Universidad ORT Uruguay

Facultad de Ingeniería

Generador de parsers en Scala para BNFC

Un generador de analizadores sintácticos en Scala utilizando Backus-Naur Form Converter como base

Entregado como requisito para la obtención del título Ingeniero en Sistemas

Juan Pablo Poittevin - 169766

Guillermo Poladura - 237268

Tutor: Álvaro Tasistro

2025

Declaratoria de Autoría

Nosotros, Juan Pablo Poittevin, y Guillermo Poladura, declaramos que el trabajo que se presenta en esa obra es de nuestra propia mano. Podemos asegurar que:

- La obra fue producida en su totalidad mientras realizábamos nuestra tesis de grado, para la carrera de Ingeniería en Sistemas.
- Cuando hemos consultado el trabajo publicado por otros, lo hemos atribuido con claridad;
- Cuando hemos citado obras de otros, hemos indicado las fuentes. Con excepción de estas citas, la obra es enteramente nuestra;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, hemos explicado claramente qué fue contribuido por otros, y qué fue contribuido por nosotros;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto donde se han realizado las aclaraciones correspondientes.

Juan Pablo Poittevin

20/03/2025

Guillermo Poladura

20/03/2025

Dedicatoria

A toda mi familia, especialmente mis abuelos que ya no están,
por haberme enseñado el valor de la educación, el esfuerzo, y la constancia.

Guillermo Poladura Pignata

Agradecimientos

A mi hermano mayor Ricardo, con quien decidí y comencé a estudiar esta carrera, y a Alejandro, quien nos acompañó poco después, por ser un gran apoyo y motivación a lo largo de toda la formación, y por compartir conmigo interminables conversaciones de ingeniería.

A mis amigos y compañeros, especialmente a Ari y Juan Pablo (mi compañero de tesis), por acompañarme, apoyarme y ayudarme siempre que lo necesité, haciendo que este camino recorrido fuera mucho más liviano y disfrutable.

A los excelentes profesores que tuve durante la carrera, por ser piezas fundamentales en mi formación académica y por su invaluable aporte a mi crecimiento profesional.

Guillermo Poladura Pignata

A nuestro tutor Álvaro Tasistro, por su gran ayuda, y disposición.

Abstract

Nuestra tesis consiste en la creación de un generador de analizadores sintácticos (*parser generator*) en *Scala* (*Scalable language*). Con este fin, nos basamos en un proyecto ya existente llamado *BNFC* (*Backus Naur Form Converter*), que consiste en un *parser generator* en otros lenguajes como *Haskell*, *Java* y *C*, entre otros.

El *Parser* es el componente que se encarga de llevar a cabo el análisis sintáctico, segundo paso dentro de las etapas de compilación (siendo la primera primera el análisis léxico, efectuado por el *Lexer*).

BNFC genera además un *Abstract Syntax Tree* (*AST*) de cada lenguaje, por lo que, si debemos ser más estrictos en la definición de nuestro proyecto, lo que buscamos es desarrollar *lexer generator*, *parser generator* y un *AST*. De todas formas, dado que no se puede crear un *parser* sin un *lexer*, y que *BNFC* siempre genera un *AST* de cada lenguaje, decir *parser generator* es suficiente.

Para generar un *Parser*, independientemente del lenguaje objetivo, debemos comenzar desde una gramática, la cual puede ser escrita de distintas formas, en este caso en *LBNF* (*Labelled BNF*). El objetivo final entonces es, tomar una gramática en *LBNF* y generar automáticamente un *parser* de dicha gramática, en *Scala*.

Como solución, hemos diseñado e implementado un nuevo módulo (como una carpeta más de *backends* dentro de la estructura del proyecto) para *BNFC* en *Scala*, extendiendo así la funcionalidad existente del mismo. Este módulo permite tomar especificaciones gramaticales escritas en *LBNF*, y generar automáticamente *AST*, *lexer*, y *parser* en código *Scala* funcional.

Palabras Clave

- Compiladores
- Lexer
- Parser
- Abstract Syntax Tree
- Scala
- Haskell
- BNF
- BNFC

Índice

Introducción

1.1 Motivación

1.1.1 El curso de Lenguajes de Programación

En cierto punto de nuestra carrera de Ingeniería de Sistemas, nos encontramos cursando una materia perteneciente al área de Teoría de la Computación denominada ”Lenguajes de Programación”[1]. En ésta abordamos, entre otras cosas, la manera de definir formalmente un lenguaje de programación, y el funcionamiento general de un compilador, elemento esencial para la traducción del código fuente escrito en un lenguaje de programación de alto nivel a instrucciones ejecutables por una máquina.

A lo largo del curso pudimos aprender sobre las distintas etapas formales de un compilador (*Lexer*, *Parser*, *Type Checker*, y *Code Generator*) y sus responsabilidades en el proceso de compilación, además de los distintos tipos de errores que pueden surgir en cada una de ellas.

La metodología del curso combinaba fundamentos teóricos con aplicaciones prácticas, permitiéndonos no solo comprender los conceptos abstractos detrás de la definición de un lenguaje, y del funcionamiento de los compiladores, sino también experimentar sobre éstos de manera progresiva, a través de tres laboratorios.

Para la definición formal de un lenguaje, utilizamos la denominada *Backus-Naur Form* (BNF), una notación formal utilizada para describir la sintaxis de los lenguajes, definiendo gramáticas libres de contexto.

El objetivo del primer laboratorio era definir una gramática libre de contexto para un subconjunto del lenguaje C++. Para lograrlo, nos apoyamos en la herramienta BNFC (*Backus-Naur Form Converter*), que permite generar *Lexers* y *Parsers* para nuestro lenguaje definido (inde-

[1] Actualmente reemplazada por la materia ”Lenguajes y compiladores”.

pendientemente de cuál fuera), con el fin de poder compilarlo y ejecutarlo. Ésto nos permitía comprobar su correcto funcionamiento o identificar cualquier falencia en su definición.

1.1.2 BNFC

Organización interna de BNFC

El proyecto de *BNFC* es de código abierto, por lo que tenemos acceso completo a su código fuente. Esto sumado al hecho que el libro de referencia *Implementing Programming Languages* de Aarne Ranta [1] utiliza *BNFC* en todos sus ejemplos, nos ayuda a entender cómo funciona *BNFC* internamente.

Es importante mencionar que *BNFC* en sí mismo está escrito en *Haskell*, por lo que en el desarrollo de este trabajo, utilizaremos de forma regular la sintaxis de *Haskell*, así como la de *Scala*, para explicar los conceptos relevantes y mostrar la implementación del nuevo *backend*.

Link al *github* de *BNFC*: Github-BNFC

Lo primero que debemos entender es la estructura que sigue *BNFC* internamente.

```
bnfc
├── docs
│   └── dev
├── document
│   ├── ...
    Esto es una lista de ejemplos de gramáticas en LBNF
├── examples
│   ├── Alfa
│   ├── C
│   ├── Calc Calc es el ejemplo mas básico posible
             Útil para estudiar el funcionamiento de BNFC
```

```
│   ├── GF
│   ├── Java
│   ├── Javalette
│   ├── LBNF
│   ├── OCL
│   ├── cpp
│   ├── cubicaltt
│   ├── define
│   ├── haskell-core
│   └── prolog
├── source
│   ├── license-report
│   ├── main
│   │   ├── Main.hs Como el nombre indica
                    éste es el ejecutable principal de BNFC
│   ├── src
│   │   ├── BNFC
│   │   │   ├── Backend Aquí se encuentran los distintos lengaujes
                        aceptados por BNFC, y es donde debemos
                        agregar Scala y toda nuestra lógica
│   │   │   │   ├── C
│   │   │   │   ├── CPP
│   │   │   │   │   ├── NoSTL
│   │   │   │   │   └── STL
│   │   │   │   ├── Common
│   │   │   │   ├── Haskell
│   │   │   │   ├── HaskellGADT
│   │   │   │   ├── Java
│   │   │   │   ├── OCaml
│   │   │   │   ├── Scala
```

```
│   │   │   │   └── TreeSitter
                    Aquí también encontramos los puntos de entrada
                    a los distintos backends (archivos .hs)
                    Main.hs invoca a estos .hs (dependiendo el lenguaje)
                    y éstos pueden resolver todo en un solo archivo, u
                    organizarse en una estructura interna de Backend
│   │   │   │   ├── Agda.hs
│   │   │   │   ├── Base.hs
│   │   │   │   ├── C.hs
│   │   │   │   ├── Haskell.hs
│   │   │   │   ├── HaskellGADT.hs
│   │   │   │   ├── Java.hs
│   │   │   │   ├── Latex.hs
│   │   │   │   ├── OCaml.hs
│   │   │   │   ├── Pygments.hs
│   │   │   │   ├── TreeSitter.hs
│   │   │   │   ├── Txt2Tag.hs
│   │   │   │   └── XML.hs
│   │   │   ├── CF.hs CF.hs tiene la definición interna de ContextFree
                          esta clase es la que reciben los distintos
                          Backends y representa el LBNF
                          ingresado por el usuario
│   │   │   ├── Abs.hs
│   │   │   ├── Check
│   │   │   │   └── EmptyTypes.hs
│   │   │   ├── ErrM.hs
│   │   │   ├── GetCF.hs
│   │   │   ├── Lex.x
│   │   │   ├── Lexing.hs
│   │   │   ├── License.hs
```

```
│   │   │   ├── Options.hs
│   │   │   ├── Par.y
│   │   │   ├── PrettyPrint.hs funcionalidades auxiliares
│   │   │                      de formateo y resaltado de sintaxis,
│   │   │                      útiles a la hora de generar un
│   │   │                      archivo fácilmente legible
│   │   │   ├── Print.hs
│   │   │   ├── Regex.hs
│   │   │   ├── TypeChecker.hs
│   │   │   ├── Utils.hs
│   │   │   └── regexMinus.hs
│   │   └── layout

│   └── test
│       └── BNFC
│           └── Backend
│               ├── CPP
│               ├── Common
│               └── Haskell
└── testing
    ├── ...
```

Dentro de toda esta estructura hemos resaltado lo más importante, en particular el *Main.hs*, punto de entrada a *BNFC* y principal ejecutable.

Los distintos lenguajes soportados por *BNFC* se organizan dentro del directorio *Backends*, contando cada uno con un archivo *.hs* principal, usado como punto de entrada por *BNFC*, en donde nosotros podemos decidir implementar toda la lógica (como lo hace, por ejemplo, *Pygments*[2]), o dividirlo en múltiples módulos (como se observa con *Java*), a organizar dentro de un

[2]Un resaltador de sintaxis de código fuente de propósito general escrito en *Python*. A diferencia de otros *back-*

directorio con el mismo nombre que nuestro *Backend*.

Para que el flujo de datos dentro de *BNFC* sea lo mas claro posible, tenemos el siguiente diagrama:

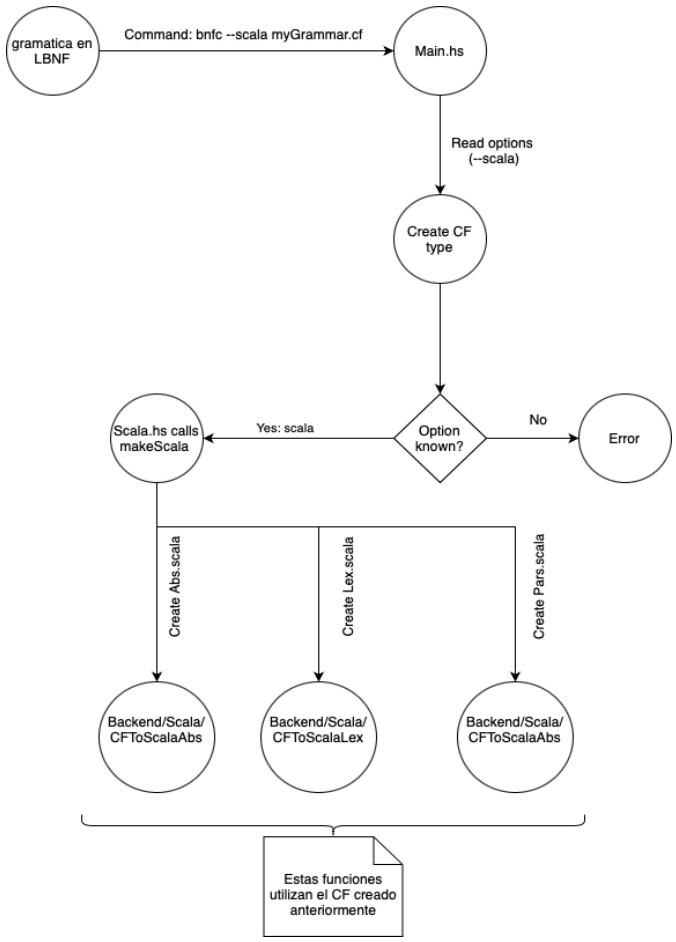

Figure 1.1: BNFC Workflow

La gramática expresada en *LBNF* debe estar en un archivo *.cf* y es el parámetro mas importante a la hora de utilizar *BNFC*.

Para utilizar el *backend* seleccionado, debemos incorporar al comando a pasarle a *BNFC* la opción *–scala*, la que el *Main.sh* leerá y utilizará para invocar a la función *makeScala* que es donde comenzará verdaderamente el flujo para en cuanto a *Scala* se refiere.

ends que generan *parsers* completos, este se enfoca únicamente en la generación de código para el resaltado de sintaxis, y su implementación está contenida principalmente en un único archivo *.hs*, lo que ilustra un enfoque de diseño diferente al de *backends* más complejos.

En definitiva, la función *makeScala* es nuestra interfaz con el resto de *BNFC*. A continuación detallamos su firma:

```
makeScala :: SharedOptions -> CF -> Backend
```

Para entender esta firma, es necesario comenzar a entender los distintos tipos utilizados internamente en *BNFC*

Los tipos y datos utilizados en *BNFC*

BNFC utiliza internamente algunos tipos y datos que nos serán de utilidad durante el desarrollo de nuestro *backend*. Probablemente el principal sea el *CF* (abreviatura de *Context Free grammar*), que representa en *Haskell* el CF del *LBNF* ingresado por el usuario.

```haskell
-- | A context free grammar consists of a set of rules and some
-- extended information (e.g. pragmas, literals, symbols, keywords).

type CF = CFG RFun

data CFG function = CFG
    { cfgPragmas        :: [Pragma]
    , cfgUsedCats       :: Set Cat    -- ^ Categories used by the parser.
    , cfgLiterals       :: [Literal]  -- ^ @Char, String, Ident, Integer,
                                      --   Double@.
                                      --   @String@s are quoted strings,
                                      --   and @Ident@s are unquoted.
    , cfgSymbols        :: [Symbol]   -- ^ Symbols in the grammar,
                                      --   e.g. “*”, “->”.
    , cfgKeywords       :: [KeyWord]  -- ^ Reserved words, e.g. @if@,
                                      --   @while@.
    , cfgReversibleCats :: [Cat]      -- ^ Categories that can be made
                                      --   left-recursive.
    , cfgRules          :: [Rul function]
    , cfgSignature      :: Signature  -- ^ Types of rule labels,
                                      --   computed from 'cfgRules'.
    } deriving (Functor)

type RFun    = RString
type RString = WithPosition String

data WithPosition a = WithPosition
  { wpPosition :: Position
  , wpThing    :: a
```

```
  } deriving (Show, Functor, Foldable, Traversable)

data Position
  = NoPosition
  | Position
    { posFile    :: FilePath  -- ^ Name of the grammar file.
    , posLine    :: Int       -- ^ Line in the grammar file.
    , posColumn  :: Int       -- ^ Column in the grammar file.
    } deriving (Show, Eq, Ord)
```

El tipo de *CF* es algo bastante complejo. Afortunadamente, *BNFC* cuenta con algunas funciones auxiliares utilizadas en común entre distintos *Backends*, que nos ayuda a simplificar el uso del mismo.
Un ejemplo sería la función *cf2data* que convierte un *CF* en un *Data*. El *Data* es un tipo más simple, el cual podemos utilizar para trabajar con nuestra *CF*, y que resultó de gran utilidad durante el desarrollo del *CFtoScalaAbs.hs* (el archivo que *BNFC* emplea para convertir la gramática definida por el usuario en *LBNF* a una representación más abstracta, para poder ser procesado por el resto de los módulos de nuestro *Backend*).

```
-- | The abstract syntax of a grammar.
type Data = (Cat, [(String, [Cat])])
```

Data es un *type* de *BNFC* que representa a una regla de la gramática y su lista de categorías. A modo de expresar con mayor claridad este concepto, y facilitar su comprensión, procedemos con el siguiente ejemplo (empleando *Calc*, que define reglas básicas para una aritmética simplificada):

```
-- file Calc.bnfc

EAdd.      Exp  ::= Exp  "+" Exp1 ;
ESub.      Exp  ::= Exp  "-" Exp1 ;
EMul.      Exp1 ::= Exp1 "*" Exp2 ;
EDiv.      Exp1 ::= Exp1 "/" Exp2 ;
EInt.      Exp2 ::= Integer       ;
coercions  Exp  2                 ;
```

La última línea ***"coercions Exp 2"*** es particularmente importante. Esta es una característica especial de *LBNF* que indica que puede haber conversiones implícitas (*coercions*) entre categorías sintácticas hasta 2 niveles de profundidad. En este caso, significa que:

- *Exp2* puede aparecer en cualquier lugar donde se espere *Exp1*.
- *Exp1* puede aparecer en cualquier lugar donde se espere *Exp*

Las *coercions* en *LBNF* permiten definir una gramática con precedencia de operadores sin tener que escribir todas las reglas de producción explícitamente. En este ejemplo, establece la precedencia estándar de operaciones aritméticas (multiplicación/división tienen mayor precedencia que suma/resta).

Cada una de las reglas (es decir, las *Exp*) son representados en *BNFC* como un *Data*, pudiendo tener éstas una sub lista, con las categorías que componen dicho *data*. En este caso, se vería así:

```haskell
-- Expresion en Haskell
("Exp", [
          "EAdd", [Exp, Exp]
          "ESub", [Exp, Exp]
          "EMul", [Exp, Exp]
          "EDiv", [Exp, Exp]
          "EInt", [Integer]
        ])
```

Durante la fase de análisis sintáctico del proceso de compilación (después del análisis léxico y antes de la generación del *AST*) se genera el Árbol de Análisis Concreto (o *CAT*, *Concrete Analysis Tree*), una estructura de datos que representa la derivación completa de una cadena de entrada según las reglas de la gramática.

Su función principal es capturar exactamente cómo se deriva la entrada desde el símbolo inicial de la gramática, mostrando cada paso de la derivación y preservando todos los detalles sintácticos, incluidos símbolos auxiliares como paréntesis o puntuación. A diferencia del *AST*, que elimina detalles superfluos y se enfoca en la estructura semántica, el *CAT* preserva la estructura sintáctica completa.

El *CAT* está compuesto por 'terminales' y 'no terminales' (estos últimos también referidos como N y que denominaremos de esa forma más adelante, en la sección "El generador de *parsers*"). Los 'no terminales' son aquellos elementos en la gramática que representan categorías sintácticas abstractas y actúan como nodos intermedios en el árbol sintáctico, requiriendo una expansión adicional según las reglas gramaticales. Los 'terminales', por su parte, son los símbolos básicos del lenguaje que aparecen directamente en el texto de entrada y constituyen las hojas del árbol.

Por ejemplo, en una estructura condicional como *'if'* condición, la palabra clave *'if'* es un terminal, mientras que la condición (que puede ser una expresión) no lo es. La condición eventualmente se resolverá a una secuencia de terminales (como variables, operadores y valores), pero tanto la condición como la estructura completa del *'if'* son considerados no terminales en

el contexto de la gramática, ya que representan construcciones sintácticas que se definen mediante reglas de producción.

1.1.3 Scala

Scala es un lenguaje de programación moderno, multiparadigma, que integra características de programación orientada a objetos y funcional. Está diseñado para expresar patrones de programación comunes de manera elegante y con tipado seguro. [2]

Los tipos y comportamientos se describen mediante clases y *traits*, y estas clases pueden extenderse a través de subclases y una composición flexible basada en *mixins* [3].

En su aspecto funcional, *Scala* trata las funciones como ”ciudadanos de primera clase”, ofreciendo soporte para funciones anónimas y de orden superior, *pattern matching* y colecciones inmutables. Su sistema de tipos es estático y avanzado, incorporando *generics*, anotaciones e inferencia de tipos automática.

Una de las ventajas significativas de *Scala* es su completa interoperabilidad con *Java* (con quien comparte varias similitudes, como un sistema de tipos estático y fuerte que verifica la compatibilidad de tipos en tiempo de compilación, o estructuras de control básicas con conceptos similares, aunque con sintaxis ligéramente diferentes.), permitiendo utilizar todas sus bibliotecas y compilando a bytecode de la *JVM*[4]. Esta combinación de programación orientada a objetos y funcional en un lenguaje estáticamente tipado ofrece una gran versatilidad a la hora de programar.

Como lenguaje multiparadigma, *Scala* ofrece una ventaja educativa única al permitir aprender tanto programación orientada a objetos como funcional simultáneamente. Esta dualidad otorga una visión más amplia de los paradigmas de programación que aprender con un lenguaje puramente orientado a objetos como *Java* o puramente funcional como *Haskell*.

La interoperabilidad de *Scala* con *Java* también representa una ventaja práctica, ya que da

[3] Los *mixins* son una suerte de subclases que se utilizan para componer una, y que no necesitan implementar los miembros de la clase abstracta que extiende.

[4] *Java Virtual Machine*

acceso inmediato a un vasto ecosistema de bibliotecas y *frameworks*, incluso facilitando el desarrollo de aplicaciones reales mientras se aprende.

Posiblemente por esos motivos, la Cátedra de Teoría de la Computación de la Universidad ORT del Uruguay ha expresado interés en incorporar este lenguaje a sus cursos (en particular, en la materia ”Lenguajes y compiladores”), con la idea de usarlo en lugar de *Haskell*.

Sin embargo, *BNFC* (la herramienta anteriormente mencionada) no incluye a *Scala* como uno de los lenguajes con el que sea posible operar (dentro de los que se cuentan *Haskell*, *Java*, y *C++*, por mencionar algunos). Es por ello que comenzamos analizar la posibilidad de extender la herramienta, incorporando el lenguaje *Scala* como un nuevo módulo.

1.2 Especificación del Problema

El proceso de análisis sintáctico es el segundo paso en la construcción de un compilador, encargándose de transformar una secuencia de *tokens* en una estructura jerárquica, típicamente un árbol de sintaxis abstracta (*AST*[5]).

Implementar un *parser* en *Scala* puede ser un reto debido a la necesidad de manejar múltiples reglas gramaticales y evitar la ambigüedad, todo mientras se mantiene un código legible y mantenible. Una solución efectiva en *Scala* es el uso de la librería *Scala Parser Combinators*, que permite construir *parsers* de manera modular y declarativa mediante la composición de pequeñas funciones que representan reglas gramaticales.

Esta aproximación facilita el desarrollo y la extensibilidad del *parser*, reduciendo la complejidad en comparación con implementaciones manuales basadas en autómatas o técnicas de *recursive descent parsing* tradicionales. Además, en nuestro caso, no solo estamos desarrollando un *parser*, sino que también estamos construyendo un generador de *parsers*. Esto significa que necesitamos una solución flexible y reutilizable que nos permita definir reglas gramaticales de manera clara y modular.

Este enfoque resulta aún más adecuado considerando la estructura de *BNFC* (*Backus-Naur*

[5] *Abstract Syntax Tree*

Form Converter). *BNFC* nos proporciona una gramática libre de contexto (*CF*[6]) junto con utilidades para obtener información estructurada sobre la misma. Gracias a esto, podemos aprovechar la modularidad de los combinadores de *parsers* en *Scala* para transformar directamente la gramática *CF* en un *parser* funcional, asegurando así una implementación más limpia, mantenible y alineada con la estructura proporcionada por *BNFC*.

1.3 Métodos de Solución

Para la implementación del *parser* en *Scala*, tomamos como base el tutorial disponible en *Building a lexer and parser with Scala's Parser Combinators*, que fue desarrollado por los mismos creadores de la librería *Scala Parser Combinators*. Este tutorial nos permitió comprender los principios fundamentales para la construcción de un *parser* en *Scala*, guiándonos a través del proceso de análisis sintáctico y la composición de combinadores de *parsers*. A partir de esta base, nuestro trabajo consiste en transpolar estos conceptos hacia la construcción de un generador de *parsers*, asegurando que la implementación final sea flexible y modular.

La metodología descrita en el tutorial nos sirvió como referencia para estructurar nuestras reglas gramaticales y optimizar la forma en que se interpretan en el contexto de un *parser* autogenerado. Además, para que el código autogenerado sea ejecutable en *Scala*, nos fue necesario estudiar *SBT* (*Simple Build Tool*), comprendiendo su estructura y configuración para gestionar dependencias y la compilación del proyecto de manera eficiente. Asimismo, ha sido fundamental aprender sobre el uso de *packages* en *Scala*, ya que los distintos archivos generados deben poder acceder entre sí de manera estructurada. Esto nos ha permitido organizar correctamente el código y garantizar que cada componente del *parser* generado pueda interactuar sin conflictos dentro del entorno de ejecución en *Scala*.

[6] *Context Free*

El generador de *Lexers*

2.1 Implementación

Para la construcción del *Lexer*, primero analizamos los *tokens* definidos en la gramática libre de contexto (*CF*) proporcionada por *BNFC*. Para ello, utilizamos la función ***unicodeAndSymbols*** de *BNFC*, que nos permite extraer todos los símbolos y palabras clave definidos en la gramática. Una vez obtenidos estos símbolos, procedemos a convertir cada uno de ellos en funciones de *Scala*, siguiendo el patrón de combinadores de *parsers*.

Cada símbolo se transforma en una función que lo reconoce dentro del *lexer*, siguiendo un formato como el siguiente:

```
def semi = positioned { ";" ^^ (_ => SEMI()) }
```

Este procedimiento se aplica a todos los símbolos extraídos de la gramática. Sin embargo, algunos elementos como los identificadores (***Ident***) requieren un tratamiento especial, ya que no son símbolos fijos, sino que deben seguir ciertas reglas sintácticas. Para estos casos, definimos funciones específicas, por ejemplo:

```
def ident: Parser[IDENT] = {
    "[a-zA-Z_][a-zA-Z0-9_]*".r ^^ { str => IDENT(str) }
}
```

Aquí, la función ***ident*** devuelve un objeto de tipo ***Parser[IDENT]*** que reconoce patrones de texto que cumplen con la expresión regular `"[a-zA-Z_][a-zA-Z0-9_]*"`[1]. El operador

[1] Esta expresión captura secuencias que comienzan con una letra (mayúscula o minúscula) o un guion bajo, seguidas de cero o más caracteres que pueden ser letras, dígitos o guiones bajos (el patrón típico para identificadores en muchos lenguajes de programación).

∧∧ toma la cadena de texto reconocida y la transforma en un *token* ***IDENT*** que contiene el texto original como parámetro, preservando así el valor exacto del identificador para su posterior uso en el análisis sintáctico o la generación de código.

El *lexer* es implementado extendiendo la clase ***RegexParsers*** de *Scala Parser Combinators*, lo que nos permite definir reglas de reconocimiento basadas en expresiones regulares de manera concisa y estructurada.

Una vez que todas las funciones han sido creadas, debemos asegurarnos de que el lexer pueda reconocerlas en conjunto. Para ello, las incluimos dentro de la función ***tokens***, que es responsable de agrupar todas las reglas y definir cómo se estructuran los flujos de entrada:

```
def tokens: Parser[List[WorkflowToken]] = {
    phrase(rep1( semi | dot | dcoloneq | lbrack | rbrack | underscore |
                 lparen | colon | rparen | comma | bar | minus | star |
                 plus | question | lbrace | rbrace | ident | char |
                 integer | string))
}
```

Finalmente, definimos la función de entrada del *lexer*, llamada ***apply***, la cual recibe el código fuente como una cadena de texto y aplica el análisis léxico sobre él. Esta función sigue una estructura estándar y siempre se define de la misma manera:

```
def apply(code: String): Either[WorkflowLexerError, List[WorkflowToken]]
= {
    parse(tokens, code) match {
        case NoSuccess(msg, next) => Left(WorkflowLexerError(
                        Location(next.pos.line, next.pos.column), msg))
        case Success(result, next) => Right(result)
    }
}
```

Para el manejo de errores en el *lexer* y el *parser*, hemos definido las clases ***WorkflowLexerError***, ***WorkflowParserError*** y ***Location***. Estas clases nos permiten almacenar información detallada sobre los errores encontrados durante el proceso de análisis. La clase ***Location*** es utilizada para registrar la posición exacta (línea y columna) en la que ocurre un error, facilitando la depuración del código.

Con esta implementación, logramos generar un *lexer* funcional basado en la gramática extraída de *BNFC*, asegurando una integración estructurada con el *parser* y un manejo eficiente de los errores.

El generador de *Parsers*

3.1 Implementación

Existen dos métodos bien estudiados para realizar el análisis sintáctico (*parsing*): el análisis determinista de izquierda a derecha y descendente (*LL method*) y el análisis determinista de izquierda a derecha y ascendente (*LR* y *LALR methods*). Además, existe una tercera técnica emergente conocida como *Generalized LR*.

El término *left-to-right* indica que el texto del programa, o más precisamente la secuencia de *tokens*, se procesa de izquierda a derecha, analizando un *token* a la vez. De manera intuitiva, el término *determinista* significa que el proceso no requiere retroceso ni búsqueda: cada *token* acerca al *parser* un paso más a la construcción del árbol sintáctico, sin necesidad de deshacer pasos previos. Desde la teoría de lenguajes formales, se puede definir con más rigor que estos métodos funcionan en tiempo lineal con respecto a la longitud de la entrada, es decir, son algoritmos de tiempo lineal.

Además, la determinación es importante por otra razón: una gramática para la cual se puede generar un *parser* determinista está garantizada como no ambigua. La no ambigüedad es una propiedad crucial en las gramáticas de los lenguajes de programación, ya que evita múltiples interpretaciones de una misma estructura sintáctica. Si bien ser determinista y poder generar un *parser* no ambiguo no son exactamente lo mismo (la determinación implica no ambigüedad, pero no viceversa), exigir determinismo es la mejor prueba técnica de no ambigüedad de la que disponemos.

3.1.1 *Scala Parser Combinator* y el enfoque *Top-Down*

En nuestro caso, estamos construyendo un parser utilizando *Scala Parser Combinator*[3], el cual implementa la estrategia de análisis sintáctico descendente de izquierda a derecha (*Left-Right Top-Down Parsing*) mediante el método de *Recursive Descent Parsing*.

Esto significa que, dado un no terminal N y un *token* t en la posición p de la entrada, un *parser* descendente debe decidir qué alternativa de N debe aplicarse para que el sub-árbol encabezado por el nodo etiquetado como N sea el correcto en la posición p. Sin embargo, no podemos definir con certeza qué hace que un árbol sea correcto, pero sí sabemos cuándo es incorrecto: cuando tiene un *token* diferente de t como su hoja más a la izquierda en la posición p. Esto nos proporciona una aproximación razonable de cómo debe ser un árbol correcto: un árbol que comienza con t o está vacío.

La forma más obvia de decidir qué alternativa de N es la correcta es tener una función booleana recursiva que pruebe las alternativas de N en sucesión y que tenga éxito cuando encuentre una alternativa capaz de generar un árbol posible. Para hacer que el método sea determinista, decidimos no realizar ningún *backtracking*: la primera alternativa que puede producir un árbol posible se asume como la alternativa correcta. Sin embargo, esta suposición ocasionalmente puede generar problemas. Este enfoque da lugar a un *Recursive Descent Parser*.

Los *Recursive Descent Parsers* han sido utilizados durante muchos años por los desarrolladores de compiladores y continúan siendo una opción popular debido a su simplicidad y facilidad de implementación. De hecho, escribir un *Recursive Descent Parser* sigue siendo la forma más sencilla de construir un *parser* básico.

3.1.2 Implementación de un *Recursive Descent Parser*

En este enfoque, cada regla N de la gramática corresponde a una rutina que devuelve un valor booleano:

- Retorna ***true*** si se encuentra una producción terminal de N en la posición actual del flujo de entrada, consumiendo la parte correspondiente de la secuencia de *tokens*.
- Retorna ***false*** si no se encuentra una producción terminal de N, en cuyo caso no se consume ningún *token*.

Para ello, la rutina prueba cada una de las alternativas de N en orden, verificando si una de ellas está presente. Para determinar si una alternativa es válida, se comprueba la presencia de su

primer componente de manera recursiva. Si el primer componente está presente, la alternativa se considera correcta y se procede a evaluar los siguientes componentes. Si el primer componente no está presente, no se consume entrada y se prueba la siguiente alternativa.

Si ninguna de las alternativas tiene éxito, N no está presente en la entrada y la rutina para N devuelve ***false***, sin consumir *tokens*, ya que ninguna llamada fue exitosa. En caso de que un componente de una alternativa sea obligatorio y no se encuentre en la entrada, se genera un error de sintaxis, el cual se reporta y el *parser* se detiene.

3.1.3 Implementación de un *Recursive Descent Parser* utilizando *Scala Parser Combinator*

En el caso de *Scala Parser Combinator*, este enfoque es implementado a través de la clase `Parsers`. Esto nos proporciona una estructura de trabajo que nos permite enfocarnos en el análisis de las reglas de la gramática. La forma en que funciona es mediante la creación de *parsers* modulares, lo que se comprende mejor con un ejemplo.

Supongamos que queremos crear un *parser* para una gramática que consiste en una palabra seguida de un número. En lugar de construir un árbol de sintaxis abstracto (AST), simplemente generamos una instancia de una clase que contiene la palabra y el número. En *Scala Parser Combinator*, este *parser* se definiría de la siguiente manera:

```
class SimpleParser extends RegexParsers {
  def word: Parser[String]   = """[a-z]+""".r       ^^ { _.toString }
  def number: Parser[Int]    = """(0|[1-9]\d*)""".r ^^ { _.toInt }

  def exp: Parser[WordFreq]  = word ~ number        ^^ {
    case wd ~ fr => WordFreq(wd, fr)
  }
}
```

En este caso, nuestro *parser* se compone de tres partes principales. Las dos primeras, `word`

y `number`, emplean expresiones regulares para la tokenización (*lexer*) de la gramática. La tercera regla, `exp`, es la que realiza el análisis sintáctico propiamente dicho. Analicemos la siguiente línea:

```
def exp: Parser[WordN] = word ~ number ^^
{ case wd ~ n => WordN(wd, n) }
```

Aquí, `exp` es una función que devuelve un objeto de tipo `Parser[WordN]`, es decir, una instancia de `Parser`. Esto es crucial, ya que permite construir el *parser* de forma modular.

La función `exp` utiliza el operador $\wedge\wedge$, el cual evalúa si la expresión a su izquierda se cumple. En caso afirmativo, aplica la función a su derecha, que en este caso crea una instancia de `WordN` con los valores obtenidos.

La expresión a la izquierda en este caso utiliza el símbolo $\sim$ para indicar que un operador es seguido de otro, es decir en nuestro ejemplo que esperamos una palabra seguida de un número

Estos módulos de *Scala Parser Combinator* serán generados automáticamente en nuestro sistema, tomando como punto de partida las reglas de la gramática ingresadas en *LBNF*.

Es decir que para construir nuestro *parser*, debemos crear una clase que herede de ***Regex-Parsers*** y definir las N reglas de nuestra gramática.

A continuación, presentamos un ejemplo de una calculadora simple implementada con ***Scala Parser Combinator*** :

```
object Calculator extends RegexParsers {
  def number: Parser[Double] = """\d+(\.\d*)?""".r ^^ { _.toDouble }
  def factor: Parser[Double] = number | "(" ~> expr <~ ")"

  def term: Parser[Double] = factor ~ rep("*" ~ factor | "/" ~ factor)
    ^^ {
      case number ~ list => list.foldLeft(number) {
```

```
        case (x, "*" ~ y) => x * y
        case (x, "/" ~ y) => x / y
      }
    }

  def expr: Parser[Double] = term ~ rep("+" ~ log(term)("Plus term")
    | "-" ~ log(term)("Minus term")) ^^ {
      case number ~ list => list.foldLeft(number) {
        case (x, "+" ~ y) => x + y
        case (x, "-" ~ y) => x - y
      }
    }

  def apply(input: String): Double = parseAll(expr, input) match {
    case Success(result, _) => result
    case failure: NoSuccess => scala.sys.error(failure.msg)
  }
}
```

En este caso, las reglas gramaticales están representadas por las funciones ***number***, ***factor***, ***term*** y ***expr***, las cuales corresponden a las distintas reglas de nuestra gramática.

La principal diferencia con el primer ejemplo mostrado radica en que las funciones `term` y `expr` presentan una mayor complejidad. Sin embargo, al analizarlas detenidamente, podemos observar que su estructura sigue siendo clara y comprensible. Consideremos la definición de `term`:

```
def term: Parser[Double] = factor ~ rep("*" ~ factor | "/" ~ factor)
  ^^ {
    case number ~ list => list.foldLeft(number) {
      case (x, "*" ~ y) => x * y
```

```
      case (x, "/" ~ y) => x / y
    }
  }
```

Explicaremos la función `term`, esperamos que con esta explicacion, el lector pueda ser capaz de entender también la función `exor`.

La función `term` devuelve un `Parser[Double]`. Para construir este *parser*, primero evalúa si el inicio de la expresión corresponde a un `factor`. Según la definición de `factor`, este puede ser un número seguido de una expresión entre paréntesis.

A continuación, utilizamos la función `rep`, que nos permite hacer una llamada recursiva sobre el mismo *parser*. Esto se debe a que, tras un `factor`, podemos encontrar el operador * seguido de otro `factor`, o bien el operador `/` seguido de otro `factor`.

Dado que el objetivo de nuestra calculadora es evaluar la expresión aritmética, utilizamos `pattern matching` con la palabra clave `case` de Scala. En este proceso, la función toma el número inicial junto con la lista de operaciones, y luego aplica `foldLeft` para recorrer la lista y computar el resultado. Los casos en los que se encuentra un número seguido de un operador (* o `/`) y otro número se resuelven de manera trivial.

Por último tenemos la función apply:

```
def apply(input: String): Double = parseAll(expr, input) match {
    case Success(result, _) => result
    case failure: NoSuccess => scala.sys.error(failure.msg)
  }
```

La función ***apply*** actúa como el punto de entrada del *parser*. Utiliza la función ***parseAll*** de ***Scala Parser Combinator*** para iniciar el proceso de análisis sintáctico, tomando como regla inicial ***expr***. Posteriormente, maneja los casos de éxito y error, devolviendo el resultado o generando un mensaje de error en caso de fallo.

Este enfoque modular y declarativo permite definir parsers de manera concisa, aprovechando el poder de *Scala Parser Combinator* para estructurar y procesar la gramática de forma eficiente, y simplifica el trabajo a la hora de auto-generar el código utilizando *BNFC*.

3.1.4 Generar el parser con BNFC

Para implementar el *parser generator* en BNFC, el primer paso es identificar los *entry points* del lenguaje. Un *entry point* corresponde a las reglas de entrada de la gramática, es decir, aquellas que definen el punto de inicio del análisis sintáctico.

Como ejemplo, consideremos la gramática definida en el archivo ***Calc.cf***:

```
-- file Calc.cf
EAdd.      Exp  ::= Exp  "+" Exp1 ;
ESub.      Exp  ::= Exp  "-" Exp1 ;
EMul.      Exp1 ::= Exp1 "*" Exp2 ;
EDiv.      Exp1 ::= Exp1 "/" Exp2 ;
EInt.      Exp2 ::= Integer       ;
coercions  Exp  2                 ;
```

En este caso, el *entry point* del lenguaje es la regla ***Exp***, ya que representa la estructura principal de las expresiones matemáticas.

Para *Scala Parser Combinator*, es necesario marcar esta regla como *entry point* de la siguiente manera:

```
def program: Parser[WorkflowAST] = positioned {
    phrase(expr)
}
```

Luego navegaremos a través de los distintos niveles en que una expresión `expr` puede reducirse.

Cabe destacar que, en *Scala Parser Combinator*, la función principal utilizada para invocar el *parser* es ***apply***. No obstante, con el fin de simplificar la implementación, hemos diseñado ***apply*** de manera que siempre llame a la función ***program***. Para más detalles, se puede consultar el código en el Anexo.

Para generar esta función contamos con la ayuda de la función *allEntryPoints* de *BNFC.CF*. En lugar de utilizar todos los puntos de entrada como alternativas, tomamos solo el primero y lo convertimos a minúscula para que sea compatible con la sintaxis de *Scala*. El código al final queda:

```
getProgramFunction :: CF -> [Doc]
getProgramFunction cf = [
    "def program: Parser[WorkflowAST] = positioned {"
    , nest 4 $ text $ "phrase(" ++ eps ++ ")"
    ,"}"
  ]
  where
    eps = firstLowerCase $ show $ head $ map normCat $ DF.toList
        $ allEntryPoints cf
```

El siguiente paso en la construcción del generador de *parser* consiste en manejar las reglas marcadas como *coercion* en la gramática *BNFC*, estas son por ejemplo *Exp*, *Exp1*, y *Exp2* en el ejemplo de *Calc.cf*
Para ello es necesario procesar la gramática libre de contexto (*CF*) de *BNFC* y generar manualmente las reglas correspondientes.

Para lograrlo, hemos desarrollado las siguientes funciones en *Haskell*:

```
prSubRule :: Rule -> [String]
prSubRule r@(Rule fun _ _ _)
  | isCoercion fun = []
  | otherwise = ["case (" ++ head vars ++ ", " ++ intercalate " ~ "
    (safeTail vars) ++ ") => " ++ fnm ++ "("
    ++ intercalate ", " (filterSymbs vars) ++ ")"]
  where
    vars = disambiguateNames $ map modifyVars (prPrintRule_ r)
    fnm = funName fun
    modifyVars str = if "()" `isSuffixOf` str then str
        else [toLower (head str)]


coerCatDefSign :: Cat -> Doc
coerCatDefSign cat =
      text $ "def " ++ pre ++ ": Parser[WorkflowAST] = positioned {"
  where
    sCat = render $ catToType id empty cat
    pre = firstLowerCase $ show cat

prSubRuleDoc :: Rule -> [Doc]
prSubRuleDoc r =  map text (prSubRule r)


filterNotEqual :: Eq a => a -> [a] -> [a]
filterNotEqual element list = filter (/= element) list


rulesToString :: [Rule] -> [Doc]
rulesToString [] = [""]
```

```
rulesToString rules@(Rule fun cat _ _ : _) =
    let
        -- Filtramos las reglas que pertenecen a la misma
        -- categoría `cat`
        (sameCat, rest) = span (\(Rule _ c _ _) -> c == cat) rules

        -- Obtenemos los símbolos de las reglas filtradas
        vars = concatMap prPrintRule_ sameCat
        fnm = funName fun
        exitCatName = firstLowerCase $ head $ filterNotEqual
            (show (wpThing cat)) $ filterSymbs vars

        -- Generamos la cabecera única con todas las alternativas
        -- dentro de `rep(...)`
        header = text $ exitCatName ++ " ~ rep((" ++ intercalate
            " | " (onlySymbs (safeTail vars)) ++ ") ~ "
            ++ exitCatName ++ ") ^^ {"
        subHeader = text $ "case " ++ exitCatName ++ " ~ list
            => list.foldLeft(" ++ exitCatName ++ ") {"

        -- Generamos los `case` correspondientes a las reglas
        -- de `sameCat`
        rulesDocs = concatMap prSubRuleDoc sameCat
    in [header] ++ map (nest 4) [subHeader] ++ map (nest 8) rulesDocs
        ++ [nest 4 "}"] ++ ["}"] ++ rulesToString rest

ruleGroupsCFToString :: [(Cat,[Rule])] -> [Doc]
ruleGroupsCFToString [] = [""]
```

```
ruleGroupsCFToString ((c, r):crs) =
    [coerCatDefSign c] ++ map (nest 4) (rulesToString r) ++ ["}"]
        ++ ruleGroupsCFToString crs
```

A continuación, analizamos cada una de sus funciones en detalle:

- **`prSubRule`**: Esta función toma una regla (`Rule`) y devuelve una lista de cadenas que representan la regla en el formato esperado por el *parser*.
  - Si la función asociada a la regla es una *coercion*, se retorna una lista vacía.
  - En caso contrario, genera una estructura de caso que maneja el primer elemento separadamente y conecta los elementos restantes con el operador `~`, además de aplicar los nombres de variables desambiguados.
  - Utiliza la función `disambiguateNames` y `modifyVars` para garantizar nombres de variables únicos y bien formados.
- **`coerCatDefSign`**: Esta función genera la firma de una función de *parsing* para una categoría gramatical, definiendo un *parser* posicionado que devuelve un tipo `WorkflowAST`.
- **`prSubRuleDoc`**: Convierte el resultado de `prSubRule` en un tipo de documento (`Doc`).
- **`filterNotEqual`**: Función auxiliar que filtra elementos de una lista que son diferentes al elemento proporcionado.
- **`rulesToString`**: El propósito de sta función es generar *parsers* recursivos a izquierda, de modeo qué:
  - Agrupa las reglas por categoría y las procesa en conjunto.
  - Genera una estructura que utiliza el *combinator* `rep` de *Scala Parser Combinators* para manejar la recursión.
  - Construye una expresión de plegado (`foldLeft`) que aplica secuencialmente las diferentes reglas.
  - Formatea el código con la indentación adecuada para mejorar la legibilidad.

- **`ruleGroupsCFToString`**: Convierte un conjunto de reglas agrupadas por categorías en documentos formateados, manteniendo la estructura anidada correcta y añadiendo los corchetes de cierre necesarios.

Estas funciones procesan la gramática y generan automáticamente las reglas de *parsing* correspondientes.

La implementación genera parsers que pueden manejar gramáticas recursivas a izquierda, lo que es crucial para expresiones aritméticas y otras construcciones comunes en lenguajes de programación. El enfoque utiliza los *combinators* de *Scala* para crear una estructura de *parsing* más robusta y mantener la precedencia de operadores correctamente.

3.1.5 Ejemplo: Generación de código a partir de *Calc.cf*

Para ilustrar este proceso, tomemos nuevamente como ejemplo la gramática ***Calc.cf***. A partir de ella, nuestro generador produce el siguiente código en *Scala*:

```
def exp: Parser[WorkflowAST] = positioned {
    exp1 ~ rep((PLUS() | MINUS()) ~ exp1) ^^ {
        case exp1 ~ list => list.foldLeft(exp1) {
            case (e1, PLUS() ~ e2) => EAdd(e1, e2)
            case (e1, MINUS() ~ e2) => ESub(e1, e2)
        }
    }
}

def exp1: Parser[WorkflowAST] = positioned {
    factor ~ rep((STAR() | SLASH()) ~ exp2) ^^ {
        case exp2 ~ list => list.foldLeft(exp2) {
            case (e1, STAR() ~ e2) => EMul(e1, e2)
            case (e1, SLASH() ~ e2) => EDiv(e1, e2)
        }
```

```
    }
}

def exp2: Parser[WorkflowAST] = positioned {
    integer
}

def integer: Parser[EInt] = positioned {
    accept("integer", { case INTEGER(i) => EInt(i.toInt) })
}
```

Procedemos a explicar el código autogenerado:

- **`exp`**: Define la regla para expresiones de nivel superior, que pueden ser una suma o una resta entre dos expresiones de nivel inferior, que denominamos *exp1*.
  - Si se encuentra una expresión seguida por un operador `+` y otra expresión de nivel inferior, se construye un nodo `EAdd` en el árbol sintáctico.
  - De manera similar, si el operador encontrado es `-`, se crea un nodo `ESub`.
- **`exp1`**: Maneja la precedencia de los operadores de multiplicación y división.
  - Si se encuentra una expresión seguida de un `*` y otra expresión de nivel inferior (`exp2`), se genera un nodo `EMul`.
  - Si el operador es `/`, se construye un nodo `EDiv`.
- **`exp2`**: Se trata de un paso intermedio que usamos para simplificarnos la tarea de identificar las categorías definidas en LBNF, remite a la definición de *integer*
- **`integer`**: Define el caso base de la recursión, que corresponde a un número entero.
  - Cuando el *parser* detecta un número entero, lo encapsula en un nodo `EInt`.

Cada uno de los operadores generados (Emul, Ediv, etc) puede a su vez contener otra gramática, siendo el caso base de *integer* el único que no, y el cual nos permite evitar un *loop* infinito.

Si por ejemplo, intentáramos *parsear* la expresión *"3*1+2"*, el proceso sería el siguiente:

- Primero, al llamar a `exp` se evalúa *exp1* para *"3*1"*:
  - El *parser* intenta analizar un `expl` (*"3*1"*)
  - Dentro de `expl`, llama a `exp2` para "3".
  - `exp2` llama a `integer`, que crea *EInt(3)*
  - Luego encuentra * y busca otro factor, que es "1".
  - El resultado de analizar `expl` es *EMul(EInt(3), EInt(1))*
- A continuación, se evalúa *rep((PLUS() | MINUS()) exp1)*:
  - Esta parte busca cualquier operador + o - seguido de otro `expl`
  - En este caso, encuentra *"+2"*
  - Por lo tanto, *list* contiene un solo elemento: *PLUS() EInt(2)*
- Después, se aplica *list.foldLeft(term)* que toma un valor inicial y una lista, y acumula un resultado aplicando una función binaria de izquierda a derecha:
  - El valor inicial es `expl`, que es *EMul(EInt(3), EInt(1))*
  - Por cada elemento en *list*, aplica la función de caso: Para *PLUS() EInt(2)*, ejecuta para obtener como resultado final: *EAdd(EMul(EInt(3), EInt(1)), EInt(2))*

El patrón *foldLeft* permite construir el *AST* progresivamente, aplicando operadores de izquierda a derecha. Las operaciones del mismo nivel (suma y resta) se evalúan de izquierda a derecha, lo que refleja la asociatividad natural de los operadores matemáticos. La estructura anidada del *parser* (*exp -> exp1 -> exp2*) garantiza que la multiplicación y la división tengan mayor precedencia que la suma y la resta, obteniendo así una expresión matemáticamente correcta, y no ambigua.

```
scala> WorkflowCompiler("1+(3*2)")
res1: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Left(WorkflowParserError(1:3,integer expected))

scala> WorkflowCompiler("1+3*2")
res2: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Right(EAdd(EInt(1),EMul(EInt(3),EInt(2))))

scala> WorkflowCompiler("2+3*4/1+1-5")
res3: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Right(ESub(EAdd(EAdd(EInt(2),EDiv(EMul(EInt(3),EInt(4)),EInt(1))),EInt(1)),EInt(5)))

scala> WorkflowCompiler("3+4*2-1")
res4: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Right(ESub(EAdd(EInt(3),EMul(EInt(4),EInt(2))),EInt(1)))

scala> WorkflowCompiler("3*4+2/6")
res5: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Right(EAdd(EMul(EInt(3),EInt(4)),EDiv(EInt(2),EInt(6))))

scala> WorkflowCompiler("3+4*2")
res6: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Right(EAdd(EInt(3),EMul(EInt(4),EInt(2))))

scala> WorkflowCompiler("4*2+3")
res7: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Right(EAdd(EMul(EInt(4),EInt(2)),EInt(3)))

scala> WorkflowCompiler("2+3+x")
res8: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Left(WorkflowLexerError(1:5,string matching regex '[0-9]+' expected but 'x' found))

scala> WorkflowCompiler("/2+3+")
res9: Either[Calc.workflowtoken.CalcLex.WorkflowCompilationError,Calc.workflowtoken.CalcParser.WorkflowAST] =
Left(WorkflowParserError(1:1,integer expected))
```

Figure 3.1: Ejemplos de expresiones de prueba con diferentes niveles de complejidad

Un detalle a aclarar sobre el resultado de las pruebas, es la presencia de un *Right* o *Left* en el retorno. Esto es así por la forma en que definimos nuestro método `apply` (el punto de partida en nuestro *parser*).

```
def apply(tokens: Seq[WorkflowToken]):
    Either[WorkflowParserError, WorkflowAST] = {
        val reader = new WorkflowTokenReader(tokens)
        program(reader) match {
            case NoSuccess(msg, next) =>
                Left(WorkflowParserError(Location
                    (next.pos.line, next.pos.column), msg))
        case Success(result, next) => Right(result)
    }
}
```

En *Scala*, *Either* es un tipo que puede contener uno de dos valores posibles: uno a la izquierda (*Left*) o uno a la derecha (*Right*). Es comúnmente usado para manejar operaciones que pueden fallar, donde:

- *Left* generalmente se usa para representar un error o fallo.
- *Right* generalmente se usa para representar un resultado exitoso.

Así, cuando el *parsing* es exitoso, se obtiene un valor *Right* que contiene nuestro *AST*. Si hubiera un error de compilación (*lexer* o *parser*), se obtendrá un valor *Left* con información sobre el error (como se observa para la expresión *("3*(4+2)/6)")*.

La estructura de estas reglas es similar a la del ejemplo de la calculadora basado en ***RegexParser***, pero con una diferencia fundamental: en nuestro código autogenerado, el parser no evalúa directamente las expresiones (es decir, no realiza las operaciones aritméticas aunque disponga de toda la información necesaria para hacerlo). En su lugar, nuestro objetivo es construir un **árbol sintáctico abstracto (AST)**, que posteriormente puede ser utilizado en otras etapas del procesamiento del lenguaje, como la evaluación, optimización o traducción a otro formato.

3.1.6 Manejo de Categorías Base en *BNFC*

Por último, es necesario agregar las reglas correspondientes a las categorías base de *BNFC*. Estas categorías incluyen:

```
-- Built-in categories constants
catString, catInteger, catDouble, catChar, catIdent :: TokenCat
catString  = "String"
catInteger = "Integer"
catDouble  = "Double"
catChar    = "Char"
catIdent   = "Ident"
```

Para cada una de estas categorías se agregan reglas predefinidas en *Scala*. Por ejemplo, la regla para manejar números de punto flotante (***Double***) es la siguiente:

```
getDoubleFunction :: Doc
getDoubleFunction = vcat [
      "def double: Parser[Double] = {"
    , nest 4 "\"[0-9]+.[0-9]+\".r ^^ {i => Double(i)}"
    , "}"
  ]
```

Este enfoque nos permite generar automáticamente un *parser* funcional para cualquier gramática definida en *LBNF*, manteniendo un diseño modular y extensible.

Conclusiones

4.1 Proceso de desarrollo

Nuestro proceso de desarrollo se enfocó principalmente en la investigación y en responder las preguntas fundamentales que surgieron tanto al inicio del proyecto como a lo largo de su evolución. Algunas de las principales cuestiones que guiaron nuestro trabajo fueron:

- ¿Cómo podemos escribir un *parser* en *Scala*?
- ¿Cómo podemos automatizar la generación del *parser*?
- ¿Cómo leemos las reglas de *LBNF* para poder trabajar con ellas?
- ¿Existe alguna herramienta preexistente que nos ayude con esto?
- ¿Qué herramientas de *Scala* nos permiten crear un *parser*? ¿Existe algún estándar en *Scala* para este propósito?

Para abordar estas preguntas, las primeras etapas del proyecto consistieron en un proceso de aprendizaje e investigación sobre diversas herramientas y tecnologías clave. En particular, nos enfocamos en aprender el lenguaje de programación *Scala* y su ecosistema, *Haskell* como base para trabajar con *BNFC*, y el funcionamiento interno de *BNFC* en sí mismo. Esto nos permitió comprender cómo aprovechar esta herramienta para la generación automática de *parsers* y cómo integrarla eficientemente dentro de nuestro flujo de trabajo.

Además, exploramos diferentes enfoques para la construcción de *parsers* en *Scala*, analizando las ventajas y limitaciones de cada uno. Identificamos que el uso de *Scala Parser Combinators* era la mejor opción debido a su enfoque modular y declarativo, lo que facilitó su integración con las reglas extraídas de *BNFC*.

Este enfoque basado en la investigación y el aprendizaje continuo nos permitió construir una solución bien fundamentada, asegurando que cada decisión tomada estuviera respaldada por un análisis técnico sólido.

4.1.1 Fases del desarrollo

El desarrollo del proyecto se estructuró en varias fases, cada una enfocada en la construcción progresiva de los distintos componentes necesarios para la generación automática de un *parser* en *Scala* a partir de una gramática en *BNFC*. Estas fases fueron las siguientes:

- **Escribir el *CFtoScalaAbs.hs***
  - Estudiar *Haskell*.
  - Estudiar *Scala*.
  - Estudiar *BNFC*.
  - Agregar la estructura de nuestro *backend* de *Scala* en *BNFC*.
- **Escribir el *CFtoScalaLex.hs***
  - Estudiar *Scala Parser Combinators*.
  - Generar un proyecto *SBT* funcional para poder probar los avances.
  - Trabajar con los ejemplos existentes en *BNFC* para poder verificar el código generado.
- **Escribir el *CFtoScalaParser.hs***
  - Continuar con el estudio de *Scala Parser Combinators*.
  - Aprender sobre compiladores modernos y cómo generar un *parser* lo más genérico posible.

4.2 Dificultades

Durante los últimos meses nos hemos encontrado con varios problemas durante el desarrollo de nuestra tesis. Todos relativamente ”sencillos” de resolver, pero ciertamente inesperados:

- A pesar de estar bastante bien documentado, *BNFC* es un proyecto complejo, que nos demandó más tiempo del que esperábamos, para lograr entender, y para realmente estar en condiciones de trabajar sobre él y extenderlo. En este punto creemos tener una buena base de cómo funciona, aunque con cada paso nos encontramos con nuevas sorpresas.

- Generar el *AST* fue bastante simple (una vez comprendimos como funciona *BNFC* internamente), pero nos quedó claro que nuestro conocimiento de *Haskell* estaba muy por detrás del necesario para desarrollar un nuevo modulo en *BNFC*

- A la hora de generar el *lexer* y el *parser* en los distintos lenguajes que *BNFC* soporta, éste siempre se apoya en alguna otra herramienta, como por ejemplo *Happy* [1] o *Alex3* [2], para el caso de *Haskell*. En nuestro caso, tuvimos que trabajar con *Scala Parser Combinator*, una librería estándar de *Scala* la cual, aunque muy útil, esta muy mal documentada y es menos potente que las herramientas homólogas de Haskell o Java.

- Durante una parte significativa del desarrollo, trabajamos con la guía avanzada de *Scala Parser Combinator*. Sin embargo, dicha guía está orientada a la creación de un *parser* específico para una gramática en particular, mientras que nuestro objetivo era generar un *parser* para distintas gramáticas de manera más abstracta. Debido a esta diferencia, nos vimos obligados a apartarnos considerablemente de la guía y recurrir a la escasa documentación disponible sobre *Scala Parser Combinator*.

- Debido a la naturaleza de nuestro proyecto, fue necesario utilizar diversas herramientas, incluyendo *Haskell*, *LBNF*, *BNFC*, *Scala* y *SBT*. Esta variedad de tecnologías agregó una complejidad significativa al proceso de desarrollo, especialmente al momento de probar nuestros avances. No solo debíamos contar con una gramática en *LBNF* adecuada para las pruebas, sino también con código compatible con dicha gramática, lo que incrementó el desafío en cada etapa del desarrollo.

[1] Un generador de analizadores sintácticos (*parsers*) para *Haskell*, que *BNFC* utiliza para convertir una gramática definida en una forma específica (como *BNF*) en un código *Haskell* que puede analizar cadenas de entrada según esa gramática.

[2] Un generador de analizadores léxicos (*lexers*) también para *Haskell*. Su función es tomar una definición de patrones léxicos y generar un código que puede dividir la entrada en *tokens*

4.3 Trabajo futuro

Este trabajo estuvo acotado por nuestras propias limitaciones al adentrarnos en territorio completamente nuevo (en lo que a *Scala*, *Haskell* y *BNFC* respecta), lo cual abre varias líneas de trabajo futuro para explorar.

Entre estas podrían incluirse

- La incorporación de casos de prueba que exploren específicamente las singularidades de *Scala*.
- Agregar compatibilidad con lenguajes que usen *tabs/spaces* como definición de bloques.
- Una posible migración del proyecto de *Scala Parser combinator* a *FastParse*[3] [4] o *atto*[4] [5].
- La optimización del módulo en general.

4.4 Evaluación del proyecto

En esta sección presentamos una evaluación crítica del proyecto que desarrollamos para esta tesis: la creación de un *Parser generator* para el lenguaje *Scala*, capaz de aceptar una gramática libre de contexto escrita en *LBNF*, extendiendo *BNFC* con un módulo *backend* para dicho lenguaje.

En primer lugar, cabe preguntarnos, ¿qué utilidad tiene este proyecto?

Inicialmente, la utilidad deriva del interés de la Cátedra de Teoría de la Computación en fomentar el uso de *Scala* en sus cursos, fundamentalmente para el dictado de la materia Lenguajes y Compiladores. Contar con una extensión de *BNFC* para ese lenguaje facilitaría la transición de *Haskell* a *Scala*; fue ese interés el que motivó un trabajo exploratorio previo a esta tesis, y

[3] Una biblioteca de *Scala* para analizar cadenas y bytes en datos estructurados; permite escribir fácilmente un analizador para cualquier formato de datos textuales (p. ej., código fuente de programa, *JSON*, etc.) y ejecutarlo a una velocidad aceptable, con excelente capacidad de depuración y generación de informes de errores.

[4] Una biblioteca de análisis de texto incremental, compacta y puramente funcional para *Scala*

que derivó, finalmente, en la misma. Partiendo de esa base, reconocemos que existe un valor real, al menos para la cátedra, que justifica la existencia de este proyecto.

Por otro lado, ¿qué nos aporta a nosotros, como futuros ingenieros?

Si bien existían múltiples posibilidades a la hora de realizar una tesis, desde un principio buscábamos una experiencia distinta a la habitual en nuestros trabajos, por lo que descartamos de plano el desarrollo de una aplicación convencional. Nos interesaba un trabajo de investigación, como una suerte de despedida significativa de nuestra etapa universitaria (ya que difícilmente podríamos encarar una actividad así en nuestros roles profesionales cotidianos).

Y a pesar de que ciertos temas gozan de mayor popularidad actualmente (especialmente los relacionados con Inteligencia Artificial), abordar este desafío suponía tres aspectos significativos para nosotros:

En primer lugar, el trabajo con compiladores implicaba volver a los fundamentos, a los cimientos teóricos sobre los cuales construimos el conocimiento a lo largo de nuestra carrera. Un adecuado cierre para un ciclo académico iniciado años atrás.

En segundo lugar, dado el valor que le aporta a la cátedra, nos permitía retribuir de alguna forma a la Universidad y a los docentes que nos brindaron su dedicación durante nuestra formación profesional.

Finalmente, al tener como objetivo final la incorporación oficial del módulo de *Scala* a la herramienta *BNFC*, este trabajo abría la posibilidad de contribuir, mediante la liberación de este recurso, no solo a nuestra institución sino potencialmente a cualquier estudiante de ingeniería del mundo. Esto permite aportar a la formación de futuras generaciones de ingenieros, sumando al estado del arte de las ciencias de la computación y proporcionando una herramienta con utilidad concreta.

En definitiva, consideramos que nuestro trabajo posee un valor educativo significativo para los estudiantes de ingeniería y/o ciencias de la computación que están aprendiendo sobre teoría de lenguajes y compiladores, al proporcionar una herramienta práctica que permite vincular el

estudio de las gramáticas formales con su implementación concreta en un lenguaje moderno como *Scala*.

Desde una perspectiva técnica, este proyecto extiende la funcionalidad de *BNFC* a un ecosistema adicional, ampliando su alcance y utilidad. Creemos que el módulo tiene potencial para impactar positivamente en la enseñanza de compiladores y lenguajes de programación, ofreciendo un recurso valioso para cursos prácticos.

En conclusión, consideramos este trabajo un aporte modesto pero valioso al campo, y una experiencia profundamente satisfactoria que nos permitió consolidar conceptos técnicos, ampliar nuestro conocimiento tanto en nuevas tecnologías como en las ya conocidas, y quizás lo más significativo, tener la oportunidad de contribuir al ámbito académico que nos formó. Esta tesis representa no solo el cierre de nuestra etapa universitaria, sino también el comienzo de una nueva fase profesional donde esperamos seguir aplicando los principios y conocimientos adquiridos, manteniéndonos fieles al espíritu de investigación y mejora continua que la universidad nos inculcó.

Referencias bibliográficas

[1] A. Ranta, *Implementing Programming Languages - An introduction to Compilers and Interpreters*. College Publications, 2012.

[2] The Scala Programming Language. (2025) Tour of scala. [En línea]. Disponible en: https://docs.scala-lang.org/tour/tour-of-scala.html. [Accedido: 8 ene. 2025].

[3] P. Palma Ramos. (2016) Building a lexer and parser with scala's parser combinators. [En línea]. Disponible en: https://enear.github.io/2016/03/31/parser-combinators/. [Accedido: 11 feb. 2025].

[4] H. Li. (2022) Fastparse: Fast parser combinators for scala. [En línea]. Disponible en: https://com-lihaoyi.github.io/fastparse/. [Accedido: 3 mar. 2025].

[5] R. Norris. (2022) Atto: Compact, efficient parsers in scala. [En línea]. Disponible en: https://tpolecat.github.io/atto/. [Accedido: 3 mar. 2025].

Apéndice

A.1 Código de *BNFC* (en *Haskell*)

A.1.1 *CFtoScalaAbs.hs*

```
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE PatternGuards     #-}
{-# LANGUAGE OverloadedStrings #-}

{-
    BNF Converter: Scala Abstract syntax
    Copyright (Scala) 2024  Author:  Juan Pablo Poittevin,
 ↪  Guillermo Poladura

    Description   : This module generates the Scala Abstract
 ↪  Syntax
                    tree classes. It generates both a Header
 ↪  file
                    and an Implementation file

    Author        : Juan Pablo Poittevin, Guillermo Poladura
    Created       : 30 September, 2024
-}

module BNFC.Backend.Scala.CFtoScalaAbs (cf2ScalaAbs) where

import Prelude hiding ((<>))
```

```haskell
import BNFC.CF
import BNFC.PrettyPrint
import BNFC.Options

-- | The result is two files (.H file, .C file)
cf2ScalaAbs
  :: SharedOptions
  -> CF     -- ^ Grammar.
  -> Doc    -- ^ @.H@ file, @.C@ file.
cf2ScalaAbs Options{ lang } cf = vsep . concat $
  [
      []
    , ["package" <+> text lang]
    , [hang classSign 4 $ functions]
    , defaultFunction lang
    , ["}"]
    -- , [text $ concat $ map dataToString datas]
  ]
  where
    datas     = cf2data cf
    functions = vcat (map prData datas)

classSign :: Doc
classSign = "abstract class AbstractVisitor[R, A] extends
 ↪  AllVisitor[R, A]{"

prData ::  Data -> Doc
prData (_, rules) = vcat $ concat
  [
    [constructors rules]
```

```
  ]
  where
    prRule (fun, _)   = hsep $ concat [ ["def visit(p: " <+>
↪   text ("Absyn." ++ fun) <+> text ", arg: A): R =
↪   visitDefault(p, arg)"] ]
    constructors []     = empty
    constructors (h:t)  = sep $ [prRule h] ++ map prRule t

defaultFunction :: String -> [Doc]
defaultFunction lang =
  [
      nest 4 $ text $ "def visitDefault(p: Absyn." ++ lang ++
↪   ", arg: A): R = {"
    , nest 8 "throw new
↪   IllegalArgumentException(this.getClass.getName + \": \" +
↪   p)"
    , nest 4 "}"
  ]
```

A.1.2 *CFtoScalaLex.hs*

```
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE PatternGuards     #-}
{-# LANGUAGE OverloadedStrings #-}

{-
    BNF Converter: Scala Lextract syntax
    Copyright (Scala) 2024  Author:  Juan Pablo Poittevin,
↪   Guillermo Poladura
```

```
    Description   : This module generates the Scala Lextract
↪   Syntax
                    tree classes. Using Scala Parser Combinator
    Author        : Juan Pablo Poittevin, Guillermo Poladura
    Created       : 30 September, 2024
-}

module BNFC.Backend.Scala.CFtoScalaLex where

import Prelude hiding ((<>))
import Data.Char (ord, showLitChar)
import qualified Data.List as List

import BNFC.CF
import BNFC.Options
import BNFC.PrettyPrint
import BNFC.Utils (when, unless)
import BNFC.Backend.Common (unicodeAndSymbols)
import BNFC.Lexing (mkRegMultilineComment)
import BNFC.Abs

-- | Generate the Scala lexer code
cf2ScalaLex :: String -> CF -> Doc
cf2ScalaLex name cf = vcat $ List.intercalate [""]
    [ prelude name
    , characterClasses
    , reservedSymbols cf
    , lexerRules cf
    ]
```

```haskell
-- | File prelude with imports and package declaration
prelude :: String -> [Doc]
prelude name =
    [ "// Scala Lexer generated by the BNF Converter"
    , "package" <+> text name
    , ""
    , "import scala.util.matching.Regex"
    , "import scala.util.parsing.combinator.RegexParsers"
    , ""
    , "object ScalaLexer extends RegexParsers {"
    ]

-- | Character class definitions
characterClasses :: [Doc]
characterClasses =
    [ "// Predefined character classes"
    , "val letter  = \"[A-Za-zÀ-ÖØ-öø-ÿ]\""
    , "val capital = \"[A-ZÀ-ÖØ-Þ]\""
    , "val small   = \"[a-zß-öø-ÿ]\""
    , "val digit   = \"[0-9]\""
    , "val id      = letter + \"(\" + letter + \"|\" + digit +
 ↪  \"|[_'])*\""
    , ""
    ]

-- | Generate lexer rules for reserved symbols
reservedSymbols :: CF -> Doc
reservedSymbols cf = vcat $
    [ "// Reserved symbols and keywords"
    , "val reserved: Set[String] = Set("
```

```
    ] ++
    [ nest 2 $ hsep (punctuate "," (map show $ reservedWords
↪   cf))
    , ")"
    ] ++
    map makeSymbolRule (unicodeAndSymbols cf)
  where
    makeSymbolRule sym =
        "def" <+> text (symbolName sym) <+> "=" <+>
        text (show sym) <+> "^^^ { _ => " <> text (tokenName
↪   sym) <> " }"

-- | Generate lexer rules for comments
lexComments :: ([(String, String)], [String]) -> Doc
lexComments (block, line) = vcat $ concat
    [ map lexLineComment line
    , map (uncurry lexBlockComment) block
    ]
  where
    lexLineComment s = vcat
        [ "def lineComment =" <+> text (show s) <+> "~> \".*\"
↪   ^^^ { _ => Skip }"
        ]

    lexBlockComment start end = vcat
        [ "def blockComment =" <+> text (show start) <+> "~>"
        , nest 2 $ "rep(not(" <> text (show end) <> ") ~> any)
↪   <~"
        , nest 2 $ text (show end) <+> "^^^ { _ => Skip }"
        ]
```

```
-- | Generate all lexer rules
lexerRules :: CF -> Doc
lexerRules cf = vcat
    [ lexComments (comments cf)
    , ""
    , userDefinedTokens cf
    , defaultTokens cf
    ]
  where
    userDefinedTokens cf = vcat
        [ text name <+> "=" <+> text (printRegScala exp) <+>
          "^^^ { s => UserToken(" <> text (show name) <> ", s)
↪   }"
        | (name, exp) <- tokenPragmas cf
        ]

    defaultTokens cf = vcat $ concat
        [ whenUsed catString
            [ "// String literals"
            , "def stringLit = \"\\\"\" ~> rep(chrExcept('\"'))
↪   <~ \"\\\"\" ^^ { s => StringLit(s.mkString) }"
            ]
        , whenUsed catChar
            [ "// Character literals"
            , "def charLit = \"'\" ~> (chrExcept('\\'') |
↪   escapeSequence) <~ \"'\" ^^ { c => CharLit(c) }"
            ]
        , whenUsed catInteger
            [ "// Integer literals"
```

```
            , "def integerLit = digit+ ^^ { s =>
↪   IntLit(s.toInt) }"
            ]
        , whenUsed catDouble
            [ "// Double literals"
            , "def doubleLit = digit+ ~ \".\" ~ digit+ ~
↪   opt(\"e\" ~ opt(\"-\") ~ digit+) ^^ { case n ~ _ ~ d ~ e =>
↪   DoubleLit(s\"$n.$d${e.getOrElse(\"\")}\".toDouble) }"
            ]
        ]
      where
        whenUsed cat = when (isUsedCat cf (TokenCat cat))

-- | Print a regular expression in Scala syntax
printRegScala :: Reg -> String
printRegScala = \case
    RSeq r1 r2   -> printRegScala r1 ++ printRegScala r2
    RAlt r1 r2   -> "(" ++ printRegScala r1 ++ "|" ++
↪   printRegScala r2 ++ ")"
    RStar r      -> printRegScala r ++ "*"
    RPlus r      -> printRegScala r ++ "+"
    ROpt r       -> printRegScala r ++ "?"
    REps         -> ""
    RChar c      -> escapeChar c
    RAlts str    -> "[" ++ concatMap escapeChar str ++ "]"
    RDigit       -> "\\d"
    RLetter      -> "[A-Za-z]"
    RUpper       -> "[A-Z]"
    RLower       -> "[a-z]"
    RAny         -> "."
```

```
    RMinus r1 r2 -> "(" ++ printRegScala r1 ++ "&~" ++
↪   printRegScala r2 ++ ")"
  where
    escapeChar c | c `elem` "\\[].*+?()^${}|" = '\\' : [c]
                 | otherwise = showLitChar c ""
```

A.1.3 *CFtoScalaLexToken.hs*

```
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE PatternGuards     #-}
{-# LANGUAGE OverloadedStrings #-}

{-
    BNF Converter: Scala Lextract syntax
    Copyright (Scala) 2024  Author:  Juan Pablo Poittevin,
↪   Guillermo Poladura

    Description   : This module generates the Scala Lextract
↪   Syntax
                    tree classes. It generates both a Header
↪   file
                    and an Implementation file

    Author        : Juan Pablo Poittevin, Guillermo Poladura
    Created       : 30 September, 2024
-}

module BNFC.Backend.Scala.CFtoScalaLexToken (cf2ScalaLexToken)
↪   where
```

```haskell
import Prelude hiding ((<>))

import BNFC.CF
import BNFC.PrettyPrint
import BNFC.Options
import BNFC.Backend.Common (unicodeAndSymbols)
import BNFC.Utils (symbolToName)
import Data.Char (toUpper)
import Data.List (nub)
import BNFC.Backend.Scala.Utils (scalaReserverWords,
 ↪  mapManualTypeMap)
import Data.Maybe (fromMaybe)

cf2ScalaLexToken
  :: SharedOptions
  -> CF
  -> Doc
cf2ScalaLexToken Options{ lang } cf = vsep . concat $
  [
    headers lang
    , [text $ concat $ map generateSymbClass (symbs)]
    , [generateStringClasses liters]
    , [generateKeyWordClasses (keyWords ++ ["empty"])]
    -- , [text $ "Symbols: " ++ show symbs]
    -- , [text $ "Literals: " ++ show liters]
    -- , [text $ "Keywords: " ++ show keyWords]
  ]
  where
    liters = nub $ literals cf
```

```
    symbs = unicodeAndSymbols cf
    keyWords = reservedWords cf


generateSymbClass :: String -> String
generateSymbClass symb = case symbolToName symb of
  Just s -> "case class " ++ fromMaybe s (scalaReserverWords s)
 ↪  ++ "() extends WorkflowToken \n"
  Nothing -> mempty


generateKeyWordClasses :: [String] -> Doc
generateKeyWordClasses params = text $ concat $ map
 ↪  generateKeyWordClass params

generateKeyWordClass :: String -> String
generateKeyWordClass key = "case class " ++ param' ++ "()
 ↪  extends WorkflowToken \n"
        where
          param = map toUpper key
          param' = fromMaybe param $ mapManualTypeMap param


generateStringClasses :: [String] -> Doc
generateStringClasses params = text $ concat $ map
 ↪  generateStringClass params

generateStringClass :: String -> String
generateStringClass param = "case class " ++ (map toUpper
 ↪  param) ++ "(str: String) extends WorkflowToken \n"
```

```
headers :: String -> [Doc]
headers name = [
  text $ "package " ++ name ++ ".workflowtoken." ++ name ++
↪  "Lex"
  , "import scala.util.parsing.input.Positional"
  , "sealed trait WorkflowToken extends Positional"
  ]
```

A.1.4 *CFtoScalaParserAST.hs*

```
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE PatternGuards     #-}
{-# LANGUAGE OverloadedStrings #-}

{-
    BNF Converter: Scala Lextract syntax
    Copyright (Scala) 2024  Author:  Juan Pablo Poittevin,
↪  Guillermo Poladura

    Description   : This module generates the Scala Lextract
↪  Syntax
                    tree classes. It generates both a Header
↪  file
                    and an Implementation file

    Author        : Juan Pablo Poittevin, Guillermo Poladura
    Created       : 30 September, 2024
-}
```

```haskell
module BNFC.Backend.Scala.CFtoScalaParserAST
 ↪  (cf2ScalaParserAST, getASTNames) where

import Prelude hiding ((<>))

import BNFC.CF
    ( ruleGroups,
      CF,
      Cat(ListCat),
      IsFun(funName),
      Rul(Rule),
      Rule, TokenCat, literals, wpThing )
import BNFC.PrettyPrint ( text, vcat, Doc )
import BNFC.Options ( SharedOptions(lang, Options) )
import BNFC.Backend.Scala.Utils (generateVarsList, isLeft,
 ↪  baseTypeToScalaType, wrapList, scalaReserverWords,
 ↪  isCoercionRule)
import Data.List (intercalate)
import BNFC.Utils ((+++))
import Data.Maybe (fromMaybe)
import GHC.OldList (nub)

-- | Main function that generates the AST code
cf2ScalaParserAST :: SharedOptions -> CF -> Doc
cf2ScalaParserAST Options{ lang } cf = vcat $
  -- Generate headers
  headers lang ++
  -- Add an empty line after the trait definition
  [text ""] ++
```

```
  -- Generate case class definitions
  generateRuleDefs rules ++
  generateLiteralsDefs allLiterals
  where
    rules = ruleGroups cf
    allLiterals = nub $ literals cf

getASTNames :: [Rule] -> [String]
getASTNames rules = rulesNames
  where
    rulesNames = map (\(Rule fun _ _ _) -> fromMaybe (funName
↪   fun) $ scalaReserverWords $ funName fun) $
          filter (\(Rule _ cat _ _) -> not $ case wpThing cat
↪   of ListCat _ -> True; _ -> False) filteredRules
    filteredRules = filter (not . isCoercionRule) $ rules

-- | Generate all case class definitions
generateRuleDefs :: [(Cat, [Rule])] -> [Doc]
generateRuleDefs [] = []
generateRuleDefs rules = concatMap processRuleGroup rules

-- | Process a single rule group and generate case classes
processRuleGroup :: (Cat, [Rule]) -> [Doc]
processRuleGroup (_, rules) = map createCaseClass (filter (not
↪   . isCoercionRule) rules)


-- | Generate the class params
generateClassParams :: Rule -> String
generateClassParams (Rule _ _ rhs _) =
```

```haskell
  intercalate ", " $ zipWith (\x y -> x ++ ":" +++ y)
↪  (generateVarsList filteredRhs) (map catParams filteredRhs)
  where
    -- Function to format parameters based on whether they are
↪  Cat or String
    catParams :: Either Cat String -> String
    catParams (Left c)  = formatParamType c
    catParams (Right _) = "WorkflowAST"

    filteredRhs = filter isLeft rhs

-- | Format a parameter with its type
formatParamType :: Cat -> String
formatParamType cat = wrapList cat "WorkflowAST"

-- | Create a single case class definition
createCaseClass :: Rule -> Doc
createCaseClass rule@(Rule fun cat _ _)
  | ListCat _ <- (wpThing cat) = mempty
  | otherwise = text $ formatCaseClass className params
  where
    className = fromMaybe (funName fun) $ scalaReserverWords $
↪  funName fun
    params = generateClassParams rule

-- | Helper function to format the case class definition
formatCaseClass :: String -> String -> String
formatCaseClass className params = "case class " ++ className
↪  ++ "(" ++ params ++ ") extends WorkflowAST"
```

```haskell
-- | Generate the Scala types for basic LBNF types
generateLiteralsDefs :: [TokenCat] -> [Doc]
generateLiteralsDefs tokens = map (
    \token -> text $ formatCaseClass ("p" ++ token) ("var1: "
 ↪  ++ fromMaybe "String" (baseTypeToScalaType token))
  ) tokens

-- | Generate the header part of the file
headers :: String -> [Doc]
headers name = [
  text $ "package " ++ name ++ ".workflowtoken." ++ name ++
 ↪  "Parser",
  text "",
  text "import scala.util.parsing.input.Positional",
  text "",
  text "sealed trait WorkflowAST extends Positional"
  ]
```

A.1.5 *CFtoScalaParser.hs*

```haskell
{-# LANGUAGE LambdaCase        #-}
{-# LANGUAGE PatternGuards     #-}
{-# LANGUAGE OverloadedStrings #-}

{-
    BNF Converter: Scala Parser syntax
    Copyright (Scala) 2024  Author:  Juan Pablo Poittevin,
 ↪  Guillermo Poladura
```

```
    Description   : This module generates the Scala Parser
 ↪  Syntax
                    Using Scala Parser Combinator
    Author        : Juan Pablo Poittevin, Guillermo Poladura
    Created       : 30 September, 2024
-}

module BNFC.Backend.Scala.CFtoScalaParser (cf2ScalaParser)
 ↪  where

import qualified Data.Foldable as DF (toList)
import Prelude hiding ((<>))

import BNFC.Backend.Scala.Utils (safeCatName, hasTokenCat,
 ↪  rhsToSafeStrings, disambiguateNames, getRHSCats,
 ↪  isSpecialCat, isListCat, isLeft, safeHeadChar,
 ↪  wildCardSymbs, scalaReserverWords, mapManualTypeMap,
 ↪  isCoercionCategory, getTerminalFromListRules, isSymbol)
import BNFC.CF
    ( allEntryPoints,
      catChar,
      catIdent,
      catInteger,
      catString,
      isNilCons,
      normCat,
      ruleGroups,
      rulesForNormalizedCat,
      sameCat,
      sortRulesByPrecedence,
```

```
      CF,
      Cat,
      IsFun(isCoercion, funName),
      Rul(Rule, valRCat, rhsRule),
      Rule,
      TokenCat,
      WithPosition(wpThing), strToCat, catDouble, rulesForCat )
import BNFC.PrettyPrint
import BNFC.Options ( SharedOptions(lang, Options) )
import BNFC.Backend.Common.NamedVariables (firstLowerCase,
↪  fixCoercions)

import Data.List (find, intercalate, nub)
import Data.Char (toLower)
import Data.Maybe (listToMaybe, fromMaybe, isJust)
import BNFC.Utils ((+++), symbolToName)
import BNFC.Backend.Scala.CFtoScalaParserAST (getASTNames)
import GHC.Unicode (toUpper)

-- | Main function that generates the Scala parser code
cf2ScalaParser :: SharedOptions -> CF -> Doc
cf2ScalaParser Options{ lang } cf = vcat $
  -- Generate header and imports
  imports lang ++
  -- Start the WorkflowParser object
  initWorkflowClass ++
  -- Indent the class contents
  [nest 4 $ vcat $
    -- Add extra classes
    addExtraClasses ++
```

```
    [text ""] ++
    -- Add apply function
    getApplyFunction ++
    [text ""] ++
    -- Add program function
    getProgramFunction cf ++
    [text ""] ++
    -- Add parser rules
    generateAllRules cf (ruleGroups cf) ++
    [text ""]
    -- inspect rules, only for debugging
    -- ++ inspectListRulesByCategory (ruleGroups cf)
  ] ++
  -- End the WorkflowParser object
  endWorkflowClass ++
  -- Add the WorkflowCompiler object
  addWorkflowCompiler

-- | Generate all parser rules from rule groups
generateAllRules :: CF -> [(Cat, [Rule])] -> [Doc]
generateAllRules cf catsAndRules =
  let
    -- filter categories with all the rules isNilCons
    rulesToProcess = filter (not . all isNilCons . snd)
↪   catsAndRules
    -- Generate regular rules
    mainRules = map text $ concatMap (generateRuleGroup cf)
↪   (fixCoercions rulesToProcess)

    -- existe isUsedCat seguramente nos simplifique esto
```

```
    -- existe specialCats seguramente nos simplifique esto
    -- existe sigLookup wtf con esto
    -- Generate special rules for integer and string if needed
    integerRule = generateSpecialRule catInteger "integer"
↪   "INTEGER" "toInt" "pInteger" catsAndRules
    doubleRule = generateSpecialRule catDouble "double"
↪   "DOUBLE" "toInt" "pDouble" catsAndRules
    stringRule = generateSpecialRule catString "string"
↪   "STRING" "toString" "pString" catsAndRules
    charRule = generateSpecialRule catChar "char" "CHAR"
↪   "toString" "pChar" catsAndRules
    identRule = generateSpecialRule catIdent "ident" "IDENT"
↪   "toString" "pIdent" catsAndRules

  in mainRules ++ integerRule ++ stringRule ++ charRule ++
↪   identRule ++ doubleRule


getRuleFunName :: Cat -> Rule -> String
getRuleFunName cat (Rule fnam _ _ _) = firstLowerCase $
 ↪  prefixIfNeeded $ funName fnam
  where
    prefixIfNeeded name
      | map toLower name == map toLower (safeCatName cat) ||
↪   isJust (scalaReserverWords (map toLower name))  =
↪   "internal_" ++ map toLower name
      | otherwise = name

getRulesFunsName :: [Rule] -> Cat -> [String]
```

```haskell
getRulesFunsName rules cat = nub $ map (getRuleFunName cat)
 ↪  rules

-- | Generate a rule group (definition for a single category)
generateRuleGroup :: CF -> (Cat, [Rule]) -> [String]
generateRuleGroup cf (cat, rules) =
  ["def " ++ catName ++ ": Parser[WorkflowAST] = positioned {"]
 ↪  ++
  [replicate 4 ' ' ++ intercalate " | " subFuns] ++
  ["}"] ++
  concatMap (generateSingleRuleBody cf cat) nonCoercionRules
  where
    subFuns = getRulesFunsName nonCoercionRules cat
    catName = safeCatName cat
    nonCoercionRules = reverse $ map snd $
 ↪  sortRulesByPrecedence $ filter (not . isCoercion) rules

generateSingleRuleBody :: CF -> Cat -> Rule -> [String]
generateSingleRuleBody cf cat rule = [generateRuleDefinition cf
 ↪  cat rule ++ generateRuleTransformation rule]

generateRuleDefinition :: CF -> Cat -> Rule -> String
generateRuleDefinition cf cat rule =
  "def " ++ getRuleFunName cat rule ++ ": Parser[WorkflowAST]
 ↪  ="
    +++ intercalate " ~ " (generateRuleForm cf rule)


getBaseCatOfRecursiveRule :: Rule -> [String]
getBaseCatOfRecursiveRule rule@(Rule _ _ rhs _) =
```

```
  nub $ concatMap extractBaseCat rhs
  where
    -- Extrae las categorías base de un elemento del RHS
    extractBaseCat :: Either Cat String -> [String]
    extractBaseCat (Left cat)
      | isBaseCat cat = [getRuleFunName cat rule]
      | otherwise = []
    extractBaseCat (Right s) = if isSymbol s then [] else
↪   [getRuleFunName (strToCat s) rule]

    isBaseCat :: Cat -> Bool
    isBaseCat cat = isSpecialCat $ normCat cat


getBasesOfRecursiveRule :: CF -> Rule -> String
getBasesOfRecursiveRule cf rule =
  let
    allRulesForCat = rulesForNormalizedCat cf (normCat $
↪   wpThing $ valRCat rule)
    baseTypes = nub $ concatMap getBaseCatOfRecursiveRule
↪   allRulesForCat
  in
    case baseTypes of
      [] -> ""
      x:[] -> x
      x:xs -> "(" ++ intercalate " | " (x:xs) ++ ")"


generateRuleForm :: CF -> Rule  -> [String]
generateRuleForm cf rule@(Rule _ _ rhs _) =
```

```
  if isRecursiveRule rule
    then snd $ generateRecursiveRuleForm rhs False
    else case rhs of
      [Right s] -> [fromMaybe (paramS s) (mapManualTypeMap
↪   (paramS s)) ++ "()"]
      _ -> map (addRuleForListCat cf rhs) (rhsToSafeStrings
↪   rhs)
  where
    generateRecursiveRuleForm :: [Either Cat String] -> Bool ->
↪   (Bool, [String])
    generateRecursiveRuleForm [] added = (added, [])
    generateRecursiveRuleForm (r:rest) added =
      case r of
        Left cat ->
          if isCoercionCategory cat && not added
            then
              let (_, strsRest) = generateRecursiveRuleForm
↪   rest True
              in (True, getBasesOfRecursiveRule cf rule :
↪   strsRest)
            else
              let (addedRest, strsRest) =
↪   generateRecursiveRuleForm rest added
              in (addedRest, rhsToSafeStrings [r] ++ strsRest)
        Right _ ->
          let (addedRest, strsRest) = generateRecursiveRuleForm
↪   rest added
          in (addedRest, rhsToSafeStrings [r] ++ strsRest)

    paramS s = fromMaybe (map toUpper s) (symbolToName s)
```

```
generateRuleTransformation :: Rule -> String
generateRuleTransformation rule =
  if isRuleOnlySpecials rule
    then case rhsRule rule of
      [] -> "EMPTY() ^^ {" ++ generateCaseStatement rule ++ "}"
      _ -> ""
    else " ^^ { " ++ generateCaseStatement rule ++ " }"

isRecursiveRule :: Rule -> Bool
isRecursiveRule (Rule _ cat rhs _) =
  any (sameCat (wpThing cat)) (getRHSCats rhs)

addRuleForListCat :: CF -> [Either Cat String] -> String ->
 ↪  String
addRuleForListCat cf rhs s =
  case find (\cat -> isListCat cat && safeCatName cat == s)
 ↪  (getRHSCats rhs) of
    Just c -> case (getTerminalFromListRules $ rulesForCat cf
 ↪  c) of
      Just term -> "repsep(" ++ firstLowerCase s ++ ", " ++
 ↪  term ++ "())"
      _         -> "rep(" ++ firstLowerCase s ++ ")"
    Nothing -> s

isRuleOnlySpecials :: Rule -> Bool
isRuleOnlySpecials (Rule _ _ rhs _) =
  all isSpecialCat (getRHSCats rhs) && all isLeft rhs

-- -- | Generate a case statement for a rule
```

```haskell
generateCaseStatement :: Rule -> String
generateCaseStatement rule@(Rule fun c rhs _)
  | isCoercion fun = ""
  | null vars = "_ => " ++ fnm ++ "()"
  | otherwise =
      "case (" ++ intercalate " ~ "  params ++ ") => "
      ++ fnm ++ "(" ++ intercalate ", " vars ++ ")"
  where
    getBaseType :: Cat -> String
    getBaseType cat
      | isListCat cat = "List[WorkflowAST]"
      | otherwise = "WorkflowAST"

    fnm = fromMaybe (getRuleFunName (wpThing c) rule) $
↪   listToMaybe $ getASTNames [rule]

    -- generate a list of with (rule, finalName)
    zipped = zip rhs (disambiguateNames $ map getSymb rhs)

    getSymb (Left cat) = [toLower $ safeHeadChar $ safeCatName
↪   cat]
    getSymb (Right str) = [toLower $ safeHeadChar $ fromMaybe
↪   "_" $ symbolToName str]

    params = map (wildCardSymbs.snd) zipped

    -- For Left cat we assign types, for Right str (tokens), we
↪   ignore (or use "_")
    vars = [ p ++ ".asInstanceOf[" ++ getBaseType cat ++ "]"
           | (Left cat, p) <- zipped
```

```
          ]

-- | Generate a special rule for tokens like Integer or String
generateSpecialRule :: TokenCat -> String -> String -> String
 ↪  -> String -> [(Cat, [Rule])] -> [Doc]
generateSpecialRule tokenCat ruleName tokenName conversion
 ↪  pTypeName catsAndRules =
  case find (\(_, rules) -> any (hasTokenCat tokenCat) rules)
 ↪  catsAndRules of
    Just (_, rules) -> case find (hasTokenCat tokenCat) rules
 ↪  of
      Just _ ->
        let
          conversionPart = if null conversion then "" else "."
 ↪  ++ conversion
        in
          [ text $ "def " ++ ruleName ++ ": Parser[" ++
 ↪  pTypeName ++ "] = {"
          , nest 4 $ text $ "accept(\"" ++ ruleName ++ "\", {
 ↪  case " ++ tokenName ++ "(i) => " ++ pTypeName ++ "(i" ++
 ↪  conversionPart ++ ") })"
          , text "}"
          ]
      Nothing -> []
    Nothing -> []

-- | Generate the imports section
imports :: String -> [Doc]
imports name = [
```

```
  text $ "package " ++ name ++ ".workflowtoken." ++ name ++
↪   "Parser",
  text "",
  text $ "import " ++ name ++ ".workflowtoken." ++ name ++
↪   "Lex._",
  text "",
  text "import scala.util.parsing.combinator.Parsers",
  text "",
  text "import scala.util.parsing.input.{NoPosition, Position,
↪   Reader}"
  ]

-- | Generate the extra classes section
addExtraClasses :: [Doc]
addExtraClasses = [
  text "override type Elem = WorkflowToken",
  text "",
  text "class WorkflowTokenReader(tokens: Seq[WorkflowToken])
↪   extends Reader[WorkflowToken] {",
  nest 4 $ text "override def first: WorkflowToken =
↪   tokens.head",
  nest 4 $ text "override def atEnd: Boolean = tokens.isEmpty",
  nest 4 $ text "override def pos: Position =
↪   tokens.headOption.map(_.pos).getOrElse(NoPosition)",
  nest 4 $ text "override def rest: Reader[WorkflowToken] = new
↪   WorkflowTokenReader(tokens.tail)",
  text "}"
  ]

-- | Generate the WorkflowCompiler object
```

```haskell
addWorkflowCompiler :: [Doc]
addWorkflowCompiler = [
  text "",
  text "object WorkflowCompiler {",
  nest 4 $ text "def apply(code: String):
 ↪  Either[WorkflowCompilationError, WorkflowAST] = {",
  nest 8 $ text "for {",
  nest 12 $ text "tokens <- WorkflowLexer(code).right",
  nest 12 $ text "ast <- WorkflowParser(tokens).right",
  nest 8 $ text "} yield ast",
  nest 4 $ text "}",
  text "}"
  ]

-- | Start the WorkflowParser object
initWorkflowClass :: [Doc]
initWorkflowClass = [
  text "",
  text "object WorkflowParser extends Parsers {"
  ]

-- | End the WorkflowParser object
endWorkflowClass :: [Doc]
endWorkflowClass = [
  text "}"
  ]

-- | Generate the apply function
getApplyFunction :: [Doc]
getApplyFunction = [
```

```
  text "",
  text "def apply(tokens: Seq[WorkflowToken]):
↪  Either[WorkflowParserError, WorkflowAST] = {",
  nest 4 $ text "val reader = new WorkflowTokenReader(tokens)",
  nest 4 $ text "program(reader) match {",
  nest 8 $ text "case NoSuccess(msg, next) =>
↪  Left(WorkflowParserError(Location(next.pos.line,
↪  next.pos.column), msg))",
  nest 8 $ text "case Success(result, next) => Right(result)",
  nest 4 $ text "}",
  text "}"
  ]

-- | Generate the program function
getProgramFunction :: CF -> [Doc]
getProgramFunction cf = [
  text "",
  text "def program: Parser[WorkflowAST] = positioned {",
  nest 4 $ text $ "phrase(" ++ fromMaybe entryPoint
↪  (scalaReserverWords entryPoint) ++ ")",
  text "}"
  ]
  where
    entryPoint = case listToMaybe (map normCat $ DF.toList $
↪  allEntryPoints cf) of
      Just ep -> firstLowerCase $ show ep
      Nothing -> error "No entry points found in the
↪  context-free grammar."
```

A.2 Código generado (en *Scala*)

A.2.1 *CalcLex.scala*

```scala
// File generated by the BNF Converter (bnfc 2.9.6).

package Calc.workflowtoken.CalcLex

import scala.util.parsing.combinator.RegexParsers

sealed trait WorkflowCompilationError

case class WorkflowLexerError(location: Location, msg: String)
 ↪  extends WorkflowCompilationError

case class WorkflowParserError(location: Location, msg: String)
 ↪  extends WorkflowCompilationError

case class Location(line: Int, column: Int) {

    override def toString = s"$line:$column"

}

object WorkflowLexer extends RegexParsers {

    def apply(code: String): Either[WorkflowLexerError,
     ↪  List[WorkflowToken]] = {

        parse(tokens, code) match {
```

```
      case NoSuccess(msg, next) =>
       ↪  Left(WorkflowLexerError(Location(next.pos.line,
       ↪  next.pos.column), msg))

      case Success(result, next) => Right(result)

    }

}

def tokens: Parser[List[WorkflowToken]] = {

    phrase(rep1( plus | minus | star | slash | lparen |
     ↪  rparen | integer))

}

def integer: Parser[INTEGER] = {
    "[0-9]+".r ^^ {i => INTEGER(i)}
}

def plus = positioned { "+" ^^ (_ =>PLUS()) }

def minus = positioned { "-" ^^ (_ =>MINUS()) }

def star = positioned { "*" ^^ (_ =>STAR()) }

def slash = positioned { "/" ^^ (_ =>SLASH()) }
```

```
    def lparen = positioned { "(" ^^ (_ =>LPAREN()) }

    def rparen = positioned { ")" ^^ (_ =>RPAREN()) }

}
```

A.2.2 *CalcLexToken.scala*

```
// File generated by the BNF Converter (bnfc 2.9.6).

package Calc.workflowtoken.CalcLex

import scala.util.parsing.input.Positional

sealed trait WorkflowToken extends Positional

case class PLUS() extends WorkflowToken
case class MINUS() extends WorkflowToken
case class STAR() extends WorkflowToken
case class SLASH() extends WorkflowToken
case class LPAREN() extends WorkflowToken
case class RPAREN() extends WorkflowToken

case class INTEGER(str: String) extends WorkflowToken
```

A.2.3 *CalcParser.scala*

```
// File generated by the BNF Converter (bnfc 2.9.6).

package Calc.workflowtoken.CalcParser
```

```
import Calc.workflowtoken.CalcLex._

import scala.util.parsing.combinator.Parsers

import scala.util.parsing.input.{NoPosition, Position, Reader}

object WorkflowParser extends Parsers {

    override type Elem = WorkflowToken

    class WorkflowTokenReader(tokens: Seq[WorkflowToken])
     ↪  extends Reader[WorkflowToken] {
        override def first: WorkflowToken = tokens.head
        override def atEnd: Boolean = tokens.isEmpty
        override def pos: Position =
         ↪  tokens.headOption.map(_.pos).getOrElse(NoPosition)
        override def rest: Reader[WorkflowToken] = new
         ↪  WorkflowTokenReader(tokens.tail)
    }

    def apply(tokens: Seq[WorkflowToken]):
     ↪  Either[WorkflowParserError, WorkflowAST] = {
        val reader = new WorkflowTokenReader(tokens)
        program(reader) match {
            case NoSuccess(msg, next) =>
             ↪  Left(WorkflowParserError(Location(next.pos.line,
             ↪  next.pos.column), msg))
            case Success(result, next) => Right(result)
        }
    }
```

```
def program: Parser[WorkflowAST] = positioned {
    phrase(exp)
}

def exp: Parser[WorkflowAST] = positioned {
    exp1 ~ rep((PLUS() | MINUS()) ~ exp1) ^^ {
        case exp1 ~ list => list.foldLeft(exp1) {
            case (e1, PLUS() ~ e2) => EAdd(e1, e2)
            case (e1, MINUS() ~ e2) => ESub(e1, e2)
        }
    }
}

def exp1: Parser[WorkflowAST] = positioned {
    exp2 ~ rep((STAR() | SLASH()) ~ exp2) ^^ {
        case exp2 ~ list => list.foldLeft(exp2) {
            case (e1, STAR() ~ e2) => EMul(e1, e2)
            case (e1, SLASH() ~ e2) => EDiv(e1, e2)
        }
    }
}

def exp2: Parser[WorkflowAST] = positioned {
    integer
}

def integer: Parser[EInt] = positioned {
    accept("integer", { case INTEGER(i) => EInt(i.toInt) })
}
```

```
}
object WorkflowCompiler {
    def apply(code: String): Either[WorkflowCompilationError,
     ↪  WorkflowAST] = {
        for {
            tokens <- WorkflowLexer(code).right
            ast <- WorkflowParser(tokens).right
        } yield ast
    }
}
```

A.2.4 *CalcParserAST.scala*

```
// File generated by the BNF Converter (bnfc 2.9.6).

package Calc.workflowtoken.CalcParser

import scala.util.parsing.input.Positional

sealed trait WorkflowAST extends Positional


case class EAdd(exp: WorkflowAST, exp1: WorkflowAST) extends
 ↪  WorkflowAST
case class ESub(exp: WorkflowAST, exp1: WorkflowAST) extends
 ↪  WorkflowAST
case class EMul(exp1: WorkflowAST, exp2: WorkflowAST) extends
 ↪  WorkflowAST
case class EDiv(exp1: WorkflowAST, exp2: WorkflowAST) extends
 ↪  WorkflowAST
```

```
case class EInt(exp2: Int) extends WorkflowAST
```