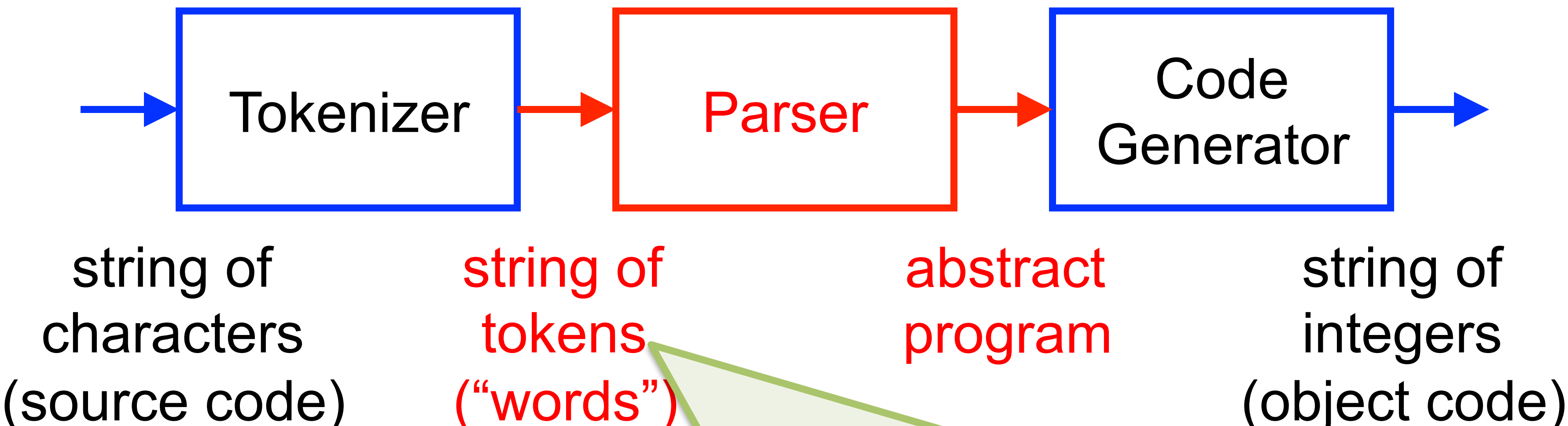
Recursive-Descent Parsing

BL Compiler Structure



Note that the parser starts with a string of *tokens*.

Plan for the BL Parser

- Design a context-free grammar (CFG) to specify syntactically valid BL programs
- Use the grammar to implement a ***recursive-descent parser*** (i.e., an algorithm to ***parse*** a BL program and construct the corresponding `Program` object)

Parsing

- A CFG can be used to ***generate*** strings in its language
  - "Given the CFG, construct a string that is in the language"
- A CFG can also be used to ***recognize*** strings in its language
  - "Given a string, decide whether it is in the language"
  - And, if it is, construct a derivation tree (or AST)

Parsing

- A CFG ca
  its langua
  - "Given t
    the lang

Parsing generally refers to this last step, i.e., going from a string (in the language) to its derivation tree or—for a programming language—perhaps to an AST for the program.

- A CFG can als ed to ***recognize*** strings in its l uage
  - "Given a st g, decide whether it is in the language"
  - And, if it is, construct a derivation tree (or AST)

A Recursive-Descent Parser

- One parse method per non-terminal symbol
- A non-terminal symbol on the right-hand side of a rewrite rule leads to a call to the parse method for that non-terminal
- A terminal symbol on the right-hand side of a rewrite rule leads to “consuming” that token from the input token string
- | in the CFG leads to “if-else” in the parser

Example: Arithmetic Expressions

expr	→	*expr add-op term* | *term*
term	→	*term mult-op factor* | *factor*
factor	→	`(` *expr* `)` | *digit-seq*
add-op	→	`+` | `-`
mult-op	→	`*` | `DIV` | `REM`
digit-seq	→	*digit digit-seq* | *digit*
digit	→	`0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

A Problem

expr → *expr add-op term* | *term*
term → *term mult-op factor* | *factor*
factor → `(` *expr* `)` | *digit-seq*
add-op → `+` | `-`
mult-op → `*` | `DIV` | `REM`
digit-seq → *digit digit-seq* |
digit → `0` | `1` | `2` | `3` | `4`

Do you see a problem with a recursive descent parser for this CFG? (Hint!)

A Solution

expr → *term* { *add-op term* }

term → *factor* { *mult-op factor* }

factor → `(` *expr* `)` | *digit-seq*

add-op → `+` | `-`

mult-op → `*` | `DIV` | `REM`

digit-seq → *digit digit-seq* | *digit*

digit → `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

A Solution

expr	→	*term* { *add-op term* }
term	→	*factor* { *mult-op factor* }
factor	→	(*expr*) | *digit-seq*
add-op		
mult-op		
digit-seq		
digit		

The special CFG symbols { and } mean that the enclosed sequence of symbols occurs *zero or more times*; this helps change a ***left-recursive*** CFG into an equivalent CFG that can be parsed by recursive descent.

The special CFG symbols *{* and *}* also simplify a non-terminal for a *number* that has no leading zeroes.

expr

term → *fa* *t-op factor }*

factor →) | *number*

add-op | -

mult-op → * | DIV | REM

number → 0 | *nz-digit { 0 | nz-digit }*

nz-digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

A Recursive-Descent Parser

- One parse method per non-terminal symbol
- A non-terminal symbol on the right-hand side of a rewrite rule leads to a call to the parse method for that non-terminal
- A terminal symbol on the right-hand side of a rewrite rule leads to "consuming" that token from the input token string
- | in the CFG leads to "if-else" in the parser
- *{...}* in the CFG leads to "while" in the parser

More Improvements

If we treat every *number* as a token, then things get simpler for the parser: now there are only 5 non-terminals to worry about.

expr →

term →

factor →

add-op → + |

mult-op → | DIV | REM

number → 0 | *nz-digit* { 0 | *nz-digit* }

nz-digit → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

More Improvements

If we treat every *add-op* and *mult-op* as a token, then it's even simpler: there are only 3 non-terminals.

expr →

term →

factor → (*expr*) | *number*

add-op → `+` | `-`

mult-op → `*` | `DIV` | `REM`

number → `0` | *nz-digit* { `0` | *nz-digit* }

nz-digit → `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

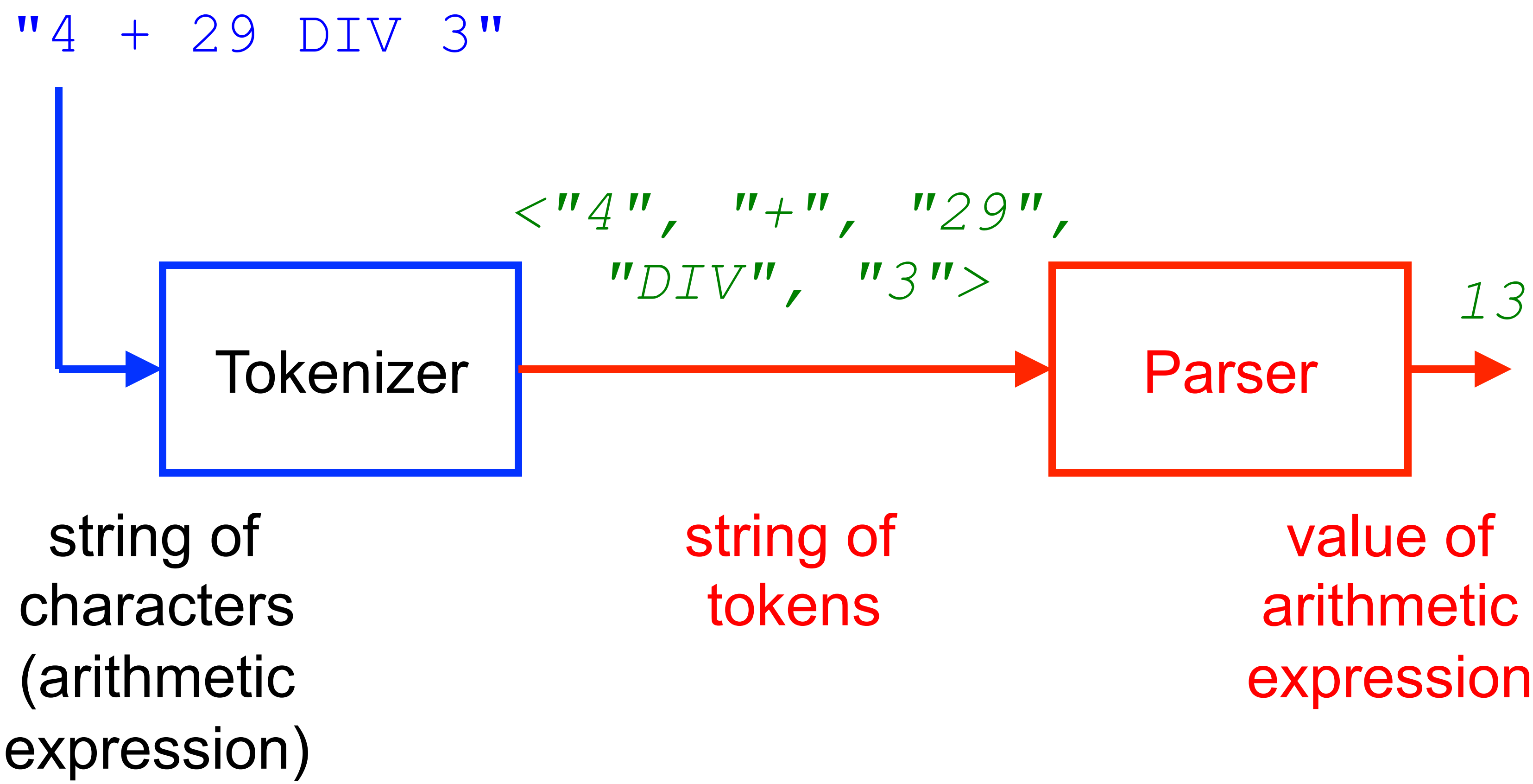
Can you write the tokenizer for this language, so every *number*, *add-op*, and *mult-op* is a token?

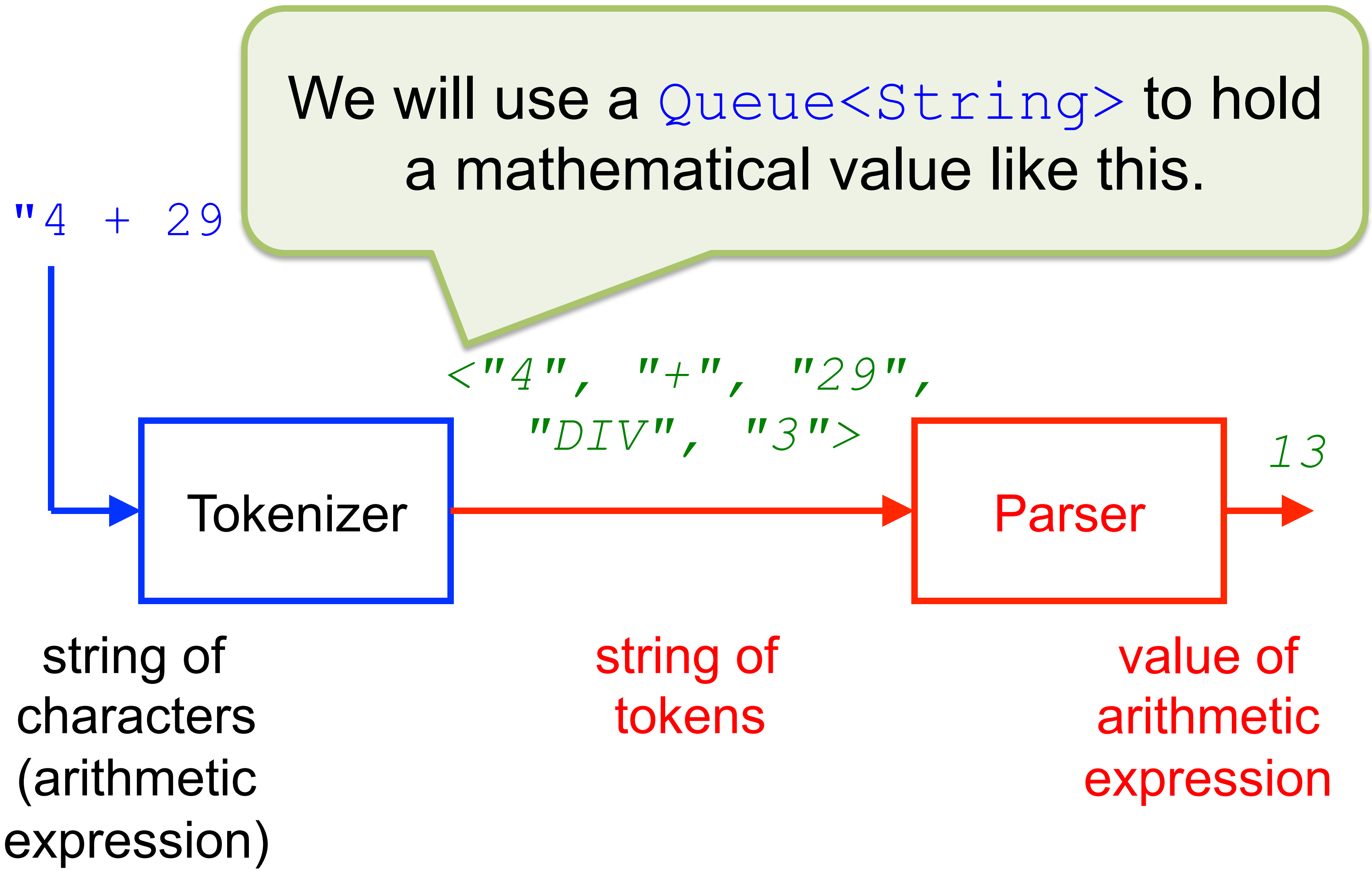
expr	→	*term* { *add-op* *term* }
term	→	*factor* { *mult-op* *factor* }
factor	→	`(` *expr* `)` | *number*
add-op	→	`+` | `-`
mult-op	→	`*` | `DIV` | `REM`
number	→	`0` | *nz-digit* { `0` | *nz-digit* }
nz-digit	→	`1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Evaluating Arithmetic Expressions

- For this problem, ***parsing*** an arithmetic expression means ***evaluating*** it
- The parser goes from a string of tokens in the language of the CFG on the previous slide, to the value of that expression as an **`int`**

Structure of Solution





We will use a Queue<String> to hold a mathematical value like this.
"4 + 29
<"4", "+", "29", "DIV", "3">
13
Tokenizer
Parser
string of characters (arithmetic expression)
string of tokens
value of arithmetic expression

Parsing an *expr*

- We want to parse an *expr*, which must start with a *term* and must be followed by zero or more (pairs of) *add-op*s and *terms*:

  expr → *term* { *add-op* *term* }

- An *expr* has an **`int`** value, which is what we want returned by the method to parse an *expr*

Contract for Parser for *expr*

```
/**
 * Evaluates an expression and returns its value.
 * ...
 * @updates ts
 * @requires
 * [an expr string is a proper prefix of ts]
 * @ensures
 * valueOfExpr = [value of longest expr string at
 *                start of #ts]  and
 * #ts = [longest expr string at start of #ts] * ts
 */
private static int valueOfExpr(Queue<String> ts) {...}
```

Parsing a *term*

- We want to parse a *term*, which must start with a *factor* and must be followed by zero or more (pairs of) *mult-op*s and *factor*s:

  term → *factor* { *mult-op* *factor* }

- A *term* has an **`int`** value, which is what we want returned by the method to parse a *term*

Contract for Parser for *term*

```
/**
 * Evaluates a term and returns its value.
 * ...
 * @updates ts
 * @requires
 * [a term string is a proper prefix of ts]
 * @ensures
 * valueOfTerm = [value of longest term string at
 *                start of #ts]  and
 * #ts = [longest term string at start of #ts] * ts
 */
private static int valueOfTerm(Queue<String> ts) {...}
```

Parsing a *factor*

- We want to parse a *factor*, which must start with the token `"("` followed by an *expr* followed by the token `")"`; or it must be a *number* token:

  factor → `(` *expr* `)` | *number*

- A *factor* has an **`int`** value, which is what we want returned by the method to parse a *factor*

Contract for Parser for *factor*

```
/**
 * Evaluates a factor and returns its value.
 * ...
 * @updates ts
 * @requires
 * [a factor string is a proper prefix of ts]
 * @ensures
 * valueOfFactor = [value of longest factor string at
 *                 start of #ts]  and
 * #ts = [longest factor string at start of #ts] * ts
 */
private static int valueOfFactor(Queue<String> ts){
  ...
}
```

Code for Parser for *expr*

```
private static int valueOfExpr(Queue<String> ts) {
  int value = valueOfTerm(ts);
  while (ts.front().equals("+") ||
         ts.front().equals("-")) {
    String op = ts.dequeue();
    if (op.equals("+")) {
      value = value + valueOfTerm(ts);
    } else /* "-" */ {
      value = value - valueOfTerm(ts);
    }
  }
  return value;
}
```

expr → *term* { *add-op* *term* }

add-op → + | −

```
private static int valueOfExpr(Queue<String> ts) {
  int value = valueOfTerm(ts);
  while (ts.front().equals("+") ||
         ts.front().equals("-")) {
    String op = ts.dequeue();
    if (op.equals("+")) {
      value = value + valueOfTerm(ts);
    } else /* "-" */ {
      value = value - valueOfTerm(ts);
    }
  }
  return value;
}
```

Code for Parser for *expr*

```
private static int valueOfExpr(Queue<String> ts) {
  int value = valueOfTerm(ts);
  while (ts.front().equals("+") |
         ts.front().equals("-")) {
    String op = ts.dequeue();
    if (op.equals("+")) {
      value = value + valueOfTerm(ts
    } else /* "-" */ {
      value = value - valueOfTerm(ts);
    }
  }
  return value;
}
```

This method is very similar to `valueOfExpr`.

Code for Parser for *expr*

```
private static int valueOfExpr(Queue<String> ts) {
  int value = valueOfTerm(ts);
  while (ts.front().equals("+") ||
         ts.front().equals("-")) {
    String op = ts.dequeue();
    if (op.equals("+")) {
      value = value + valueOfTerm(ts);
    } else /* "-" */ {
      value = value - valueOfTerm(ts);
    }
  }
  return value;
}
```

Look ahead one token in `ts` to see what's next.

Code for Parser for *expr*

```
private static int valueOfExpr(Queue<String> ts) {
  int value = valueOfTerm(ts);
  while (ts.front().equals("+") ||
         ts.front().equals("-")) {
    String op = ts.dequeue();
    if (op.equals("+")) {
      value = value + valueOfTerm(ts);
    } else /* "-" */ {
      value = value - valueOfTerm(ts);
    }
  }
  return value;
}
```

"Consume" the next token from `ts`.

Code for Parser for *expr*

```
private static int valueOfExpr(Queue<String> ts) {
  int value = valueOfTerm(ts);
  while (ts.front().equals("+") ||
         ts.front().equals("-")) {
    String op = ts.dequeue();
    if (op.equals("+")) {
      value = value + valueOfTerm(ts);
    } else /* "-" */ {
      value = value - valueOfTerm(ts);
    }
  }
  return value;
}
```

Evaluate (some of) the expression.

Code for Parser for *term*

```
private static int valueOfTerm(Queue<String> ts) {



}
```

Can you write the body, using `valueOfExpr` as a guide?

Code for Parser for *factor*

```
private static int valueOfFactor(
    Queue<String> ts) {
  int value;
  if (ts.front().equals("(")) {
    ts.dequeue();
    value = valueOfExpr(ts);
    ts.dequeue();
  } else {
    String number = ts.dequeue();
    value = Integer.parseInt(number);
  }
  return value;
}
```

factor → (*expr*) | *number*

```
private static int valueOfFactor(
      Queue<String> ts) {
   int value;
   if (ts.front().equals("(")) {
      ts.dequeue();
      value = valueOfExpr(ts);
      ts.dequeue();
   } else {
      String number = ts.dequeue();
      value = Integer.parseInt(number);
   }
   return value;
}
```

Code for Parser for *factor*

```
private static int valueOfFactor(
      Queue<String> ts) {
   int value;
   if (ts.front().equals("(")) {
      ts.dequeue();
      value = valueOfExpr(ts);
      ts.dequeue();
   } else {
      String number = ts.dequeue();
      value = Integer.parseInt(number);
   }
   return value;
}
```

Look ahead one token in `ts` to see what's next.

Code for Parser for *factor*

```
private static int valueOfFactor(
      Queue<String> ts) {
   int value;
   if (ts.front().equals("(")) {
      ts.dequeue();
      value = valueOfExpr(ts);
      ts.dequeue();
   } else {
      String number = ts.dequeue();
      value = Integer.parseInt(number);
   }
   return value;
}
```

What token does this throw away?

Co

Though method is called `parseInt`, it is not one of our parser methods; it is a static method from the Java library's `Integer` class (with **`int`** utilities).

```
private s
    Queue
  int valu
  if (ts.front()            (")) {
    ts.dequeue();
    value = valueOf        s);
    ts.dequeue();
  } else {
    String number = ts.dequeue();
    value = Integer.parseInt(number);
  }
  return value;
}
```

Code for Parser for *factor*

```
private static int valueOfFactor(
      Queue<String> ts) {
   int value;
   if (ts.front().equals("(")) {
      ts.dequeue();
      value = valueOfExpr(ts);
      ts.dequeue();
   } else {
      Strin
      value
   }
   return
}
```

Recursive descent: notice that `valueOfExpr` calls `valueOfTerm`, which calls `valueOfFactor`, which here may call `valueOfExpr`.

Code for Parser for *factor*

```
private static int valueOfFactor(
      Queue<String> ts) {
   int value;
   if (ts.front().equals("(")) {
      ts.dequeue();
      value = valueOfExpr(ts);
      ts.dequeue();
   } else {
      String number = ts.dequeue();
      value = Integer.parseInt(number);
   }
   return value;
}
```

How do you know this (indirect) recursion terminates?

A Recursive-Descent Parser

- One parse method per non-terminal symbol
- A non-terminal symbol on the right-hand side of a rewrite rule leads to a call to the parse method for that non-terminal
- A terminal symbol on the right-hand side of a rewrite rule leads to “consuming” that token from the input token string
- | in the CFG leads to “if-else” in the parser
- *{...}* in the CFG leads to “while” in the parser

Observations

- This is so formulaic that tools are available that can generate RDPs from CFGs
- In the lab, you will write an RDP for a language similar to the one illustrated here
  - The CFG will be a bit different
  - There will be no tokenizer, so you will parse a string of characters in a Java `StringBuilder`
    - See methods `charAt` and `deleteCharAt`

Resources

- Wikipedia: Recursive Descent Parser
  - http://en.wikipedia.org/wiki/Recursive_descent_parser
- Java Libraries API: `StringBuilder`
  - http://docs.oracle.com/javase/7/docs/api/