Lexical Analysis

Scanners, Regular expressions, and Automata
Phases of compilation

Compilers

Read input program → optimization → translate into machine code

front end        mid end        back end

Lexical analysis  →  parsing  →  Semantic analysis

Code generation  →  Assembler

Characters  Characters  Sentences/  Meaning  translation
Characters  Words/strings  statements

Linker
Lexical analysis

- The first phase of compilation
  - Also known as lexer, scanner
  - Takes a stream of characters and returns tokens (words)
  - Each token has a “type” and an optional “value”
  - Called by the parser each time a new token is needed.

- if (a == b)  c = a;

```plaintext
IF
LPARAN
<ID “a”>
EQ
<ID “b”>
RPARAN
<ID “c”>
ASSIGN
<ID “a”>
```
Lexical analysis

- Typical tokens of programming languages
  - Reserved words: class, int, char, bool,…
  - Identifiers: abc, def, mmm, mine,…
  - Constant numbers: 123, 123.45, 1.2E3…
  - Operators and separators: (, ), <, <=, +, -, …

- Goal
  - recognize token classes, report error if a string does not match any class

  Each token class could be

  | A single reserved word: | CLASS, INT, CHAR,… |
  | A single operator: | LE, LT, ADD,… |
  | A single separator: | LPARAN, RPARAN, COMMA,… |
  | The group of all identifiers: | <ID “a”>, <ID “b”>,… |
  | The group of all integer constant: | <INTNUM 1>,… |
  | The group of all floating point numbers | <FLOAT 1.0>… |
Simple recognizers

- Recognizing keywords
  - Only need to return token type

```java
    c ← NextChar()
    if (c == ‘f’) {
        c ← NextChar()
        if (c == ‘e’) {
            c ← NextChar()
            if (c == ‘e’) return <FEE>
        }
    }
    report syntax error
```
Recognizing integers

- Token class recognizer
  - Return <type,value> for each token

```c
    c ← NextChar();
    if (c = ‘0’) then return <INT,0>
    else if (c >= ‘1’ && c <= ‘9’) {
        val = c – ‘0’;
        c ← NextChar()
        while (c >= ‘0’ and c <= ‘9’) {
            val = val * 10 + (c – ‘0’);
            c ← NextChar()
        }
        return <INT,val>
    }
    else report syntax error
```
Multi-token recognizers

c ← NextChar()
if (c == 'f') { c ← NextChar()
    if (c == 'e') { c ← NextChar()
        if (c == 'e') return <FEE> else report error }
    else if (c == 'i') { c ← NextChar()
        if (c == 'e') return <FIE> else report error }
}
else if (c == 'w') { c ← NextChar()
    if (c == 'h') { c ← NextChar(); ...} else report error; }
else report error
c ← NextChar();
while (c==' ' || c=='\n' || c=='\r' || c=='\t')
c ← NextChar();
if (c == '0') then return <INT,0>
else if (c >= '1' && c <= '9') {
    val = c - '0';
c ← NextChar()
while (c >= '0' and c <= '9') {
    val = val * 10 + (c - '0');
c ← NextChar()
}
return <INT,val>
} else report syntax error
Recognizing operators

c $\leftarrow$ NextChar();
while (c==' ' || c=='\n' || c=='\r' || c=='\t')
    c $\leftarrow$ NextChar();
if (c == '0') then return <INT,0>
else if (c >= '1' && c <= '9') {
    val = c - '0';
    c $\leftarrow$ NextChar()
    while (c >= '0' and c <= '9') {
        val = val * 10 + (c - '0');
        c $\leftarrow$ NextChar()
    }
    return <INT,val>
}
else if (c == '<') return <LT>
else if (c == '*') return <MULT>
else ...
else report syntax error
Reading ahead

- What if both “\(\leq\)” and “\(<\)” are valid tokens?

```
c ← NextChar();

......
else if (c == ‘<’) {
    c ← NextChar();
    if (c == ‘=’) return <LE>
    else {PutBack(c); return <LT>;} }
}
else ... else report syntax error
```

```
static char putback=0;
NextChar() {
    if (putback==0) return GetNextChar();
    else { c = putback; putback=0; return c; }
}
Putback(char c) { if (putback==0) putback=c; else error; }
```
Recognizing identifiers

- Identifiers: names of variables <ID, val>
  - May recognize keywords as identifiers, then use a hash-table to find token type of keywords

```c
    c ← NextChar();
    if (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c == '_') {
        val = STR(c);
        c ← NextChar();
        while (c >= 'a' && c <= 'z' || c >= 'A' && c <= 'Z' || c >= '0' && c <= '9' || c == '_') {
            val = AppendString(val, c);
            c ← NextChar();
        }
        return <ID, val>
    }
```

else .....
Describing token types

- Each token class includes a set of strings

```
CLASS = {"class"}; LE = {"="}; ADD = {"+"};
ID = {strings that start with a letter}
INTNUM = {strings composed of only digits}
FLOAT = { ... }
```

- Use formal language theory to describe sets of strings

**An alphabet** $\sum$ is a finit set of all characters/symbols
  
e.g.  \{a,b,\ldots,z,0,1,\ldots\}, {+, -, *, /, <, >, (, )}

**A string over** $\sum$ is a sequence of characters drawn from $\sum$
  
e.g.  “abc” “begin” “end” “class” “if a then b”

**Empty string**: $\varepsilon$

**A formal language is a set of strings over** $\sum$
  
  {“class”}  {“<+”}  {abc, def, ...}, {…-3, -2,-1,0, 1,...}

The C programming language
English
Operations on strings and languages

- **Operations on strings**
  - Concatenation: “abc” + “def” = “abcdef”
    - Can also be written as: $s_1s_2$ or $s_1 \cdot s_2$
  - Exponentiation: $s^i = \underbrace{s \cdots s}_i$

- **Operations on languages**
  - Union: $L_1 \cup L_2 = \{ x \mid x \in L_1 \text{ or } x \in L_2 \}$
  - Concatenation: $L_1L_2 = \{ xy \mid x \in L_1 \text{ and } x \in L_2 \}$
  - Exponentiation: $L^i = \{ x^i \mid x \in L \}$
  - Kleene closure: $L^* = \{ x^i \mid x \in L \text{ and } i \geq 0 \}$
Regular expression

- Compact description of a subset of formal languages
  - \( L(\alpha) \): the formal language described by \( \alpha \)

- Regular expressions over \( \Sigma \),
  - the empty string \( \varepsilon \) is a r.e., \( L(\varepsilon) = \{\varepsilon\} \)
  - for each \( s \in \Sigma \), \( s \) is a r.e., \( L(s) = \{s\} \)
  - if \( \alpha \) and \( \beta \) are regular expressions then
    - \( (\alpha) \) is a r.e., \( L((\alpha)) = L(\alpha) \)
    - \( \alpha \beta \) is a r.e., \( L(\alpha \beta) = L(\alpha)L(\beta) \)
    - \( \alpha | \beta \) is a r.e., \( L(\alpha | \beta) = L(\alpha) \cup L(\beta) \)
    - \( \alpha^\star \) is a r.e., \( L(\alpha^\star) = L(\alpha)^\star \)
Regular expression example

- $\Sigma = \{a, b\}$
  - $a \mid b \Rightarrow \{a, b\}$
  
  $(a \mid b) (a \mid b) \Rightarrow \{aa, ab, ba, bb\}$

- $a^* \Rightarrow \{\varepsilon, a, aa, aaa, aaaa, \ldots\}$

- $aa^* \Rightarrow \{a, aa, aaa, aaaa, \ldots\}$

- $(a \mid b)^* \Rightarrow$ all strings over $\{a, b\}$

- $a (a \mid b)^* \Rightarrow$ all strings over $\{a, b\}$ that start with $a$

- $a (a \mid b)^* b \Rightarrow$ all strings start with and end with $b$
Describing token classes

letter = A | B | C | ... | Z | a | b | c | ... | z
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
ID = letter (letter | digit)*
NAT = digit digit*
FLOAT = digit* . NAT | NAT . digit*
EXP = NAT (e | E) (+ | - | ε) NAT
INT = NAT | - NAT

What languages can be defined by regular expressions?
  alternatives (|) and loops (*)
each definition can refer to only previous definitions
no recursion
Shorthand for regular expressions

- **Character classes**
  - \([abcd]\) = a | b | c | d
  - \([a-z]\) = a | b | ... | z
  - \([a-f0-3]\) = a | b | ... | f | 0 | 1 | 2 | 3
  - \([^a-f]\) = \(\Sigma - [a-f]\)

- **Regular expression operations**
  - Concatenation: \(\alpha \circ \beta = \alpha \beta = \alpha \cdot \beta\)
  - One or more instances: \(\alpha^+ = \alpha \alpha^*\)
  - \(i\) instances: \(\alpha^i = \alpha \alpha \alpha \alpha \alpha\)
  - Zero or one instance: \(\alpha? = \alpha | \varepsilon\)
  - Precedence of operations
    - \(* \gg \circ \gg | \) **when in doubt, use parenthesis**
What languages can be defined by regular expressions?

letter = A | B | C | ... | Z | a | b | c | ... | z
digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
ID = letter (letter | digit)*
NAT = digit digit*
FLOAT = digit* . NAT | NAT . digit*
EXP = NAT (e | E) (+ | - | ε) NAT
INT = NAT | - NAT

What languages can be defined by regular expressions?
alternatives (|) and loops (*)
each definition can refer to only previous definitions
no recursion
Writing regular expressions

- Given an alphabet $\Sigma = \{0, 1\}$, describe
  - the set of all strings of alternating pairs of 0s and pairs of 1s
  - The set of all strings that contain an even number of 0s or an even number of 1s

- Write a regular expression to describe
  - Any sequence of tabs and blanks (white space)
  - Comments in C programming language
Recognizing token classes from regular expressions

- Describe each token class in regular expressions
- For each token class (regular expression), build a recognizer
  - Alternative operator (|) → conditionals
  - Closure operator (*) → loops
- To get the next token, try each token recognizer in turn, until a match is found
  
  ```java
  if (IFmatch()) return IF;
  else if (THENmatch()) return THEN;
  else if (IDmatch()) return ID;
  ....
  ```
Building lexical analyzers

- Manual approach
  - Write it yourself; control your own file IO and input buffering
  - Recognize different types of tokens, group characters into identifiers, keywords, integers, floating points, etc.

- Automatic approach
  - Use a tool to build a state-driven LA (lexical analyzer)
    - Must manually define different token classes

- What is the tradeoff?
  - Manually written code could run faster
  - Automatic code is easier to build and modify
Finite Automata --- finite state machines

- **Deterministic finite automata (DFA)**
  - A set of states $S$
    - A start (initial) state $s_0$
    - A set $F$ of final (accepting) states
  - Alphabet $\Sigma$: a set of input symbols
  - Transition function $\delta: S \times \Sigma \rightarrow S$
    - Example: $\delta(1, a) = 2$

- **Non-deterministic finite automata (NFA)**
  - Transition function $\delta: S \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^S$
    - Where $\varepsilon$ represents the empty string
    - Example: $\delta(1, a) = \{2, 3\}$, $\delta(2, \varepsilon) = 4$

- **Language accepted by FA**
  - All strings that correspond to a path from the start state $s_0$ to a final state $f \in F$
Implementing DFA

Char ← NextChar()
state ← s0
while (char ≠ eof and state ≠ ERROR)
    state ← δ(state, char)
    char ← NextChar()
if (state ∈ F) then report acceptance
else report failure

S = \{s0, s1, s2\}
Σ = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}
δ(s0, 0) = s1
δ(s0, 1-9) = s2
δ(s2, 0-9) = s2
F = \{s1, s2\}
DFA examples

Accepted language: $(a|b)^*abb$

Accepted language: $a^+ \mid b^+$
NFA examples

Accepted language: \((a|b)^*abb\)

Accepted language: \(a^+ \mid b^+\)
Automatically building scanners

- Regular Expressions/lexical patterns $\rightarrow$ NFA
- NFA $\rightarrow$ DFA
- DFA $\rightarrow$ Lexical Analyzer

DFA interpreter:

Char $\leftarrow$ NextChar()
state $\leftarrow$ s0
While (char $\neq$ eof and state $\neq$ ERROR)
    state $\leftarrow\delta$ (state, char)
    char $\leftarrow$ NextChar()
if (state $\in$ F) then report acceptance
Else report failure
Converting RE to NFA

- Thompson’s construction
  - Takes a regexp r and returns NFA N(r) that accepts L(r)
- Recursive rules
  - For each symbol $c \in \Sigma \cup \{\varepsilon\}$, define NFA $N(c)$ as
    - Alternation: if $(r = r_1 \mid r_2)$ build $N(r)$ as
    - Concatenation: if $(r = r_1r_2)$ build $N(r)$ as
    - Repetition: if $(r = r_1^*)$ build $N(r)$ as
RE to NFA examples

\[
\text{a*b*} \\
\text{start} \\
8 \to 2 \to 0 \to 1 \to 3 \to 6 \to 4 \to 5 \to 7 \to 9
\]

\[
(a|b)^* \\
\text{start} \\
6 \to 4 \to 0 \to 1 \to 5 \to 7
\]
Automatically building lexical analyzer

- Token ➔ Pattern
- Pattern ➔ Regular Expression
- Regular Expression ➔ NFA or DFA

NFA:

DFA:

- NFA/DFA ➔ Lexical Analyzer
Lexical analysis generators

**Lexical analysis Specification** → **Lex compiler** → **Transition table**

- **declarations**:
  - N1 RE1
  - ...,
  - Nm REM
  - %{
    - typedef enum {...} Tokens;
  - %}
  - %%
  - P1 {action_1}
  - P2 {action_2}
  - ....
  - Pn {action_n}
  - %%
  - int main() {...}

- **Token classes**

- **Help functions**

**Lexical analyzer**

- **Input buffer**
  - Finite automata simulator
    - Transition table
  - NFA or DFA

- **Transition table**
Using Lex to build scanners

cconst '([^\"]+|\\\")'
sconst "[^"]*"

%pointer

{%
  /* put C declarations here*/
%
%
foo { return FOO; }
bar { return BAR; }
{cconst} { yylval=*yytext;
    return CCONST; }
{sconst} { yylval=mk_string(yytext,yyleng);
    return SCONST; }
[ \t\n\r]+ {}
.
  { return ERROR; }

Lex program (Lex.l)
Lex compiler
lex.yy.c

C compiler
lex.yy.c

Input stream
a.out
a.out
tokens
NFA-based lexical analysis

Specifications

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>action_1</td>
</tr>
<tr>
<td>P2</td>
<td>action_2</td>
</tr>
<tr>
<td>......</td>
<td></td>
</tr>
<tr>
<td>Pn</td>
<td>action_n</td>
</tr>
</tbody>
</table>

(1) Create a NFA $N(p_i)$ for each pattern
(2) Combine all NFAs into a single composite NFA
(3) Simulate the composite NFA: must find the longest string matched by a pattern then continue making transitions until reaching termination
Simulate NFA

- Movement through NFA on each input character
  - Similar to DFA simulation, but must deal with multiple transitions from a set of states
- Idea: each DFA state correspond to a set of NFA states
  - s is a single state
    \[ \varepsilon\text{-closure}(t) = \{s \mid s \text{ is reachable from } t \text{ through } \varepsilon\text{-transitions}\} \]
  - T is a set of states
    \[ \varepsilon\text{-closure}(T) = \{s \mid \exists t \in T \text{ s.t. } s \in \varepsilon\text{-closure}(t)\} \]

```plaintext
S = \varepsilon\text{-closure}(s0); a = nextchar();
while (a != eof)
    S = \varepsilon\text{-closure}( \text{move}(S,a) );
    a = nextchar();
If (S \cap F \neq \emptyset) \text{return "yes"; else return "no"}
```
DFA-based lexical analyzers

- Convert composite NFA to DFA before simulation
  - Match the longest string before termination
  - Match the pattern specification with highest priority

```
add \( \varepsilon\text{-closure}(s0) \) to \( Dstates \) unmarked

while there is unmarked \( T \) in \( Dstates \) do
  mark \( T \);
  for each symbol \( c \) in \( \Sigma \) do begin
    \( U := \varepsilon\text{-closure}(\text{move}(T, c)) \);
    \( Dtrans[T, c] := U \);
    if \( U \) is not in \( Dstates \) then
      add \( U \) to \( Dstates \) unmarked
```
Convert NFA to DFA example

NFA:

```
Dstates = \{ \varepsilon\text{-closure}(s0) \} = \{ \{s0\} \};
Dtrans[{s0},a] = \varepsilon\text{-closure}(\text{move}({s0},a)) = \{s0,s1\};
Dtrans[{s0},b] = \varepsilon\text{-closure}(\text{move}({s0},b)) = \{s0\};
```

```
Dstates = \{\{s0\}, \{s0,s1\}\};
Dtrans[{s0,s1},a] = \varepsilon\text{-closure}(\text{move}({s0,s1},a)) = \{s0,s1\};
Dtrans[{s0,s1},b] = \varepsilon\text{-closure}(\text{move}({s0,s1},b)) = \{s0,s2\};
```

```
Dstates = \{\{s0\}, \{s0,s1\}, \{s0,s2\}\};
Dtrans[{s0,s2},a] = \varepsilon\text{-closure}(\text{move}({s0,s2},a)) = \{s0,s1\};
Dtrans[{s0,s2},b] = \varepsilon\text{-closure}(\text{move}({s0,s2},b)) = \{s0,s3\};
```

```
Dstates = \{\{s0\}, \{s0,s1\}, \{s0,s2\}, \{s0,s3\}\};
Dtrans[{s0,s3},a] = \varepsilon\text{-closure}(\text{move}({s0,s3},a)) = \{s0,s1\};
Dtrans[{s0,s3},b] = \varepsilon\text{-closure}(\text{move}({s0,s3},b)) = \{s0\};
```
Convert NFA to DFA example

DFA:

\[
\begin{align*}
Dstates &= \{\{s0\}, \{s0,s1\}, \{s0,s2\}, \{s0,s3\}\}; \\
Dtrans[\{s0\},a] &= \{s0,s1\}; \\
Dtrans[\{s0\},b] &= \{s0\}; \\
Dtrans[\{s0,s1\},a] &= \{s0,s1\}; \\
Dtrans[\{s0,s1\},b] &= \{s0,s2\}; \\
Dtrans[\{s0,s2\},a] &= \{s0,s1\}; \\
Dtrans[\{s0,s2\},b] &= \{s0,s3\}; \\
Dtrans[\{s0,s3\},a] &= \{s0,s1\}; \\
Dtrans[\{s0,s3\},b] &= \{s0\};
\end{align*}
\]