Simplifying extensions of KOOL using Labeled BNF

Replacing the lexer and parser with generated ones

Compiler Construction '15 Final Report

Daniel Chavez Erik Norell
KTH
{dace,eriknore}@kth.se

1. Introduction

The compiler project consists of two parts, where this report is mostly about the second part. In the first part of the project we created a compiler for the KOOL language, where we have implemented

1. a lexer, tokenizing an input given a language specification
2. a parser, building an abstract syntax tree (AST) from the tokens
3. name analysis and type checking of the code
4. a JVM code generator making the code runnable using Java

The second part of the project is an extension of the compiler where we have chosen to replace our lexer and parser with a generated lexer and parser using a program called BNF Converter[1] (BNFC). The replacement of our lexer and parser might not be an *extension* per definition, the user of the compiler can barely tell and there are no new features for the user - however we expect some advantages none the less.

Lexer and parser generators are commonly used tools in compiler construction, however there is still an educational purpose in writing your own lexer and parser. Manually writing lexers and parsers is usually more error-prone and require some extra work to include position information and error reporting [Ranta 2012].

The intended difference of our compiler before and after the project is a more compact and flexible lexer and parser. The short `Kool.cf` file provides an easy way to extend or shorten the KOOL language and it is also self documenting (the file alone can be used as a language specification, or BNFC can be used to generate a specification written in e.g. LaTeX or HTML).

2. Background

BNFC produces a program which does lexing and parsing of a file given a so called Labeled BNF (LBNF) specification of a language. BNFC creates this program using a lexer, more specifically Alex (Haskell), Jex (Java) or Flex (C), and a parser, Happy (Haskell), CUP (Java) or Bison (C++, C or Java). We chose the lexer and parser written in Haskell for this extension.

[1] `http://bnfc.digitalgrammars.com/`

2.1 LBNF

The BNFC expects a grammar written in Labelled BNF grammar (LBNF). The KOOL specification is written in BNF grammar which has to be converted to LBNF. The main difference between BNF and LBNF is that each rule in BNF is also given a label in LBNF. In turn, each label will be a node in the AST[Forsberg and Ranta 2005].

A well known rule from the KOOL BNF grammar is the addition expression:

```
Expr ::= Expr + Expr
```

And its equivalence in LBNF:

```
EPlus. Expr ::= Expr "+" Expr1
```

Labels are followed by a dot and terminals are quoted, the label in this case was chosen to match the name of the ExprTree Plus-object from the labs. Note the digit on the nonterminal `Expr1`, it gives precedence to the next operand i.e. the next operand might be an arbitrary expression and it might need parentheses. The LBNF syntax is slightly wordier but the advantages are clear, we have taken care of precedence and labels for the AST.

A less trivial example is the method definition in KOOL, where we could have none or more arguments, variable declarations and statements. These are denoted by brackets below, and `[Arg]` for example, could be empty or non-empty in which case it is separated by its separator:

```
MDecl. Method ::=  "def" Ident "(" [Arg] ")" ":" Type "=" "{"
                       [Var]
                       [Stmnt]
                       "return" Expr ";"
                   "}" ;
separator Arg "," ;
```

The definition of an `Arg` is straightforward, however there is no argument node in the AST (an argument is a Formal inside the MethodDecl object), in that case we give it a meaningful label but we won't really use it:

```
MArgs. Arg ::= Ident ":" Type ;
```

The full LBNF grammar for the KOOL language can be found in the Appendix below.

2.2 Alex the lexer generator

Lexer generators produce lexers given regular expressions that describe the language. Alex[2] creates a DFA with the regular expressions from the LBNF grammar and produces an efficient lexer. It also uses the maximal munch rule, i.e. trying to match the longest possible token.

2.3 Happy the parser generator

Parser generators produce syntax analyzers given a grammar. In the case of Happy[3], it generates a parser given annotated BNF. The old parser was a recursive descent parser i.e. LL(1), while its replacement Happy is a LALR(1) parser (lookahead 1, left-to-right parsing, rightmost derivations). This fact has no consequences outside of the parser except that only having one lookahead makes us modify the grammar in order to achieve

[2] `https://www.haskell.org/alex/`

[3] `https://www.haskell.org/happy/`

optional statements. For example, the else-statement is optional and having one lookahead gives us a shift-reduce conflict:

```
SIfElse. Stmnt  ::= "if" "(" Expr ")" Stmnt "else" Stmnt ;
SIfEmp.  Stmnt  ::= "if" "(" Expr ")" Stmnt ;
```

After parsing the statements in the if-body we could either shift (read the else-token) or reduce (pop the stack i.e. the second rule). This is not really a problem since Happy always shifts on shift-reduce conflicts, which means the dangling-else would always apply to the closest if-statement.

2.4 BNF Converter

We did not have to interact with the lexer and parser directly. BNFC provides a high-level approach to indirectly interact with Alex and Happy, the BNFC itself can be seen as a generator of generators. This compilation process of lexing and parsing before and after our project is shown in Figure 1 below.

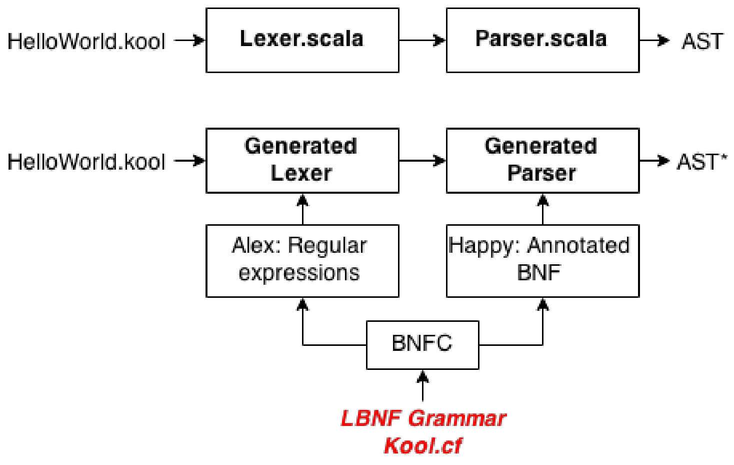

Figure 1: *The lexing and parsing process before and after our project.*

The red text in Figure 1 denotes the part written by us. `Kool.cf` is a labelled BNF grammar (LBNF) similar to the BNF grammar given in lab 2 and the AST* produced by BNFC requires some tweaking to produce the same AST from the first part of the project (see 3.2 for further details).

3. Implementation

For this implementation a correct LBNF grammar was defined in `Kool.cf` and a converter was written in Scala to transform the BNFC output to the Tree structure defined in `Trees.scala` and used in the rest of the front-end `NameAnalysis.scala`, `TypeChecking.scala` and in the back-end `CodeGeneration.scala`. From within Scala the converter executes the BNFC parser, checks if parsing was successful and builds the tree by parsing the BNFC output which is printed on `stdout`. If parsing was not successful the error message from BNFC is printed on `stderr` and the execution of the compiler is aborted.

3.1 KOOL grammar

As already mentioned BNFC requires the language specification to be written in LBNF. The original language specification for the KOOL language is written in BNF[4], so the rewritten specification in LBNF is similar to

[4] The specification can be found at `http://www.csc.kth.se/~phaller/compilers/labs/labs2.html`

the original one (see Appendix). Precedence is handled as expected with `expr7` having the strongest binding, determining the structure because of precedence is similar to what was done in `Parser.scala`. It looks easier and more straight forward using the LBNF notation however this is not entirely the case as the same amount of consideration and understanding of precedence is needed to get the labels right and avoid shift/reduce conflicts - it took a little time to get this exactly right. Other than that it was a matter of learning the syntax expected by BNFC.

3.2 AST converter

As we previously mentioned the BNFC outputs an AST. However the format is different than the one used in the original lexer and parser, meaning we had to write a converter which read the output from `stdout` and generated a tree as defined in `Trees.scala`. The differences is seen in the following example

```
Expression:    answer = calc.query(a, 2);

current AST:   Assign(Identifier(answer),MethodCall(Identifier(calc),
               Identifier(query),List(Identifier(a),IntLit(2))))

BNFC AST:      SAssign (EIdent "answer") (EMethodCall (EIdent "calc")
               (EIdent "query") [EIdent "a",EIntLit 2])
```

As can be seen these are rather similar and converting this AST from BNFC in a string format to the AST used in the rest of the front-end is a matter of matching keywords and consuming white space and parentheses.

3.3 Problems encountered

Although we did encounter some problems during the implementation these were in most cases manageable after some reading and debugging. The last two problems however have implications on the use of the compiler, to fix these we would have to make changes to the BNFC source code which is out of the scope for this extension.

3.3.1 Positions of nodes

To be able to output detailed information about errors identified in the later stages by name analysis and type checking, the position of each node is needed and it is not given in the BNFC AST by default. To get the position of each node an extra definition of each token is needed with the keyword `position` (see Appendix for examples). However it does not *only* produce the position

```
Expression:       answer = calc.query(a, 2);

BNFC AST:         SAssign (EIdent "answer") (EMethodCall (EIdent "calc")
                  (EIdent "query") [EIdent "a",EIntLit 2])

with positions:   SAssign (PosIdent ((3,9),"answer")) (EMethodCall (EIdent (PosIdent
                  ((3,18),"calculator"))) (PosIdent ((3,29),"query")) [EIdent (PosIdent
                  ((3,35),"factor")), EIntLit (PosInteger ((3,43),"2"))])
```

The output is polluted with extra labels which are not needed to convert the output to the AST used in the rest of the program. It is still a matter of consuming white space, parentheses and now discarding unneeded tags. It required some extra effort to write the converter but did not really make it harder. It can be found in `src/main/scala/koolc/utils/AstConverter.scala`, it is not included in the Appendix as it is more or less multiple cases of String matching in Scala - not particularly interesting

3.3.2 Unexpected tokens

The `main`-function is always required in the main object of any KOOL-file, this function takes no arguments. A natural way to tokenize this part of a file is therefore

```
MainObject. Main ::= ... "def main()" ...
```

However to get information on the position we had to do some special tricks (as already explained in Section 3.3.1), but also allow for space between `main` and `()`, we ended up with the following definition

```
MainObject. Main ::= ... "def" PosMain "()" ...
```

Naturally this implies that the underlying lexer (`Alex`) creates a token for both `def` and `()`. What we didn't understand at first was that this caused some errors when tokenizing a method declaration, defined as

```
MDecl. Method ::= PosDef PosIdent "(" [Arg] ")" ... ;
position token PosDef {"def"} ;
```

As explained earlier the `PosDef` token is needed to get the position of the declaration node. The token created for `def` was always matched instead of `PosDef` and for a method without arguments the token from matching the main function `()` was matched instead of the tokens `(`, `[arg]` (although empty) and `)`, this is because of the maximal-munch rule. In retrospect this seems perfectly natural, however the error messages from BNFC are rather limited (see 3.3.3 for more info) so it was not self-evident what caused the error. It was finally solved by using the same tokens as in the method declaration (note that `PosDef` is not needed in `main` and is therefore simply discarded)

```
MainObject. Main ::= ... PosDef PosMain "(" ")" ...
```

3.3.3 Limited error feedback

With BNFC we cannot output the same errors as with our own lexer and parser. In fact we forward the errors detected and output by BNFC to the user. The problem with this is the quality of the error messages from BNFC. Errors in the old lexer contained exact position of an unknown token and what the token was while BNFC is not that specific outputting a list of all tokens found and the same message for all types of lexer errors

Error: `answer = calc.query(_a, 2);` (_a not a valid token)

Old:
```
file:3:29: Fatal: Invalid character: _
 answer = calc.query(_a, 2);
                     ^
```

BNFC:
```
Tokens: [PT (Pn 0 1 1) (T_PosObject "object"), ...
syntax error at line 3 due to lexer error
```

It is the same case with parser error although the BFNC error message is different and at least somewhat more informative

Error: `answer = calc.query(a 2);` (forgotten ',' between arguments)

Old:
```
file:3:31: Fatal: expected: COMMA, found: INT(2)
 answer = calc.query(a 2);
                         ^
```

BNFC:
```
Tokens: [PT (Pn 0 1 1) (T_PosObject "object"), ...
syntax error at line 3 before 2 ) ; }
```

These limited error outputs are simply forwarded as error messages to users using the compiler with BNFC. To produce more informative errors from BNFC we would have to make changes to the source code of BNFC.

3.3.4 Parsing error found in BNFC

Testing the new lexer and parser we found it did not detect a type of error which the old lexer and parser successfully detected

```
def foo(bar: Int, ): Int { ... }
```

The BNFC generated parser accepts the extra ',' without complaints. It does not affect the rest of the compiler but the code contains errors none the less. We traced the reason for this error to be a consequence of how `separator` is implemented in BNFC [Forsberg and Ranta 2005, Section 7.1]. The keyword "`separator Arg "," ;`" in conjunction with "`[Arg]`" is syntactic sugar for

```
[].    [Arg] ::= ;
(:[]). [Arg] ::= Arg ;
(:).   [Arg] ::= Arg "," [Arg] ;
```

This clearly shows that `","` followed by an empty list is correct according to the grammatic specification. Ranta and Forsberg are aware of this:

> *"This might be considered a bug, but a set of rules forbidding the terminating semicolon would be much more complicated."*
>
> [Forsberg and Ranta 2005, p. 7, in the report the example is `Stmnt` separated with `";"`, a semicolon.)].

4. Testing the new front-end: adding the modulo operator

To test our hypothesis that it would be easier to extend the KOOL language with BNFC we implemented the modulo operator (%). A detailed change log is available in the Appendix. This operator is fairly similar to multiplication and division so it was mostly a matter of copy pasting.

It was somewhat easier to add the modulo operator using the new lexer and parser, although having to make changes to (and thereby needing to know the code within) `AstConverter.scala` did make it harder than we had hoped or thought initially. Already knowing `Lexer.scala` and `Parser.scala` it was fairly easy to add the operator, however compared to `Kool.cf` and `AstConverter.scala` we had to make changes in three files at five different places compared with two files in three places. Of course adding an operator requires changes in the rest of the compiler as well and in this case these changes more or less canceled the advantage of using BNFC.

To test that our extension was successful we made changes to one of the supplied test files `GCD.kool`. It defined a method called `modulo` which performed modulo calculations using a `while`-loop, we simply replaced this with the modulo operator and compared comilations of both - they were identical. This file can be found at `testprograms/extension/GCD_mod.kool`.

5. Conclusions

Our expectation was that it would be easier to extend the KOOL language with a generated lexer and parser by converting the language specification written in BNF to Labeled BNF and using the program BNF Converter. We found that this might not be the case as extending the language implies making changes to the other parts as well, and the potentially saved work from using BNFC instead of making changes to the original lexer and parser is negligible compared with the rest of the work. We also found other negative sides to using BNFC: less informative error feedback from lexer and parser errors and that the generated parser fail to detect certain types of errors.

It is our opinion that even if this was not the case there is a clear educational benefit from writing the lexer and parser yourself, especially gaining a greater understanding of abstract syntax trees and how these are affected by rules of precedence.

6. Possible Extensions

Extending this project further by using generated parsers and lexers is not really possible. One can try other generators, perhaps comparing with writing code for a lexer generator (e.g Alex, Jlex or Flex) and parser generator (e.g. Happy, CUP or Bison) directly. Although we feel it would be more interesting to extend the compiler in other directions, given more time we would look further into preprocessing: catching programmer errors (e.g. unreachable code) and making optimizations before code generation.

References

M. Forsberg and A. Ranta. The Labelled BNF grammar formalism. *Department of Computing Science, Chalmers University of Technology and the University of Gothenburg*, 2005.

A. Ranta. *Implementing Programming Languages*. College Publications, 1st edition, 2012.

7. Appendix

7.1 Kool.cf - *LBNF definition of the KOOL language*

```
Program. PDef  ::= Main [Class];

comment "//" ;
comment "/*" "*/";

MainObject. Main  ::= PosObject PosIdent "{" PosDef "main" "(" ")" ":" "Unit"
       "=" "{" [Stmnt] "}" "}" ;

-- Class Declaration
separator Class "" ;
CDecl. Class        ::= PosClass PosIdent Extends "{" [Var] [Method] "}" ;
   EExt. Extends    ::= "extends" PosIdent ;
   EEmp. Extends    ::= ;

-- Variable Declaration
separator Var "";
VDecl. Var ::= PosVar PosIdent ":" Type ";" ;

-- Method Declaration
separator Method "";
MDecl. Method ::= PosDef PosIdent "(" [Arg] ")" ":" Type "=" "{"
                     [Var]
                     [Stmnt]
                     "return" Expr ";"
                  "}" ;
   separator Arg "," ;
   MArgs. Arg   ::= PosIdent ":" Type ;

-- Types
TIntArray. Type ::= PosTIntArray ;
TInt. Type      ::= PosTInt ;
TBoolean. Type  ::= PosTBool ;
TString. Type   ::= PosTString ;
TIdent. Type    ::= PosIdent ;

-- Statements
separator Stmnt "" ;
SBlock. Stmnt        ::= "{" [Stmnt] "}";
SIfElse. Stmnt       ::= PosIf "(" Expr ")" Stmnt "else" Stmnt ;
SIfEmp. Stmnt        ::= PosIf "(" Expr ")" Stmnt ;
SWhile. Stmnt        ::= PosWhile "(" Expr ")" Stmnt ;
SPrintln. Stmnt      ::= PosPrintln "(" Expr ")" ";" ;
SAssign. Stmnt       ::= PosIdent "=" Expr ";" ;
SArrayAssign. Stmnt  ::= PosIdent "[" Expr "]" "=" Expr ";" ;
```

```
-- Expressions
-- Parentheses automagically handled using "coercions Expr 7 ;"
separator Expr "," ;

EMod.      Expr5 ::= Expr5  PosMod      Expr6 ;
ETimes.    Expr5 ::= Expr5  PosTimes    Expr6 ;
EDiv.      Expr5 ::= Expr5  PosDiv      Expr6 ;
EPlus.     Expr4 ::= Expr4  PosPlus     Expr5 ;
EMinus.    Expr4 ::= Expr4  PosMinus    Expr5 ;
ELessThan. Expr3 ::= Expr3  PosLessThan Expr4 ;
EEquals.   Expr2 ::= Expr2  PosEquals   Expr3 ;
EAnd.      Expr1 ::= Expr1  PosAnd      Expr2 ;
EOr.       Expr  ::= Expr   PosOr       Expr1 ;

ENew.          Expr6 ::= PosNew PosIdent "(" ")" ;
ENewIntArray.  Expr6 ::= PosNewIntArr "[" Expr "]" ;
EMethodCall.   Expr6 ::= Expr6 "." PosIdent "(" [Expr] ")" ;
EArrayLength.  Expr6 ::= Expr6 ".length" ;
EArrayRead.    Expr6 ::= Expr6 "[" Expr "]" ;
ENot.          Expr6 ::= PosNot Expr6 ;

EThis.       Expr7  ::= PosThis ;
EIntLit.     Expr7  ::= PosInteger ;
EStringLit.  Expr7  ::= PosString ;
EBoolTrue.   Expr7  ::= PosTrue ;
EBoolFalse.  Expr7  ::= PosFalse ;
EIdent.      Expr7  ::= PosIdent ;

coercions Expr 7 ;

-- Mixed Keywords
position token PosObject {"object"} ;
position token PosClass {"class"} ;
position token PosVar {"var"} ;
position token PosDef {"def"} ;

position token PosPrintln {"println"} ;
position token PosWhile {"while"} ;
position token PosIf {"if"} ;

-- Types
position token PosTIntArray {"Int[]"} ;
position token PosTInt {"Int"} ;
position token PosTBool {"Bool"} ;
position token PosTString {"String"} ;
```

```
-- Expressions
position token PosNew {"new"} ;
position token PosNewIntArr {"new Int"} ;
position token PosTrue {"true"} ;
position token PosFalse {"false"} ;
position token PosThis {"this"} ;
position token PosNot ('!') ;
position token PosPlus ('+') ;
position token PosMinus ('-') ;
position token PosTimes ('*') ;
position token PosDiv ('/') ;
position token PosMod ('%') ;
position token PosLessThan ('<') ;
position token PosEquals {"=="} ;
position token PosAnd {"&&"} ;
position token PosOr {"||"} ;
position token PosInteger (digit (digit)*) ;
position token PosString ('"' (char - '"')* '"') ;

-- keep at the bottom trying all cases from top to bottom, this regex cathes e.g. new
position token PosIdent (letter (letter | digit | '_')*) ;
```

7.2 Modulo operator implementation - *change log*

```
-------------------------------------
----------------BNFC-----------------
-------------------------------------
Kool.cf (added two rows):

49:  EMod.  Expr5 ::= Expr5  PosMod   Expr6 ;
100: position token PosMod ('%') ;

AstConverter.scala (copy-pase Times, change to Mod):

398: case "Mod" =>
399:           val lhs = readExprWithParen
400:           ensureNextIs(' ')
401:           val t = extractNode
402:           ensureNextIs(' ')
403:           val rhs = readExprWithParen
404:           Mod(lhs, rhs).setPos(f, t.getPos)

-------------------------------------
-----------------OLD-----------------
-------------------------------------
Tokens.scala (added row):

54: case object MOD extends TokenKind

Lexer.scala (added two rows):

129: case '%' => true      (in isSingleSpecialChar)
186: "%"     ->    new Token(MOD))

Parser.scala (added "c == MOD"):

430: def currentIsFollower = {
431:   val c = currentToken.kind
432:   (c == AND || c == OR || c == EQUALS || c == LESSTHAN || c == PLUS || c ==
433:       MINUS || c == TIMES || c == MOD || c == DIV || c == LBRACKET || c == DOT)
434: }

329: case MOD => eat(MOD); Mod(expr, atom).setPos(file, pos)
```

```
-------------------------------------
---------------BOTH------------------
-------------------------------------
```

Trees.scala (added row):

```
39: case class Mod(lhs: ExprTree, rhs: ExprTree) extends ExprTree
```

Printer.scala (added row):

```
237: case Mod(lhs: ExprTree, rhs: ExprTree) => add("%", lhs, rhs)
```

NameAnalysis.scala (copy-paste Times and change to Mod):

```
413: case Mod(lhs, rhs) => handleExpression(sym, lhs); handleExpression(sym, rhs)
```

TypeChecking.scala (copy-paste Times and change to Mod):

```
89: case Mod(lhs: ExprTree, rhs: ExprTree) =>
90:          mathematical(lhs, rhs, TInt)
91:          expr.setType(TInt)
92:          TInt
```

CodeGeneration.scala (copy-paste Times and change to Mod and IREM):

```
282: case Mod(lhs, rhs) =>
283:          generateExprCode(ch, lhs)
284:          generateExprCode(ch, rhs)
285:          ch << IREM
```