Parsing Computer Languages with an Automaton Compiled from a Single Regular Expression

Adrian D. Thurston

Software Technology Laboratory, Queen's University, Kingston, Canada
thurston@cs.queensu.ca

When a programmer is faced with the task of producing a parser for a context-free language there are many tools to choose from. We find that programmers avoid such tools when making parsers for simpler, domain-specific computer languages, such as file formats, communication protocols and end-user inputs. Since these languages often meet the criteria for regular languages, the extra run-time effort required for supporting the recursive nature of context-free languages is wasted.

Existing parsing tools based on regular expressions such as Lex, TLex, Re2C, Sed, Awk and Perl focus on building parsers by combining small regular expressions using some form of program logic. For example, Lex defines a token sequence model. None of these tools support the construction of an entire parser using a single regular expression. Doing so has a number of advantages. From the regular expression we gain a clear and concise statement of the solution. From the state machine we obtain a very fast and robust executable that lends itself to many kinds of analysis and visualization. In this work we present the machine construction and action execution model of Ragel, which allows the embedding of user code into regular expressions to support the single-expression model.

The Ragel language provides the regular expression operators *union*, *concatenation*, *kleene star*, *difference* and *intersection* for constructing parsers. The full set of operators is given in the manual, available from Ragel's homepage.

User actions can be embedded into regular expressions in arbitrary places using action embedding operators. The *entering 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 every transition. The *pending out transition* operator `%` enqueues an embedding for the yet-unmade leaving transitions. It allows the user to specify an action to be taken upon the termination of a sequence, prior to the definition of the termination characters.

When a parser is built by combining expressions with embedded actions, transitions which need to execute a number of actions on one input character are often synthesized. To yield an action ordering that is intuitive and predictable for the user, we recursively traverse the parse tree of regular expressions and assign timestamps to action embeddings. When the traversal visits a parse tree node it assigns timestamps to all *entering* action embeddings, recurses on the

children, then assigns timestamps to the remaining embedding types in the order in which they appear.

During the composition of a parser, the programmer must be careful to ensure that only the intended sub-components of the parser are active at any given time. Otherwise, there is a danger that actions which are irrelevant to the current section of the parser will be executed. In the context of embedded actions, unintended nondeterminism causes spurious action execution.

In most situations, regular expression operators are adequate for segmenting the components of a parser, but they sometimes lead to complicated and verbose parser specifications. In one case, there is no regex-based means of controlling nondeterminism; when we attempt to use the standard kleene star operator to parse a token stream we create an ambiguity between extending a token and wrapping around the machine to begin a new token.

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. To avoid unintended side-effects, priorities were made into named entities; only priority embeddings with the same name are allowed to interact.

Using priority embeddings for controlling nondeterminism can be tedious and confusing for the programmer. Fortunately, the use of priorities has been necessary only in a small number of scenarios. This allows us to encapsulate the priority functionality into a set of operators and hide priority embeddings from the user.

The *left-guarded concatenation* operator, given by the `<:` compound symbol, 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. The *entry-guarded concatenation* operator, given by `:>`, terminates the first machine when the second machine begins. The *finish-guarded concatenation* operator, given by `:>>`, terminates the first machine when the second machine moves into a final state. The *longest-match kleene star* operator, given by `**`, first embeds a high priority into all transitions and a low priority into pending out transitions. When it makes the epsilon transitions from the final states into the start state, they will be given a lower priority than the existing transitions.

```
header_list := ( lower+ ':' ' '* <: (
    ( lower ( lower | digit )* ) >mark %id |
    [ \t]+ >mark %ws |
    '\n\t' @cont )** '\n' )*;
```