Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Concepts of Programming Languages

BNF and Parse Trees

Dr. Amin Allam

[For more details, refer to "Concepts of Programming Languages" by *Robert Sebesta*]

1 Introduction

Every programming language must have a concise yet understandable description. Programming language *implementors* (writers of compilers or interpreters) must be able to determine how the programming constructs are formed, and their intended effect when executed. Programming language *users* (programmers) must be able to code software systems by referring to the language reference manual. The study of programming languages is divided into examining *syntax* and *semantics*.

The *syntax* of a programming language is the *form* of its expressions, statements, and programming units. The *semantics* of a programming language is the *meaning* of its expressions, statements, and programming units. For example, the *syntax* of a *Java* `while` statement is:

```
while (<boolean_expression>) <statement>
```

The *semantics* of this statement form is that when the current value of the boolean expression is true, the embedded statement is executed then the control returns to the boolean expression. Otherwise, control continues after the `while` statement.

A program consists of a sequence of *statements*. A *statement* consists of a sequence of small units called *lexemes*. A *token* of a language is a category of its *lexemes*. For example, consider the following *Java* statement:

```
index=2*num+17;
```

The *lexemes* and *tokens* of this *statement* are:

Lexeme	Token
index	identifier
=	=
2	integer_constant
*	*
num	identifier
+	+
17	integer_constant
;	;

2 Backus-Naur Form (BNF) grammar

Grammars are formal mechanisms used to describe the *syntax* of programming languages. One of these mechanisms is called *Backus-Naur Form (BNF)* which is very similar to *Context Free Grammars (CFGs)* for natural languages. The following is an example of *BNF grammar* for a small language:

```
<program>     ->  begin <stmt_list> end
<stmt_list>   ->  <stmt> | <stmt> ; <stmt_list>
<stmt>        ->  <var> = <expression>
<var>         ->  A | B | C
<expression>  ->  <var> + <var> | <var> - <var> | <var>
```

The above *BNF grammar* consists of 5 *rules*. Each *rule* contains at its *right hand side (RHS)* one or more *definitions* of the *nonterminal* at the *left hand side (LHS)* of the *rule*. *Definitions* are separated by the *metasymbol* `|` meaning *logical OR*. The *LHS* and the *RHS* are separated by the *metasymbol* `->`. A *nonterminal* is an abstract symbol that helps describe part of the grammar, but cannot appear in any program. Contrary, a *terminal* can appear in programs, and distinguished from *nonterminals* by not being placed inside *pointed brackets*. The first *nonterminal* in a grammar (such as `<program>` in the above grammar) is the *start symbol* of that grammar which defines all programs compatible with that grammar.

The first *rule* defines the *start symbol* of the grammar, which is the `<program>` *nonterminal* occurring in the *LHS* of the *rule*:

```
<program>     ->  begin <stmt_list> end
```

The *RHS* of the above *rule* explains how the `<program>` *nonterminal* can be possibly expanded. The *RHS* consists of the `begin` *terminal*, followed by the `<stmt_list>` *nonterminal* which will be defined later on, followed by the `end` *terminal*. We conclude that a programmer should write the special word `begin` at the beginning of any program conforming with that *BNF grammar* and the special word `end` at the end of these programs. To know what can be written in-between, we should look at the *rule* defining `<stmt_list>`.

The second *rule* defines the `<stmt_list>` *nonterminal*:

```
<stmt_list>   ->  <stmt> | <stmt> ; <stmt_list>
```

which is equivalent to the following two *rules*:

```
<stmt_list>   ->  <stmt>
<stmt_list>   ->  <stmt> ; <stmt_list>
```

which mean that a `<stmt_list>` can be one of: (1) One `<stmt>`. (2) A `<stmt>` followed by the semicolon (`;`) *terminal*, followed by a `<stmt_list>`. This recursive definition means that a `<stmt_list>` may consist of any number (≥ 1) of `<stmt>`s separated by semicolons.

Similarly, we conclude from the remaining *rules* that each statement is an assignment where the RHS can be a variable or two added/subtracted variables. Allowed variables are A, B, and C.

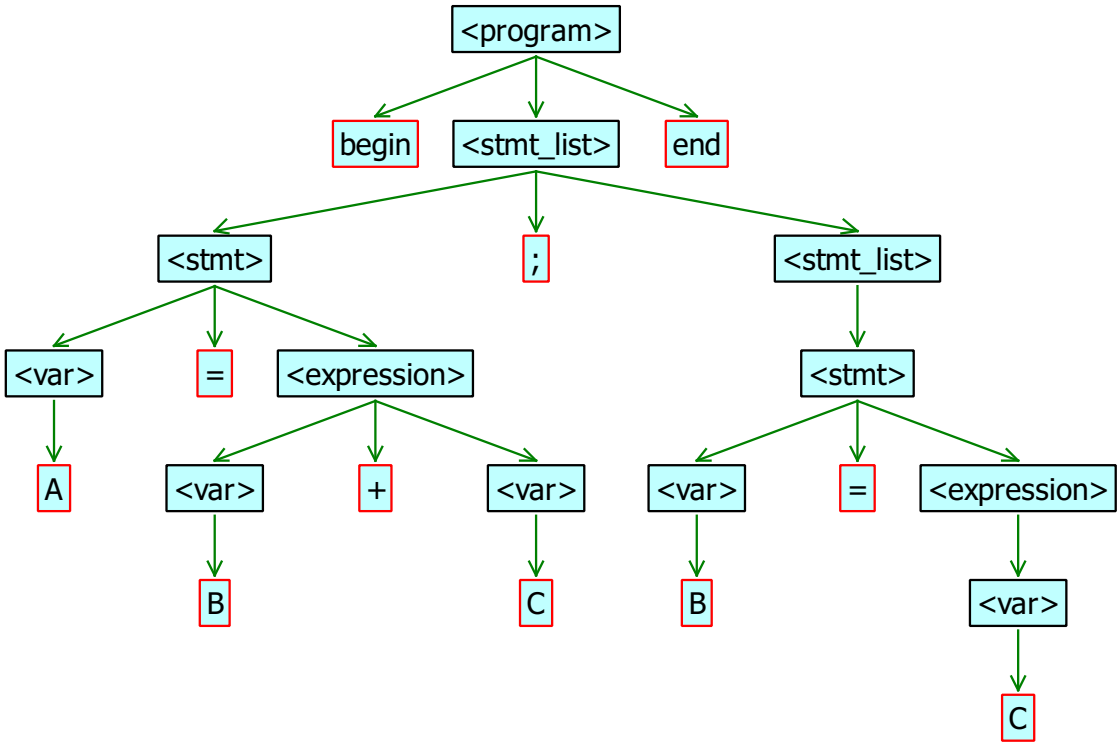
3 Derivations and parse trees

The *BNF grammar* can be viewed as a *generating* device which can generate all programs compatible with that grammar. Generating a specific program is called a *derivation*. The following example is a *derivation* from the above BNF to the program: `begin A=B+C; B=C end`. Any derivation must begin from the *start symbol*.

```
<program> => begin <stmt_list> end
          => begin <stmt> ; <stmt_list> end
          => begin <var>=<expression> ; <stmt_list> end
          => begin A=<expression> ; <stmt_list> end
          => begin A=<var>+<var> ; <stmt_list> end
          => begin A=B+<var> ; <stmt_list> end
          => begin A=B+C ; <stmt_list> end
          => begin A=B+C ; <stmt> end
          => begin A=B+C ; <var>=<expression> end
          => begin A=B+C ; B=<expression> end
          => begin A=B+C ; B=<var> end
          => begin A=B+C ; B=C end
```

The above *derivation* is called *leftmost derivation* because of the performed order. There can be different *derivation* orders for the same program. A program which cannot be derived from a specific *BNF grammar* does not belong to the language generated by that *BNF*.

The hierarchical structure of a program is called a *parse tree*. The following figure is the *parse tree* of the program: `begin A=B+C; B=C end` from the *BNF* of the previous section:

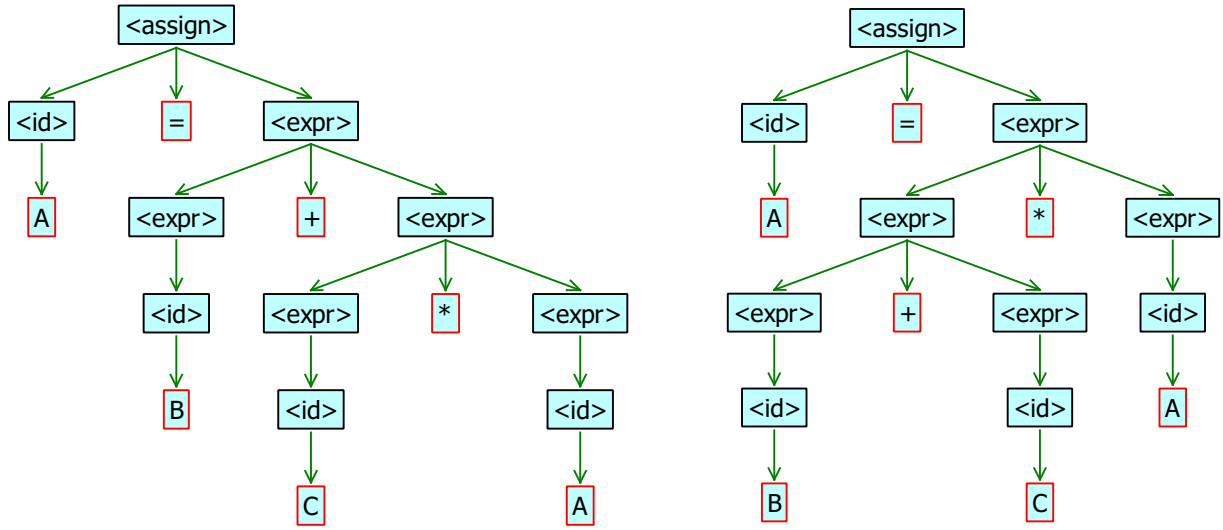


4 Ambiguity, precedence, and associativity

If there exists a program (or part of program) that has two or more distinct parse trees compatible with a given grammar, the grammar is said to be *ambiguous*. For example, consider the following grammar for simple assignment statements:

```
<assign> -> <id> = <expr>
<expr> -> <expr> + <expr> | <expr> * <expr> | ( <expr> ) | <id>
<id>   -> A | B | C
```

The above grammar is *ambiguous* because there exists a statement: `A=B+C*A` which has the following two distinct parse trees:



For the above parse trees of the statement `A=B+C*A`, the compiler may choose to implement the second parse tree which is equivalent to the statement `A=(B+C)*A` although the programmer may have intended to write the statement `A=B+(C*A)` which corresponds to the first parse tree and has very different meaning.

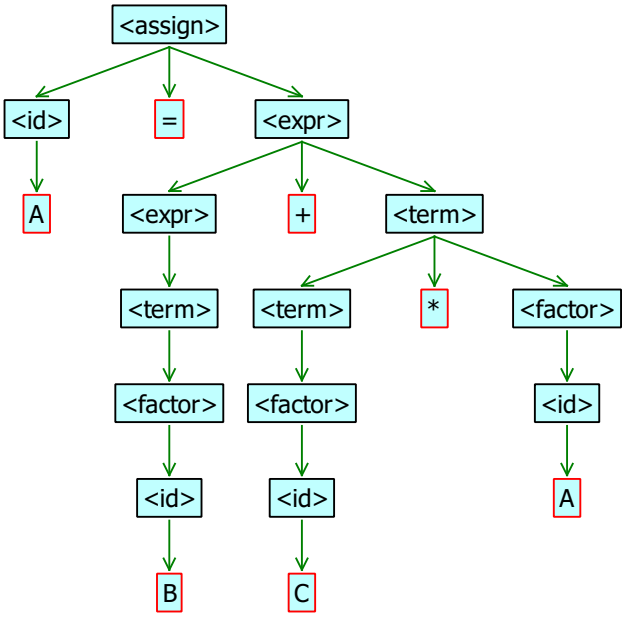
Generally, if a language structure has more than one parse tree, then the meaning of the structure cannot be determined uniquely. The compiler may choose to implement the meaning which was not intended by the programmer. An *ambiguous* grammar should be written to be *unambiguous*. Specifying operator *precedence* and *associativity* can resolve ambiguities caused by the existence of multiple operators without parentheses in the same statement.

Operator precedence is the order of associating operands to operators in the same statement. That is, *precedence* determines which operator gets its operands first. An operator with *higher precedence* is associated to its close operands before an operator with *lower precedence* does. For example, if multiplication `*` has higher precedence that addition `+`, the statement `A=B+C*A` should be mapped to the first parse tree which is equivalent to the statement `A=B+(C*A)`. Note that if parentheses are absent, operators with *higher* precedence should appear *lower* in the parse tree. In the first parse tree, multiplication must be *evaluated* before addition because the addition operation needs the multiplication result. *Parentheses* `()` can be viewed as a construct that has a *precedence* higher than other operators in most programs.

The following BNF grammar is unambiguous since it assigns higher precedence to multiplication than addition by forcing multiplication to be lower in the parse tree. Since parentheses have the highest precedence, they are forced to be lowest in the parse tree:

```
<assign> -> <id> = <expr>
<expr>   -> <expr> + <term>   | <term>
<term>   -> <term> * <factor> | <factor>
<factor> -> ( <expr> ) | <id>
<id>     -> A | B | C
```

In this BNF grammar, there exists the following unique parse tree for the statement `A=B+C*A`:



The previous discussion explained the parse tree if multiple operators with different precedence appear in the same statement without parentheses. However, we did not discuss what should be the parse tree if multiple operators with the same precedence appear in the same statement.

Operator associativity is the order of associating operands to operators having the same *precedence* in the same statement. That is, for operators with equal *precedence*, *associativity* determines which operator gets its operands first. *Left associative* operators with the same *precedence* are associated to their close operands in their order from left to right. *Right associative* operators with the same *precedence* are associated to their close operands in their order from right to left.

The multiplication and addition operators in last BNF grammar are *left associative*, since the related BNF rules are *left recursive*. A rule is said to be *left recursive* if its LHS (left hand side) also appears at the beginning of its RHS (right hand side):

```
<expr>   -> <expr> + <term> | <term>
```

To indicate *right associativity*, the related BNF rule should be *right recursive*. A rule is said to be *right recursive* if its LHS also appears at the right end of its RHS, such as exponentiation $\wedge$:

```
<factor>   -> <newexp> ∧ <factor> | <newexp>
```

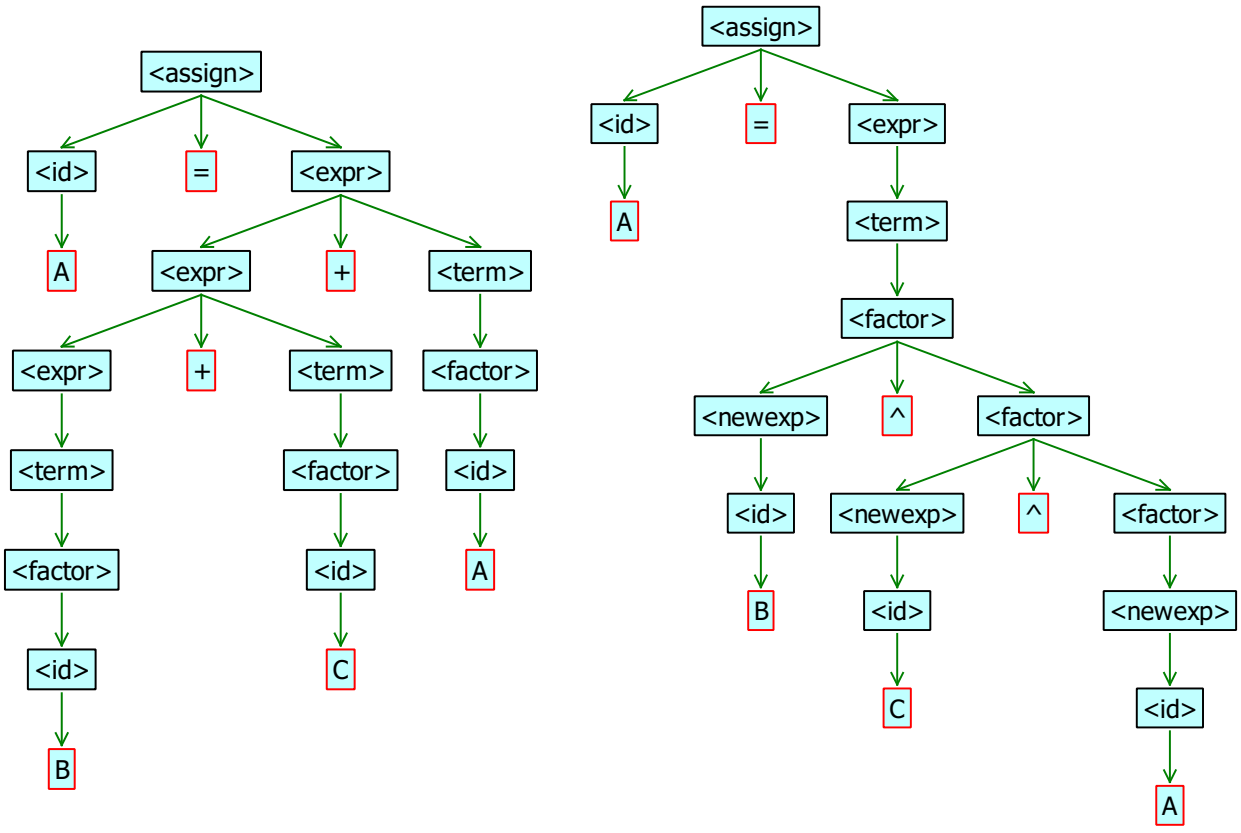
Consider the following table of some operators ordered in groups from highest precedence to lowest precedence:

Operator	Description	Associativity	Type	Example
^	Exponentiation	right to left	Binary	a^b
* /	Multiplication and division	left to right	Binary	a*b
+ -	Addition and subtraction	left to right	Binary	a+b

The following BNF grammar is compatible with the above table:

```
<assign> -> <id> = <expr>
<expr>   -> <expr> + <term> | <expr> - <term> | <term>
<term>   -> <term> * <factor> | <term> / <factor> | <factor>
<factor> -> <newexp> ^ <factor> | <newexp>
<newexp> -> ( <expr> ) | <id>
<id>     -> A | B | C
```

According to the above grammar, the statements A=B+C+A and A=B^C^A have the following parse trees:



The above BNF grammar forces the left addition of the statement A=B+C+A to be lower in its parse tree, hence it will be evaluated first, because addition is left associative. While the right exponentiation of the statement A=B^C^A is lower in its parse tree, hence it will be evaluated first, because exponentiation is right associative.

The following table summarizes properties of some important C++ operators (ordered in groups from highest precedence to lowest precedence). Similar tables exist for other languages:

Operator	Description	Associativity	Type	Example
`++ --`	Postfix increment and decrement	left to right	Unary	`a++`
`+ -`	Unary plus and minus	right to left	Unary	`-a`
`++ --`	Prefix increment and decrement	right to left	Unary	`++a`
`!`	Logical NOT	right to left	Unary	`!a`
`(type)`	C-style cast	right to left	Unary	`(double)a`
`* / %`	Multiplication, division, and remainder	left to right	Binary	`a*b`
`+ -`	Addition and subtraction	left to right	Binary	`a+b`
`< <=`	Relational $<$ and $\leq$	left to right	Binary	`a<b`
`> >=`	Relational $>$ and $\geq$	left to right	Binary	`a>b`
`== !=`	Relational $=$ and $\neq$	left to right	Binary	`a==b`
`&&`	Logical AND	left to right	Binary	`a&&b`
`||`	Logical OR	left to right	Binary	`a||b`
`=`	Assignment	right to left	Binary	`a=b`
`*= /= %=`	Compound assignment	right to left	Binary	`a*=b`
`+= -=`	Compound assignment	right to left	Binary	`a+=b`

5 Extended BNF (EBNF) grammar

Extended BNF (EBNF) is a similar grammar to BNF which attempts to improve its *readability* and *writability*. The following BNF grammar is almost equivalent to the last BNF grammar in the previous section, except that *associativity* information is missing:

```
<assign> -> <id> = <expr>
<expr>   -> <term> { ( + | - ) <term> }
<term>   -> <factor> { ( * | / ) <factor> }
<factor> -> <newexp> { ∧ <newexp> }
<newexp> -> ( <expr> ) | <id>
<id>     -> A | B | C
```

The *braces (curly brackets)* {} metasymbol can be used to indicate that the inside construct can be repeated *zero or more* times. For example, the following rule:

```
<factor> -> <newexp> { ∧ <newexp> }
```

means that `<factor>` can be expanded into `<newexp>` or `<newexp>∧<newexp>` or `<newexp>∧<newexp>∧<newexp>`, etc.

The $\{\}^+$ metasymbol can be used to indicate that the inside construct can be repeated *one or more* times. For example, the following rule:

```
<program> -> begin { <stmt> }+ end
```

means that a `<program>` can include one or more `<stmt>`.

The *parentheses* `()` metasymbol can be used with the `|` metasymbol to indicate that exactly one of the inside constructs must be chosen. For example `( + | - )` can be replaced by + or −. Hence, the following rule:

```
<expr> -> <term> { ( + | - ) <term> }
```

means that `<expr>` can be expanded into `<term>` or `<term>+<term>` or `<term>-<term>` or `<term>+<term>+<term>` or `<term>+<term>-<term>` or `<term>-<term>+<term>` or `<term>-<term>-<term>` or `<term>+<term>+<term>+<term>`, etc.

The *brackets* `[]` metasymbol can be used to indicate that the inside construct is optional. That is, it can be ignored or used (*zero or one* times). For example, the following rule:

```
<selection> -> if ( <expr> ) <stmt> [ else <stmt> ]
```

means that `<selection>` can be expanded into `if(<expr>) <stmt>`
or `if(<expr>) <stmt> else <stmt>`.

Since *associativity* information is missing, it should be included verbally with the *EBNF* rules.

6 Attribute grammars

Attribute grammars are extensions to BNF grammars that can describe some aspects of *syntax* which are impossible or difficult to be described using BNF grammars. Such *syntactic aspects* are called *static semantics*, although such naming does not seem to be accurate. *Attribute grammars* are just BNF grammars augmented with:

- *Attributes* associated with some grammar symbols (terminals and nonterminals). Most attributes get their values determined from other attributes, except for *intrinsic attributes* which are associated with leaf nodes and get their values determined from outside the parse tree,
- *Attribute computation functions* associated with some grammar rules to specify how attribute values are computed.
- *Predicates* associated with some grammar rules to specify additional syntactic rules.

Consider the following BNF grammar:

```
<assign> -> <var> = <expr>
<expr>   -> <var> + <var> | <var>
<var>    -> A | B | C
```

Suppose we wish to add the following syntactic restrictions to the above grammar:

- Each variable and expression has a type which is either *int* or *real*.
- An *int* can be added to another *int* to result an *int*.
- A *real* can be added to another variable (*int* or *real*) to result a *real*.
- The *LHS* type of any assignment must match the type of its *RHS*.

To incorporate the above restrictions, we associate an attribute called `type` to the nonterminals `<var>` and `<expr>`. Additionally, we augment the BNF rules with *attribute computation functions* and *predicates* as follows:

Syntax rule: `<assign> -> <var> = <expr>`
Predicate: `<var>.type == <expr>.type`

Syntax rule: `<expr> -> <var>[1] + <var>[2]`
Attribute computation function:

```
if (<var>[1].type == int AND <var>[2].type == int)
then <expr>.type <- int
else <expr>.type <- real
```

Syntax rule: `<expr> -> <var>`
Attribute computation function:

```
<expr>.type <- <var>.type
```

Syntax rule: `<var> -> A | B | C`
Attribute computation function:

```
<var>.type <- lookup(<var>.string)
```

The `lookup` function looks up a variable name in the symbol table and returns the variable's type. The parse tree is constructed using the original BNF rules which are called *syntax rules* in the above *attribute grammar*. While constructing the parse tree, *attribute computation functions* are applied to determine the *attribute values*. *Predicates* are checked for validation. If a *predicate* value is not true, the whole construction is considered invalid.

7 Operational semantics

Operational semantics is a method to describe the *semantics* of a programming construct. Such described *semantics* are sometimes called *dynamic semantics* to differentiate them from *static semantics* described in the previous section. The basic idea of *operational semantics* is to describe the language constructs in terms of simpler well-known language constructs! Although the terms *simpler* and *well-known* cannot be well-defined, this method is probably the most practical way to describe semantics. The following is an example to describe a C statement in terms of a simpler language:

C Statement	Meaning

`for (expr1; expr2; expr3)`
`{`
`  ...`
`}`
	`  expr1;`
	`loop:  if expr2==0 goto out`
	`  ...`
	`  expr3;`
	`  goto loop`
	`out:  ...`