Computer Systems

Sixth edition

Chapter 7

Assembling to the ISA Level

- 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/10 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> ⇒ <identifier> <digit> Rule 3

⇒ <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`

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

Alternative notation for production rules

$$
\begin{array}{lcl}
\mathtt{I} & \rightarrow & \mathtt{FM} \\
\mathtt{F} & \rightarrow & \mathtt{+} \mid \mathtt{-} \mid \varepsilon \\
\mathtt{M} & \rightarrow & \mathtt{d} \mid \mathtt{dM}
\end{array}
$$

Some derivations

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

Some derivations

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

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

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

A derivation

A

A derivation

$A \Rightarrow aABC$ Rule 1

⋮

⋮

⋮

A derivation

$$
\begin{aligned}
A &\Rightarrow aABC && \text{Rule 1}\\
&\Rightarrow aaABCBC && \text{Rule 1}\\
&\;\;\vdots\\
&\;\;\vdots
\end{aligned}
$$

A derivation

$$
\begin{aligned}
A &\Rightarrow aABC && \text{Rule 1}\\
&\Rightarrow aaABCBC && \text{Rule 1}\\
&\Rightarrow aaabCBCBC && \text{Rule 2}
\end{aligned}
$$

A derivation

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

A derivation

$$
\begin{aligned}
A &\Rightarrow \mathtt{a}ABC && \text{Rule 1}\\
&\Rightarrow \mathtt{aa}ABCBC && \text{Rule 1}\\
&\Rightarrow \mathtt{aaab}CBCBC && \text{Rule 2}\\
&\Rightarrow \mathtt{aaab}BCCBC && \text{Rule 3}\\
&\Rightarrow \mathtt{aaab}BCBCC && \text{Rule 3}
\end{aligned}
$$

.

.

.

.

.

.

.

A derivation

$$
\begin{aligned}
A &\Rightarrow \texttt{a}ABC && \text{Rule 1}\\
&\Rightarrow \texttt{aa}ABCBC && \text{Rule 1}\\
&\Rightarrow \texttt{aaab}CBCBC && \text{Rule 2}\\
&\Rightarrow \texttt{aaab}BCCBC && \text{Rule 3}\\
&\Rightarrow \texttt{aaab}BCBCC && \text{Rule 3}\\
&\Rightarrow \texttt{aaab}BBCCC && \text{Rule 3}\\
&\vdots
\end{aligned}
$$

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
⋮	
⋮	
⋮	

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
⋮	
⋮	

A derivation

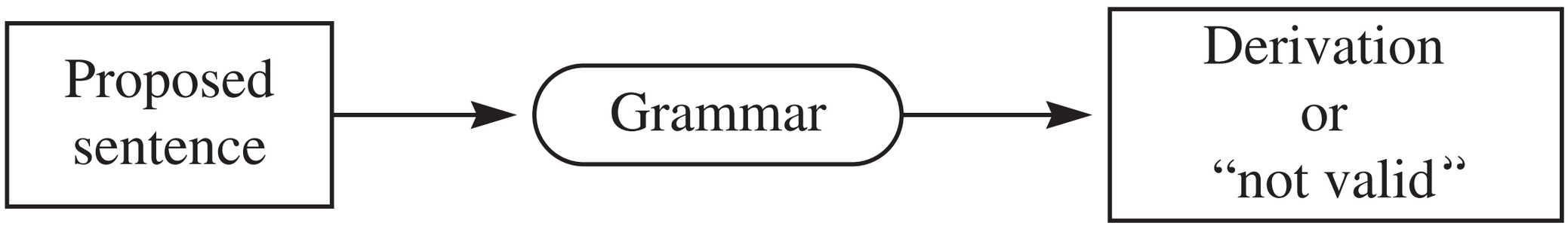
A $\Rightarrow$ aABC	Rule 1
$\Rightarrow$ aaABCBC	Rule 1
$\Rightarrow$ aaabCBCBC	Rule 2
$\Rightarrow$ aaabBCCBC	Rule 3
$\Rightarrow$ aaabBCBCC	Rule 3
$\Rightarrow$ aaabBBCCC	Rule 3
$\Rightarrow$ aaabbBCCC	Rule 4
$\Rightarrow$ aaabbbCCC	Rule 4
$\Rightarrow$ aaabbbcCC	Rule 5
$\Rightarrow$ aaabbbccC	Rule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

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
$\Rightarrow$ aaabbbccc	Rule 6

The parsing problem

(a) Deriving a valid sentence.



(b) The parsing problem.

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

Parse `( a * a ) + a`

E

Parse (a * a) + a

$E \Rightarrow E + T$ Rule 1

Parse `( a * a ) + a`

$$E \Rightarrow E + T \quad \text{Rule 1}$$
$$\Rightarrow T + T \quad \text{Rule 2}$$

Parse `( a * a ) + a`

$$
\begin{aligned}
E &\Rightarrow E + T && \text{Rule 1} \\
&\Rightarrow T + T && \text{Rule 2} \\
&\Rightarrow F + T && \text{Rule 4}
\end{aligned}
$$

Parse `( a * a ) + a`

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

Parse (a * a) + a

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

Parse `( a * a ) + a`

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

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`

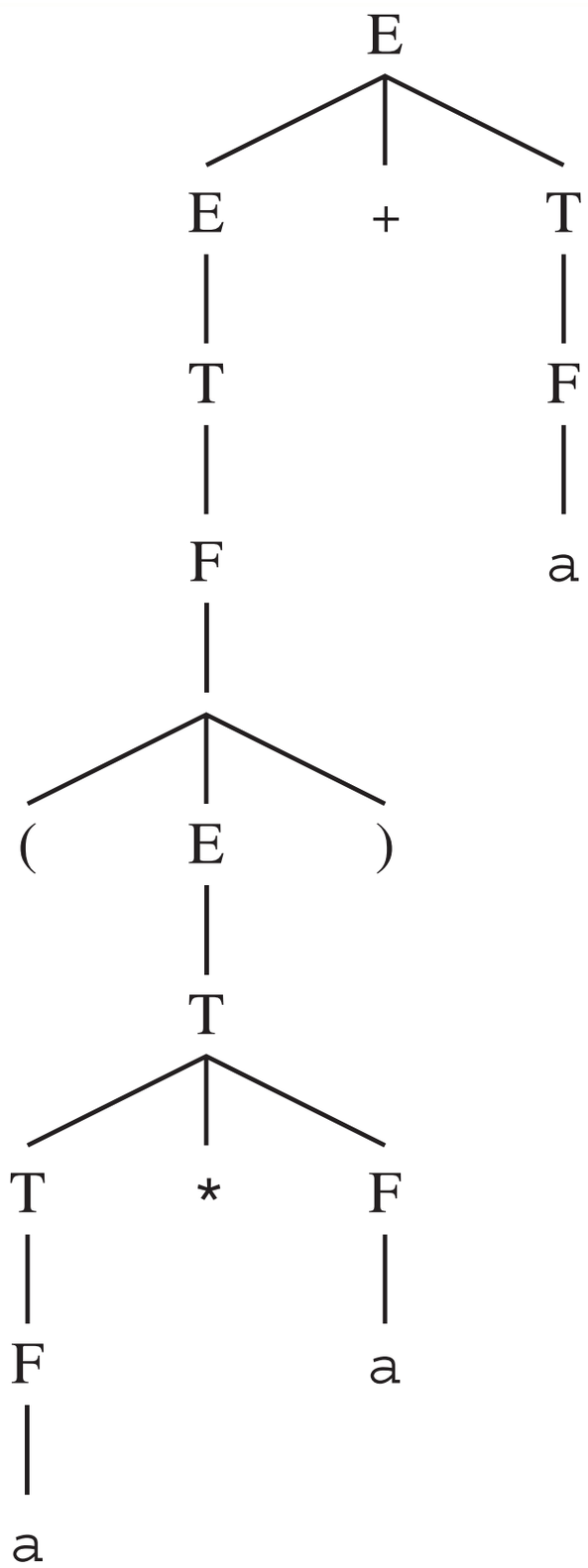
E ⇒ E + T	Rule 1
⇒ T + T	Rule 2
⇒ F + T	Rule 4
⇒ (E) + T	Rule 5
⇒ (T) + T	Rule 2
⇒ (T * F) + T	Rule 3
⇒ (F * F) + T	Rule 4
⇒ (a * F) + T	Rule 6
⇒ (a * a) + T	Rule 6

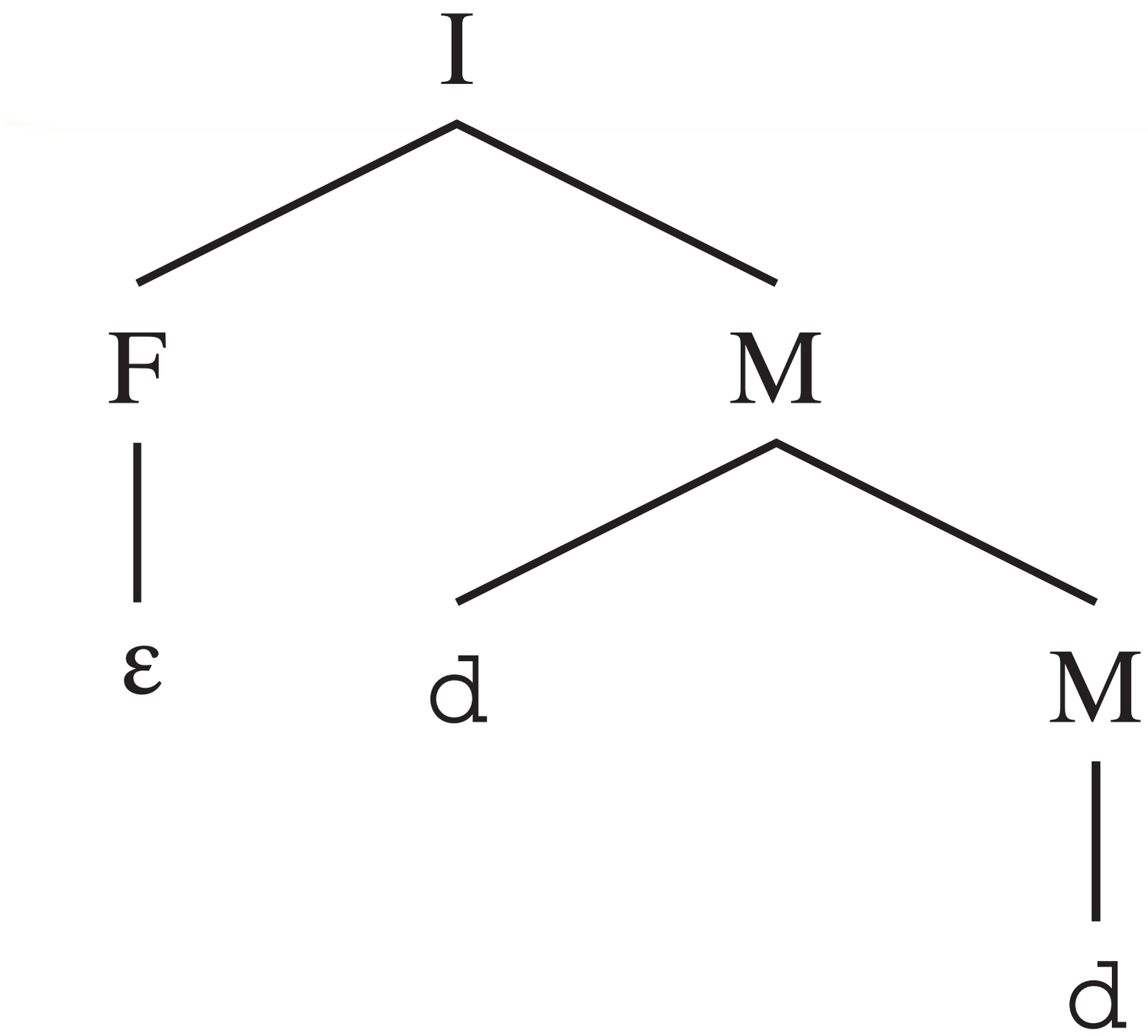
Parse `( a * a ) + a`

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 ⇒ 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
⇒ (a * a) + a	Rule 6


E
E
+
T
T
F
F
a
(
E
)
T
T
*
F
F
a


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>

```
<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

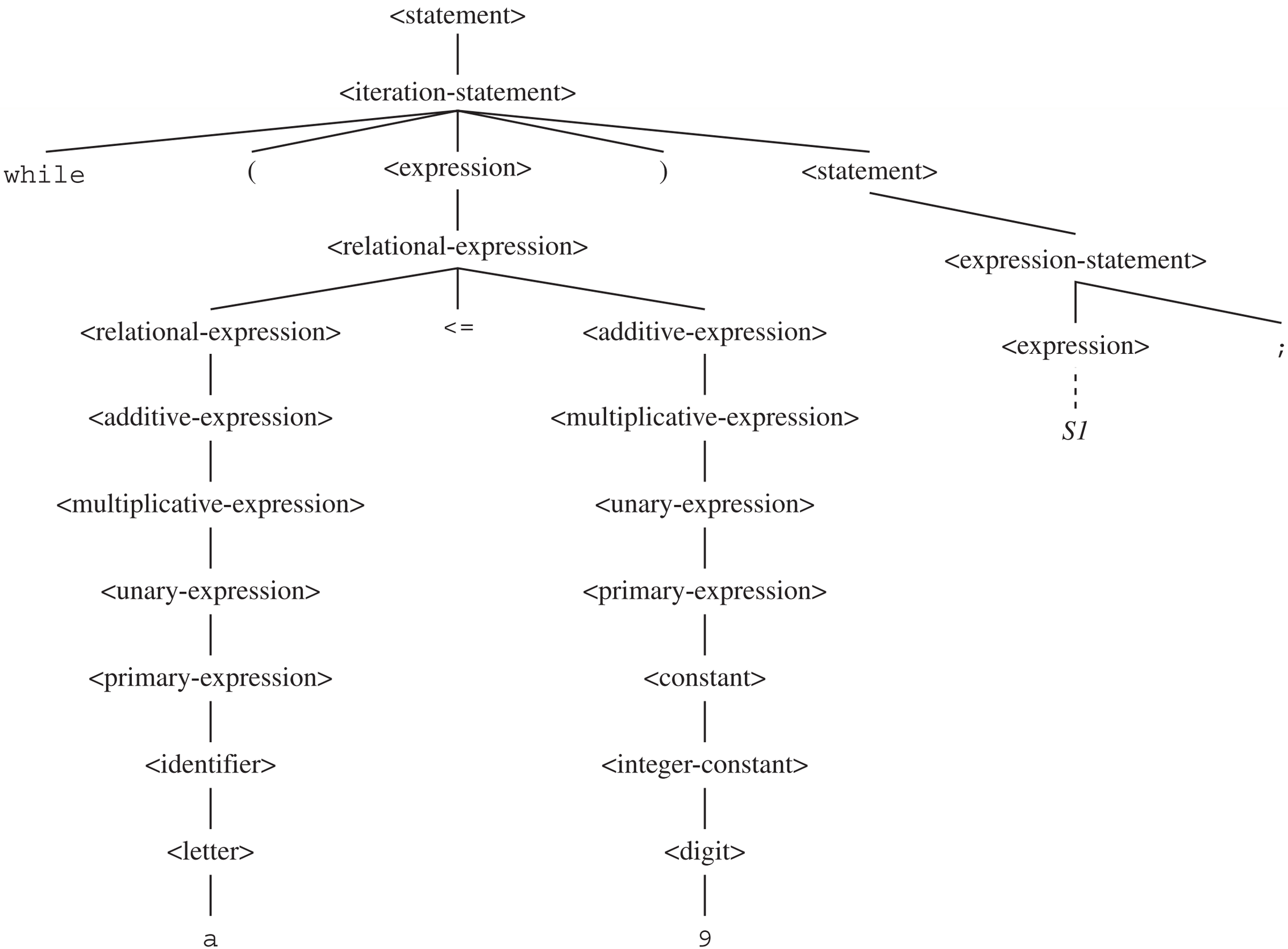
<digit> →
    0|1|2|3|4|5|6|7|8|9

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* `;`

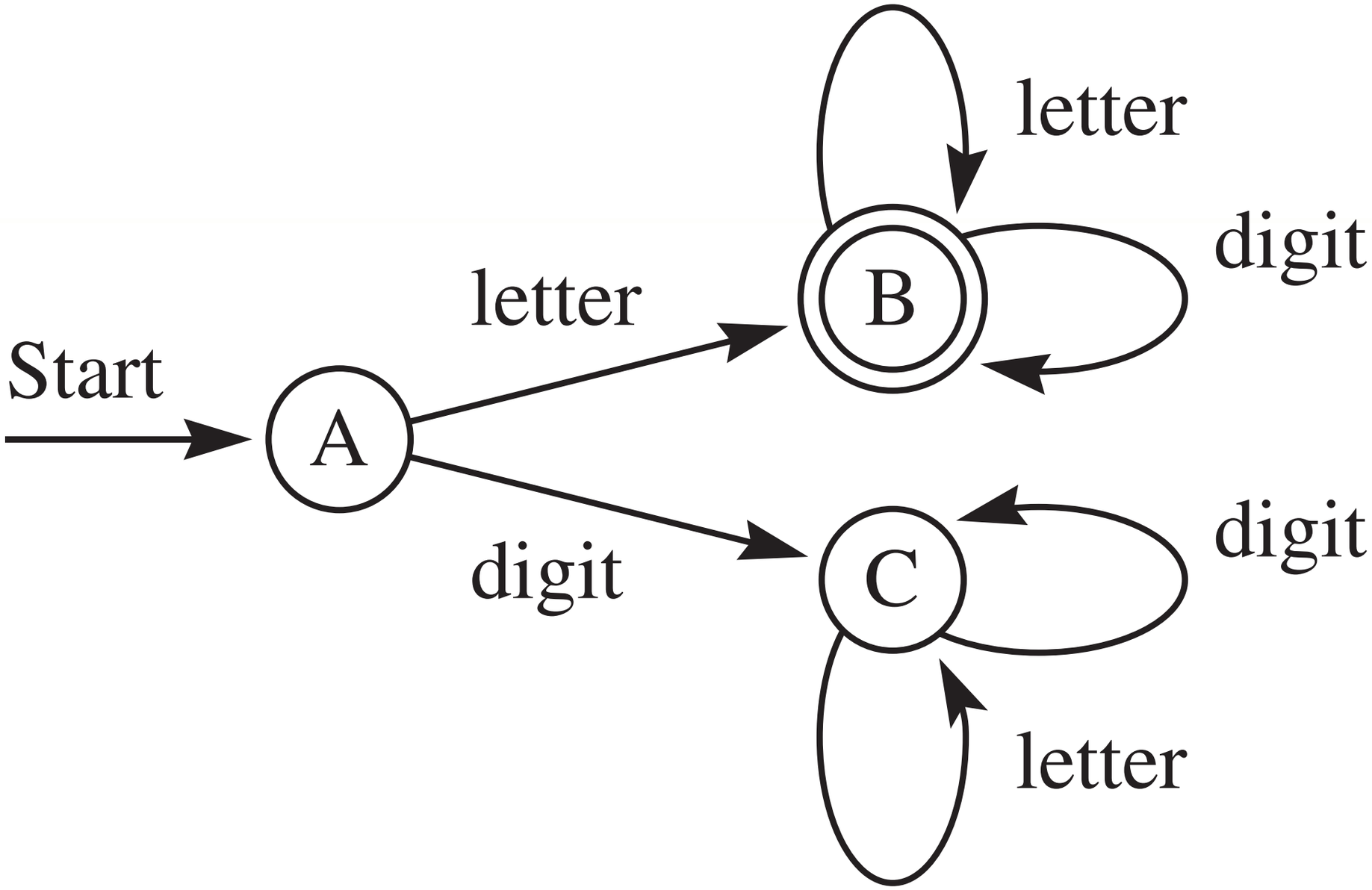

<statement>
<iteration-statement>
while
(
<expression>
)
<statement>
<relational-expression>
<expression-statement>
<relational-expression>
<=
<additive-expression>
<expression>
;
S1
<additive-expression>
<multiplicative-expression>
<multiplicative-expression>
<unary-expression>
<unary-expression>
<primary-expression>
<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


Start
A
letter
B
letter
digit
digit
C
digit
letter

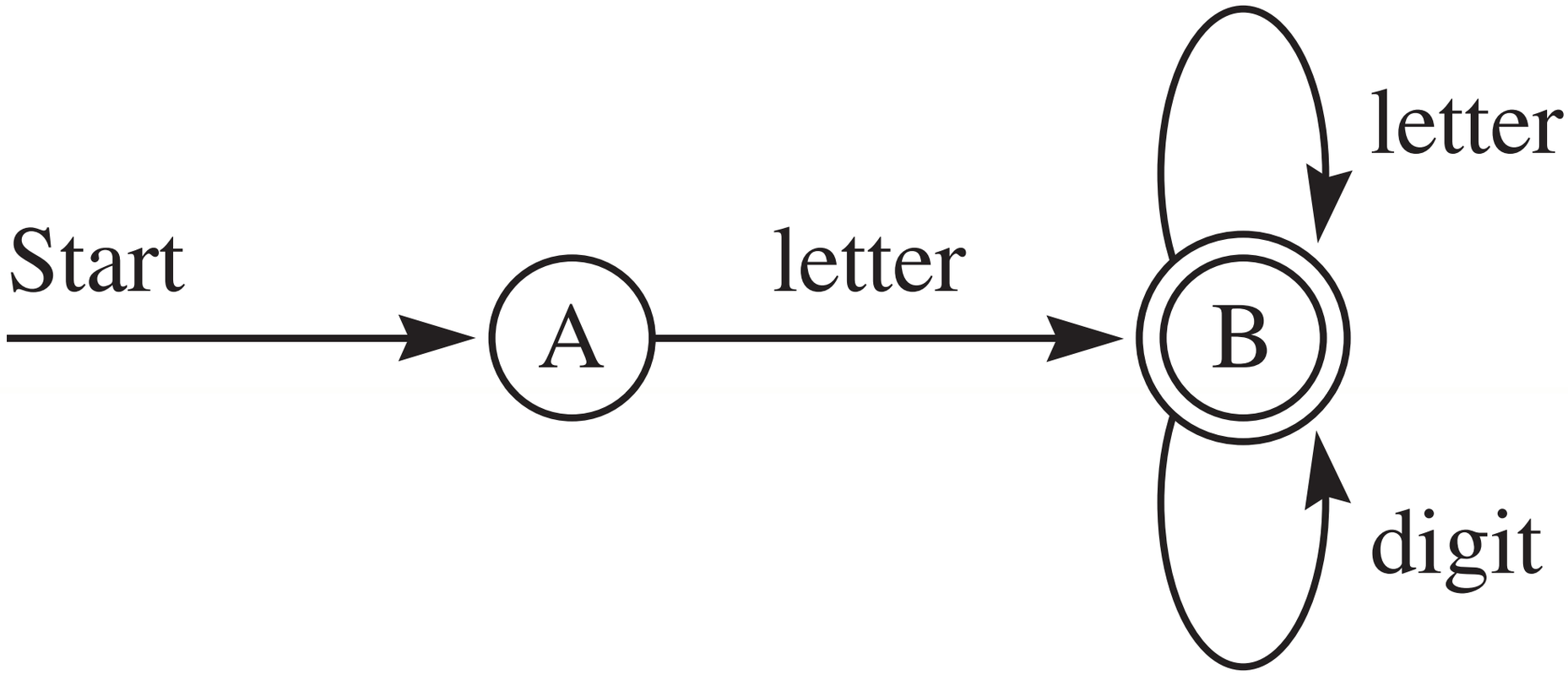
Parsing rules

- 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	Digit
→A	B	C
(B)	B	B
C	C	C

Simplified FSM

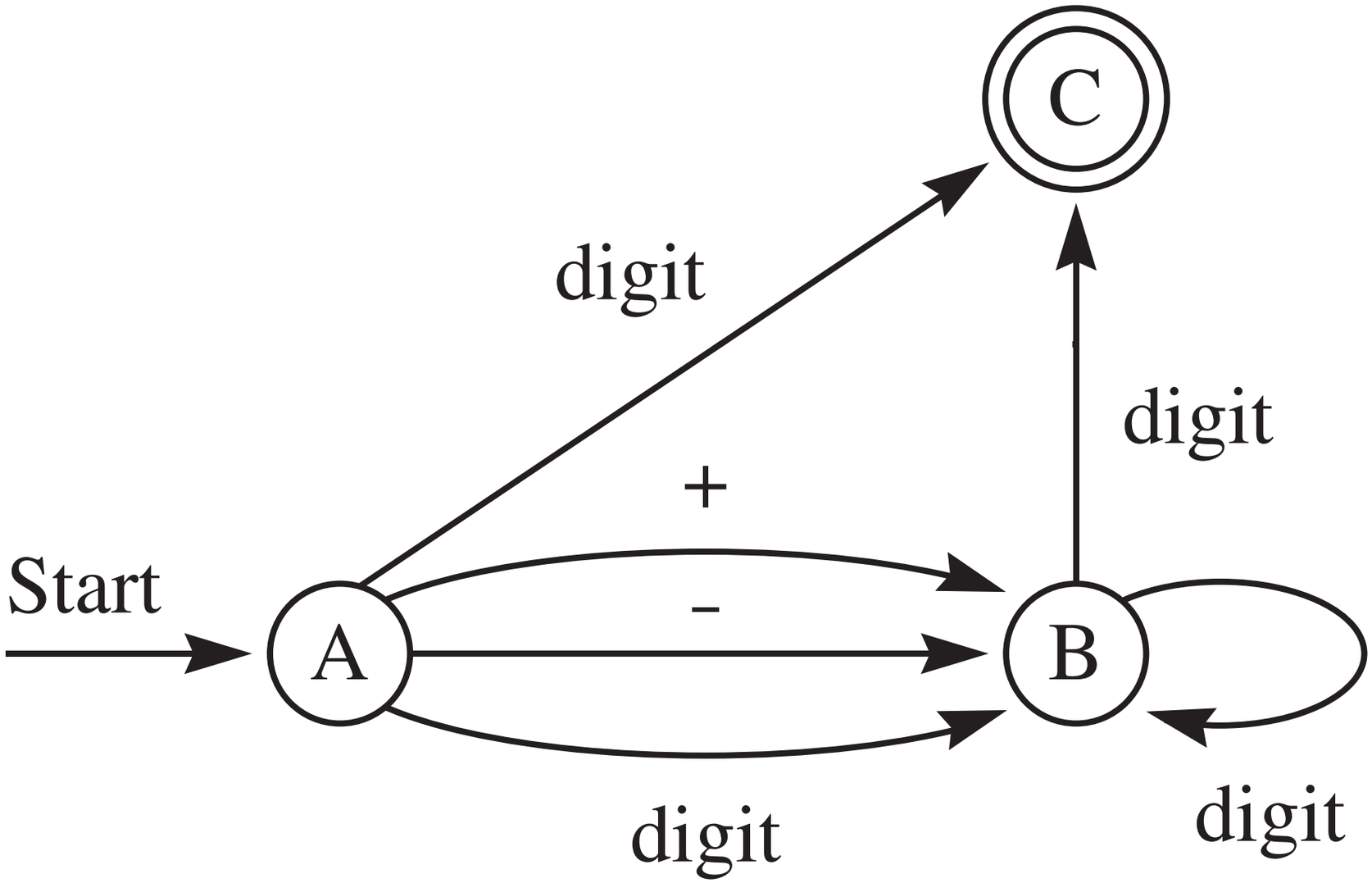
- Not all states have transitions on all terminal symbols
- Two ways to detect an illegal string
  - You may run out of input, and not be in a final state
  - You may be in some state, and the next input character does not correspond to any of the transitions from that state


Start
A
letter
B
letter
digit

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

Nondeterministic FSM

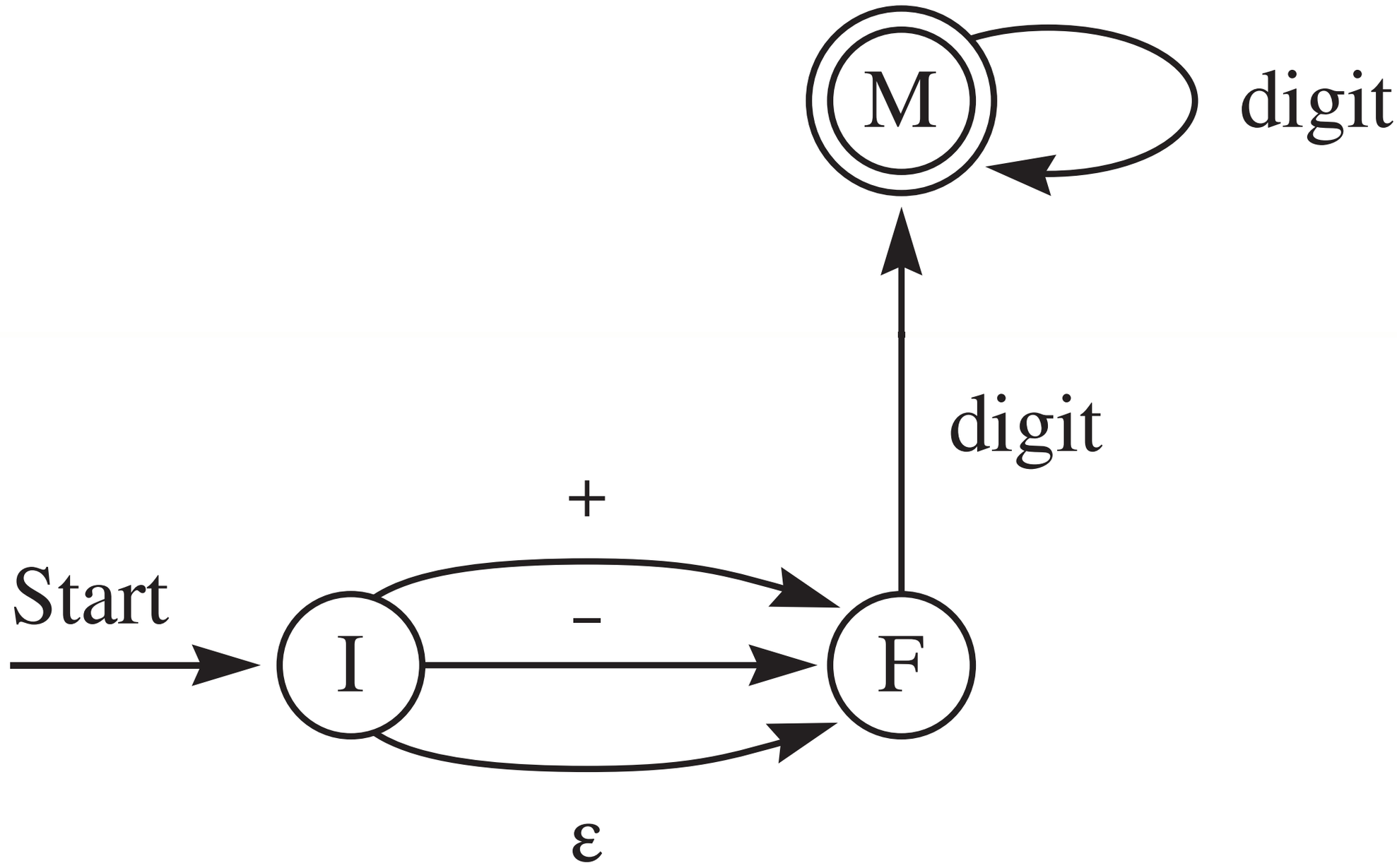
- At least one state has more than one transition from it on the same character
- If you scan the last character and you *are* in a final state, the string *is valid*
- If you scan the last character and you are *not* in a final state, the string *might be invalid*
- To prove invalid, you must try all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

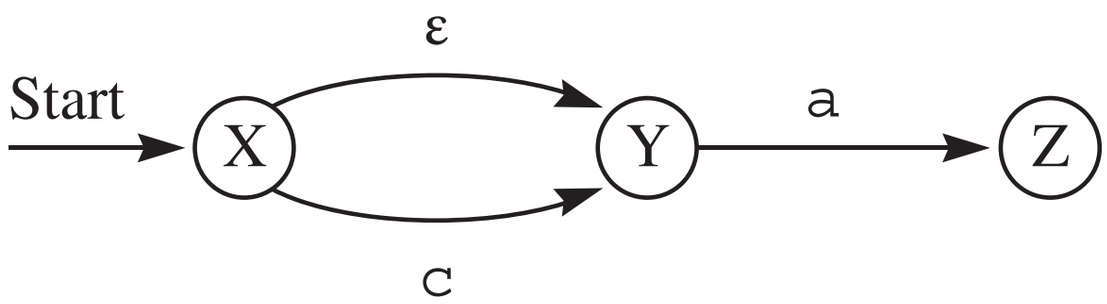
- An empty transition allows you to go from one state to another state without scanning a terminal character
- All finite state machines with empty transitions are considered nondeterministic


M
digit
digit
+
Start
–
I
F
ε

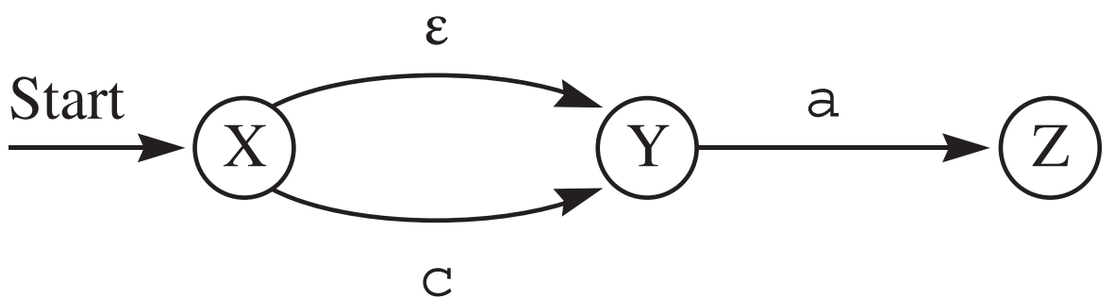
Current State	Next State			
	+	-	Digit	ε
$\rightarrow$I	F	F		F
F			M	
(M)			M	

Removing empty transitions

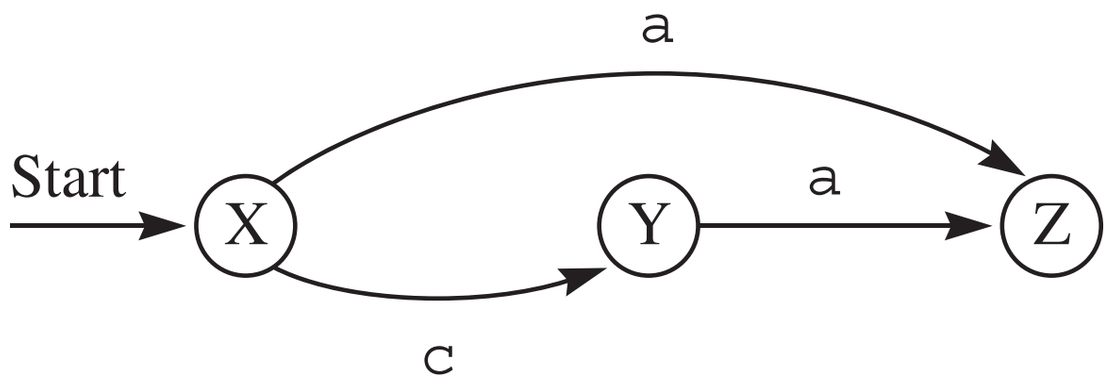
- Given a transition from p to q on ε, for every transition from q to r on a, add a transition from p to r on a.
- If q is a final state, make p a final state



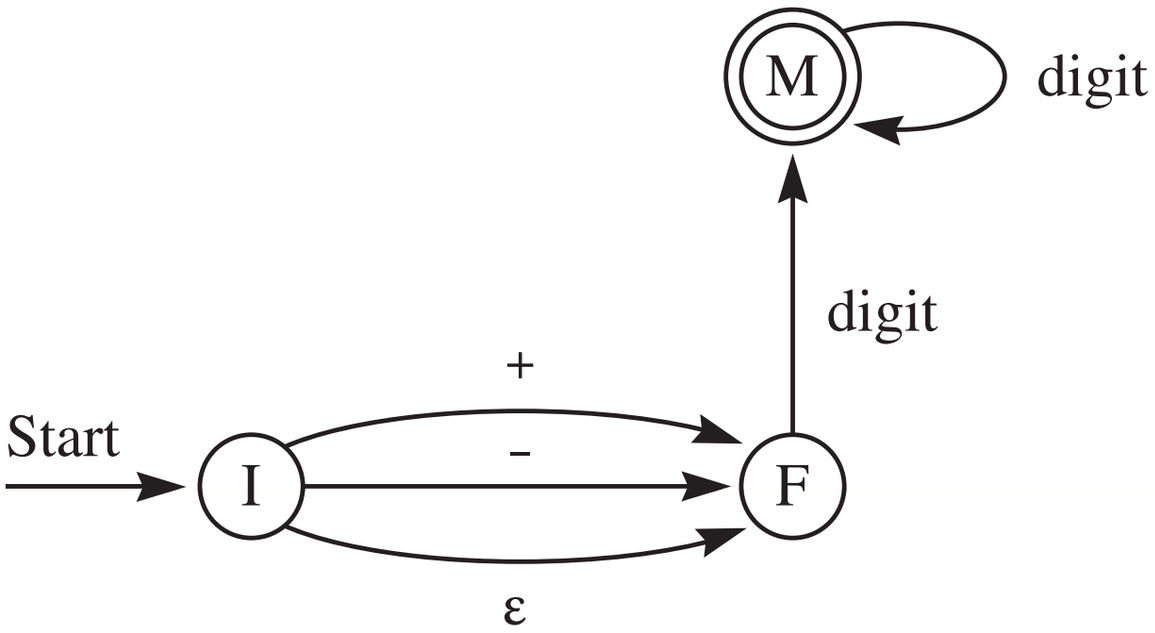
(a) The original FSM.



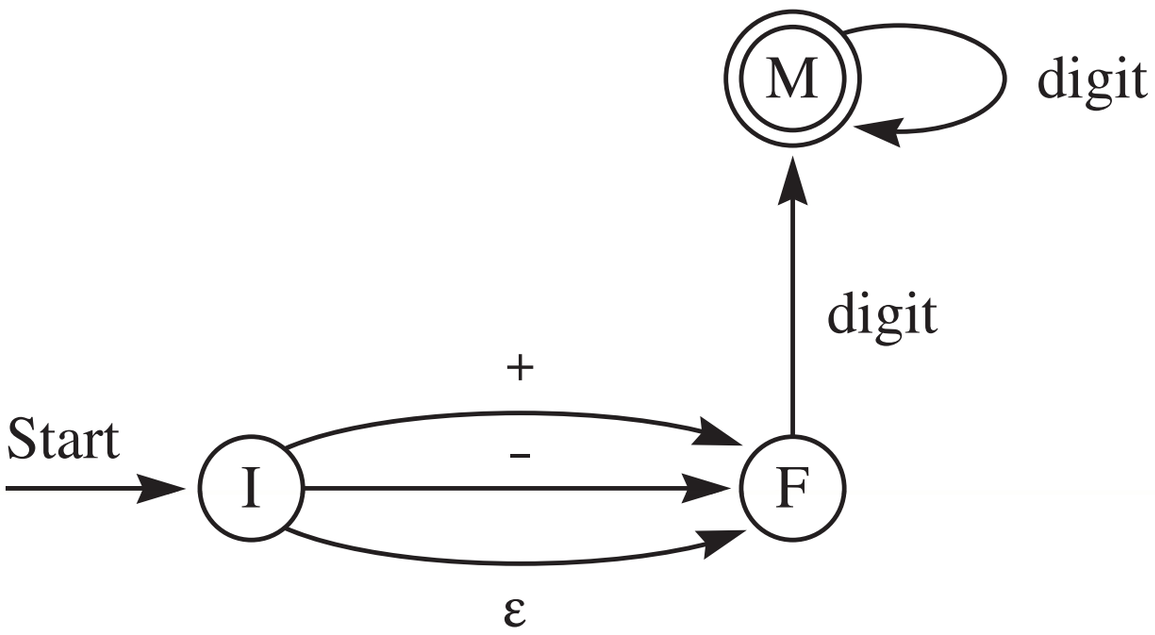
(a) The original FSM.



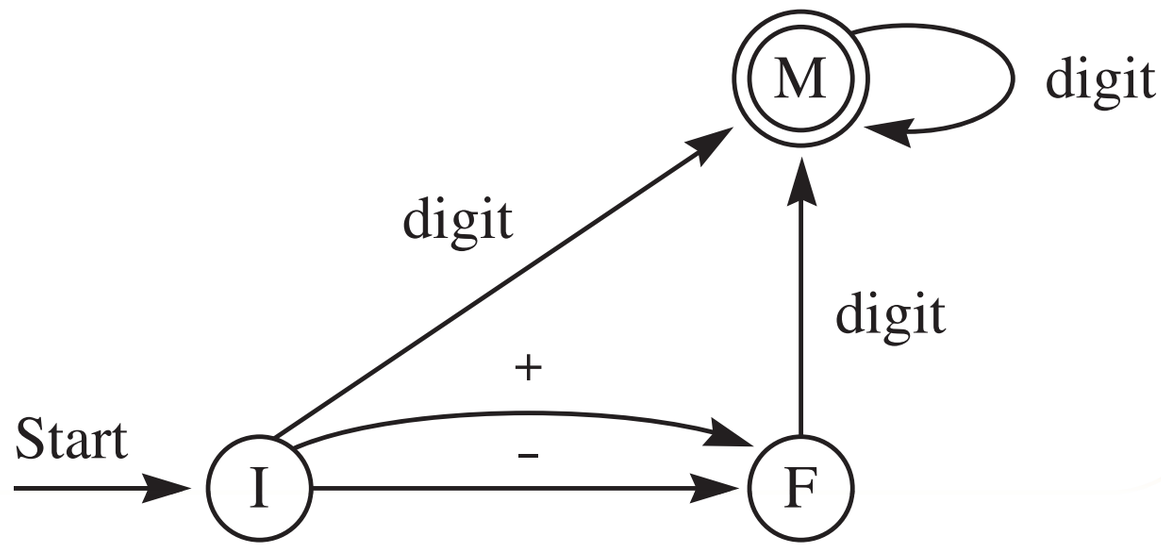
(b) The equivalent FSM without an empty transition.



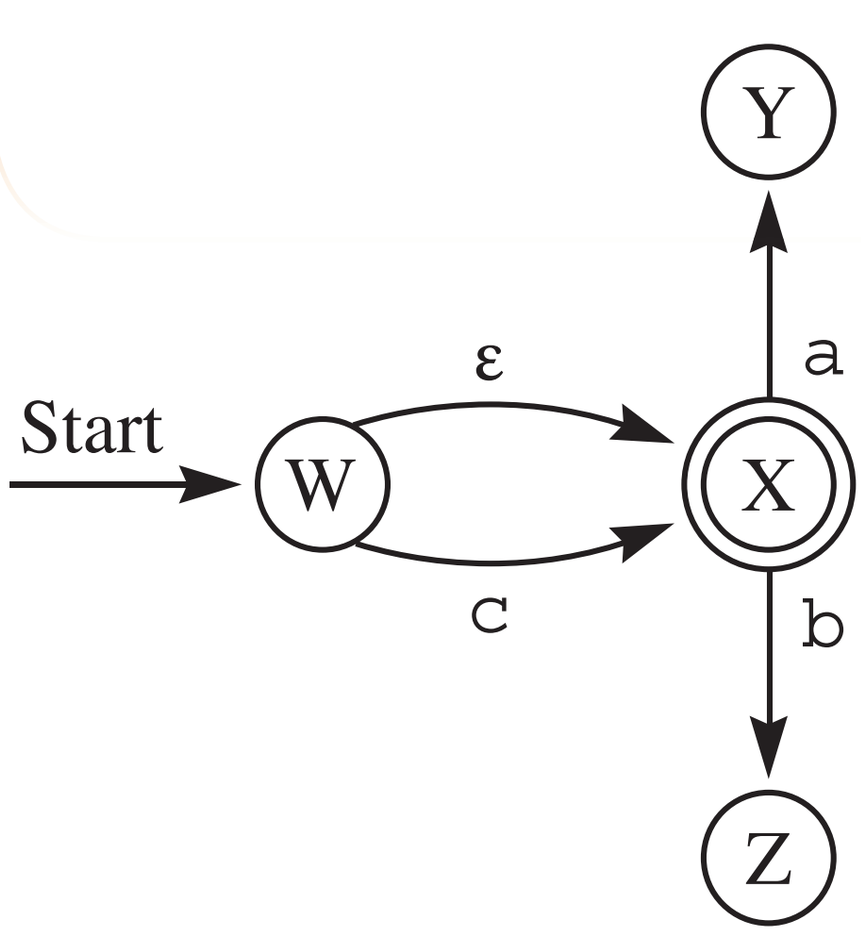
(a) The original FSM.



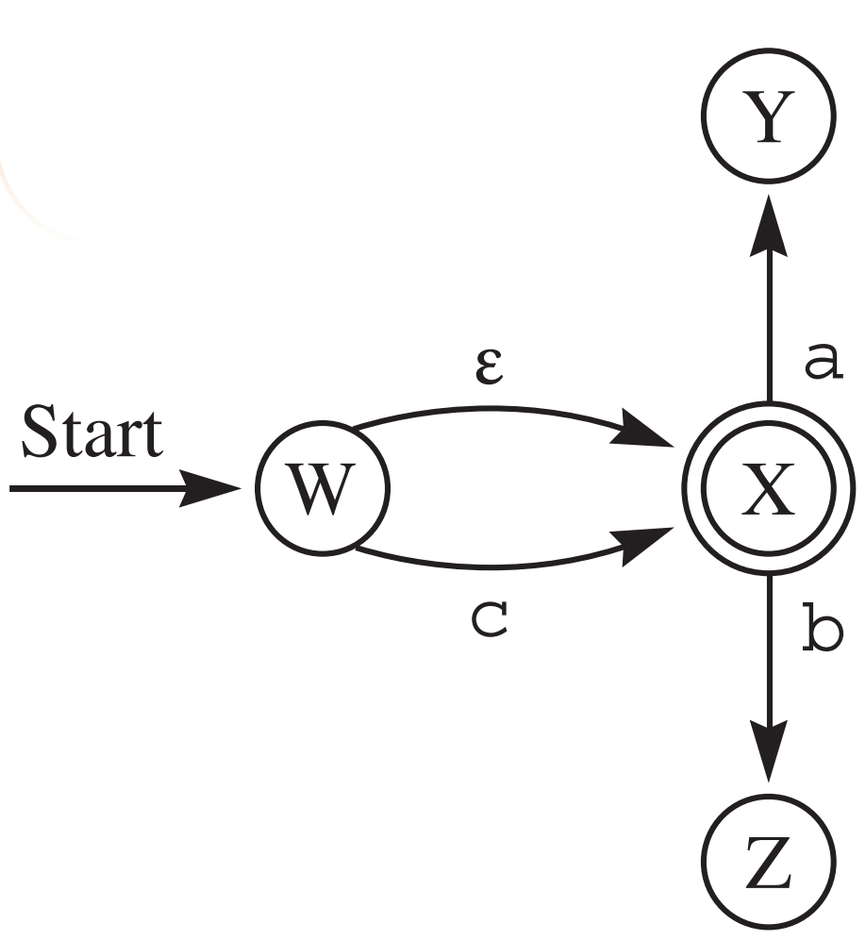
(a) The original FSM.



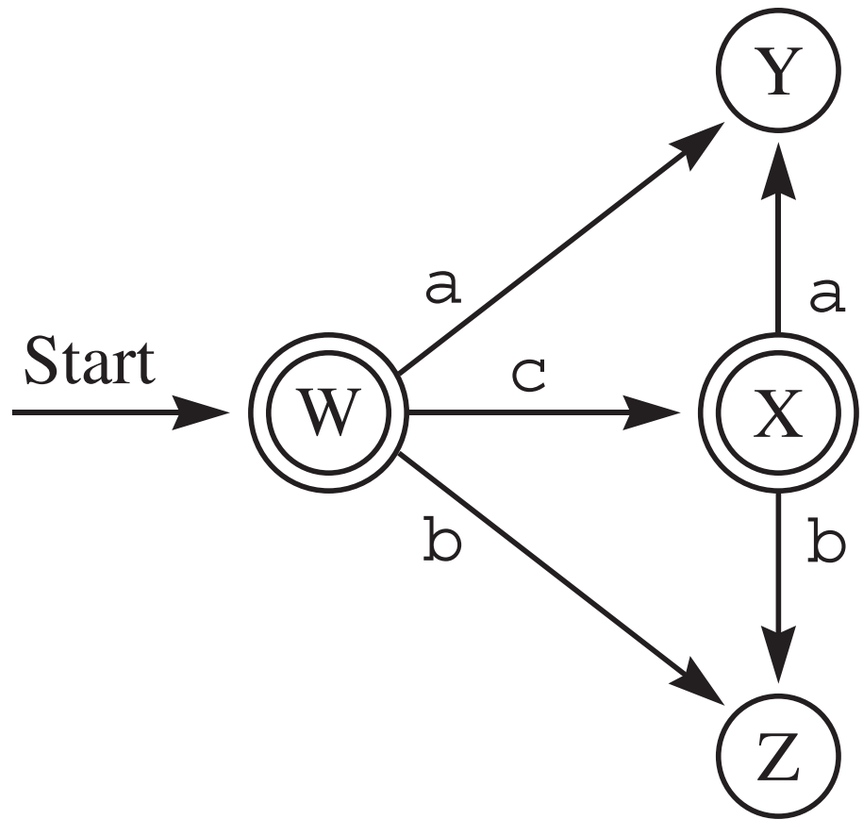
(b) The empty transition removed.



(a) The original FSM.



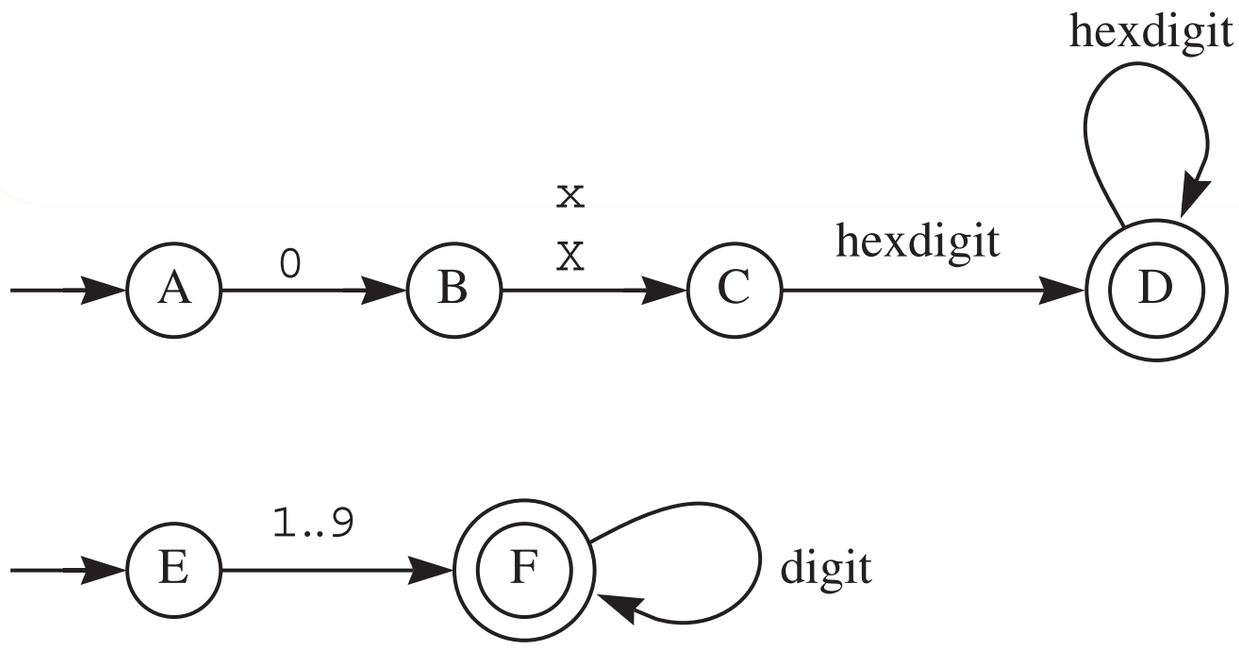
(a) The original FSM.



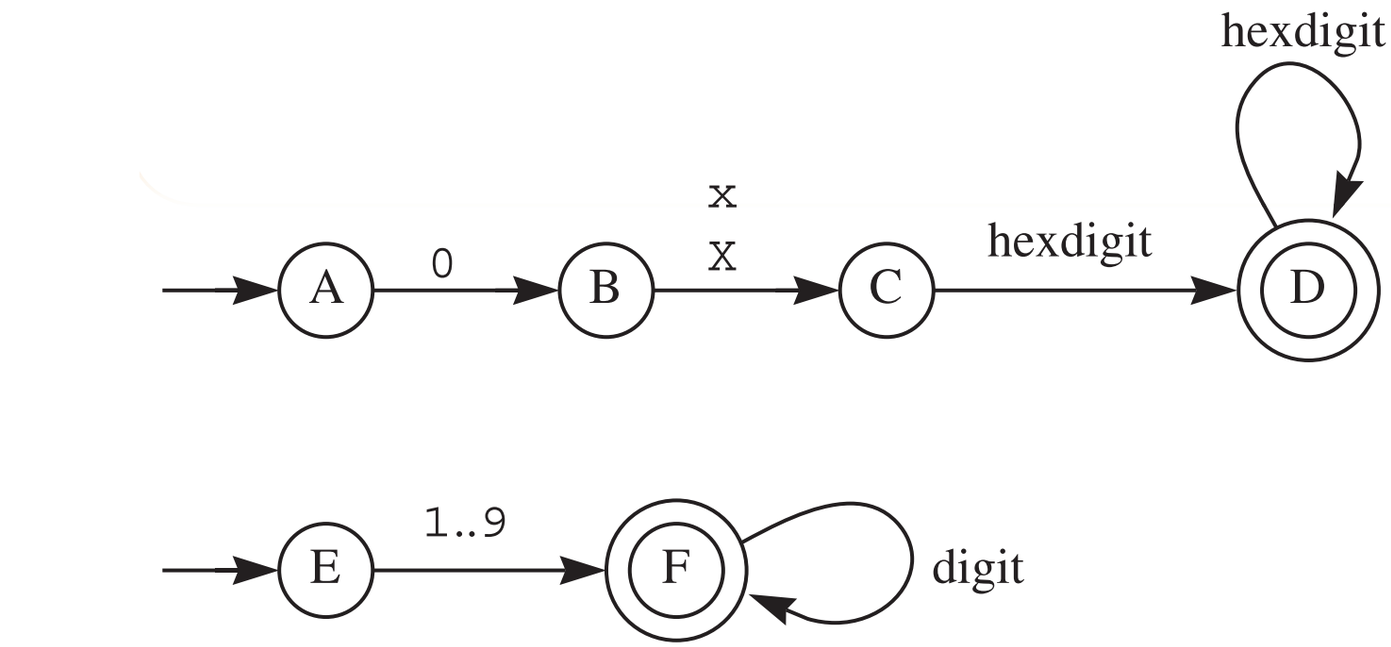
(b) The equivalent FSM without an empty transition.

Multiple token recognizers

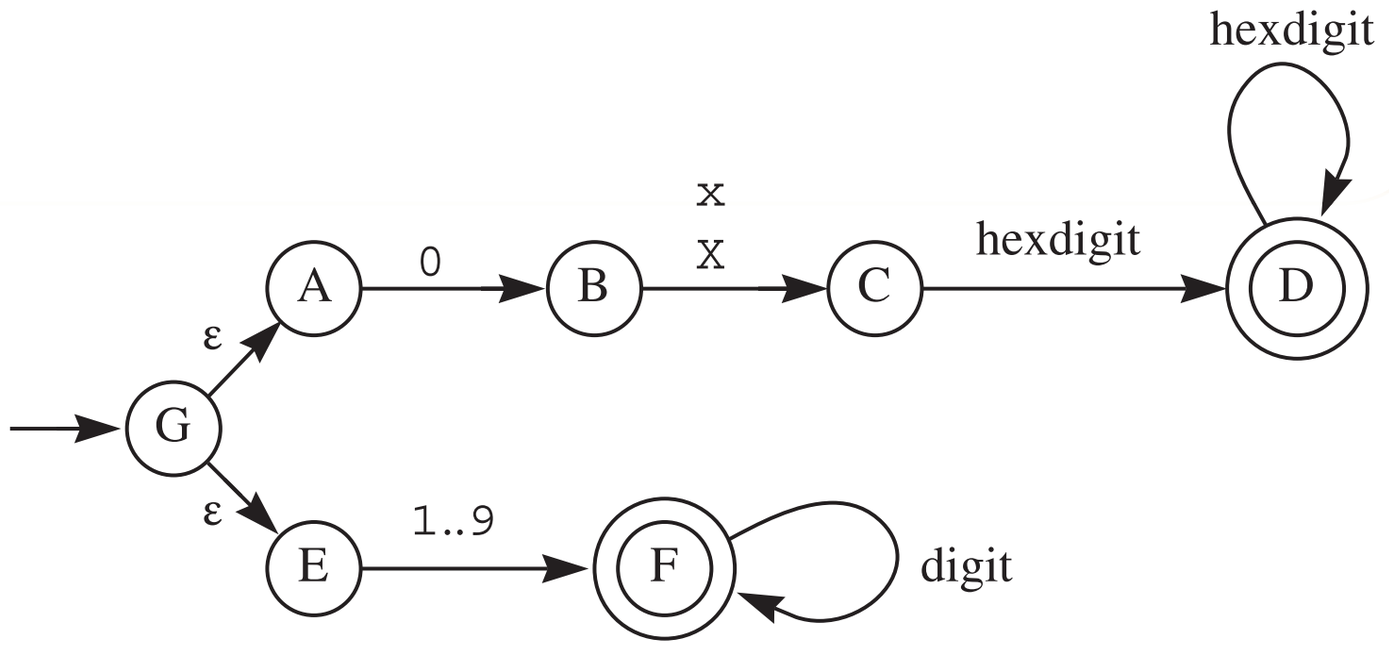
- Token
  - A string of terminal characters that has meaning as a group
- FSM with multiple final states
- The final state determines the token that is recognized



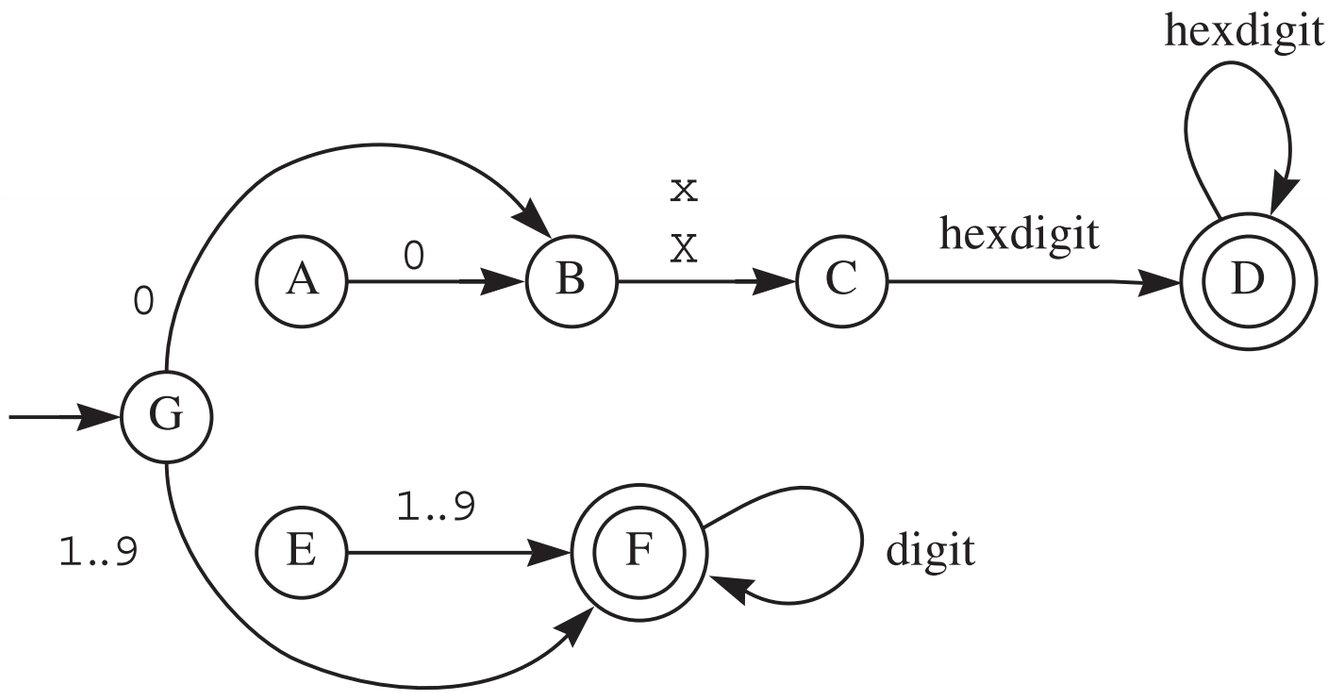
(a) Separate machines for a hexadecimal constant and an unsigned decimal integer.



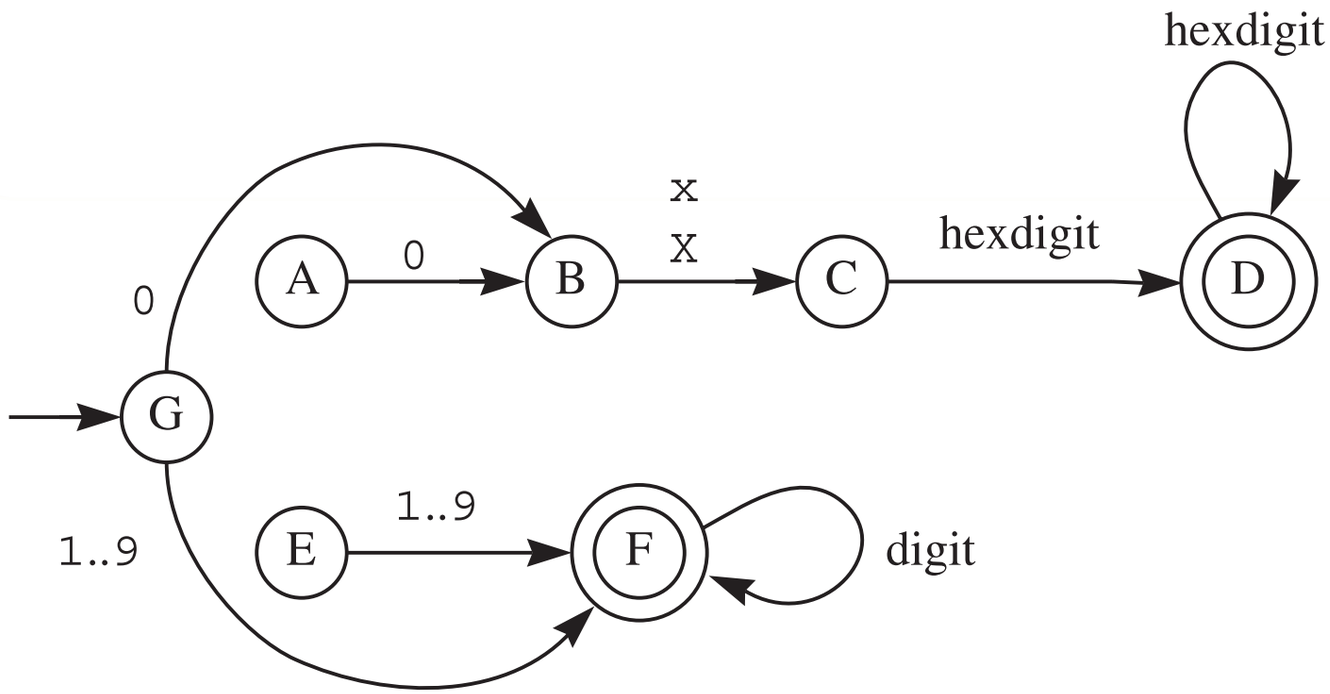
(a) Separate machines for a hexadecimal constant and an unsigned decimal integer.



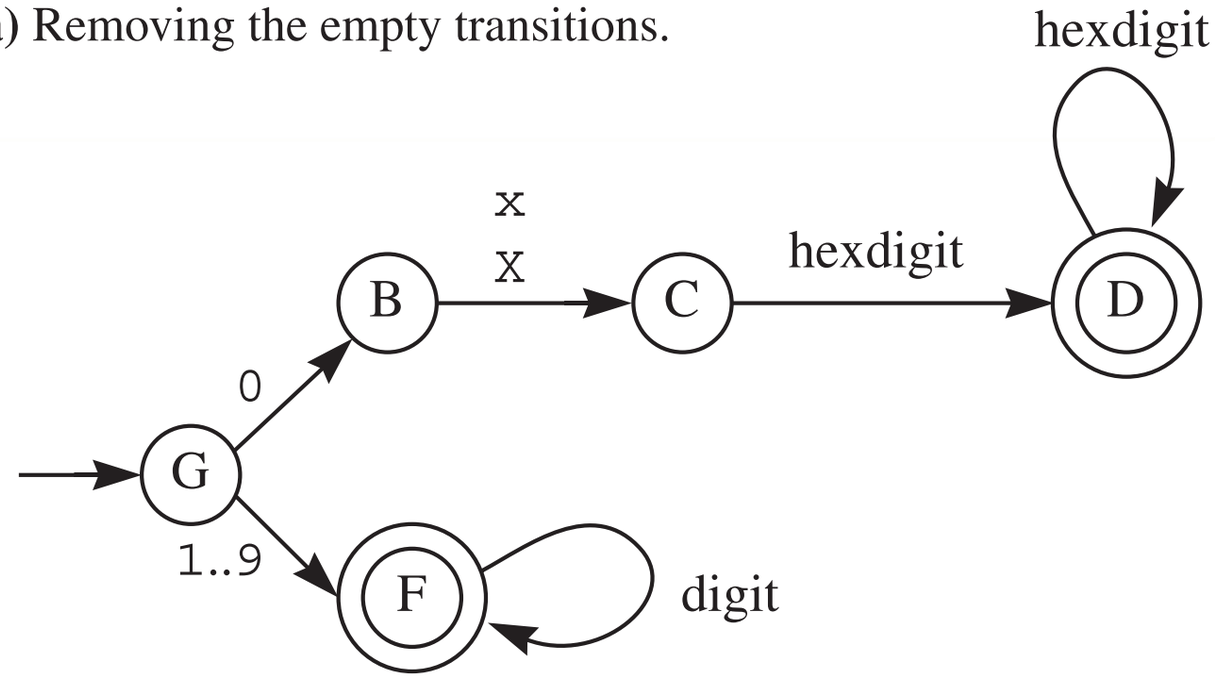
(b) One nondeterministic FSM that recognizes a hexadecimal constant or an unsigned integer token.



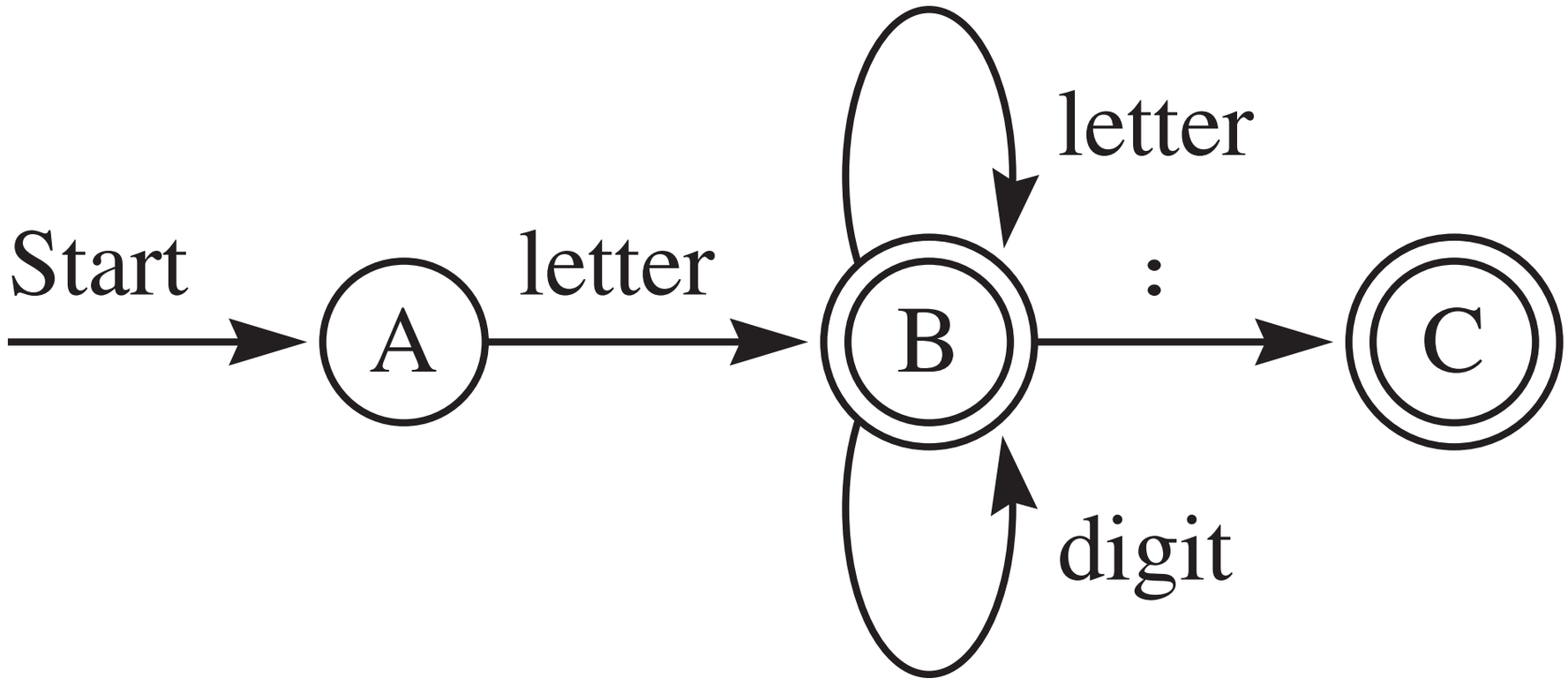
(a) Removing the empty transitions.



(a) Removing the empty transitions.

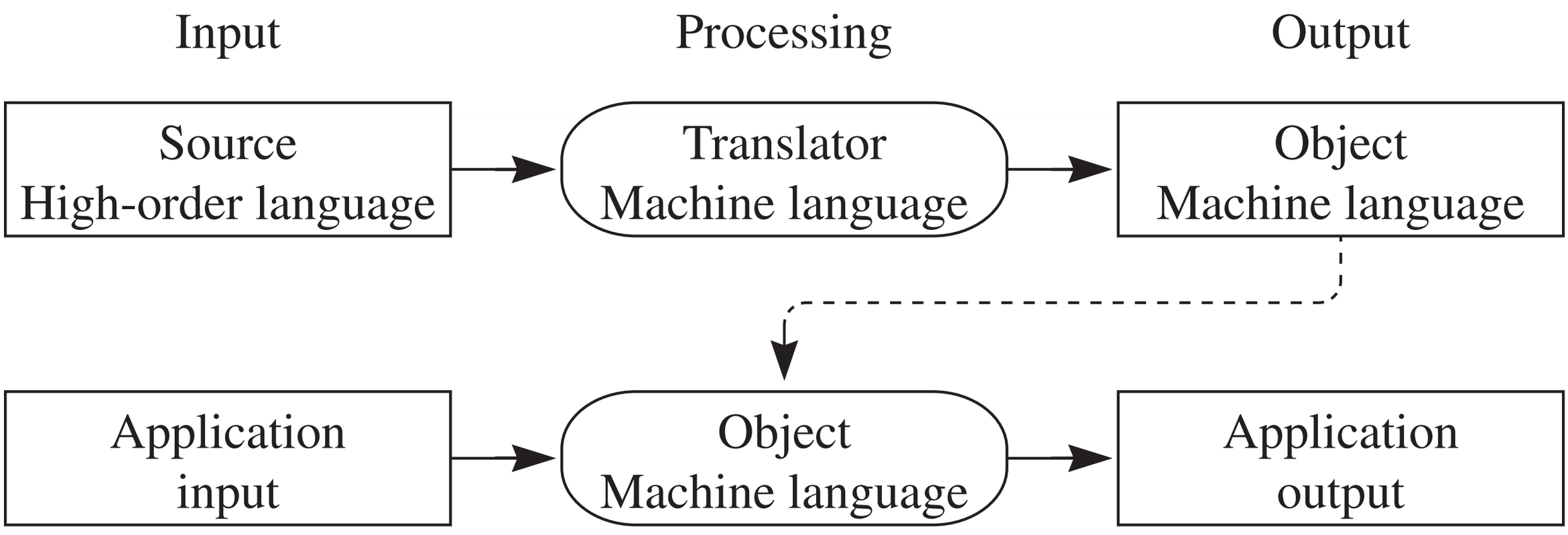


(b) Removing the inaccessible states.


Start
A
letter
B
letter
:
C
digit

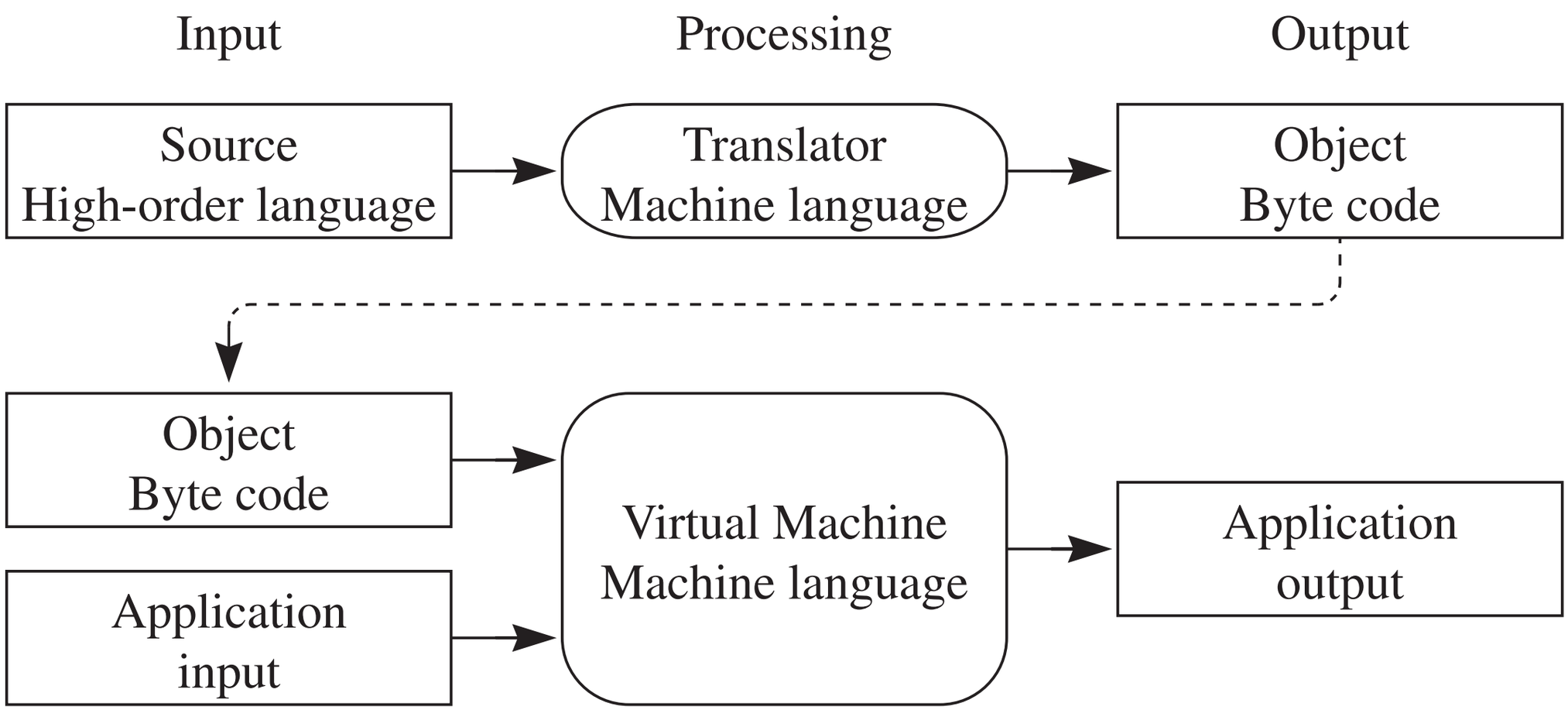

Grammars
More powerful
Less powerful
Finite-state machines
Regular expressions

Compilation



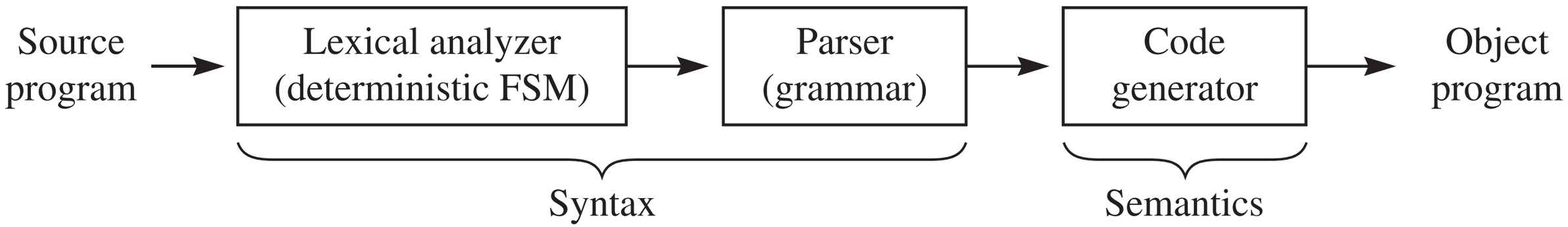
(a) Compilation.

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 lexical analyzer

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

```
def main(text: str):
  fsm = Table()
  valid = fsm.parse(text)
  print(f"{text} is {'' if valid else 'not'} a valid identifier")
```

Input
```
h3ll0
```

Output
```
h3ll0 is a valid
identifier
```

(a) First run.

Input
```
3cab
```

Output
```
3cab is not a valid
identifier
```

(b) Second run.

Input
```
cab#3
```

Output
```
cab#3 is a valid
identifier
```

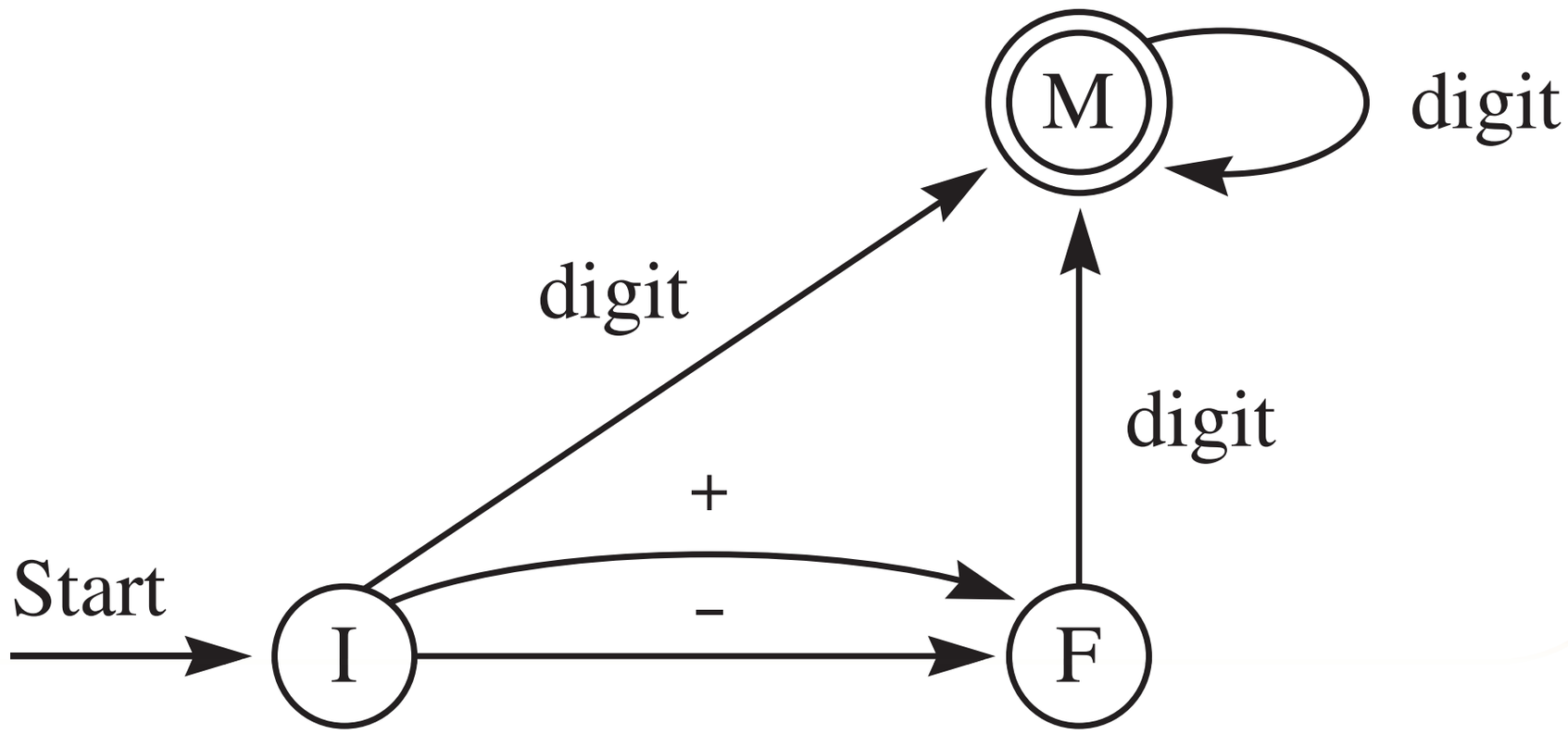
(c) Third run.

```
class Table:
    class States(enum.IntEnum):
        A = 0
        B = 1
        C = 2

    class Kind(enum.IntEnum):
        Letter = 0
        Digit = 1
```

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

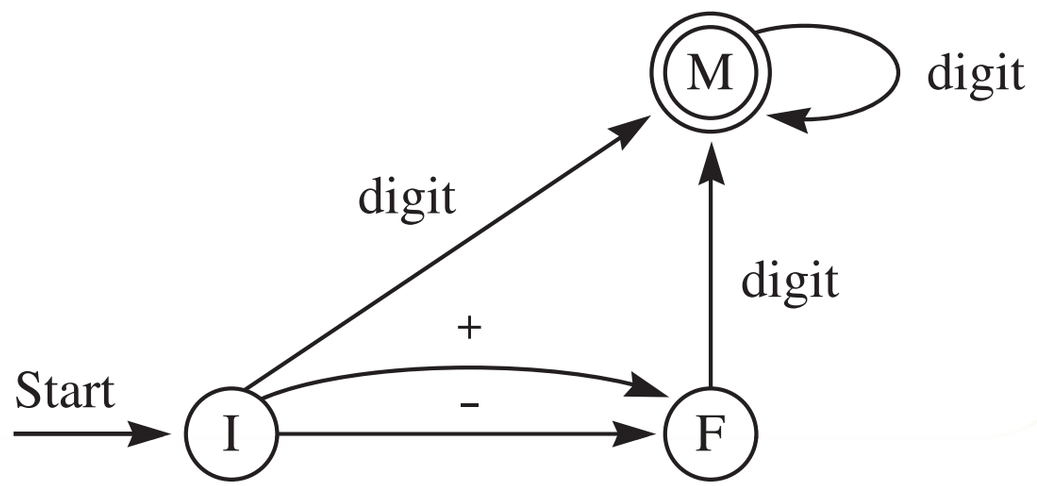
```
transitions: Dict[States, Dict[Kind, States]] = {
    States.A: {Kind.Letter: States.B, Kind.Digit: States.C},
    States.B: {Kind.Letter: States.B, Kind.Digit: States.B},
    States.C: {Kind.Letter: States.C, Kind.Digit: States.C},
}

def parse(self, text: str):
    state = Table.States.A
    for ch in text:
        kind = Table.Kind.Letter if ch.isalpha() else Table.Kind.Digit
        state = Table.transitions[state][kind]
    return state == Table.States.B
```

A direct-code lexical analyzer

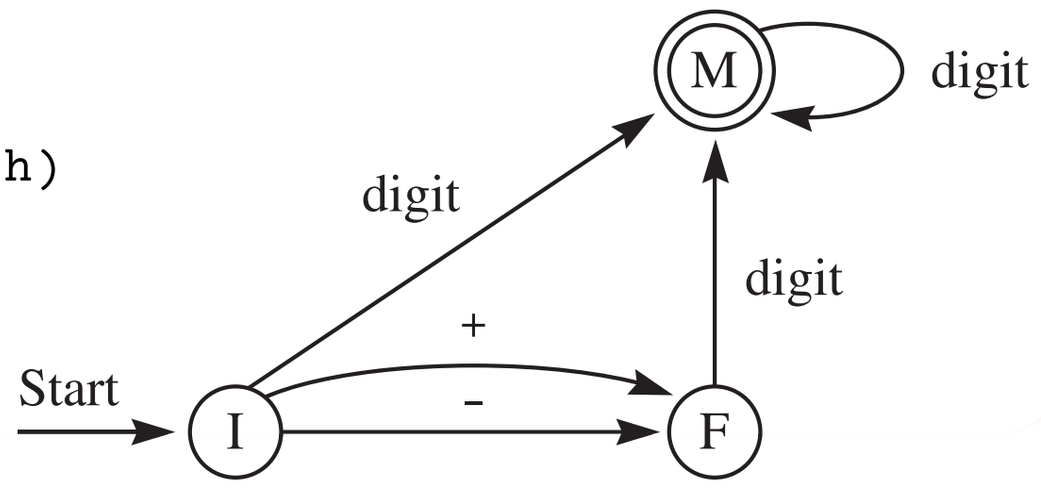

M
digit
digit
digit
+
-
Start
I
F

```
class Direct:
    class States(enum.Enum):
        I = 0
        F = 1
        M = 2
        STOP = 3

    def parse(self, text: str):
        text = text + "\n"
        state = Direct.States.I
        valid, magnitude, sign = True, 0, +1

        while state != Direct.States.STOP and valid:
            ch, text = text[0], text[1:] if len(text) > 1 else ""
            match state:
                case Direct.States.I:
                    if ch == "+":
                        sign, state = 1, Direct.States.F
                    elif ch == "-":
                        sign, state = -1, Direct.States.F
                    elif ch.isdigit():
                        magnitude, state = int(ch), Direct.States.M
                    else:
                        valid = False

                case Direct.States.F:
                    if ch.isdigit():
                        magnitude, state = int(ch), Direct.States.M
                    else:
                        valid = False

                case Direct.States.M:
                    if ch.isdigit():
                        magnitude = 10 * magnitude + int(ch)
                    elif ch == "\n":
                        state = Direct.States.STOP
                    else:
                        valid = False

        return valid, sign * magnitude if valid else None
```

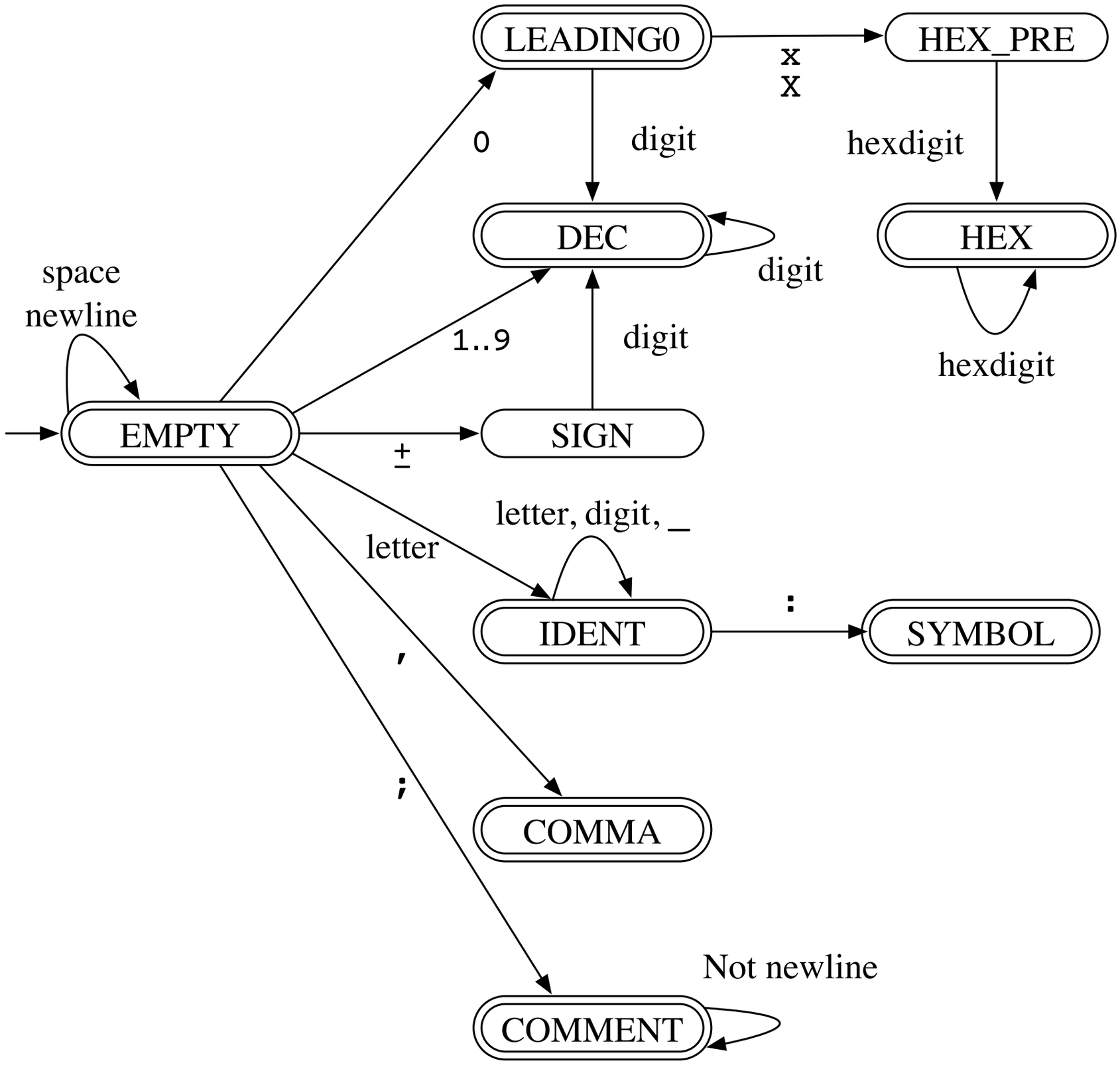
```
class Direct:
    class States(enum.Enum):
        I = 0
        F = 1
        M = 2
        STOP = 3

def parse(self, text: str):
    text = text + "\n"
    state = Direct.States.I
    valid, magnitude, sign = True, 0, +1
```



```
while state != Direct.States.STOP and valid:
    ch, text = text[0], text[1:] if len(text) > 1 else ""
    match state:
        case Direct.States.I:
            if ch == "+":
                sign, state = 1, Direct.States.F
            elif ch == "-":
                sign, state = -1, Direct.States.F
            elif ch.isdigit():
                magnitude, state = int(ch), Direct.States.M
            else:
                valid = False

        case Direct.States.F:
            if ch.isdigit():
                magnitude, state = int(ch), Direct.States.M
            else:
                valid = False

        case Direct.States.M:
            if ch.isdigit():
                magnitude = 10 * magnitude + int(ch)
            elif ch == "\n":
                state = Direct.States.STOP
            else:
                valid = False

return valid, sign * magnitude if valid else None
```



A multiple-token lexical analyzer


LEADING0
HEX_PRE
x
X
0
digit
hexdigit
DEC
HEX
digit
space
newline
1..9
digit
hexdigit
EMPTY
SIGN
±
letter, digit, _
letter
IDENT
:
SYMBOL
'
;
COMMA
Not newline
COMMENT

Final state	Python class name	Description	Sample values
EMPTY	`Empty`	The terminal `\n`	
COMMENT	`Comment`	Comments beginning with `;`	`;local variable`
COMMA	`Comma`	The terminal `,`	`,`
DEC, LEADING0	`Decimal`	Decimal integers	`0`, `257`, `-15`
HEX	`Hexadecimal`	Hexadecimal integers	`0x0`, `0xFEED`
IDENT	`Identifier`	Identifiers	`main`, `a1_`
SYMBOL	`Symbol`	Symbol declarations	`main:`, `a1_:`

Input

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

Output

```
Identifier(value='Here')
Identifier(value='is')
Identifier(value='A47')
Decimal(value=48)
Identifier(value='B')
Empty()
Identifier(value='C')
Decimal(value=-49)
Identifier(value='ALongIdentifier')
Decimal(value=50)
Identifier(value='D16')
Decimal(value=-51)
Empty()
```

(a) First run.

Input

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

Output

```
Identifier(value='Here')
Identifier(value='is')
Identifier(value='A47')
Invalid()
Decimal(value=48)
Identifier(value='B')
Empty()
Identifier(value='C')
Decimal(value=49)
Empty()
```

(b) Second run.

Lexer design principles

- If a transition exists from the current state on the next terminal, that transition must be taken. This greedy strategy produces the longest match possible. It is also referred to as *maximal munch*.
- 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.

The Pep/10 token classes

```
 4  @dataclass
 5  class Identifier:
 6      value: str
 7
 8
 9  @dataclass
10  class Symbol:
11      value: str
12
13
14  @dataclass
15  class Comment:
16      value: str
```

```
19  @dataclass
20  class Empty: ...
21
22
23  @dataclass
24  class Invalid: ...
25
26
27  @dataclass
28  class Comma: ...
```

```
31  @dataclass
32  class Decimal:
33      value: int
34
35
36  @dataclass
37  class Hex:
38      value: int
```

The `io.StringIO` class

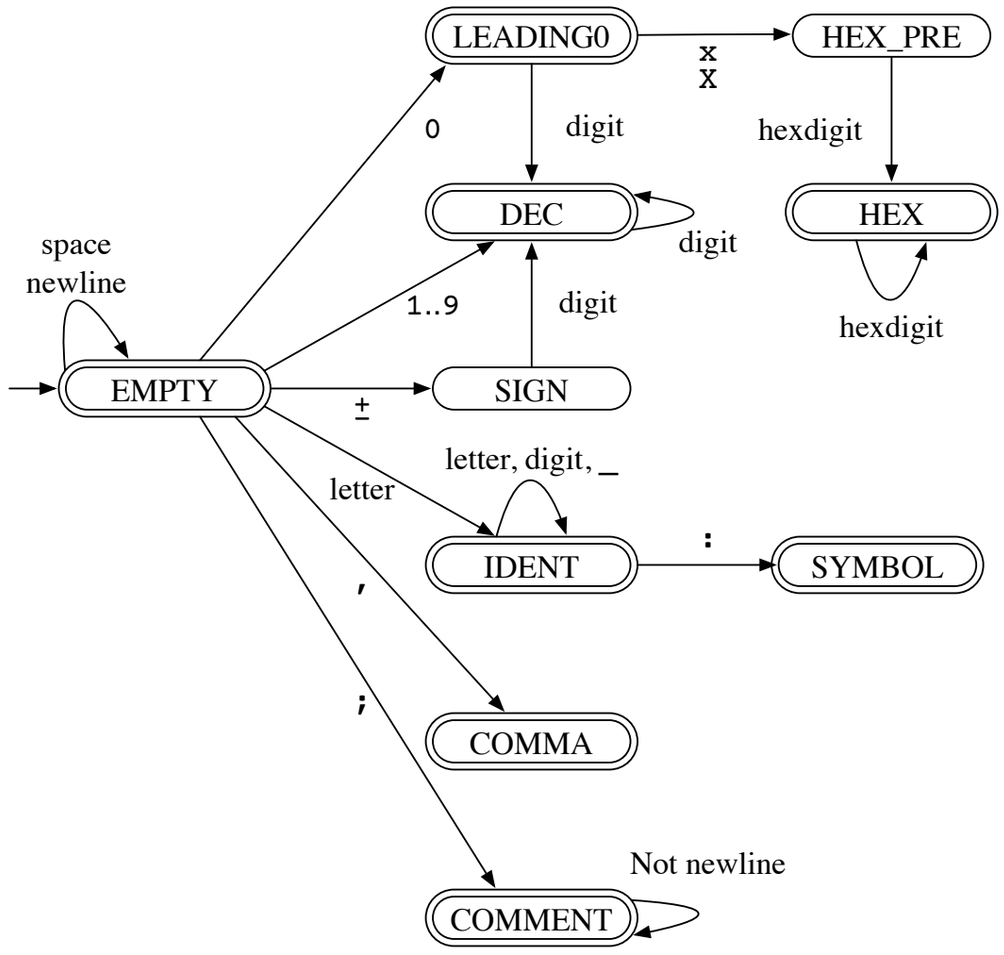
A text stream using an in-memory text buffer.
Contains an internal cursor.

- `read(1)` Scans at most one character, and advances the internal cursor.
- `tell()` Returns the position of the internal cursor.
- `seek(position)` Moves the internal cursor to `position`.

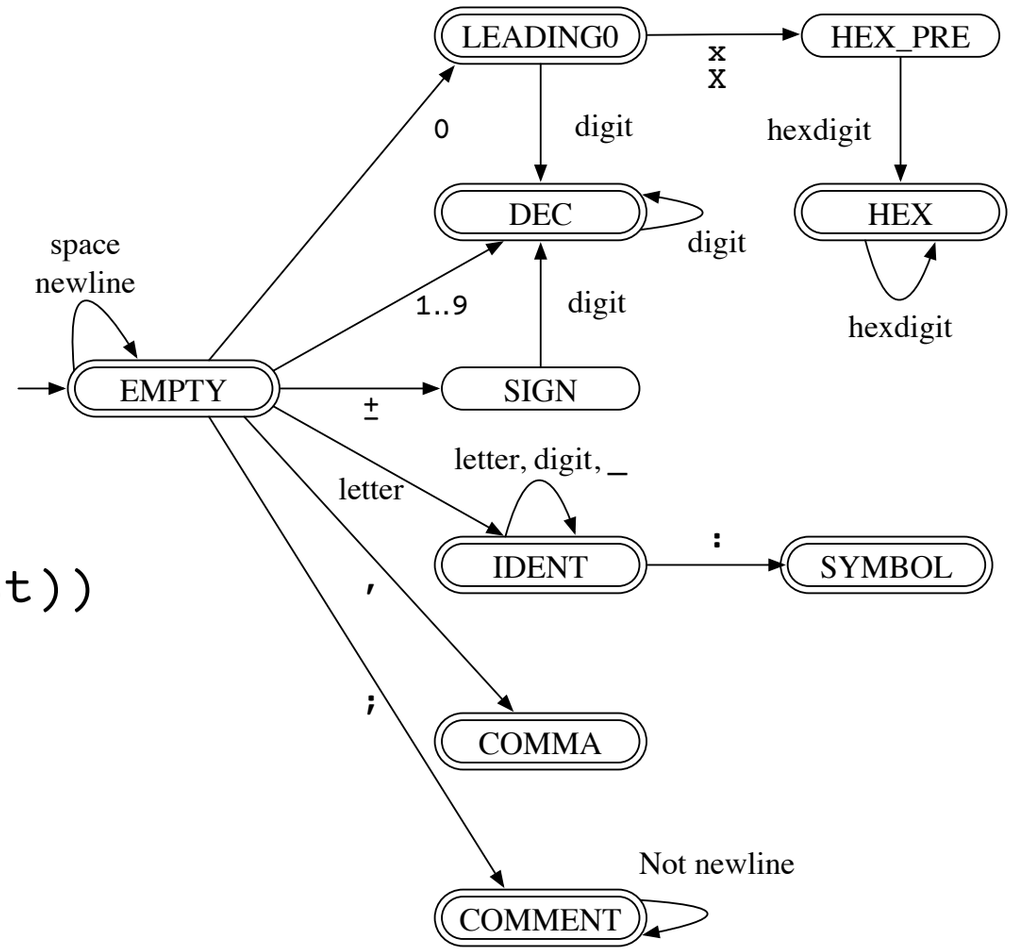
Figure 7.34

```
class Lexer(TokenProducer[Token]):
    class States(Enum):
        START, COMMENT, IDENT, LEADING0, HEX_PRE = range(0, 5)
        HEX, SIGN, DEC, STOP = range(5, 9)

    def __init__(self, buffer: io.StringIO) -> None:
        self.buffer: io.StringIO = buffer

    def __iter__(self) -> "Lexer":
        return self

    def __next__(self) -> Token:
        state: Lexer.States = Lexer.States.START
        as_str_list: List[str] = []
        as_int: int = 0
        sign: Literal[-1, 1] = 1
token: Token = tokens.Empty()
initial_pos = self.buffer.tell()

while state != Lexer.States.STOP and (
    type(token) is not tokens.Invalid
):
    prev_pos = self.buffer.tell()
    ch: str = self.buffer.read(1)
    if len(ch) == 0:
        if initial_pos == prev_pos:
            raise StopIteration()
        ch = "\n"

    ...

return token
```

```
match state:
    case Lexer.States.START:
        if ch == "\n":
            state = Lexer.States.STOP
        elif ch == ",":
            state = Lexer.States.STOP
            token = tokens.Comma()
        elif ch.isspace():
            pass
        elif ch == ";":
            state = Lexer.States.COMMENT
        elif ch.isalpha():
            as_str_list.append(ch)
            state = Lexer.States.IDENT
        elif ch == "0":
            state = Lexer.States.LEADING0
        elif ch.isdecimal():
            state = Lexer.States.DEC
            as_int = ord(ch) - ord("0")
        elif ch == "+" or ch == "-":
            state = Lexer.States.SIGN
            sign = -1 if ch == "-" else 1
        else:
            token = tokens.Invalid()
```



```
case Lexer.States.IDENT:
    if ch.isalnum() or ch == "_":
        as_str_list.append(ch)
    elif ch == ":":
        state = Lexer.States.STOP
        token = tokens.Symbol("".join(as_str_list))
else:
    self.buffer.seek(prev_pos, os.SEEK_SET)
    state = Lexer.States.STOP
    as_str = "".join(as_str_list)
    token = tokens.Identifier(as_str)
```



Here is

```
buf_pos
init_pos
prev_pos
```

```
H e r e   i s
buf_pos  ^
init_pos
prev_pos

    self.buffer: io.StringIO = buffer
```

H e r e i s

```
buf_pos  ^
init_pos ^
prev_pos
```

```
init_pos = self.buffer.tell()
```

```
H e r e   i s
buf_pos  ^
init_pos ^
prev_pos ^
```

```
prev_pos = self.buffer.tell()
```

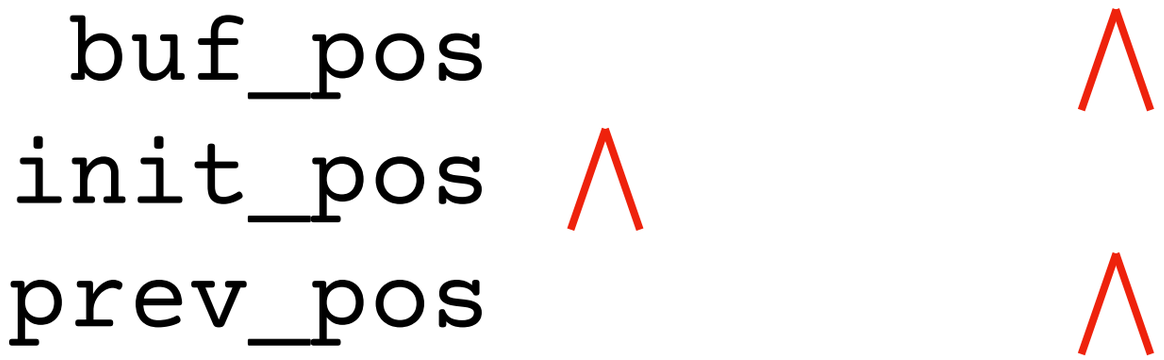
```
H e r e   i s
buf_pos    ^
init_pos   ^
prev_pos   ^

        ch: str = self.buffer.read(1)
```

```
H e r e   i s
buf_pos    ^
init_pos   ^
prev_pos       ^
```

```
prev_pos = self.buffer.tell()
```

```
H e r e   i s
buf_pos
init_pos  ^
prev_pos      ^

      ch: str = self.buffer.read(1)
```

```
H e r e   i s
```




```
        prev_pos = self.buffer.tell()
```

```
H e r e   i s
buf_pos      ^
init_pos  ^
prev_pos             ^

    ch: str = self.buffer.read(1)
```

```
H e r e   i s
```

buf_pos ^

init_pos ^

prev_pos ^

```
prev_pos = self.buffer.tell()
```

```
H e r e   i s
buf_pos
init_pos  ^
prev_pos                ^

    ch: str = self.buffer.read(1)
```

```
H e r e   i s
buf_pos
init_pos  ^
prev_pos                    ^

        prev_pos = self.buffer.tell()
```

```
H e r e   i s

buf_pos
init_pos  ^
prev_pos                  ^

      ch: str = self.buffer.read(1)
```

```
H e r e   i s
```

```
 buf_pos
init_pos
prev_pos
```

```
 self.buffer.seek(prev_pos, os.SEEK_SET)
```

```
H e r e   i s
```

```
buf_pos
```

```
init_pos
prev_pos
```

Return `Identifier` token

H e r e i s

buf_pos
init_pos
prev_pos

```
init_pos = self.buffer.tell()
```

H e r e i s

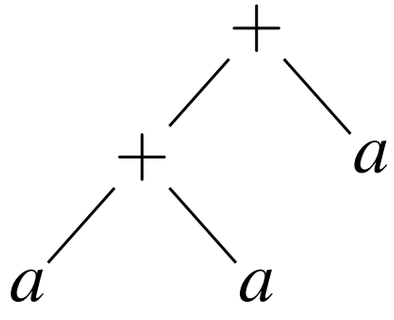
buf_pos
init_pos
prev_pos

```
prev_pos = self.buffer.tell()
```

A recursive descent parser

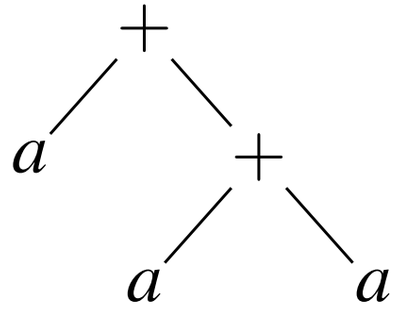
$$
\begin{aligned}
N &= \{ \text{E}, \text{T}, \text{F} \} \\
T &= \{ \texttt{+}, \texttt{*}, \texttt{(}, \texttt{)}, \texttt{a} \} \\
P &= \text{the productions} \\
&\qquad 1.\ \text{E} \rightarrow \text{E} \texttt{ + } \text{T} \\
&\qquad 2.\ \text{E} \rightarrow \text{T} \\
&\qquad 3.\ \text{T} \rightarrow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

E ⇒	**E + T**	Rule 1
⇒	**E + T** + T	Rule 1
⇒	**T** + T + T	Rule 3
⇒	**F** + T + T	Rule 4
⇒	**a** + T + T	Rule 6
⇒	a + **F** + T	Rule 4
⇒	a + **a** + T	Rule 6
⇒	a + a + **F**	Rule 4
⇒	a + a + **a**	Rule 6



(a) Derivation and parse tree using left-to-right, leftmost derivation (LL)

E ⇒	**E + T**	Rule 1
⇒	E + **F**	Rule 4
⇒	E + **a**	Rule 6
⇒	**E + T** + a	Rule 1
⇒	E + **F** + a	Rule 4
⇒	E + **a** + a	Rule 6
⇒	**T** + a + a	Rule 3
⇒	**F** + a + a	Rule 4
⇒	**a** + a + a	Rule 6



(b) Derivation and parse tree using left-to-right, rightmost derivation (LR)

The left-recursive grammar problem

The left-recursive grammar problem

- LL parsers apply the production rules in numeric order.

The left-recursive grammar problem

- LL parsers apply the production rules in numeric order.
- An LL parse of the grammar of Figure 7.5 results in infinite recursion.

The left-recursive grammar problem

- LL parsers apply the production rules in numeric order.
- An LL parse of the grammar of Figure 7.5 results in infinite recursion.

$N =$ { E, T, F }
$T =$ { `+`, `*`, `(`, `)`, `a` }
$P =$ the productions
1. E → E `+` T
2. E → T
3. T → T `*` F
4. T → F
5. F → `(` E `)`
6. F → `a`

$S =$ E

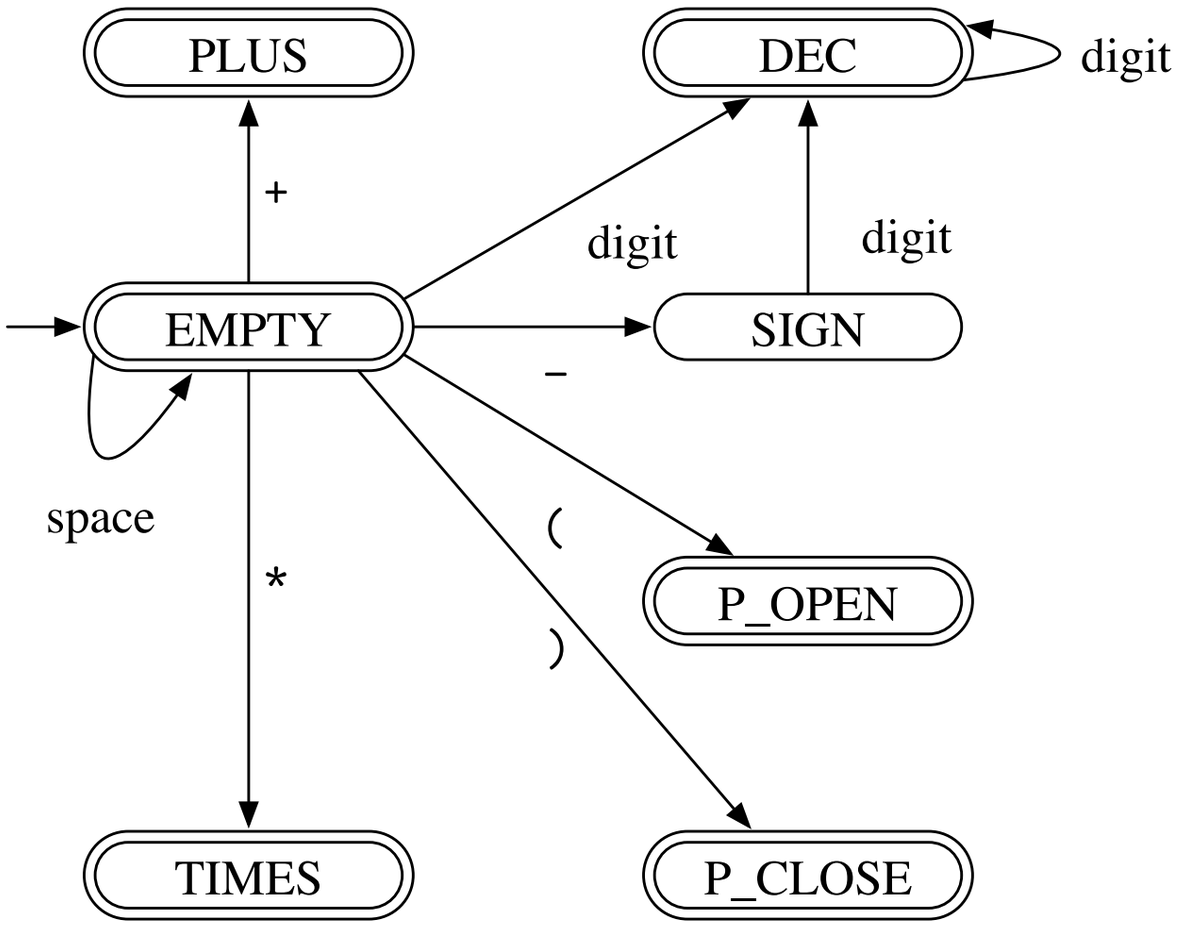
The left-recursive grammar problem

- LL parsers apply the production rules in numeric order.
- An LL parse of the grammar of Figure 7.5 results in infinite recursion.

$N =$ { E, T, F }
$T =$ { +, *, (,), a }
$P =$ the productions
1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → a

$S =$ E

$$
\begin{aligned}
\mathrm{E} &\Rightarrow \mathbf{E + T} && \text{Rule 1}\\
&\Rightarrow \mathbf{E + T} + \mathrm{T} && \text{Rule 1}\\
&\Rightarrow \mathbf{E + T} + \mathrm{T} + \mathrm{T} && \text{Rule 1}\\
&\Rightarrow^{*} \mathbf{E + T} + \mathrm{T} + \cdots + \mathrm{T} + \mathrm{T}
\end{aligned}
$$

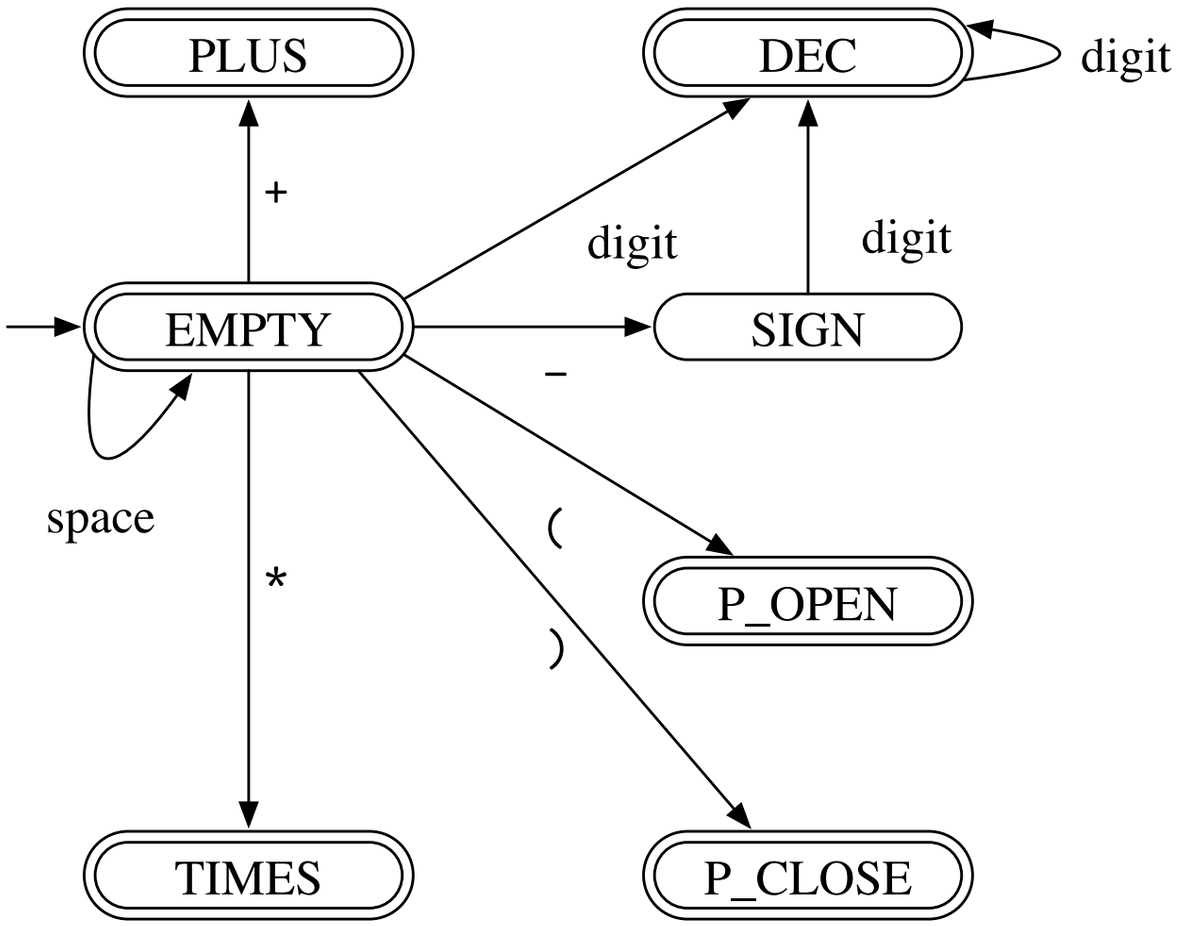


(a) A FSM of a multi-token lexical analyzer.

$N =$ { <E>, <T>, <F> }
$T =$ { PLUS, TIMES, P_OPEN, P_CLOSE, DECIMAL }
$P =$ the productions
1. <E> → <T> [PLUS <E>]
2. <T> → <F> [TIMES <T>]
3. <F> → P_OPEN <E> P_CLOSE
4. <F> → DECIMAL

$S =$ <E>

(b) An expanded expression grammar.



(a) A FSM of a multi-token lexical analyzer.

$N =$ { <E>, <T>, <F> }
$T =$ { PLUS, TIMES, P_OPEN, P_CLOSE, DECIMAL }
$P =$ the productions
1. <E> → <T> [PLUS <E>]
2. <T> → <F> [TIMES <T>]
3. <F> → P_OPEN <E> P_CLOSE
4. <F> → DECIMAL

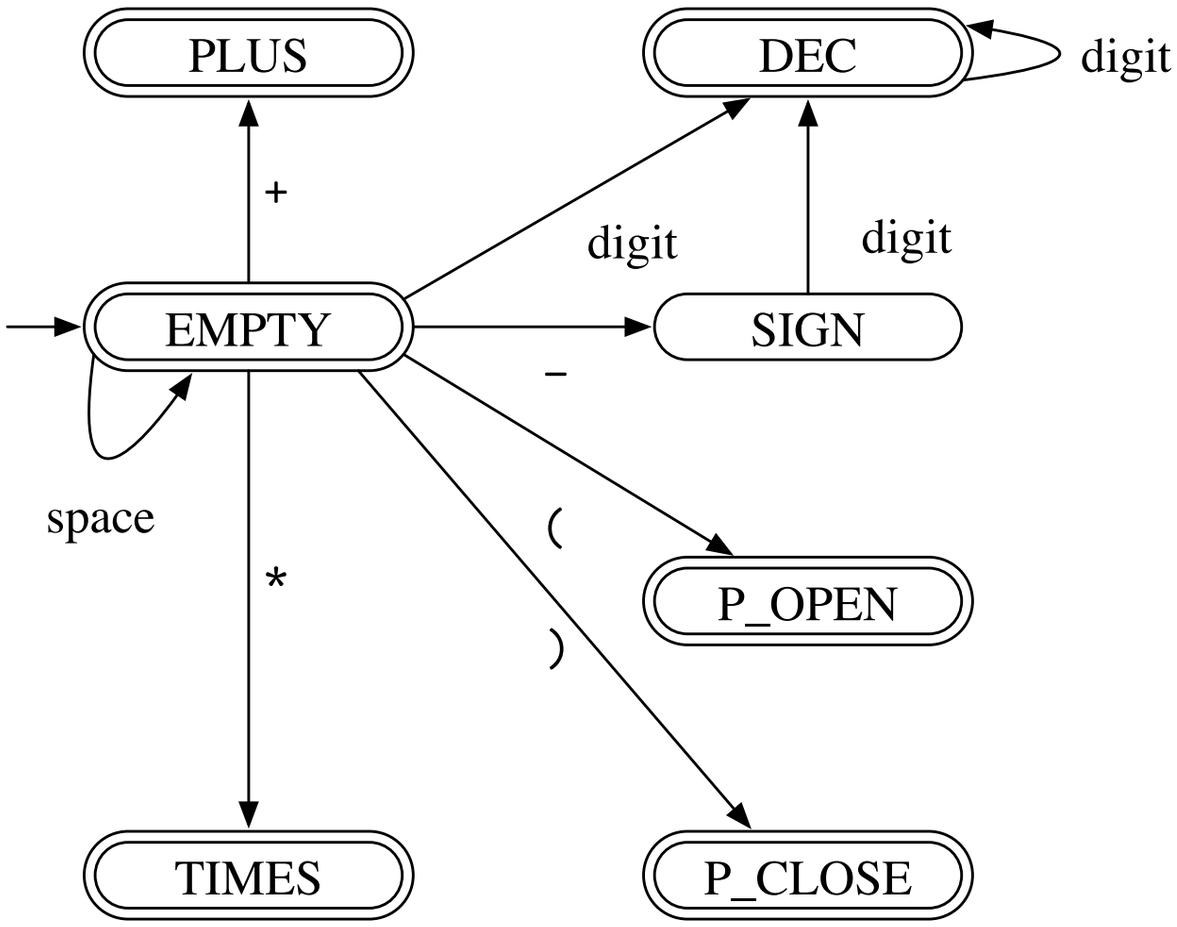
$S =$ <E>

(b) An expanded expression grammar.

<E> → <T> [PLUS <E>]

means

<E> produces <T> followed by zero or one occurrence of PLUS <E>.



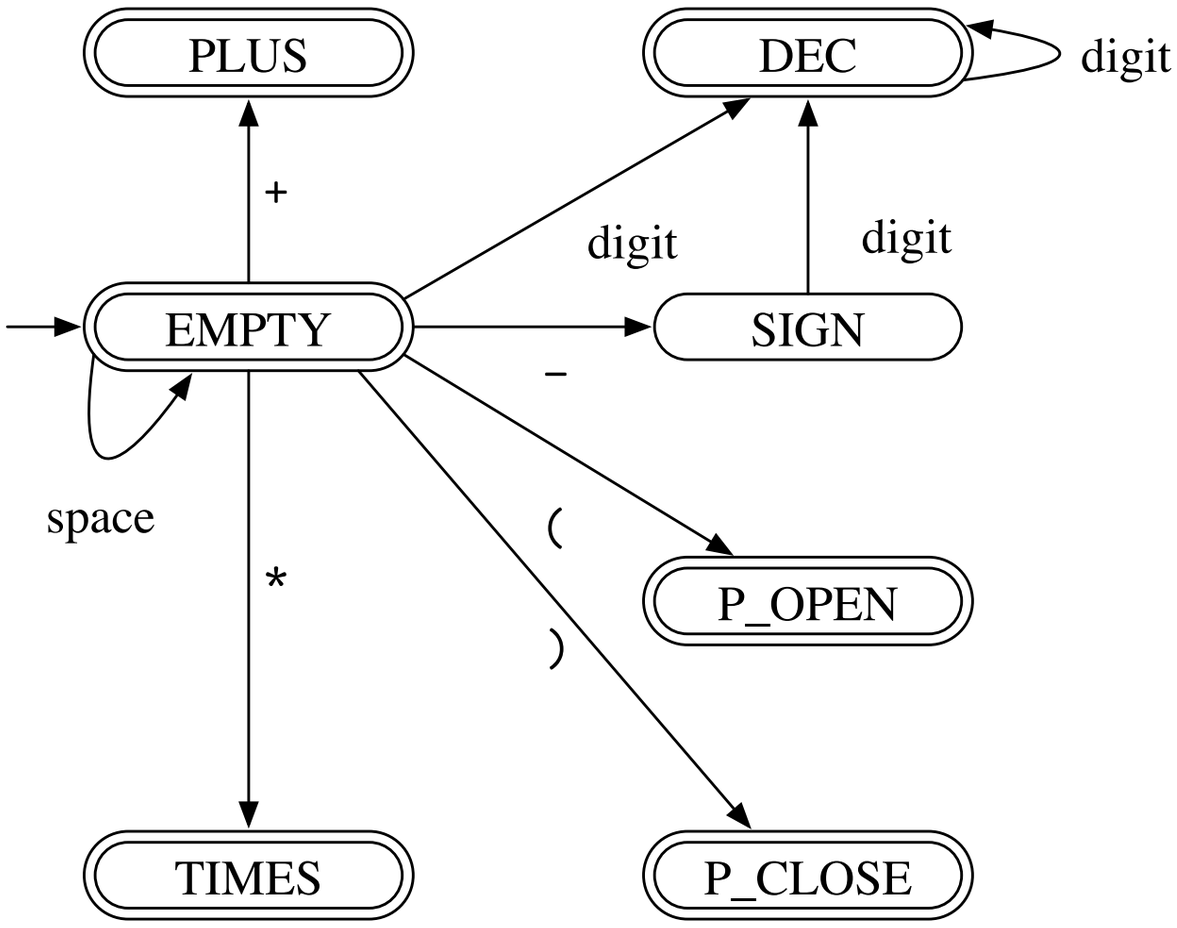
(a) A FSM of a multi-token lexical analyzer.

$N =$ { <E>, <T>, <F> }
$T =$ { PLUS, TIMES, P_OPEN, P_CLOSE, DECIMAL }
$P =$ the productions
1. <E> → <T> [PLUS <E>]
2. <T> → <F> [TIMES <T>]
3. <F> → P_OPEN <E> P_CLOSE
4. <F> → DECIMAL

$S =$ <E>

(b) An expanded expression grammar.

Not a left-recursive grammar.
Can be parsed by an LL parser.



(a) A FSM of a multi-token lexical analyzer.

$N =$ { <E>, <T>, <F> }
$T =$ { PLUS, TIMES, P_OPEN, P_CLOSE, DECIMAL }
$P =$ the productions
1. <E> → <T> [PLUS <E>]
2. <T> → <F> [TIMES <T>]
3. <F> → P_OPEN <E> P_CLOSE
4. <F> → DECIMAL

$S =$ <E>

(b) An expanded expression grammar.

Parse `10 * (-30 + 20)`

The lexer returns DEC TIMES P_OPEN DEC PLUS DEC P_CLOSE

```
class ParserBuffer:
    def __init__(self, producer: TokenProducer):
        self._producer = producer
        self._buffer: List = []

    def peek(self):
        if len(self._buffer) == 0:
            try:
                self._buffer.append(next(self._producer))
            except StopIteration:
                return None
        return self._buffer[0]

def may_match(self, expected_type):
    if (token := self.peek()) and type(token) is expected_type:
        return self._buffer.pop(0)
    return None

def must_match(self, expected_type):
    if ret := self.may_match(expected_type):
        return ret
    raise SyntaxError()
```

`ParserBuffer`

- `peek()` extracts next token from lexical analyzer if no token is currently buffered.
- `may_match()` takes a token type as its argument. If the next token has that type, it removes the token from the buffer and returns it.
- `must_match()` raises a syntax error where `may_match()` returns `None`.

`ParserBuffer`

- `peek()` is called by `may_match()`.
- `may_match()` is called by `must_match()` and the parser.
- `must_match()` is called by the parser.

There are three kinds of arithmetic notation

- Infix: `3 + 5`
- Prefix: `+ 3 5`
- Postfix: `3 5 +`

Postfix notation does not require parentheses

Infix	Postfix
10 * 50	10 50 *

Postfix notation does not require parentheses

Infix	Postfix
`10 * 50`	`10 50 *`
`-30 + 20`	`-30 20 +`

Postfix notation does not require parentheses

Infix	Postfix
10 * 50	10 50 *
-30 + 20	-30 20 +
10 * (-30 + 20)	10 -30 20 + *

Postfix notation does not require parentheses

Infix	Postfix
10 * 50	10 50 *
-30 + 20	-30 20 +
10 * (-30 + 20)	10 -30 20 + *
10 * (-30 + 20) + 70	10 -30 20 + * 70 +

Python starred expressions

```
>>> L1 = [10, 20]
>>> L2 = [30, 40]
>>> print([L1,L2])
>>> [[10, 20], [30, 40]]
>>> print([*L1, *L2])
>>> [10, 20, 30, 40]
```

In function `E()`

```
if times := self._buffer.may_match(tokens.Times):
```

- If `may_match()` returns a token, the `if` statement interprets the token as `True`.
- If `may_match()` returns `None`, the `if` statement interprets `None` as `False`.

```
class ExpressionParser:
    def __init__(self, buffer: io.StringIO):
        self._lexer = Lexer(buffer)
        self._buffer = ParserBuffer(self._lexer)

    # 3. <F> -> ( <E> )
    # 4. <F> -> DECIMAL
    def F(self) -> list[Token]:
        if self._buffer.may_match(tokens.ParenOpen):
            e = self.E()
            self._buffer.must_match(tokens.ParenClose)
            return e
        return [self._buffer.must_match(tokens.Decimal)]
```

```
# 2. <T> -> <F> [* <T>]
def T(self) -> list[Token]:
    f = self.F()
    if times := self._buffer.may_match(tokens.Times):
        t = self.T()
        return [*f, *t, times]
    return f

# 1. <E> -> <T> [+ <E>]
def E(self) -> list[Token]:
    t = self.T()
    if plus := self._buffer.may_match(tokens.Plus):
        e = self.E()
        return [*t, *e, plus]
    return t
```

$N =$ { <E>, <T>, <F> }
$T =$ { PLUS, TIMES, P_OPEN, P_CLOSE, DECIMAL }
$P =$ the productions
1. <E> → <T> [PLUS <E>]
2. <T> → <F> [TIMES <T>]
3. <F> → P_OPEN <E> P_CLOSE
4. <F> → DECIMAL

$S =$ <E>

Function	Next Token	Matched Input
E()	10	
|-T()	10	
| |-F()	10	
| | |-may_match(OpenParen) -> None	10	
| | |-must_match(Decimal) -> 10	*	10
| | |-return [10]	*	10
| |-may_match(Plus) -> None	*	10
| |-return [10]	*	10
|-may_match(Times) -> Times	(	10 *
|-E()	(	10 *
| |-T()	(	10 *
| | |-F()	(	10 *
| | | |-may_match(OpenParen) -> OpenParen	-30	10 * (
| | | |-E()	-30	10 * (
| | | | | T()	-30	10 * (
| | | | | | F()	-30	10 * (
| | | | | | |-may_match(OpenParen) -> None	-30	10 * (
| | | | | | |-must_match(Decimal) -> -30	+	10 * (-30
| | | | | |-return [-30]	+	10 * (-30
| | | | |-may_match(Plus) -> Plus	20	10 * (-30 +
| | | | |-T()	20	10 * (-30 +
| | | | | |-F()	20	10 * (-30 +
| | | | | | |-may_match(OpenParen) -> None	20	10 * (-30 +
| | | | | | |-must_match(Decimal) -> 20	)	10 * (-30 + 20
| | | | | | |-return [20]	)	10 * (-30 + 20
| | | | | |-may_match(Plus) -> None	)	10 * (-30 + 20
| | | | | |-return [20]	)	10 * (-30 + 20
| | | | |-return [-30, 20, +]	)	10 * (-30 + 20
| | | |-may_match(Times) -> None	)	10 * (-30 + 20
| | | |-return [-30, 20, +]	)	10 * (-30 + 20
| | |-must_match(CloseParen) -> CloseParen		10 * (-30 + 20)
| | |-return [-30, 20, +]		10 * (-30 + 20)
| |-may_match(Plus) -> None		10 * (-30 + 20)
| |-return [-30, 20, +]		10 * (-30 + 20)
|-may_match(Times) -> None		10 * (-30 + 20)
|-return [-30, 20, +]		10 * (-30 + 20)
|-return [10, -30, 20, +, *]		10 * (-30 + 20)

Input

```
10 * (-30 + 20)
```

Output

```
10 -30 20 + *
```

Symbol tables

Symbol tables

A symbol table is a dictionary that allows a parser to record symbol names and usage details.

Symbol tables

A symbol table is a dictionary that allows a parser to record symbol names and usage details.

Each symbol entry records the following:

Symbol tables

A symbol table is a dictionary that allows a parser to record symbol names and usage details.

Each symbol entry records the following:

- A string `name`, which holds the name of the symbol

Symbol tables

A symbol table is a dictionary that allows a parser to record symbol names and usage details.

Each symbol entry records the following:

- A string `name`, which holds the name of the symbol
- An integer `definition_count`, which holds the number of times the symbol is defined

Symbol tables

A symbol table is a dictionary that allows a parser to record symbol names and usage details.

Each symbol entry records the following:

- A string `name`, which holds the name of the symbol
- An integer `definition_count`, which holds the number of times the symbol is defined
- An integer `value`, which holds an address assigned by the code generator

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i            ;push #number
          @DECI    number,s       ;scanf("%d", &number)
if:       LDWA     number,s       ;if (number < 0)
          BRGE     endIf
          LDWA     number,s       ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s       ;printf("%d", number)
          ADDSP    2,i            ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		0

reference()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		0
number		1

define()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1

define()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1

reference()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1

define()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1

reference()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i            ;push #number
          @DECI    number,s       ;scanf("%d", &number)
if:       LDWA     number,s       ;if (number < 0)
          BRGE     endIf
          LDWA     number,s       ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s       ;printf("%d", number)
          ADDSP    2,i            ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1
endif		0

reference()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i            ;push #number
          @DECI    number,s       ;scanf("%d", &number)
if:       LDWA     number,s       ;if (number < 0)
          BRGE     endIf
          LDWA     number,s       ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s       ;printf("%d", number)
          ADDSP    2,i            ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1
endif		0

reference()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1
endif		0

`reference()`

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1
endif		1

define()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i           ;push #number
          @DECI    number,s      ;scanf("%d", &number)
if:       LDWA     number,s      ;if (number < 0)
          BRGE     endIf
          LDWA     number,s      ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s      ;printf("%d", number)
          ADDSP    2,i           ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
main		1
number		1
if		1
endif		1

reference()

Source program

```
          BR       main
number:   .EQUATE  0              ;local variable #2d
;
main:     SUBSP    2,i            ;push #number
          @DECI    number,s       ;scanf("%d", &number)
if:       LDWA     number,s       ;if (number < 0)
          BRGE     endIf
          LDWA     number,s       ;number = -number
          NEGA
          STWA     number,s
endIf:    @DECO    number,s       ;printf("%d", number)
          ADDSP    2,i            ;pop #number
          RET
```

Symbol table

Symbol name	Symbol value	Definition count
`main`	0003	1
`number`	0000	1
`if`	000C	1
`endif`	0019	1

Code generation

```
class SymbolEntry:
    def __init__(self, name: str):
        self.name: str = name
        self.definition_count: int = 0
        self.value: int | None = None

    def is_undefined(self):
        return self.definition_count == 0

    def is_multiply_defined(self):
        return self.definition_count > 1
```

```
class SymbolTable:
    def __init__(self) -> None:
        self._table: Dict[str, SymbolEntry] = {}

    def reference(self, name: str) -> SymbolEntry:
        if name not in self._table:
            self._table[name] = SymbolEntry(name)
        return self._table[name]

    def define(self, name: str) -> SymbolEntry:
        (sym := self.reference(name)).definition_count += 1
        return sym

    def __contains__(self, name: str) -> bool:
        return name in self._table

    def __getitem__(self, name: str):
        return self._table[name]
```

Intermediate representation (IR)

- Output from the parser
- Input to the code generator

The `Protocol` hierarchy

The `Protocol` hierarchy

- Defines a contract based on method signatures rather than class inheritance

The `Protocol` hierarchy

- Defines a contract based on method signatures rather than class inheritance
- Any class that has the required methods automatically implements the protocol.

The `Protocol` hierarchy

```
@runtime_checkable
class IRLine(Protocol):
    def source(self) -> str: ...
```

```
@runtime_checkable
class GeneratesObjectCode(IRLine, Protocol):
    memory_address: int | None

    def object_code(self) -> bytearray: ...
    def __len__(self) -> int: ...
```

The `Protocol` hierarchy

```
@runtime_checkable
class IRLine(Protocol):
    def source(self) -> str: ...
```

Ellipsis
This method must exist with this signature

```
@runtime_checkable
class GeneratesObjectCode(IRLine, Protocol):
    memory_address: int | None

    def object_code(self) -> bytearray: ...
    def __len__(self) -> int: ...
```

The `Protocol` hierarchy

- `IRLine`
  Implements `source()`
  Source line example:

```
;******* main()
```

The `Protocol` hierarchy

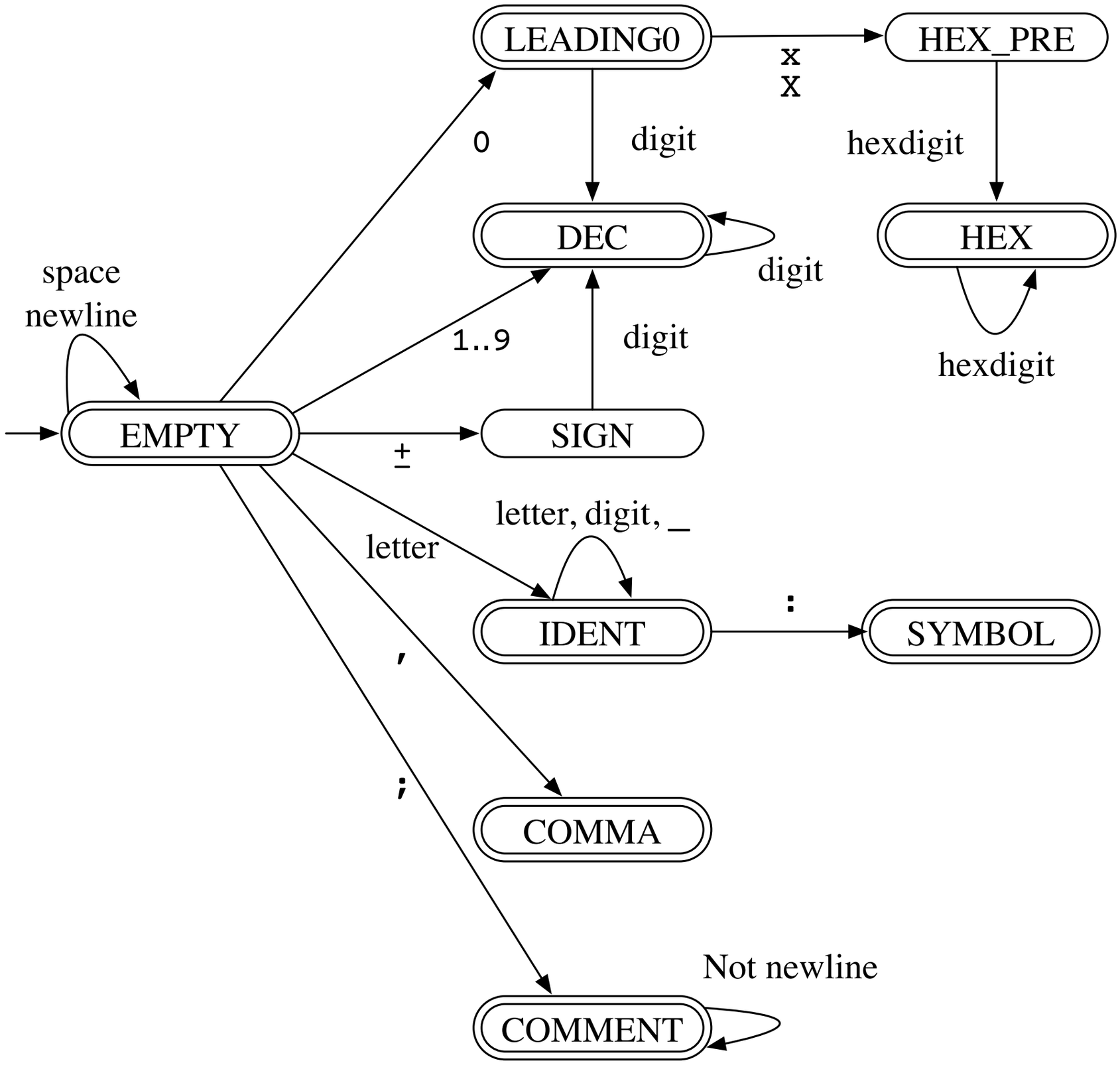
- `IRLine`
  Implements `source()`
  Source line example:
  `;******* main()`
- `GeneratesObjectCode`
  Additionally implements `object_code()`, `__len__()`, and has a `memory_address` field
  Source line example:
  `for2:    CPWA    numPts,d    ;j <= numPts`

The `Protocol` hierarchy

```
@dataclass
class ErrorLine:
    comment: str | None = None


@dataclass
class EmptyLine:


class CommentLine:
    comment: str


@dataclass
class DyadicLine:
    mnemonic: str
    operand_spec: OperandType
    addressing_mode: AddressingMode
    symbol_decl: SymbolEntry | None = None
    comment: str | None = None
    memory_address: int | None = None
```

Parser

- Input: Stream of tokens from the lexer
- Output: A Python list of IR lines, one line for each line of source code


LEADING0
HEX_PRE
x
X
0
digit
hexdigit
DEC
HEX
digit
space
newline
1..9
digit
hexdigit
EMPTY
SIGN
±
letter, digit, _
letter
IDENT
:
SYMBOL
'
;
COMMA
Not newline
COMMENT

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions

1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

COMMENT EMPTY
: A line containing only a comment.

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

COMMENT EMPTY
: A line containing only a comment.

[SYMBOL] <line> EMPTY
: A line containing a dyadic instruction.

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

COMMENT EMPTY
: A line containing only a comment.

[SYMBOL] <line> EMPTY
: A line containing a dyadic instruction.

EMPTY
: A line containing only spaces and a newline.

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

```
for2:    CPWA    numPts,d    ;j <= numPts
```

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

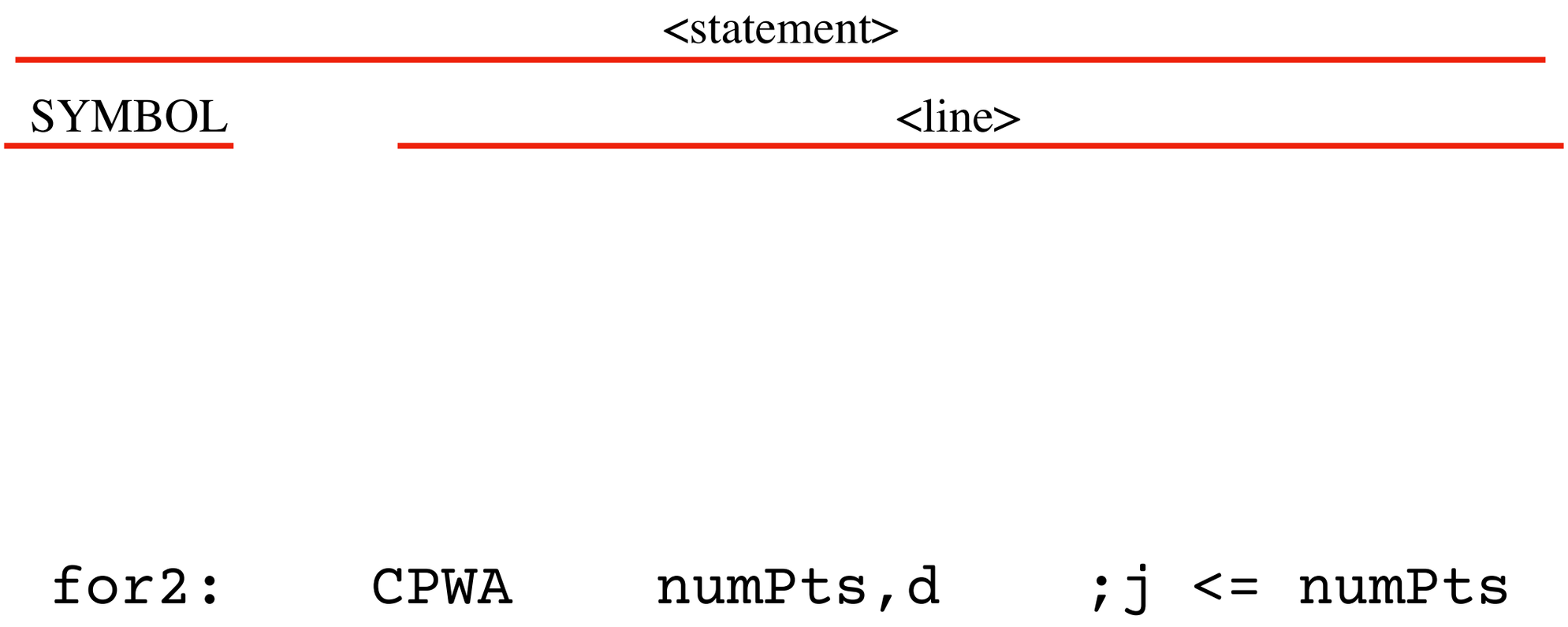
<statement>

```
for2:    CPWA    numPts,d    ;j <= numPts
```

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

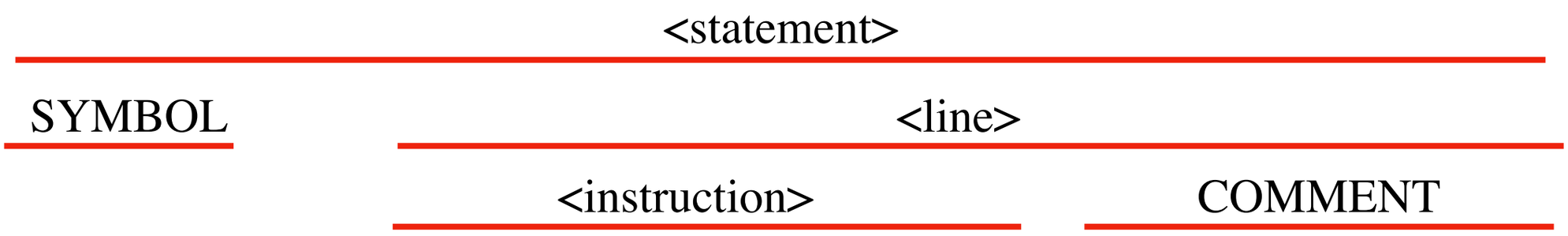



```
for2:    CPWA    numPts,d    ;j <= numPts
```

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>



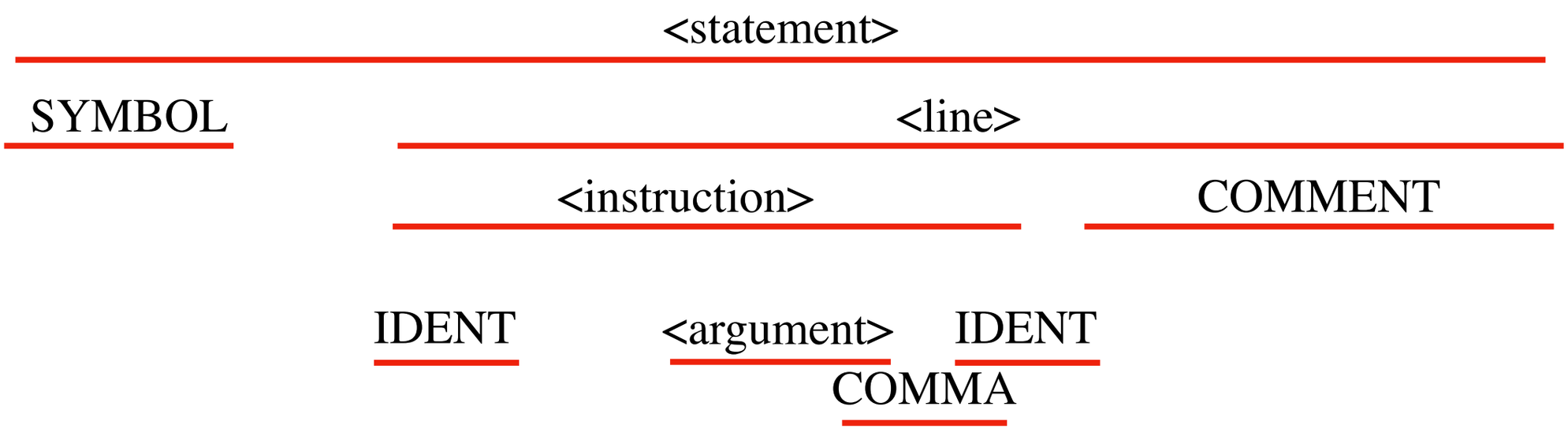

```
for2:    CPWA    numPts,d    ;j <= numPts
```

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions

1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>



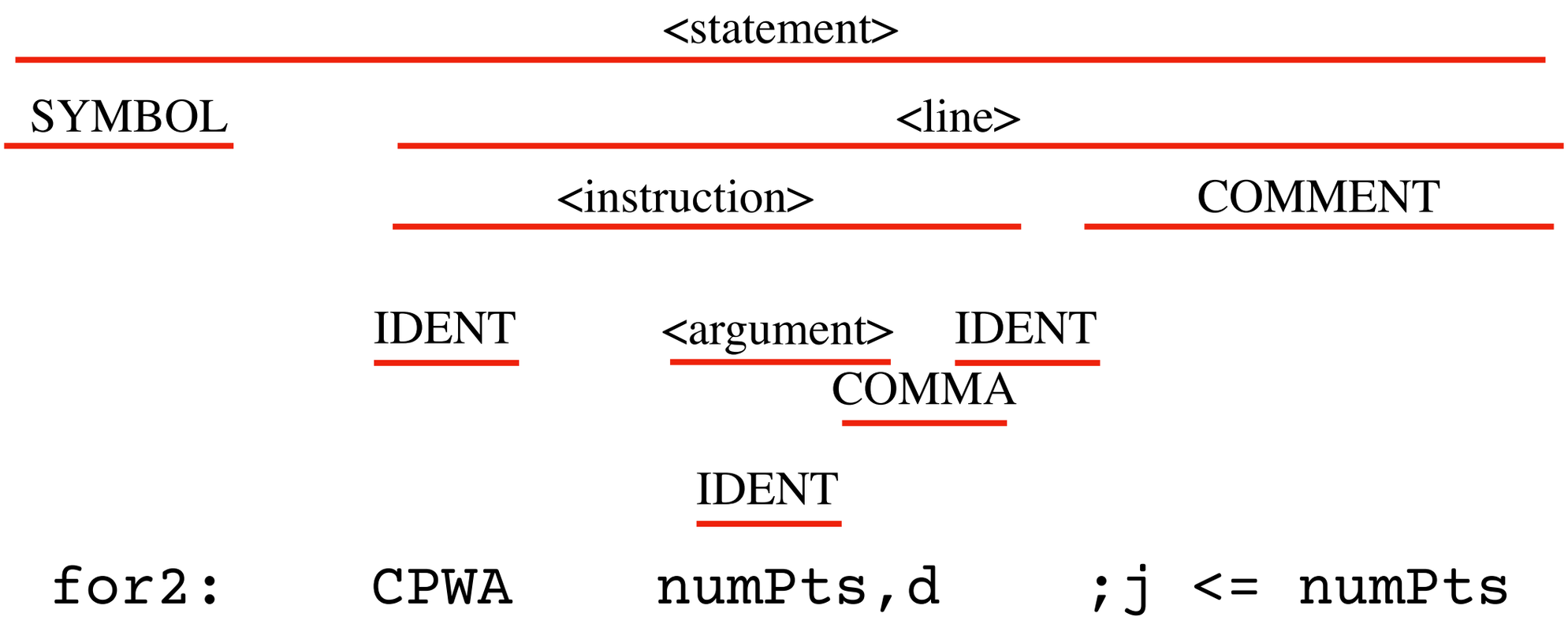

```
for2:   CPWA    numPts,d    ;j <= numPts
```

A grammar for a subset of Pep/10 assembly language

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions

1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>



```
class Parser:
    def __init__(
        self, buffer: io.StringIO, symbol_table: SymbolTable | None = None
    ):
        self.lexer = Lexer(buffer)
        self._buffer = ParserBuffer(self.lexer)
        self.symbol_table = symbol_table if symbol_table else SymbolTable()

    def __iter__(self):
        return self

    def __next__(self) -> IRLine:
        if self._buffer.peek() is None:
            raise StopIteration()
        try:
            return self.statement()
        except SyntaxError as s:
            self._buffer.skip_to_next_line({tokens.Empty})
            return ErrorLine(comment=s.msg if s.msg else None)
        except KeyError:
            self._buffer.skip_to_next_line({tokens.Empty})
            return ErrorLine()
```

```
class Parser:
    def __init__(
        self, buffer: io.StringIO, symbol_table: SymbolTable | None = None
    ):
        self.lexer = Lexer(buffer)
        self._buffer = ParserBuffer(self.lexer)
        self.symbol_table = symbol_table if symbol_table else SymbolTable()

    def __iter__(self):
        return self

    def __next__(self) -> IRLine:
        if self._buffer.peek() is None:
            raise StopIteration()
        try:
            return self.statement()
        except SyntaxError as s:
            self._buffer.skip_to_next_line({tokens.Empty})
            return ErrorLine(comment=s.msg if s.msg else None)
        except KeyError:
            self._buffer.skip_to_next_line({tokens.Empty})
            return ErrorLine()
```

The start symbol

```
def parse(text: str, symbol_table: SymbolTable | None = None) -> List[IRLine]:
    # Ensure input is terminated with a single \n.
    parser = Parser(io.StringIO(text.rstrip() + "\n"), symbol_table)
    return [item for item in parser]
```

In function `E()`

```
if times := self._buffer.may_match(tokens.Times):
```

- If `may_match()` returns a token, the `if` statement interprets the token as `True`.
- If `may_match()` returns `None`, the `if` statement interprets `None` as `False`.

```
# statement ::= [COMMENT  | ([SYMBOL] line)] EMPTY
def statement(self) -> IRLine:
    return_ir: IRLine | None = None
    if self._buffer.may_match(tokens.Empty):
        return EmptyLine()
    elif comment_token := self._buffer.may_match(tokens.Comment):
        return_ir = CommentLine(comment_token.value)
    elif symbol_token := self._buffer.may_match(tokens.Symbol):
        symbol_entry = self.symbol_table.define(symbol_token.value)
        if not (return_ir := self.line(symbol_entry)):
            message = "Symbol declaration must be followed by instruction"
            raise SyntaxError(message)
    elif not (return_ir := self.line(None)):
        raise SyntaxError("Failed to parse line")

    self._buffer.must_match(tokens.Empty)
    return return_ir
```

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

```
# line ::= instruction [COMMENT]
def line(self, symbol_entry: SymbolEntry | None) -> DyadicLine | None:
    return_ir = self.instruction(symbol_entry)
    if not return_ir:
        return None
    elif comment := self._buffer.may_match(tokens.Comment):
        return_ir.comment = comment.value
    return return_ir
```

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

```
# instruction ::= IDENT argument COMMA IDENT
def instruction(
    self, symbol_entry: SymbolEntry | None
) -> DyadicLine | None:
    if not (mn_token := self._buffer.may_match(tokens.Identifier)):
        return None
    mn_str = mn_token.value.upper()
    if mn_str not in INSTRUCTION_TYPES:
        raise SyntaxError(f"Unrecognized mnemonic: {mn_str}")
    elif not (arg_ir := self.argument()):
        raise SyntaxError(f"Missing argument")

    try:
        arg_int = int(arg_ir)
        arg_int.to_bytes(2, signed=arg_int < 0)
    except OverflowError:
        raise SyntaxError("Number too large")
```

Continues

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

Continued

```
self._buffer.must_match(tokens.Comma)
addr_str = self._buffer.must_match(tokens.Identifier).value.upper()
try:
    # Check if addressing mode is valid for this mnemonic
    addr_mode = cast(AddressingMode, AddressingMode[addr_str])
    mn_type = INSTRUCTION_TYPES[mn_str]
    if not mn_type.allows_addressing_mode(addr_mode):
        err = f"Invalid addressing mode {addr_str} for {mn_str}"
        raise SyntaxError(err)
except KeyError:
    raise SyntaxError(f"Unknown addressing mode: {addr_str}")
return DyadicLine(mn_str, arg_ir, addr_mode, symbol_decl=symbol_entry)
```

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

```
# argument ::= HEX | DEC | IDENT
def argument(self) -> OperandType | None:
    if hex_token := self._buffer.may_match(tokens.Hexadecimal):
        return operands.Hexadecimal(hex_token.value)
    elif dec_token := self._buffer.may_match(tokens.Decimal):
        return operands.Decimal(dec_token.value)
    elif ident_token := self._buffer.may_match(tokens.Identifier):
        symbol_entry = self.symbol_table.reference(ident_token.value)
        return operands.Identifier(symbol_entry)
    return None
```

$N =$ { <argument>, <instruction>, <line>, <statement> }
$T =$ { HEX, DEC, COMMA, EMPTY, IDENT, SYMBOL, COMMENT }
$P =$ the productions
1. <argument> → HEX | DEC | IDENT
2. <instruction> → IDENT <argument> COMMA IDENT
3. <line> → <instruction> [COMMENT]
4. <statement> → [COMMENT | ([SYMBOL] <line>)] EMPTY

$S =$ <statement>

Pep/10 Source Code

```
start:    BR       main
; SUBX,i is ASCII 'h'
h:        SUBX     0xFEED,i

main:     LDBA     h,d
          STBA     charOut,d
          ADDA     1,i          ;Increment accumulator to convert h to i
        StBa  charOut   ,  D
          STBA     pwrOff,d
```

Intermediate Representation

```
DyadicLine('start:BR main,i')
CommentLine(' SUBX,i is ASCII \'h\'')
DyadicLine('h:SUBX 0xfeed,i')
EmptyLine()
DyadicLine('main:LDBA h,d')
DyadicLine('STBA charOut,d')
DyadicLine('ADDA 1,i',comment='Increment accumulator to convert h to i')
DyadicLine('STBA charOut,d')
DyadicLine('STBA pwrOff,d')
```

Pep/10 Source Code

```
start: ;BR    main,i
       LDWY   0x10000,i
       ADDA   0x10000,i
       SUBSP  15,y
       STWA   0,i
```

Intermediate Representation

```
ErrorLine(';ERROR: Symbol declaration must be followed by instruction')
ErrorLine(';ERROR: Unrecognized mnemonic: LDWY')
ErrorLine(';ERROR: Number too large')
ErrorLine(';ERROR: Unknown addressing mode: Y')
ErrorLine(';ERROR: Invalid addressing mode I for STWA')
```