COMS22201: 2015/16

Language Engineering (Semantics)

Dr Oliver Ray
(csxor@Bristol.ac.uk)

Department of Computer Science
University of Bristol

Tuesday 2nd February, 2016

Week 14 Lab: Question 1

1. Consider the following C program, which uses the variable `y` to compute the factorial of the value stored in the variable `x`:

   ```
   int y=x; while (x --> 1) y *= x;
   ```

   Note that the syntax '`-->`' is intended to represent a 'down-to' operator that decrements the variable on its left (while returning `true`) until it reaches the value of the expression on its right (when it returns `false`).

   (a) First explain how this program actually works, given that the C language does not officially support such a 'down-to' operator.

   (b) Now use a loop invariant to prove this program does in fact compute `x`! for all `x` > 0.

Week 14 Lab: Question 1

Given the initial program:

```
int y = x ; while (x --> 1 ) y *= x ;
```

Rewrite using whitespace to clarify the syntactic structure:

```
int y = x ; while (x-- > 1 ) y *= x ;
```

Rewrite using assignments to make explicit any side-effects:

```
int y = x ;
while (x > 1 ) {
      x = x-1 ;
      y = y*x ;
}
x = x-1 ;
```

Week 14 Lab: Question 1

Given the equivalent program:

```
int y = x ;
while (x > 1 ) {
      x = x-1 ;
      y = y*x ;
}
x = x-1 ;
```

Add Pre- and Post-Conditions:

```
// pre: x0 > 0
int y = x ;
while (x > 1 ) {
      x = x - 1 ;
      y = y * x ;
}
x = x - 1 ;
// post: y = x0 !
```

Week 14 Lab: Question 1

Given Pre- and Post-Conditions:

```
// pre: x0 > 0
int y = x ;
while (x > 1 ) {
      x = x - 1 ;
      y = y * x ;
}
x = x - 1 ;
// post: y = x0 !
```

Trace loop for some particular values (e.g. `x`=4):

loop var	0	1	2	3
x	4	3	2	1
Y	4	12	24	24

Week 14 Lab: Question 1

Check that the invariant

$y.(x-1)! = x_0! \ \wedge \ x \geq 1$

Satisfies

(a) Initialisation:

as $x = y = x_0 > 0$ we have $y.(x-1)! = x.(x-1)! = x! = x_0!$ and $x \geq 1$ ☺

(b) Termination:

if $y.(x-1)! = x_0! \wedge x \geq 1$ and $\neg(x > 1)$ then $x=1$ and $y = x_0!$ ☺

(c) Maintenance

if $y.(x-1)! = x_0! \wedge x \geq 1$ and $x > 1$ - before
and $x' = x - 1$ and $y' = y.(x-1)$ - after
then $y'.(x'-1)! = y.(x-1)(x-1-1)! = y.(x-1)! = x_0!$ and $x' \geq 1$ - done ☺

Week 14 Lab: Question 2

2. Consider the language of signed decimal numerals $(\ldots, -1, 0, 1, 2, 3, \ldots)$

   (a) Write an EBNF grammar for this language which ensures there is exactly one numeral representing each and every integer.

   (b) Convert your grammar to BNF.

   (c) Explain how you can represent such numerals using Haskell data types.

   (d) Write a Haskell function that computes the integer associated with each such numeral.

lecture 3 slide 47 | lecture 3 slide 45 | lecture 3 slide 37

Week 14 Lab: Question 2

- We can distinguish zero and non-zero digits

  ⟨non-zero-digit⟩ ::= '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

  ⟨zero-digit⟩ ::= '0'

  ⟨digit⟩ ::= ⟨zero-digit⟩ | ⟨non-zero-digit⟩

- We can now define numerals in EBNF

  ⟨numeral⟩ ::= '0' | '–'? ⟨non-zero-digit⟩ ⟨digit⟩*

- Or we can use an auxiliary rule to represent this in pure BNF

  ⟨natural⟩ ::= ⟨non-zero-digit⟩ | ⟨natural⟩ ⟨digit⟩

  ⟨numeral⟩ ::= '0' | ⟨natural⟩ | '–' ⟨natural⟩

- And we can represent the grammar using data types in Haskell

```
data NonZeroDigit = One | Two | Three | Four | Five | Six | Seven | Eight | Nine
data ZeroDigit = Zero
data Digit = ZD ZeroDigit | NZD NonZeroDigit
data Natural = Val NonZeroDigit | Seq Natural Digit
data Numeral = Nul ZeroDigit | Pos Natural | Neg Natural
```

Week 14 Lab: Question 2

- We can write some example numerals

```
n = Neg (Seq (Seq (Val Three) (NZD Two)) (NZD One))
m = Pos (Seq (Seq (Val One) (ZD Zero)) (NZD Two))
```

- And we can convert digits, naturals and numerals to integers

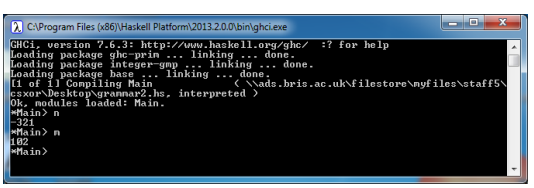
```
dig2int (ZD Zero)   = 0
dig2int (NZD One)   = 1
dig2int (NZD Two)   = 2
dig2int (NZD Three) = 3
dig2int (NZD Four)  = 4
dig2int (NZD Five)  = 5
dig2int (NZD Six)   = 6
dig2int (NZD Seven) = 7
dig2int (NZD Eight) = 8
dig2int (NZD Nine)  = 9

nat2int (Val d) = dig2int (NZD d)
nat2int (Seq n d) = (nat2int n)*10 + (dig2int d)

num2int (Nul _) = 0
num2int (Pos n) = nat2int n
num2int (Neg n) = -(nat2int n)
```

Week 14 Lab: Question 2

- We can print numerals

```
instance Show Numeral where show n = show (num2int n)
```

- And check the result in GHCI



Week 14 Lab: Question 3

3. Prove for all $n > 0$ there is a legal English sentence of the form

   Buffalo^n.

   In other words prove there is an infinite sequence of grammatically correct sentences of the form

   Buffalo. Buffalo buffalo. Buffalo buffalo buffalo. ...

   Hint: the word 'buffalo' can be a noun (i.e. a bison-like animal), an adjectival noun (i.e. relating to the city of Buffalo in the state of New York), or a verb (meaning to bully or intimidate)!

Week 14 Lab: Question 3

- The first point to note is that the required proof can only be conducted with respect to a formal grammar describing (at least some subset of) the English language. Our first step is to define a suitable grammar; and for this we can use the one from last week's lecture slides.
- Then we can prove the theorem by induction on the length n of the sentence. Our next step is to work out the trick that will make this possible. It is not difficult to see that we can use relative clauses to increase the length of a Buffalo sentence by 2 – which suggests we can work separately on even and odd length sentences.
- Although the base cases will turn out to be very easy, we do need to be very precise about the induction hypothesis and to formalise exactly what it is we are trying to prove. So our final step will be to pin this down and complete the proof.

Week 14 Lab: Question 3

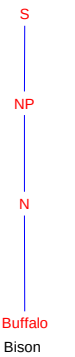
It suffices to start with the following English grammar fragment:

S	→	NP | NP VP	Sentence	
NP	→	N | NP RC | NA N	Noun Phrase	
VP	→	V NP	Verb Phrase	
RC	→	NP V	Relative Clause	
NA	→	PN	Noun Adjunct	
V	→	buffalo	Verb	(to bully)
N	→	buffalo	Noun	(a bison)
PN	→	Buffalo	Proper Noun	(NY state)

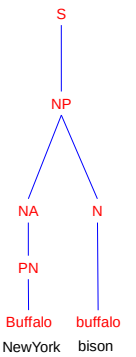
And then prove the theorem by induction on the length n of the sentence:

Week 14 Lab: Question 3

For n=1 there is exactly one parse:



For n=2 there is exactly one parse:

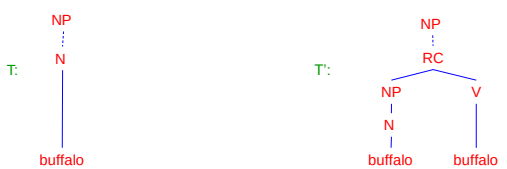


Week 14 Lab: Question 3

We will now formally show that for all n>0 there is a legal parse tree of the sentence buffalo^n in which there is at least one subtree of the form T below (on the left).

BASE CASES (n=1 and n=2): Both base cases already shown by construction above.

INDUCTIVE STEP (n=k for some k>2): Assume there is a legal parse of the sentence buffalo^{k-1} in which there is at least one subtree of the form T shown below. Chose any such subtree and replace it by one of the form T' shown below



Then result is a parse tree of buffalo^k with at least one subtree of the form T.

Week 14 Lab: Question 4

4. Observe for all $n > 0$ there is a legal English sentence of the form
   A white male (whom a white male)^n (hired)^n hired a white male.
   Use this fact to prove that English is *not* a regular language.

Week 14 Lab: Question 4

- It is important to note is this result can't be shown by simply providing a non-regular subset of English.

  That would demonstrate nothing because it is well known that every non-finite language has infinitely many non-regular sub-languages by simple counting arguments!

- Your approach should be show that set of L sentences of this form is a non-regular language N which is the intersection of English with a regular language R.

  So you need to find the regular language R and show the non-regularity of N. Then the result follows by contradiction using the closure of regular languages under intersection.

- For the non-regularity proof you could consider using the Pumping Lemma.

  But this won't work here if it is believed that the adjective can be indefinitely repeated (e.g. "A white white male" could mean a "A very very white male")?

  If so, then you would need to reason directly about regular automata.