Excursions in Compiler-Construction

Rafael L. Cepeda

October 30, 2017

Contents

1 Preface

Compiler-construction has always fascinated, daunted, and inspired me. How does a compiler receive a sequence of bytes, represented in some arbitrary language, and turn it into something the machine understands? Inadvertently, complex text-processing issues arose, and thankfully my ignorance went unnoticed: “It takes too much time.”, “Do we really need to write that?”, “I can’t write now.” I wondered how compilers ticked and set out to learn this black art. With the process demystified, I learned that compiler-construction mainly involved finite-state machines, infinite-state machines, efficient data-structures and algorithms, CPU architecture, and the understanding of the theories governing computation.

2 Lexical Scanners

Our journey begins with the understanding of characters, their arrangement, and their syntactic category or part of speech. Is a word a language keyword, number, or operator? This is the job of the lexical scanner; to group characters into words and assign a syntactic category, or part of speech, to them. How do we model this arrangement in an abstract form? Finite-state machines.

3 Finite-state Machines

A finite-state machine represents a theoretical machine, though possibly physical, that is in exactly one state at any time. A vending machine takes fifty cents, two quarters or one half-dollar, before dispensing a drink. I insert the first; now the machine awaits the next. Inserting the second, the machine dispenses my drink, and rightly so. If the drink requested is empty, the machine enters an error state, allowing me to reclaim coin by pressing a strange button. This is a classic example of a finite-state machine; the vending machine is always in one state at any time.

Imagine a state machine that accepts the inputs `no` and `note`. When the machine receives `n` as input, it awaits `o` before advancing to the next state. How does the machine know whether to finish accepting `no`, or to await more characters to accept `note`? To resolve this non-determinism, the finite-state machine must look ahead to see if there is a following `t`, indicating a possible `note` token, or end of input.

3.1 Properties of Finite-state Machines

Imagine writing a finite State machine to accept an integer. the integer could be potentially of infinite length, so modeling this would be near impossible without a cyclic notation. To represent this behavior, the state can return to itself upon a valid digit input and can be accepted at any time; this is called *kleene closure* or simply closure. Using our vending machine example, each quarter insertion transitioned us to a different state. Oddly enough, each state transition is a finite-state machine in its own right. Accepting two quarters, is the joining of two finite-state machines that accept one quarter each; this property unsurprisingly is called *concatenation.* Lastly, finite-state machines have an *alternation* property, which allows them to change their transition flow; our vending machine may accept two quarters, or one half-dollar. This or control flow represents the alternation property. Using basic program control flow constructs, one can successfully implement a finite-state machine: `if-then-else` statements and `while` loops (to cover the cyclic behavior of the closure property). With this notion, I realized that all programs are, in fact, in one state at any time; all programs can be represented as a finite-state machine.

4 Parsers

The parser's job is to model the grammatical structure derived from a group of words. Unfortunately, our lexical scanner models a machine with a finite number of states. Could you write a finite-state machine that accepted recursive `if-then-else` expressions? Unfortunately, no, because that represents infinitely many states.

4.1 Context-free Grammar

Similar to finite-state machines, context-free grammar, allows us to abstractly convey an infinite-state machine; allowing one to validate sentences grammatically. A potentially recursive expression, optionally enclosed in parentheses, can be represented as `Expr -> Expr number | (Expr) | number | e-string`. This CFG accepts inputs like `(((3)))`, `((3 3 3))`, and simply `3`. `e-string` is a special rule meaning *no input.* To derive a grammar, substitute `Expr` with one of the rules on the right side of the arrow until all non-terminals have been replaced by terminals; rules are separated by the pipe symbol. The term `Expr` is a *non-terminal* symbol, a symbol that must be substituted recursively until it reaches a *terminal* symbol like `number`. In the CFG, `number` represents a syntactic category, or part of speech, returned from the lexical scanner, rather than a specific number. Thus, the lexical scanner's job is to feed the parser words, and the parser's job is to validate those words, and construct a data-structure, typically a parse tree, that represents the grammatical structure of the input.

4.1.1 The Gotchas of CFGs

There are some pitfalls of CFG construction, however, such as ambiguous grammar and left-recursion. Both must be eliminated for proper implementation. From the example above, which rule should I use to substitute `Expr`? In other words, should I substitute `Expr` with `number`, `Expr`, `(Expr)`, or `e-string`? If the program tries the rules in order, the substitutions would continue indefinitely because each `Expr` yields an `Expr` This is called *left-recursive grammar*, and needs to be rewritten to try terminal substitutions first. “What if I choose the wrong terminal rule?” you ask. Though costly, you can resolve this ambiguity by backtracking and trying the next rule. Though, in practice, a one word look-ahead is enough to resolve most ambiguities.

4.2 Types of Parsers

Several parser implementations exist, namely, table-driven, LL(1), LR(1), and recursive descent parsers. Table driven parsers use a two-dimensional table data-structure to look up a non-terminals appropriate input. LL(1) parsers read input left-to-right, substitute non-terminals left-to-right, and look-ahead at most one word. LR(1) parsers read input left-to-right, substitute non-terminals right-to-left, and look-ahead at most one word. Lastly, and my favorite, recursive descent parsers, implemented using recursive function calls.

5 Case study

To test my understanding of these concepts, I wrote an infix notation calculator that that honors algebraic precedence. My implementation efficiently uses a direct-coded scanner and a recursive descent parser.[1]

6 Conclusion

Delving into compiler-construction has given me the confidence, ability, and understanding of language processing. I look forward to diving headfirst into natural language processing and the performance issues surrounding language processing, such as, data-parallelization, more efficient data-structures, and zero allocation implementations.

[1] https://github.com/MrOutput/jscalc

7 Appendix A

7.1 Parser

```
const Lexer = require("./Lexer");

class Calc {
    constructor() {
        this.stack = [];
        this.w = null;
        this.l = null;
    }

    static calc(a, op, b) {
        return this._chooseFn(op)(a, b);
    }

    static _chooseFn(op) {
        switch (op) {
        case "+": return this.sum;
        case "-": return this.diff;
        case "/": return this.quo;
        case "*": return this.prod;
        case "^": return this.expo;
        }
    }

    static expo(a, b) {
        return Math.pow(a, b);
    }

    static sum(a, b) {
        return a + b;
    }

    static diff(a, b) {
        return a - b;
    }

    static prod(a, b) {
        return a * b;
    }

    static quo(a, b) {
        return a / b;
    }
    _lrecmut(nt, pfn) {
        if (nt.call(this))
            return pfn.call(this);
    }

    _ophit(ops, nt, pfn, follow) {
        if (ops.some(this._istok, this)) {
            var _op = this.w.token;
            var a = this.stack.pop();
            this.w = this.l.next();
            if (nt.call(this)) {
                var b = this.stack.pop();
                this.stack.push(Calc.calc(a, _op, b));
                return pfn.call(this);
            }
        } else if (follow.some(this._istok, this) || this.w.category === null) {
            return true;
        }
    }

    _istok(o) {
        return (o === this.w.token);
    }

    exec(expr) {
        this.l = new Lexer(expr);
        this.w = this.l.next();

        if (!(this.e() || this.w.category === null))
            throw new SyntaxError(this.w.token);

        return (Calc.ans = this.stack.pop());
    }

    e() {
        return this._lrecmut(this.p, this.ep);
    }

    ep() {
        return this._ophit(["+", "-"], this.p, this.ep, [")"]);
    }

    p() {
        return this._lrecmut(this.x, this.pp);
    }
```

```
pp() {
    return this._ophit(["*", "/"], this.x, this.pp, ["+", "-", ")"]);
}

x() {
    return this._lrecmut(this.z, this.xp);
}

xp() {
    return this._ophit(["^"], this.z, this.xp, ["*", "/", "+", "-", ")"]);
}

z() {
    if (this.w.token === "(") {
        this.w = this.l.next();
        if (this.e()) {
            if (this.w.token === ")") {
                this.w = this.l.next();
                return true;
            }
        }
    } else if (["INT", "IDENT", "PCNT"].some(k => Lexer.cat[k] === this.w.category))
        this.stack.push(Calc._val(this.w));
        this.w = this.l.next();
        return true;
    }
}

static _val(w) {
    var n;
    if (w.category === Lexer.cat.INT || w.category === Lexer.cat.PCNT) {
        n = parseInt(w.token);
        if (w.category === Lexer.cat.PCNT)
            n /= 100;
    } else if (w.category === Lexer.cat.IDENT) {
        if (w.token === "pi") {
            n = Math.PI;
        } else if (w.token === "e") {
            n = Math.E;
        } else if (w.token === "ans") {
            n = this.ans;
        }
    }
    return n;
}
```

```
}
Calc.ans = 0;
module.exports = Calc;
```

7.2 Lexer

```
const Enum = require("./Enum");

class Lexer {
    constructor(string) {
        this.string = string;
        this.i = 0;
    }

    next() {
        var lexeme = { token: "", category: null };
        var c = this.getchar();

        lexeme.token = c;

        if (c === "0") {
            lexeme.category = Lexer.cat.ZERO;
        } else if (c === "*" || c === "/" ||
                   c === "+" || c === "^") {
            lexeme.category = Lexer.cat.OP;
        } else if (c === "e" || (c === "p" && this.getchar() === "i") ||
                  (c === "a" && this.getchar() === "n" && this.getchar() === "s")) {
            if (c === "p") {
                lexeme.token = "pi";
            } else if (c === "a") {
                lexeme.token = "ans";
            }
            lexeme.category = Lexer.cat.IDENT;
        } else if (c === "(") {
            lexeme.category = Lexer.cat.LP;
        } else if (c === ")") {
            lexeme.category = Lexer.cat.RP;
        } else if ((c === "-") || (c >= "1" && c <= "9")) {
            lexeme.token = "";
            lexeme.category = Lexer.cat.INT;
            if (c === "-") {
                this.rewind();
                this.rewind();
                var p = this.getchar();
                this.getchar();
                var n = this.getchar();
                this.rewind();
                if ((p === ")" || Lexer.isdigit(p)) &&
                    (n === "(" || n === "-" || Lexer.isdigit(n))) {
                    lexeme.category = Lexer.cat.OP;
                    lexeme.token = c;
                    return lexeme;
                }
            }
            do {
                lexeme.token += c;
                c = this.getchar();
            } while (Lexer.isdigit(c));
            if (c === "%")
                lexeme.category = Lexer.cat.PCNT;
            else
                this.rewind();
            this.rewind();
            var c = this.getchar();
            if (c === "-")
                throw new SyntaxError();
        } else if (c !== "") {
            throw new SyntaxError();
        }

        return lexeme;
    }

    static isdigit(c) {
        return (c >= "0" && c <= "9");
    }

    static isws(c) {
        return (c === " " || c === "\t");
    }

    getchar() {
        var c;
        do {
            c = this.string.charAt(this.i++);
        } while (Lexer.isws(c));
        return c;
    }

    rewind() {
        var c;
        do {
            c = this.string.charAt(--this.i);
        } while (Lexer.isws(c));
    }
}
```

```
Lexer.cat = new Enum(["OP", "ZERO", "LP", "RP", "INT", "IDENT", "PCNT"], 1);
module.exports = Lexer;
```

7.3 Enum

```
class Enum {
    constructor(keys, start = 0) {
        return keys.reduce((_enum, k) => (_enum[k] = start++, _enum), {});
    }
};
module.exports = Enum;
```