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.

```
class Coordinate (val x : Int = 0, val y : Int = 0)

val someVal : String = "Royal Wedding"
```

Observation 1

Semantically different constructs sometimes have *identical* syntax.

For example, variable and parameter declarations.

```
class Coordinate (val x : Int = 0, val y : Int = 0)

val someVal : String = "Royal Wedding"
```

$$
\begin{array}{lll}
\textit{var\_decl} & ::= & \textit{var\_key}\ \textit{ID}\ \text{':'}\ \textit{TYPE}\ \textit{opt\_expr} \\
\textit{var\_key} & ::= & \texttt{"val"} \mid \texttt{"var"} \\
\textit{opt\_expr} & ::= & \textit{expr} \mid \epsilon \\
\textit{expr} & ::= & \ldots
\end{array}
$$

Observation 2

Different constructs of a language may have *similar* syntax.

For example, a parameter list and an argument list.

```
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.

```
class Coordinate (val x : Int = 0, val y : Int = 0)

new Coordinate (4,2);
```

$$
\begin{array}{lcl}
\mathit{param\_list} & ::= & \text{'('}\ \mathit{multiple\_params}\ \text{')'} \\
\mathit{multiple\_params} & ::= & \epsilon \mid \mathit{var\_decl}\ \mathit{multiple\_params'} \\
\mathit{multiple\_params'} & ::= & \epsilon \mid \text{','}\ \mathit{var\_decl}\ \mathit{multiple\_params'} \\
\mathit{args\_list} & ::= & \text{'('}\ \mathit{multiple\_exprs}\ \text{')'} \\
\mathit{multiple\_exprs} & ::= & \epsilon \mid \mathit{expr}\ \mathit{multiple\_exprs'} \\
\mathit{multiple\_exprs'} & ::= & \epsilon \mid \text{','}\ \mathit{expr}\ \mathit{multiple\_exprs'}
\end{array}
$$

Observation 3

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)

$$
\begin{aligned}
\textit{var\_decl} &::= \textit{var\_key}\ \textit{ID}\ \texttt{':'}\ \textit{TYPE}\ \textit{opt\_expr} \\
\textit{var\_key} &::= \texttt{"val"} \mid \texttt{"var"} \\
\textit{opt\_expr} &::= \textit{expr} \mid \epsilon
\end{aligned}
$$

Formal

A BNF specification captures context-free grammars directly.

Executable

Generalised parsing, $O(n^3)$ parsers for all grammars:
Earley (1970), GLR (1985), GLL (2010/2013)

Extended BNF (EBNF)

Extensions to BNF capture common patterns.

$$\begin{aligned}
\textit{var_decl} &::= (\texttt{"val"} \mid \texttt{"var"})\ \textit{ID}\ \texttt{':'}\ \textit{TYPE}\ \textit{expr}^{?} \\
\textit{param_list} &::= \texttt{'('}\ \{\textit{var_decl}\ \texttt{','}\}\ \texttt{')'} \\
\textit{args_list} &::= \texttt{'('}\ \{\textit{expr}\ \texttt{','}\}\ \texttt{')'}
\end{aligned}$$

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

$$\{a\ b\} ::= \epsilon \mid a\ b\ a \mid a\ b\ a\ b\ a \mid \ldots$$

What if the provided extensions are not sufficient?

Parameterised BNF (PBNF)

Parameterised non-terminals enable user-defined extensions:

$$
\begin{aligned}
\mathit{either}(a, b) &::= a \mid b \\
\mathit{maybe}(a) &::= a \mid \epsilon \\
\mathit{var\_decl} &::= \mathit{either}(\texttt{"val"}, \texttt{"var"})\ \mathit{ID}\ \texttt{':'}\ \mathit{TYPE}\ \mathit{maybe}(\mathit{expr})
\end{aligned}
$$

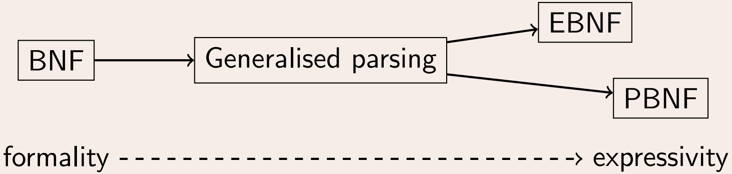
$$
\begin{aligned}
\mathit{sepBy}(a, b) &::= \epsilon \mid \mathit{sepBy1}(a, b) \\
\mathit{sepBy1}(a, b) &::= a \mid a\ b\ \mathit{sepBy1}(a, b) \\
\mathit{args\_list} &::= \texttt{'('}\ \mathit{sepBy}(\mathit{expr}, \texttt{','})\ \texttt{')'}
\end{aligned}
$$

A simple closure algorithm transforms such specifications into BNF.

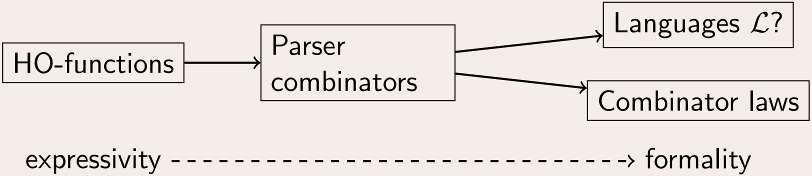
This algorithm may fail to terminate!

Overview

BNF route



Parser combinator route



The Parser Combinator Approach

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 $I_{k,r}$

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

The Parser Combinator Approach

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

$$\begin{aligned} seq(p,q)(I,k) &= \{r \mid r' \in p(I,k), r \in q(I,r')\} \\ alt(p,q)(I,k) &= p(I,k) \cup q(I,k) \\ succeeds(I,k) &= \{k\} \\ fails(I,k) &= \emptyset \end{aligned}$$

Parse function p *recognises* string I if $|I| \in p(I,0)$

$$recognise(p)(I) = \begin{cases} \textbf{true} & \textbf{if } |I| \in p(I,0) \\ \textbf{false} & \text{otherwise} \end{cases}$$

Example parser

$$sepBy1(p, s) ::= p \mid p\ s\ sepBy1(p, s)$$

$$\boxed{sepBy1(p, s) = alt(p, seq(p, seq(s, sepBy1(p, s))))}$$

Parse function $parens(sepBy1(tm(\texttt{'a'}), tm(\texttt{','})))$ recognises:

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

Formal reasoning I - Languages

What is the language recognised by a parse function?

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

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:

$$
\begin{aligned}
alt(fails, q) &= q \\
alt(p, fails) &= p \\
alt(p, p) &= p \\
alt(p, q) &= alt(q, p) \\
alt(p, alt(q, r)) &= alt(alt(p, q), r) \\
seq(succeeds, q) &= q \\
seq(p, succeeds) &= p \\
seq(fails, q) &= fails \\
seq(p, fails) &= fails \\
seq(p, seq(q, r)) &= seq(seq(p, q), r)
\end{aligned}
$$

Formal reasoning II - Equalities

We can also prove distributivity of *seq* over *alt*

$$seq(p, alt(q, r)) = alt(seq(p, q), seq(p, r))$$
$$seq(alt(p, q), r) = alt(seq(p, r), seq(q, r))$$

The first law can be used to 'refactor' the definition of *sepBy1*

$$\begin{aligned} sepBy1(p, s) &= alt(\overline{p}, seq(p, seq(s, sepBy1(p, s)))) \\ &= alt(\overline{seq(p, succeeds)}, seq(p, seq(s, sepBy1(p, s)))) \\ &= seq(p, alt(succeeds, seq(s, sepBy1(p, s)))) \end{aligned}$$

Problems

- 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

$$alt(p, q)(I, k) = \begin{cases} p(I, k) & \textbf{if } p(I, k) \neq \emptyset \\ q(I, k) & \text{otherwise} \end{cases}$$

- 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)

Conclusion

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

PBNF - algorithm

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 $f_{a_1,\ldots,a_n}$

PBNF - algorithm

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 $f_{a_1,\ldots,a_n}$

$$
\begin{array}{lcl}
\mathit{var\_decl} & ::= & \mathit{either}(\texttt{"val"}, \texttt{"var"})\ \mathit{ID}\ \texttt{':'}\ \mathit{TYPE}\ \mathit{maybe}(\mathit{expr}) \\
\mathit{either}(a, b) & ::= & a \mid b \\
\mathit{maybe}(a) & ::= & a \mid \epsilon
\end{array}
$$

PBNF - algorithm

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 $f_{a_1,\ldots,a_n}$

$$
\begin{array}{lcl}
\textit{var\_decl} & ::= & \textit{either}(\texttt{"val"}, \texttt{"var"})\ \textit{ID}\ \texttt{':'}\ \textit{TYPE}\ \textit{maybe}(\textit{expr}) \\
\textit{either}(a, b) & ::= & a \mid b \\
\textit{maybe}(a) & ::= & a \mid \epsilon
\end{array}
$$

$$
\begin{array}{lcl}
\textit{var\_decl} & ::= & \textit{either}_{\texttt{"val"},\texttt{"var"}}\ \textit{ID}\ \texttt{':'}\ \textit{TYPE}\ \textit{maybe}_{\textit{expr}} \\
\textit{either}_{\texttt{"val"},\texttt{"var"}} & ::= & \texttt{"val"} \mid \texttt{"var"} \\
\textit{maybe}_{\textit{expr}} & ::= & \textit{expr} \mid \epsilon
\end{array}
$$

PBNF - algorithm

Fails to terminate when arguments are ‘growing’:

$$\begin{array}{lcl} scales(a) & ::= & a \mid a\ scales(parens(a)) \\ parens(a) & ::= & \texttt{'('}\ a\ \texttt{')'} \end{array}$$

Language of *scales*('a'):

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

PBNF - algorithm

Fails to terminate when arguments are 'growing':

$$
\begin{array}{lcl}
\textit{scales}(a) & ::= & a \mid a\ \textit{scales}(\textit{parens}(a)) \\
\textit{parens}(a) & ::= & \texttt{'('}\ a\ \texttt{')'} \\
\hline
\textit{scales}_{\texttt{'a'}} & ::= & \texttt{'a'} \mid \texttt{'a'}\ \textit{scales}_{\textit{parens}_{\texttt{'a'}}} \\
\textit{scales}_{\textit{parens}_{\texttt{'a'}}} & ::= & \textit{parens}_{\texttt{'a'}} \mid \textit{parens}_{\texttt{'a'}}\ \textit{scales}_{\textit{parens}_{\textit{parens}_{\texttt{'a'}}}} \\
\ldots & & \\
\textit{parens}_{\texttt{'a'}} & ::= & \texttt{'('}\ a\ \texttt{')'} \\
\textit{parens}_{\textit{parens}_{\texttt{'a'}}} & ::= & \texttt{'('}\ \textit{parens}_{\texttt{'a'}}\ \texttt{')'} \\
\textit{parens}_{\textit{parens}_{\textit{parens}_{\texttt{'a'}}}} & ::= & \texttt{'('}\ \textit{parens}_{\textit{parens}_{\texttt{'a'}}}\ \texttt{')'} \\
\ldots & &
\end{array}
$$

Language of *scales*('a'):

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