Lexical Analysis

Introduction
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
The Front End

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. $goal \rightarrow expr$
2. $expr \rightarrow expr \ op \ term$
3. $\mid \ term$
4. $term \rightarrow number$
5. $\mid \ id$
6. $op \rightarrow +$
7. $\mid -$  

$S = goal$

$T = \{ \text{number, id, +, -} \}$

$N = \{ goal, expr, term, op \}$

$P = \{ 1, 2, 3, 4, 5, 6, 7 \}$
The Big Picture

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

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

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

---

No words here!

Parts of speech, not words!
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

Specifications written as “regular expressions”
What is a Lexical Analyzer?

Source program text    Tokens

• Example of Tokens
  • Operators
  • Keywords
  • Numeric literals
  • Character literals
  • String literals

• Example of non-tokens
  • White space
  • Comments

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

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

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

for var1 = 10 var1 <=
Lexical Analyzer in Action

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

for_key
Lexical Analyzer in Action

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

for_key
Lexical Analyzer in Action

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

for_key
Lexical Analyzer in Action

for var1 = 10 var1 <= 

for_key
Lexical Analyzer in Action

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

for_key
Lexical Analyzer in Action

for var1 = 10 var1 <=

for_key
Lexical Analyzer in Action

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

for_key \quad \text{ID(“var1”)}
Lexical Analyzer in Action

for\_var\_1 = 1 0 \ var\_1 < =

for\_key ID(“var1”)
Lexical Analyzer in Action

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

for_key ID(“var1”)
Lexical Analyzer in Action

for var1 = 10 var1 <=

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

```
for var1 = 1 0 var1 <=
```

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

for var1 = 10 var1 <=

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

`for var1 = 10 var1 <=`  

for_key      ID("var1")  eq_op
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

```plaintext
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

for var1 = 10 var1 <=

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

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

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

for var1 = 10 var1 <=

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

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

```
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")
```
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
  - \( x | y \) 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 LM</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:

Let the alphabet be defined as:

$\text{Letter} \to \{a, b, c, \ldots, z, A, B, C, \ldots, Z\}$

$\text{Digit} \to \{0, 1, 2, \ldots, 9\}$

Then an identifier can be defined as:

$\text{Identifier} \to \text{Letter} (\text{Letter} \mid \text{Digit})^*$

Numbers:

Integers are defined as:

$\text{Integer} \to (\pm | \epsilon) (0 | (1|2|3|\ldots|9)(\text{Digit}^*))$

Decimals are defined as:

$\text{Decimal} \to \text{Integer} \cdot \text{Digit}^*$

Real numbers are defined as:

$\text{Real} \to (\text{Integer} \mid \text{Decimal}) \cdot (\pm | \epsilon) \text{Digit}^*$

Complex numbers are defined as:

$\text{Complex} \to (\text{Real} \cdot \text{Real})$

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!
Example

Consider the problem of recognizing Register names

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

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

RE corresponds to a recognizer (or DFA)

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 ($S_2$)

\[
\begin{array}{c}
S_0 \xrightarrow{r} S_1 \xrightarrow{(0|1|2|\ldots|9)} S_2
\end{array}
\]

Recognizer for Register

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

Char ← next character
State ← s₀

while (Char ≠ EOF)
    State ← δ(State, Char)
    Char ← next character

if (State is a final state)
    then report success
else report failure

<table>
<thead>
<tr>
<th>δ</th>
<th>0,1,2,3,4,5</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>s₂</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</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 specific action
Char ← next character

if (State is a final state)
    then report success
else report failure

Skeleton recognizer

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0,1,2,3,4,5, 6,7,8,9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₀</td>
<td>s₁</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>start</td>
<td>error</td>
<td>error</td>
</tr>
<tr>
<td>s₁</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>s₂</td>
<td>sₑ</td>
<td>s₂</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>add</td>
<td>error</td>
</tr>
<tr>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
<td>sₑ</td>
</tr>
<tr>
<td></td>
<td>error</td>
<td>error</td>
<td>error</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{Register} & \rightarrow r \ ( (0|1|2) (Digit \mid \varepsilon) \mid (4|5|6|7|8|9) \mid (3|30|31) ) \\
\text{Register} & \rightarrow r0 \mid r1 \mid r2 \mid \ldots \mid r31 \mid r00 \mid r01 \mid r02 \mid \ldots \mid r09
\end{align*}
\]

Produces a more complex DFA

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

The DFA for

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

- 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