Building scanners with (f|(jf))?lex

Michal Young

April 6, 2006
Basic approach, and complications

- Simple case: Write one pattern for each token type — identifiers, each keyword, each operator, (*, <=, ...), etc. Language conventions mostly make this easy.
- Complications:
  - White space and comments (skipped, or not)
  - Supporting error messages (including in parser)
  - Strings
Lexical conventions & exceptions

Using a scanner generator (with example)
By common convention in modern programming languages,

- white space separates tokens
- white space is not a token
- newline is white space
- any span of white space is equivalent
- no syntactic or semantic information is needed to recognize a token (equivalently: lexical analysis is purely local)
- except constants, tokens carry no semantic or type information
These conventions make lexical analysis simple and distinct from subsequent phases.

They developed over time, and are violated in many early languages.

Some exceptions remain even in current languages.
"for n" is not "f or n"

But in Fortran IV (in the 70s) and Basic (in the 80s), white space was ignored. Famously,

```
DO 100 I = 1, 10
```

is a loop header (like "for (i=1; i < 10; ++i)"") but

```
DO 100 I = 1. 10
```

is an assignment to a variable named DO100I which supposedly led to loss of the Mercury spacecraft
Convention:
White space is not a token

... and any span of white space is equivalent

Usually ... but consider Python, where a change of indentation level is a token (starting a block by increasing indentation, or ending a block by decreasing indentation)
Convention:
Newline is white space

But not in several shell and scripting languages, such as Awk, sh (and its cousins bash, ksh, csh, et al), or Makefiles. (Semicolon avoidance syndrome)
Convention:
Lexical analysis is entirely local

No syntactic or semantic information is needed to recognize a token

Not true in C, where typedef changes the category of a token.
Convention:
Tokens don’t carry values

(Except for literal constants)

But Perl extensively uses lexical markings ($, @, etc.) to mark the data type of a variable.
Convention:
The morals of the story

- Theory and architectural conventions evolved together with language design
- Real lexical analyzers are complicated by deviations from the accepted conventions
Most are descendants of the original `lex` program for parsing in C.

- **flex**: Faster (and more flexible) replacement for `lex`
- **aflex, jflex, ...**: Reimplementations in Ada, Java, ...

One of the first tools that gets built for a new general-purpose programming language
Example:
CSV to XML

- Convert comma-separated values (CSV) to XML
  - CSV is generated by spreadsheets, calendars, ...
- Simple ... except
  - quoted fields (and parts of fields)
  - escaped characters in CSV
  - escaped characters (entities) in XML
Simple Case

Input:

foo, "bar, baz", zot

Output:

<record>
  <field>foo</field>
  <field>bar, baz</field>
  <field>zot</field>
</record>
Can you write the regular expression for a field? (And if you can, can you read it?)

The state machine is actually easier to write and understand!
Rules for State Machine

- \x anywhere is a special character
- quote (") starts and ends quoted material (distinguished as “quoted” and “unquoted” states)
- comma (,) terminates field only in unquoted state
- newline is error within quotes
- accumulate buffer up to end of field
import java_cup.runtime.Symbol;
%

%implements java_cup.runtime.Scanner
... more directives ...

TBL = [tT][aA][bB][lL][eE]
COMMENT = "#".*[\n]
... more pattern macros ...
%

{TBL} { return new Cymbal(tokens.TBL, yychar); }
{COMMENT} { ; /* skip */ }
... more rules ...

Directives and Patterns

... included java code ...

%%% 
%implements java_cup.runtime.Scanner 
%function next_token 
%type java_cup.runtime.Symbol 
%class scanner 
%line  
%char  

TBL = [tT][aA][bB][lL][eE]  
TEXT = ((([^\ ,\{\}\n\t])|(\.))+)  
QUOTED = "([^\"]|(\\\"))*"  
SPACE = [ \n\t]+  
COMMENT = "#".*[\n]  
%%

... rules ...
... included java code ...

%%
... directives and pattern macros ...
%%

{TBL} { return new Cymbal(tokens.TBL, yychar); }

"," { return new Cymbal(tokens.COMMA, yychar); }

"{" { return new Cymbal(tokens.LBRACE, yychar); }

"}" { return new Cymbal(tokens.RBRACE, yychar); }

{TEXT} { return new Cymbal(tokens.TEXT, yychar, yytext()); }

{QUOTED} { String quoted = yytext();
    return new Cymbal(tokens.TEXT, yychar,
                        quoted.substring(1, quoted.length()-1));
}

{SPACE} { ; /* skip */ }

{COMMENT} { ; /* skip */ }

Michal Young
Building scanners with (f|(j)|f)?lex
Using Scanner States
(from a different project)

```java
{  
    StringBuffer string = new StringBuffer();
    ...
}  
%state STRING
%
\"  
  { string.setLength(0);
    yybegin(STRING);  }

<STRING>  {  
  \"
  { yyybegin(YYINITIAL);
    return symbol(sym.WORD,
      string.toString());  }
  [^\n\r"\"]+  
  \t 
  { string.append( yytext() );  }
  \n  { string.append( '\t' );  }
}  ```
terminal TBL ; /* Introduces table construct */
terminal COMMA, LBRACE, RBRACE ;/* Delimit table and individual */
terminal TEXT ; /* Text to put in the table */

goal ::= table {: System.out.println("Successfully parsed"};

table ::= TBL LBRACE rows:tbl RBRACE {: tbl.dump(); :}
 | TBL LBRACE error RBRACE
 | {: report_fatal_error("Couldn’t recover, skipping rest of
 | null);}
 |

...
For the Project ...

- Scanner should return each token of a Cool program
- I have supplied a stub for the parser
  A stand-in for the CUP parser you will build next
- I have supplied a JFlex skeleton that recognizes a single keyword
- Fill in the rest. Avoid changing the stubs.

Available later today from the web site; due Thursday.