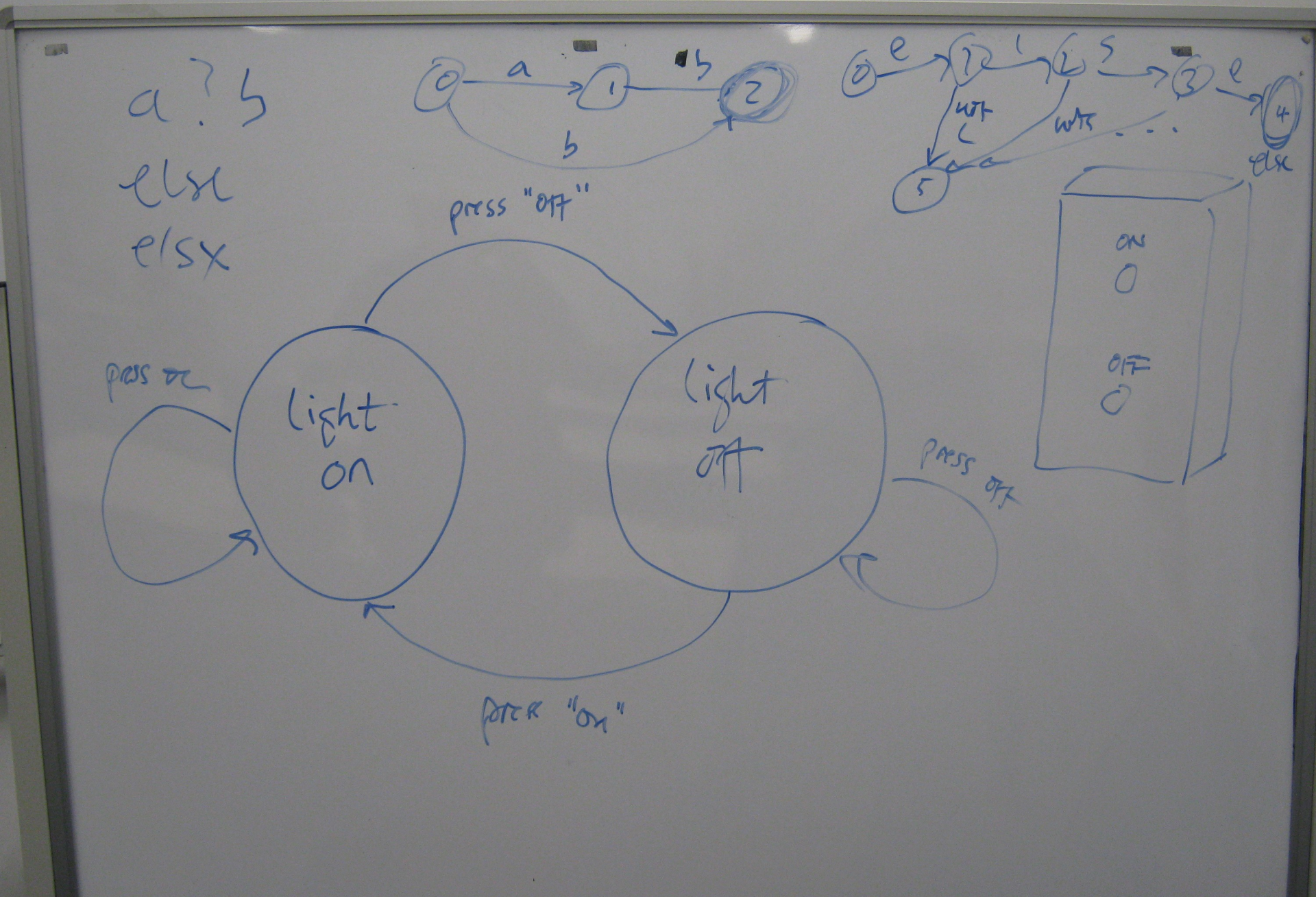
Languages and Compilers
Scanner Implementation

Allan Milne and Adam Sampson
School of Design and Informatics
Abertay University


a ? b
else
else
0
a
1
b
2
b
0
e
1
l
2
s
3
e
4
else
wh
wh
5
press "off"
press on
light on
light off
press off
press "on"
on
off

Agenda.

- Finite State Machines.
- FSM and Scanners.
- Making It Real.

Finite State Machines.

- Finite State Machine (FSM).
  - An FSM Diagram.
    - FSM Operation.
- FSM From Regular Expressions.
  - An FSM Example.
- Text FSM Diagrams.

Finite State Machine (FSM).

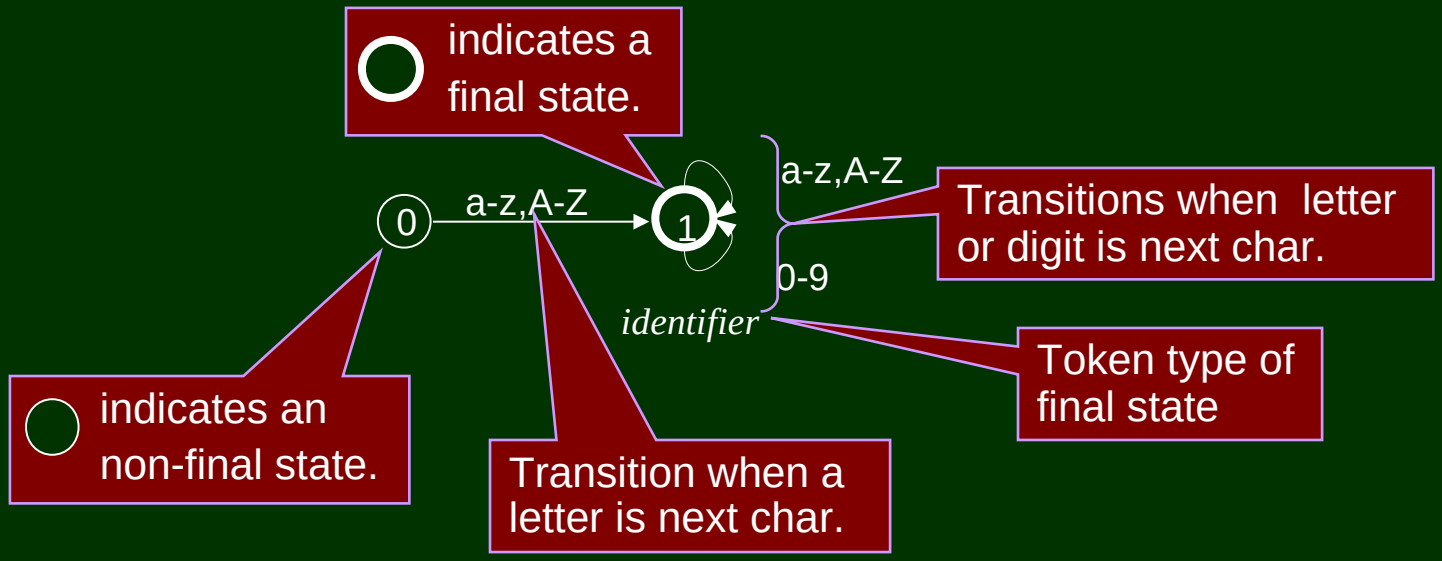
- An approach to defining a process through specifying transitions between states based on some current attribute.
- An FSM is in one of a number of states.
  - It starts in some start state,
  - a transition from one state to another is defined by the current attribute,
  - until no transitions are defined from the current state by the current attribute.
- The FSM approach can be used in many different application domains.

The FSM in Lexical Analysis.

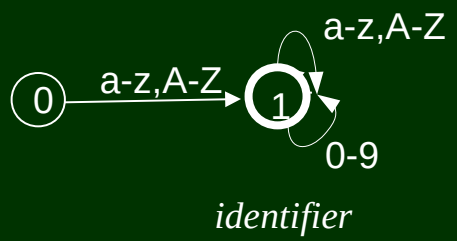
- In the lexical analysis context the process being defined by the FSM is that of finding the next input token.
- The attribute determining the transition between states is the next input character.
- Note that an FSM finds only a single token at a time; it must be restarted each time a new token is required.
- This reflects the functionality of the get*NextToken()* method of the *Scanner* class.

An FSM Diagram.

```
microsyntax
Identifier <|[a-zA-Z]\w*
```



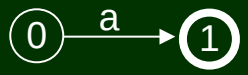
FSM Operation.



- Always begin at the start state (S0).
- Follow the transition that matches the current input character.
  - On following a transition the input is moved on one character.
- Continue until either end of input or there is no transition defined for the current character.
  - Finishing at a final state defines the token found.
  - Finishing at an non-final state means there is an error in the input (an invalid character or token).

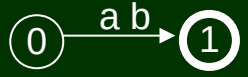
FSM From Regular Expressions.

a

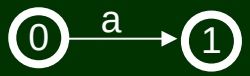


a b

[ab]

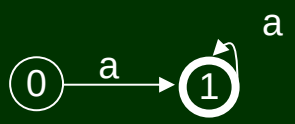


a?



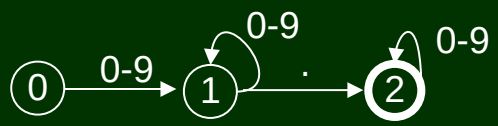
a*

a+



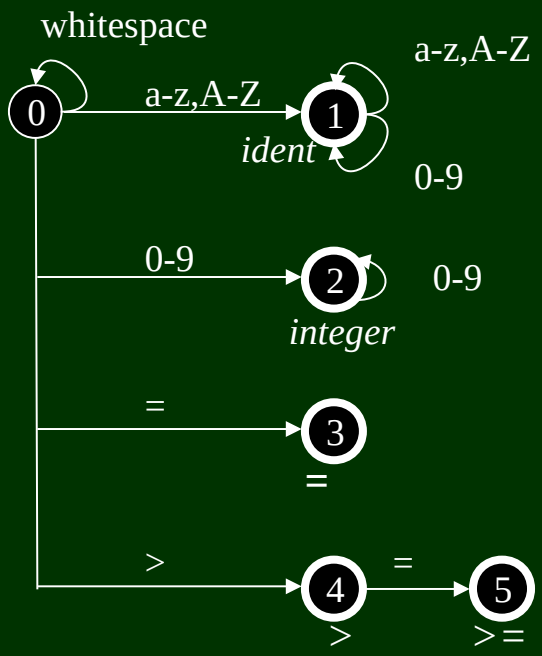
- The first two patterns are used for simple and microsyntax terminals.
- Ultimately there will be one branch from state 0 for each simple and microsyntax terminal.
- The patterns can be combined to reflect microsyntax rules.

- e.g. Number <|\d+\.[0-9]*



An Example FSM.

```
// Rules with = > >= tokens & ignoring whitespace
... ... ...
microsyntax
Identifier <|[a-zA-Z]\w*
Integer <|\d+
```



Example

input: **`max >=30`**

s0 —(m a x)→ s1

- transition with "m" to s1
- "ax" stays in s1
- no " " from s1: *identifier* found

s0 —(> =)→ s5

- skips blanks at s0
- ">" to s4,does not stop in s4
- "=" to s5
- no "3" from s5: >= found

s0 —(3 0)→ s2

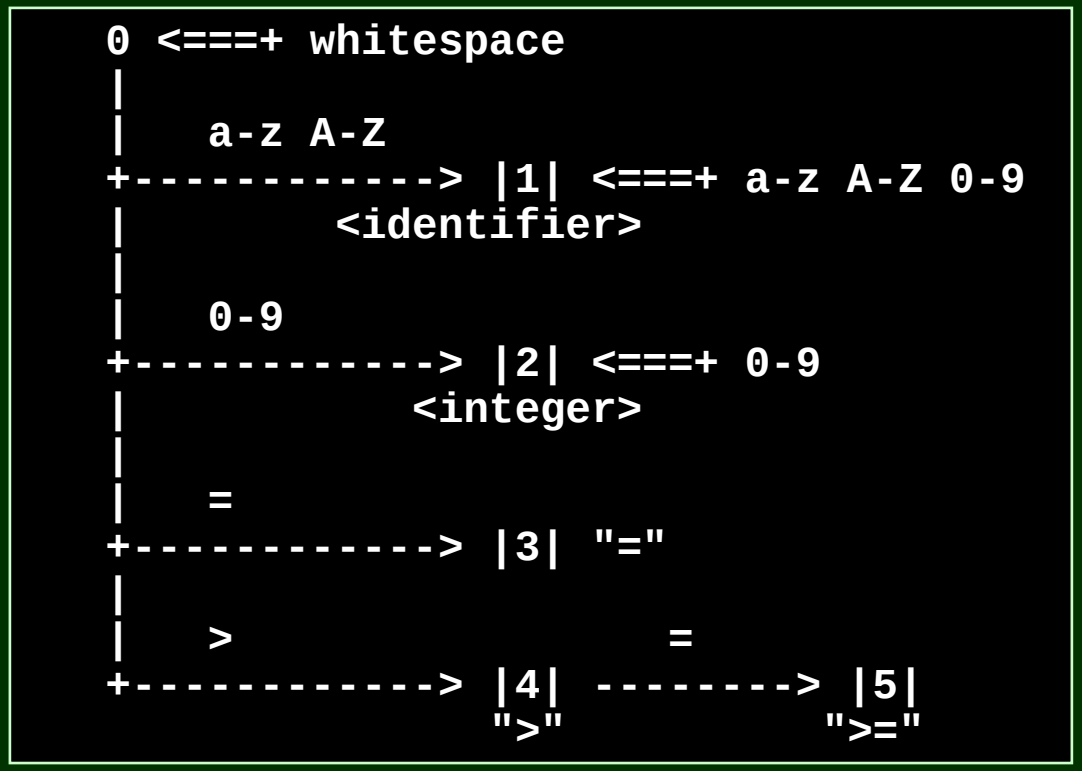
- "3" to s2
- "0" stays in s2
- no transition from s2: integer found

After every recognized token the scanner starts in S0 again

Text FSM Diagrams.

Don't do this in the exam!

- Text FSM diagrams may be easier to "draw" (e.g. in comments!)
- use **+ | - >** for the transition lines.
- **<==+** for looped transitions.
- **1** for non-final state
- **|1|** for a final state.



FSM and Scanners.

- The Scanner As An FSM.
- EOF, Invalid Tokens And Characters.
- State Transition Pattern.
- Example Implementation.

The Scanner As An FSM.

- The FSM finds a single token.
- This is encapsulated as the *getNextToken()* method of the scanner class.
- Each time the method is called it will return the next token found.
- The scanner class must keep track of the input position between method calls.
- See previous lecture for encapsulating functionality.

The *Ardkit* Framework.

- In this framework the *Scanner* class provides the base functionality including the public *NextToken()* method.
- For a specific language the FSM will be implemented in the *getNextToken()* method of a subclass.
- See *Ardkit: Scanner* class reference.
- See *Ardkit: Using a Scanner* page.

EOF, Invalid Tokens And Characters.

- Represent by *Token* objects of *endOfFile*, *invalidToken* and *invalidChar* type.
- End-of-File.
  - use an additional final state in the FSM with a transition from state 0 on the EOF character.
- Invalid tokens.
  - already implemented in the FSM when no transition is defined from a non-final state.
- Invalid characters.
  - use an additional final state from state 0 on any character not used for any other state 0 transition.

State Transition Pattern.

Set state to 0.
While token not found {
Switch on state {
state *n*: if current char defines transition
then set state to next state.
else if … *each transition from state n*
… … …
else *either* token is found *(n is a final state)*
or* invalid token *(n is a non-final state)
… … … *repeat above for all states in FSM.*
}
If token not found then read next character.
}

Outline *getNextToken*() - 1.

```
protected override IToken getNextToken () {
  IToken token = null;
  int state = 0;
  while (token == null) {
    switch (state) {
    case 0:
      if (Char.IsWhiteSpace(currentChar))      state = 0;
      else if (Char.IsLetter(currentChar))     state = 1;
      else if (Char.IsDigit(currentChar))      state = 2;
      else if (currentChar == '=')             state = 3;
      else if (currentChar == '>')             state = 4;
      else if (currentChar == eofChar)         state = 98;
      else                                     state = 99;
      break;
```

Outline *getNextToken()* - 2.

```
case 1:   // Identifier
  if (Char.IsLetter(currentChar) ||
      Char.IsDigit(currentChar))
       state = 1;
  else token = new Token (Token.IdentifierToken, …);
  break;
case 2:   // Integer
  if (Char.IsDigit(currentChar)) state = 2;
  else token = new Token (Token.IntegerToken, …);
  break;
case 3:   // “=” token found.
  token = new Token(“=”, …);
  break;
case 4:   // > or >=
  if (currentChar == '=') state = 5;
  else token = new Token (“>”, …);
  break;
```

Outline *getNextToken()* - 3.

```
  case 5:
      token = new Token (“>=”, …);
      break;
   case 98:
      token = new Token (Token.EndOfFile, …);  break;
   case 99:
      token = new Token (Token.InvalidChar, …);  break;
    } // end switch.
    if (token == null) getNextChar();
  } // end while token not found.
  return token;
} // end getNextToken method.
```

Oops – We Have A Problem!

- We are still missing information required to create the token objects representing the terminals found.
  - Need the line and column where the token string begins.
  - Need the actual source program string value for microsyntax tokens.

- Also: it may be more efficient to implement this using lookup tables – lexer generators would normally do that...

Making It Real.

- Getting The Token Info.
  - Comments.
  - Keyword Tokens.
  - String Tokens.

Getting The Token Info.

- As well as finding the type of token the scanner must also get the starting position of the token in the source code and build up the actual token string.
  - Remember where a token starts.
  - Use a *StringBuilder* object to build up the string char by char as it's read in (except for whitespace).
  - When token is found, call the *ToString()* method to convert the buffer to a string.

Revised *getNextToken()* Method.

```
protected override IToken getNextToken () {
  … … …
  StringBuilder strbuf = null;
  int startLine = 0, startCol = 0;
  while (token == null) {
    switch (state) {
    case 0:
      if (Char.IsWhiteSpace(currentChar))   state = 0;
      else {
        startLine = line;  startCol = column;
        strbuf = new StringBuilder();
         … … …
      }
      break;
    … … …
    } // end switch.
```

getNextToken() Continued.

```
    … … …
    } // end switch.
    if (token == null) {
      if (state != 0) {
        strbuf.Append (currentChar);
      }
      getNextChar();
    }
  } // end while token not found.
  return token;
} // end getNextToken method.
```

Handling Comments.

- Many languages include comments that do not form part of the executable code.
- Comments (mostly) take two forms:
  - start & end symbols (e.g. `/* … */`).
  - start symbol to end-of-line (e.g. `// …`).
- Define the format in microsyntax and the FSM.
  - But may not return a token.
- … or consume comments in *getNextChar()*.
  - Watch for end-of-file!

Handling Keywords.

- Usually have keyword (or reserved word) terminals.
- These have the same microsyntax as identifiers, but are simple rather than microsyntax tokens.
- Handle these as identifiers and then look them up in a keyword table.
  - Are they case-sensitive in the language?

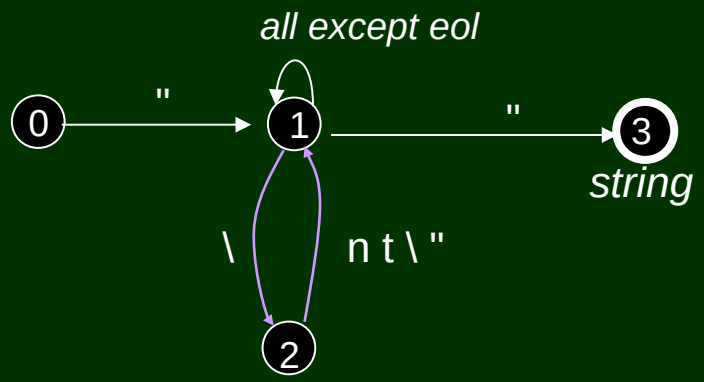
```
class MyScanner : Scanner { … … …
  private static List<String> keywords
    = new List<String>
        (new String[] {
           "begin", "end", ...
        });
}
```

```
String s = strbuf.ToString().ToLower();
if (keywords.Contains(s))
     token = new Token (s, startLine, startCol);
else token = new Token (Token.IdentifierToken, s,
                        startLine, startCol);
```

What is the significance of *.ToLower()*?

Handling Literal String Terminals.

- A language may contain string literals.
  - Usually enclosed in quotation marks.
  - May use escape sequences; eg \n, \t, \\, \".
    - Require to be stored as the actual ASCII code.
  - Usually require to be on a single source line.
  - Allow the empty string "".



- Implement as normal for an FSM.
- What happens if newline is found?
- In state 2 the ASCII code for the escape char can be assigned to *currentChar*.

Summary.

So Now You Can …

… describe the definition and operation of an FSM.

… create an FSM from an EBNF.

… implement an FSM for a scanner using a state transition pattern.

… implement a scanner for a language.

… use the scanner in an application.