CS 4610 — Midterm 2 KEY

- **Write your name and UVa ID on the exam.** Pledge the exam before turning it in.
- There are 12 pages in this exam (including this one) and 7 questions, each with multiple parts. Some questions span multiple pages. All questions have some easy parts and some hard parts. If you get stuck on a question move on and come back to it later.
- You have 1 hour and 20 minutes to work on the exam.
- The exam is closed book, but you may refer to your two pages of notes.
- Even vaguely looking at a cellphone or similar device (e.g., tablet computer) during this exam **is cheating**.
- Please write your answers in the space provided on the exam, and clearly mark your solutions. You may use the backs of the exam pages as scratch paper. Please do not use any additional scratch paper.
- Solutions will be graded on correctness and clarity. Each problem has a relatively simple and straightforward solution. We might deduct points if your solution is far more complicated than necessary. Partial solutions will be graded for partial credit.
  - *Good Writing Example:* Python and Ruby have implemented some Smalltalk-inspired ideas with a more C-like syntax.
  - *Bad Writing Example:* Im in ur class, @cing ur t3stz!1!
- If you leave a non-extra-credit portion of the exam blank, **you will receive one-third of the points for that portion (rounded down) for not wasting our time.** If you randomly guess and throw likely words at us, we will be much less sanguine.

UVa ID: KEY

NAME (print): KEY

UVa ID: (yes, again!) KEY _____

Problem	Max points	Points
1 — Type Checking	16	
2 — Operational Semantics	24	
3 — Code Generation	12	
4 — Analysis and Optimization	15	
5 — Linking, Loading	10	
6 — Game Theory	10	
7 — Short Answer	18	
Extra Credit	0	
TOTAL	105	

Honor Pledge:

How do you think you did? _____

1 Type Checking (16 points)

Consider an extension of Cool to support arrays of objects. We introduce an `Array` class that inherits from `Object`. Other classes cannot inherit from the `Array` class. We introduce four new expressions for manipulating Cool arrays:

$$
\begin{array}{rcl}
e & ::= & \texttt{new Array}[e] \\
& | & e_1[e_2] \\
& | & e_1[e_2] \leftarrow e_3 \\
& | & \texttt{foreach } v_i, v_e \texttt{ in } e_1 \texttt{ do } e_2
\end{array}
$$

Subexpressions are evaluated left-to-right (e.g., e_1 before e_2). The first expression form creates a new array of size e. The array initially holds e separate copies of `new Object`. The size must be non-negative at runtime to avoid an exception. The second expression form reads from array e_1 at index e_2, returning the object stored there. The third writes to array e_1 at index e_2 the value e_3 (and returns e_3). For reads and writes, the index must be between 1 and the size of the array at runtime to avoid an exception. The final expression executes e_2 for every element in array e_1 with variable name v_i bound to the that element's index and variable name v_e bound to that element's value. Each element is considered in ascending order starting from 1. For example, this code:

```
let arr : Array <- new Array[3] in
arr[3] <- 5309;
arr[1] <- 867;
arr[2] <- "unicorn";
foreach i, elt in arr do {
        out_string("element ") ; out_int(i) ;
        out_string(" is ");
        case elt of
                n : Int => out_int(n) ;
                s : String => out_string(s) ;
        esac;
        out_string("\n");
} ;
```

Produces:

```
element 1 is 867
element 2 is unicorn
element 3 is 5309
```

Give typing rules for the four new array expressions. Be as permissive as possible without permitting any unsafe programs. *[4 pts]* each.

$$\frac{O, M, C \vdash e : Int}{O, M, C \vdash \texttt{new Array}[e] : Array}$$

$$\frac{O, M, C \vdash e_1 : Array \quad O, M, C \vdash e_2 : Int}{O, M, C \vdash e_1[e_2] : Object}$$

$$\frac{O, M, C \vdash e_1 : Array \quad O, M, C \vdash e_2 : Int \quad O, M, C \vdash e_3 : T}{O, M, C \vdash e_1[e_2] \leftarrow e_3 : T}$$

$$\frac{O, M, C \vdash e_1 : Array \quad O[v_i \mapsto Int][v_e \mapsto Object], M, C \vdash e_2 : T}{O, M, C \vdash \texttt{foreach } v_i, v_e \texttt{ in } e_1 \texttt{ do } e_2 : Object}$$

2 Operational Semantics and Exceptions (24 points)

Following the previous problem, we extend our notion of Cool values v to include array objects and a generic array exception used to report any array problem at run-time. Recall our use of a generalized return value g to model exceptions.

$$
\begin{array}{rcl}
v & ::= & \texttt{void} \\
& | & X(a_1 = l_1, \ldots, a_n = l_n) \\
& | & \text{Array}(l_1, \ldots, l_n) \\
& | & \text{ArrayException}
\end{array}
\qquad\qquad
\begin{array}{rcl}
g & ::= & \mathsf{Norm}(v) \\
& | & \mathsf{Exc}(v)
\end{array}
$$

Give the new operational semantics rules. Write only those rules where all subexpressions e, e_1 and e_2 return Normal values. *[3 pts]* each.

$$
\frac{\begin{array}{l} so, E, S_1 \vdash e : \mathsf{Norm}(Int(n)), S_2 \\ n > 0 \\ l_1, \ldots, l_n = \ldots newloc(S_1) \ldots \\ S_3 = S_2[l_1 \mapsto D_{\text{Object}}, \ldots, l_n \mapsto D_{\text{Object}}] \\ v = \mathsf{Norm}(\text{Array}(l_1, \ldots, l_n)) \end{array}}{so, E, S_1 \vdash \texttt{new Array}[e] : v, S_3}
$$

$$
\frac{\begin{array}{l} so, E, S_1 \vdash e : \mathsf{Norm}(Int(n)), S_2 \\ n \leq 0 \\ v = \mathsf{Exc}(\text{ArrayException}) \end{array}}{so, E, S_1 \vdash \texttt{new Array}[e] : v, S_2}
$$

$$
\frac{\begin{array}{l} so, E, S_1 \vdash e_1 : \mathsf{Norm}(Array(l_1 \ldots l_n)), S_2 \\ so, E, S_2 \vdash e_2 : \mathsf{Norm}(Int(i)), S_3 \\ 1 \leq i \leq n \\ v = \mathsf{Norm}(S_3[l_i]) \end{array}}{so, E, S_1 \vdash e_1[e_2] : v, S_3}
$$

$$
\frac{\begin{array}{l} so, E, S_1 \vdash e_1 : \mathsf{Norm}(Array(l_1 \ldots l_n)), S_2 \\ so, E, S_2 \vdash e_2 : \mathsf{Norm}(Int(i)), S_3 \\ (i < 1) \ \vee \ (i > n) \\ v = \mathsf{Exc}(\text{ArrayException}) \end{array}}{so, E, S_1 \vdash e_1[e_2] : v, S_3}
$$

$$
\frac{\begin{array}{l} so, E, S_1 \vdash e_1 : \mathsf{Norm}(Array(l_1 \ldots l_n)), S_2 \\ so, E, S_2 \vdash e_2 : \mathsf{Norm}(Int(i)), S_3 \\ so, E, S_3 \vdash e_3 : \mathsf{Norm}(v), S_4 \\ 1 \leq i \leq n \\ S_5 = S_4[l_i \mapsto v] \end{array}}{so, E, S_1 \vdash e_1[e_2] \leftarrow e_3 : \mathsf{Norm}(v), S_5}
$$

$$
\frac{\begin{array}{l} so, E, S_1 \vdash e_1 : \mathsf{Norm}(Array(l_1 \ldots l_n)), S_2 \\ so, E, S_2 \vdash e_2 : \mathsf{Norm}(Int(i)), S_3 \\ so, E, S_3 \vdash e_3 : \mathsf{Norm}(v), S_4 \quad \text{(optional)} \\ (i < 1) \ \vee \ (i > n) \\ v = \mathsf{Exc}(\text{ArrayException}) \end{array}}{so, E, S_1 \vdash e_1[e_2] \leftarrow e_3 : v, S_4}
$$

Answer Variant #1:

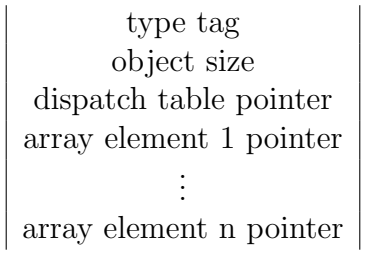
$$\frac{\begin{array}{l} so, E, S_1 \vdash e_1 : \mathsf{Norm}(Array(l_1 \ldots l_n)), S_2 \\ l_{v_i} = newloc(S_2) \\ so, E[v_i \mapsto l_{v_i}, v_e \mapsto l_1], S_2[l_{v_i} \mapsto Int(1)] \vdash e_2 : v_1, S_{2+1} \\ \vdots \\ so, E[v_i \mapsto l_{v_i}, v_e \mapsto l_n], S_{2+n-1}[l_{v_i} \mapsto Int(n)] \vdash e_2 : v_1, S_{2+n} \end{array}}{so, E, S_1 \vdash \texttt{foreach}\ v_i, v_e\ \texttt{in}\ e_1\ \texttt{do}\ e_2 : void, S_{2+n}}$$

Answer Variant #2:

$$\frac{\begin{array}{l} so, E, S_1 \vdash e_1 : \mathsf{Norm}(Array(l_1 \ldots l_n)), S_2 \\ c = \text{“}\texttt{let v}_\texttt{i}\ \texttt{: Int} \leftarrow \texttt{1 in} \\ \quad \texttt{while v}_\texttt{i} \leq \texttt{n loop} \\ \quad\quad \texttt{let v}_\texttt{e}\ \texttt{: Object} \leftarrow \texttt{e}_\texttt{1}[\texttt{v}_\texttt{i}]\ \texttt{in} \\ \quad\quad e_2; \\ \quad\quad v_i \leftarrow v_i + 1; \\ \quad \texttt{pool}\text{”} \\ so, E, S_2 \vdash c : v, S_3 \end{array}}{so, E, S_1 \vdash \texttt{foreach}\ v_i, v_e\ \texttt{in}\ e_1\ \texttt{do}\ e_2 : void, S_3}$$

3 Code Generation (12 points)

In this problem we consider code generation for the arrays introduced in the previous problems. Following the approach described in class, we decide to lay out array objects as follows:



[8 pts] Give the stack machine code generation rule for $cgen(e_1[e_2] \leftarrow e_3)$. You may assume there is a function at label raise_ArrayException; call to that label with no arguments to raise an *ArrayException* at run-time.

```
cgen(e1[e2] <- e3) =
        cgen(e1)
        push acc
        cgen(e2)
        tmp1 <- acc             # tmp1 holds index
        pop tmp2                # tmp2 holds array
        ld tmp3 <- tmp2[1]      # load array object size
        tmp3 <- tmp3 - 3        # number of elements
        if tmp1 < 1 goto err    # index too low
        if tmp2 > tmp3 goto err # index too high
        tmp1 <- tmp1 + 2
        tmp1 <- tmp1 + tmp2     # tmp1 points to e1[e2]
        push tmp1
        cgen e3
        pop tmp1
        tmp1 <- acc
        jmp end
err:
        call raise_ArrayException
end:
```

[4 pts] Give the general formula for $NT(e_1[e_2] \leftarrow e_3)$, the number of temporaries required.

```
Option A: max( NT(e1), 1 + NT(e2), 1 + NT(e3) )

Option B: max( NT(e1), 1 + NT(e2), 2 + NT(e3) )
```

4 Analysis and Optimization (15 points)

We desire to eliminate expensive array bounds checks at run-time in cases where we can prove statically that the indices are in bounds. To this end we introduce an internal analysis that tracks upper and lower bounds for integer variables. We write $[low, high]$ to indicate the bounds for a variable. The special values ∞ and $-\infty$ are also allowed in bounds. Thus, $[3, 3]$ means the variable is exactly 3, $[3, 5]$ means the variable must be between 3 and 5, and $[-\infty, \infty]$ means the variable could be anything. We also write # for information at a location that is not (yet) reachable. For example:

```
arr <- new Array[10];
x <- 5;
if user_input() > 2:
        x <- 3;
else:
        x <- 7;
x <- x + 1;
arr[x] <- "alicorn";
```

At the end of the code snippet, $[4, 8]$ is a bound for x, so we can eliminate the normal bounds check from the array write. We write $B_{in}(x, s)$ and $B_{out}(x, s)$ for the bounds values for x before and after statement s, respectively. Give the local transfer functions for B_{out} in terms of B_{in}. Assume $B_{in}(x, s) = [a, b]$; do not consider # for these four.

(a) *[1 pt]* $B_{out}(x, x \leftarrow const) = [const, const]$

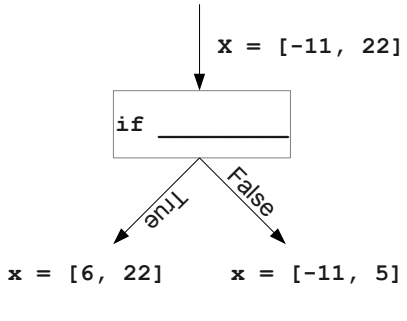
(b) *[1 pt]* $B_{out}(x, y \leftarrow \ldots) = B_{in}(x, s)$ or $[a, b]$

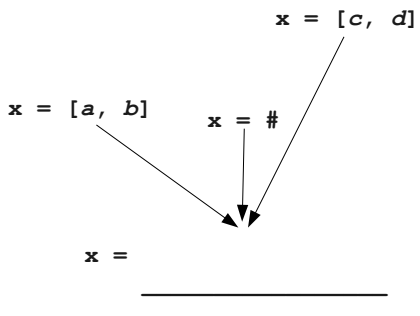
(c) *[2 pts]* $B_{out}(x, x \leftarrow x + const) = [a + const, b + const]$

(d) *[1 pt]* $B_{out}(x, x \leftarrow function()) = [-\infty, \infty]$

(e) Fill in each blank with the most general and precise answer. *[5 pts]* each.

`x > 5` or `x >= 6` `[min(a,c), max(b,d)]`



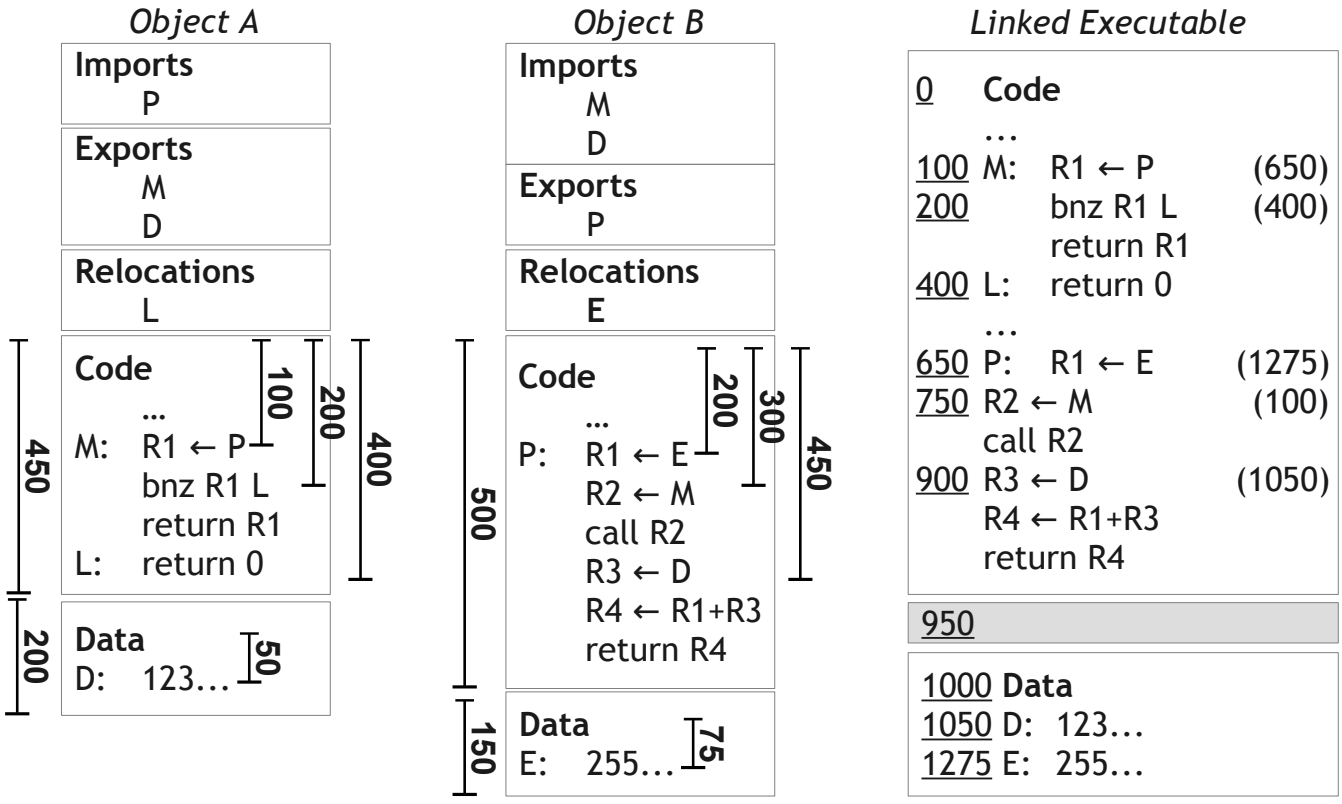


5 Linking, Loading and Libraries (10 points)

In the diagram below, two relocatable objects and their resulting linked executable are shown. The final addresses in the linked executable are shown underlined.

(a) *[4 pts]* Fill in each box labeled "Imports", "Exports" or "Relocations" with the appropriate variable or label names.

(b) *[6 pts]* Label each offset measurement bar with the appropriate amount. Four of the bars are already labeled for you. Fill in the remaining five bars for Object A and three bars for Object B.



6 Game Theory (10 points)

Following our study of two-person impartial games of perfect information, we decide to extend Nim to be more forgiving. We consider a version of Nim in which you must remove one or more items from the same heap, but may also elect to take *one* item from a *different* heap if you like. (You don't have to take that one extra item from a separate heap — you can skip that option.) We call this the "Nim With Interesting Takes" variant, or *Nimwit*.

Note that in Nimwit, two identical heaps need *not* be a loss for the current player. If you are facing two singleton heaps A and B, you could take one from heap A then elect to take one from heap B, leaving your opponent facing the empty board (and thus a game loss).

Consider the following Nimwit game of three heaps:

A A | B | C

(a) *[7 pts]* If you are going first, what move would you make to win? *Circle* the items you would *remove*.

`Answer: Take one A.`

(b) *[3 pts]* We mentioned in class that all impartial games are equivalent to Nim. What standard Nim sum does this Nimwit game equal? (That is, what height singleton standard Nim heap does it equal?)

`Answer: 4.`

7 Short Answer (18 points)

For each short answer question, limit your answer to at most four sentences.

(a) *[3 pts]* One weakness of the array approach considered here is that a case expression is typically required to make use of a retrieved object. Argue for or against the claim that a small change to SELF_TYPE could alleviate this problem.

Against. SELF_TYPE allows us to typecheck good methods (that we would otherwise reject) that return self in the presence of inheritance. Our arrays cannot be inherited from, so SELF_TYPE_{Array} is Array or any subtype of Array, which is just Array, so it does not help directly. Instead, we would want to extend our type system so that Array_{Int} is treated differently from Array_{String}, for example.

(b) *[3 pts]* How would you change our array object layout to support inheriting from and extending the `Array` class?

Our array object layout treats the array entries as fields. Different arrays can have different number of entries. But for dynamic dispatch and inheritance to work, all fields must be at

a known offset. If Point inherits from Array and adds x and y fields, the x field would be at offset 4 in an array of size 1, but at offset 5 in an array of size 2. One solution: rather than "inlining" each array element separately, instead have one object field that is a pointer to all of the array elements. Now the x field in our example is always at offset 4, regardless of the array size.

(c) *[3 pts]* Argue for or against the claim that the debugging table approach described in class can be applied to programs that use stop-and-copy garbage collection.

For. The debugging table must be able to locate local and global variables in memory. Those local and global variables are exactly the *roots* in garbage collection. Stop-And-Copy does not move the roots, it moves the heap objects pointed to by the roots. For example, if Register 1 holds local variable X, after Stop-And-Copy, Register 1 will still hold local variable X (although its value may have changed if X is a pointer). So a debugging table that says local variable X lives in Register 1 will be correct in with copying garbage collection. (Note that more complicated heap variables, like X-¿Y-¿Z, still just start from the root X, and so work fine.)

(d) *[3 pts]* What information would need to be communicated by the compiler and/or runtime system to admit accurate sampling-based profiling in the presence of garbage collection?

The profiler would need to be able to tell the difference between normal user program execution and garbage collection. This should be as simple as indicating which program counter ranges correspond to the garbage collector code. The profiler could then report garbage collection time and user function time separately.

(e) *[3 pts]* Suppose we are creating a Cool-to-C native interface. Fill in the source code below for a C method to convert Cool `Int` objects to C `int` values.

```
int cool_Int_to_c_int(cool_object * cobj) {
  int * cobj_as_int = (int *) cobj;
  return cobj_as_int[3]; // int value
  // Offset 0 is TypeTag, Offset 1 is Size, Offset 2 is VtablePtr
}
```

(f) *[3 pts]* Suppose you are writing C code that is called from OCaml as part of a multi-language project. When would the runtime system ever need to invoke the garbage collector inside your C code?

Whenever your C glue code needs to allocate memory. This is typically when you need to create a new OCaml value, such as a Tuple or a String.

8 Extra Credit (0 points)

"No Answer" is *not* valid on extra credit questions.

(a) *[1 pt]* Answer the following true-false questions about `SELF_TYPE`.

i. FALSE: $T \leq \texttt{SELF\_TYPE}_T$

ii. FALSE: `SELF_TYPE` helps us to reject incorrect programs that are not rejected by the normal type system.

iii. FALSE: `SELF_TYPE` is a dynamic type.

iv. FALSE: A formal parameter to a method can have type `SELF_TYPE`.

v. TRUE: If the return type of method f is `SELF_TYPE` then the static type of $e_0.f(e_1, \ldots, e_n)$ is the static type of e_0.

(b) *[1 pt]* List one thing you would like to see changed or improved in the class. List one thing that you are enjoying.

(c) *[2 pts]* Cultural literacy. Below are the English translations or names for ten concepts or figures in world folklore, legend, religion or mythology. Each concept is associated with one of the ten most common languages (by current number of native-language speakers; Ethnologue estimate). For each concept, give the associated language. Be specific.

- German. The Pied Piper.
- Mandarin/Chinese. The Eight Immortals.
- English. King Arthur.
- Japanese. Kami.
- Hindi. Kamayani.
- Arabic. Jinn.
- Russian/Slavic. Ilya Muromets.
- Portuguese. Endovelicus.
- Spanish. Don Juan.
- Bengali. Bankubabur Bandhu.

9 Twtr: Which tongues work best for microblogs? (0 points)

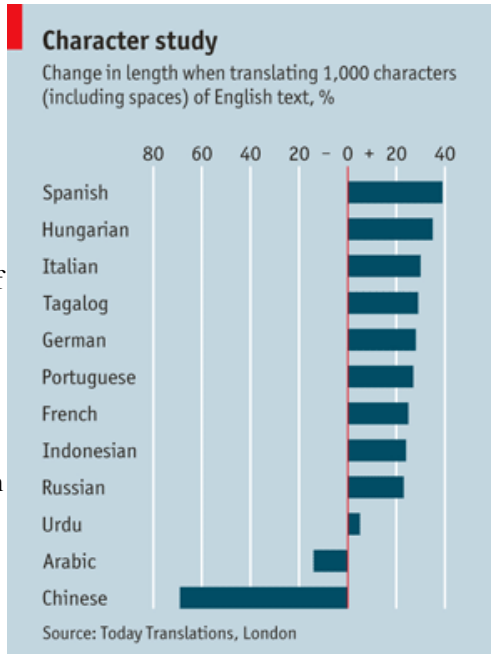
(for fun and stress relief only — no questions on this page)

The Economist – March 31st 2012

THIS 78-character tweet in English would be only 24 characters long in Chinese. That makes Chinese ideal for micro-blogs, which typically restrict messages to 140 symbols. Though Twitter, with 140m active users the world's best-known microblogging service, is blocked in China, Sina Weibo, a local variant, has over 250m users. Chinese is so succinct that most messages never reach that limit, says Shuo Tang, who studies social media at the University of Indiana.

Japanese is concise too: fans of haiku, poems in 17 syllables, can tweet them readily. Though Korean and Arabic require a little more space, tweeters routinely omit syllables in Korean words; written Arabic routinely omits vowels anyway. Arabic tweets mushroomed last year, though thanks to the uprisings across the Middle East rather than any linguistic features. It is now the eighth most-used language on Twitter with over 2m public tweets every day, according to Semiocast, a Paris-based company that analyses social-media trends.

Romance tongues, among others, generally tend to be more verbose (see chart). So Spanish and Portuguese, the two most frequent European languages in the Twitterverse after English, have tricks to reduce the number of characters. Brazilians use "abs" for *abraços* (hugs) and "bjs" for *beijos* (kisses); Spanish speakers need never use personal pronouns ("I go" is denoted by the verb alone: *voy*). But informal English is even handier. It allows personal pronouns to be dropped, has no fiddly accents and enjoys a well developed culture of abbreviation. "English is unmatched in its acronyms, such as DoD for department of defence," says Mohammed al-Basha, a spokesman for the Yemeni government, who tweets in English and Arabic.



Twitter's growth around the world has reduced the proportion of total global tweets in English to 39% from two-thirds in 2009, but polyglot tweeters still often favour the language because of its ubiquity. Many Arabic-speaking revolutionaries used it to get their messages to a larger audience during the Arab spring, sometimes using automatic translation services. Until a recent upgrade, users of Arabic, Farsi and Urdu had trouble using hashtags (words prefixed with the # sign to mark a tweet's subject). Some people use English to avoid censorship. Micro-bloggers on Sina Weibo (where messages containing some characters are automatically blocked) wrote "Bo" in English in order to comment freely about Bo Xilai, a purged party chief.

Though ubiquity and flexibility may give English hegemony, Twitter is also helping smaller and struggling languages. Basque- and Gaelic-speakers tweet to connect with other far-flung speakers. Kevin Scannell, a professor at St Louis University, Missouri, has found 500 languages in use on Twitter and has set up a website to track them. Gamilaraay, an indigenous Australian language, is thought to have only three living speakers. One of them is tweeting—handy for revivalists.