Diseño e implementación del lenguaje LTA y del compilador LTA a Ecmascript 2017

Teoría de lenguajes, autómatas y compiladores
2C 2017

Autores

Ramiro Olivera Fedi
rolivera@itba.edu.ar
56498

Julián Antonielli
jantonielli@itba.edu.ar
56650

Código

https://github.com/jjant/LTAlang

Índice

Introducción

El lenguaje LTA surge como consecuencia de dos principios que nos planteamos a la hora de diseñar y desarrollar el lenguaje:

- Diseñar un lenguaje introductorio, sencillo pero poderoso.

  Hoy en día los lenguajes de programación disponible pueden separarse en dos grandes categorías: Aquellos lenguajes introductorios, que permiten al desarrollador ingresar al mundo de la programación de una manera lenta y reducida (*Logos* o *Scratch*); Y aquellos lenguajes todo-poderosos, que permiten a grandes rasgos desarrollar tanto un servidor como un *hola mundo* (*C* o *Java*). En nuestra experiencia estas dos categorías no se suelen solapar, obligando a los desarrolladores principiantes a introducirse en un lenguaje de poca utilidad, o a sufrir ingresando al mundo de los lenguajes productivos.

- Diseñar un lenguaje versátil y dinámico.

  Consideramos que la creatividad y apertura de los desarrolladores es uno de sus activos más importantes. Creemos entonces que el tamaño de las ideas no deben restringirse al lenguaje que el desarrollador utilice, sino que al contrario, el lenguaje debe adaptarse a la creatividad y el carácter difuso de las ideas.

Al desarrollar LTA buscamos desarrollar un lenguaje que permitiese a nuevos desarrolladores ingresar al mundo de la programación, sin necesidad de grandes trabas, logrando una curva de aprendizaje suave y llevadera, pero que les permita al mismo tiempo construir aplicaciones y programas poderosos. Al mismo tiempo, nos enfocamos en quitar las restricciones que suelen interponerse en el camino de la expresión de las ideas (Como el tipado de las variables).

Por eso nos gusta decir que LTA es un lenguaje dinámico y versátil, pensado tanto para expertos desarrolladores, como para jóvenes principiantes.

En vista a estos puntos, elegimos desarrollar un compilador LTA a Ecmascript 2017 (o Javascript, utilizamos ambos términos de manera intercambiable.) ya que es un lenguaje con una gran comunidad, un gran mercado profesional y alto potencial desarrollo para sus desarrolladores, que además comparte el principio de favorecer la versatilidad y el dinamismo.

Consideraciones

Hay varios puntos a considerar sobre el desarrollo de nuestro compilador.

Por un lado se utilizaron varias dependencias:

- **prompt-sync**[4]: Una librería que permite leer desde *stdin* en *nodejs* de manera bloqueante.
- **prepack**[3]**:** Un compilador de javascript a javascript, con foco en la optimización de funciones y operaciones.
- **npm**[5]**:** Es el administrador de paquetes de *nodejs* que nos permiten agregar las dependencias previamente mencionadas.

Más tarde se explicará que finalmente se tomó la decisión de no utilizar *prepack* como optimizador del código debido a su inestabilidad y generación de distintos errores.

Por otro lado, para lograr determinadas funcionalidades, nos apoyamos en las mismas funcionalidades de Javascript. Por ejemplo, nuestra librería nativa *Math* es tan solo un wrapper para la librería homónima de Javascript; La palabra reservada auto-referenciante *this* es un wrapper para el *arguments.callee* de Javascript; Las funciones asincrónicas obtenidas con el operador |> se basan en el ciclo de eventos de javascript, y en su función native *setTimeout.*

Desarrollo del trabajo práctico

El trabajo se dividió en 3 partes:

- **Diseño y desarrollo de la gramática**

  En esta parte nos enfocamos en diseñar la gramática a partir de los principios que establecimos en la introducción. Utilizamos la sintaxis de YACC para generar las reglas necesarias. Nos basamos en gran parte en la gramática para el lenguaje C[1], con el objetivo de hacerlo suficientemente parecido a este para que a los desarrolladores de lenguajes imperativos no les sea tan grande el cambio, pero aún más importante, para que los nuevos desarrolladores que comienzan con LTA puedan pasar a un lenguaje de bajo nivel como C sin mayores dificultades.

  La gramática se encuentra en el archivo **grammar.y**.

- **Diseño y desarrollo del AST**

  Utilizando YACC se construyó regla a regla el AST de nuestro lenguaje. Utilizamos una implementación de un árbol con listas encadenadas, en lugar de arrays. Así por ejemplo, la raíz del árbol es en realidad un puntero al nodo de la primera instrucción del programa, que a su vez apunta a la siguiente instrucción y así sucesivamente. Cada instrucción contiene a su vez un *tipo*, y un puntero a sus hijos.

  Se definieron 25 tipos de nodos, representando distintos tipos de operaciones y definiciones de nuestro lenguaje. Los nodos del árbol y el código puede encontrarse en el archivo **structures.c** y en el archivo **nodes.h**.

- **Diseño y desarrollo del generador de Javascript**

  Para la generación de código a partir del AST utilizamos un patrón muy interesante, que permitirá que la extensión del lenguaje sea fácil y mantenible. Como se explicó antes, cada nodo guarda su *tipo*.

  Diseñamos un array con punteros a función, indexable por el *tipo* de los nodos. De esta manera desarrollamos una función que genera el código para cada *tipo* de nodo, permitiendo así que las funciones cumplan con el principio de única responsabilidad.

  Así logramos reducir la cantidad de líneas de código, reducir la complejidad de la generación del lenguaje, y aumentar la mantenibilidad del proyecto.

  El código puede encontrarse en el archivo **main.c**

Descripción de la gramática

Primero enumeramos las *features* del lenguaje, y más tarde realizamos una pequeña introducción a la sintaxis del mismo. En el repositorio de código puede encontrarse la carpeta **programs** que cuenta con varios códigos de ejemplo comentados mostrando funcionalidades y ejemplos en mayor profundidad.

Features del lenguaje

- Operadores como en C
- Tipado débil (Pero con primitivos tipados: *Number*, *String*, *Boolean*, *Array* y *Function*)
- Constantes (Por ejemplo: 1.232, -4, *true*, "E incluso cadenas")
- Ciclos de repetición (*loop*)
- Condicionales (*if* y operador ternario)
- Lamdas (Funciones anónimas) con |*x*| => *{ ... }*
- Funciones como ciudadanos de primera clase (Puede definirse una función on-the-fly, pasandosela por parámetro a otra)
- Funciones sincrónicas y asincrónicas con los operadores => y |>
- Manejo de *input* y *output* con las funciones nativas *read* y *puts* respectivamente
- Hashes nativos (Pares clave-valor) con | *clave : valor* |
- Comentarios de línea simple con (:
- Librería de utilidades matemáticas bajo el nombre *Math*
- Autorreferencia en funciones con la palabra reservada *this*
- ¡No hay puntos y comas!

Como fuertes diferencias con el lenguaje C, por fuera de la sintaxis y las extensiones:

- No existen *funciones* en el sentido de C
- No existe la función *main*
- No pueden tiparse las variables
- Se abstrae el manejo de punteros

Introducción al lenguaje LTA

Un programa en LTA comienza a ejecutarse desde la primera línea del archivo. Como ejemplo de un programa válido para el lenguaje LTA puede usarse el siguiente, que dado un número, dice si es o no primo:

```
prime = |n| => {
  if (n % 1 || n < 2) {
    return (false)
  }

  if (n % 2 == 0) {
    return (n==2)
  }
```

```
  m = Math.sqrt(n);

  i = 3;

  loop(i<=m) {
    if (n % i == 0) {
      return(false)
    }
    i += 2
  }

  return(true)
}
```

puts(prime(2)) (: TRUE

El programa empieza entonces en la primera línea, asignando a la variable *prime* una función anónima sincrónica, que recibe un parámetro n.

Dentro de la función, pueden verse diversas estructuras, como el conocido *if,* que no permite utilizarse en su sabor de única línea, sino que requiere que se utilicen siempre las llaves, obligando de esta forma al desarrollador novato a tomar noción de la importancia de los detalles en los lenguajes de programación

Las funciones, al igual que C, retornan un valor a través de *return(...)*. En LTA, *return* se comporta como una función, por lo que se deben utilizar paréntesis.

La librería *Math*, es una librería nativa que le permite utilizar a los usuarios varias funciones matemáticas. Su api, es igual a la api del objeto *Math* de javascript.

La estructura *loop* funciona de manera idéntica al *while* de C, con la misma restricción que los *if* sobre el uso de las llaves.

La función *puts* nativa del lenguaje, permite al desarrollador mostrar al usuario valores por la salida estándar.

Finalmente, puede verse como las llamadas a funciones se comportan igual que en C.

Funciones como ciudadanos de primera clase

Una de las funcionalidades que más nos gusta del lenguaje LTA que representa perfectamente el principio de versatilidad es la utilización de funciones como ciudadanos de primera clase. Esto significa, que ¡Las funciones son valores, como los números y las cadenas! De esta manera, pueden lograrse programas como el siguiente:

```
applyToString = |a, b| => {
  return(puts(b(a)))
}

applyToString("Hi frIenD", |str| => {
  return(str.lower());
})
```

La función *applyToString* recibe 2 parámetros. Pero lo más interesante, es que al momento de llamarla, puede definirse una nueva función, sin nombre.

Asincronismo

Otra de las funciones más que interesantes, es la de funciones asincrónicas. Para explicar mejor esta funcionalidad, puede utilizarse el siguiente ejemplo:

```
sync = ~ => {
  read("Input: ")
}

async = ~ |> {
  read("Input: ")
}

async()
puts("Hi :)")
sync()
puts("Hi :)")
```

En primer lugar, notar que el símbolo ~ representa el conjunto de parámetros vacíos. El mismo símbolo se utiliza para definir un *hash nativo* vacío.

Volviendo al asincronismo, las funciones definidas bajo el operador |>, no se ejecutan en el orden impuesto en el código, sino que se ejecutan un tiempo corto después, utilizando el stack de funciones asincrónicas provisto por Ecmascript. Por ejemplo, en este ejemplo particular la salida es la siguiente:

```
> Hi :)
< Input: …
> Input: …
> Hi :)
```

Nótese que con el caso de las funciones sincrónicas, la operación bloqueante de leer input del usuario evita que se ejecute la siguiente instrucción, mostrando el mensaje luego de haberse ingresado el texto. Por otro lado, con la función asincrónica, primero se muestra el

texto, y más tarde se le pide el ingreso de texto al usuario.

Más ejemplos

Para ver más ejemplos comentados, puede accederse en el repositorio a la carpeta **programs**, que cuenta con varios ejemplos comentados para ayudar a iniciarse en el lenguaje LTA.

Dificultades

La primer dificultad se trató en decidir claramente nuestro lenguaje y el lenguaje destino del compilador. Ya se explicó en la introducción la decisión sobre la gramática de LTA y sobre la elección de Ecmascript. No obstante, este segundo punto trae tanto ventajas como beneficios.

Por un lado, el código resultante del proceso de compilación es código ES2017 válido. No obstante, debido a la modernidad de las *features* del lenguaje utilizadas, solo puede utilizarse en las consolas de javascript de navegadores modernos, versiones modernas de *node*, o a través de un proceso de compilación a una versión más antigua mediante programas como *babel*.

Por otro lado, se había tomado la decisión de utilizar un optimizador de código. No obstante, esto generaba varios problemas: Por un lado, se utiliza una dependencia externa, prepack[3]. Pero aún más grave, es que al ser un optimizador muy moderno, su soporte es limitado, ocasionando que sea una feature inestable, y entre otras cosas, generando problemas a la hora de compilar algunos programas.

Finalmente, como se explicó en el apartado de consideraciones, se utilizaron muchas funcionalidades del lenguaje objetivos, ocasionando que la creación de un compilador a un lenguaje con mayores restricciones sea un proceso bastante engorroso.

Futuras extensiones

Se nos presentaron muchísimas funcionalidades que descartamos en pos de una primera implementación de un lenguaje más accesible. Entre ellas, algunas de las que más nos interesaban fueron:

- **Clases, herencia e interfaces**

  La orientación a objetos es probablemente el paradigma con mayor uso en el desarrollo de software, ya que permite generar código mantenible y claro, solamente suscribiéndose a unos pocos principios. Creemos que LTA podría beneficiarse de sintaxis y comportamiento nativo de objetos.

- **Mejora en la versatilidad de la gramática**

  Hay varios detalles que podrían beneficiarse de una repasada. Entre ellos, por ejemplo, la estructura de *} else {...}* obliga al usuario a escribir la palabra reservada *else* justo a la derecha del cierre de llaves. Esto es resultado de distintas optimizaciones en la gramática, en pos de conseguir quitar los molestos *;* del lenguaje C. No obstante, creemos que LTA, siguiendo a sus principios, debería permitir mayor flexibilidad a la hora de escribir código.

- **Compresión de listas à la python**

  Hay varias funcionalidades de otros lenguajes más que interesantes que se podrían implementar en LTA. De todas ellas, una de las más llamativas y poderosas, es *list comprehension* del lenguaje *python*. Esto permite no solo optimizar el código al compilador, sino reducir la cantidad de líneas de código en pos de uno mucho más claro y limpio. De esta manera, por ejemplo, para generar un array de números pares podría escribirse *[x loop x < 100 && if x % 2 == 0]*

Como se explicó en la sección de *Desarrollo del trabajo práctico,* se implementó un patrón que permite que la adición de funcionalidades sea bastante sencilla. La regla gramatical, el nodo con su tipo, y la función que *handlee* el nodo es suficiente para que le compilador siga funcionando con una funcionalidad extra.

Referencias

1. BNF para el lenguaje ANSI C https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm
2. http://qriollo.github.io/
3. https://prepack.io/
4. https://github.com/0x00A/prompt-sync
5. https://www.npmjs.com/