CSCI312 Principles of Programming Languages

Chapter 3
Regular Expression and Lexer

Xu Liu
Recap
Clite: Lexical Syntax

**Input:** a stream of characters from the ASCII set, keyed by a programmer.

**Output:** a stream of *tokens* or basic symbols, classified as follows:

- **Identifiers**
  - e.g., Stack, x, i, push
- **Literals**
  - e.g., 123, 'x', 3.25, true
- **Keywords**
  - bool char else false float if int main true true while
- **Operators**
  - = || && == != < <= > >= + - * / !
- **Punctuation**
  - ; , { } ( )
Clite: Concrete Syntax

Based on a parse of its Tokens

; is a statement terminator

(Algol-60, Pascal use ; as a separator)

Rule for IfStatement is ambiguous:

“The else ambiguity is resolved by connecting an else with the last encountered else-less if.”

[Stroustrup, 1991]
Abstract Syntax Tree for
\[ z = x + 2*y; \]
Fig. 2.10
Compilers and Interpreters
Contents

3.1 Chomsky Hierarchy
3.2 Lexical Analysis
3.3 Syntactic Analysis
3.1 Chomsky Hierarchy

Regular grammar -- least powerful
Context-free grammar (BNF)
Context-sensitive grammar
Unrestricted grammar
Regular Grammar

Simplest; least powerful

Equivalent to:

- Regular expression
- Finite-state automaton

Right regular grammar: $\omega \in T^*$, $B \in N$

\[ A \rightarrow \omega B \]
\[ A \rightarrow \omega \]
Example

\( \text{Integer} \rightarrow 0 \text{ Integer} \mid 1 \text{ Integer} \mid \ldots \mid 9 \text{ Integer} \mid \)

\[
0 \mid 1 \mid \ldots \mid 9
\]
Regular Grammars

Left regular grammar: equivalent
Used in construction of tokenizers
Less powerful than context-free grammars
Not a regular language

\[
\{ a^n b^n \mid n \geq 1 \}
\]

i.e., cannot balance: ( ), { }, begin end

\[
A = a A b \mid \varepsilon
\]

\[
A = a B b
B = a B b \mid \varepsilon
\]
Context-free Grammars

BNF a stylized form of CFG
Equivalent to a pushdown automaton
For a wide class of unambiguous CFGs, there are table-driven, linear time parsers
Context-Sensitive Grammars

Production:
\[ \alpha \rightarrow \beta \quad |\alpha| \leq |\beta| \]
\[ \alpha, \beta \in (N \cup T)^* \]
i.e., lefthand side can be composed of strings of terminals and nonterminals
Undecidable Properties of CSGs

Given a string $\omega$ and grammar $G$: $\omega \in L(G)$

$L(G)$ is non-empty

Defn: Undecidable means that you cannot write a computer program that is guaranteed to halt to decide the question for all $\omega \in L(G)$. 
Unrestricted Grammar

Equivalent to:

- Turing machine
- von Neumann machine
- C++, Java

That is, can compute any computable function.
Review: Compilers and Interpreters

Source Program → Lexical Analyzer → Syntactic Analyzer → Semantic Analyzer → Code Optimizer → Code Generator → Machine Code
Lexical Analysis

Purpose: transform program representation
Input: printable ASCII characters
Output: tokens
Discard: whitespace, comments

Defn: A token is a logically cohesive sequence of characters representing a single symbol.
Example Tokens

Identifiers

Literals: 123, 5.67, 'x', true

Keywords: bool char ...

Operators: + - * / ...

Punctuation: ; , ( ) { }
Other Sequences

Whitespace: space tab

Comments

    // any-char* end-of-line

End-of-line

End-of-file
Why a Separate Phase?

Simpler, faster machine model than parser
75% of time spent in lexer for non-optimizing compiler
Differences in character sets
End of line convention differs
## Regular Expressions

<table>
<thead>
<tr>
<th>RegExpr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a character x</td>
</tr>
<tr>
<td>\x</td>
<td>an escaped character, e.g., \n</td>
</tr>
<tr>
<td>{ name }</td>
<td>a reference to a name</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>M N</td>
<td>M followed by N</td>
</tr>
<tr>
<td>M*</td>
<td>zero or more occurrences of M</td>
</tr>
<tr>
<td>RegExpr</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>M+</td>
<td>One or more occurrences of M</td>
</tr>
<tr>
<td>M?</td>
<td>Zero or one occurrence of M</td>
</tr>
<tr>
<td>[aeiou]</td>
<td>the set of vowels</td>
</tr>
<tr>
<td>[0-9]</td>
<td>the set of digits</td>
</tr>
<tr>
<td>.</td>
<td>Any single character</td>
</tr>
</tbody>
</table>
## Clite Lexical Syntax

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>anyChar</td>
<td>[ -~]</td>
</tr>
<tr>
<td>Letter</td>
<td>[a-zA-Z]</td>
</tr>
<tr>
<td>Digit</td>
<td>[0-9]</td>
</tr>
<tr>
<td>Whitespace</td>
<td>[ \t]</td>
</tr>
<tr>
<td>Eol</td>
<td>\n</td>
</tr>
<tr>
<td>Eof</td>
<td>\004</td>
</tr>
<tr>
<td>Category</td>
<td>Definition</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Keyword</td>
<td>bool</td>
</tr>
<tr>
<td>Identifier</td>
<td>{Letter}({Letter}</td>
</tr>
<tr>
<td>integerLit</td>
<td>{Digit}+</td>
</tr>
<tr>
<td>floatLit</td>
<td>{Digit}+.{Digit}+</td>
</tr>
<tr>
<td>charLit</td>
<td>‘{anyChar}’</td>
</tr>
<tr>
<td>Category</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Operator</td>
<td>`=</td>
</tr>
<tr>
<td>Separator</td>
<td>`:</td>
</tr>
<tr>
<td>Comment</td>
<td>`// ({anyChar}</td>
</tr>
</tbody>
</table>
Generators

Input: usually regular expression
Output: table (slow), code
C/C++: Lex, Flex
Java: JLex
Finite State Automata

Set of states: representation – graph nodes
Input alphabet + unique end symbol
State transition function

*Labelled (using alphabet) arcs in graph*

Unique start state
One or more final states
Deterministic FSA

Defn: A finite state automaton is deterministic if for each state and each input symbol, there is at most one outgoing arc from the state labeled with the input symbol.
A Finite State Automaton for Identifiers

What is a non-deterministic FSA?
Definitions

A configuration on an fsa consists of a state and the remaining input.

A move consists of traversing the arc exiting the state that corresponds to the leftmost input symbol, thereby consuming it. If no such arc, then:

– If no input and state is final, then accept.
– Otherwise, error.
An input is *accepted* if, starting with the start state, the automaton consumes all the input and halts in a final state.
Example

(S, a2i$) \vdash (I, 2i$)
\vdash (I, i$
\vdash (I, i$
\vdash (F, )

Thus: \((S, a2i$) \vdash^* (F, )\)
Chomsky Hierarchy

Regular grammar – least powerful
Context-free grammar (BNF)
Context-sensitive grammar
Unrestricted grammar
Contents

3.1 Chomsky Hierarchy
3.2 Lexical Analysis
3.3 Syntactic Analysis
Review: Compilers and Interpreters

Source Program → Lexical Analyzer → Syntactic Analyzer → Semantic Analyzer → Code Optimizer → Code Generator → Machine Code

Steps in the Compilation Process:
- Lexical Analyzer
- Syntactic Analyzer
- Semantic Analyzer
- Code Optimizer
- Code Generator
Syntactic Analysis

Phase also known as: parser
Purpose is to recognize source structure
Input: tokens
Output: parse tree or abstract syntax tree
A recursive descent parser is one in which each nonterminal in the grammar is converted to a function which recognizes input derivable from the nonterminal.