CS 37

Algorithms in the Real World

Welcome to CS 37!

Outline

What Am I Getting Into? 1

Course Material

- Lossless Compression
- Semi-Numerical Algorithms
- Randomness Generation
- Prime Generation
- Private Key Cryptography
- Error-Correcting Codes
- Parallel Algorithms

Projects

- P0: Puzzle Hunt
- P1: `zip`
- P2: Wiki
- P3: `ssh` & `wget`
- P4: Barcodes

What Am I Getting Into? 2

Course Goals

- Learn some new algorithms
- Implement algorithms in a real world context
- Prepare you for the real world
- Write programs with and without existing libraries
- Learn how to collaborate effectively when programming

"Real World" Focus

- Implementation
- Design Decisions
- Real Applications
- Collaboration

Support and Asking for Help 3

Resources

- Office Hours!
- Piazza!

Asking for help is not a sign of weakness; it's a sign of strength.

Boring Administrivia

Course Website

`http://courses.cms.caltech.edu/cs37/18fa`

Grading

- 100% programming projects
- Ask if you need to submit late

Partner Projects

- All programming projects are "partners projects"
- If you want to work alone, you must petition to do it.

Textbook

Textbooks are (generically) a waste of money.

My Philosophy

Do what helps you most.

. . . but **active learning** has been proven to result in better comprehension.

"Real World"

It's really hard to describe what we mean by "real world"; so, we'll model the pieces of the course with condensed version of a topic.

The process will look something like this:

- Lecture introduces the topic, the basic algorithm(s), and applications
- Project implements the details and specifics

The Topic: Parsing

Evaluate a Mathematical Expression

Input: A string representing a mathematical expression (e.g., $1+3*4$)
Output: A single number representing the value of the input expression

Evaluation of a mathematical expression actually involves three steps:

1. Consume the string input, and produce a **list** of "tokens"
2. Consume the token list, and produce a **tree** representation
3. Consume the tree, and produce a **numerical** representation

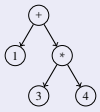
We will focus on step 2 which is called **parsing**. This has broader applications than just mathematical expressions (such as programming languages).

The Specifics: Parsing

The Output Tree ("Abstract Syntax Tree")

We can represent any mathematical expression as a tree where the root is the operation and the children are the operands.

For example, given $1+3*4$, we would have:

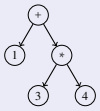


Note that it would be incorrect to use the $*$ node as the root because of the order of operations.

The Specifics: Parsing

The Output Tree ("Abstract Syntax Tree")

For example, given $1+3*4$, we would have:



We will represent these trees as nested dictionaries. Translating the above example, we'd get:

```
{
    type: "binaryop",
    value: "+",
    left: { type: "number", value: 1 },
    right: {
        type: "binaryop",
        value: "*",
        left: { type: "number", value: 3 },
        right: { type: "number", value: 4 },
    }
}
```

Parsing

We will write an algorithm `parse(-)` which takes a list of tokens and outputs an AST.

1	+	2	*	3	*	4

1	*	2	*	3	+	4

1	+	2	*	3	*	4	+	5	*	6

Parenthesization

Start and Continue Contexts


```
parse():
    result = eat()
    op = eat()
    if op:
        result = "(" + left + op + parse() + ")"
    return result
```

```
start():
    return eat()

cont(left):
    op = eat()
    right = parse()
    return "(" + left + op + right + ")"

parse():
    result = start()
    if peek():
        result = cont(left)
    return result
```

Contexts


```
start():
    return eat()

cont(left):
    op = eat()
    right = parse()
    return "(" + left + op + right + ")"

parse(context=0):
    result = left()
    while peek() and should_continue():
        result = right(left)
    return result
```

Contexts


```
start():
    return eat()

cont(left):
    op = eat()
    right = parse()
    return "(" + left + op + right + ")"

binding_power(token):
    BPS = {'+': 10, '*': 20}
    if token in BPS:
        return BPS[token]
    else:
        return 0

parse(context=0):
    result = left()
    while peek() and binding_power(peek()) > binding_power(context):
        result = right(left)
    return result
```

Why Bother?

Pratt parsing is used in various applications in the real world, but we'll focus on one:

JSLint

JSLint is a Javascript program that finds style errors in Javascript source code. It does this by parsing Javascript using a Pratt Parser and consuming the resulting AST.

If this were a real topic, the project would be to implement JSLint

What Next?

That's the general gist of what lecture will be like with a few exceptions:

- Usually, we'll cover multiple algorithms rather than just a single one. (There are lots of ways to parse strings. . .)
- Often, we'll discuss intuition on why the algorithms are actually correct (Very rarely, we'll do a proof.)
- It will often take multiple lectures to discuss nuances of implementing the algorithm for real (How do we handle right-associative operators?)

This course is about learning and implementing the algorithms behind every-day tools.