November 29, 2017.

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Santa Fe
Escuela de Ingeniería y Ciencias, Región Ciudad de México
Departamento de Computación

The Legendary Compiler Design Final Exam

Instructor: Ariel Ortiz

Course Number and Section: Tc3048.1

Name: Jonathan Ginsburg Yagar

Student ID: A01021617

Apegándome al Código de Ética de los Estudiantes del Tecnológico de Monterrey, me comprometo a que mi actuación en este examen esté regida por la honestidad académica. En congruencia con el compromiso adquirido con dicho código, realizaré este examen de forma honesta y personal, para reflejar, a través de él, mi conocimiento y aceptar, posteriormente, la evaluación obtenida.

Firma: ______________________

VERY IMPORTANT: This exam should be solved individually. Any type of plagiarism or copying will be sanctioned with a grade of 1/100 and will also be reported to the institution's Academic Integrity Committee. This sanction is for both the person who copies and for the person who allows to be copied.

General Instructions

You must solve the following problem on your own using your mobile computer. The complete solution to the problem must be stored in one, and only one, source file called `trillian.cs`. Once you have finished the exam upload this file using the course website. Make sure the source file includes at the top the author's personal information (name and student ID) within comments. **10 points will be subtracted from your exam if you omit this last piece of information.**

Time limit: 180 minutes.

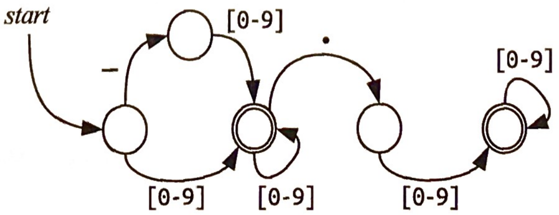
Problem Description

Trillian is a simple language that allows you to perform maximum, duplication, and summation operations over 64-bit floating point numbers. The language is named after Tricia McMillan, also known as Trillian, from Douglas Adams' book *The Hitchhiker's Guide to the Galaxy.* This is *Trillian*'s grammar using the Backus–Naur form notation:

$$
\begin{aligned}
\langle \textit{max} \rangle &\to \langle \textit{max} \rangle \text{ “!” } \langle \textit{simple} \rangle \\
\langle \textit{max} \rangle &\to \langle \textit{simple} \rangle \\
\langle \textit{simple} \rangle &\to \langle \textit{float} \rangle \\
\langle \textit{simple} \rangle &\to \text{“*” } \langle \textit{simple} \rangle \\
\langle \textit{simple} \rangle &\to \text{“[” } \langle \textit{max-list} \rangle \text{ “]”} \\
\langle \textit{max-list} \rangle &\to \langle \textit{max-list} \rangle \text{ “,” } \langle \textit{max} \rangle \\
\langle \textit{max-list} \rangle &\to \langle \textit{max} \rangle
\end{aligned}
$$

$\langle max \rangle \to \langle simple \rangle$ ("!" $\langle simple \rangle$)*
$\langle simple \rangle \to \langle float \rangle$ | ("*" $\langle simple \rangle$) | ("[" $\langle max\text{-}list \rangle$ "]")
$\langle max\text{-}list \rangle \to \langle max \rangle$ ("," $\langle max \rangle$)*
⇒ $\langle program \rangle \to \langle max \rangle$ "eof".

The start production is ⟨*max*⟩. The double-quoted elements represent terminal symbols. Spaces and tabs are allowed but should be ignored. ⟨*float*⟩ is a terminal symbol defined using the following deterministic finite automaton:



The operators supported by the language are described in the following table:

Operator	*Description* Supremum
exp_1 ! exp_2	Returns the **maximum** (largest value) of exp_1 and exp_2.
*exp	Returns exp **duplicated**, that is: $exp + exp$.
[exp_1, exp_2, ..., exp_n]	Returns the **summation** of all its elements, that is: $exp_1 + exp_2 + ... + exp_n$.

Examples:

Source Program	*Executable Program Output*
`-5 ! -7.8`	`-5`
`*-4.2`	`-8.4`
`[4, 10.3, -5.2, 0]`	`9.1`
`[1, **4.1 ! 4.5 ! *-4.2, -3] ! *[10]`	20

Using C#, write a recursive descent parser that allows compiling a *Trillian* program. For any given input, your program must scan and parse it, build an abstract syntax tree (AST), and generate CIL assembly language.

A few things that you must consider:

- The input program is always received as a command line argument.
- If a syntax error is detected, a "parse error" message should be displayed, and the program should end.
- If no syntax errors are detected, the AST should be displayed.
- The assembly language output must always be stored in a file called "`output.il`".
- The purpose of the generated code is to evaluate at runtime the input program and print the result to the standard output (emitting a call to the `System.Console.WriteLine` method from the CLI class library).
- The `ilasm` command will be called manually from the OS terminal.

A Complete Example

When given this command at the terminal:

```
    mono trillian.exe '[1, **4.1 ! 4.5 ! *-4.2, -3] ! *[10]'
```

the following AST should be displayed at the standard output:

```
Program
  Max
    Sum
      1
      Max
        Max
          Dup
            Dup
              4.1
          4.5
        Dup
          -4.2
      -3
    Dup
      Sum
        10
```

TIP: Override the `ToString()` method in the class that represents your floating point literal nodes so that it displays the corresponding lexeme instead of the name of the class.

The contents of the generated `output.il` file should be:

```
.assembly ’trillian’ {}

.class public ’final_exam’ extends [’mscorlib’]’System’.’Object’ {
    .method public static void ’start’() {
        .entrypoint
        ldc.r8 1
        ldc.r8 4.1
        dup
        add
        dup
        add
        ldc.r8 4.5
        call float64 [’mscorlib’]’System’.’Math’::’Max’(float64, float64)
        ldc.r8 -4.2
        dup
        add
        call float64 [’mscorlib’]’System’.’Math’::’Max’(float64, float64)
        add
        ldc.r8 -3
        add
        ldc.r8 10
        dup
        add
        call float64 [’mscorlib’]’System’.’Math’::’Max’(float64, float64)
        call void [’mscorlib’]’System’.’Console’::’WriteLine’(float64)
        ret
    }
}
```

Once assembled using `ilasm`, the above program outputs the following when executed:

```
20
```

Code Generation Overview

- The `!` operator involves generating the code for the left and right operands, and then emitting a call to the `System.Math.Max` method from the CLI class library (see previous example).

- The `*` operator involves generating code for its operand, and then emitting a `dup` instruction followed by an `add` instruction.

- The summation operation $[x_1, x_2, x_3, ..., x_n]$ involves generating the code for the first element x_1, then for each element x_i from $x_2, x_3, ..., x_n$ you must generate the code for x_i followed by an `add` instruction.

Grading

The grade depends on the following:

1. **(10)** The C# code compiles without errors.
2. **(30)** Fulfills point 1, plus: performs lexical and syntax analysis without any errors.
3. **(50)** Fulfills points 1 and 2, plus: builds and prints the AST without any errors.
4. **(100)** Fulfills points 1, 2, and 3, plus: generates CIL assembly language without any errors.

> *There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.*
>
> — Douglas Adams