# Xtext language-based editor

Project in Computer Science - EDAN70
January 25, 2016

Mikael Johnsson
D12, Lund University, Sweden
dat12mj2@student.lu.se

Alexander Olsson
D12, Lund University, Sweden
dat12aol@student.lu.se

## Abstract

How easy is it to make an Eclipse like interactive editor for a DSL, Domain Specific Language, using Xtext? How simple is it to make syntax validation analysis using Xtext and how does that help the user to learn and write program for the language. This report also look at the editor functions, like open new domain specific editor windows and create an I/O console that is done using other frameworks.

## 1. Introduction

Domain Specific Languages (DSLs) makes it possible to focus on a particular problem domain and make development in that area more efficient, fast and easier that when a general purpose language is used. It exist plenty of different tools to make it easier to develop DSLs[1]. We have worked with Xtext to make an interactive editor for the a C-like language called SimpliC. An interactive editor is an editor that, for example, helps the developer by giving suggestions and validates the written code and reports problems during writing.

Xtext is a scalable framework for both developing a DSL as well as creating an interactive editor for the language. When starting a new project with Xtext, it gives the developer some starting stubs and example code which can be altered to make it easier to get going. When creating the model for the DSL in Xtext an Extended Backus-Naur Form[2] like grammar is used.

This report will cover how it is to work with Xtext, as well as similarities and differences to other frameworks. It also include an evaluation of the Xtext framework where we compare Xtext with JastAdd2, Beaver and JFlex. These specific languages were used to implement the same functionality of the SimpliC language as Xtext and therefore used in the evaluation. The evaluation compares lines of code and the number of words used in the different languages. There is also a section covering how it is to work in Xtext e.g. how to do type analysis, name analysis, scoping , interpreting, quick-fixes and context assist.

## 2. SimpliC

SimpliC is a language that implements a small subset of the functionalities of C and Java. SimpliC requires all source code to be in one file and supports only if, while, function calls as constructs for controlling the control flow. The variables and expressions are statically typed with no type inference and may be of boolean or int type. Statically typed variables means that each variable has to be given a type when it is defined. Functions may be of void type returning nothing or int type returning an integer value.

The structure of SimpliC is that a program is composed of functions that are composed of statements. All programs have access to the two predefined functions read and print. There are no global variables. The types of statements that are included are assignments, blocks, declarations, if, while, return and call. Many of the types of statements may take one or several values that are expressions. The program starts with the execution of the function named "main" with no arguments that must exist in a valid program.

Expressions are any mathematical expression containing numbers,variables and the following operators (+, -, *, /, %, ==, !=, >=, <=, >, <). The presented example in 1 is the code given during the course EDAN65 to introduce the language and it presents all the aspects of SimpliC[3].

In our implementation in Xtext some small extensions to the language was done, for example allowing global variables and the interpreter to use a breakpoint keyword "DEBUG" for stepwise execution and debugging purposes. During our Xtext implementation we also made the extension to allowing functions and variables to be of boolean type and the use of boolean literals *True*, *False*. Float variables has also been added and as a consequence the printf function has been added that writes out in float format. The print functions are special in that it is the only method that takes a value of any type and print out in a specific format.

## 3. Xtext

Xtext is a framework for both developing a DSL, with the functionality for validating syntax and interpreting but also the capacity for making a helpful interactive editor for the language. Xtext use EMF, Eclipse Modeling Framework, for representing it's data structures and abstract syntax tree (AST) model an ECore model[4, 5]. This is to allow Xtext to be use together with other tools using also using EMF.

When constructing a DLS with Xtext an Extended BackusNaur Form (EBNF)-like grammar language is used to set-up the language.The use of EBNF means that easy use of optional and repetition is supported. Xtext uses it's own generator to create a parser, an AST-meta model as well as start a full-featured Eclipse Text Editor where programs using the DSL can be implemented[6, 7].

When implementing a project, Xtext allows developers to extend existing grammars that can serve as starting ground for a new languages. By extending the included grammar *xtext.common.Terminals* you get predefined terminals for integers, comments and identifiers defined like in Java. By using Xbase, that is an implementation of

```
int gcd1(int a, int b) {
        while (a != b) {
                if (a > b) {
                        a = a - b;
                } else {
                        b = b - a;
                }
        }
        return a;
}
int gcd2(int a, int b) {
        if (b == 0) {
                return a;
        }
        return gcd2(b, a % b);
}

int main() {
        int a;
        int b;
        a = 10;
        b = 2;
        /* Test two different implementations of GCD
        *  algorithms and print the result.
        *  The results should be equal provided that
        *  both inputs are positive.
        */
        a = read();
        b = read();
        print(gcd1(a, b));
        print(gcd2(a, b));
        int diff = gcd1(a, b) - gcd2(a, b);
        print(diff);
        return 0;
}
```

**Figure 1.** Example program in the SimpliC language used to introduce the language during EDAN65

the complete javatype system, as the basis for your DSL you get access to a language model supporting all java types that gives you access to additional tools for development for the JVM.

Xtext is also scalable and makes it possible to customize every aspect in simple ways to form full programming language implementations.

Many products, both internal and external which is based on Xtext, have been implemented by companies like Google, IBM, BMW and several others[8].

### 3.1 Similar workbenches to Xtext

There are several similar language workbenches compared to Xtext, e.g. Spoofax.

Spoofax is a language workbench which is bound to Eclipse since it's built on top of the Eclipse platform. Unlike Xtext which can easily be downloaded for the Eclipse platform but isn't bound to it, since it can be run on any JVM independently of the Eclipse IDE[1, 4]. Xtext is also released in a version for use on the IntelliJ IDEA platform.

Both Xtext and Spoofax uses a text based approach on developing DSLs. As well as Xtext, Spoofax offers tools to define grammars. One part that's different with Spoofax compared with Xtext is that in Spoofax there's no need to run multiple instances of Eclipse at the same time. This makes it more smooth to debug errors and get feedback. Since in Xtext, when it's time to test the DSL an extra application of Eclipse is booted.

It's easier to create an Eclipse plug-in for the final language with Spoofax than Xtext since Spoofax uses the Eclipse IDE Metatooling Platform, IMP.

Similarities between Xtext and other frameworks can be viewed in Figure 2.

## 4. Working with Xtext

The first thing that needs to be done when implementing a DSL in Xtext is specifying the languages syntax grammar.

### 4.1 Designing Grammar

Xtext uses a high level grammar supporting all EBNF constructs, meaning it allows the use of optionals, alternatives, list and arbitrary repetition for DSL parser specification. The constituent terminals and non terminals in the parser rules can be named. The Xtext grammar joins both the steps of terminal parsing using regex and constructing an AST with a context grammar[7].

The basic syntax, use of +,*,?,|, closely resembles standard regex

| | | Ensō | Más | MetaEdit+ | MPS | Onion | Rascal | Spoofax | SugarJ | Whole | Xtext |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Notation | Textual | ● | ● | | ● | ● | ● | ● | ● | ● | ● |
| | Graphical | ● | ◒ | ● | | | ◒ | | | ● | |
| | Tabular | | ● | ● | ● | | | | | ● | |
| | Symbols | | | ● | ● | | | | | ● | |
| Semantics | Model2Text | | ● | ● | ● | ● | ● | ● | ● | ● | ● |
| | Model2Model | | | ● | ● | ● | ● | ● | ● | ● | ● |
| | Concrete syntax | | | ● | ● | ● | ● | ● | ● | | |
| | Interpretative | ● | | ● | ● | | ◒ | ● | | ● | ● |
| Validation | Structural | ● | ● | ● | ● | ● | ● | ● | ● | ● | ● |
| | Naming | ◒ | ● | ● | ● | ● | | ● | | ● | ◒ |
| | Types | | | | ● | | | | ● | | ● |
| | Programmatic | ● | | | ● | ● | ● | ● | ● | | ● |
| Testing | DSL testing | | | | ● | | ◒ | ● | | ● | ● |
| | DSL debugging | ● | | ● | ● | | ● | | | ● | ● |
| | DSL prog. debugging | ● | | | ● | | | | | ● | ● |
| Composability | Syntax/views | ● | | ● | ● | ● | ● | ● | ● | ● | ◒ |
| | Validation | | | ● | ● | ● | ● | ● | ● | ● | ● |
| | Semantics | ● | | ● | ● | ● | ● | ● | ● | | ● |
| | Editor services | | | ● | ● | ● | ● | ● | ● | | ● |
| Editing mode | Free-form | ● | | ● | | ● | ● | ● | ● | | ● |
| | Projectional | | ● | | ● | ● | | | | ● | |
| Syntactic services | Highlighting | | ◒ | ● | ● | ● | ● | ● | ● | ● | ● |
| | Outline | | | ● | ● | ● | ● | ● | ● | ● | ● |
| | Folding | | ● | ● | ● | ● | ● | ● | ● | ● | ● |
| | Syntactic completion | | | ● | ● | ● | | ● | ● | | ● |
| | Diff | ● | | ● | ● | ● | ● | ● | ● | | ● |
| | Auto formatting | ● | ● | ● | ● | ● | ● | ● | | ● | ● |
| Semantic services | Reference resolution | | ● | ● | ● | ● | ● | ● | ● | | ● |
| | Semantic completion | | ● | ● | ● | ● | ● | ● | ● | ● | ● |
| | Refactoring | | ◒ | ● | ● | | ● | ● | | ● | |
| | Error marking | | ● | ● | ● | ● | ● | ● | ● | ● | ● |
| | Quick fixes | | | | ● | | | | | | ● |
| | Origin tracking | ● | | ● | ● | | ● | ● | ● | | ● |
| | Live translation | | | ● | | ● | ◒ | ● | | ● | ● |

**Figure 2.** A table describing similarities between Xtext and other frameworks, full circle is equal to full support, half filled circle is equal to limited/partial support.

notation. This make it very easy for people familiar with regex to write a specification for a language. Xtext syntax grammar language however have many functionalities in addition to token parsing and can directly specify the abstract grammar model for the language.

Rules can have "returns" and the "{Rule_name}" instantiation specifiers that together determine what sort of node a rule should be generated and if it need to appear in the AST model. The grammar support several other actions for shaping the AST beyond these. For example writing *Argument returns Decl:* means that when the rule "Argument" is parsed a node in the AST of type "Decl" should be created.

Giving a node a named attribute is a third way to shape the AST since doing so implicitly make the rule an actual node unless changed by a "returns" declaration. The Special node attribute "name" informs the model that a node type should be referable and is the referable identifier.

In Figure 3 a small part of our grammar is shown displaying many of the features for the Xtext language. In the rule "Assign" the *name* attribute is written as *name=[Decl]* meaning that the *name* attribute should refer to a "Decl" node and must be the name of a decl node.

The recommended parser used with Xtext, Antlr(ANother-Tool-for-Language-Recognition), is a LL(*) parser[9]. Since LL(k) parsers can't handle left recursion the user is forced to write more rules or use more advanced rules using repetitions in some situations when a LR parser would have needed less rules and less work. Antlr has however the option to use backtracking for solving common prefix but it costs some performance. Apart from removing left recursion it's easy to implement any EBNF grammar[4, 7].

The only time we had to think how we implemented the grammar was for the expressions but there was simple ways around the problem. The use of so called syntactic predicates can tell the

```
Assign:
name=[Decl] '=' value=Expr
;

Expr:
Relation ({RelationTerm.lh=current} op=RelationOp right=Relation)*
;

Relation:
Term ({TermTerm.lh=current} op=TermOp right=Term)*
;

Term : Factor ({FactorTerm.lh=current} op=FactorOp right=Factor)*;
```

**Figure 3.** Constructing the language syntax using Xtext

parser what to do in certain situation like the dangling else problem. A syntactic predicate tells the parser to use the rule alternative that contains it when it encounters a rule that has one in atleast one of its alternatives and can match the the token following the => . The parser must then use that alternative before any of the other alternatives is attempted [10, 11].This prioritizes that alternative over others and is a way to avoid problem with common prefix or identical syntax. A predicate => in a rule like *if (cond) stmt* (=¿ else stmt*)* tells the parser that we want the innermost if statement to try an match a *else* token in the case of nested if statements. In the SimpliC language this is not a problem since the conditional actions are encapsulated by the rule for a block statement that removes the syntactical ambiguity.

In our case we use syntactic predication for telling Antlr to match greedily when parsing the function arguments in function declarations and in function calls without allowing dangling commas.

Another example is that we use *Argument returns Decl : => type=Type name=ID;* where the predicate => allows the parser to prioritize the argument rule over a general *Decl*. Antlr would otherwise complain that both a argument and a unassigned variable declaration have the same pattern. The predicate resolves this conflict of multiple parser alternatives being able to consume the same pattern. A simple alternative fix with out the predicate is placing the *;* token in the declaration rule instead of at the statement rule. The two conflict rules would then have different syntax and the problem would disappear.

Developing an editor for a C like language, like the SimpliC language, is in several aspects very easy because the normal modules for scoping, referencing essentially works as wanted per default. We made some extensions to SimpliC like extending the type system to using bool, int, float so we could test type analysis more extensively.

### 4.2 End User Assisting editor functions

From start in the default Xtext editor, after the model has been specified, are a lot of functions presents. The editor provides real time error messages for parsing errors, write suggestion based on the rules and does semantic cross referencing possibly needing a different scoping implementation[12, 13]. The outline view show one node for every node in the model of current file except for predefined terminals and nodes with the "name" attribute the labels will only show "<nnamed>". The default name scope is a project global scope but SimpliC works on single file and also have restricted visibility inside functions and blocks and uses shadowing like in C or Java. This needs to be implemented manually.

Language specific validation, quickfixes, tooltips, descriptions and labels for the model outline is not done but stubs for each of these

functionalities has been generated, by the Xtext project templete, in the workspace. Figure 5 shows the different catagories of stubs that xtext generates. All of these functions are implemented in a similar way which is to specify a set of functions / checks that should be run on all nodes in the model of each type that needs some specialized behavior. There are only some minor differeneces in that for validations and quickfixes the keywords @check and @fix respectively are use to denote the functions that the editor should run on changes. Labels for the outline model and the tool tip descriptions, displayed when hovering over the code representing the model node, are written using a ”text(Nodetype)” function for each node returning the text to display.

The functions could be written in standard Java or in Xtext with the included Xtend language that is a dialect of Java that can simplify the work.

```
513
514        @Check
515        def HasMainFunction(Model m) {
516            for (Function fun : EcoreUtil2.getAllContentsOfType(m, Function))
517                if (fun.name.equals("main") && (fun.arglist.size == 0)) {
518                    return;
519                }
520            info(
521                'The program doesn\'t have a main() function and is not runnable',
522                SimplicPackage.Literals.MODEL__PROGRAM,
523                NOMAIN,
524                ""
525            );
526        }
```

**Figure 4.** Constructing the languages syntax using Xtext

Xtend and Java are very similar but Xtend contains several improvements that decreases the amount of code that needs to be written. This is due to that Xtend both type infer and uses implicit parentheses for function calls and several new features and keywords[14, 15]. Java and Xtend is totally interoperable and most code easily be converted between the two. An automatic converter is included for translating Java to Xtend .

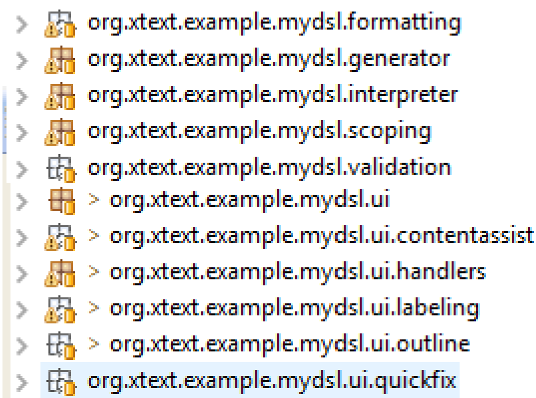


**Figure 5.** All Xtext framework stub categories for various editor and DSL functions

### 4.3 Testing

Xtext also sets up a plugin for running JUnit tests on the model and provides some functions for parsing an AST from text while counting the number of syntax errors. This means that you get syntax validating form the start without any coding. Once you have written some validation test in your language validator these can be run by the framework in the JUnit tests.

## 5. Extensions to the Xtext editor

There are some aspects that are outside the scope of Xtext that we wanted to have for our editor. Xtext project wizard gives you code for making and running a compiler that is run every time a file is saved by we also wanted to run a interpreter and display the output in the console in a easy way. As far as we can determine after reading various tutorials and the documentations on Xtext’s homepage is that this functionality is not part of the scope of Xtext and we therefore had to do some work with the Eclipse framework. We needed to create a new type of console for our editor since the Java console is not available[16]. We wrote the Interpreter using Java and then connected to the GUI by adding new Eclipse extensions. In the Figure 6 we show how the IDE environment looks like after writing and executing a small example for calculating Fibonacci numbers in the our SimpliC console.

Fortunately for most of the extensions we need, like adding buttons and views, Eclipse have good template wizards that immediately gives working skeleton code. In these cases we only had to add the actual actions of the button or the view.

Adding a console and a drop down option when selecting a file was harder since Eclipse had no complete template for it, which required us to look up some examples and tutorials. The problem for us was that you don’t have access to write to the standard Java console in the new application since the standard output data stream is associated with the Eclipse environment that launched the Xtext editor environment. We also do not interpret our programs in new JVMs.

Finally we followed the Eclipse template for making a standalone Eclipse application for Eclipse along with the alternative of exporting the finished editor in the form of eclipse plugins that can be installed locally or from a install website.

## 6. Evaluation

We have firstly compared the code and work methodology of working with Xtext with the result of using the tool combination JastAdd2, Beaver and JFlex since we have implemented the SimpliC language using all languages.

Our conclusion when it comes to syntax specification is that the quantity of rules maybe slightly larger for Xtext’s model parsing due to that Antlr is LL(*) compared to Beaver that is an LR parser. In the case of our language the difference is minimal and the only real difference is in rules for the expressions.

The Xtext specification however does much more than the specification for Beaver, JFlex or JustAdd. The Xtext specification does all 3 generation steps at once and each rule can be written much more compact using less number of words since there is no need to write node construction actions passing attributes to AST node constructors and only one specification is needed. Beaver is the worst in readability and uses the most words since Beaver have to define the node type of rules separate from the rule , doesn’t support list attribute but these need extra rules. Beaver needs to both specify the terminals in the parsing rules,name them and then pass the to a constructor.[17] This creates a write redundancy since all names are written at least twice and unnecessary writing like ”return new ...”.

The Xtext grammar both support more constructs that simplify the grammar and is also shorter and according to us more readable. Table 6 which is shown in Figure 7, show how much code and how many lines that we needed to specify the language.

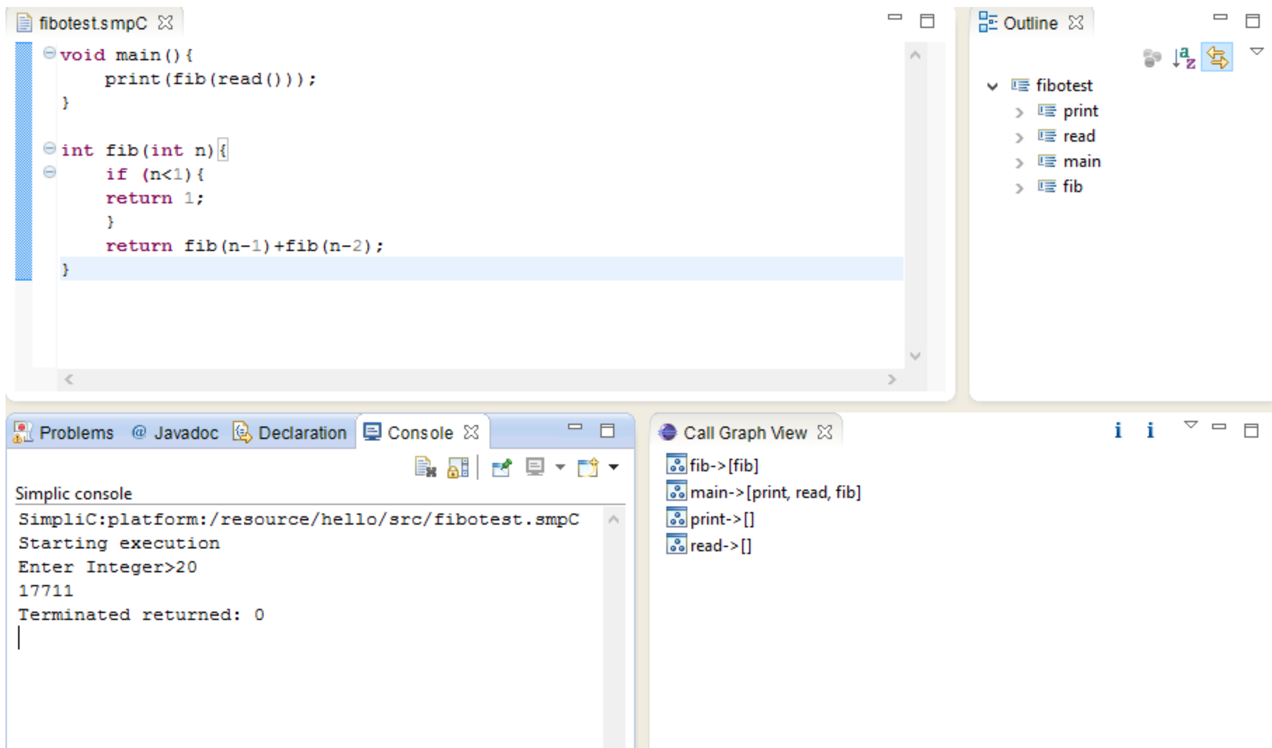


**Figure 6.** Simplic environment in Eclipse showing the text editor,console,call graph view and outline after running a example calculating fibonacci numbers

In the following table aspects between the two ways of design a DSL is compared. Extensions that we have made to SimpliC this time is not included in code amount evaluations.

| Metric | Xtext | JastAdd & co |
|---|---|---|
| Syntax specification | | 211 words |
| Grammar specification | 181 words | 529 words |
| AST specification | | 136 words |
| Syntax specification | | 76 LOC |
| Grammar specification | 114 LOC | 162 LOC |
| AST specification | | 39 LOC |
| Rules in Grammar | 28 st | 34 st (26 st*) |
| Name analysis | 763 words | 461 words |
| Reachability analysis | 188 words | 287 words |
| Type analysis | 623 words | 228 words |
| How easy is it to make Grammar? | 4 | 2 |
| How easy is it to make validation? | 3 | 4 |
| How easy is it to make a interpreter? | 3 | 4 |

* if optimized similarly to the Xtext version

**Figure 7.** Comparative measurements for Xtext vs JFlex, Beaver and JastAdd

The various type of analysis of the AST is done differently with Xtext compared to JastAdd since JastAdd works with RAG (references attribute grammar) that are not supported in Xtend or Java. Analysis is done using a paradigm resembling the visitor pattern or logic heavy methods executing on top nodes checking there constituents. As a result interpreting and much of the analysis is not as readable as with the JustAdd implementations.

Xtend is a improvement over Java since it is more compact and the keyword 'dispatch' can hide much of the type casting to specialized types that needs to be done using plain Java. The 'dispatch' keyword creates a method that does multiple dispatch for the function. Multiple dispatch is selecting a specific function/behavior depending on the type of the input. Along with the Eclipse plugins included for Eclipse are a functionality for converting Java code to Xtend so its not much difference and performances wise similar.

## 7. Conclusions and future work

Xtext is an easy to use framework and you quickly get a usable editor for simple development. The grammar language enables the language rules to be written short and very readable.

We implemented the SimpliC language with syntactic and semantic validations. We didn't implement an advanced debugger like the one included with Java to complete a full IDE with interpreter,editor and debugger. In our implementation the only debug capability is with the program statement "DEBUG;" that results in displaying current variable values. A more advanced debugger would have more functionality and allow variable editing during breakpoints like the Java debugger. The Java debugger also have associated views in the IDE. If we had taken the time to learn and

work with Xbase and the XbaseInterpreter we might had been able to use the existing debug functionality in the Java debugger for Eclipse.

SimpliC is an very simple language and similar to standard general purpose languages and now when we have had some experience with Xtext we could have implemented a more useful and powerful language. Some Editor features is not implemented. We didn't write many code templates and we could have written even more validation checks, writing style rules. Syntax highlighting meaning the use of color to highlight important elements in the editor could have been extended and optimized. Auto formatting could have be fully implemented instead of only limited block indentation.

When coding for a larger more complex language it would be important to do more automatic testing with JUnit. Xtext prepares a plugin for running language tests but we haven't really used it because our language is so simple and Xtext so modular that manual testing of changed language aspects can be done quickly. During continued development more active writing of test would be useful. Most of our test cases were added at the end of the development when the editor was done.

A lot more could be done for the standalone Eclipse version of the IDE where a suitable intro page and help manuals could be created. A full dot format like actual graph for reachability could have been implemented instead of only a table view. This could be done using some graphical framework also using EMF.

The code analyzing could be optimized and extended by also using the tool JavaRAG that would allow the use of RAG for java based AST models including EMFs Ecore model used by Xtext [18]. That would allow some types of checks to be implemented more easily and efficient.

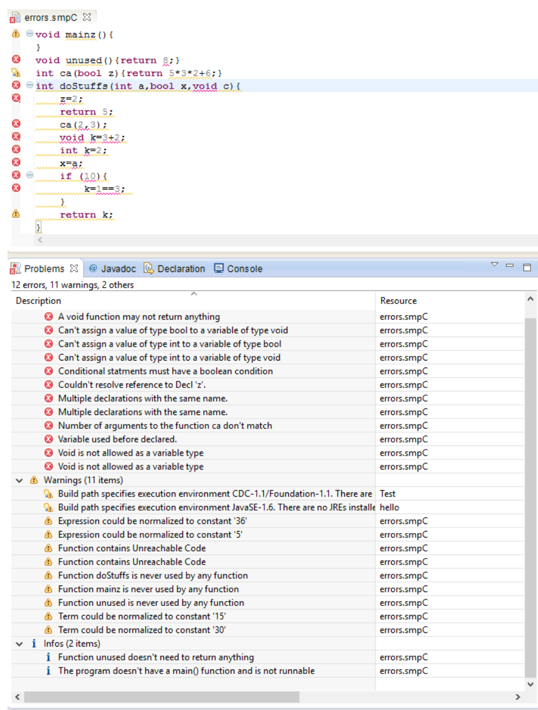

**Figure 8.** A short program failing a large subset of all validation tests we have written

## References


[1] Leonard Elezi, Spoofax vs Xtext, Last edited: 20 December 2013, Retrieved: 8 January 2016, http://msdl.cs.mcgill.ca/people/hv/teaching/MSBDesign/201314/projects/Leonard.Elezi/

[2] Wikipedia, Extended BackusNaur Form, Last edited: 2 January 2016, Retrieved: 8 January 2016, https://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form

[3] Grel Hedin, EDAN65 - Compilers, Last edited: 6 November 2015, Retrieved: 8 January 2016, http://cs.lth.se/edan65

[4] Eclipse, LANGUAGE ENGINEERING FOR EVERYONE!, Retrieved: 8 January 2016, https://eclipse.org/Xtext/ 2015-11-09

[5] Eclipse, Eclipse Modeling Framework (EMF), Retrieved: 8 January 2016, http://www.eclipse.org/modeling/emf/

[6] TechWars, Xtext, Retrieved: 8 January 2016, http://www.techwars.io/tools/xtext/

[7] Jianan Yue, Transition from EBNF to Xtext, Retrieved: 6 December 2015 http://ceur-ws.org/Vol-1258/src5.pdf

[8] S. Erdweg et al, The State of the Art in Language Workbenches, International Conference on Software Language Engineering, SLE 2013, Retrieved: 2 December 2015. https://www.student.informatik.tu-darmstadt.de/ xx00seba/publications/language-workbench-state.pdf

[9] ANTLR, About The ANTLR Parser Generator, Retrieved: 8 January 2016, http://www.antlr.org/about.html 2015-12-31

[10] Wincent, ANTLR predicates, Last Edited: 7 April 2011, Retrieved: 6 January 2016 https://wincent.com/wiki/ANTLR_predicates

[11] Eclipse, Syntactic Predicates, Retrieved: 14 December 2015, https://eclipse.org/Xtext/documentation/301_grammarlanguage.html#syntactic-predicates

[12] Efftinge, S., & Vlter, M. (2006, October). oAW xText: A framework for textual DSLs. In Workshop on Modeling Symposium at Eclipse Summit (Vol. 32, p. 118). Retrieved: 2 December 2015.

[13] Eysholdt, Moritz, and Heiko Behrens. "Xtext: implement your language faster than the quick and dirty way." Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion. ACM, 2010.

[14] Eclipse, Xtend, Retrieved: 6 December 2015, http://www.eclipse.org/xtend/

[15] Eclipse, Java Interoperability, Retrieved: 6 December 2015, https://eclipse.org/xtend/documentation/201_types.html

[16] Eclipse, FAQ How do I write to the console from a plug-in?, Last edited: 8 February 2011, Retrieved: 8 January 2016, http://wiki.eclipse.org/FAQ_How_do_I_write_to_the_console_from_a_plug-in%3F

[17] Beaver SourceForge, Beaver - a LALR Parser Generator, Retrieved: 2 December 2015, http://beaver.sourceforge.net/spec.html

[18] Niklas Fors,Gustav Cedersjö,Görel Hedin,"JavaRAG: a Java library for reference attribute grammars", MODULARITY 2015 Proceedings of the 14th International Conference on Modularity Pages 55-67