The University of Newcastle

School of Electrical Engineering and Computing

COMP3290 Compiler Design

Semester 2, 2020

Project Part 1 (20%) Due: August 30th

Project Part 1A - Writing Programs in CD20 (5%)

You are to write a **suite of programs** in the language **CD20**.

This part of the project has 3 main purposes

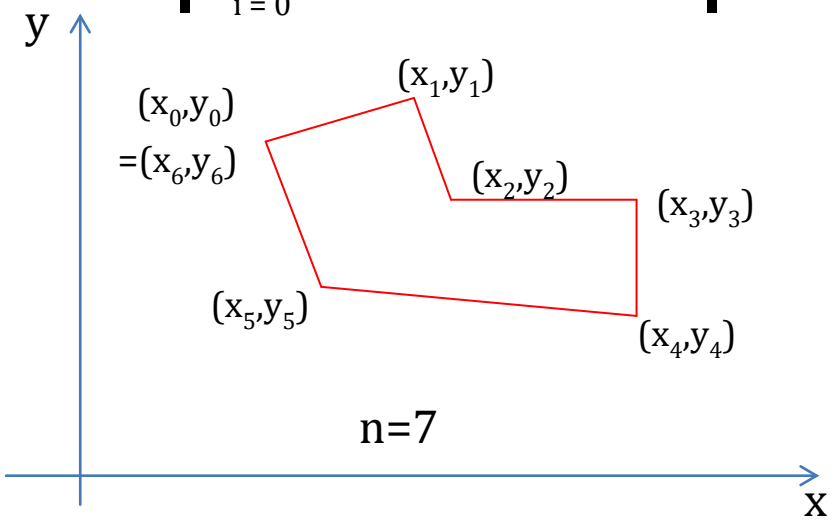
1) To help you to **learn** the CD20 Programming Language.
2) To ensure that you have the beginnings of an adequate suite of **test data** for the later stages of your compiler.
3) To see if you can uncover any inconsistencies in the definition of CD20.

Your suite of programs should contain:

a) at least one that fails syntactically (but succeeds lexically).
b) at least one that fails semantically (but succeeds lexically and syntactically).
c) what you consider to be the simplest possible working program in CD20.
d) several that you expect to run successfully when later compiled by your part 2 and 3. These should include at least one that just has a main program and simple variables, one that has arrays, and one that has function(s)/procedure(s) using all three methods of parameter passing that are allowed in CD20.
e) Write a program in CD20 that will compute the area of a polygon from a set of (x, y) Cartesian points. The program should have functions to (i) input the number of points (up to 20) and the points themselves, and (ii) compute the area of the polygon. The main program should be used to output the result. The operative formula is:

$$A = \tfrac{1}{2}\left| \sum_{i=0}^{n-2} (x_{i+1} + x_i)(y_{i+1} - y_i) \right|$$

e.g.



The submission for Part 1A may be *hand-written and scanned to a PDF*, or *typed as text files*, and submitted via Blackboard.

No program in parts (a) to (d) need be *longer than 1 page of hand-written CD20* code.
Please include comments that explain exactly what the program is doing.

Also include (separate page) comments on any inconsistencies or ambiguities that you may find in the specifications of CD20.

The marker *may* also require you to submit one or more of your programs as simple text files (produced from a simple editor like Notepad, Wordpad, etc.). These will then become part of a standard test suite for the whole class.

Your files for Part 1A will be submitted as part of a single submission archive, detailed in Part 1B next.

Project Part 1B - A Scanner for CD20 (15%)

Write a scanner for the language CD20. Your scanner will be written in Java. The lexical description for CD20 follows, notice that you do not need to know anything of how the tokens are used once your scanner has recognised them. Make sure that you do not try to do syntactic processing in the scanner.

White Space:

CD20 is a free format language. Whitespace characters such as spaces and tabs are lexical delimiters in all cases except within comments and strings. Newline characters are also whitespace, except that they delimit single-line comments and also except that a newline character within a string is a lexical error. When reading from a text file, you may also need to specifically handle the carriage-return character as a whitespace character, depending upon which editor was used to produce your CD20 source file.

Keywords:

These are: **CD20 constants types is arrays main begin end array of func void const int real bool for repeat until if else input print println return not and or xor true false**

The keywords of CD20 are reserved words and so cannot be used as identifier names. They are *not case sensitive* (so ***Begin***, ***BEGIN*** and ***BEgin*** are the same as the keyword ***begin***). The programming convention in CD20, however, is to use keyword variants which only have lower case characters, except for the **CD20** keyword which is uppercase.

Delimiters and Operators:

The following characters and character sequences are used to identify particular language elements. The definitive list can be found in the CD20 Programming Language Specifications. CD20 also has comment delimiters as outlined below.

semicolon (**;**) leftbracket (**[**) rightbracket (**]**) comma (**,**) leftparen (**(**) rightparen (**)**) equals (=) plus (+) minus (-) star (*****) slash (/) percent (**%**) carat (^) less (<) greater (>) exclamation (!) quote (**“**) colon (**:**) dot (**.**). Some of these are combined to form operators such as: <=, >=, **!=, ==, +=, -=, *=** /=. These composite operators may NOT contain embedded whitespace characters, i.e. *equals-space-equals* will be returned as *two equals tokens*.

Comments:

Single line comments may begin with **/--** whereupon they continue until the end of the current line (they are delimited by the next newline character). Multi-line comments start with /** and are terminated by **/ or the end of file

Identifiers and Reserved Keywords:

Identifiers begin with a letter and contain any number of letters and digits. Keywords such as *CD20, constants, types, arrays*, etc are reserved and may not be used as identifier names. All identifiers are CaSE senSItivE, which means, for example, that xModule and Xmodule are different identifiers.

Integer Constants:

Integers contain any number of digits (and therefore can have leading zeroes). Please note that the value associated with the integer token cannot be negative (the string –3 should be returned to the parser as 2 successive tokens *minus* and *3*).

Floating Point Constants:

These follow the usual fixed point structure of <*integer*>.<*integer*>. Like integers real literals cannot be negative.

String Constants:

String constants are sequences of characters enclosed in double-quotes (**"**……**"**) but may not contain newline characters. A string constant which is terminated by a newline character is an undefined (or error) token. There is *no provision* to cater for special characters using a mechanism such as escaping as used in C/C++ and Java. For example a string may not contain **"** character.

Structure of the Scanner:

Constant declarations for the tokens and a string array that allows these values to be printed easily as strings are available via Blackboard under Project Specs.

The tokens returned will minimally be tuples **(tokNo, lexeme)** where **tokNo** is the token value and **lexeme** is usually null, but is a reference to a lexeme string for identifiers, integer constants, floating point constants, and string constants.

Other data that could be useful to error reporting and debugging would be the line number and column number of the token within the input file. Addition of a line and column number makes the token a 4-tuple **(tokNo, lexeme, line, col)**.

The scanner should be a single object, driven by a very small driver of the style:

```
while not scanner.eof( ) do {
   token = scanner . gettoken( );
   scanner . printtoken(token);
}
```

A single message to a scanner object is the best structure because it will become the input processor for your parser and later parts of the project. If you do not follow this design style, it is highly likely that you will need to rewrite major parts of your scanner for **Project Part 2** (the Parser).

Output of the Scanner:

The output of your scanner must be a stream of tokens. You will need to write a special (henceforth useless) debug routine which will print the tokens as they are produced. This debug routine will print to standard output.

The token values should be printed as ascii strings, i.e. print the token value *TCD20* as the string **TCD20**, the *TPLUS* token value as **TPLUS**, etc. There will be a list of token numbers and an associated array of String constants that will print what is required for token values – you will note that these Strings are all 6 characters in length and contain trailing space characters. These tokens must be used in your output. The end-of-file token *TEOF* will be the last token output (as **TEOF**).

Each line of output, in the absence of errors, will exceed 60 characters in length. Once any line of output has exceeded 60 characters then you should terminate that output line.

For identifiers, integers, reals, strings: print the token value followed by the lexeme for the id, integer, real, or string (for strings, output the double-quote characters, even though they are not part for the string itself). This second field is rounded up in length to the next multiple of 6 characters, trailing space filled, and must contain at least one trailing space.

In other words, if you have a row of only tokens, it will extend to 66 characters and then wrap. Or if you have a row that is currently upto 60 characters in length, you will print the next token (and any arguments/values) before wrapping.

For lexical errors: Print the token value **TUNDF** followed by the sequence of characters that constitutes the *undefined token* and then proceed to find the *next valid token* to be returned to the caller. See Error Handling.

Eventually the scanner will have to produce the program listing (it will be the only part of the compiler which will know about comments, for example), so you may like to start thinking about how to do this. It is not required but you'll have to do it eventually. If you decide to produce a special listing file then make sure you dump it out before your program exits. At the end of these specifications you will find some design hints on how to do this in a way that will best benefit later parts of the project.

OUTPUT FORMAT IS IMPORTANT.

Error Handling:

Errors found (e.g. *a hash character which is not within a comment*) will be output as if they are undefined tokens (as outlined above), but they should also be reported separately as an error message from the compiler. This error reporting will survive into later project phases, where it will be augmented with other types of error messages as other errors are found.

Note that this is only the first stage of error reporting - misspelt keywords will be returned as valid identifiers, etc. and these errors may not be found until the parser. Misspelt identifiers may not be found until semantic checking is completed.

Also note that your scanner does not recognise sequences of valid items as being incorrect, even in cases such as <<= (which would be **`tless`** returned, and then **`tleql`** returned by the next request for a token).

A sequence such as /--= would be ignored, with the = being recognised as part of an inline comment (with the rest of that source line also being ignored as comment).

Your scanner only reports lexical errors. When a lexical error is found, an error message is to be printed on a line by itself (the next valid token gets printed on the line following the error report, beginning in col.1). After the TUNDF token is printed, itself on a new line:

```
TDOTT TIDNT seven TLPAR TRPAR
TUNDF
lexical error ?@@#
TIDNT next TIDNT tokens TIDNT here TDOTT
```

For invalid strings or characters print out **lexical error** *X* (where *X* is the offending character or string). When a lexical error is found then the associated "undefined token" incorporates all characters up to but not including the next space, tab, newline, alphanumeric or operator character.

Note that a sequence such as `123abc` could be detected as a lexical error, but for this project, you will follow the "return next valid token" semantics and therefore a string such as this would be returned as two tokens – the integer constant `123`, and the identifier `abc`.

Restrictions:

As a formality, it must be mentioned that you are to write your own Scanner and NOT use any form of *third-party compiler tool* or *library* to achieve this. Additionally you are NOT to use *regular expressions* to match your *keywords*, *numerics*, or other *glyphs*.

Testing Your Scanner:

You are responsible for making up sufficient data files which will adequately test your scanner. There may be a class suite of standard test programs, but this may not be exhaustive for the purposes of testing your scanner.

Note that you do not need to know what the grammatical structure of the language is in order to do this project, you only need to know what constitutes a valid lexical item in the language. If you find yourself consulting the syntax specification of CD20, you are probably going outside these specifications.

It is recommended that you plan out your attack on this project, don't write the whole thing and then go looking for bugs – you will finish up with a mess, impossible to read, understand and extend later. A short while writing (henceforth useless) debug routines will probably save you lots of time later.

Submission:

Project Part 1, is due on **Sunday August 30 at 23:59pm**.

Please zip up all your files and submit them via the **Part 1 submission point** within the assessments tab on **Blackboard**. Your submission will be named with your student number (e.g. **`c7090832.zip`**), and put your name into the associated comment field for the submission. Remember to incorporate an assignment cover sheet into your submission.

You may put your 1A and 1B files in the root folder of the archive.

Keep your own copy of what you submit.

Please ensure that your project can be compiled on the standard *University Lab Java environment*, using the command **`javac A1.java`**, and executed similarly with the command **`java A1 source.txt`**, where source.txt will be specified by the end user (note also, it *may or may not be a txt file*); **do not hardcode this filename**.

If you have enjoyed Part 1B of the Project and want to work ahead …

Listing:

Your scanner responds to a **`nextToken()`** request by returning the *next valid token* and by reporting any errors it finds along the way. The first extension to this is to re-produce the input which has been scanned as a separate listing file. This is just a separate text file which mirrors your input file except that it should have line numbers added at the start of each line. It can also output errors associated with any line after the line has been produced on the listing, or it can save up any errors messages until the end of the program and report them all (with their associated line numbers).

This is best done by sending messages to a separate output object and this output object can be used as a single place for the control of the output of the listing and the output of any error messages. You can call this object *OutputController* and it will respond to messages to *print a source code character*, *report an error*, etc.

Please note that this is separate to any token output required for Part 1B (which goes directly to standard output). If you work ahead with this, you may dump your listing file out to standard output before your program exits.

And even more …

Symbol Table:

Start to think about how specific identifier values, integer and real literal values, and string values should be stored. They will not be *needed* until the later parts of the project, but they will have to be remembered for then and if they are not remembered now, then they can't be resurrected later. If we declare an identifier X, and then refer to it later, then we will have to tell that it is the *same* X. This will be done using a *Symbol Table*, which is another stand alone object (or set of objects) which will allow these lexeme values to be inserted and looked up later. For now it is best to have a hash table that can

insert and look up string values as keys to a simple record/object structure which records the line and column number of where a lexeme is first found and then increments a counter each time it appears in the CD20 source program.

If you build a symbol table then do *not* include it in your Part 1 submission.

DB & NN **v1.0 2020-08-07**