DEFINING PROGRAM SYNTAX

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—more on this in Chapter 23*

A2.1 – Language Understanding [5 minutes]

- How do we understand human languages?
- What is a grammar?
- What is a syntax?
- What is a semantic?

A2.2 – Syntax Checking [5 minutes]

- How do we know whether the following sentences are syntactically correct or incorrect?
  - *The dog loves the cat*
  - *A rat lives in the house*
  - *I kicks dog that*
  - *Love you I do*
  - *We school go to*

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 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`

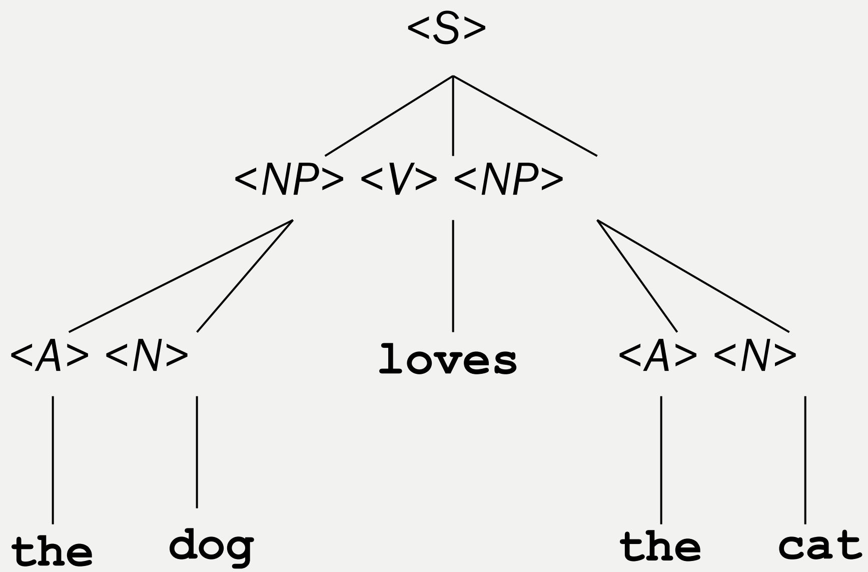
How The Grammar Works

- 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 Parse Tree



A2.3 – Grammar [10 minutes]

- Design a grammar that captures the following code fragments
  - *a + b*
  - *c * a / b*
  - *c + b - a*

A Programming Language Grammar

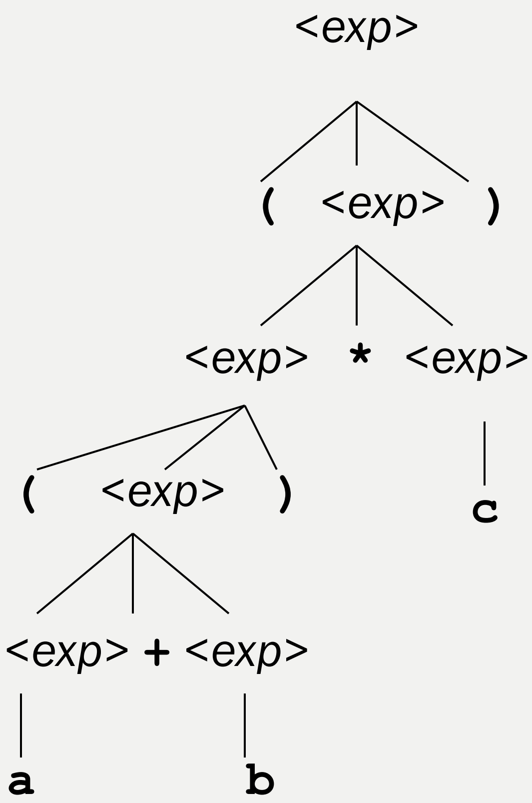
<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> **`*`** <*exp*> | **(** <*exp*> **)**
| **`a`** | **`b`** | **`c`**

- 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`**

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

A Parse Tree

((a+b)*c)



Outline

- BNF and parse tree definitions

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

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*

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 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

A2.4 – Parse Tree [5 minutes]

<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...

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 be followed by an initializer:

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

A2.5 – Constructing Grammar [10 minutes]

- Construct a grammar that captures Java declarations

```
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*>

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?

- 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

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

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

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} to mean zero 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

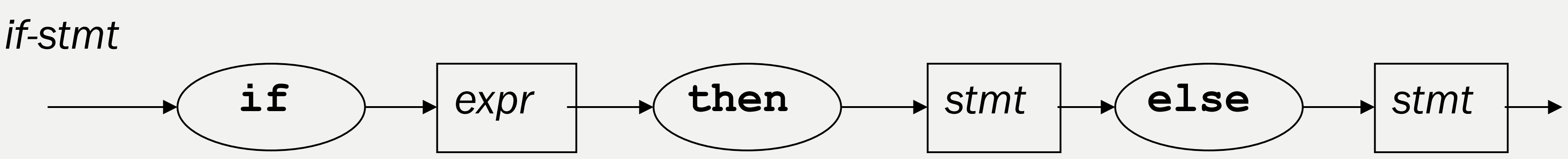
```
<if-stmt> ::= if <expr> then <stmt> [else <stmt>]

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

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

<mystery1> ::= a[1]

<mystery2> ::= 'a[1]'
```

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

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):

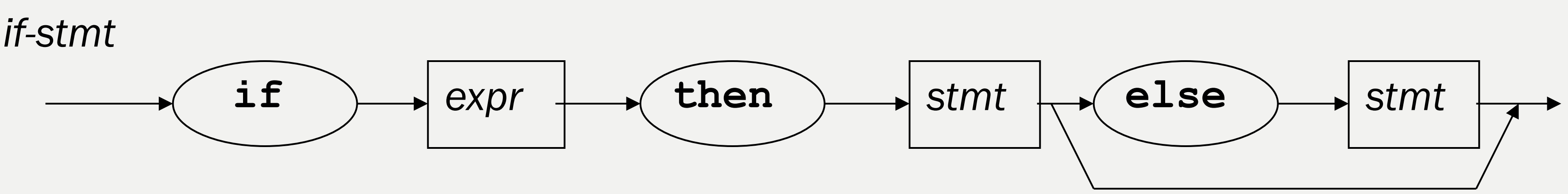
<if-stmt> `::=` **`if`** *<expr>* **`then`** *<stmt>* **`else`** *<stmt>*



Bypasses

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

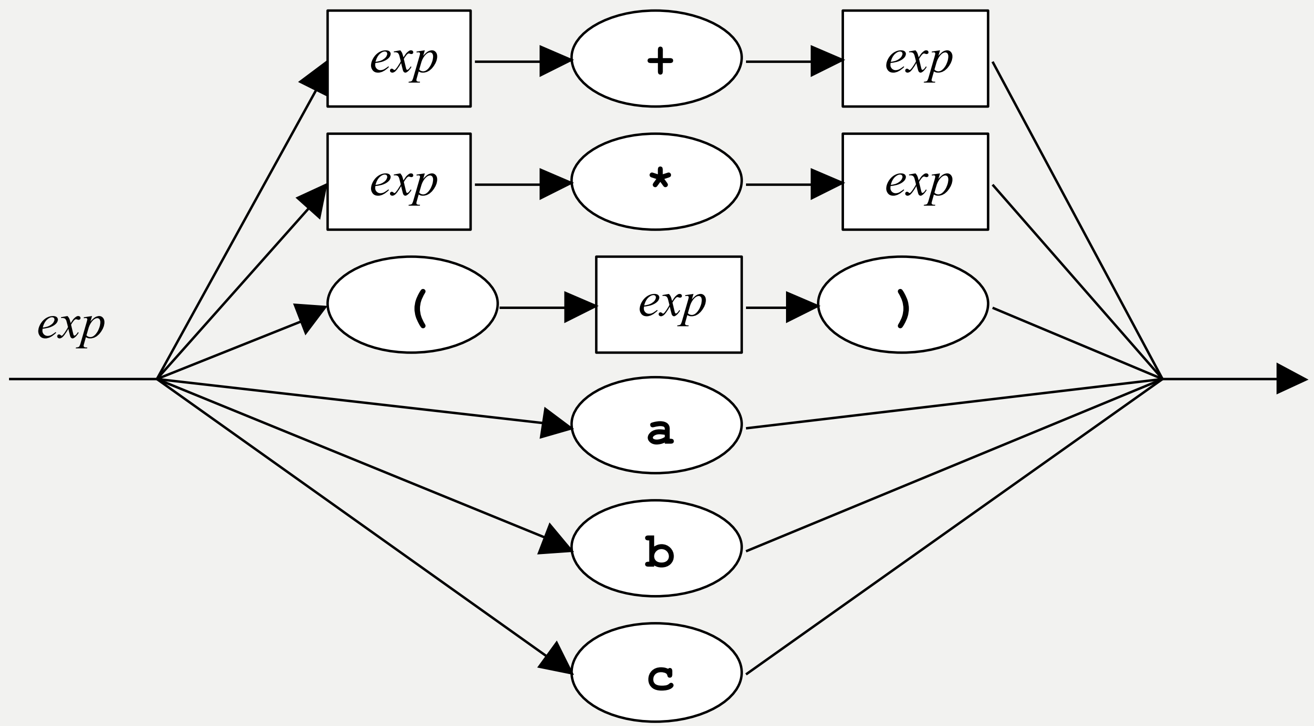
<*if-stmt*> **::=** **`if`** <*expr*> **`then`** <*stmt*> [**`else`** <*stmt*>]



Branching

- Use branching for multiple productions

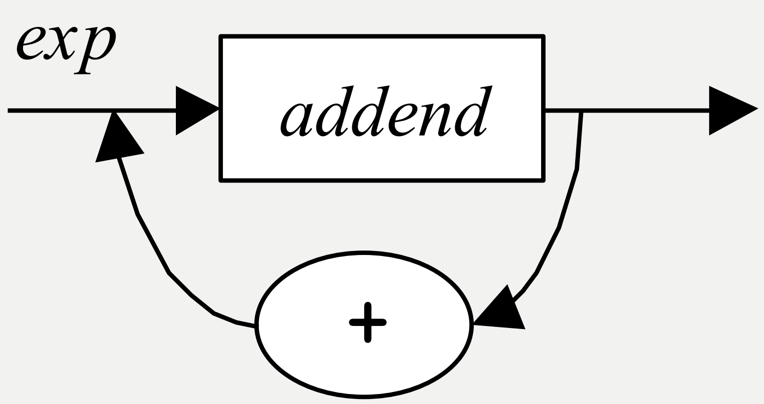
<*exp*> ::= <*exp*> **+** <*exp*> | <*exp*> ***** <*exp*> | **(** <*exp*> **)**
| **a** | **b** | **c**



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:

$$S \rightarrow aSb \mid X$$
$$X \rightarrow cX \mid \varepsilon$$

- These are called *context-free grammars*
- 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*) ;

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

[from *The Java™ Language Specification*,
Third Edition, 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*