Introduction to Lexical Analysis
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:
  
  ```java
  if (i == j)
  then
    Z = 0;
  else
    Z = 1;
  ```

• The input is just a string of characters:
  
  ```latex
  \textbf{\texttt{\textbackslash t\textbackslash i\textbackslash f (i == j)\textbackslash n\textbackslash then\n}}\textbackslash n\n\textbackslash t\textbackslash z = 0;\textbackslash n\textbackslash t\textbackslash el\textbackslash s\textbackslash n\n\textbackslash t\textbackslash t\textbackslash 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 used 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

• Recall
  ```
  \text{if} \ (i == j) \ \text{then} \ z = 0; \ \text{else} \ z = 1;
  ```

• Useful tokens for this expression:
  - Integer, Keyword, Relation, Identifier, Whitespace, (, ), =, ;
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:
  ```
  if (i == j) then
  z = 0;
  else
  z = 1;
  ```

• Token-lexeme groupings:
  - **Identifier**: i, j, z
  - **Keyword**: if, then, else
  - **Relation**: ==
  - **Integer**: 0, 1
  - **(, ), =, ;**: single character of the same name
Why do Lexical Analysis?

• Dramatically simplify parsing
  - The lexer usually discards “uninteresting” tokens that don’t contribute to parsing
    • E.g. Whitespace, Comments
  - Converts data early

• Separate out logic to read source files
  - Potentially an issue on multiple platforms
  - Can optimize reading code independently of parser
True Crimes of Lexical Analysis

- Is it as easy as it sounds?
- Not quite!
- Look at some programming language history . . .
Lexical Analysis in FORTRAN

• FORTRAN rule: Whitespace is insignificant

• E.g., VAR1 is the same as VA R1

• Footnote: FORTRAN whitespace rule was motivated by inaccuracy of punch card operators
A terrible design! Example

- **Consider**
  - DO 5 I = 1, 25
  - DO 5 I = 1.25

- The first is **DO 5 I = 1, 25**
- The second is **DO 5 I = 1.25**

- Reading left-to-right, cannot tell if **DO 5 I** is a variable or **DO** stmt. until after “,” is reached
Lexical Analysis in FORTRAN. Lookahead.

Two important points:

1. The goal is to partition the string. This is implemented by reading left-to-write, 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. ==
Another Great Moment in Scanning

• PL/1: Keywords can be used as identifiers:

\[
\text{IF THEN THEN THEN } = \text{ ELSE; ELSE ELSE } = \text{ IF}
\]

can be difficult to determine how to label lexemes
More Modern True Crimes in Scanning

• Nested template declarations in C++

```cpp
    vector<vector<int>> myVector
```

```cpp
    vector<vector<int>> myVector
```

```cpp
    (vector<vector<int>> myVector))
```
Review

• The goal of lexical analysis is to
  - Partition the input string into *lexemes* (the smallest program units that are individually meaningful)
  - 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 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* $\Lambda$ *over* $\Sigma$ is a set of strings of characters drawn from $\Sigma$.

($\Sigma$ is called the *alphabet* of $\Lambda$)
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 of strings we want our language to contain

• The standard notation for regular languages is regular expressions
Atomic Regular Expressions

• Single character
  \[ 'c' = \{ "c" \} \]

• Epsilon
  \[ \varepsilon = \{ "\"\" \} \]
Compound Regular Expressions

• Union

\[ A + B = \{ s \mid s \in A \text{ or } s \in B \} \]

• Concatenation

\[ AB = \{ ab \mid a \in A \text{ and } b \in B \} \]

• Iteration

\[ A^* = \bigcup_{i \geq 0} A^i \text{ where } A^i = A \ldots \text{i times} \ldots A \]
Regular Expressions

• **Def.** The *regular expressions over* $\Sigma$ *are the smallest set of expressions including*

\[
\epsilon \\
'c' \quad \text{where } c \in \Sigma \\
A + B \quad \text{where } A, B \text{ are rexp over } \Sigma \\
AB \\
A^* \quad \text{where } A \text{ is a rexp over } \Sigma
\]
Syntax vs. Semantics

• To be careful, we should distinguish syntax and semantics (meaning) of regular expressions

\[ L(\varepsilon) = \{"\"\} \]
\[ L('c') = \{"c"\} \]
\[ L(A + B) = L(A) \cup L(B) \]
\[ L(AB) = \{ab \mid a \in L(A) \text{ and } b \in L(B)\} \]
\[ L(A^*) = \bigcup_{i \geq 0} L(A^i) \]
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 1: Phone Numbers

• Regular expressions are all around you!
• Consider +46(0)18-471-1056

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

- country = digit digit
- city = digit digit
- univ = digit digit digit
- extension = digit digit digit digit
- phone_num = ‘+’country(’0‘)’city’–’univ’–’extension"
Example 2: Email Addresses

• Consider kostis@it.uu.se

\[
\begin{align*}
\Sigma & \quad = \quad \text{letters} \cup \{.,@\} \\
\text{name} & \quad = \quad \text{letter}^+ \\
\text{address} & \quad = \quad \text{name} '(@' \text{name} '.' \text{name} '.)' \text{name}
\end{align*}
\]
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 regular expression \( R \), is

\[ s \in L(R) \]?