CS 320
Computer Language Processing
Midterm

April 4, 2025

Name:	Ada Lovelace	Room:	INM 200
Sciper:	111111	Seat:	314

1. The exam starts at 13:15 and ends at 15:45. You have **150 minutes** to complete the exam.
2. Place your CAMIPRO card on your desk.
3. Put all electronic devices in a bag away from the bench.
4. Write your final answers using a permanent pen (no pencils, no erasable pens).
5. This exam is 18 pages long, including this cover page. Check that you have all the pages.
6. The exercises are not ordered by how difficult they may be. If you are stuck on an exercise, you can skip it and come back to it later.
7. Answer all questions in the provided space. Do not submit additional sheets. Do not unstaple the given sheets. Material in the scratch area will not be graded.
8. Any multiple-choice questions have a **single correct answer**. Clearly circle the letter corresponding to your choice on the page itself.
9. The maximum number of points on the exam is **50**.

Question:	1	2	3	4	5	6	Total
Points:	10	5	10	10	5	10	50
Score:							

Page intentionally left blank.

Question 1. (5 points)

Consider the following language L over $\{a, b, c, d\}$:

$$\{a^l b^m c^n d^{l+m+n} \in \{a, b, c, d\}^* \mid l, m, n \geq 0\}$$

(i) $\boxed{\text{3 points}}$ Produce a context-free grammar G such that $L(G) = L$, i.e. that G generates the language L, and show a derivation of the word ab^2cd^4 in G.

(ii) $\boxed{\text{2 points}}$ Prove that L is not a regular language.

Question 2. (5 points)

Consider a simple language with variables (alphanumeric strings starting with a letter), assignments (`=`), equality (`==`), conditionals (`if then else`), single-line comments (`// ...`), and block comments (`/* ... */`).

We wish to write a lexer for this language, by defining each of the following tokens (in the given priority order):

`IF, THEN, ELSE, EQ, ASSIGN, LPAREN, RPAREN, ID, INT, LINESKIP, BLOCKSKIP, WS`

The following example strings must be tokenized as given below. `\n` is a single character, representing a newline.

String	Token Stream
`x=1`	`ID ASSIGN INT`
`if then else`	`IF WS THEN WS ELSE`
`(x == 1)`	`LPAREN ID WS EQ WS INT RPAREN`
`===`	`EQ ASSIGN`
`if // comment if`	`IF WS LINESKIP`
`// comment \n if`	`LINESKIP WS IF`
`x ==/* comment */5`	`ID WS EQ BLOCKSKIP INT`
`/* comment // other */ 5`	`BLOCKSKIP WS INT`
`/* comment /* other */ */`	*Lexing error*

The lexer must enforce that block comments cannot be nested, i.e., a block comment start `/*` must not appear inside another block comment.

The lexing priority follows longest match rule and the priority of tokens as given. You may assume Σ is the total alphabet, A is the set of English letters (a-z, A-Z), and D is the set of digits (0-9).

For each of the tokens below, fill in the blank, choosing **one** regular expression or non-deterministic finite automaton (NFA) (**next page**) that represents the words to be matched by the token. `WS` (accepting spaces, tabs, and newlines) is defined.

(i) $\boxed{\text{0.4 points}}$ `IF`: ______________

(ii) $\boxed{\text{0.4 points}}$ `THEN`: ______________

(iii) $\boxed{\text{0.4 points}}$ `ELSE`: ______________

(iv) $\boxed{\text{0.4 points}}$ `EQ`: ______________

(v) $\boxed{\text{0.4 points}}$ `ASSIGN`: ______________

(vi) $\boxed{\text{0.4 points}}$ `LPAREN`: ______________

(vii) $\boxed{\text{0.4 points}}$ `RPAREN`: ______________

(viii) $\boxed{\text{0.6 points}}$ `ID`: ______________

(ix) $\boxed{\text{0.4 points}}$ `INT`: ______________

(x) $\boxed{\text{0.6 points}}$ `LINESKIP`: ______________

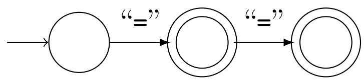
(xi) $\boxed{\text{0.6 points}}$ `BLOCKSKIP`: ______________

(xii) `WS`: (“\s” | “\t” | “\n”)$^+$

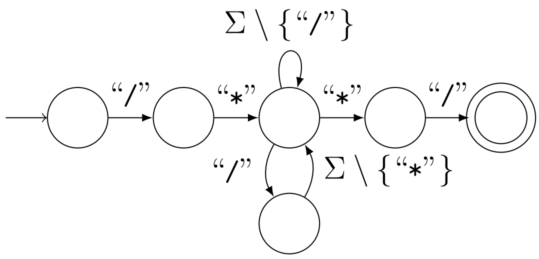
We use the following notation for regular expressions and sets, where e_1 and e_2 are regular expressions, and a, b, and c are characters in the alphabet:

1. $e_1 e_2$ is the concatenation of e_1 and e_2.
2. $e_1 \mid e_2$ is the union or disjunction of e_1 and e_2.
3. e^* represents zero or more repetitions of e. e^+ is one or more repetitions of e.
4. Characters in the alphabet are written with quotes for clarity, e.g. “a”. A string of characters, e.g. “abc” represents the concatenation of characters, i.e., “a” “b” “c”.
5. A finite set of characters represents the disjunction of its elements, e.g. $\{a, b, c\} =$ “a” | “b” | “c”.
6. The binary operations union ($\cup$), intersection ($\cap$), and set difference ($\setminus$) are interpreted as usual on sets of characters.

A : “`if`” **B** : “`then`” **C** : “`===`” **D** :

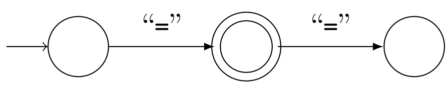


E :

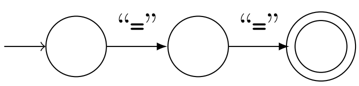


F : “`/*`” $(\Sigma \setminus \{\text{“/”}\})^*$ “`*/`”

G : “`//`” $(\Sigma \setminus \{\text{“\texttt{\textbackslash n}”}\})^*$ **H** :

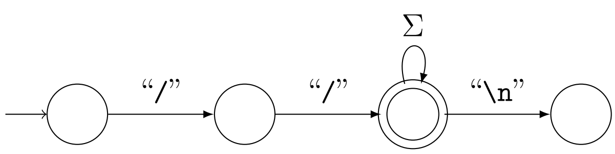


I :

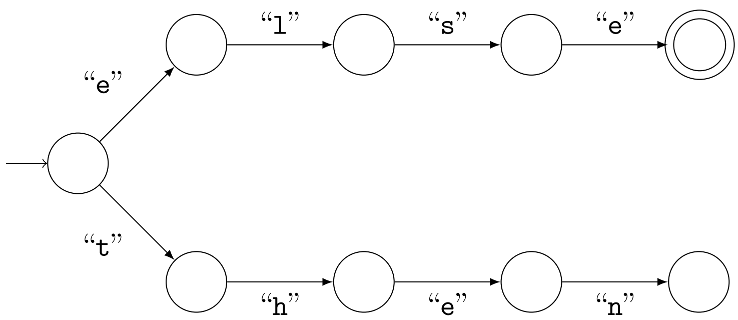


J : $(A \cup D)^*$ **K** : D^+ **L** : “`)`” **M** : “`(`”

N : $A(A \cup D)^*$ **O** :



P :



Question 3. (5 points)

Consider a context-free language L over alphabet Σ defined by some grammar G, with start symbol S. We define the language L' by the following grammar G':

$$R ::= RS \mid \epsilon$$

where R is the start symbol of G', and $L(G') = L'$.

(i) $\boxed{\text{1 point}}$ Express the grammar G' as a set of rules defining an inductive relation. You may assume that the inductive relation $S \subseteq \Sigma^*$ has been defined. Note that a set is a special case of an inductive relation, having one argument.

(ii) $\boxed{\text{4 points}}$ Use these inductive definitions to prove that $L' = L^*$. Use the fact that for any grammar G and word w, $w \in L(G) \iff w \in G$ where G is defined as an inductive relation.

Recall that $w \in L^*$ if and only if for some $n \geq 0$, there exist $w_1, \ldots, w_n$ such that $w = w_1 \ldots w_n$ and $w_i \in L$ for each $0 < i \leq n$.

Question 4. (5 points)

Consider the following grammar for a language consisting of variables, constructors, and match-case statements:

$$
\begin{aligned}
S &::= \mathit{Expr}\ \textbf{EOF} && (0)\\
\mathit{Expr} &::= \mathit{SimpleExpr}\ \mathit{Expr}' && (1)\\
\mathit{Expr}' &::= \epsilon \mid \mathit{Match} && (2)\\
\mathit{SimpleExpr} &::= \texttt{var} \mid \mathit{Cons} \mid (\mathit{Expr}) && (3)\\
\mathit{Cons} &::= \texttt{id}(\mathit{ExprList}) && (4)\\
\mathit{ExprList} &::= \epsilon \mid \mathit{NExprList} && (5)\\
\mathit{NExprList} &::= \mathit{Expr} \mid \mathit{Expr},\ \mathit{NExprList} && (6)\\
\mathit{Match} &::= \texttt{match}\ \mathit{CaseList} && (7)\\
\mathit{CaseList} &::= \mathit{Case} \mid \mathit{Case}\ \mathit{CaseList} && (8)\\
\mathit{Case} &::= \texttt{case}\ \mathit{SimpleExpr}\ \texttt{=>}\ \mathit{Expr} && (9)
\end{aligned}
$$

where `var`, `id`, `match`, `case`, `=>`, `(`, `)`, `,`, and **EOF** are all terminal tokens.

This question has four (4) subparts, one on each of the following pages.

(i) $\boxed{\text{1 point}}$ Compute nullable for each non-terminal in the grammar.

$$\text{nullable}(S) =$$

$$\text{nullable}(Expr) =$$

$$\text{nullable}(Expr') =$$

$$\text{nullable}(SimpleExpr) =$$

$$\text{nullable}(Cons) =$$

$$\text{nullable}(ExprList) =$$

$$\text{nullable}(NExprList) =$$

$$\text{nullable}(Match) =$$

$$\text{nullable}(CaseList) =$$

$$\text{nullable}(Case) =$$

(ii) $\boxed{\text{1 point}}$ Compute the first(·) sets for each non-terminal in the grammar.

$$\text{first}(S) =$$

$$\text{first}(Expr) =$$

$$\text{first}(Expr') =$$

$$\text{first}(SimpleExpr) =$$

$$\text{first}(Cons) =$$

$$\text{first}(ExprList) =$$

$$\text{first}(NExprList) =$$

$$\text{first}(Match) =$$

$$\text{first}(CaseList) =$$

$$\text{first}(Case) =$$

(iii) $\boxed{\text{1 point}}$ Compute the follow($\cdot$) sets for each non-terminal except S in the grammar.

$$\text{follow}(Expr) =$$

$$\text{follow}(Expr') =$$

$$\text{follow}(SimpleExpr) =$$

$$\text{follow}(Cons) =$$

$$\text{follow}(ExprList) =$$

$$\text{follow}(NExprList) =$$

$$\text{follow}(Match) =$$

$$\text{follow}(CaseList) =$$

$$\text{follow}(Case) =$$

(iv) $\boxed{\text{2 points}}$ Construct the parsing table for this grammar. Use the production options in order as given to fill in the parse table. For example, with $Expr' ::= \epsilon \mid Match$, write 1 for choosing the rule $Expr' ::= \epsilon$, and 2 for choosing $Expr' ::= Match$. Show that the grammar is *not* LL(1) by marking every conflict in the table.

Non-terminal	`var`	`id`	`(`	`)`	`match`	`case`	`=>`	`,`	**EOF**
Expr									
Expr'									
SimpleExpr									
Cons									
ExprList									
NExprList									
Match									
CaseList									
Case									

Question 5. (5 points)

For the grammar defined in Question 4, produce an equivalent LL(1) grammar. You do not need to show the parsing table or otherwise prove that the new grammar is LL(1).

Question 6. (5 points)

Consider the following type system for a programming language. The language contains integers, Booleans, functions, and pairs.

Pairs $(\cdot,\cdot)$ and functions $\cdot \Rightarrow \cdot$ are distinct binary type constructors.

$$\frac{n \text{ is an integer value}}{\Gamma \vdash n : \texttt{Int}}\,\text{Int}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}}\,\text{Add}$$

$$\frac{b \text{ is a Boolean value}}{\Gamma \vdash b : \texttt{Bool}}\,\text{Bool}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\,\text{If}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 == e_2 : \texttt{Bool}}\,\text{Eq} \qquad \frac{\Gamma \vdash e_1 : \texttt{Int} \qquad \Gamma \vdash e_2 : \texttt{Int}}{\Gamma \vdash e_1 < e_2 : \texttt{Bool}}\,\text{Lt}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (x : \tau_1) \Rightarrow e : \tau_1 \Rightarrow \tau_2}\,\text{Fun} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \Rightarrow \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\,\text{App}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)}\,\text{Pair}$$

$$\frac{\Gamma \vdash e : (\tau_1, \tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1}\,\text{Fst} \qquad \frac{\Gamma \vdash e : (\tau_1, \tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2}\,\text{Snd}$$

(i) $\boxed{\text{2 points}}$ Consider the term t below, with type variables $\tau_1, \tau_2, \ldots, \tau_5$ ascribing subterms of t as shown:

$$((\mathtt{x} : \tau_1) \Rightarrow (\mathtt{y} : \tau_2) \Rightarrow (\mathtt{if}\ (\mathtt{fst}(x) : \tau_3)\ \mathtt{then}\ \mathtt{snd}(x)\ \mathtt{else}\ 1 + \mathtt{snd}(x)) : \tau_4) : \tau_5$$

Which of the following statements are true about assignments to the type variables such that t is well-typed?

i. In every valid assignment, $\tau_1 = (\mathtt{Int}, \mathtt{Bool})$:
A. True B. False

ii. In every valid assignment, $\tau_3 = \mathtt{Bool}$:
A. True B. False

iii. There is a valid assignment where $\tau_2 = \mathtt{Int}$:
A. True B. False

iv. In every valid assignment, $\tau_5 = (\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_4)$:
A. True B. False

(ii) $\boxed{\text{1 point}}$ Which of the following types τ given below apply to the term t above, i.e. there is a derivation of $\vdash t : \tau$?

A. $((\mathtt{Bool}, \mathtt{Int}) \Rightarrow \mathtt{Bool}) \Rightarrow \mathtt{Int}$

B. $(\mathtt{Bool}, \mathtt{Int}) \Rightarrow \mathtt{Int} \Rightarrow \mathtt{Int}$

C. $(\mathtt{Int}, \mathtt{Bool}) \Rightarrow (\mathtt{Bool}, \mathtt{Int}) \Rightarrow \mathtt{Int}$

D. $(\mathtt{Int}, \mathtt{Bool}) \Rightarrow \mathtt{Bool} \Rightarrow \mathtt{Int}$

(iii) $\boxed{\text{2 points}}$ Consider the following term r:

$$\mathtt{x} \Rightarrow \mathtt{fst}(x)(\mathtt{snd}(x)) + \mathtt{snd}(x)$$

where we assign type variables to the subterms as follows:

$$\mathtt{x} : \tau_1 \quad \mathtt{fst}(\mathtt{x}) : \tau_2 \quad \mathtt{snd}(\mathtt{x}) : \tau_3$$
$$\mathtt{fst}(\mathtt{x})(\mathtt{snd}(\mathtt{x})) : \tau_4$$
$$\mathtt{fst}(x)(\mathtt{snd}(\mathtt{x})) + \mathtt{snd}(\mathtt{x}) : \tau_5$$
$$\mathtt{x} \Rightarrow \mathtt{fst}(x)(\mathtt{snd}(\mathtt{x})) + \mathtt{snd}(\mathtt{x}) : \tau_6$$

The initial unification constraints for type checking r are:

$$\tau_6 = \tau_1 \Rightarrow \tau_5$$
$$\tau_5 = \mathtt{Int}$$
$$\tau_4 = \mathtt{Int}$$
$$\tau_3 = \mathtt{Int}$$
$$\tau_2 = \tau_3 \Rightarrow \tau_4$$
$$\tau_1 = (\tau_3', \tau_3)$$
$$\tau_1 = (\tau_2, \tau_2')$$

for fresh type variables τ_2' and τ_3'. Note that $=$ has lower precedence than the type constructors $(\Rightarrow, (\cdot, \cdot))$, so $\tau_6 = \tau_1 \Rightarrow \tau_5$ is parsed as "τ_6 equals $\tau_1 \to \tau_5$".

Consider the following possible set of constraints at different unification steps (**this page and next**). The current set of unsolved constraints is listed *below the bar*. Whenever we substitute a type variable, we add the mapping to the list *above the bar*.

$$\textbf{Init}: \begin{array}{r} \emptyset \\ \hline \tau_6 = \tau_1 \Rightarrow \tau_5 \\ \tau_5 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_2 = \tau_3 \Rightarrow \tau_4 \\ \tau_1 = (\tau_3', \tau_3) \\ \tau_1 = (\tau_2, \tau_2') \end{array}$$

$$\textbf{1}: \begin{array}{r} \tau_1 = (\texttt{Int}, \texttt{Int}) \\ \tau_2 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \tau_6 = (\texttt{Int}, \texttt{Int}) \Rightarrow \texttt{Int} \\ \tau_2' = \texttt{Int} \\ \tau_3' = \texttt{Int} \Rightarrow \texttt{Int} \\ \hline \emptyset \end{array}$$

$$\textbf{2}: \begin{array}{r} \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_2 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \hline \tau_6 = (\tau_3', \texttt{Int}) \Rightarrow \texttt{Int} \\ (\tau_3', \texttt{Int}) = (\texttt{Int}, \tau_2') \end{array}$$

$$\textbf{3}: \begin{array}{r} \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \hline \tau_6 = \tau_1 \Rightarrow \texttt{Int} \\ \tau_2 = \texttt{Int} \\ \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_1 = (\tau_2, \tau_2') \end{array}$$

$$\textbf{4}: \begin{array}{r} \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \hline \tau_6 = (\tau_3', \texttt{Int}) \Rightarrow \texttt{Int} \\ \tau_3' = \texttt{Int} \Rightarrow \texttt{Int} \\ \texttt{Int} = \tau_2' \end{array}$$

$$\textbf{5}: \begin{array}{r} \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \hline \tau_6 = (\tau_3', \texttt{Int}) \Rightarrow \texttt{Int} \\ (\tau_3', \texttt{Int}) = (\texttt{Int} \Rightarrow \texttt{Int}, \tau_2') \end{array}$$

$$\textbf{6}: \begin{array}{r} \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \hline \tau_6 = \tau_1 \Rightarrow \texttt{Int} \\ \tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_1 = (\tau_2, \tau_2') \end{array}$$

7 :
$$\frac{\begin{array}{r}\tau_1 = (\texttt{Int} \Rightarrow \texttt{Int}, \texttt{Int}) \\ \tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \tau_6 = (\texttt{Int} \Rightarrow \texttt{Int}, \texttt{Int}) \Rightarrow \texttt{Int} \\ \tau_2' = \texttt{Int} \\ \tau_3' = \texttt{Int} \Rightarrow \texttt{Int}\end{array}}{\emptyset}$$

8 :
$$\frac{\begin{array}{r}\tau_1 = (\texttt{Int} \Rightarrow \texttt{Int}, \texttt{Int}) \\ \tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \tau_2' = \texttt{Int} \\ \tau_3' = \texttt{Int} \Rightarrow \texttt{Int}\end{array}}{\tau_6 = (\texttt{Int} \Rightarrow \texttt{Int}, \texttt{Int}) \Rightarrow \texttt{Int}}$$

9 :
$$\frac{\begin{array}{r}\tau_1 = (\tau_3', \texttt{Int}) \\ \tau_2 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int}\end{array}}{\begin{array}{r}\tau_6 = (\tau_3', \texttt{Int}) \Rightarrow \texttt{Int} \\ \tau_3' = \texttt{Int} \\ \texttt{Int} = \tau_2'\end{array}}$$

10 :
$$\frac{\begin{array}{r}\tau_2 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int}\end{array}}{\begin{array}{r}\tau_6 = \tau_1 \Rightarrow \texttt{Int} \\ \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_1 = (\texttt{Int}, \tau_2')\end{array}}$$

11 :
$$\frac{\begin{array}{r}\tau_1 = (\texttt{Int}, \texttt{Int}) \\ \tau_2 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int} \\ \tau_2' = \texttt{Int} \\ \tau_3' = \texttt{Int}\end{array}}{\tau_6 = (\texttt{Int}, \texttt{Int}) \Rightarrow \texttt{Int}}$$

12 :
$$\frac{\begin{array}{r}\tau_1 = (\tau_3', \texttt{Int}) \\ \tau_2 = \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int}\end{array}}{\begin{array}{r}\tau_6 = (\tau_3', \texttt{Int}) \Rightarrow \texttt{Int} \\ \texttt{Int} = \texttt{Int} \\ \tau_3' = \tau_2'\end{array}}$$

13 :
$$\frac{\begin{array}{r}\tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int}\end{array}}{\begin{array}{r}\tau_6 = \tau_1 \Rightarrow \texttt{Int} \\ \tau_1 = (\tau_3', \texttt{Int}) \\ \tau_1 = (\texttt{Int} \Rightarrow \texttt{Int}, \tau_2')\end{array}}$$

14 :
$$\frac{\begin{array}{r}\tau_1 = (\tau_3', \texttt{Int}) \\ \tau_2 = \texttt{Int} \Rightarrow \texttt{Int} \\ \tau_3 = \texttt{Int} \\ \tau_4 = \texttt{Int} \\ \tau_5 = \texttt{Int}\end{array}}{\begin{array}{r}\tau_6 = (\tau_3', \texttt{Int}) \Rightarrow \texttt{Int} \\ \texttt{Int} = \texttt{Int} \\ \tau_3' = \tau_2'\end{array}}$$

Circle an order of unification steps that leads to a correct and complete type inference for r, i.e. ending with assignment of all type variables.

A. **Init, 6, 13, 5, 14, 11, 1**
B. **Init, 3, 13, 5, 4, 8, 7**
C. **Init, 6, 10, 12, 11, 1**
D. **Init, 6, 10, 9, 4, 8, 7**
E. **Init, 3, 10, 12, 11, 1**
F. **Init, 3, 10, 5, 4, 8, 1**
G. **Init, 6, 13, 5, 4, 8, 7**
H. **Init, 3, 10, 9, 11, 7**

Scratch area

End