Lecture 5

Lexical Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

January 29, 2025

Takeaways from the last class

- Difference between **lexeme** and **token**

Takeaways from the last class

- Difference between **lexeme** and **token**
- Approaches for implementing lexical analyzer

there are three approaches - Assembly languages, High level languages and Generator tools

Takeaways from the last class

- Difference between **lexeme** and **token**
- Approaches for implementing lexical analyzer
- Difficulties while doing lexical analysis

Takeaways from the last class

- Difference between **lexeme** and **token**
- Approaches for implementing lexical analyzer
- Difficulties while doing lexical analysis
- Impact on symbol table while doing lexical analysis

Takeaways from the last class

- Difference between **lexeme** and **token**
- Approaches for implementing lexical analyzer
- Difficulties while doing lexical analysis
- Impact on symbol table while doing lexical analysis
- Partitioning input into tokens.

Takeaways from the last class

- Difference between **lexeme** and **token**
- Approaches for implementing lexical analyzer
- Difficulties while doing lexical analysis
- Impact on symbol table while doing lexical analysis
- Partitioning input into tokens.
- **Maximal Munch**

Extended Regular Expression

Extended Regular Expression

- One or more instances

Extended Regular Expression

- One or more instances

  $$r^+ = rr*$$

Extended Regular Expression

- One or more instances
  $$r^{+} = rr*$$
- Zero or one instance

Extended Regular Expression

- One or more instances

  $r^{+} = rr*$

- Zero or one instance

  r?

Extended Regular Expression

- One or more instances

  $r^+ = rr*$
- Zero or one instance

  r?
- Character classes

Extended Regular Expression

- One or more instances

  $$r^+ = rr*$$

- Zero or one instance

  r?

- Character classes

  $$[a_1 a_2 \ldots a_n]$$

Extended Regular Expression

- One or more instances

  $$r^+ = rr*$$

- Zero or one instance

  r?

- Character classes

  $$[a_1 a_2 \ldots a_n]$$

- Any character

Extended Regular Expression

- One or more instances

  $$r^{+} = rr*$$

- Zero or one instance

  r?

- Character classes

  $$[a_1 a_2 \ldots a_n]$$

- Any character

  $$.$$

Extended Regular Expression

- One or more instances

  $r^+ = rr*$

- Zero or one instance

  r?

- Character classes

  $[a_1 a_2 \dots a_n]$

- Any character

  .

- Beginning of the line

Extended Regular Expression

- One or more instances

  $r^{+} = rr*$

- Zero or one instance

  r?

- Character classes

  $[a_1 a_2 \ldots a_n]$

- Any character

  .

- Beginning of the line

  ^

Extended Regular Expression

- One or more instances

  $r^{+} = rr*$

- Zero or one instance

  r?

- Character classes

  $[a_1 a_2 \dots a_n]$

- Any character

  .

- Beginning of the line

  ^

- End of the line

Extended Regular Expression

- One or more instances

  $r^{+} = rr*$

- Zero or one instance

  r?

- Character classes

  $[a_1 a_2 \ldots a_n]$

- Any character

  .

- Beginning of the line

  ^

- End of the line

  $

Extended Regular Expression

- One or more instances

  $r^{+} = rr*$
- Zero or one instance

  r?
- Character classes

  $[a_1 a_2 \dots a_n]$
- Any character

  .
- Beginning of the line

  ^
- End of the line

  $
- Between m and n occurrence

Extended Regular Expression

- One or more instances

  $r^{+} = rr*$

- Zero or one instance

  r?

- Character classes

  $[a_1 a_2 \ldots a_n]$

- Any character

  .

- Beginning of the line

  ^

- End of the line

  $

- Between m and n occurrence

  r{m,n}

Lexical analyzer generator

- Input to the generator

Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order

Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression (generates kind of token and other bookkeeping information)

Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression (generates kind of token and other bookkeeping information)
- Output of the generator

Lexical analyzer generator

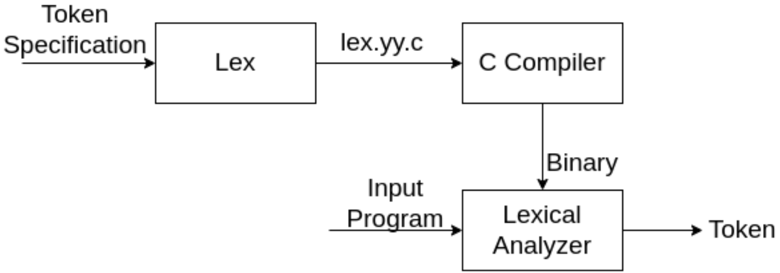
- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression (generates kind of token and other bookkeeping information)
- Output of the generator
  - Program that reads input character stream and breaks that into tokens

Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression (generates kind of token and other bookkeeping information)
- Output of the generator
  - Program that reads input character stream and breaks that into tokens
  - Reports lexical errors (unexpected characters), if any

Lexical analyzer generator

- Input to the generator
  - List of regular expressions in priority order
  - Associated actions for each of regular expression (generates kind of token and other bookkeeping information)
- Output of the generator
  - Program that reads input character stream and breaks that into tokens
  - Reports lexical errors (unexpected characters), if any



Sample Lex File

File Format

declaration

%%

transition rules

%%

auxiliary functions

Sample Lex File

File Format

declaration
%%
transition rules
%%
auxiliary functions

Declaration

- Variables which is going to be used in the rules must be defined in this section.

Sample Lex File

File Format

declaration
%%
transition rules
%%
auxiliary functions

Declaration

- Variables which is going to be used in the rules must be defined in this section.
  D [0-9]

Sample Lex File

File Format

declaration
%%
transition rules
%%
auxiliary functions

Declaration

- Variables which is going to be used in the rules must be defined in this section.
  D [0-9]
- Section enclosed in %{%} delimiter lines are copied to the lex.yy.c file.

Sample Lex File

File Format

declaration
%%
transition rules
%%
auxiliary functions

Declaration

- Variables which is going to be used in the rules must be defined in this section.
  D [0-9]
- Section enclosed in %{%} delimiter lines are copied to the lex.yy.c file.
  %{
  $include < math.h >$
  $int \quad count;$
  %}

Sample Lex file

Transition Rule

Pattern $\{Action\}$

Sample Lex file

Transition Rule

Pattern {*Action*}

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Sample Lex file

Transition Rule

Pattern {*Action*}

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Important functions/variables

- `yyin`

Sample Lex file

Transition Rule

Pattern $\{Action\}$

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Important functions/variables

- `yyin`
- `yylex`

Sample Lex file

Transition Rule

Pattern $\{Action\}$

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Important functions/variables

- `yyin`
- `yylex`
- `yytext`

Sample Lex file

Transition Rule

Pattern $\{Action\}$

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Important functions/variables

- `yyin`
- `yylex`
- `yytext`
- `yyerror`

Sample Lex file

Transition Rule

Pattern {*Action*}

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Important functions/variables

- `yyin`
- `yylex`
- `yytext`
- `yyerror`
- `yyleng`

Sample Lex file

Transition Rule

Pattern {*Action*}

Auxiliary Functions

C Code, going directly into `lex.yy.c`

Important functions/variables

- `yyin` input stream to take the source program
- `yylex` a function called to perform lexical analysis
- `yytext` lexeme that matches pattern
- `yyerror` It is a function ; It is used for explicit error flagging in the code.
- `yyleng` length of yytext or lexeme
- `yylineno` line number tracking

How lex works?

- Regular expressions describe the languages that can be recognized by finite automata

How lex works?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non-deterministic finite automaton

How lex works?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non-deterministic finite automaton
- Convert the NFA into an equivalent DFA

How lex works?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non-deterministic finite automaton
- Convert the NFA into an equivalent DFA
- Minimize the DFA to reduce number of states

How lex works?

- Regular expressions describe the languages that can be recognized by finite automata
- Translate each token regular expression into a non-deterministic finite automaton
- Convert the NFA into an equivalent DFA
- Minimize the DFA to reduce number of states
- Emit code driven by the DFA tables

How to break up text

- How to tokenize $elsex = 0$

How to break up text

- How to tokenize $elsex = 0$
  - $else\ x = 0$
  - $elsex = 0$
- Regular expressions alone are not enough

How to break up text

- How to tokenize $elsex = 0$
  - $else\ x = 0$
  - $elsex = 0$
- Regular expressions alone are not enough
- the longest match wins

How to break up text

- How to tokenize $elsex = 0$
  - $else\ x = 0$
  - $elsex = 0$
- Regular expressions alone are not enough
- the longest match wins
- Ties are resolved by prioritizing tokens

How to break up text

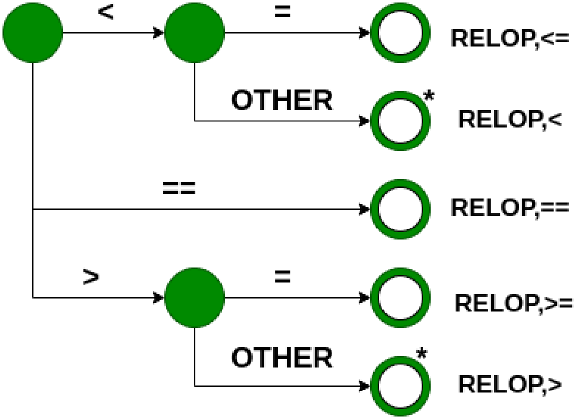
- How to tokenize $elsex = 0$
  - $else\ x = 0$
  - $elsex = 0$
- Regular expressions alone are not enough
- the longest match wins
- Ties are resolved by prioritizing tokens
- Lexical definitions consist of **regular definitions, priority rules**, and **maximal munch principle**

for implementation, all the three are required.

How to break up text

- How to tokenize $elsex = 0$
  - $else\ x = 0$
  - $elsex = 0$
- Regular expressions alone are not enough
- the longest match wins
- Ties are resolved by prioritizing tokens
- Lexical definitions consist of **regular definitions, priority rules**, and **maximal munch principle**
- Construct an analyzer that will return $\langle \texttt{token, lexeme} \rangle$ pairs

Transition Diagram for Relational Operator



Correctness check

- The lexeme for a given token must be the longest possible

Correctness check

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56

Correctness check

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- It can be matched as $\langle \texttt{int}, 12 \rangle$

Correctness check

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- It can be matched as $\langle \texttt{int}, 12 \rangle$
- the matching should always start with the first transition diagram

Correctness check

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- It can be matched as $\langle \texttt{int}, 12 \rangle$
- the matching should always start with the first transition diagram
- If failure occurs in one transition diagram, then retract the forward pointer to the start state and activate the next diagram

Correctness check

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- It can be matched as $\langle \texttt{int}, 12 \rangle$
- the matching should always start with the first transition diagram
- If failure occurs in one transition diagram, then retract the forward pointer to the start state and activate the next diagram
- If failure occurs in all diagrams, then a lexical error has occurred

Correctness check

- The lexeme for a given token must be the longest possible
- Assume input to be 12.34E56
- It can be matched as $\langle \texttt{int}, 12 \rangle$
- the matching should always start with the first transition diagram
- If failure occurs in one transition diagram, then retract the forward pointer to the start state and activate the next diagram
- If failure occurs in all diagrams, then a lexical error has occurred
- This can be implemented using **lots of** `switch-cases` in C programming language.

see book

Interface to other passes