Tests Template B/testt

Purpose

The command grammar and I6 implementation for testing commands such as TEST, ACTIONS and PURLOIN.

---

§**1. Abstract Command.** The code below is compiled only if the symbol `DEBUG` is defined, which it always is for normal runs in the Inform user interface, but not for Release runs.

Not all of these commands are documented; this is intentional. They may be changed in name or function. This is all of the testing commands except for the GLKLIST command, which is in Glulx.i6t (and does not exist when the target VM is the Z-machine).

We take the commands in alphabetical order, beginning with ABSTRACT, which moves an object to a new position in the object tree.

```
[ XAbstractSub;
    if (XTestMove(noun, second)) return;
    move noun to second;
    "[Abstracted.]";
];
[ XTestMove obj dest;
    if ((obj <= InformLibrary) || (obj == LibraryMessages))
        "[Can't move ", (name) obj, ": it's a system object.]";
    if (obj.component_parent)
        "[Can't move ", (name) obj, ": it's part of ",
        (the) obj.component_parent, ".]";
    while (dest) {
        if (dest == obj) "[Can't move ", (name) obj, ": it would contain itself.]";
        dest = CoreOfParentOfCoreOf(dest);
    }
    rfalse;
];
```

§**2. Actions Command.** ACTIONS turns tracing of actions on.

```
[ ActionsOnSub; trace_actions = 1; say__p = 1; "Actions listing on."; ];
[ ActionsOffSub; trace_actions = 0; say__p = 1; "Actions listing off."; ];
```

§**3. Gonear Command.** GONEAR teleports the player to the vicinity of some named item.

```
[ GonearSub;
    PlayerTo(LocationOf(noun));
];
```

§**4. Purloin Command.** To PURLOIN is to acquire something without reference to any rules on accessibility.

```
[ XPurloinSub;
    if (XTestMove(noun, player)) return;
    move noun to player; give noun moved ~concealed;
    say__p = 1;
    "[Purloined.]";
];
```

§**5. Random Command.** RANDOM forces the random-number generator to a predictable seed value.

```
[ PredictableSub;
    VM_Seed_RNG(-100);
    say__p = 1;
    "[Random number generator now predictable.]";
];
```

§**6. Relations Command.** RELATIONS lists the current state of the mutable relations.

```
[ ShowRelationsSub;
IterateRelations(ShowOneRelation);
];
[ ShowOneRelation rel;
if ((rel-->RR_PERMISSIONS) & (RELS_SHOW)) {
    (rel-->RR_HANDLER)(rel, RELS_SHOW);
}
];
```

§**7. Rules Command.** RULES changes the level of rule tracing.

```
[ RulesOnSub;
    debug_rules = 1; say__p = 1;
    "Rules tracing now switched on. Type ~rules off~ to switch it off again,
    or ~rules all~ to include even rules which do not apply.";
];
[ RulesAllSub;
    debug_rules = 2; say__p = 1;
    "Rules tracing now switched to ~all~. Type ~rules off~ to switch it off again.";
];
[ RulesOffSub;
    debug_rules = 0; say__p = 1;
    "Rules tracing now switched off. Type ~rules~ to switch it on again.";
];
```

§**8. Scenes Command.** SCENES switches scene-change tracing on or off, and also shows the current position.

```
[ ScenesOnSub;
    debug_scenes = 1;
    ShowSceneStatus(); say__p = 1;
    "(Scene monitoring now switched on. Type ~scenes off~ to switch it off again.)";
];
[ ScenesOffSub;
    debug_scenes = 0; say__p = 1;
    "(Scene monitoring now switched off. Type ~scenes~ to switch it on again.)";
];
```

§**9. Scope Command.** SCOPE prints a numbered list of all objects in scope to the player.

```
Global x_scope_count;
[ ScopeSub;
    x_scope_count = 0;
    LoopOverScope(Print_ScL, noun);
    if (x_scope_count == 0) "Nothing is in scope.";
];
[ Print_ScL obj; print_ret ++x_scope_count, ": ", (a) obj, " (", obj, ")"; ];
```

§**10. Showheap Command.** SHOWHEAP is for debugging the memory heap, and is intended for Inform maintainers rather than users.

```
[ ShowHeapSub;
    DebugHeap();
];
```

§**11. ShowMe Command.** SHOWME is probably the most useful testing command: it shows the state of the current room, or a named item.

```
[ ShowMeSub t_0 na;
    t_0 = noun;
    if (noun == nothing) noun = real_location;
    if (ShowMeRecursively(noun, 0, (noun == real_location))) {
        if (noun == real_location)
            print "* denotes things which are not in scope^";
    }
    if (t_0 ofclass K2_thing) {
        print "location:"; ShowRLocation(noun, true); print "^";
    }
    {-call:Plugins::Showme::compile_SHOWME_details}
];
[ ShowRLocation obj top;
    if (obj ofclass K1_room) return;
    print " ";
    if (parent(obj)) {
        if (obj has worn) print "worn by ";
        else {
```

```
            if (parent(obj) has animate) print "carried by ";
            if (parent(obj) has container) print "in ";
            if (parent(obj) ofclass K1_room) print "in ";
            if (parent(obj) has supporter) print "on ";
        }
        print (the) parent(obj);
        ShowRLocation(parent(obj));
    } else {
        if (obj.component_parent) {
            if (top == false) print ", which is ";
            print "part of ", (the) obj.component_parent;
            ShowRLocation(obj.component_parent);
        }
        else print "out of play";
    }
];
[ ShowMeRecursively obj depth f c i k;
    spaces(2*depth);
    if (f && (depth > 0) && (TestScope(obj, player) == false)) { print "*"; c = true; }
    print (name) obj;
    if (depth > 0) {
        if (obj.component_parent) print " (part of ", (name) obj.component_parent, ")";
        if (obj has worn) print " (worn)";
    }
    if (obj provides KD_Count) {
        k = KindHierarchy-->((obj.KD_Count)*2);
        if ((k ~= K2_thing) || (depth==0)) {
            print " - ";
            if (k == K4_door or K5_container) {
                if (obj has transparent) print "transparent ";
                if (obj has locked) print "locked ";
                else if (obj has open) print "open ";
                else print "closed ";
            }
            print (I7_Kind_Name) k;
        }
    }
    print "^";
    if (obj.component_child) c = c | ShowMeRecursively(obj.component_child, depth+2, f);
    if ((depth>0) && (obj.component_sibling))
        c = c | ShowMeRecursively(obj.component_sibling, depth, f);
    if (child(obj)) c = c | ShowMeRecursively(child(obj), depth+2, f);
    if ((depth>0) && (sibling(obj))) c = c | ShowMeRecursively(sibling(obj), depth, f);
    return c;
];
[ AllowInShowme pr;
    if (pr == workflag or concealed or mentioned) rfalse;
    rtrue;
];
```

§**12. Showverb Command.** SHOWVERB is a holdover from old I6 days, but still quite useful. It writes out the I6 command verb grammar for the supplied command.

```
[ ShowVerbSub address lines meta i x;
    wn = 2; x = NextWordStopped();
    if (x == 0 || ((x->#dict_par1) & 1) == 0)
        "Try typing ~showverb~ and then the name of a verb.";
    meta = ((x->#dict_par1) & 2)/2;
    i = DictionaryWordToVerbNum(x);
    address = VM_CommandTableAddress(i);
    lines = address->0;
    address++;
    print "Verb ";
    if (meta) print "meta ";
    VM_PrintCommandWords(i);
    new_line;
    if (lines == 0) "has no grammar lines.";
    for (: lines>0 : lines--) {
        address = UnpackGrammarLine(address);
        print "    "; DebugGrammarLine(); new_line;
    }
    ParaContent();
];
[ DebugGrammarLine pcount;
    print " * ";
    for (: line_token-->pcount ~= ENDIT_TOKEN : pcount++) {
        if ((line_token-->pcount)->0 & $10) print "/ ";
        print (DebugToken) line_token-->pcount, " ";
    }
    print "-> ", (DebugAction) action_to_be;
    if (action_reversed) print " reverse";
];
[ DebugToken token;
    AnalyseToken(token);
    switch (found_ttype) {
    ILLEGAL_TT:
        print "<illegal token number ", token, ">";
    ELEMENTARY_TT:
        switch (found_tdata) {
        NOUN_TOKEN:           print "noun";
        HELD_TOKEN:           print "held";
        MULTI_TOKEN:          print "multi";
        MULTIHELD_TOKEN:      print "multiheld";
        MULTIEXCEPT_TOKEN:    print "multiexcept";
        MULTIINSIDE_TOKEN:    print "multiinside";
        CREATURE_TOKEN:       print "creature";
        SPECIAL_TOKEN:        print "special";
        NUMBER_TOKEN:         print "number";
        TOPIC_TOKEN:          print "topic";
        ENDIT_TOKEN:          print "END";
        }
    PREPOSITION_TT:
        print "'", (address) found_tdata, "'";
```

```
    ROUTINE_FILTER_TT:
        print "noun=Routine(", found_tdata, ")";
    ATTR_FILTER_TT:
        print (DebugAttribute) found_tdata;
    SCOPE_TT:
        print "scope=Routine(", found_tdata, ")";
    GPR_TT:
        print "Routine(", found_tdata, ")";
    }
];
```

§**13. Test Command.** TEST runs a short script of commands from the source text.

```
#Iftrue ({-value:NUMBER_CREATED(test_scenario)} > 0);
[ TestScriptSub;
    switch(special_word) {
{-call:Plugins::Parsing::TestScripts::compile_switch}
    default:
        print ">--> The following tests are available:^";
{-call:Plugins::Parsing::TestScripts::compile_printout}
    }
];
#ifdef TARGET_GLULX;
Constant TEST_STACK_SIZE = 128;
#ifnot;
Constant TEST_STACK_SIZE = 48;
#endif;
Array test_stack --> TEST_STACK_SIZE;
Global test_sp = 0;
[ TestStart T R l k;
    if (test_sp >= TEST_STACK_SIZE) ">--> Testing too many levels deep";
    test_stack-->test_sp = T;
    test_stack-->(test_sp+1) = 0;
    test_stack-->(test_sp+3) = l;
    test_sp = test_sp + 4;
    if ((R-->0) && (R-->0 ~= real_location)) {
        print "(first moving to ", (name) R-->0, ")^";
        PlayerTo(R-->0, 1);
    }
    k=1;
    while (R-->k) {
        if (R-->k notin player) {
            print "(first acquiring ", (the) R-->k, ")^";
            move R-->k to player;
        }
        k++;
    }
    print "(Testing.)^"; say__p = 1;
];
[ TestKeyboardPrimitive a_buffer a_table p i j l spaced ch;
    if (test_sp == 0) {
        test_stack-->2 = 1;
```

```
        return VM_ReadKeyboard(a_buffer, a_table);
    }
    else {
        p = test_stack-->(test_sp-4);
        i = test_stack-->(test_sp-3);
        l = test_stack-->(test_sp-1);
        print "[";
        print test_stack-->2;
        print "] ";
        test_stack-->2 = test_stack-->2 + 1;
        style bold;
        while ((i < l) && (p->i ~= '/')) {
            ch = p->i;
            if (spaced || (ch ~= ' ')) {
                if ((p->i == '[') && (p->(i+1) == '/') && (p->(i+2) == ']')) {
                    ch = '/'; i = i+2;
                }
                a_buffer->(j+WORDSIZE) = ch;
                print (char) ch;
                i++; j++;
                spaced = true;
            } else i++;
        }
        style roman;
        print "^";
        #ifdef TARGET_ZCODE;
        a_buffer->1 = j;
        #ifnot; ! TARGET_GLULX
        a_buffer-->0 = j;
        #endif;
        VM_Tokenise(a_buffer, a_table);
        if (p->i == '/') i++;
        if (i >= l) {
            test_sp = test_sp - 4;
        } else test_stack-->(test_sp-3) = i;
    }
];
#IFNOT;
[ TestScriptSub;
    ">--> No test scripts exist for this game.";
];
#ENDIF;
```

§**14. Trace Command.** Another holdover from I6: TRACE sets the level of parser tracing, on a scale of 0 (off, the default) to 5.

```
[ TraceOnSub; parser_trace=1; say__p = 1; "[Trace on.]"; ];
[ TraceLevelSub;
    parser_trace = parsed_number; say__p = 1;
    print "[Parser tracing set to level ", parser_trace, ".]^";
];
[ TraceOffSub; parser_trace=0; say__p = 1; "Trace off."; ];
```

§**15. Tree Command.** TREE prints out the I6 object tree, though this is not always very helpful in I7 terms. It should arguably be withdrawn, but doesn't seem to do any harm.

```
[ XTreeSub i;
    if (noun == 0) {
        objectloop (i)
            if (i ofclass Object && parent(i) == 0) XObj(i);
    }
    else XObj(noun,1);
];
[ XObj obj f;
    if (parent(obj) == 0) print (name) obj; else print (a) obj;
    print " (", obj, ") ";
    if (f == 1 && parent(obj) ~= 0)
        print "(in ", (name) parent(obj), " ", parent(obj), ")";
    new_line;
    if (child(obj) == 0) rtrue;
    if (obj == Class)
        WriteListFrom(child(obj), NEWLINE_BIT+INDENT_BIT+ALWAYS_BIT+NOARTICLE_BIT, 1);
    else
        WriteListFrom(child(obj), NEWLINE_BIT+INDENT_BIT+ALWAYS_BIT+FULLINV_BIT, 1);
];
```

§**16. Grammar.** In the old I6 parser, testing commands had their own scope hardwired in to the code: this worked by comparing the verb command word directly against `'scope'` and the like. That would go wrong if the testing commands were translated into other languages, and was a crude design at best. The following scope token is better: using this token instead of `multi` provides a noun with universal scope (but restricted to I7 objects, so I6 pseudo-objects like `compass` are not picked up) and able to accept multiple objects.

```
[ testcommandnoun obj o2;
    switch (scope_stage) {
        1: rtrue; ! allow multiple objects
        2: objectloop (obj)
            if ((obj ofclass Object) && (obj provides KD_Count))
                PlaceInScope(obj, true);
        3: print "There seems to be no such object anywhere in the model world.^";
    }
];
{-testing-command:abstract}
    * scope=testcommandnoun 'to' scope=testcommandnoun -> XAbstract;
```

```
{-testing-command:actions}
    *                                   -> ActionsOn
    * 'on'                              -> ActionsOn
    * 'off'                             -> ActionsOff;
{-testing-command:gonear}
    * scope=testcommandnoun             -> Gonear;
{-testing-command:purloin}
    * scope=testcommandnoun             -> XPurloin;
{-testing-command:random}
    *                                   -> Predictable;
{-testing-command:relations}
    *                                   -> ShowRelations;
{-testing-command:rules}
    *                                   -> RulesOn
    * 'all'                             -> RulesAll
    * 'on'                              -> RulesOn
    * 'off'                             -> RulesOff;
{-testing-command:scenes}
    *                                   -> ScenesOn
    * 'on'                              -> ScenesOn
    * 'off'                             -> ScenesOff;
{-testing-command:scope}
    *                                   -> Scope
    * scope=testcommandnoun             -> Scope;
{-testing-command:showheap}
    *                                   -> ShowHeap;
{-testing-command:showme}
    *                                   -> ShowMe
    * scope=testcommandnoun             -> ShowMe;
{-testing-command:showverb}
    * special                           -> Showverb;
{-testing-command:test}
    *                                   -> TestScript
    * special                           -> TestScript;
{-testing-command:trace}
    *                                   -> TraceOn
    * number                            -> TraceLevel
    * 'on'                              -> TraceOn
    * 'off'                             -> TraceOff;
{-testing-command:tree}
    *                                   -> XTree
    * scope=testcommandnoun             -> XTree;
```