PunyInform Technical Report

PunyInform

A fast and compact library for writing text adventure games for the Z-machine running on 8-bit computers as well as other platforms.

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, Richard Fairweather, Adam Sommerfield, auraes and Hannesss for issue reporting, advice, 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

Chapter 1

Introduction

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 availble online at: http://www.inform-fiction.org/manual/html/index.html

The PunyInform parser is to a large extent compatible with Inform 6, for example wn, NextWord() and NextWordStopped() are implemented, and noun/second/inp1/inp2/special_number/parsed_number work the same. However, the internals are completely different, and this document gives an overview of how the code works.

1.1 Program and Data Structures

PunyLib starts executing from the main routine in lib/puny.h, which contains the game loop. To support and implement the game these additional blocks are used:

- Parser, implemented in lib/parser.h
- Grammar, implemented in lib/grammar.h
- Messages, implemented in lib/messages.h
- Scope, implemented in lib/scope.h

The structure and main routines of these blocks are described in separate chapters below.

Chapter 2

Parser

2.1 Overview

The game loop, implemented in the `main` routine lib/puny.h, controls the execution of the game. In this loop the player input is read by a call to `_ReadPlayerInput`. The parser entry routine `_ParseAndPerformAction` is then called, which determines which verb the input uses, tries all patterns found in the grammar for this verb, using `_ParsePattern`, and executes the best pattern found. If no good pattern found it will instead write an error message, such as "I don't understand that sentence."

`_ParsePattern` loops over the pattern, calling `_ParseToken` for every token, and handles errors such as running out of either user or pattern data prematurely.

`_ParseToken` in its turn uses `_GetNextNoun` when detecting a noun-related token (`NOUN_OBJECT`, `CREATURE_OBJECT`, `HELD_OBJECT`, `MULTI*_OBJECT`) to parse a noun phrase. It also handles preposition handling.

`_GetNextNoun` relies on `_CheckNoun` to try parsing a noun phrase, but adds checks for pronouns and plurals, and disambiguates if needed.

`_CheckNoun` checks if objects in scope match words in the input string, either by using its `name` property or its `parse_name` routine. More than one object could match if the input is incomplete, which is reported back using the `which_object` global variable, and used in disambiguation.

2.2 State Variables

These are the most important state variables in the parser

- noun: as described in DM4
- second: as described in DM4

- action: as described in DM4
- actor: as described in DM4
- consult_word: as described in DM4
- consult_from: as described in DM4
- special_word: as described in DM4
- special_number: as described in DM4
- which_object: this array holds the objects that matched the currently parsed noun phrase
- multiple_objects: this array holds the object or objects that matched a `MULTI*_OBJECT` token
- parser_all_found: this is true if "all" was parsed, such as "get all".
- parser_check_multiple: this is set to the `MULTI*_OBJECT` token being parsed
- parser_all_except_object: this is set to Y when parsing "take all Xs except Y" or "take all but Y", so it can be skipped when executing the action.

2.3   Handling Errors

The parser uses a two phase approach to matching grammar templates to user input. In phase 1 the parser is testing several patterns to give them a score on how well they match the input. In this phase no messages are printed.

The best candidate then enters phase 2, where the same tests are run, but in this phase error messages are printed. If any errors are found and printed in phase 2, then the parser gives up (since the best candidate failed) and returns control to the game loop, which will prompt new user input.

There is an optimisation that allows phase 2 to be skipped when the pattern fit perfectly, and all processing has already been done in phase 1. This is controlled by the `phase2_necessary` variable, which is set during phase 1 to either `PHASE2_SUCCESS`, `PHASE2_ERROR` or `PHASE2_DISAMBIGUATION`. `PHASE2_ERROR` indicates that phase 1 failed silently for this pattern, but that it will produce a message if run again in phase 2.

2.4   Handling Disambiguation

When the pattern matches a single noun token such as NOUN_OBJECT with more than one object then the parser will indicate this by setting `phase2_necessary` to `PHASE2_DISAMBIGUATION` in phase 1. If the same pattern is run in phase 2 then the parser will ask the user to disambiguate ("Do you mean the blue book or the red book?"). If the user input reduces the options to a single object it will be used for the noun phrase, otherwise the parser will print an error message.

2.5 Main Routines

2.5.1 `_ParseAndPerformAction`

`_ParseAndPerformAction` uses the global variables `buffer` and `parse`. `buffer` contains the input string, while `parse` is a list of tokens, where each token contains a pointer to the word string in `buffer` and a pointer to the dictionary word. The routine returns the negative number of words when the input could be parsed successfully (so -2 if two words parsed), or true if the parser failed to parse the input, and the user needs to add new input.

`_ParseAndPerformAction` firsts determines which verb the input uses. It then checks each pattern to see if it matches the input. This is done by calling `_ParsePattern` which takes p_pattern (the current pattern to check) and p_phase (set to 1). `_ParsePattern` returns a score that indicates how well the pattern matches the input. During phase 1 this match will be done silently, so no error messages are printed even if the pattern fails to match the input.

The score is 100 if a perfect match was found, or the number of words matches by the pattern if it failed to match. This means that only part of the pattern matched, or the match silently failed during phase 1. This could happen because for example "examine dog" was matched when no dog was in scope (visible). To distinguish between these options `_ParsePattern` will set the `phase2_necessary` flag to `PHASE2_ERROR` when an error was found.

`_ParseAndPerformAction` keeps track of the score for each pattern, and if a perfect score of 100 is found then it it will set up the PunyInform variables for `noun`, `second`, `action` etc as described in the PunyInform documentation, and then run the implementation routine for the grammar line. For example, if the input was "examine me" the `ExamineSub` routine will be activated with `noun` set to the player object.

However, if no perfect score was found then the highest score is used. If `phase2_necessary` is set to PHASE2_ERROR, then `_ParsePattern` will be called again with this pattern and `p_phase` set to 2. In phase 2 `_ParsePattern` will print the error messages that were surpressed during phase 1. If the match failed and `phase2_necessary` wasn't set if means that there wasn't enough input to match the pattern, and a message such as "I think you wanted to say 'climb something'. Please try again" is printed.

Another possibility is that the pattern seems to match but the noun phrase is ambigus. In this case the pattern returns a score as if the pattern matched the noun phrase, but sets `phase2_necessary` to PHASE2_DISAMBIGUATION, and - like for PHASE2_ERROR - `_ParsePattern` is called again with `p_phase` set to 2. _ParsePattern/_GetNextNoun will then call `_AskWhichNoun` to prompt the user for additional information ("Do you mean the blue or the red book?"), and return 100 if the noun phrase is successfully parsed.

2.5.2 `_ParsePattern`

`_ParsePattern` takes a pattern and the current phase, and returns a score that indicated how many words could be successfully parsed using the pattern. If the whole pattern could be parsed, then 100 is returned, and -1 is returned if the input needs to be reparsed. This can happen if the player was asked to disambiguate, but instead of adding to the noun phrase a completely new command as given.

`_ParsePattern` calls `_ParseToken` for each token in the pattern, until either the complete pattern has been parsed, or it runs out of user input to parse against. It can also return early if `_ParseToken` returns `GPR_FAIL`.

Since patterns can contains a list of acceptable prepositions the routine needs to skip ahead until the end of the preposition list if `_ParseToken` managed to match a preposition. The routine will also not return early in case `_ParseToken` failed to parse on of the preposition alternatives.

If `_ParseToken` successfully parsed one of the noun token types, then `_UpdateNounSecond` is called to update the `noun` and `second` parser state variables.

`_ParsePattern` also detects bad input (words that are not in the dictionary) and prints error messages if in phase 2.

2.5.3 `_ParseToken`

The format of `_ParseToken` is the same as `ParseToken` for Inform 6 compatibility. `ParseToken` takes two arguments, token type and token data, and returns the object number or a failure code (`GPR_FAIL`, `GPR_MULTIPLE`, `GPR_NUMBER`, `GPR_REPARSE` or `GPR_PREPOSITION`). In addition, `_ParseToken` also takes p_phase to indicate the current phase.

`_ParseToken` handles each token type differently. If it is a preposition, then it checks if the current word in the input is a match, and returns `GPR_PREPOSITION`. If not, it returns `GPR_FAIL`. It handles topics and numbers in a similar way, using `_ParseTopic` and `TryNumber` to update `consult_from`, `consult_words`, `parsed_number`, and `special_word` as needed.

Nouns are more complicated. If the expected token is a single noun, then `_GetNextNoun` is called and the object is returned. Before returning it makes sure that `CREATURE_OBJECT` only matches something animate, and `HELD_OBJECT` tries to pick up objects if not carried by the player.

However, if the expected token is a `MULTI*_OBJECT` type, then the routine calls `_GetNextNoun` and the object into the `multiple_objects` array. However, if `_getNextNoun` has detected a plural noun, then `which_object` holds all objects that partially matches ("books") and these objects are copied into `multiple_objects` instead. It is also possible that it is a single `all`, in which case `_AddMultipleNouns` is called to fill `multiple_objects` wih all reasonable

objects that are in scope. The routine uses look-ahead to handle lists of noun phrases separated by commas or "and". It also detects the "all but X" pattern, and sets `parser_all_except_object` if found.

2.5.4   `_GetNextNoun`

`_GetNextNoun` takes the current input position and phase, and returns the object number for the next noun if no problem occured. In addition ot the return value it will also update `parser_action`.

`_GetNextNoun` first skips articles and "all", so it can parse noun phrases such as "all books", "the bird", and "an apple". It then checks if the current word is a pronoun such as "it" or "him". If it is a pronoun, a suitable objects has been referred to before so the parser knows who or what to refer to, and that object is still in scope, then the routine returns the object associated with the pronoun.

`_GetNextNoun` now calls `_CheckNoun` to get a list of objects in scope that match the current input. The matching words have their plural flag checked, and if the noun phrase indicated plurals (e.g. "books", "all birds") then `parser_action` is set to `##PluralFound`.

If a single object matches then it is returned. If more than one object matches and it is not a plural noun phrase then disambiguation takes place. In phase 1 it accepts the input for now, but if code is run again in phase 2 then `_AskWhichNoun` is called to prompt the user to disambiguate ("Do you mean the blue book or the red book?"). If the new input doesn't help then an error message is printed and the routine returns the error code.

2.5.5   `_CheckNoun`

`_CheckNoun` takes the current input position, and returns the object that matches one or more words. In addition to the return value, it updates the `which_object` global array, which contains a list of objects that matches (since there could be more than one), the number of objects that matches, and the number of words parsed against these object(s).

`_CheckNoun` loops over all objects in scope, trying to parse each of them against the words in the input, using either the `name` property or `parse_name`. There is additional logic to handle debugging verbs that need to try to match against any object, regardless of the normal scoping rules. This is only enabled if the DEBUG compiler flag is used.

`_CheckNoun` also takes into account if the object is concealed or in the open, by keeping track of a object level score. This is calculated by `_CalculateObjectLevel` and stored in the `which_level` array, which shadows `which_object`. Concealed objects will dropped if there are any openly visible objects that also match the input.

2.6 Utility Routines

2.6.1 _AskWhichNoun

`_AskWhichNoun` uses `which_object` to print a list of objects used in disambiguation. The typical output is "Do you mean X or Y?".

2.6.2 _AddMultipleNouns

`_AddMultipleNouns` is used do replace "all" with all suitable objects in scope.These objects are stored in the `multiple_objects` global array.

The routine takes the token type being processed, so that `MULTIHELD_OBJECT` will add all objects that are being held, which `MULTI_OBJECT` are all objects in scope, except for objects being animated, held or concealed.

2.6.3 _FixIncompleteSentenceOrComplain

`_FixIncompleteSentenceOrComplain` is called from `_ParsePattern` because the sentence shorter than the pattern. The routine checks if the pattern is expectign another noun phrase. If so, it can call `_GuessMissingNoun` to try adding the missing information. If `_GuessMissingNoun` manages to fix the sentence then `_ParsePattern` will return a perfect score, otherwise an error message is shown ("I think you want to say 'kill someone', please try again.").

2.6.4 _GuessMissingNoun

`_GuessMissingNoun` is used then `noun` or `second` is missing. It tries to guess the missing parts of the sentence. A typical usage is

```
> show diamond
(to Sally)
```

where `_GuessMissingNoun` checked the scope and found that only Sally was possible, so "(to Sally)" was written and `second` set to Sally to complete the parsing.

2.6.5 PronounNotice

This routine is called with an object, and update the matching pronoun. For example, the object Sally will set `herobj` to Sally, while the object Sword will set `itobj` to Sword.

2.6.6 _PrintPartialMatch

`_PrintPartialMatch` prints a grammar rule and is used to produce output such as "I only understood you are far as 'look' but then you lost me." as a reply to "look on me".

2.6.7 `_PrintUnknownWord`

Prints a word that doesn't exist in the dictionary by typing it from the `buffer` array, using `parser_unknown_noun_found` which points to an entry in the `parse` array. Used for messages such as "Sorry, I don't understand what 'sdasdasda' means."

Chapter 3

Grammar

The standard actions of PunyInform are defined in grammar.h. By default only the most essential actions are included, but different addional subsets can be enabled by defining the OPTIONAL_EXTENDED_VERBSET, OPTIONAL_EXTENDED_METAVERBS, OPTIONAL_PROVIDE_UNDO, and DEBUG constants.

For more detail and a list of standard actions defined for each subset, see the PunyInform manual.

Chapter 4

Messages

All texts and messages are located in messages.h, to make it easy to customise them. Customisation is described in the main PunyInform manual.

Puny internally accesses these messages through the `PrintMsg` function, which takes the identifier and optional arguments. For example, `PrintMsg(MSG_PARSER_NOT_MULTIPLE_VERB);` will print something like "You can't use multiple objects with thas verb.".

Chapter 5

Scope

Scope is a list of things you can interact with. 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. It is however possible to switch to manual scope updates by defining the constant `OPTIONAL_MANUAL_SCOPE`. With manual_scope enabled, scope is only updated at the start of each turn AND when the `scope_modified` variable is set to true.

The main routine is `_UpdateScope` which is called from `ParseAndPerformAction` and some other locations in the parser to update the scope when objects move or is modified by parsing the scope token. In addition, there are several utility functions that use the loop over or test if objects are in visible or touchable (that is, are in scope).

5.1   _UpdateScope

5.2   LoopOverScope

5.3   ScopeWithin

5.4   TestScope

5.5   ObjectIsUntouchable

5.6   ObjectIsInvisible

Chapter 6

Appendix: Infocom Dictionary and Grammar Formats

PunyInform uses the same dictionary and grammar formats that were created by Infocom, and which are also used by Inform 6. Infocom created two versions of the grammar tables, and we only use grammar version 2.

6.1   Dictionary

Extra data in dictionary:

```
byte 0 & $1 : Verb flag
byte 0 & $2 : Meta flag
255 - (byte 1)
```

Verb number (255 - value is for "traditional Infocom reasons")

6.2   Grammar 2

Grammar table is always located at the start of static memory (address pointed to by word at $0e in header). Word (Verb number) points to the start address for the grammar for a verb.

For a detailed description of grammar version 2, read the text starting with "GV2 is a much more suitable data structure" at https://www.inform-fiction.org/source/tm/TechMan.txt

```
Byte 0: Number of grammar lines for this verb + 1
```

```
Byte 1 ... Syntax line 0, 1, ...
```

Grammar line

```
0: Highbyte of action_value
1: Lowbyte of action_value
```

(action_value & \$0400) ~= 0 means the action is reversed

action = action_value & \$03ff

3 bytes per token:

- byte 0 & \$0f : token_type
- byte 1 + 2 : Token data

If token_type = 15 (ENDIT_TOKEN), this is the end of line, and byte 1 and 2 are not supplied.

```
ILLEGAL_TT          = 0;     ! Types of grammar token: illegal
ELEMENTARY_TT       = 1;     !    (one of those below)
PREPOSITION_TT      = 2;     !     e.g. 'into'
ROUTINE_FILTER_TT   = 3;     !     e.g. noun=CagedCreature
ATTR_FILTER_TT      = 4;     !     e.g. edible
SCOPE_TT            = 5;     !     e.g. scope=Spells
GPR_TT              = 6;     !     a general parsing routine
ENDIT_TOKEN         = 15
```

6.3   Grammar 1

Note that Grammar 1 is not used by PunyInform.

01 Number of grammar lines

```
00 params wanted
ff token 1
00 token 2
00 token 3
00 token 4
00 token 5
00 token 6
00 action number
```

```
02 Grammar line 1, highbyte of action routine
bd Grammar line 1, lowbyte of action routine
```

Tokens available in grammar version 1:

```
NOUN_TOKEN          = 0;     ! The elementary grammar tokens, and
HELD_TOKEN          = 1;     ! the numbers compiled by Inform to
MULTI_TOKEN         = 2;     ! encode them
```

```
MULTIHELD_TOKEN     = 3;
MULTIEXCEPT_TOKEN   = 4;
MULTIINSIDE_TOKEN   = 5;
CREATURE_TOKEN      = 6;
SPECIAL_TOKEN       = 7;
NUMBER_TOKEN        = 8;
ENDIT_TOKEN = $0f  ! Legal, but doesn't seem to be used
```

Token $10-$2f are routine filters (ROUTINE_FILTER_TT). e.g. noun=CagedCreature

Token $30-$4f are general parsing routines (GPR_TT). Routine address is in #preactions_table–>(token-48)

Token $50-$7f are scope tokens (SCOPE_TT). e.g. scope=Spells Routine address is in #preactions_table–>(token-80)

Token 80−b3 are attribute filters (ATTR_FILTER_TT). e.g. edible

Token b4−ff are prepositions (PREPOSITION_TT). e.g. 'into' Prepositions are located in the "adjectives table". The start address is in the constant #adjectives_table . Each entry consists of two words. If the second word is the preposition-number (like $ff), the first word is the address of the dictionary word. There is no length number or end marker. You should just expect to find the entry somewhere in there.