Lexical Analysis: Regular Expressions

CS 6620
January 26, 2010

Last Time ...

- A program that translates a program in one language to another language
- The essential interface between applications & architectures
- Typically lowers the level of abstraction
  - Analyzes and reasons about the program & architecture
- We expect the program to be optimized, i.e., better than the original
  - Ideally exploiting architectural strengths and hiding weaknesses

Phases of a Compiler

Source program

Lexical Analyzer
- Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- Characters read from a file are buffered
- Makes use of the theory of regular expressions and finite state machines
- Lex and Flex are tools that construct lexical analyzers from regular expression specifications

Parser
- Convert a linear structure – sequence of tokens – to a hierarchical tree-like structure – an AST
- The parser imposes the syntax rules of the language
- Work should be linear in the size of the input (else unusable) → type consistency cannot be checked in this phase
- Deterministic context free languages and pushdown automata for the basis
- Bison and yacc allow a user to construct parsers from CFG specifications

Semantic Analysis
- Calculates the program’s "meaning"
- Rules of the language are checked (variable declaration, type checking)
- Type checking also needed for code generation (code gen for a + b depends on the type of a and b)

Intermediate Code Generation
- Makes it easy to port compiler to other architectures (e.g., Pentium to MIPS)
- Can also be the basis for interpreters (such as in Java)
- Enables optimizations that are not machine specific

Target program

Phases of a Compiler

Source program

Lexical Analyzer
- Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- Characters read from a file are buffered
- Makes use of the theory of regular expressions and finite state machines
- Lex and Flex are tools that construct lexical analyzers from regular expression specifications

Target program

Phases of a Compiler

Source program

Lexical Analyzer
- Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- Characters read from a file are buffered
- Makes use of the theory of regular expressions and finite state machines
- Lex and Flex are tools that construct lexical analyzers from regular expression specifications

Target program

Phases of a Compiler

Source program

Lexical Analyzer
- Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- Characters read from a file are buffered
- Makes use of the theory of regular expressions and finite state machines
- Lex and Flex are tools that construct lexical analyzers from regular expression specifications

Target program
Phases of a Compiler

Source program
- Lexical analyzer
- Syntax analyzer
- Semantic analyzer
- Intermediate code generator
- Code optimizer
- Code generator
- Target program

Intermediate Code Optimization
- Constant propagation, dead code elimination, common sub-expression elimination, strength reduction, etc.
- Based on dataflow analysis – properties that are independent of execution paths

Native Code Generation
- Intermediate code is translated into native code
- Register allocation, instruction selection

Native Code Optimization
- Peephole optimizations – small window is optimized at a time

Administration
1. HW1 on website: Fun with Lex/Yacc
2. Questions about Projects?
3. Questionnaire Results...

First Step: Lexical Analysis (Tokenizing)
- Breaking the program down into words or "tokens"
- Input: stream of characters
- Output: stream of names, keywords, punctuation marks
- Side effect: Discards white space, comments

Lexical Tokens
- Keywords: if else while break
- Identifiers: x y11 elsex _j00
- Integers: 2 1000 -500 5L
- Floating point: 2.0 0.00020 .02 1.1e5 0.1e-10
- Symbols: + * ( ) ++ < << [ ] >=
- Strings: "x" "He said, "Are you?"
- Comments: /** ignore me **/

Lexical Tokens
float match0(char *s) /* find a zero */
{
    if (!strncmp(s, "0.0", 3))
        return 0.;
}

FLOAT ID(match0) _______ CHAR STAR ID(s)
RPAREN LBRACE IF LPAREN BANG LPAREN ID(s) COMMA STRING(0.0) ______
NUM(3) RPAREN RPAREN RETURN REAL(0.0) ______ RBRACE EOF
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

- 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
- More principled approach: lexer generator that generates efficient tokenizer automatically (e.g., lex, flex)

Issues

- How to describe tokens unambiguously
  - `2.00 2.0e0 2.0e-01 2.0000` 
    - “e0” “e-01” “0.0000”
- How to break text down 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

- Programming language tokens can be described using regular expressions
- A regular expression R describes some set of strings L(R)
- L(R) is the language defined by R
  - L(abc) = {abc}
  - L(hello|goodbye) = {hello, goodbye}
  - L([1-9][0-9]*) = _______________
- Idea: define each kind of token using RE

Regular expressions

**Language** - set of strings

**String** - finite sequence of symbols

**Symbols** - taken from a finite alphabet

Specify languages using regular expressions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>a</th>
<th>one instance of a</th>
</tr>
</thead>
<tbody>
<tr>
<td>Epsilon</td>
<td>ε</td>
<td>empty string</td>
</tr>
<tr>
<td>Alternation</td>
<td>R</td>
<td>string from either L(R) or L(S)</td>
</tr>
<tr>
<td>Concatenation</td>
<td>R.S</td>
<td>string from L(R) followed by L(S)</td>
</tr>
<tr>
<td>Repetition</td>
<td>R*</td>
<td></td>
</tr>
</tbody>
</table>

Convenient Shorthand

- `[abcd]` one of the listed characters (a | b | c | d)
- `[b-g]` `[bcdefg]`
- `[b-gM-Qkr]` `[bcdefgMNOPQkr]`
- `[^ab]` anything but one of the listed chars
- `[^a-f]` anything but the listed range
- `M?` Zero or one M
- `M+` One or more M
- `M*` Zero or more M
- `“a.+*”` literally a.+*
  - Any single character (except \n)
Examples

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

- Lexer generators support abbreviations
- But they cannot be recursive

More Examples

Whitespace:

Integers:

Hex numbers:

Valid UVa User IDs:

Loop keywords in C:

Breaking up Text

elsex=0; 
else \(x\) = 0 ;
else \(x\) = 0 ;

- REs alone not enough: need rules 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 Specification

- 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”)

Lex: A Lexical Analyzer Generator

Lex produces a C program from a lexical specification
http://www.epaperpress.com/lexandyacc/

\%
DIGITS [0-9]+  
ALPHA [A-Za-z]  
CHARACTER {ALPHA}  
IDENTIFIER {ALPHA}({CHARACTER}|{DIGITS})*  
\%
if {return IF; }  
{IDENTIFIER} {return ID; }  
{DIGITS} {return NUM; }  
{[0-9]+."[0-9]*} {return ____; }  
{error(); }