Project Parts 1 & 2 – Language Design and Lexer
(50 points = 5% of course grade)

Introduction

This first delivery for the project contains two parts:

1. The Cleaning World Language Design (30 points = 3% of course grade)
2. The Lexical Analyzer (20 points = 2% of course grade)

PART 1. LANGUAGE DESIGN

Your language design must consider all the parts of a programming language that would be needed to implement the Cleaning World and the task the Agent must accomplish. This includes, constants, types, variables, control structures, functions, overall program structure. Consider how you want to initialize a Cleaning World: initialize the World in the program itself or read in the World from a file, in which case you will need to think how to handle files in the language.

For control structures, you should consider some form of selection statement and some form of repetition construct, either iteration or recursion.

For functions, I **strongly advise** that you only deal with parameters passed by value and return results, if necessary, of a simple type (not structured). If your functions need to access large values for reading and/or writing, use globals and design appropriate scoping mechanisms. You are, of course, free to disregard that advice.

As you design your language, think that you'll have to implement a parser for it, so your design should think forward to the type of parser you want to work with.

Your language should include syntax for commenting your code.

Given the nature of the task, it is probably unnecessary to allow the program to use multiple files or libraries, but that's really up to you to decide.

DELIVERABLES

1) A design document that briefly introduces your design and lists all language elements in a table specifying, for each element, in the following order:
    a. The lexeme
    b. The regular expression that describes it
    c. The token returned by the lexical analyzer
    d. Any additional information associated with the token (for literals and identifiers)
    e. If relevant, additional explanation about the lexeme and token.
2) A BNF or EBNF description of the syntax of a program in the language you have designed. Each production should be accompanied by a comment (with distinctive syntax) that explains what the production parses/generates.
3) Sample programs that cover your language design's lexemes and syntax.

PART 2. LEXICAL ANALYZER

Thinking about the Lexer

The Lexer converts your text input file into a token stream, which becomes the input to your Parser. In order to use the tokens in the CFG grammar (Part 3 of the project) and make the grammar legible, you will want to use named constants that correspond to a unique numerical ID. See the excerpt "Lexical Analysis (4.2).pdf" from Sebesta's textbook (you'll find it on Canvas with Module 03) and below, in **Lexer Output**, for examples.

As it is tokenizing the text, the Lexer may need to build the literal table, when it encounters literals in the code for the types contained in the language. Normally, a literal table contains string constants that may be used in different places in the program. It is up to you to decide if and how much you will need to use a literal table.

The Lexer also begins adding symbols to the symbol table. The first draft of your symbol table will be very simple. It should be initialized to contain all reserved words. Then, as the Lexer scans the input file, it will look up identifiers and, if they are not reserved words, it will create new entries in the symbol table. Just to look ahead to Part 3 a little bit, you should think that, when you write the analyzer, you may want to distinguish between local variables and global variables in some way. You don't need to handle that right away, but chances are that, in Part 3, you will need to modify the contents of the symbol table a little.

Lexer Implementation Choices

You have some choices in how you build the Lexer. Basically, you can follow one of two general approaches and then choose a particular implementation.

Approach 1: Use a lexical analyzer generator:

- You can work with Java and use the JLex scanner generator (see below for information on getting it and using it).
- You can work use ***lex*** or ***flex***.
- You could look for other Lexer construction tools.

This site gives a list and a comparison.

Approach 2: Build your own lexical analyzer from scratch:

- You can work in Python and use the regular expression facilities in the **re** module.
- Work in some other language that has a good regular expression package.

C has a regular expression library, but *this is not really recommended*. Those who have tried this in the past have not been successful in getting them to work.

Do not build your lexical analyzer by parsing input character by character. The point here is to get you to use existing tools and regular expressions.

If you choose the second approach, i.e., building your owm lexical analyzer, keep in mind that you can think of this program as consisting of an ordered set of “rules” that:

a. Test a regular expression against the input, and

b. If a match is made, execute the action that creates the token and adds whatever information is needed to the literal or symbol table (or, alternatively, throws away the matched information, as in the case of comments or whitespace—except in Python, where indentation counts).

If some of your tokens are substrings of other tokens, (e.g., “=” is a substring of “<=”), then you will want to order your “rules” such that you test for the longer matches first.

Where Does Lexer Output Go?

Regardless of which of the approaches you use, you have a choice of writing the Lexer output (the token stream) to standard output, so it can be piped into the next stage with ‘>’ or redirected to a file from which the Parser will read. Alternatively, you can write the token stream directly to the file and read a file in the next stage (parsing). You can also try to find a way to call the Lexer from the Parser (Part 3) — more below in **Lexer-Parser Interaction**.

Lexer-Parser Interaction

Looking forward again to Part 3, where you will build the Parser, we have seen another mode of interaction between Parser and Lexer in the recursive descent parser: the Parser calls the Lexer to get it the next token. This means that, once you write the Parser, you could have the Parser control the lexical analysis process by calling the Lexer directly (easiest if you use the same language for the Parser and the Lexer). As an alternative, or you can have the Parser call a “Lexer” function that just reads from the file or stream of tokens where a separate Lexer process has placed the token (this is the model where each process runs separately communicating via the standard output stream or a file).

It is okay to use different programming languages for different parts of your project, but you must find a way to make the communicate, whether through a pipe or a file.

Lexer Output

The Lexer output used by the Parser is fundamentally a token stream, but the raw output of the Lexer will also contain additional information. This extra info will not be used by the Parser to check the acceptability of a token sequence according to its grammar but will get stripped by the function called by the Parser to get the next token. However, it is used for error reporting. For example, given an input such as the following one, and assuming a C-like language:

Line #	Statement
1	if (x==0)
2	return 1;

The output could be (see also “Lexical Analysis (4.2).pdf” for another example):

To the Parser (token stream or file)	To the screen (for debugging purposes)
1 34 IF	Line 1 Token #34: if
1 10 LPAREN	Line 1 Token #10: (
1 2 ID x	Line 1 Token #2: x
1 37 EQ	Line 1 Token #37: ==
1 3 INTCON 0	Line 1 Token #3: 0
1 11 RPAREN	Line 1 Token #11:)
2 27 RETURN	Line 2 Token #27: return
2 3 INTCON 1	Line 2 Token #3: 1
2 38 SEMICOLON	Line 2 Token #38: ;

Of the information shown above in the left column:

- The first number is the line number and is helpful for error reporting or for verbose traces of how your Lexer is working.
- The Parser will use only the 2nd and 3rd columns for syntax checking. They represent the unique token type ID and the corresponding human-readable token type. The token-type is mapped to a named constant (e.g., IF, LPAREN, ID, INTCON, RPAREN, RETURN, SEMICOLON), which is more readable by a human and will be used in the grammar rules. The mapping between the numeric token type ID and the human-readable token type is share by the Lexer and the Parser.
- The 4th column, where present, will be put in the symbol table or in the code.

DELIVERABLES

1) A brief **Lexer Report** that says why you chose this specific implementation among the options available and briefly describes the Lexer design.
2) A short **Lexer User Manual** telling you what is needed to run it and specifying assumptions about what files should be present and where.
3) A **Lexer** that correctly and completely analyzes a program in your language and builds the parts of the literal and symbol table that it can.
4) **Lexer output** for the programs in the DELIVERABLES of Part 1.

Grading/Rubrics

Part 1. Language Design

COMPONENT	POINTS
Lexeme Specification	**10**
Completeness of coverage	5
Formal specification (Table)	5
Syntax Specification	**10**
Completeness of coverage	5
Formal specification	5
Overarching	**10**
Principled design	5
Examples	5

Part 2. Lexical Analyzer

CONTENTS	POINTS
Lexer Report & Manual	**5**
Choice justification	1
Brief design description	2
Lexer User Manual	2
Lexer Code Design and Documentation	**5**
Well-structured	3
Well documented (in the file)	2
Lexer Output - Completeness and Correctness	**10**
Punctuation	1
Operators	1
Reserved words	2
Built-in functions	2
User-defined IDs	1
Numeric literals & String literals (if applicable)	1
Comments	1
Whitespace	1
TOTAL	**20**