Compiler Design

Chapter 2

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
The Structure of a Compiler

Today we start
Lexical Analysis and Parsing

- Normally a lexical analyzer is implemented as a function in a compiler
- When a parser needs the next token it calls lexical analyzer to get it

```repeat
    token = nextToken();
    do some actions on token
until end of input```
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  \else\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**:
  - Identifiers: *strings of letters or digits, starting with a letter*
  - Integers: *non-empty strings of digits*
  - Keywords: "else" or "if" or "begin" or ...
  - Whitespace: *non-empty sequences of blanks, newlines, and tabs*
  - OpenPars: *left-parentheses*
Lexical Analyzer: Implementation

- An implementation must do two things:
  
  1. Recognize substrings corresponding to tokens
  
  2. Return:
     1. The type or *syntactic category* of the token,
     2. the value or *lexeme* of the token (the substring itself).
Example

• **Our example again:**
  ```
  \tif (i == j) \nt \tz = 0; \nt \else \nt \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 t?
    • 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’
  \[ L('c') = \{ \text{“c”} \} \quad (\text{for any } c \in \Sigma) \]

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

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

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

- **Examples:**
  - ‘if’ | ‘then’ | ‘else’ = \{ “if”, “then”, “else”\}
  - ‘0’ | ‘1’ | ... | ‘9’ = \{ “0”, “1”, ..., “9” \}
    
    (note the ... are just an abbreviation)

- **Another example:**
  \[ L((’0’ | ’1’) (’0’ | ’1’)) = \{ “00”, “01”, “10”, “11” \} \]
More Compound Regular Expressions

- So far we do not have a notation for infinite languages
- Iteration: $A^*$
  \[ L(A^*) = \{ "" \} \mid L(A) \mid L(AA) \mid L(AAA) \mid ... \]
- Examples:
  - '0'* = { "", "0", "00", "000", ... }
  - '1' '0'* = { 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' | ...

('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*

\[
\text{letter} = 'A' \mid ... \mid 'Z' \mid 'a' \mid ... \mid 'z' \\
\text{identifier} = \text{letter} (\text{letter} \mid \text{digit})^* \\
\]

Is \((\text{letter}^* \mid \text{digit}^*)\) the same as \((\text{letter} \mid \text{digit})^*\) ?
Example: Whitespace

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

(' ' | '	' | '
')*

(Can you spot a subtle omission?)
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
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 $x_1...x_n$
   ($x_1 ... x_n$ are characters in the language alphabet)
   - For $1 \leq i \leq n$ check
     $x_1...x_i \in L(R)$?

5. It must be that
   $x_1...x_i \in L(R_j)$ for some $i$ and $j$

6. Remove $x_1...x_i$ 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 \( \text{Identifier} \)
  - “+” matches \( R \), more precisely ‘+’
  - ...
  - The token-lexeme pairs are
    - (Identifier, “f”), (+, “+”), (Integer, “3”)
    - (Whitespace, “ ”), (+, “+”), (Identifier, “g”)

- We would like to drop the \textbf{Whitespace} tokens
  - after matching \textbf{Whitespace}, continue matching
Ambiguities

- 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 \( x_1...x_k \in L(R) \)
  - “Maximal munch” rule: *Pick the longest possible substring that matches \( R \)*
More Ambiguities

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

- 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 = Whitespace | Integer | Identifier | '+'

- 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 = R₁ | ... | Rₙ | Error

  - Token Error matches if nothing else matches
Compiler Compilers

- Compiler compiler tools
  - Produce lexical analyzers (scanners) and syntax analyzers (parsers)
    - Lex, Flex, JavaCC, SableCC, ...
- JavaCC and SableCC are two tools for generating scanners and parsers written in Java.
- Scanners are automatically generated from lexical specifications (provided by the compiler compiler).
- Parsers are automatically generated from grammars.
- Lexical specification and grammar are contained in the same file, for both JavaCC and SableCC.
1. The compiler describes tokens using a set of regular expressions (REs).
2. Scanner generator automatically converts the set of REs into a set of equivalent finite state machines (FSMs).
3. FSMs represent computation and can be expressed as code in the implementation language.
4. This code is the scanner.
JavaCC File

options {...}  

PARSER_BEGIN(Example)
  class Example {}  
PARSER_END(Example)

Token: 
  <IF: "if">  
  <#LETTER: ["a"-"z"]>  
  <#DIGIT: ["0"-"9"]>  
  <NUM: (<DIGIT>)+>  
  <ID: <LETTER><LETTER>|<DIGIT>)*>  

Skip: { " " | \t | \n }  

void Goal();
Structure of JavaCC File

```java
options{
    /* Code to set various options flags */
}
PARSER_BEGIN(MyParser)
public class MyParser {
    /* Java program is placed here */
}
PARSER_END(MyParser)
TOKEN_MGR_DECLS:
{
    /* Declarations used by lexical analyser */
}
/* Token Rules and Actions */
/* JavaCC Rules and Actions - EBNF for language */
```
Structure of JavaCC File (cont.)

- **JavaCC** - Intermixes grammar rules and control code.
  - Syntax - Similar to regular expressions with the following points of note:
    - sequence: ["0"-"9"] rather than [0-9]
    - characters: "A"
    - not: [] is no characters, ~[] is all characters, ~["*" ] is all characters except "]*"
    - union: ["a","e","i","o","u","y"] are the vowel character
  - **PARSER\_BEGIN(MyParser)** - defines a class (e.g. MyParser) with a **main** function that:
    - constructs a lexical analyzer inputting characters from standard input (i.e. keyboard)
    - calls a generated function **Start()** to analyze input.
Structure of JavaCC File (cont.)

- **JavaCC** - Intermixes grammar rules and control code.
  - **TOKEN** - regular expressions defining valid tokens
  - **SKIP** - regular expressions defining strings to skip or ignore
  - **Start()** - analyzes input for listed tokens; 0 or more of the following are allowed:
    - `(〈IF〉 | 〈ID〉 | 〈NUM〉 | 〈REAL〉)*`
- **We will give a JavaCC file for the following tokens:**

```java
REs

if

[a-z][a-z0-9]*
[0-9]+
([0-9]+".[0-9]*")|([0-9]*".[0-9]+")
("--"[a-z]*"\n")|(" "|"\n"|"\t")+
```

**Tokens**

- IF
- ID
- NUM
- REAL
- no token, just white space
- error
options {
    DEBUG_PARSER = true; // Set to false to turn off token printing
}

PARSER_BEGIN(MyParser)
class MyParser {
    public static void main(String args[]) throws ParseException {
        new MyParser(System.in).Start();
        System.out.println("Parse successful");
    }
}
PARSER_END(MyParser)

TOKEN : {
    < IF: "if" >
    | < #DIGIT: ["0"-"9"] >
    | < ID: ["a"-"z"] (["a"-"z"] | <DIGIT>)* >
    | < NUM: <DIGIT> >
    | < REAL: ((<DIGIT>)+ "." (<DIGIT>)* ) | ((<DIGIT>)* "." (<DIGIT>)+ ) >
}
JavaCC Specification File (cont.)

```java
SKIP: {
    < "--" (~["\n", "\r"])* ("\n" | "\r" | "\r\n") >
    | " "
    | "\t"
    | "\n"
    | "\r"
}

void Start()
{
    {}
    {  (<IF> | <ID> | <NUM> | <REAL>)*
```
Steps for Lexical Analysis

- Create the Java specification file named Scanner.jj
- Use JavaCC to generate the scanner named MyParser.java
  - javacc Scanner.jj
- Compile all Java files in directory
  - javac -classpath . *.java
- Create a source file named Lexical.txt with contents
  12.34 5 if
  abc123 if4
  -- comment
- Analyze the contents of file Lexical.txt
  - java -cp . MyParser < Lexical.txt