`reflex`: A Scanner Transformer for Unicode Grammars

S. L. Huerter, S. M. Watt

July 31, 2006

Abstract

Conventional table-driven scanners and the tools to generate them evolved alongside conventional programming languages with small character sets. Lately the Universal Character Set has swept up-to-date programming languages ahead of their traditional lexical analyzers: its imposing size of more than one million characters far exceeds practical implementation limits for table-based scanners. We address this issue by transforming lexical grammars over Unicode to lexical grammars over smaller alphabets; regular expressions used by the generated scanners are those in which Unicode scalars have been expanded to corresponding code sequences under a chosen Unicode encoding scheme. We introduce an original tool, `reflex`, to do this translation and which may be applied to scanner specifications in the Flex and Jflex languages.

1 Introduction

Until recently, lexical structures of programming languages have involved only small character sets. It was therefore practical to recognize a given language's token set by using a scanner based on the direct construction of the corresponding table-driven automaton. Scanner-generating programs ([2], [4], [5],etc.) were developed to automate this commonplace construction, providing language designers with all the benefits of high-level declarative token-language specification: compactness, rapid development, simplified maintenance and extension, and re-use.

The onset of multi-lingual data-processing has led newer programming languages, and updated releases of older ones, to embrace the trend of internationalization. Characters from any script worldwide, even from different time periods and fictitious contexts are permitted within tokens such as identifiers and string literals. This superset of characters, known as the Universal Character Set, is represented by a language-independent 21-bit encoding called Unicode([6]). Unicode assigns a unique name and integer 'code point' to each abstract character, and provides for more than 1.1 million character definitions.

Unicode support is lagging in scanner-generating tools. Such tools, originally supporting character sets encodable using 7 or 8 bits, have at best only provided upgrades to 16-bit sets. 16-bit input character sets represent the limit of practical implementation for table-driven scanners, since effectively the input-alphabet size determines one dimension of the transition table for the underlying automaton. A 16-bit encoding is all that is needed to represent the most commonly used characters of Unicode, referred to as the 'Basic Multilingual Plane' (BMP). However many languages permit within their tokens the more obscure characters from other Unicode planes. So it seems that for the benefit of large numbers of uncommonly used characters, language implementors must give up the advantages of high-level token language specification and once again become mired in the details of manual scanner development.

However, we are not limited to an indivisible 21-bit wide representation of each Unicode code point. Rather, each code point can be serialized into a sequence of smaller pieces by using a suitable encoding scheme. Common encoding schemes for Unicode such as UTF-8 and UTF-16 (see Chapter 3 of [6]), provide a higher-resolution definition for each code point as a sequence of smaller atomic units such as bytes or 16-bit words. These encoding schemes serve as file formats for source files over Unicode.

It is therefore conceivable that an 8-bit scanner, capable of consuming an encoded file byte-by-byte, could recognize all Unicode code points by recognizing their corresponding 8-bit code sequences. Accordingly, patterns over Unicode code points would be matched by instead matching the equivalent pattern over encoded code points. The effect of this approach is to shift the weight of Unicode over to the other dimension of the transition table, the size of the state-set. The impact on state-set size of this straightforward substitution of code-unit sequences for Unicode code points will vary between token-languages, and in the average case is not expected to exceed the limitations of implementation.

In this report we describe a tool, `reflex`, that introduces full Unicode support to the Flex and JFlex scanner-generators. `reflex` translates extended-syntax scanner specifications, written over 21-bit Unicode code points, to equivalent scanner specifications over 8 or 16-bit encoded patterns ([2],[3]). We begin by describing how to use `reflex`, and then give some detailed examples.

2 reflex

`reflex` is a scanner-specification transformation tool, designed to transform patterns over the Unicode input alphabet in Flex (JFlex) scanner specifications into one of a selection of common Unicode encoding schemes. The result is an equivalent Flex (JFlex) scanner specification over a smaller input alphabet. In this section, we describe how to use `reflex` to transform Flex (JFlex) files, and give an overview of what the tool does.

2.1 Running reflex

Once the file `lext.jar` is included in your CLASSPATH environment variable, `reflex` is run using the form:

```
java lext.Main <options> <infile> <outfile>?
```

The output file is an optional argument: if it is not given then the result is printed to standard output. The input file is a Flex or JFlex scanner-specification file.

The options for `reflex` are:

`-s<lang>`
the source language is given by `<lang>` which is either `jflex` or `flex`;

`-e<encoding>`
the target encoding is given by `<encoding>`. One encoding supports the BMP, and the remaining encodings support full 21-bit Unicode:

- `dbyte`
  Double-byte encoding (16-bit Unicode);
- `utf8`
  UTF-8 encoding (21-bit Unicode);
- `utf16be`
  UTF-16 encoding (21-bit Unicode), with big-endian byte-order;
- `utf16le`
  UTF-16 encoding (21-bit Unicode), with little-endian byte-order;
- `dwbe`
  Double 16-bit-word encoding (21-bit Unicode), with big-endian byte-order;

`dwle`
: Double 16-bit-word encoding (21-bit Unicode), with little-endian byte-order;

`-u<user-code-mode>`
an optional argument indicating the version of Unicode interface (see sections 2.3, 3.3.2, 4.4), if any, to be included in the target scanner specification. The default version for each encoding is included if this argument is missing. Options are:

`suppress`
: Omit the Unicode interface from the target scanner specification;

`8bit`
: Include the 8-bit version of the Unicode interface in the target scanner specification. This is the default (and only possible) value for the UTF-8 and double-byte encodings.

`16bit`
: Include the 16-bit version of the Unicode interface in the target scanner specification. This is the default value for the UTF-16 and double-word encodings.

2.2 Pattern Translation in `reflex`

The general idea behind `reflex` is to select a Unicode encoding scheme, and transform the patterns in a scanner-specification file into equivalent encoded patterns. Within a pattern, input characters, input character literals, sets of input characters and strings of input characters are encoded under the scheme using the escape- sequence notation of the scanner-specification language (JFlex or Flex). A simple notational extension to the escape-sequence notation of the source scanner specification languages (see sections 3 and 4) is permitted by `reflex` in order to allow input characters to range over the entire Unicode set.

Table 1 gives a few examples of simple regular expressions and their corresponding translation under some encoding schemes. Other patterns lead to more interesting encoded patterns. For example, under encodings that support all of Unicode, the dot (‘`.`’) operator is interpreted by `reflex` as the set of all *Unicode* characters other than the newline character (`\xa`). Under UTF-8, ‘`.`’ translates to

Pattern	Encoded Pattern			
	UTF-8	Double-byte	UTF-16	Double-word
`a`	`\x61`	`\x00\x61`	`\x0061`	`\x0000\x0061`
`"a"`	`\x61`	`\x00\x61`	`\x0061`	`\x0000\x0061`
`\x61`	`\x61`	`\x00\x61`	`\x0061`	`\x0000\x0061`
`[\x61]`	`\x61`	`\x00\x61`	`\x0061`	`\x0000\x0061`
`[a-b]`	`[\x61-\x62]`	`\x00[\x61--\x62]`	`[\x0061-\x0061]`	`\x0000[\x0061--\x0062]`

Table 1: Encoding Simple Patterns

```
[\x00-\x09]|
[\x0b-\x7f]|
[\xc0-\xdf][\x80-\xbf]|
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf]
```

while under UTF-16 big-endian it gives rise (for JFlex) to the pattern

```
[\u0000-\u0009]|
[\u000b-\uffff]|
[\ud800-\udbff][\udc00-\udfff]
```

Under encodings that only represent the BMP, `reflex` interprets the dot operator as any character from the BMP other than newline. Under the double-byte encoding, '`.`' gives rise to the pattern

```
\x00[\x00-\x09]|
\x00[\x0b-\xff]|
[\x01-\xff][\x00-\xff]
```

For another example, consider the simple character interval (now in Flex notation):

```
[\x10000 - \x100000]
```

which leads under the UTF-16 big-endian encoding to the pattern

```
[\xd800-\xdbbf][xudc00-\xdfff]|\xdbc0\xdc00
```

Negated character classes, as with the dot operator, are interpreted by `reflex` in the context of the entire character set covered by the chosen encoding (i.e. all of Unicode or only th BMP). For example, the pattern representing the negation of the above class,

```
[^\x10000 - \x100000],
```

is mapped under UTF-16 big-endian by `reflex` to the pattern:

```
[\x0000-\xffff]|
\xdbc1[\xdc01-\xdfff]|
[\xdbc2-\xdbff][\xdc00-\xdfff]
```

This release of `reflex` does not perform any simplifications on the resulting patterns.

2.2.1 Representing Endian-ness in Patterns

In order to allow the generation of scanner-specification files for both big- and little-endian platforms, we include support for the little-endian variants of the UTF-16 and double-word encodings. However, under the little-endian encodings some input patterns lead to large encoded patterns. For example, character intervals, when translated under the UTF-16 or double-word little-endian encodings, lead to non-contiguous character intervals.

Consider the interval pattern `[\x0000-\x0010]`. While this pattern remains unchanged under the UTF-16 big-endian encoding, it expands to the following pattern under the UTF-16 little-endian encoding:

```
\x0000|\x0100|\x0200|\x0300|\x0400|\x0500|\x0600|\x0700|
\x0800|\x0900|\x0a00|\x0b00|\x0c00|\x0d00|\x0e00|\x0f00|\x1000
```

As the regular expression language of the scanner-specification languages does not contain a succinct notation for such character sequences, it is not recommended for the little-endian encodings to be used for input token grammars involving large character intervals.

2.3 Unicode-Extended Scanner Interface

The scanner programs generated by each of JFlex and Flex provide an interface that is transparent to the scanner's action code. The interface provides access to various scanning state information, such as the input text matching a given rule, the length of the matched input text, the number of lines or characters matched since the beginning of input, and so on.

The input to `reflex` is a high-level Unicode scanner specification, intended for compilation down to an 8 or 16-bit format suitable for consumption by a scanner-generator tool. The pattern sides of the rules are over

Unicode. It is therefore desirable for the action sides of the rules to have access to scanner-state information at the same level of granularity: in terms of *Unicode* characters. `reflex` facilitates this by providing a Unicode interface implementing an encoding-specific interpretation of various target-scanner state information.

Encodings involving 16-bit code units (UTF-16 and double-word) can further be encoded using the double-byte encoding. This is useful when the resulting scanner is intended to consume its input one byte at a time. To account for such dual-processing of a 16-bit encoded specification, `reflex` provides both 8 and 16-bit Unicode interfaces to these encodings.

Similar to the existing scanner interfaces, the Unicode interface uses a prefix of '`yy`' in naming its members. For more information about the particular Unicode interface provided for each source language, see sections 3.3.2 and 4.4.

3 Translating JFlex Specifications

The latest release of JFlex (version 1.4.1) supports 16-bit Unicode by allowing escape sequences of the form `\uXXXX`, where `X` denotes a hexadecimal digit. To represent all Unicode code points, `reflex` permits an extended Unicode escape sequence notation in the source JFlex file, of the form `\uXXXXXX`. That is, in the source file, a Unicode escape sequence is `\u` followed by from one to six hexadecimal digits.

In this section, we describe how `reflex` translates JFlex specifications. We begin by describing how regular expressions are translated, and then go on to outline how the translation impacts each of the three major sections contained in the source JFlex specification.

3.1 JFlex Pattern Translation

`reflex` translates the following JFlex pattern elements:

- character literal
- character escape-sequence (hexadecimal, octal, Unicode)
- character string
- character class (including negated classes, pre-defined classes and the dot operator)

Patterns in the input file are translated under the chosen encoding scheme to equivalent patterns using only escape sequences to represent input characters in the target input alphabet. For example, under the UTF-8 encoding,

- the character literal `a` translates to the hexadecimal escape sequence `\x61`;
- the hexadecimal escape sequence `\x61` translates to the same hexadecimal escape sequence;
- the octal escape sequence `\500` translates to the hexadecimal escape sequence `\xc5\x80`;
- the extended Unicode escape sequence `\u2f800` translates to the sequence `\xf0\xaf\xa0\x80`;
- the string `"xyz"` translates to the sequence `\x78\x79\x7a`;
- the negated character class `[^\u0000-\u024f]` translates to the expression

```
[\xc9-\xdf][\x90-\xbf]|
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf]
```

- the pre-defined character class `[:letter:]` translates to the expression !!!

The JFlex negation operator, '`!`', is unsupported in this release of `reflex`. Patterns involving this operator will not translate to equivalent patterns under the `reflex` transformation.

3.2 JFlex User Code Section

The user code section of the JFlex source file passes through the transformation untouched. Additionally, if `reflex` is *not* run with the option `-usuppress`, `reflex` adds `import` statement(s) (related to the Unicode interface, described below in section 3.3.2) to the user code section of the target JFlex file.

3.3 JFlex Options/Declarations Section

The options and declarations in the second section of a JFlex specifcation file that are affected by `reflex` are those concerning character sets, user class code, standalone scanners, character and column counting, and macro definitions. All other options included in the source file are passed through to the target specification.

3.3.1 Character Set Options

The character set option, if present in the source JFlex file, is ignored by `reflex`. The character set option inserted into the target JFlex file by `reflex` depends on the encoding being used. The option `%8bit` is inserted for UTF-8 and double-byte encodings since they use 8-bit code units. The option `%16bit` is inserted for the UTF-16 and double-word encodings since they use 16-bit code units.

3.3.2 User Class Code

If a user class code section is present in the source JFlex file, then it is passed on through to the target JFlex file. If `reflex` is run under options `-u8bit`, `-u16bit` or no `-u` option at all, then additional class code is inserted into the user class code section in the target JFlex file (if the source JFlex file does not have a user class code section, then under these options one is introduced in the target file). The `-usuppress` option causes the additional class code to be omiitted from the target file's user class code section.

The code added by `reflex` to the class code section implements an encoding-specific Unicode version of a subset of the existing JFlex scanner interface. Within actions in the target JFlex file, the Unicode interface provides an interpretation of the low-level code-unit strings as higher-level Unicode sequences. The interface consists of the following methods:

- `int[] yyUtext()`
  Similar to JFlex's `yytext()` method, `yyUtext()` represents the portion of the input text currently matched by the scanner. However, in the target scanner, where `yytext()` returns a string of code units from the target encoding, `yyUtext()` returns an integer array representing the encoding-specific interpretation of the code-unit string as a sequence of higher-level Unicode codepoints.

- `int yyUlength()`
  This method is the Unicode analogue to JFlex's `yylength()` method. The integer returned reprsents the number of Unicode characters given by the interpretation of the currently matched input text region as a sequence of Unicode codepoints.

- `int yyUcharat(int pos)`
  This method corresponds to JFlex's `yycharat(int pos)` method. The integer returned reprsents the Unicode codepoint at position `pos` in the Unicode interpretation (`yyUtext`) of the currently matched input text.

- `void yyUpushback(int num)`
  Similar to JFlex's `yypushback(int)` method, `yypushback(int num)` pushes the number of code units corresponding to `num` encoded Unicode codepoints back onto the input stream.

3.3.3 Standalone Scanner/CUP Options

If the standalone-scanner options `%debug` or `%standalone`, or the CUP-compatability option `%cupdebug` are used in the source JFlex file, they will be passed through. However, it should be noted that the `main` method generated by JFlex under these options will not use the appropriate *Reader* subclass to run the resulting scanner (see section B.1 for more details on running the a scanner generated from a specification generated by `reflex`).

3.3.4 Character/Column Counting Options

The `%char`, `%line` and `%column` options, when present in the JFlex source file, are passed through. It should be noted that this release of `reflex` does not provide Unicode versions of the JFlex scanner-API member variables `yychar`, `yyline` and `yycolumn` in the Unicode interface that it generates.

3.3.5 Macro Definitions

`reflex` translates the regular expression part of each macro definition in the JFlex source file as described in section 3.1.

3.4 JFlex Rules Section

`reflex` translates the pattern part of each rule in the JFlex source file as described in section 3.1.

Under the UTF-16 encodings, `reflex` adds a simple rule for scanning past the byte-order mark. In the big-endian case the rule is

```
\ufeff     {;}
```

and for the little-endian case it is simply:

```
\ufffe     {;}
```

4 Translating Flex Specifications

The latest release of Flex (version 2.5.33) supports an 8-bit character set, and the BMP is supported by an available 16-bit patch ([3]) which permits escape sequences of the form `\xXXXX`, where `X` denotes a hexadecimal digit. As with JFlex, `reflex` permits an extended escape-sequence notation in the Flex source scanner specification file, of the form `\xXXXXXX`.

In this section, we describe how `reflex` translates Flex specifications, beginning with how regular expressions are translated, and then outlining the effect of the translation on each of the three major sections contained in the source Flex specification.

4.1 Flex Pattern Translation

`reflex` translates the following Flex pattern elements:

- character literal
- character escape-sequence (hexadecimal, octal, ASCII)
- character string
- character class (including negated classes, pre-defined classes and the dot operator)

Patterns in the input file are translated under the chosen encoding scheme to equivalent patterns using only escape sequences to represent input characters in the target input alphabet. For example, under the UTF-16 big-endian encoding,

- the character literal `a` translates to the hexadecimal escape sequence `\x0061`;
- the hexadecimal escape sequence `\x61` translates to the hexadecimal escape sequence `\x0061`;
- the octal escape sequence `\500` translates to the hexadecimal escape sequence `\x0140`;
- the extended escape sequence `\x2f800` translates to the sequence `\xd840\xdc00`;
- the string `"xyz"` translates to the sequence `\x0078\x0079\x007a`;
- the negated character class `[^\x0000-\x024f]` translates to the pattern `[\x0250-\xffff]|[\xd800-\xdbff][\xdc00-\xdfff]`
- the pre-defined character class `[:alnum:]` translates to the expression !!!

4.2 Flex Definitions Section

Some Flex directives, when either included in the source scanner specification or passed to Flex as command line arguments, are incompatible with the target scanner specification produced by `reflex`:

`%option case-insensitive, %option caseless`
: The use of this option will invalidate the encoded patterns produced by `reflex` (command line form: `-i`);

`%option prefix`
: The Unicode interface included by `reflex` in the target Flex file relies on the default naming scheme for the scanner interface (prefix of `yy`); the use of this option will cause compile-time errors for the resulting scanner program; (command line form `-P`);

`%option c++`
: This release of `reflex` does not support this Flex option (command line form `-+`);

Directives for character sets (`%option 7bit`, `%option 8bit` are ignored by `reflex`. If the encoding used by `reflex` involves 8-bit code units (UTF-8, double byte) then `%option 8bit` is inserted in the target Flex definition section. If the encoding involves 16-bit code units (UTF-16, double-word) then no character-set option is inserted, and the target Flex file must be run using the 16-bit Flex patch.

`reflex` translates the regular expression part of each name definition in the Flex source file as described in section 4.1. Start conditions and user code sections are simply passed through the translation.

4.3 Flex Rules Section

`reflex` translates the pattern part of each rule in the Flex source file as described in section 4.1.

As for JFlex, under the UTF-16 encodings `reflex` adds a simple rule for scanning past the byte-order mark. In the big-endian case the rule is

```
\xfeff     {;}
```

and for the little-endian case it is simply:

```
\xfffe     {;}
```

4.4 Flex User Code Section

If a user code section is present in the source Flex file, then it is passed on through to the target Flex file. If `reflex` is run under options `-u8bit`, `-u16bit` or no `-u` option at all, then additional code is inserted into the user code section in the target Flex file (if the source Flex file does not have a user class section, then under these options one is introduced in the target file). The `-usuppress` option causes the additional code to be omitted from the target file's user code section.

The code added by `reflex` to the code section implements an encoding-specific Unicode version of a subset of the existing Flex scanner interface. Within actions in the target Flex file, the Unicode interface provides an interpretation of the low-level code-unit strings as higher-level Unicode sequences. The interface consists of the following methods:

- `int[] yyUtext()`
  Similar to Flex's global variable `yytext`, `yyUtext()` represents the portion of the input text currently matched by the scanner. However, in the target scanner, where `yytext` represents a string of code units from the target encoding, `yyUtext()` returns an integer array representing the encoding-specific interpretation of the code-unit string as a sequence of higher-level Unicode codepoints.

- `int yyUleng()`
  This method is the Unicode analogue to Flex's global variable `yyleng`. The integer returned reprsents the number of Unicode characters given by the interpretation of the currently matched input text region as a sequence of Unicode codepoints.

- `void yyUless(int num)`
  Similar to Flex's `yyless(int)` method, `yyUless(int num)` pushes the number of code units corresponding to all but the first *num* Unicode codepoints back onto the input stream.

5 Example

In this section we'll consider the translation of a JFlex scanner specification. The corresponding example for Flex can be found in Appendix B.

Condsider the following JFlex scanner specification [1] for a subset of XML ([1]) tokens ('start-tag', 'end-tag', 'space', 'whitespace', 'text, 'string', 'doctype-definition', 'processing-instruction' and 'entity'):)

```
%%
%public
%standalone
%{
public void print_token(String name, String lexeme){
  System.out.print(name + "\t" + lexeme + "\n");
}
public void print_token(String name){
  System.out.print(name + "\n");
}
%}
underscore =   \u0005F
colon      =   \u0003A
dot        =   \u0002E
dash       =   \u0002D
NAMESTART    =  {colon}|{underscore}|[\x41-\x5a\x61-\x7a]|[\xC0-\xD6]|    \
                [\xD8-\xF6]|[\u0F8-\u2FF]|[\u370-\u37D]|[\u37F-\u1FFF]|   \
                [\u200C-\u200D]|[\u2070-\u218F]|[\u2C00-\u2FEF]|          \
                [\u3001-\uD7FF]|[\uF900-\uFDCF]| [\uFDF0-\uFFFD]|         \
                [\u10000-\uEFFFF]
NAMECHAR   =  {NAMESTART}|{dash}|{dot}|[0-9]|\xB7|[\u0300-\u036F]|        \
              [\u203F-\u2040]
```

[1]For formatting purposes, we use the `'\'` character to indicate wrapped lines.

```
NAME       =  {NAMESTART}{NAMECHAR}*
WS         =  [\x20\x9\xD\xA]+
%xstate  IN_START_TAG IN_END_TAG IN_DTD IN_PI
%%
"<!"        {
             print_token("DTD");
             yybegin(IN_DTD);
            }
"<?"        {
             print_token("PI");
             yybegin(IN_PI);
            }
"<"{NAME}   {
             print_token("START_TAG", yytext().substring(1));
             yybegin(IN_START_TAG);
            }
"</"{NAME}  {
             print_token("END_TAG", yytext().substring(2));
             yybegin(IN_END_TAG);
            }
{WS}+       {print_token("WHITESPACE");}
[^<>&\'\"\\= \t\n\r\f]+  {print_token("TEXT" + yytext());}

"&"(\#?[a-zA-Z0-9]+)";"  {print_token("ENTITY");}
.|\n                     {;}

<IN_START_TAG>{
\'[^\']+\' | \"[^\"]+\"     {print_token("STRING",                      \
                                  yytext().substring(1,yylength()-1));}
"="                         {print_token("EQUALS");}
[ \n]+                      {print_token("SPACE"); }
[^<>&\'\"\\= \t\n\r\f]+     {print_token("TEXT", yytext());}
">"|"/>"                    {yybegin(YYINITIAL);}
}

<IN_PI>{
">" {yybegin(YYINITIAL);}
.   {;}
}

<IN_END_TAG>{
">" {yybegin(YYINITIAL);}
.   {System.out.println("Unexpected character in end tag.");}
}
```

```
68 <IN_DTD>{
69 ">" {yybegin(YYINITIAL);}
70 .   {;}
71 }
```

Line 20 illustrates the use of the extended unicode escape sequence.

Assuming the name of the file is ‘`SimpleXML.jflex`’, if we wish to generate an equivalent 16-bit JFlex scanner specification using the UTF-16 big-endian encoding, we run `reflex` using

```
java lext.Main -sjflex -eutf16be SimpleXML.jflex SimpleXML-UTF16BE.jflex,
```

which results in the file ‘`SimpleXML-UTF16BE.jflex`’, containing the following JFlex specification:

```
import java.util.Vector;
%%
%public
%standalone
%{
public void print_token(String name, String lexeme){
  System.out.print(name + "\t" + lexeme + "\n");
}
public void print_token(String name){
  System.out.print(name + "\n");
}

private int[] yyUtext(){
String token = yytext();
Vector scalars= new Vector();
int c;
for(int i=0; i<token.length(); i++){
c = (int)(token.charAt(i));
//Single 16-bit word :
if( (c >= 0x0000 && c < 0xd800) || (c > 0xdfff && c <= 0xffff) ){
scalars.add(new Integer(decode16BitUTF16BEChar(                     \
                        token.substring(i, i+1).toCharArray())));
}
//Surrogate pair (here two 16-bit words):
else if ( c >= 0xd800 && c <= 0xdbff ){
scalars.add(new Integer(decode16BitUTF16BEChar(                     \
                        token.substring(i, i+2).toCharArray())));
i+=1;
}
```

```
}
int[] result = new int[scalars.size()];
for(int i=0; i<scalars.size();i++)
result[i]=((Integer)scalars.elementAt(i)).intValue();
return result;
}
private int yyUlength() {
return yyUtext().length;
}
private int yyUcharat(int pos){
int[] t = yyUtext();
if(pos>=0 && pos < t.length)
return t[pos];
else return -1;
}
private void yyUpushback(int number){
int[] t = yyUtext();
int count=0, uscalar;
for(int i=0;i<number;i++){
uscalar=t[i];
if(uscalar>=0x0 && uscalar<=0xffff)
count+=1;
else if(uscalar>0xffff && uscalar<=0x10ffff)
count+=2;
}
yypushback(count);
}
private int decode16BitUTF16BEChar(char[] utf16bechar){
//single word:
if(utf16bechar.length==1)
return (int)utf16bechar[0];
//surrogate pair:
else if(utf16bechar.length==2){
int y5, p4;
p4 = (int)((utf16bechar[0] & 0x03c0)>>>6);
y5 = p4+1;
return y5<<16 | (utf16bechar[0] & 0x003f)<<10 |                    \
                (utf16bechar[1] & 0x03ff);
}else return -1;
}
%}
underscore =   \u005f
colon      =   \u003a
dot        =   \u002e
dash       =   \u002d
```

```
NAMESTART    =  {colon}|{underscore}|[\u0041-\u005a]|               \
                [\u0061-\u007a]|[\u00c0-\u00d6]|                    \
                [\u00d8-\u00f6]|[\u00f8-\u02ff]|                    \
                [\u0370-\u037d]|[\u037f-\u1fff]|                    \
                [\u200c-\u200d]|[\u2070-\u218f]|                    \
                [\u2c00-\u2fef]|[\u3001-\ud7ff]|                    \
                [\uf900-\ufdcf]|[\ufdf0-\ufffd]|                    \
                \ud800[\udc00-\udfff]|                              \
                [\ud801-\udb7e][\udc00-\udfff]|                     \
                \udb7f[\udc00-\udfff]
NAMECHAR   =  {NAMESTART}|{dash}|{dot}|[\u0030-\u0039]|             \
              \u00b7|[\u0300-\u036f]|[\u203f-\u2040]
NAME       =  {NAMESTART}(({NAMECHAR})*)
WS         =  (\u0020|\u0009|\u000d|\u000a)+
%xstate  IN_START_TAG IN_END_TAG IN_DTD IN_PI
%16bit
%%
\ufeff                 {;}
\u003c\u0021           {
             print_token("DTD");
             yybegin(IN_DTD);
            }
\u003c\u003f           {
             print_token("PI");
             yybegin(IN_PI);
            }
\u003c({NAME})          {
             print_token("START_TAG", yytext().substring(1));
             yybegin(IN_START_TAG);
            }
\u003c\u002f({NAME})            {
             print_token("END_TAG", yytext().substring(2));
             yybegin(IN_END_TAG);
            }
({WS})+         {print_token("WHITESPACE");}
([\u0000-\u0008]|\u000b|[\u000e-\u001f]|\u0021|[\u0023-\u0025]|     \
[\u0028-\u003b]|[\u003f-\u005b]|[\u005d-\uffff]|                    \
[\ud800-\udbff][\udc00-\udfff])+   {print_token("TEXT" + yytext());}

\u0026(((\u0023)?(([\u0061-\u007a]|[\u0041-\u005a]|                 \
[\u0030-\u0039])+))(\u003b))       {print_token("ENTITY");}
[\u0000-\u0009]|[\u000b-\uffff]|                                    \
[\ud800-\udbff][\udc00-\udfff]|\u000a          {;}

<IN_START_TAG>{
```

```
120 \u0027(([\u0000-\u0026]|[\u0028-\uffff]|                             \
121 [\ud800-\udbff][\udc00-\udfff])+(\u0027))|                           \
122 \u0022(([\u0000-\u0021]|[\u0023-\uffff]|                             \
123 [\ud800-\udbff][\udc00-\udfff])+(\u0022))                            \
124       {print_token("STRING", yytext().substring(1,yylength()-1));}
125 \u003d              {print_token("EQUALS");}
126 (\u0020|\u000a)+    {print_token("SPACE"); }
127 ([\u0000-\u0008]|\u000b|[\u000e-\u001f]|\u0021|[\u0023-\u0025]|      \
128 [\u0028-\u003b]|[\u003f-\u005b]|[\u005d-\uffff]|                     \
129 [\ud800-\udbff][\udc00-\udfff])+  {print_token("TEXT", yytext());}
130 \u003e|\u002f\u003e              {yybegin(YYINITIAL);}
131 }
132 
133 <IN_PI>{
134 \u003e          {yybegin(YYINITIAL);}
135 [\u0000-\u0009]|[\u000b-\uffff]|[\ud800-\udbff][\udc00-\udfff]       \
136          {;}
137 }
138 
139 <IN_END_TAG>{
140 \u003e          {yybegin(YYINITIAL);}
141 [\u0000-\u0009]|[\u000b-\uffff]|[\ud800-\udbff][\udc00-\udfff]       \
142         {System.out.println("Unexpected character in end tag.");}
143 }
144 
145 <IN_DTD>{
146 \u003e          {yybegin(YYINITIAL);}
147 [\u0000-\u0009]|[\u000b-\uffff]|[\ud800-\udbff][\udc00-\udfff]       \
148         {;}
149 }
```

From the resulting listing, the following observations can be made:

- Line 1:
  contains the `import` statement added by `reflex`;
- Lines 6 - 12:
  contain the original user class code;
- Lines 13 - 69:
  contain the Unicode interface introduced by `reflex`;
- Lines 71 - 88:
  contain the transformed macro definitions;

- Line 90:
  contains the character set option for the target 16-bit input alphabet;
- Line 92:
  contains the rule inserted by `reflex` for scanning past the byte order mark;
- Lines 93 - 147:
  contain the original rules, with transformed patterns;

5.1 Running the Target Scanner

Running the above target of `reflex` through JFlex using

`jflex SimpleXML-UTF16BE.jflex`

gives the Java source code specification, `Yylex.java`, of the corresponding scanner program which, when compiled gives the class `Yylex`. The constructor methods for this class (and the `main` method included in the class under JFlex's '`standalone`' option) use the `FileReader` and `InputStreamReader` objects from Java's `Reader` class hierarchy to set up the scanner program for consuming input files as *text* files. These objects perform internal translation of the bytes in their corresponding input file to Unicode characters under the platform's default encoding. However, the scanners we are generating use an explicit representation of encoded Unicode characters, and so require direct access to the bytes in the encoded input file to be scanned. That is, we need to scan the input file as a *binary stream* rather than as a *text file*. As discussed in [4], this is accomplished by using a custom `InputStreamReader` class in which no byte translation is performed. An example of such a class is the `StraightStreamReader` class, included in the `examples/binary/` directory of the JFlex distribution, which consumes its associated input file byte-by-byte. It is a straightforward exercise to construct a subclass of the `StraightStreamReader` class that consumes an input file two bytes at a time (as we need in the case of the 16-bit encodings).

Once we have a custom `InputStreamReader` class, say '`_16BitStraightStreamReader`', capable of reading two bytes at a time, we can run the scanner program using a simple application class such as that resulting from the following source code:

```
import java.io.*;
public class runJflexExample {

 public static void main(String argv[]) {
```

```
        try {
            Yylex scanner = new Yylex(                          \
                            new _16BitStraightStreamReader(       \
                            new FileInputStream(argv[0])) );
            scanner.yylex();
        }
        catch (java.io.IOException e) {
          System.out.println("IO error scanning file.");
          System.out.println(e);
        }
        catch (Exception e) {
          System.out.println("Unexpected exception:");
          e.printStackTrace();
        }
  }

}
```

For sample input and output files corresponding to running this application, see Appendix sections A.1 and A.2, respectively.

The input file to the scanner must be encoded using the same encoding as that used by `reflex` to generate the target scanner. It is a straightforward exercise to implement a program which converts between various file formats.

5.2 Dual-Processing a Scanner Specification

If it is desired to generate a scanner over an 8-bit rather than a 16-bit input alphabet (e.g. to result in the smallest possible scanner tables) and furthermore the target encoding for `reflex` uses 16-bit code units (UTF-16 and double-word), then a scanner specification can be dual-processed by `reflex`. For example, the JFlex scanner specification for XML tokens presented earlier in this section be processed twice by `reflex` to result in an 8-bit scanner specification for UTF-16 encoded patterns. This is accomplished using the following two runs of `reflex`:

```
java lext.Main -sjflex  -eutf16be      \
               -u8bit SimpleXML-UTF16BE.jflex SimpleXML-UTF16BE.jflex
```

followed by:

```
java lext.Main -sjflex  -edbyte
               -usuppress SimpleXML-UTF16BE.jflex SimpleXML-UTF16BE-8BIT.jflex
```

The resulting scanner is then run using a byte-wise custom `InputStreamReader` class, as in the previous section.

6 Conclusion

The major effort required to introduce Unicode support in scanner-specification languages by way of a total overhaul, preceded by the major delay required for updated support in the underlying programming language, can be avoided by instead considering low-level token recognition, as we have shown in this report.

We have described an original tool, `reflex`, for transforming scanner-specifications for patterns over the Unicode input alphabet into equivalent scanner specifications for patterns over 8- or 16-bit input alphabets. The patterns in the latter represent the low-level code unit sequences corresponding to an encoding of the former. A suitably encoded source file containing Unicode can be consumed by a scanner generated by a target of `reflex` and broken into a stream of Unicode tokens.

References

[1] *Extensible Markup Language (XML) 1.1.*
T. Bray et al. (Eds.). April 2004.
`http://www.w3.org/TR/2004/REC-xml11-20040204/`

[2] *Flex, version 2.5 - A fast scanner generator.*
Vern Paxson. Edition 2.5, March 1995.
`http://jflex.de`

[3] *Flex 2.5.4a Unicode Patch.*
James Laugh,
`http://software.decisionsoft.com/software/flex-2.5.4a-unicode-patch`

[4] *JFlex User's Manual.*
Gerwin Klein, `http://jflex.de/manual.html`

[5] *Lex - A Lexical Analyzer Generator.*
M.E. Lesk. Comp. Sci. Tech. Rep. No. 39. Bell Laboratories, October 1975.

[6] *The Unicode Standard Version 4.0.*
The Unicode Consortium, Addison-Wesley, 2003.
`http://www.unicode.org`

A JFlex Example Cont'd

A.1 Input File

```
<?xml version="1.0" ?>
<project name="LEXT" default="buildLext" basedir=".">

<property name="srcdir"     value="src"/>
<property name="builddir"   value="build"/>
<property name="classdir"   value="${builddir}"/>

<path id="project.classpath">
   <pathelement path="${classdir}" />
  </path>

<target name="buildLext">
    <javac srcdir="${builddir}" destdir="${classdir}"
           debug="on" optimize="off" deprecation="on">
      <classpath refid="project.classpath"/>
    </javac>
</target>

</project>
```

A.2 Scanner Output

```
PI
WHITESPACE
START_TAG       project
SPACE
TEXT    name
EQUALS
STRING  LEXT
SPACE
TEXT    default
EQUALS
STRING  buildLext
SPACE
TEXT    basedir
EQUALS
```

```
STRING  .
WHITESPACE
START_TAG       property
SPACE
TEXT    name
EQUALS
STRING  srcdir
SPACE
TEXT    value
EQUALS
STRING  src
WHITESPACE
START_TAG       property
SPACE
TEXT    name
EQUALS
STRING  builddir
SPACE
TEXT    value
EQUALS
STRING  build
WHITESPACE
START_TAG       property
SPACE
TEXT    name
EQUALS
STRING  classdir
SPACE
TEXT    value
EQUALS
STRING  ${builddir}
WHITESPACE
START_TAG       path
SPACE
TEXT    id
EQUALS
STRING  project.classpath
WHITESPACE
START_TAG       pathelement
SPACE
TEXT    path
EQUALS
STRING  ${classdir}
SPACE
WHITESPACE
```

```
60   END_TAG path
61   WHITESPACE
62   START_TAG       target
63   SPACE
64   TEXT    name
65   EQUALS
66   STRING  buildLext
67   WHITESPACE
68   START_TAG       javac
69   SPACE
70   TEXT    srcdir
71   EQUALS
72   STRING  ${builddir}
73   SPACE
74   TEXT    destdir
75   EQUALS
76   STRING  ${classdir}
77   SPACE
78   TEXT    debug
79   EQUALS
80   STRING  on
81   SPACE
82   TEXT    optimize
83   EQUALS
84   STRING  off
85   SPACE
86   TEXT    deprecation
87   EQUALS
88   STRING  on
89   WHITESPACE
90   START_TAG       classpath
91   SPACE
92   TEXT    refid
93   EQUALS
94   STRING  project.classpath
95   WHITESPACE
96   END_TAG javac
97   WHITESPACE
98   END_TAG target
99   WHITESPACE
100  END_TAG project
101  WHITESPACE
```

B Flex Example

The Flex example corresponding to the example given in section 5 is given is this section. Consider the following Flex listing:

```
%{
YY_CHAR *lexeme;
int i;

void print_token(char *type, YY_CHAR *lexeme, int len){
        int i;
        printf("%s\t\t", type);
        for(i=0;i<len;i++){
          printf("%c",lexeme[i]);
        }
        printf("\n");
}
YY_CHAR *get_lexeme(){
  return memcpy(malloc(sizeof(YY_CHAR)*yyleng),        \
                yytext,sizeof(YY_CHAR)*yyleng);
}
%}
%option noyywrap
underscore  \x0005F
colon       \x0003A
dot         \x0002E
dash        \x0002D
NAMESTART   {colon}|{underscore}|[\x41-\x5a\x61-\x7a]|            \
            [\xC0-\xD6]|[\xD8-\xF6]|[\x0F8-\x2FF]|[\x370-\x37D]|  \
            [\x37F-\x1FFF]|[\x200C-\x200D]|[\x2070-\x218F]|       \
            [\x2C00-\x2FEF]|[\x3001-\xD7FF]|[\xF900-\xFDCF]|      \
            [\xFDF0-\xFFFD]|[\u10000-\uEFFFF]
NAMECHAR    {NAMESTART}|{dash}|{dot}|[0-9]|\xB7|[\x0300-\x036F]|  \
            [\x203F-\x2040]
NAME        {NAMESTART}{NAMECHAR}*
WS          [\x20\x9\xD\xA]+
%x  IN_START_TAG IN_END_TAG IN_DTD IN_PI
%%
"<!"        {
             print_token("DTD", NULL, 0);
             BEGIN(IN_DTD);
            }
"<?"        {
             print_token("PI", NULL, 0);
             BEGIN(IN_PI);
```

```
             }
"<"{NAME}   {
             lexeme = get_lexeme();
             print_token("START_TAG", lexeme+1, yyleng-1);
             free(lexeme);
             BEGIN(IN_START_TAG);
            }
"</"{NAME}  {
             lexeme=get_lexeme();
             print_token("END_TAG", lexeme+2, yyleng-2);
             free(lexeme);
             BEGIN(IN_END_TAG);
            }

{WS}+                    {print_token("WHITESPACE", NULL,   0);     }
[^<>&\'\"\\= \t\n\r\f]+  {print_token("TEXT",       yytext, yyleng);}

"&"(\#?[a-zA-Z0-9]+)";"  {print_token("ENTITY", NULL,   0);}
.|\n                     {;}

<IN_START_TAG>{
"="                      {print_token("EQUALS", NULL,   0);}
[ \n]+                   {print_token("SPACE",  NULL,   0);}
\'[^\']+\'|\"[^\"]+\"    {
                          lexeme=get_lexeme();
                          print_token("STRING", lexeme+1,yyleng-2);
                          free(lexeme);
                         }
[^<>&\'\"\\= \t\n\r\f]+  {print_token("TEXT",   yytext, yyleng);}
"/>"|">"                 {BEGIN(INITIAL);}
 }

<IN_PI>{
">" {BEGIN(INITIAL);}
.   {;}
}

<IN_END_TAG>{
">" {BEGIN(INITIAL);}
.   {printf("Unexpected character in end tag.");}
}

<IN_DTD>{
">" {BEGIN(INITIAL);}
.   {;}
```

```
}

%%
 main( argc, argv )
          int argc;
          char **argv;
              {
              ++argv, --argc;  /* skip over program name */
              if ( argc > 0 )
                      yyin = fopen( argv[0], "r" );
              else
                      yyin = stdin;

              yylex();
              }
```

Assumming the name of the file is '`SimpleXML.flex`', if we wish to generate an equivalent 8-bit Flex scanner specification using the UTF-8 encoding, we run `reflex` using

`java lext.Main -sflex -eutf8 SimpleXML.flex SimpleXML-UTF8.flex`,

which results in the following Flex specification:

```
%{
YY_CHAR *lexeme;
int i;

void print_token(char *type, YY_CHAR *lexeme, int len){
        int i;
        printf("%s\t\t", type);
        for(i=0;i<len;i++){
          printf("%c",lexeme[i]);
        }
        printf("\n");
}
YY_CHAR *get_lexeme(){
  return memcpy(malloc(sizeof(YY_CHAR)*yyleng),yytext,sizeof(YY_CHAR)*yyleng);
}
%}
%option noyywrap
%option 8bit
underscore  \x5f
colon       \x3a
```

```
dot         \x2e
dash        \x2d
NAMESTART   {colon}|{underscore}|[\x41-\x5a]|[\x61-\x7a]|            \
            \xc3[\x80-\x96]|\xc3[\x98-\xb6]|\xc3[\xb8-\xbf]|         \
            [\xc4-\xca][\x80-\xbf]|\xcb[\x80-\xbf]|\xcd[\xb0-\xbd]|  \
            [\xcd-\xdf]\xbf|[\xe0-\xe1][\x80-\xbf][\x80-\xbf]|       \
            \xe2\x80[\x8c-\x8d]|\xe2\x81[\xb0-\xbf]|                 \
            \xe2[\x82-\x85][\x80-\xbf]|\xe2\x86[\x80-\x8f]|          \
            \xe2\xb0[\x80-\xbf]|\xe2[\xb1-\xbe][\x80-\xbf]|          \
            \xe2\xbf[\x80-\xaf]|\xe3[\x80-\xbf][\x81-\xbf]|          \
            [\xe4-\xec][\x80-\xbf][\x80-\xbf]|                       \
            \xed[\x80-\x9f][\x80-\xbf]|\xef\xa4[\x80-\xbf]|          \
            \xef[\xa5-\xb6][\x80-\xbf]|\xef\xb7[\x80-\x8f]|          \
            \xef\xb7[\xb0-\xbf]|\xef[\xb8-\xbe][\x80-\xbf]|          \
            \xef\xbf[\x80-\xbd]|                                     \
            \xf0[\x90-\xbf][\x80-\xbf][\x80-\xbf]|                   \
            [\xf1-\xf2][\x80-\xbf][\x80-\xbf][\x80-\xbf]|            \
            \xf3[\x80-\xaf][\x80-\xbf][\x80-\xbf]
NAMECHAR    {NAMESTART}|{dash}|{dot}|[\x30-\x39]|\xc2\xb7|           \
            \xcc[\x80-\xbf]|\xcd[\x80-\xaf]|\xe2\x80\xbf|            \
            \xe2\x81\x80
NAME        {NAMESTART}(({NAMECHAR})*)
WS          (\x20|\x09|\x0d|\x0a)+
%x  IN_START_TAG IN_END_TAG IN_DTD IN_PI
%%
\x3c\x21                {
             print_token("DTD", NULL, 0);
             BEGIN(IN_DTD);
            }
\x3c\x3f                {
             print_token("PI", NULL, 0);
             BEGIN(IN_PI);
            }
\x3c({NAME})            {
             lexeme = get_lexeme();
             print_token("START_TAG", lexeme+1, yyleng-1);
             free(lexeme);
             BEGIN(IN_START_TAG);
            }
\x3c\x2f({NAME})                {
             lexeme=get_lexeme();
             print_token("END_TAG", lexeme+2, yyleng-2);
             free(lexeme);
             BEGIN(IN_END_TAG);
            }
```

```

({WS})+         {print_token("WHITESPACE", NULL,   0);     }
([\x00-\x08]|\x0b|[\x0e-\x1f]|\x21|[\x23-\x25]|[\x28-\x3b]|           \
[\x3f-\x5b]|[\x5d-\x7f]|[\xc0-\xdf][\x80-\xbf]|                      \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                   \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf])+                       \
      {print_token("TEXT",       yytext, yyleng);}

\x26(((\x23)?(([\x61-\x7a]|[\x41-\x5a]|[\x30-\x39])+))(\x3b))         \
      {print_token("ENTITY", NULL,   0);}
[\x00-\x09]|[\x0b-\x7f]|[\xc0-\xdf][\x80-\xbf]|                      \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                   \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf]|\x0a     {;}

<IN_START_TAG>{
\x3d            {print_token("EQUALS", NULL,   0);}
(\x20|\x0a)+            {print_token("SPACE",  NULL,   0);}
\x27(([\x00-\x26]|[\x28-\x7f]|[\xc0-\xdf][\x80-\xbf]|                \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                   \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf])+(\x27))|               \
\x22(([\x00-\x21]|[\x23-\x7f]|[\xc0-\xdf][\x80-\xbf]|                \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                   \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf])+(\x22))                \
                         {
                          lexeme=get_lexeme();
                          print_token("STRING", lexeme+1,yyleng-2);
                          free(lexeme);
                         }
([\x00-\x08]|\x0b|[\x0e-\x1f]|\x21|[\x23-\x25]|[\x28-\x3b]|          \
[\x3f-\x5b]|[\x5d-\x7f]|[\xc0-\xdf][\x80-\xbf]|                      \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                   \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf])+                       \
                {print_token("TEXT",   yytext, yyleng);}
\x2f\x3e|\x3e           {BEGIN(INITIAL);}
 }

<IN_PI>{
\x3e            {BEGIN(INITIAL);}
[\x00-\x09]|[\x0b-\x7f]|[\xc0-\xdf][\x80-\xbf]|                      \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                   \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf]            {;}
}

<IN_END_TAG>{
\x3e            {BEGIN(INITIAL);}
```

```
[\x00-\x09]|[\x0b-\x7f]|[\xc0-\xdf][\x80-\xbf]|                     \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                  \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf]                        \
                  {printf("Unexpected character in end tag.");}
}

<IN_DTD>{
\x3e            {BEGIN(INITIAL);}
[\x00-\x09]|[\x0b-\x7f]|[\xc0-\xdf][\x80-\xbf]|                     \
[\xe0-\xef][\x80-\xbf][\x80-\xbf]|                                  \
[\xf0-\xf4][\x80-\x8f][\x80-\xbf][\x80-\xbf]            {;}
}

%%
const int UUSEG_SIZE = 20;
typedef struct {
 int len;
 int *vec;
} UUDynarray;
void UUda_init(UUDynarray *d){
int i;
d->len=0;
d->vec = (int *)malloc(sizeof(int[UUSEG_SIZE]));
for(i=0;i<UUSEG_SIZE;i++)
d->vec[i]=0;
}
void UUda_bump(UUDynarray *d){
int *iptr = (int *)malloc(sizeof(int[d->len + UUSEG_SIZE])),i;
for(i=0;i<d->len;i++){
iptr[i]=d->vec[i];
}
for(i=d->len;i<d->len + UUSEG_SIZE;i++)
iptr[i]=0;
d->vec = iptr;
}
void UUda_add(UUDynarray *d, int i){
if(d->len%UUSEG_SIZE == 0)
UUda_bump(d);
d->vec[d->len]=i;
d->len++;
}
//Copy D's int array into result, and
//append a '-1' to the end (this works
//because there are no negative Unicode
//scalars).
```

```
void UUda_copy(UUDynarray d, int **result){
int i;
*result = (int *)malloc(sizeof(int[d.len+1]));
for(i=0;i<d.len;i++)
(*result)[i]=d.vec[i];
//mark the end:
(*result)[d.len]=-1;
}
void UUda_term(UUDynarray *d){
free(d->vec);
}
typedef struct {
int len;
char *code;
} UUchar;
int * yyUtext();
//Constructor of Uchar from a string:
UUchar _UUchar(char *str, int start, int stop){
UUchar result;
char *units_ptr = (char *)malloc(sizeof(char[stop-start]));
int i;
if(stop<=start || str==NULL || yyleng==0){
result.len=-1;
result.code=NULL;
return;
}
for(i=0;i<stop-start;i++)
units_ptr[i] = str[i+start];
result.code = units_ptr;
result.len = stop-start;
return result;
}

int * yyUtext() {
char *token = yytext;
int i, *result, c;
UUDynarray D;
UUda_init(&D);
for(i=0; i<yyleng; i++){
c = (int)token[i];
//single-byte sequence:
if(c >=0x00 && c <= 0x7f){
UUda_add(&D, UUdecodeUTF8Char(_UUchar(token, i, i+1)));
//multi-byte sequence:
}else if(c >= 0xc0 && c <= 0xfd){
```

```
//two bytes:
if( ((c & 0x00F0)>>8)==0x0c || ((c & 0x00F0)>>8)==0x0d ){
UUda_add(&D, UUdecodeUTF8Char(_UUchar(token, i, i+2)));
i+=1;
//three bytes:
}else if(((c & 0x00F0)>>8)==0x0e){
UUda_add(&D, UUdecodeUTF8Char(_UUchar(token, i, i+3)));
i+=2;
//four bytes:
}else if(((c & 0x00F0)>>8)==0x0f){
UUda_add(&D, UUdecodeUTF8Char(_UUchar(token, i, i+4)));
i+=3;
}
}
}
UUda_copy(D, &result);
UUda_term(&D);
return result;
}
int yyUleng(){
int *scalars = yyUtext(),i;
while(scalars[i]!=-1)
i++;
return i;
}
int UUdecodeUTF8Char(UUchar utf8char){
int len = utf8char.len;
if(len==1){
return (int)utf8char.code[0] & 0x000000ff;
}else if(len==2){
return ((int)utf8char.code[0] & 0x1f)<<6 |                \
       ((int)utf8char.code[1] & 0x3f);
}else if(len==3){
return ((int)utf8char.code[0] & 0x0f)<<12 |               \
       ((int)utf8char.code[1] & 0x3f)<<6 |                \
       ((int)utf8char.code[2] & 0x3f);
}else if(len==4){
return ((int)utf8char.code[0] & 0x07)<<18 |               \
       ((int)utf8char.code[1] & 0x3f)<<12 |               \
       ((int)utf8char.code[2] & 0x3f)<<6 |                \
       ((int)utf8char.code[3] & 0x3f);
}
return -1;
}
 main( argc, argv )
```

```
246             int argc;
247             char **argv;
248                 {
249                 ++argv, --argc;  /* skip over program name */
250                 if ( argc > 0 )
251                         yyin = fopen( argv[0], "r" );
252                 else
253                         yyin = stdin;
254
255                 yylex();
256                 }
```

From the resulting listing, the following observations can be made:

- Lines 1 - 16:
  contain the original user code fragment;
- Line 18:
  contains the character set option for the target 8-bit input alphabet;
- Lines 19 - 42:
  contain the transformed name definitions;
- Lines 46 - 122:
  contain the original rules, with transformed patterns;
- Lines 126 - 244:
  contain the Unicode interface introduced by `reflex`;
- Lines 245 - 256:
  contain the original user-code;

B.1 Running the Target Flex Scanner

Running the above target of `reflex` through Flex using

`flex SimpleXML-UTF8.flex`

gives the C source code specification, `lex.yy.c`, of the corresponding scanner program which, when compiled gives the scanner program. Since our target encoding uses 8-bit code units, the scanner generated by Flex reads bytes from the input file directly into the C `char` type (minimum 8-bits wide), and we have included a main function in the user code section, we can simply run the resulting program on the (UTF-8 encoded) input.