Specifying Synatax with BNFs (Backus Naur Forms)

Programming Languages

CS 214

Dept of Computer Science
Calvin College

Sentence Structure

Consider the following "sentences"

- *there is hair in my soup*
- *there is soup in my hair*
- *is there soup in my hair*

What does each "sentence" mean?

What about this "sentence"?

- *hair soup there my is in*

→ The ____________________ determines the meaning...

What determines which word-orders are *meaningful*?

Grammar & Syntax

Every language has a set of rules - its ________________________ - that specifies the word-sequences that form valid sentences.

The "sentence":

- *hair soup there my is in*

is not valid, because it violates English's grammar rules (i.e., it contains ______________________).

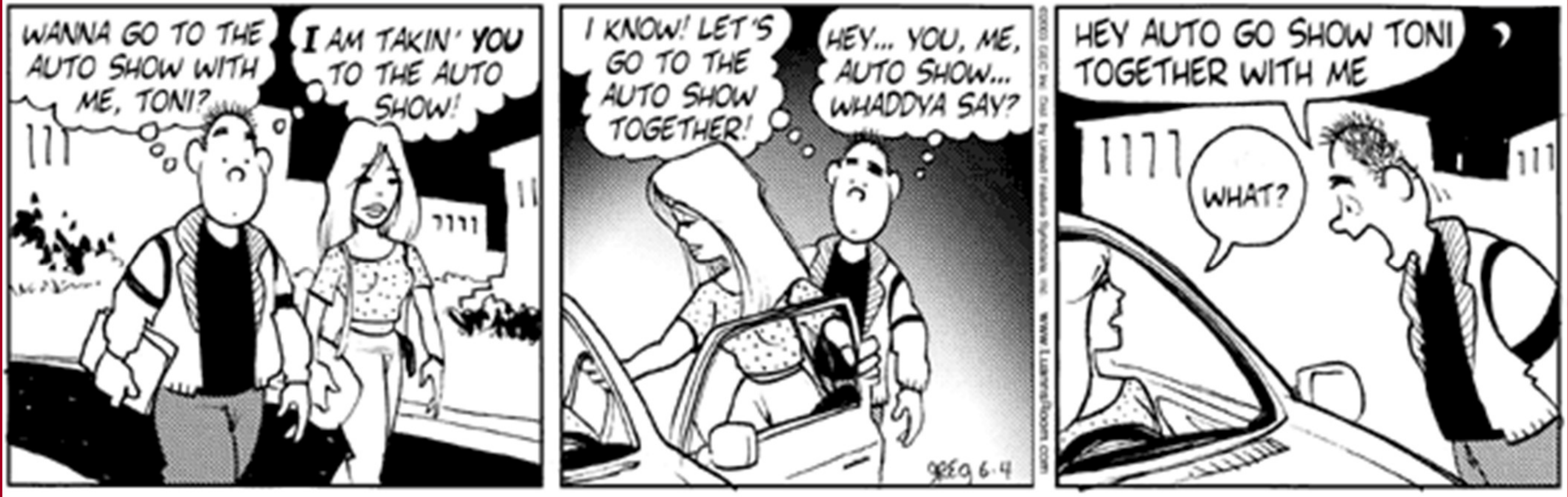
Grammar and syntax help us decode a sentence's meaning:

syntax were read to easier sentence would unimportant this if be

____

  Dept of Computer Science Calvin College

Syntax Matters In Real Life



© 2003 GEC, Inc./Dist. by UFS, Inc.

Semantics

When a sentence has correct syntax, our brains can determine what the words in the sentence mean: their ______________.

Consider: *time flies like an arrow, fruit flies like a banana*

→ *flies* and *like* have two different meanings

We decode a word's meaning using its __________________.

Since a word's semantics (meaning) depends on its context, English is a ____.

If a sentence contains syntax errors, we can't understand it, because syntax specifies what words can be adjacent, and a word's semantics depends on the words surrounding it.

Program Sentences

A program is a "sentence" in a programming language.

A program's "meaning" depends on the order of its symbols.

To be valid, the order must obey the syntax rules of the language; for example:

```
x = y + 1;
```

is a valid statement in some languages (e.g., ____________) but not in others (e.g., _____________).

The meanings of the "words" in a program are determined by the _________________ of the language in which it's written.

Syntax Errors

Syntax rules specify those symbol-orderings that are valid, allowing a compiler to determine symbol-meanings.

```
x = y + 1;          →   MOV y, R0
                        ADD #1, R0
                        STO R0, x
```

When a program contains syntax errors, a compiler is unable to translate it (i.e., determine its meaning):

```
y + 1 = x;          →   ?
```

A compiler cannot determine the meaning of any "phrase" that violates the language's syntax rules.

BNF

The ______________________________ is a tool for specifying the syntax of a high level language (HLL).

Example: A BNF giving the structure of C++ identifiers is:

```
<identifier>        ::=  <first_letter> <valid_sequence>
<first_letter>      ::=  <letter> | _
<valid_sequence>    ::=  <valid_symbol> <valid_sequence> | ε
<valid_symbol>      ::=  <letter> | <digit> | _
<letter>            ::=  A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q |
                         R | S | T | U | V | W | X | Y | Z | a | b | c | d | e | f | g | h | i |
                         j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
<digit>             ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A correct BNF specifies all valid “sentences”, and prohibits all invalid “sentences”.

Examples:

Is *R2D2* a valid sentence in our <identifier> language?

<identifier>
<first_letter> <valid_sequence>
<letter> <valid_sequence>
R <valid_sequence>
R <valid_symbol> <valid_sequence>
R <digit> <valid_sequence>
R 2 <valid_sequence>
R 2 <valid_symbol> <valid_sequence>
R 2 <letter> <valid_sequence>
R 2 D <valid_sequence>
R 2 D <valid_symbol> <valid_sequence>
R 2 D <digit> <valid_sequence>
R 2 D 2 <valid_sequence>
R 2 D 2 ε

This sequence of steps is called a *derivation*.

"Sentences" not conforming to the BNF are *invalid*.

Formal Definitions

Let Σ be a set of symbols.

- A *string over* Σ is a finite sequence of zero or more symbols from the set Σ.
- Symbols whose meaning is predefined are called *terminals*.
  - Symbols like *A*, _, *6*, etc. are terminals in our <identifier> BNF.
- Symbols whose meanings must be defined are called *non-terminals*, and are enclosed in angle-brackets (< and >).
  - Symbols like <*identifier*>, <*letter*>, etc. are non-terminals.
  - Like variables, non-terminals usually describe what they represent.
- One symbol is designated as the *starting non-terminal*.
  - The symbol <*identifier*> is the starting non-terminal in our BNF.

Formal Definitions (ii)

- Each non-terminal must be defined by a *production* (rule):

<LHS> ::= <RHS>

where: <LHS> is the nonterminal being defined,
<RHS> is a string of terminals and/or nonterminals defining <LHS>.

- Different productions defining the same non-terminal:

$\langle NT_i \rangle$ ::= Def_1
$\langle NT_i \rangle$ ::= Def_2
...
$\langle NT_i \rangle$ ::= Def_n

can be written in shorthand using the OR (|) operator:

$\langle NT_i \rangle$ ::= $Def_1 \mid Def_2 \mid \ldots \mid Def_n$

Formal Definitions (iii)

- A BNF is a quadruple: (Σ, N, P, S), where:

  Σ is the set of symbols in the BNF;

  N is the subset of Σ that are nonterminals in the language;

  P is the set of productions defining the symbols in N; and

  S is the element of N that is the starting nonterminal.

- A *derivation* is a sequence of strings, beginning with the starting nonterminal S, in which each successive string replaces a nonterminal with one of its productions, and in which the final string consists solely of terminals.

  → A derivation is sometimes called a *parse*.

Formal Definitions (iv)

- A BNF ______________________________ is a tree, such that
  - the root of the tree is the starting nonterminal (S) in the BNF;
  - the children of the root are the symbols (L to R) in a production whose <LHS> is S, the starting nonterminal;
    - each terminal child is a leaf; and
    - each nonterminal child is the root of a derivation tree for that nonterminal.
- The act of building a derivation tree for a sentence (to check its correctness) is called _________ that sentence.

  → The set of valid sentences in a language is the set of all sentences for which a parse tree exists!
- A _______________________ is a derivation built by *always expanding the left-most non-terminal* in a production.

Examples

Do parse trees (using our <identifier> BNF) exist for these?

i *a1b2* *_5_* *_$_*

Recursive Productions

Our identifier-BNF permits identifiers to have arbitrary lengths through the use of *recursive productions*:

<valid_sequence> ::= <valid_symbol> <valid_sequence> | ε

This production is recursive because the non-terminal on the <LHS> appears in the production (on the <RHS>).

The recursive production provides for *unrestricted repetition* of the non-terminal being defined, which is useful what is being defined can be appear ________________________.

The ε-production provides both:

– a *base-case* for trivial instances of the non-terminal, and

–an *anchor* to terminate the recursion.

Writing BNFs: A 3-Step Process

A. Start with ____.

B. Build the productions to define the non-terminal:

1. Start with the question: ________________________?
2. If a construct is _____________:
    - a. create a new nonterminal for that construct;
    - b. add a production for the non-optional case;
    - c. add an ε-production for the optional case.
3. If a construct can be _____________________________________:
    - a. create a new nonterminal for that construct;
    - b. add an ε-production for the zero-reps case;
    - c. add a recursive production for the other cases.

C. For each nonterminal in the <RHS> of every production:

repeat step B until all nonterminals have been defined.

Example: C++ *if* Statement

A. Create a non-terminal for what we're defining:

<if_stmt>

B. Build a production to define it: what comes first?

1. keyword ____
2. open parentheses
3. ____________
4. close parentheses
5. ____________
6. ________________________

<if_stmt> ::= ____________________________

C. Repeat B for each undefined non-terminal:

<else_part> ::= ______________________

(We'll see how to define <expr> and <stmt> a bit later...)

A Small Problem

The C++ if statement presents a small problem: Consider…

```
if ( a < b) if (a < c) S1 else S2
```

Take a moment to build a parse tree for this “sentence”...

Ambiguities

When a "sentence" has multiple parse trees, it is ____________ (i.e., it has multiple interpretations and/or meanings).

The two parses reflect different ways to associate the **`else`**:

```
if (a < b)
    if (a < c)
        S1
    else
        S2
```

```
if (a < b)
    if (a < c)
        S1
else
    S2
```

The grammar cannot resolve which is meant, so C++ uses a ___________________ to resolve the ambiguity:

An **`else`** *associates with the closest prior unterminated* **`if`**.

Example: C++ *do* Statement

A. Create a non-terminal for what we're defining:

<do_stmt>

B. Build a production to define it: what comes first?

1. keyword ______
2. ____________________
3. keyword ______
4. open parentheses
5. an expression
6. close parentheses
6. a semicolon

<do_stmt> ::= ___________________________

C. Repeat B for each undefined non-terminal…

<stmt> isn't too bad, so let's tackle it next.

Example: C++ <stmt>

A. Create a non-terminal for what we're defining:

<stmt>

B. Build a production to define it: what comes first?

- It depends on the kind of statement being described...
- There are seven different kinds of C++ statements, so let's introduce a new non-terminal for each one:

```
<stmt>          ::=  <compound_stmt> | <selection_stmt> |
                     <iteration_stmt> | <expression_stmt> |
                     <jump_stmt> | <labeled_stmt> | <declaration_stmt>
```

C. Repeat B for each undefined non-terminal…

```
<compound_stmt>  ::=  ______________________________
<stmt_list>      ::=  ______________________________
```

Example: C++ <stmt> (ii)

<selection_stmt> ::= <if_stmt> | <switch_stmt>

<iteration_stmt> ::= <while_stmt> | <do_stmt> | <for_stmt>

<expression_stmt> ::= <opt_expr> ;

<opt_expr> ::= <expr> | ε

<jump_stmt> ::= break ; | continue ; | return <opt_expr> ; | goto <identifier> ;

<labeled_stmt> ::= <identifier> : <stmt> | case <literal> : <stmt> | default : <stmt>

<declaration_stmt> ::= <object_dec> | <function_dec> | <class_dec>

<object_dec> ::= <modifier> <type> <identifier> <initializer> <more_ids> ;

<modifier> ::= register | const | static | auto | extern | mutable | ε

<type> ::= char | wchar_t | bool | short | int | long | signed | unsigned |
float | double | <identifier> | <identifier> :: <type>

<initializer> ::= = <expr> | ε

<more_ids> ::= , <identifier> <initializer> <more_ids> | ε

…

Example: C++ Assignment Exprs

A. Create a non-terminal for what we're defining:

<assign_expr>

B. Build a production to define it: what comes first?

- Many things might come first (variable, pointer, array, ...) so let's introduce a non-terminal to hide the details.
- Next comes an assignment operator; then an expression:

<assign_expr> ::= ____

C. Repeat B for each undefined non-terminal…

<lvalue> ::= ____

<unary_op> ::= _________

<id_suffix> ::= [<expr>] <id_suffix> | . <lvalue> <id_suffix> |
-> <lvalue> <id_suffix> | ε

<assign_op> ::= _____ | += | -= | *= | /= | %= | <<= | >>= | &= | |= | ^=

Expressions

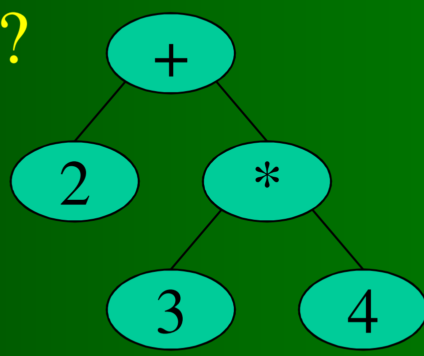
C++ expressions are complicated:

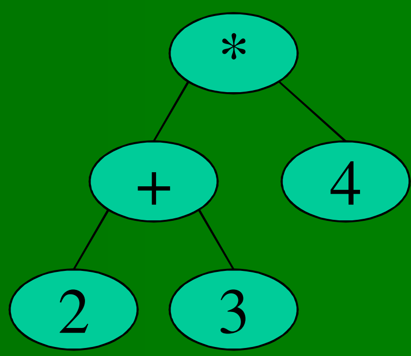
- Each of its 52 operators must be included
- Each of its 17 precedence levels must be included
- Associativity rules must be enforced
- The grammar/BNF must be unambiguous

Example: For the expression: 2 + 3 * 4
our grammar must ensure that: 2 + (3 * 4) = 14
is evaluated, not: (2 + 3) * 4 = 20

Which parse tree is correct?

Our grammar must *only* generate the correct one.





Grammar and Precedence

To ensure that higher precedence operators appear lower in the parse tree, we must build ________________ into our BNF:

<expr>	::=	… | <add_expr> | …
<add_expr>	::=	<mul_expr> | <add_expr> + <mul_expr> | <add_expr> - <mul_expr>
<mul_expr>	::=	<value> | <mul_expr> * <expr> | <mul_expr> / <expr> | <mul_expr> % <expr>

Rules like these ensure that "multiply-level" operators will be applied before "add-level" operators...

<expr>
<add_expr>
<add_expr> + <mul_expr>
<mul_expr> <mul_expr> * <expr>
<value> <value> <add_expr>
2 3 <mul_expr>
<value>
4

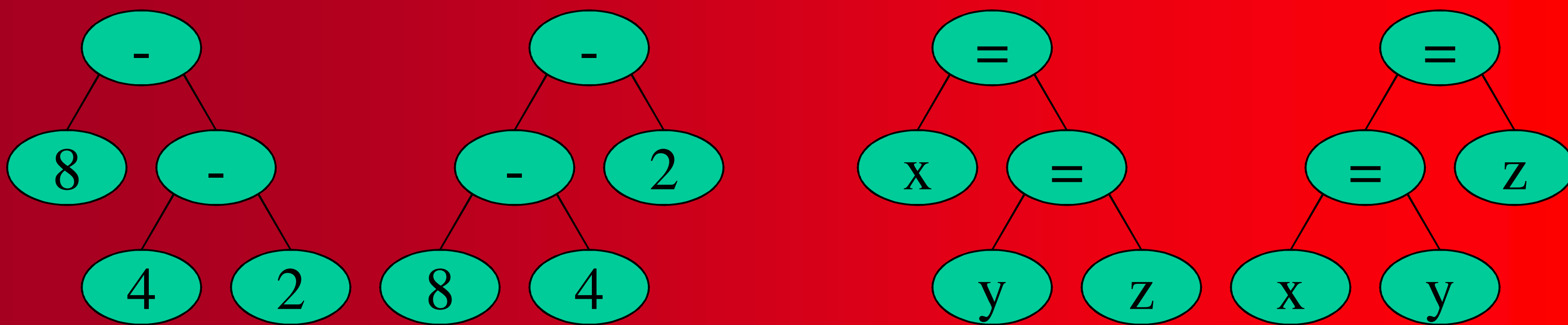
Grammar and Associativity

______________ gives ordering of equal-precedence operators.

Examples: 8 - 4 - 2 vs. x = y = z

- is left-associative; which is correct?

= is right-associative; which is correct?



Associativity can be built into a grammar by using _______ ____________ *productions* for left-associative operators; and _______________ *productions* for right-associative operators.

Associativity Examples

Example: Since +, - are left-associative, we write:

<add_expr> ::= <mul_expr> | ______________________ |
____

but since = is right-associative, we write what amounts to:

<assign_expr> ::= ______________________

These generate the correct parse trees for each expression:

<expr>
<add_expr>
<add_expr> - <mul_expr>
<add_expr> - <mul_expr> <value>
<mul_expr> <value> 2
<value> 4
8

<expr>
<assign_expr>
<lvalue> <assign_op> <assign_expr>
<identifier> = <lvalue> <assign_op> <assign_expr>
x <identifier> = <add_expr>
y <mul_expr>
<identifier>
z

C++ Expressions

```
<expr>          ::=  <assign_expr> | <expr> , <assign_expr>
<assign_expr>   ::=  <lvalue> <assign_op> <assign_expr> | <cond_expr>
<cond_expr>     ::=  <lor_expr> | <lor_expr> ? <expr> : <cond_expr>
<lor_expr>      ::=  <land_expr> | <lor_expr> || <land_expr>
<land_expr>     ::=  <bor_expr> | <land_expr> && <bor_expr>
<bor_expr>      ::=  <xor_expr> | <bor_expr> | <xor_expr>
<xor_expr>      ::=  <band_expr> | <xor_expr> ^ <band_expr>
<band_expr>     ::=  <equ_expr> | <band_expr> & <equ_expr>
<equ_expr>      ::=  <rel_expr> | <equ_expr> == <rel_expr> | <equ_expr> != <rel_expr>
<rel_expr>      ::=  <shft_expr> | <rel_expr> < <shft_expr> | <rel_expr> > <shft_expr>
                       | <rel_expr> <= <shft_expr> |  <rel_expr> >= <shft_expr>
<shft_expr>     ::=  <add_expr> | <shft_expr> << <add_expr> |
                       <shft_expr> >> <add_expr>
<add_expr>      ::=  <mul_expr> | <add_expr> + <mul_expr> | <add_expr> - <mul_expr>
<mul_expr>      ::=  <ptr_expr> | <mul_expr> * <ptr_expr> | <mul_expr> / <ptr_expr> |
                       <mul_expr> % <ptr_expr>
<ptr_expr>      ::=  …
```

Exercise

Build a parse tree for: `a = x + y / z ;`

EBNF

The BNF is the most general tool for expressing syntax.

Another tool frequently used is the ________________________.

The differences are:

- EBNF terminals are distinguished from non-terminals by
  - Capitalizing the first-letter of non-terminals, AND
  - Underlining, single-quoting, or bolding terminals (instead of surrounding non-terminals by angle-brackets).
- Parentheses may be used to denote grouping.
- ________ surround symbols that are ________________________.
  - → No recursion!
- ________ surround symbols that are ____________.
  - → No ε-productions!

Examples

1. To specify a C++ block using EBNF:
   – First comes a brace, then zero or more statements, then a brace:

   ________ ::= ________________

2. To specify a C++ int literal using EBNF:
   – An optional sign, an optional base specifier, at least one digit

   ________ ::= ________________________

   Sign ::= + | -

   Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3. To specify a C++ do statement using EBNF:
   – Keyword do, a statement, keyword while, an open-parentheses, an expression, a close-parentheses, a semicolon:

   ________ ::= ________________________

Problems

1. Specify a C++ identifier using EBNF:

2. Specify a C++ while statement using EBNF:

3. Specify a C++ if statement using EBNF:

Why use BNFs instead of EBNFs?

The recursive productions in BNFs make it easier for a compiler to parse "sentences" in the language…

Basic Parsing Algorithm:

0. Push *S* (the starting symbol) onto a stack.
1. Get the first terminal symbol *t* from the input file.
2. Repeat the following steps:
    - a. Pop the stack into *topSymbol*;
    - b. If *topSymbol* is a nonterminal:
        - 1) Choose a production *p* of *topSymbol* based on *t*
        - 2) If *p* != ε:
            - Push *p* right-to-left onto the stack.
    - c. Else if *topSymbol* is a terminal && *topSymbol* == *t*:
        - Get the next terminal symbol *t* from the input file.
    - d. Else
        - Generate a 'parse error' message.

while the stack is not empty.

Example

Suppose our rules are:

<var_dec> ::= <type> id <id_list> ; <var_dec> | ε

<type> ::= int | char | float | double | …

<id_list> ::= , <id> <id-list> | ε

Let's parse the declaration: **`int x, y;`**

assuming that <var_dec> is our starting symbol.

stack:

<var_dec>	<type>	int	id	<id_list>	,	id	<id_list>	;	<var_dec>
	id	id	<id_list>	;	id	<id_list>	;	<var_dec>	
	<id_list>	<id_list>	;	<var_dec>	<id_list>	;	<var_dec>		
	;	;	<var_dec>		;	<var_dec>			
	<var_dec>	<var_dec>			<var_dec>				

t: ______ ______ ___ ______ ___

Summary

There are different ways to specify the syntax of a language.
Two of them are:

- ______ • _______

The EBNF is simpler and easier to use, so it is frequently used in ________________________________.

The BNF's recursive- and ε-productions simplify the task of parsing, so it is more useful to ______________________.

