

**Explicitly Recursive Grammar
Combinators
The Implementation of some Grammar
Algorithms
Technical report**

Dominique Devriese Frank Piessens

Report CW 594, 3 September 2010



**Katholieke Universiteit Leuven
Department of Computer Science**

Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

**Explicitly Recursive Grammar
Combinators
The Implementation of some Grammar
Algorithms
Technical report**

Dominique Devriese Frank Piessens

Report CW 594, 3 September 2010

Department of Computer Science, K.U.Leuven

Abstract

In a companion paper, we have presented an alternative representation of grammars in Haskell, explicitizing the grammar's recursion, decoupling the grammar from its semantic actions and making semantic actions independent of matching order. In this technical report, we present the implementation of some interesting grammar algorithms in order to provide evidence for the increased power and declarative style of our novel grammar representation. The presented code is a simplification of parts of our freely available Haskell library `grammar-combinators`.

Explicitly Recursive Grammar Combinators

The Implementation of some Grammar Algorithms

Technical report

Dominique Devriese and Frank Piessens

K.U.Leuven

{dominique.devriese,frank.piessens}@cs.kuleuven.be

Abstract. In a companion paper [1], we have presented an alternative representation of grammars in Haskell, explicitizing the grammar’s recursion, decoupling the grammar from its semantic actions and making semantic actions independent of matching order. In this technical report, we present the implementation of some interesting grammar algorithms in order to provide evidence for the increased power and declarative style of our novel grammar representation. The presented code is a simplification of parts of our freely available Haskell library `grammar-combinators`¹.

1 Introduction

This technical report accompanies a paper in which we propose a novel grammar representation in a functional parsing library, based on an explicit representation of recursion in the grammar [1]. This technical report should not be read independently from that paper and throughout this text we do not make any attempt to repeat or even summarize its contents.

The goal of this text is to demonstrate the implementation of grammar algorithms using the grammar representation presented in the companion paper. We demonstrate that the representation provides enough power to implement advanced grammar algorithms in a readable and declarative way. To that end, we demonstrate the implementation of the following commonly useful algorithms.

printGrammar Pretty-print a grammar definition in an (E)BNF like formalism.

foldLoops Convert a context-free grammar defined using quantifying reference operators from the *LoopProductionRule* type class to a normal context-free grammar by replacing calls to $\langle idx \rangle^*$ with normal references $\langle idx^* \rangle$ to newly introduced Kleene-star versions of non-terminals idx^* , and defining appropriate production rules for the added non-terminals.

transformLeftCorner Apply the well-known left-corner transform [2] to convert a left-recursive grammar to an equivalent non-left-recursive grammar. We show that, contrary to other work, our implementation uses a declarative style.

¹ <http://projects.haskell.org/grammar-combinators>

unfoldRecursion Convert a context-free grammar back into an infinite tree style representation, useful for compatibility with traditional parser combinator libraries.

parseUU Parse a string according to a given grammar using Swierstra and Duponcheel’s UUParse library [3].

2 Some more infrastructure

Before we start looking at the algorithms, we define some infrastructure functions and types, generalising and abstracting those in the companion paper. We will provide Haskell definitions for regular, context-free and extended context-free rules and grammars (general, abstract and processing), and general semantic processors, and how a semantically polymorphic grammar can be combined with a concrete semantic processor into a *processing grammar*. The definitions in the companion paper are less general but otherwise equivalent to the definitions in this text.

2.1 Grammar types

We define three types of production rules: regular, context-free and extended context-free. The difference lies in the requirements for the production rule interpretation types p that they are defined for: all production rule typesgrammars can use the operations from *ProductionRule* and *CharProductionRule* type classes, context-free rules can additionally use the recursion operator $\langle \cdot \rangle$ from *RecProductionRule*, and extended production rules can additionally use the quantified reference operators $\langle \cdot \rangle^*$ and $\langle \cdot \rangle^+$ from *LoopProductionRule*.

```

type RegularRule  $\phi$   $r$   $v$  =
   $\forall p . (ProductionRule\ p, CharProductionRule\ p) \Rightarrow p\ v$ 
type ContextFreeRule  $\phi$   $r$   $v$  =
   $\forall p . (ProductionRule\ p, CharProductionRule\ p,$ 
     $RecProductionRule\ p\ \phi\ r) \Rightarrow p\ v$ 
type ExtendedContextFreeRule  $\phi$   $r$   $v$  =
   $\forall p . (ProductionRule\ p, CharProductionRule\ p,$ 
     $RecProductionRule\ p\ \phi\ r, LoopProductionRule\ p\ \phi\ r) \Rightarrow p\ v$ 

```

Grammars come in three flavours: general, abstract and processing. The difference lies in the semantic value family that production rules expect for recursive references to non-terminals and the semantic value family that they produce themselves. All three grammar types are parametrised over production rule type rt and domain ϕ .

```

type GGrammar  $rt$   $\phi$   $r$   $rr$  =  $\forall ix . \phi\ ix \rightarrow rt\ \phi\ r\ (rr\ ix)$ 
type AGrammar  $rt$   $\phi$  =  $\forall r . GGrammar\ rt\ \phi\ r\ (PF\ \phi\ r)$ 
type PGrammar  $rt$   $\phi$   $r$  =  $GGrammar\ rt\ \phi\ r\ r$ 

```

A general grammar is parametrised over both families (respectively r and rr). An abstract grammar is forall-quantified over any family r and produces values of the domain's pattern functor with r as subtree representation functor (see the discussion of semantic value family polymorphism in the companion paper). A processing grammar is a grammar in which a single semantic value family is used for both recursive references and production rule results. Such grammars are typically expected by for example parsing algorithms, since they need to pass on previously computed semantic values to subsequent matches. For context-free rules, we get the following types of context-free grammars.

```

type GContextFreeGrammar  $\phi$   $r$   $rr$  = GGrammar ContextFreeRule  $\phi$   $r$   $rr$ 
type ContextFreeGrammar  $\phi$  = AGrammar ContextFreeRule  $\phi$   $t$ 
type PContextFreeGrammar  $\phi$   $r$  = PGrammar ContextFreeRule  $\phi$   $r$ 

```

We omit corresponding definitions for *regular* and *extended context-free* grammars.

2.2 Applying semantics to a grammar

Processing grammars are commonly the result of applying a concrete semantic processor to an abstract grammar. We define a processor as a function that transforms a value of one semantic value family into the corresponding value in another, for all non-terminals in the domain. The following *applyProcessor* function allows us to apply such a semantic processor to a context-free grammar. The *applyProcessor* function is more general, but applying a *Processor* to a *ContextFreeGrammar* will yield a *PContextFreeGrammar*.

```

type GProcessor  $\phi$   $r$   $r'$  =  $\forall ix . \phi$   $ix$   $\rightarrow$   $r$   $ix$   $\rightarrow$   $r'$   $ix$ 
type Processor  $\phi$   $r$  = GProcessor  $\phi$  (PF  $\phi$   $r$ )  $r$ 
applyProcessor :: GProcessor  $\phi$   $r'$   $r''$   $\rightarrow$ 
  GContextFreeGrammar  $\phi$   $r$   $r'$   $\rightarrow$  GContextFreeGrammar  $\phi$   $r$   $r''$ 
applyProcessor proc  $g$   $idx$  = proc  $idx$   $\ggg$   $g$   $idx$ 

```

We omit corresponding *applyProcessorR* and *applyProcessorE* functions, applying a processor to regular and extended context free grammars respectively. These separate functions are needed to help the compiler deduce correct types for the resulting grammars. An example use of these function is the application of the *calcArith* semantic processor to the *grammarArith* grammar, both from this text's companion paper. We call the combined grammar *calcGrammarArith*.

```

calcGrammarArith = applyProcessorE grammarArith calcArith

```

Our algorithms will typically use a grammar by instantiating the production rules for a concrete production rule interpretation type p , to extract the information they need from the grammar's production rules. They will mostly have a meaningful rank-2 type like the following, which specifies that the *foldLoops* function (see section 3.2) turns a processing extended context-free grammar over

domain ϕ with semantic value family r into a processing normal context-free grammar over an extended domain $\phi_{\#}$ and a correspondingly adapted semantic value family $r_{\#}$.

```
foldLoops ::
  PExtendedContextFreeGrammar  $\phi$   $r$   $\rightarrow$  PContextFreeGrammar  $\phi_{\#}$   $r_{\#}$ 
```

Note that such a type is of rank 2, and we need the RankNTypes (or Rank2Types) GHC extension [4] to use it.

3 The algorithms

So, let's see take our machinery for a spin. One of the most evident things that is impossible to achieve with traditional parser combinator libraries is pretty-printing a grammar into a textual representation.

3.1 Printing grammars

The implementation of such a pretty-printing algorithm is in fact almost trivial in our framework, but it is instructive as a first demonstration of how to work with our grammars. Furthermore, as a first test bed, it will also indicate some further infrastructure we need to put in place.

The actual grammar printing algorithm in our `grammar-combinators` library is complicated by the desire to avoid spurious parentheses and epsilons in the result, and to support infinite grammars through a limitation of the recursion depth. This part of the code is uninteresting, so we do not show or discuss it. Instead, we show a simplified version demonstrating only its essential workings.

To calculate a textual representation of a single production rule, we use a custom production rule interpretation type `PrintProductionRule`, which just keeps a `String` representation of the rule. It needs to carry the domain type ϕ and semantic value family r along in its type because of the functional dependencies of the production rule interpretation type classes.

```
newtype PrintProductionRule ( $\phi :: * \rightarrow *$ ) ( $r :: * \rightarrow *$ )  $v =$ 
  IPP {printIPP :: String }
```

The `ProductionRule` operations are implemented by simply creating the proper `String` representation for the result.

```
instance ProductionRule (PrintProductionRule  $\phi$   $r$ ) where
   $\epsilon$ [ $v$ ] = IPP "epsilon"
  die = IPP "die"
  endOfInput = IPP "EOI"
   $a$  |||  $b$  = IPP $ "(" ++ printIPP  $a$  ++ " | " ++ printIPP  $b$  ++ ")"
   $a$  >>>  $b$  = IPP $ printIPP  $a$  ++ " " ++ printIPP  $b$ 
```

```
instance CharProductionRule (PrintProductionRule  $\phi$   $r$ ) where
  token  $t$  = IPP $ show  $t$ 
```

For the *RecProductionRule* instance, we need to know how to represent a non-terminal as a *String*. We therefore require our domain ϕ to be an instance of a new type class called *ShowFam*, telling us how to convert a domain proof term into a *String*.

```
class ShowFam  $\phi$  where
  showIdx ::  $\forall ix$  .  $\phi$   $ix$   $\rightarrow$  String
instance (ShowFam  $\phi$ )  $\Rightarrow$ 
  RecProductionRule (PrintProductionRule  $\phi$   $r$ )  $\phi$   $r$  where
  <math>idx</math> = IPP $ "<" ++ showIdx  $idx$  ++ ">"
```

Given this interpretation for production rules, we are ready to implement our algorithm. We first define how to print the production rules for a single non-terminal:

```
printNT :: (ShowFam  $\phi$ )  $\Rightarrow$ 
  GContextFreeGrammar  $\phi$   $r$   $rr$   $\rightarrow$   $\phi$   $ix$   $\rightarrow$  String
printNT gram  $idx$  = "<" ++ showIdx  $idx$  ++ ">" ++
  " ::= " ++ printIPP (gram  $idx$ )
```

To print a full grammar, all that is left to do, is to consecutively apply this *printNT* function to all non-terminals in a grammar. To do this, we again need information from the domain, and we again define this as a general requirement in the *FoldFam* type class. Its *foldFam* function allows us to fold a given function over all non-terminals in a given domain.

```
class FoldFam  $\phi$  where
  foldFam :: ( $\forall ix$  .  $\phi$   $ix$   $\rightarrow$   $b$   $\rightarrow$   $b$ )  $\rightarrow$   $b$   $\rightarrow$   $b$ 
printGrammar ::  $\forall \phi$   $r$   $rr$  . (FoldFam  $\phi$ , ShowFam  $\phi$ )  $\Rightarrow$ 
  GContextFreeGrammar  $\phi$   $r$   $rr$   $\rightarrow$  String
printGrammar gram = foldFam ((+ "\n" ++) . printNT gram) ""
```

The *FoldFam* and *ShowFam* type classes actually express quite general requirements for a given domain, and they will indeed be useful in other grammar algorithms as well. Their instances for our domain ϕ_{arith} are trivial:

```
instance ShowFam  $\phi_{arith}$  where showIdx Line = "Line"
  showIdx Expr = "Expr"
  showIdx Term = "Term"
  showIdx Factor = "Factor"
  showIdx Digit = "Digit"
instance FoldFam  $\phi_{arith}$  where
  foldFam  $f$   $n$  =  $f$  Line $  $f$  Expr $  $f$  Term $  $f$  Factor $  $f$  Digit  $n$ 
```

```
ghci> putStr $ printGrammar grammarArith
```

```
<Line> ::= <Expr> EOI
<Expr> ::= <Term> | (<Expr> '+' <Term>)
<Term> ::= <Factor> | (<Term> '*' <Factor>)
<Factor> ::= <Digit>+ | ('(' <Expr> ')')
<Digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Fig. 1: Printing out an (E)BNF-like representation of the arithmetic expressions grammar with the library grammar printing algorithm (manually reformatted). A simplified version of the printing algorithm code is discussed in section 3.1.

Figure 1 now shows our library performing a first useful task: printing an (E)BNF-like representation of the arithmetic expressions grammar. Note that this is what we get with the full printing algorithm, the simplified one above works but produces more spurious epsilons and parentheses.

3.2 Production Rule Origami

In this text’s companion paper, we have made use of the $\langle \cdot \rangle^+$ combinator in the definition of the *grammarArith* grammar, without going into the details of its definition. It is defined as follows in the *LoopProductionRule* type class.

```
class (ProductionRule p, RecProductionRule p  $\phi$  r)  $\Rightarrow$ 
  LoopProductionRule p  $\phi$  r | p  $\rightarrow \phi$ , p  $\rightarrow r$  where
   $\langle \cdot \rangle^*$  ::  $\phi \text{ } ix \rightarrow p [r \text{ } ix]$ 
   $\langle \cdot \rangle^+$  ::  $\phi \text{ } ix \rightarrow p [r \text{ } ix]$ 
```

The type class provides a restricted version of the traditional (ω -regular style) *many* operator, written $\langle \cdot \rangle^*$, quantifying references to non-terminals using a “zero or more” modifier. The operator $\langle \cdot \rangle^+$ represents an analogous “one or more” quantified reference. The *LoopProductionRule* type class has the same arguments ϕ and r as the *RecProductionRule* type class, and they play the same role.

We think these restricted versions of the *many* and *some* operators are still general enough for most purposes (since any other production rule to be quantified can be represented by an additional non-terminal in the grammar), but we can implement these in a better way.

There are two options available for implementing these operations. The first solution is to *unfold* the loops, essentially defining $\langle idx \rangle^* \equiv \text{many } \langle idx \rangle$ with *many* the traditional quantification combinator from libraries like Parsec [5]. We have actually implemented this transformation in a function called *unfoldLoops*, to allow the use of our grammars with traditional parser combinator libraries.

For situations where we want to avoid going back to the “infinite tree” style of parsers, an alternative solution exists.

This second solution is based on the traditional modelling of these operations in context-free grammars, through a technique that we refer to as *folding* loops. What we will do for a call to $\langle idx \rangle^*$ for some non-terminal idx is replace it with a normal reference $\langle idx^* \rangle$ to a new non-terminal idx^* , for which we define suitable (finite) production rules in the transformed grammar.

We will capture this grammar transformation in a general algorithm called *foldLoops*. The algorithm works by introducing new non-terminal types ix^1 and ix^* , parameterised over an underlying non-terminal type ix . The new non-terminal ix^1 represents the unmodified base non-terminal ix , and ix^* its quantified Kleene- $*$ -variant. We then introduce a GADT $\phi_{\#}$, parameterised over an underlying domain ϕ . The type $\phi_{\#}$ represents a new domain containing proof terms for all ix^1 and ix^* where ix is a non-terminal in the underlying domain ϕ . We do not require values for the ix^1 and ix^* types, so we define them using the `EmptyDataTypes` GHC Haskell extension.

```
data  $\cdot^1$   $ix$ 
data  $\cdot^*$   $ix$ 
data  $\cdot_{\#}$   $\phi$   $ix$  where
   $\cdot^1$   $:: \phi$   $ix \rightarrow \phi_{\#}$   $ix^1$ 
   $\cdot^*$   $:: \phi$   $ix \rightarrow \phi_{\#}$   $ix^*$ 
```

All necessary type classes (like *FoldFam* and *ShowFam*, and others which we haven’t encountered yet) can be implemented for this new domain. These instances are defined similarly to the following *FoldFam* instance, which simply uses the underlying domain ϕ ’s *foldFam* function to iterate over both types of non-terminals in the domain $\phi_{\#}$.

```
instance (FoldFam  $\phi$ )  $\Rightarrow$  FoldFam  $\phi_{\#}$  where
  foldFam  $f$   $n = foldFam$  ( $f$   $\cdot^*$ ) $ foldFam ( $f$   $\cdot^1$ )  $n$ 
```

For representing semantic values for the new domain, we introduce a semantic value family adapter $r_{\#}$, parameterised over an underlying semantic value family r . As you might expect, $r_{\#}$ wraps a value of type r ix for the new non-terminal ix^1 and a value of type $[r$ $ix]$ for the non-terminal ix^* .

```
data family  $\cdot_{\#}$  ( $r :: * \rightarrow *$ )  $ix$ 
newtype instance  $\cdot_{\#}$   $r$   $ix^1 = FLV^1$  { unFLV $^1$   $:: r$   $ix$  }
newtype instance  $\cdot_{\#}$   $r$   $ix^* = FLV^*$  { unFLV $^*$   $:: [r$   $ix]$  }

consFLV  $:: r_{\#}$   $ix^1 \rightarrow r_{\#}$   $ix^* \rightarrow r_{\#}$   $ix^*$ 
consFLV (FLV $^1$   $v$ ) (FLV $^*$   $vs$ ) = FLV $^*$  ( $v : vs$ )
```

Our *foldLoops* algorithm turns an extended context-free grammar over a domain ϕ into the equivalent non-extended context-free grammar over the larger domain $\phi_{\#}$. This leads to the following type signature:

```

foldLoops ::
  PExtendedContextFreeGrammar  $\phi$   $r$   $\rightarrow$ 
  PContextFreeGrammar  $\phi_{\#}$   $r_{\#}$ 

```

We define the transformed grammar in a declarative way, by defining what the production rules are for both types of non-terminals in domain $\phi_{\#}$. The production rules for a idx^* non-terminal are easy to define. Such a non-terminal must either be the corresponding base non-terminal idx^1 followed by another instance of non-terminal idx^* itself, or it must be empty. In both cases, we make sure to produce the correct semantic value.

```

foldLoops bgram  $idx^*$  = consFLV  $\S$   $\langle idx^1 \rangle \gg \langle idx^* \rangle$ 
  |||  $\epsilon$ [FLV* []]

```

The production rules for a base non-terminal idx^1 are obtained by taking the production rules of the unmodified grammar and replacing all references to $\langle idx \rangle^*$ with calls to $\langle idx^* \rangle$. We perform this substitution by instantiating the original grammar's production rules with a special production rule interpretation type *FLW*. The type *FLW* implements a production rule for the original context-free grammar over domain ϕ , in function of an underlying production rule for the transformed context-free grammar over the extended domain $\phi_{\#}$. The type classes *ProductionRule*, *CharProductionRule* are implemented by just passing the call through to the underlying production rules and wrapping/unwrapping the results as appropriate (not shown for brevity). The *RecProductionRule* instance transforms a reference $\langle idx \rangle$ into a reference $\langle idx^1 \rangle$ and the *LoopProductionRule* instance transforms a quantified reference $\langle idx \rangle^*$ into the desired normal reference $\langle idx^* \rangle$.

```

data FLW  $p$  unused1 unused2  $\phi$   $r$   $v$  where
  FLW ::  $p$   $v$   $\rightarrow$  FLW  $p$   $\phi_{\#}$   $r_{\#}$   $\phi$   $r$   $v$ 
  unFLW :: FLW  $p$   $\phi_{\#}$   $r_{\#}$   $\phi$   $r$   $v$   $\rightarrow$   $p$   $v$ 
  unFLW (FLW  $p$ ) =  $p$ 
instance (ProductionRule  $p$ , RecProductionRule  $p$   $\phi_{\#}$   $r_{\#}$ )  $\Rightarrow$ 
  RecProductionRule (FLW  $p$   $\phi_{\#}$   $r_{\#}$   $\phi$   $r$ )  $\phi$   $r$  where
   $\langle idx \rangle$  = FLW  $\$$  unFLV1  $\S$   $\langle idx^1 \rangle$ 
instance (ProductionRule  $p$ , RecProductionRule  $p$   $\phi_{\#}$   $r_{\#}$ )  $\Rightarrow$ 
  LoopProductionRule (FLW  $p$   $\phi_{\#}$   $r_{\#}$   $\phi$   $r$ )  $\phi$   $r$  where
   $\langle idx \rangle^*$  = FLW  $\$$  unFLV*  $\S$   $\langle idx^* \rangle$ 

```

Note how we add two dummy parameters *unused1* and *unused2* to the *FLW* type to avoid the need for GHC's UndecidableInstances extension. You can think of this as an explanation to the type checker of the reason why our type instances are decidable, but we don't go into details for this technical trick.

We can then define the production rules in the transformed grammar for non-terminals idx^1 by unwrapping the *FLW* production rule interpretation type.

```

foldLoops bgram  $idx^1$  = FLV1  $\S$  unFLW (bgram  $idx$ )

```

The above is in fact all that is required to perform the transformation of extended context-free grammars into normal context-free grammars. Note the very declarative nature of this function’s implementation. Our definitions of new non-terminals in function of the old ones avoids the need for generation of fresh identifiers or other stateful techniques.

Our *foldLoops* function only applies to *PContextFreeGrammars*, i.e. grammars which have already been combined with a concrete semantic processor. This simplifies the presentation of our algorithm, but it makes it less general than we would like. In the `grammar-combinators` library, we have a more complicated version of *foldLoops* which does not have this limitation.

In Figure 2, we show the result of applying the *foldLoops* algorithm to the arithmetic expressions grammar. We omit the base rules, since they are almost identical to the ones in Figure 1.

```
ghci> putStr $ printGrammar (foldLoops grammarArith)

<Line*> ::= (<Line> <Line*>) | epsilon
<Expr*> ::= (<Expr> <Expr*>) | epsilon
<Term*> ::= (<Term> <Term*>) | epsilon
<Factor*> ::= (<Factor> <Factor*>) | epsilon
<Digit*> ::= (<Digit> <Digit*>) | epsilon
...
```

Fig. 2: A printed version of the added production rules for \cdot^* non-terminals added by the *foldLoops* algorithm. The \cdot^1 production rules are identical to Figure 1.

3.3 The Left-Corner Transform, declaratively...

The *foldLoops* implementation in the previous section demonstrates that our representation of context-free grammars makes it possible to implement an (admittedly simple) automatic grammar transformation in a declarative and general way. We can define new non-terminals and custom production rules, either depending or not on the original grammar’s production rules. With this power, it is tempting to try and implement more involved transformations. In this section, we take a look at a transformation intended to solve one of the traditional problems associated with parser combinator libraries: their inability to properly handle left-recursion.

The problem is directly apparent in the example grammar in this text’s companion paper. The `Expr` and `Term` non-terminals have production rules directly referencing themselves in the left-most position. This is an instance of *direct* left-recursion. The grammar does not feature *indirect* left-recursion, where a circular

relation between more than one non-terminal exists, each having a production rule that refers to the next in the left-most position.

Top-down parser algorithms such as the one employed by UUParse, Parsec [5] are not able to handle direct or indirect left-recursion. A workaround solution exists, based on limiting the left-recursion depth with a memoized counter during parsing [6], but this technique seems unsuited (because of its complexity) for many applications (in particular parsing of large unambiguous input with relatively simple grammars, as is typical in for example compilers).

Better solutions exist based on grammar transformations. Baars and Swierstra have previously shown how to implement one such transformation (the *left-corner transform*) with their representation of grammars [7]. They implemented the left-corner transform as defined by Moore [2], which removes direct and indirect left recursion in a given grammar such that the size of the transformed grammar is asymptotically cubic in the size of the original grammar [8].

The implementation of this non-trivial transformation takes about 220 lines of code in our framework. Here, we take a look at the most important parts of this implementation. The first thing we do is define an extended domain ϕ_{lc} for a given domain ϕ . This is similar to what we did in the *foldLoops* implementation, but for the left-corner transform, we define three types of non-terminals: for given non-terminals a and b and a given character t , our new domain ϕ_{lc} has non-terminals a^1 (representing base non-terminal a), $a -_{NT} b$ (that will match the remainder of a non-terminal a when a non-terminal b has already been matched), $a -_t t$ (that will match the remainder of a non-terminal a when a character t has been matched).

```

data  $\cdot^1 ix$ 
data  $(\cdot -_{NT} \cdot) ix' ix$ 
data  $(\cdot -_t) ix$ 
data  $\cdot_{lc} \phi ix$  where  $\cdot^1$        $:: \phi ix \rightarrow \phi_{lc} ix^1$ 
                     $\cdot -_{NT} \cdot$   $:: \phi ix' \rightarrow \phi ix \rightarrow \phi_{lc} (ix -_{NT} ix')$ 
                     $\cdot -_t \cdot$      $:: Char \rightarrow \phi ix \rightarrow \phi_{lc} (ix -_t)$ 

```

For a semantic value family r for the underlying domain ϕ , we then define a new semantic value family r_{lc} for our new domain ϕ_{lc} , with appropriate semantic values for the newly introduced non-terminals. For example, a non-terminal $a -_{NT} b$ represents the remainder of a non-terminal a starting with a non-terminal b that has already been parsed, so its semantic value will be of type $r b \rightarrow r a$. This means that its result is a function that will return the semantic value of non-terminal a when given the already parsed value of non-terminal b .

```

data family  $\cdot_{lc} (r :: * \rightarrow *) ix$ 
newtype instance  $r_{lc} ix^1 = LCV_1 \{ unLCV_1 :: r ix \}$ 
newtype instance  $r_{lc} (ix -_{NT} ix') = LCV_{-_{NT}} \{ unLCV_{-_{NT}} :: r ix' \rightarrow r ix \}$ 
newtype instance  $r_{lc} (ix -_t) = LCV_{-_t} \{ unLCV_{-_t} :: Char \rightarrow r ix \}$ 

```

We use the production rule interpretation type *TransformLCRule* to interpret production rules in the given grammar. For a given production rule, we keep

track of four different interpretations of it. Under interpretation *tlcEmpty*, we keep track of whether the production rule can match the empty string and if so what value it will produce. Under *tlcFull*, we keep an unmodified version of the original production rule. Under *tlcNTMinNT*, we keep the original production rule with leading references to a given base non-terminal removed (or, if no such leading reference is present, a never-matching *die* rule) and under *tlcNTMinT*, we have the same thing for a given leading character.

Like the *FLW* type from section 3.2, the *TransformLCRule* type has two unused parameters *unused1* and *unused2* that again only serve to avoid the UndecidableInstances GHC Haskell extension.

```
data TransformLCRule p (unused1 :: * → *) (unused2 :: * → *) ϕ r v =
  MkTLCIR {
    tlcEmpty :: Maybe v,
    tlcFull :: p v,
    tlcNTMinNT :: ∀ ix' . ϕ ix' → p (r ix' → v),
    tlcNTMinT :: Char → p (Char → v)
  }
```

We do not show the instances for *ProductionRule* and *CharProductionRule* type classes for brevity. In the *ProductionRule* instance, we only need to make sure to properly handle empty and non-empty left hand sides in the sequencing operator (to make sure we properly detect *leading* tokens and references). In the *CharProductionRule* instance, we interpret a call to *token tt* specially under the *tlcNTMinT* interpretation, replacing it with an $\epsilon[id]$ rule, that will simply pass through the already matched token.

Something very interesting happens in the *RecProductionRule* instance. Under the *tlcNTMinNT* interpretation of the base production rule (where we assume the current rule needs to consume a given already matched non-terminal), we need to interpret a call to a base non-terminal $\langle idx \rangle$ as succeeding and simply passing through the already matched non-terminal, but clearly only if the already matched non-terminal is the requested non-terminal *idx*. Otherwise, the *tlcNTMinNT* interpretation must fail. To do this in a well typed way, we use the function *overrideIdx*, defined in the following type class:

```
class EqFam ϕ where
  overrideIdx :: (∀ ix . ϕ ix → r ix) → ϕ oix → r oix → (∀ ix . ϕ ix → r ix)
```

The *overrideIdx* function is a general tool that allows us to override a polymorphic function over a domain ϕ for one of the non-terminals ϕoix in a well-typed way. We need to instantiate it for all of our domains, but the instantiations are boilerplate:

```
instance EqFam ϕarith where
  overrideIdx f Line v Line = v
  overrideIdx f Expr v Expr = v
  overrideIdx f Term v Term = v
  overrideIdx f Factor v Factor = v
```

$$\begin{aligned} \text{overrideIdx } f \text{ Digit } v \text{ Digit} &= v \\ \text{overrideIdx } f \text{ } _ \text{ } \text{idx} &= f \text{ idx} \end{aligned}$$

With this `overrideIdx` function, the `RecProductionRule` instance above defines the `tlcNTMinNT` interpretation of an underlying production rule as a function that will fail for all non-terminals except for the requested non-terminal, in which case it is an empty rule that will pass through the given value in its result. A technical problem is that the `overrideIdx` function requires the result type of the overridden function to be directly parametric in the non-terminal type `ix` and to achieve this, we need to wrap and unwrap the returned rules in a wrapper type `WrapNTMinNTP`. The other interpretations are easily defined: the `tlcEmpty` interpretation is set to `Nothing` as a non-terminal reference is not an empty rule, for `tlcFull`, we simply get a reference $\langle \text{idx}^1 \rangle$ to the new non-terminal representing base non-terminal `idx`, and for `tlcNTMinT`, we get a failure rule `die` for all characters.

```
instance (ProductionRule p, RecProductionRule p  $\phi_{lc}$   $r_{lc}$ , EqFam  $\phi$ )  $\Rightarrow$ 
  RecProductionRule (TransformLCRule p  $\phi_{lc}$   $r_{lc}$   $\phi$  r)  $\phi$  r where
   $\langle \text{idx} \rangle = \text{let } rNTMinNT \text{ idxm} = \text{unWNMNP } \$$ 
     $\text{overrideIdx } (\_ \rightarrow \text{WNMNP } \text{die}) \text{ idx } (\text{WNMNP } \epsilon [\text{id}]) \text{ idxm}$ 
  in MkTLCIR {tlcEmpty = Nothing,
    tlcFull = unLCV1  $\ggg$   $\langle \text{idx}^1 \rangle$ ,
    tlcNTMinNT = rNTMinNT,
    tlcNTMinT = const die}

newtype WrapNTMinNTP p r ix surrIx =
  WNMNP {unWNMNP :: p (r surrIx  $\rightarrow$  r ix)}
```

Having defined the different interpretations for the base production rules, we can proceed to the actual transformation of the grammar. These are defined by the function `transformLeftCorner`, which takes a given grammar and transforms it into another grammar. It is restricted to processing grammars because the left-corner transform inherently mixes interpreted rules and standard template rules, making it difficult to work with non-processing grammars.

$$\begin{aligned} \text{transformLeftCorner} &:: (\text{Domain } \phi) \Rightarrow \\ &P\text{ContextFreeGrammar } \phi \text{ r} \rightarrow P\text{ContextFreeGrammar } \phi_{lc} \text{ } r_{lc} \end{aligned}$$

Because in our `transformLeftCorner` transformation, we require information about the FIRST sets of the non-terminals [9, pp.188–189], we make use of a `calcFirst` algorithm, which we don't discuss further. This is a general algorithm that is also useful outside of the left-corner transformation. With this extra information, we call another function `transformLeftCorner'` which will generate the actual production rules for our new non-terminals.

$$\begin{aligned} \text{transformLeftCorner}' &:: \\ &\forall p \phi \text{ r ix} . (\text{FoldFam } \phi, \text{EqFam } \phi, \text{ProductionRule } p, \\ &\text{CharProductionRule } p, \text{RecProductionRule } p \phi_{lc} \text{ } r_{lc}) \Rightarrow \end{aligned}$$

$$\begin{aligned}
& (\forall ix . \phi ix \rightarrow TransformLCRule p \phi_{lc} r_{lc} \phi r (r ix)) \rightarrow \\
& (\forall ix . \phi ix \rightarrow FirstSet) \rightarrow \phi_{lc} ix \rightarrow p r_{lc} ix \\
transformLeftCorner gram idx = \\
& transformLeftCorner' gram (calcFirst gram) idx
\end{aligned}$$

For brevity, we only show the production rules for non-terminals of the form idx^1 and $idx -_t t$. The production rules for non-terminals idx^1 all follow the same template. They first expect to see one of the tokens of the first set of the non-terminal idx and then pass on the work to the non-terminal $idx -_t t$, properly wrapping and unwrapping values along the way.

$$\begin{aligned}
transformLeftCorner' bgram calcFirst idx^1 = \\
\mathbf{let} \ ruleT \ tt = flip (\$) \gg token \ tt \gg (unLCV._t. \gg \langle idx -_t tt \rangle) \\
\mathbf{in} \ LCV_1 \gg Set.fold (|||) . ruleT \ die (firstSet \$ calcFirst idx)
\end{aligned}$$

The production rules for non-terminals of the form $idx -_t t$ come in two forms. This is because the non-terminal idx can start with character t in two ways. Either one of the original production rules for the non-terminal idx starts with character t directly, and in that case, the remainder of that production rule becomes the production rule for $idx -_t t$. This remainder of the original production rule is precisely what is represented by its interpretation under $tlcNTMinT t$.

The other possibility is that a production rule of idx starts with a reference to another non-terminal $idxB$, and that non-terminal has a production rule that directly starts with character t . We can capture this by defining a production rule for non-terminal $idx -_t t$ that starts with the remainder of the production rules for non-terminal $idxB$ starting with character t (which we again get using that production rule's interpretation under $tlcNTMinT$) and then references non-terminal $idx -_{NT} idxB$. Because non-terminal $idx -_{NT} idxB$ represents the remainder of a base non-terminal idx after a non-terminal $idxB$ has been matched, its production rules will properly match the remainder of non-terminal idx .

$$\begin{aligned}
transformLeftCorner' bgram _ (idx -_t t) = \\
\mathbf{let} \ bMinT \ idxB = flip (.) \gg tlcNTMinT (bgram idxB) t \gg \\
\quad (unLCV._{NT}. \gg \langle idx -_{NT} idxB \rangle) \\
\quad bMinTs = foldFam (|||) . bMinT \ die \\
\mathbf{in} \ \ LCV._t. \gg bMinTs \\
\quad ||| \ LCV._t. \gg tlcNTMinT (bgram idx) t
\end{aligned}$$

Note that we don't actually check that non-terminal $idxB$ actually starts with the character t , nor that there is any situation in which a match for non-terminal idx can start with a match for non-terminal $idxB$ (i.e. that $idxB$ is a left corner of idx). These would both be worthwhile optimisations, but they are not necessary, because in those cases, subsequent parts of the production rule in question become *die* rules and will be removed by subsequent postprocessing steps (dead-branch removal and dead non-terminal unfolding).

The above is actually a very declarative specification of the left-corner transform. New non-terminals are first identified and named, and their production rules are directly specified, using interpretations of the production rules in the original grammar, avoiding the need for fresh identifier generation. Both the parsing literature [2, 8], as Baars and Swierstra [7] more imperative specifications of the transformation are used, typically an algorithm that iterates over the production rules in the original grammar, generating new production rules and fresh non-terminals in the process. Such a presentation is in our opinion less clear, and more difficult to reason about. We have thoroughly checked but not formally proven the correspondence of our declarative specification to Moore’s imperative specification [2] (which we based ourselves on).

Figure 3 shows a selected part of the resulting grammar after applying the left-corner transformation to the arithmetic expressions grammar.

```
ghci> putStr $ printReachableGrammar (filterDiesE
      (transformLeftCornerE calcGrammarArith)) $ Expr1

(...)
<Expr> ::= ('(' <Expr-'>) | ('0' <Expr-'0'>) | ('1' <Expr-'1'>) |
      ('2' <Expr-'2'>) | ('3' <Expr-'3'>) | ('4' <Expr-'4'>) |
      ('5' <Expr-'5'>) | ('6' <Expr-'6'>) | ('7' <Expr-'7'>) |
      ('8' <Expr-'8'>) | ('9' <Expr-'9'>)
<Expr-'> ::= <Expr> ')' <Expr-Factor>
<Expr-Factor> ::= <Expr-Term>
<Expr-Term> ::= (('*' <Factor>) <Expr-Term>) | <Expr-Expr> | epsilon
<Expr-Expr> ::= '+' <Term> (<Expr-Expr> | epsilon) | (EOI <Expr-Line>)
<Expr-Line> ::= die
<Expr-'9'> ::= <Expr-Digit>
<Expr-Digit> ::= <Digit>* <Expr-Factor>
(...)
```

Fig. 3: Some rules from the printed version of the arithmetic expressions grammar after applying the left-corner transform and dead-branch removal. Output reformatted, reordered and selected.

We have actually also implemented another type of (in-)direct left-recursion removal transformation which we call *transformUniformPaull*. It is a variant of the traditional Paull algorithm [9, p. 177], adapted so that it is independent from any chosen non-terminal ordering, but instead behaves as if a topologically sorted ordering for some spanning tree was used for every non-terminal separately. This transformation can cause an exponential increase in grammar size, but has the advantage that it is easy to make it fall back to a simpler direct left-recursion removal for non-indirectly left-recursive non-terminals. The result of applying this transformation to the arithmetic expressions grammar is shown in

Figure 4. With some other post-processing transformations (dead rule removal, dead non-terminal removal, chain non-terminal removal), the reachable part of the transformed grammar corresponds perfectly to the manually transformed UUParse grammar in the companion paper.

```

ghci> putStr $ printReachableGrammar
      (unfoldChainNTsE (filterDiesE (unfoldDeadE
      (transformUniformPaullE calcGrammarArith)))) $ Line_1

<Line> ::= <Line_head>
<Line_head> ::= <Expr> EOI
<Expr> ::= <Term> <Expr_tail>*
<Expr_tail> ::= '+' <Term>
<Term> ::= <Factor_head> <Term_tail>*
<Term_tail> ::= '*' <Factor_head>
<Factor_head> ::= <Digit_head>+ | '(' (<Expr> ')')
<Digit_head> ::= '0' | '1' | '2' | ... | '8' | '9'

```

Fig. 4: The result of applying the *transformUniformPaull* transformation to the arithmetic expressions grammar, as well as some simple postprocessing (filtering dead branches, unfolding dead non-terminals, chain recursion removal).

3.4 Limiting recursion depth

The *EqFam* type class and *overrideIdx* function introduced in the previous section are actually more widely important than just for our left corner transform implementation. In many algorithms, it is also useful to replace calls to $\langle idx \rangle$ with the actual grammar production rules for *idx*, but with previously unfolded recursive references in those rules replaced with failure rules. This can be achieved using the *overrideIdx* function and even factored out into a general purpose algorithm, but we do not go further into it in this text.

4 Unfolding recursion and loops

This text would not be complete if we didn't demonstrate that we can actually use our grammars to parse. Possibly the least effort strategy to implement this is to reuse a pre-existing parser combinator library. In order to do this, what we need to do is to unfold our grammar definitions back into an “infinite tree” form. We can abstract this process into a general algorithm called *unfoldRecursion*.

$$\textit{unfoldRecursion} :: P\textit{ContextFreeGrammar} \phi r \rightarrow P\textit{RegularGrammar} \phi r$$

To implement this algorithm, we define a production rule interpretation *URW* that will return an underlying production rule $p v$ when given production rules for all non-terminals in the grammar. We omit *ProductionRule* and *CharProductionRule* instances, which simply pass down all operations to the underlying production rule. In the *RecProductionRule* instance, we replace the call $\langle idx \rangle$ with the received production rule for non-terminal idx . The *unfoldRecursion* process then recursively passes its own result as the production rules to be used for other non-terminals. A given *PContextFreeGrammar* is thus transformed into an “infinite tree”, non-recursive *PRegularGrammar*.

```

data URW p ϕ ixT r v =
  URW { unURW :: (∀ ix . ϕ ix → p (r ix)) → p v }
instance (ProductionRule p) ⇒
  RecProductionRule (URW p ϕ ixT r) ϕ r where
  ⟨idx⟩ = URW $ λg → g idx
  unfoldRecursion gram idx =
    unRPWRule (gram idx) $ unfoldRecursion gram

```

Note that after using the *unfoldRecursion* algorithm, the remaining production rules are in fact completely independent from each other. Every production rule now represents the full definition of the non-terminal it represents, similar to how any UUParse parser completely defines how to match a non-terminal.

We omit the definition of a very similar algorithm *unfoldLoops*, in which we define the unrolling of quantified references ($\langle \cdot \rangle^*$ and $\langle \cdot \rangle^+$) into their standard infinite tree representations.

4.1 Are we parsing yet?

All that we still need to start parsing is a compatibility component for existing parser combinator libraries. We take the UUParse library [3] as an example, because its combinators are closest to ours (except for recursion and looping combinators). The definition of the *parseUU* function uses the *unfoldRecursion* algorithm from the previous section to transform the given grammar into an ω -regular grammar. We then interpret the remaining ω -regular production rules as UUParse parsers using the *ProductionRule* and *CharProductionRule* instances for UUParse parsers of type $P (Str Char) v$.

```

parseUU :: ∀ ϕ r ix . PRegularGrammar ϕ r →
  ϕ ix → String → r ix
parseUU gram idx s = let uparser = gram idx ⊗ pEnd
  in parse uparser $ listToStr s

```

A UUParse parser consuming tokens of type *Char* and producing a value of type v is represented as a value of type $P (Str Char) v$. We provide *ProductionRule* and *CharProductionRule* instances for these parsers. The definition of these instances amounts to a direct mapping of the *Applicative* and *Alternative* combinators \otimes , *pure*, \oplus , *empty* and its primitive parsers *pSym* and *pEnd*.

```

instance ProductionRule (P (Str Char)) where
  (>>>) = (⊗)
  (|||) = (⊕)
  ε[·] = pure
  endOfInput = () ⊗ pEnd
  die = empty
instance CharProductionRule (P (Str Char)) where
  token = pSym

```

For a text about a parsing library, we admit it's taken us a long time (the full companion paper and 17 pages so far) to get to a point where we can actually parse something. But yes, we are there now. Figure 5 shows the result of parsing a test string using the transformed versions of our running example grammar combined with the semantic processor `calcArith` defined in the companion paper. In our example, we have used the uniform-Paull transformation, but we could have used the left-corner transformation just as well. Note that the grammar we are using to parse is constructed using automatic transformations from a left-recursive grammar that could not have been handled by `UUParse` directly.

```

ghci> parseUU (unfoldRecursion (unfoldLoops (transformUniformPaullE
    calcGrammarArith))) Expr_1 "(6*(4+2)+(2*3)"

```

```

UPBV {unUPBV = ArithValueE 42}

```

Fig. 5: The result of parsing a test string with the transformed grammar of Figure 4 using the `UUParse` combinator library. We use the grammar `calcGrammarArith` defined in section 2.2.

4.2 What we don't show

In the `grammar-combinators` library, we define several other grammar algorithms, which we do not demonstrate here. In this section, we provide a short overview of what is available.

Other parser algorithms The most important parser algorithm of our library that we do not have space to show, is a general implementation of a packrat parser [10]. We employ a domain memoization type class to provide the fundamental memoization we need, thus allowing our grammars to be used unmodified with the Packrat algorithm.

```

class MemoFam (ϕ :: * → *) where
  data Memo ϕ :: (* → *) → *

```

$$\begin{aligned} \text{fromMemo} &:: \text{Memo } \phi \ v \rightarrow (\forall ix . \phi \ ix \rightarrow v \ ix) \\ \text{toMemo} &:: (\forall ix . \phi \ ix \rightarrow v \ ix) \rightarrow \text{Memo } \phi \ v \end{aligned}$$

The *MemoFam* type class allows a function over a domain to be converted back and forth to a memoized representation. This type class is fundamental to our packrat parser implementation, but it is also of a more general utility.

A second feature of our library that we do not have space to discuss is the ability to pre-compute tables at compile time. By using Sheard and Peyton Jones' compile-time meta-programming Haskell extension *Template Haskell* [11], we can implement parser algorithms that employ tables pre-computed at compile-time. As an example, we have implemented a simple LL(1) parser using compile-time pre-computed tables to choose between different production rules.

Grammar analysis and transformations There is quite some grammar analysis and transformation tooling that we do not show in this paper. Most implementations are similar to what we have seen in this paper, even if some use recursion handling schemes different from the ones we have seen (e.g. limiting recursion depth to 1).

Quite useful is a reachability analysis, determining the set of non-terminals that can be reached from a given start non-terminal, as well as a *foldReachable* function folding over all reachable non-terminals in a manner similar to the *foldFam* function seen in section 3.1. In fact, the *foldReachable* function is often more useful in practice than the *foldFam* function, especially when using a grammar transformation like the left-corner transform that produces grammars with many non-terminals. Additionally, it does not require a special type class.

Furthermore, we provide a library of grammar post-processing functions, performing various relatively simple tasks like removing dead branches (*filterDies*), unfolding chain non-terminals (*unfoldChainNTs*), selective and depth-limited unfolding of non-terminals (*unfoldSelective*), removing branches that reference non-terminals that can never match, combining consecutive $\epsilon[\cdot]$'s etc. Some of these have been used in examples in this technical report.

Tools Furthermore, we provide an assortment of generally useful grammar algorithms. Among others, we provide a production rule visualization algorithm, using Martin Erwig's functional graph library [12] and the Graphviz [13] graph drawing toolset, a simple algorithm enumerating all possible grammar productions (*enumerateGrammar*), a simple algorithm assessing the size of a grammar (*assessSize*) and an algorithm that will calculate the FIRST sets of non-terminals (*calcFirst*). The *calcFirst* algorithm was used in the definition of the left-corner transform in section 3.3.

Parser generation Another algorithm that we have implemented in our library is called *liftGrammar*. The main idea is that it is often preferable to perform grammar transformations at compile time and have the transformed grammar directly available at runtime instead of requiring it to be calculated at runtime.

We have implemented this using Template Haskell [11], with a production rule interpretation producing a quoted version of the rule, such as the following:

```
data LiftedRule  $\phi$   $r = LR \{ unLR :: Q \textit{Exp} \}$ 
instance ProductionRule (LiftedRule  $\phi$   $r$   $t$ ) where
   $a \ggg b = LR [| \$(unLR\ a) \ggg \$(unLR\ b) |]$ 
   $a ||| b = LR [| \$(unLR\ a) ||| \$(unLR\ b) |]$ 
   $die = LR [| die |]$ 
   $endOfInput = LR [| endOfInput |]$ 
  -- does not work: v's type not in Lift class...
   $\epsilon[v] = LR [| \epsilon[\cdot] \$(lift\ v) |]$ 
```

The problem with this is that we cannot implement the $\epsilon[\cdot]$ function in this way without imposing a *Lift a* constraint in the definition of $\epsilon[\cdot]$ in the *ProductionRule* class, but such a constraint would cause problems throughout the rest of our library and complicate the natural interface that we can now provide. We have solved this problem by splitting out the $\epsilon[\cdot]$ combinator to a separate type class and providing an alternative *epsilonL* combinator. The *epsilonL* combinator takes a value and additionally the value's Template Haskell representation.

```
class (ProductionRule  $p$ )  $\Rightarrow$  LiftableProductionRule  $p$  where
   $\epsilonpsilonL :: a \rightarrow Q \textit{Exp} \rightarrow p\ a$ 
class (LiftableProductionRule  $p$ )  $\Rightarrow$  EpsProductionRule  $p$  where
   $\epsilon[\cdot] :: a \rightarrow p\ a$ 
```

Together with an additional *LiftFam* type class for domains, we can define the *liftGrammar* algorithm which takes a restricted grammar (using only the *LiftableProductionRule* type class and not *EpsProductionRule*) and returns its direct definition as a TH declaration. Thus, when you pass it a transformed grammar, it will perform the transformation at compile time and produce the definition of the transformed grammar to be included in the source code, in a way that is somewhat similar to what a parser generator would do.

4.3 Future work

There's clearly a lot of work still to be done further developing our grammar combinators library. We discuss some ideas that seem interesting and could constitute future work.

One thing we would like to investigate is whether it is feasible to implement bottom-up parser algorithms (GLR, LR or LALR algorithms) in our framework. The *transformUniformPaull* transformation was actually started as an implementation of a LR-parser in a recursive ascent style, and one can intuitively see the similarity in behaviour of a transformed grammar in an LL(1) parser and the original grammar in an LR(1) parser, but we are not sure about a possible equivalence of both. An equivalence between the left-corner parse algorithm

working with a given grammar and a top-down parser working with a left-corner transformed version of that grammar has previously been shown [14].

A practically important feature provided by traditional parser combinator libraries that we do not yet support is the ability to provide a library of commonly useful non-terminals, like “positiveDecimal”, “cStyleComment” or “haskellIdentifier”. There are various possible ways to implement this. One possibility is to use a type class and implementation algorithm similar to the *foldLoops* implementation in section 3.2, but it seems that a more general way to employ existing grammars as subgrammars in a new grammar could be a more powerful idea.

Another important feature that we currently do not provide is the ability to go further than what context-free grammars allow us to do. For example, many practical grammars are easier to handle if some form of lookahead is allowed. Our library could support this with another *LookaheadProductionRule* type class providing some form of lookahead functionality. Likewise, many semantic processors actually need positional information about where a given non-terminal was parsed, and we need to look into how to provide this functionality in order to make our parsing library practically useful.

One more idea comes from the observation that another deficiency of traditional parser combinator libraries is that they cannot perform sanity checking of the modelled grammars. In our approach it would actually not be difficult to perform such checks (like LL(1)- or LL(*)-ness of the grammar). Going further than that, it might even be possible to automatically infer which branches in a grammar require backtracking and use this to automatically infer calls to Parsec’s backtracking *try* combinator, yielding a strong automatic optimisation strategy.

Actually, we feel the idea of using a single grammar with different parsing algorithms for different usage scenario’s is surprisingly underused. Most parser generators and parser combinator libraries we are aware of, commit to a single parsing algorithm. Our grammars are completely abstract and provide a good environment for further research into this approach.

Finally, we think that our library is also an ideal framework for research into novel parsing algorithms. We have some ideas about using a penalty-based error handling strategy together with an automatic ambiguization transformation to provide a powerful and general error handling facility for use in interactive scenario’s where speed is less of an issue.

5 Conclusion

In this technical report, we hope to have shown the increased power of the grammar model presented in the companion paper [1] and that we have given a sense of the declarative implementation style of our algorithms. We have given an overview of the functionality implemented in the `grammar-combinators` library and discussed ideas for future work.

References

1. Devriese, D., Piessens, F.: Explicitly recursive grammar combinators. Submitted for PADL (2011)
2. Moore, R.: Removing left recursion from context-free grammars. In: NAACL. (2000) 249–255
3. Swierstra, S., Duponcheel, L.: Deterministic, error-correcting combinator parsers. *Advanced Functional Programming* (1996) 184–207
4. Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. *Journal of Functional Programming* **17**(01) (2006) 1–82
5. Leijen, D., Meijer, E.: Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht (2001)
6. Frost, R., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. In: PADL. (2008) 167–181
7. Baars, A., Swierstra, S., Viera, M.: Typed transformations of typed abstract syntax. In: TLDI. (2009) 15–26
8. Blum, N., Koch, R.: Greibach normal form transformation revisited. *Information and Computation* **150**(1) (1999) 112–118
9. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques and Tools*. 2nd edition edn. Addison-Wesley (2006)
10. Ford, B.: Packrat parsing:: simple, powerful, lazy, linear time - functional pearl. In: ICFP. (2002) 36–47
11. Sheard, T., Peyton Jones, S.: Template meta-programming for Haskell. *SIGPLAN Notices* **37**(12) (2002) 75
12. Erwig, M.: Inductive graphs and functional graph algorithms. *Journal of Functional Programming* **11**(05) (2001) 467–492
13. Ellson, J., Gansner, E., Koutsofios, L., North, S., Woodhull, G.: Graphviz - open source graph drawing tools. In: *Graph Drawing*. (2002) 594–597
14. Rosenkrantz, D.J., Lewis, P.M.: Deterministic left corner parsing. In: *Switching and Automata Theory*. (1970)