



UNIVERSIDAD DE VALLADOLID

GRUPO G

GRAMÁTICA Y LENGUAJES FORMALES

Informe Proyecto P3

Manuel de Castro Caballero
María Ruiz Molina
Andrés Trigueros Vega

13 de junio de 2020

Índice

1. Introducción	2
2. Resumen de las especificaciones del lenguaje	2
2.1. Especificaciones léxicas iniciales revisadas	2
2.2. Especificaciones sintácticas iniciales revisadas	4
2.2.1. Sentencias	4
2.2.2. Expresiones	5
2.3. Añadidos	6
2.3.1. Propuestas desechadas	7
3. Discusión de los ficheros fuente, y justificación de las decisiones de diseño e implementación	8
3.1. Lex - <code>ac.l</code>	8
3.2. Yacc - <code>ac.y</code>	9
3.3. <i>AST</i> - <code>ast.c</code> y <code>ast.h</code>	10
3.4. Tabla de símbolos - <code>symtab.c</code> y <code>symtab.h</code>	12
3.5. Utilidades - <code>autils.h</code>	12
3.6. Bibliotecas de C utilizadas	13
4. Compilación y ejecución del intérprete	13
5. Ejemplos desarrollados	14
6. Problemas conocidos	17
7. Apéndice - Sobre la elaboración del proyecto	18
7.1. Comunicación entre miembros	18
7.2. Plataformas y herramientas utilizadas	18
7.3. Reparto de trabajo	19
8. Referencias bibliográficas	20

1. Introducción

Este proyecto ha consistido en el desarrollo de un intérprete de un lenguaje de programación A, empleando `lex` y `yacc`¹ para realizar los analizadores léxico y sintáctico, respectivamente, del mismo. Además, se ha optado por utilizar un *AST*, para compilar el lenguaje a una representación intermedia, produciendo un intérprete más original y sofisticado.

A es un lenguaje de programación básico, de baja potencia expresiva, no tipado, derivado (principalmente) del archiconocido lenguaje de programación C. El lenguaje no está enfocado con ningún propósito real concreto, sino como una “prueba de concepto” de un lenguaje de programación sencillo pero perfectamente funcional, con las que se puedan realizar las tareas más básicas esperables de un lenguaje de programación.

2. Resumen de las especificaciones del lenguaje

Pese a ya haber redactado un documento previo específico donde se detallaban las especificaciones del lenguaje, se ha considerado conveniente realizar un resumen de las mismas en este documento, ya que algunas de ellas han sufrido modificaciones por inconvenientes que han ido surgiendo, y se ha añadido alguna más de las inicialmente especificadas.

2.1. Especificaciones léxicas iniciales revisadas

- Identificadores: Letra o un guión bajo, seguido por ninguna o varias letras, caracteres numéricos, o guiones bajos. Las letras, pueden ser tanto mayúsculas como minúsculas. A es un lenguaje sensible ante mayúsculas y minúsculas.
- Constantes numéricas: Las dos constantes numéricas definidas en el intérprete son:
 - Reales: Uno o más caracteres numéricos, seguidos de un punto, y cero o más caracteres numéricos o de un punto seguido de uno o más caracteres numéricos.
 - Enteras: Literales escritos en el sistema decimal conformadas por uno o más dígitos del sistema decimal.
- Separadores: Para separar sentencias entre sí, se utiliza el carácter “salto de línea” (`\n`). Para delimitar los bloques de sentencias, se utilizan los caracteres ‘{’ y ‘}’.

¹más concretamente, sus respectivas implementaciones como `flex` y `bison`

- Comentarios: Los comentarios se pueden especificar de dos maneras:
 - Los comentarios de una sola línea, están compuestos por tiras de caracteres que comienzan por “//” y acaban con un salto de línea (“\n”), teniendo una longitud arbitraria.
 - Los comentarios multilínea, están compuestos por tiras de caracteres que comienzan por “/*” y acaban con (“*/”), teniendo una longitud arbitraria.
- Operadores aritméticos:
 - ‘+’ : El operador binario “suma”, que añade el valor de la primera expresión al valor de la segunda expresión.
 - ‘-’ : El operador binario “resta”, que substrahe al valor de la segunda expresión, el valor de la primera expresión.
 - ‘*’ : El operador binario “multiplicación”, que suma el valor de la primera expresión tantas veces como indica el valor de la segunda expresión.
 - ‘/’ : El operador binario “división”, que realiza la multiplicación del valor de la primera expresión por el inverso del valor de la segunda expresión.
 - ‘^’ : El operador binario “exponenciación”, que multiplica el valor de la primera expresión tantas veces como indique el valor de la segunda expresión.
 - ‘%’ : El operador binario “módulo”, “resto” o “resíduo”, que devuelve el valor que hay que restarle a la primera expresión para que, al dividirlo entre la segunda expresión, devuelva un valor entero.
 - ‘-’ : El operador unario “cambio de signo”, que convierte el valor de la expresión en su opuesto.
 - ‘+’ : El operador unario “mantenimiento de signo”, que mantiene el valor de la expresión, y que se incluye por analogía al operador “cambio de signo”.
- Operadores lógicos:
 - ‘==’ : El operador binario “igualdad”; compara el valor del primer argumento con el del segundo, evaluándose al valor real ‘1’ en caso de que sean el mismo, y ‘0’ en caso contrario.
 - ‘!=’ : El operador binario “desigualdad”; compara el valor del primer argumento con el del segundo, evaluándose al valor real ‘1’ en caso de que sean distintos, y a ‘0’ en caso contrario.

- ‘<’ : El operador binario “menor que”; compara el valor del primer argumento con el del segundo, evaluándose al valor real ‘1’ en caso de que el primer valor sea menor que el segundo, y a ‘0’ en caso contrario.
 - ‘>’ : El operador binario “mayor que”; compara el valor del primer argumento con el del segundo, evaluándose al valor real ‘1’ en caso de que el primer valor sea mayor que el segundo, y a ‘0’ en caso contrario.
- Palabras clave:
- **read**: Dedicada a la lectura de valores numéricos por entrada estándar. Opcionalmente, puede imprimir un mensaje previo a dicha lectura.
 - **print**: Dedicada a la impresión de caracteres y valores de variables por salida estándar.
 - **if**: Dedicada a las sentencias condicionales.
 - **else**: Dedicada en conjunto con **if** a las sentencias condicionales.
 - **while**: Dedicada a los bucles.
 - **Funciones predefinidas**: Se consideran funciones aquellas que se puedan evaluar a un valor numérico:
 - **sin**: Función real cuyo dominio es \mathbb{R} y su codominio el intervalo cerrado $[-1, 1]$. El seno es una función continua con periodo 2π .
 - **cos**: Función real cuyo dominio es \mathbb{R} y su codominio el intervalo cerrado $[-1, 1]$. El coseno es una función par y continua con periodo 2π .
 - **tan**: Función impar y con periodo π , con indeterminaciones en $\pi/2 + n\pi$, $n \in \mathbb{Z}$.
 - **ln**: Función que aplica la función logaritmo neperiano a la expresión que va después.
 - **div**: Operador aritmético binario que devuelve el cociente de la división de la primera expresión entre la segunda.
- Cadenas de caracteres: Conformadas por una serie de símbolos del conjunto de caracteres ASCII escritos entre comillas dobles (“” y ‘’) que se emplearán para la impresión de textos.

2.2. Especificaciones sintácticas iniciales revisadas

2.2.1. Sentencias

Una sentencia es la unidad de ejecución mínima de un programa A.
Las sentencias en A están formadas por:

- Un identificador, un carácter '=' y una expresión.
- La palabra reservada **if**, seguida de una expresión, una sentencia o bloque de sentencias, y, opcionalmente, de la palabra reservada **else** y una sentencia o bloque de sentencias.
- La palabra reservada **while**, seguida de una expresión y una sentencia o bloque de sentencias.
- La palabra reservada **print** y un conjunto no vacío de 0 o más cadenas de caracteres y 0 o más identificadores o variables.
- La palabra reservada **read**, seguida o no por una cadena de caracteres, y seguida por un identificador.

Todas las sentencias van terminadas por un separador.

Un bloque de sentencias es un conjunto de una o varias sentencias agrupadas entre los correspondientes delimitadores de bloques.

2.2.2. Expresiones

Una expresión es una entidad que puede ser evaluada a un valor numérico (entero o real).

En A, las expresiones pueden ser:

- Un identificador (es decir, una variable con un valor asignado).
- Una constante numérica real o entera.
- Una expresión entre paréntesis '(' y ')'.
 - Un operador unario (aritmético o lógico) seguido de una expresión.
 - Una expresión seguida de un operador binario (aritmético o lógico) y otra expresión.
 - Una expresión, la palabra reservada de operador binario binaria **div**, y otra expresión.
- alguna de las palabras reservadas de funciones predefinidas unarias (**sin**, **cos**, **tan** o **ln**), y una expresión entre paréntesis.

2.3. Añadidos

Respecto a las especificaciones inicialmente propuestas del lenguaje **A**, se han realizado los siguientes añadidos de **caracter principalmente léxico**:

- Operadores lógicos: Los operadores lógicos añadidos a los ya mencionados en las secciones previas son el *and* lógico (**&&**) y el *or* lógico (**||**). De manera análoga a los ya introducidos, estos simplemente son ejecutados, siguiendo su definición.
- Operaciones de bit: Se han añadido las operaciones de bit *AND* (**&**), *OR* (**|**), *NOT* (**~**), *shift left* (**<<**) y *shift right* (**>>**). Estas solo se pueden realizar sobre números enteros, por lo que ha sido necesario añadir esta diferenciación, como se explica a continuación.
- Diferenciación entre tipos “entero” y “real”. Esta característica ha sido bastante importante, pues a la hora de tratar con operaciones de bit tan solo se puede tratar con expresiones de tipo entero.

Además de las respectivas nuevas reglas léxicas y producciones sintácticas, para implementar esta característica ha sido necesario diferenciar en el **AST** hojas de tipo “número real” y hojas de tipo “número entero”.

- Números hexadecimales: Estos se identifican por comenzar con el prefijo “0x”, seguido de uno o varios dígitos hexadecimales (números del 0 al 9 o letras, mayúsculas o minúsculas, de la ‘a’/‘A’ a la ‘f’/‘F’). A la hora de trabajar con ellos, son tratados como números enteros.

Los números hexadecimales solo son aceptados en **A** como literales dentro del propio código; y no pueden ser leídos mediante una sentencia del tipo **read**.

Con respecto a **añadidos sintácticos**, se han añadido producciones a la gramática independiente del contexto para facilitar la interpretación de programas. De este modo, aunque la sentencia siga siendo la unidad básica de ejecución de un programa **A**, han de considerarse los siguientes componentes sintácticos:

- Los programas, formados por un elemento del programa, o un programa seguido de un elemento del programa.
- Los elementos del programa, formados por una sentencia (terminada en un salto de línea), o una sentencia vacía (un salto de línea).
- Los grupos de sentencias, formados por un elemento del programa o un grupo de sentencias seguido de un elemento del programa. Aunque su definición sea equivalente a la de los programas, se diferencian en su uso e interpretación dentro del intérprete.

- Los bloques pasan a ser grupos de sentencias localizados entre los ya definidos delimitadores de bloques. Los bloques siguen siendo sentencias válidas.

Además, se ha realizado un añadido de tipo **funcional**:

- Interpretación mostrando el árbol generado: Esta característica poco o nada tiene que ver con el proceso de interpretación del lenguaje, mucho menos con sus características léxicas, sintácticas o semánticas. Sin embargo, presenta una utilidad didáctica y de detección y corrección de errores importante, por lo que se ha decidido incluirla en la versión final del intérprete.

Para ejecutar esta opción a la hora de realizar el proceso de interpretación de un programa, se debe introducir en la consola de comandos la opción `-t` después de la ruta del programa a interpretar. Esto generará un fichero cuyo prefijo es el nombre del programa interpretado, y cuyo sufijo es `.tree`. Dicho fichero contendrá, en texto plano, una representación “en horizontal” del árbol AST del programa interpretado, tal que los hijos izquierdos de cada nodo se sitúen por encima de sus padres, y, los derechos, por debajo.

2.3.1. Propuestas desechadas

A lo largo del desarrollo del intérprete, se plantearon diversas propuestas de características adicionales, no inicialmente consideradas, que finalmente no llegaron a implementarse. Se detallan a continuación, junto con los motivos por los cuales fueron desechadas.

- Bucles *for*. Al tratarse de una variante de los bucles *while*, ya implementados, se le dio a esta propuesta una prioridad muy baja. Pese a que su implementación es relativamente sencilla (aparentemente es una variación sintáctica de los bucles *while*, generando más nodos en el *AST*), no fueron implementados por falta de tiempo.
- Funciones. Se intentaron plantear como generadores de árboles *AST* adicionales; cada llamada a una función ejecutaría el código asociado a su respectivo *AST*, realizando previamente el paso de argumentos. Finalmente, cuando se planteó el problema de cómo gestionar el ámbito (*scope*) de las variables de distintas funciones, se descartó su implementación por su complejidad.
- Macros sencillas, sin argumentos; tales como la definición de constantes en `C`. Éstas habrían requerido de otro paso adicional de análisis léxico y sintáctico, previo al realizado para interpretar un programa, que no se supo muy bien cómo abordarlo, deviniendo finalmente en el desecho de esta propuesta.

3. Discusión de los ficheros fuente, y justificación de las decisiones de diseño e implementación

Para poder llevar a cabo la implementación del lenguaje *A*, se diferencia, tal y como se dijo en la introducción, la parte léxica en *lex* y la sintáctica en *yacc*. Además, la forma de interpretar el lenguaje pasa por un paso de compilación a código intermedio, implementado con un *AST*; y todo el proceso de interpretación se apoya en el uso de una tabla de símbolos.

Cabe destacar que, al primer fallo detectado, la interpretación de cualquier programa se termina. (La decisión de implementar esta característica es relativamente arbitraria.) El fallo es localizado en el programa mediante el número de línea en el que se ocasionó.

A continuación se listan las distintas componentes del intérprete y sus ficheros fuente asociados, además de las decisiones de diseño destacables, relativas a cada una de ellas, propiamente justificadas. Como nota a tener en cuenta, se ha utilizado como guía de consulta el intérprete sencillo desarrollado por DUŠAN KOLÁŘ², con el que se podrán encontrar muchas similitudes.

Los nombres del tipo *ac* son un acrónimo de *A compiler*.

3.1. Lex - ac.1

En este fichero están definidas las palabras reservadas, así como demás componentes léxicos del lenguaje *A*, y el tipo de escaneado que estos reciben.

El criterio de selección del léxico ha sido altamente influenciado por otros lenguajes de programación, especialmente *C*, aunque también se han tenido en cuenta otros lenguajes de programación no tipados.

Las operaciones aritméticas aceptadas por *A*, definidas en este fichero por sus componentes léxicas, son las básicas para poder trabajar a un nivel de cálculo sencillo (suma, resta, multiplicación, división, cambio de signo), además de alguna operación más exótica, como el módulo o el cociente entero. En cuanto a operadores lógicos, se admiten los operadores de conjunción *and* y disyunción *or*, además de los cuatro comparadores básicos: de igualdad, desigualdad, mayor que y menor que. No se admiten operadores de comparación del tipo “mayor o igual que” ni “menor o igual que” pues sus operaciones vienen implícitas con los ya definidos.

Cuando el escáner detecta un número sin cifras decimales, se interpreta como entero; pero en caso de poseer cifras decimales (es decir, un punto al principio, al final, o en el medio del literal), se interpreta como número real y es tratado

²se ha utilizado la versión presentada como *inter05* en el campus virtual de la asignatura GRAMÁTICAS Y LENGUAJES FORMALES del grado en Ingeniería Informática de la Universidad de Valladolid

correspondientemente. Esta funcionalidad es crucial para realizar las distinciones entre expresiones con valor entero o real mencionadas en anteriores secciones.

Con respecto a los comentarios, hay que destacar unas características importantes:

- Los comentarios de una línea, al finalizar con un salto de línea explícito en su expresión léxica, han de incrementar el contador de líneas del programa, así como devolver un salto de línea como caracter leído, para asegurar el correcto funcionamiento sintáctico del programa.
- Los comentarios multilínea, por su parte, deben hacer algo similar. En este caso, como el rango de líneas que abarcan puede ser una o muchas, y los saltos de línea no forman parte explícita de su expresión léxica, ha de contarse el número de saltos de línea incluidos en uno de estos comentarios para incrementar correctamente el contador de líneas del programa. Así mismo, en caso de que el número de saltos de línea contenidos en el comentario sea de uno o más, también ha de devolverse un salto de línea como caracter leído.

Por tanto, a efectos prácticos, los comentarios de una línea actúan sintácticamente como saltos de línea; y los multilínea lo hacen siempre y cuando abarquen al menos un salto de línea.

3.2. Yacc - ac.y

En esta parte se define todo lo referente a la sintaxis del lenguaje A. Se define la estructura y el tipo de los *tokens* con los que el intérprete trabaja.

Entre dichos *tokens* podemos encontrar los que representan la serie de operaciones con los que A trata, ordenados según sus prioridades y especificando el orden de su asociatividad. Además, hay un operador unario con no asociatividad definida, para poder especificar los casos de cambio de signo.

La sintaxis del programa abarca los siguientes elementos (como podrá comprobarse, acorde a como ha sido descrita en las secciones anteriores):

- Programa (PROGRAM), el cual está compuesto por uno o más elementos de programa.
- Elemento de programa (PROGRAM_ELEMENT), el cual está compuesto por una sentencia o una sentencia vacía.
- Bloque (BLOCK), conformado por un grupo de sentencias delimitado por llaves o bien un salto de línea seguido de un grupo de sentencias delimitado por llaves.

- Grupo de sentencias (`SENTENCE_GROUP`), lo cual abarca uno o más elementos de programa.
- Sentencia (`SENTENCE`), la cual puede ser o bien un bloque o bien la asignación de variables, impresión, lectura, bucles *while* y estructuras *if-else*.
- Expresión (`EXPR`), que van desde expresiones aritméticas, lógicas, operaciones de bit o elementos de expresión mínimos tales que números, caracteres... con los que operar.

De esta manera, al crear un programa, los grupos de sentencias pueden generarse de manera recursiva a través de las producciones de elementos del programa.

La idea implícita en los bloques es que, especificado mediante la etiqueta `g`, se crea un “nuevo” árbol *AST*. De esta manera se trata cada bloque de sentencias como si fuese un programa individual, que se “unen” al programa (*AST*) más general. Mediante esta estructura, el intérprete permite producciones tales que se puedan generar bloques de sentencias agrupadas unas dentro de otras.

Cada uno de estos elementos sintácticos produce la creación de un nuevo nodo, hoja, raíz, o nodo intermedio, en el árbol *AST*, para su posterior tratamiento. Las expresiones terminales que refieren a números o cadenas de caracteres son aquellos que generan hojas en el árbol, y solo los elementos definidos como *Programa* pueden ser raíces de dicho árbol. El resto de sintaxis genera nodos intermedios en el árbol, los cuales irán ramificándose en más nodos, según cada elemento sintáctico vaya derivando otros.

3.3. *AST* - `ast.c` y `ast.h`

El árbol de sintaxis abstracta *AST* es una forma de representación intermedia de código. Consiste en una estructura de tipo árbol en la que se van introduciendo cada una de las órdenes a interpretar como nodos, para su posterior tratamiento (interpretación).

El árbol concreto implementado en este intérprete es de tipo binario, y está diseñado de tal forma que diferencian exclusivamente tres tipos de hojas: de tipo numérico, tanto entero como real, y de tipo *String*.

Cada vez que se genera un nuevo nodo, se genera con la etiqueta correspondiente al elemento introducido en el árbol. En caso de ser un nodo intermedio, se especifican las ramificaciones izquierda y derecha que tendrá. Esto se debe a que los nodos intermedios simbolizan una expresión, sentencia... que actúa sobre otras, las cuales están definidas como sus nodos hijo.

Una vez estructurado el árbol, se van evaluando sus nodos, partiendo de la raíz, y dependiendo del tipo (etiqueta) de estos, se lleva a cabo la operación correspondiente. Los resultados de la evaluación de nodos de expresión se devuelven como *símbolos*

(`symbol`), tipo de datos usado en la tabla de símbolos, y que se explicará en la siguiente sección.

Para las expresiones aritméticas, lógicas y de bits, tan solo se aplica la operación correspondiente a los elementos involucrados. Se incluyen aquí las operaciones trigonométricas y de logaritmo neperiano definidas como funciones predefinidas del lenguaje. Para las evaluaciones de expresiones que tengan que ver con valores numéricos enteros, excepto para la división exacta `/` y el operador cociente `div`, se comprueba el tipo de la hoja izquierda y el de la derecha. En caso de que ambos sean enteros, el resultado del nodo se define como entero. Si alguno de ellos es real, el resultado se define como real. En los casos definidos anteriormente, la división exacta siempre produce resultados reales, y el operador cociente siempre los produce enteros.

Para otros casos, definidos por sus respectivas etiquetas:

- `=` introduce en la tabla de símbolos el hijo izquierdo como nombre de la variable y, como valor, la evaluación de su hijo derecho.
- `PRINT` primero comprueba qué nodos hijos presenta el nodo (son distintos de `NULL`). En el caso de no existir el nodo izquierdo, imprime el valor del nodo de la derecha tras haber sido evaluado. En el caso de no existir el nodo derecho, se imprime la cadena de caracteres que haya en el nodo izquierdo por la salida predeterminada. En caso de que ambos hijos existan, se imprime la cadena de caracteres representada por el hijo izquierdo, seguida del valor evaluado del nodo de la derecha.
- `READ` comprueba la existencia de su hijo izquierdo, significando pues, si ha de mostrar por pantalla algún *String* previamente a disponer la lectura de un valor por entrada estándar. La lectura del valor se realiza a la variable identificada por el valor de su hijo derecho.
- `WHILE` primeramente comprueba que exista un nodo derecho, es decir, una sentencia o bloque de sentencias a ejecutar. En caso de no haberla, simplemente se ejecuta la condición del bucle una y otra vez, cosa que en `A` solo permite crear bucles infinitos. En caso de que sí que exista un hijo izquierdo, es decir, código a ejecutar, este es evaluado y ejecutado hasta que la condición del bucle deje de cumplirse.
- `IF` evalúa la condición encontrada en el nodo izquierdo. De cumplirse, se evalúa el nodo derecho.
- `ELSE`, al solo poder encontrarse dentro de una sentencia `IF-ELSE`, se evalúa la condición encontrada en el correspondiente `IF`, es decir, su hijo izquierdo. De no cumplirse, se evalúa el nodo derecho.

- **g**, etiqueta que corresponde a un grupo de sentencias, causa la evaluación del *AST* cuya raíz es el hijo derecho de este nodo. Dicho *AST* corresponde al conjunto de sentencias agrupadas que dieron lugar al nodo de etiqueta **g**. Los nodos con esta etiqueta, se entiende que no pueden tener hijos derechos.
- **r** es una etiqueta correspondiente a nodos creados por la detección de sentencias en el programa. Los nodos con esta etiqueta no son evaluados. Dada la estructura del árbol, estos nodos simplemente indican si existen más sentencias a evaluar, definiéndose dichas sentencias en sus hijos izquierdos, de existir. El proceso de evaluación de un *AST* consiste en recorrer, mientras los haya, los nodos con esta etiqueta, evaluando para cada uno de ellos su hijo izquierdo (en caso de existir).

3.4. Tabla de símbolos - `syntab.c` y `syntab.h`

En la tabla de símbolos se almacenan las variables, definidas por sus identificadores y sus valores correspondientes. Para su implementación se utiliza una tabla *hash*, pues permite realizar esta estructuración de manera bastante eficiente y directa, accediendo a los elementos de manera más rápida de lo que lo harían otras clases de estructuras, como los árboles binarios de búsqueda.

La tabla de símbolos cuenta con dos operaciones fundamentales: **insertar o editar** una entrada de la tabla, dado el identificador de una variable y su nuevo valor; y **obtener** el valor de una variable, dado su identificador.

A nivel de implementación, la tabla de símbolos almacena *símbolos* (`symbol`), un tipo de datos formado por un identificador de tipo (número real o número entero), y su correspondiente valor. Esta distinción implícita de tipos es necesaria para permitir ciertas operaciones (notablemente, las operaciones de bits). Este tipo de datos es utilizado también dentro del árbol *AST*, a la hora de interpretar los resultados de las expresiones, como se ha indicado con anterioridad.

La tabla de símbolos, al implementarse como una tabla *hash*, dispone de un sistema automático de redimensionamiento en caso de que el número de símbolos almacenados supere cierto valor relativo al tamaño de la tabla. Pese a que el límite inicial establecido de 128 símbolos es bastante generoso, se he decidido implementar este redimensionamiento para simular el funcionamiento de un intérprete más sofisticado.

3.5. Utilidades - `autils.h`

Con motivo de mejorar la organización y presentación del código, se ha elaborado un fichero de cabecera que incluye algunas definiciones y macros utilizadas a lo largo del resto de ficheros fuente. En él se incluyen los identificadores numéricos de los

tokens sintácticos, los códigos devueltos por el programa en caso de error, y una macro para reservar memoria, y comprobar la corrección de dicha reserva.

3.6. Bibliotecas de C utilizadas

A continuación se listan las biblioteca de C que se han incluido en los ficheros fuente del intérprete, explicando detalladamente los motivos para su inclusión.

- **stdio.h**: por motivos obvios, ha de utilizarse la biblioteca estándar para entrada y salida de datos. Esta biblioteca permite, directamente, la lectura y el tratamiento de ficheros, como los programas A a interpretar. Además, permite la impresión de mensajes (como los mensajes de error a la hora de interpretar programas, o las propias impresiones declaradas en los programas) y la lectura de datos introducidos por el usuario.
- **stdlib.h**: al igual que en el caso anterior, la biblioteca estándar de C es necesaria para desarrollar cualquier programa relativamente complejo. Su inclusión permite la utilización de funciones de gran utilidad, como `malloc()` y `free()`, para la gestión de memoria, o `exit()` para la finalización prematura de la interpretación en caso de errores.
- **string.h**: aporta múltiples funciones de utilidad a la hora de trabajar con cadenas de caracteres, como `strlen()`, para obtener la longitud de una cadena de caracteres, `strcmp()`, para comparar cadenas, y `strcat()`, para concatenar varias cadenas en una. Su uso es notable en los ficheros `ast.c` y `syntab.c`.
- **math.h**: permite la utilización de funciones matemáticas comunes, como `sin()`, `cos()`, `tan()`, `ln()` o `fmod()`. Su inclusión es esencial para implementar las funciones predefinidas de A.
- **stdbool.h**: se utiliza por comodidad, y su inclusión no es estrictamente necesaria. El principal motivo por el que se ha incluido es para facilitar la lectura del código fuente en alguna sección concreta.

4. Compilación y ejecución del intérprete

En el directorio `src` del proyecto se pueden encontrar todos los ficheros fuente definidos en la sección anterior. Acompañándolos, también se encuentra un fichero `Makefile` capaz de compilar automáticamente el intérprete, sin ningún esfuerzo.

Cada vez que se desee recompilar el intérprete, para asegurar la corrección del proceso, han de ejecutarse, en el orden indicado, los siguientes comandos de terminal

UNIX en el directorio `src` (el símbolo `$` indica el *prompt* de la terminal, y no ha de ser introducido):

```
$ make clean
$ make
```

Tras la compilación del intérprete, se generará en el directorio un fichero ejecutable `ac` (además de otros ficheros intermedios para la compilación). Este ejecutable es el intérprete en sí, y ejecutarlo indicando la ruta de un programa `A` producirá la interpretación de dicho programa.

El intérprete puede probarse con los ficheros de prueba que se incluyen en el directorio `test`, tal y como se indica en la siguiente sección.

5. Ejemplos desarrollados

En el directorio de `test` se pueden encontrar diversos ficheros con la extensión `.a`, los cuales son interpretables como ficheros en código `A` (pese al uso habitual de esta extensión). Estos ejemplos sirven para demostrar el funcionamiento correcto de cada uno de los aspectos desarrollados en la práctica, así como de algunos incorrectos. En concreto son:

- Cálculo de la media (`media.a`): Este ejemplo trata de simular un programa más o menos satisfecho en código `A`.

En él, se ejecuta una sentencia `while` en la que se va pidiendo en cada iteración un valor para calcular la media de este y los anteriores introducidos, y otro valor “de control”, para saber si se pretende seguir introduciendo números. Los números que se van introduciendo se van sumando en una variable llamada `suma`. Finalmente, cuando el usuario introduce un valor de control distinto de `1`, se deja de cumplir la condición del `while`, no volviendo iterar e imprimiendo el valor medio de todos los valores introducidos por el usuario (la suma de todos los valores introducidos entre la cantidad de valores introducidos).

Ejemplo de uso (salida del programa e inputs del usuario):

```
$ ./ac ../test/media.a
```

```
Programa para calcular la media de varios numeros.
Introduzca el siguiente numero: 1
Introduzca 1 si desea introducir mas numeros: 1
Introduzca el siguiente numero: 2
Introduzca 1 si desea introducir mas numeros: 1
```

```
Introduzca el siguiente numero: 3
Introduzca 1 si desea introducir mas numeros: 1
Introduzca el siguiente numero: 4
Introduzca 1 si desea introducir mas numeros: 0
```

```
La media es 2.5
```

- Ejemplo *while-if-else* (ejemplo-while-if-else.a): En este ejemplo, se lleva a cabo una sentencia *if-else* tal que está construida dentro del bloque de sentencias de un *while*.

En concreto, lo que hace es dar un valor inicial a una variable *x* e ir incrementando su valor en el bucle *while*. A cada pasada imprime “Es par” en caso de que la variable módulo dos sea igual a cero, y “Es impar” en caso contrario.

Ejemplo de uso (salida del programa):

```
$ ./ac ../test/ejemplo-while-if-else.a

Es par 0
Es impar 1
Es par 2
Es impar 3
Es par 4
Es impar 5
Valor final: 6
```

- Ejemplo de operaciones de bit y aritméticas (ejemplo-operaciones.a): En este ejemplo se realizan, primero, múltiples operaciones de bit seguidas de otras tantas aritmética. Dentro del propio programa hay dos comentarios que especifican en qué punto comienzan a definirse cada tipo de operaciones. El primero es un comentario de una única línea, y el segundo multilínea. Además, el programa imprime por pantalla los diversos resultados que va calculando.

Ejemplo de uso (salida del programa):

```
$ ./ac ../test/ejemplo-operaciones.a

Valor de x al principio: 4
Valor de y al principio: 2
Valor de x tras la operacion shift left con y: 16
```



```
Valor de x tras la operacion and con y: 0
Valor de x tras la operacion not: -5
Nuevo valor de x: 40
Nuevo valor de y: 15.63
Valor del x * y: 625.2
Nuevo valor de x = x ^ 3 - (5 + 4) * 15 / 2 = 63932.5
Nuevo valor de x: 45
El modulo de x % 9 es: 0
Coseno (radianes) de x: 0.525322
Seno (radianes) de x: 0.850904
Tangente (radianes) de x: 1.61978
Logaritmo neperiano de x: 3.80666
Cociente de x entre 6: 7
Valor de x tras ser negado tres veces: -45
Valor de x tras aplicar tres veces el operador unario '+': 45
```

- Ejemplo de programa con error de sintaxis (ejemplo-error-sintaxis.a): Este ejemplo contiene una estructura sintáctica errónea, pues se presenta una estructura en la que un `else` está en correspondencia con un `while`, en lugar de con un `if`.

Ejemplo de uso (salida del programa):

```
$ ./ac ../test/ejemplo-error-sintaxis.a

../test/ejemplo-error-sintaxis.a(4): error -- syntax error
Compilacion abortada.
```

- Ejemplo de programa con error léxico (ejemplo-error-lexico.a): Este ejemplo contiene un carácter no esperado, `@`, y por lo tanto produce un error léxico.

Ejemplo de uso (salida del programa):

```
$ ./ac ../test/ejemplo-error-lexico.a

../test/ejemplo-error-lexico.a(5): error -- Caracter
inesperado: @
```

- Ejemplo de programa con error de variable (ejemplo-error-variable.a): Este ejemplo trata de ilustrar un problema común que se da en los lenguajes

de programación en los que no hace falta declarar una variable antes de usarla, como es el caso de A.

En el programa, primero se lee un valor del usuario a una variable `x`, y luego se trata de imprimir, erróneamente, el valor de una variable no inicializada `y`. La primera línea del programa es interpretada correctamente, pero cuando llega el momento de interpretar la segunda, no se puede encontrar la variable especificada en la tabla de símbolos, produciéndose un error de ejecución.

Ejemplo de uso (salida del programa):

```
$ ./ac ../test/ejemplo-error-variable.a

Introduzca un valor: 0
../test/ejemplo-error-variable.a(2) error -- identificador no
encotrado: y
```

6. Problemas conocidos

A continuación se detallan algunos problemas o inconvenientes que el intérprete desarrollado impone sobre los programas en lenguaje A. Se pretende, así, demostrar las limitaciones que se conocen del intérprete, al igual que exponer sus causas o especificar cómo han intentado solventarse.

- Todos los programas escritos en A deben terminar con un salto de línea, pues de otra manera se produce un error de sintaxis. Esto se debe a que, como elementos básicos de un programa, se definen construcciones formadas por sentencias seguidas cada una de un salto de línea, o simplemente saltos de línea.
- Debido también a los saltos de línea, la localización de los delimitadores de bloque es algo sensible. La primera sentencia de un bloque de sentencias no puede localizarse en la misma línea que el delimitador de apertura de dicho bloque. Además, una palabra reservada `else` solo puede localizarse después de un bloque de sentencias si se encuentra en la misma línea que el delimitador de cierre del bloque.
 - También, por la forma en que se han definido las producciones de las sentencias de tipo *if*, *if-else* y *while*, de no ir alguna de ellas seguida por un bloque de sentencias, sino por una única sentencia, dicha sentencia ha de escribirse en la misma línea que el correspondiente `if`, `else` o

while. En el caso de un *if-else* que presente dos únicas sentencias como ejecuciones del **if** y el **else** respectivamente, toda la estructura sintáctica debe escribirse en la misma línea para funcionar correctamente.

- Es conocido que la gramática independiente del contexto definida en el fichero `yacc` es ambigua, produciendo dos conflictos del tipo *shift/reduce*. Uno de estos conflictos es debido a la forma recursiva en que están definidos los bloques de sentencias, y el otro a la admisión de sentencias condicionales **if** con y sin un **else** después. Se han intentado arreglar ambos conflictos, pero, al no encontrarse ningún caso práctico en el que supongan un problema real, y debido a la dificultad que suponía su corrección, finalmente no han sido corregidos de la versión final del intérprete.

7. Apéndice - Sobre la elaboración del proyecto

7.1. Comunicación entre miembros

El proceso que se ha seguido para elaborar esta práctica comenzó en el foro de la asignatura, donde Manuel hizo una propuesta inicial de características a implementar en el intérprete. Allí mismo, debatimos qué nos parecía mejor y peor implementar, y finalmente llegamos a un consenso entre todos. Tras esto, pedimos al profesor de la asignatura que nos orientara respecto al nivel de dificultad de la práctica. Tras darnos su aprobación, nos pusimos manos a la obra.

Inicialmente teníamos pensado hablar todo lo relativo al proyecto por el foro de la asignatura, pero éste no transmite los mensajes a los integrantes del grupo de manera inmediata; por tanto, para tratar temas cruciales de manera rápida, optamos por una aplicación de mensajería instantánea (Telegram). Además, hemos utilizado DISCORD para hacer llamadas de voz y poder compartir pantalla, trabajando así todos sobre el mismo código, pero cada uno desde su casa. Durante los últimos días de desarrollo del proyecto, dado el trabajo casi continuo de todos los miembros del grupo, las comunicaciones se realizaron exclusivamente por alguno de los dos últimos medios mencionados.

7.2. Plataformas y herramientas utilizadas

Para la organización y almacenamiento del código fuente del intérprete, hemos utilizado un repositorio *git* almacenado en el *GitLab* de la escuela de Ingeniería Informática de la Universidad de Valladolid³.

³<https://gitlab.inf.uva.es/marruiz/glf-proyecto/>.

Para la realización de este documento, hemos utilizado la plataforma de edición de documentos \LaTeX *online Overleaf*. Esta plataforma permite trabajar a todos los integrantes del grupo simultáneamente en el mismo documento, que se almacena automáticamente en la nube, para poder ser descargado en cualquier momento. Debido a la naturaleza inherentemente *online* de la plataforma, y por comodidad, los ficheros fuentes de este documento no se han incluido en el repositorio *git* mencionado anteriormente hasta finalizada su elaboración.

La presentación sobre el proyecto, con la que se ha realizado el vídeo grupal de resumen y defensa del mismo, la hemos realizado mediante *Power Point*.

El vídeo lo hemos realizado en tres partes separadas, repartiendo los temas a tratar entre cada uno de los integrantes del grupo, y juntando finalmente las partes en el vídeo final. Para la grabación del vídeo, se han utilizado tres aplicaciones: María utilizó *Kaltura*, ya que es la aplicación que recomienda la Universidad de Valladolid, Andrés utilizó *ScreenCast-o-Matic* por compatibilidad y comodidad, y Manuel utilizó *OBS*, debido a su experiencia previa con dicho software. Para el montaje del vídeo final, utilizamos *Adobe Premiere Pro CC*. Para el almacenamiento del vídeo, se ha utilizado *OneDrive*⁴, como recomendó el profesor de la asignatura.

7.3. Reparto de trabajo

Inicialmente, se realizó un análisis de tareas por hacer, para lo cual Manuel estudió y comentó el código del intérprete sencillo de DUŠAN KOLÁŘ, y lo compartió por el foro del grupo en el campus virtual de la asignatura. Las primeras características léxicas y sintácticas del intérprete, que fueron las más comunes con las del intérprete sencillo del señor KOLÁŘ, se realizaron, pues, en conjunto por todos los integrantes del grupo, a través de una llamada de DISCORD.

Después, con la base del intérprete ya asentada, se realizó el análisis de tareas “extra” o más complejas por hacer, y cada integrante eligió una sobre la que trabajar: Manuel escogió la tabla de símbolos, María lo relativo a los bucles *while* y los bloques de sentencias, y Andrés lo relativo a las sentencias condicionales *if*. Conforme se fue avanzando en el desarrollo de cada una de estas partes, los ámbitos de trabajo de cada integrante se ampliaron, por lo que se retomó el trabajo en conjunto mediante llamada de DISCORD. A excepción de los ficheros fuente relativos a la tabla de símbolos, que fueron elaborados exclusivamente por Manuel, todos los integrantes trabajaron en todos los ficheros fuente del intérprete.

Tras la elaboración de todas las partes del intérprete, Manuel se encargó de la corrección de *bugs* (errores), así como del remate de las partes concernientes a los bloques de sentencias y estructuras *if-else*; eso sí, ayudado en todo momento por

⁴https://alumnosuvaes-my.sharepoint.com/:v:/g/personal/manuel_castro_alumnos_uva_es/Ef2zgArePRZIrCm42Lg7SZIB3qHNNHzod0nJsVELh0n1vRQ

sus compañeros (ya que, se insiste, las últimas etapas del trabajo se desarrollaron en llamada conjunta). Durante el proceso de *debugging*, y para facilitar dicho proceso, Manuel implementó la impresión opcional de los árboles *AST* de los programas interpretados. Como añadido final, también implementó las operaciones de bits, y las distinciones implícitas entre tipos numéricos enteros y numéricos reales necesarias para el correcto funcionamiento de estas operaciones.

Por su parte, María y Andrés se dedicaron a ayudar durante el proceso de *debugging* y puesta a punto del intérprete, investigando en internet cómo se podrían implementar ciertas características que estaban dando problemas. Además, ambos se encargaron de la elaboración conjunta de este documento. Andrés también se encargó de la elaboración de la presentación *Power Point* del trabajo, y María de la elaboración de los ejemplos de prueba del intérprete (ayudada en algún caso concreto por Manuel).

Finalmente, se acordó cómo se iba a repartir la video-defensa del grupo, y cada uno grabó su parte correspondiente. Manuel fue el encargado de montar el vídeo final, así como de dar una última revisión y puesta a punto a este documento.

NOTA: Aunque el registro de *commits* del repositorio *git* del proyecto puede utilizarse como guía para determinar el trabajo que ha realizado cada integrante del grupo, no debe interpretarse muy estrictamente. Como ya se ha dicho, en la mayoría de casos el trabajo fue realizado de forma conjunta, mediante el sistema de llamadas de DISCORD. Aunque un integrante concreto fuese el encargado de crear y *pushear* un *commit*, el código modificado en el mismo muy probablemente fuese el resultado del trabajo de todos los integrantes.

8. Referencias bibliográficas

- [AA] Jeffrey Ullman Monica S. Lam Alfred Aho, Ravi Sethi. Compilers: Principles, techniques, and tools. [http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Compiler%20Design/Alfred%20V.%20Aho,%20Monica%20S.%20Lam,%20Ravi%20Sethi,%20Jeffrey%20D.%20Ullman-Compilers%20-%20Principles,%20Techniques,%20and%20Tools-Pearson-Addison%20Wesley%20\(2006\).pdf](http://ce.sharif.edu/courses/94-95/1/ce414-2/resources/root/Text%20Books/Compiler%20Design/Alfred%20V.%20Aho,%20Monica%20S.%20Lam,%20Ravi%20Sethi,%20Jeffrey%20D.%20Ullman-Compilers%20-%20Principles,%20Techniques,%20and%20Tools-Pearson-Addison%20Wesley%20(2006).pdf).
- [Aab] Pamela Aaby. Compiler construction using flex and bison. <http://pegaso.ls.fi.upm.es/~jfuertes/Software/compiler.pdf>.
- [Com] GeeksforGeeks Community. Print binary tree in 2-dimensions. <https://www.geeksforgeeks.org/print-binary-tree-2-dimensions/>.
- [Kola] Dusan Kolar. Dusan-notes. <https://campusvirtual.uva.es/pluginfile>.

`php/1156983/mod_folder/content/0/Dusan-Notes.pdf?forcedownload=1.`

[Kolb] Dusan Kolar. Dusan-slides. [https://campusvirtual.uva.es/pluginfile.php/1156983/mod_folder/content/0/Dusan-Slides.pdf?forcedownload=1.](https://campusvirtual.uva.es/pluginfile.php/1156983/mod_folder/content/0/Dusan-Slides.pdf?forcedownload=1)

[Clib] Bibliotecas de C:

- *Standard input-output header* (`stdio.h`)
- *Standard library header* (`stdlib.h`)
- *Standard String library header* (`string.h`)
- *Standard Math library header* (`math.h`)
- *Standard Boolean type and values header* (`stdbool.h`)