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 later

Grammar and Parse Trees

An English Grammar

- A sentence <S> is a noun phrase <NP>, a verb <V>, and a noun phrase <NP>.
- A noun phrase <NP> is an article <A> and a noun <N>.
- A verb <V> is . . .
- An article <A> is . . .
- A noun <N> is . . .

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

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

<V> ::= loves | hates | eats

<A> ::= a | the

<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 the sequence of nodes <NP>, <V>, and <NP> as children of <S>

Parse Derivation

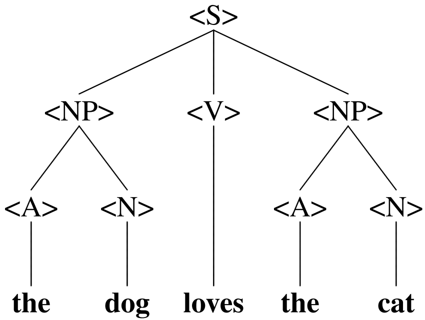
Grammar

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

Example (Derive <S> = **the dog loves the cat**)

```
<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



Example

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

Exercise

Grammar

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

- Which of the following are valid <S>?
  - the dog hates the dog
  - dog loves the cat
  - loves the dog the cat
- Draw a parse tree for:
  - a cat eats the rat
  - the dog loves cat

BNF

BNF Grammar Definition

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

BNF Explanation

- tokens: dog, cat, ...
- non-terminal symbols: <V>, <N>, ...
- start symbol: <S>
- a production: <NP> ::= <A> <N>

Example

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

More BNF Definitions

- 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 a single non-terminal that forms the root of every parse tree for the grammar

More BNF Definitions

- 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

Alternative Productions

- 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

Example (Production)

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

Example (Equivalent Productions)

```
<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:

Example

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

Grammar Parse Derivation

1. Begin with a start symbol
2. Choose a production *P* with non-terminal *N* on its left-hand side
3. Replace *N* with the right-hand side of *P*
4. Choose a non-terminal *N* in resulting string
5. If non-terminals remain, GOTO step 2

Example

```
<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
```

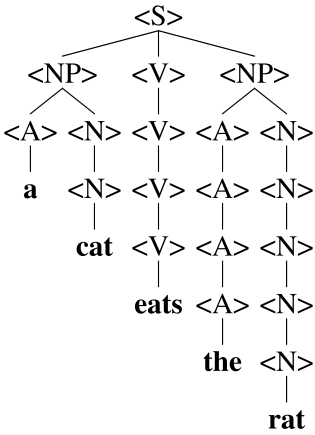
Parse Trees

- 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

Example

Example (Grammar)

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



A Programming Language Grammar

- 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`

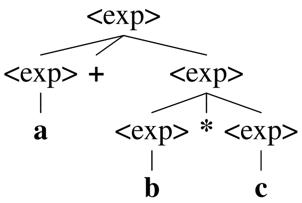
Example

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

Parse and Parse Tree: a+b*c

Example

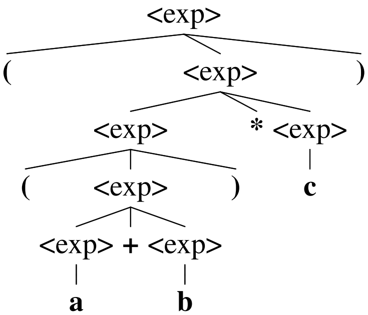
```
<exp> = <exp> + <exp>
      =   a   + <exp>
      =   a   + <exp> * <exp>
      =   a   +   b   *   c
```



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

Example

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



Exercise

Grammar

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

- Give the parse tree for each of these strings:
  - a+b
  - a*b+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
- Grammars are designed to be non-ambiguous: for each string, there is at most one valid parse tree
- There are efficient parsing algorithms for this specific purpose

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

Constructing Grammars

- The 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

Example (Java declarations)

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

Java Example

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

- Defining a declaration is 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:
  - <var-dec> ::= <type-name> <declarator-list> ;
  - <declarator-list> ::= <declarator> | <declarator> , <declarator-list>

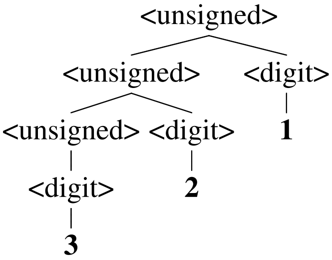
Example, Continued

- 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

1. Construct a grammar in BNF for each language:
2. `<digit>` as a character 0-9.
   - `<digit> ::= 0|1|2|3|4|5|6|7|8|9`
3. `<unsigned>` as the set of all strings with one or more `<digit>`. Note the left-recursion.
   - `<unsigned> ::= <digit> | <unsigned> <digit>`
4. `<signed>` as the set of all strings starting with $-$ or $+$ and followed by an <unsigned>.
   - `<signed> ::= +<unsigned> | -<unsigned>`

Parse 321 as `<unsigned>`



Exercise

- Construct production rules in BNF for each specification:
- `<integer>` as any strings of `<signed>` or `<unsigned>`.
- `<decimal>` as any strings of `<integer>` followed by a '.' and optionally followed by an `<unsigned>`.
- `<2or3digits>` as any strings of two or three `<digit>`.

Example

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

Exercise

- Construct production rules in BNF for each specification:
- `<2's>` as any strings of one or more 2's.
- `<1+2's>` as any strings beginning with '1' and followed by any number of 2's.
- `<2's+1>` as any strings beginning with any number of 2's and followed by a '1'.
- `<AdigitBs>` as any strings beginning with 'A' and optionally followed by any number of `<digit>` or 'B'.

Example

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

Structure

Where Do Tokens Come From?

- Tokens are pieces of program text that we think of as atomic and holding specific meaning
- 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
- *Lexical* structure: how tokens are built from a sequence of characters

Example (Phrase Structure)

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

Example (Lexical Structure)

```
<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

Example

```
<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

Example

```
<prog-file> ::= <end-of-file> | <element> <prog-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 lexical 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 language grammar

Exercise

Lexical Grammar

```
<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
```

What is the *scanner* output for `if x == 3.14 then y = x + y endif` ?

Early Languages

- 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

Early Languages

- 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
- Early dialects of Fortran, Cobol, and Basic
- Almost all modern languages are *free-format*: column positions are ignored

Grammar Forms

BNF Variations

- Some use $\rightarrow$ 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

Example

```
<if-stmt> ::= if <expr> then <stmt> [else <stmt>]
<stmt-list> ::= {<stmt> ;}
<thing-list> ::= { (<stmt> | <declaration>) ;}
<unsigned> ::= <digit>+
<signed> ::= (+|-)<unsigned>
```

- Anything that extends BNF this way is called an Extended BNF: EBNF

Formal Context-Free Grammars

- In the study of formal languages and automata, grammars are expressed in yet another notation
- 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.
- Context sensitive language elements include scope but is not generally part of a grammar.
- Other kinds of grammars exist: *regular* grammars (weaker), *context-sensitive* grammars (stronger), etc.

$S \rightarrow aSb|X$ S is a string of symbols a S b or X

$X \rightarrow cX|\varepsilon$ X is a string of symbols c X or empty

Example

Example

Java Language Specification

```
WhileStatement:
  while ( Expression ) Statement

DoStatement:
  do Statement while ( Expression ) ;

ForStatement:
  for ( ForInitopt ; Expressionopt ; ForUpdateopt)
    Statement
```

Conclusions

- 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-generator* programs can write code for those two passes automatically from grammars

Audiences

- 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