CSE401: Lexical Analysis

Larry Snyder

Autumn 2003

Objectives (today and tomorrow)

- Define overall theory and practical structure of lexical analysis
- Briefly recap regular languages, expressions, finite state machines, and their relationships
- How to define tokens with regular expressions
- How to leverage this to implement a lexer

Lexical analysis (scanning)

Source Program | stream of characters → *Lexical analysis* → sequence of tokens

- The scanner/lexer groups characters into tokens
- A *token* is a basic, atomic chunk of syntax, e.g.
  - Literals: `17`, `42`, `3.1415`, `"Hello."`, …
  - Punctuation & operators: `}`, `)`, `]`, `;`, `:=`, `<`, `<=`, …
  - Reserved words: `if`, `then`, `else`, `for`, `while`, `int`, `char`, …
  - Identifiers: `snork`, `x`, `dogbert`, `sqrt`, `printf`, …
- The lexer also removes whitespace
  - Whitespace: characters that are ignored between tokens
  - Ex: spaces, tabs, newlines, comments
- Definitions of tokens and whitespace vary among languages

Separation of lexing & parsing

- A universal separation:
  - Lexer: character stream to token stream
  - Parser: token stream to syntax tree
- Advantages:
  - Simpler design
    - Based on related but distinct theoretical underpinnings
    - Compartmentalizes some low-level issues, e.g., I/O, internationalization, …
  - Faster
    - Lexing is time-consuming in many compilers (40-60% ?)
    - By restricting the job of the lexer, a faster implementation is usually feasible

Overall approach to scanning

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

Language & automata theory: a speedy reminder

- 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 the 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

Definitions: token vs lexeme

- Token: an "atom of syntax"; set of lexemes
  - Ex: int literal, string literal, identifier, keyword-if
- Lexeme: the character string forming a token
  - Ex: 17, 42, "Hello", "Goodbye", x, dogbert, if
- A token may have attributes, if the set has more than a single lexeme
  - "int literal" token might have attribute "17" or "42"
  - "keyword-if" token probably needs no attributes

Regular expressions:
a notation for defining tokens

- Regular expressions (REs) are defined inductively:
- Base cases
  - The empty string (ε)
  - A symbol from the alphabet
- Inductive cases
  - Choice of two REs: $E_1 | E_2$
  - Sequence of two REs: E_1E_2
  - Kleene closure (zero or more occurrences) of an RE: E^*

Increasing precedence

- Use parentheses for grouping
- Whitespace is not significant

Examples

a

a b

(a | b)

(a | b) c

a | b c

a b*

(a | b)(0 | 1)*

Notational conveniences:
no additional expressive power

- E^+ means one or more occurrences of E
- E^k means k occurrences of E (k a literal constant)
- `[E]` means 0 or 1 occurrences of E (it's optional)
- `{E}` means E^*
- `not(x)` means any character in the alphabet but x

rarely implemented (potentially expensive)

- `not(E)` means any strings in the alphabet but those matching E
- $E_1 - E_2$ means any strings matching E_1 except those matching E_2

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

```
Digit    ::= 0 | … | 9
Letter   ::= a | … | z | A | … | Z
Integer  ::= Digit+
AlphaNum ::= Letter | Digit
Id       ::= Letter AlphaNum*
Keyword  ::= module | procedure | begin | end | const
           | var | if | then | while | do | input
           | output | odd | int
Punct    ::= ; | : | . | , | ( | )
Operator ::= := | * | / | + | - | = | <> | <= | < | >= | >
Token    ::= Id | Integer | Keyword | Operator | Punct
White    ::= <space> | <tab> | <newline>
Program  ::= (Token | White)*
```

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: there is a lot of non-determinism — choice — inherent in most regular expressions
  - Choice can be implemented using backtracking, 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

Finite state automaton

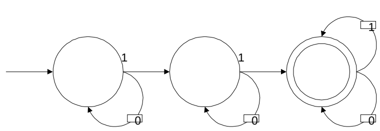
- A finite set of states
  - One marked as the initial state
  - One or more marked as final states
- A set of transitions from state to state
  - Each transition is marked with a symbol from the alphabet or with ε
- Operate by reading symbols in sequence
  - A transition can be taken if it labeled with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

DFA vs. NFA

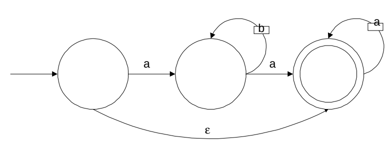
- A deterministic finite state automaton (DFA) is one in which there is no choice of which transition to take under any condition
- A non-deterministic finite state automaton (NFA) is one in which there is a choice of which transition to take in at least one situation
  - "Accept" == some way
  - "Reject" == all ways fail

  $\left.\begin{matrix}\text{"Accept" == some way}\\ \text{"Reject" == all ways fail}\end{matrix}\right\}\left\{\begin{matrix}\text{to reach final state}\\ \text{at end of input}\end{matrix}\right.$

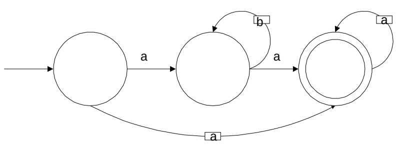
Example



Example



Example



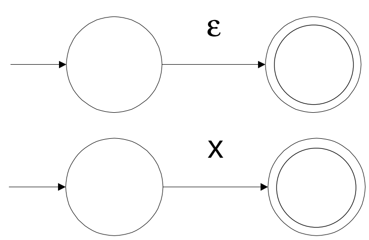
Plan of attack

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

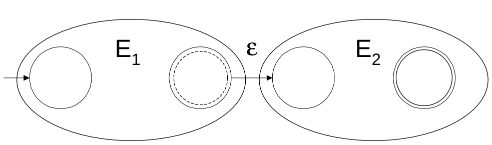
Exercise

- Consider the language that includes only those binary strings that have odd parity
- For this language, define
  - the alphabet
  - a grammar
  - an automaton

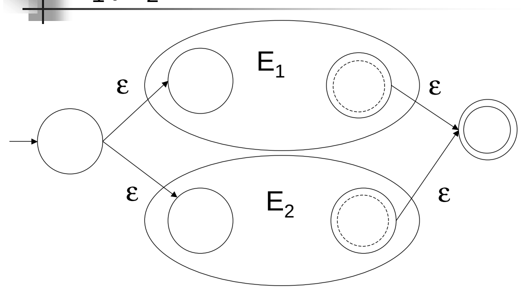
Converting REs to NFAs: *base cases*



E_1E_2



$E_1 | E_2$



E^*

?

RE to NFA

- Those rules are sufficient for constructing an equivalent NFA from a regular expression

Exercise

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

Building lexers from regular expressions

- Convert the regular expressions into deterministic finite state automata (DFA)
  - Manually
  - Mechanically by converting first to non-deterministic finite state automata (NDFA) and then into DFA
- Convert DFA into scanner implementation
  - By hand into a collection of procedures
  - Mechanically into a table-driven parser

Why convert to DFAs?

- Because
  - they are equivalent in power to NFAs
  - they are deterministic, which makes them a terrific basis for an efficient implementation of a scanner

NFA => DFA

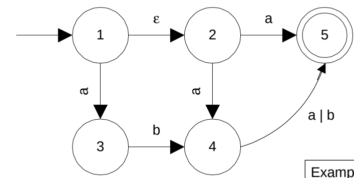
- Basic problem
  - NFA can choose among alternative paths
    - either ε transitions or
    - multiple transitions from a state with the same label
  - But a DFA cannot have this kind of choice
- Solution: subset construction
  - In the newly constructed DFA, each state represents a *set* of states in the NFA,
- Key Idea:

> *the* state of the DFA after reading $x_1x_2\ldots x_k$ is the set of *all* states that the NFA might reach after reading the same input

Subset construction algorithm
initial step

- Create start state of new DFA
  - Label it with the set of NFA states that can be reached without consuming any input
    - I.e., NFA's start state, or reachable by ε transitions
    - Think of it as all possible start states in the NFA, since there could be more than one, given the ε transitions
- Then "process" this new start state
  - Details below

Example



Example from *Crafting a Compiler*, Fischer & LeBlanc

Example (cont.)

Subset construction algorithm
processing a state

- To process a state S in the new DFA with label $\{s_1,\ldots,s_n\}$
- For each symbol x in the alphabet
  - Compute the set T of NFA states reached from any of the NFA states $s_1,\ldots,s_n$ by *one* x transition followed by *any number* of ε transitions
  - If T is not empty
    - If there is not already a DFA state with T as a label, create one, and add T to the list of states to be processed
    - Add a transition labeled x from S to T
- Repeat until no unprocessed states

Subset construction algorithm
defining final states

- After the algorithm terminates
- Mark every DFA state as final if *any* of the NFA states in its label is final

Subset construction: notes

- It is provable that this works and produces an equivalent DFA (c.f. CSE 322)
- This activity can be automated
- Question: What can be said about the number of states in the DFA relative to the NFA?
  - In theory? In practice?

Minimizing DFAs

- There is also an algorithm for minimizing the number of states in a DFA
- Given an arbitrary DFA, one can find a unique DFA with a minimum number of states that is equivalent to the original DFA
  - Except for a renaming of the states
  - Essentially, try to merge states

Constructing scanners from DFAs

- Use a table-driven scanner
- Write disciplined procedures that encode the DFA
- We'll talk about both (the first briefly)
- The second approach is used in the PL/0 compiler
  - Because it's generally easier to handle a few practical issues (but may be slower?)

Approach 1: Table-driven

- Represent the DFA as an adjacency matrix
  - One row per state
  - One column per character in the alphabet
  - Entry is state to transition to
- Mechanically walk the input, taking appropriate transitions
  - Rules for termination remain unchanged

	a	b
{1,2}	{3,4,5}	
{3,4,5}	{5}	{4,5}
{4,5}	{5}	{5}
{5}		

Approach 2: Procedural

- Define a procedure for each state in the DFA
- Use conditionals to check the input character and then make the appropriate transition
- A transition is a call to the procedure for the next state
- (Call overhead optimizable)

```
procedure {3,4,5} begin
  if nextChar() == ‘a’
    call {5}
  elsif nextChar() == ‘b’
    call {4,5}
  else
    reject(“no transition
            out of this
            state”)
end
```


The heart of the PL/0 scanner
it's not quite as clean (but it's not bad!)

```
Token ::= Id |
          Integer |
          Keyword |
          Operator |
          Punct
```

```
if (isalpha(CurrentCh)) {
   T = GetIdent()
} else if (isdigit(CurrentCh)) {
   T = GetInt()
} else {
   T = GetPunct();
}
```

- Where's the DFA?
- How come five kinds of tokens and only three branches?

PL/0's `GetIdent` method

- Is PL/0 case-sensitive?
- What does `SearchReserved` return?

```
Token* Scanner::GetIdent() {
  char ident[MaxIdLength+1];
  int LengthOfId = 0;
  while (isalnum(CurrentCh)) {
    ident[LengthOfId] =
         tolower(CurrentCh);
    LengthOfId ++;
    GetCh();
  }
  ident[LengthOfId] = '\0';
  return SearchReserved(ident);
}
```

PL/0's `GetInt` method

```
Token* Scanner::GetInt() {
  int integer = 0;
  while (isdigit(CurrentCh)) {
    integer = 10 * integer + (CurrentCh - '0');
    GetCh();
  }
  return new IntegerToken(integer);
}
```


PL/0's GetPunct method

```
Token* Scanner::GetPunct() {
  Token* T;
  switch (CurrentCh) {
    case ':':
      GetCh();
      if (CondReadCh('=')) {
        T = new Token(GETS);
      } else {
        T = new Token(COLON);
      }
      break;
```

```
case '<':
  GetCh();
  if (CondReadCh('=')) {
    T = new Token(LEQ);
  } else if (CondReadCh('>')){
    T = new Token(NEQ);
  } else {
    T = new Token(LSS);
  }
  break;
  …
```


A few PL/0 scanner notes

- There is a `Scanner` class
  - There is only one instance of this class
  - This is an example of the *Singleton* design pattern
- The high-level structure we showed has the scanner scan before the parser parses
  - Study the compiler to figure out what really happens
- Make sure (for this and all other phases) to read the interface (the `.h` file) very, very carefully

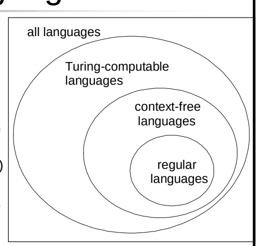
Language design issues (lexical)

- Most languages are now free-form
  - Layout doesn't matter
  - Use whitespace to separate tokens, if needed
  - Alternatives include
    - Fortran, Algol68: whitespace is ignored
    - Haskell: use layout to imply grouping
- Most languages now have reserved words
  - Cannot be used as identifiers
  - Alternative: PL/1 has keywords that are treated specially only in certain contexts, but may be used as identifiers, too
- Most languages separate scanning & parsing
  - Alternative: C/C++ *type* vs *ident*

```
typedef int mytype;
int myvar;
mytype i,j,k;
```

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



Strict inclusion of these classes of languages

Objectives: next lectures

- Understand the theory and practice of parsing
- Describe the underlying language theory of parsing (CFGs, etc.)
- Understand and be able to perform top-down parsing
- Understand bottom-up parsing