プログラミング言語処理系論 (4)
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

- 言語規格を読むことの続き
  - BNFだけでできることは限られてくる
  - 「文法の定義＋制約」という記述の発明
- 文法から、パーサを作る
- BNFをそのまま解釈する
- BISON,YACCを動かしてみます

今回のはじめ

- さて、しばらくXMLの規格文書を読んでみましょう
  - 前回の続き
  - 具体的に、次ページのXML documentを規格の文法を用いて生成してみよう

```
<?xml version="1.1" ?>
<!– Address Book -->
<!DOCTYPE addressbook SYSTEM "addressbook.dtd">
<addressbook>
   <personal number="0001">
      <name>A</name>
      <address>Kanagawa</address>
      <tel>00-0000-0000</tel>
      <email addr="xxx@xxx.xx.xx"/>
   </personal>
   <personal number="0002">
      <name>B</name>
      <address>Tokyo</address>
      <tel>11-1111-1111</tel>
      <email/>
   </personal>
</addressbook>
```

BNFのポイント(前回の繰り返し)

- 文法＋制約条件が言語の定義のこつです
  - 「制約条件」を軽視してはいけません。ここにCFGだけでは記述しきれないさまざまなルールが自然言語で記述されています
  - 文法部分をシンプルに保ちながら、自然言語の記述力を活かす事で、プログラミング言語の定義の手法は完成しました（制約が守られているかどうかのチェックにはチェック用のプログラムを書くことが必要になります）

Parserの構成

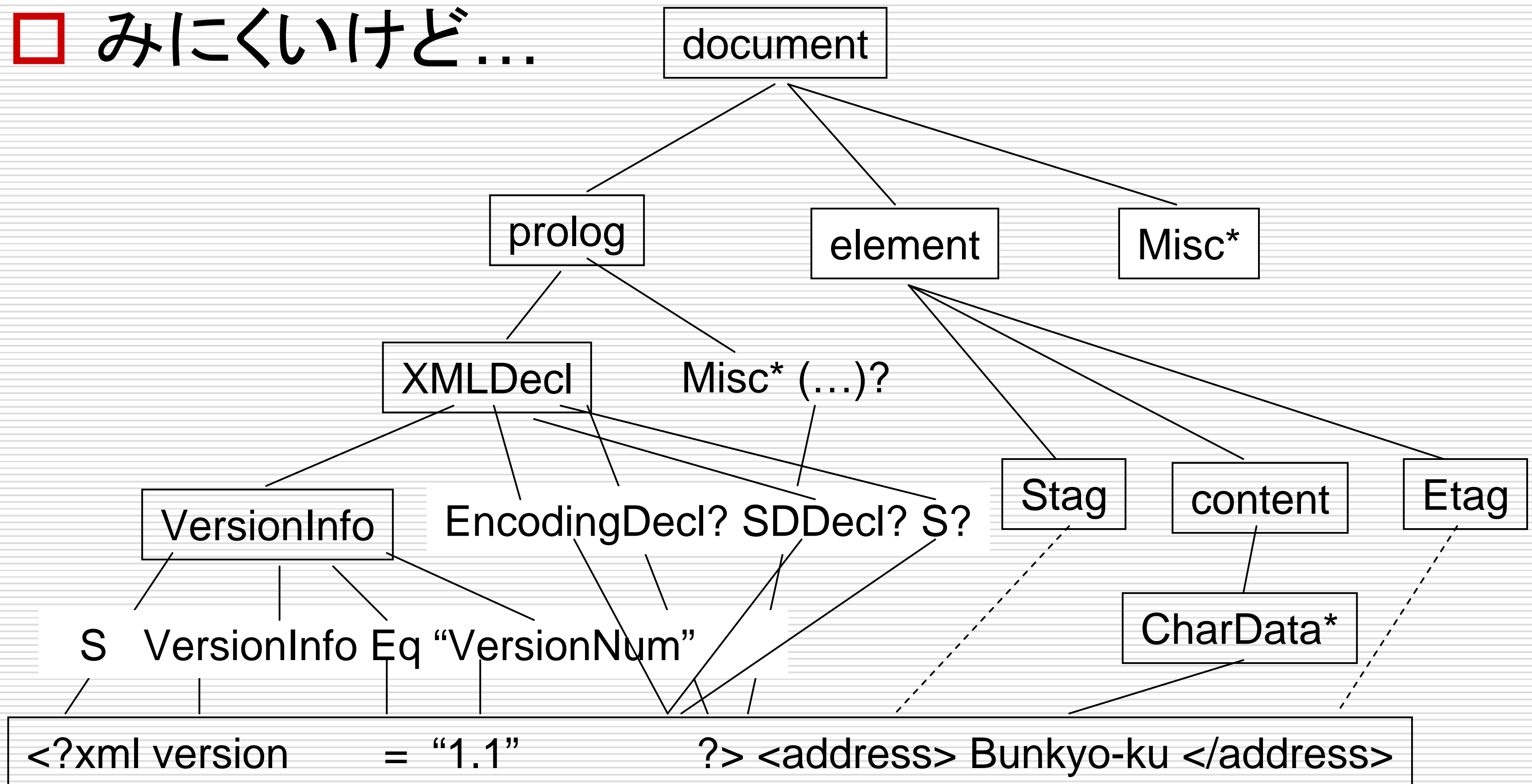
- 文法が与えられた。
- 文字列が一つ与えられた。
- その文字列が、文法に則って生成されたかどうかをチェックせよ
  ＝　文字列を生成する生成列をひとつ与えよ

- 文字列から生成列を作ることを「Parse」という

- 生成例
  document → (prolog element Misc*)
  → XMLDecl Misc* (doctypedecl Misc*)? element MIsc*
  → <?xml VersionInfo EncodingDecl? SDDecl? S? ?> Misc* (doctypedecl Misc*)? element Misc*
  → …
- 普通はTreeの形で書く。これをパース木という

パース木の例

- みにくいけど…



Parser

- Parserを作ろう
  ＝　文字列が与えられたとき、パース木を生成するプログラムを作ろう

- XMLの場合は簡単です。手でかける。
  - （課題1）XMLの文法がLL(1)であることを示し、XMLのパーサを書け。

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

Yacc & Bison

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($$)

なぜか？

- Yaccの例として出てくるものは、まず電卓。
  - 理由（推測）：標準的な教科書が導入例としてまず電卓を定義し、定着してしまった
  - （推測）式(expression)の定義は、それなりに大切だった。
  - 次のステップ（文の定義…）に進むには、勉強することが多すぎる
- 電卓は、yaccの例としてはあまりよくない。
  - Semantic actionの過大評価
  - 1パスパースの過大評価

dc.c

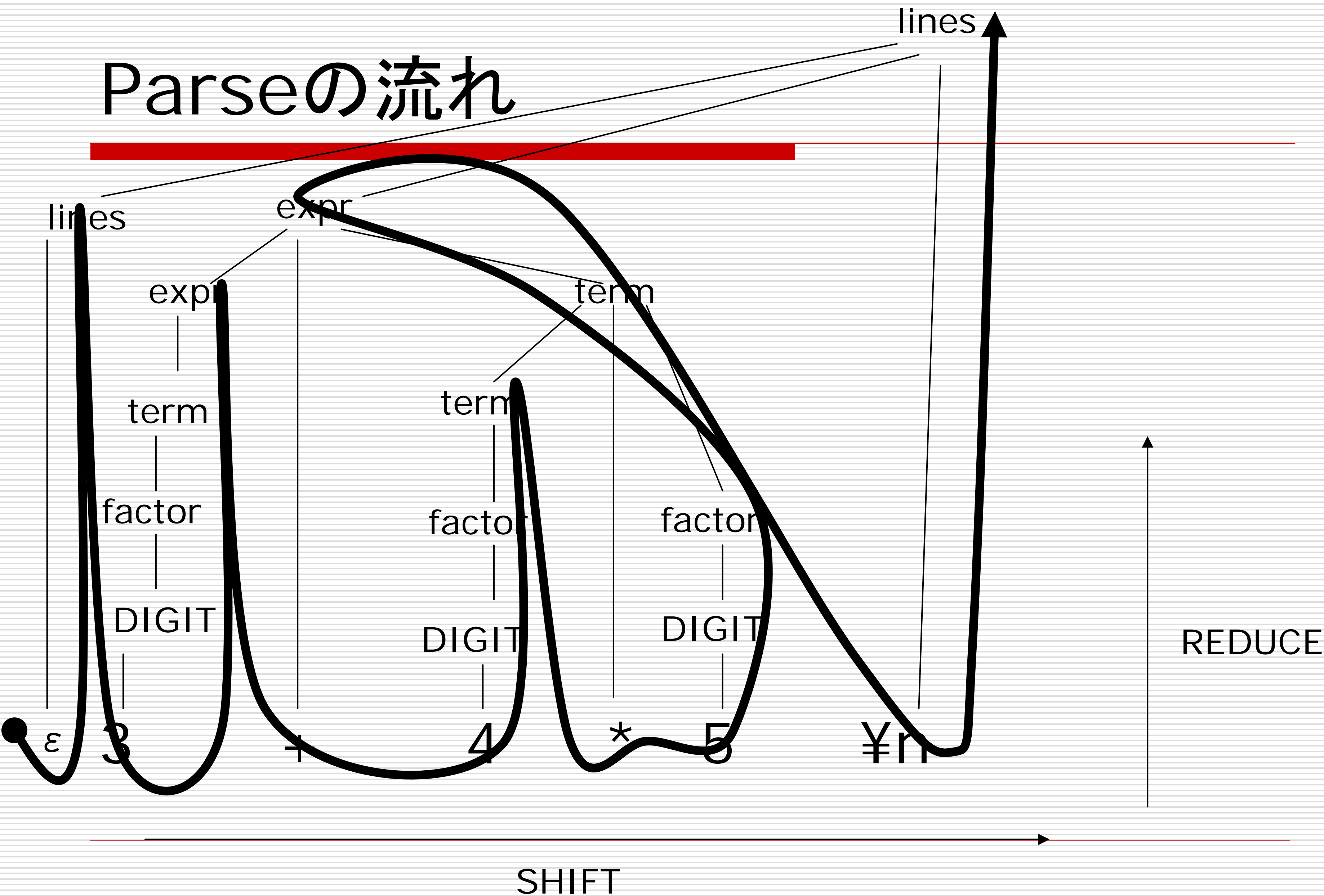
```
#include <stdio.h>

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
…
```

(課題2) dc.outputを解析し、どのような受理機械が生成されたか述べよ

(課題3) linesの定義の1行目が lines expr となっていて、 expr lines となっていない理由を受理機械の動作から説明せよ

Parseの流れ



- Def Token: パースの単位
- Shift: パース時にトークンを読み進める

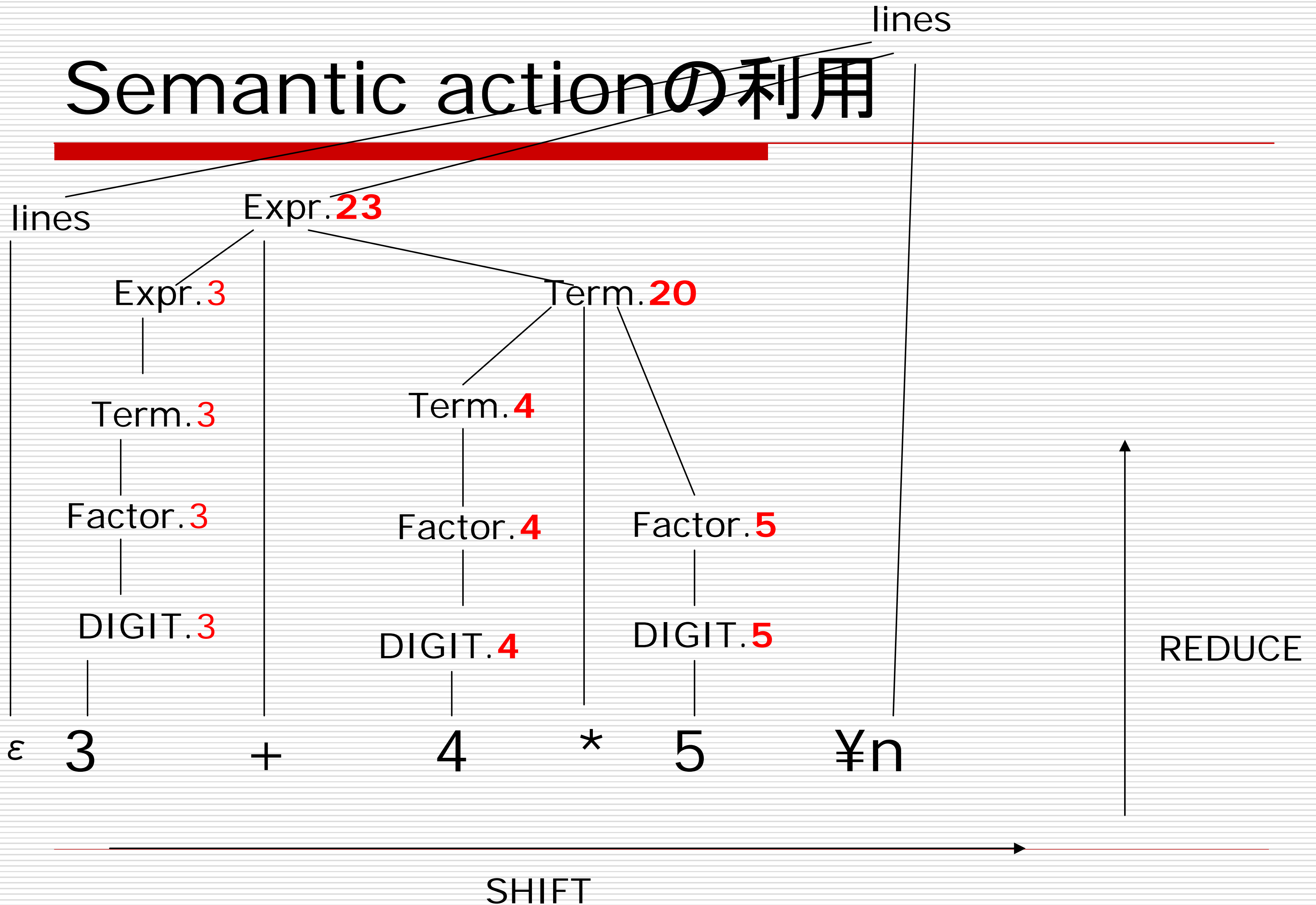
- Reduce: パース時にルールを(逆に)適用して、右辺から左辺に変換(還元)する

- 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）

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…

次回へのProlog

- 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