Lexical Analysis: Finite Automata

CS 471
September 5, 2007

University of Virginia

Phases of a Compiler

Source program
→ Lexical analyzer
→ Syntax analyzer
→ Semantic analyzer
→ Intermediate code generator
→ Code optimizer
→ Code generator
→ Target program

Lexical Analyzer

- •Group sequence of characters into lexemes – smallest meaningful entity in a language (keywords, identifiers, constants)
- •Makes use of the theory of regular expressions and finite state machines
- •Lex and Flex are tools that construct lexical analyzers from regular expression specifications

---

Last Time: Tokenizing

- **•Breaking the program down into words or "tokens"**
- **•Input: stream of characters**
- **•Output: stream of names, keywords, punctuation marks**
- **•Side effect: Discards white space, comments**

Source code: if (b==0) a = "Hi";

Lexical Analysis

Token Stream: if (b == 0) a = "Hi" ;

Parsing

Regular Expression Syntax

[abcd]	**one of the listed characters (a | b | c | d)**
[b-g]	**[bcdefg]**
[b-gM-Qkr]	**[bcdefgMNOPQkr]**
[^ab]	**anything but one of the listed chars**
[^a-f]	**anything but the listed range**
M?	**Zero or one M**
M+	**One or more M**
M*	**Zero or one M**
"a.+*"	**literally a.+***
.	**Any single character (except \n)**

Breaking up Text

elsex=0;

else | x | = | 0 | ;

elsex | = | 0 | ;

- REs alone not enough: need rules for choosing
- Most languages: longest matching token wins
- Ties in length resolved by prioritizing tokens
- RE's + priorities + longest-matching token rule = lexer definition

---

Lexer Generator Specification

- **Input** to lexer generator:
  - list of regular expressions in priority order
  - associated *action* for each RE (generates appropriate kind of token, other bookkeeping)
- **Output**:
  - program that reads an input stream and breaks it up into tokens according to the REs. (Or reports lexical error -- "*Unexpected character"*)

Lex: A Lexical Analyzer Generator

- **Lex produces a C program from a lexical specification**
- **(Please read the man page)**

```
%%
DIGITS  [0-9]+
ALPHA  [A-Za-z]
CHARACTER {ALPHA}|_
IDENTIFIER  {ALPHA}({CHARACTER}|{DIGITS})*
%%
if                                     {return IF; }
{IDENTIFIER}                           {return ID; }
{DIGITS}                               {return NUM; }
([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+)  {return REAL; }
.                                      {error(); }
```


Lexer Generator

- Reads in list of regular expressions *R*1,…*Rn*, one per token, with attached actions

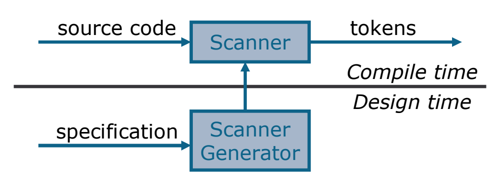
```
-?[1-9][0-9]* { return new Token(Tokens.IntConst,
                     Integer.parseInt(yytext()) }
```

- Generates scanning code that decides:
  1. whether the input is lexically well-formed
  2. corresponding token sequence
- Observation: Problem 1 is equivalent to deciding whether the input is in the language of the regular expression
- How can we efficiently test membership in L(*R*) for arbitrary *R*?

Regular Expression Matching

- Sketch of an efficient implementation:
  - start in some initial state
  - look at each input character in sequence, update scanner state accordingly
  - if state at end of input is an *accepting state*, the input string matches the RE
- For tokenizing, only need a finite amount of state: (*deterministic*) *finite automaton* (DFA) or *finite state machine*

High Level View

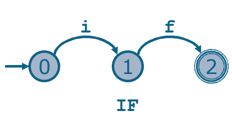


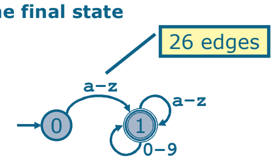
Regular expressions = specification
Finite automata = implementation

Every regex has a FSA that recognizes its language

Finite Automata

- **Takes an input string and determines whether it's a valid sentence of a language**
- **A finite automaton has a finite set of states**
- **Edges lead from one state to another**
- **Edges are labeled with a symbol**
- **One state is the start state**
- **One or more states are the final state**



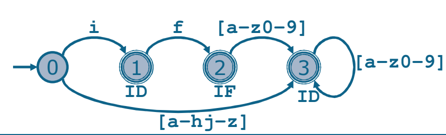


Language

Each string is accepted or rejected

1. **Starting in the start state**
2. **Automaton follows one edge for every character (edge must match character)**
3. **After n-transitions for an n-character string, if final state then accept**

Language: set of strings that the FSA accepts



Finite Automata

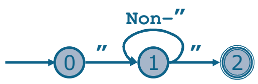
Automaton (DFA) can be represented as:

- A transition table

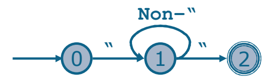
"[^"]*"

	"	non-"
0		
1		
2		

- A graph



Implementation



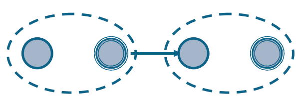

```
boolean accept_state[NSTATES] = { … };
int trans_table[NSTATES][NCHARS] = { … };
int state = 0;

while (state != ERROR_STATE) {
      c = input.read();
      if (c < 0) break;
      state = table[state][c];
}
return accept_state[state];
```


RegExp → Finite Automaton

- Can we build a finite automaton for every regular expression?
- Strategy: consider every possible kind of regular expression (define by induction)

a

R_1R_2

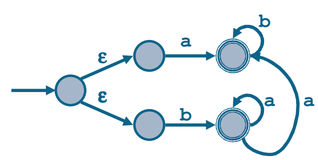


$R_1 | R_2$ **?**

Deterministic vs. Nondeterministic

Deterministic finite automata (DFA) – No two edges from the same state are labeled with the same symbol

Nondeterministic finite automata (NFA) – may have arrows labeled with ε (which does not consume input)

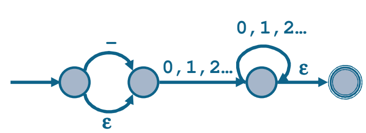


DFA vs. NFA

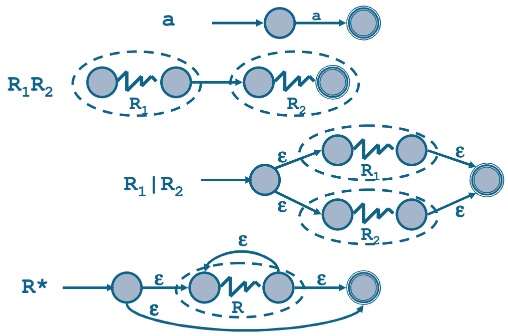
- **DFA**: action of automaton on each input symbol is fully determined
  - obvious table-driven implementation
- **NFA**:
  - automaton may have choice on each step
  - automaton accepts a string if there is *any way* to make choices to arrive at accepting state / every path from start state to an accept state is a string accepted by automaton
  - not obvious how to implement efficiently!

RegExp → NFA

-? [0-9]+ **(-|ε) [0-9][0-9]***



Inductive Construction



Executing NFA

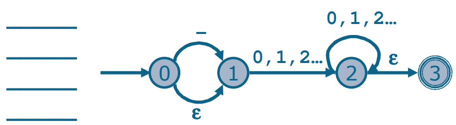
- **Problem**: how to execute NFA efficiently?

"strings accepted are those for which there is some corresponding path from start state to an accept state"

- **Conclusion**: search all paths in graph consistent with the string
- **Idea**: search paths in parallel
  - Keep track of subset of NFA states that search could be in after seeing string prefix
  - "Multiple fingers" pointing to graph

Example

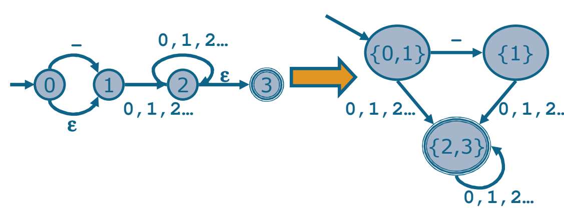
- **Input string: -23**
- **NFA States**



- **Terminology: *ε-closure* - set of all reachable states without consuming any input**
  - *ε-closure* of 0 is {0,1}

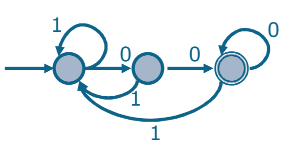
NFA→DFA Conversion

- Can convert NFA directly to DFA by same approach
- Create one DFA for each distinct subset of NFA states that could arise
- States: {0,1}, {1}, {**2, 3**}



DFA Minimization

- DFA construction can produce large DFA with many states
- Lexer generators perform additional phase of *DFA minimization* to reduce to minimum possible size



What does this DFA do?

Can it be simplified?

Automatic Scanner Construction

To convert a specification into code

1. **Write down the RE for the input language**
2. **Build a big NFA**
3. **Build the DFA that simulates the NFA**
4. **Systematically shrink the DFA**
5. **Turn it into code**

Scanner generators

- **Lex and flex work along these lines**
- **Algorithms are well known and understood**
- **Key issue is interface to the parser**

Lexical Analysis Summary

- **Regular expressions**
  - efficient way to represent languages
  - used by lexer generators
- **Finite automata**
  - describe the actual implementation of a lexer
- **Process**
  - Regular expressions (+priority) converted to NFA
  - NFA converted to DFA

<u>Next Time</u>

- **C Primer (PA1) is due tonight**
- **Lexer for Tiger (PA2) coming up**
- **Next Week: Parsing**