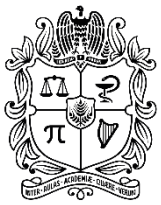


UNIVERSIDAD
NACIONAL
DE COLOMBIA

El lenguaje de programación Komodo

César Danilo Pedraza Montoya
cpedraza@unal.edu.co

Universidad Nacional de Colombia
Facultad de Ciencias
Departamento de matemáticas
Bogotá, Colombia
2025

Índice

1. Introducción

Komodo es un lenguaje de programación hecho para probar ideas rápidamente. Es ideal para problemas con estructuras discretas como números y palabras. Komodo intenta que operar con estas entidades sea tan fácil como sea posible mientras se minimiza la cantidad de código necesario para llegar a una implementación exitosa. La otra prioridad de Komodo es la sencillez: se busca que el lenguaje sea pequeño y con reglas simples, con el propósito de que pueda ser aprendido con facilidad.

Komodo está diseñado con la intención de convertirse en una herramienta útil para generar estructuras discretas que puedan depender de muchas restricciones, para así estudiarlas. Esta es una tarea común en el estudio de areas de las matemáticas como la combinatoria, la teoría de la computación, la teoría de grafos o la teoría de códigos. Usar el computador como una herramienta de exploración matemática es una práctica conocida como matemática experimental. [1, p. 2]

Este documento describe el lenguaje de programación Komodo. No es una guía de uso del lenguaje. También se exploran detalles del intérprete de Komodo creado por el autor. Sin embargo, no se describe todo el comportamiento esperado de una implementación del lenguaje, ni se proveen detalles del intérprete más allá de lo estructural.

2. Visión general

El propósito de Komodo tiene consecuencias en su diseño. Puesto que Komodo es un lenguaje para *scripting*, no es una prioridad que el lenguaje se procese a si mismo, o que la representación de los datos sea similar a la representación de los programas. Asimismo, el nivel de abstracción de Komodo y la etapa en que se encuentra el proyecto hacen preferible implementar un intérprete en lugar de un compilador.

El diseño de Komodo no es deliberado, sino que se ha llegado a él con una construcción iterativa.

En esta sección se explican brevemente las características de Komodo, que son descritas con mayor detalle en secciones posteriores.

2.1. Sistema de tipos

Komodo es un lenguaje con tipado débil y dinámico. Lo primero significa que las reglas de tipos son relativamente laxas y se realizan conversiones implicitas de tipos, y lo segundo significa que estas reglas y conversiones son verificadas y realizadas en tiempo de ejecución. Esto es así por varias razones:

- Komodo está pensado para que las anotaciones de tipos sean totalmente opcionales, por lo que en general no es posible inferir los tipos de todas las variables en tiempo de compilación.
- Hace posible la implementación de un intérprete sin añadir análisis semántico, lo que fue útil para llegar rápido a un prototipo funcional.
- Komodo está pensado para realizar algunas conversiones de tipos implicitamente, lo que necesariamente implica que las restricciones de tipos son más ligeras.

Komodo también es de tipado latente, lo que significa que los tipos están asociados a valores y no a variables o símbolos. Esto hace que un símbolo pueda tener valores tipos distintos en momentos distintos. [2, p. 2].

Además, Komodo tiene tipado gradual. Se realiza chequeo de tipos en tiempo de ejecución cuando el usuario provee restricciones de tipos en las firmas de funciones y variables.

2.2. Paradigmas

Komodo emplea dos paradigmas de programación: procedural y funcional.

Por un lado, Komodo es un lenguaje procedural porque las formas y el orden importan: las sentencias de un programa son evaluadas en el orden en que aparecen en el mismo. Esto establece una semántica clara para cambiar el valor de una variable sin que esta sea mutable, por ejemplo. (veáse Sección 4.2.4.)

Además, en Komodo las funciones tienen un papel protagónico: pueden declararse de forma nombrada o anónima, pueden rastrear patrones y pueden pasarse como argumentos a otras funciones. La búsqueda de patrones de Komodo y su inclusión en las funciones permite describir procedimientos en términos de lo que deben hacer, en lugar de como.

Esto permite que dependiendo del problema, un programa de Komodo pueda ser más imperativo o más declarativo a conveniencia. La forma en que se restringe la combinación de los dos paradigmas es limitando la mutabilidad de valores.

2.3. La estructura del intérprete

Como suele suceder con múltiples compiladores e intérpretes, el intérprete de Komodo funciona como una cadena de procesamiento. Se comienza procesando texto, y tras cada paso se obtiene una representación del programa más preparada para ser ejecutada. En el caso de Komodo, se tienen las siguientes etapas:

1. Analizador léxico,
2. Analizador sintáctico,
3. Post-analizador sintáctico o *weeder*,
4. Evaluador,
5. Entorno de tiempo de ejecución o *runtime environment*.

Este es un diagrama de secuencia de los componentes del intérprete. Las columnas son los componentes, y las flechas son interfaces. En algunos casos las interfaces son estructuras de datos, y en otros son eventos invocados por el usuario.

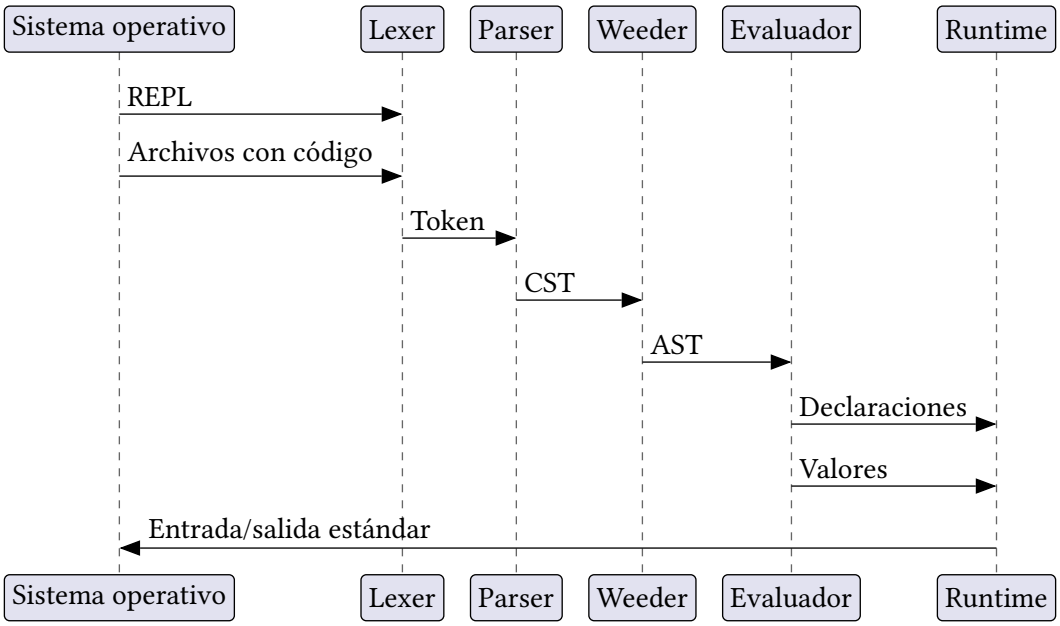


Figura 1: Componentes del intérprete y sus relaciones más importantes.

En este documento se hace una descripción del funcionamiento de cada componente y cada relación entre componentes, al mismo tiempo que se especifican aspectos de Komodo.

3. Análisis léxico y sintáctico

3.1. Analizador léxico o *lexer*

El analizador léxico convierte un programa, una sucesión de caracteres, en una sucesión de *tokens*, que son unidades más complicadas como palabras, números y símbolos. Uno de los propósitos de esta fase es que las demás fases no tengan que lidiar con detalles relacionados al texto que representa el programa: Las fases posteriores no deberían lidiar con aspectos como espacios en blanco, indentación o comentarios en el código. Toda la información necesaria debería estar incluída en los *tokens* que el analizador emite.

La entrada del analizador es un *stream* de caracteres Unicode. Sin embargo, la mayoría de palabras clave y símbolos se componen de caracteres ASCII. La salida es un *stream* de *tokens*. El *lexer* pasa una sola vez por el texto de entrada para emitir todos los *tokens* correspondientes, y el texto es recorrido conforme los *tokens* son emitidos.

```
let x := 2
```

↓

```
Let, Ident(x), Assign, Integer(2)
```

Listado 1: Ejemplo de paso de un texto a una sucesión de *tokens*

3.1.1. Rastreo de indentación y el alcance del analizador léxico

Los *tokens* `Indent` y `Dedent` indican el inicio y el final de un bloque de código indentado, respectivamente. Por ejemplo, en el siguiente fragmento de código

```
let f(x) := x + 2
```

Listado 2: Ejemplo de declaración de una función

el cuerpo de la función `f` está compuesto exactamente por la expresión `x + 2`. Si se requiere que el cuerpo de la función tenga más líneas de código, se puede iniciar un bloque en una nueva línea. Las líneas que pertenecen al bloque están espaciadas a la derecha por 4 espacios:

```
let f(x) :=
    let y := 2
    x + y
```

Listado 3: Ejemplo de declaración de una función con un bloque de código.

En este caso, la función `f` esta compuesta por un bloque de código de dos líneas.

Para indicar el inicio de este bloque, el *lexer* emite un token `Indent` antes de emitir los correspondientes al mismo. Tras haber emitido todos los *tokens* del bloque, se emite un `Dedent` para indicar el fin de este.

Este comportamiento también ocurre cuando hay bloques dentro de bloques, como en el siguiente programa:

```
for i in 0..10 do
    # se emite el primer Indent
    for j in 0..10 do
        # se emite el segundo Indent
        mat[i][j] = i + j
# Se emiten dos Dedent seguidos
```

Listado 4: Ejemplo de bloques anidados.

En este caso, el cuerpo del primer ciclo `for` es un un bloque de código, cuya única parte es otro ciclo `for`, cuyo cuerpo es otro bloque de una sola línea. Después del primer `do` se emite un `Indent`. Después del segundo `do` se emite otro `Indent`. Cuando se llega al final del texto, se emiten dos `Dedent` seguidos para «cerrar» los dos bloques de código que estaban «abiertos».

La razón para hacer esto es que al emitir estos *tokens* se pueden entender los bloques de código de la misma forma que se hace con lenguajes donde los bloques están delimitados con caracteres como corchetes. Por ejemplo en JavaScript [3], este es un programa similar:

```
for (let i = 0; i < 10; i++) {
    for (let j = 0; j < 10; j++) {
        mat[i][j] = i + j;
    }
}
```

Listado 5: Bloques anidados en JavaScript.

Los corchetes aquí cumplen la misma función que los `Indent` y `Dedent` en Komodo, solo que en este caso tienen una correspondencia directa con caracteres del texto. En el caso de Komodo son un artificio obtenido de contar espacios en blanco.

Esta es una descripción de como el *lexer* decide emitir estos *tokens*:

1. El *lexer* cuenta el nivel de indentación en el que se encuentra el programa en el punto en donde el texto está siendo leído. Cuando se está al principio del programa, este nivel es cero.
2. Cuando se llega a una nueva línea, se quiere contar su nivel de indentación. Esto se hace contando el número de espacios al principio de la línea. Cada 4 espacios son un nivel de indentación. Si quedan espacios sobrantes (es decir, el número de espacios no es múltiplo de 4), se ignora el residuo.

   Por ejemplo, la línea de código

   ```
   println(x)
   ```

   tiene 8 espacios al principio, por lo que su nivel de indentación es 2.

   Las líneas que estan compuestas exclusivamente de espacios o comentarios son ignoradas.
3. Una vez que se consumen y cuentan los espacios, y que se llega a un caracter que va a componer un *token*, se compara el nivel de indentación de la línea con el nivel de indentación que el *lexer* guarda.
   - Si son iguales, no se emiten *tokens* de más: el resto de la línea es consumida.
   - Si el nivel de la línea es mayor, se emiten tantos `Indent` como la diferencia entre el nivel de la línea y el nivel guardado en el *lexer*, y se consume el resto de la línea.
   - Si el nivel de la línea es menor, se emiten tantos `Dedent` como la diferencia entre el nivel guardado en el *lexer* y el nivel de la línea. Luego se consume el resto de la línea.

Cabe destacar que la razón por la que hay que almacenar el nivel de indentación es por que las reglas con las que estos *tokens* son emitidos son dependientes del contexto: no basta con conocer el caracter actual o una cantidad fija hacia adelante, sino, en general, es necesario poder recorrer todos los caracteres recorridos antes. Una solución más sensible es almacenar un estado útil (el nivel de indentación) para poder decidir cuando emitir los *tokens*.

Esta estrategia es la misma que usa el intérprete principal de Python, CPython.

3.2. Analizador sintáctico o *parser*

El analizador sintáctico convierte sucesiones de *tokens* en nodos de un árbol que describe la estructura sintáctica del programa, conocido como CST (del inglés *Concrete Syntax Tree*). Este árbol contiene todos los detalles del programa, y es generado casi en su totalidad de forma independiente del contexto. La mayoría de la estructura del programa se obtiene de este paso.

El *parser* recibe un *stream* de *tokens*, y retorna un *stream* de nodos del CST. En este punto, un programa es una sucesión de nodos del CST.

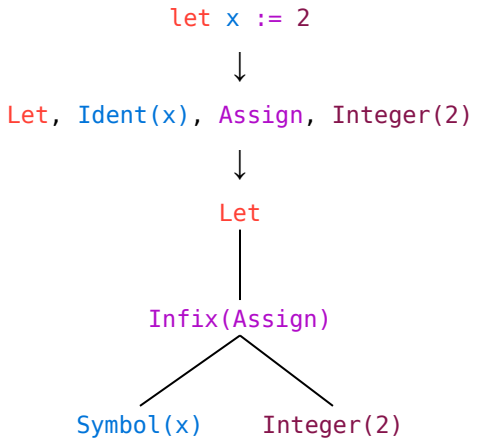
Komodo es un lenguaje orientado a expresiones, lo que significa que hay una preferencia explicita a que las sentencias del lenguaje retornen un valor.

Komodo tiene algunas sentencias cuya interpretación más natural son como declaraciones, pero aún así retornan un valor, que usualmente es la tupla vacía `()`.

Esto hace que en el análisis sintáctico todo sea considerado una expresión. No se define una distinción entre declaraciones y expresiones.

El análizador sintáctico de Komodo es de descenso recursivo. Esto significa que está compuesto de funciones que se llaman mutuamente, donde (casi siempre) una función se encarga de procesar exclusivamente una de las expresiones del lenguaje.

De forma similar a como ocurre con el analizador léxico, el analizador sintáctico solo pasa por el *stream* de *tokens* una vez para analizar todo el programa. No es necesario hacer regresos a partes del *stream* previamente recorridas.



Listado 6: Ejemplo de paso de una sucesión de *tokens* a un nodo de CST

Para el análisis de expresiones infijas, el *parser* usa el algoritmo de escalada de precedencia (*precedence climbing* en Inglés). Este es un algoritmo iterativo que funciona bien dentro de un analizador de descenso recursivo, siendo más simple que algunas de las alternativas, como el algoritmo *Shunting Yard*. [4]

3.3. Post-analizador sintáctico o *weeder*

El *weeder* toma un nodo del CST y realiza dos tareas:

- Eliminar detalles innecesarios para la evaluación del código,
- Verificar condiciones del programa que serían más dificiles de verificar en etapas anteriores.

El resultado es un nodo de un árbol de sintaxis abstracto o AST (del inglés *Abstract Syntax Tree*), que no contiene detalles como la precedencia de operadores, espacios o indentación. También convierte ciertos operadores infijos en nodos más restringidos, para facilitar la evaluación y eliminar estados

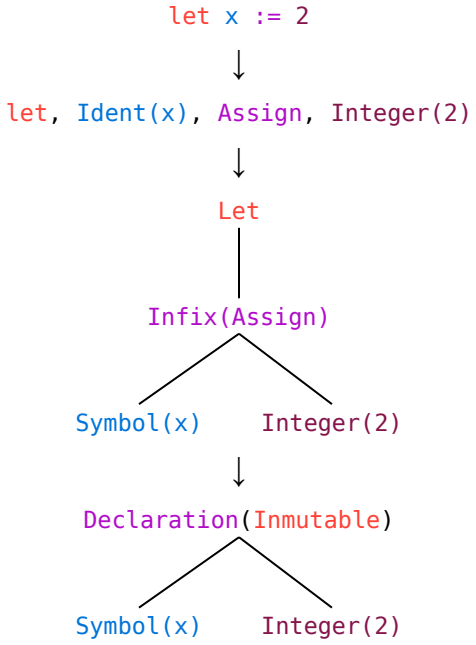
indeseables. El tipo de errores que el *weeder* captura son de naturaleza sintáctica y en muchas ocasiones, dependientes del contexto.

A diferencia del *lexer* y del *parser*, cuyas entradas son *streams*, la entrada del *weeder* es un nodo individual del CST. Cuando un programa es analizado, el *weeder* pasa por cada uno de los nodos retornados por el *parser* de forma independiente.

La tarea del *weeder* es reescribir los nodos del CST para convertirlos en nodos del AST. Este proceso puede fallar cuando el nodo de entrada no cumple características que el *weeder* verifica. Por lo tanto, el *weeder* puede retornar un nodo del AST o un error reportando la restricción que la entrada no cumple.

Otra de las razones para añadir el *weeder* como una fase independiente en lugar de integrar sus funciones al *parser*, es controlar la complejidad del *parser*, que puede empezar a abarcar muchas reglas rápidamente. Al costo de aumentar el número de componentes y hacer al intérprete potencialmente más lento, se conserva la facilidad para entender y modificar el *parser*. Por esta razón, hay transformaciones que se realizan en el *weeder* a pesar de que podrían realizarse en el *parser* sin tanta dificultad.

El cambio de nodos del CST a nodos del AST también deja atrás información que ya no es relevante, como la precedencia de operadores. Esto crea barreras más rigidas entre las fases del intérprete, evitando que se acoplen [5, p. 83] demasiado.



Listado 7: Ejemplo de paso de un nodo de CST a uno de AST

4. Ejecución de programas

4.1. El modelo de ejecución

Un programa de Komodo está hecho de módulos de código. Un modulo de código es creado cada vez que:

- se ejecuta un archivo con código,
- se inicia una sesión del REPL.

Un archivo con código es ejecutado cuando el usuario lo solicita usando la interfaz de línea de comandos, o cuando es importado desde otro módulo.

Toda ejecución de un módulo de código tiene su propio entorno.

```
# foo.komodo                    # bar.komodo
let foo(x) := x * x             let bar() := 5
```

↓

```
from "./foo.komodo" import foo
from "./bar.komodo" import bar
let x := bar()
println(foo(x))
```

Listado 8: Ejemplo de un programa de Komodo. Hay 3 módulos y 3 entornos.

El que haya una correspondencia exacta entre entornos y módulos de código es conveniente para razonar fácilmente sobre los módulos: son unidades aisladas que se comunican entre si mediante la importación de variables.

4.1.1. Entornos

Un entorno está compuesto de una pila de *scopes*. Un *scope* es una tabla que hace corresponder nombres con objetos.

El estado inicial de todo entorno tiene un *scope*, y siempre va a tener al menos un *scope*.

Se añade un nuevo *scope* al entorno cada que:

- Se ejecuta una función,
- Se ejecuta un ciclo,
- Se ejecuta un bloque de código indentado.

Después de ejecutar el código en cada uno de estos casos, el *scope* es eliminado del entorno.

Cuando se ejecutan archivos con código, los entornos también guardan la ruta del archivo en el sistema de archivos local, y la ruta de la terminal donde fue ejecutado el intérprete.

4.1.2. Evaluador

Todos los módulos de código son ejecutados por separado. Para ejecutar un módulo de código, se ejecuta cada uno de los nodos del AST que lo componen, en orden, con el mismo entorno que va siendo potencialmente modificado tras cada ejecución. El estado inicial del entorno es el descrito previamente.

Esto hace que un entorno sea el único lugar donde se conserva el estado de ejecución de un módulo.

4.2. Variables

Komodo permite la declaración de variables inmutables usando la palabra clave `let`. También se permite la creación de variables mutables con la palabra clave `var`.

```
var x := 5
let y := a -> a + a * 2
```

Listado 9: Ejemplos de declaraciones en Komodo.

4.2.1. Resolución de nombres

Cuando un nombre es referenciado en el código, se busca en el entorno de la siguiente forma:

- Si se referencia para ser mutado (por ejemplo, al escribir `x := 5`), se comienza buscando desde el *scope* al tope de la pila hasta el de más abajo. Se retorna la primera coincidencia encontrada. Si la variable no se encuentra en el *scope* al tope, se interrumpe la búsqueda cuando se pasa por el scope generado por la ejecución de una función. Si la variable no es encontrada, también se interrumpe la ejecución con un error.

Por ejemplo, en el siguiente ejemplo de código, la variable `x` es encontrada y modificada:

```
var res := 0
for i in 0..3 do
    res := res + i

assert(res = 3)
```

Listado 10: Ejemplo de uso ordinario de una variable mutable.

Sin embargo, en este ejemplo el intérprete retorna un error:

```
var res := 0
let f() :=
    for i in 0..3 do
        res := res + i

f()
assert(res = 3)
```

Listado 11: Ejemplo de uso no permitido de una variable mutable.

En la implementación actual, el intérprete de Komodo comunica que la variable existe, pero que no puede ser mutada.

- Si se referencia una variable para obtener su valor (por ejemplo, al escribir `cur + 10`), se comienza buscando desde el *scope* al tope de la pila hasta el de más abajo. Se retorna la primera coincidencia encontrada. Si la variable no es encontrada, se interrumpe la ejecución del programa con un error.

4.2.2. Copiado de valores

Komodo no tiene una noción de referencia. En el contexto de la dicotomía valor-referencia, en Komodo solo se manipulan valores. Esto hace que los detalles internos sobre referencias a valores y el copiado de valores sean invisibles al usuario.

El intérprete usa referencias siempre que es posible. Cuando una variable inmutable es asignada como el valor de otra variable inmutable, lo que se obtiene es una referencia a la variable original.

Sin embargo, siempre que esta variable haga parte de un cálculo o un procedimiento, se va a hacer una copia.

Cuando el valor de una variable es asignado a otra variable mutable, siempre se hace una copia.

Salvo por los tipos `Char` y `Bool`, la inicialización de todos los tipos de Komodo requieren la solicitud de memoria en tiempo de ejecución. Por esta razón, se prefiere la creación de referencias en lugar de crear copias.

4.2.3. Variables y tipos

En Komodo, los tipos están asociados a valores, y no a variables. Esto permite que una variable pueda ser declarada con un valor con cierto tipo, y luego se le pueda asignar otro valor, con otro tipo. Esto es conocido como tipado latente.

```
var x := 2
x := "2"
```

Listado 12: Ejemplo de una variable mutada con distintos tipos.

4.2.4. Ocultamiento o *shadowing*

Una variable puede ser declarada varias veces con el mismo nombre, incluso en el mismo *scope*. Esto se conoce como *shadowing*. Es una característica conveniente dada la tendencia del intérprete a funcionar con referencias, y es un medio para reciclar nombres en rutinas donde esto es útil.

Cabe destacar que el ocultar una variable con un nuevo valor no afecta los usos previos al ocultamiento. Un ejemplo de esto se muestra en el siguiente programa:

```
let f() := 1
let g() := f()

let f() := 2
let h() := f()
let f() := 3

assert(g() = 1)
assert(h() = 2)
```

Listado 13: Ejemplo de *shadowing*.

Aquí, la función `g` retorna 1, que es el valor de la función `f` cuando `g` fue definida. Lo mismo sucede con la función `h`, que retorna 2. El hecho de que luego `f` retorne 3 no afecta a ningún uso previo.

4.2.5. Mutabilidad restringida

La mutabilidad de variables está restringida por dos reglas:

- Las únicas variables mutables son las que han sido inicializadas usando `var`. Por ejemplo, la asignación dentro de esta función no está permitida, pues hay una asignación a uno de los argumentos:

```
let f(x) :=
    println(x)
    x := x - 1
```

Listado 14: Ejemplo de asignación ilegal a un argumento (siempre son inmutables).

- Dentro de una función, no se puede modificar el valor de una variable definida fuera de la función. Este es un ejemplo mínimo:

```
var x := 0
let f() :=
    x := 1
```

Listado 15: Ejemplo de asignación ilegal a una variable.

El propósito de estas reglas es restringir los casos de uso de un estado mutable.

4.3. Importación de código

Komodo tiene sintaxis para importar módulos de código externos, ya sea de la librería estándar o de archivos con código del sistema de archivos local.

Por ejemplo, para importar funciones del módulo `utils` de la librería estándar, basta escribir

```
from utils import (map, reduce)
(0..5)
    .map(a -> a*a)
    .reduce((acc, cur) -> acc + cur, 0)
```

Listado 16: Ejemplo de uso de la librería estándar.

Por otro lado, para importar código de un archivo local, hay que pasar una cadena con la ruta correspondiente, en lugar del nombre del módulo. Este es un ejemplo:

```
from "/tmp/foo.komodo" import VALUE
println(VALUE)
```

Listado 17: Ejemplo de importación de código externo.

Komodo permite la importación de cualquier valor, no sólo funciones.

4.3.1. Comportamiento de las sentencias `import`

Las sentencias `from <module> import <values>` pueden ponerse en cualquier parte de un programa. Siempre retornan la tupla vacía `()`.

Este es el procedimiento que realizan:

- Se obtiene un entorno derivado del módulo solicitado.
  - Si el módulo solicitado corresponde a código de Komodo, todo este código es ejecutado, obteniendo un entorno.
  - Si el módulo solicitado no corresponde a código de Komodo, entonces debe ser un módulo de la librería estándar para el que se creó un entorno correspondiente previamente, que es retornado.
- Ya con el entorno, los nombres solicitados son obtenidos del mismo e introducidos en el scope al tope del entorno del módulo que se está ejecutando actualmente (en el que fue escrita la sentencia).

Nótese que una sentencia `import` puede ponerse en cualquier punto de un programa, por lo que puede afectar un *scope* específico. En el siguiente ejemplo, se realiza una importación y solo se añaden los nombres importados al *scope* donde se realizó la importación:

```
let f(x) :=
    from math import sqrt
    x + sqrt(x)

sqrt(5) # error!
```

Listado 18: Importación en un *scope* específico.

En este caso se importó la función `sqrt` dentro del *scope* de la función `f`, por lo que no afecta a los *scopes* anteriores, y la función no será encontrada en estos.

Otra característica a destacar de la importación de módulos es que al importar elementos de un archivo con código, todo el archivo es ejecutado; independientemente de cuantos nombres se soliciten.

4.4. Búsqueda de patrones o *Pattern matching*

La búsqueda de patrones es una forma de verificar propiedades en valores de Komodo. Por ejemplo, el siguiente programa busca patrones en una lista:

```
let len(list: List) :=
    case list do
      [] => 0
      [_|tail] => 1 + len(tail)
```

Listado 19: Ejemplo de búsqueda de patrones en Komodo.

En este ejemplo hay una lista de parejas, que hacen corresponder patrones y resultados.

El primer patrón expresa que si la lista referenciada por `list` esta vacía, su longitud es 0.

El segundo dice que si la lista está compuesta de un elemento al principio y otra lista con los demás, referenciada con `tail`, su longitud es de 1 más la longitud de `tail`.

La búsqueda de patrones permite describir procedimientos como listas de reglas. También permite usar la estructura de un valor para obtener otros valores de su interior. Vamos a describir estos casos de uso.

4.4.1. Descripción de procedimientos

Como se mostró en el ejemplo anterior, se pueden describir procedimientos con patrones. Komodo tiene dos mecanismos para hacer esto.

- Patrones en funciones

Las funciones en Komodo pueden definirse en varias declaraciones separadas, donde se pueden poner patrones diferentes. Por ejemplo, esta es una forma de definir la función de Fibonacci:

```
let fib(0) := 0
let fib(1) := 1
let fib(n) := fib(n - 1) + fib(n - 2)
```

Listado 20: Función de Fibonacci en Komodo.

Los parámetros de la función pueden escribirse como patrones, que cuando la función sea llamada, serán comparados con los argumentos en orden. El resultado asociado a la primera lista de patrones que sea compatible con los argumentos será el resultado de la llamada. Sin ningún patrón es compatible, el programa se detiene con un error.

- Expresiones `case`

No es necesario usar funciones para escribir procedimientos con patrones. Se puede usar una expresión `case`:

```
case x % 2 do
    0 => "x es par"
    1 => "x es impar"
```

Listado 21: Ejemplo de uso de una expresión `case`.

El comportamiento es el mismo: los patrones son comparados con la expresión en orden, y el primer patrón compatible determina el resultado. Si ningún patrón es compatible, el programa se detiene con un error.

4.4.2. Desestructuración

Se pueden usar patrones en definiciones para extraer valores del interior de otros valores:

```
let coordinates(n) := (n + 1, n * 2)
let (x, y) := coordinates(0)
assert(x = 1)
assert(y = 0)
```

Listado 22: Ejemplo de desestructuración en Komodo.

En este ejemplo, se compara el patrón a la izquierda de la asignación con el valor de `coordinates(0)`. En este caso el patrón es compatible: el valor es `(1, 0)`. Así, se asigna a `x` el valor `1` y a `y` el valor `0`.

Cuando el patrón no es compatible, el programa se detiene con un error.

También se pueden desestructurar valores en ciclos `for`:

```
from utils import map
let coordinates(n) := (n + 1, n * 2)
for (x, y) in (0..5).map(coordinates) do println(x + y)
```

Listado 23: Ejemplo de desestructuración en un ciclo `for`.

El comportamiento es el mismo que se da cuando se hace una declaración, solo que se repite al principio de cada iteración.

4.5. Tipos

Komodo es un lenguaje con tipado latente, gradual y dinámico. Esto hace considerar fácilmente a Komodo como un lenguaje de tipado débil. Describamos estas características.

4.5.1. Latente

Los tipos de Komodo no están asociados a símbolos, sino a valores. La principal consecuencia de esto es que un símbolo puede tener valores de distintos tipos en momentos diferentes. Esto es útil para la reutilización de nombres, por ejemplo.

4.5.2. Gradual

Se pueden añadir chequeos de tipos a un programa de Komodo de manera opcional, y a conveniencia. Estos chequeos se realizan en tiempo de ejecución. Por ejemplo, en este programa hay un chequeo de tipos:

```
let first(list: List) := list[0]
```

Listado 24: Chequeo de tipos en Komodo.

Nótese que hay patrones que realizan chequeos de tipos implicitos:

```
let some({res|_}: Set) := res
```

Listado 25: Chequeo de tipos redundante.

En este ejemplo, verificar que la entrada es de tipo `Set` es redundante, pues el patrón `{res|_}` solo es compatible con valores de tipo `Set`. Bastaría con escribir la función así:

```
let some({res|_}) := res
```

Listado 26: Chequeo de tipos implícito.

4.5.3. Dinámico

Todos los chequeos de tipos en programas de Komodo se realizan en tiempo de ejecución. Esto hace que la implementación de reglas de tipado débil sea más sencilla, con la consecuencia de que deben realizarse más chequeos en tiempo de ejecución.

4.5.4. Los tipos incorporados

Komodo viene con tipos incorporados que facilitan la creación de procedimientos básicos, y son herramientas que se esperan en cualquier lenguaje de programación de propósito general. Sin embargo, la elección de los tipos incorporados de Komodo refleja sus preferencias de uso.

- La tupla vacía

Está representada por `()`. Es en la práctica el tipo nulo de Komodo.

Es importante recalcar que la tupla vacía no es un tipo separado (como sucede con el tipo unitario en lenguajes como Haskell o Rust), sino que realmente el intérprete lo considera una tupla sin valores.

Esta característica hace a `()` más cercano a un tipo nulo, típico de los lenguajes de programación imperativos; que a un tipo unitario, típico de los lenguajes de programación funcionales.

La tupla vacía es un patrón que se puede rastrear:

```
let isNull(()) := true
let isNull(_) := false
```

Listado 27: *Pattern matching* con la tupla vacía.

También puede usarse el operador de igualdad:

```
let isNull(val) := val = ()
```

Listado 28: Comparación con la tupla vacía.

Este ejemplo muestra como realmente `()` es representado como una tupla:

```
let isTuple(_: Tuple) := true
let isTuple(_) := false

assert(().isTuple())
```

Listado 29: Tipo de la tupla vacía.

- Números

Komodo tiene tres representaciones para números: Enteros, flotantes y fracciones. Todos tienen tamaño arbitrario, que crece bajo demanda. El intérprete usa las librerías GMP y MPFR, que hacen parte del proyecto GNU y están diseñadas para funcionar juntas.

- Enteros

Los enteros tienen signo y tienen las operaciones de suma, resta, multiplicación, división, residuo, exponenciación y desplazamiento de bits, tanto a la izquierda como a la derecha. Los bits más significativos están a la izquierda. Son representados en tiempo de ejecución como arreglos dinámicos de enteros de longitud de la palabra de máquina de ejecución. El signo va por separado.

La generación de enteros de Komodo requiere, en general, de solicitar memoria en tiempo de ejecución. Este es un proceso costoso en términos de tiempo.

Se pueden escribir constantes en base 2, 8, 10 y 16. No hay diferencia entre dos enteros que representan la misma magnitud, independientemente de la base en que fueron escritos.

```
let eights := {
    0b1000,
    0o10,
    8,
    0x8,
}

assert(eights = {8})
```

Listado 30: Enteros de Komodo.

La implementación de los enteros es traída de la librería GMP. [6]

- Números de punto flotante

Los números de punto flotante de Komodo son una extensión de los descritos por el estándar IEEE 754, con las siguientes diferencias:

- El tamaño de la mantisa puede ser mayor que 53 bits.
- El tamaño de la mantisa puede variar entre diferentes instancias de los números.
- El tamaño de la mantisa se decide en el momento que un número es instanciado.

Puesto que la representación de estos números es binaria, viene con las características típicas de los números de punto flotante de máquina. En particular, no todos los números decimales son representables por estos números. Este es un ejemplo común:

```
let x := 0.1
```

Listado 31: Números de punto flotante en Komodo.

En este caso, `x` tiene un redondeo muy cercano a 0.1, pero no es exactamente 0.1, por el hecho de que 0.1 no puede ser representado con un mantisa y un exponente binarios.

La generación de flotantes requiere de la solicitud de memoria en tiempo de ejecución.

La implementación de los números de punto flotante es traída de la librería MPFR [7], que es una extensión de la librería GMP.

- Fracciones

Las fracciones tienen signo y tienen las operaciones de suma, multiplicación, división y exponenciación. Son representados como un par de enteros de longitud arbitraria, por lo que se pueden realizar operaciones con números arbitrariamente grandes o pequeños.

La utilidad de las fracciones viene cuando es necesario hacer operaciones sin redondeos, con el costo de menor velocidad. Las fracciones pueden representar todos los números que los enteros y los flotantes pueden representar, y más.

Se escriben con dos barras inclinadas:

```
let a := 5
let b := 1 // 5

assert(a * b = 1)
```

Listado 32: Fracciones de Komodo.

La generación de fracciones también requiere de la solicitud de memoria en tiempo de ejecución.

La implementación de las fracciones es traída de la librería GMP.

- Funciones

Las funciones de Komodo pueden escribirse de dos formas:

- De forma anónima, como una lista de parámetros y un bloque de código:

```
(a, b) -> a + b - 5
```

Listado 33: Función anónima de Komodo.

  Estas funciones son expresiones, así que pueden ser puestas dentro de contenedores o ser guardadas como variables.

- con nombre, como una lista de parejas patrón-resultado:

```
let f(0, _) := 0
let f(_, 0) := 0
let f(a, b) := a + b - 5
```

Listado 34: Función nombrada de Komodo.

Las funciones nombradas son siempre inmutables, así que no pueden crearse con la palabra clave `var`. Ejecutar esta pieza de código retorna un error:

```
var f(x) := 2*x
```

Listado 35: Declaración ilegal de una función.

Todas las funciones de Komodo pueden ser pasadas como argumentos de otras funciones.

Los *scopes* de Komodo son creados de forma léxica, lo que significa que los nombres referenciados en la función son los que se obtienen en el contexto de la definición de la función, y no en el contexto de sus ejecuciones. Usemos como ejemplo el siguiente fragmento de código:

```
let a := 2
let func := () -> a

for i in 0..5 do
    let a := i
    assert(func() = 2)
```

Listado 36: *Scope* léxico en Komodo.

En este caso, a pesar de que la `func` es ejecutada en un *scope* donde el valor de `a` varía, siempre usa el valor que `a` tenía cuando fue definida. Esta regla limita la forma en que se puede interpretar una llamada a una función, lo que puede ser conveniente al analizarla.

- Caracteres y cadenas

Komodo, a diferencia de muchos lenguajes de *scripting*, tiene tipos separados para representar caracteres y cadenas. Esto puede ser útil a la hora de iterar sobre cadenas y de hacer tratamiento minucioso de cadenas. Los dos tipos operan juntos, y se realizan conversiones implícitas entre ellos cuando es conveniente.

- Caracteres

Los caracteres de Komodo son valores escalares de Unicode, por lo que pueden representar cualquier símbolo Unicode. Tienen una longitud fija de 32 bits.

Su sintaxis es muy similar a la de las cadenas, sólo que usa comillas simples:

```
let a := 'a'
```

Listado 37: Declaración de un caracter en Komodo.

Todos los caracteres son patrones que pueden ser rastreados, y también se puede restringir la entrada de una función por su tipo:

```
let isAnA('a' || 'A') := true
let isAnA(_) := false

let isChar(_: Char) := true
let isChar(_) := false

assert(isAnA('a'))
assert(isChar('b'))
```

Listado 38: *Patttern mathing* de caracteres.

Los caracteres pueden ser sumados entre si para sumar cadenas, y pueden ser sumados con cadenas para producir otras cadenas. También pueden ser multiplicados por un entero para concatenarse a si mismas varias veces:

```
assert('a'+'b'="ab") # Char + Char
assert('a'+"bc"="abc") # Char + String
assert('z'*3="zzz") # Char "multiplicado"
```

Listado 39: Operaciones con caracteres en Komodo.

- Cadenas

Las cadenas de Komodo están representadas como arreglos inmutables de bytes, que están codificados con UTF-8.

Se puede iterar de izquierda a derecha sobre las cadenas de Komodo de la misma forma que se hace con las listas. Este es un detalle importante y que puede ser confuso. El patrón `[first|tail]` (o patrón *cons*) es compatible con listas y cadenas. Veamos un ejemplo:

```
let length([] || "") := 0
let length([_|tail]) := 1 + len(tail)

assert(length([1, 2]) = length("ab"))
```

Listado 40: Patrón *cons* para listas y cadenas.

En este ejemplo, se muestra que para que la función `length` funcione para listas y cadenas, el patrón `[] || ""` debe usarse, y así tener en cuenta ambos casos. Sin embargo, el patrón `[_|tail]` funciona para cadenas y listas por igual. Esto hace que la compatibilidad con el patrón *cons* no garantice que el argumento pasado sea una lista. En efecto, podría ser una lista o una cadena de caracteres.

Además, nótese que una lista de caracteres es diferente a una cadena:

```
assert(['a', 'b'] /= "ab")
```

Listado 41: Diferencia entre cadenas y listas de caracteres.

La diferencia entre cadenas y listas de caracteres es una característica traída de otros lenguajes como un detalle de implementación, pero conflictúa con la preferencia de Komodo de entender a los datos con la menor cantidad de detalles de implementación posible.

- Contenedores

Los contenedores almacenan otros valores, incluyendo los de su mismo tipo. Todos los contenedores de Komodo permiten almacenar valores de diferente tipo en el mismo contenedor simultáneamente.

- Tuplas

Las tuplas son colecciones ordenadas de valores, que no crecen. Su propósito es juntar valores. Pueden escribirse como valores separados por comas, rodeados por paréntesis.

```
(5, "cinco", (a) -> a + 5)
```

Listado 42: Tuplas de Komodo.

Sin las tuplas, quedarían dos soluciones para tener valores compuestos:

- Usar un diccionario: Esta solución está bien, pero puede ser demasiado complicada para algunos problemas. Además, puede operarse con otros diccionarios, lo cual puede ser indeseable.
- Usar una lista: Es una solución muy similar, pero sigue estando el problema de que pueden ser operadas con otras listas, lo cual puede ser indeseable.

Estas dos soluciones usan tipos con un propósito muy claro, y estarían siendo usadas de manera ligeramente distinta. La mayor utilidad de las tuplas es declarar la intención de que los datos en ellas deberían estar juntos.

La tupla vacía, mencionada al principio de esta sección, es una tupla y no un tipo por separado. (véase Sección -)

- Listas

Las listas de Komodo son de longitud arbitraria.

Se puede acceder a sus elementos de tres formas:

- Con índices enteros indexados desde cero, usando la notación `list[index]`. Esto es útil para escribir procedimientos iterativos que involucran el orden en que se encuentran los elementos, y se accede a múltiples partes de la lista en un mismo paso:

```
let reverse(l: List) :=
    var res := l
    for i in 0..(len(l)/2) do
        res[i] := l[len(l)-i-1]
        res[len(l)-i-1] := l[i]

    res
```

Listado 43: Reverso de una lista en Komodo.

  El acceso por índice a un índice ilegal (negativo o, mayor o igual que la longitud de la lista) hace que el programa sea interrumpido con un error.

- Iterando sobre la lista de izquierda a derecha, con la notación `[first|tail]`. Esto funciona bien para la mayoría de casos de uso, y permite la escritura sencilla de procedimientos recursivos:

```
let max(a, b) := if a > b then a else b
let max([val]) := val
let max([first|tail]: List) := max(first, max(tail))
```

Listado 44: Máximo de una lista en Komodo

- Iterando sobre la lista con expresiones por comprensión o en ciclos:

```
let list := [1, 2, 1, 2]

let set := {val for val in list}
assert(set = {1, 2})

var acc := 0
for val in list do
    acc := acc + val
assert(acc = 6)
```

Listado 45: Iteración sobre listas.

El intérprete las almacena como arreglos dinámicos. Esta es una representación conveniente para minimizar la solicitud de memoria en tiempo de ejecución y para la velocidad del acceso por índice, pero no tanto para la creación de sublistas obtenidas de la lista donde se itera.

- Conjuntos

Los conjuntos de Komodo son de longitud arbitraria. Se puede iterar sobre ellos y verificar la pertenencia de elementos.

Están representados como árboles binarios de búsqueda.

Los conjuntos tienen su propia sintaxis, y pueden ser escritos por extensión o por comprensión:

```
let A := {1, 2, 4, 8, 16} # por extensión
let B := {2**k for k in 0..5} # por comprensión

assert(A = B)
```

Listado 46: Conjuntos de Komodo.

Se puede iterar sobre sus elementos de varias maneras:

- Usando la notación *cons* para conjuntos:

  ```
  let prod({}) := 1
  let prod({some|rest}) := some * prod(rest)
  ```

  Listado 47: Notación *cons* para conjuntos.

  Esta notación funciona de la misma forma que la notación *cons* de listas.

  La implementación actual garantiza que los elementos son recorridos en orden, pero esta característica podría cambiar.

- Usándolo como iterador en contenedores por comprensión y ciclos:

  ```
  let set := {1, 2, 2}

  var list := []
  for val in set do
      list := [val|list]
  assert(list = [1, 2] || list = [2, 1])

  let list := [val + 1 for val in set]
  assert(list = [2, 3] || list = [3, 2])
  ```

  Listado 48: Iteración sobre conjuntos.

Para verificar que un elemento pertenece a un conjunto, puede usarse el operador `in`:

```
let A := {1, 2}
assert(1 in A)
```

Listado 49: Pertenencia de conjuntos.

Los conjuntos también pueden verificar contenencia e igualdad entre ellos, y se tienen las operaciones de unión y diferencia:

```
let A := {1, 2}
let B := {2, 3}

assert(A + B = {1, 2, 3})
assert(A - B = {1})
assert(A - B < A) # contenencia estricta
assert(A <= A) # contenencia o igualdad
```

Listado 50: Operaciones entre conjuntos.

Los conjuntos pueden ser desestructurados y rastreados con patrones:

```
let {a, b} := {1, 2}
assert(a + b = 3)
```

Listado 51: Desestructuración de conjuntos.

La razón de que los conjuntos sean estructuras de primera clase es evitar que el usuario los implemente incidentalmente como parte de la implementación de ciertas rutinas. Esta situación es muy común en el tipo de problemas a los que Komodo apela.

- Diccionarios

Los diccionarios de Komodo son de longitud arbitraria. Son colecciones de parejas llave-valor, donde el tipo de ambos es arbitrario.

Los diccionarios deben ser inicializados con al menos un elemento, pues la expresión `{}` genera un conjunto:

```
let set := {} # conjunto
let dict := { () => () } # diccionario
```

Listado 52: Construcción de diccionarios.

Se puede acceder a sus elementos de dos formas:

- Notación usual: `dic[llave]` donde `dic` es un diccionario y `llave` es un valor arbitrario.

  Esta notación permite usar cualquier valor de Komodo como una llave. Por ejemplo, aquí usamos listas y conjuntos como llaves:

```
let dict := {
  [[1], [2]] => 3,
  {2, 3, 4} => 9,
}

assert(dict[[[1], [2]]] = 3)
assert(dict[{2, 3, 4}] = 9)
```

Listado 53: Diccionarios con llaves arbitrarias.

- Notación de objeto: `objeto.llave`, donde `objeto` es un diccionario y `llave` es interpretado como una cadena, que es buscada en el diccionario.

  Esto es equivalente a escribir `objeto["llave"]`. Aunque confusa, esta notación es una facilidad para usar los diccionarios de una forma muy particular: como si fueran estructuras.

  Este es un ejemplo:

```
var data := {
    "values" => [1, 2, 3],
    "length" => 3,
}

assert(data.length = 3)
assert(data.values = [1, 2, 3])

data.values := [val + 1 for val in data.values]
assert(data.values = [2, 3, 4])

assert(data.values = data["values"])
assert(data.length = data["length"])
```

Listado 54: Diccionarios como estructuras.

La búsqueda de una llave que no se encuentra en un diccionario interrumpe el programa con un error.

En la implementación actual, no se puede iterar sobre diccionarios. Sin embargo, si pueden ser buscados con patrones:

```
let dict := {
    [[1], [2]] => 3,
    {2, 3, 4} => 9,
}

let f({x => 9, ..}) := x
assert(f(dict) = {2, 3, 4})
```

Listado 55: Patrones con diccionarios.

Los diccionarios están representados como árboles binarios de búsqueda, igual que los conjuntos. Esto podría cambiar en el futuro.

4.6. El intérprete

El intérprete está escrito en el lenguaje de programación Rust. [8] El ecosistema de Rust, de manera similar a lenguajes como OCaml, [9] es favorable para construir herramientas para lenguajes de programación. El modelo de memoria de Rust no incluye manejo de memoria automático, sino un sistema que permite verificar reglas que garantizan seguridad de memoria en tiempo de compilación.

El intérprete de Komodo es distribuido como un binario enlazado estáticamente cuando es posible, y está organizado como un monolito.

Para la comunicación con el sistema operativo, se usa la librería estándar de Rust: Hasta ahora, no ha sido necesario interactuar con una interfaz más cercana.

4.7. Gestión de memoria

La memoria de un programa de Komodo es gestionada automáticamente. Además, no hay una noción de referencia (véase Sección 4.2.2.). Todo valor declarado con `let` debe ser constante, y todo valor

declarado con `var` puede cambiar de acuerdo a las restricciones de mutabilidad (véase Sección 4.2.5.). Estas son las invariantes que el usuario de Komodo puede asumir.

La implementación de estas reglas es arbitraria, y en el caso del intérprete de Komodo, está en desarrollo.

Para gestionar el uso interno de referencias y copias, el intérprete usa las siguientes reglas:

- Si un valor inmutable es asignado a otro valor inmutable, se asigna internamente una referencia en lugar de copiar.
- Si un valor inmutable es asignado a un valor mutable, se asigna una copia.
- Si un valor mutable es asignado a un valor cualquiera, se asigna una copia.
- Si un valor cualquiera es pasado como argumento a una función, se pasa una referencia inmutable, siguiendo la semántica de las funciones (véase Sección 4.2.5.).
- En cualquier otro caso, se pasa una copia.

Las reglas actuales no tienen en cuenta que debería suceder en situaciones que involucran contenedores, donde pueden aparecer referencias cíclicas, por ejemplo. El comportamiento actual en estos casos es crear copias.

El intérprete usa conteo de referencias para hacer recolección de basura automáticamente.

Se planea implementar una estrategia de *mark-and-sweep*, donde de manera periódica se recorre un grafo de referencias del programa. Las referencias alcanzadas durante el recorrido son conservadas, y las no alcanzadas son eliminadas. Luego las secciones de memoria que no tuvieron referencias alcanzadas son liberadas.

4.8. Conversiones implícitas de valores

Komodo realiza conversiones de valores sin intervención del usuario, pero con reglas simples. Estas conversiones en dos grupos de tipos.

4.8.1. Números

Cuando dos números de diferente tipo son las entradas de una operación aritmética, una de las entradas es convertida al tipo de la otra.

Los tipos numéricos son `Integer`, `Fraction` y `Float`. Cuando se realiza una operación permitida entre cualesquiera dos valores con estos tipos, y los tipos son diferentes, se realiza una conversión de acuerdo a las siguientes reglas, verificadas en orden:

- Si uno de los valores es de tipo `Float`, el otro es convertido a `Float`.
- Si uno de los valores es de tipo `Fraction`, el otro es convertido a `Fraction`.

Estas reglas abarcan todos los casos donde los operandos son de un tipo numérico distinto. Las reglas implícitamente implementan la noción de una torre numérica [10, p. 19], [11], donde los tipos numéricos respetan la siguiente jerarquía (en orden descendente):

- `Float`
- `Fraction`
- `Integer`

Aunque realmente el tipo `Fraction` es el que puede expresar más números de todos los tipos numéricos, operar con números de punto flotante es usualmente más esperado y menos sorprendente que con

fracciones. En efecto, así lo hacen las jerarquías en [10] y [11]. Esta es la razón de que `Float` esté al tope de la jerarquía.

4.8.2. Caracteres y cadenas

Cualquier concatenación que involucre un caracter tendrá como resultado una cadena. Es decir:

```
assert({"ab"} = {'a' + 'b', 'a' + "b", "a" + 'b'})
```

Listado 56: Conversión de caracteres.

Esta regla también aplica para la concatenación de un caracter consigo mismo, usando el operador *:

```
assert({"aaa"} = {'a'*3, 3*'a'})
```

Listado 57: Conversión de caracteres.

5. Aspectos periféricos

Komodo, como proyecto, incluye piezas adicionales al intérprete, así como procesos de trabajo para crearlas. Esta sección enumera estos elementos y los explica brevemente.

5.1. Software adicional

Hay software adicional al intérprete que lo asiste o extiende su alcance.

5.1.1. Editor web

Una compilación del interprete a *WebAssembly* o WASM [12] es usada para poder usar el intérprete en el editor web de Komodo. Es una versión sin la librería estándar y con una interfaz simulada del sistema operativo.

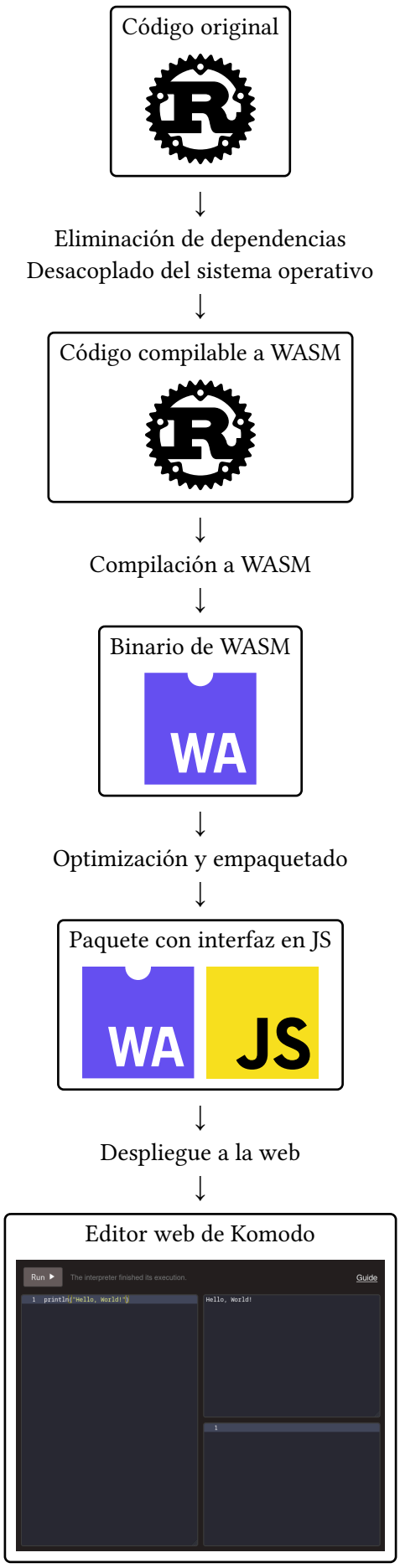
WASM es un objetivo de compilación sin una máquina de destino específica. Su enfoque es la ejecución de código en ambientes aislados e independientes de la máquina de ejecución. En este caso, el intérprete se compila a WASM para ejecutarlo en navegadores de Internet. Los navegadores más populares ya pueden ejecutar WASM.

La compilación del intérprete a *WebAssembly* se logra con el control de las dependencias de Komodo. En particular, se genera una versión del intérprete donde todas las dependencias pueden ser compiladas a WASM. Esto hace que la versión para el navegador sea ligeramente distinta a la versión nativa. Estas son las principales diferencias:

- La importación de modulos no está implementada. Ni los módulos de la librería estándar, ni la importación de archivos con código hacen parte de la versión para navegadores.
- Las funciones incorporadas que usan la entrada y salida estándar usan una interfaz simulada, que en realidad interactúa con la interfaz de usuario en el navegador.
- No hay un REPL, a diferencia de la versión nativa.

Esta versión modificada del intérprete es compilada a WASM. El binario obtenido es optimizado y luego empaquetado con una interfaz hecha en *JavaScript*, con la que el editor web de Komodo interactúa para ejecutar código escrito por el usuario. Así, se evita tener que enviar el código a un servidor, ejecutarlo allí, y devolver los resultados. La ejecución de los programas en el editor web ocurre del todo en la máquina del usuario.

Esta gráfica explica el paso desde el intérprete original a la versión para navegadores:


Código original
Eliminación de dependencias
Desacoplado del sistema operativo
Código compilable a WASM
Compilación a WASM
Binario de WASM
WA
Optimización y empaquetado
Paquete con interfaz en JS
WA
JS
Despliegue a la web
Editor web de Komodo
Run
The interpreter finished its execution.
Guide
1 println("Hello, World!")
Hello, World!
1

5.1.2. Resaltado de sintaxis

Se escribió una gramática de *TextMate* [13] para los *tokens* de Komodo, y así obtener resaltado de sintaxis en los editores de texto compatibles. Se distribuye una extensión para los editores VSCode y VSCodium, que añaden resaltado de sintaxis para Komodo a los archivos con extensión `.komodo`.

5.1.3. Instaladores

Se distribuye un script para instalar Komodo en distribuciones GNU/Linux, con máquinas con arquitectura AMD64. El instalador añade al sistema un binario enlazado estáticamente y los archivos de la librería estándar. Puede usarse ejecutando el siguiente comando, que instala la versión de Komodo más reciente:

```
curl --proto '=https' --tlsv1.2 -sSf https://komodo-lang.org/install.sh | sh
```

Se planea distribuir binarios e instaladores para MacOS y Windows, así como para más arquitecturas, en particular ARM64.

5.2. Guía de uso

Está publicada una guía de uso de Komodo en `https://komodo-lang.org/book`. El propósito del material es asistir a cualquier persona interesada en Komodo en el aprendizaje del lenguaje y en el uso del ecosistema. La guía está en constante cambio de acuerdo a como el lenguaje cambia. Se desea que la guía sea el recurso por defecto para aprender a usar Komodo.

6. Gramática de Komodo

6.1. Lista de *tokens*

Esta es una lista de los *tokens* que el analizador léxico emite, y las reglas que hacen que sean emitidos. Se muestran expresiones regulares para algunos *tokens* con el propósito de ilustrar las reglas rápidamente, pero la implementación del *lexer* no usa expresiones regulares. Los tokens que están relacionados a los bloques de indentación indentados son casos especiales, cuyo funcionamiento se describe con más detalle en la sección sobre análisis léxico (véase Sección 3.1.).

Las expresiones regulares están escritas con el estilo de Perl. [14]

Nombre	Descripción	Expresión regular
`Ampersand`	Sígno et: `&`	`&`
`Arrow`	Flecha simple: `->`	`->`
`As`	Palabra clave: `as`	`as`
`Assign`	Símbolo de asignación: `:=`	`:=`
`Bang`	Símbolo de exclamación: `!`	`!`
`Case`	Palabra clave: `case`	`case`
`Char`	Caracter Unicode	`'.'|'\\.'`
`Colon`	Dos puntos: `:`	`:`
`Comma`	Coma: `,`	`,`
`Dedent`	El final de un bloque indentado.	
`Do`	Palabra clave: `do`	`do`
`Dot`	Punto: `.`	`\.`
`DotDot`	Punto tras punto: `..`	`\.\.`
`Else`	Palabra clave: `else`	`else`

`Equals`	Símbolo de igualdad: `=`	`=`
`False`	Palabra clave: `false`	`false`
`FatArrow`	Flecha gruesa: `=>`	`=>`
`For`	Palabra clave: `for`	`for`
`From`	Palabra clave: `from`	`from`
`Greater`	Símbolo de *mayor que*: `>`	`>`
`GreaterEqual`	Símbolo de *mayor o igual que*: `>=`	`>=`
`Ident`	Un identificador.	`\p{Alphabetic}` `[\p{Alphabetic}\p{GC=Number}]`
`If`	Palabra clave: `if`	`if`
`Import`	Palabra clave: `import`	`import`
`In`	Palabra clave: `in`	`in`
`Indent`	El inicio de un bloque indentado.	
`Integer`	Un entero en base 2, 8, 10 o 16.	`(0)|(0(b|B)[0-1]+)|(0(o|O)[0-7]+)|` `([1-9][0-9]*)|(0(x|X)[0-9a-fA-F]+)`
`Lbrace`	Corchete izquierdo: `{`	`{`
`Lbrack`	Paréntesis cuadrado izquierdo: `[`	`\[`
`LeftShift`	Dos *menor que* juntos: `<<`	`<<`
`Less`	Símbolo de *menor que*: `<`	`<`
`LessEqual`	Símbolo de *menor o igual que*: `<=`	`<=`
`Let`	Palabra clave: `let`	`let`
`LogicAnd`	Dos sígnos et juntos: `&&`	`&&`
`LogicOr`	Dos barras verticales juntas: `||`	`||`
`Lparen`	Paréntesis izquierdo: `(`	`\(`
`Memoize`	Palabra clave: `memoize`	`memoize`
`Minus`	Guión: `-`	`-`
`Newline`	Salto de línea.	
`NotEqual`	Un *slash* y un símbolo de igualdad: `/=`	`\/=`
`Percent`	Símbolo de porcentaje: `%`	`%`
`Plus`	Símbolo de suma: `+`	`\+`
`Rbrace`	Corchete derecho: `}`	`}`
`Rbrack`	Paréntesis cuadrado derecho: `]`	`]`
`RightShift`	Dos *mayor que* juntos: `>>`	`>>`
`Rparen`	Paréntesis derecho: `)`	`)`
`Slash`	Una barra inclinada: `/`	`\/`
`SlashSlash`	Dos barras inclinadas: `//`	`\/\/`
`Star`	Un asterisco: `*`	`\*`
`StarStar`	Dos asteriscos: `**`	`\*\*`
`String`	Una cadena de caracteres.	`"[.\s]*"`
`Then`	Palabra clave: `then`	`then`
`Tilde`	Una virgulilla: `~`	`~`

`True`	Palabra clave: `true`	`true`
`Unknown`	Un caracter no reconocido.	
`Var`	Palabra clave: `var`	`var`
`VerticalBar`	Barra vertical: `|`	`|`
`Wildcard`	Barra baja: _	_

Tabla 1: Tokens del *lexer* de Komodo y sus reglas.

Hay algunas particularidades a mencionar:

1. El *lexer* ignora los segmentos que comienzan con un númeral `#` y terminan con un salto de línea. Estos son los comentarios de Komodo.
2. Los identificadores reciben toda la clase `Alphabetic` de Unicode en su primer caracter, y luego reciben caracteres de la clase `Alphabetic` o `Number`. [15]

   Estos nombres son propiedades de caracteres Unicode. [16]
3. Como muestra su expresión regular, los identificadores no incluyen barras bajas en ningún punto. Están exlusivamente compuestos de caracteres alfanuméricos.
4. Los ceros a la izquierda en enteros decimales no están permitidos. Un cero sólo va al principio de un *token* `Integer` cuando consiste en un solo cero, o cuando se va a escribir un prefijo para una base numérica no decimal (`0b`, `0o` o `0x`).

6.2. Reglas sintácticas

A continuación, se muestran las reglas sintácticas de Komodo. Se usan nombres en inglés para reutilizar los nombres usados en la lista de *tokens*, que se van a referenciar en esta gramática. Esto significa que si hay una regla no terminal mencionada en la gramática pero no está definida en la misma, entonces su nombre está en la lista de *tokens* y sus reglas son las mismas que las del *token* con el mismo nombre.

Los símbolos terminales se muestran entre comillas (como «+», por ejemplo) y los no terminales entre corchetes angulares (como `<Dict>`, por ejemplo).

La gramática ignora detalles como los espacios en blanco, cuyo procesamiento es responsabilidad del analizador léxico.

Esta gramática no incluye información sobre precedencias de operadores, pero esto está en la tabla de precedencias (véase Sección 6.2.1.).

La gramática mostrada es una referencia para describir la sintaxis de Komodo, pero el análizador sintáctico del intérprete no fue construido con una gramática en mente.

Nota: λ denota la cadena vacía.

```
  <Program>      ::= λ
                   | <Expression> <Program>
<Expression>     ::= <Bool>
                   | <Char>
                   | <Ident>
                   | <String>
                   | <Integer>
                   | <List>
                   | <Tuple>
                   | <Set>
                   | <Dict>
                   | <Call>
```

```
                      | <Lambda>
                      | <Prefix>
                      | <Infix>
                      | <Declaration>
                      | <Import>
                      | <Case>
                      | <If>
                      | <For>
                      | <Parenthesized>
        <Bool>        ≔ «true»
                      | «false»
        <List>        ≔ «[« <Expression> «for» <Pattern> «in» <Expression> «]»
                      | «[« <Expression> «|» <Expression> «]»
                      | «[« <Sequence> «]»
      <Pattern>       ≔ «_»
                      | <Bool>
                      | <Char>
                      | <Ident>
                      | <String>
                      | <Integer>
                      | <ListPattern>
                      | <TuplePattern>
                      | <SetPattern>
                      | <DictPattern>
                      | <InfixPattern>
    <ListPattern>     ≔ «[« <Pattern> «|» <Pattern> «]»
                      | «[« <SequencePattern> «]»
  <SequencePattern>   ≔ λ
                      | <Pattern>
                      | <Pattern> «,» <SequencePattern>
    <TuplePattern>    ≔ «(« <SequencePattern> «)»
     <SetPattern>     ≔ «{« <Pattern> «|» <Pattern> «}»
                      | «{« <SequencePattern> «}»
    <DictPattern>     ≔ «{« <DictSequencePattern> «}»
<DictSequencePattern> ≔ <Pattern> «=>» <Pattern>
                      | <Pattern> «=>» <Pattern> «,»
                      | <Pattern> «=>» <Pattern> «,» <DictSequencePattern>
    <InfixPattern>    ≔ <Pattern> «||» <Pattern>
                      | <Pattern> «:» <Signature>
      <Signature>     ≔ <Ident>
                      | <Ident> «||» <Signature>
      <Sequence>      ≔ λ
                      | <Expression>
                      | <Expression> «,» <Sequence>
        <Tuple>       ≔ «(« <TupleSequence> «)»
   <TupleSequence>    ≔ λ
                      | <Expression> «,» <Sequence>
         <Set>        ≔ «{« <Expression> «for» <Pattern> «in» <Expression> «}»
                      | «{« <Expression> «|» <Expression> «}»
                      | «{« <Sequence> «}»
        <Dict>        ≔ «{« <DictSequence> «}»
    <DictSequence>    ≔ <Expression> «=>» <Expression>
                      | <Expression> «=>» <Expression> «,»
                      | <Expression> «=>» <Expression> «,» <DictSequence>
        <Call>        ≔ <Expression> <Tuple>
       <Lambda>       ≔ <Tuple> «->» <Block>
                      | <Ident> «->» <Block>
        <Block>       ≔ <Expression>
                      | <Indent> <BlockSequence> <Dedent>
   <BlockSequence>    ≔ <Block>
                      | <Block> <BlockSequence>
```

```
                    | <Expression>
                    | <Expression> <Newline>
                    | <Expression> <Newline> <BlockSequence>
      <Prefix>      ::= <PrefixOperator> <Expression>
<PrefixOperator>    ::= «~»
                    | «!»
                    | «-»
      <Infix>       ::= <Expression> <InfixOperator> <Expression>
<InfixOperator>     ::= «in»
                    | «..»
                    | «||»
                    | «&&»
                    | «>»
                    | «>=»
                    | «<»
                    | «<=»
                    | «/=»
                    | «=»
                    | «^»
                    | «&»
                    | «<<»
                    | «>>»
                    | «-»
                    | «+»
                    | «/»
                    | «%»
                    | «*»
                    | «**»
                    | «:=»
  <Declaration>     ::= «let» <Pattern> «:=» <Block>
                    | «let» <CallPattern> «:=» <Block>
                    | «let» «memoize» <CallPattern> «:=» <Block>
                    | «let» <Pattern> «:» <Ident>
                    | «var» <Pattern> «:= <Block>»
      <Import>      ::= «import» <ImportSource>
                    | «import» <ImportSource> «as» <Ident>
                    | «from» <ImportSource> «import» <ImportTarget>
                    | «from» <ImportSource> «import» <ImportTargetTuple>
  <ImportTarget>    ::= <Ident>
                    | <String>
<ImportTargetTuple> ::= «(« <ImportTargetSequence> «)»
<ImportTargetSequence> ::= <ImportTarget>
                    | <ImportTarget> «,» <ImportTargetSequence>
      <Case>        ::= «case» <Expression> «do» <Indent> <CaseBlock> <Dedent>
   <CaseBlock>      ::= <Pattern> «=>» <Expression>
                    | <Pattern> «=>» <Expression> <Newline>
                    | <Pattern> «=>» <Expression> <Newline> <CaseBlock>
       <If>         ::= «if» <Expression> «then» <Block> «else» <Block>
      <For>         ::= «for» <Pattern> «in» <Expression> «do» <Block>
 <Parenthesized>    ::= «(« <Expression> «)»
```

6.2.1. Tabla de precedencias

Esta lista tiene todos los operadores infijos de Komodo, con su respectiva precedencia. Un operador con cierta precedencia va a ser agrupado antes que otro operador con menor precedencia.

Operador	Precedencia
`:=`	1
`in`	2

`..`	3
`||`	4
`&&`	5
`>`	6
`>=`	6
`<`	6
`<=`	6
`/=`	6
`=`	6
`^`	7
`&`	8
`<<`	9
`>>`	9
`-`	10
`+`	10
`/`	11
`%`	11
`*`	11
`**`	12

Tabla 2: Tabla de precedencias de Komodo.

Referencias

[1] J. Borwein y D. Bailey, *Mathematics by Experiment, 2nd Edition: Plausible Reasoning in the 21st Century*. en Ak Peters Series. Taylor & Francis, 2004.

[2] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[3] «262: Ecmascript Language Specification - 6th Edition», *ECMA (European Association for Standardizing Information and Communication Systems), pub-ECMA: adr*, 2015.

[4] T. Norvell, «Parsing Expressions by Recursive Descent». [En línea]. Disponible en: https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm

[5] «ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary», *ISO/IEC/IEEE 24765:2010(E)*, n.º , pp. 1-418, 2010, doi: 10.1109/IEEESTD.2010.5733835.

[6] «The GNU MP Bignum Library». [En línea]. Disponible en: https://gmplib.org/

[7] «The GNU MPFR Library». [En línea]. Disponible en: https://www.mpfr.org/

[8] «Rust Programming Language». [En línea]. Disponible en: https://www.rust-lang.org/

[9] «Ocaml Programming Language». [En línea]. Disponible en: https://ocaml.org/

[10] N. I. Adams *et al.*, «Revised5 report on the algorithmic language scheme», *SIGPLAN Not.*, vol. 33, n.º 9, pp. 26-76, sep. 1998, doi: 10.1145/290229.290234.

[11] J. Yasskin, Ed., «PEP 3141 - A Type Hierarchy for Numbers». [En línea]. Disponible en: https://peps.python.org/pep-3141/

[12] A. Rossberg, Ed., «WebAssembly Specification». [En línea]. Disponible en: https://webassembly.github.io/spec/core/

[13] «Language Grammars - TextMate 1.x Manual». [En línea]. Disponible en: https://macromates.com/manual/en/language_grammars

[14] «perlre - Perl regular expressions». [En línea]. Disponible en: https://perldoc.perl.org/perlre

[15] K. Whistler, Ed., «Unicode Character Database». [En línea]. Disponible en: https://www.unicode.org/reports/tr44/

[16] A. F. Ken Whistler, Ed., «The Unicode Character Property Model». [En línea]. Disponible en: https://www.unicode.org/reports/tr23/