Defining Program Syntax

Chapter 3

Defining a Programming Language

- Defining a programming language requires specifying its **syntax** and its **semantics**.
- Syntax:
  - The **form or structure** of the expressions, statements, and program units.
  - Example: if (<exp>) then <statement>
- Semantics:
  - The **meaning** of the expressions, statements, and program units.
  - Example: if the value of <exp> is non-zero, then <statement> is executed otherwise omitted.

Syntax and Semantics

- There is universal agreement on how to express syntax.
  - BNF is the notation.
- Backus-Naur Form (BNF)
  - Defined by John Backus and Peter Naur as a way to characterize Algol syntax (it worked.)

Who needs language definitions?

- Other language designers
  - To evaluate whether or not the language requires changes before its initial implementation and use.
- Programmers
  - To understand how to use the language to solve problems.
- Implementers
  - To understand how to write a translator for the language into machine code (compiler)

Language Sentences

- A **sentence** is a string of characters over some **alphabet**.
- A **language** is a set of sentences.
- **Syntax rules** specify whether or not any particular sentence is defined within the language.
- Syntax rules do not guarantee that the sentence “makes sense”!

Recognizers vs. Generators

- Syntax rules can be used for two purposes:

- **Recognizers**:
  - Accept a sentence, and return true if the sentence is in the language.
  - Similar to syntactic analysis phase of compilers.

- **Generators**
  - Push a button, and out pops a legal sentence in the language.

Definition of a BNF Grammar

- BNF Grammars have four parts:
  - **Terminals**:
    - the primitive tokens of the language ("a", "+", "begin",...)
  - **Non-terminals**:
    - Enclosed in "<" and ">", such as <prog>
  - **Production rules**:
    - A single non-terminal, followed by
    - "->", followed by
    - a sequence of terminals and non-terminals.
  - The **Start symbol**:
    - A distinguished nonterminal representing the “root” of the language.

Definition of a BNF Grammar

- A set of terminal symbols
  - Example: { "a" "b" "c" "(" ")" "," }
- A set of non-terminal symbols
  - Example: { <prog> <stmt> }
- A set of productions
  - Syntax: A single non-terminal, followed by a "->", followed by a sequence of terminals and non-terminals.
  - Example: <prog> -> "begin" <stmt_list> "end"
- A distinguished non-terminal, the Start Symbol
  - Example: <prog>

Example BNF Grammar

- Productions
  - <prog> -> "begin" <stmt_list> "end"
  - <stmt_list> -> <stmt>
  - <stmt_list> -> <stmt> ";" <stmt_list>
  - <stmt> -> <var> ":=" <exp>
  - <var> -> "a"
  - <var> -> "b"
  - <var> -> "c"
  - <exp> -> <var> "+" <var>
  - <exp> -> <var> "-" <var>
  - <exp> -> <var>

Extended BNF

- EBNF extends BNF syntax to make grammars more readable.
- EBNF does not make BNF more expressive, it's a short-hand.
  - Sequence :
    - <if> -> "if "<test> "then" <stmt>
  - Optional – [ ] :
    - <if> -> "if "<test> "then" <stmt> ["else" <stmt>]
  - Alternative – | :
    - <number> -> <integer> | <real>
  - Group – ( ) :
    - <exp> -> <var> | (<var> "+" <var>)
  - Repetition – { }:
    - <ident_list> -> <ident> { "," <ident>}

XML

- Allows us to define our own Programming Language
- Usage
  - SMIL: multimedia presentations
  - MathML: mathematical formulas
  - XHTML: web pages
- Consists of
  - hierarchy of tagged elements
  - start tag, e.g <data> and end tag, e.g. </data>
  - text
  - attributes

XML Example

- <university>
  <department>
  <name> ISC </name>
  <building> POST </building>
  </department>
  <student>
  <first_name> John </first_name>
  <last_name> Doe </last_name >
  </student>
  <student>
  <first_name> Abe </first_name>
  <middle_initial> B </middle_initial>
  <last_name> Cole </last_name >
  </student>
  </university>

EBNF for XML Example

- Productions
  - <institution> -> "<university>" { <unit> } {<person>} "</university>"
  - <unit> -> "<department>" <name> <place> "</department>"
  - <name> -> "<name>" <text> "</name>"
  - <place> -> "<building>" <text> "</building>"
  - <person> -> "<student>" <first> [<middle>] <last> "</student>"
  - <first> -> "<first_name>" <text> "</first_name>"
  - <middle> -> "<middle_initial>" <letter> "</middle _initial>"
  - <last> -> "<last_name>" <text> "</last_name>"
- Start symbol
  - <institution>
- No-Terminal symbols
  - <institution>, <unit>, <name>, <place>, <person>, <first>, <middle>, <last>
- Terminal symbols
  - "<university>", "</university>", "<department>", "</department>", "<name>", "</name>", "<building>", "</building>", "<student>", "</student>", "<first_name>", "</first_name>", "<middle _initial>", "</middle _initial>", "<last_name>", "</last_name>", <text>, <letter>

Definition of a XML in EBNF

- Terminal symbols
  - {"<" , “</" , ">" , <text> }
- Non-terminal symbols
  - {<element> , <elements> , <start_tag> , <end_tag> }
- Productions
  - <element> -> <start_tag> (<elements> | <text>) <end_tag>
  - <elements> -> <element> { <element> }
  - <start_tag> -> "<" <text> ">"
  - <end_tag> -> "</" <text> ">"
- Start Symbol
  - <element>

XML Grammars

- Similar to EBNF: Sequence of productions
  - Sequence
  - Group – ( ): (<elements>)
  - Alternative – | : <element> | <element>
  - Optional – [ ]: <element> ?
  - Repetition – { }: <element> *
  - Repetition at least one – { }: <element> +
- Productions
  - enclosed in "<!ELEMENT" and ">"
  - left-hand side either: (elements) or: (#PCDATA) or: EMPTY
  - e.g. EBNF: <department> -> { <employee> }
    is in XML: <!ELEMENT department (employee*)>
- Terminal symbols
  - <text> in EBNF becomes in XML: #PCDATA
- Start Symbol
  - Is found in XML document

Example XML Grammar

- <!ELEMENT department (employee*)>
- <!ELEMENT employee (name, (email+ | url))>
- <!ELEMENT name (#PCDATA)>
- <!ELEMENT email (#PCDATA)>
- <!ELEMENT url (#PCDATA)>

Generation

- A grammar can be used to generate a sentence:
  - Choose a production with the start symbol as its LHS (left-hand side).
  - Write down the RHS as the sentence-to-be.
  - For each non-terminal in the sentence-to-be:
    - Choose a production with this non-terminal as its LHS
    - Substitute the production's RHS for the non-terminal
  - Keep going until only terminal symbols remain. The result is a legal sentence in the grammar.

Example sentence generation

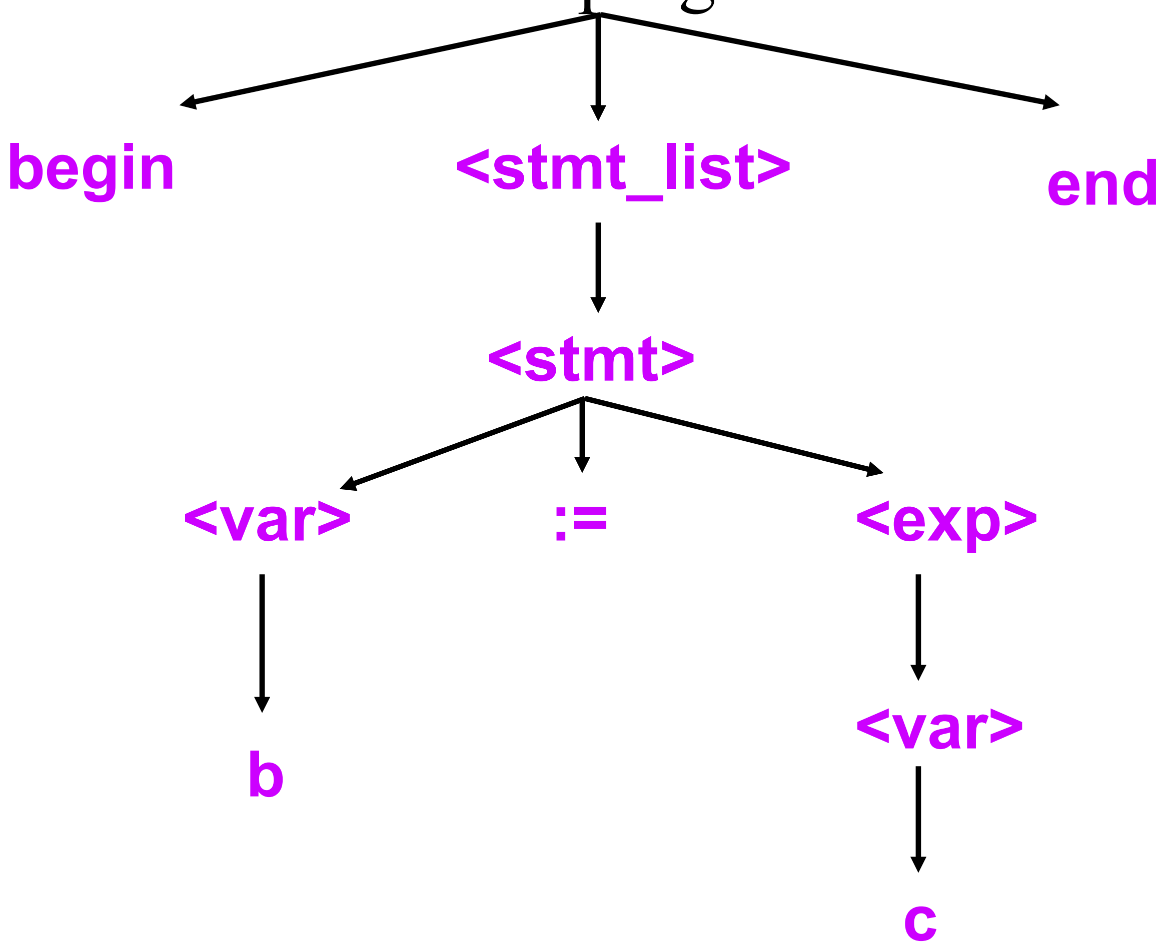
- begin <stmt_list> end
- begin <stmt> end
- begin <var> := <exp> end
- begin b := <exp> end
- begin b := <var> end
- begin b := c end

- Sentence generation is also known as “derivation”

- Derivation can be represented graphically as a “parse tree”.

Example Parse Tree

- <prog>



Recognition

- Grammar can also be used to test if a sentence is in the language. This is “recognition”.
- One form of recognizer is a “parser”, which constructs a parse tree for a given input string.
- Programs exist that automatically construct a parser given a grammar (example: yacc)
  - Not all grammars are suitable for yacc.
- Depending on the grammar, parsers can be either “top-down” or “bottom-up”.

Basic Idea of Attribute Grammars

- Take a BNF parse tree and add values to nodes.
- Pass values up and down tree to communicate syntax information from one place to another.
- Attach "semantic rules" to each production rule that describe constraints to be satisfied.

Attribute Grammar Example

- This is not a “real” example.
- BNF:
  - <proc> -> procedure <proc_name>
    <proc_body>
    end <end_name>;
- Semantic rule:
  - <proc_name>.string = <end_name>. string
- Attributes:
  - A “string” attribute value is computed and attached to <proc_name> and <end_name> during parsing.

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

Syntax Basics

- 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`**

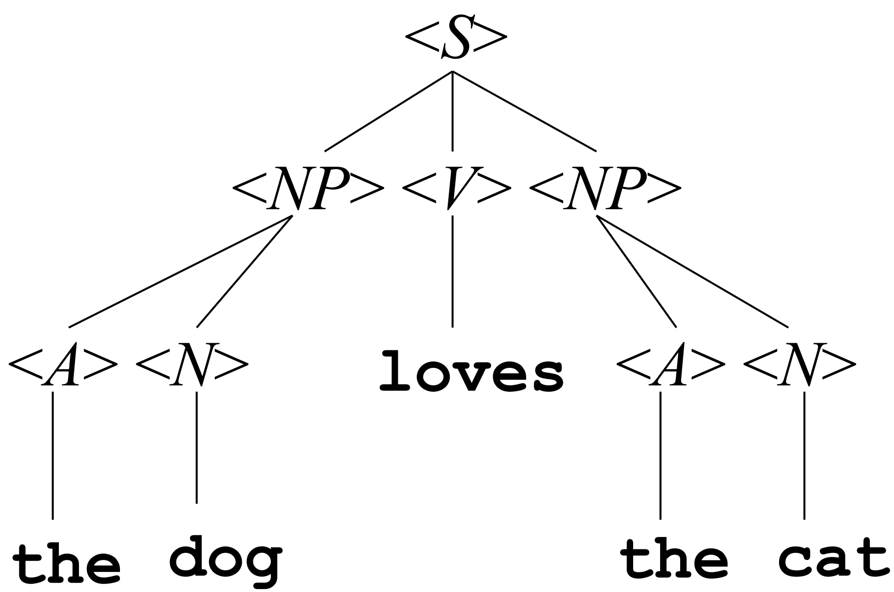
How The Grammar Works

- 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



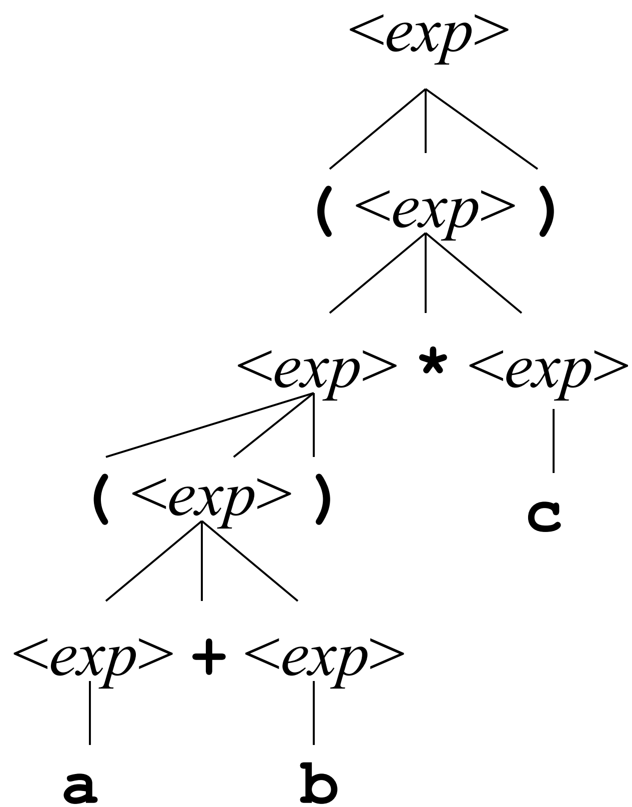
A Programming Language Grammar

$$\langle exp \rangle ::= \langle exp \rangle \;\texttt{+}\; \langle exp \rangle \mid \langle exp \rangle \;\texttt{*}\; \langle exp \rangle \mid \texttt{(}\; \langle exp \rangle \;\texttt{)} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c}$$

- 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)



Syntax Basics

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

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

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, Continued

- The tokens are the smallest units of syntax
  - Strings of one or more characters of program text
  - They are 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 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>*

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

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
- Constructing grammars is a little like programming...

Syntax Basics

- 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;
```

Example, Continued

- 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>*

Syntax Basics

- 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 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 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
- Early dialects of Fortran, Cobol, and Basic

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

Syntax Basics

- 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>* **`;`**}

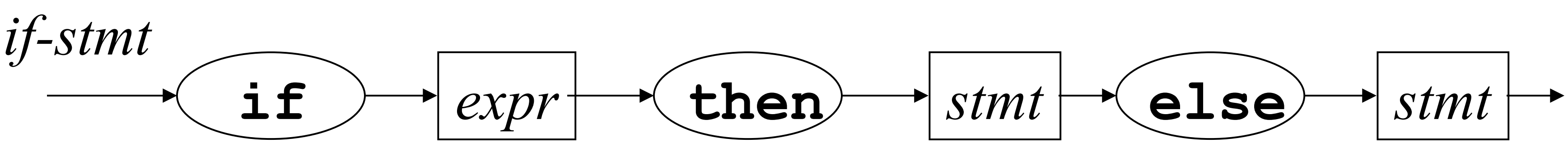
<thing-list> **::=** { (*<stmt>* | *<declaration>*) **`;`**}

- 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):

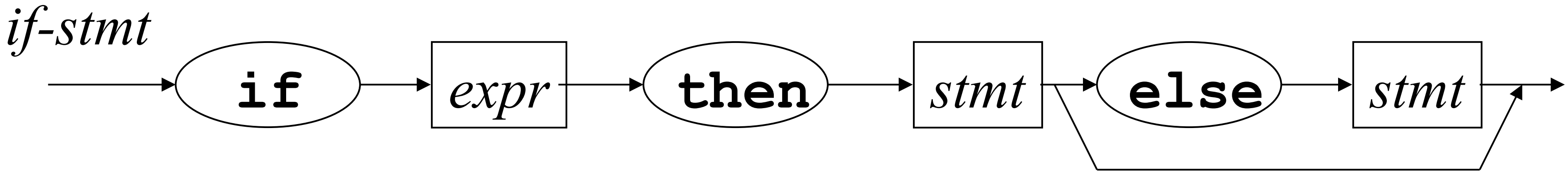
<*if-stmt*> **`::=`** **`if`** <*expr*> **`then`** <*stmt*> **`else`** <*stmt*>



Bypasses

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

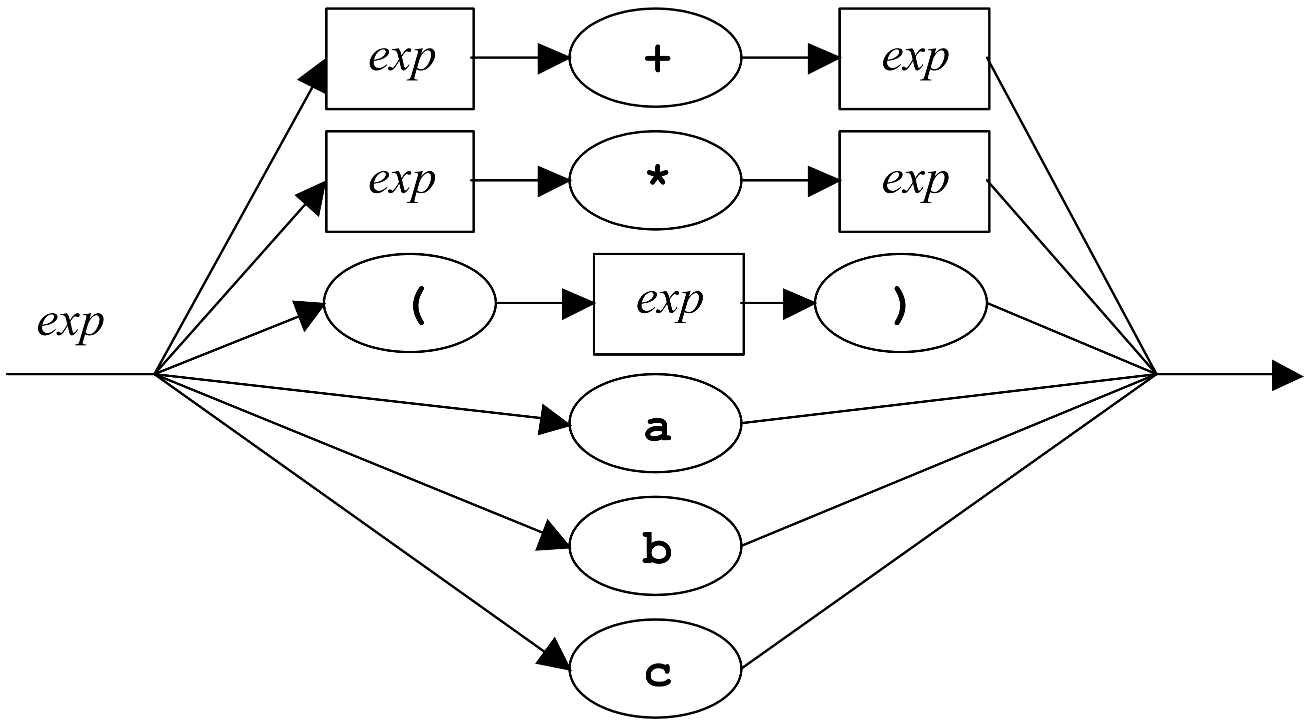
<if-stmt> **`::=`** **`if`** *<expr>* **`then`** *<stmt>* [**`else`** *<stmt>*]



Branching

- Use branching for multiple productions

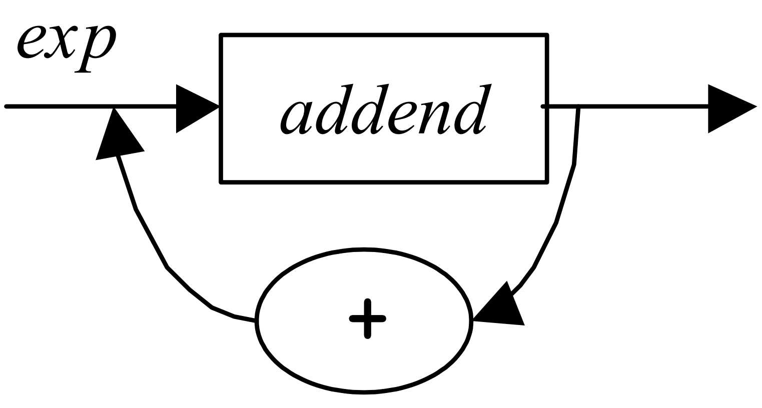
<exp> ::= *<exp>* **+** *<exp>* | *<exp>* ***** *<exp>* | **(** *<exp>* **)**
| **a** | **b** | **c**



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, grammars are expressed in yet another notation:

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

- These are called context-free grammars

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…

- 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

End of Lecture 4

- Next time – Semantics