プログラミング言語処理系論 (5)
Design and Implementation of Programming Language Processors

佐藤周行
（情報基盤センター/電気系専攻融合情報学コース）

今回の予定

- 前回の残業です。
- BNFをそのまま解釈する
- BISON,YACCを動かしてみます
- 言語を定義する

今回のはじめ

- XMLの定義はこれで最後！
  - 規格からBNFを抜き出したものがこれ。xml-bnf.txt

Parser Generator

- 文法の定義がBNFで与えられている以上、BNFからそのままParserが生成されればとても便利

- 効率的にParserが生成できる文法のクラスが研究されてきた（LL(k), LR(k), LALR)
- 以降では、Parser Generatorツールである Yacc(Bison)の説明を行なう

実世界での有用性

- ほとんどのプログラミング言語では、LALR等で書かれ、parser generatorを使ってparserを出力しています。
- プログラミング言語の開発において、parser部分を自動化できたのは大きな貢献でした。
- 「偉大な」例外はごく最近のgccです。理由は調べてません。

DragonBookの例

```
%{
#include <ctype.h>
#include <stdio.h>
%}
%token DIGIT
%%
lines : lines expr '¥n'
            {printf("%d¥n", $2);}
      | lines '¥n'
      |
      ;
expr : expr '+' term          {$$ =
      $1+$3;}
      | term
      ;
term : term '*' factor        {$$ = $1
      * $3;}
      | factor
      ;
factor : '(' expr ')'  {$$ = $2;}
      |DIGIT
      ;
%%
```

```
yylex()
{
      int c;
      c = getchar();
      if (isdigit(c)) {
            yylval = c - '0';
            return DIGIT;
      }
      return c;
}
```

Bisonの入力

```
%{
#include <ctype.h>
#include <stdio.h>
%}
%token DIGIT
%%
lines : lines expr '¥n'      {printf("%d¥n", $2);}
      | lines '¥n'
      |
      ;
expr : expr '+' term         {$$ = $1+$3;}
      | term
      ;
term : term '*' factor       {$$ = $1 * $3;}
      | factor
      ;
factor : '(' expr ')'        {$$ = $2;}
      |DIGIT
      ;
%%
```

- トークンの定義
- BNFでルールを書く

  S：S1 S2… {action}
- S→S1 S2
- ルールに対してactionが定義されているときは、パースのときにそのactionを実行する
- $nは、n番目のシンボルのパースの結果出てくる値を表す（$$）

dc.c

```
#include <stdio.h>

main()
{
    return yyparse();
}

yyerror()
{
    fprintf(stderr, "FATAL ERROR¥n");
    exit(1);
}
```

% bison –v dc.y

% cc –O dc.c dc.tab.c –o dc

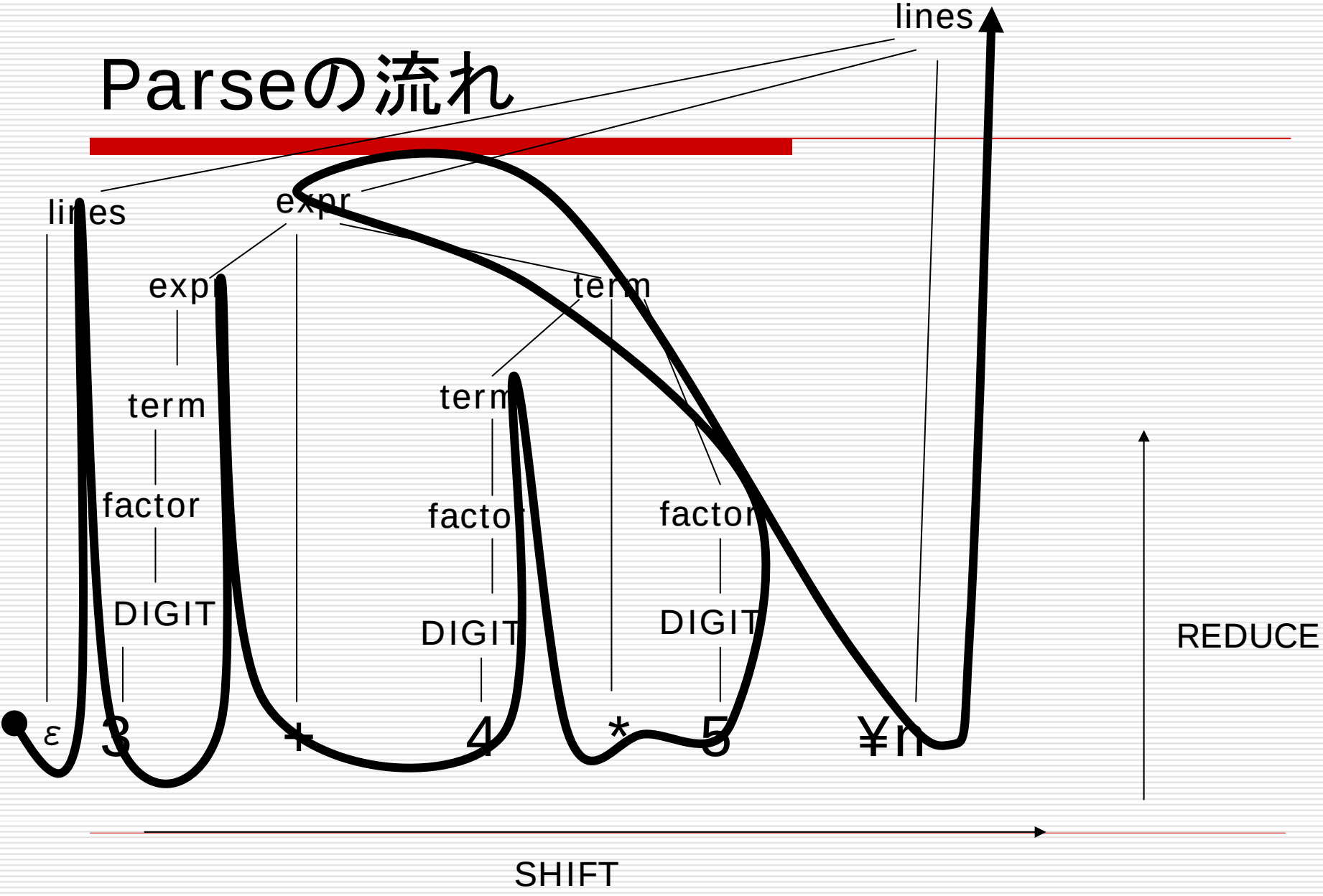
% ./dc

…

Bisonは、CYGWINをインストールすると、Windowsでも使えます

Linux等、Unix系ではbisonまたはyaccの名前で標準的に使えます

Parseの流れ


lines
lines
expr
expr
term
term
term
factor
factor
factor
DIGIT
DIGIT
DIGIT
ε
3
+
4
*
5
¥n
REDUCE
SHIFT

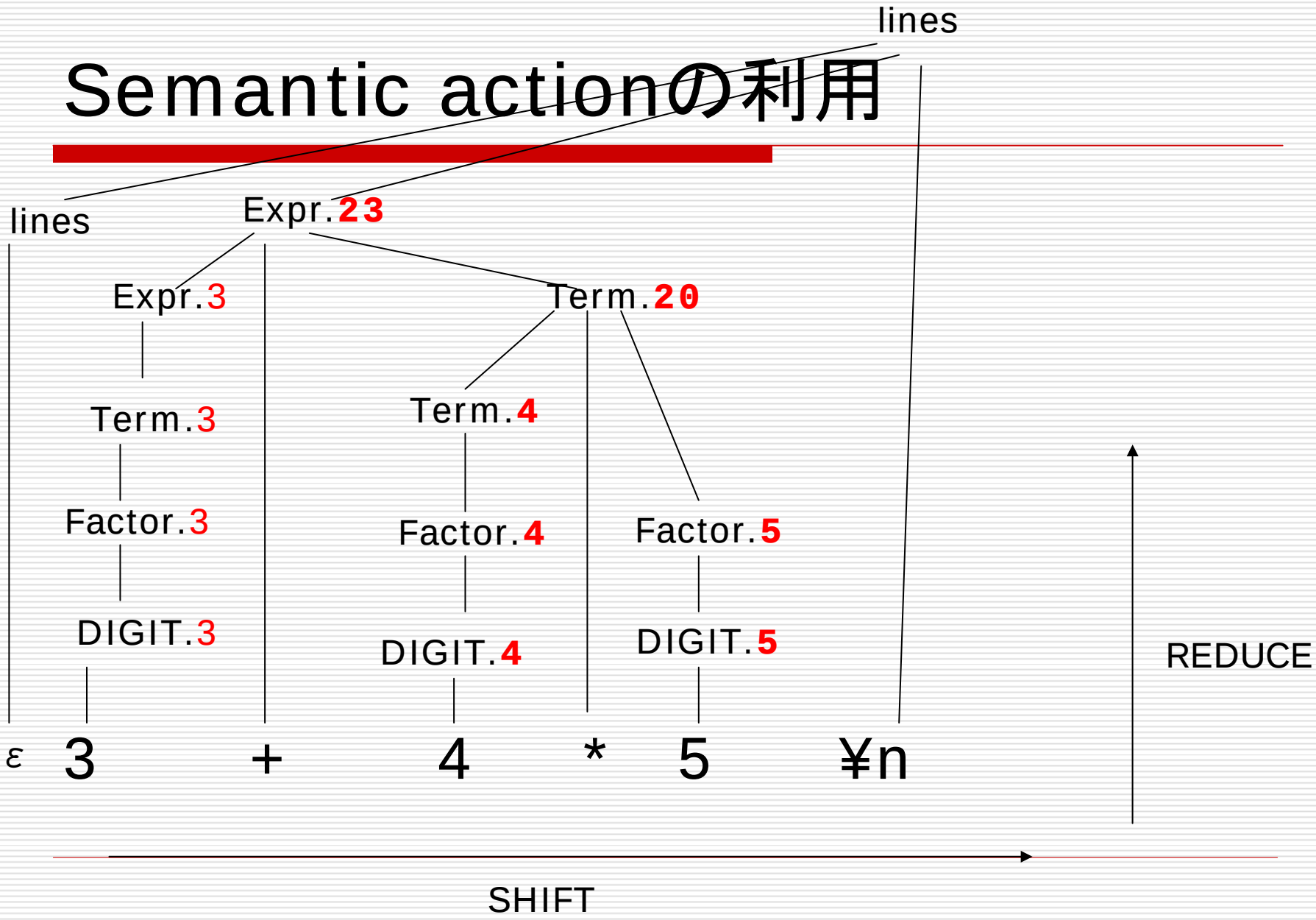
- Def Token: パースの単位
- Shift: パース時にトークンを読み進める
- Reduce: パース時にルールを(逆に)適用して、右辺から左辺に変換(還元)する
- どのタイミングでshift/reduceをするのかについてはここでは説明しないが、判断のアルゴリズムが存在するという意味でうまくいくようになっている文法を扱う

BISONの出力

- パース木は作ってくれるが、それをもとにどのような出力を構成するかはこちらの自由

- 直接解釈して値を出力
- 解析木をそのまま出力
- 計算のためのコードを出力
- …

Semantic actionの利用



プログラミング言語への進化

- コンパイラシステムの構築
  - 仮想マシンとマシン上の機械語の定義
  - 仮想マシン上での実行

- 変数の導入
- 制御構文の導入(compound, if, while)
- 関数の導入(function def/call)

優先度制御を利用したソースの合理化

```
expr : VARIABLE ASSIGN expr
     | '{' compound '}'
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr GE expr
     | expr GT expr
     | expr LE expr
     | expr LT expr
     | expr EQ expr
     | '(' expr ')'
     |DIGIT
     |VARIABLE
     | '-' expr %prec UMINUS
     |
     ;
```

優先度を制御する行

```
%left GE GT LE LT EQ
%left '+' '-'
%left '*' '/'
%right UMINUS
```

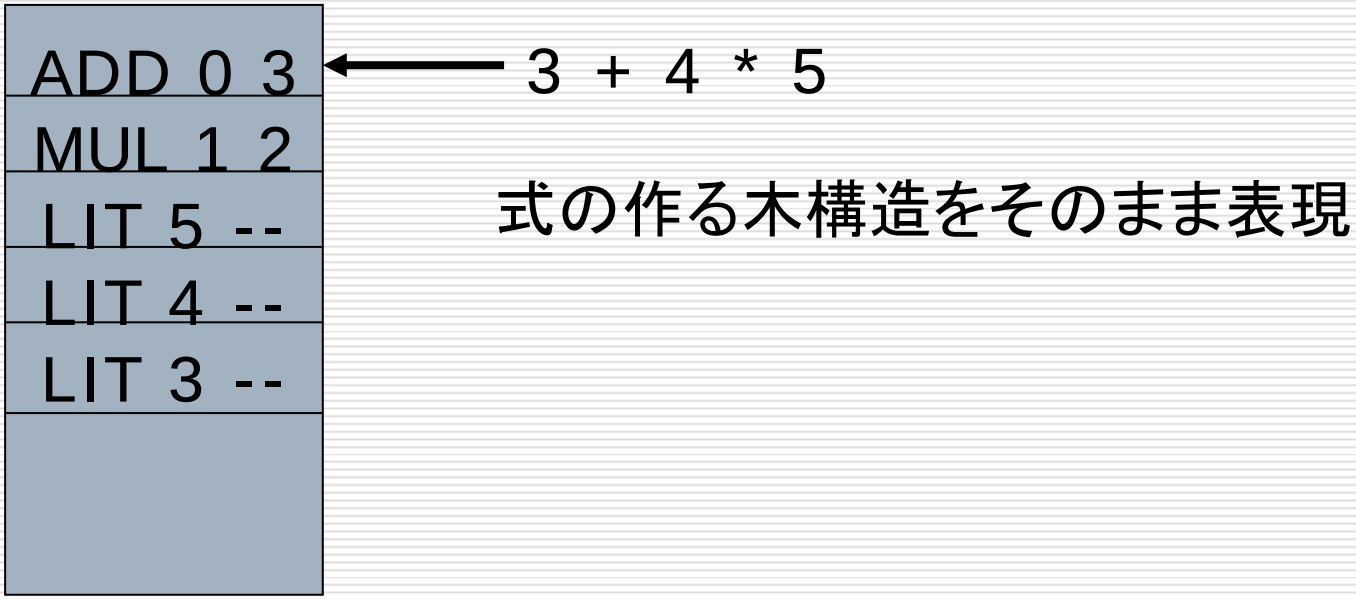
```
compound: compound ';'
      expr
      | expr
      ;

expr : VARIABLE ASSIGN
      expr
      | '(' expr ')' '?' expr ':'
      expr
      | '{' compound '}'
      | WH '(' expr ')' expr
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr GE expr
      | expr GT expr
      | expr LE expr
      | expr LT expr
      | expr EQ expr
      | '(' expr ')'
      |DIGIT
      |VARIABLE
      |VARIABLE '(' ')'
      | '-' expr %prec UMINUS
      |
      ;
defun: DE VARIABLE expr
```

- コードセグメントの出現
  - 関数コードの保存
- データセグメントの出現
  - では、一般的には、何を用意するとプログラムの実行に十分なのか？
  - 「実行環境」（次回）

Output

- 実際に何を出力するか観察してみる



- 制御構造も木構造で表現

```
wh (i > 0) {
    r := r * i;
    i := i – 1;
}
```

- この「木構造」(プログラム)を格納しておくところが「コードセグメント」

(WHILE, 2, 11)
(COMP, 6, 10)
(MOV, i, 9)
(SUB, 7, 8)
(LIT, 1)
(VAR, i)
(MOV, r, 5)
(MUL, 3, 4)
(VAR, i)
(VAR, r)
(GT, 0, 1)
(LIT, 0)
(VAR, i)

- データを格納する領域は？
- 名前空間
- スコープ

```
struct vardat {
  int kind;
  int val;
} vars[128];
```

□ 変数のバインディング

（課題4）変数の扱いについて、以下の点から考察し、コードを改良せよ。

(a) 関数には引数を渡せないのか？

(b) 引数が渡されると、変数名とデータの対応がどのように変更されるか。また、関数からリターンするとき、どのような処理が必要か？
（ヒントキーワード：環境、スコープ）

（課題5）四則演算と基本的な制御構造が実現できればプログラミング言語の骨格は完成し、関数が実現できれば文句なしである。

(a) 自分でdcc6以上のプログラミング言語を一つ定義し、その規格を書いてみよ。

(b) 規格からparserを生成せよ。Parserの出力は必ずしも実行コードである必要はない。

パス

- Def.　Pass
  - ソースプログラム(および中間生成物)を処理する一単位。
- Def.　One-Pass
  - パス一つ（パースを終了させるまで）でコード生成まで済ませる
- Def. Multi-Pass
  - 複数のパスから構成されるもの。一般的にはパースをした後で、解析（最適化等）をするパスを追加する

複数パスの表現

- ソースファイル中での表現

```
main(int ac, char **av)
{
  if (ac >= 2){
    if (!strcmp(*++av, "-d"))
      debug = 1;
  }

  code = yyparse();
  if (!code) exit(1);
  nextPass(Entry);
}
```

```
Node *Entry;

Top: ... {Entry = ...};
/* 最初か最後に、後々の
   処理のエントリーポイン
   トを設定しておく */
```

パスの考え方

- 近年の考え方では、パースまでと、パース以後の最適化をわけています。
- パースまでででできることは高が知れていると認識が一般的になりました。
  - 最適化ルーチン
- インタラクティブな言語では、パースまでの処理と、それ以後の処理（最適化等）のバランスが問題になっています。

今回触れなかったこと

- 処理の一単位となる「トークン」をうまく切り分ける必要があるので…
  - yylexはまじめにかきましょう
- パース時のエラー処理
  - 非常に重要なことはわかるのだが…

次の課題

- コンパイラがコードを生成したとき、それを実行するとはどういうことか？

- 実行環境の理解
  - Virtual machine
  - Hosted environment
  - Freestanding environment