

Lexical Analysis Part I

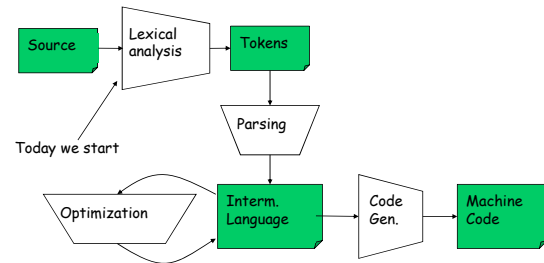
Chapter 3

1/12/2009

Chapter 3, CS 5810 Spring 2009

1

The Structure of a Compiler



1/12/2009

Chapter 3, CS 5810 Spring 2009

2

Interactions Between the Lexer & Parser



1/12/2009

Chapter 3, CS 5810 Spring 2009

3

Lexical Analysis

- What do we want to do? Example:


```
if (i == j)
    z = 0;
else
    z = 1;
```
- The input is just a sequence of characters:


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

1/12/2009

Chapter 3, CS 5810 Spring 2009

4

Goal of Lexical Analysis

- Goal: Partition input string into substrings
 - And classify them according to their role -> token
 - Reduce length of program representation (remove spaces)

1/12/2009

Chapter 3, CS 5810 Spring 2009

5

What's a Token?

- Output of lexical analysis is a stream of *useful* tokens
- A token is a syntactic category
 - In English:
 - noun, verb, adjective, ...
 - In a programming language:
 - Identifier, Integer, IF, THEN, Whitespace, ...
- Parser relies on the token distinctions:
 - E.g., identifiers are treated differently than keywords

1/12/2009

Chapter 3, CS 5810 Spring 2009

6

Tokens

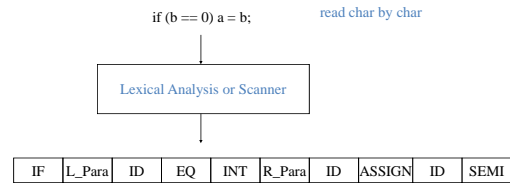
- Token represents a category. It consists of a token name + optional value
- Identifier: *strings of letters or digits, starting with a letter*
- Integer: *a non-empty string of digits*
- Keyword: *"else" or "if" or "for" or ...*
 - IF: *string "if"*
- Whitespace: *a non-empty sequence of blanks, newlines, and tabs*

1/12/2009

Chapter 3, CS 5810 Spring 2009

7

Lexical Analysis Process



1/12/2009

Chapter 3, CS 5810 Spring 2009

8

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

1/12/2009

Chapter 3, CS 5810 Spring 2009

9

Example

- Recall:


```
\tif (i == j)\n\t\tz = 0;\n\telse\n\t\tz = 1;
```
- Token-lexeme pairs returned by the lexer:
 - (Whitespace, "\t")
 - (IF, "if")
 - (OpenPar, "(")
 - (Identifier, "i")
 - (Relation, "==")
 - (Identifier, "j")
 - ...

1/12/2009

Chapter 3, CS 5810 Spring 2009

10

Lexical Analyzer: Implementation

- The lexer usually discards "uninteresting" tokens that don't contribute to parsing.
- Examples: Whitespace, Comments

1/12/2009

Chapter 3, CS 5810 Spring 2009

11

Lookahead.

- 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
 - Even our simple example has lookahead issues


```
i vs. if
              = vs. ==
```

1/12/2009

Chapter 3, CS 5810 Spring 2009

12

Next

- We need
 - A language to describe the lexemes of each token
 - A language to resolve ambiguities
 - Is `if` two variables `i` and `f`?
 - Is `==` two equal signs `=` `=`?

1/12/2009

Chapter 3, CS 5810 Spring 2009

13

Regular Language

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

1/12/2009

Chapter 3, CS 5810 Spring 2009

14

Languages

Def. Let Σ be a set of characters. A *language over Σ* is a set of strings of characters drawn from Σ (Σ is called the *alphabet*)

1/12/2009

Chapter 3, CS 5810 Spring 2009

15

Examples of Languages

- Alphabet = English characters
- Language = English sentences
- Not every string on English characters is an English sentence
- Alphabet = ASCII
- Language = C programs
- Note: ASCII character set is different from English character set

1/12/2009

Chapter 3, CS 5810 Spring 2009

16

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- For lexical analysis we care about *regular languages*, which can be described using *regular expressions*.
 - a regular language is used to describe each token in a programming language

1/12/2009

Chapter 3, CS 5810 Spring 2009

17

Regular Expressions and Regular Languages

- Each regular expression is a notation for a regular language (a set of words)
- If A is a regular expression then we write $L(A)$ to refer to the language denoted by A

1/12/2009

Chapter 3, CS 5810 Spring 2009

18

Atomic Regular Expressions

- Single character: $'c'$
 $L('c') = \{ "c" \}$ (for any $c \in \Sigma$)
- Concatenation: AB (where A and B are reg. exp.)
 $L(AB) = \{ ab \mid a \in L(A) \text{ and } b \in L(B) \}$
 – Example: $'i'$ and $'f'$ are reg. expressions so is $'if'$.
 Therefore the language is
 $L('i' 'f') = \{ "if" \}$
 (we will abbreviate $'i' 'f'$ as $'if'$)

1/12/2009

Chapter 3, CS 5810 Spring 2009

19

Compound Regular Expressions

- Union: $A|B$ is a reg. expression where A, B are reg. expression. Therefore the language is
 $L(A|B) = \{ s \mid s \in L(A) \text{ or } s \in L(B) \}$
- 1. Examples:
 $'if' \mid 'then' \mid 'else' = \{ "if", "then", "else" \}$
 $'0' \mid '1' \mid \dots \mid '9' = \{ "0", "1", \dots, "9" \}$
 (note the ... are just an abbreviation)
- 2. Another example:
 $('0' \mid '1') ('0' \mid '1') = \{ "00", "01", "10", "11" \}$

1/12/2009

Chapter 3, CS 5810 Spring 2009

20

More Compound Regular Expressions

- So far we do not have a notation for infinite languages
- Iteration: A^* is a reg expression if A is a reg. expression. Therefore the language is
 $L(A^*) = \{ "" \mid L(A) \mid L(AA) \mid L(AAA) \mid \dots \}$
- Examples:
 $'0'^* = \{ "", "0", "00", "000", \dots \}$
 $'1' '0'^* = \{ \text{strings starting with 1 and followed by 0's} \}$
- Epsilon: ϵ
 $L(\epsilon) = \{ "" \}$

1/12/2009

Chapter 3, CS 5810 Spring 2009

21

Example: Keyword

– Keyword: *"else" or "if" or "begin" or ...*

'else' | 'if' | 'begin' | ...

(Recall: *'else'* abbreviates *'e' 'l' 's' 'e'*)

1/12/2009

Chapter 3, CS 5810 Spring 2009

22

Example: Integers

Integer: *a non-empty string of digits*

digit = $'0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9'$

number = digit digit*

Abbreviation: $A^+ = A A^*$

1/12/2009

Chapter 3, CS 5810 Spring 2009

23

Example: Identifier

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

- letter = $'A' \mid \dots \mid 'Z' \mid 'a' \mid \dots \mid 'z'$
 identifier = letter (letter | digit)*
- Is $\text{letter}(\text{letter}^* \mid \text{digit}^*)$ the same ?

1/12/2009

Chapter 3, CS 5810 Spring 2009

24

Example: Whitespace

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

- `(' | \t | \n) +`

1/12/2009

Chapter 3, CS 5810 Spring 2009

25

Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510) 643-1481
 - $\Sigma = \{ 0, 1, 2, 3, \dots, 9, (,), - \}$
 - area = `digit3`
 - exchange = `digit3`
 - phone = `digit4`
 - number = `'(' area ')' exchange '-' phone`

1/12/2009

Chapter 3, CS 5810 Spring 2009

26

Example: Email Addresses

- Consider zijing@cs.wmich.edu

$\Sigma = \text{letters } \cup \{ ., @ \}$
 name = `letter+`
 address = `name '@' name ('.' name)*`

1/12/2009

Chapter 3, CS 5810 Spring 2009

27

Applications of regular expression

- In Windows
 - In windows you can use RE to search for files or texts in a file
- In unix, there are many RE relevant tools, such as Grep
 - Stands for Global Regular Expressions and Print (or Global Regular Expression and Parser ...);
 - Useful UNIX command to find patterns of characters in a text file;



1/19/2009

28

Practice regular expression using grep

Use `grep` to search for certain pattern in html files;

- Search for Canadian zip code in a text file;
- use `tcsh`. Type
 - `% tcsh`
- Prepare text file, say "test", that consists of sample postal code etc.
- Type
 - `grep '[a-z][0-9][a-z] [0-9][a-z][0-9]' test`
 - `grep -i '[a-z][0-9][a-z] [0-9][a-z][0-9]' test`

1/19/2009

Chapter 3, CS 5810 Spring 2009

Practice the following grep commands:

- `grep 'cat' grepTest`
- --you will find both "cat" and "vacation"
- `grep '^cat' grepTest`
- --find only lines start with cat
- `grep '\ccat\>' grepTest`
- --word boundary
- `grep -i '\ccat\>' grepTest`
- --ignore the case
- `grep '\cega@att.com\>' grepTest`
- --meta character
- `grep '[^*]*' grepTest`
- --find quoted string
- `egrep '[a-z][0-9][a-z]?[0-9][a-z][0-9]' grepTest`
- --find Canadian postal code, only if it is in small case
- `egrep -i '[a-z][0-9][a-z]?[0-9][a-z][0-9]' grepTest`
- --ignore the case
- --`egrep` is similar to `grep`, but somehow some patterns only work in `egrep`.

1/19/2009

Chapter 3, CS 5810 Spring 2009

Summary

- Regular expressions describe many useful languages
- Next: Given a string s and a rexp R , is

$$s \in L(R)?$$
- But a yes/no answer is not enough !
- Instead: partition the input into lexemes
- We will adapt regular expressions to this goal

1/12/2009

Chapter 3, CS 5810 Spring 2009

31

Procedure of Lexical Analysis

- Specifying lexical structure using regular expressions
- Finite automata
 - Deterministic Finite Automata (DFAs)
 - Non-deterministic Finite Automata (NFAs)
- Implementation of regular expressions

$$\text{RegExp} \Rightarrow \text{NFA} \Rightarrow \text{DFA} \Rightarrow \text{Tables}$$

1/12/2009

Chapter 3, CS 5810 Spring 2009

32

Regular Expressions \Rightarrow Lexical Spec.

(1)

1. Select a set of tokens
 - Number, Keyword, Identifier, ...
2. Write a R.E. for the lexemes of each token
 - Number = digit^+
 - Keyword = $\text{'if' | 'else' | ...}$
 - Identifier = $\text{letter (letter | digit)^*}$
 - OpenPar = '('
 - ...

1/12/2009

Chapter 3, CS 5810 Spring 2009

33

Regular Expressions \Rightarrow Lexical Spec.

(2)

3. Construct R , matching all lexemes for all tokens

$$R = R_1 \quad | \quad R_2 \quad | \quad R_3 \quad | \quad \dots$$

Facts: If $s \in L(R)$ then s is a lexeme

- Furthermore $s \in L(R_i)$ for some " i "
- This " i " determines the token that is reported

1/12/2009

Chapter 3, CS 5810 Spring 2009

34

Regular Expressions \Rightarrow Lexical Spec.

(3)

4. Let the input be $x_1 \dots x_n$

($x_1 \dots x_n$ are characters in the language alphabet)

 - For $1 \leq i \leq n$ check

$$x_1 \dots x_i \in L(R)?$$
5. It must be that

$$x_1 \dots x_i \in L(R_j) \text{ for some } i \text{ and } j$$
6. Remove $x_1 \dots x_i$ from input and go to (4)

1/12/2009

Chapter 3, CS 5810 Spring 2009

35

Lexing Example

$$R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid \text{'+'}$$

- Scan " $f + 3 + g$ "
 - " f " matches R , more precisely Identifier
 - " $+$ " matches R , more precisely '+'
 - ...
 - The token-lexeme pairs are
 - $(\text{Identifier}, "f")$, $(\text{'+'}, "+")$, $(\text{Integer}, "3")$
 - $(\text{Whitespace}, " ")$, $(\text{'+'}, "+")$, $(\text{Identifier}, "g")$
- We would like to drop the Whitespace tokens
 - after matching Whitespace , continue matching

1/12/2009

Chapter 3, CS 5810 Spring 2009

36

Ambiguities (1)

- There are ambiguities in the algorithm
- Example:
 $R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid '+'$
- Parse "foo+3"
 - "f" matches R, more precisely Identifier
 - But also "fo" matches R, and "foo", but not "foo+"
- How much input is used? What if
 - $x_1 \dots x_i \in L(R)$ and also $x_1 \dots x_k \in L(R)$
 - "Maximal munch" rule: Pick the longest possible substring that matches R

1/12/2009

Chapter 3, CS 5810 Spring 2009

37

More Ambiguities

$R = \text{Whitespace} \mid \text{'new'} \mid \text{Integer} \mid \text{Identifier}$

- Parse "new foo"
 - "new" matches R, more precisely 'new'
 - but also Identifier, which one do we pick?
- In general, if $x_1 \dots x_i \in L(R_j)$ and $x_1 \dots x_i \in L(R_k)$
 - Rule: use rule listed first (j if $j < k$)
- We must list 'new' before Identifier

1/12/2009

Chapter 3, CS 5810 Spring 2009

38

Error Handling

$R = \text{Whitespace} \mid \text{Integer} \mid \text{Identifier} \mid '+'$

- Parse "=56"
 - No prefix matches R: not "=", nor "=5", nor "=56"
- Problem: Can't just get stuck ...
- Solution:
 - Add a rule matching all "bad" strings; and put it last
- Lexer tools allow the writing of:
 $R = R_1 \mid \dots \mid R_n \mid \text{Error}$
 - Token Error matches if nothing else matches

1/12/2009

Chapter 3, CS 5810 Spring 2009

39

Summary

- Regular expressions provide a concise notation for string patterns
- Use in lexical analysis requires small extensions
 - To resolve ambiguities
 - To handle errors
- Good algorithms known (next)
 - Require only single pass over the input
 - Few operations per character (table lookup)

1/12/2009

Chapter 3, CS 5810 Spring 2009

40