Un estudio sobre modelos clásicos de cómputo

R. Córdoba García

Trabajo fin de grado EHU-UPV

Directora: Montserrat Hermo

A Julio Córdoba y Rosa Mª García, mis padres.

§1. Sobre el trabajo

El presente trabajo trata sobre la equivalencia de distintos tipos de modelos de cómputo; más concretamente, se centra en demostrar que la potencia de cómputo de los lenguajes actuales, las máquinas de Turing y las funciones recursivas es igual; esto es, que cualquier problema que se puede resolver en uno de estos modelos se puede resolver también en los otros. En toda la explicación se da por hecho que se conoce la teoría sobre las máquinas de Turing tal y como las definió originalmente su creador en [1]* y las reglas para definir funciones recursivas según están presentadas por S. C. Kleene en [2].

Para dar una primera aproximación del contenido del trabajo diremos que las demostraciones están en los apéndices al final del documento. Algunas partes de los capítulos precedentes, sobre todo el §4, ayudan a entender estas demostraciones. Todo lo demás, incluyendo esta introducción, está de relleno.

1.1. Justificación y explicación

Las máquinas de Turing y el λ-cálculo inventado por Alonzo Church, así como la teoría sobre las funciones recursivas, nacieron en el siglo XX en el marco de los estudios de lógica y el límite de lo resoluble. La tesis Church-Turing nos dice que cualquier problema para el que exista un procedimiento que lo solucione puede resolverse con una máquina de Turing o usando λ-cálculo.

La tesis no se puede demostrar pero tampoco ha sido refutada, con lo que sabemos que no hay método conocido más potente que estos dos para expresar soluciones a problemas. Cuando un lenguaje tiene la misma capacidad que ellos para definir procedimientos de resolución se dice que es Turing-completo; esto es, que puede definirse con él una solución para cualquier problema que pueda resolver una máquina de Turing. Sin embargo a la mayoria de personas que empieza a estudiar computación la duda sobre la capacidad de cómputo le surge en sentido contrario: ¿cómo es posible con un método tan simple como una máquina de Turing resolver lo que resuelve un ordenador actual?

Cuando leemos un tratado sobre una pintura conocida, o una demostración del teorema de Pitágoras posiblemente sea por lo interesante que resulta admirar desde otros puntos de vista obras e ideas que nos gustan, y en todo caso nos da una excusa para reflexionar sobre ellas. El hecho de que existan tantas obras sobre un mismo tema y docenas de demostraciones distintas del teorema de Pitágoras es buena evidencia. De modo parecido, este trabajo no contiene algo novedoso ni especialmente interesante; es un acercamiento más, desde la admiración, a algunas ideas fundamentales de la ciencia.

---

* Las indicaciones [x] hacen referencia a libros citados en la bibliografía.

1.2. Lenguaje moderno

Las demostraciones de la equivalencia con un lenguaje de programación de los utilizados actualmente se hará usando uno concreto. Se ha elegido el lenguaje C entre otras razones porque (1) es un lenguaje, al igual que el entorno del que surgió, de gran importancia histórica. Creado por Dennis Ritchie en los años 70 para ayudar a desarrollar herramientas para UNIX y el propio sistema operativo, ha influenciado desarrollos posteriores, representando pues una pieza importante en el camino que va desde ideas fundacionales de las que trata este trabajo hasta la informática del siglo XXI; (2) la sencillez de su sintaxis y semántica facilitan mucho las demostraciones; (3) se ha usado y se usa para crear programas de los tipos más variados, desde bases de datos hasta sistemas de inteligencia artificial o simuladores 3D. Esto hace ver claramente el alcance de cualquier sistema que pueda computar lo que computa un programa en este lenguaje; por ejemplo, quedará claro que si los ordenadores pueden llegar a someter a la humanidad entonces las máquinas de Turing pueden dominar el mundo.

1.3. Estructura del trabajo

La mayoría de demostraciones sobre sistemas de cómputo que se pueden encontrar se basan en conceptos teóricos. Las demostraciones dadas aquí son más pedestres por así decirlo; se saca los modelos de computación presentados de su contexto teórico presentando ejemplos prácticos completos.

Se empezará con la motivación primera del trabajo, que es la demostración de que para cualquier programa, por complejo que sea, que se escriba en un lenguaje de programación de los usados hoy, existe una máquina de Turing equivalente; es decir, que resuelve el mismo problema. Como se ha dicho anteriormente, concretamente se pretende demostrar que dado un programa en C se puede encontrar una máquina de Turing equivalente. Una manera sería encontrar un método de traducir las instrucciones del programa C en la especificación de la máquina de Turing, algo así como un compilador de C a máquina de Turing. Esto es muy complicado, y desde luego fuera de mi alcance. Otro enfoque sería usar como dato para la máquina cualquier programa en C dado, escribiéndolo en la cinta, y crear una máquina de Turing universal ejecutora de programas C. Esto parece aún más difícil, como crear una máquina que compile y luego ejecute los programas. Basado en esta idea, más sencillo parece crear una máquina que ejecute programas escritos en código ensamblador o código máquina resultante de compilar el programa en C, lo que bastaría para demostrar lo que se propone. Sigue siendo muy difícil crear una máquina así. El camino que se tomará estará basado en esta última idea: crearemos un lenguaje de programación y una máquina que ejecute programas en este lenguaje escritos en su cinta. Ésta no es, sin embargo,

la solución que se busca, ya que el programa en C computa a partir de unos datos de entrada y la máquina que se propone toma como entrada el programa entero; no obstante, una vez mostrada esta máquina, demostrar que hay otra que da la misma solución a partir de los mismos datos que recibe el programa en C será fácil.

La dificultad estará pues en demostrar que este lenguaje que crearemos es equivalente, en el mismo sentido que se ha explicado antes, al lenguaje C. Se creará primero el lenguaje, teniendo en cuenta que los programas escritos en él deben ser leídos por una máquina de Turing y debe ser sencilla la demostración de la equivalencia con C. Luego se presentará la demostración, que ocupará la mayor parte del trabajo. A este nuevo lenguaje, que será muy simple y equivalente a C en cuanto a potencia computacional le llamaremos C--.

La demostración sobre la equivalencia de las funciones recursivas también usará este lenguaje. Primero especificaremos una forma de codificar, basada en la codificación de Gödel, con la que transformar cualquier programa escrito en C-- como un número natural. El grueso de la demostración consiste en presentar † una función recursiva escrita en C que toma como argumento la codificación de (vuélvase a † si se quiere) un programa no recursivo escrito en C--, que como se habrá demostrado es equivalente a alguna máquina de Turing. Esta función recursiva devuelve un número natural que será la codificación del programa usado como entrada tal y como quedaría después de ejecutarlo; es decir, la función recursiva 'imita' la forma en que actuaría un ejecutor de programas C--.

Por último se demuestra que hay algún programa en C, y por tanto en C--, equivalente a cualquier máquina de Turing. Esto se hace mostrando un ejemplo concreto e indicando como se puede modificar el programa del ejemplo para hacerlo equivalente a otras máquinas.

El modo en el que se construyen las funciones recursivas y la simulitud en la sintaxis de C para escribir funciones y la usada por Kleene, que es la que se usa en este trabajo, hace innecesaria la desmostración de que cualquier función recursiva puede escribirse como un programa en C. Por otra parte, una función recursiva en C puede escribirse como un programa en C--, para el que existe una máquina de Turing equivalente, para la que a su vez existe un programa en C equivalente. Esto será una demostración indirecta de que cualquier programa recursivo puede escribirse como uno no recursivo, ya que el programa en C equivalente a las máquinas de Turing que se construye en la demostración es no recursivo.

§2. Algoritmos y programas

La definición más general de algoritmo podría ser la de un conjunto de instrucciones que llevan a la solución de algún problema. Según esta definición la consideración de algoritmo para un grupo de instrucciones dependería del sujeto que las interprete. Por un lado, un algoritmo puede presentarse de muchas maneras. Ejemplos de algoritmo, con datos y acciones indicadas más o menos explícitamente, podrían ser una serie de señalizaciones e indicaciones en un mapa si el problema es llegar de un punto a otro, una partitura musical si lo que se quiere resolver es como interpretar determinada pieza musical, o las instrucciones para hallar el máximo común divisor de dos número derivadas de la presentación de Euclides en los *Elementos*, probablemente el ganador si se hiciese una encuesta sobre el primer algoritmo que le viene a la cabeza a un matemático cuando se le pide un ejemplo. Por otra parte, aun suponiendo que dos sujetos saben interpretar las instrucciones y de la misma manera, lo que pueden llevar a uno a la solución de un problema no tiene por qué ser útil para otro. El algoritmo más adecuado dependerá del problema y de quién lo interprete. Aquí no nos interesa encontrar los mejores algoritmos o los más eficientes (de hecho serán terribles), sino atenernos al concepto de algoritmo tal y como se entiende en matemáticas; así, lo que trataremos es de encontrar una forma de definir algoritmos lo más libre posible de ambigüedades, tanto en la manera de definirlos como en el modo de interpretarlos, presuponiendo lo mínimo sobre las capacidades del sujeto que vaya a ejecutar el algoritmo, al que llamaremos de aquí en adelante *el ejecutor*. Concretamente queremos escribir instrucciones que una vez ejecutadas den como solución unos datos de salida a partir de unos datos de entrada.

2.1. Acciones de las intrucciones

Imaginemos que el maestro en un taller tiene unas intrucciones para llevar a cabo determinado proceso. Si en algún momento el maestro quiere delegar la tarea en un aprendiz quizás estas instrucciones no sean suficientes para conseguir llevar a buen término la tarea; el maestro tendrá que suplementar las instrucciones con otras y expandir algunas de las instrucciones por otras más sencillas que en conjunto expresen lo mismo. Si el aprendiz quiere que un alumno recién llegado al taller se encargue de la tarea tendrá probablemente que hacer lo mismo y preparar otro conjunto de instrucciones más amplio que pueda ejecutar el alumno. La utilidad de un algoritmo no recae en que pueda explicar el problema sino en que permita encontrar la solución a un problema de manera sistemática. Lo que nos interesa es seguir el proceso anterior de simplificar el conjunto de intrucciones para lograr que pueda entenderlo el mayor número de

personas, de tal modo que se puedan ejecutar las intrucciones sin necesidad de saber siquiera cuál es el problema que se está resolviendo.

El reconocimiento de la utilidad de abstraer el problema de este modo está reflejado en el propio origen de la palabra *algoritmo*, derivado del título latino del libro de Mohammed Ben Musa donde explica cómo operar con los símbolos indoarábigos para los números, con lo que se podían resolver problemas numéricos de manera sistemática. Otro ejemplo de abstracción usando símbolos para representar un problema y resolverlo sistemáticamente es el propio álgebra (cuyo nombre proviene de otro libro del mismo autor). Esta misma idea de usar operaciones que se pueden llevar a cabo 'sin pensar' está detrás de los intentos entre los siglos XIX y XX de obtener todos los teoremas a partir de axiomas mediante reglas de inferencia y está expresada de manera genial por Turing cuando postula su modelo en el que es una máquina la que ejecuta las instrucciones.

Considerando todo esto, todas las instrucciones que usaremos para definir algoritmos consistirán en acciones que hagan referencia a datos. Estas acciones serán simples y generales, sin relación con un problema concreto, con lo que se podrá usar el mismo conjunto de acciones para dar solución a distintos tipos de problemas.

2.2. Datos

Para hacer referencia a distintos tipos de objetos de una forma no ambigua en las instrucciones y que se puedan interpretar en el resultado, necesitamos poder distinguir claramente unos y otros, identificarlos univocamente; así, los conjuntos de objetos con los que trabajemos se podrán enumerar. Una manera sencilla de escribir algoritmos para problemas con cualquier tipo de objeto con esta característica será usando los números naturales como único tipo de dato. De este modo se puede establecer una función biyectiva entre los naturales y los objetos con los que trabajemos; será el escritor del algoritmo y el sujeto que haga uso de él quienes interpreten los datos según la finalidad del algoritmo y el problema que aborde. Con esto, el ejecutor no tiene necesidad de saber el significado de los datos que maneja, simplificando su labor. Un ejemplo de cómo usar números naturales para referirse a objetos de un conjunto enumerable se verá más adelante, cuando se muestre cómo se escriben programas que manejen números racionales en general.

Sin embargo, para el ejecutor simple que suponemos es demasiado requerirle conocer el concepto de número. Vamos a facilitar su labor, abstrayendo aún más el modo de resolver los problemas desde el punto de vista del escritor del programa y haciéndolo más concreto a ojos del ejecutor.

2.3. El modelo de cómputo

Los algoritmos que necesitamos definir tienen que poder expresarse gráficamente para que sean útiles para las demostraciones que se pretenden en este trabajo; es decir, tenemos que poder escribir programas (de *pro*: con anterioridad y *gramma*: escrito) que contengan las instrucciones. Vamos a ver, tomando todo lo dicho anteriormente en cuenta, cómo presentar estos programas para poder ser ejecutados.

Postulamos que se tiene una superficie sobre la que escribir tan extensa como haga falta y contamos también con un instrumento para escribir. Las instrucciones se colocarán en posiciones consecutivas como se explica a continuación, y todas las instrucciones especificarán una acción y una posición que tiene significado para la acción. Esto hace que en lo que respecta al ejecutor del programa los únicos datos con los que tiene que trabajar sean las posiciones del propio programa.

Vamos a suponer que las instrucciones se escriben en sentido vertical, desde arriba hacia abajo. Para marcar el inicio del programa dibujamos una raya horizontal y consideramos lo que está debajo de esta línea la primera posición del programa. Para hacer una nueva posición dibujamos en una posición ya definida una raya paralela a la raya de inicio; lo que está debajo de esta raya será la siguiente posición de la posición sobre la que se ha dibujado la raya. Por ejemplo, si se dibuja una raya debajo de la posición de inicio tendremos la primera posición sobre esta raya y la siguiente posición a la primera debajo. Si dibujamos una raya en la siguiente posición de la primera tendremos la primera posición, la siguiente de la primera y la siguiente de la siguiente. De este modo podemos definir las posiciones que queramos. El ejecutor del programa tendrá que leer las instrucciones escritas en las posiciones y actuar acorde a su significado, que se explica en la siguiente sección.

§3. Un lenguaje sencillo

Una vez estudiadas las características deseables en los programas que necesitamos, vamos a ver cómo escribirlos. Para ello presentaremos una gramática que defina los símbolos que se usarán y cómo combinarlos para expresar datos y acciones en forma de instrucciones, así como la semántica de éstas; es decir, definiremos un lenguaje de programación, con la misma filosofía de búsqueda de sencillez. Éste será el lenguaje `C--` que se usará en las demostraciones.

3.1. Los datos

Como se ha indicado anteriormente, sólo será necesario representar los números naturales. Tomando el orden habitual de los números naturales (1, 2, 3, ...), la representación se hará de la siguiente manera:

- El número 1 se representará con el símbolo «`1`».
- El siguiente número de un número n se representará añadiendo un símbolo «`'`» a la representación de n.

Por ejemplo, el número 2 se representaría como «`1'`» y el número 5 como «`1''''`».

3.2. Las instrucciones

Cada instrucción va a indicar una acción y una posición que tendrá significado para dicha acción. A cada acción le pondremos un nombre, que usaremos también para referirnos a una instrucción que indique una acción de este tipo. Como hemos visto, las posiciones del programa forman un conjunto enumerable en el que se considera una posición como la primera y todas las posiciones menos la primera son la siguiente de alguna posición. Por esto, la misma sintaxis usada para representar los datos nos va a servir para especificar instrucciones, lo que ayuda a simplificar el lenguaje. Así, las instrucciones estarán formadas por un símbolo inicial y por símbolos «`'`», a los que llamaremos *marcas*. El símbolo inicial, al que llamaremos *símbolo principal* de la instrucción, especificará la acción. A continuación del símbolo inicial, las marcas indicarán la posición que compete a la acción; a esta posición le llamaremos la *posición referida* por la instrucción. Se indica la posición referida como sigue:

- Si sólo hay un símbolo (el principal, no hay marcas), la posición referida será la primera.
- Si a una instrucción se le añade una marca, la posición referida será la siguiente de la posición referida por la instrucción antes de añadirle la marca.

Vamos a ver las intrucciones necesarias para el lenguaje que necesitamos.

3.2.1. Especificar datos

Para poder trabajar con los datos y escribir el resultado de la ejecución del programa se necesita alguna instrucción que escriba en las posiciones; concretamente, según se representan los datos, se requiere escribir los símbolos «1» y «'». El símbolo «'» no tiene sentido por sí solo en una posición, por lo que no hay ambigüedad si se usa una misma acción para escribir ambos símbolos: si en la posición referida no hay nada se escribe «1»; se escribe una marca si la posición referida no está vacía, es decir, si hay un símbolo principal, posiblemente con marcas. Después de ejecutar esta intrucción se pasa a la siguiente posición para seguir desde allí la ejecución del programa.

El ejecutor del programa sólo ve intrucciones, no se ve afectado por los datos. Será el escritor del programa quien defina dónde guarda los datos y los interprete. Por tanto, no hay tampoco ambigüedad si se usa para esta instrucción el mismo símbolo principal que se usa para representar las instrucciones. Usaremos, pues, el símbolo «1» como símbolo principal.

Desde el punto de vista del escritor del programa esta instrucción podría llamarse *siguiente*, ya que se puede usar para para representar el siguiente objeto según la ordenación que se haya establecido. Sin embargo, vamos a tomar el punto de vista del ejecutor del programa, que es quien motiva el modo de construir el lenguaje. Aunque la instrucción puede escribir dos símbolos diferentes, será más común que escriba marcas, por lo que llamaremos a esta instrucción *marcar*.

Nada de los dicho implica que la instrucción sólo pueda escribir marcas en posiciones que tengan «1» como símbolo principal. Las instrucciones también son un conjunto enumerable y puede considerarse que cuando se pone una marca se escribe la siguiente instrucción de la que había.

3.2.2. Decisiones

Si solamente se dispusiera de instrucciones para escribir datos los programas serían autómatas que se limitarían a realizar patrones fijos, cada programa un patrón. Lo que nos interesa es escribir programas que se comporten diferente según los datos iniciales que se le presenten. Para ello necesitamos alguna instrucción que lleve a ejecutar unas acciones u otras según se cumplan determinadas condiciones relacionadas con los datos. Estas acciones, por supuesto, estarán escritas en `C--`. Dividiremos el problema y crearemos dos tipos de intrucciones simples.

Una, a la que llamaremos *saltar* y para la que usaremos como símbolo principal «∗», indica al ejecutor que vaya a la posición referida y siga la ejecución del programa desde esta posición.

A la otra instrucción le llamaremos *comparar* y para ella usaremos «=» como símbolo principal. Al encontrar esta instrucción, el ejecutor comparará dos posiciones para ver si son iguales. Decimos que dos posiciones son iguales si y sólo si:

- Están las dos vacías; esto es, no hay símbolos en ninguna de ellas.
- En cada una de las posiciones hay un símbolo principal, y bien no hay marcas o hay la misma cantidad de marcas.

Por ejemplo, una posición con «1» es igual que una con «∗», y una con «=″» igual a una con «∗″»; dos posiciones vacías también son iguales, pero una con «1″» no es igual que una con «1‴».

Para mantener la sintaxis de las demás instrucciones, en la que sólo se especifica una posición, en las instrucciones *comparar* la primera posición estará implícitamente especificada; así, se comparará la primera posición con la posición referida por la instrución. Si las dos posiciones no son iguales el ejecutor irá a la siguiente posición y seguirá ejecutando el programa desde allí; en caso contrario, es decir, si las dos posiciones son iguales, el ejecutor irá a la siguiente de la siguiente posición y seguirá desde allí la ejecución (saltará una posición).

3.2.3. Reutilizar posiciones

Usando la instrucción *marcar* las veces necesarias se puede hacer referencia a cualquier natural. El usar en *comparar* la primera posición como fija hace ver que es necesario poder hacer referencia a naturales anteriores. Para esto utilizaremos la instrucción *borrar*, que usará el símbolo «0». Esta instrucción indica al ejecutor borrar todas las marcas de la posición referida y el símbolo principal solamente si éste es «1» (si es «∗», «=» o «0» sólo borra las marcas en la posición referida, si las hubiera). Después de ejecutar esta intrucción se pasa a la siguiente posición para seguir desde allí la ejecución del programa.

Esta asimetría del resultado de la acción según lo que haya en la posición referida puede ilustrarse si imaginamos que el símbolo «1» está escrito con el mismo instrumento con el que se escriben las marcas, por ejemplo una tiza, y el resto de símbolos principales están grabados o escritos con un marcador indeleble; cuando se pasa el borrador por una posición referida, se borra la tiza y el resto queda.

3.3. El lenguaje en su entorno

Con los cuatro tipos de instrucciones presentadas será suficiente para tener una lenguaje como el que necesitamos. En el apéndice A se especifica el lenguaje más formalmente.

Para ejecutar un programa, se empezará ejecutando la instrucción escrita en la primera posición y se seguirá según la instrucción que haya en ella como se ha explicado anteriormente (si la intrucción es *marcar* o *borrar* se sigue desde la siguiente posición, si es *comparar* según el resultado de la comparación y si es *saltar* desde la posición referida). Si en cualquier momento de la ejecución la posición en la que se busca la instrucción a ejecutar contiene cualquier cosa que no sea una instrucción formada según las reglas sintácticas descritas—por ejemplo, si la posición está vacía—, la ejecución se detiene.

Tal y como está definido el lenguaje se puede intuir que será fácil encontrar una máquina de Turing para ejecutar programas de este tipo. Lo único si no complicado sí laborioso que contiene este trabajo será la demostración de que este lenguaje es equivalente a C. Eso se hará en la siguiente sección.

§4. Equivalencia de C y C--

La demostración de la equivalencia de los dos lenguajes consistirá en mostrar cómo se puede transformar un programa en C en uno en C-- que dé el mismo resultado a partir de los mismos datos. Para explicar cómo se hace la transformación vamos a introducir un nuevo actor, al que llamaremos el *traductor*, que será quien transforme un programa en otro. Análogamente a los requerimientos que se exigían al *ejecutor* de los programas C--, se ha buscado un método para traducir programas presuponiendo el mínimo conocimiento al *traductor* para que pueda llevar a cabo su trabajo.

El modo de transformar los programas está basado en considerar los programas en C como un conjunto de macroinstrucciones de C--, es decir, cadenas de símbolos que pueden ser convertidas de una manera prefijada en otras cadenas hasta llegar a un programa C--. Casi toda la demostración, y la mayor parte de este documento, consiste en presentar las reglas de transformación de unas cadenas en otras. A partir de estas reglas y de unas pocas directrices simples el *traductor* será capaz de transformar un programa en C en uno en C-- de manera 'automática'. No todos los programas en C podrán ser traducidos con el método presentado—la sintaxis completa de C no está contemplada en las reglas—pero el subconjunto de los programas que pueden serlo es tal que es fácil ver que cualquier programa puede adaptarse a una forma que permita traducirlo.

4.1. Sobre la demostración

Todas las reglas e indicaciones para transformar programas C a otros equivalentes en C-- aparecen en el apéndice E, presentadas como una ayuda para escribir programas C-- usando macroinstrucciones. Se empieza por macros para definir operaciones básicas y se presentan posteriormente macros que usan las anteriores como texto de sustitución; el objetivo es llegar a macroinstrucciones que se asemejen a construcciones de C, de modo que un programa en C-- se pueda escribir mediante macros como si fuese en C; así, el trabajo del *traductor* sería ir en sentido contrario, y desde construcciones de C ir sustituyendo hasta llegar a un programa C-- equivalente. Al igual que a la hora de ejecutar un programa C--, la traducción de programas en C se hace manejando solamente símbolos, sin necesidad de saber nada acerca del programa o siquiera del lenguaje.

En la demostración se usan distintos recursos, que van apareciendo según sea necesario para que se entienda el modo de llevar a cabo la transformación de programas. Se pueden distinguir:

- Las *macroinstrucciones* del lenguaje C--. Indican cómo se sustituyen unas cadenas de símbolos por otras. Tienen la forma *cadena1* → *cadena2*, siendo *cadena1*

la macroinstrucción y *cadena2* la cadena por la que se sustituye. Por ejemplo, `INIT` ↠ `1` indica que a la hora de escribir el programa, `INIT` es otro nombre para la instrucción `1`; el traductor sustituirá, cuando haga la transformación, todas las apariciones de `INIT` por `1` a lo largo del texto del programa.

- *Reglas gramaticales*. Se usa notación basada en la forma de Backus–Naur. Los no-terminales se escriben en cursiva, *así*, o en cursiva entre paréntesis angulares, ⟨*así*⟩; los terminales se escriben en negrita, **así**; la cabecera de las producciones se separa del cuerpo con un símbolo «→»; las alternativas en el cuerpo de las producciones se separan con «|».

  Las reglas gramaticales no se refieren a construcciones de ningún lenguaje; su propósito es simplificar las reglas de traducción que aparecen posteriormente. Cuando un no-terminal aparece en una macro, la regla donde aparece la macro representa todas las reglas de traducción resultantes de sustituir el no-terminal en la macro y en el texto de sustitución por terminales, de acuerdo a las normas de las reglas gramaticales. Por ejemplo, con la producción $x \to \mathbf{0} \mid \mathbf{1} \mid \mathbf{=} \mid *$ en la gramática, la regla basada en macros «$x^1 \twoheadrightarrow x$» representa las cuatro reglas:

  $1^1 \twoheadrightarrow 1$; $0^1 \twoheadrightarrow 0$; $=^1 \twoheadrightarrow =$; $*^1 \twoheadrightarrow *$.

- *Indicaciones para el traductor*. En las sustituciones de algunas macros, y en traducción del programa en general, el traductor debe efectuar algunos procesos sencillos, más allá de simples traducciones de símbolos. Una serie de indicaciones y comentarios a lo largo de la explicación informan de cómo hacerlo.

- *Postulados*. Para hacer la traducción de programas, igual que para ejecutar `C--`, sólo hay que manejar símbolos. Algunos de estos símbolos son los dígitos que se usan habitualmente para representar los números naturales en notación indoarábiga. Los postulados indican cómo se consideran y usan en las macros.

4.2. Apuntes sobre la demostración

La manera de presentar la demostración por medio de macros ayuda a entender el lenguaje `C--` y cómo a partir de las cuatro instrucciones de que consta se pueden ir contruyendo programas para calcular operaciones conocidas. Por otra parte, el hecho de incluir los números racionales como tipo de dato que se pueden usar en los programas que se van a traducir complica y alarga mucho la demostración. No obstante, éste es sólo un caso particular de objetos que se pueden codificar; con entender cómo se escriben programas que usen números naturales como dato es suficiente para comprender cómo construir programas en `C--` y que el lenguaje es Turing-completo. Para esto no es necesario entender todas las macros, vale con entender cómo se traducen algunas, básicamente

las primeras. No debería ser difícil seguir las traducciones del apéndice E. Para no alargar la explicación, aquí sólamente se comentará de forma breve algunas partes, y sólo en lo que se refiere a programas que usan números naturales como único tipo de dato.

Aunque en las reglas de traducción de los programas no se haga referencia a números, sí se usarán números en las explicaciones de esta sección. De aquí en adelante se considerarán como definidas dos funciones biyectivas: una función f de naturales a cadenas de dígitos tal que: $f(1)$ = «1» y para todo natural n, si $f(n) = \beta$ entonces $f(n + 1) = \alpha$, donde α y β representan cadenas de dígitos y $\alpha = \beta$ + «1» es verdad según los postulados del apéndice E; y una función g de naturales a posiciones del programa C-- tal que: $g(1)$ = *primera posición* y $g(n + 1)$ es la siguiente de la posición $g(n)$. Por ejemplo $f(5)$ =«5», $f(13)$ =«13» y $g(3)$ es la siguiente posición de la siguiente de la primera posición.

Cuando en la explicación se haga referencia a algún número n la referencia puede ser al propio número o a $f(n)$ o $g(n)$; se distinguirá a qué se hace referencia por el contexto donde se use. Por otro lado, si la referencia es a un número guardado en una posición, nos referiremos a la codificación de naturales explicada en la sección §3. Por ejemplo, «en la posición 23 está guardado el número 4» significaría que en la posición $g(23)$ hay escrito «1′′′′».

4.2.1. Macros básicas

Las primeras macros sirven para escribir instrucciones de C-- sin necesidad de escribir las marcas. Por ejemplo, en vez de escribir «0′′′′» se puede escribir la macro «0^5». En general, una macro x^n se sustituye por la instrucción con la acción x y posición referida n.

Las siguientes macros empiezan a tomar la forma de construcciones de C. Se introduce la notación Z_Ω; las letras griegas Φ, Ψ y Ω son no-terminales de la gramática; por el momento supondremos que son números naturales. Si se ponen algunos ejemplos sustituyendo las letras griegas por números en las macros que sirven para escribir asignaciones, se comprueba que el subíndice está relacionado con posiciones finales en el programa. Por ejemplo, «$Z_{12} = 9$» se sustituiría por las instrucciones que guardar el número 9 en la posición 12.

La primera posición de los programas es especial, ya que con ella se hacen las comparaciones, y es a su vez la primera instrucción que se ejecuta en todos los programas. La macro «INIT» está pensada para ser usada al principio de todos los programas; se sustituye por «1», que puede ser borrado para guardar cualquier valor, y cuando se ejecuta como instrucción pone una marca en la misma primera posición, con lo que no afecta al resto del programa. La macro «JUMP» se sustituye por la instrucción «=», lo que significa que se salta

una instrucción (lo que hay en la primera posición siempre es igual a lo que hay en la primera posición); esta macro se usa en la expansión de otras macros. La macro «NADA» se usa en la expansión de otras macros. La macro «$Z_\Omega = Z_\Omega$», que sería poner en una posición lo que hay en esa misma posición, se ignora, se sustituye por la cadena vacía.

Las siguientes macros se sustituyen por instrucciones que hacen referencia a posiciones donde se pondrán instrucciones resultado de expandir esas mismas macros. En el programa final estas posiciones dependerán de cuándo aparece la macroinstrucción en el texto del programa. Para poder expandir la macro de la misma manera, sea cuales sean las posiciones finales que ocuparán las instrucciones resultado de expandirla, se usan señales para las posiciones, como se explica en la indicación que tiene el encabezado «←:⟨natural⟩». Así, las instrucciones de salto, en vez de escribirse «$*^n$», con n natural, se escribirán como «$*^{:n}$», con una señalización «←:n» en la posición a la que salta la instrucción. Esta n en :n no se refiere a ningún número; se usan los símbolos de los números naturales como un modo de tener un número infinito de nombres que sean fácil de construir. Las señalizaciones no tienen representación en la transformación final del programa, las pone el traductor como indicador. En los pasos finales de la traducción del programa, cuando ya se sabe en qué posición real está la señalización, se sustituye el nombre :n en la instrucción de salto por el natural que representa la posición. Por ejemplo, si la señalización es «←:27» y está en la posicion 58, en todas las intrucciónes «$x^{:27}$» se sustuirá «:27» por «58». Esta «x^{58}» tiene la forma de la primera macro que se ha comentado, y se sustituirá en un paso posterior por una instrucción con la cantidad de marcas que corresponda. En el apéndice E hay ejemplos que ayudan a entenderlo.

4.2.2. Operaciones aritméticas

La sintaxis de las macros para operaciones aritméticas es indicativa del resultado de expandirlas, y el significado de las Zs es el explicado anteriormente. Así,

$$Z_{12} = Z_3 + Z_{52}$$

se traduciría por instrucciones que ponen en la posición 12 la suma de lo que hay en las posiciones 3 y 52.

Es conveniente explicar aquí el otro significado que pueden tener las letras griegas que se han visto antes. Nos referiremos a las macros de la forma x^Ω como *preinstrucciones*. En las reglas gramaticales, la producción

$$\Omega \mid \Phi \mid \Psi \rightarrow \langle natural \rangle \mid .\langle id \rangle \mid :\langle id \rangle$$

indica que, además de por naturales, las letras griegas se pueden se sustituidas

por cadenas de texto que empiezan, bien por «.», bien por «:». Las que comentaremos ahora son las que empiezan por «.». En la página 84 se explica cómo se sustituyen los .⟨*id*⟩ por naturales. Explicado informalmente, todos las preinstrucciones con el mismo .⟨*id*⟩ se refieren a la misma posición en el programa final, independientemente de la macroinstrucción de la que se haya derivado. Por ejemplo, la posición referida por la instrucción «=$^{\cdot \texttt{op1}}$», resultante de expandir la macro «Z$_8$ += 3;», será la misma que si se expande la macro «Z$_{34}$ += 17;» o si la asignación es para Z$_n$ para cualquier natural n—las posiciones .⟨*id*⟩ se podrían considerar como registros en un procesador moderno—. Para 'reutilizar' estas posiciones, cada .⟨*id*⟩ se trata como si fuese un natural, y una vez que sólo quedan preinstrucciones, esto es, cuando se sabe qué posición ocupará cada instrucción del programa, se reserva una posición a continuación de la última posición ocupada (se añade una instrucción 1, no pensada para ser ejecutada) y se sustituye el .⟨*id*⟩ por el número de posición. Por ejemplo, si el programa de C-- tiene 850 posiciones ocupadas una vez que solamente quedan preinstrucciones después de expandir macros, y el primer .⟨*id*⟩ que aparece en alguna preinstrucción es «.res», en la posición 851 se añade la instrucción «1» y se sustituyen todas las apariciones de «.res» por «851» (así, una preinstrucción como «0$^{\cdot \texttt{res}}$» se transformaría en «0^{851}»).

Hay dos macros que empiezan por «RETURN». La primera simplemente acaba la ejecución del programa. La segunda, «RETURN λ;», guarda λ en la primera posición antes de acabar la ejecución. La producción gramatical

$$\lambda \to \langle \textit{natural} \rangle \mid \texttt{Z}_\Omega$$

indica que λ puede ser un natural o lo que hay guardado en una posición. Así, por ejemplo, «RETURN Z$_8$;» escribe en la primera posición lo que hay en la posición 8 y acaba la ejecución del programa. Esta macro ofrece una manera uniforme de terminar los programas, escribiendo el resultado en una posición conocida.

4.2.3. Reserva de posiciones

Las siguientes macros e indicaciones se usan en la traducción de macros que aparecen más adelante o para traducir tipos de datos, como vectores. Son complicadas de seguir y alargarían mucho la presente explicación. Comentaremos de manera resumida el modo de usar variables.

Las variables son nombres para referirnos a posiciones sin que el escritor del programa necesite saber qué posición representará en el programa final. Estos nombres pueden usarse en lugar de las Zs en todas las construcciones donde éstas aparezcan. Por ejemplo, se puede escribir

```
foo = bar * 9;
```

y se sustituirá por instrucciones que guardan en la posición que representa «`foo`» la multiplicación de lo que hay en la posición que representa «`bar`» por 9. Esto se podría hacer de manera parecida a como se hace con los .⟨*id*⟩, pero para que sea uniforme con cómo se hace en los subprogramas, que se explica más adelante, se hará de una manera más elaborada.

Si se usan nombres en vez de Zs hay que utilizar, antes de que aparezca el nombre, una macro que es la equivalente a la declaración de variables en C. En esta explicación llamaremos a estas macros también declaraciones. Siguiendo con el ejemplo anterior, habría que escribir:

```
unsigned int foo;
unsigned int bar;
```

antes de cualquier otra aparición de `foo` o `bar` en el texto. Estas últimas macros se eliminan sin sustituirse por ningún texto, pero tienen un efecto secundario: por cada declaración de variable que aparezca se asocia un natural al nombre, sea este i, y se sustituye cada aparición del nombre por $\mathtt{Y}_i$ a lo largo de todo el texto siguiente. Los naturales se asociacian en el orden habitual, empezando por 1. Juntando los dos ejemplos anteriores, con las declaraciones de variables antes que la expresión aritmética y suponiendo que no hay ninguna declaración de variables antes que éstas, después de expandir el texto de las declaraciones la expresión aritmética quedaría:

$$\mathtt{Y}_1 \mathtt{\ =\ Y}_2 \mathtt{\ *\ 9;}$$

Para relacionar posiciones finales con los nombres, el expansor guarda en la última posición, a la que llamaremos `:top`, el número de posiciones ocupadas en el programa después de haber reservado posiciones para los .⟨*id*⟩ y contando esta misma posición. Por ejemplo, si después de haber puesto instrucciones «`1`» para cada .⟨*id*⟩ como se explicó antes resultase un programa con instrucciones ocupando 877 posiciones, se añadiría en la posición 878 una instrucción «$\mathtt{1}^{878}$». El número de posición a la que se referirá un $\mathtt{Y}_n$ en el programa final será lo que hay en la posición `:top` más n. En el ejemplo anterior, «`foo`» sería la posición 879 y «`bar`» la 880.

Para que las macros que tratan con variables se puedan expandir de una manera uniforme, todas usan «$\mathtt{Z}_{\mathtt{.change}}$» y la operación `:ajustar:`, que indica al traductor cómo añadir instrucciones que cambian las instrucciones que usan las posiciones de las variables como posiciones referidas. De algún modo, el programa se modifica a sí mismo durante la ejecución, ajustándose según las posiciones a las que quiera acceder. En las páginas 107 a 109 se trata sobre todo esto.

Las macros son generales para poder usar vectores y punteros. Considerando sólo la posibilidad de usar números naturales es menos complicado de entender.

4.2.4. Operadores lógicos y relacionales

Estas macros son sencillas de entender. Aquí se explicarán simplemente las ideas en las que se basa la traducción, no la traducción tal y como aparece en el apéndice, considendo solamente números naturales como operandos. Representaremos en esta explicación los operandos como α y β.

Los operadores lógicos se llaman también booleanos ya que las operaciones con ellos se basan en el álgebra de Boole. La representación simbólica de los operadores que usamos en las traducciones es «&&», «||» y «!», que representan en el álgebra a *and*, *or* y *not* respectivamente. En estas operaciones se consideran sólo dos valores para los operadores: *verdadero* y *falso*. En las traducciones representamos *falso* con el 0 (o dicho de otro modo, el valor 0 se considera falso en cuanto a estas operaciones se refiere); *verdadero* se representa con cualquier otro número. La representación de las operaciones lógicas se puede sustituir por la de operaciones aritméticas del siguiente modo, separando texto y su traducción con «↠»:

- α && β ↠ α * β
- α || β ↠ α + β
- !α ↠ 1 - α

Si se sustituyen los α y β por 0 y cualquier otro valor se comprueba que el resultado que se obtendría es el esperado.

En la reglas gramaticales, el no-terminal que representa los operadores relacionales es ⟨*oprel*⟩; los operadores que se usan son especificados por la producción

⟨*oprel*⟩ → = | != | > | < | >= | <= .

El resultado de las operaciones con estos operadores es el conocido, explicado en [8] (a los operadores «==» y «!=» se les llama allí operadores de igualdad). En pocas palabras, el resultado es *verdadero* o *falso*, representados como se ha indicado antes, dependiendo del orden de los operandos que se comparan, en el orden normal de los números naturales. Junto con los operadores relacionales aparece la construcción «!(α ⟨*oprel*⟩ β)»; el resultado de la operación que representa tiene que ser el contrario del de «α ⟨*oprel*⟩ β».

En la traducción de «α > β» se introduce el subprograma «`negp()`» para que la traducción valga para cualquier número racional, natural o no, aprovechando la representación que se hace de ellos; por la misma razón en la traducción de «α == β» se usa una resta con el número 2. Si sólo se usasen naturales las traducciones podrían ser:

- α > β ⇝ β - α
- α != β ⇝ α > β || α < β
- α == β ⇝ !(α != β)

Las otras traducciones se harían igual.

4.2.5. Estructuras de control

Las estructuras de control sirven para decidir si se ejecuta parte del código y en qué orden, basándose en alguna condición. En las traducciones de la construcción «`if`» y «`if-else`» el resultado de la parte condicional se guarda en una posición identificada por «`Z.obf`» y con una combinación de instrucciones de `C--` *comparar* y *saltar* («`*`» y «`=`») se decidirá la parte del codigo a ejecutar en el programa final. La traducción de «`while`» se basa en la de «`if`» y la de «`do-while`» y la de «`for`» se basan en la de «`while`». La traducción de la construcción «`switch-case`» es la más complicada, pero es un simple adorno y no hace falta entenderla.

4.2.6. Subprogramas

Los subprogramas son programas dentro del programa general, trozos de código que resuelven problemas particulares que ayudan a resolver el problema global. Al igual que el programa general, los subprogramas deben ser capaces de recibir datos y escribir otros datos a modo de resultado. Una manera de integrar los subprogramas sería 'pegar' las instrucciones en los puntos del programa principal donde sea necesario, y tener una posiciones fijas para escribir los datos de entrada y el resultado. Sin necesidad de pegar el código cada vez, se podría poner el código una sóla vez y saltar a la primera instrucción de este código cada vez que se quiera ejecutar el subprograma. Estos modos tienen el problema de que el propio subprograma sobreescribiría los datos de entrada y de resultado si se invocase a sí mismo, es decir, no permite definir subprogramas recursivos. Es necesario para la demostración de la equivalente del lenguaje `C--` con las funciones recursivas, por lo que hay que buscar otro modo de organizar el código. El modo en que se usan las variables nos da la solución. Resumiendo, lo que se hace

es cambiar la posición `:top` cada vez que se vayan a ejecutar las instrucciones del subprograma. Las intrucciones se ajustan igual que para el programa general, pero al cambiar `:top` cada subprograma usará posiciones diferentes para los datos, incluso si se llama a sí mismo. Esto no es otra cosa que usar una pila, igual que se usa en los programas `C` compilados ejecutándose en un ordenador. Cómo se declaran y definen los subprogramas y sus traducciones es farragoso y ocuparía mucho espacio comentarlo todo; está explicado en las páginas 117 a 127.

4.2.7. Entrada/salida

La entrada y salida en los lenguajes de programación es el modo que tienen de ofrecer comunicación con el dispositivo que va a ejecutar el programa. Hay construcciones para insertar datos y otras para que el dispositivo ofrezca resultados. En `C--` no hay secretos; todos los datos e instrucciones están a la vista y se puede escribir y leer directamente en cualquier posición. Así, una construcción como

```
scanf("%d", &entrada);
```

se podría representar con un cartel para el ejecutor del programa, apuntando a la posición que se identifique como «`entrada`», diciendo: *«escribe aquí el dato que quieras usar, codificado como ya sabes»*. Para la salida, el ejecutor tendrá que mirar las posiciones que le interesen e interpretarlas. Por ejemplo, se puede guardar siempre un dato de salida en la misma posición, como se vio con la macro «`RETURN`» y mirar allí al acabar la ejecución del programa.

§5. Equivalencia de máquinas de Turing y C--

La equivalencia de las máquinas de Turing con el lenguaje C-- se demuestra haciendo que una máquina de Turing tome el papel de ejecutor de programas C--. Así, una vez escrito el programa en la cinta de la máquina, ésta actuará según indiquen las instrucciones y dejará escrito en la cinta el programa tal y como quedaría después de la ejecución del mismo. La cinta será pues el homólogo de la superficie infinita que se postuló anteriormente como el lugar donde se escriben los programas C--. La máquina que hace de ejecutor se muestra en el apéndice B; llamaremos a esta máquina $\mathcal{E}$. Vamos a ver cómo se escribe un programa C-- en la cinta de $\mathcal{E}$ para ser ejecutado.

5.1. Cómo escribir el programa

Una instrucción se escribe en la cinta escribiendo símbolos en los cuadrados de la cinta. Si el cuadrado θ es el cuadrado más a la derecha de los que contienen símbolos resultado de escribir una intrucción μ, en el proceso de escritura llamaremos *siguiente cuadrado* de μ al cuadrado inmediatamente a la derecha del cuadrado θ. Utilizando para referirnos a las instrucciones de C-- la notación x^n, que se usa en la gramática presentada anteriormente, diremos que el contenido de una posición de un programa C-- se escribe a partir de un cuadrado κ si se hace del siguiente modo:

- Si la posición está vacía se escribe «-» en el cuadrado κ.
- Si la posición contiene solamente un símbolo (el principal de la instrucción) se escribe en κ este símbolo.
- Si la posición contiene una instrucción de la forma $x^{n\prime}$ se escribe la instrucción x^n a partir del cuadrado κ y se escribe en el siguiente cuadrado de esta instrucción el símbolo «'».

Para escribir un programa P en la cinta de la máquina $\mathcal{E}$, que en un principio no tendrá ningún cuadrado ocupado, se escoge cualquier cuadrado, llamémosle κ, y escribimos P de la siguiente manera:

- Se escribe a partir del cuadrado κ el contenido de la primera posición de P.
- Si se ha escrito la instrucción μ contenida de una posición ψ, y en P hay una posición ω siguiente a ψ, se escribe el contenido de ω a partir del siguiente cuadrado de μ. Si no hay una posición ω la escritura se ha completado.

En resumen, se escriben los símbolos del programa, uno en cada cuadrado de la cinta, en el orden en el que aparecerían si se pusiera el contenido de todas las

posiciones concatenado, con «-» representando las posiciones vacías. Por ejemplo, si tenemos el programa:

1′
=
0
1
∗′′′′

se escribiría en la cinta de una máquina de la siguiente manera:

...	1	′	=	-	0	1	∗	′	′	′	′	...

5.2. Funcionamiento de la máquina $\mathcal{E}$

Comentamos aquí algunas de las partes de la máquina $\mathcal{E}$. Antes de empezar la ejecución del programa C-- escrito en la cinta de la máquina, ésta debe estar leyendo algún cuadrado con símbolos y estará en la *m*-configuration $\mathfrak{b}$. La máquina empezará, pues, con

$\mathfrak{b}$ $\mathfrak{ini}(\mathfrak{selin})$

que simplemente hace que la máquina pase a la *m*-configuration $\mathfrak{ini}(\mathfrak{selin})$. En esta *m*-configuration la máquina salta símbolos hacia la izquiera hasta leer un cuadrado vacío; entonces se moverá un cuadrado a la derecha. El resultado es que al acabar estará leyendo el primer símbolo de la instrucción en la primera posición del programa C-- (o el símbolo «-» si en el programa C-- la primera posición está vacía).

En la *m*-configuration $\mathfrak{selin}$ la máquina busca en la cinta la siguiente instrucción a ejecutar; cambia el primer símbolo principal que encuentre por «a», «b», «c» o «d», según el símbolo. Al ser un símbolo distinto para cada acción se puede saber en fases posteriores, además de qué intrucción es la que hay que ejecutar, cuál es su acción, y una vez ejecutada la acción permitirá restaurar el símbolo original. A diferencia de las máquinas originales presentadas en [1], la máquina $\mathcal{E}$ no trabaja con cuadrados en blanco entre los símbolos (los llamados por Turing «E-squares»). El equivalente a lo que Turing llama marcar se hace en $\mathcal{E}$ de la manera en la que se hace en $\mathfrak{selin}$, con símbolos asociados con otros y sobrescribiendo y restaurando los originales.

En las *m*-configuration $\mathfrak{buscop}$ se busca la posición referida de la instrucción que se va a ejecutar. Lo hace cambiando los símbolos «'» por «|» para llevar la cuenta, y cada vez que cambia un símbolo avanza una posición en el programa `C--`, empezando desde la primera. Cuando no hay más símbolos «'» en la instrucción que se está ejecutando es que ya se ha encontrado la posición referida; se pasa entonces a $\mathfrak{encop}(\alpha)$.

En la *m*-configuration $\mathfrak{encop}(\alpha)$ la máquina se va moviendo hacia la izquierda restaurando los símbolos «'», borrando para ello los símbolos «|». Cuando encuentra «a», «b», «c» o «d» va a la configuración que ejecuta la instrucción pertinente.

En $\mathfrak{saltar}(\alpha)$ la máquina va a la posición referida si hace falta (si no es la misma posición que la de la instrucción, lo que sería un bucle infinito) y seguirá la ejecución de instrucciones desde allí.

En $\mathfrak{marcar}(\alpha)$ la máquina pone «1» si lo que había es «z», que representa que la posición estaba vacía. Si no desplaza todos los símbolos a la derecha del símbolo principal de la posición referida, y escribe «'» en el cuadrado vacío que queda.

En $\mathfrak{borrar}(\alpha)$ no se hace nada si la posición referida estaba vacía; en caso de que hubiera un símbolo principal, borra el símbolo si éste es «1» y borra todos los símbolos «'» de la posición. Para ello borra un «'» de la posición referida y mueve una posición a la izquierda todos los simbolos a partir del cuadrado de la derecha del que contenía el «'», hasta que no quedan más «'» en la posición referida.

Las operaciones en la *m*-configuration $\mathfrak{comp}(\alpha)$ básicamente siguen el mismo procedimiento que se hace al buscar posiciones referidas. Se comprueba si la posición referida y la primera posición están ambas vacías; si no es así se van cambiando los símbolos «'» por simbolos «~» y «&» en la posición referida y la primera posiciones para ver si hay la misma cantidad, que puede ser ninguna o más. En caso de ser iguales se salta una posición, como indica la semántica de la instrucción.

La máquina $\mathcal{E}$ es complicada de seguir, como son casi todas las máquinas de Turing excepto las más simples; no obstante, poniendo un poco de atención se debería entender sin grandes dificultades.

5.3. Sin programa, solamente datos de entrada

Vamos a ver cómo obtener a partir de la máquina $\mathcal{E}$ una máquina que dé el mismo resultado al ejecutar un programa P sin tener que escribir en la cinta el programa `C--`, sino escribiendo simplemente los datos de entrada para el programa. A esta máquina la llamaremos $\mathcal{E}'$. Supondremos que los valores de entrada en P se colocan en las primeras posiciones de las reservadas para variables. Si no es así se modifica cambiando el orden en que se declaran las variables, según se vio en la traducción de macros.

Consideremos la secuencia de símbolos en la cinta de $\mathcal{E}$ después de escribir el programa *P* como se indicó anteriormente. El número de instrucciones que componen el programa es finito y cada instrucción está compuesta por un número finito de símbolos, por lo que la secuencia que hay en la cinta también es finita. Sea *n* el número de símbolos en la cinta y sea

$$s_1, s_2, s_3, \ldots, s_n$$

la secuencia de símbolos, donde s_1 es el símbolo colocado más a la derecha en la cinta, y para cada i, con $1 \leq i \leq n$, s_i es el símbolo a la derecha de s_{i+1}. Por cada s_i crearemos una entrada en la tabla de definición de $\mathcal{E}'$ de la forma

$\mathfrak{C}_i$ `Any` `L,P`s_i $\mathfrak{C}_{i+1}$

Por ejemplo, si hubiera al menos un símbolo, y el primer símbolo empezando por la derecha fuera «'», la primera de estas entradas sería

$\mathfrak{C}_1$ `Any` `L,P'` $\mathfrak{C}_2$

Esto hace que la máquina se mueva a la izquierda, escriba un símbolo s_i y pase a la *m*-configuration que hará lo mismo con el símbolo que estaba originalmente a la izquierda de s_i.

Añadimos todas las entradas de $\mathcal{E}$ a la tabla de $\mathcal{E}'$ y hacemos dos modificaciones: cambiamos la última entrada creada antes con *m*-configuration inicial s_n y la transformamos en

$\mathfrak{C}_i$ `Any` `L,P`s_i `selin`

Por ejemplo, si hubiese 874 símbolos en la cinta de $\mathcal{E}$ y el primer símbolo (el colocado más a la izquierda) fuese «1», la última de las entradas de las creadas por cada símbolo sería

$\mathfrak{C}_{874}$ `Any` `L,P1` `selin`

Cambiamos ahora la entrada inicial, que aparece al principio del apartado 5.2, y la transformamos en

$\mathfrak{b}$ ini($\mathfrak{C}_1$)

Con estas dos modificaciones tenemos la máquina $\mathcal{E}'$ que estábamos buscando. Al igual que la máquina $\mathcal{E}$ esta máquina empezará en la *m*-configuration $\mathfrak{b}$ y leyendo algún símbolo de los datos de entrada. Los datos de entrada se escribirán como ya se ha visto, con símbolos «1» y «'» o el símbolo «-» si no hay datos de entrada.

5.4. Equivalencia en sentido contrario

La demostración de que hay un programa `C--` equivalente a cualquier máquina de Turing es muy sencilla y no merece sección propia. En el apéndice C se muestra un programa escrito con macros, es decir, escrito en `C`, que es equivalente a una máquina concreta. Cambiando el valor de la variable `tmachine` se obtiene un programa equivalente a cualquier máquina. La explicación dada en los comentarios debería ser sufiente para entenderlo.

§6. Equivalencia de funciones recursivas y C--

Al igual que con las máquinas de Turing, la demostración de la equivalencia de las funciones recursivas consiste en presentar una función que actúa como ejecutor universal de programas C--. Esta función, llamada `computar`, aparece en el epéndice D.

Las funciones recursivas reciben como entrada y dan como resultado números naturales. Por tanto, habrá que encontrar la manera de codificar los programas, es decir, encontrar una función C cuyo dominio sean los programas C-- y tal que $C(\psi)$ sea la codificación como un número natural de un programa ψ; así, si P es el programa a ejecutar y P' el programa que queda después de la ejecución, entonces $C(P') = \texttt{computar}(C(P))$.

Vamos a ver cómo es la función C. Seguidamente se comentará brevemente la función `computar` y para finalizar la sección veremos cómo obtener, a partir de `computar` y para un programa cualquier P, una función recursiva que recibiendo como argumento unos datos de entrada calcule lo mismo que calcularía P con esos datos.

6.1. Programas como números naturales

Para ver cómo representar un programa C-- como un número natural para que pueda ser usado por las funciones recursivas dividiremos la explicación en dos partes: explicamos primero cómo se codifica el contenido de una posición y valiéndonos de esto definiremos después la función C que buscamos.

6.1.1. Codificación de contenido de posiciones

Para un programa p en C-- vamos a definir una función c_p sobre las posiciones de p tal que dada una posición devuelva la codificación como número natural del contenido de esa posición. Definiremos primero una función f, con las intrucciones de C-- como dominio, del siguiente modo:

- $f() = 0$; esto es, si no hay dato (contenido de una posición vacía).
- Para instrucciones que tienen un único símbolo:

$$f(\texttt{0}) = 1, \qquad f(\texttt{=}) = 2, \qquad f(*) = 3, \qquad f(\texttt{1}) = 4.$$

- Para instrucciones de la forma $x^{n\prime}$, con $n \in \mathbb{N}$:

$$f(x^{n\prime}) = f(x^n) + 4.$$

Por ejemplo, $f(\texttt{1}''') = 16$ y $f(\texttt{=}'') = 10$.

Definimos ahora dos sencillas funciones: una función g que dada una posición de un programa devuelve la instrucción contenida en esa posición, o nada si la

posición está vacía; y una función para un programa p, h_p, tal que $h_p(1)$ es la primera posición de p, y, para todo $n \in \mathbb{N}$, $h_p(n+1)$ es la siguiente de la posición $h_p(n)$ si tal posición existe, y no está definido si la posición no existe. La función c_p será la composición de estas tres funciones:

$$c_p = f \circ g \circ h_p.$$

Poniendo como ejemplo un programa al que llamaremos Ψ:

1''
1'''
1
$*'$

$c_\psi(1) = 12$
$c_\psi(2) = 16$
$c_\psi(3) = 4$
$c_\psi(4) = 7$

6.1.2. Codificación de programas

Para llegar hasta la definición de C que buscamos, antes definimos dos funciones: una función, pm, para todo $n \in \mathbb{N}$ de la siguiente manera:

- $pm(1) = 5$
- $pm(n+1)$ es el menor primo mayor que $pm(n)$

y otra función sobre un programa p, a la que llamaremos cm, con el mismo dominio que la función c definida anteriormente, del siguiente modo:

- $cm_p(1) = 3^{c(1)}$
- $cm_p(n+1) = cm_p(n) \times pm(n)^{c(n+1)}$

Definimos por fin la función C que buscamos, que da como resultado la codificación de un programa P, como sigue:

$$C(P) = cm_p(\alpha) \qquad \begin{array}{l}\text{donde } \alpha \geq n \text{ para todo } n \\ \text{perteneciente al dominio de } cm_p.\end{array}$$

Llamemos última posición de un programa P a la posición para la que no hay siguiente posición en P. Sea $\mathbb{U}$ el conjunto de los programas tales que la última posición está vacía y es la siguiente de alguna posición. Es fácil ver que cualquier programa en $\mathbb{U}$ se puede transformar en uno equivalente a efectos de computación, esté o no en $\mathbb{U}$, quitando la última posición. Con los programas que no pertenecen a $\mathbb{U}$ como dominio, la función C es biyectiva; por tanto, cualquier programa, o uno equivalente, se puede codificar y decodificar unívocamente.

Vamos a ver un ejemplo de codificación de un programa. Para ello tomaremos de nuevo el programa que se usó en §5 para explicar cómo se escribe en una máquina de Turing y lo usaremos otra vez de ejemplo para ver cómo se transforma en un número natural que sirva como entrada para la funcíon recursiva del apéndice D. Así, si llamamos ξ al programa

1′
=
0
1
∗′′′′

el número que lo representaría será:

$$C(\xi) = 3^8 \times 5^2 \times 7^0 \times 11^1 \times 13^4 \times 17^{19}$$

La manera de descodificar un número para obtener el programa que representa se ve en el funcionamiento de la función `computar`. Damos a continuación algunos apuntes sobre su funcionamiento.

6.2. Apuntes sobre la función `computar`

La definición de función recursiva que se usa aquí está basada en las definiciones de funciones primitivas recursivas que se dan en [2], pag. 219, y en [6], capítulo 6. Concretamente, la función `computar` y las demás funciones que se usan son de uno de los siguientes tipos, expresados con la notación en [2], donde $y' = y + 1$ y $1 \leq i \leq n$:

(i) $\varphi(x_1, \ldots, x_n) = 0$

(ii) $\varphi(x) = x + 1$

(iii) $\varphi(x_1, \ldots, x_n) = x_i$

(iv) $\varphi(x_1, \ldots, x_n) = \psi(\chi_1(x_1, \ldots, x_n), \ldots, \chi_m(x_1, \ldots, x_n))$

(v)
$$\begin{cases} \varphi(0, x_2, \ldots x_n) = \psi(x_2, \ldots, x_n), \\ \varphi(y', x_2, \ldots x_n) = \chi(y, \varphi(y, x_2, \ldots x_n), x_2, \ldots, x_n) \end{cases}$$

Las dos primeras funciones en el apéndice D, llamadas `cero`, son del tipo (i); la siguiente función que aparece, `S`, es del tipo (ii); las funciones entre las líneas 20

y 82, con nombres de la forma «U*x_y*» son del tipo (iii); todas las demás funciones son de los tipos (iv) o (v). Concretamente, `computar` es del tipo (iv).

Las funciones anteriores y el resto se muestran como parte de un programa `C--` escrito con macros, o sea, como si estuviera escrito en `C`, con lo que son sencillas de entender y no requieren mucha explicación. Veremos por encima algunas de las funciones y cómo se hace la computación. Pero antes de seguir con el repaso de las funciones conviene introducir el concepto que llamaremos *registro*, relacionado con los programas `C--` y que se maneja durante la computación.

6.2.1. Señalador y registro

Imaginemos que el ejecutor de un programa `C--` tiene otras cosas que hacer y va y viene, alternando la ejecución de algunas instrucciones del programa con sus otras tareas; o que el programa es muy largo y hay varios ejecutores que van turnándose en la ejecución del programa. Esto se podría hacer manteniendo un señalador que apunte a la próxima instrucción que será ejecutada, del mismo modo que se usa un marcapáginas en un libro. Asi se podrá dejar y retomar la tarea de ejecutar el programa a capricho. Al conjunto de programa y el señalador es a lo que llamamos *registro*.

Durante el cálculo con `computar` se trabaja con la codificación de registros. Al principio de la ejecución se obtiene un registro a partir de la codificación del programa, con el señalador en la primera posición; según se van ejecutando instrucciones se obtienen las codificaciones con el señalador donde corresponda. Para un programa P, la codificación de un registro ρ es como sigue:

- Si $\upsilon = C(P)$ es la codificación de P, entonces $\rho = 2 \times \upsilon$ es la codificación del registro con el señalador apuntando a la primera posición de P.
- Si μ es la codificación de un registro con el señalizador apuntando a la posición ψ, entonces $\rho = 2 \times \mu$ es la codificación del registro con el señalizador apuntando a la siguiente posición de ψ en P.

Una vez vistos estos conceptos y cómo se codifican, vamos a ver cómo se usan en las funciones recursivas cuando ejecutan un programa `C--`, siguiendo con el repaso del apéndice D.

6.2.2. Apuntes (continuación)

Las funciones entre las líneas 86 y 142 devuelven las constantes necesarias en otras funciones; seguidamente hay funciones que realizan operaciones aritméticas y relacionales, y basadas en ellas otras funciones que resuelven problemas más específicos y que serán necesarias para definir funciones que aparecen más adelante. Los comentarios que aparecen en el código explican lo que hacen.

La función `computar` recibe el programa y lo pasa junto con la constante 2 a la función `computar_aux`, para que ésta multiplique, creando el registro con el señalador apuntando a la primera posición. La función `evaluar` recibe el registro junto con la constante 1. Esta función decide si parar la ejecución o continuar. El parámetro `X1` indica si la última vez que se interpretó una posición ésta estaba vacía o si hay que seguir con la ejecución del programa. Si se llama con 0 la ejecución se detendría y se devolvería el programa ejecutado. `computar_aux` llama a `evaluar` con argumento 1, ya que estamos al principio de la evaluación del programa, y esta última llama a `ejecutar`, que interpretará la posición que indica el señalador.

Desde `ejecutar`, a partir del registro se saca el contenido de la posición a la que apunta el señalador y de éste la instrucción, si la hubiera. La función `eval_registro` intenta ejecutar los 4 tipos de instrucciones o nada si la posición a interpretar está vacía. Cada una de las funciones que empieza por `eval_` recibe como primer parámetro el tipo de instrucción qeu hay que ejecutar o si la posición está vacía; cada una de las funciones modificará y devolverá el registro según reciba el tipo de instrucción que puede ejecutar o devolverá 0 en caso contrario. Por ejemplo, si el primer parámetro con el que se llama a la función `eval_marcar` indica que la intrucción a ejecutar es *marcar*, pondrá una marca en la posición referida por la instrucción, avanzará el señalador una posición y devolverá el registro así modificado; si la instrucción es otra, devolverá 0 en vez del registro. La función `eval_registro` devuelve finalmente la suma de todos los resultados; sólo uno será distinto de 0, con lo que éste será el resultado global. La función `ejec3` recibe este valor y llama a `hay_inst` para comprobar si la posición estaba vacía, guardando este valor en `V1`, que será 0 o 1; con este valor y el registro después de interpretar la posición adecuada, vuelve a empezar el ciclo llamando a `evaluar`.

Casi todas las demás funciones tienen que ver con el descodificado y codificado de intrucción, posiciones, señalador..., y con la ejecución de las distintas instrucciones. Es más fácil seguirlo con el código y los comentarios que con una explicación.

6.3. Sin programa, solamente datos de entrada

Vamos a ver cómo construir una función que reciba los datos de entrada que recibiría un programa *P* en `C--` y devuelva lo mismo que devolvería la función `computar` cuando tiene como entrada el programa *P* con esos datos. Esta función lo único que tendrá que hacer es 'preparar' un programa a partir de los datos de entrada y de *P* y pasarlo como argumento a `computar` para que lo ejecute. Al igual que se hizo en la explicación para las máquinas de Turing, vamos a

suponer que los datos de entrada para P se colocarían en las primeras posiciones correspondientes a las variables o que se modifica P para que sea así. Para que la explicación sea más corta supondremos también que el programa P recibe solamente un dato de entrada. El caso para más de un dato de entrada es algo más largo pero el proceso es casi igual.

Para que la función que buscamos, a la que llamaremos `precomputar`, ejecute un programa a partir de los datos de entrada, este programa tiene que ser conocido al construir la función, y por tanto serán conocidos también su codificación y el mayor número primo que se ha usado para la codificación; sean éstos n y p respectivamente. La función sería:

```
function precomputar(var X1){
   var V1 = preparar(X1);
   return computar(V1);
}
```

La función preparar usará el dato de entrada y la codificación n, que es una constante, para devolver la codificación del programa que necesitamos. La función será:

```
function preparar(var X1){
   var V1 = programa(X1);
   var V2 = entrada(X1);
   return mult(V1, V2);
}
```

La función `programa` usará `cero` y luego `S` las veces que haga falta para devolver la constante n. La función `entrada` devolverá p^{θ}, donde θ es la codificación del contenido de lo que sería la posición siguiente a la última posición del programa P con el dato de entrada en ella. Con la multiplicación que se lleva a cabo en `mult` se obtiene la codificación del programa que queremos. La función `entrada` sería de la siguiente manera:

```
function entrada(var X1){
   var V1 = cod_instrucción(X1);
   var V2 = primo(X1);
   var V3 = uno(X1);
   return mult_veces(V1, V2, V3);
}
```

La función `primo` devolverá como constante el número primo p mencionado anteriormente, de manera análoga a se hace en `programa`. `cod_instrucción`

devuelve el θ visto antes, que será igual que la codificación del contenido de una posición con una instrucción *marcar* con la posición referida igual al dato de entrada. Las funciones `uno` y `mult_veces` aparecen en el apéndice D. La función `cod_instrucción` sería:

```
function cod_instrucción(var X1){
   var V1 = cuatro(X1);
   var V2 = U1_1(X1);
   return mult(V1, V2);
}
```

Las funciones `U1_1` y `mult` aparecen en el apéndice D. La función `cuatro` con un sólo parámetro no aparece pero a partir de las que aparecen es fácil ver cómo construirla para que devuelva la constante 4.

Apéndice A

Especificación de C--

Un programa C-- consiste en un conjunto de posiciones, cada una de las cuales contiene una instrucción o está vacía. Una posición, o bien es la primera, o bien es la siguiente de otra posición. Llamamos interpretar una posición ψ a ejecutar la instrucción que hay en ψ o parar la ejecución del programa si ψ está vacía. La ejecución de un programa empieza interpretando la primera posición del mismo.

A continuación se muestran las instrucciones que pueden aparecer en una posición. Se usa una variante de la forma de Backus-Naur para definir la sintaxis, con los no-terminales entre los signos «⟨» y «⟩», los terminales en negrita, la cabecera de la producción separada del cuerpo por el signo «→» y las opciones de sustitución separadas por el signo «|». Seguido de la definición sintáctica de las instrucciones se da la explicación para la ejecución de cada instrucción.

Instrucciones

⟨*marcar*⟩ → **1** | ⟨*marcar*⟩'

Se pone el símbolo «1» en la posición referida por la instrucción si no hay nada en dicha posición, o un símbolo «'» si la posición no está vacía. La posición referida por una instrucción de este tipo es:

- 1: primera posición
- ⟨*marcar*⟩': siguiente posición a la referida por ⟨*marcar*⟩

Si la instrucción está en la posición ψ, después de ejecutar esta instrucción se pasa a interpretar la posición siguiente a ψ.

⟨*borrar*⟩ → **0** | ⟨*borrar*⟩'

Se quitan todos los símbolos «'» de la posición referida, y el símbolo «1» si lo hubiera. La posición referida por una instrucción de este tipo es:

- 0: primera posición
- ⟨*borrar*⟩': siguiente posición a la referida por ⟨*borrar*⟩

Si la instrucción está en la posición ψ, después de ejecutar esta instrucción se pasa a interpretar la posición siguiente a ψ.

⟨*comparar*⟩ → = | ⟨*comparar*⟩'

Sea ψ la posición en la que está la instrucción. Si la posición referida por la instrucción y la primera posición no son iguales según la definición de igualdad de posiciones que se da a continuación, se interpreta la posición siguiente a ψ; si son iguales se interpreta la posición siguiente a la siguiente posición de ψ. La posición referida por una instrucción de este tipo es:

- =: primera posición
- ⟨*comparar*⟩': siguiente posición a la referida por ⟨*comparar*⟩

Dos posiciones son iguales si y sólo si una de las siguientes afirmaciones es cierta:

- En ninguna de las dos posiciones hay nada.
- En cada una de las dos posiciones hay solamente un símbolo.
- Cada una de las dos posiciones tiene símbolos «'» y si se quita a ambas uno de estos símbolos «'» las posiciones son iguales.

⟨*saltar*⟩ → ∗ | ⟨*saltar*⟩'

Se interpreta la posición referida por la instrucción. La posición referida por una instrucción de este tipo es:

- ∗: primera posición
- ⟨*saltar*⟩': siguiente posición a la referida por ⟨*saltar*⟩

Apéndice **B**

Máquina universal de Turing para programas C--

```
1  ; Máquina que ejecuta un programa C-- escrito en la
2  ; cinta. Si hay posiciones vacías en medio de otras
3  ; posiciones se indican con un símbolo '-'. El lector
4  ; debe estar al inicio en alguna posición con símbolos.
5
6  ;m-configuration inicial
7  b     ...     ini(selin)
8
9  ;ir al inicio de los símbolos
10 ini(A){
11       Any     L     ini(A)
12       None     R     A
13 }
14
15 selin{
16       '     R     selin
17       1     E, Pa     buscop(a)
18       0     E, Pb     buscop(b)
19       =     E, Pc     buscop(c)
20       *     E, Pd     buscop(d)
21 }
22
23 rest(A){
24       e     E, P1     A
25       f     E, P0     A
26       g     E, P=     A
27       h     E, P*     A
28       z     E, P-     A
29       a     A
30       b     A
31       c     A
32       d     A
33 }
34
35 ;mover a la derecha
36 d(A)       R     A
```

```
37 ;buscar símbolo α e ir cuando
38 ;se encuentre a m-configuration A.
39 f(𝔄, α){
40       α     𝔄
41       not α     L     f(𝔄, α)
42       None     R     f'(𝔄, α)
43 }
44
45 f'(𝔄, α){
46       α     𝔄
47       not α     R     f'(𝔄, α)
48 }
49
50 ;qué operación ejecutar
51 encop(α){
52       |     E, P', L     encop(α)
53       a     f(marcar(α), α)
54       b     f(borrar(α), α)
55       c     f(comp(α), α)
56       d     saltar(α)
57 }
58 ;ir al final de la cinta
59 final(𝔄){
60       Any     R     final(𝔄)
61       None     L     𝔄
62 }
63
64 sigin{
65       1     R     sigin
66       0     R     sigin
67       =     R     sigin
68       *     R     sigin
69       '     R     sigin
70       -     R     sigin
71       a     E, P1, R     selin
72       b     E, P0, R     selin
73       c     E, P=, R     selin
74 }
```

```
75  buscop(α)        ini(buscop2(α))
76
77  buscop2(α){
78        '     R     buscop2(α)
79        |     R     buscop2(α)
80        1     E, Pe     buscop3(α, e)
81        0     E, Pf     buscop3(α, f)
82        =     E, Pg     buscop3(α, g)
83        *     E, Ph     buscop3(α, h)
84        -     E, Pz     buscop3(α, z)
85        None     Pz     buscop3(α, z)
86        α     buscop3(α, α)
87  }
88
89  buscop3(α, β)        f(d(buscop4(α, β)), α)
90
91  buscop4(α, β){
92        |     R     buscop4(α, β)
93        '     E, P|     buscop5(α, β)
94        not | nor '     L     encop(β)
95        None     L     encop(β)
96  }
97
98  buscop5(α, β)        f(rest(d(buscop2(α))), β)
99
100 ;operación para la acción «*»
101 saltar(α){
102       α     E, P*     selin
103       not α     E, P*     f(rest(selin), α)
104 }
105
106 ;operación para la acción «1»
107 marcar(α){
108       z     E, P1     ini(sigin)
109       not z     final(marc(α))
110 }
111
112 hueco(α)     β     E. R. Pβ. L. L     marc(α)
```

```
113 marc(α){
114       α     R, P', L     rest(ini(sigin))
115       not α      hueco(α)
116 }
117
118 ;operación para la acción «O»
119 borrar(α){
120       z     E, P-      ini(sigin)
121       e     E, Pz      d(qmarc(z))
122       not z nor e      d(qmarc(α))
123 }
124
125 qmarc(α){
126       '     E, R     cerhueco(f(d(qmarc(α)), α))
127       not '     L     rest(ini(sigin))
128       None      L     rest(ini(sigin))
129 }
130
131 cerhuec(𝔄){
132       β     E, L, Pβ, R, R     cerhueco(𝔄)
133       None      L, L     𝔄
134 }
135
136 ;operación para la acción «=»
137 comp(α){
138       z     E, P-      ini(d(compn))
139       not z     ini(comppos(α))
140 }
141
142 compn{
143       -     ssigin
144       not -     sigin
145 }
146
147 comppos(α){
148       -     sigin
149       not -     R     compm(α)
150 }
```

```
151 saltp(𝔄){
152       ~     saltp(𝔄)
153       &     saltp(𝔄)
154       not ~ nor &     𝔄
155 }
156
157 saltot(𝔄){
158       ~     R     saltot(𝔄)
159       not ~     𝔄
160 }
161
162 compm(α){
163       '     E, P&     umo(comp2(α), α)
164       not '     umo(comf, α)
165 }
166
167 umo(𝔄, α)       f(d(saltot(𝔄)), α)
168
169 ump(α)       ini(d(saltp(compm(α))))
170
171 comp2(α){
172       &     E, P~     ump(α)
173       '     E, P~     ump(α)
174       not & nor '     L     falso
175       None     L     falso
176 }
177
178 reset(𝔄){
179       ~     E, P', L     reset(𝔄)
180       not ~     rest(reset2(𝔄))
181 }
182
183 reset2(𝔄){
184       &     E, P', L     reset2(𝔄)
185       not &     L     reset2(𝔄)
186       None     R     𝔄
187 }
188
```

```
189 compf{
190       '     L     falso
191       not '     L     verdadero
192 }
193
194 falso       reset(sigin)
195
196 verdadero       reset(ssigin)
197
198 ssigin       f(d(saltsim(selin)), c)
199
200 saltsim(A){
201       '     R     saltsim(A)
202       not '     R     A
203 }
```

Apéndice C

Ejemplo de máquina de Turing en C

```c
/*
 *  Programa en C-- para ejecutar un procedimiento definido
 *  como máquina de Turing. La máquina se define con quíntuplas
 *  del siguiente modo:
 *
 *  {mc, s, s2, m, mcf} donde:
 *    mc - m-configuration actual.
 *    s - símbolo leído.
 *    s2 - símbolo escrito.
 *    m - movimiento.
 *    mcf - m-configuración a la que se pasa.
 */

#include cmmstd

#define MOVE 1

/* Definición de la máquina. Este ejemplo ejecuta la primera máquina
 * de la sección 3 del 'On computable numbers' del maestro Turing,
 * que computa la secuencia 010101... */

unsigned int tmachine[] = {1, 0, '0', 'R', 2,
                           2, 0, '0', 'R', 3,
                           3, 0, '1', 'R', 4,
                           4, 0, '0', 'R', 1}

/* Guarda la cantidad de configuraciones.*/
unsigned int configurations = 4;
/* m-configuración actual.*/
unsigned int m_configuration = 1;
/* Símbolo escaneado.*/
unsigned int scanned = 0;
/* Fila que se comprueba de la tabla de definición.*/
unsigned int row;
/* Posiciones que representan la cinta.*/
unsigned int tape;

```

```
38 while(MOVE){
39     row = 0;
40     while(row < configurations){
41         if(tmachine[row][0] != m_configuration ||
42             tmachine[row][1] != tape[scanned]){
43             row++;
44         }
45         else{
46             tape[scanned] = tmachine[row][2];
47             switch(tmachine[row][3]){
48                 case 'R':
49                     scanned++;
50                 case 'L':
51                     scanned--;
52             }
53             mconfiguration = tmachine[row][4];
54             row = configurations + 1;
55         }
56     }
57     if(row == configurations){
58         STOP
59     }
60 }
```

Apéndice **D**

Función recursiva ejecutora de programas C--

```
1  #define function unsigned int
2  #define var unsigned int
3
4  /************************/
5     Funciones básicas.
6  /************************/
7  function cero(){
8      return 0;
9  }
10
11 function cero(var X1){
12     return 0;
13 }
14
15 function S(var X1){
16     X1++;
17     return X1;
18 }
19
20 function U1_1(var X1){
21     return X1;
22 }
23
24 function U2_1(var X1, var X2){
25     return X1;
26 }
27
28 function U2_2(var X1, var X2){
29     return X2;
30 }
31
32 function U3_1(var X1, var X2, var X3){
33     return X1;
34 }
35
36
37
```

```
38 function U3_2(var X1, var X2, var X3){
39     return X2;
40 }
41
42 function U3_3(var X1, var X2, var X3){
43     return X3;
44 }
45
46 function U4_1(var X1, var X2, var X3, var X4){
47     return X1;
48 }
49
50 function U4_2(var X1, var X2, var X3, var X4){
51     return X2;
52 }
53
54 function U4_3(var X1, var X2, var X3, var X4){
55     return X3;
56 }
57
58 function U4_4(var X1, var X2, var X3, var X4){
59     return X4;
60 }
61
62 function U5_1(var X1, var X2, var X3, var X4, var X5){
63     return X1;
64 }
65
66 function U5_2(var X1, var X2, var X3, var X4, var X5){
67     return X2;
68 }
69
70 function U5_3(var X1, var X2, var X3, var X4, var X5){
71     return X3;
72 }
73
74
75
```

```
76 function U5_4(var X1, var X2, var X3, var X4, var X5){
77     return X4;
78 }
79
80 function U5_5(var X1, var X2, var X3, var X4, var X5){
81     return X5;
82 }
83
84 /*********************************************/
85 /* Devuelve siempre 0. */
86 function cero(var X1, var X2){
87     var V1 = U2_1(X1, X2);
88     return  cero(V1);
89 }
90
91 /* Devuelve siempre 1. */
92 function uno(var X1){
93     var V1 = cero(X1);
94     return  S(V1);
95 }
96
97 /* Devuelve siempre 1. */
98 function uno(var X1, var X2, var X3){
99     var V1 = U3_2(X1, X2, X3);
100    return  uno(V1);
101 }
102
103 /* Devuelve siempre 2. */
104 function dos(var X1){
105    var V1 = uno(X1);
106    return  S(V1);
107 }
108
109 /* Devuelve siempre 2. */
110 function dos(var X1, var X2){
111    var V1 = U2_1(X1, X2);
112    return  dos(V1);
113 }
```

```
114 /* Devuelve siempre 2. */
115 function dos(var X1, var X2, var X3){
116     var V1 = U3_2(X1, X2, X3);
117     return  dos(V1);
118 }
119
120 /* Devuelve siempre 3. */
121 function tres(var X1, var X2){
122     var V1 = dos(X1, X2);
123     return  S(V1);
124 }
125
126 /* Devuelve siempre 3. */
127 function tres(var X1, var X2, var X3){
128     var V1 = dos(X1, X2, X3);
129     return  S(V1);
130 }
131
132 /* Devuelve siempre 4. */
133 function cuatro(var X1, var X2){
134     var V1 = tres(X1, X2);
135     return  S(V1);
136 }
137
138 /* Devuelve siempre 4. */
139 function cuatro(var X1, var X2, var X3){
140     var V1 = tres(X1, X2, X3);
141     return  S(V1);
142 }
143
144 /* ---------------------------------------------- */
145 /* Función auxiliar para la función «suma». */
146 function suma_aux(var X1, var X2, var X3){
147     var V1 = U3_2(X1, X2, X3){
148     return S(V1);
149 }
150
151
```

```
152 /* Devuelve la suma de X1 y X2. */
153 function suma(var X1, var X2){
154     if(X1 == 0){
155         return U1_1(X2);
156     }
157     else{
158         X1--;
159         var rr = suma(X1, X2);
160         return suma_aux(X1, rr, X2);
161     }
162 }
163
164 /* ---------------------------------------------- */
165
166 /* Auxiliar para «suma3». */
167 function suma_aux3(var X1, var X2, var X3){
168     var V1 = U3_2(X1, X2, X3);
169     var V2 = U3_3(X1, X2, X3);
170     return suma(V1, V2);
171 }
172
173 /* Devuelve la suma de X1, X2 y X3. */
174 function suma3(var X1, var X2, var X3){
175     var V1 = U3_1(X1, X2, X3);
176     var V2 = suma_aux3(X1, X2, X3);
177     return suma(V1, V2);
178 }
179
180 /* Auxiliar para «suma4». */
181 function suma_aux4(var X1, var X2, var X3, var X4){
182     var V1 = U4_2(X1, X2, X3, X4);
183     var V2 = U4_3(X1, X2, X3, X4);
184     var V3 = U4_4(X1, X2, X3, X4);
185     return suma3(V1, V2, V3);
186 }
187
188
189
```

```
/* Devuelve la suma de X1, X2, X3 y X4. */
function suma4(var X1, var X2, var X3, var X4){
    var V1 = U4_1(X1, X2, X3, X4);
    var V2 = suma_aux4(X1, X2, X3, X4);
    return suma(V1, V2);
}

/* Auxiliar para «suma5». */
function suma_aux5(var X1, var X2,var X3, var X4, var X5){
    var V1 = U5_2(X1, X2, X3, X4, X5);
    var V2 = U5_3(X1, X2, X3, X4, X5);
    var V3 = U5_4(X1, X2, X3, X4, X5);
    var V4 = U5_5(X1, X2, X3, X4, X5);
    return suma4(V1, V2, V3, V4);
}

/* Devuelve la suma de X1, X2, X3, X4 y X5. */
function suma5(var X1, var X2, var X3, var X4, var X5){
    var V1 = U5_1(X1, X2, X3, X4, X5);
    var V2 = suma_aux5(X1, X2, X3, X4, X5);
    return suma(V1, V2);
}

/* ---------------------------------------------- */

/* Función auxiliar para la función «resta».
 * devuelve X1 - 1. */
function anterior(var X1){
    if(X1 == 0){
        return cero();
    }
    else{
        X1--;
        var rr = anterior(X1);
        return U2_1(X1, rr);
    }
}

```

```
228 /* Función auxiliar para la función «resta». */
229 function resta_aux2(var X1, var X2, var X3){
230     var V1 = U3_2(X1, X2, X3);
231     return anterior(V1);
232 }
233
234 /* Función auxiliar para la función «resta». */
235 function resta_aux1(var X1, var X2){
236     if(X1 == 0){
237         return U1_1(X2);
238     }
239     else{
240         X1--;
241         var rr = resta_aux1(X1, X2);
242         return resta_aux2(X1, rr, X2);
243     }
244 }
245
246 /* Devuelve X1 menos X2. */
247 function resta(var X1, var X2){
248     var V1 = U2_2(X1, X2);
249     var V2 = U2_1(X1, X2);
250     return resta_aux1(V1, V2);
251 }
252
253 /* ---------------------------------------------- */
254
255 /* Función auxiliar para la función «mult». */
256 function mult_aux(var X1, var X2, var X3){
257     var V1 = U3_2(X1, X2, X3);
258     var V2 = U3_3(X1, X2, X3);
259     return suma(V1, V2);
260 }
261
262
263
264
265
```

```
/* Devuelve la multiplicación de X1 y X2. */
function mult(var X1, var X2){
    if(X1 == 0){
        return cero(X2);
    }
    else{
        X1--;
        var rr = mult(X1, X2);
        return mult_aux(X1, rr, X2);
    }
}

/* ---------------------------------------------- */

/* Función auxiliar para la función «mult_veces». */
function mult_veces_aux(var X1, var X2, var X3, var X4){
    var V1 = U4_2(X1, X2, X3, X4);
    var V2 = U4_3(X1, X2, X3, X4);
    return mult(V1, V2);
}

/* Devuelve X3 multiplicado X1 veces por X2;
 * esto es, X3 * (X2 ^ X1). */
function mult_veces(var X1, var X2, var X3){
    if(X1 == 0){
        return U2_2(X2, X3);
    }
    else{
        X1--;
        var rr = mult_veces(X1, X2, X3);
        return mult_veces_aux(X1, rr, X2, X3);
    }
}

/* ---------------------------------------------- */
```

```
304 /* Devuelve 0 si X1 es mayor que 0 y 1 si es 0. */
305 function no_sg(var X1){
306     var V1 = uno(X1);
307     var V2 = U1_1(X1);
308     return resta(V1, V2);
309 }
310
311 /* Devuelve 0 si X1 es 0 y 1 si es mayor que 0. */
312 function sg(var X1){
313     var V1 = uno(X1);
314     var V2 = no_sg(X1);
315     return resta(V1, V2);
316 }
317
318 /* Devuelve 1 si X1 es menor o igual que X2;
319  * 0 en caso contrario. */
320 function menor_ig(var X1, var X2){
321     var V1 = resta(X1, X2);
322     return no_sg(V1);
323 }
324
325 /* Devuelve 1 si X1 es mayor o igual que X2;
326  *  0 en caso contrario. */
327 function mayor_ig(var X1, var X2){
328     var V1 = U2_2(X1, X2);
329     var V2 = U2_1(X1, X2);
330     return menor_ig(V1, V2);
331 }
332
333 /* Devuelve 1 si X1 es igual que X2; 0 en caso contrario. */
334 function igual(var X1, var X2){
335     var V1 = mayor_ig(X1, X2);
336     var V2 = menor_ig(X1, X2);
337     return mult(V1, V2);
338 }
339
340 /* ---------------------------------------------- */
341
```

```
/* Declaración para funciones que usan recursión indirecta. */
function div_emp(var, var, var, var);

/* Función auxiliar para «div».
 * Comprueba si dividendo es mayor que el divisor. */
function div_aux_mayor_ig(var X1, var X2, var X3){
    var V1 = U3_2(X1, X2, X3);
    var V2 = U3_3(X1, X2, X3);
    return mayor_ig(V1, V2);
}

/* Función auxiliar para «div».
 * Suma uno al resultado provisional. */
function div_aux_sig(var X1, var X2,var X3,var X4,var X5){
    var V1 = U5_3(X1, X2, X3, X4, X5);
    return S(V1);
}

/* Función auxiliar para «div».
 * Resta divisor al dividendo. */
function div_aux_rt(var X1, var X2, var X3,var X4,var X5){
    var V1 = U5_4(X1, X2, X3, X4, X5);
    var V2 = U5_5(X1, X2, X3, X4, X5);
    return resta(V1, V2);
}

/* Función auxiliar para «div».
 * Suma uno al resultado, resta al
 * dividendo el divisor y empieza el ciclo. */
function div_aux2(var X1, var X2, var X3, var X4, var X5){
    var V1 = div_aux_sig(X1, X2, X3, X4, X5);

    var V3 = div_aux_rt(X1, X2, X3, X4, X5);
    var V2 = U5_5(X1, X2, X3, X4, X5);
    return div_emp(V1, V2, V3);
}
```

```
/* Función auxiliar para «div».
 * X1: ¿es dividendo mayor que divisor?
 * X2: resultado provisional.
 * X3: dividendo.
 * X4: divisor. */
function div_aux1(var X1, var X2, var X3, var X4){
    if(X1 == 0){
        return U3_1(X2, X3, X4);
    }
    else{
        X1--;
        var rr = div_aux1(X1, X2, X3, X4);
        return div_aux2(X1, rr, X2, X3, X4);
    }
}

/* Función auxiliar para «div».
 * X1: resultado provisional.
 * X2: dividendo.
 * X3: divisor. */
function div_emp(var X1, var X2, var X3){
    var V1 = div_aux_mayor_ig(X1, X2, X3);
    var V2 = U3_1(X1, X2, X3);
    var V3 = U3_2(X1, X2, X3);
    var V4 = U3_3(X1, X2, X3);
    return div_aux1(V1, V2, V3, V4);
}

/* Devuelve n, donde n = X1/X2.
 * X1: dividendo.
 * X2: divisor. */
function div(var X1, var X2){
    var V1 = cero(X1, X2);
    var V2 = U2_1(X1, X2);
    var V3 = U2_2(X1, X2);
    return div_emp(V1, V2, V3);
}

```

```
418 /* ---------------------------------------------- */
419
420 /* Función auxiliar para «resto».
421  * Devuelve multiplicación de división y divisor. */
422 function resto_aux2(var X1, var X2, var X3){
423     var V1 = U3_1(X1, X2, X3);
424     var V2 = U3_3(X1, X2, X3);
425     return mult(V1, V2);
426 }
427
428 /* Función auxiliar para «resto». Resta al
429  * dividendo la multiplicación de división y divisor. */
430 function resto_aux1(var X1, var X2, var X3){
431     var V1 = U3_2(X1, X2, X3);
432     var V2 = resto_aux2(X1, X2, X3);
433     return resta(V1, V2);
434 }
435
436 /* Devuelve el resto de X1/X2, esto es, X1 mod X2.
437  * X1: dividendo.
438  * X2: divisor. */
439 function resto(var X1, var X2){
440     var V1 = div(X1, X2);
441     var V2 = U2_1(X1, X2);
442     var V3 = U2_2(X1, X2);
443     return resto_aux1(V1, V2, V3);
444 }
445
446 /* ---------------------------------------------- */
447
448 /* Devuelve 1 si X1 es divisible entre X2;
449  * 0 en caso contrario. */
450 function es_div(var X1, var X2){
451     var V1 = resto(X1, X2);
452     return no_sg(V1);
453 }
454
455
```

```
456 /* Devuelve 1 si X1 no es divisible
457  * entre X2; 0 en caso contrario. */
458 function no_es_div(var X1, var X2){
459     var V1 = es_div(X1, X2);
460     return no_sg(V1);
461 }
462
463 /* ---------------------------------------------- */
464
465 /* Declaración para funciones que usan recursión indirecta. */
466 function vdiv_emp(var, var, var);
467
468 /* Función auxiliar para «vdiv».
469  * Devuelve 1 si es divisible X2 entre X3; si no 0. */
470 function vdiv_aux_es_div(var X1, var X2, var X3){
471     var V1 = U3_2(X1, X2, X3);
472     var V1 = U3_3(X1, X2, X3);
473     return es_div(V1, V2);
474 }
475
476 /* Función auxiliar para «vdiv».
477  * Divide X4 entre X5. */
478 function vdiv_aux_div(var X1,var X2,var X3,var X4,var X5){
479     var V1 = U5_4(X1, X2, X3, X4, X5);
480     var V2 = U5_5(X1, X2, X3, X4, X5);
481     return div(V1, V2);
482 }
483
484 /* Función auxiliar para «vdiv».
485  * Suma 1 al resultado provisional y divide
486  * dividendo entre divisor. */
487 function vdiv_aux2(var X1, var X2, var X3, var X4,var X5){
488     var V1 = div_aux_sig(X1, X2, X3, X4, X5);
489     var V2 = vdiv_aux_div(X1, X2, X3, X4, X5);
490     var V3 = U5_5(X1, X2, X3, X4, X5);
491     return vdiv_emp(V1, V2, V2);
492 }
493
```

```
494 /* Función auxiliar para «vdiv». Devuelve el resultado
495  * si X3 no es divisible entre X4; si no, sigue el ciclo. */
496 function vdiv_aux1(var X1, var X2, var X3, var X4){
497     if(X1 == 0){
498         return U3_1(X2, X3, X4);
499     }
500     else{
501         X1--;
502         var rr = vdiv_aux1(X1, X2, X3, X4);
503         return vdiv_aux2(X1, rr, X2, X3, X4);
504     }
505 }
506
507 /* Función auxiliar para «vdiv». Comienza el ciclo. */
508 function vdiv_emp(var X1, var X2, var X3){
509     var V1 = vdiv_aux_es_div(X1, X2, X3);
510     var V2 = U3_1(X1, X2, X3);
511     var V3 = U3_2(X1, X2, X3);
512     var V4 = U3_3(X1, X2, X3);
513     return vdiv_aux1(V1, V2, V3, V4);
514 }
515
516 /* Devuelve las veces que es divisible X1 entre X2. */
517 function vdiv(var X1, var X2){
518     var V1 = cero(X1, X2);
519     var V2 = U2_1(X1, X2);
520     var V3 = U2_2(X1, X2);
521     return vdiv_emp(V1, V2, V3);
522 }
523
524 /* ---------------------------------------------- */

525 /* Función auxiliar para «nd_hasta_aux2».
526  * Suma 2 para que no se compruebe si es divisible entre 0 o 1.*/
527 function nd_hasta_aux3(var X1, var X2, var X3){
528     var V1 = U3_1(X1, X2, X3);
529     var V2 = dos(X1, X2, X3);
530     return suma(V1, V2);
531 }
```

```
532 /* Función auxiliar para «nd_hasta_aux1». */
533 function nd_hasta_aux2(var X1, var X2, var X3){
534     var V1 = U3_3(X1, X2, X3);
535     var V2 = nd_hasta_aux3(X1, X2, X3);
536     return no_es_div(V1, V2);
537 }
538
539 /* Función auxiliar para «no_div_hasta». */
540 function nd_hasta_aux1(var X1, var X2, var X3){
541     var V2 = nd_hasta_aux2(X1, X2, X3);
542     var V1 = U3_2(X1, X2, X3);
543     return mult(V1, V2);
544 }
545
546 /* Función auxiliar para «es_primo».
547  * Devuelve 1 si no hay divisor de X2 entre 2 y X1 + 1. */
548 function no_div_hasta(var X1, var X2){
549     if(X1 == 0){
550         return uno(X2);
551     }
552     else{
553         X1--;
554         var rr = no_div_hasta(X1, X2);
555         return nd_hasta_aux1(X1, rr, X2);
556     }
557 }
558
559 /* Función auxiliar para «es_primo». */
560 function partir(var X1){
561     var V1 = anterior(X1);
562     var V2 = dos(X1);
563     return div(V1, V2);
564 }
565
566
567
568
569
```

```
570 /* Devuelve 1 si X1 es primo; 0 en caso contrario. */
571 function es_primo(var X1){
572     var V1 = partir(X1);
573     var V2 = U1_1(X1);
574     return no_div_hasta(V1, V2);
575 }
576
577 /* Devuelve 0 si X1 es primo; 1 en caso contrario. */
578 function no_es_primo(var X1){
579     var V1 = es_primo(X1);
580     return no_sg(V1);
581 }
582
583 /* ---------------------------------------------- */
584
585 /* Declaración de función. */
586 function sig_primo_desde(var);
587
588 /* Función auxiliar para «sig_primo_aux1». */
589 function sig_primo_aux2(var X1, var X2, var X3){
590     var V1 = U3_2(X1, X2, X3);
591     return sig_primo_desde(V1);
592 }
593
594 /* Si X2 es primo lo devuelve; si no
595  * sigue buscando el menor primo mayor que X2. */
596 function sig_primo_aux1(var X1, var X2){
597     if(X1 == 0){
598         return U1_1(X2);
599     }
600     else{
601         X1--;
602         var rr = sig_primo_aux1(X1, X2);
603         return sig_primo_aux2(X1, rr, X2);
604     }
605 }
606
607
```

```
608 /* Devuelve el menor primo mayor o igual que X1. */
609 function sig_primo(var X1){
610     var V1 = no_es_primo(X1);
611     var V2 = U1_1(X1);
612     return sig_primo_aux1(V1, V2);
613 }
614
615 /* Devuelve el menor primo mayor que X1. */
616 function sig_primo_desde(var X1){
617     var V1 = S(X1);
618     return sig_primo(V1);
619 }
620
621 /* Devuelve el menor primo mayor que el primo X2. */
622 function inst_primo_aux(var X1, var X2){
623     var V1 = U2_2(X1, X2);
624     return sig_primo_desde(V1);
625 }
626
627 /* Devuelve el primo con el que se
628  * codifica la instrucción en la posición X1. */
629 function inst_primo(var X1){
630     if(X1 == 0){
631         return dos();
632     }
633     else{
634         X1--;
635         var rr = inst_primo(X1);
636         return inst_primo_aux(X1, rr);
637     }
638 }

639 /* ---------------------------------------------- */
640 /* Devuelve el primo correspondiente a la posición X1.
641  * X1: posición. */
642 function primo_num_inst(var X1, var X2){
643     var V1 = U2_1(X1, X2);
644     return inst_primo(V1);
645 }
```

```
646 /* Devuelve la instrucción en la posición X1.
647  * X1: posición.
648  * X2: registro. */
649 function inst(var X1, var X2){
650     var V1 = U2_2(X1, X2);
651     var V2 = primo_num_inst(X1, X2);
652     return vdiv(V1, V2);
653 }
654
655 /* Devuelve 1 si X1 es instrucción; 0 en caso contrario.
656  * X1: instrucción. */
657 function hay_inst(var X1, var X2){
658     var V1 = U2_1(X1, X2);
659     return sg(V1);
660 }
661
662 /* Devuelve: 0 si no hay instrucción o ésta es '1';
663  * 1 si '0'.
664  * 2 si '='.
665  * 3 si '*'. */
666 function tipo_inst_aux(var X1, var X2){
667     var V1 = U2_1(X1, X2);
668     var V2 = cuatro(X1, X2);
669     return resto(V1, V2);
670 }
671
672 /* Devuelve, según la instrucción en X1:
673  * 0 si no hay instrucción.
674  * 1 si '1'.
675  * 2 si '0'.
676  * 3 si '='.
677  * 4 si '*'. */
678 function tipo_inst(var X1, var X2){
679     var V1 = tipo_inst_aux(X1, X2);
680     var V2 = hay_inst(X1, X2);
681     return suma(V1, V2);
682 }
683
```

```
684 /* Auxiliar para «pos_inst».
685  * suma 3 para que al dividir entre 4
686  * no dé nunca 0 y dé el resultado correcto. */
687 function pos_inst_aux(var X1, var X2){
688     var V1 = U2_1(X1, X2);
689     var V2 = tres(X1, X2);
690     return suma(V1, V2);
691 }
692
693 /* Devuelva la posición a la
694  * que se refiere la instrucción X1. */
695 function pos_inst(var X1, var X2){
696     var V1 = pos_inst_aux(X1, X2);
697     var V2 = cuatro(X1, X2);
698     return div(V1, V2);
699 }
700
701 /* Devuelva la posición a la que
702   * señala el puntero en el registro X1. */
703 function sacar_puntero(var X1){
704     var V1 = U1_1(X1);
705     var V2 = dos(X1);
706     return vdiv(V1, V2);
707 }
708
709 /* Devuelve el registro con
710  * el puntero avanzado una posición. */
711 function avanzar_inst(var X1, var X2){
712     var V1 = dos(X1, X2);
713     var V2 = U2_2(X1, X2);
714     return mult(V1, V2);
715 }
716
717 /* ---------------------------------------------- */
718
719
720
721
```

```
722 /* Devuelve el registro X2 modificado con una marca más
723  * en la posición X1, o un símbolo «1» si estaba vacía,
724  * y el puntero señalando a la siguiente posición. */
725 function marcar(var X1, var X2){
726     var V1 = cuatro(X1, X2);
727     var V2 = primo_num_inst(X1, X2);
728     var V3 = avanzar_inst(X1, X2);
729     return mult_veces(V1, V2, V3);
730 }
731 /* ---------------------------------------------- */
732
733 /* Auxiliar para «borrar_aux2». Resta para mantener
734  * el símbolo en la posición referida si éste no es '1'. */
735 function borrar_aux3(var X1, var X2){
736     var V1 = U2_1(X1, X2);
737     var V2 = tipo_inst_aux(X1, X2);
738     return resta(V1, V2);
739 }
740
741 /* Auxiliar para «borrar_aux».
742  * X1: instrucción en posición referida.
743  * X2: primo en posición referida. */
744 function borrar_aux2(var X1, var X2){
745     var V1 = borrar_aux3(X1, X2);
746     var V2 = U2_2(X1, X2);
747     var V2 = uno(X1, X2);
748     return mult_veces(V1, V2, V3);
749 }
750
751 /* Devuelve el número entre el que hay que dividir
752  * el registro X2 para borrar la posición X1.
753  * X1: posición referida.
754  * X2: registro. */
755 function borrar_aux1(var X1, var X2){
756     var V1 = inst(X1, X2);
757     var V2 = primo_num_inst(X1, X2);
758     return borrar_aux2(V1, V2);
759 }
```

```
760 /* Devuelve el registo con la posición X1 borrada.
761  * X1: posición referida.
762  * X2: registro. */
763 function borrar(var X1, var X2){
764     var V1 = avanzar_inst(X1, X2);
765     var V2 = borrar_aux1(X1, X2);
766     return div(V1, V2);
767 }
768
769 /* ---------------------------------------------- */
770
771
772 /* Devuelve la posición del puntero en registro X2. */
773 function quitar_puntero_aux2(var X1, var X2){
774     var V1 = U2_2(X1, X2);
775     return sacar_puntero(V1);
776 }
777
778 /* Devuelve el número entre el que hay
779  *  que dividir registro X2 para quitarle el puntero. */
780 function quitar_puntero_aux1(var X1, var X2){
781     var V1 = quitar_puntero_aux2(X1, X2);
782     var V2 = dos(X1, X2);
783     var V3 = uno(X1, X2);
784     return mult_veces(V1, V2, V3);
785 }
786
787 /* Devuelve el registro X2 sin puntero. */
788 function quitar_puntero(var X1, var X2){
789     var V1 = quitar_puntero_aux1(X1, X2);
790     var V2 = U2_2(X1, X2);
791     return div(V1, V2);
792 }
793
794
795
796
797
```

```
798 /* Devuelve el registro X2 con el
799  * puntero señalando a la posición X1. */
800 function saltar(var X1, var X2){
801     var V1 = U2_1(X1, X2);
802     var V2 = dos(X1, X2);
803     var V3 = quitar_puntero(X1, X2);
804     return mult_veces(V1, V2, V3);
805 }
806
807 /* ---------------------------------------------- */
808
809
810 /* Devuelve la instrucción
811  * en la primera posición de registro X2. */
812 function inst_uno(var X1, var X2){
813     var V1 = uno(X1, X2);
814     var V2 = U2_2(X1, X2);
815     return inst(V1, V2);
816 }
817
818 /* Devuelve la posición referida por la
819  * instrucción en la primera posición de registro X2. */
820 function v_pos_uno(var X1, var X2){
821     var V1 = inst_uno(X1, X2);
822     var V2 = U2_2(X1, X2);
823     return pos_inst(V1, V2);
824 }
825
826 /* Devuelve la posición referida por la
827  * instrucción X1 de registro X2. */
828 function v_pos(var X1, var X2){
829     var V1 = inst(X1, X2);
830     var V2 = U2_2(X1, X2);
831     return pos_inst(V1, V2);
832 }
833
834
835
```

```
836 /* Devuelve 1 si la posición X1 de registro X2
837  * es igual que la primera; 0 en caso contrario. */
838 function hay_salto(var X1, var X2){
839     var V1 = v_pos_uno(X1, X2);
840     var V2 = v_pos(X1, X2);
841     return es_igual(V1, V2);
842 }
843
844 /* Devuelve el registro X2 con el puntero
845  * señalando a la siguiente posición de la actual si
846  * la posición X1 es distinta de la primera posición;
847  * a la siguiente de la siguiente en caso contrario. */
848 function comparar(var X1, var X2){
849     var V1 = hay_salto(X1, X2);
850     var V2 = dos(X1, X2);
851     var V3 = avanzar_inst(X1, X2);
852     return mult_veces(V1, V2, V3);
853 }
854
855 /* ---------------------------------------------- */
856
857
858 /* Devuelve el registro X3 después de ejecutar 'marcar'. */
859 function eval_marcar_aux(var X1, var X2, var X3){
860     var V1 = U3_2(X1, X2, X3);
861     var V2 = U3_3(X1, X2, X3);
862     return marcar(V1, V2);
863 }
864
865 /* Si el tipo de instrucción X1 es 'marcar',
866  * devuelve 1; 0 en caso contrario. */
867 function es_marcar(var X1, var X2, var X3){
868     var V1 = U3_1(X1, X2, X3);
869     var V2 = uno(X1, X2, X3);
870     return es_igual(V1, V2);
871 }
872
873
```

```
874 /* Si X1 es 1 devuelve el registro X3
875  * después de ejecutar la instrucción 'marcar'
876  * en la posición X2; devuelve 0 si X1 no es 1. */
877 function eval_marcar(var X1, var X2, var X3){
878     var V1 = es_marcar(X1, X2, X3);
879     var V2 = eval_marcar_aux(X1, X2, X3);
880     return mult(V1, V2);
881 }
882
883 /* ---------------------------------------------- */
884
885 /* Devuelve el registro X3 después de ejecutar 'borrar'. */
886 function eval_borrar_aux(var X1, var X2, var X3){
887     var V1 = U3_2(X1, X2, X3);
888     var V2 = U3_3(X1, X2, X3);
889     return borrar(V1, V2);
890 }
891
892 /* Si el tipo de instrucción X1 es
893  * 'borrar', devuelve 1; 0 en caso contrario. */
894 function es_borrar(var X1, var X2, var X3){
895     var V1 = U3_1(X1, X2, X3);
896     var V2 = dos(X1, X2, X3);
897     return es_igual(V1, V2);
898 }
899
900 /* Si X1 es 2 devuelve el registro X3
901  * después de ejecutar la instrucción 'borrar'
902  * en la posición X2; devuelve 0 si X1 no es 2. */
903 function eval_borrar(var X1, var X2, var X3){
904     var V1 = es_borrar(X1, X2, X3);
905     var V2 = eval_borrar_aux(X1, X2, X3);
906     return mult(V1, V2);
907 }
908
909 /* ---------------------------------------------- */
910
911
```

```
912 /* Devuelve el registro X3 después de ejecutar 'comparar'. */
913 function eval_comparar_aux(var X1, var X2, var X3){
914     var V1 = U3_2(X1, X2, X3);
915     var V2 = U3_3(X1, X2, X3);
916     return comparar(V1, V2);
917 }
918
919 /* Si el tipo de instrucción X1 es
920  * 'comparar', devuelve 1; 0 en caso contrario. */
921 function es_comparar(var X1, var X2, var X3){
922     var V1 = U3_1(X1, X2, X3);
923     var V2 = tres(X1, X2, X3);
924     return es_igual(V1, V2);
925 }
926
927 /* Si X1 es 3 devuelve el registro X3
928  * después de ejecutar la instrucción 'comparar'
929  * en la posición X2; devuelve 0 si X1 no es 3. */
930 function eval_comparar(var X1, var X2, var X3){
931     var V1 = es_comparar(X1, X2, X3);
932     var V2 = eval_comparar_aux(X1, X2, X3);
933     return mult(V1, V2);
934 }
935 /* ---------------------------------------------- */
936 /* Devuelve el registro X3 después de ejecutar 'saltar'. */
937 function eval_saltar_aux(var X1, var X2, var X3){
938     var V1 = U3_2(X1, X2, X3);
939     var V2 = U3_3(X1, X2, X3);
940     return saltar(V1, V2);
941 }
942
943 /* Si el tipo de instrucción X1 es 'saltar', devuelve 1;
944  * 0 en caso contrario. */
945 function es_saltar(var X1, var X2, var X3){
946     var V1 = U3_1(X1, X2, X3);
947     var V2 = cuatro(X1, X2, X3);
948     return es_igual(V1, V2);
949 }
```

```
950 /* Si X1 es 4 devuelve el registro X3
951  * después de ejecutar la instrucción 'saltar'
952  * en la posición X2; devuelve 0 si X1 no es 4. */
953 function eval_saltar(var X1, var X2, var X3){
954     var V1 = es_saltar(X1, X2, X3);
955     var V2 = eval_saltar_aux(X1, X2, X3);
956     return mult(V1, V2);
957 }
958
959
960 /* ---------------------------------------------- */
961
962
963 /* Función auxiliar para «eval_no_inst». */
964 function eval_no_inst_aux(var X1, var X2, var X3){
965     var V1 = U3_1(X1, X2, X3);
966     return no_sg(V1);
967 }
968
969 /* Devuelve el registro X3 sin modificar
970  * si el tipo de instrucción X1 es 0;
971  * devuelve 0 en caso contrario. */
972 function eval_no_inst(var X1, var X2, var X3){
973     var V1 = no_inst_aux(X1, X2, X3);
974     var V2 = U3_3(X1, X2, X3);
975     return mult(V1, V2);
976 }
977
978
979 /* ---------------------------------------------- */
980
981
982 /* Devuelve 1 si hay instrucción en X1; 0 en caso contrario . */
983 function hay_inst(var X1, var X2, var X3){
984     var V1 = U3_1(X1, X2, X3);
985     return sg(V1);
986 }
987
```

```
988 /* Devuelve el registro después de ejecutar la intrucción
989  * que indica X1 o el registro sin tocar si X1 es 0. */
990 function eval_registro(var X1, var X2, var X3){
991     var V1 = eval_marcar(X1, X2, X3);
992     var V2 = eval_borrar(X1, X2, X3);
993     var V3 = eval_comparar(X1, X2, X3);
994     var V4 = eval_saltar(X1, X2, X3);
995     var V5 = eval_no_inst(X1, X2, X3);
996     return suma(V1, V2, V3, V4, V5);
997 }
998
999 /* Comprueba si hay instrucción, ejecuta
1000  * en caso de haberla y vuelve a empezar el ciclo. */
1001 function ejec3(var X1, var X2, var X3){
1002     var V1 = hay_inst(X1, X2, X3);
1003     var V2 = eval_registro(X1, X2, X3);
1004     return evaluar(V1, V2);
1005 }
1006
1007 /* Saca el tipo de instrucción, la
1008  * posición referida y llama a «ejec3».
1009  * X1: instrucción.
1010  * X2: registro. */
1011 function ejec2(var X1, var X2){
1012     var V1 = tipo_inst(X1, X2);
1013     var V2 = pos_inst(X1, X2);
1014     var V3 = U2_2(X1, X2);
1015     return ejec3(V1, V2, V3);
1016 }
1017
1018 /* Saca la instrucción y llama a «ejec2» con ésta y el registro X2.
1019  * X1: puntero.
1020  * X2: registro. */
1021 function ejec1(var X1, var X2){
1022     var V1 = inst(X1, X2);
1023     var V2 = U2_2(X1, X2);
1024     return ejec2(V1, V2);
1025 }
```

```
1026 /* Toma el registro X1 y llama a «ejec1»
1027  * con éste y el puntero contenido en él. */
1028 function ejecutar_inst(var X1){
1029     var V1 = sacar_puntero(X1);
1030     var V2 = U1_1(X1);
1031     return ejec1(V1, V2);
1032 }
1033
1034 /* X3: registro. */
1035 function ejecutar(var X1, var X2, var X3){
1036     var V1 = U3_3(X1, X2, X3);
1037     return ejecutar_inst(V1);
1038 }
1039
1040 /*
1041  * Si X1 es 0 devuelve el registro y acaba la computación;
1042  * si es 1 ejecuta la siguiente instrucción, si la hubiera,
1043  * en registro X2; es decir, ejecuta el siguienet paso.
1044  */
1045 function evaluar(var X1, var X2){
1046     if(X1 == 0){
1047         return U1_1(X2);
1048     }
1049     else{
1050         X1--;
1051         var rr = evaluar(X1, X2);
1052         return ejecutar(X1, rr, X2);
1053     }
1054 }
1055
1056 /* Auxiliar para «computar». */
1057 function computar_aux(var X1, var X2){
1058     var V1 = uno(X1, X2);
1059     var V2 = mult(X1, X2);
1060     return evaluar(V1, V2);
1061 }
1062
1063
```

```
1064 /*
1065  * Función recursiva primitiva que computa el mismo valor
1066  * que el programa C-- que recibe como argumento; en caso de termina
1067  * devuelve un registro con el puntero y programa después del cómput
1068  */
1069 function computar(var X1){
1070     var V1 = dos(X1);
1071     var V2 = U1_1(X1);
1072     return computar_aux(V1, V2);
1073 }
```

Apéndice **E**

Macroinstrucciones de C--

⟨*nd*⟩ → **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

⟨*ndc*⟩ → **0** | ⟨*nd*⟩

⟨*ndcc*⟩ → ε | ⟨*ndc*⟩⟨*ndcc*⟩

⟨*natural*⟩ → ⟨*nd*⟩⟨*ndcc*⟩

letra minúscula → ⟨*natural*⟩

⟨*id*⟩ → *cadena de caracteres*

Ω | Φ | Ψ → ⟨*natural*⟩ | **.**⟨*id*⟩ | **:**⟨*id*⟩

x → **1** | **0** | = | ∗

⟨*preinstrucción*⟩ → x^{Ω}

⟨*preinstrucciones*⟩ → ε | ⟨*preinstrucción*⟩⟨*preinstrucciones*⟩

No terminales en cursiva, entre '⟨⟩' o no, o letra griega.
Terminales en negrita.
ε – cadena vacía.
| – alternativas entre entidades.
Letra minúscula
Letras minúsculas de alfabeto español (a, b, ..., z).
cadena de caracteres
Concatenación de más de una *letra minúscula* y/o número.

Postulado 1

Es verdad:

- 2 = 1 + 1
- 3 = 2 + 1
- 4 = 3 + 1
- 5 = 4 + 1
- 6 = 5 + 1
- 7 = 6 + 1
- 8 = 7 + 1
- 9 = 8 + 1

Postulado 1'

Si $n = m + 1$ es verdad, $k = m + 1$ es verdad si y sólo si n y k son el mismo ⟨*natural*⟩.

Postulado 1''

$n⟨ndc_1⟩ = m⟨ndc_2⟩ + 1$ es verdad si y sólo si una de las siguientes afirmaciones es verdad:

- n y m son el mismo ⟨*natural*⟩, $⟨ndc_1⟩$ es 1 y $⟨ndc_2⟩$ es 0.
- n y m son el mismo ⟨*natural*⟩, $⟨ndc_1⟩$ no es 0, $⟨ndc_2⟩$ no es 0 y $⟨ndc_1⟩ = ⟨ndc_2⟩ + 1$ es verdad.
- $⟨ndc_1⟩$ es 0, $⟨ndc_2⟩$ es 9, y $n = m + 1$ es verdad.

✎ No terminales con un ⟨*natural*⟩ como subíndice (p.ej. $⟨ndc_1⟩$, m_1) representan una cadena concreta, que puede ser cualquiera, que se pueda derivar de ese no terminal.

x^1 ➡ x

x^n ➡ x^m'' donde $n = m + 1$ es verdad.

Ejemplo

$*^3$ ⇒ $*^2$' ⇒ $*^1$'' ⇒ $*$''

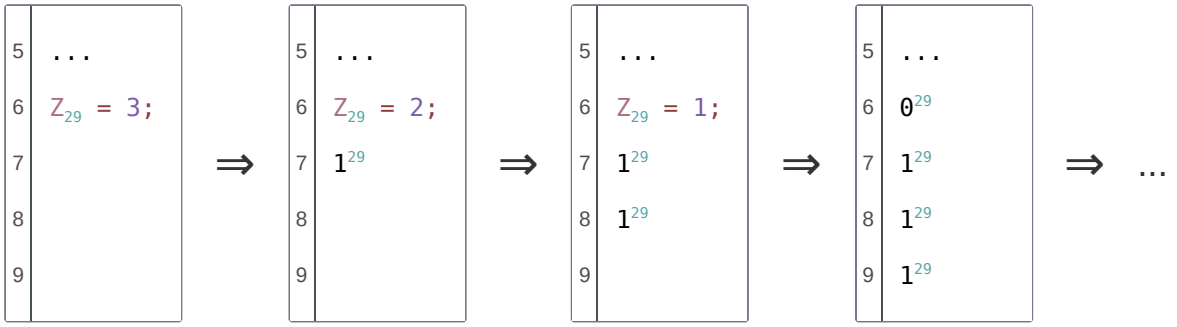
Asignaciones

$Z_Ω$ = 0; ➡ $0^Ω$

$Z_Ω$ = 1; ➡ $0^Ω1^Ω$

$Z_Ω$ = *n*; ➡ $Z_Ω$ = *m*; $1^Ω$ donde $n = m + 1$ es verdad.

Ejemplo



> ✎ Tres puntos (...) representa cualquier instrucción o cualquier expansión de instrucciones.

←:⟨*natural*⟩

Señales que pone el expansor de instrucciones, apuntando a una instrucción.

- La primera señal que pone es **←:1**.
- Si pone una señal **←:*m***, la siguiente señal será **←:*n***, donde n = m + 1 es verdad.

Las asociación de señales a instrucciones es:

- si la señal apunta a una ⟨*preinstrucción*⟩ queda ligada a esa ⟨*preinstrucción*⟩.
- si la señal apunta a una macroinstrucción, y $texto_1$ es el resultado de expandir esta macroinstrucción, después de la expansión apuntará:
  - a la siguiente macroinstrucción o ⟨*preinstrucción*⟩ si $texto_1$ es ε.
  - a la primera macroinstrucción o ⟨*preinstrucción*⟩ en $texto_1$ si $texto_1$ no es ε.

```
INIT ➡ 1
```

```
JUMP ➡ =
```

NADA ➡ JUMP 1

Z_Ω = Z_Ω; ➡ ε

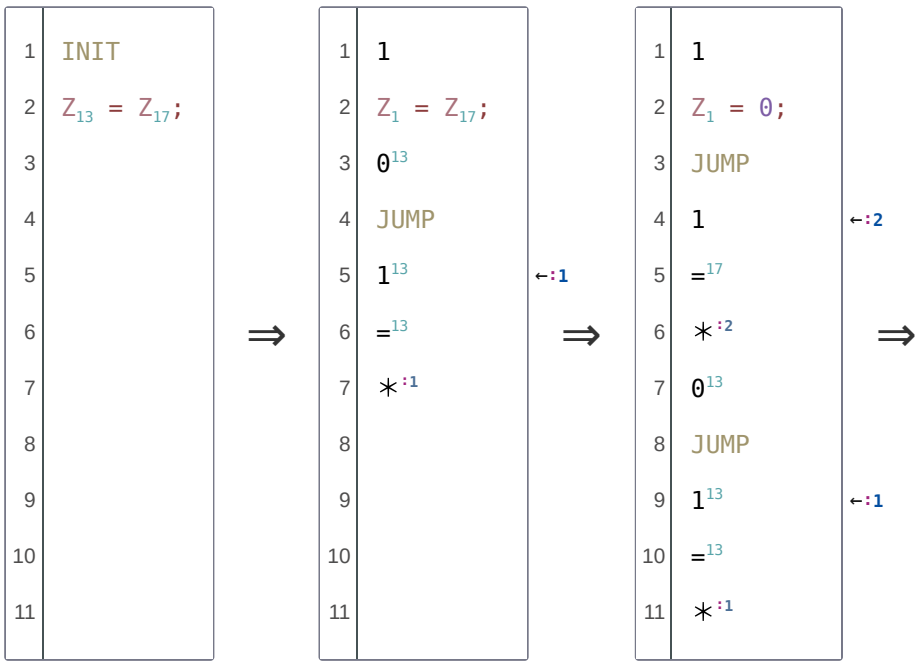
Z_1 = Z_Ω; ➡
Z_1 = 0;
JUMP 1 $^{\Omega}$
$=^{\Omega}$
$*^{:i}$ ←:i

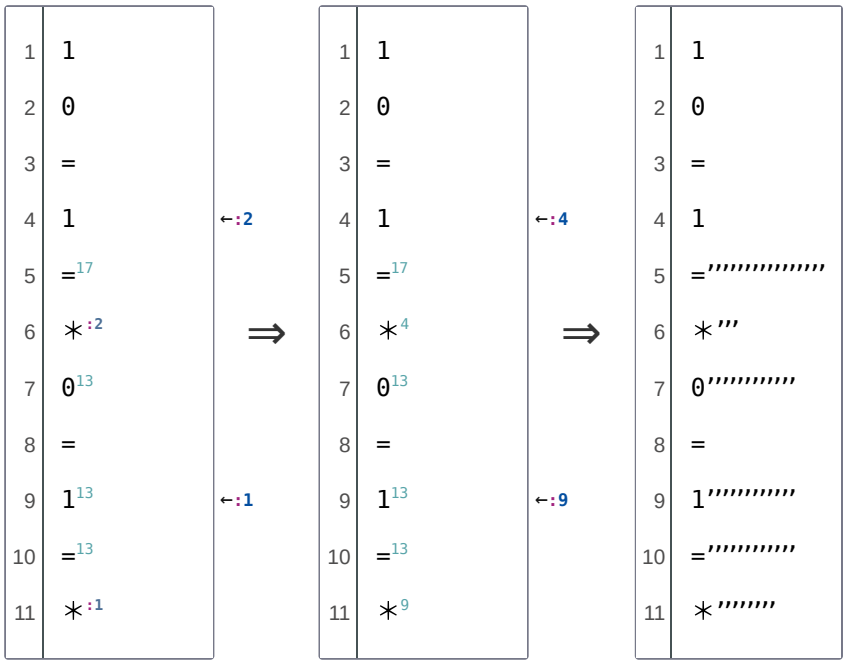
con Ω distinto de 1.

Z_Ω = Z_Φ; ➡
Z_1 = Z_Φ;
0^{Ω}
JUMP 1 $^{\Omega}$
$=^{\Omega}$
$*^{:i}$ ←:i

con Ω distinto de Φ.

Ejemplo



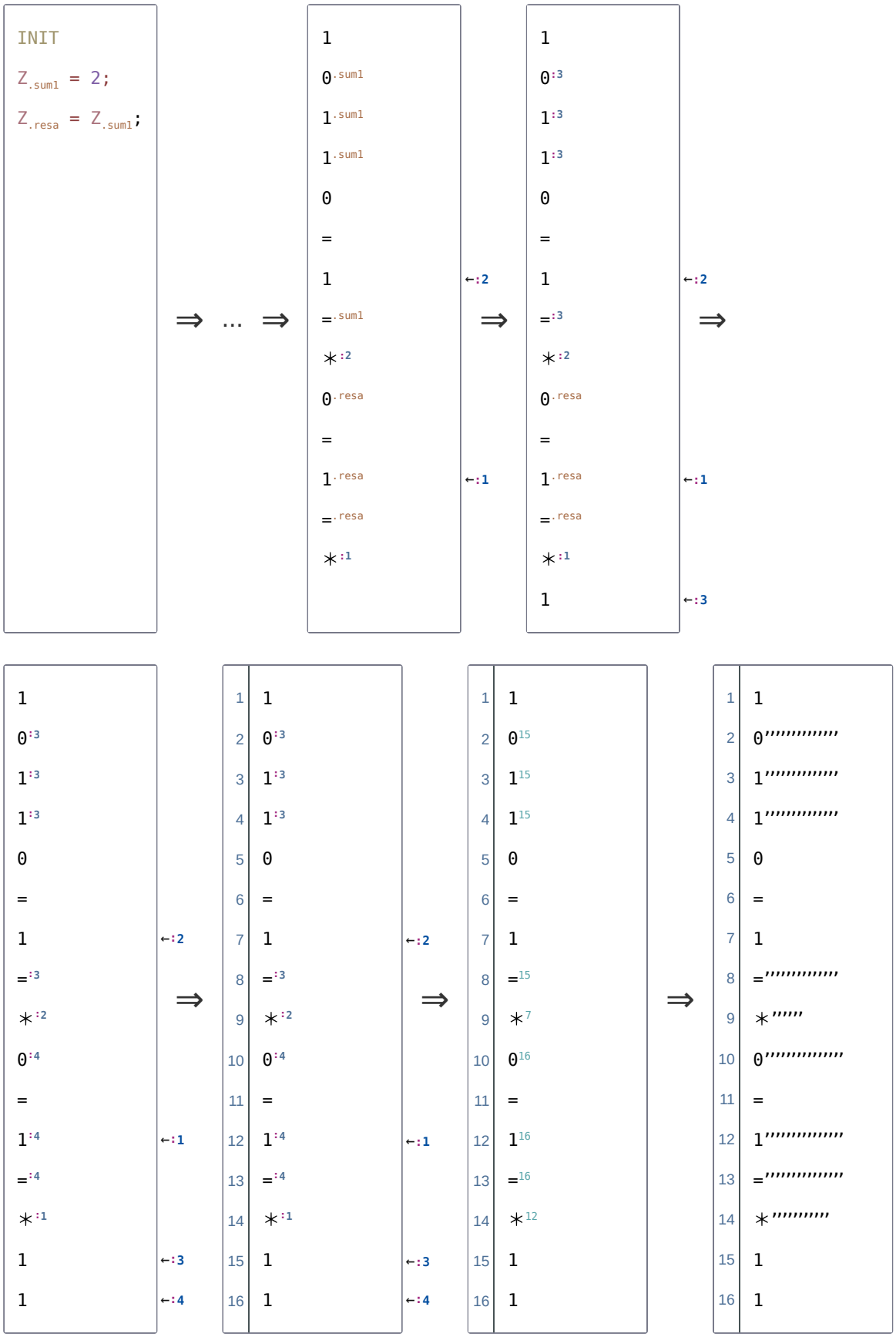


Expansiones finales

Cuando sólo quedan ⟨*preinstrucciones*⟩, el expansor:

- Por cada ⟨*id*⟩ que aparezca en alguna ⟨*preinstrucción*⟩ $x^{.\langle id\rangle}$:
  - añade al final una nueva instrucción ‘1’ y una señal ←:***i*** apuntando a esta instrucción.
  - sustituye todas las apariciones de .⟨*id*⟩ por :***i***.
- Asocia un ⟨*natural*⟩ a cada ⟨*preinstrucción*⟩ en el orden en que aparecen:
  - a la primera ⟨*preinstrucción*⟩ le asocia 1.
  - si a una ⟨*preinstrucción*⟩ le asocia *m*, a la siguiente le asociará *n*, donde $n = m + 1$ es verdad.
- Por cada señal ←:***i*** sustituye en ⟨*macroinstrucciones*⟩ todas las apariciones de :***i*** por *n*, donde *n* es el ⟨*natural*⟩ asociado a la ⟨*preinstrucción*⟩ a la que apunta ←:***i***.
- Expande las ⟨*preinstrucción*⟩ de la forma x^n.

Ejemplo


INIT
Z.sum1 = 2;
Z.resa = Z.sum1;
⇒ ... ⇒
1
0.sum1
1.sum1
1.sum1
0
=
1
=.sum1
∗:2
0.resa
=
1.resa
=.resa
∗:1
←:2
←:1
⇒
1
0:3
1:3
1:3
0
=
1
=:3
∗:2
0.resa
=
1.resa
=.resa
∗:1
1
←:2
←:1
←:3
⇒
1
0:3
1:3
1:3
0
=
1
=:3
∗:2
0:4
=
1:4
=:4
∗:1
1
1
←:2
←:1
←:3
←:4
⇒
1 1
2 0:3
3 1:3
4 1:3
5 0
6 =
7 1
8 =:3
9 ∗:2
10 0:4
11 =
12 1:4
13 =:4
14 ∗:1
15 1
16 1
←:2
←:1
←:3
←:4
⇒
1 1
2 0^15
3 1^15
4 1^15
5 0
6 =
7 1
8 =^15
9 ∗^7
10 0^16
11 =
12 1^16
13 =^16
14 ∗^12
15 1
16 1
⇒
1 1
2 0’’’’’’’’’’’’’’’
3 1’’’’’’’’’’’’’’’
4 1’’’’’’’’’’’’’’’
5 0
6 =
7 1
8 =’’’’’’’’’’’’’’’
9 ∗’’’’’’
10 0’’’’’’’’’’’’’’’
11 =
12 1’’’’’’’’’’’’’’’
13 =’’’’’’’’’’’’’’’
14 ∗’’’’’’’’’’
15 1
16 1

$\lambda \rightarrow \langle natural \rangle \mid Z_{\Omega}$

Operaciones Aritméticas

Z_{Ω} += λ; ➡

```
Z.opi = λ;
Z1 = 0;
∗:j
1                ←:i
1Ω
=.opi            ←:j
∗:i
```

$Z_{\Omega} = Z_{\Phi} + Z_{\Psi}$; ➡

```
ZΩ = ZΦ;
ZΩ += ZΨ;
```

Z_{Ω}++; ➡

```
1Ω;
```

$Z_{\Omega} = Z_{\Phi} - Z_{\Psi}$; ➡

```
ZΩ = 0;
Z1 = 0;
JUMP
Z1++;            ←:i
=Φ
JUMP
∗:j
=Ψ
∗:i
Z1++;            ←:k
ZΩ++;
=Φ
∗:k
...              ←:j
```

Z_Ω--; ➡

```
Z1 = 0;
=Ω
JUMP
*:i
Z.opi= ZΩ;
ZΩ = 0;
JUMP
ZΩ++;          ←:j
Z1++;
=.opi
*:j
...            ←:i
```

$Z_\Omega = Z_\Phi * Z_\Psi$; ➡

```
ZΩ = 0;
Z1 = 0;
*:i
Z1++;          ←:j
Z.cont = Z1;
ZΩ += ZΦ;
Z1 = Z.cont
=Ψ             ←:i
*:j
```

STOP ➡

```
0:i
1   ←:i
```

RETURN; ➡ STOP

RETURN λ; ➡

```
Z1 = λ;
STOP
```

```
Zₒ = Zᵩ / Zᵩ; ➡  Zₒ = 0;
                  Z.da = Zᵩ;
                  Z.da++;
                  Z₁ = 0;
                  =ᵩ
                  ∗:i
                  STOP
                  Zₒ++;          ←:j
                  Z.da -= Zᵩ;    ←:i
                  Z₁ = 0;
                  =.da
                  ∗:j
```

```
Zₒ = Zᵩ % Zᵩ; ➡  Zₒ = Zᵩ / Zᵩ;
                  Z.ra = Zₒ * Zᵩ;
                  Zₒ = Zᵩ - Z.ra;
```

Postulados 2

⟨*natc*⟩ → **0** | ⟨*natural*⟩

𝔞 | 𝔟 | 𝔠 → ⟨*natc*⟩

> **Postulado 1'''**
>
> 𝔞 = 0 + 0 es verdad si y sólo sí 𝔞 es 0.

> **Postulado 1''''**
>
> 1 = 1 + 0 es verdad.

Postulado 1””””

Si $n = m + 0$ es verdad, $n = m + 0$ es verdad, donde:

- $k = n + 1$ es verdad.
- $p = m + 1$ es verdad.

Postulado 1”””””

$\mathfrak{a} = 0 + n$ es verdad si y sólo si $\mathfrak{a} = n + 0$ es verdad.

Postulado 1”””””

$\mathfrak{a} = n + 0$ es verdad, $\mathfrak{c} = n + 0$ si y sólo si $\mathfrak{a}$ y $\mathfrak{c}$ son el mismo ⟨*natc*⟩.

Postulado 1”””””

$0 = n + 1$ no es verdad.

Postulado 1”””””

$\mathfrak{a} = n + m$ es verdad si y sólo si:

- m es 1 y $\mathfrak{a} = n + 1$ es verdad.
- m no es 1 y $\mathfrak{a} = p + k$ es verdad, donde:
  - $p = n + 1$ es verdad.
  - $m = k + 1$ es verdad.

$A \rightarrow \mathbf{a} \mid \mathbf{A}$

$B \rightarrow \mathbf{b} \mid \mathbf{B}$

$C \rightarrow \mathbf{c} \mid \mathbf{C}$

$D \rightarrow \mathbf{d} \mid \mathbf{D}$

$E \rightarrow \mathbf{e} \mid \mathbf{E}$

$A \rightarrow \mathbf{f} \mid \mathbf{F}$

$\langle exal \rangle \rightarrow A \mid B \mid C \mid D \mid E \mid F$

Postulado 1”””””””””

Es verdad:

- $A = 9 + 1$
- $B = A + 1$
- $C = B + 1$
- $D = C + 1$
- $E = D + 1$
- $F = E + 1$

Postulado 1”””””””””””

$\mathfrak{a} = \mathfrak{b} + \langle exal_1 \rangle$ es verdad si y sólo si una de las siguientes afirmaciones se cumple:

- $\langle exal_1 \rangle$ es A y $\mathfrak{a} = \mathfrak{c} + 9$ es verdad, donde $\mathfrak{c} = \mathfrak{b} + 1$ es verdad.
- $\langle exal_1 \rangle$ no es A y $\mathfrak{a} = \mathfrak{c} + \langle exal_2 \rangle$ es verdad, donde:
  - $\mathfrak{c} = \mathfrak{b} + 1$ es verdad.
  - $\langle exal_1 \rangle = \langle exal_2 \rangle + 1$ es verdad.

Postulado 1”””””””””””

$\mathfrak{a} = 0 * 0$ es verdad si y sólo si $\mathfrak{a}$ es 0.

Postulado 1”””””””””””

$0 = 1 * 0$ es verdad.

Postulado 1”””””””””””

Si $0 = n * 0$ es verdad, $0 = m * 0$ es verdad, donde $m = n + 1$ es verdad.

Postulado 1”””””””””””

Si $\mathfrak{a} = n * 0$ es verdad, $\mathfrak{b} = n * 0$ es verdad si y sólo si $\mathfrak{a}$ y $\mathfrak{b}$ son el mismo $\langle natc \rangle$.

Postulado 1”””””””””””””

1 = 1 * 1 es verdad.

Postulado 1”””””””””””””

Si *n* = *m* * 1 *k* = *p* * 1 es verdad, donde:

- *k* = *n* + 1 es verdad.
- *p* = *m* + 1 es verdad.

Postulado 1”””””””””””””

Si $\mathfrak{a}$ = *m* * 1 es verdad, $\mathfrak{b}$ = *m* * 1 es verdad si y sólo si $\mathfrak{a}$ y $\mathfrak{b}$ son el mismo ⟨*natc*⟩.

Postulado 1”””””””””””””

$\mathfrak{a}$ = *m* * *n* es verdad si y sólo si una de las siguientes afirmaciones se cumple:

- *n* es 1 y $\mathfrak{a}$ = *m* * 1 es verdad.
- *n* no es 1 y es verdad:
  - *n* = *k* + 1.
  - $\mathfrak{b}$ = *m* + *k*.
  - $\mathfrak{a}$ = $\mathfrak{b}$ + *m*.

Literales

⟨*signo*⟩ → ε | **+** | **-**

⟨*octd*⟩ → **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7**

⟨*octal*⟩ → ⟨*octal*⟩ | ⟨*octd*⟩⟨*octal*⟩

⟨*ndc*⟩ → ⟨*octd*⟩ | **8** | **9**

⟨*exad*⟩ → ⟨*ndc*⟩ | ⟨*exal*⟩

⟨*exa*⟩ → ⟨*exad*⟩ | ⟨*exad*⟩⟨*exa*⟩

⟨*x*⟩ → **x** | **X**

⟨*nat*⟩ → ⟨*natural*⟩ | **0**⟨*octal*⟩ | ⟨*x*⟩⟨*exa*⟩

⟨*natc*⟩ → ⟨*natural*⟩ | **0**

Z_{Ω} = ⟨$signo_1$⟩0⟨$octd_1$⟩ ➡ Z_{Ω} = ⟨$signo_1$⟩[$⟨$octd_1$⟩]

[$⟨$natc_1$⟩]⟨$octd_1$⟩ ➡ [$⟨$natc_2$⟩]

donde es verdad:
- ⟨$natc_3$⟩ = ⟨$natc_1$⟩ * 8
- ⟨$natc_2$⟩ = ⟨$natc_3$⟩ * ⟨$octd_1$⟩

[$⟨$natc_1$⟩]; ➡ ⟨$natc_1$⟩;

Ejemplo 1

$Z_{.op1}$ = 05308; ⇒ $Z_{.op1}$ = [$5]308; ⇒ $Z_{.op1}$ = [$43]08; ⇒

$Z_{.op1}$ = [$344]8; ⇒ $Z_{.op1}$ = [$2752]; ⇒ $Z_{.op1}$ = 2752;

Ejemplo 1'

$Z_{.op1}$ = -0003; ⇒ $Z_{.op1}$ = -[$0]003; ⇒ $Z_{.op1}$ = -[$0]03; ⇒

$Z_{.op1}$ = -[$0]3; ⇒ $Z_{.op1}$ = -[$3]; ⇒ $Z_{.op1}$ = -3;

Z_{Ω} = ⟨$signo_1$⟩0⟨x⟩⟨$exad_1$⟩ ➡ Z_{Ω} = ⟨$signo_1$⟩[$$0]⟨$exad_1$⟩

`[$$⟨natc1⟩]⟨exad1⟩` ➡ `[$$⟨natc2⟩]`

donde es verdad:
- $\langle natc_3 \rangle = \langle natc_1 \rangle * 16$
- $\langle natc_2 \rangle = \langle natc_3 \rangle * \langle exad_1 \rangle$

`[$$⟨natc1⟩];` ➡ `⟨natc1⟩;`

Ejemplo 1

`Z.op1 = 0x29;` ⇒ `Z.op1 = [$$0]29;` ⇒ `Z.op1 = [$2]9;` ⇒

`Z.op1 = [$41];` ⇒ `Z.op1 = 41;`

Ejemplo 1'

`Z.op1 = -0x017;` ⇒ `Z.op1 = -[$$0]017;` ⇒ `Z.op1 = -[$0]17;` ⇒

`Z.op1 = -[$1]7;` ⇒ `Z.op1 = -[$23];` ⇒ `Z.op1 = -23;`

⟨*ceros*⟩ → ε | **0**⟨*ceros*⟩

⟨*natn*⟩ → ⟨*ceros*⟩⟨*natural*⟩

⟨*!ndc*⟩ → *no* ⟨*ndc*⟩.

⟨*!s*⟩ → *no* ⟨*ndc*⟩, *no E*.

⟨*!ndc*⟩ – cualquier símbolo, incluído espacios, que no sea ⟨*ndc*⟩.
⟨*!s*⟩ – cualquier símbolo, incluído espacios, que no sea ⟨*ndc*⟩ ni *E*.

`⟨natn1⟩.` ➡ `[⟨natn1⟩,1]`

`.⟨ndc1⟩` ➡ `[0,1]⟨ndc1⟩`

`[⟨natn1⟩,⟨natural1⟩]⟨ndc1⟩` ➡ `[⟨natn1⟩⟨ndc1⟩,⟨natural1⟩0]`

$[\langle natn_1\rangle,\langle natural_1\rangle]E\langle signo_1\rangle\langle ceros\rangle\langle nd_1\rangle$ ➡ $[\langle natn_1\rangle,\langle natural_1\rangle]E\langle signo_1\rangle\langle nd_1\rangle$

$[\langle natn_1\rangle,\langle natural_1\rangle]\langle !s_1\rangle$ ➡ $[\langle natn_1\rangle,\langle natural_1\rangle]E0\langle !s_1\rangle$

$[\langle ceros\rangle\langle natc_1\rangle,\langle natural_1\rangle]E\langle signo_1\rangle\langle ceros\rangle 0\langle !ndc_1\rangle$

⬇

$[\langle natc_1\rangle\$\langle natural_1\rangle]\langle !ndc_1\rangle$

$[\langle natn_1\rangle,\langle natural_1\rangle]E+1$ ➡ $[\langle natn_1\rangle,\langle natural_1\rangle]E1$

$[\langle natn_1\rangle,\langle natural_1\rangle]E1\langle !ndc_1\rangle$ ➡ $[\langle natn_1\rangle 0,\langle natural_1\rangle]\langle !ndc_1\rangle$

$[\langle natn_1\rangle,\langle natural_1\rangle]E\text{-}1\langle !ndc_1\rangle$ ➡ $[\langle natn_1\rangle,\langle natural_1\rangle 0]\langle !ndc_1\rangle$

$[\langle natn_1\rangle,\langle natural_1\rangle]E\langle signo_1\rangle n_1$

⬇

$[\langle natn_1\rangle,\langle natural_1\rangle]E\langle signo_1\rangle 1E\langle signo_1\rangle m_1$

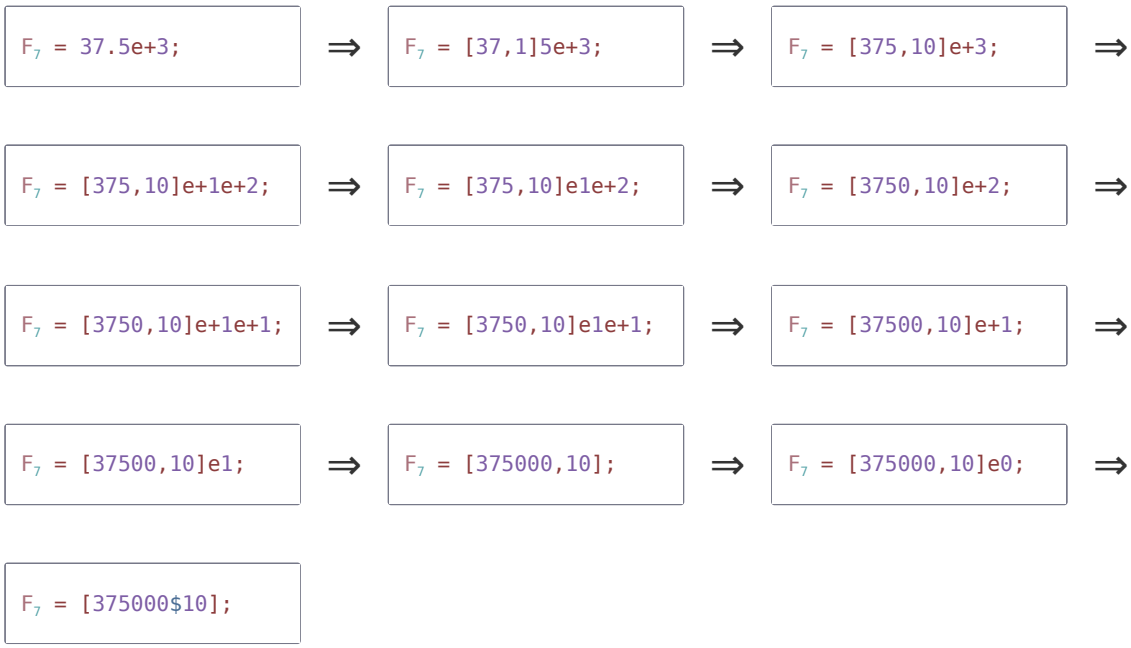
donde:

- n_1 no es 1.
- $n_1 = m_1 + 1$ es verdad.

Ejemplo 1

```
F23 = -02.520 + .8;          ⇒  F23 = -[02,1]520 + .8;          ⇒
F23 = -[025,10]20 + .8;      ⇒  F23 = -[0252,100]0 + .8;        ⇒
F23 = -[02520,1000] + .8;    ⇒  F23 = -[02520,1000]e0 + .8;     ⇒
F23 = -[2520$1000] + .8;     ⇒  F23 = -[2520$1000] + [0,1]8;    ⇒
F23 = -[2520$1000] + [08,10]e0;  ⇒  F23 = -[2520$1000] + [8$10];
```

Ejemplo 1'



Reserva de posiciones

⟨*simb*⟩ → *cualquier símbolo*

⟨*esp*⟩ → *espacio sin símbolos*

⟨*rcadena*⟩ → ε | ⟨*simb*⟩⟨*rcadena*⟩

cadena → ⟨*simb*⟩⟨*rcadena*⟩

⟨*simb_esp*⟩ → ⟨*simb*⟩ | ⟨*esp*⟩

⟨*rlínea*⟩ → ε | ⟨*simb_esp*⟩⟨*rlínea*⟩

⟨*línea*⟩ → ⟨*simb*⟩⟨*rlínea*⟩

textoc → ⟨*línea*⟩ | ⟨*línea*⟩*textoc*

texto → ε | *textoc*

EOF → *final de texto*

#definec *cadena*$_1$ *cadena*$_2$ ➡ ε

Efecto secundario

Hasta *EOF*, cualquier aparición de *cadena* se sustituye '*cadena*$_1$' por '*cadena*$_2$', siempre que '*cadena*$_1$' no forme parte de un '*enombre*' mayor.

Ejemplo

```
#definec foo bar
#definec baz 5

fooz = *foo;
foo$3 = foo + baz;
```

⇒

```
#definec baz 5

fooz = *bar;
foo$3 = bar + baz;
```

⇒

```
fooz = *bar;
foo$3 = bar + 5;
```

#definecc *cadena*$_1$ *cadena*$_2$ ➡ ε

Efecto secundario

Se sustituye igual que en #definec menos en *texto*$_1$ en las siguientes situaciones:

```
){                  :{
   texto_1     ó       texto_1
}                   }
```

Ejemplo 1

```
#definecc foo bar
:def_subp:{
   foo = 20;
   ...
}
foo = 9;
```

⇒

```
:def_subp:{
   foo = 20;
   ...
}
bar = 9;
```

Ejemplo 1'

```
#definecc bar qux
foo(){
   #definecc bar quux
   bar = 26;
   ...
}
baz(){
   bar = 9;
   ...
}
```

⇒

```
foo(){
   #definecc bar quux
   bar = 26;
   ...
}
baz(){
   bar = 9;
   ...
}
```

⇒

```
foo(){
   quux = 26;
   ...
}
baz(){
   bar = 9;
   ...
}
```

nombre → *identificador de K&R*

⟨*dims*⟩ → ε | **$***n*⟨*dims*⟩

enombre → *nombre*⟨*dims*⟩

⟨*stars*⟩ → ε | ***⟨*stars*⟩

⟨*inds*⟩ → ε | **[***n***]**⟨*inds*⟩

var → ⟨*stars*⟩*nombre*⟨*inds*⟩ | ⟨*stars*⟩*nombre***[]**

⟨*signo*⟩ → + | - | ε

⟨*racional*⟩ → **[***n***$***m***]**

litp → **0** | ⟨*natural*⟩ | ⟨*racional*⟩

lit → ⟨*sign*⟩*litp*

list_lit → ε | *lit* | *lit***,** *list_lit*

⟨*asig*⟩ → *lit* | **{***list_lit***}**

⟨*inic*⟩ → ε | **=** ⟨*asig*⟩

vari → *var* ⟨*inic*⟩

list_inic → *vari* | *vari***,** *list_inic*

⟨*signed*⟩ → **signed** | ε

type → **unsigned int** | ⟨*signed*⟩ **int** | **float**

identificador de K&R
letra o '_' seguido de cualquier cantidad de símbolos *n*, letras o '_'.

```
signed ➡ ε
```

```
int ➡ float
```

:num_position:

El expansor asocia un ⟨*natural*⟩ a **:num_position:**.

- Antes de empezar las expansiones le asocia 1.
- Si está asociado *m*, después de expandir una intrucción '*unsigned int* ⟨*stars*⟩*nombre*⟨*dims*⟩' asocia *n*, siendo verdad *n* = *m* + 1.

```
unsigned int ⟨stars1⟩ nombre1⟨dims1⟩;
                ⬇
#definec nombre1 Yn⟨dims1⟩
#definec nombre1⟨dims1⟩ Yn
```

donde *n* es el ⟨*natural*⟩ asociado a **:num_position:**.

Ejemplo

```
1 INIT
2 unsigned int *foo;
3 unsigned int bar;
4 foo = 47;
5 bar = *foo;
6
```

⇒

```
1 1
2 unsigned int *foo;
3 unsigned int bar;
4 foo = 47;
5 bar = *foo;
6
```

⇒

```
1 1
2 #definec foo Y1
3 #definec foo Y1
4 unsigned int bar;
5 foo = 47;
6 bar = *foo;
```

⇒ … ⇒

```
1 1
2 #definec bar Y2
3 #definec bar Y2
4 Y1 = 47;
5 bar = *Y1;
6
```

⇒ …

```
float ⟨stars1⟩nombre1⟨dims1⟩;
```

⬇

```
unsigned int ⟨stars1⟩nombre1⟨dims1⟩;
#definec ⟨stars1⟩Yn ⟨stars1⟩Fn
```

donde n es el ⟨*natural*⟩ asociado a **:num_position:**.

Ejemplo

```
1 INIT
2 float foo;
3 float **bar[2];
4 bar[1] = 91;
5 *bar[1] = 127;
6 foo = **bar[1];
```

⇒

```
1 1
2 float foo;
3 float **bar[2];
4 bar[1] = 91;
5 *bar[1] = 127;
6 foo = **bar[1];
```

⇒

```
1 1
2 unsigned int foo;
3 #definec Y1 F1
4 float **bar[2];
5 bar[1] = 91;
6 *bar[1] = 127;
7 foo = **bar[1];
```

⇒

```
1 1
2 #definec foo Y1
3 #definec foo Y1
4 #definec Y1 F1
5 float **bar[2];
6 bar[1] = 91;
7 *bar[1] = 127;
8 foo = **bar[1];
```

⇒

```
1 1
2 #definec foo Y1
3 #definec Y1 F1
4 float **bar[2];
5 bar[1] = 91;
6 *bar[1] = 127;
7 Y1 = **bar[1];
```

⇒

```
1 1
2 #definec Y1 F1
3 float **bar[2];
4 bar[1] = 91;
5 *bar[1] = 127;
6 Y1 = **bar[1];
```

⇒

```
1 1
2 float **bar[2];
3 bar[1] = 91;
4 *bar[1] = 127;
5 F1 = **bar[1];
```

⇒ ...

$type_1\ vari_1,\ list\_inic_1;$ ➡

```
type1 vari1;
type1 list_inic1;
```

$type_1\ \langle stars_1 \rangle enombre_1 = lit_1;$ ➡

```
type1 ⟨stars1⟩enombre1;
enombre1 = lit1;
```

Ejemplo

```
unsigned int foo = 7, bar[3] = {5, 23, 17};
```

⇒

```
unsigned int foo = 7;
unsigned int bar[3] = {5, 23, 17};
```

⇒

```
unsigned int foo;
foo = 7;
unsigned int bar[3] = {5, 23, 17};
```

⇒ ...

Vectores

$type_1\ \langle stars_1 \rangle$enombre$_1$[1]; ➡

```
type1 ⟨stars1⟩enombre1;
```

$type_1\ \langle stars_1 \rangle$enombre$_1[n_1];$

⬇

```
type1 ⟨stars1⟩enombre1[m];
type1 enombre1$n1;
```

donde $n = m + 1$ es verdad.

Ejemplo

```
int *foo[3];
```

⇒

```
int *foo[2];
int foo$3;
```

⇒

```
int *foo[1];
int foo$2;
int foo$3;
```

⇒

```
int *foo;
int foo$2;
int foo$3;
```

```
type1 ⟨stars1⟩enombre1[1] = {};
```

⬇

```
type1 ⟨stars1⟩enombre1 = 0;
```

```
type1 ⟨stars1⟩enombre1[n] = {}; ➡ type1 ⟨stars1⟩enombre1 = 0;
                                  type1 ⟨stars1⟩enombre1$$m = {};
```

donde:

- *n* no es 1.
- *n* = *m* + 1 es verdad.

```
type1 ⟨stars1⟩enombre1[n1] = {lit1};
```

⬇

```
type1 ⟨stars1⟩enombre1[n1] = {lit1,};
```

```
type1 ⟨stars1⟩enombre1[n] = {lit1, list_lit1};
```

⬇

```
type1 ⟨stars1⟩enombre1 = lit1;
type1 ⟨stars1⟩enombre1$$m = {list_lit1};
```

donde *n* = *m* + 1 es verdad.

```
type1 ⟨stars1⟩enombre1$$n1 = {lit1};
```

⬇

```
type1 ⟨stars1⟩enombre1$$n1 = {lit1,};
```

```
type1 ⟨stars1⟩enombre1$$1 = {}; ➡ type1 ⟨stars1⟩enombre1$1 = 0;
```

```
type1 ⟨stars1⟩enombre1$$1 = {lit1,}; ➡ type1 ⟨stars1⟩enombre1$1 = lit1;
```

```
type1 ⟨stars1⟩enombre1$$n1 = {};
```

⬇

```
type1 ⟨stars1⟩enombre1$n1 = 0;
type1 ⟨stars1⟩enombre1$$m = {};
```

donde:

- n_1 no es 1.
- n_1 = *m* + 1 es verdad.

$type_1\ \langle stars_1 \rangle enombre_1\$\$n_1 = \{lit_1,\ list\_lit_1\};$

⬇

$type_1\ \langle stars_1 \rangle enombre_1\$n_1 = lit_1;$

$type_1\ \langle stars_1 \rangle enombre_1\$\$m = \{list\_lit_1\};$

donde:

- n_1 no es 1.
- $n_1 = m + 1$ es verdad.

Ejemplo 1

```
int foo[3] = {5, 23, 17};
```

⇒

```
int foo = 5;
int foo$$2 = {23, 17};
```

⇒

```
int foo;
foo = 5;
int foo$$2 = {23, 17};
```

⇒ ... ⇒

```
...
int foo$2 = 23;
int foo$$1 = {17};
```

⇒

```
...
int foo$2;
foo$2 = 23;
int foo$$1 = {17};
```

⇒ ... ⇒

```
...
int foo$$1 = {17};
```

⇒

```
...
int foo$$1 = {17,};
```

⇒

```
...
int foo$1 = 17;
```

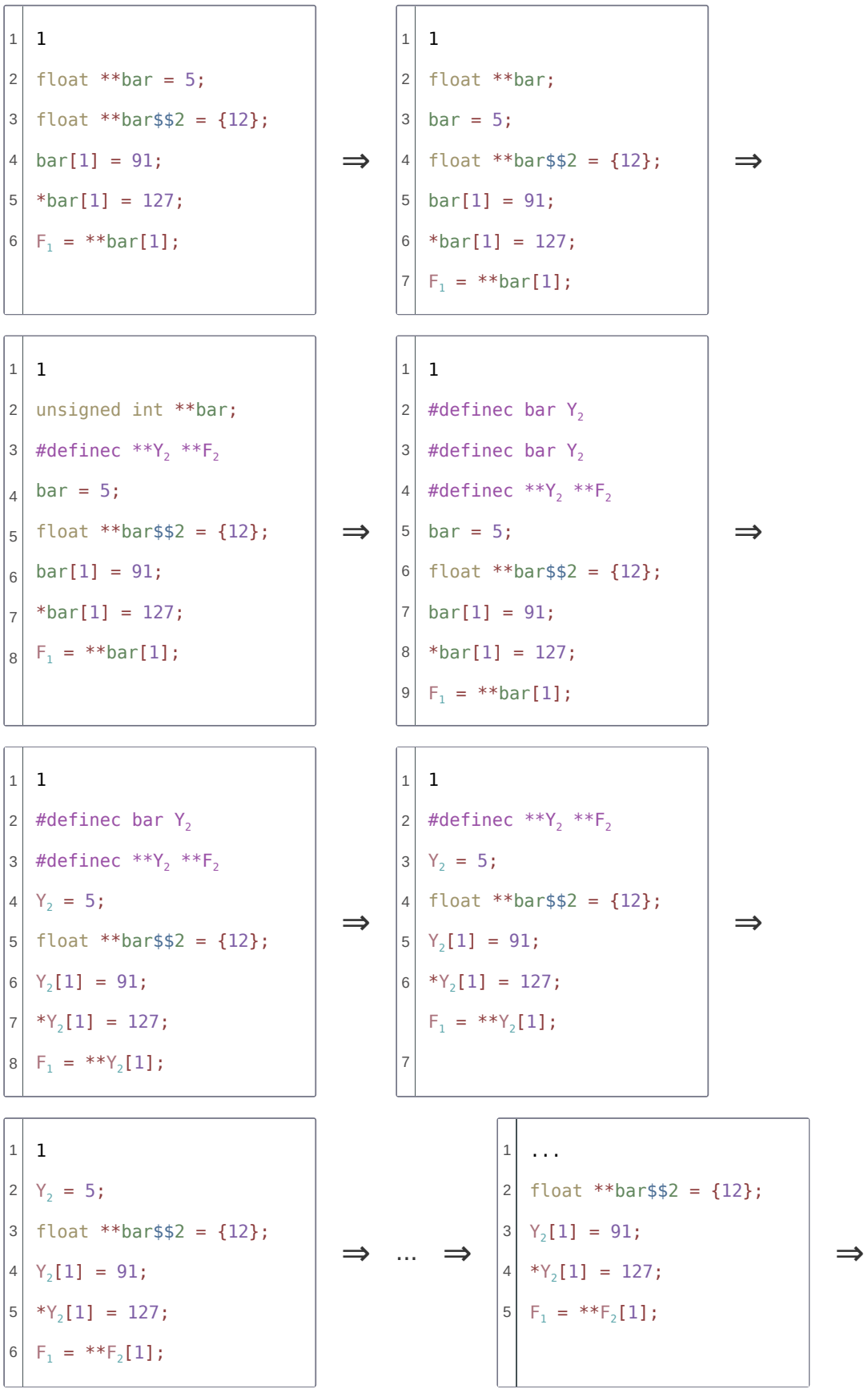
Ejemplo 1'

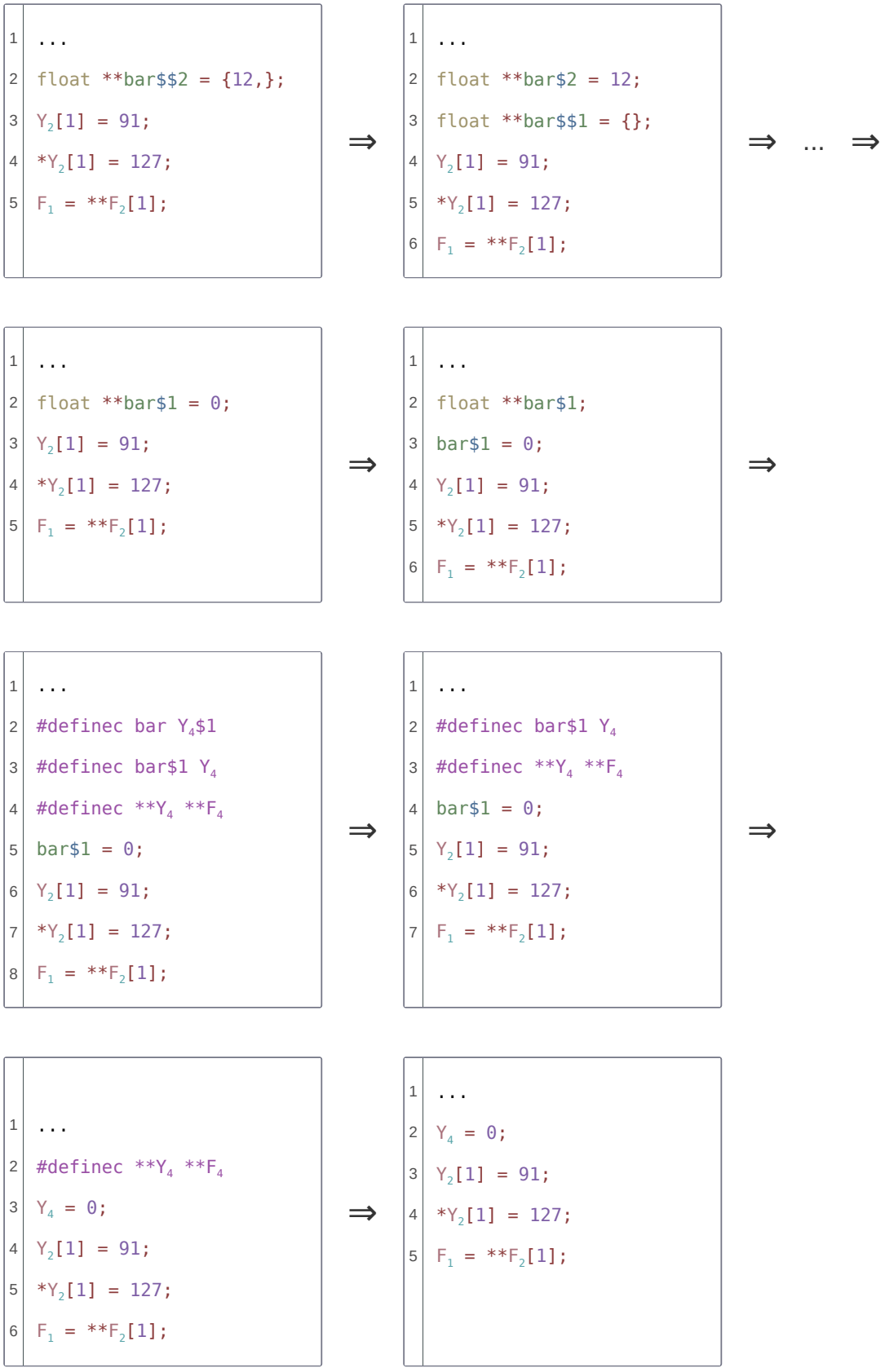
```
1 INIT
2 float foo;
3 float **bar[3] = {5, 12};
4 bar[1] = 91;
5 *bar[1] = 127;
6 foo = **bar[1];
```

⇒ ... ⇒

```
1 1
2 float **bar[3] = {5, 12};
3 bar[1] = 91;
4 *bar[1] = 127;
5 F1 = **bar[1];
```

⇒


1
float **bar = 5;
float **bar$$2 = {12};
bar[1] = 91;
*bar[1] = 127;
F1 = **bar[1];
⇒
1
float **bar;
bar = 5;
float **bar$$2 = {12};
bar[1] = 91;
*bar[1] = 127;
F1 = **bar[1];
⇒
1
unsigned int **bar;
#definec **Y2 **F2
bar = 5;
float **bar$$2 = {12};
bar[1] = 91;
*bar[1] = 127;
F1 = **bar[1];
⇒
1
#definec bar Y2
#definec bar Y2
#definec **Y2 **F2
bar = 5;
float **bar$$2 = {12};
bar[1] = 91;
*bar[1] = 127;
F1 = **bar[1];
⇒
1
#definec bar Y2
#definec **Y2 **F2
Y2 = 5;
float **bar$$2 = {12};
Y2[1] = 91;
*Y2[1] = 127;
F1 = **Y2[1];
⇒
1
#definec **Y2 **F2
Y2 = 5;
float **bar$$2 = {12};
Y2[1] = 91;
*Y2[1] = 127;
F1 = **Y2[1];
⇒
1
Y2 = 5;
float **bar$$2 = {12};
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒ ... ⇒
...
float **bar$$2 = {12};
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒


...
float **bar$$2 = {12,};
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒
...
float **bar$2 = 12;
float **bar$$1 = {};
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒ ... ⇒
...
float **bar$1 = 0;
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒
...
float **bar$1;
bar$1 = 0;
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒
...
#definec bar Y4$1
#definec bar$1 Y4
#definec **Y4 **F4
bar$1 = 0;
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒
...
#definec bar$1 Y4
#definec **Y4 **F4
bar$1 = 0;
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒
...
#definec **Y4 **F4
Y4 = 0;
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];
⇒
...
Y4 = 0;
Y2[1] = 91;
*Y2[1] = 127;
F1 = **F2[1];

$type_1\ \langle stars_1 \rangle enombre_1[n][m_1]$

⬇

$type_1\ \langle stars_1 \rangle enombre_1\$m_1[p]$

donde $p = m * n$ es verdad.

Ejemplo

```
int foo[4][2][2] = {81, 11,  8,  3,
                    19, 53,  1, 32,
                    12, 82, 13, 13,
                     9,  2,  1, 54};
```

⇒

```
int foo$2[8][2] = {81, 11,  8,  3,
                   19, 53,  1, 32,
                   12, 82, 13, 13,
                    9,  2,  1, 54};
```

⇒

```
int foo$2$2[16] = {81, 11,  8,  3,
                   19, 53,  1, 32,
                   12, 82, 13, 13,
                    9,  2,  1, 54};
```

⇒

```
int foo$2$2[16] = {81, 11,  8,  3,
                   19, 53,  1, 32,
                   12, 82, 13, 13,
                    9,  2,  1, 54};
```

⇒

```
int foo$2$2 = 81;
int foo$2$2$$15 = {11,  8,  3, 19,
                   53,  1, 32, 12,
                   82, 13, 13,  9,
                    2,  1, 54};
```

$type_1\ \langle stars_1 \rangle nombre_1[] = \{lit_1\};$ ➡ $type_1\ \langle stars_1 \rangle enombre_1[] = \{lit_1,\};$

$type_1\ \langle stars_1 \rangle nombre_1[] = \{lit_1,\ list\_lit_1\};$

⬇

$type_1\ \langle stars_1 \rangle nombre_1 = lit_1;$
$type_1\ \langle stars_1 \rangle nombre_1\$1 = \{list\_lit_1\};$

$type_1\ \langle stars_1 \rangle nombre_1\$n = \{\};$ ➡ ε

$type_1\ \langle stars_1 \rangle nombre_1\$n = \{lit_1\};$ ➡ $type_1\ \langle stars_1 \rangle nombre_1\$n = \{lit_1,\};$

$type_1\ \langle stars_1 \rangle nombre_1\$n = \{lit_1,\ list\_lit_1\};$

⬇

$type_1\ \langle stars_1 \rangle nombre_1\$n = lit_1;$
$type_1\ \langle stars_1 \rangle nombre_1\$m = \{list\_lit_1\};$

donde $m = n + 1$ es verdad.

Ejemplo

```
...
int foo[] = {55, 24, 8,}
foo[2] = 81;
```

⇒

```
...
int foo = 55;
int foo$1 = {24, 8,}
foo[2] = 81;
```

⇒ ... ⇒

```
...
int foo$1 = 24;
int foo$2 = {8,}
F9[2] = 81;
```

⇒ ... ⇒

```
...
int foo$2 = 8;
int foo$3 = {};
F9[2] = 81;
```

⇒ ... ⇒

```
...
F11 = 8;
int foo$3 = {};
F9[2] = 81;
```

⇒

```
...
1^423
int foo$3 = {};
F9[2] = 81;
```

⇒

```
...
1^423
F9[2] = 81;
```

$type_1\ \langle stars_1 \rangle enombre_1[][m_1]$ ➡ $type_1\ \langle stars_1 \rangle enombre_1\$m_1[]$

Ejemplo

```
int foo[][2][2] = {81, 11,  8,  3,
                   19, 53,  1, 32,
                   12, 82, 13, 13}
```

⇒

```
int foo$2[][2] = {81, 11,  8,  3,
                  19, 53,  1, 32,
                  12, 82, 13, 13}
```

⇒

```
int foo$2$2[] = {81, 11,  8,  3,
                 19, 53,  1, 32,
                 12, 82, 13, 13}
```

Asignaciones

$\langle opd \rangle \rightarrow$ ++ | --

$\langle yd \rangle \rightarrow \langle opd \rangle \langle stars \rangle \mathbf{Y}_n \mid \langle stars \rangle \mathbf{Y}_n \langle opd \rangle$

$\langle vindx \rangle \rightarrow \langle natural \rangle \mid \mathbf{Z}_\Omega \mid \langle stars \rangle \mathbf{Y}_n \mid \langle yd \rangle \mid \mathbf{0}$

$\langle indx \rangle \rightarrow \mathbf{[} \langle vindx \rangle \mathbf{]}$

$\langle indxs \rangle \rightarrow \varepsilon \mid \langle indx \rangle \langle indxs \rangle$

$V \rightarrow \mathbf{Y} \mid \mathbf{F}$

$V_n' \rightarrow \langle stars \rangle V_n \langle dims \rangle \langle indxs \rangle$

$\langle preinstrucción \rangle \rightarrow x^\Omega$

$\langle preinstrucciones \rangle \rightarrow \varepsilon \mid x^\Omega \langle preinstrucciones \rangle$

Expansiones finales 1'

Cuando se usan macros de la forma '*unsigned int* ⟨*stars*⟩*nombre*⟨*dims*⟩' el expansor, en las expansiones finales:

- después de agregar las instrucciones '1', añade al final una nueva instrucción $1^{:top}$ con una marca $\leftarrow$**:*i*** apuntando a ella.
- sustituye todas las apariciones de **:top** por ***:i*** en todas las $x^{:top}$.

$V_{.\langle id \rangle}$ = Z_Ω; ➡ $Z_{.\langle id \rangle}$ = Z_Ω;

Z_Ω = $V_{.\langle id \rangle}$; ➡ Z_Ω = $Z_{.\langle id \rangle}$;

Ejemplo

```
Z.opr = Y.opl;
```
⇒
```
Z.opr = Z.opl;
```

V_Ω = V_Ω; ➡ ε

Ejemplo

```
F.opf = F.opf;
F--;
```
⇒
```
F--;
```

Z_Ω = &V_n; ➡
```
Z_Ω = Z:top;
Z_Ω += n;
```

Ejemplo

```
Z.opl = &Y8;
```
⇒
```
Z.opl = Z:top;
Z.opl += 8;
```

Z_Ω = &V_n[$\langle vindx_1 \rangle$]; ➡
```
Z.indr = ⟨vindx1⟩;
Z_Ω = &Vn;
Z_Ω += Z.indr;
```

Ejemplo

```
Z.op1 = &Y8[13];
```

⇒

```
Z.indr = 13;
Z.op1 = &Y8;
Z.op1 += Z.indr;
```

$Z_{\Omega} = \&V_n\$k_1\langle dims_1\rangle[\lambda_1][\lambda_2]$ ➡

```
Z.ind1 = k1;
Z.ind2 = λ1;
Z.indr = Z.ind1 * Z.ind2;
Z.ind1 = λ2;
Z.indr += Z.ind1;
ZΩ = &Vn⟨dims1⟩[Z.indr]
```

Ejemplo

```
Z.op1 = &foo$12$15[7][Y23][4];
```

⇒

```
Z.ind1 = 12;
Z.ind2 = 7;
Z.indr = Z.ind1 * Z.ind2;
Z.ind1 = Y23;
Z.indr += Z.ind1;
Z.op1 = &foo$15[Z.indr][4];
```

⇒

... ⇒

```
...
Z.ind1 = 15;
Z.ind2 = Z.indr;
Z.indr = Z.ind1 * Z.ind2;
Z.ind1 = 4;
Z.indr += Z.ind1;
Z.op1 = &foo[Z.indr];
```

'⟨*simb*$_1$⟩' ➡ *c*

donde *c* es el ⟨*natural*⟩ que representa el código del símbolo Unicode ⟨*simb*$_1$⟩.

Ejemplo

```
foo = 'G';
```
⇒
```
foo = 71;
```

:ajustar:

Sea

```
:ajustar:{
  texto1
}
```

y Ξ cualquier Ω distinto de **:change**.

- Si *texto*$_1$ no es ⟨*preinstrucciones*⟩:

  ```
  :ajustar:{ ➡ :ajustar:{
    texto1       texto2
  }            }
  ```

  donde *texto*$_2$ es ⟨*preinstrucciones*⟩, resultado de expandir macroinstrucciones en *texto*$_1$.

- Si *texto*$_1$ es ⟨*preinstrucciones*⟩:

  ```
  :ajustar:{ ➡ ε
  }
  ```

  ```
  :ajustar:{          ➡ :ajustar:{
    ⟨preinstrucciones1⟩    ⟨preinstrucciones1⟩
    xΞ                     }
  }                     xΞ
  ```

```
:ajustar:{                    ➡   Z.i = Z.posf
  ⟨preinstrucciones1⟩               :ajustar:{
  x:change                            ⟨preinstrucciones1⟩
}                                   }
                                    x                    ←:i
```

```
Zω = Vn⟨dims1⟩⟨indxs1⟩;  ➡   Z.adjp = &Vn⟨dims1⟩⟨indxs1⟩;
                              :ajustar:{
                                  Zω = Z.change;
                              }
```

Ejemplo

```
Z.op2 = Y7;
```

⇒

```
Z.adjp = &Y7;
:ajustar:{
    Z.op2 = Z.change;
}
```

```
Vn⟨dims1⟩⟨indxs1⟩ = Zω;  ➡   Z.adjp = &Vn⟨dims1⟩⟨indxs1⟩;
                              :ajustar:{
                                  Z.change = Zω;
                              }
```

Ejemplo

```
F7[13] = Z.res1;
```

⇒

```
Z.adjp = &F7[13];
:ajustar:{
    Z.change = Z.res1;
}
```

```
Z_Ω = *V_Φ'; ➡  Z.adjp = V_Φ';
                :ajustar:{
                    Z_Ω = Z.change;
                }
```

Ejemplo

```
Z.op2 = **Y7;
```

⇒

```
Z.adjp = *Y7;
:ajustar:{
    Z.op2 = Z.change;
}
```

⇒

```
Z.adjp = Y7;
:ajustar:{
    Z.adjp = Z.change;
}
:ajustar:{
    Z.op2 = Z.change;
}
```

```
*V_Ω = Z_Φ; ➡  Z.adjp = V_Ω' ⟨dims1⟩⟨indxs1⟩;
               :ajustar:{
                   Z.change = Z_Φ;
               }
```

Ejemplo

```
*F7[13] = Z.res1;
```

⇒

```
Z.adjp = F7[13];
:ajustar:{
    Z.change = Z.res1;
}
```

⚠ *Nota sobre C*: la equivalencia en el uso de variables puntero y variables array no se recoge en las macros.

Comentarios

```
/* texto */ ➡ ε
```

Declaración de subprogramas

⟨*vindf*⟩ → ε | ⟨*nat*⟩

⟨*indf*⟩ → ε | **[**⟨*vindf*⟩**]**⟨*indf*⟩

⟨*parid*⟩ → ε | *nombre*⟨*indf*⟩

⟨*param*⟩ → ε | *type* ⟨*stars*⟩⟨*parid*⟩

⟨*rest_params*⟩ → ε | **,**⟨*param*⟩⟨*rest_params*⟩

⟨*params*⟩ → ε | ⟨*param*⟩⟨*rest_params*⟩

⟨*tpar*⟩ → **$Y** | **$F**

⟨*tpars*⟩ → ε | ⟨*tpar*⟩⟨*tpars*⟩

⟨*ftype*⟩ → *type* | **void**

⟨*fnombre*⟩ → *nombre*⟨*tpars*⟩

nombre$_1$(⟨*param*$_1$⟩⟨*rest_params*$_1$⟩); ➡ *nombre*$_1$$⟨*param*$_1$⟩(⟨*rest_params*$_1$⟩);

fnombre$_1$$*V*(,⟨*param*$_1$⟩⟨*rest_params*$_1$⟩);

⬇

fnombre$_1$$*V*,$⟨*param*$_1$⟩(⟨*rest_params*$_1$⟩);

Ejemplo

```
float foo(unsigned int **, float);
```

⇒

```
float foo$unsigned int **(, float);
```

fnombre$_1$$⟨*type*$_1$⟩⟨*stars*$_1$⟩*nombre*[⟨*vindf*⟩]⟨*indf*⟩(

⬇

fnombre$_1$$⟨*type*$_1$⟩⟨*stars*$_1$⟩*(

Ejemplo

```
unsigned int foo(float *bar[3], float);
```
⇒

```
unsigned int foo$float *bar[3](, float);
```
⇒
```
unsigned int foo$float **(, float);
```

fnombre$\langle$*type*$_1$$\rangle$$\langle$*stars*$_1$$\rangle$*nombre*(➡ *fnombre*$_1$$\langle$*type*$_1$$\rangle$$\langle$*stars*$_1$$\rangle$(

Ejemplo

```
float foo(float bar, float baz);
```
⇒
```
float foo$float bar(, float baz);
```
⇒

```
float foo$float(, float baz);
```

fnombre$_1$$\langle$*type*$_1$$\rangle$$\langle$*stars*$_1$$\rangle$*(➡ *fnombre*$_1$$Y(

Ejemplo

```
float foo$float **(, float);
```
⇒
```
float foo$Y(, float);
```

fnombre$_1$unsigned int(➡ *fnombre*$_1$$Y(

fnombre$_1$float(➡ *fnombre*$_1$$F(

Ejemplo

```
float foo(float bar, unsigned int baz, float);
```
⇒ ... ⇒

```
float foo$F$Y$(, float);
```
⇒

```
float foo$F$Y$float();
```
⇒

```
float foo$F$Y$F();
```

`void ⟨stars⟩**nombre1⟨tpars1⟩();` ➡ `void *nombre1⟨tpars1⟩();`

`void *nombre1⟨tpars1⟩();` ➡ `#define nombre1 Y:nombre1⟨tpars1⟩`

Ejemplo

```
void ***foo$F$F();
```
⇒
```
void *foo$F$F();
```
⇒
```
#define foo Y:foo$F$F
```

`void nombre1⟨tpars1⟩();` ➡ `#define nombre1 N:nombre1⟨tpars1⟩`

`unsigned int ⟨stars1⟩nombre1⟨tpars1⟩();` ➡ `#define nombre1 Y:nombre1⟨tpars1⟩`

`float ⟨stars1⟩nombre1⟨tpars1⟩();` ➡
```
#define nombre1 Y:nombre1⟨tpars1⟩
#define ⟨stars1⟩Y:nombre1 ⟨stars1⟩F:nombre1
```

Ejemplo

```
float *foo$Y();
...
bar = *foo();
baz = foo();
```
⇒
```
#define foo Y:foo$Y
#define *Y:foo *F:foo
...
bar = *foo();
baz = foo();
```
⇒
```
#define *Y:foo *F:foo
...
bar = *Y:foo$Y();
baz = Y:foo$Y();
```
⇒

```
...
bar = *F:foo$Y();
baz = Y:foo$Y();
```

ftype ⟨*stars*$_1$⟩ *nombre*$_1$(⟨*params*$_1$⟩){ ➡

```
type ⟨stars1⟩nombre1(⟨params1⟩);
nombre1(⟨params1⟩){
```

Ejemplo

```
float power(float base, float n){
   float i, p;
   p = 1;
   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
```

⇒

```
float power(float base, float n);
power(float base, float n){
   float i, p;
   p = 1;
   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
```

⇒

... ⇒

```
F:power$F$F(float base, float n){
   float i, p;
   p = 1;

   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
```

Definición de subprogramas

$\langle par\_def \rangle \rightarrow type\ \langle stars \rangle nombre \langle indf \rangle$

$\langle pars\_ant \rangle \rightarrow \varepsilon \mid \langle pars\_ant \rangle \langle par\_def \rangle,$

$\langle pars\_cp \rangle \rightarrow \varepsilon \mid \langle pars\_ant \rangle \langle par\_def \rangle$

$fvnombre \rightarrow V_{:nombre} \langle tpars \rangle \mid \mathbf{N}_{:nombre} \langle tpars \rangle$

$fvnombre_1(\langle pars\_ant_1 \rangle\ type_1 \langle stars_1 \rangle nombre_1[\langle vindf \rangle]\langle indf \rangle){$

⬇

$fvnombre_1(\langle pars\_ant_1 \rangle\ type_1 \langle stars_1 \rangle * nombre_1){$

Ejemplo

```
Y:foo$F$F(float bar, float baz[]){   ⇒   Y:foo$F$F(float bar, float *baz){
```

$fvnombre_1(\langle pars\_ant_1 \rangle \langle par\_def_1 \rangle, \langle par\_def_2 \rangle){$

⬇

$fvnombre_1\$\langle par\_def_2 \rangle(\langle pars\_ant_1 \rangle \langle par\_def_1 \rangle){$

Ejemplo

```
Y:foo$F$F(float bar, float *baz){   ⇒   Y:foo$F$F$float *baz(float bar){
```

$fvnombre_1(\langle pars\_ant_1 \rangle){$ ➡ $fvnombre_1\$\langle pars\_ant_1 \rangle(){$

Ejemplo

```
Y:foo$F(float bar){   ⇒   Y:foo$F$float bar(){
```

fvnombre$_1$Y$*type*$_1$ ⟨*stars*$_1$⟩**nombre*$_1$(⟨*pars_cp*$_1$⟩){ ➡

```
fvnombre1(⟨pars_cp1⟩){
    type1 ⟨stars1⟩*nombre1;
```

Ejemplo

```
Y:foo$Y$Y$float *bar(){
```

⇒

```
Y:foo$Y(){
    float *bar;
```

fvnombre$_1$Y$unsigned int *nombre*$_1$(⟨*pars_cp*$_1$⟩){ ➡

```
fvnombre1(⟨pars_cp1⟩){
    unsigned int nombre1;
```

fvnombre$_1$Y$float *nombre*$_1$(⟨*pars_cp*$_1$⟩){ ➡

```
fvnombre1(⟨pars_cp1⟩){
    float nombre1;
```

Ejemplo

```
float power(float base, float n){
    float i, p;
    p = 1;
    for(i = 1; i <= n; ++i){
        p = p * base;
    }
    return p;
}
```

⇒ ... ⇒

```
F:power$F$F$float n(float base){
    float i, p;
    p = 1;
    for(i = 1; i <= n; ++i){
        p = p * base;
    }
    return p;
}
```

⇒

```
F:power$F(float base){
    float n;
    float i, p;
    p = 1;
    for(i = 1; i <= n; ++i){
        p = p * base;
    }
    return p;
}
```

⇒

```
F:power$F$float base(){
   float n;
   float i, p;
   p = 1;
   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
```

⇒

```
F:power(){
   float base;
   float n;
   float i, p;
   p = 1;
   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
```

⚠ *Nota sobre C*: la posibilidad de usar variables globales no está recogida en las macros.

type ⟨*stars*⟩**V*:nombre(){ ➡ Y:nombre(){

unsigned int *V*:nombre(){ ➡ Y:nombre(){

float *V*:nombre(){ ➡ F:nombre(){

V:nombre(){ ➡
   *texto*1
}
*texto*2
EOF

*texto*2
$*V*:nombre(){
   *texto*1
}
EOF

Ejemplo

```
F:power(){
   float base;
   float n;
   float i, p;
   p = 1;
   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
...
```

⇒

```
...
$F:power(){
   float base;
   float n;
   float i, p;
   p = 1;
   for(i = 1; i <= n; ++i){
      p = p * base;
   }
   return p;
}
```

$\langle retvp \rangle \rightarrow lit \mid V'_n$

$\langle retv \rangle \rightarrow \langle sign \rangle \langle retvp \rangle$

`return;` ➡ `*:ret`

`return` $\langle retv_1 \rangle$`;` ➡

```
&R:ret = ⟨retv1⟩;
return;
```

Ejemplo

```
return 13;
```

⇒

```
$R.ret = 13;
return;
```

⇒ ... ⇒

```
...
*:ret
```

←:⟨*id*⟩

Señales que pone el expansor.

$W \rightarrow V \mid \mathbf{N}$

```
$W:nombre(){ ➡
    texto1
}
```

```
:def_subp:{
    #definec $R W
    texto1
    Z:i = *Y:top;          ←:ret
    :rest_vals:
    ∗                      ←:i
    1:np                   ←:npos
}
```

Ejemplo

```
$F:power(){
    float base;
    float n;
    float i, p;
    p = 1;
    for(i = 1; i <= n; ++i){
        p = p * base;
    }
    return p;
}
```

⇒

```
:def_subp:{
    #definec $R F
    float base;
    ...
        p = p * base;
    }
    return p;
    Z:67 = *Y:top;
    :rest_vals:            ←:ret
    ∗
    1:np                   ←:67
}                          ←:npos
```

:rest_vals:

```
:rest_vals: ➡  Z:top--;
               Z.r1 = *Y:top;
               Z:top--;
               Z.r2 = *Y:top;
               Z:top--;
               Z.and = *Y:top;
               Z:top--;
               Z.or = *Y:top;
               Z:top--;
               Z.obl = *Y:top;
               Z:top--;
               Z.opl = *Y:top;
               Z:top--;
               Z.brk = *Y:top;
               Z:top--;
               Z.sw = *Y:top;
               Z:top--;
               Z:top = *Y:top;
```

:def_subp:{ *texto*$_1$ }

Sea *m* el ⟨*natural*⟩ asociado a **:num_position:**. El expansor:

- asocia 1 a **:num_position:**.
- expande las macroinstrucciones en *texto*$_1$ hasta obtener ⟨*preinstrucciones*$_1$⟩.
- sustituye ←**:return** por una señal, sea ←***:k***, y todas las apariciones de **:return** en ⟨*preinstrucciones*$_1$⟩ por ***:k***.
- sustituye la última ⟨*preinstrucción*⟩, $1^{:np}$, por 1^n, donde *n* es el ⟨*natural*⟩ asociado a **:num_position:**.
- sustituye ←**:npos** por una señal, sea ←***:p***, y todas las apariciones de **:npos** en ⟨*preinstrucciones*$_1$⟩ por ***:p***.
- asocia *m* a **:num_position:**.

Expansiones finales 1”

Si se usan macroinstrucciones **:def_subp:**, el expansor, en las expansiones finales:

- antes de añadir la ⟨*preinstrucción*⟩ $1^{\text{:top}}$ añade una ⟨*preinstrucción*⟩ 1^n, donde *n* es el ⟨*natural*⟩ asociado a **:num_position:**.
- añade una señal **←:$npos**. apuntando a esta ⟨*preinstrucción*⟩.
- sustituye las apariciones de **:npos** en cualquier ⟨*preinstrucción*⟩ por **:$npos**.
- por cada señal **←:⟨*id*$_1$⟩** presente:
  - la sustituye por una señal **←:⟨*natural*⟩**; sea ésta **←:*i***.
  - sustituye las apariciones de **:⟨*id*$_1$⟩** en cualquier ⟨*preinstrucción*⟩ por **:*i***.

Llamada a subprogramas

⟨*argp_y*⟩ → ⟨*natural*⟩ | **0** | $\mathbf{Y}'_n$ | **&**V_n⟨*indf*⟩

⟨*arg_y*⟩ → ⟨*argp_y*⟩ | +⟨*argp_y*⟩

⟨*argp_f*⟩ → ⟨*racional*⟩ | $\mathbf{F}'_n$

⟨*arg_f*⟩ → ⟨*signo*⟩⟨*argp_f*⟩

⟨*arg*⟩ → ⟨*arg_y*⟩ | ⟨*arg_f*⟩

⟨*rest_args*⟩ → ε | **,**⟨*arg*⟩⟨*rest_args*⟩

⟨*args*⟩ → ε | ⟨*arg*⟩⟨*rest_args*⟩

$W_{:nombre}\langle tpars_1\rangle(\langle args_1\rangle);$ ➡

```
Z.fp = Z:top;
Z.fp += Z:npos;
*Y.fp = Z:top;
:guardar_vals:
Z.fp++;
*Y.fp = Z:i;
:poner_args:⟨tpars1⟩,⟨args1⟩$
Z:top = Z.fp;
✳:nombre
1:j                    ←:i
...                    ←:j
```

Ejemplo

```
F:foo$Y$Y(13, 53);
```

⇒

```
Z.fp = Z:top;
Z.fp += Z:npos;
*Y.fp = Z:top;
:guardar_vals:
Z.fp++;
*Y.fp = Z:91;
:poner_args:$Y$Y,13, 53$
Z:top = Z.fp;
✳:foo
1:92                   ←:91
...                    ←:92
```

:guardar_vals:

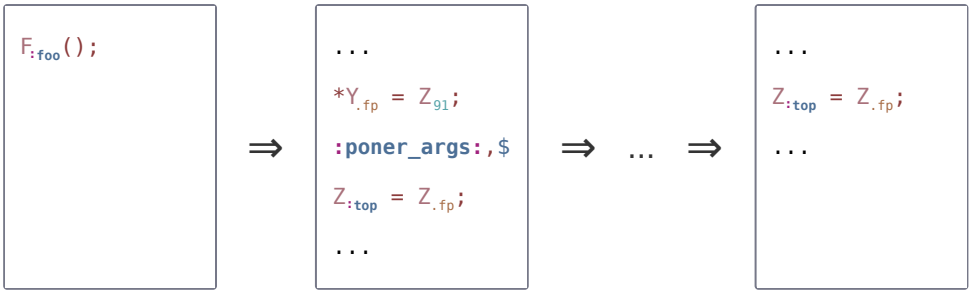
```
:guardar_vals: ➡ Z.fp++;
                 *Y:fp = Z.sw;
                 Z.fp++;
                 *Y:fp = Z.brk;
                 Z.fp++;
                 *Y:fp = Z.op1;
                 Z.fp++;
                 *Y:fp = Z.ob1;
                 Z.fp++;
                 *Y:fp = Z.or;
                 Z.fp++;
                 *Y:fp = Z.and;
                 Z.fp++;
                 *Y:fp = Z.r2;
                 Z.fp++;
                 *Y:fp = Z.r1;
```

:poner_args:

```
:poner_args:,$ ➡ ε
```

```
:poner_args:$ ➡ ε
```

Ejemplo



```
:poner_args:$V⟨tpars1⟩,⟨arg1⟩⟨rest_args1⟩$ ➡
    Z.fp++;
    :poner_args:$V,⟨arg1⟩
    :poner_args:$⟨tpars1⟩⟨rest_args1⟩$
    Z.fp--;
```

Ejemplo

```
:poner_args:$Y$Y,13,53$
```

⇒

```
Z.fp++;
:apilar:$Y,13
:poner_args:$Y,53$
Z.fp--;
```

⇒ ... ⇒

```
...
Z.fp++;
:apilar:$Y,53
:poner_args:$
Z.fp--;
Z.fp--;
```

⇒ ...

:apilar:

```
:apilar:$Y,⟨arg_y1⟩; ➡ *Y.fp = ⟨arg_y1⟩;
```

```
:apilar:$F,⟨arg_f1⟩; ➡ *F.fp = ⟨arg_f1⟩;
```

Ejemplo

```
:apilar:$Y,53
```

⇒

```
*Y.fp = 53;
```

```
:apilar:$Y,⟨arg_f1⟩; ➡
```

```
Z.opf = ⟨arg_f1⟩;
Z:top = Z.fp;
Z.opf = (unsigned int)F.opf;
Z.fp = Z:top;
*Y.fp = Z.opf;
```

```
:apilar:$F,⟨arg_y1⟩; ➡
```

```
Z.opf = ⟨arg_y1⟩;
Z:top = Z.fp;
Z.opf = (float)F.opf;
Z.fp = Z:top;
*F.fp = Z.opf;
```

Ejemplo

```
:apilar:$F,Y31
```

⇒

```
Z.opf = Y31;
Z:top = Z.fp;
Z.opf = (float)Z.opf;
Z.fp = Z:top;
*F.fp = Z.opf;
```

Operaciones aritméticas

$V'' \rightarrow V_{.\langle id \rangle}\langle tpars \rangle(\langle args \rangle)$

$\langle valp\_y \rangle \rightarrow \langle natural \rangle \mid \mathbf{0} \mid Y'_{\Omega} \mid \langle stars \rangle Y''$

$\langle val\_y \rangle \rightarrow \langle valp\_y \rangle \mid +\langle valp\_y \rangle$

$\langle valp\_f \rangle \rightarrow \langle racional \rangle \mid F'_{\Omega} \mid \langle stars \rangle F''$

$\langle val\_f \rangle \rightarrow \langle signo \rangle \langle valp\_f \rangle \mid -\langle valp\_y \rangle$

$\langle valp \rangle \rightarrow \langle valp\_f \rangle \mid \langle valp\_y \rangle$

$\alpha \mid \beta \rightarrow \langle val\_y \rangle \mid \langle val\_f \rangle$

```
Z_Ω = ⟨stars_1⟩V"; ➡ V";
                     Z_Ω = ⟨stars_1⟩V_.ret;
```

Ejemplo

```
Z_.opf = F_:foo$Y(41);
```

⇒

```
F_:foo$Y(41);
Z_.opf = F_.ret;
```

```
Z_Ω=(unsigned int)⟨val_y_1⟩; ➡ Z_Ω=⟨val_y_1⟩;
```

```
Z_Ω=(float)⟨val_f_1⟩; ➡ Z_Ω=⟨val_f_1⟩;
```

Ejemplo

```
Z_.op1 = (unsigned int)8;
```

⇒

```
Z_.op1 = 8;
```

Ejemplo 1'

```
Z_.op1 = (float)F_23;
```

⇒

```
Z_.op1 = F_23;
```

```
Z_Ω=(unsigned int)⟨val_f_1⟩; ➡ Z_Ω = ⟨val_f_1⟩;
                               Z_Ω = Y_:ftoy$F(F_Ω);
```

Ejemplo

```
Z_.opf = (unsigned int)F_2;
```

⇒

```
Z_.opf = F_2;
Z_.opf = Y_:ftoy$F(F_.opf);
```

Z_{Ω}=(float)⟨val_y_1⟩; ➡

```
Z_Ω = ⟨val_y1⟩;
Z_Ω = F_:ytof$Y(Y_Ω);
```

Ejemplo

```
Z_.opf = (float)17;
```

⇒

```
Z_.opf = 17;
Z_.opf = F_:ytof$Y(Y_.opf);
```

Y'_{Ω} = α; ➡

```
Z_.opf = (unsigned int)α;
Y'_Ω = Z_.opf;
```

F'_{Ω} = α; ➡

```
Z_.opf = (float)α;
F'_Ω = Z_.opf;
```

Ejemplo

```
*Y_:top = 29;
```

⇒

```
Z_.opf = (unsigned int)29;
*Y_:top = Z_.opf;
```

Z_{Ω} = -⟨$valp_1$⟩; ➡

```
Z_Ω = (float)⟨valp1⟩;
Z_Ω = F_:neg$F(F_Ω);
```

Ejemplo

```
Z_.opf = -31;
```

⇒

```
Z_.opf = (float)31;
Z_.opf = F_:neg$F(F_.opf);
```

Z_{Ω} = [⟨*natc*$_1$⟩$⟨*natc*$_2$⟩]; ➡ Z_{Ω} = $F_{:litf}$$Y$Y(⟨*natc*$_1$⟩,⟨*natc*$_2$⟩);

Ejemplo

```
Z.opf = -[15$1000];
```

⇒

```
Z.opf = (float)[15$1000];
Z.opf = F:neg$F(F.opf);
```

⇒

```
Z.opf = [15$1000];
Z.opf = F:neg$F(F.opf);
```

⇒

```
Z.opf = F:litf$Y$Y(15,1000);
Z.opf = F:neg$F(F.opf);
```

⟨*ops*⟩ → + | -

⟨*opm*⟩ → * | / | %

⟨*opd*⟩ → ++ | --

⟨*op*⟩ → ⟨*ops*⟩ | ⟨*opm*⟩

res_y(op) → ⟨*val_y*⟩⟨*op*⟩⟨*val_y*⟩

res_f(op) → ⟨*val_f*⟩⟨*op*⟩⟨*val_f*⟩ | ⟨*val_f*⟩⟨*op*⟩⟨*val_y*⟩ | ⟨*val_y*⟩⟨*op*⟩⟨*val_f*⟩

⟨*fna*⟩ → ε | α ⟨*ops*⟩

⟨*fnp*⟩ → ⟨*ops*⟩ | ;

Y'_n⟨*opd*$_1$⟩; ➡

```
Z.od = Y'n;
Z.od⟨opd1⟩;
Y'n = Z.od
```

Ejemplo

```
Y87++;
```

⇒

```
Z.od = Y87;
Z.od++;
Y87 = Z.od;
```

F'_n++; ➡ $F'_n = F'_n + 1;$

F'_n--; ➡ $F'_n = F'_n - 1;$

Ejemplo

F_{44}++; ⇒ $F_{44} = F_{44} + 1;$

⟨opd_1⟩V'_n; ➡ V'_n⟨opd_1⟩;

Ejemplo

++F_{23}; ⇒ F_{23}++;

Z_Ω=Y_n⟨opd_1⟩; ➡ Z_Ω=Y_n;
Y_n⟨opd_1⟩;

Ejemplo

$Z_{.opf}$ = Y_{91}--; ⇒ $Z_{.opf}$ = Y_{91};
Y_{91}--;

Z_Ω=⟨opd_1⟩Y_n; ➡ ⟨opd_1⟩Y_n;
Z_Ω=Y_n;

Ejemplo

```
Z.opf = ++Y8;
```

⇒

```
++Y8;
Z.opf = Y8;
```

$Y_{.\langle id \rangle} = \alpha \ \langle op_1 \rangle \ \beta \ \Rightarrow$

```
Z.op1 = α;
Z.op2 = β;
Z.⟨id⟩ = Z.op1 ⟨op1⟩ Z.op2;
```

Ejemplo

```
Y.r2 = 12 / Y3;
```

⇒

```
Z.op1 = 12;
Z.op2 = Y3;
Z.r2 = Z.op1 / Z.op2;
```

$F_{.\langle id \rangle} = \alpha \ \langle op_1 \rangle \ \beta \ \Rightarrow$

```
Z.op1 = (float)α;
Z.op2 = (float)β;
Z.⟨id⟩ = F:fop$Y$F$F('⟨op1⟩', F.op1, F.op2);
```

Ejemplo

```
F.r1 = 12 - [89$1];
```

⇒

```
Z.op1 = (float)12;
Z.op2 = (float)[89$1];
Z.r1 = F:fop$Y$F$F('-', F.op1, F.op2);
```

$V'_n = res\_y_1(\langle op_1 \rangle)\langle fnp_1 \rangle \ \Rightarrow$

```
Y.r1 = res_y1(⟨op1⟩);
V'n = Y.r1 ⟨fnp1⟩
```

Ejemplo

```
*Y11[0] = 4 + 1 + 7;
```

⇒

```
Y.r1 = 4 + 1;
*Y11[0] = Y.r1 + 7;
```

$V'_n = res\_f_1(\langle op_1 \rangle)\langle fnp_1 \rangle$ ➡

```
F.r1 = res_f1(⟨op1⟩);
V'n = F.r1 ⟨fnp1⟩
```

Ejemplo

```
Y24 = F1 + 8;
```

⇒

```
F.r1 = F1 + 8;
Y24 = F.r1;
```

$V'_n = \langle fna_1 \rangle res\_y_1(\langle opm_1 \rangle)$ ➡

```
Y.r2 = res_y1(⟨opm1⟩);
V'n = ⟨fna1⟩Y.r2
```

Ejemplo

```
F3 = 9 + Y1;
```

⇒

```
F.r2 = 9 + Y1;
F3 = Y.r2;
```

$V'_n = \langle fna_1 \rangle res\_f_1(\langle opm_1 \rangle)$ ➡

```
F.r2 = res_f1(⟨opm1⟩);
V'n = ⟨fna1⟩F.r2
```

Ejemplo

```
F4 = 6 + F22 % Y7;
```

⇒

```
F.r2 = F22 % Y7;
F4 = 6 + F.r2;
```

$\langle tarit \rangle \rightarrow \langle valp\_y \rangle \mid \langle valp\_f \rangle \mid \mathbf{Z}_{\omega}$

$\langle aritr \rangle \rightarrow \varepsilon \mid \langle op \rangle \langle tarit \rangle \langle aritr \rangle$

$\langle exp\_arit \rangle \rightarrow \langle tarit \rangle \langle aritr \rangle$

$V'_n \langle op_1 \rangle = \langle exp\_arit_1 \rangle ;$ ➡

```
Z.od = V'n;
V'n = ⟨exp_arit1⟩;
V'n = V.od ⟨op1⟩ V'n;
```

Ejemplo

```
Y17 *= F22 + F7 / 6;
```

⇒

```
Z.od = Y17;
Y17 = F22 + F7 / 6;
Y17 = Y17 * Y.od;
```

Operaciones relacionales

$\langle oprel \rangle \rightarrow > \mid < \mid >= \mid <= \mid == \mid !=$

$\langle opnb \rangle \rightarrow \mathbf{Z}_{\Omega} \mid \alpha \langle oprel \rangle \beta \mid \mathbf{!(}\alpha \langle oprel \rangle \beta\mathbf{)}$

$Z_{.\langle id \rangle} = \alpha > \beta;$ ➡

```
F.⟨id⟩ = β - α;
Z.⟨id⟩ = negp(F.⟨id⟩);
```

Ejemplo

```
Z.obf = 3 > Y13;
```

⇒

```
F.obf = 3 - Y13;
Z.obf = Y:negp$F(F.obf);
```

$Z_{.\langle id \rangle} = \alpha == \beta;$ ➡

$F_{.\langle id \rangle} = \alpha - \beta;$
$Z_{.op1} = 2;$
$Z_{.op2} = Z_{.\langle id \rangle};$
$Z_{.\langle id \rangle} = Z_{.op1} - Z_{.op2};$

Ejemplo

$Z_{.obf} = Y_{43} == Y_{21};$

⇒

$F_{.obf} = Y_{43} - Y_{21};$
$Z_{.op1} = 2;$
$Z_{.op2} = Z_{.obf};$
$Z_{.obf} = Z_{.op1} - Z_{.op2};$

$Z_{\Omega} = !(\alpha \langle oprel_1 \rangle \beta);$ ➡

$Z_{\Omega} = \alpha \langle oprel_1 \rangle \beta;$
$Z_{.op1} = 1;$
$Z_{.op2} = Z_{\Omega};$
$Z_{\Omega} = Z_{.op1} - Z_{.op2};$

Ejemplo

$Z_{.obf} = !(Y_{36} == F_8);$

⇒

$Z_{.obf} = Y_{36} == F_8;$
$Z_{.op1} = 1;$
$Z_{.op2} = Z_{.obf};$
$Z_{.obf} = Z_{.op1} - Z_{.op2};$

$Z_{\Omega} = \alpha != \beta;$ ➡ $Z_{\Omega} = !(\alpha == \beta);$

Ejemplo

$Z_{.obf} = Y_4 != F_9;$

⇒

$Z_{.obf} = !(Y_4 == F_9);$

$Z_{\Omega} = \alpha < \beta;$ ➡ $Z_{\Omega} = \beta > \alpha;$

Ejemplo

$Z_{.obf} = Y_4 < F_9;$ ⟹ $Z_{.obf} = F_9 > Y_{4;}$

$Z_{\Omega} = \alpha >= \beta;$ ➡ $Z_{\Omega} = !(\alpha < \beta);$

Ejemplo

$Z_{.obf} = Y_4 >= F_9;$ ⟹ $Z_{.obf} = !(Y_4 < F_9);$

$Z_{\Omega} = \alpha <= \beta;$ ➡ $Z_{\Omega} = !(\alpha > \beta);$

Ejemplo

$Z_{.obf} = Y_4 <= F_9;$ ⟹ $Z_{.obf} = !(Y_4 > F_9);$

Operaciones lógicas

⟨*opdb*⟩ → **&&** | ||

⟨*opbs*⟩ → α | **!**α

γ | δ | η → ⟨*opnb*⟩ | $\mathbf{Z}_{\Omega}$

Z_{Ω} = ⟨*opnb*$_1$⟩ || ⟨*opnb*$_2$⟩ ; ➡

$Z_{.ob1}$ = ⟨*opnb*$_1$⟩ ;
$Z_{.ob2}$ = ⟨*opnb*$_2$⟩ ;
Z_{Ω} = $Z_{.ob1}$ + $Z_{.ob2}$;

Ejemplo

```
Z.obf = Y5 < 3 || F5 != 19;
```

⟹

```
Z.ob1 = Y5 < 3;
Z.ob2 = F5 != 19;
Z.obf = Z.ob1 + Z.ob2;
```

Z_{Ω} = ⟨*opnb*$_1$⟩ && ⟨*opnb*$_2$⟩ ; ➡

$Z_{.ob1}$ = ⟨*opnb*$_1$⟩ ;
$Z_{.ob2}$ = ⟨*opnb*$_2$⟩ ;
Z_{Ω} = $Z_{.ob1}$ * $Z_{.ob2}$;

Ejemplo

```
Z.obf = Y5 < 3 && F5 != 19;
```

⟹

```
Z.ob1 = Y5 < 3;
Z.ob2 = F5 != 19;
Z.obf = Z.ob1 * Z.ob2;
```

Z_{Ω} = α ⟨*opdb*$_1$⟩ γ ; ➡ Z_{Ω} = α > 0 ⟨*opdb*$_1$⟩ γ ;

Ejemplo

```
Z.obf = Y5 || F5 != 19;
```

⟹

```
Z.obf = Y5 > 0 || F5 != 19;
```

Z_{Ω} = !α ⟨$opdb_1$⟩ γ; ➡ Z_{Ω} = α == 0 ⟨$opdb_1$⟩ γ;

Ejemplo

```
Z.obf = !Y5 && F5 != 19;
```

⇒

```
Z.obf = Y5 == 0 && F5 != 19;
```

Z_{Ω} = γ ⟨$opdb_1$⟩ ⟨$opbs_1$⟩; ➡ Z_{Ω} = ⟨$opbs_1$⟩ ⟨$opdb_1$⟩ γ;

Ejemplo

```
Z.obf = F5 != 19 && Y5;
```

⇒

```
Z.obf = Y5 && F5 != 19;
```

Z_{Ω} = γ || δ || ➡

```
Z.or = γ || δ;
ZΩ = Z.or ||
```

Ejemplo

```
Z.obf = F12 < Y4 || Y5 < 19 || Y5 > 23;
```

⇒

```
Z.or = F12 < Y4 || Y5 < 19;
Z.obf = Z.or || Y5 > 23;
```

Z_{Ω} = γ || δ && η ➡

```
Z.and = δ && η;
ZΩ = γ || Z.and
```

Ejemplo

```
Z.obf = F12 < Y4 || F12 == 11 && F33 > 87;
```

⇒

```
Z.and = F12 == 11 && F33 > 87;
Z.obf = F12 < Y4 || Z.and;
```

Z_{Ω} = γ && δ ⟨$opdb_1$⟩ ➡ $Z_{.and}$ = γ && δ;
Z_{Ω} = $Z_{.and}$ ⟨$opdb_1$⟩

Ejemplo

```
Z.obf = Y32 =< 5 && Y33 > 11 && Y34 > 20;
```

⇒

```
Z.obf = Y32 =< 5 && Y33 > 11;
Z.obf = Z.and && Y34 > 20;
```

Estructuras de control

⟨*rest_b*⟩ → ε | ⟨*opdb*⟩ γ ⟨*rest_b*⟩

⟨*exp_b*⟩ → γ ⟨*rest_b*⟩

if(⟨$opbs_1$⟩) ➡ if(⟨$opbs_1$⟩|| 0)

Ejemplo

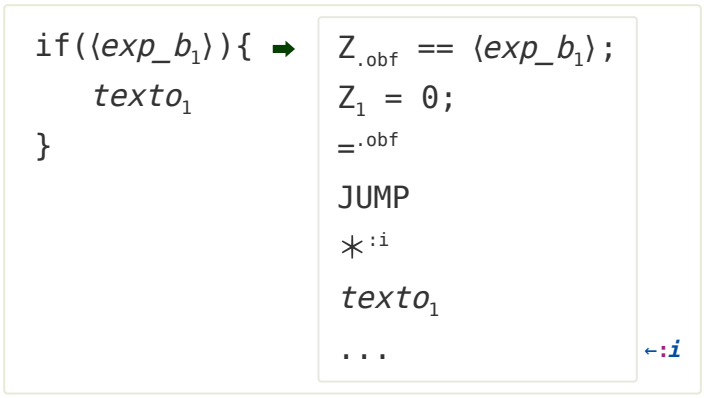
```
if(!Y3){
    Y21 = 16;
}
```

⇒

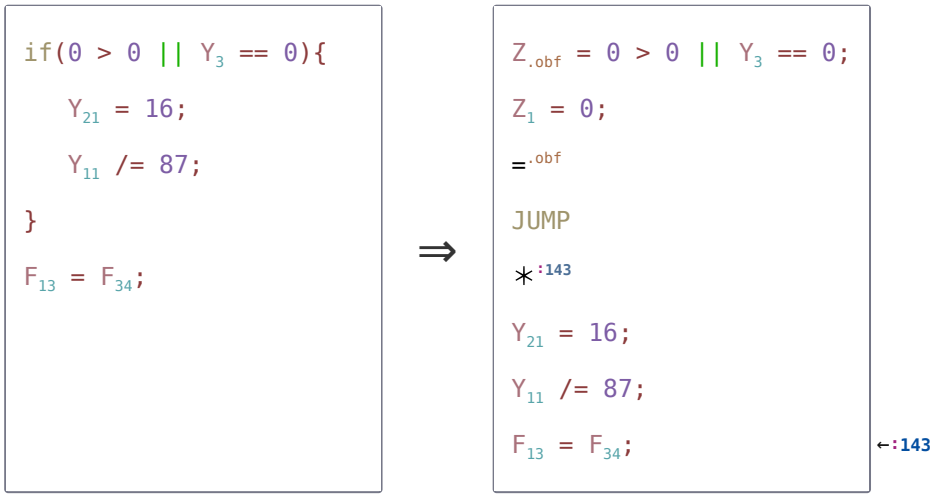
```
if(!Y3 || 0){
    Y21 = 16;
}
```

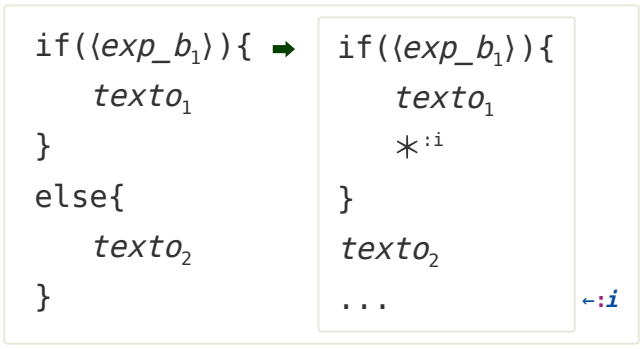
⇒ ... ⇒

```
if(0 > 0 || Y3 == 0){
    Y21 = 16;
}
```


if(⟨exp_b1⟩){
texto1
}
Z.obf == ⟨exp_b1⟩;
Z1 = 0;
=.obf
JUMP
∗:i
texto1
...
←:i

Ejemplo


if(0 > 0 || Y3 == 0){
Y21 = 16;
Y11 /= 87;
}
F13 = F34;
⇒
Z.obf = 0 > 0 || Y3 == 0;
Z1 = 0;
=.obf
JUMP
∗:143
Y21 = 16;
Y11 /= 87;
F13 = F34;
←:143


if(⟨exp_b1⟩){
texto1
}
else{
texto2
}
if(⟨exp_b1⟩){
texto1
∗:i
}
texto2
...
←:i

Ejemplo

```
if(0 > 0 || Y3 == 0){
    Y21 = 16;
    Y11 /= 87;
}
else{
    Y21 = 0;
}
F13 = F34;
```

⇒

```
if(0 > 0 || Y3 == 0){
    Y21 = 16;
    Y11 /= 87;
    *:211
}
Y21 = 0;
F13 = F34;          ←:211
```

⇒

```
Z.obf = 0 > 0 || Y3 == 0;
Z1 = 0;
=.obf
JUMP
*:212
Y21 = 16;
Y11 /= 87;
*:211
Y21 = 0;            ←:212
F13 = F34;          ←:211
```

```
while(⟨exp_b1⟩){ ➡
    texto1
}
```

```
if(⟨exp_b1⟩){       ←:i
    texto1
    *:i
}
```

Ejemplo

```
while(Y3 < 7){
    Y1 *= 2;
    Y3 ++;
}
```

⇒

```
if(Y3 < 7){
    Y1 *= 2;
    Y3 ++;
    ∗:387
}
```

←:387

```
do{                          ➡   texto1
    texto1                       if(⟨exp_b1⟩){
} while(⟨exp_b1⟩);                   ∗:i
                                 }
```

←:i

Ejemplo

```
do{
    Y1 *= 2;
    Y3 ++;
} while(Y3 < 7);
```

⇒

```
Y1 *= 2;
Y3 ++;
if(Y3 < 7){
    ∗:387
}
```

←:387

```
for(texto1;⟨exp_b1⟩;texto2){ ➡   texto1;
    texto3                        while(⟨exp_b1⟩){
}                                     texto3
                                      texto2
                                  }
```

Ejemplo

```
for(Y3 = 0; Y33[Y3] != 'c'; Y3++){
   Y21++;
}
```

⇒

```
Y3 = 0;
while(Y33[Y3] != 'c'){
   Y21++;
   Y3++;
}
```

⟨*casd*⟩ → **case** | **default**

⟨*fin_case*⟩ → **$** | ⟨*casd*⟩

```
switch(α1){ ➡
   texto1
}
```

```
Z.brk = Z:i;
F.sw = α1;
:swc:{$
   texto1
$}
JUMP
1:j
...
```

←:*i*

←:*j*

Ejemplo

```
switch(Y29){
case '0':
   Y21[0]++;
   break;
case '1':
   Y25++;
   break;
}
```

⇒

```
Z.brk = Z:58;
F.sw = Y29;
:swc:{$
   case '0':
      Y21[0]++;
      break;
   case '1':
      Y25++;
      break;
$}
JUMP
1:59                ←:58
...                 ←:59
```

```
break; ➡    Z:i = Z.brk;
}           ∗                ←:i
```

Ejemplo

```
...
break;
   case '1':
      Y25++;
...
```

⇒

```
...
Z:47 = Z.brk;
∗                   ←:47
   case '1':
      Y25++;
...
```

:swc:

✎ ⟨*texto_swc*⟩ representa *textoc* que no empieza por ⟨*fin_case*⟩.

```
:swc:{                    ➡   F.case = α1;
   ⟨texto1⟩                   Z1 = Z.case;
   $                          =.sw
   case α1:                   JUMP
     ⟨texto_swc1⟩             ∗:i
   ⟨fin_case1⟩                :swc:{
                                ⟨texto1⟩
                                ⟨texto_swc1⟩   ←:i
                                $
                                ⟨fin_case1⟩
```

Ejemplo

```
:swc:{$                  :swc:{$                  F.case = 48;
   case '0':                case 48:               Z1 = Z.case;
      Y21[0]++;                Y21[0]++;           =.case
      break;                   break;              JUMP
   case '1':                case '1':              ∗:65
      Y25++;                   Y25++;              :swc:{
      break;          ⇒        break;      ⇒             Y21[0]++;      ←:65
$}                       $}                              break;
                                                    $
                                                    case '1':
                                                       Y25++;
                                                       break;
                                                 $}
```

```
:swc:{                    ➡   :swc:{
   ⟨texto_1⟩                     ⟨texto_1⟩
   $                             ⟨fin_case_1⟩   ←:i
   default:                      break;
     ⟨texto_swc_1⟩               $
   ⟨fin_case_1⟩                  case Y.sw:
$}                                 ∗:i
                                $}
```

Ejemplo

```
:swc:{$                          :swc:{$
...                              ...
   default:                           Y36++;      ←:127
     Y36++;                      case '1':
   case '1':                          Y25++;
     Y25++;                           break;
     break;                           break;
$}                                 case Y.sw:
                                      ∗:127
                                $}
```

⇒

```
:swc:{                    ➡   :swc:{
   ⟨texto_1⟩                     ⟨texto_1⟩
   $                             $
   case α_1:                     case α_1:
   ⟨fin_case_1⟩                  NADA
                                 ⟨fin_case_1⟩
```

Ejemplo

```
:swc:{$
   ...
   case 49:
   case '9':
      Y25++;
      break;
$}
```

⇒

```
:swc:{
   ...
   $
   case 49:
   NADA
   case '9':
      Y25++;
      break;
$}
```

```
:swc:{          ➡   break;
   ⟨textoc1⟩          ⟨textoc1⟩
   $
$}
```

Librería estándar

#include tipos

⬇

```
unsigned int pow(unsigned int base, unsigned int exp){
    unsigned int res;
    Y.obf = 1 - base;
    if(Z.obf){
      return 0;
    }
    Y.obf = 1 - exp;
    if(Z.obf){
      res = 1;
    }
    else{
      exp--;
      res = base * pow(base, exp);
    }
    return res;
}

unsigned int upow(unsigned int d, unsigned int v){
    unsigned int res;
    Y.obf = v % d;
    if(Z.obf){
      res = 0;
    }
    else{
      v = v / d;
      res = 1 + upow(d, v);
   }
   return res;
}
```

```
float litf(unsigned int num, unsigned int den){
    Y.obf = 1 - num;
    if(Z.obf){
      Z.ret = 1;
      return F.ret;
    }
    Y.obf = 1 - den;
    if(Z.obf){
      STOP
    }
    num = pow(5, num);
    den = pow(3, den);
    Y.ret = num * den;
}

unsigned int negp(float v){
    Z.opf = v;
    Z.opf++;
    Y.ret = Y.opf % 2;
}

float negar(float v){
    Z.opf = v;
    Y.obf = 2 - Y.opf;
    if(Z.obf){
       return 1;
    }
    Y.obf = negp(v);
    if(Z.obf){
       Y.ret = Y.opf / 2;
    }
    else{
       Y.ret = Y.opf * 2;
    }
}
```

```
unsigned int ftoy(float v){
    unsigned int int, num, den, c;
    Z.opf = v;
    Y.obf = 2 - Y.opf;
    if(Z.obf){
       return 0;
    }
    v = Z.opf;
    num = upow(5, num);
    den = upow(3, den);
    Y.ret = num / den;
}

float ytof(unsigned int v){
    return litf(v, 1);
}
```

#include arit_float

⬇

```
#include tipos

float sumaf(float n1, float n2){
    unsigned int num1, num2,
                 den1, den2,
                 sig1, sig2, sigf;
    float res;
    den1 = upow(3, n1);
    den2 = upow(3, n2);
    num1 = upow(5, n1) * den2;
    num2 = upow(5, n2) * den1;
    den2 += den1;
    sig1 = negp(n1);
    sig2 = negp(n2);
    sigf = sig1 - sig2;
    sigf += sig2 - sig1;
    Y.obf = 1 - sigf;
    if(Z.obf){
       num2 += num1;
       sigf = sig1 + sig2;
    }
    else{
       Y.obf = num1 - num2;
       if(Z.obf){
          num2 = Z.obf;
          sigf = sig1;
       }
       else){
          num2 -= num1;
          sigf = sig2;
       }
    }
```

```
    res = litf(num2, den2);
    Z.obf = sigf;
    if(Z.obf){
       res = negar(res);
    }
    return res;
}

float restaf(float n1, float n2){
    n2 = negar(n2);
    return sumaf(n1, n2);
}

float multf(float n1, float n2){
    unsigned int numf, denf,
                 sig1, sig2, sigf;
    float res;
    numf = upow(5, n1) * upow(5, n2);
    denf = upow(3, n1) * upow(3, n2);
    sig1 = negp(n1);
    sig2 = negp(n2);
    res = litf(numf, denf);
    sigf = sig1 - sig2;
    sigf += sig2 - sig1;
    Z.obf = sigf;
    if(Z.obf){
       res = negar(res);
    }
    return res;
}
```

```
float divf(float n1, float n2){
    unsigned int num, den;
    Z.op2 = n2;
    Z.op1 = 2;
    Z.obf = Z.op1 - Z.op2;
    if(Z.obf){
       STOP
    }
    num = upow(5, n2);
    den = upow(3, n2);
    n2 = litf(den, num);
    return multf(n1, n2);
}
```

```
#include malloc ➡
```

```
void malloc(unsigned int n){
    Z.opf = n;
    Z:$npos += Z.opf;
}

void free(unsigned int n){
    Z.opf = n;
    Z:$npos -= Z.opf;
}
```

```
#include cmmstd ➡
```

```
#include arit_float
#include malloc
```

Bibliografía

[1] Alan M. Turing (1936): *On computable numbers, with an application to the entscheidungsproblem.* Incluído en «The essential Turing», Oxford University Press, 2004, editado por B. Jack Copeland.

[2] Stephen C. Kleene (1952): *Introduction to Metamathematics.* Ishi Press, 2009.

[3] A.J. Kfoury, Robert N. Moll y Michael A. Arbib (1982): *A programming approach to computability.* Springer-Verlag.

[4] Georg Cantor (1895–1897): *Contributions to the founding of the theory of transfinite numbers.* Traducido y preparado por Philip E. B. Jourdain. Dover, 1955.

[5] David Hilbert y Wilhelm Ackermann (1938): *Principles of mathematical logic.* Chelsea Publishing, 1950.

[6] George S. Boolos, John P. Burgess y Richard C. Jeffrey (2007): *Computability and logic.* Cambridge University Press.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi y Jeffrey D. Ullman (2006): *Compilers: Principles, Techniques, and Tools.* Pearson Education, Inc.

[8] Brian W. Kernighan y Dennis M. Ritchie (1988): *The C Programming Language.* Prentice Hall.

Recursos

Para consultas esporádicas he usado sitios en Internet, principalmente las páginas «fundeu.com» y «rae.es» para dudas sobre ortografía y gramática y el fantástico proyecto que es Wikipedia para dudas en general.

Para escribir el documento he usado el (algo más que) editor Emacs 26.2 en el sistema operativo Ubuntu 18.04.4 LTS. Está maquetado con TEX de Donald E. Knuth, a excepción de los apéndices B a D, que he maquetado con HTML y CSS. Para la interpretación del documento he utilizado el motor XeTeX creado por Jonathan Kew.

Las fuentes utilizadas son: «TeX Gyre Schola», de B. Jackowski, P. Strzelczyk y P. Pianowski; «Kleist-Fraktur», usada con permiso de Dieter Steffmann; «Gabriele Bad», usada con permiso de Andreas Höfeld; «Liberation», «XITS», «Waree» y «DejaVu», además de las Computer Modern de Donald Knuth.