Quiz

CS-320, Computer Language Processing, Fall 2017

Wednesday, November 22nd, 2017

General notes about this quiz

- Have your CAMIPRO card ready on the desk.
- You are allowed to use the printed material from the course page, according to email instructions from the course instructor that you received earlier.
- You are not allowed to exchange notes or anything else with other students while taking the quiz.
- No electronic devices are allowed.
- No bathroom breaks during quiz are allowed.
- Try to write your answers in the space provided in the question paper. If you need more space, write the entire answer to your question on that separate sheet (do not mix answers from multiple questions on same sheets, so we can separate them per question later). We will provide you with the paper.
- Make sure you write your name on each sheet of paper that you hand in.
- Use a permanent pen with dark ink for all answers.
- It is advisable to do the questions you know best first.
- You have in total **3 hours 45 minutes**.

Your name: ______________________ Your SCIPER: ____________

Exercise	**Points**	
1	15	
2	20	
3	20	
4	20	
5	25	
Total	100	

Problem 1: Formal Languages (15 points)

Let $\Sigma = \{a, b\}$ for a, b distinct, and $L, L_1, L_2 \subseteq \Sigma^*$. Remember that for languages, concatenation is defined by

$$L_1L_2 = \{u_1u_2 \mid u_1 \in L_1, u_2 \in L_2\}$$

a) **[5 pts]** Find all finite languages L for which the equation $L = LL$ holds. Prove that there are no finite languages other than those you identified that satisfy that equation.

The only solutions are $L = \emptyset$ and $L = \{\epsilon\}$. We can prove that any language $L = LL$ containing a non-empty word will be infinite.

Towards a contradiction, assume there is a finite language $L = LL$ such that $L \neq \emptyset$ and $L \neq \{\epsilon\}$. Then the longest words in L will be of length $n > 0$. Let w be a word in L such that $|w| = n$. From the the definition of language concatenation we deduce that $ww \in LL$, and because of $L \supseteq LL$ we must have $ww \in L$. But $|w| = n < 2n = |ww|$, so w cannot be among the longest words in L. Thus, there can be no finite language $L = LL$ other than $\emptyset$ and $\{\epsilon\}$.

b) **[10 pts]** Explain how many solutions there are for the language equation

$$(\{b\}L\{a\}L) \cup \{\epsilon\} = L$$

If there are solutions, describe one of them.

There is a single such language and it is isomorphic to the language of balanced parentheses. In other words, the unique solution can be described by the context-free grammar $L ::= b\,L\,a\,L \mid \epsilon$.

First note that any solution L will include (at least) all of the expressions of "balanced bs-and-as" — we will call the latter language L_b.

One can prove that $L \supseteq L_b$ by induction on the length of words $w \in L_b$. In the base case $|w| = 0$, so $w = \epsilon$. And indeed, any L satisfying $(\{b\}L\{a\}L) \cup \{\epsilon\} = L$ will clearly contain ϵ. In the inductive case we have $|w| = n$ for $n > 0$. Recall that in the lecture you saw a definition of balanced parentheses and a lemma stating that for any non-empty word w' in that language we will be able to decompose it into $w' = (u')v'$ where u' and v' must be shorter words. Analogously, for every w with $|w| > 0$ we have $w = buav$ for some shorter words u and v. From the induction hypothesis (IH) we deduce that u and v must be in L. Now, using the definition of language concatenation we know that $buav \in \{b\}L\{a\}L \subseteq L$. Therefore any solution L to the equation will contain $w = buav$, concluding our proof.

One can also show an inclusion in the other direction, i.e., that any solution L to the equation will only include expressions from L_b. We perform a proof by induction on the length of words w in a solution L. In the base case we again have $|w| = 0$, hence $w = \epsilon$, which is contained both in L and L_b. In the inductive case we again have $|w| = n$ for $n > 0$. Note that $w \in \{b\}L\{a\}L$, since $w \in L = (\{b\}L\{a\}L) \cup \{\epsilon\}$, but $w \neq \epsilon$. Using the definition of language concatenation we find that $w = buav$ for some words $u, v \in L$. Since u and v must be shorter than w we can invoke the IH and get $u, v \in L_b$. But if u and v are "b-and-a-balanced" (for all prefixes $\#b \geq \#a$, for the entire word $\#b = \#a$), then so are bua and $buav = w$, hence $w \in L_b$.

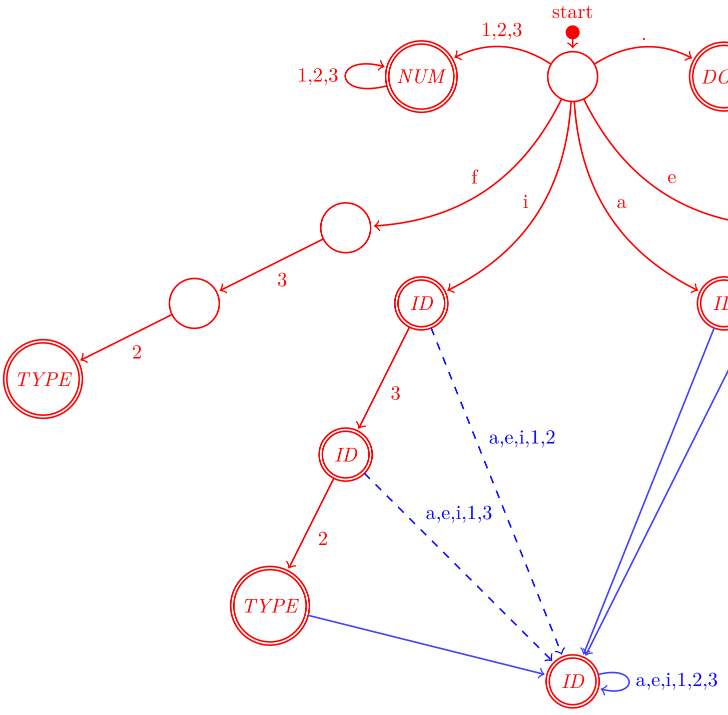
Problem 2: Lexical Analysis (20 points)

This question is motivated by the *target* language of the Amy compiler, the WebAssembly, but is self-contained. Consider the following token definitions, with * denoting regular expression Kleene star and the input alphabet being $\{1, 2, 3, a, e, i, f, d, q, n, .\}$.

$$\begin{array}{rcl}
\texttt{TYPE} & ::= & \mathit{i32} \mid \mathit{f32} \\
\texttt{DOT} & ::= & . \\
\texttt{OP} & ::= & \mathit{eq} \mid \mathit{add} \\
\texttt{END} & ::= & \mathit{end} \\
\texttt{ID} & ::= & (a \mid e \mid i)\ (a \mid e \mid i \mid 1 \mid 2 \mid 3)^* \\
\texttt{NUM} & ::= & (1 \mid 2 \mid 3)\ (1 \mid 2 \mid 3)^*
\end{array}$$

The table above defines a number of tokens using regular expressions. The tokens are listed in descending order of priority, e.g., `END` has priority over `ID`. Assume that there are no whitespaces or comments.

a) [15 pts] Construct the combined finite state machine that we would use to tokenize input strings. Make sure to mark accepting states with the token they produce. Do *not* draw the trap state and edges going into it. *(We are looking for a deterministic automaton.)*



b) [5 pts] Show the result of tokenization using the longest match rule for the following input string (indicate the boundaries of any recognized tokens in the string and the names of tokens recognized).

`i32.add1eaeqend`

The lexer will tokenize `i32` as `TYPE`, `.` as `DOT`, `add` as `OP`, `1` as `NUM`, `eae` as `ID` and then get stuck, since `q` is not a valid first character of any token. (Some lexers may also output `end` as `END`.)

Problem 3: Parsing and Grammars (20 points)

In this exercise you will construct a simplified context-free grammar for WebAssembly bytecode, which could be a starting point for building a tool that accepts WebAssembly as its *input.* Assume we are now working with the following set of tokens (not quite the same as in the previous question):

$$\begin{array}{rcl}
\texttt{TYPE} & ::= & \textit{i32} \mid \textit{i64} \\
\texttt{DOT} & ::= & . \\
\texttt{OP} & ::= & \textit{eq} \mid \textit{add} \\
\texttt{CONST} & ::= & \textit{const} \\
\texttt{BLOCK} & ::= & \textit{block} \\
\texttt{END} & ::= & \textit{end} \\
\texttt{BRIF} & ::= & \textit{br\_if} \\
\texttt{ID} & ::= & (a \mid e \mid i)\ (a \mid e \mid i \mid 1 \mid 2 \mid 3)^* \\
\texttt{NUM} & ::= & (1 \mid 2 \mid 3)\ (1 \mid 2 \mid 3)^*
\end{array}$$

a) [**20 pts**] Find an LL(1) grammar that corresponds to the language of WebAssembly bytecode expressible using the above tokens.

- Your grammar should include instructions for addition, comparing for equivalence and constant loading.
- These operations may only occur prefixed by a type and a dot, e.g., `i32.add`.
- Constant loads must be followed by the integer value being loaded, e.g., `i32.const 1`.
- Your grammar should include arbitrarily nested blocks. Blocks begin with the block keyword followed by an identifier, e.g., `block eee`, contain any number of instructions, and end with `end`.
- Your grammar should include the branching instruction `BRIF`. It must be followed by an identifier, e.g., `br_if eee`. **Branches may only appear *inside* of blocks.** (But do not try to enforce that the target of the branch is the same as the label of the block.) Non-branch instructions can appear both outside and inside blocks.
- Finally, your grammar should feature *program* as its starting non-terminal. A program contains any number of instructions.

Here is an example of a program your grammar should accept:

```
i32.const 1
block e1
  block e2
    i32.const 2
    i32.eq
    br_if e1
    i64.const 123
    i32.const 1
    br_if e2
  end
  i64.const 321
end
```

$\mathit{program} ::=$

$$\begin{array}{rcl}
\mathit{program} & ::= & \mathit{outerInstrs} \\
\mathit{outerInstrs} & ::= & \mathit{outerInstr}\ \mathit{outerInstrs} \mid \epsilon \\
\mathit{outerInstr} & ::= & \mathit{prefixOp} \mid \mathit{block} \\
\mathit{block} & ::= & \texttt{BLOCK}\ \texttt{ID}\ \mathit{innerInstrs}\ \texttt{END} \\
\mathit{innerInstrs} & ::= & \mathit{innerInstr}\ \mathit{innerInstrs} \mid \epsilon \\
\mathit{innerInstr} & ::= & \mathit{outerInstr} \mid \texttt{BRIF}\ \texttt{ID} \\
\mathit{prefixOp} & ::= & \texttt{TYPE}\ \texttt{DOT}\ \mathit{prefixOpRest} \\
\mathit{prefixOpRest} & ::= & \texttt{OP} \mid \texttt{CONST}\ \texttt{NUM}
\end{array}$$

Problem 4: Types and Implicit Conversions (20 points)

Please read all parts of this question before solving any of them. Consider a language with two distinct and disjoint types *Int* and *Bool*, and operations that correspond to the following function signatures:

$$\begin{array}{rcl} + & : & \mathit{Int} \times \mathit{Int} \to \mathit{Int} \\ < & : & \mathit{Int} \times \mathit{Int} \to \mathit{Bool} \\ \mathit{bool2int} & : & \mathit{Bool} \to \mathit{Int} \end{array}$$

Expressions have the syntax

$$e ::= e + e \mid e < e \mid \mathit{bool2int}(e) \mid x$$

where x denotes variables.
The environment Γ maps variable names to types from the set $\{\mathit{Int}, \mathit{Bool}\}$.
We define the *explicit system* for expressions using the following type rules:

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e : \mathit{Bool}}{\Gamma \vdash \mathit{bool2int}(e) : \mathit{Int}}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Int} \quad \Gamma \vdash e_2 : \mathit{Int}}{\Gamma \vdash e_1 + e_2 : \mathit{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathit{Int} \quad \Gamma \vdash e_2 : \mathit{Int}}{\Gamma \vdash e_1 < e_2 : \mathit{Bool}}$$

a) **[2 pts]** If $\{(k, \mathit{Int}), (b, \mathit{Bool})\} \subseteq \Gamma$, does the expression $b < k + k$ type check in the explicit system?

If yes, give a type derivation. If no, prove that no type derivation is possible.

b) **[8 pts]** Give rules for introducing implicit conversions.

Design a new inductively defined relation that is analogous to the explicit type system but also holds if there is a way to insert calls to the function *bool2int* in the expression. The relation also computes the resulting expression with implicit conversions inserted. Please use the following notation

$$\Gamma \vdash^{(i)} e : (\tau; e')$$

to mean that, in the environment Γ, it is possible to introduce zero or more implicit conversions inside expression e so that the result is expression e' that type-checks according to the explicit system and has type τ. Your rules should be as permissive as possible.
(Note: We are looking for a new set of typing rules using $\vdash^{(i)}$ exclusively rather than $\vdash$.)

c) **[10 pts]** Prove the following theorem showing conversion insertion behaves as expected:

For all expressions e, e', an environment Γ, and type τ, if $\Gamma \vdash^{(i)} e : (\tau; e')$ then both of the following conditions hold:

1. $\Gamma \vdash e' : \tau$ (resulting expression type checks in the explicit system)
2. if e contains no occurrences of *bool2int* then $|e'| = e$ (only coercions were introduced)

Here $|\_|$ is a function that maps expressions to expressions, defined as follows:

$$\begin{array}{rcl} |\mathit{bool2int}(e)| & = & |e| \\ |x| & = & x, \text{ if } x \text{ is a variable} \\ |e_1 + e_2| & = & |e_1| + |e_2| \\ |e_1 < e_2| & = & |e_1| < |e_2| \end{array}$$

Ad a). No, the expression does not type-check under the explicit system's rules. Any typing derivation for $b < k + k$ in the explicit system would have to use the rule for the less-than comparison at its end. From the less-than rule's definition we can see that for this rule to be applied there must be valid typing derivations that end in $b : \mathit{Int}$ and $k + k : \mathit{Int}$. Note that b is a variable and thus there is only one applicable rule in the explicit system. But the environment Γ maps b to type Bool, so there can be no such typing derivation for the less-than comparison.

Ad b).

$$\textsc{T-Var}\quad \frac{(x, \tau) \in \Gamma}{\Gamma \vdash^{(i)} x : (\tau; x)} \qquad \textsc{T-BtoI}\quad \frac{\Gamma \vdash^{(i)} e_b : (\mathit{Bool}; e'_b)}{\Gamma \vdash^{(i)} \mathit{bool2int}(e_b) : (\mathit{Int}; \mathit{bool2int}(e'_b))} \qquad \textbf{T-Coerce}\ \textit{(new)}\quad \frac{\Gamma \vdash^{(i)} e_b : (\mathit{Bool}; e'_b)}{\Gamma \vdash^{(i)} e_b : (\mathit{Int}; \mathit{bool2int}(e'_b))}$$

$$\textsc{T-Add}\quad \frac{\Gamma \vdash^{(i)} e_1 : (\mathit{Int}; e'_1) \quad \Gamma \vdash^{(i)} e_2 : (\mathit{Int}; e'_2)}{\Gamma \vdash^{(i)} e_1 + e_2 : (\mathit{Int}; e'_1 + e'_2)} \qquad \textsc{T-Less}\quad \frac{\Gamma \vdash^{(i)} e_1 : (\mathit{Int}; e'_1) \quad \Gamma \vdash^{(i)} e_2 : (\mathit{Int}; e'_2)}{\Gamma \vdash^{(i)} e_1 < e_2 : (\mathit{Bool}; e'_1 < e'_2)}$$

Ad c). We prove 1.) by induction on the structure of typing derivations in the implicit system.
We begin with $T - Var$, the rule for variables, which coerces $e = x$ to $e' = x$, and therefore is essentially a copy of the explicit version. We need to show that from $\Gamma \vdash^{(i)} x : (\tau; x)$ it follows that $\Gamma \vdash x : \tau$. Observe that given the premise of the implicit system's rule, $(x, \tau) \in \Gamma$, there is also a typing derivation for $\Gamma \vdash x : \tau$ in the explicit system: we can simply use its version of the variable rule once.
We next prove the property for $T - \mathit{BtoI}$, which depends on $\Gamma \vdash e_b : (\mathit{Bool}; e'_b)$ as its sole premise. Note that the rule simply "coerces" $\mathit{bool2int}(e_b)$ to $\mathit{bool2int}(e'_b)$. By the induction hypothesis (IH), we know that $\Gamma \vdash e'_b : \mathit{Bool}$, so there exists a typing derivation for e'_b in the explicit system. But then there is also a typing derivation for $\mathit{bool2int}(e'_b)$ in the explicit system: it ends in the explicit version of $T - \mathit{BtoI}$ and uses the typing derivation of e'_b in its premise.
Note that the same kind of argument holds for the other two rules "copied" from the explicit system, i.e., $T - \mathit{Add}$ and $T - \mathit{Less}$. Each corresponding rule in the implicit system in fact simply "coerces" its expression $e_1 \cdot e_2$ to $e'_1 \cdot e'_2$, and because of the IH we know that $e'_1 : \mathit{Int}$ and $e'_2 : \mathit{Int}$, so there is a typing derivation in the explicit system for $e'_1 \cdot e'_2 : \tau$ that ends in the corresponding rule of the explicit system and uses typing derivations of e'_1 and e'_2 for its premises.
In the case of rule $T - \mathit{Coerce}$ we note that its first premise requires that $\Gamma \vdash^{(i)} e_b : (\mathit{Bool}; e'_b)$, which (by the IH) implies that $\Gamma \vdash e'_b : \mathit{Bool}$. We conclude that indeed $\Gamma \vdash \mathit{bool2int}(e'_b) : \mathit{Int}$, since there is a typing derivation that ends in the explicit version of $T - \mathit{BtoI}$ and uses the typing derivation of e'_b for its premise.

We prove 2.) again by induction on the structure of the implicit system's typing derivations. For all expressions e, e' where e does not contain any occurrence of $\mathit{bool2int}$ and for which $\Gamma \vdash^{(i)} e : (\tau; e')$ holds (for some environment Γ and some type τ) we have to show that $|e'| = e$.
Note that any corresponding typing derivation will not use rule $T - \mathit{BtoI}$, since otherwise e would contain $\mathit{bool2int}(e_b)$ for some expression e_b.
Again, we first consider the base case $T - Var$. Since $e = x$ is coerced to $e' = x$, by the definition of the erasure function $|\_\_|$ we get $|e'| = |x| \overset{\text{def}}{=} x = e$.
Next we consider typing derviations ending in an application of rules $T - \mathit{Add}$ or $T - \mathit{Less}$. Let $e = e_1 \cdot e_2$ and $e' = e'_1 \cdot e'_2$. We know from the rule's premises that $\Gamma \vdash^{(i)} e_1 : (\mathit{Int}; e'_1)$ and $\Gamma \vdash^{(i)} e_2 : (\mathit{Int}; e'_2)$. Note that if e doesn't contain $\mathit{bool2int}$, then e_1 and e_2 won't either. By the IH we then have $|e'_1| = e_1$ and $|e'_2| = e_2$. Using this and the definition of the erasure function we can rewrite $|e'| = |e'_1 \cdot e'_2| \overset{\text{def}}{=} |e'_1| \cdot |e'_2| = e_1 \cdot e_2 = e$.
Finally, we prove the property for typing derivations ending in $T - \mathit{Coerce}$. Let $e = e_b$ and $e' = \mathit{bool2int}(e'_b)$. Again, from the premise and by the IH, we have $\Gamma \vdash^{(i)} e_b : (\mathit{Bool}; e'_b)$ and thus $|e'_b| = e_b$. So $|e'| = |\mathit{bool2int}(e'_b)| \overset{\text{def}}{=} |e'_b| = e_b = e$.

Problem 5: Optimized Code Generation (25 points)

In this last exercise you will generate WebAssembly for a simple, effectful language of arithmetic expressions, while optimizing for stack usage. The language has the following abstract syntax tree definition:

$$\mathit{expr} \ ::= c \ | \ x \ | \ \mathit{expr} \ + \ \mathit{expr} \ | \ \mathit{expr} \ / \ \mathit{expr}$$

where c represents integer constants and x represents variables. (In our examples we also use parentheses, but this is simply to denote the desired abstract syntax tree.) For example, the language allows the arithmetic expression

$$1 + (a/3)$$

where the variable a is in some global environment.

The evaluation semantics of this language are as usual for 32-bit signed modular arithmetic, with the exception that division by zero produces a side-effect reporting the illegal division. For instance, executing the code generated for

$$(1 + 2)/3/0$$

will print an error message "`Illegal division: 1 / 0`".

Finally, effectful expressions, i.e., those expressions potentially reporting a division by zero, should always behave as if evaluated left-to-right. For instance,

$$(1/0) + (2/0)$$

will first report "`Illegal division: 1 / 0`" and then "`Illegal division: 2 / 0`".

Our goal in this exercise is to minimize the amount of stack memory used during the execution of programs. To this end we will reorder the evaluation of some subexpressions. To see how one can reduce the stack size in this way, consider the evaluation of the following expression in two different ways:

$$1 + (a/3)$$

Example a) (everything left-to-right): Evaluate 1, a, 3, $a/3$, $1 + (a/3)$.
⇒ maximum stack size: 3

Example b) (optimized): Evaluate a, 3, $a/3$, 1, $1 + (a/3)$.
⇒ maximum stack size: 2

We will define a translation function $[e]$ which returns a triple $(\mathit{bc}, \mathit{pure}, \mathit{usage})$ where

- *bc* is the WebAssembly bytecode corresponding to e,
- *pure* is a boolean that conservatively approximates whether the computation of expression e will be side-effect-free (i.e. cannot encounter a division by zero), and
- *usage* is an integer indicating the amount of stack space required during the evaluation of expression e.

Below we give you the definition of the translation function for variables:

$[x] = ($ *get_local* #x , *true*, 1) *where #x denotes the slot for variable x*

Your task is to complete the translation function for the remaining kinds of expressions. Your implementation should perform the above-mentioned optimization, but maintain the order in which any potential errors will be reported.

For the tasks below you may only rely on the commutativity of + (the law $x + y = y + x$), but not its associativity or any other laws. That is, the two operands of a + operation may be swapped, but not simplified or rearranged in other ways.

a) **[2 pts]** Define a translation function for a constant literal c.

$[c] =$

```
( i32.const c , true, 1)
```

b) **[8 pts]** Define a translation function for the division expression operator.

$[e_1/e_2] =$

```
{
  val (bc1, pure1, usage1) = [e1]
  val (bc2, pure2, usage2) = [e2]
  ( bc1; bc2; i32.div_s , false, max(usage1, 1 + usage2))
}
```

c) **[15 pts]** Define a translation function for the plus expression operator.

$[e1 + e2] =$

```
{
  val (bc1, pure1, usage1) = [e1]
  val (bc2, pure2, usage2) = [e2]
  if ((pure1 || pure2) && usage1 < usage2)
     ( bc2; bc1; i32.add , pure1 && pure2, usage2)
  else
     ( bc1; bc2; i32.add , pure1 && pure2,
       max(usage1, 1 + usage2))

}
```

Selected WebAssembly byte code instructions

get_local #x	Loads the value of the local variable x onto the stack.
set_local #x	Pops and stores the current value on top of the stack in the local variable x.
i32.const c	Loads the constant value c on the stack.
i32.{add, sub, mul, div_s }	Pops two values from the stack, performs addition/subtraction/multiplication/signed-division on them and loads the result on the stack.
i32.{eq, ne, lt_s }	Pops two values from the stack, compares them for equality/inequality/signed-less-than and loads the result on the stack.
br labl	Unconditionally jumps to block labl.
br_if labl	Pops a value from the stack and jumps to block labl if the popped value is non-zero.
block labl	Starts a block named labl and must be terminated by an end instruction. Jump instructions inside the block may jump to its end.
end	Terminates a block.