Defining Program Syntax

Syntax And Semantics

- **Programming language syntax**: how programs look, their form and structure
  - Syntax is defined using a formal grammar
- **Programming language semantics**: what programs do, their behavior and meaning
  - Semantics is harder to define—more on this in Chapter 23

Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms

An English Grammar

A sentence <S> is a noun phrase <NP>, a verb <V>, and a noun phrase <NP>.

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

A noun phrase <NP> is an article <A> and a noun <N>.

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

A verb <V> is…

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

An article <A> is…

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

A noun <N> 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*
- <*S*> at the root of the tree
- The grammar's rules define how children can be added at any point in the tree
- For instance,

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

defines nodes <*NP*>, <*V*>, and <*NP*>, in that order, as children of <*S*>

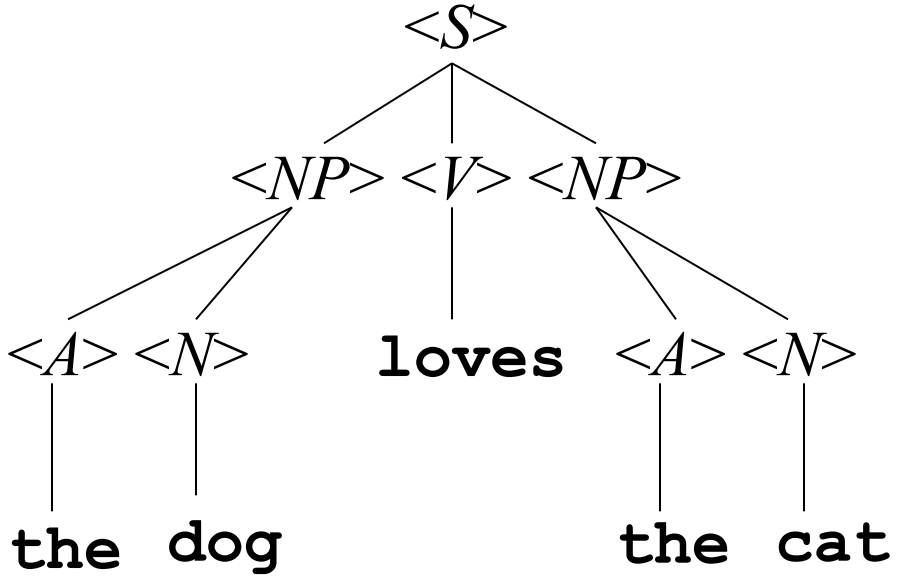
Parse Derivation

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

One **derivation** that **<S>= <u>the dog loves the cat</u>** is produced by the grammar rules:

```
<S> = <NP>     <V>        <NP>
    = <A> <N>  <V>        <NP>
    = <A> <N>  <V>    <A> <N>
    = <A> <N> loves   <A> <N>
    = the <N> loves   <A> <N>
= the dog loves <A> <N>
= the dog loves the <N>
    = the dog loves the cat
```

Parse Tree: the dog loves the cat

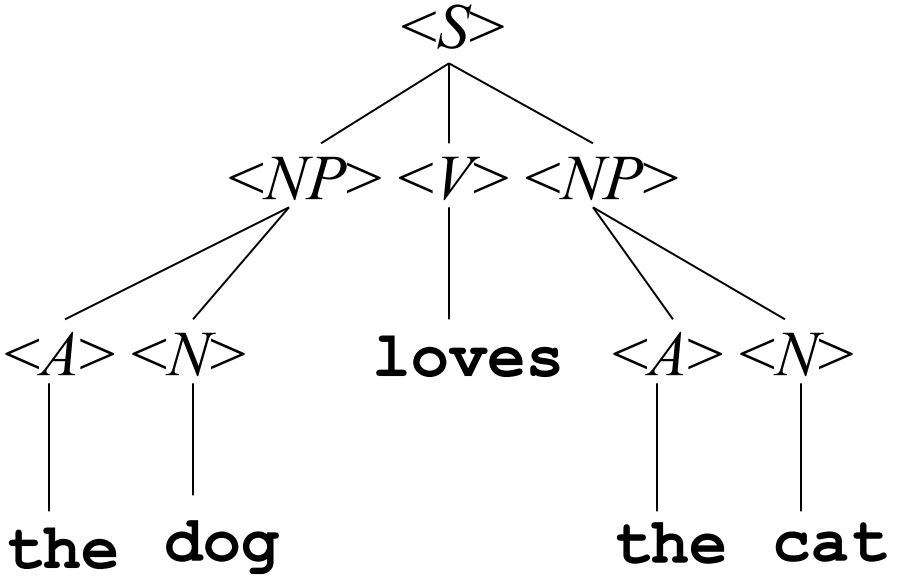


<S> = <NP> <V> <NP>
= <A> <N> loves <A> <N>
= the dog loves the cat

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

Exercise 1

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



1. Which of the following are valid <S>?
   - the dog hates the dog
   - dog loves the cat
   - loves the dog the cat
2. Parse:
   - a cat eats the rat
   - the dog loves cat

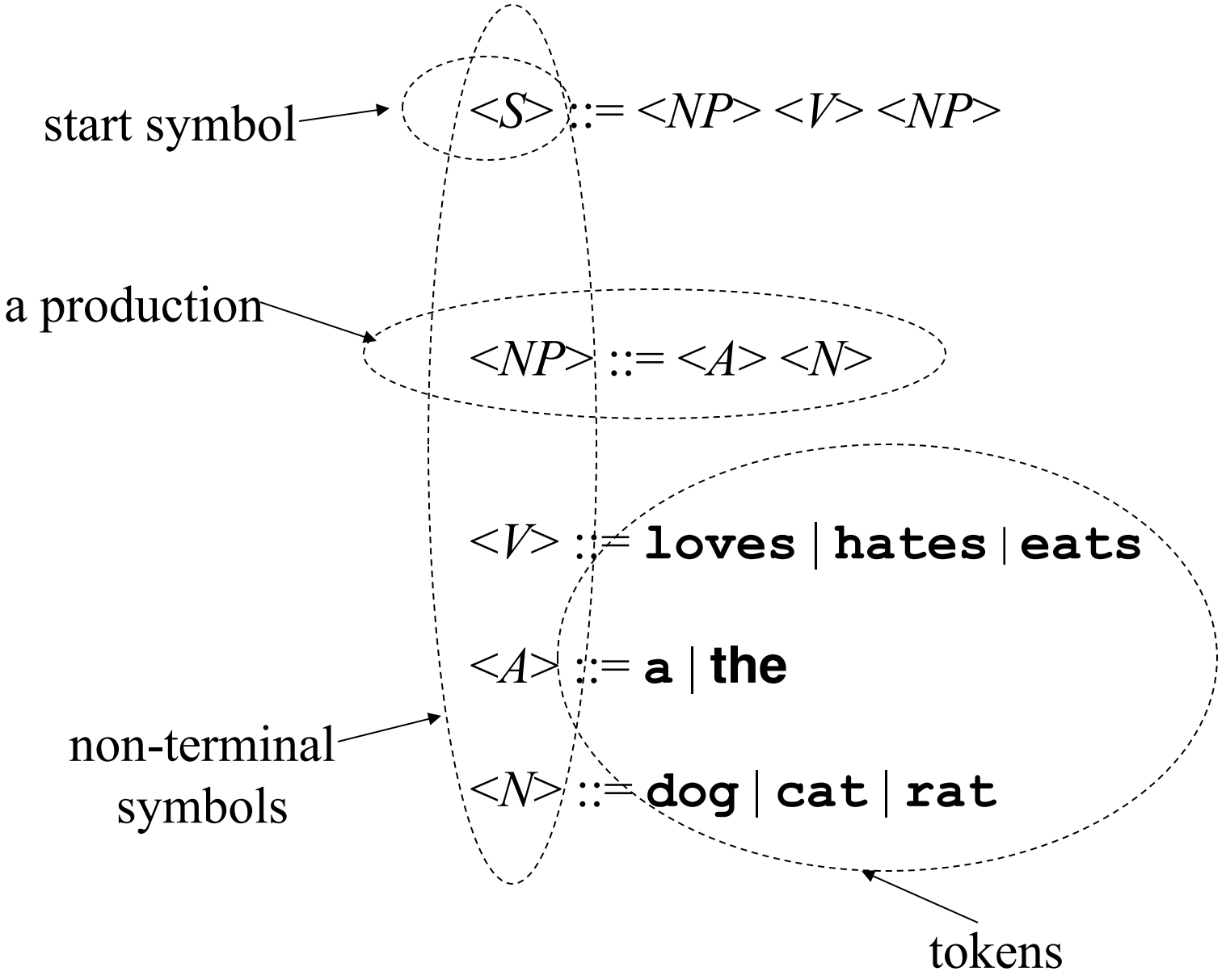
Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms

BNF Grammar Definition

- *Backus Naur Form* grammar consists of four parts:
  - The set of *tokens*
  - The set of *non-terminal symbols*
  - The *start symbol*
  - The set of *productions*

BNF Grammar Definitions Explained



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
- 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 symbols on the right-hand side, in order, as its children in a parse tree

Alternatives (OR)

- When there is more than one production with the same left-hand side, an abbreviated form can be used
- In BNF grammar:
  - Gives the left-hand side (symbol),
  - 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 non-terminal <*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*>

Grammar Parse Derivation

- Begin with a start symbol
- Choose a production with start symbol on left-hand side
- Replace start symbol with the right-hand side of that production

1. Choose a non-terminal *S* in resulting string
2. Choose a production *P* with non-terminal *S* on its left-hand side
3. Replace *S* with the right-hand side of *P*
4. Repeat process until no non-terminals remain.

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

a cat eats the rat

```
<S> = <NP>     <V>       <NP>
    = <A> <N>  <V>       <NP>
    = <A> <N>  <V>  <A>  <N>
    = <A> <N>  eats <A>  <N>
    =  a  <N>  eats <A>  <N>
  = a  cat eats <A> <N>
  = a  cat eats the <N>
    =  a  cat eats the rat
```

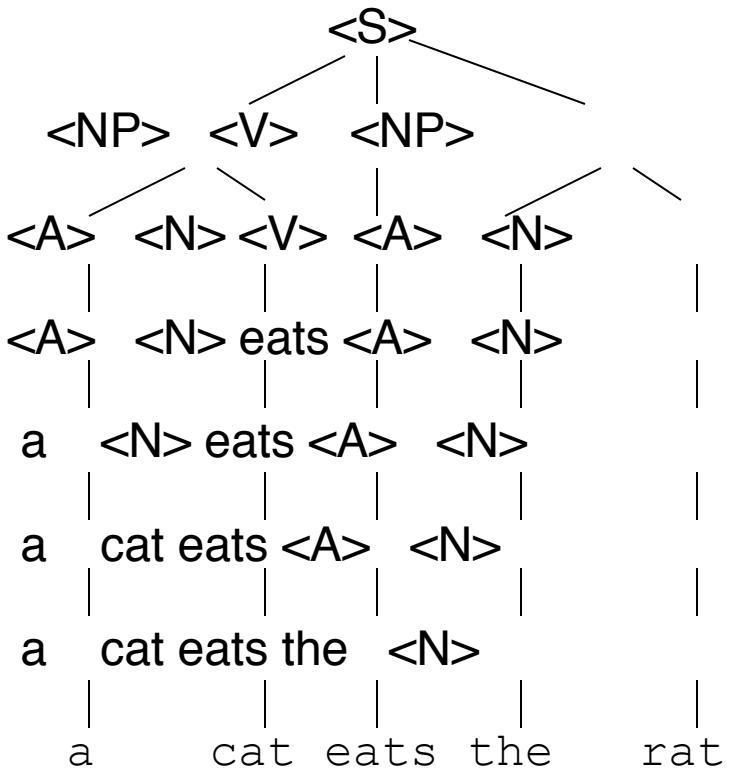
Parse Trees

- 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

```
<S> = <NP>     <V>      <NP>
    = <A> <N>  <V>      <NP>
    = <A> <N>  <V>  <A> <N>
    = <A> <N>  eats <A> <N>
    =  a  <N>  eats <A> <N>
  = a  cat eats <A> <N>
  = a  cat eats the <N>
    =  a  cat eats the rat
```

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

<S> = a cat eats the rat



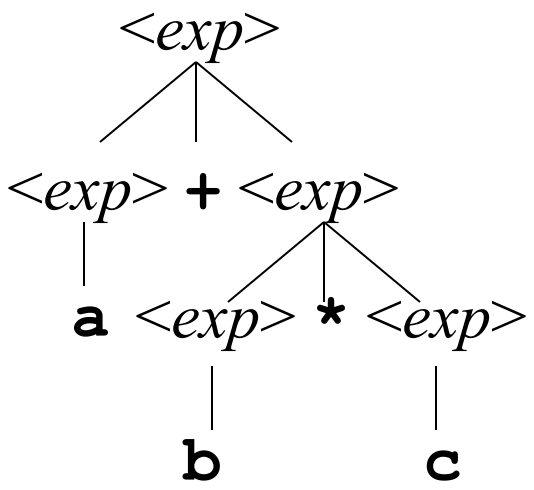
A Programming Language Grammar

<*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 **a**,
  - or **b**,
  - or **c**

Parse and Parse Tree: a+b*c

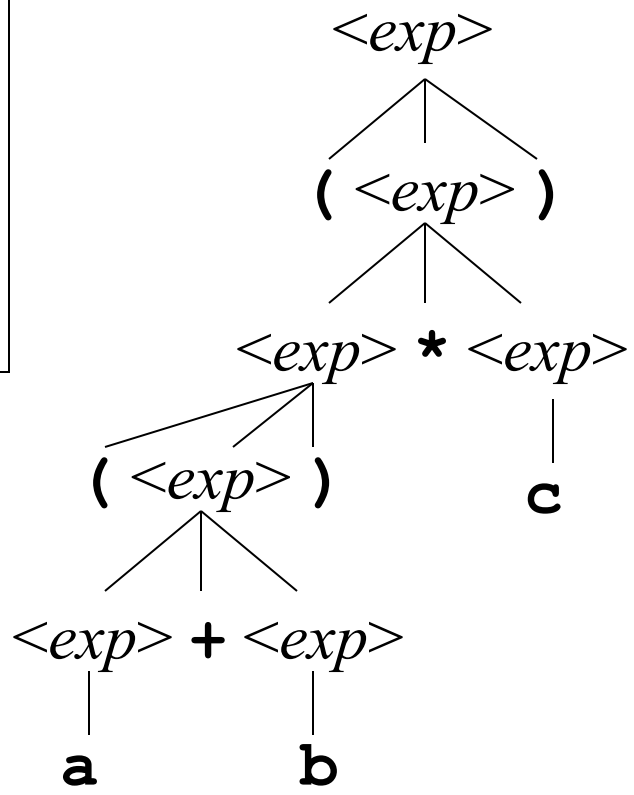
```
<exp> = <exp> + <exp>
      = a + <exp>
      = a + <exp> * <exp>
      = a +   b   *   c
```



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

Parse and Parse Tree: ((a+b)*c)

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



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

Exercise 2

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

Give the parse tree for each of these strings:

a. a+b

b. a*b+c

c. (a+b)*c

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...

Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms

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 optionally be followed by an initializer:

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

Example, Continued

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

- 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:

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

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

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

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

Example, Continued

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

- That leaves the variables with initializers:

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

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

<*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*>

Grammar Construction Example

Construct a grammar in BNF for each language:

1. <digit> as a character 0-9.

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

2. <unsigned> as the set of all strings with one or more <digit>. Note the left-recursion.

```
<unsigned> ::= <digit> | <unsigned> <digit>
```

3. <signed> as the set of all strings starting with – or + and followed by an <unsigned>.

```
<signed> ::= +<unsigned> | -<unsigned>
```

Grammar Construction Example

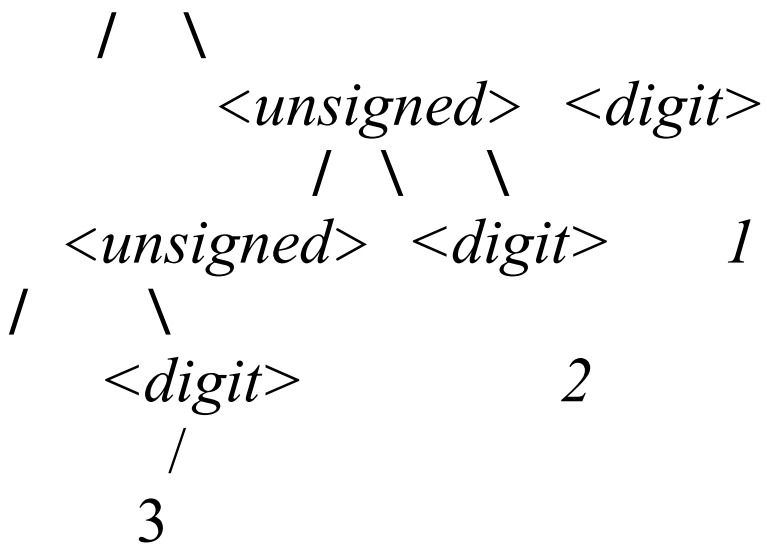
1. <digit> as a character 0-9.

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

2. <unsigned> as the set of all strings with one or more <digit>. Note the left-recursion.

<unsigned> ::= *<digit>* | *<unsigned>* *<digit>*

Parse 321



Exercise 3

<*digit*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*unsigned*> ::= <*digit*> | <*unsigned*> <*digit*>

<*signed*> ::= +<*unsigned*> | -<*unsigned*>

Construct a grammar in BNF for each language:

1. <integer> as the set of all strings of <signed> or <unsigned>.
2. <decimal> as the set of all strings of <integer> followed by a '.' and optionally followed by an <unsigned>.
3. <2or3digits> as the set of all strings of two or three <digit>.
4. <2's> as the set of all strings of one or more 2's.
5. <1+2's> as the set of all strings beginning with '1' and followed by any number of 2's.
6. <2's+1> as the set of all strings beginning with any number of 2's and followed by a '1'.
7. <AdigitBs> as the set of all strings beginning with 'A' and optionally followed by any number of <digit> or 'B'.

Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms

Where Do Tokens Come From?

<*digit*> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*unsigned*> ::= <*digit*> | <*unsigned*> <*digit*>

- Tokens are pieces of program text that we choose not 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

- *Phrase structure*: how a program is built from a sequence of tokens

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

- *Lexical structure*: how tokens are built from a sequence of characters

<*digit*> `::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

<*unsigned*> `::=` <*digit*> | <*unsigned*> <*digit*>

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
  - One says how to construct a parse tree from a sequence of tokens

<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>* | …

Separate Compiler Passes

- The *scanner* reads the input file and divides it into tokens according to the first grammar
- The scanner discards white space and comments
- The *parser* constructs a parse tree (or at least goes through the motions—more about this later) from the token stream according to the second grammar

Exercise 4

<space> ::=
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<unsigned> ::= <digit> | <unsigned> <digit>
<signed> ::= +<unsigned> | -<unsigned>
<integer> ::= <signed> | <unsigned>
<decimal> ::= <integer>.<unsigned> | <integer> .
<operator> ::= + | == | =
<identifier> ::= x | y
<constant> ::= <integer> | <decimal>
<keyword> ::= if | then | endif

List the *scanner* output from the following:

if x == 3.14 then y = x + y endif

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
- Almost all modern languages are *free-format*: column positions are ignored

Outline

- Grammar and parse tree examples
- BNF and parse tree definitions
- Constructing grammars
- Phrase structure and lexical structure
- Other grammar forms

Other Grammar Forms

- BNF variations
- EBNF variations
- Syntax diagrams

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

EBNF Variations

- Additional syntax to simplify some grammar chores:
  - {x} or x* to mean zero or more repetitions of x
  - x+ to mean one or more repetitions of x
  - [x] to mean x is optional (i.e. x | <*empty*>)
  - ( ) for grouping
  - | anywhere to mean a choice among alternatives
  - Quotes around tokens, if necessary, to distinguish from all these meta-symbols

EBNF Examples

<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> [**else** <*stmt*>]

<*stmt-list*> **::=** {<*stmt*> **;**}

<*thing-list*> **::=** { (<*stmt*> | <*declaration*>) **;**}

<*digit*> **::=** 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<*unsigned*> **::=** <*digit*>+

<*signed*> **::=** (+|-)<*unsigned*>

- Anything that extends BNF this way is called an Extended BNF: EBNF
- There are many variations

Exercise 5

Construct a grammar in EBNF for each language:

1. <unsigned> as the set of all strings with one or more <digit>.

2. <signed> as the set of all strings starting with – or + and followed by an <unsigned>.

3. <integer> as the set of all strings of <signed> or <unsigned>.

4. <decimal> as the set of all strings of <integer> followed by a '.' and optionally followed by an <unsigned>.

EBNF Extensions

{x} or x* to mean zero or more repetitions of x

x+ to mean one or more repetitions of x

[x] to mean x is optional (i.e. x | <*empty*>)

( ) for grouping

| anywhere to mean a choice among alternatives

Exercise 5 continued

{x} or x* to mean zero or more repetitions of x

x^+ to mean one or more repetitions of x

[x] to mean x is optional (i.e. x | <*empty*>)

( ) for grouping

| anywhere to mean a choice among alternatives

6. <identifier> as the set of all strings starting with <alpha> and followed by zero or more <alpha> or <digit>.

7. <1+2's> as the set of all strings beginning with '1' and followed by any number of 2's.

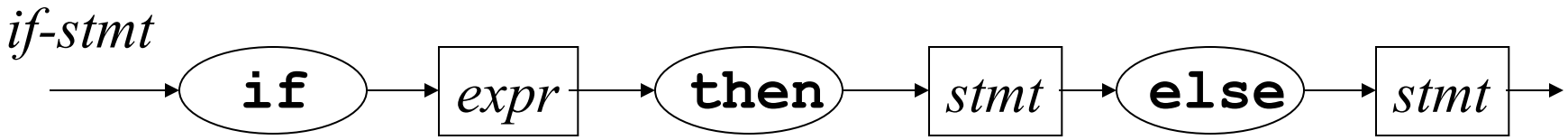
8. <AdigitBs> as the set of all strings beginning with 'A' and optionally followed by any number of <digit> or 'B'.

9. Scientific notation (e.g. 1.2E-13)

Syntax Diagrams

- Syntax diagrams ("railroad diagrams")
- Start with an EBNF grammar
- A simple production is just a chain of boxes (for nonterminals) and ovals (for terminals):

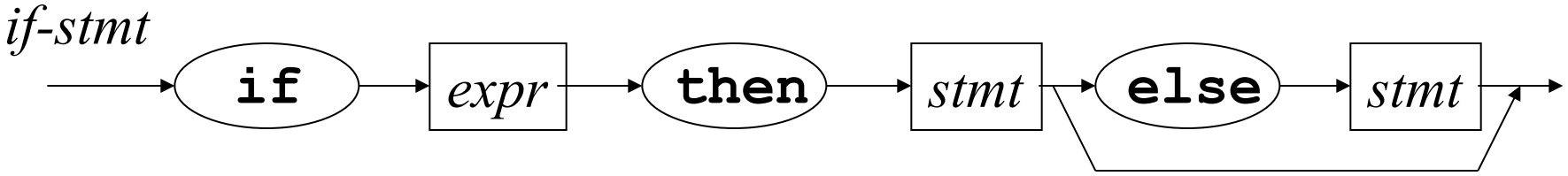
<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> **else** <*stmt*>



Bypasses

- Square-bracket pieces from the EBNF get paths that bypass them

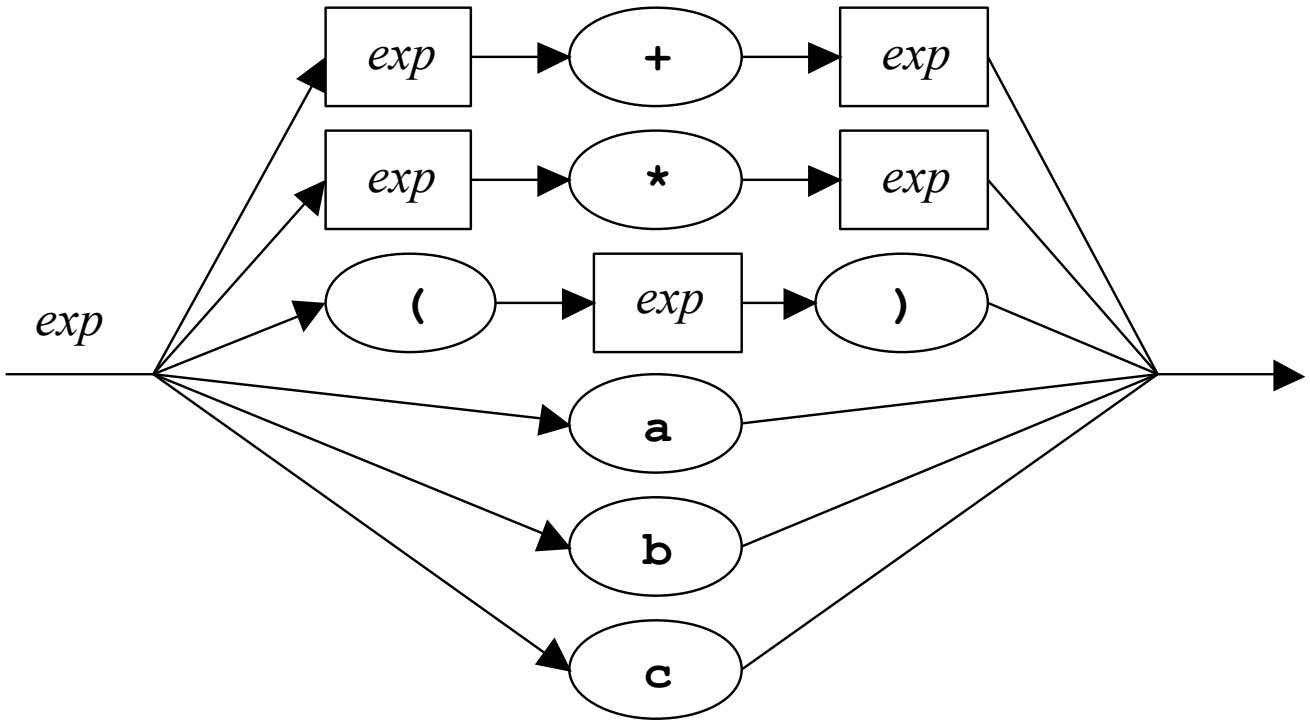
<*if-stmt*> **::=** **if** <*expr*> **then** <*stmt*> [**else** <*stmt*>]



Branching

- Use branching for multiple productions

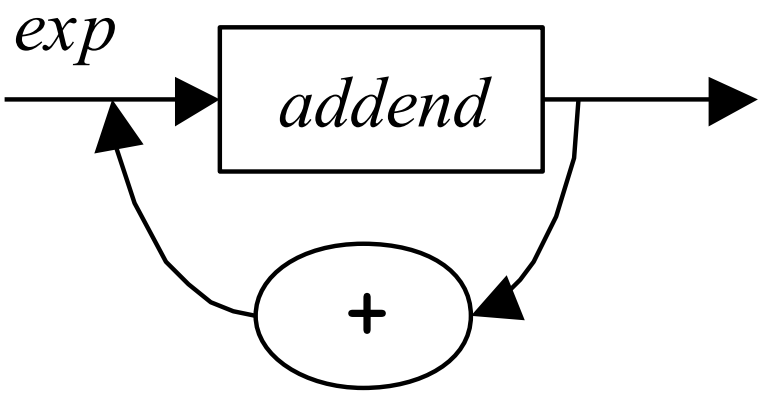
<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**



Loops

- Use loops for EBNF curly brackets

<*exp*> ::= <*addend*> { + <*addend*> }



Syntax Diagrams, Pro and Con

- Easier for people to read casually
- Harder to read precisely: what will the parse tree look like?
- Harder to make machine readable (for automatic parser-generators)

Formal Context-Free Grammars

- In the study of formal languages and automata, grammars are expressed in yet another notation:

$S \rightarrow aSb \mid X$	S is a string of symbols **`a S b`** or **`X`**.
$X \rightarrow cX \mid \in$	X is a string of symbols **`c X`** or empty

- These are called *context-free grammars* because children of a node only depend on that node's non-terminal symbol; not on the context of neighboring nodes in the tree. Simpler to define and compile.
- Context sensitive language elements include scope but is not generally part of a grammar.
- Other kinds of grammars are also studied: *regular grammars* (weaker), *context-sensitive grammars* (stronger), etc.

Many Other Variations

- BNF and EBNF ideas are widely used
- Exact notation differs, in spite of occasional efforts to get uniformity
- But as long as you understand the ideas, differences in notation are easy to pick up

Example

WhileStatement:
while (*Expression*) *Statement*

DoStatement:
do *Statement* while (*Expression*) ;

ForStatement:
for ($ForInit_{opt}$; $Expression_{opt}$; $ForUpdate_{opt}$)
Statement

[from *The Java™ Language Specification*,
James Gosling et. al.]

Conclusion

- We use grammars to define programming language syntax, both lexical structure and phrase structure
- Connection between theory and practice
  - Two grammars, two compiler passes
  - Parser-generators can write code for those two passes automatically from grammars

Conclusion, Continued

- Multiple audiences for a grammar
  - Novices want to find out what legal programs look like
  - Experts—advanced users and language system implementers—want an exact, detailed definition
  - Tools—parser and scanner generators—want an exact, detailed definition in a particular, machine-readable form