# Topic #12: Regular Expressions

CSE 413, Autumn 2004

Programming Languages

http://www.cs.washington.edu/education/courses/413/04au/



## Outline

- Basic concepts of formal grammars
- Regular expressions
- Lexical specification of programming languages
- Using finite automata to recognize regular expressions



## Programming Language Specifications

- Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  - » First done in 1959 with BNF (Backus-Naur Form or Backus-Normal Form) used to specify the syntax of ALGOL 60
  - » Borrowed from the linguistics community



## Grammar for a Tiny Language

*program* ::= *statement* | *program statement*

*statement* ::= *assignStmt* | *ifStmt*

*assignStmt* ::= *id* = *expr* ;

*ifStmt* ::= if ( *expr* ) *stmt*

*expr* ::= *id* | *int* | *expr* + *expr*

*id* ::= a | b | c | i | j | k | n | x | y | z

*int* ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



## Productions

- Meaning of

  *nonterminal* ::= <sequence of terminals and nonterminals>

  - » "In a derivation, the non-terminal on the left can be replaced by the expression on the right"
- Often, there are two or more productions for a single nonterminal – can use either at different times
- Alternative notations:

  *ifStmt* ::= **if (** *expr* **)** *stmt*

  *ifStmt* → **if (** *expr* **)** *stmt*

  <ifStmt> ::= **if (** <expr> **)** <stmt>



## Example Derivation

*program* ::= *statement* | *program statement*
*statement* ::= *assignStmt* | *ifStmt*
*assignStmt* ::= *id* = *expr* ;
*ifStmt* ::= if ( *expr* ) *stmt*
*expr* ::= *id* | *int* | *expr* + *expr*
*Id* ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

a = 1 ; b = 2 + c + 3 ;

## Parsing

- Parsing: reconstruct the derivation (syntactic structure) of a program
- In principle, a single recognizer could work directly from the concrete, character-by-character grammar
- Instead:


7

## Why Separate the Scanner and Parser?

- Simplicity & Separation of Concerns
  - » Scanner hides details from parser (comments, whitespace, input files, etc.)
  - » Parser is easier to build; has simpler input stream
- Efficiency
  - » Scanner can use simpler, faster design

8

## Principle of Longest Match

- In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- Example

```
return forbar != beginning;
```

should be recognized as 5 tokens

RETURN | ID(forbar) | NEQ | ID(beginning) | SCOLON

9

## Languages & Automata Theory

- Alphabet: a finite set of symbols
- String: a finite, possibly empty sequence of symbols from an alphabet
- Language: a set, often infinite, of strings
- Finite specifications of (possibly infinite) languages
  - » Automaton – a recognizer; a machine that accepts all strings in a language (and rejects all other strings)
  - » Grammar – a generator; a system for producing all strings in the language (and no other strings)
- A language may be specified by many different grammars and automata
- A grammar or automaton specifies only one language

10

## Regular Expressions and Finite Automata

- The lexical grammar (structure) of most programming languages can be specified with regular expressions
  - » Sometimes a little ad-hoc "cheating" is useful
- Tokens can be recognized by a deterministic finite automaton
  - » Can be either table-driven or built by hand based on lexical grammar

11

## Regular Expressions

- Defined over some alphabet $\Sigma$
- If *re* is a regular expression, $L(re)$ is the language (set of strings) generated by *re*
- Note that this is opposite of the way we often think about regular expressions
  - » either way, the relevant set of strings is $L(re)$

12

## Fundamental Regular Expressions

| *re* | *L*(*re* ) | Notes |
|---|---|---|
| a | { a } | Singleton set, for each a in Σ |
| ε | { ε } | Empty string |
| ∅ | { } | Empty language |



## Operations on Regular Expressions

<table>
<tr><td>re</td><td>L(re )</td><td>Notes</td></tr>
<tr><td>rs</td><td>L(r)L(s)</td><td>Concatenation</td></tr>
<tr><td>r|s</td><td>L(r)∪L(s)</td><td>Combination (union)</td></tr>
<tr><td>r*</td><td>L(r)*</td><td>0 or more occurrences (Kleene closure)</td></tr>
</table>

- Precedence: * (highest), concatenation, | (lowest)
- Parentheses can be used to group REs as needed



## Abbreviations

- The basic operations generate all possible regular expressions, but there are common abbreviations used for convenience. Typical examples:

<table>
<tr><td>Abbr.</td><td>Meaning</td><td>Notes</td></tr>
<tr><td>r+</td><td>(rr*)</td><td>1 or more occurrences</td></tr>
<tr><td>r?</td><td>(r | ε)</td><td>0 or 1 occurrence</td></tr>
<tr><td>[a-z]</td><td>(a|b|…|z)</td><td>1 character in given range</td></tr>
<tr><td>[abxyz]</td><td>(a|b|x|y|z)</td><td>1 of the given characters</td></tr>
</table>



## Examples

| *re* | L(*re*) |
|---|---|
| a | single character a |
| ! | single character ! |
| != | specific 2-character sequence != |
| [!<>]= | a 2-character sequence: !=, <=, or >= |
| \[ | single character [ |
| hogwash | 7 character sequence |



## More Examples

| *re* | L(*re*) |
|---|---|
| [abc]+ | |
| [abc]* | |
| [0-9]+ | |
| [1-9][0-9]* | |
| [a-zA-Z][a-zA-Z0-9_]* | |



## Abbreviations

- Can name regular expressions to make writing and reading definitions easier, e.g.,

    *digit* ::= [0-9]

  » Restriction: abbreviations may not be circular (recursive) either directly or indirectly
- Example: possible syntax for numeric constants

  *number* ::= *digits* ( . *digits* )? ( [eE] (+ | -)? digits ) ?
  *digits* ::= *digit*+
  *digit* ::= [0-9]

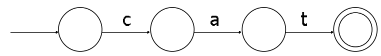
## Recognizing Regular Expressions

- Finite automata can be used to recognize strings generated by regular expressions
- Can build by hand or automatically
  - » Not totally straightforward, but can be done systematically
  - » Tools like Lex, Flex, and JLex do this automatically, given a set of REs



## Finite State Automaton

- A finite set of states
  - » One marked as initial state
  - » One or more marked as final states
- A set of transitions from state to state
  - » Each labeled with symbol from Σ, or ε
- Operate by reading input symbols (usually characters)
  - » Transition can be taken if labeled with current symbol
  - » ε-transition can be taken at any time




initial state

final state
recognize "cat"



## Accept or Reject

- Accept
  - » if in final state and
    - no more input, or
    - no valid transition for the next symbol
- Reject
  - » if not in final state and
    - no more input, or
    - no valid transition for the next symbol



## DFA vs NFA

- Deterministic Finite Automata (DFA)
  - » No choice of which transition to take under any condition
- Non-deterministic Finite Automata (NFA)
  - » Choice of transition in at least one case

  - » Accept if *some* way to reach final state on given input
  - » Reject if no possible way to final state



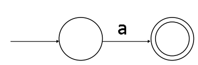
## Finite Automata in Scanners

- Want DFA for speed (no backtracking)
- Conversion from regular expressions to NFA is straightforward
- There is a procedure for converting a NFA to an equivalent DFA

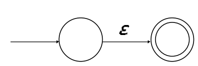


## From RE to NFA: base cases

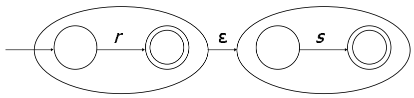
*regex: a*




*regex: ε*

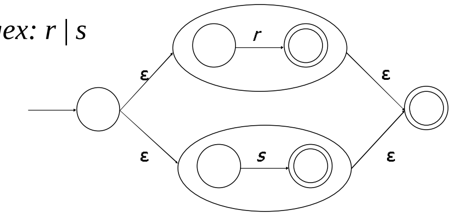


## *regex: rs*

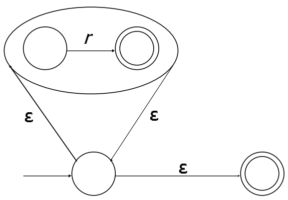



25

*regex: r | s*




*regex: r **




## Exercise: Write NFA for a*[bc]d

27

## Exercise: Write NFA for [ab]d*c

28

## Exercise: Write NFA for $b^+$[abc]*dc

29

## From NFA to DFA

- Subset construction
  - » Construct a DFA from the NFA, where each DFA state represents a *set* of NFA states
- Key idea
  - » The state of the DFA after reading some input is the set of *all* states the NFA could have reached after reading the same input
- If NFA has *n* states, DFA has at most $2^n$ states
  - » => DFA is finite, can construct in finite # steps
- Resulting DFA may have more states than needed
  - » See the books for construction and minimization details

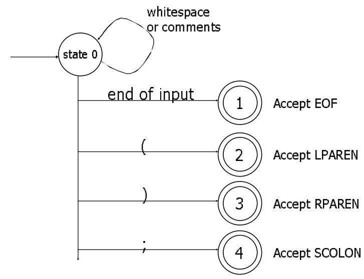
30

## Simple DFA example

- Idea: show a hand-written DFA for some typical programming language constructs
  - » Can use to construct hand-written scanner
- Setting: Scanner is called whenever the parser needs a new token
  - » Scanner stores current position in input
  - » Starting there, use a DFA to recognize the longest possible input sequence that makes up a token and return that token

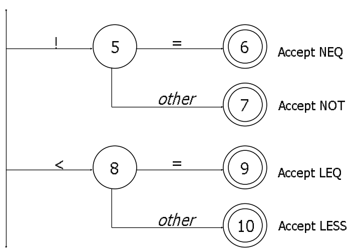
31

## Scanner DFA Example (1)



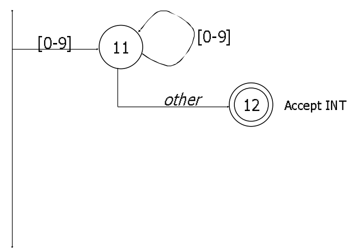

32

## Scanner DFA Example (2)



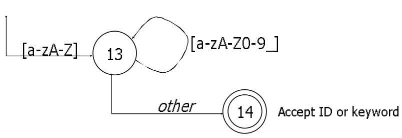

33

## Scanner DFA Example (3)




34

## Scanner DFA Example (4)




- Strategies for handling identifiers vs keywords
  - » Hand-written scanner: look up identifier-like things in table of keywords to classify (good application of perfect hashing)
  - » Machine-generated scanner: generate DFA will appropriate transitions to recognize keywords
    - Lots 'o states, but efficient (no extra lookup step)

35