Lexical Analysis

CS143

Lecture 3

Instructor: Fredrik Kjolstad

Slide design by Prof. Alex Aiken, with modifications

Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexers (aka. scanners)
  - By regular expressions (aka. regex)
  - Examples of regular expressions

Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
    Z = 0;
else
    Z = 1;
```

- The input is just a string of characters:

```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```

- Goal: Partition input string into substrings
  - Where the substrings are called tokens

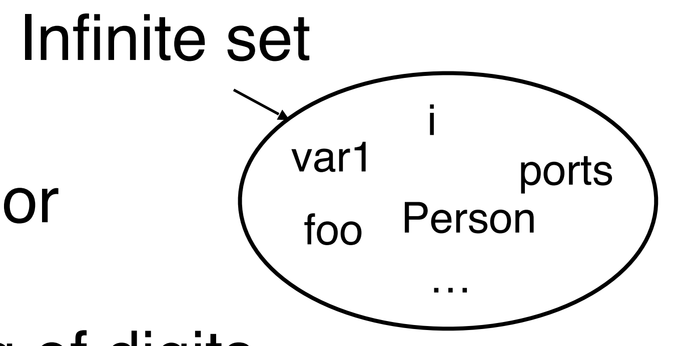
What's a Token?

- A syntactic category
  - In English:
    noun, verb, adjective, …
  - In a programming language:
    Identifier, Integer, Keyword, Whitespace, …

Tokens

- A token class corresponds to a set of strings

- Examples
  – Identifier: strings of letters or digits, starting with a letter
  – Integer: a non-empty string of digits
  – Keyword: "else" or "if" or "begin" or …
  – Whitespace: a non-empty sequence of blanks, newlines, and tabs



What are Tokens For?

- Classify program substrings according to role

- Lexical analysis produces a stream of tokens
- … which is input to the parser

- Parser relies on token distinctions
  - An identifier is treated differently than a keyword

Designing a Lexical Analyzer: Step 1

- Define a finite set of tokens
  - Tokens describe all items of interest
    - Identifiers, integers, keywords
  - Choice of tokens depends on
    - language
    - design of parser

Example

- Recall
  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

- Useful tokens for this expression:
  Integer, Keyword, Relation, Identifier, Whitespace, (,), =, ;

- N.B., (,), =, ; above are tokens, not characters

Designing a Lexical Analyzer: Step 2

- Describe which strings belong to each token

- Recall:
  - Identifier: strings of letters or digits, starting with a letter
  - Integer: a non-empty string of digits
  - Keyword: “else” or “if” or “begin” or …
  - Whitespace: a non-empty sequence of blanks, newlines, and tabs

Lexical Analyzer: Implementation

- An implementation must do two things:

  1. Classify each substring as a token

  2. Return the value or lexeme (value) of the token
     - The lexeme is the actual substring
     - From the set of substrings that make up the token

- The lexer thus returns token-lexeme pairs
  - And potentially also line numbers, file names, etc. to improve later error messages

Example

- Recall:

  \tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;

Lexical Analyzer: Implementation

- The lexer usually discards “uninteresting” tokens that don’t contribute to parsing.
- Examples: Whitespace, Comments

True Crimes of Lexical Analysis

- Is it as easy as it sounds?
- Sort of… if you do not make it hard!
- Look at some history

Lexical Analysis in FORTRAN

- FORTRAN rule: Whitespace is insignificant
- E.g., VAR1 is the same as VA R1
- A terrible design!
- Historical footnote: FORTRAN Whitespace rule motivated by inaccuracy of punch card operators

FORTRAN Example

- Consider
  - DO 5 I = 1,25
  - DO 5 I = 1.25

Lexical Analysis in FORTRAN (Cont.)

- Two important points:

  1. The goal is to partition the string. This is implemented by reading left-to-right, recognizing one token at a time

  2. “Lookahead” may be required to decide where one token ends and the next token begins

Lookahead

- Even our simple example has lookahead issues
  - i vs. if
  - = vs. ==

Lexical Analysis in PL/I

- PL/I keywords are not reserved

  IF ELSE THEN THEN = ELSE; ELSE ELSE = THEN

Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:

  DECLARE (ARG1,. . ., ARGN)

- Cannot tell whether DECLARE is a keyword or array reference until after the).
  - Requires arbitrary lookahead!

Lexical Analysis in C++

- Unfortunately, the problems continue today

- C++ template syntax:

```
Foo<Bar>
```

- C++ stream syntax:

```
cin >> var;
```

- But there is a conflict with nested templates:

```
Foo<Bar<Bazz>>
```

Closing templates, not stream

Review

- The goal of lexical analysis is to
  - Partition the input string into lexemes
  - Identify the token of each lexeme
- Left-to-right scan => lookahead sometimes required

Next

- We still need
  - A way to describe the lexemes of each token

  - A way to resolve ambiguities
    - Is if two variables i and f?
    - Is == two equal signs = =?

Regular Languages

- There are several formalisms for specifying tokens

- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

Languages

Def. Let alphabet Σ be a set of characters. A language over Σ is a set of strings of characters drawn from Σ.

Examples of Languages

- Alphabet = English characters
- Language = English sentences

- Not every string of English characters is an English sentence

- Alphabet = ASCII

- Language = C programs

- Note: ASCII character set is different from English character set

Notation

- Languages are sets of strings.

- Need some notation for specifying which sets we want

- The standard notation for regular languages is regular expressions.

Atomic Regular Expressions

- Single character

$$'c' = \{"c"\}$$

- Epsilon

$$\varepsilon = \{""\}$$

Not the empty set, but set with a single, empty, string.

Compound Regular Expressions

- Union

$$A + B = \{s \mid s \in A \text{ or } s \in B\}$$

- Concatenation

$$AB = \{ab \mid a \in A \text{ and } b \in B\}$$

- Iteration

$$A^* = \cup_{i \geq 0} A^i \text{ where } A^i = \underbrace{AA\ldots A}_{\text{i times}}$$

Regular Expressions

- **Def.** The regular expressions over Σ are the smallest set of expressions including

$$\varepsilon$$

$$'c' \quad \text{where } c \in \Sigma$$

$$A + B \quad \text{where } A, B \text{ are rexp over } \Sigma$$

$$AB \quad \text{" \quad " \quad "}$$

$$A^* \quad \text{where } A \text{ is a rexp over } \Sigma$$

Syntax vs. Semantics

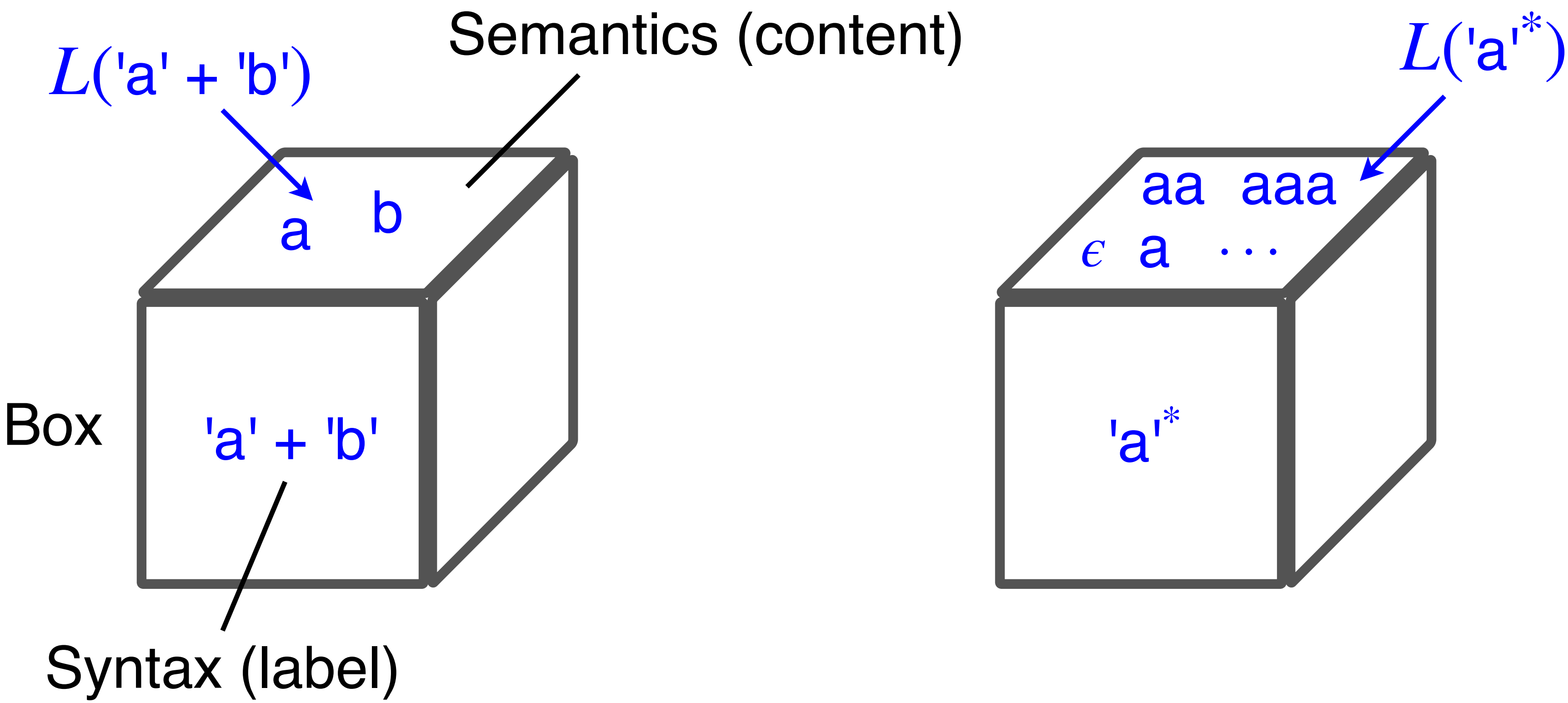
- Notation so far was imprecise

$$AB = \left\{ ab \mid a \in A \text{ and } b \in B \right\}$$

B as a piece of syntax

B as a set
(the semantics of the syntax)

Syntax vs. Semantics



Syntax vs. Semantics

- To be careful, we distinguish syntax and semantics.

$$L(\varepsilon) = \{""\}$$

$$L('c') = \{"c"\}$$

$$L(A+B) = L(A) \cup L(B)$$

$$L(AB) = \{ab \mid a \in L(A) \text{ and } b \in L(B)\}$$

$$L(A^*) = \cup_{i \geq 0} L(A^i)$$

Segue

- Regular expressions are simple, almost trivial
  - But they are useful!
- We will describe tokens in regular expressions

Example: Keyword

Keyword: “else” or “if” or “begin” or …

‘else’ + ‘if’ + ‘begin’ + . . .

Abbreviation: ‘else’ = ‘e’ ‘l’ ‘s’ ‘e’

Example: Integers

Integer: a non-empty string of digits

$$\text{digit} = \text{'0'} + \text{'1'} + \text{'2'} + \text{'3'} + \text{'4'} + \text{'5'} + \text{'6'} + \text{'7'} + \text{'8'} + \text{'9'}$$

$$\text{integer} = \text{digit digit}^*$$

Abbreviation: $A^+ = AA^*$

Abbreviation: $[0\text{-}2] = \text{'0'} + \text{'1'} + \text{'2'}$

Example: Identifier

Identifier: strings of letters or digits, starting with a letter

letter = 'A' + . . . + 'Z' + 'a' + . . . + 'z'

identifier = letter (letter + digit)*

Is (letter* + digit*) the same as (letter + digit)*?

Example: Whitespace

Whitespace: a non-empty sequence of blanks, newlines, and tabs

$$(' \ ' + '\backslash n' + '\backslash t')^+$$

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (650)-723-3232

$$\Sigma = \text{digits} \cup \{-,(,)\}$$

$$\text{exchange} = \text{digit}^3$$

$$\text{phone} = \text{digit}^4$$

$$\text{area} = \text{digit}^3$$

$$\text{phone\_number} = \text{'('}\ \text{area}\ \text{')-'}\ \text{exchange}\ \text{'-'}\ \text{phone}$$

Example: Email Addresses

- Consider anyone@cs.stanford.edu

$$\Sigma = \text{letters} \cup \{.,@\}$$

$$\text{name} = \text{letter}^{+}$$

$$\text{address} = \text{name '@' name '.' name '.' name}$$

Example: Unsigned Pascal Numbers

digit	=	'0' +'1'+'2'+'3'+'4'+'5'+'6'+'7'+'8'+'9'
digits	=	digit^+
opt_fraction	=	('.' digits) + ε
opt_exponent	=	('E' ('+' + '-' + ε) digits) + ε
num	=	digits opt_fraction opt_exponent

Other Examples

- File names
- Grep tool family

Summary

- Regular expressions describe many useful languages
  - We will look at non-regular languages next week
- Regular languages are a language specification
  - We still need an implementation
- Next time: Given a string s and a rexp R, is

$$s \in L(R)?$$