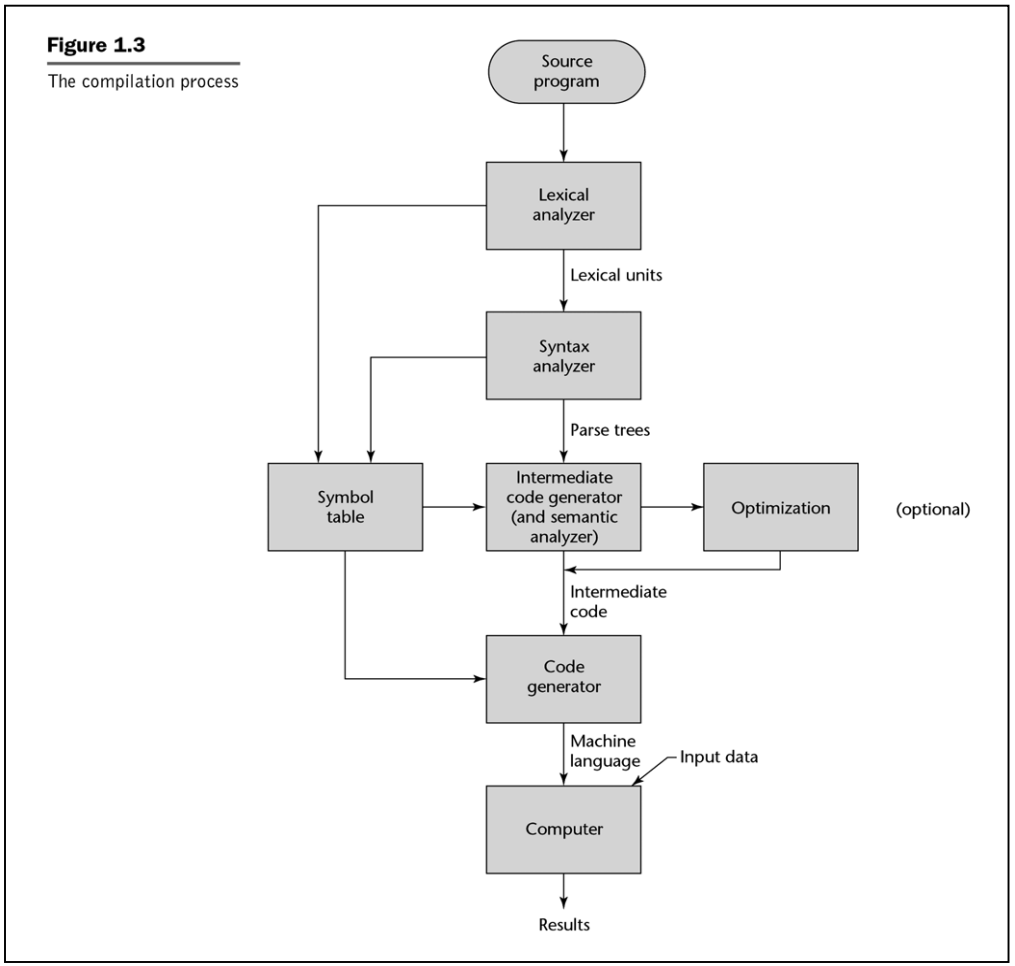
Chapter 3: Describing Syntax and Semantics

- Introduction
- Formal methods of describing syntax (BNF)

We can analyze **syntax** of a computer program on two levels:

1. Lexical level
2. Syntactic level

- Lexical analyzer collect characters into tokens.
- Syntactic analyzer determine syntax structure and determine whether the given programs are syntactically correct.



Derivation

• A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

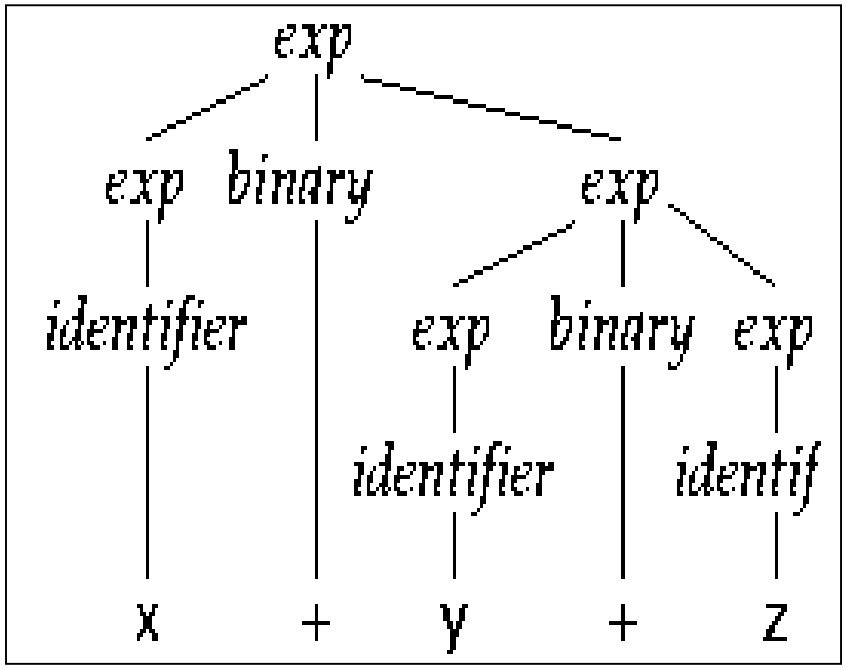
<program> => <stmts>
=> <stmt>
=> <var> = <expr>
=> a =<expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const

<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const

Parse Tree: A hierarchical representation of a derivation

For example **x + y + z** is parsed:

<exp> → <exp> <binary> <exp>
<exp> → <identifier>
| <literal>
| <unary> <exp>
| <exp> <binary> <exp>
<binary> → '<' | '>' | '+' | '-'
<unary> → '-'
<identifier> → x|y|z



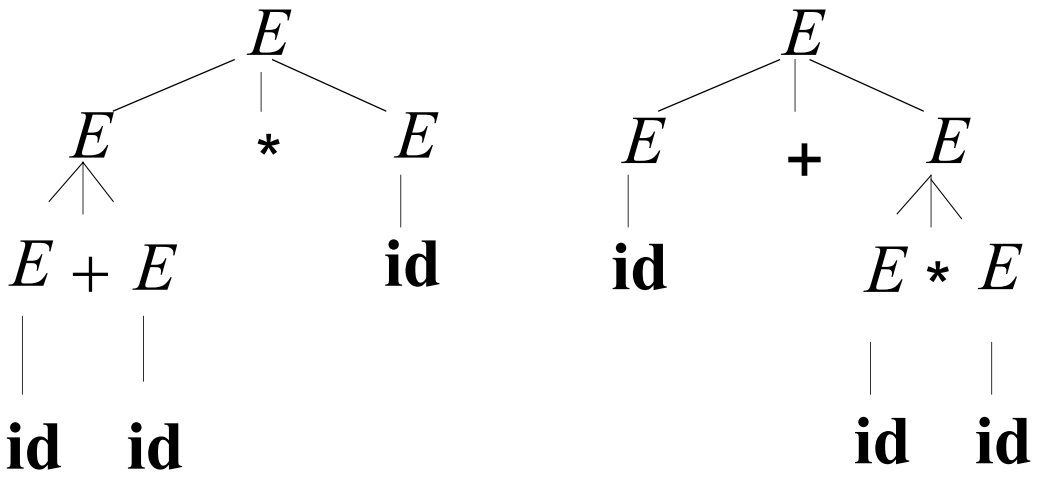
An Ambiguous Expression Grammar

A grammar that can have more than one parse tree generating a particular string is ambiguous.

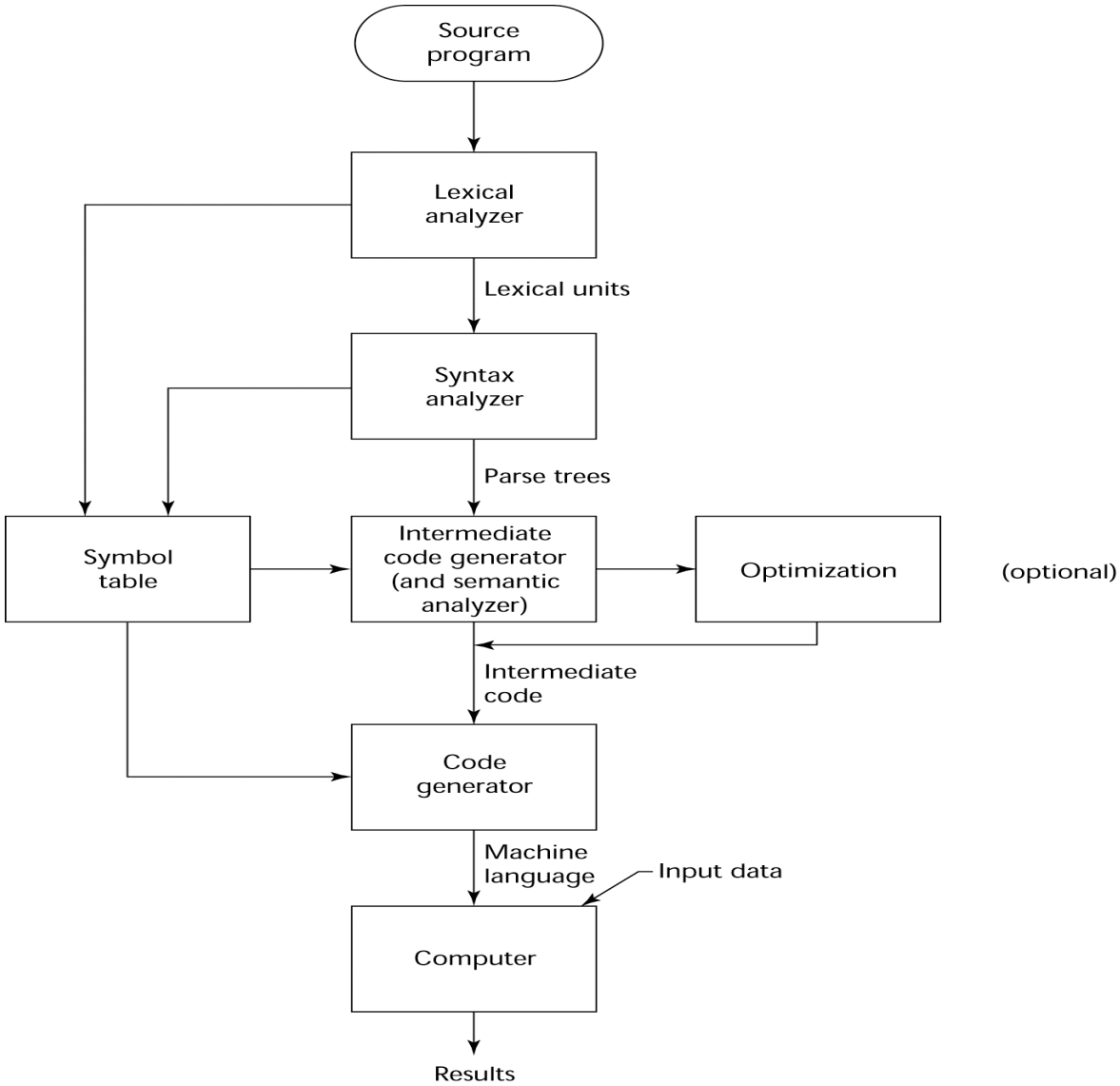
<E> → <E> + <E> | <E> - <E> | <E> * <E> | <E> /

<E> | id

There are two parse trees for the expression id + id * id



Ambiguity is a problem: compiler chooses the code to be generated for a statement by examining its parse tree. If more than one parse tree, the meaning of the structure cannot be unique.

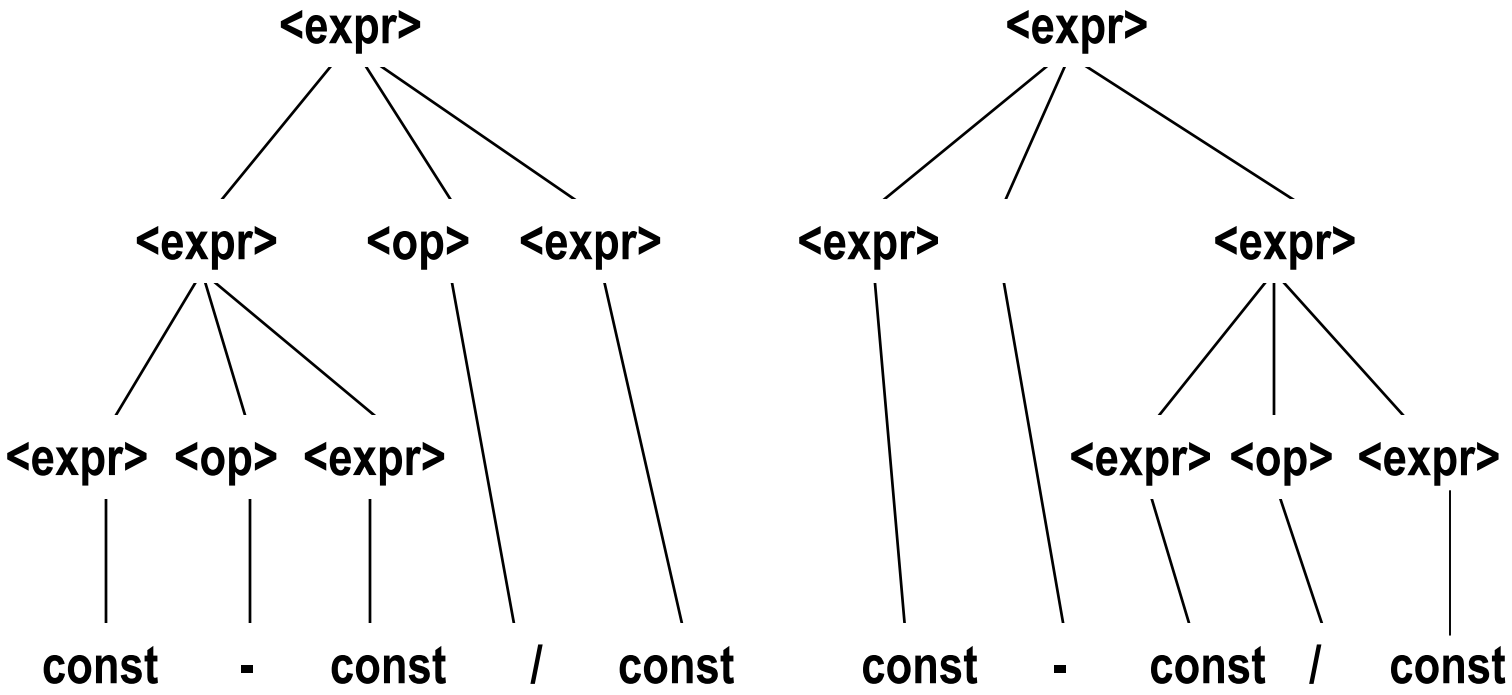


Another example:

<expr> → <expr><op><expr>|const

<op> → /|-

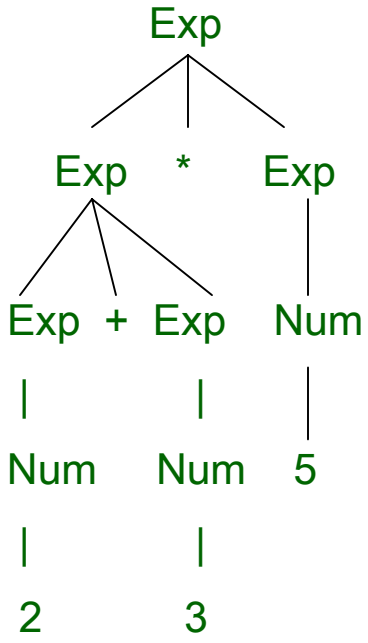
Two parse trees for expression const – const / const

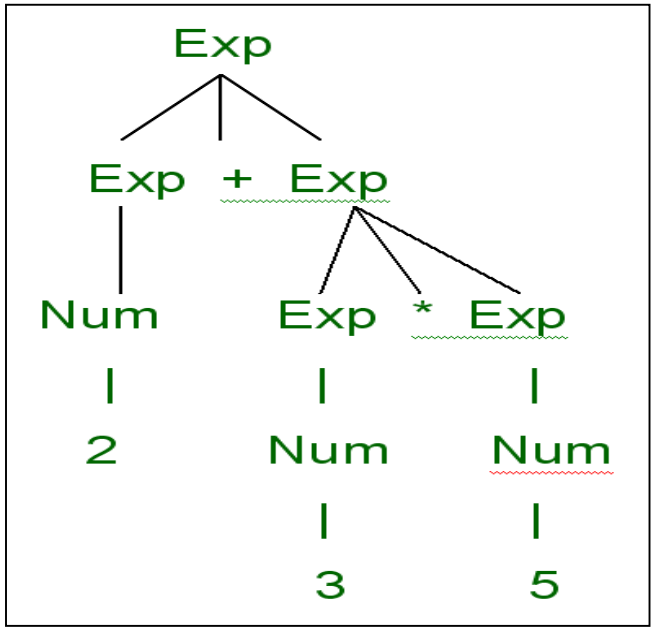


Exercise:

<Exp> → <Num> | <Exp> + <Exp> | <Exp> * <Exp>
<Num>→ 2 | 3 | 5

Two parse trees for string 2 + 3 * 5





In order to avoid ambiguity, it is essential that the grammar generate only one possible structure for each string in the language.

The ambiguity can be eliminated by imposing the precedence of one operator over the other.

Imposing the precedence of one operator over the other.

We say that **op1** has precedence over **op2** if an expression of the form

$$e_1 \text{ op1 } e_2 \text{ op2 } e_3$$

is interpreted only as

$$(e_1 \text{ op1 } e_2) \text{ op2 } e_3$$

In other words, **op1** binds tighter than **op2**.

From the point of view of derivation trees, the fact that e_1 **op1** e_2 **op2** e_3 is interpreted as $(e_1$ **op1** $e_2)$ **op2** e_3 means that the introduction of **op1** must be done at a level strictly lower than **op2.** In order to modify the grammar so that it generates only this kind of tree, a possible solution is to introduce a new syntactic category producing expressions of the form e_1 **op** e_2, and to force an order to op1 and op2.

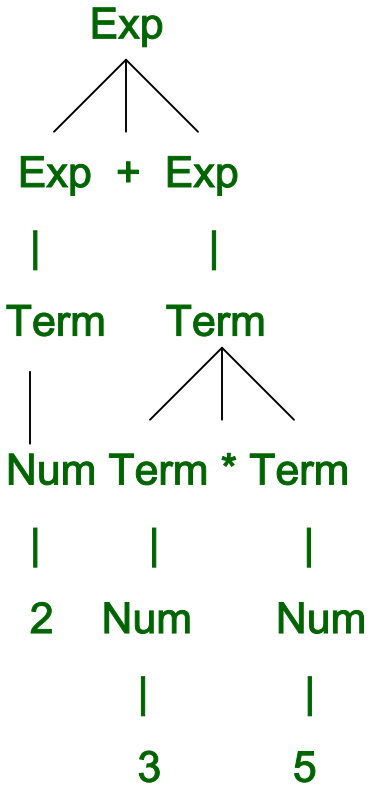
<Exp> → Num | <Exp> + <Exp> | <Exp> * <Exp>
Num→ 2 | 3 | 5

We can eliminate the ambiguities from the grammar by introducing a new syntactic category **Term** producing expressions of the form **<Exp> * <Exp>**

<Exp> → <Exp> + <Exp> | <Term>
<Term> → <Term> * <Term> | Num
Num→ 2 | 3 | 5

This modification corresponds to assigning * a higher priority than +.

In the new grammar there is only one tree which can generate it:



<Exp> → <Exp> + <Exp> | <Term>
<Term> → <Term> * <Term> | Num
Num→ 2 | 3 | 5

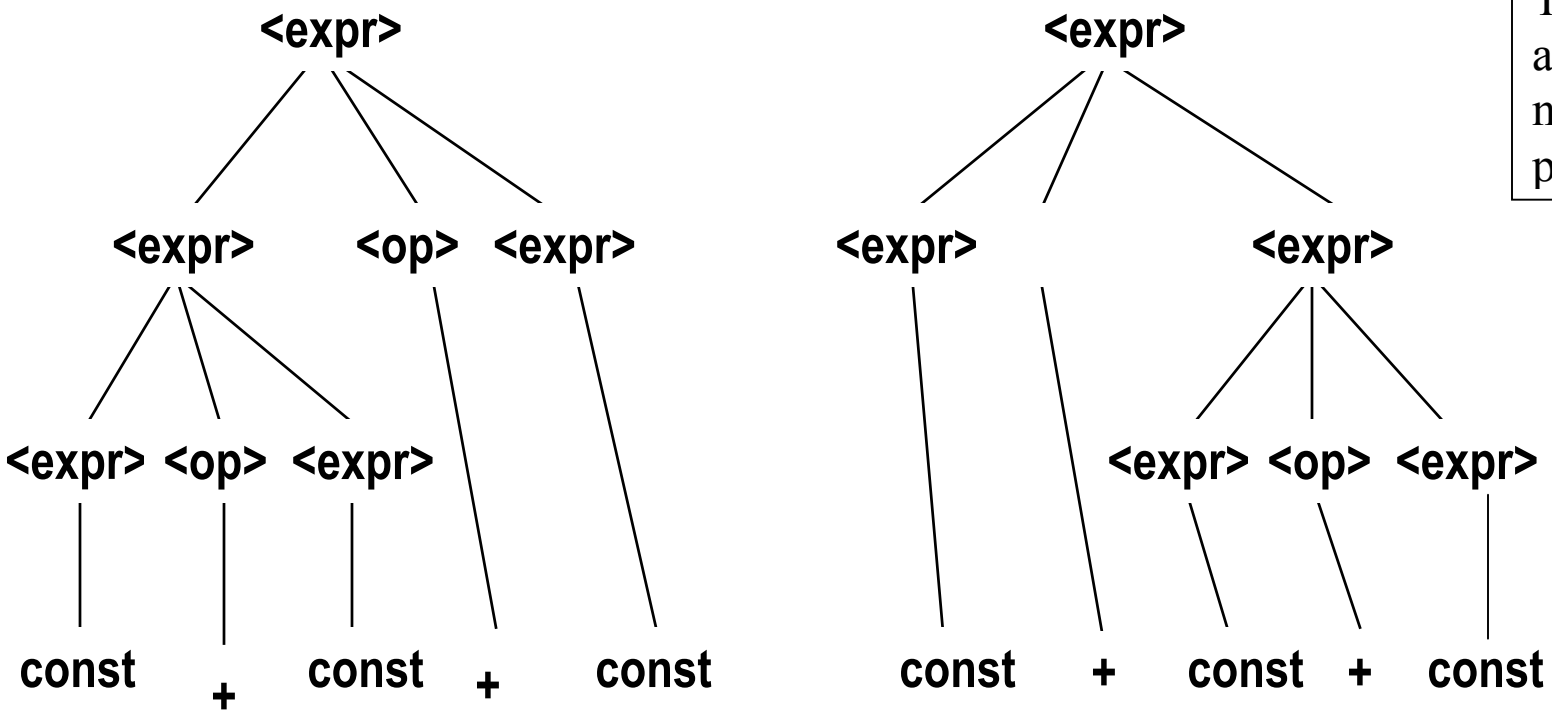
Associativity

Previous grammar is unambiguous regarding the precedence of * and +, but still has ambiguities of another kind.

It allows two different derivation trees for the string 2 + 3 + 5, one corresponding to the structure (2 + 3) + 5 and one corresponding to the structure 2 + (3 + 5).

```
<expr> → <expr> <op> <expr>  |  const
<op> → +
```

Two parse trees for expression const + const + const



This kind of ambiguity does not cause problems

However, an operator might be not associative. For instance the case for the - and ^ exponentiation operators: (5 - 3) - 2 and 5 - (3 - 2) have different values, as well as (5 ^ 3) ^ 2 and 5 ^ (3 ^ 2).

In order to eliminate this kind of ambiguity, we must establish whether the operator is left-associative or right-associative.

- Left-associative: e_1 op e_2 op e_3 is interpreted as (e_1 op e_2) op e (op associates to the left).
- Right-associative: e_1 op (e_2 op e_3) (op associates to the right).

We can impose left-associativity (resp. right-associativity) by using a left-recursive (resp. right-recursive) production for op.

<Exp> → <Exp> + <Exp> | <Term>
<Term> → <Term> * <Term> | <Num> is changed again to

<Exp> → <Exp> + <Term> | <Term>

<Term> → <Term> * Num | Num

This grammar is now unambiguous.

Another Example

<Exp> → Num | <Exp> - <Exp>

This grammar is ambiguous since it allows both the interpretations (5 - 3) - 2 and 5 - (3 - 2).

If we want to impose the left-associativity:

<Exp> → Num | <Exp> - Num

Consider the following grammar:

<Exp> → Num | <Exp> ^ <Exp>

This grammar is ambiguous since it allows both the interpretations (5 ^ 3) ^ 2 and 5 ^ (3 ^ 2).

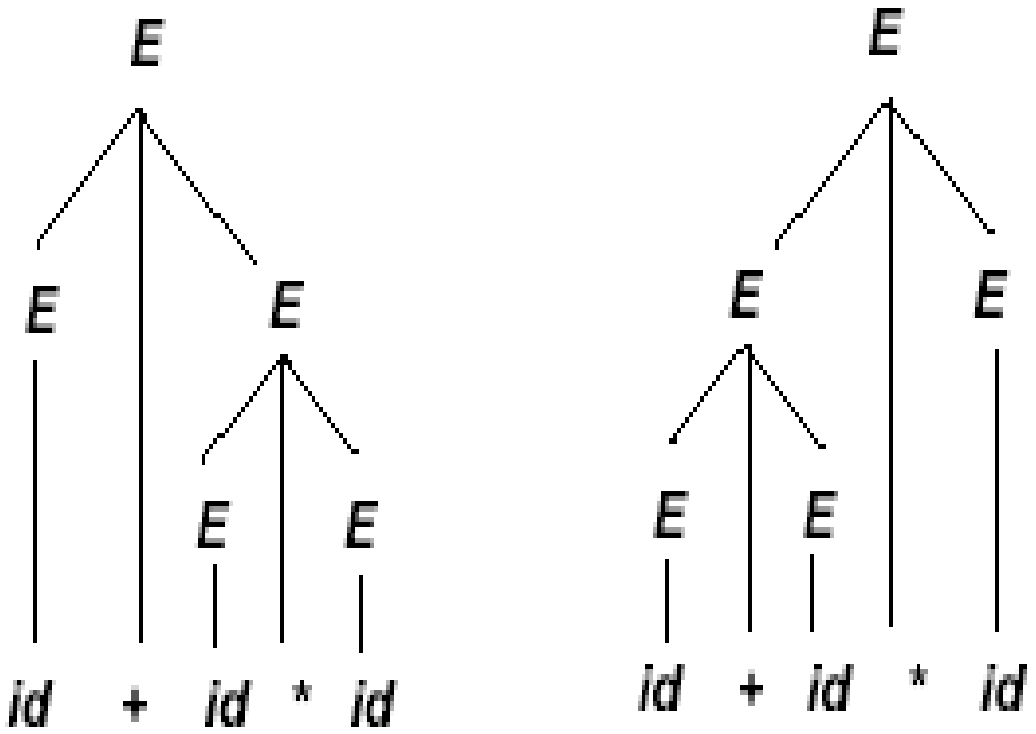
If we want to impose the right-associativity:

<Exp> → Num | Num ^ <Exp>

Generally, we can eliminate ambiguity by revising the grammar.

Grammar: <E> $\rightarrow$ <E> + <E> | <E> * <E> | (E) | id

Two parse trees for expression id + id * id



It is possible to write a grammar for arithmetic expressions that

1. is unambiguous.
2. enforces the precedence of * and / over + and -.
3. enforces left associativity.

Removal of Ambiguity:

Grammar: <E> $\dot{\rightarrow}$ <E> + <E> | <E> * <E> | (E) | id

1. Enforce higher precedence for *

<E> $\dot{\rightarrow}$<E> + <E> | <T>
<T> $\dot{\rightarrow}$<T> * <T> | id | (E)

2. Eliminate right-recursion for <E> $\dot{\rightarrow}$<E> + <E> and <T> $\dot{\rightarrow}$<T> * <T>.

<E> $\dot{\rightarrow}$<E> + <T> | <T>
<T> $\dot{\rightarrow}$<T> * id | <T> * (E) | id | (E)

Example:

```
position := initial + rate * 60
```

Lexical Analysis

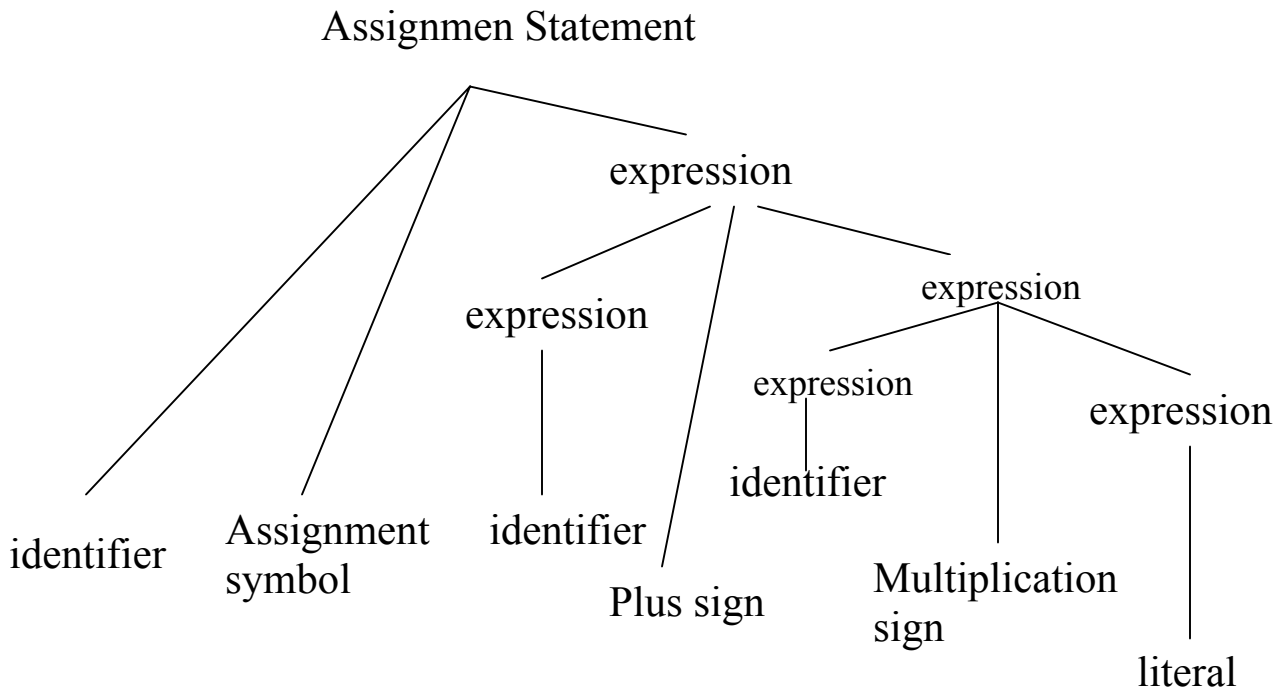
Group these character as follows tokens:

	Token
• position	identifier
• :=	assignment symbol
• initial	identifier
• +	plus sign
• rate	identifier
• *	multiplication sign
• 60	literal

Next, we want to determine that this is a structurally correct statement. This is the main province of syntax analysis or parsing. The result of parsing is a *parse tree*

<Assignment statement> → <identifier><assignment symbol> <expression>
<expression> →<expression> <plus sign> <expression>| <expression> <minus sign>
<expression>| <expression> <multiplication sign> <expression>

<expression> → <identifier> | <literal>




```
position := initial + rate * 60
```

Exercise 1. A grammar of binary numbers

Write a grammar of the language whose elements are all and only those unsigned binary numbers that contain at least three consecutive digits 1. The language includes, for example, 111, 00001111111010 and 1111110, but not 0011000101011 or 1010101010.

Answer 1. A grammar of binary numbers

<string> -> <term> | <mix> <term> | <term> <mix> | <mix> <term> <mix>
<mix> -> <bit> <mix> | <bit>
<bit> -> 0 | 1
<term> -> 1 1 1

or

<string> -> <term> | <bit> <string> | <string> <bit>
<bit> -> 0 | 1
<term> -> 1 1 1

Exercise 2 Parse trees

Consider the following grammar with three terminal symbols: a b g.

<S> -> a <A> | <B> g
<A> -> <C> g | b <A> | a <A>
<B> -> <C> b | <C> <B>
<C> -> a

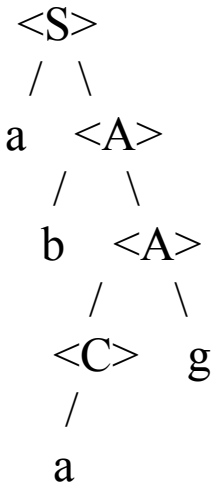
The start symbol is <S>. Consider the string:
a b a g

Show the leftmost derivation of a b a g,

Next, draw a parse tree for the string a b a g.

Answer 2. Parse trees

<S> => a <A> => a b <A> => a b <C> g => a b a g



Exercise:

(1) Describe, in English, the language defined by the following grammar:

<S>→<A><B><C>

<A>→a<A>|a

<B>→b<B>|b

<C>→c<C>|c

(2) Write a grammar for the language consisting of strings that have n copies of the letter a followed by the same number of copies of the letter b, where n>0. For example, the strings ab, aaaabbbb are in the language but a, abb are not.

(3) Convert the following BNF to EBNF

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <expr>

| <expr> * <expr>

| (<expr>)

| <id>

Answer:

(1) One or more a's followed by one or more b's followed by one or more c's.

(2)

<S> → a <S> b | a b

(3)

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> (+ | *) <expr>

| (<expr>)

| <id>