$Grail^+$

CS 4490Z Undergraduate Thesis Report

Peter Goodman

March 29, 2011

Supervisor: Prof. Sheng Yu

Acknowledgments

I am grateful to my supervisor, Professor Sheng Yu, for his academic guidance over the past three years.

I owe my deepest grattitude to my parents and family for supporting me in my undergraduate studies.

Lastly, I would like to thank my partner Helen for putting up with all of the time I spent debugging the software of this project.

Peter Goodman

Contents

List of Abbreviations

API	Application programming interface.
CFG	Context-free grammar.
CLI	Command-line interface.
EDSL	Embedded domain-specific language.
ϵ-CNF	Chomsky normal form with additional support for empty languages.
ϵ-GNF	Greibach normal form with additional support for empty languages.
ϵ-NFA	Non-deterministic finite automaton with ϵ-transitions.
ϵ-NPDA	Non-deterministic pushdown automaton with ϵ-transitions.
FLO	Formal language object.
FLTL	Formal language template library.
G++	The GNU compiler collection's `C++` compiler.
ICPC	Intel `C++` compiler.
LLVM	Low-level virtual machine.
VC++	Microsoft Visual `C++` compiler.

1 Introduction

Grail$^+$ is a system for manipulating formal languages. It operates on three types of formal language objects (FLOs): context-free grammars (CFGs), non-deterministic pushdown automata (ϵ-NPDAs), and non-deterministic finite automata (ϵ-NFAs).

Grail$^+$ is implemented as an application framework on top of the formal language template library (FLTL), a `C++` library explicitly developed for use by *Grail*$^+$.

Grail$^+$ includes implementations of the following algorithms over FLOs:

- Earley's algorithm for testing word membership in the language generated by a CFG. [3, 1]
- Removal of ϵ-productions from a CFG. [10]
- Removal of unit productions from a CFG. [10]
- Removal of unreachable and non-generating productions from a CFG. [5]
- Removal of left-recursion from a CFG. [8]
- Removal of ϵ-transitions from an ϵ-NFA. [10]
- Conversion of a CFG into a form where the right-hand side of each production has no more than two symbols.
- Conversion of a CFG into CNF. [10]
- Conversion of a CFG into GNF. [4]
- Conversion of a CFG into an ϵ-NPDA. [10]
- Conversion of a ϵ-NPDA into a CFG.[10]
- Intersection of a ϵ-NPDA with a CFG. [10]

Grail+ exposes six of the above algorithms as command-line tools.

This document gives an overview of the features of the newest version of *Grail*$^+$ as well as some implementation details of some of the interesting features of the FLTL.

2 History

$Grail^+$ has existed in numerous forms since the 1990s. Originally, $Grail^+$ was designed to be a set of command-line tools for manipulating regular languages. Since then, extensions for manipulating Mealy machines, alternating finite automata (AFAs), and CFGs have been added by various third parties.

Unfortunately, disorganization has lead to the fragmentation of the original $Grail^+$ source code. Old versions of $Grail^+$ are independently maintained by students at various different universities. Further, the foundations of prior versions of $Grail^+$ have seen little maintenance since the new millenium. As a result, the various branches of the prior versions of $Grail^+$ are messes of poorly structured `C++` classes that have been of diminishing usefulness to the academic community.

This is not to say that prior versions of $Grail^+$ are unusable; however, they are increasingly difficult to use. In order to compile versions of $Grail^+$ based on the original source code using modern compilers[1], one must disable many of their error-detection systems. This is in stark contrast with one the original design goals of $Grail^+$: the `C++` programming language was chosen as the implementation language because of stronger/stricter guarantees that `C++` compilers can make. Further, it is difficult to develop new tools for prior versions of $Grail^+$ as expert `C++` knowledge is required to understand the many implementation details of the underlying $Grail^+$ data structures.

The software developed for this project represents a re-implementation of $Grail^+$ and shares no code with any prior versions of $Grail^+$.

[1] G++, Clang++, VC++, ICPC.

3 Goals and Motivations

This version of the $Grail^+$ project had the following goals:

1. **Separation of concerns**: algorithms should be independent of the formal language data structures on which they operate, and command-line tools should be independent from the algorithms powering them. This goal was met by the separation of the FLTL from $Grail^+$ and by the distinction of command-line tools and algorithms within $Grail^+$. The motivation for this goal came from prior versions of $Grail^+$: all algorithms were implemented as methods of one of the formal language data structures. This resulted in very tight coupling between data structures and cluttered the public APIs of all of the data structures.

2. **Declarative over imperative**: APIs that allow the programmer to express an operation should be preferred to ones that require that the programmer describe how to perform that operation. This goal is embodied in the pattern matching meta-programming facilities in the FLTL. The motivation for this goal came from prior versions of $Grail^+$: by virtue of being methods of classes, each algorithm had access to the internals of their respective data structures. As a result, algorithms depended on the implementation details of every data structure involved in a particular computation.

3. **Homogeneity**: from the outside, $Grail^+$ has always been a tool for the symbolic manipulation of formal languages. From the inside, the intention of a given piece of code in prior versions of $Grail^+$ was obscured by implementation details, memory management, traversal of data, and the concerns for the memory used to represent various data structures. The new version of $Grail^+$, by means of the FLTL, had the goal of enabling symbolic manipulations of FLOs at the programming level. This goal is an extension of the first two goals, and was met by exposing opaquely typed objects to programmers, creating a declarative API, and supporting powerful and expressive pattern matching facilities. For example, CFGs have opaque types for terminals, non-terminals, symbols, symbol strings, and productions. This goal was motivated by the second goal, declarative over imperative, for the purpose of making it easier for programmers to express and implement algorithms.

4. **Genericity**: this goal is shared by all versions of $Grail^+$ and was one of the motivating factors for choosing `C++` as the implementation language of $Grail^+$. `C++` has a template system which allows one to create parameterizable code that will work for any set of parameters that support the desired operations on the types/values of those parameters. $Grail^+$ goes beyond previous versions insofar as it can parameterize basic types[1] as well as more complex types through a system of type traits.

5. **Performance**: the purpose of $Gail^+$ has always been to symbolically manipulate FLOs. As a result, $Grail^+$ has never been and likely never will be a leader where performance is concerned. Prior versions of $Grail^+$ made use of naïve data structures for representing sets and vectors. As a result, performance suffered for large input sizes. Anecdotal evidence suggests that the new $Grail^+$ has poor performance where expected (e.g. converting a CFG into GNF), but acceptable and sometimes excellent performance where it matters (e.g. converting a CFG into CNF, testing language membership).

[1] `int`, `char`, `char *`, etc.

4 Architectural Overview

4.1 Formal Language Template Library

The FLTL is composed of three primary template classes (`CFG`, `NFA`, `PDA`), their support template classes, and helper template classes.

4.1.1 `CFG`, `NFA`, `PDA` and their Support Template Classes

Each primary template class supports operations that affect the state of instances of that template class. Each primary template class also exposes certain support template classes as opaque types[1], where instances of these types are immutable.

The primary and support template classes are separated out into their logical components (e.g. the components of a CFG are symbols, terminals, non-terminals, symbol strings, and productions) to make it easier to understand what operations a given piece of code performs. Unfortunately, the interactions between the various support template classes causes high coupling. The downside is that this coupling increases the difficulty of maintaining the core data structures. The upside of this coupling is that, from the algorithm writer's perspective, the implementation of an FLO appears cohesive and interactions between the various template classes involved work as expected.

More information on these template classes can be found in the FLTL API chapter.

4.1.2 Helper Template Classes

The FLTL includes a number of helper template classes. These template classes are used internally within the FLTL. In general, the purpose of the helper classes is to manage memory in some way. For example, `BlockAllocator` allocates pools of objects, `StorageChain` allows one to specify dependencies between statically allocated objects so that object destruction of static memory is well-ordered, and `UnsafeCast` allows one to convert memory of one type into memory of another type in a way that `C++` compilers will accept, even with the strictest referential transparency warning/error settings enabled.

4.2 Grail$^+$

Grail$^+$ contains algorithm implementations as well as tools that expose some of the aforementioned algorithms to the command-line.

[1] For example, `CFG` exposes `production_type` as `fltl::CFG<AlphaT>::production_type` for some basic type or type trait `AlphaT`. Each exposed type is actually an instantiation of a support template type. `production_type` is the type `fltl::cfg::OpaqueProduction<AlphaT>`, where `fltl::cfg` is the namespace containing all support template classes for the `CFG` template class.

4.2.1 Algorithms

Algorithms are template classes that expose a static `run` method. Algorithm classes operate on FLTL data structures (`CFG`, `NFA`, `PDA`). The primary FLTL data structures tend to be "top heavy" and so the `run` method does not usually return any value. Instead, the object to be returned from an algorithm is passed by reference to that algorithm as a formal parameter to the `run` method.

Algorithms are not required to be pure; however, it is encouraged. Some algorithms have logging mechanisms to report information back to the user. Also, some algorithms modify the state of their inputs. Importantly, all of the current algorithms make the guarantee that if their input is modified, the language generated/accepted by their input is not modified. In future, stricter restrictions can be placed on the behaviour of an algorithm with respect to its inputs.

4.2.2 Tools

Grail$^+$ tools all share the same structure. Any tool that does not meet the structure will not compile. Tools are required to expose the following public API:

- A static `TOOL_NAME` member with the name of the tool. This name must uniquely identify a given tool so that the main program is able to dispatch to the proper tool. For example, the tool template class `CFG_TO_CNF` has `"cfg-to-cnf"` as its `TOOL_NAME`.
- A static method `declare` that takes in a reference to an instance of the class `CommandLineOptions` and a Boolean flag specifying whether or not the help information for the given tool is being displayed. This method is responsible for declaring which command-line options a tool accepts. When requesting the help information of a tool, it is undesirable to do extensive checking of the options, hence the Boolean flag.
- A static method `help` which, when called, prints out the help information for its tool. This information contains a short description of the tool as well as the command-line options supported by the tool. The information outputted by `help` should be specific to the tool.
- A static method `main` that takes in a reference to an instance of the class `CommandLineOptions`. `main` is responsible for handling all user input, reading descriptions of FLOs from files and importing them into memory, running algorithms on the FLOs, and the outputting the results back to the user.

The structure of tools was motivated by the command-line interface (CLI) of Valgrind [9], a tool that was used extensively in the testing of *Grail*$^+$. At first, this structure appears unusual; however, it allows for:

- separate definitions of global and local command-line options;
- conditional delaying of general error checking of command-line options;
- delaying of specific error checking of command-line options until the program is committed to running a tool, and;
- no external dependencies for managing help information.

The CLI of *Grail*$^+$ is meant to be flexible enough to provide a good user experience and to move error-checking to when and where it is needed. As such, a first-time user of *Grail*$^+$ can quickly learn to use the CLI, even if their first action is to run the executable without any arguments or inputs.

5 Command-Line Tools

Grail$^+$ exposes several algorithms over FLOs as command-line tools. Each tool is accessible through a single executable file. The following are some example usages of some of *Grail*$^+$'s tools:

```
./grail --tool=cfg-to-cnf --verbose ./math.cfg
./grail --tool=cfg-parse --verbose --predict ./math.cfg --stdin
./grail --tool=pda-intersect-nfa --help
```

The following command-line options apply to all *Grail*$^+$ tools:

	Requires Value	Accepts Value	Optional	Description
`--help` `-h`		×	✓*	By default, this option displays help information related to all of *Grail*$^+$. The help information includes this list of global command-line options. If a tool is selected using `--tool` then help information related to the specific tool is also shown.
`--test`		×	✓*	This option causes *Grail*$^+$ to run its test suite. After compiling *Grail*$^+$, one's first action should be to run *Grail*$^+$ with this command-line option. The purpose of the test suite is to test the basic functionality of the FLTL.

	Requires Value	Accepts Value	Optional	Description
`--tool`	✓	✓	✓*	Selects a specific tool to run. *Grail*$^+$ currently supports six tools: • `cfg-parse`, • `cfg-to-cnf`, • `cfg-to-gnf`, • `cfg-to-pda` • `pda-intersect-cfg` • `pda-to-cfg` For example, if one wanted to select `cfg-parse`, the one would use either `--tool=cfg-parse` or `--tool cfg-parse`.
`--tools`		×	✓*	Display a list of the tools supported by the currently compiled version of *Grail*$^+$.
`--verbose` `-v`		×	✓	Print out logging information to the user. The information printed out is printed to `stderr`. The information printed out is often helpful for determining the progress of a tool and also for determining the sizes of the inputs and outputs of a tool.
`--version`		×	✓*	Displays the current version of *Grail*$^+$.

While all of the global command-line options are marked as optional, at least one of the above options marked with a "*" is always required.

Grail$^+$ supports the following tools, selectable using the `--tool` command-line option, and their respective command-line options.

In the following sections, positional arguments to command-line tools are denoted by `<#>`[1], where `#` represents the position of the argument and `<0>` is the first positional argument. Positional (`<#>`) and keyword (starting with `-` or `--`) arguments can be interspersed. Positional arguments are ordered whereas keyword arguments are not.

5.1 `cfg-parse`

This tool takes as input the name of a file containing a description of a CFG and the name of a file containing the word to parse according to the aformentioned CFG. If the `--stdin` option is specified then

[1] Note: `<#>` is notation for describing positional arguments within this report. Actual position arguments to *Grail*$^+$ are not surrounded with `<` and `>`.

the word to parse must be supplied by the user at the command line. The output of the program is `"Yes."` if the word can be genertated by the input CFG, and `"No."` if the input word cannot be generated.

Grail$^+$ has no understanding of the lexical rules of the alphabet symbols of the words generated by the grammars that it manipulates. As such, it is up to the user to delimit the alphabet symbols of a word by line breaks. That is, any sequence of bytes—including all non-null ASCII characters and all valid UTF-8 codepoints—on a non-empty line is seen as a single alphabet symbol. At first this appears tedious; however, it ensures that *Grail*$^+$

- never projects assumptions valid for one language to other languages where the assumptions are invalid[2].
- never has to deal with context-sensitive disambiguations of strings of characters[3].

Grail$^+$ supports variable terminals (also known as dynamic terminals). From a programming perspective, there is no distinction between a variable terminal and a normal terminal. From a formatting perspective, a variable terminal is named in the same way as a variable but has no productions. From the perspective of the `cfg-parse` tool, any alphabet symbol that isn't recognized as a terminal of the grammar can be assigned[4] to any variable terminal of the grammar.

For example, `A` and `B` are variable terminals in the following grammar:

$$S \rightarrow \texttt{A B}$$

When used as the input CFG to `cfg-parse`, `cfg-parse` will report `"Yes."` if and only if its input file has only two non-empty lines.

In addition to *Grail*$^+$'s global command-line options, the `cfg-parse` tool accepts the following command-line options:

	Requires Value	Accepts Value	Optional	Description

[2]For example, insignificant whitespace in the `C` programming language versus significant whitespace in the `Python` programming language.

[3]For example, the sequence of characters `+++` can be converted into at least five distinct sequences of alphabet symbols in the `C` programming language, many of which are illegal depending on the surrounding context.

[4]Variable terminal assignments are mutually independent. That is, the same string can be assigned to different variable terminals throughout the parsing process, even if a past assignment has succeeded.

	Requires Value	Accepts Value	Optional	Description
`--predict`		✗	✓	This tells the tool to compute $FIRST(v)$ for every variable v of the input CFG. $FIRST(v)$ is defined as the set of all terminals that can appear as the first terminal in any string generated by v. Computing the $FIRST$ sets can be a slow operation, especially for large grammars with a lot of left recursive productions and nullable productions. However, in many cases, computing the $FIRST$ sets speeds up the parsing process as it allows the parser to ignore any production that provably won't appear in the final derivation of the string.
`--stdin`		✗	✓*	This option instructs the tool to read the input word from `stdin` instead of taking a file name as the second positional command-line argument.
`<0>`			✗	The name of a file containing a properly formatted description of a CFG.
`<1>`			✗*	The name of a file containing the "word" to parse, where each alphabet symbol of the word appears on a new line in the file. If `--stdin` is specified then this positional command-line argument is not used.

5.2 `cfg-to-cnf`

This tool takes as input the name of a file containing a description of a CFG and outputs a description of a CFG that is in ϵ-CNF. Specifically, every production of a grammar in ϵ-CNF must be in one of the following forms:

$$\begin{aligned} \gamma &\rightarrow \alpha\ \beta \\ \gamma &\rightarrow \texttt{t} \end{aligned}$$

Where γ, α, and β are arbitrary non-terminals and `t` is an arbitrary terminal. In addition, if S is the start variable of a CFG in ϵ-CNF then $S \rightarrow \epsilon$ is a valid production form if and only if S generates the empty string in the original grammar.

In addition to $Grail^+$'s global command-line options, the `cfg-to-cnf` tool accepts the following command-line options:

	Requires Value	Accepts Value	Optional	Description
`--stdin`		✗	✓*	This option instructs the tool to read the input CFG from `stdin` instead of taking a file name as a positional command-line argument.
`<0>`			✗*	The name of a file containing a properly formatted description of a CFG. If `--stdin` is specified then this positional command-line argument is not used.

5.3 `cfg-to-gnf`

This tool takes as input the name of a file containing a description of a CFG and outputs a description of a CFG that is in ϵ-GNF. Specifically, every production of a grammar in ϵ-GNF must have the form $\alpha \rightarrow \texttt{t}\ \gamma$, where α is an arbitrary non-terminal, `t` is an arbitrary terminal, and γ is a sequence of zero or more non-terminals. In addition, if S is the start variable of a CFG in ϵ-GNF then $S \rightarrow \epsilon$ is a valid production form if and only if S generates the empty string in the original grammar.

In addition to *Grail*$^+$'s global command-line options, the `cfg-to-gnf` tool accepts the following command-line options:

	Requires Value	Accepts Value	Optional	Description
`--stdin`		✗	✓*	This option instructs the tool to read the input CFG from `stdin` instead of taking a file name as a positional command-line argument.
`<0>`			✗*	The name of a file containing a properly formatted description of a CFG. If `--stdin` is specified then this positional command-line argument is not used.

This tool performs poorly in many cases. As such, for large grammars, this tool should not be seen as a practical method of converting a CFG into ϵ-GNF.

5.4 `cfg-to-pda`

This tool takes as input the name of a file containing a description of a CFG and outputs a description of a ϵ-NPDA that accepts the language generated by the input CFG.

In addition to *Grail*$^+$'s global command-line options, the `cfg-to-pda` tool accepts the following command-line options:

	Requires Value	Accepts Value	Optional	Description
`--stdin`		✗	✓*	This option instructs the tool to read the input CFG from `stdin` instead of taking a file name as a positional command-line argument.
`<0>`			✗*	The name of a file containing a properly formatted description of a CFG. If `--stdin` is specified then this positional command-line argument is not used.

5.5 pda-intersect-nfa

This tool takes as input the name of a file containing the description of an ϵ-NPDA and the name of a file containing the description of an ϵ-NFA and outputs an ϵ-NPDA that accepts the intersection of the languages accepted by the input ϵ-NPDA and the input ϵ-NFA.

In addition to *Grail*$^+$'s global command-line options, the `pda-intersect-nfa` tool accepts the following command-line options:

	Requires Value	Accepts Value	Optional	Description
`<0>`			✗	The name of a file containing a properly formatted description of an ϵ-NPDA.
`<1>`			✗	The name of a file containing a properly formatted description of an ϵ-NFA.

5.6 pda-to-cfg

This tool takes as input the name of a file containing the description of an ϵ-NPDA and outputs a description of a CFG that generates the language accepted by the input ϵ-NPDA.

In addition to *Grail*$^+$'s global command-line options, the `pda-to-cfg` tool accepts the following command-line options:

	Requires Value	Accepts Value	Optional	Description
`--stdin`		✗	✓*	This option instructs the tool to read the input ϵ-NPDA from `stdin` instead of taking a file name as a positional command-line argument.

	Requires Value	Accepts Value	Optional	Description
`<0>`			✕*	The name of a file containing a properly formatted description of an ϵ-NPDA. If `--stdin` is specified then this positional command-line argument is not used.

It should be noted that a grammar produced by this tool is likely to perform poorly when used as an input to the `cfg-parse` tool as the grammar is likely to be highly ambiguous—especially if the input ϵ-NPDA was produced by the `cfg-to-pda` tool.

6 File Formats

6.1 CFGs

The file format for $Grail^+$ CFGs was influenced by Yacc [7], Java CUP [6], and the Natural Language Toolkit (NLTK) [2]. The file format can be described by a regular language; however, it is presented here as a CFG. Terminals are denoted using a `teletyped` font. New lines (`\n`) are the only form of significant whitespace in the CFG format.

$$
\begin{aligned}
\mathit{CFG} &\rightarrow \mathit{ProductionList} \\
\mathit{ProductionList} &\rightarrow \mathit{YaccProduction}\ \mathit{ProductionList} \\
&\rightarrow \mathit{NLTKProduction}\ \mathit{ProductionList} \\
&\rightarrow \texttt{\textbackslash n}^*\ \mathit{ProductionList} \\
&\rightarrow \epsilon \\
\mathit{YaccProduction} &\rightarrow \texttt{NonTerminal}\ \texttt{\textbackslash n}^*\ \texttt{:}\ \texttt{\textbackslash n}^*\ \mathit{String}\ \texttt{\textbackslash n}^*\ (\texttt{|}\ \texttt{\textbackslash n}^*\ \mathit{String})^*\ \texttt{\textbackslash n}^*\ \texttt{;} \\
\mathit{NLTKProduction} &\rightarrow \texttt{NonTerminal}\ \texttt{->}\ \mathit{SingleLineString}\ \texttt{\textbackslash n} \\
&\rightarrow \texttt{NonTerminal}\ \texttt{=>}\ \mathit{SingleLineString}\ \texttt{\textbackslash n} \\
\mathit{String} &\rightarrow \texttt{\textbackslash n}^*\ \texttt{NonTerminal}\ \mathit{String} \\
&\rightarrow \texttt{\textbackslash n}^*\ \texttt{Terminal}\ \mathit{String} \\
&\rightarrow \texttt{\textbackslash n}^*\ \texttt{VariableTerminal}\ \mathit{String} \\
&\rightarrow \texttt{\textbackslash n}^*\ \texttt{epsilon}\ \mathit{String} \\
&\rightarrow \texttt{\textbackslash n}^*\ \mathit{String} \\
&\rightarrow \epsilon \\
\mathit{SingleLineString} &\rightarrow \texttt{NonTerminal}\ \mathit{SingleLineString} \\
&\rightarrow \texttt{Terminal}\ \mathit{SingleLineString} \\
&\rightarrow \texttt{VariableTerminal}\ \mathit{SingleLineString} \\
&\rightarrow \texttt{epsilon}\ \mathit{SingleLineString} \\
&\rightarrow \epsilon
\end{aligned}
$$

In the above grammar, the `NonTerminal` terminal maps to the regular language generated by the regular expression $([a-ZA-Z0-9\_]^+)|(\$[0-9]^+)$. The range of non-terminals identifiers generated by the sub-expression $(\$[0-9]^+)$ represents automatically generated non-terminals.

In the above grammar, the `VariableTerminal` maps to the same language as `NonTerminal`. This ambiguity is resolved by doing two passes over every CFG description: the first pass looks for syntax errors and collects all symbols that appear to the left of a `:`, `->`, or `=>` symbol and marks those as being represented by the `NonTerminal` terminal.

In the above grammar, `Terminal`s are delimited by a a delimiter string d^n for $n \geq 1$ and $d \in \{\texttt{"}, \texttt{'}\}$ where n is an odd number. The terminal is a sequence of zero-or-more valid UTF-8 codepoints delimited on both sides by d^n. If $n = 1$ then embedding d within a terminal requires that one must escape d by prefixing it with a \, i.e. $\backslash d$. However, for $n > 1$, one can embed up to $n - 1$ consecutive copies of d within the string without needing any additional escape characters.

Finally, in the above grammar, `epsilon` is represented by itself.

Grail$^+$ automatically ignores `C`-style comments[1], `C++`-style comments[2], `Python`-style comments[3], Yacc-style declarations[4], Yacc-style code blocks[5], and `Java` CUP-style code blocks[6] in the CFG descriptions. This allows one to annotate CFG descriptions with useful information, as well as for *Grail*$^+$ to operate on grammars meant for existing tools with only minimal modifications.

By default, the variable of the first production encountered is set to be the start variable of the grammar.

6.2 ϵ-NFAs

The file format for ϵ-NFAs is the same as that of prior versions of *Grail*$^+$, with the exception that the symbol delimiter[7] is fixed one of $\{\texttt{"}, \texttt{'}\}$, with the same rules applying to ϵ-NFA symbols as does CFG terminals. As with the CFG file format, new lines are signifcant, and are delimited by the terminal `\n`.

$$
\begin{aligned}
\mathit{NFA} &\rightarrow \mathit{TransitionList} \\
\mathit{TransitionList} &\rightarrow \mathit{Transition}\ \texttt{\textbackslash n}\ \mathit{TransitionList} \\
&\rightarrow \texttt{\textbackslash n}^*\ \mathit{TransitionList} \\
&\rightarrow \epsilon \\
\mathit{Transition} &\rightarrow \texttt{State epsilon State} \\
&\rightarrow \texttt{State Symbol State} \\
&\rightarrow \texttt{(START) |- State} \\
&\rightarrow \texttt{State -| (FINAL)}
\end{aligned}
$$

In the above grammar, `State` represents an arbitrary natural number; `epsilon`, `(START)`, `(FINAL)`, `|-`, and `-|` represents themselves; and `Symbol` represents the same language represented by the `Terminal` terminal from the CFG file format.

[1] `/* ... */`
[2] `// ...`
[3] `# ...`
[4] `%...`
[5] `%{ ... %}`
[6] `{: ... :}`
[7] The symbol delimiters are a programmer-configurable setting in prior versions of *Grail*$^+$ and are limited to two single ASCII characters: left and right delimiters, respectively.

6.3 ϵ-NPDAs

The file format for ϵ-NPDAs is backward compatible with the format for ϵ-NFAs. That is, an ϵ-NFA description can be used anywhere that an ϵ-NPDA is expected.

$$
\begin{array}{rcl}
PDAA & \rightarrow & TransitionList \\
TransitionList & \rightarrow & Transition \texttt{\textbackslash n}\ TransitionList \\
 & \rightarrow & \texttt{\textbackslash n}^*\ TransitionList \\
 & \rightarrow & \epsilon \\
Symbol & \rightarrow & \texttt{Symbol} \\
 & \rightarrow & \texttt{epsilon} \\
Transition & \rightarrow & \texttt{State epsilon State}\ OptStack \\
 & \rightarrow & \texttt{State}\ Symbol\ \texttt{State}\ OptStack \\
 & \rightarrow & \texttt{(START) |- State} \\
 & \rightarrow & \texttt{State -| (FINAL)} \\
OptSymbol & \rightarrow & Symbol \\
 & \rightarrow & \epsilon \\
OptStack & \rightarrow & \epsilon \\
 & \rightarrow & \texttt{,}\ OptSymbol \\
 & \rightarrow & \texttt{,}\ OptSymbol\ \texttt{/}\ OptSymbol
\end{array}
$$

The above grammar is the same as that of ϵ-NFAs, with the addition of the *OptStack* and *OptSymbol* variables used for describing the stack manipulations of an ϵ-NPDA transition.

7 FLTL API

The following sections include the public APIs of the primary FLTL template classes. The following sections do not include the APIs of the various supporting template classes of the primary template classes.

7.1 CFG<AlphaT>

Method	Description
`add_production`	Adds a production to the CFG. Two parameters are always required for this method. The first parameter always has the type `variable_type`. The second parameter can have any one of `variable_type`, `symbol_type`, `symbol_string_type`, `terminal_type`, or `production_builder_type` types. This method will not add a production to the grammar if the production already exists. As such, the value returned by two invocations to `num_productions`—one before and one after adding a production to a grammar—could possibly by the same. The usage of `production_builder_type` as the type of the second parameter to `add_production` is encouraged when the symbol strings of a production are being constructed from an arbitrary number of components.
`add_variable`	Creates a new, automatically named variable and returns a value of type `variable_type` that can be used to operate with that variable. Variables created by `add_variable` are given names of the form $\$([0-9]^+)$. `add_variable` guarantees that the numeric component of the name of the variable created will be greater than the numeric components of all other other automatically named variables.
`add_variable_terminal`	Adds a variable terminal to the grammar with an automatically generated name.
`epsilon`	Returns an empty symbol string.
`has_start_variable`	Returns `true` if the grammar has a start variable, `false` otherwise.

Method	Description
`has_terminal`	Returns `true` if the grammar has a terminal whose value is the same as the value passed in as a parameter. For example, if `AlphaT` = `char` then `has_terminal('a')` will return `true` if `'a'` is a terminal of the grammar.
`is_variable_terminal`	Returns `true` if a value of type `terminal_type` represents a variable terminal and `false` if it represents a normal terminal.
`get_alpha`	Returns the alphabetic value of a value of type `terminal_type`. For example, `get_alpha(get_terminal('a'))` = `'a'`.
`get_name`	Returns the string representation of the name of a variable. For example, `get_name(get_variable("START"))` = `"START"`.
`get_start_variable`	Returns a value of type `variable_type` that corresponds to the start variable of the grammar. If the grammar has no start variable then the behavior is undefined. In debug mode, this behavior is checked using an assertion.
`get_terminal`	Returns a value of type `terminal_type` corresponding to the terminal represented by some alphabet type. If no terminal with the alphabetic value exists then one is created.
`get_variable`	Returns a value of type `variable_type` for some named variable. If no variable with the specified name exists then one is created.
`get_variable_symbol`	Returns a value of type `symbol_type` for some named symbol. If the named symbol exists as a variable then the symbol returned corresponds to a variable. If the named symbol exists as a variable terminal then the symbol returned corresponds to a variable terminal. If the name is not known by the grammar then a named variable terminal is created and returned.
`num_productions`	Returns the number of productions in the grammar.
`num_terminals`	Returns the number of terminals in the grammar. This includes variable terminals.

Method	Description
`num_variables`	Returns the number of variables in the grammar.
`num_variables_capacity`	Returns the total capacity for variables in the grammar. By default, an arbitrary total ordering is imposed on grammar variables. As such, when variables are deleted, they correspond to "holes" in the ordering that can then re-filled when new variables are added. It is often useful to take advantage of the ordered nature and when doing so, knowing the total number of possible variables is imporant.
`num_variable_terminals`	Returns the number of variable terminals.
`remove_production`	Removes a production from the grammar.
`remove_variable`	Removes a variable and its productions from the grammar. This has the effect of removing all productions of the form $V \to \alpha A \beta$ where A is the variable being removed, V is an arbitrary variable, and α and β are arbitrary strings of terminals and non-terminals/variables. If, in the process of removing related productions, all productions on some variable V are removed then the variable V will also be removed.
`search`	Return a value of type `generator_type` that is capable of performing some form of pattern-matching search over the objects of a CFG. See section 8.3.1 for more information on how to use this method.
`set_start_variable`	Change or initialize the start variable of a grammar.
`unsafe_remove_variable`	Removes a variable and its productions from the grammar. This does not, however, go look for related productions.

As well as the above methods, `CFG<AlphaT>` exposes the following types:

`alphabet_type` The underlying type used to represent terminals.

`symbol_type` Represents an arbitrary terminal or non-terminal/variable of the grammar.

`terminal_type` Represents an arbitrary terminal of the grammar.

`variable_type` Represents an arbitrary non-terminal/variable of the grammar.

`symbol_string_type` Represents a string of zero or more symbols (`symbol_type`).

`production_type` Represents an arbitrary production of the grammar.

`production_builder_type` Represents a symbol buffer for use when constructing productions.

`pattern_type` Represents an arbitrary production pattern. See section §8.2 for more information on production patterns.

`generator_type` Represents an arbitrary generator over objects known to the grammar. See section §8.3 for more information on generators.

7.2 NFA<AlphaT>

Method	Description
`add_accept_state`	Takes in a value of type `state_type` and sets the state represented by that value to be an accepting state in the automaton.
`add_state`	Create and return a new state.
`add_start_state`	Takes in a value of type `state_type` and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of $Grail^+$. To actually change the single start state of an automaton, one must use `set_start_state`.
`add_transition`	Add a transition between two states. The transition added is returned as a value of type `transition_type`. Note: duplicate transitions are not added.
`epsilon`	Returns a value of type `symbol_type` that can be used when representing no input on an ϵ-transition.
`get_alpha`	Returns the alphabetic value of some symbol. For example, if `AlphaT` = `char` then `get_alpha(get_symbol('a'))` = `'a'`. Note: `get_alpha(epsilon())` is an illegal operation.
`get_start_state`	Returns the start state of the automaton as a value of type `state_type`.

Method	Description
`add_accept_state`	Takes in a value of type `state_type` and sets the state represented by that value to be an accepting state in the automaton.
`add_state`	Create and return a new state.
`add_start_state`	Takes in a value of type `state_type` and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of *Grail*$^+$. To actually change the single start state of an automaton, one must use `set_start_state`.
`get_symbol`	Returns a symbolic representation of an alphabet symbol. The alphabet symbol is added into the input alphabet of the automaton if it is not yet present.
`is_accept_state`	Returns `true` if a given state is an accepting state of the automaton, and `false` otherwise.
`num_accept_states`	Returns the number of accepting states of the automaton.
`num_states`	Returns the number of states. This includes accepting states.
`num_states_capacity`	Returns the total capacity for states in this automaton. An arbitrary total order is imposed on all states of the automaton. When states are removed[1], this results in "holes" in the ordering that can be filled by new states. It is often convenient to know the maximum possible number of states in the automaton as the total ordering is implemented in terms of the natural numbers.
`num_symbols`	Returns the number of input symbols in this automaton's alphabet. Note: ϵ is considered a meta-symbol, despite having type `symbol_type`, and is not counted in number returned by this method.

[1]Currently, states cannot be removed; however, this function exists in anticipation of the ability to remove states.

Method	Description
`add_accept_state`	Takes in a value of type `state_type` and sets the state represented by that value to be an accepting state in the automaton.
`add_state`	Create and return a new state.
`add_start_state`	Takes in a value of type `state_type` and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of $Grail^+$. To actually change the single start state of an automaton, one must use `set_start_state`.
`num_transitions`	Returns the number of transitions in the automaton.
`remove_accept_state`	Removes a state from the set of accepting states.
`remove_transition`	Removes a transition from the automaton. This does not affect states. As such, it is possible that this operation results in orphaned states.
`search`	Return a value of type `generator_type` that is capable of performing some form of pattern-matching search over the objects of an ϵ-NFA. See section 8.3.2 for more information on this method.
`set_start_state`	Change the current start state. This will not affect, nor will it redirect any ϵ-transitions added in as a result of prior invocations of `add_start_state`.

As well as the above methods, `NFA<AlphaT>` exposes the following types:

`alphabet_type` The underlying type used to represent input symbols.

`state_type` Represents an arbitrary state of an ϵ-NFA.

`symbol_type` Represents an arbitrary input symbol of an ϵ-NFA.

`transition_type` Represents an arbitrary transition in an ϵ-NFA.

`generator_type` Represents a generator over objects known to the `NFA<AlphaT>` class. See section §8.3 for more information on generators.

7.3 PDA<AlphaT>

Method	Description
add_accept_state	Takes in a value of type state_type and sets the state represented by that value to be an accepting state in the automaton.
add_state	Create and return a new state.
add_stack_symbol	Adds an automatically named symbol to the automaton's stack alphabet. The symbol created is returned as a value of type symbol_type.
add_start_state	Takes in a value of type state_type and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of *Grail*$^+$. To actually change the single start state of an automaton, one must use set_start_state.
add_transition	Add a transition between two states. The transition added is returned as a value of type transition_type.
epsilon	Returns a value of type symbol_type that can be used when representing no input on an ϵ-transition.
get_alpha	Returns the alphabetic value of some symbol. For example, if AlphaT = char then get_alpha(get_symbol('a')) = 'a'. Note: get_alpha(epsilon()) is an illegal operation.
get_name	Get the name of a symbol that is primarily in the automaton's stack alphabet. If the symbol is in both the input alphabet and the stack alphabet then it is considered to be primarily in the input alphabet. For example, get_name(get_stack_symbol("FOO")) = "FOO", and get_name(epsilon()) = "epsilon".

Method	Description
`add_accept_state`	Takes in a value of type `state_type` and sets the state represented by that value to be an accepting state in the automaton.
`add_state`	Create and return a new state.
`add_stack_symbol`	Adds an automatically named symbol to the automaton's stack alphabet. The symbol created is returned as a value of type `symbol_type`.
`add_start_state`	Takes in a value of type `state_type` and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of *Grail*$^+$. To actually change the single start state of an automaton, one must use `set_start_state`.
`get_alphabet_symbol`	Get a symbolic representation for a symbol in the automaton's input alphabet. If the symbol is not yet in the automaton's input alphabet then it is added to the automaton's input alphabet.
`get_stack_symbol`	Get a symbolic representation for a named stack symbol. If the named symbol is not yet in the automaton's stack alphabet then it is added to the automaton's stack alphabet.
`get_start_state`	Returns the start state of the automaton as a value of type `state_type`.
`is_accept_state`	Returns `true` if a given state is an accepting state of the automaton, and `false` otherwise.
`is_in_input_alphabet`	Returns `true` is a value of type `symbol_type` is in the automaton's input alphabet.
`num_accept_states`	Returns the number of accepting states of the automaton.
`num_states`	Returns the number of states. This includes accepting states.

Method	Description
`add_accept_state`	Takes in a value of type `state_type` and sets the state represented by that value to be an accepting state in the automaton.
`add_state`	Create and return a new state.
`add_stack_symbol`	Adds an automatically named symbol to the automaton's stack alphabet. The symbol created is returned as a value of type `symbol_type`.
`add_start_state`	Takes in a value of type `state_type` and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of *Grail*$^+$. To actually change the single start state of an automaton, one must use `set_start_state`.
`num_states_capacity`	Returns the total capacity for states in this automaton. An arbitrary total order is imposed on all states of the automaton. When states are removed[2], this results in "holes" in the ordering that can be filled by new states. It is often convenient to know the maximum possible number of states in the automaton as the total ordering is implemented in terms of the natural numbers.
`num_symbols`	Returns the number of input symbols in this automaton's alphabet. Note: ϵ is considered a meta-symbol, despite having type `symbol_type`, and is not counted in number returned by this method.
`num_transitions`	Returns the number of transitions in the automaton.
`remove_accept_state`	Removes a state from the set of accepting states.
`remove_transition`	Removes a transition from the automaton. This does not affect states. As such, it is possible that this operation results in orphaned states.

[2]Currently, states cannot be removed; however, this function exists in anticipation of the ability to remove states.

Method	Description
`add_accept_state`	Takes in a value of type `state_type` and sets the state represented by that value to be an accepting state in the automaton.
`add_state`	Create and return a new state.
`add_stack_symbol`	Adds an automatically named symbol to the automaton's stack alphabet. The symbol created is returned as a value of type `symbol_type`.
`add_start_state`	Takes in a value of type `state_type` and sets the state represented by that value to be a start state in the automaton. By default, there is always one start start. There can only ever be one true start state. Additional start states are added by inserting ϵ-transitions from the current start to those "pretend" start states. Thus, states added as start states by this method are not true start states; however, they behave as if they were. This function embodies the allowance of multiple start states in prior versions of *Grail*$^+$. To actually change the single start state of an automaton, one must use `set_start_state`.
`search`	Return a value of type `generator_type` that is capable of performing some form of pattern-matching search over the objects of an ϵ-NFA. See section 8.3.3 for more information on this method.
`set_start_state`	Change the current start state. This will not affect, nor will it redirect any ϵ-transitions added in as a result of prior invocations of `add_start_state`.

As well as the above methods, `PDA<AlphaT>` exposes the following types:

`alphabet_type` The underlying type used to represent input symbols.

`state_type` Represents an arbitrary state of an ϵ-NPDA.

`symbol_type` Represents an arbitrary input symbol of an ϵ-NPDA.

`symbol_buffer_type` Represents a buffer for stack symbols when creating transitions that push multiple symbols onto the stack.

`transition_type` Represents an arbitrary transition in an ϵ-NPDA.

`generator_type` Represents a generator over objects known to the `PDA<AlphaT>` class. See section §8.3 for more information on generators.

8 Implementation Details

8.1 Reference Counting

All memory and state of the various supported FLOs is managed by the FLTL. One way in which the memory is managed is by reference counting. Reference counting refers to a memory management technique where the number of references to a particular object is always known. When the number of references to an object reaches zero, the object can be safely deallocated. Reference counting can be implemented in one of two ways:

1. To every object o which we want to apply reference counting, we associate another object $Count(o)$ to be o's counter. Using o requires that the programmer operates on o indirectly through the pair $(Address(o), Address(Count(o)))$.

2. To every object o which we want to apply reference counting, we embed a counter object, $o.Count$, within o. Using o requires that the programmer operators on o indirectly through $Address(o)$.

Each approach cannot be universally applied: the former approach is appropriate when one wants to apply reference counting to an arbitrary object of unknown type, the latter is sometimes appropriate when one has total control over the implementation details of each object to which reference counting is applied. Finally, reference counting is usually not appropriate in situations where its possible for counted objects to refer to other counted objects in a cyclic fashion.

The FLTL uses the second method of implementing reference counting for CFG symbol strings, CFG productions, ϵ-NFA transitions, and ϵ-NPDA transitions. Unlike CFG terminals/variables/symbols and ϵ-NFA/ϵ-NPDA symbols/states, productions and transitions are aggregate objects that tie in to their respective FLOs in ways that are not exposed to the casual programmer.

For example, a production is implemented by the template class `fltl::cfg::Production<AlphaT>`. This class contains a counter member variable. The programmer doesn't have direct access to this class, instead they must go through `fltl::cfg::OpaqueProduction<AlphaT>`, which resolves to `fltl::CFG<AlphaT>::production_type`. Operating on a value of type `production_type` indirectly affects the reference counter of the addressed `Production<AlphaT>` type.

This indirection is necessary to allow one to do such things as operate on the values of a production, even if the production has been removed from its grammar. At first, this seems absurd: why would one ever want to operate on a deleted production? Often, an algorithm that operates on a CFG will remove productions from some grammar, but also add in other productions that are in some way related to the now removed productions. Having access to deleted productions allows us to access the related information in a safe way.

Reference counting is also appropritate for more subtle reasons, as described in section §8.3.

8.2 CFG Production Patterns

CFG production patterns are easily the most expressive tool in the FLTL. A single production pattern is able to do the following:

- Determine whether or not a production has a specific form. Production patterns can contain bound and unbound elements. Restrictions on both are imposed by the types of the objects used when constructing a pattern. When a part of a pattern is unbound, it represents a variable that can take on a value contained in a production. When a part of a pattern is bound, it represents a constraint on the structure of the production that must be satisfied. Unbound pattern parts are prefixed by `C++`'s `~` operator.

- Extract (also known as destructure) and bind the unbound parts of a production to local memory, usually named by stack-allocated (local) variables.

Production patterns are an embedded domain-specific language (EDSL) within `C++`, where "words" of the pattern language are instantiated into `C++` by using specific combinations overloaded operators. The following table gives several short examples of using patterns in-place. In the follwoing examples, `cfg` is a value of type `CFG<const char *>` and `P` is a value of type `production_type`.

`C++` **Pattern**	**Meaning**

`terminal_type a(cfg.get_terminal("a"));`
`if((cfg._ --->* cfg.__ + a + cfg.__).match(P)) {`
`  /* ... */`
`}`
	Match a production P iff it contains the terminal a in its right-hand side.

`terminal_type a(cfg.get_terminal("a"));`
`variable_type S(cfg.get_start_variable());`
`if((S --->* cfg.__ + a + cfg.__).match(P)) {`
`  /* ... */`
`}`
	Match a production P of the form $S \to \alpha a \beta$ where S is the start variable and α and β are arbitrary strings of symbols.

`terminal_type a(cfg.get_terminal("a"));`
`variable_type V;`
`symbol_string_type alpha;`
`symbol_string_type beta;`
`if(((~V) --->* ~alpha + a + ~beta).match(P)) {`
`  /* ... */`
`}`
	Match a production P of the form $V \to \alpha a \beta$ where V is an arbitrary variable and α and β are arbitrary strings of symbols.
	If P is matched then bind the values of V to `V`, α to `alpha`, and β to `beta`.

`terminal_type a(cfg.get_terminal("a"));`
`unsigned dot;`
`/* ... */`
`if((cfg._ --->* cfg.__(dot) + a + cfg.__).match(P)) {`
`  /* ... */`
`}`
	Match a production P of the form $V \to \alpha \bullet a \beta$ where V is an arbitrary variable, α and β are arbitrary symbol strings, and the value of `dot` *at the time of matching the pattern* determines the offset of the $\bullet$, and hence the length of α.

The CFG of Figure 8.2.1 describes the CFG production pattern EDSL. In the grammar, `cfg` represents an instance of `CFG<AlphaT>` for some type `AlphaT`, and $o_{\texttt{T}}$ is a terminal representing an arbitrary object o of type `T`.

Figure 8.2.1: Grammar for CFG Production Patterns

$$
\begin{aligned}
Pattern &\rightarrow LHS\ \texttt{--->*}\ RHS_0 \\
LHS &\rightarrow \texttt{cfg.\_} \\
&\rightarrow o_{\texttt{const variable\_type}} \\
&\rightarrow \texttt{(}\ \texttt{\~{}}\ o_{\texttt{variable\_type}}\ \texttt{)} \\
UnitOfKnownLength &\rightarrow o_{\texttt{const variable\_type}} \\
&\rightarrow o_{\texttt{const terminal\_type}} \\
&\rightarrow o_{\texttt{const symbol\_type}} \\
&\rightarrow o_{\texttt{const symbol\_string\_type}} \\
&\rightarrow \texttt{cfg.\_} \\
&\rightarrow \texttt{cfg.\_\_(}\ o_{\texttt{unsigned}}\ \texttt{)} \\
&\rightarrow \texttt{\~{}}\ o_{\texttt{variable\_type}} \\
&\rightarrow \texttt{\~{}}\ o_{\texttt{terminal\_type}} \\
&\rightarrow \texttt{\~{}}\ o_{\texttt{symbol\_type}} \\
UnitOfUnknownLength &\rightarrow \texttt{\~{}}\ o_{\texttt{symbol\_string\_type}} \\
&\rightarrow \texttt{cfg.\_\_} \\
RHS_0 &\rightarrow UnitOfKnownLength\ RHS_1 \\
&\rightarrow UnitOfUnknownLength\ RHS_2 \\
RHS_1 &\rightarrow \texttt{+}\ RHS_0 \\
&\rightarrow \epsilon \\
RHS_2 &\rightarrow \texttt{+}\ UnitOfKnownLength\ RHS_1 \\
&\rightarrow \epsilon
\end{aligned}
$$

The terminals of the above grammar have the following meanings in the context of a CFG production pattern:

Terminal Sequence	**Description**
`cfg._`	This represents an arbitrary symbol. A symbol is either a variable/non-terminal or a terminal of the grammar on which we are operating. This pattern component will successfully match against any symbol, but will not bind the value of the symbol to anything.
(`~`$o_{\texttt{variable\_type}}$)	Similar to `~`$o_{\texttt{T}}$ (see below), this represents an unbound variable object. However, this unbound variable object (in the above grammar) is on the left-hand side of `--->*`, which is a composition of the `--` post-decrement and `->*` member-pointed-to-by-object-pointed-to operators, and as such requires additional parentheses in order to get the right behavior given the operator precedence rules of `C++`.

Terminal Sequence	**Description**
`cfg.__(`$o_{\texttt{unsigned}}$`)`	This represents an arbitrary symbol string of length exactly o, where o is an object of type `unsigned`. Like `cfg.__`, the substring matched is never bound to any variable. Note: the value of o is not passed by value. Instead, o is passed by reference and a pointer to o is stored. As such, the value of o is resolved each time the pattern is invoked. That is, as o changes, so does the matching behavior of the pattern.
`~`$o_{\texttt{T}}$	If the object o has type `T` $\in$ {`symbol_type`, `variable_type`, `terminal_type`, `symbol_string_type`}, where `T` is non-`const`, then `~`$o_{\texttt{T}}$ represents an unbound object. Usually, a named memory location (`C++` local variable) is used in place of o; however, an unnamed object of non-`const`-qualified type is also satisfactory. If $P \rightarrow \alpha a \beta$ is a production and a pattern is attempting to match the remaining $a\beta$ of the right-hand side of the pattern against `~`$o_{\texttt{T}}$`+`$\ldots$ then the pattern will attempt to match β if: • `T` is `symbol_type` • `T` is `variable_type` and a is a variable/non-terminal. • `T` is `terminal_type` and a is a terminal. If a is successfully matched then $o \leftarrow a$. If `T` is `symbol_string_type` then all possible lengths of symbol strings for o will be tried. For this reason, one cannot put two unbound symbol strings as adjacent pattern parts as the first one will always match the empty string. This constraint is communicated by means of RHS_0, RHS_1, and RHS_2, which can be interpreted as different states of the right-hand side of a production pattern. Similar to `cfg.__(`$o_{\texttt{unsigned}}$`)`, a pointer to o is stored in order to perform this binding.
`cfg.__`	This represents an arbitrary symbol string of any length. This pattern component will always match, even if one is attempting to match an empty symbol string, and will never bind the matched string to anything.

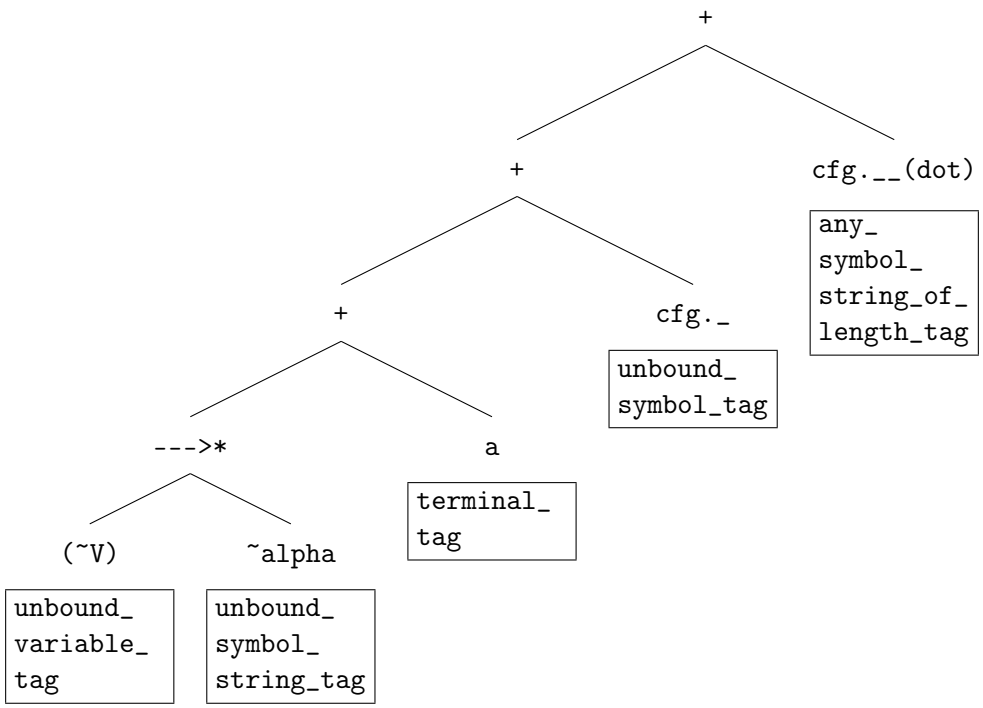
Terminal Sequence	Description
$\ldots +_{o_T} \ldots$	The implied meaning of the overloaded infix `+` operator represents pattern part concatenation. The actual meaning is pattern extension, as no real concatenation is being done behind the scenes. The `+` operator can only be used on the right-hand side of `--->*`. Outside of patterns, the `+` operator has the explicit meaning of concatenation. Thus, for the pattern `A--->* a + b + c + d`, the subtle change of `A--->* a + (b + c) + d` represents a different pattern that behaves differently and is potentially unsafe.

Figure 8.2.2: Example Production Pattern

```
terminal_type a(cfg.get_terminal("a"));
variable_type V;
symbol_string_type alpha;
unsigned dot;

/* ...*/

pattern_type p((~V) --->* ~alpha + a + cfg._ + cfg.__(dot));

/* ... */
```

At compile time, the type of a production pattern contains a left-leaning tree of type tags. For example, the type tree associated with the pattern `p` in Figure 8.2.2 is shown in Figure 8.2.3.

Figure 8.2.3: Type Tag Tree of Figure 8.2.2



The type tree of Figure 8.2.3 stores type tags instead of the actual types of the pattern parts as the types of the pattern parts can be deduced from the tags and because it simplifies the compilation process.

Each pattern is given a pointer to a static method of a template class. The template class parameterizes the type tree. As a result of template instantiation, the compiler generates a static method for each leaf in the type tree. Taken together, the invocations of these static methods can be inlined into a single pattern matching and variable binding function.

In addition, each pattern also has a pointer to the first element in a 16 cell[1] pointer array. Each cell of the array has type `void *` (pointer to `void`). The following table summarizes the deduced[2] types of the type tags:

Type Tag	**Deduced Type**	**Example Pattern Part**
`symbol_tag`	`symbol_type *`	`s`
`terminal_tag`	`terminal_type *`	`t`
`variable_tag`	`variable_type *`	`v`
`symbol_string_tag`	`symbol_string_type *`	`str`
`any_symbol_tag`		`cfg._`
`any_symbol_string_tag`		`cfg.__`
`any_symbol_string_of_length_tag`	`unsigned *`	`cfg.__(dot)`
`unbound_symbol_tag`	`symbol_type *`	`~s`
`unbound_terminal_tag`	`terminal_type *`	`~t`
`unbound_variable_tag`	`variable_type *`	`~v`
`unbound_symbol_string_tag`	`symbol_string_type *`	`~str`

Notice that the deduced types are all pointer types, as is consistent with the pattern parts being represented by an array of pointers. This detail is responsible for the flexibility of all FLTL patterns, including CFG production patterns: patterns don't store values representing pattern parts, they store memory addresses, and so when the values located at those memory addresses change, so does the behavior of the patterns. Examples of how changing the behavior of a pattern are shown in Appendix: Converting an ϵ-NPDA to a CFG.

The inputs to a pattern matching function are: a pointer to a cell in the aforementioned pointer array, a pointer to a symbol in a production's right-hand side, and the number of symbols in the production's right-hand side that have yet to be matched. With this information, and the information that can be inferred from the type tree, the pattern matching functions are able to check equality of symbols, bind

[1] This is an artificial limit imposed on patterns. That is, the maximum number of pattern parts is 16.

[2] The types of the array cells are `void *`; however, the values of those cells are pointers to heterogeneous objects. As such, the types of those objects must be known when the untyped pointers are used. As mentioned, each leaf in the type tree expands out to a pattern matching function. The type of the leaf, i.e. some type tag, is one of the formal parameters to a pattern-matching class template and so the type is known to the pattern-matching function of that template class. Pattern matching functions perform type casts from `void *` pointers to pointers of the types deduced by the type tags.

arbitrary values to the locations in memory addressed by unbound pattern parts[3], and prove that a pattern cannot match a production by knowing the minimum number[4] of symbols that a pattern must match.

8.3 Generators

Generators are the iteration mechanism of the FLTL. A generator is akin to a "for each ... in ..." loop in a high-level language such as `Python`. Generators operate using the same overloaded operators as patterns (see section §8.2). Unlike `C++` standard library iterators, generators are not cursors into some collection, although they do maintain a cursor. The following tables illustrate example generators for each FLO supported by the FLTL.

8.3.1 CFG Generators

Suppose the following prelude is given:

```
CFG<AlphaT> cfg;
/* ... */
terminal_type t;
variable_type v;
production_type prod;
pattern_type pattern( /* ... */ );
```

Further, suppose `cfg` is a representation of some CFG $G = (V, \Sigma, \rightarrow, S)$. Then the following functions return a value of type `generator_type` with the following meanings:

Usage	**Description**
`cfg.search(~t)`	Returns a generator that matches all terminals of a grammar. Each match of the generator changes the value of `t`. The order in which terminals are visited is the order in which they are added to the grammar. It is safe to add new terminals to the grammar while the generator is being used. Formally, this generator represents $\forall t \in \Sigma$.

[3] Those pattern parts prefixed by the `~` operator.

[4] This information can be roughly inferred from the distinction between *UnitOfKnownLength* and *UnitOfUnknownLength* in Figure 8.2.1. A lower bound on the minimum length can be computed at compile time from the type tree. A sharper lower bound is known at runtime by taking advantage of the lower bound computed at compile time and by resolving the lengths of bound symbol strings and unbound arbitrary symbol strings of a known length.

Usage	Description
`cfg.search(~v)`	Returns a generator that matches all variables of a grammar. Each match of the generator changes the value of `v`. It is safe to add and remove variables from the grammar while the generator is being used—even if the variable being removed is the most recently matched variable. Note: there are no guarantees that variables added while the generator is matching will ever be seen by the generator. In order to guarantee that all new variables are matched, the generator should be rewound by calling its `rewind` method. Formally, this generator represents $\forall v \in V$.
`cfg.search(~prod)`	Returns a generator that matches all productions of a grammar. Each match of the generator changes the value of `prod`. It is safe to add and remove productions from the grammar while the generator is being used—even if the production being removed is the most recently matched production. Note: there are no guarantees that productions added while the generator is matching will ever be seen by the generator. In order to guarantee that all new productions are matched, the generator should be rewound by calling its `rewind` method. Formally, this generator represents $\forall prod \in \rightarrow$.
`cfg.search(pattern)`	Returns a generator that matches and destructures all productions that can be matched by `pattern`. This generator makes the same guarantees as the production generator. Formally, this generator represents $\forall prod \in \rightarrow$ where *prod* has the form matched by `pattern`.
`cfg.search(~prod, pattern)`	Returns a generator that matches and destructures all productions that can be matched by `pattern`. This generator also binds every matched production to `prod`. This generator makes the same guarantees as the production generator. Formally, this generator represents $\forall prod \in \rightarrow$ where *prod* has the form matched by `pattern`.

Terminal generators are implemented in a straightforward way: a cursor simply moves through the list of terminals. This type of generator is simple because terminals, once added to a grammar, cannot be removed.

Variable generators are slightly more interesting. Instantiation of a new generator does not actually assign the constructed generator a "beginning" variable. This is because between the time that a generator is created and the time that a generator is used, any number of modifications can be made to the CFG, possibly leading to "holes" in the list of variables. Luckily, an instance of `CFG<AlphaT>` maintains information about the "beginning" variable for use by generators, among other things. As such, when one first attempts to use the generator (by calling its `match_next` method), the beginning variable is taken from the associated CFG.

Production generators, and by extension pattern generators, are more complex as they interact with the reference counting mechanisms of productions. Like variable generators, production generators are not initialized with a beginning production, and instead take it from the associated CFG when it is needed.

The productions of a CFG are represented by an adjacency list, i.e. each variable has a linked list of productions. Because productions are reference counted, it is possible for deleted productions to be present in the list of a variable's productions. As such, production generators must be able to skip over deleted productions.

The beginning production being automatically calculated when the generator is first used implies that each call to a generator's `match_next` method will check to see if the production currently pointed to is valid, and if not, will go look for a valid production. Thus, when `match_next` finds a valid production for the current iteration, it actually advances the generator's cursor to the next candidate production for the next iteration. However, this requires that the generator hold a reference to this production lest the production's reference counter reach zero and yield undefined behavior for the generator.

This coupling between production/pattern generators and productions is responsible for allowing flexible usage of productions.

8.3.2 ϵ-NFA Generators

Suppose the following prelude is given:

```
NFA<AlphaT> nfa;
state_type source;
state_type sink;
symbol_type read;
transition_type trans;
```

Further, suppose that `nfa` is a representation of some ϵ-NFA $N = (Q, \Sigma, \delta, q_0 \in Q, F \subseteq Q)$. Then the following functions return a value of type `generator_type` with the following meanings:

Usage	**Description**

Usage	Description
`nfa.search(~source)`	Returns a generator that matches all states of an automaton. Each match of the generator changes the value of `source`. The order in which states are visited is the order in which they were added to the automaton. It is safe to add new states to the automaton while the generator is being used. Formally, this generator represents $\forall source \in Q$.
`nfa.search(~read)`	Returns a generator that matches all symbols of an automaton. Each match of the generator changes the value of `read`. It is safe to add alphabet symbols from the automaton while the generator is being used. Note: there are no guarantees that symbols added while the generator is matching will ever be seen by the generator. In order to guarantee that all new symbols are matched, the generator should be rewound by calling its `rewind` method. Formally, this generator represents $\forall read \in \Sigma$.
`nfa.search(~trans)`	Returns a generator that matches all transitions of an automaton. Each match of the generator changes the value of `trans`. It is safe to add and remove transitions from the automaton while the generator is being used—even if the transition being removed is the most recently matched transition. Note: there are no guarantees that transitions added while the generator is matching will ever be seen by the generator. In order to guarantee that all new transitions are matched, the generator should be rewound by calling its `rewind` method. Formally, this generator represents $\forall trans = (source, read, sink)$ such that $source, sink \in Q$, $read \in \Sigma \cup \{\epsilon\}$, and $sink \in \delta(source, read)$.

Usage	Description

`nfa.search(`
`  ~source,`
`  ~read,`
`  ~sink`
`)`
	Returns a generator that matches and destructures all transitions. This generator makes the same guarantees as the transition generator.
	
	Each match of this generator changes the values of `source`, `read`, and `sink`. If, however, any of these variables are not prefixed by the `~` operator then they represent constraints on those transitions matched.
	
	Formally, this generator represents $\forall(source, read, sink)$ such that $source, sink \in Q$, $read \in \Sigma \cup \{\epsilon\}$, and $sink \in \delta(source, read)$.

`nfa.search(`
`  ~trans,`
`  ~read,`
`  ~symbol,`
`  ~sink`
`)`
	This is the same form of generator as the destructuring transition generator; however, this generator will also bind any matched transitions to `trans`. When using this form of generator, the `~` prefix operator on the transition variable is required.

The implementation of ϵ-NFA generators uses similar techniques to those involved in CFG generators.

8.3.3 ϵ-NPDA Generators

Suppose the following prelude is given:

```
PDA<AlphaT> pda;
state_type source;
state_type sink;
symbol_type read;
symbol_type pop;
symbol_type push;
transition_type trans;
```

Further, suppose that `nfa` is a representation of some ϵ-NPDA $N = (Q, \Sigma, \Gamma, \delta, q_0 \in Q, F \subseteq Q)$. Then the following functions return a value of type `generator_type` with the following meanings:

Usage	**Description**
`pda.search(~source)`	Returns a generator that matches all states of an automaton. Each match of the generator changes the value of `source`. The order in which states are visited is the order in which they were added to the automaton. It is safe to add new states to the automaton while the generator is being used. Formally, this generator represents $\forall source \in Q$.
`pda.search(~read)`	Returns a generator that matches all symbols of an automaton. This includes input and stack symbols. Each match of the generator changes the value of `read`. It is safe to add alphabet and stack symbols to the automaton while the generator is being used. Note: there are no guarantees that symbols added while the generator is matching will ever be seen by the generator. In order to guarantee that all new symbols are matched, the generator should be rewound by calling its `rewind` method. Formally, this generator represents $\forall read \in \Sigma \cup \Gamma$.

Usage	Description
`pda.search(~trans)`	Returns a generator that matches all transitions of an automaton. Each match of the generator changes the value of `trans`. It is safe to add and remove transitions from the automaton while the generator is being used—even if the transition being removed is the most recently matched transition. Note: there are no guarantees that transitions added while the generator is matching will ever be seen by the generator. In order to guarantee that all new transitions are matched, the generator should be rewound by calling its `rewind` method. Formally, this generator represents $\forall trans = (source, read, pop, push, sink)$ such that $source, sink \in Q$, $read \in \Sigma \cup \{\epsilon\}$, $pop, push \in \Sigma \cup \Gamma \cup \{\epsilon\}$, and $(sink, push) \in \delta(source, read, pop)$.

`pda.search(`
`  ~source,`
`  ~read,`
`  ~pop,`
`  ~push,`
`  ~sink`
`)`
	Returns a generator that matches and destructures all transitions. This generator makes the same guarantees as the transition generator.
	
	Each match of this generator changes the values of `source`, `read`, `pop`, `push`, and `sink`. If, however, any of these variables are not prefixed by the `~` operator then they represent constraints on the transitions matched.
	
	Formally, this generator represents $\forall(source, read, pop, push, sink)$ such that $source, sink \in Q$, $read \in \Sigma \cup \{\epsilon\}$, $pop, push \in \Sigma \cup \Gamma \cup \{\epsilon\}$, and $(sink, push) \in \delta(source, read, pop)$.

`pda.search(`
`  ~trans,`
`  ~source,`
`  ~read,`
`  ~pop,`
`  ~push,`
`  ~sink`
`)`
	This is the same form of generator as the destructuring transition generator; however, this generator will also bind any matched transitions to `trans`. When using this form of generator, the `~` prefix operator on the transition variable is required.

The implementation of ϵ-NPDA generators uses similar techniques to those involved in CFG generators.

8.4 Deviations from Design Report

Some implementation details have deviated significantly from the design report. The following list summarizes implementation deviations from the design report:

- Naming scheme of methods. Methods names are lower case, and the individual words of the method names are separated by underscores. For example, `remove_production`, as opposed to the proposed `removeProduction`. This naming scheme is consistent with the `C++` standard library.

- Orderings of symbols, alphabets, variables. The design proposal stated that the various sub-objects such as terminals, non-terminals, states, and symbols would be unordered. However, the currently implementation imposes a total ordering on many of these objects. This change allows the aforementioned objects to be used in ordered containers such as `std::set`, `std::map`, *et al.*

- Production patterns are implemented in a completely different way. For example, the design report proposed that production patterns can contain Boolean conditions. This capability was rejected as impractical early on in the implementation. Further, the design report proposed that production patterns be implemented by specializing a data-structure independent querying language implemented through meta-programming. The implementations of patterns currently makes heavy use of `C++` template meta-programming; however, each pattern type (e.g. transition pattern, production pattern) must be completely specified and implemented within the scope of its respective FLO. A general approach was first attempted; however, it proved to be overly complex and poorly specified. For example, there was no clear protocol for how a pattern was to destructure some object according to a pattern.

- The design report proposed the use of regular expressions for some tools. The original project proposal required only some representation of regular languages. ϵ-NFAs were chosen in favor of regular expressions as it was not clear how to elegantly represent regular expressions whose alphabet symbols can be objects of complex types.

9 Testing

Two forms of testing were performed:

1. A simple test suite was developed for automating the running of functions that test the behavior of the various FLOs of the FLTL. The purpose of this test suite is to ensure that FLTL behaves as expected. This is the most important set of tests as all of the algorithms are built on top of the FLTL.

   The test suite includes hundreds of tests grouped into behavioral categories.

2. The second set of tests involved creating or using pre-existing descriptions of FLOs and then transforming them with the *Grail*$^+$ tools. Transformations on simple grammars were checked manually. Transformations on larger grammars were tested in terms of other tools. These types of tests provide anecdotal evidence of the correctness of the algorithms.

   For example, to test the correctness of the `cfg-to-cnf` too, a grammar is converted to CNF by means of the `cfg-to-cnf` tool and then words are checked for membership in the language of the transformed grammar using the `cfg-parse` tool. Sequences of these sorts of tests were performed. For example, `cfg-to-pda`, `pda-to-cfg`, followed by a series of tests with `cfg-parse`.

   Extra effort was put into ensuring the correctness of `cfg-parse`, as it is used in the testing of other tools.

The *Grail*$^+$ tools were tested on a number of archictures with varying compilers. The most recent version of *Grail*$^+$ compiles with:

- Microsoft Visual Studio 2010's `C++` compiler, Visual `C++`.
- G++, the GNU `C++` compiler.
- Clang++, the `C++` compiler built on top of the low level virtual machine (LLVM).
- ICPC, the Intel `C++` compiler.

10 Results

Together, *Grail*$^+$ and the FLTL provide formal language researchers with powerful tools for performing symbolic manipulations from their command lines and for creating programs that symbolically manipulate FLOs. The FLTL mixes declarative and imperative programming styles so as to achieve the most straightforward implementation of an algorithm from a specification. Further, the FLTL manages all of its own memory, releiving the programmer of this burden. Finally, by virtue of being implemented in `C++`, advanced programmers who are more familiar with `C++` will have access to the full breadth of all available libraries implemented for the `C` and `C++` programming languages[1].

The FLTL represents a new direction for *Grail*$^+$ and for programmers, researchers, and students who want to manipulate FLOs. The FLTL deals with the details so that researchers can focus on their algorithms in the language of their algorithms rather than in the language of the internal details of some FLO implementations.

[1]For example, if one were to make a graphical user interface (GUI) for *Grail*$^+$ then said person could use a cross-platform GUI framework such as *Qt* for managing the interface and *GraphViz* for generating graphical representations of the various FLOs.

Bibliography

[1] John Aycock and R. Nigel Horspool. Practical earley parsing. *Comput. J.*, 45(6):620–630, 2002.

[2] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. O'Reilly, Beijing, 2009.

[3] Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13:94–102, February 1970.

[4] Sheila A. Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12:42–52, January 1965.

[5] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Pearson Addison-Wesley, 3. edition, 2007.

[6] Scott E. Hudson. Cup lalr parser generator in java, March 2011.

[7] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1975.

[8] Robert C. Moore. Removing left recursion from context-free grammars. In *Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference*, pages 249–255, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[9] Julian Seward. Valgrind home, March 2011.

[10] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.

Appendix: Converting an ϵ-NPDA to a CFG

This section provides an example of how ϵ-NPDA pattern-matching generators are used in one step of the process of converting an ϵ-NPDA into a CFG. This algorithm is described on page 120 of [10].

Let `pda` be a FLTL representation for an ϵ-NPDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ where every transition of `pda` either pushes a symbol onto the stack, pops a symbol off of the stack, or does not alter the stack. That is, no transition simultaneously pushes a symbol onto and pops a symbol off of the stack. Let `cfg` be a FLTL representation for the CFG that we are constructing from `pda`.

We are interested in sequences of transitions that begin by pushing a symbol t onto the stack and that end by popping t off of the stack.

For each $p, q, r, s \in Q$, $a, b \in \Sigma \cup \{\epsilon\}$, $t \in \Gamma$, if $(r, t) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(s, b, t)$ then we would like to add the production $A_{pq} \rightarrow a\ A_{rs}\ b$ to the grammar that we are constructing. Here, the variable A_{ij} for $i, j \in Q$ represents all possible paths of computation that bring the ϵ-NPDA from state i to state j on an empty stack.

The following two pattern generator intializations represent the transition forms that we are searching for:

```
// states
state_type p;
state_type q;
state_type r;
state_type s;

// symbols
symbol_type a;
symbol_type b;
symbol_type t;

// if there is a transition from p to r that brings the PDA from
// an empty stack to a stack with t on it and reads a...
generator_type p_to_r_push(pda.search(
    ~p,              // source state
    ~a,              // read symbol
    pda.epsilon(),   // pop symbol
    ~t,              // push symbol
    ~r               // sink state
));

// ... and if there is a transition from s to q that pops t off
// of the stack when reading b
generator_type s_to_q_pop(pda.search(
    ~s,
    ~b,
    t,
    pda.epsilon(),
    ~q
));
```

First, notice that both patterns can be defined in the same scope, even though there is a clear dependency in that the t of $\delta(s, b, t)$ depends on the assignment of t in $(r, t) \in \delta(p, a, \epsilon)$. This dependency is not

important when we define our generators, it is only important when we use them. This is because the transition patterns store pointers to `p`, `a`, `t`, `r`, `s`, `b`, and `q`. Thus, when `t` is given as a parameter to the last invocation of `pda.search`, it is not passed by value but instead by reference, and the address of `t` is then known to the pattern.

The following code demonstrates how to use the above patterns. Below, `translate` represents a function that takes an ϵ-NPDA alphabet symbol and converts it into an equivalent string of CFG terminals. We must convert to a string of terminals because ϵ is a symbol in an ϵ-NPDA but a string in a CFG. Also, `A(p,q)` for two states `p` and `q` represents the CFG variable A_{pq}.

```
// add A_pq -> a A_rs b
for(; p_to_r_push.match_next(); ) {

    // t is bound by p_to_r_push, s_to_q_pop sees this update

    for(s_to_q_pop.rewind();
        s_to_q_pop.match_next();) {

        cfg.add_production(
            A(p, q),
            translate(cfg, a) + A(r, s) + translate(cfg, b)
        );
    }
}
```

The outer loop is responsible for finding all transitions $(r, t) \in \delta(p, a, \epsilon)$, where `pda.epsilon()` is the only constraint of the pattern. Each time such a transition is matched, `r` $\leftarrow r$, `t` $\leftarrow t$, `p` $\leftarrow p$, and `a` $\leftarrow a$.

The inner loop begins by rewinding the transition pattern generator. This must be done otherwise the inner loop will only iterate over anything on the first iteration of the outer loop. At the point of executing the inner loop, `t` has been bound by the destructuring and binding behavior of the pattern generator of the outer loop.

The inner loop behaves in a similar way to the outer loop with the exception that the pattern generator of the inner loop has two constraints: `t` as the symbol to be popped off the stack and `pda.epsilon()`, the symbol to be pushed onto the stack.

When these constraints are all satisfied, a production is added by translating information from the ϵ-NPDA over to the CFG.