Projects for CS3050/CS5017 Theory of Computation

Aug-Nov 2023

Reading Project

The area of Theory of Computation is rich in terms of ideas. One of the goals of reading project is to get a taste of it by reading up selected topics. The key aim is to broaden your understanding and also develop a better insight as to how they are connected to each other and to the topics covered in class.

Each project can be done in groups of size at most 2. The presentations are scheduled on November 25 (Saturday).

Topics

The list of topics are given below (more topics will be added if needed):

- *Arden's Theorem and Kleene Algebra* -
  - **Curiosity questions**: Why do we need the rules 9.12 and 9.13 as part of Kleena algebra axioms ? Why care about Kleene Algebra ?
  - **Partial answer**: Finding $\leq$ least solution using Arden's Theorem. The rules 9.12 and 9.13 precisely capture this !
  - **Take away**: A different way to obtain regular expressions directly from DFA using Kleene Algebra
  - **Reading material**: For a statement, proof and examples, see here. For connection to Kleene algebra - Kozen, Supplementary Lecture A.
  - **Expectation**: Understand the statement and proof of Arden's lemma. Understand what is reflexive transitive closure and its relation to Arden's Theorem (Kozen, Miscellaneous exercises 1 and 23). Present the connection to Kleene Algebra (Kozen, Supplementary Lecture A).
- *Homomorphisms* -
  - **Curiosity questions**: The language $L_1 = \{0^n 1^n \mid n \geq 0\}$ is not regular and can be argued via pumping lemma. The following language $L_2 = \{0^n \# 1^n \mid n \geq 0\}$ which "looks" similar to L_1 is also not regular. Is it a coincidence or can we formalize this ?
  - **Answer**: (Inverse) Homomorphisms !
  - **Take away**: Using homomorphisms to prove regularity/non-regularity of languages.
  - **Reading materials**: Kozen, Lecture 10.
  - **Expectation**: What are homomorphisms and how can they be used to argue regularity/non-regularity of languages ?
- *Deterministic Push-down automatas*:
  - **Curiosity questions**: What if we restrict a Push-down automata to be deterministic ? What all languages that it can accept ?
  - **Answer**: This gives a deterministic PDA and Deterministic Context-Free Languages.
  - **Take away**: Deterministic PDAs a more nicer and closed under complementation unlike the usual push-down automatas.
  - **Reading materials**: Kozen, Supplementary Lecture F.

  - **Expectation**: Being able to show that Deterministic PDAs are closed under complementation and explain the technical issues arising in this construction.
- *The Chomsky-Schutzenberger Theorem*
  - **Curiosity questions**: For regular languages, there is a well defined notion of unique minimal DFA via Myhill-Nerode relation. Is there a unique context free grammar for every context free language ?
  - **Partial answer**: Not quite ! But all context free grammars "looks' ' the same as a particular kind of context free grammar (called Dyck grammars).
  - **Take away**: The structure of any context free languages are essentially captured by Dyck languages.
  - **Reading material**: Kozen, Lecture 20, Supplementary lecture G. Also see here
  - **Expectation**: Understand the structural characterisation and fully demonstrate it using an example.
- *String pattern matching algorithm* : Knuth-Morris-Pratt algorithm.
  - **Curiosity questions**: Are there any algorithms that are used in practise which is inspired by automata design ?
  - **An answer**: The Knuth-Morris-Pratt algorithm
  - **Take away**: How can finite automata be helpful in faster pattern search
  - **Reading material**: Original paper describing the algorithm. Also Jeffe Erickson's notes section 7.4, 7.5, 7.6, 7.7 and exercises 4, 5, 7, 8 both of lecture 7.
  - **Expectation**: Completely understand the KMP algorithm and how an automata based thought process helps in string search.
- *Ogden's Lemma and application to ambiguity of grammars* :
  - **Curiosity questions**: To show that the grammar for a language is ambiguous, it suffices to show two parse trees for the same string. Can there be languages where *all* grammars are ambiguous (aka the CFL is inherently ambiguous) ?
  - **Partial answer**: Yes. This can be checked using Ogden's lemma, a stronger form of Pumping lemma for CFLs.
  - **Take away**: We can check if the language is itself ambiguous using Ogden's Lemma in addition to non-context freeness.
  - **Reading material**: Section 4.3 (complete) and Section 4.4, Theorem 4.4.1 of this pdf
  - **Expectation**: Understand the statement of lemma, proof and its application to ambiguity.
- *Parsing techniques - LL(1) parsers*:
  - **Curiosity questions**: How is the theory of CFL helpful in designing compilers for programming languages ?
  - **Partial answer**: Key notion used is parse trees. One of the popular method, called LL(k), builds parse trees in a top-down fashion using left to right scan and using k-look aheads.
  - **Take away**: Use of LL(k) parsers in practise.
  - **Reading material**: Chapter 7 of this pdf. You can also look at this online tool for understanding grammars in various representations.
  - **Expectation**: Understand how the FIRST and FOLLOW sets are computed, proof of correctness and an example.
- *Buchi Automata - Automata theory and model checking*:
  - **Curiosity questions**: Automata theory deals with strings of finite length. What about strings of infinite length ? Such strings can be used to model behaviour of systems that runs infinitely like a satellite transponder, a traffic junction signal system or even the run of an operating system process scheduler.
  - **Partial answer**: Buchi automata
  - **Take away**: Natural ideas from automata theory that worked for finite length strings also (mostly) works in the infinite length setting too. Such an extension is also very helpful in practical settings like verification.

  - **Reading material**: Section 2 and 3 of this article by Madhavan Mukund.
  - **Expectation**: Being able to explain what is Buchi automata, closure properties, connection to LTL and how it is used in mode checking.
- *Regular Languages over Unary*:
  - **Curiosity questions**: We know that unary languages like PRIMES are not regular. So primes as a sequence cannot be recognized by a DFA. What kind of sequences will make a unary language regular ?
  - **Partial answer**: Ultimately periodic sequences
  - **Take away**: A unary language is regular if and only if the lengths of strings in the language are ultimately periodic.
  - **Reading material**: Kozen Lecture 12
  - **Expectation**: Being able to explain the **Take away** and gives a structural characterization of automata accepting unary languages. Also solve the Miscellaneous exercise 45.

Rules

The rules of this game are: pick a topic from the above and understand it well. There will be an interim meeting (this carries 40% weight) with the instructor on November 18 (a week before the presentation day) for a review and for clarifications. You need to give a black board talk to the instructor/TAs on November 25.

It is strongly recommended to use the blackboard. Nevertheless, if you plan to use slides, please have a word with the instructor and get permission.

Please write your name against the project ~~here~~ ~~if you are interested in doing this.~~ Thank you ! This is closed now.

Programming Project

The aim of this project is to build a reverse polish notation calculator for polynomials over one variable in reals that can do symbolic differentiation !

Reverse polish notation (RPN) or the postfix notation is a mathematical notation where the operator appears after the operands. For example, "5 + 3" in postfix is "5 3 +". This allows a mathematical expression to be written without the need of using parentheses.

You are given an expression in the RPN form and your task is to evaluate the RPN expression and output the result as an RPN expression.

Specifications

The implementation has been specified in stages. You are very strongly recommended to follow these stages. It is recommended (though not mandatory) that you create test cases for each of the stages. This way, you can test each stage before proceeding to the next one.

- **Stage 1**: First, build a calculator that can do integer arithmetic involving operations `^, /, *, +, -` corresponding to the usual operations of exponentiation, division, multiplication, addition and subtraction.

  The operator follow the precedence order mentioned. An example is: `2 5 ^ 10 +` should output `42` !

- **Stage 2**: Extend stage 1 to include operations over real numbers.

  Allow precision of 5 digits after the decimal point.

- **Stage 3**. Extend stage 2 to include operations involving a symbolic variable `x`.

  For instance, arithmetic involving variables must return the correct expression. For example `1 x + x 1 + +` should return `2 x 2 * +`. ~~The expression `2 2 x * +` is not valid as `x` does appear at the end.~~ Please refer to the grammar below to decide whether an expression in RPN form is valid or not. *While testing this stage, you may assume that `/` does not appear in the input and exponent of any variable `x` is non-negative.*

- **Stage 4**: Extend stage 3 to include a new operator `d` that differentiates its argument with respect to `x`.

  For example `x 2 ^ d` should output `x 2 *`. The `d` operator has the highest precedence among the rest of operators. *While testing this stage, you may assume that `/` does not appear in the input and exponent of any variable `x` is non-negative.*

- **Stage 5**: Reimplement stage 4 (as well as stage 3) to (1) allow division `/` operation in the expression and (2) allow variable x to have negative exponent.

  That is, you may get expressions of the form `x -2 ^ 1 + x /`. *While testing this stage, you may assume that `d` (differentiation) does not appear in the input.*

- **Stage 6**: Reimplement stage 5 to allow differentiation on rationals of polynomials of `x`.

  That is, `x -2 ^1 + x / d` should output `-3 x -4 ^ * -1 x -2 ^ * +`. Note: the denominator part can be an arbitrary polynomial in `x` and need not always be `x`.

Validity of expressions

A well formed RPN expression is the one generated by the following grammar rules (with `E` as the starting non-terminal).

```
E -> E E + | E E - | E E * | E E / | E d | E n ^
E -> x | n | n.PPPPP
P -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Here `n` is a signed integer of 32 bits.

Your program must output "Parse error" if the input expression is not a string generated by the above grammar.

While outputing terms involving powers of variable `x`, the terms should appear in the increasing order of degree. For example, if your final expression is $(x^2 + x + 1)/(x + 1)$, it should be printed (in RPN as) `1 x + x 2 ^ + 1 x + /`

A successful run of the program should terminate with a return 0. If the result involves division by zero, then program should output the string "NAN" (Not a number). On any error, the program should gracefully exit and return 1 and should **not** crash (messages like *stack overflow*, *memory overflow* or *segmentation fault* while running the program is not acceptable).

Choice of Programming language

- You are allowed to use *only* low level languages like C, C++ or Rust.
- You are only allowed to use the standard libraries that comes with these languages. Since your code will be tested against large test cases, language like Python will be inadequate.
- Your code will be benchmarked against a set of test cases.
- ~~Specification of the test will be given in due course.~~

Here are the contents of a sample test file `test.txt`. Each of the test expressions will be provided in a new line. Your program should process the input line-by-line till it sees "EOF" appearing in a newline.

```
1 x x 2 ^ + 1 x +
x 2 ^ 1.00001 + d
3 2 3 - 1 + /
EOF
```

Output should be

```
Parse error
x 2 *
NAN
```

Deliverables

All the stages (including stage 6) must be implemented fully and must pass all your test cases to be considered for evaluation. **Submission is open for all. There is no separate sign-up required**.

Submission link for CS3050: Click here to open.

Submission link for CS5017: Click here to open.

Final submission made in course moodle page before December 11 (Monday, 11.59 PM) alone will be considered. No further extension on this deadline is possible.

You are expected to write a `Makefile` to build your program and build it using `make` tool. Your final program should be named as `rpn` and it should be available in the `build` directory created by the `make` program. Your program will be tested this way.

```
$ make
$ cd build
$ ./rpn < test.txt > out.txt
```

Your submission to moodle should be a single zip file with all the relevant source files, `Makefile` and a `README` file explaining the purpose of each file. **Submissions deviating from these guidelines may be discarded**. Any instance of academic dishonesty will have severe consequences.