Lexical Analysis
Part I
Chapter 3

Interactions Between the Lexer & Parser

The Structure of a Compiler

Lexical Analysis
• What do we want to do? Example:
  ```c
  if (i == j)
    z = 0;
  else
    z = 1;
  ```
• The input is just a sequence of characters:
  ```
  \texttt{if (i == j)\{z = 0;\text{else}\{z = 1;\}}
  ```

Goal of Lexical Analysis
• Goal: Partition input string into substrings
  – And classify them according to their role -> token
  – Reduce length of program representation (remove spaces)

What’s a Token?
• Output of lexical analysis is a stream of \textit{useful} tokens
• A token is a syntactic category
  – In English:
    ```
    \texttt{noun, verb, adjective, ...}
    ```
  – In a programming language:
    ```
    \texttt{Identifier, Integer, IF, THEN, Whitespace, ...}
    ```
• Parser relies on the token distinctions:
  – E.g., identifiers are treated differently than keywords
Tokens
• Token represents a category. It consists of a token name + optional value
• Identifier: strings of letters or digits, starting with a letter
• Integer: a non-empty string of digits
• Keyword: “else” or “if” or “for” or ...
• IF: string “if”
• Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical Analysis Process

Lexical Analyzer: Implementation
• An implementation must do two things:
  1. Recognize substrings corresponding to tokens
  2. Return the value or lexeme of the token
     - The lexeme is the substring

Example
• Recall:
  \( \text{if (i == j)} \backslash n \right \{ tz = 0; \text{else} \backslash n \right \{ tz = 1; \}
• Token-lexeme pairs returned by the lexer:
  - (Whitespace, "\"\"")
  - (IF)
  - (OpenPar, "(")
  - (Identifier, ")")
  - (Relation, "==")
  - (Identifier, ")")
  - ...

Lexical Analyzer: Implementation
• The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.
• Examples: Whitespace, Comments

Lookahead.
• Two important points:
  1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time
  2. “Lookahead” may be required to decide where one token ends and the next token begins
     - Even our simple example has lookahead issues
       - \( i \) vs. \( if \)
       - \( == \) vs. \( === \)
Next

- We need
  - A language to describe the lexemes of each token
  - A language to resolve ambiguities
    - Is if two variables i and f?
    - Is === two equal signs == ==?

Regular Language

- There are several formalisms for specifying tokens
  - Regular languages are the most popular
    - Simple and useful theory
    - Easy to understand
    - Efficient implementations

Languages

**Def.** Let \( \Sigma \) be a set of characters. A *language over* \( \Sigma \) is a set of strings of characters drawn from \( \Sigma \) (\( \Sigma \) is called the *alphabet* )

Examples of Languages

- Alphabet = English characters
- Language = English sentences
- Not every string on English characters is an English sentence
- Alphabet = ASCII
- Language = C programs
- Note: ASCII character set is different from English character set

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- For lexical analysis we care about *regular languages*, which can be described using *regular expressions*.
  - A regular language is used to describe each token in a programming language

Regular Expressions and Regular Languages

- Each regular expression is a notation for a regular language (a set of words)
  - If \( A \) is a regular expression then we write \( L(A) \) to refer to the language denoted by \( A \)
Atomic Regular Expressions

- Single character: ‘c’
  \[ L(\text{`c'}) = \{ \text{`c'} \} \] (for any \textit{c} \in \Sigma)
- Concatenation: \(AB\) (where \(A\) and \(B\) are reg. exp.)
  \[ L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \} \]
- Example: ‘i’ and ‘f’ are reg. expressions so is ‘if’.
  \[ L(\text{`i'}\text{`f'}) = \{ \text{`if'} \} \]
  (we will abbreviate ‘i’ ‘f’ as ‘if’)

Compound Regular Expressions

- Union: \(A | B\) is a reg. expression where \(A, B\) are reg. expression.
  \[ L(A \mid B) = \{ s \mid s \in L(A) \text{ or } s \in L(B) \} \]

More Compound Regular Expressions

- So far we do not have a notation for infinite languages
- Iteration: \(A^*\) is a reg expression if \(A\) is a reg. expression.
  \[ L(A^*) = \{ \varepsilon \} \mid L(A) \mid L(AA) \mid L(AAA) \mid \ldots \]

Example: Keywords

- Keyword: “else” or “if” or “begin” or ...

  ‘else’ | ‘if’ | ‘begin’ | ...

  (Recall: ‘else’ abbreviates ‘e’ ‘l’ ‘s’ ‘e’)

Example: Integers

- Integer: a non-empty string of digits

  digit = ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

  number = digit digit*

  Abbreviation: \(A^+ = A \ A^*\)

Example: Identifier

- Identifier: strings of letters or digits, starting with a letter


  digit = digit digit*

  identifier = letter (letter | digit)*

  Is letter(letter* | digit*) the same?
Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

- `( ' | \t | \n')*`

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510) 643-1481
  \[\Sigma = \{ 0, 1, 2, 3, \ldots, 9, (, ), - \}\]
  - area = digit^3
  - exchange = digit^3
  - phone = digit^4
  - number = 'area' 'exchange' 'phone'

Example: Email Addresses

- Consider zijiang@cs.wmich.edu
  \[\Sigma = \text{letters} \cup \{ ., @ \}\]
  - name = letter+
  - address = name '@' name ('' name)^*'

Practice the following grep commands:

- grep 'cat' grepTest
  - you will find both "cat" and "vacation"
- grep '\<cat\>' grepTest
  - word boundary
- grep 'cat' grepTest
  - ignore the case
- grep '\<ega\.att\.com\>' grepTest
  - meta character
- egrep '[a-z][0-9][a-z][0-9]' grepTest
  - find Canadian postal code, only if it is in small case
- egrep '[a-z][0-9][a-z][0-9]' grepTest
  - ignore the case
- egrep is similar to grep, but sometimes cases patterns only work in egrep.
Summary

• Regular expressions describe many useful languages
• Next: Given a string $s$ and a rexp $R$, is $s \in L(R)$?
• But a yes/no answer is not enough!
• Instead: partition the input into lexemes
• We will adapt regular expressions to this goal

Procedure of Lexical Analysis

• Specifying lexical structure using regular expressions

  • Finite automata
    – Deterministic Finite Automata (DFAs)
    – Non-deterministic Finite Automata (NFAs)

  • Implementation of regular expressions
    RegExp $\Rightarrow$ NFA $\Rightarrow$ DFA $\Rightarrow$ Tables

Regular Expressions => Lexical Spec. (1)

1. Select a set of tokens
   • Number, Keyword, Identifier, ...

2. Write a R.E. for the lexemes of each token
   • Number = digit*
   • Keyword = 'if' | 'else' | ...
   • Identifier = letter (letter | digit)*
   • OpenPar = '('
   • ...

Regular Expressions => Lexical Spec. (2)

3. Construct $R$, matching all lexemes for all tokens
   
   $$R = R_1 | R_2 | R_3 | ...$$

   Facts: If $s \in L(R)$ then $s$ is a lexeme
   – Furthermore $s \in L(R_i)$ for some $i$
   – This “i” determines the token that is reported

Regular Expressions => Lexical Spec. (3)

4. Let the input be $x_1...x_n$
   ($x_1 ... x_n$ are characters in the language alphabet)
   • For $1 \leq i \leq n$ check
     $x_{1...i} \in L(R)$?

5. It must be that
   $x_{1...i} \in L(R_i)$ for some $i$ and $j$

6. Remove $x_{1...i}$ from input and go to (4)

Lexing Example

$$R = \text{Whitespace} | \text{Integer} | \text{Identifier} | '+'$$

• Scan “f + 3 * g”
  – “f” matches $R$, more precisely Identifier
  – “+” matches $R$, more precisely ‘+’
  – ...
  – The token-lexeme pairs are:
    (Identifier, “f”), (“+”), (“*”), (“g”), (Integer, “3”), (Whitespace, “ “), (“+”), (“*”), (Identifier, “g”)

• We would like to drop the WhiteSpace tokens
  – after matching WhiteSpace, continue matching
Ambiguities (1)

- There are ambiguities in the algorithm
- Example:
  \[ R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid '+' \]
- Parse "foo+3"
  - "f" matches \( R \), more precisely \( \text{Identifier} \)
  - But also "fo" matches \( R \), and "foo", but not "foo+"
- How much input is used? What if
  - \( x_1...x_i \in L(R) \) and also \( x_1...x_i \in L(R) \)
  - "Maximal munch" rule: Pick the longest possible substring that matches \( R \)

More Ambiguities

\[ R = \text{Whitespace} \mid 'new' \mid \text{Integer} \mid \text{Identifier} \]

- Parse "new foo"
  - "new" matches \( R \), more precisely 'new'
  - but also \( \text{Identifier} \), which one do we pick?
- In general, if \( x_1...x_i \in L(R_j) \) and \( x_1...x_i \in L(R_k) \)
  - Rule: use rule listed first (\( j \) if \( j < k \))
- We must list 'new' before Identifier

Error Handling

\[ R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid '+' \]

- Parse "=56"
  - No prefix matches \( R \): not "=", nor "=5", nor "=56"
- Problem: Can't just get stuck...
- Solution:
  - Add a rule matching all "bad" strings; and put it last
- Lexer tools allow the writing of:
  \[ R = R_1 \mid ... \mid R_n \mid \text{Error} \]
  - Token \text{Error} matches if nothing else matches

Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
  - To resolve ambiguities
  - To handle errors
- Good algorithms known (next)
  - Require only single pass over the input
  - Few operations per character (table lookup)