

Recursive Descent Parsing

Or: before you can parse, Grasshopper, first you must learn to parse...

Administrivia

- Today: RI -- “How to Build a Better Web Browser”
- Wed: PI rollout due

Lexical analysis, revisited

- Lexer: transformation from *character streams* to *tokens*
- Can only represent:
 - Sequences: "gattaca", "fnord", "haruspex", etc.
 - Indefinite repetition: "+", "*"
 - Finite choices/cases: "?", "[]", "|"
- Can be recognized (executed) in linear time
- Equivalent to a finite state machine

R.D. Parsing

- Ok, so you have a stream of tokens now
- How do you turn those into a parse tree?
- This turns out to be substantially different from lexing.
- Need to have (something morally equivalent to) recursion
 - Need recursiveness to handle arbitrary tree structure
 - Equivalent to handling nested rules

R.D. Parsing and CFGs

- Can only be described with a *context free grammar*
- Allows recursive rules
- Not finite state
 - Stack can be unboundedly deep
 - Needs more than a (bounded) finite # of states

CFGs and BNF

- Write rules in “Bakus-Naur Normal Form” (BNF)
- Rules made of 2 elements:
 - Terminals: Actual tokens that could be found in the data
 - “DoG”, “Haruspex”, “[0-9]+(\.[0-9]*)?”
 - Note: don’t have to be *constant* -- just have to denote a class of strings that are called a token
 - Non-terminals: Names of rules
- Rules must be of form:
 - $\text{LHS} := \text{term}_0 \text{op}_0 \text{term}_1 \text{op}_1 \dots \text{term}_N \text{op}_N$
 - LHS is a non-terminal (rule declaration)
 - term_i is a terminal or non-terminal
 - op_0 is an operator we’ve seen before: +, *, ?, |, ()

Example

EXPR := TERM ("+" TERM | "-" TERM)*

TERM := FACTOR ("*" FACTOR | "/" FACTOR)*

FACTOR := NUM | "(" EXPR ")" | "-" FACTOR
| "+" FACTOR

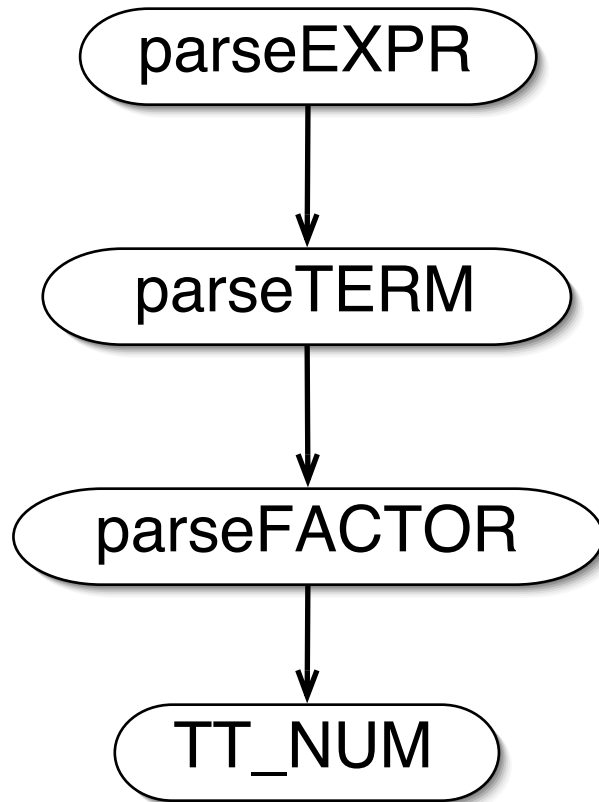
NUM := "[0-9]+"

Examples of the example

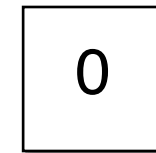
- 0
- -248
- +-+--+-192
- 004+19-23+86
- 211/437
- $((1+2/3)-(4/(5+6*7)))/8$

Parse trees of examples

- 0



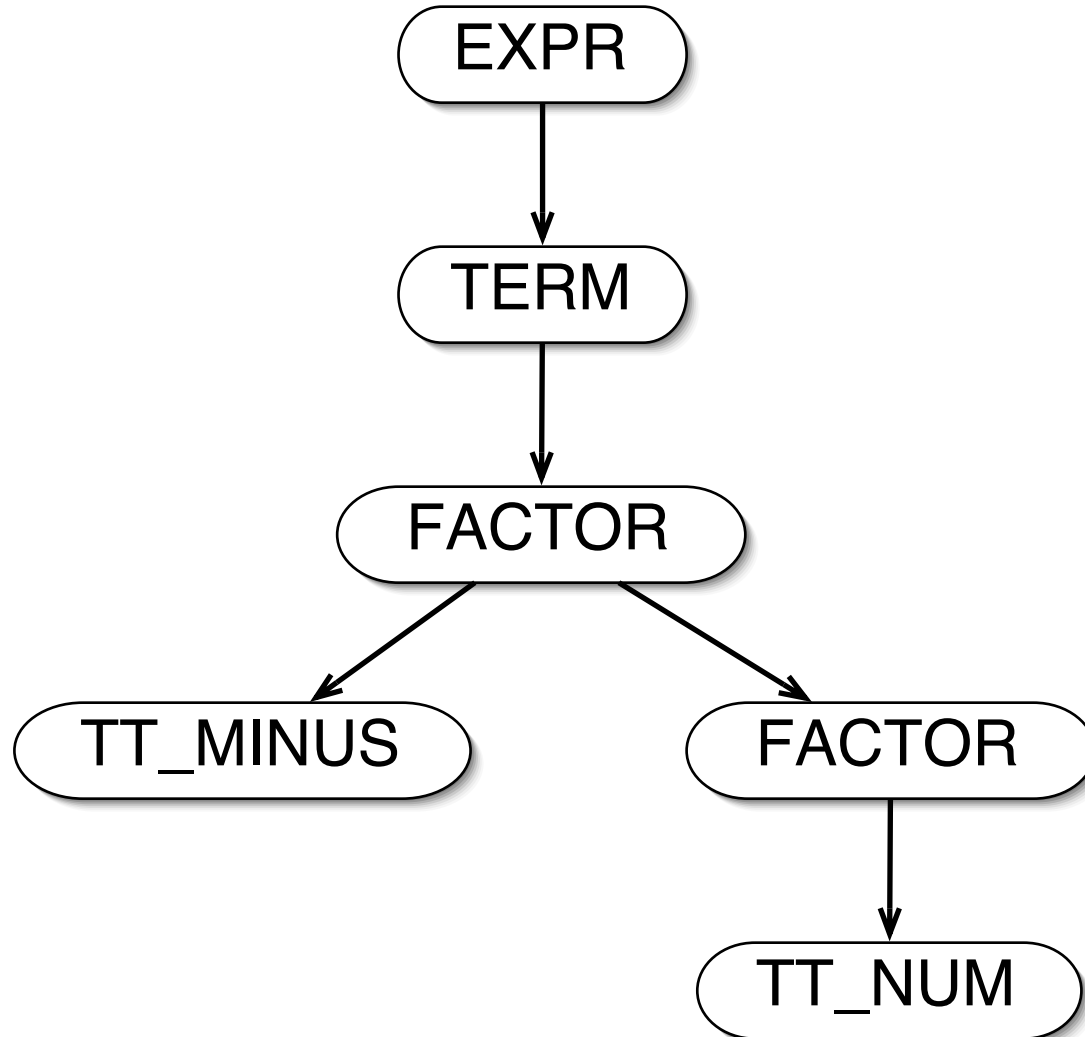
Call stack



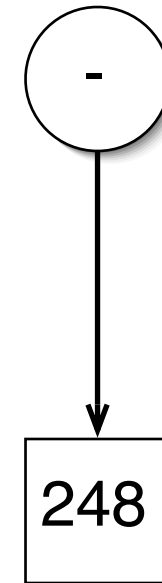
Parse tree

Parse trees of examples

- -248



Call stack

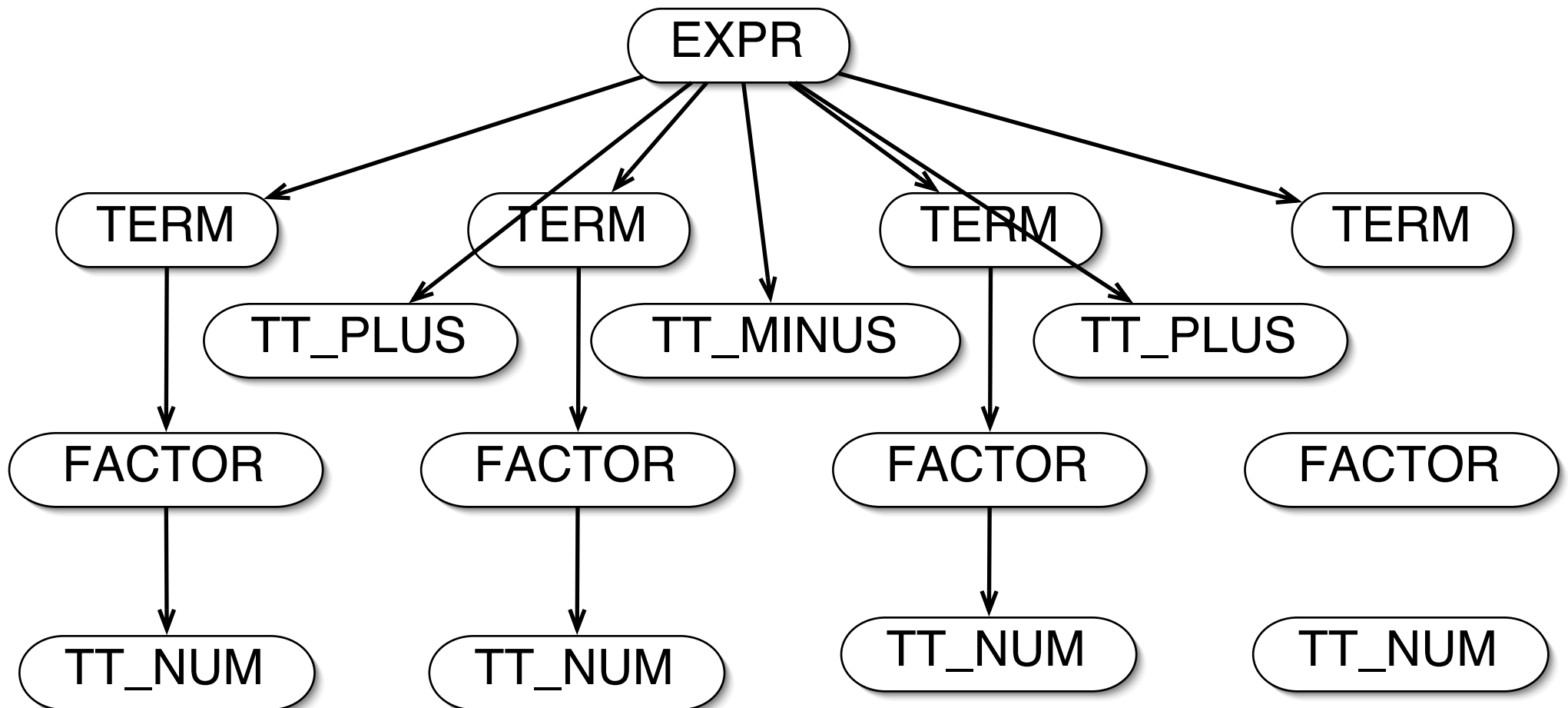


Parse tree

Parse trees of examples

- 004+19-23+86

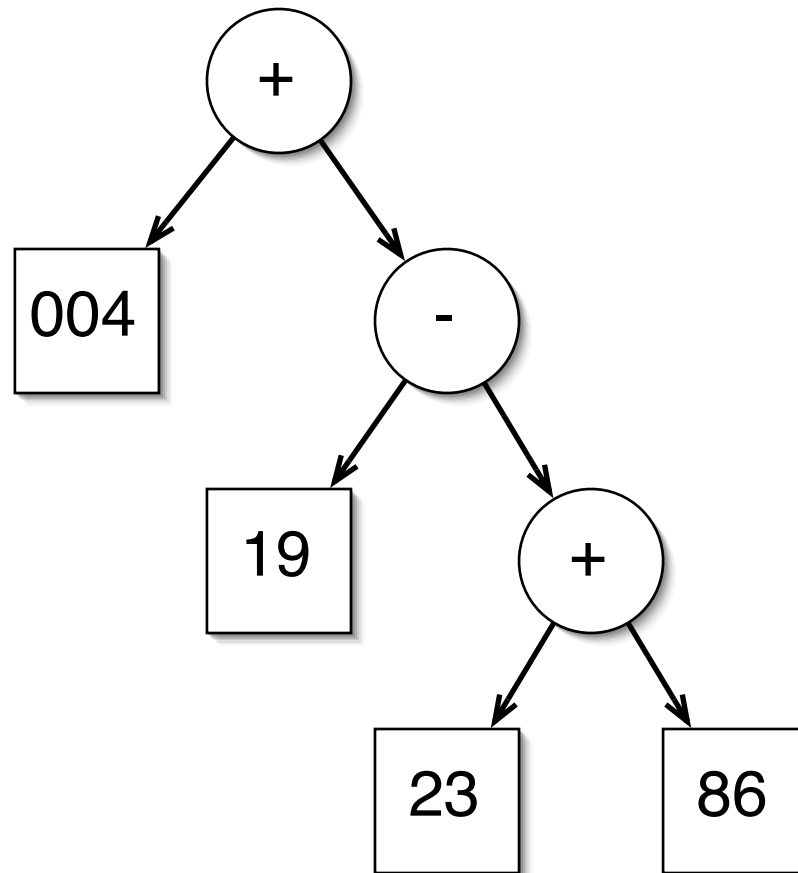
Call stack



Parse trees of examples

- $004+19-23+86$

Parse tree



Recursively defined langs.

- Note that the *definition* of `EXPR` depends on `EXPR`, definition of `FACTOR` depends on `FACTOR`, etc.
- Allows arbitrary nesting of expressions, etc.
- Acts like a recursive call
- Terminates when it sees the last thing in rule
- The “*” operator acts like a “while” statement
 - “While you keep seeing things of this form, keep extending this rule”
- The “|” and “?” act like “if” statements
 - “If you see this, then build this kind of statement; if you see that, build that kind of statement”

FACTOR → parseFACTOR()

```
public static ParseTree parseFACTOR(Lexer l) {
    // handle l.hasNext() and error trapping
    Token t=l.next();
    switch (t.getType()) {
        case TT_NUM:
            return new LeafNode(t.getNumber());
        case TT_LPAREN:
            ParseTree p=parseEXPR(l);    // recursion!
            t=l.next();
            if (t.getType()!=TT_RPAREN) { // error }
                return p;
        case TT_MINUS:
            return new MinusNode(parseFACTOR(l));
        case TT_PLUS:
            return new PlusNode(parseFACTOR(l));
        default: // error
    } }
```

TERM → parseTERM()

```
public static ParseTree parseTERM(Lexer l) {
    ParseTree p=parseFACTOR(l);
    while (l.hasNext()) {
        Token t=l.next();
        switch (l.getTokType()) {
            case TT_STAR:
                p=new MultiplyNode(p,parseFACTOR(l)); break;
            case TT_SLASH:
                p=new DivideNode(p,parseFACTOR(l)); break;
            default:
                throw new ParseError("Unexpected token " +
                    t + " while parsing FACTOR");
        }
    }
    return p;
}
```

Occasionally...

- Sometimes you'll hit a case like:
 - `FOO := "foo" (GUNK | JUNK)`
 - `GUNK := "(" STUFF ")"`
 - `JUNK := "{ " MORESTUFF "}"`
- When you're in the middle of `parseFOO()` what do you do?
- How do you know whether to call `parseGUNK()` or `parseJUNK()`?

Using Lexer.pushBack()

```
public static ParseTree parseFOO(Lexer l) {
    Token t=l.next();
    if (t.getType()!=TT_FOO) { // error }
    t=l.next();
    if (t.getType()==TT_LPAREN) {
        l.pushBack(t);
        return parseGUNK(l);
    }
    if (t.getType()==TT_RPAREN) {
        l.pushBack(t);
        return parseJUNK();
    }
    // error
}
```