The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 3 MODULE, AUTUMN SEMESTER 2016–2017

COMPILERS

ANSWERS

Time allowed TWO hours

Candidates may complete the front cover of their answer book and sign their desk card but must NOT write anything else until the start of the examination period is announced.

Answer ALL THREE questions

No calculators are permitted in this examination.

Dictionaries are not allowed with one exception. Those whose first language is not English may use a standard translation dictionary to translate between that language and English provided that neither language is the subject of this examination. Subject-specific translation directories are not permitted.

No electronic devices capable of storing and retrieving text, including electronic dictionaries, may be used.

Note: ANSWERS

Knowledge classification: Following School recommendation, the (sub)questions have been classified as follows, using a subset of Bloom's Taxonomy:

K: Knowledge

C: Comprehension

A: Application

Note that some questions are closely related to the coursework. This is intentional and as advertised to the students; the coursework is a central aspect of the module and as such partly examined under exam conditions.

Question 1

(a) Consider the following context-free grammar (CFG):

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow bcA \mid c \\ B &\rightarrow d \end{aligned}$$

S, A, and B are nonterminal symbols, S is the start symbol, and a, b, c, d, and e are terminal symbols.

Explain how a bottom-up (LR) parser would parse the string

abcbcbccde

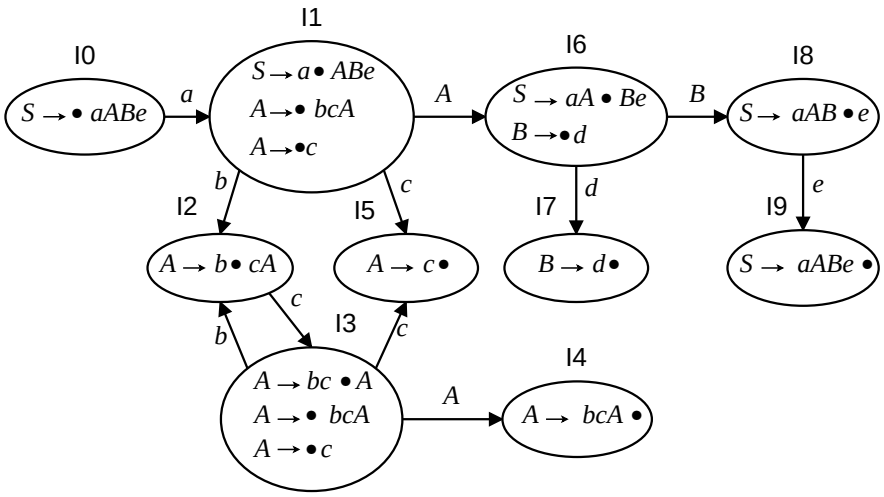
according to this grammar by reducing it step by step to the start symbol. Also state what the *handle* is for each step. (10)

Answer: *[C] A bottom-up parser traces out a right-most derivation in reverse. It would reduce the word abcbccde as follows:*

$$\begin{aligned} abcbcbc\underline{c}de &\underset{rm}{\Leftarrow} \text{(reduce by } A \rightarrow c) \\ abcbc\underline{bcA}de &\underset{rm}{\Leftarrow} \text{(reduce by } A \rightarrow bcA) \\ abc\underline{bcA}de &\underset{rm}{\Leftarrow} \text{(reduce by } A \rightarrow bcA) \\ a\underline{bcA}de &\underset{rm}{\Leftarrow} \text{(reduce by } A \rightarrow bcA) \\ aA\underline{d}e &\underset{rm}{\Leftarrow} \text{(reduce by } B \rightarrow d) \\ \underline{aABe} &\underset{rm}{\Leftarrow} \text{(reduce by } S \rightarrow aABe) \\ S & \end{aligned}$$

The handle has been underlined in each step.

(b) The DFA below recognizes the viable prefixes for the above CFG.



Show how an LR(0) shift-reduce parser parses the string *abcbcbccde* by completing the following table (copy it to your answer book; do *not* write on the examination paper):

State	Stack	Input	Move
I0	ϵ	*abcbcbccde*	Shift
I1	*a*	*bcbcbccde*	Shift
⋮	⋮	⋮	⋮
	S	ϵ	Done

(10)

Answer: *[C]*

State	*Stack*	*Input*	*Move*
I0	ϵ	*abcbcbccde*	*Shift*
I1	*a*	*bcbcbccde*	*Shift*
I2	*ab*	*cbcbccde*	*Shift*
I3	*abc*	*bcbccde*	*Shift*
I2	*abcb*	*cbccde*	*Shift*
I3	*abcbc*	*bccde*	*Shift*
I2	*abcbcb*	*ccde*	*Shift*
I3	*abcbcbc*	*cde*	*Shift*
I5	*abcbcbcc*	*de*	*Reduce by* $A \rightarrow c$
I4	*abcbcbcA*	*de*	*Reduce by* $A \rightarrow bcA$
I4	*abcbcA*	*de*	*Reduce by* $A \rightarrow bcA$
I4	*abcA*	*de*	*Reduce by* $A \rightarrow bcA$
I6	*aA*	*de*	*Shift*
I7	*aAd*	*e*	*Reduce by* $B \rightarrow d$
I8	*aAB*	*e*	*Shift*
I9	*aABe*	ϵ	*Reduce by* $S \rightarrow aABe$
	S	ϵ	*Done*

(c) Explain *shift/reduce* and *reduce/reduce* conflicts in the context of LR parsing. (5)

Answer: *[K] Shift/reduce and reduce/reduce conflicts occur as a result of ambiguities in the grammar. In a shift/reduce conflict, a state in the DFA contains both complete and incomplete items. It is thus not clear whether to shift or reduce. In a reduce/reduce conflict, a state in the DFA contains more than one complete item. It is thus not clear by what production to reduce.*

Question 2
The following is the grammar for a very simple expression language:

$$exp \;\rightarrow\; exp \texttt{ and } exp \mid exp \texttt{ or } exp \mid \texttt{not } exp \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{(}exp\texttt{)}$$

Here, *exp* is a non-terminal and `and`, `or`, `not`, `tt`, `ff`, `(`, and `)` are all terminals, with `and` denoting logical conjunction, `or` denoting logical disjunction, `not` denoting logical negation, `tt` and `ff` being literals denoting the truth values true and false respectively, and parentheses used for grouping as usual.

The following is the central part of a Happy parser specification for this grammar. We wish to implement an *interpreter* that directly evaluates a parsed expression to a Boolean. The type of the semantic value for the non-terminal `exp` is thus `Bool`:

```
exp :: { Bool }
exp : exp and exp          { 1 }
    | exp or  exp          { 2 }
    | not exp              { 3 }
    | tt                   { 4 }
    | ff                   { 5 }
    | '(' exp ')'          { 6 }
```

The grammar is ambiguous, but we assume that Happy's features for specifying operator precedence and associativity are used to disambiguate as necessary. The semantic actions for evaluating an expression have been left out, indicated by boxed numbers (like 1).

(a) Complete the fragment above by providing suitable semantic actions for evaluating the various forms of expressions. (6)

Answer: *[A]*

```
1 = $1 && $3
2 = $1 || $3
3 = not $2
4 = True
5 = False
6 = $2
```

Marking: 1 marks for each semantic action. ($6 \times 1 = 6$*)*

(b) We now wish to extend the language with a notion of let-bound variables. The Happy grammar is thus extended as follows:

```
| ident                        { 7 }
| let ident '=' exp in exp      { 8 }
```

Here, `ident`, `let`, `in`, and `=` are all new terminals. For simplicity, the semantic value of `ident` is a string; i.e., the name of the identifier.

Explain, in English, how to restructure the interpreter to handle let-bound variables. In particular, what should the type of the semantic value of the non-terminal `exp` be now? (9)

Answer: *[K,A] The key difficulty is that the value of a variable needs to be communicated from the site of its definition to each use site. As the basic flow of information is bottom-up, we need to additionally arrange for a way to also propagate information top-down. This can be achieved by turning the semantic value into a function as the function argument effectively allows information to flow downwards. In this case, we introduce an environment that maps identifiers (strings) to the value of the named variable in question. E.g.* `type Env = String -> Bool`*. (Alternatively, we could use some kind of lookup table, like a* `Map` *or a list of pairs of type* `(String, Bool)`*). The type of the semantic value of expressions then becomes* `Env -> Bool`*.*

(c) Implement an interpreter for the extended expression language by providing suitable semantic actions for all productions (1–8) following the idea you described in (b). (10)

Answer: *[A]*

```
1 = \env -> $1 env && $3 env
2 = \env -> $1 env || $3 env
3 = \env -> not ($2 env)
4 = \_ -> True
5 = \_ -> False
6 = \env -> $2 env
7 = \env -> env $1
8 = \env -> let v = $4 env in $6 (\i -> if i == $2 then v else env i)
```

Marking: 4 marks for actions 1–6 collectively, 2 marks for action 7, 4 marks for action 8. (4 + 2 + 4 = 10)

Question 3
This questions concerns types and scope: both how they are captured formally in a type system, and how they might be implemented.

(a) Consider the following expression language:

e	$\rightarrow$		*expressions:*
	$\mid$	n	*natural numbers,* $n \in \mathbb{N}$
	$\mid$	x	*variables,* $x \in$ Name
	$\mid$	e `=` e	*equality test*
	$\mid$	`if` e `then` e `else` e	*conditional*

where Name is the set of variable names. The types are given by the following grammar:

t	$\rightarrow$		*types:*
	$\mid$	`Nat`	*natural numbers*
	$\mid$	`Bool`	*Booleans*

The ternary relation $\Gamma \vdash e : t$ says that expression e has type t in the typing context Γ. It is defined by the following typing rules:

$$\Gamma \vdash n : \texttt{Nat} \qquad \text{(T-NAT)}$$

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : t} \qquad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Nat} \quad \Gamma \vdash e_2 : \texttt{Nat}}{\Gamma \vdash e_1 \texttt{ = } e_2 : \texttt{Bool}} \qquad \text{(T-EQ)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \quad \Gamma \vdash e_2 : t \quad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t} \qquad \text{(T-COND)}$$

A typing context, Γ in the rules above, is a comma-separated sequence of variable-name and type pairs, such as

$$\texttt{x} : \texttt{Nat}, \texttt{ y} : \texttt{Bool}, \texttt{ z} : \texttt{Nat}$$

or empty, denoted $\emptyset$. Typing contexts are extended on the right, e.g. Γ, `z` : `Nat`, the membership predicate is denoted by $\in$, and lookup is from right to left, ensuring recent bindings hide earlier ones.

(i) Use the typing rules given above to formally prove that the expression

```
if x = 5 then a else b
```

has type `Bool` in the typing context

$$\Gamma_1 = \texttt{a} : \texttt{Bool},\ \texttt{b} : \texttt{Bool},\ \texttt{x} : \texttt{Nat}$$

The proof should be given as a *proof tree*. (5)

Answer:

$$\dfrac{\dfrac{}{\Gamma_1 \vdash \texttt{x = 5} : \texttt{Bool}}\ \text{below} \quad \dfrac{\texttt{a} : \texttt{Bool} \in \Gamma_1}{\Gamma_1 \vdash \texttt{a} : \texttt{Bool}}\ \text{T-VAR} \quad \dfrac{\texttt{b} : \texttt{Bool} \in \Gamma_1}{\Gamma_1 \vdash \texttt{b} : \texttt{Bool}}\ \text{T-VAR}}{\Gamma_1 \vdash \texttt{if x = 5 then a else b} : \texttt{Bool}}\ \text{T-COND}$$

$$\dfrac{\dfrac{\texttt{x} : \texttt{Nat} \in \Gamma_1}{\Gamma_1 \vdash \texttt{x} : \texttt{Nat}}\ \text{T-VAR} \quad \dfrac{}{\Gamma_1 \vdash \texttt{5} : \texttt{Nat}}\ \text{T-NAT}}{\Gamma_1 \vdash \texttt{x = 5} : \texttt{Bool}}\ \text{T-EQ}$$

(ii) The expression language defined above is to be extended with let-bound variables; definition of named, possibly *recursive*, functions; and function application as follows:

e	$\rightarrow$		*expressions:*
$\ldots$	$\ldots$		$\ldots$
	$\mid$	`let var` $x = e$ `in` e	*variable definition*
	$\mid$	`let fun` f`(`x`:`t`)`$:t = e$ `in` e	*function definition*
	$\mid$	e`(`e`)`	*function application*
t	$\rightarrow$		*types:*
$\ldots$	$\ldots$		$\ldots$
	$\mid$	$t \rightarrow t$	*function (arrow) type*

Here, f is the syntactic category of function names ($f \in$ Name). Variable definition is not recursive: the let-bound variable is only in scope in the body of the let-expression, not in its defining expression. In contrast, the named function being defined is in scope, along with the named formal argument, in the expression defining the function, thus allowing for recursive functions.

For example, if we assume that the expression language has been extended with basic arithmetic operations as well, the following is a definition of the factorial function:

```
let fun fac(n : Nat) : Nat =
    if n = 0 then 1 else n * fac(n - 1)
in
    ...
```

Provide a typing rule for each of the new expression constructs, in the same style as the existing rules, reflecting the standard notions of typed let-expressions and function application augmented by the additional requirements set forth in the text above. (8)

Answer:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma,\ x : t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let var}\ x\ =\ e_1\ \texttt{in}\ e_2 : t_2} \quad \text{(T-LETVAR)}$$

$$\frac{\Gamma,\ f : t_{11} \to t_{12},\ x : t_{11} \vdash e_1 : t_{12} \quad \Gamma,\ f : t_{11} \to t_{12} \vdash e_2 : t_2}{\Gamma \vdash \texttt{let fun}\ f\texttt{(}x\texttt{:}t_{11}\texttt{)}\texttt{:}t_{12}\ =\ e_1\ \texttt{in}\ e_2 : t_2} \quad \text{(T-LETFUN)}$$

$$\frac{\Gamma \vdash e_1 : t_2 \to t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1(e_2) : t_1} \quad \text{(T-APP)}$$

(b) Consider the following code skeleton (note: *nested* procedures):

```
var a, b, c: Integer
proc P
    var x, y, z: Integer
    proc Q
        var u, v: Bool
        proc R
            var w: Bool
            begin ... Q() ... end
        begin ... R() ... end
    begin ... Q() ... end
begin ... P() ... end
```

The variables `a`, `b`, and `c` are global. The variables `x`, `y`, and `z` are local to procedure `P`, as is procedure `Q`, which in turn has two local variables, `u` and `v`, and a local procedure `R`. The latter has one local variable, `w`. The notation `P()`, `R()`, etc. signifies a call to the named procedure. Thus main calls `P`, `P` calls `Q`, `Q` calls `R`, and `R` calls `Q` (recursively).

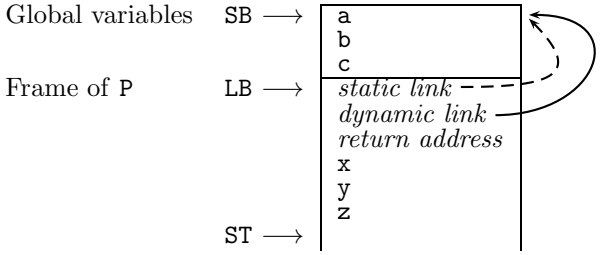
Assume stack-based memory allocation with *dynamic* and *static links*.

(i) Show the layout of the activation records on the stack after the main program has called procedure `P`. Explain how global and local variables are accessed from `P`. (3)

(ii) Show the layout of the activation records on the stack after the call sequence: `P`, `Q`, `R`, `Q`, `R` (that is, after main has called `P`, which in turn has called `Q`, etc.). Explain how global variables, `P`'s variables, `Q`'s variables, and `R`'s own local variables are accessed from the last activation of `R`. (9)

Answer:

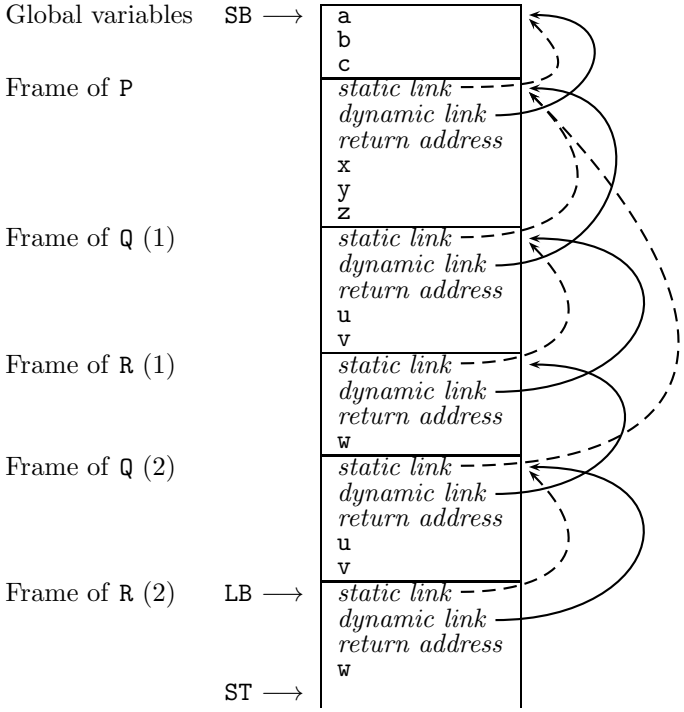
(i) Activation record layout (stack grows downwards):



SB is Stack Base, LB is Local Base (or Frame Pointer), ST is Stack Top. The activation record (or frame) of the currently active procedure/function is the one between LB and ST. The solid arrow from the current activation record represents the dynamic link, i.e. it refers to the activation record of the caller *and is thus*

equal to the previous value of LB. The dashed arrow represents the static link. Global variables are accessed relative to SB. Variables local to P are accessed relative to LB. In both cases, the offsets are statically known.

(ii) Activation record layout (stack grows downwards):



SB is Stack Base, LB is Local Base (or Frame Pointer), ST is Stack Top. The solid arrows represent the dynamic link, the dashed ones is the static link. Global variables are accessed relative to SB. Variables local to R are are accessed relative to LB. Hence the second *instance of these variables are being accessed (corresponding to the currently active procedure). As Q directly encloses R, following the static link from R's activation record takes us to the correct activation record, the most recent activation of Q, and the variables can then be found at statically known offsets within this activation record. As P is two scope levels out, we have to follow two static links: first the static link from R's activation record to get to the activation record for the immediately enclosing scope, and then the static link from that record to get to the record two levels out. The variables are again found at statically known offsets within this record.*