Tokenization

Motivation

- Construct a working tokenizer
- See how language designers specify tokens

Recall

- First step of parsing: *tokenization*
- Idea: Input is just a big blob of characters
- We need some organization so we can process more conveniently

Recall

- Tokens = terminals
  - CFG has terminals and nonterminals
- We define terminals by means of regular expressions

Goal

- Input: A stream of characters
- Output: A list of tokens, where each token has:
  - Symbol
  - Lexeme
  - Line number

Code

- Here's how we might define a token in C#:

```
public class Token
{
    public string sym;
    public string lexeme;
    public int line;

    public Token(string sym, string lexeme, int line)
    {
        this.sym = sym;
        this.lexeme = lexeme;
        this.line = line;
    }
    public override string ToString(){
        return string.Format("[{0,10} {1,4} {2,25}]",this.sym,
           this.line,this.lexeme);
    }
```

Tokenize

- How to tokenize?
- For our purposes, we'll assume we have a text file that specifies the terminals
- Format: Series of lines with
  lhs -> regex
- Suppose we've parsed these and we have them ready to use

Tokenizing

- What about whitespace?
  - Some languages: Significant (ex: Python, FORTRAN (early versions))
  - Many languages: (Mostly) not significant (ex: C, Java, ...)
- We can handle this by means of a special case
  - Define terminal:
    whitespace -> \s+
- When we tokenize, we discard whitespace tokens
- If whitespace is significant to our language, tokenizer will retain these

Tokenizing

- What about comments?
- Assume we have a terminal:
  comment -> ...something...
- Tokenizer will discard comment tokens as well

Comments

- Some comments are easy to specify
- Example: C++ comments:
  comment -> //[^\n]*
- But what about multiline comments, like in C?
  - What would our regex be?

Comments

- First thought:
  comment -> /*.**/
- Since this is our first attempt, it's wrong. Why?
  - Two reasons!

Metacharacters

- Need to escape the metacharacters
- Attempt 2:
  comment -> /*.**/
- Still wrong! Why?
- Hint:

```
/* This is
  my comment */
```

Comments

- By default, dot does *not* match newlines
- New regex:
  comment -> (?s)/*.**/
- It's still wrong!
- Hint:

```
1 x = y+z; /*Compute sum*/
2 x = x*x; /*Square it*/
```

Greedy

- Need to be lazy:
  comment -> (?s)/*.*?*/
- Note: We can use verbose regex syntax to make it more readable:
  comment -> (?sx)/* .*? */

Comments

- What about nested comments?
  /* This is
    // a nested comment */
- Language designer must decide: Does the // hide the closing */ or not?

Strings

- What about strings?
- Here's one easy way:
  string -> "[^"]*"
- But this isn't always good enough.
- Why not?

Strings

- No way for a string to contain quotation marks
- How can we fix?
  - Depends on how we want to indicate literal quotation marks
  - Ex: C++, Java: \"
  - Ex: SQL: ""

Strings

- Using C++ style escaping:
  string -> (?v) " (\\" | [^"])* "
- Note that order of alternation is important!
  - Won't work if we specify it the other way around

Note

- If we're specifying the regex in literal code the syntax gets messy:
  new Regex("(?v) \" (\\\\\"|[^\"])* ")
- Even with @ strings:
  new Regex(@"(?v) "" (\\""|[^""])* ")

Strings

- Languages like Python or Javascript allow your choice of string delimiters: ’ or "
- This allows you to write code like:
  print(’He said, "Hello!"’)
  print(“The apple is John’s.”)
- How would we write a regex for this?

Strings

- Need to balance symbols:
  string -> (?v)
  (" (\\"|[^"])* ")|
  (’ (\\’|[^’])* ’)

Preprocessing

- In languages like C, where we have a preprocessor, we can also have preprocessor directives
- Ex: #include, #define, #if, #ifdef, #endif, #elif, #else, #warning, #error, #line
- These are handled at tokenization time

Tokenization

- Suppose we define these tokens:
  IDENTIFIER -> \w+
  NUM -> \d+
  IF -> if
  ELSE -> else
  ADDOP -> [-+]
  MULOP -> [*/]
- What is the tokenization of:
  foo + 7 * bar

Tokenization

- Indeterminite!
- The item “7” matches IDENTIFIER and NUM!
- So we might tokenize as:
  IDENTIFIER, ADDOP, IDENTIFIER, MULOP, IDENTIFIER
  or
  IDENTIFIER, ADDOP, NUM, MULOP, IDENTIFIER
- Probably the second one is the one we want

Tokenization

- So the order we try to match token patterns is important:
  NUM -> \d+
  IDENTIFIER -> \w*
  IF -> if
  ELSE -> else
  EQUALS -> =
  ADDOP -> [-+]
  MULOP -> [*/]
- We could also have changed IDENTIFIER to [A-Za-z_]\w*
- But we're not out of the woods yet...

Tokenization

- What is the tokenization of:
  if x = 4

Tokenization

- According to our specification:
  IDENTIFIER (if), IDENTIFIER (x), EQUALS, NUM (4)

Tokenization

- So we must move identifier to be last:
  NUM -> \d+
  IF -> if
  ELSE -> else
  EQUALS -> =
  ADDOP -> [-+]
  MULOP -> [*/]
  IDENTIFIER -> [A-Za-z_]\w*
- But it's still not good enough...
- What is the tokenization of:
  ifone = 42

Tokenization

- IF, IDENTIFIER (one), EQUALS, NUM (42)

Solution

- Need to use boundary qualifier:
  NUM -> \d+
  IF -> \bif\b
  ELSE -> \belse\b
  EQUALS -> =
  ADDOP -> [-+]
  MULOP -> [*/]
  IDENTIFIER -> [A-Za-z_]\w

Operators

- What about operators?
  - Either carefully order them or use lookaround
- Ex:
  - Try == before =
  - Try >= before >
    - Ex: RELOP -> >=|<=|>|<|==|!=
    - Notice order is significant within piped alternatives!
  - Try ** before *
- Lookaround:
  EQUALS -> =(?!=)

Routine

- Input: A string. Output: List of Token's
- Python-style pseudocode:

```
i=0
while i < length of input:
    for sym,regex in terminals:
        m = regex.match(input,i)
        if m != None and m.start == i:
            if sym != whitespace and sym != comment:
                T.append( Token(
                    sym,
                    m.group(0),
                    line
                ))
            line += m.group(0).count("\n")
            i = m.end()
            break
```

Assignment

- Write a program which takes two command line arguments. The first will be the name of a grammar file; the second is the name of a file to tokenize
- Tokenize the file and output a list of tokens neatly to the screen.
- If the file cannot be tokenized, raise an exception

Sources

- Aho, Lam, Sethi, Ullman. Compilers: Principles, Techniques, and Tools. 2nd ed.
- K. Louden. Compiler Construction: Principles and Practice
- Python Tutorial. http://www.python.org
- Regexp. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
- Dennie Van Tassel. http://www.gavilan.edu/csis/languages/comments.html#_Toc537101

Created using LaTeX.

Main font: Gentium Book Basic, by Victor Gaultney. See http://software.sil.org/gentium/
Monospace font: Source Code Pro, by Paul D. Hunt. See https://fonts.google.com/specimen/Source+Code+Pro and http://sourceforge.net/adobe
Icons by Ulisse Perusin, Steven Garrity, Lapo Calamandrei, Ryan Collier, Rodney Dawes, Andreas Nilsson, Tuomas Kuosmanen, Garrett LeSage, and Jakub Steiner. See http://tango-project.org