

# Lexical Analysis

## Part-1: Specification

# Outline

- Informal sketch of lexical analysis
  - Identifies tokens in input string
- Issues in lexical analysis
  - Lookahead
  - Ambiguities
- Specifying lexers
  - Regular expressions
  - Examples of regular expressions

↙ the lexical analyzer called  
→ like for ex. specifying tokens.

For that  
we need

# **1. Lexical Analysis**

2. Parsing

3. Semantic Analysis

4. Optimization

5. Code Generation

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

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

# What's a Token?

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

# Tokens

- Tokens correspond to sets of strings.
- e.g. 
  - 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 (e.g., identifier, keyword, whitespace, ...)
- Output of lexical analysis is a stream of tokens
  - ...
  - ... which is input to the <sup>(syntax analyzer)</sup>parser
  - Parser relies on token distinctions
    - An identifier is treated differently than a keyword

# Example

- Input:

X1=5

- Output

<identifier,"x1">, <op,"=">, <int,"5">

↑  
Output of lexical analyzer, which is stream of tokens

- Each pair is called a token
- Token format: <class, string>
- Or <token class, lexeme>

\* Usually when token is mentioned, it refers to the token class. However, you will know if it refers to the actual token (the pair) or to the token class from the context.



(Answering What? question)

# Designing a Lexical Analyzer: Step 1

specification

specification

- Define a finite set of tokens
  - Tokens describe all items of interest
  - 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, (, ), =, ;

- Note that (, ), =, ; are tokens, not characters, here

# 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

(Answering How? question)

# Lexical Analyzer: Implementation

- An implementation must do two things:
  1. Recognize substrings corresponding to tokens
  2. Return the value or lexeme of the token
    - The lexeme is the substring

we can do  
Finite Automata  
& regular expressions  
for this.

- Lexical analysis is not as easy as it sounds
- For example in FORTRAN Whitespace is insignificant
- E.g., VAR1 is the same as VA R1

Also

key word ← DO   label ↑ 5   identifier → 1,25   int int (loop)

– DO 5 I = 1,25   (is an assignment statement)

DO 5 I = 1.25

identifier ↑   assignment operator ↓   float number ↓

\* Here the lexical analyzer need to do a lookahead for the comma to check if it's a loop or not. Because of the assumption of whitespaces.

# 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.
- FORTRAN was designed this terrible way because on punch cards machines it was easy to add whitespaces by mistake.

- 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
- Variables
- keywords



# Lexical Analysis in PL/I (Cont.)

- PL/I Declarations:
- DECLARE (ARG1,.. . ., ARGN)
- Can't tell whether DECLARE is a keyword or array reference until after the ) to see if there is = for example.
- Requires arbitrary lookahead! Because we have n args. → unbounded lookahead

- FORTRAN was designed in 1950's
- PL/I was designed in 1960's
- Things are not that bad with modern languages

- But the problems have not gone away completely.

- C++ template syntax: <sup>like Generic in java</sup>

Foo<Bar>

- C++ stream syntax:

cin(>>) var;

- But there is a conflict with nested templates:

Foo<Bar<Bazz>>

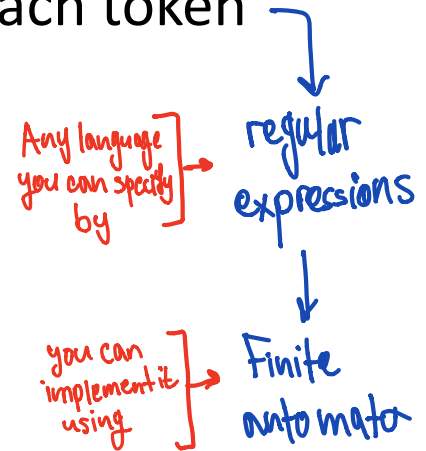
- For along time C++ compilers generated a syntax error
- The only solution was to put a space between the last >  
>

# Review

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

# Next

- We still need
  - A way to describe the lexemes of each token
  - A way to resolve ambiguities
    - Is if two variables l and f?
    - Is == two equal signs =?



# Regular Languages

→ Def.: It's the language that we can specify using a regular expression.

- There are several formalisms for specifying tokens
- Regular languages are the most popular
  - Simple and useful theory
  - Easy to understand
  - Efficient implementations

# Languages

also could  
be notate  
by:  $\Sigma$   
↑

called alphabet

formal  
↑

- Def. Let  $S$  be a set of characters. A language over  $S$  is a set of strings of characters drawn from  $S$
- **Languages are sets of strings.**
- Need **some notation** for specifying which sets we want
- The standard notation for **regular languages** is regular expressions.

# Regular Expressions

- Atomic Regular Expressions
  - Single character

$\text{'c'} = \{\text{'c'}\}$   $\xrightarrow{\text{reg. lang. that derived from reg. exp.}}$   
 $\text{reg. exp.} \rightarrow \text{Epsilon}$

$\epsilon = \{\text{' '}\} \neq \{\}$   
 $\uparrow$   
empty language



# Atomic 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

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

- Def. The regular expressions over  $S$  are the smallest set of expressions including

$\varepsilon$

' $c$ '      where  $c \in \Sigma$

$A + B$     where  $A, B$  are rexp over  $\Sigma$

$AB$       "                      "                      "

$A^*$       where  $A$  is a rexp over  $\Sigma$

# Examples

- $\Sigma = \{0,1\}$
- $1^* = "" + 1 + 11 + 111 + \dots$
- $(1+0)1 = \{ab \mid a \in 1+0 \wedge b \in 1\} = \{11, 01\}$
- $0^* + 1^* = \{0^i \mid i \geq 0\} \cup \{1^i \mid i \geq 0\}$
- $(0+1)^* = \bigcup_{i \geq 0} (0+1)^i = "" + 0+1, (0+1)(0+1), \dots, (0+1)\dots(0+1)$   
= all strings of 0's and 1's

# Syntax vs. Semantics

- To be careful, we should distinguish syntax (the reg. exp.) and semantics (the langs. they denote).
- Meaning function  $L$  maps syntax to semantics
- $L: \text{Exp} \rightarrow \text{Sets of Strings}$

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

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

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

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

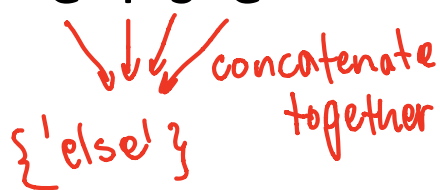
$$L(A^*) = \bigcup_{i \geq 0} L(A^i)$$

- Regular expressions are simple, almost trivial
  - But they are useful!
- Reconsider informal token descriptions . . .

# Example: Keyword

- Keyword: “else” or “if” or “begin” or ...  
‘else’ + ‘if’ + ‘begin’ + ...

Note: ‘else’ abbreviates

‘e’|’s’  
concatenate  
together  
{ else }

# Example: Integers

*Integer: a non-empty string of digits*

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

integer = digit digit<sup>\*</sup>

Abbreviation:  $A^+ = AA^*$

# 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? **NO**



# Example: Whitespace

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

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

# Example: Email Addresses

- Consider [anyone@cs.stanford.edu](mailto:anyone@cs.stanford.edu)

letter<sup>+</sup> '@' letter<sup>+</sup> '.' letter<sup>+</sup> '.' letter<sup>+</sup>

or

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

name = letter<sup>+</sup> (letter + digit)\*

address = name '@' name '.' name '.' name

# Example: Phone Numbers

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

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

exchange = digit<sup>3</sup>

phone = digit<sup>4</sup>

area = digit<sup>3</sup>

phone\_number = '(' area ')' '-' exchange '-' phone

optional



# Example: Unsigned Pascal Numbers

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

digits = digit<sup>+</sup>

opt\_fraction = ('.' digits) +  $\xi$   $\equiv$  ( '.' digits ) (?)   
 *means that all that is optional*

opt\_exponent = ('E' ('+' + '-' +  $\xi$ ) digits) +  $\xi$    
  $\equiv$  ( 'E' ( '+' + '-' ) (?) digits ) (?)

num = digits opt\_fraction opt\_exponent

# Summary

- Regular expressions describe many useful languages
- Regular languages are a language specification
  - We still need an implementation
- We still need to be able to decide given a string  $s$  and a reg. exp.  $R$ , is
$$s \in L(R) ?$$