Topic 2: Lexical Analysis

CSED 423
Compiler Design

2014 Fall
POSTECH
Prof. Hanjun Kim
The Compiler

- **Lexical Analysis**
  - Break into tokens
  - Think words, punctuation

- **Syntax Analysis**
  - Parse phase structure
  - Think document, paragraphs, sentences

- **Semantic Analysis**
  - Calculate meaning
Analogy to English – Lexical Analysis

• The first step of understanding English
  • Recognize words – the smallest unit above letters
  • Ex: This is a sentence

• Another Ex: Thi si sas entence
Lexical Analysis

• Lexical Analysis: Breaks stream of ASCII characters (source) into tokens

• Token: Sequence of character treated as a unit

• Each token has a *token type*:

  - **ID**: foo, x, className
  - **NUM**: 50, -100
  - **REAL**: 10.45, 3.14, -2.1
  - **IF**: if
  - **SEMI**: ;
  - **ASSIGN**: =
  - **LPAREN**: (, 
  - **RPAREN**: )

• Some tokens have associated semantic information:

  - **foo**: ID(foo)
  - **-100**: NUM(-100)
  - **10.45**: REAL(10.45)

• White space and comments often discarded
How to recognize the tokens?

\[ x = ( y + 4.0 ) ; \]
Implementing a Lexer

- Lexer (Lexical Analyzer)
  - The first phase of a compiler

- Implementation Options:
  - Write Lexer from scratch
  - Use Lexical Analyzer Generator

- How do we describe the source language tokens to the Lexer Generator?

Diagram:

```
Source → Lexer → Stream of Tokens

? → Lexer Generator
    ↓
Source → Lexer → Stream of Tokens
```
Regular Expressions

- Language: set of strings
- String: finite sequence of symbols that taken from a finite alphabet (ASCII is an alphabet)

Regular Expressions
- Sequence of characters that forms a search pattern
- Usage examples
  - find/replace, filter, lexer, grep, ed
- Each RE stands for a set of strings
- REs specify languages using finite descriptions
Regular Expressions

• Base Cases:
  • Symbol: for each symbol $a$ in alphabet, $a$ is a RE denoting language containing only the string $a$
    • Ex: RE $a$ -> Language $a$
    • Ex: RE 1 -> Language 1

  • Epsilon ($\epsilon$): a language containing only the empty string

• Induction Cases: (assume $M$ and $N$ are regular expressions)
  • Alternation ($M|N$): a RE denoting strings in $M$ or $N$
    • $a | b$ -> $\{a, b\}$

  • Concatenation ($MN$): a RE denoting strings in $M$ concatenated with those in $N$
    • $(a | b)(a | c)$ -> $\{aa, ac, ba, bc\}$

  • Repetition ($M^*$): a RE denoting strings formed by concatenating zero or more strings, all of which are in $M$
    • $(a | b)^*$ -> $\{\epsilon, a, b, aa, ab, ba, bb,aaa, aab, \ldots\}$
Regular Expressions

• More abbreviations
  • \([abcd] : (a | b | c | d)\)
  • \([b-g] : [bcdefg]\)
  • \([b-gM-Qkr] : [bcdefgMNOPQkr]\)
  • \(M? : (M | \epsilon)\)
  • \(M+ : MM^*\)
  • \(\cdot : \) any single character except newline
  • “a.+*” : Quotation, a string in quotes (a.+* in here)
Regular Expression Example

• Examples
  • $(0 \mid 1)^*$ : Binary numbers
  • $(a \mid b)^*aa(a \mid b)$ : Strings of a’s and b’s containing consecutive a’s

• Binary numbers that are multiples of two?
Finite Automata

• Finite Automation (Finite State Machine): a computation model of a machine with limited memory

• Language recognized by FA is a set of strings it accepts
Finite Automata

- A finite automation (Finite State Machine) has:
  - Finite number of states
  - Set of edges, each directed from one state to another, labeled with a single symbol
  - A start state
  - One or more final states
  - ACCEPT - After $n$ transitions for $n$-symbol string, if in final state
  - REJECT – If in non-final state or no valid edge was found during traversal
Finite Automata Examples

- **IF**: if

```
i  1  2  3  f
```

- **NUM**: [0-9]+  

- **ID**: [a-z][a-z0-9]*

- **REAL**: ([0-9]+\".\"[0-9]*) | (\".\"[0-9]+)

- **REAL**: ([0-9]+\".\"[0-9]*) | (\".\"[0-9]+)
Finite Automata

- Finite Automation (Finite State Machine): a computation model of a machine with limited memory

- How can we make Lexer (a program) from REs?
Finite Automata

• Finite Automation (Finite State Machine): a computation model of a machine with limited memory

• How can we make Lexer (a program) from REs?
  1. Change REs to finite automata (NFA)
  2. Link the multiple NFA with epsilon ($\epsilon$, empty string) or duplicated labels – Make one NFA
  3. Make each edge uniquely labeled – Make DFA
Class of Finite Automata

• Deterministic Finite Automata (DFA)
  • Edges leaving a node are uniquely labeled

• Non-deterministic Finite Automata (NFA)
  • Two or more edges leaving a node can be identically labeled
  • An edge can be labeled with $\epsilon$

• Implementing Lexer:
  • RE -> NFA -> DFA
RE to NFA Rules

- **a:** $S \xrightarrow{a} F$
- **ε:** $S \xrightarrow{\epsilon} F$
- **[abc]:** $S \xrightarrow{a} F$
- **a-c:** $S \xrightarrow{a-c} F$

**MN:**

**M|N:**

**M*:**
RE to NFA Examples

• IF: if

• ID: [a-z][a-z0-9]*

• ERROR: .

• NUM: [0-9]+
RE to NFA Example
NFA to DFA Conversion

• Motivation
  • Avoid guessing by trying all possibilities simultaneously
    • Ex: input in – from state 1 to 2 or 4?

• Basic Functions
  • \(edge(s, a)\) : All NFA states reachable from state \(s\) by traversing label \(a\)

  • \(closure(S)\), \(\epsilon\_closure(S)\) : All reachable NFA states from \(s \in S\) without consuming any input (\(\epsilon\))
    \[
    closure(S) = S \cup (\bigcup_{s \in S} edge(s, \epsilon))
    \]

  • \(DFAedge(D, a)\) : All reachable NFA states from \(s \in D\) by traversing \(a\) and \(\epsilon\) edges
    \[
    DFAedge(D, a) = closure(\bigcup_{s \in D} edge(s, a))
    \]
NFA to DFA Example
NFA to DFA Example

\[ \epsilon_{\text{closure}}(5) \]?

\[ \epsilon_{\text{closure}}(7) \]?
NFA to DFA Example

\[ \epsilon_{\text{closure}}(5) = \{5, 8, 6\} \]

\[ \epsilon_{\text{closure}}(7) = \{7, 8, 6\} \]
NFA to DFA Example

1, 4, 9, 14

2

f

3

i

IF

1, 4, 9, 14

a-z

5, 6, 8

ID

6, 7, 8

ID

a-z0-9

15

ERROR

10

0-9

11

12

13

NUM

10

0-9

11

12

13

NUM

\( \epsilon \_closure(10) \)?

\( \epsilon \_closure(12) \)?
NFA to DFA Example

\[ \epsilon_{\text{closure}}(10) ? \]
\[ \epsilon_{\text{closure}}(13) ? \]
NFA to DFA Example

1, 4, 9, 14

2

3

f

IF

i

a-z

ID

5, 6, 8

a-z0-9

10, 11, 13

0-9

NUM

6, 7, 8

ERROR

15

a-z?

a-h,i,j-z

.?

any single character

11, 12, 13

0-9

NUM

0-9

12
NFA to DFA Example

\[ DFAEdge(1, i) ? \]
NFA to DFA Example

DFAEdge(1, i)?
Still sub-optimal …
Minimize DFA

- States $s_1$ and $s_2$ are equivalent
  - If $s_1$ and $s_2$ are both final or both non-final
  - and for any symbol $c$, $\text{trans}[s_1,c] = \text{trans}[s_2,c]$
Minimize DFA

States:
- 1, 4, 9, 14
- 2, 5, 6, 8, 15
- 3, 6, 7, 8
- 5, 6, 7, 8, 15
- 10, 11, 12, 13, 15
- ERROR

Transitions:
- i: 1, 4, 9, 14 → 1, 4, 9, 14
- a-e, g-z, 0-9: 2, 5, 6, 8, 15 → 3, 6, 7, 8
- a-h, j-z: 1, 4, 9, 14 → 5, 6, 7, 8, 15
- other: 1, 4, 9, 14 → ERROR
- 0-9: 15 → ERROR
- 0-9: 10, 11, 12, 13, 15 → ERROR

Symbols:
- ID
- IF
- NUM
The Longest Token

• Lexer must find longest matching token
  • if       IF
  • ifz8     ID not IF, ID

• Save most recent final state and position in stream
• Update when new final state found
How does DFA work?

- if : IF
- \[a-z][a-z0-9]^* : ID
- " " : continue(); (white space)
- other : error();
How does DFA work?

<table>
<thead>
<tr>
<th>Last Final</th>
<th>Current State</th>
<th>Current Input</th>
<th>Accept Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 3</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 0</td>
<td>if iff id0 idA</td>
<td>return IF</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 5</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 0</td>
<td>if iff id0 idA</td>
<td>white space</td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 2</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 3</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 4</td>
<td>if iff id0 idA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 0</td>
<td>if iff id0 idA</td>
<td>return ID(if)</td>
<td></td>
</tr>
</tbody>
</table>

- ↓: most recent position that recognized as a final state
- ↑: current position
- |: input position at each successive call
How does DFA work?

<table>
<thead>
<tr>
<th>Last Final</th>
<th>Current State</th>
<th>Current Input</th>
<th>Accept Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>if iff id0 idA</td>
<td>white space</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>if iff id0 idA</td>
<td>return ID(id0)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>if iff id0 idA</td>
<td>white space</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
</tbody>
</table>

| 1 | i | ID | f | IF |
| 2 | a-e,g-z, 0-9 | a-h,j-z |
| 3 | a-z0-9 |
| 4 | ID |
| 5 | White space |

↓ : most recent position that recognized as a final state
↑ : current position
| : input position at each successive call

Diagram: DFA states and transitions.
# How does DFA work?

<table>
<thead>
<tr>
<th>Last Final State</th>
<th>Current State</th>
<th>Current Input</th>
<th>Accept Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>if iff id0 idA</td>
<td>return ID(id)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>if iff id0 idA</td>
<td>ERROR</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>if iff id0 idA</td>
<td></td>
</tr>
</tbody>
</table>

- `↓` : most recent position that recognized as a final state
- `↑` : current position
- `|` : input position at each successive call

The diagram illustrates the DFA with transitions labeled by input symbols (ID, f, IF) and states (1, 2, 3, 4, 5, 6).
ML-Lex, Lex, Flex ...

- Lexer (Lexical Analyzer)
  - The first phase of a compiler

- Implementation Options:
  - Write Lexer from scratch
  - Use Lexical Analyzer Generator

- `ml-lex` is a lexer generator for ML
- `lex` and `flex` are lexer generators for C
ML-Lex

• Input: a set of rules specifying a lexical analyzer
• Output: a lexical analyzer in ML
• A rule consists of a pattern and an action:
  • Pattern is a regular expression
  • Action is a fragment of ordinary ML code
    • Typically returns a token type to calling function

• Examples:
  if => (print ("Found token IF"));
  [0-9]+ => (print("Found token NUM"));
Lexical Specification

• Lexical specification consists of 3 parts:
  
  User Declarations
  %%
  ML-Lex Definitions
  %%
  Rules

• User Declarations
  • User defined values and functions that are available to the action fragments
  • Two values must be defined in this section:
    • type lexresult
      • Type of the value returned by each rule action
    • fun eof()
      • Called by lexer when end of input stream reached
Lexical Specification

- Lexical specification consists of 3 parts:
  - User Declarations
    ```
    User Declarations
    ```
  - ML-Lex Definitions
    ```
    ML-Lex Definitions
    ```
  - Rules

- ML-Lex Definitions:
  - User can define regular expression abbreviations
    ```
    DIGITS=[0-9]+;
    LETTER=[a-zA-Z];
    ```
  - Define start states to permit multiple lexers to run together:
    ```
    %s STATE1 STATE2 STATE3
    ```
Lexical Specification

- Lexical specification consists of 3 parts:
  - User Declarations
    ```
    %%
    ML-Lex Definitions
    %%
    Rules
    ```
  - Rules
    ```
    <start_state_list> regular_expression => (action_code)
    ```
  - A rule consists of a *pattern* and an *action*:
    - *Pattern* is a regular expression
    - *Action* is a fragment of ordinary ML code
      - Typically returns a token type to calling function
## Rule Patterns

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td>Individual character “a”</td>
</tr>
<tr>
<td><code>{</code></td>
<td>Reserved character {</td>
</tr>
<tr>
<td><code>[abc]</code></td>
<td>`a</td>
</tr>
<tr>
<td><code>[a-zA-Z]</code></td>
<td>Lowercase and capital letters</td>
</tr>
<tr>
<td><code>.</code></td>
<td>Any character except new line</td>
</tr>
</tbody>
</table>
| `
`    | New line |
| `	`   | Tab |
| “abc?” | abc? (abc? is literally taken) |
| `{LETTER}` | Use abbreviation LETTER defined in ML-Lex Definitions |
| `a*`   | 0 or more a’s |
| `a+`   | 1 or more a’s |
| `a?`   | 0 or 1 a |
| `a|b`   | a or b |

```plaintext
if|iff => (print(“Found token IF or IFF”));
[0-9]+ => (print(“Found token NUM”));
```
Rule Actions

- Actions can use various values defined in User Declarations section.

- Two values always available:
  - `type lexresult`
    - Type of the value returned by each rule action
  - `fun eof()`
    - Called by lexer when end of input stream reached

- Several special variables also available to action fragments
  - `yytext` – input substring matched by regular expression
  - `yypos` – file position of beginning of matched string
  - `continue()` – recursively calls lexing engine
Start States

- **Start states** permit multiple lexical analyzers to run together
- Rules prefixed with a start state is matched only when lexer is in that state
- States are entered with **YYBEGIN**
- Example:
  ```
  %%
  %s COMMENT
  %%
  <INITIAL> if => (print("Token IF"));
  <INITIAL> [a-z]+ => (print("Token ID"));
  <INITIAL> "(*)" => (YYBEGIN COMMENT; continue());
  <COMMENT> "*" => (YYBEGIN INITIAL; continue());
  <COMMENT> "\n" | . => (continue());
  ```
<start_state_list> regular_expression => (action_code)

• Regular expression matched only if lexer is in one of the start states in start state list
• If no start state list specified, the rule matches in all states
• Lexer begins in predefined start state: INITIAL

• If multiple rules match in current start state, use Rule Disambiguation
Rule Disambiguation

- **Longest match** – longest initial substring of input that matches regular expression is taken as next token
  - if8 matches ID(“if8”), not IF() and NUM(8)

- **Rule priority** – for a particular substring which matches more than one regular expression with equal length, choose first regular expression in rules section
  - If we want if to match IF(), not ID(“if”), put keyword regular expression before identifier regular expression
Example

(* ml *)

type lexresult = string
fun eof() = (print("End-of-file\n"); "EOF")

%

INT=[1-9][0-9]*;
%s COMMENT;

%

<INITIAL>"/*" => (YYBEGIN COMMENT; continue());
<COMMENT>"*/*" => (YYBEGIN INITIAL; continue());
<COMMENT>"\n" => (continue());
<INITIAL>. => (print("ERR:'" ^ yytext ^ "'.\n");"ERR");
Example in Action

$ cat x.txt
If 999 then 0999
/* This is a comment 099 if */
If 12 then 12
$sml
-CM.make "sources.cm";
...
-MyLexer.tokenize("x.txt");
Token IF
Token INT(999)
Token THEN
ERR: '0'
Token INT(999)
Token IF
Token INT(12)
Token THEN
Token INT(12)
End-of-file
val it = () : unit