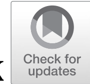
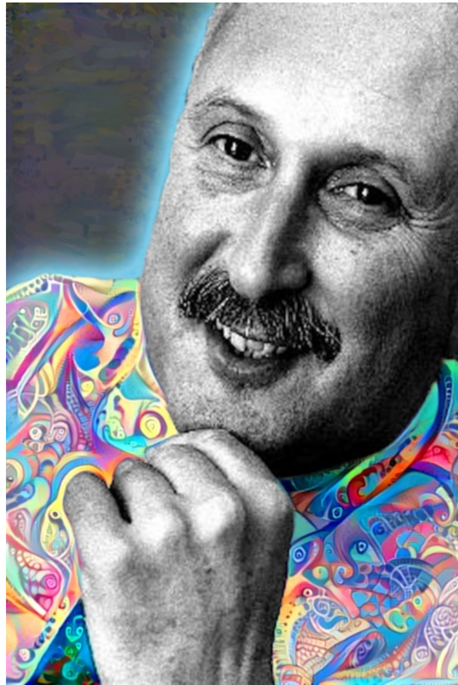
Chapter 7
Implementation of Textual Concrete Syntax





PAUL KLINT.[1]

Abstract This chapter discusses implementation aspects of textual concrete syntax: parsing, abstraction, formatting, and the use of concrete as opposed to abstract object syntax in metaprograms. We focus on how parsers, formatters, etc. are actually implemented in practice, subject to using appropriate libraries, tools, and metaprogramming techniques.

[1] Paul Klint's contributions to computer science are not limited to the implementation (or the practice or the application) of concrete syntax, but this is an area in which he has continuously driven the state of the art over the years. Some of his work on concrete syntax has focused on supporting it in interactive programming environments and language workbenches such as ASF+SDF and Rascal [21, 55, 31, 30]. In other work, he has been addressing practical challenges regarding parsing, for example, in terms of scannerless parsing and ambiguity detection in generalized (LR) parsing [13, 3]. Paul Klint loves grammars [29].

Artwork Credits for Chapter Opening: This work by Wojciech Kwasnik is licensed under CC BY-SA 4.0. This artwork quotes the artwork *DMT*, acrylic, 2006 by Matt Sheehy with the artist's permission. This work also quotes https://commons.wikimedia.org/wiki/File:Sunset_at_Montmajour_1888_Van_Gogh.jpg, subject to the attribution "Vincent van Gogh: Sunset at Montmajour (1888) [Public domain], via Wikimedia Commons." This work artistically morphes an image, http://homepages.cwi.nl/~paulk, showing the person honored, subject to the attribution "Permission granted by Paul Klint for use in this book."

© Springer International Publishing AG, part of Springer Nature 2018

R. Lämmel, *Software Languages*,
https://doi.org/10.1007/978-3-319-90800-7_7

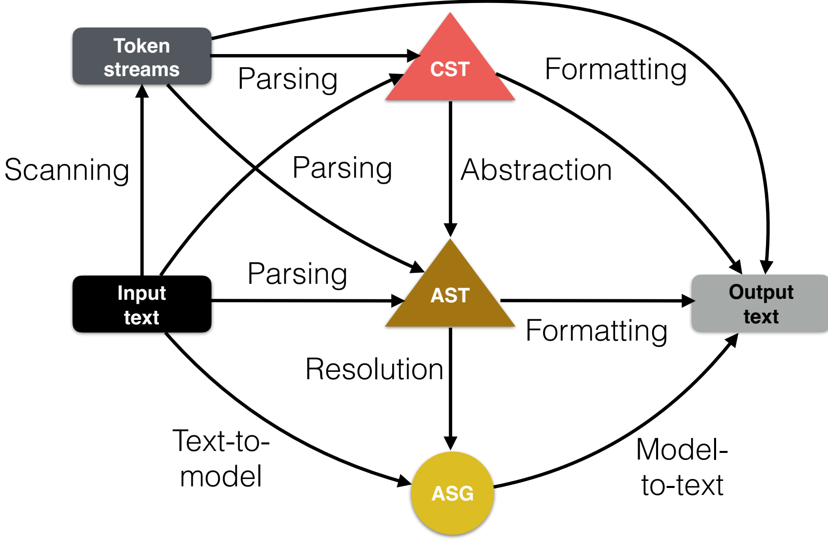


Fig. 7.1 Mappings (edges) between different representations (nodes) of language elements. For instance, "parsing" is a mapping from text or tokens to CSTs or ASTs.

7.1 Representations and Mappings

The big picture of concrete syntax implementation, as covered by this chapter, is shown in Fig. 7.1 with the exception of the special topic of concrete object syntax (Section 7.5). The nodes in the figure correspond to different representations of language elements; these representations have already been discussed, to some extent, but we summarize them here for clarity:

Text (String) Text is an important format for representing language elements. Text may serve as input or may arise as the output of language-processing activities. We do not discuss visual languages in this chapter.

Token stream Parsing may involve an extra phase, scanning, for processing text at the level of lexical syntax for units such as white space, comments, identifiers, and literals. The resulting units are referred to as tokens (or, more precisely, token-lexeme pairs). That is, a token is a classifier of a lexical unit. For instance, in FSML, we are concerned with "name" tokens as well as tokens for special characters, operators, and keywords (e.g., "/" and "state"). We use the term "lexeme" to refer to the string (text) that makes up a lexical unit. For instance, we may encounter the lexemes "locked", "unlocked", etc. in the input; we classify them as "name" tokens. In practice, the term "token" is also used to include lexemes.

CST Concrete syntax trees typically arise as the result of parsing text. These trees follow the structure of the underlying grammar; each node, with its subtrees,

represents the application of a grammar rule, except for some leaf nodes that simply represent terminals.

AST Abstract syntax trees may arise as the result of abstraction, i.e., a mapping from concrete to abstract syntax trees; they may also arise as the result of metaprograms such as transformers and generators.

ASG Abstract syntax graphs may be used to directly represent references on top of ASTs. In metamodeling, references are commonly used in models. Text-to-model transformations may also map text directly to graphs (models), without going through CSTs or ASTs explicitly. In other contexts, the explicit use of graphs is less common. For instance, in term rewriting, references are expressed only indirectly, for example, by means of traversals that look up subtrees or by an extra data structure for an environment.

The edges in the figure correspond to common forms of mapping:

Parsing This is a mapping from text or token streams to CSTs or ASTs.

Scanning This is a mapping from text to token streams, as an optional phase of parsing. It is the process of recognizing tokens. Scanning is like parsing, but at the level of lexical syntax. The input of scanning is text; the output is a token stream (to be precise, a stream of token-lexeme pairs), possibly enriched by position information. Scanning is performed by a scanner (a lexer). The underlying lexical syntax may be defined by regular or context-free grammars; see the examples in Section 6.1.4.

Abstraction This is a mapping from concrete to abstract syntax, i.e., from CSTs to ASTs.

Resolution This is a mapping from tree- to graph-based abstract syntax, i.e., from ASTs to ASGs.

Text-to-model This is about mapping from text that is an element of a language generated by a grammar to a model that conforms to a metamodel. Here, the grammar and metamodel are viewed as alternative syntax definitions for the same language.

Formatting This is a mapping from ASTs to text. We may expect that unparsing will produce textual output that follows some formatting conventions in terms of adding space, indentation, line breaks, and appropriate use of parentheses. As indicated in the figure, formatting can also be taken to mean that CSTs are mapped to text, in which case we may also speak of unparsing. Such a mapping should be straightforward because the leaf nodes of a CST should represent the text except that space, indentation, line breaks, and comments may be missing. Formatting may also operate at a lexical level such that token streams are mapped to text. Generally, we may also speak of “pretty printing” instead of formatting.

Model-to-text This is the opposite of “text-to-model”. In different terms, if “text-to-model” is the metamodeling-centric variation on parsing, then “model-to-text” is the metamodeling-centric variation on formatting or pretty printing.

7.2 Parsing

A parser maps text to concrete or possibly abstract syntax trees or graphs. Parsing is a prerequisite for metaprogramming on object programs with a textual syntax. We will discuss different approaches to the implementation of parsers; we assume that parsers are systematically, if not mechanically, derived from context-free grammars. To this end, we illustrate mainstream parsing technologies: ANTLR [46] and Parsec [37]. We also discuss parsing algorithms briefly.

7.2.1 Basic Parsing Algorithms

A grammar can be interpreted in a systematic, algorithmic manner so that one obtains an acceptor (Definition 6.5) or a parser (Definition 6.7) directly. We discuss here some simple, fundamental algorithms for top-down and bottom-up parsing. There are many options and challenges associated with parsing [20]; we only aim to convey some basic intuitions here.

7.2.1.1 Top-Down Acceptance

In top-down acceptance (or parsing), we maintain a stack of grammar symbols, which we initialize with the start symbol; we process the input from left to right. In each step, we either "consume" or "expand". In the "consume" case, we consume a terminal from the input if it is also at the top of the stack. In the "expand" case, we replace a nonterminal on the stack by a corresponding right-hand side.

Definition 7.1 (Algorithm for top-down acceptance)

Input:

- *a well-formed context-free grammar* $G = \langle N, T, P, s \rangle$*;*
- *a string (i.e., a list)* $w \in T^*$*.*

Output:

- *a Boolean value.*

Variables:

- *a stack* z *maintaining a sequence of grammar symbols;*
- *a string (i.e., a list)* i *maintaining the remaining input.*

Steps:

1. *Initialize* z *with* s *(i.e., the start symbol) as the top of the stack.*
2. *Initialize* i *with* w*.*

3. *If both i and z are empty, then return* true*.*
4. *If z is empty and i is nonempty, then return* false*.*
5. *Choose an action:*
   Consume: If the top of z is a terminal, then:
   a. *If the top of z equals the head of i, then:*
      i. *Remove the head of i.*
      ii. *Pop the top of z.*
   b. *Return* false *otherwise.*
   Expand: If the top of z is a nonterminal, then:
   a. *Choose a $p \in P$ with the top of z on the left-hand side of p.*
   b. *Pop the top of z.*
   c. *Push the symbols of the right-hand side of p onto z.*
6. *Go to 3.*

Table 7.1 Illustration of top-down acceptance

Step	Remaining input	Stack (TOS left)	Action
1	‘1’, ‘0’	*number*	Expand rule [*number*]
2	‘1’, ‘0’	*bits rest*	Expand rule [*many*]
3	‘1’, ‘0’	*bit bits rest*	Expand rule [*one*]
4	‘1’, ‘0’	‘1’ *bits rest*	Consume terminal ‘1’
5	‘0’	*bits rest*	Expand rule [*single*]
6	‘0’	*bit rest*	Expand rule [*zero*]
7	‘0’	‘0’ *rest*	Consume terminal ‘0’
8	–	*rest*	Expand rule [*integer*]
9	–	–	–

In the strict sense, the description is not a proper algorithm owing to nondeterminism (see the choice of action) and nontermination (think of infinite expansion for grammars with left recursion, as we will discuss more in detail below). Actual acceptance algorithms arise as refinements that constrain the choice or the grammar. In the sequence of steps shown in Table 7.1, we assume an oracle which tells us the “right” choice.

Exercise 7.1 (Nondeterminism of top-down acceptance) [Basic level]
Identify the steps in Table 7.1 that make a choice, and identify alternative actions. How do these alternatives reveal themselves as inappropriate?

Let us implement top-down acceptance based on the pseudo-algorithm in Definition 7.1. We aim only at a very basic implementation, meant to be useful for understanding parsing conceptually. We implement top-down acceptance in Haskell.

Assuming a suitable representation of BNL's BGL grammar, we expect to perform acceptance for binary numbers as follows:

Interactive Haskell session:

```
▶ accept bnlGrammar "101.01"
True
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ accept bnlGrammar "x"
False
```

We assume a typeful representation (Section 4.1.3) of the signature of BGL grammars (Section 6.6.1) as Haskell data types, as shown below.

Illustration 7.1 (Datatypes for grammar representation)

Haskell module Language.BGL.Syntax

```
type Grammar = [Rule]
type Rule = (Label, Nonterminal, [GSymbol])
data GSymbol = T Terminal | N Nonterminal
type Label = String
type Terminal = Char
type Nonterminal = String
```

Illustration 7.2 (The grammar of BNL represented in Haskell)

Haskell module Language.BGL.Sample

```
bnlGrammar :: Grammar
bnlGrammar = [
  ("number", "number", [N "bits", N "rest"]),
  ("single", "bits", [N "bit"]),
  ("many", "bits", [N "bit", N "bits"]),
  ("zero", "bit", [T '0']),
  ("one", "bit", [T '1']),
  ("integer", "rest", []),
  ("rational", "rest", [T '.', N "bits"])
 ]
```

Top-down acceptance is implemented in Haskell as follows.

Illustration 7.3 (Implementation of top-down acceptance)

Haskell module Language.BGL.TopDownAcceptor

```
1  accept :: [Rule] → String → Bool
2  accept g = steps g [N s]
3    where
4      -- Retrieve start symbol
5      ((_, s, _):_) = g
6
7  steps :: [Rule] → [GSymbol] → String → Bool
8  -- Acceptance succeeds (empty stack, all input consumed)
9  steps _ [] [] = True
10 -- Consume terminal at top of stack from input
11 steps g (T t:z) (t':i) | t==t' = steps g z i
12 -- Expand a nonterminal; try different alternatives
13 steps g (N n:z) i = or (map (λ rhs → steps g (rhs++z) i) rhss)
14   where
15     rhss = [ rhs | (_, n', rhs) ← g, n == n' ]
16 -- Otherwise parsing fails
17 steps _ _ _ = False
```

The implementation is based on these ideas:

- The start symbol is determined within the main function accept as the left-hand side of the first rule (line 5).
- The program maintains a parser stack, which is represented simply as a list of grammar symbols. The head of the list is the top of the stack. The stack is initialized with the start symbol (line 2).
- The regular termination case is that both the input and the stack are empty and, thus, True is returned (line 9).
- The case where a terminal t is the top of stack requires that the input starts with the same terminal (see the guard t==t'), in which case the terminal is removed from both the stack and the input before continuing with the remaining stack and input (line 11).
- The case where a nonterminal n is the top of stack forms a disjunction over all the possible right-hand sides for n; these options are collected in a list comprehension; and the right-hand sides replace n in the different attempts (lines 13–15).
- In all other cases, acceptance fails (line 17), i.e., when the terminal at the top of the stack is not met by the head of the input, or the stack is empty while the input is not empty.

This implementation is naive because it tries all alternatives without considering the input. More seriously, the implementation may exhibit nonterminating behavior if applied to a *left-recursive grammar*. Instead of defining left recursion here formally, let us just look at an example. The BNL grammar does not involve left recursion, but consider the following syntax of simple arithmetic expressions:

Illustration 7.4 (A left-recursive grammar for arithmetic expressions)

EGL resource languages/EGL/samples/left-recursion.egl

```
[add] expr : expr '+' expr ;
[const] expr : integer ;
```

The grammar is left-recursive owing to the first rule because, if we apply the first rule, then expr is replaced by a sequence of grammar symbols that again starts with expr. Thus, such a derivation or expansion process could go on forever without consuming any input. There are various techniques for dealing with or removing left recursion (see, e.g., [20, 1, 42, 38, 18, 19, 43]).

Thanks to the way in which alternatives are handled in the Haskell code for top-down acceptance, we do not commit to a particular choice, but all alternatives are potentially tried. In principle, there are two major options for combining the alternatives:

Local backtracking When a nonterminal is being expanded, the different alternatives are tried in the grammar-specified order; we commit to the first alternative for which acceptance succeeds, if there is any.

Global backtracking There is no commitment to an alternative. That is, we may consider yet other alternatives even after successful completion of an alternative triggered by failure in the enclosing scope.

These two options differ in terms of efficiency and completeness. An incomplete acceptor corresponds to the situation where some proper language elements would not be accepted. Local backtracking is more efficient, but less complete than global backtracking. The Haskell-based implementation in Illustration 7.3, as discussed above, facilitates global backtracking because the disjunction does not just model choice over alternatives; rather it models choice over all possible continuations of acceptance.

Exercise 7.2 (Backtracking variations) [Intermediate level]

Implement top-down acceptance with local backtracking.

The incompleteness of local backtracking can easily be observed on the basis of the BNL example. That is, consider the order of the rules [single] and [many] in Illustration 7.2. Local backtracking would commit us [single] even for the case of an input string with more than one digit. The rules [integer] and [rational] expose the same kind of order issue. Local backtracking is sufficient once we reorder the rules as follows.

Illustration 7.5 (BNL grammar for which local backtracking suffices)

Haskell module Language.BGL.SampleWithGreediness

```
bnlGrammar :: Grammar
bnlGrammar = [
  ("number", "number", [N "bits", N "rest"]),
  ("many", "bits", [N "bit", N "bits"]),
  ("single", "bits", [N "bit"]),
  ("zero", "bit", [T '0']),
  ("one", "bit", [T '1']),
  ("rational", "rest", [T '.', N "bits"]),
  ("integer", "rest", [])
 ]
```

There are various algorithms and grammar classes that cater for efficient top-down parsing without the issues at hand [20, 1].

7.2.1.2 Bottom-Up Acceptance

In bottom-up acceptance (or parsing), we maintain a stack of grammar symbols, starting from the empty stack; we process the input from left to right. In each step, we either "shift" or "reduce". In the "shift" case, we move a terminal from the input to the stack. In the "reduce" case, we replace a sequence of grammar symbols on the stack with a nonterminal, where the removed sequence must form the right-hand side and the added nonterminal must be the left-hand side of some grammar rule.

Definition 7.2 (Algorithm for bottom-up acceptance)

Input:

- *a well-formed context-free grammar* $G = \langle N, T, P, s \rangle$*;*
- *a string (i.e., a list)* $w \in T^*$.

Output:

- *a Boolean value.*

Variables:

- *a stack* z *maintaining a sequence of grammar symbols;*
- *a string (i.e., a list)* i *maintaining the remaining input.*

Steps:

1. *Initialize* z *with the empty stack.*
2. *Initialize* i *with* w.
3. *If* i *is empty and* z *consists of* s *alone, then return* true.

4. *Choose an action:*
   Shift: *Remove the head of i and push it onto z.*
   Reduce:
   a. *Pop a sequence x of symbols from z.*
   b. *Choose a $p \in P$ such that x equals the right-hand side of p.*
   c. *Push the left-hand side of p onto z.*
   Return false*, if no action is feasible.*
5. *Go to 3.*

Table 7.2 Illustration of bottom-up acceptance

Step	**Remaining input**	**Stack (TOS right)**	**Action**
1	'1', '0'	–	Shift terminal '1'
2	'0'	'1'	Reduce rule [*one*]
3	'0'	*bit*	Reduce rule [*single*]
4	'0'	*bits*	Shift terminal '0'
5	–	*bits* '0'	Reduce rule [*one*]
6	–	*bits bit*	Reduce rule [*many*]
7	–	*bits*	Reduce rule [*integer*]
8	–	*bits rest*	Reduce rule [*number*]
9	–	*number*	–

It is insightful to notice how top-down acceptance (Table 7.1) and bottom-up acceptance (Table 7.2) are opposites of each other in some sense. The top-down scheme starts with *s* on the stack; the bottom-up scheme ends with *s* on the stack. The top-down scheme ends with an empty stack; the bottom-up scheme starts from an empty stack.

Exercise 7.3 (Nondeterminism of bottom-up acceptance) [Basic level]
Identify the steps in Table 7.2 that make a choice, and identify alternative actions. How does the inappropriateness of the options reveal itself?

Let us implement bottom-up acceptance based on the pseudo-algorithm in Definition 7.2. We aim again at a very basic implementation, meant to be useful for understanding parsing conceptually. We implement bottom-up acceptance in Haskell as follows.

Illustration 7.6 (Implementation of bottom-up acceptance)

Haskell module Language.BGL.BottomUpAcceptor

```
1  accept :: [Rule] → String → Bool
2  accept g = steps g [] -- Begin with empty stack
3
4  steps :: [Rule] → [GSymbol] → String → Bool
5  -- Acceptance succeeds (start symbol on stack, all input consumed)
6  steps g [N s] [] | s == s' = True
7    where
8      -- Retrieve start symbol
9      ((_, s', _):_) = g
10 -- Shift or reduce
11 steps g z i = shift || reduce
12   where
13     -- Shift terminal from input to stack
14     shift = not (null i) && steps g (T (head i) : z) (tail i)
15     -- Reduce prefix on stack to nonterminal
16     reduce = not (null zs) && or (map (λ z → steps g z i) zs)
17       where
18         -- Retrieve relevant reductions
19         zs = [ N n : drop l z
20              | (_, n, rhs) ← g,
21                let l = length rhs,
22                take l z == reverse rhs ]
```

The implementation is based on these ideas:

- The program maintains a parser stack, which is represented simply as a list of grammar symbols. The head of the list is the top of the stack. We start from the empty stack (line 2).
- The regular termination case is that the input is empty and the start symbol is the sole element on the stack and, thus, True is returned (line 6). The start symbol is assumed here to be the left-hand side of the first rule (line 9).
- Otherwise, shift and reduce actions are tried and combined by "||" (line 11). The shift action is tried first (line 14) and all possible reduce actions are tried afterwards (line 16), as encoded in the order of the operands of "||".
- Possible reduce actions are determined by trying to find (reversed) right-hand sides of rules on the stack; see the list comprehension computing zs (lines 19–22). The options are combined by "or".

This implementation is naive, just as much as the earlier implementation of top-down acceptance. For one thing, the options for shift and reduce are tried in a way that a huge search space is explored. More seriously, we face the potential of nontermination again. Left recursion is not a problem this time around, but nontermination may be caused by *epsilon productions* – this is when a rule has an empty right-hand side. Nontermination can arise because any stack qualifies for application of a reduce action with an empty list of grammar symbols. The original grammar for BNL does

indeed contain an epsilon production [integer]. The following variation is needed to be able to use the naive implementation of bottom-up acceptance.

Illustration 7.7 (BNL grammar without epsilon productions)

Haskell module Language.BGL.SampleWithoutEpsilon

```
bnlGrammar :: Grammar
bnlGrammar = [
  ("integer", "number", [N "bits", N "rest"]),
  ("integer", "number", [N "bits", T '.', N "bits"]),
  ("single", "bits", [N "bit"]),
  ("many", "bits", [N "bit", N "bits"]),
  ("zero", "bit", [T '0']),
  ("one", "bit", [T '1'])
 ]
```

There are various algorithms and grammar classes that cater for efficient bottom-up parsing without termination issues [20, 1].

7.2.1.3 Top-Down Parsing

We move now from acceptance to parsing. Thus, we need to construct CSTs during acceptance. CSTs are represented in Haskell as follows.

Illustration 7.8 (CSTs for BGL)

Haskell module Language.BGL.CST

```
type Info = Either Char Rule
type CST = Tree Info
```

We use Haskell's library type Tree for node-labeled rose trees, i.e., trees with any number of subtrees. The labels (infos) are either characters for the leaf nodes or grammar rules for inner nodes. Top-down parsing is implemented in Haskell as follows.

Illustration 7.9 (Implementation of top-down parsing)

Haskell module Language.BGL.TopDownParser

```
parse :: [Rule] → String → Maybe CST
parse g i = do
    (i', t) ← tree g (N s) i
    guard (i'==[])
    return t
  where
    -- Retrieve start symbol
```

```
    ((_, s, _):_) = g

tree :: [Rule] → GSymbol → String → Maybe (String, CST)
−− Consume terminal at top of stack from input
tree _ (T t) i = do
  guard ([t] == take 1 i)
  return (drop 1 i, Node (Left t) [])
−− Expand a nonterminal
tree g (N n) i = foldr mplus mzero (map rule g)
  where
    −− Try different alternatives
    rule :: Rule → Maybe (String, CST)
    rule r@(_, n', rhs) = do
      guard (n==n')
      (i', cs) ← trees g rhs i
      return (i', Node (Right r) cs)

−− Parse symbol by symbol, sequentially
trees :: [Rule] → [GSymbol] → String → Maybe (String, [CST])
trees _ [] i = return (i, [])
trees g (s:ss) i = do
  (i', c) ← tree g s i
  (i'', cs) ← trees g ss i'
  return (i'', c:cs)
```

In this implementation, we do not model the parser stack explicitly, but we leverage Haskell's stack for function applications. This happens to imply that we are limited here to local backtracking. Thus, the parser is less complete than the acceptor implemented earlier (Section 7.2.1.1).

7.2.1.4 Bottom-Up Parsing

Exercise 7.4 (Bottom-up parsing in Haskell) [Intermediate level]
Implement bottom-up parsing in Haskell.

7.2.2 Recursive Descent Parsing

Grammars can be implemented programmatically in a systematic manner. Recursive descent parsing [1, 20] is a popular encoding scheme where grammars are implemented as recursive "procedures". Recursive descent parsing is relatively popular, as (some form of) it is often used, when a handwritten parser is implemented. Further, the overall scheme is also insightful, as it may be used for program generation – this

is when the code for a top-down parser implementation is generated directly from a grammar (Section 7.2.3).

In Java, some grammar rules for numbers in the BNL language would be represented by procedures (methods) as follows:

```
// [number] number : bits rest ;
void number() {
  bits();
  rest();
}
// [zero] bit : '0' ;
// [one] bit : '1' ;
void bit() {
  if (next == '0') match('0'); else match('1');
}
...
```

That is, there is a method for each nonterminal. Occurrences of nonterminals on right-hand sides of grammar rules are mapped to method calls. Occurrences of terminals are mapped to "match" actions on the input. When selecting alternatives (see bit), we may look ahead into the input.

Here is a more detailed description:

- Each nonterminal of the grammar is implemented as a possibly recursive procedure (a function). Success or failure of parsing may be communicated by the return value or by means of an exception. (The exception-based approach is assumed in the illustrative Java source code shown above.)
- A sequence of grammar symbols is mapped to a sequence of "actions" as follows.
  - A terminal is mapped to a "match" action to examine the head of the input stream. If the terminal is present, then it is removed and the action completes successfully. If the terminal is not present, then parsing fails.
  - A nonterminal is mapped to a procedure call (a function application). This call (application) may succeed or fail in the same sense as matching may succeed or fail.
- It remains to deal with alternatives.
  - *Parsing with look-ahead*: The procedures contain conditions on the prefix of the input stream to select the alternative to be tried. This technique can be used, for example, with $LL(k)$ [20] or $LL(*)$ [47] grammars.
  - *Parsing with backtracking*: The different alternatives are tried until one succeeds, if any does; the pointer in the input stream is reset to where it was when a failing branch was entered.

The following Haskell code represents a recursive descent parser for BNL where backtracking is supported with the help of the Maybe monad. The original grammar rules are shown as Haskell comments next to the corresponding Haskell functions.

Illustration 7.10 (Recursive descent with backtracking for BNL)

Haskell module Language.BNL.BacktrackingAcceptor

```
-- Accept and enforce complete input consumption
accept :: String → Bool
accept i = case number i of
  Just [] → True
  _ → False

-- Functions for nonterminals
number, bits, bit, rest :: String → Maybe String

-- [number] number : bits rest ;
number i = bits i >>=rest

-- [single] bits : bit ;
-- [many] bits : bit bits ;
bits i = many `mplus` single
  where
    single = bit i
    many = bit i >>=bits

-- [zero] bit : '0' ;
-- [one] bit : '1' ;
bit i = zero `mplus` one
  where
    zero = match '0' i
    one = match '1' i

-- [integer] rest : ;
-- [rational] rest : '.' bits ;
rest i = rational `mplus` integer
  where
    integer = Just i
    rational = match '.' i >>=bits

-- Match a terminal (a character)
match :: Char → String → Maybe String
match t (t':i) | t == t' = Just i
match _ _ = Nothing
```

The parser can be used as follows:

Interactive Haskell session:

```
► accept "101.01"
True
```

The encoding is based on just a few ideas:

- Each nonterminal is modeled as a function that takes the input string and returns, maybe, the remaining input string. If a function returns Nothing, then this models failure of parsing.
- Sequential composition of grammar symbols, as prescribed by the grammar rules, is modeled by the bind combinator ">>=" of the Maybe monad; in this manner, input strings are processed from left to right.
- There is a general match function which just tries to match a given terminal with the head of the input and either succeeds or fails, as described earlier for the terminal case in recursive descent parsing.
- Different alternatives for a nonterminal are combined by the mplus combinator of the Maybe monad; this implies left-biased choice, i.e., the left operand is tried first and the right operand is tried only in the case of failure for the left operand.

The implementation, as it stands, is limited to local backtracking because alternatives are combined by mplus. As noted before (Section 7.2.1.1), acceptance may be thus incomplete depending on the order of the rules. For comparison, let us also look at an acceptor that uses look-ahead instead of backtracking.

Illustration 7.11 (Recursive descent with look-ahead for BNL)

Haskell module Language.BNL.LookAheadAcceptor

```
-- [single] bits : bit ;
-- [many] bits : bit bits ;
bits i = if lookahead 2 (flip elem ['0','1']) i
           then many
           else single
  where
    single = bit i
    many = bit i >>=bits

-- [zero] bit : '0' ;
-- [one] bit : '1' ;
bit i = if lookahead 1 ((==) '0') i
          then zero
          else one
  where
    zero = match '0' i
    one = match '1' i

-- [integer] rest : ;
-- [rational] rest : '.' bits ;
rest i = if lookahead 1 ((==) '.') i then rational else integer
  where
    integer = Just i
    rational = match '.' i >>=bits

-- Look ahead in input; avoid looking beyond end of input
lookahead :: Int → (Char → Bool) → String → Bool
lookahead l f i = length i >= l && f (i!!(l−1))
```

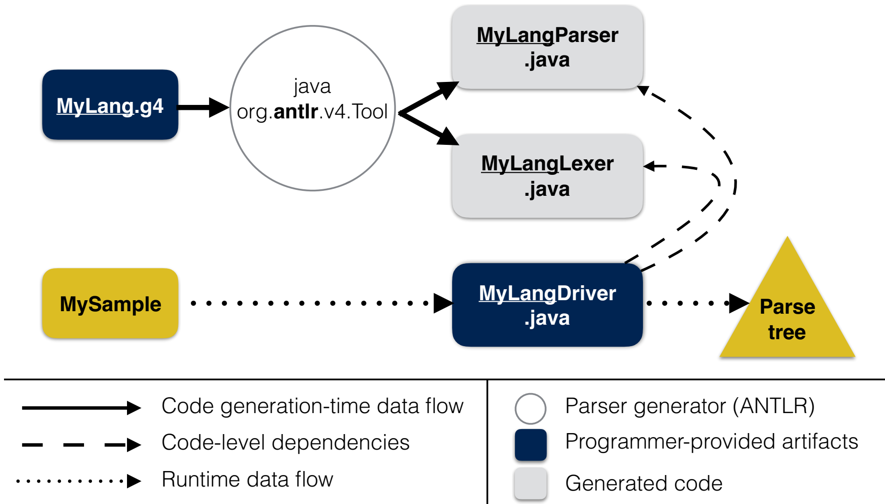


Fig. 7.2 Parser generation with ANTLR with Java as the target language: the data flow at parser-generation time and at runtime of a generated parser is shown.

That is, the functions use an additional function lookahead to perform tests on the input, thereby guarding the different branches for nonterminals with multiple alternatives.

7.2.3 Parser Generation

A popular approach to parser implementation is parser generation. The overall idea is to generate code or data structures from a (possibly enriched) grammar such that the generation process can perform some amount of grammar checking and manipulation, for example, with regard to grammar-class restrictions. Also, extra services may be provided, for example, handling parser errors by means of error messages and error recovery. We briefly discuss ANTLR here as an example of a parser generator. In Chapter 2, we have already applied ANTLR to FSML, thereby deriving a syntax checker and a parser based on walking ANTLR's CST with a listener.

Figure 7.2 summarizes the data flow for generating a parser with ANTLR [46] and using the generated parser. We focus here on Java as the target language for code generation; ANTLR also provides other backends. The input for parser generation is an ANTLR grammar (see the ".g4" file in the figure). Parser generation returns several files; we only care here about the parser and lexer files (see the ".java" files marked as generated code in the figure). Subject to some routine driver code to be provided by the developer, the generated code can be used to parse text into an

ANTLR-style parse tree (CST). In Section 7.3.2, we will revisit ANTLR and discuss how to add semantic actions to a grammar for constructing ASTs.

7.2.4 Parser Combinators

Another popular approach to parser implementation is based on parser combinators [52, 26, 24]. The simple (intriguing) idea is to model parsers as instances of an abstract data type (ADT) with function combinators that correspond to "constructs" for syntax definition. In particular, we need combinators to cover these cases:

- Terminals
- Sequential composition
- Choice (composition of alternatives)
- EBNF constructs ("?", "*", "+")

Nonterminals are modeled as (possibly recursive) functions – pretty much in the same way as in the case of recursive descent parsing (Section 7.2.2).

Let us demonstrate the use of the popular parser-combinator library *Parsec*[2] [37] in the Haskell context. Here is an acceptor for the FSML language.

Illustration 7.12 (A parser combinator-based acceptor for FSML)

Haskell module Language.FSML.Acceptor

```
fsm = many state
state =
      optional (reserved "initial")
   >> reserved "state"
   >> stateid
   >> braces (many transition)
transition =
      event
   >> optional (op "/" >> action)
   >> optional (op "−>" >> stateid)
   >> semi
stateid = name
event = name
action = name
```

These combinators are used in the example:

- >>: Sequential composition
- many: EBNF's "*";
- optional: EBNF's "?";
- reserved: reserved keywords (provided by scanner);

[2] https://wiki.haskell.org/Parsec

- braces: constructs enclosed in $\{\cdots\}$;
- op: operator symbols (provided by scanner);
- semi: ";" (provided by scanner);
- name: names or identifiers (provided by scanner).

Choice (composition of alternatives) is not present in this simple example, but there is, of course, a corresponding binary combinator "<|>" for left-biased choice.

In the case of Parsec, scanners (lexers) are also just parsers, in terms of the underlying ADT. Technically, scanning and parsing are separated. This simplifies the process of skipping white space, recognizing (and skipping) comments, and handling reserved keywords and special characters in a special manner – also in the interest of enabling "good" error messages. The lexer for FSML is derived from a default lexer as follows.

Illustration 7.13 (A lexer for FSML)

Haskell module Language.FSML.Lexer

```
fsmlDef :: Token.LanguageDef ()
fsmlDef = emptyDef
    { Token.commentStart = "/*"
    , Token.commentEnd = "*/"
    , Token.commentLine = "//"
    , Token.identStart = letter
    , Token.identLetter = alphaNum
    , Token.nestedComments = True
    , Token.reservedNames = ["initial", "state"]
    , Token.reservedOpNames = ["/", "->"]
    }

lexer :: Token.TokenParser ()
lexer = Token.makeTokenParser fsmlDef

braces :: Parser p → Parser p
braces = Token.braces lexer

semi :: Parser String
semi = Token.semi lexer

reserved :: String → Parser ()
reserved = Token.reserved lexer

op :: String → Parser ()
op = Token.reservedOp lexer

name :: Parser String
name = Token.identifier lexer
```

Thus, the definition of the lexer entails the provision of certain parameters such as the start and end sequences for comments, the initial characters of an identifier, and

the list of reserved names. Thereby, several lexical categories are readily defined, such as those used in the earlier acceptor.

The ADT for parsing provides a special operation, runParser, for applying the parser to an actual input string. This is demonstrated here at the lexical level:

Interactive Haskell session:

```
-- Recognize a name; this is Ok.
▶ runParser name () "" "foo"
Right "foo"
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Recognize two names in a sequence; this is Ok.
▶ runParser (name >> name) () "" "foo bar"
Right "bar"
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Try to recognize a name; this fails because "state" is reserved.
▶ runParser name () "" "state"
Left (line 1, column 6):
unexpected reserved word "state"
expecting letter or digit
```

The function runParser takes four arguments of which we are only interested here in the first one (i.e., the actual parser) and the last one (the input string). Running a parser returns either an error message (see Left $\cdots$) or a parse tree (see Right $\cdots$) such as a string in the two examples above.

In Section 7.3.3, we will return to the parser-combinator approach and discuss how the construction of ASTs can be accomplished with this approach. Until now we have limited ourselves to acceptance.

Exercise 7.5 (Layout in practice) [Intermediate level]

Study some grammar notation, for example, YACC/LEX [25], SDF [57], or ANTLR [46], with regard to the definition of lexical units and the handling of layout (white space and comments). Explain and illustrate your findings.

7.3 Abstraction

We turn now to the problem of how to construct appropriate ASTs during parsing. In this manner, we effectively model relationships between concrete and abstract syntax, as discussed earlier (Section 6.2). To this end, we will revisit the grammar implementation approaches that we have seen above and enhance them accordingly. Such a mapping is a diverse and complex topic in software language engineering (see, e.g., [28, 27, 48, 62, 22]).

7.3.1 Recursive Descent Parsing

Here we generalize the scheme presented in Section 7.2.2. The key idea is that "procedures" (in our case, "functions") are assumed to construct and return ASTs of the appropriate type. In our Haskell-based approach, we capture a type constructor for the signature of functions that model nonterminals:

```
type Parser a = String → Maybe (a, String)
```

That is, the input string is mapped either to Nothing or to a pair consisting of an AST (of type a) and the remaining input string. Here is the parser for FSML.

Illustration 7.14 (A recursive descent parser for BNL in Haskell)

Haskell module Language.BNL.Parser

```
-- [number] number : bits rest ;
number :: Parser Number
number i = do
  (bs, i') ← bits i
  (r, i'') ← rest i'
  Just (Number bs r, i'')

-- [single] bits : bit ;
-- [many] bits : bit bits ;
bits i = many `mplus` single
  where
    single = do (b, i') ← bit i; Just (Single b, i')
    many = do (b, i') ← bit i; (bs, i'') ← bits i'; Just (Many b bs, i'')

-- [zero] bit : '0' ;
-- [one] bit : '1' ;
bit i = zero `mplus` one
  where
    zero = do i' ← match '0' i; Just (Zero, i')
    one = do i' ← match '1' i; Just (One, i')

-- [integer] rest : ;
-- [rational] rest : '.' bits ;
rest i = rational `mplus` integer
  where
    integer = Just (Integer, i)
    rational = do
      i' ← match '.' i
      (bs, i'') ← bits i'
      Just (Rational bs, i'')
```

The differences between a parser and an acceptor (Section 7.2.2) can be summarized as follows. A parser may return an AST; an acceptor is just a predicate. In the parser, we use "do" notation for convenience of sequential composition of actions.

In particular, we bind intermediate ASTs in this way. The encoding of each rule ends in an expression of the form “Just ···” to explicitly compose and return the AST of interest. For instance, consider this:

```
-- [number] number : bits rest ;
number :: Parser Number
number i = do
  (bs, i') ← bits i
  (r, i'') ← rest i'
  Just (Number bs r, i'')
```

That is, the rule for the nonterminal is encoded by a sequential composition so that we bind ASTs bs and r and construct the AST “Number bs r”. Along the way, we thread the input (see i, i', and i'').

Exercise 7.6 (A parser monad) [Intermediate level]
Define an appropriate parser monad (or an applicative functor for parsing) which includes tracking of input and potential failure. Rewrite the recursive descent parser in Illustration 7.14 to use this monad. In this manner, we should arrive at more factored code so that the functions for the nonterminals do not need to pass the input explicitly; this would be taken care of by the bind operation of the monad.

7.3.2 Semantic Actions

In Section 2.3.3, we have already discussed the use of object-oriented listeners for walking a CST such that functionality for AST construction can be defined. We adopted ANTLR accordingly. It is common that parser generators also provide the option of injecting so-called semantic actions into a grammar so that computations can be performed during parsing. A major use case of semantic actions is indeed AST construction. A semantic action is basically some statement of the target language for parser generation; the statement is to be executed when parsing reaches the position of the semantic action within the grammar rule.

Let us inject Java code for AST construction into an ANTLR-based syntax definition.

Illustration 7.15 (An ANTLR-based parser description for FSML)

ANTLR resource languages/FSML/Java/FsmlToObjects.g4

```
1 grammar FsmlToObjects;
2 @header {package org.softlang.fsml;}
3 @members {public Fsm fsm = new Fsm();}
4
5 fsm : state+ EOF ;
6 state :
```

```
7   { boolean initial = false; }
8   ('initial' { initial = true; })?
9   'state' stateid
10  { fsm.getStates().add(new State($stateid.text, initial)); }
11  '{' transition* '}'
12  ;
13 transition :
14  { String source = fsm.getStates().get(fsm.getStates().size()−1).getStateid(); }
15  event
16  { String action = null; }
17  ('/' action { action = $action.text; })?
18  { String target = source; }
19  ('−>' stateid { target = $stateid.text; })?
20  { fsm.getTransitions().add(new Transition(source, $event.text, action, target)); }
21  ';'
22  ;
23 stateid : NAME ;
24 event : NAME ;
25 action : NAME ;
26 NAME : ('a'..'z'|'A'..'Z')+ ;
27 WS : [ \t\n\r]+ −> skip ;
```

ANTLR's semantic actions can be briefly explained as follows:

- Semantic actions are injected into the grammar by escaping them with braces. For instance, the semantic action {boolean initial=true;} (line 7) declares and initializes a local Java variable (local to the generated code for the state rule).
- Nonterminals are associated with the text consumed by parsing. If n is a nonterminal in a given rule, then $\$n.text$ can be used to access the text within semantic actions; see, for example, {...$stateid.text ...} (line 10).

The semantic actions in the example build objects of a basic, influent object model for FSML (Section 2.2.1). The following members are used in the semantic actions:

- The constructor for Fsm is invoked to construct an FSM object and keep track of it through an attribute fsm that is injected into the generated class (line 3).
- The observer members getStates and getTransitions are invoked to access collections of states and transitions (lines 10 and 20).
- The constructors for State and Transition are invoked to construct objects for states and transitions and to add them to the FSM object (line 10 and line 20).

ANTLR and other parser generators have more sophisticated mechanisms for semantic actions. In particular, ANTLR makes it possible to declare arguments and results for nonterminals. These mechanisms are inspired by the attribute grammar paradigm [32, 40] which we discuss in Section 12.2.

7.3.3 Parser Combinators

Parser combinators, as discussed in Section 7.2.4, can be used to represent grammars as systems of recursive functions for parsing. The body of each function is basically an expression over parser combinators for sequential composition, choice (composition of alternatives), EBNF operators for optionality and iteration, and constants for scanning.

Let us demonstrate the use of the popular parser-combinator library Parsec [37] in the Haskell context. Parsers composed by combinators are of an abstract data type Parser a, where a is the type of representation (AST) constructed by the parser. The Parser type constructor is a monad [60] and the bind operator ">>=" is immediately the parser combinator for sequential composition.

AST construction basically boils down to the application of appropriate data constructors to "smaller" ASTs. In basic monadic style, this means that the ASTs for phrases are bound via ">>=" or monadic do-notation with a final return to compose an AST, as shown below.

Illustration 7.16 (A monadic style parser for FSML)

Haskell module Language.FSML.MonadicParser

```
fsm :: Parser Fsm
fsm = many state >>=return . Fsm

state :: Parser State
state = do
  ini ← option False (reserved "initial" >> return True)
  source ← reserved "state" >> stateid
  ts ← braces (many (transition source))
  return (State ini source ts)

transition :: StateId → Parser Transition
transition source = do
  e ← event
  a ← optionMaybe (op "/" *> action)
  t ← option source (op "−>" *> stateid)
  semi
  return (Transition e a t)
```

The functions fsm, state, and transition are parsers with corresponding algebraic data types Fsm, State, and Transition for the ASTs. The function Transition is parameterized by a state id for the source state to which the transition belongs; the state id is used as the target state id if the id was omitted in the FSM.

We may also use the more restrained applicative functor style [41] where applicable. That is, we leverage the fact that the parser monad is also an applicative functor and thus, an AST constructor can essentially be applied to the computations for the phrases. Arguably, this leads to more "functional" code as shown below.

Illustration 7.17 (An applicative functor style parser for FSML)

Haskell module Language.FSML.ApplicativeParser

```
fsm :: Parser Fsm
fsm = Fsm <$> many state

state :: Parser State
state = do
  ini ← option False (reserved "initial" >> return True)
  source ← reserved "state" >> stateid
  ts ← braces (many (transition source))
  return (State ini source ts)

transition :: StateId → Parser Transition
transition source =
  Transition
    <$> event
    <*> optionMaybe (op "/" *> action)
    <*> option source (op "−>" *> stateid)
    <* semi
```

The functions fsm and transition are defined in applicative functor style whereas we resort to monadic style (do-notation) in the case of state because we need to intercept the state id so that it can be passed as an argument to transition.

7.3.4 Text-to-Model

For brevity, we will not discuss in any detail the problem of text-to-model transformations [27, 22], a parsing-like phase in the MDE. Conceptually, a text-to-model transformation can be viewed as consisting of parsing followed by abstraction and resolution (AST-to-ASG mapping), as discussed previously (Section 4.4). Technically or technologically, this is an involved and interesting problem.

Exercise 7.7 (Text-to-model with Xtext) [Intermediate level]
Study Xtext[3] [4] and implement a mapping from text-based syntax for the Buddy Language to a graph-based representation.

[3] http://www.eclipse.org/Xtext/

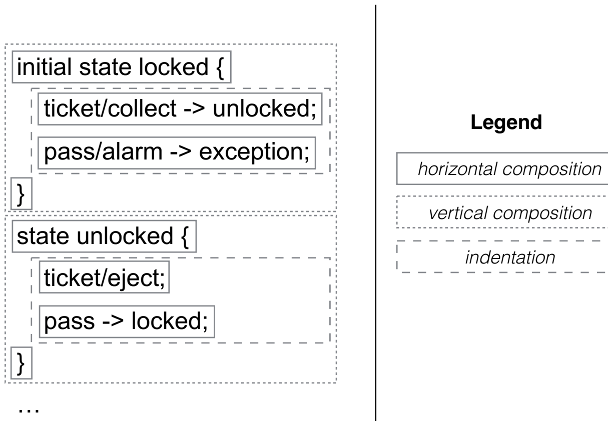


Fig. 7.3 Formatting with pretty-printer combinators: the FSM on the left-hand side is readily formatted. The boxes explain how the format has emerged from the composition of boxes by horizontal and vertical alignment and by indentation.

7.4 Formatting

We now switch from parsing to the opposite direction: formatting (or unparsing or pretty printing). Formatting is needed in the context of code generation, for example, in web programming; it also shows up as part of source-code transformation if we assume that the actual transformation is implemented at the level of ASTs and, thus, the transformation is complemented by parsing and formatting.

We describe two overall options for formatting: the combinator- versus the template-based approach. Both of these approaches are syntax-driven: formatting is essentially accomplished by recursing into an AST-like data structure. We do not discuss more lexical approaches to formatting any further here (see, e.g., [44, 2]). We also omit coverage of the interesting notion of invertible syntax descriptions such that parsing and formatting (pretty printing) can be unified [49].

7.4.1 Pretty Printing Combinators

The overall idea of the combinator-based approach is to view the output as a document that is composed from smaller documents, down to the level of pieces of text, by means of combinators for horizontal and vertical composition as well as indentation [5]; see Fig. 7.3 for an illustration. This approach is fundamentally different from the more ad hoc approach where a programmer takes care of line breaks and indentation in, say, an imperative manner, i.e., by maintaining the indentation level in a variable and producing textual output by means of "printf".

Let us demonstrate such pretty printing in Haskell with the help of a suitable combinator library [23]. There are combinators such as this:

Interactive Haskell session:

```
▶ −− Switch to the module for formatting
▶ :m Text.PrettyPrint.HughesPJ
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− The empty box
▶ :t empty
empty :: Doc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− A box exactly containing some given text
▶ :t text
text :: String → Doc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− Horizontal composition of two boxes
▶ :t (<>)
(<>) :: Doc → Doc → Doc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− Space−separated horizontal composition
▶ :t (<+>)
(<+>) :: Doc → Doc → Doc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− Vertical composition of two boxes
▶ :t ($$)
($$) :: Doc → Doc → Doc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− Vertical composition of a list of boxes
▶ :t vcat
vcat :: [Doc] → Doc
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
▶ −− Indentation of a box by a number of spaces
▶ :t nest
nest :: Int → Doc → Doc
```

The type Doc is an abstract data type; one may turn documents into text:

Interactive Haskell session:

```
▶ show $ text "hello"
"hello"
```

The combinators satisfy some convenient algebraic laws. For instance, an empty box is a left and right unit of (even space-separated) horizontal composition. Thus:

```
empty <> x = x
x <> empty= x
empty <+> x = x
x <+> empty = x
```

We are ready to present a formatter (a pretty printer) for FSML. In the following Haskell code, we assume that an FSM is given in the abstract syntactical representation, as introduced earlier.

Illustration 7.18 (Formatting FSML with pretty printer combinators)

Haskell module Language.FSML.CombinatorFormatter

```
fsm :: Fsm → Doc
fsm (Fsm ss) = vcat (map state ss)

state :: State → Doc
state (State initial source ts) =
  (if initial then text "initial" else empty)
  <+> text "state"
  <+> text source
  <+> text "{"
  $$ nest 2 (vcat (map (transition source) ts))
  $$ text "}"

transition :: String → Transition → Doc
transition source (Transition ev ac target) =
  text ev
  <> maybe empty (λ ac'→ text "/" <> text ac') ac
  <+> (if source == target
          then empty
          else text "−>" <+> text target)
  <> text ";"
```

Thus, we designate a function for each type of the abstract syntax. The states are formatted independently and the resulting boxes are vertically composed; see the use of vcat in the function fsm (line 2). Each state is formatted by a mix of horizontal and vertical composition. The vertically composed transitions are indented; see the use of nest in the function state (line 10). The function transition is parameterized by the source state id so that it can leave out the target state id if it equals the source state id (lines 17–19).

7.4.2 Template Processing

A template is essentially a parameterized text (a string). Several related templates may be organized into groups of named templates so that they can invoke each other along with parameter passing. More abstractly, a group of templates can be viewed as a mutually recursive system of functions that map parameter data to text on the basis of filling parameters or projections thereof into “holes” in the templates. This process may also be controlled by conditions and may involve “loops” to iterate over parameters that represent lists. In Section 2.4.2s, we already discussed the use of template processing for code generation, i.e., mapping one language (e.g., FSML) to another language (e.g., C). In the present section, we discuss the use of template processing for formatting, i.e., mapping the abstract syntax of a language to the concrete syntax of the same language.

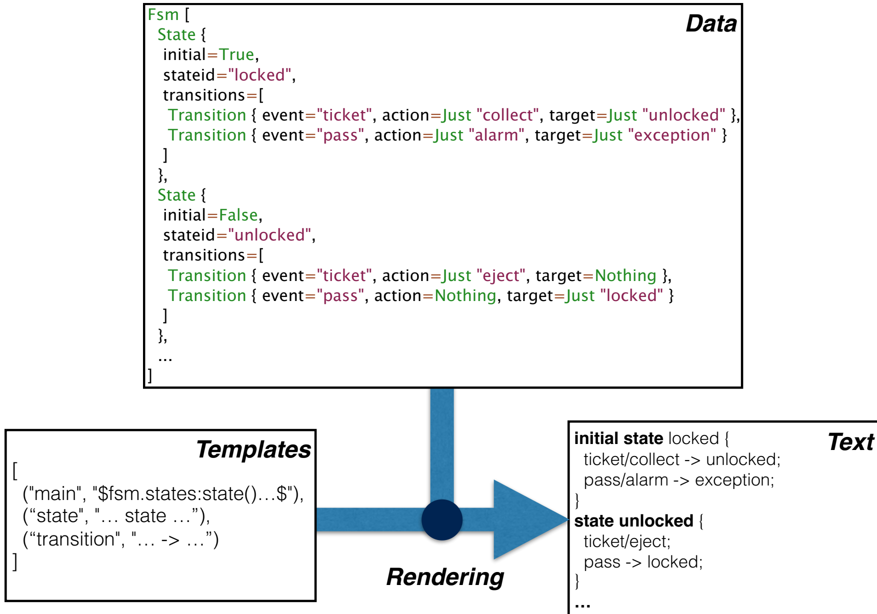


Fig. 7.4 I/O behavior of template processing for FSML.

A template processing-based formatter for FSML is illustrated in Fig. 7.4. Let us make some assumptions about the template-processing problem at hand:

- We assume the abstract syntactical representation for FSMs, as introduced earlier. This abstract representation provides the initial parameter for template processing.
- We need templates for the FSM as a whole, for states, and for transitions, as an FSM breaks down into many states, each of which breaks down into many transitions.
- Lines breaks and indentation are captured by the templates. Alternatively, a lexical formatter [44, 2] could be applied for postprocessing.
- Template parameters should be set up in such a way that no computations need to be performed during formatting, thereby separating the concerns of computation and formatting.

We are going to leverage the template processor *HStringTemplate*[4] for Haskell; this is a Haskell port of the well-known template processor *StringTemplate*[5] [45] which we used in Section 2.4.2.

[4] https://hackage.haskell.org/package/HStringTemplate

[5] http://www.stringtemplate.org/

Illustration 7.19 (Formatting FSML with template processing)

Haskell module Language.FSML.TemplateFormatter

```
templates :: STGroup String
templates = groupStringTemplates [
    ("main", newSTMP "$fsm.states:state(); separator='\n'$"),
    ("state", newSTMP $ unlines [
        "$if(it.initial)$initial $endif$state $it.stateid$ {",
        "$it.transitions:transition(); separator='\n'$",
        "}"
      ]
    ),
    ("transition", newSTMP (
          " $it.event$\
          \$if(it.action)$/$it.action$$endif$\
          \$if(it.target)$ −> $it.target$$endif$\
          \;"
      )
    )
  ]

format :: Fsm → String
format fsm =
   let Just t = getStringTemplate "main" templates
     in render $ setAttribute "fsm" fsm t
```

The formatter captures the template group (lines 1–17) as a list of name-value pairs. The values are strings which are spread out over multiple lines for readability (see the multiline strings in the *"state"* template in line 4) or for insertion of line breaks (see the use of unlines in the *"transition"* template in lines 10–15). The format function retrieves the *"main"* template, sets the *"fsm"* attribute, and starts rendering (lines 19-22). Within the templates, the following idioms are used:

- Parameter references and template invocations are included within "$\$\cdots\$$".
- Components of parameters are selected by member-like access; see the use of ".".
- A template t is invoked by the idiom "$\$\ldots t(\cdots)\$$"; parameters, if any, are passed between the parentheses.
- Lists can be processed by an idiom "$\$l : t(\cdots); separator = \cdots\$$" where l denotes access to a list-typed parameter, t is the name of the template to be invoked on each element, and the separator declaration can be used for injecting line breaks or other separators between the rendered elements. Within the template t, the element is referable to as the parameter "it"; see the uses of "it" in the templates.
- Conditional text may be modeled by the "$\$\mathbf{if}(\cdots)\$\cdots\$\mathbf{endif}\$$" idiom. There are these forms of condition: test of a Boolean parameter, test for the presence of an "optional" parameter (a Maybe in Haskell), and possibly others.

A general discussion of the features of template processing can be found elsewhere [50]; the discussion is systematic and technology-neutral, but it assumes the perspective of model-to-text transformation.

A valuable property of templates is that they encourage a separation between "model" and "view" (say, computation and formatting) by means of a simple mapping of a data structure, passed as a parameter, to text – as opposed to performing "arbitrary" computations along with text generation. A problematic property of (mainstream) template processing is that there is no static guarantee that the resulting text will be syntactically correct, and even less so that the result will be well-typed. Syntactic correctness can be achieved by constraining template systems in such a manner that templates are derived from a given grammar in a systematic manner [59].

7.5 Concrete Object Syntax

So far, we have basically assumed that metaprograms operate at the level of abstract syntactical object program representations. In this setting, if the concrete syntax of the object language is to be used, then metaprograms need to be complemented by parsing and formatting. Alternatively, subject to suitable metaprogramming support, metaprograms can operate directly at the level of concrete object syntax. In this case, the metaprogrammer can use the patterns of the object language in metaprograms. For instance, a translator from metamodels to database schemas would directly use metamodeling and SQL syntax. Likewise, the implementation of a refactoring suite for a given programming language such as Java would directly use programming language syntax of that language.

The notion of concrete object syntax has been developed in specialized metaprogramming systems over the last 30+ years (see [55, 58, 9, 10] for somewhat more recent accounts).

There is a poor man's approach towards using concrete object syntax in which object programs are encoded as strings in the metalanguage. This approach is used, for instance, in low-level database APIs such as JDBC for Java. Consider this illustration, quoted from [6, 7]; the Java code runs an SQL query to check a username/-password pair:

```
String userName = getParam("userName");
String password = getParam("password");
String query = "SELECT id FROM users "
             + "WHERE name = '" + userName + "' "
             + "AND password = '" + password + "'";
if (executeQuery(query).size() == 0)
   throw new Exception("bad user/password");
```

An obvious drawback of this poor man's approach is that the proper use of the object language's syntax is not checked at compile time. Syntax errors and issues with conformance of the query to the underlying database schema would only be

found at runtime. Perhaps a less obvious consequence of such poor checking is that programs become vulnerable to injection attacks [6, 7].

In this section, we focus on proper syntax-aware embedding of the object language into the metalanguage. In an extended Java language with SQL embedded, the above example may look as follows; this code is again adopted from [6, 7]:

```
SQL q = <| SELECT id FROM users WHERE
     name = ${userName} AND password = ${password} |>;
if (executeQuery(q.toString()).size() == 0) ...
```

The key idea is that, within the metalanguage (here: Java), we can embed object program fragments (here: SQL) by means of an appropriate escaping or quoting mechanism (see the brackets “<| ··· |>”) and we can escape back to the metalanguage to fill in details computed in the metaprogram (see the access to Java variables such as “${userName}”. Thus, the syntax of the object language and the metalanguage are amalgamated in a certain manner.

7.5.1 *Quotation*

We will discuss here an approach to concrete object syntax which combines so-called quasi-quotation and language or syntax embedding [39, 53, 61]. We begin with a trivial example. Consider the following Haskell code which exercises embedding FSML syntax into Haskell.

Illustration 7.20 (Embedding of FSML into Haskell)

Haskell module Language.FSML.QQ.Sample

```
turnstileFsm :: Fsm
turnstileFsm = [fsml|
        initial state locked {
              ticket / collect → unlocked;
              pass / alarm → exception;
        }
        state unlocked {
              ticket / eject;
              pass → locked;
        }
        state exception {
              ticket / eject;
              pass;
              mute;
              release → locked;
        }
|]
```

We use so-called quasi-quote brackets "[fsml| ··· |]" (or Oxford brackets) to *quote* an FSM within the Haskell code. Quasi-quotation is realized (in Haskell) such that the quoted text is actually parsed at compile time. What happens underneath is that the parser synthesizes an AST based on the algebraic data type-based, abstract syntactical representation of the parsed language and the resulting expression is then mapped to a Haskell expression (AST) and inserted into the AST of the module. Thus, the shown binding has exactly the same meaning as if we had written Haskell code for FSM construction instead. This can be compared with storing the FSM in a file and parsing it at runtime – except that quasi-quotation allows us to embed the FSM directly into the Haskell code and parsing (syntax checking) happens transparently at compile time.

The quasi-quote brackets specify the language to be used; this is fsml in the example. The name is, in fact, the name of a binding of type QuasiQuoter, subject to Haskell's extension for quasi-quotation [39] based also on Template Haskell [51]. A quasi-quoter essentially describes how to map strings to Haskell expressions or elements of other categories of Haskell's syntax. We are ready to present the quasi-quote for FSML.

Illustration 7.21 (A quasi-quoter for FSML)

Haskell module Language.FSML.QuasiQuoter

```
fsml :: QuasiQuoter
fsml = QuasiQuoter
    { quoteExp = quoteFsmlExp
    , quotePat = undefined
    , quoteType = undefined
    , quoteDec = undefined
    }

quoteFsmlExp :: String → Q Exp
quoteFsmlExp str = do
    x ← parseQ fsm str
    case check x of
      [] → dataToExpQ (const Nothing) x
      errs → error $ unlines errs
```

Thus, the quasi-quoter (lines 1–7) is a record with four components, each one applying to a different syntactic category. We are only concerned with expressions here (see quoteExp = ··· in line 3) and we leave the components for patterns, types, and declarations undefined. Let us describe the actual binding for quoteFsmlExp (lines 9–14) in terms of three phases:

Parsing The expression parseQ fsm str (line 11) parses the string str between the quasi-quote brackets; it uses a standard parser fsm (Section 7.3.3). The function parseQ is a convenience wrapper around the normal run function of the parser monad; it sets up location and error reporting in a uniform manner.

Analysis The expression check x (line 12) checks that the parsed FSM satisfies the usual constraints that we set up for the FSML language earlier. If the check returns any error messages, then they are communicated to the user by the invocation of the error function. Such checking is performed inside the quotation monad and, thus, errors will be communicated to the language user in the same way as type errors for the metalanguage.

Quoting The expression dataToExp (const Nothing) x (line 13) maps an FSM (represented in the established abstract syntax) to a Haskell expression, which constructs a Haskell AST. The function dataToExp is a generic function in that it can be applied to the different types that are used in an FSM representation. The always-failing component const Nothing expresses that the generic behavior is appropriate for all relevant types.

The first two phases, *parsing* and *analysis*, are relatively obvious. The last phase, *quoting*, may require some extra reflection. That is, one may expect that the quasi-quoter should somehow be able to use the AST representation of the FSM directly, as this is exactly the AST that we want to process in a metaprogram anyway. Instead, we seem to detour through Haskell expressions only to recover the same AST at runtime. This approach is appropriate because it is more general and uniform. The uniformity assumption here is that the contents of the quasi-quote brackets denote Haskell code (expressions, patterns, types, or declarations) as opposed to ASTs of another language. Thus, the object language is integrated by translation to the metalanguage. We will demonstrate generality in a moment with another quasi-quotation experiment.

Exercise 7.8 (Parsing FSMs into Haskell expressions) [Intermediate level]
Implement an FSML parser which directly constructs Haskell expressions as opposed to the present separation of parsing FSMs into FSM ASTs and converting those ASTs into Haskell expressions.

7.5.2 Antiquotation

When quasi-quoted phrases escape back to the metalanguage, then we also speak of antiquotation. Let us consider program optimization for EL expressions again. In contrast to the earlier discussion (Section 5.4.1), we would like to use concrete object syntax to author simplification rules for EL in Haskell. In Fig. 7.5, we show some simplification rules (or laws), we recall the encoding of rules as functions on abstract syntax, and we show a new encoding which relies on concrete object syntax for EL in Haskell.

Between the quasi-quote brackets, we use antiquotation in terms of “$”-prefixed identifiers to denote metavariables of the metalanguage (i.e., Haskell). For instance,

```
-- Laws on expressions
x + 0 = x
x * 1 = x
x * 0 = 0
```

```
-- Implementation based on abstract object syntax
simplify :: Expr -> Maybe Expr
simplify (Binary Add x (IntConst 0)) = Just x
simplify (Binary Mul x (IntConst 1)) = Just x
simplify (Binary Mul x (IntConst 0)) = Just $ IntConst 0
simplify _ = Nothing
```

```
-- Implementation based on concrete object syntax
simplify :: Expr -> Maybe Expr
simplify [el| $x + 0 |] = Just [el| $x |]
simplify [el| $x * 1 |] = Just [el| $x |]
simplify [el| $x * 0 |] = Just [el| 0 |]
simplify _ = Nothing
```

Fig. 7.5 Comparison of simplification rules (or laws) with an encoding as a function on abstract syntax and an encoding as a function on concrete syntax based on quasi-quotation.

the regular Haskell pattern Binary Add x (IntConst 0) is written in concrete object syntax as [el| $x + 0|] with $x as a metavariable (i.e., Haskell variable). We need this special syntax for metavariables because, without the "$" sign, we would denote an object variable, i.e., a variable of the expression language EL according to its concrete syntax. Thus, we need an extended syntax for EL with a new case for metavariables, as defined below.

Illustration 7.22 (EL syntax extension for metavariables)

Haskell module Language.EL.QQ.Syntax

```
data Expr
  = ... -- The same syntax as before
  | MetaVar String -- An additional constructor for the abstract syntax
```

Haskell module Language.EL.QQ.Parser

```
factor :: Parser Expr
factor
  = ... -- The same syntax as before
  <|> (MetaVar <$> (op "$" >> identifier)) -- An additional choice in parsing
```

In the quasi-quoter, metavariables need to be mapped from the EL representation to proper Haskell variables as shown below.

Illustration 7.23 (A quasi-quoter for EL)

Haskell module Language.EL.QuasiQuoter

```
1  el :: QuasiQuoter
2  el = QuasiQuoter
3      { quoteExp = quoteElExp
4      , quotePat = quoteElPat
5      , quoteType = undefined
6      , quoteDec = undefined
7      }
8
9  quoteElExp :: String → Q Exp
10 quoteElExp str = do
11     x ← parseQ expr str
12     dataToExpQ (const Nothing `extQ` f) x
13   where
14     f :: Expr → Maybe (Q Exp)
15     f (MetaVar v) = Just $ varE (mkName v)
16     f _ = Nothing
17
18 quoteElPat :: String → Q Pat
19 quoteElPat str = do
20     x ← parseQ expr str
21     dataToPatQ (const Nothing `extQ` f) x
22   where
23     f :: Expr → Maybe (Q Pat)
24     f (MetaVar v) = Just $ varP (mkName v)
25     f _ = Nothing
```

We point out the following aspects of the quasi-quoter:

- The quasi-quoter instantiates the components for both Haskell expressions (line 3) and patterns (line 4) because, as demonstrated in Fig. 7.5, we would like to use concrete EL syntax in the positions of both pattern matching and the right-hand sides of equations. Accordingly, the component quoteElExp returns an expression (Exp) within the quotation monad, whereas the component quoteElPat returns a pattern (Pat).
- Both of the quasi-quotation components (lines 9–25) use the same convenience function parseQ as before.
- When mapping ("quoting") the EL AST, the generic mapping functions dataToExpQ and dataToPatQ are properly customized (lines 15 and 24) so that EL's metavariables are mapped to Haskell variables. We use the constructor VarE for names in an expression context and the constructor VarP in a pattern context.

The situation at hand is more general than that for FSML above because we have properly amalgamated the syntaxes of Haskell and EL. That is, one can use EL within Haskell and one can also use Haskell (its variables) within EL. See the following exercise for a generalized amalgamation.

Exercise 7.9 (Comprehensive antiquotation) [Intermediate level]
Extend the quasi-quoter so that antiquotation can be used with arbitrary Haskell expressions as opposed to just metavariables. To this end, you need to replace the case for metavariables with one for arbitrary Haskell expressions; the Haskell parser needs to be invoked from within the EL parser.

Exercise 7.10 (Automated object syntax embedding) [Advanced level]
Metaprogramming systems such as ASF+SDF [54], TXL [10, 11], Stratego [8], the Sugar line of systems [14, 15, 16], and Rascal [31, 30] support metaprogramming with concrete object syntax without the efforts of setting up (the equivalent of) a quasi-quoter. Develop a related infrastructure on top of Haskell's quasi-quotation on the basis of the following ideas:*

- *The grammar of the object language is extended automatically to accomplish splicing according to Exercise 7.9. To this end, an alternative for meta-expressions (patterns) has to be added for each nonterminal.*
- *The grammar is interpreted by a generic component to construct a uniform, Haskell-based CST presentation. Alternatively, Template Haskell-based metaprogramming could be used to generate a parser. A parser combinator library would be used underneath.*
- *The actual quasi-quoter is essentially boilerplate code, as demonstrated for EL. One may define it as a generic abstraction that is refined into a language-specific quasi-quote by simple parameter passing for the language name and the grammar.*

Test your development by implementing simple metaprograms for different languages.

Summary and Outline

We have catalogued the different representations and mappings that arise in dealing with concrete syntax in language-based software components. We have described practical techniques for scanning, parsing, abstraction, and formatting. The list of techniques included parser generation, parser combinators, pretty printing combinators, and template processing. We have also described the quasi-quotation technique for integrating the concrete syntax of an object language into the metalanguage. The topic of parsing, in particular, could only be covered in a selective manner. There exist many different parsing approaches and technologies and they all come with their specific peculiarities. The topic of formatting was also not covered completely. In particular, lexical formatting [44, 2] was not exercised. The topic of concrete

object syntax was also covered in just one specific manner. We have not covered several aspects of concrete syntax implementation, for example, grammar-based testing [34, 35, 17] and grammar transformation, for example, in the context of language evolution [33, 12, 56, 36].

We will now turn away from syntax and discuss semantics and types. For what it matters, we are going to use abstract object syntax again in most of what follows.

References

1. Aho, A., Monica S., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison Wesley (2006). 2nd edition
2. Bagge, A.H., Hasu, T.: A pretty good formatting pipeline. In: Proc. SLE, *LNCS*, vol. 8225, pp. 177–196. Springer (2013)
3. Basten, H.J.S., Klint, P., Vinju, J.J.: Ambiguity detection: Scaling to scannerless. In: Proc. SLE 2011, *LNCS*, vol. 6940, pp. 303–323. Springer (2012)
4. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing (2013)
5. van den Brand, M., Visser, E.: Generation of formatters for context-free languages. ACM Trans. Softw. Eng. Methodol. **5**(1), 1–41 (1996)
6. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. In: Proc. GPCE, pp. 3–12. ACM (2007)
7. Bravenboer, M., Dolstra, E., Visser, E.: Preventing injection attacks with syntax embeddings. Sci. Comput. Program. **75**(7), 473–495 (2010)
8. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1-2), 52–70 (2008)
9. Bravenboer, M., Visser, E.: Concrete syntax for objects: Domain-specific language embedding and assimilation without restrictions. In: Proc. OOPSLA, pp. 365–383. ACM (2004)
10. Cordy, J.R.: The TXL source transformation language. Sci. Comput. Program. **61**(3), 190–210 (2006)
11. Cordy, J.R.: Excerpts from the TXL cookbook. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 27–91. Springer (2011)
12. Dean, T.R., Cordy, J.R., Malton, A.J., Schneider, K.A.: Grammar programming in TXL. In: SCAM, pp. 93–104. IEEE (2002)
13. Economopoulos, G., Klint, P., Vinju, J.J.: Faster scannerless GLR parsing. In: Proc. CC, *LNCS*, vol. 5501, pp. 126–141. Springer (2009)
14. Erdweg, S.: Extensible languages for flexible and principled domain abstraction. Ph.D. thesis, Philipps-Universität Marburg (2013)
15. Erdweg, S., Rendel, T., Kästner, C., Ostermann, K.: SugarJ: Library-based syntactic language extensibility. In: Proc. OOPSLA, pp. 391–406. ACM (2011)
16. Erdweg, S., Rieger, F., Rendel, T., Ostermann, K.: Layout-sensitive language extensibility with SugarHaskell. In: Proc. Haskell, pp. 149–160. ACM (2012)
17. Fischer, B., Lämmel, R., Zaytsev, V.: Comparison of context-free grammars based on parsing generated test data. In: Proc. SLE 2011, *LNCS*, vol. 6940, pp. 324–343. Springer (2012)
18. Frost, R.A., Hafiz, R.: A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. SIGPLAN Not. **41**(5), 46–54 (2006)
19. Frost, R.A., Hafiz, R., Callaghan, P.: Parser combinators for ambiguous left-recursive grammars. In: Proc. PADL, *LNCS*, vol. 4902, pp. 167–181. Springer (2008)
20. Grune, D., Jacobs, C.: Parsing Techniques: A Practical Guide. Monographs in Computer Science. Springer (2007). 2nd edition
21. Heering, J., Hendriks, P.R.H., Klint, P., Rekers, J.: The Syntax Definition Formalism SDF. reference manual. SIGPLAN Not. **24**(11), 43–75 (1989)

22. Herrera, A.S., Willink, E.D., Paige, R.F.: An OCL-based bridge from concrete to abstract syntax. In: Proc. International Workshop on OCL and Textual Modeling, *CEUR Workshop Proceedings*, vol. 1512, pp. 19–34. CEUR-WS.org (2015)
23. Hughes, J.: The design of a pretty-printing library. In: Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text, *LNCS*, vol. 925, pp. 53–96. Springer (1995)
24. Izmaylova, A., Afroozeh, A., van der Storm, T.: Practical, general parser combinators. In: Proc. PEPM, pp. 1–12. ACM (2016)
25. Johnson, S.C.: YACC—Yet Another Compiler Compiler. Computer Science Technical Report 32, AT&T Bell Laboratories (1975)
26. Jonnalagedda, M., Coppey, T., Stucki, S., Rompf, T., Odersky, M.: Staged parser combinators for efficient data processing. In: Proc. OOPSLA, pp. 637–653. ACM (2014)
27. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proc. GPCE, pp. 249–254. ACM (2006)
28. Kadhim, B.M., Waite, W.M.: Maptool – supporting modular syntax development. In: Proc. CC, *LNCS*, vol. 1060, pp. 268–280. Springer (1996)
29. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Trans. Softw. Eng. Methodol. **14**(3), 331–380 (2005)
30. Klint, P., van der Storm, T., Vinju, J.J.: RASCAL: A domain specific language for source code analysis and manipulation. In: Proc. SCAM, pp. 168–177. IEEE (2009)
31. Klint, P., van der Storm, T., Vinju, J.J.: EASY meta-programming with Rascal. In: GTTSE 2009, Revised Papers, *LNCS*, vol. 6491, pp. 222–289. Springer (2011)
32. Knuth, D.E.: Semantics of context-free languages. Mathematical Systems Theory **2**(2), 127–145 (1968)
33. Lämmel, R.: Grammar adaptation. In: Proc. FM, *LNCS*, vol. 2021, pp. 550–570. Springer (2001)
34. Lämmel, R.: Grammar testing. In: Proc. FASE, *LNCS*, vol. 2029, pp. 201–216. Springer (2001)
35. Lämmel, R., Schulte, W.: Controllable combinatorial coverage in grammar-based testing. In: Proc. TestCom, *LNCS*, vol. 3964, pp. 19–38. Springer (2006)
36. Lämmel, R., Zaytsev, V.: Recovering grammar relationships for the Java language specification. Softw. Qual. J. **19**(2), 333–378 (2011)
37. Leijen, D.: Parsec, a fast combinator parser. Tech. Rep. 35, Department of Computer Science, University of Utrecht (RUU) (2001)
38. Lohmann, W., Riedewald, G., Stoy, M.: Semantics-preserving migration of semantic rules during left recursion removal in attribute grammars. ENTCS **110**, 133–148 (2004)
39. Mainland, G.: Why it's nice to be quoted: quasiquoting for Haskell. In: Proc. Haskell, pp. 73–82. ACM (2007)
40. Maluszynski, J.: Attribute grammars and logic programs: A comparison of concepts. In: Proc. SAGA, *LNCS*, vol. 545, pp. 330–357. Springer (1991)
41. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (2008)
42. Moore, R.C.: Removing left recursion from context-free grammars. In: Proc. ANLP, pp. 249–255 (2000)
43. Nederhof, M.: A new top-down parsing algorithm for left-recursive DCGs. In: Proc. PLILP, *LNCS*, vol. 714, pp. 108–122. Springer (1993)
44. Oppen, D.C.: Prettyprinting. ACM Trans. Program. Lang. Syst. **2**(4), 465–483 (1980)
45. Parr, T.: A functional language for generating structured text (2006). Draft. http://www.stringtemplate.org/articles.html
46. Parr, T.: The Definitive ANTLR 4 Reference. Pragmatic Bookshelf (2013). 2nd edition
47. Parr, T., Fisher, K.: LL(*): The foundation of the ANTLR parser generator. In: Proc. PLDI, pp. 425–436. ACM (2011)
48. Quesada, L., Berzal, F., Talavera, J.C.C.: A domain-specific language for abstract syntax model to concrete syntax model mappings. In: Proc. MODELSWARD, pp. 158–165. SciTePress (2014)

49. Rendel, T., Ostermann, K.: Invertible syntax descriptions: Unifying parsing and pretty printing. In: Proc. Haskell, pp. 1–12. ACM (2010)
50. Rose, L.M., Matragkas, N.D., Kolovos, D.S., Paige, R.F.: A feature model for model-to-text transformation languages. In: Proc. MiSE, pp. 57–63. IEEE (2012)
51. Sheard, T., Peyton Jones, S.L.: Template meta-programming for Haskell. SIGPLAN Not. **37**(12), 60–75 (2002)
52. Swierstra, S.D.: Combinator parsing: A short tutorial. In: International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 – March 1, 2008, Revised Tutorial Lectures, *LNCS*, vol. 5520, pp. 252–300. Springer (2009)
53. Tratt, L.: Domain specific language implementation via compile-time meta-programming. ACM Trans. Program. Lang. Syst. **30**(6) (2008)
54. van den Brand, M., Sellink, M.P.A., Verhoef, C.: Generation of components for software renovation factories from context-free grammars. Sci. Comput. Program. **36**(2-3), 209–266 (2000)
55. van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P.A., Scheerder, J., Vinju, J.J., Visser, E., Visser, J.: The ASF+SDF Meta-Environment: A component-based language development environment. ENTCS **44**(2), 3–8 (2001)
56. Vermolen, S., Visser, E.: Heterogeneous coupled evolution of software languages. In: Proc. MoDELS, *LNCS*, vol. 5301, pp. 630–644. Springer (2008)
57. Visser, E.: Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam (1997)
58. Visser, E.: Meta-programming with concrete object syntax. In: Proc. GPCE, *LNCS*, vol. 2487, pp. 299–315. Springer (2002)
59. Wachsmuth, G.: A formal way from text to code templates. In: Proc. FASE, *LNCS*, vol. 5503, pp. 109–123. Springer (2009)
60. Wadler, P.: The essence of functional programming. In: Proc. POPL, pp. 1–14. ACM (1992)
61. Wielemaker, J., Hendricks, M.: Why it's nice to be quoted: Quasiquoting for Prolog. CoRR **abs/1308.3941** (2013)
62. Zaytsev, V., Bagge, A.H.: Parsing in a broad sense. In: Proc. MODELS, *LNCS*, vol. 8767, pp. 50–67. Springer (2014)