CS412/413
Introduction to Compilers
Tim Teitelbaum

Lecture 2: Lexical Analysis
24 Jan 07

Outline
• Review compiler structure
• Compilation example
• What is lexical analysis?
• Writing a lexer
• Specifying tokens: regular expressions
• Writing a lexer generator

Simplified Compiler Structure

Source code
if (b == 0) a = b;

Understand source code

Intermediate code

Optimize

Generate assembly code

Assembly code
cmp 0, ecx
cmovz edx, ecx

Simplified Front End Structure

Source code (character stream)
if (b == 0) a = b;

Lexical Analysis
Syntax Analysis
Semantic Analysis

Correct program (AST representation)

Intermediate code

How It Works

Source code (character stream)
if (b == 0) a = b;

Lexical Analysis

Syntax Analysis (Parsing)

Semantic Analysis

Abstract syntax tree (AST)

Decorated AST

Intermediate code

Errors (incorrect program)
How It Works

First Step: Lexical Analysis

Intermediary Code Generation

Intermediate Code

Optimizations

Machine Optimizations and Code Generation

Tokens

- Identifiers: \( x, y11, \text{elsen}_i00 \)
- Keywords: if, else, while, break
- Constants:
  - Integer: \( 2, 1000, -500, 5L, 0x777 \)
  - Floating-point: \( 2.0, 0.0002, 0.02, 1.0e5, 0.e-10 \)
  - String: \( "x", \text{"He said, "Are you?"yn"} \)
  - Character: \( 'c', 000 \)
- Symbols: \( +, *, {, }, ++, <, \ll, [, ] >= \)
- Whitespace (typically recognized and discarded):
  - Comment: /** don't change this **/
  - Space: <space>
  - Format characters: <newline> <return>

Ad-hoc Lexer

- Hand-write code to generate tokens
- How to read identifier tokens?
  Token readIdentifier() {
    String id = "";
    while (true) {
      char c = input.read();
      if (!identifierChar(c))
        return new Token(ID, id, lineNumber);
      id = id + String(c);
    }
  }
- Problems
  - How to start?
  - What to do with following character?
  - How to avoid quadratic complexity of repeated concatenation?
  - How to recognize keywords?

Look-ahead Character

Scan text one character at a time
Use look-ahead character (next) to determine what kind of token to read and when the current token ends
char next;
... while (identifierChar(next)) {
  id = id + String(next);
  next = input.read();
}

Ad-hoc Lexer: Top-level Loop

class Lexer {
  InputStream s;
  char next;
  Lexer(InputStream _s) { s = _s; next = s.read(); }
  Token nextToken() {
    if (identifierFirstChar(next))
      return readIdentifier();
    if (numericFirstChar(next))
      return readNumber();
    if (next == "\"" return readStringConst();
    ... return nextToken();
  }
}

Problems

• Don’t know what kind of token we are going to read from seeing first character
  – if token begins with "i" is it an identifier?
  – if token begins with "2" is it an integer constant?
  – interleaved tokenizer code hard to write correctly, harder to maintain
  – In general, unbounded lookahead may be needed

Issues

• How to describe tokens unambiguously
  2.e0 \hspace{0.1cm} 20.e-01 \hspace{0.1cm} 2.0000
  \hspace{0.1cm} "x" \hspace{0.1cm} "\" \hspace{0.1cm} \"\"\"

• How to break up text into tokens
  if (x == 0) a = x<<1;
  if (x == 0) a = x<1;

• How to tokenize efficiently
  – tokens may have similar prefixes
  – want to look at each character ~1 time

Principled Approach

• Need a principled approach
  – lexer generator that generates efficient tokenizer automatically (e.g., lex, flex, JLex)
  – a.k.a., scanner generator

• Approach
  – Describe programming language’s tokens as set of regular expressions
  – Generate scanning automaton from that set of regular expressions

Language Theory Review

• Let \( \Sigma \) be a finite set
  – \( \Sigma \) called an alphabet
  – \( a \in \Sigma \) called a symbol

• \( \Sigma^* \) is the set of all finite strings consisting of symbols from \( \Sigma \)

• A subset \( L \subseteq \Sigma^* \) is called a language

• If \( L_1 \) and \( L_2 \) are languages, then \( L_1 L_2 \) is the concatenation of \( L_1 \) and \( L_2 \), i.e., the set of all pair-wise concatenations of strings from \( L_1 \) and \( L_2 \), respectively.

Language Theory Review, ctd.

• Let \( L \subseteq \Sigma^* \) be a language

• Then
  – \( L^0 = \{\} \)
  – \( L^{n+1} = L \cdot L^n \) for all \( n \geq 0 \)

• Examples
  – if \( L = \{a, b\} \) then
    \( L^1 = L = \{a, b\} \)
    \( L^2 = \{aa, ab, ba, bb\} \)
    \( L^3 = \{aaa, aab, aba, ab, bab, bba, bbb\} \)
    \( \ldots \)

Syntax of Regular Expressions

• Set of regular expressions (RE) over alphabet \( \Sigma \) is defined inductively by
  – Let \( a \in \Sigma \) and \( R, S \in \text{RE} \). Then:
    \( a \in \text{RE} \)
    \( \varepsilon \in \text{RE} \)
    \( \emptyset \in \text{RE} \)
    \( R \cup S \in \text{RE} \)
    \( R(S) \in \text{RE} \)
    \( RS \in \text{RE} \)
    \( R^* \in \text{RE} \)

• In concrete syntactic form, precedence rules, parentheses, and abbreviations
Semantics of Regular Expressions

- Regular expression \( T \in \text{RE} \) denotes the language \( L(R) \subseteq \Sigma^* \) given according to the inductive structure of \( T \):
  - \( L(a) = \{a\} \) the string "a"
  - \( L(\epsilon) = \{\epsilon\} \) the empty string
  - \( L(\emptyset) = \emptyset \) the empty set
  - \( L(R|S) = L(R) \cup L(S) \) alternation
  - \( L(RS) = L(R) L(S) \) concatenation
  - \( L(R^*) = \{\epsilon\} \cup L(R) \cup L(R^2) \cup L(R^3) \cup \ldots \) Kleene closure

Simple Examples

- \( L(R) \) = the "language" defined by \( R \)
  - \( L(\text{abc}) = \{\text{abc}\} \)
  - \( L(\text{hello|goodbye}) = \{\text{hello, goodbye}\} \)
  - \( L(\text{1(0|1)*}) = \) all non-zero binary numerals beginning with 1

Convenient RE Shorthand

- \( R^+ \) one or more strings from \( L(R) \): \( R(R^*) \)
- \( R? \) optional \( R \): \( (R|\epsilon) \)
- \( [abce] \) one of the listed characters: \( (a|b|c|e) \)
- \( [a-z] \) one character from this range: \( (a|b|c|d|e|…|y|z) \)
- \( [^ab] \) anything but one of the listed chars
- \( [^a-z] \) one character not from this range
- \( "abc" \) the string "abc"
- \( \backslash( \) the character '('

id=re named non-recursive regular expressions

More Examples

<table>
<thead>
<tr>
<th>Regular Expression R</th>
<th>Strings in L(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>digit = [0-9]</td>
<td>&quot;0&quot; &quot;1&quot; &quot;2&quot; &quot;3&quot; …</td>
</tr>
<tr>
<td>posint = digit+</td>
<td>&quot;8&quot; &quot;412&quot; &quot;…&quot;</td>
</tr>
<tr>
<td>int = ? posint</td>
<td>&quot;-42&quot; &quot;1024&quot; &quot;…&quot;</td>
</tr>
<tr>
<td>real = int ([. posint]?</td>
<td>&quot;-1.56&quot; &quot;12&quot; &quot;1.0&quot;</td>
</tr>
<tr>
<td>[a-zA-Z_][a-zA-Z0-9_]*</td>
<td>C identifiers</td>
</tr>
<tr>
<td>else</td>
<td>the keyword &quot;else&quot;</td>
</tr>
</tbody>
</table>

How To Break Up Text

```
elsen = 0; 1 else n = 0
2 elsen = 0
```

- REs alone not enough: need rule for disambiguation
- Most languages: longest matching token wins
- Ties in length resolved by prioritizing tokens
- Lexer definition = RE’s + priorities + longest-matching-token rule + token representation

Historical Anomalies

- PL/I
  - Keywords not reserved
    - IF IF THEN THEN ELSE ELSE;
- FORTRAN
  - Whitespace stripped out prior to scanning
    - DO 123 I = 1
    - DO 123 I = 1 , 2
- By and large, modern language design intentionally makes scanning easier
Summary

- Lexical analyzer converts a text stream to tokens
- Ad-hoc lexers hard to get right, maintain
- For most languages, legal tokens are conveniently and precisely defined using regular expressions
- Lexer generators generate lexer automaton automatically from token RE’s, precedence
- Next lecture: how lexer generators work

Reading

- IC Language spec
- JLEX manual
- CVS manual
- Links on course web home page

Groups

- If you haven’t got a full group lined up, hang around and talk to prospective group members today