

# 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

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Program and Data Structures . . . . .	4
<b>2</b>	<b>Parser</b>	<b>5</b>
2.1	Overview . . . . .	5
2.2	State Variables . . . . .	5
2.3	Handling Errors . . . . .	6
2.4	Handling Disambiguation . . . . .	6
2.5	Main Routines . . . . .	7
2.5.1	_ParseAndPerformAction() . . . . .	7
2.5.2	_ParsePattern(p_pattern) . . . . .	8
2.5.3	_ParseToken(p_token_type, p_token_data) . . . . .	8
2.5.4	_GetNextNoun(p_parse_pointer) . . . . .	9
2.5.5	_ParseNounPhrase(p_parse_pointer) . . . . .	9
2.6	Utility Routines . . . . .	10
2.6.1	_AskWhichNoun(p_num_matching_nouns) . . . . .	10
2.6.2	_AddMultipleNouns(p_multiple_objects_type) . . . . .	10
2.6.3	_FixIncompleteSentenceOrComplain(p_pattern) . . . . .	10
2.6.4	_GuessMissingNoun(p_type, p_prep, p_nounphrase_num) . . . . .	10
2.6.5	PronounNotice(p_object) . . . . .	11
2.6.6	_PrintPartialMatch(p_start, p_stop) . . . . .	11
2.6.7	_PrintUnknownWord() . . . . .	11
<b>3</b>	<b>Grammar</b>	<b>12</b>
<b>4</b>	<b>Messages</b>	<b>13</b>
<b>5</b>	<b>Scope</b>	<b>14</b>
5.1	_PerformAddToScope(p_obj) . . . . .	14
5.2	_SearchScope(p_obj, p_risk_duplicate, p_no_add) . . . . .	15
5.3	_PutInScope(p_obj, p_risk_duplicate) . . . . .	15
5.4	_UpdateScope(p_actor, p_force) . . . . .	15
5.5	GetScopeCopy(p_actor) . . . . .	15

5.6	ScopeCeiling(p_actor, p_stop_before) . . . . .	15
5.7	TouchCeiling(p_actor) . . . . .	16
5.8	LoopOverScope(p_routine, p_actor) . . . . .	16
5.9	ScopeWithin(p_obj) . . . . .	16
5.10	TestScope(p_obj, p_actor) . . . . .	16
5.11	_ObjectScopedBySomething(p_obj) . . . . .	16
5.12	ObjectIsUntouchable(p_item, p_dontprint, p_checktake) .	16
5.13	_FindBarrier(p_ancestor, p_obj, p_dontprint) . . . . .	16
<b>6</b>	<b>Appendix: Infocom Dictionary and Grammar Formats</b>	<b>18</b>
6.1	Dictionary . . . . .	18
6.2	Grammar 3 . . . . .	18
6.3	Grammar 2 . . . . .	18
6.4	Grammar 1 . . . . .	19

# 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 available 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 `_ParseNounPhrase` to try parsing a noun phrase, but adds checks for pronouns and plurals, and disambiguates if needed.

`_ParseNounPhrase` 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 array, 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
- `parser_action`: this is set to `##PluralFound` if plurals, otherwise 0. Note that `##TheSame` (described in DM4) is not supported.
- `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. The current phase is stored in the global variable `parser_phase` and is either 1 or 2.

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 arrays `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). `_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 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 `parser_phase` set to 2. In phase 2 `_ParsePattern` will print the error messages that were suppressed during phase 1. If the match failed and `phase2_necessary` wasn't set it 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 ambiguous. 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



`parser_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(p_pattern)`

`_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(p_token_type, p_token_data)`

The format of `_ParseToken` is compatible with `ParseToken` for Inform 6 compatibility. `ParseToken` is mentioned in DM4 and can be used by games to provide custom parsing, so keeping the same format allows also PunyGames to offer this functionality.

The two arguments are the same as `ParseToken`: token type and token data. The routine returns the object number or a failure code (`GPR_FAIL`, `GPR_MULTIPLE`, `GPR_NUMBER`, `GPR_REPARSE` or `GPR_PREPOSITION`).

`_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 stores the object number in 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` with 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(p_parse_pointer)`

`_GetNextNoun` takes the current input position, and returns the object number for the next noun if no problem occurred. In addition to 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 `_ParseNounPhrase` 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 `_ParseNounPhrase(p_parse_pointer)`

`_ParseNounPhrase` 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). It can also modify the `wn` variable to skip words such as ‘the’ and ‘an’. If the routine is successful it will leave `wn` pointing to the first word of the found noun phrase.

`_ParseNounPhrase` loops over all objects in scope, trying to parse each of them

against the words in the input, first using the `ParseNoun` routine, if defined. If there is no `ParseNoun`, or if this routine declines to make a decision, the object's `parse_name` routine is checked. If `parse_name` isn't available or declines to make a decision, the `name` property is used. Note that `ParseNoun` is checked first. This is different from DM4.

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.

`_ParseNounPhrase` 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 be dropped if there are any openly visible objects that also match the input.

## 2.6 Utility Routines

### 2.6.1 `_AskWhichNoun(p_num_matching_nouns)`

`_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(p_multiple_objects_type)`

`_AddMultipleNouns` is used to 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(p_pattern)`

`_FixIncompleteSentenceOrComplain` is called from `_ParsePattern` because the sentence is shorter than the pattern. The routine checks if the pattern is expecting another noun phrase. If so, and if `OPTIONAL_GUESS_MISSING_NOUN` is defined, it can optionally call `_GuessMissingNoun` to try adding the missing information. If `_GuessMissingNoun` is available and 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(p_type, p_prep, p_nounphrase_num)`

`_GuessMissingNoun` is used when `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(p_object)`

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(p_start, p_stop)`

`_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 additional 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 that verb.”.

## Chapter 5

# Scope

Scope is a list of things an actor (typically the player) 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 updates enabled, scope is only updated when the `scope_modified` variable is set to true. The library sets it to true whenever library code does something that may affect scope, like when the player moves or opens or closes a container. If something happens in game code which may mean that what's in scope changes, the game programmer must set `scope_modified = 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 `_PerformAddToScope(p_obj)`

Check the contents of `p_obj.add_to_scope`. If it's an array, add all objects in the array to scope, plus any objects they want to add through their `add_to_scope` properties. If it's a routine, execute it. That routine can then add any objects it likes to scope by calling `PlaceInScope(p_obj)`.

## 5.2 `_SearchScope(p_obj, p_risk_duplicate, p_no_add)`

Place the specified object in scope, plus all its siblings and children. If `p_risk_duplicate` is `false`, check first that the objects haven't already been added to scope. If `p_no_add` is `false`, allow the `add_to_scope` property of every object to add objects to scope.

## 5.3 `_PutInScope(p_obj, p_risk_duplicate)`

(synonyms `PlaceInScope`, `AddToScope`)

Place an object in scope. If `p_risk_duplicate` is `false`, check first that the object hasn't already been added to scope. User code should ignore the parameter `p_risk_duplicate`, thus always leaving it as `false`.

This routine is used by the other scope routines in the library, as well as by `add_to_scope` routines.

## 5.4 `_UpdateScope(p_actor, p_force)`

Update the `scope` array to hold the objects currently in scope to `p_actor`. If the `scope` array seems to have the correct contents already, skip the update - *unless* `p_force` is `true`.

## 5.5 `GetScopeCopy(p_actor)`

Calculate what's in scope for `p_actor`, and create a copy of the `scope` array in the `scope_copy` array. This is needed when looping over scope items and performing operations which may change the contents of the `scope` array, like calling `TestScope` for another actor.

## 5.6 `ScopeCeiling(p_actor, p_stop_before)`

Find the innermost closed non-transparent container the actor is in, or the room the actor is in.

Start with the actor and move upwards in the object tree until a closed non-transparent container or the room is found. If, however, `p_stop_before` is found along this path, return the object that was found just before it, one step closer to the player.



## 5.7 TouchCeiling(p\_actor)

Find the innermost closed container the actor is in, or the room the actor is in.

## 5.8 LoopOverScope(p\_routine, p\_actor)

Call a routine once for every object in scope to an actor (default is the player).

## 5.9 ScopeWithin(p\_obj)

Add everything inside an object, but not the object itself, to scope. This routine should only be used in scope routines, and only when `scope_stage == 2`.

## 5.10 TestScope(p\_obj, p\_actor)

Check if an object is in scope to a certain actor (default is the player).

## 5.11 \_ObjectScopedBySomething(p\_obj)

If the specified object is in an `add_to_scope` array of any other object, anywhere in the game, return that object's object ID.

## 5.12 ObjectIsUntouchable(p\_item, p\_dontprint, p\_checktake)

Check if there's something stopping the player from touching a certain object. If parameter `p_dontprint` is set to `false`, print a message saying why the player can't get to the object. If parameter `p_checktake` is set to `true`, extend the check to decide if the player can take the object. I.e. a button that is part of a machine can be touched but not taken.

## 5.13 \_FindBarrier(p\_ancestor, p\_obj, p\_dontprint)

Utility function used by `ObjectIsUntouchable` to find out if there are barriers between an object and one of its ancestors in the object tree that prevent the player from touching or seeing the object.

To allow this function to work for z3 games, where a function can not be called with more than three arguments, three global variables are used exclusively to pass parameters to this function: \* `_g_item` - the object which the calling function is trying to figure out whether it can be seen or touched \* `_g_check_visible` - `true` means we're checking if the object can be seen, `false` means we're checking

if it can be touched. \* `_g_check_take - true` means we should check if the player can take the object.

If parameter `p_dontprint` is set to `false`, this function prints an error message if it finds such a barrier. It might be something like “But the aquarium is closed!”

## 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 3.

### 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 3

The GV3 format is documented here: <https://inform-fiction.org/manual/I6-Addendum.html#grammarformat>

### 6.3 Grammar 2

*Note that Grammar 1 is not used by PunyInform.*

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		

## 6.4 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.