CS 1622: Lexical Analysis

Jonathan Misurda
jmisurda@cs.pitt.edu

Lexical Analysis

Problem: Want to break input into meaningful units of information

Input: a string of characters
Output: a set of partitions of the input string (tokens)

Example:

```
if(x==y) {
      z=1;
} else {
      z=0;
}
```

```
"if(x==y){\n\tz=1;\n} else {\n\tz=0;\n}"
```

Tokens

Token: A sequence of characters that can be treated as a single local entity.

Tokens in English:
- noun, verb, adjective, ...

Tokens in a programming language:
- identifier, integer, keyword, whitespace, ...

Tokens correspond to sets of strings:
- *Identifier*: strings of letters and digits, starting with a letter
- *Integer*: a non-empty string of digits
- *Keyword*: “else”, “if”, “while”, ...
- *Whitespace*: a non-empty sequence of blanks, newlines, and tabs

Why Tokens?

We need to classify substrings of our source according to their role.

Since a parser takes a list of tokens as inputs, the parser relies on token distinctions:
- For example, a keyword is treated differently than an identifier

Design of a Lexer

1. Define a finite set of tokens
   - Describe all items of interest
   - Depend on language, design of parser

recall `“if(x==y){\n\tz=1;\n} else {\n\tz=0;\n}”`
   - Keyword, identifier, integer, whitespace
   - Should “==” be one token or two tokens?

2. Describe which string belongs to which token

Lexer Implementation

An implementation must do two things:
1. Recognize substrings corresponding to tokens
2. Return the value or lexeme of the token

A token is a tuple *(type, lexeme)*:
`“if(x==y){\n\tz=1;\n} else {\n\tz=0;\n}”`
- Identifier: (id, ‘x’), (id, ‘y’), (id, ‘z’)
- Keywords: if, else
- Integer: (int, 0), (int, 1)
- Single character of the same name: ( ) = ;
- The lexer usually discards “non-interesting” tokens that don’t contribute to parsing, e.g., whitespace, comments

Lexical analysis looks easy but there are problems

Lexer Challenges

FORTRAN compilation rule: whitespace is insignificant
- Rule was motivated from the inaccuracy of card punching by operators

Consider:
- `DO 5I=1,25`
- `DO 5I=1.25`

- The first: a loop iterates from 1 to 25 with step 5
- The second: an assignment

Reading left-to-right, cannot tell if DO5I is a variable or DO statement until , or . is reached.

Lexer Challenges

C++ template syntax:

```
vector<student>
```

C++ stream syntax:

```
cin >> var
```

The problem:

```
vector<vector<student>>
```

Lexer Implementation

Two important observations:
- The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time.
- **Lookahead** may be required to decide where one token ends and the next one begins.

To describe tokens, we adopt a formalism based upon **Regular Languages**:
- Simple and useful theory
- Easy to understand
- Efficient implementations

Languages

Definition:

Let Σ be a set of characters.

A **language** over Σ is a set of strings of the characters drawn from Σ.

Examples:
Alphabet = English characters
Language = English sentences

Alphabet = ASCII
Language = C programs

Not every string on English characters is an English sentence
Not all ASCII strings are valid C programs

Notation

Languages are **sets of strings.**

Need some notation for specifying which set we want to designate a language.
- Regular languages are those with some special properties.
- The standard notation for regular language is using a **regular expression**

Regular Expressions

A single character denotes a set containing the single character itself:
‘x’ = { “x” }

Epsilon (ε) denotes an empty string (not the empty set):
ε = { “” }

Empty set is { } = $\emptyset$
size($\emptyset$) = 0

size(ε) = 1
length(ε) = 0

Compound REs

Alternation: if A and B are REs, then:
A | B = { s | s ∈ A or s ∈ B }

Concatenation of sets/strings:
AB = { ab | a ∈ A and b ∈ B }

Repetition (Kleene closure):
A* = $\cup_{i \geq 0} A^i$ where A^i = A...A (i times)

A* = {ε} + A + AA + AAA + ... (zero or more As)

Convenient Abbreviations

One or more:
A+ = A + AA + AAA + ... = A A* (one or more As)

Zero or one:
A? = A | ε

Character class:
[abcd] = a | b | c | d

Wildcard:
. (dot) matches any character (sometimes excluding newline)

Examples

Regular expressions to determine Java keywords:

```
if | else | while | for | int | …
```

A literal string like “if” is shorthand for the concatenation of each letter

Integer literal:

```
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit = [0123456789]
digit = [0-9]

integer = digit digit*
integer = digit+
```

Is this good enough?

Examples

Whitespace:

```
whitespace = [ \t\n]
```

C identifiers:
Start with a letter or underscore
Allow letters or underscores or numbers after the first letter
Cannot be a keyword

```
id = [a-zA-Z_][a-zA-Z_0-9]*
```

Examples

Valid Email Addresses:

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-
9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08\x0b\x0c\x0e-
\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])*")@(?:(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-
9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:25[0-5]|2[0-4][0-
9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-
9][0-9]?|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-
\x1f\x21-\x5a\x53-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

Java RegEx Support

```
import java.util.regex.Pattern;
import java.util.regex.Matcher;

Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
boolean b = m.matches();
```

Or:

```
boolean b = Pattern.matches("a*b", "aaaaab");
```

String class:

```
String s = new String(“aaaaab”);
boolean b = s.matches ("a*b");
```

Predefined Patterns in Java

Pattern	Description
`[abc]`	a, b, or c (simple class)
`[^abc]`	Any character except a, b, or c (negation)
`\d`	A digit: `[0-9]`
`\D`	A non-digit: `[^0-9]`
`\s`	A whitespace character: `[ \t\n\x0B\f\r]`
`\S`	A non-whitespace character: `[^\s]`
`\w`	A word character: `[a-zA-Z_0-9]`
`\W`	A non-word character: `[^\w]`
`^`	The beginning of a line
`$`	The end of a line
`\b`	A word boundary
`\B`	A non-word boundary
X`{`*n*`}`	*X*, exactly *n* times
X`{`*n*`,}`	*X*, at least *n* times
X`{`*n*`,`*m*`}`	*X*, at least *n* but not more than *m* times