Lexical Analysis (Scanning)

Chapter 2
Lexical Analysis (Scanning)

• Basic Ideas
  - divide character stream into tokens
  - a token is the smallest logical unit in code
  - common categories:
    
    easy to enum. {  
      keywords: “if”, “for”, “while”, ...
      special symbols: “+”, “-”, “=”, “[”, ...
      number: “4”, “23”, “6.63”, “001”, ...
      ID: “a”, “abs”, “sum”, ...
    }
    
    hard to enum. {  
      ...
    }
Scanning

• Token Categories
  - common categories: keywords, special symbols, number, and ID.
  - e.g.,

```c
int bigger(int a, int b)
{
    int c = 0;
    if (a > b)
        c = a;
    else
        c = b;
    return c;
}
```
Scanning

- Define Tokens
  - define different kinds of tokens in `enum`
Scanning

• Token Attributes
  - A token may carry attributes (e.g., `stringval`, `numberval`, ...)

```c
typedef enum {
  IF,  // "if"
  ELSE, // "else"
  PLUS, // "+
  NUM,  // "23"
  ID,   // "a"
  ...
} TokenType;
```

```
if ... a > 5 ...
```

```
IF ... ID GREATER NUM ...
```

Token Attributes

"a"  "5"  5
Scanning

- Token Attributes
  - A token may carry attributes (e.g., stringval, numerval, ...)

```c
typedef struct
{
    TokenType tokenval;
    char *stringval;
    int numval;
    ...
} TokenRecord;
```
Scanning

- getToken()
  - scanner is often driven by the parser

Before

\[
\begin{array}{c}
a[i] = 4 + 2
\end{array}
\]

After

\[
\begin{array}{c}
a[i] = 4 + 2
\end{array}
\]

recognize and return \{ID, “a”\}
Regular Expressions
Regular Expressions

• Basics
  - A regex $r$ represents a pattern of strings, where
  - the set of strings is called regular language $L(r)$
  - the character set is called alphabet $\Sigma$
  - Given an alphabet $\Sigma$, we can construct regex $r$:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r = \epsilon$</td>
<td>$L(r) = {\epsilon}$</td>
<td>a set w/ an empty string</td>
</tr>
<tr>
<td>$r = \phi$</td>
<td>$L(r) = {}$</td>
<td>empty set</td>
</tr>
<tr>
<td>$r = a, a \in \Sigma$</td>
<td>$L(r) = {a}$</td>
<td></td>
</tr>
</tbody>
</table>
Regular Expressions

• Operations
  - alternation “a|b”
  - concatenation “ab”
  - repetition “a*”

- given regex r and s, \( L(r|s) = L(r) \cup L(s) \)
- given regex r and s, \( L(rs) = L(r)L(s) \)
- given regex r, \( L(r^*) = \{\varepsilon\} \cup L(r) \cup L(rr) \cup L(rrr) \ldots \)
Regular Expressions

• Examples
  - What is the language of $(a|b)^*$?
    \[ \{ \varepsilon, a, b, aa, ab, ba, bb, aaa, ... \} \]
  - What is the language of $a|b^*$?
    \[ \{ \varepsilon, a, b, bb, bbb, bbbb, ... \} \]

Precedence: repetition > concatenation > alternation
Regular Expressions

• Names
  - As a notational simplification

\[(0|1|2|\ldots|9) (0|1|2|\ldots|9)^*\]

\[digit = 0|1|2|\ldots|9\]
\[numseq = digit \ digit^*\]
Regular Expressions

- **Extended Regex**

  - **one or more repetitions**  \(a^+ = aa^*\)
  - **any character**  \(.b = (a|b|c)b\) if \(\Sigma = \{a, b, c\}\)
  - **a range of characters**
    - \([abc]\) or \([a-c]\) = \((a|b|c)\)
    - \([acd]\) = \((a|c|d)\)
  - **not** \(\sim(a|b)\) or \([^ab]\) = \(c\) if \(\Sigma = \{a, b, c\}\)
  - **optional subexpressions**
    - \((a|b)?c = ac|bc|c\)
Regular Expression

• Example
  - Rewrite the regex with only three core operators (concatenation/alternation/repetition)

$$(x+y)? . [^x-y] \text{ assume } \Sigma = \{x, y, z\}$$

$$xx^*y (x | y | z) z \mid (x | y | z)z$$
Token Specification

• Specify tokens with regex
  • Given the complexity, regex is perfect for this purpose

\[
\begin{align*}
\text{keywords: } & \text{"if", "for", "while", ...} \\
\text{special symbols: } & \text{"+", "-", "=", "[", ...} \\
\text{number: } & \text{"4", "23", "6.63", "001", ...} \\
\text{ID: } & \text{"a", "abs", "sum", ...} \\
& \ldots
\end{align*}
\]
Token Specification

- **Numbers**
  - sequence of digits “23”
  - signed numbers “-12”, “+17”
  - decimal numbers “1.24”
  - scientific numbers “2.74E+2”

```plaintext
nat = [0-9]+  
signedNat = [+-]? nat  
decimalNum = signedNat \ nat  
scientificNum = signedNat \ nat \ E signedNat  
number = signedNat (\ nat)? (E signedNat)?
```
Token Specification

• Reserved Words

  \[
  \text{reserved} = \text{if} | \text{while} | \text{do} | \ldots
  \]

• Identifiers
  - Begins with a letter; contains only letters and digits

  \[
  \text{letter} = [a-zA-Z]
  \]

  \[
  \text{digits} = [0-9]
  \]

  \[
  \text{identifier} = \text{letter}(\text{letter} | \text{digit})^*
  \]
Comments
- Typically are “skipped” during scanning
- Still need to be recognized so they can be skipped

\{this is a Pascal comment\} \{{[^\}]\}*\} \\

// this is a C/C++ comment  //[^\n]*
Token Specification

• Ambiguity
  - Token specification may contain ambiguities
  - Existing multiple ways to interpret the same substring

“if”  IF
“if”  identifier

“<>”  not equal
“<“  “>”  less than, greater than

Keyword is preferred!

Longer token is preferred!

(principle of longest substring)
Token Specification

• Token Delimiters
  - Characters that imply a longer string cannot be a token
  - White spaces are delimiters
  - Comments could also be delimiters

“xtemp=ytemp” “=” is not part of any token
“int x” blank/newline/tab are neither
“do//if” comments are neither

whitespace=(newline|blank|tab|comment)+
Token Specification

- Token Delimiters
  - A delimiter ends a token, but not part of that token
  - Should not be consumed, but just be examined - **Lookahead**
Finite Automata
Finite Automata

• Equivalence
  - a regex specifies a regular language
  - FA accepts a regular language
  - regex $\Leftrightarrow$ FA

$$r = b(b|a)^*$$
$$\Sigma = \{a, b\}$$

1 2 2 2 2 2

| b | b | a | a | a | b |

accept!
Finite Automata

- Extensions and Simplification
  - name transitions w/ regex names (also, other and any)
  - error state

\[
\text{identifier} = \text{letter(}\text{letter|digit})^* \\
\text{letter} = [a-zA-Z] \\
\text{digit} = [0-9]
\]

“cs4all”
Finite Automata

• Extensions and Simplification
  - name transitions w/ regex names (also, other and any)
  - error state is often omitted

  \[
  \text{identifier} = \text{letter} (\text{letter} | \text{digit})^* \\
  \text{letter} = [a-zA-Z] \\
  \text{digit} = [0-9]
  \]

  “4all”

  ![Finite Automaton Diagram]

  undefined transition: Error!
Finite Automata

• Exercise
  - FA for recognizing signed numbers

\[
digit = [0\text{-}9] \\
nat = digit+ \\
signedNat = (+\mid-)\text{?} \ nat
\]
Finite Automata

• Exercise
  - FA for recognizing numbers

\[
digit = [0\text{--}9] \\
nat = digit^+ \\
signedNat = (+|\text{--})^? \ nat \\
number = signedNat (\text{\textasciitilde}.)^? \ nat^? (E \ signedNat)^?\]

Finite Automata

• Exercise
  - FA for recognizing comments

\{[^\^\}\}*\}

/* hello */
Finite Automata

- **FA Actions**
  - "normal" state: copy a character to a token buffer
  - accept state: return a token & go back to initial state
  - error state: generate an error

$s \sum m [ i ] = 4 + 2$

always output a token each time getting here?

“longest matching principle”

always an error?

“delimiter”
Finite Automata

- Adjusted FA
  - “error” state becomes an accept state
  - which has no further transition edges
  - *other* is from lookahead

```
sum [ i ] = 4 + 2
```

- a lookahead char
- action: return ID
Finite Automata

- Recognize Multiple Types of Tokens
  - cannot track the states of all different FAs - too expensive!
  - solution: merge different FAs
Finite Automata

• Recognize Multiple Tokens
  - tokens starting with different characters
  - easier to merge: simply combine their starting states

```
>  =  return GE
<  =  return LE
=  =  return EQ
```
Finite Automata

• Recognize Multiple Tokens
  - tokens starting with the same character

- return LE
- return NE
- return LT
Finite Automata

• Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - expensive to run an NFA!

```
<  return LE
=  return NE
>  return LT
<  return LT
```
Finite Automata

- Recognize Multiple Tokens
  - it becomes an **NFA** (*non-deterministic finite automaton*)
  - NFA $\rightarrow$ **DFA** (deterministic ...)

\[
\begin{align*}
\text{return LE} \\
\text{return NE} \\
\text{return LT}
\end{align*}
\]
Finite Automata

• Recognize Multiple Tokens
  - it becomes an NFA (non-deterministic finite automaton)
  - NFA $\rightarrow$ DFA (deterministic ...)
  - DFA adjustment
Finite Automata

Example:
- Draw the FA for recognizing the following two kinds of tokens in a token string:

\[ \text{LT : <} \]
\[ \text{LE : <=} \]

\[ < < = = < \]

\[ \text{[other]} \]
Finite Automata

- Implementation
  - hard-coded

```c
state = 1
while (!EOF)
{
    switch(state)
    case 1:
        if (advance() == '<')
            state = 2
        else
            error & break
    case 2:
        if (advance() == '=')
            state = 3
        else
            state = 4
    case 3:
        output token LE
        state = 1
    case 4:
        output token LT
        stepback()
        state = 1
}
```
Finite Automata

- Implementation
  - transition table

<table>
<thead>
<tr>
<th>state</th>
<th>&lt;</th>
<th>=</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>err</td>
<td>c = advance()</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>c = advance()</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>output LE</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>output LT; c = stepback()</td>
</tr>
<tr>
<td>err</td>
<td>-</td>
<td>-</td>
<td>print error; break</td>
</tr>
</tbody>
</table>

state = 1

c = advance()
while (!EOF)
{
  state = Trans[state][c]
  action(state)
}


Putting it All Together
Putting It All Together

- [Louden Ch. 2]

\[
\begin{align*}
LE &= \leq \\
LT &= <
\end{align*}
\]

1. Regex2DFA Separately
2. Combine DFAs to NFA
3. NFA2DFA conversion
4. Adjust DFA
Putting It All Together (flex)

- flex-generated scanner

\[ LE = \leq \]
\[ LT = < \]

Regex2DFA Separately

Combine DFAs to NFA

NFA2DFA conversion
Putting It All Together (flex)

- **flex-generated scanner**
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

![Diagram](image)
Putting It All Together (flex)

- flex-generated scanner
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

latest accept output: LE
Putting It All Together (flex)

- flex-generated scanner
  - Move forward until impossible (meeting an “error”)
  - Backtrack to find the latest accept state

no accept state! real error!