Compiler Construction 2016/2017
Lexical Analysis

Peter Thiemann

November 2, 2016

Outline

Lexical Analysis

```
source --> [scanner] --> tokens
```

Scanner:

- partitions input into lexemes — the basic unit of syntax
- maps lexemes into tokens
- `x = x + 1;` becomes

  `<id, x> <sym,=> <id, x> <sym,+> <num, 1> <sym,;>`
- typical tokens: number, id, +, -, *, /, do, end
- eliminates white space (tabs, blanks, comments)
- a key issue is speed

Specification of a scanner

- lexemes
- tokens
- mapping from lexemes to tokens

- lexemes should be recognized efficiently

$\Rightarrow$ specify lexemes using regular expressions

$\Rightarrow$ compile regular expressions to deterministic finite automata

$\Rightarrow$ recognize lexemes in linear time (i.e., as fast as possible)

Regular expressions

Let Σ be a fixed alphabet (in practice Unicode).
Define the set of regular expressions (over Σ).

1. ε is a regular expression.
2. a is a regular expression, if $a \in \Sigma$.
3. If r and s are regular expressions, then
    - $(r|s)$ is a regular expression (alternation).
    - (rs) is a regular expression (concatenation).
    - $(r*)$ is a regular expression (closure).

If we adopt a precedence for operators, the extra parentheses can go away. We assume closure, then concatenation, then alternation as the order of precedence.

Language recognized by RE/Step 1

We write $N(r)$ if a RE r recognizes the empty word.

$$
\begin{aligned}
N(\varepsilon) &= \mathit{true} \\
N(a) &= \mathit{false} \\
N(r|s) &= N(r) \vee N(s) \\
N(rs) &= N(r) \wedge N(s) \\
N(r*) &= \mathit{true}
\end{aligned}
$$

Language recognized by RE/Step 2

For $a \in \Sigma$, RE r recognizes the word aw if there is an RE in $\partial_a(r)$ that recognizes word w.

$$
\begin{aligned}
\partial_a(\varepsilon) &= \emptyset \\
\partial_a(a) &= \{\varepsilon\} \\
\partial_a(b) &= \emptyset \qquad a \neq b \in \Sigma \\
\partial_a(r|s) &= \partial_a(r) \cup \partial_a(s) \\
\partial_a(rs) &= \partial_a(r) \cdot s \cup (\mathit{if}\ N(r)\ \mathit{then}\ \partial_a(s)\ \mathit{else}\ \emptyset) \\
\partial_a(r*) &= \partial_a(r) \cdot (r*)
\end{aligned}
$$

Construction of DFA

- ∂_a is transition function of a NFA
- the powerset construction yields a DFA for r
- set of states Q
  - $\{r\} \in Q$
  - for all $q \in Q$, $s \in q$, and $a \in \Sigma$: $\bigcup\{\partial_a(s) \mid s \in q\} \in Q$
- $\delta(q, a) = \bigcup\{\partial_a(s) \mid s \in q\}$
- initial state $\{r\}$

Example: Numbers

```
0|(1|2)(0|1|2)*
-0->    eps
-1,2-> (0|1|2)*
         -0,1,2-> (0|1|2)*
```

$$Q = \{\texttt{0|(1|2)(0|1|2)*}, \texttt{eps}, \texttt{(0|1|2)*}, \emptyset\}$$

Language recognized by RE/Summary

- Step 1 and 2 are easy to implement
- Optimized version of this approach is used in professional regexp matchers
- Is equivalent to a nondeterministic finite automaton
- Can be compiled to a deterministic automaton that runs in linear time (this is done by scanner generators like lex)
- Generators offer further extensions of RE for convenience: character classes, repetitions $r\{m, n\}$, context r/s

Examples

White space

```
[ \t][ \t]*
```

Keywords and operators

```
if
then
*
```

Comments (approximate)

```
/\*[^*]*\*/
```

Examples/2

Identifiers

```
[a-zA-Z][a-zA-Z0-9_]*
```

Numbers

```
0|[1-9][0-9]*
(0|[1-9][0-9]*)?.[0-9]*
```

Disambiguation and the longest match

- A scanner tries to match all specified lexeme kinds at once

$\Rightarrow$ it run several automata in parallel

- Problem: ambiguous matching
  - Keyword: `do`
  - Identifier: `door`
- Approach: Principle of the longest match
  choose the longest input accepted by one of the automata
- In this example: return `<id,door>`

Scanner implementation

- Suppose there are $n \geq 1$ token classes.
- Class i is recognized by a DFA with states Q^i, initial state q_0^i, transition function δ^i, and accepting states F^i.
- The state of the scanner is a vector $\vec{q} \in Q^1 \times \cdots \times Q^n$
- Input is available in array *in* from position p

Scanner implementation

$lc \leftarrow 0$	last accepted class: none
$lp \leftarrow p$	position after last lexeme
$\vec{q} \leftarrow \vec{q_0}$	initial state
$while(true)$	
$a \leftarrow in[p++]$	get character and advance
$\vec{q} \leftarrow \vec{\delta}(\vec{q}, a)$	apply all transitions in parallel
$c \leftarrow \min\{i \mid q^i \in F^i\}$	find matches
$if\ c > 0$	if there is a match ...
$then\ lc \leftarrow c;\ lp \leftarrow p$	save class and position
$else\ if\ \vec{q}$ is a sink state	
$then\ p \leftarrow lp;\ return\ lc$	

Optimization

- All characters in a character class behave the same

⇒ Map character to its class before applying the transition

⇒ Table `char_class`

	a-z	A-Z	0-9	other
value	letter	letter	digit	other

- Transition maps state and character class to next state

⇒ Table `next_state`

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

- Table `final_state`
- Change table ⇔ change language

Language features that can cause problems

PL/I has no reserved words

```
if then then then = else; else else = then;
```

FORTRAN and Algol68 ignore blanks

```
do 10 i = 1,25
do 10 i = 1.25
```

String constants

special characters in strings

Finite closures

- some languages limit identifier lengths
- adds states to count length
- FORTRAN 66: 6 characters