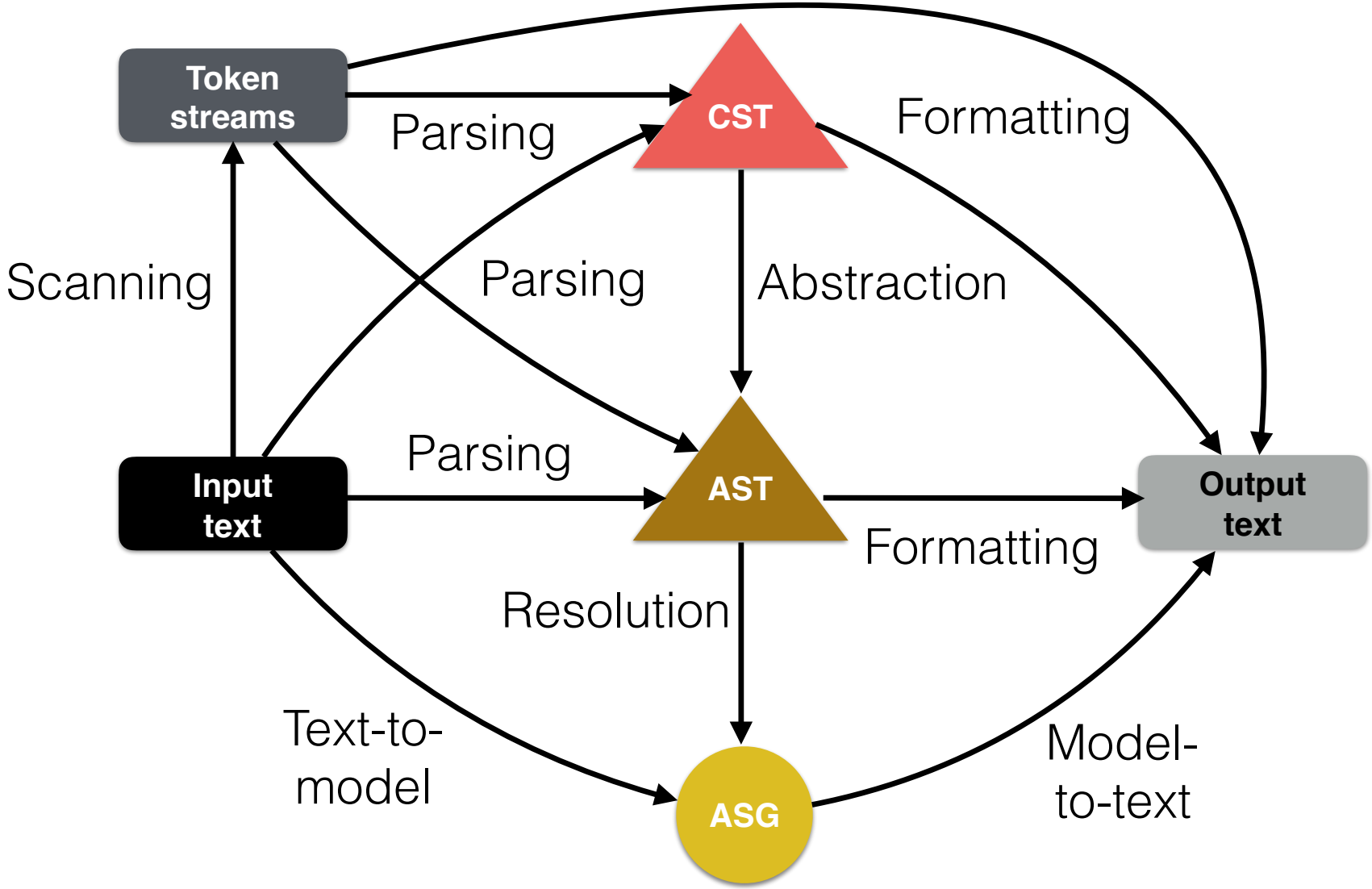
Parsing — a primer

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.



Concrete syntax of binary numbers

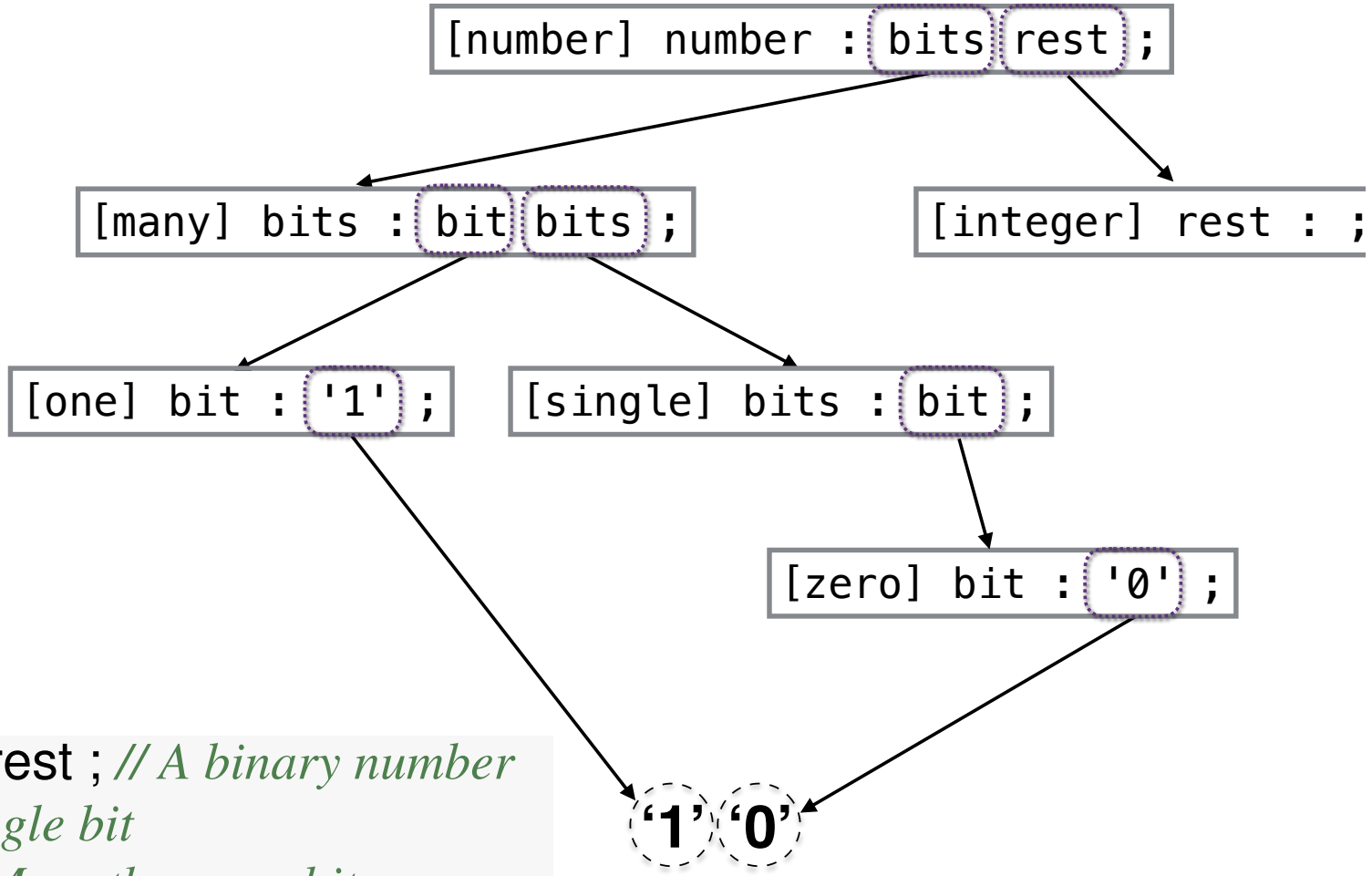
```
[number] number : bits rest ; // A binary number
[single] bits : bit ; // A single bit
[many] bits : bit bits ; // More than one bit
[zero] bit : '0' ; // The zero bit
[one] bit : '1' ; // The non-zero bit
[integer] rest : ; // An integer number
[rational] rest : '.' bits ; // A rational number
```

Derivation of ‘10’ from grammar

• *number*	*Apply rule [number]*
• *bits rest*	*Apply rule [integer] to rest*
• *bits*	*Apply rule [many] to bits*
• *bit bits*	*Apply rule [zero] to bit*
• *‘1’ bits*	*Apply rule [single] to bits*
• *‘1’ bit*	*Apply rule [zero] to bit*
• *‘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
```

Parse tree for ‘10’




```
[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 $n \to v_1 \cdots v_m$ 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.* ■

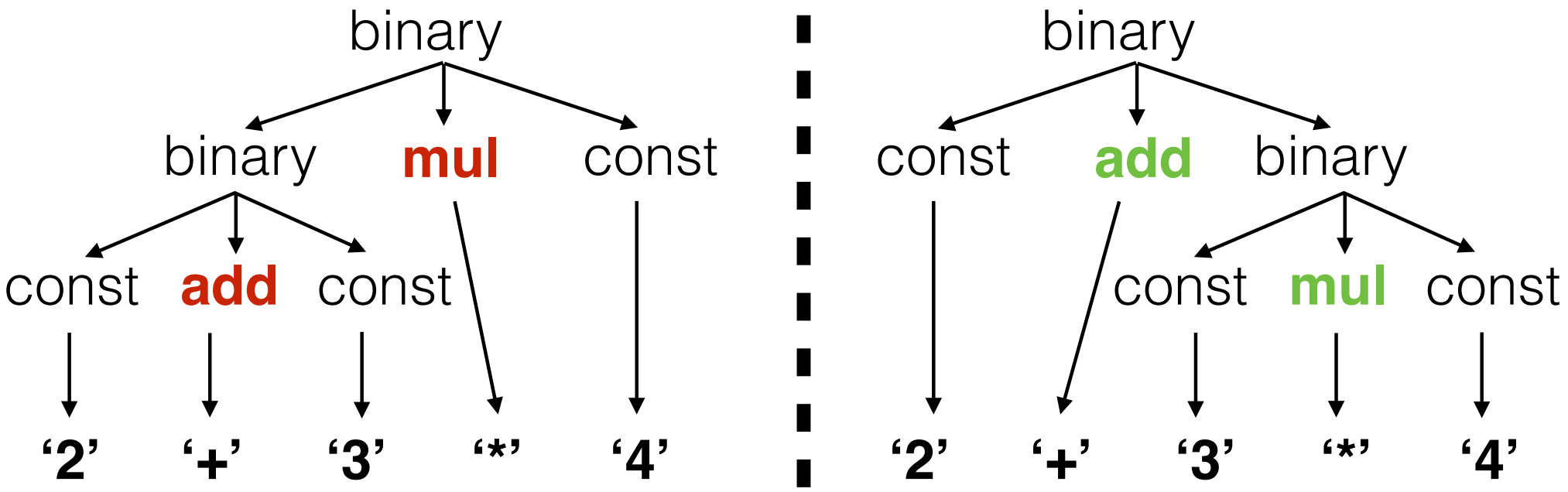
Fundamental definitions (Chapter 6)

Definition 6.8 (Ambiguous grammar). *A CFG $G = \langle N, T, P, s \rangle$ is called ambigious, if there exists a terminal string $w \in T^*$ with multiple CSTs.* ■

Example

```
[binary] expr : expr bop expr ;
[const] expr : integer ;
[add] bop : '+' ;
[mul] bop : '*' ;
```

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

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	*–*	*–*	*–*

Top-down acceptance — Implementation

```
► accept bnlGrammar "101.01"
True
► accept bnlGrammar "x"
False
```

Acceptor application

```
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"])
 ]
```

Grammar representation

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

Grammar representation types

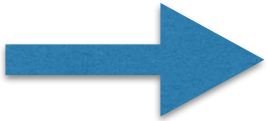
Top-down acceptance — Implementation

```
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 ;
```

Bottom-up 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 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

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*	*–*

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 ]
```

Bottom-up acceptance — Issues

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**

```
[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
```

See the Software Languages Book or texts on parsing.

Top-down parsing — implementation

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

Parse trees

```
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)
```

```
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)
```

The Metametalevel

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

EBNF grammar
of BNF grammars
(useful for parsing grammars)

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