平成 29 年 11 月 18 日
京都工芸繊維大学コンピュータ部

Lime 56

はじめに

コンピュータ部部長の北村です。このたびはお忙しい中、本学の学園祭にお越しいただき、我々コンピュータ部にご興味、ご関心を持っていただきありがとうございます。この度は Lime56 号を手に取ってくださったこと、部を代表いたしまして、厚く御礼申し上げます。この Lime は部員の活動の一部を記したものです。日々の活動の成果や部員個人の趣味全開なものまで内容は多岐にわたります。これを通じて、コンピュータ部のことを知っていただき、少しでも、我々の活動に興味を持っていただければ幸いです。そして、今年の Lime 作成にあたり、編集をしてくれた森下君、表紙の作成にあたった谷口君、忙しい中記事を書き上げた部員各位、並びに、OB・ OG の方々を含め、日々我々の活動を支えてくださっているすべての方への感謝をもって始めの挨拶とさせていただきます。

平成 29 年 11 月吉日
京都工芸繊維大学コンピュータ部部長　北村 馨

目 次

1 オレオレプログラミング言語（？）を作る

情報工学課程 1 回生 寺村英之 (aka. codeworm)

読者の皆さんはじめまして。この記事ではプログラミング言語について色々わからないなりに自作の言語を作ってみた話をします。

1.1 はじめに

プログラムを拡張できるようにするには

プログラムを拡張するのに拡張用設定を作るだけでは不十分なことがあります。例えばアプリケーションの中でごく小さなアプリケーションのようなもの[1]を動かしたいとします。それを実現するためにプチアプリのベースとなる抽象的なものをあらかじめ用意し、その動作を定義するために設定項目をいくつか作ってその設定ファイルを読み込むようにするとどうなるでしょうか。小規模なものならそれでも十分かもしれませんが、プチアプリに対するユーザーの入力手段などが増えるとそれぞれの事象ごとに設定項目を作らないといけないので規模が上がるにつれどんどん複雑になってしまいます。その上それら設定項目は元のアプリケーションの開発者が考えなければならないので自由度があまり高くならないようにも思われます。そんなときアプリケーション上でプログラムを実行するようなことをすれば単純化できるのではないかと思うことがあります。プログラミングなら似たような処理をひとまとめにできたり動作の場合分けを簡単に表現できたりするからです。でもそんなことができるのでしょうか？

搭載できるプログラミング言語

その方法の一つとして組み込み言語を用いるという方法があります。組み込み言語とはアプリケーションの中に処理系を容易に搭載できる比較的単純なプログラミング言語のことで、これを用いることでアプリケーションをプログラミングという方法で操作することができます[2]。組み込み言語には Lua や Javascript、Squirrel などがあります。今回はそんなものを実際に作ってみようと思います。

---

[1] いわゆるアドオンのようなものを想像されたい

[2] 実は組み込み言語の処理系はサンドボックスとして使われていることもある。例えば Python で書かれたアプリケーションはモジュールを動的に追加することで機能を増やせそうだがそれでは何でもアドオン側でできてしまうため都合が悪い。組み込み言語処理系上でアドオンを実行すればアプリケーションが提供する機能しか使うことができなくなり問題が解決する

1.2　とにかく作ってみる

そんなわけで夏休みに部内で行われたハッカソン＋ α の時間を使って Python [3]でオレオレ言語の処理系を実装してみました。言語は Iceberg という名前にしました。この記事でも以下そのように呼びます。

簡単な仕様

Iceberg は**命令＋引数列**の文の集まりで書かれ、文と文は改行文字の 0x0A(New Line) で区切られます[4]。下に文の構造を示します（コード 1.1）。**operation** が命令で**<arg-n>**が引数です。命令と引数列は半角スペースで区切られています。命令によっては引数列がない場合もあります。

```
operation <arg-1>,<arg-2>,...,<arg-n>
```

ソースコード 1.1: 命令文のかたち

またプログラム上の位置を指示するための「ラベル」もあります。半角英数字の前に**@**をつけるとラベルになります。これを文として書いておくと分岐命令（後述）でそこにジャンプすることができます。BASIC やアセンブラにある同名のものと同じような感じです。型は **INT**（整数型）、**FLOAT**（浮動小数点数型）、**BOOL**（真偽値型）、**STR**（文字列型)、**LABEL**（ラベル型）の 5 つがあります[5]。一部の型にはリテラルが存在します（表 1.1）。

型	リテラルのかたち
INT	小数点のない数字 0 42 -128
FLOAT	小数点のある数字 3.14 .525 25.
BOOL	true か false
STR	”や’ で囲まれた文字列 ”It’s sunny” ’It was ”unknown” for a while’

表 1.1: リテラル

変数はなければ作られます（代入が起こるときに作るかどうか判断される）。変数は値が代入された時点で型が固定されるので後で別の型の値を入れることはできません。変数を作る一つの方法は組み込みの代入命令 **let**（コード 1.2）を使うことです。

```
let ＜変数名＞,＜代入するもの＞
```

ソースコード 1.2: let 命令

変数もまた引数になることができます。変数の値は命令実行時の変数の値を使い参照渡しはしません。例えばある変数に他の変数を代入したあと元の変数を変更しても代入先の変数は影響を受けません。

Iceberg では型の暗黙的な変換が行われません。たとえば **INT** 型と **FLOAT** 型を足し算することはできません。これはカウンタとして使っていた INT 型の変数の値が無

[3]巷では人工知能でホットな言語とされているがもちろん汎用の言語である。スクリプト言語の一つとされることがある
[4]つまり一行に一文書くということ
[5]処理系のソースコードには T_UNDET や T_ANY などの型のような記述があるが今の所型ではない。用途は後述する。

意識のうちに FLOAT に変換されていて計算後にできた誤差が影響してしまうのを防ぐためです。どこで切り捨てや丸めをすればいいか意識して書くことになります（以下余談、実は言語を自作してみようと思ったモチベーションの一つとして自前のゲーム開発支援ソフトを作りたかったということがあります。ゲームでは例えば整数値の能力値を 1.2 倍するなどの処理があり、こんなときに計算誤差で妙なバグが起こってほしくなかったのです）。Iceberg では型の変換のためにキャスト命令があります。これはほぼ **let** 命令ですが命令に対応する型に代入するものが変換されるという点で異なります。

制御処理は 2 つの分岐命令で行います。**goto** はその一つで無条件に指定したラベルまでジャンプします。もう一つは **when** で与えられた引数が **true** なら **goto** します。

その他の組み込み命令も含めて表にまとめると次のようになります（表 1.2）。

表 1.2: 命令表

命令	文/処理
nop	nop 何もしません
let	let <変数名><代入するもの> 変数に値を代入します
add	add <オペランド A><オペランド B><代入先> オペランド A と B を足して代入先に結果を格納します
sub	sub <オペランド A><オペランド B><代入先> オペランド A から B を引いて代入先に結果を格納します
mul	mul <オペランド A><オペランド B><代入先> オペランド A と B をかけて代入先に結果を格納します
div	div <オペランド A><オペランド B><代入先> オペランド A を B でわって代入先に結果を格納します （結果は小数点以下切り捨て、INT 同士なら商）
div_ r	div_ r <オペランド A><オペランド B><代入先> div 命令の結果が常に FLOAT 型なバージョン
mod	mod <オペランド A><オペランド B><代入先> オペランド A を B で割ったときの余りを代入先に格納します
pow	pow <オペランド A><オペランド B><代入先> オペランド A を B でべき乗し代入先に結果を格納します
and	and <オペランド A><オペランド B><代入先> オペランド A と B の論理積を代入先に格納します
or	or <オペランド A><オペランド B><代入先> オペランド A と B の論理和を代入先に格納します
xor	xor <オペランド A><オペランド B><代入先> オペランド A と B の排他的論理和を代入先に格納します
not	not <オペランド><代入先> オペランドの否定ををを代入先に格納します
cmp	cmp <オペランド A><比較演算子><オペランド B><代入先> オペランド A と B とを比較して結果が比較演算子と一致したかどうかを BOOL 型の値で代入先に格納します
int	int <変数名><変換するもの>

命令	文/処理
	値を INT 型に変換して変数に代入します
float	float <変数名><変換するもの>
	値を FLOAT 型に変換して変数に代入します
bool	bool <変数名><変換するもの>
	値を BOOL 型に変換して変数に代入します
str	str <変数名><変換するもの>
	値を STR 型に変換して変数に代入します
cat	cat <文字列 A><文字列 B><代入先>
	2 つの文字列を結合して代入先に格納します
goto	goto <ラベル>
	ラベルの位置にジャンプします
when	when <判定値><ラベル>
	判定値が true ならラベルの位置にジャンプします
dump	dump
	命令実行時点で登録されている変数とラベルのリストを表示します（デバッグ専用）

Iceberg ではスクリプトの文字列を何度もパースする手間を省くため、実際にスクリプトを実行する前に内部用の表現（中間表現）に変換しています。本来ならこの処理のときに変数の型を確定すればよいのですが現在の実装では変数の型の処理を実行時に行っています。つまり中間表現生成時には変数に **T_UNDET** という値を印として割り当てておいて、実行時にその印がある引数を変数として扱い実際の値と型を求めるようにしています。

T_UNDET が何かを説明しましたがもう一つ **T_ANY** というものもあります。これも Iceberg 処理系の内部で使うもので「ある型でない」ことを表現するためのものです。処理系に **get_argument()** という引数の型をチェックしてその値を取得する関数があるのですが、ここで例えば **STR** 型を引数の型として禁止したいときに、引数の型として **T_ANY xor T_STR**[6] を指定することで **STR** 型以外の引数だけ通すことができます。

以上が仕様の大体の説明です。ここでソースコードの解説も入れようと思ったのですが間に合いませんでした、すみません。ソースコードは GitHub 上で公開[7]します。何か進展があれば更新するかもです[8]。なお、Iceberg の処理系はパッケージとして使うことができ、上位のプログラムから処理系を動かしたり機能を追加したりできるようになっています。例えば命令を追加することができます（後述）。

1.3 使ってみる

Iceberg を使って FuzzBuzz 問題を解いてみました。以下がそのソースコードです（コード 1.3）。

```
let cnt_now, 1
let cnt_fizz, 0
let cnt_buzz, 0
@loop
    cmp cnt_fizz, "<", 2, b_not_fizz
    cmp cnt_buzz, "<", 4, b_not_buzz
    let s_mes, ""
    when b_not_fizz, @not_fizz
        cat s_mes, "Fizz␣", s_mes
        let cnt_fizz, -1
    @not_fizz
    add cnt_fizz, 1, cnt_fizz
    when b_not_buzz, @not_buzz
        cat s_mes, "Buzz", s_mes
        let cnt_buzz, -1
    @not_buzz
    add cnt_buzz, 1, cnt_buzz
    bool b_show_mes, s_mes
    when b_show_mes, @show_mes
        str s_mes, cnt_now
    @show_mes
    print s_mes
```

[6] 型はビットフラグになっている

[7] https://github.com/H-Teramura/iceberg

[8] T_UNDET なんて Python 版の実装にないじゃないか！と言われそうですがそうなんです、T_UNDET は Go 版を書いたときに追加したものです。しかし一応この点では 2 つの処理系間に互換性が残るようにしてあるので許してください

```
    add cnt_now, 1, cnt_now
    cmp cnt_now, "<=", 10000, b_continue
    when b_continue, @loop
```

ソースコード 1.3: Iceberg のコード

結果は長いので省略…

print 命令って何やねん！と思ったでしょう。実はこれは上位のプログラムから追加した命令です[9]。これによって Iceberg プログラムから元のアプリケーションの機能を呼び出すことができるのです。追加の仕方は次のベンチマーク用コード（上位プログラムとして機能する）を参照してください。

これを 100 から 10000 回まで伸ばして実行してみると私の環境[10]ではだいたい 0.441 秒かかりました。ベンチマークに用いたコードも載せておきます（コード 1.4）。

```
import iceberg
import time

class testVM(iceberg.IcebergVM):
    def inst_print(self, args):
        operand = self.get_argument(args[0], iceberg.T_STR)
        print(operand)

    def add_other_symbols(self):
        self.const_opcode['print'] = iceberg.InstructionDesc(self.inst_print, 1)

vm = testVM()

with open('main.ib') as f:
    src = f.read()

print("Compiling...")

c_time_s = time.time()
bytecode = vm.gen_bytecode(src)
c_time_d = time.time() - c_time_s

print("Running␣FizzBuzz...")

e_time_s = time.time()
vm.run(bytecode)
e_time_d = time.time() - e_time_s

print("Result:")
print("Compilation␣time", c_time_d, "sec.")
print("Execution␣time", e_time_d, "sec.")
```

ソースコード 1.4: ベンチマーク

1.4　もっと早い実装

Python はソースコードを CPU で実行できる機械語には変換しませんが、C 言語などの CPU 用の機械語を作ることができる言語で実行ファイルを作ると実行速度が良くな

[9]他にも処理系内の変数を上位プログラムから確認することもできる

[10]Intel Core i7-7500U,8GB RAM, インターフェース不明 SSD（いっぱいではない）,Linux OS 64bit

ることがあります。そんなわけで Iceberg を Go 言語[11]を用いて実装してみました。処理系のソースコードは先程の GitHub リポジトリにあります。

先程と同様の環境で 10000 回 FuzzBuzz を実行してみたところ実行時間がだいたい 0.206 秒になり実行速度は 2.14 倍になりました。処理系のコードをもっと良くすればさらに高速にできるような気もしますが[12]。ベンチマーク用のコードはこちらです（コード 1.5）。

```
package main

import(
    "fmt"
    "io/ioutil"
    "./iceberg"
    "time"
)

type testVM struct{
    iceberg.IcebergVM
}

func (vm *testVM) inst_print(args []iceberg.Entity) {
    operand, _ := vm.Get_argument(args[0], iceberg.T_STR)
    fmt.Println(operand.(string))
}

func (vm *testVM) start() {
    vm.Init()
    vm.Inst_table["print"] = iceberg.InstructionDesc{ vm.inst_print, 1, }
}

func main() {
    byte_temp, err := ioutil.ReadFile("main.ib")
    if err != nil {
        fmt.Println("Load␣Failed")
        return
    }
    vm := testVM{iceberg.IcebergVM{}}
    vm.start()
    script := string(byte_temp)

    t0 := time.Now()

    bytecode := vm.Gen_bytecode(script)

    vm.Run(bytecode)

    t1 := time.Now()
    fmt.Printf("Execution␣time(indluding␣compilation):␣%v␣ms\n",
        int64(t1.Sub(t0) / time.Millisecond))
}
```

ソースコード 1.5: もっとベンチマーク

実はこの実装にはまずいところがあります。**mod** 命令を **FLOAT** 型を引数にとって

[11]この言語をチョイスした理由は「使ってみたかった」
[12]バンバンハッシュテーブルを使わないようにしたい

実行したときに 2 つの実装の間で結果に差が出てしまうのです。実は Python では剰余演算が整数でも浮動小数点数でも定義されるのに対し Go の剰余演算がオペランドが整数型であるときのみ定義されているので Go で Iceberg を実装するときに浮動小数点数での剰余演算をする処理を作ってやる必要があります。しかしその処理をまだ実装していないので動作が異なってしまうわけです。というわけで今の所 Iceberg で浮動小数点数同士の剰余計算をしたときの動作は未定義です。

1.5　おわりに

最初は簡単な仕組みにしたつもりでしたが細かいところを考えると意外と複雑になり実装に時間もかかりました。でも、とりあえず動作するものを作ることができたのでまあ良かったことにしておこうと思います。Iceberg はまだ仕組みや実装にたくさんの問題があるので[13]もっと勉強してとりあえず使えるところまでレベルアップできたら良いなと思っています。

自分で作った言語が自作の処理系で動作するのはこの上なく楽しいことです。今まで躊躇していた人も一つ作ってみてはいかがでしょうか。では、また。

[13]たとえば main.ib を見ると分岐を作りたいときにジャンプ先が「xxx でないとき」になる傾向が強いので人間に優しいとは言えないし、ガベージコレクタもまだ実装していない

参考文献

[1] Python 3.6.x Documentation `https://docs.python.org/3/`

[2] The Tour of Go `https://tour.golang.org/`

2 立体４目並べ

情報工学課程 1回生 川島友希

2.1 はじめに

今回、コマンドラインで重力付き立体４目並べができるものを作成しました。コンピュータの AI も搭載しています。開発に使用した言語は C++です。

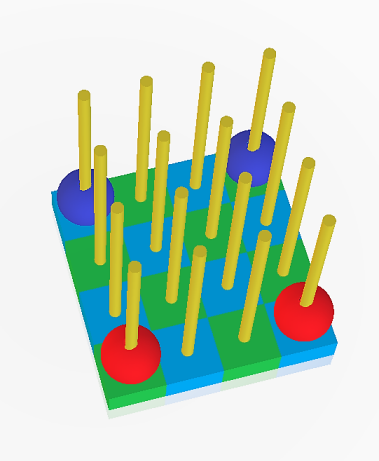


図 2.1: 立体四目並べのイメージ

2.2 ルール

ゲームの方針は普通の５目並べとだいたい同じで、２人交互に駒を置いていき、先に自分の駒を４つ並べたら勝ちになります。しかし、このゲームの特徴的なところは４×４×４の立体空間上にあるフィールドです。縦横斜め対角線計７６ラインのどこに４つ揃えてもよく、これがこのゲームの醍醐味になります。また、重力付きということで空中に浮かせて駒を置くことはできません。パスはできず、禁じ手もありません。

2.3 ゲームの流れ

ゲームは以下の流れで進みます。

1) ターンを進める。

2) 駒を置く。

3) 勝敗引き分けの判定。決着まだなら 1) へ戻る。

4) 終了。

以下は、上のループを実装した main 関数です。クラスや関数の解説は次の章でします。

```
int main(){
	//process before game start
	srand((unsigned int)time(NULL));
	std::cout << "game␣start" << std::endl;
	Player player;
	player.set_player();
	Field field;
	std::cout << "input␣:␣column␣row␣(both␣1~4)" << std::endl;
	while(1) {
		//process at turn start
		field.show_field();
		forward_turn(player, field);
		std::cout << "turn␣" << field.get_turn() << std::endl;
		std::cout << "+++␣" + player.get_name() + "'s␣turn␣+++"
		    << std::endl;
		//put piece
		while(!field.put_on_field(input(player, field)))
			std::cout << "***␣you␣can't␣put␣there.␣***" << std::endl;
		//check if win
		if(field.check_game()) {
			field.show_field();
			std::cout << "+++␣" << player.get_name() << "␣win␣+++"
			    << std::endl;
			break;
		}
		//check if draw
		if(field.get_turn() == CellNumber) {
			field.show_field();
			std::cout << "+++␣draw␣+++" << std::endl;
			break;
		}
	}
	return 0;
}
```

ソースコード 2.1: main

2.4 コード解説

ゲームに使う定数です。

```
static const int EdgeLength = 4; //1 辺の長さ、4 目並べの 4
static const int Square_of_EdgeLength = 16; //EdgeLength^2
static const int CellNumber = 64; //ゲーム盤面 6 4 マス
static const int LineNumber = 76; //ライン 7 6 本
```

ソースコード 2.2: constant value

2.4.1 クラス

このプログラムでは、Player クラスと Field クラスを作りました。

Player クラス

２人のプレイヤーの種類（人 or コンピュータ）と名前を持ち、ターン開始時や勝敗決着時の宣言や駒を置き方を決めるのに使います。プレイヤーの種類は列挙型で管理しています。

以下 Player クラスの宣言。

```
enum PlayerID {
       NULLID,
       HUMAN,
       COMPUTER1,
       COMPUTER2,
};

class Player {
       PlayerID id1;
       std::string name1;
       PlayerID id2;
       std::string name2;
       bool turn_flag; //true...first false...second

public:
       Player();
       void set_player();
       PlayerID get_PlayerID() const;
       std::string get_name() const;
       void reverse_flag();
};
```

ソースコード 2.3: class Player

Field クラス

ターン数、盤面の状態など、ゲーム進行に欠かせない変数を持ち、ゲームの主要な部分を担当します。field は各マスの情報を持ち、盤面の表示に使います。level は縦向きの１６本のラインにいくつ駒が積まれたかを持ち、field へのアクセスにループを使わないようにします。linearr は７６本のラインの情報を持ち、勝敗の判定、盤面の評価に使います。

以下 Field クラスの宣言。

```
class Field {
       struct Line {
              int first_point, second_point;
       };

       int turn;
       std::array<int, CellNumber> field;
       std::array<int, Square_of_EdgeLength> level;

public:
```

```
        std::array<Line, LineNumber> linearr;
        Field();
        void show_field() const;
        bool put_on_field(const int&);
        int get_turn() const;
        void addturn();
        void update_line(const int&);
        int evaluate() const;
        bool check_game();
};
```

ソースコード 2.4: class Field

以下メンバ関数の実装。

```
Player::Player()
        : id1(NULLID), name1("player1"),
        id2(NULLID), name2("player2"), turn_flag(0){
}

//プレイヤーの情報をセット
void Player::set_player(){
        std::cout << "1...human␣␣␣2...computer␣Lv.1␣␣␣3...computer␣Lv.2"
            << std::endl;

        int tmp;
        std::cout << "player1␣:␣";
        while(!(std::cin >> tmp) || (tmp < 1 || 3 < tmp)) {
                std::cin.clear();
                std::cin.ignore(1000, '\n');
                std::cout << "***␣imput␣error␣***" << std::endl << "player1␣:␣";
        }
        switch(tmp) {
        case 1: id1 = HUMAN; name1 = "player"; break;
        case 2: id1 = COMPUTER1; name1 = "computer␣Lv.1"; break;
        case 3: id1 = COMPUTER2; name1 = "computer␣Lv.2"; break;
        }

        std::cout << "player2␣:␣";
        while(!(std::cin >> tmp) || (tmp < 1 || 3 < tmp)) {
                std::cin.clear();
                std::cin.ignore(1000, '\n');
                std::cout << "***␣imput␣error␣***" << std::endl << "player2␣:␣";
        }
        switch(tmp) {
        case 1: id2 = HUMAN; name2 = "player"; break;
        case 2: id2 = COMPUTER1; name2 = "computer␣Lv.1"; break;
        case 3: id2 = COMPUTER2; name2 = "computer␣Lv.2"; break;
        }

        if(id1 == id2) {
                name1 += 'A';
                name2 += 'B';
        }
}

//そのターンのプレーヤーの種類・名前を取得
PlayerID Player::get_PlayerID() const {
```

```
        if(turn_flag)
                return id1;
        else
                return id2;
        return NULLID;
}
std::string Player::get_name() const {
        if(turn_flag)
                return name1;
        else
                return name2;
        return "";
}

//ターンの経過
void Player::reverse_flag(){
        turn_flag = !turn_flag;
}

Field::Field() : turn(0), field{}, level{}, linearr{}{
}

//盤面を表示
void Field::show_field() const {
        std::cout << std::endl;
        for(int z = EdgeLength - 1; z >= 0; --z) {
                std::cout << "␣␣␣␣␣1234" << std::endl;
                for(int y = 0; y < EdgeLength; ++y) {
                        int i = EdgeLength - y - 1;
                        while(i--)
                                std::cout << '␣';
                        std::cout << y + 1 << '/';
                        for(int x = 0; x < EdgeLength; ++x) {
                                switch (field[x + y*EdgeLength
                                    + z*Square_of_EdgeLength]) {
                                case 0: std::cout << '*'; break;
                                case 1: std::cout << 'O'; break;
                                case -1: std::cout << 'X'; break;
                                }
                        }
                        std::cout << '/' << std::endl;
                }
                std::cout << std::endl;
        }
}

//入力に応じてメンバを更新
bool Field::put_on_field(const int& place){
        if(level[place] != EdgeLength) {
                int position = place + level[place]*Square_of_EdgeLength;
                field[position] = turn % 2 ? 1 : -1;
                ++level[place];
                update_line(position);
                return 1;
        }
        return 0;
}
```

```
int Field::get_turn() const {
	return turn;
}

void Field::addturn(){
	++turn;
}

//ラインの更新。力業。
void Field::update_line(const int& position){
	std::function<void(const int&)> f;
	if(turn % 2)
		f = [&](const int& i){
			++linearr[i].first_point;
		    };
	else
		f = [&](const int& i){
			++linearr[i].second_point;
		    };
	switch(position) {
	case 0: f(0); f(16); f(32); f(48); f(56); f(64); f(72); break;
	case 1: f(0); f(17); f(33); f(65); break;
	case 2: f(0); f(18); f(34); f(66); break;
	case 3: f(0); f(19); f(35); f(52); f(60); f(67); f(74); break;
	case 4: f(1); f(16); f(36); f(57); break;
	case 5: f(1); f(17); f(37); f(48); break;
	case 6: f(1); f(18); f(38); f(52); break;
	case 7: f(1); f(19); f(39); f(61); break;
	case 8: f(2); f(16); f(40); f(58); break;
	case 9: f(2); f(17); f(41); f(52); break;
	case 10: f(2); f(18); f(42); f(48); break;
	case 11: f(2); f(19); f(43); f(62); break;
	case 12: f(3); f(16); f(44); f(52); f(59); f(68); f(73); break;
	case 13: f(3); f(17); f(45); f(69); break;
	case 14: f(3); f(18); f(46); f(70); break;
	case 15: f(3); f(19); f(47); f(48); f(63); f(71); f(75); break;
	case 16: f(4); f(20); f(32); f(49); break;
	case 17: f(4); f(21); f(33); f(56); break;
	case 18: f(4); f(22); f(34); f(60); break;
	case 19: f(4); f(23); f(35); f(53); break;
	case 20: f(5); f(20); f(36); f(64); break;
	case 21: f(5); f(21); f(37); f(49); f(57); f(65); f(72); break;
	case 22: f(5); f(22); f(38); f(53); f(61); f(66); f(74); break;
	case 23: f(5); f(23); f(39); f(67); break;
	case 24: f(6); f(20); f(40); f(68); break;
	case 25: f(6); f(21); f(41); f(53); f(58); f(69); f(73); break;
	case 26: f(6); f(22); f(42); f(49); f(62); f(70); f(75); break;
	case 27: f(6); f(23); f(43); f(71); break;
	case 28: f(7); f(20); f(44); f(53); break;
	case 29: f(7); f(21); f(45); f(59); break;
	case 30: f(7); f(22); f(46); f(63); break;
	case 31: f(7); f(23); f(47); f(49); break;
	case 32: f(8); f(24); f(32); f(50); break;
	case 33: f(8); f(25); f(33); f(60); break;
	case 34: f(8); f(26); f(34); f(56); break;
	case 35: f(8); f(27); f(35); f(54); break;
```

```
        case 36: f(9); f(24); f(36); f(68); break;
        case 37: f(9); f(25); f(37); f(50); f(61); f(69); f(75); break;
        case 38: f(9); f(26); f(38); f(54); f(57); f(70); f(73); break;
        case 39: f(9); f(27); f(39); f(71); break;
        case 40: f(10); f(24); f(40); f(64); break;
        case 41: f(10); f(25); f(41); f(54); f(62); f(65); f(74); break;
        case 42: f(10); f(26); f(42); f(50); f(58); f(66); f(72); break;
        case 43: f(10); f(27); f(43); f(67); break;
        case 44: f(11); f(24); f(44); f(54); break;
        case 45: f(11); f(25); f(45); f(63); break;
        case 46: f(11); f(26); f(46); f(59); break;
        case 47: f(11); f(27); f(47); f(50); break;
        case 48: f(12); f(28); f(32); f(51); f(60); f(68); f(75); break;
        case 49: f(12); f(29); f(33); f(69); break;
        case 50: f(12); f(30); f(34); f(70); break;
        case 51: f(12); f(31); f(35); f(55); f(56); f(71); f(73); break;
        case 52: f(13); f(28); f(36); f(61); break;
        case 53: f(13); f(29); f(37); f(51); break;
        case 54: f(13); f(30); f(38); f(55); break;
        case 55: f(13); f(31); f(39); f(57); break;
        case 56: f(14); f(28); f(40); f(62); break;
        case 57: f(14); f(29); f(41); f(55); break;
        case 58: f(14); f(30); f(42); f(51); break;
        case 59: f(14); f(31); f(43); f(58); break;
        case 60: f(15); f(28); f(44); f(55); f(63); f(64); f(74); break;
        case 61: f(15); f(29); f(45); f(65); break;
        case 62: f(15); f(30); f(46); f(66); break;
        case 63: f(15); f(31); f(47); f(51); f(59); f(67); f(72); break;
        }
}

//盤面の評価
int Field::evaluate() const {
        const std::array<int, 10> weight{0,5,100,200,10000,-10,
            -100,-1000000,-100000000,1};
        int value = 0;
        for(int i = 0; i < LineNumber; ++i) {
                int first = linearr[i].first_point;
                int second = linearr[i].second_point;
                if(first && second)
                        value += weight[0];
                else if(first)
                        value += weight[first];
                else if(second)
                        value += weight[second + 4];
                else
                        value += weight[9];
        }
        return value;
}

//勝利判定
bool Field::check_game(){
        if(turn % 2) {
                for(int i = 0; i < LineNumber; ++i)
                        if(linearr[i].first_point == EdgeLength)
                                return 1;
```

```
        }
        else{
                for(int i = 0; i < LineNumber; ++i)
                        if(linearr[i].second_point == EdgeLength)
                                return 1;
        }
        return 0;
}
```

ソースコード 2.5: class

2.4.2 関数

入力系統は非メンバ関数にて実装しています。

```
//クラス内のターンを進める
void forward_turn(Player& player, Field& field){
        player.reverse_flag();
        field.addturn();
}

//プレイヤーの種類に応じた入力方法を適用
int input(const Player& player, Field& field){
        int value = 0;
        switch(player.get_PlayerID()) {
        case HUMAN:
                value = human_input();
                break;
        case COMPUTER1:
                value = computer_input(5, field);
                break;
        case COMPUTER2:
                value = computer_input(7, field);
                break;
        default: break;
        }
        return value;// value = 0...15
}
```

ソースコード 2.6: function1

このゲームでは、駒を置ける場所は常に 1 6 か所以下しかないので、駒を置く場所を指定するのに必要な入力は縦横のマス目番号だけで、高さは必要ないようにしています。実は、Field クラスの level はこれを実現するためにあったりします。

```
//人間の入力はエラーが出ないように処理
int human_input(){
        int x,y;
        while(!(std::cin >> x >> y) || (x < 1 || 4 < x || y < 1 || 4 < y)) {
                std::cin.clear();
                std::cin.ignore(1000, '\n');
                std::cout << "***␣input␣error␣***" << std::endl;
        }
        --x;
        --y;
        return x + y*EdgeLength;
```

```
}
```

ソースコード 2.7: function2

コンピュータの手は、[1] [2] を参考にミニマックス法系列のネガアルファ法で先読み探索しています。評価関数は４つ駒を揃えられる７６ラインの状態に応じて点数をつけるといった単純なものですが、そこそこ強いです。

```
int computer_input(const int depth, const Field& field){
	int max = INT_MIN;
	std::array<int, Square_of_EdgeLength> arr{
	    INT_MIN, INT_MIN, INT_MIN, INT_MIN,
	    INT_MIN, INT_MIN, INT_MIN, INT_MIN,
	    INT_MIN, INT_MIN, INT_MIN, INT_MIN,
	    INT_MIN, INT_MIN, INT_MIN, INT_MIN
	};
	std::vector<int> candidate;
	for(int p = 0; p < Square_of_EdgeLength; ++p) {
		Field nextfield = field;
		if(!nextfield.put_on_field(p))
			continue;
		if(nextfield.check_game())
			arr[p] = INT_MAX;
		else
			arr[p] = -nega_alpha(nextfield, depth-1, INT_MIN, INT_MAX);
		max = (max < arr[p]) ? arr[p] : max;
	}
	for(int p = 0; p < Square_of_EdgeLength; ++p)
		if(arr[p] == max)
			candidate.push_back(p);
	int i = rand() % (int)candidate.size();
	std::cout << candidate[i] % EdgeLength + 1 << '␣'
	    << candidate[i] / EdgeLength + 1 << std::endl;
	return candidate[i];
}

int nega_alpha(const Field& field,const int depth, int alpha,const int beta){
	if(field.get_turn() == CellNumber || !depth)
		return field.evaluate();
	for(int p = 0; p < Square_of_EdgeLength; ++p) {
		Field nextfield = field;
		nextfield.addturn();
		if(!nextfield.put_on_field(p))
			continue;
		if(nextfield.check_game())
			return INT_MAX;
		int score = -nega_alpha(nextfield, depth-1, -beta, -alpha);
		if(alpha < score)
			alpha = score;
		if(alpha >= beta)
			return alpha;
	}
	return alpha;
}
```

ソースコード 2.8: function3

実装は以上。以下、実行結果。

```
game start
1...human 2...computer Lv.1 3...computer Lv.2
player1 : 1
player2 : 2
input : column row (both 1~4)

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/****/
 2/****/
3/****/
4/****/

turn 1
+++ player's␣turn␣+++
1␣1

␣␣␣␣␣1234
␣␣␣1/****/
␣␣2/****/
␣3/****/
4/****/

␣␣␣␣␣1234
␣␣␣1/****/
␣␣2/****/
␣3/****/
4/****/

␣␣␣␣␣1234
␣␣␣1/****/
␣␣2/****/
␣3/****/
4/****/

␣␣␣␣␣1234
␣␣␣1/O***/
␣␣2/****/
␣3/****/
```

```
4/****/

turn␣2
+++␣computer␣Lv.1's turn +++
2 3

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/O***/
 2/****/
3/*X**/
4/****/

turn 3
+++ player's␣turn␣+++
4␣1


(中略)

turn␣35
+++␣player's turn +++
4 4

    1234
  1/****/
 2/****/
3/****/
4/****/

    1234
  1/****/
 2/*XXO/
3/*OOO/
4/*OOO/

    1234
  1/*XO*/
 2/*OXX/
3/*XXO/
```

```
4/*XXO/

    1234
   1/OXOO/
  2/*OXO/
 3/OXXX/
4/XXXO/

turn 36
+++ computer Lv.1's turn +++
3 2

    1234
   1/****/
  2/**X*/
 3/****/
4/****/

    1234
   1/****/
  2/*XXO/
 3/*OOO/
4/*OOO/

    1234
   1/*XO*/
  2/*OXX/
 3/*XXO/
4/*XXO/

    1234
   1/OXOO/
  2/*OXO/
 3/OXXX/
4/XXXO/

+++ computer Lv.1 win +++
```

ソースコード 2.9: result

2.4.3 改善点（案）

- コマンドラインだと盤面の状態が分かりにくい。（そのせいでよく負ける。）そのために、アプリとして動くようにする。
- 1 手（あるいは 2 手）戻す機能をつける。
- AI の強さを上げる。より良い評価関数、または他の方法（機械学習とか）をとる。

2.5 おわりに

まだまだ改善すべきところは多いですが、クラスとかラムダ式とか入れて自分なりに一応満足したものが書けたと思います。最後までお読みいただきありがとうございま

した。

参考文献

[1] ネガα法 | 机上のにゅーろん [http://spheresofa.net/blog/?tag=%E3%83%8D%E3%82%AC%CE%B1%E6%B3%95]

[2] アルファ・ベータ法− wikipedia [https://ja.wikipedia.org/wiki/%E3%82%A2%E3%83%AB%E3%83%95%E3%82%A1%E3%83%BB%E3%83%99%E3%83%BC%E3%82%BF%E6%B3%95]

3 すごいPrologつくって学ぼう?!

情報工学課程 3 回生 松永大輝

3.1 プロローグ

Prolog というプログラミング言語があります。C 言語や BASIC が手続き型言語、Java や Smalltalk がオブジェクト指向言語、OCaml や Haskell が関数型言語と呼ばれるのに対して、Prolog は論理型言語と呼ばれています。Prolog は、命題の証明を背景とする実行の仕組みやバックトラック、論理変数、ユニフィケーションなど面白い特徴を多く備えています。

この記事では、そんな Prolog の処理系をつくった話をします。巷に溢れるコンパイラやインタプリタの本は、実装される言語が手続き型や関数型言語であるものがほとんどで、論理型言語を扱っているものはあまり見ません。ですが、特に WAM(後述) は非常によく考えられていて、役に立つ立たないとか関係なく面白い話題であると思います。

3.2 Prolog の実行形態

Prolog 処理系の実現方法として、まずインタプリタが挙げられます。以前私も Prolog インタプリタを実装しました[1]。ユニフィケーションや変数の管理で特に苦労したように思います (もっと良いやり方があったのかもしれませんが...)。その他には、中間言語に変換して仮想マシンで実行したり、C 言語のソースコードへ変換してから機械語にコンパイルしたりといった方法が挙げられます。

様々な方法がありますが、多くの Prolog 処理系は Warren's abstract machine(WAM) をベースにしています。WAM は David H. D. Warren によって考案された、Prolog の実行のために考えられた抽象機械で、Prolog の複雑な仕組みをうまく実現しています。

この記事ではそんな WAM の解説を主に行いたいと思います。

3.3 WAM

3.3.1 WAMって何?

WAM が Prolog の実行のための抽象機械であると 3.2 節で言いました。「こんな感じの命令セットとレジスタ、メモリがあれば Prolog の複雑な実行メカニズムを効率よく実現できるんじゃないの」と Warren さんが考えたものが Warren's Abstract Machine、WAM です。Prolog プログラムは述語・質問単位で WAM 命令にコンパイルされ、仮想マシンで実行されます。また、Prolog の処理に特化したハードウェア (Prolog マシン)

[1] https://github.com/matsud224/petit-prolog

で WAM の影響を受けているものもあります。

WAM のメモリレイアウトとレジスタを図 3.1 に示します。プログラムカウンタやスタック、ヒープ位は耳にしたことがあるかも知れませんが、トレイルやバックトラックレジスタなど聞き慣れない名前がいくつか出てきます。徐々にこれらの説明をしていくので、時々この図を見返してみてください。

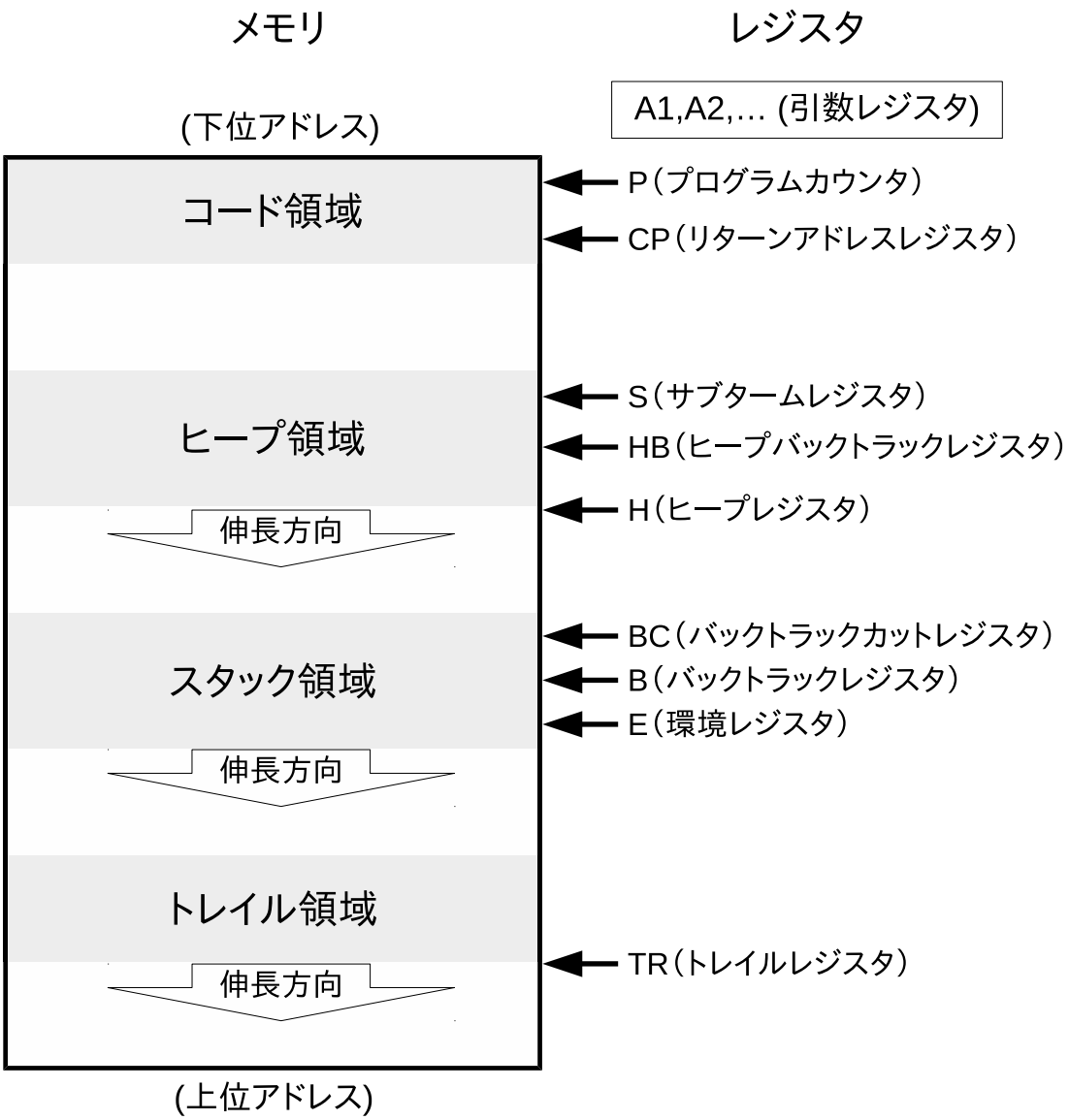


図 3.1: WAM のメモリレイアウトとレジスタ

3.3.2 レジスタとメモリ

まず、メモリとレジスタの使い方についていくつか決めておきます。メモリにはアドレスが振られていて、メモリ 1 単位にはタグと値の組が格納できるものとします。タグは、表 3.1 に挙げたものをを使用します。また、引数レジスタにもメモリと同様にタグ

と値の組が格納できるとします。その他のレジスタにはアドレスを格納できます。

タグ	値	意味
REF(reference)	アドレス	メモリへの参照
CON(constant)	アトム or 数値	定数
STR(structure)	構造体の先頭アドレス	構造体への参照
SST(structure start)	構造体名/引数の数	構造体の開始

表 3.1: タグとその意味

例えば、構造体 a(1,X) は表 3.2 のようにメモリ上に置かれます。ここで、X は未束縛変数であるとします。未束縛変数は自分自身を参照することで表現されます。

アドレス	タグ	値
100	SST	a/2
101	CON	1
102	REF	102

表 3.2: 構造体 a(1,X) のメモリ上での表現

Prolog は未束縛変数同士のユニフィケーションが可能で、どちらか一方にもう一方の変数への参照が代入されます。すると、変数同士のユニフィケーションにより何重にも参照が連鎖することが考えられます。定数もしくは構造体・リストへの参照、もしくは未束縛変数に到達するまで参照を辿る操作をデリファレンスといいます。WAM では、変数に対してまずデリファレンス操作を行い、その中身を判断します。

3.3.3 ユニフィケーション

まずはユニフィケーションから始めましょう。

```
p(X, bar, a(bar,X)).
```

という事実 p があるとき、

```
?- p(foo, X, a(X,foo)).
```

と質問してみます。これは `X=bar` となります。

事実 p と質問はそれぞれ図 3.2 と図 3.3 のような WAM 命令にコンパイルされます。なお、左端の数字は行番号です。%から行末まではコメントで、Prolog プログラムとのおおまかな対応を示しています。

基本的には、呼び出し側が put-*命令で引数レジスタに値をセットして、呼ばれた側が get-*命令で引数レジスタを検査するという流れです。引数レジスタは A1, A2, ..., An と表記し、順番に第 1 引数, 第 2 引数, ..., 第 n 引数の授受に使用します。引数レジスタの個数についてはここでは無視します。

get-*や unify-*命令はレジスタ・メモリ上の値を見てユニフィケーションの成功/失敗を判定します。ここでレジスタ・メモリ上の値同士のユニフィケーション操作が行われ、値の比較や未束縛変数への束縛が行われるのです。また、構造体は put-struct 命令に続く set-*命令、get-struct 命令に続く unify-*命令で扱います。

```
1: get-variable A4,A1   % p(X,
2: get-constant bar,A2  %   bar,
3: get-structure a/2,A3 %   a(
4: unify-constant bar   %     bar,
5: unify-value A4       %     X)
6: proceed              %   ).
```

図 3.2: 事実 p の WAM 命令

```
1: put-constant foo,A1  % ?- p(foo,
2: put-variable A4,A2   %      X,
3: put-structure a/2,A3 %      a(
4: set-value A4         %        X,
5: set-constant foo     %        foo)
6: call p/3             %      ).
```

図 3.3: 質問の WAM 命令

上記の例を見てみます。まず質問のゴール側 (呼び出し側) で、引数レジスタ A1 に 1 番目の引数を、A2 に 2 番目の引数を... と引数を順番にセットしていきます。一番な簡単なものとして、1 行目の put-constant foo,A1 はレジスタ A1 に定数 foo をセットします。

2 行目の put-variable ではレジスタに (未束縛) 変数をセットしています。引数レジスタはゴールの呼び出しが起こる度に使用 (上書き) されます。変数は後に束縛が起きる可能性があるため、それでは困ります。そのため、変数はレジスタに直接置かれるのではなくヒープ領域またはスタック領域にその領域が確保されます。スタック領域に確保する場合は後で説明するとして、ここではヒープ領域に確保を行います。ヒープレジスタ H は使用済みヒープ領域の末尾を指すレジスタです。2 行目の put-variable A4,A2 は、まず H レジスタをインクリメントして、ヒープ領域に変数 1 つ分の領域をとります。そしてそれを未束縛状態にして、そのメモリへの参照を引数レジスタ A4 と A2 に代入します。引数は 3 つしか無いのに A4 を使ってもいいの?、と思われたかもしれません。引数の個数を超えた引数レジスタは自由な用途に使えます[2]。

3-5 行目は構造体 a(X,foo) の処理です。構造体もヒープ領域に置かれます。まず 3 行目の put-structure a/2,A3 でヒープ上に構造体の開始の印 (SST) を置きます。A3 にはその構造体への参照 (STR) が入ります。set-*命令は構造体の引数をヒープ領域に置くために put-structure に続いて使用されます。まず変数 X を置くわけですが、X は既出で、A4 にはすでに変数 X への参照が入っています。そのため set-variable ではなく set-value を使います。そして定数 foo を置いて、call p/3 で述語 p/3 を呼び出します。

ここまでで、ひとまず呼び出し側の処理は終わりです。図 3.4 は 5 行目実行直後のレジスタとヒープの様子です。

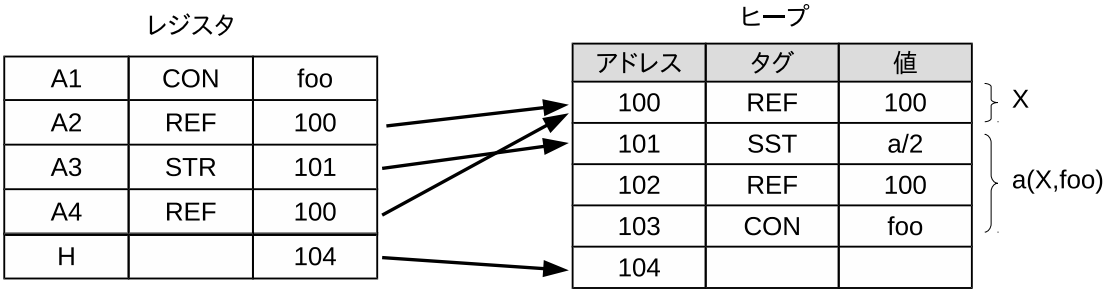


図 3.4: 5 行目実行直後のレジスタとヒープの様子

さて、今度は呼ばれた側 (事実 p) の WAM 命令を見ていきます。引数レジスタを get-*

[2] ここで A4 にも代入していますが、実はこれは無駄なので最適化で改善されるべき点なのです...

命令で順番に見ていきます。1 行目の get-variable A4,A1 は単純に A1 の値を A4 に代入しています。2 行目の get-constant bar,A2 では A2 すなわち第 2 引数と bar でユニフィケーションを行います。A2 が未束縛変数であった場合は A2 に bar を束縛します。A2 が未束縛変数でなく、定数 bar でもない場合はユニフィケーション失敗なので、後述するバックトラックが発動されます。

構造体の場合は、get-structure の後に unify-*命令が続く形と なります。3 行目の get-structure a/2,A3 は A3 すなわち第 3 引数が構造体 a/2 もしくは未束縛変数であるか調べます。

もし構造体 a/2 がすでにヒープ領域にあるなら READ モードへ移行します。サブタームポインタ S が構造体のある場所を指すようにして、後続の unify-*命令で S をインクリメントしながらヒープ領域上の構造体の引数についてユニフィケーションを行います。

もし A3 が未束縛変数だったなら WRITE モードへ移行します。後続の unify-*命令ではヒープ領域に構造体を構築していきます。このように、get-structure によって選択されたモードによって続く unify-*命令の動作が変わるのです。ユニフィケーションは双方向であることを考えるとそうなるのも納得がいきます。

今回の質問`?- p(foo, X, a(X,foo)).` では構造体を置いているので、READ モードとなります。4 行目の unify-constant bar ではヒープ上の構造体 a/2 の第 1 引数 (アドレスは 102) を見ます。対象が違うだけで get-constant と同様の動作です。

5 行目の unify-value A4 では A4(変数 X) とヒープ上の構造体 a/2 の第 2 引数 (アドレスは 103) とをユニフィケーションします。このようなレジスタ・メモリ上に置かれた値同士のユニフィケーションについても、Prolog のユニフィケーション規則と同様です。例えば、CON foo と CON foo は成功、CON foo と CON bar は失敗、REF 10(未束縛変数) と CON foo の場合は 10 番地を foo にして成功、のような感じです。未束縛変数同士のユニフィケーションの場合は一方のアドレスを他方へ代入することになるのですが、参照の方向がアドレスが大きい方から小さい方となるようにします。これはダングリングポインタとなりうる、ヒープからスタックへの参照ができないようにするためです。

今回の質問では、ユニフィケーションに成功します。ユニフィケーション終了後のレジスタとヒープの様子を図 3.5 に示します。

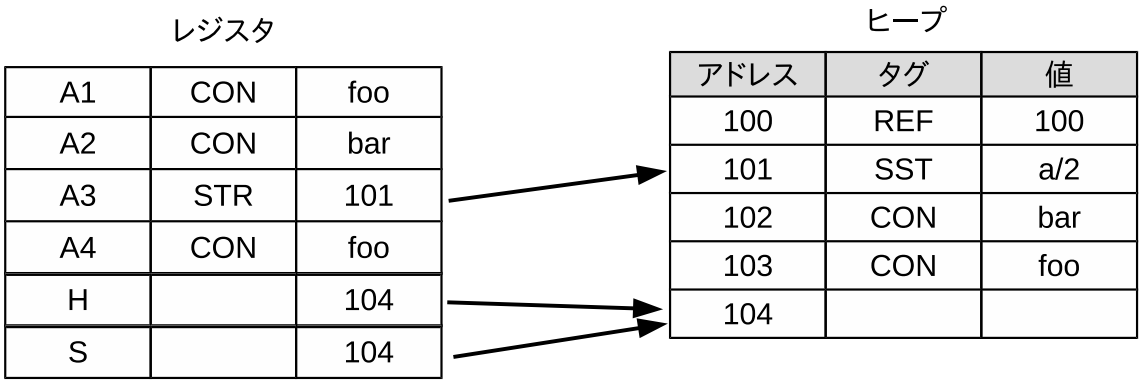


図 3.5: ユニフィケーション終了後のレジスタとヒープの様子

もし、質問が`?- p(foo, X, Y).` だったとすると、WRITE モードとなります。この時の unify-*命令は set-*命令と同じ動作となり、呼び出し側で構造体 a(bar,X) をヒー

プに構築します。

3.3.4 環境

ここでは、規則 (節でボディをもつもの) を扱えるようにします。

```
a(...) :- b(...), c(...).
```

この規則 a を WAM 命令にコンパイルすると、

```
(a の get/unify 命令列)
(b の put/set 命令列)
call b
(c の put/set 命令列)
call c
```

こんな感じになります。ボディの数だけ put/set と call を並べるだけです。

ところがここで、ある問題が発生します。次の規則 p をご覧ください。

```
p(X,Y) :- q(Y), r(X),s(Z).
```

変数 X は引数レジスタ A1 から参照されているとします。そのとき、r(X) の呼び出しの時に A1 に X への参照がある保証はありません。q の呼び出しで引数レジスタが上書きされてしまうからです。

そこで環境というものを導入します。これは C 言語のスタックフレームのようなものだと思ってください。これまでは変数をヒープに確保して引数レジスタから参照していたのですが、変数の領域をスタック上の環境内にとることで上記の問題を解決します。環境内に置かれる変数は、要するに C 言語のローカル変数みたいなものです。ここではこのような変数を恒久変数と呼びます。それと対比して、ヒープ上に取られる変数を一時変数と呼びます。ある変数が恒久変数として環境内に確保されるか、一時変数としてヒープに確保されるかはコンパイル時に決定されます。その基準は、「ボディで、複数のゴールにまたがって出現する変数は恒久変数、それ以外は一時変数とする。」です。上記の規則 p では、X が恒久変数、Y と Z が一時変数です。Y はヘッドとボディの両方に出現していますが、ヘッドでの出現はカウントしません。

環境について詳しく見ていきます。まず、現在の環境は環境レジスタ E が指しています。環境の生成は allocate 命令、破棄は deallocate 命令を使用します。allocate 命令ではスタック領域に新たな環境を確保・初期化し、E がそれを指すようにします。deallocate 命令では E が 1 つ前の環境を指すようにします。Prolog の変数のスコープは節内であるため、節が呼ばれる度に環境がスタック上にとられます。WAM コードとしては、節のはじめに allocate 命令、終わりに deallocate 命令を置いてやります。そして、環境は表 3.3 のような構造となっています。E+2+n で n 番目の恒久変数にアクセスできます。それぞれの環境は 1 つ前の環境のアドレスを持っているため連結リストになっています。

もうひとつ、リターンアドレスについて補足しておきます。call 命令では、別の節へ移ったあとに戻ってくるためにリターンアドレスをリターンアドレスレジスタ CP にセットしています。そして、事実の場合は proceed 命令が CP を P に代入して復帰します。規則の場合は、CP が書き換えられる可能性があるためリターンアドレスは環境内に退避しておきます (表 3.3 参照)。そして、deallocate 命令で環境からリターンアドレスを P へ代入して復帰します。

E	1 つ前の環境のアドレス
E+1	リターンアドレス
E+2	恒久変数の数
E+3	1 つ目の恒久変数
E+4	2 つ目の恒久変数
E+5	3 つ目の恒久変数
...	...

表 3.3: 環境の構造

```
1: allocate 1            %
2: get-variable Y1,A1    % p(X,Y) :-
3: put-value A2,A1       % q(Y)
4: call q/1              % ,
5: put-value Y1,A1       % r(X)
6: call r/1              % ,
7: put-variable A1,A1    % s(Z)
8: call s/1              % .
9: deallocate            %
```

図 3.6: 規則 p の WAM 命令

先ほどの規則 p は図 3.6 のような WAM コードにコンパイルされます。まずは 1 行目の allocate 1 で恒久変数 1 個が入る環境を確保します。2 行目には Y1 という表記がされていますが、これは 1 番目の恒久変数を示しています。get-variable Y1,A1 は A1 から Y1 に代入を行います。Yn へは E からのオフセットでアクセスがなされます。9 行目の deallocate 命令で、E でアクセスできる 1 つ前の環境のアドレスを E にセットします。要するに、E が 1 つ前の環境を指すようにします。

3.3.5 バックトラック

あとバックトラックができれば、もう Prolog です。この節では、以下のように OR 関係にある節を並べることができるようにします。

```
a(foo).
a(bar).
a(baz).
```

バックトラックは、ユニフィケーションが失敗するなどの要因で起こります。あるゴールに対して候補節が複数あってそのうちの一つを選ばなければならないとき、その時点をチョイスポイントと呼びます。バックトラックが起きると、もっとも最近に生成されたチョイスポイントに戻り、次候補を選択します。Prolog では定義順で選択を行います。バックトラック時にチョイスポイントに戻ってきて次の候補節を選択できるようにするためには、その時点でのレジスタの値や次の候補節などの情報を保持しておかなければなりません。

```
    try_me_else L1
    < a(foo). のコード >
L1: retry_me_else L2
    < a(bar). のコード >
L2: trust_me
    < a(baz). のコード >
```

図 3.7: 述語 a/0 の WAM 命令

まず、チョイスポイントに戻ってこれるようにするためにチョイスポイントフレームというものを導入します。チョイスポイントフレームには、チョイスポイントでのレジスタの値や次候補節などの情報を格納します。バックトラック時には直近のチョイスポイントへ戻るので、チョイスポイントフレームは生成するごとに積み重ねていきます。

チョイスポイントフレームの構造を表 3.4 に示します。

B	引数の個数 n
B+1	チョイスポイントでの A1 レジスタ
B+2	チョイスポイントでの A2 レジスタ
...	...
B+n	チョイスポイントでの An レジスタ
B+n+1	チョイスポイントでの E レジスタ
B+n+2	チョイスポイントでの CP レジスタ
B+n+3	1 つ前のチョイスポイントフレームのアドレス
B+n+4	次候補節のコードの開始アドレス
B+n+5	チョイスポイントでの TR レジスタ
B+n+6	チョイスポイントでの H レジスタ

表 3.4: チョイスポイントフレームの構造

節のはじめの述語 a/0 の WAM コードは図 3.7 のような感じになります。まず try-me-else 命令でチョイスポイントフレームを生成します。別候補の選択が起こった場合は、L1 から実行が行われます。L1 や L2 はラベルです。retry-me-else 命令は途中の節で、trust-me 命令は最後の節で使用します。retry-me-else 命令はチョイスポイントフレームを更新します。trust-me 命令はチョイスポイントフレームを破棄します。

チョイスポイントフレームは一体どこに置かれるのでしょうか。図 3.1 を見てみると、トレイル領域というのが余っているのでそこかと思ってしまいますが、実はスタック領域に置かれるのです。つまり、環境とチョイスポイントフレームがスタック領域に混在することになります。一番新しいチョイスポイントフレームを指しているのがバックトラックレジスタ B です。

環境とチョイスポイントフレームはそれぞれが連結リストを形成しているので、ひとつのスタック領域で混在していても独立に辿ることは可能です。しかし、スタックからポップする際に問題が起こります。例えば、環境-チョイスポイントフレーム-環境-チョイスポイントフレーム-環境の順に並んでいるときに環境をポップすると隙間ができてしまいます。

それではなぜ環境をチョイスポイントフレームをそれぞれ独立した領域に置かないの

でしょうか。それにはちゃんと理由があります。私はこれを知った時に、すごいなぁと思いました (小並感)。

以下のプログラムと質問を使用して、環境とチョイスポイントフレームを独立した領域に配置した際に起こる問題を見てみましょう。

```
a :- b(X),c(X).
b(X) :- b1(X).
c(X) :- c1(X).
b1(1).
b1(2).
c1(2).
c1(3).

?- a.
```

まず a が呼ばれて a の環境が環境スタックに乗ります。b(X) が呼ばれ、b の環境を生成します。b1 のチョイスポイントを生成した後、b1(X) と b1(1) のユニフィケーションで X=1 となります。b の環境が破棄され、次は c(X) が呼ばれます。c の環境を生成し、c1 のチョイスポイントを生成した後、c1(X) と c1(2) とのユニフィケーションに失敗します (X は 1 に束縛されているため)。この時点では図 3.8 のようになっています。ここでバックトラックが起こり、B レジスタが指している c1 のチョイスポイントフレームを見て、レジスタの復元と次候補節へのジャンプを行います。c1(3) が選択されますが、再び失敗し、バックトラックします。c1 の候補節はもう無く、c1 のチョイスポイントフレームは破棄されており B は b1 のチョイスポイントフレームを指しています。そして b1 のチョイスポイントの次候補節 b1(2) へジャンプします。と、ここで大変なことに気が付きます。b の環境がなくなっているのです！b1 は b のボディで呼ばれているため、b の環境がないとダメです。

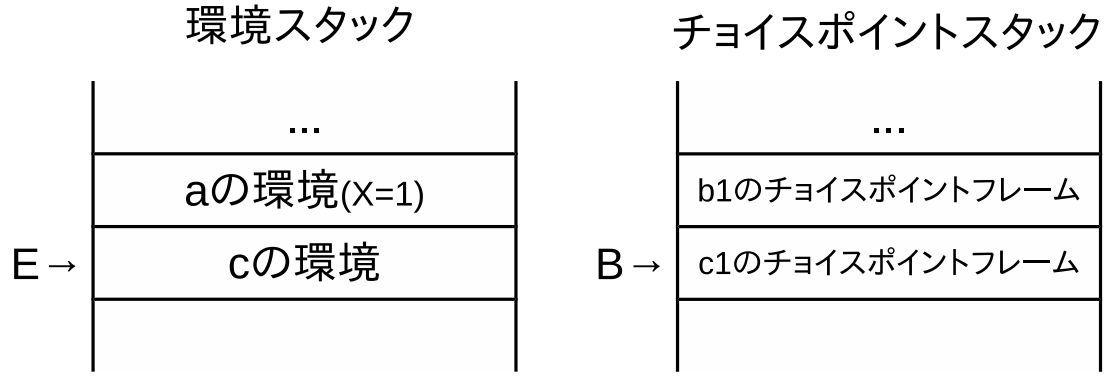


図 3.8: スタックを分ける (スタックは下へ伸びる)

チョイスポイントフレームを作っておきながらその時の環境は破棄されてしまうことが問題です。これを解決するためには、チョイスポイントフレームが存在している限りはその時点までの環境を残しておくことが必要です。そうであれば、環境とチョイスポイントフレームを混在させるとこれがシンプルに解決できます。

b1 のチョイスポイントフレーム生成後に c の環境を確保しようとしたとします。E が B よりも下 (アドレス値が大きい) の時には、これまでどおり E が指す環境の後ろに新

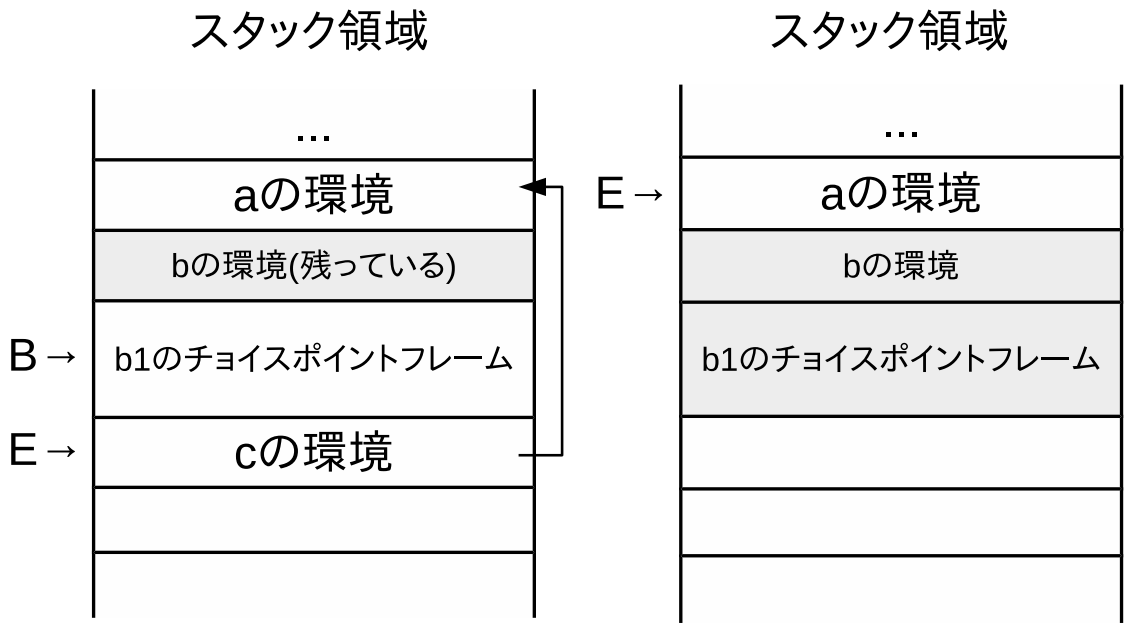


図 3.9: スタックを一緒に使う (左:c の環境確保後、右:b1 のチョイスポイントフレーム破棄後)

たな環境を確保します。今回は、b の環境を破棄した後なので、E は a を指しています。B は b1 のチョイスポイントフレームを指しています。このように B が E よりも下 (アドレス値が大きい) ときは、新たな環境は B が指すチョイスポイントフレームの後ろに確保します。チョイスポイントフレームはそれまでに生成された (下位アドレスの) 環境を保護しているのです。c の環境を確保した後のスタック領域の様子が図 3.9 の左側です。

c1 が失敗して c の環境の deallocate が起こると、E は a の環境を指します。c の環境の確保時には E は a の環境を指していたからです。再び b に戻ってきて、このときチョイスポイントフレーム内に保存していた E(b の環境を指す) が復元されます。b1(2) が選択されたら、b1 にはもう候補節はないため、b1 のチョイスポイントフレームは破棄されます。この状態 (図 3.9 の右側) で c の環境の allocate が行われると、b の環境があった領域が再利用されます。すなわち、環境とチョイスポイントフレームを混在させてはいますが、チョイスポイントフレームが有効な間は必要な環境が保護され、不要になったらちゃんと再利用されるのです。

3.3.6 束縛の取り消し

前の節ではひとつ大事なことを忘れていました。バックトラックでチョイスポイントに戻る際、レジスタを戻すだけでは不十分です。チョイスポイントからバックトラック発生時点までの間に変数を束縛した場合にそれを未束縛状態に戻さなければなりません。

束縛した変数に関する情報の記憶にトレイル領域が使用されます。変数を束縛する度にトレイル領域にその変数のアドレスが格納されていきます。Prolog では変数は一度束縛されると再代入できないため、変数のアドレスを覚えておくだけで十分なのです。

トレイルレジスタ TR は使用済みトレイル領域の末尾の次のアドレスを保持しています。ある時点からある時点までの束縛を取り消したい時には、それぞれの時点での TR レジスタの値を覚えていおけば大丈夫です。その 2 つの TR レジスタが保持するアドレスの間にあるアドレスの変数を未束縛にすればいいだけです。

全ての変数束縛を記録しておいてもいいのですが、それは明らかに無駄です。バックトラック時には、B が指す直近のチョイスポイントフレームに戻ります。そして、E レジスタは B より下位アドレスの環境を指すことになります。ヒープについても、直近のチョイスポイントの H レジスタの値が復元されます。そのために、ヒープバックトラックレジスタ HB には直近のチョイスポイントでの H レジスタがセットされています。このように、バックトラック時にはチョイスポイントから現在まで使用されてきたスタック領域とヒープ領域は破棄されます。よって、チョイスポイント以前より存在していた変数への束縛だけをトレイル領域に記録しておけば十分です。具体的には、スタック領域の変数であればそのアドレスが B 未満、ヒープ領域の変数であればそのアドレスが HB 未満であるときに記録対象とします。

3.3.7 その他の話題

ここまででユニフィケーションやバックトラックなど Prolog の大事な部分が WAM においてどのように実現されているのか説明しました。

この他にもカットの導入や、末尾呼び出し最適化・環境刈り込み、インデキシングといった最適化手法の話がありますが、詳しくは参考文献 [1][2][4] をご覧ください。特に参考文献 [1] は WAM について非常に丁寧に解説しており、本記事の作成や処理系の実装の際、大いに参考にさせて頂きました。

3.4 処理系を作った

WAM バイトコードへコンパイルし WAM 仮想マシンで実行する Prolog 処理系を CommonLisp で実装しました。なお、ソースコードは公開[3]しております。この処理系は、上記で説明した基本的な Prolog の機能に加え、以下のような機能を実装しています。

- カット
- 環境刈り込み (⊃ 末尾呼び出し最適化)
- インデキシング (第一引数を見て候補節を絞り込む)

3.4.1 構文解析器

S 式を受け取るようにして構文解析をサボるという手もありますが、そこはやっぱり Prolog の文法で書けるようにしたいものです。Prolog の文法はシンプルなので構文解析器は比較的簡単に作れます。しかし、is/2 のような述語のサポートや op/3 組み込み述語による演算子定義を可能とするため、構文解析器をを工夫しました。具体的には、演算子名・優先順位・結合法則の 3 つ組が登録してある演算子テーブルを見て構文解析をできるようにしました。Prolog は結合法則の指定が充実していて、{ 右結合, 左結合,

[3] https://github.com/matsud224/wamcompiler

結合なし } と { 単項前置, 単項後置, 二項 } の組み合わせで 7 種類から選べます。非常に柔軟に演算子定義ができるのですが、実装する側としては辛いです。演算子順位構文解析法を試したりもしましたが、最終的に再帰下降構文解析法を工夫してこれを実現しました。

parse 関数が構文解析を行う関数です。

3.4.2 コード生成

compile-*関数でコンパイルを行います。その後、optimize-*関数などで WAM 命令レベルのちょっとした最適化を行います。例えば、リスト 3.1 を最適化するとリスト 3.2 となります。

```
get-constant [],A1
get-variable X4,A2
get-variable X4,A3
proceed
```

ソースコード 3.1: 最適化前の WAM コード

```
get-constant [],A1
get-value X2,A3
proceed
```

ソースコード 3.2: 最適化後の WAM コード

3.4.3 WAM 仮想マシン

参考文献 [1] の WAM 命令の疑似コードを参考に命令を実装しました。

3.4.4 REPL

Prolog は対話的に使用することができます。REPL を起動し質問を入力すると、解がある場合は解を表示し入力待ちとなります。本処理系では、

- y で終了
- n で次の解を表示
- a で全ての解を表示

という動作を行います。解がこれ以上ない場合は”no.”と表示されます。

REPL は repl 関数で実装しています。repl 関数は入力を読み取り、構文解析を行います。入力されたものが事実や規則であればコンパイルし、ハッシュテーブルに登録します。質問であればコンパイルし仮想マシンで実行します。

仮想マシンでの実行中に解が見つかったり失敗したりすると、REPL 側に制御を移します。単純に見つかった解をリターンするだけで良さそうですが、ここはちょっと面倒です。解が見つかったときには一旦 REPL に制御を移しますが、次の解を要求された場合は再度仮想マシンに制御を移し、**続きから**実行を行う必要があるのです。

続きから実行を行うのに、仮想マシンの状態をまるごと保存したり継続渡しスタイルで書いたりといった方法が考えられますが、今回は CommonLisp の Condition System を用いました。

Condition System は CommonLisp における例外処理機構で、一般的な言語のそれと比べて非常に強力です。Condition System には、例外ハンドラ側で原因を解決した後に例外発生箇所から実行を再開させる場面で用いられる restart という機能があります。これを今回は利用しました。なお、本処理系では通常の例外処理にも Condition System を用いています。Condition System については参考文献 [5] などをご覧ください。

3.4.5 組み込み述語

実用性を高めるためには組み込み述語が必要です。例えば、型判定の述語、数値計算用の述語、入出力を行う述語などです。

組み込み述語はある程度の数を定義することとなります。そのため、できるだけストレスなく定義できるとうれしいです。そこで、組み込み述語を定義しやすくするためのマクロや関数をいくつか書きました。例えば、整数ならば成功する integer/1 述語はリスト 3.3 のように定義できます。これは最も単純な述語のひとつであって、少し複雑な組み込み述語を定義しようとすると途端にしんどくなるので、まだまだ改良の余地があります。例えば、findall/3 はすべての解をリストにまとめる述語で、これはうまく Lisp で書けないので動的に WAM 命令を生成するという汚い実装となっています。

```
(define-prolog-builtin "integer" (x)
  (if (and (wamvalue-constant-p x) (integerp (wamvalue-content x)))
      (prolog-success)
      (prolog-fail)))
```

ソースコード 3.3: integer/1 組み込み述語の定義

現在、実装した組み込み述語を以下に示します。

- 型判定系... atom/1, atomic/1, number/1, integer/1, float/1, compound/1, var/1, nonvar/1
- 数値計算系... is/2[4], </2, >/2, >=/2, =</2, =:=/2, ==/2
- 出力系... write/1, nl/0
- その他... op/2, consult/1, findall/3, call/1, =../2, fail/0, halt/0

3.4.6 実行例

以下の実行例では、作成した Prolog 処理系で append/3 述語を定義し質問しています。その後、REPL を抜け show-wamcode 関数で append/3 述語がどのような WAM コードへコンパイルされたのかを表示させています。なお、"CL-USER>" は Lisp のプロンプト、"prolog>" は Prolog のプロンプトです。

```
CL-USER> (repl)
prolog> append([], Xs, Xs).
```

[4] 四則演算、剰余、べき乗、sqrt, sin, cos, tan... などを使える

```
prolog> append([X | Ls], Ys, [X | Zs]) :- append(Ls, Ys, Zs).
prolog> ?-append(X, Y, [k,i,t,c,c]).
X = NIL
Y = [k,i,t,c,c]
?a
X = [k]
Y = [i,t,c,c]

X = [k,i]
Y = [t,c,c]

X = [k,i,t]
Y = [c,c]

X = [k,i,t,c]
Y = [c]

X = [k,i,t,c,c]
Y = NIL

no.
prolog> ?-halt.

yes.
NIL
CL-USER> (show-wamcode "append" 3)
          switch-on-term G950,G954,G953,FAIL
G954:     switch-on-constant {[] => G952}
G950:     try-me-else G951
G952:     get-constant [],A1
          get-value X2,A3
          proceed
G951:     trust-me
G953:     get-list A1
          unify-variable X4
          unify-variable X1
          get-list A3
          unify-value X4
          unify-variable X3
          execute append/3
```

ソースコード 3.4: 実行例

3.5 エピローグ

「Prolog が面白い言語なら、Prolog 処理系作りも面白いのでは?」ということで、実際に作ってみたのですが、とても楽しかったです。Prolog インタプリタを作るのも楽しかったのですが、コンパイラだと生成された WAM コードを眺めて楽しめるので良いです。

参考文献

[1] Hassan Aït-Kaci, Warren's Abstract Machine: A Tutorial Reconstruction, MIT Press, http://wambook.sourceforge.net/

[2] David H.D.Warren, AN ABSTRACT PROLOG INSTRUCTION SET(日本語訳), http://www.takeoka.org/~take/ailabo/prolog/wam/wam.html

[3] 柴山 潔, 並列記号処理, コロナ社

[4] Kostis Sagonas, Logic Programming Implementation Part I: The WAM, http://www.it.uu.se/edu/course/homepage/logpro/ht04/LP_Impl.pdf

[5] 工藤千加子, 黒田寿男, Common Lisp における例外処理 -Condition System の活用-, 株式会社数理システム 知識工学部, http://cl-www.msi.co.jp/solutions/knowledge/lisp-world/tutorial/condition-system.pdf

4 単純な数式の計算

情報工学課程 1 回生 兼光 琢真

4.1 概要

私は数式を与えてやれば計算してくれる電卓のようなものを作ってみました。今回紹介するプログラムのおおまかな処理の流れは、入力により与えられた文字列が数式として扱えるならば、数や演算子などの最小単位に分割し、再帰下降構文解析法を用い、より処理がしやすい構造に変換した上で、適切な計算をしてから数値として表示する、という感じです。

4.2 解説

今回は Java を用いてプログラムを記述することにしました。このプログラムの中核を担っているのは再帰下降構文解析法なので、これこそを解説すべきなのでしょうが、ここで説明しますと私の理解度の低さも相まって長くなるので、興味をお持ちの方は適切な文献に当たって頂ければと思います。また、ソースコードを全て紹介するとこれも長くなるので、ここでは各クラスの紹介をした後に一部のクラスについてより詳細な説明をしていきます。

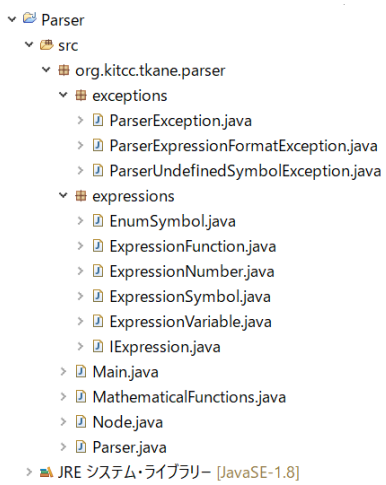


図 4.1: クラス一覧

ParserException

Exception を継承。この下の 2 クラスがこれを継承する

ParserExpressionFormatException
引数のない関数が呼び出されたり括弧が閉じていなかったりすると投げられる。

ParserUndefinedSymbolException
定義されていない定数、変数、関数名が評価時に存在したり無効な記号を検知すると投げられる。

EnumSymbol
利用可能な関数や演算子の名前と文字列が組で定義されている。文字列と EnumSymbol 型のお互いへの変換と関数か？演算子か？という判別が出来る。新しい関数を定義するときは、ここに追加しておくと後々便利に使える。

IExpression
クラス検査をしたりこれを実装したクラスが保有する情報を Object 型として返すメソッドなどが宣言されている。この下の 4 クラスがこれを実装する。

ExpressionFunction
ユーザが定義した関数についての情報を持つ。

ExpressionNumber
BigDecimal 型を持つ。初期化するときに文字列を与えると BigDecimal 型に変換し保持する。BigDecimal 型とは JavaAPI により提供されている任意精度演算が可能な型です。

ExpressionSymbol
EnumSymbol 型を持つ。適切な個数の引数を与えると結果を返すメソッドを持つ。

ExpressionVariable
ユーザが定義した変数についての情報を持つ。

Main
プログラムの起動時に始めに呼ばれる関数を持つ。Swing による GUI 生成や Event の登録などもここで一括して行う。

MathematicalFunctions
BigDecimal 型の演算を行うための一部の数学的関数などが用意されている。

Node
数式を構文解析によって変換した構文木を表せる。親、子ノードへの参照値を持ち自身の状態を示す IExpression 型も持つ。もともとは二分木のみを表現していたが、引数が 3 つ以上の関数を定義出来るように子ノードを任意の数だけ持つことができる木構造を採用することになった。

Parser
数式を表す文字列を与えて初期化すると同時に数式の木構造に変換する。数式の字句解析/構文解析が出来る。与えられた数式がユーザによる変数や関数の定義とみなせるならばそれも適当に処理する。実装当時、クラスの構成を深く考えていなかったために数式の評価までも受け持っていて、コードはぐちゃぐちゃに…。

4.2.1 Main クラス

Main 関数での Parser クラスを利用している部分を紹介します。これは static フィールドとして宣言された input を初期化し、細々とした設定をしているところです。5 行目から、キーボードから入力操作した際に呼び出されるリスナーを登録し、入力キーに応じた処理が行われるよう設定しています。Esc キーが押されたときは入力枠をクリア、Enter キーが押されたときは入力枠に書かれている文字列を読み取り Parser クラスを使って適当に処理した後に、返された数値を Main クラス内にある log 関数で出力するようになっています。

```
input = new JTextField();
input.setSelectionColor(Color.CYAN);
input.setBackground(Color.getHSBColor(0.63f, 0.22f, 1.0f));
input.setEditable(true);
input.addKeyListener(new KeyAdapter(){
    @Override
    public void keyTyped(KeyEvent e) {
        char keyCode = e.getKeyChar();
        if(keyCode == KeyEvent.VK_ESCAPE){
            input.setText("");
        }else if(keyCode == KeyEvent.VK_ENTER){
            long time = System.currentTimeMillis();
            formula = input.getText();
            try {
                Parser parser = new Parser(formula);
                BigDecimal result = parser.eval();
                if(result != null){
                    log(formula, "␣=␣",
                        result.setScale(Math.min(result.scale(), scale),
                            RoundingMode.HALF_EVEN).toString());
                }
                if(checkBoxIsDebugging.isSelected()
                    || checkBoxProcessingTime.isSelected()){
                    log("␣␣␣␣␣ProcessingTime␣",
                        System.currentTimeMillis() - time, "ms");
                }
            } catch (ParserException ex) {
                log(ex.getClass().getSimpleName(), "\n", ex.getMessage());
                ex.printStackTrace();
            }}}}); //はみ出さないように改行挟んでるので見づらくなってます
```

ソースコード 4.1: Main.java

4.2.2 Node クラス

このクラス型のオブジェクトは数式を分割した数や演算子などの最小単位を表すのに使われます。また、親 Node と子 Node への参照値を保持することで構文木が表現されます。

```
public class Node {
    private IExpression expr;
    private Node parent;
    private List<Node> childs = new ArrayList<Node>();
```

```
    public Node(IExpression expr){
        this.expr = expr;
    }

    public Node getRoot(){
        return parent == null ? this : parent.getRoot();
    }

    public void addChilds(List<Node> nodes){
        for(Node node : nodes){
            addChild(node);
        }
    }

    public void addChild(Node node){
        childs.add(node);
        node.setParent(this);
    }

    public IExpression getExpression(){
        return this.expr;
    }

    //その他の雑多なメソッドは省略

}
```

ソースコード 4.2: Node.java

4.2.3 Parser クラス

これのコンストラクタを呼び出すには引数として（数式として扱える）文字列型が必要です。コンストラクタに文字列が与えられると、アルファベットを全て小文字に揃え、空白記号を全て消し、適当な方法で数や演算子などの最小単位を表す IExpression 型を持つ Node にそれぞれ分割し、リストへ順番に入れていきます。

ここでの分割の仕方には次の点に注意しています。

- '-' が演算子ではなく負数の表現に用いられるとき'-1'，'*' への変換
- '(' の前が数か変数ならば'*' を間に補う
- その他必要とされる場所で'*' を間に補う

乗算記号の省略が可能になるため $f(x) = 10 \sin x$ という数式も宣言可能な一方で、現状では $f(x) = x \sin x$ などの比較的区別が難しい数式には対応しきれていません。

Node 型のオブジェクトに分割後、再帰下降構文解析法を用いて、演算の優先度などにも考慮して親、子 Node への参照値を設定していき木構造を表現します。次の図である関数における木構造への変換例を示します。

これを計算するには、上の Node から帰納的にチェックをしていきます。ある Node において、その子 Node のいずれかが数値でなければ、その子 Node をある Node として同様にチェックし、戻り値を数値として扱います。その後 Node が持つ演算子や関数などの記号に応じて計算してからその計算結果を数値として上に渡します。ここで、実

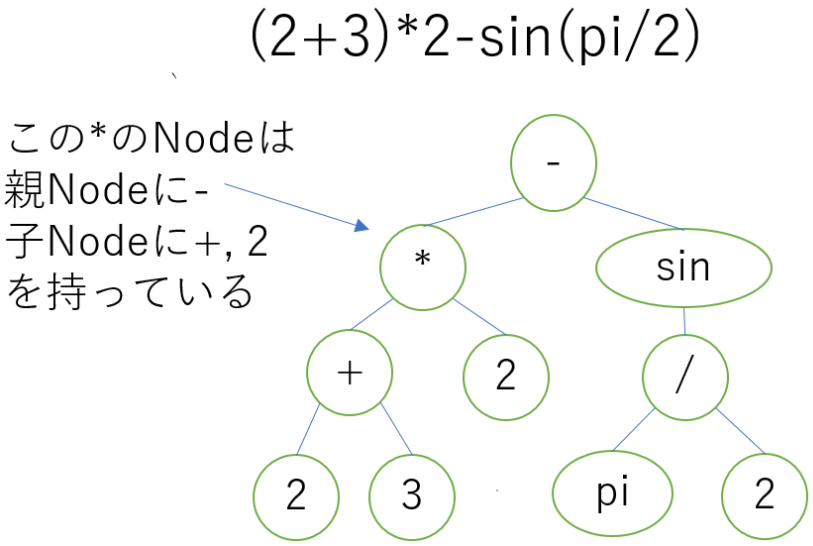


図 4.2: 数式から木構造の構築

際に使用している関数とは異なるものの、この例において以上の動作を満たす関数の一例を示します。

```
BigDecimal evalNode(Node node){
    if(node.isNumber()){
        return ((ExpressionNumber)node.getExpression()).getNumber();
    }
    List<BigDecimal> args = new ArrayList<BigDecimal>();
    for(Node child : node.getChilds()){
        args.add(evalNode(child));
    }
    return ((ExpressionSymbol)node.getExpression()).eval(args);
}
```

ソースコード 4.3: 木構造を順番に簡約化していく関数

この関数が行っていることを説明します。まず node が数値を示すならばその数値を返して終わりです。node が数値でなければ、node が示す演算記号に応じ、子 Node に対して同じ evalNode 関数を再帰的に適用させて返ってきた数値を用いて演算、その結果を返して関数の処理は終了します。

BigDecimal result = evalNode(一番上の Node);

と利用することで、木構造に問題が無ければ result に数式の答えが代入されます。これを先程の木構造に適応させてみると次のようになります。

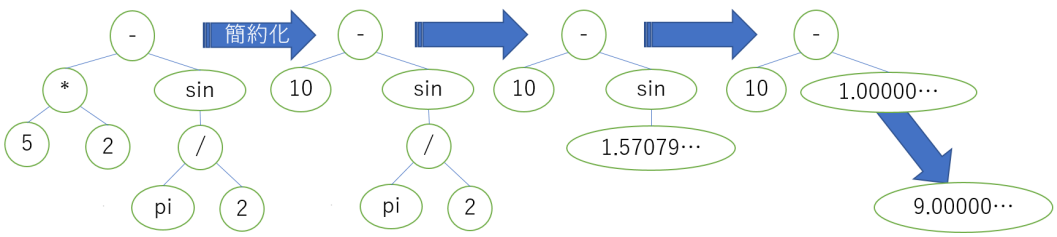


図 4.3: 木構造の簡約化

4.2.4 MathematicalFunctions クラス

ここには BigDecimal 型の数値演算のための関数が用意されています。それぞれの関数においての実装を紹介します。

$\sin x, \cos x$ の計算には周期性を利用して引数を $(0, \pi/2)$ の範囲に収め、Maclaurin 展開による級数表示を利用して適当な項で打ち切り近似しています。$\tan x$ も周期性を利用したあとに $\sin x / \cos x$ から近似値を算出しています。

$\exp x$ の計算には $x = \lfloor x \rfloor + (x - \lfloor x \rfloor)$ と分け、用意された e に $\lfloor x \rfloor$ 乗と、$x - \lfloor x \rfloor$ に対し級数展開を用いた近似値とを掛け合わせることで処理速度あたりでより精度の高い近似を可能としています。

$\log x$ の計算には $\frac{\log x}{2}$ の級数展開を利用しています。ただし大きな x に対しては収束が遅いので、$\log 2$ を先に計算しておき、$\log x = \log(2^n \cdot x\prime) = n \log 2 \cdot \log x\prime \quad (1 < x\prime < 2)$ として級数展開を用いた近似を $\log x\prime$ に適用させています。

これらは [1] を参考にしました。ここでは省きましたが誤差の評価までも論じられています。

逆三角関数は、まず $\arctan x$ について Arctan の加法定理を利用して x を収束しやすい範囲に収め、連分数展開による表示を利用して近似値を算出しています。その後 arcsin, arccos は arctan との関係式を利用して求めます。

その他、一般の実数の実数乗は exp と log の関数に帰着、ただし平方根は Newton 法による専用の関数 sqrt が用意されています。また、大きな数の階乗を計算する際は、[2] を参考に並列化させています。非整数の階乗は Lanczos 近似を用いた gamma 関数の近似を利用しています。今回は [3] に助けられ、gamma 関数を実装することができました。

4.3 仕様

プログラムを起動すると下にある画面が表示されます。上側に薄く水色がかったフィールドが数式入力する場所です。このフィールドの特徴としては Esc キーで記述した数式をクリアでき、Enter キーで数式を計算、その結果が下の Output Window に表示されます。下の出力枠は、最大で小数点以下 10 桁まで表示しますが、例えば整数同士の演算など、一定以上の精度が必要でないときは適切な桁数までしか表示されないようになっています。また、文字が下まで到達すると自動的にスクロールしてくれます。右側にある縦長いウィンドウは設定などを変更する場所として用意されています。現在の機能としては、スライドバーを調整することによるフォントサイズの変更や、数式計算の実行時間の表示、途中に行われた処理を書き出すデバッグモードへの切り替えがあります。

入力可能な数式の特徴

四則演算/累乗の左結合/括弧/数学的関数/定数 e、pi/積記号の省略/変数と関数の定義

数式に含まれる空白はすべて無視され、大文字小文字の区別はされません。

変数、関数の識別子として利用可能な記号はアルファベットとアンダースコアのみです。

現在利用可能な関数一覧

数学的関数の誤差は 1.0e-10 以下を目指しています。

max/min/step/sin/cos/tan/asin/acos/atan/sinh/cosh/tanh/asinh/acosh/atanh/
exp/log/sqrt/sigmoid/erf/erfc/factorial/gamma/loggamma/beta

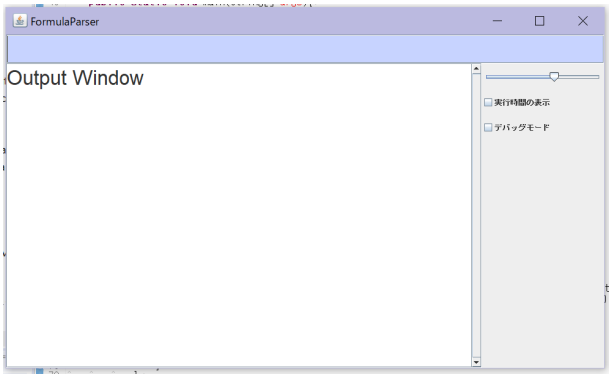


図 4.4: 起動直後の画面

論理的な関数は 0 を false、それ以外を true として扱います。
if/lt/le/eq/ge/gt/ne/and/or/not

4.4 利用例

計算可能な数式の例を次の図の左側に、変数と関数を利用したものを右側に示します。

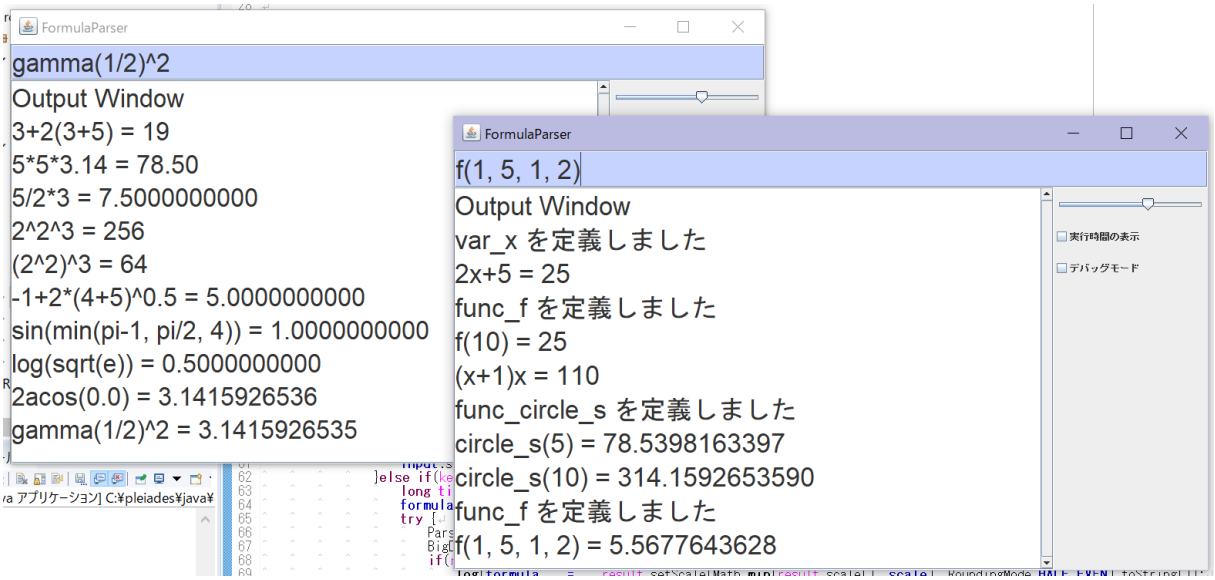


図 4.5: 出力画面 1

なお、右側の画面において入力された文字列は以下の通りです。

```
x = 10
2x+5
f(x) = 2x+5
f(10)
(x+1)x
circle_s(r) = pi*r^2
circle_s(5)
circle_s(10)
f(x, y, z, w) = sqrt(x^2+y^2+z^2+w^2)
f(1, 5, 1, 2)
```

ソースコード 4.4: 入力内容 1

定義の際注意されるべき点として、現在内部的には変数や関数と紐づけされるのは値ではなく木構造であるため、次のように入力すると、$x = 2x = 2 \cdot 2x = 2 \cdot 2 \cdot 2x = \cdots$ とのように再帰的に呼び出され、処理が終わらないのでエラーとなります。

```
x=2x
x
```

ソースコード 4.5: StackOverflowError

また、あり合わせ的に追加された if などの論理的な関数を用いると階乗関数などをユーザ側で定義が可能です。次の入力において最終行の factorial 関数は用意されている関数です。また if 以外の関数では、処理する前に引数を全て評価するのですが、if 関数ではそのようにすると延々と評価し続ける場合があるため、必要な引数のみ評価されるようになっています。

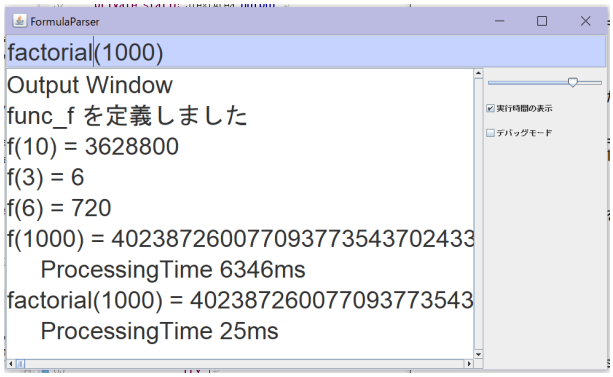


図 4.6: 出力画面 2

```
f(x) = if(le(x,0),1,x*f(x-1))
f(10)
f(3)
f(6)
f(1000) //実行時間の表示にチェックを入れた
factorial(1000)
```

ソースコード 4.6: 入力内容 2

4.5 最後に

私はこのプログラムを夏季休暇中に部内で行われたハッカソンに参加したときから作り始め、何とかですがこうして動くまでのものにすることが出来ました。かなりの時間を費やしましたが、その分だけ得られたものも多かったように感じられました。

しかしまだ勉強不足な点が多く、仕様を付け加えていくうちに、動作修正に時間がかかるようになってしまいました（全体的なコード整理をすれば良かったかも…）。私自身が書いたはずなのにこんな状態では、後々行き詰ってしまいそうですから、今後より多くの知識を得て、表現したい動作を素早く簡潔にコーディングできるようになりたいと思います。

以上となります。最後まで読んでくださりありがとうございました。

参考文献

[1] 『初等関数の計算』
https://na-inet.jp/nasoft/chap06.pdf

[2] 『factorial function - parallel processing』
https://stackoverflow.com/questions/10469552/factorial-function-parallel-processing

[3] 『Gamma.java』
http://introcs.cs.princeton.edu/java/91float/Gamma.java.html

(いずれも 最終閲覧日：2017/09/23)

5 ノベルゲームエディタ「Tale Master」

情報工学課程 1 回生 中森陸斗

5.1 はじめに

どうも。初めましての方は初めまして。情報工学課程の中森です。コンピュータ部に所属してから早くも 6 か月が経とうとしています。そこそこ部内の雰囲気にも慣れてきて、プログラミングにも慣れてくる... かと思いきや、まだ本腰を入れて使えるプログラミング言語はＨＳＰ（Hot Soup Processor）だけであるという、自分の勉強不足を呪う今日この頃です。前置きはこのぐらいにしておいて、本文に入りたいと思います。拙い文章ですが、最後まで読んでくだされば幸いです。

5.2 作ったものの概要

さて、今回私が作ったものは、簡単にノベルゲームを作れるエディタ「Tale Master」です。テキストファイルにノベルゲームの構造を入力し、このエディタで開くと、ノベルゲームになるというものです。なので正確に言えば、今回作ったものは言語処理プログラムの方だけなんですけどね。

このプログラムを作るために使った言語は、唯一僕が使える HSP という言語です。HSP には文字列操作の命令や関数がいろいろ入っているので、知っている言語が HSP で良かったです。

このプログラムの説明をします。今回は、「魔王討伐物語」というノベルゲームを作るとします。まず、メモ帳を開きます。(まあ、テキストエディタならなんでもいいです。) メモ帳を開いたら、そこにコマンドを箇条書きします。そして、Tale Master を開いて物語の名前を打ち込むと、自分で打ったコマンドが Tale Master に読み込まれ、ノベルゲームが始まります。

以下にコマンドを打ち込んだテキストファイルの内容を置いておきます。

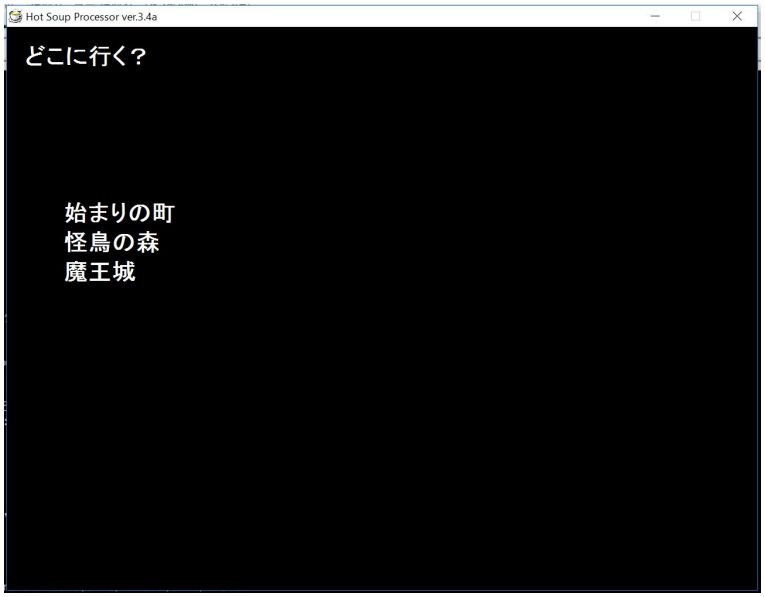
```
EVE start
    flag motimono, なし
    flag kaityou,1
    mes 魔王を倒しに行くぞ！
Eend

EVE select
    mes 持ち物は「＃ flag[motimono] ＃」です。
    sel どこに行く?,始まりの町,怪鳥の森,魔王城
    1:
        sgo mati
    2:
        sgo mori
```

```
    3:
        sgo siro
    Send
    go select
Eend

EVE mati
    mes 勇者は町に着いた！
    mes 老人／「この町では最強の拳を買うことができるぞ。」
    if motimono, お金
        Y/N 最強の拳を買いますか？
        Y:
            mes 勇者は最強の拳を手に入れた！
            flag motimono, 最強の拳
        Yend
        N:
        Nend
    Iend
    if motimono, なし
        mes 勇者は何もすることがない！
    Iend
    mes 町を去った！
Eend

EVE mori
    if kaityou,1
        mes 勇者は森に着いた！
        mes 怪鳥／「立ち去れ。」
        mes 勇者は怪鳥と戦った！
        mes 勇者は怪鳥に勝った！
        mes 勇者はお金を手に入れた！
        flag motimono, お金
        flag kaityou,0
        Eend
    Iend
    mes 勇者は森に着いた！
    mes 勇者は何もすることがない！
    mes 森を去った！
Eend

EVE siro
    mes 勇者は魔王城に着いた！
    mes 魔王／「ここまでよく来たな勇者よ...」
    mes 勇者は魔王と戦った！
    mes 魔王が攻撃してきた！
    if motimono, 最強の拳
        mes 勇者は拳で抵抗した！
        mes 勇者は魔王を倒した！
        mes 世界に平和が訪れた！
        TALEend
    Iend
    mes 勇者は死んでしまった！
    mes GAMEOVER
    TALEend
Eend
TEXTend
```

上のような形でコマンドを打ち込みます。それぞれのコマンドの意味は後で少し触れます。コマンド文の書き方は、Basic 系の言語に似ているかもしれません。これは、私が初めて学んだプログラミング言語が Basic 系のものだったので、そこから影響されたからかもしれません。まあ、コマンドのスペルは全然違うものになっていますが...。

5.3 動かしてみた

それじゃ、先ほどのテキストを Tale Master で起動してみます。すると...



このようになります。テキストの上にあるコマンドから順番に処理を行っていきます。写真の状態は、テキストファイルの 8 行目を処理しているところです。この後プレイを続けていっても、エラーなく終了することができました。この物語は、魔王を倒すために「最強の拳」というアイテムを持って魔王城に行くとクリアする、という物語です。アイテムを持たずに魔王城に行くとゲームオーバーとなります。

5.4 機能紹介

このエディタでできる基本的なことを紹介します。まず最初は、ノベルゲームには欠かせない文字表示です。コマンドは「mes」で、この後に表示する文字をいれて使います。

「/」文字で改行を入れられます。また、「＃」で文字を囲うと、その文字が特殊文字

扱いになって、flag コマンドによって作られたフラグの中身に置換できるようになっています。（ここの処理にかなり手間取った...）

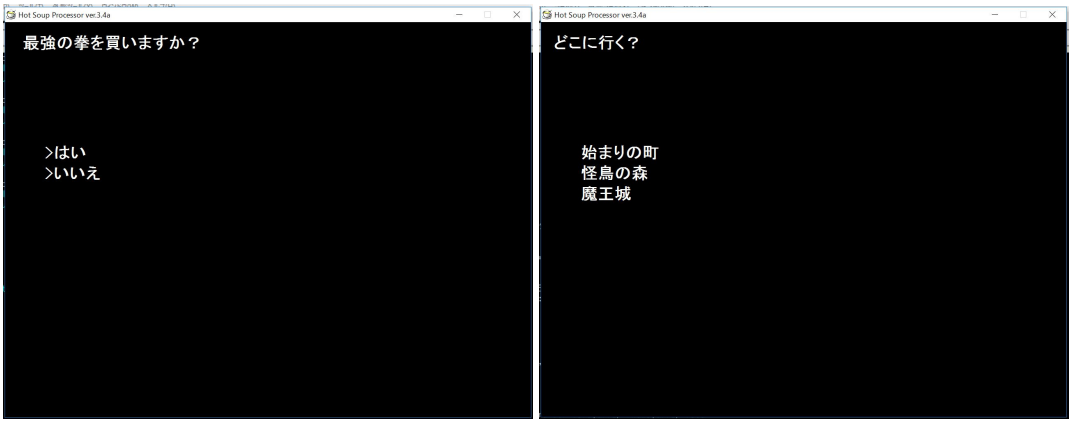
特殊文字付きメッセージの例

```
mes 持ち物は「＃ flag［motimono］＃」です。
```

これを実行すると...

このようになります。現在、フラグ「motimono」に代入されている値が「なし」なので、文章中の＃ flag[motimono] ＃の部分は「なし」に置換されます。（コマンド名が flag という名前なのに、中身に 0、1 以外の文字が入れられるのは、もともと 0、1 のみ入るフラグだったものに文字を入れられたら便利じゃね？と思い立ってしまって実装したからです。その名残で今もフラグという名前がついてしまっています。お許しください。）今のところ、使える特殊文字はフラグ置換だけなので、これからもっと機能を拡張していきたいです。

次に「Y/N」コマンドや、「sel」コマンドでプレイヤーによる分岐の説明をします。「Y/N」コマンドは、属に言う「はい、いいえ」の分岐です。「sel」コマンドはユーザーがボタンの名前を決めることができる分岐です。



左が「Y/N」コマンドで、右が「sel」コマンドの実行例です。実は、どちらも処理に少しごまかしが入っているのですが、まあ問題なく動いているのでいいでしょう。（プログラマーのクズ）

最後に、フラグの値による条件分岐「if」コマンドを説明します。これは、フラグの中身が、指定された文字列と同じかを評価し、同じであれば以下のコマンドを実行する

というものです。

「if」コマンドの例

```
if motimono，最強の拳
        mes 勇者は拳で抵抗した！
        mes 勇者は魔王を倒した！
        mes 世界に平和が訪れた！
        TALEend
Iend
mes 勇者は死んでしまった！
mes GAMEOVER
TALEend
```

これは、魔王城の中のコマンドの一部です。「if」コマンドによって、フラグ「motimono」の中身が「最強の拳」であるかどうかを評価しています。もし、フラグ「motimono」の中身が「最強の拳」でなかったとき、処理が「Iend」のところまで跳んで、ゲームオーバーになってしまいます。しかし、motimono の中身が「最強の拳」であった場合、「if」コマンドから「Iend」の間の処理を行って、ゲームクリアとなります。

5.5 Tale Master コマンド集

いままで紹介しなかったコマンドを含めた、Tale Master のコマンド表を表 5.1 に載せておきます。

表 5.1: Tale Master コマンド表

コマンド	処理内容
EVE	イベントの始まりを宣言する。
Eend	イベントの終わりを宣言する。
mes	文字の表示をする。
title	タイトルバーの文字を変える。
go	指定したイベントへと処理を移動する。
sgo	指定したイベントにジャンプして、イベントが終わったら元の場所に戻る。
pic	指定位置に画像を表示する。
bgm	BGM をループ再生する。
sound	効果音を鳴らす。
mstop	音声再生をストップする。
Y/N	はい、いいえ分岐を表示する。
sel	選択肢による分岐を行う。
flag	フラグの生成か、既存フラグに値を代入する。
input	キーボードから入力した文字をフラグに代入する。
list	大量の要素から選択したものをフラグに代入する。
if	フラグの値による分岐を行う。
TALEend	物語の終了を宣言する。
TEXTend	テキストファイルの終了を宣言する。

5.6 終わりに

「そうだ、ノベルゲームエディタを作ろう。」

こう思い立って、作り上げるのに半月ちょいくらいかかりました。最近は家に居るときはこのエディタをずっと作成していました。なんせバグが多いもので、一つのコマンドを作ると、必ず一つはバグが出てくるのでコーディングよりもはるかにデバッグのほうに時間がかかりました。まだまだ搭載したい機能も多いので、これからもがんばりたいです。さて、実はこのプログラム、完成させるにはもう一つのプログラムが必要なのです。そのプログラムの機能は、コマンドを選択すると、自動的にコマンドをテキストエディタに書き込むというものです。このプログラムが完成すれば、初めて「Tale Master」というエディタが完成するのです。こちらのプログラムはまだ手を付けていないので、また時間があれば作りたいと思います。また他には、HSP 以外の言語を使えるようになりたいので、他言語の勉強もしたいと思います。（やりたいこと増え過ぎて結局やらないやつ）

ここまで読んでいただきありがとうございました。

編集後記

編集担当の森下です。今回で 56 冊目の Lime になります。今回 Lime 担当としてあまり 1 回生にアドバイスなどは出来ませんでしたが、時間に限りがあるなかでも執筆者達はすばらしい記事を書いてくれました。僕自身も編集しながら目を通して、楽しんでいました。みなさんも楽しんでいただけていれば幸いです。今年は展示だけではなく店舗のほうも行っているのでどちらも楽しんでいってください。

最後にこの冊子を手にとって、最後まで目を通していただき、ありがとうございました。

平成 29 年 11 月 12 日
編集担当　森下　和馬

Lime Vol.56

平成 29 年 11 月 18 日 発行 第 1 刷

発行 京都工芸繊維大学コンピュータ部
http://www.kitcc.org/