薛德明
B97902122

# Language support for compiler development
## Compiler Design 2011 – Final Report

## Part A

### 1. Goal
Add a new language construct to Javac to reduce boilerplate and increase implied structure in the development of arbitrary compilers. Naturally, the construct resembles the definition of a CFG grammar, and is thus called a "grammar". That grammar can then be used by a compiler developer to easily assemble an AST, using a set of grammar-agnostic universal tools, including a scanner and parser. In practice, there are still some restrictions which are covered in section 3.
It is vital to give compiler developers sufficient language support and eliminate the two-phase compilation strategy of popular compiler development tools to create more meaningful compile- and run-time error messages and allow for debugging.

### 2. Implementation & Design
You will find that almost all of my changes reside at the bottom of Javac's parser and inside a new package called "grammar", both to be found in Javac's *com.sun.tools.javac.parser* package. The additions to the parser parse the new construct and the *GrammarToClassConverter* class, then, converts the parsed intermediate representation into a *JCClassDecl* AST node which then can be compiled without further changes necessary. You can find the definition of a grammar structure in Appendix I.
The grammar for CMM, called "CMMGrammar", can be found in:
*lab_4\b97902122\compiler\src\edu\ntu\compilers\lab4\cmmgrammar\CMMGrammar.java*

The resulting class is located in:
*lab_4*\b97902122\project\cmmsrc\edu\ntu\compilers\lab4\cmmgrammar\CMMGrammar.java

I added four new tokens: *grammar*, *grammarpasses*, *grammarmembers* and *grammarnode* to Javac. The developer can choose a name for the grammar and must provide the type of Tokens and the enum for Token names. Next follows an optional set of passes: A return type followed by a name that then may be implemented by individual rules. Passes will be converted to methods and are different from normal methods in two ways: First, they will be propagated automatically through the AST and secondly, they are grouped together by name and not by signature: It is important to support different parameters for pass implementations in different nodes to allow for inherited attributes – For example variable declarations in CMM are in a list that follows their type. We want to pass down that type to all variable declaration nodes.

The grammar will be converted into the following set of classes:

- **Grammar**: The container of all other classes, all NonTerminal and Terminal instances, as well as some utility methods.
- **Symbol:** Base class for the **Terminal** and **NonTerminal** classes.
- **Terminal**: The occurrence of every terminal in any rule of the grammar will yield one instance of the Terminal class. The name of every terminal must be a member of the specified token name enum, else a meaningful compiler error will be raised. All terminal instances are

collectively put into the *Terminals* array inside the **Grammar** class, indexed by their rolling id.
- **NonTerminal**: The definition of any NonTerminal, contains all of it's rules. The grammar holds one instance of all symbols and, in addition, an array of all NonTerminals, indexed by their rolling id.
- **Rule**: The definition of any rule, contains all of it's symbols and the *newChild* method used by parsers.
- **AST**: An almost empty class to store the root. Can be used later to hold more relevant information.
- **Node**: The base class for all AST nodes. All nodes have an index, a Parent and two methods to get/set their children.
- **TerminalNode**: All found terminals can be used by the parser to create an instance of this node that contains the Token and the Terminal.
- One node class per NonTerminal: Every such class is abstract and inherits from Node. It also holds a set of developer-given members, such as methods and attributes that are shared by all nodes, that inherit from it. It also defines an abstract method for each pass declaration. If any rule provided parameters for any pass, all of this NonTerminal's rule's implementations of that pass must have the same parameters.
- One node class per production Rule: These classes are all final and inherit from their corresponding NonTerminal's node class. The Rule AST node classes represent the nodes of the AST that will be automatically created, using the *newChild* methods. They contain an optional set of custom members, and the actual pass implementations. All passes that are not implemented explicitly, will automatically be propagated to it's children. Every pass implementation that does not explicitly propagate the pass to it's children, can call the propagate_<passname> method to do so. That method does not exist if any children's implementations of that pass have parameters. And, if in that case, the pass is not implemented explicitly, a warning will be issued, to avoid that some passes will not propagate through the entire tree.
  Every rule has a finite set of children, which are public attributes with the names $1, $2, $3 etc. The type for Terminals is TerminalNode, and for NonTerminals it is their NonTerminal's AST node class.
  Every rule can also be optionally named - For example CMM's WhileStmt has it's own name so we can save the loops in a central stack to correctly generate code for the break and continue statements.

The exact construction of the grammar can be found in the well-documented code of the GrammarToClassConverter's *genBody* method which is located in:
`lab_4\b97902122\javac\com\sun\tools\javac\parser\grammar\GrammarToClassConverter.java`

## 3. Future Work

More powerful grammar representation by employing grammar-rewriting algorithms

A more powerful grammar representation would make the grammar more readable and eliminate sources for bugs. The following features should be included:

- Support BNF
- Automatically eliminate left-recursion and common prefixes

- Use operator precedence input to resolve ambiguities

Make grammars generic
As I mentioned in Section 1, any parser for this kind of grammar can be grammar-agnostic, and thus universal, however, in practice that is not quite possible yet, since the generated classes of different grammars do not have common base classes (for Grammar, Terminal, NonTerminal, Node etc.). Originally I was working on an implementation that requires every program with a grammar construct to be delivered with a small run-time library that delivers all those base classes, but I had to let that go due to time constraints. As a consequence, the CMM parser is actually working against CMMGrammar, instead of a generic grammar base class, even though, none of what it does is CMM-specific.

Allow customization of AST creation
Currently there is no way to customize the creation of the AST, which makes it quite tricky to accommodate for certain language features that are difficult to handle by parsers, such as the dangling else and C-style array initialization. It should eventually be possible to hook to the creation of nodes, so one can invoke the scanner and other tools to easily work-around that sort of issue.

Named symbols
The name of symbol occurrences in rules could be used to identify the children of rule AST nodes rather than having to use meaningless numbers $1, $2, $3, etc.

Reliably detect pass propagation (or lack thereof)
When doing the first test runs with my CMM compiler, I spent almost an hour fixing bugs related to missing explicit pass propagations. Assuming that pass propagation only happens within a pass itself, it would not be hard to automatically detect that. However, even in the CMM compiler, that is not the case: The print statement needs to generate some code before and after loading the variable to be printed, so the printable statement will be generated from the Gen class.

Support grammars to be spread out over multiple files
Most compilers have one file or directory to cover one pass, naturally a grammar should also be able to be distributed over multiple files to accommodate for that: One file has the actual definitions and other files have implementations of some passes of one or more rules.

Support explicit pass implementations on NonTerminal level
It could prove useful to define passes that are the same for all rules of one NonTerminal that do not explicitly define their own.

Attach javadoc to source
Currently, the comments written inside a grammar definition are not attached to the resulting AST nodes, so no javadoc will be available inside an IDE when operating on a grammar.

## 4. Changes from initial proposal

I decided that inlining and accumulators would not support arbitrary grammar structures. Inlining grammar symbols and using accumulators to provide lists of symbols are just quick&dirty substitutions

of a more powerful grammar structure representation, such as BNF.

**5. IDE support**

I would argue that most Java developers are developing their programs inside a powerful IDE. If one changes the language in a way that no IDE would recognize it, it would be almost impossible to convince anyone of the usefulness of the feature. I myself have written the CMM compiler in IntelliJ IDEA by Jetbrains. In order to do so, I simply generated a source file from the AST that represents the grammar. Of course, that goes against one of the goals of the projects, which was to eliminate two-phase compilation, but it simply was required due to insufficient IDE support at this point. In the future, I might write addons to support the grammar construct in the popular IDEs.
Of course, when compiling with the modified version of Javac, the compiler errors and run-time-generated stack-traces are entirely meaningful and always point to the right spot inside the grammar definition.

**6. A CMM compiler**

I initially wrote almost 1000 lines of the class that is supposed to represent the CMMGrammar by hand, to be able to start developing the compiler, while figuring out what kind of features it would require. After I wrote about half of the compiler, I settled for the final layout of a grammar that was rarely modified thereafter. Then I made the additions to the Javac's parser, implemented the *GrammarToClassConverter*, and finally finished and debugged the compiler. Once, all bugs in the grammar construction were squished, debugging and fixing bugs inside the compiler itself was only a matter of several hours. The CMMGrammar definition contains about 720 lines, whereas the CMMGrammar class that results from it, has more than 5300 – without any comments or documentation!
That shows that the new construct eliminates almost all of the boilerplate code, leaving less than 30% of the code to be written to the developer. Of course, the grammar itself, only describes the structure of the language, as well as the traversal logic for preparation and code generation, but the remaining classes (excluding the scanner), only contain relatively little code, of which the Parser and Gen classes are the biggest, with 265 and 300 lines, respectively.

**7. CMM compiler performance**

I did not address one very important feature of compilers: Performance. Gladly, performance is not a weak spot of this new approach. This is some performance information of the CMMCompiler when compiling three different CMM programs on my Windows desktop machine:

T1A_Hello.cmm:
Compilation took 1.192 seconds: 1.179 init (98.91 %), 0.012 parsing (1.01 %), 0.001 code gen (0.08 %)

T6A_Scope.cmm:
Compilation took 1.201 seconds: 1.178 init (98.08 %), 0.019 parsing (1.58 %), 0.004 code gen (0.33 %)

T7F_mxm.cmm:
Compilation took 1.239 seconds: 1.171 init (94.51 %), 0.041 parsing (3.31 %), 0.027 code gen (2.18 %)

One second is of course not an acceptable time frame for the compilation of such simple programs. Luckily, that one second is not caused by the parser or the way a grammar is implemented. You can see that initialization always takes up almost all of the time, and is constant, as well. That is due to my generic scanner which first generates an NFA and then a DFA from a given set of token descriptors. Parsing and code generation are quite fast. However, I cannot explain why, unlike code generation, parsing execution time is not proportional to code input size. It is not a coincidence either, seeing how I ran the programs several times and they always yielded about the same results.

## 8. Conclusion

The new grammar structure is a powerful tool for compiler developers. It produces all AST classes, and allows for simple and fast propagation of different compiler passes on a context-free grammar definition. It is still missing some features but most of them can be quite easily added.
On the downside, the tight coupling between the AST and grammar structure, makes it quite difficult to implement multi-language compilers with exchangeable front- and back-ends, such as GCC.
But if I feel quite confident, that with the changes, proposed in section 3, one can relatively easily implement compilers for languages, as complex as Java or C#.

## Part B – A bug fix in Javac

When I looked at Javac the first time in March, I downloaded the version "openjdk-6-src-b22-28_feb_2011" from http://download.java.net/openjdk/jdk6/. It is also the version of Javac that I used for Part A.
The download link is:
http://download.java.net/openjdk/jdk6/promoted/b22/openjdk-6-src-b22-28_feb_2011.tar.gz

That version had a bug that I fixed:

- I imported langtools/javac/ into my IntelliJ IDEA IDE, and compiled it
- I then used it to compile it a simple HelloWorld program, which worked fine
- Then I used it to compile itself, which resulted into a NullPointerException in Lower.java
- I narrowed down the source of the problem to enums not compiling anymore
- The Lower class is used to add statements that are assumed to be present by the programmer, to keep the code shorter and more consistent
- Enums are one of many things that need quite a bit of synthesized code before they can be compiled
- After a bit of tracing, I found that the problem was with enum constructors
- The Lower class adds 2 additional parameters and arguments to every constructor definition and invocation: The name and ordinal of each enum member
- Those arguments are then passed to any super(...) or this(...) call of all enums in order to, eventually, invoke the underlying Enum(String, int) constructor
- The Lower class visits the super(...) and this(...) invocations in the Lower.visitApply() method
- It tried to box arguments of the invocation in the Lower.boxArgs method
- Lower.boxArgs(...) iterates over all parameters of the method/ctor and assumes the invocation to have at least as many arguments, without verification

- The underlying constructors already have the parameters but the name and ordinal arguments have not been added to the invocation of the constructors yet (they are added just a little bit later)
- Since there are less arguments than parameters, the boxArgs method tries to read beyond the end of the argument list and throws a NullPointerException
- I fixed the problem by removing the two artifical parameters from the list of parameters given to the boxArgs method, since they don't require any boxing
- You can find the old and new versions of Lower.java, as well as their diff patch under: `lab_4\b97902122\project\bugfix\`
- I also appended the patch file in *Appendix III*

**Final Words**

This concludes my adventure into the exiting world of compilers for now. I would like to thank the reader for not falling asleep and, specifically, my lecturer, Mr. Chen, for giving me the opportunity to work on this very interesting project!

Note, that if you want to run all tests without having to wait for javac and the compiler to compile for every program, simply run the following three supplied shell scripts (I did not invest the time, to let make take care of that):

```
lab_4/b97902122/compileJavac.sh
lab_4/b97902122/compileCompiler.sh
lab_4/b97902122/testall.sh
```

# Appendix

## Appendix I - Definition of the new grammar construct

Note: I borrowed quantifiers from regular expressions, specifically: * (0 or more), ? (0 or 1) and + (at least 1)

```
Grammar :
      "grammar" Name "<" JavaType "," JavaType ">" "{" GrammarBody "}"
    ;

Name :
      JavaId
    ;

GrammarBody :
      PassDeclarations? MemberDeclarations? BaseNode? NonTerminal+
    ;

PassDeclarations :
      "grammarpasses" "{" PassDeclaration* "}"
    ;

BaseNode :
      "grammarnode" Name ImplementingInterfaces MemberDeclarationBlock
```

```
    ;

NonTerminal :
      Name MemberDeclarations? ":" Rules? ";"
    ;

PassDeclaration :
      JavaType Name ";"
    ;

MemberDeclarations :
      "grammarmembers" ImplementingInterfaces MemberDeclarationBlock
    ;

ImplementingInterfaces  :
      "implements" JavaInterfaceType*
    |
    ;

MemberDeclarationBlock:
      "{" JavaClassMemberDeclaration* "}"
    ;

Rules :
      Rule SeparatedRule*
    ;

SeparatedRule:
      "|" Rule
    ;

Rule :
      RuleName? Symbol* MemberDeclarations? Pass*
    ;

RuleName :
      "[" Name "]"
    ;

Symbol :
       JavaId
    ;

Pass :
      "("Name JavaParameterList")" JavaMethodBlock
    ;
```

Javac helps us with the following symbols:

JavaId
Java-style identifier.

JavaStringLiteral
Java-style string literal - e.g. "this is a \"string literal\"".

JavaType
Any run-time type - e.g. int, PrintStream etc.

JavaInterfaceType
Any run-time interface type; i.e. any type defined with the "interface" keyword.

JavaClassMemberDeclaration
Anything that is allowed within a class declaration block; can be a field (attribute), method, nested type etc.

JavaParameterList
A comma-separated list of parameters to a method. Every parameter has a type and a name.

JavaMethodBlock
A block of code, enclosed by "{" and "}" braces.

## Appendix II – Directory structure

To accommodate the directory structure required by lab4, you can find all files inside:

```
lab_4/b97902122/
      – javac/ - Modified version of Javac
      – compiler/src/ - The CMM compiler
      – project/ - Files related to this report
```

## Appendix III – Bug fix patch

```
2506,2509d2505
<
< // Domi edit:
<         Name methName = TreeInfo.name(tree.meth);
<
2511,2524c2507,2509
<         if (meth.name==names.init) {
<             if (allowEnums &&   // allowEnums is false even if enums are allowed
<                     meth.owner == syms.enumSym) {
<                 argtypes = argtypes.tail.tail;
<             }
<             else {
<                 // remove name and ordinal parameters from nested enum ctor calls,
<                         // because they have not been added to the argument list yet, and
do not require boxing anyway
<                 Symbol constructor = accessConstructor(tree.pos(), meth);
<                 ClassSymbol c = (ClassSymbol)constructor.owner;
<                 if ((c.flags_field&ENUM) != 0 || c.getQualifiedName() ==
names.java_lang_Enum) {
<                     int argCount = tree.args.size() + (tree.varargsElement != null ? 1 :
0);
<                     if (argCount == argtypes.size() - 2) {
<                                     // only remove parameter info if argument count does
not match parameter count
---
>         if (allowEnums &&
>             meth.name==names.init &&
>             meth.owner == syms.enumSym)
2526,2529d2510
<                     }
<                 }
```

薛德明
B97902122

```
<             }
<         }
2532c2513
<
---
>         Name methName = TreeInfo.name(tree.meth);
2604,2607d2584
< // Domi edit:
<         if (varargsElement == null && parameters.length() != args.length()) {
<             throw new AssertionError(args);
<         }
2630,2632c2607
<             if (args.length() != 1) {
<                 throw new AssertionError(args);
<             }
---
>             if (args.length() != 1) throw new AssertionError(args);
```