Abstract

We present the design and implementation of a probabilistic, nondeterministic programming language called Invocat for defining and generating from context-free grammars. We define Invocat’s lexical, concrete, and abstract syntaxes as well as its operational semantics. We implement the language in Lua as a lexical analyzer, parser, and interpreter.

1 Introduction

In this paper we present the design and implementation of a probabilistic, nondeterministic programming language called Invocat for defining and generating from context-free grammars. We discuss the various parts of the language and provide formal definitions as well as examples.

The design of a programming language, a creative process influenced by any number of considerations, is ultimately and succinctly formalized in the language’s syntax and semantics. The syntax determines the language’s outward characteristics, while the semantics define how the language’s terms are evaluated and thus their meanings.

Syntax and semantics are the basis of implementation. For a programming language, the path from input to output comprises lexical analysis, parsing, and evaluation; the syntax and semantics act as specifications for these processes.

In the following section we introduce Invocat. We proceed to formalize the language’s lexical, concrete, and abstract syntaxes as well as its operational semantics. We implement Invocat in Lua [3] as a lexical analyzer, parser, and interpreter.

2 Invocat

Invocat is a language for randomly generating strings from recursive grammars. Our inspiration comes from online random generators like Abulafia [8] and the Donjon Random Generator [2], and more indirectly from the random tables in various role-playing game materials, both in print and on the Web. The latter provide a clarity of vision into the possibility space that the former
obscure with excessive syntax. Published random tables tend not to recurse, but cross-references are not uncommon. A significant part of their understandability and appeal is in their presentation. Invocat is an attempt to harness the wonderful combinatorial expressivity of a random generator with minimal trappings.

By way of introduction, consider the following Invocat program:

```
s: hello, world! | (silence)
(s)
→₀ hello, world!
→₁
```

The notation → introduces output. Because Invocat is nondeterministic, we shall use subscripts to separate possible outcomes.

In the program above, the first line defines a list named s that has two elements: the string hello, world! and a reference to an as yet undefined list named silence. Definitions do not produce any output. The second line is a reference to s, which we just defined. This reference is evaluated, resulting in either hello, world! or the empty string, which results from evaluating an undefined list.

If all we wanted to do was print hello, world! we could do that in one line:

```
hello, world!
→₀ hello, world!
```

Invocat also features resolutions, which are references that retain their evaluations. This allows a particular evaluation of a reference to be reused throughout the program. Consider the following example in which Auge finds a new artifact. Since we’d like to refer to it more than once without having it potentially evaluate to something new, we can create a resolution, a₁, for that purpose.

```
artifact: (quality) (weapon)
  quality: gleaming | dull | rusty | battered | bent
  weapon: sword | axe | spear | hammer | mace | dagger
a₁ <- (artifact)
Auge found a (a₁) among the ruins, and Io a (artifact), but all \ anyone would talk about was Auge’s new (a₁).
→₀ Auge found a gleaming mace among the ruins, and Io a dull hammer, but all anyone would talk about was Auge’s new gleaming mace.
```

Invocat takes as design goals readability and ease of use. These are certainly, though not exclusively, syntactic considerations. Let us now examine Invocat’s syntax.

### Syntax

Syntax defines the set of strings that belong to a language. It governs the language’s physical structure and appearance; and on the surface level, determines the ease of reading and writing programs.
comment ::= [-][-]\s+*$
name ::= [\w-_,.?!]+
lparen ::= [[]]
rparen ::= []]
colon ::= \s?[\-]
arrow ::= \s?[<-][-]
pipe ::= [[]]
escape ::= \\[n()]
break ::= \$ 
punct ::= \p
white ::= \s

Figure 1: Lexical syntax of Invocat.

statement ::= definition
| resolution
| item

definition ::= name ‘:’ itemlist
resolution ::= name ‘<-' itemlist
itemlist ::= item {itemlist}
item ::= literal {item}
| reference {item}
reference ::= ‘(’ name ‘)’
literal ::= ink {white literal}

ink ::= name {ink}
| punct {ink}
| escape {ink}

escape ::= ‘\(' 
| ‘\)’
| ‘\n’

Figure 2: Concrete syntax of Invocat.

definition ::= name: list
resolution ::= name <- list
item ::= literal
| (name)
| item1 item2

Figure 3: Abstract syntax of Invocat.
Invocat has a minimalist syntax. Literal strings are defined without quotation marks or other delimiters. Whitespace is significant inside of literals, but not outside. The present implementation uses the pipe to separate list items; we have in mind an alternate syntax using newlines, but we save this relatively minor feature for future work.

The first step in implementing a language is lexical analysis, the process that turns an input file into a token stream to be consumed by the next higher level of analysis. Although tools such as Lex are commonly used to automatically generate lexical analyzers, we wrote Invocat’s lexer, and indeed the rest of the language, in Lua. Invocat’s lexical syntax is given using regular expressions in Figure 1.

Invocat’s concrete syntax is shown in Figure 2. Here we use the notation \{A\} to denote 0 or more As. The concrete syntax is naturally more interesting than the lexicon: it is the bones of the language. To parse Invocat, we chose to employ the recursive descent parsing algorithm. We studied the top down operator precedence [1][6] technique but chose recursive descent for its conceptual simplicity and direct correspondence with the syntax rules: it simply requires a function to parse each element of the concrete syntax. The result of parsing is, of course, an abstract syntax tree.

In our implementation, abstract syntax elements are represented as tagged tables. An abstract syntax for Invocat is given in Figure 3. The definitions retain elements of the concrete syntax for readability, especially in anticipation of the semantics to follow. In the definitions, the term literal denotes any string, including the empty string; and list denotes a list of items \{item_1,...,item_n\}.

**Semantics**

Not all of a language’s properties derive from its syntax: a language without semantics is meaningless. Features such as definitions and resolutions and the admission of recursion are semantic in nature.

The call-by-value big-step operational semantics for Invocat are defined by the rules given in Figure 4. In the initial state \(\sigma\) all keys map to a list containing the empty string. The notation list(\(\omega\)) indicates a uniformly random selection from list; and literal_1 \cdot literal_2 is concatenation.

Invocat is nondeterministic in that the results of evaluating an expression more than once in state \(\sigma\) are not guaranteed to be the same. However, we note that the evaluation is syntax directed, and observe that items always evaluate to literals if the evaluation terminates.

The evaluation of a definition updates the value of name in state \(\sigma\) to list.

The evaluation of a reference in state \(\sigma\) takes an item at random from the list to which it refers and yields its evaluation in state \(\sigma\). Note that the recursive definition of item means that the thing chosen may itself be a reference, but such evaluations will eventually result in a literal assuming the program doesn’t contain any infinite referential loops.

Of particular note is the final rule, which describes the evaluation of a resolution. It says that to evaluate a resolution in state \(\sigma\), select an element at random from the list, evaluating it in \(\sigma\) to obtain a literal. Finally, update the value of name in state \(\sigma\) to a list containing only the resulting literal.

Having formalized the language, let us examine Invocat in use.
Example

We exercise Invocat by recognizing its potential to express Markov chains for text generation. The idea is essentially to serialize a document into a set of Invocat definitions. We have written a short Lua program to parse an input document and produce Invocat statements based on a few simple conventions.

Whereas in other languages, the data structure used to generate the chain exists in memory, in Invocat it is made explicit in the form of a valid program!

For every pair of words appearing adjacent to each other in the text, we produce a definition that has as its name the two words joined by an underscore, and as its value the second word as a literal followed by a reference to a definition whose name is the second word joined with the next word from the text.

If two adjacent words already appear in the list of definitions, we simply add subsequent entries as alternate expansions.

Consider the input text:

it is possible to be at home and possible to feel at sea

To start, we observe that two empty strings must produce the first word in the text. By this reasoning, we can also begin generating by giving a single reference to (._). We end up with the following short program.

```lua
-- definitions
_: (._it)
_it: it (it_is)
it_is: is (is_possible)
```
We have developed a simple example here, but a complete listing for a chain trained on the first chapter of Melville’s *Moby Dick* is included with the source code. Additionally we include a detailed arms and armor generator.

### 3 Conclusion

We have presented Invocat, a small, nondeterministic language for generating strings from recursive grammars. It is focused on minimal syntax and ease of expression. We have provided a reference implementation and documented its syntax and semantics. We have demonstrated its use in generating item lists and representing Markov chains.

Invocat is not Turing complete. Rather, it is comparable in power to nondeterministic pushdown automata and offers a fairly direct way to describe and exercise those abstract machines.

In future work, we hope to extend its capabilities. Weighted sampling is an obvious enhancement though it would necessarily clutter the syntax and its effects can be simulated in the current version of the language simply by repeating list entries until the probabilities are correct.

We also see the need for an option to draw from a list without replacement. This would be of use ensuring a series of resolutions had distinct results—fixing a list of distinct names or weapon types for instance. Also this would make it simple to do things like draw Tarot cards from a deck.

We would eventually like to add referential indirection—the ability to make a reference not just by a literal name, but by the result of another expression.

We would like to explore allowing references to take parameters. We made several false starts on this path and have yet to settle on the syntax, let alone the semantics of substitution in the language. However, the basic idea is that, given a reference \( r \) and an applied parameter \( p \), any of \( r \)'s inner references bound by the parameter are replaced with \( p \).

Finally, although Invocat’s syntax is quite minimal and legible, it could be improved with more attention. Even without any of these features, however, Invocat is already magical.
References


