Format for describing grammars for predictive parsing

Abstract

This document describes the syntax for two different judges, one focused exclusively on lexical descriptions (first section) and one for descriptions of lexical & syntactic analysis together with construction of abstract syntax trees (remaining sections). The syntax for both judges is a simplified variant of ANTLR3.

1 Lexical descriptions

To describe a lexical class it suffices to define it as a regular language using a notation reminiscent of regular expressions. More precisely, such notation uses the following operators, listed from lowest to highest precedence:

- Binary operator `|` for alternative.
- Concatenation, a binary operation denoted without any symbol.
- Unary postfix operators `*`, `+`, `?` for Kleene's closure (i.e., 0 or more repetitions), positive closure (i.e., 1 or more), and optional (i.e., 0 or 1), respectively. Note that e`+` is equivalent to `(`e e`*)` and that e`?` is equivalent to `(`e `| )`, for any subexpression e.
- Unary prefix operator `~` for complement (with respect to the alphabet, i.e., to the set of all single-character words). This operator can only be applied to languages of single-character words (i.e., to subsets of the alphabet).
- Binary operator `..` for range of alphabet symbols. Both operands must be single-character words.

And a basic expression is any of the following:

- A literal string between quotes; e.g., `'abc'` for the word *abc*. Escape sequences denoting special symbols are introduced with a backslash and can be any of the following: `'\n'` for new line, `'\t'` for tabulator, `'\\'` for backslash, `'\''` for single quote, `'\"'` for double quote. Note that the latter two escapes sequences are always avoidable by choosing the appropriate outer quotes, i.e., by writing `"'"` or `'"'` for a single or double quote, respectively.
- Nothing, denoting the empty word.

According to the precedence, the implicit parenthesization of an expression like `~'0'..'9'*` is `(~('0'..'9'))*` and, thus, has to be interpreted as denoting the words of any length (due to `*`) that do not contain (due to `~`) any digit (due to the range `'0'..'9'`).

2 Lexical and syntactic analysis

A grammar for lexical and syntactic analysis has the following structure:

variable1: $expr_1$;
variable2: $expr_2$;
...
TOKEN1: $texpr_1$;
TOKEN2: $texpr_2$;
...

where `variable1`, `variable2`, ... are names of variables (which must start with a lowercase letter), `TOKEN1`, `TOKEN2`, ... are names of lexical classes (which must start with an uppercase letter), each $expr_i$ is an expression defining the context-free language associated to the corresponding variable, and each $texpr_i$ is an expression defining the regular language associated to the corresponding lexical class. The previous section has already detailed the kind of expressions allowed for each $texpr_i$; the kind of expressions allowed for each $expr_i$ is similar, but with two modifications:

- The operators `~` and `..` are forbidden.
- Names of variables and of lexical classes are allowed as basic expressions.

The first variable defined in the grammar is considered the starting point of the parser; the remaining variables may appear in any order. On the other hand, the order in which lexical classes are listed is significant. This is because when the lexer fetches the next token from the input it proceeds as follows: first, the lexer consumes the longest part of the input that can be matched against any class, and second, that part is labeled as belonging to the matching class that appears first in the description. Note that literal strings occurring in the definition of the variables implicitly define their own nameless lexical classes, and that all of them have precedence over the named classes.

To conclude, we mention two simplifications on the tokenization. First, it is not necessary to specify ignored lexical classes since they are considered implicit. Ignoring a class means that any token of that class is never forwarded from the tokenization step into the syntactic step. Usual ignored classes are, e.g., those matching white space or comments. Second, the empty word is implicitly removed from all the named lexical classes. Thus, a class defined like `'0'..'9'+` or as `'0'..'9'*` behaves identically.

2.1 Parser semantics: lookahead, conflicts, and eagerness

We interpret the grammar as a predictive parser. When these parsers reach a state that has several paths to proceed forward, they decide which one to follow by analyzing the lookahead information: the parser chooses a promising alternative by looking (without consuming) the next tokens that will be read from the input. Unfortunately, in some cases the lookahead might not suffice to provide an unambiguous decision between all the promising alternatives. When this happens, it is said that there are conflicts.

We consider an interpretation with one single token of lookahead, combined with an eager resolution of conflicts that guarantees that decisions will always be unambiguous. To describe such resolution of conflicts, we show with pseudocode how the basic constructions of the grammar are interpreted:

- `a: T ;`
  This base case simply reads the input, checking whether there has been an error, i.e.:

```
procedure a() {
  if lookahead is T
    CONSUME_TOKEN
  else
    SYNTAX_ERROR
}
```

- `a: ;`
  This other base case is trivial to parse:

```
procedure a() {
  // nothing to do
}
```

- **a: b c ;**
  This simply delegates the parsing to **b** and afterwards to **c**, i.e.:

```
procedure a() {
  b()
  c()
}
```

- **a: b | c ;**
  This requires to choose between delegating the parsing to **b** or to **c**. The first one that seems promising is chosen, i.e.:

```
procedure a() {
  if lookahead in First(b) or (nullable(b) and lookahead in Follow(a))
    b()
  elif lookahead in First(c) or (nullable(c) and lookahead in Follow(a))
    c()
  else
    SYNTAX_ERROR
}
```

  where, intuitively, $First(\texttt{b})$ and $First(\texttt{c})$ are the lexical classes of the tokens that may appear in first place in the part of the input consumed by the procedures for **b** and **c**, respectively, $Follow(\texttt{a})$ are the lexical classes of the tokens that may immediately follow any part of the input consumed by the procedure for **a** (possibly including the special end-of-input), and $nullable(\texttt{b})$ and $nullable(\texttt{c})$ are true iff the procedures for **b** and **c** may consume nothing from the input, respectively. To define these concepts in a simple way, let us treat the grammar as a generative formalism instead of describing a parser, i.e., as a classical CFG. Let G be a CFG with a set of variables $\mathcal{V}$, start symbol S, and alphabet Σ. Let \$ be a symbol not in $\Sigma \uplus \mathcal{V}$ denoting the end-of-input. Let λ denote the empty word. Then, for each variable $X \in \mathcal{V}$ we have:

$$\begin{aligned} First(X) &= \{a \in \Sigma \mid \exists w \in \Sigma^* : X \to_G^* aw\} \\ Follow(X) &= \{a \in \Sigma \uplus \{\$\} \mid \exists \alpha, \beta \in (\Sigma \uplus \mathcal{V} \uplus \{\$\})^* : S\$ \to_G^* \alpha X a \beta\} \\ nullable(X) &= \text{true if } X \to_G^* \lambda\text{, false otherwise} \end{aligned}$$

- **a: b* ;**
  This delegates the parsing to **b** repeatedly while it seems promising to do so, i.e.:

```
procedure a() {
  while lookahead in First(b) do
    b()
}
```

The semantics of any other kind of construction of the grammar can be easily deduced from the previous ones: it suffices to observe that any of the missing constructions can be reduced to the previous ones by, e.g., adding auxiliary variables or named lexical classes. Also note that, even though the previous interpretations resolve any conflict, it is still possible that the parser enters an infinite recursion/loop.

3 Abstract syntax tree (AST) construction

The process of parsing the input also involves constructing a tree that describes its syntactic structure. This tree is called the abstract syntax tree (AST), although it is actually a forest (i.e., a list of trees). Thus, the pseudocode shown in the previous section was a simplification, as the procedure associated to each variable of the grammar also handles the construction of

an AST. Initially, each such procedure starts its execution with an empty AST and updates it with each consumed token and with each sub-AST obtained when calling other procedures. The way this update is done can be configured in the grammar with the directives `!` and `^`:

- `variable: ... TOKEN ... ;`
  `variable: ... 'token' ... ;`
  `variable: ... other ... ;`
  When consuming a token that has neither `!` nor `^`, it is simply appended as the right-most child of the root of the AST thus far constructed by `variable`. The last case behaves analogously, i.e., the sub-AST obtained when calling the variable `other` is appended as the right-most child of the root.

- `variable: ... TOKEN! ... ;`
  `variable: ... 'token'! ... ;`
  A consumed token that has the directive `!` is not inserted into the AST.

- `variable: ... TOKEN^ ... ;`
  `variable: ... 'token'^ ... ;`
  A consumed token that has the directive `^` becomes the root of the AST thus far constructed by `variable`.

- `variable^: ... ;`
  This directive `^` signals that the variable itself becomes the root of the constructed AST once `variable` finishes its execution.

We detail the behaviour of the AST construction with examples. Consider for instance

```
sum: NUMBER ('+'^ NUMBER)* ;
NUMBER: '0'..'9'+ ;
```

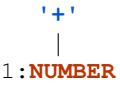
and the input `1+2+3`. The parsing and the AST construction proceed as follows:

- The execution starts at `sum` by consuming the token `1:NUMBER` and appending it to the current AST, which was empty. Hence, the current AST is just:

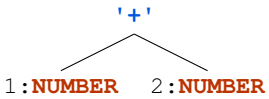
  `1:NUMBER`

  Note that this node is not considered the root of the AST since it did not have `^`. Hence, if the next token read also lacked any directive, it would be appended as a sibling of `1:NUMBER` instead of as its child or its parent (thus, the resulting AST would be a forest of two single-node trees).

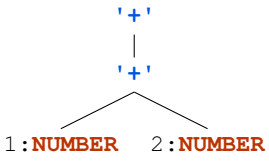
- Next, the execution enters into the `*`-loop: it reads the first literal `'+'` and acts according to the directive `^`, that is, `'+'` becomes the root of the current AST:

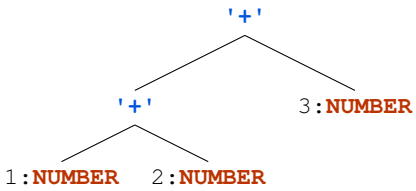
- Next, it consumes the token `2:NUMBER` and appends it as the right-most child of the root since it specifies no directive:

- Then, the execution enters into a new iteration of the `*`-loop: it reads the second literal `'+'` and makes it the new root of the AST due to the directive `^`:

- Finally, it consumes the token `3:NUMBER`, appending it as the right-most child of the root due to the lack of any directives:



Note that the constructed AST corresponds to an interpretation of `+` as an operator with left associativity. If we wanted to parse the same language but construct the AST as if `+` had right associativity we could use

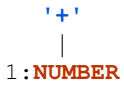
```
sum: NUMBER ('+'^ sum)? ;
NUMBER: '0'..'9'+ ;
```

which on the same input `1+2+3` proceeds as follows:

- The execution starts at `sum` by consuming the token `1:NUMBER` and appending it to the current AST, which was empty. Hence, the current AST is just:

  `1:NUMBER`

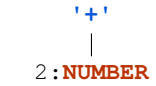
- Next, the execution enters into the optional `?`-block: it reads the first literal `'+'` and acts according to the directive `^`, that is, `'+'` becomes the root of the current AST:



- It then makes a recursive call into `sum`, which begins the construction of a new AST:
  - Consumes the token `2:NUMBER` and appends it to the new AST that is being constructed, which was empty:

    `2:NUMBER`

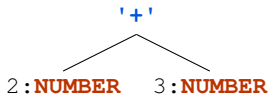
  - The execution enters into the optional `?`-block and reads the second literal `'+'`, which becomes the root of the new AST due to the directive `^`:



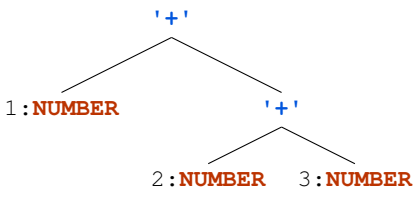
  - Makes a new recursive call into `sum`:
    - Consumes the token `3:NUMBER` and appends it to the brand new AST that is being constructed, which was empty. The execution of this call does not enter into the optional `?`-block, and thus, returns the AST:

      `3:NUMBER`

    The recursively-obtained sub-AST is appended as the right-most child of the root, finishing the execution of this call and thus returning the AST:



  The recursively-obtained sub-AST is appended as the right-most child of the root:



As a last example, if we wanted to parse the same language but discard the + operators from the AST and just keep the list of operands, we could use

```
sum^: NUMBER ('+'! NUMBER)* ;
NUMBER: '0'..'9'+ ;
```

where the ^ of **sum** has been added for convenience to guarantee that the AST is rooted by **sum** (otherwise, if ^ was missing, the AST constructed would be a forest of single-node trees). The AST produced for the same input `1+2+3` is: