Rust$^-$: A Simple Rust Programming Language

Programming Assignment 1

Lexical Definition

Due Date: 1:20PM, April 17, 2018

Your assignment is to write a scanner for the *Rust*$^-$ language in **lex**. This document gives the lexical definition of the language, while the syntactic definition and code generation will follow in subsequent assignments.

Your programming assignments are based around this division and later assignments will use the parts of the system you have built in the earlier assignments. That is, in the first assignment you will implement the scanner using **lex**, in the second assignment you will implement the syntactic definition in **yacc**, and in the last assignment you will generate assembly code for the Java Virtual Machine by augmenting your **yacc** parser.

This definition is subject to modification as the semester progresses. You should take care in implementation that the programs you write are well-structured and easily changed.

1 Character Set

Rust$^-$ programs are formed from ASCII characters. Control characters are not used in the definition of the language. In addition, sF is case-sensitive, i.e. CASE $\neq$ Case $\neq$ CaSe.

2 Lexical Definitions

Tokens are divided into two classes: tokens that will be passed to the parser and tokens that will be discarded by the scanner (i.e. recognized but not passed to the parser).

2.1 Tokens That Will Be Passed to the Parser

The following tokens will be recognized by the scanner and will be eventually passed to the parser:

Delimiters

Each of these delimiters should be passed back to the parser as a token.

comma	,
colon	:
semicolon	;
parentheses	( )
square brackets	[ ]
brackets	{ }

Arithmetic, Relational, and Logical Operators

Each of these operators should be passed back to the parser as a token.

arithmetic	`+ - * / ++ --`
remainder	`%`
relational	`< <= >= > == !=`
logical	`&& || !`
assignment	`=`
compound operators	`+= -= *= /=`

Keywords

The following keywords are reversed words of sF (Note that the case is significant):

> **bool break char continue do else enum extern false float for fn if in int let loop match mut print println pub return self static str struct true use where while**

Each of these keywords should be passed back to the parser as a token.

Identifiers

An identifier is a string of letters and digits beginning with a letter. Case of letters is relevant, i.e. **ident**, **Ident**, and **IDENT** are not the same identifier. Note that keywords are not identifiers.

Integer Constants

A sequence of one or more digits.

Boolean Constants

Either **true** or **false**.

Real Constants

A sequence of one or more digits containing a decimal point, optinally preceded by a sign ($+$ or $-$), and optionally following by an exponent letter and exponent.

String Constants

A string constant is a sequence of zero or more ASCII characters appearing between double-quote (**"**) delimiters. A double-quote appearing with a string must be written after a **"**. For example, **"aa""bb"** denotes the string constant **aa"bb**.

2.2 Tokens that will be discarded

The following tokens will be recognized by the scanner, but should be discarded rather than passing back to the parser.

Whitespace

A sequence of blanks (spaces), tabs, and newlines.

Comments

Comments can be denoted in several ways:

- *C-style* is text surrounded by "/*" and "*/" delimiters, which may span more than one line;
- *C++-style* comments are a text following a "//" delimiter running up to the end of the line.

Whichever comment style is encountered first remains in effect until the appropriate comment close is encountered. For example

```
// this is a comment // line */ /* with /* delimiters */ before the
end
```

and

```
/* this is a comment // line with some /* and
C delimiters */
```

are both valid comments.

3 Symbol Tables

You must implement symbol tables to store all identifiers. Symbols tables should be designed for efficient insertion and retrieval operations, and hence they are usually organized as hash tables. In order to create and manage the tables, at least the following functions should be provided:

create(): Creates a symbol table.

lookup(s): Returns index of the entry for string s, or nil if s is not found.

insert(s): Inserts s into a new entry of the symbol table and returns index of the entry.

dump( ): Dumps all entries of the symbol table. returns index of the entry.

4 Implementation Hints

You should write your scanner actions using the macros `token`, `tokenInteger`, and `tokenString`. The macro `tokenInteger` is used for tokens that return an integer value as well as a token (e.g. integer constants), while `tokenString` is used for tokens that return a string as well as a token. The macro `token` is used for all other tokens. The first argument of all three macros is a string. This string names the token that will be passed to the parser. The macro `tokenInteger` takes a second argument that must be an integer and `tokenString` takes a second argument that must be a string. Following are some examples:

Token	Lexeme	Macro Call
left parenthesis	`(`	token('(')
IF	`if`	token(IF)
identifier	`ab123`	tokenString(identifier, "ab123");
		tokenString(identifier, yytext);
integer constant	`23`	tokenInteger(integer, "23");
boolean constant	`true`	token(TRUE);

5 What Should Your Scanner Do?

Your goal is to have your scanner print each token on a separate line, surrounded by angle brackets. Each line should be listed along with a line number. In addition, the identifiers that are stored in the symbol table must be dumped. For example, given the input:

```
// Hello World Example
fn main() {
  // Print text to the console
  println ("Hello World");
}
```

Your scanner should output:

```
1: // Hello World Example
<FN>
<id: main>
<'('>
<')'>
<'{'>
2: fn main() {
3:   // Print text to the console
<PRINTLN>
<'('>
<string:Hello World>
<')'>
<';'>
4:   println ("Hello World");
<'}'>
5: }

Symbol Table:
main
```

6 *lex* Template

This template may be found online on the Compilers home page.

```
%{
#define MAX_LINE_LENG 256
#define LIST     strcat(buf,yytext)
#define token(t) {LIST; printf("<%s>\n","t");}
#define tokenInteger(t,i) {LIST; printf("<%s:%d>\n","t",i);}
#define tokenString(t,s) {LIST; printf("<%s:%s>\n","t",s);}

int linenum = 0;
char buf[MAX_LINE_LENG];
%}

%%
"("     {token('(');}

\n      {
        LIST;
        printf("%d: %s", linenum, buf);
        linenum++;
        buf[0] = '\0';
        }

[ \t]*  {LIST;}

.       {
        LIST;
        printf("%d:%s\n", linenum, buf);
        printf("bad character:'%s'\n",yytext);
        exit(-1);
        }
%%
```