Name: SHIVAM VERMA Entry: 2012CS10253 Gp: 3

3

Indian Institute of Technology Delhi
Department of Computer Science and Engineering

CSL302
February 7, 2014

Programming Languages
16:00–17:00

Minor I
Maximum Marks: 60

Open notes. Write your name, entry number and group at the top of each sheet in the blanks provided. Answer all questions in the space provided, in blue or black ink (no pencils, no red pens). Budget your time according to the marks. Do rough work on separate sheets.

Q0. (1x4 marks) **Lexical and syntactic analysis.** Hopefully, you have been programming in OCaml. Write an *example* each of simple but incorrect OCaml expressions that will produce

1. a lexical error detected by the scanner: ~~the~~ tpye var = Var of itn;; not lexical

2. a syntax error, detected by the parser: type var = ~~var~~ ant of int;; should be ca

3. a type error, detected by the type checker: type var = ~~Ant of var~~ Var of ant;;

4. a run-time error, detected during program execution:

1/4

Q1. (8 marks) **Compilers and Interpreters.** *Byte code* is the code that is interpreted on a *virtual machine*, whereas *native code* is machine code for a physical machine, and *source code* is program code written in a high-level programming language. Remember a program in source code cannot run, unless it is translated into byte code to be run on a virtual machine that is running on a physical machine, or else translated into native code for the physical machine. Suppose I have a *Macos* machine and a *Wintel* machine, as well as the following software objects:

1. The source code, written in language *j*, of a compiler *jc*; *jc* takes programs in language *j*, and produces *native code* for a *Macos*-machine;
2. A compiler *jbc* already compiled into *byte code* language *B*; *jbc* takes programs in language *j* and produces code in byte code *B*;
3. A byte-code *B*-interpreter BI_{Wintel} (this interpreter is in native code), which runs on a *Wintel*-machine, which takes code in *B* and interprets it.

Using the notation $X(Y)$ to mean running the software entity X on input Y, and in particular $BI_{Wintel}(X)$ to denote running the byte-code interpreter BI_{Wintel} to interpret (execute) the B program called X, indicate the steps by which, following a small variant on boot-strapping, I can produce a native code compiler for language *j* programs, which runs on *Macos*-machines (and produces native *Macos*-machine code). Name the output objects (files/programs/...) of each step appropriately.

Step 1.

Step 2.

0/8

~~BWintel~~

Step 1. jbc(X) → B

Step 2. BWintel(B) → native code

Q2. (8 marks) **Type checking** Consider the simple OCaml program `filter`, and determine its type by solving the various type constraint *equations*. I have already provided you with the typing constraints for various sub-expressions. You have to formulate the equations between the various type expressions, and solve them.

```
let rec filter p l = match l with           δ list        ε list
                                [ ] -> [ ]
                              | (x::xs) -> if (p x) then x :: (filter p xs)
                                                    else filter p xs
;;
```

p : α

l : β

rhs : γ

lhs [] : δ list

rhs [] : ϵ list

x : θ

xs : θ list

(p x) : bool

filter p xs : ξ

filter : p → (δ list → ε list) ~~→ ((p → θ) → bool)~~ ~~ξ~~

92

Q3. (6 marks) **Lexical Analysis.** Consider a simplified version of datatype definitions in OCaml. Such a definition consists of the keyword **type** followed by an *identifier* starting with a lowercase letter (the datatype being defined), then an "=" sign followed by a series of *cases* separated by the vertical bar "|". Each case has the following form: an identifier (the constructor name, starting with an Uppercase letter), then the key word of, followed by a *type expression*. A *type expression* τ is either an identifier or a (cartesian) product of type expressions $\tau_1 * \ldots * \tau_n$. *Parentheses* may be used around type expressions, to indicate associativity. The definition terminates with a double semicolon.

Write down a specification of the tokens that may appear in a type definition in a lex like notation.

Consider type exp = Label of int | Mult of int * int | Value of label * int

symbols: =, |, *, ;; , ()

keywords: type , of , int

user's choice : exp , Label , Mult , Value , Cube

6

Name:

Entry:

Gp:

Q4. (14 marks) Context Free Grammars. Consider the following grammar for boolean expressions:

$$B ::= p \mid \mathsf{T} \mid \mathsf{F} \mid \neg B \mid B \vee B \mid B \wedge B \mid (B)$$

where p is an element from a set of *propositional letters* $\mathcal{P}$ considered terminals. This grammar is ambiguous. Assume we wish to avoid unnecessary parenthesization so that parenthesization has highest priority, then **not** binds tighter than $\wedge$, which in turn binds tighter than $\vee$, and both $\wedge$ and $\vee$ are right associative. Provide an unambiguous grammar that is not left-recursive for this language enforcing this convention:

give the grammar

$B \to N \vee N$

$B \to N \vee B$

$N \to E \wedge E$

$N \to E \wedge N$

$E \to \neg T$

$E \to \neg F$

$T \to (B)$

$F \to (B)$

owing

$P = \{$

This is right - recursive

$K \to p \mid T \mid F$

Q5. (6+4 marks) Abstract syntax. Consider a simple imperative language consisting of commands c which can be: (i) Assignment statements of the form $x := e$, where x is an identifier and e is a numerical expression, (ii) Sequential composition of commands, (iii) A conditional "*if-then-else*" construct, with a boolean test and two branches, and (iv) An iterative "*while loop*", with a boolean test and a body which is a command. Provide the following:

1. An *abstract syntax* for commands, assuming you have abstract syntax for identifiers, expressions and boolean expressions. (Present it as an abstract grammar).

0

~~whileloop $\to$ bool * E~~

Name: Shivam Verma   Entry: 2012CS10253   3.

2. Encode the above abstract syntax as an OCaml datatype assuming datatypes *ident*, *exp*, and *bexp*.

```
type command =
```

Q6. (5+5 marks) **Σ-homomorphisms.** Consider the following signature $\Sigma = \{0^{(0)}, 1^{(0)}, +^{(2)}\}$. Let $\mathcal{T}_\Sigma$ be the corresponding term algebra.

In the following, complete the following definitions of the functions *odd* (which returns true if the input represents an odd number, and false otherwise) and *post* (which returns a post-order traversal of the term/tree), such that they are Σ-homomorphisms to the indicated Σ-algebras B and C respectively. Clearly define the meanings of the operations ⊕ and #, which represent $+_B$ and $+_C$ respectively.

(i) Let $B = (\mathbb{B}, F, T, \oplus)$

$$
\begin{aligned}
odd(0) &= T \\
odd(1) &= F \\
odd(n_1+n_2) &= \oplus(odd(n_1), odd(n_2))
\end{aligned}
$$

where $\oplus(x, y) =$ computes odd(x) and odd(y) which are both ~~boolean~~, and then applies the operation ⊕ on the ~~pair~~ of boolean

What is ⊕?

(ii) Let $C = (\{0, 1, +\}^*, "0", "1", \#)$.

$$
\begin{aligned}
post(0) &= \\
post(1) &= \\
post(n_1+n_2) &=
\end{aligned}
$$

where $\#(x, y) =$ ___