NScLisper開発者
zick
全面監修!

監修　zick
著者　zick

あどばんすど NScLisper

えぬすくりすぱー

オフィシャルガイド

λ組

注意

- 本書は存在自体がネタです。あまり本気にすると痛い目にあうかもしれません。間違いが無いかを気にしながら読むのではなく、正しい箇所がどこにあるか注意しながら読むことを推奨します。
- 本書に掲載されているソースコードは自由に使用してかまいません。煮るなり、焼くなり、炒めるなり、揚げるなり好きにしてください。
- 今更ですが実は筆者は、LispもNScripterも普段あまり使っていません。間違い等を見つけられた方は「若いっていいなぁ……」と思いながらスルーしてください。こっそり報告してくれるとなおよいです。なお、筆者は若いです。
- 本書を読まれる際は、1時間につき15分程度の休憩を入れてください。
- 本書を読むときは部屋を明るくして、本書から離れて読んでください。
- 本書は大学生であるzickが講義をサボりながら作成したものです。そのzickは、早くも「単位が危ないかも……」などと危惧しています。学生の方は講義のサボりすぎに注意しましょう。
- 言い忘れてましたが、著作権法の定める範囲を超え、無断で複製、複写することを一応禁じておきます。引用程度ならご自由にしていただいてかまいません。

はじめに

NScripter のバイブルである [あどばんすど NScripter オフィシャルガイド] にて、のぞみ女史[1]は仰られました。

「うまく動かなかったり思い通りにできなかったりするかもしれないけど**プログラミング**っていうのは…」

はっきりと ”プログラミング ”と仰られています。

これは、NScripter でプログラミングをやれという御達しと私は受け取りました。自然言語の中に人工言語を少し埋め込むようなアドベンチャーゲーム作りに満足してはいけません。美しき人工言語をガリガリ書いてこそプログラミング。いざ、本当の NScripter の世界へ！

謝辞

最も感謝しなければならないのは、妻の ゆのは[2] です。普段、文章をあまり書いていない私にとって、長い文章を書くのは大変な作業でしたが、妻は常に私に協力的で、私を癒してくれました。

また、本書のレヴューを引き受けてくれた、 nathki 氏 に多くの感謝をささげます。彼がいなければ、多くの誤字を含んだままこの本を出すことになっていたでしょう。

最後に、編集者である bugyo 氏 に感謝しなければなりません。彼は私に多くの的確なアドバイスをしてくれました。

[1] http://www.shuwasystem.co.jp/cgi-bin/detail.cgi?isbn=4-7980-1104-5　表紙の右側の少女のことです。

[2] http://www.pulltop.com/gp04/

NScripterってなにさ

NScripter とは、高橋直樹氏が作成したアドベンチャーゲーム作成用のスクリプトエンジンです。記法は BASIC やアセンブリに似ていますが、日本語 (全角文字) を書くと、それがそのまま文字列として画面に表示されるなど、非常に簡単にゲームを作れるように工夫されています。

しかし、その真髄は、簡単に作れることなどではなく、ちゃんとしたプログラミングができることにある……と、私は信じています。

NScLisperってなにさ

NScLisper とは、ここで説明する NScripter で作成された Lisp インタプリタのことです。NScripter を使って表示したテキストフィールドから文字列を読み込み、それを Lisp の S 式として評価し、結果を表示します。GC(Garbage Collection) も自分で実装しています。NScripter 使いなら一度はインタプリタを NScripter の上で書いてみたいものですよね！

ここには何が書いてあるの？

この文章の目的は、『Lisp は知ってるけど、NScripter は知らない』とか、『NScripter は知ってるけど、Lisp は知らない』といった人のギャップを埋めることです[3]。

全体としては、NScripter の基礎の説明に始まり、最終的には、Lisp インタプリタを作るうえで**土台**となる部分を作り上げていきます。Lisp インタプリタ作成の華 (?) である、リーダや評価関数については全く触れません。そのあたりは資料がゴロゴロ転がっているはずなので、お好きなものを読んでください。

[3]後になって読み返してみたらこの目的は全く達成されていませんでした (笑)

第1章　NScripterの基礎

既に言ったとおり、NScripter は簡単なんですが、変数の管理などにちょっと癖があるので、知らない人はここを読んでおいた方がいいかと思います。ここは読み飛ばして、あとで分からないところがあればここを読み返すという方針をとってもかまいません。使えない文章ですが、どうぞ、ご自由にお使いください。

1.1　変数

NScripter で扱えるデータは**数値**と**文字列**に分かれていて、それぞれを保持するための**数字変数**[1]と**文字変数**[2]が用意されています。

1.2　数字変数

数字変数の名前として使えるものは次のようになっています。

```
%n
```

n は $0 \leq n \leq 4095$ を満たす整数です。なお、この n のことを**変数番号**と呼びます。例えば、%0 や%123 といったものが数字変数です。

数字変数は符号付 32bit 整数[3]で、mov 命令を使うことにより値を代入できます。

[1]数値変数の方が正しい気がするんですが、本家では数字変数となっています。
[2]これも、文字列変数といった方が正しい気がしますが、本家にあわせます。
[3]実際に値を入れて確かめた結果。後に仕様が変わるかもしれません。

```
mov %0, 2500
mov %1, %0
```

これは、変数番号 0 の数字変数に数値 2500 を代入して、その後、変数番号 1 の数字変数に%0 の値 (=2500) を代入します。

NScripter では C 言語のポインタによる参照のようなことができます。

```
%%m
```

m は $0 \leq m \leq 4095$ を満たす整数であり、かつ、%m は $0 \leq \%m \leq 4095$ を満たします。これは変数番号%m の数字変数を表します。変数番号を変数のアドレスと考えると、変数番号 m の数字変数は、アドレスを保持するポインタの役割を果たし、左側の%は参照演算子の役割を果たすとも考えられます。

変数の参照の例を以下に示します。

```
mov %0, 10
mov %%0, 777
```

これは、変数番号 0 の数字変数に 10 を代入して、変数番号%0(=10) の数字変数に 777 を代入します。よって、変数番号 10 の変数に 777 が代入されます。

1.3　変数のスコープ

NScripter にはブロック構造がないため、ローカル変数というものはありません。すべての変数にどこからでもアクセスすることができます。

1.4　文字変数

文字変数の名前として使えるものは次のようになっています。

```
$n
```

n は $0 \leq n \leq 4095$ を満たす整数。数字変数と同様に、この n のことを**変数番**

号と呼びます。例えば、$0 や$123 といったものが文字変数です。

変数同様に mov 命令で値を代入できます。なお、文字列は二重引用符でくくります。

```
mov $0, "hoge"
mov $1, $0
```

また、数字変数と組み合わせ、間接的にアクセスもできます。

```
mov %0, 10
mov $%0, "foo"
```

1.5 エリアス

変数番号を覚える手間を省くのに、エリアスというものが使えます。

```
numalias var_name, 0
```

こう書くことにより、%0 や$0 と書く代わりに%var_name や$var_name と書くことができます。なお、エリアスは変数番号の代わりだけではなく、値の代わりとしても使えます。すなわち、『mov %0, var_name』のような記法が許されます。

1.6 処理の流れ

NScripter では基本的に、書かれた命令が上から順に実行されていきます。関数のようなものは用意されていませんが、ラベルジャンプ、ラベル呼び出しといったことができます。

1.6.1　ラベルジャンプ

```
mov %0, 1
goto *foo
mov %0, 2
*foo
```

もちろん%0 の値は 1 になります。ラベルの名前はアスタリスクから始まるということ以外に特に言うことはないでしょう。

1.6.2　ラベル呼び出し

```
mov %0, 1
gosub *foo
mov %0, 3
goto *baa
*foo
  mov %0, 2
  return
*baa
```

多くの方が想像されたとおり、%0 の値は 3 になります。gosub で指定したラベルに制御を移し、return で戻ります。

1.7　関数

ラベル呼び出しだけでは、引数を渡したり、戻り値を返したりすることは出来ません。関数がある言語ばかり触っていた私にはこれは非常にやりにくいので、ラベル呼び出しを関数呼び出しのように扱えないかと考えました。そこで、特定の変数を ”引数を保持する変数” や、”戻り値を保持する変数” として利用しました。

```
*increase
  inc %arg0              ;1 番目の引数に 1 を加える
  mov %ret, %arg0        ; それを戻り値として設定
  return

*addition
  add %arg0, %arg1       ;1 番目の引数に 2 番目の引数を加える
  mov %ret, %arg0        ; それを戻り値として設定
  return
```

ただし、このままではラベル呼び出しを使うと、変数の中身が書き換わるため、スタックを自分で作成します。これについては後で説明します。

これにより、ラベル呼び出しは実質、他の言語の関数呼び出しと同等に扱えるので、ラベル呼び出しで呼び出される**ラベルのことを以後、関数と呼びます**。

第2章　数字変数の配置

さて、ここからいよいよ Lisp インタプリタの設計に入ってきます。NScripter では数字変数は 4096 個しか用意されていないので、計画的に使わないとすぐなくなります。NScLisper は次のように目的別に区切って変数を使用しました。

0〜49	拡張用変数
50〜99	汎用変数および特定の値へのポインタ
100〜2999	Lisp 用セル
3000〜4000	スタック
4001〜4095	未使用

2.1　拡張用変数

変数番号 0〜49 は拡張用の変数として残しておきました。リリカル Lisp はこの拡張用の変数を利用しています。

2.2　汎用変数および特定の値へのポインタ

変数番号 50〜99 は汎用の変数として使用しました。計算結果を一時的に保持したり、先ほど出てきた引数や戻り値に利用します。あと、特定の Lisp の値 (nil や#t や#f など) を含む変数へのポインタとしても利用しました。

2.3 Lisp 用セル

変数番号 100〜2999 は Lisp の値を保持するために使用しました。変数一つを一つのセル (Lisp の値の入れ物) と見なし、最大 2900 個のコンスを作成できます。この部分が足りなくなれば GC を行います。

2.4 スタック

変数番号 3000〜4000 はスタックとして使用しました。ラベル呼び出しを関数呼び出しのように見せかけるため、ラベル呼び出しの際にはこのスタックに必要な変数を退避させます。

2.5 未使用

変数番号 4001〜4095 は特に何にも使っていません。別に 4095 までスタックとして使えばよかったんですが……なんで使わなかったのかはよく覚えていません (笑)

第3章　スタック

3.1　Push

スタックへ Push するための関数は次のようになっています。

```
*push
  if %sp >= STACK_OVER /* エラー */
  mov %%sp, %arg0
  inc %sp
  return
```

sp はスタックポインタ、STACK_OVER は、スタックの範囲を超えた点 (4001) です。Push はスタックポインタの指し示す位置に変数 arg0 の値をコピーし、スタックポインタを動かします。

NScripter のコメントは セミコロン; を用い、/**/の形式のコメントは受け付けませんが、ここでは、コードを省略して説明をいれるときなどに/**/を使うことにします。

スタックに Push するときは、まず変数 arg0 に値をセットして、push を呼び出します。

```
mov %arg0, 100
gosub *push  ; スタックに 100 を積む
```

3.2　Pop

スタックから Pop するための関数は次のようになっています。

```
*pop
  if %sp <= STACK /* エラー */
  dec %sp
  mov %ret, %%sp
  return
```

STACK はスタックの開始位置 (3000) です。Pop はスタックポインタを動かし、その指し示す内容を変数 ret にコピーします。

Push はスタックの内容とスタックポインタしか書き換えませんが、Pop は変数 ret を書き換えるというのが重要です。

Pop を呼び出すときは次のようにします。

```
gosub *pop
/* 変数 ret を利用 */
```

3.3 関数呼び出しへの利用

先ほど説明したスタックを使う例として、再帰をする関数を書いてみます。

```
;; 呼び出すときは arg0 に整数をセットすること
*factorial
  ;"引数"が 0 ならば、1 を返す
  if %arg0 == 0 mov %ret, 1 : return
  gosub *push       ; スタックに"引数"が積まれる
  dec %arg0         ;"引数"から 1 引いたものを arg0 にセット
  gosub *factorial ; 再帰呼び出し
  mov %arg0, %ret   ; 再帰呼び出しで得られた値を arg0 にセット
  gosub *pop        ; スタックから"引数"が取り出され、ret にセッ
トされる
  mul %ret, %arg0   ;ret に arg0 を掛ける
  return            ;ret を戻り値として返す


mov %arg0, 10       ; 引数に 10 を設定
gosub *factorial    ; 関数 factorial の呼び出し
;; %ret に 10! = 3628800 がセットされている
```

スタックを使うことによって、本来、関数と呼べるものがない NScripter で再帰が書けてしまいました。ただし、ラベル呼び出しの深さは 200 まで[1]なので、200 回以上再帰すると強制的に NScripter に落とされます[2]。

[1] NScripter 側の制限なのでどうしようもありません。

[2] NScLisper ではこの対策をしっかりとしていないため、Lisp 側で末尾再帰でない深い再帰をすると強制終了します。

第4章　Lispの値のデータ構造

NScLisper では数字変数一つ (=32bit) をセル (Lisp の値の入れ物) 一つ分として利用します。また、どの型の値でも、1bit 目は GC のマークビット、2〜6bit 目はデータ型を表すビット (タグ) として利用します。タグの部分にはデータ型に応じて、固有の定数[1]を代入します。

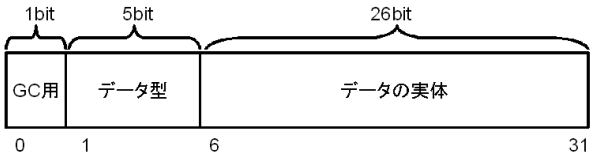


図 4.1: セルの割り当て

4.1　コンス

1bit　GC のマークビットとして利用

2〜6bit　タグとして利用

7〜19bit　CAR 部 (セルへのポインタ)

19〜32bit　CDR 部 (セルへのポインタ)

[1]定数は numalias を利用します。

4.2　数

1bit GC のマークビットとして利用

2〜6bit タグとして利用

7〜32bit 数をそのまま格納

4.3　シンボル

1bit GC のマークビットとして利用

2〜6bit タグとして利用

7〜32bit 実体の入った文字変数へのポインタ[2]

[2] ここでは文字変数の管理については触れないので、文字変数へのポインタといっても、細かいところはあまり気にしないでください (笑)

第5章　getterとsetter

一つのセルには三つ (またはそれ以上) のデータが入っているので、それらを個別に取得、設定するための関数を書きます。

5.1　getter

これらの関数はセルへのポインタを受け取り、その GC のマークビット/タグ/データ部分を取得します。

本来は、右シフトと and を使いたいんですが、NScripter にはビット操作命令が用意されていないので、除算と剰余で代用しています。

```
;; これら三つの関数を呼び出す際には arg0 にセルへのポインタをセッ
トすること
*get_gc
  mov %ret, %%arg0
  mod %ret, 2
  return

*get_tag
  mov %tmp, %%arg0
  mod %tmp, 64
  mov %ret, %tmp
  div %ret, 2
  return

*get_data
  mov %ret, %%arg0
  div %ret, 64
  return
```

これらの関数は連続して呼び出すことが多い (タグを取得して、それから内容を取得する等) ので arg0 は書き換えないようにしています。

5.2 setter

これらの関数はセルへのポインタと、設定する値を受け取り、その GC のマークビット/タグ/データ部分の値を設定します。

左シフト、or を乗算、加算で代用しています。

```
;; これら三つの関数を呼び出す際には
;;arg0 にセルへのポインタを、
;;arg1 に設定する値をセットすること
*set_gc
  mov %tmp, %%arg0
  div %tmp, 2
  mul %tmp, 2
  mov %%arg0, %tmp + %arg1
  return

*set_tag
  mov %tmp, %%arg0
  mov %tmp1, %tmp
  div %tmp1, 64
  mul %tmp1, 64  ;DATA
  mod %tmp, 2    ;GC
  mov %tmp2, %arg1
  mul %tmp2, 2   ;TAG
  mov %%arg0, %tmp + %tmp1 + %tmp2
  return

*set_data
  mov %tmp, %%arg0
  mod %tmp, 64
  mov %tmp1, %arg1
  mul %tmp1, 64
  mov %%arg0, %tmp + %tmp1
  return
```

getter 同様、これらの関数は連続して呼び出すことが多いので、arg0、arg1 は書き換えないようにしています。

5.3 既知のバグ

実は、この方式には大きな落とし穴があります。それは、最上位ビットが 1 になると、数字変数が負数として扱われるため、四則演算によるビット演

算が上手く働かないということです。

今のところ、その状況に実際に遭遇したことはありませんが、頑張れば遭遇できるはずです (笑)

今になって思えば、GC のマークビットを最上位ビットにしてしまえば、この問題が発生するときが GC の時だけになり、簡単に対策できたんですが、すでにリリースしてしまった今となってはどうしようもありません (泣)

第6章　セルの管理

NScLisper では新たな値が生まれるたびに[1]、それを格納するセルを割り当てます。セルは 2900 個しか用意されていないため、例えばフィボナッチ数列を求めるなど、再帰を多用する式を評価すると、すぐに使い切ってしまいます。

そのため、Garbage Collection を行い、セルを再利用します。まず、起動時に全てのセルをフリーリストに繋ぎ[2]、Lisp の値を作成するときには、フリーリストの先頭のセルを割り当て、フリーリストの先頭を一つ後ろにずらします。そして、セルが足りなくなったら処理を停止し、単純な Mark and Sweep の GC をします。

6.1　フリーリストの作成

フリーリストの作成は次のように行います。

1. フリーリストの開始点を一つ目のセル (MEM=100) に設定
2. 変数番号 MEM(100)〜MEM_END(2999) について次の操作を行う
   - それらのタグを全て TAG_FREE に設定
   - それらの内容を次のセル (変数番号が一つ大きいもの) に設定

具体的には次のようになります。

[1] 例えば、(+ 1 1) を評価すると、2 という値が生まれる訳です。
[2] これがリリカル Lisp の起動時間を長くする一つの大きな要因です。

```
mov %free_lst, MEM
for %i=MEM to MEM_END
  mov %arg0, %i
  mov %arg1, TAG_FREE
  gosub *set_tag
  mov %arg1, %i + 1
  gosub *set_data
next
```

6.2　セルの割り当て

新しいセルを割り当てる時は次のように行います。

1. フリーリストの先頭のセルの内容 (=次のセル) を読み込む
2. 先頭のセルを割り当てるセルに設定
3. フリーリストの先頭を読み込んだ値に書き換える

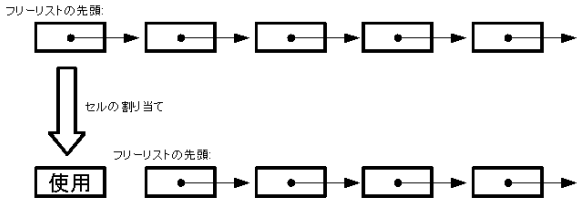


図 6.1: セルの割り当て

```
*next_cell
  ;; メモリ不足なら一度 GC を行い、なお不足すれば終了
  if %free_lst == MEM_OVER gosub *gc
  if %free_lst == MEM_OVER /* メモリ不足 */
  mov %arg0, %free_lst
  gosub *get_data
  mov %tmp, %free_lst
  mov %free_lst, %ret
  mov %ret, %tmp
  return
```

フリーリストの最後は必ず MEM_OVER(3001) なので、先頭がこの値なら GC を行い、それでもなお、先頭がこの値の場合、メモリ不足で終了します。

6.3 Garbage Collection

Garbage Collection は以下の手順で行います。

1. 汎用変数の値を全てスタックに退避
2. スタックに積まれているセルのマークビットを 1 に設定
3. そのセルから辿り着ける (ポインタが入っている) セルのマークビットも 1 に設定
4. マークビットが 0 であるセルをフリーリストに繋ぐ
5. フリーリストの最後を MEM_OVER に設定

6.3.1 スタックの走査

基本的には現在のスタックポインタの位置からスタックの先頭までに積まれたセルのマークビットを 1 に設定するだけです。ただし、コンスなどセルへのポインタを持った値に遭遇した場合、その先のセルのマークビットも 1 に設定します。そのセルがポインタを持っている場合も同様です。

スタックにはセル以外の値が積まれることもあります。セルは 100〜2999 の間の値と決まっていますが、偶然この範囲の数値が積まれることもあります。しかし、余計なセルを保護したところで、それほど損をするわけではないので[3]、この範囲の数値がスタックに積まれていればセルかどうかは気にせず、その番号のセルのマークビットを 1 にすることにしています[4]。

```
*gc_mark
  for %i = STACK to %sp-1
    ; セルの範囲外ならなにもしない
    if %%i < MEM goto *gc_mark_l1
    if %%i > MEM_END goto *gc_mark_l1
    mov %arg0, %%i
    ; セルのマークビットを 1 にする (内容は省略)
    gosub *gc_mark_lobject
  *gc_mark_l1
  next
  return
```

6.3.2　フリーリストの再構築

マーク付けが終わると、セルの領域を先頭から辿って行き、マークビットが 0 のセルに出会うと、そこをフリーリストの開始点に設定します。次にマークビットが 0 のセルに出会うと、前回出会ったセルの内容をそのセルに設定し、以下これを繰り返します。全ての領域を辿った後に、最後に出会ったセルの内容を MEM_OVER(3001) に設定します。

[3] 大きな損をする場合も考えられますが、それは気にしないでください (笑)
[4] いわゆる保守的 GC というやつです。

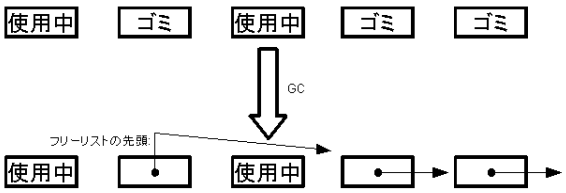


図 6.2: フリーリストの再構築

```
*gc_sweep
  mov %free_lst, MEM_OVER
  ; 一つ目の未使用セルを見つける
  for %i=MEM to MEM_END
    mov %arg0, %i
    gosub *get_gc
    if %ret == 0 mov %free_lst, %i : break
    mov %arg1, 0
    gosub *set_gc
  next
  if %free_lst == MEM_OVER return  ; 未使用セルは無し
  mov %gc_tmp, %free_lst ; 一つ前の空きセルを保持
  for %i=%i+1 to MEM_END
    mov %arg0, %i
    gosub *get_gc
    ; 一つ前の未使用セルに新しい未使用セルを繋ぐ
    if %ret == 0 mov %arg0, %gc_tmp : mov %arg1, %i
    if %ret == 0 gosub *reuse_cell : mov %gc_tmp, %i
    ; マークビットを 0 に設定する
    mov %arg0, %i
    mov %arg1, 0
    gosub *set_gc
  next
  mov %arg0, %gc_tmp
  mov %arg1, MEM_OVER  ; フリーリストの終端はこの値
  gosub *reuse_cell
  return

*reuse_cell
  gosub *set_data
  mov %arg1, TAG_FREE
  gosub *set_tag
  return
```

おまけ

リリカル Lisp が出来るまで

リリカル Lisp の誕生から完成までを時系列順に簡単にまとめて見ました。何の役にも立ちませんが、ごゆっくりご覧ください。

07/01/11 NScripter で Lisp インタプリタを作ろうと思い立つ

07/01/16 インタプリタが完成。これを使って何かをやろうと決意

07/01/18 リリカル Lisp の第 1 話の元となるものが完成

07/01/20 なのは Festival2 に出展することになる

07/02/02 体験版を出す予定だったが間に合わなかった

07/02/03 体験版を公開

07/02/04 zick が力尽きる

07/03/02 zick が作業の遅れに焦りだす

07/03/14 zick の頭がオーバーヒート

07/03/29 シナリオが完成

07/03/31 用語集/補足説明が完成

07/04/01 デバッグが始まる

07/04/07 デバッグが終了。CD を焼き始める

07/04/08 なのは Festival2 にて頒布

NScLisper 占い

以下の質問に Yes か No で答えてね♪

- 一度座ると動きたくない
- けど、走るのは好き
- カップヌードルばかり食べても大丈夫
- でも、密かに健康のことを心配している
- 好きなゲームはもちろん CLANNAD
- あと、なのは より ゆのは の方が好き

Yes が 0〜1 個のあなたは……

将来は医者か弁護士です！　既に就職されている方はきっと転職します。そんなあなたのラッキーカラーは群青色です。

Yes が 2〜4 個のあなたは……

今日はちょっといいことあるかも☆　けど、頑張りすぎには気をつけて！そんなあなたのラッキーカラーは緑みの青です。

Yes が 5〜6 個のあなたは……

お前は俺か！　大学生なら今日は講義をサボるでしょう。そんなあなたのラッキーカラーは光の原色の緑です。透明色に使用されるような一日になるでしょう。

あとがき

なのは Festival2 にて、リリカル Lisp を頒布した際、『プログラミングを学べます』という売り文句を見た人に、「どのプログラミング言語なんですか？」と聞かれることが何度かありました。はっきりと、リリカル **Lisp** と書いてあったのに。

その日、Lisp という言語の知名度は想像以上に低いということを肌で感じました。よくよく考えると、私も大学に入るまで Lisp を知りませんでした (汗)

あまり出来のいいものではありませんが、リリカル Lisp が一人でも多くの人に Lisp という言語を紹介することが出来れば幸いです。

稚拙な文章に最後までお付き合いいただき本当にありがとうございました。

2007 年 5 月 2 日 zick

参考文献

[あどばんすど NScripter オフィシャルガイド] 高橋直樹 監修　畔田英明 森皿尚行 著．あどばんすど NScripter オフィシャルガイド，秀和システム

■著者略歴

zick （じっく）

200X年　某大学入学
現　在　　λ組
http://lambda.bugyo.tk/

エヌスクリスパー
あどばんすどNScLisper
オフィシャルガイド

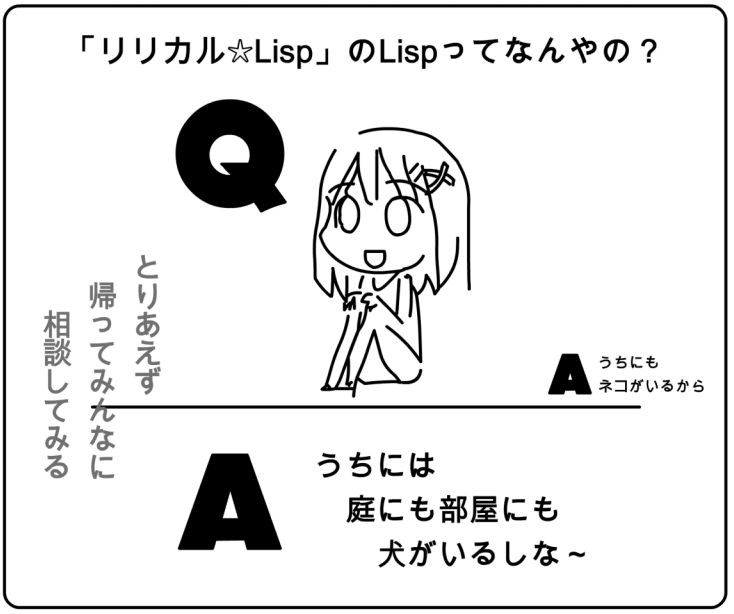
平成19年5月2日　初版　第1刷発行

著者　zick
印刷　カンプリ高槻店

定価は裏表紙に表示されています

あどばんすどNScLisper
オフィシャルガイド

入組
定価：時価＋税



～あらすじ～

東大に合格が決まった並岳荘の面々。しかし、達也の受験番号をもう一度見直してみると、認識していた番号と微妙に違っていた。達也は皆にこれを言い出せるのだろうか？　一方、墜落して捕虜となったポーリンを救出に向かったセドリックは白骨化した姿で発見された。ゆのはの神力をもってしてもPS3は売れないのだろうか？　月が赤く煌めく今夜、全ての謎が解き明かされる！！