

Lexical Analysis

Instructor: Ramzan Shahid Khan
Namal University Mianwali

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 Scanner?

- Output of lexical analysis is a stream of tokens
- Input is just a sequence of characters
- Goal:
 - Partition input string into substrings:
 - Classify them according to their role

What's a Token?

- A token is a syntactic category
 - In English:
 - noun, verb, adjective, ...
 - e.g. “He wrote the program”
 - Words: “He”, “Wrote”, “the”, “Program”
 - In a programming language:
 - Identifier, Integer, Keyword, Whitespace, ...
 - e.g. “if(b==0) a=b”
 - Words: “if”, “(”, “b”, “==”, “0”, “a”, “=”, “b”

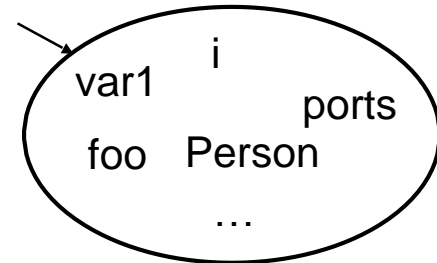
Tokens

- A token class corresponds to a set of strings

Infinite set

- 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\ttz = 0;\n\telse\n\t\ttz = 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\ttz = 0;\n\telse\n\t\ttz = 1;
```

Lexical Analyzer: Implementation

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

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

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 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

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

Regular Expressions

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

ε

$'c'$ where $c \in \Sigma$

$A + B$ where A, B are rexp over Σ

AB " " "

A^* where A is a rexp over Σ

Syntax vs. Semantics

- To be careful, we distinguish syntax and semantics.

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

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

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

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

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

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

digit = '0' + '1' + '2' + '3' + '4' + '5' + '6' + '7' + '8' + '9'

integer = digit digit*

Abbreviation: $A^+ = AA$

Abbreviation: $[0-2] = '0' + '1' + '2'$

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)*?