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

- プログラミング言語の定義をする
  - コンセプトを決める
  - 文法を決める（BNF）（必要ならば制約も）
- 処理系全体の構造を俯瞰する
- 文法からパーサを作成する
  - BNF → パースするプログラムの生成
  - 正常系だけを相手にするなら（エラー処理を考えないなら）言語の実装はとても簡単

プログラミング言語の定義

- 必要なものはなんであったか？
  - (High Level) コンセプト
    - プログラムの定義
    - 実行モデルの定義
    - データオブジェクトの定義
  - 文法要素へのブレークダウン
    - Syntax
    - Semantics

言語仕様の構成

- High Levelコンセプト
  - （もし）自分なりの新しい概念を導入したいときは、ここに書く。
  - 書く内容は Execution Model/Data Objectに加えてその「新しい概念」
- 文法要素とセマンティクス
- 他言語とのインターフェイス
- ライブラリ関数の仕様リスト

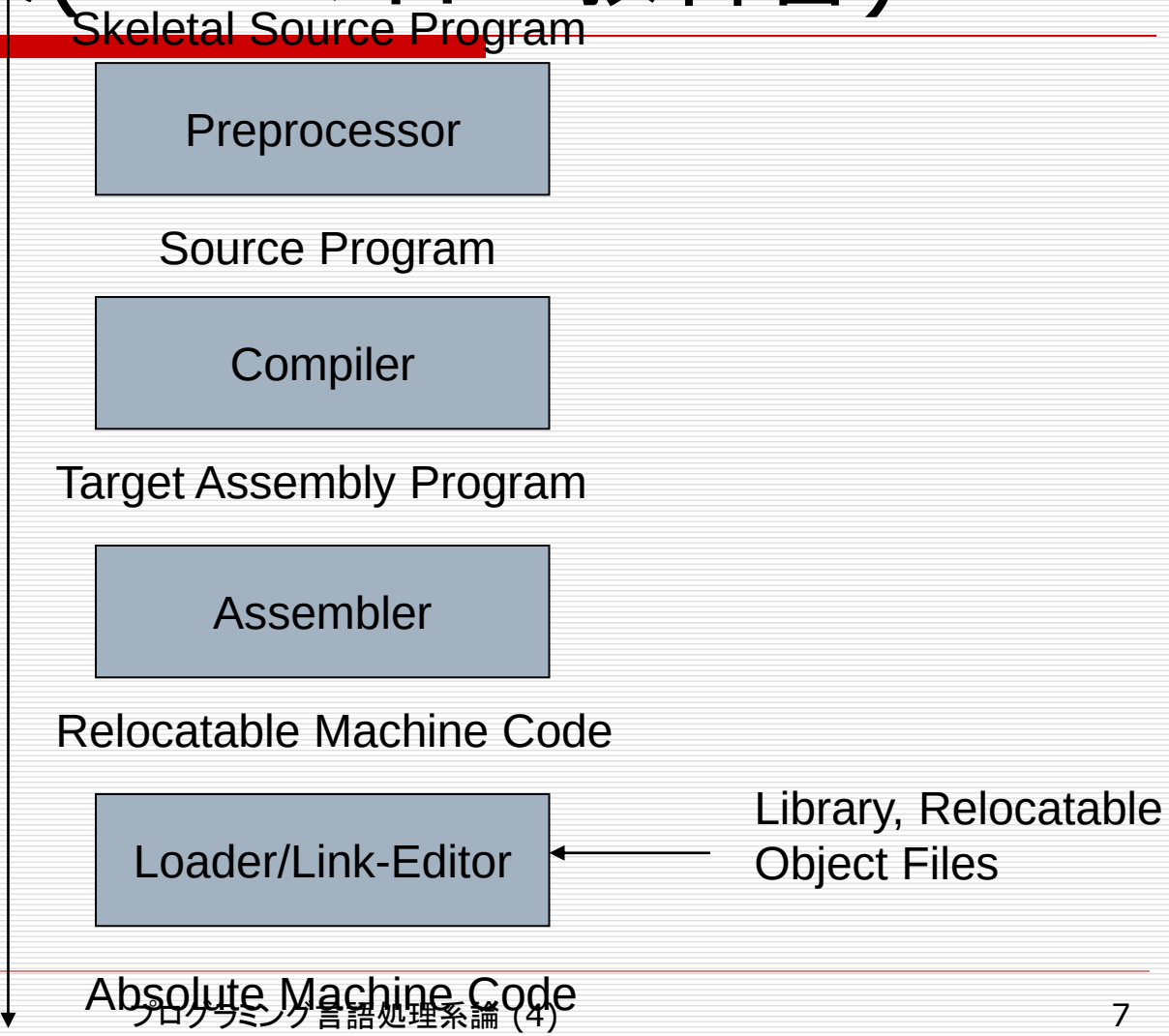
Fortranの仕様書を復習

- Rxxxとは何か
- Cxxxとは何か
- 地に書かれている文章は何か
  - 自然言語で書かれていて曖昧性はないのか？
    - ここでの曖昧性＝人によって解釈の揺れを許す

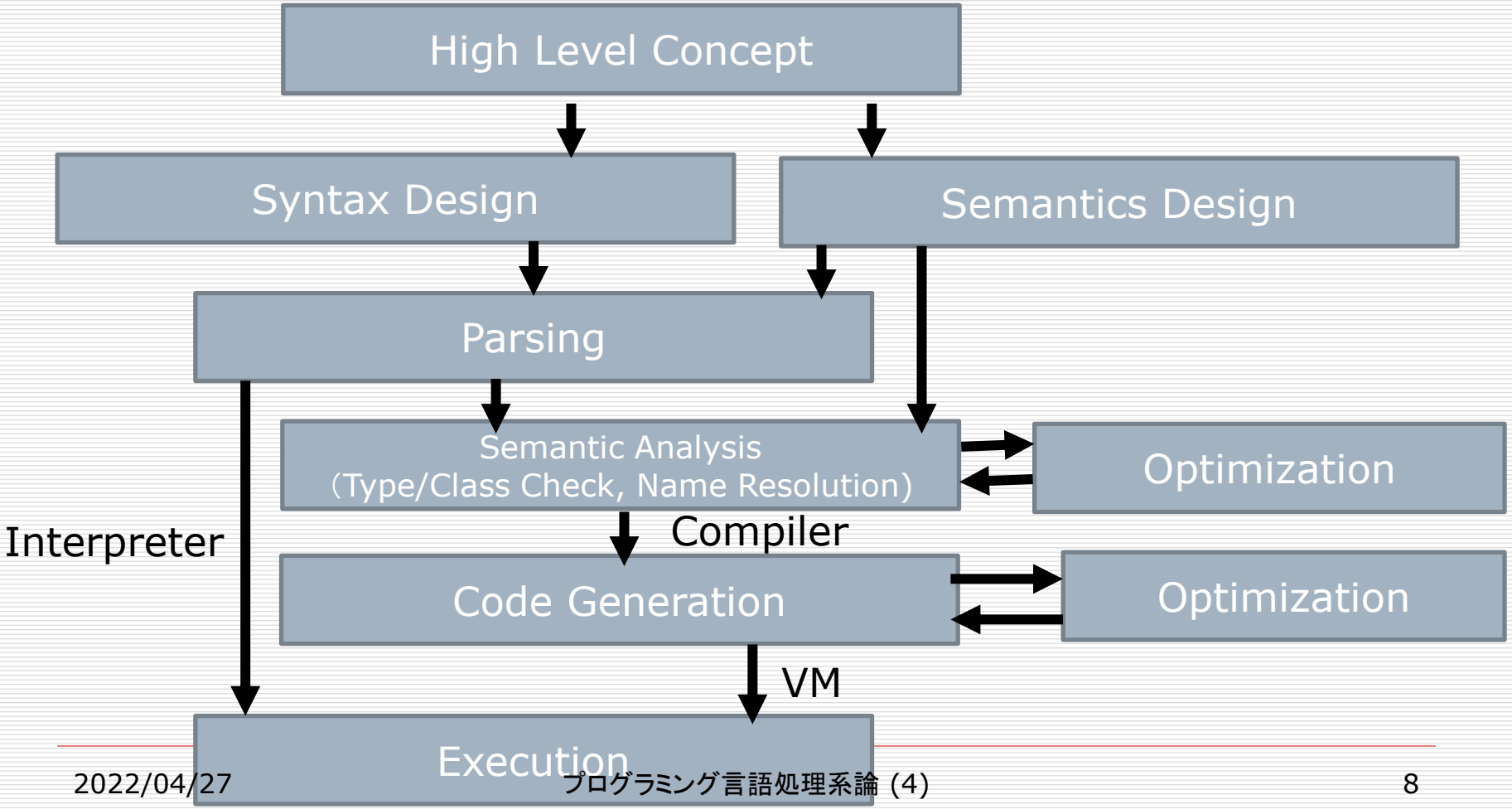
たとえば、こういう風に（ODC.pdf）

1. はじめに
2. 用語
3. ODCのコンセプト
4. データ型とクラス
5. データ実体
6. 式
7. 定義と参照
8. 関数
9. オブジェクト
10. リスト
11. スコープと引数のバインディング
12. インポート
13. 終わりに

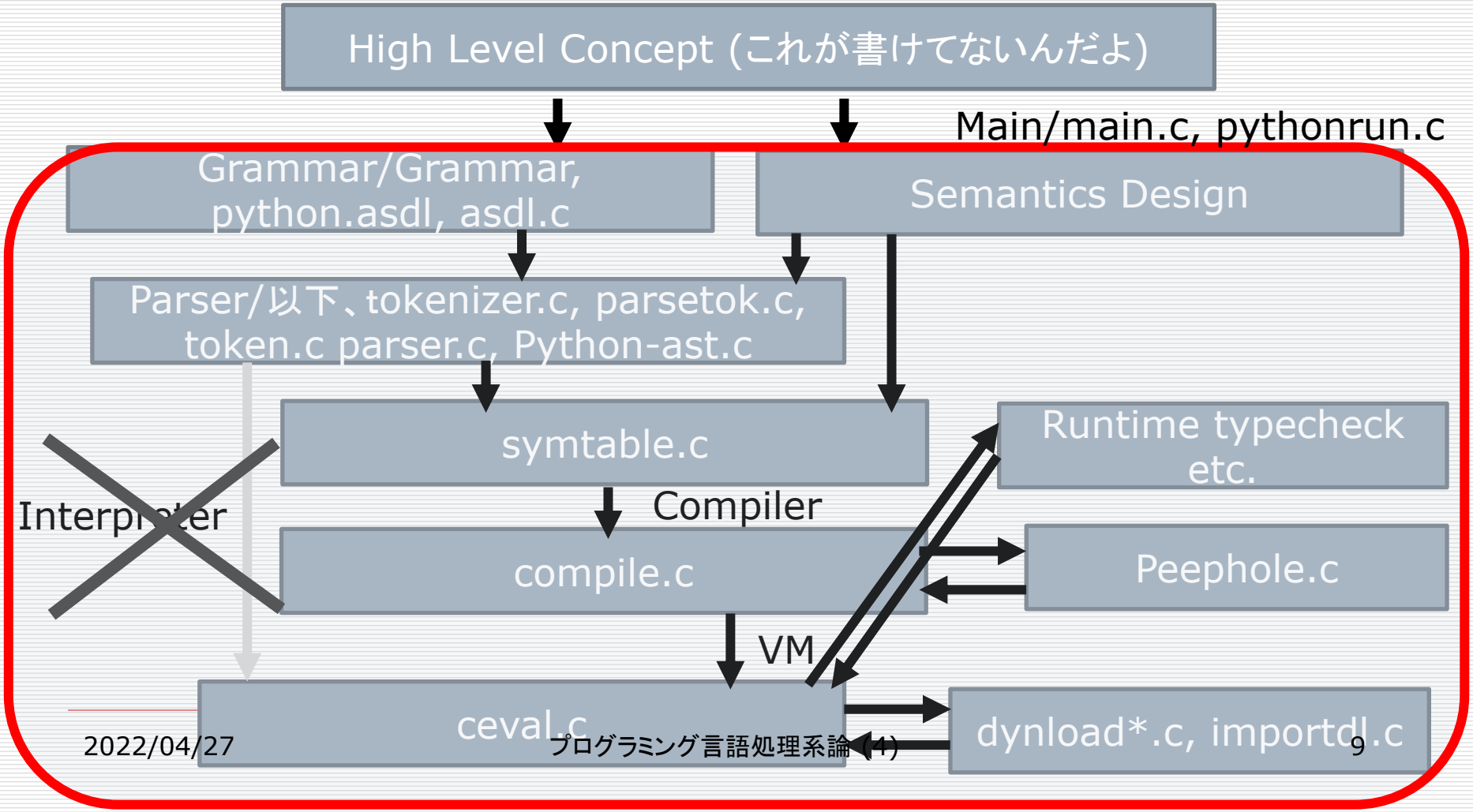
言語処理系(Ahoの古い教科書)



プログラミング言語処理系(classic)



CPythonでは何に対応するか



pythonrun.cの中心的なところを見る

- PyRun_InteractiveLoopFlags() の処理の流れ
  1. PyParser_ASTFromFileObject()
  2. PyImport_AddModuleObject()
  3. PyModule_GetDict()
  4. run_mod()
     1. PyAST_CompileObject()
     2. PyEval_evalCode()

具体的に

- High Level Conceptは、頭の中である程度固まったとして…
- どのように表現するか　＝　文法定義
- 文法要素の定義、特にSyntaxの定義
  - 標準的なプログラミング言語で行われているものを見てみる
  - BNFは記述の標準です

  Grammar/Grammar, Python.asdl

例

- 整数を対象にしたDesktop Calculatorを基本にして
- リストを扱い、
- オブジェクトも扱うのがベースライン

  以上、データ構造の設計
- if, while等の基本的な制御構造を入れ、
- 関数（subroutine）を扱うのがベースライン

  以上、制御構造の設計
- Cとのinterfaceが取れるようにする

  パフォーマンスはここでかせぐ

Pythonではどうか

- コンセプトは？
- データ構造の設計
- オブジェクトの設計
- 制御構造の設計
- モジュールとインポート

(Pythonには、正式な言語の定義が与えられていない
（利用者マニュアルのみ）)

（問題3）Pythonの適当なサブセットをさだめその言語定義を与え、仕様書にまとめよ。30ページ程度の簡略版で構わない。

（ヒント）仕様書の章組みは、他のオブジェクト指向言語で仕様が与えられているもの（Java, Ruby等）を参考にするとよい。Annotationやdecorationを無視した簡略版で構わない（ここらへんの捨て方は少し経験が必要…）

- プログラミング言語は定義された

↓

- プログラミング言語処理系を設計、実装する
  - プログラムのパース、コード生成← Next Step
- 実行のためのプラットフォームを設計、実装する
  - Source Code Traversal (for Interpreter)
  - Virtual Machine (for Compiler)

First Step

- 文法をBNFで定める
- トップダウンで決めることができる
- たいてい、仕様のトップに`program'(文法のルート要素)が出てくる

- ということで、見てみる

- 文法が定まったら、Semanticsを定める
  - 一意に定まる記述ができるか(わかりやすさよりも正確さが求められる)
- そしてParserの構築

具体的に言語を定義する

- 簡単な式を定義する

```
lines : lines expr '¥n'
      | lines '¥n'
      |
      ;
expr : expr '+' term
      | term
      ;
term : term '*' factor
      | factor
      ;
factor : '(' expr ')'
      |DIGIT
      ;
```

ここからどのような言語が生成されるかを観察してみよう

プログラミング言語の定義

- ほとんどすべてのプログラミング言語は、CFGを用いて定義されます
- 定義に用いる記法をBNFといいます。BNFはこの意味でメタな意味の言語です
- BNFは　CFG+α　で定義されます
- BNFからParserを自動生成できるか？
  - 少しの制限を加えるだけで、大部分のプログラミング言語のParserは自動生成できるということがわかった
  - 逆に、自動生成できるようなクラスに文法を制限することが行われた

定義の規模

- XMLの場合
  - ルール83 (欠番が少々あり)
- Fortran2018の場合
  - ルール　478
  - 制約　　606

BNFの（＋α）部分

Syntax ::= Alt1 | Alt2 (選択肢)
Syntax ::= [A] B ([ ] は省略可能 A?とも)
Syntax ::= A+, A* (一回以上の繰り返し、ゼロ回以上の繰り返し)

その他にも、アドホックな拡張がなされることがあります。

CPython, Perlではどうなっているか

- まずは、BNFを用いての文法記述
  - パース → Abstract Syntax Tree
- Python.asdl の観察
  - ASDL = Abstract Syntax Tree Description Language ← BNF + アルファ
    - ここでASDLの処理系はpythonで書かれていて…
- Perly.yの観察
  - yaccのソースプログラム
    - Yacc as a parser generator
    - 文法と、Parse後のAbstract Syntax Tree構築

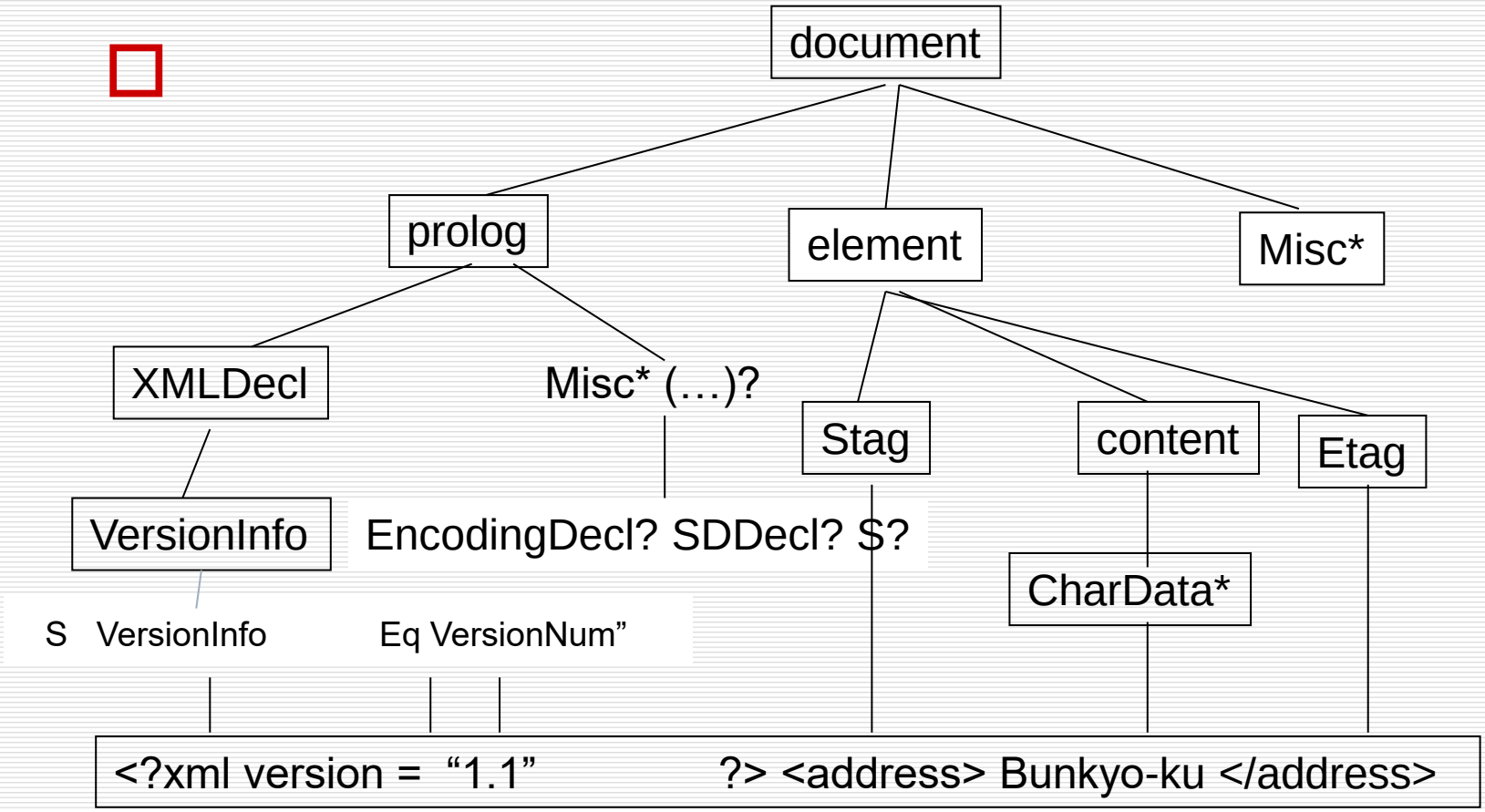
Parserの構成

- 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


document
prolog
element
Misc*
XMLDecl
Misc* (…)?
Stag
content
Etag
VersionInfo
EncodingDecl? SDDecl? S?
CharData*
S VersionInfo
Eq VersionNum”
<?xml version = “1.1”
?> <address> Bunkyo-ku </address>

Parse Treeの表現方法

- Treeくらいは自分で定義してOKだが…
- PythonはParse TreeからASDLを使って Abstract Syntax Treeを構築する
  - Abstract Syntax Description Language
  - Daniel C. Wang, Andrew W. Appel, Jeff L. Korn, and Christopher S. Serra : “The Zephyr Abstract Syntax Description Language,” Proceedings of the Conference on Domain-Specific Languages, 1997.

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

- Parserについては、成熟した理論があります
  - CFG (Context Free Grammar) のサブクラスとしてのLL(k), LR(k), LALR
- 亜種としていろいろなものが提案されています。例えばPEG (Parsing Expression Grammar)

大原則

- 文法をBNFで定義する（人手をかけるのはしょうがない）
- BNFから
  - 入力としての文字列を読み込んで
  - Parse Treeを生成する
  - プログラムを「自動で」生成する
- Parser Generator

Pythonのパース

- プログラム → ノードの木構造 (parser.c)
  - 文字列 → トークン (tokenizer.c)
  - トークン → ノード木 (parsetok.c)
  - ここにASDLで記述した文法から生成したパーサを使う
  - パーサーは一般にオートマトン
    - /* graminit.c (みてもわからないよ) */

より一般にbisonを使ったもの

- LALRを用いた文法の記述 (perly.y)
  - tokenの記述 (lex.c)
  - BNF + semantic action
    - BNFでパース木、semantic actionの部分でabstract syntax treeを生成

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

Parserで遊びたい人へ

- Parseは、トップダウンに行うもの（決め打ち）とボトムアップに行うものがあります
  - LL(k) → トップダウン (recursive descent)
    - 手で書ける。
    - Horn節などとの親和性を指摘する人もいる
  - LR(k), LALR → ボトムアップ
- いい加減枯れているので、ここにエネルギーを注ぐのは無駄です

PEG

- トップダウンにパースを行う
- !や&を用いて、陽にlookaheadを表現する

- B. Ford: “Parsing expression grammars: a recognition-based syntactic foundation,” POPL04, 111—122, 2004.

Parsing Expression Grammar

- BNFに加えて以下のルールを置く
  - !e (eが出現しない)
  - &e (eが必ず出現する)
  - r/s (ルールrがsに優先する)
- 例:
  - id ::= !reserved letter+
  - expr ::= expr [+] factor /factor

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(C compiler)です。Parser部分はべたなCプログラムとして提供されています

Yacc & Bison

Yacc & Bison

- Cプログラムを出力するものが有名ですが、
- Javaプログラムを出力するもの（Java Yacc, CUP, …）、Perlプログラムを出力するもの等、同じ原理で、異なる言語上で動くものがたくさんあります

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
  - Qiitaに偉そうに書くんじゃねぇよ、と声を潜めて主張したいと思います

dc.c

```
#include <stdio.h>

main()
{
    return yyparse();
}

yyerror(char *msg)
{
    fprintf(stderr, “%s¥n“, msg);
}
```

教材ではもう少しまともです

```
main(int ac, char **av)
{
  int r;
  FILE *fp;

  yyin = stdin;
  fname[0] = 0;
  while ((r = getopt(ac, av, "Odf:")) != -1) {
    switch (r) {
    case 'O': optimize = 1; break;
    case 'd': debug = 1; break;
    case 'f': strncpy(fname, optarg,
FILELEN);
    }
  }

  yyinit();

  if (fname[0]) {
    if ((fp = fopen(fname, "r")) == NULL) {
      fprintf(stderr, "file %s not found¥n",
fname);
    } else {
      yyin = fp;
      yyparse();
      yyin = stdin;
    }
  }
  r = setjmp(topenv);

  return yyparse();
}

yyerror(char *msg)
{
        fprintf(stderr, "%s¥n", msg);
}
```

% bison –v odci.y

% cc –O dc.c odci.tab.c –o odci

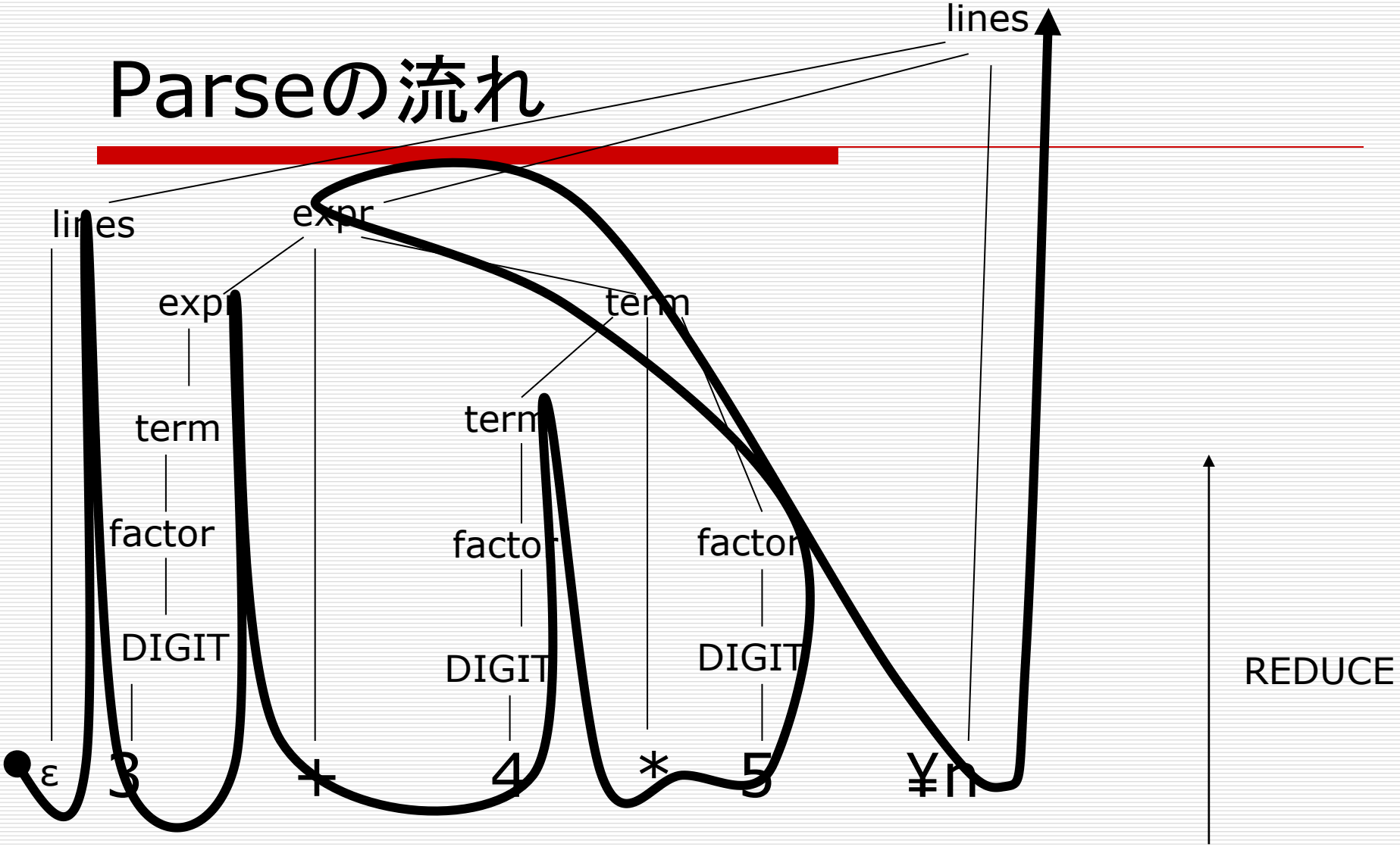
% ./dci

…

Bisonは、CYGWINをインストールすると、Windowsでも使えます

Linux等、Unix系、Mac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
  - Unicodeだとねぇ (Python)

- 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

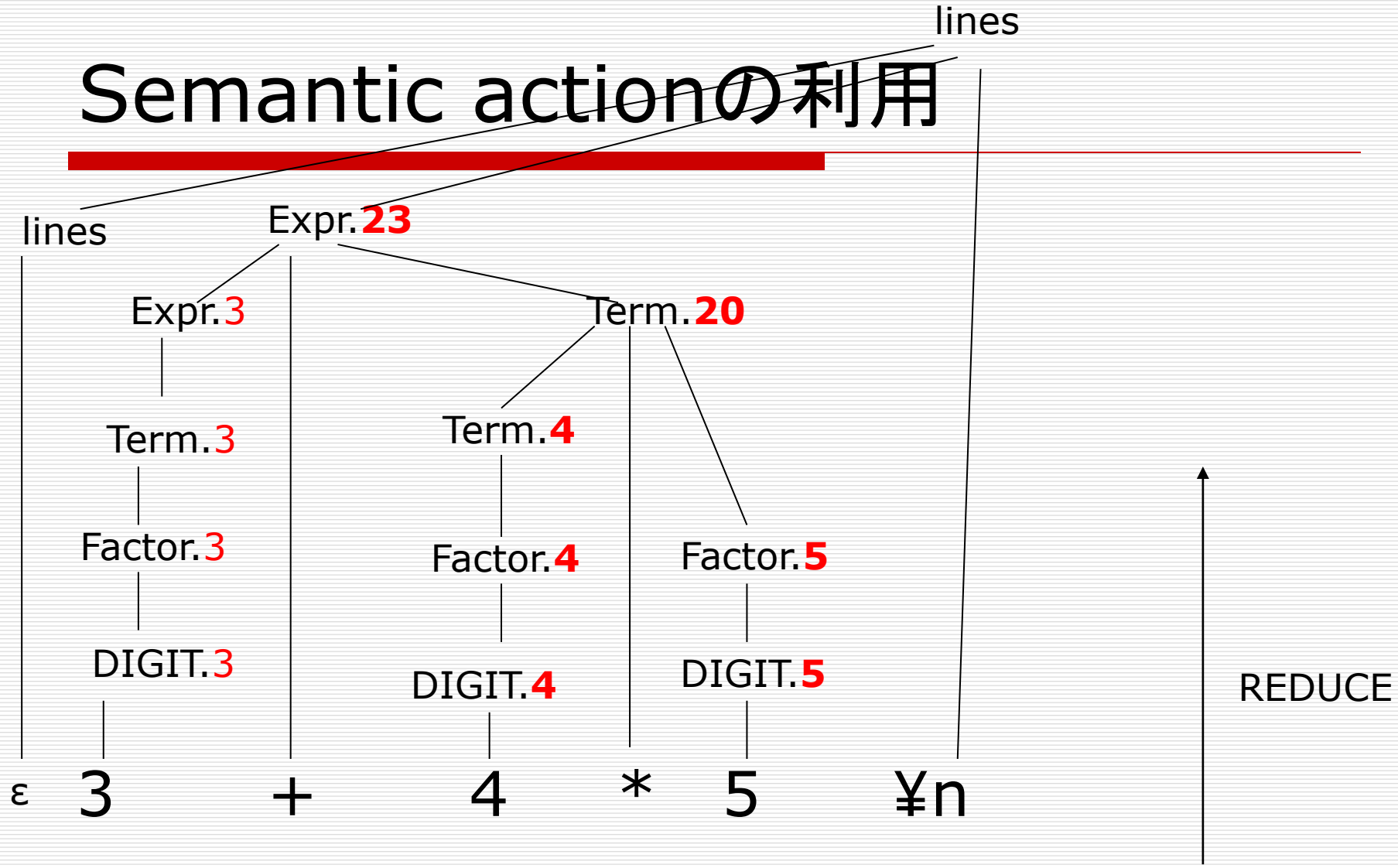
- どのタイミングでshift/reduceをするのかについてはここでは説明しないが、判断のアルゴリズムが存在するという意味でうまくいくようになっている文法を扱う

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（Semantic Action直接）
- 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(仕様の観点から)

- データ(オブジェクト)の概念の記述
  - オブジェクトが定義できるか？
  - とりあえずは「整数」だけにするか
- 実行(Execution)Controlの記述
  - Compound Statementsだけで十分か？
  - While等、繰り返しは必須か…
- Statement/Expression (プログラムの構成単位)
  - Expressionは十分だろう
  - Statementの種類は…
- 前のスライドと平仄があっていることに注意

プログラミング言語への進化 (cont'd)

- パース木をそのまま実行
- 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
- 制御構造の導入(複文, if, while, …)
- 環境の導入
- 関数の導入(function def/call)
  - フレームの設計
- オブジェクトの導入

優先度制御を利用したソースの合理化(ごく簡単なものを除いてやっちゃいけないが)

```
expr : VARIABLE ASSIGN expr          優先度を制御する行
     | '{' compound '}'
     | expr '+' expr
     | expr '-' expr                 %left GE GT LE LT EQ
     | expr '*' expr                 %left '+' '-'
     | expr GE expr                  %left '*' '/'
     | expr GT expr                  %right UMINUS
     | expr LE expr
     | expr LT expr
     | expr EQ expr
     | '(' expr ')'
     |DIGIT
     |VARIABLE
     | '-' expr %prec UMINUS
     |
     ;
```

```
expr : assignment
     | ifstatement
     | loopstatement
     | '{' compound '}'
     | aexpr
     | condexpr
     | RE expr
     | MY '(' varlist ')'
     | DIGIT
     | VARIABLE
     | VARIABLE '(' exprlist ')'
     | listref
     | objref

assignment: lhs ASSIGN expr
          | lhs ASSIGN newobject
          | lhs listdef

lhs: VARIABLE
   | objref
   | listref
```

エラー処理

- エラー処理として：
  - エラーが起きた所で処理を中断し、適当な場所まで巻き戻す
    - スクリプト言語（特に成熟していないもの）はここからはじまります
  - エラーが起きた所で処理を中断し、エラーを報告し、さらに処理を続ける
    - C,Java等きちんと作られている言語はほとんどこれです
    - プロダクションレベルのコンパイラと、趣味で作るコンパイラの差はエラー処理に端的に現れます
  - とりあえず、教材のようにyyerrokを使って…

- シンボルテーブルの管理
  - 大域変数の管理
  - 動的な言語では、実行時にも使用
  - 静的にすべての名前解決をする処理系ではリンク時まで
- 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
  - 変数にデータをバインドする
  - 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」（次回以降のテーマ）

CPythonでは…

- symtable.c でやっていること
  - ASTをトラバースして、
  - 変数の出現の観察
  - どの変数と変数の参照が同一のものかの同定
  - コード生成のときに本質的な情報として利用

- Pythonの不思議な変数利用規約
  - 未定義の変数への参照は、大域変数
  - 一旦代入されると局所変数へ
- 次回、まとめてやります

関数の導入

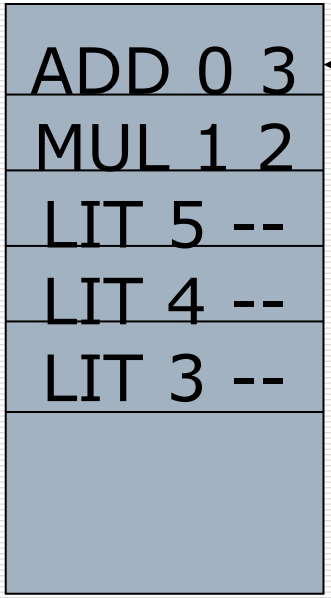
- 関数の出現につれて考えなければならない問題
  - コードセグメントの管理
  - フレームの管理
    - ローカル変数、グローバル変数
    - スコープの管理
    - 引数の渡し方
      - コンパイル言語なら、パラメタと実引数の対応をきちんととることが前提
- 次回、まとめてやります

オブジェクトの導入

- Classの導入
  - Class 階層の導入
    - Extends keyword
  - Access の許認可の問題
    - Private/Protected/Public
- Type/Class Check
  ← Semantic Analysis の最大の問題の一つ
  - Static/Dynamic
  - Subclass
  - Polymorphic Types
- 次回、まとめてやります

とにもかくにも

- Parserが実際に何を出力するか観察してみる



３ ＋ ４ ＊ ５

式の作る木構造をそのまま表現

- 制御構造も木構造で表現

```
while (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AS」(Abstract Syntax Tree)
- VMを設計する場合は、ここからさらにコンパイルする

```
(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)
```

Symbol table

- 「名前」を格納する領域は？
- 名前空間
- スコープ
  - ローカル変数、グローバル変数
- 何を格納する場所を用意するのが良いのか？（ヒープの設計）
- まずは変数管理のためにオブジェクトのテーブルを作る

```
typedef struct _vardat {
  int kind;
  int name;
  void *val;
  int param;
  int paramlen;
  int res_t;
} t_vardat;

t_vardat vars[];
```

では、本格的なプログラミング言語では…

- Perl5を見てみましょう。
  - Parse Tree → ASTをほぼそのまま保存
  - Tree Traversalでコードを実行
    - Interpreter方式で古典的な方式の一つ
- 今まで説明に使ってきた（電卓＋）は、Parse Treeをコードにしていた。

実は、Perlにおいて

- 実はPerl5において
  - 変数はグローバル
  - myを使ってローカルな変数を定義できる
  - Perl5の前近代的な部分
  - Objectが導入できなかった
  - 名前空間を制御するmoduleでclassとobjectが管理できると考えたのが…
  - Rakuはどうなるのでしょうか

Perl5で関数を実現？

- この方針にしたがって関数コールを実現してみる
  - フレームを作る
  - （スタック）フレームとは：関数呼び出しごとに作られるローカルな情報を格納する場所
- 関数の中でのスコープ（次回に説明）の扱いは？

もっとおそろしい言語があってな

- Fortranのごく初期においては
  - 関数呼び出しにおいて、関数コールごとの実行環境（フレーム）は関数ごとに固定
  - グローバルな変数は存在せず、EQUIVALENCE文で関数コールごとに対応を指定
- （課題*：考古学）Fortranの関数コールにおけるフレームの作り方について調査せよ。Fortranは、「再帰」を理解できないプログラマを大量に養成したといわれる（半分デマ）が、実際Fortranでは特に指定しない限り再帰が書けない。その理由をフレームの作り方と関連付けて述べよ

Perlの実際

- Perl –MO=Concise,関数名,-src ファイル名
  - B::Conciseモジュールを使ってみる
- perl –MO=Concise,factorial,-src fact.pl

fact.pl

```
sub factorial {

    $r = 1;

    while ($i>0) {
      $r = $r * $i;
      $i = $i-1;
    }
    return $r;
}


$i = 7;
print factorial();
```

```
$ perl -MO=Concise,factorial,-src fact.pl
main::factorial:
t  <1> leavesub[1 ref] K/REFC,1 ->(end)
-     <@> lineseq KP ->t
# 3:      $r = 1;
1         <;> nextstate(main 1 fact.pl:3) v:{ ->2
4         <2> sassign vKS/2 ->5
2           <$> const[IV 1] s ->3
-           <1> ex-rv2sv sKRM*/1 ->4
3              <#> gvsv[*r] s ->4
# 5:      while ($i>0) {
5         <;> nextstate(main 3 fact.pl:5) v:{ ->6
```

```
o        <2> leaveloop vKP/2 ->p
6          <{> enterloop(next->j last->o redo->7) v ->k
-          <1> null vK/1 ->o
n            <|> and(other->7) vK/1 ->o
m              <2> gt sK/2 ->n
-                <1> ex-rv2sv sK/1 ->l
k                  <#> gvsv[*i] s ->l
l                <$> const[IV 0] s ->m
-              <@> lineseq vKP ->-
# 6:    $r = $r * $i;
7                <;> nextstate(main 1 fact.pl:6) v:{ ->8
c                <2> sassign vKS/2 ->d
```

```
a                    <2> multiply[t6] sK/2 ->b
-                      <1> ex-rv2sv sK/1 ->9
8                        <#> gvsv[*r] s ->9
-                      <1> ex-rv2sv sK/1 ->a
9                        <#> gvsv[*i] s ->a
-                    <1> ex-rv2sv sKRM*/1 ->c
b                      <#> gvsv[*r] s ->c
# 7:    $i = $i-1;
d                  <;> nextstate(main 1 fact.pl:7) v:{ ->e
i                 <2> sassign vKS/2 ->j
g                    <2> subtract[t9] sK/2 ->h
-                      <1> ex-rv2sv sK/1 ->f
```

```
e                       <#> gvsv[*i] s ->f
f                     <$> const[IV 1] s ->g
-                   <1> ex-rv2sv sKRM*/1 ->i
h                     <#> gvsv[*i] s ->i
j                 <0> unstack v ->k
# 9:    return $r;
p       <;> nextstate(main 3 fact.pl:9) v:{ ->q
s       <@> return K ->t
q         <0> pushmark s ->r
-         <1> ex-rv2sv sK/1 ->s
r           <#> gvsv[*r] s ->s
```

PerlのParse Tree

- Perlは、コードセグメントはほぼAST
- 実行のための最小限のヒープ、スタックが用意されている
- B::Conciseで内容を見ることができる

- 似たことは、Javaのdisassemble
  - javap -c
- Pythonのdisassemble
  - import dis; dis.dis()でも
- disassembleは、仮想マシン上の命令列を出力します

- この違い（Source Tree Traversal vs. Compile）が次のセクションの大きなテーマ

- Perl5の実行について
  （課題4）PerlのB::Conciseモジュールを利用して、以下についてレポートせよ
  (1) 適当なプログラムに対してのソースとパース木の対応（-src）の観察
  (2) 実行に際して必要となるデータ構造（スタック、フレーム、ヒープ）
  - (2)について無理する必要はありません