December 1, 2021.

Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
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

By solving and handing in this exam, you agree to the following:

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, para reflejar, a través de él, mi conocimiento y aceptar, posteriormente, la evaluación obtenida.

General Instructions

The complete solution to the problem must be stored in one, and only one, source file called `meraxes.cs`. Once you have finished the exam upload this file using the course website. Make sure the source file includes at the top all the authors' personal information (name and student ID) within comments.

Time limit: 120 minutes.

Problem Description

Meraxes[1] is a simple language that allows you to obtain the duplication, addition and multiplication of sequences of 64-bit integer point numbers. This is its grammar using the BNF (Backus–Naur form) notation:

$$
\begin{aligned}
&\langle expr\rangle \rightarrow \langle dup\rangle\\
&\langle expr\rangle \rightarrow \langle sum\rangle\\
&\langle expr\rangle \rightarrow \langle mul\rangle\\
&\langle expr\rangle \rightarrow \langle int\rangle\\
&\langle dup\rangle \rightarrow \text{“*”}\ \langle expr\rangle\\
&\langle sum\rangle \rightarrow \text{“[”}\ \langle list\rangle\ \text{“]”}\\
&\langle mul\rangle \rightarrow \text{“\{”}\ \langle list\rangle\ \text{“\}”}\\
&\langle list\rangle \rightarrow \langle list\rangle\ \text{“,”}\ \langle expr\rangle\\
&\langle list\rangle \rightarrow \langle expr\rangle
\end{aligned}
$$

The start production is $\langle expr\rangle$. The double-quoted elements represent terminal symbols. Spaces and tabs are allowed but should be ignored. The terminal symbol $\langle int\rangle$ represents a decimal positive integer comprised of one or more digits from 0 to 9. The operators supported by the language are described in the following table:

Operator	*Description*
`*`exp	Returns the **duplication** of *exp*, that is, twice the value of *exp*.
`[`exp_1`,` exp_2`,` … `,` exp_n`]`	Returns the **addition** of all its elements, that is, $exp_1 + exp_2 + \cdots + exp_n$.
`{`exp_1`,` exp_2`,` … `,` exp_n`}`	Returns the **multiplications** of all its elements, that is, $exp_1 \times exp_2 \times \cdots \times exp_n$.

[1] *Meraxes* was a she-dragon of House Targaryen from the *Game of Thrones* universe.

Examples:

Source Program	*Executable Program Output*
`5`	`5`
`* 5`	`10`
`[1, 2, 3, 4, 5]`	`15`
`{1, 2, 3, 4, 5}`	`120`
`[*1, *0, **[*2, {*1, 3, 1}, 0]]`	`42`

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

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 WebAssembly text code output must always be stored in a file called "`output.wat`".
- The purpose of the generated code is to evaluate at runtime the input expression and return its result.
- The `execute.py` command will be called manually from the terminal.

A Complete Example

When given this command at the terminal:

```
mono meraxes.exe '[*1, *0, **[*2, {*1, 3, 1}, 0]]'
```

the following AST should be displayed at the standard output:

```
Program
  Sum
    Dup
      1
    Dup
      0
    Dup
      Dup
        Sum
          Dup
            2
          Mul
            Dup
              1
            3
            1
          0
```

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

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

```
(module
  (func
    (export "start")
    (result i64)
    i64.const 1
    i64.const 2
    i64.mul
    i64.const 0
    i64.const 2
    i64.mul
    i64.add
    i64.const 2
    i64.const 2
    i64.mul
    i64.const 1
    i64.const 2
    i64.mul
    i64.const 3
    i64.mul
    i64.const 1
    i64.mul
    i64.add
    i64.const 0
    i64.add
    i64.const 2
    i64.mul
    i64.const 2
    i64.mul
    i64.add
  )
)
```

The provided `execute.py` script takes care of compiling and running the WebAssembly code. At the terminal, this command:

```
python execute.py
```

should produce the following output:

```
42
```

Tips for code generation

- The `*` operator involves generating code for its operand, and then emitting an "`i64.const 2`" instruction followed by an "`i64.mul`" instruction.
- The addition 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 "`i64.add`" instruction.
- Likewise, the multiplication 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 "`i64.mul`" 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 WebAssembly text code without any errors.