For this assignment, you will use Racket, a functional programming language, to write a simple parser.

Note that we're writing only a parser, not a full interpreter (although interpreters in Racket aren't that difficult[1]). Also, your program only needs to pass a verdict on the syntactical correctness of the program, not produce a full parse tree.

The standard interpreter for Racket is DrRacket, and is installed on all Flarsheim lab machines as well as available as a free download from https://racket-lang.org/.

Racket is a functional language, so it requires a different approach than what you may be used to. It's expression-oriented; functions take in parameters and return function values, with no side effects. I'd suggest spending some time getting familiar with it, perhaps working out some common algorithms in a functional form. You should also familiarize yourself with the Maybe and Either types.

Your code should have a function called `parse`, which takes one parameter—the name of the file of source code to be processed. From the input line of the DrRacket environment:

```
(parse "source1.txt")
```

It should return a string: either "Accept", indicating the program is syntactically correct; or a message as to which line the first syntax error was found.

So output will be either:
`Accept`
or something like:
`Syntax error found on line 25`

In the case of a syntax error, printing the offending line would also be helpful. It is not necessary to continue scanning for additional syntax errors.

Coding considerations:

- One of the main goals of this assignment is to give you some experience with functional programming. Thus, **your program is expected to have most (70% or more) of functions return either a Maybe or Either type.** (See the notes on Functional Programming for more information about these monadic types.) The project will be easier if you choose one or the other of those forms and use it consistently. Either is more flexible than Maybe, but slightly more complex to use.
- Your source code should be in a single file, should read the input file from the default directory, and should not require/assume a particular structure of subdirectories.

Use of Large Language Models and other AI tools (ChatGPT, GPT-4, GitHub CoPilot, etc): These tools are being adopted rapidly by industry, and are allowing significant gains in productivity, with the largest gains accruing to the least experienced developers. At the same time, they raise significant issues of intellectual property, bias in results, and equity of access. In general, I do not like to make rules I have no hope of enforcing. Therefore, use of these tools is allowed for this project, to any extent

1 There's an online book, *Beautiful Racket*, that shows how to write an interpreter for the Basic language, along with a couple of specialized languages.

desired, in any manner you please, **subject to the following limits,** which are similar to requirements of industry:

- **Use of these tools must be documented.** You must list which tools were used, and briefly describe what they were used for (initial brainstorming, problem decomposition, first drafts of functions, generating test data, etc).
  - Code produced by large language models tends to have certain peculiarities and is not hard to identify. If you claim to have written everything yourself without these tools, and submit bot-generated code…. That's academic dishonesty. Again, *there's no problem with using these tools, just document what you're doing.*
- **You are still responsible for submitting working code.** While these tools are good at solving the blank-screen problem and can write code for CS-101 level problems, code for more advanced problems often has errors. These tools are **not** good at assessing the correctness of their own code.
- **Your prompts are part of your deliverables.** Your work for this project is expected to be your own, original, unaided work. Code generation by these tools is not deterministic: Two people entering the same prompts to the same tool will get slightly different code. However, with these tools, the prompt engineering—working out how to goad the LLM into producing code that's 'close enough' to what you want that you can modify it from there—is part of the process. Your prompts should be your own original work, and can be (and **will be**) checked for plagiarism. You do not need to submit the LLM's responses or code; only your prompts.

The grammar you are to use is listed in an attached document. This is a simplified version of an ALGOL-derived language. Statements can be (optionally) labeled. There can be more than one statement on a line. You have selection (if-then) but not alternatives (if-then-else, case/switch). You have one iteration construct, `while`, with an explicit end-of-loop marker. There is only one level of arithmetic operator (which would make the language less writeable, but easier to parse). You must define the FIRST, FOLLOW, and PREDICT sets.

Deliverables. You will submit:

- The text file containing your source code. (A Racket source code is a text file with the file extension .rkt) Your code will be tested by running it using some additional files of source code.
- A document file (MS-Word or LibreOffice) listing all references or outside sources you consulted, including any from which you took code that you modified or used in your program.
- A document file (MS-Word or LibreOffice) including your LLM prompts. You do NOT need to include the LLM responses. Include at the top of the file which model(s) you used.

Programming notes:

- You will need other functions besides `parse`, of course. This will be a top-down recursive-descent parser. You will need one function per nonterminal in the grammar. Of course, these may need further decomposition.
- The single priority level for arithmetic operators leaves us open to potential ambiguity, as a leftmost derivation and rightmost derivation will no longer be equivalent. This will not matter for the assignment, as you need only determine whether a line of code follows the syntax, and do not need to build the parse tree. Also, we can deal with the problem simply by declaring that arithmetic operations will be carried out left to right, so a leftmost derivation is correct.
- You're not limited to printing *just* a final verdict; progress messages will probably be helpful in development.

- Submit your source code (.rkt file) and a short document listing any resources you used in developing your program. (This refers to things where you copied & modified someone else's code, used a workaround verbatim, or something like that. General references for Racket don't need to be listed.) You can either upload your files, or submit a link to a repository on GitHub, GitLab, BitBucket, or other service.
- In addition to the references at the Racket website, you may find some other programming resources helpful:
  - *Structure and Interpretation of Computer Programs* has examples in Scheme, Racket's parent language; all of its examples will run in Racket without modification. The section on symbolic data (2.3, 2.4) will have some good examples of working with abstract data types.
  - Two good general-purpose books on Racket programming are:
    - **Racket Programming the Fun Way: From Strings to Turing Machines,** by James W. Stelly (No Starch Press, ISBN-13: 978-1-7185-0082-2 (print) ISBN-13: 978-1-7185-0083-9 (ebook)); or
    - **Realm of Racket: Learn to Program, One Game at a Time,** by Bice et al. (Also by No Starch Press, ISBN-13: 978-1-59327-491-7).
    - Stelly's book draws mostly on recreational mathematics and classical computer-science problems; Bice et al. focus on game development, from simple guess-my-number games to multiplayer client-server games.