CS412/413
Introduction to Compilers
Radu Rugina

Lecture 2: Lexical Analysis
28 Jan 04

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
Intermediate code

Generate assembly code
Assembly code
cmp $0, ecx
cmpw edx, ecx

Simplified Front End Structure

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

Lexical Analysis
Syntax Analysis
Semantic Analysis

Errors (incorrect program)

Correct program!
(AST representation)

Intermediate Code Generation
Intermediate code

More Precise Front End Structure

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

Lexical Analysis
Syntax Analysis
Semantic Analysis

Errors (incorrect program)

Intermediate code

How It Works

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

Lexical Analysis

Token stream
if (b == 0) a = b;

Syntax Analysis (Parsing)

Abstract syntax tree (AST)

Semantic Analysis

Decorated AST
How It Works

Decorated AST

Intermediate Code

Assembly Code

Intermediate Code Generation

Optimizations

Machine Optimizations and Code Generation

First Step: Lexical Analysis

Source code (character stream)

if (b == 0) a = b;

Lexical Analysis

Token stream

if (b == 0) a = b;

Syntax Analysis

Semantic Analysis

Tokens

- Identifiers: x y11 else _i00
- Integers: 2 1000 500 5L
- Floating point: 2.0 0.00020 .02 1.1e5 0.e-10
- Strings: "x" "He said, \"Are you?\""
- Comments: /** don't change this **/
- Keywords: if else while break
- Symbols: + * { } ++ < << [ ] >=

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);
    }
  }

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

Ad-hoc Lexer: Top-level Loop

class Lexer {
  InputStream s;
  char next;
  Lexer(InputStream _s) { s = _s; next = s.read(); }
  Token nextToken() {
    if (identifierChar(next))
      return readIdentifier();
    if (numericChar(next))
      return readNumber();
    if (next == '\n') return readStringConst();
    ...
  }
}
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 is hard to write correctly, harder to maintain
- Need a more principled approach: lexer generator that generates efficient tokenizer automatically (e.g., lex, flex, JLex)

Issues

- How to describe tokens unambiguously
  
  2.e0  20.e-01  2.0000
  “!” “#” “\" \"

- How to break text up 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

How to Describe Tokens?

- We can describe programming language tokens using regular expressions!
- A regular expression (RE) is defined inductively:
  \textbf{a} ordinary character stands for itself
  \textbf{e} the empty string
  \textbf{R|S} either R or S (alternation), where \textbf{R}, \textbf{S} = \textbf{RE}
  \textbf{RS} \textbf{R} followed by \textbf{S} (concatenation), where \textbf{R}, \textbf{S} = \textbf{RE}
  \textbf{R*} concatenation of a \textbf{RE} \textbf{R} zero or more times
  \textbf{R*} = \varepsilon\textbf{R}|\textbf{RR}|\textbf{RRR}|\textbf{RRRR}|...

Simple Examples

- A regular expression \( R \) describes a set of strings of characters denoted \( L(R) \)
- \( L(R) = \) the ”language” defined by \( R \)
  -- \( L(\textbf{abc}) = \{ \textbf{abc} \} \)
  -- \( L(\textbf{hello|goodbye}) = \{ \textbf{hello, goodbye} \} \)
  -- \( L(\textbf{1|0|1}^*) = \) all non-zero binary numbers
- We can define each kind of token using a regular expression

Convenient RE Shorthand

\( R^* \) one or more strings from \( L(R) \): \( R(R^*) \)
\( R? \) optional \( R \): \( (R|\varepsilon) \)
[\textbf{abc}] one of the listed characters: \( \{ \textbf{a|b|c|e} \} \)
[\textbf{a-z}] one character from this range: \( \{ \textbf{a|b|c|d|e|...|y|z} \} \)
[^\textbf{ab}] anything but one of the listed chars
[^\textbf{a-z}] one character not from this range

Examples

<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Strings in L(R)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{a}</td>
<td>“a”</td>
</tr>
<tr>
<td>\textbf{ab}</td>
<td>“ab”</td>
</tr>
<tr>
<td>\textbf{a</td>
<td>b}</td>
</tr>
<tr>
<td>\varepsilon</td>
<td>“”</td>
</tr>
<tr>
<td>( \textbf{ab}^* )</td>
<td>“ab” “abab” ...</td>
</tr>
<tr>
<td>( \textbf{a</td>
<td>e} \textbf{b} )</td>
</tr>
</tbody>
</table>
More Examples

Regular Expression Strings in L(R)
digit = [0-9] "0" "1" "2" "3" ...
posint = digit+ "8" "412" ...
int = n? posint ".42" "1024" ...
real = int (e | (. posint)) ".1.56" "12.1.0"
= ?[0-9]+(e | (. [0-9]+))

[a-zA-Z_] [a-zA-Z0-9_]* C identifiers

• Lexer generators support abbreviations – cannot be recursive

How To Break Up Text

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

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

Lexer Generator Spec

• Input to lexer generator:
  − list of regular expressions in priority order
  − associated action for each RE (generates appropriate kind of token, other bookkeeping)

• Output:
  − program that reads an input stream and breaks it up into tokens according to the REs. (Or reports lexical error --> "Unexpected character")

Example: JLex

```plaintext
%%
digits = 0[1-9][0-9]*
letter = [A-Za-z]
identifier = (letter)(letter)[[0-9_]]*
whitespace = [\t\n\r]+
%%
(whitespace) {/* discard */}
(digits) ( return new IntegerConstant(Integer.parseInt(yytext())); )
"if" ( return new IfToken(); )
"while" ( return new WhileToken(); )
...
( identifier) ( return new IdentifierToken(yytext()); )
```

Summary

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

Groups

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

• Submit questionnaire!