Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

# Un lenguaje de dominio específico para sistemas de condiciones.

Tesis de Licenciatura en Ciencias de la Computación

Lucas Bekier

Director: Ceria, Santiago

Codirector: de Caso, Guido

Buenos Aires, 2020

# UN LENGUAJE DE DOMINIO ESPECÍFICO PARA SISTEMAS DE CONDICIONES.

Así como a veces existen lenguajes de programación de propósito general para programadores, otras veces son necesarios lenguajes específicos para usuarios finales que son una especie de versión simplificada, generalmente para aplicar en dominios específicos.
En el marco de este trabajo se propone un lenguaje de dominio específico (DSL) que permita a usuarios no expertos en la programación poder escribir reglas condicionales de manera sencilla. Existen muchos contextos donde es necesario escribirlas: un ejemplo muy común es la necesidad de mostrarle a usuarios diferente información basada en reglas predefinidas, ya sea de autorización o en base a acciones que desencadenan diferentes flujos de datos.
Asimismo, se construye una herramienta que traduce código escrito en JavaScript (un lenguaje de propósito general) que tenga una estructura dada, al DSL anteriormente propuesto y se mide la eficacia de la herramienta en base a un corpus de programas preexistentes.
Por último se analiza la correctitud de la implementación del traductor a través de técnicas de *fuzzing*.

**Palabras claves:** Lenguaje de dominio específico (DSL), *parser*, *Abstract Syntax Tree* (AST), traducción, *branches*, *static single assignment*, *fuzzing*.

*A mi familia, que siempre está en todas.*

# Índice general


**1.. Introducción** . . . . . . . . 1
- 1.1. Lenguajes de Dominio Específico (DSL) . . . . . . . . 1
  - 1.1.1. Cuándo usar lenguajes de dominio específico . . . . . . . . 1
- 1.2. Nuestro problema . . . . . . . . 2
- 1.3. Propuesta . . . . . . . . 3
- 1.4. Estructura del trabajo . . . . . . . . 4

**2.. Análisis de casos de uso reales** . . . . . . . . 5
- 2.1. Haciendo del corpus de programas uno con un tamaño tratable . . . . . . . . 5
- 2.2. Análisis de los programas . . . . . . . . 6
  - 2.2.1. Características frecuentes . . . . . . . . 6

**3.. Definición del lenguaje** . . . . . . . . 9
- 3.1. Tipos . . . . . . . . 9
- 3.2. Términos . . . . . . . . 9
- 3.3. Axiomas de tipado . . . . . . . . 10
- 3.4. Semántica . . . . . . . . 12

**4.. Implementación** . . . . . . . . 19
- 4.1. Lenguaje . . . . . . . . 19
  - 4.1.1. Abstract Sintax Tree . . . . . . . . 19
  - 4.1.2. Parser . . . . . . . . 20
  - 4.1.3. Tecnologías utilizadas . . . . . . . . 22
  - 4.1.4. Ejecución del lenguaje . . . . . . . . 23
- 4.2. Anonimizador + Estandarizador de código JavaScript . . . . . . . . 25
- 4.3. Clasificador de programas . . . . . . . . 25

**5.. Traducción** . . . . . . . . 27
- 5.1. Branching . . . . . . . . 27
- 5.2. Definición de variables . . . . . . . . 29
- 5.3. Operadores . . . . . . . . 30

**6.. Resultados** . . . . . . . . 33
- 6.1. Corpus de datos e invariantes . . . . . . . . 33
- 6.2. En la práctica . . . . . . . . 34
- 6.3. Resultados . . . . . . . . 36
- 6.4. Correctitud . . . . . . . . 36
  - 6.4.1. Correctitud de nuestra implementación . . . . . . . . 37

**7.. Conclusiones** . . . . . . . . 43
- 7.1. Trabajo relacionado . . . . . . . . 43
  - 7.1.1. Equivalencia entre dos códigos diferentes . . . . . . . . 43
  - 7.1.2. Verificación de compilación de código de un lenguaje a otro . . . . . . . . 43

7.2. Conclusiones . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
7.2.1. Trabajo futuro . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44

# 1. INTRODUCCIÓN

## 1.1. Lenguajes de Dominio Específico (DSL)

En el campo del desarrollo de software muchas veces nos encontramos en una situación donde es necesario el uso de un lenguaje de especificación, modelado o programación que nos permita resolver un problema con características particulares. Dicho lenguaje puede tener distintos tipos de usuarios finales. Por ejemplo, pueden estar hechos para los programadores para resolver problemas particulares, o pueden ser parte de un sistema para un usuario final, que no necesariamente sea técnico.

Es así que nace el concepto de DSL (*domain specific language* o lenguaje de dominio específico) para poder proveer al usuario de nuestros sistemas un lenguaje sencillo y que represente conceptos que él mismo conozca. A diferencia de los lenguajes multipropósito, estos son lenguajes acotados que sólo aplican a dominios particulares.

Son ejemplos típicos:

- **SQL**, un lenguaje utilizado para resolver consultas en bases de datos.
- **HTML**, un lenguaje de modelado para estructurar como se ve una página web.
- **Gherkin**, un lenguaje utilizado para construir casos de prueba de una aplicación basado en lenguaje natural para especificar comportamiento sin importar cómo la aplicación esté implementada.

### 1.1.1. Cuándo usar lenguajes de dominio específico

Dado que existen casos de éxito donde este tipo de lenguajes son ampliamente usados, es conveniente pensar si un DSL propio será útil al momento de construir una pieza de software que requiera que el usuario deba escribir en un lenguaje parseable.

Pensando el problema en términos generales, ¿Qué ventajas tiene usar un DSL?

- Si el usuario que debe "programar" no es muy técnico, un DSL parece ser una mejor alternativa que un GPL (*general purpose language* o lenguaje de propósito general).
- Las expresiones generadas pueden ser hechas en el mismo "idioma" que el usuario conoce y no debe aprender un lenguaje más complejo.
- Es más fácil de aprender debido al alcance mucho más limitado.
- En caso que esté destinado para programadores, hace que la comunicación entre expertos en el dominio y programadores sea mucho más fluida.

Sin embargo, construir un lenguaje de uso específico lleva aparejada una gran cantidad de trabajo: definición de la sintaxis, la gramática, la semántica; la construcción de herramientas para poder entenderlo y ejecutarlo. También hay que tener en cuenta el mantenimiento del código y el costo del agregado de nuevas funcionalidades.

## 1.2. Nuestro problema

Un requerimiento muy común en una aplicación es la posibilidad de darle al usuario la posibilidad de definir reglas condicionales. Veamos algunos ejemplos:

- Un usuario de una red social puede definir si una publicación alcanza a todos sus contactos, a un determinado grupo o a todos exceptuando a alguien.
- Un usuario de una aplicación para construir encuestas puede decidir si determinada pregunta se realiza a todo el público o no.
- Un usuario de una aplicación que sirve para armar páginas web sencillas puede decidir si determinado contenido es accesible en todo momento o en determinadas situaciones.

Un caso particular que se usará como referencia para este trabajo es el de una herramienta de "Enterprise Customer Experience Management" que cuenta con una aplicación para construcción de encuestas y permite mostrar u ocultar partes de esas encuestas dependiendo de una condición escrita por el usuario. Al momento de construirse dicha herramienta, los requerimientos eran tales que se debía poder satisfacer cualquier pedido del cliente para poder condicionar los componentes de la encuesta.

Teniendo en cuenta estos requerimientos, construir únicamente un DSL no fue suficiente y se decidió permitir también escribir al usuario en el lenguaje de programación JavaScript un programa que devuelve un valor booleano. De esta manera, al ser éste un lenguaje de programación Turing Complete, se cubrió cualquier escenario que el usuario pudiera necesitar. Esta solución es un arma de doble filo, ya que da una gran flexibilidad al usuario pero trae a su vez muchos problemas:

- Necesidad de construcción de un entorno de ejecución cuidado (*sandbox*) para que el sistema no quede colgado si la ejecución del programa en JavaScript no termina.
- Posibilidad de agregar bugs al utilizar un lenguaje tan flexible.
- Gran complejidad para personas no técnicas para escribir condiciones.

Sin embargo, con el paso del tiempo, el usuario al que apunta la aplicación fue mutando. El desarrollo comenzó siendo para personas con cierto background técnico y hoy apunta a cualquier usuario. Es por esto que mantener el uso de un lenguaje Turing Complete como parte de las opciones es muy riesgoso.

Para entender de manera clara el caso de uso real veamos un ejemplo. En la siguiente figura se muestra un posible flujo en una encuesta para un usuario. Básicamente se desea mostrar una pregunta o terminar la misma según una respuesta anterior del usuario.

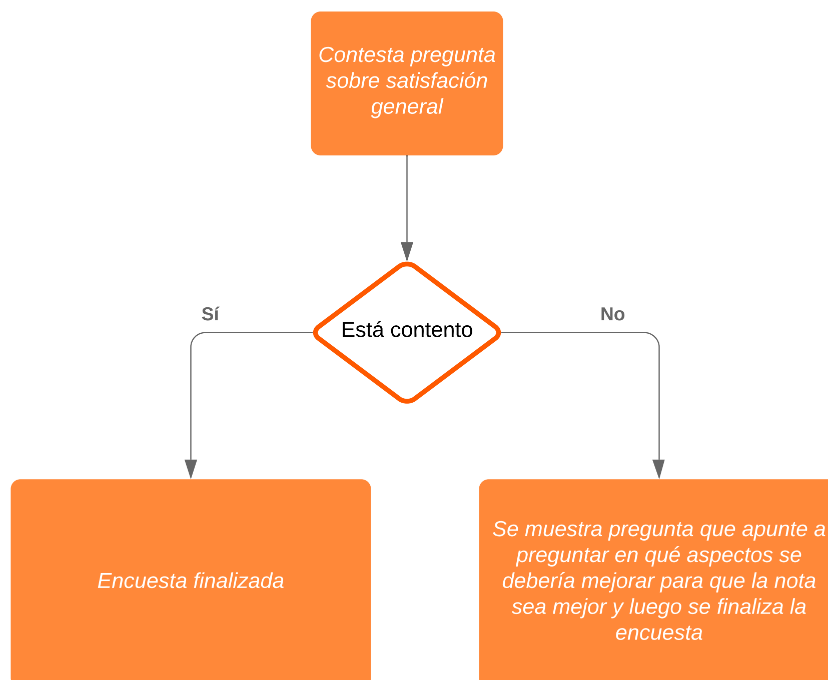


**Fig. 1.1:** Flujo de una posible encuesta

Básicamente para lograr dicho comportamiento se desea condicionar si se muestra o no la última pregunta basado en una respuesta anterior del usuario. Para lograrlo se debe escribir la siguiente condición asociada a la misma.

```
function estaContento() {
    let respuesta = context.get("satisfacción_general");
    // La pregunta ha sido contestada y con un valor mayor a 7
    if (respuesta !== undefined && respuesta > 7)
        return false;
    return true;
};
```

## 1.3. Propuesta

Se propone la creación de un lenguaje de dominio específico que permita predicar sobre un conjunto de variables tipadas que sea de fácil entendimiento y que le agregue mayor valor a los tipos primitivos de datos.

Podrían ser ejemplos a tener en cuenta, predicar

- si una variable pertenece a un conjunto de valores posibles.
- si una variable de tipo string es un mail válido.
- si dado un conjunto de variables, al menos una tiene contenido asignado.

Dicho lenguaje aplica no solo a nuestro caso de uso, sino también podría ser aplicado en muchos otros casos al tener como precondición contar con un diccionario de variables-valor y como poscondición responder con un valor booleano.

Para llevar a cabo dicha tarea se plantea utilizar como base de datos de entendimiento los códigos JavaScript utilizados en el sistema anteriormente descripto para así identificar los patrones de uso más frecuentes y cubrir una gran variedad de casos.

Como siguiente paso se plantea la construcción de una herramienta que parsee código escrito en JavaScript (que reciban como input un diccionario de variables y devuelvan un valor booleano) y luego traduzca a nuestro DSL, preservando la semántica. Esto permitirá migrar las encuestas existentes para poder aprovechar así de las ventajas de los lenguajes de dominio específico como la entendibilidad y mantenibilidad y evitar los problemas de los GPLs ya mencionados.

Se analiza el uso de esta herramienta de traducción sobre un corpus basado en los casos reales que usamos de base de datos. Idealmente, se contará con la mayor cantidad de programas migrados a nuestro DSL. Para poder tener mayor seguridad sobre la correctitud de la implementación se plantea el uso de herramientas que generan casos de tests automáticamente para poder verificar que la implementación es precisa. Estos casos se utilizan como input del programa original y del escrito en el DSL y se espera que el resultado de ejecución de uno y otro coincidan.

## 1.4. Estructura del trabajo

La estructura del trabajo se basa en diferentes capítulos, los cuales se detallan a continuación:

- En el capítulo 2 se hace un análisis del estado del arte para poder usar de base de entendimiento en el resto del trabajo.
- En el capítulo 3 se define un lenguaje basado en los requerimientos encontrados en el capítulo anterior.
- En el capítulo 4 se hace referencia a la implementación de dicho lenguaje y todos los artefactos de código relacionados que fueron desarrollados.
- En el capítulo 5 se define la herramienta de traducción de códigos escritos en JavaScript al lenguaje definido previamente.
- En el capítulo 6 se hace un análisis de que tan bien funciona la herramienta en el corpus de datos que tenemos.
- En el capítulo 7 se termina con una discusión de los resultados obtenidos y trabajos relacionados.

# 2. ANÁLISIS DE CASOS DE USO REALES

## 2.1. Haciendo del corpus de programas uno con un tamaño tratable

Para poder construir un DSL que no sea muy complejo pero que pueda ser de utilidad se realizó un análisis de un corpus de programas de tamaño considerable (unos miles). Es éste un corpus muy grande para observar, con lo cual se decidió hacer un análisis de los datos existentes en busca de programas duplicados para minimizar dicho trabajo.

Como primera opción, se planteó hashear los archivos y ver cuantos diferían. Sin embargo ésta no es una buena idea debido a que dos programas se considerarían diferentes si uno tiene un espacio de diferencia, o bien si dos variables fueron nombradas de manera diferente.

Es necesario entonces buscar una mejor estrategia para obtener mejores resultados.

Para resolver el problema de nombres definiremos el concepto de *programas equivalentes bajo etiquetas*. Dos programas son considerados *equivalentes bajo etiquetas* si tras reemplazar todos los nombres de variables, literales (textos, números, expresiones regulares) por un identificador genérico sin usar (determinado por un estado global), ambos programas resultan ser sintácticamente iguales.

Para resolver el problema de estilos/espaciado al escribir programas, es posible estandarizar los programas a partir de parsearlos y traducirlos con un formato determinado. Esto significa que el corpus se deja de considerar sólo como un conjunto entradas de texto para pasar a ser considerados como programas escritos en JavaScript.

En base a las ideas planteadas, se decide construir el siguiente algoritmo:

1. Parsear la entrada y obtener el AST asociado (*Abstract Syntax Tree*).
2. Recorrer el AST en busca de variables, literales (textos, números, expresiones regulares).
3. En caso de encontrar uno de los elementos citados más arriba, renombrarlos por el siguiente identificador sin usar (determinado por un estado global).
4. Traducir el programa bajo un formato determinado.

```
function f() {var a = context.get('a'); return a >= 234;}
```

Código 1

```
function esMayor() {
    var edad = context.get('age');
    return edad >= 18;
}
```

Código 2

```
function esMayor() {
    var edad = context.get('age');
    var copiaDeEdad = edad;
```

```
        return copiaDeEdad >= 18;
    }
```

Código 3

Los códigos 1 y 2 comparan una variable con un número para identificar si es o no mayor. Sin embargo las constantes utilizadas son diferentes y el espaciado del texto escrito también difiere (uno utiliza espacios para separar sentencias mientras que el otro utiliza saltos de línea). Al aplicar nuestro algoritmo, estandarizamos el formato del texto y renombramos todas las variables y constantes de manera similar. Luego nos quedan dos programas exactamente iguales. Por lo tanto podemos determinar que ambos códigos inicialmente eran *equivalentes bajo etiquetas* si se deja de lado el formato del texto.

Ahora bien, tras correr el algoritmo, el código 3 no será clasificado como equivalente a los programas anteriores. A simple vista podemos reconocer que semánticamente es equivalente al código 2, sin embargo no estamos pudiendo capturar que este código es equivalente en la clasificación.

Lo que podemos inferir tras hacer esta observación es que lo que nuestro algoritmo permite identificar es cuando un usuario potencialmente tomó prestado el código de otro lado (es decir hizo un *copy and paste* del código) y cambió los valores de las variables, constantes y espacios del código. Esta hipótesis radica en la presencia de programas bastante complejos para usuarios no expertos, los cuales podrían estar basados en códigos de ejemplo y adaptados con las variables de contexto asociadas.

Tras la ejecución del algoritmo se redujo de miles a cientos la cantidad de programas, el cual, es un número más manejable.

## 2.2. Análisis de los programas

Siendo ahora el corpus mucho más pequeño nos enfrentamos a la decisión de hacer una heurística que nos trate de separar los programas en familias en base a características predefinidas de antemano o bien hacer una clasificación manual de los mismos.

Se decidió ir por el camino manual debido únicamente a una cuestión de tiempos. El número de programas a clasificar es lo suficientemente pequeño como para poder revisarlos en tan solo cuestión de minutos/horas.

Gracias a la construcción de un programa que presenta en pantalla un código y da la opción de definir una clasificación en base a características, el proceso se hizo de manera asincrónica. Más detalles de su implementación se encuentran en el capítulo 4.

En la subsección que sigue se detallan algunas de las características frecuentes que aparecen repetidas en varios programas, las cuales permitirán entender mejor cuales son los casos para tener en cuenta a la hora de definir el DSL.

### 2.2.1. Características frecuentes

Hay muchos patrones u operaciones de código repetidos en los programas analizados, los cuales aparecen muchas veces en conjunción.

#### Anidamiento de condicionales (`ifs`)

Uno de los patrones más usados es el anidamiento de `ifs`, con sentencias de retorno en cada una de las ramas. Es esto una sorpresa debido a que los programas tienen como

retorno un valor booleano y podrían simplificarse en tan solo una expresión booleana. Es posible que esto ocurra debido a que presentar los programas en tan solo una expresión es más difícil de seguir que un código con `ifs` donde es más legible en cada caso donde se está parado. Otra opción bien podría ser que los usuarios no tengan mucho conocimiento de lógica proposicional y por eso prefieran cadenas de ifs.

```
function volveriaContestóMucho() {
    const habitacion = context.get("comentario_sobre_habitación");
    const comidas = context.get("valoración_de_comidas");
    const vuelve = context.get("volverías");
    if (vuelve === "Si") {
        if (habitacion == null || comidas == null) {
            return false;
        }
        if (habitacion.length > 40 || comidas > 7) {
            return true;
        }
        return false;
    } else {
        return false;
    }
};
```

### Chequeo de presencia de una variable o si ha sido llenada

Es esta una operación que aparece con frecuencia debido a que permite en el caso de uso de estos programas determinar si un usuario ha realizado una acción.

```
function noRealizoLlamadas() {
    const respuesta1 = context.get("calidad_llamada");
    const respuesta2 = context.get("calidad_videollamada");
    if (respuesta1 == null || respuesta2 == null || respuesta1.length
        == 0) return true;
    else return false;
}
```

### Aplicación de expresiones regulares sobre variables

Es una operación muy común y se suele utilizar para la validación de datos ingresados por usuarios. La aplicación de expresiones regulares es una herramienta muy poderosa pero que requiere de bastante conocimiento para utilizarla de manera correcta.

```
function fechaEnFormatoInvalido() {
    const fecha = context.get("fecha");
    const codeRegexp = /\d{4}-\d{2}-\d{2}/;
    if (codeRegexp.test(fecha)) return false;
    else return true;
}
```

### Validaciones sobre la longitud de una variable

Esta operación es muy útil para poder determinar si el usuario no ha respondido o bien la respuesta es muy corta. También suele aparecer en conjunto con el `split` de `strings` para determinar que cada una de las partes del valor tenga el largo correcto (por ejemplo es común para corroborar validez en números de teléfonos).

```
function telefonoValido() {
    const test = context.get("telefono");
    if (test == null) return false;
    if (test.split('-').length == 2) return true;
    return false;
};
```

### Contabilizar la cantidad de variables que cumplen con una propiedad

En general se determina la cantidad de variables que tienen un valor y ese valor se lo compara con una constante para determinar el valor de verdad. Esto es logrado a partir de o bien hacer un `if` con la suma de todos los valores o bien a través de la iteración de un arreglo con todas las variables donde se acumulan estos valores en una variable y finalmente se compara.

```
function clienteMayormenteContento() {
    const vendedor = context.get("satisfaccion_vendedor");
    const producto = context.get("satisfaccion_producto");
    const limpieza = context.get("satisfaccion_limpieza");
    const entrega = context.get("satisfaccion_entrega");
    if (vendedor + producto + limpieza + entrega > 32)
        return true;
    return false;
};
```

El programa más común fue aquel en el que se chequea la presencia de una variable, luego que contenga contenido y en tal caso la aplicación de una expresión regular para chequear que haya sido bien formado.

```
function reportoTarjetaInvalida() {
    const numero_tarjeta = context.get("numero_tarjeta");
    if (numero_tarjeta == null) {
       return true;
    } else if (numero_tarjeta == "") {
        return true;
    } else {
        var formato_visa = /^4[0-9]{12}(?:[0-9]{3})?$/;
        if (!formato_visa.exec(numero_tarjeta)) {
            return true;
        }
        return false;
    }
};
```

Código 4

Existen grandes pistas que marcan el camino por donde construir un DSL que tenga un gran poder expresivo y que mantenga simplicidad para su uso.

# 3. DEFINICIÓN DEL LENGUAJE

El lenguaje debe ser capaz de representar expresiones booleanas sobre diferentes tipos de variables: números, cadenas de caracteres, listas; todas ellas sin *side effects*, es decir de solo lectura y que no afectan a ningún estado global. Son indispensables por ejemplo inecuaciones, operaciones sobre listas que prediquen sobre existencia de elementos en las mismas, o bien una conjunción de operaciones entre listas y números que resulten en una operación booleana, como bien puede ser contar la cantidad de elementos de una lista y luego predicar sobre esa cantidad.

El Lambda Cálculo es una gran herramienta para definir formalmente lenguajes. Es por eso que definiremos un lambda cálculo tipado que se ajuste a las necesidades del lenguaje que estamos definiendo.

## 3.1. Tipos

Los tipos asociados son $\sigma ::= Bool \mid Nat \mid []_\sigma \mid \{l_i = \sigma^{i \in 1..n}\} \mid String \mid Regexp$

Están definidos valores booleanos, numéricos, listas, registros, strings y expresiones regulares, todos ellos con operaciones asociadas que se definen a partir de los siguientes términos.

## 3.2. Términos

A continuación se detallan los términos que acepta el lenguaje definido con una pequeña explicación de qué representa cada uno.

$\langle$ M, N, P $\rangle$ :=

| | Término | Descripción |
|---|---|---|
| | `true` | *Verdadero* |
| \| | `false` | *Falso* |
| \| | `if M then N else P` | *Condicional* |
| \| | `M v N` | *Disyunción* |
| \| | `M & N` | *Conjunción* |
| \| | `not M` | *Negación* |
| \| | `exactly M of N` | *Hay exactamente M booleanos verdaderos en la lista N* |
| \| | `less than M of N` | *Hay menos de M booleanos verdaderos en la lista N* |
| \| | `more than M of N` | *Hay más de M booleanos verdaderos en la lista N* |
| \| | `at least M of N` | *Hay al menos M booleanos verdaderos en la lista N* |
| \| | `at most M of N` | *Hay a lo mucho M booleanos verdaderos en la lista N* |
| \| | `0` | *Valores numéricos (cero)* |
| \| | `suc(M)` | *Sucesor de un número* |
| \| | `pred(M)` | *Predecesor de un número* |
| \| | `isZero(M)` | *Determinar si un número es cero* |
| \| | `M = N` | *Igual* |
| \| | M $\neq$ N | *Distinto* |
| \| | `M < N` | *Menor* |
| \| | `M <= N` | *Menor o igual* |
| \| | `M > N` | *Mayor* |
| \| | `M >= N` | *Mayor o igual* |
| \| | `M + N` | *Suma de dos números* |
| \| | `M - N` | *Resta de dos números* |
| \| | `M * N` | *Multiplicación de dos números* |
| \| | `M / N` | *División entera de dos números* |

| | | |
|---|---|---|
| \| | `[]` | *Lista vacía* |
| \| | `M::N` | *Agrega un elemento a la lista.* |
| \| | `M[N]` | *Elemento en la posición N de la lista M* |
| \| | `count N M` | *Cuenta la cantidad de valores N que contiene la lista M.* |
| \| | `length M` | *Longitud de la lista M* |
| \| | `M in N` | *Determina si M está en la lista N* |
| \| | $\{l_i = M^{\text{i in 1..n}}\}$ | *Registros* |
| \| | $M.l$ | *Devuelve la posición l del registro M* |

Dado un alfabeto finito $\Sigma$,
Expresiones regulares

$\langle$ r, w $\rangle$ :=

| | | |
|---|---|---|
| | $\emptyset$ | *Expresión regular vacía* |
| \| | `a` | *Elemento de* $\Sigma$ |
| \| | `r | r` | *Unión* |
| \| | `r + r` | *Concatenación* |
| \| | `r*` | *Clausura de Kleene de r* |
| \| | `r = w` | *Igual* |
| \| | r $\neq$ w | *Distinto* |

Strings

$\langle$ S, T, R $\rangle$ :=

| | | |
|---|---|---|
| | $\lambda$ | *String vacío* |
| \| | `aS` | *Donde a es parte de* $\Sigma$ |
| \| | `toLower(S)` | *El String S en minúscula* |
| \| | `toUpper(S)` | *El String S en mayúscula* |
| \| | `validEmail(S)` | *El String S representa un email válido* |
| \| | `trim(S)` | *El String S sin espacios al principio ni final* |
| \| | `S matches R` | *El String S matchea la expresión regular R* |
| \| | `S = T` | *Igual* |
| \| | S $\neq$ T | *Distinto* |
| \| | `split S a` | *El String S partido en base al caracter a* |

## 3.3. Axiomas de tipado

Se definen a continuación los axiomas de tipado necesarios para poder definir cuando una expresión está bien formada en base a la los tipos que esperan determinadas expresiones y la cohesión de los tipos definidos previamente.

### Valores Booleanos y operaciones básicas

$$\frac{}{\Gamma \triangleright true : Bool}\text{ T-true} \qquad \frac{}{\Gamma \triangleright false : Bool}\text{ T-false}$$

$$\frac{\Gamma \triangleright M : Bool}{\Gamma \triangleright not(M) : \ Bool}\text{ T-not}$$

$$\frac{\Gamma \triangleright M : Bool \quad \Gamma \triangleright P : Bool}{\Gamma \triangleright M \ \vee \ P \ : Bool}\text{ T-or} \qquad \frac{\Gamma \triangleright M : Bool \quad \Gamma \triangleright P : Bool}{\Gamma \triangleright M \ \& \ P \ : Bool}\text{ T-and}$$

### If

$$\frac{\Gamma \triangleright M : Bool \quad \Gamma \triangleright P : \sigma \quad \Gamma \triangleright Q : \sigma}{\Gamma \triangleright if \;\; M \;\; then \;\; P \;\; else \;\; Q \;\; : \sigma} \text{T-if}$$

### Operadores sobre conjunto de condiciones

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : []_{Bool}}{\Gamma \triangleright exactly \;\; M \;\; of \;\; P : Bool} \text{T-exactly} \qquad \frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : []_{Bool}}{\Gamma \triangleright at \;\; least \;\; M \;\; of \;\; P : Bool} \text{T-at-least}$$

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : []_{Bool}}{\Gamma \triangleright at \;\; most \;\; M \;\; of \;\; P : Bool} \text{T-at-most} \qquad \frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : []_{Bool}}{\Gamma \triangleright less \;\; than \;\; M \;\; of \;\; P : Bool} \text{T-less-than}$$

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : []_{Bool}}{\Gamma \triangleright more \;\; than \;\; M \;\; of \;\; P : Bool} \text{T-more-than}$$

### Operadores sobre listas

$$\frac{}{\Gamma \triangleright []_\sigma : []_\sigma} \text{T-empty-list} \qquad \frac{\Gamma \triangleright M : \sigma \quad \Gamma \triangleright P : []_\sigma}{\Gamma \triangleright M : P : []_\sigma} \text{T-list}$$

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : []_\sigma}{\Gamma \triangleright P[M] : \sigma} \text{T-at} \qquad \frac{\Gamma \triangleright M : \sigma \quad \Gamma \triangleright P : []_\sigma}{\Gamma \triangleright count \;\; M \;\; P : Nat} \text{T-count}$$

$$\frac{\Gamma \triangleright M : \sigma \quad \Gamma \triangleright P : []_\sigma}{\Gamma \triangleright M \;\; in \;\; P : Bool} \text{T-in} \qquad \frac{\Gamma \triangleright M : []_\sigma}{\Gamma \triangleright length \;\; M : Nat} \text{T-length}$$

### Operadores numéricos

$$\frac{}{\Gamma \triangleright 0 : Nat} \text{T-zero} \qquad \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright suc(M) : Nat} \text{T-suc}$$

$$\frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright pred(M) : Nat} \text{T-pred} \qquad \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright isZero(M) : Bool} \text{T-is-zero}$$

### Operadores lógicos de orden

$$\frac{\Gamma \triangleright M : \sigma \quad \Gamma \triangleright P : \sigma}{\Gamma \triangleright M \;\; = \;\; P : Bool} \text{T-equal} \qquad \frac{\Gamma \triangleright M : \sigma \quad \Gamma \triangleright P : \sigma}{\Gamma \triangleright M \;\; \neq \;\; P : Bool} \text{T-not-equal}$$

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \;\; < \;\; P : Bool} \text{T-less} \qquad \frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \;\; \leq \;\; P : Bool} \text{T-less-or-equal}$$

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \;\; > \;\; P : Bool} \text{T-greater} \qquad \frac{\Gamma \triangleright M : Nat \; \Gamma \triangleright P : Nat}{\Gamma \triangleright M \;\; \geq \;\; P : Bool} \text{T-greater-or-equal}$$

### Operadores aritméticos

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \ + \ P : Nat} \text{T-sum} \qquad \frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \ - \ P : Nat} \text{T-diff}$$

$$\frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \ * \ P : Nat} \text{T-mult} \qquad \frac{\Gamma \triangleright M : Nat \quad \Gamma \triangleright P : Nat}{\Gamma \triangleright M \ / \ P : Nat} \text{T-div}$$

### Registros

$$\frac{\Gamma \triangleright M_i : \sigma_i \quad para \ \ cada \ \ i \ \ \in 1..n}{\Gamma \triangleright \{l_i = M_i^{\text{i} \in 1..\text{n}}\} \ : \ \{l_i : \sigma_i^{\text{i} \in 1..\text{n}}\}} \text{T-rcd} \qquad \frac{\Gamma \triangleright M : \{l_i : \sigma_i^{\text{i} \in 1..\text{n}}\} \quad j \in 1..n}{\Gamma \triangleright M.l_j \ : \sigma_j} \text{T-proj}$$

### Strings y Expresiones regulares

$$\frac{}{\Gamma \triangleright \lambda : String} \text{T-empty-string} \qquad \frac{\Gamma \triangleright M : String, \quad \Gamma \triangleright a : \Sigma}{\Gamma \triangleright aM : String} \text{T-string-concat}$$

$$\frac{\Gamma \triangleright M : String}{\Gamma \triangleright toUpper(M) : String} \text{T-to-upper} \qquad \frac{\Gamma \triangleright M : String}{\Gamma \triangleright toLower(M) : String} \text{T-to-lower}$$

$$\frac{\Gamma \triangleright M : String}{\Gamma \triangleright trim(M) : String} \text{T-trim} \qquad \frac{\Gamma \triangleright M : String \quad \Gamma \triangleright a : \Sigma}{\Gamma \triangleright split \ \ M \ \ a \ \ : []_{String}} \text{T-split}$$

$$\frac{\Gamma \triangleright M : String}{\Gamma \triangleright validEmail(M) : Bool} \text{T-valid-email}$$

$$\frac{}{\Gamma \triangleright \emptyset : Regexp} \text{T-empty-regexp} \qquad \frac{\Gamma \triangleright M : Regexp, \quad \Gamma \triangleright P : Regexp}{\Gamma \triangleright M \ + \ P : Regexp} \text{T-regexp-concat}$$

$$\frac{\Gamma \triangleright M : Regexp, \quad \Gamma \triangleright P : Regexp}{\Gamma \triangleright M \ | \ P : Regexp} \text{T-regexp-union} \qquad \frac{\Gamma \triangleright M : Regexp}{\Gamma \triangleright M^* : Regexp} \text{T-regexp-kleene}$$

$$\frac{\Gamma \triangleright M : String \quad \Gamma \triangleright P : Regexp}{\Gamma \triangleright M \ \ matches \ \ P : Bool} \text{T-matches-regexp}$$

## 3.4. Semántica

Nos concentraremos en la definición de las reglas de evaluación en un paso necesarias para poder definir la semántica operacional de los términos.

### Not

$$\frac{}{not\ \ false \to true}\ \text{E-not-false} \qquad \frac{}{not\ \ true \to false}\ \text{E-not-true}$$

### Or

$$\frac{M_1 \to M_1'}{M_1 \vee M_2 \to M_1' \vee M_2}\ \text{E-or}$$

$$\frac{}{true \vee M_2 \to true}\ \text{E-or-true} \qquad \frac{}{false \vee M_2 \to M_2}\ \text{E-or-false}$$

### And

$$\frac{M_1 \to M_1'}{M_1 \,\&\, M_2 \to M_1' \,\&\, M_2}\ \text{E-and}$$

$$\frac{}{true \,\&\, M_2 \to M_2}\ \text{E-and-true} \qquad \frac{}{false \,\&\, M_2 \to false}\ \text{E-and-false}$$

### If

$$\frac{}{if\ \ true\ \ then\ \ M_1\ \ else\ \ M_2\ \ \to M_1}\ \text{E-if-true}$$

$$\frac{}{if\ \ false\ \ then\ \ M_1\ \ else\ \ M_2\ \ \to M_2}\ \text{E-if-false}$$

$$\frac{M_1 \to M_1'}{if\ \ M_1\ \ then\ \ M_2\ \ else\ \ M_3\ \ \to if\ \ M_1'\ \ then\ \ M_2\ \ else\ \ M_3}\ \text{E-if}$$

### Operador = en general

$$\frac{}{M \neq N \to not(M = N)}\ \text{E-not-equal}$$

$$\frac{M_1 \to M_1'}{M_1 = M_2 \to M_1' = M_2}\ \text{E-equal} \qquad \frac{M_2 \to M_2'}{M_1 = M_2 \to M_1 = M_2'}\ \text{E-equal-2}$$

### Operador = sobre valores booleanos

$$\frac{}{true = true \to true}\ \text{E-equal-true-true} \qquad \frac{}{true = false \to false}\ \text{E-equal-true-false}$$

$$\frac{}{false = true \to false}\ \text{E-equal-false-true} \qquad \frac{}{false = false \to true}\ \text{E-equal-false-false}$$

### Operadores sobre conjunto de condiciones

$$\frac{}{exactly\ \ n\ \ of\ \ L \rightarrow n = count\ \ true\ \ L}\ \text{E-exactly}$$

$$\frac{}{less\ \ than\ \ n\ \ of\ \ L \rightarrow n > count\ \ true\ \ L}\ \text{E-less-than}$$

$$\frac{}{more\ \ than\ \ n\ \ of\ \ L \rightarrow n < count\ \ true\ \ L}\ \text{E-more-than}$$

$$\frac{}{at\ \ least\ \ n\ \ of\ \ L \rightarrow n \leq count\ \ true\ \ L}\ \text{E-at-least}$$

$$\frac{}{at\ \ most\ \ n\ \ of\ \ L \rightarrow n \geq count\ \ true\ \ L}\ \text{E-at-most}$$

### Operadores sobre listas

$$\frac{N \rightarrow N'}{(M : L)[N] \rightarrow (M : L)[N']}\ \text{E-at}$$

$$\frac{}{(M : L)[suc(N)] \rightarrow L[N]}\ \text{E-at-num} \qquad \frac{}{(M : L)[0] \rightarrow M}\ \text{E-at-zero}$$

$$\frac{}{count\ \ M\ \ [] \rightarrow 0}\ \text{E-count-empty}$$

$$\frac{}{count\ \ O\ \ (M : N) \rightarrow \textbf{if}\ \ M = O\ \ \textbf{then}\ \ suc\ \ (count\ \ O\ \ N)\ \ \textbf{else}\ \ count\ \ O\ \ N}\ \text{E-count}$$

$$\frac{}{M\ \ in\ \ N \rightarrow count\ \ M\ \ N > 0}\ \text{E-in}$$

$$\frac{}{length\ \ [] \rightarrow 0}\ \text{E-length-empty} \qquad \frac{}{length\ \ (M : N) \rightarrow suc(length(N))}\ \text{E-length-list}$$

$$\frac{}{[]\ \ =\ \ [] \rightarrow true}\ \text{E-equal-both-lists-empty} \qquad \frac{}{[]\ \ = (M : N) \rightarrow false}\ \text{E-equal-first-list-empty}$$

$$\frac{}{M : N\ \ = [] \rightarrow false}\ \text{E-equal-second-list-empty} \qquad \frac{}{(M : N) = (O : P) \rightarrow M = O\ \ \&\ \ N = P}\ \text{E-equal-non-e}$$

### Operadores aritméticos

$$\frac{M_1 \to M_1'}{suc(M_1) \to suc(M_1')} \text{ E-suc}$$

$$\frac{}{pred(0) \to 0} \text{ E-pred-zero} \qquad \frac{}{pred(suc(M)) \to M} \text{ E-pred-suc}$$

$$\frac{M_1 \to M_1'}{pred(M_1) \to pred(M_1')} \text{ E-pred}$$

$$\frac{}{isZero(0) \to true} \text{ E-iszero-zero} \qquad \frac{}{isZero(suc(M)) \to false} \text{ E-iszero-suc}$$

$$\frac{M_1 \to M_1'}{isZero(M_1) \to isZero(M_1')} \text{ E-iszero}$$

$$\frac{N \to N'}{M + N \to M + N'} \text{ E-sum} \qquad \frac{}{M + 0 \to M} \text{ E-sum-zero}$$

$$\frac{}{M + suc(N) \to suc(M) + N} \text{ E-sum-num} \qquad \frac{M \to M'}{M + N \to M' + N} \text{ E-sum-left-reduction}$$

$$\frac{N \to N'}{M - N \to M - N'} \text{ E-diff} \qquad \frac{}{M - 0 \to M} \text{ E-diff-zero}$$

$$\frac{}{M - suc(N) \to pred(M) - N} \text{ E-diff-num} \qquad \frac{M \to M'}{M - N \to M' - N} \text{ E-diff-letf-reduction}$$

$$\frac{}{M * 0 \to 0} \text{ E-mult-zero} \qquad \frac{}{M * suc(0) \to M} \text{ E-mult-one}$$

$$\frac{N \to N'}{M * N \to M * N'} \text{ E-mult} \qquad \frac{}{M * suc(N) \to M + (M * (N - suc(0)))} \text{ E-mult-num}$$

$$\frac{N \to N'}{M/N \to M/N'} \text{ E-div} \qquad \frac{}{M/N \to if(M < N) \;\; then \;\; 0 \;\; else \;\; suc((M - N)/N))} \text{ E-div-num}$$

### Operadores lógicos de orden

$$\frac{}{M \leq N \rightarrow N > M}\ \text{E-less-or-equal} \qquad \frac{}{M < N \rightarrow N \geq M}\ \text{E-less}$$

$$\frac{}{M \geq N \rightarrow M > N \vee M = N}\ \text{E-greater-or-equal}$$

$$\frac{}{N = M \rightarrow \begin{array}{l} if\ isZero(N)\ \&\ isZero(M)\ then \\ \quad true \\ else\ if\ isZero(N) \vee isZero(M)\ then \\ \quad false \\ else \\ \quad pred(N) = pred(M) \end{array}}\ \text{E-equal-nat}$$

$$\frac{}{N > M \rightarrow \begin{array}{l} if\ isZero(N)\ then \\ \quad false \\ else\ if\ isZero(M)\ then \\ \quad true \\ else \\ \quad pred(N) > pred(M) \end{array}}\ \text{E-greater}$$

### Registros

$$\frac{j \in 1..n}{\{l_i = M^{\text{i} \in \text{1..n}}\}.l_j \rightarrow M_j}\ \text{E-proj-reg} \qquad \frac{M \rightarrow M'}{M.l \rightarrow M'.l}\ \text{E-proj}$$

$$\frac{M_j \rightarrow M_j'}{\{l_i = M^{\text{i} \in \text{1..j-1}}, l_j = M_j, l_i = M^{\text{i} \in \text{j+1..n}}\} \rightarrow \{l_i = M^{\text{i} \in \text{1..j-1}}, l_j = M_j', l_i = M^{\text{i} \in \text{j+1..n}}\}}\ \text{E-rcd}$$

$$\frac{}{\{m_i = M^{\text{i} \in \text{1..m}}\} = \{n_i = N^{\text{i} \in \text{1..n}}\} \rightarrow m = n\ \&\ (\forall i \in 1..n)\ m_i = n_i}\ \text{E-equal-reg}$$

### Strings

$$\frac{}{\lambda = \lambda \rightarrow true}\ \text{E-equal-empty-strings} \qquad \frac{}{aS = \lambda \rightarrow false}\ \text{E-equal-first-string-empty}$$

$$\frac{}{\lambda = aS \rightarrow false}\ \text{E-equal-second-string-empty} \qquad \frac{}{aS = bT \rightarrow a = b\ \&\ S = T}\ \text{E-equal-non-empty-strings}$$

Para definir la semántica de algunas expresiones tales como `M matches P` nos tomaremos una licencia para hacerlo de manera coloquial.

Una expresión regular representa en definitiva una manera de determinar un criterio de búsqueda dentro de un string. Por lo tanto el término en cuestión (`M matches P`) devuelve un valor booleano que representa si se pudo encontrar utilizando el patrón de búsqueda P en el string dado M.

`toLowerCase(M)` devuelve un nuevo string partiendo del string de parámetro, el cual reemplaza todos los caracteres que representan letras por su equivalente en minúscula. Similarmente, `toUpperCase(M)` hace lo propio devolviendo los caracteres en mayúscula.

`validEmail(M)` determina si el string dado representa un email válido.

`trim(M)` devuelve un nuevo string eliminando los caracteres que representan espacios que aparecen al principio y final del string.

`split M a` devuelve una lista de substrings a partir de separar el string original en base al caracter a.

Definir la igualdad de expresiones regulares no es una tarea tan sencilla como para el resto de los tipos. En este caso es necesario evaluar si el lenguaje generado por ambas expresiones es el mismo. Para poder hacerlo hay que chequear isomorfismo sobre los autómatas finitos determinísticos mínimos asociados a las expresiones regulares. La gran desventaja asociada a la implementación de dicho chequeo es que no se puede resolver en tiempo polinomial.

# 4. IMPLEMENTACIÓN

## 4.1. Lenguaje

Partiendo de la base que tenemos como objetivo la migración de programas escritos en JavaScript a nuestro propio lenguaje, hemos decidido construir el nuestro sobre JavaScript. Es decir, tanto la implementación del AST, el parser y la ejecución de nuestro lenguaje se basarán completamente en el motor de JavaScript. Sin embargo, es esto totalmente transparente para el usuario de nuestro lenguaje.

La decisión se basa en que las herramientas necesarias para poder parsear y traducir un lenguaje dado, definitivamente tendrán mejor soporte en el mismo lenguaje para poder hacer una especie de metaprogramación. Por ejemplo, es muy común en el uso de herramientas que detectan malas prácticas de programación, estilos o *bugs* (conocidas como (*linters*).

### 4.1.1. Abstract Sintax Tree

En la construcción de nuestro AST se determinó construir una jerarquía de herencia para los distintos tipos de condiciones. El elemento principal del AST es una `Condition`, la cual puede estar compuesta por varias condiciones adentro.

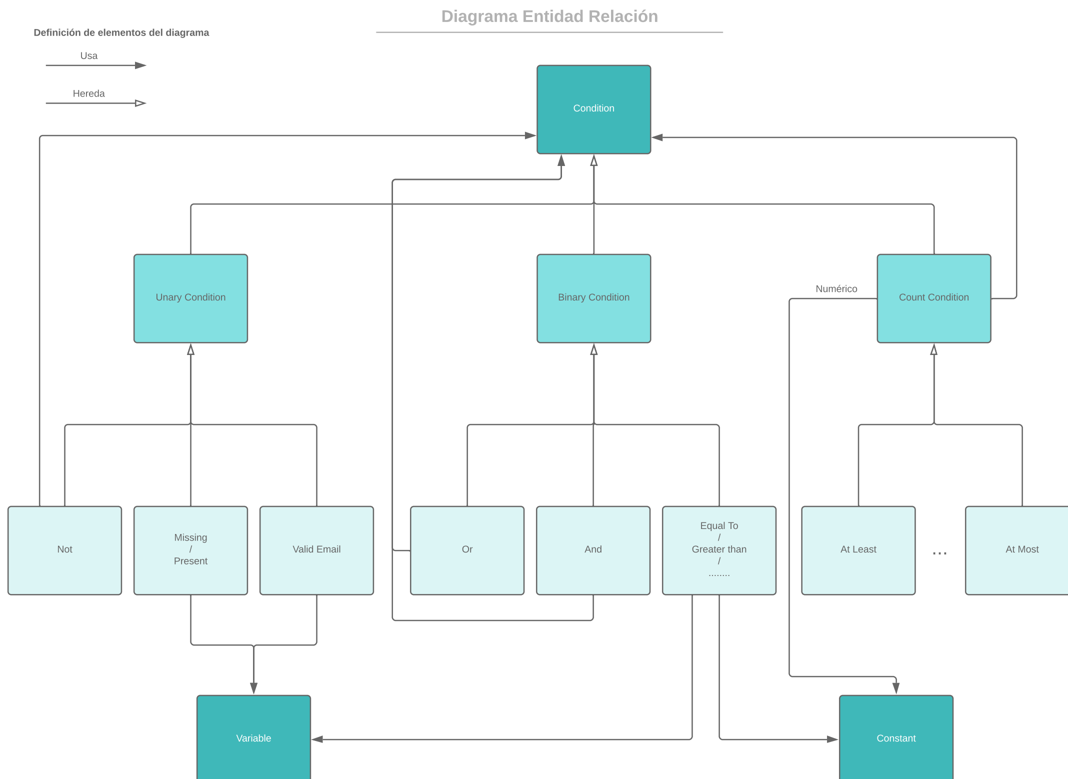


**Fig. 4.1:** Diagrama entidad relación para un subconjunto de entidades

Una *Condition* responde ante dos mensajes importantes: `toString` y `evaluate(context)`.
El primero permite que dado un objeto de nuestro AST, éste pueda devolverse en su representación String igual a la que el usuario debe escribir para que nuestro parser lo comprenda.
El segundo permite que podamos evaluar las condiciones en base a un conjunto de variables con valores determinado por un contexto. De esta forma, podemos evaluar una expresión obteniendo como resultado el valor dictado por la semántica del lenguaje. Para tratar de garantizar que la implementación de la evaluación es correcta, se han escrito tests de unidad sobre cada uno de los tipos de condiciones existentes cubriendo el cien por ciento de las ramas.

### 4.1.2. Parser

Dado ya el *Abstract Syntax Tree* del lenguaje, se deben poder generar dichos objetos basados en un programa escrito en el lenguaje. Estos programas deben seguir una sintaxis asociada, la cual está definida a partir de la gramática presentada a continuación.

#### Gramática

Definimos los elementos sintácticos del lenguaje mediante una gramática libre de contexto. Para eso se debe definir cada uno de sus componentes: *elementos terminales (Σ), elementos no terminales **(N)**, conjunto de producciones **(P)** y el símbolo distinguido **(S)*** que determina el símbolo de comienzo. Cabe destacar que la gramática presentada a continuación se ha definido de manera que resulte sencilla de comprender al lector, tomándose así la licencia de que sea ambigua a simple vista.

$\Sigma = \{$ *true* *false* ( ) $<$ $>$ $=$ *is_integer* $\leq$ $\geq$ $\neq$ *not* *and* *or* [ ] *filled* *in* *matches* *length* *context.get* *at least* *at most* *less than* *more than* *exactly* *of* *valid_email* *as_integer* $\}$

$N = \{$ CONDITION CONDITION_LIST COUNT_OPERATOR DIGIT V INT_VARIABLE LITERAL STRING_LITERAL INT_LITERAL LITERALS LITERAL_LIST INEQ_OP ELEMENTS INTEGER LETTER_OR_DIGIT $\}$

P

```
CONDITION  -->    true  |
                  false |
                  not CONDITION |
                  CONDITION or CONDITION |
                  CONDITION and CONDITION |
                  COUNT_OPERATOR CONDITION_LIST |
                  (CONDITION) |

                  VARIABLE filled |
                  VARIABLE missing |
                  VARIABLE in LITERAL_LIST |
                  VARIABLE = LITERAL_LIST[INTEGER] |
                  VARIABLE is_integer |

                  STRING BINARY_STR_OP STRING |
                  STRING valid_email |
```

```
                   INTEGER INEQ_OP INTEGER |

CONDITION_LIST-->  [ELEMENTS] | []
ELEMENTS      -->  CONDITION |
                   CONDITION, ELEMENTS

COUNT_OPERATOR-->  at least INT_LITERAL of  |
                   at most INT_LITERAL of   |
                   more than INT_LITERAL of |
                   less than INT_LITERAL of |
                   exactly INT_LITERAL of

STRING        -->  VARIABLE |
                   as_string(context.get(STRING_LITERAL)) |
                   STRING_LITERAL |
                   to_upper STRING |
                   to_lower STRING |
                   trim STRING |
                   LITERAL_LIST[INTEGER]

BINARY_STR_OP -->  = |
                   matches

INTEGER       -->  VARIABLE |
                   as_integer(context.get(STRING_LITERAL)) |
                   INT_LITERAL |
                   length VARIABLE |
                   INTEGER + INTEGER |
                   INTEGER - INTEGER |
                   INTEGER * INTEGER |
                   INTEGER / INTEGER |
                   (INTEGER) |
                   LITERAL_LIST[INTEGER]

INEQ_OP       -->  < |
                   > |
                   ≥ |
                   ≤ |
                   = |
                   ≠

LITERAL_LIST  -->  [] | [LITERALS] |
                   split STRING 'LETTER_OR_DIGIT'
LITERALS      -->  LITERAL |
                   LITERAL, LITERALS

VARIABLE      --> context.get(STRING_LITERAL)

LITERAL       -->  INT_LITERAL |
                   STRING_LITERAL
```

Las definiciones de la gramática que siguen más abajo siguen un esquema de expresión regular para simplificar la notación.

```
INT_LITERAL   -->  DIGIT+
```

```
STRING_LITERAL  ⟶  '(LETTER_OR_DIGIT)+'

DIGIT           ⟶  [0-9]

LETTER_OR_DIGIT ⟶  DIGIT |
                   [a-zA-Z_]
```

S = CONDITION.

Hay pequeñas diferencias entre la sintaxis del lenguaje definida por la gramática y la que utilizamos para el cálculo lambda. Sin embargo, dichas diferencias no afectan a como se entiende el lenguaje.

Una de ellas es el armado de las listas, donde en un caso se utiliza la notación de agregado entre elementos mediante dos puntos(:) y se comienza con un par de corchetes([]), mientras que en la gramática utilizamos la notación que se conoce en la mayoría de los lenguajes de programación: comenzando con un corchete, luego elementos separados por coma y cerrando con otro corchete.

Otra diferencia que requiere de un mayor análisis es como se piensan las variables dentro del lenguaje. Como en nuestro lenguaje asumimos de entrada un diccionario de variables inmutable, decidimos modelarlo en el cálculo lambda a través de un registro con muchos términos adentro. La única diferencia entre uno y otro es que el diccionario acepta strings como claves mientras que los registros aceptan números para referenciar a cada uno de los términos. Como los diccionarios de entrada no son infinitos, podemos establecer una biyección entre la lista de claves y una lista de números, con lo cual podemos determinar que en términos prácticos, un registro nos alcanza para modelar el mismo comportamiento que el lenguaje expresa.

### 4.1.3. Tecnologías utilizadas

ANTLR (ANother Tool for Language Recognition)[6] es una herramienta que genera parsers para poder leer, procesar, ejecutar y traducir un texto que cumple con una gramática dada. Es ésta entonces una herramienta que cumple con nuestra necesidad de poder parsear un texto y convertirlo en nuestra representación del abstract syntax tree del lenguaje implementado.

ANTLR recibe como entrada la gramática del lenguaje, que previamente ya definimos, y un lenguaje de entrada y genera como output: un lexer, para poder tokenizar la entrada, un parser que junto con el lexer permite generar luego como salida un árbol de la sintaxis el cual puede ser recorrido mediante un visitor para poder traducirlo a objetos de dominio. Todo el código generado se define en el lenguaje de entrada especificado por el usuario.

En nuestro caso particular, elegimos JavaScript ya que es el lenguaje en el que ya tenemos definidos los objetos del AST de nuestro lenguaje.

Otra ventaja de ANTLR es que permite agregar etiquetas a cada una de las producciones de la gramática, haciendo que los nodos del árbol generado sean fácilmente identificables. De esta manera, podemos hacer que el mapeo entre los nodos del árbol y nuestros objetos del AST sea mucho más sencillo, y en muchos casos una relación 1 a 1.

El código 5 define como se visitan las operaciones binarias **or** y **and** dentro del lenguaje. El objeto ctx representa el nodo del árbol de parseo, y como se puede ver, contiene

definidas la propiedades **left**, **op** y **right**. Toda esa metadata está definida en el archivo de las producciones tal como se ve en el comentario de la implementación del código. **Conditions.Or** y **Conditions.And** son los objetos de dominio del AST del lenguaje.

```
// expr := left=expr op=booleanOperation right=expr    # binaryExpr
// booleanOperation := ‘or’ | ‘and’;
handleBinaryOperation(ctx) {
    switch (ctx.op.getText()) {
        case "and":
            return new Conditions.And(ctx.left.accept(this), ctx.right.
                accept(this));
        case "or":
            return new Conditions.Or(ctx.left.accept(this), ctx.right.
                accept(this));
    }
}
```

Código 5

Para tratar de garantizar que la interpretación es correcta, se han escrito tests que iteran sobre los distintos elementos del lenguaje y que tras convertirlos en el string asociado son parseables y los elementos obtenidos son los mismos de los cuales se partió.

### 4.1.4. Ejecución del lenguaje

Dado un programa escrito en nuestro lenguaje y un diccionario de variables, los pasos para ejecutarlo son muy sencillos:

1. El parser lee el string de entrada y lo traduce a un AST.
2. Se consigue el diccionario de variables a través de parsear un string de entrada.
3. Se le pide al nodo root del AST que evalúe si la condición es verdadera basado en el diccionario de variables.

Por ejemplo, dado el siguiente programa:

`not context.get('pregunta') missing and length context.get('pregunta') <10`

El árbol de derivación asociado es el de la figura 4.2 y al traducir dicho árbol a nuestro árbol de sintaxis abstracto, obtenemos el de la figura 4.3

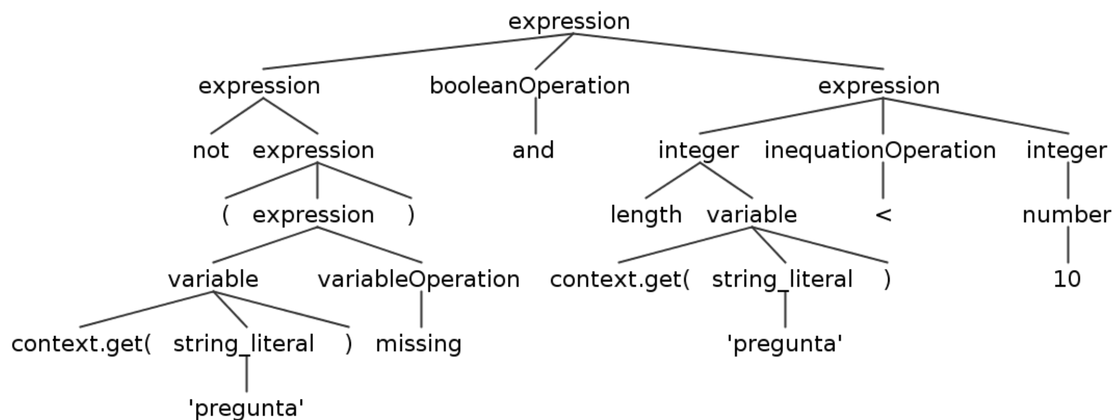


**Fig. 4.2:** Árbol de derivación

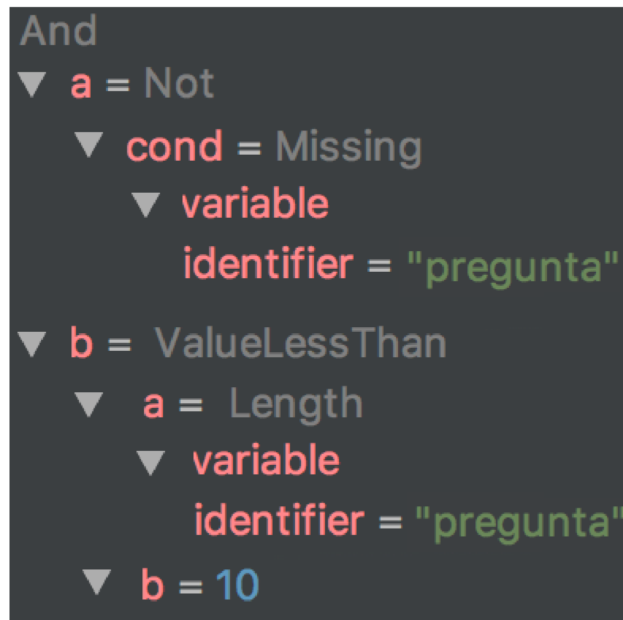


**Fig. 4.3:** *Abstract Syntax Tree*

Ahora bien, analicemos que ocurre con diferentes inputs para dichos programas.

Caso 1: {'pregunta': 'respuesta'}
La longitud de la respuesta es nueve (9), por lo tanto la respuesta que nuestro lenguaje debe dar es **VERDADERO**.

```
not missing context.get('pregunta') and length context.get('pregunta') < 10
not false and length 'respuesta' < 10
true and 9 < 10
true and true
true
```

Caso 2: {'pregunta': 'respuesta bien larga'}
En este caso la longitud de la cadena respondida es mayor que nueve, con lo cual la ejecución debe dar **FALSO**.

```
not missing context.get('pregunta') and length context.get('pregunta') < 10
not false and length 'respuesta bien larga' < 10
true and 20 < 10
true and false
false
```

Caso 3: {}
En este caso la variable no está definida, con lo cual la ejecución debe dar **FALSO**. Esto se debe a la lógica de cortocircuito que ocurre en los operadores binarios *or* y *and*.

```
not missing context.get('pregunta') and length context.get('pregunta') < 10
not true and length context.get('pregunta') < 10
false and length context.get('pregunta') < 10
false
```

## 4.2. Anonimizador + Estandarizador de código JavaScript

En el capítulo 2 se introduce un algoritmo para poder estandarizar programas JavaScript en pos de minimizar la cantidad de programas a analizar. Para la implementación del mismo se utilizó la biblioteca Babel de Node-JS que nos permite no solo parsear la entrada si no escribir traductores sobre el AST de manera sencilla y por último transformar ASTs nuevamente a código.

## 4.3. Clasificador de programas

En el capítulo 2 se menciona la creación de un programa para poder clasificar los códigos de manera sencilla. El algoritmo llevado a cabo en la práctica es el siguiente:

**for** *each file en un directorio* **do**

1. Mostrar su contenido por System-Out junto con todas las características frecuentes tipeadas hasta el momento
2. Leer de System-In la característica principal del programa
3. Mover dicho archivo a un subdirectorio con el nombre de la característica (creándolo si no existe)

**endfor**

**Algoritmo 1:** Clasificador de código

Para su implementación[20] se utilizó Python3 siendo este un script fácil de portar ya que no requiere de ninguna dependencia externa.

# 5. TRADUCCIÓN

A partir del corpus de programas escritos en JavaScript, queremos traducirlos a nuestro lenguaje preservando semántica.

Como JavaScript es un lenguaje de propósito general que es Turing complete y permite realizar muchas cosas más que el nuestro (como bien puede ser generar una conexión HTTP con otra computadora), y el nuestro uno de dominio específico el cual permite predicar sobre muy pocas cosas, sabemos que una biyección entre los lenguajes no será posible. Sin embargo, corremos con la ventaja de que los programas que queremos traducir no usan todo el poder expresivo que provee el lenguaje, con lo cual se establecerá una traducción entre un subconjunto del lenguaje JavaScript y el nuestro tratando de cubrir la mayor cantidad de casos que se pueda.

Nuestro lenguaje permite escribir una expresión booleana que asume como entrada un diccionario de variables. Lo mismo se espera de los programas escritos en JavaScript.

Para hacer la traducción de manera más sencilla y que no necesite del modelado de memoria para entender en qué momento de la traducción cada variable tiene qué valor, se asume que los programas se encuentran escritos de manera que existe *static single assignment*[7]. Esto significa que cada variable es asignada una única vez en el programa y cada vez que dicha variable se va a utilizar, ésta ha sido definida previamente. En caso de que el programa no se encontrara escrito de dicha forma, existen técnicas que permiten traducir programas a dichas formas. Éstas suelen ser utilizadas en su mayoría para permitir optimizar algoritmos de compilación como bien pueden ser la propagación de constantes por el código[8] o la eliminación de código muerto[9].

Otra cosa que asumiremos de los programas es que únicamente constan de una función que devuelve un valor booleano, y que será invocada con los parámetros necesarios que ésta defina. Es esta una restricción que nos beneficiará en la manera de generar la traducción ya que para nosotros una función solo será considerada de tipo booleano en cualquier caso. De lo contrario, habría que modelar la traducción de distintos tipos de funciones y esto sería más complejo. En caso de que el programa tuviera más funciones que la principal que queremos traducir, se resuelve utilizando el refactor[10] de tipo *inline* el cual nos permite reemplazar todas las llamadas a la función por el cuerpo de la misma. Es ésta la inversa del refactor de tipo *extract* que nos permite extraer una porción de código a una función aparte y nos permite reuso más fácilmente.

## 5.1. Branching

Nuestro lenguaje únicamente permite expresar un valor booleano a fin de cuentas.

Al desarrollar un programa en JavaScript, es posible escribir código que realice diferentes cálculos o acciones dependiendo de si una condición booleana especificada se evalúa como verdadera o falsa. Las dos estructuras más comunes para generarlo son *if - then - else* y *switch*. De esta manera, se pueden generar distintos valores de retorno para una función según la *branch del if* en la que se retorne la respuesta.

Como los programas a traducir devuelven un valor booleano, es posible reescribir el programa como una única expresión booleana. Hay que buscar una manera de poder escribirla basándose en todos los valores de retorno que ésta especifica y el estado general de

las variables qué hizo que se llegue a esa *branch* en la ejecución.

Vale aclarar nuevamente que al haber *static single assignment*, toda variable que sea considerada al momento de aplicar un condicional tendrá un único valor durante toda la ejecución. Luego, al pensar en una traducción del código, no será necesario contemplar en qué situación se llegó a un *return value* ya que esto no influirá en los valores de las variables.

Esto nos hace pensar en que una pasada lineal sobre el código será suficiente para traducirlo.

```
if (A) {
  if (B){
    return E;
  } else {
    return F;
  }
} else {
  if (C) {
    return G;
  }
}
return H;
```

Código 6

En el código 6 podemos ver un caso donde hay muchas ramas. A simple vista podemos decir que devolverá:

- **E** en caso de que **A** y **B** sean verdaderas.
- **F** en caso de que **A** sea verdadera y **B** falsa.
- **G** en caso de que **A** sea falsa y **C** verdadera.
- **H** en caso de que no haya entrado en ninguno de los anteriores return. Es éste caso un poco más complejo que los anteriores ya que no debió haber valido **A** o bien si no valió tampoco debió haber valido **C**.

Generalizar el pensamiento utilizado para el caso donde H vale es la mejor forma de encarar el problema.

Se puede decir que si se recorre el código línea por línea, cada vez que uno se encuentra con un *return statement* solo puedo haber llegado en la ejecución hasta ese lugar si ninguna de las condiciones que me llevaban a otros *return statements* anteriores fueron ciertas. Luego, estamos en condiciones de decir que ese return ocurre solo si los anteriores *returns* no ocurrieron y valen todas las condiciones necesarias para llegar a esa rama en particular.

```
// retVal = false;
if (A) {
  if (B){
    // Vale A & B (por estar adentro de los dos if)
    // RETVAL |= (A & B & E);
    return E;
  } else {
    // No vale A & B porque si no, ya no habría llegado a esta rama y
       vale A & !B (por estar en el if del A y en el else del B)
    // RETVAL |= (!(A&B)) & (A & !B & F)
```

```
      return F;
    }
  } else {
    if (C) {
      // No valen A&B ni A&!B porque si no, no habría llegado a esta rama y
           vale !A (por estar adentro del else) & C por estar dentro del if
      // RETVAL |= (!(A&B)) & (!(A&!B)) & (!A & C & G)
      return G;
    }
  }
  // No valen A&B ni A&!B ni !A&C por el mismo motivo anterior pero no
       estoy adentro de ningún if así que no se nada más respecto a ninguna
       variable
  // RETVAL |= (!(A&B)) & (!(A&!B)) & !(!A & C) & H
  return H;
```

Análisis sobre código 6

A fin de cuentas nos queda la expresión:
`(A & B & E) ||`
`(!(A & B) & (A & !B & F) ||`
`(!(A & B) & !(A & !B) & (!A & C & G)) ||`
`(!(A & B) & !(A & !B) & !(!A & C)) & H`

Si la pasamos luego por un simplificador de expresiones booleanas el valor final es:
`(A & B & E) || (A & !B & F) || (!A & C & G) || (!A & !C & H)`
que es básicamente lo que pudimos identificar al principio.

## 5.2. Definición de variables

Partiendo de la base que establecimos donde cada variable se asigna una única vez en toda la ejecución del programa, podemos decir que si una variable pertenece al diccionario de variables de entrada o es una definida por el usuario en el código, no habrá necesidad de distinguir una de otra en cuanto a las operaciones que pueden utilizarlas. Sin embargo existen algunas diferencias.

Por un lado, al momento de realizar la traducción, las variables del diccionario deben traducirse como variables del diccionario ya que dicho comportamiento es el esperado en nuestro lenguaje, mientras que las variables del código deben traducirse al contenido que se les es asignado ya que no existe la posibilidad de definir nuevas variables en nuestro lenguaje.

Por otro, la definición de una variable en el medio del código puede constar de una operación entre variables definidas previamente. Sin embargo este no es un problema, ya que a fin de cuentas, cada vez que las variables que dependen de otras sean referenciadas, establecimos que se reemplazan por la definición de las mismas.

Por lo tanto, en el proceso de la traducción, tener un diccionario global de variables con sus respectivos valores es suficiente para alcanzar nuestro objetivo. Tan solo será necesario al final del proceso reemplazar las variables que dependen de otras por la definición de las mismas de manera recursiva.

```
1  const edadBase = 38;
2  const maximoDeEdad = 45;
3  const suma = context.get("edadDeUsuario") + edadBase;
```

```
4   return suma < maximoDeEdad;
```

Código 7

En el código 7 podemos ver que existen por un lado variables definidas por el usuario, algunas que son constantes, otras que dependen de variables definidas previamente y variables que pertenecen al diccionario de entrada.

Si hacemos un paso a paso en el proceso de traducción podemos ver que:

1. En la línea 1 sabemos que existen todas las variables del diccionario y ahora `edadBase` que vale 38.
2. En la línea 2 ahora sabemos que también existe `maximoDeEdad` que vale 45.
3. En la línea 3 se define la variable suma que es la suma entre dos variables previamente definidas.
4. En la línea 4 se devuelve como resultado una operación de relación menor entre dos valores definidos previamente (`suma` y `maximoDeEdad`). Al realizar el reemplazo recursivo de los valores de las variables llegamos a que el resultado final es verdadero si y solo si `context.get("edadDeUsuario") + 38 < 45`

## 5.3. Operadores

Existen determinados operadores cuya traducción a nuestro lenguaje es bastante sencilla. Los operadores lógicos sobre elementos booleanos en general tienen una correspondencia uno a uno con nuestro lenguaje. Por ejemplo:

- *!x* se traduce a *not x*
- *x || y* se traduce a *x or y*
- *x && y* se traduce a *x and y*
- $x == y$ se traduce a $x = y$
- $x === y$ se traduce a $x = y$
- Todos los operadores de inecuaciones ($\leq, \geq, >, <$) y las operaciones matemáticas (+, -, *, /) se mantienen igual.
- La función *length* que se puede aplicar a listas (o strings) debe cambiar simplemente la sintaxis. En lugar de *lista.length*, en nuestro lenguaje se expresará como *length lista*.
- Ocurre similar con el operador *split*. En lugar de *lista.split(separator)*, en nuestro lenguaje se expresará como *split lista separator*.
- Para la aplicación de expresiones regulares tenemos varias opciones en JavaScript. En particular existe un método que nos permite determinar si un string matchea una expresión regular dada (*regexp.test(string)*), pero también existen otros donde se puede determinar que substrings matchean dentro del dado (*string.match(regexp)* o

*regexp.exec(string)*). Para simplificar asumiremos que en todos los casos se busca saber si la expresión regular matchea o no, con lo cual ante la aparición de cualquiera de estos elementos se traducirá a *string matches regexp* que devuelve un valor booleano. Como bien se explicará en el capítulo 6, en los programas que analizamos donde se utilizan las operaciones que devuelven la lista de apariciones, éstas se utilizan únicamente para saber si existe alguna ocurrencia o no. Esto hace que nuestra traducción sea correcta y nos evita la necesidad de extender el lenguaje con operaciones que nos parecen innecesarias.

- La construcción de un string basado en un número *new String(x)* se traduce a *as_string(x)*
- La construcción de un número basado en un string *parseInt(x)* se traduce a *as_integer(x)*
- La construcción de un número flotante basado en un string *parseFloat(x)* se traduce a *as_float(x)*
- *str.toUpperCase()* se traduce a *to_upper str*
- *str.toLowerCase()* se traduce a *to_lower str*
- *str.trim()* se traduce a *trim str*
- *isNaN(x)* se traduce a *not (x is_integer)*

# 6. RESULTADOS

## 6.1. Corpus de datos e invariantes

En el capítulo 5 donde se explica cuál es el método de traducción se plantean diferentes restricciones que los programas deben cumplir para que el proceso sea exitoso. Es por eso que se deben realizar diferentes búsquedas y manipulaciones del conjunto de entrada (en los casos que corresponda).

### Static Single assigment

Determinar si los programas cumplen con *static single assignment* resulta ser una tarea sencilla. En la sección 2 del trabajo se diseñó un algoritmo que permite normalizar los programas en pos de reducir la cantidad de programas a analizar. Como parte de dicho proceso, una de las etapas consiste en renombrar las variables utilizadas con un identificador único, el cual no había sido utilizado previamente. Por lo tanto, para determinar si un programa cumple con esta propiedad (*static single assignment*) basta con modificar levemente dicho algoritmo verificando que una variable sea renombrada una única vez. De no ser así, se puede decir que la propiedad no se cumple.

Al correr dicho algoritmo encontramos once (11) ejemplos que no lo cumplían. Al ser tan pocos los ejemplos que no lo cumplían se decidió arreglarlos manualmente para no descartar casos que podrían ser valiosos para la traducción misma.

```
f = function (x1, x2, x3) {
  if (x1 == 0 || x1 == 1) {
    if (x2 == 2) {
      var x4 = /r0/;
      var x5 = context.get("s0");
      if (x5 != 3) {
        x4 = /r1/;
      }
      if (!x4.exec(x3)) {
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  }
  return false;
};
f(context.get("s1"),context.get("s2
    "), context.get("s3"));
```

Código 8

```
f = function (x1, x2, x3) {
  if (x1 == 0 || x1 == 1) {
    if (x2 == 2) {
      var x5 = context.get("s0");
      if (x5 != 3 &&
          !/r1/.exec(x3)) {
        return true;
      } else if (x5 == 3 &&
          !/r0/.exec(x3)) {
        return true;
      } else {
        return false;
      }
    } else {
      return false;
    }
  }
  return false;
};
f(context.get("s1"),context.get("s2
    "), context.get("s3"));
```

Código 8 refactorizado

El código 8 es un ejemplo donde se determina el valor de una expresión regular a utilizar posteriormente basado en el valor de otra variable lo cual es equivalente a aplicar la expresión regular correspondiente según el valor de la otra variable.

### Expresiones regulares

Se plantea que la aplicación de expresiones regulares se traducirá a nuestro lenguaje de manera que se verifique si la expresión matchea o no a un string dado. Los casos de uso donde se plantea capturar todos los *substrings* que cumplan con la expresión regular son descartados.

Haciendo diferentes búsquedas en el corpus de datos se llegó a los siguientes resultados:

1. Existen cuarenta y cuatro (44) casos de uso donde se utiliza *regexp.test(string)*.

2. Existen setenta y nueve (79) casos donde se realiza *regexp.exec(string)*. Sin embargo en todos los casos luego se pregunta si el resultado es nulo o no, lo cual es equivalente a realizar un *regexp.test(string)* que devuelve si hubo o no matches. Un ejemplo de este comportamiento es el código 8.

3. Existe un (1) caso donde se realiza *string.match(regexp)*. En este caso nuevamente se consulta luego si el resultado es o no nulo.

En conclusión, podemos decir que no existe caso de prueba que no cumpla con las restricciones impuestas para que el algoritmo funcione.

### Una única función que devuelva un valor booleano

Otra de las restricciones impuestas fue la presencia de una única función booleana en los programas de entrada. Esto ocurrió en la mayoría de casos salvando algunas excepciones donde o bien había más de una función la cual terminó siendo *inlineada* en cada uno de sus usos o bien no existía función alguna y el valor de retorno estaba implícito sin *return* precedente. En estos últimos casos se debió encerrar toda la porción del código en una función que no reciba parámetros y hacer el valor de retorno explícito.

## 6.2. En la práctica

A continuación se presentan algunos programas que han sido correctamente traducidos de un lenguaje al otro y que representan casos de éxito relevantes.

El código 9 presenta tanto una cadena de condicionales como la aplicación de expresiones regulares mediante la sentencia `match` para identificar si parte de una determinada variable cumple con dos patrones diferentes. Se puede apreciar que la condición resultante en el DSL respeta la misma lógica: o bien al tomar el fragmento asociado cumple con el primer patrón (llamémosle fragmento 1) o bien cumple con el segundo (fragmento 2). Es interesante notar que la negación del fragmento 1 aparece repetido en el fragmento 2. La primera aparición se debe a la parte del algoritmo que descarta todos los posibles valores de retorno que pudieran haber ocurrido antes y la segunda se debe a que el valor de retorno del segundo fragmento ocurre en el contexto del `else` de un condicional: es por eso que el antecedente no debe cumplirse para poder ejecutar este fragmento del código.

```
var test = context.get("str0")
    ;
var x5 = test.split(" ");
var x8 = x5[0];
var x11 = /r0/;
var x12 = /r1/;
if (x11.exec(x8)) return true;
else if (x12.exec(x8)) return
    true;
else return false;
```

Código 9

```
((split context.get('str0') ' '
    ) [0] matches 'r0') or
(not((split context.get('str0')
    ' ') [0] matches 'r0') and
(not((split context.get('str0')
    ' ') [0] matches 'r0') and
((split context.get('str0') ' '
    ) [0] matches 'r1')))
```

Código 9 traducido

El código 10 presenta el chequeo de presencia de una variable o que no contenga contenido para ir por la negativa y en caso de existir que tenga contenido para ir por la positiva. La traducción únicamente presenta los casos por los cuales la condición es verdadera, con lo cual el único caso que se ve es aquel en el que la variable exista (o bien no ocurre que falte) y que la longitud asociada a la misma sea mayor a 0.

```
var test = context.get("str0");
if (test == null) return false;
var x6 = test.length;
if (x6 < 0) return true;
return false;
```

Código 10

```
not(context.get('str0') missing
    )
          and
length context.get('str0') < 0
```

Código 10 traducido

Sin embargo no todo programa ha sido traducido satisfactoriamente al DSL.

El código 11 es un ejemplo de un programa que no ha sido traducido al lenguaje. Presenta por un lado la operación de reemplazo dentro un string, la cual no es parte del DSL y se ha dejado de lado al no ser ésta una operación común dentro de las que existen. De caso de ser necesario, es posible dar soporte de dicha operación en el lenguaje y la traducción debería ser viable.

```
function f() {
  var x1 = new String(context.get("str0"));
  var x5 = context.get("str1");
  var x8 = x1.replace(/r0/g, "str2");
  var x10 = new Number(x8);
  var x12 = x10 > 14 && x10 < 18;
  if (x12 === false && x5 == 2) {
    return true;
  } else if (x12 !== false && x5 == 1) {
    return true;
  } else if (x1 != "str2" && x5 == 1) {
    return true;
  } else {
    return false;
  }
};
```

Código 11

El código 12 tampoco ha sido traducido al lenguaje. En este caso el programa toma la fecha actual del sistema y la compara con la fecha de una variable para determinar si es anterior o no. En el caso del DSL las fechas tampoco son soportadas. Es otro caso de uso que debería modelarse para que la traducción sea viable.

```
function f() {
  var x0 = context.get("str0");
  if (x0 == null || x0 == "") {
    return false;
  } else {
    var x3 = new Date(x0);
    var x8 = new Date();
    if (x3.getMillis() < x8.getMillis()) {
      return true;
    } else {
      return false;
    }
  }
};
```

Código 12

## 6.3. Resultados

En la tabla 6.1 se puede identificar el porcentaje de casos de éxito en las traducciones agrupados por los patrones frecuentes identificados en la sección 3 y los motivos por los cuales determinados programas no han podido ser traducidos. Se consideran casos de éxito a aquellos programas que pudieron ser traducidos (sin saber si la traducción fue correcta o no).

| Patrón | Porcentaje de casos exitosos | Operaciones que no permitieron traducción |
|---|---|---|
| *Anidamiento de condicionales* | 80 % | 1. Ternary operator ?:<br>2. Index of en Strings<br>3. Replace en Strings |
| *Aplicación de expresiones regulares sobre variables* | 91 % | 1. Concatenación de Strings<br>2. Replace en Strings<br>3. Or entre Strings |
| *Validaciones sobre la longitud de una variables* | 71 % | 1. Método toString() |
| *Contabilizar la cantidad de variables que cumplen con una propiedad* | 87.5 % | 1. For Loops |

**Tab. 6.1:** Porcentaje de casos que han podido ser traducidos al nuevo lenguaje

Resulta interesante destacar que algunos programas que entran en más de una categoría pero se han puesto en una particular de manera arbitraria para poder sacar estadísticas más fácilmente. Por otro lado, el porcentaje de programas traducidos total es del ochenta y seis (86) por ciento. Esto se debe a que la mayoría de los programas se concentran en las primeras dos categorías.

## 6.4. Correctitud

Probar la correctitud de la implementación de un algoritmo es una tarea compleja. Probar terminación de un programa no es posible (*halting problem*), luego comprobar que un programa es correcto en todas sus ejecuciones tampoco lo es. Es por eso que existen diferentes técnicas que dan una mayor aproximación a comprender la correctitud de las implementaciones de los algoritmos.

Existe, por ejemplo, un método de verificación formal que permite predicar sobre propiedades de un sistema a través de la inspección de los estados de un modelo. A esto se lo conoce como *Model Checking*. Una de las ventajas que provee dicho método es la oferta de contraejemplos donde dichas propiedades no se cumplen.

Otra opción que es ampliamente adoptada en la industria del software es testing. Esta técnica sugiere escribir casos de prueba para los que se conoce el resultado de manera previa y luego verificar que al correr el programa se obtiene el resultado esperado. De esta manera se pueden encontrar fallas en la implementación, más bien conocidas como *bugs*. Sin embargo, no se puede determinar a través del testing que en todos los casos el programa funcionará. Tampoco es una tarea sencilla escribir buenos casos de prueba. Una de las métricas más conocidas para identificar que tan buena es una suite de tests es la cobertura de código. A través de la instrumentación del código es posible saber por qué líneas del mismo se pasó al momento de ejecutarlos. A mayor cobertura del código podemos decir con mayor certeza habrá menos chance de fallos inesperados en la ejecución de las mismas. Nuevamente, nada asegura que haya otro caso de test que pase por las mismas líneas y falle.

### 6.4.1. Correctitud de nuestra implementación

Para poder entender si la implementación de la traducción es correcta se decidió separar el proceso en dos etapas.

La primera etapa consta en hacer un chequeo manual de las mismas. Esto se debe a que al ser la primera aproximación al uso de la herramienta era muy probable que haya errores de código que arreglar. Para poder hacer esta traducción tolerante a cambios se decidió construir una herramienta que ante la decisión humana de determinar que una traducción es correcta convertirla en un caso de test de regresión. Luego, ante cada cambio en la herramienta se puede volver a utilizar estos casos de éxito para verificar que no se introdujera ningún *bug*.

Sin embargo, la primera etapa no es suficiente. Lo único que se está comprobando es si a ojos de un humano las condiciones en un lenguaje y en el otro parecen ser semánticamente iguales.

Es por eso que es necesaria una segunda etapa donde se pueda poner a prueba los programas ante diferentes inputs y verificar que los resultados obtenidos en ambas ejecuciones son iguales. Escribir casos de prueba manualmente para cada uno de los programas es una tarea imposible de realizar. Recordemos que contamos con un corpus de cientos de programas. La solución encontrada a este problema fue aplicar una técnica de *fuzzing* para la generación automática de casos de prueba.

#### Fuzzing

Se denomina *fuzzing* a una técnica de generación de inputs de manera semiautomática para ser inyectados en programas y detectar *bugs* y vulnerabilidades. Es ésta una herramienta que tiene un enfoque bastante diferente a los métodos tradicionales ya que el programador no debe ser quien se encargue de testear sino es éste un proceso automático.

Basados en la definición de John Neystadt en "Automated Penetration Testing with White-Box Fuzzing" [11], existen distintas taxonomías para construir un algoritmo de fuzzing.

*Basado en la estructura del input*

- *Dumb fuzzing* (fuzzing bobo): las entradas generadas son totalmente aleatorias sin siquiera considerar los tipos de datos que se esperan. Son estos muy fáciles de implementar pero poco efectivos.

- *Smart fuzzing* (fuzzing inteligente): las entradas generadas tienen en cuenta el formato esperado, siendo más complejos de implementar pero más efectivos.

- *Mutation based fuzzing* (fuzzing basado en mutación): se basa en generar nuevas entradas con pequeñas diferencias en búsqueda de nuevos defectos.

- *Generation based fuzzing* (fuzzing basado en generación): las entradas no se basan en ejemplos anteriores, pero sí tratan de respetar la estructura de las mismas.

*Basado en la estructura del código*

- *White box fuzzing* (fuzzing de caja blanca): tiene la posibilidad de hacer uso del código que se está ejecutando para buscar errores.

- *Gray box fuzzing* (fuzzing de caja gris): no tiene la posibilidad de hacer uso del código que se está ejecutando para buscar errores, solamente a las interfaces que se desean testear.

- *Black box fuzzing* (fuzzing de caja negra): no tiene la posibilidad de hacer uso del código que se está ejecutando para buscar errores ni de las interfaces que se desean testear.

- *Code Coverage based fuzzing* (fuzzing basado en cobertura de código): utiliza la cobertura de código alcanzada hasta el momento para generar las nuevas entradas. De esta forma se pueden lograr alcanzar mejores tests.

*JsFuzz*[12] es una herramienta de fuzzing escrita en JavaScript basada en cobertura de código. Ésta fue la herramienta que se decidió utilizar para testear que las traducciones preserven semántica al ser ejecutadas.

Los resultados fueron satisfactorios: gracias a la herramienta se pudo encontrar pequeños errores que a simple vista no eran fácilmente identificables.

El código 13 fue un caso de ejemplo que nos dio resultados muy satisfactorios tras ejecutar fuzzing.

```
f = function() {
  var x1 = context.get("str0");
  if ((x1 == "str1" || parseFloat(x1) == parseInt(x1) && !isNaN(x1)) &&
      x1 >= 1 && x1 <= 2010) {
    return false;
  }
  else {
    return true;
  }
};
f();
```

Código 13

Por un lado se pudieron encontrar errores en la implementación del lenguaje. Por ejemplo $\geq$ estaba implementado como $>$ y $\leq$ como $<$. Por otro lado, en un principio se había decidido únicamente soportar en el lenguaje la posibilidad de *parsear* un texto e interpretarlo como entero pero no como número flotante. En caso de que un código intentase *parsear* a punto flotante, la traducción sería convertir a un número entero. Dicha decisión fue errónea, y fue el fuzzer quien nos encontró como caso de ejemplo el valor **7E2** que si se toma como entero luego el valor devuelto por *parseInt* es **7** mientras que si es tomado como número flotante luego el valor devuelto por *parseFloat* es **700**. Esto se debe a que según la definición de ECMAScript® [13] al *parsear* un entero a partir de un *String*, se interpreta únicamente la porción del principio que forma parte de un número entero, se ignoran todos los caracteres posteriores al primero que no es válido y no se le indica nada al usuario que dichos caracteres fueron ignorados.

```
f = function(x1, x2) {
  if (x1 == 1) {
    if (x2 != 1 && x2 != 2) {
      return true;
    } else {
      return false;
    }
  }
  return false;
};
f(context.get("str0"), context.get("str1"));
```

Código 14

Otro ejemplo captado gracias a esta técnica fue el código 14 donde se espera que una variable sea igual al valor 1. La comparación se hace a partir del operador `==` el cual hace coerción de tipos y por ejemplo hace que `‘‘1’’ == 1` de como resultado `true`. Sin embargo, en la implementación de la traducción, al encontrar tanto el operador `==` como `===` (el cual no hace coerción de tipos) son ambos traducidos al mismo elemento del lenguaje `EqualTo`, el cual utiliza el operador `===` en la implementación. El fuzzer necesitó de doscientos quince (215) iteraciones hasta encontrar la diferencia entre una implementación y otra generando el caso de prueba `x1 = ‘‘1’’, x2 = ‘‘A6’’`. A partir de dicho momento, varios casos similares fueron captados por la herramienta todos siguiendo el mismo patrón.

Cabe destacar que la herramienta comienza sin un corpus de test cases, con lo cual comienza a generarlos desde cero (*Generation based fuzzing*). Sin embargo, con el correr de las iteraciones va guardando en su base de datos casos interesantes para generar nuevos a partir de la mutación de los casos previos (*Mutation based fuzzing*). Por lo tanto, repetir un experimento para encontrar el mismo bug no sirve sin borrar la base de datos original, pues los casos de prueba que generaron fallas son tomados como base para la búsqueda. Es así, que al probar repetir un experimento, tan solo 10 iteraciones bastaron para encontrar un bug que había llevado miles de iteraciones en la pasada original.

También cabe destacar que al tratar de reproducir un experimento, puede ocurrir que un bug encontrado rápidamente en la primera ocasión, no es encontrado rápidamente. Esto se debe a que la herramienta no es determinística en los caminos que usa para explorar y explotar la cobertura de código. Por ejemplo, para encontrar el error de la implementación de *parseInt* igual a *parseFloat* se volvió a correr el experimento sin un corpus de datos. Dos corridas fueron necesarias para poder encontrarlo: la primera fue abortada tras no ser identificada en más de cien mil (100.000) iteraciones, y en la segunda tras seiscientas (600)

iteraciones el bug fue descubierto con un caso de prueba diferente: un número con coma (1.9) en lugar de un número escrito con notación científica (7E2).

### Detalles de la implementación

Como bien se explicó anteriormente se utilizó como base la herramienta *JsFuzz*, la cual está disponible como biblioteca para varios lenguajes. En nuestro caso fue utilizada en su versión JavaScript.

Su funcionamiento es muy sencillo: llama indefinidamente a un método especificado por el usuario con un stream de datos differente hasta que se corte manualmente la ejecución o un error aparezca.

Para nuestro caso particular, es necesario partir el stream en tantos valores como variables de contexto involucradas en el programa a testear existan. Una vez resuelto ese problema, es decir que las variables de contexto tengan valor, se corre tanto el programa original como el programa traducido a nuestro lenguaje y se falla en caso de que las respuestas difieran.

Como nota de color, cabe destacar que para saber cuáles variables de contexto son necesarias llenar para cada programa, fue necesaria la construcción de otra herramienta automática que lo calculase. Este programa, similar al estandarizador de código, traduce el programa de entrada a una lista de variables de contexto. Dicha lista es luego usada como parte del input necesario para correr el fuzzer.

```
Function test(stream de datos, programa Javascript, programa traducido,
 variables de contexto):
    contexto ← Partir el stream de datos en las variables del programa a testear
      basada en variablesUtilizadas
    a ← Correr programa JavaScript con contexto como input
    b ← Correr programa en Lenguaje con contexto como input
    return a = b
Function Main(programa Javascript, programa traducido, variables de contexto):
            // La biblioteca nos provee una función que genera el stream de datos.
    while test(generateStream(), programa Javascript, programa traducido,
     variables de contexto) do
            // Mientras que no de false significa que los outputs de ambos programas
         siguen están dando igual.
    end
    return false;
return
```

**Algoritmo 2:** Fuzzer

### Resultados en números

Se utilizaron diez (10) programas representativos dentro del corpus y con mayor complejidad para poder identificar bugs en la implementación. La herramienta fue capaz de identificar seis (6) inputs para los cuales la salida del programa original y el programa traducido diferían. De los seis inputs:

- Dos (2) son considerados *bugs* de implementación en la semántica de los elementos del AST.

- Tres (3) son considerados errores en la definición de la traducción que posteriormente fueron corregidos para la definición final del trabajo.
- Uno (1) es considerado una diferencia esperada: la no distinción entre == y === en nuestro lenguaje que sí difiere en JavaScript.

Ninguna diferencia encontrada fue debido a un posible error en la definición del algoritmo principal de traducción (ni en cuanto al manejo de ramas o de variables definidas en el cuerpo del programa).

# 7. CONCLUSIONES

## 7.1. Trabajo relacionado

Se presentan a continuación distintos trabajos relacionados que han sido de interés a lo largo del desarrollo del trabajo y han servido o bien como soporte o como herramientas alternativas no exploradas que bien podrían ser probadas a futuro.

### 7.1.1. Equivalencia entre dos códigos diferentes

En el comienzo del trabajo se realizaron distintas técnicas para poder identificar cuando dos programas son equivalentes a pesar de tener diferencias sintácticas. Se logró un gran avance debido a que se redujo la cantidad de programas diferentes considerablemente. Sin embargo, existen otras técnicas basadas en diferencias semánticas que no fueron abordadas.

Por ejemplo la herramienta *semantic* [14] permite analizar dos códigos fuentes diferentes y encontrar las diferencias semánticas entre uno y otro. Para ello, genera árboles basados en el código los cuales son estandarizados en una sintaxis generalizada y luego ejecuta algoritmos para identificar las diferencias sobre dichos árboles. Las bases en las que reposan dichos algoritmos son ideas planteadas por diferentes trabajos de investigación entre los que se destacan: *RWS-Diff* una técnica para poder identificar diferencias entre árboles de manera aproximada [15]; el algoritmo de Myers [16] en el que se demuestra buscar el camino más corto de llegar de un árbol a otro es equivalente a encontrar la subsecuencia común más larga entre dos strings dados; entre otros. Es esta una herramienta que podría haber sido interesante para utilizar en el trabajo, pero no ha sido utilizada debido a los resultados positivos obtenidos con las herramientas desarrolladas.

También cabe destacar *SymDiff* [17]: una herramienta agnóstica a lenguajes que chequea equivalencias y muestra diferencias semánticas (de comportamiento) sobre programas imperativos. Como ventaja frente a otras herramientas, genera una separación de responsabilidades - se basa en el análisis del núcleo de los algoritmos, los cuales son independientes del lenguaje en que están escritos.

### 7.1.2. Verificación de compilación de código de un lenguaje a otro

Existe una rama de la ciencia de la computación dedicada a verificar la correctitud de la implementación de compiladores basada en la especificación de los lenguajes. En particular, existen dos enfoques que se basan en las dos técnicas mencionadas anteriormente: métodos formales de verificación y testing.

Dentro de los métodos de verificación formal, uno de los trabajos conocidos es el de Xavier Leroy [18] quien construyó una herramienta que permite verificar a través de un asistente de pruebas formales que la implementación del compilador CompCert (que compila un subconjunto grande del lenguaje C) es correcta. Esta herramienta garantiza que las propiedades de *safetiness* probadas en el código fuente también se cumplen en el código compilado.

Por otro lado, dentro de los métodos de testing podemos encontrar el caso de Pierre-Loïc Garoche et al. [19] en el que se centran en generar casos de test a partir de la extensión del compilador para que así se pueda testear el mismo con alta cobertura de código y así

descubrir las fallas en el mismo. Dicha implementación fue realizada en un compilador que traduce código Lustre a C.

## 7.2. Conclusiones

La herramienta utilizada en el análisis del trabajo presentaba como base un lenguaje de propósito general muy flexible para poder expresar condiciones. Sin embargo, como se planteó en el análisis, este lenguaje es difícil de comprender para un usuario inexperto, además de propenso a errores. Es por ello que invertir tiempo en construir un lenguaje de condiciones específico que sea lo suficientemente flexible para que un usuario pueda expresar lo que necesita, y que al mismo tiempo sea sencillo de utilizar, vale la pena.

Consideramos que el poder expresivo con el que se construyó el lenguaje es suficiente para un usuario estándar y a su vez es utilizable en otros contextos que no sean la aplicación que utilizamos como base. Como resultado se espera que la cantidad de errores que pueden cometer los usuarios al escribir en este lenguaje sea significativamente menor, y por otro lado se reducen las chances de que se escriba un programa que no termine al no incluir *loops* en el lenguaje. El tiempo necesario que un usuario necesita para aprender dicho lenguaje se reduce significativamente y más aún si se presenta con una interfaz gráfica que ayude a escribir en el mismo. Es esto posible ya que la estructura de las condiciones es estructurada y fácil de mostrar en forma de árbol.

Enfrentarse al desafío de construir un traductor de lenguajes fue una tarea muy interesante. Tanto la complejidad que conlleva la definición teórica del mismo y la demostración de su correctitud como llevarlo a la práctica en un lenguaje de programación fueron tareas que valieron su fruto: más del 80 % de los programas pudieron ser traducidos.

Nuestra traducción utiliza un subconjunto de todo el poder expresivo de JavaScript y es por eso que muchas estructuras del lenguaje quedan afuera. Sin embargo, hay algunas que podrían ser soportadas por nuestra herramienta de traducción que no lo son porque con la traducción existente se lograron capturar un gran porcentaje de casos de prueba. Por ejemplo, iteración de listas para determinar si un elemento cumple con una propiedad es algo que en nuestro lenguaje podemos alcanzar a través del uso del operador *ANY*, pero no podemos traducir automáticamente. Lo mismo con los operadores de conteo *AT LEAST*, *AT MOST*, etcétera. Sería esta una tarea interesante como para continuar con el desarrollo del traductor.

Aplicar *fuzzing* para evaluar correctitud de la implementación dejó una gran impresión sobre el poder que tiene esta técnica. En particular, que la herramienta utilizada se base en en cobertura de código, colaboró en poder cubrir tanto todas las ramas del código base como del traducido haciendo que se encontraran casos donde una rama en un caso dé un valor y en el otro código de otro resultado. Utilizar un *fuzzer* a la hora de testear implementaciones que pueden tener una gran diversidad de inputs o bien donde se desconoce los detalles de implementación parece ser una gran alternativa.

### 7.2.1. Trabajo futuro

A continuación, una serie de aspectos que son interesantes para desarrollar en el futuro

1. Extender el lenguaje con un mayor poder expresivo, agregando conceptos de dominios específicos, tales como validaciones sobre números de teléfono, códigos de países, entre otros, los cuales son frecuentes en diversos escenarios.

2. Extender el traductor de JavaScript para soporte de listas, objetos, etc.

3. Aplicar el traductor sobre los programas escritos para la herramienta que se hizo análisis en producción. Una manera de probarlo, antes de generar una migración total de manera más segura, es a partir de correr tanto el programa original como la traducción en producción durante varios meses y comparar la salida de ambos programas.

4. Construir una interfaz gráfica que permita construir expresiones válidas del lenguaje.

## Bibliografía


[1] Martin Fowler - Domain-Specific Languages.

[2] Uwe Zdun, Mark Strembeck - Reusable architectural decisions for DSL design: Foundational decisions in DSL development (2009)

[3] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Survey 37(4), 316–344 (2005)

[4] ShapingJiang, J., Xiong, Y., Zhang, H., Gao, Q., Chen, X.: Shaping Program Repair Space with Existing Patches and Similar Code. 10.1145/3213846.3213871. (ISSTA 2018)

[5] Alfred Aho & Jeffrey Ullman - The Theory of Parsing, Translation, and Compiling (Volume I: Parsing)

[6] ANTLR (ANother Tool for Language Recognition) - https://www.antlr.org/

[7] Barry Rosen, Mark N. Wegman, F. Kenneth Zadeck - Global Value Numbers and Redundant Computations (1988)

[8] Mark N. Wegman, F. Kenneth Zadeck - Constant Propagation With Conditional Branches (1991)

[9] Ron K. Cytron, Jeanne Ferrante, Barry K. Rosen, F. Kenneth Zadeck - Efficiently Computing Static Single Assignment Form and the Program Dependence Graph (1991)

[10] Martin Fowler - Refactoring.

[11] John Neystadt - Automated Penetration Testing with White-Box Fuzzing (2008)

[12] Coverage guided fuzz testing for JavaScript - https://github.com/fuzzitdev/jsfuzz

[13] ECMAScript® 2019 Language Specification https://www.ecma-international.org/ecma-262/

[14] Semantic - https://github.com/github/semantic

[15] J. Finis, M. Raiber, N. Augsten, R. Brunel, A. Kemper, F. Färber - RWS-Diff: Flexible and Efficient Change Detection in Hierarchical Data

[16] E. W. Myers - An O(ND) Difference Algorithm and Its Variations

[17] Lahiri, Shuvendu and Hawblitzel, Chris and Kawaguchi, Ming and Rebêlo, Henrique - SymDiff : A language-agnostic semantic diff toolfor imperative programs (2012)

[18] Xavier Leroy - Formal verification of a realistic compiler

[19] Pierre-Loïc Garoche, F. Howar, T. Kashai, X. Thirioux - Testing-Based Compiler Validation for Synchronous Languages

[20] Categorization - https://github.com/LucasBek/categorization