

PunyInform

PunyInform

An Inform library for writing small and fast text adventures.

Version 2.5, 14 May 2021

PunyInform was conceived and designed by Johan Berntsson and Fredrik Ramsberg. Coding by Johan Berntsson, Fredrik Ramsberg, Pablo Martinez and Tomas Öberg. Includes code from the Inform 6 standard library, by Graham Nelson. Thanks to Stefan Vogt, Jason Compton, John Wilson, Hugo Labrande and Richard Fairweather, Adam Sommerfield, auraes and Hannesss for issue reporting, testing, code contributions and promotion. Thanks to David Kinder and Andrew Plotkin for helping out with compiler issues and sharing their deep knowledge of the compiler. Huge thanks to Graham Nelson for creating the Inform 6 compiler and library in the first place.

Contents

1	Introduction	4
2	Comparison with the Inform 6 Standard Library	5
	Getting Started	5
	Actions	6
	The basic actions	6
	OPTIONAL_EXTENDED_VERBSET actions	7
	OPTIONAL_EXTENDED_METAVERBS actions	7
	DEBUG actions	7
	UNDO	7
	Moving objects	7
	Animate objects	7
	Articles	8
	Plural	8
	Capacity	9
	Doors	9
	Simple doors	9
	The with_key property	10
	Daemons and Timers	10
	The reactive attribute	11
	Score/Turns vs Time in the statusline	12
	Turns	12
	Library Messages and Customization	12
	Printing the Contents of an Object	13
	Direction Handling	14
	Fake direction objects.	15
	Disabling directions	15
	Ship directions	15
	Look	16
	Scoring	17
	Box Statements and Menus	17
	Scope	17
	Manual Scope	18

Parser	19
3 Programming Advice	20
Error messages	20
Debugging	20
DebugParseNameObject	21
Customizing the Library	21
Optionals	21
Parameters	24
Abbreviations	24
Limitations for z3	25
Properties	25
4 Extensions	27
cheap_scenery	27
flags	29
menu	29
Extract from DM3	30
quote_box	30
waittime	31
5 Appendix A: List of Routines	33
Library Routines	33
Entry Point Routines	34
PunyInform Entry Point Routines	35
Additional Public Routines	35
PunyInform Public Routines	35
6 Appendix B: List of Properties	36
7 Appendix C: List of Attributes	39
8 Appendix D: List of Variables	41
9 Appendix E: List of Constants	43
10 Appendix F: Grammar	45

Chapter 1

Introduction

PunyInform is a library written in Inform 6 which allows people to create text adventures/interactive fiction using the Z-machine virtual machine.

The main goal of PunyInform is to allow for games which are fast and have a small memory footprint. This should make the games run well on older architectures, such as the 8 bit computers of the 1980s. Our main target is to make it suitable for games on the Commodore 64 using Ozmoo (<https://github.com/johanberntsson/ozmoo>)

PunyInform is based on the Inform 6 standard library, developed by Graham Nelson. In this document DM4 refers to the *Inform Designer's Manual*, 4th edition, which is available online at: <http://www.inform-fiction.org/manual/html/index.html>

A PunyInform game can be compiled to Z-code version 3, 5 or 8 (z3, z5 or z8), but not Glulx. To compile games using PunyInform, you need the official Inform compiler maintained by David Kinder, at <https://github.com/DavidKinder/Inform6>. Binaries can be found at if-archive. Please note that PunyInform uses features that were introduced in Inform v6.34 and using earlier versions of the compiler will cause errors.

Chapter 2

Comparison with the Inform 6 Standard Library

A game written in PunyInform is very similar to a game written with the Inform 6 standard library. However, there are some major differences that are documented in this section.

Getting Started

To compile a game, unpack the files, place the Inform 6.34 compiler binary (Get the source or an executable at <http://www.ifarchive.org/indexes/if-archive/Xinfocom/Xcompilers/Xinform6.html>) in the base directory, and type i.e. `inform6 +lib -v3 -s -e library_of_horror.inf` (type `inform6 -h2` for an explanation of all commandline switches).

You can use the `minimal.inf` file, supplied with PunyInform, as a starting point for developing a new game.

The general pattern of a PunyInform game is:

```
Constant INITIAL_LOCATION_VALUE = ...;

! Change "score" to "time" if you want time on the statusline
Constant STATUSLINE_SCORE; Statusline score;

! define library constants here

Include "globals.h";

! define your own global variables here
```

```

! add extension routines and other library customizations here

Include "puny.h";

! add normal game code here

[Initialise;
    "Welcome to the game!";
];

```

All library constants, including `Story`, `Headline`, `MAX_SCORE`, `OBJECT_SCORE`, `ROOM_SCORE`, `NUMBER_TASKS`, `TASKS_PROVIDED`, `AMUSING_PROVIDED`, `MAX_CARRIED` and `SACK_OBJECT` should be defined before including `globals.h`, if needed. The roles of these constants are documented in DM4.

Library customization, such as supplying an entry point routine such as `PrintTask`, goes between the `globals.h` and `puny.h` inclusions.

After the includes you add game code and an `Initialise` routine, as in other Inform games.

Actions

PunyInform has most of the actions that the standard library has, but they are divided into four sets. The basic set of actions is part of the core library. Then there is a set of normal actions which can be enabled by defining the constant `OPTIONAL_EXTENDED_VERBSET` and a set of meta actions which can be enabled by defining `OPTIONAL_EXTENDED_METAVERBS`. `OPTIONAL_PROVIDE_UNDO` provides the undo verb (z5 and z8 only). Finally, just as in the standard library, there is a set of debug verbs, which can be enabled by defining the symbol `DEBUG`.

The basic actions

Normal actions: `Answer`, `Ask`, `AskTo`, `AskFor`, `Attack`, `Close`, `Consult`, `Cut`, `Dig`, `Disrobe`, `Drink`, `Drop`, `Eat`, `Enter`, `Examine`, `Exit`, `Fill`, `GetOff`, `Give`, `Go`, `Inv`, `Insert`, `Jump`, `JumpOver`, `Listen`, `Lock`, `Look`, `Open`, `Pull`, `Push`, `PushDir`, `PutOn`, `Remove`, `Rub`, `Search`, `Show`, `Smell`, `SwitchOff`, `SwitchOn`, `Take`, `Tie`, `Tell`, `ThrowAt`, `Touch`, `Transfer`, `Turn`, `Unlock`, `Wait`, `Wear`.

Meta actions: `Again`, `FullScore`, `LookModeNormal`, `LookModeLong`, `LookModeShort`, `NotifyOn`, `NotifyOff`, `Oops`, `OopsCorrection`, `Quit`, `Restart`, `Restore`, `Save`, `Score`, `Version`.

OPTIONAL_EXTENDED_VERBSET actions

Normal actions: Blow, Burn, Buy, Empty, EmptyT, GoIn, Kiss, Mild, No, Pray, Set, SetTo, Sing, Sleep, Sorry, Strong, Squeeze, Swim, Swing, Taste, Think, Wake, WakeOther, Wave, WaveHands, Yes.

OPTIONAL_EXTENDED_METAVERBS actions

Meta actions: CommandsOn, CommandsOff, CommandsRead, Places, Objects, ScriptOn, ScriptOff, Verify.

Note: Places and Objects can be disabled by defining the constant NO_PLACES.

DEBUG actions

Meta actions: ActionsOn, ActionsOff, GoNear, Pronouns, Purloin, RandomSeed, RoutinesOn, RoutinesOff, Scope, TimersOn, TimersOff, Tree.

UNDO

Adding OPTIONAL_PROVIDE_UNDO activates the ‘undo’ command, which can be used on interpreter that support undo. Note that undo is not supported in z3 games.

Moving objects

If you move an object into the player’s possession in your own code, like `move Screwdriver to player;`, you should always do `update_moved = true;`. This is necessary to have PunyInform update the `moved` attribute and, when applicable, score points for the object being picked up. If you issue a `##Take` action or some action that does an implicit take, you don’t need to bother with this.

If you have defined `OPTIONAL_MANUAL_SCOPE` and you do something in code which may affect scope (Essentially what the player can see), you need to do `scope_modified = true;`. As a rule of thumb, do this whenever you move an object using `move` or `remove` or you give or remove any of the attributes `open`, `transparent` and `light` for an object. Library routines like `PlayerTo` and all action routines do this when needed.

Animate objects

If you want to allow the player to take certain animate objects, you can add an entry point routine called `DisallowTakeAnimate` and have it return false if noun is one of those objects. I.e:

```
[ DisallowTakeAnimate;  
  if (noun ofclass Puppy) rfalse;
```



```
];
```

Articles

PunyInform, unlike the Inform standard library, will not figure out when an object should have the indefinite article “an”. You need to specify it using the `article` property every time it should be “an”. Example:

```
Object Umbrella "umbrella"
  with
    name 'umbrella',
    article "an";
```

Another difference is that PunyInform doesn’t support the `articles` (note the s) property. This was only added to the Inform library because it’s useful for some languages other than English.

Plural

PunyInform can handle a collection of objects as long as they can be described with full names, but it does not offer support for indistinguishable objects. The library supports `pluralname` and the plural marking on dictionary words with the `//p` suffix.

For example

```
Object -> RedBook "red book"
  with name 'red' 'book' 'books//p';
```

```
Object -> BlueBook "blue book"
  with name 'blue' 'book' 'books//p';
```

can be used like

```
> take book
```

```
Do you mean the red book, or the blue book? > red
```

```
Taken.
```

```
> drop book
```

```
Dropped.
```

```
> take all books
```

```
red book: Taken.
```

```
blue book: Taken.
```

Capacity

The `capacity` property doesn't have a default value in PunyInform. To check the capacity of an object, call `ObjectCapacity(object)`. If the object has a value, it's returned (unless the value is a routine, in which case it is executed and the return value is returned). If the object doesn't have a value for capacity, the value `DEFAULT_CAPACITY` is returned. This value is 100, unless you have defined it to be something else.

Doors

Simple doors

PunyInform supports defining doors just the way it's described in DM4. In addition to this, PunyInform supports a more convenient way to define a door. To enable it, define the constant `OPTIONAL_SIMPLE_DOORS`. This means two new mechanisms come into play:

- If the door object has an array value for `found_in` with exactly two locations, it can leave out the `door_to` property. Instead, the library will assume that if the player is in the first location in the `found_in` array the door leads to the second location in the array and vice versa.
- If the door object has an array value for `found_in` with exactly two locations, it can also have an array value for `door_dir`. The first entry in `door_dir` corresponds to the first entry in `found_in` and the second entry in `door_dir` corresponds to the second entry in `found_in`. Use parenthesis around the values to avoid compiler warnings.

In any single door object, you can use both of these mechanisms, either one of them, or none.

Example of a regular door in PunyInform:

```
Object -> BlueDoor "blue door"
  with
    name 'blue' 'door',
    door_to [;
      if(self in Hallway) return Office;
      return Hallway;
    ],
    door_dir [;
      if(self in Hallway) return n_to;
      return s_to;
    ],
    found_in Hallway Office,
  has static door openable;
```

And this is how to define the same door using OPTIONAL_SIMPLE_DOORS:

```
Object -> BlueDoor "blue door"
  with
    name 'blue' 'door',
    door_dir (n_to) (s_to),
    found_in Hallway Office,
    has static door openable;
```

Note: OPTIONAL_SIMPLE_DOORS adds 86 bytes to the library size, but it saves 22 bytes per door which uses both of the features. So if you use these features for at least four doors, it saves space.

The with_key property

Just as in the standard library, you can use the with_key property to say which key fits the lock for a lockable object. As an alternative to specifying an object as a value, PunyInform allows you to specify a routine. The routine should return false or the object id of the key that fits the lock. When this routine is called, second holds the object currently being considered as a key. This can be used to allow multiple keys fit a lock. Example:

```
Object RedDoor "red door"
  with
    name 'red' 'door',
    with_key [;
      if(second == RedKey or RubyKey or SmallKey) return second;
    ],
    ...
  has static door lockable locked;
```

Daemons and Timers

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.

If you need your daemons/timers to execute in a certain order, you can define the constant OPTIONAL_ORDERED_TIMERS and then set the property timer_order to any number for some or all objects with daemons/timers. A lower number means the daemon/timer will execute earlier. The default value is 100. Note that this number should not be changed while a daemon or timer is running.

The reactive attribute

NOTE: This section deals with an optimization you may want to perform before releasing a game. You can safely skip it while learning PunyInform and return to it when you're close to releasing a game, or not at all.

Each turn, PunyInform needs to check if any of the objects in scope provide any of these properties:

- `react_before`
- `react_after`
- `each_turn`
- `add_to_scope`
- `parse_name` (special, read below!)

These checks are rather time consuming, and they make the game noticeably slower when many objects are in scope, at least on 8-bit machines. To alleviate this problem, PunyInform has an attribute **reactive** which all objects providing any of these properties must have (except possibly `parse_name` - keep reading for details). This allows PunyInform to only check for these properties in objects which have the attribute (checking an attribute is a lot faster than checking a property), so gameplay becomes faster. By default, PunyInform figures out which objects need to have this attribute when the game starts. This process causes a delay of about 0.014 seconds per object on a C64, so up to about 3.5 seconds for a full-size z3 game. If you want the game startup to be quicker and/or you want to save some bytes, you can choose to set the **reactive** attribute manually instead. To do this, define the constant `OPTIONAL_MANUAL_REACTIVE`, compile the game in debug mode and type "DEBUG REACTIVE", and you will get a list of all objects that should have the attribute. Add the **reactive** attribute to each of these objects in the code. If all objects of a certain class should have the attribute, you can of course add the attribute to the class instead.

The `parse_name` property is special. By default, objects should NOT have the **reactive** attribute just because they provide the `parse_name` property. However, if you use `parse_name` rather sparingly, you can speed up parsing by defining the constant `OPTIONAL_REACTIVE_PARSE_NAME`. Of course, if you define this constant *and* `OPTIONAL_MANUAL_REACTIVE`, you must manually give all objects providing `parse_name` the **reactive** attribute. You should only define `OPTIONAL_REACTIVE_PARSE_NAME` if about 10% of your objects or less provide `parse_name`. If you use `parse_name` for lots and lots of objects, the game will be faster without this constant.

Note: `react_before` is an alias of `u_to`, `react_after` is an alias of `d_to`, `add_to_scope` is an alias of `in_to` and `parse_name` is an alias of `out_to`. For this reason, locations which provide `u_to`, `d_to`, `in_to` or `out_to` will be included in the DEBUG REACTIVE report as needing the **reactive** attribute. Unless these locations provide `each_turn`, they actually don't need the attribute. Giving them the attribute makes the game a tiny bit slower in these locations.

Note: The player object (AKA selfobj) provides `each_turn` and `add_to_scope`, both set to `NULL` or `0`, to allow the game author to set it to one of their own routines. If you don't set them to anything (typically done in `Initialise`), there is no need to give the player object the `reactive` attribute. If you do, you will lose it from the debug report, but each turn of the game will also be (very slightly) slower.

Note: If you define `OPTIONAL_MANUAL_REACTIVE` and you have an object which has a `react_before` routine but doesn't have the `reactive` attribute, its `react_before` routine will never be run. Same thing with `react_after`, `each_turn`, `add_to_scope` and `parse_name`.

Score/Turns vs Time in the statusline

Unless you start replacing routines (and avoid the `z3` format) a `PunyInform` game always shows a statusline. You can select between two different types of statusline:

- To show score and turns in the status line, put `Constant STATUSLINE_SCORE;` `Statusline score;` in the beginning of the source.
- To show time in the status line, put `Constant STATUSLINE_TIME;` `Statusline time;` in the beginning of the source, and add a call to `SetTime` in the `initialise` routine (See example below).

```
Constant STATUSLINE_TIME; Statusline time;
Include "globals.h";
Include "puny.h";
[Initialise;
    SetTime(1 * 60 + 5, 5); ! 1:05 am, each turn 5 minutes
];
```

Turns

The global variable `turns` has the value `-1` until the first turn starts. This allows user-supplied code being invoked during initialization and the first room description to act differently based on this.

Library Messages and Customization

All system messages that can be replaced can be found in the file `messages.h`.

`PunyInform` uses two forms of library messages: static strings and complex messages. A typical static string is `"Taken."`. If a message has parts that vary, if the same message should be shared by several different message identifiers, or

a newline should NOT be printed after the message, the message needs to be a complex message. A complex message has its own piece of code to print it.

Each message is defined as either a static string or a complex message in `messages.h`. You replace a message by defining constants and possibly a `LibraryMessages` routine before the inclusion of `puny.h`.

NOTE: A static string message can be replaced by a static string or a complex message, but a complex message can only be replaced by a complex message.

To replace a message with a static string, define a constant with the same name as the message identifier and give it a string value, i.e:

```
Constant MSG_INSERT_NO_ROOM "It's kinda full already, I'm afraid.";
```

To replace a message with a complex message, define a constant with the same name as the message identifier, give it a value in the range 1000-1299 and provide a `LibraryMessages` routine to handle it, i.e:

```
Constant MSG_EXAMINE_NOTHING_SPECIAL 1000;
```

```
[LibraryMessages p_msg p_arg_1 p_arg_2;
    switch(p_msg) {
        MSG_EXAMINE_NOTHING_SPECIAL:
            print_ret (The) noun, " looks perfectly normal in every way.";
    }
];
```

The `LibraryMessages` routine takes three arguments - a message identifier (`p_msg`) and two optional arguments (`p_arg_1` and `p_arg_2`) which a few messages use. The return value of this routine is unimportant.

IMPORTANT: If you have defined a constant to replace a certain error message with a complex message, you *have to* print something for this message.

Printing the Contents of an Object

The standard library provides the routine `WriteListFrom()`. `PunyInform` provides `PrintContents()` instead. While not quite as versatile as `WriteListFrom`, it's meant to be easy to use, easy to remember how to use, and powerful enough to cover the needs for most situations. This is how it works:

```
PrintContents(p_first_text, p_obj, p_check_workflag);
Print what's in/on p_obj recursively.
```

`p_first_text:`

A string containing a message to be printed before printing the first item in/on `p_obj`. Can also be 0 to not print a text, or a routine, which will then be called with `p_obj` as

```

        an argument.
p_obj:
    The container/supporter/person whose contents we want to list.
p_check_workflag:
    If true, only list objects which have the workflag set (this
    rule only applies on the top level.
Return value:
    true if any items were printed, false if not.

Typical usage:

if(PrintContents("On the table you can see ", OakTable)) print ".";

```

Direction Handling

The Compass and the twelve direction objects, as described in DM4, are not available in PunyInform. Instead, there is a single object called `Directions` and two global variables called `selected_direction` and `selected_direction_index`. This helps in keeping the object count down, considering that a z3 game can have no more than 255 objects.

Whenever the player has typed a direction, `noun` is `Directions` and `selected_direction` contains the property number for the direction the player typed. If the player didn't type a direction, `selected_direction` is 0. The name of the Direction object is always the currently selected direction, or "direction" if no direction is selected. So, to implement a robot which will stop the player from going north or east, one might write a `react_before` routine like this:

```

Object Robot "Floyd"
    with
        react_before [;
            Go:
                if(selected_direction == n_to or e_to)
                    "~My mother always told me to avoid going ",
                    (name) Directions, ".~, says Floyd.";
        ],
    has animate;

```

`selected_direction_index` can be used to look up the dictionary words which refer to that direction, the property number and the name of the direction:

```

! prints the short dictionary word, like 'n/'
print (address) abbr_direction_array-->selected_direction_index;
! prints the long dictionary word, like 'north'
print (address) full_direction_array-->selected_direction_index;
! prints the property number, like 7
print direction_properties_array-->selected_direction_index;

```

```
! prints the direction name, like "north"
print (string) direction_name_array-->selected_direction_index;
```

Each of these arrays is a table, so all of them have the number of directions as element 0. The number of directions is also held in the constant `DIRECTION_COUNT`. This is useful if you're writing a library extension and want to iterate over all directions in a safe manner. Please note that the directions are stored in element 1 .. `DIRECTION_COUNT` in these arrays.

Fake direction objects.

For each direction, there is also a fake direction object: `FAKE_N_OBJ`, `FAKE_SW_OBJ`, `FAKE_OUT_OBJ` etc. If you need to generate an action in code which has a direction in it, this requires using the corresponding fake direction object, like this:

```
<<Go FAKE_N_OBJ>>;
<<PushDir Stone FAKE_NW_OBJ>>;
```

If you want to go in a direction and you know the property number for that direction, you can find the corresponding fake direction object by calling `DirPropToFakeObj()`:

```
dir_prop = ne_to; ! Or any direction you like
fake_obj = DirPropToFakeObj(dir_prop);
<<Go fake_obj>>;
```

There is also an inverse of this function, called `FakeObjToDirProp()`, which may come in handy in some situations.

Each fake direction object is just a constant. `PunyInform` recognizes these constants and sets `selected_direction` and `selected_direction_index` properly. As far as we can tell, the only use for the fake direction objects is in actions in code as outlined above.

Disabling directions

If you (perhaps temporarily) don't want the game to recognize any directions, you can set the global variable `normal_directions_enabled` to false.

Ship directions

If you define the constant `OPTIONAL_SHIP_DIRECTIONS`, the parser will recognize 'fore' and 'f' as synonyms for north, 'aft' and 'a' as synonyms for south, 'port' and 'p' as synonyms for west, and 'starboard' and 'sb' as synonyms for east.

If you (temporarily) don't want the game to recognize ship directions, you can set the global variable `ship_directions_enabled` to false.

Look

When performing a Look action, PunyInform, just like the standard library, prints the room name, a newline and then the description of the room, held in the **description** property.

The standard library has a deprecated feature, allowing you to write the text describing the location in the **describe** property of the location object rather than the standard **description** property. This is not supported by PunyInform.

When deciding how to show objects, these are the rules that apply in PunyInform:

- If the object provides **describe**, print or run it. If it's a string, or it's a routine and it returns true, the object will not be described any further. Note that this string or routine should start by printing a newline, unless it's a routine which decides not to print anything at all.
- We will now figure out which the current description property of the object is:
 - If the object is a container or a door, it's **when_open** or **when_closed**, depending on its state.
 - If the object is a switchable object, it's **when_on** or **when_off**, depending on its state.
 - Otherwise, it's **initial**.
- If the object provides this property AND the object hasn't moved or the property is **when_off** or **when_closed**, then print a newline and run or print the string or routine held in the property.
- If, according to the above rules, nothing has yet been printed, include the object in the list of objects printed at the end.
- If **OPTIONAL_PRINT_SCENERY_CONTENTS** has been defined, print what can be seen in/on containers and supporters which have the scenery attribute.

Note: Thanks to aliasing, PunyInform uses only 27 common properties, which is 21 less than the Inform 6 library. This is necessary to support compiling to z3. However, this also means the library can't tell if an object provides **initial**, **when_on** or **when_open** - these are in fact all aliases for the same property. For this reason, the printing rules described above must be a little restrictive. In fact, the Inform Designer's Manual, 4th ed. describes rules which are equally restrictive, since Inform 6 used aliasing as well when the DM4 was released, but newer versions of the Inform 6 library are actually smarter than the DM4 says and will look at which properties are provided and act accordingly. For PunyInform, whenever you have problems getting the results you want using **when_on**, **when_open** etc, write the logic you like in a **describe** routine instead. That way you can make it work exactly the way you want.

Scoring

Scoring works as in DM4, but it is divided into three parts:

- Basic scoring using the `score` variable and the `MAX_SCORE` constant
- Scoring using the `scored` attribute and the `OBJECT_SCORE` and `ROOM_SCORE` constants, enabled by `OPTIONAL_SCORED`
- The `fullscore` verb, enabling the player to see a breakdown of the score, enabled by `OPTIONAL_FULL_SCORE`

If the game has added points in code, like `score = score + 10;`, The “full score” output will also show a line summarizing these points as “Performing noteworthy actions” (this doesn’t happen in the standard library).

If `OPTIONAL_FULL_SCORE` is enabled, you can also choose to define `TASKS_PROVIDED` to enable support for tasks. Read DM4 for details on how to use this.

If constant `NO_SCORE` is defined, no scoring mechanism is included. If it’s a z3 game and the statusline is of type “score”, a score is displayed on the statusline. The interpreter is responsible for showing the statusline in z3, and it can’t be made not to show a value for score, unless it is set to show the time on the statusline. If the constant `NO_SCORE` is defined, the value of this constant (default is 0) is shown as the score on the statusline.

Box Statements and Menus

The box statement is not available in version 3 games, and the usual menu extensions will not work either since version 3 games lack cursor control commands. Instead PunyInform provides an extension that approximates this functionality. See the Extensions section for more detail.

Scope

Scope in PunyInform is a list of things you can interact with. This includes things you can see in the room description, but can also include abstract concepts such as directions and discussion topics. Two library routines enable you to see what’s in scope and what isn’t. The first, `TestScope(obj, actor)`, simply returns true or false according to whether or not `obj` is in scope. The second is `LoopOverScope(routine, actor)` and calls the given routine for each object in scope. In each case the `actor` given is optional, and if it’s omitted, scope is worked out for the player as usual.

The routines `ScopeCeiling`, `LoopOverScope`, `ScopeWithin` and `TestScope` are implemented as described in DM4. Two routines are used to determine if you can touch or see an object: `ObjectIsUntouchable(obj, flag)` and `ObjectIsInvisible(obj, flag)`. Both functions return true if the `obj` is

untouchable or invisible from the player's point of view. If `flag` is true, then the routine never writes anything and only returns true or false to say if the `obj` was untouchable/invisible or not. If `flag` is false, the routine will also write messages like "You can't because ... is in the way." when a problem was found.

The standard Inform parser uses a number of internal scope variables that are not used in PunyInform, including `scope_reason`. Code that relies on these variables has to be rewritten. However, `scope_stage` is supported and is used when the `scope` token is used, so constructs like the code fragment below work as described in DM4.

```
Object questions "questions";
Object -> "apollo"
  with name 'apollo',
        description "Apollo is a Greek god.";

[ QueryTopic;
  switch (scope_stage) {
    1: rfalse;
    2: ScopeWithin(questions); rtrue;
    3: "At the moment, even the simplest questions confuse you.";
  }
];

[ QuerySub; noun.description();];
Verb 'what' * 'is'/'was' scope=QueryTopic -> Query;
```

Manual Scope

Normally, PunyInform updates the scope when a turn starts, before the after routines are run, before the timers and daemons are run, and before each `_turn` is run. To get the best possible performance, you can switch to manual scope updates. You do this by defining the constant `OPTIONAL_MANUAL_SCOPE`. With manual scope enabled, scope is only updated when the program signals that an update may be needed. You signal this by setting the variable `scope_modified` to `true`. A simple rule is to do this anytime you use `move` or `remove` or you alter any of the attributes `open`, `transparent`, `light`. This is already in place in the `PlayerTo` routine as well as in the action routines for `Open`, `Close` etc. Sample usage:

```
Object Teleporter "teleporter"
  with
    name 'teleporter',
    capacity 1,
    before [ c;
      SwitchOn:
        c = child(self);
```

```

        if(c ~= 0) {
            move c to SecretChamber;
            scope_modified = true;
            print_ret (The) c, " disappears!";
        }
    ],
    has container openable transparent;

```

Parser

The parser is to a large extent compatible with Inform, for example `wn`, `NextWord()` and `NextWordStopped()` 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.

Chapter 3

Programming Advice

Error messages

The Inform standard veneer routine for printing informative messages for all sorts of runtime errors that can occur is replaced with a simpler routine in PunyInform, saving about 1.5 KB. However, the original routine is used if the constant `RUNTIME_ERRORS` is set to 2.

Debugging

By defining the constant `DEBUG` (or adding `-D` to the inform 6 compiler commandline), the game is compiled in debug mode. This means a number of meta verbs are available for inspecting the game world and examining which routines and actions are executed. These are the debug verbs supplied:

TREE : Show the object tree for the current location. *TREE [object]* : Show the object tree for this object.

GONEAR [object] : Teleport to the location of the object.

SCOPE [actor] : List the objects which are currently in scope for the actor. Actor defaults to player.

PRONOUNS : List what he, she, it and them are currently referring to.

RANDOM [number] : Seed the pseudo-random number generator, to make randomization predictable. Number defaults to 100.

PURLOIN [object] : Teleport the object into your inventory, no matter where it is.

ROUTINES [on/off] : Show which routines are being executed.

ACTIONS [on/off] : Show which actions are being invoked.

TIMERS [on/off] : Show which timers and daemons are being executed.

DEBUG REACTIVE : Show which object have the reactive attribute but shouldn't and which don't have it but probably should. See The reactive attribute.

DebugParseNameObject

Some debug verbs take an object or an actor as an argument. The scope for these verbs are unlimited - they can refer to objects which are in a different location or even in no location. It can be hard or even impossible for the parser to decide if an object which doesn't have a parent is a room or a normal object. This causes problems when an object has a parse_name routine. If you have problems referring to an object which has a parse_name routine and may lack a parent in the object tree, you can create a routine called DebugParseNameObject to help out. It should return true for every such object and false for other objects. It's okay if it returns true for other normal objects, but it must never return true for a room. Example implementation which returns true for the three objects in the list and false for all other objects:

```
#Ifdef DEBUG;
[ DebugParseNameObject p_obj;
  if(p_obj ~= RecordPlayer or Record or Robot) rfalse;
];
#Endif;
```

Customizing the Library

PunyInform is designed to be as small as possible to run well on old computers, and some features that add to the size have been made optional. If you want to enable these features, add a line like "Constant OPTIONAL_GUESS_MISSING_NOUN;" before including globals.h, but keep in mind that it will make the game larger. You can also change some parameters in the library from their default values to further adjust the library size as needed. Finally you can use abbreviations to reduce the game size further. PunyInform includes a set of standard abbreviations which can be enabled as needed.

These customizations are described in detail in the following sections.

Optionals

The optional parts of PunyLib can be enabled with these constants:

Option	Bytes	Comment
DEBUG	4068	enable some debugging verbs for game development. These include ‘scope’, ‘random’, ‘pronouns’, ‘tree’, ‘purloin’, ‘gonear’, ‘routines’, ‘actions’ and ‘timers’/‘daemons’.
NO_SCORE	-310	don’t include any code for keeping track of score. The value of the constant is shown as the score on the statusline in z3.
OPTIONAL_ALLOW_ WRITTEN_NUMBERS	300	to be able to parse “one”, “two” etc as numbers.
OPTIONAL_ EXTENDED_ METAVERBS	940	add a set of less important, but nice to have, meta verbs to the grammar.
OPTIONAL_ EXTENDED_ VERBSET	2152	add a set of less important, but nice to have, verbs to the grammar.
OPTIONAL_FULL_ DIRECTIONS	112	Include directions NW, SW, NE and SE. Including them also makes the parsing process slightly slower in z3 mode.
OPTIONAL_FULL_ SCORE	258	adds the fullscore verb, and optional support for tasks as described in DM4. Size grows by another 78 if OPTIONAL_SCORED is defined.
OPTIONAL_GUESS_ MISSING_NOUN	290	add code to guess missing parts of an incomplete input, such as a door when typing only ‘open’, and accepting the input with a “(assuming the wooden door)” message.
OPTIONAL_MANUAL_ REACTIVE	-44	leave it to the author to set the reactive attribute, for faster game start. See The reactive attribute for instructions on how to use it.

Option	Bytes	Comment
OPTIONAL__MANUAL__ SCOPE	12	let the game code say when scope needs to be updated, for better performance. See Manual Scope for instructions on how to use it.
OPTIONAL__NO__ DARKNESS	-360	skip support for light and darkness - there is always light everywhere. Unlike the other optionals, this one makes the game smaller.
OPTIONAL__ORDERED__ TIMERS	104	lets you assign an order number (property <code>timer_order</code> , default = 100) to each timer or daemon, defining the order of execution - low numbers execute early.
OPTIONAL__PRINT__ SCENERY__CONTENTS	80	Have Look show what is in/on containers and supporters which have the scenery attribute.
OPTIONAL__PROVIDE__ UNDO	212	Adds undo functionality (z5 and z8 only).
OPTIONAL__REACTIVE__ PARSE_NAME	8	let object which provide <code>parse_name</code> have the <code>reactive</code> attribute. See The reactive attribute for instructions on how to use it.
OPTIONAL__SCORED	28	adds the scored attribute as described in DM4.
OPTIONAL__SIMPLE__ DOORS	86	Allow for a simpler way of defining doors. This also ends up saving space if you have more than three doors that use this mechanism. See Doors for instructions on how to use it.
OPTIONAL__SHIP__ DIRECTIONS	116	Add fore, aft, port and starboard as directions. See Ship Directions for instructions on how to use it.

Parameters

The parameters listed in the table below can be adjusted in a game by redefining them before `globals.h` is included.

Parameter	Default	Comment
MAX_CARRIED	32	Max. number of items the user can carry at once
MAX_WHICH_OBJECTS	10	Max. number to include in a “which X do you mean?” parser question
MAX_MULTIPLE_OBJECTS	32	Max. number of objects that match “all” in an input such as “get all”
MAX_INPUT_CHARS	78	Max. number of characters in one line of input from the player
MAX_INPUT_WORDS	20	Max. number of words in a parsed sentence
MAX_FLOATING_OBJECTS	32	Max. number of floating objects
MAX_TIMERS	32	Max. number of timers/daemons running at once
MAX_SCOPE	32	Max. number of objects to consider when calculating the scope of the player
RUNTIME_ERRORS	1 or 2	Runtime error reporting: 0 = minimum, 1 = report all errors using error codes, 2 = report all errors using error messages. Default is 2 in DEBUG mode, and 1 when not in DEBUG mode.

Abbreviations

PunyInform can use a set of standard abbreviations to make strings more compact. If you want to provide your own abbreviations, define the constant `CUSTOM_ABBREVIATIONS` in your game. Keep in mind that you need to compile with the “-e” flag to make the compiler use abbreviations.

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 30 common properties. PunyInform defines 27 common properties.
- A game can use no more than 32 attributes. PunyInform defines 29 attributes (+1 if `OPTIONAL_SCORED` is defined, -1 if `OPTIONAL_NO_DARKNESS` is defined).
- Arrays in common properties can only hold four values. Arrays in individual properties however, can hold 32 values.
- 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.)
- Routine calls can have no more than three arguments.
- Dynamic object creation and deletion can not be used.
- If you need more than four names for an object in a z3 game, give it a `parse_name` routine.
- The room name printed on the statusline is always the object name string. It can’t be overridden with `short_name` in a class or in the same object. Read below for a possible workaround.
- It is not possible to support the ‘undo’ verb.
- The interpreter is responsible for displaying the statusline, and it will show a score even if you define `NO_SCORE`.

IMPORTANT: In the Inform compiler, version 6.34, there is a bug which corrupts objects that have common property arrays of length > 4 in z3 games. The compiler prints a warning and keeps compiling, but the object is broken. This has been fixed in v6.35.

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, typically like “box”. This behavior in z3 games is part of the Z-machine specification so it’s nothing that the game or the library can change. 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;
```

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 is in a common property but 32 values if it is 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 z5 or z8 game can use a maximum of 62 common properties, while a z3 game can use a maximum of 30 common properties. PunyInform uses 27 common properties, so if you're building a z3 game, you can only add three 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 is a common property or to see if it can be read or written if it is an individual property), use *provides*:

```
If(Dog provides description) ...
```

Chapter 4

Extensions

PunyInform keeps the library code size down by only providing the most fundamental functionality by default, but ships with several extensions which can easily be added to games.

cheap__scenery

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 in 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 is a string, it is printed. If it is a routine it is called. If the routine prints something, it should return true,

otherwise false. The routine is called with two parameters - the words which are listed in the `cheap_scenery` property as referring to the object that was matched. Note that this may not be exactly what the player typed, i.e. the player may have typed “examine water” but the words listen in the property are ‘blue’ ‘water’. In this case, the first parameter will be ‘blue’ and the second ‘water’.

If constant `DEBUG` is defined, the extension will complain about programming mistakes it finds in the `cheap_scenery` data in rooms. Without `DEBUG`, it will keep silent.

Note: If you include this extension, you must either declare `cheap_scenery` as a common property, or use it as an individual property in at least one object, or you will get a compilation error (No such constant as “cheap_scenery”).

Example usage:

```
[SceneryReply word1 word2;
Push:
    if(word1 == 'blue' && word2 == 'water') "If you mean you want to swim, just say so!";
    "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;
```

flags

Flags is a mechanism for keeping track of story progression. 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 static memory, and one byte is added to dynamic memory for every eight flags. All in all this is a very memory-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 routine `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.

menu

This is an extension to let games show a menu of text options (for instance, when producing instructions which have several topics, or when giving clues). This can be done with the `DoMenu` routine, which is very similar to the `DoMenu` in the standard Inform library. In version 3 mode it will create a simple text version instead because of technical limitations.

A common way of using `DoMenu` is from a “help” verb, which can be declared like so:

```
Include "ext_menu.h";

! add HelpItem, HelpMenu and HelpInfo here

[ HelpSub;
    DoMenu(HelpItems, HelpMenu, HelpInfo);
```

```
];
```

```
Verb 'help' * -> Help;
```

Below is how DoMenu was described in the *Inform Designer's Manual, 3rd edition*.

Extract from DM3

Here is a typical call to DoMenu:

```
DoMenu("There is information provided on the following:~  
      ^      Instructions for playing  
      ^      The history of this game  
      ^      Credits~",HelpMenu, HelpInfo);
```

Note the layout, and especially the carriage returns.

The second and third arguments are themselves routines. (Actually the first argument can also be a routine to print a string instead of the string itself, which might be useful for adaptive hints.) The HelpMenu routine is supposed to look at the variable menu_item. In the case when this is zero, it should return the number of entries in the menu (3 in the example). In any case it should set item_name to the title for the page of information for that item; and item_width to half its length in characters (this is used to centre titles on the screen). In the case of item 0, the title should be that for the whole menu.

The second routine, HelpInfo above, should simply look at menu_item (1 to 3 above) and print the text for that selection. After this returns, normally the game prints "Press [Space] to return to menu" but if the value 2 is returned it doesn't wait, and if the value 3 is returned it automatically quits the menu as if Q had been pressed. This is useful for juggling submenus about. Menu items can safely launch whole new menus, and it is easy to make a tree of these (which will be needed when it comes to providing hints across any size of game).

quote__box

This is an extension to let games show a simple quote box. For z5+ games, the extension will try to center the quote box on the screen, by reading the screen width reported by the interpreter in the header.

For z3, this information is not available. Instead, it can do it two ways: 1. The game programmer tells the extension to assume the screen has a certain width and the extension uses this information to center the quote box. 2. The game programmer tells the extension to just indent the quote box a fixed number of characters.

To use (1), set the constant QUOTE_V3_SCREEN_WIDTH to the desired width, which has to be > 6.

To use (2), set the constant QUOTE_V3_SCREEN_WIDTH to the desired number of characters to indent by, which must be in the range 0-6.

By default, method (2) will be used, with 2 characters of indentation.

To display a quote box, create a word array holding the number of lines, the number of characters in the longest line, and then a string per line, and call QuoteBox with the array name as the argument.

```
Include "ext_quote_box.h";

Array quote_1 --> 5 35
"When I die, I want to go peacefully"
"in my sleep like my grandfather."
"Not screaming in terror, like the"
"passengers in his car."
"          -- Jack Handey";
!
[AnyRoutine;
  QuoteBox(quote_1);
];
```

waittime

This extension gives players an extended Wait command, which can be used to wait a certain number of turns, minutes or hours, or to wait until a certain time of day is reached.

In a game showing time on the statusline, the player can use commands such as:

```
wait for 5 minutes wait 1 hour wait until 1:20 wait till three o'clock
wait till quarter to five wait till 5 am wait 3 turns/moves
```

A turn or a move may be the same as a minute, depending on the time scale (how many minutes the clock is advanced per turn, or how many moves it takes before the clock is advanced one minute).

Note: Using words for numbers requires OPTIONAL_ALLOW_WRITTEN_NUMBERS.

While the player is waiting, the global variable waittime_waiting has the value true. A daemon or each_turn routine may show an event which could make the player want to abort the waiting and spring to action. If this happens, set waittime_waiting to false.

If game time is suddenly changed, typically using a SetTime() call, it is a good idea to abort any ongoing waiting.

This extension also includes the parse routine **parsetime** to parse times of day, like "1:20", "quarter to five", "3:10 pm" etc, which can also be used for other

verbs which need this, like setting a watch or clock. The parsed time (in minutes after midnight) comes in **noun** or **second**.

In a game showing score/turns on the statusline, the commands to wait for a certain number of turns, minutes or hours still work. A minute is considered the same as a turn. The command to wait until a certain time of day is not available, and neither is the parsetime routine.

This extension must be included after including “puny.h”. Before including it, you may define the constants MAX_WAIT_MINUES and MAX_WAIT_MOVES to say how long the player is allowed to wait for using a single command.

Chapter 5

Appendix A: List of Routines

PunyInform defines both public and private routines. The private routines are prefixed with an underscore (for example, `_ParsePattern`) and should not be used by a game developer. The public routines do not have this prefix, and are for general use. Most of the public routines work the same, or in a very similar manner, to corresponding routines in DM4, but PunyInform also offers a few extra routines not available in Inform. All public routines are listed below in this section.

Library Routines

These library routines are supported by PunyInform, as described in DM4.

Library Routine	Comment
Banner	
CommonAncestor	
DrawStatusLine	Not available in version 3 games
IndirectlyContains	
LoopOverScope	
NextWord	
NextWordStopped	
NumberWord	
ObjectIsUntouchable	
PlayerTo	
ParseToken	
PlaceInScope	
PronounNotice	

Library Routine	Comment
SetTime	
ScopeWithin	
TestScope	
TryNumber	
WordAddress	
WordLength	
YesOrNo	

Entry Point Routines

These entry point routines are supported by PunyInform, as described in the DM4.

Entry Point Routine	Comment
AfterLife	
AfterPrompt	
Amusing	
BeforeParsing	
DarkToDark	
DeathMessage	
GamePostRoutine	
GamePreRoutine	
Initialise	Mandatory.
InScope	The et_flag isn't supported.
LookRoutine	
NewRoom	
ParseNumber	
PrintRank	OPTIONAL_FULL_SCORE
PrintTaskName	OPTIONAL_FULL_SCORE + TASKS_PROVIDED
PrintVerb	
TimePasses	
UnknownVerb	

These entry point routines are not supported

Entry Point Routine	Comment
ChooseObjects	The parser internals differ too much
ParseNoun	
ParserError	The parser internals differ too much

PunyInform Entry Point Routines

These entry point routines are supported by PunyInform, but not by the standard library.

Entry Point Routine	Comment
DebugParseNameObject	See DebugParseNameObject
DisallowTakeAnimate	See Animate Objects
LibraryMessages	See Library Messages and Customization

Additional Public Routines

These routines are supported by both the standard library and PunyInform, but are not documented in DM4.

Routine Name	Comment
PrintOrRun	
RunRoutines	
CTheyreorThats	Print directive
ItorThem	Print directive
IsOrAre	Print directive

PunyInform Public Routines

These public routines are provided by PunyInform, but not by the standard library.

Routine Name	Comment
OnOff	Print directive
ObjectCapacity	See Capacity
ObjectIsInvisible	Similar to ObjectIsUntouchable (DM4)
PrintContents	
PrintMsg	
RunTimeError	

Chapter 6

Appendix B: List of Properties

These are the properties defined by the library:

Property	Read more below
add_to_scope	
after	
article	
before	
cant_go	
capacity	Y
d_to	
daemon	
describe	
description	
door_dir	
door_to	
each_turn	
e_to	
found_in	
in_to	
initial	
inside_description	
invent	
life	
n_to	
name	
ne_to	

Property	Read more below
number	
nw_to	
orders	
out_to	
parse_name	Y
react_after	
react_before	
s_to	
se_to	
short_name	
sw_to	
time_left	
time_out	
timer_order	Y
u_to	
w_to	
when_closed	
when_off	
when_on	
when_open	
with_key	Y

The properties `articles`, `grammar`, `list_together`, `plural` and `short_name_indef`, which are supported by the Inform 6 library, are not supported by PunyInform.

- The `capacity` property doesn't have a default value in PunyInform. To check the capacity of an object, call `ObjectCapacity(object)`. If the object has a value, it's returned (unless the value is a routine, in which case it is executed and the return value is returned). If the object doesn't have a value for capacity, the value `DEFAULT_CAPACITY` is returned. This value is 100, unless you have defined it to be something else.
- The `parse_name` property works as described in DM4 except that, since PunyInform doesn't support identical objects, it is never called to check whether or not two objects which share the same `parse_name` routine are identical.
- `timer_order` is only used if `OPTIONAL_ORDERED_TIMERS` is defined. It is an individual property, unless you declare it as a common property using `Property timer_order;`. Read more about order timers under Daemons and timers.
- The `with_key` property can also hold a routine. The routine should return false or the object id of the key that fits the lock. When this routine is called, `second` holds the object currently being considered as a key. This

can be used to allow multiple keys fit a lock.

Chapter 7

Appendix C: List of Attributes

These attributes are the same as in DM4.

Attribute	Read more below
absent	
animate	
clothing	
concealed	
container	
door	
edible	
enterable	
female	
general	
light	Y
lockable	
locked	
moved	Y
neuter	
on	
open	
openable	
pluralname	
proper	
scored	Y
scenery	
static	
supporter	

Attribute	Read more below
switchable	
talkable	
transparent	
visited	
workflag	
worn	

- `light` is not defined if `OPTIONAL_NO_DARKNESS` is defined.
- `scored` is only defined if `OPTIONAL_SCORED` is defined.
- For `moved` to be updated and `scored` to be considered, you need to set `update_moved` to true whenever moving objects into the player's possession in code. See Moving objects.

These attributes are used in the Inform standard library and are listed in DM4, but are not used in PunyInform.

Attribute	Comment
male	not needed, assumed if an object is animate and it is not female or neuter

These attributes are used in PunyInform but not in the Inform standard library.

Attribute	Comment
reactive	See The reactive attribute for instructions

Chapter 8

Appendix D: List of Variables

These variables are the same as in DM4.

Variable
action
actor
consult_from
consult_words
deadflag
herobj
himobj
inp1
inp2
inventory_stage
itobj
keep_silent
location
lookmode
num_words
parsed_number
parser_action
real_location
scope_stage
score
second
special_number
verb_word
verb_wordnum

Variable
wn

These variables are PunyInform only.

Variable

These variables are used in the Inform standard library and are listed in DM4, but are not used in PunyInform.

Variable
c_style
et_flag
listing_together
lm_n
lm_o
notify_mode
parser_one
parser_two
scope_reason
standard_interpreter
the_time
vague_object

Chapter 9

Appendix E: List of Constants

These constants are the same as in DM4.

Constant Name
AMUSING_PROVIDED
GPR_FAIL
GPR_MULTIPLE
GPR_NUMBER
GPR_PREPOSITION
GPR_REPARSE
Headline
MAX_CARRIED
MAX_SCORE
MAX_TIMERS
NUMBER_TASKS
OBJECT_SCORE
ROOM_SCORE
SACK_OBJECT
Story
TASKS_PROVIDED

These constants are used in the Inform standard library and are listed in DM4, but are not used in PunyInform. Most of them are parser specific for the standard lib, and the PunyInform parser works differently.

Constant Name
ANIMA_PE
ASKSCOPE_PE
CANTSEE_PE
DEATH_MENTION_UNDO
EACHTURN_REASON
ELEMENTARY_TT
EXCEPT_PE
ITGONE_PE
JUNKAFTER_PE
LOOPOVERSCOPE_REASON
MMULTI_PE
MULTI_PE
NO_PLACES
NOTHELD_PE
NOTHING_PE
NUMBER_PE
PARSING_REASON
REACT_AFTER_REASON
REACT_BEFORE_REASON
SCENERY_PE
SCOPE_TT
STUCK_PE
TALKING_REASON
TESTSCOPE_REASON
TOOFEW_PE
TOOLIT_PE
UPTO_PE
USE_MODULES
VAGUE_PE
VERB_PE

Chapter 10

Appendix F: Grammar

Here are the standard verbs defined in the library.

Verbs

answer say shout speak
ask
attack break crack destroy
climb scale
close cover shut
cut chop prune slice
dig
drink sip swallow
drop discard throw
eat
enter cross
examine x
exit out outside
fill
get
give feed offer pay
go run walk
insert
inventory inv i
jump hop skip
leave
listen hear
lock
look l
open uncover unwrap
pick

Verbs
pull drag
push clear move press shift
put
read
remove
rub clean dust polish scrub
search
shed disrobe doff
show display present
sit lie
smell sniff
stand
switch
take carry hold
tell
tie attach fasten fix
touch feel fondle grope
turn rotate screw twist unscrew
unlock
wait z
wear don

This set of extended verbs are not included by default, but can be added by defining `OPTIONAL_EXTENDED_VERBSET`.

Verbs	Comment
blow	OPTIONAL_EXTENDED_VERBSET
bother curses darn drat	OPTIONAL_EXTENDED_VERBSET
burn light	OPTIONAL_EXTENDED_VERBSET
buy purchase	OPTIONAL_EXTENDED_VERBSET
consult	OPTIONAL_EXTENDED_VERBSET
empty	OPTIONAL_EXTENDED_VERBSET
in inside	OPTIONAL_EXTENDED_VERBSET
kiss embrace hug	OPTIONAL_EXTENDED_VERBSET
no	OPTIONAL_EXTENDED_VERBSET
peel	OPTIONAL_EXTENDED_VERBSET
pray	OPTIONAL_EXTENDED_VERBSET
pry prise prize lever jemmy force	OPTIONAL_EXTENDED_VERBSET
set adjust	OPTIONAL_EXTENDED_VERBSET
shit damn fuck sod	OPTIONAL_EXTENDED_VERBSET
sing	OPTIONAL_EXTENDED_VERBSET
sleep nap	OPTIONAL_EXTENDED_VERBSET
sorry	OPTIONAL_EXTENDED_VERBSET

Verbs	Comment
squeeze squash	OPTIONAL_EXTENDED_VERBSET
swim dive	OPTIONAL_EXTENDED_VERBSET
swing	OPTIONAL_EXTENDED_VERBSET
taste	OPTIONAL_EXTENDED_VERBSET
think	OPTIONAL_EXTENDED_VERBSET
transfer	OPTIONAL_EXTENDED_VERBSET
wake awake awaken	OPTIONAL_EXTENDED_VERBSET
wave	OPTIONAL_EXTENDED_VERBSET
yes y	OPTIONAL_EXTENDED_VERBSET

This set of PunyInform debug verbs are not included by default, but can be added by defining DEBUG.

Verbs	Comment
actions	DEBUG
gonear	DEBUG
pronouns nouns	DEBUG
purlain	DEBUG
random	DEBUG
routines messages	DEBUG
scope	DEBUG
timers daemons	DEBUG
tree	DEBUG

These debug verbs defined in the library are not supported by PunyInform.

Verbs	Comment
abstract	not in PunyInform
changes	not in PunyInform
goto	not in PunyInform
showobj	not in PunyInform
showverb	not in PunyInform
trace	not in PunyInform

These are the meta verbs. Some are only included when OPTIONAL_EXTENDED_METAVERBS is defined, and some are not defined if NO_PLACES is defined.

Verbs	Comment
brief normal	
fullscore full	

Verbs	Comment
noscript unscript	OPTIONAL_EXTENDED_METAVERBS
notify	
objects	OPTIONAL_EXTENDED_METAVERBS and not NO_PLACES
places	OPTIONAL_EXTENDED_METAVERBS and not NO_PLACES
quit q die	
recording	OPTIONAL_EXTENDED_METAVERBS
replay	OPTIONAL_EXTENDED_METAVERBS
restart	
restore	
save	
score	
script transcript	OPTIONAL_EXTENDED_METAVERBS
superbrief short	
verify	OPTIONAL_EXTENDED_METAVERBS
verbose long	
version	