

COMPILADOR

Actividad aplicativa colaborativa 6

Alejandro Hernán, Diego Gil y Jennyfer González

COMPILADORES Y LENGUAJES FORMALES

ÍNDICE

<i>Introducción al problema</i>	4
<i>Solución propuesta</i>	4
Hito 1	4
Flex	5
Definición	6
Reglas	6
Código de usuario	8
Bison	8
Definición	9
Reglas	9
Subrutinas de usuario	12
Hito 2	12
Flex	12
Bison	13
Definición	13
Reglas	16
Subrutinas de usuario	19
SymTab.h	19
AST.h	23
Creación de árboles	23
Hito 3	26
Flex	26
Bison	26
Definición	26
Reglas	27
Subrutinas de usuario	30
SymTab.h	30
AST.h	30
MIPS	33
Estructura de MIPS	33
Operaciones aritméticas	34
Sentencia If	35
Bucle While	37
Manual de usuario	39
Problemas encontrados	42
Batería de pruebas	42
Tests con programas correctos	42
Tests con programas con errores	44
Conclusión	45

ÍNDICE DE FIGURAS

Figura 1: Logo Ada	4
Figura 2: Logo Flex.....	5
Figura 3: Identidicadores.....	7
Figura 4: Atoi y Atof.....	7
Figura 5: Código token	9
Figura 6: Ada	10
Figura 7.....	11
Figura 8: Recursividad 1	11
Figura 9: Recursividad 2	11
Figura 10: Fichero Salida	12
Figura 11: Identificadores.....	13
Figura 12: Código de usuario.....	14
Figura 13: BISON	14
Figura 14: Tokens	15
Figura 15 % Union	15
Figura 16: Comprobación de tipos	16
Figura 17: Comprobaciones	17
Figura 18: Comprobación de variable existente.....	17
Figura 19: Control de uso de variables.....	18
Figura 20: Control de divisiones entre 0	18
Figura 21: Comprobación de funciones con el mismo nombre.....	18
Figura 22: Comprobación de sentencia "return"	18
Figura 23: Comprobación de tipos	19
Figura 24 Symtab 1.....	19
Figura 25: Inicialización de tabla de símbolos	20
Figura 26: Configuración de tabla de símbolos	20
Figura 27: COnfiguración de tabla de símbolos 2.....	20
Figura 28: Configuración de tabla de símbolos 3	21
Figura 29: Comprobación de tabla de símbolos 4	21
Figura 30: Comprobación de tabla de símbolos 5	21
Figura 31: Comprobación de tabla de símbolos 6	22
Figura 32: Configuración de tabla de símbolos 7	22
Figura 33: Configuración de tabla de símbolos 8	22
Figura 34: Configuración de tabla de símbolos 9	23
Figura 35: Creación del árbol	24
Figura 36: Árbol de constantes.....	24
Figura 37: Inicialización de los árboles	24
Figura 38: Estructura del árbol	24
Figura 39: Recogida de constantes.....	25
Figura 40: Operaciones aritméticas.....	25
Figura 41: Función de liberación	26
Figura 42: Nuevas librerías y variables	26
Figura 43: Struct fncall	27
Figura 44: Pasarela MIPS	28
Figura 45: GlobalContadorNum	28

Figura 46: Identificador	29
Figura 47: Comprobación if	29
Figura 48: Bucle while	30
Figura 49: Extra generación de árboles	30
Figura 50: Estructura de los árboles	30
Figura 51: Árbol para funciones	31
Figura 52: Inicialización de árboles	31
Figura 53: Inicialización de la función	31
Figura 54: Detección de tipos de función	31
Figura 55: Árbol flow	32
Figura 56: Árbol de funciones	32
Figura 57: Ejemplo de operación aritmética	34
Figura 58: Código comprobaciones	34
Figura 59: test de código	35
Figura 60: Sentencia if preparada para MIPS	35
Figura 61: Resultado generado	36
Figura 62: Ventana de código.....	37
Figura 63: Bucle while	37
Figura 64: Resultado.....	38
Figura 65: Ventana de código.....	39
Figura 66: Ventana de código.....	40
Figura 67: Ventana de herramientas y mensajes de código	40
Figura 68: Ventana de código.....	41
Figura 69: Ventana de código.....	41
Figura 70: ADA.....	43
Figura 71: Solución ADA	43
Figura 72: Ejemplo ADA incorrecto	44
Figura 73: Solución	44

Introducción al problema

El objetivo de esta actividad es construir un compilador utilizando Flex y Bison para el lenguaje de programación Ada 2012 o Spark 2014 que genere código ejecutable en ensamblador de MIPS y que se validará en el emulador MARS (MIPS Assembly and Runtime Simulator).

Se partirá de un programa escrito en Ada. Donde se deberá reconocer los siguientes tokens o grupos de tokens, siendo necesario incluir una serie de palabras reservadas necesarias.

Solución propuesta

En este trabajo se explica el desarrollo de un compilador completo, al ser un proceso muy complejo se dividió en tres fases o hitos, que se explicarán en los siguientes puntos.

Hito 1

Para realizar este trabajo lo primero que se hizo fue programar un código en Ada para realizar en base a ese fichero el análisis léxico. En él se tuvo que añadir todos los elementos indicados en el enunciado.

Ada es un lenguaje de programación orientado a objetos y fuertemente tipado de forma estática. Es un lenguaje multipropósito, orientado a objetos y concurrente, pudiendo llegar desde la facilidad de Pascal hasta la flexibilidad de C++.

Fue diseñado con la seguridad en mente y con una filosofía orientada a la reducción de errores comunes y difíciles de descubrir. Está basado en un tipado muy fuerte y en chequeos en tiempo de ejecución (desactivables en beneficio del rendimiento).

Ada se usa principalmente en entornos en los que se necesita una gran seguridad y fiabilidad como la defensa, la aeronáutica (Boeing o Airbus), la gestión del tráfico aéreo (como Indra en España) y la industria aeroespacial entre otros.



Figura 1: Logo Ada

El código Ada que hemos realizado se encuentra en el fichero “Ada.adb” adjunto. En este primer hito se definió la gramática para el lenguaje indicado, y se procesa un programa válido, esto se indica al terminar la ejecución del mismo.

Este programa contiene las siguientes funcionalidades:

- Operaciones aritméticas con números enteros y reales
- Operaciones booleanas
- Un bucle while
- Sentencia if
- Comentarios
- Control de errores en las operaciones indicadas

Flex

Una vez terminado el programa en Ada, se tuvo que realizar el analizador léxico en flex. Este fichero en cuestión tendrá que leer el archivo que he creado anteriormente en Ada, analizarlo y mostrar los resultados en otro fichero de salida.

FLEX (Fast Lexical Analyzer Generator) es un programa de herramienta / computador que genera analizadores léxicos (escáneres o lexers) escrito en C. Se utiliza junto con el generador Berkeley Yacc o el generador de analizador GNU Bison. Flex y Bison son más flexibles que Lex y Yacc y producen un código más rápido, y por esto es por lo que decidimos trabajar con ellos.

Bison produce un analizador a partir del archivo de entrada proporcionado por el usuario. La función yylex () es generada automáticamente por el flex cuando se le proporciona un archivo .l y el analizador espera que esta función yylex () llame para recuperar tokens de la corriente actual / este token.

La función yylex () es la función flex principal que ejecuta la sección de reglas y la extensión (.l) es la extensión utilizada para guardar los programas.



Figura 2: Logo Flex

El código en flex se divide en tres zonas:

- Definición
- Reglas
- Subrutinas de usuario

Y tiene la siguiente estructura:

```
{definición}

%%

{reglas}

%%

{subrutinas de usuario}
```

Definición

En este apartado se indican las declaraciones que son previas al uso del scanner.

```
%{
#include "prueba.tab.h"
%}
```

Estas líneas de código se copian exactamente igual en la cabecera del fichero lex.yy.c, en este caso se declara que tiene que estar incluido la clase “prueba.tab.h”

```
%option noyywrap
```

Ahora se declara las directivas, donde se utiliza *noyywrap* para utilizar Lex (Lexical Analyzer Generator) sin tener que definir la función yywrap que maneja varios ficheros de entrada.

```
digito [0-9]
letra [a-zA-Z]
letreysimb [a-zA-Z_\.]
```

Y por último en este apartado, se declaran las definiciones. Donde se nombra a los patrones para facilitar la legibilidad.

Reglas

En este apartado se codifican los patrones de entrada y acciones que se realizan cuando coincide la entrada.

Para obtener el token deseado se siguió la siguiente estructura:

```
"+" return MAS;
```

Donde buscaba la cadena literal en el código, en este caso “+”. Una vez que encontrada una coincidencia devuelve al fichero BISON ese token. Este proceso se ha realizado para todos los símbolos terminales de la gramática que se piden en el enunciado y el resultado final se muestra en la figura 3.

```
%%
[ \t]+ ;

 "+"      return MAS;
 "_"      return MENOS;
 "*"      return POR;
 "/"      return DIV;
 "("      return PAR_I;
 ")"      return PAR_D;
 "if"     { return IF; }
 "then"   { return THEN; }
 ">"     return MAYOR_Q;
 "<"     return MENOR_Q;
 "else"   return ELSE;
 "procedure" return PROCEDURE;
 "is"     return IS;
 "end"    return END;
 "begin"  return BEGINN;
 ":"      return DOSPUNTOS;
 "Integer" return INTEGER;
 "Float"   return FLOAT;
 "String"  return STRING;
 "Boolean" return BOOLEAN;
 ";"      return DOSPUNTOS_IGUAL;
 "("      return PIZQ_COM;
 ")"      return PDERCH_COM;
 "Put_Line" return PUTLINE;
 "end if" return ENDIF;
 "True"   return TRUE;
 "False"  return FALSE;
 "__"     return COMENTARIO;
 "while"  return WHILE;
 "loop"   return LOOP;
 "end loop" return ENDLOOP;
 "="      return IGUAL;
 "for"    return FOR;
 "in"     return IN;
 ".."     return RANGO;
 "function" return FUNCION;
 "return" return RETURN;
 |
```

Figura 3: Identificadores

Por otro lado, en este mismo apartado, se identificó los tokens “identificador”, “real” y “entero” aplicando la nomenclatura que se declaró en el apartado anterior mediante el uso de expresiones regulares.

```
{letra}({letra}|{digito})* yyval.string=strdup(yytext); return IDENTIFICADOR;
{letraysimb}+ return IDENTIFICADORSIMB;

[-+]?{digito}+ yyval.number=atoi(yytext); return NUMENTERO;
[-+]?{digito}+({.{digito}}+)?((E|e)[-+]?{digito}+)? yyval.numberf=atof(yytext); return NUMREAL;
```

Figura 4: Atoi y Atof

La función atoi y atof han sido utilizadas para convertir las respectivas cadenas a su valor numérico, ya sea entero (atoi) o de coma flotante (atof)

Código de usuario

En este último apartado se incluye el código programado en C que es copiado al fichero lex.yy.c. Para realizar este proyecto no se incluyó nada ya que las operaciones en C se realizaban en Bison.

Bison

Bison (Yet Another Compiler-Compiler) es una herramienta para generar analizadores sintácticos para gramáticas LALR.

Al realizar análisis sintácticos haciendo uso de SLR y LR(1), es importante saber que los LR(1) trabajan con gramáticas que son ambiguas para los SLR, aunque para ello necesitan generar autómatas con más estados. Esto se debe a la utilización de los símbolos de anticipación.

Podemos ver como estados que tienen el mismo núcleo, son diferentes debido a los símbolos de anticipación. Con los analizadores sintácticos LALR se trata de conseguir los beneficios que proporciona el método LR(1), pero con el coste del método SLR y utilizando, por tanto, el menor número de estados que este último necesita. Es decir, este método consigue reducir el número de estados.

Se debe tener en cuenta que, para un lenguaje como Pascal, estamos hablando de varios cientos de estados. Pongamos un ejemplo: para una gramática pequeña el SLR genera 11 estados y el LR(1) necesitaba 15, podemos ver que necesitamos un 25% más de estados y por tanto de coste a la hora de obtener los mismos resultados. No todos son beneficios con el LALR.

Como hemos demostrado, el número de estados del LALR, se reduce hasta alcanzar el del autómata SLR, lo cual permite mejorar en eficiencia de recursos y en tiempo a la hora de reconocer una sentencia. Además, hemos podido comprobar que la construcción de la tabla de análisis es idéntica a la del LR(1) del que procede.

Finalmente, y a través de un ejemplo completo, hemos podido ver todos los pasos para llegar desde el autómata SLR al LALR, pasando por el LR(1).

La forma natural de trabajar en este aspecto es en conjunto con el lenguaje FLEX. Su estructura principal es:

```
{definición}  
%%  
{reglas}  
%%  
{subrutinas de usuario}
```

Definición

En este apartado se añaden las sentencias previas al uso del scanner. Se puede dividir en dos secciones diferentes, el código de usuario y las declaraciones.

En la sección del código de usuario se incluye la parte que se copia a la salida. En este caso ha sido:

```
%{
#include <ctype.h>
#include <stdio.h>

extern FILE *yyin;
extern FILE *yyout;
%}
```

Donde se indican las bibliotecas necesarias y se declaran los ficheros de salida y entrada.

En la sección de declaraciones, se incluye:

- **Terminales y no terminales:** en este caso estos son los token que se analizan en este hito.

```
%token NUMENTERO, MAS, MENOS, POR, DIV, PAR_I, PAR_D, IF, THEN, MAYOR_Q, MENOR_Q, ELSE,
SALTOLINEA, PROCEDURE, IS, IDENTIFICADOR, END, BEGINN, DOSPUNTOS, ENTERO, FLOAT, STRING,
BOOLEAN, IDENTIFICADORSIMB, PIZQ_COM, DOSPUNTOS_IGUAL, PDECH_COM, PUTLINE, ENDIF, NUMREAL,
TRUE, FALSE, COMENTARIO, WHILE, LOOP, ENDLOOP, IGUAL, FOR, IN, RANGO
```

Figura 5: Código token

- **Símbolo de arranque de la gramática:** en este caso se indica que el axioma de la gramática es “programa”

```
%start programa
```

Reglas

En este apartado se añaden las reglas sintácticas y lo que se debe hacer cuando se activan. En este caso, desde el primer momento se tuvo claro la estructura que sigue el lenguaje ADA, con esto nos referimos a que comienza con “procedure”, después le siguen las declaraciones....

Por lo tanto, en esta sección se incluyó una regla por cada estructura clave del lenguaje ADA. Por ejemplo:

```
cuerpo : inicio SALTOLINEA sentencias fin;
```

En todo lenguaje ADA, tiene como encabezado “procedure x is”, que se especificará en la regla denominada como “inicio”, en la regla “sentencias” irán todas las demás líneas de código y en la regla “fin”, irá “end x”. Esta estructura será obligatoria, cualquiera que no cumpla esta regla no se le considerará como gramática válida, y por lo tanto no

sería un código en ADA correcto. En la figura 6 se puede mostrar algunas de estas reglas.

```

programa : cuerpo
;
cuerpo : inicio SALTOLINEA sentencias fin
;

inicio : PROCEDURE IDENTIFICADOR IS {fprintf(yyout, "Inicio -> procedure\n");}
;
fin : END IDENTIFICADOR {fprintf(yyout, "Final -> end\n");}
;
sentencias : declaraciones SALTOLINEA comienzo SALTOLINEA sentencia SALTOLINEA
;

comienzo: BEGINN {fprintf(yyout, "Comienzo -> begin\n");}
;

declaraciones : declaraciones SALTOLINEA declaracion
| declaracion
;
declaracion : IDENTIFICADOR DOSPUNTOS tipo
:|
tipo: INTEGER {fprintf(yyout, "Declaracion -> int\n");}
|FLOAT {fprintf(yyout, "Declaracion -> float\n");}
|STRING {fprintf(yyout, "Declaracion -> string\n");}
|BOOLEAN {fprintf(yyout, "Declaracion -> boolean\n");}
;

sentencia : sentencia SALTOLINEA expr
| expr
;

```

Figura 6: Ada

Se continuó evaluando el resto de la estructura, declaraciones, begin, end... y además todas las posibles sentencias que podría haber. En este caso para limitar el proyecto se decidió detectar únicamente:

- Sentencia if
- Put_Line
- Operaciones aritméticas
- Comentarios
- Bucle while
- Bucle for

En la figura 7 se puede mostrar las diferentes reglas que identifican las sentencias anteriores.

```

sentencia : sentencia SALTOLINEA expr.
           | expr
;

expr: operaciones
|sentencia_if {fprintf(yyout, "Sentencia IF\n");}
|PUTLINE_PIZQ_COM IDENTIFICADOR PDECH_COM {fprintf(yyout, "Put_Line\n");}
|IDENTIFICADOR DOSPUNTOS_IGUAL factor {fprintf(yyout, "Asignacion\n");}
|COMENTARIO IDENTIFICADOR {fprintf(yyout, "Comentario\n");}
|bucle_while {fprintf(yyout, "Bucle WHILE\n");}
|bucle_for {fprintf(yyout, "Bucle FOR\n);}

;

operaciones: IDENTIFICADOR DOSPUNTOS_IGUAL factor MAS factor {fprintf(yyout, "Operación SUMA\n");}
|IDENTIFICADOR DOSPUNTOS_IGUAL factor MENOS factor {fprintf(yyout, "Operación MENOS\n");}
|IDENTIFICADOR DOSPUNTOS_IGUAL factor POR factor {fprintf(yyout, "Operación MULTIPLICACION\n");}
|IDENTIFICADOR DOSPUNTOS_IGUAL factor DIV factor {fprintf(yyout, "Operación DIVISION\n");}
|factor {fprintf(yyout, "expr --> term\n");}
;

factor: NUMENTERO {fprintf(yyout, " factor--> NUMENTERO(%d)\n", $1);}
| NUMREAL {fprintf(yyout, " factor--> NUMREAL(%d)\n", $1);}
|IDENTIFICADOR {fprintf(yyout, " factor --> variable\n");}
|TRUE {fprintf(yyout, " factor --> True\n");}
|FALSE {fprintf(yyout, " factor --> False\n");}
;

sentencia_if: IF condicion THEN SALTOLINEA sentencia SALTOLINEA ENDIF
           | IF condicion THEN SALTOLINEA sentencia SALTOLINEA ELSE SALTOLINEA sentencia SALTOLINEA ENDIF

;

bucle_while: WHILE condicion LOOP SALTOLINEA sentencia SALTOLINEA ENDOLOOP
;

bucle_for: FOR factor IN rangos LOOP SALTOLINEA sentencia SALTOLINEA ENDOLOOP
;
rangos: factor RANGO factor {fprintf(yyout, "Variable\n");}
;
condicion: factor MAYOR_Q factor {fprintf(yyout, "Condición\n");}
           | factor MENOR_Q factor {fprintf(yyout, "Condición\n");}
           | factor IGUAL factor {fprintf(yyout, "Condición\n");}
;

```

Figura 7

En este punto cabe destacar el principal problema con el que nos encontramos, la recursividad. Ya que tanto en la regla de “declaraciones” como en la de “sentencias” es necesario aplicar recursividad porque puede haber más de una. Para ello se aplicó el código que se muestra en la figura 8.

```

sentencia : sentencia SALTOLINEA expr.
           | expr
;
```

Figura 8: Recursividad 1

```

declaraciones : declaraciones SALTOLINEA declaracion
               | declaracion
;

declaracion : IDENTIFICADOR DOSPUNTOS tipo {
};

;
```

Figura 9: Recursividad 2

De esta manera se consiguió la identificación de una y más declaraciones, al igual que la detección de una o más sentencias.

Subrutinas de usuario

En este apartado se incluyen las funciones en C que se copian al fichero de salida. En este caso, se incluyó tanto la lectura del fichero de entrada como la escritura del fichero salida. El fichero de entrada en cuestión se llama “Ada.adb” y en él se encuentra el código programado en lenguaje ADA.

Por otro lado, el fichero de salida donde se escriben los resultados que hemos indicado en el apartado anterior recibe el nombre de “salidaAda.txt”. En la figura 10 se muestra el código de esta sección.

```
%%
int main(int argc, char *argv[]) {
    if (argc == 1) {
        yyparse();
    }
    if (argc == 2) {
        yyout = fopen( "./salidaAda.txt", "wt" );
        yyin = fopen(argv[1], "rt");
        yyparse();
    }
    return 0;
}

yyerror()
{
} |
```

Figura 10: Fichero Salida

Hito 2

Se incorporarán controles semánticos al compilador y se generará código intermedio mediante Árboles de Sintaxis Abstracta (ASA). Además, se enviará este código intermedio a un fichero de salida.

Flex

En este hito, en Flex se han añadido más símbolos terminales que son interesantes de analizar, principalmente han sido terminales para identificar las funciones en lenguaje ADA. Estos cambios se pueden observar en la figura 11. Hemos asociado cada símbolo a un tipo de operación aritmética en los casos de la suma, resta, multiplicación y división. Por otro lado, en este hito también hemos incorporado en el archivo Flex otras declaraciones como condiciones booleanas, condiciones if, else, y bucles, entre otros.

```

"+"          return MAS;
"_"          return MENOS;
"*"          return POR;
"/"          return DIV;
 "("          return PAR_I;
 ")"          return PAR_D;
 "if"         { return IF; }
 "then"       { return THEN; }
 ">"          return MAYOR_Q;
 "<"          return MENOR_Q;
 "else"       return ELSE;
 "procedure"  return PROCEDURE;
 "is"         return IS;
 "end"        return END;
 "begin"      return BEGINN;
 ":"          return DOSPUNTOS;
 "Integer"    return INTEGER;
 "Float"      return FLOAT;
 "String"     return STRING;
 "Boolean"    return BOOLEAN;
 "=="         return DOSPUNTOS_IGUAL;
 "(("          return PIZQ_COM;
 "))"         return PDCH_COM;
 "Put_Line"   return PUTLINE;
 "end if"    return ENDIF;
 "True"       return TRUE;
 "False"      return FALSE;
 "--"         return COMENTARIO;
 "while"      return WHILE;
 "loop"       return LOOP;
 "end loop"   return ENDLOOP;
 "="          return IGUAL;
 "for"         return FOR;
 "in"          return IN;
 ".."          return RANGO;
 "function"   return FUNCION;
 "return"     return RETURN;

```

Figura 11: Identificadores

Además de modificar los tokens identificador, numero entero y numero real, para adaptarlo a los valores semánticos que se han declarado en Bison. Que se realizaría de la siguiente manera:

```

{letra}({letra}|{dígito})* yylval.string=strdup(yytext); return IDENTIFICADOR;
{letraysimb}+ return IDENTIFICADORSIMB;

[-+]?{dígito}+ yylval.number=atoi(yytext); return NUMENTERO;

[-+]?{dígito}+({dígito}+)?((E|e)[-+]?{dígito}+)? yylval.numberf=atof(yytext); return
NUMREAL;

```

Bison

Este archivo se modificó para este hito 2, se incorporó las nuevas funcionalidades que estaban indicadas en el enunciado. Principalmente, esta etapa se basó en la creación de la tabla de símbolos y la generación de árboles.

Definición

En la sección del código de usuario se incluye la parte que se copia a la salida. En este caso se realizó la declaración de todas las variables que se utilizaban a lo largo del código. En la figura 12, se muestra primero la inclusión de todas las librerías y clases necesarias para la compilación del código. Entre ellas destacan “syntab.h” y “AST.h”, que son los archivos donde se encuentran la tabla de símbolos y la generación de árboles respectivamente. Que se explicará más adelante con mayor profundidad.

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "symtab.h"
#include "AST.h"

extern FILE *yyin;
extern FILE *yyout;

int globalError =0;
char *globalTipo;
```

Figura 12: Código de usuario

Después se declararon las funciones para añadir y actualizar datos en la tabla de símbolos. Por un lado teníamos la inserción y actualización de valores numéricos a la tabla. Un ejemplo podría ser:

```
Numero :Integer;
Numero := 10;
Numero := 10 + 5;
```

En este ejemplo, se comprobaría si la variable “Numero” se encuentra en la tabla, al no ser así se añade, el nombre “Numero”, el valor “10” y el tipo “Integer”. En la tercera sentencia vemos como el valor de esta variable cambia a 15. Por lo tanto en este caso se utiliza la función “Update” donde busca la variable en concreto y la actualiza al nuevo valor, que en este caso será 15. En la figura 13 se muestra el código integrado en BISON.

```
-----NUMEROS-----
add_SymNum( char *sym_name, int sym_val, char *sym_type ) {
symrec *s;
s = getsymNum( sym_name,sym_val );
if ( s == 0){
    s = putsymNum( sym_name,sym_val, sym_type );
    printf( "%s no esta definida, es %d. (tipo=%s)\n", sym_name, sym_val, sym_type );
}else {
    printf( "%s ya esta definida, es %d. (tipo=%s)\n", sym_name, sym_val, sym_type );
}
}

Update_SymNum( char *sym_name, int sym_val ){
symrec *act;

if ( getsymNum( sym_name, sym_val ) == 0 ) {
    printf( "%s no esta en la lista, es %d\n", sym_name, sym_val );
}else{
    act = updatesymNum( sym_name, sym_val );
    printf( "%s se actualiza a: %d\n", act->name, act->num );
}
}

-----TEXTOS-----
add_SymText( char *sym_name, char *sym_text, char *sym_type ) {
symrec *s;
s = getsymText( sym_name,sym_text );
if ( s == 0){
    s = putsymText( sym_name,sym_text, sym_type );
    printf( "%s no esta definida, es %.s. (tipo=%s) \n", sym_name, sym_text, sym_type );
}else {
    printf( "%s ya esta definida, es %.s. (tipo=%s) \n", sym_name, sym_text, sym_type );
}
}

Update_SymText( char *sym_name, char *sym_text ){
symrec *act;

if ( getsymText( sym_name, sym_text ) == 0 ) {
    printf( "%s no esta en la lista, es %d\n", sym_name, sym_text );
}else{
    act = updatesymText( sym_name, sym_text );
    printf( "%s se actualiza a: %.s\n", act->name, act->text );
}
}
```

Figura 13: BISON

Por otro lado, se encuentra la inserción y actualización de texto en la tabla de símbolos. Esto se utilizó para algunas comprobaciones que se pedían en el enunciado. Por ejemplo que el nombre del procedimiento fuera el mismo al principio y al final. Funciona igual que el proceso anterior, lo único que en vez de añadir un valor numérico se añade una cadena de caracteres.

En la sección de declaraciones, se incluye:

- **Terminales y no terminales:** En esta parte se incluyeron varios cambios con respecto al anterior hito.

Se añadió los tokens “Funcion” y “Return” para conseguir identificar las funciones en el lenguaje ADA

```
%token MAS, MENOS, POR, DIV, PAR_I, PAR_D, IF, THEN, MAYOR_Q, MENOR_Q, ELSE, SALTOLINEA, PROCEDURE, IS, END, BEGINN,  
DOSPUNTOS, INTEGER, FLOAT, STRING, BOOLEAN, IDENTIFICADORSIMB, PIZQ_COM, DOSPUNTOS_IGUAL, PDECH_COM, PUTLINE, ENDIF,  
TRUE, FALSE, COMENTARIO, WHILE, LOOP, ENDLOOP, IGUAL, FOR, IN, RANGO, FUNCION, RETURN
```

Figura 14: Tokens

Para introducir valores semánticos se añadió %union, aquí se declaró tres variables y un struct. Un struct es una tabla de elementos, cada una con su nombre de campo y su tipo. La operación de acceso a un registro es mediante el operador punto. Además dentro de ese struct se declaró dos más, struct ast *a para el árbol de operaciones aritméticas y struct flow *f para sentencias if y bucles while.

```
%union {  
    int number;  
    double numberf;  
    char *string;  
    struct {  
        char *tipo;  
        int valor;  
        double valord;  
        char *texto;  
        int booleanCond;  
        struct ast *a;  
        struct flow *f;  
    } snum;  
}
```

Figura 15 % Union

Se asoció los valores semánticos del %union a los token: “NUMENTERO”, “NUMREAL”, “IDENTIFICADOR”, que al ser terminales se realizó de la siguiente manera:

```
%token <number> NUMENTERO  
%token <numberf> NUMREAL  
%token <string> IDENTIFICADOR
```

Por otro lado, para asociar los valores semánticos a las reglas, es decir, para un no terminal se realizó de la siguiente manera:

```
%type<snum> bucle_while sentencia_if  
%type<snum> nombreF declaracion calc expr Fun
```

- **Símbolo de arranque de la gramática:** al igual que con el hito 1 el axioma de la gramática es “programa”

```
%start programa
```

Reglas

En este apartado con respecto al anterior hito se incorporó la utilización de la tabla de símbolos además de la generación de árboles. Asimismo, se añadió las comprobaciones indicadas en el enunciado.

El primer paso que se hizo fue integrar las funciones que se han comentado anteriormente para añadir y actualizar la tabla de símbolos, sin esto sería imposible realizar algunas de las comprobaciones indicadas.

Para simplificar la explicación de este apartado se ha dividido según las comprobaciones que se han añadido al código.

— **La comprobación de tipos**

Se realizó la comprobación de que tipo era cada variable a la hora de realizar una operación. Se muestra un ejemplo de ello en la figura 16.

```
calc: calc MAS calc {
    printf("Suma\n");
    if (globalError == 0) { //No hay errores
        if (($1.tipo == "entero") && ($3.tipo == "entero")) {
            $$ .a = newast('+', $1.a,$3.a);
            globalTipo = $1.tipo;
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($1.tipo == "real") && ($3.tipo == "real")){
            $$ .a = newast('+', $1.a,$3.a);
            globalTipo = $1.tipo;
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($1.tipo == "string")){
            $$ .a = newast('+', $1.a,$3.a);
            globalTipo = $3.tipo;
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($3.tipo == "string")){
            $$ .a = newast('+', $1.a,$3.a);
            globalTipo = $1.tipo;
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
    } else {
        printf("HAY ERRORES NO SE REALIZA LA SUMA\n");
    }
}
```

Figura 16: Comprobación de tipos

En ella se muestra la comprobación de tipos en la operación suma. Donde se comprobaba de que tipo era cada operador, esto sirvió para tener en cuenta que se sumaba tipos iguales, por ejemplo, entero y entero o real y real. Para a la hora de

almacenar el valor en la tabla de símbolos guardar su respectivo tipo, para ello se empleó la variable “globalTipo”.

— El nombre del procedimiento es el mismo al principio y al final

```

inicio : PROCEDURE nombreI IS {fprintf(yyout, "Inicio -> procedure\n");}
;

fin : END nombreF {fprintf(yyout, "Final -> end\n");}
;

nombreI: IDENTIFICADOR {fprintf(yyout, "Nombre Inicio\n"); add_SymText ( "Nombre", $1, "string" )}
;

nombreF: IDENTIFICADOR {
    fprintf(yyout, "Nombre Final\n");
    if (strcmp($1, getvalsymText("Nombre")) == 0) {
        printf("CORRECTO - Coinciden los nombres");
    } else {
        printf("ERROR - No es igual el nombre del inicio y el del final");
    }
}
;

```

Figura 17: Comprobaciones

Para realizar esta comprobación se guardó en primer lugar el nombre de inicio del programa en ADA, en la tabla de símbolos. Se almacenó como variable “Nombre”. En la regla “fin” para realizar la comprobación de que tiene el mismo nombre final se busca el valor de esa variable. Si coincide el mismo, es correcto y sino se informa del error.

— Comprobación de que una variable existe antes de haber sido declarada

```

expr: IDENTIFICADOR DOSPUNTOS_IGUAL calc {
    $$ .texto= $1;

    if (globalError ==0) { //No hay errores
        if (getvalsymText($$.texto) == 0 ) {
            printf("ERROR - No se ha declarado la variable: %s \n", $$ .texto);
        } else {
            printf("CORRECTO - Se ha declarado la variable: %s \n", $$ .texto);
            add_SymNum( $1, eval($3.a), globalTipo);
            Update_SymNum( $1, eval($3.a) );
            printf("RESULTADO - De %s: %4.4g\n", $1, eval($3.a));
        }
    } else { //HAY
        globalError =0;
    }
}

```

Figura 18: Comprobación de variable existente

En “expr” se realizan las operaciones de suma, resta, multiplicación y división, por ello lo primero que hay que hacer es la comprobación de que esté declarada la variable donde se va a almacenar el valor. Para ello se busca la variable en la tabla, si no se encuentra envía un error de que no se ha declarado la variable, de sí encontrarse, se realizará las operaciones y además se actualizará la variable con el nuevo valor.

— No se pueden utilizar variables no declaradas previamente

```
declaraciones : declaraciones SALTOLINEA declaracion
    | declaracion
;

declaracion : IDENTIFICADOR DOSPUNTOS tipo {
    printf("Se declara la variable: %s \n", $1);
    $$ . texto = $1; add_SymText ( $$ . texto, $$ . texto, globalTipo);
}
;
```

Figura 19: Control de uso de variables

En esta sección, se añade en la tabla de símbolos todas las variables declaradas. Para posteriormente realizar la comprobación de si se encuentra o no ella, como se puede observar en el anterior ejemplo.

— No se permitirán divisiones por 0

```
| calc DIV calc {
    printf("Division\n");
    if ($3.valor == 0) { //No se puede dividir entre 0
        globalError =1;
    }
}
```

Figura 20: Control de divisiones entre 0

— Comprobar que no existen funciones con el mismo nombre.

```
funcion: FUNCION nombreFuncion PAR_I declaraciones PAR_D RETURN tipo IS SALTOLINEA BEGINN SALTOLINEA sentencia
SALTOLINEA Fun SALTOLINEA END
;

nombreFuncion: IDENTIFICADOR {
    if (getvalsymNum($1) == 0 ) {
        add_SymNum($1, 1 , "string");
        printf("CORRECTO - No existe ninguna función con ese nombre\n");
    } else {
        printf("ERROR - Existe ya una función con ese nombre\n");
    }
    fprintf(yyout, "Nombre función\n");
}
;
```

Figura 21: Comprobación de funciones con el mismo nombre

En la parte de operaciones “calc”, más concretamente en la parte de división se realizará la comprobación de que el denominador no sea 0. Si es así reportará un error y no se realizará la división.

— Comprobar que la sentencia “return” existe

```
funcion: FUNCION nombreFuncion PAR_I declaraciones PAR_D RETURN tipo IS SALTOLINEA BEGINN SALTOLINEA sentencia SALTOLINEA Fun
SALTOLINEA END
;
```

Figura 22: Comprobación de sentencia “return”

Al realizar la identificación de la función se realiza también la comprobación de que exista la sentencia “return”, de no ser así no se identificará la función.

- Comprobar que el tipo devuelto es del mismo tipo que el especificado en la función

```

Fun: RETURN IDENTIFICADOR {

    if (getvalsymText($2) == 0 ) {
        printf("ERROR - No se ha declarado la variable: %s \n", $2);
    } else {
        printf("Tiene al menos un return\n");
        if (strcmp(globalTipo, gettypesymText($2)) == 0) {
            printf("CORRECTO - Coincide el tipo de return\n");
        } else {
            printf("ERROR - No coincide el tipo de return\n");
        }
    }
}
;

```

Figura 23: Comprobación de tipos

Se realiza la comprobación de si la variable que ha sido declarada en la función coincide con la variable que se devuelve en el return. Si es así, devuelve que ha sido todo correcto pero si no es así el programa informa de un error.

Subrutinas de usuario

En esta sección no se realizó ninguna modificación con respecto al hito 1, se continuó realizando de la misma manera la lectura y la escritura de archivos.

SymTab.h

Este archivo está dedicado a la creación de la tabla de símbolos. Dentro de este archivo tenemos funciones que añaden y actualizan los valores de las variables numéricas y de texto que están contenidas en la tabla de símbolos.

Se ha creado una estructura con los atributos básicos de la tabla de símbolos, junto con un puntero para poder hacer las referencias.

```

struct symrec {
    char *name;           /* name of symbol      */
    int *num;             /* val of symbol       */
    char *text;            /* text of symbol      */
    char *type;            /* type of symbol      */
    struct symrec *next;  /* link field          */
};


```

Figura 24 Symtab 1

Como premisa, la tabla de símbolos se inicializa vacía

```
symrec *sym_table = (symrec *)0;
```

Figura 25: Inicialización de tabla de símbolos

Las funciones realizadas en este archivo son las siguientes:

- **symrec *putsymNum ()**: añade un valor numérico a la tabla de símbolos.
Almacena el nombre de la variable, el valor y el tipo.

```
symrec * putsymNum ( char *sym_name, int sym_val, char *sym_type ) {
    //printf("AÑADIR: %s ... %d ...||||| ", sym_name, sym_val);

    symrec *ptr;
    ptr = (symrec *) malloc (sizeof(symrec));
    ptr->name = (char *) malloc (strlen(sym_name)+1);
    ptr->num = (int) malloc ((sym_val)+1);
    ptr->type = (char *) malloc (strlen(sym_type)+1);
    strcpy (ptr->name,sym_name);
    ptr->num = sym_val;
    strcpy (ptr->type, sym_type);
    ptr->next = (struct symrec *)sym_table;
    sym_table = ptr;
    return ptr;
}
```

Figura 26: Configuración de tabla de símbolos

- **symrec *getsymNum ()**: busca el nombre de la variable de carácter numérico en la tabla de símbolos.

```
symrec * getsymNum ( char *sym_name, int sym_val) {
    symrec *ptr;
    for (ptr = sym_table; ptr != (symrec *) 0;
        ptr = (symrec *)ptr->next)
        if (strcmp (ptr->name,sym_name) == 0) {

            sym_val = ptr->num;
            sym_name = ptr->name;
            //printf("get %s ... %d ...||||| ", ptr->name, ptr->num);
            return ptr;
        }
    return 0;
}
```

Figura 27: Configuración de tabla de símbolos 2

- **symrec * updatesymNum()**: realiza una búsqueda de la variable en la tabla de símbolos y si se encuentra, actualiza su valor numérico.

```
symrec * updatesymNum ( char *sym_name, int sym_val) {
    symrec *ptr;
    for (ptr = sym_table; ptr != (symrec *) 0;
        ptr = (symrec *)ptr->next)
        if (strcmp (ptr->name,sym_name) == 0) {
            ptr->num = sym_val;
            //printf("update %s ... %d ...||||| ", ptr->name, ptr->num );
            return ptr;
        }
    return 0;
}
```

Figura 28: Configuración de tabla de símbolos 3

- **symrec * getvalsymNum()**: realiza una búsqueda de la variable en la tabla de símbolos y si se encuentra, devuelve la referencia de su valor numérico.

```
symrec * getvalsymNum ( char *sym_name ) {
    symrec *ptr;
    for (ptr = sym_table; ptr != (symrec *) 0;
        ptr = (symrec *)ptr->next)
        if (strcmp (ptr->name,sym_name) == 0) {
            //printf("get val of %s ... %d ...||||| ", ptr->name, ptr->num);
            return ptr->num;
        }
    return 0;
}
```

Figura 29: Comprobación de tabla de símbolos 4

- **symrec *putsymText ()**: añade un texto numérico a la tabla de símbolos.
Almacena el nombre de la variable, el texto y el tipo.

```
symrec * putsymText ( char *sym_name, char *sym_text, char *sym_type ) {
    //printf("ANADIR: %s ... %s ...||||| ", sym_name, sym_text);

    symrec *ptr;
    ptr = (symrec *) malloc (sizeof(symrec));
    ptr->name = (char *) malloc (strlen(sym_name)+1);
    ptr->text = (char *) malloc (strlen(sym_name)+1);
    ptr->type = (char *) malloc (strlen(sym_type)+1);
    strcpy (ptr->name,sym_name);
    strcpy (ptr->text,sym_text);
    strcpy (ptr->type, sym_type);

    ptr->next = (struct symrec *)sym_table;
    sym_table = ptr;
    return ptr;
}
```

Figura 30: Comprobación de tabla de símbolos 5

- **symrec * getsymText ()**: busca el nombre de la variable que almacena una cadena de caracteres en la tabla de símbolos.

```
symrec * getsymText ( char *sym_name, char *sym_text) {  
    symrec *ptr;  
    for (ptr = sym_table; ptr != (symrec *) 0;  
        ptr = (symrec *)ptr->next)  
        if (strcmp (ptr->name,sym_name) == 0) {  
            sym_text = ptr->text;  
            sym_name = ptr->name;  
            //printf("get %s ... %s ...||||| ", ptr->name, ptr->text);  
            return ptr;  
        }  
    return 0;  
}
```

Figura 31: Comprobación de tabla de símbolos 6

- **symrec * updatesymText()**: realiza una búsqueda de la variable en la tabla de símbolos y si se encuentra, actualiza el contenido del texto.

```
symrec * updatesymText ( char *sym_name, char *sym_text) {  
    symrec *ptr;  
    for (ptr = sym_table; ptr != (symrec *) 0;  
        ptr = (symrec *)ptr->next)  
        if (strcmp (ptr->name,sym_name) == 0) {  
            ptr->text = sym_text;  
            //printf("update %s ... %s ...||||| ", ptr->name, ptr->text );  
            return ptr;  
        }  
    return 0;  
}
```

Figura 32: Configuración de tabla de símbolos 7

- **symrec * getvalsymText()**: realiza una búsqueda de la variable en la tabla de símbolos y si se encuentra, devuelve la referencia de su texto.

```
symrec * getvalsymText ( char *sym_name ) {  
    symrec *ptr;  
    for (ptr = sym_table; ptr != (symrec *) 0;  
        ptr = (symrec *)ptr->next)  
        if (strcmp (ptr->name,sym_name) == 0) {  
            //printf("get val of %s ... %s ...||||| ", ptr->name, ptr->text);  
            return ptr->text;  
        }  
    return 0;  
}
```

Figura 33: Configuración de tabla de símbolos 8

- **symrec * gettypesymText()**: realiza una búsqueda de la variable en la tabla de símbolos y si se encuentra, obtiene su atributo tipo.

```

symrec * gettypesymText ( char *sym_name ) {
    symrec *ptr;
    for (ptr = sym_table; ptr != (symrec *) 0;
        ptr = (symrec *)ptr->next)
        if (strcmp (ptr->name,sym_name) == 0) {
            //printf("get type of %s ... %s ...|||| ", ptr->name, ptr->type);
            return ptr->type;
        }
    return 0;
}

```

Figura 34: Configuración de tabla de símbolos 9

AST.h

El uso de árboles sintácticos como representación intermedia permite que la traducción se separe del análisis sintáctico. Es importante indicar, que los tipos de árboles que nos interesan son los Árboles de Sintaxis Abstracta (ASA) que es un árbol de análisis sintáctico al que se le han eliminado los símbolos superfluos y es muy útil para representar las construcciones del lenguaje. en un ASA nos interesa las operaciones y los operadores y es por eso por lo que eliminamos los símbolos superfluos (terminales de la gramática) generándose un árbol condensado.

Creación de árboles

Se genera un nodo para cada operador y cada operando
 Los hijos de un nodo operador son las raíces de los nodos que representan las subexpresiones que constituyen los operandos de dicho operador.

Para llevarlo a la práctica, necesitamos unas funciones auxiliares:

- **hazNodo (operador, izquierda, derecha)**
- **hazHoja (id, entrada)**
- **hazHoja (num, val)**

Se debe tener en cuenta que hay una gran tipología de expresiones a la hora de diseñar el ASA, no solo las binarias.

Este archivo está dedicado única y exclusivamente a la creación , inicialización y funcionamiento de los árboles.

Dentro se han creado los árboles:

- **ast**: árbol por defecto, el cual marca la estructura básica para los siguientes árboles.

```
struct ast {
    int nodetype;
    struct ast *l;
    struct ast *r;
};
```

Figura 35: Creación del árbol

- **numval:** árbol donde se almacenan las constantes

```
struct numval {
    int nodetype;
    double number;
};
```

Figura 36: Árbol de constantes

Después los árboles se inicializan

```
/* construir un AST*/
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newnum(double d);
```

Figura 37: Inicialización de los árboles

Finalmente creamos la estructura del árbol basándonos en el árbol base.

- **Funcionalidad árbol operaciones:** Donde se recoge el operador y los números de la operación.

```
struct ast * newast(int nodetype, struct ast *l, struct ast *r) {
    struct ast *a = malloc(sizeof(struct ast));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype; a->l = l;
    a->r = r;

    //printf("NEW AST %s ..|| ", a);

    return a;
}
```

Figura 38: Estructura del árbol

- Para el **árbol numval**, se crea la funcionalidad. Donde se recoge las constantes.

```

struct ast * newnum(double d) {
    struct numval *a = malloc(sizeof(struct numval));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'K';
    a->number = d;
    //printf("NEW NUM: %s .. %lf ..|| ", a, a->number);

    return (struct ast *)a;
}

```

Figura 39: Recogida de constantes

Por otro lado, se han creado dos funciones, una para evaluar el contenido del árbol y otra para liberarlo.

- **Función de evaluación:** se realizan las operaciones aritméticas para posteriormente mostrar su resultado

```

double eval(struct ast *a) {
    double v;

    switch(a->nodetype) {
        case 'K': v = ((struct numval *)a)->number;
                    break;
        case '+': v = eval(a->l) + eval(a->r);
                    break;
        case '-': v = eval(a->l) - eval(a->r);
                    break;
        case '*': v = eval(a->l) * eval(a->r);
                    break;
        case '/': v = eval(a->l) / eval(a->r); break;
        case '|': v = eval(a->l); if(v < 0) v = -v;
                    break;
        case 'M': v = -eval(a->l);
                    break;
        case 'F': v = callbuiltin((struct fncall *)a);
                    break;
        default: printf("internal error: bad node %c\n", a->nodetype);
    }
    return v;
}

```

Figura 40: Operaciones aritméticas

- **Función de liberación:** se liberan los nodos de los árboles para no consumir memoria

```
void treefree(struct ast *a) {
    switch(a->nodetype) {
        /* two subtrees */ case '+':
        case '-':
        case '*':
        case '/':
            treefree(a->r);
        /* one subtree */
        case '|':
        case 'M':
            treefree(a->l);

        /* no subtree */
        case 'K':
            free(a);
            break;
        default: printf("internal error: free bad node %c\n", a->nodetype);
    }
}
```

Figura 41: Función de liberación

Hito 3

Flex

En este hito, el archivo de flex no se modificó con respecto al anterior hito.

Bison

Definición

Se han añadido librerías y variables nuevas, quedando de la siguiente manera:

```
%{
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "syntab.h"
#include "AST.h"

extern FILE *yyin;
extern FILE *yyout;

int globalError =0;
int globalContadorNum =0;
int globalContadorOper =0;
int globalBoolCond = 0;
char globalSignCond;
char *globalTipo;
```

Figura 42: Nuevas librerías y variables

También se incorporó precedencia de operadores, añadiendo las siguientes líneas:

```
%left MAS MENOS
%left POR DIV
```

Se utilizó “%left” para añadir asociatividad por la izquierda, el último declarado es el que tiene más precedencia. Que en este caso fueron la multiplicación y la división.

Con respecto al %union creado en el anterior hito se añadió el struct fncall, para generar el árbol para las funciones.

```
%union {
    int number;
    double numberf;
    char *string;
    struct {
        char *tipo;
        int valor;
        double valord;
        char *texto;
        int booleanCond;
        struct ast *a;
        struct flow *f;
        struct fncall *fun;
    } snum;
}
```

Figura 43: Struct fncall

Reglas

En este hito se ha centrado en la generación de código a través de los árboles de sintaxis abstracta (ASA). Se generó código ejecutable en MIPS para las operaciones aritméticas, bucles while y sentencias if.

En primer lugar, se definió todas las operaciones para que se guarden en árboles. Estos árboles posteriormente serán la pasarela para generar un archivo en formato MIPS.
 Para ello se han generado contadores de variables, sentencias, etc.
 Para guardar un contador de la cantidad de operadores en una operación aritmética se cuentan en “globalContadorOper”

```

calc: calc MAS calc {
    printf("Suma\n");
    globalContadorOper = globalContadorOper + 1;

    if (globalError ==0) { //No hay errores
        if (($1.tipo == "entero") && ($3.tipo == "entero")) {
            $$ .a = newast('+', $1.a,$3.a);
            evalprint($$.a);
            globalTipo = $1.tipo;
            contadorOperadores(globalContadorOper, $$ .a);
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($1.tipo == "real") && ($3.tipo == "real")){
            $$ .a = newast('+', $1.a,$3.a);
            evalprint($$.a);
            globalTipo = $1.tipo;
            contadorOperadores(globalContadorOper, $$ .a);
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($1.tipo == "string")){
            $$ .a = newast('+', $1.a,$3.a);
            evalprint($$.a);
            globalTipo = $3.tipo;
            contadorOperadores(globalContadorOper, $$ .a);
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($3.tipo == "string")){
            $$ .a = newast('+', $1.a,$3.a);
            evalprint($$.a);
            globalTipo = $1.tipo;
            contadorOperadores(globalContadorOper, $$ .a);
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
        else if (($3.tipo == "string") && ($1.tipo == "string")){
            $$ .a = newast('+', $1.a,$3.a);
            evalprint($$.a);
            globalTipo = $1.tipo;
            contadorOperadores(globalContadorOper, $$ .a);
            //printf("Suma (tipo=%s)\n", $$ .tipo);
        }
    } else {
        printf("HAY ERRORES NO SE REALIZA LA SUMA\n");
    }
}

```

Figura 44: Pasarela MIPS

Por otro lado, para poder contabilizar las variables numéricas se usa el contador “globalContadorNum”

```

| IDENTIFICADOR {
    printf("variable\n");
    $$ .tipo = "string";

    if (getvalsymText($1) == 0 ) {
        globalError = 1;
        printf("ERROR - Variable no existe: %s \n", $1);

    } else {
        globalError = 0;
        $$ .valor = getvalsymNum($1);
        $$ .a= newnum($$ .valor);
        globalContadorNum = globalContadorNum +1;
        contadorNumeros(globalContadorNum, $$ .valor );

        printf("CORRECTO - Variable si existe: %s \n", $1);
    }
}

```

Figura 45: GlobalContadorNum

Una vez se ha identificado la operación aritmética se procede a generar código ejecutable en MIPS. En la figura 46 se muestran las llamadas a las clases encargadas de esto, que se encuentran en la clase “AST.h”. Como argumento se le pasa el árbol que se ha creado al realizar la identificación de los números y operadores

```
expr: IDENTIFICADOR DOSPUNTOS_IGUAL calc {
    if (globalBoolCond == 0) {
        globalBoolCond =0;
        fprintf(yyout, ".....\n");
        fprintf(yyout, ".data\n");
        dataOper($3.a);
        fprintf(yyout, ".text\n");
        textOper($3.a);
        fprintf(yyout, ".....\n");
        globalContadorNum =0;
        globalContadorOper =0;
        globalBoolCond =0;

    }

}
```

Figura 46: Identificador

Se realizó el mismo proceso para la generación de código de las sentencias if. En la figura 47 se muestran las llamadas a las clases encargadas de esto, que se encuentran en la clase “AST.h”. Como argumento se le pasa el árbol que se ha creado al realizar la condición y las sentencias que se encuentran dentro del if. En este caso el cuarto parámetro de “newflow”, es NULL porque no tiene apartado else.

```
sentencia_if: IF calc THEN SALTOLINEA sentencia SALTOLINEA ENDIF {
    $$ .f = newflow('I', $2.f , $5.f, NULL);
    fprintf(yyout, ".....\n");
    fprintf(yyout, ".data\n");
    dataOper($2.a);
    fprintf(yyout, ".text\n");
    textIf(globalSignCond,$5.f);
    fprintf(yyout, ".....\n");
    globalContadorNum =0;
    globalContadorOper =0;
    globalBoolCond =0;

}
```

Figura 47: Comprobación if

Así mismo, se continuó con la generación de código para los bucles while. En la figura 48 se muestran las llamadas a las clases encargadas de esto, que se encuentran en la clase “AST.h”. Como argumento se le pasa el árbol que se ha creado al realizar la condición y las sentencias que se encuentran dentro del while. En este caso el cuarto parámetro de “newflow”, es NULL porque no tiene apartado else.

```

bucle_while: WHILE calc LOOP SALTOLINEA sentencia SALTOLINEA ENDLOOP {
    $$._f = newflow('W', $2._f, $5._f, NULL);
    fprintf(yyout, ".....\n");
    fprintf(yyout, ".data\n");
    dataOper($2._f);
    fprintf(yyout, ".text\n");
    textWhile(globalSignCond, $$._f);
    globalContadorNum =0;
    globalContadorOper =0;
    globalBoolCond =0;
    fprintf(yyout, ".....\n");
}
;

```

Figura 48: Bucle while

Por otro lado, como extra se implementó la generación de los árboles para las funciones. En la figura 49 se muestra la función “newfunc” que se encuentra en el archivo “AST.h”, donde se realiza la creación del árbol.

```

funcion: FUNCION nombreFuncion PAR_I declaraciones PAR_D RETURN tipo IS SALTOLINEA BEGINN SALTOLINEA sentencia
SALTOLINEA Fun SALTOLINEA END {
    $$._fun = newfunc($4._fun, $11._fun);
}
;

```

Figura 49: Extra generación de árboles

Subrutinas de usuario

En esta sección no se realizó ninguna modificación con respecto al hito 2, se continuó realizando de la misma manera la lectura y la escritura de archivos.

SymTab.h

En este hito, el archivo de symtab.h no se modificó con respecto al anterior hito.

AST.h

En este hito, a este archivo se le han añadido todo lo relacionado con los bucles WHILE, con la sentencia IF y con las funciones.

Estructuras de los árboles

- En el árbol FLOW dependiendo del tipo de nodo que se identifique hará un IF o un WHILE. En caso de que hiciera un WHILE o el IF sin ELSE como cuarto parámetro, se pondrá un NULL.

```

struct flow {
    int nodetype; /* tipo I o W*/
    struct ast *cond; /* condición */
    struct ast *tl; /* la rama o la lista */
    struct ast *el; /* opcional else rama */
};

```

Figura 50: Estructura de los árboles

— Este es árbol que se utiliza para las funciones:

```
struct fncall {
    int nodetype;
    struct ast *l;
    enum bifs functype;
};
```

Figura 51: Árbol para funciones

Después se inicializan dichos árboles:

```
struct ast *newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *tr);
struct ast *newfunc(int functype, struct ast *l);
```

Figura 52: Inicialización de árboles

Se inicializa la función que determina el tipo de función que hay.

```
static double callbuiltin(struct fncall *);
```

Figura 53: Inicialización de la función

Se crea la funcionalidad para la detección del tipo de función. Los diferentes tipos de funciones vienen definidas en la lista “bifs”.

```
static double callbuiltin(struct fncall *f) {
    enum bifs functype = f->functype; double v = eval(f->l);
    switch(functype) {
        case B_sqrt:
            return sqrt(v);
        case B_exp:
            return exp(v);
        case B_log:
            return log(v);
        case B_print:
            printf("= %4.4g\n", v);
            return v;
        default:
            yyerror("Unknown built-in function %d", functype);
            return 0.0;
    }
}

enum bifs {
    B_sqrt = 1,
    B_exp,
    B_log,
    B_print
};
```

Figura 54: Detección de tipos de función

Finalmente se crea la funcionalidad de ambos árboles:

- **Funcionalidad árbol FLOW:** esta función crea tanto el árbol de sentencias IF como los bucles WHILE.

```
struct ast * newflow(int nodetype, struct ast *cond, struct ast *tl, struct ast *el) {
    //printf("FLOW");
    struct flow *a = malloc(sizeof(struct flow));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->cond = cond;
    a->tl = tl;
    a->el = el;

    //fprintf(yyout, "cond: %d\n", a->cond);
    //fprintf(yyout, "tl: %lf\n", a->tl);
    //fprintf(yyout, "el: %lf\n", a->el);

    return (struct ast *)a;
}
```

Figura 55: Árbol flow

- **Funcionalidad del árbol de funciones:** esta función crea el árbol para las funciones reconocidas en Ada.

```
struct ast * newfunc(int functype, struct ast *l) {
    struct fncall *a = malloc(sizeof(struct fncall));
    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'F';
    a->l = l;
    a->functype = functype;
    return (struct ast *)a;
}
```

Figura 56: Árbol de funciones

Para la implementación de instrucciones en MIPS se han definido varias funciones:

- **void dataOper (struct ast *a):** crea las variables necesarias en lenguaje MIPS, en el apartado “.data” .
- **void textOper (struct ast *a):** genera las instrucciones de las operaciones aritméticas que hay en el archivo ADA, pero en lenguaje MIPS.
- **void textIf (char signo, struct ast *f):** genera las instrucciones para crear y realizar una sentencia IF que hay en el archivo ADA, pero en lenguaje MIPS.
- **void textWhile(char signo, struct ast *f):** genera las instrucciones para crear y realizar el bucle WHILE que hay en el archivo ADA, pero en lenguaje MIPS.
- **void contadorNumeros (int contador, int val):** contabiliza el número de operaciones, cuando el contador es 0, la siguiente operación es una nueva.
- **void contadorOperadores (int contador, struct ast *a):** hace exactamente lo mismo pero para el contenido de los árboles.

MIPS

Se conoce con el nombre de MIPS (Microprocessor without Interlocked Pipeline Stages) se conoce a toda una familia de microprocesadores desarrollados por MIPS Technologies.

Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI, una empresa fabricante de ordenadores, tanto hardware como software. Se usa en muchos sistemas embebidos; en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable. Más recientemente, la NASA usó uno de ellos en la sonda New Horizons 1. Las primeras arquitecturas MIPS fueron fsfs en 32 bits (generalmente rutas de datos y registros de 32 bits de ancho), si bien versiones posteriores fueron implementadas en 64 bits. Existen cinco revisiones compatibles hacia atrás del conjunto de instrucciones del MIPS, llamadas MIPS I, MIPS II, MIPS III, MIPS IV y MIPS 32/64.

En la última de ellas, la MIPS 32/64 Release 2, se define a mayores un conjunto de control de registros. Así mismo están disponibles varias "extensiones", tales como la MIPS-3D, consistente en un simple conjunto de instrucciones SIMD en coma flotante dedicadas a tareas 3D comunes, la MDMX(MaDMAx) compuesta por un conjunto más extenso de instrucciones SIMD enteras que utilizan los registros de coma flotante de 64 bits, la MIPS16 que añade compresión al flujo de instrucciones para hacer que los programas ocupen menos espacio (presuntamente como respuesta a la tecnología de compresión Thumb de la arquitectura ARM) o la reciente MIPS MT que añade funcionalidades multithreading similares a la tecnología HyperThreading de los procesadores Intel Pentium 4.

Debido a que los diseñadores crearon un conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades y escuelas técnicas a menudo se basan en la arquitectura MIPS. El diseño de la familia de CPU's MIPS influiría de manera importante en otras arquitecturas RISC posteriores como los DEC Alpha.

Estructura de MIPS

MIPS consta de dos apartados:

- .data: lugar donde se declaran las variables.
- .text: lugar donde se realizan las instrucciones necesarias utilizando las variables anteriores.

Operaciones aritméticas

MIPS es una máquina de arquitectura carga-almacenamiento: para usar un dato almacenado en memoria, primero hay que pasarlo a un registro.

Las operaciones aritméticas básicas en MIPS se caracterizan por:

- Utilizar tres registros (2 para los operandos y 1 para el resultado).
- Sintaxis: operación resultado op1 op2.
- El último de los operandos puede ser una constante de 16 bits (“inmediato”).

En la figura 57 se muestra la operación aritmética que se va a pasar a código MIPS.

Marcos:= 50 * 2 - 10;

Figura 57: Ejemplo de operación aritmética

Y en la figura 58 se muestra el resultado que se ha generado al ejecutar el programa, se puede observar cómo se han declarado correctamente las tres variables y realizado primero la operación de multiplicación y después la resta. Para ver que funciona correctamente también se ha incluido código para imprimir el resultado.

```
.data
number3: .word 50
number2: .word 2
number1: .word 10
message: .asciiz "FIN"
.text
lw $t0, number3($zero)
lw $t1, number2($zero)
lw $t2, number1($zero)
mul $t3, $t1, $t0
sub $t5, $t3, $t2
li $v0, 1
add $a0, $zero, $t5
syscall
```

Figura 58: Código comprobaciones

En este código tenemos declaradas en memoria 3 variables.

- **.word:** inicializa una zona en memoria para Integers.
- **.asciiz:** inicializa una zona de memoria con STRING y un carácter NULL al final.

Las instrucciones que se realizan son:

- **lw:** load word, carga una palabra (32 bits) con extensión de signo. En el registro \$t0 se guarda en memoria el contenido de la variable “number3”.
- **mul:** multiplica el contenido de los registros \$t1 y \$t0, guardando el resultado en \$t3.
- **sub:** resta el contenido de los registros \$t3 y \$t2, guardando el resultado en \$t5.
- **li:** carga en el registro \$v0 un número entero (esto es lo que significa el 1)
- **add:** añade al registro argumento \$a0, la suma de \$zero más el contenido en el registro \$t5.
- **syscall:** coge el contenido en el argumento \$a0 y lo imprime.

En la figura 59 se puede ver como ejecutando el siguiente código funciona correctamente y devuelve el valor 90.

```

1 .data
2 number3: .word 50
3 number2: .word 2
4 number1: .word 10
5 message: .asciz "FIN"
6 .text
7 lw $t0, number3($zero)
8 lw $t1, number2($zero)
9 lw $t2, number1($zero)
10 mul $t3, $t1, $t2
11 sub $t5, $t3, $t0
12 li $v0, 1
13 add $t0, $t0, $t5
14 syscall

```

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000001
\$v1	3	0x00000000
\$a0	4	0x0000005a
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000032
\$t1	9	0x00000002
\$t2	10	0x0000000a
\$t3	11	0x00000064
\$t4	12	0x00000000
\$t5	13	0x0000005a
\$t6	14	0x00000000
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000000
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x00400038

Figura 59: test de código

Sentencia If

Lo que distingue a un computador de una calculadora simple es la habilidad de tomar decisiones. Basándose en los datos de entrada y los valores creados durante la computación, el computador u ordenador ejecuta diferentes instrucciones. La toma de decisiones se representa comúnmente en los lenguajes de programación usando la sentencia if, combinada a veces con sentencias go to

En la figura 60 se muestra la sentencia if que se va a pasar a código MIPS.

```

if 1 < 10 then
    Marcos := 50 * 20;
    Marcos := 5 - 2;
end if;

```

Figura 60: Sentencia if preparada para MIPS

Y en la figura 61 se muestra el resultado que se ha generado al ejecutar el programa, se puede observar cómo se han declarado correctamente todas las variables. Si se cumple la condición se ejecutará la función “operation”, donde en este caso se encuentran la multiplicación y la resta. Para ver que funciona correctamente también se ha incluido código para imprimir el resultado.

```

.data
number6: .word 1
number5: .word 10
number4: .word 50
number3: .word 20
number2: .word 5
number1: .word 2
message: .asciiiz "FIN"
.text
addi $t0, $zero, 1
addi $t1, $zero, 10
blt $t0, $t1, operation
li $v0, 10
syscall
operation:
lw $t2, number1($zero)
lw $t3, number2($zero)
lw $t4, number3($zero)
lw $t5, number4($zero)
mul $t6, $t5, $t4
sub $t8, $t3, $t2
li $v0, 1
add $a0, $zero, $t8
syscall
add $a0, $zero, $t6
syscall
li $v0, 10
syscall

```

Figura 61: Resultado generado

addi: almacena en el registro \$t0 la suma de \$zero con la constante 1

blt: establece un código de condición si se cumpla la condición “menor que”. Si lo almacenado en el registro \$t0 es menor que en \$t1, entonces se va a la zona OPERATION. Si lo almacenado en el registro \$t0 es mayor o igual que en \$t1, entonces continúa con la siguiente línea.

En la figura 62 se puede ver como ejecutando el siguiente código funciona correctamente y devuelve el valor 1000 y el valor 3.

The screenshot shows the MARS 4.5 assembly debugger. The code window displays the assembly source code for 'prueba4.asm'. The registers window shows the state of all 32 general-purpose registers (\$zero to \$pc) and three coprocessor registers (\$fpr, \$sp, \$gp). The messages window shows the output '31000 -- program is finished running --'.

```

1 .data
2 number6: .word 1
3 number5: .word 10
4 number4: .word 50
5 number3: .word 20
6 number2: .word 5
7 number1: .word 2
8 message: .asciz "FIN"
9 .text
10 addi $t0, $zero, 1
11 addi $t1, $zero, 10
12 blt $t0, $t1, operation
13 li $v0, 10
14 syscall
15 operation:
16 lw $t2, number1($zero)
17 lw $t3, number2($zero)
18 lw $t4, number3($zero)
19 lw $t5, number4($zero)
20 mul $t6, $t5, $t4
21 sub $t8, $t3, $t2
22 li $t1, 1
23 add $a0, $zero, $t8
24 syscall
25 add $a0, $zero, $t6

```

Line: 28 Column: 8 Show Line Numbers

Mars Messages Run I/O

Clear 31000 -- program is finished running --

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x000003e8
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000001
\$t1	9	0x00000000
\$t2	10	0x00000002
\$t3	11	0x00000000
\$t4	12	0x00000014
\$t5	13	0x00000032
\$t6	14	0x000003e8
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000003
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00000000
pc		0x0040006c

Figura 62: Ventana de código

Bucle While

Los bucles MIPS son similares a los bucles en otros lenguajes de programación como c ++ o java. Este bucle se usa en una condición en la que no sabemos el número de iteraciones. Puede ser un caso en el que un usuario ingresa valores repetidamente, y el ciclo no termina a menos que el usuario ingrese un valor particular. La estructura básica de el ciclo while que hemos aplicado se expone a continuación:

En la figura 63 se muestra el bucle while que se va a pasar a código MIPS.

```

while 1 < 10 loop
    Marcos := 2 * 3;
    Paula := Marcos + 1;
end loop;

```

Figura 63: Bucle while

Y en la figura 64 se muestra el resultado que se ha generado al ejecutar el programa, se puede observar cómo se han declarado correctamente todas las variables. Si se cumple la condición se ejecutará la función “operation”, donde en este caso se encuentran la multiplicación y la suma. Esto se ejecutará hasta que la condición no se cumpla, que en este caso son 10 veces. Para ver que funciona correctamente también se ha incluido código para imprimir el resultado.

```

.data
number6: .word 1
number5: .word 10
number4: .word 2
number3: .word 3
number2: .word 6
number1: .word 1
message: .asciiz "FIN"
.text
addi $t0, $zero, 1
addi $t1, $zero, 10
while:
bgt $t0, $t1, exit
jal operation
addi $t0, $t0, 1
j while
exit:
li $v0, 4
la $a0, message
syscall
li $v0, 10
syscall
operation:
lw $t2, number1($zero)
lw $t3, number2($zero)
lw $t4, number3($zero)
lw $t5, number4($zero)
mul $t6, $t5, $t4
add $t8, $t3, $t2
li $v0, 1
add $a0, $zero, $t8
syscall
add $a0, $zero, $t6
syscall
jr $ra
syscall

```

Figura 64: Resultado

Se crea una zona WHILE:

- **bgt**: establece un código de condición si se cumpla la condición “mayor que”. Si lo almacenado en el registro \$t0 es mayor que en \$t1, se dirige a la zona EXIT. Sí lo almacenado en el registro \$t0 es mayor o igual que en \$t1, entonces continúa con la siguiente línea.
- **jal**: almacena la dirección de retorno en un registro. Salta a la zona OPERATION.
- **j**: salta, pero a una zona de memoria restringida. Salta a la zona WHILE.
- **la**: carga en el registro argumento \$a0 el contenido de la variable “message”.
- **jr**: realiza un “return”.

En la figura 65 se puede ver como ejecutando el siguiente código funciona correctamente y devuelve el valor 7 y el valor 6, 10 veces.

The screenshot shows the MARS 4.5 assembly debugger. The main window displays assembly code for a program named 'prueba4.asm'. The code includes various instructions like bgt, addi, lw, and syscall, along with labels for loops and operations. The Registers window on the right lists 32-bit registers (\$zero to \$pc) with their current values, mostly initialized to zero or specific memory addresses. The status bar at the bottom indicates the program has finished running.

Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x10010015
\$a1	5	0x00000000
\$a2	6	0x00000000
\$a3	7	0x00000000
\$t0	8	0x00000000
\$t1	9	0x00000000
\$t2	10	0x00000001
\$t3	11	0x00000000
\$t4	12	0x00000003
\$t5	13	0x00000002
\$t6	14	0x00000005
\$t7	15	0x00000000
\$s0	16	0x00000000
\$s1	17	0x00000000
\$s2	18	0x00000000
\$s3	19	0x00000000
\$s4	20	0x00000000
\$s5	21	0x00000000
\$s6	22	0x00000000
\$s7	23	0x00000000
\$t8	24	0x00000007
\$t9	25	0x00000000
\$k0	26	0x00000000
\$k1	27	0x00000000
\$gp	28	0x10000000
\$sp	29	0x7ffffeffc
\$fp	30	0x00000000
\$ra	31	0x00400014
pc		0x00400034

Figura 65: Ventana de código

Manual de usuario

Para poder ejecutar todo este proceso hay que realizar varias acciones.

```
bison -d -v prueba.y
```

```
flex -o prueba.lex.c prueba.flex
```

```
gcc -o analiza prueba.tab.c prueba.lex.c
```

El fichero generado, analiza.exe, se va a utilizar para ejecutar el código ADA del archivo AdaP.adb. Para ello le pasaremos por argumento al ejecutable analiza.exe el programa:

```
analiza.exe AdaP.adb
```

El fichero de salida que se genera después de esta instrucción contendrá un código en MIPS, el cual nos servirá para llevarlo al programa MARS y de esta manera compilarlo y obtener el resultado correcto. En este fichero se observan varios códigos que se dividen por “.....”, por lo que habrá que seleccionar manualmente un código y pegarlo en el programa MARS para poder realizar lo que viene a continuación.

Para compilar en el programa MARS, primero tenemos que importar este archivo generado. En nuestro caso, se pegará el código y se guardará el archivo.

Name	Number	Value
\$zero	0	0
\$t0	1	0
\$v0	2	0
\$t1	3	0
\$t2	4	0
\$t3	5	0
\$t4	6	0
\$t5	7	0
\$t6	8	0
\$t7	9	0
\$t8	10	0
\$t9	11	0
\$t10	12	0
\$t11	13	0
\$t12	14	0
\$t13	15	0
\$t14	16	0
\$t15	17	0
\$t16	18	0
\$t17	19	0
\$t18	20	0
\$t19	21	0
\$t20	22	0
\$t21	23	0
\$t22	24	0
\$t23	25	0
\$t24	26	0
\$t25	27	0
\$gp	28	209469324
\$sp	29	214749549
\$t26	30	0
\$t27	31	4194306
\$t28	32	0
\$t29	33	0

Figura 66: Ventana de código

Después tenemos que compilar dicho archivo para comprobar que no hay fallos.
“Run→ Assemble (F3)”

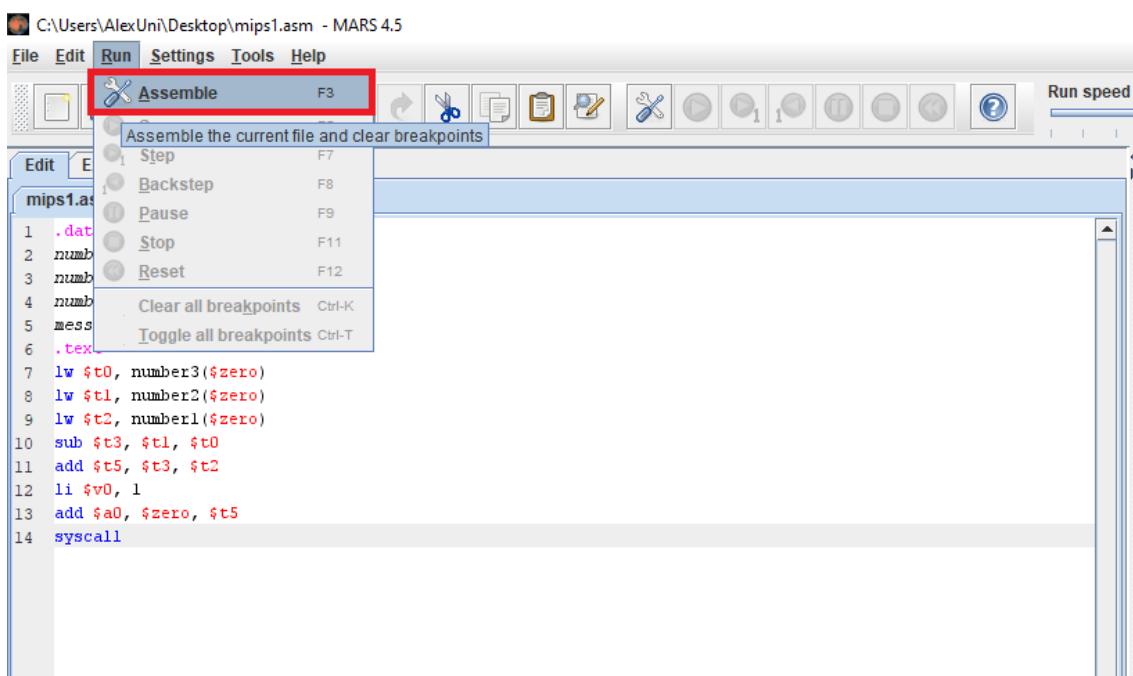


Figura 67: Ventana de herramientas y mensajes de código

Sí la compilación ha ido correctamente obtendremos el siguiente mensaje. En esta pantalla también podemos observar como se ha guardado en memoria algunos datos.

The screenshot shows the MARS 4.5 assembly editor interface. The top menu bar includes File, Edit, Run, Settings, Tools, and Help. A toolbar below the menu contains various icons for file operations and simulation. The main window has several panes:

- Text Segment:** Shows assembly code for the mips1.asm file. The code includes instructions like `lw` and `add` to manipulate registers \$t0 through \$t5.
- Data Segment:** Displays memory starting at address 265501092, with values for each byte.
- Registers:** A table showing the current values of the \$zero through \$t5 registers.
- Mars Messages:** A pane showing the output of the assembly process, including "Assembler operation completed successfully."

Figura 68: Ventana de código

Finalmente ejecutamos el programa tras la compilación y obtendremos el resultado final “Run→ Go (F5)”

The screenshot shows the MARS 4.5 assembly editor after running the program. The interface is similar to Figure 68, but the Registers pane highlights the \$t0 and \$t5 registers, both of which now have the value -3. The Mars Messages pane shows the output "Reset: reset completed." and "- program is finished running (dropped off bottom) --".

Figura 69: Ventana de código

Podemos observar que el resultado es “-3”, y se obtiene de lo guardado en el registros \$t0.

Problemas encontrados

A lo largo de la realización de este trabajo se encontraron numerosos problemas por el camino, pero con esfuerzo y dedicación, se consiguieron resolver. En este punto se nombrarán alguno de ellos.

Uno de los primeros problemas fue la parte de recursividad, se empleó mucho tiempo para conseguir identificar más de una sentencia en cada regla. Por ejemplo, para la parte de declaraciones, en un programa en Ada es probable que haya más de una. Se consiguió llamando recursivamente a esa misma regla.

Por otro lado, también se tuvo problemas con la creación de árboles para poder recorrerlos ya que al ser ascendente, es decir, realiza la identificación de arriba hacia abajo se complicó la lectura. Porque se debía tener claro en qué punto de análisis se debía generar los árboles y su posterior lectura.

En la parte de generación de código para MIPS, la mayor dificultad es que trabaja con un número limitado de variables y es necesario liberarlas para otra vez utilizarlas. Esto complicó a medida que se fueron aumentando el número de variables, ya que era necesario liberarlas para su reutilización. Se ha realizado en un único archivo con varias instrucciones, en vez de un archivo por instrucciones. Esto quiere decir que en el archivo de salida en vez de aparece el código ADA entero, aparece dividido según operaciones aritméticas, bucles, etc...

Pero sin lugar a duda la mayor dificultad de este trabajo se ha encontrado en la generación de código, ya que se debía realizar todo el lenguaje C. Este lenguaje únicamente se ha tenido en consideración en una asignatura al inicio de la carrera y por lo tanto ha sido empezar de cero.

Batería de pruebas

Para la realización de este trabajo se ha tenido en consideración todos los posibles fallos que se pueden encontrar en un programa escrito en Ada mal escrito. Especialmente se ha tenido en cuenta las comprobaciones indicadas en el enunciado.

Tests con programas correctos

En la figura 70 se muestra un ejemplo de Ada correcto.

```

procedure Hello is
    Marcos :Integer;
    Paula :Integer;
    Maria :Integer;
    Jose :Integer;
    Rosa :Integer;
    Alberto :Integer;
    Numero :Integer;

begin
    while 1 < 10 loop
        Marcos := 2 * 3;
        Paula := Marcos + 1;
    end loop;

    Maria:= 10 - 2 + 5;
    Jose:= Maria * 2;
    Rosa:= 500 / 2;
    Alberto:= Rosa + Jose + 5;
    Marcos:= 50 * 2 - 10;

    if 1 < 10 then
        Marcos := 50 * 20;
        Marcos := 5 - 2;
    end if;

    function Funcion1 (Edad :Integer) return Integer is
    begin
        Put_Line("Mayor");
        Rosa := 200 - 5;
        return Edad;
    end
end Hello

```

Figura 70: ADA

Y en la figura71, su solución, vemos como todas las comprobaciones aparecen como correcto.

```

-- -----
[jennygonzalez@MacBook-Pro-de-Jenny bison_hito3.1 % ./analiza /Users/jennygonzalez/Docu-
ments/bison_hito3.1/AdaP.adb
CORRECTO - Es menor
CORRECTO - Se ha declarado la variable: Marcos
RESULTADO - De Marcos: 6
CORRECTO - Variable si existe: Marcos
CORRECTO - Se ha declarado la variable: Paula
RESULTADO - De Paula: 7
CORRECTO - Se ha declarado la variable: Maria
RESULTADO - De Maria: 13
CORRECTO - Variable si existe: Maria
CORRECTO - Se ha declarado la variable: Jose
RESULTADO - De Jose: 26
CORRECTO - Se ha declarado la variable: Rosa
RESULTADO - De Rosa: 250
CORRECTO - Variable si existe: Rosa
CORRECTO - Variable si existe: Jose
CORRECTO - Se ha declarado la variable: Alberto
RESULTADO - De Alberto: 281
CORRECTO - Se ha declarado la variable: Marcos
RESULTADO - De Marcos: 90
CORRECTO - Es menor
CORRECTO - Se ha declarado la variable: Marcos
RESULTADO - De Marcos: 1000
CORRECTO - Se ha declarado la variable: Marcos
RESULTADO - De Marcos: 3
CORRECTO - No existe ninguna función con ese nombre
CORRECTO - Se ha declarado la variable: Rosa
RESULTADO - De Rosa: 195
Tiene al menos un return
CORRECTO - Coincide el tipo de return
CORRECTO - Coincidien los nombres
TODO OK

```

Figura 71: Solución ADA

En este ejemplo se encuentra un programa ADA correcto, por lo tanto se identifica correctamente todas las sentencias.

Tests con programas con errores

En la figura 72 se muestra un ejemplo de Ada incorrecto.

```

procedure Hello is
    Marcos :Integer;
    Paula :Integer;
    Rosa :Integer;
    Alberto :Integer;
    Numero :Integer;
    Letra :String;

begin
    while 1 < 10 loop
        Marcos := 2 * 3;
        Paula := Luis + 1;
    end loop;

    Paco:= 10 - 2 + 5;
    Rosa:= 500 / 0;

    if 1 > 10 then
        Marcos := 50 * 20;
    end if;

    function Funcion1 (Edad :Integer) return Integer is
    begin
        Put_Line("Mayor");
        Rosa := 200 - 5;
        return Letra;
    end

    function Funcion1 (Edad :Integer) return Integer is
    begin
        Put_Line("Mayor");
        return Letra;
    end
end Adios

```

Figura 72: Ejemplo ADA incorrecto

Y en la figura 73, su solución, vemos como algunas de estas comprobaciones aparecen con errores.

```

[jennygonzalez@MacBook-Pro-de-Jenny bison_hito3.1 % ./analiza /Users/jennygonzalez/Docum
[ents/bison_hito3.1/AdaP.adb
[CORRECTO - Es menor
[CORRECTO - Se ha declarado la variable: Marcos
[ERROR - Variable no existe: Luis
HAY ERRORES NO SE REALIZA LA SUMA
ERROR - No se ha declarado la variable: Paco
HAY ERRORES NO SE REALIZA LA DIVISION
ERROR - No es mayor
CORRECTO - Se ha declarado la variable: Marcos
CORRECTO - No existe ninguna función con ese nombre
CORRECTO - Se ha declarado la variable: Rosa
Tiene al menos un return
ERROR - No coincide el tipo de return
ERROR - Existe ya una función con ese nombre
Tiene al menos un return
ERROR - No coincide el tipo de return
ERROR - No es igual el nombre del inicio y el del final

```

Figura 73: Solución

En este ejemplo se encuentra un programa en Ada con errores, utilización de variables sin definir, funciones con el mismo nombre... Es decir, todas las comprobaciones que se han explicado en anteriores puntos se muestran que funciona, en la figura 73 se ven en pantalla todos los errores encontrados.

Conclusión

La complejidad de crear un compilador no es únicamente la dificultad en la realización de las sentencias, sino en la cantidad de dependencias que hay que tener entre diferentes archivos, los cuales tienen su propia funcionalidad. Por ejemplo, si en el FLEX hay un fallo, puede que al compilar dicho archivo no se vea el fallo, ya que gramaticalmente está bien formulado, pero luego a la hora de compilar el BISON sí da el fallo, por lo que hay que estar muy atento a las propias dependencias.

En todo momento se tuvo en cuenta que se realizaba el análisis de forma ascendente, es decir, realiza la identificación de arriba hacia abajo.

Esta estructuración es muy abstracta, por lo que plasmarla no ha sido una tarea tan fácil como los esquemas que se han estado observando en las búsquedas de información. La abstracción a la que nos referimos es por ejemplo al programar los árboles o la tabla de símbolos, para luego referenciarlos en el BISON.

Por otra parte, nos ha hecho comprender todos los conceptos aprendidos a lo largo de la asignatura. Por ejemplo, como funciona realmente el análisis léxico, además del beneficio que tiene la utilización de estructuras como la tabla de símbolos, ya que su implantación ha sido primordial para almacenar variables. Al igual que con la generación de árboles de sintaxis abstracta, que sin ellos no hubiese sido posible la generación de código de una manera ordenada.

Como conclusión final, podemos decir que la realización de un compilador desde cero es una tarea bastante compleja y que se necesita emplear una gran cantidad de tiempo. Sobretodo si como objetivo final se quiere obtener un compilador perfecto, preparado para soportar cualquier tipo de errores. Por ello la mejor comprensión de todo por parte de un miembro del equipo ha sido crucial para poder seguir adelante.