MDSDP Take-Home Exam 2021

Ulrik Pagh Schultz, MMMI

June 9, 2021

1 Introduction

The MDSDP 2021 take-home exam concerns the implementation of the *X21* language in xtext. This document describes the language informally using BNF and examples. The attached archive contains the source of six X21 programs, the corresponding generated Java code, and test programs for the generated Java code. Two different variants of generated Java code are provided: the basic version without type inference, and the complete version with type inference.

Your goal is to implement a compiler for X21 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 pass the test cases. The example programs provided exercise all parts of the language, your implementation is expected to work for the supplied X21 programs and for similar programs. For a complete answer your compiler must support type inference and be able to generate the typed variant of the code. Note that the exam does not put any emphasis on validation or interactive features such as content assist, but that scoping rules are relevant, and that implementing type inference is closely related to implementing validation.

The rest of this document is organized as follows. First, Section 2 introduces the X21 language, after which Section 3 gives you hints on how you can implement your solution. Then, Section 4 describes the formal requirements on providing a solution, and last Section 5 describes how grading will be performed.

2 The X21 Language

The X21 language is designed to express streams of computations, similar to Java streams but with some important differences. Streams can be basic data types such as integers or compound data types.

2.1 Basic stream elements

Streams read values from input nodes, use function nodes to compute on these values, and store the computed values in outputs nodes.

The following is a small example of an X21 program:

```
function add1(x: int) { x+1 }
input number: int
node add1node [add1]
stream number to add1node to output inc_number_1
```

This defines the function “add1” that increments an integer by one, defines an input “number” of type integer, and a node “add1node” that is implemented using the

"add1" function. Last, these are all connected using a stream from the input to "add1node" and onwards to an output. Given a stream of integers 1,2,3 as input, the output will contain the numbers 2,3,4.

Note that multiple streams can be attached to the same node, for example the following code could be added at the end of the above program:

```
stream add1node to output inc_number_2
```

This would cause a copy of the output of "add1node" to be created, i.e., all values generated by "add1node" are now also sent to this new output (in addition to the original one).

Rather than implementing computation nodes using named functions, an anonymous lambda can also be written directly inside the node, i.e., the previous program could have used the following:

```
node add1node [(x: int) { x+1 }]
```

to declare the node "add1node", and the functionality of the program would have been the same.

2.2 User-defined datatypes

X21 has integers and strings as built-in datatypes, but programs can also define their own dataypes, for example:

```
data course { name: string, ects: int }
data teacher { name: string, age: int, course_taught: course }
```

defines two new types, one making use of the other. Data of a user-defined type can be read on an input, passed to computation nodes, and sent to an output. Functions can access the individual members and can create new values. Identity is by-value meaning two data values are equal if their members are equal.

A function adding one to the age of a teacher can be defined as follows:

```
function age_1_year(t: teacher) {
  new teacher[name=t.name,age = t.age+1,course_taught=t.course_taught]
}
```

Note the requirement to initialize all members of the new data value, even those whose values stays the same.

2.3 Expressions

Expressions in X21 include ordinary binary operations for arithmetic, conditional if-then-else expressions similar to xtend, local variables defined using functional-style let, as well as creation of user-defined data and access to individual fields of user-defined data. X21 programs have "parameters" which are variables that can be set prior to running the program and can be accessed read-only during execution. Last, there is a special value "none" that causes a function can return, causing the corresponding node to not produce any output.

```
X21: 'program' ID Declaration*
Declaration: Function | Input | Node | Stream | DataDecl | Parameter
Parameter: 'parameter' ID ':' Type
Function: 'function' ID Lambda
Lambda: '(' ID ':' Type ')' '{' Exp '}'
Input: 'input' ID ':' Type
Node: 'node' ID '[' (ID |  Lambda) ']'
Stream: 'stream' ID (',' ID)* ('to' Element (',' Element)* )+
Element: ID | '[' (ID | Lambda) ']' | 'output' ID
DataDecl: 'data' ID '{' ID ':' Type (',' ID ':' Type)* '}'
Type: 'int' | 'string' | ID
Exp: Exp '+' Exp | Exp '-' Exp | Exp '*' Exp | Exp '/' Exp
   | INT | '(' Exp ')' | 'let' ID '=' Exp 'in' Exp 'end'
   | ID ( '.' ID )* | 'none' | 'if' LogicExp 'then' Exp 'else' Exp 'end'
   | 'new' ID '[' ID '=' Exp (',' ID '=' Exp)* ']'
LogicExp: Exp ('=' | '<' | '>' | '<=' | '>=') Exp
```

Figure 1: BNF of X21

2.4 Syntax

The syntax of X21 is shown in Figure 1 as an abstract BNF. A program has a name and a number of declarations. Declarations are single-parameter functions, network inputs of a given type, network nodes made either using a function or an anonymous lambda, data declarations defining new datatypes, streams that describe input/output connections between nodes, or network parameters that define a value that can be accessed by the network. Types are either integers, strings, or user-defined datatypes.

Expressions include basic arithmetic expressions, local variables defined using functional-style let (as seen in the math exercises), variables (using the dot operator to access fields of user-defined datatypes), the 'none' value which causes a node to not generate any output, conditionals, and data instantiation.

The BNF does not show precedence for expressions, but let expressions, data access, conditionals, and data instantiation are primitive expressions that can be composed using the operators. As for the binary operators, multiplication and division have the highest precedence, with addition and subtraction being the weakest. All binary operators are required to be left-associative.

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 six X21 programs, the corresponding generated programs in two different versions (with and without type inference), abstract classes that can be used as basis for your code generation as outlined later in this section, and testing programs that your generated programs should pass. See the `README.txt` file for specific details.

3.2 Concrete example

The material provided includes six test programs, one of which is the P1 program shown in Figure 2. An outline of the generated code is shown in Figure 3. Note that

```
program P1
// Function that can be used to implement a node
function add1(x: int) { x+1 }
// Input to the stream
input number: int
// Nodes that can be streamed to/from
node add1node [add1]
node add2node [add1]
node add3node [add1]
// Two identical streams from same input
stream number to add1node to output inc_number_1
stream number to add2node to output inc_number_2
// A branching stream attached to add1node
stream add1node to add2node to output inc_number_3
```

Figure 2: Example X21 program

```
package p1;
...
public class P1Main extends GenericMainX21 {
  // Code for function add1
  private Object fun_add1(Object arg) {
    return funimpl_add1((Integer)arg);
  }
  private Object funimpl_add1(Integer _x){ return ((_x)+(1)); }
  // Code for input number
  private ComputeNode<Object,Object> node_number = new InputNode<Object>();
  public void inputNumber(Integer input) {
    node_number.put(input);
  }
  // Code for node add1node
  private ComputeNode <Object,Object> node_add1node = new AbstractComputeNode<Object,Object>() {
    protected Object function(Object input) {
      return fun_add1(input);
    }
  };
  ...
  // Output nodes
  private OutputNode<Object> node_inc_number_1 = new OutputNode<Object>();
  public List<Object> getInc_number_1() { return node_inc_number_1.getData(); }
  ...
  // Initialization of specific nodes
  protected void initializeNodes() {
    super.addNode(node_number);
    super.addNode(node_add1node);
    ...
  }
  // Initialize network as a whole
  protected void initializeNetwork() {
    node_number.addOutputNode(node_add1node);
    node_add1node.addOutputNode(node_inc_number_1);
    ...
  }
}
```

Figure 3: Example *untyped* code generation for the program of Figure 2

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

As can be seen, code is generated as a single class. If there were any user-defined datatypes, they would have been generated as separate classes. Since no type inference was used most nodes are "untyped", i.e., mapping Object to Object. Code is generated for each declaration in the same order as it appears in the X21 program. Functions (and lambdas) use a wrapper to make sure that the parameter can actually have the right type, while also being able to fit into an "untyped" (Object to Object) node[1]. Computation nodes are generated as anonymous inner classes. Input nodes are accompanied by an input method that can feed data into the node. Output nodes are similarly accompanied by an accessor method. The initialization code first adds the node to the network, and then sets up the streaming links.

3.3 Relevant material

Completely solving this exam requires working with type inference, and some of the analysis needed can be done as scoping rules. These two topics that have only been covered in a limited way in class. Thankfully Bettini's book has an excellent coverage of these topics as part of the course curriculum. Since your time is short, here are a few hints.

First, type inference 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 rules for a local variable are more complex compared to what you have seen in-class, since a local variable may refer to a parameter of the enclosing lambda, a node parameter, or a local variable from a let expression. Thus, 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]
```

The use of the "SJSymbol" technique will most likely will make it easier for you to solve the basic task of generating untyped programs. Note that this assignment has been designed such that you can complete substantial parts of it without needing type inferencing or scoping rules.

4 Solution

4.1 General principles

You will be evaluated on your ability to use xtext to implement an X21 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 X21 source code provided.

[1] Comparing the typed and untyped versions of the code shows you what the wrapper does. If there was only an interest in generating typed programs then the wrapper would have been redundant, but having a generator that could do both was useful for demonstration purposes.

2. Generate Java code which generates the same output when running and which passes the testing programs provided.

3. Compile programs similar to the ones provided (i.e., adding more declarations to one of the provided X21 programs does not break your compiler).

This means that: (1) If you need to make changes to the X21 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, (2) there is no requirement that the code that you generate be exactly the same as the code that has been provided, just that it is similar in functionality. Last, (3) hard-coded solutions tailored to the precise programs that are provided are not considered acceptable.

This assignment does not include any xtext-style validation, and does not include anything regarding interactive usage, i.e., content assist. Manipulation of the metamodel is however still needed; type inference is for example done by computations on the metamodel instance.

Please note that as a specific example it is possible to generate code that partially satisfies the requirements (i.e., can compile and passes the test programs provided) without using type inferencing, as shown above. Moreover, this can be done without writing any scoping rules, so it is possible to provide a fairly good answer without either of these (just an xtext grammar and a generator). The generated source code provided as an example includes both typed and untyped variants to show you both options. Similarly there are other restrictions you can make that partially fulfil the requirements, i.e., you may not have managed to implement the "let expression" feature. Such an answer 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 in an iterative fashion, where you first produce something that works on a minimal example, and then subsequently extend this to a more complete functionality. *Note however that if you are unable to implement an X21 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 4. 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're writing 6 pages then you're 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>/x21cc.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

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 X21 BNF), such that they can refer both to parameters of lamnda, parameters of the network, and local variables defined in let expressions?
3. Scoping Rules
   (a) Did you implement scoping rules for accessing members of user-defined datatypes? If not, what are the limitations.
   (b) Describe your implementation of any scoping rules included with your system.
4. Type Inference
   (a) Did you implement type inferencing? If yes, does it generate correct types for all node outputs in the provided X21 examples? If it does not generate correct types, what is the problem?
   (b) If you implemented type inferencing, briefly describe your approach.
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 inference, generator) verbatim.

Figure 4: Outline of the response template

file `x21cc.jar` must be a stand-alone version of your compiler generated using the approach described by Bettini in Chapter 5, "Standalone command-line compiler" (pages 94–97): an Xtend main is generated and exported as "Runnable jar file". The directory `ExamEclipse/` must contain your complete xtext project, everything included (i.e., the eclipse project containing your xtext and Xend files). The directory `ExamRuntime/` must contain your version of the runtime directory used by the runtime eclipse instance (so primarily X21 source files that work with your compiler and the corresponding generated Java files).

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 six provided examples, making use of type inferencing to compute the types of node input/outputs.
- Is implemented in a way that is understandable and makes appropriate use of xtext/Xtend features (such as cross references in the grammar file and type-switch or dispatch methods in the Xtend files).
- Is robust in the sense that illegal programs will not result in errors during code generation, and that the compiler always generates code for legal X21 programs (but given the reduced focus on validation, some X21 programs may result in illegal 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 X21 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 then assessed in terms of how well you have managed to solve the problem. The assessment includes running your compiler (using the provided jar file), inpecting 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. You are not allowed to interact with other students regarding this exam, this will be considered cheating. The santions put in place by SDU against cheating in online exams such as this one are quite severe. You are however allowed to contact Ulrik and make use of all materials (i.e., information on itlearning, the curriculum, material found on the Internet). Please see itslearning for additional details on how to contact Ulrik during the exam.