Lexical Analysis

Lecture 3

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

Lexical Analysis

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

- The input is just a string of characters:
  ```
  \[\text{if}\ (i == j)\ \text{\{z = 0;\ \text{\}}else\ \text{\{z = 1;\}}}\]
  ```

- Goal: Partition input string into substrings
  - Where the substrings are tokens

What’s a Token?

- A syntactic category
  - In English:
    - noun, verb, adjective, ...
  - In a programming language:
    - Identifier, Integer, Keyword, Whitespace, ...

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

What are Tokens For?

- Classify program substrings according to role

- Output of lexical analysis is a stream of tokens ...

- ... which is input to the parser

- Parser relies on token distinctions
  - An identifier is treated differently than a keyword
**Designing a Lexical Analyzer: Step 1**

- Define a finite set of tokens
  - Tokens describe all items of interest
  - Choice of tokens depends on language, design of parser

**Example**

- Recall
  \[
  \text{\texttt{\if (i == j)\textbackslash n\textbar\textbackslash tz = 0;\textbackslash n\textbar\textbackslash else\textbackslash n\textbar\textbackslash tz = 1;}}
  \]

- Useful tokens for this expression:
  - Integer, Keyword, Relation, Identifier, Whitespace, (, ), =, ;

  *N.B., (, ), =, ; are tokens, not characters, here*

**Designing a Lexical Analyzer: Step 2**

- Describe which strings belong to each token

  **Recall:**
  - **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

**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:
  \[
  \text{\texttt{\if (i == j)\textbackslash n\textbar\textbackslash tz = 0;\textbackslash n\textbar\textbackslash else\textbackslash n\textbar\textbackslash tz = 1;}}
  \]

**Lexical Analyzer: Implementation**

- The lexer usually discards "uninteresting" tokens that don’t contribute to parsing.

  **Examples:** Whitespace, Comments
True Crimes of Lexical Analysis

• Is it as easy as it sounds?
• Not quite!
• Look at some history . . .

Lexical Analysis in FORTRAN

• FORTRAN rule: Whitespace is insignificant
• E.g., VAR1 is the same as VA R1
• A terrible design!

Example

• Consider
  - DO 5 I = 1, 2.5
  - DO 5 I = 1.25

Lexical Analysis in FORTRAN (Cont.)

• 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

Lookahead

• Even our simple example has lookahead issues
  - i vs. if
  - = vs. ==

• Footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

Lexical Analysis in PL/I

• PL/I keywords are not reserved
  IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN
Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:
  DECLARE (ARG1, . . ., ARGN)

- Can't tell whether DECLARE is a keyword or array reference until after the ).
  Requires arbitrary lookahead!

- More on PL/I's quirks later in the course . . .

Lexical Analysis in C++

- Unfortunately, the problems continue today

- C++ template syntax:
  Foo<Bar>

- C++ stream syntax:
  cin >> var;

- But there is a conflict with nested templates:
  Foo<Bar<Bazz>>

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme

- Left-to-right scan => lookahead sometimes required

Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if two variables i and j?
    - 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 S be a set of characters. A language over S is a set of strings of characters drawn from S
Examples of Languages

- Alphabet = English characters
- Language = English sentences
- Not every string of 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
- The standard notation for regular languages is regular expressions.

Atomic Regular Expressions

- Single character
  \[ 'c' = \{ c^n \} \]
- Epsilon
  \[ \varepsilon = \{ \varepsilon \} \]

Compound Regular Expressions

- Union
  \[ A + B = \{ s | s \in A \text{ or } s \in B \} \]
- Concatenation
  \[ AB = \{ ab | a \in A \text{ and } b \in B \} \]
- Iteration
  \[ A^* = \bigcup_{i \geq 0} A^i \text{ where } A^i = A \ldots A \text{ } i \text{ times} \]

Regular Expressions

- Def. The regular expressions over \( S \) are the smallest set of expressions including
  \[ \varepsilon \]
  \[ 'c' \text{ where } c \in \Sigma \]
  \[ A + B \text{ where } A, B \text{ are rexp over } \Sigma \]
  \[ AB \]
  \[ A^* \text{ where } A \text{ is a rexp over } \Sigma \]

Syntax vs. Semantics

- To be careful, we should distinguish syntax and semantics.
  \[ L(\varepsilon) = \{ \varepsilon \} \]
  \[ L('c') = \{ c^n \} \]
  \[ L(A + B) = L(A) \cup L(B) \]
  \[ L(AB) = \{ ab | a \in L(A) \text{ and } b \in L(B) \} \]
  \[ L(A^*) = \bigcup_{i \geq 0} L(A)^i \]
Segue

- Regular expressions are simple, almost trivial
  - But they are useful!

- Reconsider informal token descriptions . . .

Example: Keyword

Keyword: "else" or "if" or "begin" or ...  
'else' + 'if' + 'begin' + . . .

Note: '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'
integer = digit digit'

Abbreviation: $A^* = AA^*$

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

( ' ' + 'n' + 't' )*

Example: Phone Numbers

- Regular expressions are all around you!
  - Consider (650)-723-3232

\[ \Sigma = \text{digits} \cup \{\text{,}, (, )\} \]

exchange = digit'
phone = digit'
area = digit'
phone_number = '(' area ')-' exchange '-' phone
**Example: Email Addresses**

- Consider anyone@cs.stanford.edu

\[ \Sigma = \text{letters } \cup \{.,@\} \]

\[ \text{name} = \text{letter}^* \]

\[ \text{address} = \text{name } '@' \text{name } '.' \text{name } '.' \text{name} \]

---

**Example: Unsigned Pascal Numbers**

\[
\begin{align*}
\text{digit} & = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9' \\
\text{digits} & = \text{digit}^* \\
\text{opt_fraction} & = ('.') \text{ digits} + \varepsilon \\
\text{opt_exponent} & = (E('+' + ' -' + '+') \text{ digits}) + \varepsilon \\
\text{num} & = \text{digits opt_fraction opt_exponent}
\end{align*}
\]

---

**Other Examples**

- File names
- Grep tool family

---

**Summary**

- Regular expressions describe many useful languages
- Regular languages are a language specification
  - We still need an implementation
- Next time: Given a string \( s \) and a rexp \( R \), is \( s \in L(R) \)?