UNIVERSITY OF BRISTOL

Winter 2025 Examination Period

SCHOOL OF COMPUTER SCIENCE

Second Year Examination for the Degrees
of
Bachelor of Science
Master of Engineering
Master of Science

COMS20007W
Programming Languages and Computation

TIME ALLOWED:
2 Hours

This paper contains *three* questions, worth *34*, *33* and *33* marks respectively. Answer *all* questions. The maximum for this paper is *100 marks*. Credit will be given for partial answers.

Other Instructions:

Candidates may bring to the exam room 1 double-sided A4 page of notes in any format. A reminder of key definitions is provided at the back of this paper. No calculators allowed.

TURN OVER ONLY WHEN TOLD TO START WRITING

Reminder of Important Definitions

Grammars

A *Context Free Grammar (CFG)* consists of four components:

- An alphabet of *terminal* symbols.
- A finite, non-empty set of *non-terminal* symbols, disjoint from the terminals.
- A finite set of *production rules*.
- A designated non-terminal called the *start symbol*.

A *sentential form*, usually α, β, γ and so on, is just a finite sequence of terminals and nonterminals.

The sentential form α can make a *derivation step* to β, written $\alpha \to \beta$, just if:

- α has shape $\gamma_1 \, X \, \gamma_2$ and β has shape $\gamma_1 \, \delta \, \gamma_2$
- and there is a production rule $X ::= \delta$ in the grammar

A *derivation sequence* is a non-empty sequence of sentential forms α_1, α_2, ... α_{k-1}, α_k in which consecutive elements of the sequence are derivation steps:

$$\alpha_1 \to \alpha_2 \to \cdots \to \alpha_{k-1} \to \alpha_k$$

A sentential form β is *derivable* from α, written $\alpha \to^* \beta$ just if there is a derivation sequence starting with α and ending with β.

We say that a word w is in the *language of a grammar* G with start symbol S, and write $w \in L(G)$ just if $S \to^* w$.

Nullable

On nonterminals:

$$\text{Nullable}(X) \text{ iff } X \to^* \epsilon$$

On sentential forms:

$$\text{Nullable}_s(\alpha) = \begin{cases} \text{true} & \text{if } \alpha = \epsilon \\ \text{false} & \text{if } \alpha \text{ is of shape } a\beta \\ \text{Nullable}(X) \wedge \text{Nullable}_s(\beta) & \text{if } \alpha \text{ is of shape } X\beta \end{cases}$$

First

On nonterminals:

$$\mathsf{First}(X) = \{a \mid \exists\beta.\, X \to^* a\beta\}$$

On sentential forms:

$$\mathsf{First}_s(\alpha) = \begin{cases} \emptyset & \text{if } \alpha = \epsilon \\ \{a\} & \text{if } \alpha \text{ is of shape } a\beta \\ \mathsf{First}(X) & \text{if } \alpha \text{ is of shape } X\beta \text{ and } \neg\mathsf{Nullable}(X) \\ \mathsf{First}(X) \cup \mathsf{First}_s(\beta) & \text{if } \alpha \text{ is of shape } X\beta \text{ and } \mathsf{Nullable}(X) \end{cases}$$

Follow

On nonterminals:

$$\mathsf{Follow}(X) = \{a \mid \exists\alpha\beta.\, S \to^* \alpha X a\beta\}$$

Parse Tables and LL(1)

We define the *parsing table*, usually T, for a given grammar as a 2d array indexed by pairs of a nonterminal and a terminal. Each entry $T[X, a]$ is a set of production rules from the grammar, such that some rule $X \longrightarrow \beta$ is in the set $T[X, a]$ just if, either:

1. $a \in \mathsf{First}_s(\beta)$
2. or, $\mathsf{Nullable}_s(\beta)$ and $a \in \mathsf{Follow}(X)$

A grammar whose parsing table contains at most one rule in each cell is called *LL(1)*.

Abstract Syntax of Arithmetic Expressions

An *arithmetic expression* is a tree described by the following grammar:

$$A ::= n \mid x \mid A + A \mid A - A \mid A * A$$

where n ranges over integer literals, and x ranges over variables. Parentheses are used to resolve ambiguity and to indicate the structure of the tree. We write $\mathcal{A}$ for the set of arithmetic expressions.

Abstract Syntax of Boolean Expressions

A *Boolean expression* is a tree described by the following grammar.

$$B ::= \mathsf{false} \mid \mathsf{true} \mid !B \mid B \;\&\&\; B \mid B \parallel B \mid A = A \mid A \leq A$$

Parentheses are used to resolve ambiguity and to indicate the structure of the tree. We write $\mathcal{B}$ for the set of Boolean expressions.

Abstract Syntax of Statements

A *statement* is a tree described by the following grammar:

$$S ::= \text{skip} \mid x \leftarrow A \mid S; S \mid \text{if } B \text{ then } S \text{ else } S \mid \text{while } B \text{ do } S$$

Braces "$\{\cdots\}$" are used to resolve ambiguity and to indicate the structure of the tree. We write $\mathcal{S}$ for the set of statements.

States

A *state* is a total function from the set State = Var $\to \mathbb{Z}$, where Var is the set of variables. We write $[x_1 \mapsto v_1,\, x_2 \mapsto v_2,\, \ldots,\, x_n \mapsto v_n]$ to indicate the state that maps the variable $x_i \in$ Var to the value $v_i \in \mathbb{Z}$ for all $i \leq n$. By convention, any variable not explicitly mentioned by a given state σ is assigned the value 0.

For a given state $\sigma \in$ State, we write $\sigma[x \mapsto v]$ for some variable $x \in$ Var and $v \in \mathbb{Z}$ to denote the state that maps the variable x to v and any other variable y to the value $\sigma(y)$.

Semantics of Arithmetic Expressions

The denotation function for arithmetic expressions $[\![\cdot]\!]_{\mathcal{A}} \in \mathcal{A} \to (\text{State} \to \mathbb{Z})$, which is defined by recursion in Figure 1. We say that two arithmetic expressions $e_1,\, e_2 \in \mathcal{A}$ are *semantically equivalent* if, and only if, $[\![e_1]\!]_{\mathcal{A}}(\sigma) = [\![e_2]\!]_{\mathcal{A}}(\sigma)$ for all states $\sigma \in$ State.

$$\begin{aligned}
[\![n]\!]_{\mathcal{A}}(\sigma) &= n \\
[\![x]\!]_{\mathcal{A}}(\sigma) &= \sigma(x) \\
[\![e_1 + e_2]\!]_{\mathcal{A}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) + [\![e_2]\!]_{\mathcal{A}}(\sigma) \\
[\![e_1 - e_2]\!]_{\mathcal{A}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) - [\![e_2]\!]_{\mathcal{A}}(\sigma) \\
[\![e_1 * e_2]\!]_{\mathcal{A}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) \cdot [\![e_2]\!]_{\mathcal{A}}(\sigma)
\end{aligned}$$

Figure 1: Definition of the denotational semantics of arithmetic expressions.

Semantics of Boolean Expressions

The denotation function for Boolean expressions $[\![\cdot]\!]_{\mathcal{B}} \in \mathcal{B} \to (\text{State} \to \mathbb{B})$ is defined by recursion in Figure 2. We say that two Boolean expressions $e_1,\, e_2 \in \mathcal{B}$ are *semantically equivalent* if, and only if, $[\![e_1]\!]_{\mathcal{B}}(\sigma) = [\![e_2]\!]_{\mathcal{B}}(\sigma)$ for all states $\sigma \in$ State.

$$
\begin{aligned}
[\![\mathsf{false}]\!]_{\mathcal{B}}(\sigma) &= \bot \\
[\![\mathsf{true}]\!]_{\mathcal{B}}(\sigma) &= \top \\
[\![!e]\!]_{\mathcal{B}}(\sigma) &= \neg [\![e]\!]_{\mathcal{B}}(\sigma) \\
[\![e_1 \ \&\& \ e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{B}}(\sigma) \wedge [\![e_2]\!]_{\mathcal{B}}(\sigma) \\
[\![e_1 \parallel e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{B}}(\sigma) \vee [\![e_2]\!]_{\mathcal{B}}(\sigma) \\
[\![e_1 = e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) = [\![e_2]\!]_{\mathcal{A}}(\sigma) \\
[\![e_1 \leq e_2]\!]_{\mathcal{B}}(\sigma) &= [\![e_1]\!]_{\mathcal{A}}(\sigma) \leq [\![e_2]\!]_{\mathcal{A}}(\sigma)
\end{aligned}
$$

Figure 2: Definition of the denotational semantics of Boolean expressions.

Semantics of Statements

The small-step operational semantics relation $\rightarrow \subseteq \mathcal{C} \times \mathcal{C}$ is defined by the rules in Figure 3 where the set of configurations $\mathcal{C}$ is $(\mathcal{S} \times \text{State}) \cup \text{State}$.

$$
\frac{}{\langle \mathsf{skip},\, \sigma \rangle \rightarrow \sigma}
\qquad
\frac{}{\langle x \leftarrow e,\, \sigma \rangle \rightarrow \sigma[x \mapsto [\![e]\!]_{\mathcal{A}}(\sigma)]}
$$

$$
\frac{\langle S_1,\, \sigma_1 \rangle \rightarrow \langle S_1',\, \sigma_2 \rangle}{\langle S_1; S_2,\, \sigma_1 \rangle \rightarrow \langle S_1';\ S_2,\, \sigma_2 \rangle}
\qquad
\frac{\langle S_1,\, \sigma_1 \rangle \rightarrow \sigma_2}{\langle S_1; S_2,\, \sigma_1 \rangle \rightarrow \langle S_2,\, \sigma_2 \rangle}
$$

$$
\frac{}{\langle \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2,\, \sigma \rangle \rightarrow \langle S_1,\, \sigma \rangle}\ [\![e]\!]_{\mathcal{B}}(\sigma) = \top
$$

$$
\frac{}{\langle \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2,\, \sigma \rangle \rightarrow \langle S_2,\, \sigma \rangle}\ [\![e]\!]_{\mathcal{B}}(\sigma) = \bot
$$

$$
\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ S,\, \sigma \rangle \rightarrow \langle S;\ \mathsf{while}\ e\ \mathsf{do}\ S,\, \sigma \rangle}\ [\![e]\!]_{\mathcal{B}}(\sigma) = \top
$$

$$
\frac{}{\langle \mathsf{while}\ e\ \mathsf{do}\ S,\, \sigma \rangle \rightarrow \sigma}\ [\![e]\!]_{\mathcal{B}}(\sigma) = \bot
$$

Figure 3: Definition of the operational semantics of statements.

Hoare Triples

A Hoare triple $\{P\}\ S\ \{Q\}$ for $P, Q \subseteq \text{State}$ asserts that, for any state $\sigma \in \text{State}$, if $\sigma \in P$ and $\langle S, \sigma \rangle \rightarrow^* \sigma'$, then $\sigma' \in Q$. The sets P and Q can be represented as Boolean expressions extended with quantifiers.

The rules for constructing Hoare triples are given in Figure 4.

$$\frac{}{\{P\}\ \textsf{skip}\ \{P\}} \qquad \frac{}{\{P\}\ x \leftarrow e\ \{\exists x'.\, P[x'/x]\ \&\&\ x = e[x'/x]\}}$$

$$\frac{\{P\}\ S_1\ \{Q\} \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1; S_2\ \{R\}} \qquad \frac{\{P\ \&\&\ e\}\ S_1\ \{Q_1\} \quad \{P\ \&\&\ !e\}\ S_3\ \{Q_2\}}{\{P\}\ \textsf{if}\ e\ \textsf{then}\ S_1\ \textsf{else}\ S_2\ \{Q_1 \parallel Q_2\}}$$

$$\frac{\{P\ \&\&\ e\}\ S\ \{P\}}{\{P\}\ \textsf{while}\ e\ \textsf{do}\ S\ \{P\ \&\&\ !e\}} \qquad \frac{\{P_1\}\ S\ \{Q_1\}}{\{P_2\}\ S\ \{Q_2\}}\ \begin{array}{l} P_2 \subseteq P_1 \\ Q_1 \subseteq Q_2 \end{array}$$

Figure 4: Rules of Hoare logic.

Computable Functions

We write $[\mathtt{x} \mapsto n]$ for the state that maps the variable $\mathtt{x}$ to the number $n \in \mathbb{N}$, and every other variable to 0.

A 'while' program $\mathtt{S}$ *computes* a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ (with respect to $\mathtt{x}$) just if $f(m) \simeq n$ exactly when $\langle S, [\mathtt{x} \mapsto m] \rangle \Downarrow [\mathtt{x} \mapsto n]$.

A function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is *computable* just if there is a program S that computes f with respect to the variable $\mathtt{x}$.

Predicates

The *characteristic function* of U is the function

$$\chi_U : \mathbb{N} \to \mathbb{N}$$
$$\chi_U(n) = \begin{cases} 1 & \text{if } n \in U \\ 0 & \text{if } n \notin U \end{cases}$$

The *semi-characteristic function* of U is the partial function

$$\xi_U : \mathbb{N} \rightharpoonup \mathbb{N}$$
$$\xi_U(n) \begin{cases} \simeq 1 & \text{if } n \in U \\ \uparrow & \text{otherwise} \end{cases}$$

A predicate $U \subseteq \mathbb{N}$ is *decidable* just if its characteristic function $\chi_U : \mathbb{N} \to \mathbb{N}$ is computable.

The 'while' program that computes the characteristic function χ_U of a predicate $U \subseteq \mathbb{N}$ is called a *decision procedure*. Any predicate for which there is no decision procedure is called *undecidable*.

A predicate $U \subseteq \mathbb{N}$ is *semi-decidable* just if its semi-characteristic function ξ_U is computable.

The *Halting Problem* is the following predicate:

$$\mathsf{HALT} = \{\langle \ulcorner S \urcorner, n\rangle \mid [\![S]\!]_{\mathtt{x}}(n)\downarrow\}$$

Bijections

A function $f : A \to B$ is *injective* (or 1-1) just if for any $a_1, a_2 \in \mathcal{A}$ we have that $f(a_1) = f(a_2)$ implies $a_1 = a_2$. We sometimes write $f : A \rightarrowtail B$ whenever f is an injection.

A function $f : A \to B$ is *surjective* just if for any $b \in \mathcal{B}$ there exists $a \in \mathcal{A}$ such that $f(a) = b$. We sometimes write $f : A \twoheadrightarrow B$ whenever f is a surjection.

A function $f : A \to B$ is a *bijection* just if it is both injective and surjective.

Let $f : A \to B$ be a function. f is an *isomorphism* just if it has an *inverse*. That is, if there exists a function $f^{-1} : B \to A$ such that:

- for all $a \in \mathcal{A}$ we have $f^{-1}(f(a)) = a$
- for all $b \in \mathcal{B}$ we have $f(f^{-1}(b)) = b$

Encoding Data

A *pairing function* is a bijection $\mathbb{N} \times \mathbb{N} \xrightarrow{\cong} \mathbb{N}$. We assume that we have a fixed pairing function

$$\langle -, - \rangle : \mathbb{N} \times \mathbb{N} \xrightarrow{\cong} \mathbb{N}$$

with the following inverse:

$$\mathsf{split} : \mathbb{N} \xrightarrow{\cong} \mathbb{N} \times \mathbb{N}$$

Reflections

Suppose we have two bijections:

$$\phi : A \xrightarrow{\cong} \mathbb{N} \quad \psi : B \xrightarrow{\cong} \mathbb{N}$$

The *reflection* of $f : A \rightharpoonup B$ under (ϕ, ψ) is the function

$$\tilde{f} : \mathbb{N} \rightharpoonup \mathbb{N}$$
$$\tilde{f}(n) = \psi(f(\phi^{-1}(n)))$$

Gödel Numbering

Let **Stmt** be the set of Abstract Syntax Trees of While. We assume that we have a Gödel numbering

$$\ulcorner - \urcorner : \textbf{Stmt} \xrightarrow{\cong} \mathbb{N}$$

which encodes While programs as natural numbers.

A *code transformation* is a function f : **Stmt** $\to$ **Stmt**.

Universal Function

The *universal function*, U, is defined as follows:

$$U : \mathbf{Stmt} \times \mathbb{N} \rightharpoonup \mathbb{N}$$
$$U(P, n) = [\![P]\!]_{\mathtt{x}}(n)$$

Reductions

Let $U, W \subseteq \mathbb{N}$ be predicates, and let $f : \mathbb{N} \to \mathbb{N}$. The function f is a *many-one reduction* from U to W just if it is computable, and it is also the case that

$$n \in U \Leftrightarrow f(n) \in W$$

We may write $f : U \lesssim V$ (read "f is a reduction from U to V").

END OF PAPER