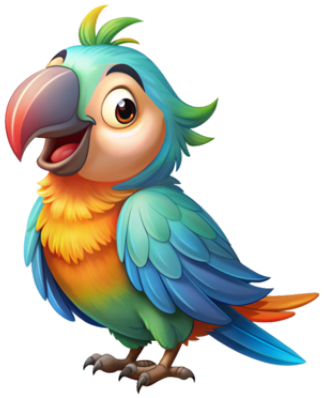
CS202 Spring 2025 - Assignment 1

Translating a Parrot Program.



1 Description

This assignment is designed to refresh and reinforce your C++ programming skills by constructing a program that simulates the execution of a simplified low-level programming language called Parrot (a made up language). Through this task, you will:

1. Review key C++ concepts such as syntax, variables, data types, operators, control flow, functions, input/output, and debugging.
2. Gain a deeper understanding of how a high-level language like C++ translates into low-level machine instructions.
3. Develop logical and critical thinking skills by implementing a two-stage translation process.

The assignment requires you to:

1. Simulate the actions your computer performs when running a C++ program.
2. Utilize memory management techniques, instruction parsing, and the fetch-decode-execute cycle to interpret and execute code written in the Parrot language, with memory represented and managed as an array.

Features of the Parrot Language

1. A set of 18 instructions (see Table 1) for various operations such as arithmetic, comparison, and control flow.
2. An integer-based memory model represented by an array of size 1000.
3. Instructions stored in memory as integers with the first two digits representing the operation and the last three digits representing the memory address.

2 Collaboration

Students are encouraged to collaborate within the boundaries outlined in the syllabus. However, it is crucial to emphasize that the primary goal of these assignments is to develop the fundamental skills necessary for success in computer science. As such, students are expected to independently write their own code, ensuring it is entirely free from plagiarism—whether from classmates, previous students, or others. **To uphold academic integrity, all assignments will be checked for similarities across all sections of CS202 using the Measure of Software Similarity (Moss) tool, which automatically identifies program similarities. While collaborative discussions and study sessions are encouraged, the work you submit must be your own. Assignment submissions will be scanned for plagiarism via Moss. Sharing or copying code constitutes a violation of academic integrity. Moss can detect similarities even if variables are renamed, source code is reformatted, or functions are reordered.**

3 Prohibited Use of AI

Students are not allowed to use advanced automated generative AI tools on learning activities in this course. Each student is expected to complete each assignment without substantive assistance from others, including automated tools. The student is expected to:

- Do their own work and cite any sources you use properly.
- Not use any generative AI tools such as, but not limited to, chatbots, text generators, paraphrasers, summarizers, or solvers, to complete any part of your coursework.
- Consult with their professor before submitting work if there are any questions about what constitutes acceptable use of generative AI tools.

4 Banned Tools

The use of the following, or similar, tools is banned in this course when working on assignments, exams, or any other deliverable content: ChatGPT, Repl.it, Co-pilot, Gemini. Note that repl.it or any of these tools may be used by the instructor during lecturing to easily share code with students, but may not be used by students when working on assignments due to the assisted autocomplete software (all autocomplete are banned) and it's lack of user privacy. For example, in Repl.it anyone can see your code which means they can submit it as their own, which will then be caught by MOSS Anticheat and you will be found guilty of sharing code with students. **If anyone is caught using these tools as in the previously mentioned scenario, they will receive an automatic F in the course and be banned from retaking CS202 for the duration of 1 year. The student is resposible for their code. Ignorance or Negligence is not an acceptable excuse. Your code, your responsibility. When in doubt please consult with your instructor and report any suspicious activity.**

5 Task

Objectives

1. Implement a two-stage translator:
   - First Stage (Preparation Stage):
     - Read and parse the Parrot program file.
     - Remove comments and allocate memory for labels.
     - Generate an intermediate representation of operations and operands in a temporary file.
   - Second Stage (Execution Preparation):

- Read the intermediate file.
- Translate symbolic operands into memory locations.
- Encode instructions into integer memory representations.
- Store the results in an array of integers for execution.

2. Execute the instructions stored in memory using a fetch-decode-execute cycle.

Actions to Take if Uncertain

- Refer to Table 1: Verify the operation codes and their meanings.
- Check Input Format: Ensure the input file adheres to the specified structure (labels in column 1 or 3, instructions in column 2).
- Debug Intermediate Files: Inspect the temporary file generated during the first stage to confirm correctness.
- Consult CodeGrade Feedback: Address errors and resubmit the solution if needed.

6 Skilled Practice, Knowledge Gained, and Long-Term Relevance

Skilled Practice

- Parsing and processing structured text files.
- Translating between symbolic and numeric representations.
- Memory management in programming.
- Designing and implementing a fetch-decode-execute cycle.
- Debugging complex systems with multiple stages.

Knowledge Gained

- Hands-on experience with low-level language translation.
- Enhanced understanding of the relationship between high-level programming and machine-level execution.
- Improved problem-solving skills in translating abstract specifications into executable code.

Long-Term Relevance

- Prepares you for courses and tasks involving compilers, interpreters, and operating systems.
- Strengthens foundational skills in algorithm design and systems programming.
- Builds practical debugging techniques applicable to larger projects.

7 Criteria for Success

1. Input Handling:
   - Proper parsing and cleanup of input files, including removal of comments.
   - Correct identification of labels and instructions based on column placement.
2. Intermediate File Generation:

- Accurate representation of operations and operands in the temporary file.
- Handling errors such as undefined labels or invalid instructions.

3. Translation and Encoding:
   - Correct translation of symbolic operands into memory addresses.
   - Accurate encoding of operations into integer representations.
4. Execution:
   - Proper implementation of the fetch-decode-execute cycle.
   - Correct output matching the provided solution in CodeGrade.
5. Code Quality:
   - Clear and well-documented code.
   - Efficient use of memory and computational resources.
   - Error handling for edge cases and invalid inputs.

8 Checklist/Rubric

By following these guidelines and criteria, you will develop a solid understanding of basic programming concepts and their application in solving real-world problems

- ☐ To achieve the highest marks, you must prevent errors. Good programming practice dictates that your program should proactively avoid errors whenever possible, detecting them when necessary.
- ☐ All declared variables must be used.
  - ☐ Do not use global variables unless specified in the assignment.
  - ☐ Use local variables whenever possible.
  - ☐ Choose meaningful names for your variables. While I acknowledge that some of my variable names, like `r` for register, were not ideal, it is acceptable in this case since I used them solely to explain the assignment.
- ☐ Use proper indentation and include comments in your code. Comment your source code according to the guidelines provided in the rubric. Please format your program using a text editor (or an IDE, if available) to ensure it is readable. If the format is not clear, we will be unable to assist you and will ask you to fix the formatting before proceeding.
  - ☐ Each line of code that is at the same block level needs to have the same indentation.
  - ☐ Inner blocks must be indented by at least two spaces compared to the outer block.
- ☐ Do not attempt to do everything at once. I repeat, do not try to tackle everything in one go. Instead, focus on writing one execution routine (or function) at a time, testing it thoroughly before moving on to the next. In fact, it is a good practice to write a few lines of code, then test to ensure your program compiles and runs correctly.
- ☐ Make sure your program is visually appealing by neatly formatting the output and error messages. Ensure your output matches the given sample exactly, as any deviations could negatively impact your grade.
- ☐ **We will test your code with different inputs, so hard-coding the output will be detected. If you do this, you will receive a score of zero for the assignment. Please note that this action will be regarded as academic misconduct.**

9 Assignment

The goal of this assignment is to develop a program that simulates the steps your computer performs after translating a C++ program into machine code that your computer can understand. For this exercise, we will use a custom, low-level language called Parrot. Since Parrot is a low-level language, it requires a translator. To execute a program comparable to C++, you will need to complete two key passes.

The first pass involves reading the Parrot source code, which includes removing comments and allocating memory for each label (we will discuss this further shortly). During this pass, you will also generate an intermediate representation of the operations and operands, which will be written to a temporary file. The second pass takes the temporary file created by the first pass, translates the symbolic operands into memory locations calculated earlier, encodes the operations and operands, and then stores the result in an integer array. This array represents the program, and executing it simulates the steps your computer would take to run the original C++ program.

There are three main aspects to how your translator should behave:

1. **File Parsing:** Your translator should read a valid Parrot program from a file, clean it up (i.e., parse and remove unnecessary elements), and then output the modified version of the program to a file.
2. **Output Consistency:** The output produced by your translator must exactly match the output of the provided solution in the code grading system.
3. **Translation Accuracy:** Your translator must correctly follow the specification outlined below. Additionally, it should be able to execute a series of instructions, which are listed in Table 1. If the output generated during the first pass is incorrect, your translator will fail to work, as it will not be able to execute the program properly.

Translation Specification

You will assume that your computer uses a variable called *register* (a temporary storage used to move data around), 18 instructions (as outlined in Table 1), and an array of integers to represent the system's memory (RAM). Since memory is limited to integers, and you have a fixed amount of memory, you can only store up to 1000 values at a time. Exceeding this limit will result in an execution error, such as an array index out of bounds issue. To keep things simple, we will refer to integers as *words*. A *word* is essentially an **int** in this context, and it consists of 5 decimal digits. The first two digits represent the operation code (opcode), and the last three digits are used as the address, indicating a specific location in memory where data is stored.

Table 1: Parrot language instructions

Opcode	Instruction	Meaning
00	final	Define a constant value or memory location
01	read	Reads a number from input
02	write	Writes data to terminal
03	load	Puts data from memory into the register variable
04	set	Stores register value in memory
05	plus	Adds memory value to the register variable
06	minus	Subtracts memory value from the register variable
07	times	Multiplies memory value with the register variable
08	divide	Divides memory value by the register variable
09	compare	Compares memory value with the register variable
10	js	Jumps if register value is positive
11	jz	Jumps if register value is zero
12	j	Jumps unconditionally
13	jl	Jumps if register value is less than zero
14	jle	Jumps if register value is less than or equal zero

15	jg	Jumps if register value is greater than zero
16	jge	Jumps if register value is greater or equal zero
17	skip	Terminates program execution

For example, consider the following snippet of Parrot code (i.e., the instructions), as shown in Table 2.

Table 2: A `while` loop in Parrot

Index	Memory Location	Instructions		
0	03015	#init	load	one
1	04017		set	i
2	12006		j	#loop
3	03017	...		
...	...	...		
15	00001	one	final	1
16	00005	five	final	5
17	00000	i	final	

The above table describes a while loop that iterates from 1 to 5 and introduces the concept of how a Parrot program can be visualized as a table with three columns:

1. **Index:** Represents the index of each chunk of memory.
2. **Memory Location:** Represents an array of integers containing the addresses of each memory chunk.
3. **Instructions:**Represents the code executed by the computer.

Although we write code in high-level languages like C^{++}, it must be translated into a form the computer can understand. Table 2 illustrates something similar to what the computer "sees" and executes. For example, if we examine the data stored in the memory location at index 0, we find `03` (interpreted as `load`) and `015` (interpreted as the index of the data to retrieve from memory). Retrieving data from memory involves transferring it to a register variable, which we will call `r` in this explanation. This process demonstrates how data moves between memory and the register. However, this approach is inefficient because there is only one register variable (`r`). To address this inefficiency, direct memory access is permitted for various arithmetic operations (e.g., `plus`, `minus`, `times`, `divide`, etc.) and for the comparison instruction `diff`, provided that the register variable (`r`) already contains data.

Let us use the `times` instruction as an example. Multiplying two operands involves the following steps:

1. Load the value of the first operand into the register variable `r` (this value is retrieved from an address in the memory location).
2. Access the value of the second operand directly from the same or a different memory address.

Here is how this might appear:

```
load x
times x
```

Although `x` appears twice in the instructions, only a single copy of `x`'s value is assigned to `r`. The load instruction retrieves the value of x from its memory location, copies it, and assigns it to the register variable `r`. This process is illustrated below, where @ represents the label address of `x`.

input		equivalent to		interpretation
`load x` `times x`	$\rightarrow$	$r = x$ $r = r + x$	$\rightarrow$	$r = memory[@]$ $r = r + memory[@]$

Moreover, the value of `x` remains unchanged in memory during the operation. To update the value of `x`, we need to move the updated value from the register variable `r` back to memory. This process requires the following operation, as demonstrated below:

input		equivalent to	interpretation
`load`	`x`	$r = x$	$r = memory[@]$
`times`	`x`	$r = r + x$	$r = r + memory[@]$
`set`	`x`	$x = r$	$memory[@] = r$

Let uss examine another example. Consider the following 'if' statement in C++:

```
if (x > 5) {
    // ...
}
```

This code would be translated into Parrot as:

```
             load       x              ; Load the value of x
             compare    five           ; Compare x with 5
             jl         done           ; If x <= 5, go to "done"
             ...                       ; Code to execute if x > 5
 done        ...                       ; Code to execute after the if-statement
```

To skip the statement "guarded" by the conditional expression, you can force a jump if the condition evaluates to true or allow execution to fall through if it evaluates to false. But how should the Parrot code for such a case be interpreted? The key detail lies in the compare instruction. This instruction always precedes a jump instruction (refer to Table 1 for the list of jump instructions). Since we are already familiar with the load instruction, we know that by the time the compare instruction is executed, the register `r` will already hold the value of `x`. All that remains is to subtract the comparison value (5, in this case, stored in memory) from `r`. This subtraction is crucial because it ensures the register retains a result that the jump instruction (e.g., `go<`, meaning "jump if less") can evaluate to decide whether to proceed to the "done" label or continue executing.

input		equivalent to	interpretation
`load`	`x`	$r = x$	$r = memory[@]$
`compare`	`five`	$r = r - 5$	$r = r - memory[@]$
`go<`	`done`	$ip = done$ `if` $r < 0$	$ip =$ @ `if` $r < 0$

It is essential to understand that @ refers to different elements depending on the context in which it is used:

1. It indicates the memory position where the value of x is stored.
2. It represents the value of 5 (used for comparison).
3. It specifies the location to jump to if $r < 0$.

This makes sense because instructions are executed sequentially, meaning @ can only represent one thing at any given time. This distinction is particularly important when compared to the earlier example involving the `times` instruction. When comparing values, the process requires subtracting one value from another, resulting in a value that is either:

- Less than zero,
- Equal to zero, or
- Greater than zero.

Understanding this flow is critical for determining the condition outcome and deciding whether a jump instruction is executed.

Computing Addresses

To compute address, you will use the address of the labels along with the opcodes' mnemonic values as follows:

```
memory[nextMemory++] = MEMORY_LEN * opcodes[<instruction>] + labels[<label>]
```

where *instruction* is any opcode value from Table 1, and *label* is any label that appears in the Parrot file. For example, for the command `load one`, the address is computed as follows

$$\begin{aligned} \text{memory location} &= 1000 \times 3 + 15 \\ &= 3015 \end{aligned}$$

where 1000 is the size of our array, 3 represents the `load` instruction, and 15 is the index where the value `one` can be found. Now, in order to obtain the address of label `one`, we would mod 3015 (the memory location) by the size of the array as follows

$$\begin{aligned} \text{label address} &= 3015 \ \% \ 1000 \\ &= 15 \end{aligned}$$

and in order to obtain the opcode value, we would simple divide 3015 by the array size

$$\begin{aligned} \text{opcode value} &= 3015 \ / \ 1000 \\ &= 3 \end{aligned}$$

Clearly, we can get the first one by combining the last two

memory location	
3015	
opcode value	label address
03	015

During the second pass of the translator, it is crucial to calculate each memory location accurately. The translator plays a key role in determining the final two memory locations, as they are only required during the code interpretation process.

Basic Computation

Your translator should be capable of processing any number of commands and handling errors as needed. To achieve this, you will employ a three-step evaluation process: *fetch*, *decode*, and *execute*.

1. **Fetch:** Retrieve the next instruction to be executed from the current code.
2. **Decode:** Identify and interpret the fetched instruction.
3. **Execute:** Carry out the action specified by the instruction's semantics.

The translator's primary task is straightforward: it should continuously loop through the fetch-decode-execute cycle. The pseudo-code below outlines the general approach you need to implement and complete.

Code for the first and second pass goes inside the main function.

1. Set "register" and "instruction pointer" to 0
2. For as long as the "instruction pointer" is greater than or equal to 0 do
   (a) Set "address" the address at memory index "instruction pointer"
   (b) Set "code" to instruction value at memory index "instruction pointer"
   (c) If code is equal to `load` then execute instruction
   (d) Else If code is equal to `set` then execute instruction
   (e) ...

(f) Else If code is equal to `skip` then set "instruction pointer" to -1

(g) else set "instruction pointer" to -1

The above is basically a large case of **if-else** statements with a conditional expression for each if-statement that checks each instruction.

Input

An input file may contain single-line comments, each beginning with a semicolon (`;`), which must be removed to avoid undefined behavior during the execution of your interpreter. Labels always appear in column one but can occasionally be found in column three, while operators or instructions are consistently located in column two. To process the input effectively, it is advisable to define a field separator variable during the translator's first pass, which can identify all combinations of blanks and tabs within the input line. Leading whitespace, as well as whitespace between fields, should be treated as separators. By using these separators, you can accurately determine the position of labels and instructions, ensuring proper parsing and interpretation of the file's content.

Consider the following Parrot code snippets. The first snippet contains two labels: one in column one and the other in column three.

```
+-------------+----------+----------+
|           Instructions             |
+-------------+----------+----------+
|    #init      load       one      |
+-------------+----------+----------+
```

The second snippet has a label only in column three.

```
+-------------+----------+----------+
|           Instructions             |
+-------------+----------+----------+
|               load       one      |
+-------------+----------+----------+
```

In both examples, the `load` instruction appears only in column two. Therefore, when processing and parsing an input file similar to the second snippet, remember that opcodes will always be found in column two.

Specification

From a design perspective, and to maintain simplicity, we will use a struct to organize a collection of related or unrelated items – this is a logical and practical choice. The information stored will include symbols and opcodes.

1. Write a forward declaration of a Symbol struct, which should include:
   - Label Name: The name of a label.
   - Memory Location: An integer representing either the memory location where an instruction should jump to or the location of a value in memory.
2. Write a forward declaration of an Opcode struct, which should include:
   - Opcode Name: The name of an opcode.
   - Mnemonic Representation: An integer representation of the instruction.

You are only allowed to use the following constant global variables:

- `const int LABEL_LEN;`
- `const int OPCODE_LEN;`

- `const int MAX_CHARS;`
- `const int MEMORY_LEN;`
- `const std::string OPCODE_LIST;`

Your program must include the following functions:

1. **breakOpcodes:** A function that splits a string containing multiple instructions into individual words, each representing a distinct instruction. These words are then stored in an array of opcodes. The input string consists of all the instructions combined. (Hint: Use the global variable `OPCODE_LIST` to assist with this.)
2. **isOpcode:** A function that checks if a given string matches any instruction in the array of opcodes. It returns `true` if a match is found.
3. **getOpcode:** A function that returns an integer corresponding to the value of a string that matches one of the opcodes in the array. If the string is not found, the function returns -1.
4. **getLocation:** A function that determines the memory location of a label. It returns the label's location if found or -1 if the label does not exist.
   - This function is used during the second pass of the translation.
   - It takes the name of a label, an array of labels, and the position of the next label in memory.
   - It checks if the given label exists in the array of labels.
5. **isNumber:** A function that checks if a string represents a valid number. It returns `true` if the string can be interpreted as a number. (Hint: Since some labels may represent numeric values like 1, 10, etc., ensure you correctly transform such strings into integers.)

Additionally, four utility functions have been provided for testing and debugging your program's output:

- pad
- convert
- dumpOpcodes
- dumpSymbols
- dumpMemory

Restrictions on Including Additional Libraries

You are not allowed to include any additional libraries or external files in your program. Everything you need to complete this assignment has already been provided in the starter code or is part of the standard functionality required for the task. Do not include anything else. Including any libraries or files beyond what is explicitly provided or required for the assignment will result in severe consequences, including a failing grade for the course. We have instructed the TAs to carefully review all submissions for unauthorized inclusions.

This restriction is in place for several important reasons:

- **Sufficient Resources Provided:** The starter code and the guidelines contain all the necessary functions, utilities, and structures to complete the assignment. You do not need any additional tools or libraries to implement the required features.
- **Learning Objectives:** The goal of this assignment is to reinforce your understanding of core C++ programming concepts, including string manipulation, arrays, control structures, and functions. By restricting additional libraries, we ensure that you focus on these concepts without relying on shortcuts or pre-built solutions.

- **Standardized Testing:** The provided framework ensures that all submissions are tested under identical conditions. Introducing additional libraries could lead to inconsistencies in behavior or compatibility issues, making it difficult to evaluate your program fairly and accurately.
- **Avoiding Security Risks:** Allowing external libraries introduces potential risks, such as unsafe code or dependency vulnerabilities. By prohibiting unauthorized inclusions, we maintain a secure and controlled environment for evaluating your work.
- **Encouraging Creativity Within Constraints:** Programming often involves solving problems within a set of constraints. By adhering to the provided resources, you develop critical problem-solving skills that are directly applicable to real-world scenarios.

Examples of Violations

To ensure clarity, here are some examples of what you must not do:

- Do not include external libraries, such as `<vector>`, `<algorithm>`, `<cmath>`, or any others not explicitly mentioned in the assignment instructions.
- Do not import third-party frameworks, plugins, or dependencies.
- Do not modify the provided files beyond the scope of the assignment requirements.

Consequences of Non-Compliance

Failure to adhere to this rule will result in:

- An automatic failing grade for the course.
- Submission flagged for review by academic integrity policies.
- Permanent record of the violation with possible further disciplinary actions.

Final Reminder

Everything you need to succeed in this assignment has already been given to you. Focus on understanding and using the provided tools effectively. Do not jeopardize your academic standing by attempting to include unnecessary or unauthorized elements in your code.

If you are unsure about any aspect of the assignment, please consult your instructor or TAs for clarification.

Agreement

By submitting your assignment to CodeGrade for grading, you are agreeing to the terms and conditions outlined in this assignment. This agreement signifies your acknowledgment of the rules, requirements, and restrictions stated within the assignment instructions. Furthermore, you understand and accept the actions that will be taken if you fail to follow these instructions. This includes, but is not limited to, compliance with restrictions on including unauthorized libraries or external files, adherence to submission deadlines, and ensuring your program adheres to the specified criteria for success.

Any violation of these terms – whether intentional or accidental – will result in consequences such as a failing grade for the assignment or course, as well as potential review under academic integrity policies. By proceeding with your submission, you affirm your commitment to completing the work in accordance with the provided guidelines and confirm that your submission is your own, original work, completed within the parameters established by this assignment. If you are unclear on any aspect of these terms, it is your responsibility to seek clarification before submitting your work.

10 Submission

Save your solution as **main.cpp** and submit it to CodeGrade before the deadline. Use the feedback provided by CodeGrade to identify and correct any errors, then resubmit if necessary.

11 Sample Outpus with Correct Values

Let us consider the following Fibonacci sequence written in Parrot.

```
; compute the Fibonacci sequence
; Approach:
; fibonacci($n) = $n if $n=0 or $n=1
; fibonacci($n-2) +
; fibonacci($n-1) if $n>=2

        load    zero    ; $first = 0
        store   first
        load    one     ; $second = 1
        store   second

        read            ; get $n from the user
        store   n

        load    one     ; $i = 1
        store   i
#loop   load    first   ; display $first
        write
        plus    second  ; $temp = $first + $second
        store   tmp
        load    second  ; $first = $second
        store   first
        load    tmp     ; $second = $tmp
        store   second
        load    i       ; $i++
        plus    one
        store   i
        compare n       ; if $i <= $n
        jle     #loop   ; go back and compute the next Fibonacci number

        skip            ; terminate the program
zero    final   0
one     final   1
first   final
second  final
tmp     final
n       final
i       final
```

You are required to remove all comments, track labels, store instructions or opcodes along with their addresses, and then write the processed code to an output file. This output file will later be processed during the second pass of your translator. The generated file should be free of comments, and any labels that appeared in the first column must also be removed. While the exact format is not crucial, ensure that each line maintains a clear separation between instructions and labels by leaving a space between them. Below is an example of what the resulting output file should look like:

```
load zero
set first
load one
set second
read
set n
load one
set i
load first
write
plus second
set tmp
load second
set first
load tmp
set second
load i
plus one
set i
compare n
jle loop
skip
final 0
final 1
final
final
final
final
final
```

The output generated from the first pass of your translator will serve as the input for the second pass. During the second pass, your translator must encode the processed instructions and labels into an array of integers. This array acts as the simulated memory, representing the storage location where all commands – including both instructions and labels – are organized and stored.

```
$> g++ main.cpp fibonacci.parrot
$> ./a.out

Running program...
10
read: 10
result: 0
result: 1
result: 1
result: 2
result: 3
result: 5
result: 8
result: 13
result: 21
result: 34
** Program terminated **
```