CMPE 152: Compiler Design
October 10 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak

www.cs.sjsu.edu/~mak

Midterm Solutions: Question 1

- Briefly explain why it is appropriate to use <u>stacks</u> for the following:

  a. At compile time for symbol tables (symbol table stack).

     - As the compiler parses a Pascal source program from top to bottom, it enters and leaves <u>nested scopes</u>, and therefore it needs to push and pop symbol tables on and off the stack to enable the parser to access local and nonlocal variables.

Midterm Solutions: Question 1, *cont'd*

- Briefly explain why it is appropriate to use stacks for the following:

  a. At run time for activation records (runtime stack).

     - As calls and returns are made to and from procedures and functions, activation records need to be pushed and popped to enable access to the values of local and nonlocal variables.

Midterm Solutions: Question 2

- How does the parser look up a variable name while it is parsing variable declarations, and for what purpose.
  - The parser searches the local symbol table (the one at the top of the symbol table stack) to check that the name is not already declared in the local scope

Midterm Solutions: Question 2, *cont'd*

- How does the parser look up a variable name while it is parsing the boolean expression of an IF statement, and for what purpose.
  - The parser searches the entire <u>symbol table stack</u> to find where that name is <u>already declared</u>, either in the local scope or an enclosing scope.

Midterm Solutions: Question 2, *cont'd*

- How does the executor look up a variable while it is executing the boolean expression of an IF statement, and for what purpose.
  - The executor uses the nesting level of the variable and the <u>runtime display</u> to access the appropriate <u>activation record</u> on the <u>runtime stack</u> and get the variable's <u>value</u>.

Midterm Solutions: Question 3

- Three-operand conditional expression:

  <expression-1> **?** <expression-2> **:** <expression-3>

- Examples:

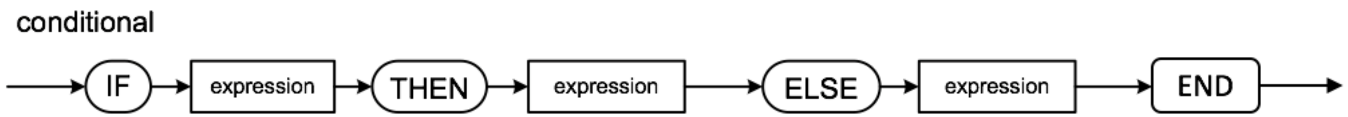
```
float value = 3.14*(x > y ? x : y)/2;
string name = text != null ? "The name is " + text : "Anonymous"
```

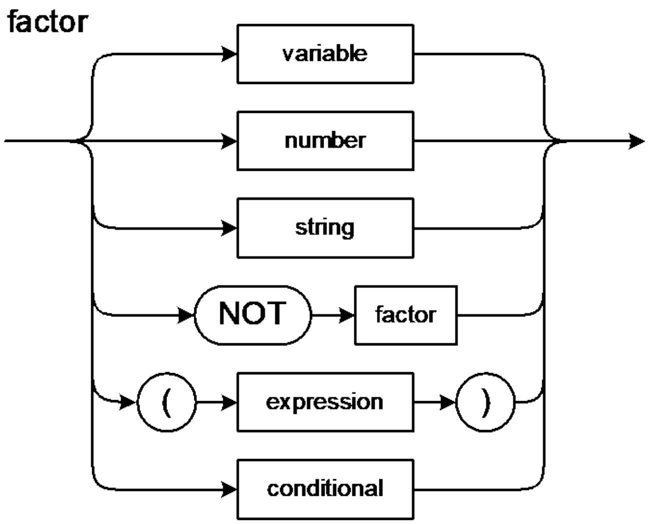
- Pascal implementation:

```
VAR
    i, j, k, m, n : integer;

BEGIN
    i := IF j > k THEN j ELSE k END;
    k := i - j*IF m-n = 0 THEN m*n ELSE m+n END;
...
```

Midterm Solutions: Question 3, *cont'd*





The result at run time of evaluating the conditional operator is a **single value**, the result of evaluating either *<expression-2>* or *<expression-3>*.

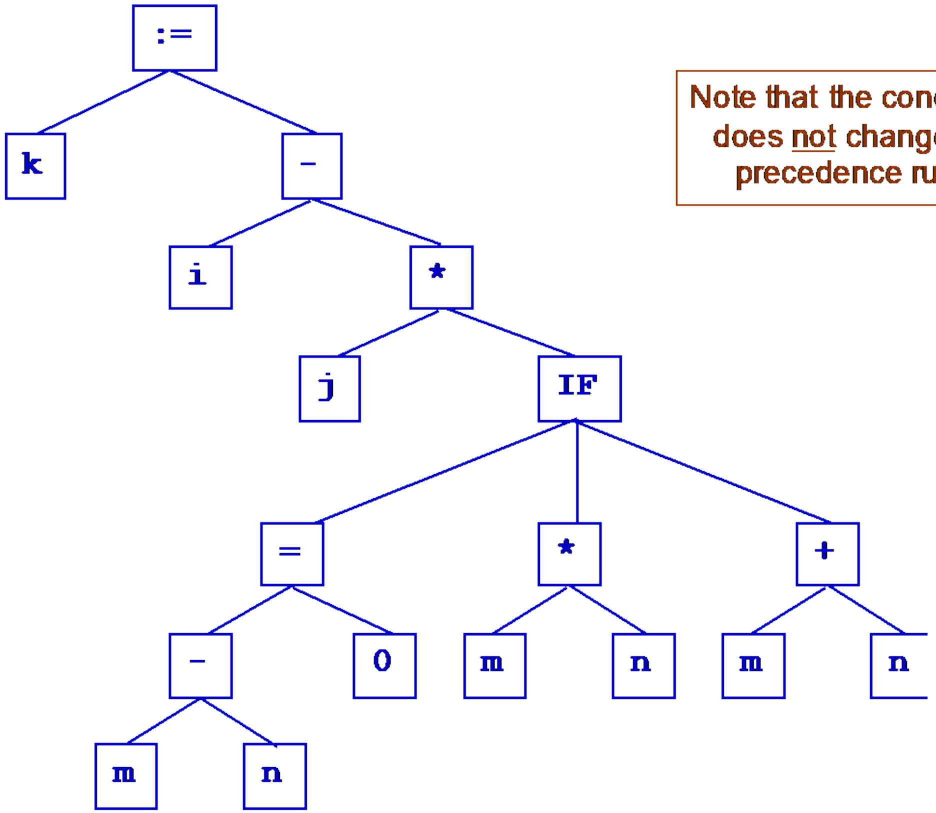
Therefore, a conditional expression must be a **factor**.

Midterm Solutions: Question 3, *cont'd*

- What type checking operations are necessary while parsing a conditional operator?
  - <*expression-1*> must be Boolean.
    <*expression-2*> and <*expression-3*>
    must have the same type.

Midterm Solutions: Question 3, *cont'd*

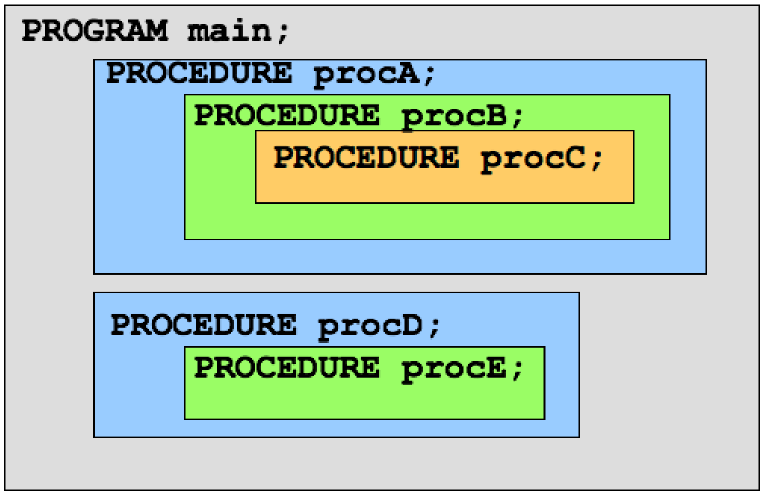
- Parse tree for `k := i - j*IF m-n = 0 THEN m*n ELSE m+n END`

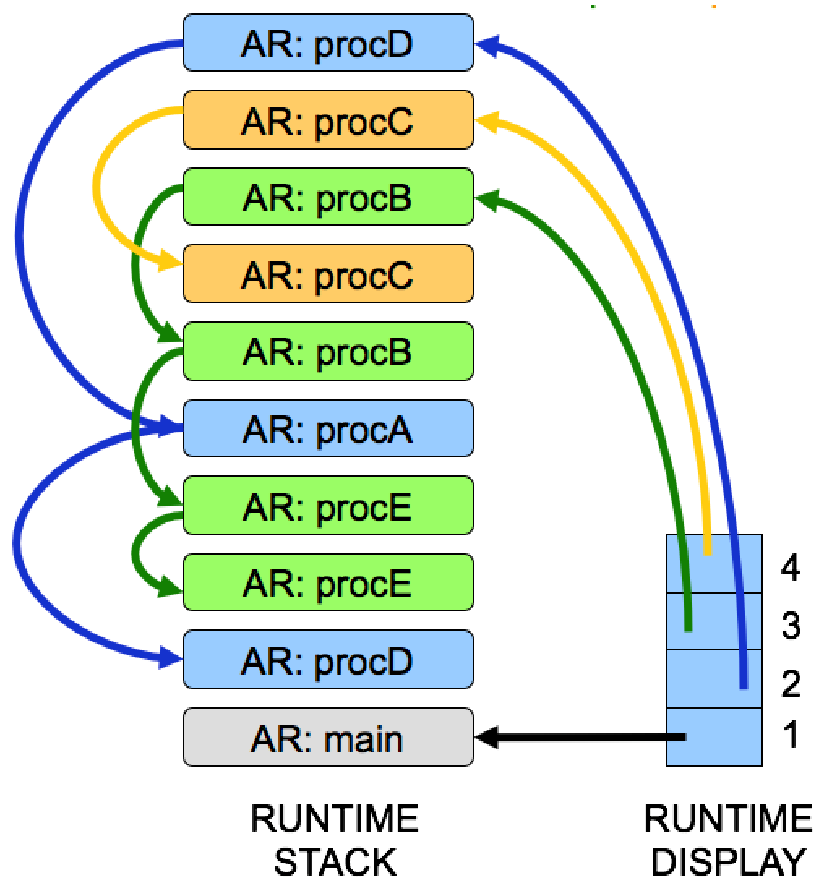


Note that the conditional does not change any precedence rules.

Midterm Solutions: Question 4

`main → procD → procE → procE → procA → procB → procC → procB → procC → procD`





Midterm Solutions: Question 4, *cont'd*

- Given the program structure on the previous page, how does the interpreter enable during run time a statement in **`procC`** to access the values of variables declared in **`procA`** and in **`main`**?
  - The parse tree nodes for the variables indicate the nesting levels of the variables. Using the nesting level and the runtime display, the executor can access the appropriate activation records.

Midterm Solutions: Question 4, *cont'd*

- During run time, what prevents a statement in **`procC`** from accessing the value of a variable declared in **`funcE`**?
  - **`funcE`** is at level 2. While executing **`procC`**, the topmost level 2 activation record is for **`procB`**.

Minimum Acceptable Compiler Project

- At least two data types with type checking.
- Basic arithmetic operations with operator precedence.
- Assignment statements.
- At least one conditional control statement (e.g., IF).
- At least one looping control statement.
- Procedures or functions with calls and returns.
- Parameters passed by value or by reference.
- Basic error recovery (skip to semicolon or end of line).
- "Nontrivial" sample programs written in the source language.
- Generate Jasmin code that can be assembled.
- Execute the resulting **`.class`** file standalone (preferred) or with a test harness.
- No crashes (e.g., null pointer exceptions).

70 points/100

Ideas for Programming Languages

- A language that works with a database such as MySQL
  - Combines Pascal and SQL for writing database applications.
  - Compiled code hides JDBC calls from the programmer.
  - Not PL/SQL – use the language to write client programs.
- A language that can access web pages
  - Statements that "scrape" pages to extract information.
- A language for generating business reports
  - A Pascal-like language with features that make it easy to generate reports.

DSL = Domain-Specific Language

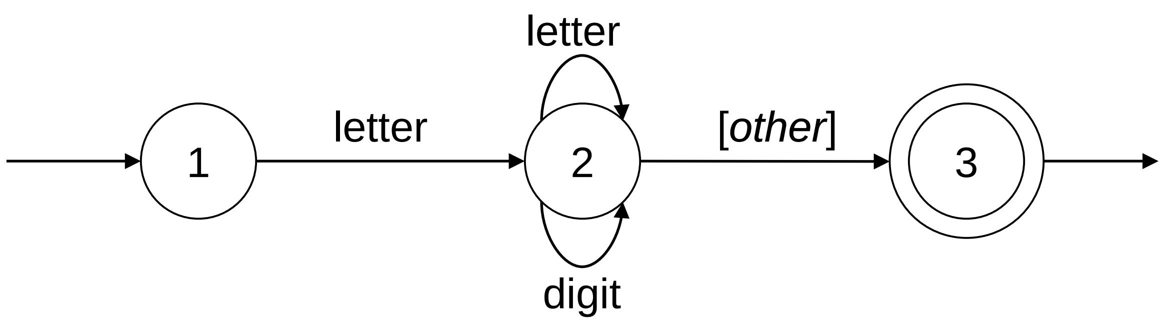
- A string-processing language
  - Combines Pascal and Perl for writing applications that involve pattern matching and string transformations.

Can We Build a Better Scanner?

- Our scanner in the front end is relatively easy to understand and follow.
  - Separate scanner classes for each token type.
- However, it's big and slow.
  - Separate scanner classes for each token type.
  - Creates lots of objects and makes lots of method calls.
- We can write a <u>more compact and faster</u> scanner.
  - However, it may be harder to understand and follow.

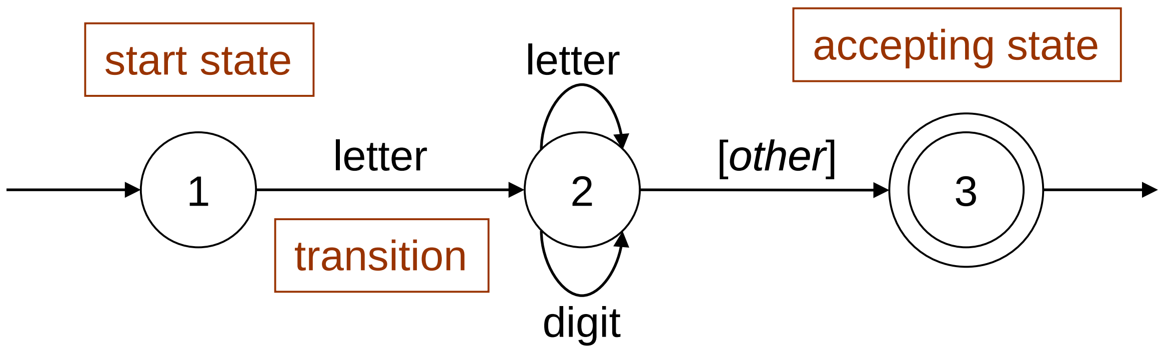
Deterministic Finite Automata (DFA)

- Pascal identifier
  - Regular expression: <letter> (<letter> | <digit>)*
  - Implement the regular expression with a finite automaton (AKA finite state machine):

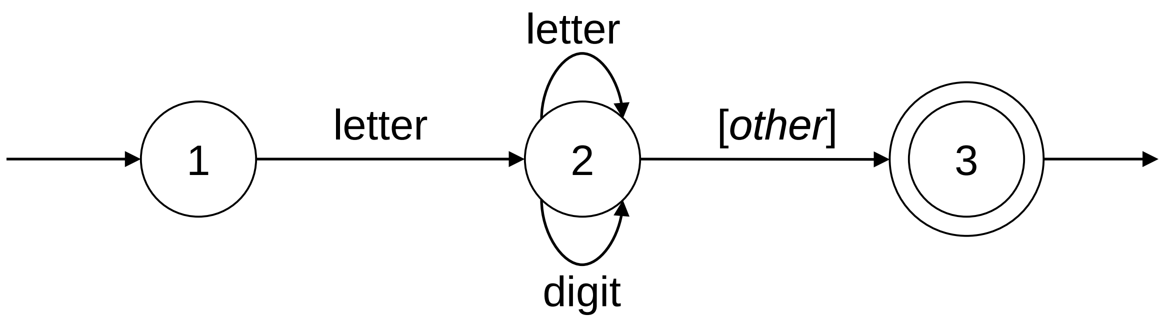


Deterministic Finite Automata (DFA)

- This automaton is a deterministic finite automaton (DFA).
  - At each state, the next input character uniquely determines which transition to take to the next state.



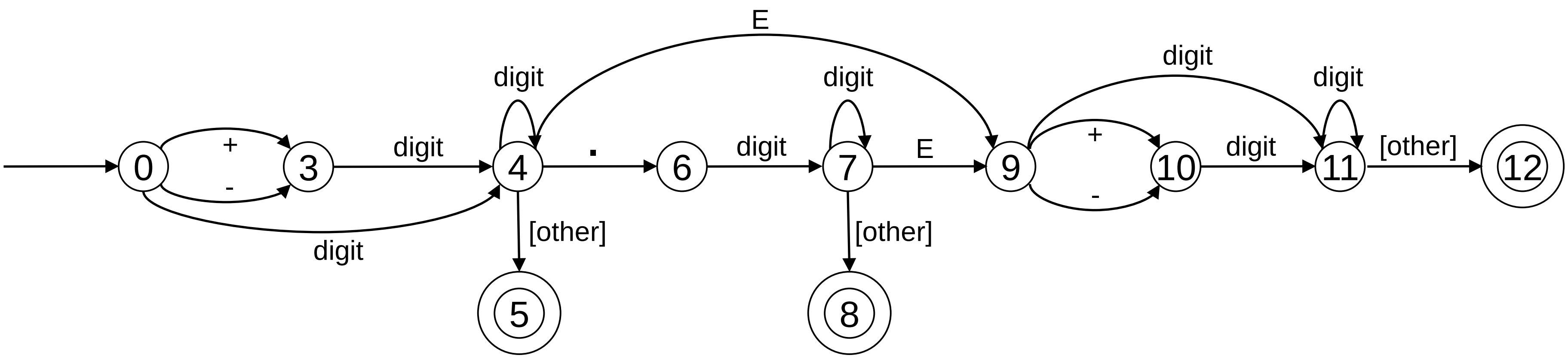
State-Transition Matrix



- Represent the behavior of a DFA by a state-transition matrix:

	Input character		
State	Letter	Digit	other
1	2		
2	2	2	3
3			

DFA for a Pascal Number

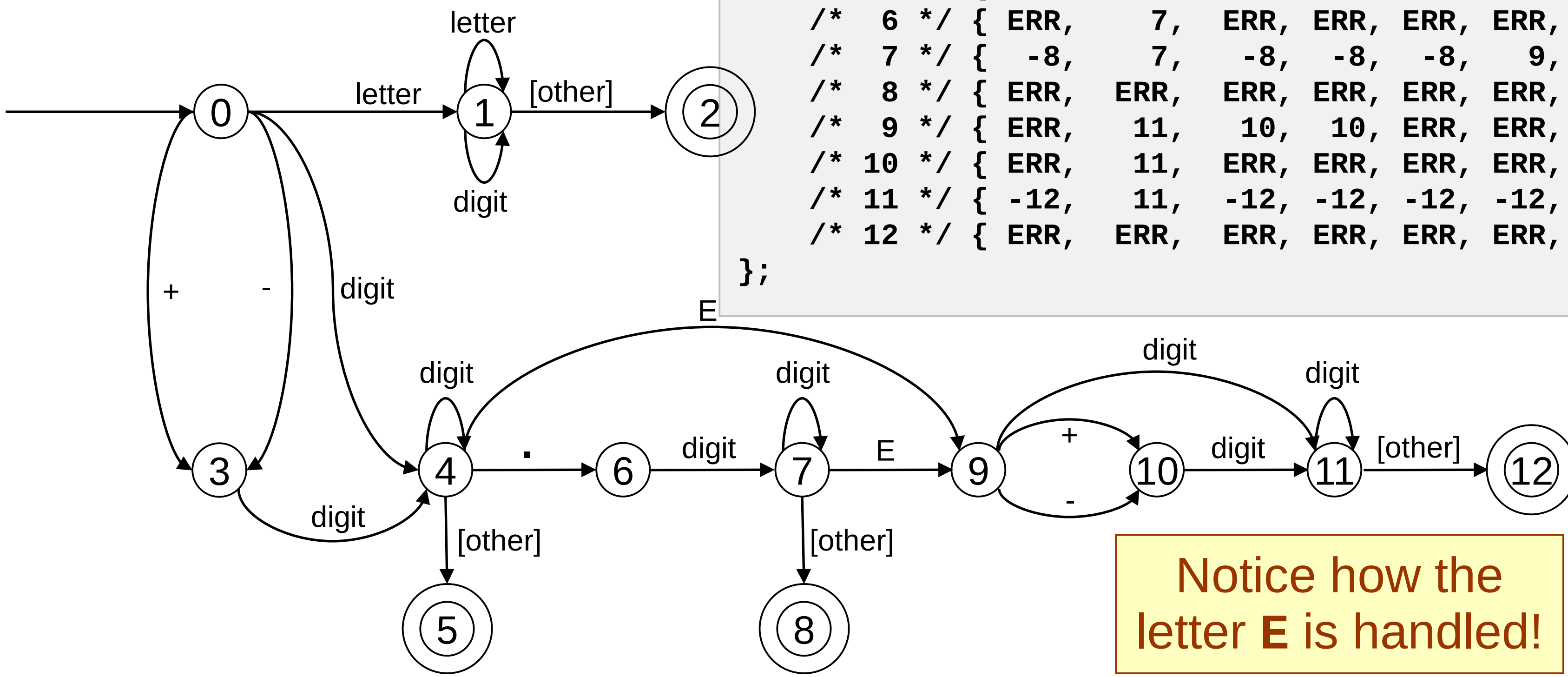


Note that this diagram allows only an upper-case **E** for an exponent. What changes are required to also allow a lower-case **e**?

DFA for a Pascal Identifier or Number

```
const int SimpleDFAScanner::matrix[13][7] = {
    /*          letter digit   +    -    .    E other */
    /*  0 */ {    1,     4,    3,   3, ERR,   1, ERR },
    /*  1 */ {    1,     1,   -2,  -2,  -2,   1,  -2 },
    /*  2 */ {  ERR,   ERR,  ERR, ERR, ERR, ERR, ERR },
    /*  3 */ {  ERR,     4,  ERR, ERR, ERR, ERR, ERR },
    /*  4 */ {   -5,     4,   -5,  -5,   6,   9,  -5 },
    /*  5 */ {  ERR,   ERR,  ERR, ERR, ERR, ERR, ERR },
    /*  6 */ {  ERR,     7,  ERR, ERR, ERR, ERR, ERR },
    /*  7 */ {   -8,     7,   -8,  -8,  -8,   9,  -8 },
    /*  8 */ {  ERR,   ERR,  ERR, ERR, ERR, ERR, ERR },
    /*  9 */ {  ERR,    11,   10,  10, ERR, ERR, ERR },
    /* 10 */ {  ERR,    11,  ERR, ERR, ERR, ERR, ERR },
    /* 11 */ {  -12,    11,  -12, -12, -12, -12, -12 },
    /* 12 */ {  ERR,   ERR,  ERR, ERR, ERR, ERR, ERR },
};
```

Negative numbers in the matrix are the accepting states.



Notice how the letter **E** is handled!

A Simple DFA Scanner

```
class SimpleDFAScanner
{
public:
    SimpleDFAScanner(string source_path);
    virtual ~SimpleDFAScanner();

    /**
     * Scan the source file.
     */
    void scan() throw(string);

private:
    // Input characters.
    static const int LETTER = 0;
    static const int DIGIT  = 1;
    static const int PLUS   = 2;
    static const int MINUS  = 3;
    static const int DOT    = 4;
    static const int E      = 5;
    static const int OTHER  = 6;

    // Error state.
    static const int ERR = -99999;
```

A Simple DFA Scanner, *cont'd*

```
// State-transition matrix (acceptance states < 0)
static const int matrix[13][7];

char ch;    // current input character
int state;  // current state
ifstream reader;
string line;
int line_number;
int line_pos;
```

A Simple DFA Scanner, *cont'd*

```
int SimpleDFAScanner::type_of(char ch)
{
    return   (ch == 'E')  ? E
           : isalpha(ch)  ? LETTER
           : isdigit(ch)  ? DIGIT
           : (ch == '+')  ? PLUS
           : (ch == '-')  ? MINUS
           : (ch == '.')  ? DOT
           :                OTHER;
}
```

A Simple DFA Scanner, *cont'd*

```
string SimpleDFAScanner::next_token() throw(string)
{
    // Skip blanks.
    while (isspace(ch)) next_char();

    // At EOF?
    if (reader.fail()) return "";

    state = 0;  // start state
    string buffer;

    // Loop to do state transitions.
    while (state >= 0)  // not acceptance state
    {
        state = matrix[state][type_of(ch)];  // transition

        if ((state >= 0) || (state == ERR))
        {
            buffer += ch;  // build token string
            next_char();
        }
    }

    return buffer;
}
```

This is the heart of the scanner.

Table-driven scanners can be very fast!

A Simple DFA Scanner, *cont'd*

```
void SimpleDFAScanner::scan() throw(string)
{
    next_char();

    while (ch != 0)  // EOF?
    {
        string token = next_token();

        if (token != "")
        {
            cout << "=====> \"" << token << "\" ";
            string token_type =
                  (state ==  -2) ? "IDENTIFIER"
                : (state ==  -5) ? "INTEGER"
                : (state ==  -8) ? "REAL (fraction only)"
                : (state == -12) ? "REAL"
                :                  "*** ERROR ***";
            cout << token_type << endl;
        }
    }
}
```

How do we know which token we just got?

Demo