Lesson 4

- Lexical analysis: implementing a scanner
The Reason Why Lexical Analysis is a Separate Phase

• Simplifies the design of the compiler
  – LL(1) or LR(1) parsing with 1 token lookahead would not be possible (multiple characters/tokens to match)

• Provides efficient implementation
  – Systematic techniques to implement lexical analyzers by hand or automatically from specifications
  – Stream buffering methods to scan input

• Improves portability
  – Non-standard symbols and alternate character encodings can be normalized (e.g. UTF8, trigraphs)
Interaction of the Lexical Analyzer with the Parser

Source Program

Lexical Analyzer

Parser

Symbol Table

Token, tokenval

Get next token

error

error
Attributes of Tokens

\[ y := 31 + 28x \]

Lexical analyzer

Parser

\[ \langle \text{id}, "y" \rangle \langle \text{assign}, \rangle \langle \text{num}, 31 \rangle \langle '+' , \rangle \langle \text{num}, 28 \rangle \langle '*' , \rangle \langle \text{id}, "x" \rangle \]

token (lookahead)

tokenval (token attribute)
Tokens, Patterns, and Lexemes

• A token is a classification of lexical units
  – For example: id and num

• Lexemes are the specific character strings that make up a token
  – For example: abc and 123

• Patterns are rules describing the set of lexemes belonging to a token
  – For example: “letter followed by letters and digits” and “non-empty sequence of digits”

• The scanner reads characters from the input till when it recognizes a lexeme that matches the patterns for a token
## Example

<table>
<thead>
<tr>
<th>Token</th>
<th>Informal description</th>
<th>Sample lexemes</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>Characters i, f</td>
<td>if</td>
</tr>
<tr>
<td>else</td>
<td>Characters e, l, s, e</td>
<td>else</td>
</tr>
<tr>
<td>relation</td>
<td>&lt; or &gt; or &lt;= or &gt;= or == or !=</td>
<td>&lt;, !=</td>
</tr>
<tr>
<td>id</td>
<td>Letter followed by letter and digits</td>
<td>pi, score, D2</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric constant</td>
<td>3.14159, 0, 6.02e23</td>
</tr>
<tr>
<td>literal</td>
<td>Anything but “ surrounded by “</td>
<td>“core dumped”</td>
</tr>
</tbody>
</table>
Using Buffer to Enhance Efficiency

| E | = | M | * | C | * | * | 2 | eof |

lexeme beginning

forward (scans ahead to find pattern match)

```
if forward at end of first half then begin
  reload second half;
  forward := forward + 1
end
else if forward at end of second half then begin
  reload first half;
  move forward to beginning of first half
end
else forward := forward + 1;
```
Algorithm: Buffered I/O with Sentinels

```
|   | E | = | M | * | eof | C | * | * | 2 | eof |
```

lexeme beginning

```
forward := forward + 1;
if forward is at eof then begin
  if forward at end of first half then begin
    reload second half;
    forward := forward + 1
  end
else if forward at end of second half then begin
  reload first half;
  move forward to beginning of first half
end
else /* eof within buffer signifying end of input */
  terminate lexical analysis
end
```

Current token

forward (scans ahead to find pattern match)

2nd eof ⇒ no more input!
Specification of Patterns for Tokens: Definitions

• An alphabet $\Sigma$ is a finite set of symbols (characters)
• A string $s$ is a finite sequence of symbols from $\Sigma$
  – $|s|$ denotes the length of string $s$
  – $\varepsilon$ denotes the empty string, thus $|\varepsilon| = 0$
• A language is a specific set of strings over some fixed alphabet $\Sigma$
Specification of Patterns for Tokens: 

*String Operations*

- The *concatenation* of two strings $x$ and $y$ is denoted by $xy$
- The *exponentiation* of a string $s$ is defined by

$$s^0 = \varepsilon$$
$$s^i = s^{i-1}s \quad \text{for } i > 0$$

note that $s\varepsilon = \varepsilon s = s$
Specication of Patterns for Tokens: 

*Language Operations*

- **Union**
  \[ L \cup M = \{ s \mid s \in L \text{ or } s \in M \} \]

- **Concatenation**
  \[ LM = \{ xy \mid x \in L \text{ and } y \in M \} \]

- **Exponentiation**
  \[ L^0 = \{ \varepsilon \}; \quad L^i = L^{i-1}L \]

- **Kleene closure**
  \[ L^* = \bigcup_{i=0,...,\infty} L^i \]

- **Positive closure**
  \[ L^+ = \bigcup_{i=1,...,\infty} L^i \]
Language Operations: Examples

\[ L = \{A, B, C, D\} \quad D = \{1, 2, 3\} \]

- \(L \cup D = \{A, B, C, D, 1, 2, 3\}\)
- \(LD = \{A1, A2, A3, B1, B2, B3, C1, C2, C3, D1, D2, D3\}\)
- \(L^2 = \{AA, AB, AC, AD, BA, BB, BC, BD, CA, ... DD\}\)
- \(L^4 = L^2 \cdot L^2 = ??\)
- \(L^* = \{All \ possible \ strings \ of \ L \ plus \ \varepsilon\}\)
- \(L^+ = L^* - \{\varepsilon\}\)
- \(L \ (L \cup D) = ??\)
- \(L \ (L \cup D)^* = ??\)
Specification of Patterns for Tokens: 

*Regular Expressions*

- **Basis symbols:**
  - $\varepsilon$ is a regular expression denoting language $\{\varepsilon\}$
  - $a \in \Sigma$ is a regular expression denoting $\{a\}$

- **If** $r$ and $s$ are regular expressions denoting languages $L(r)$ and $M(s)$ respectively, then
  - $(r) \mid (s)$ is a regular expression denoting $L(r) \cup M(s)$
  - $(r)(s)$ is a regular expression denoting $L(r)M(s)$
  - $(r)^*$ is a regular expression denoting $L(r)^*$
  - $(r)$ is a regular expression denoting $L(r)$

- **To avoid too many brackets we impose:**
  - Precedence of operators: $(\_)^* > (\_)(\_) > (\_)\mid(\_)$
  - Left-associativity of all operators

- **Example:** $(a) \mid ((b)^*(c))$ can be written as $a \mid b^*c$
EXAMPLES of Regular Expressions

$L = \{ A, B, C, D \}$  \hspace{1cm} $D = \{ 1, 2, 3 \}$

\[
A \mid B \mid C \mid D = L
\]
\[
(A \mid B \mid C \mid D) (A \mid B \mid C \mid D) = L^2
\]
\[
(A \mid B \mid C \mid D)^* = L^*
\]
\[
(A \mid B \mid C \mid D) ((A \mid B \mid C \mid D) \mid (1 \mid 2 \mid 3)) = L (L \cup D)
\]

- A language defined by a regular expression is called a *regular set*
# Algebraic Properties of Regular Expressions

<table>
<thead>
<tr>
<th>AXIOM</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r \mid s = s \mid r$</td>
<td>is commutative</td>
</tr>
<tr>
<td>$r \mid (s \mid t) = (r \mid s) \mid t$</td>
<td>is associative</td>
</tr>
<tr>
<td>$(r \mid s) \mid t = r \mid (s \mid t)$</td>
<td>concatenation is associative</td>
</tr>
<tr>
<td>$r(s \mid t) = rs \mid rt$</td>
<td>concatenation distributes over $\mid$</td>
</tr>
<tr>
<td>$(s \mid t) \mid r = sr \mid tr$</td>
<td></td>
</tr>
<tr>
<td>$\varepsilon r = r$</td>
<td>$\varepsilon$ is the identity element for concatenation</td>
</tr>
<tr>
<td>$r \varepsilon = r$</td>
<td></td>
</tr>
<tr>
<td>$r^* = (r \mid \varepsilon)^*$</td>
<td>relation between $*$ and $\varepsilon$</td>
</tr>
<tr>
<td>$r^{**} = r^*$</td>
<td>$*$ is idempotent</td>
</tr>
</tbody>
</table>
Specification of Patterns for Tokens:  

*Regular Definitions*

- Regular definitions introduce a naming convention with name-to-regular-expression bindings:

  \[
  \begin{align*}
  d_1 & \rightarrow r_1 \\
  d_2 & \rightarrow r_2 \\
  & \cdots \\
  d_n & \rightarrow r_n
  \end{align*}
  \]

  where each \( r_i \) is a regular expression over

  \[ \Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\} \]

- Any \( d_j \) in \( r_i \) can be textually substituted in \( r_i \) to obtain an equivalent set of definitions
Specification of Patterns for Tokens:  
*Regular Definitions*

• Example:

\[
\begin{align*}
\text{letter} & \rightarrow \text{A} | \text{B} | \ldots | \text{Z} | \text{a} | \text{b} | \ldots | \text{z} \\
\text{digit} & \rightarrow 0 | 1 | \ldots | 9 \\
\text{id} & \rightarrow \text{letter} (\text{letter} | \text{digit})^* 
\end{align*}
\]

• Regular definitions cannot be recursive:

\[
\begin{align*}
\text{digits} & \rightarrow \text{digit digits} | \text{digit} \quad \text{wrong!}
\end{align*}
\]
Specification of Patterns for Tokens: Notational Shorthand

- The following shorthands are often used:

  \[ r^+ = rr^* \]
  \[ r? = r | \varepsilon \]
  \[ [a-z] = a | b | c | \ldots | z \]

- Examples:
  - `digit` → `[0-9]`
  - `num` → `digit^+ (. digit^+)? ( E (+ | -)? digit^+ )`
Context-free Grammars and Tokens

• Given the context-free grammar of a language, *terminal symbols* correspond to the tokens the parser will use.

• Example:

• The tokens are:

  if, then, else, relop, id, num

\[
\begin{align*}
  stmt & \rightarrow \text{if } expr \text{ then } stmt \\
  & \quad \mid \text{if } expr \text{ then } stmt \text{ else } stmt \\
  & \quad \mid \varepsilon \\
  expr & \rightarrow \text{term relop term} \\
  & \quad \mid \text{term} \\
  term & \rightarrow \text{id} \\
  & \quad \mid \text{num}
\end{align*}
\]
## Informal specification of tokens and their attributes

<table>
<thead>
<tr>
<th>Pattern of lexeme</th>
<th>Token</th>
<th>Attribute-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Any ws</em></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td><em>if</em></td>
<td><em>if</em></td>
<td>-</td>
</tr>
<tr>
<td><em>then</em></td>
<td><em>then</em></td>
<td>-</td>
</tr>
<tr>
<td><em>else</em></td>
<td><em>else</em></td>
<td>-</td>
</tr>
<tr>
<td><em>Any id</em></td>
<td><em>id</em></td>
<td>pointer to table entry</td>
</tr>
<tr>
<td><em>Any num</em></td>
<td><em>num</em></td>
<td>pointer to table entry</td>
</tr>
<tr>
<td>&lt;</td>
<td><em>relop</em></td>
<td>LT</td>
</tr>
<tr>
<td>&lt;=</td>
<td><em>relop</em></td>
<td>LE</td>
</tr>
<tr>
<td>=</td>
<td><em>relop</em></td>
<td>EQ</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td><em>relop</em></td>
<td>NE</td>
</tr>
<tr>
<td>&gt;</td>
<td><em>relop</em></td>
<td>GT</td>
</tr>
<tr>
<td>&gt;=</td>
<td><em>relop</em></td>
<td>GE</td>
</tr>
</tbody>
</table>
Regular Definitions for tokens

• The specification of the patterns for the tokens is provided with regular definitions

\[
\begin{align*}
\text{if} & \rightarrow \text{if} \\
\text{then} & \rightarrow \text{then} \\
\text{else} & \rightarrow \text{else} \\
\text{relop} & \rightarrow < \mid \leq \mid \geq \mid > \mid >= \mid = \\
\text{id} & \rightarrow \text{letter ( letter | digit )}* \\
\text{num} & \rightarrow \text{digit}^+ (\cdot \text{digit}^+)? (E (+ \mid -)? \text{digit}^+ )? \\
\end{align*}
\]
From Regular Definitions to code

• From the regular definitions we first extract a *transition diagram*, and next the code of the scanner.

• We do this by hand, but it can be automatized.

• In the example the lexemes are recognized either when they are completed, or at the next character. In real situations a longer lookahead might be necessary.

• The diagrams guarantee that the longest lexeme is identified.
Coding Regular Definitions in Transition Diagrams

relop → < | <= | <> | > | >= | =

id → letter ( letter | digit )*

return(relop, LE)
return(relop, NE)
return(relop, LT)
return(relop, EQ)
return(relop, GE)
return(relop, GT)

return(gettoken(), install_id())
Coding Regular Definitions in Transition Diagrams (cont.)

Transition diagram for unsigned numbers

\[ \text{num} \rightarrow \text{digit}^+ (\cdot \text{digit}^+)? (E (\text{+} | -)? \text{digit}^+) ? \]
From Individual Transition Diagrams to Code

• Easy to convert each Transition Diagram into code
• Loop with multiway branch (switch/case) based on the current state to reach the instructions for that state
• Each state is a multiway branch based on the next input channel
Coding the Transition Diagrams for Relational Operators

```java
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing
        until a return or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                if(c == '<') state = 1;
                else if (c == '=') state = 5;
                else if (c == '>') state = 6;
                else fail(); /* lexeme is not a relop */
                break;
            case 1: ...
            ...
            case 8: retract();
                retToken.attribute = GT;
                return(retToken);
        }
    }
    return(retToken);
}
```
Putting the code together

token nexttoken()
{ while (1) {
    switch (state) {
        case 0: c = nextchar();
            if (c==blank || c==tab || c==newline) {
                state = 0;
                lexeme_beginning++;
            }
            else if (c=='<') state = 1;
            else if (c=='=') state = 5;
            else if (c=='>') state = 6;
            else state = fail();
            break;
        case 1:
            ...
        case 9: c = nextchar();
            if (isletter(c)) state = 10;
            else state = fail();
            break;
        case 10: c = nextchar();
            if (isletter(c)) state = 10;
            else if (isdigit(c)) state = 10;
            else state = 11;
            break;
        ...
    }
    return state;
}
Putting the code together: Alternative solutions

- The diagrams can be checked in parallel
- The diagrams can be merged into a single one, typically *non-deterministic*: this is the approach we will study in depth.
Lexical errors

• Some errors are out of power of lexical analyzer to recognize:

\[ fi (a == f(x)) \ldots \]

• However, it may be able to recognize errors like:

\[ d = 2r \]

• Such errors are recognized when no pattern for tokens matches a character sequence
Error recovery

• Panic mode: successive characters are ignored until we reach to a well formed token
• Delete one character from the remaining input
• Insert a missing character into the remaining input
• Replace a character by another character
• Transpose two adjacent characters
• Minimal Distance
The Lex and Flex Scanner Generators

• *Lex* and its newer cousin *flex* are *scanner generators*

• Scanner generators systematically translate regular definitions into C source code for efficient scanning

• Generated code is easy to integrate in C applications
Creating a Lexical Analyzer with Lex and Flex

lem source program lex.l

lex (or flex)

lex.yy.c

C compiler

a.out

input stream

a.out

sequence of tokens
Lex Specification

- A *lex specification* consists of three parts:
  - *regular definitions*, *C declarations in* `% { % }`%
  - *translation rules*
  - *user-defined auxiliary procedures*

- The *translation rules* are of the form:
  
  $p_1 \{ \text{action}_1 \}$
  $p_2 \{ \text{action}_2 \}$
  $\ldots$
  $p_n \{ \text{action}_n \}$
Regular Expressions in Lex

- \x\  match the character \x\n- \.\  match the character .
- "string" match contents of string of characters
- .\  match any character except newline
- ^\  match beginning of a line
- $\  match the end of a line
- [xyz]\  match one character \x, y, or z\ (use \ \ to escape -)
- [^xyz]\ match any character except \x, y, and z
- [a-zA-Z]\ match one of a to z
- r*\  closure (match zero or more occurrences)
- r+\  positive closure (match one or more occurrences)
- r?\  optional (match zero or one occurrence)
- r1r2\ match r1 then r2\ (concatenation)
- r1|r2\  match r1 or r2\ (union)
- ( r )\  grouping
- r1\r2\  match r1 when followed by r2
- \{ d \}\ match the regular expression defined by d
Example Lex Specification 1

Translation rules

Contains the matching lexeme

Invokes the lexical analyzer

```c
{%
#include <stdio.h>
%
%%
[0-9]+ { printf("%s\n", yytext); } 
. | \n { }
%
main()
{ yylex(); }
%
```

```
lex spec.l
gcc lex.yy.c -ll
./a.out < spec.l
```
Example Lex Specification 2

```c
#include <stdio.h>
int ch = 0, wd = 0, nl = 0;
%
delim [ \t]+%
\n % { ch++; wd++; nl++; }
^{delim} { ch+=yyleng; }
{delim} { ch+=yyleng; wd++; }
. { ch++; }
%
main()
{ 
yylex();
    printf("%8d%8d%8d\n", nl, wd, ch);
}
```
Example Lex Specification 3

Translation rules

```c
 %{  #include <stdio.h>  %}
digit     [0-9]
letter    [A-Za-z]
id        {letter}({letter}|{digit})*
%
{digit}+    { printf("number: %s\n", yytext); }  
{id}       { printf("ident: %s\n", yytext); }  
.          { printf("other: %s\n", yytext); }  
%
main()     { yylex();  }
```
Example Lex Specification 4

```c
 %{ /* definitions of manifest constants */
#define LT (256)
...
%
}
delim  [ \t\n]
ws     {delim}+
letter [A-Za-z]
digit  [0-9]
id     {letter}({letter}|{digit})*
number {digit}+(\.{digit}+)?(E[+\-]{digit}+)?
%
{ws}   { }
if     {return IF;}  
then   {return THEN;} 
else   {return ELSE;} 
{id}   {yyval = install_id(); return ID;} 
{number} {yyval = install_num(); return NUMBER;} 
"<"   {yyval = LT; return RELOP;} 
"<="  {yyval = LE; return RELOP;} 
"="   {yyval = EQ; return RELOP;} 
"<>"  {yyval = NE; return RELOP;} 
">"   {yyval = GT; return RELOP;} 
">="  {yyval = GE; return RELOP;}
%
int install_id() ...
%
Return token to parser
Token attribute
Install yytext as identifier in symbol table
```