

## Lexical Analysis

### Lecture 3-4

Prof. Necula CS 164 Lecture 3

1

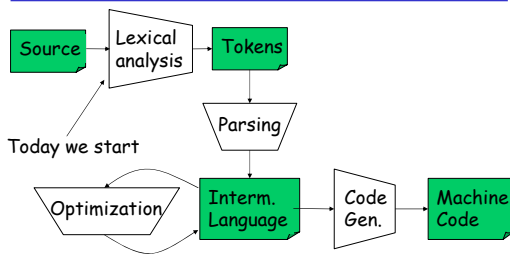
## Outline

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

Prof. Necula CS 164 Lecture 3

2

## Recall: The Structure of a Compiler



Prof. Necula CS 164 Lecture 3

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;
```
- Goal: Partition input string into substrings
  - And classify them according to their role

Prof. Necula CS 164 Lecture 3

4

## What's a Token?

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

Prof. Necula CS 164 Lecture 3

5

## Tokens

- Tokens correspond to sets of strings.
- 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*
- OpenPar: *a left-parenthesis*

Prof. Necula CS 164 Lecture 3

6

## 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

Prof. Necula CS 164 Lecture 3

7

## Example

---

- Recall:

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

Prof. Necula CS 164 Lecture 3

8

## Lexical Analyzer: Implementation

---

- The lexer usually discards "uninteresting" tokens that don't contribute to parsing.
- Examples: Whitespace, Comments
- Question: What happens if we remove all whitespace and all comments prior to lexing?

Prof. Necula CS 164 Lecture 3

9

## 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. ==
```

Prof. Necula CS 164 Lecture 3

10

## Next

---

- We 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 `=` `=`?

Prof. Necula CS 164 Lecture 3

11

## Regular Languages

---

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

Prof. Necula CS 164 Lecture 3

12

## Languages

---

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

Prof. Necula CS 164 Lecture 3

13

## 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

Prof. Necula CS 164 Lecture 3

14

## 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*.

Prof. Necula CS 164 Lecture 3

15

## 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  $A$

Prof. Necula CS 164 Lecture 3

16

## 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:  $L('i' 'f') = \{ "if" \}$   
(we will abbreviate 'i' 'f' as 'if')

Prof. Necula CS 164 Lecture 3

17

## Compound Regular Expressions

---

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

Prof. Necula CS 164 Lecture 3

18

## More Compound Regular Expressions

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

Prof. Necula CS 164 Lecture 3

19

## Example: Keyword

- Keyword: "else" or "if" or "begin" or ...

$'else' \mid 'if' \mid 'begin' \mid \dots$

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

Prof. Necula CS 164 Lecture 3

20

## Example: Integers

Integer: *a non-empty string of digits*

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

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

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

Prof. Necula CS 164 Lecture 3

21

## Example: Identifier

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

$\text{letter} = 'A' \mid \dots \mid 'Z' \mid 'a' \mid \dots \mid 'z'$

$\text{identifier} = \text{letter} (\text{letter} \mid \text{digit})^*$

Is  $(\text{letter}^* \mid \text{digit}^*)$  the same as

$(\text{letter} \mid \text{digit})^* ?$

Prof. Necula CS 164 Lecture 3

22

## Example: Whitespace

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

$( ' ' \mid '\t' \mid '\n' )^*$

(Can you spot a subtle omission?)

Prof. Necula CS 164 Lecture 3

23

## Example: Phone Numbers

- Regular expressions are all around you!
- Consider (510) 643-1481  
 $\Sigma = \{ 0, 1, 2, 3, \dots, 9, (, ), - \}$   
 $\text{area} = \text{digit}^3$   
 $\text{exchange} = \text{digit}^3$   
 $\text{phone} = \text{digit}^4$   
 $\text{number} = '(' \text{ area } ')' \text{ exchange } '-' \text{ phone}$

Prof. Necula CS 164 Lecture 3

24

## Example: Email Addresses

- Consider [necula@cs.berkeley.edu](mailto:necula@cs.berkeley.edu)

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

name = letter<sup>+</sup>

address = name '@' name ('.' name)\*

Prof. Necula CS 164 Lecture 3

25

## 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

Prof. Necula CS 164 Lecture 3

26

## Next: Outline

- Specifying lexical structure using regular expressions

- Finite automata

- Deterministic Finite Automata (DFAs)
- Non-deterministic Finite Automata (NFAs)

- Implementation of regular expressions

RegExp  $\Rightarrow$  NFA  $\Rightarrow$  DFA  $\Rightarrow$  Tables

Prof. Necula CS 164 Lecture 3

27

## 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<sup>+</sup>
- Keyword = 'if' | 'else' | ...
- Identifier = letter (letter | digit)\*
- OpenPar = '('
- ...

Prof. Necula CS 164 Lecture 3

28

## Regular Expressions $\Rightarrow$ Lexical Spec. (2)

3. Construct  $R$ , matching all lexemes for all tokens

$R = \text{Keyword} \mid \text{Identifier} \mid \text{Number} \mid \dots$   
 $= R_1 \mid R_2 \mid R_3 \mid \dots$

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

Prof. Necula CS 164 Lecture 3

29

## 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

$x_1 \dots x_i \in L(R_j)$  for some  $i$  and  $j$

6. Remove  $x_1 \dots x_i$  from input and go to (4)

Prof. Necula CS 164 Lecture 3

30

## Lexing Example

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

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

Prof. Necula CS 164 Lecture 3

31

## 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  $\text{Identifier}$
  - But also "fo" matches  $R$ , and "foo", but not "foo+3"
- 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  $R$

Prof. Necula CS 164 Lecture 3

32

## 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  $\text{Identifier}$ , 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  $\text{Identifier}$

Prof. Necula CS 164 Lecture 3

33

## 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  $\text{Error}$  matches if nothing else matches

Prof. Necula CS 164 Lecture 3

34

## 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)

Prof. Necula CS 164 Lecture 3

35

## Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- A finite automaton consists of
  - An input alphabet  $\Sigma$
  - A set of states  $S$
  - A start state  $n$
  - A set of accepting states  $F \subseteq S$
  - A set of transitions  $\text{state} \xrightarrow{\text{input}} \text{state}$

Prof. Necula CS 164 Lecture 3

36

## Finite Automata

- Transition




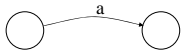
$$s_1 \xrightarrow{a} s_2$$

- Is read  
In state  $s_1$  on input "a" go to state  $s_2$
- If end of input
  - If in accepting state  $\Rightarrow$  accept, otherwise  $\Rightarrow$  reject
- If no transition possible  $\Rightarrow$  reject

Prof. Necla CS 164 Lecture 3

37

## Finite Automata State Graphs

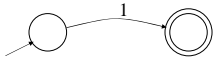
- A state 
- The start state 
- An accepting state 
- A transition 

Prof. Necla CS 164 Lecture 3

38

## A Simple Example

- A finite automaton that accepts only "1"



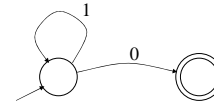
- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Prof. Necla CS 164 Lecture 3

39

## Another Simple Example

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



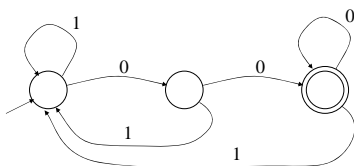
- Check that "1110" is accepted but "110..." is not

Prof. Necla CS 164 Lecture 3

40

## And Another Example

- Alphabet {0,1}
- What language does this recognize?

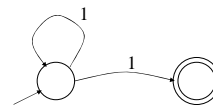


Prof. Necla CS 164 Lecture 3

41

## And Another Example

- Alphabet still {0, 1}



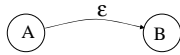
- The operation of the automaton is not completely defined by the input
  - On input "11" the automaton could be in either state

Prof. Necla CS 164 Lecture 3

42

## Epsilon Moves

- Another kind of transition:  $\epsilon$ -moves



- Machine can move from state A to state B without reading input

Prof. Necula CS 164 Lecture 3

43

## Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No  $\epsilon$ -moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have  $\epsilon$ -moves
- Finite automata have finite memory
  - Need only to encode the current state

Prof. Necula CS 164 Lecture 3

44

## Execution of Finite Automata

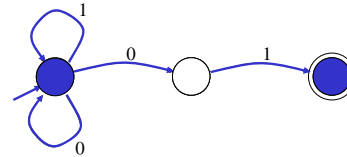
- A DFA can take only one path through the state graph
  - Completely determined by input
- NFAs can choose
  - Whether to make  $\epsilon$ -moves
  - Which of multiple transitions for a single input to take

Prof. Necula CS 164 Lecture 3

45

## Acceptance of NFAs

- An NFA can get into multiple states



- Input: 1 0 1
- Rule: NFA accepts if it can get in a final state

Prof. Necula CS 164 Lecture 3

46

## NFA vs. DFA (1)

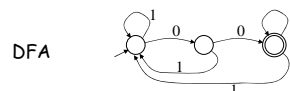
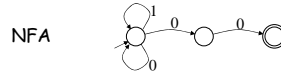
- NFAs and DFAs recognize the same set of languages (regular languages)
- DFAs are easier to implement
  - There are no choices to consider

Prof. Necula CS 164 Lecture 3

47

## NFA vs. DFA (2)

- For a given language the NFA can be simpler than the DFA



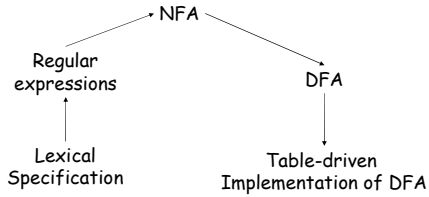
- DFA can be exponentially larger than NFA

Prof. Necula CS 164 Lecture 3

48

## Regular Expressions to Finite Automata

- High-level sketch



Prof. Necla CS 164 Lecture 3

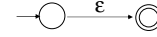
49

## Regular Expressions to NFA (1)

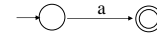
- For each kind of rexp, define an NFA
  - Notation: NFA for rexp  $A$



- For  $\epsilon$



- For input  $a$

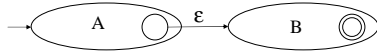


Prof. Necla CS 164 Lecture 3

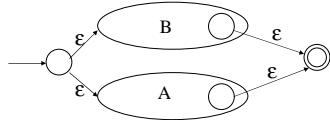
50

## Regular Expressions to NFA (2)

- For  $AB$



- For  $A | B$

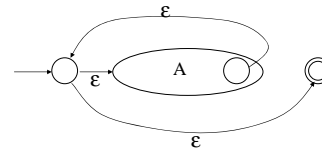


Prof. Necla CS 164 Lecture 3

51

## Regular Expressions to NFA (3)

- For  $A^*$

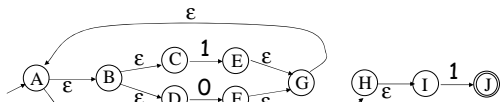


Prof. Necla CS 164 Lecture 3

52

## Example of RegExp -> NFA conversion

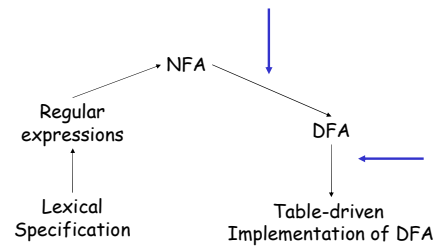
- Consider the regular expression  $(1 | 0)^*1$
- The NFA is



Prof. Necla CS 164 Lecture 3

53

## Next



Prof. Necla CS 164 Lecture 3

54

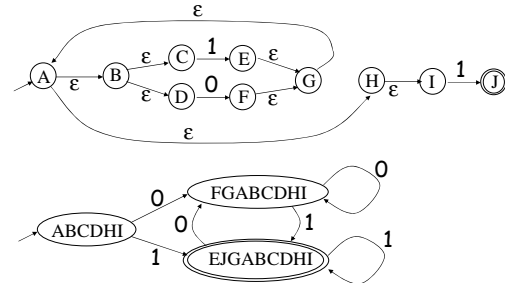
## NFA to DFA. The Trick

- Simulate the NFA
- Each state of resulting DFA
  - = a non-empty subset of states of the NFA
- Start state
  - = the set of NFA states reachable through  $\epsilon$ -moves from NFA start state
- Add a transition  $S \xrightarrow{a} S'$  to DFA iff
  - $S'$  is the set of NFA states reachable from the states in  $S$  after seeing the input  $a$ 
    - considering  $\epsilon$ -moves as well

Prof. Necula CS 164 Lecture 3

55

## NFA -> DFA Example



Prof. Necula CS 164 Lecture 3

56

## NFA to DFA. Remark

- An NFA may be in many states at any time
- How many different states ?
- If there are  $N$  states, the NFA must be in some subset of those  $N$  states
- How many non-empty subsets are there?
  - $2^N - 1$  = finitely many, but exponentially many

Prof. Necula CS 164 Lecture 3

57

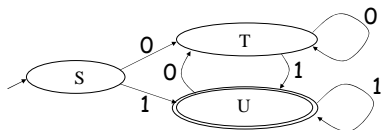
## Implementation

- A DFA can be implemented by a 2D table  $T$ 
  - One dimension is "states"
  - Other dimension is "input symbols"
  - For every transition  $S_i \xrightarrow{a} S_k$  define  $T[i,a] = k$
- DFA "execution"
  - If in state  $S_i$  and input  $a$ , read  $T[i,a] = k$  and skip to state  $S_k$
  - Very efficient

Prof. Necula CS 164 Lecture 3

58

## Table Implementation of a DFA



	0	1
S	T	U
T	T	U
U	T	U

Prof. Necula CS 164 Lecture 3

59

## Implementation (Cont.)

- NFA -> DFA conversion is at the heart of tools such as flex or jlex
- But, DFAs can be huge
- In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

Prof. Necula CS 164 Lecture 3

60

## PA2: Lexical Analysis

---

- Correctness is job #1.
  - And job #2 and #3!
- Tips on building large systems:
  - Keep it simple
  - Design systems that can be tested
  - Don't optimize prematurely
  - It is easier to modify a working system than to get a system working