CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Sempar — A Tool for Construction-Time Specification of Linting in Parsers

Master's thesis in Computer science and engineering

Lucas Glimfjord
Hugo Simonsson

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Master's thesis 2023

Sempar — A Tool for Construction-Time Specification of Linting in Parsers

Lucas Glimfjord
Hugo Simonsson

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Göteborg, Sweden 2023

Sempar — A Tool for Construction-Time Specification of Linting in Parsers

Lucas Glimfjord
Hugo Simonsson

Supervisor: Patrik Jansson, Department of Computer Science and Engineering
Advisor: Karl Bristav, Antura AB
Examiner: David Sands, Department of Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Sempar — A Tool for Construction-Time Specification of Linting in Parsers

LUCAS GLIMFJORD, HUGO SIMONSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

A linter is a good tool to aid developers, however they are most common in larger, well-used languages such as Java or Python. They aid by giving feedback to the developer regarding their code, for example about code style or bad practices. A common problem is that it is hard to find linters for small languages or even more so for domain specific languages. In order to make it easier to implement linting rules for these languages we have implemented a tool that gives the language developer the ability to add linting rules when creating the language specification. We go into detail about how the tool works as well as show it in action with a case study. The case study is used to demonstrate how the tool can be used to implement a new syntax for an existing domain specific language used by Antura AB in their product.

Keywords: Domain specific language, linting, static analysis, compiler compiler, meta compiler, static checking.

Contents

List of Figures

List of Listings

1

Introduction

Code quality relies on many factors such as the competence of the programmer, tooling, type checking and linting. Our focus in this project lies in linting. Linters are tools that can flag errors in programs that are otherwise well-formed and well typed. The name 'linting' comes from a C-program called `lint` that examined C-programs for bugs, error prone constructions and unnecessary instructions [1]. While the most common case for linting aims at preventing future errors, some linters also flag issues in code style.

Commonly used linters such as ESLint for Javascript, PyLint for Python or clippy for Rust are usually separate programs developed separately from the main language, even if the line between a compiler and linter is being more and more blurred as the industry moves towards stricter checking in order to avoid costly bugs. Having a separate linter works well for large languages such as the aforementioned Javascript, Python or Rust. However, linters are less common for both smaller languages and entirely bespoke DSLs. These languages would still benefit from having such tools to give feedback to the user in how the code could be improved, but the smaller user base means that less energy is spent on infrastructure such as linters.

Therefore, we have developed a tool called Sempar for F# that allows creating linting rules while creating the parser of a language. This allows the parser to return diagnostics in the form of warnings and errors to be raised during parsing of the written code in the language. The tool is written in F#, a functional programming language in the ML family on Microsoft's .NET environment. For lexing and parsing we use the libraries FsLex and FsYacc, based on the original Yacc as proposed by Johnson [2]. FsLex and FsYacc are explained in detail in section 2.1. The linting rules that can be applied could either cover the language as a whole with, for example, warnings about common misspellings. They could also be more specific and applied only to certain constructions. For example, if a typesetting DSL similar to the well-known DSL LaTeX is created, a more specific linting rule could be that paragraphs should not be too long, or to warn if the title of a report is empty.

To show the tool working we have done a case study where we have implemented a DSL used by Antura AB with our tool in chapter 4. We first explain the domain, the DSL's semantics and the current JSON syntax used by Antura. This is followed by details regarding the language specification written for Sempar, the relevant steps Sempar takes to create the parser and finally we also have an example input for the generated parser.

2

Background

In this section we provide a background to some concept used in the report. We shortly explain some concepts that are used related to the language F#. Specifically, we will introduce F#'s computation expressions and two libraries we use for lexing and parsing: FsLex & FsYacc.

2.1 The FsLex & FsYacc libraries

FsLex & FsYacc are two F# libraries for lexing and parsing respectively. They are based on the popular parser generator Yacc & Lex [2] and is heavily inspired by the OCaml implementations OCamlLex & OCamlYacc.

2.1.1 FsLex

The syntax of FsLex is based on the original Lex specification from 1975 [3]. A FsLex file has three parts: arbitrary F# code, patterns used in rules and the lexing rules. The arbitrary F# code is at the top of the file inside `{ }`. Then, the actual lexing code follows. Here there are two main things to define that can be used as shorthand later on, written in regex. A pattern for a number can be defined as `let number = ['0'-'9']+`. Finally, matching rules can be defined. The main rule is called `read` and is the first rule in the file. An example of `read` can be seen in lines 1–6 in listing 2.1. In it, the structure of rules is shown. The `read` rule returns a token (in all uppercase) that can be used in the parser. The tokens can have arguments that have for example strings or integers inside.

```
1 rule read =
2   parse
3   | "%%" { DOUBLEPERCENT }
4   | "|"  { PIPE }
5   | "{"  { CODE (read_bracket "" 0 lexbuf) }
6 // ... (More cases omitted)
```

Listing 2.1: An excerpt from a FsLex file, defining three tokens.

2.1.2 FsYacc

The FsYacc syntax is inspired by the original Yacc specification [2]. It uses the same core concept of defining grammar rules that can then be used in other grammar rules, creating a tree of rules that defines the parsing order. We first define the name of the rule, which can then be used in other rules. The name is followed by a colon and a sequence of tokens that determine what the parser should look for. This can be other rules or predefined tokens from the lexer. Finally, inside of the curly braces, generic F# code can be written that is run when the rule matches. As part of this code the variables `$1` to `$n` will be defined and refer to the value of the nth ordered token.

As an example we can look at a case of how addition is parsed when part of a bigger rule to parse arithmetic expressions. `expr: expr PLUS expr { Add ($1, $3) }`. In this case the code gives the first and third token to the type constructor Add and returns it. This rule only contains one case to match on. However, each rule can contain multiple cases. This is achieved by putting a `'|'` before the tokens for each case. The parser then tries the cases in order until it finds a case that matches, returning the result from that case.

Other than the rules, there is some boilerplate at the top of the file. Firstly, generic F# code can be put at the top between `%{` and `%}`. This is useful for importing code from other places or for defining functions that can be used for the output results of a rule where multiple rules have the same logic. Finally there are also definitions for tokens from the lexer, which rule to start parsing at, and the output type that needs to be defined for the parser to work correctly.

2.2 F#'s computation expressions

F# includes a language feature called *Computation expressions*. The syntax allows users to easily write computations that can be sequenced and combined using regular control structures, much like do-notation in Haskell. They can represent many different structures such as monads, monoids and applicative functors. Some examples of builtin computation expressions are `async` and `sequence`. One of the most common operations is `let!`, which is equivalent to Haskell's `<-` in do-notation. `let!` and `<-` desugars the same way, to a call to `.Bind()` or `>>=`. It is possible to define your own computation expressions by creating a class with the methods that the operations desugar to. As an example, if one wants to support `let!` in a custom computation expression, one needs to define the previously mentioned `.Bind()` [4]. Concretely, computation expressions are used in this project in order to simplify code generation. We use it to generate code that uses our `Diagnostics<'a>` type in an idiomatic way, much like do-notation. An example of this use can be seen on line 69–75 in listing 3.6.

3

Sempar: Creating parsers with embedded linting rules

In order to add linting rules to a parser specification we have created a tool called Sempar. Sempar allows a language developer to add these aforementioned linting rules as arbitrary F# code as a part of parsing.

3.1 General overview

Sempar is interacted with in two ways: by a language developer that wants to make a new language and by the user that wants to use such a language. The language developer creates the specification using the FsLex & FsYacc syntax with the addition of linting rules. Sempar then transforms the specification, embedding the linting rules into the parser and encapsulates values with the type constructor Diagnostics, explained in section 3.4. Finally this is converted into a parser that, when given code, returns an abstract syntax tree (AST) with diagnostics as specified in the language specification. In order to benefit from the added diagnostics information the developer also has to implement diagnostic handling before using the AST in their compiler or interpreter. An overview of the structure can be seen in figure 3.1.

For a user of the language created by the language developer, code can be written using the syntax defined in the specification and then given to the parser. This code is then parsed into an AST with diagnostics information before being sent through the diagnostics handling previously mentioned, and then run through the compiler or interpreter to receive the final result.

In this chapter, a simple example with arithmetic expressions will be used as an aid for understanding. The expressions can contain addition, subtraction, multiplication, division, parentheses and variables. The program takes an expression, parses it and simplifies it. The output is an AST of the simplified expression. The listings will often be excerpts, where missing parts will be marked with `[...]`. The full files are available in appendix A. Worth noting is that multiplication and division does not have a higher precedence in the example in order to make the code easier to understand.

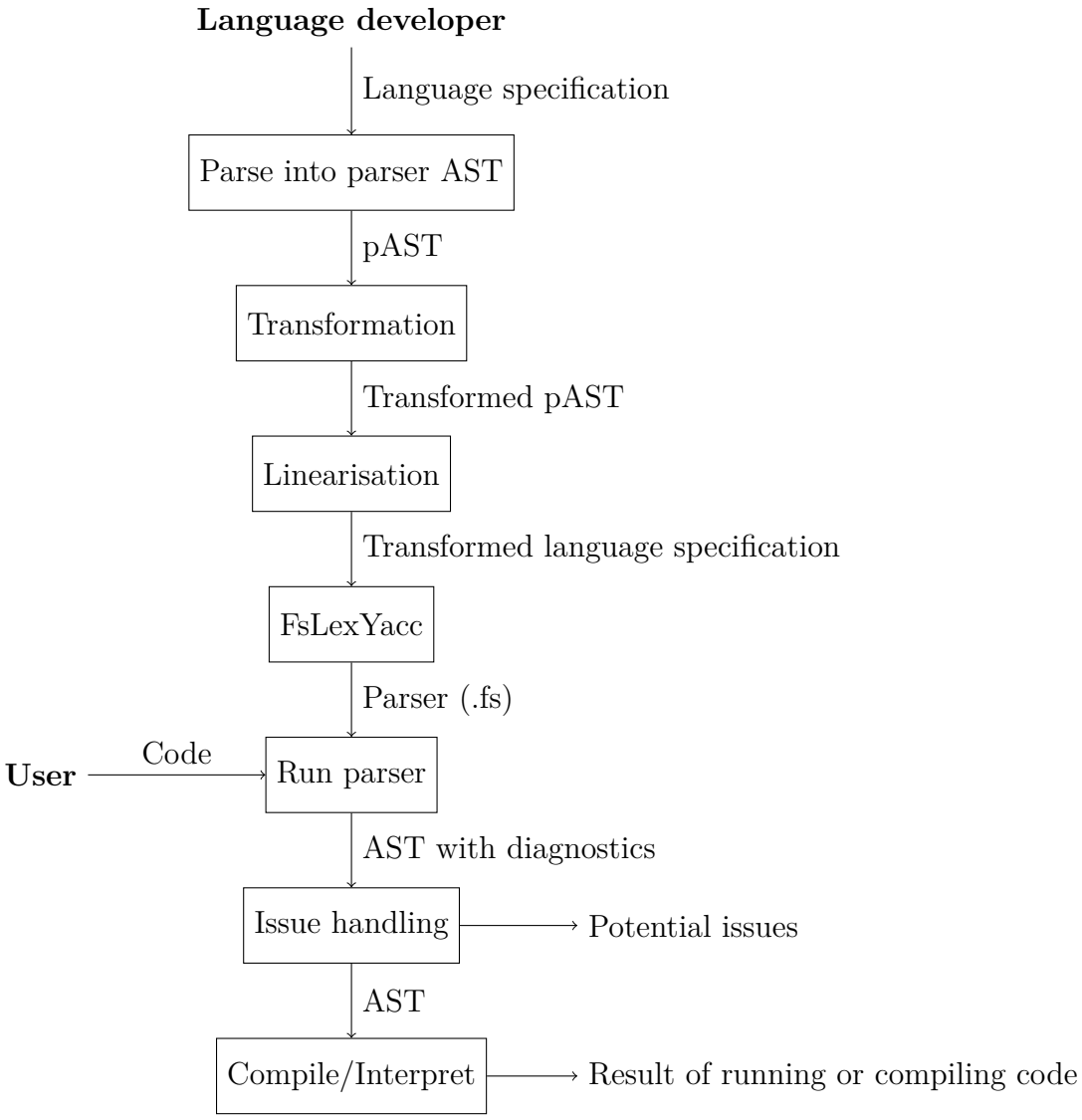


Figure 3.1: An overview of Sempar's structure.

```
1  {
2  module Lexer
3  open FSharp.Text.Lexing
   [...other preamble code...]
8  let newline (lexbuf: LexBuffer<_>) =
9    lexbuf.StartPos <- lexbuf.StartPos.NextLine
10 }
11 let digit = ['0'-'9']
12 let frac = '.' digit*
13 let exp = ['e' 'E'] ['-' '+']? digit+
14 let float = '-'? digit+ frac? exp?
   [...other patterns...]
23   parse
24   | white    { read lexbuf }
25   | newline  { newline lexbuf; read lexbuf }
26   | var      { VAR (lexeme lexbuf) }
27   | float    { NUM (float (lexeme lexbuf)) }
28   | '+'      { PLUS }
29   | '-'      { MINUS }
30   | '*'      { TIMES }
   [...other cases...]
```

Listing 3.1: Example of lexer specification for simple arithmetic expressions with addition, subtraction, multiplication and division of numbers and variables.

3.2 Language specification

The language specification is defined using the Yacc and Lex syntax used by FsLex & FsYacc with the addition of linting rules. Linting rules are defined on each case of each rule. The definition is done by starting the line above with `//!` followed by an F# expression of the type `Diagnostics<'a>`. An example of such a rule can be seen on line 20–21 in listing 3.2.

These expressions can be manually written or make use of our predefined functions for generating warnings or errors. Either `warn` or `error` of the type `string -> Diagnostics<'a>`, that unconditionally output the string they receive as a warning or error respectively. Alternatively `warnWhen`, `errorWhen`, `warnUnless`, and `errorUnless` of the type `string -> bool -> Diagnostics<'a>`. The functions ending in `unless` emits the string as a warning or error if the condition is false while the functions ending in `when` do the same if the condition is true.

A specification of the lexer and parser for the example described above can be seen in listing 3.1 and listing 3.2 respectively.

```
%{
open ArithAST
%}
%start expr
%token <float> NUM
[...more token definitions...]
%left PLUS MINUS TIMES DIV
%type <Arith> expr
%%
expr:
  | LPAR expr RPAR  { $2 }
[...addition, subtraction and multiplication...]
  //! errorWhen "Division by 0 will not work." ($3 = Num 0)
  //! warnWhen "Division by 1 will be simplified." ($3 = Num 1)
  | expr DIV expr   { Div ($1, $3) }
[...variables and literal numbers...]
```

Listing 3.2: Example of parser specification for simple arithmetic expressions with addition, subtraction, multiplication and division of numbers and variables.

3.3 Parsing the language specification

For parsing the specification, FsLex & FsYacc is used. The specification is turned into a parser AST (pAST) that represents the language specification. The pAST is an internal representation in Sempar that is the same regardless of how the language developer defines their language. The types used for the internal representation can be seen in listing 3.3.

In short, the representation is separated into a preamble and rules. The preamble contains boilerplate needed for FsYacc, such as token definitions from the lexer, the output type from the parser and the rule to start with. A rule contains a name and cases where each case contains tokens and the case's output code as needed by FsYacc. Additionally it also contains our linting rules, called constraints in the type. An example where the representation is used for the arithmetic expression example can be seen in listing 3.4.

3.4 Diagnostics

The output from the generated parser will be of the type `Diagnostics<'a>`. Diagnostics behaves lika a monad in that it supports the two functions `.Bind()` and `.Return()`. In our implementation the type argument is the original output type of the language specification. `Diagnostics<'a>` enhances the original type with warnings and errors generated during the parsing passes. The definition of Diagnostics can be seen in listing 3.5.

```
type Constraint = Constr of string
type Code = Code of string
type Token = Token of string
type RuleCase = {
        tokens: Token list;
        code: Code;
        constraints: Constraint list;
}
type Rule = { name: string; cases: RuleCase list; }
type Rules = Rule list
type PreaItem = { name: string; value: string list; }
type Preamble = { preaCode: PreaCode; preaItems: PreaItem list; }
type FSY = { preamble: Preamble; rules: Rule list; }
```

Listing 3.3: The types that represents the pAST used internally in Sempar, see listing 3.4 for an example value of the type.

```
{ preamble =
   { preaCode = PreaCode "open ArithAST "
     preaItems =
      [{ name = "start"
         value = ["expr"] };
       { name = "token"
         value = ["<float>"; "NUM"] };
[...other tokens...]
       { name = "left"
         value = ["PLUS"; "MINUS"; "TIMES"; "DIV"] };
       { name = "type"
         value = ["<Arith>"; "expr"] }] }
    rules =
     [{ name = "expr"
        cases =
         [{ tokens = [Token "LPAR"; Token "expr"; Token "RPAR"]
            code = Code " $2 "
            constraints = [] };
[...addition, subtraction and multiplication...]
          { tokens = [Token "expr"; Token "DIV"; Token "expr"]
            code = Code " Div ($1, $3) "
            constraints =
             [Constr
                "errorWhen "Division by 0 will not work." ($3 = Num 0)";
              Constr
                "warnWhen "Division by 1 will be simplified." ($3 = Num 1)"] };
[...variables and literal numbers...]
```

Listing 3.4: Example of the specification in listing 3.2 turned into a pAST.

```
1 type Diagnostics<'a> =
2     | Errors of Error list
3     | Warnings of ('a * Warning list)
4     | OK of 'a
5
```

Listing 3.5: Definition of Diagnostics.

The errors and warnings come from the linting rules defined in the language specification. Semantically, an error means that there is some kind of fault during parsing that results in it not making sense to generate an output because running the code will crash or otherwise not work. In listing 3.2, dividing by zero raises an error since evaluating the program would result in undesirable behaviour. Hence, raising an error means that no output will be generated and only the error will be passed back.

In contrast, warnings mean that there might be a problem but the code should still work. Raising a warning therefore means that the result will still be returned along with the raised warnings. Warnings can therefore be ignored while errors have to be fixed to get a parse result.

3.5 Transformation

The pAST created from parsing the specification is transformed in order to embed the linting rules and encapsulate the output in the Diagnostics monad. Listing 3.6 illustrates the result of transforming the example parser's AST. In the preamble the Diagnostics module is imported and the output type is changed from `'a` to `Diagnostics<'a>`, a value of type Diagnostics with the type parameter being the type that the developer defined as the result from the parser. The change of output type makes it possible to leverage the properties of the Diagnostics type so that we can embed the linting rules such that they can later be evaluated during parsing in the defined language.

The linting rules are embedded by using the computation expression syntax in F# as explained earlier in section 2.2. The code of each case is changed to output a `Diagnostics<'a>`. To achieve this, the variables corresponding to the tokens `$1` throuh `$n` are redefined by replacing the `$` with `semparVar`. If the token is another rule, we know that it will return a Diagnostics. Therefore we have to use `let!` to access the wrapped value. In contrast, if the token is a construct from the lexer, we know that the type is not a Diagnostics and a standard variable declaration is done, using `let`. The result of the redefinitions can be seen on line 70–71 in listing 3.6.

```
{ preamble =
    { preaCode = PreaCode "open ArithAST
  open Diagnostics"
      preaItems =
       [{ name = "start"
          value = ["expr"] };
        { name = "token"
          value = ["<float>"; "NUM"] };
[...other tokens...]
        { name = "left"
          value = ["PLUS"; "MINUS"; "TIMES"; "DIV"] };
        { name = "type"
          value = ["<Diagnostics<Arith>>"; "expr"] }] }
    rules =
     [{ name = "expr"
        cases =
         [{ tokens = [Token "LPAR"; Token "expr"; Token "RPAR"]
            code =
             Code "sempar {
    let! semparVar2 = $2
    return ( semparVar2 )
  }"
            constraints = [] };
          { tokens = [Token "expr"; Token "TIMES"; Token "expr"]
[...addition, subtraction and multiplication...]
            code =
             Code
               "sempar {
    let! semparVar1 = $1
    let! semparVar3 = $3
    do! errorWhen "Division by 0 will not work." (semparVar3 = Num 0)
    do! warnWhen "Division by 1 will be simplified." (semparVar3 = Num 1)
    return ( Div (semparVar1, semparVar3) )
  }"
[...variables and literal numbers...]
```

Listing 3.6: Transformed pAST based on the pAST in listing 3.4.

```
%{
open ArithAST
open Diagnostics
%}
%start expr
%token <float> NUM
[...other tokens...]
%left PLUS MINUS TIMES DIV
%type <Diagnostics<Arith>> expr
%%
expr:
| LPAR expr RPAR {sempar {
  let! semparVar2 = $2
  return ( semparVar2 )
}}
[...addition, subtraction and multiplication...]
//! do! errorWhen "Division by 0 will not work." (semparVar3 = Num 0)
//! do! warnWhen "Division by 1 will be simplified." (semparVar3 = Num 1)
| expr DIV expr {sempar {
  let! semparVar1 = $1
  let! semparVar3 = $3
  do! errorWhen "Division by 0 will not work." (semparVar3 = Num 0)
  do! warnWhen "Division by 1 will be simplified." (semparVar3 = Num 1)
  return ( Div (semparVar1, semparVar3) )
}}
[...variables and literal numbers...]
```

Listing 3.7: Generated .fsy-file after the linting rules have been inserted.

3.6 Linearisation and FsLex & FsYacc

In order to generate the parser we need to generate a parser specification that FsLex & FsYacc can use. After the previous step, we have a new pAST. This new pAST needs to be linearised into a file that FsYacc can use. See listing 3.7 for the one generated for the arithmetic expressions example. The linearisation is done by each type in the parser AST having a member function which returns the code that would generate it. That is, defining an approximate inverse function to the parser. The functions are inverses as long as one disregards whitespace. After the linearisation, the generated `.fsy` is given to FsLex & FsYacc together with the lexer definition from the first step generating the final parser.

```
(VAL0 + (VAL1 - 0) * (5 / 5)) / ((1 * 1) + (VAL2 * 0))
```

(a) Input to example.

```
Warnings (Div (Mul (Add (Var "VAL0", Sub (Var "VAL1", Num 0.0)), Div (Num 5.0,
↪  Num 5.0)), Add (Mul (Num 1.0, Num 1.0), Mul (Var "VAL2", Num 0.0))),
 ["Subtracting 0 will be simplified."; "Multiplication by 1 will be simplified.";
↪  "Multiplication by 0 will be simplified."])
```

(b) `Diagnostics<Arith>` returned from parser.

```
Add (Var "VAL0", Var "VAL1")
```

(c) Result after simplification and deconstruction of the `Diagnostics<Arith>`.

Listing 3.8: Listings describing the structure of values returned after running the chapter's example.

3.7 Issue handling & compilation/interpretation

The implementation of how generated errors and warnings are presented to the user is left to the language developer. That is, there is no function of the type `Diagnostics<'a> -> 'a` that deconstructs the result of parsing. The language developer has to manually deconstruct and handle the different cases. This allows for more flexibility in where and how the language is used. It gives the possibility for the developer to, for example, decide how to get the warnings in their editor or if the language is used in a web application decide how to return the diagnostics through a REST API.

The developer also needs to implement how the AST is to be 'executed', if it should be compiled, transformed into something else, interpreted as is, or simply saved in a file.

In this chapter's example, the program that 'executes' the code handled the diagnostics by printing them, and then simplifies the AST before printing the result in the console as well. An example can be seen in listing 3.8. Sublisting a) describes an example input, sublisting b) describes the result after running the parser and sublisting c) describes the result after simplification.

3.8 Limitations

There are some limitations on how one can use Sempar. One of those is that the linting rules must be one line long. Longer rules should be written as a separate function, and called from the linting rule instead.

Sempar does not currently give any position information about where diagnostic messages appeared in the code. FsLex and FsYacc does provide functionality that provides this information, but we do not currently use it because of time constraints. That would have provided information about where in the file lexing is taking place, which we could've used. However, it could also have been useful to know which node in the syntax tree the error is at.

Since the linting rules are on each parsing rule without having knowledge of the environment they are in, they can only depend on the particular rule and its child parse rules. While it can do arbitrary logic on the parsing rule's children, the rules do not have access to their parents. Therefore, some kinds of linting rules are difficult to implement. As an example, take runtime errors. A linter should only raise an error if the code that triggers a runtime error will actually be run in order to not disallow valid programs. However, this is difficult when the linting rules only have access to their children.

This can be avoided by defining the rule on the parent instead since one can write arbitrary linting rules. However, this means that the rules are not where someone might expect them to be and that, if we had implemented position information, that the positions wouldn't be correct as they would be on the parent node and not the child node where it would be expected.

4

Case study: The Antura DSL

In this chapter we show how Sempar can be used in a real world scenario. The case study aimed to do two things: test Sempar by adding some proof of concept linting rules and make the Antura DSL more user-oriented by creating a new syntax for it. The end goal is to make the Antura DSL easier to use for non-programmers, who are the most common users.

We also give relevant context to the Antura system in combination with an example of the Antura DSL, followed by our implementation and the different steps generated by Sempar. Finally we end with a short reflection concerning the usefulness of the case study.

4.1 The Antura DSL

Antura currently has a JSON-based DSL that configures certain automated actions that occurs whenever some events happen. This section will briefly explain the domain and the semantics of the language. Then, there will be an example that shows the syntax of the language.

4.1.1 The domain — Antura

Antura is a project portfolio management system that allows organisations to manage their projects and project portfolios [5]. In the system, users can track the progress, budget and other such attributes related to their projects or portfolios of projects. A portfolio consists of several projects and provides a more high level view of the included projects. A project can have many different subparts, such as tasks, budgets and milestones. Both the individual project and each of these parts can have different *properties* depending on their type. For example, projects might have a status property, tasks might have start and end date properties, budgets might have several budget items and so on. Portfolios can have the same kinds of properties as projects.

4.1.2 The semantics of the DSL

The DSL, as mentioned before, configures automated actions that should occur in Antura. The actions take the value of one or more properties of a project, poten-

tially transforms them in some way, and sets another property to the result of the transformation. A property in this context is a general value connected to a project. They can be many different things, including dates, numbers and strings. The general semantics of the DSL is that a 'program' in the DSL defines a set of rules. Each rule is a combination of four parts: a predicate that defines which kinds of projects to apply the rule to, a selector that defines which properties to use as input, a function that might transform the input and a destination to output the result to. The transformations can currently only be applied to numeric values and can be any combination of the four ordinary arithmetic operations and grouping by parenthesis.

In pseudo code, evaluating a rule is approximately the following:

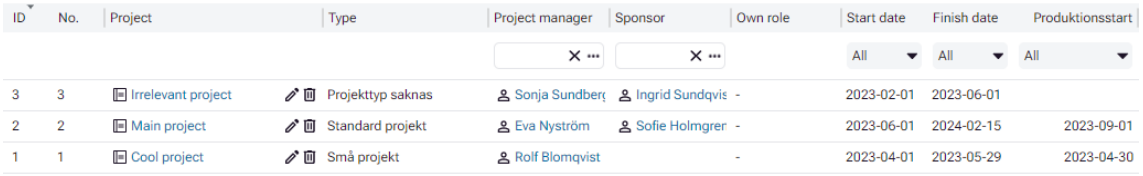
```
portfolio : [Project]
predicate : Project -> Bool
selectors : [Property -> Bool]
transform : [Property] -> Value
destination : Property

for each project in portfolio where the predicate matches:
  source values := empty list
  for each selector in selectors:
    if there is more than one property such that the selector matches:
      error
    else:
      append the value of the property that matches the selector to source values
  output value := transform(source values)
  set destination to output value
```

Quite a large limitation in this procedure is that this only works if the number of arguments to `transform` matches the number of properties that are matched. As an example, this means that there can't be any transforms that take a variable number of arguments. If a selector matches more than one property, there is an error.

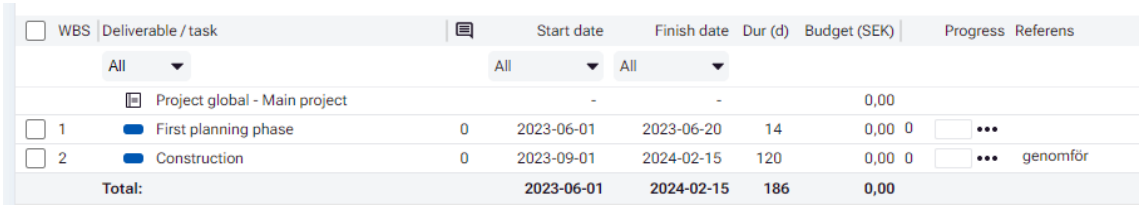
4.1.3 The syntax and an example

An example of the DSL can be seen on the left in listing 4.1 on page 19. The JSON begins with some boilerplate code that is always included (lines 2–3). Then, there is a list of rules, starting at line 4. In this example, there is only one rule. Each rule has a predicate that defines which projects this rule will be applied to (lines 6–13). In this case, it is valid for any project with the type 'Små projekt' or 'Standard projekt'. Then, the source of the values is defined (lines 14–24). Here, it will take the start date (line 22) from any task that has the property 'Referens' set to the value 'genomför' (lines 19–20). No transformation will be done (lines 25–27). That is, the transformation function is the identity function. The destination is a property in a project, in this case the property 'Produktionsstart' (lines 28–31). Screenshots from Antura that shows the effect of the rule defined in the example can be found in figure 4.1.



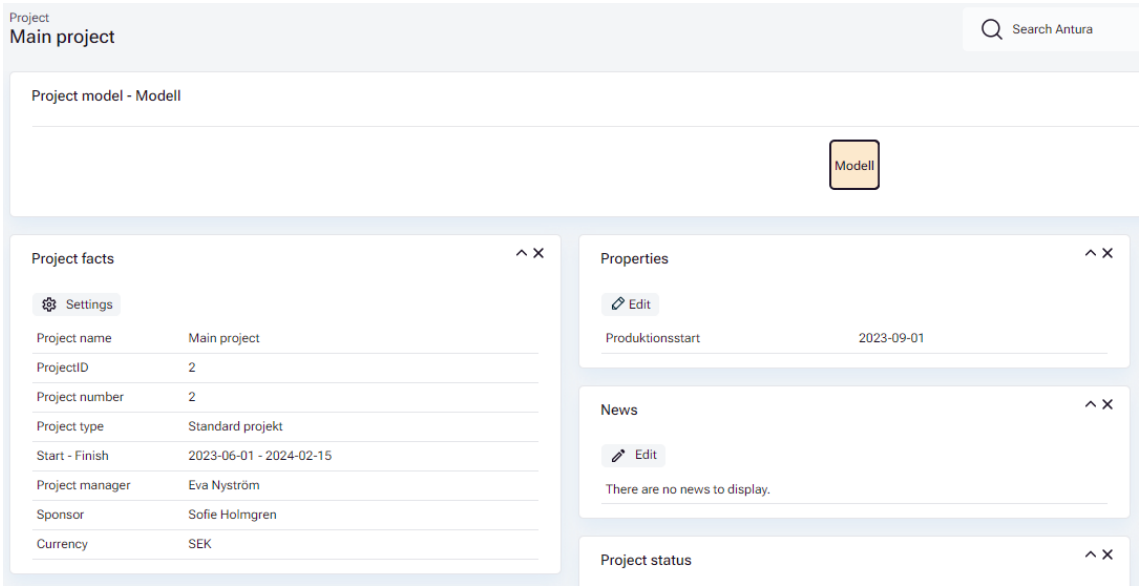
ID	No.	Project	Type	Project manager	Sponsor	Own role	Start date	Finish date	Produktionsstart
							All	All	All
3	3	Irrelevant project	Projekttyp saknas	Sonja Sundberg	Ingrid Sundqvis	-	2023-02-01	2023-06-01	
2	2	Main project	Standard projekt	Eva Nyström	Sofie Holmgren	-	2023-06-01	2024-02-15	2023-09-01
1	1	Cool project	Små projekt	Rolf Blomqvist		-	2023-04-01	2023-05-29	2023-04-30

(a) An image showing the list of projects in a portfolio, here three projects. Two of them, 'Main Project' and 'Cool Project' are affected by the example defined in listing 4.1, as can be seen from their project type being 'Standard projekt' and 'Små projekt' respectively.



WBS	Deliverable / task		Start date	Finish date	Dur (d)	Budget (SEK)	Progress	Referens
	All		All	All				
	Project global - Main project		-	-		0,00		
1	First planning phase	0	2023-06-01	2023-06-20	14	0,00 0		
2	Construction	0	2023-09-01	2024-02-15	120	0,00 0		genomför
	Total:		**2023-06-01**	**2024-02-15**	**186**	**0,00**		

(b) An image showing the list of tasks in 'Main project'. The project has two tasks, 'First planning phase' and 'Construction'. The 'Start date' property of the second of these tasks will be copied to the 'Produktionsstart' property of the project.



(c) A view of a single project, 'Main Project'. It shows the same information as figure 4.1a but shows that 'Produktionsstart' has been set to the expected value.

Figure 4.1: Screenshots from Antura, showing the results from running the rule defined in listing 4.1.

4.2 Our reimplementation

In our case study we implemented a new syntax for the Antura DSL, moving away from the JSON version described earlier in this chapter. The syntax is extended with a couple of different linting rules to show how they in practice are added. We also briefly discuss our findings from implementing the new syntax. The implementation and relevant intermediary representations can be found in appendix B. This section will discuss the design choices and point out relevant parts of the implementation.

4.2.1 Our syntax

Our new syntax aimed to model the underlying semantics while making the syntax easier to read. We also added some linting rules to show the functionality of Sempar. In listing 4.1, a comparison between our syntax (on the right) and the existing JSON syntax (on the left) can be seen. It clearly shows the relation between them and that our syntax is more compact and uses fewer special characters. After our new DSL is parsed, our implementation transforms it into the original JSON format that Antura uses.

4.2.2 Linting rules

Our implementation adds two linting rules as a proof of concept. We believe more could be added with better knowledge about how the language is used, however we refrained from this since it was outside our scope. The first rule gives an error if the precision value is negative. The second rule forces tasks to be either 'StartDate', 'EndDate' or 'Property'. Both rules can be seen in listing 4.2.

If these linting rules are not satisfied, the errors will be outputted instead of the final code since they are implemented as errors, not warnings. If the precision is negative the error `Precision must be positive` is returned. The second rule returns the error `Tasks must be certain values: StartDate, EndDate or Property` if the name of the task does not match one of the strings.

4.2.3 Data representation

The data representation used in the case study can be seen in listing 4.3. Together with the language specification, it is used to make an AST that models a 'program' in the Antura DSL. In listing 4.4, the representation of the input on the right side of listing 4.1 can be seen. From this AST, a transformation is done to the original JSON format that Antura uses. The output will be functionally identical to listing 4.1, though the ordering of the fields might differ.

```
{
  "schemaVersion": 1,
  "decimalPrecision": 2,
  "rules": [
    {
      "validFor": {
        "kind":
        ↪  "MultipleProjectTypes",
        "multipleProjectTypes": [
          "Små projekt",
          "Standard projekt"
        ]
      },
      "valueSources": [
        {
          "kind": "Task",
          "findTaskBy": {
            "kind":
            ↪  "PropertyNameValue",
            "taskPropertyName":
            ↪  "Referens",
            "taskPropertyValue":
            ↪  "genomför"
          },
          "taskSourceValue":
          ↪  "StartDate"
        }
      ],
      "transformation": {
        "kind": "NoTransformation"
      },
      "destination": {
        "kind": "ProjectProperty",
        "property":
        ↪  "Produktionsstart"
      }
    }
  ]
}
```

```
precision 2
rule Sätt automatiskt produktionsstart:
  project types:
    "Små projekt"
    "Standard projekt"
  sources:
    from task where Referens = genomför
    ↪  get StartDate
  transformation:
    none
  destination:
    project property Produktionsstart
```

Listing 4.1: A comparison between Antura's current JSON based syntax (left) and our syntax (right). The colours represent the same meaning on both sides.

```
36  rule_name:
37    |  { "" }
38    | ID rule_name { $1 + " " + $2 }
    [...other rules...]
67                | Errors es -> None
68    }
69
```

Listing 4.2: An excerpt of lines 36–38 and 67–69 of listing B.1 showing the linting rules concerning precision and allowed values of tasks.

```
1  open ArithAST
2
3  type Type = string
4  type Transformation = Arith
5  type Destination = Property of string
6  type Source =
7    Task  of (string * string * string)
8    | DecisionPoint of (int * string)
9    | NumericProjectProperty of (string * string)
10 type Rule = {
11     name: string option;
12     types: Type list;
13     sources: Source list;
14     transformation: Transformation option;
15     destination: Destination; }
16 type Rules = { precision: int option; rules: Rule list; }
```

Listing 4.3: The model used to build the data representation seen in listing 4.4.

```
1  {
2    precision = Some 2
3    rules = [
4      {
5        name = Some "Sätt automatiskt produktionsstart "
6        types = ["1 Små projekt"; "2 Standard projekt"]
7        sources = [Task ("Referens", "genomför", "StartDate")]
8        transformation = None
9        destination = Property "Produktionsstart"
10     }
11   ]
12 }
```

Listing 4.4: The AST representing the right side of listing 4.1.

4.3 Usefulness of the case study

Firstly, the relative simplicity of the Antura DSL was beneficial for us since the time to understand and implement the DSL was quite short. However, we did not have the deep domain knowledge needed to properly understand the nuances and probable use cases for the DSL. This meant that we had a hard time coming up with useful linting rules. The two linting rules we did implement were of a proof of concept nature, instead of being actually useful. We believe that with better domain knowledge the amount of, and usefulness of, implemented rules would be vastly improved.

Secondly, the redefinition of the syntax to make it more user-oriented was successful. We did not do any proper user testing to make sure that there were improvement. However anecdotally the syntax seems to feel easier to understand.

Finally, the fact that the DSL is JSON-based means that there are many tools available that already validates JSON documents based on standard *JSON Schema* [6]. JSON Schema only allows certain specified rules to be validated, while Sempar allows arbitrary F# code in linting rules, which means that arbitrary validation can take place. However, both of the two linting rules we used in our proof of concept can be represented in a JSON Schema if the original syntax was used. The redefined syntax would however be more complicated to validate in such a way since the syntax would have to be transformed into JSON first in order for JSON Schema to validate it.

5

Discussion

5.1 Related work

Work similar to ours has been done in the past on applying linting or static analysis to custom languages. Some of those approaches, and their main differences to our approach, will be described in this section.

5.1.1 Existing tools for static analysis

There are many different static analysis tools available, such as SonarQube[1], Codacy[2], and SQuORE[3]. They scan for possible bugs and bad practices in code bases, doing a combination of static checking and linting. These kinds of tools are commonly used by companies with large code bases. However, such tools generally only support certain languages that are commonly used in the real world. It is possible to write custom plugins to support other languages, but this takes a lot of development effort. This makes them impractical to use for custom languages.

The approach described in a paper by Ruiz-Rube et al. [7] is to transform a description of the grammar of a DSL into a plugin for SonarQube. However, SonarQube analysis passes are written as separate Java classes, which makes the code fragmented and harder to overlook. In contrast, Sempar combines the defintion of a DSL and the linting rules in the same file.

5.1.2 Runtime verification tools

A complementary approach to static analysis is runtime verification. One common way that this is done is by adding pre- and postconditions to functions or other areas in the code. Violating these linting rules is considered a serious issue, often justifying crashing the program. The approach in a paper by Ahrendt et al. [8] is to define a single formalism that encompasses both static analysis and runtime verification. In their paper, the authors define a specification notation called *ppDATE*. The specification is then processed and used in a theorem prover to partially prove some

[1] `https://www.sonarqube.org/`
[2] `https://www.codacy.com/`
[3] `http://www.squoring.com/`

static properties. The partial proofs are used together with the original specification to do runtime verification.

Compared to Sempar, Ahrendt et al. focused on formal verification of certain properties. Sempar, in contrast, does not provide any formal guarantees about the properties that one might emit diagnostics because of.

5.1.3 Language workbenches

As we have shown in our case study, one possible application of Sempar is to develop DSLs. However, Sempar is not the only tool for that. According to Iung et al. [9] there are 59 tools for DSL development with at least one associated research paper published between 2012 and 2019. Iung et al. categorises these tools in three categories, two of which are DSL construction tools and language workbenches.

Most are categorised as DSL construction tools, a tool that is helpful for developing a DSL. These tools' features vary, some have many different features which make them useful in many cases, while others are created to handle specific situations or fields and can only be applied in specific scenarios.

The other category, language workbenches, take this a step further and aim to be a full fledged IDE for developing DSLs. They have functionality such as debugging, auto completion and other features commonly seen in modern IDEs. Some workbenches are even made with graphical drag and drop interfaces. To achieve this high level functionality, some of them use DSL construction tools internally.

Sempar has more in common with DSL construction tools than language workbenches. It adds one functionality: defining linting rules. In contrast the language workbenches makes use of multiple features to improve the created DSL.

The benefit of using Sempar under the hood in a language workbench would however be marginal. One of Sempar's benefits is that the linting rules are part of the same file as the definition of the constructions and syntax. In a language workbench that aims to be a full IDE one could imagine that having the same file for everything might not be beneficial, especially in the case of workbenches with a graphical interface.

5.1.4 Attribute grammars

Attribute grammars were first described in 1968 by Knuth [10]. Attribute grammars enhance formal grammars by adding *attributes* to each node of an AST. These attributes represent some semantic information about the parsed language. The attributes in each node can either be *synthesised* or *inherited*. Synthesised attributes depend only on the attributes of the children of the node in question, while inherited attributes can also depend on the node's parents. There are several implementations of attribute grammars, one of which is JastAdd [11]. JastAdd adds some other kinds of attributes such as circular attributes and reference attributes to the original kinds described by Knuth.

Enhancing an attribute grammar parser with the ability to emit diagnostics in a similar way to what we have done with a simple parser would give the ability to create more powerful linting rules. We did not implement an attribute grammar in order to decrease the complexity of the project. However, it would probably have increased the usefulness of Sempar. In particular, one could imagine that the errors and warnings could be attributes at each node in the AST. The diagnostics being connected to a node in that way would mean that fixing them would be much easier, since one knows where the problem is. Using synthesised attributes, the errors from child nodes could be propagated upwards in the parse tree. Similarly, inherited attributes could be used to propagate information about the context around a parse node. Linting rules at parse nodes not having access to their context is a limitation of Sempar, as described in section 3.8.

5.1.5 Query/transformation languages and schemas

Query and transformation languages are DSLs that define queries into, and transformations of, structured documents. For JSON, examples include *jq*[4] and *JSONata*[5]. One of the earlier avenues of work in this field were about XML, where there are several standards such as *XSLT*[6] and *XQuery*[7]. Both of these were developed at the World Wide Web Consortium as separate, but cooperating standards.

While these languages have some support for validating the structure of documents, that is not their main purpose. Instead, a tool meant for validating the structure of documents are *schemas*. For JSON and XML, there are established standards [6], [12]. As mentioned in section 4.3, the schemas can only validate certain properties of documents. In contrast, Sempar can validate any property, as the linting rules are arbitrary F# code.

5.2 Ethics

The ethical aspects of this project are subtle. One of the main aims of this projects is to enable more correct and valid software, and hopefully to make software development faster. However, this comes with a flip side. Not all software is ethically good, there are both strictly malicious software like viruses and trojans as well as ethically dubious software, for example software for surveillance. We believe that increasing the efficiency of software development is ethically good, since we think that there is more ethically positive compared to negative software being developed.

One could also argue that those creating a tool shouldn't be held responsible for what someone else does with that tool. That is however a stance that we do not entirely agree with. Instead, we think that the person creating a tool should in some circumstances be considered responsible for what someone else does with that tool. For example, someone creating deadly poison, but not using it for something bad

[4]https://stedolan.github.io/jq/
[5]https://jsonata.org/
[6]https://www.w3.org/TR/xslt20/
[7]https://www.w3.org/XML/Query/

should be held responsible together with the murderer even if they did not actually murder someone. However, in this case, since our tool is not meant for causing harm we think that we should not be held responsible for what someone else does with our tool.

A way to limit the ethical concerns is to restrict other people's access to our tool. Theoretically, we could then restrict ethically dubious usage, while allowing ethically good usage. The easiest way to do this is to licence it or restrict availability of the tool or its source code. Having either more restrictive licensing or having it closed source would dramatically limit the ethically dubious uses of the software. However this would also limit the use of the tool for good. Therefore we argue that having the tool more open and more available is ethically better.

5.3 Conclusion

Sempar successfully gives the creator of a DSL the possibility of adding rules which, if not followed, adds diagnostics to the result of parsing. From our own experience with using it to implement both the Antura DSL as well as a simple arithmetic expression parser the ability to add linting rules on a construction helped us get a better grasp on what the semantics of the DSLs are.

As a whole, Sempar works as we expect, but the variety of rules that is definable is more limited than expected. The current implementation allows for easy creation of rules that only rely on a single node in the syntax tree and its descendants. However, coming up with rules that only rely on one node is difficult. Many rules that we would have liked to implement relies on context from other parts of the code which is not possible to express in our implementation. This is another aspect that attribute grammars might have helped with.

5.4 Future Work

The rules implemented and shown in this paper are simple in their construction and either only make use of the result of a specific construction. Future work could be how the set of possible rules could be expanded. For example, one could look at how to define rules dependent on other constructions, by building a global state accessible to all linting rules. This would however require a defined parsing order. Currently, the parsing order is entirely dependent on the parsing order of FsYacc, which is unknown. This could also perhaps be helped by the addition of a state monad so that there could be some state that could be used for the linting rules. How to pass the state so that it could be used in an intuitive way without requiring knowledge of the parsing order would be an important part of such future work.

Another approach for future work is to look at whether or not this approach is beneficial. We believe so, but as authors of Sempar we have a biased understanding and knowledge of the tool. It would be worth looking both at if the DSL developer is benefiting from this and if the linting rules helps the DSL user when using the language.

The experience of using the parser might be improved by creating a whole new language and parser to write rules and linting rules in. Our approach was to embed the linting rules in comments in FsYacc's syntax and to desugar them into code suitable for FsYacc. A custom language and parser might make it even easier and natural to write DSLs together with linting rules.

Having a custom parser could also enable even more powerful linting rules and features. For example, it could then leverage attribute grammars, as described previously. A custom parser could also allow the language developer to write linting rules based on whitespace, which is not reasonably possible right now, despite whitespace and style linting being a common use case for linters.

Bibliography

[1] S. C. Johnson, *Lint, a C Program Checker* (Computing science technical report 65). Bell Laboratories, 1977. [Online]. Available: https://books.google.se/books?id=b8nWGwAACAAJ.

[2] S. C. Johnson, 'Yacc: Yet Another Compiler-Compiler', *AT&T Bell Laboratories*, [Online]. Available: https://web.wlu.ca/science/physcomp/ikotsireas/CP465/W3-BNF-LEX-YACC/Yacc_Introduction.pdf (visited on 02/02/2023).

[3] M. E. Lesk and E. Schmidt, *Lex: A lexical analyzer generator*. Bell Laboratories Murray Hill, NJ, 1975.

[4] Microsoft. 'Computation Expressions - F#'. (29th Sep. 2022), [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/computation-expressions (visited on 14/03/2023).

[5] 'Antura Projects - A complete solution for Project, Portfolio and Resource Management - Antura', Antura.com. (5th Apr. 2023), [Online]. Available: https://www.antura.com/ (visited on 05/04/2023).

[6] F. Pezoa, J. L. Reutter, F. Suarez, M. Ugarte and D. Vrgoč, 'Foundations of JSON schema', in *Proceedings of the 25th international conference on World Wide Web*, 2016, pp. 263–273.

[7] I. Ruiz-Rube, T. Person, J. M. Dodero, J. M. Mota and J. M. Sánchez-Jara, 'Applying static code analysis for domain-specific languages', *Software and Systems Modeling*, vol. 19, no. 1, pp. 95–110, 1st Jan. 2020, ISSN: 1619-1374. DOI: 10.1007/s10270-019-00729-w. (visited on 02/02/2023).

[8] W. Ahrendt, J. M. Chimento, G. J. Pace and G. Schneider, 'A Specification Language for Static and Runtime Verification of Data and Control Properties', in *FM 2015: Formal Methods*, N. Bjørner and F. de Boer, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2015, pp. 108–125, ISBN: 978-3-319-19249-9. DOI: 10.1007/978-3-319-19249-9_8.

[9] A. Iung, J. Carbonell, L. Marchezan *et al.*, 'Systematic mapping study on domain-specific language development tools', *Empirical Software Engineering*, vol. 25, no. 5, pp. 4205–4249, 1st Sep. 2020, ISSN: 1573-7616. DOI: 10.1007/s10664-020-09872-1. (visited on 02/02/2023).

[10] D. E. Knuth, 'Semantics of context-free languages', *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, 1968, Publisher: Springer.

[11] G. Hedin and E. Magnusson, 'Jastadd—an aspect-oriented compiler construction system', *Science of Computer Programming*, vol. 47, no. 1, pp. 37–58, 2003, Special Issue on Language Descriptions, Tools and Applications (L DTA'01),

ISSN: 0167-6423. DOI: https://doi.org/10.1016/S0167-6423(02)00109-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642302001090.

[12] 'Schema - W3C'. (17th May 2023), [Online]. Available: https://www.w3.org/standards/xml/schema (visited on 17/05/2023).

A

The arithmetic DSL

This appendix contains the full contents of the files that were excerpted in chapter 3.

A.1 Language specification

This section contains the language and lexer specification that is written for the DSL.

A.1.1 Language specification (.sempar)

```
%{
open ArithAST
%}
%start expr
%token <float> NUM
%token <string> VAR
%token PLUS MINUS TIMES DIV LPAR RPAR EOF
%left PLUS MINUS TIMES DIV
%type <Arith> expr
%%
expr:
  | LPAR expr RPAR  { $2 }
  //! warnWhen "Multiplication by 0 will be simplified." ($3 = Num 0)
  //! warnWhen "Multiplication by 1 will be simplified." ($3 = Num 1)
  | expr TIMES expr { Mul ($1, $3) }
  //! warnWhen "Adding 0 will be simplified." ($3 = Num 0)
  | expr PLUS expr  { Add ($1, $3) }
  //! warnWhen "Subtracting 0 will be simplified." ($3 = Num 0)
  | expr MINUS expr { Sub ($1, $3) }
  //! errorWhen "Division by 0 will not work." ($3 = Num 0)
  //! warnWhen "Division by 1 will be simplified." ($3 = Num 1)
  | expr DIV expr   { Div ($1, $3) }
  | NUM             { Num $1 }
  | VAR             { Var $1 }
```

Listing A.1: The parser specification that Sempar parses and from which an FsYacc file will be generated.

A.1.2 Lexer specification (.fsl)

```
{
module Lexer
open FSharp.Text.Lexing
open System
open ArithParser
exception SyntaxError of string
let lexeme = LexBuffer<_>.LexemeString
let newline (lexbuf: LexBuffer<_>) =
  lexbuf.StartPos <- lexbuf.StartPos.NextLine
}
let digit = ['0'-'9']
let frac = '.' digit*
let exp = ['e' 'E'] ['-' '+']? digit+
let float = '-'? digit+ frac? exp?
let white = [' ' '\t']+
let newline = '\r' | '\n' | "\r\n"
let aao = ['å' 'ä' 'ö' 'Å' 'Ä' 'Ö']
let letter = ['a'-'z' 'A'-'Z'] | aao
let special = ['_' '-']
let var = letter (letter | special | digit)*

rule read =
  parse
  | white    { read lexbuf }
  | newline  { newline lexbuf; read lexbuf }
  | var      { VAR (lexeme lexbuf) }
  | float    { NUM (float (lexeme lexbuf)) }
  | '+'      { PLUS }
  | '-'      { MINUS }
  | '*'      { TIMES }
  | '/'      { DIV }
  | '('      { LPAR }
  | ')'      { RPAR }
  | eof      { EOF }
  | _ { raise (Exception (sprintf "SyntaxError: Unexpected char: '%s' Line: %d
↪  Column: %d" (lexeme lexbuf) (lexbuf.StartPos.Line+1) lexbuf.StartPos.Column))
↪  }
```

Listing A.2: The lexer specification for the arithmetic expressions. It is sent as is to FsLex to create a lexer.

A.1.3 Data model

```
module ArithAST

type Arith =
    | Add of (Arith * Arith)
    | Sub of (Arith * Arith)
    | Mul of (Arith * Arith)
    | Div of (Arith * Arith)
    | Num of float
    | Var of string
```

Listing A.3: The data structure that represents arithmetic expressions.

A.2 Sempar's internal language specification representation

A.2.1 Representation after parsing

```
{ preamble =
   { preaCode = PreaCode "open ArithAST "
     preaItems =
      [{ name = "start"
         value = ["expr"] };
       { name = "token"
         value = ["<float>"; "NUM"] };
       { name = "token"
         value = ["<string>"; "VAR"] };
       { name = "token"
         value = ["PLUS"; "MINUS"; "TIMES"; "DIV"; "LPAR"; "RPAR"; "EOF"] };
       { name = "left"
         value = ["PLUS"; "MINUS"; "TIMES"; "DIV"] };
       { name = "type"
         value = ["<Arith>"; "expr"] }] }
    rules =
     [{ name = "expr"
        cases =
         [{ tokens = [Token "LPAR"; Token "expr"; Token "RPAR"]
            code = Code " $2 "
            constraints = [] };
          { tokens = [Token "expr"; Token "TIMES"; Token "expr"]
            code = Code " Mul ($1, $3) "
            constraints =
             [Constr
                "warnWhen "Mutliplication by 0 will be simplified." ($3 = Num
↪  0)";
              Constr
                "warnWhen "Mutliplication by 1 will be simplified." ($3 = Num
↪  1)"] };
          { tokens = [Token "expr"; Token "PLUS"; Token "expr"]
            code = Code " Add ($1, $3) "
            constraints =
             [Constr
                "warnWhen "Adding 0 will be simplified." ($3 = Num 0)"] };
          { tokens = [Token "expr"; Token "MINUS"; Token "expr"]
            code = Code " Sub ($1, $3) "
            constraints =
             [Constr
                "warnWhen "Subtracting 0 will be simplified." ($3 = Num 0)"] };
          { tokens = [Token "expr"; Token "DIV"; Token "expr"]
            code = Code " Div ($1, $3) "
            constraints =
             [Constr
                "errorWhen "Division by 0 will not work." ($3 = Num 0)";
              Constr
```

```
45                  "warnWhen "Division by 1 will be simplified." ($3 = Num 1)"] };
46            { tokens = [Token "NUM"]
47              code = Code " Num $1 "
48              constraints = [] };
49            { tokens = [Token "VAR"]
50              code = Code " Var $1 "
51              constraints = [] }] }] }
```

Listing A.4: The internal representation that Sempar parses into before applying transformations and embedding the constraints into the generated parser.

A.2.2 Sempar's internal representation after inserting constraints

```
1  { preamble =
2      { preaCode = PreaCode "open ArithAST
3    open Diagnostics"
4        preaItems =
5         [{ name = "start"
6            value = ["expr"] };
7          { name = "token"
8            value = ["<float>"; "NUM"] };
9          { name = "token"
10           value = ["<string>"; "VAR"] };
11         { name = "token"
12           value = ["PLUS"; "MINUS"; "TIMES"; "DIV"; "LPAR"; "RPAR"; "EOF"] };
13         { name = "left"
14           value = ["PLUS"; "MINUS"; "TIMES"; "DIV"] };
15         { name = "type"
16           value = ["<Diagnostics<Arith>>"; "expr"] }] }
17     rules =
18      [{ name = "expr"
19         cases =
20          [{ tokens = [Token "LPAR"; Token "expr"; Token "RPAR"]
21             code =
22              Code "sempar {
23     let! semparVar2 = $2
24     return ( semparVar2 )
25   }"
26             constraints = [] };
27           { tokens = [Token "expr"; Token "TIMES"; Token "expr"]
28             code =
29              Code
30                "sempar {
31     let! semparVar1 = $1
32     let! semparVar3 = $3
33     do! warnWhen "Multiplication by 0 will be simplified." (semparVar3 = Num 0)
34     do! warnWhen "Multiplication by 1 will be simplified." (semparVar3 = Num 1)
35     return ( Mul (semparVar1, semparVar3) )
36   }"
37             constraints =
38              [Constr
39                 "do! warnWhen "Multiplication by 0 will be simplified."
↪  (semparVar3 = Num 0)";
```

```
40              Constr
41                "do! warnWhen "Multiplication by 1 will be simplified."
    ↪  (semparVar3 = Num 1)"] };
42            { tokens = [Token "expr"; Token "PLUS"; Token "expr"]
43              code =
44               Code
45                 "sempar {
46    let! semparVar1 = $1
47    let! semparVar3 = $3
48    do! warnWhen "Adding 0 will be simplified." (semparVar3 = Num 0)
49    return ( Add (semparVar1, semparVar3) )
50  }"
51              constraints =
52               [Constr
53                  "do! warnWhen "Adding 0 will be simplified." (semparVar3 = Num
    ↪  0)"] };
54            { tokens = [Token "expr"; Token "MINUS"; Token "expr"]
55              code =
56               Code
57                 "sempar {
58    let! semparVar1 = $1
59    let! semparVar3 = $3
60    do! warnWhen "Subtraction by 0 will be simplified." (semparVar3 = Num 0)
61    return ( Sub (semparVar1, semparVar3) )
62  }"
63              constraints =
64               [Constr
65                  "do! warnWhen "Subtraction by 0 will be simplified." (semparVar3
    ↪  = Num 0)"] };
66            { tokens = [Token "expr"; Token "DIV"; Token "expr"]
67              code =
68               Code
69                 "sempar {
70    let! semparVar1 = $1
71    let! semparVar3 = $3
72    do! errorWhen "Division by 0 will not work." (semparVar3 = Num 0)
73    do! warnWhen "Division by 1 will be simplified." (semparVar3 = Num 1)
74    return ( Div (semparVar1, semparVar3) )
75  }"
76              constraints =
77               [Constr
78                  "do! errorWhen "Division by 0 will not work." (semparVar3 = Num
    ↪  0)";
79                Constr
80                  "do! warnWhen "Division by 1 will be simplified." (semparVar3 =
    ↪  Num 1)"] };
81            { tokens = [Token "NUM"]
82              code =
83               Code
84                 "sempar {
85    let semparVar1 = $1
86
87    return ( Num semparVar1 )
88  }"
89              constraints = [] };
90            { tokens = [Token "VAR"]
```

```
            code =
              Code
                "sempar {
    let semparVar1 = $1

    return ( Var semparVar1 )
  }"
```

Listing A.5: The internal representation after transformations by Sempar in order to embed the constraints into the generated parser. Note that it is the same type as in the previous listing.

A.3 Generated language specification (.fsy)

```
%{
open ArithAST
open Diagnostics
%}
%start expr
%token <float> NUM
%token <string> VAR
%token PLUS MINUS TIMES DIV LPAR RPAR EOF
%left PLUS MINUS TIMES DIV
%type <Diagnostics<Arith>> expr
%%
expr:
| LPAR expr RPAR {sempar {
  let! semparVar2 = $2
  return ( semparVar2 )
}}
//! do! warnWhen "Multiplication by 0 will be simplified." (semparVar3 = Num 0)
//! do! warnWhen "Multiplication by 0 will be simplified." (semparVar3 = Num 1)
| expr TIMES expr {sempar {
  let! semparVar1 = $1
  let! semparVar3 = $3
  do! warnWhen "Multiplication by 0 will be simplified." (semparVar3 = Num 0)
  do! warnWhen "Multiplication by 1 will be simplified." (semparVar3 = Num 1)
  return ( Mul (semparVar1, semparVar3) )
}}
//! do! warnWhen "Adding 0 will be simplified." (semparVar3 = Num 0)
| expr PLUS expr {sempar {
  let! semparVar1 = $1
  let! semparVar3 = $3
  do! warnWhen "Adding 0 will be simplified." (semparVar3 = Num 0)
  return ( Add (semparVar1, semparVar3) )
}}
//! do! warnWhen "Subtracting 0 will be simplified." (semparVar3 = Num 0)
| expr MINUS expr {sempar {
  let! semparVar1 = $1
  let! semparVar3 = $3
  do! warnWhen "Subtracting 0 will be simplified." (semparVar3 = Num 0)
  return ( Sub (semparVar1, semparVar3) )
}}
//! do! errorWhen "Division by 0 will not work." (semparVar3 = Num 0)
```

```
41 //! do! warnWhen "Division by 1 will be simplified." (semparVar3 = Num 1)
42 | expr DIV expr {sempar {
43   let! semparVar1 = $1
44   let! semparVar3 = $3
45   do! errorWhen "Division by 0 will not work." (semparVar3 = Num 0)
46   do! warnWhen "Division by 1 will be simplified." (semparVar3 = Num 1)
47   return ( Div (semparVar1, semparVar3) )
48 }}
49 | NUM {sempar {
50   let semparVar1 = $1
51   return ( Num semparVar1 )
52 }}
53 | VAR {sempar {
54   let semparVar1 = $1
55   return ( Var semparVar1 )
56 }}
```

Listing A.6: Sempar's generated output based on the language specification in listing A.1. This is the file sent to FsYacc to generate the parser.

A.4 Input

```
1 (VAL0 + (VAL1 - 0) * (5 / 5)) / ((1 * 1) + (VAL2 * 0))
```

Listing A.7: The input code written by the user that is to be parsed.

A.5 Resulting AST

```
1 Warnings (Div (Mul (Add (Var "VAL0", Sub (Var "VAL1", Num 0.0)), Div (Num 5.0,
  ↪  Num 5.0)), Add (Mul (Num 1.0, Num 1.0), Mul (Var "VAL2", Num 0.0))),
2  ["Subtracting 0 will be simplified."; "Multiplication by 1 will be simplified.";
  ↪  "Multiplication by 0 will be simplified."])
```

Listing A.8: The internal representation of the Antura DSL as created by Sempar, based on the data model.

A.6 Transformed output

```
1 Add (Var "VAL0", Var "VAL1")
```

Listing A.9: The output from the program after transformation. In this case the arithmetic expression has been simplified.

B

Antura DSL

This appendix contains detailed context for the case study where we implemented the Antura DSL with the use of Sempar.

B.1 Language specification

This section contains the Antura DSL language specification that is written for Sempar.

B.1.1 Language specification (.sempar)

```
%{
open DataModel
open ArithLib
%}

%token <int> INT
%token <string> STRING
%token <string> ID
%token COLON EQUAL PROJECT_TYPES RULE SOURCES
%token TRANSFORMATION DESTINATION PRECISION
%token PROJECT_PROPERTY FROM_TASK_WHERE
%token FROM_DECISION_POINT FROM_NUMERIC_PROPERTY
%token TASK GET WHEN EOF

%type <DataModel.Rules> start
%start start
%%

start:
  | precision rule_list { { precision = Some $1; rules = $2 } }
  | rule_list { { precision = None; rules = $1 } }

precision:
//! errorWhen "Precision must be positive" ($2 <= 0)
  | PRECISION INT { $2 }

rule_list:
  |  { [] }
  | rule rule_list { $1 :: $2 }

rule:
```

```
32    | RULE rule_name COLON types sources transformations destination {
33  {name = Some $2; types = $4; sources = $5;
34  transformation = $6; destination = $7} }
35
36  rule_name:
37    |  { "" }
38    | ID rule_name { $1 + " " + $2 }
39
40  types:
41    | PROJECT_TYPES type_list { $2 }
42
43  type_list:
44    |  { [] }
45    | STRING type_list { $1 :: $2 }
46
47  sources:
48    | SOURCES source_list { $2 }
49
50  source_list:
51    |  { [] }
52    | source source_list { $1 :: $2 }
53
54  source:
55  //! errorUnless "Tasks must be certain values" ($6 = "StartDate" || $6 =
    ↪  "EndDate" || $6 = "Property")
56    | FROM_TASK_WHERE ID EQUAL ID GET ID { Task ($2, $4, $6) }
57    | FROM_DECISION_POINT INT GET ID { DecisionPoint ($2, $4) }
58    | FROM_NUMERIC_PROPERTY ID GET ID { NumericProjectProperty ($2, $4) }
59
60  transformations:
61    | TRANSFORMATION ID {
62        match $2 with
63        | "none" -> None
64        | s -> match (parse s) with
65               | OK t -> Some t
66               | Warnings (t, ws) -> Some t
67               | Errors es -> None
68    }
69
70  destination:
71    | DESTINATION property { $2 }
72
73  property:
74    | PROJECT_PROPERTY ID { Property $2 }
```

Listing B.1: The parser specification that Sempar parses and from which an FsYacc file will be generated.

B.1.2 Lexer specification (.fsl)

```
1  {
2
3  module Lexer
4
5  open FSharp.Text.Lexing
```

X

```
6  open System
7  open Parser
8
9  exception SyntaxError of string
10
11 let lexeme = LexBuffer<_>.LexemeString
12
13 let newline (lexbuf: LexBuffer<_>) =
14   lexbuf.StartPos <- lexbuf.StartPos.NextLine
15 }
16
17 let int = ['-' '+']? ['0'-'9']+
18 let digit = ['0'-'9']
19 //let frac = '.' digit*
20 //let exp = ['e' 'E'] ['-' '+']? digit+
21 //let float = '-'? digit* frac? exp?
22
23 let white = [' ' '\t']+
24 let newline = '\r' | '\n' | "\r\n"
25
26 let aao = ['å' 'ä' 'ö' 'Å' 'Ä' 'Ö']
27 let word = (['a'-'z' 'A'-'Z'  '_' '-'] | aao)+
28
29 rule read =
30   parse
31   | white    { read lexbuf }
32   | newline  { newline lexbuf; read lexbuf }
33   | int      { INT (int (lexeme lexbuf)) }
34   | '"'      { read_string "" false lexbuf }
35   | ':'      { COLON }
36   | '='      { EQUAL}
37   | "project types:" { PROJECT_TYPES }
38   | "rule"   { RULE }
39   | "sources:" { SOURCES }
40   | "transformation:" { TRANSFORMATION }
41   | "destination:" { DESTINATION }
42   | "precision" { PRECISION }
43   | "from task where " { FROM_TASK_WHERE }
44   | "from decision point" { FROM_DECISION_POINT }
45   | "from numeric property" { FROM_NUMERIC_PROPERTY }
46   | "project property" { PROJECT_PROPERTY }
47   | "get" { GET }
48   | word { ID(lexeme lexbuf)}
49   | eof      { EOF }
50   | _ { raise (Exception
51     (sprintf "SyntaxError: Unexpected char: '%s' Line: %d Column: %d"
52     (lexeme lexbuf) (lexbuf.StartPos.Line+1) lexbuf.StartPos.Column)) }
53
54
55 and read_string str ignorequote =
56   parse
57   | '"'           { if ignorequote
58                     then (read_string (str+"\\\"") false lexbuf)
59                     else STRING (str) }
60   | '\\'          { read_string str true lexbuf }
61   | ([^ '"' '\\'] | aao)+ { read_string (str+(lexeme lexbuf)) false lexbuf }
```

Listing B.2: The lexer specification for FsLex. It isn't modified by Sempar and is sent as is to FsLex to create a lexer.

B.1.3 Data model

```
open ArithAST

type Type = string
type Transformation = Arith
type Destination = Property of string
type Source =
  Task  of (string * string * string)
  | DecisionPoint of (int * string)
  | NumericProjectProperty of (string * string)
type Rule = {
    name: string option;
    types: Type list;
    sources: Source list;
    transformation: Transformation option;
    destination: Destination; }
type Rules = { precision: int option; rules: Rule list; }
```

Listing B.3: The data structure that represents the Antura DSL.

B.2 Sempar's internal language specification representation

B.2.1 Representation after parsing

```
{ preamble =
     { preaCode = PreaCode "open ArithAST "
       preaItems =
        [{ name = "start"
           value = ["expr"] }; { name = "token"
                                 value = ["<float>"; "NUM"] };
         { name = "token"
           value = ["<string>"; "VAR"] };
         { name = "token"
           value = ["PLUS"; "MINUS"; "TIMES"; "DIV"; "LPAR"; "RPAR"; "EOF"] };
         { name = "left"
           value = ["PLUS"; "MINUS"; "TIMES"; "DIV"] };
         { name = "type"
           value = ["<Arith>"; "expr"] }] }
    rules =
     [{ name = "expr"
        cases =
         [{ tokens = [Token "LPAR"; Token "expr"; Token "RPAR"]
            code = Code " $2 "
            constraints = [] };
          { tokens = [Token "expr"; Token "TIMES"; Token "expr"]
```

```
22              code = Code " Mul ($1, $3) "
23              constraints =
24               [Constr
25                  "warnWhen "Are you sure that you want to multiply by 0? It will
    ↪  be simplified." ($3 = Num 0)";
26                Constr
27                  "warnWhen "Are you sure that you want to multiply by 1? It will
    ↪  be simplified." ($3 = Num 1)"] };
28            { tokens = [Token "expr"; Token "DIV"; Token "expr"]
29              code = Code " Div ($1, $3) "
30              constraints =
31               [Constr
32                  "errorWhen "Are you sure that you want to divide by 0? It will
    ↪  not work." ($3 = Num 0)";
33                Constr
34                  "warnWhen "Are you sure that you want to divide by 1? It will be
    ↪  simplified." ($3 = Num 1)"] };
35            { tokens = [Token "expr"; Token "PLUS"; Token "expr"]
36              code = Code " Add ($1, $3) "
37              constraints =
38               [Constr
39                  "warnWhen "Are you sure that you want to add 0? It will be
    ↪  simplified." ($3 = Num 0)"] };
40            { tokens = [Token "expr"; Token "MINUS"; Token "expr"]
41              code = Code " Sub ($1, $3) "
42              constraints =
43               [Constr
44                  "warnWhen "Are you sure that you want to subtract 0? It will be
    ↪  simplified." ($3 = Num 0)"] };
45            { tokens = [Token "NUM"]
46              code = Code " Num $1 "
47              constraints = [] }; { tokens = [Token "VAR"]
48                                    code = Code " Var $1 "
49                                    constraints = [] }] }] }
```

Listing B.4: The internal representation that Sempar parses into before applying transformations and embedding the constraints into the generated parser.

B.2.2 Sempar's internal representation after inserting constraints

```
1  { preamble =
2      { preaCode = PreaCode "open DataModel
3    open ArithLib
4    open Diagnostics"
5         preaItems =
6          [{ name = "start"
7             value = ["start"] }; { name = "token"
8                                    value = ["<int>"; "INT"] };
9           { name = "token"
10            value = ["<string>"; "STRING"] }; { name = "token"
11                                                value = ["<string>"; "ID"] };
12          { name = "token"
13            value = ["COLON"] }; { name = "token"
```

```
                                    value = ["EQUAL"] };
          { name = "token"
            value = ["PROJECT_TYPES"] }; { name = "token"
                                           value = ["RULE"] };
          { name = "token"
            value = ["SOURCES"] }; { name = "token"
                                     value = ["TRANSFORMATION"] };
          { name = "token"
            value = ["DESTINATION"] }; { name = "token"
                                         value = ["PRECISION"] };
          { name = "token"
            value = ["PROJECT_PROPERTY"] }; { name = "token"
                                              value = ["FROM_TASK_WHERE"] };
          { name = "token"
            value = ["FROM_DECISION_POINT"] };
          { name = "token"
            value = ["FROM_NUMERIC_PROPERTY"] }; { name = "token"
                                                   value = ["TASK"] };
          { name = "token"
            value = ["GET"] }; { name = "token"
                                 value = ["WHEN"] }; { name = "token"
                                                       value = ["EOF"] };
          { name = "type"
            value = ["<Diagnostics<DataModel.Rules>>"; "start"] }] }
      rules =
       [{ name = "start"
          cases =
           [{ tokens = [Token "precision"; Token "rule_list"]
              code =
               Code
                 "sempar {
      let! semparVar1 = $1
      let! semparVar2 = $2

      return ( { precision = Some semparVar1; rules = semparVar2 } )
    }"
              constraints = [] };
            { tokens = [Token "rule_list"]
              code =
               Code
                 "sempar {
      let! semparVar1 = $1

      return ( { precision = None; rules = semparVar1 } )
    }"
              constraints = [] }] };
        { name = "precision"
          cases =
           [{ tokens = [Token "PRECISION"; Token "INT"]
              code =
               Code
                 "sempar {
      let semparVar2 = $2
      do! errorWhen "Precision must be positive" (semparVar2 <= 0)
      return ( semparVar2 )
    }"
```

```
70              constraints =
71               [Constr
72                  "do! errorWhen "Precision must be positive" (semparVar2 <= 0)"]
    ↪  }] };
73        { name = "rule_list"
74          cases =
75           [{ tokens = []
76              code = Code "sempar {
77
78
79      return ( [] )
80    }"
81              constraints = [] };
82           { tokens = [Token "rule"; Token "rule_list"]
83              code =
84               Code
85                 "sempar {
86      let! semparVar1 = $1
87      let! semparVar2 = $2
88
89      return ( semparVar1 :: semparVar2 )
90    }"
91              constraints = [] }] };
92        { name = "rule"
93          cases =
94           [{ tokens =
95                [Token "RULE"; Token "rule_name"; Token "COLON"; Token "types";
96                 Token "sources"; Token "transformations"; Token "destination"]
97              code =
98               Code
99                 "sempar {
100     let! semparVar2 = $2
101     let! semparVar4 = $4
102     let! semparVar5 = $5
103     let! semparVar6 = $6
104     let! semparVar7 = $7
105
106     return ( {name = Some semparVar2; types = semparVar4; sources = semparVar5;
    ↪  transformation = semparVar6; destination = semparVar7} )
107   }"
108             constraints = [] }] };
109       { name = "rule_name"
110         cases =
111          [{ tokens = []
112             code = Code "sempar {
113
114
115     return ( "" )
116   }"
117             constraints = [] };
118          { tokens = [Token "ID"; Token "rule_name"]
119             code =
120              Code
121                "sempar {
122     let semparVar1 = $1
123     let! semparVar2 = $2
```

```
124
125     return ( semparVar1 + " " + semparVar2 )
126   }"
127             constraints = [] }] };
128       { name = "types"
129         cases =
130          [{ tokens = [Token "PROJECT_TYPES"; Token "type_list"]
131             code =
132              Code "sempar {
133     let! semparVar2 = $2
134
135     return ( semparVar2 )
136   }"
137             constraints = [] }] };
138       { name = "type_list"
139         cases =
140          [{ tokens = []
141             code = Code "sempar {
142
143
144     return ( [] )
145   }"
146             constraints = [] };
147           { tokens = [Token "STRING"; Token "type_list"]
148             code =
149              Code
150                "sempar {
151     let semparVar1 = $1
152     let! semparVar2 = $2
153
154     return ( semparVar1 :: semparVar2 )
155   }"
156             constraints = [] }] };
157       { name = "sources"
158         cases =
159          [{ tokens = [Token "SOURCES"; Token "source_list"]
160             code =
161              Code "sempar {
162     let! semparVar2 = $2
163
164     return ( semparVar2 )
165   }"
166             constraints = [] }] };
167       { name = "source_list"
168         cases =
169          [{ tokens = []
170             code = Code "sempar {
171
172
173     return ( [] )
174   }"
175             constraints = [] };
176           { tokens = [Token "source"; Token "source_list"]
177             code =
178              Code
179                "sempar {
```

```
180       let! semparVar1 = $1
181       let! semparVar2 = $2
182
183       return ( semparVar1 :: semparVar2 )
184     }"
185               constraints = [] }] };
186         { name = "source"
187           cases =
188            [{ tokens =
189                [Token "FROM_TASK_WHERE"; Token "ID"; Token "EQUAL"; Token "ID";
190                 Token "GET"; Token "ID"]
191               code =
192                Code
193                  "sempar {
194       let semparVar2 = $2
195       let semparVar4 = $4
196       let semparVar6 = $6
197       do! errorUnless "Tasks must be certain values" (semparVar6 = "StartDate" ||
    ↪  semparVar6 = "EndDate" || semparVar6 = "Property")
198       return ( Task (semparVar2, semparVar4, semparVar6) )
199     }"
200               constraints =
201                [Constr
202                   "do! errorUnless "Tasks must be certain values" (semparVar6 =
    ↪  "StartDate" || semparVar6 = "EndDate" || semparVar6 = "Property")"] };
203             { tokens =
204                [Token "FROM_DECISION_POINT"; Token "INT"; Token "GET"; Token "ID"]
205               code =
206                Code
207                  "sempar {
208       let semparVar2 = $2
209       let semparVar4 = $4
210
211       return ( DecisionPoint (semparVar2, semparVar4) )
212     }"
213               constraints = [] };
214             { tokens =
215                [Token "FROM_NUMERIC_PROPERTY"; Token "ID"; Token "GET"; Token "ID"]
216               code =
217                Code
218                  "sempar {
219       let semparVar2 = $2
220       let semparVar4 = $4
221
222       return ( NumericProjectProperty (semparVar2, semparVar4) )
223     }"
224               constraints = [] }] };
225         { name = "transformations"
226           cases =
227            [{ tokens = [Token "TRANSFORMATION"; Token "ID"]
228               code =
229                Code
230                  "sempar {
231       let semparVar2 = $2
232
233       return (
```

```
234         match semparVar2 with
235         | "none" -> None
236         | s -> match (parse s) with
237                | OK t -> Some t
238                | Warnings (t, ws) -> Some t
239                | Errors es -> None
240     )
241   }"
242             constraints = [] }] };
243       { name = "destination"
244         cases =
245          [{ tokens = [Token "DESTINATION"; Token "property"]
246             code =
247              Code "sempar {
248     let! semparVar2 = $2
249
250     return ( semparVar2 )
251   }"
252             constraints = [] }] };
253       { name = "property"
254         cases =
255          [{ tokens = [Token "PROJECT_PROPERTY"; Token "ID"]
256             code =
257              Code
258                "sempar {
259     let semparVar2 = $2
260
261     return ( Property semparVar2 )
262   }"
```

Listing B.5: The internal representation after transformations by Sempar in order to embed the constraints into the generated parser. Note that it is the same type as in the previous listing.

B.3 Generated language specification (.fsy)

```
1  %{
2  open DataModel
3  open ArithLib
4  open Diagnostics
5  %}
6
7  %start start
8  %token <int> INT
9  %token <string> STRING
10 %token <string> ID
11 %token COLON
12 %token EQUAL
13 %token PROJECT_TYPES
14 %token RULE
15 %token SOURCES
16 %token TRANSFORMATION
17 %token DESTINATION
18 %token PRECISION
```

```
%token PROJECT_PROPERTY
%token FROM_TASK_WHERE
%token FROM_DECISION_POINT
%token FROM_NUMERIC_PROPERTY
%token TASK
%token GET
%token WHEN
%token EOF
%type <Diagnostics<DataModel.Rules>> start

%%

start:
| precision rule_list {sempar {
  let! semparVar1 = $1
  let! semparVar2 = $2

  return ( { precision = Some semparVar1; rules = semparVar2 } )
}}
| rule_list {sempar {
  let! semparVar1 = $1

  return ( { precision = None; rules = semparVar1 } )
}}

precision:
//! do! errorWhen "Precision must be positive" (semparVar2 <= 0)
| PRECISION INT {sempar {
  let semparVar2 = $2
  do! errorWhen "Precision must be positive" (semparVar2 <= 0)
  return ( semparVar2 )
}}

rule_list:
|  {sempar {


  return ( [] )
}}
| rule rule_list {sempar {
  let! semparVar1 = $1
  let! semparVar2 = $2

  return ( semparVar1 :: semparVar2 )
}}

rule:
| RULE rule_name COLON types sources transformations destination {sempar {
  let! semparVar2 = $2
  let! semparVar4 = $4
  let! semparVar5 = $5
  let! semparVar6 = $6
  let! semparVar7 = $7

  return ( {name = Some semparVar2; types = semparVar4;
  sources = semparVar5; transformation = semparVar6;
```

```
75    destination = semparVar7} )
76  }}
77
78  rule_name:
79  |  {sempar {
80
81
82    return ( "" )
83  }}
84  | ID rule_name {sempar {
85    let semparVar1 = $1
86    let! semparVar2 = $2
87
88    return ( semparVar1 + " " + semparVar2 )
89  }}
90
91  types:
92  | PROJECT_TYPES type_list {sempar {
93    let! semparVar2 = $2
94
95    return ( semparVar2 )
96  }}
97
98  type_list:
99  |  {sempar {
100
101
102   return ( [] )
103 }}
104 | STRING type_list {sempar {
105   let semparVar1 = $1
106   let! semparVar2 = $2
107
108   return ( semparVar1 :: semparVar2 )
109 }}
110
111 sources:
112 | SOURCES source_list {sempar {
113   let! semparVar2 = $2
114
115   return ( semparVar2 )
116 }}
117
118 source_list:
119 |  {sempar {
120
121
122   return ( [] )
123 }}
124 | source source_list {sempar {
125   let! semparVar1 = $1
126   let! semparVar2 = $2
127
128   return ( semparVar1 :: semparVar2 )
129 }}
130
```

```
source:
//! do! errorUnless "Tasks must be certain values" (semparVar6 = "StartDate" ...
| FROM_TASK_WHERE ID EQUAL ID GET ID {sempar {
  let semparVar2 = $2
  let semparVar4 = $4
  let semparVar6 = $6
  do! errorUnless "Tasks must be certain values"
    (semparVar6 = "StartDate"
    || semparVar6 = "EndDate"
    || semparVar6 = "Property")
  return ( Task (semparVar2, semparVar4, semparVar6) )
}}
| FROM_DECISION_POINT INT GET ID {sempar {
  let semparVar2 = $2
  let semparVar4 = $4

  return ( DecisionPoint (semparVar2, semparVar4) )
}}
| FROM_NUMERIC_PROPERTY ID GET ID {sempar {
  let semparVar2 = $2
  let semparVar4 = $4

  return ( NumericProjectProperty (semparVar2, semparVar4) )
}}

transformations:
| TRANSFORMATION ID {sempar {
  let semparVar2 = $2

  return (
      match semparVar2 with
      | "none" -> None
      | s -> match (parse s) with
             | OK t -> Some t
             | Warnings (t, ws) -> Some t
             | Errors es -> None
  )
}}

destination:
| DESTINATION property {sempar {
  let! semparVar2 = $2

  return ( semparVar2 )
}}

property:
| PROJECT_PROPERTY ID {sempar {
  let semparVar2 = $2

  return ( Property semparVar2 )
}}

```

Listing B.6: Sempar's generated output based on the language specification in listing B.1. This is the file sent to FsYacc to generate the parser.

B.4 Input

```
precision 2
rule Sätt automatiskt produktionsstart:
  project types:
    "Små projekt"
    "Standard projekt"
  sources:
    from task where Referens = genomför get StartDate
  transformation:
    none
  destination:
    project property Produktionsstart
```

Listing B.7: The input code written by the user that is to be parsed. This example is based on an actual example in use by Antura. For the same representation in JSON, see the transformed output in listing B.9.

B.5 Resulting AST

```
{
  precision = Some 2
  rules = [
    {
      name = Some "Sätt automatiskt produktionsstart "
      types = ["1 Små projekt"; "2 Standard projekt"]
      sources = [Task ("Referens", "genomför", "StartDate")]
      transformation = None
      destination = Property "Produktionsstart"
    }
  ]
}
```

Listing B.8: The internal representation of the Antura DSL as created by Sempar, based on the data model.

B.6 Transformed output

```
{
  "decimalPrecision": 2,
  "rules": [
    {
      "_name": "Sätt automatiskt produktionsstart ",
      "destination": {
        "kind": "ProjectProperty",
        "property": "Produktionsstart"
      },
      "transformation": {
        "kind": "NoTransformation"
      },
      "validFor": {
```

```
14          "kind": "MultipleProjectTypes",
15          "multipleProjectTypes": [
16            "1 Små projekt",
17            "2 Standard projekt"
18          ]
19        },
20        "valueSource": [
21          {
22            "findTaskBy": {
23              "kind": "PropertyNameValue",
24              "taskPropertyName": "Referens",
25              "taskPropertyValue": "genomför"
26            },
27            "kind": "Task",
28            "taskSourceValue": "StartDate"
29          }
30        ]
31      }
32    ],
33    "schemaVersion": 1
34  }
```

Listing B.9: The output from the program after transformation. In this case the AST has been transformed into the JSON format that is currently in use by Antura.