
El lenguaje de programación Plis

Juan Pedro Bolívar Puente

Ingeniería Informática
Universidad de Granada

Curso 07/08

Índice

1. Introducción	3
1.1. ¿Por qué ese nombre?	3
2. Sintáxis	3
2.1. Comentarios	4
3. Funciones	4
3.1. La función <code>main</code>	6
3.2. Nota sobre los símbolos del usuario	6
4. Tipos de datos	6
5. Variables	7
6. Constantes	8
7. Sentencias	8
7.1. Expresiones	9
7.1.1. Llamadas a funciones	9
7.1.2. Operador de asignación	9
7.1.3. Operadores aritméticos	9
7.1.4. Operadores de incremento y decremento	10
7.1.5. Operadores a nivel de bit	10
7.1.6. Operadores de desplazamiento	10
7.1.7. Operador de indirección	11
7.1.8. Operador de dereferencia	11
7.1.9. Operador índice	11
7.1.10. Operador tamaño	11

7.1.11. Operador parámetro	11
7.1.12. El casting	12
7.1.13. Operadores comparadores	12
7.1.14. Operadores lógicos	12
7.2. Estructuras de control	13
7.2.1. Estructuras <code>if/else</code>	13
7.2.2. Bucles <code>while</code>	14
7.2.3. Bucles <code>repeat</code>	14
7.2.4. Manipulación de bucles	15
7.3. La sentencia <code>return</code>	15
8. El preprocesador	15
8.1. Inclusión de archivos	15
8.2. Substitución de tókens	16
8.3. Protección de bloques de código	16
9. El compilador	17
9.1. Compilando un programa	17
9.1.1. Utilizando <code>gcc</code>	17
9.1.2. Utilizando <code>as</code>	18
9.2. TO-DO	18
9.2.1. Mejoras pequeñas	19
9.2.2. Mejoras grandes	21

1. Introducción

El lenguaje de programación Plis ha sido ideado para experimentar con la generación de código en ensamblador de Intel x86, con el fin de aprender los entresijos del lenguaje ensamblador y adentrarse en el mundo de las estructuras de los computadores a nivel de lenguaje máquina. Es por eso que su sintaxis extraña y rudimentaria no tiene otra finalidad más que facilitar el análisis léxico y semántico del código, pudiendo centrarnos así en las partes relacionadas puramente con la generación de código máquina.

1.1. ¿Por qué ese nombre?

A pesar de ser aparentemente absurdo, el nombre del lenguaje es muy representativo de varias características del mismo:

1. Plis ha sido diseñado e implementado *en un plis*.
2. El uso de la notación prefija hace que se parezca a Lisp (pero en Plis los paréntesis son opcionales, hurra!), así que hemos tomado el nombre de ese lenguaje y le hemos aplicado la operación ROR del x86.
3. PLIS viene a ser un *Programming Language for Idiots and Students*.

2. Sintáxis

Plis es un lenguaje imperativo y pretende implementar la mayor parte de las funcionalidades de C de una forma compatible con éste (es decir, que pueda llamarse a funciones escritas en C desde Plis y vice-versa).

Un programa escrito en Plis puede definirse como una secuencia de tokens. Podemos definir un token como cualquier secuencia de caracteres del conjunto $ASCII - S$, dónde $S = \{ , \backslash f, \backslash r, \backslash v, \backslash t, \backslash n, (,) \} \subset ASCII$ es el conjunto de caracteres separadores. Cada token puede separarse de otro por una secuencia cualquier de caracteres de S . Existe una clase de tóken especial que puede contener caracteres separadores y es aquél que empieza por un token " y acaba por un token " que no esté precedido por una contrabarra \.

Puede resultar extraño en un principio observar que los caracteres (y) son caracteres separadores. En Plis se conoce casi siempre el número de

operandos, por lo que no es necesario rodear los parámetros de una forma especial y, cuando no es así, basta con especificar el final de la lista con el token `]`, cuyo uso concreto ya se discutirá más adelante. Es por esto que las expresiones largas a veces pueden ser confusas para quien no está acostumbrado, por lo que opcionalmente pueden usarse estos caracteres para estructurar el programa como más convenga.

2.1. Comentarios

Existe un elemento de sintaxis especial que no se ha mencionado, son los comentarios. Son secuencias de caracteres especiales que no son interpretadas por el compilador. Los hay de dos tipos:

- *Comentarios de bloque.* Se considera comentario de bloque a cualquier carácter que se encuentre entre un carácter `{` y un carácter `}`. Los comentarios de bloque pueden anidarse. Ejemplo:

```
{ Esto es un Comentario
  { Esto también } Y esto
}
```

```
esto_no_es_un_comentario
```

- *Comentarios en línea.* Se considera comentario en línea a cualquier carácter que se encuentre entre un carácter `;` y un carácter `\n`. Este tipo de comentarios no puede anidarse. Ejemplo:

```
esto_no_es_un_comentario ; Esto si ; lo es
esto_tampoco_es_un_comentario
```

3. Funciones

En Plis todo el código del programa debe estar organizado en funciones o procedimientos. Una función recibe una serie de parámetros y devuelve opcionalmente un valor. La cabecera de una función debe tener la siguiente estructura:

```
func nombre_de_funcion tipo_de_retorno parámetros ]
```

El tipo de retorno puede ser o bien un *tipo de datos*, que se describirán más adelante, o bien el token `none` para indicar que la función no devuelve ningún valor. Los parámetros se indicarán como un tipo de datos seguido de un símbolo definido por el usuario que servirá para referenciar al parámetro en la definición de la función. Opcionalmente el nombre del parámetro puede omitirse, lo cual, si bien es útil en las declaraciones de la función, carece de sentido -aunque se permite- en la definición ya que hace inaccesible al parámetro salvo mediante el uso del operador `param`. Al final de la lista de parámetros puede especificarse el tóken `...` para indicar que la función puede recibir un número variable de parámetros a parte de los descritos explícitamente, a los cuales se podrá acceder mediante el operador `param`. La lista de parámetros finaliza con el carácter `]`.

La cabecera de la función debe estar presente tanto en la declaración como en la definición de la función. Estos dos conceptos se describen así:

- *Declaración.* Para poder ser usada una función, la función llamadora debe conocer los parámetros de ésta y su valor de retorno. Si la función está definida después de la función llamadora o en otro fichero, podemos proporcionar esta información a la función llamadora incluyendo la declaración antes de ésta. Una función puede ser declarada varias veces siempre que los parámetros y valor de retorno sean los mismos en todas las declaraciones.

Para declarar la función simplemente ponemos el tóken `]` detrás de la cabecera. Ejemplo:

```
func max int int a int b ] ]
```

- *Definición.* La definición de una función es la descripción de lo que hace. Una función sólo puede ser definida una vez. Los parámetros y valor de retorno de la definición de la función deben coincidir con los de la declaración de la función. Si la función no ha sido declarada aún, la definición también declara la función.

Para definir una función debemos añadir una secuencia de sentencias, opcionalmente precedidas por la declaración de variables -se describirá más adelante- tras la cabecera acabada por el tóken `]`. Ejemplo:

```
func max int int a int b ]
  if > a b
    return a
```

```

    ]
    return b
]

```

3.1. La función `main`

La función `main` es aquella que se ejecutará al principio del programa. No todo fichero fuente debe contener una función `main`, pero en todo programa debe haber una -y sólo una- función `main`.

La función `main` no debe tener parámetros, y debe devolver un entero que será el valor de retorno del programa.

3.2. Nota sobre los símbolos del usuario

Es necesario llegados a este punto realizar una pequeña aclaración sobre los símbolos de usuario. Como habrá observado el lector, no hay ninguna restricción sobre el conjunto de los caracteres ASCII utilizables en los nombres de funciones. Sin embargo, el compilador tiene que transformar esos nombres en símbolos identificables tanto por el código de este fichero como por código objeto externo.

Por desgracia, el ensamblador no puede reconocer en esos símbolos cualquier carácter. Es por eso que internamente, tanto para los símbolos de funciones como de variables globales, la implementación de Plis actual sustituye los caracteres no alfanuméricos salvo el `_` y el dígito que pueda abrir la cadena como: `__número ascii del carácter__`. Esto significa que si somos muy guasones y definimos dos funciones con nombres `:-]` y `__58____45____93__` respectivamente, vamos a tener problemas durante el ensamblado del código.

4. Tipos de datos

Se han mencionado lo que es un tipo de datos. Todo dato del programa (variable o parámetro) debe tener un tipo, que determinará su tamaño y las operaciones que se pueden hacer con él. Los tipos de datos básicos sirven para almacenar números y son los siguientes:

Tipo	Valores	Tamaño
<code>byte</code>	$[-2^7 + 1, 2^7 - 1]$	1 byte
<code>short</code>	$[-2^{15} + 1, 2^{15} - 1]$	2 byte
<code>int</code>	$[-2^{31} + 1, 2^{31} - 1]$	4 byte
<code>unsigned byte</code>	$[0, 2^8 - 1]$	1 byte
<code>unsigned short</code>	$[0, 2^{16} - 1]$	2 byte
<code>unsigned int</code>	$[0, 2^{32} - 1]$	4 byte

Existe un tipo de dato especial que es el *puntero*. El puntero ocupa 4 bytes y contiene la dirección de un espacio en memoria de un tipo de dato, es decir “apunta” a un dato de un tipo dado. Un tipo de datos puntero se forma anteponiendo el tóken `ptr` al tipo de dato al que apunta. El tipo de dato al que apunta no tiene por que ser un tipo de datos simple, siendo válidos estos dos ejemplos de punteros:

```
ptr short
ptr ptr ptr unsigned int
```

5. Variables

Todos los datos del programa son *variables*. Las variables tienen un tipo de dato y un identificador con el que hacemos referencia a ellas. En ellas podemos almacenar valores y operar con ellos. Las variables pueden ser de tres tipos:

1. *Variables globales*: Se declaran fuera de las funciones. Puede accederse a ellas desde cualquier función del programa que esté definida después de la variable. Para declararla debemos especificar su tipo de dato seguido de su identificador. Por defecto contienen siempre el valor 0.
2. *Variables locales*: Son variables que se declaran dentro de las funciones y a las que sólo se puede acceder dentro de ellas. No conservan su valor entre distintas llamadas a la función. Las variables locales *deben declararse al principio de la definición de la función*, es decir, antes de cualquier sentencia, especificando el tipo de dato seguido del identificador. Si existe una variable global con el mismo identificador la variable global quedará inaccesible desde la función. Hasta que no se les asigne un valor no está determinado que valor contienen, es decir, contienen “basura” de residuos del uso anterior de la zona de memoria dónde se ha creado la variable.

3. *Parámetros de función*: Se declaran de la forma en la que se especificó en la sección de sobre las funciones. Comparten las propiedades de las variables locales, con la salvedad de que inicialmente contienen el valor que se haya especificado en la llamada a la función.

Para utilizar una variable como parámetro de alguna expresión basta con especificar su identificador. Existe un operador especial que veremos más adelante que propone un método alternativo para acceder a los parámetros de una función.

6. Constantes

A lo largo del programa se pueden especificar valores constantes. Estos pueden ser de dos tipos:

1. *Constantes numéricas*: Permiten especificar números. Las siguientes expresiones regulares definen los lenguajes que los representan:

- a) $(+|-)?(0|1|2|3|4|5|6|7|8|9)+$ Constantes numéricas decimales.
- b) $(+|-)?0x(0|1|2|3|4|5|6|7|8|9)+$ Constantes numéricas especificadas en hexadecimal.

2. *Constantes de cadena*: Representan cadenas de caracteres. Es una constante de cadena cualquier secuencia de caracteres arbitraria situada entre dos dobles comillas. La cadena puede contener todas las secuencias de escape válidas en las cadenas de C, incluidas la secuencia `\`, indispensable para representar las dobles comillas en la cadena.

7. Sentencias

Un programa en Plis es esencialmente una secuencia de sentencias encapsuladas en funciones. Una sentencia puede modificar el curso de ejecución si se trata de una estructura de control, o realizar acciones en torno a unos parámetros y devolver un valor si se trata de una expresión.

Veamos primero las expresiones.

7.1. Expresiones

En Plis, todas las expresiones tienen notación prefija, esto es, siguen la siguiente sintaxis:

```
expr param_1 param_2 ... param_n
```

Si la expresión tiene un número variable de parámetros, la lista de parámetros debe ir finalizada con el token `]`. Notar que cualquier expresión puede servir de parámetro para otra expresión. La expresión elemental, obviamente, es el identificador de variable, o una constante.

7.1.1. Llamadas a funciones

Se puede invocar la ejecución de una función utilizando su nombre como identificador de la expresión. El tipo y número de parámetros dependerá de la función.

7.1.2. Operador de asignación

El operador de asignación `:=` tiene dos parámetros y graba en la variable del primer parámetro a el resultado de la expresión pasada como segundo parámetro. La expresión devuelve a la variable a la que se ha asignado el valor.

7.1.3. Operadores aritméticos

En Plis existen los siguientes operadores binarios para la realización de operaciones aritméticas. Estos operadores reciben dos operandos numéricos y devuelven, según el operador:

- `+`: La suma de los dos parámetros.
- `-`: La diferencia de los dos parámetros.
- `*`: El producto de los dos parámetros.
- `/`: El cociente entero entre los dos parámetros.
- `%`: El resto de dividir el primer parámetro entre el segundo.

Notar que estos dos últimos operadores acabarán con el flujo de ejecución del programa si el segundo parámetro vale 0.

Mencionar también que el operador suma puede ser utilizado con punteros en el contexto especial en el que uno de los parámetros es puntero y otro es un número n . En ese caso, el resultado de la operación será la posición de memoria a la que apuntaba el puntero incrementada en n celdas, por lo que el valor real sumado dependerá del tamaño del tipo apuntado.

7.1.4. Operadores de incremento y decremento

Los operadores de incremento y decremento suman y restan 1 al contenido de la variable pasada como parámetro y, como en el caso de la suma, de ser éste de tipo puntero se sumará o restará el valor adecuado para que el aumento o decremento sea de una celda completa.

Existen dos variaciones de estos operadores. Los primeros, preincremento y predecremento, respectivamente $++>$ y $-->$, devuelven el valor después de actuar. Los segundos, postincremento y postdecremento, respectivamente $>++$ y $>--$, devuelven el valor que tenía la variable antes de que se le aplicara el incremento o decremento.

7.1.5. Operadores a nivel de bit

Los siguientes operadores binarios realizan operaciones a nivel de bit, es decir, aplican una operación sobre los pares de bits *i-ésimos* de los dos parámetros. Son los siguientes:

- $\&$: Realiza la operación *and* sobre los bits de los dos parámetros.
- $|$: Realiza la operación *or* sobre los bits de los dos parámetros.
- \wedge : Realiza la operación *xor* sobre los bits de los dos parámetros.

7.1.6. Operadores de desplazamiento

Los operadores de desplazamiento desplazan los bits de la primera expresión a la izquierda o a la derecha tantas posiciones como se indique en el segundo parámetro. Los nuevos bits que se introducen son puestos a cero y los que sobrepasen el lado hacia al que se desplaza se pierden. Según el

desplazamiento sea la izquierda o derecha el símbolo del operador será << o >> respectivamente.

7.1.7. Operador de indirección

El operador de indirección <- devuelve la posición en memoria dónde se encuentra una variable. El valor de retorno será por tanto de tipo puntero al tipo contenido en la variable pasada como parámetro.

7.1.8. Operador de dereferencia

El operador de dereferencia -> recibe como parámetro un puntero y devuelve la variable a la que apunta. El valor de retorno será del tipo al que apuntaba el parámetro.

7.1.9. Operador índice

El operador índice ' recibe dos parámetros. El primero debe ser un entero n y el segundo un puntero, y el resultado será la dereferencia de la celda n -ésima a partir de la que apunta el puntero.

7.1.10. Operador tamaño

El operador `size` devuelve el tamaño del tipo de datos de la expresión pasada como parámetro. Notar que, aunque el tamaño puede determinarse en tiempo de compilación, la expresión se evalúa de todas formas. Este operador es especial en cuanto a que el parámetro puede no ser una expresión sino un tipo de dato, en cuyo caso devolverá el tamaño del tipo de dato.

7.1.11. Operador parámetro

El operador `param` recibe como parámetro un entero n y devuelve el parámetro n -ésimo de la función en la que estamos. Ésto es necesario si estamos en una función con número variable de parámetros.

Notar que modificar elementos que no se nos han pasado como parámetros puede romper la pila y causar todo tipo de comportamientos inesperados, así que si se usan parámetros variables lo mejor es asegurarse mediante una

correcta documentación de la función y algún parámetro adicional que proporcione información sobre los parámetros que se pasarán, como puede ser un entero que indique el número de parámetros pasados o una cadena de formato como usa la función `printf`.

El valor de retorno será de tipo entero, independientemente de que el parámetro fuese explícito o no, y es tarea del programador hacer el casting adecuado al usar el valor.

7.1.12. El casting

El casting devuelve el resultado de una expresión pero como si fuese de otro tipo distinto. Para realizar un casting basta con anteponer un tipo de datos delante de una expresión.

7.1.13. Operadores comparadores

Esta serie de operadores sirven para comparar los dos valores que se le pasan como parámetros. Devuelven 1 si la comparación es cierta y 0 si es falsa. Estos operadores son:

- `<` La comparación es cierta si el primer parámetro es menor que el segundo.
- `<=` La comparación es cierta si el primer parámetro es menor o igual que el segundo.
- `>` La comparación es cierta si el primer parámetro es mayor que el segundo.
- `>=` La comparación es cierta si el primer parámetros es mayor o igual que el segundo.
- `=` La comparación es cierta si los dos parámetros son iguales.
- `!=` La comparación es cierta si los dos parámetros son distintos.

7.1.14. Operadores lógicos

Los operadores lógico sirven para evaluar expresiones lógicas como el anterior. En Plis cualquier valor significará verdad si es distinto de cero y falso

si es cero. Por tanto cuando a continuación nos refiramos a los valores de los parámetros de los operadores lógicos en términos de verdadero o falso, debemos asumir que estos parámetros pueden ser cualquier expresión y que nos referimos al valor de esas expresiones según el criterio convenido. Los operadores lógicos son los siguientes:

- *Operador NO !* Recibe un parámetro y devolverá 1 si el parámetro es falso y 0 en caso contrario.
- *Operador Y &&* Recibe dos parámetros y devolverá 1 si ambos son ciertos y falso si cualquiera de los dos es falso.
- *Operador O ||* Recibe dos parámetros y devolverá 1 si cualquiera de los dos es cierto y falso si ambos son falsos.

7.2. Estructuras de control

Las estructuras de control sirven para modificar el flujo del programa según el resultado de ciertas condiciones. A continuación se describen.

7.2.1. Estructuras if/else

Esta estructura permite que un bloque de sentencias se ejecute solamente si se cumple cierta condición. Opcionalmente podemos incluir otro bloque de sentencias que se ejecutará solamente si la condición fue falsa. La forma es la siguiente:

```
if condicion
    bloque_sentencias_1
] else
    bloque_sentencias_2
]
bloque_sentencias_3
```

Dónde *condicion* es una expresión. Si devuelve verdadero se ejecutará el primer bloque de sentencias. Si devuelve falso se ejecutará el segundo bloque de sentencias, *bloque_sentencias_2*. En ambos casos se ejecutará después *bloque_sentencias_3*, continuando con la ejecución normal del programa.

Notar que la parte `else ...]` es opcional, si queremos puede omitirse, no existiendo ningún conjunto de sentencias que se ejecute exclusivamente si la condición es falsa.

7.2.2. Bucles while

Los bucles nos permiten ejecutar reiteradamente una serie de sentencias hasta que se cierta condición. Los bucles `while` en concreto evaluarán esa condición antes de empezar a ciclar, y ciclarán mientras la condición devuelva cierto. Su sintaxis es la siguiente:

```
while condicion
    bloque_sentencias_1
]
bloque_sentencias_2
```

Dónde `condicion` es una expresión. Al llegar a la sentencia `while` se evaluará `condicion`. Si es cierta se ejecutará `bloque_Sentencias_1`, y si es falsa se pasará directamente a `bloque_sentencias_2`. En caso de ejecutarse `bloque_sentencias_1`, tras ejecutarse el bloque se volverá a evaluar `condicion` y si es cierto se volverá a repetir este proceso hasta que en algún momento la condición sea falsa y se pase a `bloque_sentencias_2`.

7.2.3. Bucles repeat

Un bucle `repeat` es similar a un bucle `while` salvo que la primera vez no se evaluará la condición ejecutándose directamente el bloque de sentencias. La sintaxis es la siguiente:

```
repeat
    bloque_sentencias_1
] condicion
bloque_sentencias_2
```

Cuando se llega a la sentencia `repeat` se pasa directamente a la ejecución de `bloque_sentencias_1`. Después se evaluará `condicion` y en caso de ser cierto se volverá a ejecutar `bloque_sentencias_1`, proceso que ocurrirá hasta que `condicion` sea falsa y se continúe con la ejecución del programa en `bloque_sentencias_2`.

7.2.4. Manipulación de bucles

Existen dos sentencias especiales que permiten alterar el curso de los bucles que se han descrito, tanto **while** como **repeat**. Estas son las siguientes:

- **loop** La ejecución de esta sentencia implica que se pase a la próxima iteración del bucle. Esto es, se dejarán de ejecutar el resto de sentencias y se evaluará de nuevo la condición del bucle y, si devuelve verdadero, se comenzará desde el principio una nueva iteración.
- **stop** Esta sentencia causa que se abandone la ejecución del bucle, saltando directamente al bloque de sentencias posterior.

7.3. La sentencia return

La sentencia **return** sirve para abandonar la ejecución de una función en el punto en el que se encuentre la sentencia. Si la función devuelve **none** la sentencia debe ponerse tal cual. En caso contrario debe ir seguida de una expresión, cuyo resultado será el valor de retorno de la función.

8. El preprocesador

Antes de compilar un programa el compilador de Plis realiza un preprocesado del código, aunque bastante elemental. Esto es, realiza ciertas operaciones elementales de sustitución de código entre otras que pueden ser muy convenientes. A continuación se describen las posibilidades que ofrece.

8.1. Inclusión de archivos

El preprocesador de Plis permite copiar virtualmente el contenido de otro fichero en un punto dado del programa. La funcionalidad principal de esto es tener, por ejemplo, las cabeceras de las funciones en un fichero a parte. Utilizando esta funcionalidad puedes incluir el fichero antes de usar esas funciones en alguna otra parte.

La sintaxis para realizar la inclusión de archivos es la siguiente:

```
include "fichero"
```


Notar que `fichero` puede ser o bien una ruta absoluta, o bien una ruta relativa al directorio en el que se encuentra el fichero dónde se realiza el `include`.

8.2. Substitución de tókens

A través de esta funcionalidad se permite, mediante la directiva `def`, sustituir un tóken por otro en todo el código siguiente.

La sintaxis es así:

```
def token_orig token_dest
```

A partir de esa sentencia cada aparición de `token_orig` será sustituida por `token_dest`. Los tókens del preprocesador también pueden redefinirse. Además, en futuros `def`, `token_orig` no es sustituido, de tal forma que puede darse un nuevo significado a un tóken. Sin embargo, `token_dest` sí es sustituido, por lo que no se puede redefinir un tóken y volverle a dar su sentido original.

8.3. Protección de bloques de código

A través de inclusiones de ficheros sucesivas podemos caer en la eventualidad de que un mismo fichero sea incluido varias veces en el mismo fichero, con lo que podríamos tener problemas por redefiniciones de símbolos, etc...

Para solucionar este problema se han incluido los comandos del preprocesador `lock` y `unlock`. La sintaxis es como sigue:

```
lock identificador
    un bloque de codigo
unlock
```

```
lock identificador
    otro bloque de codigo
unlock
```

La acción `lock` se realiza sobre un identificador. Si antes de realizar el `lock` no ha habido ningún otro cerrojo con ese mismo identificador, se sigue

sustituyendo normalmente. En cambio, si ha existido otro `lock` con el mismo identificador, el código que haya hasta el `unlock` correspondiente será omitido.

9. El compilador

En la actualidad el lenguaje Plis sólo tiene un compilador que, además, es compatible únicamente con máquinas corriendo GNU/Linux sobre procesadores Intel 386 o superior. El compilador genera código ensamblador con sintaxis AT&T.

El compilador de Plis se llama `plisc` y su modo de empleo puede encontrarse directamente en la ayuda del programa:

```
plisc, plis compiler
(c) 2007 Juan Pedro Bolívar Puente
```

Usage:

```
plisc input_files -o output_file.s [options]
```

Options:

<code>-h, --help</code>	Show help.
<code>-v, --version</code>	Show version.
<code>-o, --output <file></code>	Write output to a file. Else, I will use stdout.
<code>-s, --start</code>	Write starter function to assemble the output without GCC.

9.1. Compilando un programa

Para compilar un programa con `plisc` tenemos que seguir tres pasos, primero compilar el código en Plis, luego ensamblar el código y, por último, enlazarlo para obtener el ejecutable. Esto podemos hacerlo de dos formas.

9.1.1. Utilizando gcc

Lo más sencillo es utilizar `gcc` para el ensamblado y el enlazado.

Un ejemplo podría ser:

```
$ plisc programa.plis -o salida.s
$ gcc salida.s -o programa
```

O tal vez más cómodamente podemos redireccionar las salidas para no utilizar ficheros temporales:

```
$ plisc programa.plis | gcc -x assembler -o programa -
```

Utilizar `gcc` es especialmente interesante si queremos utilizar funciones de C ya que `gcc` se encarga del enlazado e incluso de compilar otros módulos escritos en C.

9.1.2. Utilizando `as`

Otra forma de compilar es utilizando el ensamblador *GNU as* y en enlazador manualmente. De ésta forma ganamos control sobre con qué librerías se enlaza. En este caso tenemos que tener en cuenta que debemos pasar la directiva `-s` a `plisc` para que incluya el código de inicio y salida del programa que `gcc` genera automáticamente.

Un ejemplo de ensamblado de esta forma puede ser:

```
$ plisc -s programa.plis | as -o programa.o
```

Ahora en enlazado lo podemos hacer así:

```
$ ld programa.o -o programa
```

Tenemos que tener en cuenta que, si queremos utilizar funciones de la librería estandar de C, debemos pasarle los siguientes parámetros al enlazador:

```
$ ld programa.o -o programa \
  --dynamic-linker /lib/ld-linux.so.2 -lc
```

9.2. TO-DO

El compilador no es un proyecto acabado. Aunque cumple, espero, sobradamente los requisitos para aprobar la práctica, está aún lejos de ser un compilador completo. A continuación expongo algunas mejoras que he pensado y que ahí están en el tintero -o mejor dicho, en el teclado- hasta que o bien me anime retomar el proyecto por afición personal o bien el profesor me sugiera que la implementación de alguna de estas mejoras podría contribuir a mejorar la nota :)

9.2.1. Mejoras pequeñas

En esta categoría entran mejoras que no suponen un cambio sustancial en la concepción del programa y estimo el tiempo que llevaría implementarlas sería comparable al que ha llevado implementar lo que ya existe. Estas son:

1. *Vectores en la pila y en sección de datos*

Actualmente Plis soporta vectores indirectamente gracias a que tiene punteros y varios operadores relacionados y que puede acceder a las funciones de C para manejo de memoria como `malloc` y `free`.

Creo que sería interesante implementar aún así vectores en la pila y en sección de datos que pueden ser útiles en distintos contextos. Realmente sería algo sencillo y sólo me falta perfilar un par de detalles y ponerme a implementarlo.

2. *Estructuras*

En el Plis actual se puede implementar cualquier algoritmo y se pueden, como se acaba de discutir, vectores de datos. Pero la carencia de estructuras de datos supone una limitación importante. Por ejemplo, implementar algo tan elemental como una lista enlazada se convierte en algo incómodo y peligroso.

Implementar estructuras realmente no sería complicado. Basta con guardar la información de la estructura e implementar el operador `.`, que intuyo que puede ser sencillo. Por tanto, esta es otra mejora interesante que se puede realizar.

3. *Más tipos de datos*

Sería interesante tener algún equivalente a los `long`, `float` y `double` de C, ya que en muchos contextos se hacen indispensables. Implementar los `floats` no debe ser excesivamente difícil si se usan las instrucciones del coprocesador matemático del. Por otro lado, implementar tipos de 8 bytes implicaría realizar algunos cambios en cómo se manejan actualmente los registros, así que habría que estudiarlo con detenimiento.

4. *Punteros a funciones*

Otra gran carencia del lenguaje que sería interesante suplir. Realmente tampoco creo que sea complicado de implementar, puede que sea incluso la más sencilla de las tres mejoras que acabo de proponer.

5. Optimizaciones

Aunque el código que genera el compilador en la actualidad no es del todo malo. De hecho es comparable al que genera `gcc` cuando no se le activan las optimizaciones, lo cual considero un logro habiendo sido desarrollado en dos semanas y con muchas otras cosas que hacer :-)

Sin embargo, aún se le podrían incluir más optimizaciones. Algunas requerirían grandes cambios, pero otras creo que tal vez no serían muy complicadas. Por ejemplo, algo tan sencillo como tratar de recordar si el valor de una variable está residualmente en un registro fruto de operaciones anteriores puede ahorrarnos transferencias innecesarias a memoria. También es posible que se puedan hacer algunas predicciones sencillas de los valores de las variables y cosas así que pueden mejorar la eficiencia del código.

6. Refactorización del código

Como el código de `plisc` ha sido escrito con bastantes prisas a veces se ha escrito el código sin pensarse mucho las cosas y a veces el código se ha vuelto especialmente sucio y maloliente.

Un punto para empezar a refactorizar podría ser la clase `Value` y sus funciones asociadas en `compiler_value.cpp`. Esa clase ha ido creciendo a saltos y ahora hay cosas poco limpias y oscuras, constantes que han perdido su significado original, etc... Ya tengo alguna idea de por dónde podrían ir las mejoras y sería cuestión de pensar un poco en como definirlas e implementarlas.

Otro punto importante es el manejo de errores. Actualmente se está usando excepciones, pero esto tiene dos problemas. Primero, que no toda la memoria que reservamos es liberada por un destructor. Por lo tanto al lanzarse una excepción dentro de expresiones y similar se producen *memory leaks*¹. Además sería interesante incluir un modelo de gestión de errores un poco más complejo que trate de volver a un punto dónde pueda recuperarse y continuar simular la compilación del código siguiente para detectar otros posibles errores y evitar tener que seguir un tedioso proceso de *compilar, arreglar error, compilar, arreglar error....* Además, los mensajes de error que se muestran a veces no son muy descriptivos y sería conveniente enriquecerlos, mostrar *warnings*, etc... La verdad es que en todo esto no he pensado mucho y tendría que valorar hasta que punto es factible realizar estos cambios.

¹Aunque he sido cuidadoso y he comprobado, `valgrind` en mano, que no se producen *memory leaks* cuando no se producen excepciones

Otra refactorización importante que me gustaría hacer es cambiar el modelo imperativo de la traducción de expresiones por una orientada a objetos dónde cada expresión vaya representada por un objeto. Antes de ponerme a implementar la versión actual estuve pensando un poco al respecto pero abandoné la idea porque el otro ya lo tenía más o menos pensado y era más rápido. Es interesante y creo que posiblemente sea bastante factible hacerlo y ayudaría a hacer mucho más versátil y fácil el código y solucionaría alguno de los problemas citados como el de los *memory leaks*. El problema es que tal vez esta mejora debería de entrar en las *mejoras grandes*.

Y por supuesto, otra cosa que hay que hacer es poner comentarios, que actualmente creo que el código roza las 0 líneas de comentarios.

9.2.2. Mejoras grandes

Estas mejoras llevarían bastante tiempo desarrollarlas e investigar sobre temas en los que aún estamos un poco verdes. Entre otras cosas, una de las características principales del **plisc** actual es que todo el proceso de compilación se realiza en dos pasadas: el lexer y el generador de código. Además cada uno de ellos realiza una única pasada por el código, lo cual nos limita al hacer algunas cosas pero permite que el programa sea sencillo y eficiente.

La idea sustancial de las mejoras grandes es convertir al compilador en algo que no sea un *cutre-compilador* sino en algo serio. Para esto, lo fundamental es dividir el proceso de compilación en tres etapas: análisis léxico, análisis semántico y generación de código.

A su vez, para realizar estas dos primeras etapas creo que sería sustancial usar herramientas como **lex** y **bison** para que sean realizadas de forma eficiente. La idea fundamental es que el generador de código no reciba código, o una lista de tokens, como hace ahora, sino una estructura en árbol que represente un lenguaje genérico de semi-bajo nivel parecido a C. Algo parecido es lo que, de hecho, hace **gcc**.

En cierto modo es lo que ya hacemos. El lenguaje Plis, por su sintaxis con notación prefija y su funcionalidad básica representa en cierto modo esa estructura en árbol. Sin embargo sería mucho más conveniente definir una interfaz adecuada que nos permita añadir funcionalidad en ambos sentidos, es decir, escribir generadores de código eficientes de forma cómoda para distintas arquitecturas y, a la vez, escribir interfaces para lenguajes variados.

Realmente podría pasarme horas disertando sobre los problemas que podríamos encontrarnos, las cosas que habría que investigar, proponer algunas cosas al diseño inicial... Sin embargo, realmente estas grandes mejoras carecen de sentido porque todo eso ya lo hace en genial y glorioso `gcc`. De hecho, tal vez la gran mejora podría ser implementar Plis como un *frontend* de `gcc`, aunque de ese modo abandonaríamos la finalidad de esta práctica, aprender las entrañas del ensamblador, y pasaríamos al sobrecogedor mundo de los procesadores de lenguaje. Pero no he podido evitar ojear muy por encima la documentación del `gcc` y de un par de *howtos* y realmente promete ser una aventura divertida.