Enderton 1.3: Unambiguity of grammars / Uniqueness of parse trees

October 2, 2012

1 Basic Result

We need to show that there is just one way to form a wff using the formula-building rules. That is, there is just one parse tree. The parentheses insure this. (For the wff ϕ = $((\mathbf{A} \vee \mathbf{B}) \wedge (\mathbf{C} \vee \mathbf{D}))$, it doesn't matter whether we form the fragment $(\mathbf{A} \vee \mathbf{B})$ first or $(\mathbf{C} \vee \mathbf{D})$ first; the tree is the same.)

The problem is that, without parentheses, $\mathbf{A} \vee \mathbf{B} \wedge \mathbf{C}$ could be read as $(\mathbf{A} \vee \mathbf{B}) \wedge \mathbf{C}$ or $\mathbf{A} \vee (\mathbf{B} \wedge \mathbf{C})$. There would be no good way to assign semantics to this.

Recall that earlier we showed that a non-empty proper prefix of a wff has more open parens than close parens. It follows that a non-sentence symbol wff phi is of the form (alpha conn beta), where alpha and beta have balanced parentheses (alpha may be empty), and conn is one of the five connectives. Furthermore, there is a unique way to write phi this way.

So an algorithm can find the connective at the root of the parse tree by counting parentheses from left to right.

1.1 Problems and Issues

The parsing algorithm described above may need to read in the entire wff before making any decisions. It may need time $\Omega(n^2)$ to process a wff of length n. Neither is desirable.

2 Related examples

2.1 Natural Language

Here is a (very ambiguous) grammar from natural language. The pipe symbol | indicates a choice of several rules. So, for example, a Sentence can be a noun-phrase followed by verb, or a noun-phrase, (transitive) verb, and noun-phrase (object). In places, I've included words in square brackets that are optional but may improve readability.

Sentence	:	noun-phrase verb | noun-phrase verb noun-phrase
noun-phrase	:	adjective noun | noun | noun-phrase relative-clause
relative-clause	:	[that] noun-phrase verb
noun	:	badger | bovine | butterfly | buffalo
verb	:	bother | bully | bedevil | buffalo
adjective	:	big | blue | Boston | Binghamton | Buffalo

E.g., a sentence could be

[The] Binghamton badgers [, that] Boston bovines bedevil [,] bother blue butterflies.

(The commas and "the" actually change the meaning somewhat in English.)

or

Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo.

There is more than one way to parse this; find some! In fact, there are exponentially-many ways to parse a string of n buffalos. Ignore capitalization for this exercise.

2.2 Parenthesized wffs, revisited

Here is an alternative grammar for our wffs. We'll use S ("start") for wffs, C for connectives, and A for alphabet symbols.

$$\begin{array}{lcl} S & : & A \,|\, (\neg S) \,|\, (SCS) \\ C & : & \wedge \,|\, \vee \,|\, \rightarrow \,|\, \Leftrightarrow \\ A & : & \mathbf{A} \,|\, \mathbf{B} \,|\, \mathbf{C} \,|\, \cdots \end{array}$$

With this grammar, we claim:

1. We derive exactly the same set of wffs as before.
2. A wff of length n can be parsed in time $O(n)$.
3. A wff can be parsed left-to-right. To decide which rule to apply (*i.e.*, for wffs), a parser need only look at one unread character in addition to the currently read symbol (*i.e.*, does the suffix start with a letter, with "$(\neg$," or with "$(($").

Wffs can be parsed using a *stack machine*, in which the unbounded memory consists of a stack. The last item *pushed* onto the stack is the only one available to be read, by a *pop* operation. Following that pop, the previously-pushed item is available, etc.

The algorithm is as follows. Start with "wff" on the stack. Repeatedly pop the stack, getting a *variable*, v. Determine from the input (suffix of a supposed wff) which *rule* applies, and push the variables and *terminals* on the right-hand-side of the rule onto the stack, or consume input without pushing anything.

An example should make this clear. Time proceeds down.

stack	Remaining wff
S	$((\mathbf{A} \vee (\neg \mathbf{B})) \wedge \mathbf{C})$
(SCS)	$((\mathbf{A} \vee (\neg \mathbf{B})) \wedge \mathbf{C})$
$SCS)$	$(\mathbf{A} \vee (\neg \mathbf{B})) \wedge \mathbf{C})$
$(SCS)CS)$	$(\mathbf{A} \vee (\neg \mathbf{B})) \wedge \mathbf{C})$
$SCS)CS)$	$\mathbf{A} \vee (\neg \mathbf{B})) \wedge \mathbf{C})$
$ACS)CS)$	$\mathbf{A} \vee (\neg \mathbf{B})) \wedge \mathbf{C})$
$CS)CS)$	$\vee(\neg \mathbf{B})) \wedge \mathbf{C})$
$S)CS)$	$(\neg \mathbf{B})) \wedge \mathbf{C})$
$(\neg S))CS)$	$(\neg \mathbf{B})) \wedge \mathbf{C})$
$\neg S))CS)$	$\neg \mathbf{B})) \wedge \mathbf{C})$
$S))CS)$	$\mathbf{B})) \wedge \mathbf{C})$
$A))CS)$	$\mathbf{B})) \wedge \mathbf{C})$
$))CS)$	$)) \wedge \mathbf{C})$
$)CS)$	$) \wedge \mathbf{C})$
$CS)$	$\wedge \mathbf{C})$
$S)$	$\mathbf{C})$
$A)$	$\mathbf{C})$
$)$	$)$

You can play with this using the MATLAB function `popupparse`. Select "full" grammar (for "full parentheses").

2.3 Dropping Parentheses

At the end of section 1.3, Enderton gives (very informally) rules for dropping some parentheses and still recovering the wff unambiguously. In my view, the definition is somewhat lacking. Unambiguity is asserted, but not proven. The expression $(\mathbf{A})$ is *not* allowed.

A grammar for the partially-parenthesized wffs is given under `popupparse`. You can check that, for each left-hand-side of a rule and each possible upcoming character, at most one non-empty right-hand side applies. (Apply the empty right-hand side, when present, only as a last resort.) We won't prove that this grammar is correct (which means that it derives the collection of expressions that you expect—we've never defined these), but you can play with it on MATLAB.