PunyInform

An Inform library for writing small and fast text adventures.

# Design decisions left to document

* If you need more than four names for an object in a z3 game, give it a parse\_name routine.
* If you want the library to handle darkness, specify the constant HANDLE\_DARKNESS. Hm, are we sure?
* Verbs and actions are divided into groups, which can be enabled or disabled with constants:
  + Basic actions which the library knows how to perform (Take, Drop, Go etc) – always enabled
  + Common actions which don’t do anything except display a standard reply – enabled when ACTION\_GROUPS >= 1
  + Less common actions which don’t do anything except display a standard reply – enabled when ACTION\_GROUPS >= 2
* All words describing directions are synonyms of the Directions object. Calling Directions.parse\_name() returns the number of words matched and sets selected\_direction\_index to the direction# (or 0 for no direction found) and selected\_direction to the property# used to store that exit in a room (or 0 for no direction found). Whenever a direction has been found in a player command, these properties have already been set. The programmer can print the name of the currently selected direction with print (name) Directions;
* When the player is inside an object, in a z5 game, the library will print the name of the object on the statusline, in definite form (“The box”). In a z3 game, the object name string will be printed as-is, typicall like “box”. This behavior in z3 games is part of the Z-machine specification. If you want a z3 game to print a different name for when the player is inside the object, you can set the object name string to the desired name, and override it with short\_name for all other uses, like this:   
  Object box “The box”   
   with short\_name “box”   
   has container openable enterable;
* Property daemon is an alias for property time\_out. This means you can’t have a daemon and a timer on the same object. If you want both, put one of them in another object, possibly a dummy object whose only purpose is to hold the timer/daemon.
* The game author has to define the constant FULL\_DIRECTIONS if they want the game to include the directions NE, SE, SW, NW.
* Inform’s standard veneer routine for printing informative messages for all sorts of runtime errors that can occur is replaced with a simpler routine, saving about 1.5 KB. However, the original routine is used when at least one of the constants DEBUG or RUNTIME\_ERRORS is defined.
* Puny, unlike I6 lib, will not figure out when it should be article “an”. You need to specify it using the article property every time it should be “an”.

# Parser

The parser is to a large extent compatible with Inform, for example wn and NextWord are implemented, and noun/second/inp1/inp2/special\_number/parsed\_number work the same.

General parse routines are supported with the exception of GRP\_REPARSE which isn’t supported. The reason for this is that version 3 games cannot retokenise the input from the reconstructed string.

Note that ParseToken is not available in PunyInform.

# Library extensions

## ext\_flags.h

Flags is a mechanism for keeping track of story progression that is available as part of PunyInform. If you choose to use flags, four procedures with a total size of about 165 bytes are added to the story file. Also, an eight byte array is added to dynamic memory, plus one byte for every eight flags. All in all this is a very efficient way of keeping track of progress.

If you want to use flags, after including globals.h, set the constant FLAG\_COUNT to the number of flags you need, and then include ext\_flags.h.

You then specify a constant for each flag, like this:

Constant F\_FED\_PARROT 0; ! Has the parrot been fed?  
Constant F\_TICKET\_OK 1; ! Has Hildegard booked her plane tickets?  
Constant F\_SAVED\_CAT 2; ! Has the player saved the cat in the tree?

You get the idea – you give each flag a symbolic name so it’s somewhat obvious what it does. Note that the first flag is flag #0, not flag #1.

Setting a flag on or off means calling the routing SetFlag(flag#) or ClearFlag(flag#)

To indicate that the player has saved the cat, call SetFlag(F\_SAVED\_CAT), and to turn off that flag, call ClearFlag(F\_SAVED\_CAT).

Testing a flag is accomplished by calling FlagIsSet or FlagIsClear. So if you have a piece of code that should only be run if the parrot has been fed, you would enclose it in an   
if (FlagIsSet(F\_FED\_PARROT)) { ... }" statement.

Naturally, you can test if a flag is clear by calling FlagIsClear instead.

## ext\_cheap\_scenery.h

This library extension provides a way to implement simple scenery objects which can only be examined, using just a single object for the entire game. This helps keep both the object count and the dynamic memory usage down. For z3 games, which can only hold a total of 255 objects, this is even more important. To use it, include ext\_cheap\_scenery.h after globals.h. Then add a property called cheap\_scenery to the locations where you want to add cheap scenery objects. You can add up to ten cheap scenery objects to one location in this way. For each scenery object, specify, in this order, one adjective, one noun, and one description string or a routine to print one. Instead of an adjective, you may give a synonym to the noun. If no adjective or synonym is needed, use the value 1 in that position.

Note: If you want to use this library extension is a Z-code version 3 game, you must NOT declare cheap\_scenery as a common property, or it will only be able to hold one scenery object instead of ten. For z5 and z8, you can declare it as a common property if you like, or let it be an individual property.

If you want to use the same description for a scenery object in several locations, declare a constant to hold that string, and refer to the constant in each location.

Before including this extension, you can also define a string or routine called SceneryReply. If you do, it will be used whenever the player does something to a scenery object other than examining it. If it's a string, it's printed. If it's a routine it's called. If the routine prints something, it should return true, otherwise false.

Example usage:

[SceneryReply;  
 Push:  
 "Now how would you do that?";  
 default:  
 rfalse;  
];

Include "ext\_cheap\_scenery.h";

Constant SCN\_WATER = "The water is so beautiful this time of year, all clear and glittering.";

[SCN\_SUN;   
 deadflag = 1;  
 "As you stare right into the sun, you feel a burning sensation in your eyes.   
 After a while, all goes black. With no eyesight, you have little hope of  
 completing your investigations.";   
];

Object RiverBank "River Bank"  
 with  
 description "The river is quite wide here. The sun reflects in the blue water, the birds are   
 flying high up above.",  
 cheap\_scenery  
 'blue' 'water' SCN\_WATER  
 'bird' 'birds' "They seem so careless."  
 1 'sun' SCN\_SUN,  
 has light;

# Properties

A property can be used to store a 16-bit value, or an array of values. In z5, a property array can hold up to 32 values. In z3, a property array can only hold 4 values if it’s in a common property but 32 values if it’s in an individual property.

If a property is declared as additive, the values for an object are concatenated with the values of its class, if any, and put into an array.

A property can either be common or individual. Common properties are a little faster to access and use a little less memory than individual properties. A game can use a maximum of 62 common properties. The value of a common property can always be read, but it can only be written if it has been included in the object declaration. If you don’t include it, there is no memory allocated to store a value. If you read the value of such a property, you just get the default value (typically 0).

A common property is created by declaring it with

Property *propertyname*;

To access a property, you write object.*propertyname*, like this:

Dog.description = “The dog looks sleepy.”;  
  
To check if an object has a value for a property (to see if it can be written if it’s a common property or to see if it can be read or written if it’s an individual property, use *provides*:

If(Dog provides description) …

## Limitations for z3

If you want to compile a game to z3 format, this is what you need to keep in mind:

* A game can use no more than 32 attributes and 31 common properties
* When using message passing (like “MyBox.AddWeight(5)” ), no more than one argument may be passed. (In regular Inform, message passing doesn’t work at all in z3.)
* Dynamic object creation and deletion can not be used.

These are the properties defined by the library:

<TBD>