

# コンパイラ及び演習

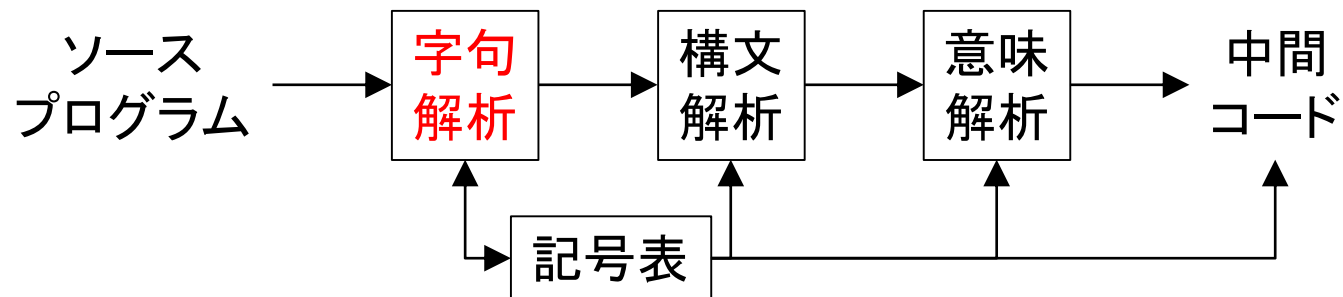
関澤 俊弦

日本大学 工学部 情報工学科

# 復習

## ■ 字句解析

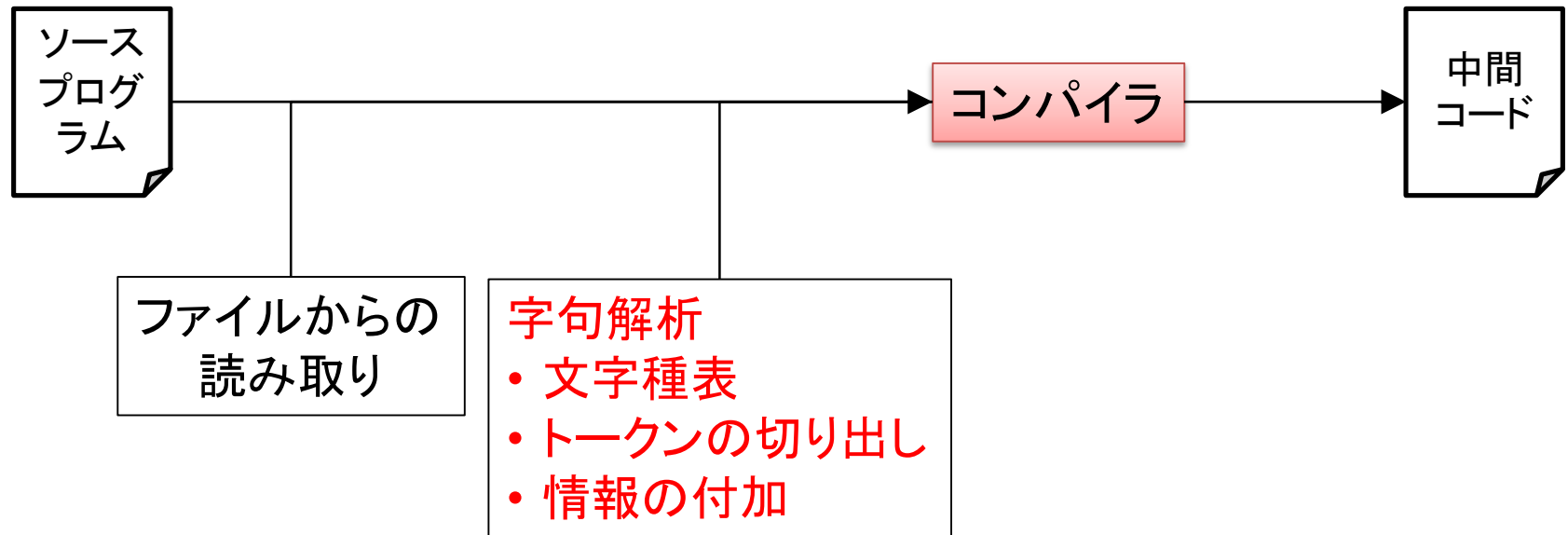
- 字句, 字句解析の役割, 字句の読み取り
- 情報の付加
- 文字種表
- 字句を表わす構造体



# これまでの実装

## ■ 字句解析

- 字句の切り出し
- 字句への情報の付加



# これまでの実装

---

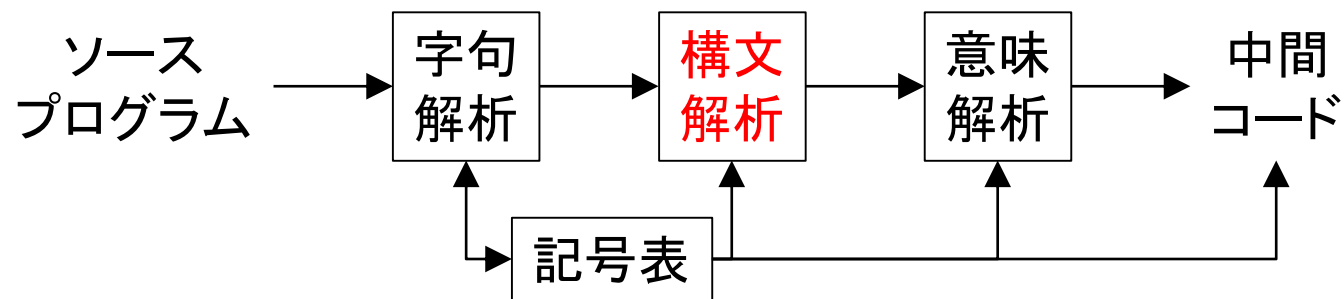
- 各種の定義
  - 文字種, トークンの種類, トークンの構造体
- 文字種表の実装
  - `Kind charKind[]`
  - `void initializeCharKind(void)`
- ファイルから次の1文字を取得
  - `int nextChar(void)`
- ファイルからトークン次のトークンを取得
  - `Token nextToken(void)`

## ■ 構文解析

- 演算子の優先順位による式の解析
  - ・ 逆ポーランド記法

## ■ プログラミング技法

- #defineを用いた表示の制御



# 構文解析に向けて

---

## ■ プログラミング言語の文法

- 文脈自由文法やBNF記法で記述される
- プログラムは文法に従う必要がある

## ■ 字句解析

- ソースプログラムはトークンに分割される

各トークンが正しくとも、  
トークンを並べただけでは文法に従った記号列とは限らない。

# 式の解析

- 構文解析では，式の解析が重要となる
  - 代入処理，条件式の比較要素など
- 式の解析の例

$(123 + 45) * 6$



字句解析

LParen	IntNum	Plus	IntNum	RParen	Multi	IntNum
(	123	+	45	)	*	6



式の解析(評価)

1008



構文解析では，式の解析の他に，制御文の解析も行なう

# 式の表現

---

## ■ 演算子の位置による表現

- 中置記法 ... 演算子を操作対象の中間に置く
- 前置記法 ... 演算子を操作対象の前に置く
- 後置記法 ... 演算子を操作対象の後に置く

## ■ 例:「aにbを足す」

- 中置記法:  $a + b$
- 前置記法:  $+ a b$
- 後置記法:  $a b +$



# 逆ポーランド記法

## ■ 演算子の優先順位を考慮した後置記法

### □ 隣接する演算子の優先順位(強さ)を比較する

- 演算子の優先順位は, 一般的な数学の規約
  - $()$ 内は先に計算される,  $*$ ,  $/$  は  $+$ ,  $-$  より優先順位が高い

### □ 式の処理

- 中置記法の式を逆ポーランド記法に変換する
- 逆ポーランド記法の式を評価して答を得る

中置記法の式	逆ポーランド記法
$a + b$	$a \ b \ +$
$a + b + c$	$a \ b \ + \ c \ +$
$a + b * c$	$a \ b \ c \ * \ +$
$(a + b) * c$	$a \ b \ + \ c \ *$

# 逆ポーランド記法

## ■ 特徴

### □ 区切り文字(デリミタ)が必要

- 中置記法では演算子が区切り文字となるため不要
- 字句解析後は, トークンで区切りが判別可能

### □ 式の評価(値を得る)に括弧が不要となる

- 逆ポーランド記法は演算子の優先順位を考慮する必要がない



コンパイラでは, 演算に必要な要素を読み込んだ後, 処理を決める逆ポーランド記法が適する.

# 逆ポーランド記法への変換

---

## ■ 逆ポーランド記法への変換の概要

- スタックを活用する
- トークンに対して,
  - 変数や定数ならば, そのまま出力する
  - 演算子ならば, 現在のスタックの上端にある要素と, トークンを比較して, 演算子の意味に沿う処理を行なう

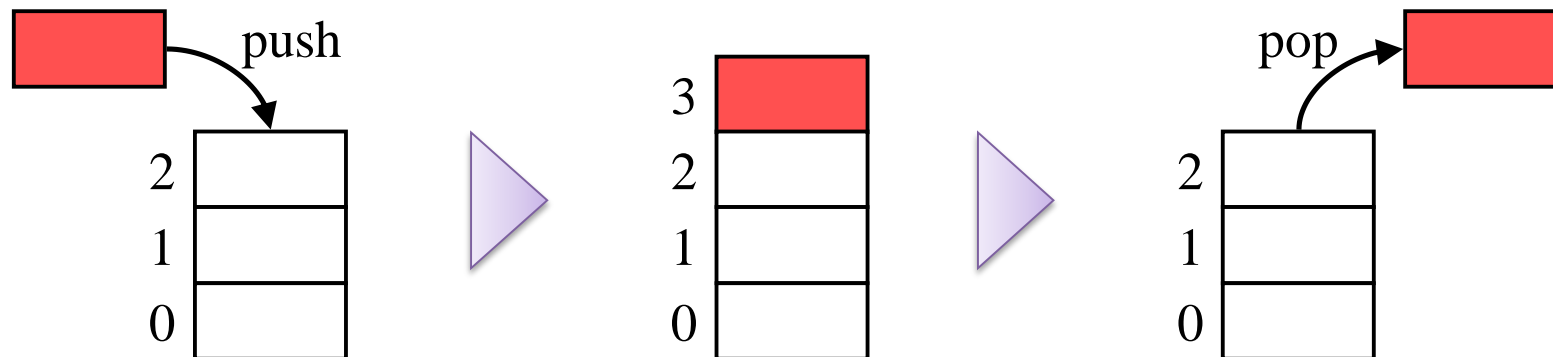
# スタック

## ■ スタック

- データを、後入れ先出しで保持するデータ構造
  - 後入れ先出し (LIFO: Last In First Out)

## ■ スタックの操作

- push : スタックにデータを追加する
- pop : 最後に追加されたデータを取り出す



# 逆ポーランド記法への変換

## ■ 中置記法から逆ポーランド記法への変換

1. 演算子の優先順位を定める
2. 末尾の文字まで, 以下の処理を繰り返す
  - 文字が変数または定数の場合, そのまま出力する
  - 文字が '(' ならば, pushする
  - 文字が ')' ならば, スタックの上端が '(' になるまでpopして, 出力する. ')'は読み捨てる.
    - '(' が存在しない場合はエラー
3. 末尾に至った場合, スタックに残る要素をpopして出力する
4. 上記以外の要素の場合, スタックの上端の優先順位が現在の文字の優先順位以上の間, popして出力する. その後, 現在の文字をpushする.

# 逆ポーランド記法への変換

## ■ 演算子の優先順位

### □ 演算子には優先順位がある

- 四則演算では,  
括弧内の演算 > 乗法と除法 > 加法と減法

### □ 優先順位表の例

優先順位	演算子
3	*, /
2	+, -
1	

# 逆ポーランド記法: 動作例

## ■ 中置記法の式: $(123 + 45) * 6$

トークン	スタック	出力
	$\epsilon$ (空)	
LParen, (		
	(	
IntNum, 123		
	(	123
Plus, +		
	(, +	123
IntNum, 45		
	(, +	123 45

トークン	スタック	出力
LParen, )		
	$\epsilon$ (空)	123 45 +
Multi, *		
	*	123 45 +
IntNum, 6		
	*	123 45 + 6
(末尾)		
	$\epsilon$ (空)	123 45 + 6 *

# 逆ポーランド記法の式の評価

## ■ 逆ポーランド記法の式の評価方法

1. スタックが空であることを確認する
2. 式の先頭から、トークンがある間、以下の処理を繰り返す
  - トークンが定数(や変数)ならば, pushする
  - トークンが演算子ならば, popしたトークンを $t_2$ , 続けてpopしたトークンを $t_1$ として, 演算 " $t_1$  演算子  $t_2$ " を行ない, 結果をトークンとしてpushする
3. 最後にスタックに残った値をpopし, 結果とする(スタックには1つのみ残る)



# 逆ポーランド記法の式の評価例

## ■ 逆ポーランド記法の式: 123 45 + 6 \*

トークン	スタック	補足
	$\epsilon$ (空)	
IntNum, 123		
	123	
IntNum, 45		
	123, 45	
Plus, +		
	168	$123 + 45 = 168$
IntNum, 6		
	168, 6	
Multi, *		
	1008	$168 * 6 = 1008$

# 後置記法と構文解析

---

- 後置記法でプログラムの構文解析が可能？
  - if - else文などは、後置記法では解析が困難
  
- 初期の式の評価方法
  - 括弧などによる構文の明示
    - 例: `"a+(b*(c+d)/e)-f"`
  
- 発展した構文解析
  - 下向き構文解析
  - 上向き構文解析

## ■ 構文解析

- 演算子の優先順位による式の解析
  - ・ 逆ポーランド記法

## ■ プログラミング技法

- #defineを用いた表示の制御

# #defineを用いた表示の制御

## ■ #define, #ifdef, #else, #endif

- マクロの定義の有無で処理を分ける

- 例

DEBUGが定義されているとき, #ifdef DEBUG と #endif の間のプログラムがコンパイルされる

```
#define DEBUG
:
#ifdef DEBUG
    printf("Valid if DEBUG is defined.");
#endif
```

# #defineを用いた表示の制御

## ■ 講義のサンプルコードでの使用例

### □ 動作確認用のメッセージ表示の有無に用いる

- プログラム開発中はメッセージを表示した方が分かりやすいが、最終版(提出版)のプログラムでは表示しない場合などに有効

```
#define VERBOSE
:
#ifdef VERBOSE
    printf("initializing charKind[]... ");
#endif
```



課題提出時はコメントアウトして, 定義しないこと.

# #defineを用いた表示の制御

## ■ gccのコンパイルオプションを用いた定義

- -D オプションにより, プログラム中で#defineしなくとも定義ができる

```
#include <stdio.h>

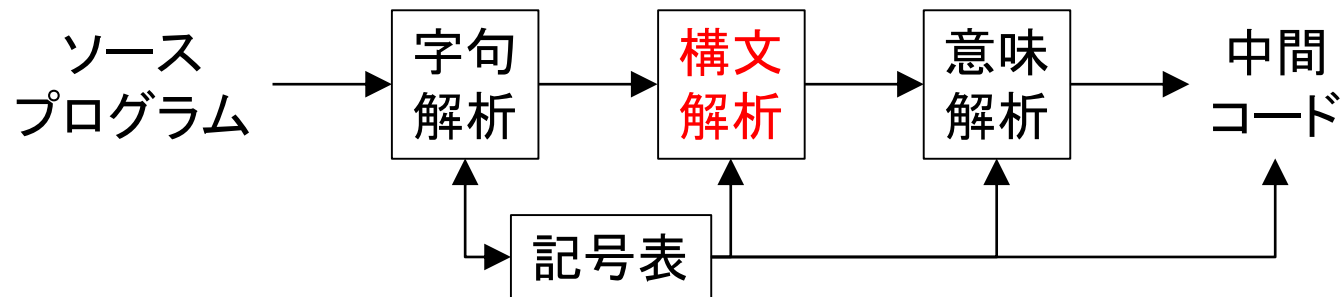
int main(int argc, char *argv[]) {
#ifdef DEBUG
    printf("DEBUG is defined¥n");
#else
    printf("DEBUG is not defined¥n");
#endif
    return 0;
}
```

# まとめ

## ■ 構文解析

### □ 演算子の優先順位による式の解析

- 逆ポーランド記法



# 予告: 中間試験

---

## ■ 第8回の内容

- 講義(主に試験問題の解説)
- 演習とレポート作成

## ■ 中間試験

- レポート形式. ただし, プログラム提出あり
- 提出期限は検討中.
  - 講義後1 or 2週間を予定



# 演習

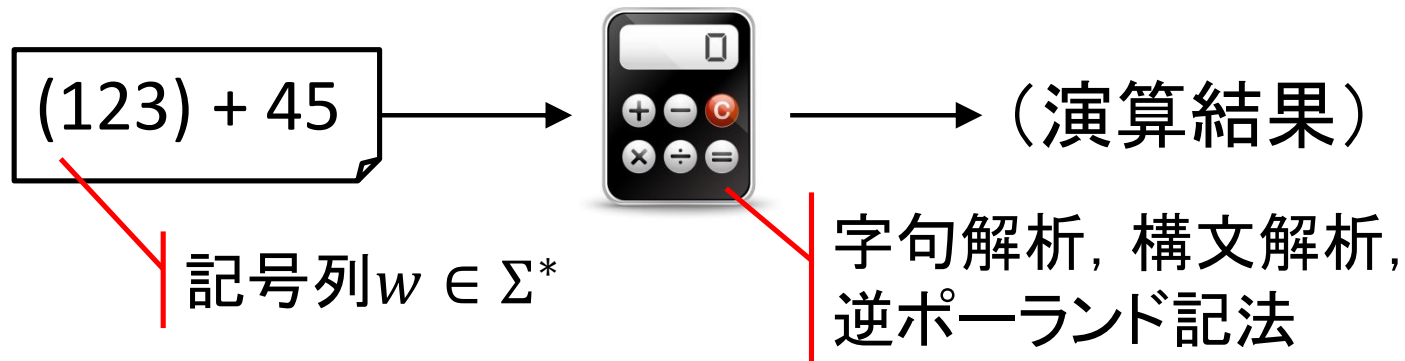
# 演習の目的

---

- 簡易電卓として、式の解析システムを実装
  - これまでに実装した字句解析をベースとする
    - ・ トークンの抽出
    - ・ トークンの種類(情報)の付加
    - ・ 空白文字の削除
    - ・ コメントの除去は行なわない
  - 四則演算に関する構文解析
  - インタプリタとして実現する
    - ・ 字句解析・構文解析を行なった後、同じプログラムで式を解析して演算結果を求める

# 簡易電卓

- 入力: 数式が記述されたファイル
- 出力: 数式の演算結果
  - 数式に誤りがある場合, エラーを出力する
- 処理
  - 四則演算に関する字句解析と構文解析による
  - 逆ポーランド記法による演算



# 簡易電卓

---

## ■ 仕様

- 記号列 $w$ は, ファイルに記載された文字列
  - 空白記号をトークンの区切り文字とする
- アルファベット $\Sigma$ 
  - $0, 1, 2, \dots, 9, (, ), +, -, *, /$ , 空白文字
- 記号列 $w \in \Sigma^*$ は, 中置記法の式とする
- 数式を逆ポーランド記法で評価する

# 簡易電卓

## ■ 動作イメージ

$(123 + 45) * 6$  中置記法の式



逆ポーランド記法への変換

- スタック
- 演算子の優先順位

123 45 + 6 \* 逆ポーランド記法の式



逆ポーランド記法の式の評価

- スタック

1008

解

# 演習6-1: 演習の手順

---

## ■ 前提

- 字句解析は実装済みとする

## ■ 処理の手順

1. スタックの実装
2. 逆ポーランド記法への変換
3. 式の評価



## 演習6-1: Step0

---

- サンプルファイル "compiler06\_1\_step0.c"

# 演習6-1: Step1: スタックの実装

## ■ スタックの実装手順

- スタックを表わすグローバル変数stack
  - Token型の1次元配列
- スタックサイズは十分確保されているとする
  - 本演習では, 十分と思われる領域を確保する
- 操作pop, pushを実装する
  - スタックの中身を表示する関数があると便利



本演習では1次元配列で実装しますが, オブジェクト指向言語や可変長配列を用いると, きれいに実装できます



# 演習6-1: Step1: スタックの実装

## ■ push

関数名	push	
引数	Token	スタックにpushするトークン
戻り値	void	
機能	スタックstackに引数で与えられるToken型変数をpushする. スタックの要素の上限を超えて関数が呼ばれた場合, "stack overflow"のエラーを表示し, プログラムを終了する.	



この関数は, サンプルソースでは実装済み.



本演習では, stack overflowでプログラム終了としていますが, エラー処理を継続する実装も考えられます.

# 演習6-1: Step1: スタックの実装

## ■ pop

関数名	pop	
引数	void	
戻り値	Token	スタックからpopしたトークン
機能	スタックstackからpopしたトークンを返す. スタックに要素がない場合, "stack underflow" のエラーを表示し, プログラムを終了する.	



本演習では, stack underflowでプログラム終了としていますが, エラー処理を継続する実装も考えられます.

# 演習6-1: Step1: スタックの実装

## ■ スタックの内容を表示する関数

関数名	printStack	
引数	void	
戻り値	void	
機能	スタックstackの内容を表示する.	



この関数は, サンプルソースでは実装済み.



この関数は主にデバック用です.  
作りこむ必要はありません.

## 演習6-1: Step2: 逆ポーランド記法への変換

---

### ■ 逆ポーランド記法への変換手順

#### □ 優先順位(表)の実装

- トークンの種類(Token.kind)から優先順位を得る
- この演習では,  $\Sigma$ に含まれる文字のみ対象とする

#### □ 逆ポーランド記法への変換

- 変換アルゴリズムの実装

## 演習6-1: Step2-1: 逆ポーランド記法への変換

### ■ 優先順位(表)の実装

□トークンの種類(Token.kind)から優先順位を得る

- 本演習では,  $\Sigma$ に含まれる文字のみ対象とする

関数名	getOrder	
引数	Token	優先順位を判定するトークン
戻り値	int	優先順位
機能	引数のトークンに対して, 下の優先順位を返す. <ul style="list-style-type: none"><li>• Multi('*'), Div('/') : 3</li><li>• Plus('+'), Minus('-') : 2</li><li>• LParen('(') : 1</li><li>• 上記以外 : -1</li></ul>	

## 演習6-1: Step2-2: 逆ポーランド記法への変換

### ■ 中置記法の式を逆ポーランド記法に変換する

#### □ 入出力: Token型の1次元配列

- 各配列は, 終端記号としてToken.kindがNULLTokenのトークンが格納されているとし, 配列の長さは与えないとする.

関数名	rpn	
引数	Token *	中置記法のトークン列
戻り値	Token *	逆ポーランド記法のトークン列
機能	Token型の1次元配列に格納されている中置記法のトークン列から, スタックとトークンの優先順位表を用いて, 逆ポーランド記法のトークン列に変換してToken型の1次元配列に格納する.	



RPN: Reverse Polish Notation



## 演習6-1: Step2-2: 逆ポーランド記法への変換

### □ エラー処理

- '(' が不足している ( ')' が多い ) とき,  
"error: less '('"  
を表示し, プログラムを正常終了する.
- ')' が不足している ( '(' が多い ) とき,  
"error: much '('"  
を表示し, プログラムを正常終了する.



## 演習6-1: Step3: 式の評価

---

- 逆ポーランド記法の式の評価
  - 逆ポーランド記法で記述されている式を評価する



## 演習6-1: Step3: 式の評価

- 逆ポーランド記法の式を評価する関数
  - この講義では,  $\Sigma$ に含まれる文字のみ対象とする

関数名	evaluate	
引数	Token *	逆ポーランド記法のトークン列
戻り値	int	式を評価した値
機能	逆ポーランド記法のトークン列が表わす式の値を評価し, その値を返す.	

# 演習6-1: Step4: 式の評価機能の実装

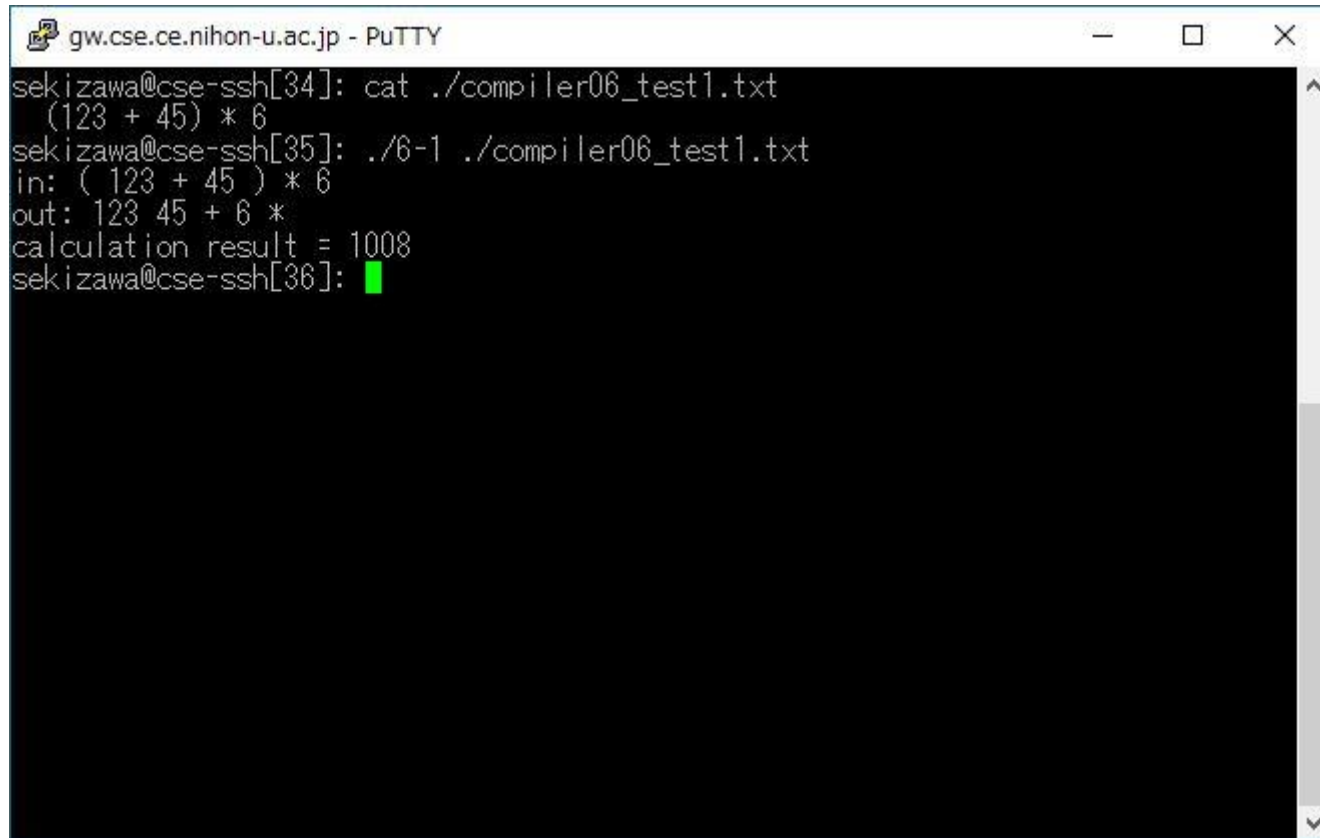
---

## ■ 式の解析・評価機能の実装

- これまでのStepでの実装を統合し, 式の解析・評価機能を実現する
- 実装上の仕様
  - 演習課題提出システムの課題を参照のこと  
(エラーメッセージなど)

# 演習6-1: 実行例

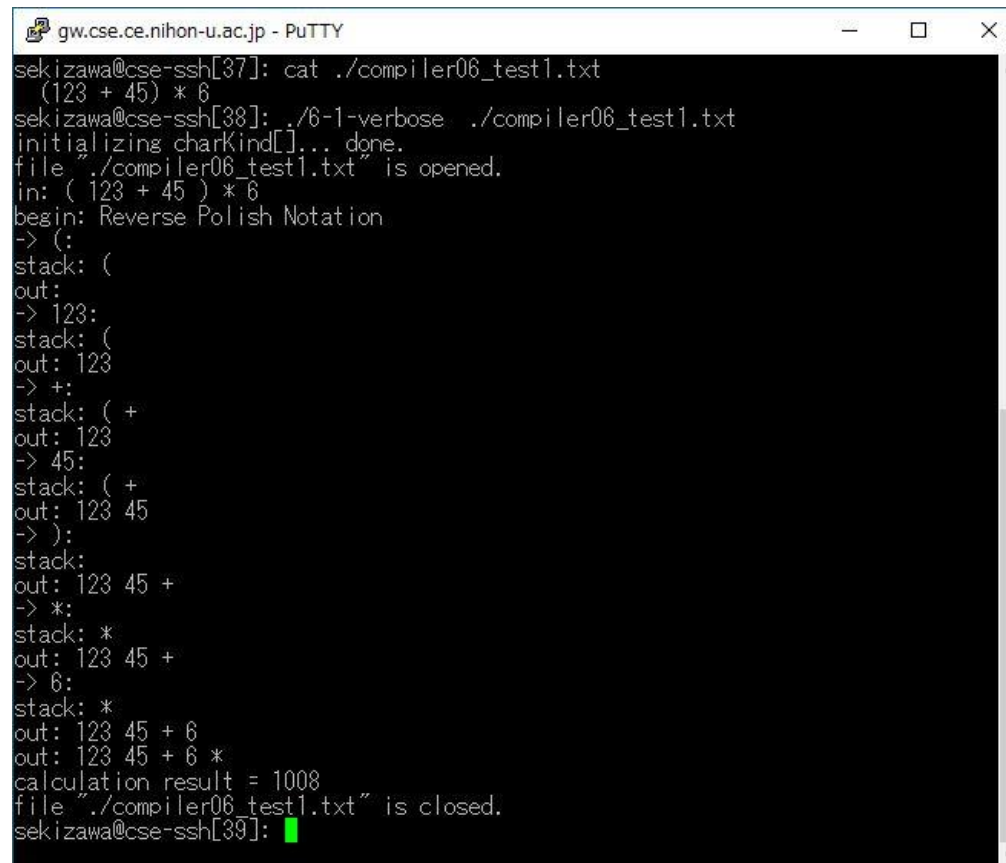
- " (123 + 45) \* 6 "
  - "compiler06\_test1.txt" (答: 1008)



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[34]: cat ./compiler06_test1.txt
(123 + 45) * 6
sekizawa@cse-ssh[35]: ./6-1 ./compiler06_test1.txt
in: ( 123 + 45 ) * 6
out: 123 45 + 6 *
calculation result = 1008
sekizawa@cse-ssh[36]:
```

# 演習6-1: 実行例

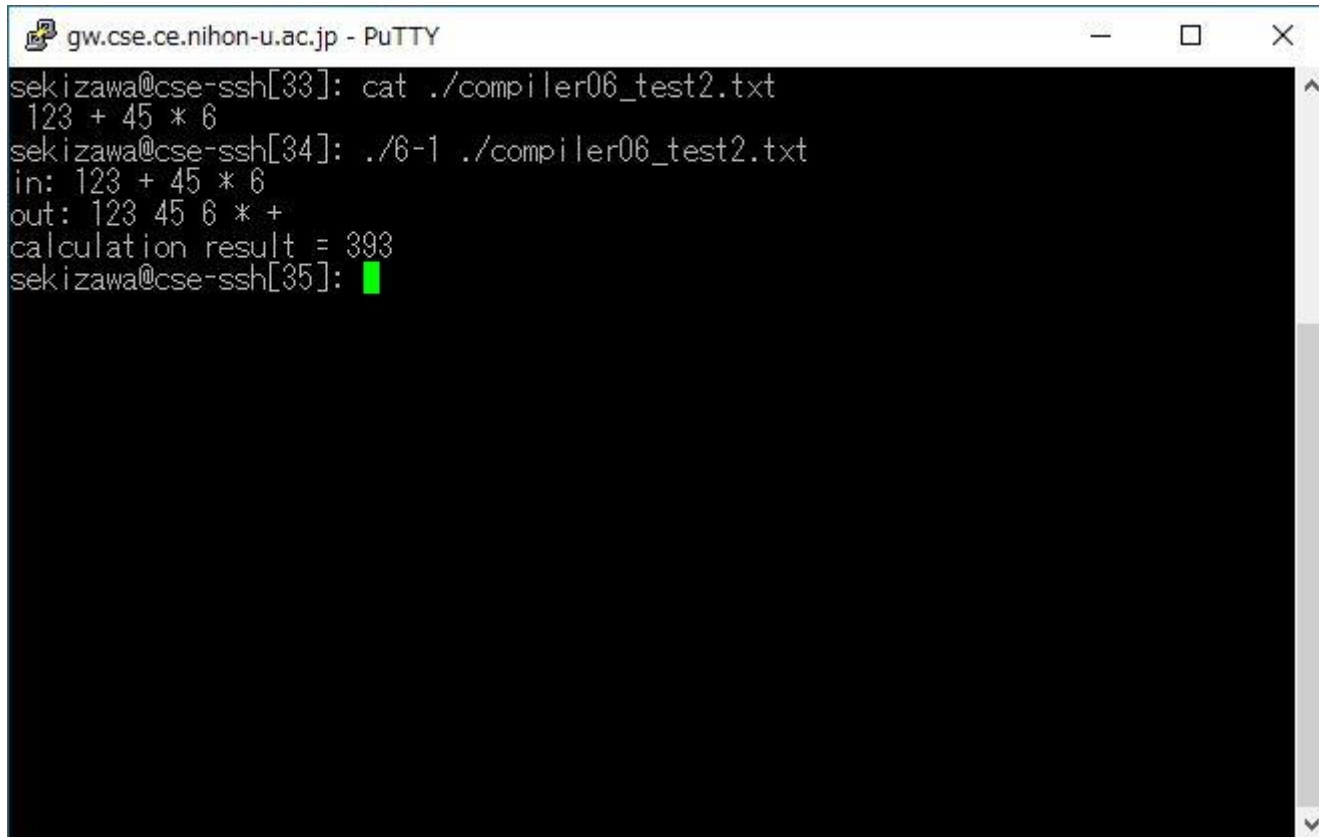
- "(123 + 45) \* 6" (#define VERBOSE あり)
- "compiler06\_test1.txt" (答: 1008)



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[37]: cat ./compiler06_test1.txt
(123 + 45) * 6
sekizawa@cse-ssh[38]: ./6-1-verbose ./compiler06_test1.txt
initializing charKind[]... done.
file "./compiler06_test1.txt" is opened.
in: ( 123 + 45 ) * 6
begin: Reverse Polish Notation
-> (:
stack: (
out:
-> 123:
stack: (
out: 123
-> +:
stack: ( +
out: 123
-> 45:
stack: ( +
out: 123 45
-> ):
stack:
out: 123 45 +
-> *:
stack: *
out: 123 45 +
-> 6:
stack: *
out: 123 45 + 6
out: 123 45 + 6 *
calculation result = 1008
file "./compiler06_test1.txt" is closed.
sekizawa@cse-ssh[39]:
```

# 演習6-1: 実行例

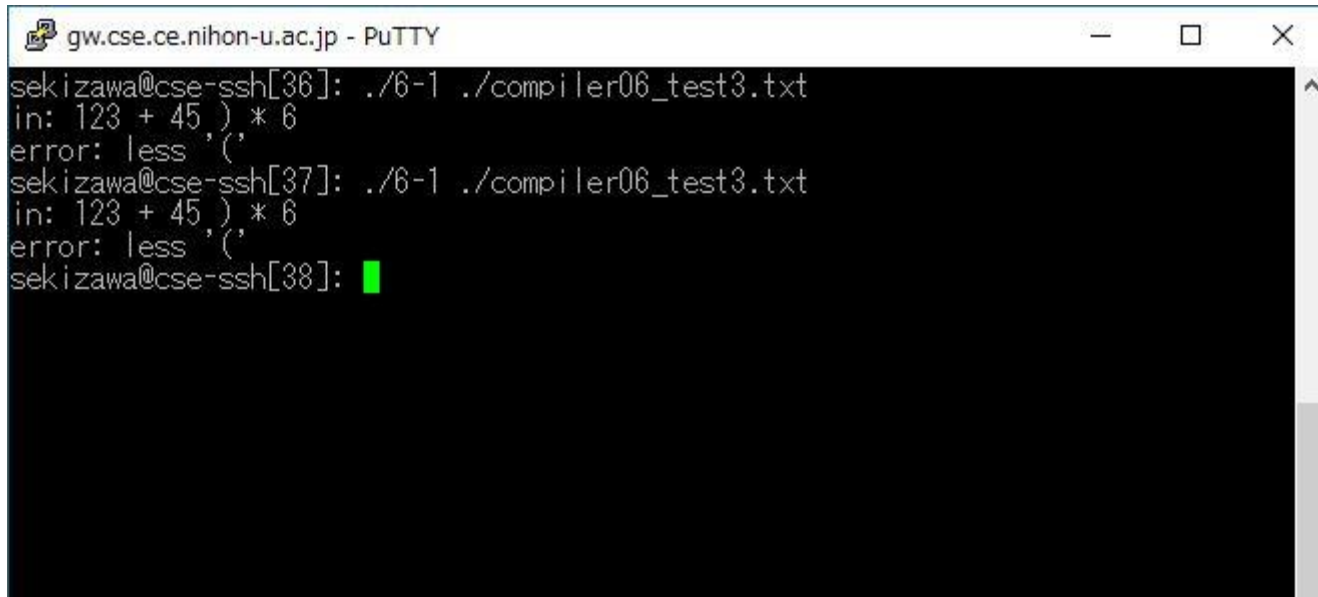
- " 123 + 45 \* 6 "
- "compiler06\_test2.txt" (答: 393)



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[33]: cat ./compiler06_test2.txt
123 + 45 * 6
sekizawa@cse-ssh[34]: ./6-1 ./compiler06_test2.txt
in: 123 + 45 * 6
out: 123 45 6 * +
calculation result = 393
sekizawa@cse-ssh[35]:
```

## 演習6-1: 実行例

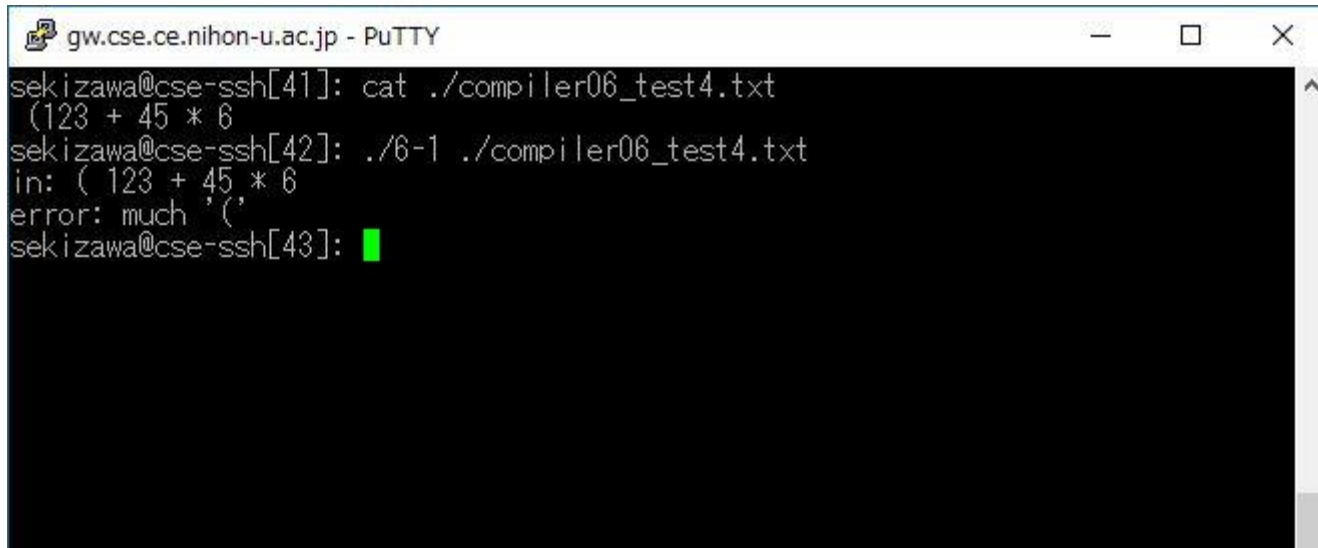
- " 123 + 45 ) \* 6"
  - "compiler06\_test3.txt"
  - (がなく, )が記述されているためエラー



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[36]: ./6-1 ./compiler06_test3.txt
in: 123 + 45 ) * 6
error: less '('
sekizawa@cse-ssh[37]: ./6-1 ./compiler06_test3.txt
in: 123 + 45 ) * 6
error: less '('
sekizawa@cse-ssh[38]:
```

## 演習6-1: 実行例

- "(123 + 45 \* 6"
  - "compiler06\_test4.txt"
  - (に対応する)がないためエラー



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[41]: cat ./compiler06_test4.txt
(123 + 45 * 6
sekizawa@cse-ssh[42]: ./6-1 ./compiler06_test4.txt
in: ( 123 + 45 * 6
error: much '('
sekizawa@cse-ssh[43]:
```

## 演習6-1: 実行例

- test5: "2 / 1 + 3/1"
- test6: "1 / 2"
- test7: "( ((12)\*3 + ((4))) )"
- test8: "1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + (略. 2回繰り返す)"
- test9:  
"((((((((((((((((((((((((((((((((123))))))))))))))))))))))))))))"
- test10: "123456789 + (987654321 \* 1) / 1"



test9はスタックのサイズによって結果が変わります. スタックのサイズが24(以下)の場合はstack overflowとなります.