Aflex – An Ada Lexical Analyzer Generator

Version 1.1

John Self

Arcadia Environment Research Project
Department of Information and Computer Science
University of California, Irvine

UCI-90-18 *

May 1990

*This work was supported in part by the National Science Foundation under grants CCR–8704311 and CCR–8451421 with cooperation from the Defense Advanced Research Projects Agency, and by the National Science Foundation under Award No. CCR-8521398.

Contents

1 Introduction **1**

2 Command Line Options **2**

3 *Aflex* Output **4**

4 Regular Expressions **5**
4.1 Predefined Variables & Routines 8

5 *Aflex* Source Specification **10**
5.1 Definitions Section 10
5.1.1 Macros 10
5.1.2 Start Conditions 11
5.2 Rules Section 12
5.3 User Defined Section 12

6 Ambiguous Source Rules **12**

7 *Aflex* and *Ayacc* **13**

8 Appendix A: A Detailed Example **14**

9 Appendix B: *Aflex* Dependencies **18**
9.1 Command Line Interface 18

10 Appendix C: Differences between *Aflex* and *Lex* **19**

11 Appendix D: Differences between *Aflex* and *Alex* **19**

12 Appendix E: Known Bugs and Limitations **21**

1 Introduction

Aflex is a lexical analyzer generating tool written in Ada designed for lexical processing of character input streams. It is a successor to the *Alex*[NF88] tool from UCI. *Aflex* is upwardly compatible with *alex 1.0*, but is significantly faster at generating scanners, and produces smaller scanners for equivalent specifications. Internally *aflex* is patterned after the *flex* tool from the GNU project. *Aflex* accepts high level rules written in regular expressions for character string matching, and generates Ada source code comprising a lexical analyzer along with two auxiliary Ada packages. The main file includes a routine that partitions the input text stream into strings matching the expressions. Associated with each rule is an action block composed of program fragments. Whenever a rule is recognized in the input stream, the corresponding program fragment is executed. This feature, combined with the powerful string pattern matching capability, allows the user to implement a lexical analyzer for any type of application efficiently and quickly. For instance, *aflex* can be used alone for simple lexical analysis and statistics, or with *ayacc* [TT86] to generate a parser front-end. *Ayacc* is an Ada parser generator that accepts context-free grammars.

Aflex is a successor to the Arcadia tool *Alex*[NF88] which was inspired by the popular Unix operating system tool, *lex* [Les75],. Consequently, most of *lex*'s features and conventions are retained in *aflex*; however, a few important differences are discussed in section 10. There are also a few minor differences between *aflex* and *alex* which will be discussed in section 11.

This paper is intended to serve as both the reference manual and the user manual for *aflex*. Some knowledge of *lex*, while not required, would be very useful in understanding the use of*aflex*. A good introduction to *lex*, as well as lexical and syntactic analysis, can be found in [ASU86], frequently referred to as "the Dragon Book." Topics to be covered in this paper include the usage of *aflex*, the operators' description, the source file format, the generated output, the necessary interfaces with *ayacc*, and ambiguity among rules. The appendices provide a simple example, *aflex* dependencies, the differences between *aflex*,*alex*, and *lex*, known bugs and limitations, and references.

2 Command Line Options

Command line options are given in a different format than in the old UCI alex. Aflex options are as follows

-t Write the scanner output to the standard output rather than to a file. The default name of the scanner file for base.l is base.a Note that this option is not as useful with aflex because in addition to the scanner file there are files for the externally visible dfa functions (base_dfa.a) and the external IO functions (base_io.a)

-b Generate backtracking information to aflex.backtrack. This is a list of scanner states which require backtracking and the input characters on which they do so. By adding rules one can remove backtracking states. If all backtracking states are eliminated and **-f** is used, the generated scanner will run faster (see the **-p** flag). Only users who wish to squeeze every last cycle out of their scanners need worry about this option.

-d makes the generated scanner run in *debug* mode. Whenever a pattern is recognized the scanner will write to *stderr* a line of the form:

```
--accepting rule #n
```

Rules are numbered sequentially with the first one being 1. Rule #0 is executed when the scanner backtracks; Rule #(n+1) (where n is the number of rules) indicates the default action; Rule #(n+2) indicates that the input buffer is empty and needs to be refilled and then the scan restarted. Rules beyond (n+2) are end-of-file actions.

-f has the same effect as lex's -f flag (do not compress the scanner tables); the mnemonic changes from *fast compilation* to (take your pick) *full table* or *fast scanner.* The actual compilation takes *longer,* since aflex is I/O bound writing out the big table. The compilation of the Ada file containing the scanner is also likely to take a long time because of the large arrays generated.

-i instructs aflex to generate a *case-insensitive* scanner. The case of letters given in the aflex input patterns will be ignored, and the rules will be matched regardless of case. The matched text given in *yytext* will have the preserved case (i.e., it will not be folded).

-p generates a performance report to stderr. The report consists of comments regarding features of the aflex input file which will cause a loss of performance in the resulting scanner. Note that the use of the ^ operator and the **-I** flag entail minor performance penalties.

-s causes the *default rule* (that unmatched scanner input is echoed to *stdout)* to be suppressed. If the scanner encounters input that does not match any of its rules, it aborts with an error. This option is useful for finding holes in a scanner's rule set.

-v has the same meaning as for lex (print to *stderr* a summary of statistics of the generated scanner). Many more statistics are printed, though, and the summary spans several lines. Most of the statistics are meaningless to the casual aflex user, but the first line identifies the version of aflex, which is useful for figuring out where you stand with respect to patches and new releases.

-E instructs aflex to generate additional information about each token, including line and column numbers. This is needed for the advanced automatic error option correction in ayacc.

-I instructs aflex to generate an *interactive* scanner. Normally, scanners generated by aflex always look ahead one character before deciding that a rule has been matched. At the cost of some scanning overhead, aflex will generate a scanner which only looks ahead when needed. Such scanners are called *interactive* because if you want to write a scanner for an interactive system such as a command shell, you will probably want the user's input to be terminated with a newline, and without **-I** the user will have to type a character in addition to the newline in order to have the newline recognized. This leads to dreadful interactive performance.

If all this seems to confusing, here's the general rule: if a human will be typing in input to your scanner, use **-I,** otherwise don't; if you don't care about how fast your scanners run and don't want to make any assumptions about the input to your scanner, always use **-I.**

Note, **-I** cannot be used in conjunction with *full* i.e., the **-f** flag.

-L instructs aflex to not generate **#line** directives (see below).

-T makes aflex run in *trace* mode. It will generate a lot of messages to stdout concerning the form of the input and the resultant non-deterministic and deterministic finite automatons. This option is mostly for use in maintaining aflex.

-Sskeleton_file overrides the default internal skeleton from which aflex constructs its scanners. You'll probably never need this option unless you are doing aflex maintenance or development.

3 *Aflex* Output

Aflex generates a file containing a lexical analyzer function along with two auxiliary packages, all of which are written in Ada. The context in which the lexical analyzer function is defined is flexible and may be specified by the user. For instance, the file may only contain the lexical analyzer function as a single compilation unit which may be called by *ayacc*, or it may be placed within a package body or embedded within a driver routine. This scanner function, when invoked, partitions the character stream into tokens as specified by the regular expressions defined in the rules section of the source file. The name of the lexical analyzer function is *yylex*. Note that it returns values of type *token*. Type *token* must be defined as an enumeration type which contains, at a minimum, (*End_of_Input, Error*). It is up to the user to make sure that this type is visible (see Section 7). The general format of the output file which contains this function is found in Figure 3.

The auxiliary packages include a DFA and an IO package. The DFA package contains externally visible functions and variables from the scanner. Many of the variables in this package should not be modified by normal user programs, but they are provided here to allow the user to modify the internal behavior of aflex to match specific needs. Only the functions YYText and YYLength will be needed by most programs.

The IO package contains routines which allow *yylex* to scan the input source file. These include the unput, input, output, and yywrap functions from *lex*,
plus Open_Input, Create_Output, Close_Input and Close_Output provided for compatibility with *alex*. It is also possible to write your own IO and DFA packages.

Redefining input is possible by changing the YY_INPUT procedure. As an example you might wish to take input from an array instead of from a file. By changing the calls to the TEXT_IO routines to access elements of the array you can change the input strategy. If you change the IO or DFA packages you should make a copy of the generated files under a different name and change that, because *aflex* will overwrite them whenever you rerun *aflex*.

```
with <rootname>_DFA;
with <rootname>_IO;
with TEXT_IO;

-- User Specified Context

    function yylex return Token is
    begin
        -- Analysis of expressions
        -- Execution of user-defined actions
    end yylex;

-- User Specified Context
```

Figure 3: Example of File Containing Lexical Analyzer

Before showing the general layout of the specification file, we will describe the specification language of *aflex*, namely, regular expressions.

4 Regular Expressions

Aflex distinguishes two types of character sets used to define regular expressions: text characters and operator characters. A regular expression specifies how a set of strings from the input string can be recognized. It contains text characters (which match the corresponding characters in the strings being compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters.

A rule specifies a sequence of characters to be matched. It **must** begin in column one. The set of *aflex* operators consists of the following:

```
" \ { } [ ] ^ $ < > ? . * + | ( ) /
```

The meaning of each operator is summarized below:

```
x       -- the character "x"
"x"     -- an "x", even if x is an operator.
\x      -- an "x", even if x is an operator.
^x      -- an x at the beginning of a line.
x$      -- an x at the end of line.
x+      -- 1 or more instances of x.
```

```
x*      -- 0 or more instances of x.
x?      -- an optional x.
(x)     -- an x.
.       -- any character but newline.
x|y     -- an x or y.
[xy]    -- the character x or the character y.
[x-z]   -- the character x, y or z.
[^x]    -- any character but x.
<y>x    -- an x when aflex is in start condition y.
{xx}    -- the translation of xx from the definitions section.
```

If any of these operators is used in a regular expression as a character literal, it must be either preceded by an escape character or surrounded by double quotes. For example, to recognize a dollar sign `$`, the correct expression is either `\$` or `"$"`. Note a quote cannot be quoted and should therefore be escaped.

A regular expression may **not** contain any spaces unless they are within in a quoted string or character class or they are preceded by the `"\"` operator.

When in doubt, use parentheses. When an *aflex* operator needs to be embedded in a string, it is often neater to quote the entire string rather than just the operator, e.g. the string `"what?"` is more readable than both `What"?"`, and `What\?`.

```
Rules              Interpretations
-----              ---------------
a or "a"           The character a
Begin or "Begin"   The string Begin
\"Begin\"          The string "Begin"
^\t or ^"\t"       The tab character \t at the beginning of line.
\n$                The newline character \n at the end of line.
```

There are a few special characters which can be specified in a regular expression:

```
\n      -- newline
\b      -- backspace
\t      -- tab
\r      -- carriage return
\f      -- form feed
\ddd    -- octal ASCII code
```

Here is the precedence of the above operators that have precedence.

" [] ()	Highest
+ * ?	⋮
concatenation	⋮
|	Lowest

Character Classes: Classes of characters can be specified using the operator pair []. Within these square brackets, the operator meanings are ignored except for three special characters: \ and − and ^.

Rules	Interpretations
[^abc]	Any character except a, b, or c.
[abc]	The single character a, b, or c.
[-+0-9]	The - or + sign or any digit from 0 to 9.
[\t\n\b]	The tab, newline, or backspace character.

Arbitrary and Optional Characters: The dot, ".", operator matches all characters except newline. The operator ? indicates an optional character of an expression.

Rules	Interpretations
ab?c	Matches either abc or ac.
ab.c	Matches all strings of length 4 having a, b and c as the first, second and fourth letter where the third character is not a newline.

Repeated Expressions: Repetitions of classes are indicated by the operators ∗ and +.

Rules	Interpretations
[a-z]+	Matches all strings of lower case letters.
[A-Za-z][A-Za-z0-9]*	Indicates all alphanumeric strings with a leading alphabetic character.

Alternation and Grouping: The operator | indicates alternation and parentheses are used for grouping complex expressions.

```
Rules           Interpretations
-----           ---------------
ab|cd           Matches either ab or cd.
(ab|cd+)?(ef)*  Matches such strings as abefef, efefef, cdef,
                or cddd; but not abc, abcd, or abcdef.
```

Context Sensitivity: *aflex* will recognize a small amount of surrounding context. Two simple operators for this are ^ and $. If the first character of an expression is ^, the expression will only be matched at the beginning of a line. If the very last character is $, the expression will only be matched at the end of a line.

```
Rules          Interpretations
-----          ---------------
^ab            Matches ab at the beginning of line.
ab$            Matches ab at the end of line.
```

Definitions: The operators { } enclosing a name specify a macro definition expansion.

```
Rules          Interpretations
-----          ---------------
{INTEGER}      If INTEGER is defined in the macro definition
               section, then it will be expanded here.
```

4.1 Predefined Variables & Routines

Once a token is matched, the textual string representation of the token may be obtained by a call to the function *yytext* which is located in the *dfa package*. This function returns type string.

The IO package contains routines which allow *yylex* to scan the input source file. These include the input, output, unput and yywrap functions from lex,
plus Open_Input, Create_Output, Close_Input and Close_Output provided for compatibility with *alex*. Note that in *alex 1.0* it was mandatory to call the *Open_Input* and *Create_Output* routines before calling *YYLex.* This is not required in *Aflex.* The default input and output are attached to the files that Ada considers to be the STANDARD_INPUT and STANDARD_OUTPUT.
The following routines must be used in lieu of the normal TEXT_IO routines because of internal buffering and read-ahead done by *aflex.*

input function input return character – inputs a character from the current *aflex* input stream.

unput procedure unput(c : character) – returns a character already read by input to the input stream. Note that attempting to push back more than one character at a time can cause *aflex* to
raise the exception PUSHBACK_OVERFLOW.

output procedure output(c : character) – outputs a character to the current *aflex* output stream.

yywrap function yywrap return boolean – This function is called when *aflex* reaches the end of file. If *yywrap* returns true, *aflex* continues with normal wrapup at end of input. If you wish to arrange for more input to arrive from a new source then you provide a yywrap which returns false. The default yywrap return true.

Open_Input Open_Input(fname : in String) – Uses the file named fname as the source for input to *YYLex.* If this function is not called then the default input is the Ada STANDARD_INPUT.

Open_Input Create_Output(fname : in String) – Uses the file named fname as output for *YYLex.* If this function is not called then the default output is the Ada STANDARD_OUTPUT.

Close_Input and Close_Output These functions have null bodies in *aflex* and are provided only for compatibility with *alex.*

There are a few predefined subroutines that may be used once a token is matched. In many lexical processing applications, the printing of the string returned by *yytext*, i.e. `put(yytext)`, is desired and this action is so common that it may be written as `ECHO`.

5 *Aflex* Source Specification

The general format of the source file is

```
definitions section
%%
rules section
%%
user defined section
\#\#
user defined section
```

where `%%` is used as a delimiter between sections and `##` indicates where function *yylex* will be placed. Both `%%` and `##` *must* occur in column one.

The definitions section is used to define macros which appear in the rules section and also to define start conditions. The rules section defines the regular expressions with their corresponding actions. These regular expressions, in turn, define the tokens to be identified by the scanner. The user defined section allows the user to define the context in which the *yylex* function will be located. The user can include routines which may be executed when a certain token or condition is recognized.

5.1 Definitions Section

The definitions section may contain both macro definitions and start condition definitions. Macro and start condition definitions must begin in column one and may be interspersed.

5.1.1 Macros

Macro definitions take the form:

```
name   expression
```

where `name` must begin with a letter and contain only letters, digits and underscores, and `expression` is any string of characters that `name` will be textually substituted to if found in the rule section. At least one space must separate `name` from `expression` in the definition. No syntax checking is done in the expression, instead the whole rule is parsed after expansion. The macro facility is very useful in writing regular expressions which have common substrings, and in defining often-used ranges like *digit* and *letter*. Perhaps its best advantage is to give a mnemonic name to a rather

strange regular expression – making it easier for the programmer to debug the expressions. These macros, once defined, can be used in the regular expression by surrounding them with { and }, e.g., `{DIGIT}`. For example, the rule

```
[a-zA-Z]([0-9a-zA-Z])*   {put_line ("Found an identifier");}
[0-9]+                   {put_line ("Found a number");}
```

defines identifiers and integer numbers. With macros, the source file is

```
LETTER [a-zA-Z]
DIGIT  [0-9]
%%
{LETTER}({DIGIT}|{LETTER})*   {put_line ("Found an identifier");}
{DIGIT}+                      {put_line ("Found a number");}
```

It is customary, although not necessary, to use all capital letters for macro names. This allows macros to be easily identified in complex rules. Macro names are case sensitive, e.g., `{DIGIT}` and `{Digit}` are two different macro names.

5.1.2 Start Conditions

Left context is handled in *aflex* by start conditions that are defined in the macro definition section. Start conditions are declared as follows,

```
%Start cond1 cond2 ...
```

where cond1 and cond2 indicate start conditions. Note that %Start may be abbreviated as %S or %s.

A condition is set only when the *aflex* command `ENTER` in the action part is executed, e.g. `ENTER cond1;`. Thus the expression which has the form `<condition>rule` will only be matched when `condition` is set. Note that *aflex* uses `ENTER` instead of `BEGIN` which is used in *lex.* This is done because `BEGIN` is a keyword in Ada. The `ENTER` command must have parentheses surrounding its argument.

```
ENTER(cond1);
```

Aflex also provides *exclusive start conditions.* These are similar to normal start conditions except they have the property that when they are active no other rules are active. Exclusive start conditions are declared and used like normal start conditions except that the declaration is done with %x instead of %s.

5.2 Rules Section

Contained in the rule section are regular expressions which define the format of each token to be recognized by the scanner. Each rule has the following format:

```
pattern  {action}
```

where `pattern` is a regular expression and `action` is an Ada code fragment enclosed between { and }. A `pattern` must always begin in column one.

While a pattern defines the format of the token, the action portion defines the operation to be performed by the scanner each time the corresponding token is recognized. Therefore, the user must provide a syntactically correct Ada code fragment. *aflex* does not check for the validity of the program portion, but rather copies it to the output package and leaves it to the Ada compiler to detect syntax and semantics errors. There can be more than one Ada statement in the code fragment. For example, the rule

```
%%
begin|BEGIN     {copy (yytext, buffer);
                 Install (yytext,symbol_table);
                 return RESERVED;}
```

recognizes the reserved word "begin" or "BEGIN", copies the token string into the buffer, inserts it in the symbol table and returns the value, RESERVED.
Note that the user must provide the procedures `copy` and `install` along with all necessary types and variables in the user defined section.

5.3 User Defined Section

The user defined section allows the user to specify the context surrounding the *yylex* function. `##` is used to indicate where the *yylex* function should be placed. It must be present in this section and must occur in the first column. Any text following `##` on the same line is ignored.

6 Ambiguous Source Rules

When a set of regular expressions is ambiguous, *aflex* uses the following rules to choose among the regular expressions that match the input.

1. The longest string is matched.

2. If the strings are of the same length, the rule given **first** is matched.

For example, if input `"aabb"` matches both `"a*"` and `"aab*"` the action associated with `"aab*"` is executed because it matches four as opposed to two characters.

7 *Aflex* and *Ayacc*

As briefly mentioned in Section 1, *aflex* can be integrated with *ayacc* to produce a parser.

Since the parser generated by *ayacc* expects a value of type *token*, each *aflex* rule should end with

```
return (token_val);
```

to return the appropriate token value. *Ayacc* creates a package defining this token type from its specification file, which in turn should be *with*'ed at the beginning of the user defined section. Thus, this token package must be compiled before the lexical analyzer. The user is encouraged to read the Ayacc User Manual [TT86] for more information on the interaction between *aflex* and *ayacc*.

8 Appendix A: A Detailed Example

This section shows a complete *aflex* specification file for translating all characters to uppercase. The following file, *example.l*, defines rules for recognizing lowercase and uppercase words. If a word is in lowercase, the scanner converts it to uppercase. In addition, the frequencies of lower and uppercase words are retained in the two variables defined in the global section. All other characters (spaces, tabs, punctuation) remain the same.

```
LOWER    [a-z]
UPPER    [A-Z]

%%

{LOWER}+       { Lower_Case := Lower_Case + 1;
                 TEXT_IO.PUT(To_Upper_Case(Example_DFA.YYText)); }

  -- convert all alphabetic words in lower case
  -- to upper case

{UPPER}+       { Upper_Case := Upper_Case + 1;
                 TEXT_IO.PUT(Example_DFA.YYText); }

  -- write uppercase word as is

\n             { TEXT_IO.NEW_LINE;}

.              { TEXT_IO.PUT(Example_DFA.YYText); }
                 -- write anything else as is

%%
with U_Env;  -- VADS environment package for UNIX
procedure Example is

  type Token is (End_of_Input, Error);

  Tok        : Token;
  Lower_Case : NATURAL := 0;   -- frequency of lower case words
  Upper_Case : NATURAL := 0;   -- frequency of upper case words

  function To_Upper_Case (Word : STRING) return STRING is
  Temp : STRING(1..Word'LENGTH);
```

```
  begin
    for i in 1.. Word'LENGTH loop
      Temp(i) := CHARACTER'VAL(CHARACTER'POS(Word(i)) - 32);
    end loop;
    return Temp;
  end To_Upper_Case;


-- function YYlex will go here!!
##


begin  -- Example

  Example_IO.Open_Input     (U_Env.argv(1).s);

  Read_Input :
  loop
    Tok := YYLex;
    exit Read_Input
      when Tok = End_of_Input;
  end loop Read_Input;

  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT_LINE("Number of lowercase words is => " &
   INTEGER'IMAGE(Lower_Case));
  TEXT_IO.PUT_LINE("Number of uppercase words is => " &
   INTEGER'IMAGE(Upper_Case));
end Example;
```

This source file is run through *aflex* using the command

```
% aflex example.l
```

aflex produces an output file called *example.a* along with two packages, *example_dfa.a* and *example_io.a*. Assuming that the main procedure, *Example*, is used to construct an object file called *example.out*, the Unix command

```
% example.out example.l
```

prints to the screen the exact file *example.l* with letters in uppercase, i.e. the output to the screen is

```
LOWER    [A-Z]
UPPER    [A-Z]

%%

{LOWER}+       { LOWER_CASE := LOWER_CASE + 1;
                 TEXT_IO.PUT(TO_UPPER_CASE(EXAMPLE_DFA.YYTEXT)); }

  -- CONVERT ALL ALPHABETIC WORDS IN LOWER CASE
  -- TO UPPER CASE

{UPPER}+       { UPPER_CASE := UPPER_CASE + 1;
                 TEXT_IO.PUT(EXAMPLE_DFA.YYTEXT); }

  -- WRITE UPPERCASE WORD AS IS

\N             { TEXT_IO.NEW_LINE;}

.              { TEXT_IO.PUT(EXAMPLE_DFA.YYTEXT); }
                 -- WRITE ANYTHING ELSE AS IS

%%
WITH U_ENV;  -- VADS ENVIRONMENT PACKAGE FOR UNIX
PROCEDURE EXAMPLE IS

  TYPE TOKEN IS (END_OF_INPUT, ERROR);

  TOK        : TOKEN;
  LOWER_CASE : NATURAL := 0;   -- FREQUENCY OF LOWER CASE WORDS
  UPPER_CASE : NATURAL := 0;   -- FREQUENCY OF UPPER CASE WORDS

  FUNCTION TO_UPPER_CASE (WORD : STRING) RETURN STRING IS
  TEMP : STRING(1..WORD'LENGTH);
  BEGIN
    FOR I IN 1.. WORD'LENGTH LOOP
      TEMP(I) := CHARACTER'VAL(CHARACTER'POS(WORD(I)) - 32);
    END LOOP;
    RETURN TEMP;
  END TO_UPPER_CASE;


-- FUNCTION YYLEX WILL GO HERE!!
```

```
##


BEGIN  -- EXAMPLE

  EXAMPLE_IO.OPEN_INPUT     (U_ENV.ARGV(1).S);

  READ_INPUT :
  LOOP
    TOK := YYLEX;
    EXIT READ_INPUT
      WHEN TOK = END_OF_INPUT;
  END LOOP READ_INPUT;

  TEXT_IO.NEW_LINE;
  TEXT_IO.PUT_LINE("NUMBER OF LOWERCASE WORDS IS => " &
   INTEGER'IMAGE(LOWER_CASE));
  TEXT_IO.PUT_LINE("NUMBER OF UPPERCASE WORDS IS => " &
   INTEGER'IMAGE(UPPER_CASE));
END EXAMPLE;
Number of lowercase words is =>  144
Number of uppercase words is =>  120
```

9 Appendix B: *Aflex* Dependencies

This release of *aflex* was successfully compiled by VADS 5.5 and Telesoft 1.3a.01 running under Sun Unix 4.0.3. Other machines/systems may support *aflex* but have not been tested.

9.1 Command Line Interface

The following files are host dependent :

command_lineS.a
command_lineB.a
file_managerS.a
file_managerB.a

The command_line package function INITIALIZE_COMMAND_LINE breaks up the command line into a vector containing the arguments passed to the program. Note that modifications may need to be made to this file if the host system doesn't allow differentiation of upper and lower case on the command line. The external_file_manager

package is host dependent in that it chooses the names and suffixes for the generated files. It also sets up the file_type STANDARD_ERROR to allow error output to appear on the screen.

If *aflex* is to be rehosted, only these files should need modification. For more detailed information see the file PORTING in the *aflex* distribution.

10 Appendix C: Differences between *Aflex* and *Lex*

Although *aflex* supports most of the conventions and features of *lex*, there are some differences that the user should be aware of in order to port a *lex* specification to an *aflex* specification.

- Source file's format:

```
definitions section
%%
rules section
%%
user defined section
## -- places yylex function
user defined section
```

- Although *aflex* supports most *lex*'s constructs, it does not implement the following features of *lex*.

  – REJECT
  – %x — changes to the internal array sizes, but see below.

- Ada style comments are supported instead of C style comments.
- All template files are internalized.
- The input source file name must end with a ".l" extension.
- In start conditions ENTER is used instead of BEGIN. This is done because BEGIN is a keyword in Ada.

11 Appendix D: Differences between *Aflex* and *Alex*

While *aflex* is intended to be upwardly compatible with *Alex*, there are a few minor differences. Any major inconsistencies with *alex* should be considered bugs and reported.

- The `ENTER` calls must have parentheses around their arguments. Parentheses were optional in *alex*.

- It is no longer mandatory to call Open_Input and Create_Output before calling YYLex. Previously if output was to be directed to Standard_Output it was recommended that a call of

  ```
  Create_Output("/dev/tty");
  ```

  be made. This will still work but because of differences in implementation this may cause difficulties in redirecting output using the UNIX shell pipes and redirection. Instead just don't call Open_Input and output will go to the default STANDARD_OUTPUT.

- Compilation order. In previous versions of *alex* the DFA and IO packages could be compiled in any order. With *aflex* it is necessary to compile the DFA package first, because it contains declarations used by the IO package.

12 Appendix E: Known Bugs and Limitations

- Some trailing context patterns cannot be properly matched and generate warning messages ("Dangerous trailing context"). These are patterns where the ending of the first part of the rule matches the beginning of the second part, such as "zx*/xy*", where the 'x*' matches the 'x' at the beginning of the trailing context. (Lex doesn't get these patterns right either.)

- *variable* trailing context (where both the leading and trailing parts do not have a fixed length) entails a substantial performance loss.

- For some trailing context rules, parts which are actually fixed-length are not recognized as such, leading to the abovementioned performance loss. In particular, parts using '—' or n are always considered variable-length.

- Nulls are not allowed in aflex inputs or in the inputs to scanners generated by aflex. Their presence generates fatal errors.

- Pushing back definitions enclosed in ()'s can
  result in nasty, difficult-to-understand problems like:

```
{DIG}  [0-9] -- a digit
```

  In which the pushed-back text is "([0-9] – a digit)".

- Due to both buffering of input and read-ahead, you cannot intermix calls to text_io routines, such as, for example, **text_io.get()** with aflex rules and expect it to work. Call **input()** instead.

- There are still more features that could be implemented (especially REJECT.) Also the speed of the compressed scanners could be improved.

- The utility needs more complete documentation, especially more information on modifying the internals.

References

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Les75] M. E. Lesk. Lex – a lexical analyzer generator. Technical Report Computing Science Technical Report, 39, Bell Telephone Laboratories, Murray Hill, NJ, 1975.

[NF88] T. Nguyen and K. Forester. Alex – an ada lexical analysis generator. Arcadia Document UCI-88-17, University of California, Irvine, 1988.

[TT86] D. Taback and D. Tolani. Ayacc user's manual. Arcadia Document UCI-85-10, University of California, Irvine, 1986.