

# Language Translation Principles

- The fundamental question of computer science:

“What can be automated?”

- One answer—Translation from one programming language to another.

- Alphabet—A nonempty set of characters.
- Concatenation—joining characters to form a string.
- The empty string—The identity element for concatenation.

{ a, b, c, d, e, f, g, h, i, j, k, l, m, n,  
o, p, q, r, s, t, u, v, w, x, y, z, A, B,  
C, D, E, F, G, H, I, J, K, L, M, N, O, P,  
Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3,  
4, 5, 6, 7, 8, 9, +, -, \*, /, =, <, >, [,  
, (, ), {, }, ., /, :, ;, &, !, %, ' , "  
\_, \, #, ?, }, |, ~ }

{ a, b, c, d, e, f, g, h, i, j, k, l, m, n,  
o, p, q, r, s, t, u, v, w, x, y, z, A, B,  
C, D, E, F, G, H, I, J, K, L, M, N, O, P,  
Q, R, S, T, U, V, W, X, Y, Z, 0, 1, 2, 3,  
4, 5, 6, 7, 8, 9, \, ., /, :, ;, ', " }

$\{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, \cdot \}$

# Concatenation

- Joining two or more characters to make a string
- Applies to strings concatenated to construct longer strings

# The empty string

- $\epsilon$
- Concatenation property

$$\epsilon x = x \epsilon = x$$



# Languages

- The closure  $T^*$  of alphabet  $T$ 
  - ▶ The set of all possible strings formed by concatenating elements from  $T$
- Language
  - ▶ A subset of the closure of its alphabet

# Techniques to specify syntax

- Grammars
- Finite state machines
- Regular expressions

# The four parts of a grammar

- $N$ , a nonterminal alphabet
- $T$ , a terminal alphabet
- $P$ , a set of rules of production
- $S$ , the start symbol, an element of  $N$

$N = \{ \langle \text{identifier} \rangle, \langle \text{letter} \rangle, \langle \text{digit} \rangle \}$

$T = \{ \text{a, b, c, 1, 2, 3} \}$

$P =$  the productions

1.  $\langle \text{identifier} \rangle \rightarrow \langle \text{letter} \rangle$
2.  $\langle \text{identifier} \rangle \rightarrow \langle \text{identifier} \rangle \langle \text{letter} \rangle$
3.  $\langle \text{identifier} \rangle \rightarrow \langle \text{identifier} \rangle \langle \text{digit} \rangle$
4.  $\langle \text{letter} \rangle \rightarrow \text{a}$
5.  $\langle \text{letter} \rangle \rightarrow \text{b}$
6.  $\langle \text{letter} \rangle \rightarrow \text{c}$
7.  $\langle \text{digit} \rangle \rightarrow 1$
8.  $\langle \text{digit} \rangle \rightarrow 2$
9.  $\langle \text{digit} \rangle \rightarrow 3$

$S = \langle \text{identifier} \rangle$

$$N = \{I, F, M\}$$

$$T = \{+, -, d\}$$

$$P = \text{the productions}$$

$$1. I \rightarrow FM$$

$$2. F \rightarrow +$$

$$3. F \rightarrow -$$

$$4. F \rightarrow \epsilon$$

$$5. M \rightarrow dM$$

$$6. M \rightarrow d$$

$$S = I$$

# Grammars

- Context-free
  - ▶ A single nonterminal on the left side of every production rule
- Context-sensitive
  - ▶ Not context-free

$$N = \{A, B, C\}$$

$$T = \{a, b, c\}$$

$$P = \text{the productions}$$

$$1. A \rightarrow aABC$$

$$2. A \rightarrow abC$$

$$3. CB \rightarrow BC$$

$$4. bB \rightarrow bb$$

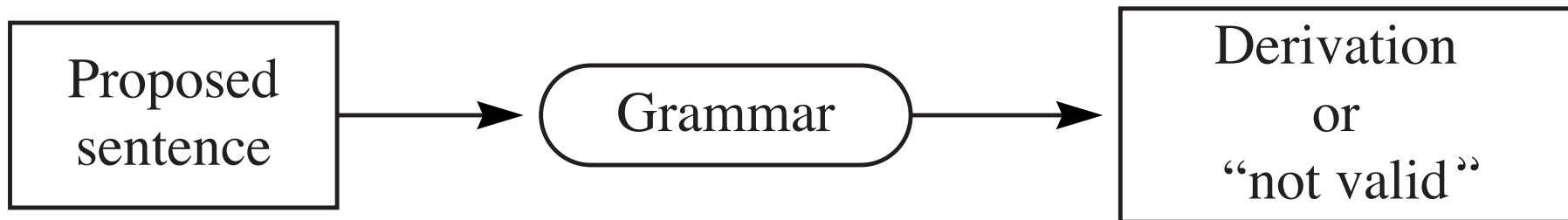
$$5. bC \rightarrow bc$$

$$6. cC \rightarrow cc$$

$$S = A$$



(a) Deriving a valid sentence.



(b) The parsing problem.



$$N = \{E, T, F\}$$

$$T = \{+, *, (, ), a\}$$

$$P = \text{the productions}$$

$$1. E \rightarrow E + T$$

$$2. E \rightarrow T$$

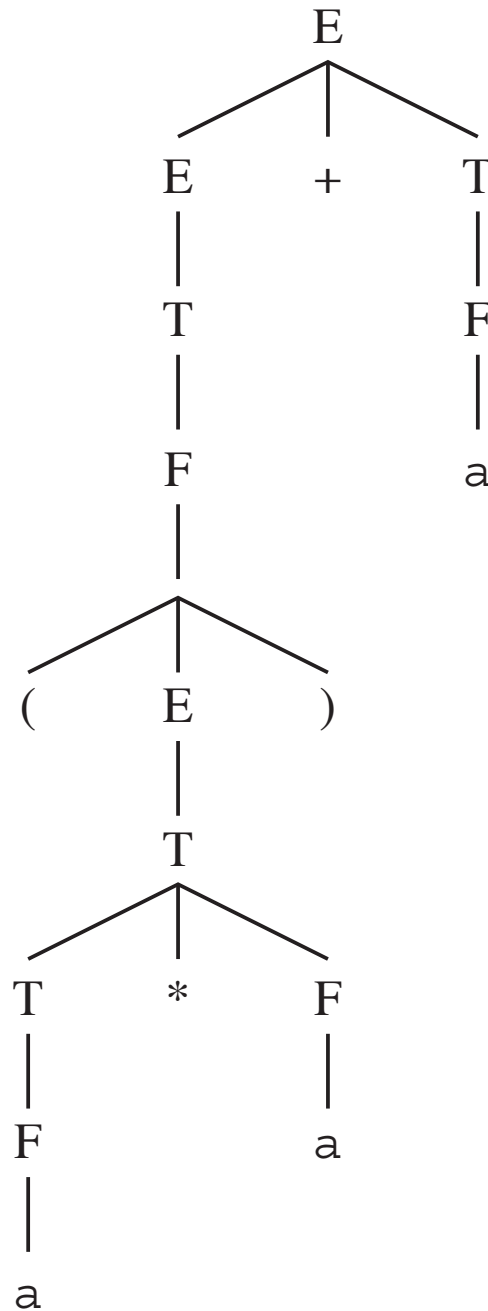
$$3. T \rightarrow T * F$$

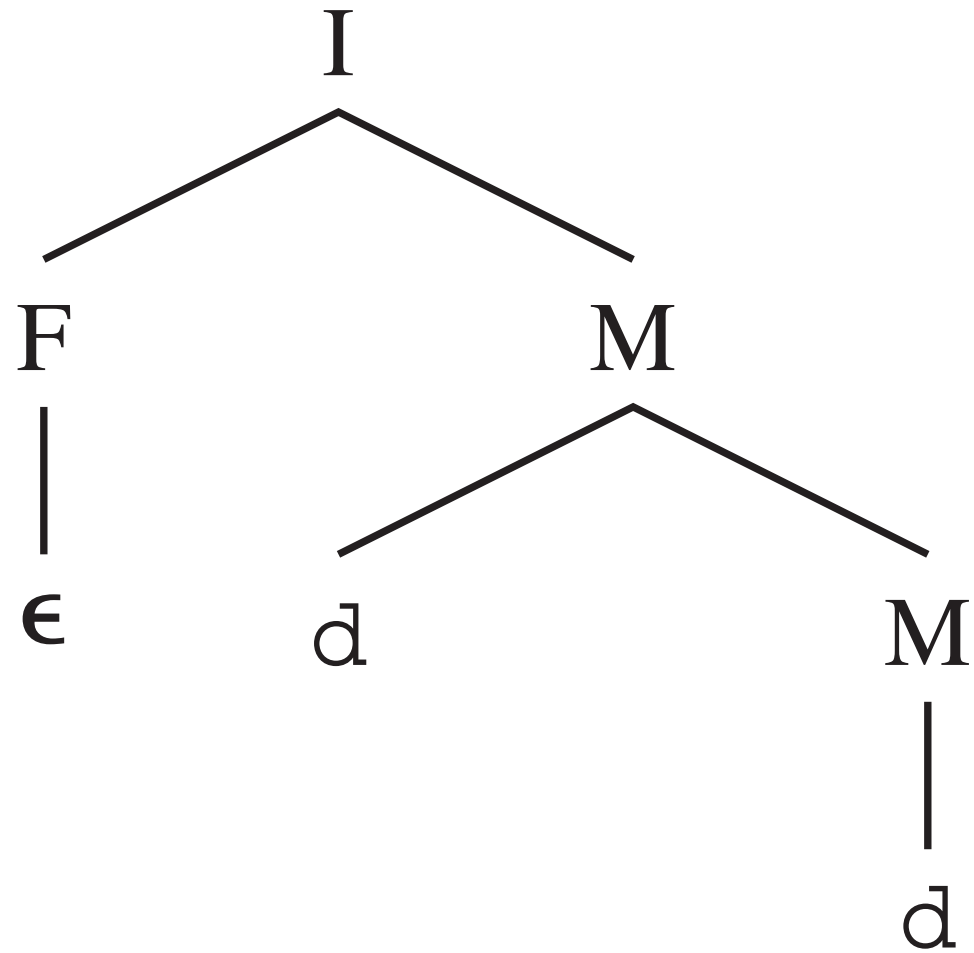
$$4. T \rightarrow F$$

$$5. F \rightarrow ( E )$$

$$6. F \rightarrow a$$

$$S = E$$





`<translation-unit> →`  
    `<external-declaration>`  
    | `<translation-unit> <external-declaration>`

`<external-declaration> →`  
    `<function-definition>`  
    | `<declaration>`

`<function-definition> →`  
    `<type-specifier> <identifier> ( <parameter-list> ) <compound-statement>`  
    | `<identifier> ( <parameter-list> ) <compound-statement>`

`<declaration> → <type-specifier> <declarator-list> ;`

`<type-specifier> → void | char | int`

`<declarator-list> →`  
    `<identifier>`  
    | `<declarator-list> <identifier>`

<statement> →

    <compound-statement>

    | <expression-statement>

    | <selection-statement>

    | <iteration-statement>

<expression-statement> → <expression> ;

<selection-statement> →

    if ( <expression> ) <statement>

    | if ( <expression> ) <statement> else <statement>

<iteration-statement> →

    while ( <expression> ) <statement>

    | do <statement> while ( <expression> ) ;

<expression> →

    <relational-expression>

    | <identifier> = <expression>

$\langle \text{declarator-list} \rangle \rightarrow$

$\langle \text{identifier} \rangle$

|  $\langle \text{declarator-list} \rangle \langle \text{identifier} \rangle$

$\langle \text{parameter-list} \rangle \rightarrow$

$\epsilon$

|  $\langle \text{parameter-declaration} \rangle$

|  $\langle \text{parameter-list} \rangle , \langle \text{parameter-declaration} \rangle$

$\langle \text{parameter-declaration} \rangle \rightarrow \langle \text{type-specifier} \rangle \langle \text{identifier} \rangle$

$\langle \text{compound-statement} \rangle \rightarrow \{ \langle \text{declaration-list} \rangle \langle \text{statement-list} \rangle \}$

$\langle \text{declaration-list} \rangle \rightarrow$

$\epsilon$

|  $\langle \text{declaration} \rangle$

|  $\langle \text{declaration} \rangle \langle \text{declaration-list} \rangle$

$\langle \text{statement-list} \rangle \rightarrow$

$\epsilon$

|  $\langle \text{statement} \rangle$

|  $\langle \text{statement-list} \rangle \langle \text{statement} \rangle$

<relational-expression> →

<additive-expression>

| <relational-expression> < <additive-expression>

| <relational-expression> > <additive-expression>

| <relational-expression> <= <additive-expression>

| <relational-expression> >= <additive-expression>

<additive-expression> →

<multiplicative-expression>

| <additive-expression> + <multiplicative-expression>

| <additive-expression> - <multiplicative-expression>

<multiplicative-expression> →

<unary-expression>

| <multiplicative-expression> \* <unary-expression>

| <multiplicative-expression> / <unary-expression>

<unary-expression> →

<primary-expression>

| <identifier> ( <argument-expression-list> )



$\langle \text{primary-expression} \rangle \rightarrow$   
     $\langle \text{identifier} \rangle$   
    |  $\langle \text{constant} \rangle$

$\langle \text{argument-expression-list} \rangle \rightarrow$   
     $\langle \text{expression} \rangle$   
    |  $\langle \text{argument-expression-list} \rangle , \langle \text{expression} \rangle$

$\langle \text{constant} \rangle \rightarrow$   
     $\langle \text{integer-constant} \rangle$   
    |  $\langle \text{character-constant} \rangle$

$\langle \text{integer-constant} \rangle \rightarrow$   
     $\langle \text{digit} \rangle$   
    |  $\langle \text{integer-constant} \rangle \langle \text{digit} \rangle$

$\langle \text{character-constant} \rangle \rightarrow ' \langle \text{letter} \rangle '$



<identifier> →

<letter>

| <identifier> <letter>

| <identifier> <digit>

<letter> →

a | b | c | d | e | f | g | h | i | j | k | l | m |

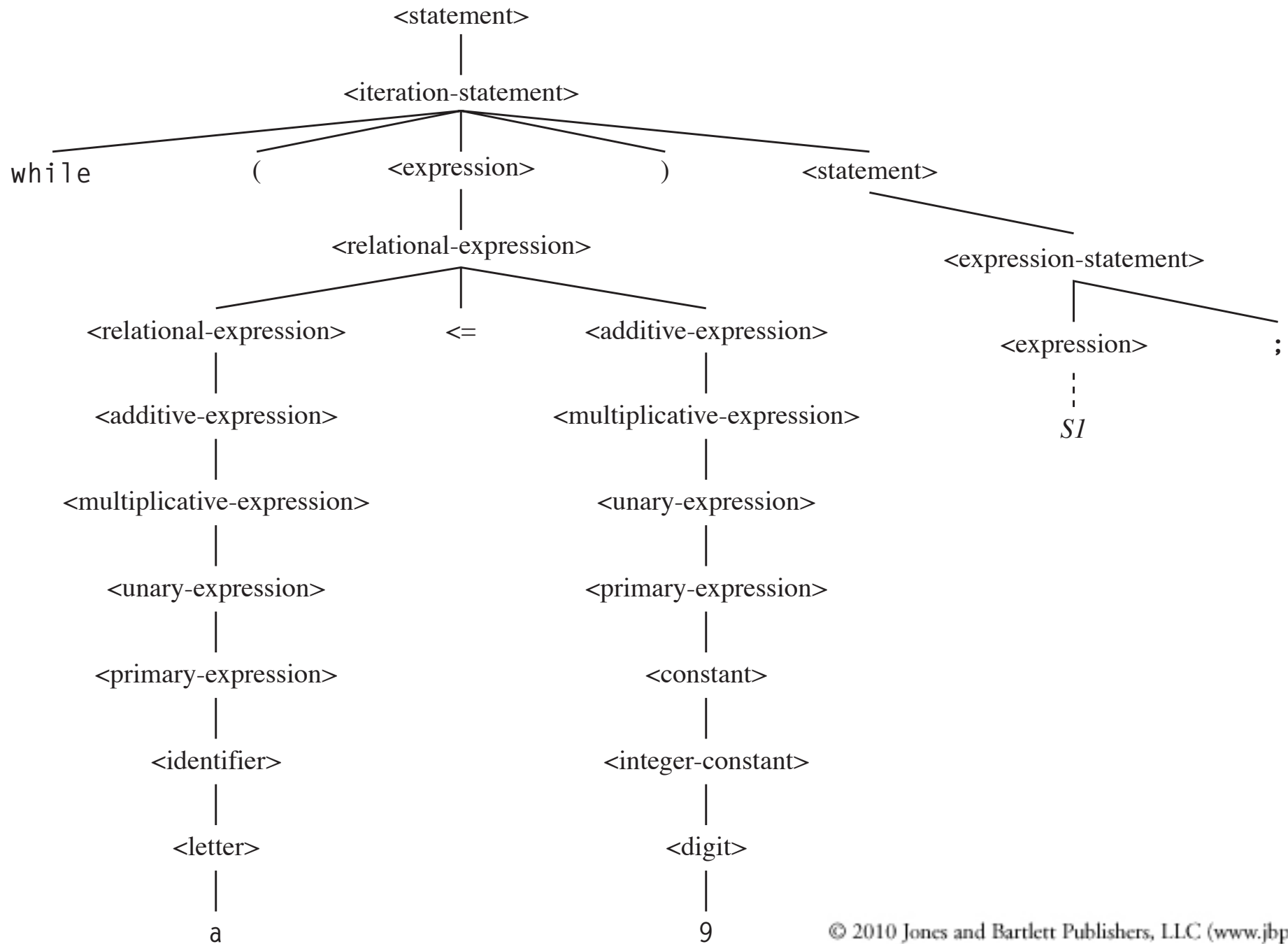
n | o | p | q | r | s | t | u | v | w | x | y | z |

A | B | C | D | E | F | G | H | I | J | K | L | M |

N | O | P | Q | R | S | T | U | V | W | X | Y | Z

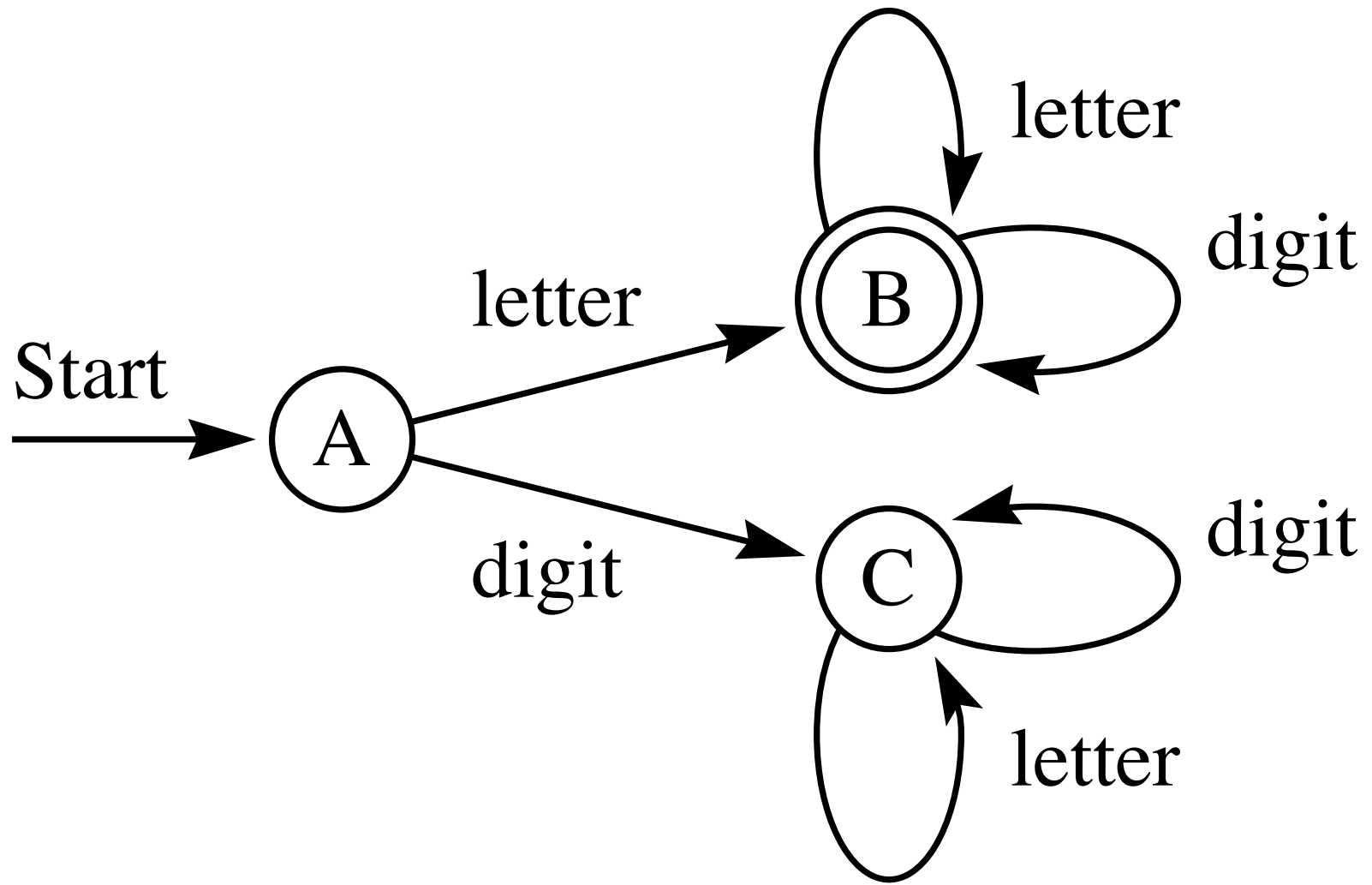
<digit> →

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



# Finite state machines

- Finite set of states called nodes represented by circles
- Transitions between states represented by directed arcs
- Each arc labeled by a terminal character
- One state designated the start state
- A nonempty set of states designated final states



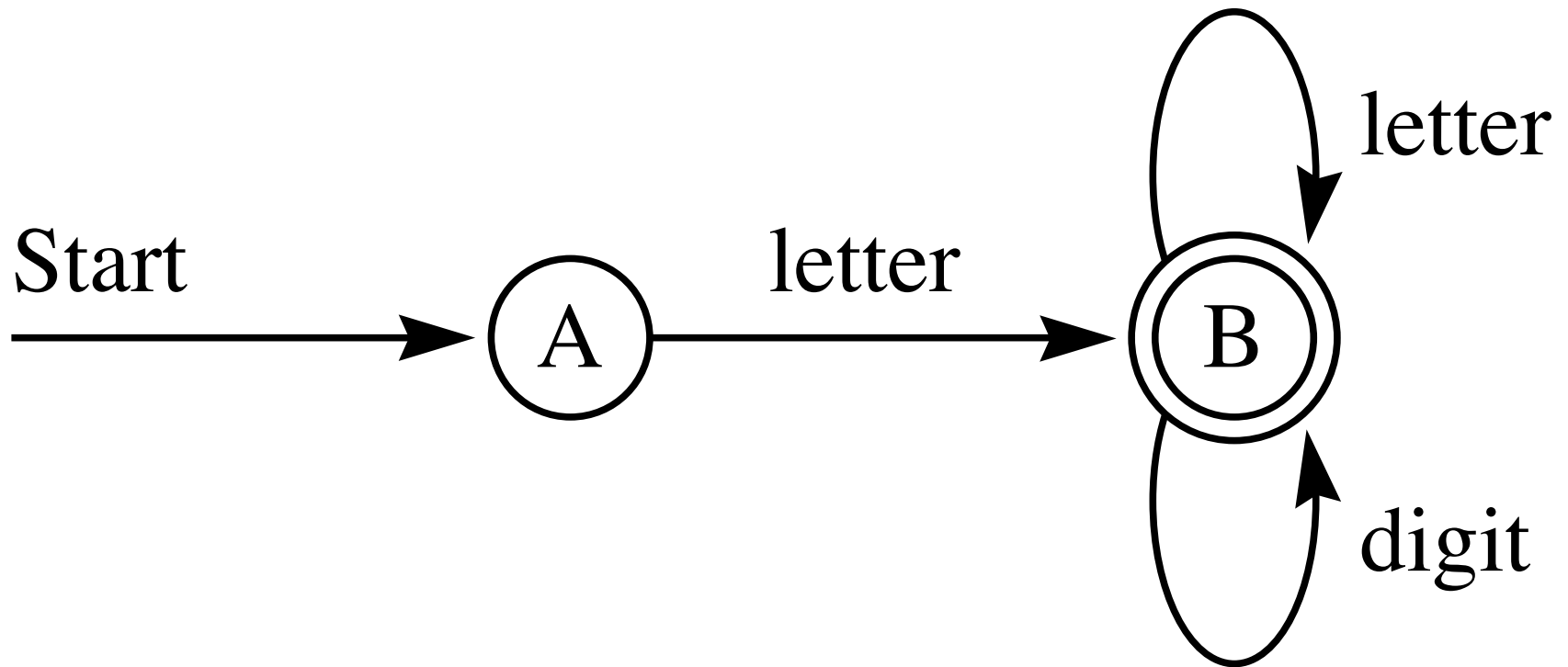
# Parsing rules

- Start at the start state
- Scan the string from left to right
- For each terminal scanned, make a transition to the next state in the FSM
- After the last terminal scanned, if you are in a final state the string is in the language
- Otherwise, it is not

Current State	Next State	
	Letter	Digit
→ A	B	C
ⓑ	B	B
C	C	C

# Simplified FSM

- Not all states have transitions on all terminal symbols
- Two ways to detect an illegal string
  - ▶ You may run out of input, and not be in a final state
  - ▶ You may be in some state, and the next input character does not correspond to any of the transitions from that state

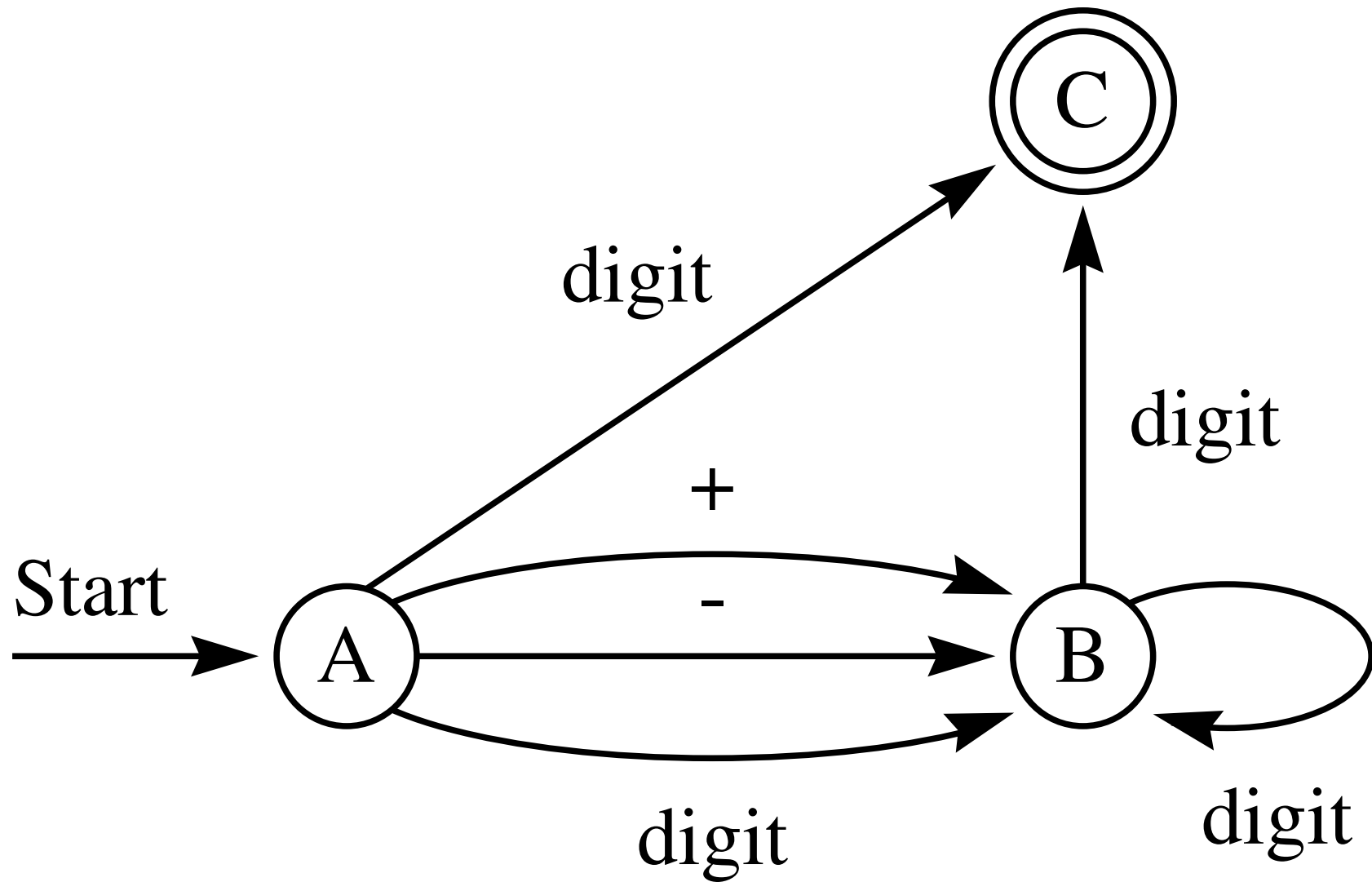




Current State	Next State	
	Letter	Digit
→ A	B	
ⓑ	B	B

# Nondeterministic FSM

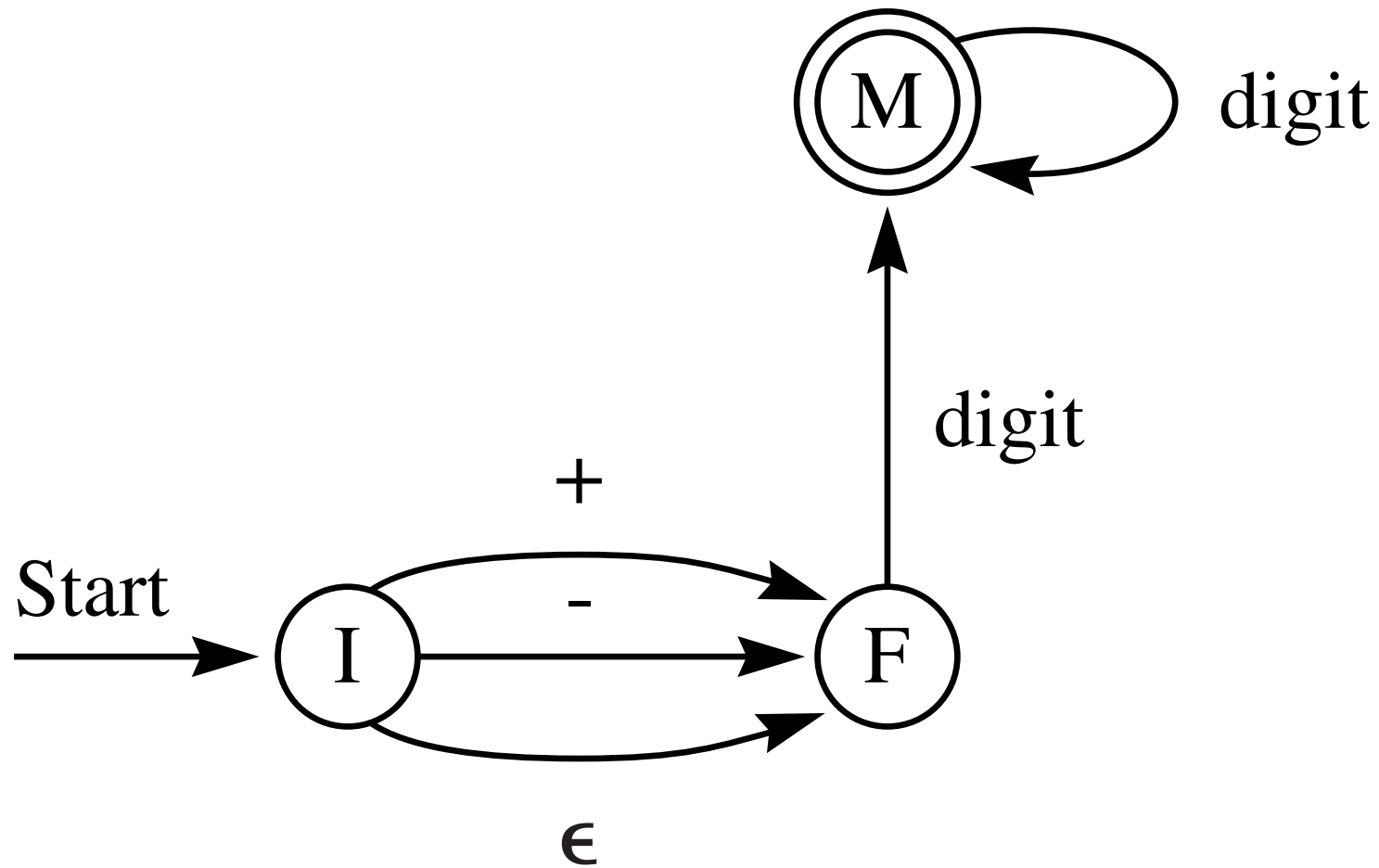
- At least one state has more than one transition from it on the same character
- If you scan the last character and you *are* in a final state, the string *is valid*
- If you scan the last character and you are *not* in a final state, the string *might be invalid*
- To prove invalid, you must try all possibilities with backtracking



Current State	Next State		
	+	-	Digit
→ A	B	B	B, C
B			B, C
Ⓒ			

# Empty transitions

- An empty transition allows you to go from one state to another state without scanning a terminal character
- All finite state machines with empty transitions are considered nondeterministic

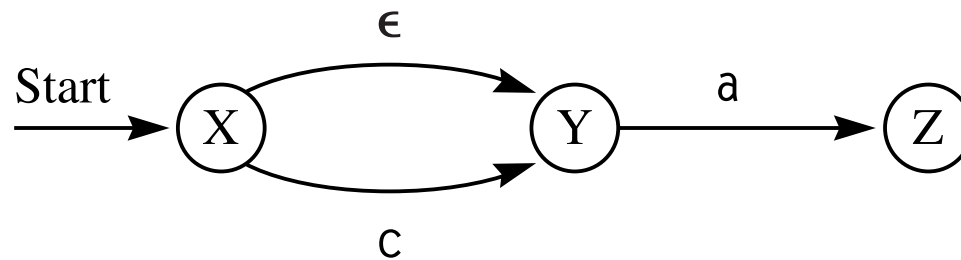


Current State	Next State			
	+	-	Digit	€
→ I	F	F		F
F			M	
Ⓜ			M	

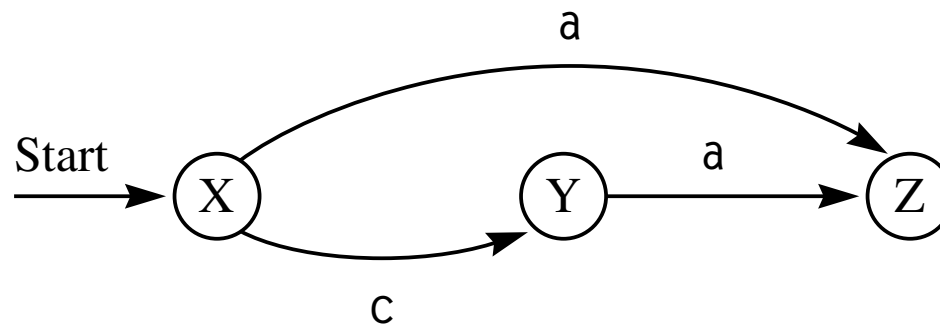
# Removing empty transitions

- Given a transition from  $p$  to  $q$  on  $\epsilon$ , for every transition from  $q$  to  $r$  on  $a$ , add a transition from  $p$  to  $r$  on  $a$ .
- If  $q$  is a final state, make  $p$  a final state

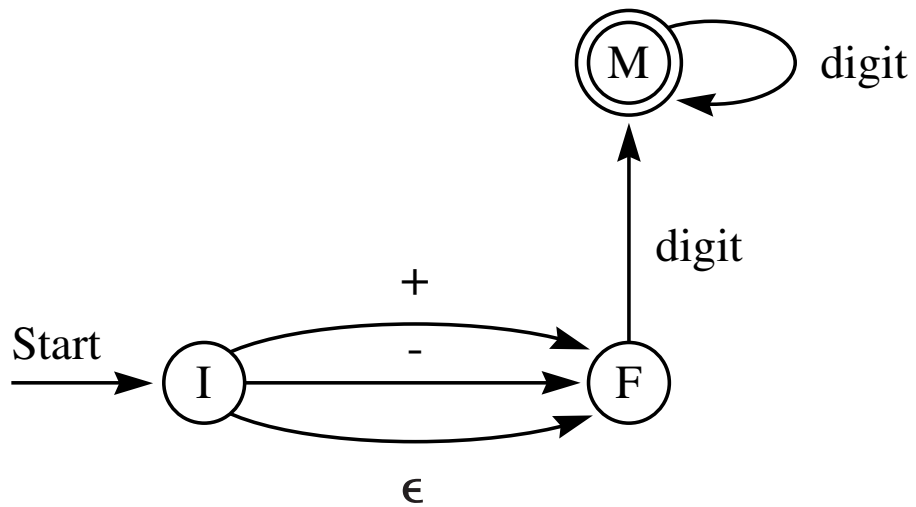




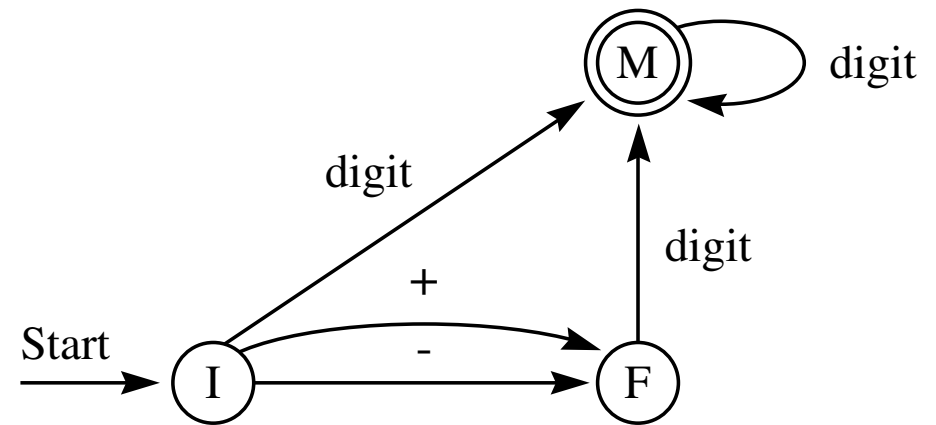
**(a)** The original FSM.



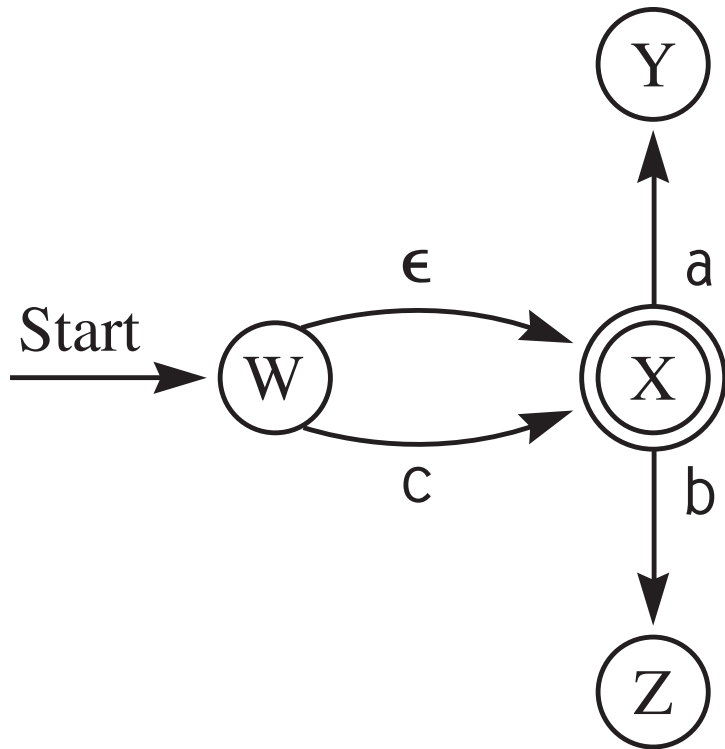
**(b)** The equivalent FSM without an empty transition.



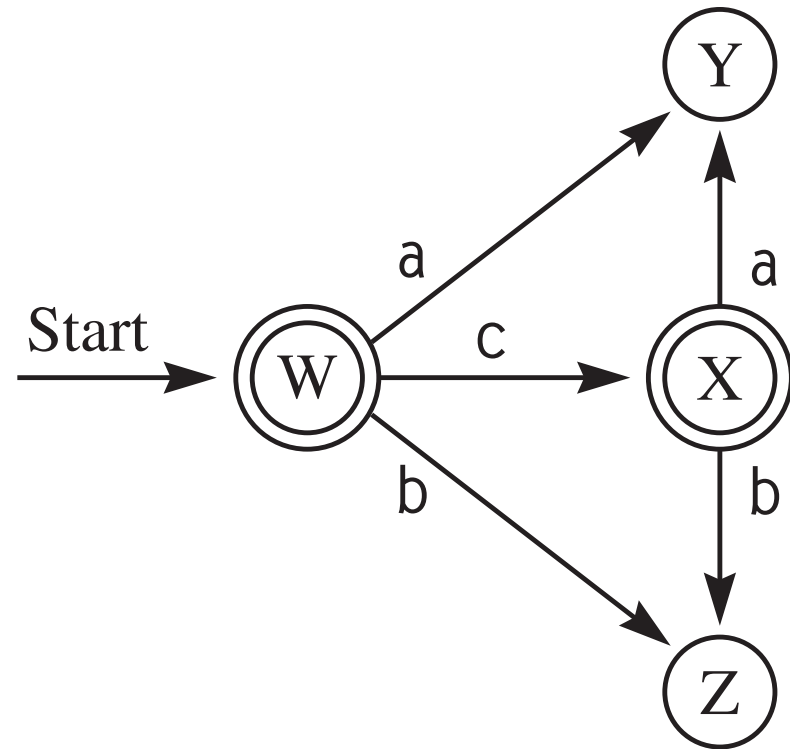
(a) The original FSM.



(b) The empty transition removed.



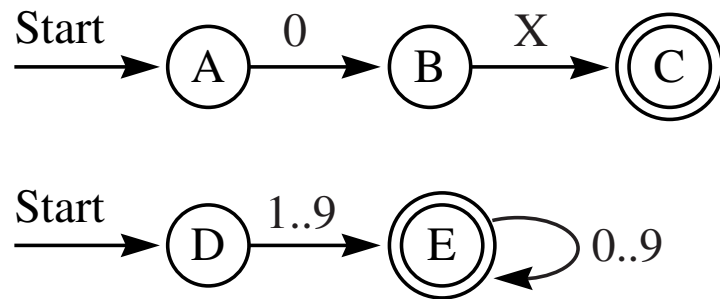
**(a)** The original FSM.



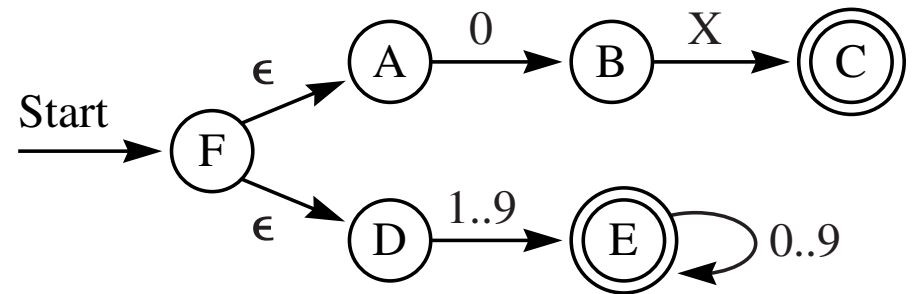
**(b)** The equivalent FSM without an empty transition.

# Multiple token recognizers

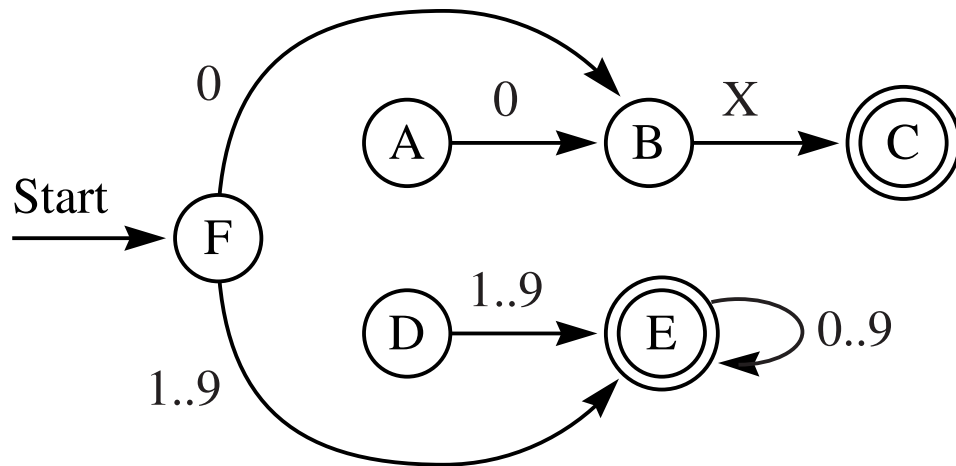
- Token
  - ▶ A string of terminal characters that has meaning as a group
- FSM with multiple final states
- The final state determines the token that is recognized



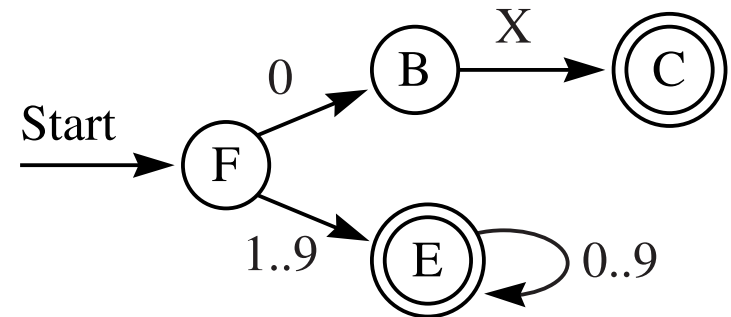
(a) Separate machines for the 0X and unsigned integer tokens.



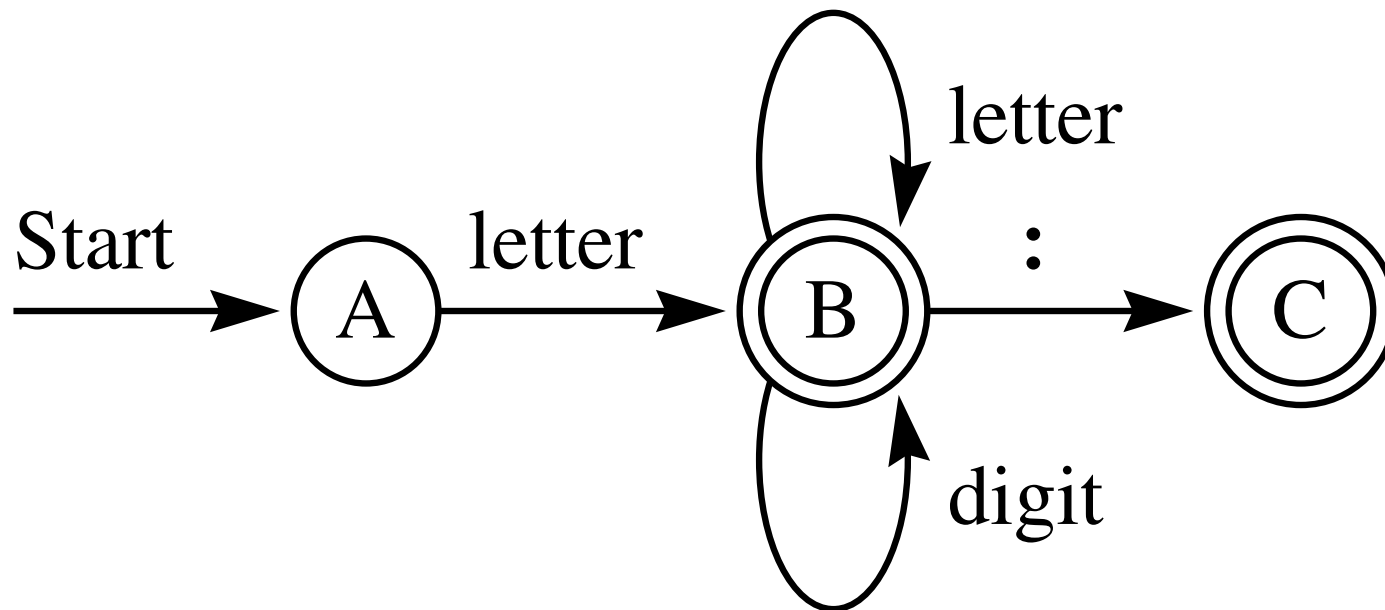
(b) One nondeterministic FSM that recognizes the 0X or unsigned integer token.

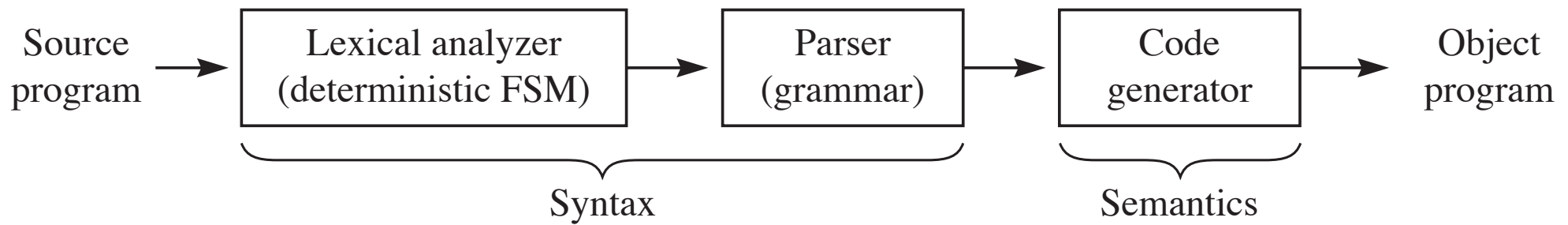


(a) Removing the empty transitions.



(b) Removing the inaccessible states.



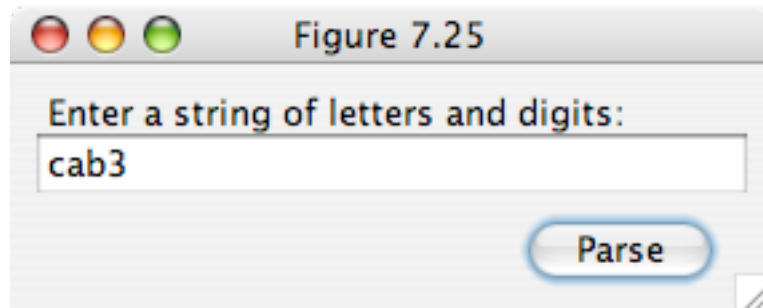




# FSM implementation techniques

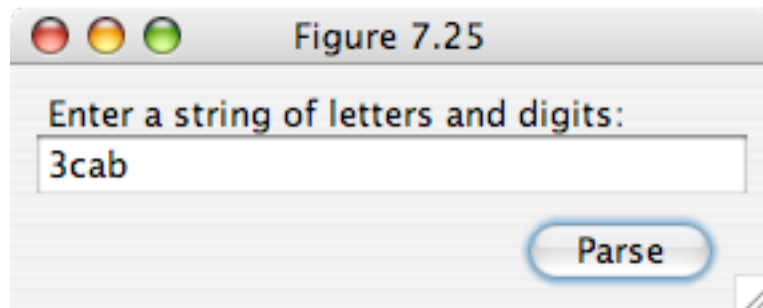
- Table-lookup
- Direct-code

Current State	Next State	
	Letter	Digit
→ A	B	C
ⓑ	B	B
C	C	C



## Console output

**cab3 is a valid identifier.**



## Console output

**3cab is not a valid identifier.**

```
package fig0725;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Figure 7.25 of J Stanley Warford, Computer Systems, Fourth edition,
 * Jones & Bartlett, 2010.
 *
 * <p>Implementation of the FSM of Figure 7.10 with the table-lookup technique.
 *
 * <p>File:
 * <code>Fig0725Main.java</code>
 */
public class Fig0725Main implements ActionListener {

    JFrame mainWindowFrame;
    JPanel inputPanel;
    JLabel label;
    JTextField textField;
    JPanel buttonPanel;
    JButton button;
```

```
public Fig0725Main() {  
    // Set up the main window.  
    mainWindowFrame = new JFrame("Figure 7.25");  
    mainWindowFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    mainWindowFrame.setSize(new Dimension(240, 120));  
  
    // Lay out the label and text field input panel from top to bottom.  
    inputPanel = new JPanel();  
    inputPanel.setLayout(new BoxLayout(inputPanel, BoxLayout.PAGE_AXIS));  
    label = new JLabel("Enter a string of letters and digits:");  
    inputPanel.add(label);  
    textField = new JTextField(20);  
    inputPanel.add(textField);  
    inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));  
  
    // Lay out the button from left to right.  
    buttonPanel = new JPanel();  
    buttonPanel.setLayout(new BoxLayout(buttonPanel, BoxLayout.LINE_AXIS));  
    buttonPanel.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));  
    buttonPanel.add(Box.createHorizontalGlue());  
    button = new JButton("Parse");  
    buttonPanel.add(button);  
    buttonPanel.add(Box.createRigidArea(new Dimension(10, 0)));  
}
```

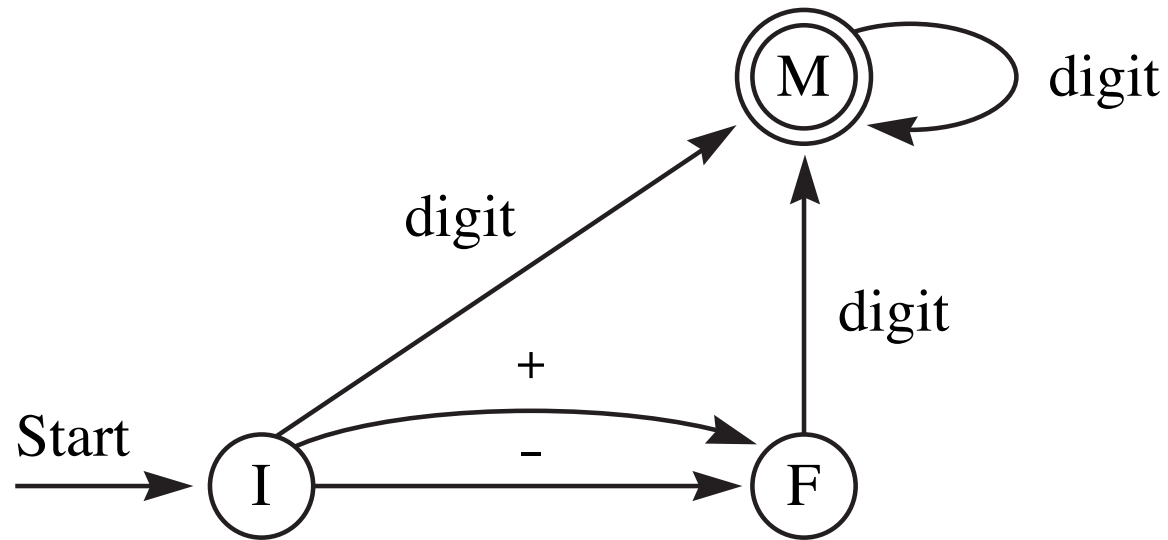
```
// Combine the input panel and the button panel in the main window.  
mainWindowFrame.add(inputPanel, BorderLayout.CENTER);  
mainWindowFrame.add(buttonPanel, BorderLayout.PAGE_END);  
  
textField.addActionListener(this);  
button.addActionListener(this);  
  
mainWindowFrame.pack();  
mainWindowFrame.setVisible(true);  
}
```

```
private static void createAndShowGUI() {  
    JFrame.setDefaultLookAndFeelDecorated(true);  
    Fig0725Main mainWindow = new Fig0725Main();  
}  
  
public static void main(String[] args) {  
    javax.swing.SwingUtilities.invokeLater(new Runnable() {  
        @Override  
        public void run() {  
            createAndShowGUI();  
        }  
    });  
}
```

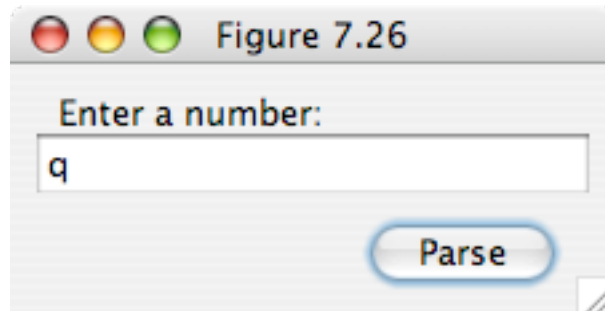
```
public static boolean isAlpha(char ch) {  
    return ('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z');  
}  
// States  
static final int S_A = 0;  
static final int S_B = 1;  
static final int S_C = 2;  
// Alphabet  
static final int T_LETTER = 0;  
static final int T_DIGIT = 1;  
// State transition table  
static final int[][] FSM = {  
    {S_B, S_C},  
    {S_B, S_B},  
    {S_C, S_C}  
};
```



```
@Override
public void actionPerformed(ActionEvent event) {
    String line = textField.getText();
    char ch;
    int FSMChar;
    int state = S_A;
    for (int i = 0; i < line.length(); i++) {
        ch = line.charAt(i);
        FSMChar = isAlpha(ch) ? T_LETTER : T_DIGIT;
        state = FSM[state][FSMChar];
    }
    if (state == S_B) {
        System.out.printf("%s is a valid identifier.\n", line);
    } else {
        System.out.printf("%s is not a valid identifier.\n", line);
    }
}
}
```

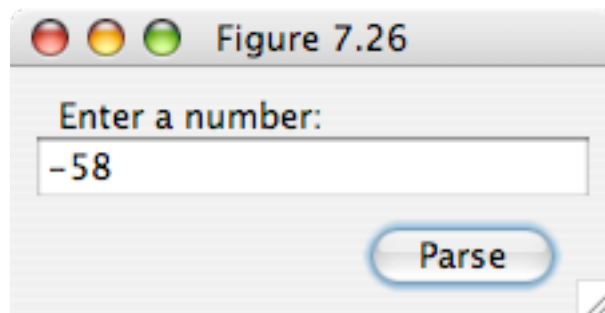


(b) The empty transition removed.



## Console output

**Invalid entry.**



## Console output

**Number = -58**

```
package fig0726;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Figure 7.26 of J Stanley Warford, Computer Systems, Fourth edition,
 * Jones & Bartlett, 2010.
 *
 * <p>A programmer-designed parse of an integer string.
 *
 * <p>File:
 * <code>Fig0726Main.java</code>
 */
public class Fig0726Main implements ActionListener {

    JFrame mainWindowFrame;
    JPanel inputPanel;
    JLabel label;
    JTextField textField;
    JPanel buttonPanel;
    JButton button;
```

```
public Fig0726Main() {  
    // Set up the main window.  
    mainWindowFrame = new JFrame("Figure 7.26");  
    mainWindowFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    mainWindowFrame.setSize(new Dimension(240, 120));  
  
    // Lay out the label and text field input panel from top to bottom.  
    inputPanel = new JPanel();  
    inputPanel.setLayout(new BoxLayout(inputPanel, BoxLayout.PAGE_AXIS));  
    label = new JLabel("Enter a number:");  
    inputPanel.add(label);  
    textField = new JTextField(20);  
    inputPanel.add(textField);  
    inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));  
  
    // Lay out the button from left to right.  
    buttonPanel = new JPanel();  
    buttonPanel.setLayout(new BoxLayout(buttonPanel, BoxLayout.LINE_AXIS));  
    buttonPanel.setBorder(BorderFactory.createEmptyBorder(0, 10, 10, 10));  
    buttonPanel.add(Box.createHorizontalGlue());  
    button = new JButton("Parse");  
    buttonPanel.add(button);  
    buttonPanel.add(Box.createRigidArea(new Dimension(10, 0)));  
}
```

```
// Combine the input panel and the button panel in the main window.  
mainWindowFrame.add(inputPanel, BorderLayout.CENTER);  
mainWindowFrame.add(buttonPanel, BorderLayout.PAGE_END);  
  
textField.addActionListener(this);  
button.addActionListener(this);  
  
mainWindowFrame.pack();  
mainWindowFrame.setVisible(true);  
}
```

```
private static void createAndShowGUI() {
    JFrame.setDefaultLookAndFeelDecorated(true);
    Fig0726Main mainWindow = new Fig0726Main();
}

public static void main(String[] args) {
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            createAndShowGUI();
        }
    });
}

public void actionPerformed(ActionEvent event) {
    String line = textField.getText();
    Parser parser = new Parser();
    parser.parseNum(line);
    if (parser.isValid()) {
        System.out.printf("Number = %d\n", parser.getNumber());
    } else {
        System.out.printf("Invalid entry.\n");
    }
}
}
```



```
package fig0726;

/**
 * The enumerated values for the states of the finite state machine.
 *
 * <p>File:
 * <code>State.java</code>
 */
public enum State {

    S_I, S_F, S_M, S_STOP
}
```



```
package fig0726;

/**
 * The direct code implementation of a finite state machine to parse
 * a signed integer.
 *
 * <p>File:
 * <code>Parser.java</code>
 */
public class Parser {

    private boolean valid = false;
    private int number = 0;

    public boolean getValid() {
        return valid;
    }

    public int getNumber() {
        return number;
    }

    public boolean isDigit(char ch) {
        return ('0' <= ch) && (ch <= '9');
    }
}
```

```
public void parseNum(String line) {
    line = line + '\n';
    int lineIndex = 0;
    char nextChar;
    int sign = +1;
    valid = true;
    State state = State.S_I;
    do {
        nextChar = line.charAt(lineIndex++);
        switch (state) {
            case S_I:
                if (nextChar == '+') {
                    sign = +1;
                    state = State.S_F;
                } else if (nextChar == '-') {
                    sign = -1;
                    state = State.S_F;
                } else if (isDigit(nextChar)) {
                    sign = +1;
                    number = nextChar - '0';
                    state = State.S_M;
                } else {
                    valid = false;
                }
            break;
        }
    }
```

```
case S_F:
    if (isDigit(nextChar)) {
        number = nextChar - '0';
        state = State.S_M;
    } else {
        valid = false;
    }
    break;
case S_M:
    if (isDigit(nextChar)) {
        number = 10 * number + nextChar - '0';
    } else if (nextChar == '\n') {
        number = sign * number;
        state = State.S_STOP;
    } else {
        valid = false;
    }
    break;
}
} while ((state != State.S_STOP) && valid);
}
}
```

# An input buffer

- Used to process one character at a time from a Java String as if from an input stream
- Provides a special feature needed by multiple-token parsers
- Ability to back up a character into the input stream after being scanned

```
package fig0731;

/**
 * The input buffer, which processes \n delimited lines.
 *
 * <p>File:
 * <code>InBuffer.java</code>
 */
public class InBuffer {

    private String inString;
    private String line;
    private int lineIndex;

    public InBuffer(String string) {
        inString = string + "\n\n";
        // To guarantee inString.length() == 0 eventually
    }
}
```

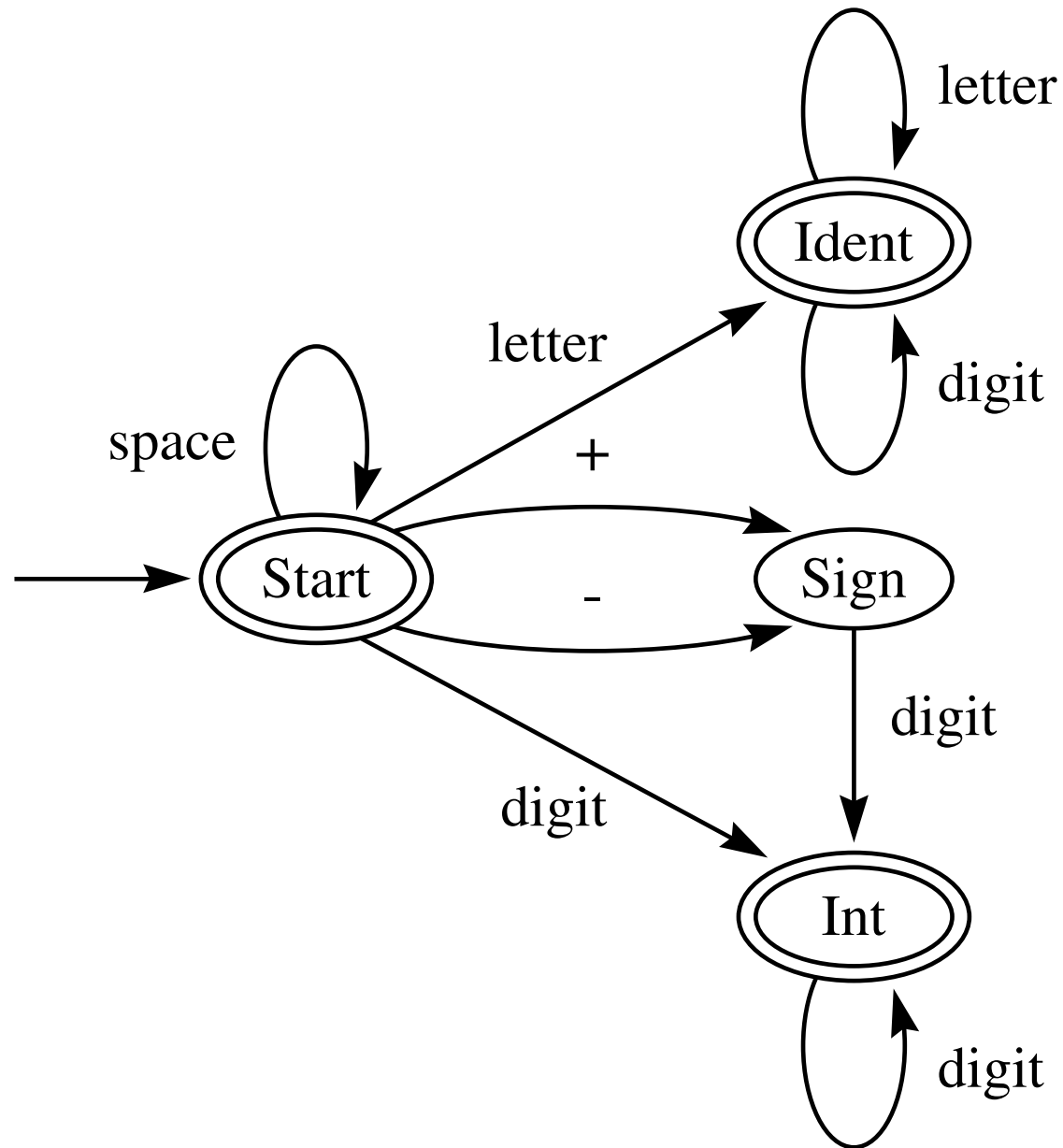
```
public void getLine() {  
    int i = inString.indexOf('\n');  
    line = inString.substring(0, i + 1);  
    inString = inString.substring(i + 1);  
    lineIndex = 0;  
}
```

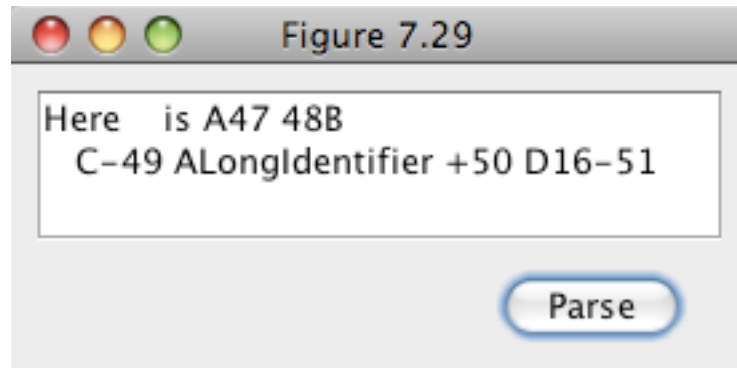
```
public boolean inputRemains() {  
    return inString.length() != 0;  
}
```

```
public char advanceInput() {  
    return line.charAt(lineIndex++);  
}
```

```
public void backUpInput() {  
    lineIndex--;  
}
```

```
}
```

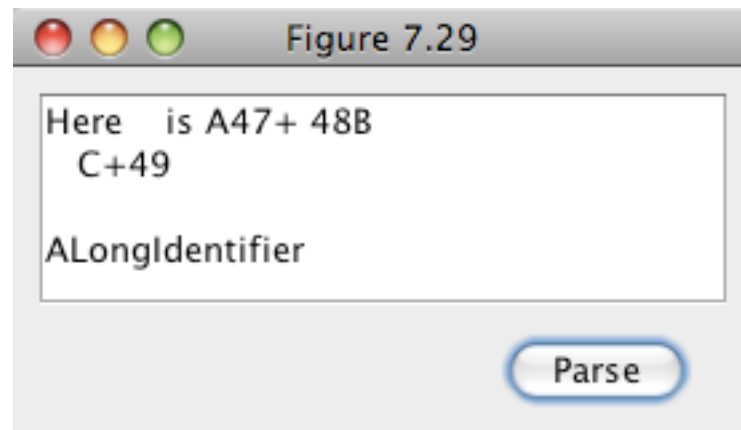




## Console output

```
Identifier = Here
Identifier = is
Identifier = A47
Integer    = 48
Identifier = B
Empty token
Identifier = C
Integer    = -49
Identifier = ALongIdentifier
Integer    = 50
Identifier = D16
Integer    = -51
Empty token
```





## Console output

```
Identifier = Here
Identifier = is
Identifier = A47
Syntax error
Identifier = C
Integer    = 49
Empty token
Empty token
Identifier = ALongIdentifier
Empty token
```

```
package fig0731;

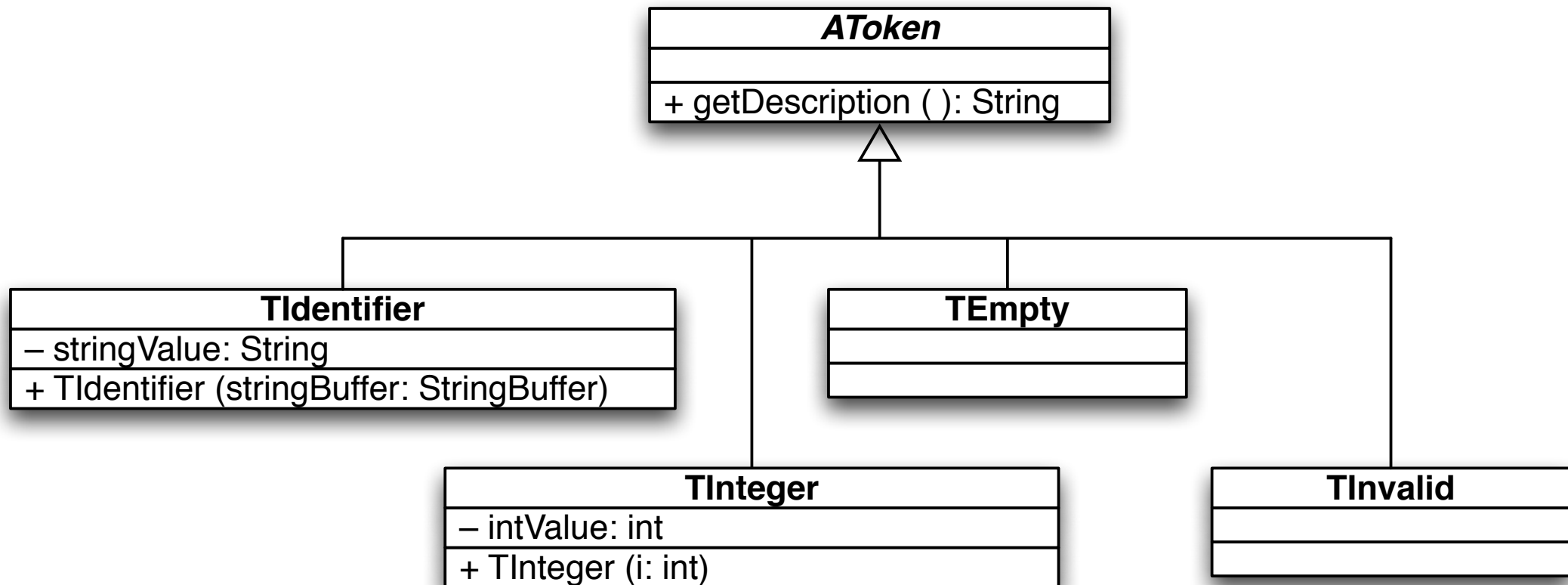
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Figure 7.31 of J Stanley Warford, Computer Systems, Fourth edition,
 * Jones & Bartlett, 2010.
 *
 * A multiple-token parser that recognizes identifiers and integers.
 *
 * <p>File:
 * <code>Fig0731Main.java</code>
 */
public class Fig0731Main implements ActionListener {

...

```

```
public void actionPerformed(ActionEvent event) {  
    InBuffer inBuffer = new InBuffer(textArea.getText());  
    Tokenizer t = new Tokenizer(inBuffer);  
    AToken aToken;  
    inBuffer.getLine();  
    while (inBuffer.inputRemains()) {  
        do {  
            aToken = t.getToken();  
            System.out.println(aToken.getDescription());  
        } while (!(aToken instanceof TEmpty) && !(aToken instanceof TInvalid));  
        inBuffer.getLine();  
    }  
}
```



```
package fig0731;

/**
 * The abstract token class.
 *
 * <p>File:
 * <code>AToken.java</code>
 */
abstract public class AToken {

    public abstract String getDescription();
}
```

```
package fig0731;

/**
 * The empty token subclass of the abstract token.
 *
 * <p>File:
 * <code>TEmpy.java</code>
 */
public class TEmpty extends AToken {

    public String getDescription() {
        return "Empty token";
    }
}
```

```
package fig0731;

/**
 * The invalid token subclass of the abstract token.
 *
 * <p>File:
 * <code>TInvalid.java</code>
 */
public class TInvalid extends AToken {

    public String getDescription() {
        return "Syntax error";
    }
}
```

```
package fig0731;

/**
 * The integer token subclass of the abstract token.
 *
 * <p>File:
 * <code>TInteger.java</code>
 */
public class TInteger extends AToken {

    private int intValue;

    public TInteger(int i) {
        intValue = i;
    }

    public String getDescription() {
        return String.format("Integer      = %d", intValue);
    }
}
```



```
package fig0731;

/**
 * The identifier token subclass of the abstract token.
 *
 * <p>File:
 * <code>TIdentifier.java</code>
 */
public class TIdentifier extends AToken {

    private String stringValue;

    public TIdentifier(StringBuffer stringBuffer) {
        stringValue = new String(stringBuffer);
    }

    public String getDescription() {
        return String.format("Identifier = %s", stringValue);
    }
}
```

```
package fig0731;

/**
 * Utility functions to test for digit characters and alphabetic characters.
 *
 * <p>File:
 * <code>Util.java</code>
 */
public class Util {

    public static boolean isDigit(char ch) {
        return ('0' <= ch) && (ch <= '9');
    }

    public static boolean isAlpha(char ch) {
        return (('a' <= ch) && (ch <= 'z') || ('A' <= ch) && (ch <= 'Z'));
    }
}
```

```
package fig0731;

/**
 * The enumerated values for the states of the finite state machine.
 *
 * <p>File:
 * <code>State.java</code>
 */
public enum State {

    S_START, S_IDENT, S_SIGN, S_INTEGER, S_STOP
}
```

```
package fig0731;

/**
 * The tokenizer implemented as a finite state machine.
 *
 * <p>File:
 * <code>Tokenizer.java</code>
 */
public class Tokenizer {

    private InBuffer b;

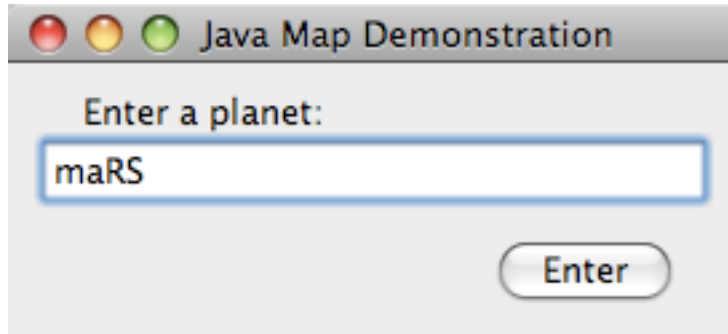
    public Tokenizer(InBuffer inBuffer) {
        b = inBuffer;
    }

    public AToken getToken() {
        char nextChar;
        StringBuffer localStringValue = new StringBuffer("");
        int localIntValue = 0;
        int sign = +1;
        AToken aToken = new TEmpty();
        State state = State.S_START;
```

```
do {
    nextChar = b.advanceInput();
    switch (state) {
        case S_START:
            if (Util.isAlpha(nextChar)) {
                localStringValue.append(nextChar);
                state = State.S_IDENT;
            } else if (nextChar == '-') {
                sign = -1;
                state = State.S_SIGN;
            } else if (nextChar == '+') {
                sign = +1;
                state = State.S_SIGN;
            } else if (Util.isDigit(nextChar)) {
                localIntValue = nextChar - '0';
                state = State.S_INTEGER;
            } else if (nextChar == '\n') {
                state = State.S_STOP;
            } else if (nextChar != ' ') {
                aToken = new TInvalid();
            }
        }
    break;
}
```

```
case S_IDENT:
    if (Util.isAlpha(nextChar) || Util.isDigit(nextChar)) {
        localStringValue.append(nextChar);
    } else {
        b.backUpInput();
        aToken = new TIdentifier(localStringValue);
        state = State.S_STOP;
    }
    break;
case S_SIGN:
    if (Util.isDigit(nextChar)) {
        localIntValue = 10 * localIntValue + nextChar - '0';
        state = State.S_INTEGER;
    } else {
        aToken = new TInvalid();
    }
    break;
```

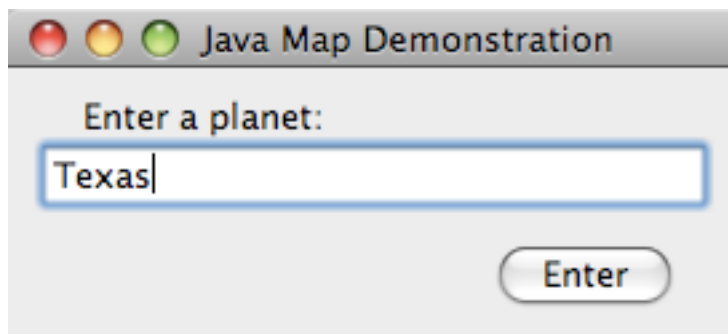
```
case S_INTEGER:
    if (Util.isDigit(nextChar)) {
        localIntValue = 10 * localIntValue + nextChar - '0';
    } else {
        b.backUpInput();
        aToken = new TInteger(sign * localIntValue);
        state = State.S_STOP;
    }
    break;
}
} while ((state != State.S_STOP) && !(aToken instanceof TInvalid));
return aToken;
}
}
```



A screenshot of a Java application window titled "Java Map Demonstration". It features a label "Enter a planet:" above a text input field containing "maRS". A button labeled "Enter" is positioned to the right of the input field.

## Console output

**Planet Mars is red.  
Enumerated output: P\_MARS  
Ordinal output: 3**



A screenshot of the same Java application window titled "Java Map Demonstration". The text input field now contains "Texas". The "Enter" button remains to the right.

## Console output

**Texas is not a planet.**



```
package mapdemo;

/**
 * The enumerated values for the planets in the map demo.
 *
 * <p>File:
 * <code>Planet.java</code>
 */
public enum Planet {

    P_MERCURY, P_VENUS, P_EARTH, P_MARS, P_JUPITER, P_SATURN,
    P_URANUS, P_NEPTUNE, P_PLUTO
}
```

```
package mapdemo;

import java.util.EnumMap;
import java.util.HashMap;
import java.util.Map;

/**
 * Two maps between string values and enumerated values.
 *
 * <p>File:
 * <code>Maps.java</code>
 */
public class Maps {

    public static final Map<String, Planet> planetTable;
    public static final Map<Planet, String> planetStringTable;
```

```
static {  
    planetTable = new HashMap<String, Planet>();  
    planetTable.put("mercury", Planet.P_MERCURY);  
    planetTable.put("venus", Planet.P_VENUS);  
    planetTable.put("earth", Planet.P_EARTH);  
    planetTable.put("mars", Planet.P_MARS);  
    planetTable.put("jupiter", Planet.P_JUPITER);  
    planetTable.put("saturn", Planet.P_SATURN);  
    planetTable.put("uranus", Planet.P_URANUS);  
    planetTable.put("neptune", Planet.P_NEPTUNE);  
    planetTable.put("pluto", Planet.P_PLUTO);  
  
    planetStringTable = new EnumMap<Planet, String>(Planet.class);  
    planetStringTable.put(Planet.P_MERCURY, "Mercury");  
    planetStringTable.put(Planet.P_VENUS, "Venus");  
    planetStringTable.put(Planet.P_EARTH, "Earth");  
    planetStringTable.put(Planet.P_MARS, "Mars");  
    planetStringTable.put(Planet.P_JUPITER, "Jupiter");  
    planetStringTable.put(Planet.P_SATURN, "Saturn");  
    planetStringTable.put(Planet.P_URANUS, "Uranus");  
    planetStringTable.put(Planet.P_NEPTUNE, "Neptune");  
    planetStringTable.put(Planet.P_PLUTO, "Pluto");  
}
```

```
package mapdemo;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A demonstration of Java maps and enumerated types.
 *
 * <p>File:
 * <code>MapDemoMain.java</code>
 */
public class MapDemoMain implements ActionListener {

    JFrame mainWindowFrame;
    JPanel inputPanel;
    JLabel label;
    JTextField textField;
    JButton button;

    public MapDemoMain() {

...

```

```
public void actionPerformed(ActionEvent event) {  
    String line = textField.getText();  
    if (Maps.planetTable.containsKey(line.toLowerCase())) {  
        Planet planet = Maps.planetTable.get(line.toLowerCase());  
        String planetString = Maps.planetStringTable.get(planet);  
        switch (planet) {  
            case P_MERCURY:  
            case P_VENUS:  
                System.out.printf("%s is close to the sun.\n", planetString);  
                break;  
            case P_EARTH:  
                System.out.printf("The %s is indeed a planet.\n", planetString);  
                break;  
            case P_MARS:  
                System.out.printf("Planet %s is red.\n", planetString);  
                break;  
        }  
    }  
}
```

```
        case P_JUPITER:
        case P_SATURN:
            System.out.printf("%s is a big planet.\n", planetString);
            break;
        case P_URANUS:
        case P_NEPTUNE:
        case P_PLUTO:
            System.out.printf("%s is far from the sun.\n", planetString);
    }
    System.out.printf("Enumerated output: %s\n", planet);
    System.out.printf("Ordinal output: %d\n", planet.ordinal());
} else {
    System.out.println(line + " is not a planet.");
}
}
```

## Input

```
set (Time, 15)
neg ( Time)
abs(Time )

set (Accel, 3)
set (TSquared, Time)
MUL (TSquared, Time)
set (Position, TSquared)
mul(Position , Accel)
div (Position,2 )
Stop
End
```

## Output

Object code:

```
Time := 15
Time := -Time
Time := |Time|
Accel := 3
TSquared := Time
TSquared := TSquared * Time
Position := TSquared
Position := Position * Accel
Position := Position / 2
stop
```

Program listing:

```
set (Time, 15)
neg (Time)
abs (Time)

set (Accel, 3)
set (TSquared, Time)
mul (TSquared, Time)
set (Position, TSquared)
mul (Position, Accel)
div (Position, 2)
stop
end
```



## Input

```
set (Alpha,, 123)
set (Alpha)
sit (Alpha, 123)
set, (Alpha)
mul (Alpha, Beta
set (123, Alpha)
neg (Alpha, Beta)
set (Alpha, 123) x
```

## Output

9 errors were detected.

Program listing:

```
ERROR: Second argument not an identifier or integer.
ERROR: Comma expected after first argument.
ERROR: Line must begin with function identifier.
ERROR: Left parenthesis expected after function.
ERROR: Right parenthesis expected after argument.
ERROR: First argument not an identifier.
ERROR: Right parenthesis expected after argument.
ERROR: Illegal trailing character.
```

ERROR: Missing "end" sentinel.



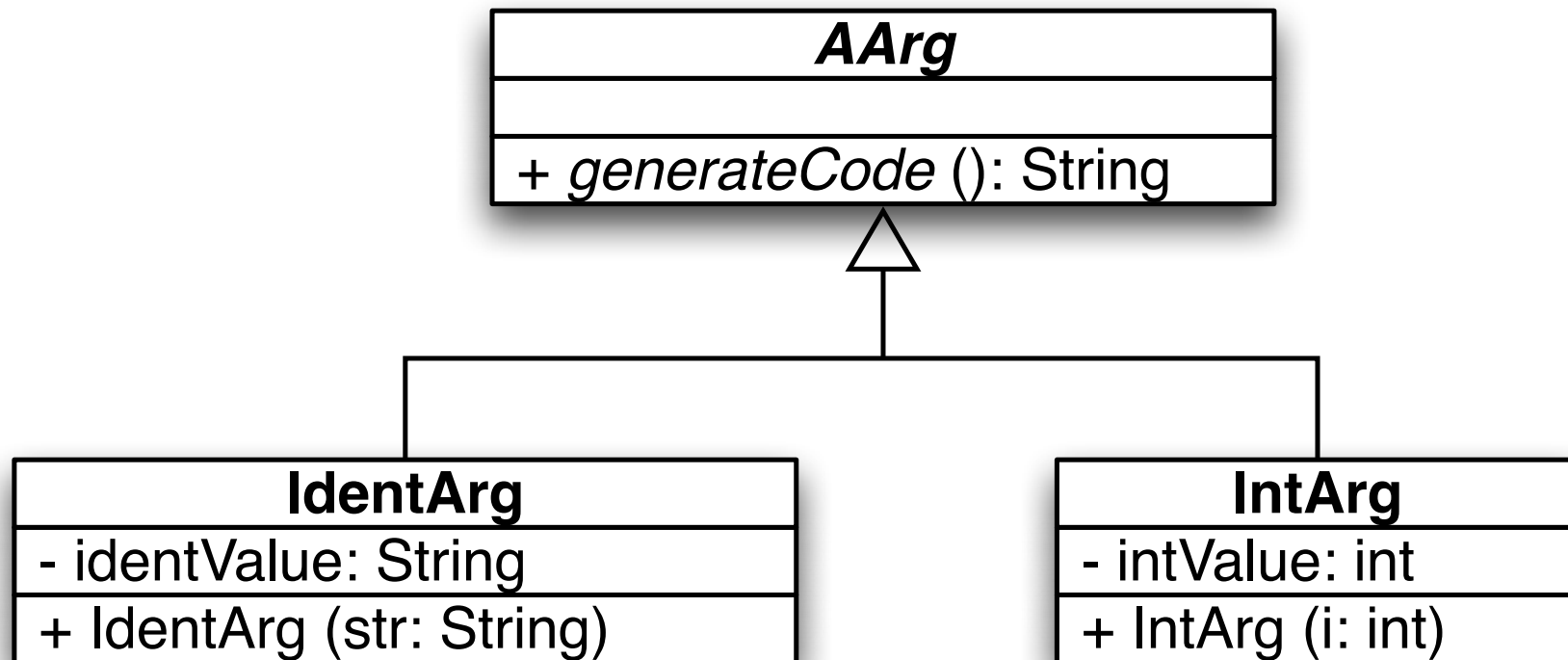
```
package fig0735;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * Figure 7.35 of J Stanley Warford, Computer Systems, Fourth edition,
 * Jones & Bartlett, 2010.
 *
 * <p>A translator with a lexical analyzer, a parser, and a code generator.
 *
 * <p>File:
 * <code>Fig0735Main.java</code>
 */
public class Fig0735Main implements ActionListener {

    ...

    public void actionPerformed(ActionEvent event) {
        InBuffer inBuffer = new InBuffer(textArea.getText());
        Translator tr = new Translator(inBuffer);
        tr.translate();
    }
}
```



```
package fig0735;

/**
 * The abstract argument class.
 *
 * <p>File:
 * <code>AArg.java</code>
 */
abstract public class AArg {

    abstract public String generateCode();
}
```

```
package fig0735;

/**
 * The identifier argument subclass of the abstract argument class.
 *
 * <p>File:
 * <code>IdentArg.java</code>
 */
public class IdentArg extends AArg {

    private String identValue;

    public IdentArg(String str) {
        identValue = str;
    }

    public String generateCode() {
        return identValue;
    }
}
```

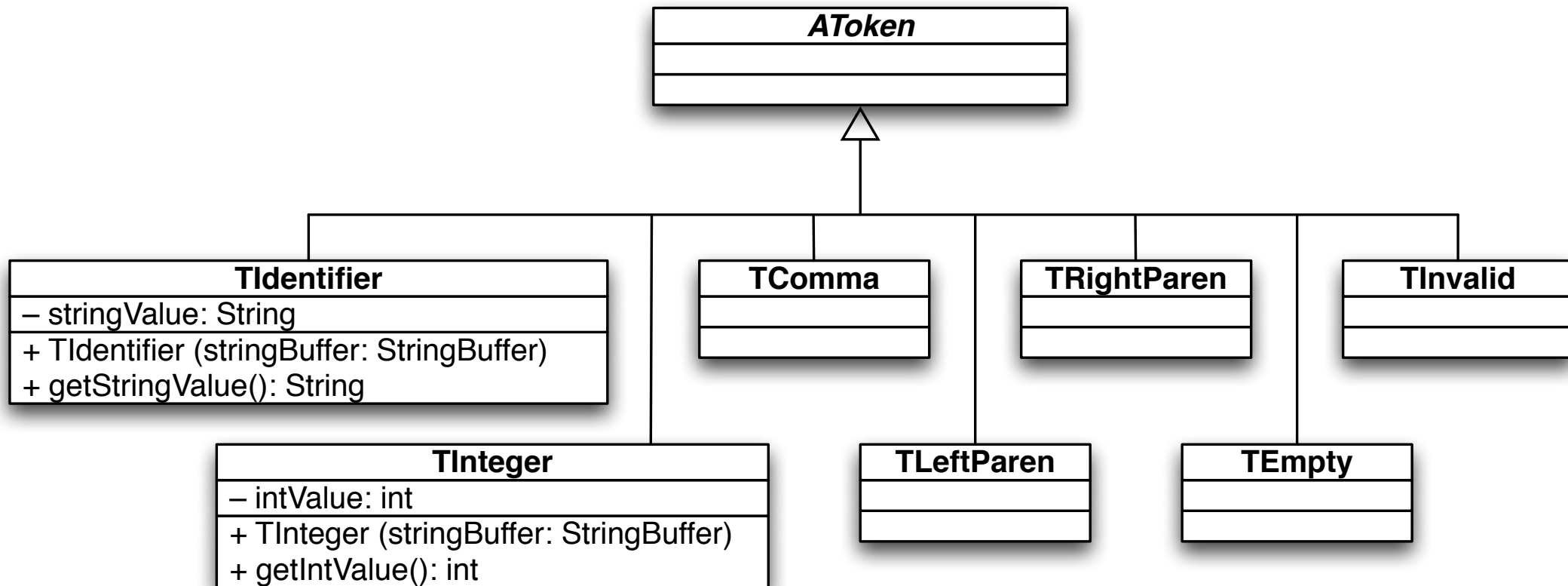
```
package fig0735;

/**
 * The integer argument subclass of the abstract argument class.
 *
 * <p>File:
 * <code>IntArg.java</code>
 */
public class IntArg extends AArg {

    private int intValue;

    public IntArg(int i) {
        intValue = i;
    }

    public String generateCode() {
        return String.format("%d", intValue);
    }
}
```



```
package fig0735;

/**
 * The abstract token class.
 *
 * <p>File:
 * <code>AToken.java</code>
 */
abstract public class AToken {
}
```

```
package fig0735;

/**
 * The identifier token subclass of the abstract token.
 *
 * <p>File:
 * <code>TIdentifier.java</code>
 */
public class TIdentifier extends AToken {

    private String stringValue;

    public TIdentifier(StringBuffer stringBuffer) {
        stringValue = new String(stringBuffer);
    }

    public String getStringValue() {
        return stringValue;
    }
}
```



```
package fig0735;

/**
 * The integer token subclass of the abstract token.
 *
 * <p>File:
 * <code>TInteger.java</code>
 */
public class TInteger extends AToken {

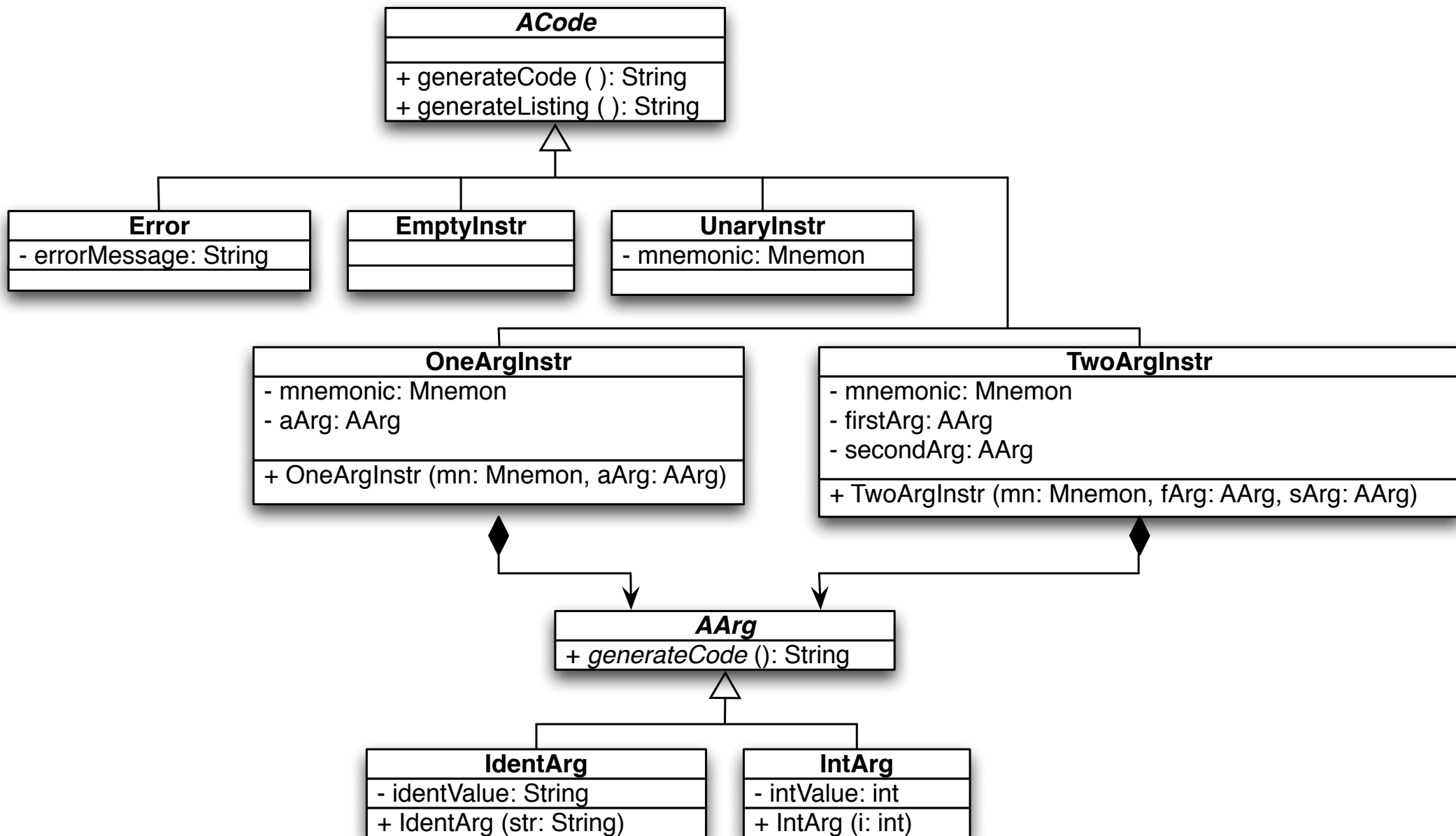
    private int intValue;

    public TInteger(int i) {
        intValue = i;
    }

    public int getIntValue() {
        return intValue;
    }
}
```

```
package fig0735;

/**
 * The comma token subclass of the abstract token.
 *
 * <p>File:
 * <code>TComma.java</code>
 */
public class TComma extends AToken {
}
```



```
package fig0735;

/**
 * The abstract code class.
 *
 * <p>File:
 * <code>ACode.java</code>
 */
abstract public class ACode {

    abstract public String generateCode();

    abstract public String generateListing();
}
```

```
package fig0735;

/**
 * The error code subclass of the abstract code class.
 *
 * <p>File:
 * <code>Error.java</code>
 */
public class Error extends ACode {

    private String errorMessage;

    public Error(String errMessage) {
        errorMessage = errMessage;
    }

    public String generateListing() {
        return "ERROR: " + errorMessage + "\n";
    }

    public String generateCode() {
        return "";
    }
}
```

```
package fig0735;

/**
 * The empty instruction code subclass of the abstract code class.
 *
 * <p>File:
 * <code>EmptyInstr.java</code>
 */
public class EmptyInstr extends ACode {
    // For an empty source line.

    public String generateListing() {
        return "\n";
    }

    public String generateCode() {
        return "";
    }
}
```

```
package fig0735;

/**
 * The unary instruction subclass of the abstract code class.
 *
 * <p>File:
 * <code>UnaryInstr.java</code>
 */
public class UnaryInstr extends ACode {

    private Mnemon mnemonic;

    public UnaryInstr(Mnemon mn) {
        mnemonic = mn;
    }
}
```

```
public String generateListing() {
    return Maps.mnemonStringTable.get(mnemonic) + "\n";
}

public String generateCode() {
    switch (mnemonic) {
        case M_STOP:
            return "stop\n";
        case M_END:
            return "";
        default:
            return ""; // Should not occur.
    }
}
}
```



```
package fig0735;

/**
 * The one-argument instruction subclass of the abstract code class.
 *
 * <p>File:
 * <code>OneArgInstr.java</code>
 */
public class OneArgInstr extends ACode {

    private Mnemon mnemonic;
    private AArg aArg;

    public OneArgInstr(Mnemon mn, AArg aArg) {
        mnemonic = mn;
        this.aArg = aArg;
    }
}
```

```
public String generateListing() {
    return String.format("%s (%s)\n",
        Maps.mnemonStringTable.get(mnemonic), aArg.generateCode());
}

public String generateCode() {
    switch (mnemonic) {
        case M_ABS:
            return String.format("%s := |%s|\n",
                aArg.generateCode(), aArg.generateCode());
        case M_NEG:
            return String.format("%s := -%s\n",
                aArg.generateCode(), aArg.generateCode());
        default:
            return ""; // Should not occur.
    }
}
```

```
package fig0735;

/**
 * The two-argument instruction subclass of the abstract code class.
 *
 * <p>File:
 * <code>TwoArgInstr.java</code>
 */
public class TwoArgInstr extends ACode {

    private Mnemon mnemonic;
    private AArg firstArg;
    private AArg secondArg;

    public TwoArgInstr(Mnemon mn, AArg fArg, AArg sArg) {
        mnemonic = mn;
        firstArg = fArg;
        secondArg = sArg;
    }

    public String generateListing() {
        return String.format("%s (%s, %s)\n",
            Maps.mnemonStringTable.get(mnemonic),
            firstArg.generateCode(),
            secondArg.generateCode());
    }
}
```

```
public String generateCode() {  
    switch (mnemonic) {  
        case M_SET:  
            return String.format("%s := %s\n",  
                firstArg.generateCode(),  
                secondArg.generateCode());  
        case M_ADD:  
            return String.format("%s := %s + %s\n",  
                firstArg.generateCode(),  
                firstArg.generateCode(),  
                secondArg.generateCode());  
        case M_SUB:  
            return String.format("%s := %s - %s\n",  
                firstArg.generateCode(),  
                firstArg.generateCode(),  
                secondArg.generateCode());  
    }  
}
```

```
case M_MUL:
    return String.format("%s := %s * %s\n",
        firstArg.generateCode(),
        firstArg.generateCode(),
        secondArg.generateCode());
case M_DIV:
    return String.format("%s := %s / %s\n",
        firstArg.generateCode(),
        firstArg.generateCode(),
        secondArg.generateCode());
default:
    return ""; // Should not occur.
}
}
}
```

```
package fig0735;

/**
 * The enumerated values for the states of the lexical analysis
 * finite state machine.
 *
 * <p>File:
 * <code>LexState.java</code>
 */
public enum LexState {

    LS_START, LS_IDENT, LS_SIGN, LS_INTEGER, LS_STOP
}
```

```
package fig0735;

/**
 * The lexical analyzer implemented as a finite state machine.
 *
 * <p>File:
 * <code>Tokenizer.java</code>
 */
public class Tokenizer {

    private InBuffer b;

    public Tokenizer(InBuffer inBuffer) {
        b = inBuffer;
    }
}
```



```
public AToken getToken() {
    char nextChar;
    StringBuffer localStringValue = new StringBuffer("");
    int localIntValue = 0;
    int sign = +1;
    AToken aToken = new TEmpty();
    LexState state = LexState.LS_START;
    do {
        nextChar = b.advanceInput();
        switch (state) {
            case LS_START:
                if (Util.isAlpha(nextChar)) {
                    localStringValue.append(nextChar);
                    state = LexState.LS_IDENT;
                } else if (nextChar == '-') {
                    sign = -1;
                    state = LexState.LS_SIGN;
                } else if (nextChar == '+') {
                    sign = +1;
                    state = LexState.LS_SIGN;
                } else if (Util.isDigit(nextChar)) {
                    localIntValue = nextChar - '0';
                    state = LexState.LS_INTEGER;
                }
            }
        }
    } while (nextChar != '\0');
```



```
} else if (nextChar == ',') {  
    aToken = new TComma();  
    state = LexState.LS_STOP;  
} else if (nextChar == '(') {  
    aToken = new TLeftParen();  
    state = LexState.LS_STOP;  
} else if (nextChar == ')') {  
    aToken = new TRightParen();  
    state = LexState.LS_STOP;  
} else if (nextChar == '\\n') {  
    state = LexState.LS_STOP;  
} else if (nextChar != ' ' ) {  
    aToken = new TInvalid();  
}  
break;
```

```
case LS_IDENT:
    if (Util.isAlpha(nextChar) || Util.isDigit(nextChar)) {
        localStringValue.append(nextChar);
    } else {
        b.backUpInput();
        aToken = new TIdentifier(localStringValue);
        state = LexState.LS_STOP;
    }
    break;
case LS_SIGN:
    if (Util.isDigit(nextChar)) {
        localStringValue.append(nextChar);
        state = LexState.LS_INTEGER;
    } else {
        aToken = new TInvalid();
    }
    break;
```

```
case LS_INTEGER:
```

```
    if (Util.isDigit(nextChar)) {  
        localIntValue = 10 * localIntValue + nextChar - '0';
```

```
    } else {  
        b.backUpInput();  
        aToken = new TInteger(localIntValue);  
        state = LexState.LS_STOP;
```

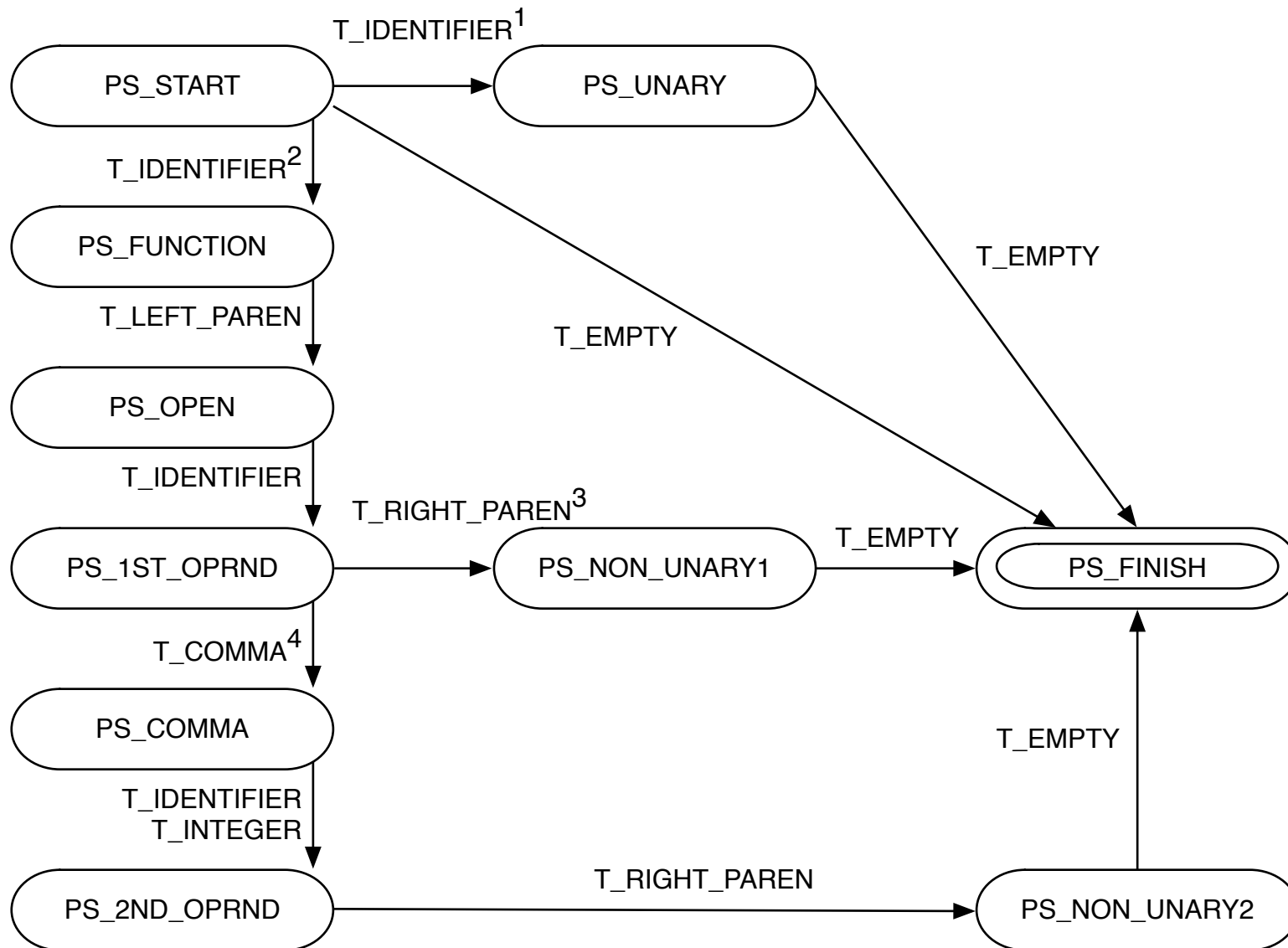
```
    }  
    break;
```

```
}
```

```
} while ((state != LexState.LS_STOP) && !(aToken instanceof TInvalid));  
return aToken;
```

```
}
```

```
}
```



Note 1: Only the identifiers stop and end.

Note 2: Only the identifiers set, add, sub, mul, div, neg, and abs.

Note 3: Only for mnemonics M\_NEG and M\_ABS.

Note 4: Only for mnemonics M\_SET, M\_ADD, M\_SUB, M\_MUL, M\_DIV.

```
package fig0735;

import java.util.EnumMap;
import java.util.HashMap;
import java.util.Map;

/**
 * The maps connecting string values with enumerated values.
 *
 * <p>File:
 * <code>Maps.java</code>
 *
 * @author J. Stanley Warford
 */
public final class Maps {

    public static final Map<String, Mnemon> unaryMnemonTable;
    public static final Map<String, Mnemon> nonUnaryMnemonTable;
    public static final Map<Mnemon, String> mnemonStringTable;
```

```
static {  
    unaryMnemonicTable = new HashMap<>();  
    unaryMnemonicTable.put("stop", Mnemon.M_STOP);  
    unaryMnemonicTable.put("end", Mnemon.M_END);  
  
    nonUnaryMnemonicTable = new HashMap<>();  
    nonUnaryMnemonicTable.put("neg", Mnemon.M_NEG);  
    nonUnaryMnemonicTable.put("abs", Mnemon.M_ABS);  
    nonUnaryMnemonicTable.put("add", Mnemon.M_ADD);  
    nonUnaryMnemonicTable.put("sub", Mnemon.M_SUB);  
    nonUnaryMnemonicTable.put("mul", Mnemon.M_MUL);  
    nonUnaryMnemonicTable.put("div", Mnemon.M_DIV);  
    nonUnaryMnemonicTable.put("set", Mnemon.M_SET);  
  
    mnemonicStringTable = new EnumMap<>(Mnemon.class);  
    mnemonicStringTable.put(Mnemon.M_NEG, "neg");  
    mnemonicStringTable.put(Mnemon.M_ABS, "abs");  
    mnemonicStringTable.put(Mnemon.M_ADD, "add");  
    mnemonicStringTable.put(Mnemon.M_SUB, "sub");  
    mnemonicStringTable.put(Mnemon.M_MUL, "mul");  
    mnemonicStringTable.put(Mnemon.M_DIV, "div");  
    mnemonicStringTable.put(Mnemon.M_SET, "set");  
    mnemonicStringTable.put(Mnemon.M_STOP, "stop");  
    mnemonicStringTable.put(Mnemon.M_END, "end");  
}
```

```
package fig0735;

/**
 * The enumerated values for the states of the parser finite state machine.
 *
 * <p>File:
 * <code>ParseState.java</code>
 */
public enum ParseState {

    PS_START, PS_UNARY, PS_FUNCTION, PS_OPEN, PS_1ST_OPRND, PS_NONUNARY1,
    PS_COMMA, PS_2ND_OPRND, PS_NON_UNARY2, PS_FINISH
}
```



Translator
<ul style="list-style-type: none"><li>- b: InBuffer</li><li>- t: Tokenizer</li><li>- aCode: ACode</li></ul>
<ul style="list-style-type: none"><li>+ Translator (inBuffer: InBuffer)</li><li>- parseLine ( ): boolean</li><li>+ translate ( ): void</li></ul>

translate ( )

1. Instantiates tokenizer t, passing a reference to the buffer

parseLine ( )

1. Implements the parser FSM for one line of source code

2. Sets aCode to the code object from the parse

3. Returns true if it has processed the end statement



```
package fig0735;

import java.util.ArrayList;

/**
 * The parser implemented as a finite state machine.
 *
 * <p>File:
 * <code>Translator.java</code>
 */
public class Translator {

    private InBuffer b;
    private Tokenizer t;
    private ACode aCode;

    public Translator(InBuffer inBuffer) {
        b = inBuffer;
    }
}
```

```
// Sets aCode and returns boolean true if end statement is processed.
private boolean parseLine() {
    boolean terminate = false;
    AArg localFirstArg = new IntArg(0);
    AArg localSecondArg = new IntArg(0);
    Mnemon localMnemon = Mnemon.M_END;
    // Compiler requires useless initialization.
    AToken aToken;
    aCode = new EmptyInstr();
    ParseState state = ParseState.PS_START;
    do {
        aToken = t.getToken();
        switch (state) {
```

```
case PS_START:
    if (aToken instanceof TIdentifier) {
        TIdentifier localTIdentifier = (TIdentifier) aToken;
        String tempStr = localTIdentifier.getStringValue();
        if (Maps.unaryMnemonTable.containsKey(
            tempStr.toLowerCase())) {
            localMnemon = Maps.unaryMnemonTable.get(
                tempStr.toLowerCase());
            aCode = new UnaryInstr(localMnemon);
            terminate = localMnemon == Mnemon.M_END;
            state = ParseState.PS_UNARY;
        } else if (Maps.nonUnaryMnemonTable.containsKey(
            tempStr.toLowerCase())) {
            localMnemon = Maps.nonUnaryMnemonTable.get(
                tempStr.toLowerCase());
            state = ParseState.PS_FUNCTION;
        } else {
            aCode = new Error(
                "Line must begin with function identifier.");
        }
    }
```

```
} else if (aToken instanceof TEmpty) {  
    aCode = new EmptyInstr();  
    state = ParseState.PS_FINISH;  
} else {  
    aCode = new Error(  
        "Line must begin with function identifier.");  
}  
break;
```

```
case PS_UNARY:
    if (aToken instanceof TEmpty) {
        state = ParseState.PS_FINISH;
    } else {
        aCode = new Error("Illegal trailing character.");
    }
    break;
case PS_FUNCTION:
    if (aToken instanceof TLeftParen) {
        state = ParseState.PS_OPEN;
    } else {
        aCode = new Error(
            "Left parenthesis expected after function.");
    }
    break;
case PS_OPEN:
    if (aToken instanceof TIdentifier) {
        TIdentifier localTIdentifier = (TIdentifier) aToken;
        localFirstArg = new IdentArg(
            localTIdentifier.getStringValue());
        state = ParseState.PS_1ST_OPRND;
    } else {
        aCode = new Error("First argument not an identifier.");
    }
    break;
```

```
case PS_1ST_OPRND:
    if (localMnemon == Mnemon.M_NEG || localMnemon == Mnemon.M_ABS) {
        if (aToken instanceof TRightParen) {
            aCode = new OneArgInstr(localMnemon, localFirstArg);
            state = ParseState.PS_NONUNARY1;
        } else {
            aCode = new Error(
                "Right parenthesis expected after argument.");
        }
    } else if (aToken instanceof TComma) {
        state = ParseState.PS_COMMA;
    } else {
        aCode = new Error("Comma expected after first argument.");
    }
    break;
case PS_NONUNARY1:
    if (aToken instanceof TEmpty) {
        state = ParseState.PS_FINISH;
    } else {
        aCode = new Error("Illegal trailing character.");
    }
    break;
```

```
case PS_COMMA:
    if (aToken instanceof TIdentifier) {
        TIdentifier localTIdentifier = (TIdentifier) aToken;
        localSecondArg = new IdentArg(
            localTIdentifier.getStringValue());
        aCode = new TwoArgInstr(
            localMnemon, localFirstArg, localSecondArg);
        state = ParseState.PS_2ND_OPRND;
    } else if (aToken instanceof TInteger) {
        TInteger localTInteger = (TInteger) aToken;
        localSecondArg = new IntArg(localTInteger.getIntValue());
        aCode = new TwoArgInstr(
            localMnemon, localFirstArg, localSecondArg);
        state = ParseState.PS_2ND_OPRND;
    } else {
        aCode = new Error(
            "Second argument not an identifier or integer.");
    }
    break;
```



```
case PS_2ND_OPRND:
    if (aToken instanceof TRightParen) {
        state = ParseState.PS_NON_UNARY2;
    } else {
        aCode = new Error(
            "Right parenthesis expected after argument.");
    }
    break;
case PS_NON_UNARY2:
    if (aToken instanceof TEmpty) {
        state = ParseState.PS_FINISH;
    } else {
        aCode = new Error("Illegal trailing character.");
    }
    break;
}
} while (state != ParseState.PS_FINISH && !(aCode instanceof Error));
return terminate;
}
```



```
public void translate() {
    ArrayList<ACode> codeTable = new ArrayList<ACode>();
    int numErrors = 0;
    t = new Tokenizer(b);
    boolean terminateWithEnd = false;
    b.getLine();
    while (b.inputRemains() && !terminateWithEnd) {
        terminateWithEnd = parseLine(); // Sets aCode and returns boolean.
        codeTable.add(aCode);
        if (aCode instanceof Error) {
            numErrors++;
        }
        b.getLine();
    }
    if (!terminateWithEnd) {
        aCode = new Error("Missing \"end\" sentinel.");
        codeTable.add(aCode);
        numErrors++;
    }
}
```

```
if (numErrors == 0) {
    System.out.printf("Object code:\n");
    for (int i = 0; i < codeTable.size(); i++) {
        System.out.printf("%s", codeTable.get(i).generateCode());
    }
}
if (numErrors == 1) {
    System.out.printf("One error was detected.\n");
} else if (numErrors > 1) {
    System.out.printf("%d errors were detected.\n", numErrors);
}
System.out.printf("\nProgram listing:\n");
for (int i = 0; i < codeTable.size(); i++) {
    System.out.printf("%s", codeTable.get(i).generateListing());
}
}
```

$$N = \{A, B\}$$

$$T = \{0, 1\}$$

$P$  = the productions

1.  $A \rightarrow 0B$

2.  $B \rightarrow 10B$

3.  $B \rightarrow \epsilon$

$$S = A$$

$$N = \{C\}$$

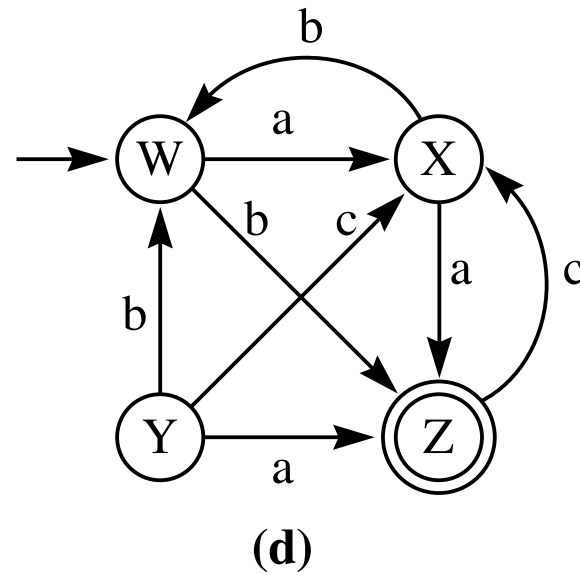
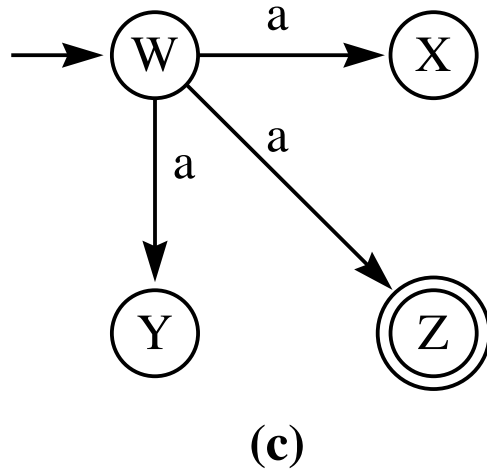
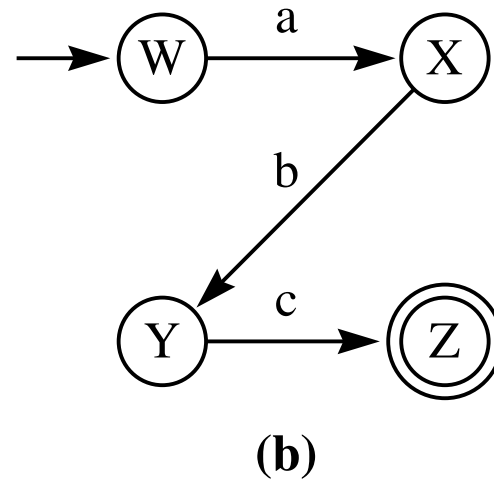
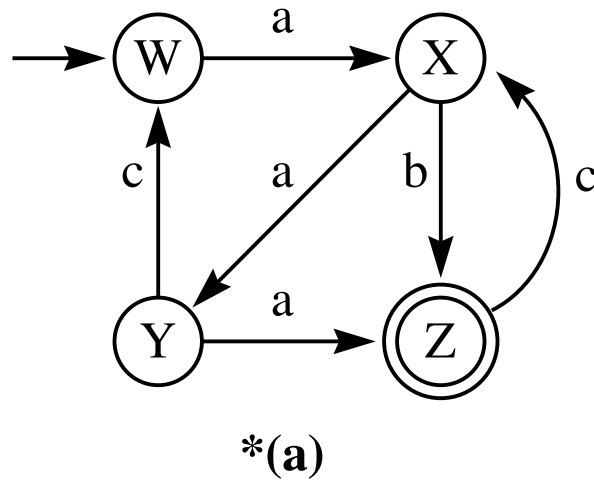
$$T = \{0, 1\}$$

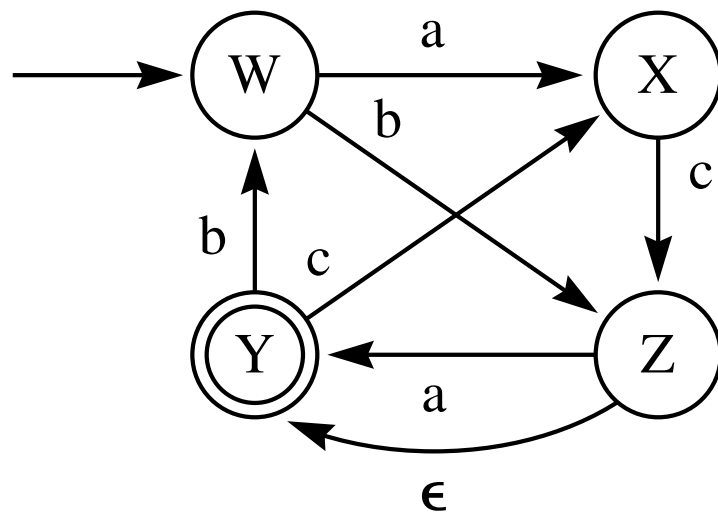
$P$  = the productions

1.  $C \rightarrow C10$

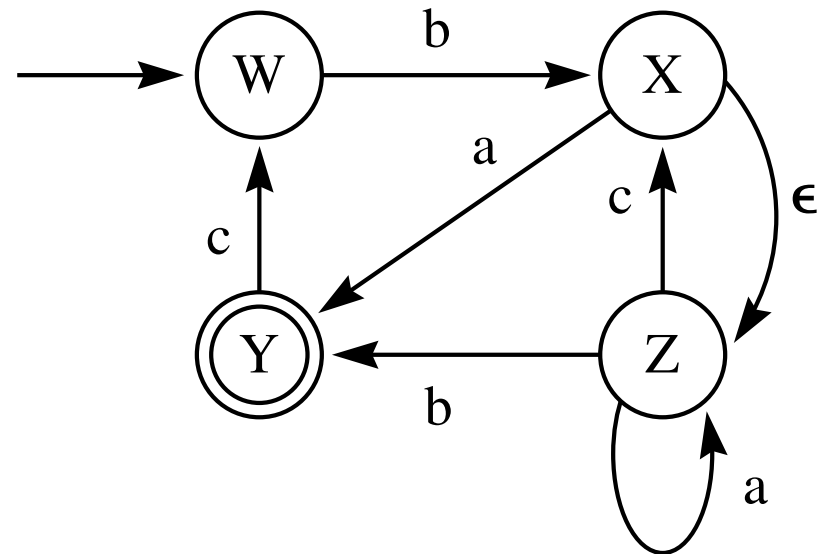
2.  $C \rightarrow 0$

$$S = C$$





(a)



(b)

Figure 7.50

