Implementación de un Parser y Analizador Semántico Para el Lenguaje Little Duck

Hiram Muñoz

Introducción

En el presente trabajo se documenta el proceso de diseño e implementación de un compilador para el lenguaje de programación llamado "LittleDuck". Este es un lenguaje de programación de aprendizaje simple y rígido, hecho para introducir los conceptos y mecanismos clave de un compilador.

El documento actualmente contiene las secciones relevantes al diseño del lenguaje y la gramática, el análisis sintáctico, el análisis léxico y el análisis semántico, lo que produce un aplicativo capaz de decir si un dado código de Little Duck es válido.

Lenguaje

Little Duck es un lenguaje de programación imperativo, con una estructura rígida. El lenguaje está intencionalmente limitado en sus capacidades, esto con el objetivo de simplificar su implementación. Está compuesto por varios tipos de entidades, cada una realizando operaciones específicas dentro de un programa. Estas son los que le permiten al lenguaje realizar trabajo útil. Hay cuatro tipos de estas entidades:

- Símbolos: representan variables y procedimientos definidos por el programador.
- Expresiones: representan un valor específico, que se resuelve al ser evaluado por la máquina.
- Estatutos: son instrucciones que se le dan a la máquina para realizar trabajo.
- Palabras clave: son delimitadores que le dan estructura al programa

Estas entidades son las que constituyen un programa completo de Little Duck.

Estructura de un programa

El siguiente bloque de código muestra un programa válido de Little Duck:

```
program HelloWorld;
// Preámbulo

x: int; // soporte para variables

void foo() [
    a: int; // variables locales en una función
    {
        x = 5; // cuerpo de la función
    }
];

void bar(y: int) [
    {
        if (y > 0) { // condiciones
            x = y;
        };
    }
];

// Cuerpo
main {
    // cuerpo principal de una función

    bar(9);
    bar(-239);

    print(x);
}

end
```

En este programa, se puede observar varias de las características clave del lenguaje. Una de estas es la división del programa en dos secciones principales: el preámbulo y el cuerpo.

Preámbulo

En Little Duck, todos los símbolos a utilizar se deben definir en la parte inicial del programa, antes de cualquier estatuto de código ejecutable (que realiza trabajo). Todo esto se realiza en la parte inicial del programa, el preámbulo.

El preámbulo contiene todas las declaraciones de símbolos que serán utilizadas en el programa. Este empieza después de establecer el nombre del programa y se divide en dos partes: la definición de variables globales, y la definición de funciones, como en el siguiente ejemplo:

```
program HelloWorld;
// Preámbulo

x: int; // soporte para variables
y: bool;

float foo() [
    a: int; // variables locales en una función
    {
        x = 5; // cuerpo de la función
        return 44.2;
    }
];

// Cuerpo
main {
// ...cuerpo
```

Para definir una variable, primero se define un nombre para esta, este es el identificador de la variable. Una definición puede contener uno o más identificadores, con lo que se puede definir varias variables del mismo tipo a la vez. Después de esto, se establece el tipo de la variable. El siguiente bloque de código son ejemplos de como definir variables.

```
x: int; // Definición única

x, y: float; // Definición múltiple
```

La declaración de funciones primero inicia con un tipo como float, int, string o bool o la palabra void (indicando que no regresa un valor). Se le asigna un nombre a esta función, y ya con esto se abren corchetes, lo que inicia el cuerpo de la función. Una función puede tener su propio bloque de declaración de variables internas, únicamente visibles dentro del cuerpo de la función. Después de este bloque, sigue el cuerpo de la función, delimitado por corchetes. Más adelante se explicará el contenido del bloque de cuerpo. El siguiente bloque de código muestra ejemplos de funciones.

```
// función con una variable sin argumentos
int foo() [
    x: int;
    {
        x = 5;
        return x;
    }
];

// función sin variables pero con un argumento
int zeroOrMore(a: int) [
```

```
    {
        if (a > 0) {
            return a;
        }
        return 0;
    }
];
```

Cuerpo

La segunda parte de un programa es el cuerpo. Este se encuentra delimitado entre las palabras main y end. Es aquí donde se encuentra la entrada al programa, y a partir de aquí se pueden realizar todas las operaciones de este. A continuación se muestra un ejemplo de dos cuerpos, el de una función y del programa en general:

```
// ...preámbulo
void bar(a: int) [
    x: int
    {
        // Cuerpo de función
        x = 2
        if (a > 0) {
            x = a;
        };
        return x
    }
];
// cuerpo
main {
    // cuerpo principal

    bar(9);
    bar(-239);

    print(x);
}

end
```

Dentro del cuerpo, se reconocen 6 tipos de estatutos diferentes a evaluar por la máquina. Estos son los siguientes:

- Asignación: `a = 12`
- Condición: `if (x > 2) { print(x); }`
- Ciclos: `while { print(x); x = x + 1; } do(x < 10);`
- Invocaciones: `foobar(x, b, 2.3)`
- Impresión a consola: `print(x)`

- Retorno: `return 23;`

Cada uno de estos estatutos son instrucciones directas (imperativas) sobre lo que debe hacer la computadora al encontrarlas. Todas estas instrucciones requieren de datos para realizarse. Para esto se utilizan las expresiones.

Expresiones

Las expresiones son maneras de *expresar* un valor a ser utilizado dentro del programa. Todos los valores de Little Duck caen dentro de 4 tipos diferentes: booleanos, números flotantes, enteros y cadenas de texto. Estos valores pueden ser expresados como "literales". Algunos ejemplos se muestran a continuación.

```
// Booleanos
true
false

// Enteros
52
-12

// Floats
231.4
-12.2

// Strings
"hola"
"mundo!"
```

El lenguaje también permite utilizar un símbolo como un valor. Para hacer esto, se requiere revisar los símbolos que han sido definidos en el preámbulo principal o de la función actual, con lo que obtiene si un símbolo existe, asi como su tipo, con lo cual se puede evaluar si se está usando en un contexto adecuado. Si un símbolo es una función, este también ocupa ser "invocado" para poder ser usado como valor. A continuación se muestran ejemplos de expresiones más complejas.

```
2 + 3
x + 44 / 2
-35 * 9
"hello" + " " + "world!"
foo("bar") > 41
```

Siguiente paso

El siguiente paso es implementar un programa que sea capaz de realizar el análisis léxico, sintáctico, y semántico para así poder compilar un programa escrito en LittleDuck. El lenguaje es muy simple, por lo cual yo no quería implementar

esto usando algúna tecnología común como ANTLR. Por lo tanto, opté por una alternativa menos conocida.

Implementación del analizador léxico y sintáctico

El *parser* es el programa encargado de analizar texto y construir un *abstract syntax tree* (AST), con el cual se pueda hacer más procesamiento sobre un programa dado. Existen muchas herramientas y tecnologías para implementar parsers, como ANTLR y Bison. Sin embargo, no me atraía utilizar una de estas herramientas, sobre todo porque me obligaría a usar lenguajes como JavaScript, Python o C++ para implementar el lenguaje. Por lo tanto, yo opté por una tecnología alternativa: decidí implementar el parser para el lenguaje LittleDuck usando el lenguaje de programación F# mediante la librería de FParsec.

F#

F# es un lenguaje de programación multi-paradigma, con un enfoque en el paradigma funcional. Está construido sobre el Common Language Interface de Microsoft, característica que comparte con los lenguajes de programación de C# y VB.NET. Esto lo hace parte de la plataforma .NET, dándole acceso a todas las librerías utilizadas dentro de esta. Asimismo, tiene su propio ecosistema de librerías escritas en el paradigma funcional, optimizadas para su uso dentro del lenguaje. Es open-source, multiplataforma, y cuenta con una multitud de herramientas de desarrollo profesional con las cuales se pueden desarrollar muchos tipos de programas, desde servicios web y servicios de procesamiento de datos, hasta sitios web y aplicaciones de escritorio.

El enfoque del lenguaje se encuentra dentro del paradigma de programación funcional. Dentro de este paradigma, los programas se construyen mediante la construcción, aplicación y composición de funciones, donde cada función funciona como un mapeo de expresiones que convierten valores a otros valores. Esto es en contraste al paradigma imperativo (como el de Little Duck, o JavaScript y Python), los cuales actualizan un estado mutable. Uno de los pilares de la programación funcional es el intentar mantener funciones puras. Esto significa que la función, dada ciertas entradas, siempre tiene que definir las mismas salidas, sin efectos secundarios. De esta manera, funciones pequeñas pueden ser compuestas con facilidad para construir programas que expresen comportamientos

Yo ya he tenido exposición al lenguaje anteriormente, con proyectos de desarrollo back-end para páginas web, por lo cual ya tenía el interés desde antes. Sin embargo, estas aplicaciones son altamente prácticas, lo que necesita un cierto nivel de prágmatismo al programar. Esto involucra sacrificar algunos aspectos de la programación funcional. Es por esto, que al verme con la oportunidad de programar algo de naturaleza más teorética, como lo es un compilador, decidí que lo iba a implementar en F#, buscando utilizar únicamente herramientas del mundo de la programación funcional. Es aquí donde entra FParsec.

FParsec

FParsec es una librería de F# basada en la librería Parsec, de Haskell. Esta librería se autodescribe como una librería de combinación de parsers, donde un parser es definido como una combinación de otros parsers. Esto le permite a su API ser sencilla, elegante y comprobable, con lo cual se puede asegurar el correcto funcionamiento de un parser. Se definen los parsers declarativamente, permitiendo un sencillo reuse del código. Aprovechando las funcionalidades de tipos algebraicos disponibles en F#, se puede construir un AST a la medida muy fácilmente.

Tipos algebraicos

Dentro de FParsec, antes de poder definir un parser se requiere tener un tipo que pueda representar a este elemento del lenguaje. Esto le permite al compilador ir aglomerando los resultados del proceso de análisis conforme va leyendo el texto, generando lo que sería el *Abstract Syntax Tree* del lenguaje. F# cuenta con un mecanismo muy intuitivo para esto: los tipos algebraicos.

En sí, un tipo algebraico podría ser considerado como un AND sobre diferentes "propiedades". Como ejemplo, un carro podría ser definido como teniendo: número de llantas AND color AND marca. Esto se podría representar en F# de la siguiente manera:

```
type Car = {
    tireCount: int
    color: Color
    brand: Brand
}
```

Estas estructuras de tipo AND tienen el nombre de **registros** (*records*).

Asímismo, se podría definir que los colores puede ser uno de varios colores y las marcas son realmente varias marcas, combinadas por medio de un OR. Así se presentaría esto en F#:

```
type CarBrand =
    | Audi
    | Mercedes
    | BMW

type Color =
    | Red
    | Green
    | Blue
```

Estas estructura de tipo OR son Uniones Discriminadas. Estas no tiene que necesariamente estar vacías, pueden contener más sub-uniones o sub-registros, que

pueden estar infinitamente anidados. Esto le permite a F# modelar fácilmente "árboles" de diferentes entidades, por medio de una composición clara de tipos.

Identificadores

Ya con esto en mente, se presenta el ejemplo del primer parser que tuve que realizar: los identificadores. Estos son las cadenas de texto que identifican a un símbolo, ya sea una variable, una función, un argumento o el nombre del programa. Todos los identificadores deben ser de la siguiente forma:

```
[^\d\\+\-/*><=\s".!:,()\[\];{}][^\\+\-/*><=\s".!:,()\[\];{}]*
```

Esto básicamente excluye a todos los caracteres que representan algún operador dentro del lenguaje. Para poder implementar esto dentro de FParsec, primero se necesita elaborar un tipo que para contener esto dentro del AST. Este es definido como un simple contenedor de una string básica, con lo cual se marca que este valor de texto representa un identificador.

```
type IdentifierNode = IdentifierNode of string
```

La implementación del parser en sí se ve de la siguiente manera:

```
open Ast
open FParsec

module Identifier =
    let parse =
        regex "[^\d\\+\-/*><=\s\".!:,()[\];{}][^\\+\-/*><=\s\".!:,()[\];{}]*"
        |>> IdentifierNode
        <?> "identifier"
```

Las primeras líneas, `open Ast` y `open FParsec` importan las librerías requeridas para el ejemplo. `Ast` es el módulo que contiene las definiciones del AST de Little Duck (la definida anteriormente). `FParsec` es el módulo principal de la librería de parsers.

La estructura del parser inicia a partir de `let parse =`. Ahí, estamos diciendo vamos a definir un valor llamado `parse`, dentro del módulo `Identifier`. El valor `parse` contiene el objeto que puede realizar el procesamiento de un identificador. Este inicia con la palabra `regex`. Esta es una función de la librería FParsec, que en este caso está recibiendo un argumento, la expresión regular definida anteriormente para los identificadores. La función nos regresa un parser que solo tiene éxito cuando se encuentra con texto que concuerde con la expresión regular. Después de esto, el operador `|>>` agarra el parser anteriormente definido, y establece que cualquier valor que salga de ahí debe de ser embebido dentro de una instancia de `IdentifierNode`. Finalmente, el operador `<?>` le asigna un nombre a este parser, con lo cual se facilita el proceso de debugging después.

En sí, este pequeño pedazo de código genera un parser que es capaz de procesar cadenas de texto como las siguientes:

```
helloWorld <- EXITO
LoremIpsum <- EXITO
ß <- EXITO
asdfiu23 <- EXITO
21kd <- NO PERMITIDO
/2131 <- NO PERMITIDO
```

En caso de encontrar un fallo, el parser inmediatamente regresaría el punto en donde encontró el error, así como el nombre del parser donde falló.

Un ejemplo más complejo: valores

Él parsing de valores es más complejo, ya que estamos posiblemente tratando con tres diferentes tipos de valores: números, booleanos y cadenas de texto. Al igual que con los identificadores, primero se creó el tipo del AST para estos:

```
type ValueNode =
    | Float of float
    | Int of int
    | String of string
    | Boolean of bool
```

Esta es una union discriminada que puede contener cualquiera de los diferentes valores como un solo tipo. Ya teniendo esto, se crearon los parsers para cada uno de los casos.

Números

FParsec cuenta con parsers ya definidos para números. Para LittleDuck, se utilizó uno de estos, configurado para estar limitado en los diferentes números que es capaz de procesar. A continuación se muestra el parser de números:

```
let pNumber =
    numberLiteral NumberLiteralOptions.AllowFraction "number"
    |>> fun nl ->
        if nl.IsInteger then
            Int(Int32.Parse(nl.String))
        else
            Float(Double.Parse(nl.String))
```

Este es un objeto creado a partir de la función `numberLiteral`, de FParsec. Esta función acepta dos argumentos, las opciones del parser y un nombre para el reportaje de errores. En el primero argumento, únicamente se activó la opción de números fraccionales (con decimales). Para el segundo argumento, se le dio el nombre de `"number"`. Una vez se ejecute el parser, usando el operador `|>>` establecemos la transformación que se le debe dar al valor después de ser obtenido. Establecemos que si el resultado del parser es un número entero, lo convertimos a un Int32, que es el tipo de número entero básico disponible en .NET, y eso lo envolvemos en nuestra union discriminada de `ValueNode`, específicamente en el

caso `Int`. En cambio, si el número no es entero, se convierte a un Double y se envuelve en el caso `Float` de `ValueNode`.

Strings

Para las cadenas de texto, nos importa obtener el valor de texto que está dentro de un par de comillas. Esto se puede expresar en FParsec de la siguiente manera:

```
let pString = skipChar '"' >>. manyCharsTill anyChar (skipChar '"') |>> String
```

Aquí, se hace uso por primera vez del combinador de parsers `>>.`. Lo que este operador hace es combina dos parsers de manera que se ejecute el primero y luego el segundo, sobre el mismo texto. El punto al final le indica a FParsec que el valor que nos interesa se encuentra a la derecha, se ignora la salida del parser de la izquierda.

El primero de los parsers es `skipChar`. Esta función regresa un parser que busca consumir un solo carácter, en este caso `"`. Del lado derecho, la función manyCharsTill genera un parser que consume todos los caracteres usando `anyChar`, hasta encontrar una cadena que logre ser parseada por el parser (`skipChar '"'`). Es decir, consume todos los caracteres hasta encontrar una comilla doble `"`. Como este es el parser de la derecha (donde está el punto), este es el valor que el parser completo va a regresar. Finalmente, el valor de salida se envuelve dentro de `String`, un `ValueNode`.

Booleanos

Finalmente, para parsear valores booleanos únicamente es de interés parsear dos cadenas de texto: `true` y `false`. Similar a `skipChar`, `skipString` permite indicarle a FParsec que consuma caracteres que sean iguales a una string dada. En este, caso el parse se define usando un operador nuevo: `<|>`. Este define una alternativa, si el primer parser falla, se procede al segundo. Por lo tanto, el siguiente parser primero intenta procesar el string `true`, y si esto falla, el string `false`. En ambos casos, se usa el operador `>>%`. Esto significa que si el parser anterior tuvo éxito, su valor va a ser reemplazado por el de la derecha. En caso de que `skipString "true"` tenga éxito, el parser regresará `Boolean true`, asímismo con el segundo caso, pero entonces se regresa `Boolean false`.

```
let pBoolean =
    (skipString "true" >>% (Boolean true))
    <|> (skipString "false" >>% (Boolean false))
```

Combinando los tres parsers:

Una vez elaborados los tres parsers, combinarlos es tan simple como hacer lo siguiente:

```
module Value
    let parse = pNumber <|> pString <|> pBoolean
```

Esto le dice a FParsec que primero intente procesar un número, si eso falla que intente procesar un string, y si eso falla finalmente intente procesar un booleano. Si ninguno de los tres parsers tiene éxito, el procesamiento termina en error.

AST Completo

A continuación, se muestra la estructura del AST completo. Cada parte de este árbol tiene su propio parser, y estos componen el parser completo.

```
module LittleDuck.Ast

type IdentifierNode = IdentifierNode of string

type ValueNode =
    | Float of float
    | Int of int
    | String of string
    | Boolean of bool

// Recursive on ExpressionNode
type InvocationNode = InvocationNode of IdentifierNode * ExpressionNode list

and CTENode =
    | NegativeCTE of ValueNode
    | PositiveCTE of ValueNode
    | ValueCTE of ValueNode
    | IdentifierCTE of IdentifierNode
    | InvocationCTE of InvocationNode


and FactorNode =
    | ParenthesizedExprFactor of ExpressionNode
    | CTEFactor of CTENode

and TermNode =
    | FactorTerm of FactorNode
    | DivisionTerm of TermNode * FactorNode
    | MultiplicationTerm of TermNode * FactorNode

and ExpNode =
    | TermExp of TermNode
    | SumExp of ExpNode * TermNode
    | SubtractionExp of ExpNode * TermNode

and ExpressionNode =
    | LessThanExpression of ExpNode * ExpNode
```

```
    | GreaterThanExpression of ExpNode * ExpNode
    | NotEqualExpression of ExpNode * ExpNode
    | EqualsExpression of ExpNode * ExpNode
    | ExpExpression of ExpNode

// End of recursion

type PrintNode = PrintNode of ExpressionNode list

// Recursive on BodyNode
type CycleNode = CycleNode of BodyNode * ExpressionNode

and IfNode = IfNode of ExpressionNode * BodyNode

and ElseNode = ElseNode of ExpressionNode * BodyNode

and ConditionNode =
    | IfElseCondition of IfNode * ElseNode
    | IfCondition of IfNode

and AssignmentNode = AssignmentNode of IdentifierNode * ExpressionNode

and ReturnNode = ReturnNode of ExpressionNode option

and StatementNode =
    | AssignmentStatement of AssignmentNode
    | ConditionStatement of ConditionNode
    | CycleStatement of CycleNode
    | InvocationStatement of InvocationNode
    | PrintStatement of PrintNode
    | ReturnStatement of ReturnNode

and BodyNode = BodyNode of StatementNode list

// End of recursion

type TypeNode =
    | FloatType
    | IntType
    | StringType
    | BooleanType

type VariableDeclarationNode = VariableDeclarationNode of IdentifierNode list * TypeNode

type VariableDeclarationsNode = VariableDeclarationsNode of VariableDeclarationNode list
```

```
type ArgumentNode = ArgumentNode of IdentifierNode * TypeNode

type FunctionReturnType =
    | Void
    | ReturnType of TypeNode

type FunctionDeclarationNode =
    | FunctionDeclarationNode of FunctionReturnType *  IdentifierNode * ArgumentNode list *

type FunctionDeclarationsNode = FunctionDeclarationsNode of FunctionDeclarationNode list

type ProgramNode = ProgramNode of IdentifierNode * VariableDeclarationsNode * FunctionDecla
```

Suite de pruebas

Para asegurar el correcto funcionamiento del parser, se construyó un conjunto de pruebas unitarias que buscan corroborar el correcto funcionamiento de cada parser en el lenguaje. Estas pruebas fueron escritas usando la framework de XUnit, una librería estándar de pruebas unitarias en el ecosistema .NET. Esto asegura que el parser completo logre procesar correctamente el lenguaje.

Implementación completa

Explicar la implementación del parser a detalle está fuera del alcance del presente documento. La implementación completa del código se encuentra disponible en el repositorio de GitHub de Little Duck, disponible en github.com/Stock44/LittleDuck.

Análisis semántico

Una vez se ha validado que el lenguaje tenga una sintaxis válida, y se ha construido una representación en memoria de un programa, se debe de analizar que el programa tenga sentido. Existen muchos errores que pueden ocurrir incluso aun cuando sintácticamente un programa parezca correcto. Para esto se realiza el proceso de análisis semántico.

El análisis semántico consiste en verificar que un programa obedezca las reglas definidas por lenguaje que no son comprobables con simple sintaxis, como el sistema de tipos de un programa. Esto ocupa un proceso de análisis sobre él el AST, con el cual se pueda obtener una lista de errores de un dado programa.

Recorrido de árboles en F#

Los registros y las uniones discriminadas hacen natural el modelar árboles en F#. Sin embargo, para poder recorrer y realizar operaciones sobre estos tipos. Para esto, F# ofrece el *pattern matching*. Esta es una capacidad del lenguaje de

descomponer uniones discriminadas, obteniendo sus valores internos dependiendo de cuál caso sea, y así pudiendo hacer operaciones sobre estos. Por ejemplo, si se quisiera hacer una operación diferente para cada tipo de `ValueNode`, se podría definir una función como la siguiente:

```
let processValueNode (value: ValueNode) =
    match value with
    | Float f ->
        printfn "Processing float: %f" f
    | Int i ->
        printfn "Processing int: %d" i
    | String s ->
        printfn "Processing string: %s" s
    | Boolean b ->
        printfn "Processing boolean: %b" b
```

Esta función recupera el valor interno de un `ValueNode` e imprime algo diferente en cada caso. Esto se puede extender por medio del uso de funciones recursivas para así poder evaluar árboles de naturaleza recursiva, como es el caso del AST de Little Duck.

Errores a analizar

Durante el proceso de desarrollo, se identificarón varios tipos de errores semánticos que se pueden encontrar al analizar un programa de Little Duck. Estos son representados por la siguiente unión discriminada:

```
type SemanticError =
    | UnknownSymbol of symbolName: string
    | NameConflictsWithProgram of symbolName: string
    | ProgramUsedAsValue
    | FunctionUsedAsValue of functionName: string
    | VoidFunctionUsedAsValue of functionName: string
    | InvalidBinaryOperandTypes of operator: string * lhsType: TypeNode * rhsType: TypeNode
    | AssignmentWithWrongType of variableName: string * variableType: TypeNode * receivedTyp
    | AssignmentToFunction of functionName: string
    | AssignmentToProgram
    | AssignmentToArgument of argumentName: string
    | InvalidConditionType of receivedType: TypeNode
    | InvokedVariable of variableName: string
    | InvokedArgument of argumentName: string
    | InvokedProgram
    | InvocationArgumentCountMismatch of functionName: string * receivedArgumentCount: int *
    | InvocationArgumentTypeMismatch of
        functionName: string *
        argumentName: string *
        receivedArgumentType: TypeNode *
```

```
      expectedArgumentType: TypeNode
  | ReturnedValueFromVoidFunction of functionName: string
  | ReturnedNothingFromValueReturningFunction of functionName: string * expectedType: Type
  | ReturnTypeMismatch of functionName: string * returnedType: TypeNode * expectedType: Ty
  | UnresolvedType
```

En sí, el programa de análisis semántico debe de ser capaz de generar una lista de valores de tipo `SemanticError`, en caso de que el programa presente cualquiera de esos errores.

Fases del análisis

El compilador fue construido mediante una arquitectura de capas. Cada capa tiene responsabilidades diferentes, lo que promueve su modularidad e independencia, facilitando las pruebas unitarias. Estas son las siguientes:

1. Recolección del contexto semántico
2. Cubo semántico
3. Resolución de tipos
4. Recolección de errores
5. Análisis semántico

Capa 1: Recolección del contexto semántico

La recolección del contexto semántico se basa en la construcción de una estructura de datos que provea *contexto*, mediante el cual el resto del programa pueda determinar errores. Esta estructura está definida de la siguiente manera:

```
type SemanticContext =
    { Name: string
      ReturnType: FunctionReturnType
      SymbolTable: Map<string, Symbol> }
```

El registro de `SemanticContext` tiene tres propiedades. Estas se explican a continuación:

- `Name`: Este es el nombre del entorno de ejecución actual. En el cuerpo del programa este es el nombre del programa, y en el cuerpo de una función es el nombre de la función.
- `ReturnType`: Este es el tipo de retorno de la función actual. En el cuerpo del programa es `Void`, y en el cuerpo de una función es el tipo de retorno de esta.
- `SymbolTable`: La tabla de símbolos definidos por el programa.

De estas propiedades, la más importante es la tabla de símbolos. Esta tabla es una estructura de datos tipo Mapa, que mapea el nombre de los símbolos a una union discriminada llamada `Symbol`. Esta tiene cuatro casos, que son los siguientes:

```
type Symbol =
    | Program
    | Variable of TypeNode
    | Argument of TypeNode
    | Function of FunctionDeclarationNode
```

Por lo tanto, la tabla de símbolos almacena las definiciones para cada símbolo del programa: el tipo de símbolo y propiedades relacionadas.

Esta fase del análisis semántico debe de llenar correctamente todos los campos del programa, alertando de cualquier error que pase. Si bien el código de esta sección es largo, algo que es clave entender es la manera en que F# se asegura de que todos los errores encontrados durante el análisis sean tratados correctamente. Esto se puede ver en las funciones definidas para manipular las tablas de símbolos. Como ejemplo, a continuación se muestra el tipo (type signature_, no la implementación) de la función que agrega una definición de variable al contexto (`addVariableDeclaration`):

```
TypeNode -> string -> SemanticContext -> Result<SemanticContext, SemanticError>
```

Este tipo se puede interpretar como una función que acepta un `TypeNode`, luego un `string`, y finalmente un `SemanticContext`, con lo cual finalmente retorna un valor de tipo `Result<SemanticContext, SemanticError>`. Es este el primer acercamiento que se tiene dentro del compilador al concepto de un *monad*.

> **Monads** Un monad puede ser entendido como un posible valor que solo está disponible dentro de un contexto. Para concretar esto, se puede observar al tipo anteriormente mencionado, `Result<'t, 'e>`. Este es un tipo genérico, que puede contener cualquier otro tipo dentro. Un `Result` puede encontrarse en dos estados, puede que esté en el estado `Ok`, lo que significa que está disponible un valor de tipo `'t`, o que esté en el estado `Error`, en cuál caso solo esta disponible el tipo `'e`. Esto permite modelar de manera natural el resultado de un cómputo que puede fracasar, y obliga al programador a tratar con este valor. Por lo tanto, se puede definir a Result como un monad representando a un valor que puede existir dentro de un contexto donde un cómputo tuvó éxito, o donde existe otro valor donde ese cómputo fracasó.
>
> Por su cuenta esto es bastante útil, sin embargo, los monads representan una abstracción de naturaleza matemática, lo que significa que **todos los monads comparten ciertas propiedades**, lo que los vuelve uno de los bloques fundamentales de la programación funcional.
>
> Un ejemplo de esto, es la operación `bind`. Digamos que se tiene dos funciones:

`f: x -> Result<y, Error>` `g: y -> Result <z, Error>`

Se quiere encadenar estas operaciones de manera que al final consigamos un valor tipo z, de la siguiente manera:

```
x |> f |> g
```

Sin embargo, la función `f` regresa un valor de tipo `Result`, pudiendo fracasar, lo cual no es compatible con el tipo de la función `g`. Sin embargo, al ser un monad, `Result` ofrece la operación `bind`. Esta es una función que tiene la capacidad de convertir funciones similares a `x -> M x` a funciones tipo `M x -> M x`. Por lo tanto, aplicando esta función, lo siguiente es válido:

```
x |> f |> bind g
```

O escrito con sintaxis más funcional,

```
x |> f >>= g
```

Todas las operaciones sobre él `SemanticContext` están definidas como funciones que regresan `Result` y `Option`, ambos monads, denotando casos de fracaso manipulando el contexto semántico. Estos resultados son recibidos por capas superiores del analizador semántico, y son luego procesadas correctamente.

Al aceptar un `SemanticContext` y devolver otro `SemanticContext` las funciones de esta capa se pueden interpretar como transformaciones sobre el contexto del programa, que pueden ser encadenadas una después de la otra por medio de composición monádica.

Capa 2: Cubo semántico

Esta es probablemente la capa más sencilla del programa. El cubo semántico define la validez de una determinada operación. Por ejemplo, este define si una operación como `2 + 45.2` es válida. En el programa, este está definido como varias funciones independientes que tienen el siguiente tipo:

```
TypeNode -> TypeNode -> bool
```

Esto significa que evalúan si una dada combinación de tipos es válida para una operación específica, como sumar o dividir.

Capa 3: Resolución de tipos

La tercera capa tiene la responsabilidad de procesar expresiones para así obtener el tipo que representan. Todo esto está construido sobre la infraestructura establecida por las capas inferiores. Es decir, la capa 3 require utilizar información del contexto semántico y del cubo semántico para así determinar el tipo de una expresión, si es que este es válido.

La base de la resolución es la función que procesa el tipo de un `ValueNode`. A continuación se muestra el código:

```
module Resolve =
    let valueType (value: ValueNode) =
        match value with
        | Float _ -> FloatType
        | Int _ -> IntType
        | String _ -> StringType
        | Boolean _ -> BooleanType
```

Este únicamente sirve como un mapeo de un dado tipo de `ValueNode`, a un `TypeNode`.

Las cosas se ponen más interesantes una vez se involucran a los CTEs:

```
let rec cteType (cte: CTENode) : ReaderT<SemanticContext, TypeNode option> =
    monad' {
        let! ctx = Reader.ask |> ReaderT.hoist

        match cte with
        | NegativeCTE valueNode -> valueNode |> valueType
        | PositiveCTE valueNode -> valueNode |> valueType
        | ValueCTE valueNode -> valueNode |> valueType
        | IdentifierCTE(IdentifierNode name) ->
            // Si es un identificador, intentamos conseguir el tipo del simbolo. match! se
            // si no encontramos el simbolo, el valor final sera None
            match! name |> SemanticContext.tryGetSymbol ctx |> ReaderT.lift with
            | Program -> return! None |> ReaderT.lift
            | Function _ -> return! None |> ReaderT.lift
            | Argument ty -> ty
            | Variable ty -> ty
        | InvocationCTE(InvocationNode(IdentifierNode name, _)) ->
            // Si es una invocacion, intentamos conseguir el tipo de retorno del simbolo. m
            // si no encontramos el simbolo, el valor final sera None
            match! name |> SemanticContext.tryGetSymbol ctx |> ReaderT.lift with
            | Program -> return! None |> ReaderT.lift
            | Argument _ -> return! None |> ReaderT.lift
            | Variable _ -> return! None |> ReaderT.lift
            | Function(FunctionDeclarationNode(Void, _, _, _, _)) -> return! None |> Reader
            | Function(FunctionDeclarationNode(ReturnType ty, _, _, _, _)) -> ty
    }
```

Este código requiere algo de explicación. Este es el tipo de la función anterior:

```
CTENode -> ReaderT<SemanticContext, TypeNode option>
```

En sí, esta es una función que acepta un nodo del AST (un `CTENode`), y regresa un valor. Dando algunos pasos atrás, el objetivo de este programa es, dado un nodo, conseguir su tipo. Sin embargo, esto es una operación que puede fallar, por lo que en caso de vernos incapaz de resolver el tipo de alguna expresión, podríamos no retornar nada. Asímismo, necesitamos información del contexto

semántico para así poder resolver el tipo de identificadores y funciones. Estas dos diferentes situaciones se pueden modelar por medio de dos monads diferentes: `Reader` y `Option`. El primero define el conjunto de valores que están disponibles solo cuando tenemos disponible un contexto dado (en este caso, ese sería nuestro `SemanticContext`). El segundo define el conjunto de valores que puede que no existan. Para combinar ambos monads, se utilizó el concepto de un *monad transformer*, que permite combinar dos monads diferentes en uno solo. Es por esto que nuestro valor de salida, él `TypeNode`, se encuentra envuelto en dos diferentes tipos `ReaderT` (un monad transformer basado en `Reader`) y `option`.

Es aquí donde también nos encontramos con otro concepto nuevo de F#, las *computation expressions*. Estas expresiones son un apoyo al programar usando monads, ya que nos permiten ocultar las llamadas a métodos como `bind` y `map`, que son utilerías que normalmente se usan al trabajar con monads. Agregan nuevas operaciones posibles, como `let!`. Esta es una operación que le indica al programa que trate al valor contenido dentro de un monad, como `Reader`, que actué como si ya estuviera disponible. Mediante esto, se utiliza el método `Reader.ask`, con el cual se puede obtener directamente valores desde el contexto actual.

Computation Expression Dentro de una *computation expression*, se tiene acceso a varias operaciones. Entre ellas están:

- `let!`: Actúa como `bind` o `map`, permite utilizar el valor de un monad y hacer operaciones sobre este.
- `return!`: Te permite regresar directamente un monad. Si se usa directamente `return`, se va a generar un valor de

```
> tipo `M<M<'t>>`, lo cual no es ideal. `return!` se encarga de aplanar esta estructura, por
> termina
> con un `M<'t>`.
```

- `match!`: Es una combinación de `let!` y `match`, permite hacer *pattern matching* sobre un valor dentro de un

```
> monad.
```

- `yield`: Permite hacer un "generador", es decir, genera una secuencia de valores a partir de una sola computation

```
> expression.
```

- `yield!`: Es una combinación de `return!` y `yield`, aplana una estructura monádica antes de devolverla como

```
> `yield`.
```

Con estas herramientas, se puede realizar el procesamiento del nodo. A grandes rasgos, la lógica es la siguiente:

- Revistamos que tipo de CTE es
  - Si es `ValueCTE`, `NegativeCTE` o `PositiveCTE`, se usa directamente el tipo del `ValueNode` interno calculado con la función `valueType`, y se regresa ese tipo.
  - Si es un `IdentifierCTE`, se usa `match!` para extraer el posiblemente existente tipo del símbolo obtenido mediante `SemanticContext.tryGetSymbol`. Si este no existe, `match!` se encarga de que **toda** la computation expression regrese `None`, significando que falló la resolución del tipo.
    - Si encuentra el símbolo, pero no es un `Identifier` o `Argument`, regresa `None`.
    - Si es un `Identifier` o `Argument`, regresa el tipo de estos símbolos.
  - Si es un `InvocationCTE`, se usa `match!` para conseguir la información de la función.
    - Si se encuentra el símbolo, pero no es un `Function`, regresa `None`.
    - Si es un `Function`, regresa el `ReturnType` de la función.

El resto de las funciones de resolución de tipos están escritas en términos de estas dos funciones. Este módulo en su totalidad, permite a las capas superiores resolver el tipo de cualquíer expresión, permitiendo hacer más análisis después.

Capa 4: Recolección de errores

Las capas anteriores a estas determinan la existencia de ciertos símbolos, así como si se están usando como funciones o valores. Sin embargo, para analizar si los *tipos* de los valores son correctos para el contexto en el que se están usando, se necesita evaluar cada estatuto de cada cuerpo dentro del programa. Por lo tanto, para cada nodo debajo de `BodyNode`, se necesita una función que sea capaz de acumular todos los errores de cada nodo. Dentro de esta capa, todas las funciones de análisis comparten la siguiente estructura:

```
Nodo -> ReaderT<SemanticContext, SemanticError seq>
```

Si bien esto es parecido a la estructura de la capa 3, difieren en el monad interno de ReaderT, aquí se utiliza un `SemanticError seq`. `seq` en sí, representa una secuencia de valores de un dado tipo, en este caso `SemanticError`. Por lo tanto, se podría decir que las funciones de esta capa, dadas un nodo, devuelven una secuencia de errores dentro de un contexto donde hay un `SemanticContext`. Algo interesante aquí es que se está tratando a `seq` como un monad. Generalizando, esto significa que todos los tipos que modelen listas se pueden tratar como si fueran monads. En este caso, queremos acumular errores, por lo que esta abstracción resulta útil para manipular estos valores.

Como ejemplo, se tiene el recolector de errores de `CTENode`:

```
let rec cteErrors (cte: CTENode) : ReaderT<SemanticContext, _ seq> =
    monad.plus {
        match cte with
```

```
        | NegativeCTE _
        | PositiveCTE _
        | ValueCTE _ -> ()
        | IdentifierCTE(IdentifierNode name) ->
            match! name |> getSymbol |> ReaderT.hoist with
            | Ok(Program) -> yield ProgramUsedAsValue
            | Ok(Function _) -> yield FunctionUsedAsValue name
            | Ok(Argument _) -> ()
            | Ok(Variable _) -> ()
            | Error error -> yield error
        | InvocationCTE(InvocationNode(IdentifierNode name, _) as invocation) ->
            match! name |> getSymbol |> ReaderT.hoist with
            | Ok(Program) -> yield InvokedProgram
            | Ok(Argument _) -> yield InvokedArgument name
            | Ok(Variable _) -> yield InvokedVariable name
            | Ok(Function(FunctionDeclarationNode(Void, _, _, _))) ->
                yield VoidFunctionUsedAsValue name
                yield! invocationErrors invocation
            | Ok(Function(FunctionDeclarationNode _)) -> yield! invocationErrors invocation
            | Error error -> yield error
    }
```

Aquí se cambió el *computation expression* que se está usando, de `monad'` a `monad.plus`. Estas son categorías de valores ligeramente similares, pero que tienen una diferencia clave: `monad.plus` solo funciona para valores que sean "monad plus". Para que una categorize cumpla con esta característica deben de cumplir con tres aspectos:

- Ser un monad
- La existencia de un valor "0".
- La capacidad de "sumar" dos diferentes "monad plus" en uno solo.

Un `seq` cumple con todas estas características. Este es un valor dentro del contexto de que puede haber varios valores. Asímismo, el valor "0" es definido como la secuencia vacía. Finalmente, sumar dos valores `seq` es definido como la operación de concatenación de secuencias. Al usar un *monad transformer*, `ReaderT`, todas las propiedades del monad interno se comparten al monad externo, por lo que un `ReaderT<_ seq>` es también un "monad plus".

La función en sí ópera a nivel "monad plus" por medio de la *computation expression*. Hace *pattern matching* sobre el nodo, realizando una operación diferente por cada tipo de CTE:

- Si es un `NegativeCTE`, `PositiveCTE`, o `ValueCTE`, no hace nada, ya que no puede haber errores de utilización n este tipo de nodos.
- Si es un `IdentifierCTE`, intenta conseguir el tipo de el identificador. Si encuentra que es un `Program` o un `Function`, realiza un `yield` con un error, agregando este error a la secuencia de errores.

  - Si es otro tipo válido de identificador, no hace nada, ya que no hay error.
  - Si ocurrió un error al consultar el símbolo, se hace `yield` al error.
- Se repite lo anterior si es un `InvocationCTE`, pero se espera que sea una función. Algo clave aquí es la capacidad del programa de hacer `yield` más de una vez. Primero se evalúa si la función es `Void`, y si así es, se hace yield a ese error. Luego, se revisan todos los argumentos de la función, y si hubo errores, se usa `yield!`. Esta operación aplana el monad que regresa la función de `invocationErrors`, básicamente concatenando otra `seq` a la actual dentro del contexto actual (gracias al *monad transformer*).

Se realiza un proceso similar para procesar otros tipos de expresiones, como los `ExpNode`:

```
and expErrors (exp: ExpNode) =
    monad.plus {
        match exp with
        | TermExp termNode -> return! termErrors termNode
        | SumExp(expNode, termNode) ->
            yield! expErrors expNode
            yield! termErrors termNode

            let! expType = expNode |> Resolve.expType |> switchOption
            let! termType = termNode |> Resolve.termType |> switchOption
            yield! validateAddition <!> expType <*> termType |> resolveOpValidation
        | SubtractionExp(expNode, termNode) ->
            yield! expErrors expNode
            yield! termErrors termNode

            let! expType = expNode |> Resolve.expType |> switchOption
            let! termType = termNode |> Resolve.termType |> switchOption
            yield! validateSubtraction <!> expType <*> termType |> resolveOpValidation
    }
```

Al igual que en casos anteriores, se usa *pattern matching* para hacer diferentes operaciones dependiendo de qué tipo de nodo es, y se ejecuta `yield` multiples veces para regresar los varios errores que pueden surgir durante el proceso de validación.

Al final de todos los recolectores de errores, se recolectan los errores del cuerpo:

```
and bodyErrors (BodyNode body) =
    monad.plus {
        let! statement = body |> ReaderT.lift

        match statement with
        | AssignmentStatement node -> return! assignmentErrors node
        | ConditionStatement node -> return! conditionErrors node
```

```
        | CycleStatement node -> return! cycleErrors node
        | InvocationStatement node -> return! invocationErrors node
        | PrintStatement node -> return! printErrors node
        | ReturnStatement node -> return! returnErrors node
    }
```

Este finalmente construye una secuencia completa de errores a través de cualquier tipo de nodo, recorriendo el árbol, implícitamente manteniendo él `SemanticContext` y recopilando errores en una secuencia de errores.

Capa 5: Análisis semántico

La capa final del análisis une a todas las demás capas, finalmente ejecutando cada parte del compilador. Esta agrupa el estado total del programa en un solo registro:

```
type SemanticAnalysis =
    { Context: SemanticContext
      Errors: SemanticError list }
```

Este contiene el contexto, y los errores acumulados. A partir de esto, se construyen transformadores sobre este estado, con los cuales finalmente se puede obtener un análisis completo del AST. Estas funciones tienen la siguiente estructura:

```
NodoRelevante -> SemanticAnalaysis -> SemanticAnalysis
```

En este nivel, el objetivo es obtener un resultado concreto para el análisis semántico, por lo que se busca eliminar la presencia los monads encontrados en niveles inferiores.

Procesamiento de variables El primer paso de análisis es agregar todas las variables a la tabla de símbolos del contexto semántico. Esto se realiza por medio de las siguientes funciones:

```
let processVariableDeclaration state (VariableDeclarationNode(identifiers, ty)) =
    identifiers |>> unwrapIdentifier
    |> foldResults (fun ctx x -> SemanticContext.defineVariable ty x ctx) state

let processVariableDeclarations (VariableDeclarationsNode declarations) state =
    declarations |> List.fold processVariableDeclaration state
```

La primera función toma un nodo individual de declaración de variable, y usándola, realiza una operación de *fold* con la función de la capa 1 `SemanticContext.defineVariable`. Lo que hace esto es, si la operación de definir la variable tuvo éxito, se usa el resultado `Ok` de esta operación como el nuevo estado actual. En cambio, si falló, se agrega el error a la lista de errores. De esta manera se logra eliminar el `Result` del tipo final.

Folds Un *fold* es una operación de unificación de valores. Permiten resumir una colección de varios valores en uno solo mediante una operación definida por el programador. Los monads tienen la peculiaridad de que pueden ser considerados una colección. Por ejemplo, en el caso de `Result`, este puede ser una colección de 1 o 0 valores, y entonces se le puede aplicar un *fold*.

La segunda función entonces toma la el nodo de todas las declaraciones, extrae la lista de declaraciones individuales, y hace un *fold* sobre todas las declaraciones del programa mediante la función anterior. Esto agrega todas las variables globales al programa, y si alguna tuvo un error y no pudo ser agregada, agrega el error, o los errores en caso de haber multiples. Esto permite al programa reportar **todos** los errores, ya que puede continuar el análisis incluso después de haber encontrado un error.

Procesamiento de funciones Para las declaraciones de funciones, se construyeron funciones que permiten agregar las declaraciones relacionadas con esta. Estas funcionan de manera similar a las de declaración de variables, utilizando `foldResults` para agregar todos los resultados al estado.

```
let processArguments arguments state =
    arguments
    |> foldResults (fun ctx x -> SemanticContext.defineArgument x ctx) state

let processFunctionDeclaration functionDeclaration state =
    functionDeclaration |> foldResult (flip SemanticContext.defineFunction) state
```

Procesamiento de cuerpos Este paso es el que se encarga de recolectar todos los errores que se encuentren un cuerpo, y agregarlos al estado. Como se vio en la capa 4, `Collect.bodyErrors` regresa un valor de tipo `ReaderT`, requiriendo un contexto para poder ejecutarse. Este contexto lo tenemos dentro de el `state` que recibimos para transformar. Por lo tanto, usamos la función `ReaderT.run` para ejecutar el compúto dado el contexto dentro de `state`, y asi conseguir la secuencia final de errores encontrados, eliminando `ReaderT`. Estos son finalmente agregados al estado final por medio de un `fold` y `reportError`.

```
let processBody body state =
    ReaderT.run (Collect.bodyErrors body) state.Context
    |> fold (flip reportError) state
```

Manejo de *scopes* Dentro de una función se tiene acceso a valores adicionales, como argumentos y variables internas. Para esto, se debe de extender el estado mediante las declaraciones locales, y luego se debe poder extraer errores del estado interno hacia el externo. Para esto se definieron las funciones `scopeTo` y `subsume`, que realizan esas acciones.

```
let scopeTo (FunctionDeclarationNode(returnType, IdentifierNode name, arguments, variableDec
    state
    |> setlContext SemanticContext._name name
    |> setlContext SemanticContext._returnType returnType
    |> setl _errors []
    |> processArguments arguments
    |> processVariableDeclarations variableDeclarations

let subsume other state =
    state |> over _errors (fun x -> (view _errors other) @ x)
```

`scopeTo` toma una declaración de función, y usando esta información extiende un `SemanticAnalysis`. Esta actualiza el nombre y el tipo de retorno del contexto, y agrega argumentos y declaraciones de variables adicionales al nuevo contexto.

La segunda función, `subsume`, tiene el trabajo opuesto: tomando un estado interno y el estado global, añade todos los errores encontrados por el estado interno al global.

Procesamiento de los cuerpos de funciones Una vez se tiene todo lo anterior, ahora se puede analizar el cuerpo de cada función. Este proceso tiene cuatro pasos:

1. Se agrega la declaración de la función actual.
2. Se crea el estado local de la función.
3. Se procesa el cuerpo de la función usando este estado interno.
4. Se subsume el estado interno al externo.

```
let processFunctionDeclarations declarations state =
    declarations
    |> fold
        (fun state declaration ->
            let (FunctionDeclarationNode(_, _, _, _, body)) = declaration

            let state = state |> processFunctionDeclaration declaration

            let scopedState = state |> scopeTo declaration |> processBody body

            state |> subsume scopedState)
        state
```

Todo este proceso se realiza para cada declaración de función dentro de `FunctionDeclarationsNode` mediante un *fold*, con lo que se van acumulando los símbolos y errores de manera secuencial.

Procesamiento del programa completo El paso final es ejecutarlo todo secuencialmente. La función que junta todo es `processProgram`, que dado un nodo de programa, realiza todo el proceso:

1. Inicializa el estado de análisis
2. Procesa las declaraciones de variables
3. Procesa las declaraciones de funciones
4. Procesa el cuerpo principal

```
let processProgram
    (ProgramNode(IdentifierNode(programName),
                 variableDeclarations,
                 FunctionDeclarationsNode(functionDeclarations),
                 body))
    =
    init programName
    |> processVariableDeclarations variableDeclarations
    |> processFunctionDeclarations functionDeclarations
    |> processBody body
```

Esto finalmente regresa un estado de análisis que contiene todos los errores semánticos encontrados por el lenguaje.

Conclusión

Este documento demostró todo el viaje que se recorrió para implementar un analizador léxico y semántico para el lenguaje de programación Little Duck. Se creó un parser con el cual se valida la estructura del lenguaje mediante la librería FParsec, de manera que todo está estructurado de manera funcional que permite la sencilla composición de este parser con el resto del programa. Asímismo se diseñó un analizador semántico que modela este proceso como una serie de transformaciones y verificaciones para asegurar la coherencia y validez de un AST de LittleDuck.