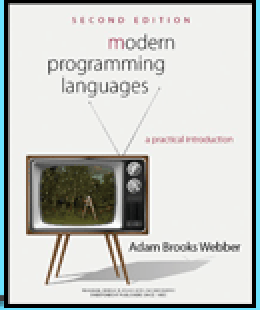
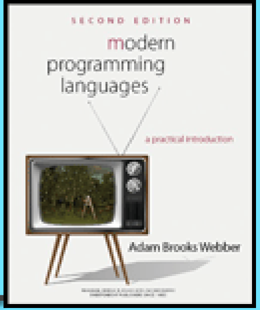


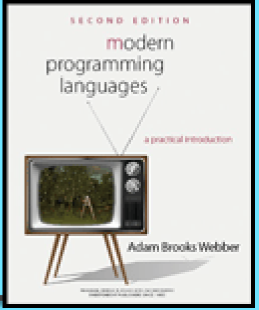
Defining Program Syntax

In formal language theory, a grammar describes how to form strings from a language's alphabet that are valid according to the language's syntax.



Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- YACC



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…

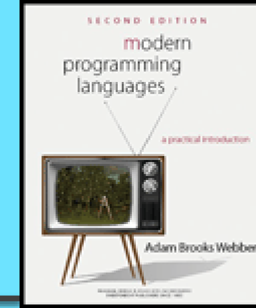
<*V*> ::= **`loves`** | **`hates`** | **`eats`**

An article is…

<*A*> ::= **`a`** | **`the`**

A noun is...

<*N*> ::= **`dog`** | **`cat`** | **`rat`**

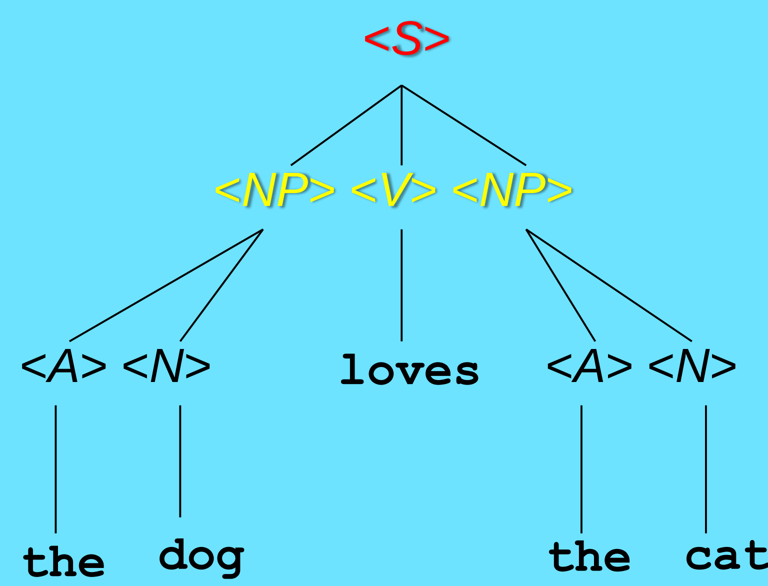


How The Grammar Works

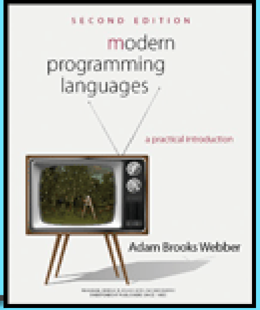
- The grammar is a set of rules that say how to build a tree—a *parse tree*
- You put *<S>* at the root of the tree
- The grammar's rules say how children can be added at any point in the tree
- For instance, the rule

  <S> ::= *<NP>* *<V>* *<NP>*

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

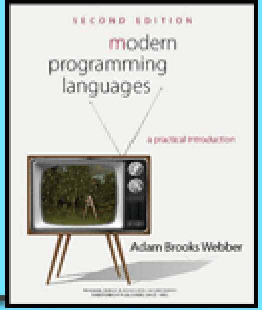


A Programming Language Grammar



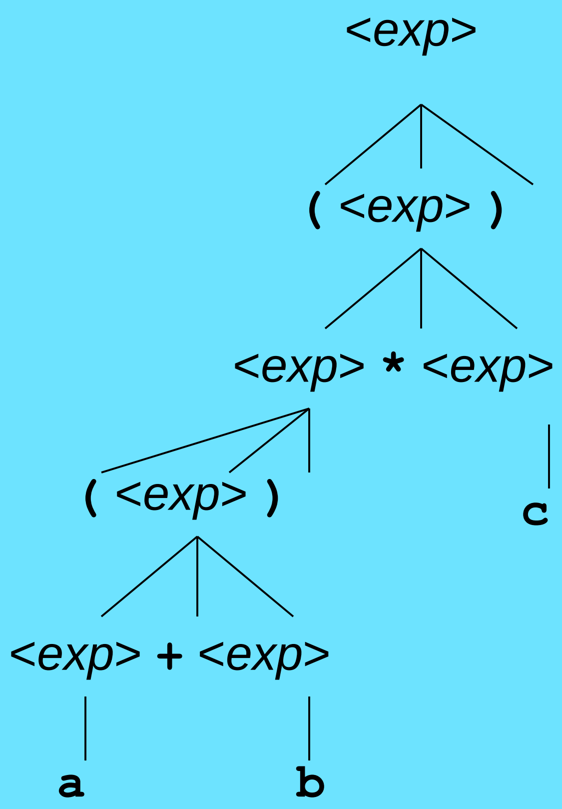
<exp> ::= *<exp>* `+` *<exp>* | *<exp>* `*` *<exp>* | `(` *<exp>* `)`
| `a` | `b` | `c`

- 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`



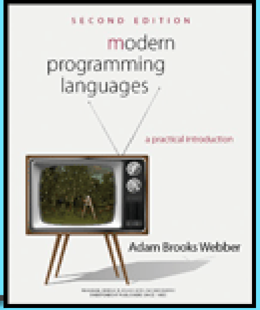
A Parse Tree

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



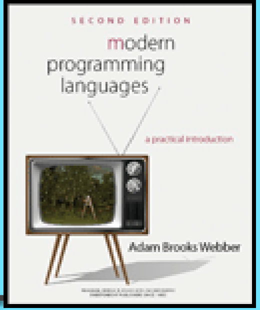
`((a+b)*c)`

Note: Finding a parse tree for a given string (with respect to a given grammar) is called parsing the string.



Outline

- ✧ Grammar and parse tree examples
- ✧ BNF and parse tree definitions
- ✧ Constructing grammars
- ✧ Phrase structure and lexical structure
- ✧ YACC



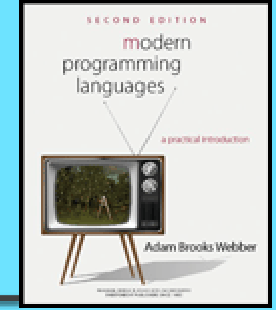
BNF Grammar Definition

✧ A BNF grammar consists of four parts:

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

Dr. BC Note:

terminals of a grammar = tokens returned by the scanner



start symbol

<*S*> ::= <*NP*> <*V*> <*NP*>

a production

<*NP*> ::= <*A*> <*N*>

<*V*> ::= **loves** | **hates** | **eats**

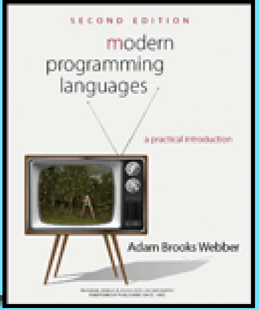
<*A*> ::= **a** | **the**

<*N*> ::= **dog** | **cat** | **rat**

non-terminal symbols

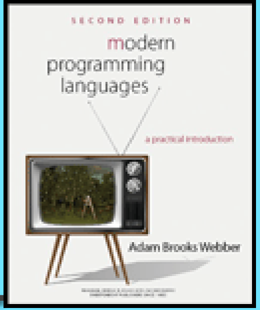
tokens

Definition, Continued



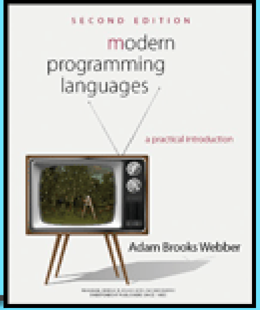
- The *tokens* are the smallest units of syntax
  - Strings of one or more characters of program text
  - They are atomic: not treated as being composed from smaller parts
- The *non-terminal symbols* stand for larger pieces of syntax
  - They are strings enclosed in angle brackets, as in *<NP>*
  - They are not strings that occur literally in program text
  - The grammar says how they can be expanded into strings of tokens

Definition, Continued

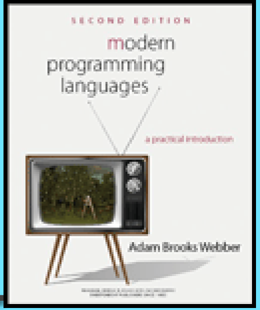


- The *start symbol* is the particular non-terminal that forms the root of any parse tree for the grammar

Definition, Continued



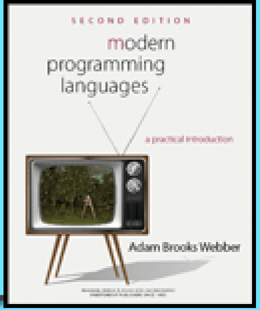
- The *productions* are the tree-building rules
- Each one has a left-hand side, the separator **::=**, and a right-hand side
  - The left-hand side is a single non-terminal
  - The right-hand side is a sequence of one or more things, each of which can be either a token or a non-terminal
- A production gives one possible way of building a parse tree: it permits the non-terminal symbol on the left-hand side to have the things on the right-hand side, in order, as its children in a parse tree



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 `|`

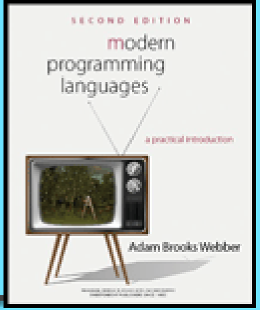
Example



<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**

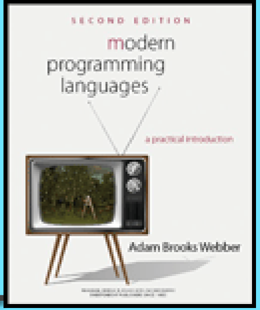


Empty

- 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:

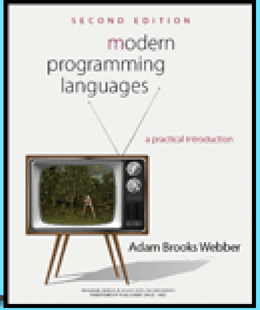
```
<if-stmt> ::= if <expr> then <stmt> <else-part>
<else-part> ::= else <stmt> | <empty>
```

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



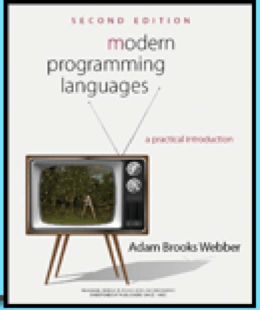
<exp> ::= *<exp>* **+** *<exp>* | *<exp>* ***** *<exp>* | **(** *<exp>* **)**
| **a** | **b** | **c**

Show a parse tree for each of these strings:

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

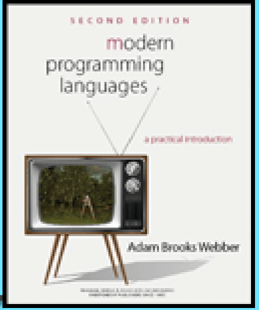
Note: This form for writing grammars is called Backus-Naur Form (BNF). It was developed by John Backus and Peter Naur around 1960.

Compiler Note

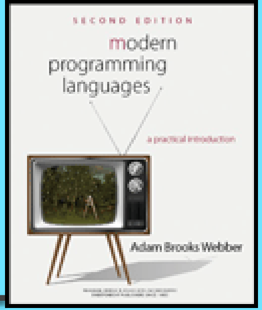


- What we just did is *parsing*: trying to find a parse tree for a given string
- That's what compilers do for every program you try to compile: try to build a parse tree for your program, using the grammar for whatever language you used

Language Definition

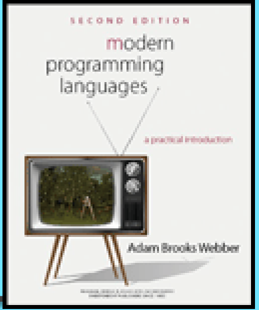


- ✧ We use grammars to define the syntax of programming languages
- ✧ The language defined by a grammar is the set of all strings that can be derived by some parse tree for the grammar
- ✧ As in the previous example, that set is often infinite (though grammars are finite)
- ✧ Constructing grammars is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



Outline

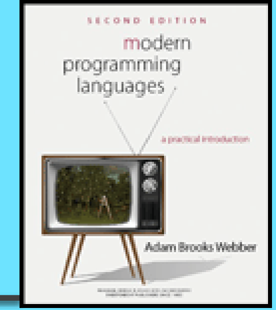
- ✧ Grammar and parse tree examples
- ✧ BNF and parse tree definitions
- ✧ Constructing grammars
- ✧ Phrase structure and lexical structure
- ✧ YACC



Constructing Grammars

- ✧ Most important trick: divide and conquer
- ✧ Example: the language of Java 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

<var-dec> is the start symbol

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

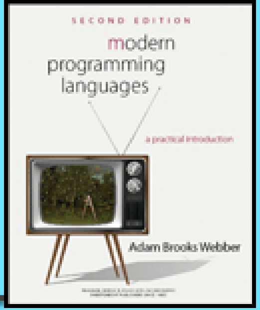
  <var-dec> `::=` *<type-name>* *<declarator-list>* `;`

- Primitive type names are easy enough too:

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

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

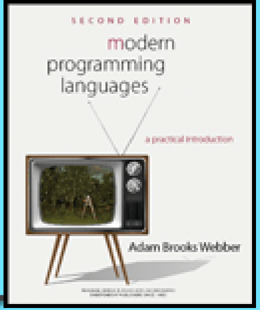
Example, Continued



- 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*>

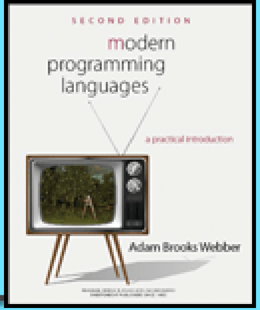
Example, Continued



- That leaves the variables with initializers:

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

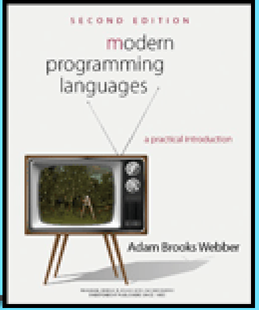
- For full Java, we would need to allow pairs of square brackets after the variable name
- There is also a syntax for array initializers
- And definitions for *<variable-name>* and *<expr>* - but we end here



Outline

- ✧ Grammar and parse tree examples
- ✧ BNF and parse tree definitions
- ✧ Constructing grammars
- ✧ Phrase structure and lexical structure
- ✧ YACC

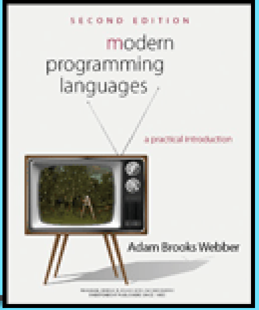
Where Do Tokens Come From?



- 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?

DrBC445 Note: Your parser.l.

Lexical Structure And Phrase Structure



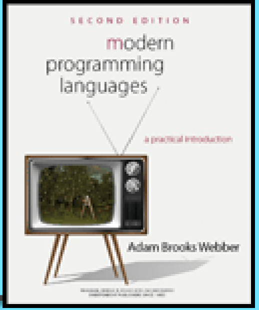
- We need to define *lexical structure*: how a text file is divided into tokens

DrBC445 Note: Your parser.l.

- And also *phrase structure*: how a program is built from a sequence of tokens

DrBC445 Note: Your parser.y.

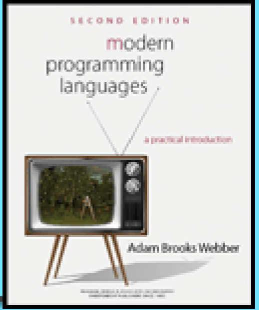
One Grammar For Both



- You could do it all with one grammar by using characters as the only tokens
- Not done in practice: things like white space and comments would make the grammar too messy to be readable

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

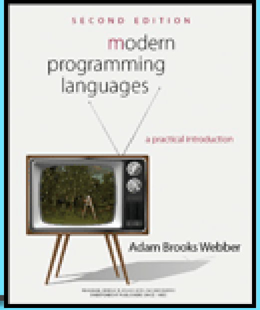
Separate Grammars



- ✧ Usually there are two separate grammars
  - One says how to construct a sequence of tokens from a file of characters - lexical structure → parser.l
  - One says how to construct a parse tree from a sequence of tokens - phrase structure → parser.y

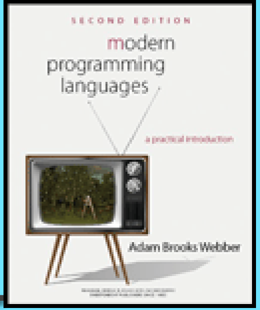
<program-file> ::= <end-of-file> | <element> <program-file>
<element> ::= <token> | <one-white-space> | <comment>
<one-white-space> ::= <space> | <tab> | <end-of-line>
<token> ::= <identifier> | <operator> | <constant> | …

Outline



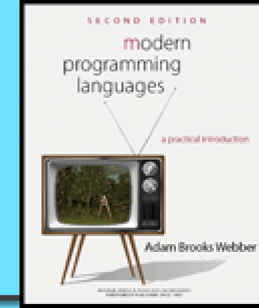
- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- YACC

YACC Intro



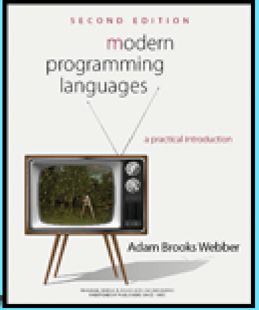
- Let's make a parser for this language in YACC
  - <B> ::= <S> <B> | <S>
  - <S> ::= <NP> <V> <NP>
  - <NP> ::= <A> <N>
  - <V> ::= loves | hates|eats
  - <A> ::= a | the
  - <N> ::= dog | cat | rat
- We need to create two files:
  - myFile.l - Specify all pattern matching rules for lex () and
  - myFile.y - grammar rules for yacc ().

dog.l - preamble



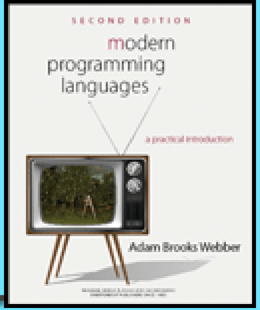

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define YYSTYPE
#include "y.tab.h" // generated via yacc -d yacc2.y

%}
/* This tells flex to read only one input file */
%option noyywrap
/* Needed if you do not compile with -ll */
```

dog.l – continued: Pattern matching for tokens

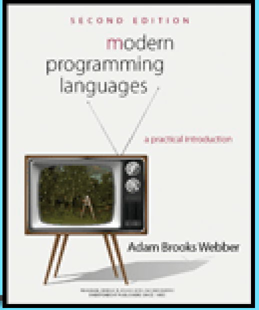



```
%%
[ \n]+          /* eat whitespace */;
cat             {return CAT;}
dog             {return DOG;}
rat             {return RAT;}
a               {return A;}
the             {return THE;}
loves           {return LOVES;}
hates           {return HATES;}
eats            {return EATS;}
%%
```

dog.y - Preamble




```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int yylex();
%}

%union {             /*lists all possible types for values associated with parts of the grammar and gives each a field-name */
char *tok;
}
```



dog.y - continued

/* Note on comments: Precede with a space or tab to shove it to C code */

%type <tok> B S NP V R N /* These are the non-terminal listed below */

%type <tok> CAT DOG RAT A THE LOVES HATES EATS

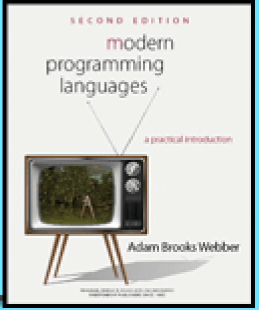
%token CAT DOG RAT A THE LOVES HATES EATS

%%

<B> ::= <S> <B> | <S>
<S> ::= <NP> <V> <NP>
<NP> ::= <A> <N>
<V> ::= loves | hates|eats
<A> ::= a | the
<N> ::= dog | cat | rat

Note: Renamed A to R since A is also a token

dog.y – continued: Productions




```
B : S          {printf("Book\n");}
  | S B        {printf("Book Next Sentence\n");}
  ;

S : NP V NP    {printf("Sentence\n");}
  ;

NP : R N       {printf("Noun Phrase\n");}
  ;
```

<B> ::= <S> <B> | <S>
<S> ::= <NP> <V> <NP>
<NP> ::= <A> <N>
<V> ::= loves | hates|eats
<A> ::= a | the
<N> ::= dog | cat | rat

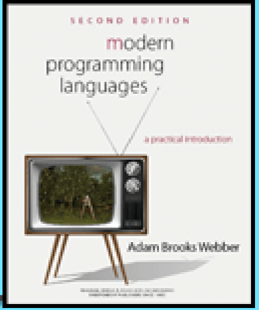
This will be C Code

<exp> ::= <exp> **+** <exp>

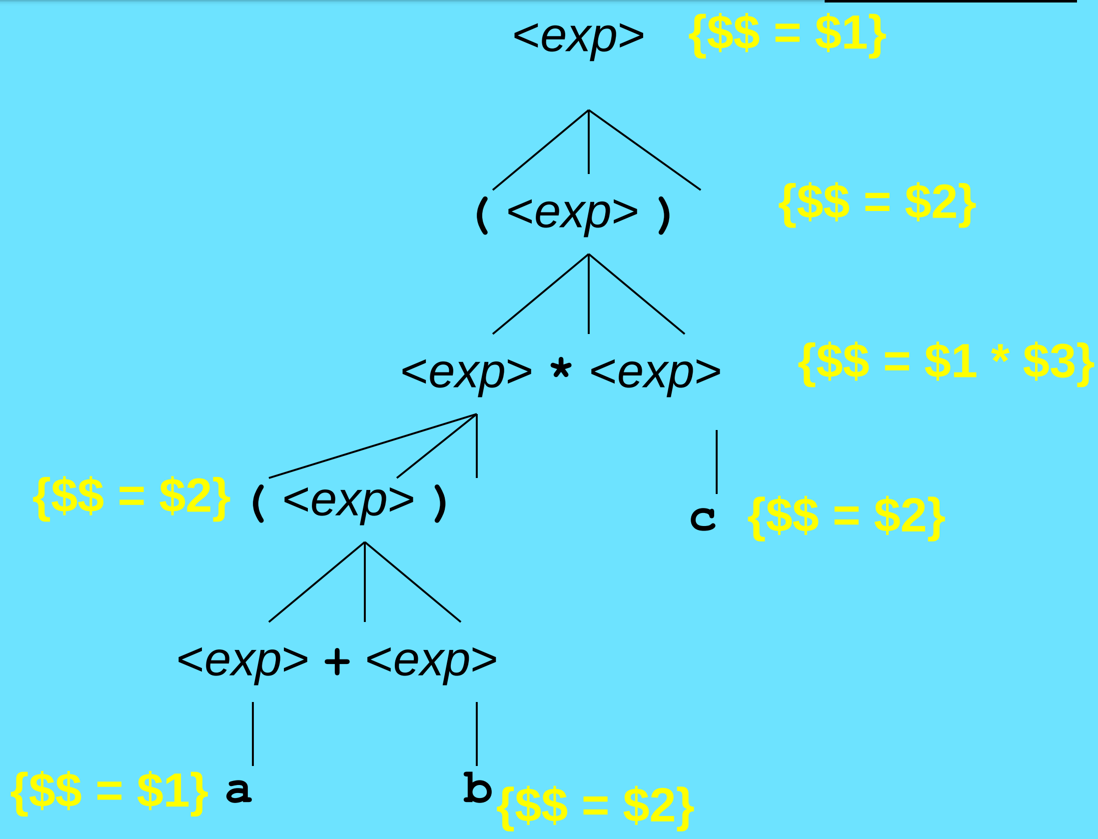
| <exp> ***** <exp>

| **(** <exp> **)**

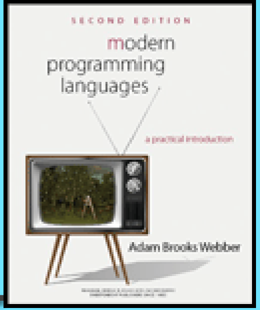
| **a** | **b** | **c**



((a+b)*c)




```
S : E        {printf("%f\n", $1);}
  ;
E : E '+' T   {$$ = $1 + $3;}
  | T         {$$ = $1;}
  ;
T : T '*' F   {$$ = $1 * $3;}
  | F         {$$ = $1;}
  ;
F : '(' E ')' {$$ = $2;}
| NUM       {$$ = $1;}
  ;
```

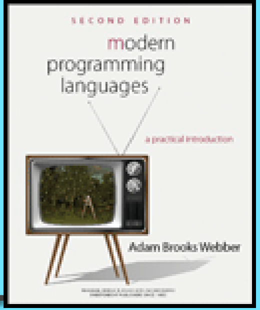


dog.y – continued Productions

```
V : LOVES       {printf("Verb: loves\n");}
  | HATES       {printf("Verb: eats\n");}
  | EATS        {printf("Verb: eats\n");}
  ;

R : A           {printf("Artical: a\n");}
  | THE         {printf("Artical: the\n");}
  ;

N : CAT         {printf("Noun: cat\n");}
  | DOG         {printf("Noun: dog\n");}
  | RAT         {printf("Noun: rat\n");}
  ;
```

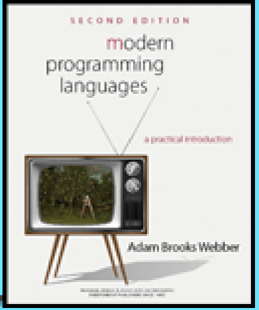
```
<B> ::= <S> <B>  | <S>
<S> ::= <NP> <V> <NP>
<NP> ::= <A> <N>
<V> ::= loves | hates|eats
<A> ::= a | the
<N> ::= dog | cat | rat
```

dog.y - continued



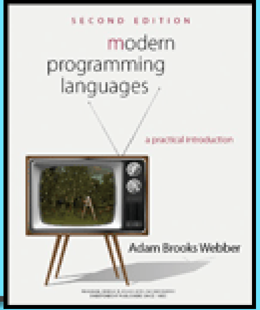

```
%%

void yyerror(char *msg) {
   fprintf(stderr, "%s\n", msg);
   exit(1);
}

int main() {
   yyparse();
   return 0;
}
```

Compiling



- yacc -d dog.y
  - This creates tab.c from dog.y
- gcc -c y.tab.c -o y.tab.o
  - This creates tab.o
- lex dog.l
  - This creates lex.yy.c from dog.l
- gcc -c lex.yy.c -o lex.yy.o
  - This creates lex.yy.o
- gcc lex.yy.o y.tab.o
  - Putting it all together into an executable. (You can use the –o option)

Running



- ./a.out < inputFile.txt
- Or just run ./a.out and input on the command line.
  - Use ctrl-d to finish