Syntax of Programming Languages *(cont'd)*

Syntactic Ambiguity

In English

Syntactically ambiguous sentences of English:

- "I saw the dog with the binoculars."
- "The friends you praise sometimes deserve it."
- "He seemed nice to her."

Other kinds of ambiguity in English:

Aside: We can often "disambiguate" ambiguous sentences. **Question:** How?

But we can be wrong.
Example: "I put the box on the table   ."

In a programming language

Example:

```
<stmt>    --> <assnt-stmt> | <loop-stmt> | <if-stmt>
<if-stmt> --> if <boolean-expr> then <stmt>
           |  if <boolean-expr> then <stmt> else <stmt>
```

Example sentence:

```
if (x odd) then
if (x == 1) then
print "bleep";
else
print "boop";
```

Exercise: Draw the two parse trees.

Definition: A **grammar is ambiguous** iff it generates a sentence for which there are two or more distinct parse trees

To prove that a grammar is ambiguous, give a string and two parse trees for it.

A **sentence is ambiguous** with respect to a grammar iff that grammar generates two or more distinct parse trees for the sentence.

Note that having two distinct *derivations* does not make a sentence ambiguous. A derivation corresponds to a traversal through a parse tree, and one can traverse a single tree in many orders.

Example

Grammar: if statement two slides ago.

Sentence:

```
if (x odd) then
print "bleep";
```

One parse tree:

Two derivations:

Want: When specifying a programming language, we want the grammar to be completely unambiguous.

Research question: Is there a procedure one can follow to determine whether or not a given grammar is ambiguous?

Notation and Terminology

We say that $L(G)$ is the language generated by grammar G.

So G is ambiguous if $L(G)$ contains a sentence which has more than one parse tree, or more than one *leftmost* (or *canonical*) derivation.

Dealing with ambiguity

We have two strategies:

1. Change the *language* to include **delimiters**

2. Change the *grammar* to impose **associativity** and **precedence**

Changing the language to include delimiters

Algol 68 if-statement grammar:

```
<stmt>    --> <assnt-stmt> | <loop-stmt> | <if-stmt>
<if-stmt> --> if <boolean-expr> then <stmt> fi
           |  if <boolean-expr> then <stmt>
                                else <stmt>
              fi
```

Example: A CFG for Arithmetic Expressions

Grammar 1:

```
<expn> --> <expn> + <expn>  |
           <expn> - <expn>  |
           <expn> * <expn>  |
           <expn> / <expn>  |
           <expn> ^ <expn>  |
           <identifier>  |
           <literal>
```

Example: parse `8 - 3 * 2`

Changing the language to include delimiters

Grammar 2:

```
<expn> --> ( <expn> ) - ( <expn> )  |
           ( <expn> ) * ( <expn> )  |
           <identifier>  |
           <literal>
```

`(8)-((3)*(2))` $\in L(G)$

`((8)-(3))*(2)` $\in L(G)$

`8 - 3 * 2` $\notin L(G)$

Grammar 3:

```
<expn> --> <expn> - <expn>  |
           <expn> * <expn>  |
           <identifier>  |
           <literal>  |
           ( <expr> )
```

Accepts all expressions, but still ambiguous!

Changing the grammar to impose precedence

Grammar 4:

```
<expn> -->
```


Grouping in parse tree now reflects precedence

Example: parse `8 - 3 * 2`

Precedence

- Low Precedence:
  Addition + and Subtraction -
- Medium Precedence:
  Multiplication * and Division /
- Higher Precedence:
  Exponentiation ^
- Highest Precedence:
  Parenthesized expressions (`<expr>`)

⇒ Ordered lowest to highest in grammar.

Approach: Introduce a non-terminal for every precedence level.

Associativity

- Deals with operators of same precedence
- Implicit grouping or parenthesizing
- Left associative: *, /, +, -
- Right associative: ^

Approach: For left-associative operators, put the recursive term *before* the nonrecursive term in a production rule. For right-associative operators, put it *after*.

Associativity (cont.)

Examples:

- We want multiplication to be left-associative, so we wrote:

```
<term>  ->  <term> * <factor>
```

- We want exponentiation to be right-associative, so might write:

```
<expo>  ->  <number> ** <expo>  |  <number>
```


Dealing with Ambiguity

1. Can't *always* remove an ambiguity from a grammar by restructuring productions.
2. When specifying a programming language, we want the grammar to be completely unambiguous.
3. An inherently ambiguous language does not possess an unambiguous grammar.
4. There is no algorithm that can examine an arbitrary context-free grammar and tell if it is ambiguous, i.e., detecting ambiguity in context-free grammars is an *undecidable* problem.

An Inherently Ambiguous Language

Suppose we want to generate the following language:

$$\mathcal{L} = \{a^i b^j c^k \mid i, j, k \geq 1, i = j \text{ or } j = k\}$$

Grammar:

Two parse trees for $a^i b^i c^i$

Limitations of CFGs

CFGs are not powerful enough to describe some languages.

Example:

- The language consisting of strings with one or more a's followed by the same number of b's then the same number of c's. *I.e.*, $\{ a^i b^i c^i \mid i \geq 1 \}$.
- $\{ a^m b^n c^m d^n \mid m, n \geq 1 \}$.

Research question: Exactly what things can and cannot be expressed with a CFG?

Research question: Can we write an algorithm which examines an arbitrary CFG and tells if it is ambiguous or not? – *Undecidable!*

Research question: Is there an algorithm that can examine two arbitrary CFGs and determine if they generate the same language? – *Undecidable!*

The Chomsky Hierarchy

Recall: There are several categories of grammar that are more and less expressive, forming a hierarchy:

Phrase-structure grammars

Context-sensitive grammars

Context-free grammars

Regular grammars

This is called the Chomsky hierarchy, after linguist Noam Chomsky, who did much of the original research.

Regular vs. Context-Free Languages

Regular languages are simpler than programming languages (e.g., numbers, identifiers).

- Context-free grammars can describe nested constructs, matching pairs of items.
- Regular grammars can only describe linear, not nested, structure.

Using CFGs for PL Syntax

Some aspects of programming language syntax can't be specified with CFGs:

- Cannot declare the same identifier twice in the same block.
- Must declare an identifier before using it.
- A[i,j] is valid only if A is two-dimensional.
- The number of actual parameters must equal the number of formal parameters.

Other things are awkward to say with CFGs:

- Identifier names must be no more than 50 characters long.

These aspects of a programming language are usually specified informally, separately from the formal grammar.

Implementations

The Translation Process

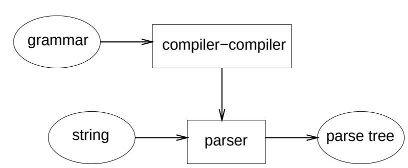
1. Lexical Analysis: Converts source code into sequence of tokens.
We use **regular grammars** *and* **finite state automata** *(recognizers).*

2. Syntactic Analysis: Structures tokens into initial parse tree.
We use **CFGs** *and* **parsing algorithms**.

3. Semantic Analysis: Annotates parse tree with *semantic actions.*

4. Code Generation: Produces final machine code.

Compiler-compilers



Examples:

- `yacc` ("yet another compiler-compiler").
  See: `man yacc`.
- `bison` (the GNU replacement for yacc)
- JavaCC.
  See: `http://www.webgain.com/products/java_cc`

So why does anyone still write compilers by hand?

Parsing Techniques

Two general strategies:

- Bottom-up: Beginning with the leaves (the sentence to be parsed), work upwards to the root (the start symbol).
- Top-down: Beginning with the root (the start symbol), work downwards to the leaves (the sentence to be parsed).

Recursive descent parsing (top-down)

Every non-terminal is represented by a subprogram that parses strings generated by that non-terminal, according to its production rules.

When it needs to parse another non-terminal, it calls the corresponding subprogram.

Requires: No left-recursion in the productions; ability to know which RHS applies without looking ahead.

Addressing the "no left-recursion" problem

Problem: Left Recursion

```
<expr> --> <expr> + <term> | <term>
```

Possible Solutions:

1. Right Recursion? E.g.,

```
<expr> --> <term> | <term> + <expr>
```

2. Left Recursion Removal, E.g.,

```
<expr> --> <term> {+ <term>}
```

3. Left Factoring, E.g.,

```
<expr> --> <term> [+ <expr>]
```

The EBNF corresponds to the code you'd write.

Other Applications of Formal Grammars

Identifying strings for an operating system command

Examples
(Unix commands that use extended REs):

- `ls s[y-z]*`
- `grep Se.h syntax.tex`
- Scripting languages like `awk` use regular expressions.
  `awk '/to[kg]e/ {print $1}' syntax.tex`

Voice recognition

Problem: Given recorded speech, produce a string containing the words that were spoken.

Difficulties:

How can a grammar help?