1 Lab 1 Syntax

Date: Feb 25, 2021

This document first describes the aims of this lab followed by exercises which need to be performed.

You can perform exercises using any suitable programming language. However, some of the exercises provide code in JavaScript and Python to help you get started.

[Note that the Python code may not be particularly pythonic as it is not a language I am particularly familiar with. I tend to avoid it as I find I am always fighting the language when attempting to use it.]

1.1 Aims

The aim of this lab is to introduce you to techniques used for analyzing the syntactic structure of programming languages. After completing this lab, you should have some familiarity with the following topics:

- Using regular expressions to specify the lexical syntax of programming languages.
- Enhancing a CFG to support additional constructs.
- Using recursive-decent parsing to evaluate arithmetic expressions and construct abstract syntax trees.

1.2 Exercises

1.2.1 Starting up

Follow the *provided directions* for starting up this lab in a new git `lab1` branch and a new `submit/lab1` directory. You should have copied over the `~/cs471¬ /labs/lab1/exercises` over to your `submit/lab1` directory.

1.2.2 Exercise 1: Regular Expression

Change over to your `~/i471?/submit/lab1/exercises/regex` *Exercise 1* directory.

Among the contained files you should find a copy of lex1.mjs. This file defines a JavaScript table which maps token kind's to the regex's which define that kind. You will be concentrating on this file. The way this file is used is that the code

in table-lexer.mjs converts the table into a form which can be used by a crude scanner with lexer.mjs providing a driver program.

Run the provided `lex1.mjs`:

```
$ ./lexer.mjs
usage: lexer.mjs RE_TABLE_FILE INPUT_FILE
$ ./lexer.mjs lex1.mjs data.txt
Token { kind: 'INT', lexeme: '21' }
Token { kind: 'INT', lexeme: '23' }
Token { kind: 'CHAR', lexeme: 'A' }
Token { kind: 'INT', lexeme: '34' }
Token { kind: 'CHAR', lexeme: '_' }
Token { kind: 'INT', lexeme: '22' }
...
Token { kind: 'CHAR', lexeme: 'f' }
Token { kind: 'CHAR', lexeme: 's' }
$
```

The `lex1.mjs` table is set up to recognize integers consisting of one-or-more digits, ignore whitespace (using the special `$Ignore kind`) and catch all unknown characters as single character tokens having the `CHAR kind`.

Copy `lex1.mjs` into `lex2.mjs`. Modify the table in `lex2.mjs` so as to recognize identifiers consisting of one-or-more alphanumeric characters or underscores `_` with the restriction that the first character must be alphabetical or `_`. Note that the regex literals in the table are JavaScript regex literals and that JavaScript does not allow whitespace within regex literals.

```
$ ./lexer.mjs lex2.mjs data.txt
Token { kind: 'INT', lexeme: '21' }
Token { kind: 'INT', lexeme: '23' }
Token { kind: 'ID', lexeme: 'A34' }
Token { kind: 'ID', lexeme: '_22' }
Token { kind: 'ID', lexeme: 'ID_44' }
Token { kind: 'CHAR', lexeme: '/' }
Token { kind: 'CHAR', lexeme: '/' }
Token { kind: 'ID', lexeme: 'asd' }
Token { kind: 'ID', lexeme: 'sdfs' }
$
```

Copy `lex2.mjs` into `lex3.mjs`. Modify the table in `lex3.mjs` so as to ignore `//` C-style comments: i.e. comments starting with `//` and extending to end-of-line.

- You can only have a single `$Ignore` entry in the table; hence you will

need to modify the existing `$Ignore` entry to ignore both whitespace and //-comments.

- / is a special regex character in that it terminates a regex. Hence if you want to match a /, it will need to be escaped using a \.
- . is a regex matching any character other than newline.

```
$ ./lexer.mjs lex3.mjs data.txt
Token { kind: 'INT', lexeme: '21' }
Token { kind: 'INT', lexeme: '23' }
Token { kind: 'ID', lexeme: 'A34' }
Token { kind: 'ID', lexeme: '_22' }
Token { kind: 'ID', lexeme: 'ID_44' }
$
```

1.2.3 Exercise 2: Building a Lexer

Change over to your `~/i471?/submit/lab1/exercises/lexer` *Exercise 2* directory.

This directory contains a more procedural version of a lexer similar to the table-driven lexer from the previous exercise. Specifically, it contains both a *JavaScript lexer* and a *Python lexer*. Both lexers are set up to run very similarly to the lexer from the previous exercise except that instead of defaulting single-char tokens to have a `kind` of `CHAR`, the `kind` for those tokens is set to the string corresponding to the single character.

Run either of the programs. The results are similar to your initial run of `lex¬1.mjs` in the previous exercise except for the aforementioned difference with tokens corresponding to single character lexemes:

```
$ ./lexer.py
usage: ./lexer.py DATA_FILE
$ ./lexer.py data.txt
Token(kind='INT', lexeme='21')
Token(kind='INT', lexeme='23')
Token(kind='A', lexeme='A')
Token(kind='INT', lexeme='34')
Token(kind='_', lexeme='_')
...
Token(kind='s', lexeme='s')
Token(kind='d', lexeme='d')
Token(kind='f', lexeme='f')
Token(kind='s', lexeme='s')
```

```
$
```

Your task in this exercise is to choose one of the Python or JavaScript lexers and modify it as in the previous exercise: i.e. recognize identifiers and ignore //-comments. Alternately, you may use the provided code as pseudo-code and roll your own in your favorite programming language.

1.2.4 Exercise 3: A Calculator

Change over to your `~/i471?/submit/lab1/exercises/calc` *Exercise 3* directory.

This directory contains `calc.{mjs,py}` JavaScript/Python calculators for `;`-separated expressions over integers containing left-associative `+` and `-` binary operators, unary `-` and parentheses used for grouping.

```
program
  : expr ';' program
  | #empty
  ;
expr
  : term ( ( '+' | '-' ) term )*
  ;
term
  : '-' term
  | factor
  ;
factor
  : INT
  | '(' expr ')'
  ;
```

The provided code works as a calculator:

```
$ cat data.txt
1 + 2 - 3;

--2 - (3 - 4);
$ ./calc.mjs
usage: calc.mjs INPUT_FILE
$ ./calc.mjs data.txt
[ 0, 3 ]
```

Your task in this exercise is to modify the grammar and your chosen implementation to support a right-associative `**` exponentiation operator. The new

operator should have a higher precedence than unary - as shown in the following log:

```
$ cat expn-data.txt
-2 ** 2 ** 3 ;
1 + 2**(1 + 2) ;
1 + -2**(1 + 2) ;
$ ./calc.py expn-data.txt
[-256, 9, -7]
$
```

- Modify the grammar to support the `**` operator. At first glance it appears necessary to add another non-terminal to support the additional precedence level, but since unary-- is prefix and exponentiation is infix, this can be done within the existing `term` non-terminal.
- Modify the scanner to support the `**` operator.
- Make changes in the parser as per your grammar. This should require very few additional lines of code.

1.2.5 Exercise 4: Building ASTs

Change over to your `~/i471?/submit/lab1/exercises/ast` *Exercise 4* directory.

This directory contains `ast.{mjs,py}` implementations for building AST's for the grammar from the previous exercise. After building the AST's the implementations dump the AST's as JSON on standard output without any whitespace, except for a terminating newline.

As in the previous exercise, modify the implementation to add the exponentiation operator and build the corresponding AST's.

The JSON output for *input*.txt is available in *input*.json. You can pretty-print JSON by piping it into `json_pp`.

- Every AST node contains a `tag` field which is set to the operator for operator AST's.
- The `tag` field for an AST corresponding to an integer is set to `INT`. The value of the integer is added to the AST using an extra `value` field; note that the type of `value` must be an integer.
- The `kids` field of an AST node is a list containing the operands for that AST node.