# ASSIGNMENT 2

CS 473: Compiler Design / Fall 2025

## Description

In this assignment, you will implement a syntactic parser for the language Cardinal. The language is described in a separate document. Your syntactic parser should reject programs with invalid programs and print a disambiguation of the expressions for valid programs. You should use the lexical parser that you implemented in Assignment 1 to generate tokens for Assignment 2.

## The Cardinal Language

The Cardinal language is defined in a separate document.

## Format of the output

Your syntactic parser should output each recognized top-level expression in a different line, surrounded by parenthesis ( ) to disambiguate precedence and associativity. It should also highlight blocks by surrounding them with brackets { }.

For instance, consider the following valid program in Cardinal:

```
while (1 <> 2 / 3 + 4) {
  if (5 == 6 % 7)
    return 8 + 9 - 10 * 11 % 12;
  return 13;
}
return 14;
```

The expected output for this program is:

```
((1)<>(((2)/(3))+(4)))
((5)==((6)%(7)))
(((8)+(9))-(((10)*(11))%(12)))
(13)
(14)
```

## Statements

- **return**: print the expression on the value in a new line
- **if**: print the expression on the condition in a new line, followed by all the statements in the then branch (as described in this document), followed by all the statements in the else branch when present
- **while**: print the expression on the condition in a new line, followed by all the statements in the body, followed by all the statements in the otherwise when present
- **do-while**: print all the statements in the body, followed the expression on the condition in a new line
- **repeat**: print the expression on the number of repetitions in a new line, followed by all the statements in the body
- **assignment**: print the expression in the right-hand side in a new line
- **function call**: same as the function call expression (see below) but in a new line

## Expressions

- **integer constants**: print the constant surrounded by parentheses
- **string constants**: print the constant surrounded by double-quotes, and that surrounded by parentheses
- **binary operations**: print the left-hand side expression (as defined in this document), followed by the symbol for the operation (same as in the language specification), followed by the right-hand side. Surround the whole operation with parentheses
- **unary operations**: print the symbol for the operation (same as in the language specification), followed by the argument. Surround everything in parentheses.
- **variables**: print the variable name surrounded by parentheses.
- **function calls**: print the function name, and the arguments as a comma-separated list surrounded by parentheses.
- **extra parentheses**: print the inner expression, drop the parentheses

# Unneeded parentheses

Your submission should only print parenthesis to disambiguate expressions. If the original program has too many parentheses, your output should not print them.

For instance, consider the following valid program in Cardinal:

```
return (((((1+ (2)) * 3)));
```

The expected output for this program is:

```
(((1)+(2))*(3))
```

# Return value and syntax errors

When parsing a file successfully, your submission should print the output described above and return with code zero. When there is a syntax error, the output is left unspecified but your submission should exit with a non-zero code.

# Document contents

Given that Cardinal is a templating language, your submission should print all the document contents around Cardinal code inside square brackets [ and ]. For instance, for the following Cardinal program

```
This is some original document<?cardinal?>return ((473));</?cardinal?>
```

Your submission should print:

```
[This is some original document]
(473)
```

# Entry Point

You will modify file `lexer.lex` with all the rules for the lexer from Assignment 1.

You will modify file `parser.y` to:

- generate all the tokens needed for the lexer
- add all the grammar rules to recognize Cardinal programs
- add all the actions to generate the strings described in this document
- add any auxiliary functions you need to generate those strings
- write the result consisting of all the strings to global variable **char * result**

**You will not modify any other file in the provided code.**

# Due Date and Resubmission Policy

This assignment is due on **September 27th 2025** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

~~To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** for undergraduate students~~. You can resubmit your assignment until **October 4th 2025** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

# Bonus Points

This assignment has a total of **20% bonus points**.

You can earn 10% extra points by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment2` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.

You can earn 10% extra points by fixing all memory leaks in your program. Each passing test without memory leaks is worth 1% bonus points.

# Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

# Graduate students

On top of the behavior described above, graduate students have to implement support for reading/writing arrays. Students should come up with their own syntax for array types, array read, and array write. Students do not need to support multi-dimensional arrays (only one dimension is OK).

Each regular test is only worth 8% for graduate students. The remaining 20% will be graded as described below.

This language extension will be graded via one video screen-cast (**through Gradescope**) that answers the questions below by explaining how your code works, together with positive and negative example programs:

1. Does your submission support expressions that read from arrays? (e.g., 10 + arr_read(arr, 10)):
2. Does your submission support expressions that write to arrays? (e.g., arr_write(10, 10);)
3. Does your submission support expressions that read/write to/from arrays and use many nested sub-expressions to compute the index? (e.g., arr[10*12+var] := arr[2-var%function(20)];)
4. Does your submission support expressions that read/write to/from arrays and use many nested array read sub-expressions to compute the index? (e.g., arr1[arr2[0]] := arr3[arr4[10+var] – arr5[arr0[0]]];)

You can record such a video using Zoom, which you may already have installed to attend lectures remotely. Simply start a meeting (without any other participants), share your screen, and start recording. Note that Zoom requires some time to process your video after you record it, so plan accordingly. Extension requests to upload videos after the due time and date because Zoom is still processing them will be denied.

**The maximum length for the video is 5 minutes, instructors will stop watching at the 5 minute mark (nothing past that point in the video will be graded).** This video should be a screencast of your IDE open on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

# Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

# Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.