Lexical Analysis

Lecture 3-4
Course Administration

• PA1 due September 16 11:59:59 PM

• Read Chapters 1-3 of Red Dragon Book

• Continue Learning about Flex or JLex
Outline

• Informal sketch of lexical analysis
  - Identifies tokens in input string

• Issues in lexical analysis
  - Lookahead
  - Ambiguities

• Specifying lexers
  - Regular expressions
  - Examples of regular expressions
Recall: The Structure of a Compiler

Source  \rightarrow \text{Lexical analysis} \rightarrow \text{Tokens} \rightarrow \text{Parsing} \rightarrow \text{Interm. Language} \rightarrow \text{Code Gen.} \rightarrow \text{Machine Code}

Today we start
Lexical Analysis

• What do we want to do? Example:
  
  ```
  if (i == j)
    z = 0;
  else
    z = 1;
  ```

• The input is just a sequence of characters:
  
  ```
  \tif (i == j)\n  \tz = 0;\n  \telse\n  \tz = 1;
  ```

• Goal: Partition input string into substrings
  – And classify them according to their role
What's a Token?

- Output of lexical analysis is a stream of tokens
- A token is a syntactic category
  - In English:
    - noun, verb, adjective, ...
  - In a programming language:
    - Identifier, Integer, Keyword, Whitespace, ...
- Parser relies on the token distinctions:
  - E.g., identifiers are treated differently than keywords
Tokens

• Tokens correspond to **sets of strings**.

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

• **Integer**: *a non-empty string of digits*

• **Keyword**: "else" or "if" or "begin" or ...

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

• **OpenPar**: *a left-parenthesis*
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:
  
  \[\text{tif } (i == j)\text{\ newline } tz = 0;\text{\ newline } t\else\text{\ newline } tz = 1;\]

• Token-lexeme pairs returned by the lexer:
  
  - (Whitespace, "\t")
  - (Keyword, "if")
  - (OpenPar, "(")
  - (Identifier, "i")
  - (Relation, "==")
  - (Identifier, "j")
  - ...
Lexical Analyzer: Implementation

• The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.

• Examples: Whitespace, Comments

• Question: What happens if we remove all whitespace and all comments prior to lexing?
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 way to describe the lexemes of each token
  - A way to resolve ambiguities
    • Is if two variables i and f?
    • Is == two equal signs = =?
Regular Languages

- 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.
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’
  \[ L('c') = \{ "c" \} \text{  (for any } c \in \Sigma) \]

• Concatenation: \(AB\) (where A and B are reg. exp.)
  \[ L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \} \]

• Example: \(L('i' 'f') = \{ "if" \}
  \text{ (we will abbreviate ‘i’ ‘f’ as ‘if’)}
Compound Regular Expressions

• Union

\[ L(A \mid B) = \{ s \mid s \in L(A) \text{ or } s \in L(B) \} \]

• Examples:

  'if' | 'then' | 'else' = \{ "if", "then", "else"\}
  '0' | '1' | ... | '9' = \{ "0", "1", ..., "9" \}

  (note the ... are just an abbreviation)

• Another example:

  ('0' | '1') ('0' | '1') = \{ "00", "01", "10", "11" \}
More Compound Regular Expressions

- So far we do not have a notation for infinite languages
- Iteration: \( A^* \)
  \[ L(A^*) = \{ "" \} [ L(A) [ L(AA) [ L(AAA) [ ... \]
- Examples:
  \( '0'^* = \{ "", "0", "00", "000", ... \} \)
  \( '1' '0'^* = \{ \text{strings starting with 1 and followed by 0's} \} \)
- Epsilon: \( \varepsilon \)
  \[ L(\varepsilon) = \{ "" \} \]
Example: Keyword

- 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*

letter = 'A' | ... | 'Z' | 'a' | ... | 'z'
identifier = letter (letter | digit) *

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

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

\[(\ ' | \ 't' | \ 'n')^+\]

*(Can you spot a small mistake?)*
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 "necula@cs.berkeley.edu"

\[ \Sigma = \text{letter} | \cdot | @ \]

name = letter^+

address = name '@' name ('.name)^*
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
Outline

• Specifying lexical structure using regular expressions

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

• Implementation of regular expressions
  RegExp => NFA => DFA => 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

$$R = \text{Keyword} \mid \text{Identifier} \mid \text{Number} \mid \ldots$$
$$= R_1 \mid R_2 \mid R_3 \mid \ldots$$

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 $x_1 \ldots x_n$
   
   ($x_1 \ldots x_n$ are characters in the language alphabet)
   
   - For $1 \leq i \leq n$ check
     
     $x_1 \ldots x_i \in L(R)$ ?

5. It must be that

   $x_1 \ldots x_i \in L(R_j)$ for some $i$ and $j$

6. Remove $x_1 \ldots x_i$ from input and go to (4)
Lexing Example

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

- Parse “\( f +3 +g \)"
  - “\( f \)” matches \( R \), more precisely Identifier
  - “\( + \)” matches \( R \), more precisely ‘+’
  - ...
  - The token-lexeme pairs are
    (Identifier, “\( f \”), (+, “+”), (Integer, “3”))
    (Whitespace, “ “), (+, “+”), (Identifier, “g”)

- We would like to drop the \text{Whitespace} tokens
  - after matching \text{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
  • \( x_1 \ldots x_i \in L(R) \) and also \( x_1 \ldots x_K \in L(R) \)
  - “Maximal munch” rule: Pick the longest possible substring that matches \( R \)
More Ambiguities

R = Whitespace | 'new' | Integer | Identifier

• Parse “new foo”
  - “new” matches R, more precisely 'new'
  - but also 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

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

- 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:
  - \( R = R_1 \mid \ldots \mid R_n \mid \text{Error} \)
  - 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)
Finite Automata

• Regular expressions = specification
• Finite automata = implementation

• A finite automaton consists of
  - An input alphabet $\Sigma$
  - A set of states $S$
  - A start state $n$
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \rightarrow \text{input state}$
Finite Automata

• Transition

\[ s_1 \xrightarrow{a} s_2 \]

• Is read

In state \( s_1 \) on input “a” go to state \( s_2 \)

• If end of input (or no transition possible)
  - If in accepting state => accept
  - Otherwise => reject
Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition
A Simple Example

• A finite automaton that accepts only “1”

\[
\begin{array}{c}
\text{Start State} \\
\text{1} \\
\text{Accepting State}
\end{array}
\]

• A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state
Another Simple Example

• A finite automaton accepting any number of 1’s followed by a single 0
• Alphabet: \{0,1\}

• Check that “1110” is accepted but “110…” is not

Diagram:

[Diagram of a finite automaton accepting any number of 1’s followed by a single 0]
And Another Example

- Alphabet \{0,1\}
- What language does this recognize?
And Another Example

- Alphabet still \{ 0, 1 \}

- The operation of the automaton is not completely defined by the input
  - On input “11” the automaton could be in either state
Epsilon Moves

• Another kind of transition: \( \varepsilon \)-moves

• Machine can move from state A to state B without reading input
Deterministic and Nondeterministic Automata

• Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No $\varepsilon$-moves

• Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves

• Finite automata have finite memory
  - Need only to encode the current state
Execution of Finite Automata

• A DFA can take only one path through the state graph
  - Completely determined by input

• NFAs can choose
  - Whether to make $\varepsilon$-moves
  - Which of multiple transitions for a single input to take
Acceptance of NFAs

• An NFA can get into multiple states

• Input: 1 0 1

• Rule: NFA accepts if it can get in a final state
NFA vs. DFA (1)

- NFAs and DFAs recognize the same set of languages (regular languages)

- DFAs are easier to implement
  - There are no choices to consider
NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA

\[ \text{NFA} \]

\[ \text{DFA} \]

- DFA can be exponentially larger than NFA
Regular Expressions to Finite Automata

• High-level sketch

Diagram:

- Regular expressions
- DFA
- Table-driven Implementation of DFA
- Lexical Specification
- NFA
Regular Expressions to NFA (1)

- For each kind of rexp, define an NFA
  - Notation: NFA for rexp A

- For $\varepsilon$

- For input $a$
Regular Expressions to NFA (2)

- For $AB$

- For $A \mid B$
Regular Expressions to NFA (3)

• For $A^*$
Example of RegExp -> NFA conversion

- Consider the regular expression
  $$(1 \mid 0)^*1$$
- The NFA is

![NFA Diagram](image-url)
Next

- Regular expressions
  - Lexical Specification
- NFA
- DFA
  - Table-driven Implementation of DFA
NFA to DFA. The Trick

• Simulate the NFA
• Each state of DFA
  = a non-empty subset of states of the NFA
• Start state
  = the set of NFA states reachable through $\varepsilon$-moves from NFA start state
• Add a transition $S \xrightarrow{a} S'$ to DFA iff
  - $S'$ is the set of NFA states reachable from the states in $S$ after seeing the input $a$
    • considering $\varepsilon$-moves as well
NFA → DFA Example
NFA to DFA. Remark

• An NFA may be in many states at any time

• How many different states?

• If there are $N$ states, the NFA must be in some subset of those $N$ states

• How many non-empty subsets are there?
  - $2^N - 1 =$ finitely many
Implementation

- A DFA can be implemented by a 2D table $T$
  - One dimension is “states”
  - Other dimension is “input symbols”
  - For every transition $S_i \rightarrow^a S_k$ define $T[i,a] = k$

- DFA “execution”
  - If in state $S_i$ and input $a$, read $T[i,a] = k$ and skip to state $S_k$
  - Very efficient
Table Implementation of a DFA

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>
Implementation (Cont.)

• NFA -> DFA conversion is at the heart of tools such as flex or jlex

• But, DFAs can be huge

• In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations
PA2: Lexical Analysis

• Correctness is job #1.
  - And job #2 and #3!

• Tips on building large systems:
  - Keep it simple
  - Design systems that can be tested
  - Don’t optimize prematurely
  - It is easier to modify a working system than to get a system working