Lexical Analysis

Introduction

Copyright 2016, Pedro C. Diniz, all rights reserved.

Students enrolled in the Compilers class at the University of Southern California have explicit permission to make copies of these materials for their personal use.
The purpose of the front end is to deal with the input language

- Perform a membership test: code \( \in \) source language?
- Is the program well-formed (semantically)?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic
Scanner

- Maps stream of characters into words
  - Basic unit of syntax
  - $x = x + y$ becomes $<\text{id},x> <\text{eq},=> <\text{id},x> <\text{pl},-> <\text{id},y> <\text{sc},;>$
- Characters that form a word are its *lexeme*
- Its *part of speech* (or *syntactic category*) is called its *token type*
- Scanner discards white space & (often) comments

Speed is an issue in scanning
⇒ use a specialized recognizer
The Front End

Parser
- Checks stream of classified words (parts of speech) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We’ll come back to parsing in a couple of lectures
The Big Picture

- Language syntax is specified with *parts of speech*, not *words*
- Syntax checking matches *parts of speech* against a grammar

1. \( \text{goal} \rightarrow \text{expr} \)
2. \( \text{expr} \rightarrow \text{expr} \text{ op } \text{term} \)
3. \( \mid \text{term} \)
4. \( \text{term} \rightarrow \text{number} \)
5. \( \mid \text{id} \)
6. \( \text{op} \rightarrow + \)
7. \( \mid - \)

\[ S = \text{goal} \]
\[ T = \{ \text{number}, \text{id}, +, - \} \]
\[ N = \{ \text{goal}, \text{expr}, \text{term}, \text{op} \} \]
\[ P = \{ 1, 2, 3, 4, 5, 6, 7 \} \]
The Big Picture

• Language syntax is specified with parts of speech, not words

• Syntax checking matches parts of speech against a grammar

1. \textit{goal} \rightarrow \textit{expr}
2. \textit{expr} \rightarrow \textit{expr} \textit{op} \textit{term}
3. \text{ | } \textit{term}
4. \textit{term} \rightarrow \textit{number}
5. \text{ | } \textit{id}
6. \textit{op} \rightarrow +
7. \text{ | } -

\begin{align*}
S &= \textit{goal} \\
T &= \{ \text{number, id, +, -} \} \\
N &= \{ \textit{goal, expr, term, op} \} \\
P &= \{ 1, 2, 3, 4, 5, 6, 7 \}
\end{align*}
The Big Picture

Why study Lexical Analysis?
• We want to avoid writing Scanners by hand
• We want to harness the theory classes

Goals:
– To simplify specification & implementation of scanners
– To understand the underlying techniques and technologies
What is a Lexical Analyzer?

Source program text  →  Tokens

• Example of Tokens
  • Operators = + - > ( { := == <>
  • Keywords if while for int double
  • Numeric literals 43 4.565 -3.6e10 0x13F3A
  • Character literals ‘a’ ‘~’ ‘\’
  • String literals "4.565" "Fall 10" "\"" = empty"

• Example of non-tokens
  • White space space(‘ ‘) tab(‘t’) end-of-line(‘\n’)
  • Comments /*this is not a token*/
Lexical Analyzer in Action

\[ \text{for } \text{var1} = 10 \text{ var1} \leq \]
Lexical Analyzer in Action

```
for var1 = 10 var1 <=
```
Lexical Analyzer in Action

\[ \text{for var l = 10 var l <= } \]
Lexical Analyzer in Action

\[
\text{for } \quad \text{var 1} = 10 \quad \text{var 1} \leq
\]

for_key
Lexical Analyzer in Action

for var l = 10 var l <=

for_key
Lexical Analyzer in Action

\[
\text{for} \quad \text{var} \quad 1 \quad = \quad 10 \quad \text{var} \quad 1 \quad \leq
\]

for_key
Lexical Analyzer in Action

for var l = 10 var l <=

for_key
Lexical Analyzer in Action

```
for var1 = 10 var1 <= 
```

for_key
Lexical Analyzer in Action

```
for var l = 10 var l <=
```

for_key
Lexical Analyzer in Action

```
for var l = 10 var l <= 
```

for_key ID("var1")
Lexical Analyzer in Action

for var1 = 1 0 var1 <=

for_key ID(“var1”)

for var1 = 1 0 var1 <=
Lexical Analyzer in Action

\[
\text{for}\ \text{var1} = 10 \text{ var1} <= \text{var1}
\]

for_key  ID("var1")
Lexical Analyzer in Action

\[
\text{for \ var\ l \ = \ 10 \ var\ l \ < \ = }
\]

for_key \ ID(“var1”) \ eq\_op
Lexical Analyzer in Action

\[
\text{for} \quad \text{var1} \quad = \quad 1 \quad 0 \quad \text{var1} \quad \leq \quad \text{var1}
\]

for_key \quad \text{ID("var1")} \quad \text{eq_op}
Lexical Analyzer in Action

for var1 = 10 var1 <=

for_key ID("var1") eq_op
Lexical Analyzer in Action

\[ \text{for} \quad \text{var1} = 10 \quad \text{var1} \leq \]

for_key \quad ID(“var1”) \quad eq_op
Lexical Analyzer in Action

\[ \text{for} \var_1 = 10 \quad \var_1 \leq \]

for_key ID(“var1”) eq_op Num(10)
Lexical Analyzer in Action

for var1 = 10 var1 <=

for_key ID("var1") eq_op Num(10)
Lexical Analyzer in Action

\[
\text{
\begin{array}{c}
\text{for \ var1 = 10 \ var1 <= }
\end{array}
\]

for_key \ ID(“var1”) \ eq_op \ Num(10)
Lexical Analyzer in Action

for var1 = 10 var1 <=

for_key ID("var1") eq_op Num(10)
Lexical Analyzer in Action

```
for var1 = 10 var1 < =
```

```
for_key ID("var1") eq_op Num(10)
```
Lexical Analyzer in Action

\[
\text{for } \text{var} 1 = 10 \text{ var} 1 \leq \text{...}
\]

for_key \ ID(“var1”) eq_op Num(10)
Lexical Analyzer in Action

\[
\text{for} \quad \text{var1} = 10 \quad \text{var1} \leq \text{for_key} \quad \text{ID(“var1”)} \quad \text{eq_op} \quad \text{Num(10)} \quad \text{ID(“var1”)}
\]
Lexical Analyzer in Action

```
for var1 = 10 var1 <=
```

for_key ID("var1") eq_op Num(10) ID("var1")
Lexical Analyzer in Action

```plaintext
for var1 = 10 var1 <=
```

```plaintext
for_key ID("var1") eq_op Num(10) ID("var1")
```
Lexical Analyzer in Action

```
for var1 = 10 var1 <=
```

```
for_key ID("var1") eq_op Num(10) ID("var1") le_op
```
Lexical Analyzer in Action

```
for var1 = 10 var1 <=
```

```
for_key ID("var1") eq_op Num(10) ID("var1") le_op
```
Lexical Analyzer needs to...

- Partition Input Program Text into Subsequence of Characters Corresponding to Tokens
- Attach the Corresponding Attributes to the Tokens
- Eliminate White Space and Comments
Lexical Analysis: Basic Issues

• How to Precisely Match Strings to Tokens

• How to Implement a Lexical Analyzer
Regular Expressions

Lexical patterns form a *regular language*

*** any finite language is regular  ***

Regular expressions (REs) describe regular languages

Regular Expression (over alphabet $\Sigma$)

- $\varepsilon$ is a RE denoting the set $\{\varepsilon\}$
- If $a$ is in $\Sigma$, then $a$ is a RE denoting $\{a\}$
- If $x$ and $y$ are REs denoting $L(x)$ and $L(y)$ then
  - $x \mid y$ is an RE denoting $L(x) \cup L(y)$
  - $xy$ is an RE denoting $L(x)L(y)$
  - $x^*$ is an RE denoting $L(x)^*$

**Precedence:**
closure, then concatenation, then alternation
## Set Operations (review)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Union of L and M</strong></td>
<td>( L \cup M = { s \mid s \in L \text{ or } s \in M } )</td>
</tr>
<tr>
<td><strong>Written L \cup M</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Concatenation of L and M</strong></td>
<td>( LM = { st \mid s \in L \text{ and } t \in M } )</td>
</tr>
<tr>
<td><strong>Written L \cdot M</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Kleene closure of L</strong></td>
<td>( L^* = \bigcup_{0 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td><strong>Written L^</strong>*</td>
<td></td>
</tr>
<tr>
<td><strong>Positive Closure of L</strong></td>
<td>( L^+ = \bigcup_{1 \leq i \leq \infty} L^i )</td>
</tr>
<tr>
<td><strong>Written L^+</strong></td>
<td></td>
</tr>
</tbody>
</table>

These definitions should be well known.
Examples of Regular Expressions

Identifiers:

- `Letter` → `(a|b|c|…|z|A|B|C|…|Z)`
- `Digit` → `(0|1|2|…|9)`
- `Identifier` → `Letter ( Letter | Digit )*`

Numbers:

- `Integer` → `(+|-|ε) (0|1|2|3|…|9)(Digit*)`
- `Decimal` → `Integer * Digit *`
- `Real` → `( Integer | Decimal ) E (±|-|ε) Digit *`
- `Complex` → `( Real , Real )`

Numbers can get much more complicated!
Regular Expressions (the point)

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer

- Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions

- Some of you may have seen this construction for string pattern matching

⇒ We study REs and associated theory to automate scanner construction!
Consider the problem of recognizing Register names

\[ \text{Register} \rightarrow r \ (0 | 1 | 2 \ | \ldots | 9) \ (0 | 1 | 2 \ | \ldots | 9)^* \]

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

**Example**

**Recognizer for Register**

Transitions on other inputs go to an error state, \( s_e \)
Example (continued)

DFA operation

- Start in state \( S_0 \) & take transitions on each input character
- DFA accepts a word \( x \) iff \( x \) leaves it in a final state \( (S_2) \)

So,

- \( r17 \) takes it through \( s_0, s_1, s_2 \) and accepts
- \( r \) takes it through \( s_0, s_1 \) and fails
- \( a \) takes it straight to \( s_e \)
Example (continued)

To be useful, recognizer must turn into code

\[
\begin{align*}
\text{Char} & \leftarrow \text{next character} \\
\text{State} & \leftarrow s_0 \\
\text{while (Char} & \neq \text{ EOF)} \\
\text{State} & \leftarrow \delta(\text{State,Char}) \\
\text{Char} & \leftarrow \text{next character} \\
\text{if (State is a final state )} \\
\text{then report success} \\
\text{else report failure}
\end{align*}
\]

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>r</th>
<th>$0,1,2,3,4,5$</th>
<th>$6,7,8,9$</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
<td></td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td></td>
</tr>
</tbody>
</table>

Skeleton recognizer

Table encoding RE
Example (continued)

To be useful, recognizer must turn into code

Char ← next character
State ← s₀
while (Char ≠ EOF)
    State ← δ(State, Char)
    perform specified action
    Char ← next character
if (State is a final state )
    then report success
else  report failure

δ | r     | 0,1,2,3,4,5, 6,7,8,9 | All others |
---|------|----------------------|------------|
₀  | s₁   | sₑ                  | sₑ         |
    | start| error               | error      |
₁  | sₑ   | s₂                  | sₑ         |
    | error| add                 | error      |
₂  | sₑ   | s₂                  | sₑ         |
    | error| add                 | error      |
ₑ  | sₑ   | sₑ                  | sₑ         |
    | error| error               | error      |

Skeleton recognizer

Table encoding RE
What about a Tighter Specification?

$ r \ Digit \ Digit^{*} $ allows arbitrary numbers

- Accepts $ r00000 $
- Accepts $ r99999 $
- What if we want to limit it to $ r0 $ through $ r31 $?

Write a tighter regular expression

- $ Register \rightarrow r \ ( \ (0|1|2) \ (Digit \ | \ \epsilon) \ | \ (4|5|6|7|8|9) \ | \ (3|30|31) )$
- $ Register \rightarrow r0 | r1 | r2 | ... | r31 | r00 | r01 | r02 | ... | r09$

Produces a more complex DFA

- Has more states
- Same cost per transition
- Same basic implementation
Tighter Register Specification (cont’ d)

The DFA for

\[ \text{Register} \rightarrow r \left( (0|1|2) \ (\text{Digit} \mid \epsilon) \mid (4|5|6|7|8|9) \mid (3|30|31) \right) \]

- Accepts a more constrained set of registers
- Same set of actions, more states
### Tighter Register Specification (cont’d)

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_4$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_5$</td>
<td>$s_e$</td>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

---

Runs in the same skeleton recognizer
Summary

• The Role of the Lexical Analyzer
  – Partition input stream into tokens

• Regular Expressions
  – Used to describe the structure of tokens

• DFA: Deterministic Finite Automata
  – Machinery to recognize Regular Languages