COMP 520 - Compilers

Lecture 5 (Thu Jan 27, 2022)

Lexical Analysis

- Reading
  - PLPJ Section 4.5 (pp 118 – 124)

- Project
  - With the material and example code covered today you should be ready to build the miniJava parser for the first checkpoint (due Feb 1).

Topics

- Scanning
  - motivation
  - scanner grammars
  - construction of a scanner

- Example scanner and parser
  - `simpleScannerParser`
  - Illustrates scanner construction and integration with parser
  - Illustrates compiler project package structure

- Scanner topics
  - true crimes of scanning!
  - scanner generators

Scanners

- Purpose
  - extract *tokens* from a character stream
    - a token is a terminal symbol in the *parser grammar*
      - e.g. a *number,* an *identifier,* a specific operator, or a specific keyword

- Approach
  - Scanning is just like parsing, except
    - the scanner grammar is simple
      - the *terminals* are individual characters
      - the *nonterminals* are the terminals (tokens) of the parser
      - the *rule* for each nonterminal in S is a regular expression
        - » no recursion needed in scanner grammar
    - must ignore whitespace and comments
      - skip these while scanning for the start of a token
    - must recognize end-of-input
      - end of input condition yields a distinguished token

Simple scanner grammar

- The Token nonterminal in the scanner grammar derives all possible parser terminals

Token ::= Operator | Number | Identifier | Keyword

Operator ::= `+` | `-` | `*` | `/` | `>` | `=` | `>=` | …

Number ::= Digit Digit*

Identifier ::= Alpha AlphaNum*

Keyword ::= `for` | `begin` | `end` | …

Digit ::= `0` | `1` | `2` | … | `8` | `9`

Alpha ::= `a` | `b` | … | `z` | `A` | `B` | … | `Z`

AlphaNum ::= Alpha | Digit

Preparing the grammar for scanning

- Substitute definitions to create a single rule for Token
  - grammar may be ambiguous and require more than 1 char lookahead
  - ad hoc resolution
    - keyword vs identifiers
      - scan Alpha (AlphaNum)*
      - use spelling and a hashmap to determine keyword vs identifier
    - > vs. >=
      - choose longer token
  - scanner hides a multitude of little parsing sins

- Simple example

  Token ::= + | * | (|) | Digit Digit*

Constructing the scanner

- The scanner consists of
  - a way to inspect the current character
    - currentChar
  - a method to advance to the next character
    - nextChar()
  - a method scan() that
    1. skips over whitespace and comments
    2. parses a single token according to the scanner grammar
       - returning an instance of the Token class providing
         - » token kind
         - » token spelling

Scanner examples

1. Scanner for simple arithmetic expressions

   For use with the simpleScannerParser example

   Token ::= Num | Oper | (|)

   Digit ::= 0 | 1 | 2 | … | 8 | 9

   Num ::= Digit Digit*

   Oper ::= + | *

2. Distinguishing integer literals and float literals

   Token ::= Int | Float

   Int ::=

   Float ::=

Simple scanner grammar

- Specification of a scanner using a scanner grammar
  - example grammar (below)
    - ambiguous rules: Keyword/Identifier, Int/Real
      - longest match determines correct token
      - if tie, use order in specification (i.e. Keyword before Identifier)
    - ignore whitespace (WS)
    - what about input that doesn't match any rule?

Token ::= Keyword | Identifier | Int | Real

Keyword ::= `for`
Identifier ::= Alpha AlphaNum*
Int ::= Digit Digit*
Real ::= Int `.` Int?

Digit ::= `0` | `1` | `2` | … | `8` | `9`
Alpha ::= `a` | `b` | … | `z` | `A` | `B` | … | `Z`
AlphaNum ::= Alpha | Digit
WS ::= `' '` | `\n` | `\t`

Ad-hoc Scanner

```
public Token scan() {
  … skip over whitespace
  switch current_char {
     case ‘a’..’z’: case ‘A’..’Z’:
        … accumulate chars in spelling until current_char not alphanum
        if (spelling is a keyword)
              return new Token(TokenKind.KEYWD, spelling);
        else
              return new Token(TokenKind.ID, spelling);
     case ‘0’..’9’:
        … accumulate input in spelling until non-digit
        if (current_char != “.”)
           return new Token(TokenKind.INT_LITERAL, spelling);
        else {
           … accumulate input until non-digit
           return new Token(TokenKind.REAL_LITERAL, spelling)
     default:
           … report scan error
           return new Token(TokenKind.ERROR, spelling);
   }
}
```

Integrating Scanner and Parser

- Approach
  - Scanner reads text input and returns Tokens with a TokenKind and Spelling
  - Parser terminals are Tokens

- Create a simple example to study
  - import into Eclipse
    - download simpleScannerParser.zip file from the Examples section on the course web page
    - open Eclipse and choose File/import/Existing Projects into workspace/Next
    - choose select archive file browse for download simpleScannerParser.zip and Select it
    - this should create the simpleScannerParser project in Eclipse

Integrating Scanner and Parser (contd)

- The example illustrates the package structure of the miniJava compiler
  - miniArith package
    - “Recognizer” mainclass and an “ErrorReporter” class
      - » Parses keyboard input and reports whether it is an instance of a simple arithmetic expression or whether it is syntactically incorrect
      - » note: your “miniJava” package should have a “Compiler” mainclass in place of the “Recognizer” class and read the source file specified in args[0] and compile an output file for execution by a (virtual) machine
  - subpackage miniArith.SyntacticAnalyzer
    - includes “Parser” and “Scanner” classes
    - parser grammar
      - » S ::= E *eot*
      - » E ::= T (*oper* T)*
      - » T ::= *num* | *lparen* E *rparen*
    - scanner grammar
      - » *num* ::= digit digit*   *lparen* ::= ‘(‘
      - » *oper* ::= '+' | '*‘   *rparen* ::= ‘)’
      - » digit ::= '0' | ... | '9‘   *eot*

True crimes of scanning!

- Scanning seems simple?
  - Scanning Fortran
    - whitespace is uniformly ignored

      VAR1   same as VA   R1
    - consider

      DO 5 I = 1, 25   (do, intlit, id, =, intlit, “,”, intlit)

      DO 5 I = 1. 25   (id, =, floatlit)

      can’t tell which token to return without reading (far) ahead

      Complicated!
  - Scanning PL/1
    - PL/1 keywords are not reserved, so can also be identifiers

      IF THEN = ELSE THEN ELSE = THEN ELSE THEN = ELSE

      which are identifiers and which are keywords? Context sensitive!

True crimes of scanning!

- Even modern languages present hard problems
  - Scanning C
    - predecrement, postdecrement, subtraction, negation

      `--x   x--   x-y   -x`
    - how to scan these?

      `x---x     x----x    x-----x`

      `x-- - x   x-- - -x  x-- - --x`

    sometimes whitespace is *required*

  - Scanning C++
    - C++ template syntax `Foo<Bar>`
    - C++ stream syntax `cin >> var`

    how do we scan “>>” in `Foo<Bar<Bazz>>` ?

  - Scanning Ada
    - real numbers `1.3  0.1  1.0  1.  .1`
    - range syntax `1..10`

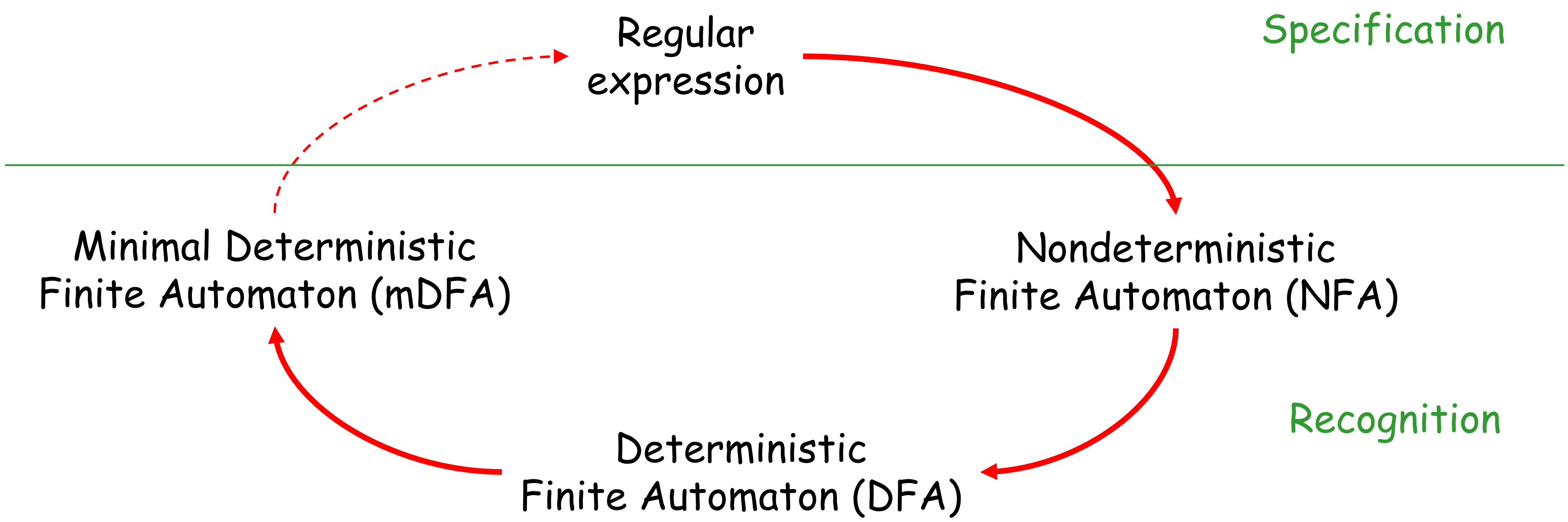
  - more: C nested comments, typedefs, preprocessor

Scanner Generation

- From specification to implementation
  - Ad hoc scanner
    - implement recognizer for each regular expression
    - resolve ambiguities individually
      - longest match resolution can be complicated
      - large or unbounded lookahead may be needed
    - can be quite efficient
  - Systematically generated scanner
    - construct a *finite automaton* from specification
    - scanner is implementation of the automaton
    - correctness is guaranteed
    - efficiency generally quite good

Systematic scanner generation

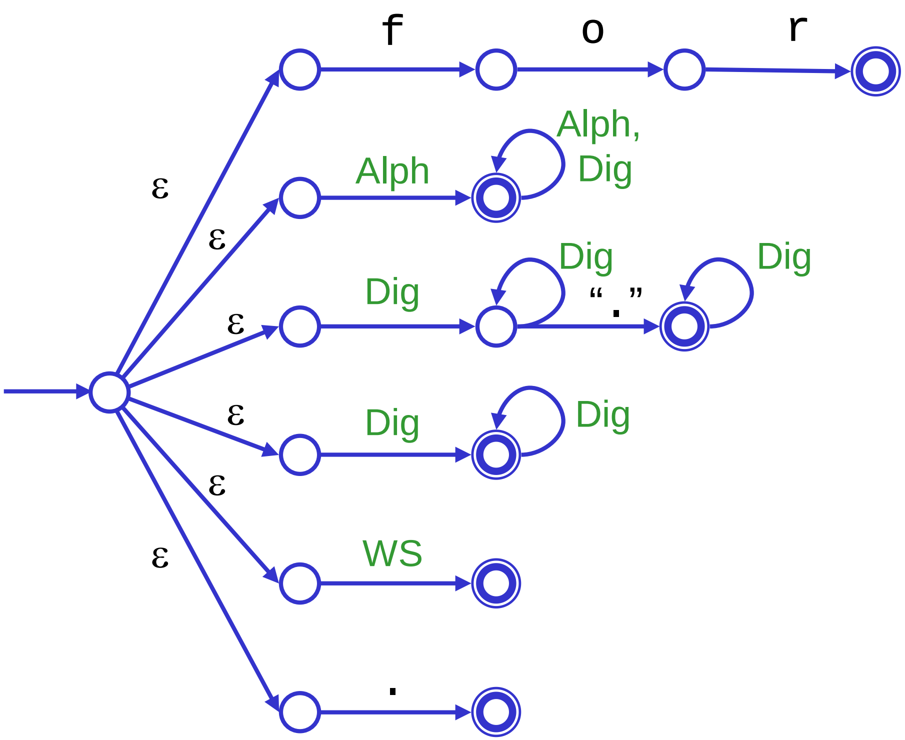
- Idea



- Use construction of DFA from regular expression
  - modified for
    - recognition and output of tokens
    - ambiguity in specification
    - efficiency of resulting scanner

Scanner NFA construction

NFA



Token	Regular Expr
FOR	“for”
ID	Alpha AlphaNum*
REAL	Int ”.” Int?
INT	Int
(no token)	WS
ERR	. (meaning any char)

NFAs

- NFA $M = (Q, \Sigma, \delta, F, q_0)$
  - Q - set of states
  - Σ - alphabet
  - δ - transition function $\delta: Q \rightarrow 2^Q$
  - F - set of final states $F \subseteq Q$
  - q_0 - initial state

- definitions
  - ε - closure: $2^Q \rightarrow 2^Q$

$$\varepsilon - \text{closure}(S) = \bigcup_{q \in S} (\text{states reachable from } q \text{ via } \varepsilon \text{ edges})$$

  - extended transition function $\hat{\delta}: 2^Q \rightarrow 2^Q$

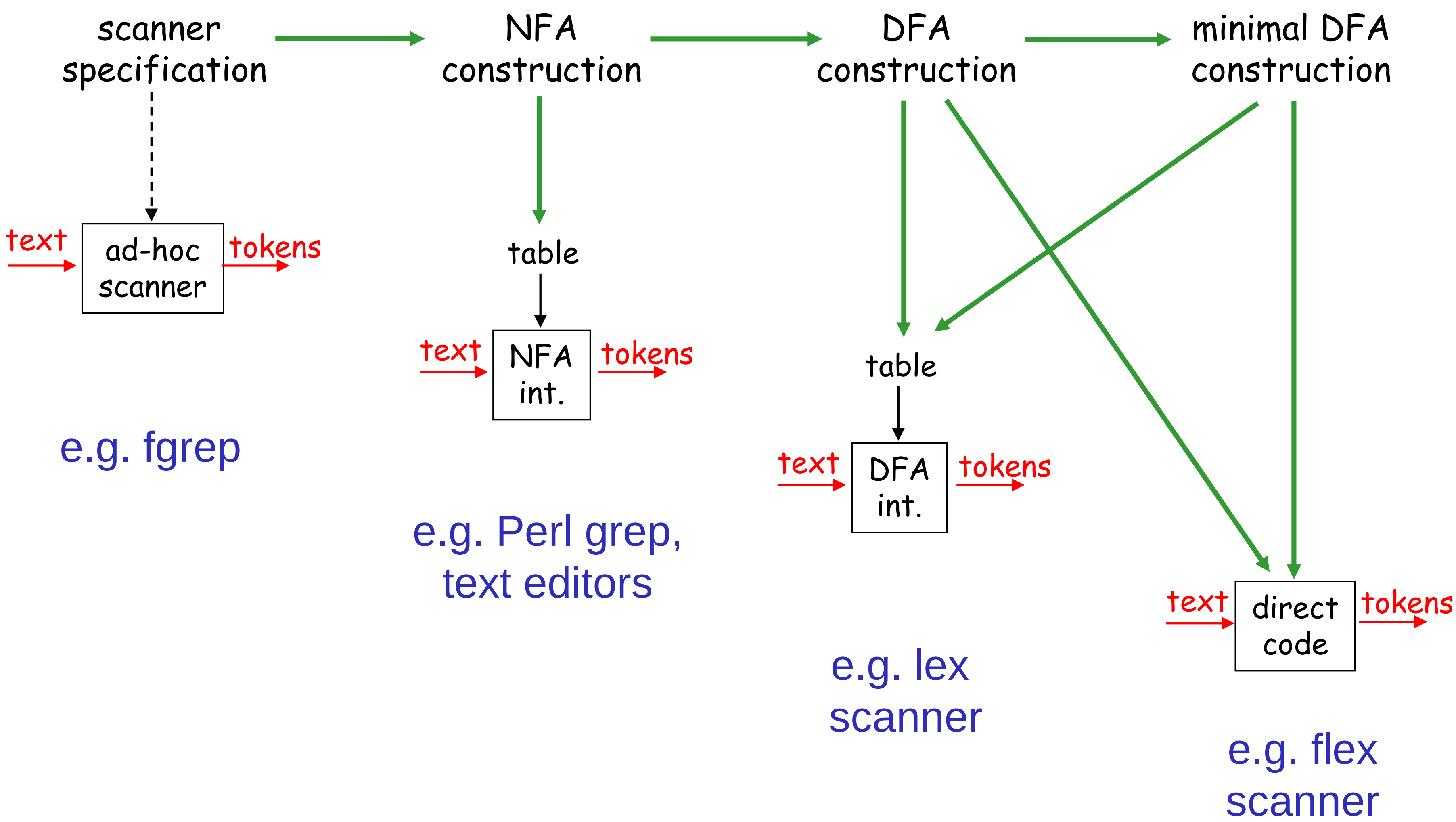
$$\hat{\delta}(S,a) = \bigcup_{q \in S} \delta(q,a)$$

Scanner as NFA interpreter

- Idea
  - track possible states of NFA on input until no state reachable
  - back up input and NFA to most recent final state
    - return token associated with that state

```
S := ε-closure({q0})
a := next_char(); token := NONE; save_state(input posn, S);
while (S ≠ ∅) do
   S := ε-closure( δ^(S, a) )
   a := next_char()
   if (S ∩ F ≠ ∅) then token := min(S ∩ F); save_state(input posn, S)
end do
if (token == NONE)
  then lexical error
  else restore_state(); return(token)
```

Scanner Generation strategies



Scanner generators

- scanner generators
  - lex, flex (C)
  - flex++ (C++)
  - jflex (Java)

- integrated scanner/ parser generators for Java
  - javaCC
  - SableCC

```
Alpha     [a-zA-Z]
Digit     [0-9]
WS        [ \t\n]
AlphaNum  {Alpha}|{Digit}
Int       {Digit}{Digit}*
%%
"for"                   return(FOR);
{Alpha}{AlphaNum}*      return(ID);
{Int}                   return(INT);
{Int}"."{Int}?          return(DOUBLE);
{WS}                    ;
.                       return(ERROR);
```