lexical analysis !

Language Processor Front-end Overview

Input

An input string is provided with access to its individual characters.

" `1 + 2 *10` "

Lexeme

Lexical Analysis
a.k.a. Scanning/Tokenization

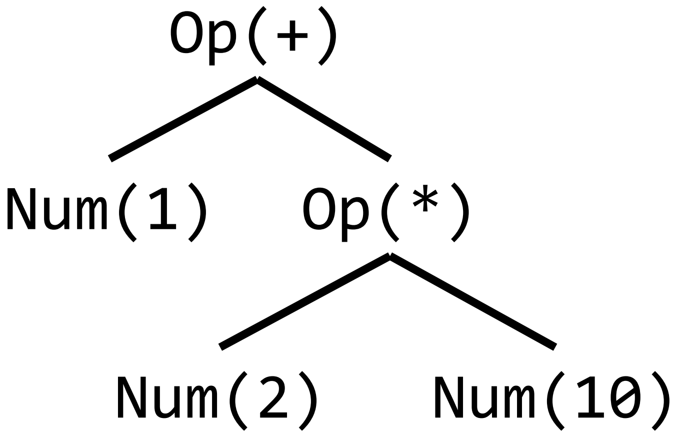
A tokenizer identifies *lexemes* in the input string and yields *tokens* while filtering out spaces and comments.

Num(1)	Op(+)	Num(2)	Op(*)	Num(10)

Token

Syntax Analysis
a.k.a. Parsing

A parser constructs a *parse tree* data structure out of the *tokens* produced during lexical analysis.



Parse Tree

Why separate lexical analysis from syntax analysis? Generally, both stages can be implemented simpler and more elegantly if their concerns are separated.

Lexical Analysis

- Today's focus is lexical analysis:

1. What are the key concepts and terms to understand?
2. How can you specify the textual patterns/rules of lexemes?
3. Given a specification, how do you approach tokenization?

Key Terms

- **Lexeme** - one or more characters in a string with a single unit of meaning
  - There are two number lexemes in the string **`"40 20"`**
  - Think of these as the *words* of our language
- **Pattern** - specification of the form or rules of a lexeme
  - Regular expressions like **`(1|2|4)(0)*`** can specify the patterns lexemes must match. You'll learn the details of these patterns this week.
- **Token** - a value in a program that has the token's type and often some associated data. Examples:
  - **`Number(40.0)`**
  - **`Number(20.0)`**
  - **`Op('+')`**
  - **`LeftParen, RightParen`**

Regular Expressions ("regex")

- A ***regular expression*** is a notation for specifying textual patterns
  - In language frontends they are used to specify lexeme patterns
  - Have everyday utility in searching for text in files and verifying user inputs

- Regular Expressions describe a Regular Grammar
  - In COMP455 you will explore the theoretical basis of regular grammars
  - Our goal is pragmatic: what are their rules and how can we *apply* them?

- A Regular Grammar is more constrained than the next kind of grammar we will find applications in (Context-Free Grammar)
  - The Chomsky Hierarchy (1956) identifies the broad classes of grammars according to their expressive power.

Regular Expressions... *pragmatically*

Operation: **Concatenation**

- The simplest regular expression "operator" is *concatenation*
- **Any two regular expressions, r_1 and r_2, can be concatenated to r_1r_2**
  - In practical notations, as we'll use and shown above, *concatenation is implicit*.
  - In formal notations you may see the concatenation operator explicitly represented with an underscore or dot, for example $r_1 \cdot r_2$
- Suppose r_1 is "c" and r_2 is "o", we can *concatenate* these two regular expressions to form regular expression r' as "co"
  - Further, if r_3 is "m" and r_4 is "p", you could concatenate $r'r_3r_4$ to form r_e "comp"
- The way to *read* concatenation is **"and then"**
  - r_e can be read as "c" *and then* "o" *and then* "m" *and then* "p"
- This operator *should* feel natural and obvious.
  - When you search a web page with Ctrl+F it is the only operator you have available.

Regular Expressions... *pragmatically*

Operation: **Alternation** via **|**

- *Union* is the more formal name for alternation because you are forming a grammar that is the union of two simpler grammars.

- **Any two regular expressions, r_1 and r_2, can be alternated with $r_1|r_2$**
  - The vertical bar symbol is effectively universal

- Suppose r_1 is "c" and r_2 is "o", we can *alternate* these two regular expressions to form regular expression ***r'*** as "**`c|o`**"
  - Further, if $\mathbf{r_3}$ is "m" and $\mathbf{r_4}$ is "p", you could form the alternation $r'|r_3|r_4$ to form r_e "**`c|o|m|p`**"

- The way to *read* alternation is **"or"**
  - r_e can be read as "c" *or* "o" *or* "m" *or* "p"

Regular Expressions... *pragmatically*

Operation: **Zero or More Repetitions** via *

- *Closure* is the more formal name for zero or more repetitions.

- **Any regular expression *r* can be *repeated zero or more times* with r***
    - The asterisk symbol, called the Kleene Star after its inventor, is universal.

- Suppose ***r*** is "c", we can *repeat r zero or more times with* "**c***"

- The way to *read* the star is **"is repeated zero or more times"**
    - ***r*** can be read as "c" is repeated zero or more times

- This operator *is strange* in isolation but *powerful* in composition...

Regular Expressions Compose by Combining Operators (1/2)

- You now know two operators, how can you *compose* them?

- Just like in *arithmetic expressions* you can group terms with parenthesis to make the order of operations explicit. Compare the following two regular expressions:

  (comp)|(sci)

  ("c" and then "o" and then "m" and then "p") OR ("s" and then "c" and then "i")
  matches either "comp" or "sci"

  (com)(p|s)(ci)

  ("c" and then "o" and then "m") and then ("p" OR "s") and then ("c" and then "i")
  matches "com" and then "p" or "s" and then "ci", so either "compci" or "comsci"

Composing Regular Expressions (2/2)

- When would it ever be valuable to specify *zero or more repetitions*?

- Suppose you specify a regular expression to match any single digit:

  $$\mathbf{r_{digit} = '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9'}$$

- Now, you *could try* specify a *whole number* as combinations of digits using only concatenation and alternation:

  $$\mathbf{r_{whole} = r_{digit} \mid (r_{digit}\ r_{digit}) \mid (r_{digit}\ r_{digit}\ r_{digit}) \mid (r_{digit}\ r_{digit}\ r_{digit}\ r_{digit})}$$

- But that only describes whole numbers made of 1 to 4 digits! This is where the Kleene star comes to the rescue:

  $$\mathbf{r_{whole} = r_{digit}\ r_{digit}{}^*}$$

- A whole number is "a digit *or* (a digit *and then* zero or more repetitions of a digit)"

- Breaking the rules of a regular expressions into *regular definitions* helps their legibility. Compare with:

  $\mathbf{r_{whole} = ('0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9')\ ('0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9')^*}$

The Fundamental Operators of Regular Expressions

The three regular expression operators you *need to know* are:

1. Any two regular expressions, r_1 and r_2, can be **concatenated** as $\boldsymbol{r_1 r_2}$
   "r1 AND THEN r2"

2. Any two regular expressions, r_1 and r_2, can be **alternated** as $\boldsymbol{r_1 | r_2}$
   "r1 OR r2"

3. Any regular expression r can be **repeated zero or more times** with **r***
   "**r is repeated zero or more times**"

<u>Composition is *the Very Big Deal*</u>: When you apply any of these operators you are composing another regular expression that can further be composed with other regular expressions.

You will learn additional regular expression operators that help you write patterns more succinctly. They are not fundamental. All other regex operators are defined in terms of the three operators above.

Regular Definitions

- A *regular definition* is a conventional notation to break down regular expressions into *named subexpressions*
  - Just like we did when forming a regular expression for whole numbers!

$$
\begin{aligned}
&d_1 \texttt{ -> } r_1 \\
&d_2 \texttt{ -> } r_2 \\
&\texttt{...} \\
&d_n \texttt{ -> } r_n
\end{aligned}
$$

- Regular definitions are non-recursive. This means each $\boldsymbol{r_n}$ is limited to*:*
  1. Terminal Characters, or
  2. Any *previously defined* non-terminal definitions (formally, $\boldsymbol{\{d_1 \ldots d_{n-1}\}}$)

- The next class of grammar we study (context-free) does not have restriction #2.

Regular Expressions - Additional Operators

- The three operators discussed last lecture are **fundamental**:
  - Concatenation
  - Alternation (Union)
  - Zero or More Repetitions (Closure / Kleene Star)

- There are very common real world patterns you will want to specify that are tedious using only those three operators.

- Most regex implementations offer additional operators for improved ergonomics. The ones we'll see today are built into egrep, Java, JavaScript, Python, etc.

Regex Character Classes - Character Lists (1/3)

- What regular expression matches single characters 'a' through 'f'?

  `r -> a | b | c | d | e | f`

- Character classes allow you to express the above pattern as:

  `r -> [abcdef]`

- When you need to match a specific set of individual characters, this is commonly helpful. For example, punctuations:

  `r -> [,.:;]`

Regex Character Classes - Character Ranges (2/3)

- What regular expression matches single characters 'a' through 'z'?

  `r -> a | b | c | d | e | f | ... | x | y | z`

- Character classes allow you to express the above pattern as:

  `r -> [a-z]`

  - How does a regex library *know* the range? It's based on ASCII ordinal numbers for each char. ASCII code for a is 97 and z is 122, so it accepts chars whose ASCII ordinals are between those two numbers.

- You can combine multiple ranges in singular regular expressions. For example, valid hexadecimal digits which are case insensitive:

  `r -> [a-fA-F0-9]`

Regex Character Classes - Escaping (3/3)

- You can directly capture *'s, ()'s, and |'s in character classes

  ```
  r -> [*()|]
  ```

- Why? The square brackets signify "treat these characters as character literals."

- You usually need to *escape* the characters [ ] and - to use them inside a character class.
  - How regex implementations handle escaping inside of character classes varies.
  - No point in memorizing, just search references when needed.

Regex Repetitions - **N** to **M** repetitions

- Often you will want a pattern matched between a ranged number of times

  $d_{2\text{-}4}$ `-> r r | r r r | r r r r`

- The **{N,M}** operator provides ***N to M repetitions*** semantics

  d_2 `-> r{2,4}`

- For **at most M** repetitions, 0 inclusive, you can leave off the `N`:

  $d_{<=M}$ `-> r{,M}`

- For **at least N** repetitions, you can leave off the `M`

  $d_{>=N}$ `-> r{N,}`

Regex Repetitions - Exactly **N** repetitions

- Often you will want a pattern matched a specific number of times

  d_5 **-> r r r r r**

- You could achieve this with N to M repetitions, but it's redundant:

  d_5 **-> r{5,5}**

- The **{N}** operator provides ***N repetitions*** semantics

  d_5 **-> r{5}**

Regex Repetitions - One or More Repetitions

- Often you will want *at least one* of some pattern

  `d -> r r*`

- Using the N to M Repetitions operator, you could as:

  `d -> r{1,}`

- This is *so commonly useful,* there's a special **+** operator for it:

  `d-> r+`

Regex Repetitions - Zero or One - "Optional"

- Often you will want *at most one* of some pattern

  `d -> r | ε`

- The empty string is **ε** and it matches against nothing.

- Using the N to M Repetitions operator, you could as:

  `d -> r{0,1}`

- This is *so commonly useful,* there's a special **?** operator for it:

  `d-> r?`

Regular Expression Operator Precedence

Highest

1. Repetitions (left binding, unary operators)
   - *
   - +
   - ?
   - {N,M}'s

2. Concatenation

3. | Alternations

Lowest

Case Study: The **loldigit** language

- **`digit        -> [0-9]`**
- **`out_louds    -> [ol]`**
- **`lol          -> 'l' 'o' 'l' out_louds*`**
- **`tokens       -> lol | digit`**

A Tokenizer Finds Lexemes and Yields Tokens

- It does so by iterating through the characters of an input string one-by-one
- To simplify the implementation of a tokenizer it is often helpful to be able to **"peek"** ahead of the current character by one additional character without actually ***taking*** it. Why is this helpful?
- When you start looking for the next lexeme you can peek ahead one character to know what type of lexeme it *should* be and jump to a subroutine to *take* it.
  - Variable names in most programming languages can't start with a number. This is so the language's tokenizer can peek at the first character of what's next and decide if it's going to be a number or not.
- If you did not know you reached the *end* of a lexeme until you *took* the next character after the *lexeme* you'd need to do gymnastics to "give it back" or use additional state to keep track of what it was.

Follow-along

- Let's explore the code in **lecture/ls35-lexical**
- The demo app we're working on is **tokens.c**
- The purpose of this app is to tokenize an input string using a simple architecture**.**
  - It demos the practices of *peeking* ahead at characters and taking them
  - It also demos *matching* characters using alternation