Chapter 3: Lexical Analysis

Principles of Programming Languages

Contents

- Terminology
- Chomsky Hierarchy
- Lexical analysis in syntax analysis
- Using Finite Automata to describe tokens
- Using Regular Expression to describe tokens
- Regex Library in Scala

Introduction

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
    - Programmers (the users of the language)

Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences

Terminology

- Sentences: `a = b + c;` or `c = (a + b) * c;`
- Syntax:

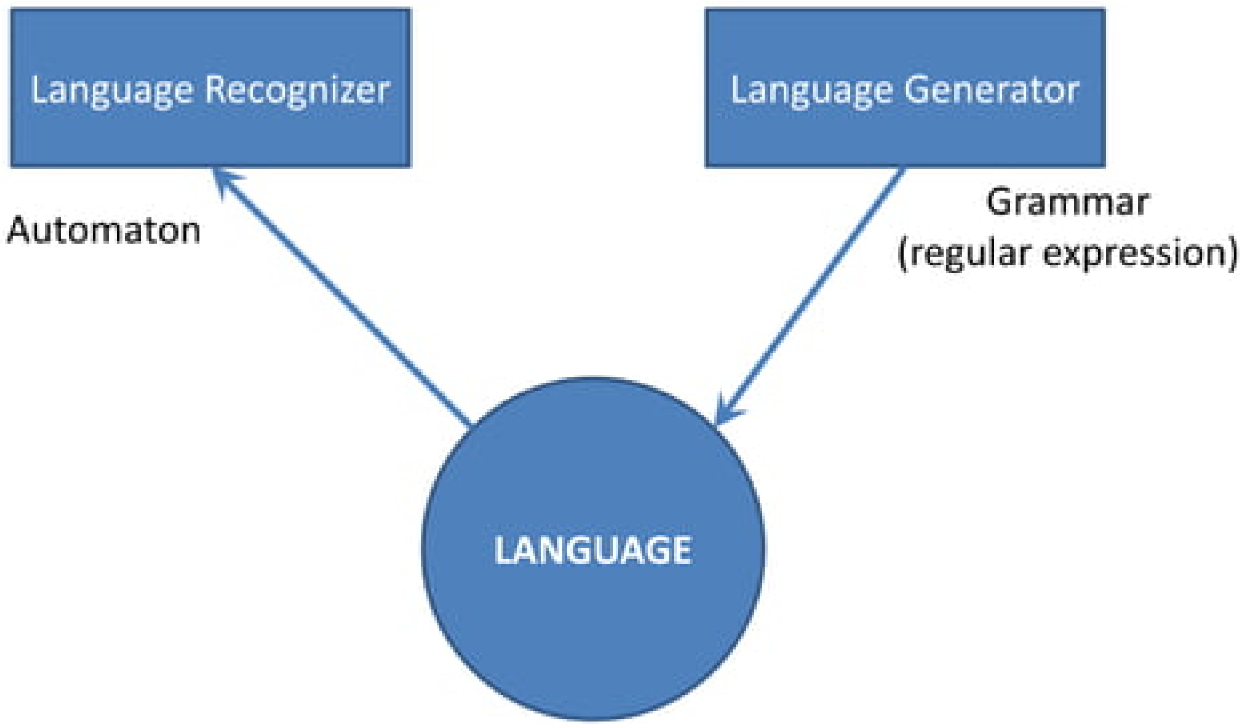
```
<assign> → <id> = <expr> ;
<id>     → a | b | c
<expr>   → <id> + <expr>
           | <id> * <expr>
           | ( <expr> )
           | <id>
```

- Sematics of <u>`a = b + c;`</u>

Formal Definition of Languages

- **Recognizers**
  - A recognition device reads input strings of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler
- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is correct by comparing it to the structure of the generator

Recognizers vs. Generators



Chomsky Hierarchy

Grammars	Languages	Automaton	Restrictions (w1 → w2)
Type-0	Phrase-structure	Turing machine	w1 = any string with at least 1 non-terminal w2 = any string
Type-1	Context-sensitive	Bounded Turing machine	w1 = any string with at least 1 non-terminal w2 = any string at least as long as w1
Type-2	Context-free	Non-deterministic pushdown automaton	w1 = one non-terminal w2 = any string
Type-3	Regular	Finite state automaton	w1 = one non-terminal w2 = tA or t (t = terminal A = non-terminal)

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* - separation allows optimization of the lexical analyzer
- *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a "front-end" for the parser
- Identifies substrings of the source program that belong together – *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*

Lexeme vs. Token

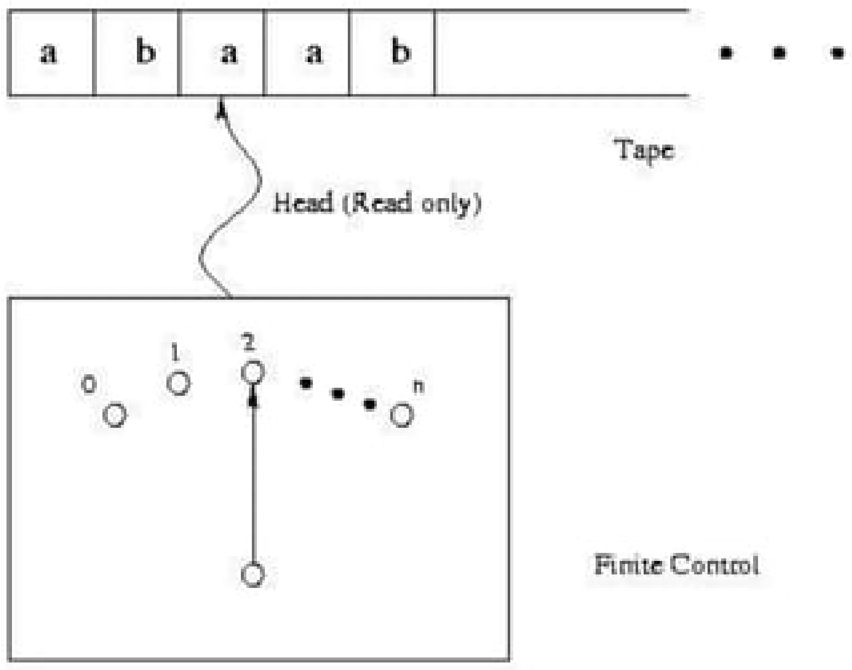
```
result = oldsum – value / 100;
```

Lexemes	*Tokens*
`result`	`IDENT`
`=`	`ASSIGN_OP`
`oldsum`	`IDENT`
`–`	`SUBSTRACT_OP`
`value`	`IDENT`
`/`	`DIVISION_OP`
`100`	`INT_LIT`
`;`	`SEMICOLON`

Lexical Analysis

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram
  - Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description

Deterministic Finite Automata



Finite Automaton

DFA

- DFA is a 5-tuple $M = (K, \Sigma, \delta, s, F)$
- K = a finite set of states
- Σ = alphabet
- $s \in K$ is the initial state
- $F \subseteq K$ is the set of final states
- δ transition function, a function from $K \times \Sigma$ to K

DFA

- E.g., $M = (K, \Sigma, \delta, s, F)$

$K = \{q_0, q_1\}$ $\Sigma = \{a, b\}$ $s = q_0$ $F = \{q_0\}$

δ

q	**σ**	**δ(q, σ)**
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

What is the language accepted by *M, a.k.a. L(M)?*

- Test with the input *aabba*

DFA

- Test with the input *aabba*

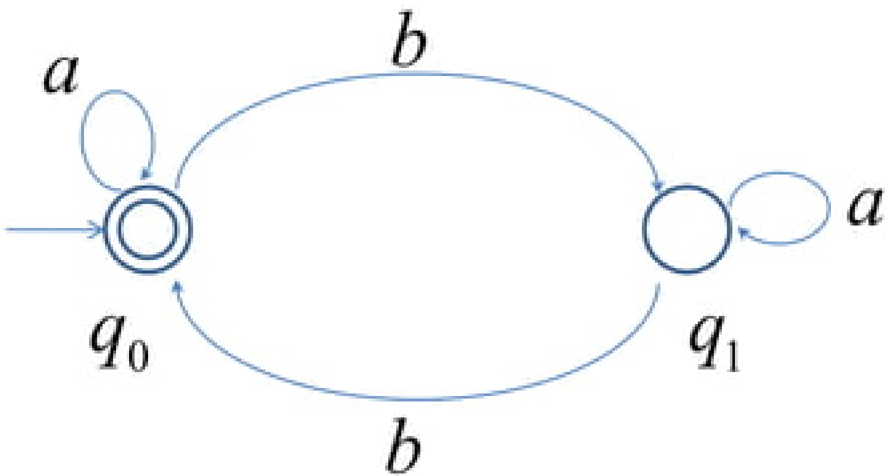
$$(q_0, aabba) \rightarrow (q_0, abba) \rightarrow (q_0, bba) \rightarrow (q_1, ba) \rightarrow (q_0, a) \rightarrow (q_0, e)$$

- Or we can say

$$(q_0, aabba) \xrightarrow{*} (q_0, e)$$

- So, *aabba* is accepted by *M*

State Diagram



q	σ	δ(q, σ)
q_0	a	q_0
q_0	b	q_1
q_1	a	q_1
q_1	b	q_0

Example

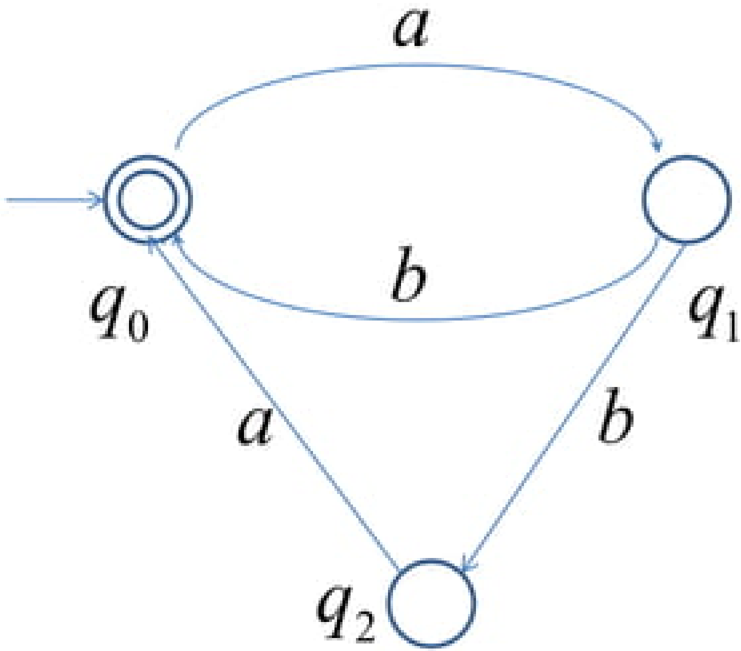
- Design a DFA M that accepts the language $L(M) = \{w: w \in \{a,b\}^*$ and w does not contain three consecutive b's$\}$.

Nondeterministic Finite Automata

- Permit several possible “next states” for a given combination of current state and input symbol
- Accept the empty string *e* in state diagram
- Help simplifying the description of automata
- Every NFA is equivalent to a DFA

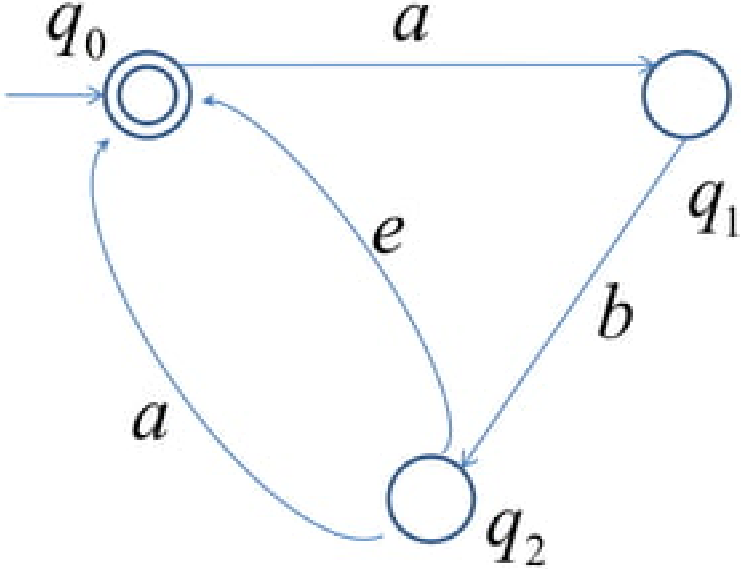
Example

- Language $L = (ab \cup aba)^*$



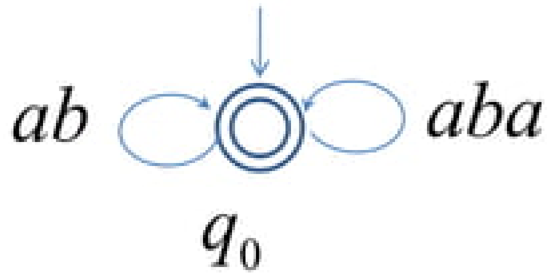
Example

- Language $L = (ab \cup aba)^*$



Example

- Language $L = (ab \cup aba)^*$



Example

- Design a NFA that accepts the following definition for IDENT
  - Starts with a letter
  - Has any number of letter or digit or "_" afterwards

Regular Expression (regex)

- Describe “regular” sets of strings
- Symbols other than ( ) | * stand for themselves
- **Concatenation** $\alpha\ \beta$ = First part matches α, second part β
- **Union** $\alpha \mid \beta$ = Match α or β
- **Kleene star** α^* = 0 or more matches of α
- Use ( ) for grouping

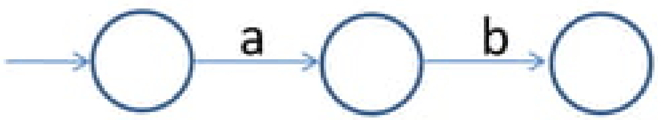
Regular Expression (regex)

`E(0|1|2|3|4|5|6|7|8|9)*`

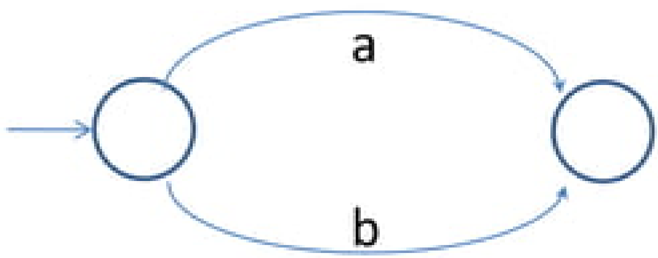
- An E followed by a (possibly empty) sequence of digits

```
E123
E9
E
```

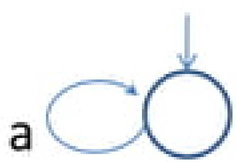
Regular Expression (regex)



ab



a | b



a*

Convenience Notation

- $\alpha+$ = one or more (i.e. $\alpha\alpha^*$)
- $\alpha?$ = 0 or 1 (i.e. $(\alpha|e)$)
- `[xyz]` = `x|y|z`
- `[x-y]` = all characters from `x` to `y`, e.g. `[0-9]` = all ASCII digits
- `[^x-y]` = all characters other than `[x-y]`

Convenience Notation

- `\p{`*Name*`}`, where *Name* is a Unicode category (ex. L, N, Z for letter, number, space)
- `\P{`*Name*`}`: complement of `\p{`*Name*`}`
- `.` matches any character
- `\` is an escape. For example, `\.` is a period, `\\` a backslash

Regex Examples

- **Reserved words**: easy

```
WHILE = while          BEGIN = begin
DO    = do             END   = end
```

- **Integers:** `[+-]?[0-9]+`, or maybe `[+-]?\p{N}+`
- Note: + loses its normal meaning inside [ ], and a - just before] denotes itself

Regex Examples

- Hexadecimal numbers `0[Xx][0-9A-Fa-f]+`
- Quoted C++ strings: `".*"`
- Well, actually not; the . will match a quote
- Better: `"[^"]*"`
- Well, actually not; you can have a `\"` in a quoted string
- `"([^"\\]|\\.)*"`

Exercises

- IDENT
  - Starts with a letter
  - Has any number of letter or digit or "_" afterwards
- C++ floating-point literals
  - See http://msdn.microsoft.com/en-us/library/tfh6f0w2.aspx

Scala Regex Library

- Find all matches

```
import scala.util.matching._
val regex = new Regex("[0-9]+")
regex.findAllIn("99 bottles, 98 bottles").toList
List[String] = List(99, 98)
```

Check whether beginning matches

```
regex.findPrefixOf("99 bottles, 98
  bottles").getOrElse(null)
String = 99
```

Scala Regex Library

- Groups

```
val regex = new Regex("([0-9]+) bottles")
val matches = regex.findAllIn("99 bottles,
  98 bottles, 97 cans").matchData.toList
matches :
  List[scala.util.matching.Regex.Match] =
  List(99 bottles, 98 bottles)
matches.map(_.group(1))
List[String] = List(99, 98)
```

Exercises

- Find NFA and regex for $a^n b^m$: $n+m$ is even
- Find NFA and regex for $a^n b^m$: $n \geq 1$, $m \geq 1$

Remind

- Design a state diagram that describes the tokens and write a program that implements the state diagram
- Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram
- Write a formal description of the tokens and use a software tool that constructs table-driven lexical analyzers given such a description

What do lexical analyzers do?

- Lexical analyzers extract lexemes from a given input string and produce the corresponding tokens
- Old compilers: processed an entire source program
- New compilers: locate the next lexeme with token code, then return to syntax analyzer

What else?

- Skip comments and blanks outside lexemes
- Insert user-defined lexemes into the symbol table
- Detect syntactic errors in tokens
  - e.g. ill-formed floating-point literals

In the next lecture

- How can we describe grammar?
- What do syntax analyzers do after receiving lexemes from lexical analyzers?
- Build grammar for some parts of your popular programming languages

Summary

- Syntax analysis is a common part of language implementation
- A lexical analyzer is a pattern matcher that isolates small-scale parts of a program
- Regular expressions are built based on Finite Automata