilador mínimo viable requiere mucho tiempo, a la hora de añadir funcionalidad a éste (si está bien matizada desde el principio) no se requiere de tanto esfuerzo. Como en flex ya están la mayoría de regex y tokens especificados, se podrían hacer uso de éstos para implementar otros nuevos más complejos, al igual que ocurriría con bison para añadir nuevas expresiones, haciendo uso de las ya declaradas.
En cuanto al desarrollo específico de nuestro compilador, hemos procurado mantener una arquitectura de código suficientemente viable que permita futuras expansiones y mejoras del mismo sin necesidad de un rediseño. De ahí que se haya optado por diseñar nodos específicos para el AST, así como grafos con sus referencias específicas también para MIPS.

Finalmente, se quisieran señalar las mejoras posibles del compilador si tuviera más tiempo de desarrollo y por ende más tiempo disponible para implementarse.

- Implementación de funciones desde el AST a MIPS

- Implementación de bucles, exceptuando el ya implementado *while*, desde el AST a MIPS (a su vez son llamadas a funciones)
- Implementación de conversiones de tipos para operaciones aritméticas imposibles en MIPS (sumar un double con un float en un registro, por ejemplo).
- Implementación del uso del *pointer stack sp* para agilizar los relacionados con gestión de memoria en multitud de operaciones.

Bibliografía

[1] V. d. Barrio Terceño, «Compilador,» 1994. [En línea]. Disponible en: https://repositorio.comillas.edu/xmlui/handle/11531/9589. [Último acceso: 15 Mayo 2022].

[2] G. y. A. c. c. (. d. I. e. I. Prácticas de Lenguajes, «Introducción a Flex y Bison,» Junio 2011. [En línea]. Disponible: http://webdiis.unizar.es/asignaturas/LGA/material_2003_2004/Intro_Flex_Bison.pdf. [Último acceso: 15 Abril 2022].