CMSC 124

Design and Implementation of Programming Languages

Kristine Bernadette Pelaez
Institute of Computer Science
University of the Philippines Los Baños

Lexical Analysis

done by the
lexical analyzer (scanner)

groups the sequence
of *characters into lexemes*

Lexeme

smallest syntactic unit

the lexical analyzer also
identifies the type of each lexeme

Token

a data structure containing
the lexeme and its type


```
printf("Age is ", age);
```



```
printf("Age is ", age);
```

LEXEME	TYPE	LEXEME	TYPE
printf	function identifier	**,**	separator
(	delimiter_oparen	**age**	variable identifier
"	delimiter_str	**)**	delimiter_cparen
Age is	string literal	**;**	delimiter_end
"	delimiter_str		

Symbol Table

a data structure used to store tokens and the information about these tokens


```
printf("Age is ", age);
```

LEXEME	TYPE	VALUE	REFERENCING ENVIRONMENT
"	variable identifier		
,	separator		
age	variable identifier		
)	delimiter_cparen		
;	delimiter_scol		

the lexical analyzer finds each lexeme by using *pattern matching*

lexemes follow a specific pattern based on their type

the lexical analyzer will *match the patterns* for each type of lexeme *to a string of characters*

if a *string did not match* with any of the patterns, an *error will be produced*


```
 124
   2.2534
-100
  -5.55
    .2
   0.751
   -.0006
```



```
12-30-1995
18-02-1997
2015-05-26
07-18-93
96-08-23
```


Ways of Matching Strings

1. State-transition diagrams
2. Regular Expressions

1. State-transition Diagrams

1. State-transition diagrams

aka *state diagrams*;
they are used to give an abstract
description of the behavior of a system

1. State-transition diagrams

lexical analyzers use
the state diagram
called *finite automata*

1. State-transition diagrams

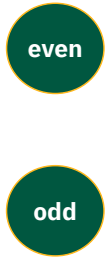
state diagrams are
just *directed graphs*

1. State-transition diagrams

nodes in a state diagram
are called *states*

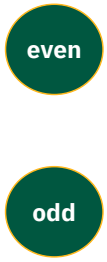
1. State-transition diagrams

usually represented
by a *circle containing
the name of the state*



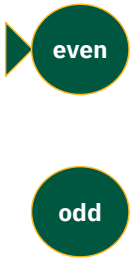
1. State-transition diagrams

one of the states
will be the *start state*
which will be *where
the pattern must start*



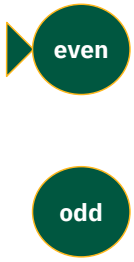
1. State-transition diagrams

the start state has
an *inverted triangle*
drawn on its side



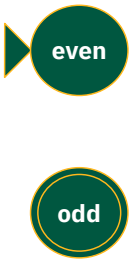
1. State-transition diagrams

some states can also
be a *final state*, which
are the states that
accepts the input string



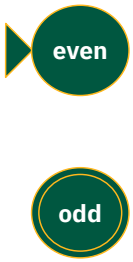
1. State-transition diagrams

they are drawn using
concentric circles



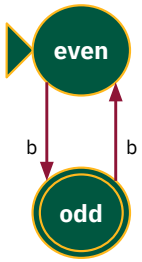
1. State-transition diagrams

arcs on state diagrams
are also called *transitions*



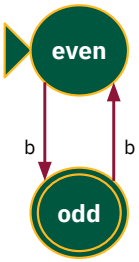
1. State-transition diagrams

drawn using *arrows*,
and must have
an *assigned symbol*



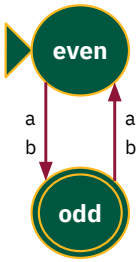
1. State-transition diagrams

the symbol assigned to a
transition is needed to
move from one state to another



1. State-transition diagrams

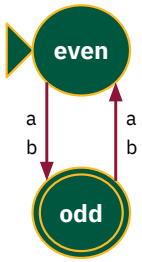
if *multiple transitions* exist
for the same source and
destination state, *all symbols*
can be written in the same arrow



1. State-transition diagrams

What does this
finite automata do?

What strings does it accept?



Example

```
 124
   2.2534
-100
  -5.55
    .2
   0.751
   -.0006
```

1. may start with a negative sign
2. may have a decimal point
3. if it has a decimal point, it must be followed by at least one digit
4. may have numbers at the left of the decimal point

2. Regular Expressions

2. Regular Expressions

aka *regex*;
set of *characters* that
formally defines a pattern

2. Regular Expressions

more info here!

SYMBOL	DESCRIPTION	EXAMPLE	SYMBOL	DESCRIPTION	EXAMPLE
^	string starts with	^car	{n}	occurs exactly n times	a{3}
$	string ends with	ry$	{n,}	occurs n times or more	a{2,}
|	or operator	pas(s|t)	{n,m}	occurs n-m times	a{2,3}
[]	one of these characters	[axy]	.	any one character	err.rs
+	one or more consecutive occurrences	bar+	\	escape character	u\.
*	zero or more consecutive occurrences	bar*	[^]	not any of these characters	[^a]
?	optional; may or may not occur in the string	t?rust	\d	digits	\d{3}

Example

```
 124
   2.2534
-100
  -5.55
    .2
   0.751
   -.0006
```

1. may start with a negative sign
2. may have a decimal point
3. if it has a decimal point, it must be followed by at least one digit
4. may have numbers at the left of the decimal point

Lexical Analysis

Which is better?
State-diagrams or regular expressions?

Lexical Analysis

They are equal in capabilities.

If you have a regex,
you can create a finite automaton for it.

If you have a finite automaton,
you can create a regex for it.

CMSC 124

Design and Implementation of Programming Languages

Kristine Bernadette Pelaez
Institute of Computer Science
University of the Philippines Los Baños