• Write your name and UVa ID on the exam. Pledge the exam before turning it in.

• There are 15 pages in this exam (including this one) and 9 questions, each with multiple sub-questions.

• You have up to 1 hour and 30 minutes to work on the exam.

• The exam is closed book, but you may refer to your two pages of handwritten notes.

• 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!!

• If you leave a subquestion blank or write “no answer” for a sub-question (e.g., 1a or 3b) you will receive one-third of the points for that sub-question (rounded down) since you did not waste our time. If you randomly guess and throw words at us, we will be significantly less sanguine.

UVa ID: \hspace{1cm} \text{KEY} \\

NAME (print): \hspace{1cm} \text{KEY}
<table>
<thead>
<tr>
<th>Problem</th>
<th>Max</th>
<th>Your Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 — Type Checking and Dispatch</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>2 — Operational Semantics</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>3 — Optimization</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>4 — Exceptions</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5 — Security</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>6 — Automatic Memory Management</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>7 — Debugging and Profiling</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>8 — Code Generation</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9 — Linking</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Extra Credit</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>

Honor Pledge:
1 Type Checking and Static Dispatch (15 points)

Consider the following incorrect typing judgment for the non-SELF_TYPE case of method dispatch:

\[
O, M, C \vdash e_0 : T_0 \\
\vdots \\
O, M, C \vdash e_n : T_n \\
T_0 \leq T \quad T \neq \text{SELF_TYPE} \\
M(T, f) = (T'_1, \ldots, T'_n, T'_{n+1}) \\
\forall 1 \leq i \leq n. T'_i \leq T_i \\
O, M, C \vdash e_0 \circ T.f(e_1, \ldots, e_n) : T'_{n+1} \quad \text{wrong}
\]

In addition, consider this Cool code:

```cool
class Language { getName() : String { ... }; }

class OOLanguage inherits Language { wellLoved() : Bool { ... }; }

class Cool inherits OOLanguage { hasCase() : Bool { ... }; 
    wellLoved() : Bool { false }; -- hmmm ...
};

class Main {
    willUseOO(l : OOLanguage) : Bool { l.wellLoved() = true }

    main() : Object {
        (* your code will go here *)
    }
}
```

(a) [7 pts] The modified typing rule wrong is too strict: it rejects good programs that are accepted by the normal typing rules. Give Cool code for the body of main above that is accepted by the normal typing rules but is rejected by rule wrong.

```cool
self@Main.willUseOO(new Cool)
```

Note: The mistake is \( T'_i \leq T_i \) for the arguments — that should be \( \geq \) instead.

(b) [8 pts] The modified typing rule wrong is also unsound: it allows programs that will lead to run-time errors. Give Cool code for the body of main above that is accepted by rule wrong but that is rejected by the normal typing rules.

```cool
self@Main.willUseOO(new Language)
```
2 Operational Semantics (15 points)

We would like to add a simple for loop to Cool:

\[ \text{for } i = e_{\text{low}} \text{ to } e_{\text{high}} \text{ do } e_{\text{body}} \]

It has the expected intuitive semantics: the expression \( e_{\text{body}} \) is executed multiple times, with the loop induction variable \( i \) bound to every integer between \( e_{\text{low}} \) and \( e_{\text{high}} \) (inclusive). Thus this loop:

```java
class Main {
    main() : Object { for i = 1 to 8 do {
        out_int(i); out_string(" "); i <- i + 2;
    } } ;
}
```

outputs 1 4 7. Note that only \( e_{\text{body}} \) may depend on \( i \). Informally and imprecisely, we can define for in terms of let and while:

\[
\begin{align*}
\text{let } i : \text{Int} & \leftarrow e_{\text{low}} \text{ in } \\
\text{while } i \leq e_{\text{high}} \text{ loop } \\
\text{for } i = e_{\text{low}} \text{ to } e_{\text{high}} \text{ do } e_{\text{body}} & = \text{e_body} ; \\
\ & i \leftarrow i + 1 ; \\
\} \text{ pool }
\end{align*}
\]

(a) [10 pts] Just as there are two operational semantics rules for if and two rules for while, we will have two rules for for. Complete the following “inductive step” operational semantics rule for for. In this rule \( e_{\text{body}} \) does execute. Evaluate \( e_{\text{low}} \) before \( e_{\text{high}} \). Your answer may not involve the syntax “;” or the word “while”. Remember, we are asking for a single inference rule, not an entire derivation.

\[
\begin{align*}
E, S_1 & \vdash e_{\text{low}} : \text{Int}(i_{\text{low}}), S_2 \\
E, S_2 & \vdash e_{\text{high}} : \text{Int}(i_{\text{high}}), S_3 \\
i_{\text{low}} & \leq i_{\text{high}} \\
l & = \text{newloc}(S_3) \\
E[l/i], S_3[l/\text{Int}(i_{\text{low}})] & \vdash e_{\text{body}} : v_1, S_4 \\
i_{\text{next}} & = S_4[l] \\
E, S_4 & \vdash \text{for } i = i_{\text{next}} + 1 \text{ to } e_{\text{high}} \text{ do } e_{\text{body}} : v_2, S_5 \\
& \quad \text{for true}
\end{align*}
\]

so, \( E, S_1 \vdash \text{for } i = e_{\text{low}} \text{ to } e_{\text{high}} \text{ do } e_{\text{body}} : \text{Int}(0), S_5 \)

Note: The Cool opsem rule for while \( e_1 \text{ loop } e_2 \text{ pool } \) also evaluates \( e_2 \) “multiple times”. This is the desired behavior. However, we only want to evaluate \( e_{\text{low}} \) once. We’re not using the let rule directly because we need to extract the value \( i_{\text{next}} \).
(b) [5 pts] Consider the following incorrect operational semantics rule for for.

\[
\begin{align*}
\text{so, } E, S_1 &\vdash e_{low} : \text{Int}(i_{low}), S_2 \\
\text{so, } E, S_2 &\vdash e_{high} : \text{Int}(i_{high}), S_3 \\
i_{low} \leq i_{high} &\quad \text{so, } E, S_3 \vdash \text{let } i : \text{Int} \leftarrow i_{low} \text{ in } e_{body} : v_1, S_4 \\
\text{so, } E, S_4 &\vdash \text{for } i = i_{low} + 1 \text{ to } e_{high} \text{ do } e_{body} : v_2, S_5 \\
\text{so, } E, S_1 &\vdash \text{for } i = e_{low} \text{ to } e_{high} \text{ do } e_{body} : \text{Int}(0), S_5
\end{align*}
\]

for wrong

Write a simple Cool expression that does not behave according to our original definition for for when evaluated using the for wrong rule. Strive for a succinct answer; you should not need more than about twenty words.

```java
class Main {
    main() : Object { for i = 1 to 8 do {
        out_int(i); out_string("_"); i <- i + 2;
    } }
}
```

Note: Actually, the example from bfore works fine. The problem with this rule is that it does not notice updates to i inside the loop.
3 Optimization (15 points)

(a) [9 pts] The following block of code makes use of five variables: a, b, c, d, and e. However, we have erased many of these variable references from the original program. In the right-hand column, we provide the results of liveness analysis (i.e., the variables that are live at each program point). Please fill in each blank with a variable so that the program is consistent with the results of liveness analysis.

Please note that there are no dead instructions in this program. You will need this information to fill in some of the blanks correctly!

<table>
<thead>
<tr>
<th>Code</th>
<th>Live Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>d := b + e</td>
<td>{b, c, e}</td>
</tr>
<tr>
<td>c := c + e</td>
<td>{c, d, e}</td>
</tr>
<tr>
<td>b := c + d</td>
<td>{c, d}</td>
</tr>
<tr>
<td>b := b * b</td>
<td>{b}</td>
</tr>
<tr>
<td>a := 42</td>
<td>{a, b}</td>
</tr>
<tr>
<td>c := a * b</td>
<td>{c, b}</td>
</tr>
<tr>
<td>print b</td>
<td>{c, b}</td>
</tr>
<tr>
<td>print c</td>
<td>{}</td>
</tr>
</tbody>
</table>

Note: The “Please note that there are no dead instructions in this program.” actually mattered!
(b) [6 pts] Draw a control-flow graph for the following code. Each node in your control-flow graph should be a basic block. Yes, this is tricky. Every statement in the code should appear somewhere in your control-flow graph.

START
  if a < b then { a <- 11 }
  else { b <- 22 }
  d <- 33
here:  e <- e + 44
  f <- 55
  if g < h then { goto here }
  else { goto there }
  i <- 66
there: j <- 77
END

Dig my ASCII art:

```
+-------+ +---------+     +---------+     +---------+
| START | | a <- 11 | | a < b |------>| |
+-------+ +---------+     +---------+     +---------+
     |     |     |     |     |     |     |
     v     v     v     v     v     v
+---------+     +---------+     +---------+
| b <- 22 | | d <- 33 | <----------/
+---------+     +---------+     +---------+
     |     |     |     |     |     |
     v     v     v     v     v
+--------+ +----------+
| e <- e + 44 | | f <- 55 |
+--------+ +----------+
     |     |     |     |     |
     v     v     v     v
+----------+ +--------+
| g < h |------>| |
+----------+ +--------+
     |     |     |     |     |
     v     v     v     v
+--------+ +----------+
| i <- 66 | | j <- 77 |
+--------+ +----------+
     |     |     |     |
     v     v     v     v
+----------+
| END |     |
+----------+```
4 Exceptions (10 points)

(a) [2 pts] Give one example of an exception that is the clear result of a mistake in a program. Then give one example of an exception that might be raised even in a perfect program.

Clear mistake: division by zero, null pointer dereferenced, segmentation violation, etc.
Could happen to anyone: out of disk space, network send failed, print queue full, etc.

(b) [4 pts] Explain where and why the least-upper-bound operator ⊔ is used in our formal treatment of language-level exception handling.

A full-credit explanation should relate try-catch to if-then and talk about how ⊔ is used to compute a conservative approximation to the type (because we cannot be sure at compile-time whether the exception will be raised or not).

(c) [4 pts] Consider the following incorrect unsound typing rule for try-finally.

\[
\frac{O, M, C \vdash e_1 : T_1 \quad O, M, C \vdash e_2 : T_2}{O, M, C \vdash \text{try } e_1 \text{ finally } e_2 : T_1} \quad \text{wrong}
\]

Give a Cool expression that typechecks using this rule but fails at run-time.

(try 1 finally "hello") * 2

Note: The problem is that we always use the type of \( e_1 \). To fail at run-time we must invoke an operation on a type that does not support it.
5 Security (10 points)

(a) [5 pts] Write a C program that is vulnerable to a stack buffer overflow attack on a hypothetical architecture where the stack grows up positively toward high addresses. Normally the stack grows down toward lower addresses, so the saved return address will be placed at a higher address than a stack-allocated buffer. Include the string "char buf[100];" in your answer. Merely overflowing unused space will not qualify: the attacker must be able to execute arbitrary injected code. (Hint: this can be done in as few as seven short lines.)

```c
myfun(char *p) {
    unsafe_read_from_user(p);
    return;
}
main() {
    char buf[100];
    myfun(buf);
    return;
}
```

(b) [5 pts] Let’s add string-indexing syntax to Cool. The following code prints out the string “e”:

```cool
let x : String <- "Buffer" in out_string(x[4])
```

Here is a typing rule for the string-indexing operator:

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

To avoid buffer overruns, our string-indexing operation will raise an exception if the index is out of bounds. The value of that exception should be the integer value of the out-of-bounds index. We’ll use the generalized return values approach to operational semantics. Complete the index – good and index – bad rules below. For simplicity, you may write “a is the ith character of s” and “len(s)” in hypotheses.

\[
\frac{so, E, S_1 \vdash e_1 : Exc(v), S_2}{so, E, S_1 \vdash e_1[e_2] : Exc(v), S_2} \quad \text{index} - 1
\]

\[
\frac{so, E, S_1 \vdash e_1 : Norm(v_1), S_2 \quad so, E, S_2 \vdash e_2 : Norm(v_2), S_3}{so, E, S_1 \vdash e_1[e_2] : Exc(v_2), S_3} \quad \text{index} - 2
\]

- good: so, E, S_1 \vdash e_1 : Norm(String(l, s)), S_2
- bad: so, E, S_1 \vdash e_1 : Exc(Int(l)), S_2

- so, E, S_2 \vdash e_2 : Norm(Int(i)), S_3
- i < 0 or i ≥ l

- a is the ith character of s
6 Automatic Memory Management (10 points)

(a) [2 pts] What can happen to a program that mistakenly frees memory too early and then later tries to access it?

“Anything.” More formally, the behavior is undefined. Attackers can leverage this sort of mistake to crash or take over your machine, for example.

(b) [3 pts] Why is Stop and Copy a poor choice for C and C++ programs? Why does Mark and Sweep work better?

Stop and Copy is a poor choice because it moves allocated objects. In C or C++, you cannot move allocated objects because there may be hidden references to them (e.g., cast as integers). Mark and Sweep would work better because it does not need to move objects: there are conservative Mark and Sweep collectors for C++ that find places to store mark bits (e.g., externally or in malloc() header information) and conservatively trace all possible pointers.

(c) [3 pts] Consider the following program:

```c
while not_done() {
    ptr = malloc(100 * MEGABYTE);
    do_work(ptr);
    /* done with ptr */
}
```

You are running this program with 4 gigabytes of physical memory and want to use automatic memory management. Would you choose Mark and Sweep or Stop and Copy? Why?

Stop and Copy is better here because it takes time proportional to the amount of live memory only (which in this case is either 100 MB or 0 MB depending on how you count). By contrast, Mark and Sweep takes time proportional to all of memory (4 GB). So even though Stop and Copy will run twice as often (because it cuts memory in half), each of those runs is more than twice as fast!

(d) [2 pts] In the context of automatic memory management, what are the roots?

The roots are local variables, temporaries, function arguments and other variables on the stack (e.g., arguments or local variables for caller functions). Tracing for live objects begins from the roots.
7 Debugging and Profiling (10 points)

(a) [2 pts] Name one advantage of sampling-based profiling. Then name one disadvantage.

Advantage: Sample-based profiling is light-weight and efficient: it is unlikely to greatly disturb the program being profiled. Disadvantage: it can miss periodic or time-dependent behavior (e.g., if you sample at $0 + 10k$ seconds and I do something at $5 + 10k$ seconds, you’ll never see me).

(b) [2 pts] When during debugging is operating system intervention required? What bad things would happen if the operating system were not required?

OS intervention is required to “attach” to a process and begin to debug it. When you are debugging a process you can change and inspect its state. A security check is thus required: if anyone could attach to the login process, anyone could sniff passwords or other critical information.

(c) [2 pts] Name a code optimization and explain why it would complicate debugging.

Dead code elimination can complicate “single step” debugging because steps may be removed. Dead code elimination can complicate “variable inspection” debugging because a variable may be removed entirely. And so on. For example, loop unrolling or function inlining also complicate “single step” or “step over” debugging, etc.

(d) [4 pts] In the paper “gprof: a call-graph execution profiler”, two major types of profiles are discussed. What are they? Describe at least two differences between them.

The major types were call-graph and flat. A call-graph execution profile keeps track of context: rather than lumping all of the time spent in foo() into one big box, we can tell “main() calls foo()” apart from “bar() calls foo()”. The paper also mentioned “execution time” versus “execution count” profiling, but that was not actually the major contribution (reading for comprehension!). Reread the paper for more.
8 Code Generation (10 points)

(a) [5 pts] Consider the following incorrect stack-machine code generation rule:

\[
\text{cgen}(\text{if } e1 = e2 \text{ then } e3 \text{ else } e4) = \\
c\text{cgen}(e1) \\
p\text{ush } r1 \\
c\text{cgen}(e2) \\
p\text{op } t1 \\
b\text{neq } r1 \ t1 \ \text{false_branch} \\
jm\text{p } \text{true_branch} \\
\text{true_branch: } \text{cgen}(e3) \\
\text{false_branch: } \text{cgen}(e4) \\
\text{end_if:}
\]

Write an expression \( e \) for which the above rule generates incorrect code. Indicate specifically what should happen when your expression \( e \) is evaluated as well as what mistakenly happens when the above rule is used to generate code.

\[
\text{if } 1 = 1 \text{ then print } 5 \text{ else print } 6
\]

Should print 5, will actually print 5 6. Note that our in-class coverage of stack-machine code generation did not talk about “boxing integers” either — so answers related to that were off the mark.

(b) [5 pts] Consider the following Cool class declarations:

Class A {
  p : Int;
  k : String;
  w : Int;
  x() : String { ... };
  z() : String { ... };
}

Class B inherits A {
  e : Int ;
  z : Int ;
  y() : String { ... } ;
  x() : String { ... } ;
}

Complete the following table describing the object (field) layouts:

<table>
<thead>
<tr>
<th></th>
<th>0 – 2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>(header)</td>
<td>p</td>
<td>k</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>(header)</td>
<td>p</td>
<td>k</td>
<td>w</td>
<td>e</td>
<td>z</td>
</tr>
</tbody>
</table>

As well as the following table describing the dispatch table (vtable) layout:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A’s x()</td>
<td>A’s z()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B’s x()</td>
<td>A’s z()</td>
<td>B’s y()</td>
<td></td>
</tr>
</tbody>
</table>
9 Linking (5 points)

(a) [5 pts] In the following scenario, two relocatable objects are being linked together to form an executable. Each object has an import table, an export table, a relocation table, a code segment and a data segment. The rounded braces indicate offsets within segments. For example, the use of glob_a within Object A’s code segment occurs 30 words in, and that segment has a total length of 60. Similarly, the variable glob_b is located 15 words into Object B’s data segment, and that segment has total size 20.

The linked executable is shown on the right. The page size is 100 words, causing the indicated amount of padding (greyed areas). Fill in the ten blanks with absolute post-link addresses. The linked code segment starts at absolute address 0. For example, you should fill in the (___) to the right of glob_b with the final linked address of glob_b.

<table>
<thead>
<tr>
<th>Relocatable Object A</th>
<th>Relocatable Object B</th>
<th>Executable Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Imports</strong></td>
<td><strong>Exports</strong></td>
<td></td>
</tr>
<tr>
<td>fun_b</td>
<td>fun_a</td>
<td></td>
</tr>
<tr>
<td><strong>Exports</strong></td>
<td><strong>Exports</strong></td>
<td></td>
</tr>
<tr>
<td>fun_a</td>
<td>fun_b</td>
<td></td>
</tr>
<tr>
<td>glob_a</td>
<td>main</td>
<td></td>
</tr>
<tr>
<td><strong>Relocations</strong></td>
<td><strong>Relocations</strong></td>
<td></td>
</tr>
<tr>
<td>glob_a</td>
<td>glob_b</td>
<td></td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td><strong>Code</strong></td>
<td></td>
</tr>
<tr>
<td>fun_a:</td>
<td>main:</td>
<td></td>
</tr>
<tr>
<td>local = glob_a</td>
<td>local = glob_b</td>
<td></td>
</tr>
<tr>
<td>fun_b(local)</td>
<td>fun_a(local)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td><strong>Data</strong></td>
<td></td>
</tr>
<tr>
<td>glob_a</td>
<td>glob_a</td>
<td></td>
</tr>
<tr>
<td>(5)</td>
<td>(205)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(160)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(40 words of padding)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(60 words of padding)</td>
<td></td>
</tr>
<tr>
<td>0 Code</td>
<td>15 fun_a</td>
<td></td>
</tr>
<tr>
<td>15 fun_a</td>
<td>glob_a (205)</td>
<td></td>
</tr>
<tr>
<td>80 main</td>
<td>fun_b (160)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>glob_b (235)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fun_a (15)</td>
<td></td>
</tr>
<tr>
<td>160 fun_b</td>
<td>(40 words of padding)</td>
<td></td>
</tr>
<tr>
<td>200 Data</td>
<td>glob_a</td>
<td></td>
</tr>
<tr>
<td>205 glob_a</td>
<td>glob_b</td>
<td></td>
</tr>
<tr>
<td>235 glob_b</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
10 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: SELF_TYPE is a dynamic type.

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

iii. FALSE: $T \leq \text{SELF\_TYPE}$

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) [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.

- BENGALI. Bankubabur Bandhu.
- SPANISH. Don Juan.
- PORTUGUESE. Endovelicus.
- RUSSIAN/SLAVIC. Ilya Muromets.
- ARABIC. Jinn.
- HINDI. Kamayani.
- JAPANESE. Kami.
- ENGLISH. King Arthur.
- MANDARIN/CHINESE. The Eight Immortals.
- GERMAN. The Pied Piper.
11 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.