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.

The C++ alphabet

```
{ 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, +, -, *, /, =, <, >, [,
  ], (, ), {, }, ., ,, :, ;, &, !, %, ', "
  _, \, #, ?, }, |, ~ }
```

The alphabet for Pep/8 assembly language

```
{ 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, \, ., ,, :, ;, ‘, “  }
```

The alphabet for real numbers

{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -, . }

Concatenation

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

The empty string

- ϵ
- 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

$$
\begin{aligned}
N &= \{< \text{identifier} >, < \text{letter} >, < \text{digit} >\} \\
T &= \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{1}, \texttt{2}, \texttt{3}\} \\
P &= \text{the productions} \\
&\quad 1.\ < \text{identifier} > \rightarrow < \text{letter} > \\
&\quad 2.\ < \text{identifier} > \rightarrow < \text{identifier} > < \text{letter} > \\
&\quad 3.\ < \text{identifier} > \rightarrow < \text{identifier} > < \text{digit} > \\
&\quad 4.\ < \text{letter} > \rightarrow \texttt{a} \\
&\quad 5.\ < \text{letter} > \rightarrow \texttt{b} \\
&\quad 6.\ < \text{letter} > \rightarrow \texttt{c} \\
&\quad 7.\ < \text{digit} > \rightarrow \texttt{1} \\
&\quad 8.\ < \text{digit} > \rightarrow \texttt{2} \\
&\quad 9.\ < \text{digit} > \rightarrow \texttt{3} \\
S &= < \text{identifier} >
\end{aligned}
$$

Figure 7.2

$$
\begin{aligned}
N &= \{\mathrm{I}, \mathrm{F}, \mathrm{M}\} \\
T &= \{\texttt{+}, \texttt{-}, \texttt{d}\} \\
P &= \text{the productions} \\
&\qquad 1.\ \mathrm{I} \rightarrow \mathrm{FM} \\
&\qquad 2.\ \mathrm{F} \rightarrow \texttt{+} \\
&\qquad 3.\ \mathrm{F} \rightarrow \texttt{-} \\
&\qquad 4.\ \mathrm{F} \rightarrow \epsilon \\
&\qquad 5.\ \mathrm{M} \rightarrow \texttt{dM} \\
&\qquad 6.\ \mathrm{M} \rightarrow \texttt{d} \\
S &= \mathrm{I}
\end{aligned}
$$

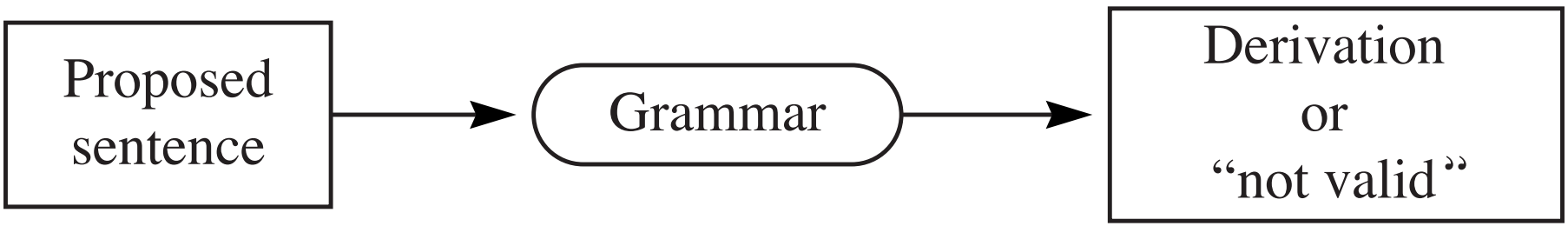
Grammars

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

Figure 7.3

$$
\begin{aligned}
N &= \{\mathrm{A}, \mathrm{B}, \mathrm{C}\} \\
T &= \{\mathtt{a}, \mathtt{b}, \mathtt{c}\} \\
P &= \text{the productions} \\
&\qquad 1.\ \mathrm{A} \rightarrow \mathtt{a}\mathrm{ABC} \\
&\qquad 2.\ \mathrm{A} \rightarrow \mathtt{ab}\mathrm{C} \\
&\qquad 3.\ \mathrm{CB} \rightarrow \mathrm{BC} \\
&\qquad 4.\ \mathtt{b}\mathrm{B} \rightarrow \mathtt{bb} \\
&\qquad 5.\ \mathtt{b}\mathrm{C} \rightarrow \mathtt{bc} \\
&\qquad 6.\ \mathtt{c}\mathrm{C} \rightarrow \mathtt{cc} \\
S &= \mathrm{A}
\end{aligned}
$$

(a) Deriving a valid sentence.



(b) The parsing problem.

$$
\begin{aligned}
N &= \{\mathrm{E}, \mathrm{T}, \mathrm{F}\} \\
T &= \{\texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{a}\} \\
P &= \text{the productions} \\
&\qquad 1.\ \mathrm{E} \rightarrow \mathrm{E}\ \texttt{+}\ \mathrm{T} \\
&\qquad 2.\ \mathrm{E} \rightarrow \mathrm{T} \\
&\qquad 3.\ \mathrm{T} \rightarrow \mathrm{T}\ \texttt{*}\ \mathrm{F} \\
&\qquad 4.\ \mathrm{T} \rightarrow \mathrm{F} \\
&\qquad 5.\ \mathrm{F} \rightarrow \texttt{(}\ \mathrm{E}\ \texttt{)} \\
&\qquad 6.\ \mathrm{F} \rightarrow \texttt{a} \\
S &= \mathrm{E}
\end{aligned}
$$

Figure 7.6

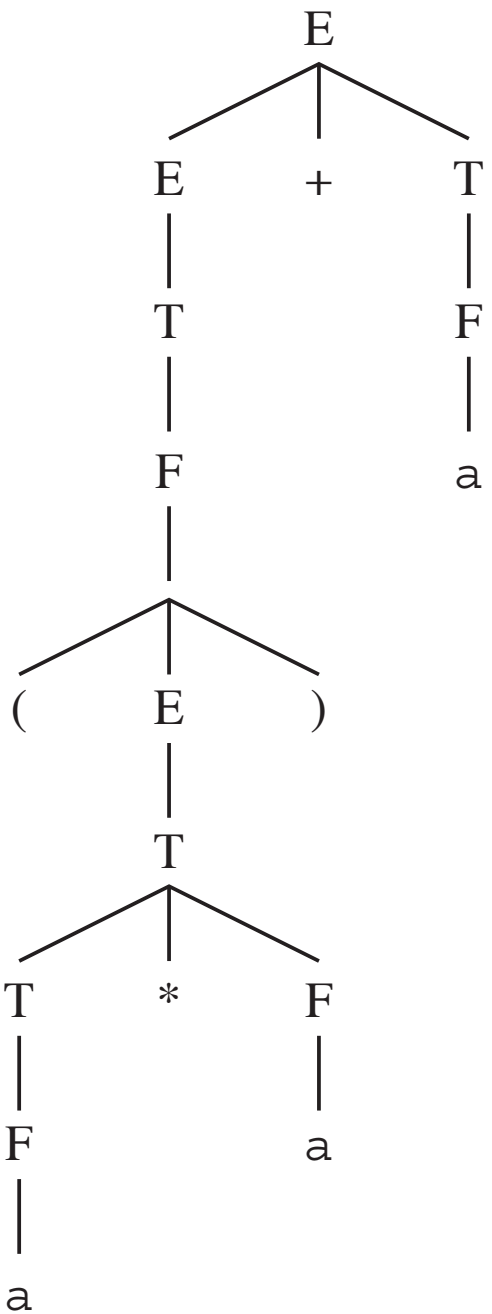


Figure 7.7

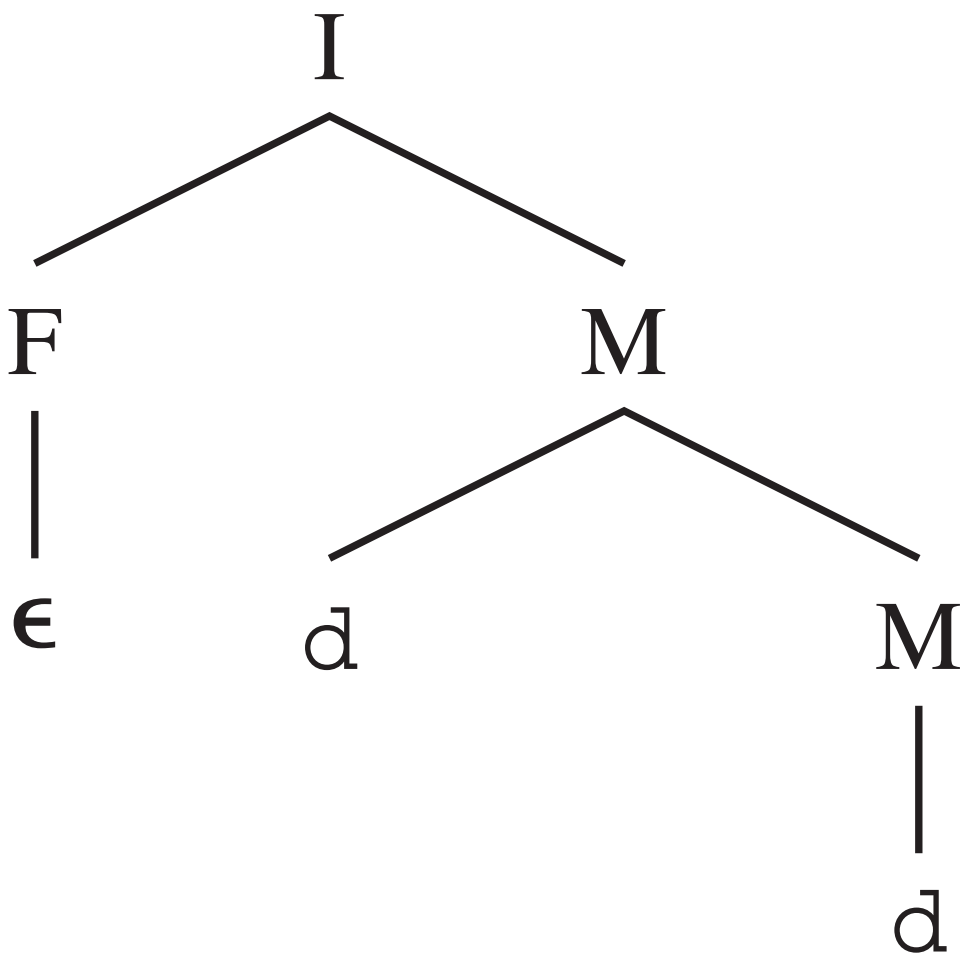


Figure 7.8

<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>

Figure 7.8

<parameter-list> →
    ϵ
    | <parameter-declaration>
    | <parameter-list> , <parameter-declaration>

<parameter-declaration> → <type-specifier> <identifier>

<compound-statement> → { <declaration-list> <statement-list> }

<declaration-list> →
    ϵ
    | <declaration>
    | <declaration-list> <declaration>

<statement-list> →
    ϵ
    | <statement>
    | <statement-list> <statement>

Figure 7.8

```
<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>
```

Figure 7.8

```
<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>
```

Figure 7.8

```
<unary-expression> →
        <primary-expression>
      | <identifier> ( <argument-expression-list> )

<primary-expression> →
        <identifier>
      | <constant>
      | ( <expression> )

<argument-expression-list> →
        <expression>
      | <argument-expression-list> , <expression>

<constant> →
        <integer-constant>
      | <character-constant>

<integer-constant> →
        <digit>
      | <integer-constant> <digit>
```

Figure 7.8

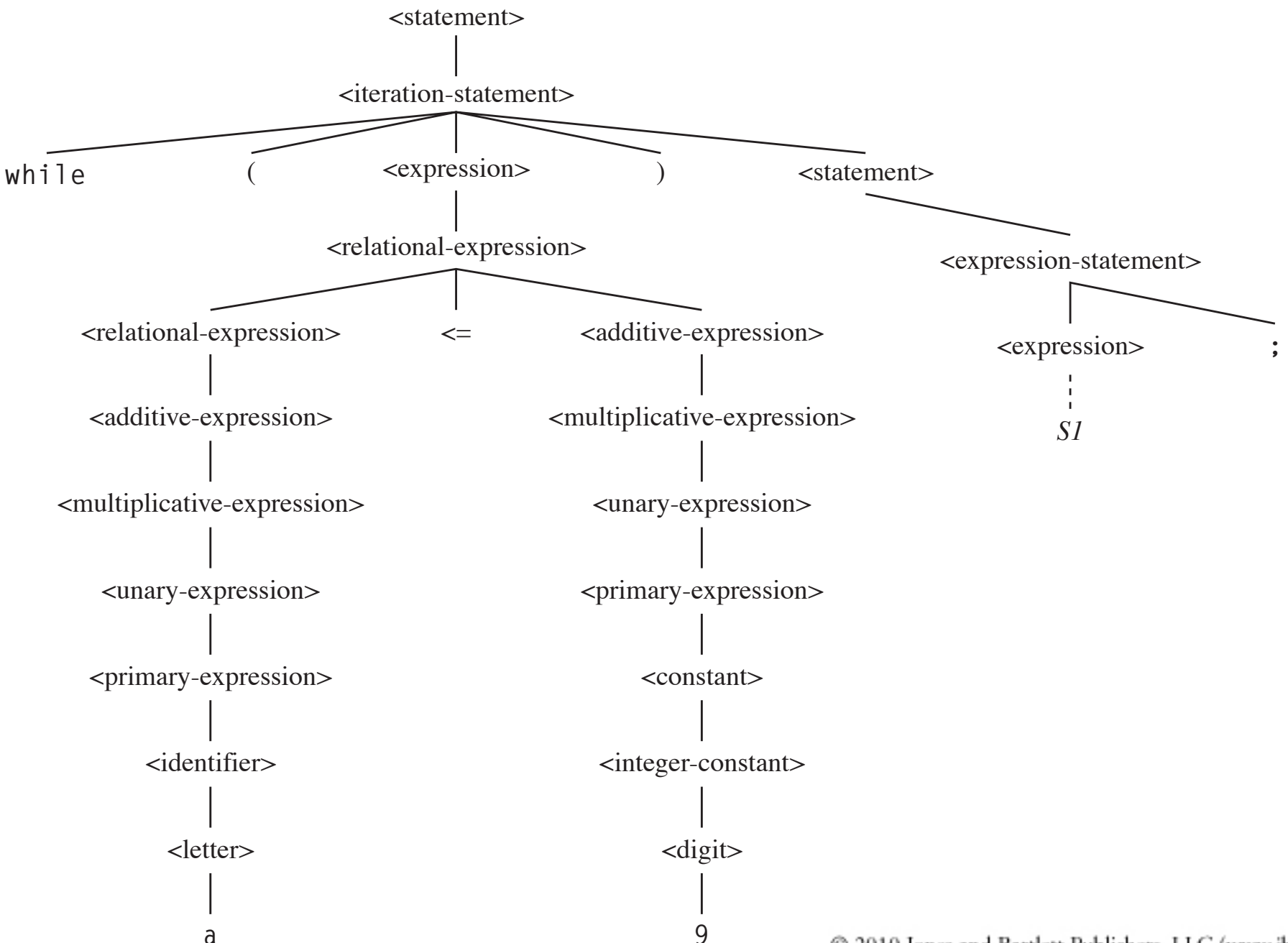
<character-constant> → ' <letter> '

<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

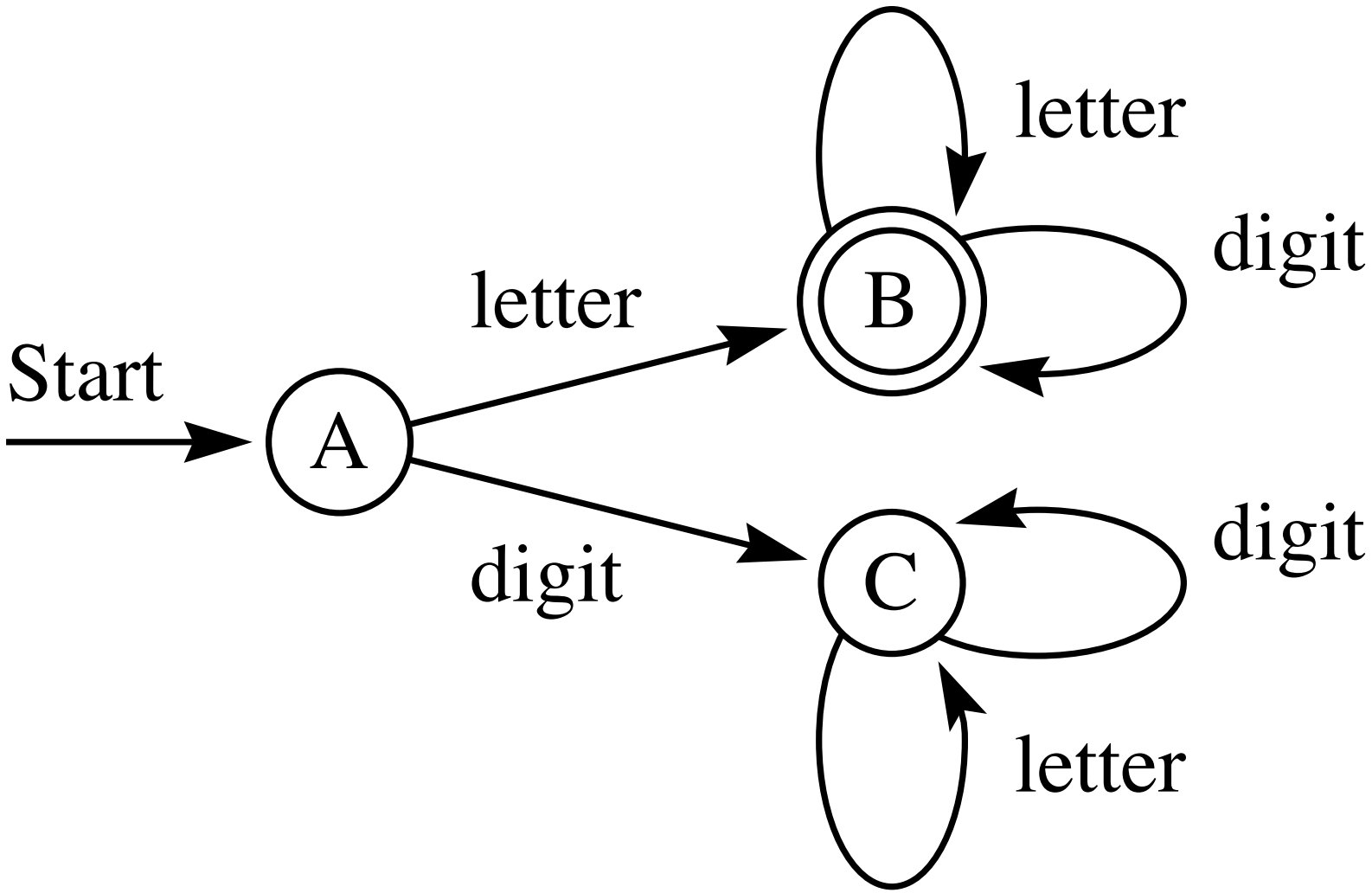
Figure 7.9



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

Figure 7.10



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

Figure 7.11

Current State	Next State	
	Letter	Digit
→ A	B	C
(B)	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

Figure 7.12

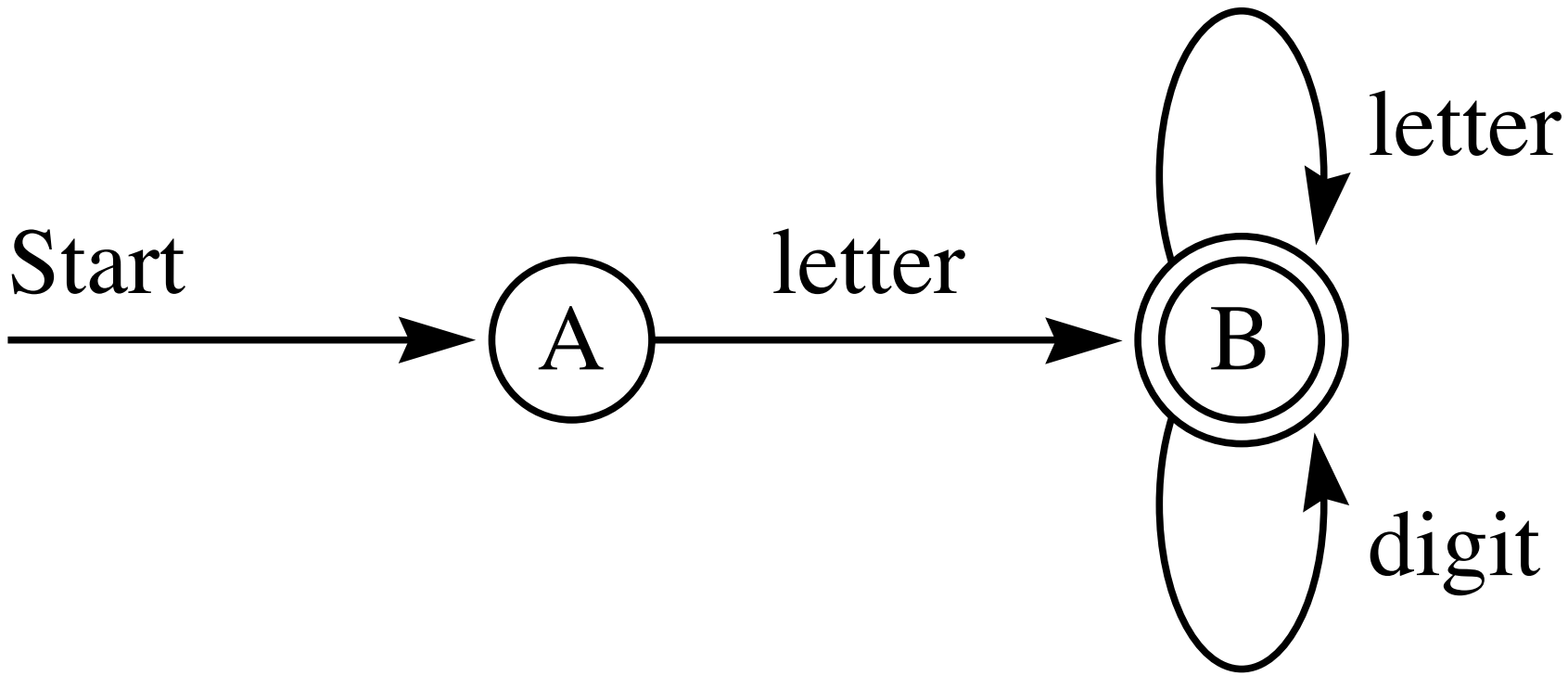


Figure 7.13

Current State	Next State	
	Letter	Digit
→ A	B	
(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

Figure 7.14

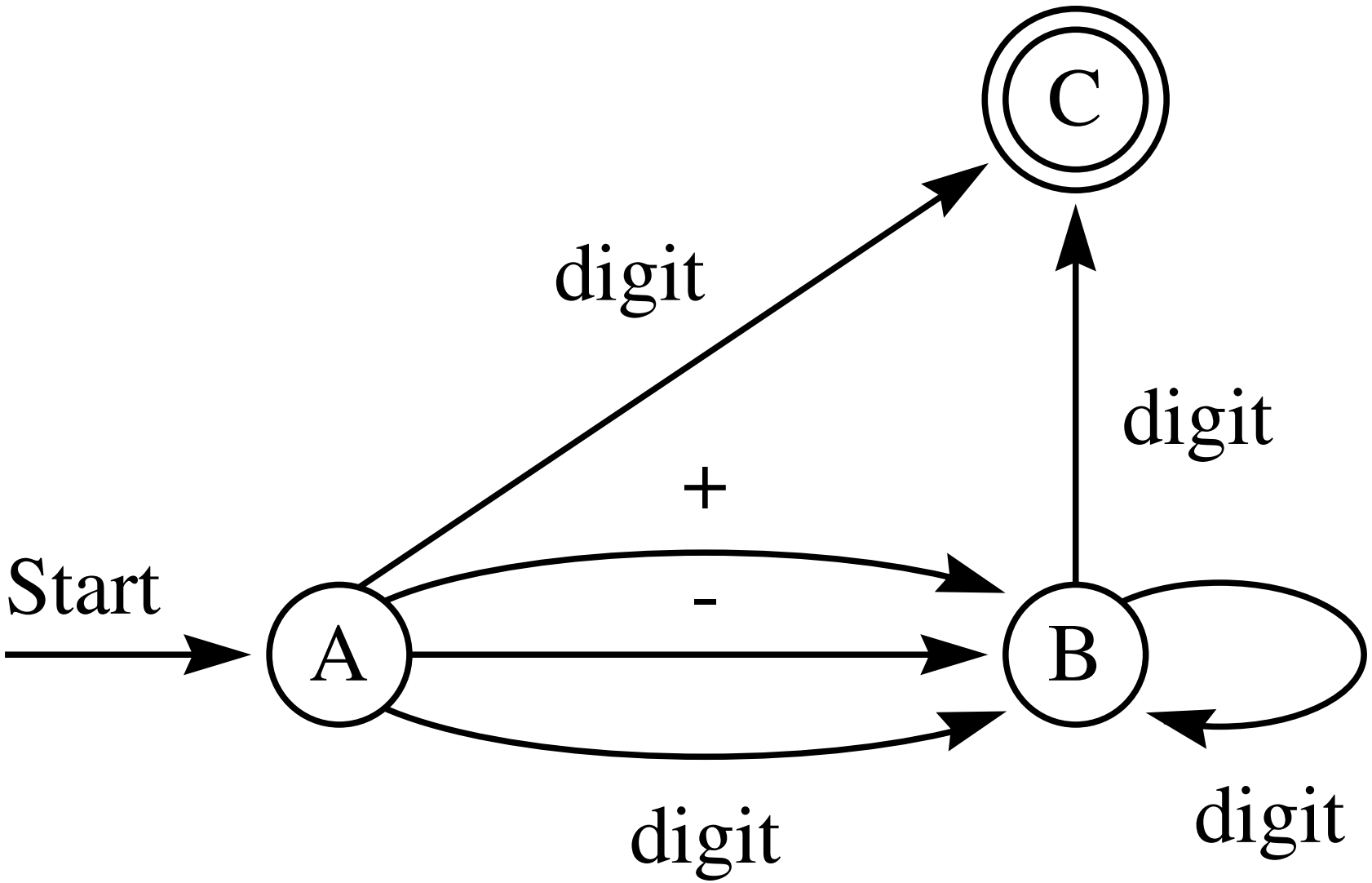


Figure 7.15

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

Figure 7.17

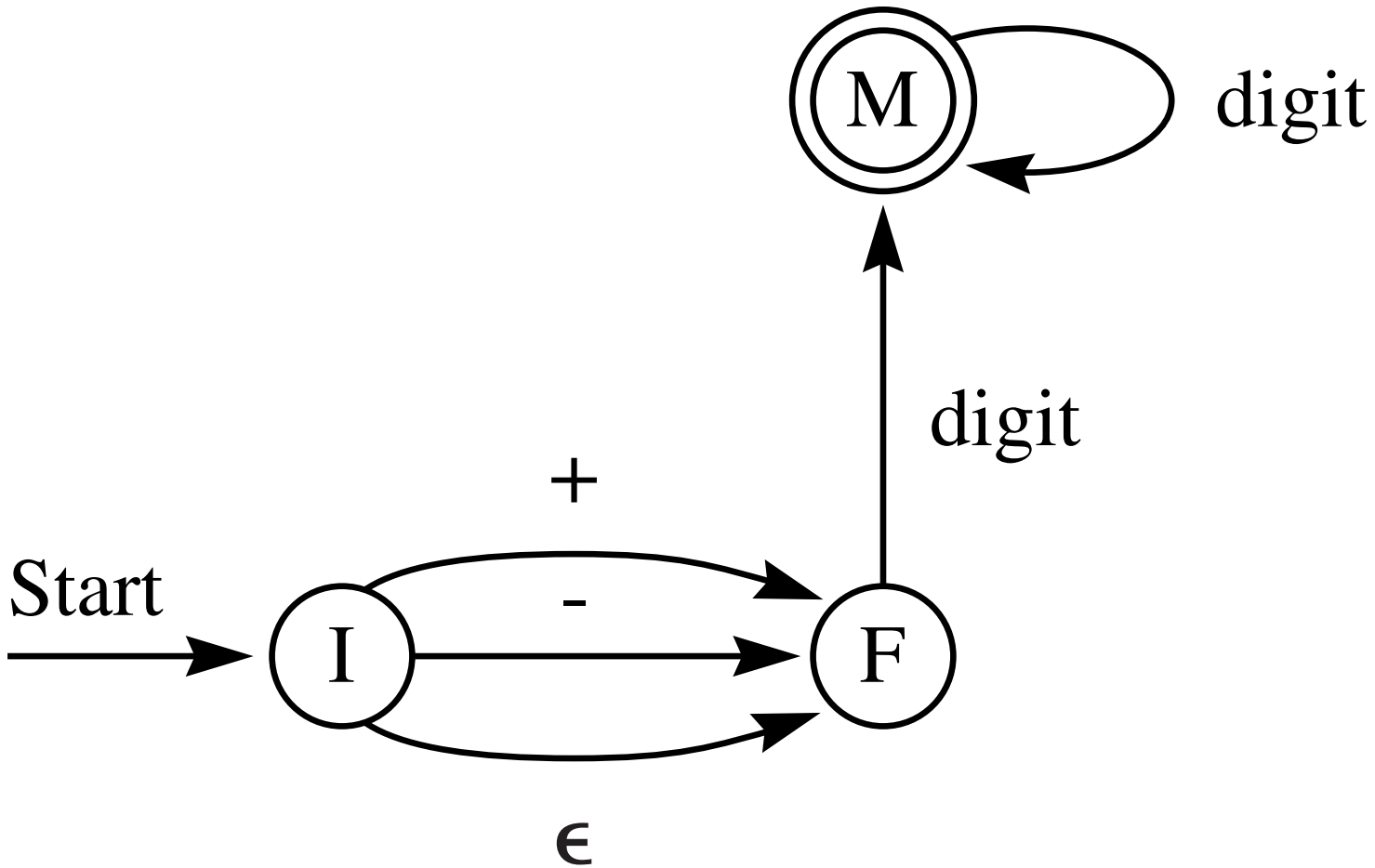
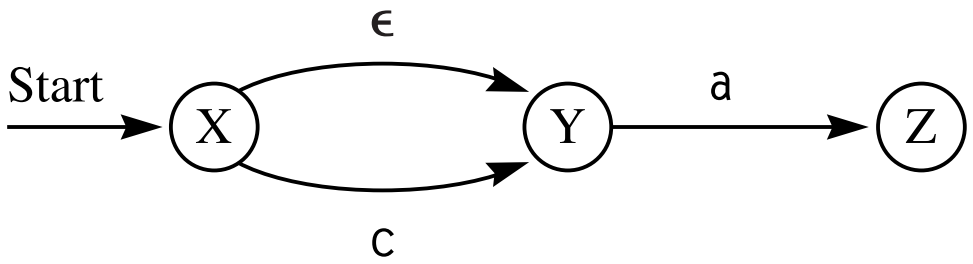


Figure 7.16

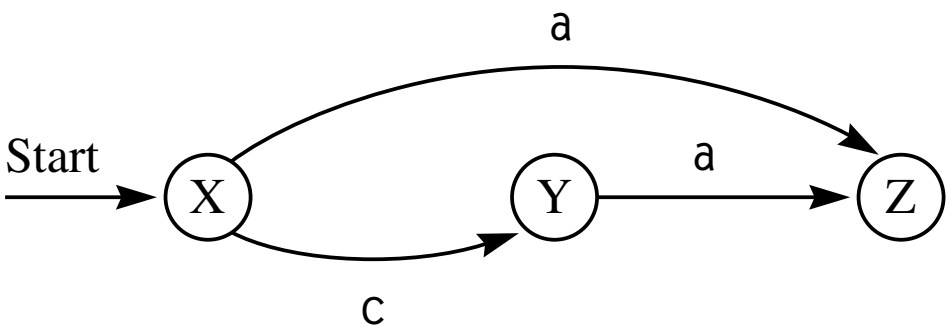
Current State	Next State			
	+	–	Digit	ϵ
→ I	F	F		F
F			M	
Ⓜ			M	

Removing empty transitions

- Given a transition from p to q on ϵ, 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

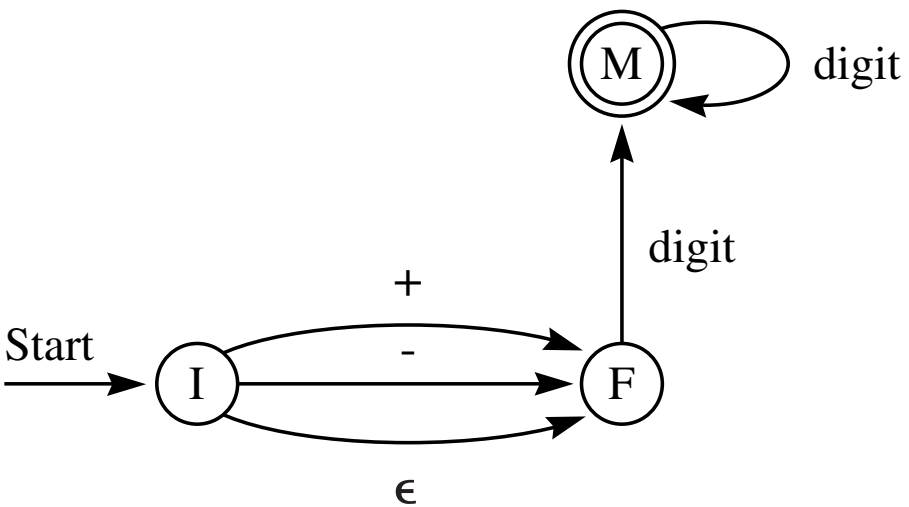


(a) The original FSM.

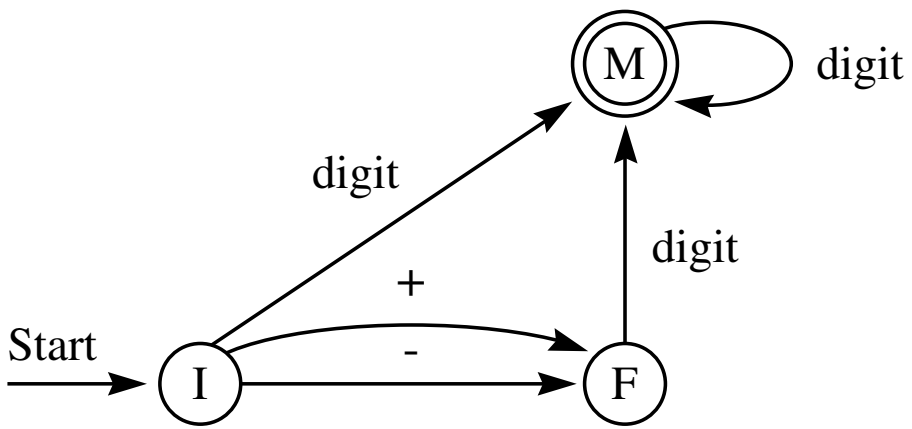


(b) The equivalent FSM without an empty transition.

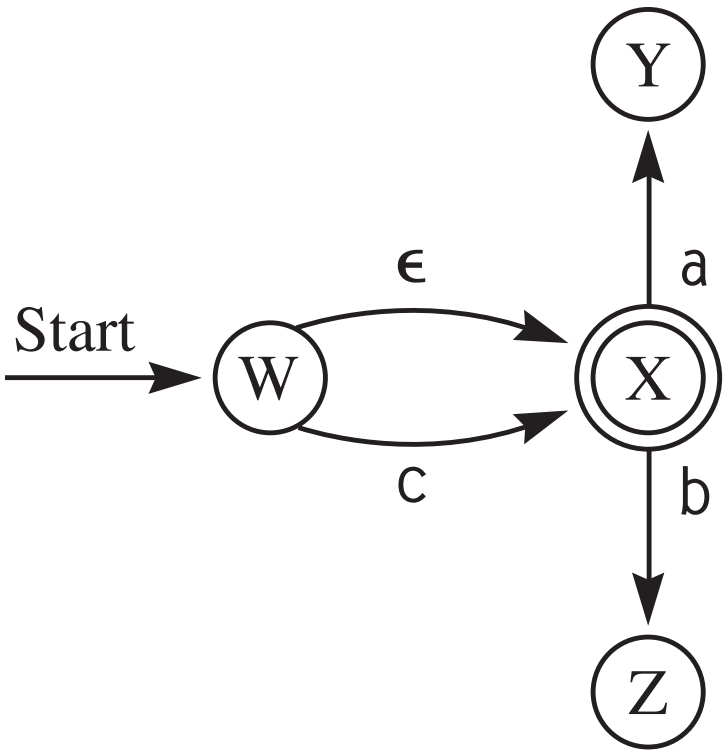
Figure 7.19



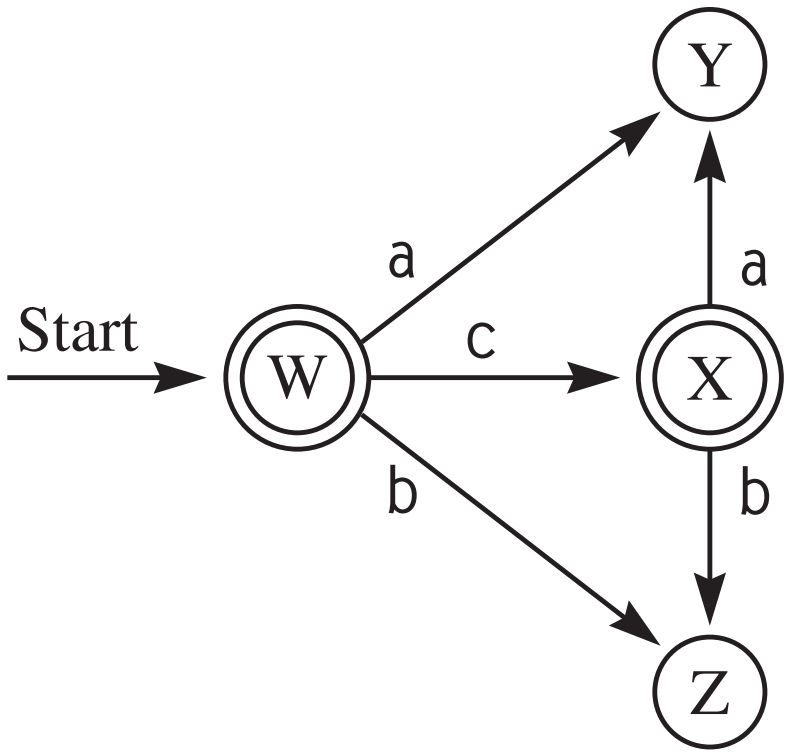
(a) The original FSM.



(b) The empty transition removed.



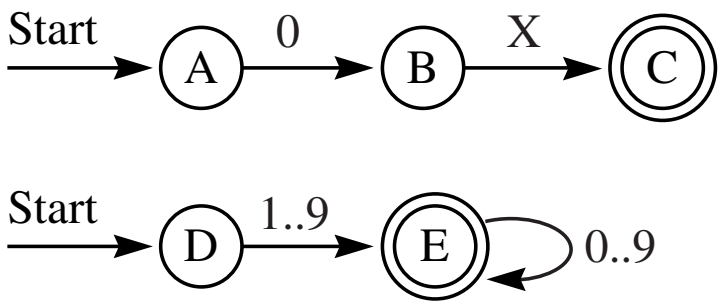
(a) The original FSM.



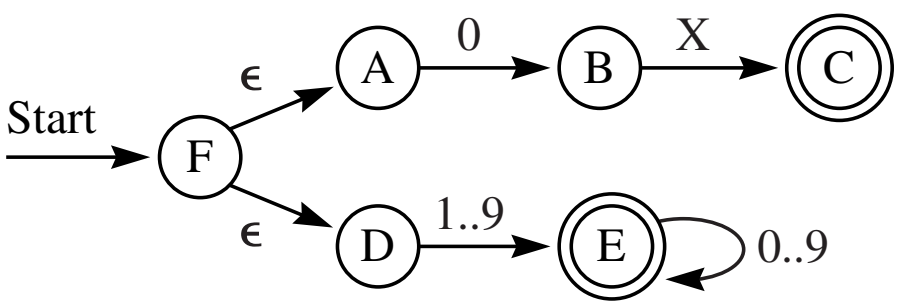
(b) The equivalent FSM without an empty transition.

Multiple token recognizers

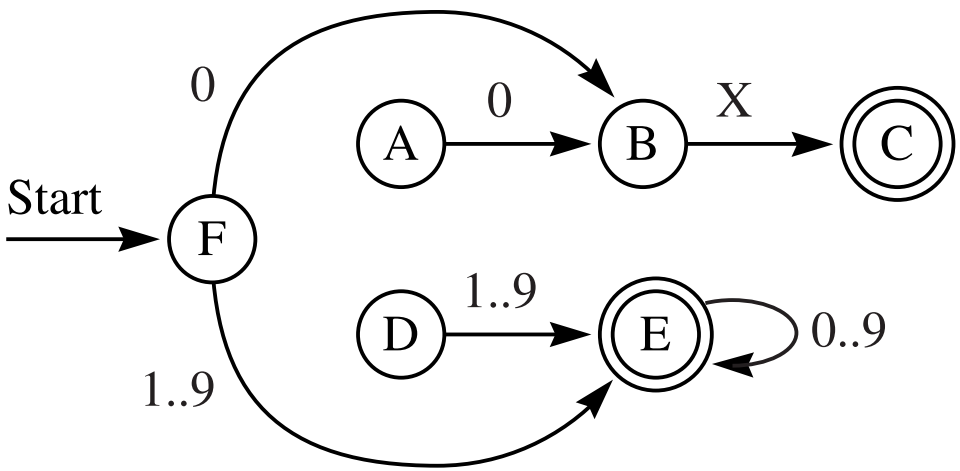
- Token
  - A string of terminal characters that has meaning as a group
- FSM with multiple final states
- The final state determines the token that is recognized



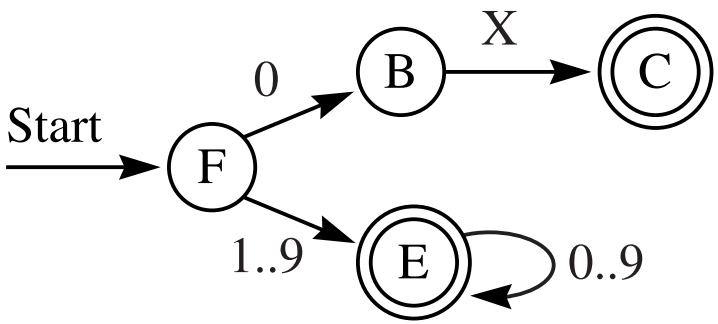
(a) Separate machines for the 0X and unsigned integer tokens.



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



(a) Removing the empty transitions.



(b) Removing the inaccessible states.

Figure 7.23

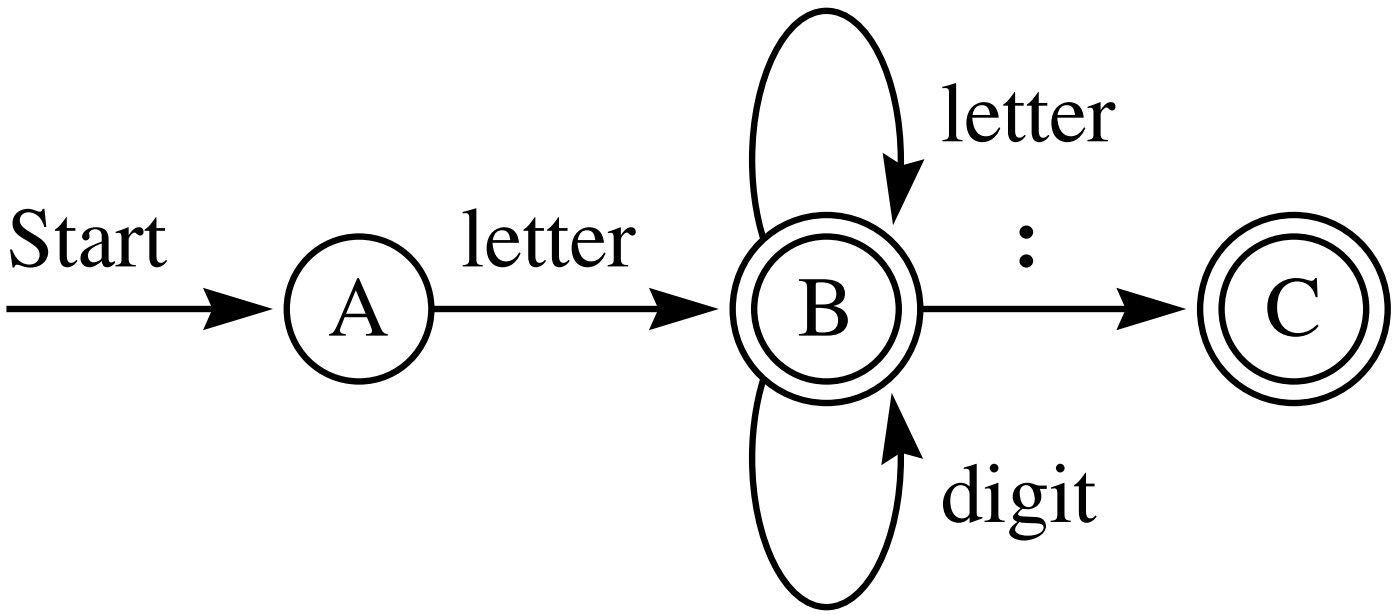
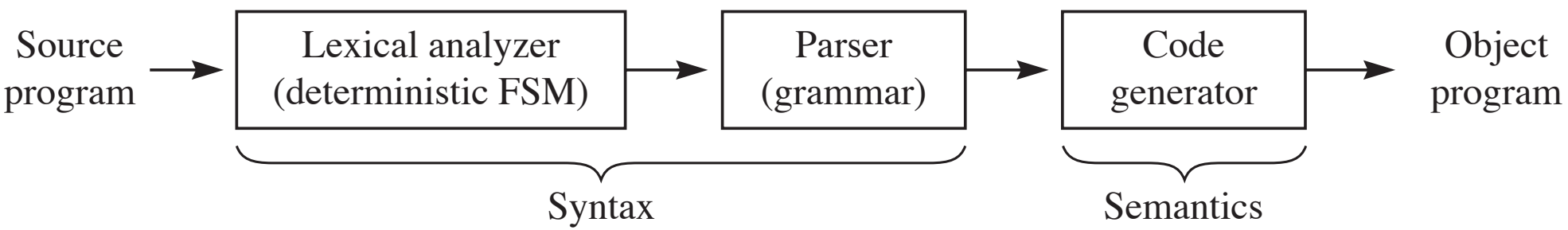


Figure 7.24



FSM implementation techniques

- Table-lookup
- Direct-code

Figure 7.11

Current State	Next State	
	Letter	Digit
→ A	B	C
(B)	B	B
C	C	C

Figure 7.25
Input/Output

Interactive Input/Output

```
Enter a string of letters and digits: cab3
The string is a valid identifier.
```

Interactive Input/Output

```
Enter a string of letters and digits: 3cab
Invalid identifier.
```

Figure 7.25

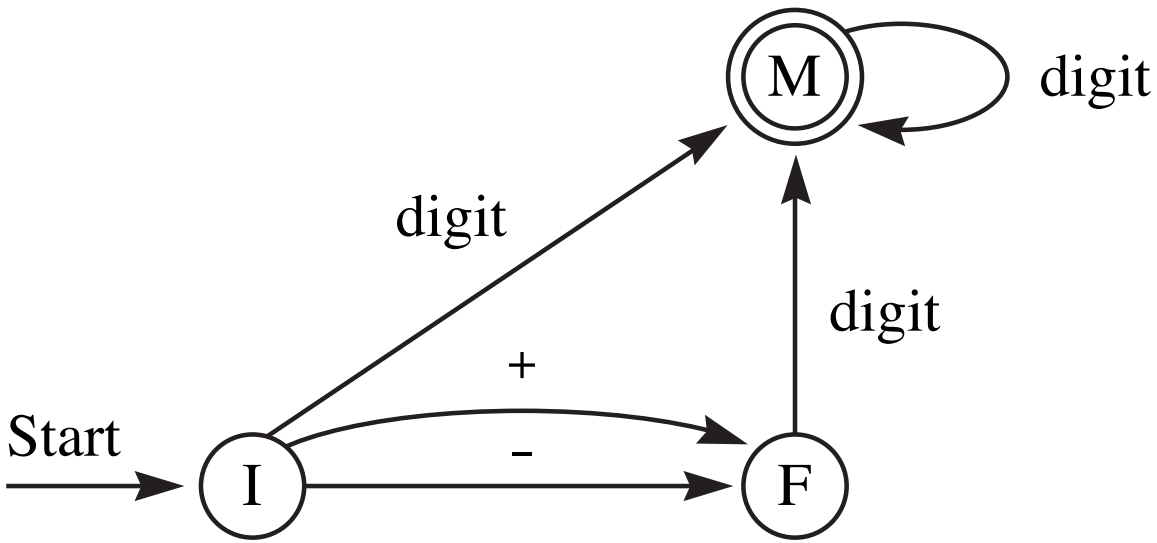
```
#include <iostream>
#include <cctype> // isalpha
#include <string> // string
using namespace std;

enum State {eA, eB, eC};
enum Alphabet {eLETTER, eDIGIT};
State FSM[eC + 1][eDIGIT + 1] = {  // Three rows, two columns
   eB, eC,  // The state transition table of Figure 7.11
   eB, eB,
   eC, eC
};
```

Figure 7.25
(Continued)

```
int main () {
   char ch;
   string line;
   Alphabet FSMChar;
   State state = eA;
   cout << "Enter a string of letters and digits: ";
   cin >> line;
   for (int i = 0; i < line.size (); i++) {
      ch = line[i];
      if (isalpha (ch)) {
         FSMChar = eLETTER;
      }
      else {
         FSMChar = eDIGIT;
      }
      state = FSM[state][FSMChar];
   }
   if (state == eB) {
      cout << line << " is a valid identifier." << endl;
   }
   else {
      cout << line << " is not a valid identifier." << endl;
   }
   return 0;
}
```

Figure 7.19(b)



(b) The empty transition removed.

Figure 7.26
Input/Output

Interactive Input/Output

```
Enter a number: q
Invalid entry.
```

Interactive Input/Output

```
Enter a number: -58
Num = -58
```

Figure 7.26

```
#include <iostream>
#include <cctype> // isdigit
#include <string> // string, getline
using namespace std;

// Global buffer
string line;
int lineIndex;

void getLine () {
   // Get a line of characters.
   // Install a newline character as sentinel.
   getline (cin, line);
   line.push_back ('\n');
   lineIndex = 0;
}
```

Figure 7.26
(Continued)

```
int main () {
   bool valid;
   int num;
   cout << "Enter number: ";
   getLine ();
   parseNum (valid, num);
   if (valid) {
      cout << "Number = " << num << endl;
   }
   else {
      cout << "Invalid entry." << endl;
   }
   return 0;
}
```

Figure 7.26
(Continued)

```
enum State {eI, eF, eM, eSTOP};

void parseNum (bool& v, int& n) {
   int sign;
   State state;
   char nextChar;
   v = true;
   state = eI;
   do {
      nextChar = line[lineIndex++];
      switch (state) {
```

Figure 7.26
(Continued)

```
case eI:
   if (nextChar == '+') {
      sign = +1;
      state = eF;
   }
   else if (nextChar == '-') {
      sign = -1;
      state = eF;
   }
   else if (isdigit (nextChar)) {
      sign = +1;
      n = nextChar - '0';
      state = eM;
   }
   else {
      v = false;
   }
   break;
case eF:
   if (isdigit (nextChar)) {
      n = nextChar - '0';
      state = eM;
   }
   else {
      v = false;
   }
   break;
```

Figure 7.26
(Continued)

```
            case eM:
               if (isdigit (nextChar)) {
                  n = 10 * n + nextChar - '0';
               }
               else if (nextChar == '\n') {
                  n = sign * n;
                  state = eSTOP;
               }
               else {
                  v = false;
               }
               break;
         }
      }
      while ((state != eSTOP) && v);
}
```

An input buffer

- Used to process one character at a time from a C++ 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

Figure 7.27

```
#include <string> // string, getline

class InBuffer {
private:
   string line;
   int lineIndex;

public:
   void getLine () {
      getline (cin, line);
      line.push_back ('\n');
      lineIndex = 0;
   }

   void advanceInput (char& ch) {
      ch = line[lineIndex++];
   }

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

Figure 7.28

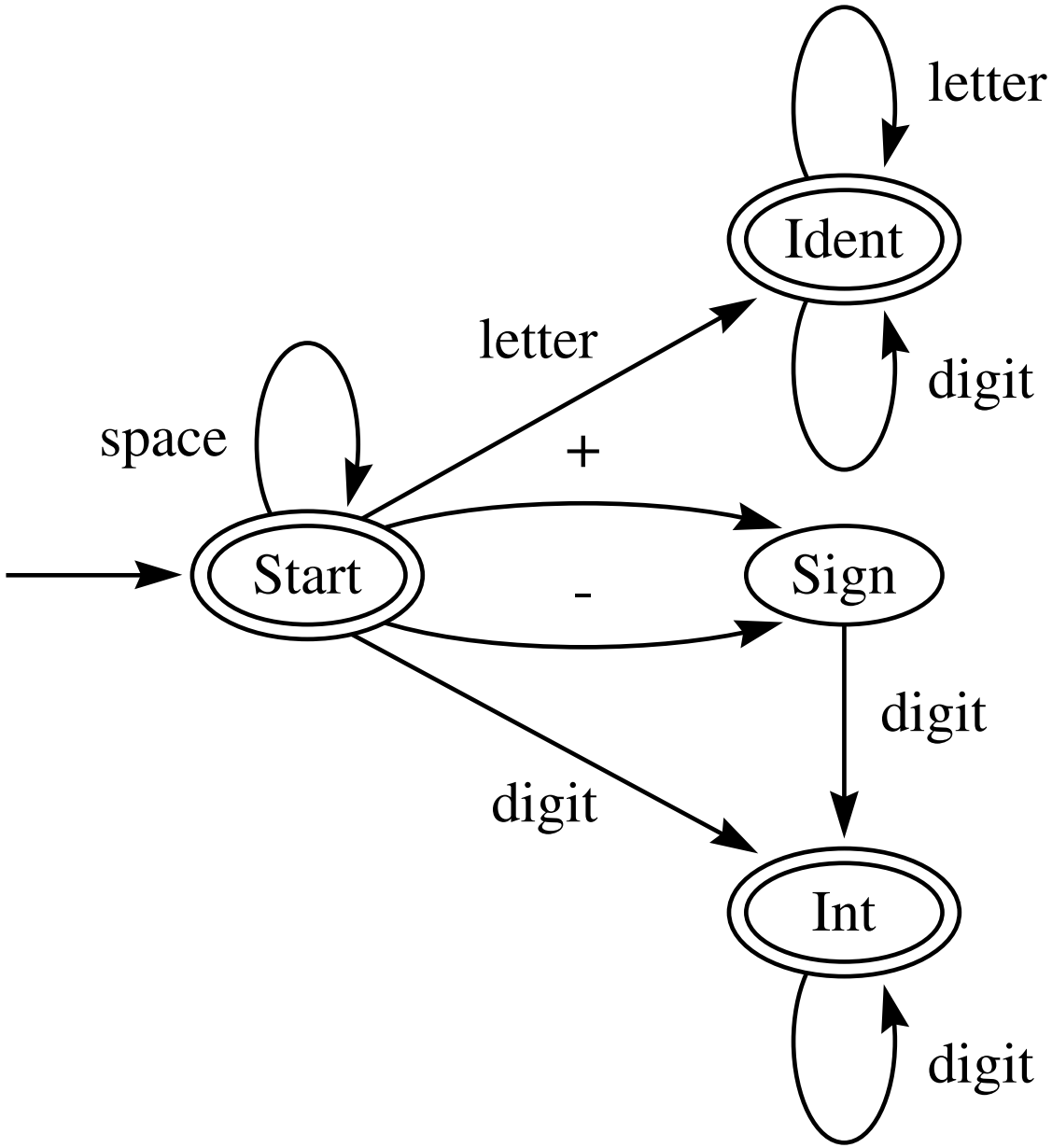


Figure 7.29
Input/Output

Input

```
Here    is A47 48B
   C-49 ALongIdentifier +50 D16-51
```

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
```

Figure 7.29
Input/Output

Input

```
Here    is A47+ 48B
   C+49

ALongIdentifier
```

Output

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

Figure 7.30

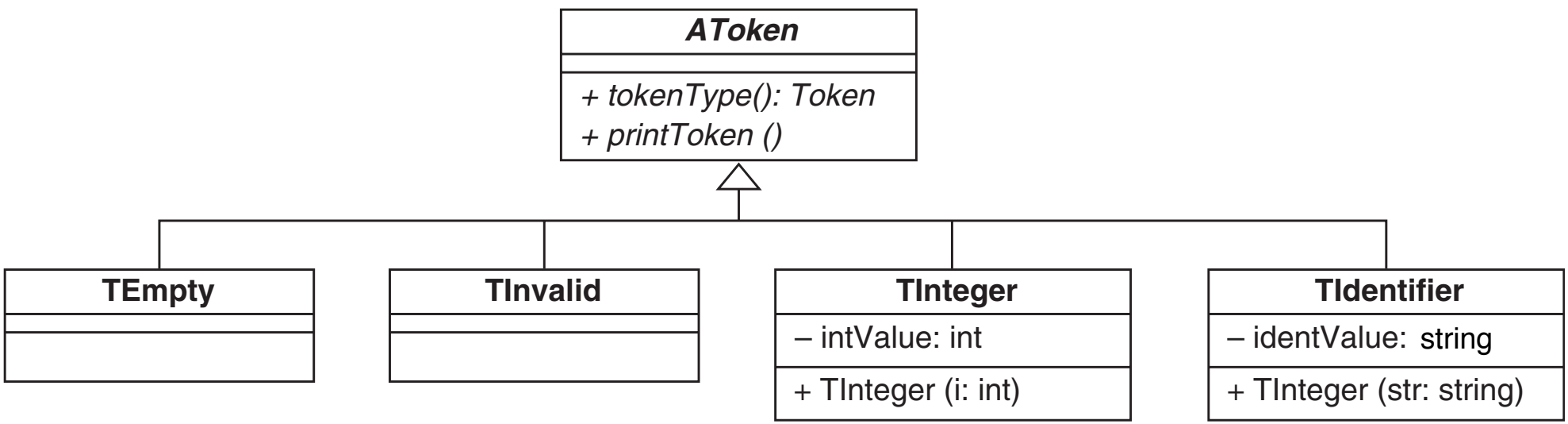


Figure 7.31

```
#include <iostream>
#include <cctype> // isalpha, isdigit
#include <string> // string, getline
using namespace std;

#include "inBuffer.hpp" // InBuffer
InBuffer inBuffer;

enum Token {eT_IDENTIFIER, eT_INTEGER, eT_EMPTY, eT_INVALID};

class AToken {
public:
   virtual Token tokenType () = 0;
   virtual void printToken () = 0;
};

class TEmpty : public AToken {
public:
   Token tokenType () { return eT_EMPTY; }
   void printToken () { cout << "Empty token" << endl; }
};
```

Figure 7.31
(Continued)

```
class TInvalid : public AToken {
public:
   Token tokenType () { return eT_INVALID; }
   void printToken () { cout << "Syntax error" << endl; }
};

class TInteger : public AToken {
private:
   int intValue;
public:
   TInteger (int i) { intValue = i; }
   Token tokenType () { return eT_INTEGER; }
   void printToken () { cout << "Integer    = " << intValue << endl; }
};

class TIdentifier : public AToken {
private:
   string identValue;
public:
   TIdentifier (string str) { identValue = str; }
   Token tokenType () { return eT_IDENTIFIER; }
   void printToken () { cout << "Identifier = " << identValue << endl; }
};
```

Figure 7.32

```
enum State {eS_START, eS_IDENT, eS_SIGN, eS_INTEGER, eS_STOP};

void getToken (AToken*& pAT) {
   // Pre: pAT is set to a token object.
   // Post: pAT is set to a token object that corresponds to
   // the next token in the input buffer.
   State state;
   char nextChar;
   int sign;
   int localIntValue;
   string localIdentValue;
   delete pAT;
   pAT = new TEmpty;
   state = eS_START;
   do {
      inBuffer.advanceInput (nextChar);
      switch (state) {
```

Figure 7.32
(Continued)

```
case eS_START:
   if (isalpha(nextChar)) {
      localIdentValue = nextChar;
      state = eS_IDENT;
   }
   else if (nextChar == '-') {
      sign = -1;
      state = eS_SIGN;
   }
   else if (nextChar == '+') {
      sign = +1;
      state = eS_SIGN;
   }
   else if (isdigit(nextChar)) {
      localIntValue = nextChar - '0';
      sign = +1;
      state = eS_INTEGER;
   }
   else if (nextChar == '\n') {
      state = eS_STOP;
   }
   else if (nextChar != ' ') {
      delete pAT;
      pAT = new TInvalid;
   }
   break;
```

Figure 7.32
(Continued)

```
case eS_IDENT:
   if (isalpha (nextChar) || isdigit (nextChar)) {
      localIdentValue.push_back (nextChar);
   }
   else {
      inBuffer.backUpInput ();
      delete pAT;
      pAT = new TIdentifier (localIdentValue);
      state = eS_STOP;
   }
   break;
case eS_SIGN:
   if (isdigit (nextChar)) {
      localIntValue = nextChar - '0';
      state = eS_INTEGER;
   }
   else {
      delete pAT;
      pAT = new TInvalid;
   }
   break;
```

Figure 7.32
(Continued)

```
      case eS_INTEGER:
         if (isdigit (nextChar)) {
            localIntValue = 10 * localIntValue + nextChar - '0';
         }
         else {
            inBuffer.backUpInput ();
            delete pAT;
            pAT = new TInteger (sign * localIntValue);
            state = eS_STOP;
         }
         break;
      }
   }
   while ((state != eS_STOP) && (pAT->tokenType () != eT_INVALID));
}
```

Figure 7.33

```
int main () {
   AToken* pAToken = new TEmpty;
   inBuffer.getLine ();
   while (!cin.eof ()) {
      do {
         getToken (pAToken);
         pAToken->printToken ();
      }
      while ((pAToken->tokenType () != eT_EMPTY)
          && (pAToken->tokenType () != eT_INVALID));
      inBuffer.getLine ();
   }
   delete pAToken;
   return 0;
}
```

A design principle for multiple-token recognizers

You can never fail once you reach a final state. Instead, if the final state does not have a transition from it on the terminal just input, you have recognized a token and should back up the input. The character will then be available as the first terminal for the next token.

Figure 7.34
Input/Output

Input

```
set (Time, 15)
set (Accel, 3)
set (TSquared, Time)
MUL (TSquared, Time)
set (Position, TSquared)
mul (Position, Accel)
div (Position, 2)
end
```

Output

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

Figure 7.34
Input/Output

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

```
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: The argument of "neg" is malformed.
Error: Illegal trailing character.
Error: Missing "end" sentinel.
```

Figure 7.35

```
// ======== Global tables
enum Mnemon {
    eM_ADD, eM_SUB, eM_MUL, eM_DIV,
    eM_NEG, eM_SET, eM_END, eM_EMPTY
};

map <Mnemon, string> operatorTable;
map <string, Mnemon> mnemonTable;

void initGlobalTables () {
    operatorTable [eM_ADD] = "+";
    operatorTable [eM_SUB] = "-";
    operatorTable [eM_MUL] = "*";
    operatorTable [eM_DIV] = "/";
    mnemonTable ["add"] = eM_ADD;
    mnemonTable ["sub"] = eM_SUB;
    mnemonTable ["mul"] = eM_MUL;
    mnemonTable ["div"] = eM_DIV;
    mnemonTable ["neg"] = eM_NEG;
    mnemonTable ["set"] = eM_SET;
    mnemonTable ["end"] = eM_END;
}
```

Figure 7.35
(Continued)

```
void lookUpMnemon (string id, Mnemon& mn, bool& fnd) {
   string lowerId = "";
   for (int i = 0; i < id.size (); i++) {
      lowerId.push_back (tolower (id[i]));
   }
   if (mnemonTable.count (lowerId) == 1) {
      mn = mnemonTable[lowerId];
      fnd = true;
   }
   else {
      fnd = false;
   }
}
```

Figure 7.36

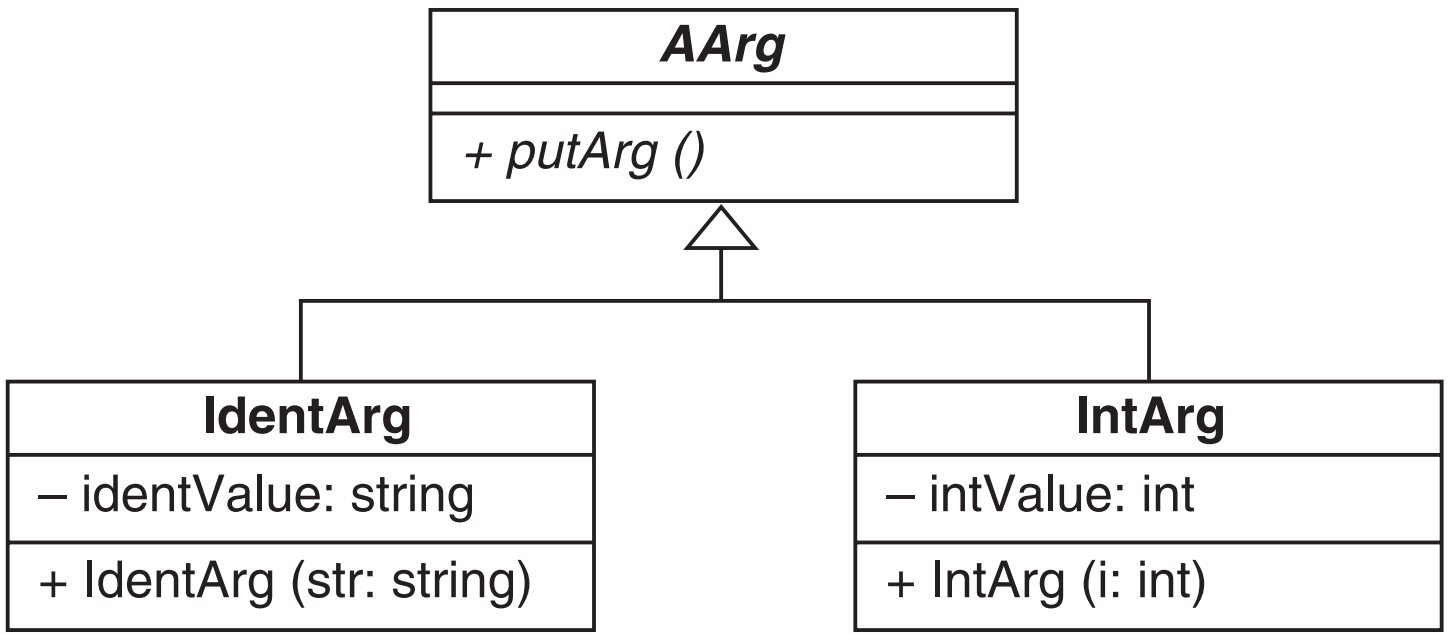


Figure 7.37

```
// ======== The argument classes
class AArg {
public:
   virtual void putArg () = 0;
};

class IdentArg : public AArg {
private:
   string identValue;
public:
   IdentArg (string str) { identValue = str; }
   void putArg () { cout << identValue; }
};

class IntArg : public AArg {
private:
   int intValue;
public:
   IntArg (int i) { intValue = i; }
   void putArg () { cout << intValue; }
};
```

Figure 7.38

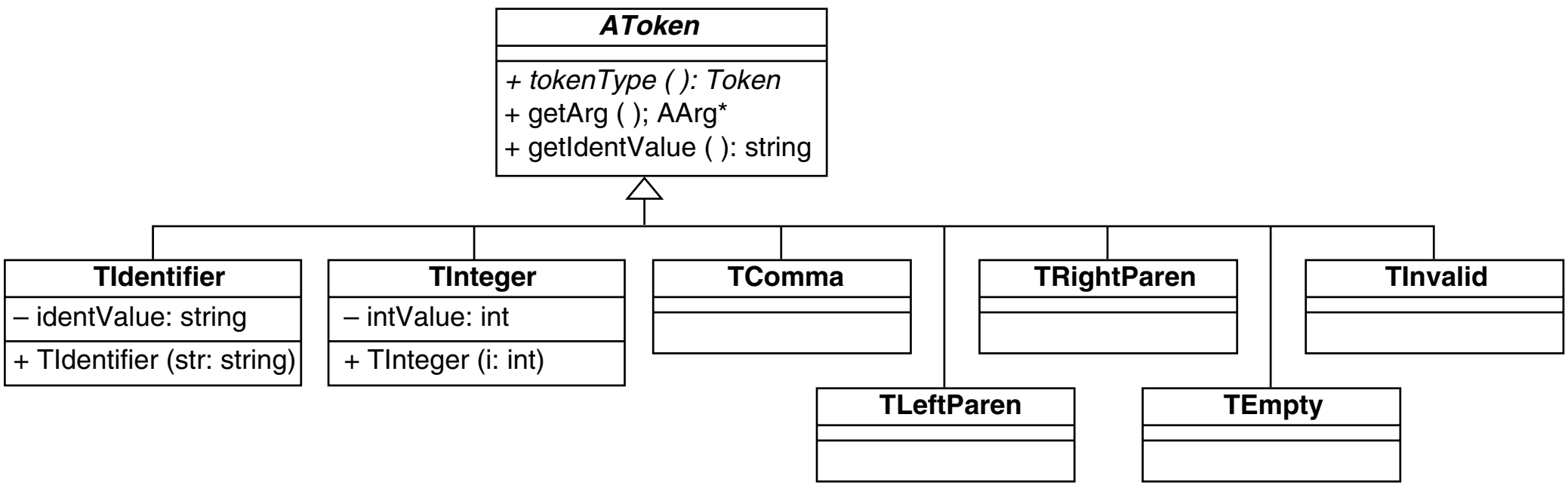


Figure 7.39

```
// ======== The token classes
enum Token {
   eT_IDENTIFIER, eT_INTEGER, eT_COMMA, eT_LEFT_PAREN,
   eT_RIGHT_PAREN, eT_EMPTY, eT_INVALID
};

class AToken {
public:
   virtual Token tokenType () = 0;
   virtual AArg* getArg () { return 0; }
   virtual string getIdentValue () { return ""; }
};

class TIdentifier : public AToken {
private:
   string identValue;
public:
   TIdentifier (string str) { identValue = str; }
   AArg* getArg () {
      AArg* pArg = new IdentArg (identValue);
      return pArg;
   }
   Token tokenType () { return eT_IDENTIFIER; }
   string getIdentValue () { return identValue; }
};
```

Figure 7.39
(Continued)

```
class TInteger : public AToken {
private:
   int intValue;
public:
   TInteger (int i) { intValue = i; }
   AArg* getArg () {
      IntArg* pArg = new IntArg (intValue);
      return pArg;
   }
   Token tokenType () { return eT_INTEGER; }
};


class TComma : public AToken {
public:
   Token tokenType () { return eT_COMMA; }
};
```

Figure 7.39
(Continued)

```
class TLeftParen : public AToken {
public:
   Token tokenType () { return eT_LEFT_PAREN; }
};

class TRightParen : public AToken {
public:
   Token tokenType () { return eT_RIGHT_PAREN; }
};

class TEmpty : public AToken {
public:
   Token tokenType () { return eT_EMPTY; }
};

class TInvalid : public AToken {
public:
   Token tokenType () { return eT_INVALID; }
};
```

Figure 7.40

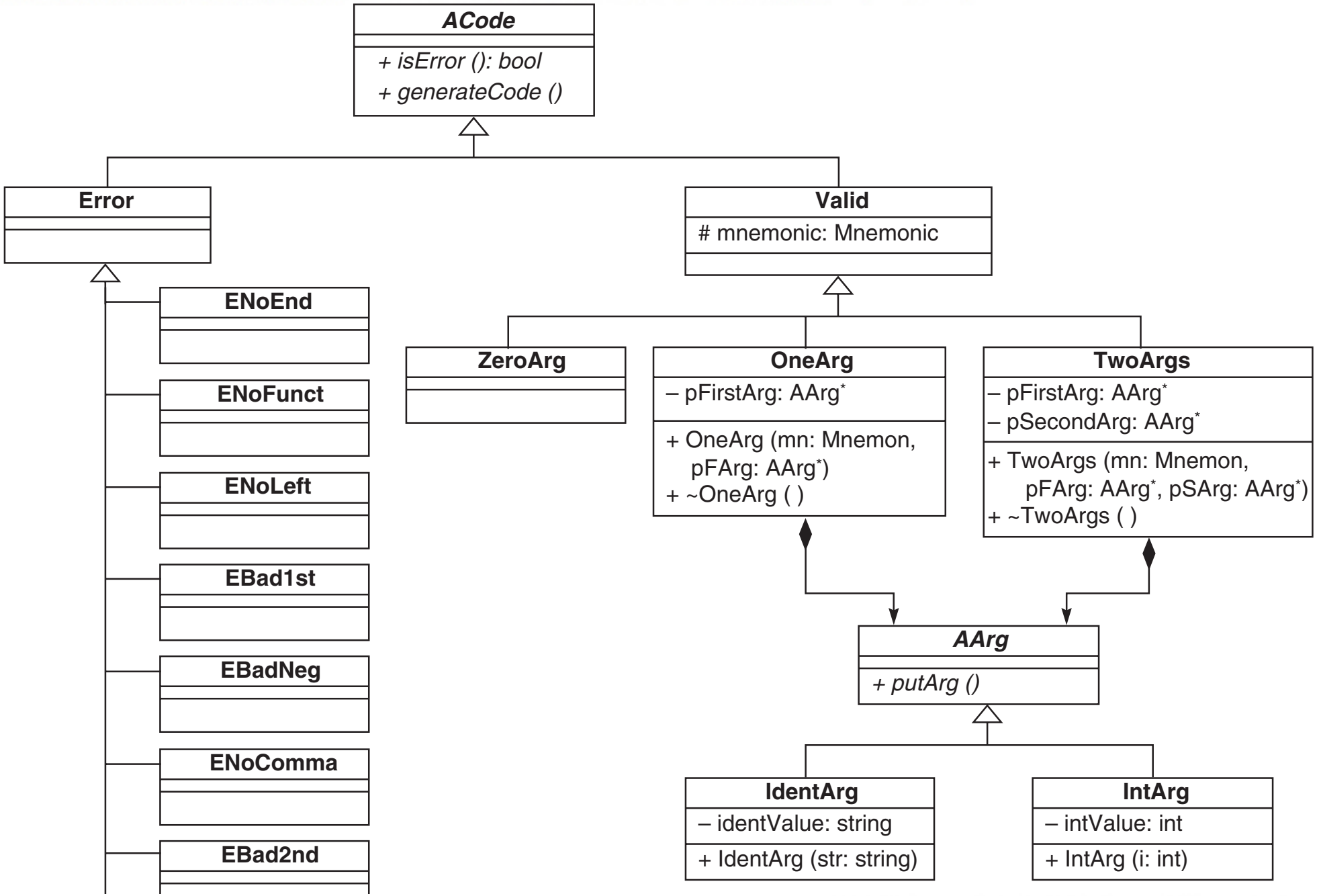


Figure 7.41

```
// ======== The code classes
class ACode {
public:
   virtual bool isError () = 0;
   virtual void generateCode () = 0;
};


// ======== The error code classes
class Error : public ACode {
public:
   bool isError () { return true; }
};

class ENoEnd : public Error {
public:
   void generateCode () {
      cout << "Error: Missing \"end\" sentinel.";
   }
};
```

Figure 7.41
(Continued)

```
class ENoFunct : public Error {
public:
   void generateCode () {
      cout << "Error: Line must begin with function identifier.";
   }
};

class ENoLeft : public Error {
public:
   void generateCode () {
      cout << "Error: Left parenthesis expected after function.";
   }
};

class EBad1st : public Error {
public:
   void generateCode () {
      cout << "Error: First argument not an identifier.";
   }
};
```

Figure 7.41
(Continued)

```
class EBadNeg : public Error {
public:
   void generateCode () {
      cout << "Error: The argument of \"neg\" is malformed.";
   }
};

class ENoComma : public Error {
public:
   void generateCode () {
      cout << "Error: Comma expected after first argument.";
   }
};

class EBad2nd : public Error {
public:
   void generateCode () {
      cout << "Error: Second argument not an identifier or integer.";
   }
};
```

Figure 7.41
(Continued)

```
class ENoRight : public Error {
public:
   void generateCode () {
      cout << "Error: Right parenthesis expected after argument.";
   }
};

class EExtraChars : public Error {
public:
   void generateCode () {
      cout << "Error: Illegal trailing character.";
   }
};
```

Figure 7.41
(Continued)

```
// ======== The valid code classes
class Valid : public ACode {
protected:
   Mnemon mnemonic;
public:
   bool isError () { return false; }
};

class ZeroArg : public Valid {
public:
   ZeroArg (Mnemon mn) { mnemonic = mn; }
   void generateCode () { } // cout nothing
};
```

Figure 7.41
(Continued)

```
class OneArg : public Valid {
private:
   AArg *pFirstArg;
public:
   OneArg (Mnemon mn, AArg* pFArg) {
      mnemonic = mn;
      pFirstArg = pFArg;
   }
   ~OneArg () { delete pFirstArg; }
   void generateCode () {
      pFirstArg->putArg ();
      cout << " := -";
      pFirstArg->putArg ();
   }
};
```

Figure 7.41
(Continued)

```
class TwoArgs : public Valid {
private:
   AArg *pFirstArg, *pSecondArg;
public:
   TwoArgs (Mnemon mn, AArg* pFArg, AArg* pSArg) {
      mnemonic = mn;
      pFirstArg = pFArg;
      pSecondArg = pSArg;
   }
   ~TwoArgs () {
      delete pFirstArg;
      delete pSecondArg;
   }
```

Figure 7.41
(Continued)

```
    void generateCode () {
        switch (mnemonic) {
        case eM_SET:
            pFirstArg->putArg ();
            cout << " := ";
            pSecondArg->putArg ();
            break;
        case eM_ADD: case eM_SUB: case eM_MUL: case eM_DIV:
            pFirstArg->putArg ();
            cout << " := ";
            pFirstArg->putArg ();
            cout << " " << operatorTable[mnemonic] << " ";
            pSecondArg->putArg ();
            break;
        }
    }
};
```

Figure 7.42

```
// ======== The lexical analyzer
enum LexState {
   eLS_START, eLS_IDENT, eLS_SIGN, eLS_INTEGER, eLS_STOP
};


void getToken (AToken*& pAT) {
   // Pre: pAT is allocated to an irrelevant value.
   char nextChar;
   string localIdentValue;
   int localIntValue;
   int sign;
   delete pAT;
   pAT = new TEmpty;
   LexState state = eLS_START;
   do {
      inBuffer.advanceInput (nextChar);
      switch (state) {
```

Figure 7.42
(Continued)

```
case eLS_START:
   if (isalpha (nextChar)) {
      localIdentValue = nextChar;
      state = eLS_IDENT;
   }
   else if (nextChar == '-') {
      sign = -1;
      state = eLS_SIGN;
   }
   else if (nextChar == '+') {
      sign = +1;
      state = eLS_SIGN;
   }
   else if (isdigit (nextChar)) {
      localIntValue = nextChar - '0';
      sign = +1;
      state = eLS_INTEGER;
   }
   else if (nextChar == ',') {
      delete pAT;
      pAT = new TComma;
      state = eLS_STOP;
   }
```

Figure 7.42
(Continued)

```
else if (nextChar == '(') {
   delete pAT;
   pAT = new TLeftParen;
   state = eLS_STOP;
}
else if (nextChar == ')') {
   delete pAT;
   pAT = new TRightParen;
   state = eLS_STOP;
}
else if (nextChar == '\n') {
   // pAT is initialized to TEmpty.
   state = eLS_STOP;
}
else if (nextChar != ' ') {
   delete pAT;
   pAT = new TInvalid;
}
break;
```

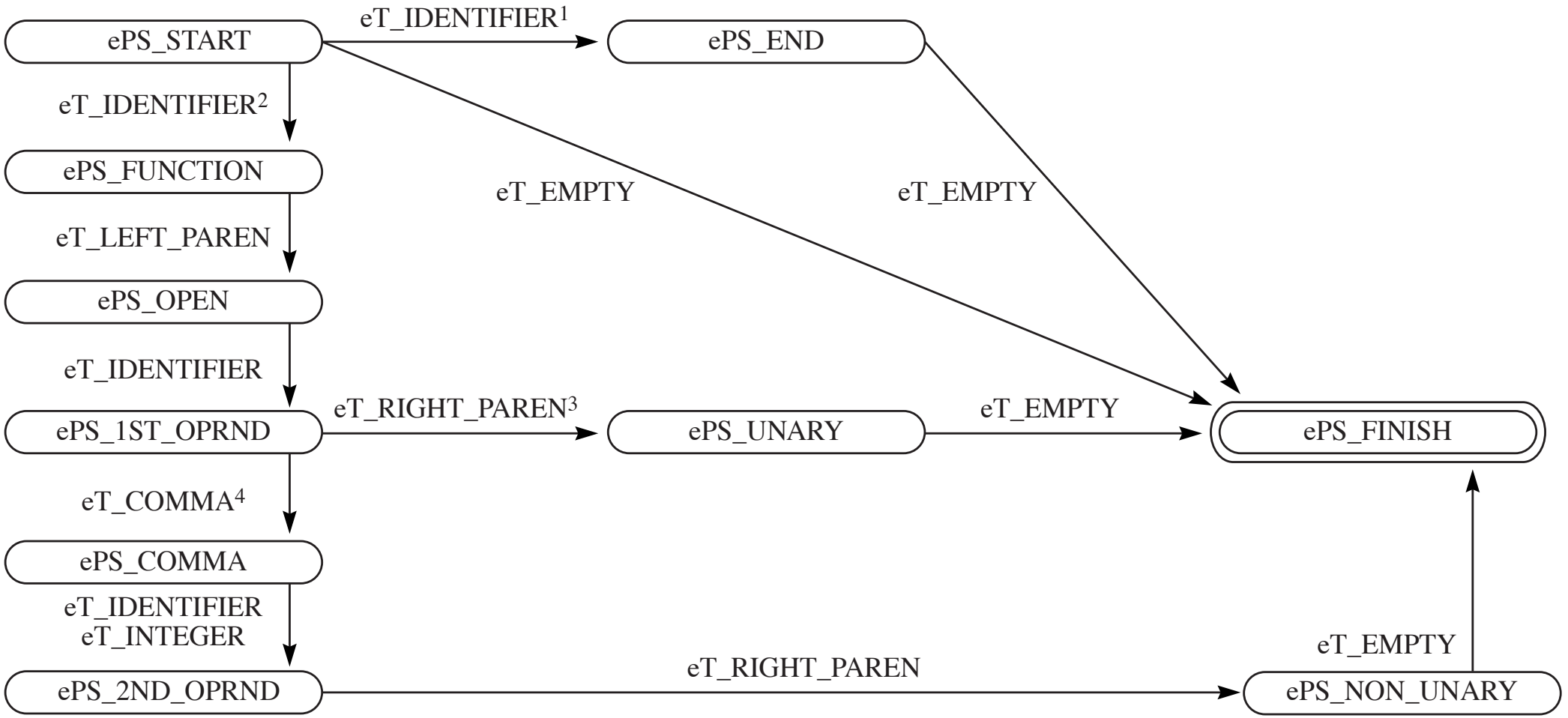
Figure 7.42
(Continued)

```
case eLS_IDENT:
   if (isalpha (nextChar) || isdigit (nextChar)) {
      localIdentValue.push_back (nextChar);
   }
   else {
      inBuffer.backUpInput();
      delete pAT;
      pAT = new TIdentifier (localIdentValue);
      state = eLS_STOP;
   }
   break;
case eLS_SIGN:
   if (isdigit (nextChar)) {
      localIntValue = nextChar - '0';
      state = eLS_INTEGER;
   }
   else {
      delete pAT;
      pAT = new TInvalid;
   }
   break;
```

Figure 7.42
(Continued)

```
      case eLS_INTEGER:
         if (isdigit (nextChar)) {
            localIntValue = 10 * localIntValue + nextChar - '0';
         }
         else {
            inBuffer.backUpInput();
            delete pAT;
            pAT = new TInteger (sign * localIntValue);
            state = eLS_STOP;
         }
         break;
      }
   }
   while ((state != eLS_STOP) && (pAT->tokenType () != eT_INVALID));
}
```

Figure 7.43



Note 1: Only the identifier `end`.
Note 2: Only the identifiers `set`, `add`, `sub`, `mul`, `div`, and `neg`.
Note 3: Only for mnemonic eM_NEG.
Note 4: Only for mnemonics eM_SET, eM_ADD, eM_SUB, eM_MUL, and eM_DIV.

Figure 7.44

```
// ======== The parser
enum ParseState {
   ePS_START, ePS_END, ePS_FUNCTION, ePS_OPEN, ePS_1ST_OPRND,
   ePS_UNARY, ePS_COMMA, ePS_2ND_OPRND, ePS_NON_UNARY, ePS_FINISH
};

void processSourceLine (ACode*& pAC, bool& term) {
   // Pre: term is false.
   // A source line is in inBuffer ready for processing.
   // pAC is allocated to an irrelevant value.
   AArg* pLocalFirstArg;
   AArg* pLocalSecondArg;
   Mnemon mnemon;
   delete pAC;
   pAC = new ZeroArg (eM_EMPTY);
   AToken* pAToken = new TEmpty;
   ParseState state = ePS_START;
   do {
      getToken (pAToken);
      switch (state) {
```

Figure 7.44
(Continued)

```
case ePS_START:
   if (pAToken->tokenType () == eT_IDENTIFIER) {
      string tempStr = pAToken->getIdentValue ();
      bool found;
      lookUpMnemon (tempStr, mnemon, found);
      if (found) {
         if (mnemon == eM_END) {
            delete pAC;
            pAC = new ZeroArg (eM_END);
            term = true;
            state = ePS_END;
         }
         else {
            state = ePS_FUNCTION;
         }
      }
      else {
         delete pAC;
         pAC = new ENoFunct;
      }
   }
```

Figure 7.44
(Continued)

```
else if (pAToken->tokenType () == eT_EMPTY) {
   // pAC initialized to ZeroArg (eM_EMPTY)
   state = ePS_FINISH;
}
else {
   delete pAC;
   pAC = new ENoFunct;
}
break;
```

Figure 7.44
(Continued)

```
case ePS_END:
   if (pAToken->tokenType () == eT_EMPTY) {
      state = ePS_FINISH;
   }
   else {
      delete pAC;
      pAC = new EExtraChars;
   }
   break;
case ePS_FUNCTION:
   if (pAToken->tokenType () == eT_LEFT_PAREN) {
      state = ePS_OPEN;
   }
   else {
      delete pAC;
      pAC = new ENoLeft;
   }
   break;
```

Figure 7.44
(Continued)

```
case ePS_OPEN:
   if (pAToken->tokenType () == eT_IDENTIFIER) {
      pLocalFirstArg = pAToken->getArg ();
      state = ePS_1ST_OPRND;
   }
   else {
      delete pAC;
      pAC = new EBad1st;
   }
   break;
```

Figure 7.44
(Continued)

```
case ePS_1ST_OPRND:
   if (mnemon == eM_NEG) {
      if (pAToken->tokenType () == eT_RIGHT_PAREN) {
         delete pAC;
         pAC = new OneArg (mnemon, pLocalFirstArg);
         state = ePS_UNARY;
      }
      else {
         delete pAC;
         pAC = new EBadNeg;
      }
   }
   else if (pAToken->tokenType () == eT_COMMA) {
      state = ePS_COMMA;
   }
   else {
      delete pAC;
      pAC = new ENoComma;
   }
   break;
```

Figure 7.44
(Continued)

```
case ePS_UNARY:
   if (pAToken->tokenType () == eT_EMPTY) {
      state = ePS_FINISH;
   }
   else {
      delete pAC;
      pAC = new EExtraChars;
   }
   break;
case ePS_COMMA:
   if ((pAToken->tokenType () == eT_IDENTIFIER)
   ||  (pAToken->tokenType () == eT_INTEGER)) {
      pLocalSecondArg = pAToken->getArg ();
      delete pAC;
      pAC = new TwoArgs (mnemon, pLocalFirstArg, pLocalSecondArg);
      state = ePS_2ND_OPRND;
   }
   else {
      delete pAC;
      pAC = new EBad2nd;
   }
   break;
```

Figure 7.44
(Continued)

```
      case ePS_2ND_OPRND:
         if (pAToken->tokenType () == eT_RIGHT_PAREN) {
            state = ePS_NON_UNARY;
         }
         else {
            delete pAC;
            pAC = new ENoRight;
         }
         break;
      case ePS_NON_UNARY:
         if (pAToken->tokenType () == eT_EMPTY) {
            state = ePS_FINISH;
         }
         else {
            delete pAC;
            pAC = new EExtraChars;
         }
         break;
      }
   }
   while ((state != ePS_FINISH) && (!pAC->isError ()));
   delete pAToken;
}
```

Figure 7.45

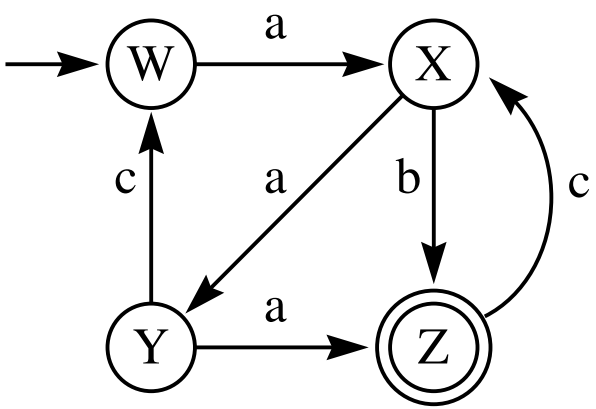
```
// ======== The main program

int main () {
   bool terminateWithEnd = false;
   ACode* pACode = new ZeroArg (eM_EMPTY);
   initGlobalTables ();
   inBuffer.getLine ();
   while (! (cin.eof() || terminateWithEnd)) {
      processSourceLine (pACode, terminateWithEnd);
      pACode->generateCode ();
      cout << endl;
      inBuffer.getLine ();
   }
   if (!terminateWithEnd) {
      delete pACode;
      pACode = new ENoEnd;
      pACode->generateCode ();
      cout << endl;
   }
   delete pACode;
   return 0;
}
```

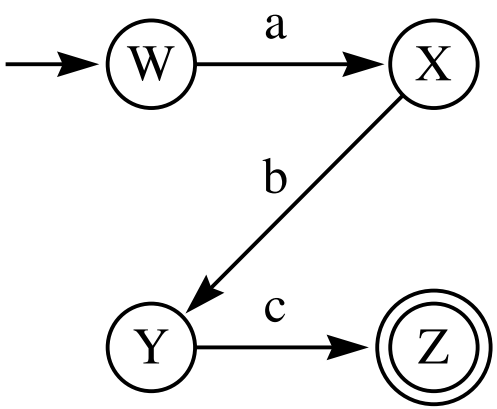
$$
\begin{aligned}
N &= \{\mathrm{A}, \mathrm{B}\} \\
T &= \{\mathtt{0}, \mathtt{1}\} \\
P &= \text{the productions} \\
&\quad 1.\ \mathrm{A} \rightarrow \mathtt{0B} \\
&\quad 2.\ \mathrm{B} \rightarrow \mathtt{10B} \\
&\quad 3.\ \mathrm{B} \rightarrow \epsilon \\
S &= \mathrm{A}
\end{aligned}
$$

$$
\begin{aligned}
N &= \{\mathrm{C}\} \\
T &= \{\mathtt{0}, \mathtt{1}\} \\
P &= \text{the productions} \\
&\quad 1.\ \mathrm{C} \rightarrow \mathrm{C}\mathtt{10} \\
&\quad 2.\ \mathrm{C} \rightarrow \mathtt{0} \\
S &= \mathrm{C}
\end{aligned}
$$

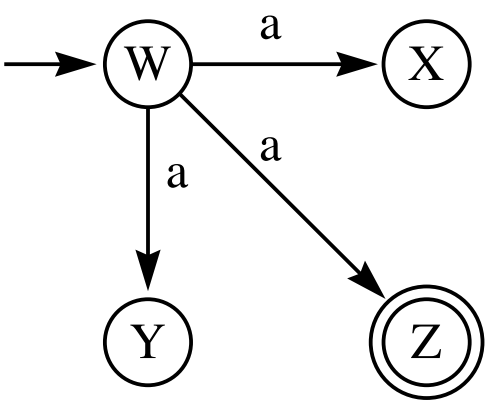
Figure 7.48



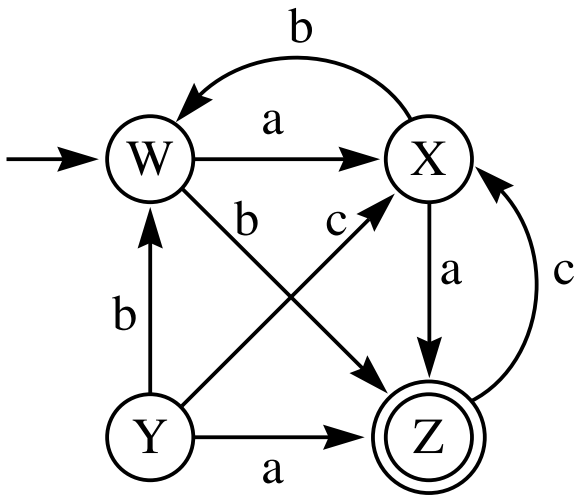
*(a)



(b)

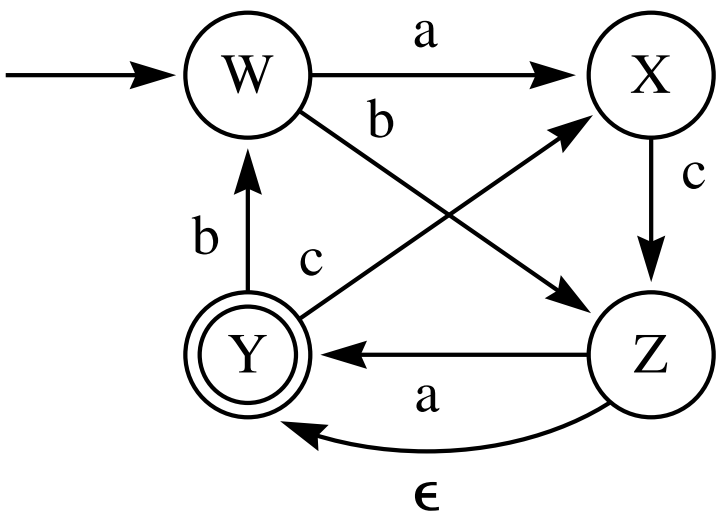


(c)

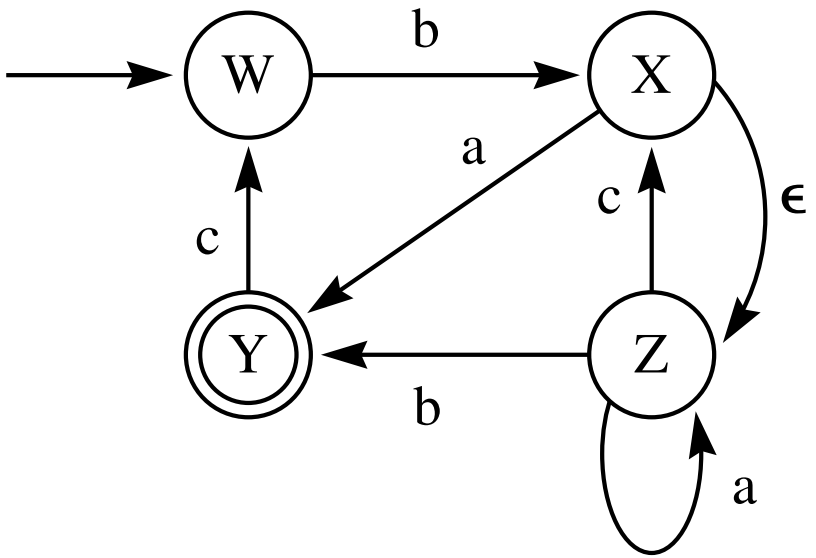


(d)

Figure 7.49

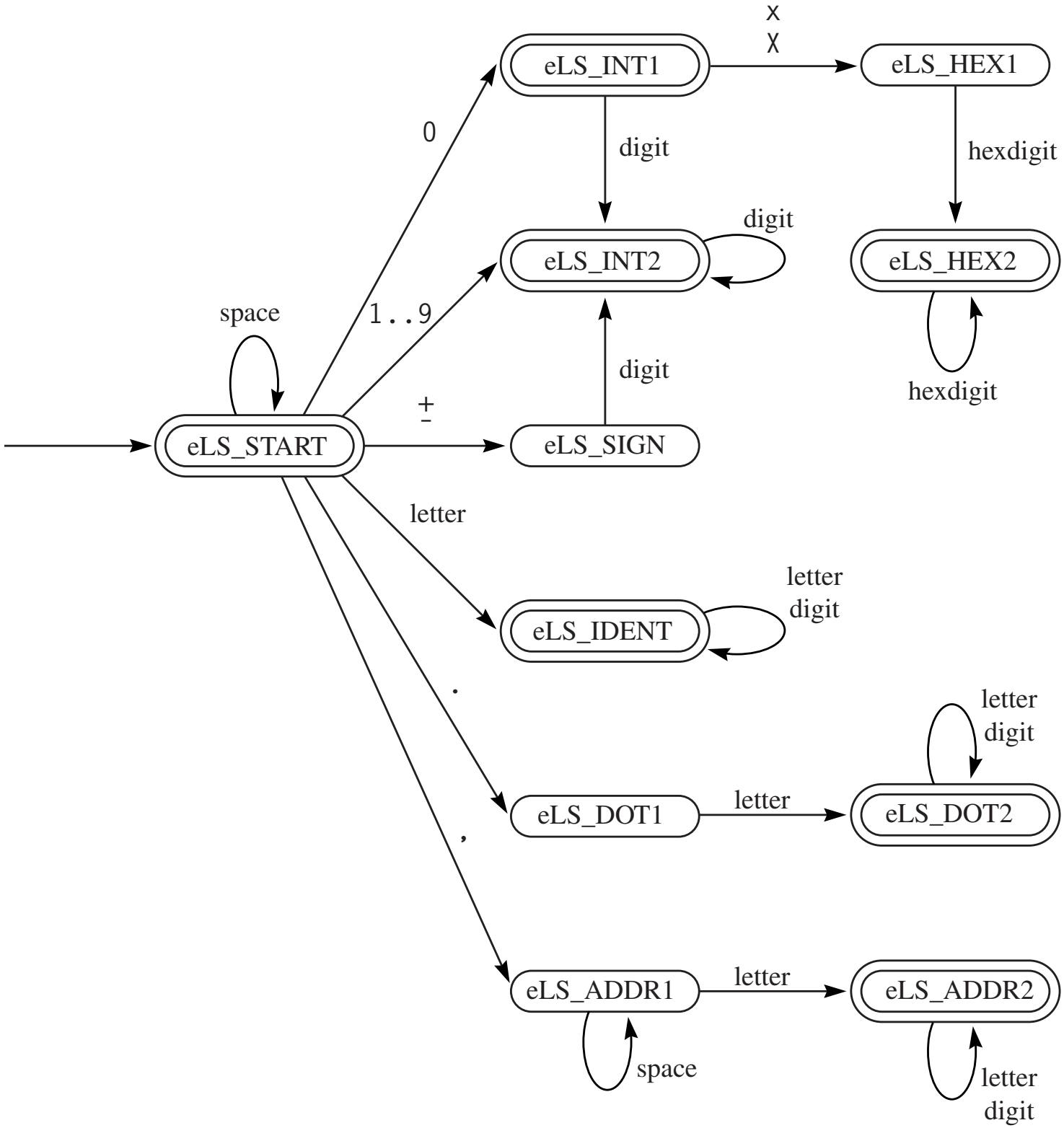


(a)



(b)

Figure 7.50


eLS_START
eLS_INT1
eLS_INT2
eLS_HEX1
eLS_HEX2
eLS_SIGN
eLS_IDENT
eLS_DOT1
eLS_DOT2
eLS_ADDR1
eLS_ADDR2
space
0
1..9
+
-
digit
hexdigit
letter
.
,
X
X