Compiladores

Memoria de proyecto final

Curso académico: 2019/20

Autores:

Beatriz Navidad Vilches
beatriz.navidadv@um.es
Grupo 1.4

Francisco Aguilar Martínez
francisco.aguilarm@um.es
Grupo 1.4

Profesor:
Eduardo Martínez Gracia

23 de mayo de 2020

Índice

Índice de imágenes

1. Introducción

Esta práctica forma parte de la trayectoria académica de la PCEO de Ingeniería Informática y Matemáticas de la **Universidad de Murcia**, en particular del **Grado en Ingeniería Informática**, durante el **curso académico 2019/20**.

El **objetivo** de este proyecto es **programar un pequeño compilador** que lea un programa escrito en **"miniPascal"**, una versión simplificada de Pascal, y que genere otro equivalente **en código MIPS**. Este código generado podrá ser ejecutado usando intérpretes del ensamblador de MIPS como *spim* o *Mars*.

A modo de aclaración, el lenguaje **miniPascal** es una versión simplificada de Pascal que cuenta con diversas características presentes en la mayoría de lenguajes de alto nivel que hemos manejado hasta ahora, tales como:

- **Diferenciación entre *constantes*** (no pueden redefinirse) y ***variables***.
- **Asignaciones de valores**.
- **Escritura (`write`)** de "*strings*", expresiones y variables, y lectura (`read`) de variables desde la entrada estándar.
- **Estructuras de control de flujo**
  - de selección: `if-then` e `if-then-else`.
  - iterativas: `for` y `while`.
- **Funciones** que permiten la reutilización de código dentro del programa.[1]

Sin embargo, al ser una versión reducida de Pascal, presenta algunas carencias, como por ejemplo.

- **Solo** hay un **tipo** de datos **(`integer`)**.
- **Limitación de la sentencia *for*** (solo permite incrementar la variable de control).
- **Ausencia de operadores lógicos y relacionales**.

A lo largo de este documento, explicaremos el proceso de implementación de las distintas partes que componen el compilador (*analizador léxico*, *analizador sintáctico*, *analizador semántico* y *generador de código*), describiendo las funciones implementadas, estructuras de datos utilizadas y decisiones de programación tomadas
También se proporcionará un manual de usuario para el correcto uso del programa.

[1] Ampliación sin realizar en esta práctica

2. Funcionalidad obligatoria

En esta sección vamos a describir aspectos importantes relacionados con la implementación de la funcionalidad obligatoria que se especifica en el enunciado de esta práctica.

2.1. Analizador léxico

2.1.1. Errores léxicos

Durante el análisis léxico, debemos identificar todos los tokens del lenguaje, además hemos de detectar algunos fallos de carácter léxico que pueden surgir al escribir el código de miniPascal:

- **Comentarios multilínea sin cerrar**. Para detectar esta situación, hacemos uso de las *condiciones de contexto*. El **token** que marca **el cambio de contexto** es un '(*' y a partir de aquí se permite cualquier tipo de carácter hasta detectar el **token** de **fin de contexto** que es el cierre de comentario, '*)'.
  En el contexto de **comentarios** si detectamos el fin de fichero, **'«EOF»'**, se procedería a imprimir un **error** por salida estándar que indique que **el comentario no está cerrado**, además se incrementaría la variable auxiliar *err_lexicos*.
  Esta variable se utiliza a lo largo de todo el programa y contabiliza los errores léxicos encontrado, se utiliza junto a otras variables para el conteo de errores.

- **Identificador demasiado largo**. Para cada identificador hemos de comprobar que **no excede la longitud de 16 caracteres**. Para ello nos apoyamos en la longitud de *yytext*. En caso de que la exceda, se imprime el respectivo mensaje de error y se aumenta la variable *err_lexicos*.

- **Entero demasiado grande**. No se permiten enteros que superen a 2^{31} en valor absoluto. Para controlar esto comprobamos que *atoll(yytext)*, cumpla la restricción. En caso de que no la cumpla, se imprime el respectivo mensaje de error y se aumenta la variable *err_lexicos*.

- **Comillas sin cerrar**. Este caso es parecido al de los comentarios sin cerrar, pero bastante más sencillo de controlar. Para detectar el error se utiliza una expresión regular que detecta este tipo de situación:

  ```
  \"([^\n"]|\\\")*
  ```

  Cada vez que se encuentra un *match* para esta expresión se procede a imprimir un mensaje de error por pantalla y a aumentar la variable *err_lexicos*.

2.1.2. Recuperación de errores en modo pánico

Los tokens están formados a partir de un conjunto específico de caracteres, el resto son **caracteres no permitidos** que producirán errores léxicos al ser detectados en la entrada.
Para tratar la aparición de estos elementos se utiliza una **expresión regular que acepta repeticiones de 1 o más caracteres del estilo**, de esta manera el escáner léxico agrupará las secuencias de caracteres no permitidos e **imprimirá un solo error**, en vez de imprimir tantos errores como caracteres no permitidos se encuentren[2].

[2]Si se imprimiese un error por cada carácter no permitido que se encontrase en la entrada, podríamos dar lugar a salidas de error excesivamente largas, por ejemplo si pasásemos un código con todos sus caracteres no permitidos.

2.2. Analizador sintáctico

2.2.1. Lista de símbolos

Con el objetivo de almacenar las **variables**, **constantes** y **strings** utilizados en el programa, así como su respectivo valor, hemos utilizado la estructura **listaSimbolos**.
Definida en los ficheros *listaSimbolos.h* y *listaSimbolos.c* y utilizada para la **generación del código de la sección de datos**. Como su nombre indica se trata de una lista simplemente enlazada, en la que cada uno de sus nodos almacena un Simbolo.
Un **Simbolo** es una estructura de datos que tiene un **tipo** (*enum*), un **nombre** (*char**) y **valor** (*int*), y que nos servirá para almacenar indistintamente:

- ***identificadores*** de variable o constante, en cuyo caso **se almacena** el **nombre** del identificador precedido por '_' para evitar posibles colisiones entre los nombres de variables usadas en el fichero de entrada del programa y nombres de operaciones del ensamblador de MIPS, el **valor** de la variable o constante con la que se corresponde y el **tipo**, es decir, si es variable o constante.

- ***strings***. Como no se implementa el tipo *string*, **las cadenas de texto** que aparezcan en el programa **no tendrán** ningún **identificador** con el que relacionarlas, por lo que para poder guardarlas en la lista (y también para facilitar la generación de código) **les daremos un nombre compuesto**[3] por la cadena $*str* seguida de un número que identifique ese *string*. Para ello hemos utilizado un **contador de strings**, tal que comience con el valor 0 y se incremente por cada nuevo *string* encontrado.
  Es **importante** remarcar que **no guardamos un mismo *string* dos veces en la lista**, comprobando para ello antes de insertar si ya hay algún otro nodo con la misma cadena de texto en el campo *nombre* y de tipo *string*.

- **Nombres de funciones.**

- **Argumentos de funciones.**

De estas dos últimas no ofreceremos detalles ya que no hemos hecho la implementación opcional de funciones.
En la siguiente imagen podemos ver un pequeño esquema de la estructura de la lista de símbolos:

[3]Para concatenar la cadena y el número utilizamos una función *contanetaInt* que simplemente concatena la cadena y el entero haciendo uso de *sprintf* y devuelve la nueva cadena.

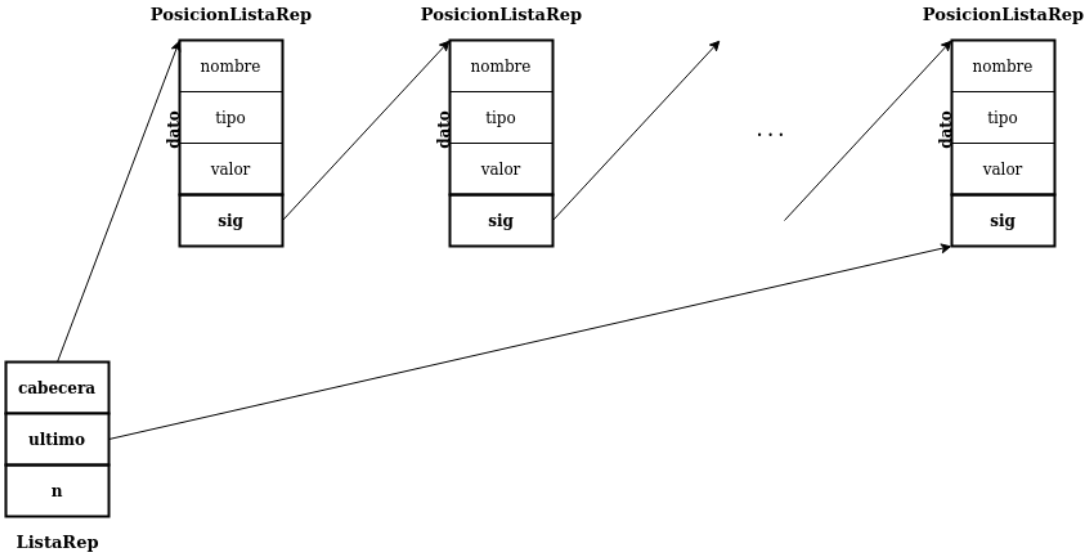

Imagen 1: Lista de símbolos

2.3. Errores sintácticos

El parser de Bison detecta errores sintácticos cuando lee un token que no satisface ninguna regla de sintáctica. Dicho parser reporta los errores llamando a la función **yyerror**, la cual hemos redefinido en nuestro código para que muestre un mensaje con la línea de código que causó el error y el número de errores de cada tipo encontrados hasta ese punto:

```
void yyerror(const char *msg)
{
    printf("Error en la lìnea %d: %s\n", yylineno, msg);
    err_sintacticos++;
    printf("\n%d errores léxicos\n"
           "%d errores sintácticos\n"
           "%d errores semánticos\n"
           "%d errores en la generaciòn "
           "del còdigo\n", err_lexicos, err_sintacticos, err_semanticos
   , err_gen_cod);
}
```

Esta función es llamada por *yyparse* cada vez que se encuentra un error sintáctico y, como no hemos implementado las reglas gramaticales necesarias para realizar una recuperación de errores, ***yyerror* develve un 1** a dicha función y se finaliza la ejecución del compilador sin generar ningún código y sin continuar el análisis semántico.

2.4. Analizador semántico

De forma **simultánea al análisis sintáctico** realizaremos el análisis semántico, detectando errores de este tipo que se puedan haber producido en el código de entrada. Los errores semánticos que se detecten se irán contabilizando en la variable auxiliar *err_semánticos*. Distinguimos dos tipos de errores semánticos:

- **Definición de variables no declaradas**. Cada vez que en una regla (que no forme parte de *definitions*) **aparezca un identificador** comprobaremos que este esté **almacenado en la lista enlazada**. Si no lo encontramos implicará que no ha sido definido, por tanto imprimiremos el mensaje de error correspondiente y aumentaremos la variable *err_sintacticos*.
- **Redefinición de constantes**. Cada vez que encontremos en la sección de *statements* una regla de definición de un identificador (`id := expression`) o una lectura (`read(id)`) comprobaremos que el *id* esté guardado en la lista de símbolos y que sea de tipo *variable*. Si el tipo fuese *constante* se procedería a imprimir el correspondiente mensaje de error y a aumentar la variable *err_sintacticos*.

2.5. Generador de código

Para la **generación de código** hemos empleado un subconjunto de instrucciones del ensamblador de **MIPS**.
Una vez superados los análisis previos, y tras **asegurarnos de que no se hayan detectado errores** léxicos, sintácticos o semánticos[4] en la entrada suministrada por el usuario, podemos pasar a la generación del código MIPS. Dicha generación **se realiza de manera simultánea a los análisis sintáctico y semántico**

El fichero obtenido al compilar la entrada escrita en miniPascal comenzará con la **declaración** de las **variables** o **constantes** enteras que se han encontrado en el fichero de entrada, tras esto, aparecerá una sección de **declaraciones** de las **cadenas de caracteres** que se han utilizado en el programa fuente. El formato de estas será el siguiente

```
#Secciòn de datos
        .data
_a:
        .word 0
_b:
        .word 0
$str1:
        .asciiz "Inicio del programa\n"
$str2:
        .asciiz "a = "
$str3:
        .asciiz "\n"
$str4:
        .asciiz "Final"
```

Después de la sección de datos comenzaremos con la sección de texto que contiene las instrucciones del código ensamblador.
Para la escritura de estas secciones nos apoyaremos principalmente en dos estructuras de datos, ambas suministradas por los docentes de la asignatura:

- **Lista de símbolos.** Ya explicada anteriormente 2.2.1.
- **Lista de código.** Definida en los ficheros *listaCodigo.h* y *listaCodigo.c*. Utilizada para la generación del código de la sección de texto. Se trata de una lista simplemente enlazada en la que **cada uno de sus nodos representa una operación en código ensamblador**.
  Los elementos **terminales generan listas de código**, que van ascendiendo a lo largo de todo el proceso de sintáctico/semántico hasta que finalmente obtenemos una **única lista que contiene todas las operaciones MIPS** que componen nuestro programa.

[4]Se comprueba con una función auxiliar `ok()` que suma los valores de todas las variables de conteo de errores y devuelve el resultado, de manera que devolverá 0 cuando no hay ningún error y un número > 0 cuando sí los haya.

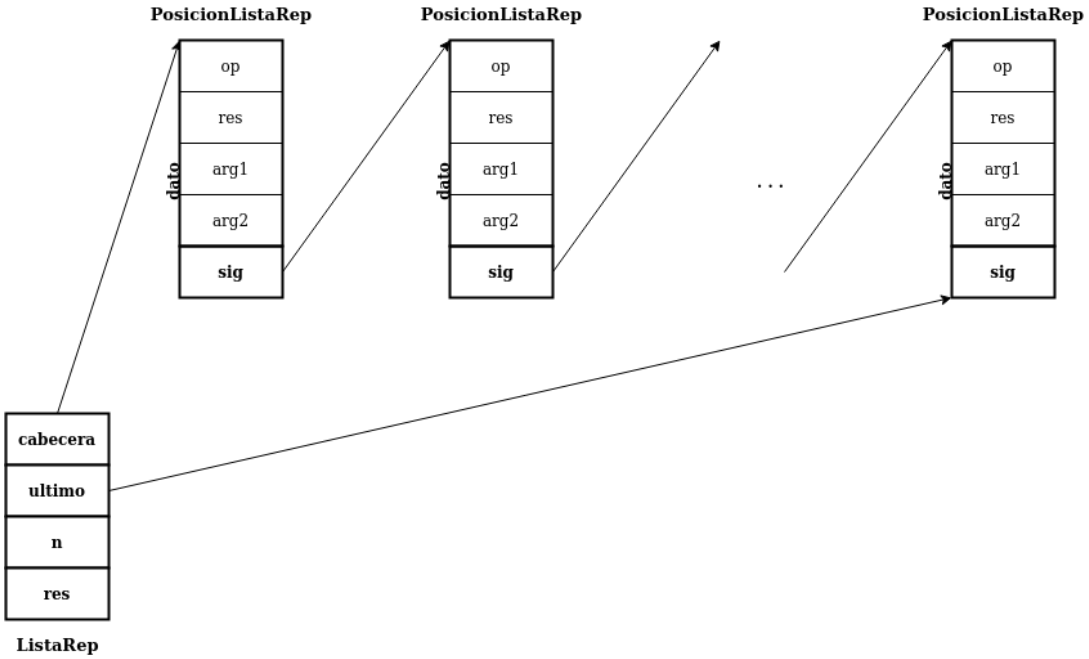

Imagen 2: Lista de código

Un aspecto relevante a la hora de la generación de código ha sido el **manejo de los registros temporales** de los que disponemos, para su correcto uso nos hemos apoyado en:

- **Un array de booleanos** de tamaño 10 que nos permitía saber qué registros se encontraban disponibles.
- **Dos funciones**: *obtenerReg()* y *liberarReg(char* reg)*.

```
char* obtenerReg()
{
    for(int i = 0; i < 10; i++)
    {
        if(temp[i] == 0)
        {
            temp[i] = 1;
            char aux[4];
            sprintf(aux, "$t%d", i);
            return strdup(aux);
        }
    }
    printf("Error: registros agotados\n");
    exit(1);
}

void liberarReg(char* reg)
{
    if((reg[0]=='$') && (reg[1]=='t'))
    {
        int aux = reg[2]-'0';
        assert(aux >= 0);
```

```
            assert(aux < 10);
            temp[aux] = 0;
        }
    }
```

La primera de ellas lleva a cabo una búsqueda del menor registro temporal libre y retorna su nombre, por otro lado, *liberarReg* marca el registro temporal pasado como parámetro como disponible en nuestro array.

Llegamos ahora a la parte principal de la generación de código, las funciones **imprimirLS** e **imprimirLC**.
Tanto la implementación como el objetivo de estas es muy simple. Se trata de **imprimir** (1) las declaraciones de variables, constantes y strings almacenados en la lista de símbolos que se define al iniciar el análisis sintáctico/semántico y (2) las Operaciones de la lista de código, manteniendo en todo caso el formato tanto de instrucciones como de declaraciones y estructura del código de MIPS.

3. Funcionalidad opcional

Como ampliación de la práctica se nos ofrecían varias opciones

- **Recuperación de errores** en el analizador sintáctico.
- Implementación de la sentencia **for**.
- Implementación de **funciones** y **llamadas a funciones**.

De todas estas mejoras, nosotros hemos implementado la sentencia **for**, la cual pasamos a describir.

3.1. Sentencia for

La estructura de la sentencia **for** es la siguiente:

$$for\ id := \textbf{ expression } to \textbf{ expression } do \textbf{ statement}$$

Tras las pertinentes comprobaciones semánticas, comprobamos que no haya ningún error previo y procedemos a la inserción de instrucciones en la lista de código pertinente, para ello hemos ido concatenando las listas de código asociadas a los no terminales que aparecen en la regla de producción del for e insertando otras instrucciones necesarias. La estructura del código sigue el siguiente esquema:

1. **Almacenamos** el valor de **Expression 1** en memoria.[5]
2. **Expression 2**.
3. **Etiqueta** a la comprobación del for.
4. **Carga** de la **variable de control** en un registro temporal.
5. **Comprobación de la condición de salto**[6].
6. **Statement**.
7. **Carga** de la **variable de control** en un registro temporal.
8. **Incremento** de la variable de control .
9. **Almacenamos** el valor de la **variable de control** en memoria[7].
10. **Salto incondional**.
11. **Etiqueta final.**

[5] Además liberamos el registro temporal que almacena el valor de la expresión

[6] En este punto se vuelve a libera el registro temporal que hemos utilizado para almacenar el valor de la *expression 1* previo a la comprobación

[7] Volvemos a liberar el registro temporal

A primera vista puede parecer que la instrucción 7 no es necesaria, pues podríamos no liberar el registro temporal en el que se encuentra almacenado el valor de la varible de control y evitar esta última carga. Pero debemos recordar que nuestro compilador tiene la **limitación de solo disponer de 10 registros temporales**. Si hiciesemos esto, correriamos el riesgo de **sufrir un fallo en tiempo de ejecución** al quedarnos **sin registros temporales libres**.
Aún con esta decisión de diseño, nuestro compilador se encuentra **limitado** por su incapacidad de traducir un código en mini-Pascal que **anide más de 9 bucles for**, ya que el valor de la *expression 2* queda almacenado en un registro temporal a lo largo de toda la sentencia for.

4. Manual de usuario

4.1. Programas de utilidad

4.1.1. Spim

Se puede descargar directamente desde el terminal, usando el comando `sudo apt-get install spim`.
Spim es un emulador que lee y ejecuta programas en lenguaje *ensamblador*. También cuenta con un *debugger*.
Se recomienda la ejecución del código generado por el compilador con esta herramienta.

4.1.2. Mars

Se puede descargar de la siguiente página http://courses.missouristate.edu/kenvollmar/mars/.
Mars es un entorno de desarrollo integrado (IDE) para la programación en el lenguaje ensamblador MIPS.
Esta herramienta es menos recomendada porque no acepta la finalización del programa ensamblador con la instrucción `jr $ra`, sino que hay que cambiar esta por las instrucciones:

```
li $v0, 10
syscall
```

4.2. Operadores y expresiones

4.2.1. Operador de asignación

Una expresión de asignación será de la forma `id := expr`.
Por ejemplo:

```
1  program ejemplo ();
2  var a,b: integer;
3
4  begin
5      a := 2 * 3;
6      b := a + 1
7  end.
```

4.2.2. Operadores aritméticos

Operadores que se utilizan para realizar operaciones matemáticas:

Operador	Descripción	Ejemplo
+	Suma	res := id1 + id2
-	Resta	res := id1 - id2
*	Multiplicación	res := id1 * id2
/	División	res := id1 / id2
-	Cambio de signo	res := -id1

4.2.3. Otros componentes

- **Identificadores**: cualquier secuencia de caracteres del alfabeto en mayúsculas y/o minúsculas, dígitos entre 0 y 9 y '_'. Ejemplos: `a, b, var1, expr1, contador_bucle`.
- **Cadenas**: ``` ``cadena de texto'' ```.
- **Agrupación de sentencias**: `( - expr1 ) * expr2`.
- **Comentarios**: \\`Esto es un comentario`.
- **Comentarios multilínea**:

```
(*
  Esto es un comentario
  que se escribe en varias
  líneas
*)
```

4.3. Sentencias

4.3.1. IF expr THEN statement

Se evalúa la expresión *expr* y, si es distinta de 0, se procesan las lineas de *statement*. Ejemplo de uso:

```
1  program ejemplo ();
2  var a: integer;
3
4  begin
5      if a then
6          write("a")
7  end.
```

4.3.2. IF expr THEN statement1 ELSE statement2

Se evalúa la expresión *expr* y, si es distinta de 0, se procesan las lineas de *statement1*. En caso contrario se procesan las de *statement2*.
Ejemplo de uso:

```
1  program ejemplo ();
2  var a: integer;
3
4  begin
5      if a then
6          write("a", "\n")
7      else
8          write("no a", "\n")
9  end.
```

4.3.3. WHILE expr DO statement

Mientras que la expresión *expr* tenga un valor distinto de 0 se procesan las líneas de *statement*.
Ejemplo de uso:

```
1  program ejemplo ();
2  var a: integer;
3
4  begin
5      a := 2
6      while a do
7      begin
8          write(a);
9          a := a - 1
10     end;
11 end.
```

4.3.4. FOR id := expr1 TO expr2 DO statement

Se establece el valor de la variable *id* igual al valor de *expr1*, y mientras sea menor o igual que el valor de *expr2* se procesan las líneas de *statement*. En cada iteración se incrementa la variable *id*. **Se desaconseja modificar *id* dentro de *statement***.
Ejemplo de uso:

```
1  program ejemplo ();
2  var a: integer;
3  const b := 2;
4
5  begin
6      for a := 0 to b do
7      begin
8          write(a)
9      end
10 end.
```

4.4. Funciones permitidas por el compilador

4.4.1. Funciones de I/O

- **write** para escribir en la salida estándar. Ejemplo de uso:

```
1  program ejemplo ();
2  var a: integer;
3
4  begin
5      a := 2*(3 + 4);
6      write(a)
7  end.
```

- **read** para leer desde la entrada estándar. Ejemplo de uso:

```
1  program ejemplo ();
2  var a,b,c: integer;
3
4  begin
5      read(a,b);
6      b := a + b
7  end.
```

4.5. Definición de datos

La definición de una variable se realiza anteponiendo la palabra reservada *var* al identificador de la variable. Anteponiendo la palabra reservada *const* en vez de *var* se especifica que es una constante.
Solo hay un tipo de dato (enteros) que se especifica con la palabra reservada *integer* y separado del identificador de la variable por dos puntos ':'.
Ejemplo de uso:

```
1  program ejemplo ();
2  var a,b: integer;
3  const c := 1;
4  const d := 2;
5
6  begin
7      a := d - c;
8      b := a + c;
9  end.
```

4.6. Estructura de un programa en *miniPascal*

Ejemplo de un programa de *miniPascal* completo:

```
1  program prueba ();
2  var a,b: integer;
3  const c := 1;
4  const d := 3;
5
6  begin
7      write("Programa de ejemplo\n");
8      write("Inicio\n");
9      for a := 0 to d do
10     begin
11         b := 2 + a;
12         write("Iteraciòn ",a+1, "\n");
13         while b do
14               begin
15                   write("b = ",b,"\n");
16                   b := b - c
17               end
18         write("\n");
19     end;
20
21     if (b) then
22         write("b","\n")
23     else if (a) then
24         write("a y no b\n")
25     else
26         write("No a y no b\n");
27
28     write("Final","\n")
29 end.
```

4.7. Ejecución del compilador

Para poder utilizar el compilador, acceda con el terminal a la carpeta *mp_compiler*, en el interior de esta encontrará un archivo *makefile* que le facilitará la tarea.
Lo primero que debe hacer es ejecutar el comando **make**, tras esto aparecerá en la carpeta un ejecutable con nombre *mini_Pascal*, para utilizarlo tiene dos opciones:

- Ejecutar el comando **./mini_Pascal < input_file >**
- Ejecutar el comando **make run**.

Para esta segunda opción deberá guardar el código a compilar en un fichero con nombre *entrada.txt*, este fichero será el input_file del compilador, la salida se almacenará en un nuevo fichero *salida.s*.
Es importante destacar que cada vez que ejecute el comando **make run** *salida.s* será sobreescrito.

-- Fin del manual --

5. Pruebas de funcionamiento

Respecto a las pruebas de funcionamientos hemos recurrido a los ficheros de prueba que se han puesto a nuestra disposición a lo largo del cuatrimestre. Además de esto hemos realizado algunos test propios, pasamos a mostrar algunos de los más representativos.

(1) **Código miniPascal (izq) y equivalente en Pascal (dcha).**

```
1  program prueba ();
2  var a,b: integer;
3  const c := 1;
4  const d := 4;
5
6  begin
7    write("Programa de prueba\n");
8    write("Inicio\n");
9    for a := 0 to d-c do
10   begin
11      b := 2 + a;
12      write("Iteraciòn ",a+1,"\n");
13      while b do
14          begin
15             write("b = ",b,"\n");
16             b := b - c
17          end;
18          write("\n")
19   end;
20
21   if (b) then
22     write("b","\n")
23   else if (a) then
24     write("a y no b\n")
25   else
26     write("No a y no b\n");
27     write("Final","\n")
28 end.
```

```
program prueba;
var a,b: integer;
const c = 1;
const d = 4;

begin
    writeln('Programa de prueba');
    writeln('Inicio');
    for a := 0 to d-c do
    begin
        b := 2 + a;
        writeln('Iteraciòn ',a+1);
        while b <> 0 do
           begin
              writeln('b = ',b);
              b := b - c
           end;
        writeln('')
    end;

    if (b <> 0) then
        writeln('b')
    else if (a <> 0) then
        writeln('a y no b')
    else
        writeln('No a y no b');
    writeln('Final')
end.
```

Para corroborar la validez de las salidas proporcionadas por nuestro programa hemos utilizado **Spim** para ejecutar el código MIPS y así comparar su salida con la generada por el equivalente en *Pascal* de la entrada

Salida de MIPS y Pascal

```
Programa de prueba
Inicio
Iteraciòn 1
b = 2
b = 1

Iteraciòn 2
b = 3
b = 2
b = 1

Iteraciòn 3
b = 4
b = 3
b = 2
b = 1

Iteraciòn 4
b = 5
b = 4
b = 3
b = 2
b = 1

a y no b
Final
```

(2) **Prueba de errores. Código miniPascal y equivalente en Pascal.**

```
program prueba ();
var a,b: integer;
var avixuela8000000000000000: integer;
const c := 1;
const d := 4515181215151515151515;
begin
    write("Prueba de detecciòn de
errores\n");
    write("Inicio\n");
    for a := 0 to d-c do
    begin
        b := 2 + a;
        c := 2;
        write("Iteraciòn ",a+1, "\n");
        while b do
              begin
                  write("b = ",b,"\n");
                  b := b - c
              end;
        write("\n")
    end;
    if (b) then
        begin
            write("b","\n");
            read(x)
        end
    else if (a) then
        write("a y no b\n)
    else
        write("No a y no b\n");
    write("Final","\n")
end.
```

```
program prueba;
var a,b,x: integer;
var avixuela8000000000000000: integer;
const c = 1;
const d = 4515181215151515151515;
begin
    writeln('Prueba de detecciòn de
    errores');
    writeln('Inicio');
    for a := 0 to d-c do
    begin
        b := 2 + a;
        //c := 2;
        writeln('Iteraciòn ',a+1);
        while b>0 do
              begin
                  writeln('b = ',b);
                  b := b - c;
              end;
        writeln(' ');
    end;
    if (b<>0) then
        begin
            writeln('b');
            read(x)
        end
    else if (a<>0) then
        writeln('a y no ')
    else
        writeln('No a y no b');
    writeln('Final')
end.
```

La salida de errores que ha producido este fichero de entrada es la siguiente:

```
Error en la lìnea 3: el identificador debe contener menos de 16
 caracteres
Error en la lìnea 5: el valor absoluto de cada entero debe ser
menor que pow(2,31)
Error en la lìnea 12: c es una constante
Error en la lìnea 24: identificador x no declarado
Error en la lìnea 27: comillas sin cerrar: "a y no b\n)
Error en la lìnea 28: syntax error, unexpected else

3 errores léxicos
1 errores sintácticos
2 errores semánticos
0 errores en la generaciòn del còdigo
```

6. Tiempo estimado de trabajo

Por un lado, contamos las horas dedicadas al proyecto en las clases de prácticas, lo cual sumaría unas 12h. Por otro lado, de trabajo autónomo hemos dedicado aproximadamente 33h, que se dividen en (1) terminar la programación del programa compilador 15h y (2) 18 h elaboración de la memoria del proyecto (9h cada uno trabajando en paralelo) . En total quedarían unas 45h de trabajo .

7. Conclusiones y valoraciones personales

Para finalizar este documento, nos gustaría reflexionar sobre las impresiones que nos llevamos de este proyecto y sobre posibles cambios de cara a cursos futuros.

En primer lugar, nos gustaría comentar que esta práctica nos ha resultado bastante interesante y nada tediosa de realizar, pues es un campo que nunca antes habíamos explorado y nos ha permitido no solo conocer más a fondo cómo funciona internamente un compilador y cuáles son las partes que lo componen, sino que también hemos experimentado de primera mano los problemas que hay que afrontar a la hora de elegir e implementar los métodos de detección y emisión de errores de los programas.

En segundo lugar, remarcar que no hemos dedicar tanto tiempo a este proyecto como nos habría gustado, pues nos hemos visto inundados de tareas y entregas de distinta índole por parte de las asignaturas de matemáticas, además del resto de proyectos que teníamos de las asignaturas de informática que, en conjunto con lo anterior, nos han quitado mucho tiempo de nuestro día a día.

Por último y como cierre de este apartado, nos gustaría felicitar a los profesores, en especial a nuestro profesor Eduardo, ya que aunque la situación ha resultado ser completamente adversa, siempre ha estado ahí para responder cualquier duda de forma casi inmediata, además esta es una de las asignaturas en las que mejor explicada hemos encontrado la práctica y todo lo necesario para resolverla, lo cual puede ser bastante determinante en la impresión final que deja la asignatura en los alumnos y propicia que este trabaje más placenteramente en el proyecto. Solo comentar que quizás sería más interesante comenzar el proyecto con un compilador medio hecho, por ejemplo este mismo, y añadir funcionalidades más interesantes y complejas.

8. Referencias

[1] Eduardo Martínez Gracia. *Prácticas Compiladores 19/20 miniPascal.* Practicas2019_20_miniPASCAL.pdf

[2] Free Software Foundation. *GNU Bison 3.6.* https://www.gnu.org/software/bison/manual/html_node/index.html