THE UNIVERSITY OF
NEWCASTLE
AUSTRALIA

School of Information and Physical Sciences
COMP3290 Compiler Design
Semester 2, 2024
Project Part 1 (Due: August 16th, 2024)

IMPORTANT!

This Assignment has two parts **Part 1A** and **Part 1B**.

- **Part 1A** must be completed by each student individually and independently from the other member(s) in their group and without using Generative AI (GenAI).
- **Part 1B** to be completed in a group of 2. The same group will continue in the Part 2 and Part 3 of the project. See Canvas for details about the policy and procedure for groupwork management in this course.
- Also note the submission instructions which are different for Group Leader and Group Member. See the **Submission** section below.
- For completing the Part1B, you are encouraged to use Generative AI (GenAI) tools. Please check the document "*Use of Generative AI in Assignment 1*" for information, explanation and possible use cases on how such tools to be used in preparing the solution before you start working on this assignment.

Project Part 1A - Writing Programs in CD24 [50 marks]

You are to write a **suite of programs** in the language **CD24**.

This part of the assignment has 3 main purposes

1) To help you to **learn** the CD24 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 CD24.

Program Suite

Each team member is to complete their own program suite independently. Your suite of programs will contain:

a) at least *two* that fails lexically.
b) at least *two* that fails syntactically (*but succeeds lexically*).
c) at least *two* that fails semantically (*but succeeds lexically and syntactically*).
d) what you consider to be the *simplest possible* working program in CD24.
e) several that you expect to run successfully when later compiled by the part 2 and 3 of your compiler. 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 CD24. Also write a few using different types of loop and branching constructs provided in CD24.
f) Write a program in CD24 that computes the final score and grade of a group of students in COMP3290 and find the student who received the highest score. The program should have functions to:
   (i) input the number of students (up to 20) and for each student their ID (an integer number) and score in each assessment (each out of 100),
   (ii) calculate the final score for each student in COMP3290,
   (iii) print the final score and grade for each student, and
   (iv) print the ID and score of the student who receive the highest score. The main function should be used to output part (iv).

   Your program should also use CD24 structures, array and function. Check the COMP3290 course outline for assessment and grade information.

Source should have some degree of commenting, for clarity.

Format:

The submission for Part 1A will be *typed as text files*, and submitted as part of your .zip submission via Canvas.

No program in parts (a) to (e) need be *longer than 1 page of CD24* code. Please include comments that explain exactly what the program is doing.

These may then become part of a standard test suite for the whole class – you are also free (*even encouraged*) to exchange any of your Test Suite after Part 1A has been graded, throughout the cohort.

If you wish to use a unique extension, you can use **`.cd`** – however at this stage **`.txt`** is perfectly fine.

CD24 Inconsistencies and Ambiguities:

Also include (*as a separate PDF called* **`CD24issues.pdf`**) any comments on any inconsistencies or ambiguities that you may find in the specifications of CD24 in the same folder as your CD24 source files.

Refer to the **Submission** section in Part 1B below, for information on how to submit these source files.

Project Part 1B - A Scanner for CD24 [100 marks]

Write a scanner for the language CD24. The lexical description for CD24 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.

Note: the CD24 Scanner has nothing to do with the Java `Scanner` library – which I encourage you to use.

Input

Input will be text source files, in line with the **CD24 Programming Language Specifications.**

White Space:

CD24 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 CD24 source file.

Keywords:

These are:

```
CD24 constants typedef def arraydef main begin end
array of func void const int float bool for repeat
until do while if else elif switch case break default
input print println return not and or xor true false
```

The **keywords** of CD24 are reserved words and so cannot be used as identifier names. **Keywords** are *not case sensitive* (so `Begin`, `BEGIN` and `BEgin` are the same as the keyword `begin`).

The programming convention in CD24, however, is to use keyword variants which only have lower case characters, except for the `CD24` 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 **CD24 Programming Language Specifications.**

CD24 also has common delimiters as outlined below.

comma (**,**) semicolon (**;**) leftbracket (**[**) rightbracket (**]**) leftparen (**(**) rightparen (**)**) equals (**=**) plus (**+**) minus (**-**) star (*****) slash (**/**) percent (**%**) carat (**^**) less (**<**) greater (**>**) 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 *CD24, constants, typedef, arraydef,* 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 Canvas under CD24 Language Specification.

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)**.

It is *HEAVILY suggested* that your Scanner builds a list of Tokens before progressing to the printing phase – this list of Tokens will also become the basis of the input into your Parser (*Part 2*), either as a call on the Scanner; such as:

```
myScanner.getNextToken();
```

…or by instantiating your Parser with the Token list; such as:

```
myParser = new Parser(myTokenList, mySyntaxTree);
```

If you diverge from these design principles, 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 will 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 (*ie: the terminal window*)

The token values should be printed as ascii strings, i.e. print the token value *TCD24* as the string **TCD24**, the *TPLUS* token value as **TPLUS**, etc. You have been given 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. The end-of-file token *TTEOF* will be the last token output (as **TTEOF**).

Refer to **CD24-Tokens** file for a list of the required Token names and enumeration.

Line Format:

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.

Tuples:

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 up to 60 characters in length, you will print the next token (*and any arguments/values*) before wrapping.

Lexical Errors:

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** section below.

Listing File Production:

The scanner will also have to produce a text format program listing file (*it will be the only part of the compiler which will know about comments, for example*), which will add line numbers for every line, and include identified errors and error messages. You may include these error messages as inline messages (*occurring under the line that contains the error*) or you may block these messages and append them to the end of the listing. See **Error Handling** section below.

This file will be produced in the same location as A1.java, and maybe named the same as the source, but with the extension `.lst`. For more information, see **Listing File Overview** section below.

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* **`TDVEQ`** *returned, and then* **`TEQEQ`** *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 ?@@# (line 2, column 8)
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. Finally you should add the location in the source where this error began.

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`.

Listing File Overview:

The program listing is just a separate text file which mirrors your input file except that it will 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 listing and report them all as a block (*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. ie: this object might be called *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*).

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 CD24, 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.

Report on the use of GenAI:

You should write a 2 to 3 A4 page reflective report on the use of generative AI in competing this assignment. In your reflective note, you should provide a detailed account of your experience and should include, but not limited to, the following elements – (i) overview of the tool(s) used (ii) how the tool(s) was/were integrated in your workflow (iii) what was the quality/correctness of the response generated by the tool(s) and/or how the GenAI tool(s) improved the quality of your solution (iv) how the tool(s) improved your learning experience and problem solving skills (v) what challenges you faced in using those tools and how you solved those challenges (vi) how useful was the tool for your productivity e.g. how much time was saved by using GenAI tool(s) (vii) any ethical consideration or concerns in using GenAI tools in this assessment etc.

Marking breakdown for Part 1B:

If you did not use GenAI tool(s):

1. Scanner Functionality and Code Form: 100

If you used GenAI tool(s):

1. Scanner Functionality and Code Form: 80
2. GenAI report: 20

Submission

Project Part 1, is due on **Sunday August 16th at 23:59pm**.

Although the project is to be completed in group, each student should submit his own assignment. Note the difference between the submission of the group leader and group member.

Group Leader: Group leader should submit the group project (Part 1B), his individual component (Part 1A), group reports (*Group pre-action plan and meeting minutes*), confidential individual reports (*Activities report and Peer evaluation*) and individual report on the use of GenAI tools.

Zip up all your files and submit them via the Part 1 Submission Point within the assessments tab on Canvas. Use a subfolder called **`/CD24Source`** in the root for Part 1A files, and leave your Part 1B files in the root. Your Submission Zip file will be named with your student number (eg. **`c1234567.zip`**).

You will use the subfolder **`/GroupReports`** in the root containing (1) Group's Pre-action plan, and (2) Meeting minutes.

You will create a subfolder **`/PersonalReports`** in the root containing (1) Individual Activity Report, (2) Peer Evaluation, and (3) Report on the use of GenAI. Please note that these last 3 reports are confidential.

Thus, the Group Leader will submit their personal reports with the main submission in their .zip file.

Group Member: Group Member should submit only their individual components (Part 1A), and confidential individual reports (Activities Report, Peer Evaluation, and Report on the use of GenAI) in their .zip.

Zip up all your files and submit them via the Part 1 Submission Point within the assessments tab on Canvas. Use a subfolder called **`/CD24Source`** in the root for Part 1A files, and another subfolder **`/PersonalReports`** in the root containing (1) individual activity report and (2) peer evaluation and (3) Report on the use of GenAI. Please note that these 3 reports are confidential. Your Submission Zip file will be named with your student number (eg. **c1234567.zip**).

Environment:

Please ensure that your project can be compiled on the standard *University Lab Java environment* (*this is currently **Java 17.0**, the current LTS version – you may use ES209 as a verification environment*).

Compilation and Execution:

Compilation and execution will be from the command line terminal – compilation will use 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**.

Your project MUST confirm to these compilation and execution requirements to be valid and able to be marked.

Project Part 2 Introduction

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

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 CD24 source program.

If you build a symbol table, do *not* output anything from it in your Part 1 submission; but it would be wise to factor this into your Part 1B design.