AAAAA 11 User's Manual

Anders Andersson

August 19, 1999

Introduction

AAAAA is an easy-to-use tool for working with regular expressions, finite automata and grammars. The target audience is mainly students taking courses in automata theory. AAAAA supports regular expressions, generalized finite automata (subsumes NFAs), regular grammars, context-free grammars, and Type 0 grammars. A number of operators on languages are also supported. Finally, there is also a built in parser generator for full LR(1) grammars [1].

Command Line Options

The configuration of AAAAA 11 is by command line options.

`-buggywm`	Some window managers have bugs that interfere with AAAAA. This option turns off the normal window placement, letting the window manager place the windows itself. Try this option if some of AAAAA's windows inexplicably disappear from the screen. (When this happens the window has typically only been visible a fraction of a second.)
`-debug`	Gives more detailed information in case of internal errors. Please note that (on most platforms) this includes writing a file called `a5.log` in the user's home directory.
`-font` *size*	Default is `normal`, `large` enlarges the input font used, and `Large` also enlarges some of the output fonts.
`-lang` *language*	Decides which language to use for the user interface. Any unique prefix of the language names `american`,`english`, or `swedish` is acceptable. (It is not case sensitive.) Defaults to English.
`-source` *file*	Source the given Tcl [3, 4] file after startup. Mainly used to provide custom evaluators, see the section "Advanced Topics II".
`-stack` *kilobytes*	The amount of global stack that AAAAA allocates. Defaults to 8192, i.e. 8 MB. (Note that AAAAA also allocates other memory.)

Summary of Keys, Buttons and Menus

Escape Key

It is possible to abort certain lengthy computations by hitting the Escape key. This is mostly relevant for "Generate strings" and "Show strings".

Keyboard Input

Many of the text fields in AAAAA support the use of special characters. For example, regular expressions support $\cup$ and grammars support $\rightarrow$. These characters are available through buttons in the lower right corner of the widget, but there are also keyboard shortcuts available, according to the following table.

Keyboard Shortcuts

Symbol	Key
$\emptyset$	Alt o *or* Alt 0
ε	Alt e *or* Alt 1
$\cup$	Alt u
$\ldots$	Alt .
$\neg$	Alt !
$\rightarrow$	Alt - *or* Alt p

File Menu

New Session	Discard all languages and start over.
Open Session	Discard all, then open new.
Open Additional Languages	Open new without discarding existing.
Save All	Save all the languages to file.
Save Partial	Lets the user specify which languages to save.
Source Tcl Code	Load (source) a Tcl file.[3, 4]
About	Display information about the program.
Quit	Exits AAAAA.

New Menu

Regular expression	
Generalized Finite Automaton	See comment below
Regular Grammar	
Context-Free Grammar	
Type 0 Grammar	
Regular lexer + LR(1) Parser	See comment below

Most of the above items should be self explanatory, but it is worth making two comments. First, a generalized finite automaton is a non-deterministic finite automaton which allows regular expressions instead of just strings to guide the transitions. Hence, it describes a regular language. NFAs can be considered to be a special case.

Second, "Regular lexer + LR(1) Parser" refers to the AAAAA parser generator, described more fully in later sections. See also [1, 2] for background on parser generators.

Comments on Selected Buttons

Most of the buttons that appear in the user interface is fairly self explanatory, but some comments may be helpful.

Operators and Equality

The buttons below the "New" menu button in the left-hand tool-bar is referred to as *operator buttons*. They each launch a menu, which is cascaded in the case of a binary operator. Some buttons are only defined for regular languages, others also allow languages formed from context-free languages and other operators. Note that $\bullet$ should be read as a placeholder for an argument.

Included among the operator buttons is also a button for language comparison for regular languages, although that is not, strictly speaking, an operator. It is placed with the operator buttons mainly because it behaves like one from the point of view of the user interface.

Operator Buttons

$\bullet \cup \bullet$		Union of two languages
$\bullet \cap \bullet$		Intersection of two languages
$\bullet - \bullet$		Set difference between two languages
$\bullet \triangle \bullet$		$L_1 \triangle L_2 = (L_1 - L_2) \cup (L_2 - L_1)$
$\bullet \ \bullet$		Concatenation of two languages
$\bullet / \bullet$	*	$L_1/L_2 = \{w \mid wu \in L_1 \ \wedge \ u \in L_2\}$
$? = ?$	*	Compares two regular languages for equality.
$\bullet^*$		$L^* = \varepsilon \cup L \cup LL \cup LLL \cup \cdots$
$\bullet^+$		$L^+ = L \cup LL \cup LLL \cup \cdots$
Reverse		Reversal of the strings in a language
Prefix	*	Prefices of the strings in a language
Suffix	*	Suffices of the strings in a language

* Regular languages only

Micellaneous Buttons

Rename

When first created, a language recieves a generated temporary name, -L1-, -L2-, -L3-, and so on. Click the "Rename" button to change the name of the active language (the one displayed). If there is a name clash, AAAAA will rename the language slightly, by adding a number to it.

Clone

Creates a new language with the same definition as the active one. Useful when one wants to create several similar languages.

Kill

Deletes the active language.

Show Strings

Most languages has a button marked "Show strings". It opens a window listing the strings in the language, which is scrollable and bounded in length only by available memory. The strings are listed shortest first, and in alphabetical (ASCII) order within each length. This is not available for Type 0 grammars, since it is not computable in that case, but Type 0 grammars instead offer "Generate strings", which is similar but not ordered.

Generate Strings

Similar to "Show strings" but only apply to Type 0 grammars. The strings are listed in order of derivation. It is guaranteed that shorter derivations appear before longer, otherwise the order is intentionally undefined.

As an additional feature, you may click on a string, thereby highlighting it, and then click "Show production" to study the derivation leading to the string. This feature is available to substitute for the lack of a "Member?" button for Type 0 grammars.

Member?

Next to the "Member?" button there is a text field. When the "Member?" button is invoked, AAAAA checks if the string in this field is in the language or not, and reports the result. It also tries to offer an explanation (or proof) of why this is the case. This takes different form depending on how the language was defined. For example, if the language was defined by a grammar, it displays a parse tree as proof of membership, and offers no proof of non-membership. In most cases, it can only offer proof of membership, but not of non-membership.

Build

Available in parsers only. Constructs a parser from the description of it. Opens a window representing the generated parser, allowing inputs to the parser to be evaluated. The window also contains a "Clear input" button, which deletes the previous input to enter a new one, a "Reinitialize" button which has no function in the standard setup, but may be defined by a custom evaluator (see the section on parsers), and a "Parse" button that submits the input to the parser.

Update

Roughly the same as dismissing a window, and then relaunching it by the same command which originally created it.

Add State

When entering a generalized finite automaton in tabular form, this button adds another state to the automaton, and hence another row and column to the table.

Clean up

When entering a generalized finite automaton in tabular form, this button removes any isolated states from the automaton, i.e. it removes a row and the corresponding column if they are both empty.

Draw it

When entering a generalized finite automaton in tabular form, this button displays the corresponding graphical representation.

Edit

When AAAAA displays a read-only automaton in tabular form, this button adds a new language described by a GFA identical to the displayed one.

The Regular Language Buttons

Common to the regular languages, but no other languages, there is the following set of buttons:

Minimal DFA with natural alphabet	Displays the minimal DFA in tabular form, under the assumption that the alphabet consists of the symbols which occurs in the language definition.
Minimal DFA without dead states	As above, but it may formally be an NFA, since any states from which no accepting states can be reached are removed.
Minimal DFA with full alphabet	As above, but the alphabet is assumed to be full ASCII, and all states are shown. May have one more state than the minimal without dead states.
Grammar	Displays a regular grammar for the language.
Regular expression	Displays a "natural" regular expression for the language. It is simplified somewhat, but need not be normal.
Normalized reg. expr.	Displays a regular expression unique to the language, constructed from the minimal DFA.

Notes on Syntax

Regular Expressions

The symbols $\emptyset$, ε, $\cup$, (,), and $*$ have their standard meaning. In addition, $+$ is analogous to $*$, except that $\alpha+$ does not match the empty string unless α does.

The backslash character is used to escape special characters such as $*$ and (. For example, `\␣` matches the blank character, and `\)` matches a right parenthesis. To match a backslash character, use `\\`. You cannot escape a special character not present in the ASCII alphabet, such as $\cup$.

Further, we allow the syntax #*num* to denote ASCII character with the number *num*. For example, `#97` is equivalent to `a`. This feature is mainly used to match non-printing characters.

Finally, we allow a more compact form for character sets. The symbol ... (i.e. a single symbol, *not* three periods) is used to denote a range of characters, such as $0 \ldots 9$ which matches any single digit. The symbol $\neg$ is used to get the complement of a set of characters (specified as the union of a set of ranges or single characters). For example, $\neg$`a` matches any ASCII character except "a", $\neg 0 \ldots 9$ matches any character except a digit, and $\neg(\mathtt{a} \ldots \mathtt{z} \cup \mathtt{A} \ldots \mathtt{Z} \cup \_)$ matches any character except (English alphabet) letters and underscores.

The special ways to write characters with \ and # has highest precedence, followed by the operator ..., followed by the operator $\neg$, followed by the operators $*$ and $+$, followed by concatination. Union has the lowest precedence.

Grammars

The special characters $\rightarrow$ and ϵ has the standard meaning, and $|$ is used to provide alternative right-hand sides to the same left-hand side.

Non-terminals begin with an upper-case letter or an underscore, and consists of letters, digits and underscores. Note that $\mathtt{S} \rightarrow \mathtt{SS}$ is *not* the same as $\mathtt{S} \rightarrow \mathtt{S}\ \mathtt{S}$, since the former is interpreted as just a single symbol `SS` on the right-hand side.

There cannot be more than one rule per line, and newlines inside a rule can only occur before and/or after a $|$.

The # notation is used in the same way as for regular expressions, as is the principle of using backslash to escape special characters.

LR(1) Grammars

The only difference in syntax between an LR(1) grammar and any other context-free grammar is that you may use regular expressions inside curly braces as terminals of the grammar. Se the next section for the semantics. Note, however, that terminals must be written exactly identical to be considered to be the same token.

The LR(1) Parser Generator

For general information on parser generators, consult [1, 2]. For specifics on LR(1), consult [1]. A parser generated by AAAAAs parser generator operates in two phases, lexer and parser. In the first phase, the input is divided into words by matching a set of regular expressions to the longest possible prefix of the input. That prefix is called a *lexeme*. If the lexeme is matched by a special regular expression called the *skip expression*, then it is discarded, otherwise the expression matching it, called its *token type*, is treated as the next non-terminal with respect to the second phase. The token type together with the lexeme is also called a *token*.

The regular expressions giving the token types are generally enclosed in curly braces, for example `{a+}`. For single characters, this is not necessary, so the syntax becomes a pure extension of the one used for ordinary context-free grammars. Note that the token type for two such occurrences are the same only if their textual representation is identical. In particular, `{aa*}` is a different token type than `{a+}`, even though they match the same lexemes.

Advanced Topic I – Notes on Perfomance

When using some of AAAAA's features, performance can be sluggish. This section describes when and why this is likely to happen, and how to avoid unnecessary bad performance.

Showing Strings in Operator Defined Languages

Occasionally languages defined in terms of operators can consume huge amounts of machine resources during the "Show strings" operation. We will explain when and why this commonly occurs, and how to avoid it when possible.

How the Operators Work

When regular languages are combined with operators, the result is another regular language. This means that there is no speed penalty at all for languages defined in terms of operators.

When an operator is applied to languages of which at least one is non-regular, then the operators are computed in a brute-force manner. For union and symmetric difference, this means that strings are generated for both languages, then combined in the appropriate way, and for intersection and set difference AAAAA generates the left-hand language, and tests the strings for membership in the right-hand language. This means that there is a possibility that a lot of strings has to be generated, but only very few will appear in the answer.

Pitfall 1 – intersection left and right

The easiest way to get poor performance is by the intersection operator with a non-sparse non-regular language on the right, and a sparse language on the right. For example, if you take the well-formed parenthesis on the left:

$$S \;\rightarrow\; (\,S\,) \;\mid\; S\,S \;\mid\; \varepsilon$$

and the strings consisting of a block of left parenthesis followed by a block of right parenthesis, `\(*\)*`, and consider the number of strings in the language as a function of the string length, you realize that the left-hand language has an exponential growth, the right-hand side has a quadratic growth, and that the intersection contains exactly one string of each even length. Thus, writing a separate grammar for the language is the most efficient solution, but simply interchanging the order of the two languages in the intersection is a reasonable alternative.

As a general practical tip: If the intersection is slow, try it the other way around. It may well improve drastically.

Pitfall 2 – doing things in the wrong order

When combining a set of languages together with operators, try to keep the regular languages together as much as possible. Suppose you wish to form the union of three languages, L_1, L_2, and L_3, where L_1 and L_3 are regular, and L_2 is non-regular. Forming this union as $(L_1 \cup L_2) \cup L_3$ is less efficient than forming it as $(L_1 \cup L_3) \cup L_2$. In the former case, all three languages will be required to generate strings, which are then merged in two steps. In the latter case, a DFA for $L_1 \cup L_3$ will be formed, and only two generations and one merge will be performed. In the case of union, the difference is not too critical, but when it comes to intersection and set difference it may well be, depending on the sparseness of the languages involved.

Type 0 Grammars

It is very easy to write Type 0 grammars which executes very slowly in AAAAA, and it is relatively hard to write grammars that executes efficiently. This is due to the inherent non-deterministic nature of the grammars, and the deterministic nature of the computer which simulates it.

The key to efficient grammars is to avoid non-deterministic choices as much as possible, and especially to avoid multiple non-terminals which are simultanously active in the production. It is much more efficient to have a single working non-terminal going from end to end in the string than it is to have a set of non-terminals all going in a single direction, for example. Using non-terminals to mark the end of the string is still a good idea, but you should avoid letting them disappear on their own accord. It is much preferable to have the worker seek them out and destroy them near the end of the computation.

$$\begin{array}{rcl} S & \to & \varepsilon \ \mid \ a\,b\,N\,S\,c \\ b\,N\,a & \to & a\,b\,N \\ b\,N\,b & \to & b\,b\,N \\ b\,N\,c & \to & b\,c \end{array}$$

This grammar for the language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is straightforward and clear, but unfortunately not too effecient when simulated on a computer. The reason is that there are so many ways to produce the same string. All the N-s move to the right, but it is possible to move the first one all the way before starting on the next one, or just part of the way, and so on. Another grammar for the same language, which is more suited to computer simulation, is the following:

$$\begin{array}{rcl} S & \to & \varepsilon \ \mid \ a\,L\,b\,c \\ a\,L & \to & a \ \mid \ a\,a\,R \\ b\,L & \to & L\,b \\ R\,b & \to & b\,R \\ R\,c & \to & L\,b\,c\,c \end{array}$$

This grammar is slightly more complex to understand, but much more efficient. There is a single worker, moving back and fourth over the block of b-s, alternating roles between a left-mover (L) and a rigth-mover (R). When it hits the left end of the block of b-s, it either terminates or it drops an extra a and changes direction. That non-deterministic choice is necessary of course, since it determines the length of the string that is produced. When it hits the right end of the block of b-s, it drops an extra c and an extra b and changes direction.

$$\begin{array}{rcl} S & \to & L\,1\,R \\ L & \to & L\,D \\ D\,1 & \to & 1\,1\,D \\ D\,R & \to & R \\ L & \to & \varepsilon \\ R & \to & \varepsilon \end{array}$$

This grammar for the language $\{1^{2^n} \mid n \in \mathbb{N}\}$ suffers from both a large number of workers (the D-s), and the disappearing end-markers. If an end-marker (an L or an R) is destroyed prematurely, no string can be produced at all. Unfortuneately, this is not discovered immediately, but rather after trying a lot of unsuccessful production paths. A more efficient grammar is the admittedly more complex:

$$\begin{array}{rcl} S & \to & M\,R\,1\,M \ \mid \ M\,1\,R\,M \\ R\,1 & \to & 1\,1\,R \\ 1\,L & \to & L\,1\,1 \\ R\,M & \to & L\,M \ \mid \ E \\ 1\,E & \to & E\,1 \\ M\,E & \to & \varepsilon \\ M\,L & \to & M\,R \end{array}$$

Again, we use a single worker running multiple passes instead of using multiple workers. Also, notice the manner in which the end-markers (M) are disposed of, by replacing both the right-going worker (R) and the right end-marker (M) with an eraser worker (E) which travels to the left end-marker, destroys it and itself, and terminates.

Advanced Topic II – Custom Evaluators For LR(1) Parsers

This section describes the requirements for custom procedures needed to extend AAAAA LR(1) parsers with user actions. It is primarily aimed at those teachers who wishes to create other environments than expression evaluation, or customize the existing one, and thereby creating new exercises for students. It may also be of interest for the student who wishes to move one step closer to Yacc [2] and knows some Tcl [3, 4].

To benefit from this section, the reader should have a working knowledge of how to write LR(1) parsers in AAAAA, some knowledge of bottom-up parsing [1, 2], and a working knowledge of Tcl [3, 4].

Overview of Evaluation

The AAAAA LR(1) parser parses the entire text using the grammar. Then, if no errors occurred, it re-traces its execution in two additional passes. In the first pass, it simulates ordinary reductions by passing them to a procedure specified by the user, propagating user specified data upwards in the parse tree. In the second pass it asks the user to extract some printable information from the computation. The advantage of this multi-pass technique is that very flexible computation can be performed if desired, and it is only moderatly more complex than a straight one-pass parse. (Efficiency is of course not a prime concern for a teaching tool like AAAAA.)

The Evaluation Procedure

The procedure named by the "Eval.proc." field in an AAAAA LR(1) parser is written in Tcl [3, 4] and is called at various points during the parse. The first argument given to it is an identifier to separate the parsers from each other. You may use it as a global (by uplevel-ing it). The second argument is a command, and the other arguments depend on the command. In each synopsis given below, the procedure name is denoted *proc* and the identifier is denoted *id*.

The initialization is made by `clear`, usually called only once in the lifetime of a parser (but also when user clicks "Reinitialize"), and cleanup is performed by `destroy`. Other than that, the following steps are followed:

1. Call `new`.
2. Call `token` and `reduce` in the order corresponding to shifts and reductions. Repeated some number of times.
3. Call `accept`.
4. Call `format` some number of times, corresponding to shifts and reductions.

Call Synopsis

proc id `accept` *value*

Informs the user that *value* was the result of the parse. May perform some extra processing, and return a modified value, or just return *value*.

proc id `clear`

Called when parser is created (including by "Update"), and when the user clicks "Reinitialize". May be a no-op unless there is some state kept between inputs, in which case it should clear it.

proc id `destroy`

Called when parser is destroyed (clicking either "Update" or "Dismiss", or starting a new session). May be used to free memory (unset) or the like.

proc id `format` *headsymbol headval*

After the string has reached accept, the tree is traversed a second time in the same order as the first time. This time the string displayed in show values should be returned, given the value computed by the reduce command.

proc id `new`

Executed each time the parser is created or the user clicks "Update" on the parser. May be a no-op unless there is some state to initialize, in which case it should initialize it.

proc id `reduce` *headsymbol ?sym val...?*

Executed each time the parser performs a reduction. (Well, actually not, but you should think of it that way anyway.) The *headsymbol* is the non-terminal it reduces to (the LHS of the production rule), and the rest of the arguments corresponds to the RHS of the production rule, giving the grammar symbol as *sym*, and its value as *val*. Should return the new value for *headsymbol*.

proc id `token` *tokentype lexeme*

Executed for each shifted symbol. The *tokentype* value is the regular expression that matched (note that if the expression was not enclosed in braces AAAAA may add such and possibly alter the expression in other ways), and the *lexeme* value is the string which matched. Should return the value for the token.

The Standard Evaluator

In AAAAA version 11, the standard evaluator looks like this:

```
proc expr_eval {id cmd args} {
    switch -- $cmd {
        accept {
            return [lindex $args 0]
        }
        destroy -
        new -
        clear {
            return
        }
        token -
        format {
            return [lindex $args 1]
        }
        reduce {
            set e {}
            set rest [lrange $args 1 end]
            foreach {sym val} $rest {
                lappend e $val
            }
            if {[catch {set res [expr $e]}]} {
                set res "?"
            }
            return $res
        }
    }
}
```

Restrictions on Tcl Names

The procedure and variable names that may used are restricted, since AAAAA runs on pre-namespace Tcl versions, such as Tcl 7.6. Therefore, there is a potential for clashes between AAAAA procedures and variables and those supplied by the user.

AAAAA reserves all names which starts with `a5_`, and also the global variables `a`, `s`, and `dummy`. All other globals (except for Tcl/Tk globals) are free to be used in user code.

AAAAA will not use any procedures which begin with upper case letters nor any procedure names which has a letter as its first character and an underscore as it second character. It is recommended that user code use some of these prefixes for their procedures. In addition, the procedure `expr_eval` may be overridden by defining a new one in user code.

Adding Tcl Code to AAAAA

There are two ways to add Tcl code to AAAAA. During development and testing, the preferred way is to use the "Source Tcl Code" command on the "File" menu. This allows changes to the evaluator without restarting AAAAA. The other way is to use the `-source` option when starting AAAAA.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] S. C. Johnson. *Yacc – yet another compiler compiler.* Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J.

[3] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley.

[4] Online information on Tcl/Tk: `http://www.sunscript.sun.com/`, `http://www.tclconsortium.org/`, and `http://www.scriptics.com/`.