CSE401: Lexing

David Notkin
Autumn 2000

Objectives for today and tomorrow

- Define overall theory and practical structure of lexical analysis
- Briefly recap regular expressions, finite state machines, and their relationship
  - Even briefer recap of the language hierarchy
  - Show how to define tokens with regular expressions
  - Show how to leverage this style of token definition in implementing a lexer

Lexical analysis (Scanning)

- A token is a group of characters forming basic, atomic chunks of syntax
  - Example: 17, :=, 3.1415, else, snork, ...
- The lexer also removes whitespace
  - Whitespace consists of characters between tokens that are ignored
  - Examples: spaces, tabs, newlines, comments
  - The definition of whitespace varies from language to language

After scanning: syntactic analysis

- The sequence of tokens produced by the scanner is parsed as part of syntactic analysis
- This separation is followed universally
- Lexing and parsing are theoretically and practically different activities
  - Character stream to token stream vs. token stream to syntax tree
  - Scanning is time-consuming in many compilers, largely due to handling I/O
  - By restricting the job of the lexer, a faster implementation is often feasible

Overall approach to scanning

1. Define the tokens for the language using regular expressions
   - Natural representation for tokens
   - But difficult to produce a scanner from REs
2. Convert the regular expressions into non-deterministic finite state automata (NFA)
   - Straightforward conversion
   - Can produce a scanner from NFA, but an inefficient one
3. Convert the NFA into deterministic finite state automata (DFA)
   - Straightforward conversion
4. Convert the DFA into an efficient scanner implementation

Language and automata theory: a speedy reminder

- **Alphabet**: a finite set of symbols
- **String**: a finite, possibly empty, sequence of characters in an alphabet
- **Language**: a (possibly empty or infinite) set of strings
- **Grammar**: a finite specification of a language
  - Even if the language is infinite
- **Language automaton**: a machine for accepting a language and rejecting all other strings
  - A language can be specified by many different grammars and automata
  - A grammar or automaton specifies precisely one language
Definitions

- **Lexeme**: a group of characters that form a token
- **Token**: a set of lexemes that match a pattern
  - We’ll use regular expressions to define tokens
  - A token may have attributes, if the set has more than a single lexeme
    - Ex: integers are a token, but each integer lexeme must also know its value

Regular expressions: a notation for defining tokens

- The syntax of regular expressions (REs) is defined inductively
  - Base cases
    - The empty string (ε)
    - A symbol from the alphabet
  - Inductive cases
    - Sequence of two REs: E₁E₂
    - Choice of two REs: E₁|E₂
    - Kleene closure (zero or more occurrences) of an RE: E*
    - Can use parentheses for grouping
    - Precedence
      - * (highest)
      - sequence
      - | (lowest)
    - Whitespace is not significant

Notational conveniences: no additional expressive power

- E+ means one or more occurrences of E
- Ek means k occurrences of E
- [E] means zero or one occurrences of E (it’s optional)
- {E} means E*
- not(x) means any character in the alphabet but x
- not(E) means any strings in the alphabet but those matching E
- E₁ - E₂ means any strings matching E₁ except those matching E₂

Naming regular expressions: simplify RE definitions

- Can assign names to regular expressions
- Can use these names in the definition of another regular expression
- Examples
  - letter ::= a | b | … | z
  - digit ::= 0 | 1 | … | 9
  - alphanum ::= letter | digit
- Can eliminate names by macro expansion
- **No recursive definitions are allowed!** Why?

Regular expressions for PL/0

Program ::= (Token | White)*
Token ::= Id | Integer | Keyword | Operator | Punct
Punct ::= ; | | . | , | ( | )
Keyword ::= module | procedure | begin | end | const
Operator ::= := | * | / | + | - | = | <> | =< | >= | >
Integer ::= Digit*
Id ::= Letter AlphNum*
AlphNum ::= Letter | Digit
Digit ::= 0 | … | 9
Letter ::= a | … | z | A | … | Z
White ::= <space> | <tab> | <newline>

Generate scanner from regular expressions?

- This would be ideal: REs as input to a scanner generator, and a scanner as output
  - Indeed, some tools can mostly do this
- But it’s not straightforward to do this
  - One reason is that there is a lot of non-determinism — choice — that is inherent in regular expressions in general
  - Choice can be implemented using recursion, but it’s generally very inefficient
  - In any case, these tools go through a process like the one we’ll look at
Next steps

- Convert regular expressions to non-deterministic finite state automata (NFA)
- Then convert the NFA to deterministic finite state automata (DFA)
- Then convert DFA into code

Classes of languages

- Regular languages can be specified by:
  - regular expressions
  - regular grammars
  - finite-state automata (FSA)
- Context-free languages (CFL) can be specified by:
  - context-free grammars (CFG)
  - push-down automata (PDA)
- Turing-computable languages can be specified by:
  - arbitrary grammars
  - Turing machines

Finite state automata

- A finite set of states
  - One marked as the initial state
  - One or more marked as final states
- A set of transitions from state to state
  - Each transition is marked with a symbol from the alphabet or with ε
  - Operate by reading symbols in sequence
  - A transition can be taken if it labeled with the current symbol
  - An ε-transition can be taken at any point, without consuming a symbol
  - Accept if done with input and in a final state
  - Reject if no transition can be taken or if input is done and not in a final state

DFA vs. NFA

- A deterministic finite state automata (DFA) is one in which there is no choice of which transition to take under any condition
- A non-deterministic finite state automata (NFA) is one in which there is a choice of which transition to take in at least one situation

Plan of attack

- Convert from regular expressions to NFAs because there is an easy construction
  - However, NFAs encode choice, and choice implies recursion, and recursion is slow in a scanner
- Convert from NFAs to DFAs, because there is a well-defined procedure
  - And DFAs lay the foundation for an efficient scanner implementation
Example: in groups

Consider the language that includes only those binary strings that have odd parity.

For this language, define:
- the alphabet
- a grammar
- an automaton

Converting REs to NFAs:

- base cases

E₁E₂

E₁ | E₂

E∗

Those rules are sufficient for constructing an equivalent NFA from a regular expression.
Time permitting

*in groups*

- Define a regular expression that recognizes comments of the form
  - `/* ... */`
  - Be careful in defining "..."
  - Then convert that regular expression to an NFA

Next lecture

- NFA to DFA
- DFA to scanner