Project descriptions
compiler projects

This document describes available compiler projects for the course Project in Computer Science (EDAN70). All projects include constructing some artifact, evaluating it, and relating to some literature. All projects are related to ongoing teaching and research at the department. All artifacts will be open-sourced under the modified BSD license.

Projects

1. LR conflict demonstrator
2. CUP backend to JastAddParser
3. Object-oriented metrics for Java programs
4. JastAdd library for name analysis
5. Oberon-0 compiler in JavaRAG
6. Lab language compiler in Scala and Kiama
7. Xtext language-based editor
8. Spoofax language-based editor
9. Extending the Lab language with an LLVM backend
10. Java call graphs and comparison with Soot
1 LR conflict demonstrator

Resolving conflicts in LR grammars can be difficult. A first step is to understand why the conflict occurs is to construct example programs that demonstrate the conflict. Such programs can be constructed automatically by analyzing the LR state machine: find a path of terminals and nonterminals from the start state to the conflict state. Then expand each nonterminal to a sequence of terminals. Few LR parser generators have such debugging support, but many have an option for printing out the state machine, so that an external tool can synthesize such example programs.

The task is to implement such a tool for the CUP LALR(1) parser generator. The CUP parser generator is similar to the Beaver parser generator, but is currently better maintained than Beaver. Possibly, the EDAN65 Compilers course will switch from using Beaver to using CUP next year, and the new tool could then be used in the course. The tool should be evaluated by constructing example grammars that give conflicts, showing that it works on these examples. In addition to the usual textbook examples of expression ambiguity and the dangling else problem, real life LR conflicts can be studied. For example, Java 8 has several documented LR conflicts. A possible extension of the project is to generate a .dot file for visualizing the state machine using GraphViz. If time permits, a small user study can be performed to compare how easily users understand the cause of an LR conflict with and without the tool.

The tool should be implemented using JastAdd and JastAddParser.

Resources:

- Draft paper on LR conflicts in Java 8:

- The CUP project:
  CUP LALR parser generator for Java. [http://www2.cs.tum.edu/projects/cup/](http://www2.cs.tum.edu/projects/cup/)

- PhD thesis on ambiguity detection, primarily in the context of generalized LR parsing:
  B. Basten: Ambiguity detection for programming language grammars. PhD thesis. University of Amsterdam, 2011. [http://hal.inria.fr/docs/00/64/40/79/PDF/PhDThesis_Bas_Basten.pdf](http://hal.inria.fr/docs/00/64/40/79/PDF/PhDThesis_Bas_Basten.pdf)
2 CUP backend to JastAddParser

JastAddParser is a preprocessor for the Beaver parser generator, providing a slightly higher-level parser specification. In particular, JastAddParser supports that a parser specification can be split into several modules, whereas Beaver supports only a single specification file. JastAddParser is used in many JastAdd projects, including the JastAddJ Java compiler and the JModelica.org compiler for Modelica and Optimica. JastAddParser is itself implemented using JastAdd (and JastAddParser).

CUP is an LALR(1) parser generator that is currently better maintained than Beaver. It might be beneficial to use CUP instead of Beaver as the backend for JastAddParser, in particular if the parsing speed of CUP turns out to be higher than that of Beaver.

The task is to implement a new backend to JastAddParser, generating specification files for CUP instead of for Beaver. The resulting parser should be evaluated on the test suite for JastAddParser, and obstacles to implementing a full CUP backend should be identified. If relevant, the test suite should be extended. A comparison should be done on the parsing speed of the generated CUP and Beaver parsers.

Preferably, the implementation should be done in such a way that both the CUP and Beaver backends can be easily maintained, and allowing new backends for other parser generators to be added.

If the resulting backend is sufficiently powerful, and if time permits, it would be very interesting to do an evaluation on the JastAddJ and JModelica parsers.

Resources:

- The JastAddParser project: [https://bitbucket.org/jastadd/jastaddparser](https://bitbucket.org/jastadd/jastaddparser)
- The Beaver project: [http://beaver.sourceforge.net](http://beaver.sourceforge.net)
- The CUP project: [http://www2.cs.tum.edu/projects/cup/](http://www2.cs.tum.edu/projects/cup/)
3 Object-oriented metrics for Java programs

A number of different metrics for object-oriented programs have been defined in the literature. One example is Robert Martin’s metrics for packages, which are also discussed in the courses on Object-Oriented Modeling and Design (EDAF10 and EDA061). These include efferent and afferent coupling, indicating how many classes in a package that depend on or are dependent on classes in other packages. These metrics are used for computing the stability or abstractness of a package. Another metrics suite is the CK metrics (Chidamber and Kemerer) for classes, computing things like the number of methods in a class, the depth of the inheritance hierarchy, the number of subclasses of a class, etc.

The task is to implement Martin’s metrics for Java, by extending the JastAddJ compiler, and to perform measurements on example libraries and projects. An important aspect is that it should be possible to remove standard libraries from the measurements. It is also interesting to measure at different levels of granularity, for example package group, package, class, etc. Performance of the tool should also be measured. The implementation should include suitable test cases for the metrics. If there is time, it would also be interesting to compare to other tools implementing these or similar metrics. Most other tools perform metrics on the bytecode, instead of on the source code. This can be limiting, and preclude the computation of some metrics. Examples of other tools computing some variant of Martin’s metrics are CodePro and JDepend. It would be very interesting to run these tools on the new test suite, to see if they compute the same results, and to compare the performance with the new tool.

As a starting point for the implementation, an existing implementation of the CK metrics for Java 5 could be used, providing some inspiration for how metrics can be implemented on top of JastAddJ.

This work is interesting for an ongoing research project on large scale software development, run by Prof. Martin Höst, who will also act as an extra resource person in the project.

Resources:

- J. Öqvist and G. Hedin: Extending the JastAdd extensible Java compiler to Java 7. PPPJ 2013. [http://dl.acm.org/citation.cfm?id=2500843](http://dl.acm.org/citation.cfm?id=2500843)
4 JastAdd library for name analysis

Many languages have similar kinds of name analysis. It would be useful to write a JastAdd library, capturing common kinds of name analysis, and reuse it for different languages. This can be done by placing attributes and equations in interfaces, and using aspects to declare how AST classes for a particular language implement these interfaces.

The task is to explore this idea, and come up with JastAdd libraries that are useful for doing name analysis for a variety of common naming schemes in programming languages, for example block nesting with and without declare-before-use, object-orientation with inheritance, etc.

A similar approach is used in a tool called Spoofax, where a domain-specific language NaBL (Name Binding Language) is used for defining name analysis. The project should investigate to what extent the NaBL constructs can be expressed as a JastAdd library. The work can start with supporting a simple procedural language like the lab language in EDAN65. The work can then be extended to more advanced constructs, for example to support inheritance.

Another source of related work is work done by Uwe Kastens and William Waite on modular name analysis in standard attribute grammars (without reference attributes).

The evaluation should compare the JastAdd specification size with and without the library for some smaller examples. Obstacles, such as incomplete support in JastAdd, should be identified. There should also be a qualitative evaluation discussing similarities and differences to NaBL.

Resources:

- NaBL examples: [http://metaborg.org/wiki/nabl/examples](http://metaborg.org/wiki/nabl/examples)
5 Oberon-0 compiler in JavaRAG

JavaRAG is a Java library implementing RAGs, developed by PhD students Gustav Cedersjö and Niklas Fors at the department. Compilers written using JavaRAGs are slower and more verbose than those written using JastAdd, but the benefit is that you can program in pure Java, avoid the generation phase, and do RAG evaluation on any Java abstract syntax tree. For example, you can define RAGs on top of hand-implemented ASTs, or use parser-generator specific ASTs. The JavaRAG library is currently used in the CAL research project, implementing parts of a CAL compiler. JavaRAG has also been evaluated on some smaller languages.

The goal of the project is to implement an Oberon-0 compiler using JavaRAG, and this way provide more evaluation data for the JavaRAG library. Oberon-0 is a tiny language used for benchmarking compiler tools. There are Oberon-0 compilers implemented in a number of different tools, including JastAdd. The benchmark includes several language modules and analyses, as well as code generation to C.

The Oberon-0 compiler should reuse the parser and abstract syntax tree from the JastAdd implementation, but the semantic analysis and code generation should be implemented using JavaRAG. The resulting implementation should be compared to the JastAdd implementation concerning compilation performance and specification size.

Resources:

• N. Fors and G. Hedin: A JastAdd implementation of Oberon-0. Draft 2014.


• The JavaRAG library: https://bitbucket.org/javarag/javarag

• The LDTA tool challenge description (describes the Oberon-0 benchmark project): http://ldta.info/tool.html
6 Lab language compiler in Scala and Kiama

Scala is an advanced programming language combining support for object-oriented and functional programming. Kiama is a Scala library for attribute grammars. It implements many of the constructs in JastAdd, including reference attributes, parameterized attributes, and circular attributes. However, collection attributes are not supported. For parsing, Kiama normally uses a Scala parser combinator library.

The task is to implement the lab language in Kiama, making it pass the same test cases as your own lab implementation in JastAdd. To evaluate the approach, the implementations should be compared with respect to specification size and performance. If time permits, both implementations can be extended with more language constructs.

Resources:

- The Kiama library: [https://code.google.com/p/kiama/](https://code.google.com/p/kiama/)
- Parsing in Kiama: [https://code.google.com/p/kiama/wiki/Parsing](https://code.google.com/p/kiama/wiki/Parsing)
7 Xtext language-based editor

Xtext is a framework for developing tooling for programming languages and domain specific languages, including the development of interactive Eclipse IDEs with syntax coloring, content assist, etc., for the language. Xtext uses the Eclipse Modeling Framework (EMF) for internal representation of ASTs. Instead of using a formalism like attribute grammars, it provides APIs for solving common compilation problems.

The task is to use Xtext to implement an editor and possibly also a compiler for Oberon-0. Oberon-0 is a tiny language used for benchmarking compiler tools. There are Oberon-0 compilers implemented in a number of different tools, including JastAdd. The new tool should use an EMF abstract grammar that is as similar as possible to the JastAdd abstract grammar for Oberon-0. This is because in a subsequent project, we would like to combine the new Xtext Oberon-0 frontend with JavaRAG (a Java library for reference attribute grammars).

The new tool should be evaluated from the following aspects:

- Can the editor and compiler share parts of the specification? How does the modularization of the compiler compare to JastAdd?
- What parts of the Oberon-0 test suite was supported? Qualitative evaluation of what aspects were easy and what were more difficult.
- Performance comparison of the compiler with the corresponding JastAddJ implementation.

Resources:

- The Xtext project: [http://www.eclipse.org/Xtext](http://www.eclipse.org/Xtext)

Variants

Instead of implementing Oberon-0, implement an editor for the .ast language. The tool should preferably handle several .ast modules. A goal here would be to provide content assist in the editor, and perhaps some simple refactorings. It would also be useful to generate a visual representation in the .dot format, to visualize grammars using the GraphViz tool. An interactive tool like this would be useful both for the Compilers course and for projects using JastAdd. As evaluation, the tool should run test cases for .ast, and a small user study should be conducted.
8 Spoofax language-based editor

Spoofax is a platform for developing interactive language editors as Eclipse plugins, including syntax highlighting and code completion. Spoofax supports parsing using GLR parsing (generalized LR parsing). Spoofax does not support attribute grammars, but instead uses a special name binding language (NaBL) for doing name analysis, and rewrite rules using the Stratego language, for other semantic analyses and code generation.

The task is to implement the lab language in Spoofax, generating an interactive editor for the lab language. The implementation should pass the same test cases as your own lab implementation in JastAdd. To evaluate the approach, the implementations should be compared with respect to specification size (for modules that are comparable in functionality). If possible, performance should also be compared in some way.

Spoofax resources:

- The spoofax platform: [http://strategoxt.org/Spoofax](http://strategoxt.org/Spoofax)

Variants:

Instead of implementing the lab language, implement an editor for the .ast language. The tool should preferably handle several .ast modules. A goal here would be to provide content assist in the editor, and perhaps some simple refactorings. It would also be useful to generate a visual representation in the .dot format, to visualize grammars using the GraphViz tool. An interactive tool like this would be useful both for the Compilers course and for projects using JastAdd. As evaluation, the tool should run test cases for .ast, and a small user study should be conducted.
9 Extending the Lab language with an LLVM backend

LLVM is a compiler infrastructure project for programs written in arbitrary languages. Despite its acronym, it is not a virtual machine, but rather an optimization framework. It defines a low-level assembly-like language, but that is platform independent.

The task is to implement a new backend to the lab language, that generates code for LLVM instead of for x86, and extending the lab language to cover more constructs. Performance measurements should be made, comparing the x86 implementation with the LLVM implementation, using LLVM tools to compile the LLVM IR code to optimized x86.

LLVM resources:

- LLVM site: [http://llvm.org](http://llvm.org)

Variant:
Do a similar project, but generate Java bytecode instead.

Java Bytecode resources:

10 Java call graphs and comparison with Soot

An important component of many Java analyses is to be able to compute call graphs. Because of polymorphism, this requires the complete set of classes to be known, ignoring dynamic loading.

The task is to implement Java call graphs as an extension of JastAddJ, and to compare performance with the Soot tool. Soot is a Java optimization framework, working at the byte code level, and has several methods for computing call graphs. A test suite should be developed to be able to verify that the tool computes the correct result.

A basic version of the call graph should only follow calls according to the static types. Additional possible extensions are:

- Take subclassing into account. This is probably necessary for a fair comparison with Soot.
- Compute reachability from the main method. This will require using circular attributes.
- Generate visualizations of the call graph, by generating a .dot file that can be visualized using GraphViz.

Resources: