

LARGE CS3723 HOMEWORK #3, DUE: 2pm, 2/19/08

Note: If you collaborated your classmates or used their notes, please note which classmates you collaborated. If you use an external source, besides the text book, lectures, notes provided by the instructor, and your own intellect, please cite that source. Use quote marks if you are quoting material word-for-word from any source (including the text book).

V. FORMAL LANGUAGES / BNF / PARSING (Section 4.1, Lecture 2/12)

1. Order the following types of languages/grammars according to the Chomsky Hierarchy: context-free, context-sensitive, recursively enumerable, and regular. That is, rank them from most general (most difficult to parse) to least general (easiest to parse).
2. Which of these type of languages from the Chomsky hierarchy do BNF grammars describe?
3. What does it say about a grammar if there it can produce a sentence which has more than one parse tree with that grammar?

Consider the BNF grammar for arithmetic expressions:

```
<expr> ::= <expr> + <expr> | <expr> - <expr> | <expr> × <expr>
          | (<expr>) | <num>
<num>  ::= 0 | 1 | 2 | 3 | 4 | ...
```

4. Give a derivation showing how to produce each of the following expressions from $\langle \text{expr} \rangle$:
(a) $1 + 1$ (b) $1 + 1 \times 2$ (c) $(1 + 1) \times 2$ (d) $5 - 3 - 2$
5. Draw a parse tree showing how to produce each of the following sentences from the non-terminal $\langle \text{expr} \rangle$:
(a) $1 + 1$ (b) $1 + 1 \times 2$ (c) $(1 + 1) \times 2$ (d) $5 - 3 - 2$
6. Is there more than one parse tree showing how to produce each of the following sentences for the non-terminal $\langle \text{expr} \rangle$? If so draw a second one, and indicate which if any of your parse trees are correct according to arithmetic's standard rules of precedence (i.e., multiplication has higher precedence than addition) and associativity (i.e., subtraction is left associative).
(a) $1 + 1$ (b) $1 + 1 \times 2$ (c) $(1 + 1) \times 2$ (d) $5 - 3 - 2$
7. What are lex and yacc? How are they different?

Consider the following BNF grammar for lambda terms:¹

```
<term> ::= <var> | <term><term> | λ<var>.<term> | (<term>)
<var>  ::= x | y | z | ...
```

8. Give two derivations showing how to produce each of the following sentences from the non-terminal $\langle \text{term} \rangle$:
(a) $\lambda x.x$ (b) $\lambda x.xx$ (c) $\lambda f.(\lambda x.fxx)$ (d) $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
9. Draw a parse tree showing how to produce each of the following sentences from the non-terminal $\langle \text{term} \rangle$:
(a) $\lambda x.x$ (b) $\lambda x.xx$ (c) $\lambda f.(\lambda x.fxx)$ (d) $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$
10. Is there more than one parse tree showing how to produce each of the following sentences for the non-terminal $\langle \text{term} \rangle$? If so draw a second one, and indicate which if any of your parse trees are correct according to lambda calculus's rules of precedence (i.e., application has higher precedence than abstraction) and associativity (i.e., application is left associative).
(a) $\lambda x.x$ (b) $\lambda x.xx$ (c) $\lambda f.(\lambda x.fxx)$ (d) $\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$

¹for comparison (cf.), see the grammar on Mithcell page 59

VI. COMPILATION AND INTERPRETATION (Section 4.1, Lecture 2/14, R5RS)

1. What is the name of the process by which a program written in some programming language is *translated* into instructions for some hardware machine?
2. What kind of program executes another program 'P' written in some programming language 'L' by simulating 'L' and *directly performing* the tasks required by the statements in 'P' as it executes those statements?
3. What advantages does compilation have over interpretation? What advantages does interpretation have over compilation?
4. Which is usually faster, the interpretation of some program directly or the direct execution of machine code compiled from it?
5. Why is it a bit of a misnomer to say that a language is an 'interpreted' rather than a 'compiled' language? Explain more precisely what is meant when someone says *X* is an interpreted language.
6. What is a virtual machine (in the sense of the Java Virtual Machine)? And what role does the just-in-time (JIT) compiler play in the virtual machine?
7. List six compilation phases that might be found in a typical optimizing compiler. Pick one and briefly describe what happens during that phase.
8. List three optimizations that can be applied during compilation. Briefly explain what one of them does.
9. The interactive prompt (like in the bottom pane of Dr. Scheme) for Lisp is traditionally defined using a read-eval-print loop (REPL): `(loop (print (eval (read))))` What does the `eval` function in Scheme/Lisp do?

EXTRA CREDIT

1. In Homework #1, you were asked to explain how to build a Halt program that takes a program *P* and an input *I* and determines whether *P* halts on *I* using a magical Halt_0 routine that can determine whether an input-less program halts. (The answer to this question, by the way, is basically giving a proof by contradiction that Halt_0 , like Halt, is undecidable.) The solution was to have Halt rewrite *P* so that the input is hard-coded.

Write a Scheme function *H*. *H* should take two list parameters: (1) a list *P* representing a Scheme expression that requires input and (2) a list *I* that should be the input for *P*. *H* should return a new list *P'*. *P'* should represent an expression that does not require input but will halt if and only if *P* would have halted if it was given the input *I*.

You may assume the program *P* is of the form:

```
(let (... (input (read)) ...) (...))
```

where `(read)` is a function that reads in input, parses it into a list and returns that list.

You may wish to use `eval` to test your function *H* by evaluating *P'*.

2. In class, we discussed how the fixed-point combinator **Y** can be used to build a recursive function. It was also mentioned that the fixed-point combinator **Y** will not work if translated directly into Lisp/Scheme, because Lisp/Scheme will evaluate the parts of the expression in the wrong order and never terminate. There is, however, a variant of **Y**, called the call-by-value (or applicative-order) fixed-point operator: $\mathbf{Z} \equiv \lambda f.(\lambda x.f(\lambda y.(xx)y))(\lambda x.f(\lambda y.(xx)y))$. This can be translated to Scheme as:

```
(define Z (lambda (f) ((lambda (x) (f (lambda (y) ((x x) y)))) (lambda (x) (f (lambda (y) ((x x) y)))))))
```

Define a non-recursive Scheme function **F**, such that when **Z** is applied to **F**, the result will be a recursive function. And that this recursive function, when applied to some number *n*, results in the *n*th number in the Fibonacci sequence. For example: `((Z F) 0) ⇒ 0 ((Z F) 1) ⇒ 1 ((Z F) 2) ⇒ 1 ((Z F) 8) ⇒ 21`

Now, this isn't a practical use of higher-order functions, since it is much simpler to just use `define` or `letrec` so that the recursive function can simply call itself. But it does demonstrate how powerful higher-order functions are, in that they even subsume recursion.