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
Semantically different constructs sometimes have *identical* syntax.

For example, variable and parameter declarations.

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

The parameter and variable declarations follow the pattern:
(“val” or “var”) identifier ‘:’ type ‘==’ expression
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"
```

The parameter and variable declarations follow the pattern:
("val" or "var") identifier `:` type `=` expression

```
var_decl ::= var_key ID `:` TYPE opt_expr
var_key ::= "val" | "var"
opt_expr ::= expr | ε
expr ::= ...
```
Observation 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);
```
Observation 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 | var_decl multiple_params'
multiple_params' ::= epsilon | ',' var_decl multiple_params'
args_list ::= ' ( ' multiple_exprs ' ) '
multiple_exprs ::= epsilon | expr multiple_exprs'
multiple_exprs' ::= epsilon | ',' expr multiple_exprs'
```
Programming languages often have syntax in common.

For example, if-then-else, or “assignment” to a variable using ‘=’.

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 the process of defining syntax by reusing aspects of language itself as well as from other languages
- Rapid prototyping
- Apply test-driven development in language design
- Syntax comparison based on specification (a.o.t. examples)
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. A string is derived from a nonterminal according to productions:

```
var_decl => var_key ID ' : ' TYPE opt_expr
=> var_key ID ' : ' TYPE => "val" ID ' : ' TYPE
```

Executable

A BNF specification captures context-free grammars directly. A string is derived from a nonterminal according to productions:

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

Extensions to BNF capture common patterns.

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

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

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

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

\[
(a \ | \ b) \Rightarrow a \\
(a \ | \ b) \Rightarrow b \\
\{a \ b\} \Rightarrow \\
\{a \ b\} \Rightarrow a \ b \ a \\
\{a \ b\} \Rightarrow a \ b \ a \ b \ a \\
\ldots
\]

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

```
var_decl ::= either("val","var") ID ':' TYPE maybe(expr)

either(a, b) ::= a | b
maybe(a) ::= a | ε

param_list ::= tuple(var_decl)
args_list ::= tuple(expr)
tuple(a) ::= '(' sepBy(a, ',') ')

sepBy(a, b) ::= ε | sepBy1(a, b)
sepBy1(a, b) ::= a | a b sepBy1(a, b)
```

A simple algorithm transforms such specifications into BNF. This algorithm fails to terminate when there is no “fixed point”.
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 $f_{a_1, \ldots, a_n}$

$\text{var\_decl} ::= \text{either( } \text{"val", } \text{"var"}\text{) ID } \text{'}:\text{'} \text{ TYPE } \text{maybe( expr)\n
either}(a, b) ::= a | b$

$\text{maybe}(a) ::= a | \epsilon$
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’:

\[
\begin{align*}
scales(a) & ::= a \mid a \text{ scales}(\text{parens}(a)) \\
\text{parens}(a) & ::= (\ ' a ' )
\end{align*}
\]
Fails to terminate when arguments are ‘growing’:

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

BNF route

- BNF
- Generalised parsing
- EBNF
- PBNF

formality → expressivity

Parser combinator route

- HO-functions
- Parser combinators
- Languages $\mathcal{L}$?
- Combinator laws

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

$$tm(x)(I, k) = \begin{cases} 
\{k + 1\} & \text{if } I_k = x \\
\emptyset & \text{otherwise}
\end{cases}$$

For example, $tm(x)$ is a parse function recognising $I_{k,k+1}$ for all $I$ and $k$ with $I_k = x$.
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 parsers

parens(p) = seq(tm(' '), seq(p, tm(' '))))
sepBy1(p, s) = alt(p, seq(p, seq(s, sepBy1(p, s)))))

Parse function parens(sepBy1(tm('a'), tm(' ', ' '))) recognises:

{"(a)", "(a,a)", "(a,a,a)", ...}

scales(p) = alt(p, seq(p, scales(parens(p)))))

Parse function scales(tm('a')) recognises:

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

\[ \mathcal{L}(p) = \{ I \mid I \in W^*, \text{recognise}(p)(I) \} \]

How about a constructive definition?

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

Can be used to attempt proofs of the form: \( \mathcal{L}(p) = \mathcal{L}(q) \)
Formal reasoning II - Equalities

The combinators are defined such that the following laws hold:

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

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

\[
\text{seq}(p, \text{alt}(q, r)) = \text{alt}(\text{seq}(p, q), \text{seq}(p, r)) \\
\text{seq}(\text{alt}(p, q), r) = \text{alt}(\text{seq}(p, r), \text{seq}(q, r))
\]

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

\[
\text{sepBy1}(p, s) = \text{alt}(\overline{p}, \text{seq}(p, \text{seq}(s, \text{sepBy1}(p, s)))) \\
= \text{alt}(\text{seq}(p, \text{succeeds}), \text{seq}(p, \text{seq}(s, \text{sepBy1}(p, s)))) \\
= \text{seq}(p, \text{alt}(\text{succeeds}, \text{seq}(s, \text{sepBy1}(p, s))))
\]
Problems

- In practice, many more combinators are provided

\[
\text{eof}(p)(l, k) = \{ k \mid k = |l| \}
\]
\[
\mathcal{L}(\text{eof}(p)) = \{ \epsilon \} \text{ or } \emptyset
\]

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

- Common variations of \textit{alt} and \textit{seq} do not have the same laws

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

- 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)
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
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