Formal, Executable and Reusable Components for Syntax Specification

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
http://hackage.haskell.org/package/gll

Royal Holloway, University of London

25 May, 2018
Observation 1

Semantically different constructs sometimes have *identical* syntax.

For example, variable and parameter declarations.

```scala
class Coordinate (val x : Int = 0, val y : Int = 0)
val someVal : String = "Royal Wedding"
```
Semantically different constructs sometimes have *identical* syntax.

For example, variable and parameter declarations.

```scala
class Coordinate (val x : Int = 0, val y : Int = 0)
val someVal : String = "Royal Wedding"
```

```
var_decl ::= var_key ID ' : ' TYPE opt_expr
var_key ::= "val" | "var"
opt_expr ::= expr | ε
expr ::= ...
```
Different constructs of a language may have similar syntax.

For example, a parameter list and an argument list.

```java
class Coordinate (val x : Int = 0, val y : Int = 0)
new Coordinate (4,2);
```
Different constructs of a language may have similar syntax.

For example, a parameter list and an argument list.

```java
class Coordinate (val x : Int = 0, val y : Int = 0)
new Coordinate (4,2);
```

```
param_list ::= '(' 'multiple_params' ')
multiple_params ::= \epsilon \mid var_decl multiple_params'
multiple_params' ::= \epsilon \mid ',,' var_decl multiple_params'
args_list ::= '(' 'multiple_exprs' ')
multiple_exprs ::= \epsilon \mid expr multiple_exprs'
multiple_exprs' ::= \epsilon \mid ',,' expr multiple_exprs'
```
Programming languages often have syntax in common.

However, there are often subtle differences:

---- JAVA ----
if (i < y) {
    System.out.println(...);
} else {
    arr[i] = myObj.getField();
}

---- HASKELL ----
if (i < y)
    then i+1
else let {f x = x + i;
          g x = x + 2}
in ...
Goal

Techniques for reuse within and between syntax specifications.

**formal:** We should be able to make mathematical claims about the defined languages, and support these claims by proofs.

**executable:** A parser for the language is mechanically derivable.

Motivation

- Simplify syntax definition
- Rapid prototyping
- Apply test-driven development in language design
BNF (Backus-Naur Form)

var_decl ::= var_key ID :: TYPE opt_expr
var_key ::= "val" | "var"
opt_expr ::= expr | ε

Formal
A BNF specification captures context-free grammars directly.

Executable
Generalised parsing, $O(n^3)$ parsers for all grammars:
Extensions to BNF capture common patterns.

\[
\text{var\_decl} ::= ("val" | "var") \text{ ID } ':' \text{ TYPE expr?}
\]

\[
\text{param\_list} ::= (\text{\{ var\_decl \, ,\} } \text{ )})
\]

\[
\text{args\_list} ::= (\text{\{ expr \, ,\} } \text{ )})
\]

The extensions either generate underlying BNF, or are associated with implicit rules like:

\[
\{ a \ b \} ::= \epsilon | a \ b \ a | a \ b \ a \ b \ a | \ldots
\]

What if the provided extensions are not sufficient?
Parameterised non-terminals enable user-defined extensions:

\[
\begin{align*}
    \text{either}(a, b) & ::= a \mid b \\
    \text{maybe}(a) & ::= a \mid \epsilon \\
    \text{var} \_ \text{decl} & ::= \text{either}("val", "var") \ ID \ ': ' \ \text{TYPE} \ \text{maybe}(\text{expr}) \\
    \text{sepBy}(a, b) & ::= \epsilon \mid \text{sepBy1}(a, b) \\
    \text{sepBy1}(a, b) & ::= a \mid a \ b \ \text{sepBy1}(a, b) \\
    \text{args} \_ \text{list} & ::= '( ' \ \text{sepBy}(\text{expr}, ',', ') ' ') '
\end{align*}
\]

A simple closure algorithm transforms such specifications into BNF. This algorithm may fail to terminate!
Overview

BNF route

BNF → Generalised parsing → EBNF → PBNF

formality → expressivity

Parser combinator route

HO-functions → Parser combinators → Languages $L$?

expressivity → formality
A parse function $p$ takes an input string $l$ and an index $k$ and returns indices $r \in p(l, k)$ if $p$ recognises string $l_{k,r}$

$$tm(x)(l, k) = \begin{cases} 
\{ k + 1 \} & \text{if } l_k = x \\
\emptyset & \text{otherwise}
\end{cases}$$
The Parser Combinator Approach

Parsers are formed by combining parse functions with *combinators*:

\[
\begin{align*}
  \text{seq}(p, q)(l, k) &= \{ r \mid r' \in p(l, k), r \in q(l, r') \} \\
  \text{alt}(p, q)(l, k) &= p(l, k) \cup q(l, k) \\
  \text{succeeds}(l, k) &= \{ k \} \\
  \text{fails}(l, k) &= \emptyset
\end{align*}
\]

Parse function \( p \) recognises string \( l \) if \( |l| \in p(l, 0) \)

\[
\text{recognise}(p)(l) = \begin{cases} 
  \text{true} & \text{if } |l| \in p(l, 0) \\
  \text{false} & \text{otherwise}
\end{cases}
\]
Example parser

\[
sepBy1(p, s) ::= p | p \ s \ sepBy1(p, s)
\]

\[
sepBy1(p, s) = alt(p, seq(p, seq(s, sepBy1(p, s))))
\]

Parse function \textit{parens}(sepBy1(tm(‘a’), \text{\texttt{,}})) \textit{recognises}:

\{"(a)", "(a,a)", "(a,a,a)", \ldots\}
What is the language recognised by a parse function?

\[
\begin{align*}
\mathcal{L}(tm(x)) &= \{x\} \\
\mathcal{L}(seq(p, q)) &= \{\alpha\beta \mid \alpha \in \mathcal{L}(p), \beta \in \mathcal{L}(q)\} \\
\mathcal{L}(alt(p, q)) &= \mathcal{L}(p) \cup \mathcal{L}(q) \\
\mathcal{L}(succeeds) &= \{\epsilon\} \\
\mathcal{L}(fails) &= \emptyset
\end{align*}
\]

Can be used to attempt proofs of the form: \( \mathcal{L}(p) = \mathcal{L}(q) \)
The combinators are defined such that the following laws hold:

\[
\begin{align*}
alt(fails, q) &= q \\
alt(p, fails) &= p \\
alt(p, p) &= p \\
alt(p, q) &= alt(q, p) \\
alt(p, alt(q, r)) &= alt(alt(p, q), r) \\
seq(succeeds, q) &= q \\
seq(p, succeeds) &= p \\
seq(fails, q) &= fails \\
seq(p, fails) &= fails \\
seq(p, seq(q, r)) &= seq(seq(p, q), r)
\end{align*}
\]
We can also prove distributivity of \texttt{seq} over \texttt{alt}

\[
\begin{align*}
\text{seq}(p, \text{alt}(q, r)) &= \text{alt}(\text{seq}(p, q), \text{seq}(p, r)) \\
\text{seq}(\text{alt}(p, q), r) &= \text{alt}(\text{seq}(p, r), \text{seq}(q, r))
\end{align*}
\]

The first law can be used to ‘refactor’ the definition of \texttt{sepBy1}

\[
\text{sepBy1}(p, s) = \text{alt}(\overline{p}, \text{seq}(p, \text{seq}(s, \text{sepBy1}(p, s)))) \\
= \text{alt}(\text{seq}(p, \text{succeeds}), \text{seq}(p, \text{seq}(s, \text{sepBy1}(p, s)))) \\
= \text{seq}(p, \text{alt}(\text{succeeds}, \text{seq}(s, \text{sepBy1}(p, s))))
\]
In practice, many more combinators are provided.

In practice, parsers produce a single result, or a list of results.

Common variations of \(alt\) and \(seq\) do not have the same laws.

\[
alt(p, q)(l, k) = \begin{cases} 
p(l, k) & \text{if } p(l, k) \neq \emptyset \\
q(l, k) & \text{otherwise}
\end{cases}
\]

Parsers often require refactoring for efficiency (backtracking) or even termination (left-recursion).

Generalisations complicate combinators definitions.
A third route: Grammar Combinators (Embedded BNF)

**Formal**

*Combinator expressions produce grammar objects.*
The usual notions of productions and derivations apply.

**Executable**

*Grammars given to stand-alone parsing procedure.* (Ljunglöf 2002)
So-called “semantic actions” can be integrated. (Ridge 2014)
Formal

*Combinator expressions produce grammar objects.*
The usual notions of productions and derivations apply.

Executable

*Grammars given to stand-alone parsing procedure.*  (Ljunglöf 2002)
So-called “semantic actions” can be integrated.   (Ridge 2014)

+ Rich abstraction mechanism provided by the host language
+ Borrows host language’s module-system, type-system, etc.
+ Generalised parsing techniques available
A third route: Grammar Combinators (Embedded BNF)

**Formal**

Combinator expressions produce grammar objects.
The usual notions of productions and derivations apply.

**Executable**

Grammars given to stand-alone parsing procedure.  (Ljunglöf 2002)
So-called “semantic actions” can be integrated.  (Ridge 2014)

+ Rich abstraction mechanism provided by the host language
+ Borrows host language’s module-system, type-system, etc.
+ Generalised parsing techniques available
  - Not as flexible and expressive as parser combinators
  - Inherently restricted to (context-free) grammars
    (The types of grammars accepted by the parsing procedure.)
  - Static computation requires meta-programming (lookahead)
We saw three methods for achieving reuse in syntax specifications:

- PBNF
- Parser combinators
- Grammar combinators

PBNF is formal and executable, but restricted to BNF.

Parser combinators offer tremendous power and flexibility. However, formality and expressivity are at odds.

Grammar combinators implement BNF with the benefits of EDSLs: abstraction (PBNF), user-extensible, static type-checking, etc.
Formal, Executable and Reusable Components for Syntax Specification

L. Thomas van Binsbergen
ltvanbinsbergen@acm.org
http://hackage.haskell.org/package/gll

Royal Holloway, University of London

25 May, 2018
**Algorithm**

- Copy all nonterminals without parameters; add their rules
- While there is a right-hand side application $f(a_1, \ldots, a_n)$:
  - Generate nonterminal $f_{a_1, \ldots, a_n}$, if necessary, and if so
  - ‘Instantiate’ the alternates for $f$ and add to $f_{a_1, \ldots, a_n}$
  - Replace application with $f_{a_1, \ldots, a_n}$
Algorithm

- Copy all nonterminals without parameters; add their rules
- While there is a right-hand side application \( f(a_1, \ldots, a_n) \):
  - Generate nonterminal \( f_{a_1, \ldots, a_n} \), if necessary, and if so
  - ‘Instantiate’ the alternates for \( f \) and add to \( f_{a_1, \ldots, a_n} \)
  - Replace application with \( f_{a_1, \ldots, a_n} \)

\[
\begin{align*}
\text{var\_decl} & ::= \text{either(“val”, “var”)} \ ID \ ’:\’ \ TYPE \ \text{maybe(}expr\text{)} \\
\text{either}(a, b) & ::= a \mid b \\
\text{maybe}(a) & ::= a \mid \epsilon
\end{align*}
\]
Algorithm

- Copy all nonterminals without parameters; add their rules
- While there is a right-hand side application $f(a_1, \ldots, a_n)$:
  - Generate nonterminal $f_{a_1,\ldots,a_n}$, if necessary, and if so
  - ‘Instantiate’ the alternates for $f$ and add to $f_{a_1,\ldots,a_n}$
  - Replace application with $f_{a_1,\ldots,a_n}$

```
var_decl ::= either("val","var") ID ':' TYPE maybe(expr)
either(a, b) ::= a | b
maybe(a) ::= a | $\epsilon$
```

```
var_decl ::= either "val","var" ID ':' TYPE maybe_{expr}
either "val","var" ::= "val" | "var"
maybe_{expr} ::= expr | $\epsilon$
```
Fails to terminate when arguments are ‘growing’:

\[
\begin{align*}
scales(a) & \ ::= \ a \mid a \ scales(parens(a)) \\
parens(a) & \ ::= \ '(\ a \ ')' \\
\end{align*}
\]

Language of \textit{scales} (‘a’):

\{"a", "a(a)", "a(a)((a))", "a(a)((a))((a)))", \ldots\}
Fails to terminate when arguments are ‘growing’:

\[ scales(a) ::= a \mid a \text{scales}(parens(a)) \]
\[ parens(a) ::= '( 'a' ')' \]

\[ scales\_a ::= 'a' \mid 'a' \text{scales}\_parens\_a \]
\[ scales\_parens\_a ::= parens\_a \mid parens\_a \text{scales}\_parens\_parens\_a \]

\[ parens\_a ::= '( 'a' ')' \]
\[ parens\_parens\_a ::= '( 'parens\_a' ')' \]
\[ parens\_parens\_parens\_a ::= '( 'parens\_parens\_a' ')' \]

Language of \textit{scales}'(a)\):

\{"a", "a(a)", "a(a)((a))", "a(a)((a))((a)))", \ldots\}