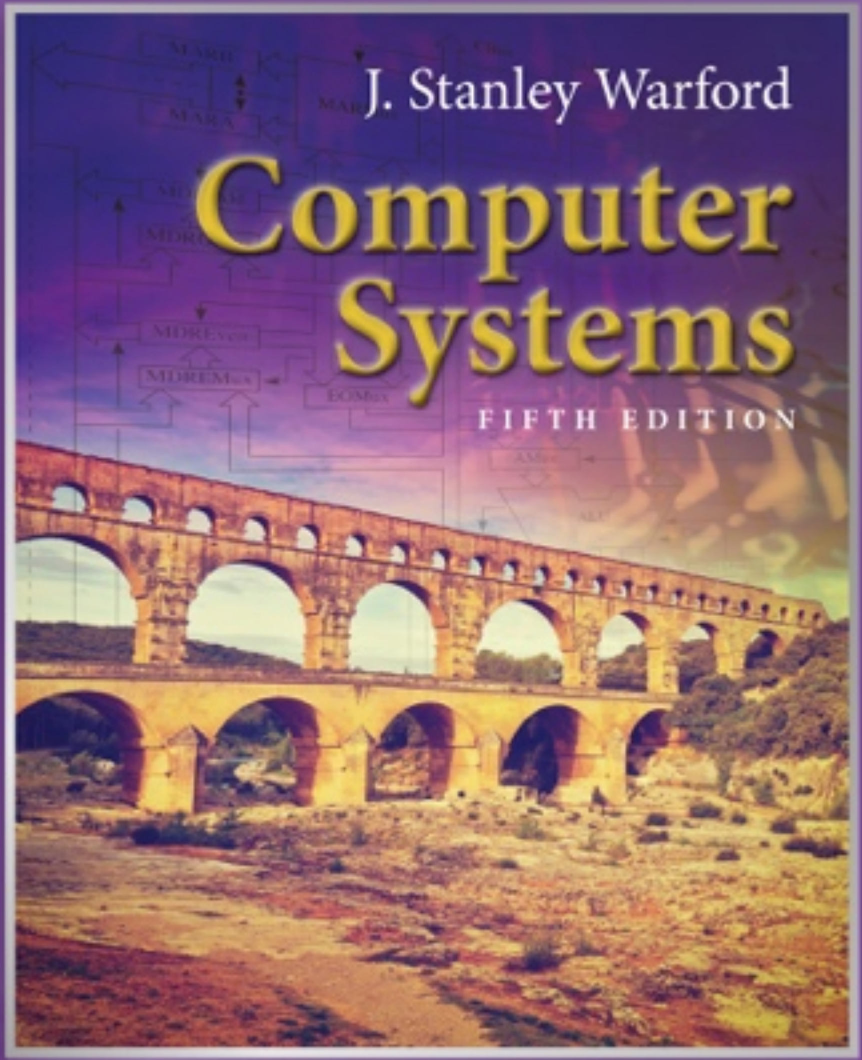
Chapter 7

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 Pep/9 assembly language 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 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

$$\varepsilon x = x\varepsilon = x$$

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 = { <identifier> , <letter> , <digit> }

T = { `a` , `b` , `c` , `1` , `2` , `3` }

P = the productions

1. <identifier> → <letter>
2. <identifier> → <identifier> <letter>
3. <identifier> → <identifier> <digit>
4. <letter> → `a`
5. <letter> → `b`
6. <letter> → `c`
7. <digit> → `1`
8. <digit> → `2`
9. <digit> → `3`

S = <identifier>

A derivation

<identifier>

A derivation

<identifier> ⇒ <identifier> <digit> Rule 3

A derivation

<identifier>	$\Rightarrow$ <identifier> <digit>	Rule 3
	$\Rightarrow$ <identifier> 3	Rule 9

A derivation

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> 3	Rule 9
	⇒ <identifier> <letter> 3	Rule 2

A derivation

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> 3	Rule 9
	⇒ <identifier> <letter> 3	Rule 2
	⇒ <identifier> b 3	Rule 5

A derivation

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> `3`	Rule 9
	⇒ <identifier> <letter> `3`	Rule 2
	⇒ <identifier> `b 3`	Rule 5
	⇒ <identifier> <letter> `b 3`	Rule 2

A derivation

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> `3`	Rule 9
	⇒ <identifier> <letter> `3`	Rule 2
	⇒ <identifier> `b 3`	Rule 5
	⇒ <identifier> <letter> `b 3`	Rule 2
	⇒ <identifier> `a b 3`	Rule 4

A derivation

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> 3	Rule 9
	⇒ <identifier> <letter> 3	Rule 2
	⇒ <identifier> b 3	Rule 5
	⇒ <identifier> <letter> b 3	Rule 2
	⇒ <identifier> a b 3	Rule 4
	⇒ <letter> a b 3	Rule 1

A derivation

<identifier>	⇒ <identifier> <digit>	Rule 3
	⇒ <identifier> `3`	Rule 9
	⇒ <identifier> <letter> `3`	Rule 2
	⇒ <identifier> `b 3`	Rule 5
	⇒ <identifier> <letter> `b 3`	Rule 2
	⇒ <identifier> `a b 3`	Rule 4
	⇒ <letter> `a b 3`	Rule 1
	⇒ `c a b 3`	Rule 6

A derivation

You can summarize the previous eight derivation steps as

<identifier> $\Rightarrow^*$ `c a b 3`

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

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

$P =$ the productions

1. $\mathrm{I} \rightarrow \mathrm{FM}$
2. $\mathrm{F} \rightarrow +$
3. $\mathrm{F} \rightarrow -$
4. $\mathrm{F} \rightarrow \varepsilon$
5. $\mathrm{M} \rightarrow \mathrm{dM}$
6. $\mathrm{M} \rightarrow \mathrm{d}$

$S = \mathrm{I}$

Alternative notation for production rules

$$I \rightarrow FM$$

$$F \rightarrow + \mid - \mid \varepsilon$$

$$M \rightarrow d \mid dM$$

Some derivations

$$
\begin{aligned}
\mathrm{I} &\Rightarrow \mathrm{FM} \\
&\Rightarrow \mathrm{F}\texttt{d}\mathrm{M} \\
&\Rightarrow \mathrm{F}\texttt{dd}\mathrm{M} \\
&\Rightarrow \mathrm{F}\texttt{ddd} \\
&\Rightarrow \texttt{-ddd}
\end{aligned}
$$

Some derivations

$$\begin{aligned} I &\Rightarrow \texttt{FM} \\ &\Rightarrow \texttt{FdM} \\ &\Rightarrow \texttt{FddM} \\ &\Rightarrow \texttt{Fddd} \\ &\Rightarrow \texttt{-ddd} \end{aligned}$$

$$\begin{aligned} I &\Rightarrow \texttt{FM} \\ &\Rightarrow \texttt{FdM} \\ &\Rightarrow \texttt{Fdd} \\ &\Rightarrow \texttt{dd} \end{aligned}$$

Some derivations

$$\begin{aligned} I &\Rightarrow FM \\ &\Rightarrow FdM \\ &\Rightarrow FddM \\ &\Rightarrow Fddd \\ &\Rightarrow -ddd \end{aligned}$$

$$\begin{aligned} I &\Rightarrow FM \\ &\Rightarrow FdM \\ &\Rightarrow Fdd \\ &\Rightarrow dd \end{aligned}$$

$$\begin{aligned} I &\Rightarrow FM \\ &\Rightarrow FdM \\ &\Rightarrow FddM \\ &\Rightarrow FdddM \\ &\Rightarrow Fdddd \\ &\Rightarrow +dddd \end{aligned}$$

Grammars

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

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

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

$P =$ the productions

1. $\mathtt{A} \rightarrow \mathtt{aABC}$
2. $\mathtt{A} \rightarrow \mathtt{abC}$
3. $\mathtt{CB} \rightarrow \mathtt{BC}$
4. $\mathtt{bB} \rightarrow \mathtt{bb}$
5. $\mathtt{bC} \rightarrow \mathtt{bc}$
6. $\mathtt{cC} \rightarrow \mathtt{cc}$

$S = \mathtt{A}$

A derivation

A

A derivation

$A \Rightarrow aABC$ Rule 1

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

A derivation

$$
\begin{aligned}
A &\Rightarrow \mathtt{a}ABC && \text{Rule 1} \\
&\Rightarrow \mathtt{aa}ABCBC && \text{Rule 1} \\
&\vdots \\
&\vdots \\
&\vdots \\
&\vdots \\
&\vdots \\
&\vdots \\
&\vdots \\
&\vdots \\
&\vdots
\end{aligned}
$$

A derivation

A	$\Rightarrow$ aABC	Rule 1
	$\Rightarrow$ aaABCBC	Rule 1
	$\Rightarrow$ aaabCBCBC	Rule 2

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

$\vdots$

A derivation

$A \Rightarrow \texttt{a}ABC$	Rule 1
$\Rightarrow \texttt{aa}ABCBC$	Rule 1
$\Rightarrow \texttt{aaab}CBCBC$	Rule 2
$\Rightarrow \texttt{aaab}BCCBC$	Rule 3
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	
⋮	

A derivation

$A \Rightarrow$ aABC	Rule 1
$\Rightarrow$ aaABCBC	Rule 1
$\Rightarrow$ aaabCBCBC	Rule 2
$\Rightarrow$ aaabBCCBC	Rule 3
$\Rightarrow$ aaabBCBCC	Rule 3

$\vdots$

A derivation

A	$\Rightarrow$ aABC	Rule 1
	$\Rightarrow$ aaABCBC	Rule 1
	$\Rightarrow$ aaabCBCBC	Rule 2
	$\Rightarrow$ aaabBCCBC	Rule 3
	$\Rightarrow$ aaabBCBCC	Rule 3
	$\Rightarrow$ aaabBBCCC	Rule 3

⋮

⋮

⋮

⋮

⋮

A derivation

A	$\Rightarrow$ aABC	Rule 1
	$\Rightarrow$ aaABCBC	Rule 1
	$\Rightarrow$ aaabCBCBC	Rule 2
	$\Rightarrow$ aaabBCCBC	Rule 3
	$\Rightarrow$ aaabBCBCC	Rule 3
	$\Rightarrow$ aaabBBCCC	Rule 3
	$\Rightarrow$ aaabbBCCC	Rule 4

⋮

⋮

⋮

⋮

A derivation

$A \Rightarrow aABC$	Rule 1
$\Rightarrow aaABCBC$	Rule 1
$\Rightarrow aaabCBCBC$	Rule 2
$\Rightarrow aaabBCCBC$	Rule 3
$\Rightarrow aaabBCBCC$	Rule 3
$\Rightarrow aaabBBCCC$	Rule 3
$\Rightarrow aaabbBCCC$	Rule 4
$\Rightarrow aaabbbCCC$	Rule 4
$\vdots$	
$\vdots$	
$\vdots$	

A derivation

$A \Rightarrow aABC$	Rule 1
$\Rightarrow aaABCBC$	Rule 1
$\Rightarrow aaabCBCBC$	Rule 2
$\Rightarrow aaabBCCBC$	Rule 3
$\Rightarrow aaabBCBCC$	Rule 3
$\Rightarrow aaabBBCCC$	Rule 3
$\Rightarrow aaabbBCCC$	Rule 4
$\Rightarrow aaabbbCCC$	Rule 4
$\Rightarrow aaabbbcCC$	Rule 5

$\vdots$

$\vdots$

A derivation

A	$\Rightarrow$ aABC	Rule 1
	$\Rightarrow$ aaABCBC	Rule 1
	$\Rightarrow$ aaabCBCBC	Rule 2
	$\Rightarrow$ aaabBCCBC	Rule 3
	$\Rightarrow$ aaabBCBCC	Rule 3
	$\Rightarrow$ aaabBBCCC	Rule 3
	$\Rightarrow$ aaabbBCCC	Rule 4
	$\Rightarrow$ aaabbbCCC	Rule 4
	$\Rightarrow$ aaabbbcCC	Rule 5
	$\Rightarrow$ aaabbbccC	Rule 6

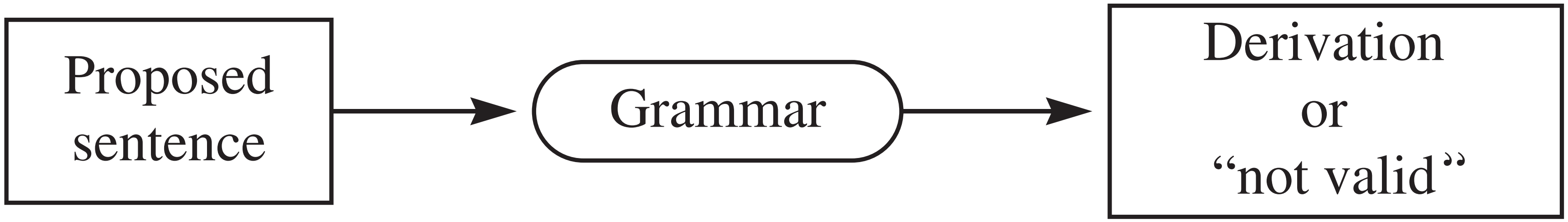
$\vdots$

A derivation

$A \Rightarrow \mathtt{a}ABC$	Rule 1
$\Rightarrow \mathtt{aa}ABCBC$	Rule 1
$\Rightarrow \mathtt{aaab}CBCBC$	Rule 2
$\Rightarrow \mathtt{aaab}BCCBC$	Rule 3
$\Rightarrow \mathtt{aaab}BCBCC$	Rule 3
$\Rightarrow \mathtt{aaab}BBCCC$	Rule 3
$\Rightarrow \mathtt{aaabb}BCCC$	Rule 4
$\Rightarrow \mathtt{aaabbb}CCC$	Rule 4
$\Rightarrow \mathtt{aaabbbc}CC$	Rule 5
$\Rightarrow \mathtt{aaabbbcc}C$	Rule 6
$\Rightarrow \mathtt{aaabbbccc}$	Rule 6

The parsing problem

(a) Deriving a valid sentence.



(b) The parsing problem.

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

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

$P =$ the productions

1. $\mathrm{E} \rightarrow \mathrm{E} + \mathrm{T}$
2. $\mathrm{E} \rightarrow \mathrm{T}$
3. $\mathrm{T} \rightarrow \mathrm{T} * \mathrm{F}$
4. $\mathrm{T} \rightarrow \mathrm{F}$
5. $\mathrm{F} \rightarrow \texttt{(}\ \mathrm{E}\ \texttt{)}$
6. $\mathrm{F} \rightarrow \mathrm{a}$

$S = \mathrm{E}$

Parse `( a * a ) + a`

E

Parse `( a * a ) + a`

$E \Rightarrow E + T$ Rule 1

Parse `( a * a ) + a`

$E \Rightarrow E + T$ Rule 1

$\Rightarrow T + T$ Rule 2

Parse `( a * a ) + a`

$E \Rightarrow E + T$	Rule 1
$\Rightarrow T + T$	Rule 2
$\Rightarrow F + T$	Rule 4

Parse `( a * a ) + a`

$E \Rightarrow E + T$	Rule 1
$\Rightarrow T + T$	Rule 2
$\Rightarrow F + T$	Rule 4
$\Rightarrow (E) + T$	Rule 5

Parse `( a * a ) + a`

$E \Rightarrow E + T$	Rule 1
$\Rightarrow T + T$	Rule 2
$\Rightarrow F + T$	Rule 4
$\Rightarrow (E) + T$	Rule 5
$\Rightarrow (T) + T$	Rule 2

Parse (a * a) + a

$$
\begin{aligned}
E &\Rightarrow E + T && \text{Rule 1}\\
&\Rightarrow T + T && \text{Rule 2}\\
&\Rightarrow F + T && \text{Rule 4}\\
&\Rightarrow (E) + T && \text{Rule 5}\\
&\Rightarrow (T) + T && \text{Rule 2}\\
&\Rightarrow (T * F) + T && \text{Rule 3}
\end{aligned}
$$

Parse (a * a) + a

$$
\begin{aligned}
E &\Rightarrow E + T && \text{Rule 1}\\
&\Rightarrow T + T && \text{Rule 2}\\
&\Rightarrow F + T && \text{Rule 4}\\
&\Rightarrow (E) + T && \text{Rule 5}\\
&\Rightarrow (T) + T && \text{Rule 2}\\
&\Rightarrow (T * F) + T && \text{Rule 3}\\
&\Rightarrow (F * F) + T && \text{Rule 4}
\end{aligned}
$$

Parse (a * a) + a

E ⇒ E + T	Rule 1
⇒ T + T	Rule 2
⇒ F + T	Rule 4
⇒ (E) + T	Rule 5
⇒ (T) + T	Rule 2
⇒ (T * F) + T	Rule 3
⇒ (F * F) + T	Rule 4
⇒ (a * F) + T	Rule 6

Parse `( a * a ) + a`

$E \Rightarrow E + T$	Rule 1
$\Rightarrow T + T$	Rule 2
$\Rightarrow F + T$	Rule 4
$\Rightarrow (E) + T$	Rule 5
$\Rightarrow (T) + T$	Rule 2
$\Rightarrow (T * F) + T$	Rule 3
$\Rightarrow (F * F) + T$	Rule 4
$\Rightarrow (a * F) + T$	Rule 6
$\Rightarrow (a * a) + T$	Rule 6

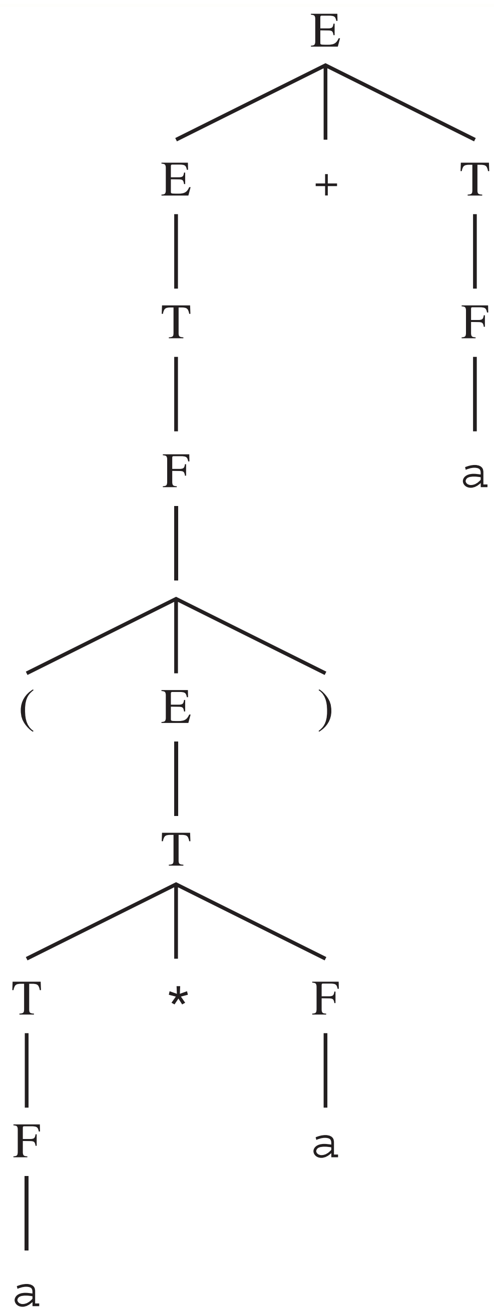
Parse (a * a) + a

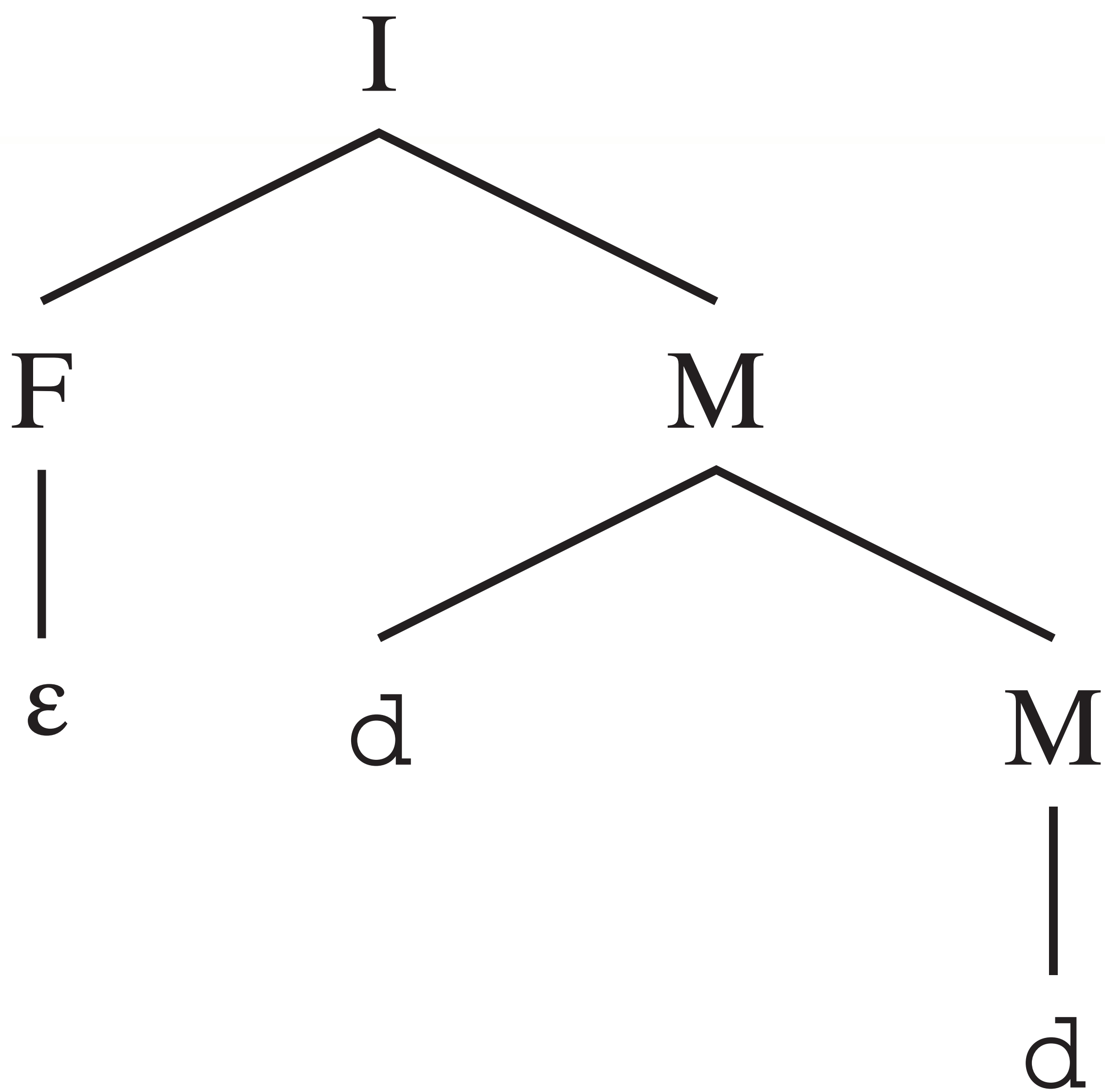
E ⇒ E + T	Rule 1
⇒ T + T	Rule 2
⇒ F + T	Rule 4
⇒ (E) + T	Rule 5
⇒ (T) + T	Rule 2
⇒ (T * F) + T	Rule 3
⇒ (F * F) + T	Rule 4
⇒ (a * F) + T	Rule 6
⇒ (a * a) + T	Rule 6
⇒ (a * a) + F	Rule 4

Parse `( a * a ) + a`

$E \Rightarrow E + T$	Rule 1
$\Rightarrow T + T$	Rule 2
$\Rightarrow F + T$	Rule 4
$\Rightarrow (E) + T$	Rule 5
$\Rightarrow (T) + T$	Rule 2
$\Rightarrow (T * F) + T$	Rule 3
$\Rightarrow (F * F) + T$	Rule 4
$\Rightarrow (\mathtt{a} * F) + T$	Rule 6
$\Rightarrow (\mathtt{a} * \mathtt{a}) + T$	Rule 6
$\Rightarrow (\mathtt{a} * \mathtt{a}) + F$	Rule 4
$\Rightarrow (\mathtt{a} * \mathtt{a}) + \mathtt{a}$	Rule 6

Figure 7.6




I
F
M
ε
d
M
d

<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 (continued)

<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>

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

<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> <declaration-list>

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

<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>)

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

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

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

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

<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

The following example of a parse with this grammar shows that

```
while ( a <= 9 )
      S1;
```

is a valid <statement>, assuming that *S1* is a valid <expression>.

<statement>

⇒ <iteration-statement>

⇒ `while` (<expression>) <statement>

⇒ `while` (<relational-expression>) <statement>

⇒ `while` (<relational-expression> `<=` <additive-expression>) <statement>

⇒ `while` (<additive-expression> `<=` <additive-expression>) <statement>

⇒ `while` (<multiplicative-expression> `<=` <additive-expression>) <statement>

⇒ `while` (<unary-expression> `<=` <additive-expression>) <statement>

⇒ `while` (<primary-expression> `<=` <additive-expression>) <statement>

⇒ `while` (<identifier> `<=` <additive-expression>) <statement>

⇒ `while` (<letter> `<=` <additive-expression>) <statement>

⇒ `while ( a <=` <additive-expression>) <statement>

⇒ `while ( a <=` <multiplicative-expression>) <statement>

⇒ `while ( a <=` <unary-expression>) <statement>

⇒ `while ( a <=` <primary-expression>) <statement>

⇒ `while ( a <=` <constant>) <statement>

⇒ `while ( a <=` <integer-constant>) <statement>

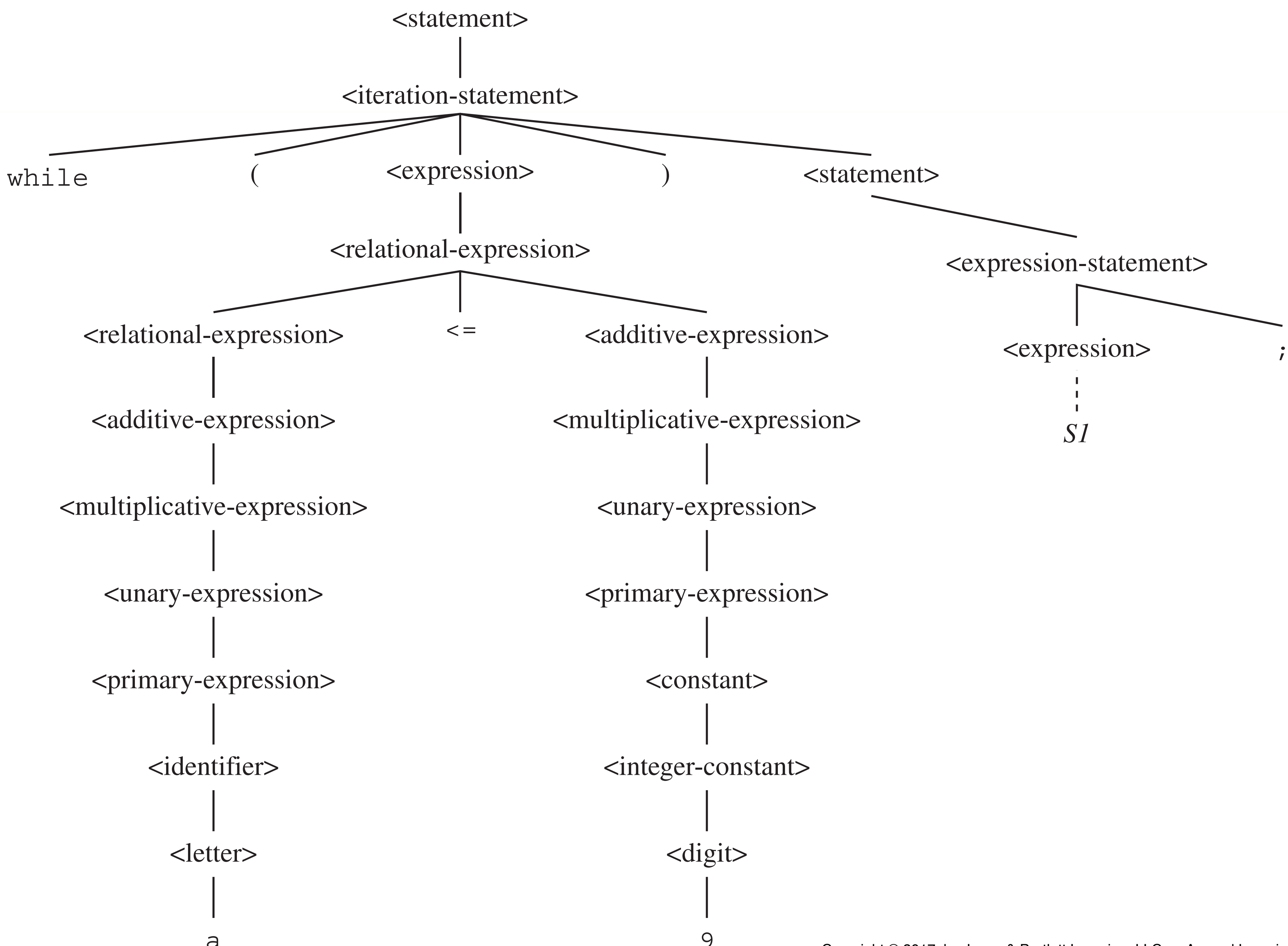
⇒ `while ( a <=` <digit>) <statement>

⇒ `while ( a <= 9 )` <statement>

⇒ `while ( a <= 9 )` <expression-statement>

⇒ `while ( a <= 9 )` <expression> ;

⇒* `while ( a <= 9 )` *S1* ;

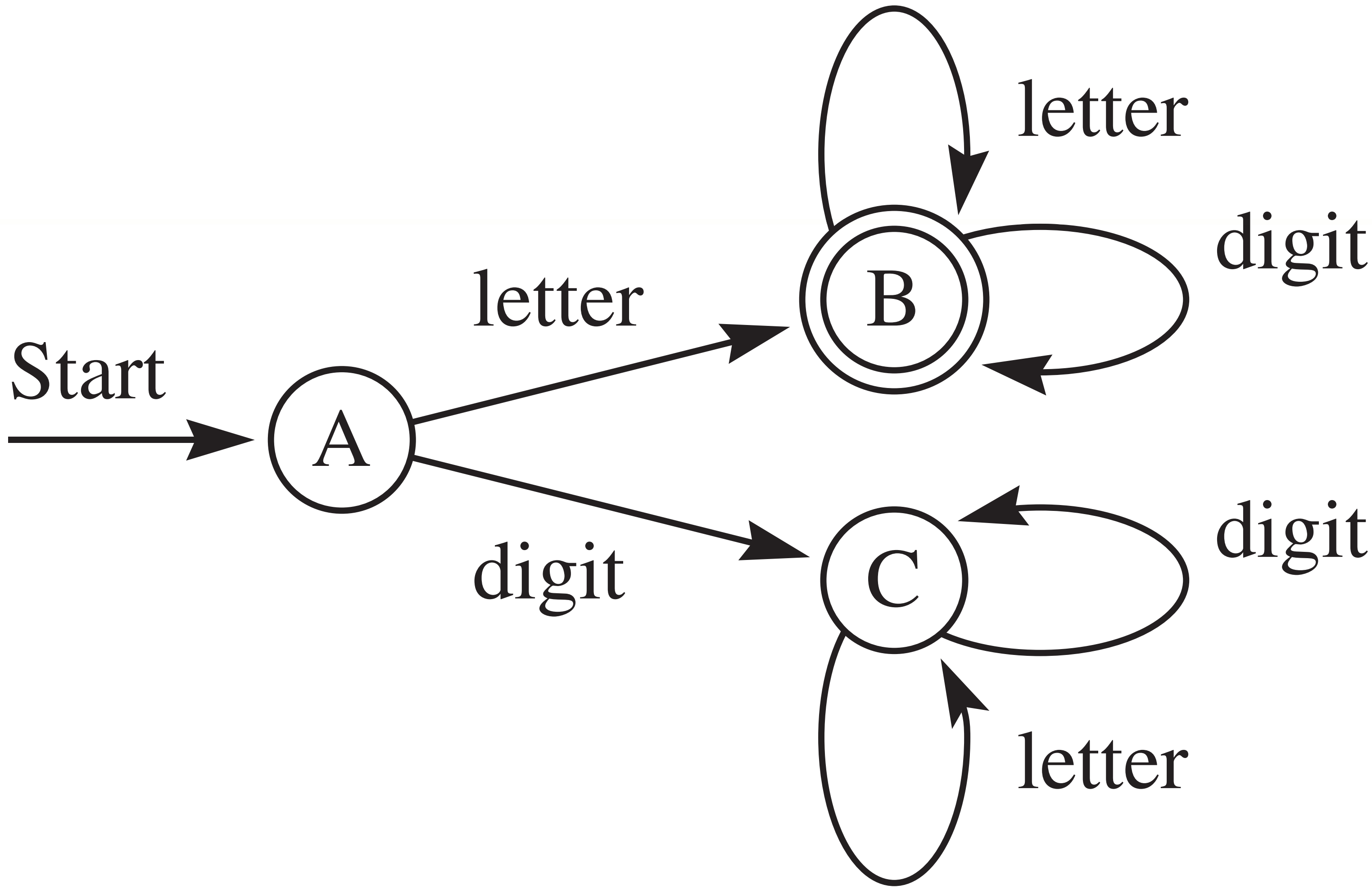

<statement>
<iteration-statement>
while
(
<expression>
)
<statement>
<relational-expression>
<expression-statement>
<relational-expression>
<=
<additive-expression>
<expression>
;
<additive-expression>
<multiplicative-expression>
S1
<multiplicative-expression>
<unary-expression>
<unary-expression>
<primary-expression>
<primary-expression>
<constant>
<identifier>
<integer-constant>
<letter>
<digit>
a
9

The C language

- C has a context-free grammar.
- C is not a context-free language.

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


letter
digit
letter
Start
A
B
digit
C
digit
letter

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	Next State: 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

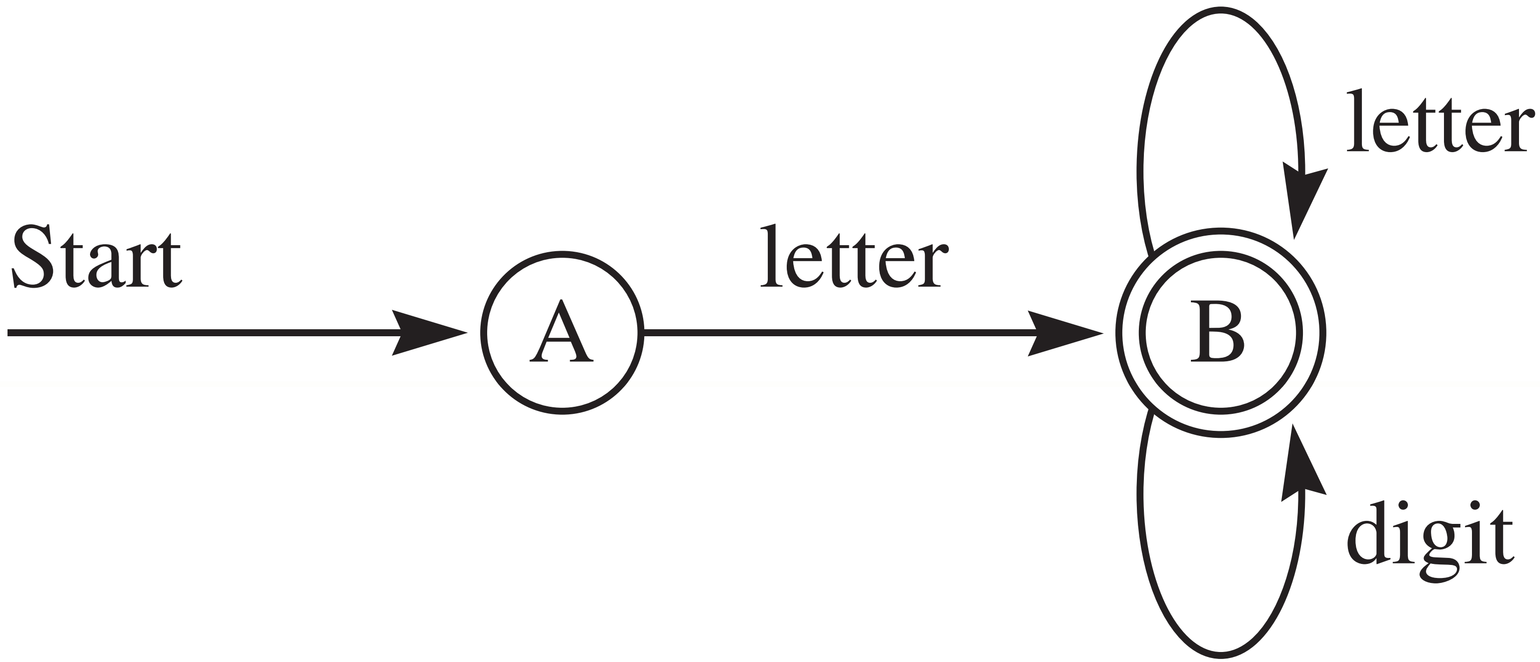
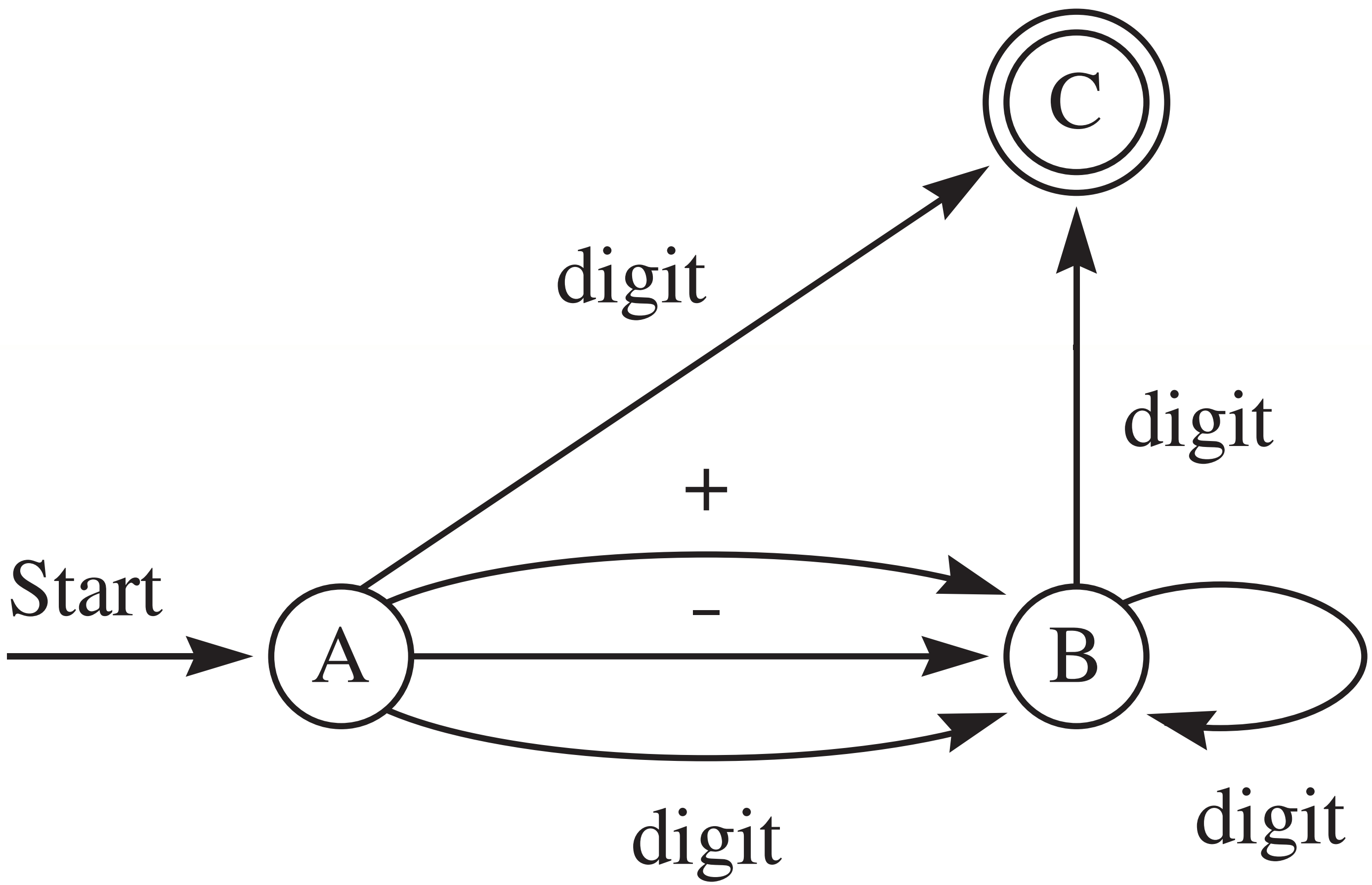

Start
A
letter
B
letter
digit

Figure 7.14

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

Nondeterministic FSM

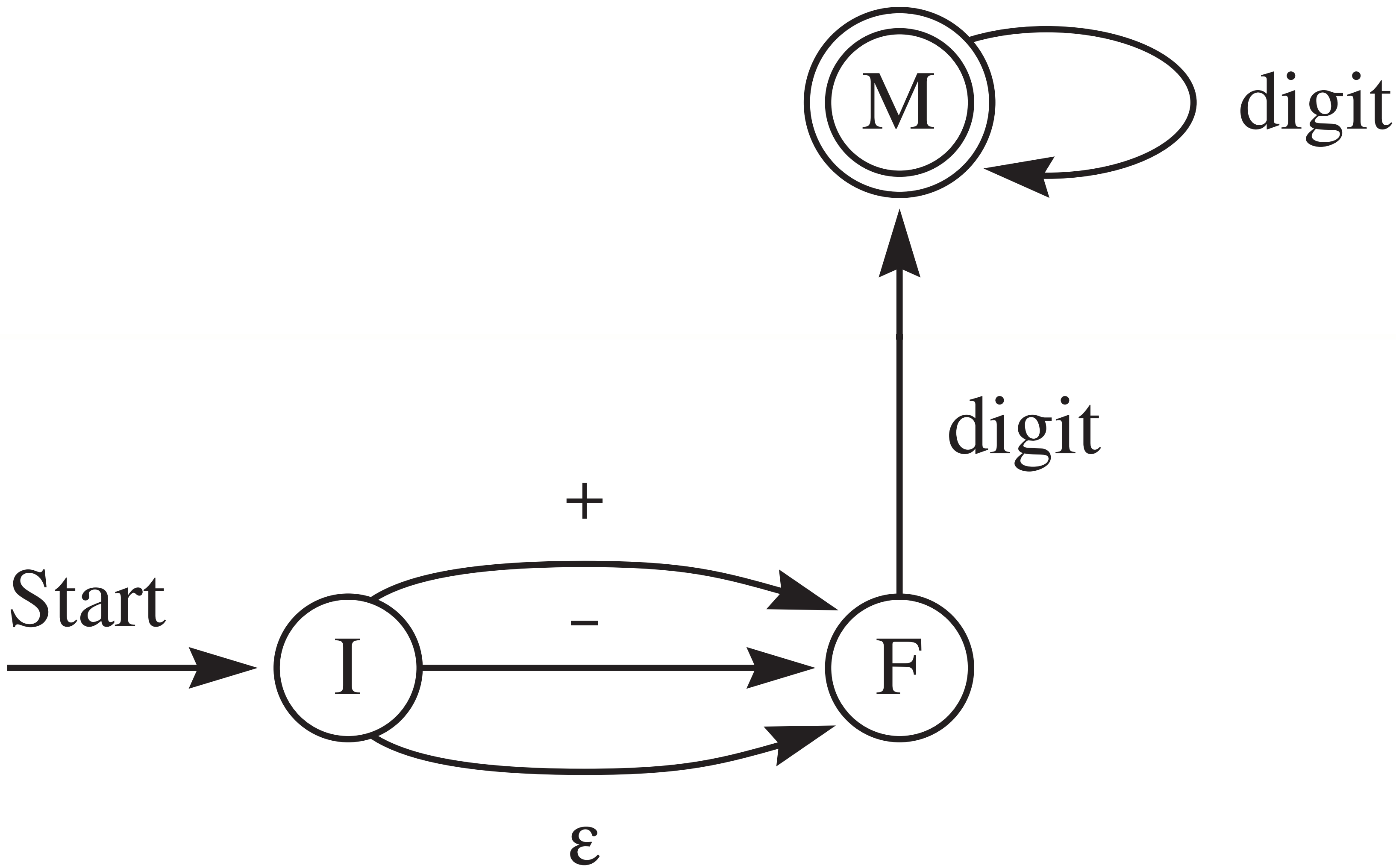
- At least one state has more than one transition from it on the same character
- If you scan the last character and you *are* in a final state, the string *is valid*
- If you scan the last character and you are *not* in a final state, the string *might be invalid*
- To prove invalid, you must try all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


C
digit
digit
+
Start
A
-
B
digit
digit

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

Empty transitions

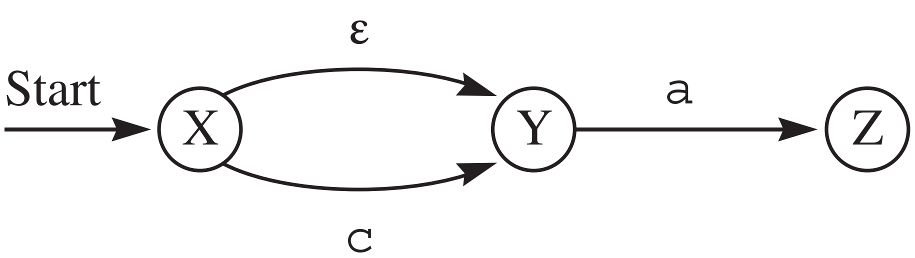
- An empty transition allows you to go from one state to another state without scanning a terminal character
- All finite state machines with empty transitions are considered nondeterministic


M
digit
digit
+
Start
I
–
F
ε

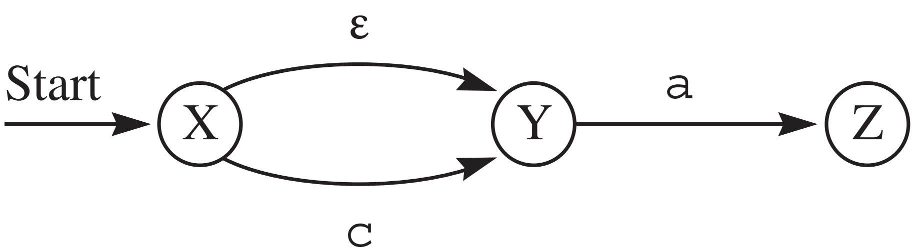
Current State	Next State			
	+	−	Digit	ε
→ I	F	F		F
F			M	
M			M	

Removing empty transitions

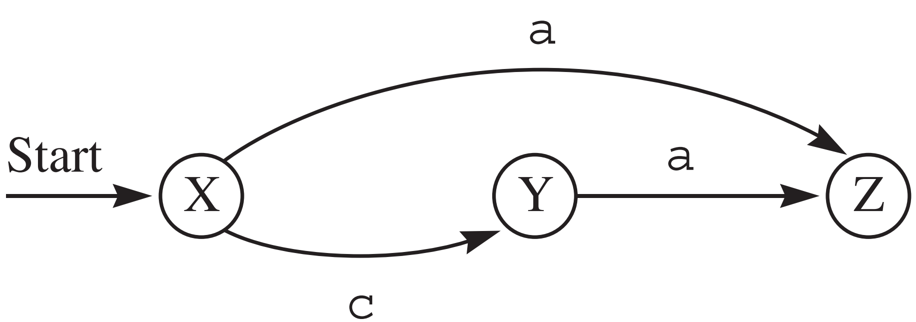
- Given a transition from p to q on ε, for every transition from q to r on a, add a transition from p to r on a.
- If q is a final state, make p a final state



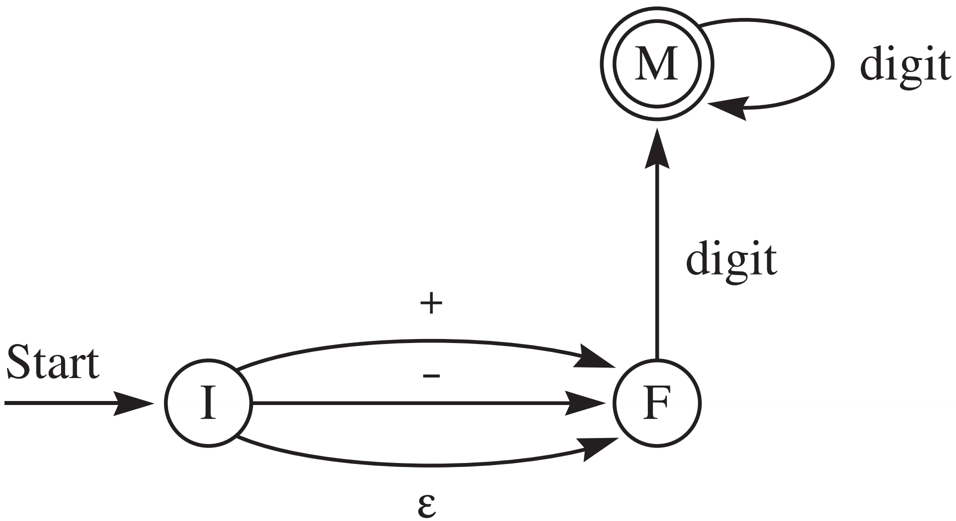
(a) The original FSM.



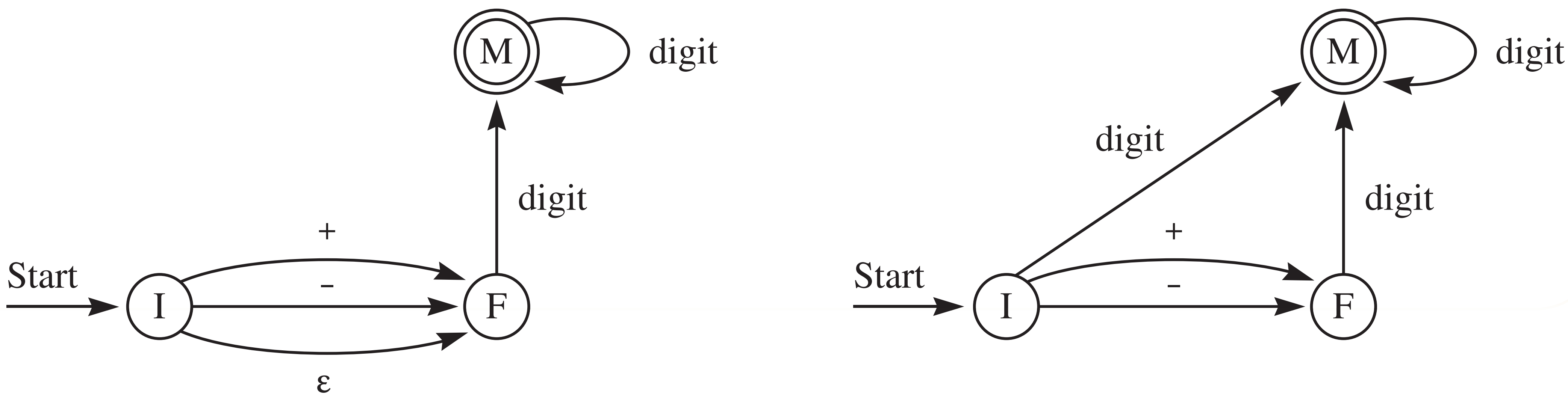
(a) The original FSM.



(b) The equivalent FSM without an empty transition.

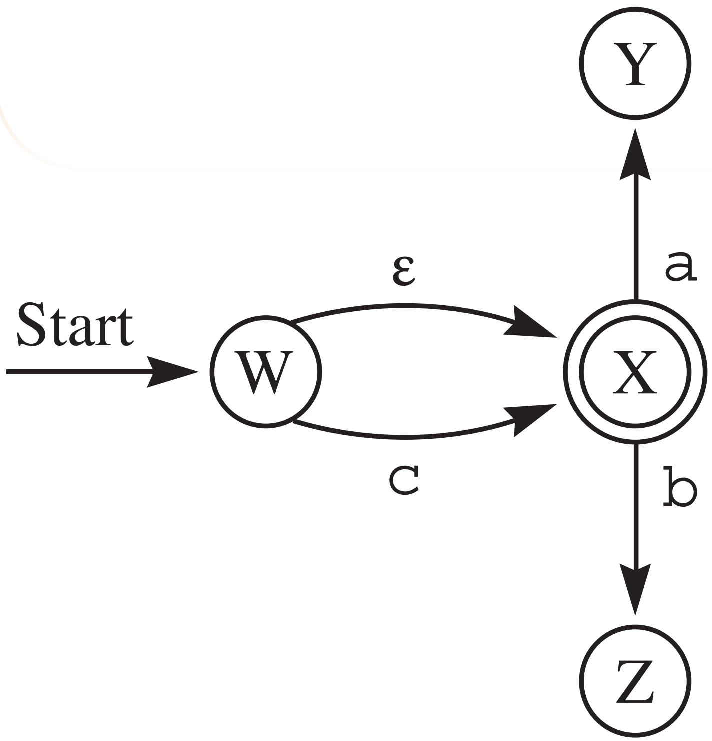


(a) The original FSM.



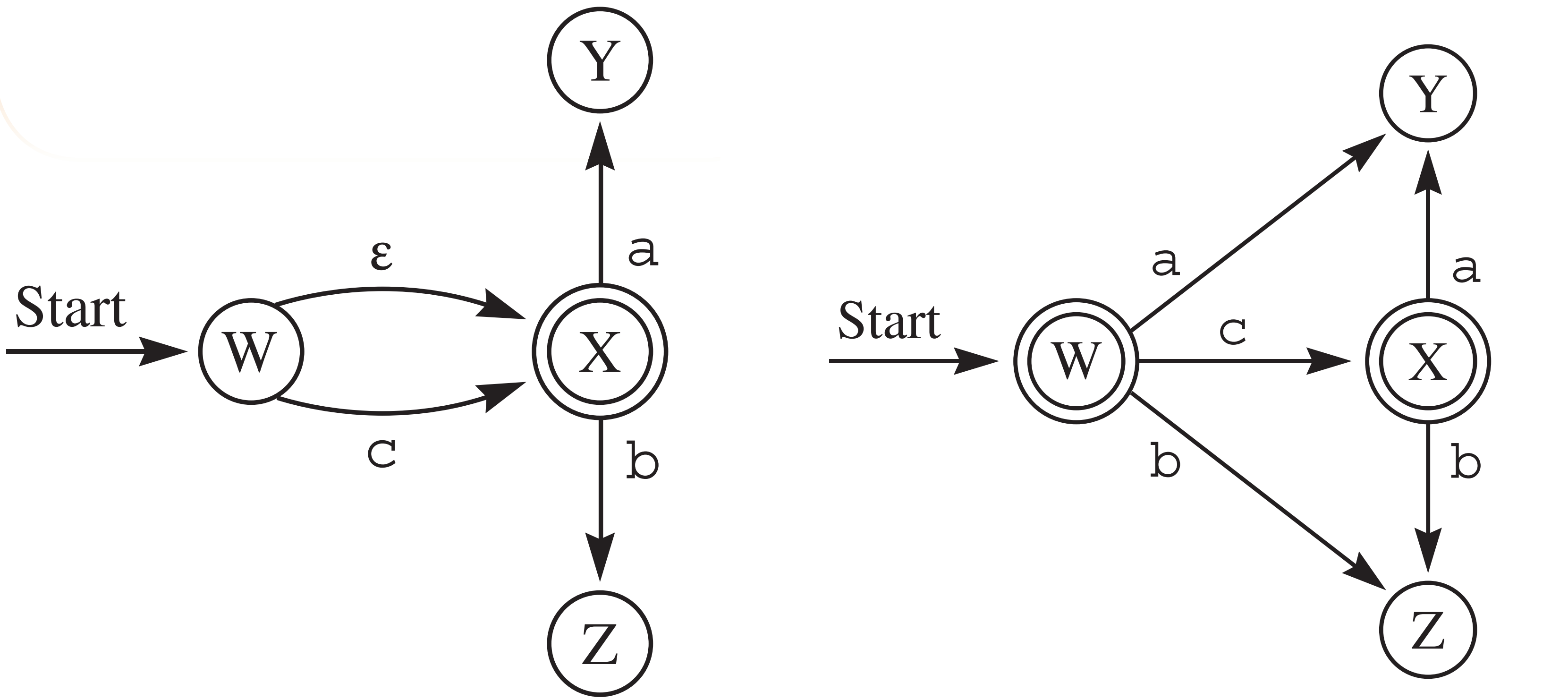
(a) The original FSM.

(b) The empty transition removed.



(a) The original FSM.

Figure 7.21

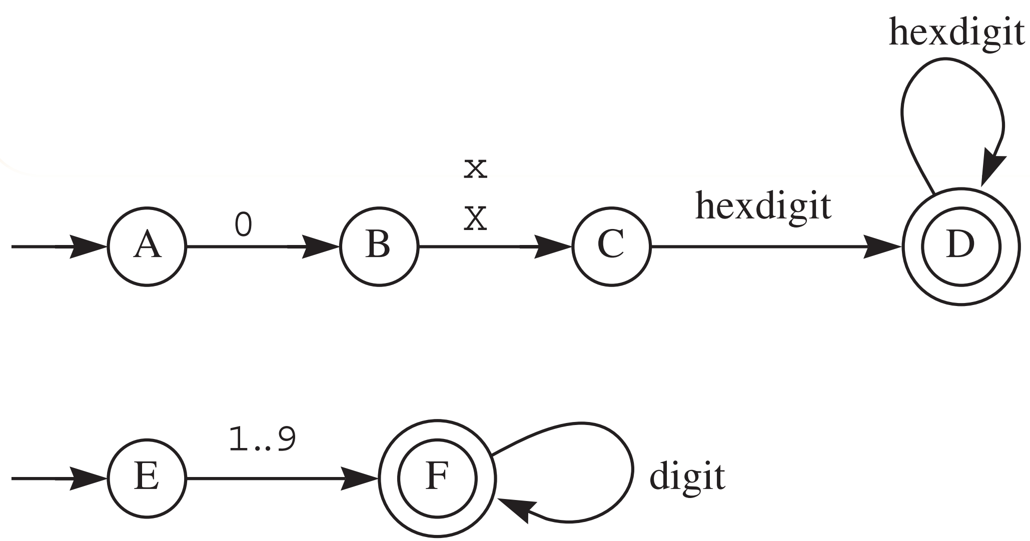


(a) The original FSM.

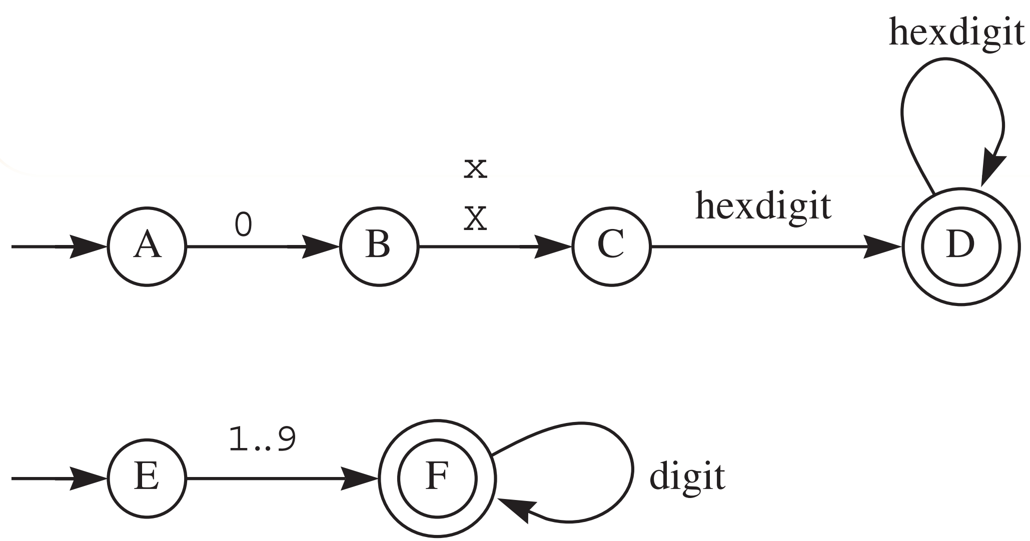
(b) The equivalent FSM without an empty transition.

Multiple token recognizers

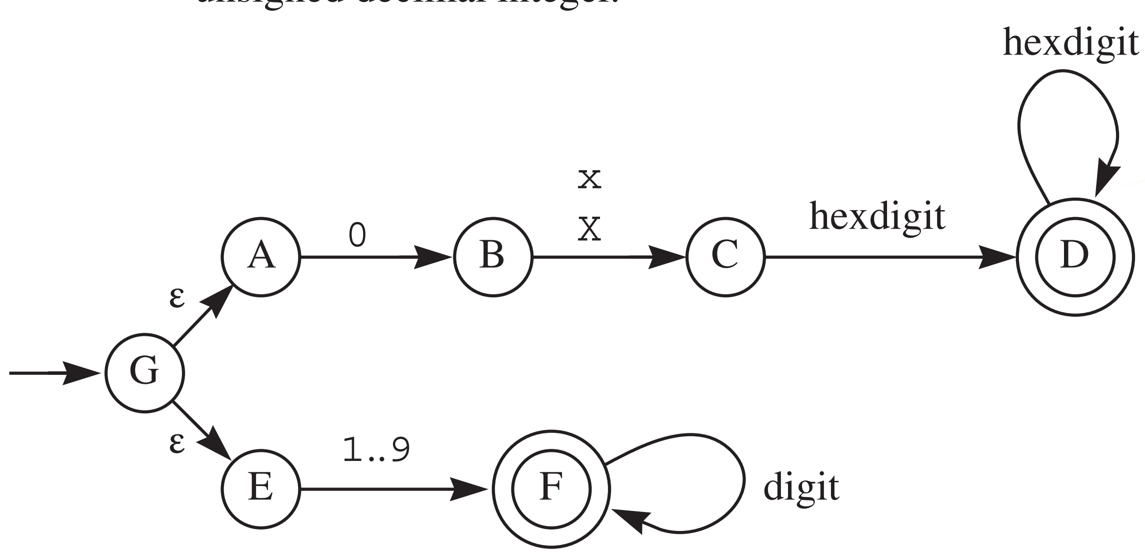
- Token
  - A string of terminal characters that has meaning as a group
- FSM with multiple final states
- The final state determines the token that is recognized



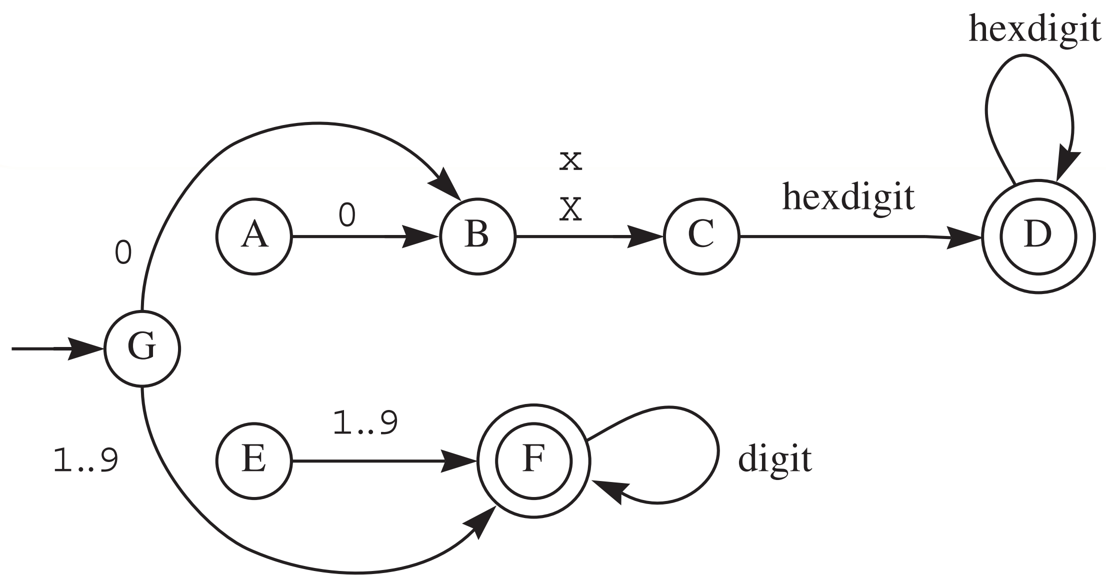
(a) Separate machines for a hexadecimal constant and an unsigned decimal integer.



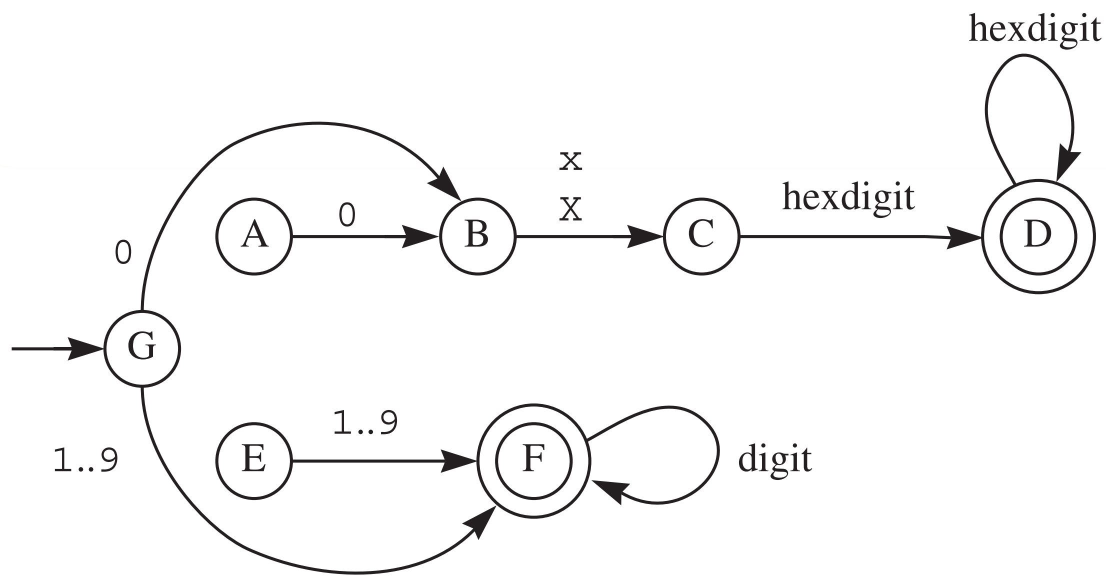
(a) Separate machines for a hexadecimal constant and an unsigned decimal integer.



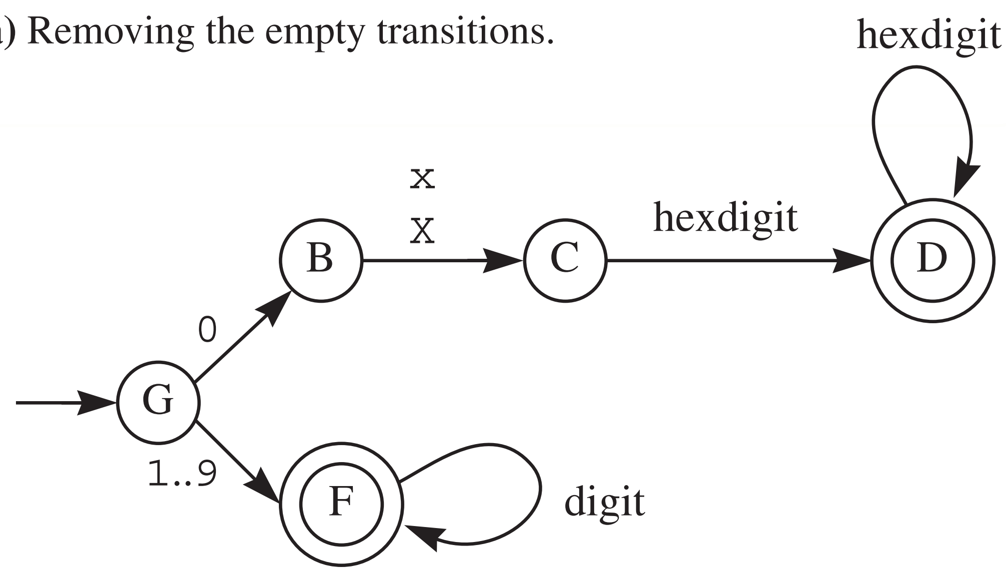
(b) One nondeterministic FSM that recognizes a hexadecimal constant or an unsigned integer token.



(a) Removing the empty transitions.

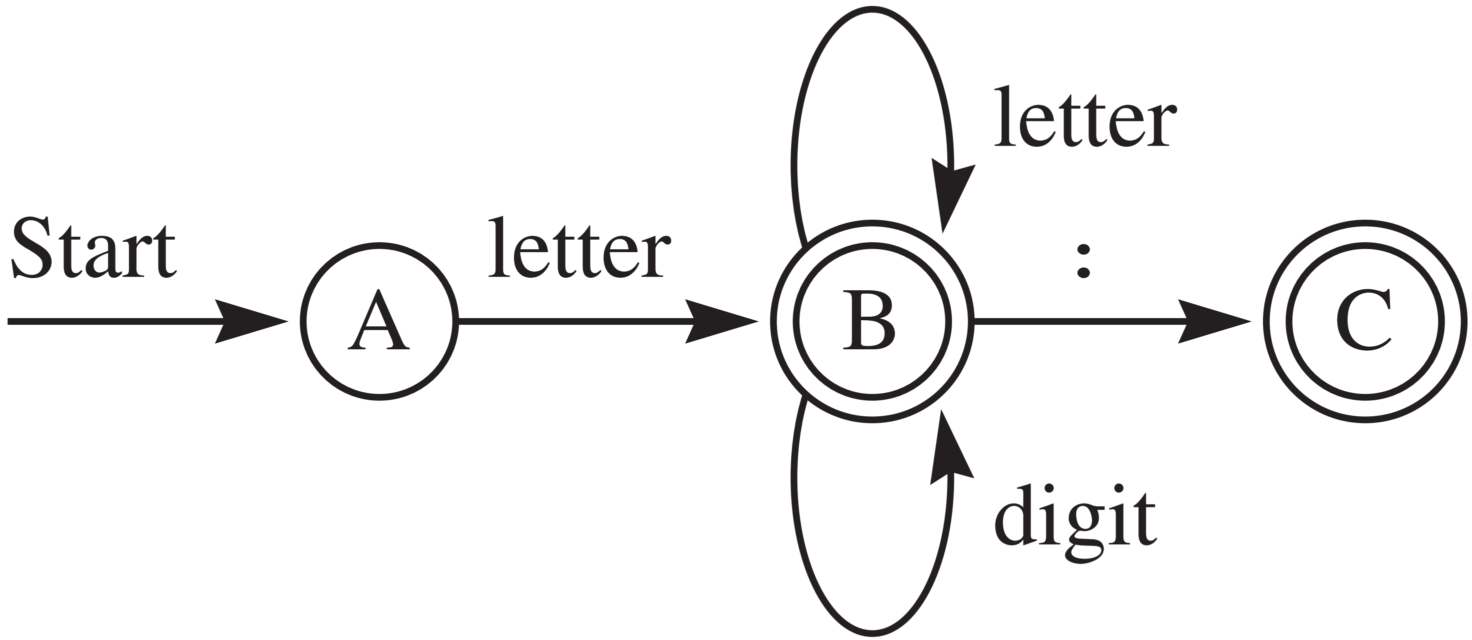


(a) Removing the empty transitions.



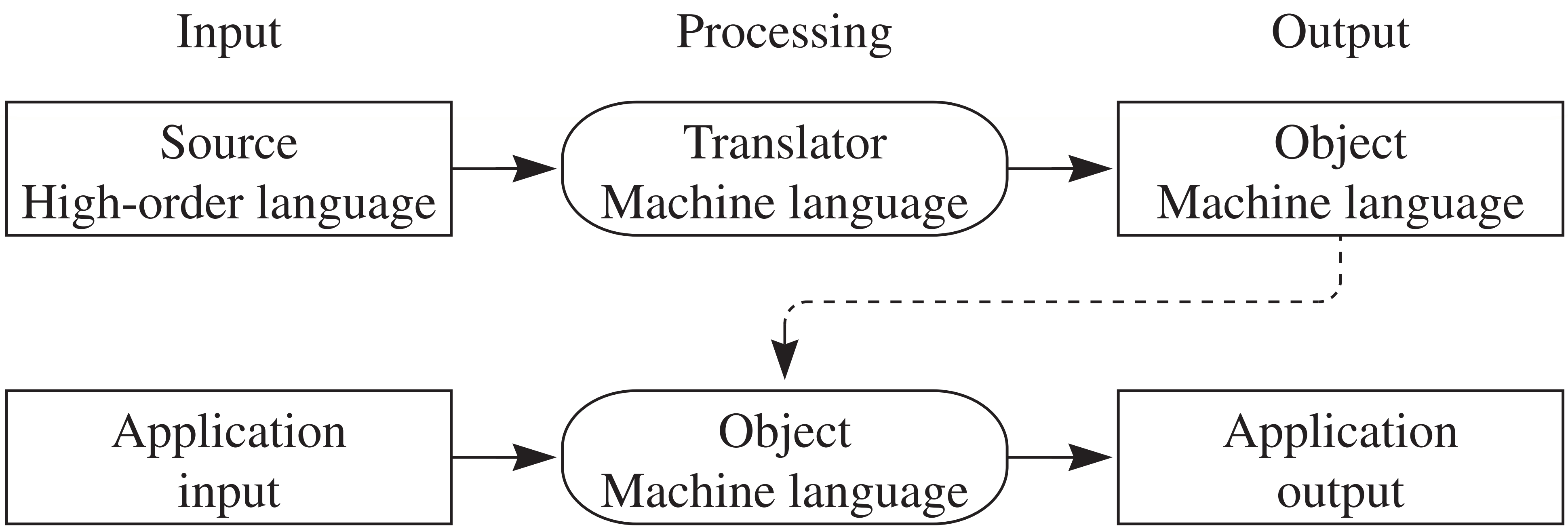
(b) Removing the inaccessible states.

Figure 7.24



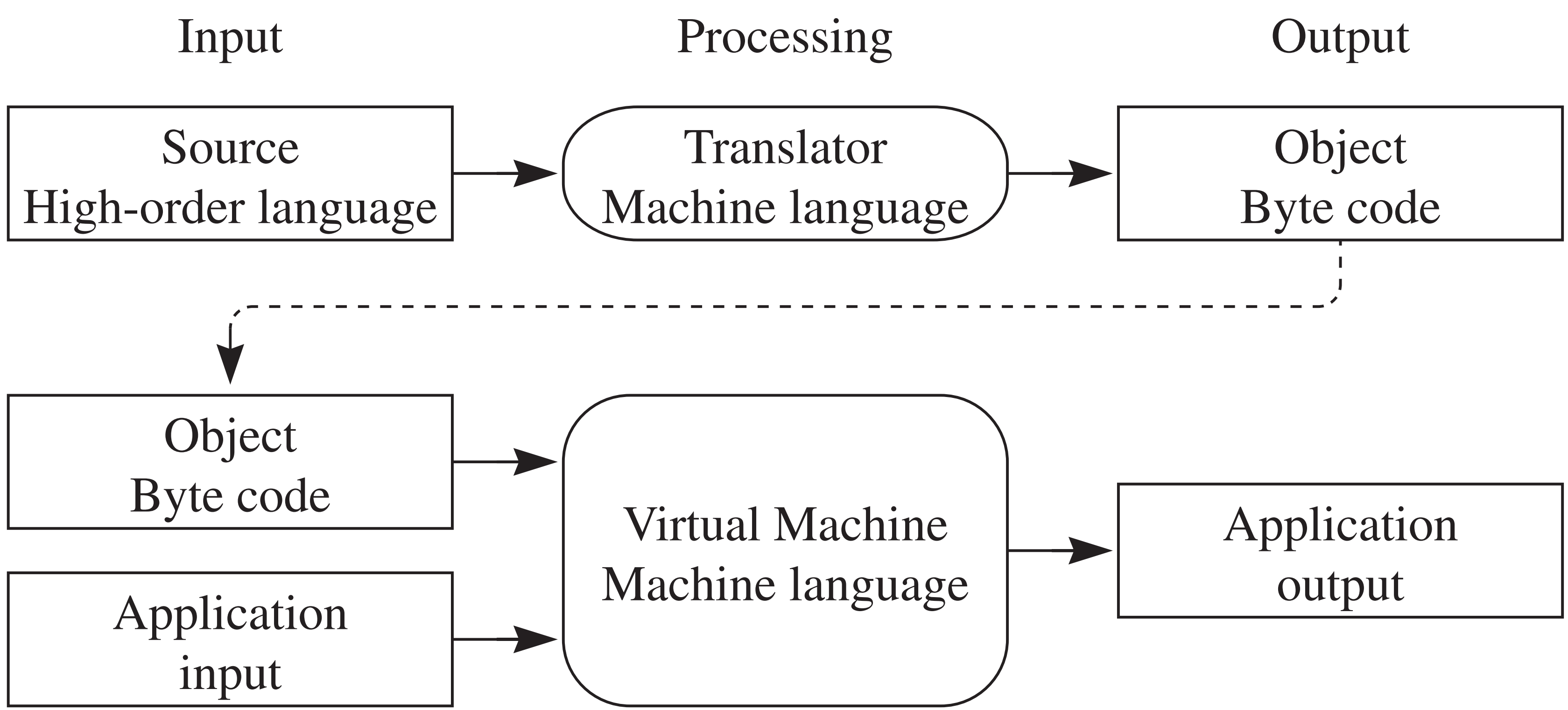

Grammars
More powerful
Finite-state machines
Regular expressions
Less powerful

Compilation



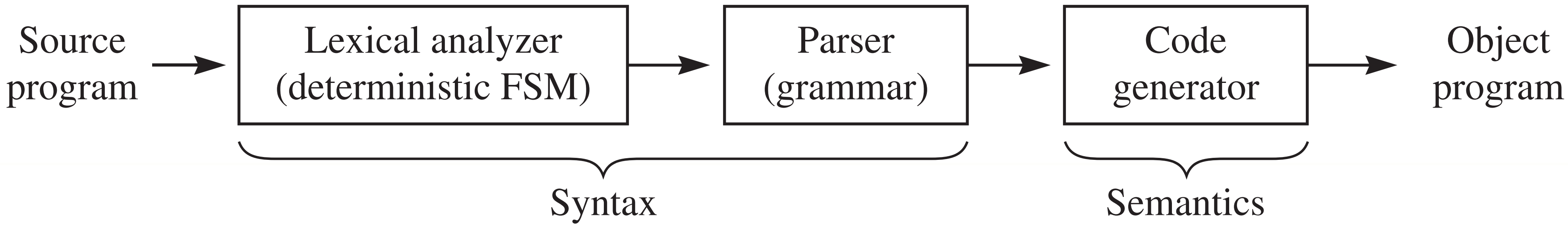
(a) Compilation.

Interpretation



(b) Interpretation.

Stages of translation



Stages of translation

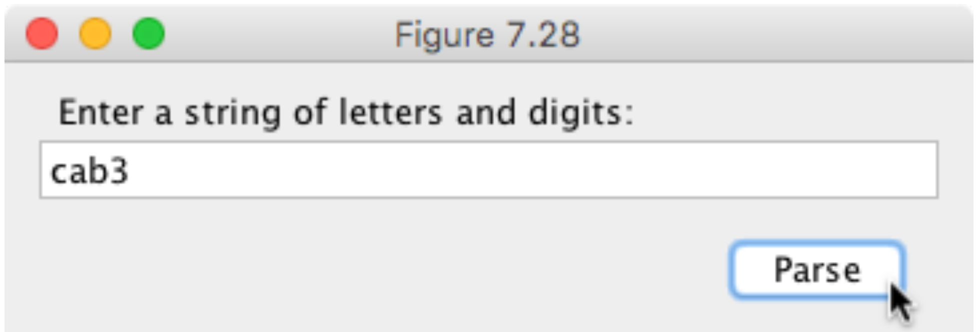
- Input of lexical analyzer – string of terminal characters
- Output of lexical analyzer and input of parser – stream of tokens
- Output of parser and input of code generator – syntax tree and/or program in low-level language
- Output of code generator – object program

FSM implementation techniques

- Table-lookup
- Direct-code

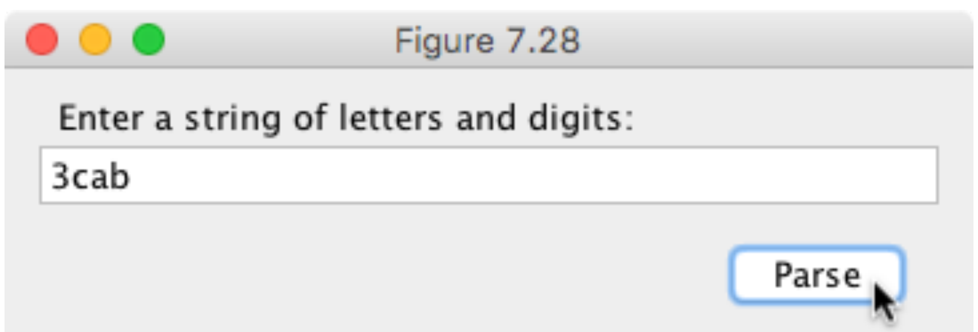
A table-lookup implementation

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



Console output

`cab3 is a valid identifier.`



Console output

`3cab is not a valid identifier.`

```
package fig0728;

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

/**
 * Figure 7.28 of J Stanley Warford, <i>Computer Systems</i>, Fifth edition,
 * Jones &amp; Bartlett, 2017.
 *
 * <p>
 * Implementation of the FSM of Figure 7.11 with the table-lookup technique.
 *
 * <p>
 * File: <code>Fig0728Main.java</code>
 *
 * @see <a href="http://computersystemsbook.com"><i>Computer Systems</i></a>
 * book home page,
 * <a href="http://www.cslab.pepperdine.edu/warford/cosc330/">course</a>
 * home page.
 * @author J. Stanley Warford
 */
```

Javadoc

PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES

SEARCH: Search

Package fig0728

Class Summary

Class	Description
Fig0728Main	Figure 7.28 of J Stanley Warford, *Computer Systems*, Fifth edition, Jones & Bartlett, 2017.

PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES

PACKAGE CLASS TREE DEPRECATED INDEX HELP

ALL CLASSES

SEARCH: Search

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Package fig0728

Class Fig0728Main

java.lang.Object
  fig0728.Fig0728Main

All Implemented Interfaces:

`java.awt.event.ActionListener, java.util.EventListener`

```
public class Fig0728Main
extends java.lang.Object
implements java.awt.event.ActionListener
```

Figure 7.28 of J Stanley Warford, *Computer Systems*, Fifth edition, Jones & Bartlett, 2017.

Implementation of the FSM of Figure 7.11 with the table-lookup technique.

File: `Fig0728Main.java`

See Also:

Computer Systems book home page, course home page.

```
public class Fig0728Main implements ActionListener {

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

```
public Fig0728Main() {
   // Set up the main window.
   mainWindowFrame = new JFrame("Figure 7.28");
   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);
}
```

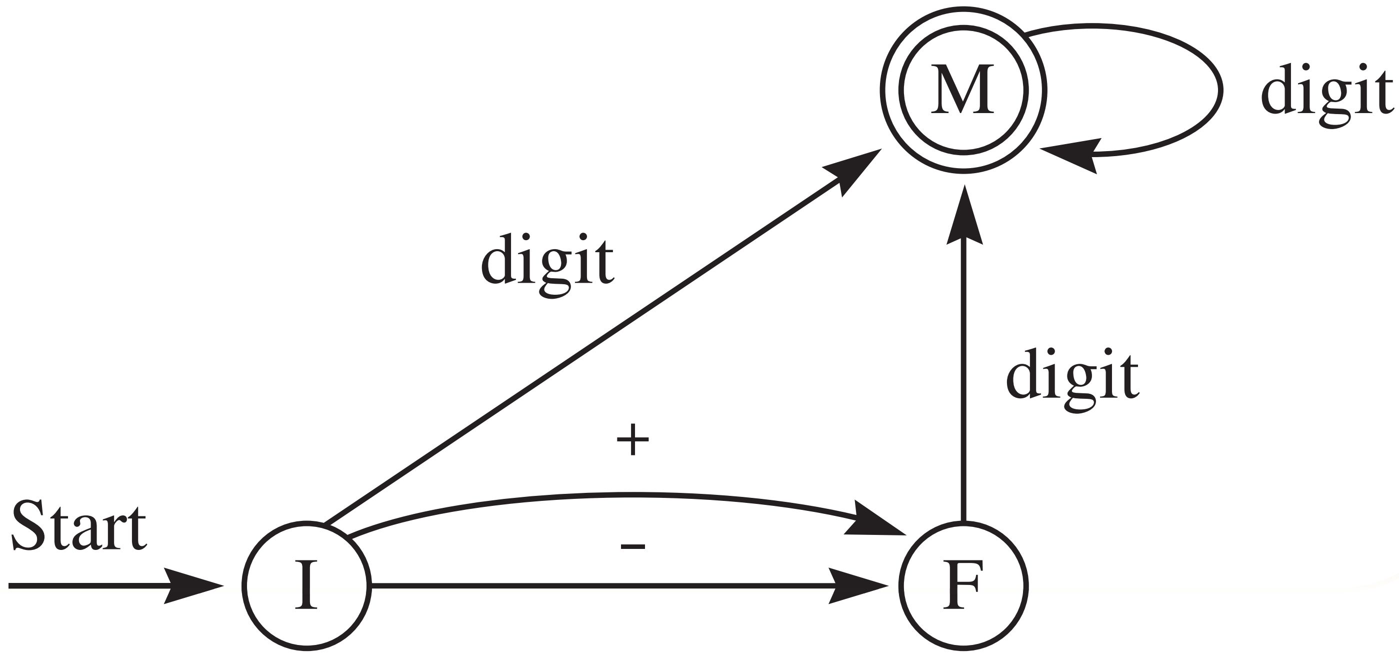
```
private static void createAndShowGUI() {
   JFrame.setDefaultLookAndFeelDecorated(true);
   new Fig0728Main();
}

public static void main(String[] args) {
   javax.swing.SwingUtilities.invokeLater(Fig0728Main::createAndShowGUI);
}
```

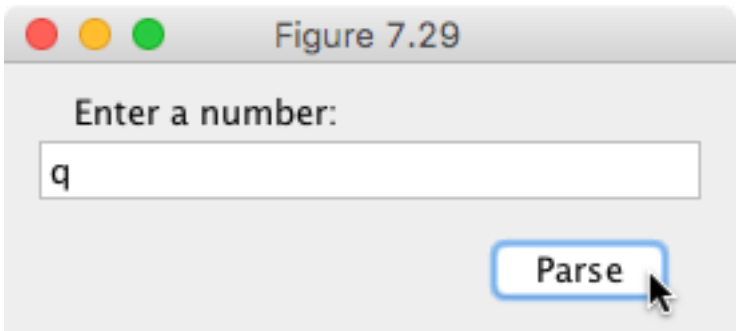
Figure 7.28 (continued)

```
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);
        }
    }
}
```

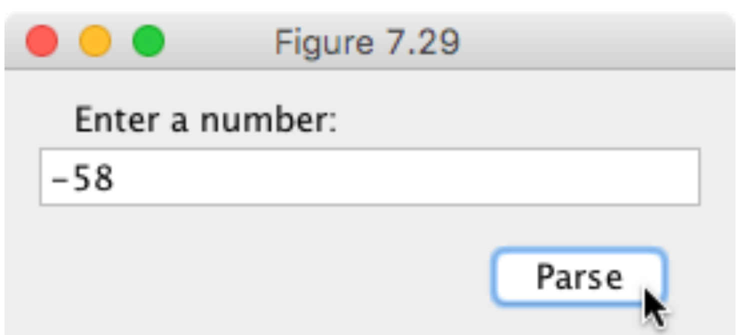
A direct-code implementation


M
digit
digit
digit
+
Start
–
I
F



Console output

```
Invalid entry.
```



Console output

```
Number = -58
```

```
public class Fig0729Main implements ActionListener {

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

...

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

```
package fig0729;

enum State {
   S_I, S_F, S_M, S_STOP
}
```

Figure 7.29 (continued)

```
package fig0729;

public class Parser {

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

    public boolean getValid() {
        return valid;
    }

    public int getNumber() {
        return number;
    }

    private 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

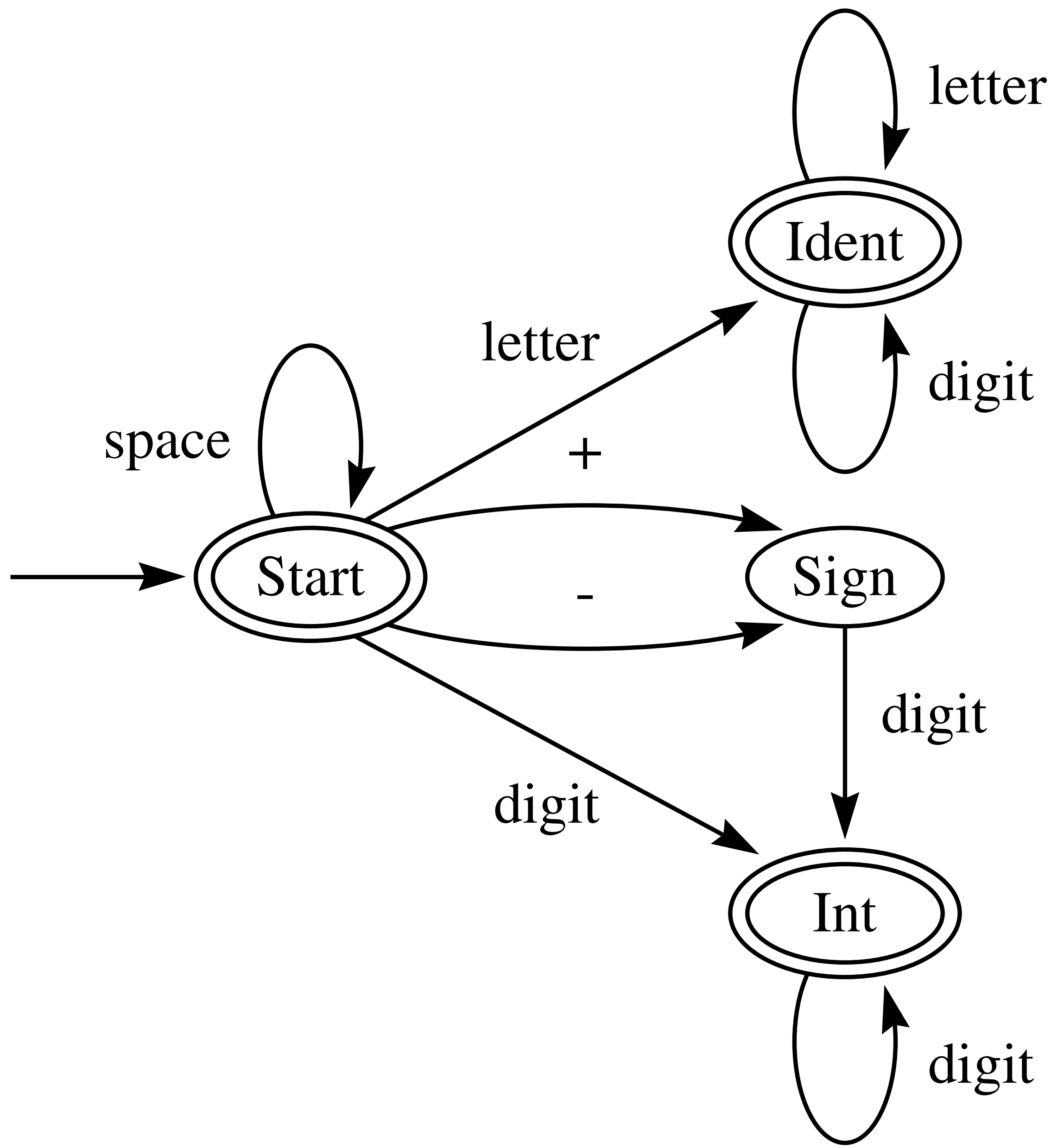
- Used to process one character at a time from a Java String as if from an input stream
- Provides a special feature needed by multiple-token parsers
- Ability to back up a character into the input stream after being scanned

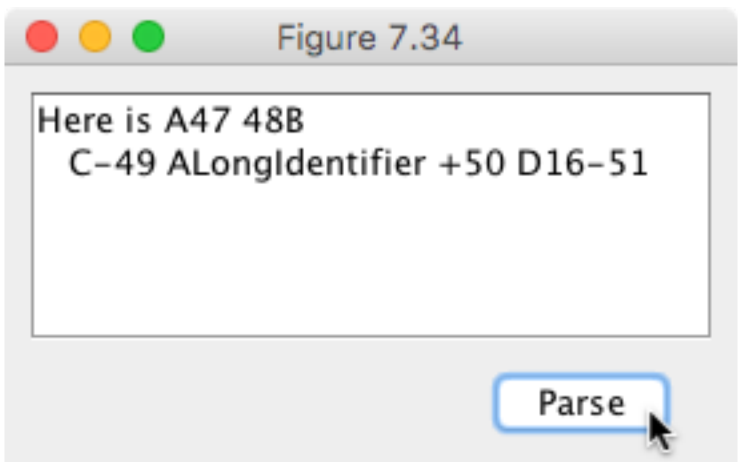
```
public class InBuffer {

    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--;
    }
}
```

A multiple-token parser

Figure 7.31

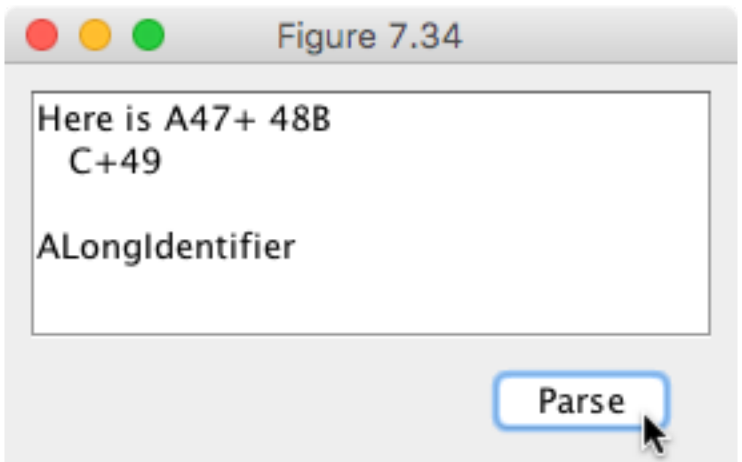

letter
Ident
letter
digit
space
+
Start
Sign
-
digit
digit
Int
digit



Console output

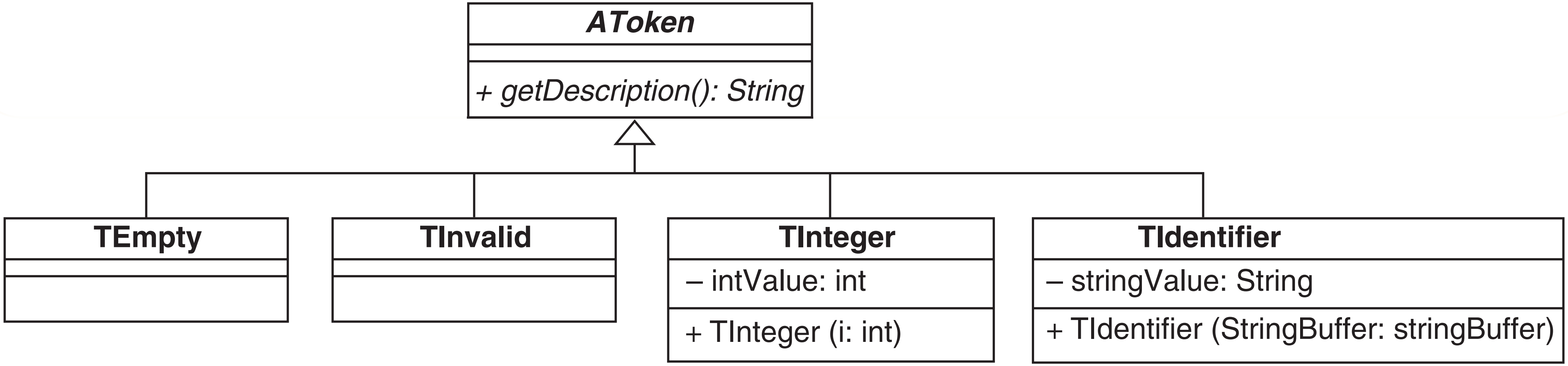
```
Identifier = Here
Identifier = is
Identifier = A47
Integer    = 48
Identifier = B
Empty token
Identifier = C
Integer    = -49
Identifier = ALongIdentifier
Integer    = 50
Identifier = D16
Integer    = -51
Empty token
```

Figure 7.32 (continued)



Console output

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


AToken
+ getDescription(): String
TEmpty
TInvalid
TInteger
– intValue: int
+ TInteger (i: int)
TIdentifier
– stringValue: String
+ TIdentifier (StringBuffer: stringBuffer)

```
abstract public class AToken {
    public abstract String getDescription();
}

public class TEmpty extends AToken {

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

public class TInvalid extends AToken {

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

```
public class TInteger extends AToken {
    private final int intValue;

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

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

public class TIdentifier extends AToken {
    private final String stringValue;

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

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

```
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'));
    }
}

public enum LexState {
    LS_START, LS_IDENT, LS_SIGN, LS_INTEGER, LS_STOP
}
```

```
public class Tokenizer {

    private final 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;
```

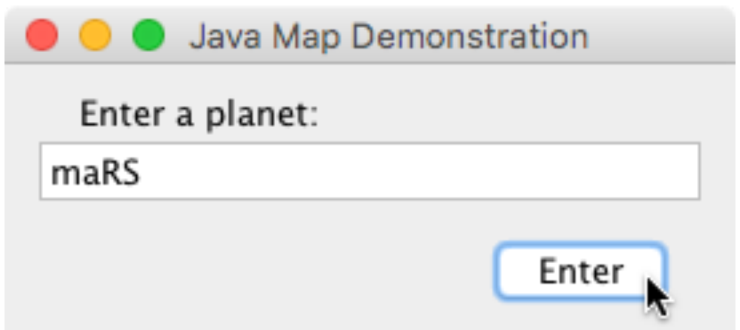
```
do {
    nextChar = b.advanceInput();
    switch (state) {
        case LS_START:
            if (Util.isAlpha(nextChar)) {
                localStringValue.append(nextChar);
                state = LexState.LS_IDENT;
            } else if (nextChar == '-') {
                sign = -1;
                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;
            } 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)) {
        localIntValue = 10 * localIntValue + nextChar - '0';
        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(sign * localIntValue);
                        state = LexState.LS_STOP;
                    }
                    break;
            }
        } while ((state != LexState.LS_STOP) && !(aToken instanceof TInvalid));
        return aToken;
    }
}
```

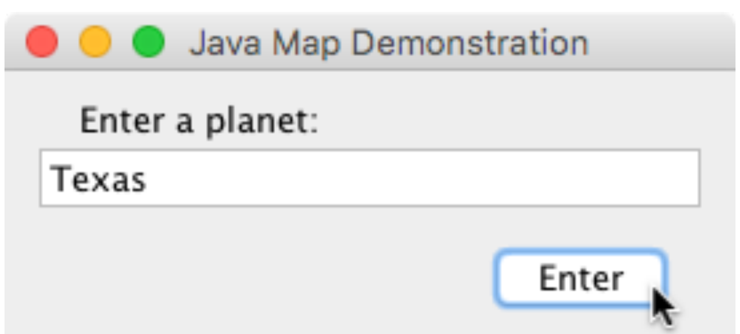
```
    public void actionPerformed(ActionEvent event) {
        InBuffer inBuffer = new InBuffer(textArea.getText());
        Tokenizer t = new Tokenizer(inBuffer);
        AToken aToken;
        inBuffer.getLine();
        while (inBuffer.inputRemains()) {
            do {
                aToken =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

Java map demo



Console output

```
Planet Mars is red.
Enumerated output: P_MARS
Ordinal output: 3
```



Console output

```
Texas is not a planet.
```

```
public enum Planet {
    P_MERCURY, P_VENUS, P_EARTH, P_MARS, P_JUPITER, P_SATURN,
    P_URANUS, P_NEPTUNE, P_PLUTO
}

public class Maps {

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

    static {
        planetTable = new HashMap<>();
        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.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");
    }
}
```

```
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.");
        }
    }
}
```

A language translator

Input

```
set (Time, 15)
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
Accel <- 3
TSquared <- Time
TSquared <- TSquared * Time
Position <- TSquared
Position <- Position * Accel
Position <- Position / 2
stop

Program listing:
set (Time, 15)
set (Accel, 3)
set (TSquared, Time)
mul (TSquared, Time)
set (Position, TSquared)
mul (Position, Accel)
div (Position, 2)
stop
end
```

Input

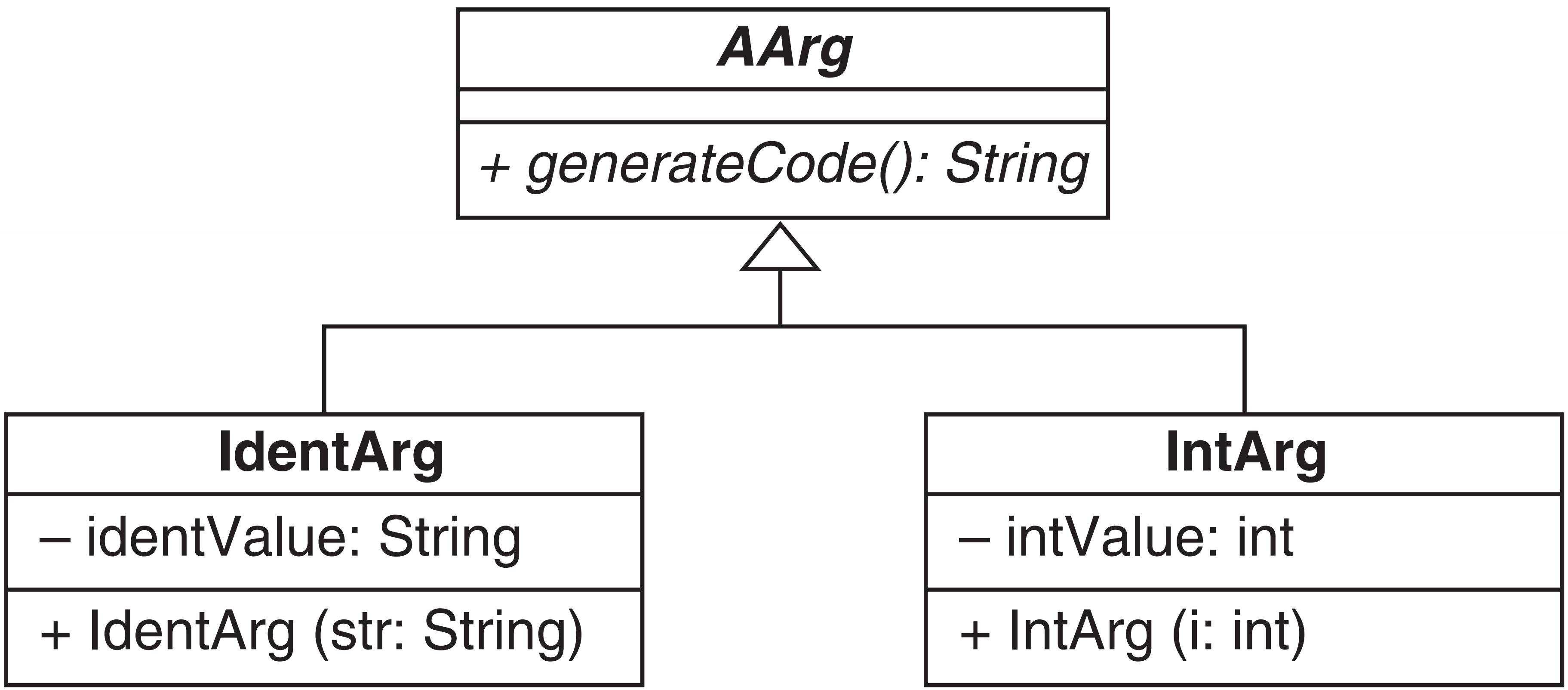
```
set (Alpha,, 123)
set (Alpha)
sit (Alpha, 123)
set, (Alpha)
mul (Alpha, Beta
set (123, Alpha)
neg (Alpha, Beta)
set (Alpha, 123) x
```

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.
```

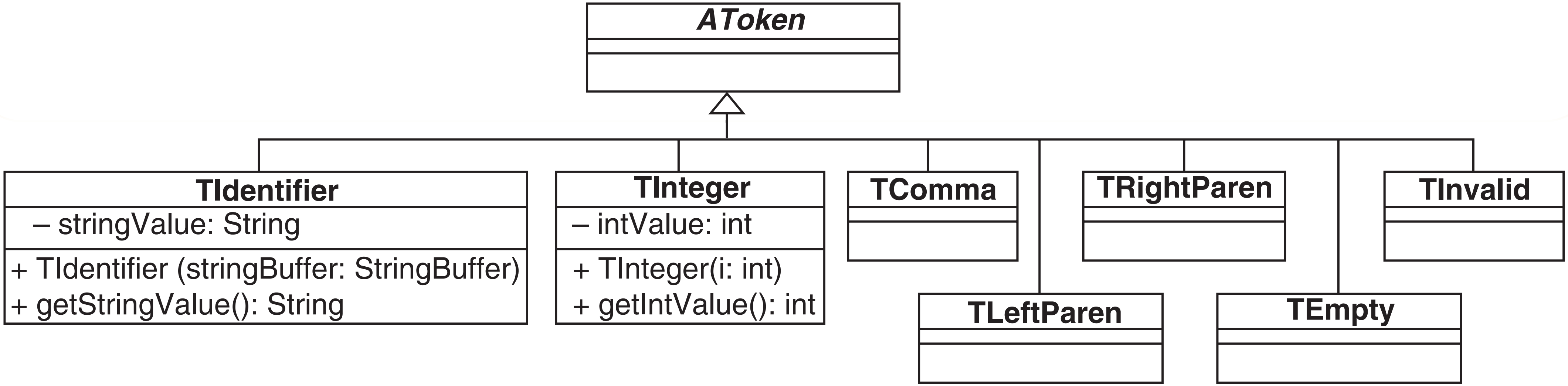
```
public enum Mnemon {
   M_ADD, M_SUB, M_MUL, M_DIV, M_NEG, M_ABS, M_SET, M_STOP, M_END
}

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 {
      unaryMnemonTable = new HashMap<>();
      unaryMnemonTable.put("stop", Mnemon.M_STOP);
      unaryMnemonTable.put("end", Mnemon.M_END);

      nonUnaryMnemonTable = new HashMap<>();
      nonUnaryMnemonTable.put("neg", Mnemon.M_NEG);
      nonUnaryMnemonTable.put("abs", Mnemon.M_ABS);
      nonUnaryMnemonTable.put("add", Mnemon.M_ADD);
      nonUnaryMnemonTable.put("sub", Mnemon.M_SUB);
      nonUnaryMnemonTable.put("mul", Mnemon.M_MUL);
      nonUnaryMnemonTable.put("div", Mnemon.M_DIV);
      nonUnaryMnemonTable.put("set", Mnemon.M_SET);
```

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

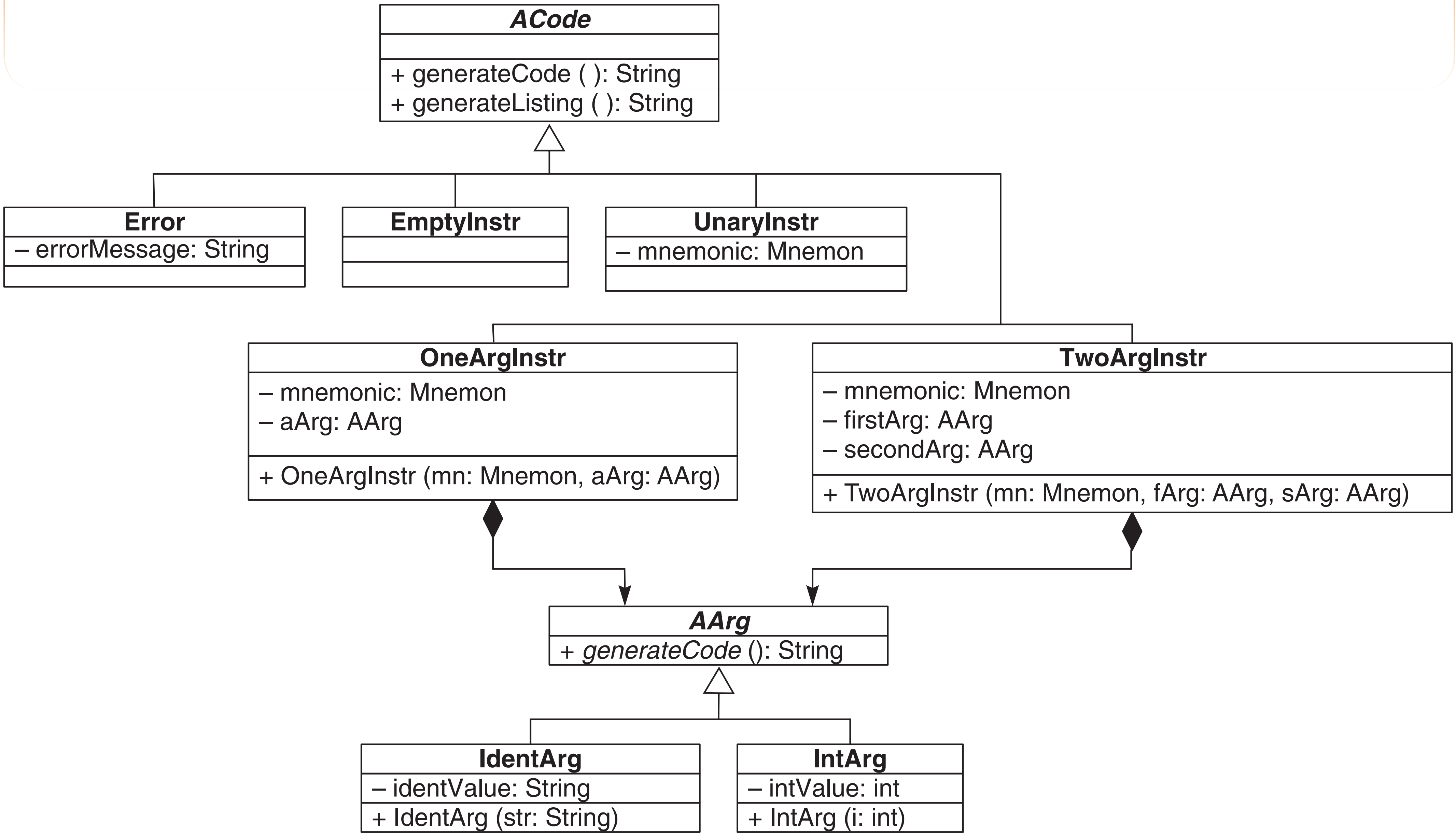

AArg
+ generateCode(): String
IdentArg
– identValue: String
+ IdentArg (str: String)
IntArg
– intValue: int
+ IntArg (i: int)

```
abstract public class AArg {
    abstract public String generateCode();
}

public class IdentArg extends AArg {
    private final String identValue;
    public IdentArg(String str) {
        identValue = str;
    }
    @Override
    public String generateCode() {
        return identValue;
    }
}

public class IntArg extends AArg {
    private final int intValue;
    public IntArg(int i) {
        intValue = i;
    }
    @Override
    public String generateCode() {
        return String.format("%d", intValue);
    }
}
```

Figure 7.41


AToken
TIdentifier
– stringValue: String
+ TIdentifier (stringBuffer: StringBuffer)
+ getStringValue(): String
TInteger
– intValue: int
+ TInteger(i: int)
+ getIntValue(): int
TComma
TRightParen
TInvalid
TLeftParen
TEmpty

```
abstract public class AToken {
}

public class TIdentifier extends AToken {
   private final String stringValue;
   public TIdentifier(StringBuffer stringBuffer) {
      stringValue = new String(stringBuffer);
   }
   public String getStringValue() {
      return stringValue;
   }
}

public class TInteger extends AToken {
   private final int intValue;
   public TInteger(int i) {
      intValue = i;
   }
   public int getIntValue() {
      return intValue;
   }
}

public class TComma extends AToken {
}
```


ACode
+ generateCode ( ): String
+ generateListing ( ): String
Error
– errorMessage: String
EmptyInstr
UnaryInstr
– mnemonic: Mnemon
OneArgInstr
– mnemonic: Mnemon
– aArg: AArg
+ OneArgInstr (mn: Mnemon, aArg: AArg)
TwoArgInstr
– mnemonic: Mnemon
– firstArg: AArg
– secondArg: AArg
+ TwoArgInstr (mn: Mnemon, fArg: AArg, sArg: AArg)
AArg
+ generateCode (): String
IdentArg
– identValue: String
+ IdentArg (str: String)
IntArg
– intValue: int
+ IntArg (i: int)

```
abstract public class ACode {
    abstract public String generateCode();
    abstract public String generateListing();
}

public class Error extends ACode {
    private final String errorMessage;
    public Error(String errMessage) {
        errorMessage = errMessage;
    }

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

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

Figure 7.44 (continued)

```
public class EmptyInstr extends ACode {
    // For an empty source line.

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

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

```
public class UnaryInstr extends ACode {
    private final Mnemon mnemonic;
    public UnaryInstr(Mnemon mn) {
        mnemonic = mn;
    }

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

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

```
public class OneArgInstr extends ACode {
   private final Mnemon mnemonic;
   private final AArg aArg;
   public OneArgInstr(Mnemon mn, AArg aArg) {
      mnemonic = mn;
      this.aArg = aArg;
   }
    @Override
   public String generateListing() {
      return String.format("%s (%s)\n",
                    Maps.mnemonStringTable.get(mnemonic), aArg.generateCode());
   }
    @Override
   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.
      }
   }
}
```

```
public class TwoArgInstr extends ACode {
    private final Mnemon mnemonic;
    private final AArg firstArg;
    private final AArg secondArg;
    public TwoArgInstr(Mnemon mn, AArg fArg, AArg sArg) {
        mnemonic = mn;
        firstArg = fArg;
        secondArg = sArg;
    }

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

```
@Override
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.
        }
    }
}
```

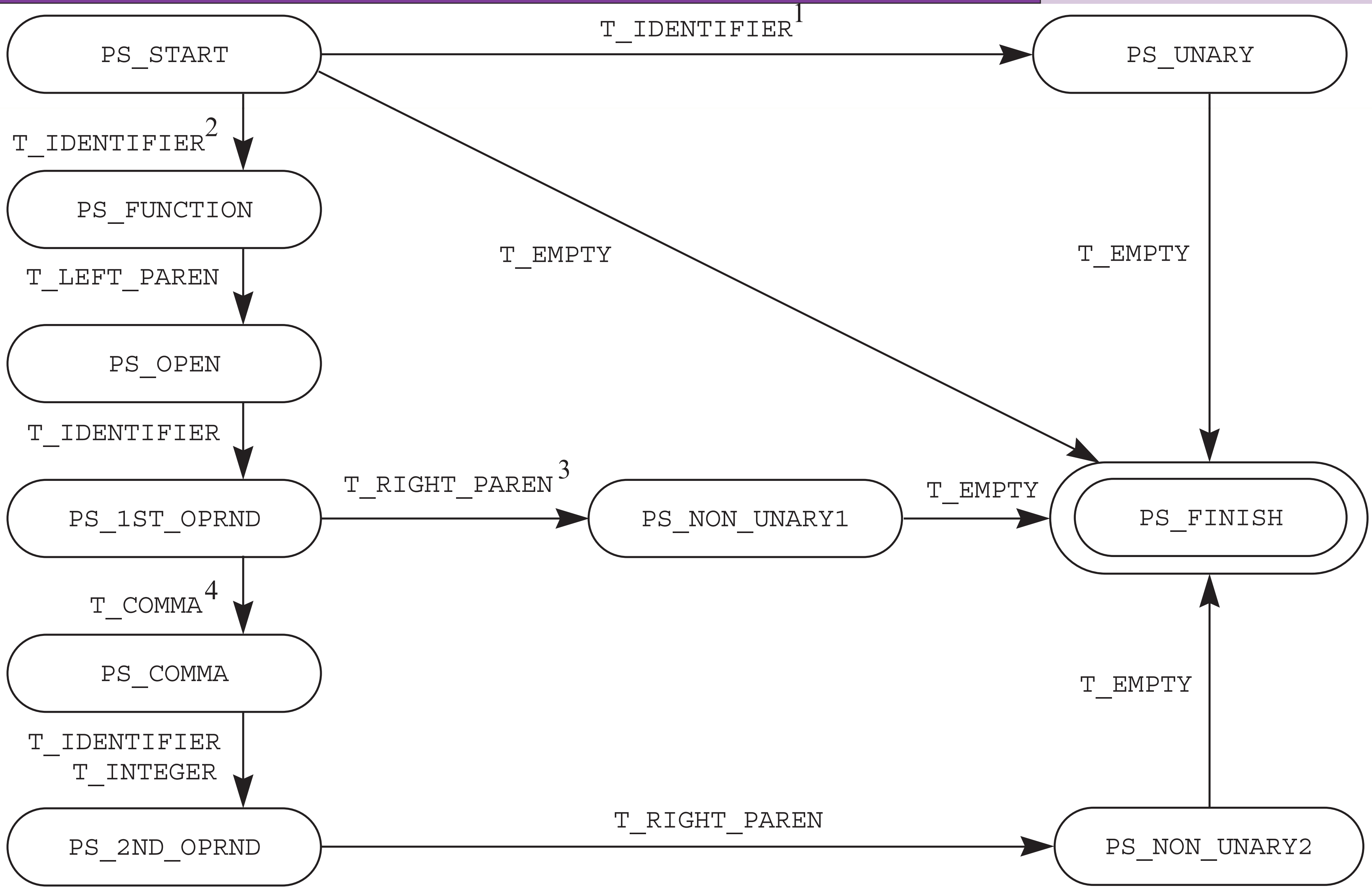
```
public enum LexState {
    LS_START, LS_IDENT, LS_SIGN, LS_INTEGER, LS_STOP
}

public class Tokenizer {

   private final 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;
         } 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;
```

Figure 7.45 (continued)

```
    } 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, and M_MUL, M_DIV.

```
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
}

public class Translator {
   private final 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;
      // Compiler requires following useless initialization.
      Mnemon localMnemon = Mnemon.M_END;
      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_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_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<>();
   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());
        }
    }
}
```

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

Translation phases

- Lexical analyzer – `getToken()`
- Parser– `parseLine()`
- Code generator– `generateCode()`

$N = \{ A, B \}$

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

$P =$ the productions

1. $A \rightarrow \texttt{0}\ B$
2. $B \rightarrow \texttt{1}\ \texttt{0}\ B$
3. $B \rightarrow \varepsilon$

$S = A$

$N = \{ C \}$

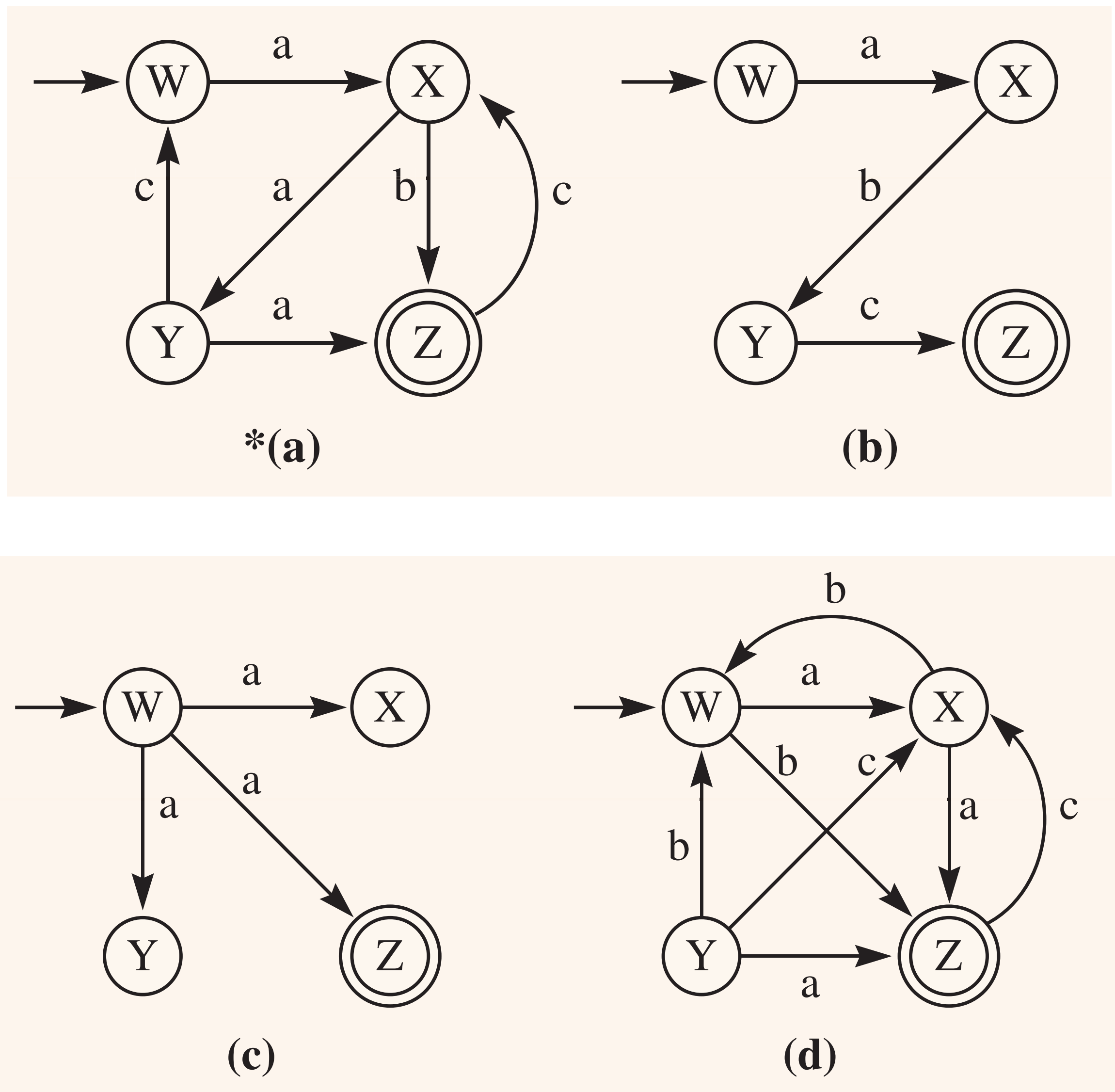
$T = \{ \texttt{0}, \texttt{1} \}$

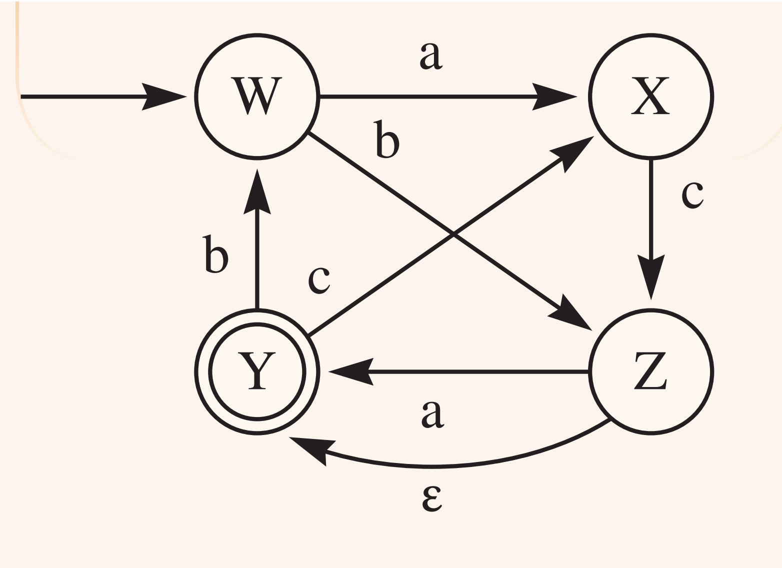
$P =$ the productions

1. $C \rightarrow C\ \texttt{10}$
2. $C \rightarrow \texttt{0}$

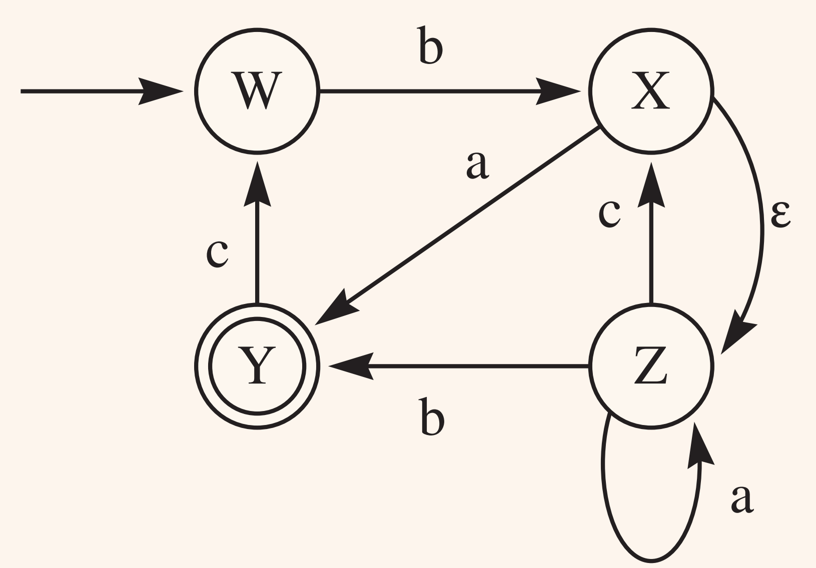
$S = C$

Figure 7.51

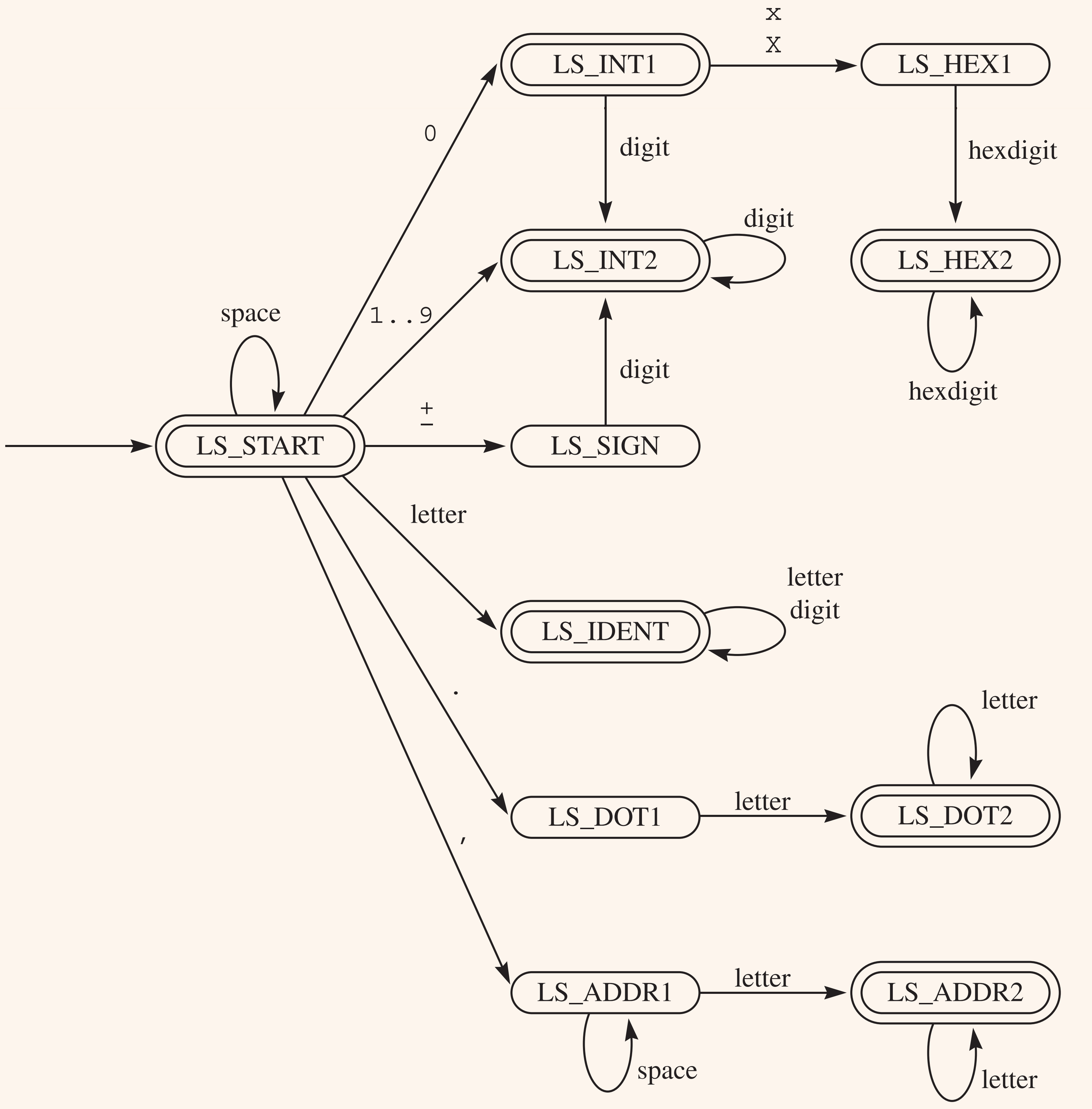




(a)



(b)



LS_START
space
0
1..9
+
−
letter
.
,
LS_INT1
x
X
LS_HEX1
digit
hexdigit
LS_INT2
digit
LS_HEX2
hexdigit
LS_SIGN
digit
LS_IDENT
letter
digit
LS_DOT1
letter
LS_DOT2
letter
LS_ADDR1
letter
LS_ADDR2
space
letter

Figure 7.53