$\boxed{17}$ Parsing and Semantic Algorithms

17.1 An Arithmetic Interpreter

We will write a program that interprets a small arithmetic language. An example run is:

```
Please enter your program, ending with a '.'.
Type ^D to quit.

x = 10;
y = 100;
z = x + 5 * y;
.

Value:
510
```

$\boxed{17.2}$ Interpreter (or Compiler) Program Structure

character stream (file)

↓ **lexical analyzer**

lexeme stream

↓ **parser**

parse tree

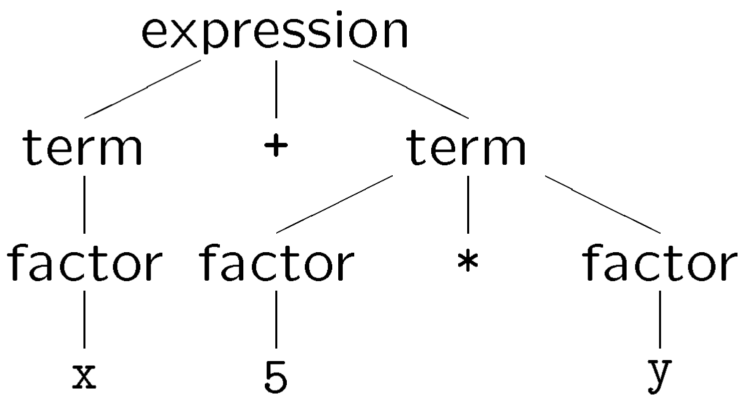
↓ **semantic algorithm**

values (or compiled code)

17.3 Parse Trees

The parser converts streams of lexemes into parse trees:

`x + 5 * y` $\Longrightarrow$



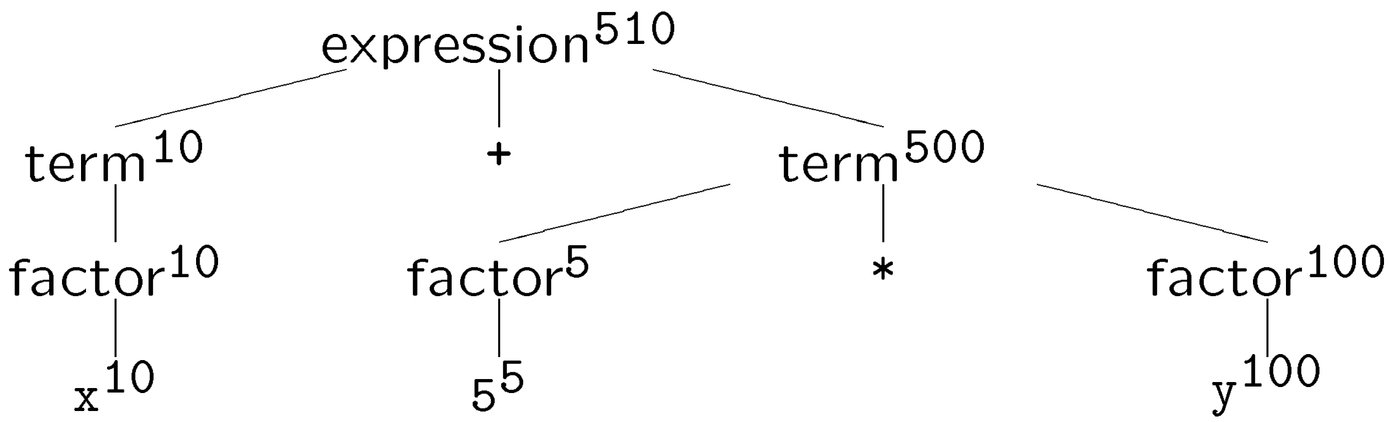
where the grammar is:

expression	::=	term { `+` term }*
term	::=	factor { `*` factor }*
factor	::=	number | symbol

$\boxed{17.4}$ Semantics

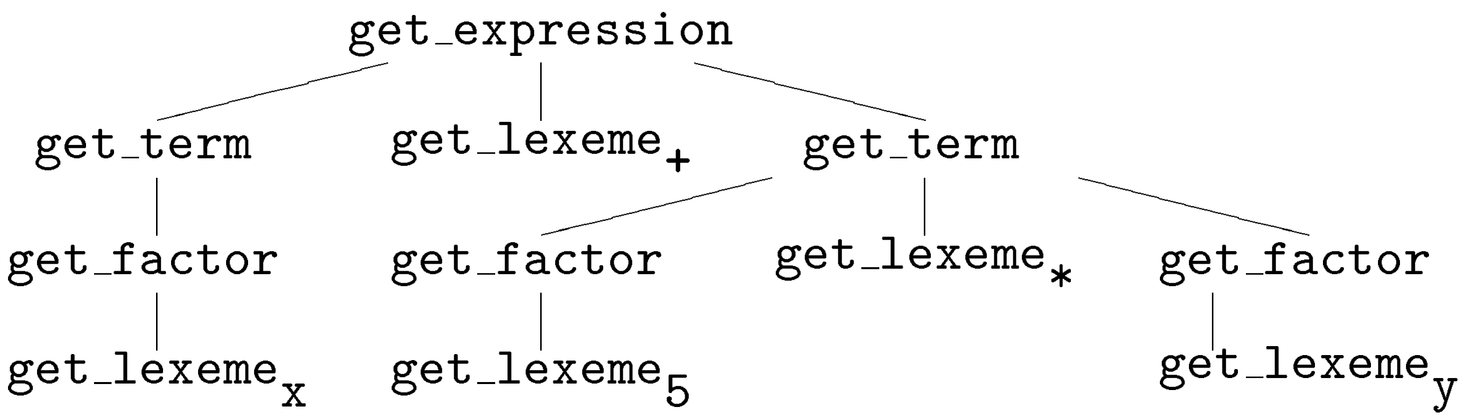
A semantic algorithm may ‘decorate’ a parse tree with values.

For example, if $\quad$ `x = 10` $\quad$ and $\quad$ `y = 100`:



17.5 Recursive Descent Parser

A recursive descent parser is a parser whose call tree has the same structure as the parse tree it produces:



17.6 Recursive Descent Parser Pseudocode

```
get_expression ()
{
  get_term();
  while ( next lexeme == + )
  {
    get_lexeme();
    get_term();
  }
}
```

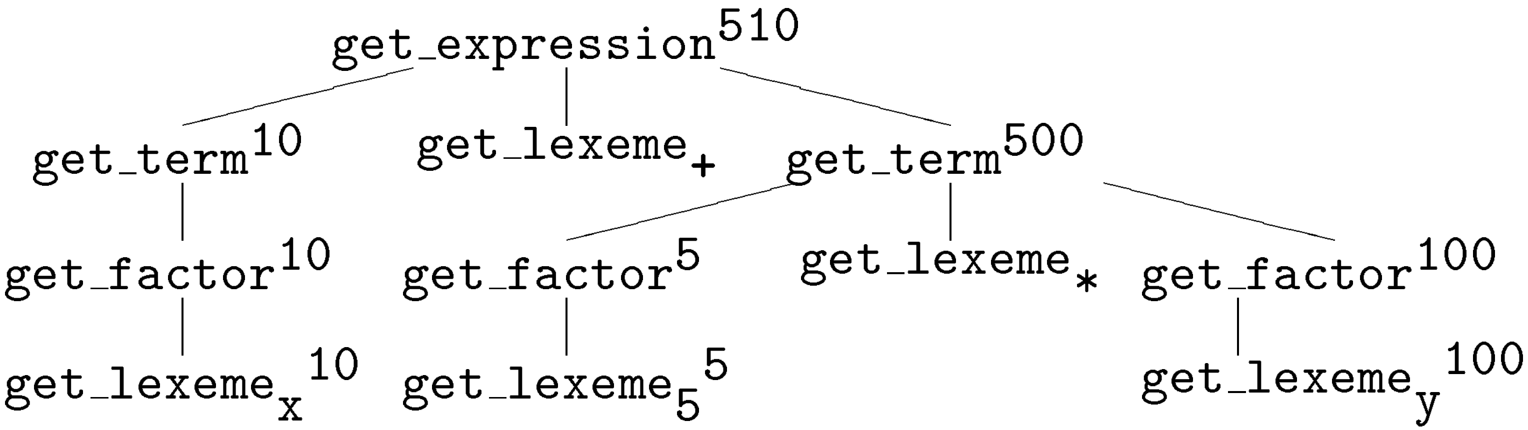
```
get_term ()
{
  get_factor();
  while ( next lexeme == * )
  {
    get_lexeme();
    get_factor();
  }
}
```

```
get_factor ()
{
  get_lexeme();
}
```

The parser must be able to look at the next lexeme.

17.7 Assigning Values with a Recursive Descent Parser

With a recursive descent parser, the value of a tree node becomes the value of a function.



17.8 Pseudocode for Assigning Values

```
avalue get_term ()
{
  avalue v1 =
        get_factor();
  while
    ( next lexeme == * )
  {
    get_lexeme();
    avalue v2 =
          get_factor();
    v1 = v1 * v2;
  }
  return v1;
}
```

```
avalue get_factor ()
{
  avalue v1;
  if ( next lexeme is number )
    v1 = get_lexeme();
  else if
    ( next lexeme is symbol )
  {
    symbol s = get_lexeme();
    if ( s has a value )
      v1 = value of s;
    else error
          ( s has no value )
  }
  return v1;
}
```

17.9 Converting the Interpreter to a Compiler

```
Please enter your program, ending with a '.'.
Type ^D to quit.

x = 10;
        li $t0, 10
        move $s0, $t0
y = 100;
        li $t0, 100
        move $s1, $t0
z = x + 5 * y;
        li $t0, 5
        mul $t1, $t0, $s1
        add $t0, $s0, $t1
        move $s2, $t0
.

Value:
$s2
```

17.10 Pseudocode for +

```
avalue operator + ( avalue x, avalue y)
{
  avalue result = allocate_temporary_register ();

  cout << "\t" << "add "
       << result << ", "
       << x <<< ", "
       << y << ";" << endl;

  free_temporary_register ( x );
  free_temporary_register ( y );

  return result;
}
```

17.11 Register Assignment

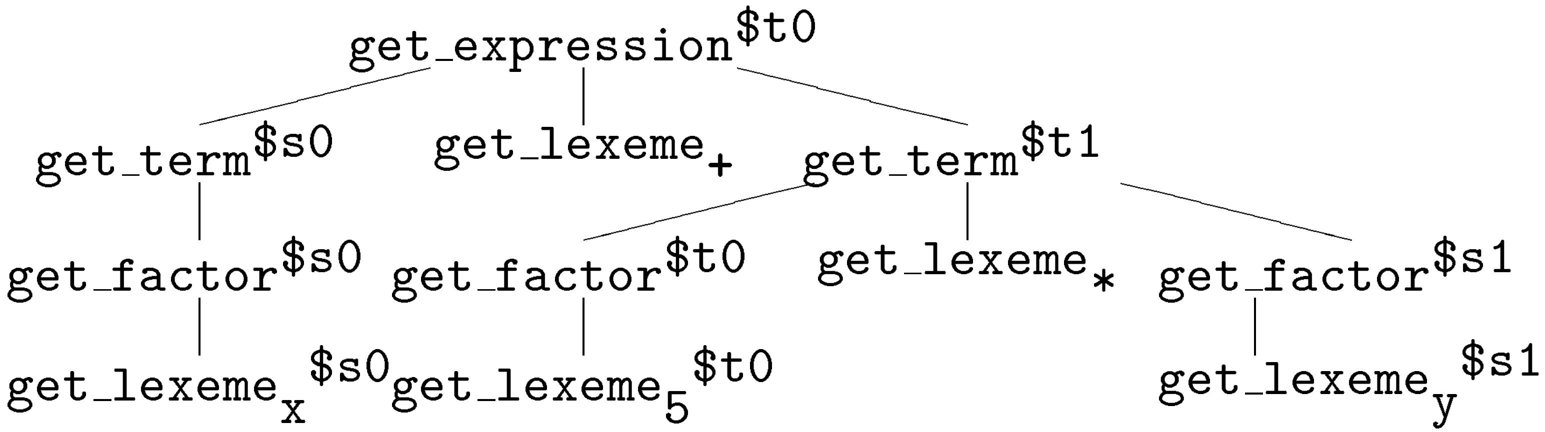
In the compiler, values of type `avalue` hold register names.

The `$s` registers are allocated to the variables in a program. These registers are allocated but never freed.

The `$t` registers are allocated to hold the output of expressions temporarily. These registers are freed as soon as they are used: they can be used only once.

The `free_temporary_register` function frees the given register if and only if that register is a `$t` register,
and not a `$s` register.

17.12 Assigning "Values" during Compilation



17.13 One Lexeme Look-Ahead Parsers

expression	::=	term { `+` term }*	E.g.: `8 x + 9(y + z)x!`
term	::=	factor factor*	
factor	::=	variable | (expression) | factor `!`	

Pseudo-Code:

```
get_term ()
{
  get_factor();
  while ( next lexeme ∈ FIRST (factor) )
  {
    get_lexeme();
    get_factor();
  }
}
```

FIRST (factor) is the set of all lexemes that can be the first lexeme of a factor.

$\boxed{17.14}$ Computing the Set of First Lexemes

The previous grammar gives the set equations:

$$
\begin{array}{ll}
\text{FIRST (expression)} & \supseteq \text{FIRST (term)} \\
\text{FIRST (term)} & \supseteq \text{FIRST (factor)} \\
\text{FIRST (factor)} & \supseteq \text{variable} \\
\text{FIRST (factor)} & \supseteq \{\ (\ \} \\
\text{FIRST (factor)} & \supseteq \text{FIRST (factor)}
\end{array}
$$

which when solved for the <u>smallest</u> solutions gives:

$$
\begin{array}{ll}
\text{FIRST (expression)} & = \text{FIRST (term)} \\
\text{FIRST (term)} & = \text{FIRST (factor)} \\
\text{FIRST (factor)} & = \text{variable} \cup \{\ (\ \} \quad ::= \text{variable} \mid (
\end{array}
$$

Comments:

1. Names appearing in a grammar, like ‘term’, ‘symbol’, and ‘number’, are names of syntax units.

2. Syntax units not defined by a grammar are said to be the terminals of the grammar, and name the leaves of parse trees produced using the grammar. Syntax units defined by the grammar are non-terminals, and name non-leaf nodes of parse trees.

   A lexeme can be used to name a syntax unit containing only that lexeme.

3. The children of a non-leaf parse tree node, taken in left to right order, are an instance of the right side of a grammar equation defining the parse tree node label. Such an instance only has syntax unit names, and not the special characters $|$, $\star$, $\{$, or $\}$.

4. Recursive descent parsers are natural for interpreters and compilers because they provide a way to compute values of parse tree nodes while parsing. The node values become the parsing function (`get_` function) return values.

5. To write a one-lexeme-lookahead parser, one must be able to look ahead one lexeme, or backup over one previously read lexeme.

6. If set theoretic symbols, such as $\cup$, are not available, then one can write out what FIRST (factor) is as a grammar equation.