07 - Lexical Analysis

Dr. Robert Lowe

Division of Mathematics and Computer Science
Maryville College

Outline

1. Lexical Analysis
2. Regular Expressions
3. Tokenization
4. L++

Outline

1 Lexical Analysis
2 Regular Expressions
3 Tokenization
4 L++

Lexical Analysis Tasks

- The lexical analyzer, or **lexer**, processes the micro-syntax of a language.

Lexical Analysis Tasks

- The lexical analyzer, or **lexer**, processes the micro-syntax of a language.
- The lexer converts sequences of terminals into **lexemes**.

Lexical Analysis Tasks

- The lexical analyzer, or **lexer**, processes the micro-syntax of a language.
- The lexer converts sequences of terminals into **lexemes**.
- A lexeme is the basic building block of a language.

Lexical Analysis Tasks

- The lexical analyzer, or **lexer**, processes the micro-syntax of a language.
- The lexer converts sequences of terminals into **lexemes**.
- A lexeme is the basic building block of a language.
- The lexer has two basic layers/tasks:

Lexical Analysis Tasks

- The lexical analyzer, or **lexer**, processes the micro-syntax of a language.
- The lexer converts sequences of terminals into **lexemes**.
- A lexeme is the basic building block of a language.
- The lexer has two basic layers/tasks:
  1. Scanner

Lexical Analysis Tasks

- The lexical analyzer, or **lexer**, processes the micro-syntax of a language.
- The lexer converts sequences of terminals into **lexemes**.
- A lexeme is the basic building block of a language.
- The lexer has two basic layers/tasks:
    1. Scanner
    2. Screener

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \rightarrow a$ and $A \rightarrow aB$

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \rightarrow a$ and $A \rightarrow aB$
- These productions are the **lexemes** of a language.

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \rightarrow a$ and $A \rightarrow aB$
- These productions are the **lexemes** of a language.
- Example lexemes include:

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \rightarrow a$ and $A \rightarrow aB$
- These productions are the **lexemes** of a language.
- Example lexemes include:
    - Literals

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \rightarrow a$ and $A \rightarrow aB$
- These productions are the **lexemes** of a language.
- Example lexemes include:
    - Literals
    - Identifiers

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \rightarrow a$ and $A \rightarrow aB$
- These productions are the **lexemes** of a language.
- Example lexemes include:
  - Literals
  - Identifiers
  - Keywords

Analyzing the Micro-Syntax

- The micro-syntax of a language is the portion of the language which is expressible as a regular grammar.
- Recall that regular grammars consist of these productions: $A \to a$ and $A \to aB$
- These productions are the **lexemes** of a language.
- Example lexemes include:
  - Literals
  - Identifiers
  - Keywords
- It is possible to use recursive descent for this, but this would be overkill for a lexer!

Scanning

- In the scanning layer, the lexer consumes sequences of characters and transforms them into the productions of the micro-syntax.

Scanning

- In the scanning layer, the lexer consumes sequences of characters and transforms them into the productions of the micro-syntax.
- Scanning can be implemented as recursive descent, but this is not necessary.

Scanning

- In the scanning layer, the lexer consumes sequences of characters and transforms them into the productions of the micro-syntax.
- Scanning can be implemented as recursive descent, but this is not necessary.
- Scanning is typically implemented as a simple state machine.

Scanning

- In the scanning layer, the lexer consumes sequences of characters and transforms them into the productions of the micro-syntax.
- Scanning can be implemented as recursive descent, but this is not necessary.
- Scanning is typically implemented as a simple state machine.
- This is also known as “regular expression parsing”, because the language processed by the lexer is a regular grammar.

Screening

- Screening is the layer which excludes improper characters or sequences of characters.

Screening

- Screening is the layer which excludes improper characters or sequences of characters.
- Usually, screening is done as an exclusionary sort of process.

Screening

- Screening is the layer which excludes improper characters or sequences of characters.
- Usually, screening is done as an exclusionary sort of process.
- Sometimes screened characters are skipped (for example, unneeded whitespace or comments).

Screening

- Screening is the layer which excludes improper characters or sequences of characters.
- Usually, screening is done as an exclusionary sort of process.
- Sometimes screened characters are skipped (for example, unneeded whitespace or comments).
- If something does not match the rules of the lexer's language, it is flagged as an invalid sequence.

The Lexer Interface

- A global variable `symbol`

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`
  - This is where most of the lexer exists.

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`
  - This is where most of the lexer exists.
  - Includes the state machine that consumes the input stream.

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`
  - This is where most of the lexer exists.
  - Includes the state machine that consumes the input stream.
  - Rather than producing single characters, this procedure will process as many characters as needed for the regular productions of the language.

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`
  - This is where most of the lexer exists.
  - Includes the state machine that consumes the input stream.
  - Rather than producing single characters, this procedure will process as many characters as needed for the regular productions of the language.
- The remaining two procedures provide convenience for matching symbols.

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`
  - This is where most of the lexer exists.
  - Includes the state machine that consumes the input stream.
  - Rather than producing single characters, this procedure will process as many characters as needed for the regular productions of the language.
- The remaining two procedures provide convenience for matching symbols.
  - **procedure** `mustbe(s)`

The Lexer Interface

- A global variable `symbol`
- **procedure** `next_symbol`
  - This is where most of the lexer exists.
  - Includes the state machine that consumes the input stream.
  - Rather than producing single characters, this procedure will process as many characters as needed for the regular productions of the language.
- The remaining two procedures provide convenience for matching symbols.
  - **procedure** `mustbe(s)`
  - **procedure** `have(s)`

Outline

1. Lexical Analysis
2. Regular Expressions
3. Tokenization
4. L++

Simple Regular Expression Syntax

- **grouping** `( . . . )`

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`
- **wildcard** `.`

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`
- **wildcard** `.`
- **quantifiers** Quantifiers follow a symbol or a group and specify how many occurrences match.

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`
- **wildcard** `.`
- **quantifiers** Quantifiers follow a symbol or a group and specify how many occurrences match.
  - $*$: zero or more

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`
- **wildcard** `.`
- **quantifiers** Quantifiers follow a symbol or a group and specify how many occurrences match.
  - $*$: zero or more
  - $+$: one or more

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`
- **wildcard** `.`
- **quantifiers** Quantifiers follow a symbol or a group and specify how many occurrences match.
  - `*`: zero or more
  - `+`: one or more
  - `?`: exactly zero or one

Simple Regular Expression Syntax

- **grouping** `(...)`
- **boolean or** `a | b`
- **wildcard** `.`
- **quantifiers** Quantifiers follow a symbol or a group and specify how many occurrences match.
  - `*`: zero or more
  - `+`: one or more
  - `?`: exactly zero or one
- **escaping literals** The special characters in regular expressions can all be escaped in the usual way. `\(, \), \|, \., \*, \*, \*, \?, \\`

Regular Expressions and DFA's

- A regular expression is equivalent to a deterministic finite automaton (DFA).

Regular Expressions and DFA's

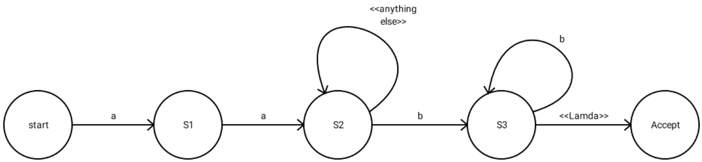
- A regular expression is equivalent to a deterministic finite automaton (DFA).
- A DFA is a graph where states are represented by vertices.

Regular Expressions and DFA's

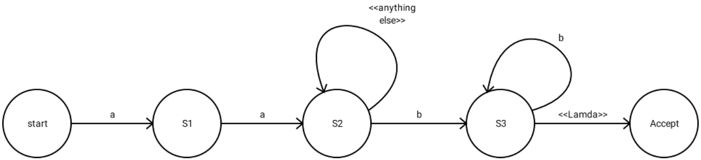
- A regular expression is equivalent to a deterministic finite automaton (DFA).
- A DFA is a graph where states are represented by vertices.
- Edges show transitions from each state for a given character or set of characters.

Regular Expressions and DFA's

- A regular expression is equivalent to a deterministic finite automaton (DFA).
- A DFA is a graph where states are represented by vertices.
- Edges show transitions from each state for a given character or set of characters.
- For example: `aa.*b+` becomes the DFA:

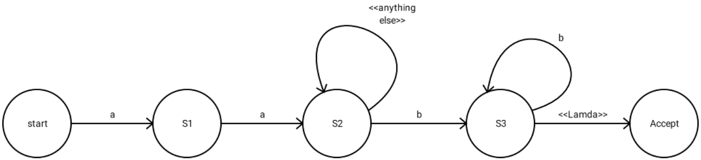


Coding a DFA



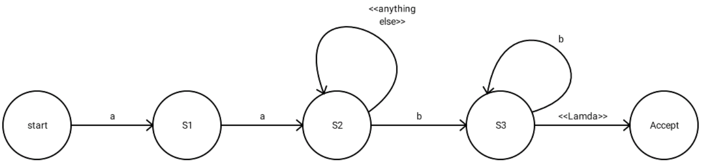
- A DFA can be readily converted into code.

Coding a DFA



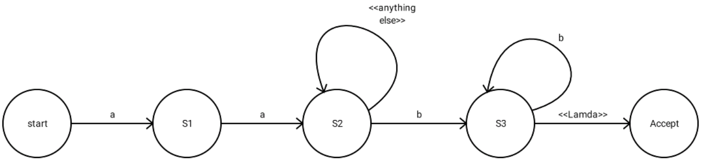
- A DFA can be readily converted into code.
- First, we need an enumeration for the states.

Coding a DFA



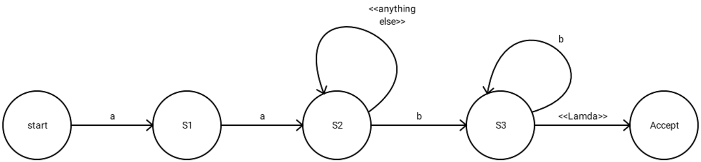
- A DFA can be readily converted into code.
- First, we need an enumeration for the states.
- Then we set the initial state.

Coding a DFA



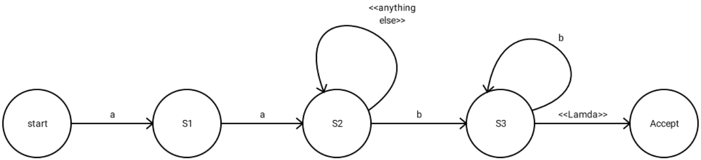
- A DFA can be readily converted into code.
- First, we need an enumeration for the states.
- Then we set the initial state.
- Next, we write a loop that will scan the input.

Coding a DFA



- A DFA can be readily converted into code.
- First, we need an enumeration for the states.
- Then we set the initial state.
- Next, we write a loop that will scan the input.
- Within the loop, we add `if` statements to transition the state. Any invalid transition returns an error.

Coding a DFA



- A DFA can be readily converted into code.
- First, we need an enumeration for the states.
- Then we set the initial state.
- Next, we write a loop that will scan the input.
- Within the loop, we add `if` statements to transition the state. Any invalid transition returns an error.
- **Activity**: Let's code the above DFA!

Outline

1. Lexical Analysis
2. Regular Expressions
3. Tokenization
4. L++

Tokens

- A **token** is a sequence of characters with meaning. (A token is equivalent to a lexeme.)

Tokens

- A **token** is a sequence of characters with meaning. (A token is equivalent to a lexeme.)
- One approach in C++ would be to represent a token using an enumeration:

```
enum Token_Type {INVALID_TOK, OPERATOR_TOK,
                 INTEGER_TOK, LPAREN_TOK,
                 RPAREN_TOK, EOF_TOK};
```

Tokens

- A **token** is a sequence of characters with meaning. (A token is equivalent to a lexeme.)
- One approach in C++ would be to represent a token using an enumeration:

```
enum Token_Type {INVALID_TOK, OPERATOR_TOK,
                 INTEGER_TOK, LPAREN_TOK,
                 RPAREN_TOK, EOF_TOK};
```

- We often need to store some additional information about a token:

```
struct Token {
    Token_Type type;
    string text;
};
```

Identifying Tokens

- Tokens are typically represented by implementing the DFA which represents the regular grammar of the lexer.

Identifying Tokens

- Tokens are typically represented by implementing the DFA which represents the regular grammar of the lexer.
- The basic strategy is this:

Identifying Tokens

- Tokens are typically represented by implementing the DFA which represents the regular grammar of the lexer.
- The basic strategy is this:
  1. Set the state to the start state.

Identifying Tokens

- Tokens are typically represented by implementing the DFA which represents the regular grammar of the lexer.
- The basic strategy is this:
  1. Set the state to the start state.
  2. Peek at the next character and transition to an appropriate state.

Identifying Tokens

- Tokens are typically represented by implementing the DFA which represents the regular grammar of the lexer.
- The basic strategy is this:
  1. Set the state to the start state.
  2. Peek at the next character and transition to an appropriate state.
  3. Continue following transitions until the production ends.

Identifying Tokens

- Tokens are typically represented by implementing the DFA which represents the regular grammar of the lexer.
- The basic strategy is this:
  1. Set the state to the start state.
  2. Peek at the next character and transition to an appropriate state.
  3. Continue following transitions until the production ends.
  4. Emit the token indicated by the current state.

Tokenizing a String

- The global `symbol` variable should be a `Token` structure.

Tokenizing a String

- The global `symbol` variable should be a `Token` structure.
- Each call to `next_symbol` should perform the DFA on the stream.

Tokenizing a String

- The global `symbol` variable should be a `Token` structure.
- Each call to `next_symbol` should perform the DFA on the stream.
- `symbol` is set to the emitted token.

Outline

1. Lexical Analysis
2. Regular Expressions
3. Tokenization
4. L++

The Grammar of L++

$\langle \textit{program} \rangle$ ::= <expression>

$\langle \textit{expression} \rangle$ ::= <term> <expression-tail>

$\langle \textit{expression-tail} \rangle$ ::= λ | ’+’ <term> <expression-tail>

$\langle \textit{term} \rangle$ ::= <factor> <term-tail>

$\langle \textit{term-tail} \rangle$ ::= λ | ’*’ <factor> <term-tail>

$\langle \textit{factor} \rangle$ ::= <integer> | ’(’ <expression> ’)’

$\langle \textit{integer} \rangle$ ::= <unit> | <unit><integer>

$\langle \textit{unit} \rangle$ ::= ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

Regular Expressions of L++

- `integer := (0|1|2|3|4|5|6|7|8|9)+`

Regular Expressions of L++

- `integer := (0|1|2|3|4|5|6|7|8|9)+`
- `operator := +|*`

Regular Expressions of L++

- `integer := (0|1|2|3|4|5|6|7|8|9)+`
- `operator := +|*`
- `lparen := \(`

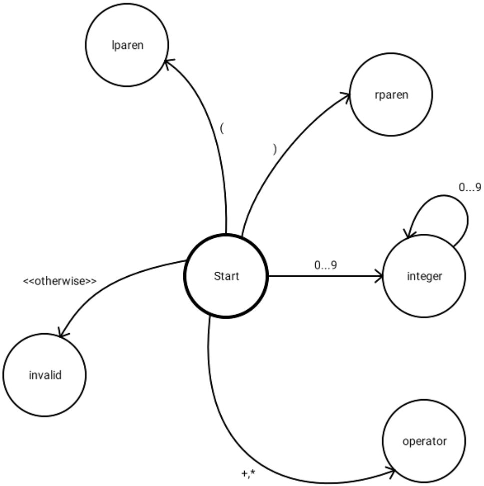
Regular Expressions of L++

- `integer := (0|1|2|3|4|5|6|7|8|9)+`
- `operator := +|*`
- `lparen := \(`
- `rparen := \)`

Regular Expressions of L++

- `integer := (0|1|2|3|4|5|6|7|8|9)+`
- `operator := +|*`
- `lparen := \(`
- `rparen := \)`
- `invalid :=` anything else.

L++ DFA


lparen
rparen
(
)
0...9
Start
0...9
integer
<<otherwise>>
invalid
operator
+,*