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

```
module Plugin

import IO;
import ParseTree;
import util::Reflective;
import util::IDEServices;
import util::LanguageServer;
import Relation;
import Syntax;

PathConfig pcfg = getProjectPathConfig(|project://proyecto3|);
Language tdslLang = language(pcfg, "ALU", "alu", "Plugin", "contribs");

set[LanguageService] contribs() = {
  parser(start[Program](str program, loc src){
    println("Run parser");
    return parse(#start[Program], program, src, allowAmbiguity = true);
  })
};

Run in new Rascal terminal
void main() {
  registerLanguage(tdslLang);
}      You, 2 days ago • plugin arreglado
```

Figura 2. Highlighting con solucion implementada

Cambios en la gramática:

La nueva gramática representa una evolución clara y estructuralmente más rica que la original. En la primera versión, el lenguaje no contaba con un sistema explícito de tipos, lo que hacía que todas las variables, parámetros y campos de `data` o `struct` fueran simplemente identificadores sin anotación de tipo. Tampoco existían reglas que describieran variables tipadas ni anotaciones de tipo en declaraciones. La nueva gramática, en cambio, introduce un sistema de tipos formal mediante la producción `Type`, que ahora reconoce palabras reservadas como `Int`, `Bool`, `Char`, `String` y `Float`. Con esta incorporación, el lenguaje deja de depender de valores primitivos para inferir tipo y permite declarar variables y campos tipados de forma explícita.

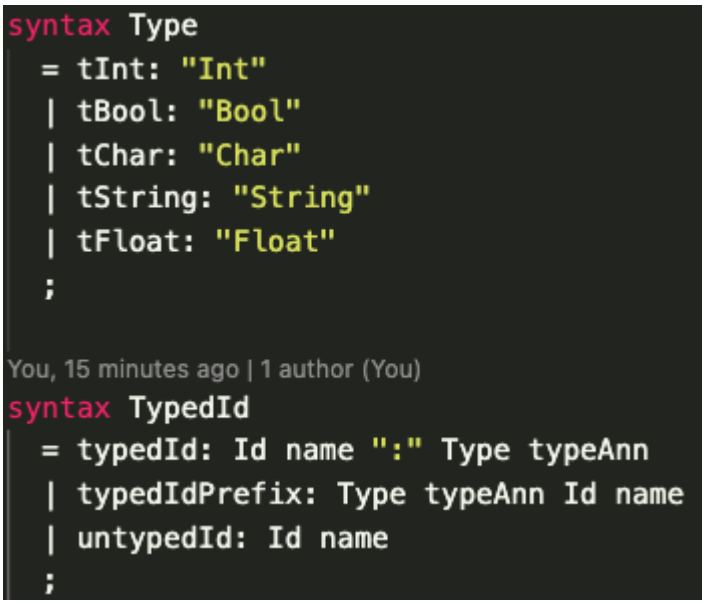

Figura 3. Gramática de tipos.

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.

En el componente de sentencias, también se amplía de manera notable el soporte para anotaciones de tipo. Antes, toda asignación era simplemente `Id = Expression`. En la versión extendida, existen dos nuevas variantes: una que utiliza `TypedId` como lado izquierdo de la asignación, y otra que admite la forma `Type Id = Expression`. Esto permite que las variables sean declaradas y tipadas directamente en su asignación, algo que la sintaxis anterior no contemplaba. Los iteradores y los rangos (`from`/`to`) también se benefician, dado que ahora pueden definirse con `TypedId`, lo que habilita expresiones como `Int i = from 1 to 10`.

Otra modificación estructural importante es la eliminación de la regla `Principal`. En la primera gramática, los valores primitivos utilizados en rangos o expresiones simples se describían con esta superproducción, que unificaba literales y nombres. Esto se ha reemplazado por el uso directo de `Expression`, lo que hace el lenguaje más uniforme y coherente con el resto del sistema de expresiones. La eliminación de `Principal` obliga a que todos los elementos de un rango sean expresiones completas, y no un subconjunto de valores especiales.

Además de estas transformaciones, la nueva gramática amplía la lista de palabras reservadas incorporando los nombres de los tipos (`Int`, `Bool`, `Char`, `String`, `Float`). Esto evita ambigüedades al tokenizar el código y asegura que las anotaciones de tipo no puedan confundirse con identificadores válidos, algo que sí podía ocurrir en la versión anterior.

En conjunto, estas modificaciones tienen un impacto profundo en la semántica del lenguaje. La capacidad de expresar tipos explícitos, declarar variables tipadas y estructurar datos mediante anotaciones refuerza la expresividad y la seguridad del diseño. También exige que el checker realice verificaciones más rigurosas, puesto que ya no basta con validar usos de variables: ahora debe comparar tipos declarados con tipos inferidos, controlar que las anotaciones aparezcan en lugares válidos y asegurar que las construcciones `data` y `struct` tengan coherencia interna. En esencia, la nueva gramática introduce un sistema de tipos declarativo y consistente, ausente en la versión original, y reorganiza varias producciones para adaptarse a este nuevo modelo.

Para verificar que los cambios fueron satisfactorios se ejecutó el main, verificando que los archivos de instance compilaban, permitiendo continuar con el proyecto.

Verificación de tipos:

El proceso inicia definiendo un conjunto de tipos abstractos (`AType`), que representan los tipos reconocidos por el lenguaje: enteros, flotantes, booleanos, caracteres, cadenas, tipos personalizados y un tipo desconocido. Junto a esto, se define una función que convierte los nodos sintácticos `Type` del lenguaje fuente en estos tipos abstractos internos. Esta conversión es esencial, porque el sistema de tipos funciona únicamente sobre los tipos abstractos. También se incluyen funciones auxiliares para validar que los tipos mencionados en declaraciones realmente existen en esta versión, se valida únicamente que pertenecen al conjunto de tipos básicos.

```
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";
```

Figura 4. Asignación de tipos

El checker se apoya en los *roles de identificadores* de TypePal. Cada identificador del código fuente puede desempeñar un rol: variable, función, dato o campo. Asignar estos roles permite que TypePal controle de qué manera puede usarse cada nombre y qué tipos asociados tiene. Por ejemplo, un nombre declarado como `dataId` representa un tipo de dato personalizado, mientras que uno marcado como `variableId` representa una variable con tipo asociado.

El análisis comienza en la regla para `Program`, que simplemente delega el proceso a cada módulo encontrado. Cada módulo puede ser una función o una declaración de datos. Las funciones, cuando se procesan, introducen un nuevo ámbito. Esto significa que sus parámetros y variables internas quedan definidas dentro de un scope propio y no afectan el espacio global. Para cada función, se define su nombre, luego se definen los parámetros y se recorren sentencias que forman su cuerpo, propagando así la recolección de restricciones y de información de tipo. Si el nombre del final de la función no coincide con el inicial, se genera un error semántico.

El manejo de las declaraciones `data` es más elaborado. Cada declaración de datos introduce un tipo personalizado. El checker registra su nombre como `dataId` y abre un nuevo ámbito

donde declara los campos definidos en el bloque `data with ...`. Cada campo puede estar anotado con un tipo explícito (ya sea en formato posfijo o prefijo), o puede carecer de anotación. En los dos primeros casos, el checker valida el tipo, convierte la anotación a un tipo abstracto y declara el campo con ese tipo. En el caso de que falte la anotación de tipo, el checker declara el campo con tipo desconocido y emite una advertencia. Además, se verifica que el nombre que cierra la declaración coincida con el nombre que la abre, emitiendo un error si no coinciden.

Los constructores de estructuras (`struct`) se procesan de manera similar. Cada constructor se considera como una función que construye una instancia del tipo, y sus campos son tratados igual que los campos del `data`, registrándose con sus respectivos tipos. Nuevamente se introduce un ámbito para los campos del constructor, lo que aísla estos nombres de otros espacios del programa.

Las sentencias representan la mayoría de casos donde se imponen restricciones de tipos. En una asignación basada en `TypedId`, el checker distingue entre las tres formas posibles: anotación posfija, prefixada o sin anotación. Si hay una anotación explícita, se valida el tipo declarado, se registra la variable con ese tipo y se compara el tipo declarado con el tipo inferido de la expresión. La comparación se expresa como una restricción: el tipo de la expresión debe ser igual al declarado. Si no se declara tipo, la variable se registra como desconocida y se advierte al usuario. Las sentencias que asignan explícitamente un tipo mediante la producción `Type Id = Expression` generan restricciones equivalentes.

El checker también maneja condicionales, cláusulas condicionales y bucles. En todos estos casos se recogen las expresiones que se encuentran en las condiciones, se evalúan sus tipos y se impone la restricción de que deben ser booleanos (o enteros, en el caso de rangos numéricos). Cada estructura de control introduce su propio ámbito para las variables internas.

La parte más extensa del checker es el recorrido de expresiones. Aquí el sistema determina el tipo abstracto del nodo actual en función de los tipos de sus componentes. Por ejemplo, los operadores lógicos `or` y `and` producen un `boolType` y exigen que ambos operandos sean booleanos. Los operadores aritméticos producen valores enteros y exigen que ambos operandos también lo sean. Todas las expresiones generan restricciones mediante `c.requireEqual`, que compara los tipos de operandos con el tipo esperado, emitiendo errores si existe una incompatibilidad. Los literales reciben su tipo natural mediante `c.fact`.

Cuando se encuentra un identificador usado como expresión, este se marca como uso de una variable. Esto permite a TypePal relacionar usos con definiciones y detectar referencias a variables no declaradas.

Finalmente, las construcciones `sequence`, `tuple` y `struct` no generan tipos propios en esta versión del checker; simplemente recogen las expresiones internas para que sus tipos se procesen. Una versión extendida podría asignar tipos específicos a estas estructuras.

El proceso termina con un método público `typeCheck`, que recibe un programa y retorna el modelo de tipos resultante, o bien un conjunto de errores y advertencias si se encuentran inconsistencias. El motor de TypePal se encarga de resolver las restricciones recolectadas, verificar que cada uso corresponda a una declaración válida y que los tipos coincidan con las expectativas. En conjunto, el checker opera como un verificador semántico que asegura que el código no solo siga las reglas sintácticas del lenguaje, sino también todas sus reglas de tipo, alcance y uso correcto de identificadores.

Existencia de variables.

Para garantizar la existencia de variables dentro del lenguaje, el checker implementa un sistema explícito de *scopes*, o ámbitos, que determina desde qué partes del programa es visible cada variable. La idea central es que, antes de permitir el uso de un identificador, el checker debe confirmar que dicho nombre fue previamente declarado en un ámbito accesible desde el punto de uso. Esta verificación evita errores como el acceso a variables inexistentes o a nombres definidos en contextos incorrectos.

```
    collect(m, c);
  }
}

// =======================
// Module collection
// =======================
void collect(current: (Module) `<FunctionDef fd>`, Collector c) {    Undefined `Collector`
  collect(fd, c);
}

void collect(current: (Module) `<Data d>`, Collector c) {    Undefined `Collector`
  collect(d, c);
}

// =======================
// Data declarations
```

Figura 5. Function Definition y Data

El sistema de scopes se construye abriendo un nuevo ámbito cada vez que se ingresa a una estructura que introduce definiciones propias. En el proyecto, los scopes principales corresponden a los módulos del programa como funciones y declaraciones de datos, que actúan como contenedores globales para variables y tipos definidos por el usuario. Dentro de estos, el checker también crea scopes adicionales para estructuras de control como condicionales y ciclos, de forma que las variables internas a un `if` o a un `for` no escapen de esa región y no puedan ser utilizadas fuera de ella.

Cuando TypePal analiza un identificador usado en una expresión, lo marca como un “uso” de variable y busca su definición más cercana en los scopes activos. Si la definición existe en algún ámbito superior (es decir, visible desde donde se hace el uso), entonces el identificador

es válido. Si no se encuentra ninguna definición compatible, TypePal genera un error de variable indefinida. Esto cubre también los casos en los que estructuras de datos intentan usar variables que no fueron declaradas dentro de su mismo ámbito, lo que también produce un error, tal como se evidenció en las pruebas del proyecto.

Gracias a este manejo jerárquico de scopes, el checker garantiza que cada variable empleada durante la ejecución esté efectivamente declarada y sea accesible, cumpliendo así con el requerimiento fundamental de verificación de existencia de variables planteado en el proyecto.

Ejecución de pruebas:

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

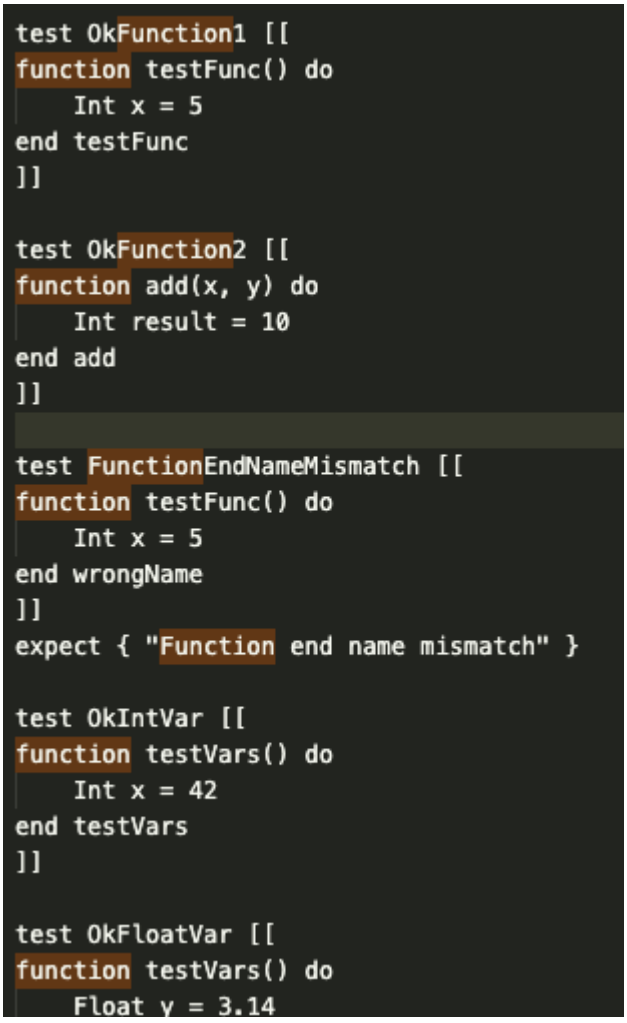

Figura 6. Pruebas realizadas