Parsing — a primer

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
http://www.softlang.org/
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 Figure 16 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 were already discussed, to some extent, previously, but we summarize them here for clarity:

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

- **Token streams**: Parsing may involve an extra phase, scanning, for processing text at the level of lexical syntax for units such as whitespace, comments, identifiers, and literals. The resulting units are referred to as tokens (or, in fact, 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 (the 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 the lexeme.

- **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 rep-

- **AST**: Abstract syntax trees are another common representation of language elements. They capture the syntactic structure of the source code but are often simpler than CSTs.

- **ASG**: Abstract syntax graphs provide a visual representation of the abstract syntax, often used in parsing tools to visualize the structure of the input.

- **Model-to-text**: This step involves converting the abstract syntax into a human-readable or machine-executable format.

- **Text-to-model**: This is the reverse process of converting a human-readable or machine-executable format into the abstract syntax.

- **Formatting**: Formatting is the process of rendering the abstract syntax into a human-readable format, often with consideration for presentation and style.

Fig. 16
### Concrete syntax of binary numbers

| [number] number : bits rest ; // A binary number |
| [single] bits : bit ; // A single bit |
| [many] bits : bit bits ; // More than one bit |
| [zero] bit : '0' ; // The zero bit |
| [one] bit : '1' ; // The non–zero bit |
| [integer] rest : ; // An integer number |
| [rational] rest : '.' bits ; // A rational number |
Derivation of ‘10’ from grammar

- number
- bits rest
- bits
- bit bits
- ‘1’ bits
- ‘1’ bit
- ‘1’ ‘0’

Apply rule [number]
Apply rule [integer] to rest
Apply rule [many] to bits
Apply rule [zero] to bit
Apply rule [single] to bits
Apply rule [zero] to bit

[number] number : bits rest ; // A binary number
[single] bits : bit ; // A single bit
[many] bits : bit bits ; // More than one bit
[zero] bit : '0' ; // The zero bit
[one] bit : '1' ; // The non-zero bit
[integer] rest : ; // An integer number
[rational] rest : '.' bits ; // A rational number
Parse tree for ‘10’

[number] number : bits rest;

[many] bits : bit bits;

[one] bit : '1';

[single] bits : bit;

[zero] bit : '0';

1 0

[number] number : bits rest; // A binary number
[single] bits : bit; // A single bit
[many] bits : bit bits; // More than one bit
[zero] bit : '0'; // The zero bit
[one] bit : '1'; // The non-zero bit
[integer] rest : ; // An integer number
[rational] rest : '.' bits; // A rational number
BNF

[number] number : bits rest ; // A binary number
[single] bits : bit ; // A single bit
[many] bits : bit bits ; // More than one bit
[zero] bit : '0' ; // The zero bit
[one] bit : '1' ; // The non-zero bit
[integer] rest : ; // An integer number
[rational] rest : '.'bits ; // A rational number

EBNF

[number] number : { bit }+ { '.'{ bit }+ }? ;
[zero] bit : '0' ;
[one] bit : '1' ;
Concrete syntax of expressions

[unary] expr : uop subexpr ;
[binary] expr : '(' bop ')' subexpr subexpr ;
[subexpr] expr : subexpr ;
[apply] expr : name { subexpr }+ ;
[intconst] subexpr : integer ;
[brackets] subexpr : '(' expr ')' ;
[if] subexpr : 'if' expr 'then' expr 'else' expr ;
[arg] subexpr : name ;
Concrete syntax of statements

[skip] stmt : ';';
[assign] stmt : name '==' expr ';';
[block] stmt : '{' { stmt }* '}'
[if] stmt : 'if' '(' expr ')' stmt { 'else' stmt }? ;
[while] stmt : 'while' '(' expr ')' stmt ;
Fundamental definitions (Chapter 6)

**Definition 6.1** (Context-free grammar (CFG)). A CFG $G$ is a quadruple $\langle N, T, P, s \rangle$ where $N$ is a finite set of nonterminals, $T$ is a finite set of terminals, with $N \cap T = \emptyset$, $P$ is a finite set of rules (or productions) as a subset of $N \times (N \cup T)^*$, and $s \in N$ is referred to as the startsymbol.

**Definition 6.2** (Context-free derivation). Given a CFG $G = \langle N, T, P, s \rangle$ and a sequence $p n q$ with $n \in N$, $p, q \in (N \cup T)^*$, the sequence $p r q$ with $r \in (N \cup T)^*$ is called a derivation, as denoted by $p n q \Rightarrow p r q$, if there is a production $\langle n, r \rangle \in P$.

**Definition 6.3** (Language generated by a CFG). Given a CFG $G = \langle N, T, P, s \rangle$, the language $L(G)$ generated by $G$ is defined as the set of all the terminal sequences that are derivable from $s$. That is:

$$L(G) = \{ w \in T^* \mid s \Rightarrow^+ w \}$$
Fundamental definitions (Chapter 6)

**Definition 6.5** (Acceptor). Given a CFG $G = \langle N, T, P, s \rangle$, an acceptor for $G$ is a computable predicate $a_G$ on $T^*$ such that for all $w \in T^*$, $a_G(w)$ holds iff $s \Rightarrow^+ w$. ■

**Definition 6.6** (Concrete syntax tree (CST)). Given a CFG $G = \langle N, T, P, s \rangle$ and a string $w \in T^*$, a CST for $w$ according to $G$ is a tree as follows:

- Nodes hold a rule or a terminal as info.
- The root holds a rule with $s$ on the left-hand side as info.
- If a node holds a terminal as info, then it is a leaf.
- If a node holds rule $n \rightarrow v_1 \cdots v_m$ with $n \in N$, $v_1, \ldots, v_m \in N \cup T$ as info, then the node has $m$ branches with subtrees $t_i$ for $i = 1, \ldots, m$ as follows:
  - If $v_i$ is a terminal, then $t_i$ is a leaf with terminal $v_i$ as info.
  - If $v_i$ is a nonterminal, then $t_i$ is tree with a rule as info such that $v_i$ is the left-hand side of the rule.
- The concatenated terminals at the leaf nodes equal $w$.

**Definition 6.7** (Parser). Given a CFG $G = \langle N, T, P, s \rangle$, a parser for $G$ is a partial function $p_G$ from $T^*$ to CSTs such that for all $w \in L(G)$, $p_G(w)$ returns a CST of $w$ and for all $w \notin L(G)$, $p_G(w)$ is not defined. ■
Definition 6.8 (Ambiguous grammar). A CFG \( G = (N, T, P, s) \) is called ambiguous, if there exists a terminal string \( w \in T^* \) with multiple CSTs.
Top-down acceptance — algorithm

**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:**
Steps:

1. Initialize $z$ with $s$ (i.e., the start symbol) as top of 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.
Top-down acceptance — example

<table>
<thead>
<tr>
<th>Step</th>
<th>Remaining input</th>
<th>Stack (TOS left)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>‘1’, ‘0’</td>
<td>number</td>
<td>Expand rule [number]</td>
</tr>
<tr>
<td>2</td>
<td>‘1’, ‘0’</td>
<td>bits rest</td>
<td>Expand rule [many]</td>
</tr>
<tr>
<td>3</td>
<td>‘1’, ‘0’</td>
<td>bit bits rest</td>
<td>Expand rule [one]</td>
</tr>
<tr>
<td>4</td>
<td>‘1’, ‘0’</td>
<td>‘1’ bits rest</td>
<td>Consume terminal ‘1’</td>
</tr>
<tr>
<td>5</td>
<td>‘0’</td>
<td>bits rest</td>
<td>Expand rule [single]</td>
</tr>
<tr>
<td>6</td>
<td>‘0’</td>
<td>bit rest</td>
<td>Expand rule [zero]</td>
</tr>
<tr>
<td>7</td>
<td>‘0’</td>
<td>‘0’ rest</td>
<td>Consume terminal ‘0’</td>
</tr>
<tr>
<td>8</td>
<td>–</td>
<td>rest</td>
<td>Expand rule [integer]</td>
</tr>
<tr>
<td>9</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Top-down acceptance — Implementation

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

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"])
]

type Grammar = [Rule]
type Rule = (Label, Nonterminal, [GSymbol])
data GSymbol = T Terminal | N Nonterminal
type Label = String
type Terminal = Char
type Nonterminal = String
Top-down acceptance — Implementation

accept :: [Rule] → String → Bool
accept g = steps g [N s]
    where
        -- Retrieve start symbol
        ((_, s, _) : _) = g

steps :: [Rule] → [GSymbol] → String → Bool
-- Acceptance succeeds (empty stack, all input consumed)
steps _ [] [] = True
-- Consume terminal at top of stack from input
steps g (T t:z) (t':i) | t==t' = steps g z i
-- Expand a nonterminal; try different alternatives
steps g (N n:z) i = or (map (λ rhs → steps g (rhs++z) i) rhss)
    where
        rhss = [ rhs | (_, n', rhs) ← g, n == n' ]
-- Otherwise parsing fails
steps _ _ _ = False
Top-down acceptance — Issues

- Non-determinism
- Backtracking
- Look ahead
- Left recursion

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.

[add] expr : expr ' + ' expr ;
[const] expr : integer ;
7.2 Parsing

7.2.1.2 Bottom-up acceptance

In bottom-up acceptance (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 by a nonterminal while 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 $\epsilon$ as 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 $z$.
     d. Return $false$, if no action is feasible.
5. Go to 3.

It is insightful to notice how top-down and bottom-up acceptance are opposites 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.

Illustration 7.7 (Bottom-up acceptance)
Steps:

1. **Initialize** \(z\) with as 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.
## Bottom-up acceptance — example

<table>
<thead>
<tr>
<th>Step</th>
<th>Remaining input</th>
<th>Stack (TOS right)</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>‘1’, ‘0’</td>
<td>–</td>
<td>Shift terminal ‘1’</td>
</tr>
<tr>
<td>2</td>
<td>‘0’</td>
<td>‘1’</td>
<td>Reduce rule [one]</td>
</tr>
<tr>
<td>3</td>
<td>‘0’</td>
<td>bit</td>
<td>Reduce rule [single]</td>
</tr>
<tr>
<td>4</td>
<td>‘0’</td>
<td>bits</td>
<td>Shift terminal ‘0’</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>bits ‘0’</td>
<td>Reduce rule [one]</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>bits bit</td>
<td>Reduce rule [many]</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>bits</td>
<td>Reduce rule [integer]</td>
</tr>
<tr>
<td>8</td>
<td>–</td>
<td>bits rest</td>
<td>Reduce rule [number]</td>
</tr>
<tr>
<td>9</td>
<td>–</td>
<td>number</td>
<td>–</td>
</tr>
</tbody>
</table>
Bottom-up acceptance — Implementation

accept :: [Rule] → String → Bool
accept g = steps g [] —— Begin with empty stack

steps :: [Rule] → [GSymbol] → String → Bool
—— Acceptance succeeds (start symbol on stack, all input consumed)
steps g [N s] [] | s == s’ = True
  where
    —— Retrieve start symbol
    ((_, s’, _):_) = g
    —— Shift or reduce
steps g z i = shift || reduce
  where
    —— Shift terminal from input to stack
    shift = not (null i) && steps g (T (head i) : z) (tail i)
    —— Reduce prefix on stack to nonterminal
    reduce = not (null zs) && or (map (λ z → steps g z i) zs)
      where
        —— Retrieve relevant reductions
        zs = [ N n : drop l z
          | (_, n, rhs) ← g,
            let l = length rhs,
            take l z == reverse rhs ]
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.

- **Non-determinism**

- **Epsilon rules**

See the Software Languages Book or texts on parsing.
Top-down parsing — implementation

type Info = Either Char Rule

Parse trees

type CST = Tree Info

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

Parse trees

tree :: [Rule] → GSymbol → String → Maybe (String, CST)
We use Haskell's library type `Tree` for node-labeled rose tree, 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 7.11 (Implementation of top-down parsing)

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

```
tree :: [Rule] → GSymbol → String → Maybe (String, CST)
tree _ (T t) i = do
  guard ([t] == take 1 i)
  return (drop 1 i, Node (Left t) [])
```

```
tree g (N n) i = foldr mplus mzero (map rule g)
  where
    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)
```

```
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 the function-application stack of Haskell. This happens to imply that we limit ourselves to local backtracking. Thus, the parser is less complete than the acceptor implemented earlier (Section 7.2.1.1).
The Metametalevel

grammar : {rule}* ;
rule : '[' label ']' nonterminal ':' gsymbols ';' ;
gsymbols : {gsymbol}* ;
[t] gsymbol : terminal ;
[n] gsymbol : nonterminal ;
label : name ;
terminal : qstring ;
nonterminal : name ;
Online resources

YAS (Yet Another SLR (Software Language Repository))

http://www.softlang.org/yas

YAS’ GitHub repository contains all code.

The Software Languages Book

http://www.softlang.org/book