Lexical Analysis

Regular expressions and Finite Automata
Phases of compilation

Compilers

Read input program $\rightarrow$ optimization $\rightarrow$ translate into machine code

front end $\rightarrow$ mid end $\rightarrow$ back end

Lexical analysis $\rightarrow$ parsing $\rightarrow$ Semantic analysis $\cdots$ Codegen $\rightarrow$ Assembler $\rightarrow$ Linker

Characters $\downarrow$ Words/strings $\rightarrow$ Sentences/statements $\rightarrow$ Meaning $\cdots$ translation
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.

```plaintext
if (a == b)  c = a;
```

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

\[
c \leftarrow \text{NextChar()}
\]
if (c ≠ 'f') then do something
else c \leftarrow \text{NextChar()}
  if (c ≠ 'e') then do something
  else c \leftarrow \text{NextChar()}
  if (c ≠ 'e') then do something
  else report success

\[
c \leftarrow \text{NextChar();}
\]
if (c = '0') then report success
else if (c < '1' or c > '9') then do something
  else c \leftarrow \text{NextChar()}
  while (c >= '0' and c <= '9')
    c \leftarrow \text{NextChar()}
  report success
Multiple token recognizers

```plaintext
c ← NextChar()
if (c ≠ ‘f’) then if (c ≠ ‘w’) then do something
   else  c ← NextChar()
         if (c ≠ ‘h’) then do something
               else ....
else c ← NextChar()
   if (c ≠ ‘e’) then if (c ≠ ‘i’) then do something
               else ...
   else c ← NextChar()
         if (c ≠ ‘e’) then do something
               else report success
```

![Diagram of state transitions](image)
What about automation?

- Each recognizer is a finite state machine (finite automata)
  - Each state remembers what characters have been read and what characters to expect
  - Each state corresponds to a distinct program point in the scanning algorithm
  - No additional storage (other than the input buffer and the current input pointer) is required

- Can we automatically generate the scanning algorithm?
  - Need an language to describe what tokens to recognize
  - Need to translate token descriptions to a finite automata (finite state machine)
  - Need to implement (compile/interpret) the finite automata
Describing tokens

- Each token type is 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 $\Sigma$ is a finite set of all characters/symbols e.g. $\{a,b,...z,0,1,...9\}, \{+, -, *, /, <, >, (, )\}$ A string over $\Sigma$ is a sequence of characters drawn from $\Sigma$ e.g. "abc" "begin" "end" "class" "if a then b" Empty string: $\epsilon$ A formal language is a set of strings over $\Sigma$ $\{"class"\} \ \{"<="\} \ \{abc, def, ...\}, \{...-3, -2,-1,0, 1,...\}$ The C programming language English |
Regular expression

- A subset of formal languages
  - $L(\alpha)$: the formal language described by $\alpha$

- Regular expressions over $\Sigma$ (a recursive definition)
  - 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., and $L((\alpha)) = L(\alpha)$ (parentheses)
    - $\alpha \beta$ is a r.e., and $L(\alpha \beta) = L(\alpha)L(\beta)$ (string concatenation)
    - $\alpha | \beta$ is a r.e., $L(\alpha | \beta) = L(\alpha) \cup L(\beta)$ (alternatives)
    - $\alpha^i$ is a r.e., $L(\alpha^i) = L(\alpha)^i$ (exponentiation $\underbrace{\alpha \cdots \alpha}_{i}$)
    - $\alpha^*$ is a r.e., $L(\alpha^*) = L(\alpha)^*$ (closure: $\varepsilon, \alpha, \alpha \alpha, \alpha \alpha \alpha, \ldots$)
Regular Expression Examples

- **Examples**
  - $a | b \Rightarrow \{a, b\}$
  - $(a | b) (a | b) \Rightarrow \{aa, ab, ba, bb\}$
  - $a^* \Rightarrow \{\epsilon, a, aa, aaa, aaaa, \ldots\}$
  - $aa^* \Rightarrow \{a, aa, aaa, aaaa, \ldots\}$
  - $(a | b)^* \Rightarrow \text{all strings over \{a,b\}}$
  - $a (a | b)^* \Rightarrow \text{all strings over \{a,b\} that start with a}$
  - $a (a | b)^* b \Rightarrow \text{all strings start with and end with b}$

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

- The expressive power of regular expressions
  - Alternatives (|) and loops (*)
  - Each definition can refer to only previous definitions
  - No recursion

- Exercises
  - Strings over \{a,b,c\} that start with a and contain at least 2 c’s
  - How to describe C/C++ comments?
Finite Automata

- Deterministic Finite Automata (DFA)
  - $S$: A set of states; $S_0$: start state; $F$: a set of final states
  - Alphabet $\Sigma$: a set of input symbols
  - Transition function $\delta: S \times \Sigma \rightarrow S$ e.g. $\delta(1, a) = 2$

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

![Diagram of DFA](image)

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

![Diagram of DFA](image)

Accepted language: $a^+ | b^+$
Non-Deterministic Finite Automata (NFA)

- Transition function $\delta: S \times (\Sigma \cap \{\varepsilon\}) \rightarrow 2^S$, where
  - $\varepsilon$ represents the empty string
  - Example: $\delta(1, a) = \{2, 3\}$, $\delta(2, \varepsilon) = 4$

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

Accepted language: $a^+ | b^+$
Implementing DFA

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

$S = \{s0,s1,s2\}$
$\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
$\delta(s0,0) = s1$
$\delta(s0,1-9) = s2$
$\delta(s2,0-9) = s2$
$F = \{s1,s2\}$
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 r.e. $r$ and returns NFA $N(r)$ that accepts $L(r)$

- Recursive rules
  - For each symbol $c \in \Sigma \cap \{\varepsilon\}$, 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

\[ a^*b^* \]

\[ (a|b)^* \]
Converting NFA to DFA

- Each DFA state \( \Leftrightarrow \) a set of equivalent NFA states
- For each NFA state \( s \), compute
  - \( \varepsilon\)-closure(s) = all states reachable from \( s \) via \( \varepsilon \)-transitions

```
add \( \varepsilon\)-closure(s0) to Dstates unmarked
while there is unmarked \( T \) in Dstates do
  mark \( T \);
  for each symbol \( c \) in \( \Sigma \) do begin
    \( U := \varepsilon\)-closure(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:

\[
\begin{array}{c}
\text{start} \\
0 \\
\text{a}
\end{array}
\quad
\begin{array}{c}
\text{a} \\
1 \\
\text{b}
\end{array}
\quad
\begin{array}{c}
b \\
2 \\
\text{b}
\end{array}
\quad
\begin{array}{c}
\text{3}
\end{array}
\]

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

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

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

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

DFA:

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\};