MDSD Take-Home Exam 2023

Miguel Campusano
MMMI, Software Engineering Section

1 Introduction

The MDSD 2023 take-home exam concerns implementing the UI23 language in Xtext. This document describes the language informally using EBNF and examples. The attached archive contains the source of 8 UI23 programs, the corresponding generated Java code, and test programs for the generated Java code.

Your goal is to implement a compiler for UI23 using Xtext. The code generated by your compiler does not need to be identical to the provided generated code but should be similar (as described later) and should work with the test scenarios. The provided example programs exercise all parts of the language. Your implementation is expected to work for the supplied UI23 programs and similar programs. For a complete answer, your compiler must support validation and be able to guarantee that the Java-generated code does not have errors. Note that the exam does not emphasize other interactive features, such as content assist. Scoping rules are very relevant, and validation requires implementing type validation (among others). Still, the exam can be completed with a medium grade without any scoping or validation, as described later. The rest of this document is organized as follows. First, Section 2 introduces the UI23 language, after which Section 3 gives hints on how to implement your solution. Then, Section 4 describes the formal requirements for providing a solution, and last Section 5 describes how grading will be performed.

2 The UI23 Language

The UI23 language is designed to create a simple user interface form using Java. This type of interface accepts inputs from users, validates those inputs, and allows for input manipulation (such as saving).

The basic element of this language is a *form*. In UI23, forms have 3 different elements: labels, input texts and, buttons. These elements can be displayed horizontally or vertically using the keywords *row* and *column*. The different elements have different structures, depending on the type of element. There are conditionals over values that are used to validate if the input value should be saved or rejected. Moreover, a form can be composed using other forms.

2.1 Basic form

A basic program starts with a *title* and then a form definition. A form consists of a layout definition and at least one element inside. A form contains exactly one layout definition. However, inside a layout definition, there can be several elements and even more layout definitions. The following Form creates a basic window with a single label that says *My first form*.

```
title FirstForm
form Display{
```

```
    column {
        label : "My first form"
    }
}
```

2.2 Layout

Layouts are defined by the keywords *row* and *column*. If using column, the listed elements will be displayed one below to the other.

```
title LayoutExample
form Display{
    column {
        label : "Above"
        label : "Below"
    }
}
```

If using row, the listed elements will be displayed one next to the other.

```
title LayoutExample
form Display{
    row {
        label : "Left"
        label : "Right"
    }
}
```

Moreover, layouts can be nested, creating more elaborated user interfaces:

```
title NestedLayout
form Display{
    column {
        row {
            label : "Up Left"
            label : "Up Right"
        }
        row {
            label : "Down Left"
            label : "Down Right"
        }
    }
}
```

2.3 Elements

Developers can define several elements: labels, input texts, and buttons. The following code shows all possible elements in the user interface.

```
title PersonalInformation
form Name{
    column {
        row {
            label : "What is your name?"
            input name : string
        }
```

```
        button : "Save"
    }
}
```

2.4 Validation

Input elements define validation over their inputs. Inputs can be validated in terms of types: number and string. Moreover, validations allow any type of expression, including binary operations for arithmetic, conditional expressions such as `&&` *and*, `||` *or*, `<=` *less than equal*, `!` *negation*, etc. We provide the full extent of the operations in the EBNF in Section 2.7.

The following program asks for a user's name, age, and password While the input name could be any string, the age input can only be a positive number. Moreover, the length of the provided password should be over 8. Finally, the UI specifies a field where the user should repeat the password, and the programs validate that both password and repeatPassword are the same string.

```
title Validation
form Registration{
    column {
        row {
            label : "Name"
            input name : string
        }
        row {
            label : "Age"
            input age : number > 0
        }
        row {
            label : "Password"
            input password: #string > 8
        }
        row {
            label : "Repeat Password"
            input repeatPassword: password == string
        }
        button* : "Save"
    }
}
```

When a user clicks a button marked with a star (*), the program should validate the inputs. If every input has the proper value, then a dialog is open displaying the message *Data has been saved.* If at least one of the inputs is incorrect, then a dialog displays the message *Validation Error.*

For referring to the same input, we use the type keywords string and number. For referring to other elements of the form, we use the name of the input in the expression. This can be seen in the previous example.

Notice that a string over an input always validates to true, due to everything written in the input, numbers or text, can be parsed to a string.

2.5 Extending Forms

Forms can also be composed using other defined forms by defining multiple forms in UI23 programs. Then, a layout in a form can define another layout, a form

element, or another form. The programs are always compiled from top to bottom. This means that the root form is the first defined form. The remaining forms in the program are not displayed in the UI, unless they are part of another form.

```
title MultipleForms
form Student{
    column {
        label: "Register Student"
        PersonalInformation()
    }
}

form PersonalInformation{
    row {
        input name : text
        input age : number
    }
}
```

Using multiple forms also allow for reusing predefined structures. For example, the following program asks information for a professor.

```
title MultipleForms2
form RegisterProfessor{
    column {
        label : "Registration"
        PersonalInformation("Professor")
    }
}

form PersonalInformation2(name: string){
    column{
        label : "Fill with " & name & " Information"
        row {
            input name : string
            input age : number
        }
    }
}
```

Then, by changing the argument of the previous form, the following program asks for the information for a student.

```
title MultipleForms3

form RegisterStudent{
    column {
        label : "Registration"
        PersonalInformation("Student")
    }
}

form PersonalInformation2(name: string){
    column{
        label : "Fill with " & name & " Information"
        row {
```

```
            input name : string
            input age : number
        }
    }
}
```

As you can see in the last example, inputs that receive a text (such as labels and buttons) actually accept an expression that returns a string. In the previous example, the label is defined using the concatenation operation &.

2.6 External Functions

The program also allows the definition of external functions:

```
title External

function validMail(string) : boolean
form Information {
    row {
        input name : string
        input mail : validMail(string)
    }
}
```

External functions are defined at the beginning of the program using the `function(type, type, type, ...) : returnType` expression. Developers can define as many external functions as they want, with any number of parameters and only one return value. Types are defined for the parameters and the return value.

2.7 Syntax

Figure 1 shows the EBNF of the UI23 syntax. A program defines its name, external functions, and forms. External functions are defined with type on their parameters and a return type. Forms can have a constructor, and they start with one Layout. A layout starts with a keyword and contains a body with several components: another layout, form usage, and basic elements.

Expressions include basic boolean, arithmetic, and string expressions. Expressions can also use the keywords for Type (`string`, `number`, `boolean`), as shown when defining an input validation from an input. In addition, expressions support basic values, such as numbers `NUMBER` and strings defined with quotes `STRING`. Finally, expressions can call external functions or reference defined variables.

The EBNF does not show precedence or associativity for expressions. Moreover, all binary operators are required to be left-associative. The precedence for expressions, from lower to greater, is:

1. And, Or boolean operators: `&&`, `||`
2. Equality comparison: `==`, `!=`
3. Less/greater comparison: `<`, `<=`, `>`, `>=`
4. Concatenation: `&`
5. Addition and subtraction
6. Multiplication and division
7. Primitives (same priority):

```
UI23: 'title' ID Function* Form*
Function: 'function' ID '(' (Type (',' Type)*)? ')' ':' Type
Form: 'form' ID ('('(Parameter (',' Parameter)*)?')')?
          '{' Layout '}'
Layout: ('row' | 'column') '{' Component*'}'
Component: Layout | Element | FormUse
Element: Label | InputText | Button
Label: 'label' ':' Exp
InputText: 'input' ID ':' Exp
Button: 'button' '*'? ID ':' Exp
FormUse: ID '('(Exp (',' Exp)*)?')'
Parameter: ID ':' Type
Type: 'boolean' | 'string' | 'number'
Exp: LogicExp | MathExp | TextExp
    | '(' Exp ')'
    | Type
    | STRING
    | NUMBER
    | ID '('(Exp (',' Exp)*)?')'
    | ID
LogicExp: Exp ('==' | '!=' | '<' | '>' | '<=' | '>=') Exp
        | Exp ('&&' | '||') Exp
        | '!' Exp
MathExp: Exp ('+' | '-' | '*' | '/') Exp
TextExp: Exp '&' Exp | '#' Exp
```

Figure 1: EBNF of UI23

- Constant: Number, String
- Not boolean expression: `!` (negate the value of the expression `!true == false`)
- String length operator: `#` (returns the length of the string `#"hello world" == 11`)
- Keyword: type keywords (`number`, `string`)
- References: Variable reference and function calls
- Parenthesis

2.8 Scope

It is not mandatory to apply Xtext scope mechanism, and some scoping rules can be implemented using validation rules. However, we encourage you to use Xtext scope when needed, as it will help in the implementation of your solution.

2.9 Validation

Validation should check for a range of different properties. From the following list, you should pick any 2 of them:

- Duplicated names of elements. For example, if you have the same name for an input and a button, then the program would not be able to know which one the developer is referring to.
- Only use *one* type keyword in a question input validation expression. It is illegal to use `number > 0 && text == "yes"` as input validation.
- The function and form call arguments should match their parameters definition. The same amount of arguments and correspondent type.
- Type of expressions
  - Input validation expression should return a boolean, and expression of labels and buttons should return a string
  - The use of a variable in an expression should be consistent. `aVar > 0` only makes sense if the declaration of the variable is `var aVar : number`
  - General expression rules. For example: equality checks expressions of the same type, and/or expressions work over booleans, mathematical operations work over numbers, concatenation works when at least one of the expressions is a text (`"Hi "&1`, and `100&" apples"` are legal, but `42&24` is illegal)

3 Implementation Hints

3.1 Material provided

The zip file included with this document contains the response template (see Section 4) and the directory of the runtime eclipse instance. Inside this directory are 8 UI23 programs, the corresponding generated programs, and testing programs that your generated programs should execute to display the user interfaces. The user interfaces should behave accordingly to the rules expressed in this document. See the `README.txt` file for specific details.

3.2 Concrete example

Provided generated programs are *examples* of how code can be generated, you are not required to precisely match this code, as described in more detail in Section 4.

From the files, you can see that each form generates its own class, which are subclasses of a common `Form` class. Moreover, an extra class called `UserInterface` is generated per each UI23 file, which instantiates the user interface.

Every UI23 file generates a Java package containing all the classes. This package is of the form `user_interface.<title_of_program>`. Moreover, an extra `user_interface.common` package is created for the `Form` class.

For UI23 programs with external function definitions, an extra `External` interface is generated, with the definition of every function method. Remember, when using external functions, it is the developer's job to connect this interface with an actual implementation. For testing purposes, we provided the implementation of the interfaces for programs P7 and P8.

3.3 Relevant material

Completely solving this exam requires working with validation in general and type validation in particular, and some of the analysis needed can be done as scoping rules. Bettini's book has excellent coverage of these topics as part of the course curriculum. Since your time is short, here are a few hints.

First, type validation is introduced in Chapter 8 "Typing expressions" with additional details in Chapter 9 "Type checking". In general, Chapter 9 on SmallJava contains many concepts that you may find useful in solving the exam.

Second, the scoping rule for a local variable and a parameter is more complex than a simple cross-reference. Luckily, we covered this during assignment 2 with local and global variables (let and var binding). Nevertheless, if you forgot how to do this, the standard approach of a variable simply being an Xtext cross-reference to an element of the grammar does not work. In SmallJava, the solution is to introduce a new rule that only is used for cross-referencing, not for the grammar (page 210 of Bettini):

```
SJSymbol: SJVariableDeclaration | SJParameter ;
```

Variables can now be described as a cross-reference:

```
{SJSymbolRef} symbol=[SJSymbol]
```

4 Solution

4.1 General principles

You will be evaluated on your ability to use Xtext to implement a UI23 compiler performing similarly to the system described in this document and the included source code. Ideally, you will implement a compiler that can:

1. Parse the UI23 source code provided.
2. Generate Java code that allows a person to run and use the user interface.
3. Compile programs similar to those provided (i.e., adding more declarations to one of the provided UI23 programs does not break your compiler).
4. Use validation to check if UI23 programs are legal before attempting to generate code, as presented in Section 2.8 and Section 2.9

This means that: (i) If you need to make changes to the UI23 syntax to make your compiler work, for example, by using syntactic modifiers to indicate if variables are parameters or local variables, or require the use of parentheses because you did not implement operator precedence, it is still a solution (just not as good as if you did not need to make changes). Moreover, (ii) there is no requirement that the code you generate exactly matches the provided code, just that it is similar in functionality. In addition, (iii) hard-coded solutions tailored to the precise programs that are provided are not considered acceptable (and you will be failed most likely). Last, (iv) validation should catch errors as outlined in Section 2.9 inside the eclipse DSL environment, such that the user does not have to resolve typing errors or illegal identifiers in the generated Java code. This assignment thus includes Xtext-style validation but does not otherwise include anything regarding interactive usage, i.e., content assist.

Note that it is possible to generate code that partially satisfies the requirements without using scoping or validation. This can be done by treating all identifiers as simple `ID` types and not implementing any validation. In this case, errors in the UI23 program will manifest in the Java program but provided correct UI23 programs generate valid Java code, and this is still a meaningful solution. An answer without scoping or validation will, of course, not be given full credit but will earn you partial credit (see next section for details on grading). For this reason, you are **strongly encouraged to work iteratively**, where you first produce something that works and then subsequently extends this to a complete functionality (this is why we provided several programs, of which P1 is the simplest one, and P8 is the most difficult one). *Note, however, that if you are unable to implement a UI23 compiler that can generate legal Java code (code that can compile using a standard Java compiler), you are in danger of not passing the course!*

4.2 Specific requirements

You are required to hand in two different files: a pdf file with a short written description of how you have solved the different parts of the exam and a zip archive containing your solution.

The pdf file must follow the template in Figure 2. You are expected to provide short and concise answers to the questions. The response to questions 1–5 is not expected to take up more than 3 pages of text (this is not a hard requirement, but if you are writing 6 pages, you are not being concise, if you only have 1 page, then there are not enough details). The response to question 6 is your Xtext grammar file and all of your (manually written) Xtend files, which must be included at the end as part of the pdf (which will then be rather long, but that is the intention).

The zip archive file must contain files precisely matching the following structure (failure to follow this structure will result in a lower grade):

```
<SDU username>/ui23cc.jar
<SDU username>/ExamEclipse/...
<SDU username>/ExamRuntime/...
```

Here, `<SDU username>` is the part of your SDU student email address that comes before `@`, so your zip file contains a single directory holding all of the contents. The file `ui23cc.jar` must be a standalone version of your compiler generated using the approach described by Bettini in Chapter 5, "Standalone command-line compiler" (pages 94–97). The directory `ExamEclipse/` must contain your complete Xtext project, everything included (i.e., the eclipse project containing your Xtext and Xtend files). The directory `ExamRuntime/` must contain your version of the runtime directory used by the runtime eclipse instance (primarily UI23 source files that work with your compiler and the corresponding generated Java files).

1. Name, email
2. Xtext Grammar:
   (a) Does your grammar support all of the example programs? If not, what are the limitations?
   (b) How did you implement operator precedence and associativity?
   (c) How did you implement the syntax of variables (`ID` in rule `Exp` of the UI23 BNF) such that they can refer both to parameters of forms and inputs?
3. Scoping Rules
   (a) Describe your implementation of any scoping rules included with your system.
4. Validation
   (a) What validation rule did you implement, if any?
   (b) Describe the implementation of any validation rule.
   (c) For the non-implemented validation rules, describe the problems of not having them.
5. Generator
   (a) Does your code generator correctly generate code for all of the examples provided, and are you confident that it will also work for "similar" programs? If not, then what limitations are there?
   (b) Briefly describe how your code generator works.
6. Implementation: include your Xtext grammar file, and all implemented Xtend files (scoping, type validation, generator, and any additional file).

Figure 2: Outline of the response template

5 Grading

Maximal grade is awarded for a response that:

- Generates legal (i.e., can be compiled using the Java compiler) and readable code for all provided examples, using scoping rules and validation.
- Is implemented in an understandable way and appropriately uses Xtext/Xtend features (such as cross-references in the grammar file and type-switch or dispatch methods in the Xtend files). This also implies that using scoping rules to find missing identifiers is preferred to validation rules since this is the "Xtext way" of solving the problem.
- Is robust in the sense that the compiler always generates correct code for legal UI23 programs, and illegal UI23 programs do not generate Java programs.
- Provides clear and concise answers to questions 1–5 in the response template.
- Follows the guidelines outlined in the previous section.

The minimally acceptable response is one that:

- Generates legal (i.e., can be compiled using the Java compiler) code for a minimal set of slightly modified UI23 examples included in the response.
- Is implemented in Xtext.
- Provides answers to questions 1–5 in the response template.
- Manages to include enough material that it is possible to evaluate the answer.

Anything in between is assessed in terms of how well you have managed to solve the problem. A compiler that correctly generates code for all of the example programs but does not include any scoping rules or validation will be awarded a grade of "7". Remember to work in an iterative way using the UI23 example programs as a base. Build your grammar and generator to work first with P1, then move on to P2, and so on.

The assessment includes running your compiler (using the provided jar file), inspecting your code (Xtext and Xtend) and the generated code, and may also include interactively inspecting your project from within Eclipse.

Last, keep in mind that this is an *individual* exam. Therefore, you are not allowed to interact with other students regarding this exam; this will be considered cheating. The sanctions put in place by SDU against cheating in online exams such as this one are quite severe. You are, however, allowed to contact Miguel and/or Niels and use all materials (i.e., information on itslearning, the curriculum, and material found on the Internet). Please see itslearning for additional details on how to contact Miguel and/or Niels during the exam.