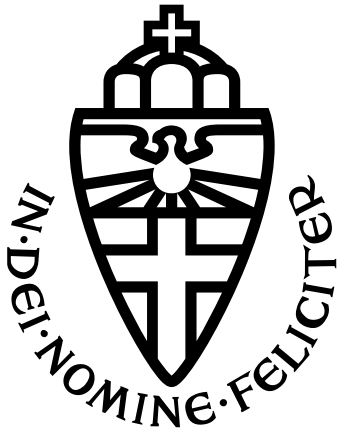
Rascal DSL Tutorial

Mathijs Schuts

August 23, 2023
v1.0



Contents

1 Introduction **3**

2 Installation **4**

3 Overview **5**

4 Concrete Syntax **7**
4.1 Grammar 7
4.2 Plugin 9
4.3 Instance 11

5 Abstract Syntax **12**

6 Code Generation Based on Abstract Syntax **13**
6.1 Function overloading 13
6.2 Visiting 20
6.3 Comprehensions 22

7 Code Generation Based on Concrete Syntax **24**

8 Validation **27**

9 TypePal **29**

10 Tips & Tricks **35**
10.1 Resources 35
10.2 Debugging 36

1 Introduction

Welcome to this tutorial on creating a Domain-Specific Language (DSL) in Rascal. DSLs are specialized programming languages that are designed to solve a specific problem within a particular domain. They are used to simplify complex tasks and improve productivity by allowing developers to express their ideas in a more natural way.

Rascal is a programming language that is designed specifically for language development and transformation. It provides a rich set of features for creating and manipulating languages, including DSLs. With Rascal, you can easily create a DSL tailored to your specific needs, and use it to simplify and automate tasks in your domain.

In this tutorial, we will walk you through the process of creating a DSL in Rascal. We will cover the concepts of DSLs, the syntax of Rascal, and the features available for language development in Rascal. By the end of this tutorial, you will have a basic understanding of how to create your own DSL in Rascal and how to use it to solve problems in your domain. So, let's get started!

2 Installation

To begin, we must install the Language WorkBench (LWB). Follow these steps:

- Download and install VS Code onto your system. You can download it from: https://code.visualstudio.com/download
- Once installed, launch VS Code and navigate to the *Extensions* tab.
- To install Rascal, go to *View* → *Extensions* and search for "Rascal". Press the install button to proceed.

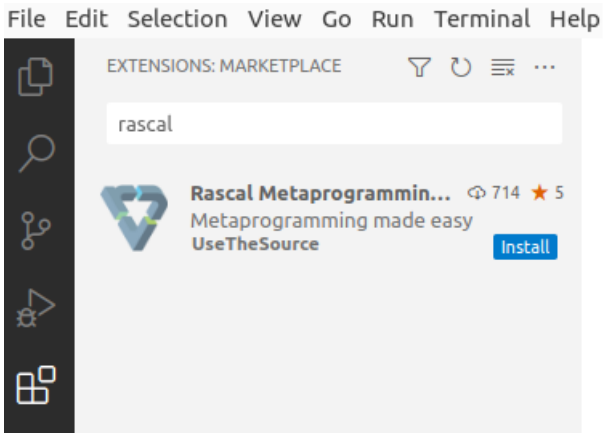


Figure 1: Install Rascal from the Marketplace

- Next we must open a Rascal terminal to initiate the Java installation. Go to *View* → *Command Pallet* . . . and search for "Rascal".
- Choose *Create Rascal Terminal*.
- If prompted, always select *Automatically download Java* 11 when asked about JDK 11 and choose *Microsoft Build of OpenJDK*; do *not* choose one of the other options. Finally, accept the license.

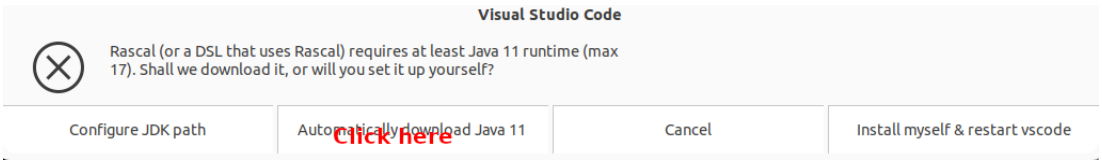


Figure 2: Install Java

- To verify the installation, we can begin by typing *import IO*; followed by *println*("*Hello World*!");. Click on the blue text to open an attached file to copy and paste the text.

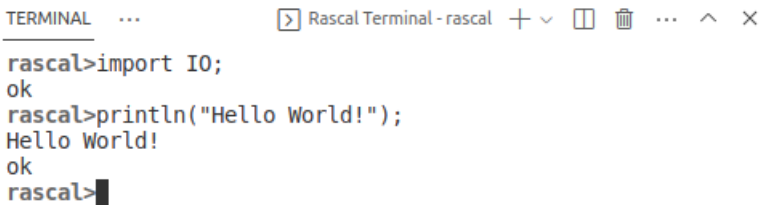


Figure 3: Verify installation

3 Overview

In Rascal, creating a DSL involves defining its concrete syntax or grammar, abstract syntax, and one or more code generators. Figure 4 provides a graphical overview of this process.

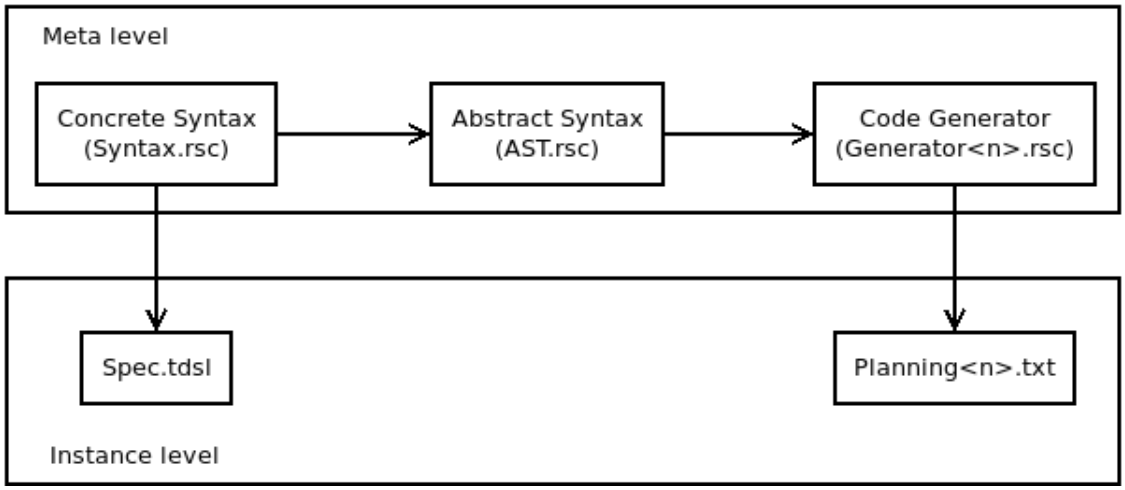


Figure 4: Overview for DSLs in Rascal

Concrete syntax is the representation of a DSL that can be read and understood by both humans and computers. It includes the symbols, keywords, and structures that make up the DSL's syntax and determines how an instance of the DSL is written, formatted, and presented.

Abstract syntax defines the meaning and structure of the DSL's constructs, without specifying how they are represented in concrete syntax. This provides a higher-level way of analyzing and manipulating code than working directly with concrete syntax, which can be more complex and difficult to understand.

Code generation is the process of automatically generating files from a high-level description of a program or system, which in this case is an instance of the DSL. A specification is written in the DSL, and one or more files are generated based on this specification. These files can take the form of instances in another programming language, plain text, or a graphical representation. This process occurs at the instance level.

To provide a comprehensive tutorial on DSL development in Rascal, we will not only cover how to create a code generator based on the abstract syntax but also demonstrate how to develop a code generator using the concrete syntax.

The tutorial will guide you through the process of creating a DSL using a planning example. In the following sections, we will explain how to describe the concrete syntax for the DSL in Section 4, and then define the abstract syntax in Section 5. Additionally, in Section 6, we will demonstrate how to implement code generation based on the abstract syntax. In Section 7, we will focus on code generation using the concrete syntax. How to validate instances is described in Section 8. The TypePal library for name and type analysis is briefly introduced in Section 9. Finally, in Section 10, we provide some tips & tricks for developing languages in Rascal.

4 Concrete Syntax

In this section, we will provide instructions on how to establish a project and define the concrete syntax for our example project, building upon the information outlined in Section 2. We will then proceed to explain how to enable syntax highlighting in VS Code. Finally, we will provide an example of our DSL in use.

4.1 Grammar

Here are the steps to create a project and file with the concrete syntax:

- Click on the blue text to open an attached file to copy and paste the text. In the Rascal terminal type "import util::Reflective;" to load the module required to create a project. Depending on your system, type the appropriate command below.
  - For Windows: "newRascalProject(|file:///c:/path/to/rascaldsl|);".
  - For Unix-based systems (Linux or MacOS): "newRascalProject(|file:///path/to/rascaldsl|);".

  The path and project name should all contain lowercase characters to prevent problems later on[1]. You can choose another location as depicted

```
rascal>import util::Reflective;
ok
rascal>newRascalProject(|file:///home/des/rascaldsl|);
ok
rascal>
```

Figure 5: Create project (for Unix-based system)

  in the Figure 5.

- Add the project to the VS Code workspace by going to *File* → *Add Folder to Workspace...* → *select path/to/rascaldsl*. If you are prompted to trust the authors of the files in this folder, choose "Yes".
- In the *Explorer*, you should see the following folder structure of Figure 6.

[1]Previous versions of Rascal required: newRascalProject(|file:///c:/path/to/rascaldsl|, name="rascaldsl");

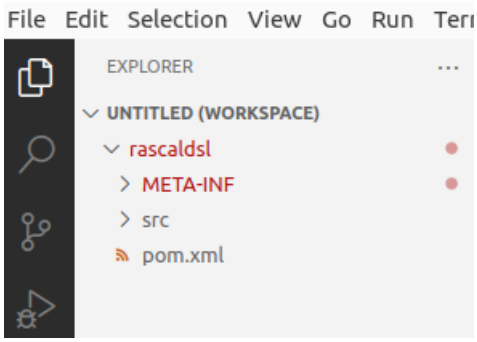


Figure 6: Project structure

- Navigate to the "src/main/rascal" folder using the *Explorer*. You should see the "Main.rsc" file. Click on the file to open it.
- Above the main function, you should see *Run in new Rascal terminal*. Click on it.

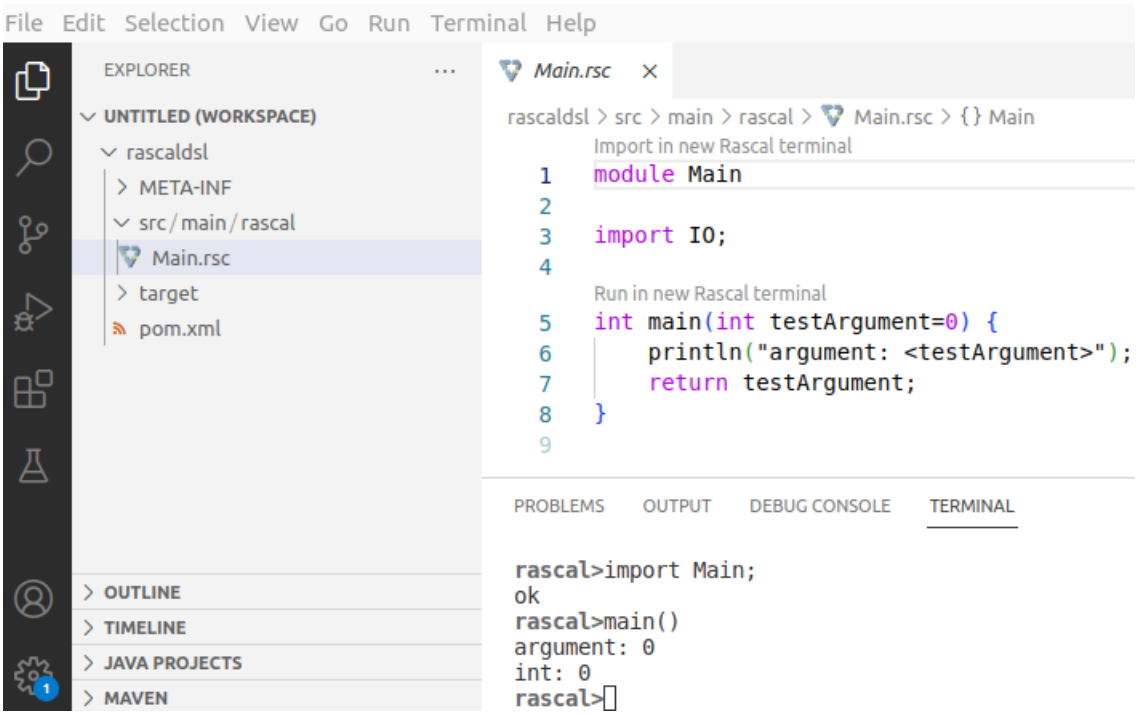


Figure 7: Run main function

- In the "META-INF" folder see Figure 8, open the "RASCAL.MF" file and add |lib://dsl-project| behind *Require – Libraries* :, and remove the four spaces on the last empty line (the line without spaces should stay).
- To create a file named "Syntax.rsc", right-click on the "src/main/rascal" folder in the *Explorer* and select *New File*....

Figure 8: Change project

- Lastly, copy and paste the contents of the attached file into the newly created "Syntax.rsc" file.

Listing 1 shows a Rascal module that defines a grammar for describing tasks of persons using a set of production rules. The grammar includes nonterminals such as Planning, PersonTasks, Task, Duration, Action, LunchAction, MeetingAction, PaperAction, PaymentAction and TimeUnit. These nonterminals are defined using keywords, features, optional features and enumerations.

The start nonterminal is Planning, which requires one or more PersonTasks, indicated by the "+" notation. The PersonTasks rule starts the keyword "Person", followed by a feature "name", and one or more of Task nonterminals. Each Task begins with the keyword "Task", followed by an Action, a priority defined as an INT and an optional duration; the notation "?" indicates zero or one. Duration is defined as an INT and a TimeUnit which can be one of four types: Minute, Hour, Day or Week. The "|" notation is used in a grammar to represent a choice between multiple options.

The grammar defines four types of Actions: LunchAction, MeetingAction, PaperAction and PaymentAction, each with their own set of features. Additionally, the grammar includes lexical definitions for INT, STRING, and ID, and a reserved keyword list to avoid ambiguities between keywords and IDs.

4.2 Plugin

To enable syntax highlighting for files written in our grammar, we need to create a VS Code plugin. Listing 2 provides the "Plugin.rsc" file we are going to use. To do this, we need to follow these steps:

- Create an empty file called "Plugin.rsc" in the same folder as our "Syntax.rsc" file.

```
module Syntax
layout Layout = WhitespaceAndComment* !>> [\ \t\n\r#];
lexical WhitespaceAndComment = [\ \t\n\r] | @category="Comment" "#" ![\n]* $;
start syntax Planning = planning: PersonTasks+ personList;
syntax PersonTasks = personTasks: 'Person' ID name Task+ tasks;
syntax Task = task: 'Task' Action action
           'priority' ':' INT prio
           Duration? duration;
syntax Duration = duration: 'duration' ':' INT dl TimeUnit unit;
syntax Action = lunch: LunchAction lunchAction
             | meeting: MeetingAction meetingAction
             | paper: PaperAction paperAction
             | payment: PaymentAction paymentAction;
syntax LunchAction = lunchAction: 'Lunch' ID location;
syntax MeetingAction = meetingAction: 'Meeting' STRING topic;
syntax PaperAction = paperAction: 'Report' ID report;
syntax PaymentAction = paymentAction: 'Pay' INT amount 'euro';
syntax TimeUnit = minute: Minute minute
               | hour: Hour hour
               | day: Day day
               | week: Week week;
syntax Minute = minute: 'min';
syntax Hour = hour: 'hour';
syntax Day = day: 'day';
syntax Week = week:'week';
lexical INT = ([\-0-9][0-9]* !>> [0-9]);
lexical STRING = "\"" ![\"\n]* "\"";
lexical ID = ([a-zA-Z/.\-][a-zA-Z0-9_/.]* !>> [a-zA-Z0-9_/.]) \ Reserved;
keyword Reserved = "Person" | "Task" | "priority" | "duration" | "Lunch" | "Meeting"
                 | "Report" | "Pay" | "euro" | "min" | "hour" | "day" | "week" | ":";
```

Listing 1: Syntax.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);
}
```

Listing 2: Plugin.rsc

- Copy and paste the contents of the attached file into our newly created "Plugin.rsc" file.
- The contents of the "Plugin.rsc" file include importing our concrete syntax from the "Syntax.rsc" file and registering our DSL. It also maps our DSL to files with the extension "tdsl". When a file with the "tsdl" extension is opened, the "contribs" function is called, the file is parsed and the syntax highlighting is displayed.
- Just above the main function in the "Plugin.rsc" file, we should see *Run in new Rascal terminal*. Click on it to register the DSL and then wait until you no longer see new messages appearing on the blue bar at the bottom of VS Code.
- In VS Code's "Problems" field, you may notice some messages; these can be ignored for now. Also the warning about $|lib://dsl-project|$ can be ignored.

Note that if we make changes to the grammar, we need to run the Plugin's main function again to register the DSL.

4.3 Instance

Listing 3 provides an example of our DSL. We show here how to create a file with this contents in VS Code, follow these steps:

- Right-click on the "rascaldsl" folder in the *Explorer* and select *New Folder...* to create an "instance" folder.
- Right-click on the "instance" folder in the *Explorer* and select *New File...* to create a file named "spec1.tdsl".
- Copy and paste the contents of the attached file into the newly created "spec1.tdsl" file.
- Click on the "spec1.tdsl" file to open it. You should be able to see the contents of the file with syntax highlighting.

5 Abstract Syntax

As shown in Figure 4, we provided a graphical overview of the process to create a DSL with a code generator. Before proceeding with the code generator based on the abstract syntax, we need to define the abstract syntax that corresponds to the concrete syntax. The abstract syntax is provided in Listing 4.

In the abstract syntax, each nonterminal in Listing 1 has a corresponding constructor. The start nonterminal is Planning, which requires one or more PersonTasks. To represent this, we created a constructor with a set of PersonTasks. The PersonTask nonterminal consists of a name (a string) and a list of tasks. The choices made here have implications for the code generator. Notice the use of set or list in the previous two constructors.

For an Action, there is a priority defined as an **int** and an optional duration. To represent the optional duration, we use a **list**, which can have zero or one elements. A TimeUnit can take one of four types: Minute, Hour, Day, or Week. The "|" notation in the grammar denotes a choice between multiple options. In the abstract syntax, we create a constructor for each choice.

For the remaining nonterminals, we define constructors similar to the previous description. It is important to note that we use int for terminals such as INT, and **str** for STRING and ID.

To add the abstract syntax to the project, follow these steps:

- Create a new file named "AST.rsc". If you need guidance on creating a new file, refer to the instructions in Section 4.1.
- Then copy and paste the contents of the attached file into the newly created "AST.rsc" file.

```
Person Alice Task Report Strategy priority: 5
Person Bob Task Pay 5000 euro priority: 2
Person Fred Task Lunch Canteen priority: 8 duration: 1 hour
Person Alice Task Meeting "Demo" priority: 4 duration: 90 min
Person Carol Task Meeting "Training" priority: 7 duration: 3 day
Person Dave Task Report Overview priority: 2 duration: 9 week
Person Bob Task Pay 3500 euro priority: 3
```

Listing 3: spec1.tdsl

```
module AST

data Planning = planning(set[PersonTasks] personList);
data PersonTasks = personTasks(str name, list[Task] tasks);
data Task = task(Action action, int prio, list[Duration] duration);
data Duration = duration(int dl, TimeUnit unit);
data Action = lunch(LunchAction lunchAction)
            | meeting(MeetingAction meetingAction)
            | paper(PaperAction paperAction)
            | payment(PaymentAction paymentAction);
data LunchAction = lunchAction(str location);
data MeetingAction = meetingAction(str topic);
data PaperAction = paperAction(str report);
data PaymentAction = paymentAction(int amount);
data TimeUnit = minute(Minute minute)
              | hour(Hour hour)
              | day(Day day)
              | week(Week week);
data Minute = minute();
data Hour = hour();
data Day = day();
data Week = week();
```

Listing 4: AST.rsc

6 Code Generation Based on Abstract Syntax

We will proceed with creating code generators for the DSL. During this process, you will learn how to implement code generators using function overloading, the visitor pattern, and comprehensions. These approaches provide different techniques for generating code. When you create a code generator for your own DSL in the future, you can choose to utilize a combination of these approaches based on your needs and preferences.

6.1 Function overloading

Our first code generator utilizes function overloading. The concept behind this code generator is illustrated in Figure 9. In the parse tree, we encounter nodes that can be either terminals or nonterminals. The start nonterminal in our grammar is Planning, which represents the complete language instance and consists of a list of PersonTasks.

To match the Planning nonterminal, we define a function called generate with Planning as a parameter. Moving on to the next nonterminal, PersonTasks, its syntax is defined as `'Person' ID name Tasks+ tasks`. An example of this syntax in the language instance could be **`Person`** `Bob task` **`Pay`** `3500` **`euro priority:`**`3`. We match this syntax with a function also named generate, but with a different pa-

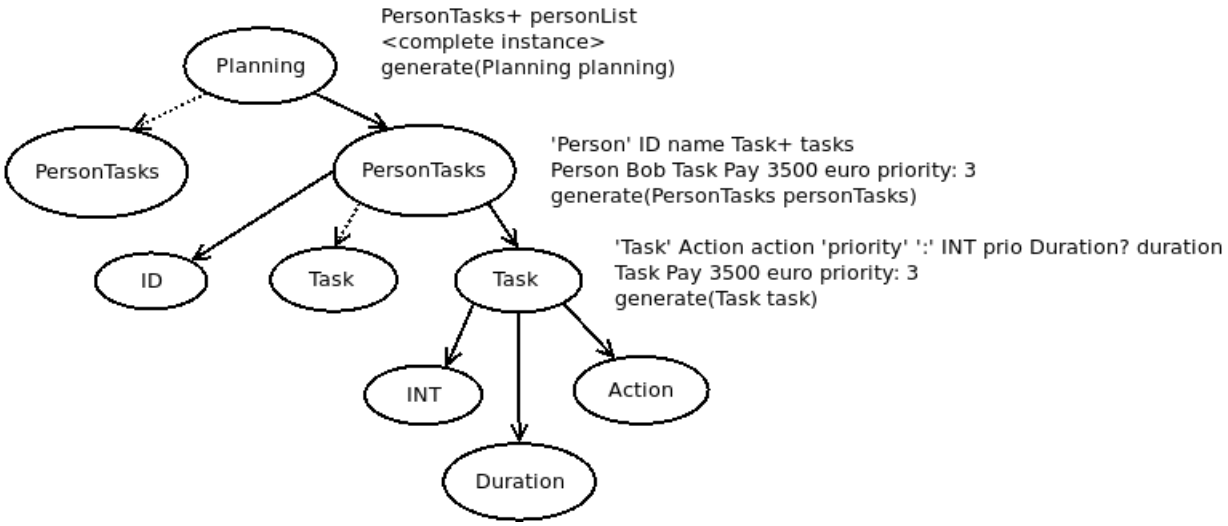


Figure 9: Generator using function overloading

rameter for Task. This is where function overloading comes into play, allowing us to differentiate between different variations of the generate function based on the parameters.

The figure illustrates additional nodes in the tree for PersonTasks. Each PersonTask consists of one or more nonterminal Tasks and a terminal ID. The rest of the tree follows a similar pattern as described.

Listing 5 presents the corresponding generator code, which includes the following components:

- The first line defines an Algebraic Data Type (ADT) called "GenTask". This composite type consists of a "genTask" constructor with typed fields such as action, prio and duration fields. All fields are initialized with default values.

- Two global variables are declared: one as a list of strings and the other as a list of "GenTasks".

- Next, overloaded generator functions are defined. The first function takes a Planning parameter, representing the root of the parse tree. Within the function, a for-loop iterates over the personList and calls the generate function for each personTask.

- The generate function for PersonTasks adds the person's name to the AllPersons list using the "+=" operator. It then iterates over the tasks and calls the generate function for each task.

- The generate function for Task creates a "GenTask" instance by invoking

its “genTask” constructor. The action field is initialized with the result of the generateAction function. Other fields retain their default values, which are later overwritten. The prio field is casted from string to int and assigned to the prio field of the ADT. The duration is optional and stored in a list with either zero or one element. A for-loop iterates over the list, calling the generateDuration function and storing the value in the ADT’s duration field. The ADT instance is then added to the allPlans list.

- The generateAction function utilizes the “?” notation to check if an option is present.
- The rest of the code generator employs concepts that have already been explained.

Now that we have parsed the language instance and stored the relevant information in two global variables, we can process and format the data, and then write it to a file.

Listing 6 demonstrates how to accomplish this:

- The main function parses a language instance and stores the concrete syntax tree in the “cast” variable.
- The generator1 function uses the implode function to generate an Abstract Syntax Tree (AST).
- It then clears the global variables.
- Next, it calls the generate function with the root of the parse tree, which populates the global variables as shown in Listing 5.
- The “rVal” variable, of type string, is used to store the resulting string. The string is constructed as follows:
  - Strings begin and end with quotation marks (”).
  - To continue the string on the next line, the apostrophe (’) is used.
  - Line endings are automatically added.
  - String interpolation is achieved using angle brackets (< and >).
- All persons are listed using a for-loop enclosed in angle brackets (< and >).
- The intercalate function is used to join a list of strings with “ &\n” as the separator.

```
data GenTask = genTask(str action = "", int prio = 0, str duration = "");
list[str] allPersons = []; // normally do not use global variables
list[GenTask] allPlans = [];

void generate(Planning planning) {
    for (personTask <- planning.personList) {
        generate(personTask);
    }
}
void generate(PersonTasks personTasks) {
    allPersons += "<personTasks.name>";
    for (task <- personTasks.tasks) {
        generate(task);
    }
}
void generate(Task task) {
    rVal = genTask(action = generateAction(task.action));
    rVal.prio = toInt("<task.prio>");
    for (dur <- task.duration) {
        rVal.duration = generateDuration(dur);
    }
    allPlans += rVal;
}
str generateAction(Action action) {
    if (action.lunchAction?) return generateAction(action.lunchAction);
    ...
    if (action.paymentAction?) return generateAction(action.paymentAction);
    return "Unknown action!";
}
str generateAction(LunchAction lunchAction) {
    return "Lunch at location <lunchAction.location>";
}
...
str generateAction(PaymentAction paymentAction) {
    return "Pay <paymentAction.amount> Euro";
}
str generateDuration(Duration dur) {
    return "with duration: <dur.dl> <generateDuration(dur.unit)>";
}
str generateDuration(TimeUnit timeUnit) {
    if (timeUnit.minute?) return "m";
    if (timeUnit.hour?) return "h";
    if (timeUnit.day?)  return "d";
    if (timeUnit.week?) return "w";
}
```

Listing 5: Generator with function overloading

```
void main() {
    cast = parsePlanning(|project://rascaldsl/instance/spec1.tdsl|);
    generator1(cast);
}

void generator1(cast) {
    ast = implode(cast);
    allPersons = []; // init to empty for the case you want to generate a second file
    allPlans = [];
    generate(ast);
    rVal =
        "Info of the planning DepartmentABC
        'All Persons:
        '       <for (person <- allPersons) {><person>
        '       <}>
        'All actions of tasks:
        '======
        '       <intercalate(" &\n",
            [ "<plan.action> <plan.prio> <plan.duration>" | plan <- allPlans])>
        '=====
        'Other way of listing all tasks:
        '       <intercalate(" ,\n",
            [ "<plan.action> <plan.prio>" | plan <- allPlans])>
        '";
    println(rVal);
    writeFile(|project://RascalDSL/instance/output/generator1.txt|, rVal);
}
```

Listing 6: Generator with function overloading

```
data Command = gen1(Planning p);

set[LanguageService] contribs() = {
    parser(start[Planning] (str program, loc src) {
        return parse(#start[Planning], program, src);
    }),
    lenses(rel[loc src, Command lens] (start[Planning] p) {
        return {
            <p.src, gen1(p.top, title="Generate text file")>
        };
    }),
    executor(exec)
};
value exec(gen1(Planning p)) {
    rVal = generator1(p);
    outputFile = |project://RascalDSL/instance/output/generator1.txt|;
    writeFile(outputFile, rVal);
    edit(outputFile);
    return ("result": true);
}
```

Listing 7: Plugin

- The "rVal" variable is printed using the println function and written to a file using the writeFile function.

We can trigger the generator in two ways: through the terminal or via the GUI. Listing 7 presents the necessary modifications needed to invoke the generator from the GUI. Here's how it works:

- A new command "gen1" is defined.
- The contribs function is expanded to include lenses. It registers "gen1" with the title "Generate text file". Additionally, the exec function is registered with the executor function.
- The exec function is defined, taking the "gen1" command as a parameter.
  - Within the function body, the generator1 function is used to store a string in the "rVal" variable.
  - Subsequently, a location is assigned to the "outputFile" variable.
  - Using the writeFile function, the string is written to the specified "outputFile" location.
  - The edit function opens the newly generated output file in VS Code.

To make the above workflow function in your environment, follow these steps:

- Create a file named "Parser.rsc" by right-clicking on the "rascal" folder in the *Explorer* and select *New File...*.
- Open the "Parser.rsc" file, and copy the contents of the attached file into it. Save the file.
- Create and open the "Implode.rsc" file, and copy and paste the contents of the attached file into it and save the file.
- Idem for "Generator1.rsc". Create and open the file, and copy and paste the contents of the attached file into it.
- Above the main function in "Generator1.rsc" file, you should see *Run in new Rascal terminal* after saving the file. Click on it to generate the output file.
- Inspect the generated "generator1.txt" file stored in the "instance/output" folder.
- Remove the generated generator1.txt file by right-clicking on it and choosing the delete option.
- The next steps are required to generate the same output file from the GUI.
- Open the already existing "Plugin.rsc" file.
- Copy and paste the contents of the attached file into the opened "Plugin.rsc" file.
- Rerun the main function in the "Plugin.rsc" file. Note that it may take some time to reinitialize the project.
- Open the language instance file "spec1.tdsl". After reinitialization, syntax highlighting should be visible.
- At the top of the "spec1.tdsl", you should now see "Generate text file". Click on it to generate the "generator1.txt" file. Inspect the contents of the opened file.

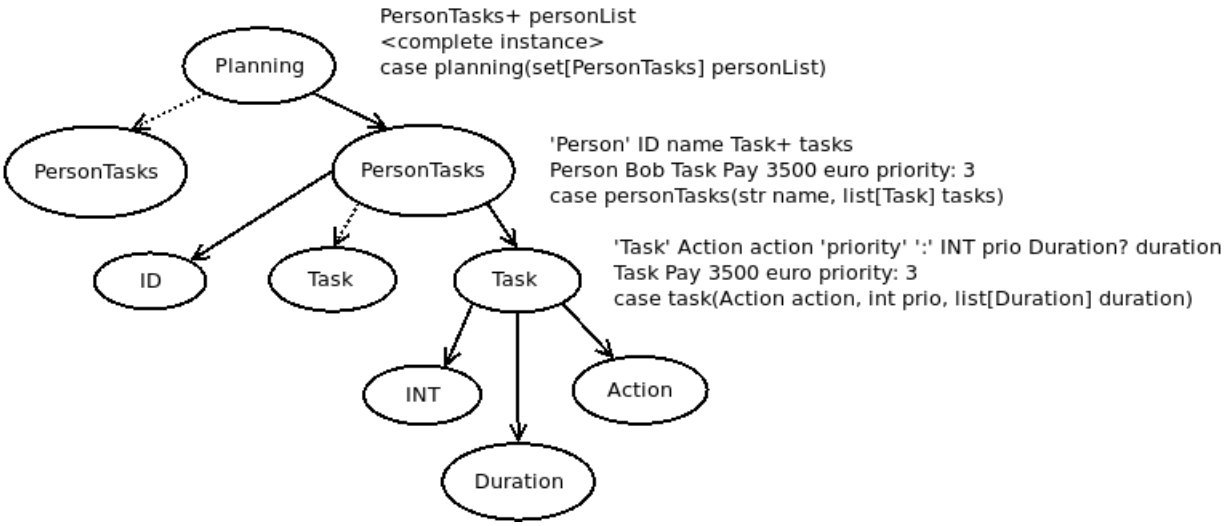


Figure 10: Generator using visiting pattern

6.2 Visiting

Our second code generator utilizes the visiting pattern, which is illustrated in Figure 10.

In the previous code generator based on function overloading, we needed to have a function for every node in the parse tree. Even if we wanted to match a node halfway through the parse tree, we had to create overloaded functions to reach that point. However, with the visiting pattern, this is not necessary. We can start at any point in the parse tree and match a specific node. As shown in Figure 10, we use the constructors defined in the abstract syntax file "AST.rsc" to perform the matching. The tree is traversed until a match is found. It is also possible to have multiple matches. Additionally, there is no requirement to start at the root of the tree. You can begin at any node and traverse the subtree below it.

To utilize the visiting pattern, we use the `visit` keyword, which takes the starting node as input. Matching is performed using the `case` keyword followed by a constructor defined in the abstract syntax.

Listing 8 provides relevant fragments of the corresponding generator code, which includes the following components:

- The printTaskWithDuration function takes the root of the parse tree as a parameter. It utilizes the AST to match tasks using the "task" constructor. The fields of the constructor can be assigned to variables, as demonstrated with "action" and "duration", or ignored using the "_" notation, as shown with priority. For ignoring lists, the "_*" notation can be utilized. When a match is found, a string is constructed by invoking

```
str printTaskWithDuration(ast) {
    rVal = [];
    visit (ast) {
        case task(action, _, duration):
            rVal += "      <printAction(action)> <printDuration(duration)>";
    }
    return intercalate(" &\n", rVal);
}
str printDuration(duration) {
    visit (duration) {
        case duration(dl, unit): {
            u = "";
            visit (unit) {
                case minute(): u = "m";
                ...
                case week():  u = "w";
            }
            return "with duration: <dl> <u>";
        }
    }
    return ""; // duration is optional
}
```

Listing 8: Generator using visiting pattern

the printAction function with the value of the "action" variable as input. This value represents a node halfway through the parse tree. Similarly, the printDuration function is called to construct the remainder of the string, using the "duration" value.

- Moving on to the printDuration function, it receives a list of durations as a parameter, representing a node halfway through the parse tree. The visiting pattern is employed to match a duration, extracting its value and storing it in the "dl" variable, as well as obtaining the unit and storing it in the "unit" variable. Another visit is used to obtain the correct unit and store it in the "u" variable. Finally, a string is created using the "dl" and "u" variables. It's important to note that a duration list can also be empty, in which case there will be no match and an empty string will be returned.

- Similar functions are employed to match the other nodes.

To call this code generator from the GUI, follow these steps:

- Right-clicking on the "rascal" folder in the *Explorer* and select *New File...* to create a file named "Generator2.rsc".

```
str printTaskWithDuration(ast) {
    rVal = [];
    comp = [ <action, duration> | /task(action, _, duration) := ast ];
    for (<a, d> <- comp) {
        rVal += "<printAction(a)> <printDuration(d)>";
    }
    return intercalate(" &\n", rVal);
}
str printTaskWithoutDuration(ast) {
    rVal = [];
    for (a <- { action | /task(action, _, _) := ast }) {
        rVal += "<printAction(a)>";
    }
    return intercalate(" ,\n", rVal);
}
```

Listing 9: Generator using comprehensions

- Open the "Generator2.rsc" file, and copy and paste the contents of the attached file into it. Save the file.
- Open the existing "Plugin.rsc" file.
- Copy and paste the contents of the attached file into the opened "Plugin.rsc" file.
- Rerun the main function in the "Plugin.rsc" file.
- Open the language instance file "spec1.tdsl". After reinitialization, syntax highlighting should be visible.
- At the top of the "spec1.tdsl", you should now see "Generate text file" and "Text generator2". Click on "Text generator2" to generate the "generator2.txt" file. The generated file will be opened, and you can inspect its contents.

6.3 Comprehensions

Our third code generator utilizes comprehensions to generate a file. The concept behind this code generator is similar to the code generator based on the visit pattern. Any node within the parse tree can be directly matched. We explain comprehensions using fragments of the third code generator shown in Listing 9. The code generator includes the following functions:

- The printTaskWithDuration function takes the root of the AST, similar to the previous code generator. It initializes the "rVal" variable, which is

a list of strings. The "comp" variable stores the result of a comprehension. The comprehension returns a list of tuples containing "action" and "duration" values. It uses the match operator (":=") to match the "task" constructor. The fields of the constructor are assigned to the "action" and "duration" variables, which are used to create a tuple. The presence of the "/" operator before "task" allows for deep matching, ensuring that any node in the tree matching the "task" constructor is included in the "comp" variable. Finally, the "rVal" list of strings is joined and separated by " &\n" before being returned as the result.

- The printTaskWithoutDuration function works similarly to the printTaskWithDuration function, with two main differences. Firstly, the comprehension is directly used in the for-loop without being stored in a variable. Secondly, it results in a set of "actions" instead of a list of "action", "duration" tuples. Lists are denoted by square brackets ("[" and "]"), while sets are denoted by curly brackets ("{" and "}").

To call this code generator from the GUI, follow these steps:

- Create a file named "Generator3.rsc".
- Open the "Generator3.rsc" file, and copy and paste the contents of the attached file into it. Save the file.
- Open the existing "Plugin.rsc" file.
- Copy and paste the contents of the attached file into the opened "Plugin.rsc" file.
- Rerun the main function in the "Plugin.rsc" file.
- Open the language instance file "spec1.tdsl".
- At the top of the "spec1.tdsl", you should now see "Generate text file" and "Text generator3". Click on "Text generator3" to generate and open the "generator3.txt" file.

To conclude this section, when creating a code generator for your own DSL, you have the flexibility to choose and combine different approaches described in this section based on your specific requirements and preferences. You can leverage techniques such as function overloading, visiting pattern, and comprehensions to achieve the desired code generation functionality for your DSL. Experimentation and exploration of different approaches can help you find the most suitable solution for your specific use case.

```
str printTaskWithDuration(ast) {
    rVal = [];
    visit (ast) {
        case Task) `Task <Action action> priority: <INT prio> <Duration? duration>`:
            rVal += "<printAction(action)> <printDuration(duration)>";
    }
    return intercalate(" &\n", rVal);
}
str printTaskWithoutDuration(ast) {
    rVal = [];
    for (a <- { action |
            /(Task) `Task <Action action> priority: <INT prio> <Duration? duration>` := ast }) {
        rVal += "<printAction(a)>";
    }
    return intercalate(" ,\n", rVal);
}
```

Listing 10: Generator using concrete syntax

7 Code Generation Based on Concrete Syntax

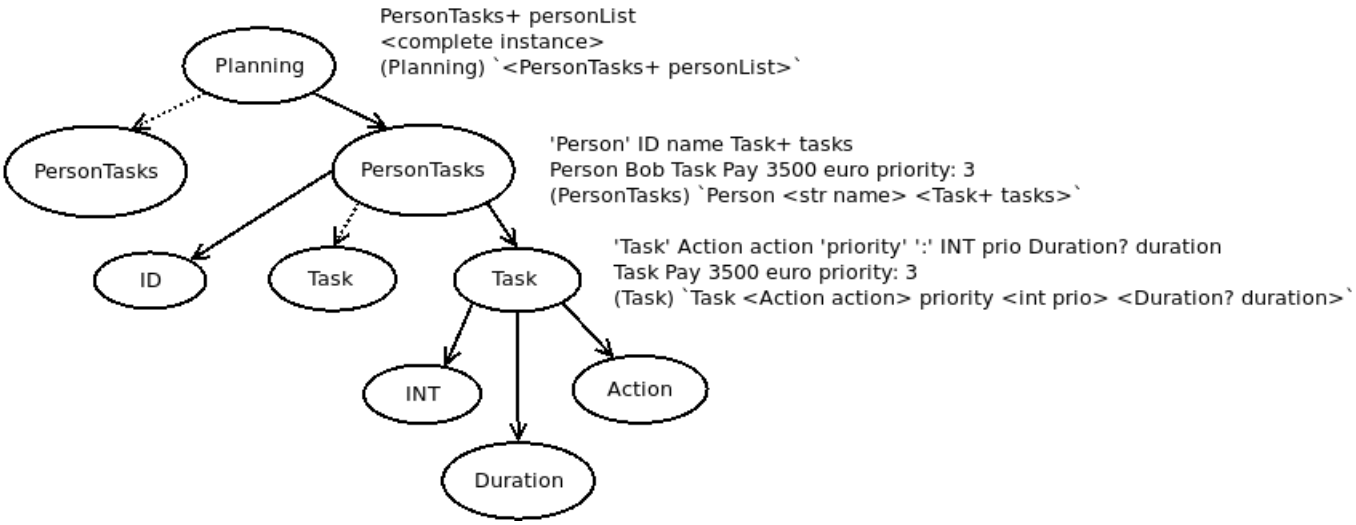


Figure 11: Generator using concrete syntax

In the previous sections, you learned about defining an abstract syntax for a DSL and utilizing the AST to create code generators. Here, we present an alternative approach to creating code generators based on concrete syntax.

The concept behind this code generator is illustrated in Figure 11. In the parse tree, we encounter nodes that can be either terminals or nonterminals. For example, consider the grammar for PersonTasks, which is defined as `'Person' ID name Tasks+ tasks`. An example of this syntax in the language instance could be `Person Bob task Pay 3500 euro priority:3`. To match this grammar, we use the

concrete syntax pattern `(PersonTasks) `Person <str name> <Task+ tasks>``. Above and below this example, there are two more examples illustrating how to translate the grammar into concrete syntax patterns.

We explain a code generator based on concrete syntax by providing fragments in Listing 10. The techniques learned in the previous section, such as function overloading, the visitor pattern, and comprehensions, can all be utilized with concrete syntax as well. Listing 10 showcases the same functions as in Listing 8.

The example includes the following functions:

- The printTaskWithDuration function describes a concrete syntax based on the visitor pattern. Recall the concrete syntax for "Task":
  ```
  syntax Task = task:'Task' Action action 'priority' ':' INT prio Duration? duration;
  ```
  Matching a concrete syntax pattern starts by describing its type between parentheses. In this case, the type is "Task". Next, the concrete syntax is written between backtick (‘) symbols. Keywords are written without apostrophes (’), while terminals and nonterminals are enclosed in angle brackets (< and >).

- The printTaskWithoutDuration function describes a concrete syntax based on a comprehension. The rest of the function is the same as in the previous example.

It is important to remember that when defining the abstract syntax, we choose between a list and a set when zero or more elements are possible. However, when using concrete syntax, everything becomes lists. Optional features are represented as lists with zero or one element.

To call this code generator from the GUI, follow these steps:

- To illustrate that we no longer depend on the abstract syntax for this code generator, delete the "AST.rsc" and "Implode.rsc" files. Right-click on each file and choosing the delete option.

- Create and open a new file named "Generator4.rsc". Copy and paste the contents of the attached file into it. Save the file.

- Open the existing "Plugin.rsc" file, and copy and paste the contents of the attached file into it.

- When rerunning the main function in the "Plugin.rsc" file, you may encounter some error messages regarding missing "AST.rsc" and "Implode" files. However, these errors can be disregarded.

- Open the language instance file “spec1.tdsl”.
- At the top of the “spec1.tdsl”, you should now see “Generate text file”. Click on “Generate text file” to generate and open the “generator4.txt” file.

In this section, we have demonstrated that defining an abstract syntax is not strictly required. With less effort, it is possible to create a code generator based on concrete syntax.

8 Validation

In order to enhance the language instance and ensure its correctness, we can implement validation rules to provide feedback to DSL users. In this section, we will define validation rules using concrete syntax to check for the following three conditions:

- Payment Limit: We want to ensure that payments do not exceed 10000 euros. If a payment exceeds this budget limit, we will issue a warning to the user.
- Priority Conflict: To maintain consistency, we will raise an error if there are two or more tasks with the same priority. This helps avoid ambiguity and conflicting priorities
- Duration Conversion: We will verify if durations specified in minutes can be converted to hours. This check ensures that durations are expressed consistently and can be easily understood by users.

By implementing these validation rules, we can enhance the DSL by providing real-time feedback on the language instance.

```
Summary check(Planning p) {
    overLimit = {<"<a>", pay.src> | /pay:(PaymentAction)
        `Pay <INT a> euro` := p, toInt("<a>") > 10000};
    tasks = {<"<prio>", prio.src> | /(Task)
        `Task <Action action> priority :<INT prio> <Duration? duration>` := p};
    tasksWithSamePrio = {<n1, p1> | <n1, p1> <- tasks, <n2, p2> <- tasks, n1 == n2, p1 != p2};
    durations = {<"<dl>", dur.src> | /dur:(Duration)
        `duration :<INT dl> <TimeUnit unit>` := p, "<unit>" == "min", toInt("<dl>") % 60 == 0};

    return summary(p.src,
        messages = {<l, warning("There is a budget limit of 10000. So <e> is too big. ", l)>
                  | e <- overLimit<0>, l <-overLimit[e]} +
                  {<l, error("Priorities need to be unique: <e> is used somewhere else. ", l)>
                  | e <- tasksWithSamePrio<0>, l <-tasksWithSamePrio[e]} +
                  {<l, warning("Rewrite duration in <toInt(e)/60> hours. ", l)>
                  | e <- durations<0>, l <-durations[e]}
    );
}
```

Listing 11: Validation using concrete syntax

The implementation of the three validation rules is provided in Listing 11. The code consists of two parts: extracting the facts and creating the corresponding messages.

In the first part, we use a comprehension based on concrete syntax to iterate over each PaymentAction nonterminal. We check if the amount exceeds 10000

euros by casting the value "a" to an integer via a string. If the amount is higher than the limit, we store a tuple containing the amount as a string and its source location in the language instance within the "overLimit" variable.

Next, we address the priority conflict rule. We start by storing the priority and its source location for all tasks in the "tasks" variable. Then, we create a new set called "tasksWithSamePrio" to store unique tasks with the same priority.

In the second part, we generate the messages based on the extracted facts. Each message is assigned a severity, which can be an error or a warning.

```
set[LanguageService] contribs() = {
    parser(start[Planning] (str program, loc src) {
        return parse(#start[Planning], program, src);
    }),
    lenses(rel[loc src, Command lens] (start[Planning] p) {
        return {
            <p.src, gen4(p.top, title="Generate text file")>
        };
    }),
    summarizer(Summary (loc _, start[Planning] p) {
        return check(p.top);
    }),
    executor(exec)
};
```

Listing 12: Plugin.rsc

To enable the validation functionality, add the summarizer and call the check function from Listing 11, as shown in Listing 12. Take the following steps:

- Create and open a new file named "Checker.rsc". Copy and paste the contents of the attached file into it. Save the file.
- Open the "Plugin.rsc" file and replace its contents with the attached file. Save the file.
- Open the "spec1.tdsl" file and replace its contents with the attached file. Save the file.
- Rerun the main function in the "Plugin.rsc" file.
- Open the "spec1.tdsl" file, add a space somewhere within the file and save it. After saving the file, the validation results should be visible similar to Figure 12.

Please note that it is crucial for the project's directory path and project name to consist of lowercase characters. If any uppercase characters are present, the validation rules may not function properly.

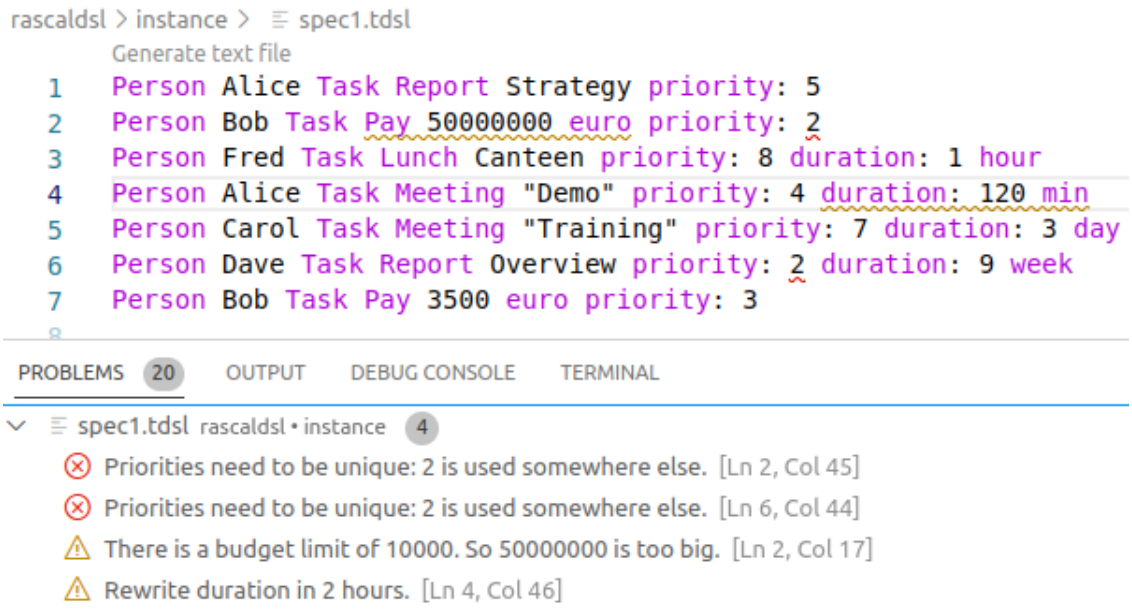


Figure 12: Validation rules

9 TypePal

To utilize TypePal for creating supported validation rules and definition/usage relations in our language, we need to install it. Follow these steps to install TypePal:

- Add TypePal to the project's POM file. Copy and paste the contents of the attached file into the "POM.xml" file.
- If "Maven for Java" by Microsoft is not already installed, install it. Navigate to the Marketplace, go to *View* → *Extensions* and search for

```
<dependencies>
  <dependency>
    <groupId>org.rascalmpl</groupId>
    <artifactId>rascal</artifactId>
    <version>0.33.3</version>
  </dependency>
  <dependency>
     <groupId>org.rascalmpl</groupId>
     <artifactId>typepal</artifactId>
     <version>0.8.6</version>
  </dependency>
</dependencies>
```

Listing 13: Dependencies in POM file

"Maven for Java". Press the install button to proceed with the installation.

- Once Maven is installed, we can install the dependencies. Right-click on the POM file and choose *Open in Integrated Terminal*. This will open a command prompt.

- In the command prompt, type the following command: $mvn\ -U\ install$. This command will download and install the required dependencies.

rascaldsl > META-INF > RASCAL.MF

```
1  Manifest-Version: 0.0.1
2  Project-Name: rascaldsl
3  Source: src/main/rascal
4  Require-Libraries: |lib://dsl-project|, |lib://typepal|
5
```

Figure 13: Updated RASCAL.MF file

- Open the "META-INF" folder and locate the "RASCAL.MF" file. Add ,|lib://typepal| after |lib://dsl-project|.

After completing these steps, TypePal will be installed in your project, enabling you to use its features for creating supported validation rules and establishing definition/usage relations between language features.

To incorporate the desired changes in your project, you need to update the grammar by introducing a section at the top of the file for defining Persons. Within this section, you will specify the properties of a Person, such as Role and age, using the appropriate syntax and data types. After defining the Person properties, you will update the feature to describe the tasks and establish a relationship with an organizing person. This means creating a rule that associates tasks with an existing Person who serves as the organizer. It is important to ensure that only already defined Persons can be used as organizers. Additionally, you will need to modify the code generator to include the properties of the organizer when generating text for the tasks. This can be achieved by utilizing the definition/usage relation created with TypePal.

Recall Listing 1 where we had a nonterminal PersonTask. In the updated grammar (Listing14), we have made modifications to the nonterminal PersonTask. It has been divided into two separate nonterminals: Person and Task. The Person nonterminal represents an individual and includes properties such as name, role and age. The role can be either a Manager or an Employee. The Tasks nonterminal remains similar to the previous grammar, but now includes

```
start syntax Planning
    = planning:
    'Persons:' Person+ persons
    Task+ tasks
;
syntax Person
    = person: ID name '{' Role role ',' 'age' INT age '}'
;
syntax Role
     = manager: 'Manager'
     | employee: 'Employee'
;
syntax Task
     = task:
    'Task' Action action
    'person' ID name
    'priority:' INT prio
    Duration? duration
;
```

Listing 14: Changes to Syntax.rsc

an additional property: the person associated with the task. The intention is to ensure that the person referenced in a Task is defined in the Persons list. To verify this relation, we employ TypePal to create a checker that validates the consistency between Persons and Tasks.

```
Persons:
Alice { Manager, age 40 }
Bob { Employee, age 35 }
Task Report Strategy person Alice priority: 5
Task Pay 5000 euro person Bob priority: 2
```

Listing 15: spec2.tdsl

The updated grammar is exemplified in Listing 15, which showcases a language instance reflecting the modifications we made.

In Listing 16, we can observe the TypePal checker file, which introduces the keyword **extend** used to extend one module from another. In this case, the Checker module inherits from the TypePal module. The file defines several Abstract Data Types (ADTs) that override the default types provided by TypePal.

One such ADT is IdRole, which includes a constructor called personId. Similarly, the AType ADT has a constructor called personType. Additionally, the custom Person ADT is defined, which includes fields for role and age, with default values specified. Another TypePal ADT called DefInfo is present, featuring an optional field of type Person. The field is optional because the list can have zero or one element.

```
extend analysis::typepal::TypePal;

data IdRole = personId();
data AType = personType();
data Person = person(str role = "Employee", int age = 0);
data DefInfo(list[Person] person = []);

void collect(current: (Person) `<ID name> { <Role role> , age <INT age> }`, Collector c) {
     dt = defType(personType());
     dt.person = [person(role="<role>", age=toInt("<age>"))];
     c.define("<name>", personId(), name, dt);
}
void collect(current: (Task) `Task <Action action>
                        'person <ID name>
                        'priority: <INT prio>
                        '<Duration? duration>`, Collector c) {
     c.use(name, {personId()});
}
public TModel TModelFromTree(Tree pt) {
    if (pt has top) pt = pt.top;
    TypePalConfig a = getModulesConfig();
    c = newCollector("collectAndSolve", pt, a);
    collect(pt, c);
    return newSolver(pt, c.run()).run();
}
```

Listing 16: Checker.rsc

Following the ADT definitions, there are two overloaded collect functions that match concrete syntax patterns. The first collect function matches a Person and creates a variable “dt” of type DefInfo. It assigns the defType as the personType ADT. The function then creates a list of one person with role and age and assigns it to the person field of DefInfo. This information is stored as a person definition, including its name, in the collector using the define function. The second collect function matches a Task and stores the name of the used person in the collector using the use function.

The TModelFromTree function establishes a solver that iteratively traverses the parse tree, attempting to match the two collect functions and storing all the collected information in the collector. After solving the collected information, a TModel is created, which will be utilized in a subsequent phase.

In Listing 17, we can observe the changes made to the code generator introduced in Section 7. The following modifications have been made:

- The printTaskWithoutDuration function now includes a TModel (tm) as a parameter. In comparison to the previous version, the concrete syntax pattern within the comprehension has been adjusted to accommodate the new grammar. Additionally, a new function called printOrganizer is

```
str printTaskWithoutDuration(cst, tm) {
    rVal = [];
    for (<a, m> <- { <action, name> | /(Task)
           `Task <Action action> person <ID name> priority: <INT prio> <Duration? duration>`
           := cst }) {
        rVal += "<printOrganizer(m, tm)> <printAction(a)>";
    }
    return intercalate(" ,\n", rVal);
}
str printOrganizer(name, tm) {
    DefInfo defInfo = findReference(tm, name);
    if (p <- defInfo.person) {
        return "Organizer is: <name>, role: <p.role>, age: <p.age> -\>";
    }
    throw "Fix references in language instance";
}
DefInfo findReference(tm, use) {
    defs = getUseDef(tm);
    if (def <- defs[use.src]) {
        return tm.definitions[def].defInfo;
    }
    throw "Fix references in language instance";
}
```

Listing 17: Generator.rsc

invoked within the string added to "rVal".

- The printOrganizer function is responsible for printing the organizer, including their name, role and age. In the Task definition, we have an usage of a Person, which only includes the name. However, we also need to retrieve the role and age properties of the Person. To accomplish this, we utilize the findReference function, which returns the DefInfo ADT. If a person is present in DefInfo, it is stored in the "p" variable. From this variable, we can access the role and age fields and use them to construct a string representing the organizer.

- The findReference function is used to retrieve definition/usage relations. It is employed to locate the definition of a usage. This information is obtained from the TModel. When a definition does not exist for a usage, an exception is thrown using the **throw** keyword.

In order to display the definition/usage relations in the GUI, a summarizer has been added in Listing 18. By utilizing TypePal in the GUI, users receive feedback when using an undefined Person. Additionally, it becomes possible to navigate from a Person name in a Task to the definition of that Person by using Ctrl+Click. To implement these functionalities, the following modifications

```
Summary tdslSummarizer(loc l, start[Planning] input) {
    tm = TModelFromTree(input);
    defs = getUseDef(tm);
    return summary(l, messages =
        {<m.at, m> | m <- getMessages(tm), !(m is info)}, definitions = defs);
}
set[LanguageService] contribs() = {
    parser(start[Planning] (str program, loc src) {
        return parse(#start[Planning], program, src);
    }),
    summarizer(tdslSummarizer)
};
```

Listing 18: Plugin.rsc

were made to "Plugin.rsc":

- The tdslSummarizer function is introduced. This function is responsible for providing feedback messages and establishing the definition/usage relations. It creates a TModel and utilizes the information from the TModel to generate a Summary.
- The contribs function is modified to invoke the tdslSummarizer function. This ensures that the summarizer is called and integrated into the plugin's functionality.

To have the TypePal functionality and invoke the updated code generator from the GUI, follow these steps:

- Create an empty file named "Generator.rsc". Copy and paste the code generator's contents of the attached file into our newly created "Generator.rsc" file.
- Create an empty file named "spec2.tdsl". Copy and paste the contents of the language instance, found in the attached file, into our newly created "spec2.tdsl" file.
- Copy and paste the contents of the grammar definition, found in the attached file, into the "Syntax.rsc" file.
- Copy and paste the TypePal checker's contents of the attached file into the "Checker.rsc" file.
- Copy and paste the contents of the modified plugin, from the attached file, into the "Plugin.rsc" file.

- Run the main function of "Plugin.rsc" in a new Rascal terminal.

Now let's explore some of TypePal's features:

- Hover your mouse pointer over the task where Bob is the organizer while holding down the Ctrl key. You will notice that Bob becomes highlighted in blue with an underline. Press Ctrl+Click on Bob to instantly jump to its definition.
- Alternatively, you can also navigate from usage to definition by placing the cursor on Bob and pressing F12, or by Right-Clicking and selecting "Go to Definition".
- TypePal also detects the usage of undefined Persons and highlights it as an error. To see this in action, modify Bob to Bobb and save the file. You will observe a red line appearing under Bobb, indicating the error.

Please note that it is crucial for the project's directory path and project name to consist of lowercase characters. If any uppercase characters are present, the validation rules may not function properly.

In this section, we provided a brief introduction to TypePal. For more detailed information and comprehensive examples of TypePal usage, we recommend referring to TypePal's official documentation and examples. These resources will provide a deeper understanding of TypePal's features and functionalities.

10 Tips & Tricks

In this tips & tricks section, we offer additional resources for more in-depth information about Rascal. Next some useful example code is provided. Additionally, we provide insights into debugging your Rascal code.

10.1 Resources

- Rascal:
  https://www.rascal-mpl.org/
- Getting started:
  https://www.rascal-mpl.org/docs/GettingStarted/

- Cheat sheet:
  https://raw.githubusercontent.com/cwi-swat/rascal-cheat-sheet/master/sheet.pdf
- GitHub:
  https://github.com/usethesource/rascal
- Example DSL:
  https://github.com/cwi-swat/yop-rascal-nl
- Stackoverflow:
  https://stackoverflow.com/questions/tagged/rascal
- TypePal:
  https://www.rascal-mpl.org/docs/Packages/typepal/
- TypePal examples:
  https://github.com/usethesource/typepal/tree/master/src/examples

10.2 Example code

When developing a code generator for a DSL, you often create prototypes of the desired source code. To seamlessly incorporate these prototypes into your code generator, you can employ the provided function to automate this procedure.

10.3 Debugging

Rascal has experimental support for debugging in VS Code. You can try it by downloading the vsix file in the artifacts zip-file. The download url:

- https://github.com/usethesource/rascal-language-servers/suites/14635107905/artifacts/830068550

Once the vsix file is installed, you can enable debugging by typing the following command in the terminal ":set debugging true". If the debugger distribution causes issues, you can downgrade your Rascal extension in VS Code.

Another approach to debug your code is to try code examples or groups of functions in the terminal. In the terminal, you can easily inspect the contents of variables. Another approach is to add debug "println" function calls.