オレオレ言語の開発

Daiki Matsunaga

2015年11月

1 はじめに

プログラミング言語はすでに星の数ほど存在していますが, やはり自作してみたくなるものです. この記事では, 3月〜6月頃に作ったプログラミング言語の説明を行います. 前半でその言語の特徴を, 後半で処理系の実装について説明します. ちなみに今回作った言語の名前は"soramame"です. 開発にはC++を使用しました.

2 soramame 言語ひとめぐり

soramame 言語をざっくりと紹介します. まずはお決まりのものを.

```
fun main(){
    print("hello,world")
}

---出力---
hello,world
```

この言語は手続き型言語です. 文法はCやJavaScriptを知っている人ならなんとなく雰囲気がつかめるであろうものとなっています. 実行は`main`関数から始まります.

データ型・変数

```
data Point{ x:int; y:int } //構造体宣言(トップレベルに記述)

var a:int=9
var b=5.1,c=[("java",true),("c",true),("python",false)]
```

```
var d=Point{x=12,y=200}
a=d.y;
print(c[2][0]) //python と出力される
b="kitcc" //コンパイルエラー「型に問題があります。二項演算子 = 左辺:double 右辺:string」が表示される
```

変数への再代入が可能です．静的型付き言語で, ローカル変数は型推論します．上の例では, b は double 型, c は string と bool のタプルのリスト ([(string,bool)]), d は Point 型であると推論します．基本型に int, double, bool, string があります．また, 複合型にリスト ([T], 1 つの型, 可変長), タプル ((T1,T2,...), 異なる型, 固定長), 構造体があります．その他に関数型 (fun(T1,T2,...)=>T), 継続型 (continuation(T)), チャンネル型 (channel(T)) があります.

クロージャ

```
fun makecounter()=> fun()=>void{
    var n=0
    return fun(){ n=n+1; print_int(n); return }
}

fun main(){
    var n=10
    var a=makecounter(),b=makecounter()
    a(); b(); a(); b()
}

---出力---
1122
```

関数は第一級オブジェクトです．つまり, 変数に代入したり, 引数や返り値にできるということです．main 関数では makecounter 関数から返ってきた関数 (a,b に代入されている) を呼び出しています．レキシカルスコープを採用しているので, 変数 n は無名関数が定義されたときのものが使われます．関数 makecounter からは関数ポインタではなく, クロージャが返ってきます．クロージャとは, 関数ポインタと環境のセットのことを言います．a,b それぞれ独立した環境を持っているため, カウントは独立して行われます.

継続

```
fun main(){
    var cont:continuation(void)
    print("do "); print("re "); print("mi ")
    callcc(c){ /*c に継続が代入されている*/ cont=c  }
    print("fa "); print("so "); print("ra ")
    cont()
}

---出力---
do re mi fa so ra fa so ra fa so ra fa so ra .....
```

継続も第一級オブジェクトです．継続というのは残りの計算をオブジェクトにしたものです．C で大域脱出に使われる `setjmp/longjmp` では内側からの脱出しかできませんが, 継続はどこからでも呼び出せます[*1]．上の例は, `do re mi` と表示されたあと, 変数 `cont` に継続を代入しています．`callcc` というのはコンピュータクラブではなく call-with-current-continuation の略で, Scheme に倣いました．これは継続を取り出すためのものです．Scheme では関数ですが, soramame では専用の構文となっています．`fa so ra` と表示された後に, `cont` に入れておいた継続を呼び出すのでまた `fa so ra` の表示が始まります[*2]．

並行実行・チャンネル通信

```
//ボール投げ
fun main(){
    var player=[newchannel(bool,1),
                    newchannel(bool,1),newchannel(bool,1)]
    var player1:fun()=>void = fun(){
        player[0]?
        print("Player 1 catch!\n"); sleep(3000)
        player[randitem([1,2])]!true
        player1()
    }
    var player2:fun()=>void = fun(){
        player[1]?
```

[*1] 別のスレッドからでも呼び出せます．
[*2] この後無限ループとなります．

```
        print("Player 2 catch!\n"); sleep(1000)
        player[randitem([0,2])]!true
        player2()
    }
    var player3:fun()=>void = fun(){
        player[2]?
        print("Player 3 catch!\n"); sleep(2000)
        player[randitem([0,1])]!true
        player3()
    }

    async player1(); async player2(); async player3()
    player[0]!true
    sleep(100000)
}

fun randitem(list:[int])=>int{
    return list[rand()%(@?list)]
}

---出力---
Player 1 catch!
(3 秒待つ)
Player 2 catch!
(1 秒待つ)
Player 1 catch!
(3 秒待つ)
Player 3 catch!
(2 秒待つ)
Player 2 catch!
(1 秒待つ)
Player 3 catch!
(2 秒待つ)
.....
```

```
//ストリーム
fun main(){
```

```
    var c1=newchannel(int,1)
    var c2=newchannel(int,1)
    var c3=newchannel(int,1)
    async intgen(c1,1)
    async map(fun(x:int){return x*x},c1,c2)
    async filter(fun(x:int){return x%17==0},c2,c3)

    print_int(c3?); print("\n")
    print_int(c3?); print("\n")
    print_int(c3?); print("\n")
}

fun intgen(chan:channel(int),start:int){
    chan ! start
    intgen(chan,start+1)
}

fun filter(f:fun(int)=>bool,in:channel(int),out:channel(int)){
    var value=in?
    if(f(value)){ out!value }
    filter(f,in,out)
}

fun map(f:fun(int)=>int,in:channel(int),out:channel(int)){
    out ! f(in?)
    map(f,in,out)
}

---出力---
289
1156
2601
```

この言語は並行処理を書きやすくするための機能を備えています．関数呼び出しの前に async というキーワードが付いている箇所がありますが，そうすることで別スレッドで関数が実行されます．Go 言語の goroutine は Go のランタイムが管理する軽量スレッドで，メモリ使用量が小さく大量に生成できるそうですが，これは単なるネイティブスレッドです．

プログラムが並列に実行されるにあたり，スレッド間のデータのやり取りや同期が重要となってきます．そ

のために soramame ではチャンネルというものが用意されています．チャンネルはスレッド間の多対多での通信手段を提供します．チャンネルを利用することによりセマフォやモニタのような排他制御も可能となります．このチャンネルは双方向であり, FIFO バッファを持ちます．　バッファのサイズは 0 以上でチャンネルオブジェクトの生成時に指定します．[**チャンネル**]?で受信, [**チャンネル**]![**値**] で送信です．受信時に値がなければブロックします．送信時にバッファがいっぱいだったらブロックします．

上記の 1 つ目の例はボール投げをシミュレーションするもので, 図 1 のような構造となっています．player1〜3 はそれぞれ, 自身のチャンネルにボール (ここでは値 true) が送られてくるのを待ちます．送られてくると, メッセージを表示し数秒待った後, 他の player をランダムに選びボールを送ります．そして自身を再帰的に呼び出すことでその動作を繰り返すようにしています．C 言語等ではこのような書き方をするとスタックオーバーフローになってしまいますが, soramame 言語では末尾呼び出し最適化が行われるため問題ありません (後述)．player1〜3 を起動し, player1 にはじめのボールを投げて開始すると, ボール投げの様子が出力されてゆきます．main 関数を抜けてしまうと終了してしまうため，　最後に余分な待ち時間を設けています．

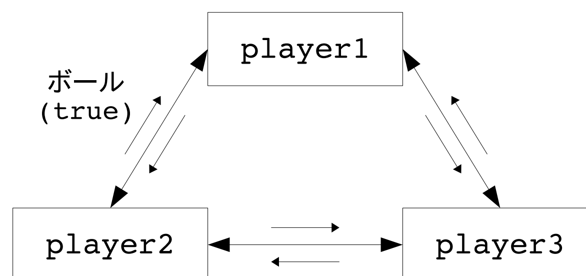

図 1　ボール投げの様子

2 つ目の例は, ストリームをチャンネルを用いて実現したものです．ストリームとはデータの流れを抽象化したものです．例えばファイル入出力はストリームと見ることができます．この例ではチャンネルを用いてそれを表現しています．intgen 関数は, start から 1 ずつ増加させた数値をチャンネルへ送信します．つまり初項 start で公差 1 の等差数列を表現します．filter 関数は, in からの入力を述語 (真偽値を返す関数)f でフィルタし, out へ送信します．map 関数は in からの入力に f を適用した値を out へ送信します．これらの関数間でデータの通り道となるのがチャンネルです．main 関数では c1,c2,c3 の 3 つのチャンネルを通信路とし, 図 2 のようなデータの流れをつくっています．

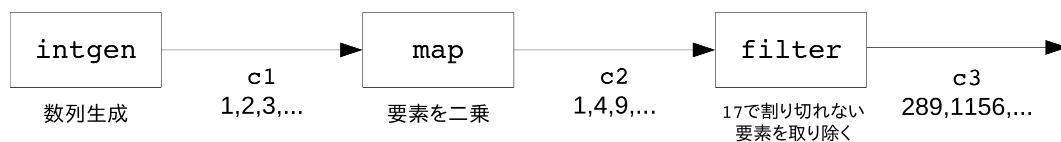

図 2　データの流れ

intgen 関数は, 無限ループとなっていることから分かるように無限の数列を表現します．メモリは有限なので無限に続くデータの列を生成することは不可能ですが, 無限とは言っても実際に必要なのはその一部分です．データの列を一度に生成するのではなく, 必要とされた時点で必要な分だけ生成するようにすることで, 見かけ上無限の列を表現することができます．そのようなストリームを遅延ストリームと呼びます．ここではチャ

ンネルの持つ「送信時にバッファがいっぱいだったらブロックする」性質を利用して遅延ストリームを実現しています. それぞれの関数はデータををひたすらチャンネルへ送信しようとしますが, バッファがいっぱいだとそこで一旦停止します. チャンネルへ値が要求され (?で受信され) バッファに空きができると 空いたバッファをまた埋めようとします. map や filter では入力が間に合っていない場合, チャンネルの性質により入力が来るまで待ちます. この例では, main 関数の終わりで末端のチャンネル c3 から 3 回受信しています. 受信する度に徐々にデータが計算されていきます.

intgen,filter,map 関数の呼び出しにはそれぞれ async が付いていて, 別スレッドで実行されます. それぞれ並行して実行されないと, 一度ブロックしたらブロックしたままで進まなくなってしまうからです[*3]. データの列が揃ってから次の工程へ進むのではなくデータを連続的に処理し, またそれぞれの処理は並列に実行されるので, これはパイプとよく似ています.

ユーザ定義演算子

```
fun main(){
    print_int(2^3^2);print("\n")
    var f= fun(x:int)=>int{ return x*x }
    var g= fun(x:int)=>int{ return x+2 }
    print_int( (f$g)(3) )
}
//累乗
fun ^ binary,right,50(x:int,y:int)=>int{
    return pow(x,y)
}
//関数合成(int->int に限る)
fun $ binary,right,60(f:fun(int)=>int,g:fun(int)=>int)=>fun(int)=>int{
    return fun(x:int){ return f(g(x)) }
}

---出力---
512
25
```

演算子を自分で定義することができます. 関数定義と同じような構文ですが, 引数リストの前に演算子の情報を記述します. 単項演算子であるか二項演算子であるか, 結合規則, 優先順位, の順です.

[*3] 今回チャンネルを用いて実現したからであって, 「遅延ストリームの実現には並行性が必要だ」という意味ではありません.

組み込み関数

```
fun main(){
    glut_openwindow("Lissajous")
    glut_setdisplayfunc(fun(){
        glut_clear(); glut_begin_point()
        var a=4.0,b=3.0
        for(0.0,1000.0,1.0,fun(t:double){
            glut_color3i(255,0,0)
            glut_vertex2i(d2i(sin(a*t)*50.0)+100,
                    d2i(cos(b*t)*50.0)+100)
        })
        glut_end(); glut_flush()
    })
    glut_mainloop()
}

fun for(start:double,end:double,step:double,block:fun(double)=>void){
    if(start>end){return}
    block(start)
    for(start+step,end,step,block)
}
```

現在, 以下の組み込み関数が利用できます.

- 数学 (sin, cos, tan, abs, rand 等)
- 型変換 (int と double の相互変換)
- コンソール出力 (print, print_int, print_double, print_bool)
- 並列 (スレッドのスリープ, プロセッサのコア数取得)
- グラフィックス (GLUT を利用, 簡単な 2D グラフィックスとマウス・キーボード・タイマーイベント)

上記の例は, リサージュ曲線を描画するものです. glut_setdisplayfunc に無名関数を渡して描画関数を登録しています. 組み込み関数とは関係ない話ですが, 多くの言語で構文が用意されている for ループは残念ながら soramame にはありません[*4]. しかし再帰とクロージャを使って関数で for ループを実現できます. 図 3 が実行結果です.

また, 図 4 は soramame で書いたテトリスの実行結果です. キーボードで操作します. 図 5 はライフゲームです. マウスで, セルの状態を変更したりシミュレーション速度を変更したりできます.

[*4] while ループはあります.

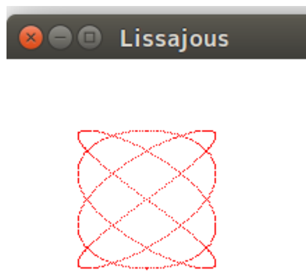

図 3　リサージュ曲線

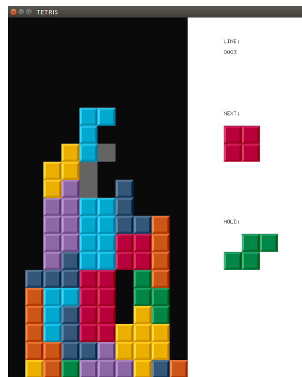

図 4　テトリス

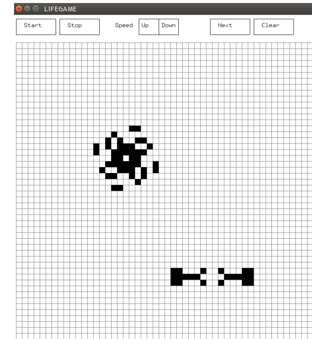

図 5　ライフゲーム

3　処理系

このセクションでは, 処理系の実装についての説明を行います. 今回作った処理系は, バイトコードを生成し, 仮想マシン上で実行する方式です. ソースコードを字句解析, 構文解析しながら構文木を生成し, 型検査を行った後にバイトコードを生成して仮想マシン上で実行します[*5].

再帰下降構文解析

再帰下降構文解析という手法を使えば比較的簡単に構文解析器を作れます. この手法では非終端子ごとに入力のトークン列を消費する関数を定義し, お互いに呼び出しあいます. 当初は再帰下降構文解析を使っていました. しかし, 文法規則と構文解析プログラムを分離できないため文法規則が複雑になるにつれてプログラムの見通しが悪くなってきました. そこで LR 構文解析と呼ばれる手法を使うことにしました.

LR 構文解析

LR 構文解析は, 文法規則から構文解析表を生成してそれを元に構文解析するという手法です. 再帰下降構文解析よりもプログラムはずっと複雑になりますが, 文法を変更する際に文法規則を書き換えるだけで済むため大変便利です. LR 構文解析器を生成するツールとして有名なものに yacc というソフトウェアがあります. それを使おうとしたのですが, ビルドがなかなかうまくいかずイライラし, 自作することにしました. ここからは LR 構文解析の簡単な説明を行います.

$$< whilestatement >::= "while"" (" < expression > ")"" \{" < block > "\}"$$

これは, while 文の BNF による文法規則です. `while` の後に左丸括弧, 式が来て, 右丸括弧, 左波括弧, 文の並びが来て, 右波括弧, であれば, $< whilestatement >$ とみなす, という意味です. 今回作成した構文解析器では, 文法規則を配列の形でソースコード内に直接記述するようにしました. 以下がその一部です.

[*5] 生成したコードをファイルに保存できないので, 実行毎にこれらを行います.

```
SyntaxRule SYNTAXRULE[SYNTAXRULECOUNT]={
（中略）
{{whilestatement,WHILE,LPAREN,expression,RPAREN,
         LBRACE,block,RBRACE,SYNTAXEND},while_reduce},
（中略）
};
```

構文解析表の作成方法については省略しますが, 構文解析表は LR 構文解析の要となります. 構文解析表は,「状態番号△の時にトークン〇〇が来たら, ☆☆しろ」という情報を持っています. 「トークン〇〇が来たら, 状態□へ移れ」は shift と呼ばれます. 様子見するといった感じです. 「トークン〇〇が来たら, ★番目の文法規則を適用して状態◇へ移れ」は, reduce と呼ばれます. 文法規則の右側の並びが揃ったので, まとめるということです. while 文の例でいえば, shift を続けていき, 途中で $< expression >$ や $< block >$ に reduce されながら, 最終的に $< whilestatement >$ に reduce されます. reduce される時に, その文法規則に対応した関数が呼ばれます. 上記のソースコードでは一番後ろの `while_reduce` が, $< whilestatement >$ へ reduce されるときに呼ばれる関数です. この関数には文法規則中の $< expression >$ や $< block >$ が渡されるためそれを元に構文木を構築していきます.

式の構文解析

式の構文解析で問題となるのは, 優先順位と結合規則です. これらの問題は, 文法規則を工夫することにより解決します. しかし, 今回作る言語は新たな演算子を定義できるようにしようと思っていました. 構文解析しながら動的に文法規則をいじって... 気が狂いそうです. 今回は, 式の構文解析を後から, 別の方法で行うという方法をとりました. はじめの構文解析で式を, 演算子と項の並びとして保持しておきます.

$$< expression >::= \{< primary > | < operator > | < parenexpr >\}+$$

$< primary >$ は数値や文字列, 識別子などです. この段階で優先順位や結合法則はまったく考慮されず, リストとして構文木に入れられます. また, 演算子定義を見つけるたびに演算子テーブルへ登録しておきます. そして型検査時に構文木をたどるタイミングで, 優先順位や結合法則を考慮した構文木に展開します. 操車場アルゴリズムを利用して中置記法から逆ポーランド記法へ変換した後, 構文木を作っています. このタイミングで定数畳み込みを行います.

型検査

静的型付け言語なので, コンパイル時に型の整合性をチェックします. 構文木をたどり, 木の深いところから型を決定していきます. そして記述されている型と一致するかを確認します. グローバル変数や関数定義には型を明記する必要がありますが, ローカル変数は型推論を行います. つまり, 得られた型の情報をそのまま使います. また C 言語等の言語では暗黙の型変換 (例えば, 整数と実数の間での変換) が行われることがあります

が, soramame ではすべて明示的に型変換を行う必要があります[*6]. これは暗黙の型変換による, 発見が難しいバグを生まないようにするためです.

コード生成

生成されるコードは仮想マシン (後述) のバイトコードです. この仮想マシンはスタックマシンであるため, 例えば $3 + 4 * 5$ だと,

```
ipush3 //整数 3 をプッシュ
ipush4 //整数 4 をプッシュ
ipush5 //整数 5 をプッシュ
imul //スタック上の 2 つの整数を掛けたものをプッシュ
iadd //スタック上の 2 つの整数を足したものをプッシュ
```

のようなコードが出力されます[*7].

バイトコードにおいて, 変数の参照は「さかのぼるスタックフレームの数」と「スタックフレームの変数領域での変数の位置」の組で表されます. スタックフレームをさかのぼるというのは, ここでは, 呼び出し元へとさかのぼるのではなく, 「自身を生成した関数のスタックフレームへのポインタ」をさかのぼる, ということを意味します. つまり, 構文構造上いくつ外側のブロックに変数が存在するか, ということです.

```
fun main(){
    var x=12, y=34
    var f=fun(){
        print_int(x)
    }
    print_int(x)
    print_int(y)
}
```

main 関数の内側で定義された関数 f 内の print_int(x) の x は, ひとつ外側の main 関数で定義された x なので, (1, 0) と表されます. また, main 関数内の print_int(x) の x は, (0, 0) と表されます. print_int(y) の y は, main 関数の 2 つ目のローカル変数であるため, (0, 1) と表されます. レキシカルスコープを採用しているため, これらの値をコンパイル時に決めることができます. 変数の参照をこのように表すことで, 実行時に変数探索をしなくて済むのです.

[*6] 組み込み関数 i2d(int **から** double), d2i(double **から** int) を使用できます.
[*7] 実際は定数畳み込みにより直接計算結果をプッシュするコードとなる場合もあります.

仮想マシン

処理系には主にコンパイラ方式とインタプリタ方式があります．今回は仮想マシンのバイトコードにコンパイルして, 仮想マシン上で実行させることにしました．有名な仮想マシンに JVM や CLR 等がありますが, そんな立派なものを使いこなせるようになるよりも自分でしょぼい仮想マシンを作ったほうが楽しそうなので作ることにしました．とはいえどんな命令が必要となるのか, JVM の命令セットを参考にしました．コンスタントプールの仕組みも JVM のまねです．今回作った仮想マシンは, スタックマシンです．オペランドスタックを使いながら演算を進めていきます．この仮想マシンのスタックフレームは以下の要素で構成されます.

- 変数領域
- プログラム・カウンタ
- オペランドスタック
- 関数へのポインタ
- 呼び出し元のスタックフレームへのポインタ
- 自身を生成した関数のスタックフレームへのポインタ

文字列や浮動小数点数, 関数等はコンパイル時に定数領域 (コンスタントプール) に保管されます.

3.0.1 継続

スタックフレームをたどり, オペランドスタックをコピーするという単純な方法を取りました．継続を取り出すたびに大きなコストのかかる, おそらく最も効率が悪い実装です[*8]. `makecontinuation` という命令で継続オブジェクトが生成され, `resumecontinuation` で呼び出します.

3.0.2 末尾呼び出し最適化

```
int main(){
    f();
    return 0;
}

void f(){
    printf("f");
    g();
}

void g(){
    printf("g");
```

[*8] Lime34 の湯浅先輩の Scheme では継続を取り出すのにコストのかからないような実装となっています.

```
    f();
}
```

これは C で書いたプログラムです．これはすぐにスタックオーバーフローしてしまいます．ところが, これに相当するプログラムを soramame で実行するとスタックオーバーフローすることなく `fgfgfgfgfg...` と無限に `fg` を出力します．ところで, これは次のように書き換えることができます．

```
int main(){
    goto f;
f:
    printf("f");
    goto g;
g:
    printf("g");
    goto f;
    return 0;
}
```

このように, 末尾呼び出しは `goto` で書きなおすことができます．つまりスタックを消費する必要がないということです．soramame は末尾呼び出しで余分なスタックを消費しないようになっているのです．

以下の `VM::Run` 関数は, 仮想マシンの核となる関数です．

```
//vm.cpp の一部
VMValue VM::Run(shared_ptr<Flame> CurrentFlame,bool currflame_only){
    (中略)
    try{
        while (true){
            (中略)
            bytecode = OPERAND_GET;
            switch (bytecode){
            case ipush:
                v.int_value = OPERAND_GET;
                STACK_PUSH(v);
                break;
            case pushim1:
                v.int_value = -1;
```

```
                STACK_PUSH(v);
                break;
            （以下ひたすら case が続く）
            }
        }
    }catch (exception& ex){
        （中略）
    }
}
```

バイトコードの種類だけひたすら case で分岐しています．この仮想マシンでは，オペランドスタック上に引数を右から順番に積み，関数へのポインタを積み，invoke 命令を呼ぶことで関数が呼び出されます．invoke 命令は 2 つの引数を取ります．末尾呼び出しかどうかのフラグと，別スレッドで実行するかどうかのフラグです．コンパイル時に末尾呼び出しだと分かると，即値で印を付けておき実行時にそれを見て不要なものを記憶しないようになっています．

3.0.3 並列実行

同じように，実行時に invoke 命令の引数をみて並列実行すべき (async 文を使った) だと分かれば，VM::Run を別スレッドで開始するだけです．

3.0.4 チャンネル通信

sendchannel/receivechannel 命令によって通信を行います．キューと条件変数を使い，待ちスレッドを管理します．ひとつのチャンネルへ複数のスレッドから同時にアクセスされる可能性があるため，ロックを使って排他制御を行います．

4 おわりに

今回，はじめてプログラミング言語をつくりました．最初に電卓のようなものを作ってから，変数，関数，クロージャ，継続，並行実行とチャンネル通信... といった具合に徐々に拡張していきました．soramame 言語には自慢できる特徴が無いので，今度はもう少し個性的な言語を作りたいです．また，この記事の中で GC について触れませんでした．GC は手を抜いてスマートポインタ (shared_ptr) を利用しているのですが，shared_ptr は参照カウント方式の GC であるため循環参照ができた場合解放されないという欠点があります．この処理系ではクロージャを作ると循環参照ができてしまうためこれは大きな問題となります．GC のこともちゃんと考えないといけないなと思いました．

soramame のソースコードは，https://github.com/matsud224/soramame で公開しています．また，第一回 カーネル／ VM 探検隊@名古屋で発表させていただいた際のスライドが http://www.slideshare.net/matsud224/soramame にあります．

参考文献

[1] 前橋和弥, プログラミング言語を作る (技術評論社)

[2] LLVM によるプログラミング言語の実装, http://peta.okechan.net/blog

[3] 文法と言語-自由文脈文法と LR 構文解析-, http://vrl.sys.wakayama-u.ac.jp/SS/

[4] Scheme 演習, http://www-ui.is.s.u-tokyo.ac.jp/~hara2001/scheme

[5] Java アセンブラ「Jasmin」でバイトコードの世界を覗いてみよう, http://www.hakkaku.net/series/java アセンブラ「jasmin」でバイトコードの世界を覗いて

[6] 数理科学的バグ撲滅方法論のすすめ, http://itpro.nikkeibp.co.jp/article/COLUMN/20060915/248230/?s2p