CSC488/2107 Winter 2019 — Compilers & Interpreters

https://www.cs.toronto.edu/~csc488h/

Peter McCormick
pdm@cs.toronto.edu
Agenda

• Recognize, Analyze, Transform

• Lexical analysis

• Building lexical analyzers
Recognize  Analyze  Transform

Frontend  Backend
Recognize

- Lexical structure
- Syntactic structure
- Highly language/syntax specific

Data flow:
- Stream of *Characters*
  - Stream of *Tokens*
  - *Parse Tree* (Concrete Syntax)
Analyze

- Semantic meaning
- Less language specific

Data flow:

- Parse Tree
- Abstract Syntax Tree (possibly with annotations and/or associated symbol tables)
Transform (Lower)

- Memory layout
- Optimization (optional)
- Code generation
- Very target specific

Data flow:

- Abstract Syntax Tree
- Intermediate Languages/Representations (optional)
- Target Machine Code
Lexical Analysis

Syntax Analysis

Semantic Analysis

Code Generation

Source Code

Tokens

Intermediate Language

Object Code

Characters

Parse tree

Machine code / Bytecode
C pre-processor
#include <stdio.h>

/* complete contents of stdio.h */
Pre-processed:

```c
#define PI 3.1415
float pi = PI;
```

Post-processed:

```c
float pi = 3.1415;
```
Lexical Analysis
Syntax Analysis
Semantic Analysis
Code Generation

Source Code
Characters
Parse tree
Tokens
Intermediate Language
Machine code / Bytecode
Object Code
Lexical Analysis
Recognizing the textual building blocks of source code
A **scanner** or **lexer** converts a stream of **characters** into a stream of **lexical tokens**
Characters

• Visual representation (human):
  • ASCII characters or Unicode code points

• Physical byte representation:
  • Fixed length: 7 bit ASCII, UCS-4
  • Variable length: UTF-8/16/32

• Integers to the compiler
Lexical Token

• One of a fixed set of distinguishing categories:
  • Identifiers
    • Reserved identifiers / keywords
  • Literal constants: numeric, string
  • Special punctuation (braces, symbols, etc.)
  • Comments
• Language specific
Scanner/Lexer

- Consumes character input
- Identifies lexical boundaries
- Emits a stream of tokens
- Identifies malformed input and emits errors
- Chooses what to ignore (comments, whitespace)
- Manages additional bookkeeping like source coordinates (input filenames, line and column numbers)
if x < y { v = 1 }
if \( x < y \) \{ v = 1 \}
if \( x < y \) \{ v = 1 \}
if x < y { v = 1 }
Careful language design choices can enable fast scanners
Building lexical analyzers
struct Token {
    enum {
        IF, LT, IDENT, LITERAL, ...
    } type;
    union {
        char *ident;
        int literal;
    }
    // more bookkeeping
};
data Token
  = If
  | Lt
  | Ident String
  | Literal Integer
  | ...
  | ...
Idea: Use finite automata (state machines) to recognize tokens out of a stream of characters
Example: Addition expressions
Example expressions

1

123+456

1+2+3+456
Lexical structure

• 2 token types
  • Plus
  • Positive integer literal
• No whitespace handling
\[ \Sigma \quad \text{— Vocabulary} \]

\[ \Sigma = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, + \} \]
Represent finite automata (state machines) using a state transition diagrams
State transition diagram: Plus
State transition diagram: Positive integer literals
Non-deterministic finite automata (NFA)
Deterministic finite automata (DFA)
# Table driven DFA

<table>
<thead>
<tr>
<th>Input \ State</th>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>error</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>S+</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Emit Plus</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>U+</td>
<td>Emit Literal</td>
</tr>
</tbody>
</table>

Notation: V to change state, V+ to change while consuming 1 input character
while True:
    c = curInput()
    if state is START:
        if c == '+':
            emitPlus()
            nextInput()
        elif c in digits19:
            save(c)
            state = LITERAL
            nextInput()
        else:
            error()
    elif state is LITERAL:
        if c in digits09:
            save(c)
            state = LITERAL
            nextInput()
        else:
            emitLiteral(getSaved())
            resetSaved()
            state = START
    if c is EOF: break
Regular Expressions

- A *regular expression* is a rigorous mathematic statement defining the members of a *regular set*

- Very compact means of specifying the structure of lexical tokens
Notation & Definitions
Let $\emptyset$ be the *empty* set
Let $\Sigma$ be a finite set of distinguished characters (the *vocabulary*).

May use quote marks to avoid confusion:

$$\Sigma = \{ \{'\,', '}'\}, ', ' \}$$
A string is defined inductively by cases:

1. The *empty* or *null* string, denoted \( \lambda \)
   - \( \emptyset \neq \lambda \)

2. A character from \( \Sigma \) is itself a string

3. The *concatenation* of two strings is a string
   - For any strings \( S \) and \( T \), both \( S \ T \) and \( T \ S \) are strings
   - For any string \( S \), \( \lambda \ S = S \lambda = S \)
Ø is also a *regular expression* denoting the empty set
Any string $S$ is a regular expression, denoting the set containing that string.
Forming regular expressions

For any two regular expressions $A$ and $B$, the following are also regular expressions:

1. **Alternation:** $A \mid B$
   - Set union

2. **Concatenation:** $A \ B$
   - Set of all strings formed by the concatenation of any string from $A$ and any string from $B$

3. **Kleene Closure:** $A^*$
   - Zero or more concatenations of $A$

4. **Parenthesis:** $( A )$
   - Disambiguation
Useful shorthands

1. **Positive Closure**: $A^+$
   
   - $A A^*$ (one or more concatenations)

2. **Optional**: $A?$
   
   - $A | \lambda$ (zero or one $A$)

3. **Complement**: $\text{Not}(A)$
   
   - Match anything from $\Sigma$ that does *not* match $A$

4. **Character ranges**: [$“A$ … “$Z$”]
   
   - “$A$” | “$B$” | … | “$Y$” | “$Z$”
   
   - When it’s clear what the “…” ranges over
Examples
Addition expressions tokens as regular expressions

\[ \Sigma = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, + \} \]

\[
\text{Digits19} = ( 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )
\]

\[
\text{Digits} = ( 0 | \text{Digits19} )
\]

\[
\text{Plus} = ( + )
\]

\[
\text{Literal} = ( \text{Digits19} \text{Digits}^* )
\]

\[
\text{Token} = \text{Plus} | \text{Literal}
\]
More examples (1)

Digit = “0” ... “9”
Letter = “a” ... “z” | “A” ... “Z”
Identifier = ( Letter | “_” ) ( Letter | Digit | “_”)*
More examples (2)

Digit1 = “1” … “9”
Digit = “0” | Digit1
HexDigit = Digit | “a” … “f” | “A” … “F”
DecLiteral = Digit1 Digit*
HexLiteral = “0” ( “x” | “X” ) HexDigit+
Literal = (“-“)? DecLiteral | HexLiteral
More examples (3)

EOL = ‘\r’ | ‘\n’ | ‘\r’ ‘\n’
PythonComment = ‘#’ Not(EOL)* EOL
CComment = ‘/‘ ‘*’ Not(‘*’ ‘/‘)* ‘*’ ‘/‘
Great... but can we use them in a lexer?
By Thompson’s construction, a regular expression can always be converted into an NFA.
NFA’s are equivalent to DFA’s

It’s always possible to convert an NFA into a DFA
DFA scanners can be implemented extremely efficiently

See re2c for an even faster, code generation approach
Scanner development options

- Write by hand
- Use scanner generator tools
  - Provide a specially formatted definition file containing regular expressions and code fragments
  - Generates source code implementing the scanner
  - GNU Flex, ANTLR, PLY (Python Lex-Yacc), Ragel, re2c
- Use built-in language support for regular expressions
scanner.py
if x < y { v = 1 }

IF
IDENT x
LT
IDENT y
LBRACE
IDENT v
EQ
INTEGER 1
RBRACE
import re

SPEC = r'''
(?P<IDENT> \[\_a-zA-Z\] \[\_a-zA-Z0-9\]* ) |
(?P<NUMBER> \[-\]? \[1-9\] \[0-9\]* ) |
(?P<LT> < ) |
(?P<EQ> = ) |
(?P<LBRACE> { ) |
(?P<RBRACE> } ) |
(?P<WS> \s+ )
'''

lex = re.compile(SPEC, re.VERBOSE).match
Next Week

• Syntax analysis & parsing
No tutorial on Tuesday Jan 22