Compiler Construction

CSCI-742 **Project 1**

Term 20145 **January 30, 2015**

LangF Scanner

Due: February 13, 2015

1 Introduction

The first project is to implement a scanner (or lexer or tokenizer) for LangF, which will convert an input stream of characters into a stream of tokens. While such components of a compiler are often best written using a *lexical analyzer generator* (e.g., Lex or Flex (for C), JLex (for Java), ML-Lex or ML-ULex (for Standard ML), Alex (for Haskell), SILex (for Scheme)), you will write a scanner by hand.

2 LangF Lexical Conventions

LangF has four classes of *tokens*: keywords, delimiter and operator symbols, identifiers, and integer and string literals. Tokens can be separated by *whitespace* and/or *comments*.

LangF has the following set of keywords:

`and`	`andalso`	`case`	`datatype`
`else`	`end`	`fn`	`fun`
`if`	`in`	`let`	`of`
`orelse`	`then`	`type`	`val`

A keyword may not be used as an identifier.

LangF also has the following collection of delimiter and operator symbols:

`+`	`-`	`*`	`/`	`%`	`~`	`==`	`<>`	`<=`	`<`
`>=`	`>`	`^`	`!`	`:=`	`#`	`(`	`)`	`[`	`]`
`{`	`}`	`->`	`=>`	`=`	`:`	`,`	`;`	`|`	`_`

LangF has three classes of *identifiers*: type variables, constructors, and variables. An identifier in LangF can be any string of letters, digits, underscores, and (single) quote marks, with the following restrictions:

- a type variable must begin with a single quote character and a lower-case letter (e.g., `'alpha`)
- a constructor must begin with an upper-case letter (e.g., `List`)
- a variable must begin with a lower-case letter (e.g., `length`)

Identifiers are case-sensitive (e.g., `treemap` is different from `treeMap`).

Integer literals in LangF are written using decimal notation, with an optional negation symbol "`~`".

String literals in LangF start with a double quote "`"`", followed by zero or more printable characters (ASCII codes 32 – 126), and terminated with a double quote. A string literal can contain the following C-like escape sequences:

`\a`	—	bell (ASCII code 7)
`\b`	—	backspace (ASCII code 8)
`\f`	—	form feed (ASCII code 12)
`\n`	—	newline (ASCII code 10)
`\r`	—	carriage return (ASCII code 13)
`\t`	—	horizontal tab (ASCII code 9)
`\v`	—	vertical tab (ASCII code 11)
`\\`	—	backslash
`\"`	—	double quote

A character in a string literal may also be specified by its numerical value (ASCII code) using the escape sequence "`\`*`ddd`*", where *`ddd`* is a sequence of exactly three decimal digits. Strings in LangF may contain any 8-bit character, including embedded zeros, which can be specified as "`\000`".

Comments in LangF start anywhere outside a string literal with a "`(*`" and are terminated with a matching "`*)`". Comments in LangF may be nested.

Whitespace in LangF is any non-empty sequence of spaces (ASCII code 32), horizontal tabs, newlines, vertical tabs, form feeds, or carriage returns.

3 Requirements

Your implementation should include (at least) the following module:

```
structure LangFHandScanner: LANGF_HAND_SCANNER
```

where the LANGF_HAND_SCANNER signature is as follows:

```
signature LANGF_HAND_SCANNER =
sig
   val scan : {reportError: string -> unit} ->
              (char, 'strm) StringCvt.reader ->
              (Tokens.token, 'strm) StringCvt.reader
end
```

The `StringCvt.reader` type is defined in the SML Basis Library as follows:

```
type ('item,'strm) reader = 'strm -> ('item * 'strm) option
```

A reader is a function that takes a stream and returns either `SOME` pair of the next item and the rest of the stream, or `NONE` when the end of the stream is reached. Thus `scan` is a function that takes a character reader and returns a token reader.

The `structure Tokens : TOKENS` module is provided in the seed code; then `TOKENS` signature is as follows:

```
signature TOKENS =
sig
   datatype token =
      KW_and
    | KW_andalso
    | ...

    | KW_type
    | KW_val
    | PLUS | MINUS | ASTERISK | SLASH | PERCENT | TILDE
    | EQEQ | LTGT | LTEQ | LT | GTEQ | GT
    | CARET
    | HASH (* # *) | BANG (* ! *) | COLON_EQ (* := *)
    | LPAREN (* ( *) | RPAREN (* ) *)
    | LSBRACK (* [ *) | RSBRACK (* ] *)
    | LCBRACK (* { *) | RCBRACK (* } *)
    | MINUS_ARROW (* -> *) | EQ_ARROW (* => *)
    | EQ | COLON | COMMA | SEMI | VBAR (* | *) | UNDERSCORE
    | TYVAR_NAME of Atom.atom
    | CON_NAME of Atom.atom
    | VAR_NAME of Atom.atom
    | INTEGER of IntInf.int
    | STRING of String.string

   val toString : token -> string
end
```

The tokens correspond to the various keywords, delimiter and operator symbols, identifiers, and literals. The identifier tokens (`TYVAR_NAME`, `CON_NAME`, and `VAR_NAME`) carry a unique string representation of the identifier. The **`structure`** `Atom : ATOM` module is provided by the SML/NJ Library[1]; you can view the `ATOM` signature and the `Atom` structure implementation at `http://smlnj-gforge.cs.uchicago.edu/scm/viewvc.php/*checkout*/smlnj-lib/trunk/Util/atom-sig.sml?root=smlnj` and `http://smlnj-gforge.cs.uchicago.edu/scm/viewvc.php/*checkout*/smlnj-lib/trunk/Util/atom.sml?root=smlnj`, respectively. The integer constant token (`INTEGER`) carries the value as an arbitrary precision integer (`IntInf.int`). The string constant token (`STRING`) carries the value as a string (`String.string`).

3.1 Errors

To support error reporting, the `LangFScanner.scan` function takes an initial argument of the type `{reportError: string -> unit}`. Characters in the input stream that cannot be recognized as part of a token or whitespace or part of a comment should be reported and discarded. For example, if the input program is the (almost) expression:

```
1 + @ + 3
```

then the scanner should report an error (for example `Error: bad character '@'`) *and* return the following stream of tokens:

```
INTEGER (1)
+
+
INTEGER (3)
```

Any non-printable character that is not whitespace should be reported as an error and discarded from the character stream. Similarly, any printable character that is not part of a token, is not whitespace, and is not within a comment should be reported as an error and discarded from the character stream. Invalid escape sequences in strings should be reported as an error and discarded from the character stream. Finally, string literals and comments that are unterminated at the end of the character stream (that is, when the input character reader returns `NONE`) should be reported as errors.

[1]The SML/NJ Library a collection of utility libraries that are distributed with SML/NJ (and available in some other SML implementations). To access the library, use `$/smlnj-lib.cm` in a CM file and use `$(SML_LIB)/smlnj-lib/Util/smlnj-lib.mlb` in a MLB file.

4 FusionForge and Submission

We have set up FusionForge accounts (using your RIT computer account id (*ritid*)) and an SVN repository for each student on the course FusionForge server (`https://asgard.cs.rit.edu:443`). To *checkout* a copy of your repository, run the command:

```
svn checkout svn+ssh://ritid@asgard.cs.rit.edu/var/lib/gforge/chroot/scmrepos/svn/ritidcc
```

On your first checkout, you should be prompted for your FusionForge password. Upon a successful checkout, a directory called `ritidcc` will be created in the current directory. Over the course of the quarter, homeworks and projects will be stored as sub-directories within the `ritidcc` directory.

Sources for Project 1 have been (or will shortly be) committed to your repository in the `project1` sub-directory. If you checkout your repository before the Project 1 sources have been committed, then you will need to *update* your local copy, by running the command:

```
svn update
```

from the `ritidcc` directory.

We will collect projects from the SVN repositories at 11:59pm on Fri, February 13, 2015; make sure that you have committed your final version before that time. To do so, run the command:

```
svn commit
```

from the `ritidcc` directory.

5 Hints

- Start early!
- To complete the assignment, you should only need to make changes to the *ritid*`cc/project1/langfc-src/scanner/langf-hand-scanner.sml` file.
- Although regular expressions are used to specify tokens for scanner-generator tools and a regular-expression engine is used by a tool-generated scanner, a hand-written scanner can and should be much simpler. (Indeed, few programming languages have a token specification that require complex regular expressions. Nonetheless, the flexibility and efficiency of regular expressions make them a good choice for scanner-generator tools.)
- The **`structure`** `Char : CHAR` module provides basic operations and predicates on characters; you can view the `CHAR` signature at `http://standardml.org/Basis/char.html`. In particular, you may find operations like `Char.chr` and predicates like `Char.isAlphaNum` and `Char.isDigit` helpful.
- The **`structure`** `String : STRING` module provides basic operations on strings; you can view the `STRING` signature at `http://standardml.org/Basis/string.html`. In particular, you may find operations like `String.implode` (which converts a list of characters to a string) helpful.
- The **`structure`** `IntInf : INT_INF` module provides basic operations on arbitrary precision integers; you can view the `INT_INF` signature and `INTEGER` signature at `http://standardml.org/Basis/int-inf.html` and `http://standardml.org/Basis/integer.html`, respectively. In particular, while it is a useful exercise to convert a list of digit characters to an integer value, you may find it simpler to use `IntInf.fromString` or `IntInf.scan`.
- Work on error reporting last. Detecting errors and producing good error messages can be difficult; it is more important for your scanner to work on good programs than for it to "work" on bad programs.
- Executing the compiler (from the *ritid*`cc/project1` directory) with the command

  ```
  ./bin/langfc -Ckeep-scan=true file.lgf
  ```

  will produce a *file*`.scan.toks` file that contains the sequence of tokens returned by the scanner. Use this control and its output to check that your scanner is working as expected. The `tests/scanner` directory includes a number of tests (of increasing complexity); for each `test`*NNN*`.lgf` file, there is a `test`*NNN*`.scanner.soln.toks` file containing the sequence of tokens to be returned by the scanner and, if the test has lexical errors, a `test`*NNN*`.scanner.soln.err` file containing sample error messages to be reported by the scanner.
- A reference solution is available on the CS Department file system at:

  ```
  /usr/local/pub/mtf/cc/project/project1-soln/langfc
  ```

  Use the reference solution to confirm your understanding of the project and to develop additional tests.
- The most difficult tokens to handle are integer and string constants. Nested comments are also somewhat difficult to handle.

Document History

January 30, 2015

Original version