Proposal for a Formalism for Sublanguage Specification

Stuart M. Shieber

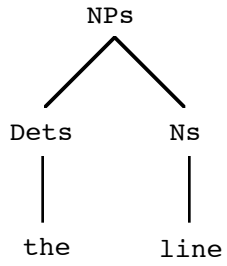
January 27, 1996

This document describes a formalism for specifying command and control sublanguages and their semantics (in terms of generated code) for voice-enabling applications. The formalism allows definition of two modules:

Front end A context-free grammar, to be used for converting a sequence of spoken words into a tree.

Back end A cascaded tree-rewriting system, to be used for converting the tree into generated code for the application.

The trees that are processed by these modules are ordered trees with labeled nodes. As an example, the following tree has a root node labeled with the nonterminal `NPs`.



We will notate trees such as this with the following notation:

```
NPs(Dets(the), Ns(line))
```

Lexical Issues and Meta-Notation

Below, we use BNF notation to define the syntax of the formalism, and we describe the semantics of the formalism informally.

The lowest level nonterminals in the BNF metagrammar are

‹*variable*› Any sequence of alphabetic and numeric characters and underscore beginning with a capitalized alphabetic character or underscore. Examples:

```
NPs
Dets
OrdinalNum
Ordinal_num
_number
```

‹*string*› Any sequence of alphabetic and numeric characters and underscore not beginning with a capitalized alphabetic character or underscore, or any sequence of characters surrounded by double quotes, e.g.,

```
value
"a long string"
42
```

Tokens comprise variables and symbols (as above) and the following special tokens:

```
,  ...  |  (  )  {  }  {}  -->  ==>
```

Ignored whitespace includes the normal whitespace characters and comments. Comments may be introduced by "//" and continue to the end of the line, or may be surrounded by "`/* ... */`".

In the BNF metanotation, all terminals are given in quotes and vertical bar specifies alternatives. An ellipsis following two constituents specifies zero or more occurrences of the first separated by occurrences of the second. For instance, the BNF rule:

‹*list*› ::= ‹*string*› "`,`" ...

would specify a comma-separated list of strings, e.g., the empty list or the lists

```
foo, 37, "Third element."
5
```

and so forth.

The Front End Module: Context-Free Grammars

A front end is defined by a context-free grammar.

Terminals. Terminals in the grammar are notated as strings. (Recall that symbols may optionally be quoted.)

‹*terminal*› ::= ‹*string*›

Nonterminals. Nonterminals are notated as variables.

‹*nonterminal*› ::= ‹*variable*›

Grammars. A grammar is given as a series of grammar rules.

‹*grammar*› ::= ‹*rule*› "" ...

Start nonterminal. The nonterminal occurring as the left-hand side of the first grammar rule is the start nonterminal of the grammar.

Grammar rules. Grammar rules are specified with the following notation:

‹*decl*› ::= ‹*nonterminal*› "`-->`" ‹*rhs*›

Right-hand sides. The right-hand sides of rules allow for sequences of terminals and nonterminals, along with optional elements and alternatives.

```
‹rhs› ::= ‹rhs-alt›
        | ‹rhs-alt› "|" ‹rhs›                  // alternatives
‹rhs-alt› ::= ‹rhs-element›
        | ‹rhs-element› ‹rhs-alt›              // sequences
‹rhs-element› ::= ‹nonterminal›                // nonterminals
        | ‹terminal›                           // terminals
        | "{}"                                 // empty string
        | "{" ‹rhs› "}"                        // optional element
        | "{" ‹rhs› ==> ‹tree-spec› "}"        // optional element with default value
        | "(" ‹rhs› ")"                        // grouping

‹tree-spec› ::= ‹terminal›
        | ‹nonterminal› "(" ‹tree-spec› "," ... ")"
```

Optional elements can be given a default value, specified as an explicit tree structure.

Specifying Trees

A front-end context-free grammar is used to generate trees to send to the back end rewriting module. Therefore, we must define not only how a grammar specifies a set of strings (actually token sequences), but also how it specifies a set of trees. To do so, we specify how terminals and nonterminals specify trees, and how right-hand-sides specify tree sequences.

Terminals. A terminal "`term`" specifies a token sequence "`term`" and a tree of a single terminal node labeled "`term`". The special symbol "`{}`" specifies the empty token sequence and a tree of a single terminal node labeled "`{}`".

Given right-hand sides *x* and *y*, token and tree sequences specified by right-hand sides are as follows:

Grouping. The right-hand side `(`*x*`)` specifies the same token and tree sequences as *x* itself.

Sequencing. Suppose that two right-hand sides *x* and *y* specify token (resp. tree) sequences *x′* and *y′* respectively. The right-hand side *x y* specifies the token (resp. tree) sequence that is the concatenation of *x′* and *y′*.

Optionality. The right-hand side `{`*x*`}` specifies the same token and tree sequence as `(`*x* `|` `{}``)`. The right-hand side `{`*x* `==>` *t*`}` specifies the same token sequence as `(`*x* `|` `{}``)` but the same tree sequence as `(`*x* `|` *t*`)`.

Alternation. The right-hand side *x* `|` *y* specifies any token (resp. tree) sequence that is specified by either *x* or *y*.

Nonterminals. Finally, suppose the nonterminal `X` is defined by a rule `X` `-->` *y*, and *y* specifies a token sequence *y′* and a tree sequence *y″*. Then, the nonterminal `X` specifies the token sequence *y′* and the tree whose root is labeled `X` and whose children are the trees in the tree sequence *y′*.

For example, given the following grammar:

```
NP --> {Dets ==> Dets(a)} Ns | Detp Np | it
Dets --> the | a
Detp --> these
Noms --> line
Nomp --> lines
```

the nonterminal `NP` — the start nonterminal of the grammar — generates the following token sequences and trees:

```
it               NP(it)
the line         NP(Dets(the), Ns(line))
a line           NP(Dets(a), Ns(line))
line             NP(Dets(a), Ns(line))
these lines      NP(Detp(these), Np(lines))
```

The Full Front End Metagrammar

The full metagrammar for front-end grammars is as follows:

‹*grammar*› ::= ‹*rule*› "" ...

‹*rule*› ::= ‹*nonterminal*› "`-->`" ‹*rhs*›

‹*rhs*› ::= ‹*rhs-alt*›
| ‹*rhs-alt*› "|" ‹*rhs*› // alternatives
‹*rhs-alt*› ::= ‹*rhs-element*›
| ‹*rhs-element*› ‹*rhs-alt*› // sequences
‹*rhs-element*› ::= ‹*nonterminal*› // nonterminals
| ‹*terminal*› // terminals
| "`{}`" // empty string

| "{" ‹*rhs*› "}" // optional element
| "{" ‹*rhs*› ==> ‹*tree-spec*› "}" // optional element with default value
| "(" ‹*rhs*› ")" // grouping

‹*tree-spec*› ::= ‹*terminal*›
| ‹*nonterminal*› "(" ‹*tree-spec*› "," ... ")"

‹*nonterminal*› ::= ‹*variable*›
‹*terminal*› ::= ‹*string*›

The Back End Module: Tree-Rewriting Systems

The back end is specified as a cascade of tree-rewriting systems. Each such system is defined by a set of tree-rewriting rules composed of a pattern and a result. The rules are applied according to a deterministic postorder traversal method: To rewrite a tree, the first rule whose pattern matches the tree is used. The result of rewriting is obtained by interpreting the result in the context of the match. The interpretation in turn may require further rewriting of subparts of the matched tree.

Specifying Patterns

Tree variables. Patterns can be specified with tree variables:

‹*pattern*› ::= ‹*tree-variable*›
‹*tree-variable*› ::= ‹*label*›
| ‹*label*› "_" ‹*index*›
| "_" ‹*index*›
‹*label*› ::= ‹*variable*›
| ‹*string*›
‹*index*› ::= ‹*string*›

Examples:

```
move  NP_subject  _tree
```

Variables with no ‹*label*› (such as "_" and "`_tree`") match any tree. Variables starting with a ‹*label*› match any tree whose root node is labeled with the given symbol. Thus, the variable "_" matches any tree, and the variable "`NP_subject`" matches any tree whose root node is labeled with "`NP`". (The intended use for the indices is to allow specification of and later reference to multiple variables that match trees with the same root label.) Upon matching, the variable becomes bound to the matched tree, and can then be used to refer to that tree in the result portion of a rewrite rule. No tree variable may occur more than once in a given pattern.

Trees. Patterns can match against the children of a node as well, using the following notation, a further generalization of the preceding rule:

‹*pattern*› ::= ‹*tree-variable*› "`(`" ‹*pattern*› "`,`" ... "`)`"

Such a pattern matches the tree against the tree variable as before, and matches the children of the root node of the tree against the ‹*pattern*›s in order.

Examples:

```
NP (Det, N)
```

This pattern matches a tree with root labeled `NP` and two children labeled `Det` and `N` respectively.

```
Command(move, down, Number, Units)
```

Tree sequences. To allow for matching against an indeterminate number of children, a variable matching a sequence of children is allowed at the end of the list of patterns.

‹*pattern*› ::= ‹*tree-variable*› "`(`" ‹*pattern*› "`,`" ... "`,`" ‹*seq-variable*› "`)`"
‹*seq-variable*› ::= "`...`"
| "`...`" ‹*index*›

Example:

```
NP (_, ...)
```

This pattern matches a node with root labeled NP and more than one child. The sequence of children comprising all but the first child is bound to the variable "`...`".

Specifying Results

Given a binding of trees to tree variables and tree sequences to sequence variables (as generated by a pattern match), the following notations allow specifying the tree that results from the rewrite. In the examples below, we assume that the following bindings are in force:

```
NP_subj       NP(Det(the), N(lines))
Det           Det(the)
N             N(lines)
```

Tree references. Results can be specified with references to tree variables that were bound by the corresponding pattern part of the rewrite rule:

‹*result*› ::= ‹*tree-variable*›

A tree-variable reference specifies the output of recursively rewriting whatever tree is bound to the referenced variable unless the variable matched the entire tree in the ‹pattern› portion of the rule. In this case, no rewriting is performed. When rewriting is performed, it is done using the rules of the current system of rules (the current pass) only.

Trees. Results can be specified with a tree construction similar to that for patterns:

‹*result*› ::= ‹*terminal*›
| ‹*nonterminal*› "`(`" ‹*result*› "`,`" ... "`)`"
| ‹*nonterminal*› "`(`" ‹*result*› "`,`" ... "`,`" ‹*seq-variable*› "`)`"

The tree constructed by the first alternative is the single node labeled with the given ‹*terminal*›. The tree specified by the second and third alternatives is constructed with root labeled by the ‹*nonterminal*› and with children as specified by the parenthesized results. If a sequence variable is given, all trees in the sequence are separately recursively rewritten and added as children of the result tree.

Failure. The special result specification "`FAIL`" specifies that the rewriting fails.

‹*result*› ::= "`FAIL`" ‹*symbol*›

The symbol may be used for error reporting, perhaps by reporting it directly or by using it as an index into a table of more complete information.

String concatenation. Results can be specified as the concatenation of two other result strings using the following notation:

‹*result*› ::= ‹*result*› "`.`" ‹*result*›

The two result trees are computed and the final result is the symbol generated by concatenating the strings labeling the roots of the two result trees. This is intended primarily for generating a code string as the output of the final pass.

Specifying Rewrite Rules

Given the languages for patterns and results, the following notation allows specifying rewrite rules:

‹*rule*› ::= ‹*pattern*› "`==>`" ‹*result*›

For instance, the following are some rewrite rules:

```
NP_subj (Det, N) ==> Det(N)
Det(D) ==> D
N(_Noun) ==> _Noun
```

```
Command(move, down, Number, Units)
             ==> Move(down, Number, Units)
Number( _n ) ==> _n
Units(lines) ==> line
Units( _unit) ==> _unit
```

A rewrite rule in which the ‹*result*› leaves the matched tree unchanged except for recursive rewriting of all the children, as if by the rule

```
_(...) ==> _(...)
```

may be abbreviated by giving its pattern only:

‹*rule*› ::= ‹*pattern*›

For instance, the rule

```
NP(Det, N) ==> NP(Det, N)
```

may be abbreviated

```
NP(Det, N)
```

Such abbreviated rules specify that a certain type of tree is allowed but left unchanged by the pass. For instance, a default rule for applying all other rules at the top level only, leaving the tree unchanged if no other rules apply:

```
_ ==> _
```

can be abbreviated

```
_
```

Similarly, a default rule for applying all other rules recursively, leaving the tree unchanged if no other rules apply can be given as:

```
_(...)
```

Specifying Systems Of Rules

A system of rules (called a "pass") is specified according to the following rules:

‹*pass*› ::= "`Pass`" ‹*label*› ‹*rule*› "" ...

For instance, the following are two systems:

```
Pass "as per Bentz"

Command(move, down, Number, Units)
```

```
      ==> Move(down, Number, Units)
Number( _n ) ==> _n
Units(lines) ==> line
Units( _unit ) ==> _unit
_

Pass "as per Pilato"

NP(Det, N) ==> Det(N)
Det(D) ==> D
N( _noun ) ==> _noun
_(...)
```

Applying a system of rules to a tree works as follows: The textually first rule whose pattern matches the tree is selected. Its result is generated and returned as the rewritten form of the input tree. If no rules match, the system of rules fails, and no rewritten version is generated.

For instance, the first pass above would rewrite the tree

```
Command(move, down, Number(3), Units(lines))
```

to the tree

```
Move(down,3,line)
```

by virtue of the following rule applications: First, the tree is matched against the first rule in the pass, giving the following bindings:

```
Command =     Command(move, down, Number(3), Units(lines))
Number =      Number(3)
Units =       Units(lines)
```

(Although the pattern of the last rule also matches, the textually earlier one is used.)

Then the result is generated, namely

```
Move(down, Number, Units)
```

This specifies a tree with root labeled "`Move`" and three children, one the single node tree "`down`" and the other two being the rewritten versions of the trees "`Number(3)`" and "`Units(lines)`". These are, respectively, "`3`" (as per the second rule) and "`line`" (as per the third rule, which takes precedence over the fourth rule). Note that the fifth rule, which returns the whole tree unchanged, is always applicable, but is used only if no other rule matches. This serves as a kind of default rule.

Similarly, the second pass above rewrites the tree

```
NP(Det(the), N(lines))
```

to the tree

```
the(lines)
```

Specifying Cascades Of Systems

A cascade of systems is specified as a sequence of passes:

‹*cascade*› ::= ‹*pass*› "" ...

A cascade of rule systems operates on a tree by applying the first pass to the tree, the second pass to the output of the first, the third pass to the output of the second, and so forth until the output of the final pass is returned or until some pass fails.

For instance, the entire cascade of two passes given above rewrites

```
Command(move, down, Number(3), Units(lines))
```

to

```
Move(down,3,line)
```

and

```
NP(Det(the), N(lines))
```

to

```
the(lines)
```

since each system has a default rule that passes through anything it does not handle.

The Full Back End Metagrammar

‹*cascade*› ::= ‹*pass*› "" ...

‹*pass*› ::= "`Pass`" ‹*label*› ‹*rule*› "" ...

‹*rule*› ::= ‹*pattern*› "`==>`" ‹*result*›
| ‹*pattern*›

‹*pattern*› ::= ‹*tree-variable*›
| ‹*tree-variable*› "`(`" ‹*pattern*› "`,`" ... "`)`"
| ‹*tree-variable*› "`(`" ‹*pattern*› "`,`" ... "`,`" ‹*seq-variable*› "`)`"

‹*tree-variable*› ::= ‹*label*›
| ‹*label*› "_" ‹*index*›

| "_" ‹*index*›
‹*label*› ::= ‹*variable*›
| ‹*string*›
‹*index*› ::= ‹*string*›

‹*seq-variable*› ::= "`...`"
| "`...`" ‹*index*›

‹*result*› ::= ‹*tree-variable*›
| ‹*terminal*›
| ‹*nonterminal*› "`(`" ‹*result*› "`,`" ... "`)`"
| ‹*nonterminal*› "`(`" ‹*result*› "`,`" ... "`,`" ‹*seq-variable*› "`)`"
| "`FAIL`" ‹*symbol*›
| ‹*result*› "`.`" ‹*result*›

Augmentations To Consider

Adding pass identifiers to rules:

Rather than associating passes to rules by collecting together all of the rules in a given pass, it might be better to give an explicit pass identifier to each rule. This would allow intermixing of the rules so that all of the rules that are involved in processing a single construction through all of the passes could be kept together.

Adding a reporting construct:

To facilitate error reporting and debugging, a construct that reports information to the user might be useful, e.g.,

‹*result*› ::= "`report`" ‹*string*› "`for`" ‹*result*›

which reports the given ‹*string*› and then computes the ‹*result*› as usual. The reporting might provide not only the user-specified string but also the current pass number and binding context.

Adding call-outs:

A similar construct would allow calling out to arbitrary user code.

‹*result*› ::= "`compute`" ‹*pattern*› "`as`" ‹*symbol*› "`in`" ‹*result*›

This would execute some code as specified by the ‹*symbol*› and then pattern match the result against the ‹*pattern*›, adding the match to the current context before generating the ‹*result*›. This obviously subsumes the reporting construct above, but is highly dependent on the details of the language in which the interpreter is implemented.

Forcing further rewriting:

A great increase in expressivity can be achieved by allowing the user to specify further rewriting explicitly, e.g.,

‹*result*› ::= " ‹ " ‹*result*› " › "

This would cause the embedded ‹*result*› to be generated and then the current pass of rules would be again applied to it. This would give a single pass of computation Turing-equivalent power.

A Sample Grammar

```
/*-------------------------------------------------------------------

                    Grammar for Natural Numbers

-------------------------------------------------------------------*/

// Front end grammar

/* Starting nonterminal is NatNum
     NatNum       covers natural numbers 0 through 10^12 - 1
     NatNumX      covers natural numbers 1 through 10^X - 1
     NatDigit     covers natural digits 1 through 9
     NatLeadDig   similarly
     NatTeen      covers 10 through 19
     NatTy        covers the multiples of 10, 20 through 90
*/

NatNum   --> zero | NatNum12
NatNum12 --> NatNum3 | NatNum3 billion NatNum9
NatNum9  --> NatNum3 | NatNum3 million NatNum6
NatNum6  --> NatNum3 | NatNum3 thousand NatNum3
NatNum3  --> NatNum2 | NatLeadingDigit hundred {and} NatNum2
NatNum2  --> NatDigit | NatTeen | NatTy NatDigit
NatDigit --> one | two | three | four | five
           | six | seven | eight | nine
NatLeadDig --> a | NatDigit
NatTeen  --> ten | eleven | twelve | thirteen | fourteen
           | fifteen | sixteen | seventeen | eighteen | nineteen
NatTy    --> twenty | thirty | forty | fifty
           | sixty | seventy | eighty | ninety

//-------------------------------------------------------------------

// Back end rewriting system

//...................................................................
Pass "compute expression"

// Rewrites natural numbers into an arithmetic expression tree that
// computes the corresponding numeric value.

NatNum(NatNum12)  ==>   NatNum12
NatNum(zero)      ==>   0

NatNum12(NatNum9) ==>   NatNum9
NatNum12(NatNum3, billion, NatNum9)
                  ==>   Plus(NatNum9, Times(NatNum3, Exp(10, 9)))

NatNum9(NatNum6)  ==>   NatNum6
NatNum9(NatNum3, million, NatNum6)
                  ==>   Plus(NatNum6, Times(NatNum3, Exp(10, 6)))

NatNum6(NatNum3)  ==>   NatNum3
NatNum6(NatNum3_1, thousand, NatNum3_2)
                  ==>   Plus(NatNum3_2, Times(NatNum3_1, Exp(10, 3)))
```

```
NatNum3(NatNum2)  ==>   NatNum2
NatNum3(NatLeadingDigit, hundred, _, NatNum2)
                  ==>   Plus(NatNum2, Times(NatLeadDig,
                                            Exp(10, 2)))

NatNum2(NatDigit) ==>   NatDigit
NatNum2(NatTy, NatDigit) ==> Plus(NatTy, NatDigit)
NatNum2(NatTeen)  ==>   NatTeen

NatDigit(one)     ==>   1
NatDigit(two)     ==>   2
NatDigit(three)   ==>   3
NatDigit(four)    ==>   4
NatDigit(five)    ==>   5
NatDigit(six)     ==>   6
NatDigit(seven)   ==>   7
NatDigit(eight)   ==>   8
NatDigit(nine)    ==>   9

NatLeadDig(NatDigit) ==> NatDigit
NatLeadDig(a)     ==>   1

NatTeen(ten)      ==>   10
NatTeen(eleven)   ==>   11
NatTeen(twelve)   ==>   12
NatTeen(thirteen) ==>   13
NatTeen(fourteen) ==>   14
NatTeen(fifteen)  ==>   15
NatTeen(sixteen)  ==>   16
NatTeen(seventeen)==>   17
NatTeen(eighteen) ==>   18
NatTeen(nineteen) ==>   19

NatTy(twenty)     ==>   20
NatTy(thirty)     ==>   30
NatTy(forty)      ==>   40
NatTy(fifty)      ==>   50
NatTy(sixty)      ==>   60
NatTy(seventy)    ==>   70
NatTy(eighty)     ==>   80
NatTy(ninety)     ==>   90

//.................................................................
Pass "generate code"

// Converts arithmetic Expression trees into corresponding VB string

Plus(X, Y)        ==>   "(" . X . " + " . Y . ")"
Times(X, Y)       ==>   "(" . X . " * " . Y . ")"
Exp(X, Y)         ==>   "(" . X . " ^ " . Y . ")"
_                 ==>   _
```