Defining Program Syntax
BNF
EBNF

Syntax And Semantics

- Programming language syntax: how programs look, their form and structure
  - Syntax is defined using a kind of formal grammar
- Programming language semantics: what programs do, their behavior and meaning
  - Semantics is harder to define

An English Grammar

A sentence is a noun phrase, a verb, and a noun phrase.	<*S*> ::= <*NP*> <*V*> <*NP*>
A noun phrase is an article and a noun.	<*NP*> ::= <*A*> <*N*>
A verb is…	<*V*> ::= **`loves`** | **`hates`** | **`eats`**
An article is…	<*A*> ::= **`a`** | **`the`**
A noun is...	<*N*> ::= **`dog`** | **`cat`** | **`rat`**

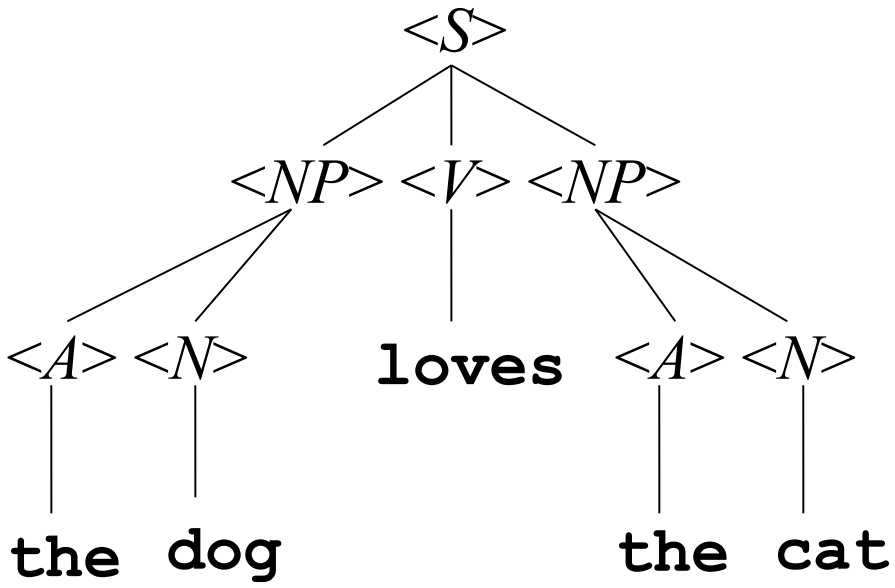
How The Grammar Works

- The grammar is a set of rules that say how to build a tree—a *parse tree*
- You put <*S*> at the root of the tree
- The grammar's rules say how children can be added at any point in the tree
- For instance, the rule

  says you can add nodes <*NP*>, <*V*>, and <*NP*>, in that order, as children of <*S*>

$$<S> ::= <NP> <V> <NP>$$

A Parse Tree



- What is BNF?
- Why need BNF?
- What does BNF look like?
- Examples of BNF

What is BNF?

- BNF = Backus-Naur Form
- It is a formal, mathematical way to specify context-free grammars
- BNF is Meta language
- It is precise and unambiguous

A metalanguage is a language that is used to describe another language.
BNF is a metalanguage for programming languages.

History

- In the mid-1950s, Chomsky, a noted linguist (among other things), described four classes of generative devices or grammars that define four classes of languages (Chomsky, 1956, 1959). Two of these grammar classes, named *context-free* and *regular*, turned out to be useful for describing the syntax of programming languages.
- The syntax of whole programming languages, with minor exceptions, can be described by context-free grammars.
- Because Chomsky was a linguist, his primary interest was the theoretical nature of natural languages. He had no interest at the time in the artificial languages used to communicate with computers. So it was not until later that his work was applied to programming languages.
- Shortly after Chomsky's work on language classes, the ACM-GAMM group began designing ALGOL 58. A landmark paper describing ALGOL 58 was presented by John Backus, a prominent member of the ACM-GAMM group, at an international conference in 1959 (Backus, 1959). This paper introduced a new formal notation for specifying programming language syntax. The new notation was later modified slightly by Peter Naur.
- This revised method of syntax description became known as Backus-Naur Form, or simply BNF. BNF is a natural notation for describing syntax. In fact, something similar to BNF was used by *Panini* to describe the syntax of Sanskrit *several hundred years before Christ* (Ingerman, 1967).

Why need BNF?

EXAMPLE of NATURAL LANGUAGE

"He fed her cat food."

Can have different meanings,

- He fed a woman's cat some food.
- He fed a woman some food that was intended for cats.

What does BNF look like?

BNF grammar consists of 4 parts,

- The set of tokens
- The set of non terminal symbols
- The start symbol
- The set of productions

LHS = RHS

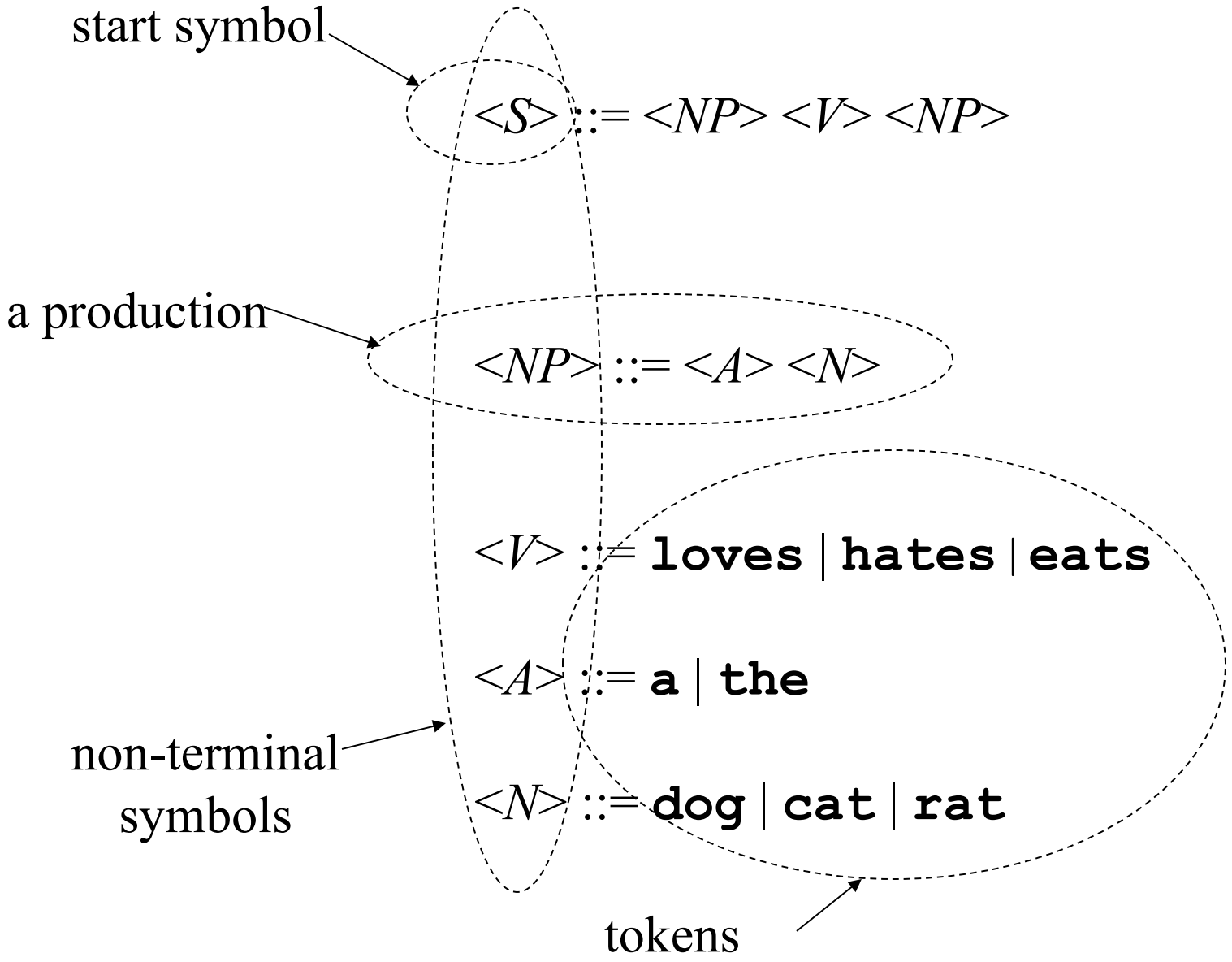
Non-terminals = Terminals/non-terminals

BNF uses abstractions for syntactic structures. A simple C or Java assignment statement, for example, might be represented by

<assign> → <var> = <expression>

LHS is the abstraction being defined. (nonterminal symbols)
RHS consists of some mixture of tokens, lexemes, and references to other abstractions. (terminal symbols)
Altogether, the definition is called a rule, or production.

A BNF description, or grammar, is a collection of rules.


start symbol
<S> ::= <NP> <V> <NP>
a production
<NP> ::= <A> <N>
<V> ::= loves | hates | eats
<A> ::= a | the
non-terminal symbols
<N> ::= dog | cat | rat
tokens

Definition

- tokens - terminal symbols - are the smallest units of syntax
  - Strings of one or more characters of program text
  - They are atomic: not treated as being composed from smaller parts
- *nonterminal symbols* - stand for larger pieces of syntax are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- *Productions* - are rules for replacing (or rewriting) nonterminal symbols (on the left side of the production) in a string with other nonterminal or terminal symbols (on the right side of the production).
- a *start symbol* - is a special nonterminal symbol that appears in the initial string generated by the grammar.

Recursive Rule

A rule is recursive if its LHS appears in its RHS. The following rules illustrate how recursion is used to describe lists:

<ident_list> → identifier | identifier, <ident_list>

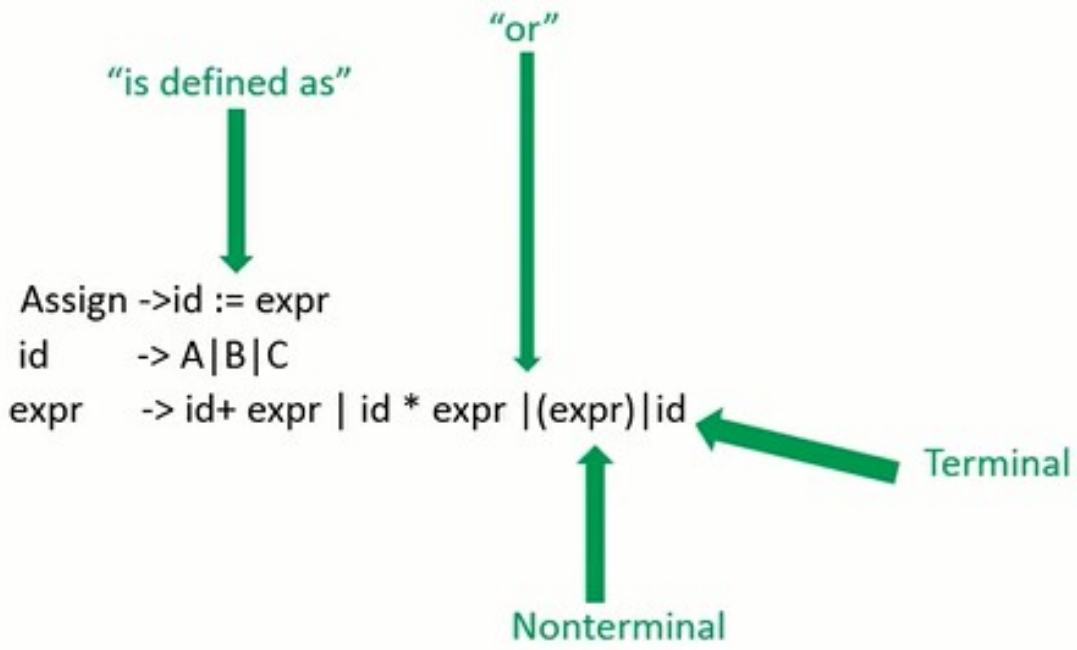
This defines <ident_list> as either a single token (identifier) or an identifier followed by a comma and another instance of <ident_list>

The sentences of the language are generated through a sequence of applications of the rules, beginning with a special nonterminal of the grammar called the ***start symbol***.

This sequence of rule applications is called a ***derivation***.

In a grammar for a complete programming language, the start symbol represents a complete program and is often named <program>.

What does BNF look like?



An Example Grammar

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

An Example Derivation

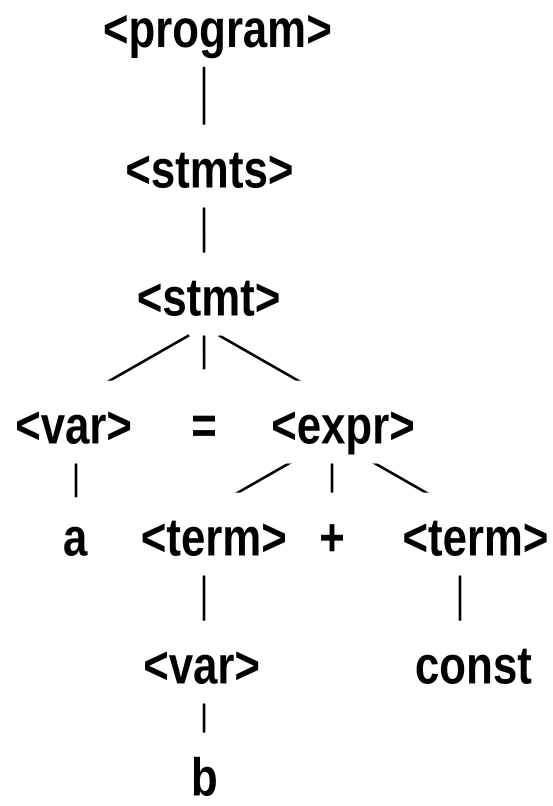
```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



Example

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt>
| <stmt> ; <stmt_list>

<stmt> → <var> = <expression>

<var> → A | B | C

<expression> → <var> + <var>
| <var> – <var>
| <var>

A derivation of a program in this language follows:

<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 symbol => is read “derives”

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

The above grammar describes assignment statements whose right sides are arithmetic expressions with multiplication and addition operators and parentheses.

A = B * (A + C) lets attain it with leftmost derivation:

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

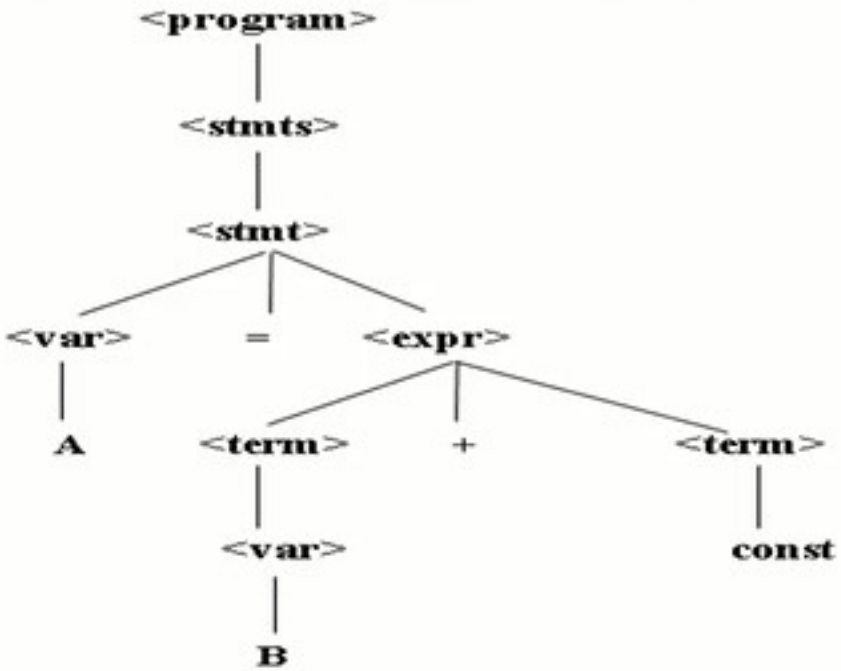
Parse Trees

- A Parse tree is a hierarchical representation of a derivation

Every internal node of a parse tree is labeled with a *nonterminal symbol;*

every leaf is labeled with a *terminal symbol.*

Every subtree of a parse tree describes *one instance of an abstraction* in the sentence.



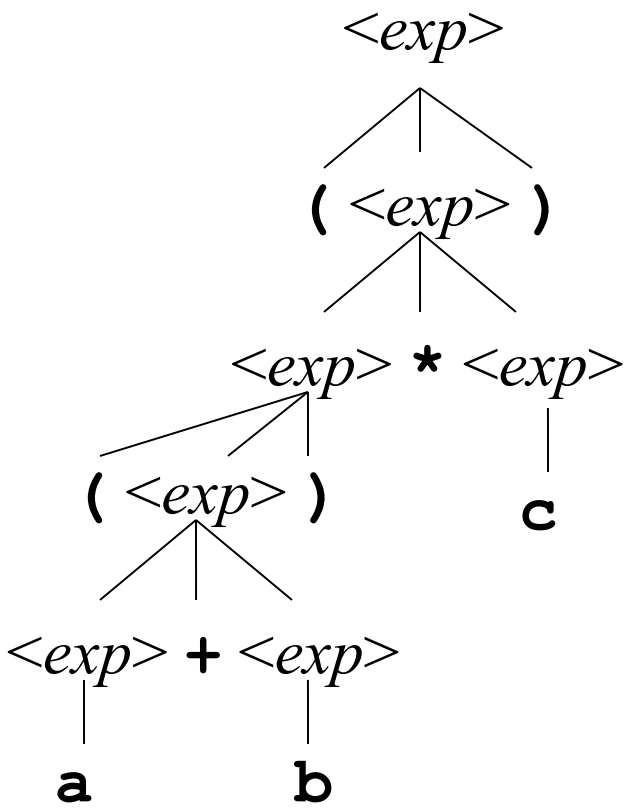
A Programming Language Grammar

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

- An expression can be the sum of two expressions, or the product of two expressions, or a parenthesized subexpression
- Or it can be one of the variables **a**, **b** or **c**

A Parse Tree

((a+b)*c)



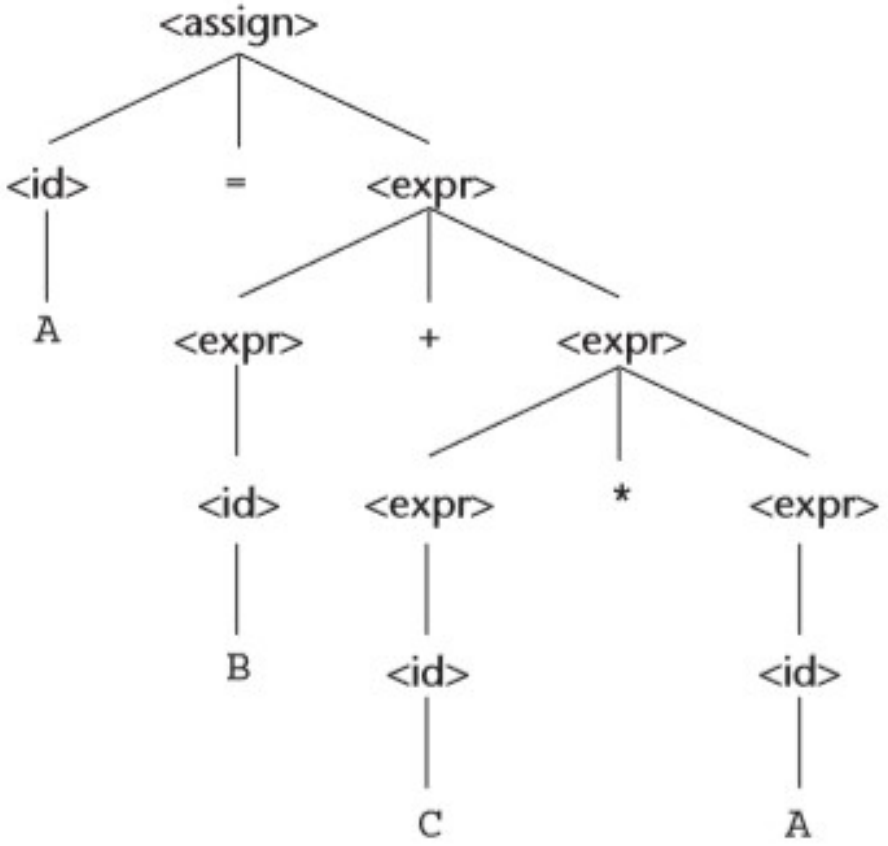
Ambiguity

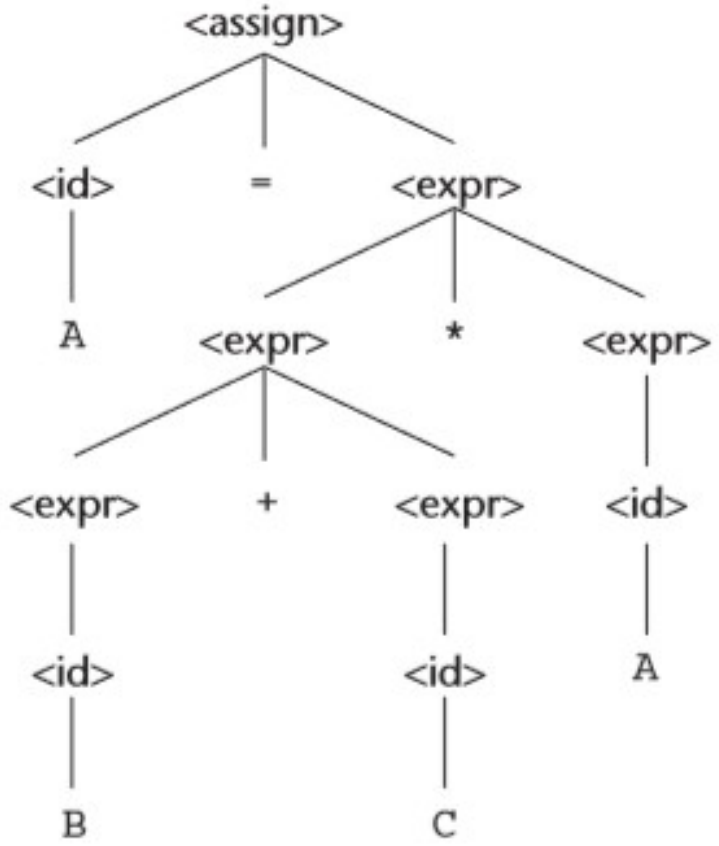
- A grammar that generates a sentential form for which there are two or more distinct parse trees is said to be ambiguous

An Ambiguous Grammar for Simple Assignment Statements

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

It is ambiguous because the sentence
A = B + C * A has two distinct parse trees


<assign>
<id>
=
<expr>
A
<expr>
+
<expr>
<id>
<expr>
*
<expr>
B
<id>
<id>
C
A


<assign>
<id>
=
<expr>
A
<expr>
*
<expr>
<expr>
+
<expr>
<id>
<id>
<id>
A
B
C

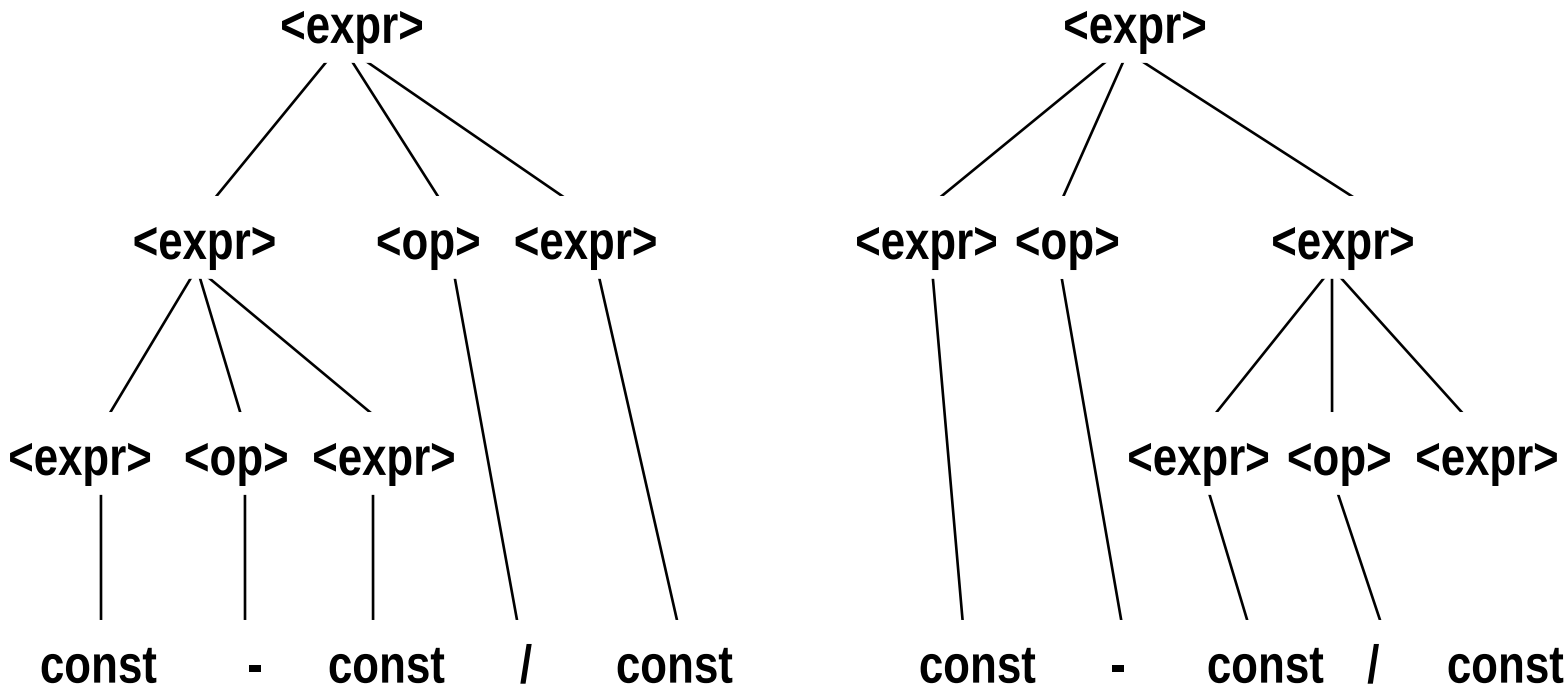
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

<expr> → <expr> <op> <expr> | const

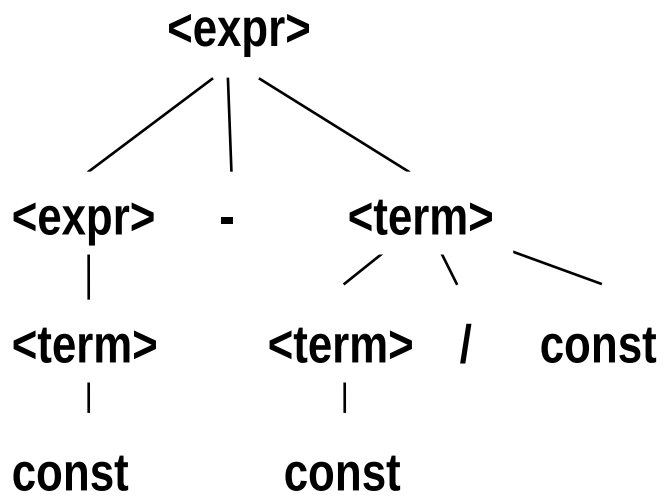
<op> → / | -



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



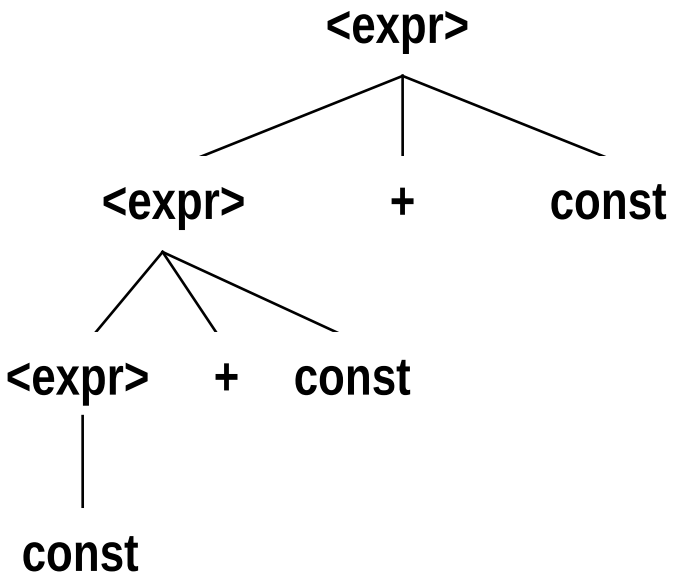
Operators

- Special syntax for frequently-used simple operations like addition, subtraction, multiplication and division
- The word *operator* refers both to the token used to specify the operation (like **+** and *****) and to the operation itself
- Usually predefined, but not always
- Usually a single token, but not always

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Operator Terminology

- *Operands* are the inputs to an operator, like **`1`** and **`2`** in the expression **`1+2`**
- *Unary* operators take one operand: **`-1`**
- *Binary* operators take two: **`1+2`**
- *Ternary* operators take three: **`a?b:c`**

More Operator Terminology

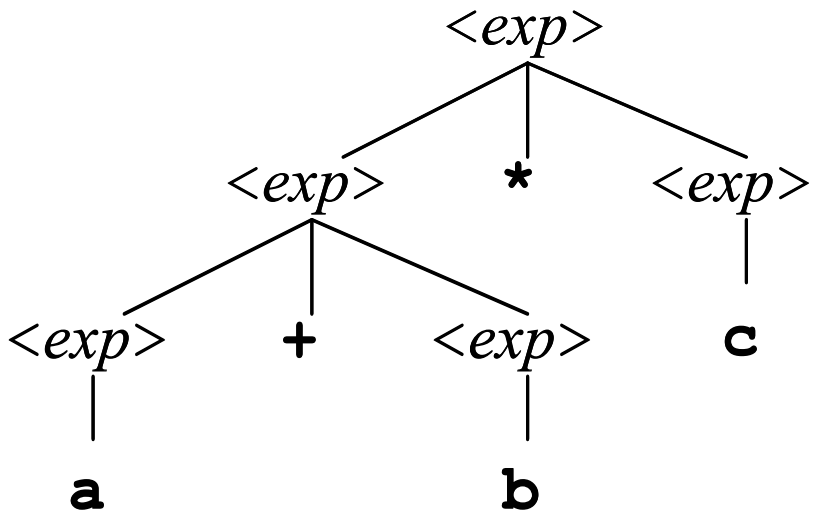
- In most programming languages, binary operators use an *infix* notation: **`a + b`**
- Sometimes you see *prefix* notation: **`+ a b`**
- Sometimes *postfix* notation: **`a b +`**
- Unary operators, similarly:
  - (Can't be infix, of course)
  - Can be prefix, as in **`-1`**
  - Can be postfix, as in **`a++`**

Working Grammar

G4: <*exp*> **::=** <*exp*> **+** <*exp*>
| <*exp*> ***** <*exp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

This generates a language of arithmetic expressions using parentheses, the operators **+** and *****, and the variables **a**, **b** and **c**

Issue #1: Precedence



Our grammar generates this tree for **`a+b*c`**. In this tree, the addition is performed before the multiplication, which is not the usual convention for operator *precedence*.

Operator Precedence

- Applies when the order of evaluation is not completely decided by parentheses
- Each operator has a *precedence level*, and those with higher precedence are performed before those with lower precedence, as if parenthesized
- Most languages put **`*`** at a higher precedence level than **`+`**, so that

  `a+b*c` = **`a+(b*c)`**

Precedence Examples

- C (15 levels of precedence—too many?)

```
a = b < c ? * p + b * c : 1 << d ()
```

- Pascal (5 levels—not enough?)

- Smalltalk (1 level for all binary operators)

```
a <= 0 or 100 <= a
```

Error!

```
a + b * c
```

Precedence In The Grammar

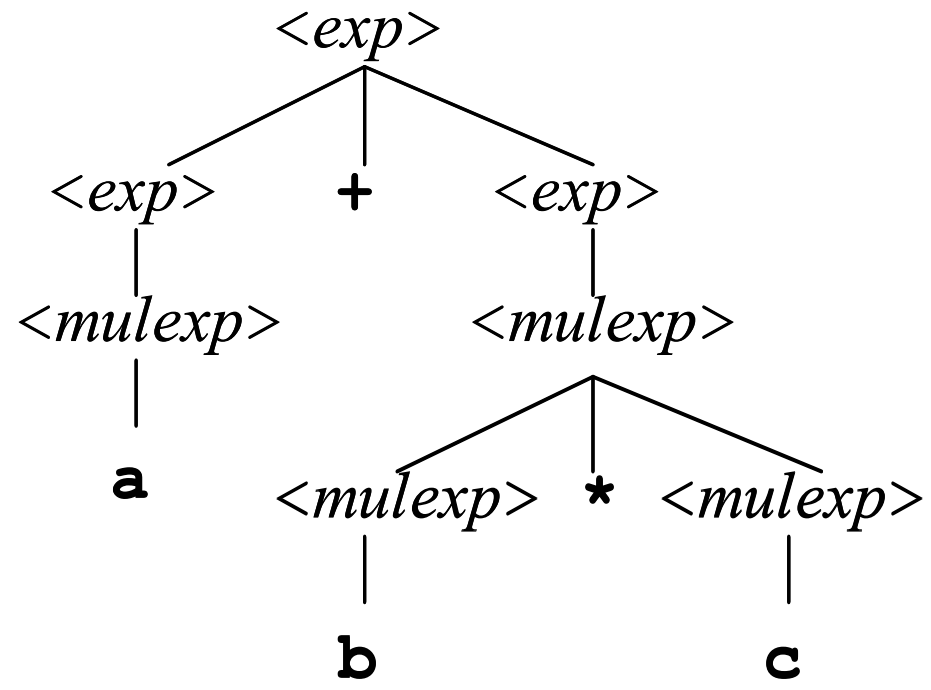
G4: <*exp*> **::=** <*exp*> **+** <*exp*>
| <*exp*> ***** <*exp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

To fix the precedence problem, we modify the grammar so that it is forced to put ***** below **+** in the parse tree.

G5: <*exp*> **::=** <*exp*> **+** <*exp*> | <*mulexp*>
<*mulexp*> **::=** <*mulexp*> ***** <*mulexp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

Correct Precedence

G5 parse tree:



Our new grammar generates this tree for **`a+b*c`**. It generates the same language as before, but no longer generates parse trees with incorrect precedence.

Alternatives

- When there is more than one production with the same left-hand side, an abbreviated form can be used
- The BNF grammar can give the left-hand side, the separator `::=`, and then a list of possible right-hand sides separated by the special symbol `|`

Example

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

Note that there are six productions in this grammar. It is equivalent to this one:

<*exp*> ::= <*exp*> **+** <*exp*>
<*exp*> ::= <*exp*> ***** <*exp*>
<*exp*> ::= **(** <*exp*> **)**
<*exp*> ::= **a**
<*exp*> ::= **b**
<*exp*> ::= **c**

Empty

- The special nonterminal <*empty*> is for places where you want the grammar to generate nothing
- For example, this grammar defines a typical if-then construct with an optional else part:

<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> <*else-part*>
<*else-part*> **::=** **else** <*stmt*> | <*empty*>

Building Parse Trees

- To build a parse tree, put the start symbol at the root
- Add children to every non-terminal, *following any one of the productions for that non-terminal in the grammar*
- Done when all the leaves are tokens
- Read off leaves from left to right—that is the string derived by the tree

Practice

<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**

Show a parse tree for each of these strings:

a+b
a*b+c
(a+b)
(a+(b))

Compiler Note

- What we just did is *parsing*: trying to find a parse tree for a given string
- That's what compilers do for every program you try to compile: try to build a parse tree for your program, using the grammar for whatever language you used
- Take a course in compiler construction to learn about algorithms for doing this efficiently

Language Definition

- We use grammars to define the syntax of programming languages
- The language defined by a grammar is the set of all strings that can be derived by some parse tree for the grammar
- As in the previous example, that set is often infinite (though grammars are finite)
- Constructing grammars is a little like programming…

Constructing Grammars

- Most important trick: divide and conquer
- Example: the language of C declarations: a type name, a list of variables separated by commas, and a semicolon
- Each variable can be followed by an initializer:

```
float a;
boolean a,b,c;
int a=1, b, c=1+2;
```

Example, Continued

- Easy if we postpone defining the comma-separated list of variables with initializers:

- Primitive type names are easy enough too:

<var-dec> **::=** *<type-name>* *<declarator-list>* **;**

- (Note: skipping constructed types: class names, interface names, and array types)

<type-name> **::= boolean** | **byte** | **short** | **int**
| **long** | **char** | **float** | **double**

Example, Continued

- That leaves the comma-separated list of variables with initializers
- Again, postpone defining variables with initializers, and just do the comma-separated list part:

<*declarator-list*> ::= <*declarator*>
| <*declarator*> , <*declarator-list*>

Example, Continued

- That leaves the variables with initializers:

<*declarator*> **::=** <*variable-name*>
| <*variable-name*> **=** <*expr*>

- For full Java, we would need to allow pairs of square brackets after the variable name
- There is also a syntax for array initializers
- And definitions for <*variable-name*> and <*expr*>

Where Do Tokens Come From?

- Tokens are pieces of program text that we do not choose to think of as being built from smaller pieces
- Identifiers (**`count`**), keywords (**`if`**), operators (**`==`**), constants (**`123.4`**), etc.
- Programs stored in files are just sequences of characters
- How is such a file divided into a sequence of tokens?

Lexical Structure And Phrase Structure

- Grammars so far have defined *phrase structure*: how a program is built from a sequence of tokens
- We also need to define *lexical structure*: how a text file is divided into tokens

Historical Note #1

- Early languages sometimes did not separate lexical structure from phrase structure
  - Early Fortran and Algol dialects allowed spaces anywhere, even in the middle of a keyword
  - Other languages like PL/I allow keywords to be used as identifiers
- This makes them harder to scan and parse
- It also reduces readability

Historical Note #2

- Some languages have a *fixed-format* lexical structure—column positions are significant
  - One statement per line (i.e. per card)
  - First few columns for statement label
  - Etc.
- Early dialects of Fortran, Cobol, and Basic
- Most modern languages are *free-format*: column positions are ignored

BNF Variations

- Some use → or = instead of ::=
- Some leave out the angle brackets and use a distinct typeface for tokens
- Some allow single quotes around tokens, for example to distinguish ‘|’ as a token from | as a meta-symbol

Extended BNF

- Optional parts are placed in brackets [ ]

  `<proc_call> -> ident [(<expr_list>)]`

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

  `<term> → <term> (+|-) const`

- Repetitions (0 or more) are placed inside braces { }

  `<ident> → letter {letter|digit}`

BNF and EBNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
 <term> → <term> * <factor>
        | <term> / <factor>
       | <factor>
```

- EBNF

```
<expr> → <term> {(+ | -) <term>}
 <term> → <factor> {(* | /) <factor>}
```

Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of =>
- Use of `opt` for optional parts
- Use of `oneof` for choices

Pattern Matching

- Programs that manipulate text often have a need to search a string for things other than simple substrings.

For example: “Find all integer numerals in this string” or “Find all Scheme tokens in this program text.”

- Another application might be to check input: “Does this user’s response have the proper form?”
- Numerous programming languages provide some kind of pattern-matching facility to do this sort of thing.
- We can think of this as a kind of declarative programming, because the programmer is saying, e.g., “find somethin that looks like this” rather than “search for the substring ‘(’, then look for a ’)’ after that” to check for a parenthesized expression.
- It’s up to library code to figure out how to find convert “looks like” into actual steps to search for that condition.

Regular Expressions

- One of the most widely available and useful mechanisms is the regular expression.
- Formally, regular expressions denote sets of strings that are called regular languages.
- But normally, we think of them as patterns that match certain strings

- A regular expression is a sequence of characters that is used to search pattern. It is mainly used for pattern matching with strings, or string matching, etc. They are a generalized way to match patterns with sequences of characters. It is used in every programming language like C++, Java, and Python.

Regular expressions in C

Expression	Description
[]	Used to find any of the characters or numbers specified between the brackets.
[:number:]	Used to find any digit.
[:lower:]	Used to find lowercase alphabets.
[:word:]	Used to find letters numbers and underscores.

Creation of Regular Expression

For compiling or creating the regular expression **regcomp()** function is used. It takes three arguments:

Syntax: regcomp(&regex, expression, flag)

Where:

regex is a pointer to a memory location where expression is matched and stored.

expression is a string type

flag to specify the type of compilation

Return Value:

- **0**: when successful compilation is done.
- **Error_code**: When there is unsuccessful compilation of the expression.

Example

```
#include <regex.h>
#include <stdio.h>

int main()
{
        regex_t reegex;
        int value;
        value = regcomp( &reegex, "[:word:]", 0);
        if (value == 0) {
                printf("RegEx compiled successfully.");
        }
        else {   printf("Compilation error.");   }
        return 0;
}
```

Matching of Pattern using Regular Expression

The regexec() function is used to match a string against a pattern. It takes in five arguments:

- A precompiled pattern
- A string in which the pattern needs to be searched for.
- Information regarding the location of matches.
- Flags to specify a change in the matching behaviour.

Syntax: regexec(&regex, expression, 0, NULL, 0);

Return Value:

- **0**: If there is a match.
- **REG_NOMATCH**: If there is no match.

Using a regular expression with scanf()

```c
#include<stdio.h>
#include<stdlib.h>
int main() {
    char name[15];
    // Taking a name as an input.
    // name can only include alphabets
    scanf("%[a-zA-Z]",name);
    printf("%s",name);
    return 0;
}
```

Using a regular expression to accept only letters as input.

Powerful Scanf 

char str[100];

scanf("%[^\n]", str); - ?

scanf(" %[^5] ", str); -?

scanf(" **%[0–9]** ", str); -?

`scanf("%*[^:]%*2c%[^\n]", str); - ?`
`Everything before:` would be removed by `**%*[^:]**`
“*: ”* this, a colon and a space total two character would be removed by `**%*2c**`
If you write the following in the console:
`Any combination: You are free to use any combination.`
`You’ll get?  You are free to use any combination.`