PrintScript

Trabajo Práctico - Primera parte

Desarrollo

En este trabajo práctico, se espera que diseñen e implementen tres herramientas fundamentales para un nuevo lenguaje de programación llamado "PrintScript". Este lenguaje tiene como objetivo principal facilitar la impresión y el formateo de texto y números.

Las herramientas que se deben desarrollar son:

1. **“Interpreter”**: un intérprete para este lenguaje que pueda ejecutar programas escritos en este lenguaje. El intérprete debe ser capaz de leer el código fuente de un programa, analizarlo y ejecutarlo.
2. **“Formatter”**: una herramienta de formateo para este lenguaje que pueda tomar como entrada un programa escrito en este lenguaje y formatearlo de acuerdo a un estilo específico determinado por una serie de criterios. El formateador debe ser capaz de aplicar indentación, espacios en blanco, saltos de línea, etc. de manera consistente en todo el programa.
3. **“Static code analyzer”**: un analizador estático de código para el lenguaje "PrintScript" que pueda detectar errores y posibles problemas en el código fuente. El analizador debe ser capaz de identificar errores de sintaxis, uso incorrecto de variables, funciones o estructuras de control, entre otros.

La intención es lograr que estas herramientas sirvan como base de un ecosistema alrededor de dicho lenguaje en los próximos años. Esto quiere decir que no solo debe poder ejecutar el código sino proveer una interfaz para analizar sintácticamente y de esta manera poder luego operar sobre el “Abstract syntax tree” (AST) [1] resultante. Este proceso se conoce como “Parsing”[2] y consiste en analizar la estructura del programa.

Para poder desarrollar el analizador sintáctico se debe contar previamente con un analizador léxico o “Lexer”[3]. El análisis léxico se refiere a la identificación de los elementos léxicos del lenguaje, como palabras clave, identificadores, operadores, etc.

Al realizar los distintos componentes del sistema se debe tener en cuenta que el lenguaje irá creciendo, a lo largo del tiempo, en complejidad y funcionalidad. Por lo tanto los mismo

[1] https://en.wikipedia.org/wiki/Abstract_syntax_tree
[2] https://en.wikipedia.org/wiki/Parsing#Computer_languages
[3] https://en.wikipedia.org/wiki/Lexical_analysis

deberán ser lo suficientemente flexibles para poder incorporar dicha funcionalidad con el menor esfuerzo posible.

También se debe tener en cuenta que los códigos fuentes pueden ser tan extensos que no es posible contenerlos de forma completa en memoria. Por lo tanto los distintos componentes del sistema deben soportar mecanismos que permitan manejar este tipo de fuentes, entendiendo que la información entre los mismos va fluyendo. Esto quiere decir que no se puede primero hacer todo el análisis léxico para luego pasar al análisis semántico ya que el primero consumiría toda la memoria disponible.

Lenguaje: PrintScript 1.0

Este lenguaje es un subset de TypeScript[4].
A continuación se detalla que partes del lenguaje incluye:

- Declaración de variables
  - Variables con el keyword “let”
  - Sin inferencia de tipos, es decir se debe aclarar el tipo de la variable (“let x: number;”).
  - Se puede declarar sólo una variable por sentencia.
  - Se puede declarar y asignar un valor en una misma sentencia.
- Asignación de variables ya declaradas (“x = 5”).
- Soporte solo de los siguientes tipos básicos:
  - “number”.
  - “string”.
- “string”: puede declararse con comillas simples o dobles (“ o ‘).
- Se debe terminar todas las sentencias con “;”.
- Las variables y literales de tipo “number” deben soportar operaciones aritméticas binarias:
  - Suma.
  - Resta.
  - Multiplication.
  - Division.
- El tipo “number” incluye enteros y decimales.
- Concatenación de variables y literales de tipo “string”.

[4] http://typescriptlang.org

    - Con el símbolo “+”
    - Esto incluye la concatenación de “string” y “number”, si la expresión incluye ambos tipos el resultado es “string”.
- PrintLn: Una expresión que no pertenece a TypeScript, que permite imprimir valores. Su sintaxis es equivalente a una función de nombre “println” en TypeScript. Esta función de impresión recibe como argumento una expresión de cualquier tipo básico (“string” o “number”), imprime su valor a la consola junto con un salto de línea. Un ejemplo podría ser “println(168)” o “println(a)”.

Ejemplos

Ejemplo 1

Codigo:

```
let name: string = "Joe";
let lastName: string = "Doe";

println(name + " " + lastName);
```

Salida esperada del “Interpreter”:

```
Joe Doe
```

Ejemplo 2

Codigo:

```
let a: number = 12;
let b: number = 4;
let c: number = a / b;

println("Result: " + c);
```

Salida esperada del “Interpreter”:

```
Result: 3
```

Ejemplo 3

Codigo:

```
let a: number = 12;
let b: number = 4;
a = a / b;

println("Result: " + a);
```

Salida esperada del “Interpreter”:

```
Result: 3
```

Formatter

Esta herramienta tiene el objetivo de dar un formato uniforme al código en base a algunas reglas. Estas pueden ser, por ejemplo, si hay espacios o no después de un identificador o antes de un tipo. Dichas reglas podrán poder configurarse desde un YAML o JSON que definirá si están prendidas o apagadas y con qué valores lo están.

Las reglas que se deben soportar para este trabajo son:

- Espacio antes de los “:” en una declaración. Puede haber o no espacio.
- Espacio después de los “:” en una declaración. Puede haber o no espacio.
- Espacio antes y después del “=” en una asignación. Puede haber o no espacio.
- Salto de línea antes de un llamado a “println”. Puede haber 0, 1 o 2 espacios.
- Debe haber un salto de línea a continuación de un “;”. No se configura.
- Debe haber un solo espacio, como máximo, entre los distintos “tokens”. No se configura.
- Debe haber un espacio antes y después de un operador sin importar la expresión. No se configura.

Static code analyzer

Esta herramienta tiene el objetivo de notificar del incumplimiento de algunas reglas en el código, ya sea para imponer convenciones como para detectar malas prácticas. Es importante no solo notificar del incumplimiento sino también de la posición exacta del mismo dentro del código, igual que se hace para los errores. Al igual que el “Formatter” esta herramienta se podrá configurar usando un YAML o JSON.

Las reglas que se deben soportar para este trabajo son:

- Los identificadores deben tener un formato determinado. Puede ser “camel case” o “snake case”.
- La función “println” solo puede llamarse con un identificador o un literal pero no con una expresión. Esto puede estar prendido o apagado.

CLI

Estas herramientas deben poder ejecutarse desde una “Command Line Interface” (CLI). Esta interfaz debe permitir ejecutar la tarea correspondiente sobre un archivo que esté en el file system del usuario. También debe poder correr en un modo que solo valide la sintaxis y semántica del archivo. Como argumentos deberían incluir:

- La operación a realizar: “Validation”, “Execution”, “Formatting” o "Analyzing”.
- El archivo fuente.
- La versión del archivo a interpretar. Opcional. Por ahora solo 1.0.
- Los argumentos especificados de cada operación, por ejemplo el archivo de configuración para realizar el “Formatting”.

En caso de que se encuentre errores, ya sean de sintácticos o semánticos, se deberá mostrar al usuario un mensaje describiendo el error así como su ubicación en el archivo. La posición deberá incluir la columna y fila de inicio y fin del problema.

Para hacer más cómoda la experiencia del usuario del CLI, mientras se realiza el proceso de “parsing” del archivo se debe ir mostrando en pantalla el progreso.

Requisitos y restricciones

- El sistema que se desarrolle debe estar separado en una serie de módulos que dependen unos de otros según sea necesario.
- La implementación puede ser en Java, Kotlin o Scala. También puede ser una mezcla de los mismos, pero se sugiere no mezclar Scala con los otros lenguajes.
- La implementación debe contar con test automáticos y escalables que prueben el correcto funcionamiento de los distintos componentes.
- A lo largo del tiempo que dura el desarrollo de este trabajo se irán introduciendo en la materia una serie de prácticas y herramientas que deben ser aplicadas al mismo. Para cada una se darán los detalles correspondientes sobre su integración.

- Se deben tener en cuenta los criterios aprendidos en Diseño de Sistemas. (Cohesión y acoplamiento, principios de SOLID, uso de patrones cuando corresponda, entre otros)

Entregables

- Diagrama de componentes del sistema.
- Una explicación de la responsabilidad que tiene cada módulo y cómo interactúan entre ellos.
- El código fuente que cumpla con los requisitos del trabajo. Algunos que se definirán en las sucesivas clases.
- Tests automáticos para cada componente del sistema.