Lecture 4: Scanning and parsing
28 Jan 2010

Nate Nystrom
University of Texas at Arlington
http://groups.google.com/group/uta-cse-3302
- I will add you to the group soon

TA
- Derek White – dwhite@uta.edu
- office hours TBA
Syntax defines the structure of a program
  - set of rules defining which symbols are a legally structured program

Semantics defines the “meaning” of a program
  - without semantics, programs are just sequences of characters
Traditional two-pass compiler

- intermediate representation (IR)
- front end maps legal code to IR
- back end maps IR onto target machine
- simplify retargeting
- allow multiple front ends
Traditional interpreter

- intermediate representation (IR)
- front end maps legal code to IR
- evaluator interprets program, producing output
Front end

- Responsibilities:
  - recognize legal code
  - report errors
  - produce IR
  - preliminary storage map
  - shape the code for the back end

- Much of the front end construction can be automated
Scanning

Pre-process the input before parsing

Break stream of bytes into a stream of tokens
- IF, WHILE, INT, STRING, INT

Scanner eliminates whitespace, comments

Simplifies the parser
- deal with ~100 tokens vs. 65536 characters
- smaller state machine
- simpler specification (no need to handle whitespace/comments)
- uses less memory, faster
How to specify tokens?

Use *patterns*

**Whitespace**
- \( WS ::= \ ' ' | \ 't' | WS \ ' ' | WS \ 't' \)

**Comments**
- ‘/’ ‘/’ <any number of characters> ‘\n’

**Keywords**
- “if”, “while”, “do”
More patterns

Identifiers
- `<letter> <optional letters or digits>`

Numbers
- Integer ::= ‘0’ | ( ‘1’ to ‘9’ <optional digits> )
- Float ::= <digits> ‘.’ <optional digits>
  | <optional digits> ‘.’ <digits>
Regular expressions

Can formalize these patterns as a *regular language*
- a (formal) *language* is just a set of strings
- different classes: regular, context-free, context-sensitive, recursively enumerable
- different language classes have different formal properties
  - closure under union, intersection, etc.
- different language classes can be recognized by different classes of machines
  - regular languages recognized by finite automata
  - c.f. languages recognized by pushdown automata (i.e., FA + stack)
  - r.e. languages recognized by Turing machines

Notation for defining a regular language is *regular expression*
## Regular expressions

A regular expression $r$ defines a language $L(r)$ over an alphabet $\Sigma$

<table>
<thead>
<tr>
<th>RE</th>
<th>Description</th>
<th>$L(\text{RE})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>empty string</td>
<td>{\varepsilon}</td>
</tr>
<tr>
<td>$a$</td>
<td>self</td>
<td>{a}</td>
</tr>
<tr>
<td>$r</td>
<td>s$</td>
<td>alternation</td>
</tr>
<tr>
<td>$rs$</td>
<td>concatenation</td>
<td>{ ab \mid a \in L(r), b \in L(s) } = L(r)L(s)</td>
</tr>
<tr>
<td>$r^*$</td>
<td>Kleene closure (zero or more)</td>
<td>$L(r) \cup L(rr) \cup L(rrr) \cup ...$</td>
</tr>
</tbody>
</table>

*the following are non-standard:*

<table>
<thead>
<tr>
<th>RE</th>
<th>Description</th>
<th>$L(\text{RE})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r?$</td>
<td>zero or one</td>
<td>$L(r) \cup {\varepsilon}$</td>
</tr>
<tr>
<td>$r+$</td>
<td>one or more</td>
<td>$L(r^*) - {\varepsilon}$</td>
</tr>
<tr>
<td>$r[i,j]$</td>
<td>repeat from $i$ to $j$</td>
<td>$\bigcup_{k = i..j} L(r^k)$</td>
</tr>
<tr>
<td>. or _</td>
<td>any character</td>
<td>$\Sigma$</td>
</tr>
<tr>
<td>$[x_i-x_j]$</td>
<td>characters in range $x_i$ to $x_j$</td>
<td>{x_i, ..., x_j}</td>
</tr>
</tbody>
</table>
**RE examples**

**Whitespace**
- `( ' ' | \t )*`

**Identifiers**
- `Letter = [a-zA-Z]`
- `Digit = [0-9]`
- `Identifier = Letter ( Letter | Digit )*`

**Numbers**
- `Integer ::= 0 | ( [1-9] Digit* )`
- `Decimal ::= Integer . Digit*`
- `Real = ( Integer | Decimal ) ( e | E ) (+ | -)? Digit+`
- Can get much more complicated:
  - `1. .2 -3.14 1e-6 1.0e12F 0xdeadbeef 0377 65536ULL`
  - `1.toString 0..9`
Recognizing a RE

From an RE, can construct a *finite automaton* or *recognizer*

\[
\text{Identifier} = \text{Letter} \ (\text{Letter} \ | \ \text{Digit})^*
\]

\[
\begin{align*}
0 & \xrightarrow{\text{letter}} 1 \\
0 & \xrightarrow{\text{digit}} 3 \\
1 & \xrightarrow{\text{letter}} 1 \\
1 & \xrightarrow{\text{digit}} 2 \\
1 & \xrightarrow{\text{other}} 2 \\
2 & \xrightarrow{\text{other}} 2 \\
3 & \xrightarrow{\text{other}} 3 \\
& \xrightarrow{\text{error}} 3
\end{align*}
\]
### Representation of recognizer

Two tables:

<table>
<thead>
<tr>
<th>char_class</th>
<th>( a - z )</th>
<th>( A - Z )</th>
<th>( 0 - 9 )</th>
<th>other</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>letter</td>
<td>letter</td>
<td>digit</td>
<td>other</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>letter</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>digit</td>
<td>3</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>other</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Diagram:

- States: 0, 1, 2, 3
- Transitions:
  - From 0 to 1 on letter
  - From 0 to 2 on other
  - From 0 to 3 on digit
  - From 2 to 3 on other
- Accept state: 2
- Error state: 3

Two tables:

- char_class:
  - \( a - z \): letter
  - \( A - Z \): letter
  - \( 0 - 9 \): digit
  - other
- next_state:
  - letter: 1
  - digit: 3
  - other: 3

To change languages, we can just change the tables.
Code for recognizer

```c
char ← next_char();
state ← 0;    /* code for state 0 */
done ← false;
token_value ← "" /* empty string */
while( not done ) {
    class ← char_class[char];
    state ← next_state[class,state];
    switch(state) {
        case 1:    /* building an id */
            token_value ← token_value + char;
            char ← next_char();
            break;
        case 2:    /* accept state */
            token_type = identifier;
            done = true;
            break;
        case 3:    /* error */
            token_type = error;
            done = true;
            break;
    }
}
return token_type;
```

To change languages, just change the tables
A nondeterministic finite automaton consists of:

- a finite set of states
- a start state
- a set of final (or accepting) states
- an alphabet – set of input symbols
- a transition relation that maps a state to the next state given an input symbol or the empty string (ε)
RE to NFA

\[ N(\varepsilon) \]

\[ N(a) \]

\[ N(A) \]

\[ N(A|B) \]

\[ N(AB) \]

\[ N(A^*) \]
RE to NFA example

\[(a | b)^*abb\]
NFA example

\[(a \mid b)^*abb\]

Note, \(s_0\) has multiple transitions on \(a\)
This is what makes the FA nondeterministic
A deterministic FA is an NFA with at most one transition for each state and symbol, and no transitions on ε.
Subset construction

From any NFA, can construct an equivalent DFA

\[
\begin{array}{c|c|c}
    & a & b \\
\hline
\{s_0\} & \{s_0, s_1\} & \{s_0\} \\
\{s_0, s_1\} & \{s_0, s_1\} & \{s_0, s_2\} \\
\{s_0, s_2\} & \{s_0, s_1\} & \{s_0, s_3\} \\
\{s_0, s_3\} & \{s_0, s_1\} & \{s_0\} \\
\end{array}
\]
Subset construction

Idea is to map create a state in the DFA for each reachable set of states in the NFA

Algorithm sketched in the book
Driving the scanner

Can build a single DFA for all input tokens
Wrap in an interface like this:

```
interface Scanner {
    Token nextToken();
}
```

Parser works by asking for tokens until end-of-file, building up a parse tree as it goes.
Limitations

Do we need the parser at all?

Yes, because regular languages have some limitations

- regular languages can’t “count”
  - \( \{a^n b^n\} \) is not regular
- regular languages can’t do recursion
  - balanced parens are not regular: \( \text{Exp} \Rightarrow^* (\text{Exp})' \)