Ragel State Machine Compiler

User Guide

by

Adrian Thurston

License

Ragel version 5.25, December 2007
Copyright © 2003-2007 Adrian Thurston

This document is part of Ragel, and as such, this document is released under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Ragel is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Ragel; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Contents

1 Introduction **1**
- 1.1 Abstract . . . 1
- 1.2 Motivation . . . 1
- 1.3 Overview . . . 2
- 1.4 Related Work . . . 4
- 1.5 Development Status . . . 5

2 Constructing State Machines **6**
- 2.1 Ragel State Machine Specifications . . . 6
  - 2.1.1 Naming Ragel Blocks . . . 7
  - 2.1.2 Machine Definition . . . 7
  - 2.1.3 Machine Instantiation . . . 7
  - 2.1.4 Including Ragel Code . . . 7
  - 2.1.5 Importing Definitions . . . 7
- 2.2 Lexical Analysis of a Ragel Block . . . 8
- 2.3 Basic Machines . . . 8
- 2.4 Operator Precedence . . . 11
- 2.5 Regular Language Operators . . . 11
  - 2.5.1 Union . . . 12
  - 2.5.2 Intersection . . . 13
  - 2.5.3 Difference . . . 13
  - 2.5.4 Strong Difference . . . 14
  - 2.5.5 Concatenation . . . 14
  - 2.5.6 Kleene Star . . . 15
  - 2.5.7 One Or More Repetition . . . 16
  - 2.5.8 Optional . . . 16
  - 2.5.9 Repetition . . . 17
  - 2.5.10 Negation . . . 17
  - 2.5.11 Character-Level Negation . . . 17
- 2.6 State Machine Minimization . . . 17
- 2.7 Visualization . . . 17

3 User Actions **19**
- 3.1 Embedding Actions . . . 19
  - 3.1.1 Starting Action . . . 19
  - 3.1.2 Finishing Action . . . 20
  - 3.1.3 All Transition Action . . . 20

3.1.4 Pending Out (Leaving) Actions . . . 21
3.2 State Action Embedding Operators . . . 22
3.2.1 To-State and From-State Actions . . . 22
3.2.2 EOF Actions . . . 23
3.2.3 Handling Errors . . . 23
3.3 Action Ordering and Duplicates . . . 25
3.4 Values and Statements Available in Code Blocks . . . 25

4 Controlling Nondeterminism 28
4.1 Priorities . . . 30
4.2 Guarded Operators that Encapsulate Priorities . . . 31
4.2.1 Entry-Guarded Concatenation . . . 32
4.2.2 Finish-Guarded Concatenation . . . 32
4.2.3 Left-Guarded Concatenation . . . 32
4.2.4 Longest-Match Kleene Star . . . 33

5 Interface to Host Program 35
5.1 Variables Used by Ragel . . . 35
5.2 Alphtype Statement . . . 37
5.3 Getkey Statement . . . 37
5.4 Access Statement . . . 37
5.5 Variable Statement . . . 37
5.6 Pre-Push Statement . . . 37
5.7 Post-Pop Statement . . . 38
5.8 Write Statement . . . 38
5.8.1 Write Data . . . 38
5.8.2 Write Init . . . 39
5.8.3 Write Exec . . . 39
5.8.4 Write EOF Actions . . . 39
5.8.5 Write Exports . . . 39
5.9 Maintaining Pointers to Input Data . . . 40
5.10 Running the Executables . . . 41
5.11 Choosing a Generated Code Style (C/D only) . . . 42

6 Beyond the Basic Model 44
6.1 Parser Modularization . . . 44
6.2 Referencing Names . . . 45
6.3 Scanners . . . 45
6.4 State Charts . . . 48
6.4.1 Join . . . 49
6.4.2 Label . . . 49
6.4.3 Epsilon . . . 49
6.4.4 Simplifying State Charts . . . 49
6.4.5 Dropping Down One Level of Abstraction . . . 50
6.5 Semantic Conditions . . . 51
6.6 Implementing Lookahead . . . 52
6.7 Parsing Recursive Language Structures . . . 52

Chapter 1

Introduction

1.1 Abstract

Regular expressions are used heavily in practice for the purpose of specifying parsers. However, they are normally used as black boxes linked together with program logic. User actions are executed in between invocations of the regular expression engine. Adding actions before a pattern terminates requires patterns to be broken and pasted back together with program logic. The more user actions are needed, the less the advantages of regular expressions are seen.

Ragel is a software development tool that allows user actions to be embedded into the transitions of a regular expression's corresponding state machine, eliminating the need to switch from the regular expression engine and user code execution environment and back again. As a result, expressions can be maximally continuous. One is free to specify an entire parser using a single regular expression. The single-expression model affords concise and elegant descriptions of languages and the generation of very simple, fast and robust code. Ragel compiles finite state machines from a high level regular language notation to executable C, C++, Objective-C, D, Java or Ruby.

In addition to building state machines from regular expressions, Ragel allows the programmer to directly specify state machines with state charts. These two notations may be freely combined. There are also facilities for controlling nondeterminism in the resulting machines and building scanners using patterns that themselves have embedded actions. Ragel can produce code that is small and runs very fast. Ragel can handle integer-sized alphabets and can compile very large state machines.

1.2 Motivation

When a programmer is faced with the task of producing a parser for a context-free language there are many tools to choose from. It is quite common to generate useful and efficient parsers for programming languages from a formal grammar. It is also quite common for programmers to avoid such tools when making parsers for simple computer languages, such as file formats and communication protocols. Such languages often meet the criteria for the regular languages. Tools for processing the context-free languages are viewed as too heavyweight for the purpose of parsing regular languages because the extra run-time effort required for supporting the recursive nature of context-free languages is wasted.

When we turn to the regular expression-based parsing tools, such as Lex, Re2C, and scripting languages such as Sed, Awk and Perl we find that they are split into two levels: a regular expression matching engine and some kind of program logic for linking patterns together. For example, a Lex

program is composed of sets of regular expressions. The implied program logic repeatedly attempts to match a pattern in the current set, then executes the associated user code. It requires the user to consider a language as a sequence of independent tokens. Scripting languages and regular expression libraries allow one to link patterns together using arbitrary program code. This is very flexible and powerful, however we can be more concise and clear if we avoid gluing together regular expressions with if statements and while loops.

This model of execution, where the runtime alternates between regular expression matching and user code exectution places severe restrictions on when action code may be executed. Since action code can only be associated with complete patterns, any action code that must be executed before an entire pattern is matched requires that the pattern be broken into smaller units. Instead of being forced to disrupt the regular expression syntax and write smaller expressions, it is desirable to retain a single expression and embed code for performing actions directly into the transitions that move over the characters. After all, capable programmers are astutely aware of the machinery underlying their programs, so why not provide them with access to that machinery? To achieve this we require an action execution model for associating code with the sub-expressions of a regular expression in a way that does not disrupt its syntax.

The primary goal of Ragel is to provide developers with an ability to embed actions into the transitions and states of a regular expression's state machine in support of the definition of entire parsers or large sections of parsers using a single regular expression. From the regular expression we gain a clear and concise statement of our language. From the state machine we obtain a very fast and robust executable that lends itself to many kinds of analysis and visualization.

1.3 Overview

Ragel is a language for specifying state machines. The Ragel program is a compiler that assembles a state machine definition to executable code. Ragel is based on the principle that any regular language can be converted to a deterministic finite state automaton. Since every regular language has a state machine representation and vice versa, the terms regular language and state machine (or just machine) will be used interchangeably in this document.

Ragel outputs machines to C, C++, Objective-C, D, Java or Ruby code. The output is designed to be generic and is not bound to any particular input or processing method. A Ragel machine expects to have data passed to it in buffer blocks. When there is no more input, the machine can be queried for acceptance. In this way, a Ragel machine can be used to simply recognize a regular language like a regular expression library. By embedding code into the regular language, a Ragel machine can also be used to parse input.

The Ragel language has many operators for constructing and manipulating machines. Machines are built up from smaller machines, to bigger ones, to the final machine representing the language that needs to be recognized or parsed.

The core state machine construction operators are those found in most theory of computation textbooks. They date back to the 1950s and are widely studied. They are based on set operations and permit one to think of languages as a set of strings. They are Union, Intersection, Difference, Concatenation and Kleene Star. Put together, these operators make up what most people know as regular expressions. Ragel also provides a scanner construction operator and provides operators for explicitly constructing machines using a state chart method. In the state chart method, one joins machines together without any implied transitions and then explicitly specifies where epsilon transitions should be drawn.

The state machine manipulation operators are specific to Ragel. They allow the programmer

to access the states and transitions of regular language's corresponding machine. There are two uses of the manipulation operators. The first and primary use is to embed code into transitions and states, allowing the programmer to specify the actions of the state machine.

Ragel attempts to make the action embedding facility as intuitive as possible. To do so, a number of issues need to be addressed. For example, when making a nondeterministic specification into a DFA using machines that have embedded actions, new transitions are often made that have the combined actions of several source transitions. Ragel ensures that multiple actions associated with a single transition are ordered consistently with respect to the order of reference and the natural ordering implied by the construction operators.

The second use of the manipulation operators is to assign priorities in transitions. Priorities provide a convenient way of controlling any nondeterminism introduced by the construction operators. Suppose two transitions leave from the same state and go to distinct target states on the same character. If these transitions are assigned conflicting priorities, then during the determinization process the transition with the higher priority will take precedence over the transition with the lower priority. The lower priority transition gets abandoned. The transitions would otherwise be combined into a new transition that goes to a new state that is a combination of the original target states. Priorities are often required for segmenting machines. The most common uses of priorities have been encoded into a set of simple operators that should be used instead of priority embeddings whenever possible.

For the purposes of embedding, Ragel divides transitions and states into different classes. There are four operators for embedding actions and priorities into the transitions of a state machine. It is possible to embed into start transitions, finishing transitions, all transitions and pending out transitions. The embedding of pending out transitions is a special case. These transition embeddings get stored in the final states of a machine. They are transferred to any transitions that may be made going out of the machine by a concatenation or kleene star operator.

There are several more operators for embedding actions into states. Like the transition embeddings, there are various different classes of states that the embedding operators access. For example, one can access start states, final states or all states, among others. Unlike the transition embeddings, there are several different types of state action embeddings. These are executed at various different times during the processing of input. It is possible to embed actions which are exectued on all transitions that enter into a state, all transitions out of a state, transitions taken on the error event, or transitions taken on the EOF event.

Within actions, it is possible to influence the behaviour of the state machine. The user can write action code that jumps or calls to another portion of the machine, changes the current character being processed, or breaks out of the processing loop. With the state machine calling feature Ragel can be used to parse languages that are not regular. For example, one can parse balanced parentheses by calling into a parser when an open bracket character is seen and returning to the state on the top of the stack when the corresponding closing bracket character is seen. More complicated context-free languages such as expressions in C, are out of the scope of Ragel.

Ragel also provides a scanner construction operator which can be used to build scanners much the same way that Lex is used. The Ragel generated code, which relies on user-defined variables for backtracking, repeatedly tries to match patterns to the input, favouring longer patterns over shorter ones and patterns that appear ahead of others when the lengths of the possible matches are identical. When a pattern is matched the associated action is executed.

The key distinguishing feature between scanners in Ragel and scanners in Lex is that Ragel patterns may be arbitrary Ragel expressions and can therefore contain embedded code. With a Ragel-based scanner the user need not wait until the end of a pattern before user code can be executed.

Scanners do take Ragel out of the domain of pure state machines and require the user to maintain the backtracking related variables. However, scanners integrate well with regular state machine instantiations. They can be called to or jumped to only when needed, or they can be called out of or jumped out of when a simpler, pure state machine model is appropriate.

Two types of output code style are available. Ragel can produce a table-driven machine or a directly executable machine. The directly executable machine is much faster than the table-driven. On the other hand, the table-driven machine is more compact and less demanding on the host language compiler. It is better suited to compiling large state machines.

1.4 Related Work

Lex is perhaps the best-known tool for constructing parsers from regular expressions. In the Lex processing model, generated code attempts to match one of the user's regular expression patterns, favouring longer matches over shorter ones. Once a match is made it then executes the code associated with the pattern and consumes the matching string. This process is repeated until the input is fully consumed.

Through the use of start conditions, related sets of patterns may be defined. The active set may be changed at any time. This allows the user to define different lexical regions. It also allows the user to link patterns together by requiring that some patterns come before others. This is quite like a concatenation operation. However, use of Lex for languages that require a considerable amount of pattern concatenation is inappropriate. In such cases a Lex program deteriorates into a manually specified state machine, where start conditions define the states and pattern actions define the transitions. Lex is therefore best suited to parsing tasks where the language to be parsed can be described in terms of regions of tokens.

Lex is useful in many scenarios and has undoubtedly stood the test of time. There are, however, several drawbacks to using Lex. Lex can impose too much overhead for parsing applications where buffering is not required because all the characters are available in a single string. In these cases there is structure to the language to be parsed and a parser specification tool can help, but employing a heavyweight processing loop that imposes a stream "pull" model and dynamic input buffer allocation is inappropriate. An example of this kind of scenario is the conversion of floating point numbers contained in a string to their corresponding numerical values.

Another drawback is the very issue that Ragel attempts to solve. It is not possible to execute a user action while matching a character contained inside a pattern. For example, if scanning a programming language and string literals can contain newlines which must be counted, a Lex user must break up a string literal pattern so as to associate an action with newlines. This forces the definition of a new start condition. Alternatively the user can reprocess the text of the matched string literal to count newlines.

The Re2C program defines an input processing model similar to that of Lex. Re2C focuses on making generated state machines run very fast and integrate easily into any program, free of dependencies. Re2C generates directly executable code and is able to claim that generated parsers run nearly as fast as their hand-coded equivalents. This is very important for user adoption, as programmers are reluctant to use a tool when a faster alternative exists. A consideration to ease of use is also important because developers need the freedom to integrate the generated code as they see fit.

Many scripting languages provide ways of composing parsers by linking regular expressions using program logic. For example, Sed and Awk are two established Unix scripting tools that allow the programmer to exploit regular expressions for the purpose of locating and extracting text

of interest. High-level programming languages such as Perl, Python, PHP and Ruby all provide regular expression libraries that allow the user to combine regular expressions with arbitrary code.

In addition to supporting the linking of regular expressions with arbitrary program logic, the Perl programming language permits the embedding of code into regular expressions. Perl embeddings do not translate into the embedding of code into deterministic state machines. Perl regular expressions are in fact not fully compiled to deterministic machines when embedded code is involved. They are instead interpreted and involve backtracking. This is shown by the following Perl program. When it is fed the input `abcd` the interpretor attempts to match the first alternative, printing `a1 b1`. When this possibility fails it backtracks and tries the second possibility, printing `a2 b2`, at which point it succeeds.

```
print "YES\n" if ( <STDIN> =~
        /( a (?{ print "a1 "; }) b (?{ print "b1 "; }) cX ) |
         ( a (?{ print "a2 "; }) b (?{ print "b2 "; }) cd )/x )
```

In Ragel there is no regular expression interpretor. Aside from the scanner operator, all Ragel expressions are made into deterministic machines and the run time simply moves from state to state as it consumes input. An equivalent parser expressed in Ragel would attempt both of the alternatives concurrently, printing `a1 a2 b1 b2`.

1.5 Development Status

Ragel is a relatively new tool and is under continuous development. As a rough release guide, minor revision number changes are for implementation improvements and feature additions. Major revision number changes are for implementation and language changes that do not preserve backwards compatibility. Though in the past this has not always held true: changes that break code have crept into minor version number changes. Typically, the documentation lags behind the development in the interest of documenting only the lasting features. The latest changes are always documented in the ChangeLog file.

Chapter 2

Constructing State Machines

2.1 Ragel State Machine Specifications

A Ragel input file consists of a host language code file with embedded machine specifications. Ragel normally passes input straight to output. When it sees a machine specification it stops to read the Ragel statements and possibly generate code in place of the specification. Afterwards it continues to pass input through. There can be any number of FSM specifications in an input file. A multi-line FSM spec starts with `%%{` and ends with `}%%`. A single-line FSM spec starts with `%%` and ends at the first newline.

While Ragel is looking for FSM specifications it does basic lexical analysis on the surrounding input. It interprets literal strings and comments so a `%%` sequence in either of those will not trigger the parsing of an FSM specification. Ragel does not pass the input through any preprocessor nor does it interpret preprocessor directives itself so includes, defines and ifdef logic cannot be used to alter the parse of a Ragel input file. It is therefore not possible to use an `#if 0` directive to comment out a machine as is commonly done in C code. As an alternative, a machine can be prevented from causing any generated output by commenting out the write statements.

In Figure 2.1, a multi-line machine is used to define the machine and single line machines are used to trigger the writing of the machine data and execution code.

```
#include <string.h>
#include <stdio.h>

%%{
    machine foo;
    main :=
        ( 'foo' | 'bar' )
        0 @{ res = 1; };
}%%

%% write data;

int main( int argc, char **argv )
{
    int cs, res = 0;
    if ( argc > 1 ) {
        char *p = argv[1];
        char *pe = p + strlen(p) + 1;
        %% write init;
        %% write exec;
    }
    printf("result = %i\n", res );
    return 0;
}
```

Figure 2.1: Parsing a command line argument.

2.1.1 Naming Ragel Blocks

```
machine fsm_name;
```

The `machine` statement gives the name of the FSM. If present in a specification, this statement must appear first. If a machine specification does not have a name then Ragel uses the previous specification name. If no previous specification name exists then this is an error. Because FSM specifications persist in memory, a machine's statements can be spread across multiple machine specifications. This allows one to break up a machine across several files or draw in statements that are common to multiple machines using the `include` statement.

2.1.2 Machine Definition

```
<name> = <expression>;
```

The machine definition statement associates an FSM expression with a name. Machine expressions assigned to names can later be referenced by other expressions. A definition statement on its own does not cause any states to be generated. It is simply a description of a machine to be used later. States are generated only when a definition is instantiated, which happens when a definition is referenced in an instantiated expression.

2.1.3 Machine Instantiation

```
<name> := <expression>;
```

The machine instantiation statement generates a set of states representing an expression. Each instantiation generates a distinct set of states. The entry point is written in the generated code using the instantiation name. If the `main` machine is instantiated, its start state is used as the specification's start state and is assigned to the `cs` variable by the `write init` command. If no `main` machine is given, the start state of the last machine instantiation is used as the specification's start state.

From outside the execution loop, control may be passed to any machine by assigning the entry point to the `cs` variable. From inside the execution loop, control may be passed to any machine instantiation using `fcall`, `fgoto` or `fnext` statements.

2.1.4 Including Ragel Code

```
include FsmName "inputfile.rl";
```

The `include` statement can be used to draw in the statements of another FSM specification. Both the name and input file are optional, however at least one must be given. Without an FSM name, the given input file is searched for an FSM of the same name as the current specification. Without an input file the current file is searched for a machine of the given name. If both are present, the given input file is searched for a machine of the given name.

2.1.5 Importing Definitions

```
import "inputfile.h";
```

The `import` statement takes a literal string as an argument, interprets it as a file name, then scrapes the file for sequences of tokens that match the following forms. If the input file is a Ragel

program then tokens inside the Ragel sections are ignored. See Section 5.8.5 for a description of exporting machine definitions.

- `name = number`
- `name = lit_string`
- `"define" name number`
- `"define" name lit_string`

2.2 Lexical Analysis of a Ragel Block

Within a machine specification the following lexical rules apply to the parse of the input.

- The `#` symbol begins a comment that terminates at the next newline.
- The symbols `""`, `''`, `//`, `[]` behave as the delimiters of literal strings. With them, the following escape sequences are interpreted:

  ```
  \0 \a \b \t \n \v \f \r
  ```

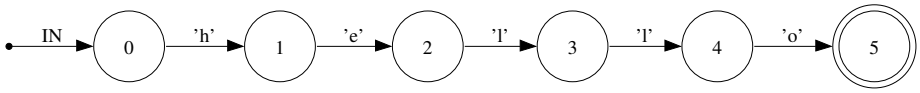
  A backslash at the end of a line joins the following line onto the current. A backslash preceding any other character removes special meaning. This applies to terminating characters and to special characters in regular expression literals. As an exception, regular expression literals do not support escape sequences as the operands of a range within a list. See the bullet on regular expressions in Section 2.3.
- The symbols `{}` delimit a block of host language code that will be embedded into the machine as an action. Within the block of host language code, basic lexical analysis of C/C++ comments and strings is done in order to correctly find the closing brace of the block. With the exception of FSM commands embedded in code blocks, the entire block is preserved as is for identical reproduction in the output code.
- The pattern `[+-]?[0-9]+` denotes an integer in decimal format. Integers used for specifying machines may be negative only if the alphabet type is signed. Integers used for specifying priorities may be positive or negative.
- The pattern `0x[0-9A-Fa-f]+` denotes an integer in hexadecimal format.
- The keywords are `access`, `action`, `alphtype`, `getkey`, `write`, `machine` and `include`.
- The pattern `[a-zA-Z_][a-zA-Z_0-9]*` denotes an identifier.
- Any amount of whitespace may separate tokens.

2.3 Basic Machines

The basic machines are the base operands of regular language expressions. They are the smallest unit to which machine construction and manipulation operators can be applied.

In the diagrams that follow the symbol `df` represents the default transition, which is taken if no other transition can be taken. The symbol `cr` represents the carriage return character, `nl` represents the newline character (aka line feed) and the symbol `sp` represents the space character.

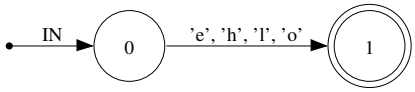
- `'hello'` – Concatenation Literal. Produces a machine that matches the sequence of characters in the quoted string. If there are 5 characters there will be 6 states chained together with the characters in the string. See Section 2.2 for information on valid escape sequences.



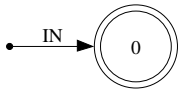
  It is possible to make a concatenation literal case-insensitive by appending an `i` to the string, for example `'cmd'i`.

- `"hello"` – Identical to the single quoted version.

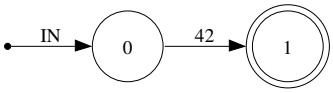
- `[hello]` – Or Expression. Produces a union of characters. There will be two states with a transition for each unique character between the two states. The `[]` delimiters behave like the quotes of a literal string. For example, `[ \t]` means tab or space. The `or` expression supports character ranges with the `-` symbol as a separator. The meaning of the union can be negated using an initial `^` character as in standard regular expressions. See Section 2.2 for information on valid escape sequences in `or` expressions.



- `''`, `""`, and `[]` – Zero Length Machine. Produces a machine that matches the zero length string. Zero length machines have one state that is both a start state and a final state.

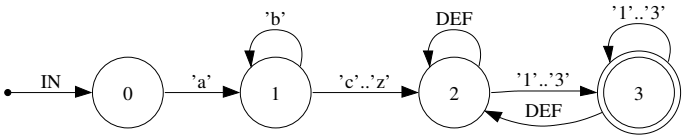


- `42` – Numerical Literal. Produces a two state machine with one transition on the given number. The number may be in decimal or hexadecimal format and should be in the range allowed by the alphabet type. The minimum and maximum values permitted are defined by the host machine that Ragel is compiled on. For example, numbers in a `short` alphabet on an i386 machine should be in the range `-32768` to `32767`.

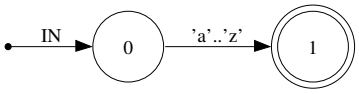


- `/simple_regex/` – Regular Expression. Regular expressions are parsed as a series of expressions that will be concatenated together. Each concatenated expression may be a literal character, the any character specified by the `.` symbol, or a union of characters specified by the `[]` delimiters. If the first character of a union is `^` then it matches any character not in the list. Within a union, a range of characters can be given by separating the first and last characters of the range with the `-` symbol. Each concatenated machine may have repetition specified by following it with the `*` symbol. The standard escape sequences described in Section 2.2 are supported everywhere in regular expressions except as the operands of a range within in a list. This notation also supports the `i` trailing option. Use it to produce case-insensitive machines, as in `/GET/i`.

Ragel does not support very complex regular expressions because the desired results can always be achieved using the more general machine construction operators listed in Section 2.5. The following diagram shows the result of compiling `/ab*[c-z].*[123]/`.



- `'a' .. 'z'` – Range. Produces a machine that matches any characters in the specified range. Allowable upper and lower bounds of the range are concatenation literals of length one and numerical literals. For example, `0x10..0x20`, `0..63`, and `'a'..'z'` are valid ranges. The bounds should be in the range allowed by the alphabet type.



- `variable_name` – Lookup the machine definition assigned to the variable name given and use an instance of it. See Section 2.1.2 for an important note on what it means to reference a variable name.

- `builtin_machine` – There are several built-in machines available for use. They are all two state machines for the purpose of matching common classes of characters. They are:

  - `any` – Any character in the alphabet.
  - `ascii` – Ascii characters. `0..127`
  - `extend` – Ascii extended characters. This is the range `-128..127` for signed alphabets and the range `0..255` for unsigned alphabets.
  - `alpha` – Alphabetic characters. `[A-Za-z]`
  - `digit` – Digits. `[0-9]`
  - `alnum` – Alpha numerics. `[0-9A-Za-z]`
  - `lower` – Lowercase characters. `[a-z]`
  - `upper` – Uppercase characters. `[A-Z]`
  - `xdigit` – Hexadecimal digits. `[0-9A-Fa-f]`
  - `cntrl` – Control characters. `0..31`
  - `graph` – Graphical characters. `[!-~]`
  - `print` – Printable characters. `[ -~]`
  - `punct` – Punctuation. Graphical characters that are not alphanumerics. `[!-/:-@[-‘{-~]`
  - `space` – Whitespace. `[\t\v\f\n\r ]`
  - `zlen` – Zero length string. `""`
  - `empty` – Empty set. Matches nothing. `^any`

2.4 Operator Precedence

The following table shows operator precedence from lowest to highest. Operators in the same precedence group are evaluated from left to right.

1	`,`	Join
2	`| & - --`	Union, Intersection and Subtraction
3	`. <: :> :>>`	Concatenation
4	`:`	Label
5	`->`	Epsilon Transition
6	`> @ $ %`	Transitions Actions and Priorities
	`>/ $/ %/ </ @/ <>/`	EOF Actions
	`>! $! %! <! @! <>!`	Global Error Actions
	`>^ $^ %^ <^ @^ <>^`	Local Error Actions
	`>~ $~ %~ <~ @~ <>~`	To-State Actions
	`>* $* %* <* @* <>*`	From-State Action
7	`* ** ? + {n} {,n} {n,} {n,m}`	Repetition
8	`! ^`	Negation and Character-Level Negation
9	`( <expr> )`	Grouping

2.5 Regular Language Operators

When using Ragel it is helpful to have a sense of how it constructs machines. The determinization process can produce results that seem unusual to someone not familiar with the NFA to DFA conversion algorithm. In this section we describe Ragel's state machine operators. Though the operators are defined using epsilon transitions, it should be noted that this is for discussion only. The epsilon transitions described in this section do not persist, but are immediately removed by the determinization process which is executed in every operation. Ragel does not make use of any nondeterministic intermediate state machines.

To create an epsilon transition between two states `x` and `y` is to copy all of the properties of `y` into `x`. This involves drawing in all of `y`'s to-state actions, EOF actions, etc., in addition to its transitions. If `x` and `y` both have a transition out on the same character, then the transitions must be combined. During transition combination a new transition is made that goes to a new state that is the combination of both target states. The new combination state is created using the same epsilon transition method. The new state has an epsilon transition drawn to all the states that compose it. Since every time an epsilon transition is drawn the creation of new epsilon transitions may be triggered, the process of drawing epsilon transitions is repeated until there are no more epsilon transitions to be made.

A very common error that is made when using Ragel is to make machines that do too much at once. That is, to create machines that have unintentional nondeterminism. This usually results from being unaware of the common strings between machines that are combined together using the regular language operators. This can involve never leaving a machine, causing its actions to be propagated through all the following states. Or it can involve an alternation where both branches are unintentionally taken simultaneously.

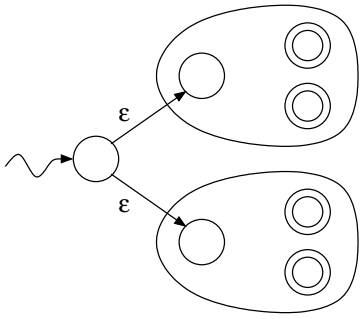
This problem forces one to think hard about the language that needs to be matched. To guard against this kind of problem one must ensure that the machine specification is divided up using boundaries that do not allow ambiguities from one portion of the machine to the next. See Chapter 4 for more on this problem and how to solve it.

The Graphviz tool is an immense help when debugging improperly compiled machines or otherwise learning how to use Ragel. In many cases, practical parsing programs will be too large to completely visualize with Graphviz. The proper approach is to reduce the language to the smallest subset possible that still exhibits the characteristics that one wishes to learn about or to fix. This can be done without modifying the source code using the `-M` and `-S` options at the frontend. If a machine cannot be easily reduced, embeddings of unique actions can be very useful for tracing a particular component of a larger machine specification, since action names are written out on transition labels.

2.5.1 Union

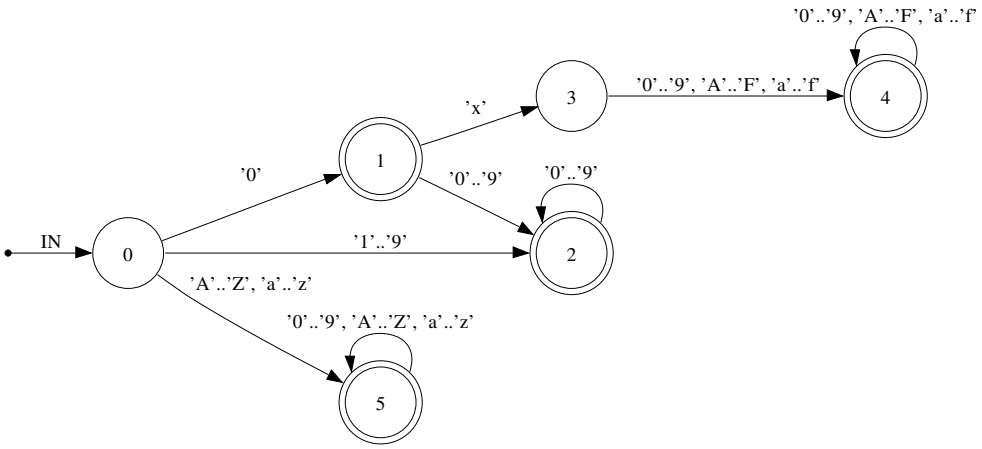
`expr | expr`

The union operation produces a machine that matches any string in machine one or machine two. The operation first creates a new start state. Epsilon transitions are drawn from the new start state to the start states of both input machines. The resulting machine has a final state set equivalent to the union of the final state sets of both input machines. In this operation, there is the opportunity for nondeterminism among both branches. If there are strings, or prefixes of strings that are matched by both machines then the new machine will follow both parts of the alternation at once. The union operation is shown below.



The following example demonstrates the union of three machines representing common tokens.

```
# Hex digits, decimal digits, or identifiers
main := '0x' xdigit+ | digit+ | alpha alnum*;
```

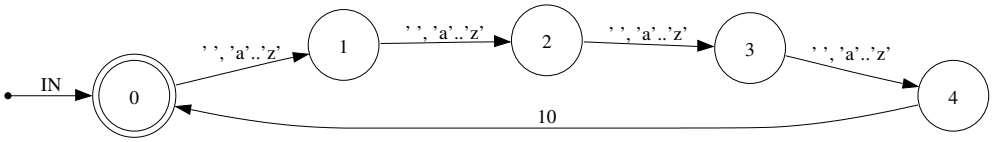


2.5.2 Intersection

```
expr & expr
```

Intersection produces a machine that matches any string that is in both machine one and machine two. To achieve intersection, a union is performed on the two machines. After the result has been made deterministic, any final state that is not a combination of final states from both machines has its final state status revoked. To complete the operation, paths that do not lead to a final state are pruned from the machine. Therefore, if there are any such paths in either of the expressions they will be removed by the intersection operator. Intersection can be used to require that two independent patterns be simultaneously satisfied as in the following example.

```
# Match lines four characters wide that contain
# words separated by whitespace.
main :=
    /[^\n][^\n][^\n][^\n]\n/* &
    (/[a-z][a-z]*/ | [ \n])**;
```

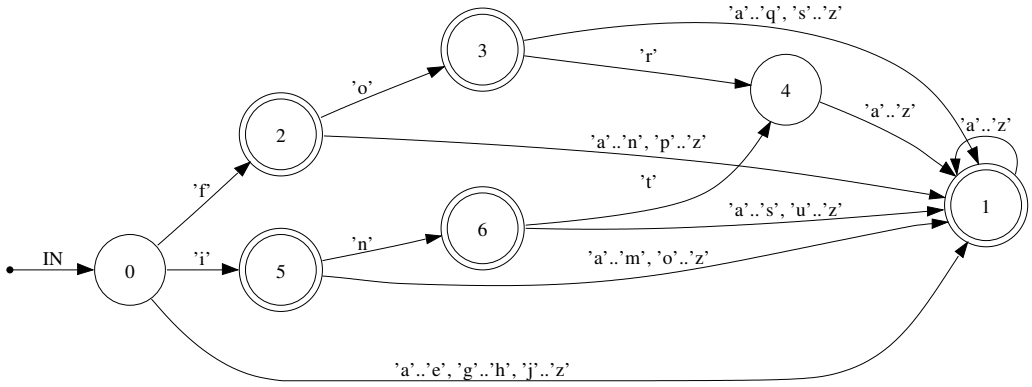


2.5.3 Difference

```
expr - expr
```

The difference operation produces a machine that matches strings that are in machine one but are not in machine two. To achieve subtraction, a union is performed on the two machines. After the result has been made deterministic, any final state that came from machine two or is a combination of states involving a final state from machine two has its final state status revoked. As with intersection, the operation is completed by pruning any path that does not lead to a final state. The following example demonstrates the use of subtraction to exclude specific cases from a set.

```
# Subtract keywords from identifiers.
main := /[a-z][a-z]*/ - ( 'for' | 'int' );
```

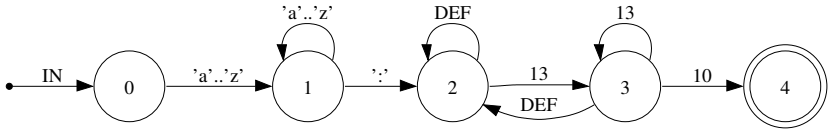


2.5.4 Strong Difference

```
expr -- expr
```

Strong difference produces a machine that matches any string of the first machine that does not have any string of the second machine as a substring. In the following example, strong subtraction is used to excluded `CRLF` from a sequence. In the corresponding visualization, the label `DEF` is short for default. The default transition is taken if no other transition can be taken.

```
crlf = '\r\n';
main := [a-z]+ ':' ( any* -- crlf ) crlf;
```



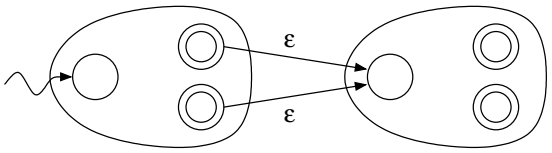
This operator is equivalent to the following.

```
expr - ( any* expr any* )
```

2.5.5 Concatenation

```
expr . expr
```

Concatenation produces a machine that matches all the strings in machine one followed by all the strings in machine two. Concatenation draws epsilon transitions from the final states of the first machine to the start state of the second machine. The final states of the first machine lose their final state status, unless the start state of the second machine is final as well. Concatenation is the default operator. Two machines next to each other with no operator between them results in the machines being concatenated together.

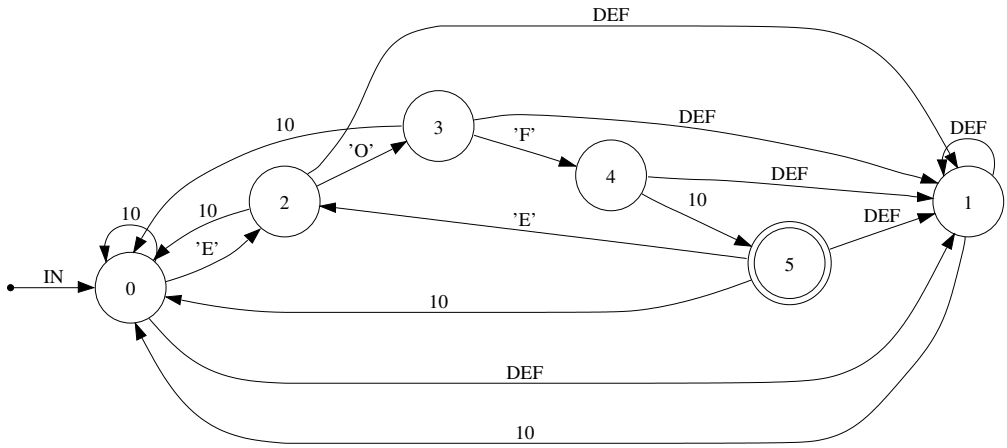


The opportunity for nondeterministic behaviour results from the possibility of the final states of the first machine accepting a string that is also accepted by the start state of the second machine. The most common scenario that this happens in is the concatenation of a machine that repeats some pattern with a machine that gives a termination string, but the repetition machine does not exclude the termination string. The example in Section 2.5.4 guards against this. Another example is the expression `("'" any* "'")`. When executed the thread of control will never leave the `any*` machine. This is a problem especially if actions are embedded to process the characters of the `any*` component.

In the following example, the first machine is always active due to the nondeterministic nature of concatenation. This particular nondeterminism is intended however because we wish to permit EOF strings before the end of the input.

```
# Require an eof marker on the last line.
```

```
main := /[^\n]*\n/* . 'EOF\n';
```

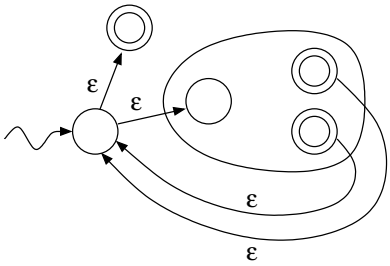


Note: There is a language ambiguity involving concatenation and subtraction. Because concatenation is the default operator for two adjacent machines there is an ambiguity between subtraction of a positive numerical literal and concatenation of a negative numerical literal. For example, `(x-7)` could be interpreted as `(x . -7)` or `(x - 7)`. In the Ragel language, the subtraction operator always takes precedence over concatenation of a negative literal. Precedence was given to the subtraction-based interpretation so as to adhere to the rule that the default concatenation operator takes effect only when there are no other operators between two machines. Beware of writing machines such as `(any -1)` when what is desired is a concatenation of `any` and -1. Instead write `(any . -1)` or `(any (-1))`. If in doubt of the meaning of your program do not rely on the default concatenation operator; always use the `.` symbol.

2.5.6 Kleene Star

`expr*`

The machine resulting from the Kleene Star operator will match zero or more repetitions of the machine it is applied to. It creates a new start state and an additional final state. Epsilon transitions are drawn between the new start state and the old start state, between the new start state and the new final state, and between the final states of the machine and the new start state. After the machine is made deterministic the effect is of the final states getting all the transitions of the start state.

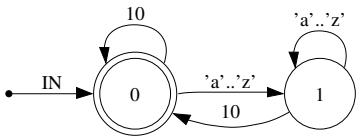


The possibility for nondeterministic behaviour arises if the final states have transitions on any of the same characters as the start state. This is common when applying kleene star to an alternation of tokens. Like the other problems arising from nondeterministic behavior, this is discussed in

more detail in Chapter 4. This particular problem can also be solved by using the longest-match construction discussed in Section 6.3 on scanners.

In this simple example, there is no nondeterminism introduced by the exterior kleene star due to the newline at the end of the regular expression. Without the newline the exterior kleene star would be redundant and there would be ambiguity between repeating the inner range of the regular expression and the entire regular expression. Though it would not cause a problem in this case, unnecessary nondeterminism in the kleene star operator often causes undesired results for new Ragel users and must be guarded against.

```
# Match any number of lines with only lowercase letters.
main := /[a-z]*\n/*;
```

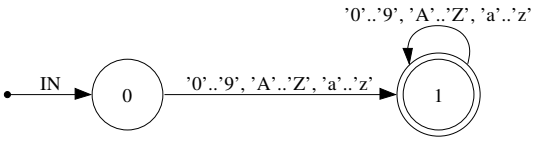


2.5.7 One Or More Repetition

`expr+`

This operator produces the concatenation of the machine with the kleene star of itself. The result will match one or more repetitions of the machine. The plus operator is equivalent to `(expr . expr*)`. The plus operator makes repetitions that cannot be zero length.

```
# Match alpha-numeric words.
main := alnum+;
```

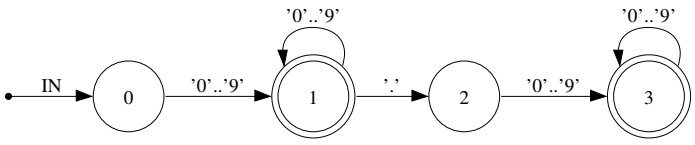


2.5.8 Optional

`expr?`

The *optional* operator produces a machine that accepts the machine given or the zero length string. The optional operator is equivalent to `(expr | '' )`. In the following example the optional operator is used to extend a token.

```
# Match integers or floats.
main := digit+ ('.' digit+)?;
```



2.5.9 Repetition

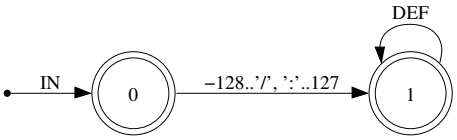
`expr {n}` – Exactly N copies of expr.
`expr {,n}` – Zero to N copies of expr.
`expr {n,}` – N or more copies of expr.
`expr {n,m}` – N to M copies of expr.

2.5.10 Negation

`!expr`

Negation produces a machine that matches any string not matched by the given machine. Negation is equivalent to `(any* - expr)`.

```
# Accept anything but a string beginning with a digit.
main := ! ( digit any* );
```



2.5.11 Character-Level Negation

`^expr`

Character-level negation produces a machine that matches any single character not matched by the given machine. Character-Level Negation is equivalent to `(any - expr)`.

2.6 State Machine Minimization

State machine minimization is the process of finding the minimal equivalent FSM accepting the language. Minimization reduces the number of states in machines by merging equivalent states. It does not change the behaviour of the machine in any way. It will cause some states to be merged into one because they are functionally equivalent. State minimization is on by default. It can be turned off with the `-n` option.

The algorithm implemented is similar to Hopcroft's state minimization algorithm. Hopcroft's algorithm assumes a finite alphabet that can be listed in memory, whereas Ragel supports arbitrary integer alphabets that cannot be listed in memory. Though exact analysis is very difficult, Ragel minimization runs close to $O(n \times log(n))$ and requires $O(n)$ temporary storage where n is the number of states.

2.7 Visualization

Ragel is able to emit compiled state machines in Graphviz's Dot file format. Graphviz support allows users to perform incremental visualization of their parsers. User actions are displayed on transition labels of the graph. If the final graph is too large to be meaningful, or even drawn, the user is able to inspect portions of the parser by naming particular regular expression definitions

with the `-S` and `-M` options to the `ragel` program. Use of Graphviz greatly improves the Ragel programming experience. It allows users to learn Ragel by experimentation and also to track down bugs caused by unintended nondeterminism.

Chapter 3

User Actions

Ragel permits the user to embed actions into the transitions of a regular expression's corresponding state machine. These actions are executed when the generated code moves over a transition. Like the regular expression operators, the action embedding operators are fully compositional. They take a state machine and an action as input, embed the action, and yield a new state machine that can be used in the construction of other machines. Due to the compositional nature of embeddings, the user has complete freedom in the placement of actions.

A machine's transitions are categorized into four classes, The action embedding operators access the transitions defined by these classes. The *starting transition* operator `>` isolates the start state, then embeds an action into all transitions leaving it. The *finishing transition* operator `@` embeds an action into all transitions going into a final state. The *all transition* operator `$` embeds an action into all transitions of an expression. The *pending out transition* operator `%` provides access to yet-unmade leaving transitions.

3.1 Embedding Actions

```
action ActionName {
    /* Code an action here. */
    count += 1;
}
```

The action statement defines a block of code that can be embedded into an FSM. Action names can be referenced by the action embedding operators in expressions. Though actions need not be named in this way (literal blocks of code can be embedded directly when building machines), defining reusable blocks of code whenever possible is good practice because it potentially increases the degree to which the machine can be minimized. Within an action some Ragel expressions and statements are parsed and translated. These allow the user to interact with the machine from action code. See Section 3.4 for a complete list of statements and values available in code blocks.

3.1.1 Starting Action

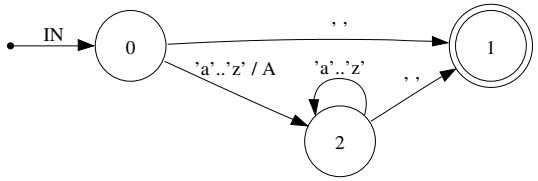
```
expr > action
```

The starting transition operator embeds an action into all transitions that leave the start state. In some machines the start state has in transtions from within the machine and the start state is effectively reused. In these cases the start state is first isolated from the rest of the machine and

the starting actions do not get re-executed.

If the start state is a final state then it is possible for the machine to never be started and the starting transitions by-passed. In the following example, the action is executed on the first transition of the machine. If the repetition machine is bypassed the action is not executed.

```
# Execute A at the beginning of a string of alpha.
action A {}
main := ( lower* >A ) . ' ';
```

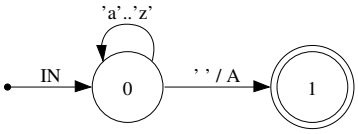


3.1.2 Finishing Action

`expr @ action`

The finishing action operator embeds an action into any transitions that go into a final state. Whether or not the machine accepts is not determined at the point the action is executed. Further input may move the machine out of the accepting state, but keep it in the machine. As in the following example, the into-final-state operator is most often used when no lookahead is necessary.

```
# Execute A when the trailing space is seen.
main := ( lower* ' ' ) @A;
```

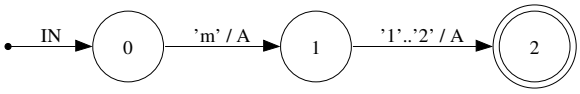


3.1.3 All Transition Action

`expr $ action`

The all transition operator embeds an action into all transitions of a machine. The action is executed whenever a transition of the machine is taken. In the following example, A is executed on every character matched.

```
# Execute A on any characters of machine one or two.
main := ( 'm1' | 'm2' ) $A;
```

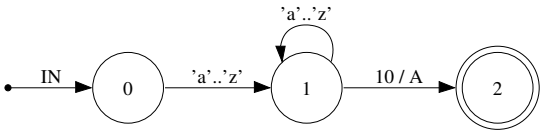


3.1.4 Pending Out (Leaving) Actions

`expr % action`

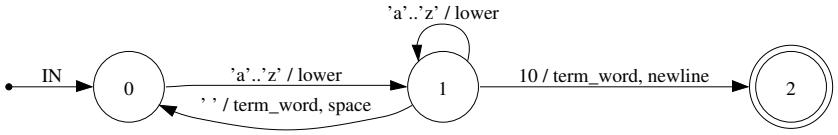
The pending out action operator embeds an action into the pending out transitions of a machine. The action is first embedded into the final states of the machine and later transferred to any transitions made going out of the machine. The transfer can be caused either by a concatenation or kleene star operation. This mechanism allows one to associate an action with the termination of a sequence, without being concerned about what particular character terminates the sequence. In the following example, A is executed when leaving the alpha machine by the newline character.

```
# Match a word followed by a newline. Execute A when
# finishing the word.
main := ( lower+ %A ) . '\n';
```



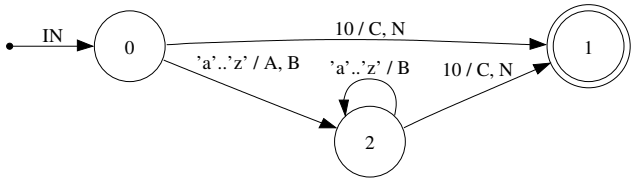
In the following example, the `term_word` action could be used to register the appearance of a word and to clear the buffer that the `lower` action used to store the text of it.

```
word = ( [a-z] @lower )+ %term_word;
main := word ( ' ' @space word )* '\n' @newline;
```



In this final example of the action embedding operators, A is executed upon the first character of the alpha machine, B is executed on all transitions of the alpha machine, C is executed when the alpha machine is exited by moving into the newline machine and N is executed when the newline machine moves into a final state.

```
# Execute A on starting the alpha machine, B on every transition
# moving through it and C upon finishing. Execute N on the newline.
main := ( lower* >A $B %C ) . '\n' @N;
```



3.2 State Action Embedding Operators

The state embedding operators allow one to embed actions into states. Like the transition embedding operators, there are several different classes of states that the operators access. The meanings of the symbols are similar to the meanings of the symbols used by the transition embedding operators. The design of the state selections was driven by a need to cover the states of an expression with a single error action.

Unlike the transition embedding operators, the state embedding operators are also distinguished by the different kinds of events that embedded actions can be associated with. Therefore the state embedding operators have two components. The first, which is the first one or two characters, specifies the class of states that the action will be embedded into. The second component specifies the type of event the action will be executed on. The symbols of the second component also have equivalent kewords.

The different classes of states are:

- `>` – the start state
- `$` – all states
- `%` – final states
- `<` – any state except the start state
- `@` – any state except final states
- `<>` – any except start and final (middle)

The different kinds of embeddings are:

- `~` – to-state actions (`to`)
- `*` – from-state actions (`from`)
- `/` – EOF actions (`eof`)
- `!` – error actions (`err`)
- `^` – local error actions (`lerr`)

3.2.1 To-State and From-State Actions

To-State Actions

```
>~action     <~action     $~action     %~action     @~action     <>~action
```

Verbose forms:

```
>to(act)     <to(act)     $to(na)      %to(name)    @to(name)    <>to(name)
>to{...}     <to{...}     $to{...}     %to{...}     @to{...}     <>to{...}
```

To-state actions are executed whenever the state machine moves into the specified state, either by a natural movement over a transition or by an action-based transfer of control such as `fgoto`. They are executed after the in-transition's actions but before the current character is advanced and tested against the end of the input block. To-state embeddings stay with the state. They are irrespective of the state's current set of transitions and any future transitions that may be added in or out of the state.

Note that the setting of the current state variable `cs` outside of the execute code is not considered by Ragel as moving into a state and consequently the to-state actions of the new current state are not executed. This includes the initialization of the current state when the machine begins. This is because the entry point into the machine execution code is after the execution of to-state actions.

From-State Actions

```
>*action     <*action     $*action     %*action     @*action     <>*action
```

Verbose forms:

```
>from(act)   <from(act)   $from(na)    %from(name)  @from(name)  <>from(name)
>from{...}   <from{...}   $from{...}   %from{...}   @from{...}   <>from{...}
```

From-state actions are executed whenever the state machine takes a transition from a state, either to itself or to some other state. These actions are executed immediately after the current character is tested against the input block end marker and before the transition to take is sought based on the current character. From-state actions are therefore executed even if a transition cannot be found and the machine moves into the error state. Like to-state embeddings, from-state embeddings stay with the state.

3.2.2 EOF Actions

```
>/action     </action     $/action    %/action      @/action      <>/action
```

Verbose forms:

```
>eof(act)    <eof(act)    $eof(na)    %eof(name)    @eof(name)    <>eof(name)
>eof{...}    <eof{...}    $eof{...}   %eof{...}     @eof{...}     <>eof{...}
```

The EOF action embedding operators enable the user to embed EOF actions into different classes of states. EOF actions are stored in states and generated with the `write eof` statement. The generated EOF code switches on the current state and executes the EOF actions associated with it.

3.2.3 Handling Errors

In many applications it is useful to be able to react to parsing errors. The user may wish to print an error message that depends on the context. It may also be desirable to consume input in an attempt to return the input stream to some known state and resume parsing. To support error handling and recovery, Ragel provides error action embedding operators. There are two kinds of error actions, regular (global) error actions and local error actions. Error actions can be used to simply report errors, or by jumping to a machine instantiation that consumes input, can attempt to recover from errors.

Global Error Actions

```
>!action     <!action     $!action    %!action      @!action      <>!action
```

Verbose forms:

```
>err(act)    <err(act)    $err(na)    %err(name)    @err(name)    <>err(name)
>err{...}    <err{...}    $err{...}   %err{...}     @err{...}     <>err{...}
```

Error actions are stored in states until the final state machine has been fully constructed. They are then transferred to the transitions that move into the error state. This transfer entails the creation of a transition from the state to the error state that is taken on all input characters that are not already covered by the state's transitions. In other words it provides a default action. Error actions can induce a recovery by altering `p` and then jumping back into the machine with `fgoto`.

Local Error Actions

```
>^action     <^action     $^action    %^action      @^action      <>^action
```

Verbose forms:

```
>lerr(act)   <lerr(act)   $lerr(na)   %lerr(name)   @lerr(name)   <>lerr(name)
>lerr{...}   <lerr{...}   $lerr{...}  %lerr{...}    @lerr{...}    <>lerr{...}
```

Like global error actions, local error actions are also stored in states until a transfer point. The transfer point is different however. Each local error action embedding is associated with a name. When a machine definition has been fully constructed, all local error action embeddings associated the same name as the machine are transferred to error transitions. Local error actions can be used to specify an action to take when a particular section of a larger state machine fails to make a match. A particular machine definition's "thread" may die and the local error actions executed, however the machine as a whole may continue to match input.

There are two forms of local error action embeddings. In the first form the name defaults to the current machine. In the second form the machine name can be specified. This is useful when it is more convenient to specify the local error action in a sub-definition that is used to construct the machine definition where the transfer should happen. To embed local error actions and explicitly state the machine on which the transfer is to happen use `(name, action)` as the action.

Example

The following example uses error actions to report an error and jump to a machine that consumes the remainder of the line when parsing fails. After consuming the line, the error recovery machine returns to the main loop.

```
action cmd_err {
    printf( "command error\n" );
    fhold; fgoto line;
}
action from_err {
    printf( "from error\n" );
    fhold; fgoto line;
}
action to_err {
    printf( "to error\n" );
    fhold; fgoto line;
}

line := [^\n]* '\n' @{ fgoto main; };

main := (
    (
        'from' @err(cmd_err)
            ( ws+ address ws+ date '\n' ) $err(from_err) |
        'to' @err(cmd_err)
            ( ws+ address '\n' ) $err(to_err)
    )
)*;
```

3.3 Action Ordering and Duplicates

When building a parser by combining smaller expressions that themselves have embedded actions, it is often the case that transitions that need to execute a number of actions on one input character are made. For example when we leave an expression, we may execute the expression's pending out action and the subsequent expression's starting action on the same input character. We must therefore devise a method for ordering actions that is both intuitive and predictable for the user and repeatable by the state machine compiler. The determinization processes cannot simply order actions by the time at which they are introduced into a transition – otherwise the programmer will be at the mercy of luck.

We associate with the embedding of each action a distinct timestamp that is used to order actions that appear together on a single transition in the final compiled state machine. To accomplish this we traverse the parse tree of regular expressions and assign timestamps to action embeddings. This algorithm is recursive in nature and quite simple. When it visits a parse tree node it assigns timestamps to all *starting* action embeddings, recurses on the parse tree, then assigns timestamps to the remaining *all*, *finishing*, and *leaving* embeddings in the order in which they appear.

Ragel does not permit actions (defined or unnamed) to appear multiple times in an action list. When the final machine has been created, actions that appear more than once in a single transition or EOF action list have their duplicates removed. The first appearance of the action is preserved. This is useful in a number of scenarios. First, it allows us to union machines with common prefixes without worrying about the action embeddings in the prefix being duplicated. Second, it prevents pending out actions from being transferred multiple times when a concatenation follows a kleene star and the two machines begin with a common character.

```
word = [a-z]+ %act;
main := word ( '\n' word )* '\n\n';
```

3.4 Values and Statements Available in Code Blocks

The following values are available in code blocks:

- `fpc` – A pointer to the current character. This is equivalent to accessing the `p` variable.
- `fc` – The current character. This is equivalent to the expression `(*p)`.
- `fcurs` – An integer value representing the current state. This value should only be read from. To move to a different place in the machine from action code use the `fgoto`, `fnext` or `fcall` statements. Outside of the machine execution code the `cs` variable may be modified.
- `ftargs` – An integer value representing the target state. This value should only be read from. Again, `fgoto`, `fnext` and `fcall` can be used to move to a specific entry point.
- `fentry(<label>)` – Retrieve an integer value representing the entry point `label`. The integer value returned will be a compile time constant. This number is suitable for later use in control flow transfer statements that take an expression. This value should not be compared against the current state because any given label can have multiple states representing it. The value returned by `fentry` will be one of the possibly multiple states the label represents.

The following statements are available in code blocks:

- `fhold;` – Do not advance over the current character. If processing data in multiple buffer blocks, the `fhold` statement should only be used once in the set of actions executed on a character. Multiple calls may result in backing up over the beginning of the buffer block. The `fhold` statement does not imply any transfer of control. It is equivalent to the `p--;` statement.

- `fexec <expr>;` – Set the next character to process. This can be used to backtrack to previous input or advance ahead. Unlike `fhold`, which can be used anywhere, `fexec` requires the user to ensure that the target of the backtrack is in the current buffer block or is known to be somewhere ahead of it. The machine will continue iterating forward until `pe` is arrived at, `fbreak` is called or the machine moves into the error state. In actions embedded into transitions, the `fexec` statement is equivalent to setting `p` to one position ahead of the next character to process. If the user also modifies `pe`, it is possible to change the buffer block entirely.

- `fgoto <label>;` – Jump to an entry point defined by `<label>`. The `fgoto` statement immediately transfers control to the destination state.

- `fgoto *<expr>;` – Jump to an entry point given by `<expr>`. The expression must evaluate to an integer value representing a state.

- `fnext <label>;` – Set the next state to be the entry point defined by `label`. The `fnext` statement does not immediately jump to the specified state. Any action code following the statement is executed.

- `fnext *<expr>;` – Set the next state to be the entry point given by `<expr>`. The expression must evaluate to an integer value representing a state.

- `fcall <label>;` – Push the target state and jump to the entry point defined by `<label>`. The next `fret` will jump to the target of the transition on which the call was made. Use of `fcall` requires the declaration of a call stack. An array of integers named `stack` and a single integer named `top` must be declared. With the `fcall` construct, control is immediately transferred to the destination state.

- `fcall *<expr>;` – Push the current state and jump to the entry point given by `<expr>`. The expression must evaluate to an integer value representing a state.

- `fret;` – Return to the target state of the transition on which the last `fcall` was made. Use of `fret` requires the declaration of a call stack with `fstack` in the struct block. Control is immediately transferred to the destination state.

- `fbreak;` – Save the current state and immediately break out of the execute loop. This statement is useful in conjunction with the `noend` write option. Rather than process input until the end marker of the input buffer is arrived at, the fbreak statement can be used to stop processing input upon seeing some end-of-string marker. It can also be used for handling exceptional circumstances. The fbreak statement does not change the pointer to the current character. After an `fbreak` call the `p` variable will point to the character that was being traversed over when the action was executed. The current state will be the target of the current transition.

Note: Once actions with control-flow commands are embedded into a machine, the user must exercise caution when using the machine as the operand to other machine construction operators. If an action jumps to another state then unioning any transition that executes that action with another transition that follows some other path will cause that other path to be lost. Using commands that manually jump around a machine takes us out of the domain of regular languages because transitions that may be conditional and that the machine construction operators are not aware of are introduced. These commands should therefore be used with caution.

Chapter 4

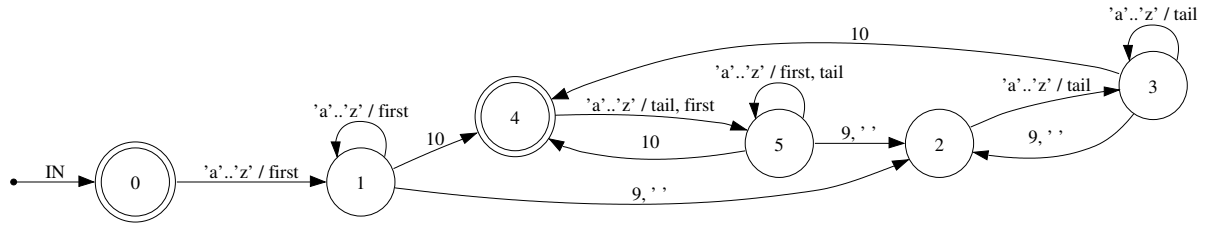
Controlling Nondeterminism

Along with the flexibility of arbitrary action embeddings comes a need to control nondeterminism in regular expressions. If a regular expression is ambiguous, then sub-components of a parser other than the intended parts may become active. This means that actions that are irrelevant to the current subset of the parser may be executed, causing problems for the programmer.

Tools that are based on regular expression engines and used for recognition tasks will usually function as intended regardless of the presence of ambiguities. It is quite common for users of scripting languages to write regular expressions that are heavily ambiguous and it generally does not matter. As long as one of the potential matches is recognized, there can be any number of other matches present. In some parsing systems the run-time engine can employ a strategy for resolving ambiguities, for example always pursuing the longest possible match and discarding others.

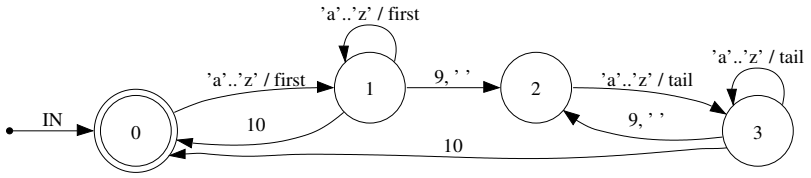
In Ragel, there is no regular expression run-time engine, just a simple state machine execution model. When we begin to embed actions and face the possibility of spurious action execution, it becomes clear that controlling nondeterminism at the machine construction level is very important. Consider the following example.

```
ws = [\n\t ];
line = word $first ( ws word $tail )* '\n';
lines = line*;
```



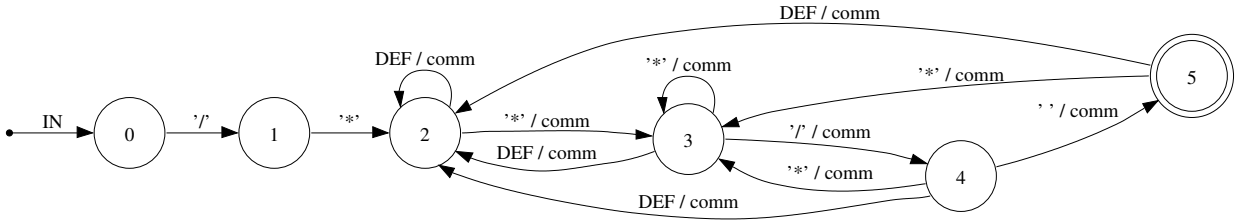
Since the `ws` expression includes the newline character, we will not finish the `line` expression when a newline character is seen. We will simultaneously pursue the possibility of matching further words on the same line and the possibility of matching a second line. Evidence of this fact is in the state tables. On several transitions both the `first` and `tail` actions are executed. The solution here is simple: exclude the newline character from the `ws` expression.

```
ws = [\t ];
line = word $first ( ws word $tail )* '\n';
lines = line*;
```



Solving this kind of problem is straightforward when the ambiguity is created by strings that are a single character long. When the ambiguity is created by strings that are multiple characters long we have a more difficult problem. The following example is an incorrect attempt at a regular expression for C language comments.

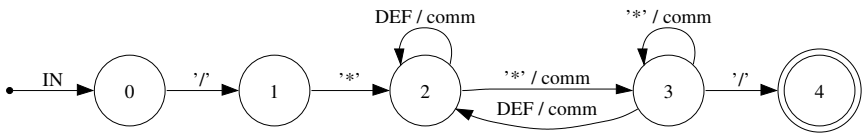
```
comment = '/*' ( any @comm )* '*/';
main := comment ' ';
```



Using standard concatenation, we will never leave the `any*` expression. We will forever entertain the possibility that a `'*/'` string that we see is contained in a longer comment and that, simultaneously, the comment has ended. The concatenation of the `comment` machine with `SP` is done to show this. When we match space, we are also still matching the comment body.

One way to approach the problem is to exclude the terminating string from the `any*` expression using set difference. We must be careful to exclude not just the terminating string, but any string that contains it as a substring. A verbose, but proper specification of a C comment parser is given by the following regular expression.

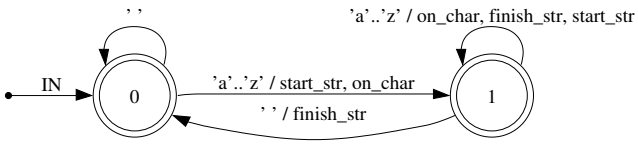
```
comment = '/*' ( ( any @comm )* - ( any* '*/' any* ) ) '*/';
```



Note that Ragel's strong subtraction operator `--` can also be used here. In doing this subtraction we have phrased the problem of controlling non-determinism in terms of excluding strings common to two expressions that interact when combined. We can also phrase the problem in terms of the transitions of the state machines that implement these expressions. During the concatenation of `any*` and `'*/'` we will be making transitions that are composed of both the loop of the first expression and the final character of the second. At this time we want the transition on the `'/'` character to take precedence over and disallow the transition that originated in the `any*` loop.

In another parsing problem, we wish to implement a lightweight tokenizer that we can utilize in the composition of a larger machine. For example, some HTTP headers have a token stream as a sub-language. The following example is an attempt at a regular expression-based tokenizer that does not function correctly due to unintended nondeterminism.

```
header_contents = (
    lower+ >start_str $on_char %finish_str |
    ' '
)*;
```



In this case, the problem with using a standard kleene star operation is that there is an ambiguity between extending a token and wrapping around the machine to begin a new token. Using the standard operator, we get an undesirable nondeterministic behaviour. Evidence of this can be seen on the transition out of state one to itself. The transition extends the string, and simultaneously, finishes the string only to immediately begin a new one. What is required is for the transitions that represent an extension of a token to take precedence over the transitions that represent the beginning of a new token. For this problem there is no simple solution that uses standard regular expression operators.

4.1 Priorities

A priority mechanism was devised and built into the determinization process, specifically for the purpose of allowing the user to control nondeterminism. Priorities are integer values embedded into transitions. When the determinization process is combining transitions that have different priorities, the transition with the higher priority is preserved and the transition with the lower priority is dropped.

Unfortunately, priorities can have unintended side effects because their operation requires that they linger in transitions indefinitely. They must linger because the Ragel program cannot know when the user is finished with a priority embedding. A solution whereby they are explicitly deleted after use is conceivable; however this is not very user-friendly. Priorities were therefore made into named entities. Only priorities with the same name are allowed to interact. This allows any number of priorities to coexist in one machine for the purpose of controlling various different regular expression operations and eliminates the need to ever delete them. Such a scheme allows the user to choose a unique name, embed two different priority values using that name and be confident that the priority embedding will be free of any side effects.

In the first form of priority embedding the name defaults to the name of the machine definition that the priority is assigned in. In this sense priorities are by default local to the current machine definition or instantiation. Beware of using this form in a longest-match machine, since there is only one name for the entire set of longest match patterns. In the second form the priority's name can be specified, allowing priority interaction across machine definition boundaries.

- `expr > int` – Sets starting transitions to have priority int.
- `expr @ int` – Sets transitions that go into a final state to have priority int.
- `expr $ int` – Sets all transitions to have priority int.
- `expr % int` – Sets pending out transitions from final states to have priority int.
  When a transition is made going out of the machine (either by concatenation or kleene star) its priority is immediately set to the pending out priority.

The second form of priority assignment allows the programmer to specify the name to which the priority is assigned.

- `expr > (name, int)` – Starting transitions.
- `expr @ (name, int)` – Finishing transitions (into a final state).
- `expr $ (name, int)` – All transitions.
- `expr % (name, int)` – Pending out transitions.

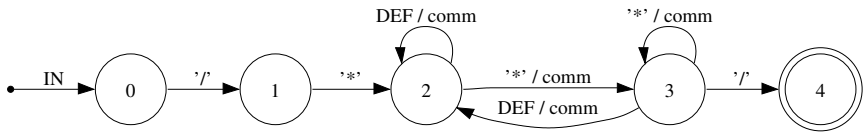
4.2 Guarded Operators that Encapsulate Priorities

Priority embeddings are a very expressive mechanism. At the same time they can be very confusing for the user. They force the user to imagine the transitions inside two interacting expressions and work out the precise effects of the operations between them. When we consider that this problem is worsened by the potential for side effects caused by unintended priority name collisions, we see that exposing the user to priorities is rather undesirable.

Fortunately, in practice the use of priorities has been necessary only in a small number of scenarios. This allows us to encapsulate their functionality into a small set of operators and fully hide them from the user. This is advantageous from a language design point of view because it greatly simplifies the design.

Going back to the C comment example, we can now properly specify it using a guarded concatenation operator which we call *finish-guarded concatenation*. From the user's point of view, this operator terminates the first machine when the second machine moves into a final state. It chooses a unique name and uses it to embed a low priority into all transitions of the first machine. A higher priority is then embedded into the transitions of the second machine that enter into a final state. The following example yields a machine identical to the example in Section 4.

```
comment = '/*' ( any @comm )* :>> '*/';
```



Another guarded operator is *left-guarded concatenation*, given by the `<:` compound symbol. This operator places a higher priority on all transitions of the first machine. This is useful if one must forcibly separate two lists that contain common elements. For example, one may need to tokenize a stream, but first consume leading whitespace.

Ragel also includes a *longest-match kleene star* operator, given by the `**` compound symbol. This guarded operator embeds a high priority into all transitions of the machine. A lower priority is then embedded into pending out transitions (in a manner similar to pending out action embeddings, described in Section 3.1.4). When the kleene star operator makes the epsilon transitions from the final states into the start state, the lower priority will be transferred to the epsilon transitions. In cases where following an epsilon transition out of a final state conflicts with an existing transition out of a final state, the epsilon transition will be dropped.

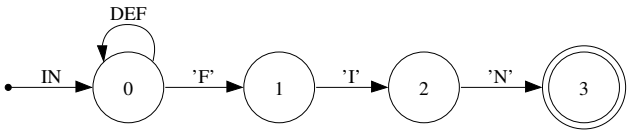
Other guarded operators are conceivable, such as guards on union that cause one alternative to take precedence over another. These may be implemented when it is clear they constitute a frequently used operation. In the next section we discuss the explicit specification of state machines using state charts.

4.2.1 Entry-Guarded Concatenation

```
expr :> expr
```

This operator concatenates two machines, but first assigns a low priority to all transitions of the first machine and a high priority to the starting transitions of the second machine. This operator is useful if from the final states of the first machine, it is possible to accept the characters in the start transitions of the second machine. This operator effectively terminates the first machine immediately upon starting the second machine, where otherwise they would be pursued concurrently. In the following example, entry-guarded concatenation is used to move out of a machine that matches everything at the first sign of an end-of-input marker.

```
# Leave the catch-all machine on the first character of FIN.
main := any* :> 'FIN';
```



Entry-guarded concatenation is equivalent to the following:

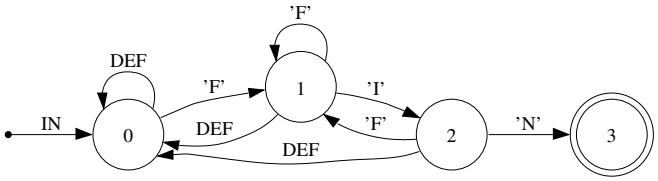
```
expr $(unique_name,0) . expr >(unique_name,1)
```

4.2.2 Finish-Guarded Concatenation

```
expr :>> expr
```

This operator is like the previous operator, except the higher priority is placed on the final transitions of the second machine. This is useful if one wishes to entertain the possibility of continuing to match the first machine right up until the second machine enters a final state. In other words it terminates the first machine only when the second accepts. In the following example, finish-guarded concatenation causes the move out of the machine that matches everything to be delayed until the full end-of-input marker has been matched.

```
# Leave the catch-all machine on the last character of FIN.
main := any* :>> 'FIN';
```



Finish-guarded concatenation is equivalent to the following:

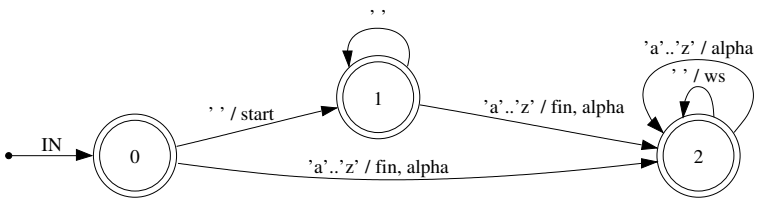
```
expr $(unique_name,0) . expr @(unique_name,1)
```

4.2.3 Left-Guarded Concatenation

```
expr <: expr
```

This operator places a higher priority on the left expression. It is useful if you want to prefix a sequence with another sequence composed of some of the same characters. For example, one can consume leading whitespace before tokenizing a sequence of whitespace-separated words as in:

```
main := ( ' '* >start %fin ) <: ( ' ' $ws | [a-z] $alpha )*;
```



Left-guarded concatenation is equivalent to the following:

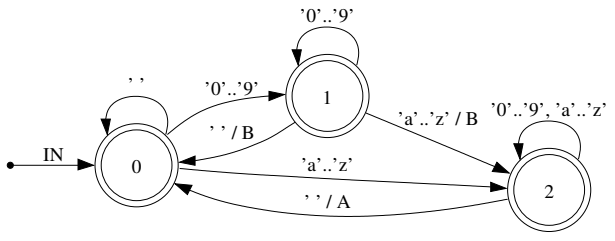
```
expr $(unique_name,1) . expr >(unique_name,0)
```

4.2.4 Longest-Match Kleene Star

```
expr**
```

This version of kleene star puts a higher priority on staying in the machine versus wrapping around and starting over. The LM kleene star is useful when writing simple tokenizers. These machines are built by applying the longest-match kleene star to an alternation of token patterns, as in the following.

```
# Repeat tokens, but make sure to get the longest match.
main := (
    lower ( lower | digit )* %A |
    digit+ %B |
    ' '
)**;
```



If a regular kleene star were used the machine above would not be able to distinguish between extending a word and beginning a new one. This operator is equivalent to:

```
( expr $(unique_name,1) %(unique_name,0) )*
```

When the kleene star is applied, transitions that go out of the machine and back into it are made. These are assigned a priority of zero by the pending out transition mechanism. This is less than the priority of one assigned to the transitions leaving the final states but not leaving the machine. When two of these transitions clash on the same character, the differing priorities cause

the transition that stays in the machine to take precedence. The transition that wraps around is dropped.

Note that this operator does not build a scanner in the traditional sense because there is never any backtracking. To build a scanner in the traditional sense use the Longest-Match machine construction described in Section 6.3.

Chapter 5

Interface to Host Program

The Ragel code generator is very flexible. The generated code has no dependencies and can be inserted in any function, perhaps inside a loop if so desired. The user is responsible for declaring and initializing a number of required variables, including the current state and the pointer to the input stream. These can live in any scope. Control of the input processing loop is also possible: the user may break out of the processing loop and return to it at any time.

In the case of C and D host languages, Ragel is able to generate very fast-running code that implements state machines as directly executable code. Since very large files strain the host language compiler, table-based code generation is also supported. In the future we hope to provide a partitioned, directly executable format that is able to reduce the burden on the host compiler by splitting large machines across multiple functions.

In the case of Java and Ruby, table-based code generation is the only code style supported. In the future this may be expanded to include other code styles.

Ragel can be used to parse input in one block, or it can be used to parse input in a sequence of blocks as it arrives from a file or socket. Parsing the input in a sequence of blocks brings with it a few responsibilities. If the parser utilizes a scanner, care must be taken to not break the input stream anywhere but token boundaries. If pointers to the input stream are taken during parsing, care must be taken to not use a pointer that has been invalidated by movement to a subsequent block. If the current input data pointer is moved backwards it must not be moved past the beginning of the current block.

Figure 5.1 shows a simple Ragel program that does not have any actions. The example tests the first argument of the program against a number pattern and then prints the machine's acceptance status.

5.1 Variables Used by Ragel

There are a number of variables which Ragel expects the user to declare. At a very minimum the `cs`, `p` and `pe` variables must be declared. In Java and Ruby code the `data` variable must also be declared. If stack-based state machine control flow statements are used then the `stack` and `top` variables are required. If a scanner is declared then the `act`, `tokstart` and `tokend` variables must be declared.

- `cs` - Current state. This must be an integer and it should persist across invocations of the machine when the data is broken into blocks that are processed independently.
- `p` - Data pointer. In C/D code this variable is expected to be a pointer to the character data

```
#include <stdio.h>
#include <string.h>
%%{
    machine foo;
    write data;
}%%
int main( int argc, char **argv )
{
    int cs;
    if ( argc > 1 ) {
        char *p = argv[1];
        char *pe = p + strlen( p );
        %%{
            main := [0-9]+ ( '.' [0-9]+ )?;

            write init;
            write exec;
            write eof;
        }%%
    }
    printf("result = %i\n", cs >= foo_first_final );
    return 0;
}
```

Figure 5.1: A basic Ragel example without any actions.

to process. It should be initialized to the beginning of the data block on every run of the machine. In Java and Ruby it is used as an offset to `data` and must be an integer. In this case it should be initialized to zero on every run of the machine.

- `pe` - Data end pointer. This should be initialized to `p` plus the data length on every run of the machine. In Java and Ruby code this should be initialized to the data length.
- `data` - This variable is only required in Java and Ruby code. It must be an array containting the data to process.
- `stack` - This must be an array of integers. It is used to store integer values representing states. If the stack must resize dynamically the Pre-push and Post-Pop statements can be used to do this (Sections 5.6 and 5.7).
- `top` - This must be an integer value and will be used as an offset to `stack`, giving the next available spot on the top of the stack.
- `act` - This must be an integer value. It is a variable sometimes used by scanner code to keep track of the most recent successful pattern match.
- `tokstart` - This must be a pointer to character data. In Java and Ruby code this must be an integer. See Section 6.3 for more information.
- `tokend` - Also a pointer to character data.

5.2 Alphtype Statement

```
alphtype unsigned int;
```

The alphtype statement specifies the alphabet data type that the machine operates on. During the compilation of the machine, integer literals are expected to be in the range of possible values of the alphtype. Supported alphabet types are `char`, `unsigned char`, `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`. The default is `char`.

5.3 Getkey Statement

```
getkey fpc->id;
```

Specify to Ragel how to retrieve the character that the machine operates on from the pointer to the current element (`p`). Any expression that returns a value of the alphabet type may be used. The getkey statement may be used for looking into element structures or for translating the character to process. The getkey expression defaults to `(*p)`. In goto-driven machines the getkey expression may be evaluated more than once per element processed, therefore it should not incur a large cost nor preclude optimization.

5.4 Access Statement

```
access fsm->;
```

The access statement allows one to tell Ragel how the generated code should access the machine data that is persistent across processing buffer blocks. This includes all variables except `p` and `pe`. This includes `cs`, `top`, `stack`, `tokstart`, `tokend` and `act`. This is useful if a machine is to be encapsulated inside a structure in C code. The access statement can be used to give the name of a pointer to the structure.

5.5 Variable Statement

```
variable p fsm->p;
```

The variable statement allows one to tell ragel how to access a specific variable. All of the variables that are declared by the user and used by Ragel can be changed. This includes `p`, `pe`, `cs`, `top`, `stack`, `tokstart`, `tokend` and `act`. In Ruby and Java code generation the `data` variable can also be changed.

5.6 Pre-Push Statement

```
prepush {
/* stack growing code */
}
```

The prepush statement allows the user to supply stack management code that is written out during the generation of fcall, immediately before the current state is pushed to the stack. This

statement can be used to test the number of available spaces and dynamically grow the stack if necessary.

5.7 Post-Pop Statement

```
postpop {
/* stack shrinking code */
}
```

The postpop statement allows the user to supply stack management code that is written out during the generation of fret, immediately after the next state is popped from the stack. This statement can be used to dynamically shrink the stack.

5.8 Write Statement

```
write <component> [options];
```

The write statement is used to generate parts of the machine. There are four components that can be generated by a write statement. These components are the state machine's data, initialization code, execution code and EOF action execution code. A write statement may appear before a machine is fully defined. This allows one to write out the data first then later define the machine where it is used. An example of this is shown in Figure 5.2.

5.8.1 Write Data

```
write data [options];
```

The write data statement causes Ragel to emit the constant static data needed by the machine. In table-driven output styles (see Section 5.11) this is a collection of arrays that represent the states and transitions of the machine. In goto-driven machines much less data is emitted. At the very minimum a start state `name_start` is generated. All variables written out in machine data have both the `static` and `const` properties and are prefixed with the name of the machine and an underscore. The data can be placed inside a class, inside a function, or it can be defined as global data.

Two variables are written that may be used to test the state of the machine after a buffer block has been processed. The `name_error` variable gives the id of the state that the machine moves into when it cannot find a valid transition to take. The machine immediately breaks out of the processing loop when it finds itself in the error state. The error variable can be compared to the current state to determine if the machine has failed to parse the input. If the machine is complete, that is from every state there is a transition to a proper state on every possible character of the alphabet, then no error state is required and this variable will be set to -1.

The `name_first_final` variable stores the id of the first final state. All of the machine's states are sorted by their final state status before having their ids assigned. Checking if the machine has accepted its input can then be done by checking if the current state is greater-than or equal to the first final state.

Data generation has several options:

- `noerror` - Do not generate the integer variable that gives the id of the error state.
- `nofinal` - Do not generate the integer variable that gives the id of the first final state.

- `noprefix` - Do not prefix the variable names with the name of the machine.

5.8.2 Write Init

```
write init;
```

The write init statement causes Ragel to emit initialization code. This should be executed once before the machine is started. At a very minimum this sets the current state to the start state. If other variables are needed by the generated code, such as call stack variables or scanner management variables, they are also initialized here.

The `nocs` option to the write init statement will cause ragel to skip intialization of the cs variable. This is useful if the user wishes to use custom logic to decide which state the specification should start in.

5.8.3 Write Exec

```
write exec [options];
```

The write exec statement causes Ragel to emit the state machine's execution code. Ragel expects several variables to be available to this code. At a very minimum, the generated code needs access to the current character position `p`, the ending position `pe` and the current state `cs`, though `pe` can be excluded by specifying the `noend` write option. The `p` variable is the cursor that the execute code will used to traverse the input. The `pe` variable should be set up to point to one position past the last valid character in the buffer.

Other variables are needed when certain features are used. For example using the `fcall` or `fret` statements requires `stack` and `top` variables to be defined. If a longest-match construction is used, variables for managing backtracking are required.

The write exec statement has one option. The `noend` option tells Ragel to generate code that ignores the end position `pe`. In this case the user must explicitly break out of the processing loop using `fbreak`, otherwise the machine will continue to process characters until it moves into the error state. This option is useful if one wishes to process a null terminated string. Rather than traverse the string to discover then length before processing the input, the user can break out when the null character is seen. The example in Figure 5.2 shows the use of the `noend` write option and the `fbreak` statement for processing a string.

5.8.4 Write EOF Actions

```
write eof;
```

The write EOF statement causes Ragel to emit code that executes EOF actions. This write statement is only relevant if EOF actions have been embedded, otherwise it does not generate anything. The EOF action code requires access to the current state.

5.8.5 Write Exports

```
write exports;
```

The export feature can be used to export simple machine definitions. Machine definitions are marked for export using the `export` keyword.

```
#include <stdio.h>
%% machine foo;
int main( int argc, char **argv )
{
    %% write data noerror nofinal;
    int cs, res = 0;
    if ( argc > 1 ) {
        char *p = argv[1];
        %%{
            main :=
                [a-z]+
                0 @{ res = 1; fbreak; };
            write init;
            write exec noend;
        }%%
    }
    printf("execute = %i\n", res );
    return 0;
}
```

Figure 5.2: Use of `noend` write option and the `fbreak` statement for processing a string.

```
export machine_to_export = 0x44;
```

When the write exports statement is used these machines are written out in the generated code. Defines are used for C and constant integers are used for D, Java and Ruby. See Section 2.1.5 for a description of the import statement.

5.9 Maintaining Pointers to Input Data

In the creation of any parser it is not uncommon to require the collection of the data being parsed. It is always possible to collect data into a growable buffer as the machine moves over it, however the copying of data is a somewhat wasteful use of processor cycles. The most efficient way to collect data from the parser is to set pointers into the input then later reference them. This poses a problem for uses of Ragel where the input data arrives in blocks, such as over a socket or from a file. If a pointer is set in one buffer block but must be used while parsing a following buffer block, some extra consideration to correctness must be made.

The scanner constructions exhibit this problem, requiring the maintenance code described in Section 6.3. If a longest-match construction has been used somewhere in the machine then it is possible to take advantage of the required prefix maintenance code in the driver program to ensure pointers to the input are always valid. If laying down a pointer one can set `tokstart` at the same spot or ahead of it. When data is shifted in between loops the user must also shift the pointer. In this way it is possible to maintain pointers to the input that will always be consistent.

In general, there are two approaches for guaranteeing the consistency of pointers to input data. The first approach is the one just described; lay down a marker from an action, then later ensure that the data the marker points to is preserved ahead of the buffer on the next execute invocation. This approach is good because it allows the parser to decide on the pointer-use boundaries, which can be arbitrarily complex parsing conditions. A downside is that it requires any pointers that are set to be corrected in between execute invocations.

The alternative is to find the pointer-use boundaries before invoking the execute routine, then

```
int have = 0;
while ( 1 ) {
    char *p, *pe, *data = buf + have;
    int len, space = BUFSIZE - have;

    if ( space == 0 ) {
        fprintf(stderr, "BUFFER OUT OF SPACE\n");
        exit(1);
    }

    len = fread( data, 1, space, stdin );
    if ( len == 0 )
        break;

    /* Find the last newline by searching backwards. */
    p = buf;
    pe = data + len - 1;
    while ( *pe != '\n' && pe >= buf )
        pe--;
    pe += 1;

    %% write exec;

    /* How much is still in the buffer? */
    have = data + len - pe;
    if ( have > 0 )
        memmove( buf, pe, have );

    if ( len < space )
        break;
}
```

Figure 5.3: An example of line-oriented processing.

pass in the data using these boundaries. For example, if the program must perform line-oriented processing, the user can scan backwards from the end of an input block that has just been read in and process only up to the first found newline. On the next input read, the new data is placed after the partially read line and processing continues from the beginning of the line. An example of line-oriented processing is given in Figure 5.3.

5.10 Running the Executables

Ragel is broken down into two parts: a frontend that compiles machines and emits them in an XML format, and a backend that generates code or a Graphviz Dot file from the XML data. The purpose of the XML-based intermediate format is to allow users to inspect their compiled state machines and to interface Ragel to other tools such as custom visualizers, code generators or analysis tools. The split also serves to reduce the complexity of the Ragel program by strictly separating the data structures and algorithms that are used to compile machines from those that are used to generate code.

The frontend program is called `ragel`. It takes as an argument the host language. This can be:

- `-C` for C/C++/Objective-C code (default)
- `-D` for D code.
- `-J` for Java code.
- `-R` for Ruby code.

There are four code backend programs. These are:

- `rlgen-cd` generate code for the C-based and D languages.
- `rlgen-java` generate code for the Java language.
- `rlgen-ruby` generate code for the Ruby language.
- `rlgen-dot` generate a Graphviz Dot file.

5.11 Choosing a Generated Code Style (C/D only)

There are three styles of code output to choose from. Code style affects the size and speed of the compiled binary. Changing code style does not require any change to the Ragel program. There are two table-driven formats and a goto driven format.

In addition to choosing a style to emit, there are various levels of action code reuse to choose from. The maximum reuse levels (`-T0`, `-F0` and `-G0`) ensure that no FSM action code is ever duplicated by encoding each transition's action list as static data and iterating through the lists on every transition. This will normally result in a smaller binary. The less action reuse options (`-T1`, `-F1` and `-G1`) will usually produce faster running code by expanding each transition's action list into a single block of code, eliminating the need to iterate through the lists. This duplicates action code instead of generating the logic necessary for reuse. Consequently the binary will be larger. However, this tradeoff applies to machines with moderate to dense action lists only. If a machine's transitions frequently have less than two actions then the less reuse options will actually produce both a smaller and a faster running binary due to less action sharing overhead. The best way to choose the appropriate code style for your application is to perform your own tests.

The table-driven FSM represents the state machine as constant static data. There are tables of states, transitions, indices and actions. The current state is stored in a variable. The execution is simply a loop that looks up the current state, looks up the transition to take, executes any actions and moves to the target state. In general, the table-driven FSM can handle any machine, produces a smaller binary and requires a less expensive host language compile, but results in slower running code. Since the table-driven format is the most flexible it is the default code style.

The flat table-driven machine is a table-based machine that is optimized for small alphabets. Where the regular table machine uses the current character as the key in a binary search for the transition to take, the flat table machine uses the current character as an index into an array of transitions. This is faster in general, however is only suitable if the span of possible characters is small.

The goto-driven FSM represents the state machine using goto and switch statements. The execution is a flat code block where the transition to take is computed using switch statements and directly executable binary searches. In general, the goto FSM produces faster code but results in a larger binary and a more expensive host language compile.

The goto-driven format has an additional action reuse level (`-G2`) that writes actions directly into the state transitioning logic rather than putting all the actions together into a single switch.

Generally this produces faster running code because it allows the machine to encode the current state using the processor's instruction pointer. Again, sparse machines may actually compile to smaller binaries when `-G2` is used due to less state and action management overhead. For many parsing applications `-G2` is the preferred output format.

Code Output Style Options	
`-T0`	binary search table-driven
`-T1`	binary search, expanded actions
`-F0`	flat table-driven
`-F1`	flat table, expanded actions
`-G0`	goto-driven
`-G1`	goto, expanded actions
`-G2`	goto, in-place actions

Chapter 6

Beyond the Basic Model

6.1 Parser Modularization

It is possible to use Ragel's machine construction and action embedding operators to specify an entire parser using a single regular expression. In many cases this is the desired way to specify a parser in Ragel. However, in some scenarios, the language to parse may be so large that it is difficult to think about it as a single regular expression. It may shift between distinct parsing strategies, in which case modularization into several coherent blocks of the language may be appropriate.

It may also be the case that patterns that compile to a large number of states must be used in a number of different contexts and referencing them in each context results in a very large state machine. In this case, an ability to reuse parsers would reduce code size.

To address this, distinct regular expressions may be instantiated and linked together by means of a jumping and calling mechanism. This mechanism is analogous to the jumping to and calling of processor instructions. A jump command, given in action code, causes control to be immediately passed to another portion of the machine by way of setting the current state variable. A call command causes the target state of the current transition to be pushed to a state stack before control is transferred. Later on, the original location may be returned to with a return statement. In the following example, distinct state machines are used to handle the parsing of two types of headers.

```
action return { fret; }
action call_date { fcall date; }
action call_name { fcall name; }

# A parser for date strings.
date := [0-9][0-9] '/'
        [0-9][0-9] '/'
        [0-9][0-9][0-9][0-9] '\n' @return;

# A parser for name strings.
name := ( [a-zA-Z]+ | ' ' )** '\n' @return;

# The main parser.
headers =
    ( 'from' | 'to' ) ':' @call_name |
    ( 'departed' | 'arrived' ) ':' @call_date;
```

```
main := headers*;
```

Calling and jumping should be used carefully as they are operations that take one out of the domain of regular languages. A machine that contains a call or jump statement in one of its actions should be used as an argument to a machine construction operator only with considerable care. Since DFA transitions may actually represent several NFA transitions, a call or jump embedded in one machine can inadvertently terminate another machine that it shares prefixes with. Despite this danger, theses statements have proven useful for tying together sub-parsers of a language into a parser for the full language, especially for the purpose of modularizing code and reducing the number of states when the machine contains frequently recurring patterns.

Section 3.4 describes the jump and call statements that are used to transfer control. These statements make use of two variables that must be declared by the user, `stack` and `top`. The `stack` variable must be an array of integers and `top` must be a single integer, which will point to the next available space in `stack`. Sections 5.6 and 5.7 describe the Pre-Push and Post-Pop statements which can be used to implement a dynamically resizable array.

6.2 Referencing Names

This section describes how to reference names in epsilon transitions and action-based control-flow statements such as `fgoto`. There is a hierarchy of names implied in a Ragel specification. At the top level are the machine instantiations. Beneath the instantiations are labels and references to machine definitions. Beneath those are more labels and references to definitions, and so on.

Any name reference may contain multiple components separated with the `::` compound symbol. The search for the first component of a name reference is rooted at the join expression that the epsilon transition or action embedding is contained in. If the name reference is not contained in a join, the search is rooted at the machine definition that the epsilon transition or action embedding is contained in. Each component after the first is searched for beginning at the location in the name tree that the previous reference component refers to.

In the case of action-based references, if the action is embedded more than once, the local search is performed for each embedding and the result is the union of all the searches. If no result is found for action-based references then the search is repeated at the root of the name tree. Any action-based name search may be forced into a strictly global search by prefixing the name reference with `::`.

The final component of the name reference must resolve to a unique entry point. If a name is unique in the entire name tree it can be referenced as is. If it is not unique it can be specified by qualifying it with names above it in the name tree. However, it can always be renamed.

6.3 Scanners

Scanners are very much intertwined with regular-languages and their corresponding processors. For this reason Ragel supports the definition of Scanners. The generated code will repeatedly attempt to match patterns from a list, favouring longer patterns over shorter patterns. In the case of equal-length matches, the generated code will favour patterns that appear ahead of others. When a scanner makes a match it executes the user code associated with the match, consumes the input then resumes scanning.

```
<machine_name> := |*
        pattern1 => action1;
```

```
        pattern2 => action2;
        ...
    *|;
```

On the surface, Ragel scanners are similar to those defined by Lex. Though there is a key distinguishing feature: patterns may be arbitrary Ragel expressions and can therefore contain embedded code. With a Ragel-based scanner the user need not wait until the end of a pattern before user code can be executed.

Scanners can be used to process sub-languages, as well as for tokenizing programming languages. In the following example a scanner is used to tokenize the contents of a header field.

```
word = [a-z]+;
head_name = 'Header';

header := |*
    word;
    ' ';
    '\n' => { fret; };
*|;

main := ( head_name ':' @{ fcall header; } )*;
```

The scanner construction has a purpose similar to the longest-match kleene star operator `**`. The key difference is that a scanner is able to backtrack to match a previously matched shorter string when the pursuit of a longer string fails. For this reason the scanner construction operator is not a pure state machine construction operator. It relies on several variables which enable it to backtrack and make pointers to the matched input text available to the user. For this reason scanners must be immediately instantiated. They cannot be defined inline or referenced by another expression. Scanners must be jumped to or called.

Scanners rely on the `tokstart`, `tokend` and `act` variables to be present so that it can backtrack and make pointers to the matched text available to the user. If input is processed using multiple calls to the execute code then the user must ensure that when a token is only partially matched that the prefix is preserved on the subsequent invocation of the execute code.

The `tokstart` variable must be defined as a pointer to the input data. It is used for recording where the current token match begins. This variable may be used in action code for retrieving the text of the current match. Ragel ensures that in between tokens and outside of the longest-match machines that this pointer is set to null. In between calls to the execute code the user must check if `tokstart` is set and if so, ensure that the data it points to is preserved ahead of the next buffer block. This is described in more detail below.

The `tokend` variable must also be defined as a pointer to the input data. It is used for recording where a match ends and where scanning of the next token should begin. This can also be used in action code for retrieving the text of the current match.

The `act` variable must be defined as an integer type. It is used for recording the identity of the last pattern matched when the scanner must go past a matched pattern in an attempt to make a longer match. If the longer match fails it may need to consult the `act` variable. In some cases, use of the `act` variable can be avoided because the value of the current state is enough information to determine which token to accept, however in other cases this is not enough and so the `act` variable is used.

```
a)          A stream "of characters" to be scanned.
            |        |          |
            p        tokstart   pe

b)          "of characters" to be scanned.
            |          |        |
            tokstart   p        pe
```

Figure 6.1: Following an invocation of the execute code there may be a partially matched token (a). The data of the partially matched token must be preserved ahead of the new data on the next invocation (b).

When the longest-match operator is in use, the user's driver code must take on some buffer management functions. The following algorithm gives an overview of the steps that should be taken to properly use the longest-match operator.

- Read a block of input data.
- Run the execute code.
- If `tokstart` is set, the execute code will expect the incomplete token to be preserved ahead of the buffer on the next invocation of the execute code.
  - Shift the data beginning at `tokstart` and ending at `pe` to the beginning of the input buffer.
  - Reset `tokstart` to the beginning of the buffer.
  - Shift `tokend` by the distance from the old value of `tokstart` to the new value. The `tokend` variable may or may not be valid. There is no way to know if it holds a meaningful value because it is not kept at null when it is not in use. It can be shifted regardless.
- Read another block of data into the buffer, immediately following any preserved data.
- Run the scanner on the new data.

Figure 6.1 shows the required handling of an input stream in which a token is broken by the input block boundaries. After processing up to and including the "t" of "characters", the prefix of the string token must be retained and processing should resume at the "e" on the next iteration of the execute code.

If one uses a large input buffer for collecting input then the number of times the shifting must be done will be small. Furthermore, if one takes care not to define tokens that are allowed to be very long and instead processes these items using pure state machines or sub-scanners, then only a small amount of data will ever need to be shifted.

Since scanners attempt to make the longest possible match of input, in some cases they are not able to identify a token upon parsing its final character, they must wait for a lookahead character. For example if trying to match words, the token match must be triggered on following whitespace in case more characters of the word have yet to come. The user must therefore arrange for an EOF character to be sent to the scanner to flush out any token that has not yet been matched. The user can exclude a single character from the entire scanner and use this character as the EOF character, possibly specifying an EOF action. For most scanners, zero is a suitable choice for the EOF character.

Alternatively, if whitespace is not significant and ignored by the scanner, the final real token can be flushed out by simply sending an additional whitespace character on the end of the stream.

```
int have = 0;
bool done = false;
while ( !done ) {
    /* How much space is in the buffer? */
    int space = BUFSIZE - have;
    if ( space == 0 ) {
        /* Buffer is full. */
        cerr << "TOKEN TOO BIG" << endl;
        exit(1);
    }

    /* Read in a block after any data we already have. */
    char *p = inbuf + have;
    cin.read( p, space );
    int len = cin.gcount();

    /* If no data was read, send the EOF character. */
    if ( len == 0 ) {
        p[0] = 0, len++;
        done = true;
    }

    char *pe = p + len;
    %% write exec;

    if ( cs == RagelScan_error ) {
        /* Machine failed before finding a token. */
        cerr << "PARSE ERROR" << endl;
        exit(1);
    }

    if ( tokstart == 0 )
        have = 0;
    else {
        /* There is a prefix to preserve, shift it over. */
        have = pe - tokstart;
        memmove( inbuf, tokstart, have );
        tokend = inbuf + (tokend-tokstart);
        tokstart = inbuf;
    }
}
```

Figure 6.2: A processing loop for a scanner.

If the real stream ends with whitespace then it will simply be extended and ignored. If it does not, then the last real token is guaranteed to be flushed and the dummy EOF whitespace ignored. An example scanner processing loop is given in Figure 6.2.

6.4 State Charts

In addition to supporting the construction of state machines using regular languages, Ragel provides a way to manually specify state machines using state charts. The comma operator combines

machines together without any implied transitions. The user can then manually link machines by specifying epsilon transitions with the `->` operator. Epsilon transitions are drawn between the final states of a machine and entry points defined by labels. This makes it possible to build machines using the explicit state-chart method while making minimal changes to the Ragel language.

An interesting feature of Ragel's state chart construction method is that it can be mixed freely with regular expression constructions. A state chart may be referenced from within a regular expression, or a regular expression may be used in the definition of a state chart transition.

6.4.1 Join

```
expr , expr , ...
```

Join a list of machines together without drawing any transitions, without setting up a start state, and without designating any final states. Transitions between the machines may be specified using labels and epsilon transitions. The start state must be explicity specified with the "start" label. Final states may be specified with an epsilon transition to the implicitly created "final" state. The join operation allows one to build machines using a state chart model.

6.4.2 Label

```
label: expr
```

Attaches a label to an expression. Labels can be used as the target of epsilon transitions and explicit control transfer statements such as `fgoto` and `fnext` in action code.

6.4.3 Epsilon

```
expr -> label
```

Draws an epsilon transition to the state defined by `label`. Epsilon transitions are made deterministic when join operators are evaluated. Epsilon transitions that are not in a join operation are made deterministic when the machine definition that contains the epsilon is complete. See Section 6.2 for information on referencing labels.

6.4.4 Simplifying State Charts

There are two benefits to providing state charts in Ragel. The first is that it allows us to take a state chart with a full listing of states and transitions and simplify it in selective places using regular expressions.

The state chart method of specifying parsers is very common. It is an effective programming technique for producing robust code. The key disadvantage becomes clear when one attempts to comprehend a large parser specified in this way. These programs usually require many lines, causing logic to be spread out over large distances in the source file. Remembering the function of a large number of states can be difficult and organizing the parser in a sensible way requires discipline because branches and repetition present many file layout options. This kind of programming takes a specification with inherent structure such as looping, alternation and concatenation and expresses it in a flat form.

If we could take an isolated component of a manually programmed state chart, that is, a subset of states that has only one entry point, and implement it using regular language operators then we

could eliminate all the explicit naming of the states contained in it. By eliminating explicitly named states and replacing them with higher-level specifications we simplify a state machine specification.

For example, sometimes chains of states are needed, with only a small number of possible characters appearing along the chain. These can easily be replaced with a concatenation of characters. Sometimes a group of common states implement a loop back to another single portion of the machine. Rather than manually duplicate all the transitions that loop back, we may be able to express the loop using a kleene star operator.

Ragel allows one to take this state map simplification approach. We can build state machines using a state map model and implement portions of the state map using regular languages. In place of any transition in the state machine, entire sub-state machines can be given. These can encapsulate functionality defined elsewhere. An important aspect of the Ragel approach is that when we wrap up a collection of states using a regular expression we do not lose access to the states and transitions. We can still execute code on the transitions that we have encapsulated.

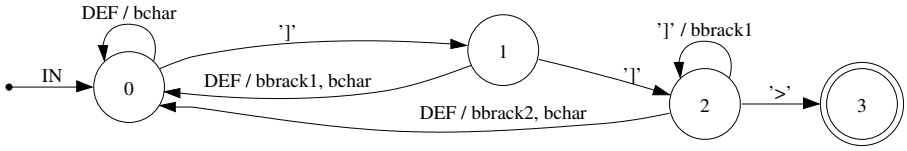
6.4.5 Dropping Down One Level of Abstraction

The second benefit of incorporating state charts into Ragel is that it permits us to bypass the regular language abstraction if we need to. Ragel's action embedding operators are sometimes insufficient for expressing certain parsing tasks. In the same way that is useful for C language programmers to drop down to assembly language programming using embedded assembler, it is sometimes useful for the Ragel programmer to drop down to programming with state charts.

In the following example, we wish to buffer the characters of an XML CDATA sequence. The sequence is terminated by the string `]]>`. The challenge in our application is that we do not wish the terminating characters to be buffered. An expression of the form `any* @buffer :>> ']]>'` will not work because the buffer will always contain the characters `]]` on the end. Instead, what we need is to delay the buffering of `]` characters until a time when we abandon the terminating sequence and go back into the main loop. There is no easy way to express this using Ragel's regular expression and action embedding operators, and so an ability to drop down to the state chart method is useful.

```
action bchar { buff( fpc ); }    # Buffer the current character.
action bbrack1 { buff( "]" ); }
action bbrack2 { buff( "]]" ); }

CDATA_body =
start: (
     ']' -> one |
     (any-']') @bchar ->start
),
one: (
     ']' -> two |
     [^\]] @bbrack1 @bchar ->start
),
two: (
     '>' -> final |
     ']' @bbrack1 -> two |
     [^>\]] @bbrack2 @bchar ->start
);
```

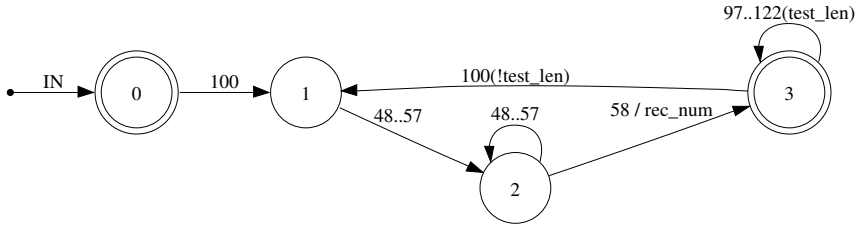


6.5 Semantic Conditions

Many communication protocols contain variable-length fields, where the length of the field is given ahead of the field as a value. This problem cannot be expressed using regular languages because of its context-dependent nature. The prevalence of variable-length fields in communication protocols motivated us to introduce semantic conditions into the Ragel language.

A semantic condition is a block of user code that is executed immediately before a transition is taken. If the code returns a value of true, the transition may be taken. We can now embed code that extracts the length of a field, then proceed to match n data values.

```
action rec_num { i = 0; n = getnumber(); }
action test_len { i++ < n }
data_fields = (
    'd'
    [0-9]+ %rec_num
    ':'
    ( [a-z] when test_len )*
)**;
```

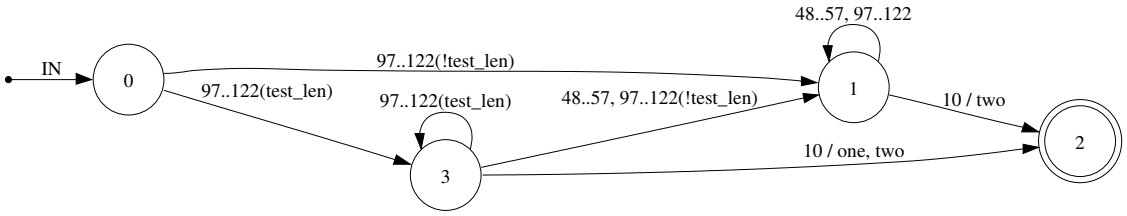


The Ragel implementation of semantic conditions does not force us to give up the compositional property of Ragel definitions. For example, a machine that tests the length of a field using conditions can be unioned with another machine that accepts some of the same strings, without the two machines interfering with one another. The user need not be concerned about whether or not the result of the semantic condition will affect the matching of the second machine.

To see this, first consider that when a user associates a condition with an existing transition, the transition's label is translated from the base character to its corresponding value in the space that represents "condition c true". Should the determinization process combine a state that has a conditional transition with another state that has a transition on the same input character but without a condition, then the condition-less transition first has its label translated into two values, one to its corresponding value in the space that represents "condition c true" and another to its corresponding value in the space that represents "condition c false". It is then safe to combine the two transitions. This is shown in the following example. Two intersecting patterns are unioned, one with a condition and one without. The condition embedded in the first pattern does not affect the second pattern.

```
action test_len { i++ < n }
action one { /* accept pattern one */ }
```

```
action two { /* accept pattern two */ }
patterns =
    ( [a-z] when test_len )+ %one |
    [a-z][a-z0-9]* %two;
main := patterns '\n';
```



There are many more potential uses for semantic conditions. The user is free to use arbitrary code and may therefore perform actions such as looking up names in dictionaries, validating input using external parsing mechanisms or performing checks on the semantic structure of input seen so far. In the next section we describe how Ragel accommodates several common parser engineering problems.

Note: The semantic condition feature works only with alphabet types that are smaller in width than the `long` type. To implement semantic conditions Ragel needs to be able to allocate characters from the alphabet space. Ragel uses these allocated characters to express "character C with condition P true" or "C with P false." Since internally Ragel uses longs to store characters there is no room left in the alphabet space unless an alphabet type smaller than long is used.

6.6 Implementing Lookahead

There are a few strategies for implementing lookahead in Ragel programs. Pending out actions, which are described in Section 3.1.4, can be used as a form of lookahead. Ragel also provides the `fhold` directive which can be used in actions to prevent the machine from advancing over the current character. It is also possible to manually adjust the current character position by shifting it backwards using `fexec`, however when this is done, care must be taken not to overstep the beginning of the current buffer block. In both the use of `fhold` and `fexec` the user must be cautious of combining the resulting machine with another in such a way that the transition on which the current position is adjusted is not combined with a transition from the other machine.

6.7 Parsing Recursive Language Structures

In general Ragel cannot handle recursive structures because the grammar is interpreted as a regular language. However, depending on what needs to be parsed it is sometimes practical to implement the recursive parts using manual coding techniques. This often works in cases where the recursive structures are simple and easy to recognize, such as in the balancing of parentheses

One approach to parsing recursive structures is to use actions that increment and decrement counters or otherwise recognize the entry to and exit from recursive structures and then jump to the appropriate machine defnition using `fcall` and `fret`. Alternatively, semantic conditions can be used to test counter variables.

A more traditional approach is to call a separate parsing function (expressed in the host language) when a recursive structure is entered, then later return when the end is recognized.