CS412/413
Introduction to Compilers
Radu Rugina

Lecture 2: Lexical Analysis
23 Jan 02

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 CX, 0
CMI V DX CX

Simplified Front End Structure

Source code
Lexical Analysis
Character stream
if (b == 0) a = b;

Syntax Analysis
Semantic Analysis

Correct program
(AST representation)

Errors
(Incorrect
program)

Intermediate Code
Generation

More Precise Front End Structure

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

Syntax Analysis
Semantic Analysis

Correct program
(AST representation)

Intermediate Code
Generation

How It Works

Source code
Lexical Analysis
Character stream
if (b == 0) a = b;

Syntax Analysis
(Parsing)

Abstract syntax
tree (AST)

Decorated
AST

Semantic Analysis
How It Works

Decoded ABT

Intermediate Code Generation

Intermediate Code

CNUM(a=a, b, L2)

a = b

LABEL(L1)

CNUM(a=a, b, L2)

a = 0

LABEL(L1)

Intermediate Code

Machine Optimizations and Code Generation

Assembly code

cmp ebx, 0

cmov [ebx+8], 0

First Step: Lexical Analysis

Source code

if (b == 0) a = b;

Lexical Analysis

Token stream

if (b == 0) a = b;

Optimizations

Syntax Analysis

Semantic Analysis

Tokens

- Identifiers: x y11 else _io0
- Keywords: if else while break
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.00020 .02 1.1e5 0.e-10
- Symbols: + * { } ++ < << [ ] >=
- Strings: “x” “He said, "Are you?"”
- Comments: /** don’t change this **/

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 == "") return readStringConst();
        ...
    }
}
**Problems**

- Don't know what kind of token we are going to read from seeing first character
  - if token begins with "?" 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, JLex)

**Issues**

- How to describe tokens unambiguously
  2.0.000
- 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:
  - ordinary character stands for itself
  - the empty string
  - either $R$ or $S$ (alternation), where $R, S = RE$
  - $R$ followed by $S$ (concatenation), where $R, S = RE$
- $R^*$ = concatenation of a RE $R$ zero or more times
- $R^+ = (R)RRRRRRRRRRRRRRR...

**Simple Examples**

- A regular expression $R$ describes a set of strings of characters denoted $L(R)$
- $L(R) =$ the "language" defined by $R$
  - $L(abc) = \{ abc \}$
  - $L(\text{hello} | \text{goodbye}) = \{ \text{hello}, \text{goodbye} \}$
  - $L(1(0)1)^*) =$ 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} $ 
- $[abc]$ one of the listed characters: $ (a|b|c|e) $
- $[a-z]$ one character from this range: $ (a|b|c|e)...[y|z]$ 
- $[^ab]$ anything but one of the listed chars
- $[^a-z]$ 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>$a$</td>
<td>&quot;a&quot;</td>
</tr>
<tr>
<td>$ab$</td>
<td>&quot;ab&quot;</td>
</tr>
<tr>
<td>$a</td>
<td>b$</td>
</tr>
<tr>
<td>$(ab)^*$</td>
<td>&quot;ab&quot; &quot;abab&quot; ...</td>
</tr>
<tr>
<td>$(a</td>
<td>\varepsilon) b$</td>
</tr>
</tbody>
</table>
More Examples

Regular Expression  | Strings in L(R)
digit = [0-9]        | "0" "1" "2" "3" ...
posit = digit+       | "8" "412" ...
int = -? posit       | "-42" "1024" ...
real = int (\s | (, posit)) | "-1.56" "12" "1.0"
                           | = -?[0-9]+(\s | (, [0-9]+))
[a-zA-Z]_[a-zA-Z0-9]*  | C identifiers

- Lexer generators support abbreviations
  - cannot be recursive

How To Break Up Text

def x:
    if x:
        x = 0
    else:
        x = 0

- REs alone not enough: need rule for choosing
- Most languages: longest matching token wins
  - even if a shorter token is only way
- 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

```%
% digit = 0([1-9][0-9]*)
letter = [A-Za-z]
[identifier = (letter)(([letter][0-9]+))]
[whitespace = [ ]|\t|\n]+$%

(whitespace) /* discard */
(digit) { return new IntegerConstant(Integer.parseInt(yytext)); }
"r" { return new Token(TOKEN); }
"while" { return new Whilertoken(); }
... (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
- Send mail to cs412 if you still cannot
  make a full group

- Submit questionnaire!