CS 5363, Fall 2013
Lecture Notes 2 (revised 9/16/13)
Context Free Grammars

1 Reading / Source Material

- Cooper and Torczon: Sections 3.1–3.2
- Scott: Section 2.1

2 Key concepts

context free grammars, Backus-Naur Form (BNF), derivation, parse-trees, ambiguity, expressions, operator precedence and associativity

3 Objectives: Be able to ...

1. given a BNF grammar be able to derive sentences using that grammar
2. given a BNF grammar and a (simple) sentence from the language described by that grammar, diagram the parse tree for that sentence
3. given the derivation of a sentence using a BNF grammar, be able to determine whether the sentence is a left-most derivation, a right-most derivation, or neither
4. given a BNF grammar and a parse tree, write the corresponding right-most (or left-most) derivation
5. given an ambiguous BNF grammar and a (simple) sentence that has multiple rightmost derivations from the language described by that grammar, diagram multiple parses tree for that sentence
6. given a BNF grammar for an expression language, determine whether one rule represents an operation of higher precedence than another
7. given a BNF grammar for an expression language, determine whether a rule represents a left-associative or right-associative operation

4 Outline

1. preliminaries:
   (a) sentence = string of symbols
   (b) formal language set of sentences
   (c) grammar a way of describing a language with
      i. set of symbols Σ
      ii. terminal symbols $T \subseteq \Sigma$
      iii. non-terminal symbols $N = \Sigma - T$
      iv. start symbol $S \in N$
      v. production rules P

2. context free grammar, each production rule:
    - non-terminal "derives" string of symbols
3. BNF - notation for context free grammars
4. derivations
5. parse-trees
6. program structure and grammars
    (a) in formal languages:
        i. a language is a set of sentences (i.e., a set of (finite) strings of symbols)
        ii. a grammar is a way of defining a language
    (b) in compilers and programming languages work:
        i. care about the program's structure
            - i.e., how constructs are nested
        ii. a grammar provides a recursively-defined structure for a programming language
        iii. i.e., we can use parse trees to represent programs and their structure
7. ambiguous grammars
    (a) example of an ambiguous expression grammar

```
<e>   ::= <e> <op> <e> | <num> | (<e>)
<num> ::= 1 | 2 | 3 | 4 | 5
<op>  ::= + | − | * | /

3 − 1 − 2     // associativity: does it mean (3 − 1) − 2 or 3 − (1 − 2) ?
3 * 2 + 4     // precedence: does it mean (3 * 2) + 4 or 3 * (2 + 4)?
```

    (b) requiring explicit ordering with parentheses (changes language)

```
<e>   ::= <f> <op> <f> | <f>
<f>   ::= <num> | (<e>)
<num> ::= 1 | 2 | 3 | 4 | 5
<op>  ::= + | − | * | /
```

    (c) enforcing associativity of minus by limiting recursion
    - ambiguous ambiguous with both left- and right-recursion

```
<s> ::= <e>
<e> ::= <e> − <e> | <num>
<num> ::= 1 | 2 | 3 | 4 | 5
```

    - left-recursion ⇒ left-associative structure

```
<s> ::= <e>
<e> ::= <e> − <num> | <num>
<num> ::= 1 | 2 | 3 | 4 | 5
```

    - right-recursion ⇒ right-associative structure
      The following grammar is not what would normally be expected, it creates a parse tree for 5 - 3 - 2 structured like 5 - (3 - 2) instead of one structured like (5 - 3) - 2:

```
<s> ::= <e>
<e> ::= <num> − <e> | <num>
<num> ::= 1 | 2 | 3 | 4 | 5
```

But one usually does want right-associativity for exponentiation:

```
<s> ::= <e>
<e> ::= <num> ^ <e> | <num>
<num> ::= 1 | 2 | 3 | 4 | 5
```

(d) enforcing +/* precedence with stratified productions

```
<s> ::= <e>
<e> ::= <e> + <e> | <f>
<f> ::= <f> ∗ <f> | <num> | (<e>)
<num> ::= 1 | 2 | 3 | 4 | 5
```

(e) unambiguous grammar with left-associativity and precedence of + and - = low, * and / = medium, () = high.

```
<s> :: = <e>
<e> ::= <e> + <t> | <e> − <t> | <t>
<t> ::= <t> ∗ <f> | <t> / <f> | <f>
<f> ::= (<e>) | 1 | 2 | 3 | 4 | 5
```

Note: the order the productions are written in the grammar doesn't matter. What matters is that you can rank the non-terminal symbols, such that in every production from that non-terminal, the left- and right-most symbol on the right-hand-side of that production is either the non-terminal itself, a terminal symbol, or a non-terminal of a higher rank. The structure of the resulting parse trees will, thus, reflect higher precedence for the operators associated with productions of non-terminal symbols of a higher rank.

(f) another case of ambiguity: dangling else

```
<stmnt> ::= if <expr> then <stmnt> | if <expr> then <stmnt> else <stmnt>
<stmnt> ::= skip
<expr> ::= true | false

if true then if true then skip else skip
```

5 Questions

- Consider the following grammar:

```
<S> ::= <A>
<A> ::= <A> ! <B> | <A> @ <B> | <B>
<B> ::= <C> # <B> | <C> % <B> | <C>
<C> ::= 1 | 2 | 3 | 4 | 5
```

Assuming !, @, #, and \% are meant to be operators and are evaluated as is natural for the parse tree...

1. Is the grammar ambiguous?
2. Which of the operators are left-associative?
3. Which of the operators are right-associative?

4. Which of the operators have the highest level of precedence?
5. Which of the operators have the lowest level of precedence?

- Consider the following grammar:

```
<S> ::= <A>
<A> ::= <A> ! <A> | <A> @ <A> | 1 | 2 | 3 | 4 | 5
```

1. Is the grammar ambiguous?
2. Give two rightmost derivations of `1 ! 2 @ 3` using this grammar.
3. Draw two different parse trees for `1 ! 2 ! 3` using this grammar.

- Consider the following BNF grammars:

Grammar A

```
<S> ::= <A>
<A> ::= tick tock <A>
<A> ::= ϵ
```

Grammar B

```
<S> ::= <A>
<A> ::= <A> tick tock
      | ϵ
```

Grammar C

```
<S> ::= <A>
<A> ::= tick tock <A>
      | tick tock
```

For each answer the question, Is the grammar ambiguous? If so, give two different rightmost derivations for the same sentence. If not, try to explain, why it is not possible for there to be two rightmost derivations.

- Cooper, Chapter 3, Ex. 3 (p. 157) [Warning: this is a trick question.]