CS 4386.001, Compiler Design, Fall 2022
Project Assignment #1

Due: 11:59pm, September 30
Submission: *.tar.xz (or *.zip) via elearning
Maximum points: 100
The submitted file should be named by the assignment, your net id, and your first-last name, e.g., project01-xxx123456-jon-bell.zip

I Introduction

Throughout the semester, we will be creating a compiler broken into several parts. For this first portion, we will be implementing a scanner. We will attempt to integrate cup and JFlex to generate a scanner that follows the rules for what tokens we are looking for and what qualifies as valid identifiers, defined below.

II Cup and JFlex Setup

JFlex: https://www.jflex.de/download.html (Whole website https://www.jflex.de/)
JFlex is a tool where we can identify terminals for our language. This includes single characters, but will also allow us to use regular expressions for complex identifiers.

Cup: http://www2.cs.tum.edu/projects/cup/
JFlex comes with support for Cup which will allow us to represent the grammar as a series of non-terminal productions.

While these tools will work on any operating system, the Makefiles have been written for a linux based system, since that is an OS that every student has access to through UTD servers.

A default project setup is provided ("CompilerProject2.0-Release_0.tar.gz") with both Cup and Jflex jar files, a Makefile to build the tools, and a sample input and output to ensure the build process is working.

III Example

An example project can be found here: https://github.com/pattersonzUTD/UTDLang--
There are 2 points of interest for part 1:

- An incorrect solution
- A correct solution

Both of these will be discussed in project 1 lab

Feel free to use this example project as the starting point for the implementation of the scanner for our language defined below. NOTE: if you copy this project, Makefiles do requires tabs instead of spaces, so the makefile may not work without editing.

IV The Grammar to Implement in the Scanner

Program	→	class id { Memberdecls }
Memberdecls	→	Fielddecls Methoddecls
Fielddecls	→	Fielddecl Fielddecls | λ
Methoddecls	→	Methoddecl Methoddecls | λ
Fielddecl	→	Optionalfinal Type id Optionalexpr ;
	|	Type id [intlit] ;
Optionalfinal	→	final | λ
Optionalexpr	→	= Expr | λ
Methoddecl	→	Returntype id (Argdecls) { Fielddecls Stmts } Optionalsemi
Optionalsemi	→	; | λ
Returntype	→	Type | void
Type	→	int | char | bool | float
Argdecls	→	ArgdeclList | λ
ArgdeclList	→	Argdecl , ArgdeclList | Argdecl
Argdecl	→	Type id | Type id [ ]
Stmts	→	Stmt Stmts | λ
Stmt	→	if (Expr) Stmt OptionalElse | while (Expr) Stmt | Name = Expr ;
	|	read (Readlist) ; | print (Printlist) ; | printline (Printlinelist) ;
	|	id ( ) ; | id (Args) ; | return ; | return Expr ; | Name ++ ; | Name -- ;
	|	{ Fielddecls Stmts } Optionalsemi
OptionalElse	→	else Stmt | λ
Name	→	id | id [Expr]
Args	→	Expr , Args | Expr
Readlist	→	Name , Readlist | Name
Printlist	→	Expr , Printlist | Expr
Printlinelist	→	Printlist | λ
Expr	→	Name | id ( ) | id (Args) | intlit | charlit | strlit | floatlit | true | false
	|	(Expr) | ~ Expr | - Expr | + Expr | (Type) Expr |
	|	Expr Binaryop Expr | (Expr ? Expr : Expr)
Binaryop	→	* | / | + | - | < | > | <= | >= | == | <> | || | &&

There are a few additional rules that are important to note:

- an identifier has a leading letter followed by zero or more letters or digits
- an integer literal has a digit followed by zero or more digits

- a character literal begins with a ', is followed by a single character description and then is terminated by a ', a single character description can be any legal character other than ' or \, to signify those characters they should begin with a \, as in '\'' or '\\', some special characters include '\t' (tab character), '\n' (newline character)
- a floating point literal consists of one or more digits, followed by a decimal point (.), followed by one or more digits
- a string literal begins with a ", is followed by zero or more string characters and ends with a ", the string characters cannot include a newline character, a tab character, a backslash character or a double quote directly, these must be signified using \ (as in \n for newline, \t for tab character, \\ for backslash and \"), so "ab\tcd\n\"" is a string consisting of an a, b, tab character, c, d, newline character and a double quote character
- white space includes space, newline, return and tab characters
- comments are included as follows:
    - on any line the characters \\ start a comment that is ended at the end of that line
    - * opens a comment that continues until the first occurrence of *\

V Goal and Submission

Make sure that all tokens are identified in the grammar above, and that all regular literals and identifiers follow the guidelines above for regular expressions. Several test files are provided (“testfiles.zip”) to run, make sure that each test runs correctly and that all tokens are identified properly. **Upload a tar/zip of your program and the outputs of the test files. Make sure there is a README with directions on how to build and run your program.** (Ideally there is simply a Makefile and “$make run” works.)

VI Grading Criteria

On the eLearning assignment, you can find several test files which test the basics of identifying tokens. These will be run on the files you have submitted, along with a couple of additional tests with more complicated sequences. Your grade will be based on how well you pass these tests.