LEXICAL ANALYSIS

Baishakhi Ray

These slides are motivated from Prof. Alex Aiken: Compilers (Stanford)
Structure of a Typical Compiler

Analysis Phase
- Character stream
- Lexical Analysis
  - Token stream
  - Syntactic Analysis
  - Syntax trees
  - Semantic Analysis
  - Syntax trees
- Interpreter

Synthesis Phase
- Intermediate Code Generation
  - IR
  - Optimization
  - IR
  - Code Generation
  - Target Language
Input to Compiler

Character stream

Lexical Analysis
Token stream
Syntactic Analysis
Syntax trees
Semantic Analysis
Syntax trees
Interpreter

Intermediate Code Generation

optimization
Intermediate Representation (IR)

Code Generation
Target Language

/ * s i m p l e  e x a m p l e * /
if ( i == j )
	z = 0;
else
	z = 1;

/ * s i m p l e  e x a m p l e * /
if ( i == j )
	z = 0;
else
	z = 1;
Lexical Analysis

1. Remove comments

```plaintext
/* simple example */
if ( i == j )
  z = 0 ; 
else
  z = 1 ;
```

2.1. Identify substrings

```plaintext
'if' '(' 'i' '==' 'j' ')' 'n' 't' 'z' '=' '0' ';' 'n' 'else' 'n' 't' 'z' '=' '1' ';
```

2.2. Identify token classes

```plaintext
keyword<if> LPAR identifier<i> op<==> identifier<j> RPAR 
whitespaces identifier<z> op==> number<0> <;> whitespaces 
keyword<else> identifier<z> op==> number<1> ';
```
Token Class

• keywords, identifiers, LPAR, RPAR, number, etc.
Token Class

- Each class corresponds to a set of strings

- Identifier
  - Strings are letters or digits, starting with a letter
  - Eg:

- Numbers:
  - A non-empty strings of digits
  - Eg:

- Keywords
  - A fixed set of reserved words
  - Eg:

- Whitespace
  - A non-empty sequence of blanks, newlines, and tabs
Lexical Analysis (Example)

- Classify program substrings according to roles (token class)
- Communicate tokens to parser

\[
Z = 1
\]

- \(<\text{Id}, \text{“Z”}\)>
- \(<\text{Op}, \text{“=”}\)>
- \(<\text{Numbers}, \text{“1”}\)>

“Z”, “=” “1” are called lexemes (an instance of the corr. token class)
Lexical Analysis: HTML Examples

Here is a photo of <b> my house </b>

<text, "Here is a photo of">
<nodestart, b>
<text, "my house">
<nodeend, b>
Exercise

```c
x = p;
while ( x < 100 ) { x++ ; }
```
Exercise

```c
if(i == j)
z = 0;
else
z = 1;
```

==/=/?

Keyword/Identifier?
Lookahead

- Lexical analysis tries to partition the input string into the logical units of the language. This is implemented by reading left to right. “scanning”, recognizing one token at a time.

- “Lookahead” is required to decide where one token ends and the next token begins.

```java
if(i == j) {
    z = 0;
} else {
    z = 1;
}
```

=//=?

Keyword/Identifier?
Lookahead: Examples

- Usually, given the pattern describing the lexemes of a token, it is relatively simple to recognize matching lexemes when they occur on the input.

- However, in some languages, it is not immediately apparent when we have seen an instance of a lexeme corresponding to a token.

  **FORTRAN RULE:** White Space is insignificant: `VAR1 == VAR1`
  
  ```
  DO 5 I = 1,25
  DO 5 I = 1.25
  ```

- Lexical analysis may require to “look ahead” to resolve ambiguity.
  - Look ahead complicates the design of lexical analysis
  - Minimize the amount of look ahead
Lexical Analysis: Examples

- C++ template Syntax:
  - Foo<Bar>

- C++ stream Syntax:
  - cin >> var

- Ambiguity
  - Foo<Bar<Bar>>
  - cin >> var
Lexical Errors

- A lexical error is any input that can be rejected by the lexer.

- When a token cannot be recognized by the rules defined token class
  - Example: '@' is rejected as a lexical error for identifiers in Java (it's reserved).

Recovery

- Panic Mode: delete successive characters until a valid token is found
- Delete one character from remaining inputs
- Insert one character in the remaining input
- Replace / transpose
Lexical Errors

- Is fi lexical error?
  - It can be a function identifier
  - It is quite difficult for a lexical analyzer to decide whether fi is an error without further information

\[
\text{fi}(a==f(x))
\]
Summary So Far

- The goal of Lexical Analysis
  - Partition the input string to lexeme
  - Identify the token class of each lexeme

- Left-to-right scan => look ahead may require
  - In reality, lookahead is always needed
  - Our goal is to minimize the amount of lookahead
Recognizing Lexemes:
a simple character by character formulation

Recognize word \texttt{while}

\begin{verbatim}
c=NextChar();
if(c!='w') { /*do something*/}
else {
    c=NextChar();
    if(c!='h') { /*do something*/}
    else {
        c=NextChar();
        if(c!='i') { /*do something*/}
        else {
            c=NextChar();
            if(c!='l') { /*do something*/}
            else{
                c=NextChar();
                if(c!='e') { /*do something*/}
                else{
                    /*report success*/
                }}
            }
        }
    }
}
\end{verbatim}

S0 \rightarrow S1 \rightarrow S2 \rightarrow S3 \rightarrow S4 \rightarrow S5

$S_i$ s are all abstract states of computation
Recognizing Lexemes

- \( x = 1 \)
A Formalism of Recognizer

- A finite automaton consists of
  - An input Alphabet: $\Sigma$
  - A finite set of states: $S$
  - A start state:
  - A set of accepting states: $F \subseteq S$
  - A set of transitions state: $state_1 \xrightarrow{a} state_2$
A Formalism of Recognizer

- A finite automaton consists of
  - An input Alphabet: $\Sigma$
  - A finite set of states: $S$
  - A start state: $S_0$
  - A set of accepting states: $F \subseteq S$
  - A set of transitions state $\delta$: state1 $\xrightarrow{input} $ state2

$S = \{S_0, S_1, S_2, S_3\}$

$\Sigma = \{x, =, 1\}$

$\delta = \{S_0 \xrightarrow{x} S_1, S_0 \xrightarrow{=} S_2, S_0 \xrightarrow{1} S_3\}$

$S_0 = S_0$

$F = \{S_1, S_2, S_3\}$
A simple parser for x=1

c=NextChar();
state=S_0
while(c!='eof' and state!=S_{err}) {
    state=\delta(state,c)
    c=NextChar();
}

if(state \in F)
    /* report acceptance */
else
    /* report failure */
Example: Lexeme Recognition

- Show simple state transition of: \( e = m \times c^{**2} \)

\[
S = \{S_0, S_1, S_2, S_3, S_4, S_5, S_6, S_7\}
\]

\[
\Sigma = \{e, m, c, *, **, 2, =\}
\]

\[
\delta = \{S_0 \xrightarrow{e} S_1, S_0 \xrightarrow{=} S_2, S_0 \xrightarrow{m} S_3, S_0 \xrightarrow{*} S_4, S_4 \xrightarrow{*} S_5, S_0 \xrightarrow{c} S_6, S_0 \xrightarrow{2} S_7\}
\]

\[
S_0 = S_0
\]

\[
F = \{S_1, S_2, S_3, S_4, S_5, S_6, S_7\}
\]
Input Buffering

- \( e = m \times c \times 2m \)
Can we run out of buffer space?
Recognizing Token Class

How to describe the string patterns?
- i.e., which set of strings belongs to which token class?
- Use regular languages

Use Regular Expressions to define Regular Languages.
REGULAR LANGUAGES
Regular Expressions

- **Single character**
  - `'c' = {“c”}`

- **Epsilon**
  - `$\varepsilon = \{\varepsilon\}$`

- **Union**
  - $A + B = \{a | a \in A\} \cup \{b | b \in B\}$

- **Concatenation**
  - $AB = \{ab | a \in A \land b \in B\}$

- **Iteration (Kleene closure)**
  - $A^* = \bigcup_{i \geq 0} A^i = A \cdots A$ (i times)
  - $A^+ = \varepsilon$ (empty string)
Regular Expressions

- Def: The regular expressions over $\Sigma$ are the smallest set of expressions including

  $$R = \varepsilon$$

  1. ‘c’, ‘c’ $\in \Sigma$

  2. $R + R$

  3. $RR$

  4. $R^*$
Regular Expression Example

- \( \Sigma = \{p, q\} \)
  - \( q^* \)
  - \( (p+q)q \)
  - \( p^*+q^* \)
  - \( (p+q)^* \)

- There can be many ways to write an expression
Exercise

Choose the regular languages that are equivalent to the given regular language: \((p + q)^*q(p + q)^*\)

A. \((pq + qq)^*(p + q)^*\)
B. \((p + q)^*(qp + qq + q)(p + q)^*\)
C. \((q + p)^*q(q + p)^*\)
D. \((p + q)^*(p + q)(p + q)^*\)
Formal Languages

- Def: Let $\Sigma$ be a set of character (alphabet). A language over $\Sigma$ is a set of strings of characters drawn from $\Sigma$.
  - Regular languages is a formal language
- Alphabet = English character, Language = English Language
  - Is it formal language?
- Alphabet = ASCII, Language = C Language
Formal Language

'c' = \{"c"\}
\varepsilon = \{"
\}
A + B = \{a \mid a \in A\} \cup \{b \mid b \in B\}
AB = \{ab \mid a \in A \mbox{ and } b \in B\}
A^* = \bigcup_{i \geq 0} A^i
Formal Language

\[ L('c') = \{"c"\} \]
\[ L(\varepsilon) = \{\"\varepsilon\"\} \]

\[ L(A + B) = \{a \mid a \in L(A) \} \cup \{b \mid b \in L(B)\} \]

\[ L(AB) = \{ab \mid a \in L(A) \land b \in L(B)\} \]

\[ L(A^*) = \bigcup_{i \geq 0} L(A^i) \]

- \( L \): Expressions -> Set of strings
  - Meaning function \( L \) maps syntax to semantics
  - Mapping is many to one
  - Never one to many
Lexical Specifications

- **Keywords**: “if” or “else” or “then” or “for” ….
  - Regular expression = ‘i’ ‘f’ + ‘e’ ‘l’ ‘s’ ‘e’
    = ‘if’ + ‘else’ + ‘then’

- **Numbers**: a non-empty string of digits
  - digit = ‘1’+’0’+’2’+’3’+’4’+’5’+’6’+’7’+’8’+’9’
  - digit*
  - How to enforce non-empty string?
    - digit digit* = digit+
Lexical Specifications

- **Identifier**: strings of letters or digits, starting with a letter
    - = [a-zA-Z]
  - letter (letter + digit)*

- **Whitespace**: a non-empty sequence of blanks, newline, and tabs
  - (‘ ’ + ‘\n’ + ‘\t’)+
PASCAL Lexical Specification

- digit = ‘0’+‘1’+‘2’+‘3’+‘4’+‘5’+‘6’+‘7’+‘8’+‘9’
- digits = digit+
- opt_fraction = (‘.’ digits) + ε = (‘.’ digits)?
- opt_exponent = (‘E’ (‘+’ + ‘-’ + ε) digits ) + ε
  = (‘E’ (‘+’ + ‘-’)? digits )?
- num = digits opt_fraction opt_exponent
Common Regular Expression

- At least one $A^+ \equiv AA^*$
- Union: $A \mid B \equiv A + B$
- Option: $A? \equiv A + \epsilon$
- Range: ‘a’ + … + ‘z’ = [a-z]
- Excluded range: complement of [a-z] ≡ [^a-z]
Summary of Regular Languages

- Regular Expressions specify regular languages

- Five constructs
  - Two base expression
    - Empty and 1-character string
  - Three compound expressions
    - Union, Concatenation, Iteration
Lexical Specification of a language

1. Write a regex for the lexemes of each token class
   - Number = digit+
   - Keywords = \textquoteleft if\textquoteright + \textquoteleft else\textquoteright + ..
   - Identifiers = letter (letter + digit)*
   - LPAR = \textquoteleft (\textquoteright
Lexical Specification of a language

2. Construct $R$, matching all lexemes for all tokens

   $R = \text{Number} + \text{Keywords} + \text{Identifiers} + \ldots$

   $= R_1 + R_2 + R_3 + \ldots$

3. Let input be $x_q \ldots x_n$.

   For $1 \leq i \leq n$, check $x_1 \ldots x_i \in L(R)$

4. If successful, then we know that

   $x_1 \ldots x_i \in L(R_j)$ for some $j$

5. Remove $x_1 \ldots x_i$ from input and go to step 3.
Lexical Specification of a language

- **How much input is used?**
  - $x_1...x_i \in L(R)$
  - $x_1...x_j \in L(R)$, $i \neq j$
  - Which one do we want? (e.g., $==$ or $=$)
  - **Maximal munch**: always choose the longer one

- **Which token is used if more than one matches?**
  - $x_1...x_i \in L(R)$ where $R = R_1 + R_2 + .. + R_n$
  - $x_1...x_i \in L(R_m)$
  - $x_1...x_i \in L(R_n)$, $m \neq n$
  - Eg: Keywords = ‘if’, Identifier = letter (letter + digit)*, if matches both
  - Keyword has higher priority
  - Rule of Thumb: **Choose the one listed first**
Lexical Specification of a language

- What if no rule matches?
  - \( x_1 \cdots x_i \notin L(R) \) ... compiler typically tries to avoid this scenario
  - Error = [all strings not in the lexical spec]
  - Put it in last in priority
Summary so far

- Regular Expressions are concise notations for the string patterns

- Use in lexical analysis with some extensions
  - To resolve ambiguities
  - To handle errors

- Implementation?
  - We will study next
Finite Automata

- Regular Expression = specification
- Finite Automata = implementation

A finite automaton consists of

- An input Alphabet: $\Sigma$
- A finite set of states: $S$
- A start state: $n$
- A set of accepting states: $F \subseteq S$
- A set of transitions state: $\text{state}1 \xrightarrow{\text{input}} \text{state}2$
Transition

- $s_1 \xrightarrow{a} s_2$ (state $s_1$ on input $a$ goes to state $s_2$)
- If end of the input and in final state, the input is accepted
- Otherwise reject

- Language of FA = set of strings accepted by that FA
Example Automata

- a finite automaton that accepts only “1”
Example Automata

- A finite automaton that accepting any number of “1” followed by “0”
### Regular Expression to NFA

- For $\epsilon$ (it’s a choice)

  ![Diagram for $\epsilon$]

- For input $a$

  ![Diagram for input $a$]
Finite Automata

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No $\varepsilon$-moves
  - Takes only one path through the state graph

- **Nondeterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have $\varepsilon$-moves
  - Can choose which path to take
    - An NFA accepts if some of these paths lead to accepting state at the end of input.
Finite Automata

- An NFA can get into multiple states

Input: 1 0 0
Output: \{A\} \{A,B\} \{A,B,C\}
NFA vs. DFA

- NFAs and DFAs recognize the same set of regular languages
- DFAs are faster to execute
  - No choices to consider
- NFAs are, in general, small
Lexical Specification

Regular Expressions

NFA

DFA

Table driven implementation of automata
Lexical Specification

Regular Expressions

NFA

DFA

Table driven implementation of automata
Finite Automata

- For each kind of regex, define an equivalent NFA
  - Notation: NFA for regex M

```
M
```
Regular Expression to NFA

- For $\varepsilon$

  ![Transition Diagram for $\varepsilon$]

- For input $a$

  ![Transition Diagram for input $a$]
Regular Expression to NFA

- For AB

- For A + B
Regular Expression to NFA

- For $A^*$
Example

- \((q+p)^*q\)
Example

Choose the NFA that accepts the regular expression: 1* + 0.
NFA to DFA

Lexical Specification

Regular Expressions

NFA

DFA

Table driven implementation of automata
**$\varepsilon$-closure**

- $\varepsilon$-closure of a state is all the state I can reach following $\varepsilon$ move.

$\varepsilon$-closure(B) = \{B,C,D\}
\(\epsilon\)-closure

- \(\epsilon\)-closure of a state is all the state I can reach following \(\epsilon\) move.

\[\epsilon\text{-closure}(B) = \{B, C, D\}\]
\[\epsilon\text{-closure}(G) = \{A, B, C, D, G, H, I\}\]
An NFA can be in many states at any time

How many different states?

- If NFA has $N$ states, it reaches some subset of those states, say $S$
- $|S| \leq N$
- There are $2^N - 1$ possible subsets (finite number)
NFA to DFA

NFA

- States $S$
- Start state $s$
- Final state $F$
- Transition state
  - $a(X) = \{y \mid x \in X \land x \xrightarrow{a} y\}$
- $\varepsilon - closure$

DFA

- States will be all possible subset of $S$ except empty set
- Start state $= \varepsilon - closure(s)$
- Final state $\{X \mid X \cap F = \emptyset\}$
- $X \xrightarrow{a} Y$ if
  - $Y = \varepsilon - closure(a(X))$
NFA to DFA
NFA to DFA
NFA to DFA
NFA to DFA
NFA to DFA
Example: NFA to DFA
Example: NFA to DFA
NFA to DFA

Lexical Specification

Regular Expressions

NFA

DFA

Table driven implementation of automata
Implementing DFA

- A DFA can be implemented by a 2D table $T$
  - One dimension is states
  - Another dimension is input symbol
  - For every transition $s_i \rightarrow^a s_k$: define $T[i, a] = k$
Implementing DFA

Table A

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

i = p;
state = 0;
while(input[i]) {
    state = A[state,input[i]];
    i++;
}
Implementing DFA

Table A

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

A lot of duplicate entries

Table B

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>T</td>
<td>U</td>
</tr>
</tbody>
</table>

Compact but need an extra indirection
- Inner loop will be slower
Implementing DFA

Deal with set of states rather than single state → inner loop is complicated
Deterministic Finite Automata: Example

```plaintext
{ type token = ELSE | ELSEIF }

rule token =
    parse "else" { ELSE }
    | "elseif" { ELSEIF }
```
Deterministic Finite Automata

{ type token = IF | ID of string | NUM of string }

rule token =
parse "if" { IF }
| ['a'-'z'] ['a'-'z' '0'-'9'] as lit { ID(lit) }
| ['0'-'9'] as num { NUM(num) }