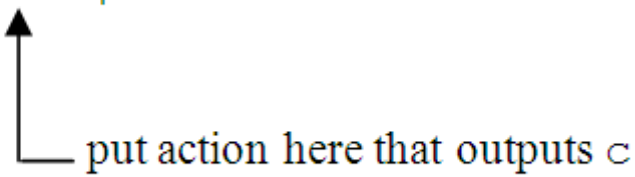
Chapter 10

Recursive-descent Translation

Translation grammar

A grammar in which actions are embedded.

```
S : "b" C "d"
C : "c" C | λ
```



Use Java-like syntax for translation grammars

G10.2

```
void S(): {}
{
   "b" C() "d"
}

void C(): {}
{
    "c" {System.out.print('c');} C()
  |
   {}
}
```

← use `S()` in place of `S` and : in place of →

← use `C()` in place of `C`

← use `C()` in place of `C`

← the vertical bar separates alternatives

← empty action represents λ

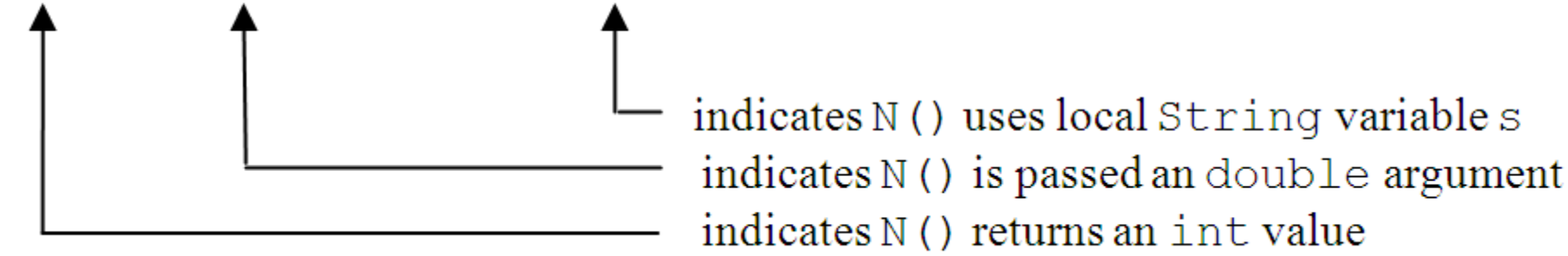
Corresponding Java code

```
private void S()
{
  consume('b');
  C();
  consume('d');
}
//-------------------------------------------------------
private void C()
{
  switch(currentToken)
  {
    case 'c':
      // apply first C production
      consume('c');
      System.out.print('c');     // action
      C();
      break;
    case 'd':
      // apply second C production
      ;
      break;
    default:
      throw new RuntimeException("\"c\" or \"d\"");
  }
}
```

Features of translation grammars

- Represent nonterminal with method call
- Separate parts of right side with spaces
- Use ":" in place of "->"
- Enclose terminals in quotes
- Use Java-like syntax:

```
int N(double d) : {String s;}
```



More features

Group together right sides with same left side:

```
void B() : {}
{
    C() D()
  |
    D() E()
  |
    "b"
}
```

More features

```
void S(): {}
{
     "b" Q()
  |
       {}
}
void Q(): {}
{
     "c"
  |
    {System.out.println("bye");}
}
```

action alone represents λ

action alone represents λ

More features

Omit semicolon at end of method call but not withing actions:

Nonterminal `S` represented with

```
S()
```

Action includes semicolon:

```
{System.out.println("bye");}
```

More features

Token categories enclosed in angle brackets:

<UNSIGNED>

More features

Can use star, plus, and question mark operators:

```
void list(); {}
{
    "b" ("b")*
}
```

More features

Can use Java-style comments:

```
/*  ...
*/
//  ...
```

More features

Can use alternation operator:

```
void S(): {}
{
    {System.out.println("hello");
    "b"
  |
    "c"
    {System.out.println("goodbye");
}
```

Translate prefix to postfix

```
void S():  {}                                          Selection Set
{
   expr() {System.out.println();}    {"+", "-", "*",
                                      "/", "b", "c", "d"}
}

void expr()  : {}
{
   "+" expr() expr() {System.out.print('+');}    {"+"}
 |
   "-" expr() expr() {System.out.print('-');}    {"-"}
 |
   "*" expr() expr() {System.out.print('*');}    {"*"}
 |
   "/" expr() expr() {System.out.print('/');}    {"/"}
 |
   "b" {System.out.print('b');}                  {"b"}
 |
   "c" {System.out.print('c');}                  {"c"}
 |
   "d" {System.out.print('d');}                  {"d"}
}
```

Translate prefix to postfix code

Fig1004.txt

L-attributed grammar

Translation with no forward dependencies.

New token manager

```
class Token
{
  // integer that identifies kind (i.e., type) of token
  public int kind;

  // location of token in source program
  public int beginLine, beginColumn, endLine, endColumn;

  // String consisting of characters that make up token
  public String image;

  // link to next Token object
  public Token next;
}
```

Tokenizing prefix expression

```
+ 17 * 2 5
```

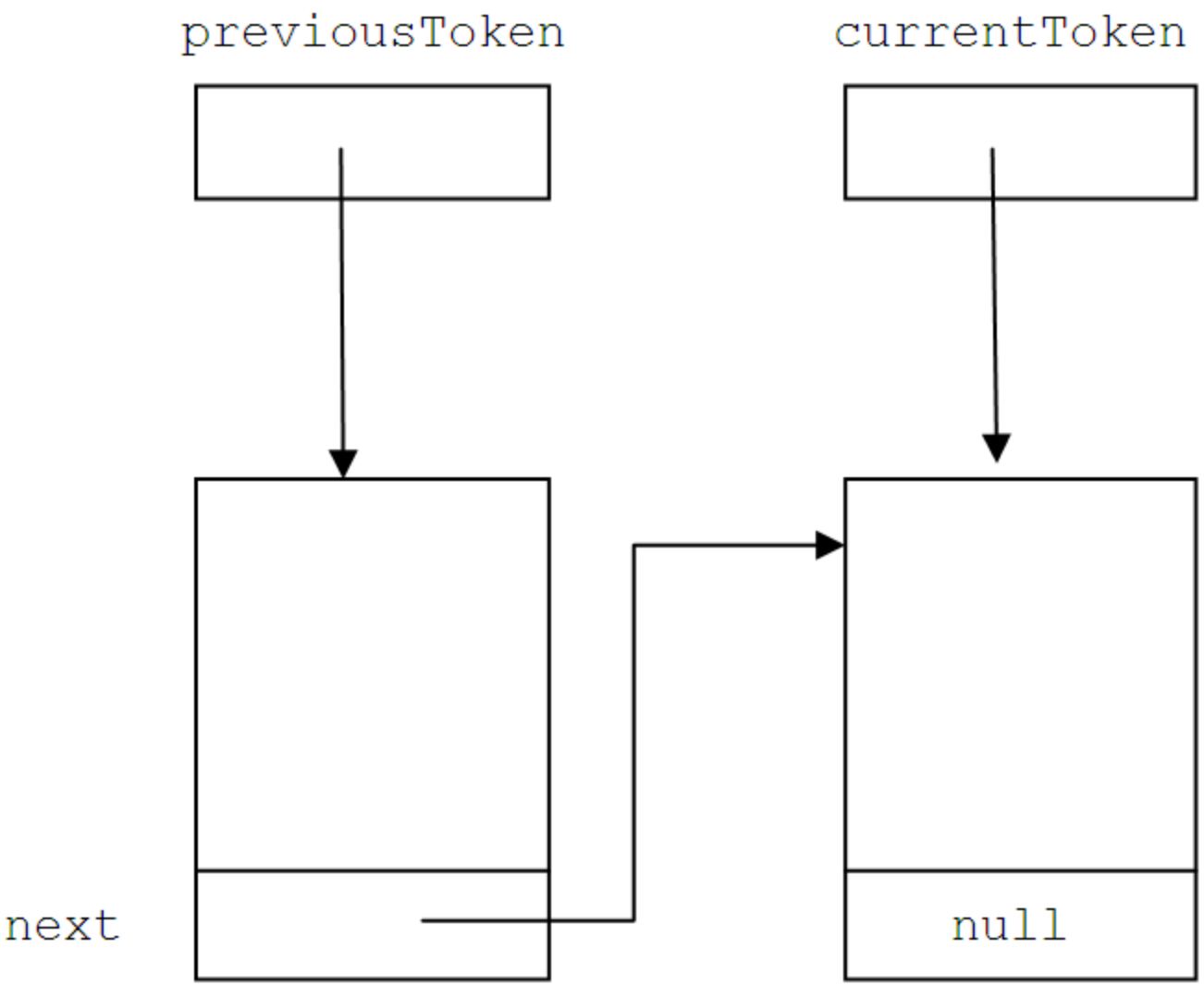
token	+	17	*	2	5	EOF
kind	2	1	4	1	1	0
beginLine	1	1	1	1	1	1
beginColumn	1	3	6	8	10	11
endLine	1	1	1	1	1	1
endColumn	1	4	6	8	10	11
image	"+"	"17"	"*"	"2"	"5"	"<EOF>"

Solving lookahead problem

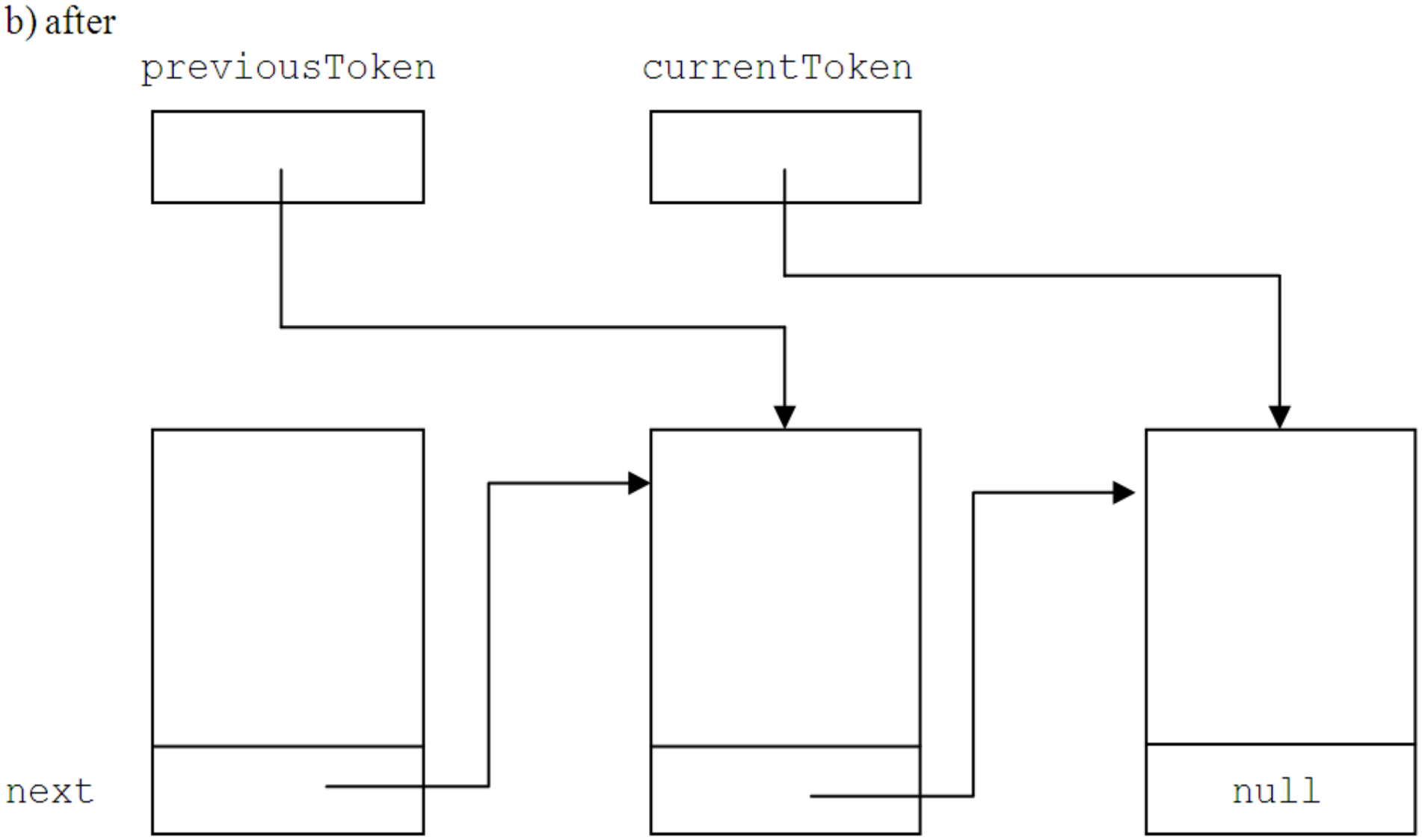
For all tokens, read on character beyond the end of the token.

Token chain

a) before



Token chain



Token trace

```
kd=   2 bL=   1 bC=   1 eL=   1 eC=   1 im= +
kd=   1 bL=   1 bC=   3 eL=   1 eC=   4 im= 17
kd=   4 bL=   1 bC=   6 eL=   1 eC=   6 im= *
kd=   1 bL=   1 bC=   8 eL=   1 eC=   8 im= 2
kd=   1 bL=   1 bC=  10 eL=   1 eC=  10 im= 5
kd=   0 bL=   2 bC=   1 eL=   2 eC=   1 im= <EOF>
```

Evaluating prefix expressions

```
void S(): {int p;}                                        Selection Set
{
   p=expr() {System.out.println(p);}                      {"+", "-", "*",
                                                           "/", <UNSIGNED>}
}
//-----------------------------------
int expr() : {int p, q; Token t}
{
   "+" p=expr() q=expr() {return p+q;}                    {"+")
 |
   "-" p=expr() q=expr() {return p-q;}                    {"-"}
 |
   "*" p=expr() q=expr() {return p*q;}                    {"*"}
 |
   "/" p=expr() q=expr() {return p/q;}                    {"/"}
 |
   t=<UNSIGNED>                                           {<UNSIGNED>}
   {p=Integer.parseInt(t.image);}
   {return p;}
}
```

Evaluating prefix expressions

Fig1010.txt

Creating symbol tables

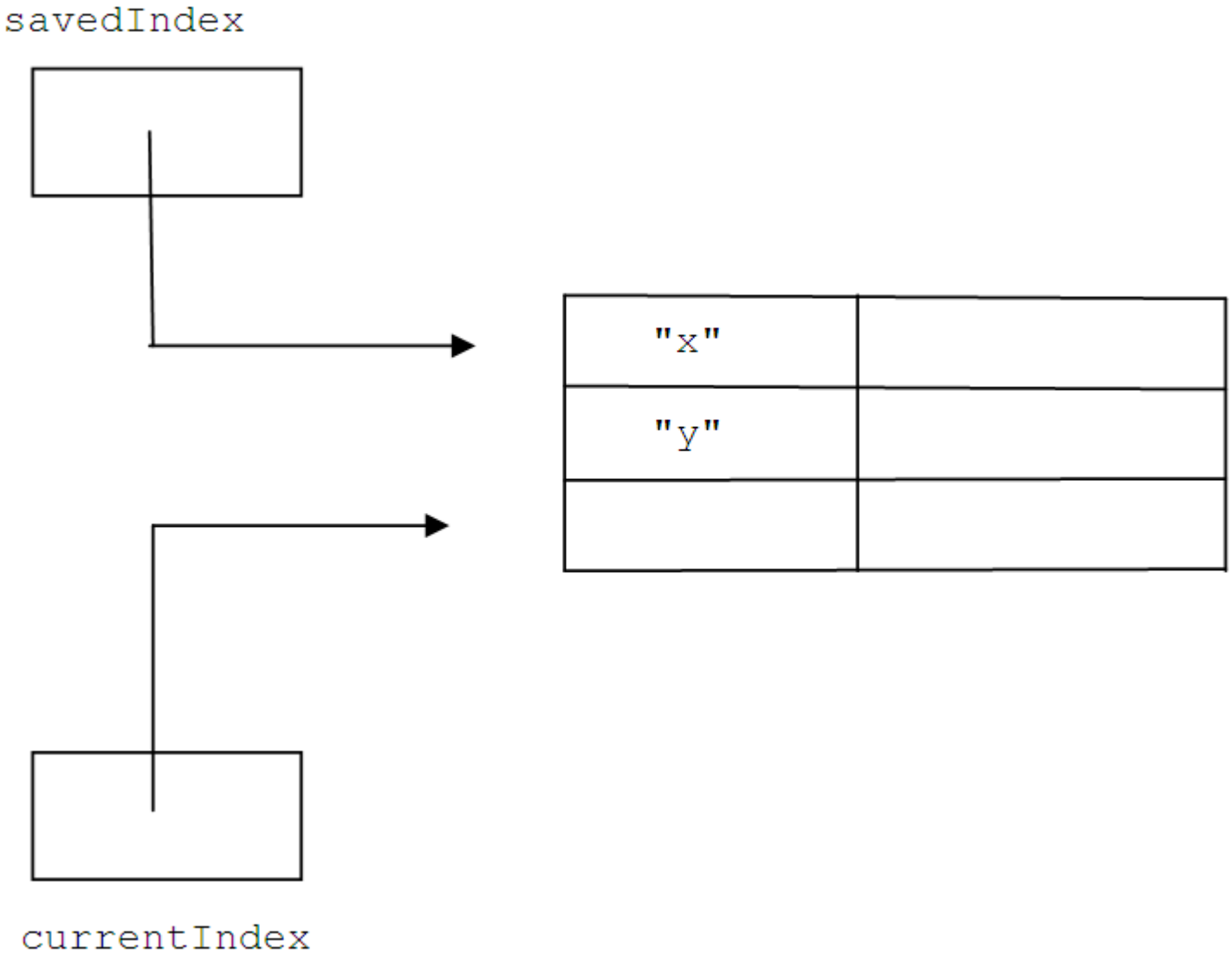
Symbol	Type
"x"	0
"y"	0

What if type follows the list of variables

```
x, y: int;
```

As variables are parsed, what is entered into the symbol table?

Enter only variable name



Use recursion

```
 1 void declaration(): {String p; int q; Token t;}
 2 {
 3     t=<ID>
 4     q=declarationTail()
 5     {symTab.enter(t.image, q);}
 6 }
 7
 8 int declarationTail() : {String p; int q; Token t;}
 9 {
10     ","
11     t=<ID>
12     q=declarationTail()
13     {symTab.enter(t.image,q); return q;}
14   |
15     ":" q=type() ";"
16     {return q;}
17 }
```

Parse tree: recursive approach