Danna García - 202320823
Santiago Manuel Blanco Villadiego - 202110868

Proyecto 3

La solución del proyecto 2 a pesar de ser prácticamente correcta tenía una observación en donde se decía que el highlighting no estaba implementado de la forma correcta.Buscando en el tutorial de Rascal proporcionado en el curso, se encontró una solución a dicha observación para implementarla en el documento Plugin.rsc.

```
module Plugin
import IO;
import ParseTree;
import util::Reflective;
import util::IDEServices;
import util::LanguageServer;
import Relation;
import Syntax;
PathConfig pcfg = getProjectPathConfig(|project://rascaldsl|);
Language tdslLang = language(pcfg, "TDSL", "tdsl", "Plugin", "contribs");
set[LanguageService] contribs() = {
    parser(start[Planning] (str program, loc src) {
        return parse(#start[Planning], program, src);
    })
};
void main() {
    registerLanguage(tdslLang);
}
```

Figura 1. Solución Plugin guía de Rascal

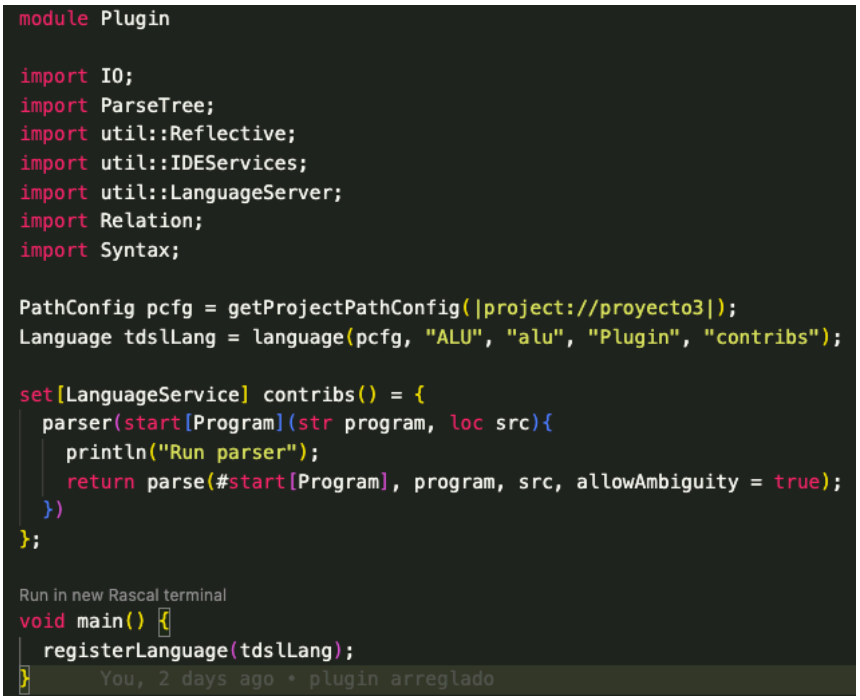

Figura 2. Highlighting con solucion implementada

Cambios en la gramática:

La segunda gramática es esencialmente la primera ampliada con un sistema de tipos explícito que se integra en todos los puntos donde antes solo existían identificadores sin información de tipo. El cambio más visible es la introducción de un nuevo no terminal `Type`, el cual declara de forma explícita los tipos básicos del lenguaje: `Int`, `Bool`, `Char`, `String` y `Float`. Estos símbolos pasan a ser palabras reservadas, de modo que dejan de comportarse como identificadores genéricos y se convierten en constructores de tipo reales dentro de la sintaxis. Esto garantiza que todos los literales enteros, booleanos, caracteres y cadenas, junto con los valores flotantes, tengan un tipo bien definido dentro del lenguaje.

A partir de esta base se incorpora también el no terminal `TypedId`, que permite asociar un tipo a cualquier identificador, ya sea mediante la notación postfija (`x: Int`) o prefija (`Int x`). La gramática conserva además la forma sin anotación (`x`) para mantener compatibilidad con fragmentos no tipados, pero ahora existe soporte formal para adjuntar un tipo siempre que se requiera. Esta extensión es la pieza central que permite propagar las anotaciones de tipo a todas las construcciones del lenguaje.

```
syntax Type
  = tInt: "Int"
  | tBool: "Bool"
  | tChar: "Char"
  | tString: "String"
  | tFloat: "Float"
  ;

You, 15 minutes ago | 1 author (You)
syntax TypedId
  = typedId: Id name ":" Type typeAnn
  | typedIdPrefix: Type typeAnn Id name
  | untypedId: Id name
  ;
```

Figura 3. Gramática de tipos.

La inclusión de `TypedId` constituye una de las transformaciones más significativas. En la versión original, los identificadores eran siempre simples y sin tipo. Ahora, un identificador puede declararse con dos formas distintas de anotación de tipo: `x: Int` o `Int x`, además de mantenerse la opción sin anotación (`x`). La gramática incorpora así mayor flexibilidad y compatibilidad con estilos sintácticos distintos. Esto afecta de forma transversal a todas las declaraciones: variables de funciones, campos en constructores, parámetros de iteradores y

asignaciones dentro de las sentencias. El checker debe ahora reconocer, registrar y validar tipos explícitos en lugar de trabajar únicamente con identificadores sin información adicional.

Otro cambio relevante se observa en las declaraciones `data` y en los constructores tipo `struct`. En el diseño original, los campos eran solamente listas de `Id`, lo cual impedía expresar tipos para cada atributo dentro de un `data`. La nueva versión reemplaza esas listas por colecciones de `TypedId`, permitiendo declarar estructuras de datos con campos tipados. Con ello, el lenguaje se aproxima más al comportamiento de lenguajes funcionales o tipo ML, donde los tipos de los constructores se declaran explícitamente.

```
syntax Data
  = dataWithAssign: Id assignName ":" Type dataType "=" "data" "with" {TypedId ","}+ vars DataBody body "end" Id endName
  | dataNoAssign: "data" Type dataType Id name "with" {TypedId ","}+ vars DataBody body "end" Id endName
  ;

You, 2 days ago | 1 author (You)
syntax DataBody
  = consBody: Constructor
  | funcBody: FunctionDef
  ;

You, 2 days ago | 1 author (You)
syntax Constructor
  = constructorDef: Id name "=" "struct" "(" {TypedId ","}+ vars ")"
  ;
```

Figura 4. Definición data y constructor

Las estructuras definidas por el usuario también se benefician de este cambio. Las declaraciones `data`, que anteriormente solo nombraban identificadores y una lista de variables sin tipo, ahora incluyen tipos explícitos tanto en el nombre del tipo declarado como en cada uno de los campos del `struct`. Las listas de campos dejan de estar formadas por simples identificadores y pasan a ser conjuntos de `TypedId`, forzando así a que cada miembro de la estructura tenga un tipo declarado. De esta forma, los tipos definidos por el usuario se vuelven entidades tipadas completas dentro del lenguaje, equivalentes a los tipos básicos.

Algo similar ocurre con las sentencias de asignación. La primera gramática solo permitía la forma `x = expr`, mientras que la nueva introduce variantes con anotación explícita tanto en forma prefija como postfija (`Int x = expr` y `x: Int = expr`). De esta manera, cualquier valor asignado puede asociarse directamente a un tipo, ya sea un tipo básico o un tipo definido por el usuario. Esto refuerza la consistencia del sistema de tipos a nivel de sentencias y facilita la verificación posterior.

En conjunto, los cambios introducidos reorganizan la gramática en torno a la noción de tipo, extendiendo su presencia a las declaraciones, las estructuras de datos, las variables y las asignaciones. Cada lugar donde antes había solo un identificador ahora puede contener una anotación de tipo. Esto convierte la sintaxis en una versión tipada del lenguaje original y asegura que todos los valores fundamentales (enteros, booleanos, caracteres, cadenas) y todas

las estructuras definidas por el usuario puedan declararse con tipos explícitos, tal como exige un sistema de tipos estático coherente.

Para verificar que los cambios fueron satisfactorios, se ejecutó el main, verificando que los archivos de instance compilaban, permitiendo continuar con el proyecto y verificando así que la gramática modificada no había causado ambigüedades.

Verificación de tipos:

El checker comienza definiendo un conjunto de tipos abstractos (`AType`) que representan de manera interna todos los tipos del lenguaje: enteros, flotantes, booleanos, caracteres, cadenas, tipos personalizados y un tipo desconocido utilizado cuando el sistema aún no dispone de suficiente información. La función `syntaxTypeToAType` convierte los tipos escritos en el programa (como `Int`, `Bool` o `Float`) en estos tipos abstractos. Esta conversión es fundamental porque toda la verificación de tipos opera sobre esta representación interna uniforme; sin ella, TypePal no podría comparar tipos de forma consistente ni detectar incompatibilidades. Además, el checker implementa funciones auxiliares como `prettyAType` para mostrar correctamente los tipos en mensajes de error.

```
module Checker

import Syntax;
import ParseTree;
extend analysis::typepal::TypePal;

// =======================
// Type definitions
// =======================
data AType
  = intType()
  | boolType()
  | charType()
  | stringType()
  | floatType()
  | customType(str name)
  | unknownType()
  ;

str prettyAType(intType()) = "int";
str prettyAType(boolType()) = "bool";
str prettyAType(charType()) = "char";
str prettyAType(stringType()) = "str";
str prettyAType(floatType()) = "float";
str prettyAType(customType(name)) = name;
str prettyAType(unknownType()) = "unknown";

AType syntaxTypeToAType(Type t) {
  if ((Type) `Int` := t) return intType();
  if ((Type) `Bool` := t) return boolType();
  if ((Type) `Char` := t) return charType();
  if ((Type) `String` := t) return stringType();
  if ((Type) `Float` := t) return floatType();
  return unknownType();
}      di.garcia, 2 days ago • cambios …
```

Figura 5. Definición de tipos en Checker.rsc

El sistema también se construye sobre los roles de identificadores definidos por TypePal. Cada nombre en el código puede ser un `variableId`, `functionId`, `dataId` o `fieldId`, y esta clasificación permite controlar el uso apropiado de cada identificador. Por ejemplo, un nombre registrado como `dataId` representa un tipo de dato personalizado declarado mediante `data`, mientras que un `fieldId` solo puede ser utilizado dentro de su respectivo ámbito de estructura. Esta separación de roles es importante porque evita confusiones semánticas: el checker puede detectar cuando un campo se usa como variable o cuando se intenta utilizar una variable como si fuera un tipo.

El análisis comienza en la declaración `Program`, donde se abren scopes y se delega el análisis a cada módulo. Los módulos pueden ser funciones o declaraciones de datos. Cuando se analiza una función, el checker la registra como `functionId`, abre un nuevo scope y define sus parámetros como variables locales. Luego recorre las sentencias del cuerpo, estableciendo restricciones de tipo para cada expresión, asignación o estructura de control. Si el nombre que figura después del `end` no coincide con el nombre de la función, se reporta un

error semántico. En las sentencias de asignación, el checker reconoce las tres formas: `x = e`, `T x = e` y `x : T = e`. En los dos últimos casos, convierte la anotación a un tipo abstracto, declara la variable con ese tipo y genera la restricción que indica que la expresión debe coincidir con el tipo declarado. Gracias a esto, el sistema asegura que los tipos empleados en el programa coinciden exactamente con las reglas del lenguaje, manteniendo una **verificación estricta y confiable de seguridad de tipos**.

El manejo de las declaraciones de datos (`data`) es más profundo. El checker define el nuevo tipo como `dataId`, abre un scope para sus campos y registra cada campo con su tipo correspondiente, ya sea anotado en formato prefijo o posfijo. Si el campo carece de anotación, se asigna por defecto `unknownType`. Después de declarar los campos, el checker verifica los constructores `struct`, asegurándose de que todos los campos mencionados en el constructor existan en la lista de campos previamente definidos. Si aparece un campo inexistente, se genera un error claro señalando la inconsistencia. Esta regla garantiza la existencia real de todos los elementos usados dentro de la definición de una estructura de datos y evita referencias incorrectas, protegiendo la coherencia interna de cada tipo definido por el usuario.

Existencia de variables.

La verificación de existencia de variables se implementa mediante un sistema explícito de ámbitos (scopes) gestionados por el `Collector` de TypePal. La idea esencial es que cada vez que el checker entra en una estructura que introduce un nuevo contexto como funciones, declaraciones de datos, condicionales o bucles, se abre un nuevo scope. Las variables, parámetros, campos y nombres definidos en ese ámbito quedan disponibles únicamente dentro de él y de los scopes interiores.

Cuando el checker encuentra un identificador usado como expresión, lo registra mediante `c.use(...)`. TypePal entonces busca una definición válida en los scopes actualmente abiertos. Si encuentra un identificador con el rol esperado (por ejemplo, `variableId` cuando se usa una variable), el uso es válido; si no existe ninguna definición accesible desde el punto de uso, TypePal produce un error de variable no declarada. Gracias a este mecanismo, el checker detecta correctamente errores como variables usadas sin declaración previa, campos utilizados fuera de la estructura que los define o intentos de acceder a nombres definidos en ámbitos no visibles.

Este sistema se aplica también dentro de estructuras de datos: cuando se construyen tipos personalizados, los campos solo existen dentro del scope del `data`, y el checker verifica estrictamente que se utilicen de manera correcta. Las pruebas confirman que si el constructor `struct` menciona un campo que no está en la lista del `data with ...`, TypePal genera un error, lo cual cumple exactamente con el objetivo de validar la existencia de los elementos utilizados. En resumen, el manejo jerárquico de scopes asegura que todo identificador

empleado en el programa esté correctamente declarado, accesible y utilizado de acuerdo con su rol, cumpliendo así con el requisito fundamental de verificación de existencia de variables que planteaba el proyecto

```
void collect(current: (Statement) `<Id lhs> = <Expression val>`, Collector c) {
  collect(val, c);
  c.define("<lhs>", variableId(), lhs, defType(unknownType()));
}

void collect(current: (Statement) `<Type typeAnn> <Id varId> = <Expression val>`, Collector c) {
  AType atype = syntaxTypeToAType(typeAnn);
  c.define("<varId>", variableId(), varId, defType(atype));
  collect(val, c);
  c.requireEqual(atype, val, error(val, "Type mismatch: expected <prettyAType(atype)>, got %t", val));
}
```

Figura 7. Declaración explícita de variables en statement

Ejecución de pruebas:

La ejecución de pruebas para la validación fue creada en un archivo test.ttl y se ejecutó a través del archivo Test.rsc donde se confirman las reglas definidas previamente, integrando la consistencia de datos entre tipos, creación de estructuras de datos y tipos de variables.

```
test DataWithFunction [[
  Counter: Int = data with value: Int
    function increment(n) do
      Int newVal = n + 1
    end increment
  end Counter
]]

test MultipleDataStructures [[
  Point: Int = data with x: Int, y: Int
    origin = struct(x, y)
  end Point

  Circle: Int = data with center: Int, radius: Int
    unitCircle = struct(center, radius)
  end Circle
]]

test NestedBooleanExpression [[
  function testNested() do
    Bool result = (true or false) and (5 > 3)
  end testNested
]]

test AllComparisons [[
  function testCmp() do
    Bool a = 5 < 10
    Bool b = 5 > 3
    Bool c = 5 <= 5
    Bool d = 5 >= 5
    Bool e = 5 = 5
    Bool f = 5 <> 3
  end testCmp
]]
```

Figura 8. Ejemplo de pruebas realizadas