Lecture 4

Lexical Analysis

Awanish Pandey

Department of Computer Science and Engineering
Indian Institute of Technology
Roorkee

January 28, 2025

Lexical Analysis

- Recognize tokens and ignore white spaces and comments

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting
- Model using regular expressions

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata
- Sentences consist of a string of **tokens** (a syntactic category), for example, number, identifier, keyword, string

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata
- Sentences consist of a string of **tokens** (a syntactic category), for example, number, identifier, keyword, string
- Sequences of characters in a token is a **lexeme** for example, 100.01, counter, const, “How are you?”

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata
- Sentences consist of a string of **tokens** (a syntactic category), for example, number, identifier, keyword, string
- Sequences of characters in a token is a **lexeme** for example, 100.01, counter, const, "How are you?"

After lexical analysis of `a = b + c`

Lexical Analysis

- Recognize tokens and ignore white spaces and comments
- Error reporting
- Model using regular expressions
- Recognize using Finite State Automata
- Sentences consist of a string of **tokens** (a syntactic category), for example, number, identifier, keyword, string
- Sequences of characters in a token is a **lexeme** for example, 100.01, counter, const, "How are you?"

After lexical analysis of `a = b + c`

tokens	id	ASSIGN	id	ADDOP	ID
lexeme	a	=	b	+	c

Approaches to Implement

Approaches to Implement

- Use Assembly language

Approaches to Implement

- Use Assembly language

  Most efficient but most difficult to implement

Approaches to Implement

- Use Assembly language

  Most efficient but most difficult to implement
- Use high-level languages like C

Approaches to Implement

- Use Assembly language
  Most efficient but most difficult to implement
- Use high-level languages like C
  Efficient but difficult to implement

Approaches to Implement

- Use Assembly language

  Most efficient but most difficult to implement
- Use high-level languages like C

  Efficient but difficult to implement
- Use tools like `lex`, `flex`

Approaches to Implement

- Use Assembly language
  Most efficient but most difficult to implement
- Use high-level languages like C
  Efficient but difficult to implement
- Use tools like `lex`, `flex`
  Easy to implement but not as efficient as the first two cases

A simple lexical analyzer

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

- Header files and global variable declaration

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

- Header files and global variable declaration
- A global variable `tokenval` is set to the value of the number

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

- Header files and global variable declaration
- A global variable `tokenval` is set to the value of the number
- Check for the whitespace

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

- Header files and global variable declaration
- A global variable `tokenval` is set to the value of the number
- Check for the whitespace
- Check for the new line

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```c
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

- Header files and global variable declaration
- A global variable `tokenval` is set to the value of the number
- Check for the whitespace
- Check for the new line
- Check for digit

A simple lexical analyzer

Allow white spaces, numbers, and arithmetic operators in an expression

```
#include <stdio.h>
#include <ctype.h>
int lineno = 1;
int tokenval = NONE;
int lex() {
  int t;
  while (1) {
    t = getchar ();
    if (t ==' ' || t == 't');
    else if (t == '\n')
      lineno = lineno + 1;
    else if (isdigit (t) ) {
      tokenval = t -'0'
      t = getchar ();
      while (isdigit(t)) {
        tokenval=tokenval * 10 + t - '0'
        t = getchar();
      }
      ungetc(t,stdin);
      return num;
    }
    else { tokenval = NONE; return t; }
  }
}
```

- Header files and global variable declaration
- A global variable `tokenval` is set to the value of the number
- Check for the whitespace
- Check for the new line
- Check for digit
- Else return the scanned character

Problems

Problems

- Scans text character by character

Problems

- Scans text character by character
- The lookahead character determines what kind of token to read and when the current token ends

Problems

- Scans text character by character
- The lookahead character determines what kind of token to read and when the current token ends
- The first character cannot determine what kind of token we are going to read

Symbol Table

Symbol Table

- Stores information for subsequent phases

Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table

Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
    - `Insert(s,t)`: save lexeme s and token t and return pointer

Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - `Insert(s,t)`: save lexeme s and token t and return pointer
  - `Lookup(s)`: return index of entry for lexeme s or 0 if s is not found

Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - `Insert(s,t)`: save lexeme s and token t and return pointer
  - `Lookup(s)`: return index of entry for lexeme s or 0 if s is not found

Implementation of Symbol Table

Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - `Insert(s,t)`: save lexeme s and token t and return pointer
  - `Lookup(s)`: return index of entry for lexeme s or 0 if s is not found

Implementation of Symbol Table

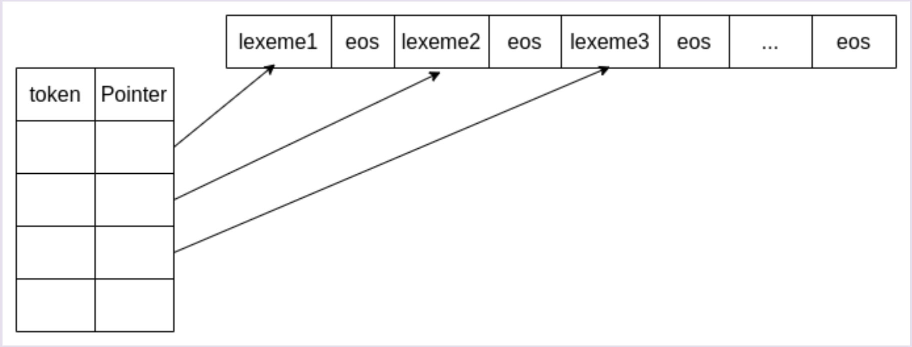
Token	**Lexeme**

Symbol Table

- Stores information for subsequent phases
- Interface to the symbol table
  - `Insert(s,t)`: save lexeme s and token t and return pointer
  - `Lookup(s)`: return index of entry for lexeme s or 0 if s is not found

Implementation of Symbol Table

Token	**Lexeme**



Handling Keyword

Handling Keyword

- Consider tokens INT and CHAR with lexemes int and char.

Handling Keyword

- Consider tokens INT and CHAR with lexemes int and char.
- Initialize symbol table with `insert( "int" , INT )` and `insert( "char" , CHAR)`.

Handling Keyword

- Consider tokens INT and CHAR with lexemes int and char.
- Initialize symbol table with `insert( "int" , INT )` and `insert( "char" , CHAR)`.
- Any subsequent lookup returns a nonzero value, therefore, cannot be used as an identifier.

Handling Keyword

- Consider tokens INT and CHAR with lexemes int and char.
- Initialize symbol table with `insert( "int" , INT )` and `insert( "char" , CHAR)`.
- Any subsequent lookup returns a nonzero value, therefore, cannot be used as an identifier.
- Not an error at lexical analysis phase.

Difficulties in designing the lexical analyzer

Difficulties in designing the lexical analyzer

- Looks simple. Right?

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In `Fortran` `counter` and `co  unter` are same.

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In `Fortran` `counter` and `co  unter` are same.
  - `DO 10 I = 1.25`
  - `DO 10 I = 1,25`

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In `Fortran` `counter` and `co  unter` are same.
  - `DO 10 I = 1.25`
  - `DO 10 I = 1,25`
  - Due to punch cards and errors in punching

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In `Fortran` `counter` and `co  unter` are same.
  - `DO 10 I = 1.25`
  - `DO 10 I = 1,25`
  - Due to punch cards and errors in punching
- Are keywords reserve word?

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In Fortran `counter` and `co  unter` are same.
  - `DO 10 I = 1.25`
  - `DO 10 I = 1,25`
  - Due to punch cards and errors in punching
- Are keywords reserve word?

```
    if then then then = else else else = then // valid in PL/1
```

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In `Fortran` `counter` and `co  unter` are same.
  - `DO 10 I = 1.25`
  - `DO 10 I = 1,25`
  - Due to punch cards and errors in punching
- Are keywords reserve word?

```
if then then then = else else else = then // valid in PL/1
```

- Requires arbitrary lookahead and very large buffers. Worse, the buffers may have to be reloaded.

```
Declare(arg1,arg2,arg3,.......,argn)
Declare(arg1,arg2,arg3,.......,argn) = 0
```

Difficulties in designing the lexical analyzer

- Looks simple. Right?
- Lexemes in a fixed position. Fix format (python) vs. free format languages (C).
- Handling the whitespaces
  - In `Fortran` `counter` and `co  unter` are same.
  - `DO 10 I = 1.25`
  - `DO 10 I = 1,25`
  - Due to punch cards and errors in punching
- Are keywords reserve word?

  ```
  if then then then = else else else = then // valid in PL/1
  ```
- Requires arbitrary lookahead and very large buffers. Worse, the buffers may have to be reloaded.

  ```
  Declare(arg1,arg2,arg3,.......,argn)
  Declare(arg1,arg2,arg3,.......,argn) = 0
  ```
- Even today `Foo<Bar<Bazz>>`

What are we looking for?

- How to describe tokens

What are we looking for?

- How to describe tokens
- How to break text into token

What are we looking for?

- How to describe tokens
- How to break text into token

  `if(x==0)` $a = x \ll 1;$

What are we looking for?

- How to describe tokens
- How to break text into token

  `if(x==0)` a $= x \ll 1;$

  `iff(x==0)` a $= x < 1;$

What are we looking for?

- How to describe tokens
- How to break text into token

  `if(x==0)` $a = x \ll 1;$

  `iff(x==0)` $a = x < 1;$
- How to break the input into tokens efficiently

What are we looking for?

- How to describe tokens
- How to break text into token

  `if(x==0)` a $= x \ll 1;$

  `iff(x==0)` a $= x < 1;$
- How to break the input into tokens efficiently
  - Tokens may have similar prefixes.

What are we looking for?

- How to describe tokens
- How to break text into token

  `if(x==0)` a $= \text{x} \ll 1$;

  `iff(x==0)` a $= \text{x} < 1$;
- How to break the input into tokens efficiently
  - Tokens may have similar prefixes.
  - Each character should be looked at only once.

How to describe token

How to describe token

- By regular languages

How to describe token

- By regular languages
- Regular languages

How to describe token

- By regular languages
- Regular languages
  - Are easy to understand

How to describe token

- By regular languages
- Regular languages
    - Are easy to understand
    - There is a well-understood and useful theory

How to describe token

- By regular languages
- Regular languages
  - Are easy to understand
  - There is a well-understood and useful theory
  - They have efficient implementation

How to describe token

- By regular languages
- Regular languages
  - Are easy to understand
  - There is a well-understood and useful theory
  - They have efficient implementation
- Let r_i be a regular expression and d_i be a distinct name

How to describe token

- By regular languages
- Regular languages
  - Are easy to understand
  - There is a well-understood and useful theory
  - They have efficient implementation
- Let r_i be a regular expression and d_i be a distinct name
- Regular definition is a sequence of definitions of the form

How to describe token

- By regular languages
- Regular languages
  - Are easy to understand
  - There is a well-understood and useful theory
  - They have efficient implementation
- Let r_i be a regular expression and d_i be a distinct name
- Regular definition is a sequence of definitions of the form

  $d_1 \rightarrow r_1$
  $d_2 \rightarrow r_2$
  $d_3 \rightarrow r_3$
  $\ldots\ldots$
  $d_n \rightarrow r_n$

How to describe token

- By regular languages
- Regular languages
  - Are easy to understand
  - There is a well-understood and useful theory
  - They have efficient implementation
- Let r_i be a regular expression and d_i be a distinct name
- Regular definition is a sequence of definitions of the form

  $d_1 \rightarrow r_1$

  $d_2 \rightarrow r_2$

  $d_3 \rightarrow r_3$

  $\ldots\ldots$

  $d_n \rightarrow r_n$
- Where each r_i is a regular expression over $\sum \cup d_1 \cup d_2 \cup \ldots \cup d_{i-1}$

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.
- Just the yes/no answer is not important.

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.
- Just the yes/no answer is not important.
- **Goal: Partition the input into tokens**.

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.
- Just the yes/no answer is not important.
- **Goal: Partition the input into tokens**.

Algorithm

1. Construct R merging all the regular expressions
   $R = R_1 + R_2 + R_3 + \ldots$

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.
- Just the yes/no answer is not important.
- **Goal: Partition the input into tokens**.

Algorithm

1. Construct R merging all the regular expressions
   $R = R_1 + R_2 + R_3 + \dots$
2. Let input be $x_1 \dots x_n$ then task is for $1 \leq i \leq n$ check $x_1 \dots x_i \in L(R)$

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.
- Just the yes/no answer is not important.
- **Goal: Partition the input into tokens**.

Algorithm

1. Construct R merging all the regular expressions
   $R = R_1 + R_2 + R_3 + \ldots$
2. Let input be $x_1 \ldots x_n$ then task is for $1 \leq i \leq n$ check $x_1 \ldots x_i \in L(R)$
3. $x_1 \ldots x_i \in L(R) \rightarrow x_1 \ldots x_i \in L(R_j)$ for some j. smallest such j is token class of $x_1 \ldots x_i$

Regular expressions in specifications

- Regular expressions are only specifications; implementation is still required.
- Given a string s and a regular expression R, does $s \in L(R)$?
- Solution to this problem is the basis of the lexical analyzers.
- Just the yes/no answer is not important.
- **Goal: Partition the input into tokens**.

Algorithm

1. Construct R merging all the regular expressions $R = R_1 + R_2 + R_3 + \ldots$
2. Let input be $x_1 \ldots x_n$ then task is for $1 \le i \le n$ check $x_1 \ldots x_i \in L(R)$
3. $x_1 \ldots x_i \in L(R) \rightarrow x_1 \ldots x_i \in L(R_j)$ for some j. smallest such j is token class of $x_1 \ldots x_i$
4. Remove $x_1 \ldots x_i$ from input; go to (2)

Cont...

- The algorithm gives priority to tokens listed earlier

Cont...

- The algorithm gives priority to tokens listed earlier

  Treats `if` as a keyword and not an identifier

Cont...

- The algorithm gives priority to tokens listed earlier

  Treats `if` as a keyword and not an identifier
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$
  - $x_1 \ldots x_j \in L(R)$

Cont...

- The algorithm gives priority to tokens listed earlier

  Treats `if` as a keyword and not an identifier
- How much input is used? What if
  - $x_1 \dots x_i \in L(R)$
  - $x_1 \dots x_j \in L(R)$
  - Pick up the longest possible string in $L(R)$

Cont...

- The algorithm gives priority to tokens listed earlier

  Treats `if` as a keyword and not an identifier
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$
  - $x_1 \ldots x_j \in L(R)$
  - Pick up the longest possible string in $L(R)$
  - The principle of **maximal munch**
- Regular expressions provide a concise and useful notation for string patterns

Cont...

- The algorithm gives priority to tokens listed earlier

  Treats `if` as a keyword and not an identifier
- How much input is used? What if
  - $x_1 \ldots x_i \in L(R)$
  - $x_1 \ldots x_j \in L(R)$
  - Pick up the longest possible string in $L(R)$
  - The principle of **maximal munch**
- Regular expressions provide a concise and useful notation for string patterns
- Good algorithms require a single pass over the input