December 2, 2020.

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 `kaida.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

Kaida (Japanese for *little dragon*) is a simple language that allows you to perform maximum, duplication, and summation operations over signed 32-bit integer numbers. This is *Kaida*'s grammar using the BNF (Backus–Naur form) notation:

⟨*max*⟩ → ⟨*max*⟩ "$" ⟨*simple*⟩
⟨*max*⟩ → ⟨*simple*⟩
⟨*simple*⟩ → ⟨*int*⟩
⟨*simple*⟩ → "@" ⟨*simple*⟩
⟨*simple*⟩ → "(" ⟨*max-list*⟩ ")"
⟨*max-list*⟩ → ⟨*max-list*⟩ "|" ⟨*max*⟩
⟨*max-list*⟩ → ⟨*max*⟩

The start production is ⟨*max*⟩. The double-quoted elements represent terminal symbols. Spaces and tabs are allowed but should be ignored. ⟨*int*⟩ is a terminal symbol that represents a signed 32 bit integer literal, which is comprised of an optional minus sign followed by a sequence of one or more base 10 digits.

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

Operator	*Description*
exp_1 `$` exp_2	Returns the **maximum** (largest value) of exp_1 and exp_2.
`@`*exp*	Returns *exp* **duplicated**, that is: *exp* * 2.
`(`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`	`-5`
`@-4`	`-8`
`(4 | 10 | -5 | 0)`	`9`
`(1 | @@4 $ 4 $ @-4 | -3) $ @(10)`	`20`

Using C#, write a recursive descent parser that allows compiling a *Kaida* 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.js` command will be called manually from the terminal.

A Complete Example

When given this command at the terminal:

```
mono kaida.exe '(1 | @@4 $ 4 $ @-4 | -3) $ @(10)'
```

the following AST should be displayed at the standard output:

```
Program
  Max
    Sum
      1
      Max
        Max
          Dup
            Dup
              4
          4
        Dup
          -4
      -3
    Dup
      Sum
        10
```

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
  (import "kaida" "max" (func $max (param i32) (param i32) (result i32)))
  (func
    (export "start")
    (result i32)
    i32.const 1
    i32.const 4
    i32.const 2
    i32.mul
    i32.const 2
    i32.mul
    i32.const 4
    call $max
    i32.const -4
    i32.const 2
    i32.mul
    call $max
    i32.add
    i32.const -3
    i32.add
    i32.const 10
    i32.const 2
    i32.mul
    call $max
  )
)
```

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

```
node execute.js
```

should produce the following output:

```
20
```

Tips for code generation

- The `$` operator involves generating the code for the left and right operands, and then emitting a "`call $max`" instruction.
- The `@` operator involves generating code for its operand, and then emitting an "`i32.const 2`" instruction followed by an "`i32.mul`" 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 "`i32.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 WebAssembly text code without any errors.