# The University of Nottingham

SCHOOL OF COMPUTER SCIENCE

A LEVEL 3 MODULE, AUTUMN SEMESTER 2016–2017

**COMPILERS**

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.*

***DO NOT turn examination paper over until instructed to do so***

**Question 1**

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

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

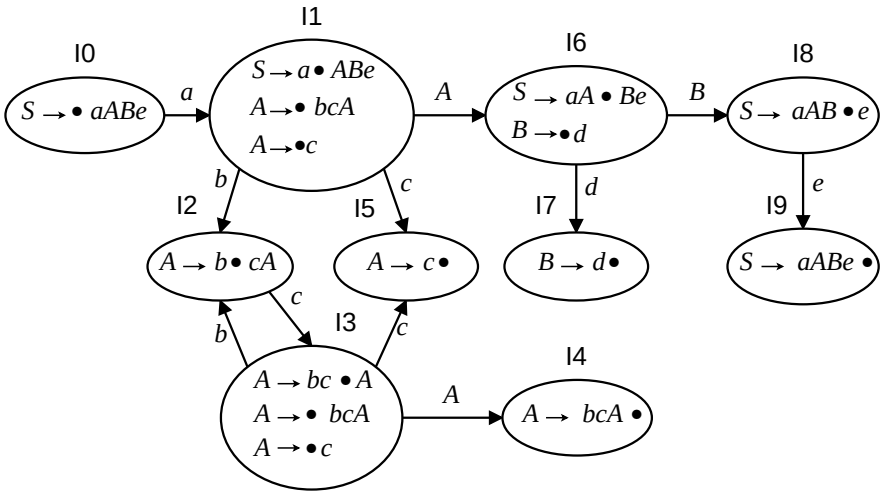
$S$, $A$, and $B$ are nonterminal symbols, $S$ is the start symbol, and $a$, $b$, $c$, $d$, and $e$ are terminal symbols.

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)

(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 | $\epsilon$ | *abcbcbccde* | Shift |
| I1 | $a$ | *bcbcbccde* | Shift |
| ⋮ | ⋮ | ⋮ | ⋮ |
| | $S$ | $\epsilon$ | Done |

(10)

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

**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)

(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)

(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)

**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 $\Gamma$. 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, $\Gamma$ 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. $\Gamma$, `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

$$\texttt{if x = 5 then a else b}$$

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)

(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:* |
| | ... | ... | ... |
| | \| | $\texttt{let var}\ x = e\ \texttt{in}\ e$ | *variable definition* |
| | \| | $\texttt{let fun}\ f\texttt{(}x\texttt{:}t\texttt{)}\texttt{:}t = e\ \texttt{in}\ e$ | *function definition* |
| | \| | $e\texttt{(}e\texttt{)}$ | *function application* |

| | | | |
|---|---|---|---|
| $t$ | $\rightarrow$ | | *types:* |
| | ... | ... | ... |
| | \| | $t \rightarrow t$ | *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)

(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)