

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

連絡

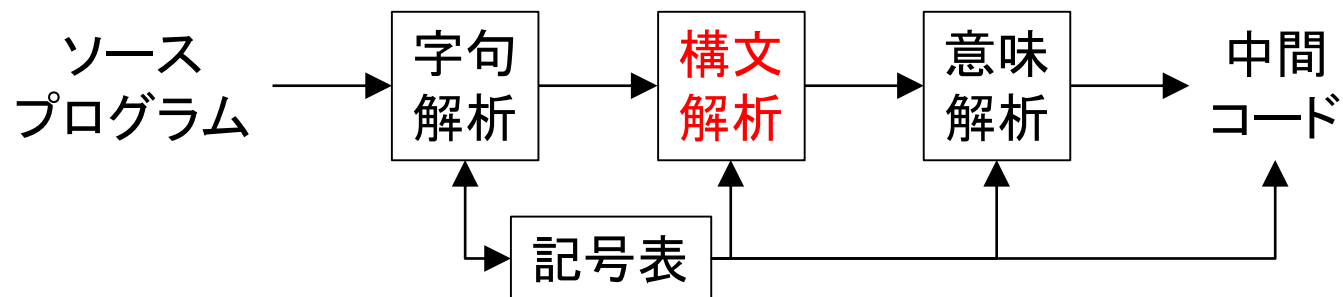
- 第8回の内容: 中間試験
 - 問題の提示
 - レポート形式のため, 第8回は演習.

復習

■ 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
 - 式の解析には有効だが、他の構文への適用は難しい
- 演習: 簡易電卓(四則演算)の実装

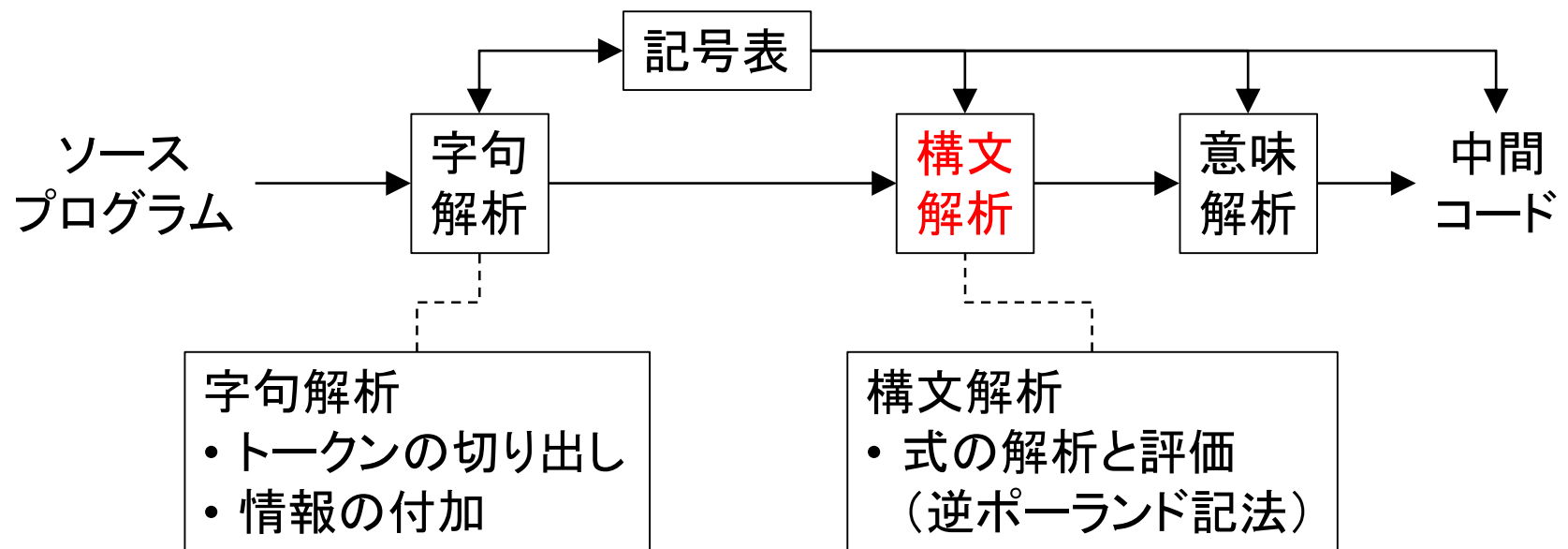


これまでの実装

■ 字句解析

■ 構文解析

□ 式の解析と評価



Tips: 用語

■ 字句解析器

- lexical analyzer (レキシカルアナライザー)
- lexer (レキサー)
- tokenizer (トークナイザー)

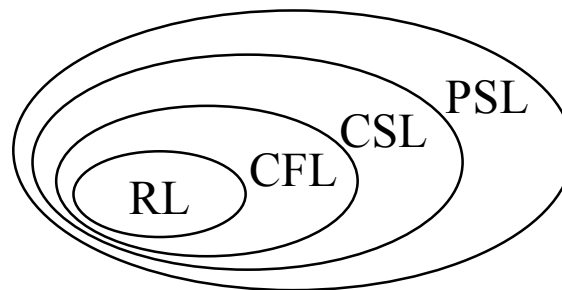
■ 構文解析器

- parser (パーサー)

- 文脈自由言語, 文脈自由文法, プッシュダウンオートマトン
 - 構文木, あいまい性
- 構文解析
 - 演算子の優先順位による式の解析
 - 構文解析木
 - 構文解析法
 - 下降型構文解析, 上昇型構文解析
- 下降型構文解析の実装関連
 - 式の評価, 記号表
- 演習

文法で生成される言語

- 正規文法 (RG)
 - ⇔ 正規言語 (RL: regular language)
- 文脈自由文法 (CFG)
 - ⇔ 文脈自由言語 (CFL: context-free language)
- 文脈依存文法 (CSG)
 - ⇔ 文脈依存言語 (CSL: context-sensitive language)
- 句構造文法 (PSG)
 - ⇔ 句構造言語 (PSL: phrase structure language)



文脈自由文法

■ 文脈自由文法 (CFL: Context Free Grammar)

■ $G = (N, \Sigma, P, S)$

□ N : 非終端記号の有限集合

□ Σ : 終端記号の有限集合

□ P : 書き換え規則の有限集合

- ただし、書き換え規則が $A \rightarrow \alpha$ の形式となっているもの。ここで、 $A \in N, \alpha \in (\Sigma \cup N)^*$ 。

□ $S \in N$: 初期記号



(拡張)BNF記法は、文脈自由文法による構文定義の記法

補足：文脈自由文法，文脈自由言語

■ 文脈自由文法

- CFGが導出する文形式は一般に複数個の非終端記号を含む. どの非終端記号に対しても書き換え規則を適用してもよい

■ 文脈自由言語 (CFL)

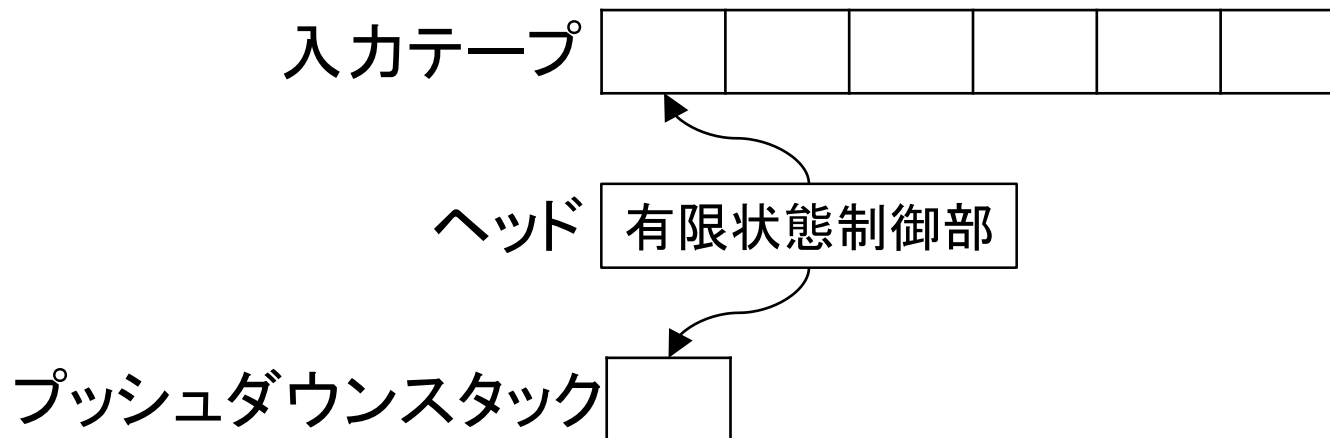
- CFG $G_{cfg} = (N, \Sigma, P, S)$ の初期記号から導出できる終端記号の列の集合

$$L(G_{cfg}) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$$

プッシュダウンオートマトン

■ プッシュダウンオートマトン (PDA)

- 有限オートマトンに, 「プッシュダウンスタック」と呼ばれる補助記憶を付け加えたオートマトン
- 次の2種類がある
 - 決定性プッシュダウンオートマトン (DPDA)
 - 非決定性プッシュダウンオートマトン (NPDA)



非決定性プッシュダウンオートマトン

- NPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
 - Q : 状態の有限集合
 - Σ : 入力記号の有限集合
 - Γ : スタック記号の有限集合
 - $\delta: Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ の部分集合 $\rightarrow 2^{Q \times \Gamma^*}$,
状態遷移関数
 - $q_0 \in Q$: 初期状態
 - $Z_0 \in \Gamma$: 初期スタック記号
 - $F \subseteq Q$: 受理状態の有限集合

文脈自由文法とプッシュダウンオートマトン

- CFGから, NPDAを構成できることより,
$$L(CFG) \subseteq L(NPDA)$$
- NPDAから, CFGを構成できることより,
$$L(NPDA) \subseteq L(CFG)$$
- \therefore

$$L(NPDA) = CFL = L(CFG)$$



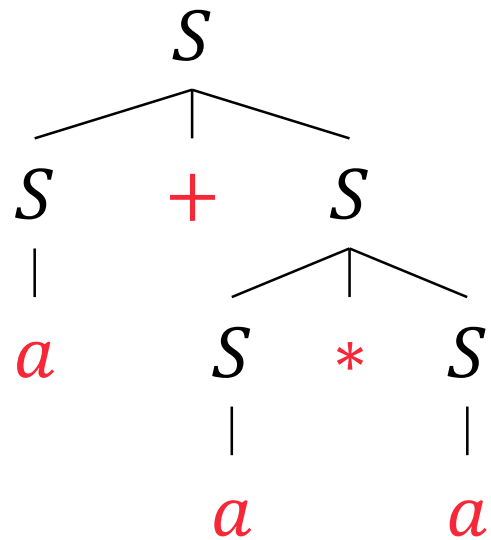
文脈自由文法で定義される言語を, プッシュダウンオートマトンで実装できる.

詳細はオートマトンの教科書等を参照のこと

- 文脈自由言語, 文脈自由文法, プッシュダウンオートマトン
 - 構文木, あいまい性
- 構文解析
 - 演算子の優先順位による式の解析
 - 構文解析木
 - 構文解析法
 - 下降型構文解析, 上昇型構文解析
- 下降型構文解析の実装関連
 - 式の評価, 記号表
- 演習

構文木(導出木)

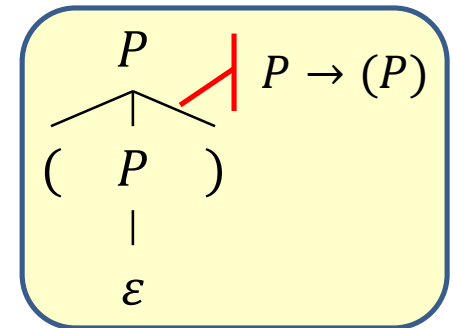
- 構文木とは, 導出を表現する木構造
 - 語の導出の過程を表現する
 - ・ コンパイラなどで頻繁に用いられる構造



構文木の構成

■ 文法 $G = (N, \Sigma, P, S)$ の構文木とは

1. 初期記号を根とする
2. 葉でない節点のラベルは N の要素
3. 葉のラベルは、次のいずれか
 - 非終端記号（導出途中の節点）
 - 終端記号（導出が終了した葉）
 - 空記号 ε （その葉が親の唯一の子である場合のみ）
4. 節点のラベルが A で、その子のラベルが左から X_1, X_2, \dots, X_k であるのは、 $A \rightarrow X_1 X_2 \dots X_k \in P$ のときのみ。
 - いずれかの X が ε であるのは、それが唯一の子で、 $A \rightarrow \varepsilon$ が書き換え規則であるときのみ。

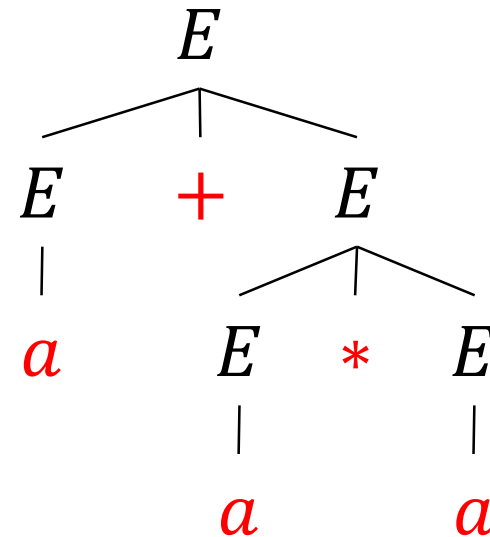


例: 文脈自由文法と構文木

- $G = (\{E\}, \{a, +, *, (,)\}, P, E)$, $P = \{E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow a\}$ とする.

□ 終端記号列 $a + a * a$ の導出

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow a + E \\ &\Rightarrow a + E * E \\ &\Rightarrow a + a * E \\ &\Rightarrow a + a * a \end{aligned}$$



構文木

あいまい性

■ あいまい

- 終端記号列だけでは、その構文(文の解釈)が一意に定まらないこと
 - ・ 文法 G はあいまいであるという

■ 本質的にあいまい

- あるCFL L に対して、 L を生成するいかなるCFGもあいまいなとき、言語 L は本質的にあいまいという。

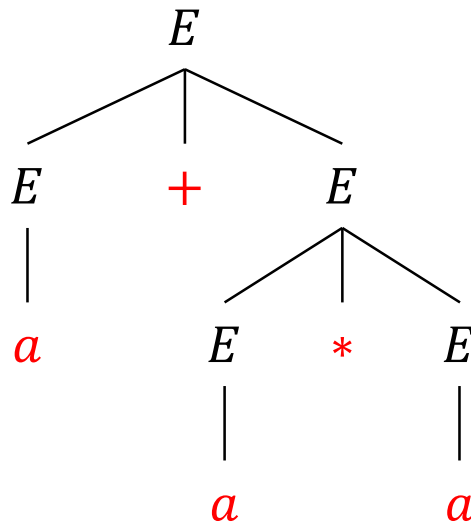


与えられたCFLに対して、それが本質的にあいまいであるか否かを判定するアルゴリズムは存在しない

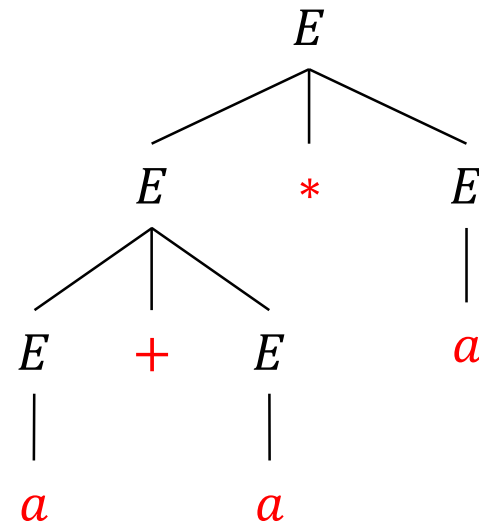
例: あいまいな文法

$G = (\{E\}, \{a, +, *, (,)\}, P, E)$, $P = \{E \rightarrow E + E, E \rightarrow E * E, E \rightarrow (E), E \rightarrow a\}$ とする.

■ 終端記号列 $a + a * a$



$a + \{a * a\}$



$\{a + a\} * a$

- 文脈自由言語, 文脈自由文法, プッシュダウンオートマトン
 - 構文木, あいまい性
- 構文解析
 - 演算子の優先順位による式の解析
 - 構文解析木
 - 構文解析法
 - 下降型構文解析, 上昇型構文解析
- 下降型構文解析の実装関連
 - 式の評価, 記号表
- 演習

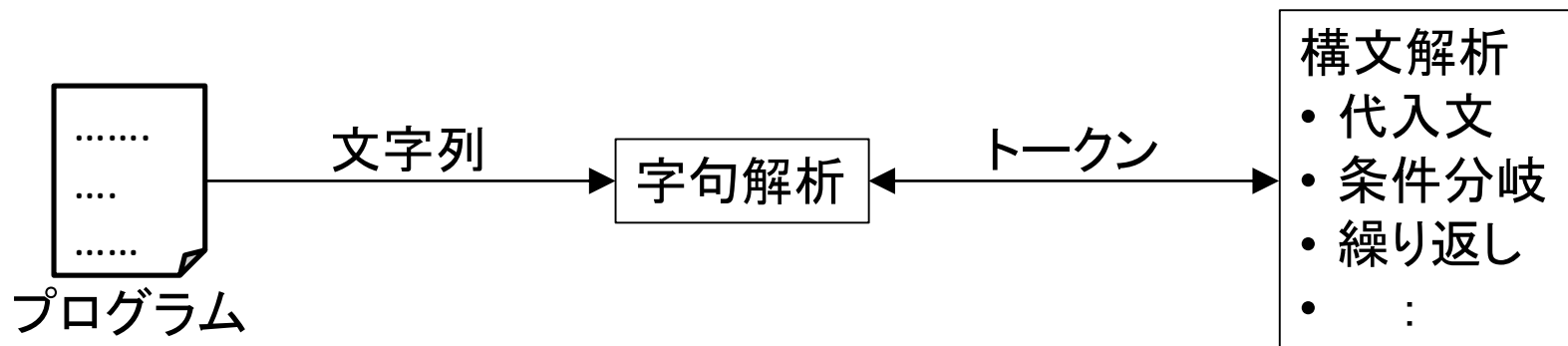
構文解析の役割

- 文脈自由文法 G と語 (トークン) $w \in L(G)$ が与えられたとき;

- 構文規則に従って, 語の構造を明らかにする
(語の導出過程を求める)

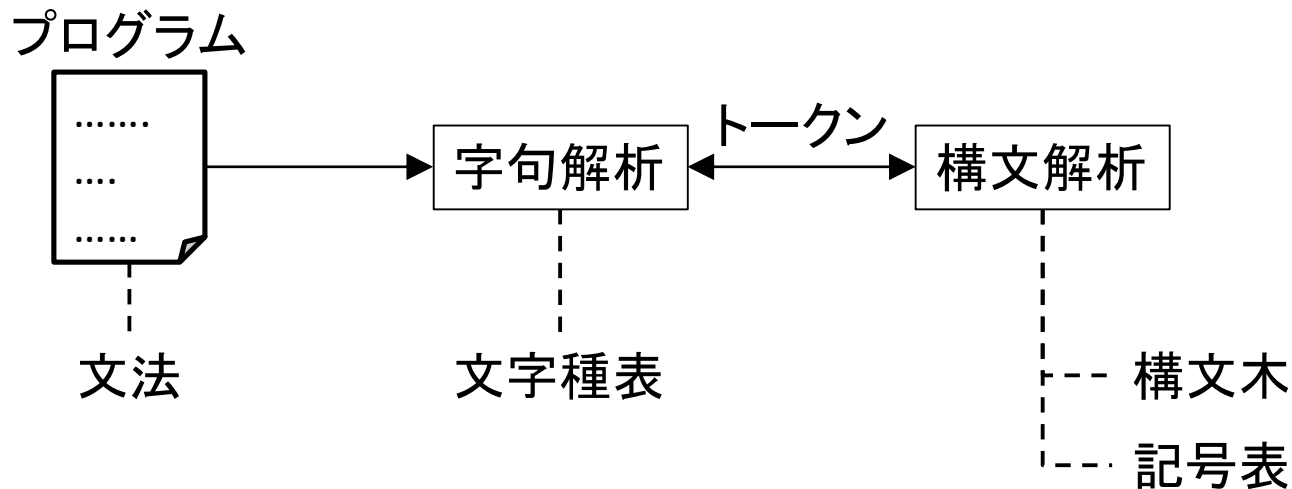
$$S(= \omega_0) \Rightarrow \omega_1 \Rightarrow \omega_2 \Rightarrow^* \omega_i \Rightarrow^* \omega_n (= w)$$

- 語の意味を確定させる



構文解析木(構文木)

- 構文解析の結果を表わす木構造
 - 節は非終端記号となる
 - 葉は終端記号となる
 - 解析木の葉を左から並べたものが式となる



例: 構文規則と構文木

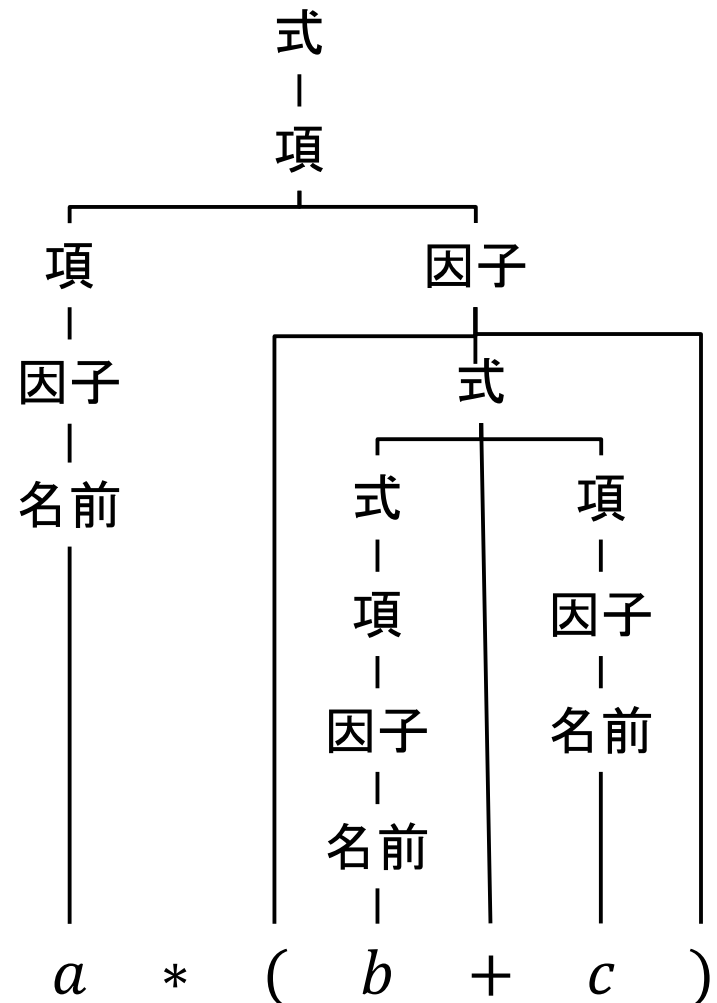
■ 式の構文規則

$\langle \text{式} \rangle ::= \langle \text{項} \rangle \mid \langle \text{式} \rangle + \langle \text{項} \rangle$
 $\langle \text{項} \rangle ::= \langle \text{因子} \rangle \mid \langle \text{項} \rangle * \langle \text{因子} \rangle$
 $\langle \text{因子} \rangle ::= \text{名前} \mid (\text{式})$

□ 導出される式の例

- a
- $a + b$
- $a * (b + c)$

□ 構文木 (右の図)



- 文脈自由言語, 文脈自由文法, プッシュダウンオートマトン
 - 構文木, あいまい性
- 構文解析
 - 演算子の優先順位による式の解析
 - 構文解析木
 - 構文解析法
 - 下降型構文解析, 上昇型構文解析
- 下降型構文解析の実装関連
 - 式の評価, 記号表
- 演習

構文解析法

■ 下降型構文解析(下向き構文解析)

- 構文解析木を, 初期記号 S が付された構文木の根から, 語 w が付された葉に向かって, 解析する
 - 非終端記号と, 終端記号列の情報を利用する

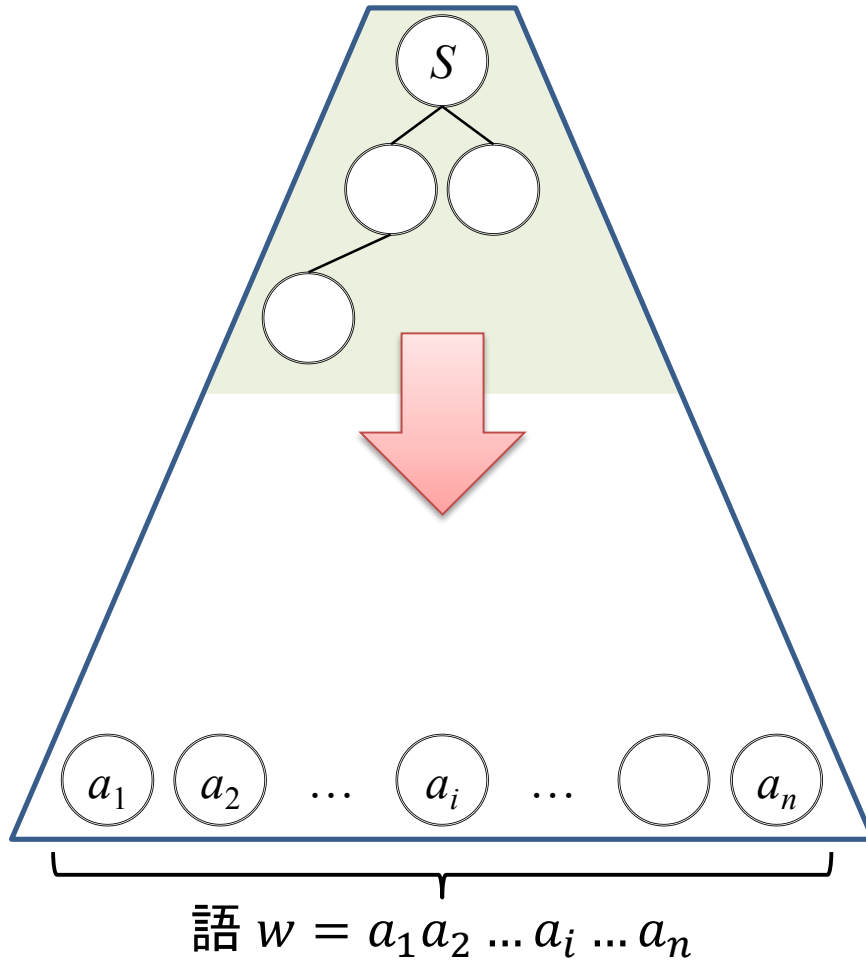
■ 上昇型構文解析(上向き構文解析)

- 構文解析木を, 語 w が付された葉から, 初期記号 S が付された構文木の根に向かって, 解析する
 - 終端記号列の情報を利用する

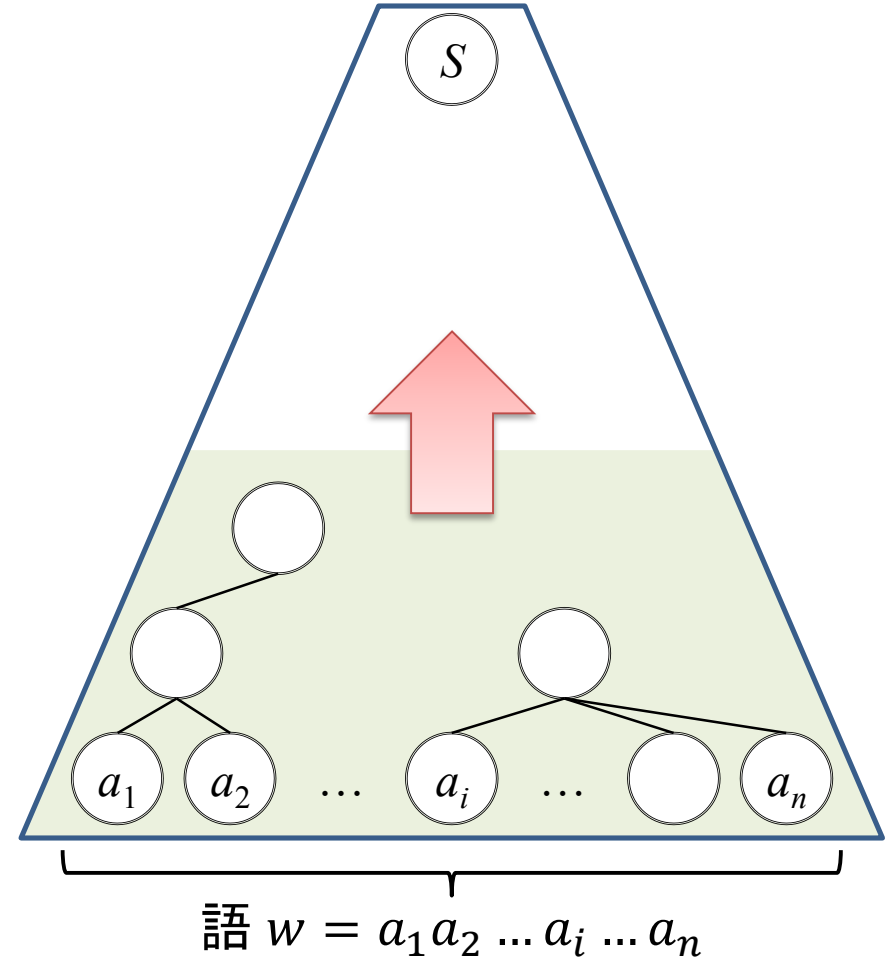


上昇型は, 非終端記号の情報が使えないため, 一般的には構文解析が難しい.

下降型/上昇型構文解析



下降型構文解析



上昇型構文解析

下降型構文解析

■ 問題点

□ 左再帰性

- 左再帰とは, 文法 G に $A \Rightarrow^+ A\alpha$ なる導出があること
- 文法が左再帰的に記述されている場合, 構文解析が行えない場合がある

□ 共通の左因子

- 1つの非終端記号を左辺として, 右辺に共通の左因子がある複数の書き換え規則がある場合, 適用する規則が一意に決まらない場合がある

左再帰性の問題

■ 直感的な説明

□ $A \rightarrow A\alpha, A \rightarrow \beta$ の書き換え規則とする

- $A \Rightarrow^+ A\alpha^k \Rightarrow \beta\alpha^k \Rightarrow^* w$ なる導出を考えたとき, 繰り返し回数 k を決定できない

■ プログラムによる直感的な説明

□ $A \rightarrow A\alpha$ の処理プログラムを考えると;

```
void A(void) {  
    A();  
    alpha();  
}
```

関数 $A()$ の,
無限回の呼び出しが発生する

共通の左因子の問題

■ 直感的な説明

- 書き換え規則 $A \rightarrow b, A \rightarrow bc$ があるとする
 - A を書き換える際, 適用する規則が決められない

■ プログラム的な説明

```
if P then S1
```

```
if P then S1 else S2
```

elseの有無が判明するまで,
構文が定まらない

下降型構文解析の問題の解決策

- 書き換え規則の修正
 - 左再帰の除去
 - 左因子のくり出し
- 問題のない言語処理系の規定

- 文脈自由言語, 文脈自由文法, プッシュダウンオートマトン
 - 構文木, あいまい性
- 構文解析
 - 演算子の優先順位による式の解析
 - 構文解析木
 - 構文解析法: 下降型構文解析, 上昇型構文解析
- プログラミング技法
 - 下降型構文解析の実装手法
 - 式の評価
 - 変数の扱いと記号表

(再帰的)下降型構文解析と実装

■ BNFの記法

- (一般に)再帰的に記述される
- 各構文規則を関数に1対1対応させる

$\langle \text{式} \rangle ::= \langle \text{項} \rangle \mid \langle \text{式} \rangle + \langle \text{項} \rangle$
$\langle \text{項} \rangle ::= \langle \text{因子} \rangle \mid \langle \text{項} \rangle * \langle \text{因子} \rangle$
$\langle \text{因子} \rangle ::= \text{名前} \mid (\text{式})$

例: BNFによる式の構文規則

(再帰的) 下降型構文解析と実装

■ 例: BNFによる構文規則

□ 変数と数値, 演算子を利用できる文法

- 変数VARIABLE: 'a' - 'z', 'A' - 'Z' からなる記号列
- 数値NUMBER: '0' - '9' からなる記号列
- 演算子: '=', '+'

```
⟨statement⟩ ::= VARIABLE '=' ⟨expression⟩  
⟨expression⟩ ::= ⟨factor⟩  
                | ⟨factor⟩ '+' ⟨factor⟩  
⟨factor⟩      ::= NUMBER
```


導出される式の例

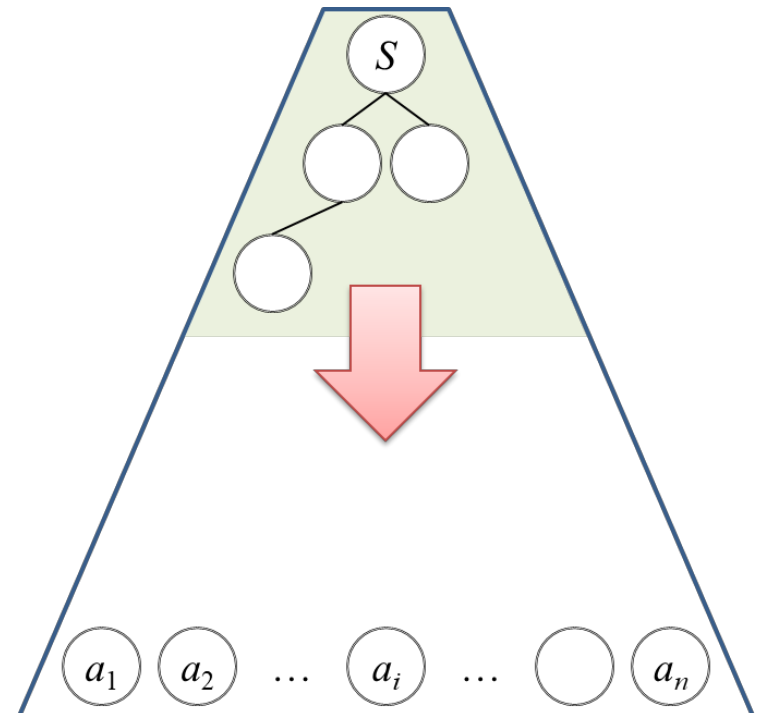
```
<statement> ::= VARIABLE '=' <expression>  
<expression> ::= <factor> | <factor> '+' <factor>  
<factor> ::= NUMBER
```

■ $abc = 12$

```
<statement>  
::= VARIABLE = <expression>  
::= abc = <factor>  
::= abc = NUMBER  
::= abc = 12
```

■ $x = 123 + 45$

```
<statement>  
::= VARIABLE = <expression>  
::= x = <factor> + <factor>  
::= x = NUMBER + NUMBER  
::= x = 123 + 45
```



(再帰的) 下降型構文解析と実装

■ 〈expression〉に対応する関数の実装例

□ 関数呼び出し前に, トークンが1つ読み込まれているとする

• グローバル変数 Token token; が宣言されているとする

```
〈expression〉 ::= 〈factor〉  
                | 〈factor〉 '+' 〈factor〉
```

```
void expression(void) {  
    factor();  
    switch (token.kind) {  
        case Plus:  
            token = nextToken();  
            factor();  
            break;  
    }
```

次に '+' があるときの処理

式の評価

■ スタックを活用する手法

□ 式 (`<expression>`) を処理する際,

- 構文木の葉 (終端記号) をスタックにpushする
(ここでは, 定数NUMBER)
- 演算要素が揃った後, 演算を行なう
- 演算結果をスタックにpushする
 - 演算終了後, スタックに残っている値が演算結果
 - 演算結果は, `<statement>` でのVARIABLEへの代入処理時に, スタックよりpopして変数に格納する
(スタックは空になる)

式の評価

■ 式の評価のコード例 (一部)

```
void expression(void) {
    Kind operator;
    factor();
    switch (token.kind) {
        case Plus:
            operator = token.kind;
            token = nextToken();
            factor();
            evaluate(operator);
            break;
        ...
    }
}
```

```
void factor() {
    ...
    switch (token.kind) {
        case IntNum:
            push(token);
            break;
    }
}
```

式の評価

■ 記号列 "x=123+45" の解析とスタック

トークン	処理する構文規則	スタック
x	statement	ϵ
=	statement	ϵ
123	term	push: 123
+	expression	123
45	term	push: 123, 45
	evaluate	pop, pop, push: 168
	statement	pop: ϵ

変数の扱いと記号表

■ 変数は処理の過程で複数回参照される

□ 複数回の参照の例

```
int score;  
score = 90;
```

2か所に出現する変数scoreは、同じ変数

□ 変数を「**記号表**」に登録して管理する

- 記号表は複数の登録情報を管理する
- 変数の登録内容: 型, 変数名, 値, ...



一般に、記号表には、変数名のみでなく、関数名なども登録されます。現時点での記号表は、初歩的な段階です。

記号表の役割

■ 変数の保持

□ 出現した変数の情報

- 変数は複数回参照される可能性があり, 同じ記号列の変数は, 同一の変数として扱われる

```
int score;  
score = 90;
```

2か所に出現する変数scoreは, 同じ変数



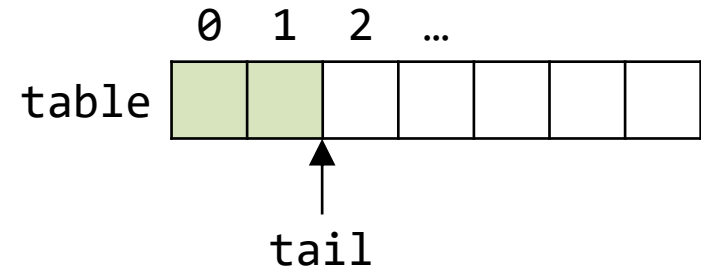
記号表の実装例と使い方は後述

記号表の実装

■ 構造体を用いた実装例

- 記号 (Token) を格納する1次元配列
- 配列の末尾を示すポインタ
(先頭は配列名から判明するため省略)

```
typedef struct {  
    Token table[TABLE_SIZE];  
    Token *tail;  
} SymbolTable;
```



記号表を操作する関数は別に定める必要があります



配列の長さを持たせる, 終端記号を入れる, など様々な実装方法があります

記号表の使い方(1)

■ 例

□ 初期状態では記号表に登録されていないとする

```
x = 1;  
y = 2 + 3;  
x = 4;  
z = y + 1;
```

	0	1	2	...			
table							

```
x = 1;  
y = 2 + 3;  
x = 4;  
z = y + 1;
```

	0	1	2	...			
table	x, 1						

1. 変数xが記号表に登録されているか調べる
2. 登録されていないため、変数xを記号表に登録する

記号表の使い方(2)

```
x = 1;
y = 2 + 3;
x = 4;
z = y + 1;
```

	0	1	2	...			
table	x,1	y,5					

1. 変数yが記号表に登録されているか調べる
2. 登録されていないため、変数yを記号表に登録する

```
x = 1;
y = 2 + 3;
x = 4;
z = y + 1;
```

	0	1	2	...			
table	x,4	y,5					

1. 変数xが記号表に登録されているか調べる
2. 変数xは登録されているため、記号表の変数xを更新する

記号表の使い方(3)

```
x = 1;  
y = 2 + 3;  
x = 4;  
z = y + 1;
```

	0	1	2	...			
table	x,1	y,5					

1. 変数yが記号表に登録されているか調べる
2. 変数yが登録されていない場合, エラー
3. 変数yが登録されている場合, yを取得する

```
x = 1;  
y = 2 + 3;  
x = 4;  
z = y + 1;
```

	0	1	2	...			
table	x,4	y,5	z,6				

1. 変数zが記号表に登録されているか調べる
2. 登録されていないため, 変数zを記号表に登録する

記号表の操作関数の例

- 初期化関数
 - 記号表を初期化する
- 要素の登録
 - 記号表に要素を登録する
- 要素の更新
 - すでに登録されている要素を更新する
- 要素の取得
 - 記号表から要素を得る
- ...

構文木と関数に対応させた表示

■ 定義済みマクロ `__func__`

- 現在の関数名が文字列として格納される
 - この識別子は関数内でのみ使用可能

```
#include <stdio.h>
void hoge(void);

int main(int argc, char *argv[]) {
    hoge();
    return 0;
}

void hoge(void) {
    printf("%s¥n", __func__);
}
```

実行結果

hoge

構文木と関数に対応させた表示

■ 構文規則を関数に1対1対応させる

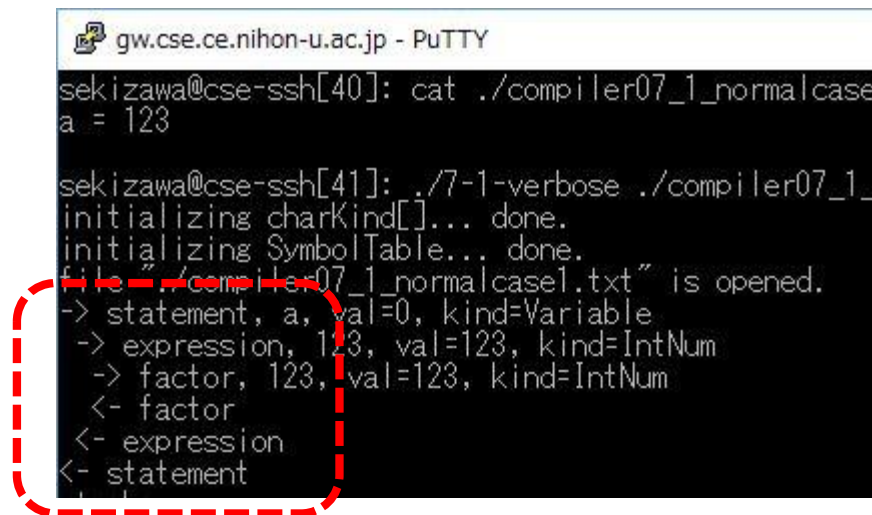
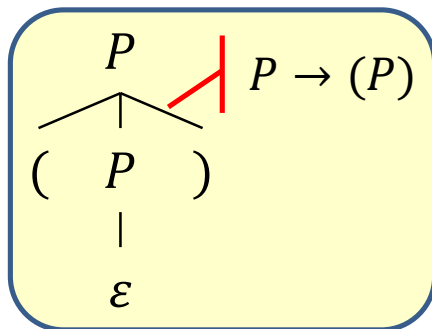
□ 関数の呼び出し・関数からの戻りの表示

- 関数に入るとき:

```
_printIndent(); printf("-> %s, ", __func__);
```

- 関数から出るとき:

```
_printIndent(); printf("<- %s¥n", __func__);
```



まとめ

- 文脈自由言語, 文脈自由文法, プッシュダウンオートマトン
 - 構文木, あいまい性
- 構文解析
 - 演算子の優先順位による式の解析
 - 構文解析木
 - 構文解析法
 - ・ 下降型構文解析, 上昇型構文解析
- 下降型構文解析の実装関連
 - 式の評価, 記号表

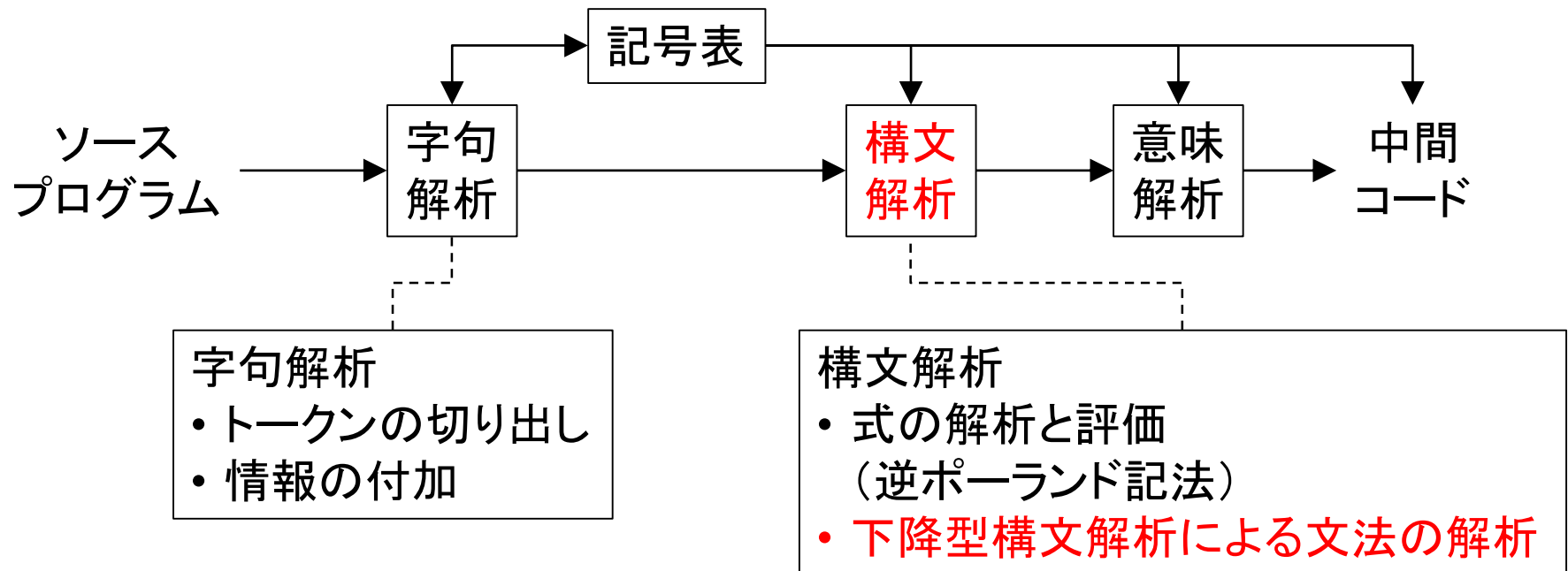
演習

目的

■ 構文解析

□ 式の解析と評価

□ 下降型構文解析器の実装（中間課題に向けて）



本演習で扱う文法

■ 変数と数値, 演算子を利用できる文法

- VARIABLE: '_', 'a' - 'z', 'A' - 'Z'からなる記号列
- 数値 NUMBER: '0' - '9'からなる記号列
- 演算子: '=', '+'
 - 本演習では演算子2つのみを扱う
- 初期記号は<statement>

```
<statement> ::= VARIABLE '=' <expression>
<expression> ::= <factor>
                | <factor> '+' <factor>
<factor>      ::= NUMBER
```

実装方針

- 各構文規則を関数化する
- 各構文規則の関数呼び出し前に, トークンが1つ読み込まれているとする
 - グローバル変数 `Token token;` が宣言されているとする

```
token = nextToken();  
statement();
```

例: 構文規則<statement>に対応する関数呼び出し

実装

■ 次の機能を実装する

- 構文解析器

- 記号表

- 構文解析後の;

- スタックの内容表示

- 正しく実装されていれば, スタックは空になっている
- step0に含まれています

- 記号表の表示

- 正しく実装されていれば, 入力ファイル中の記号が登録されている
- step0に含まれています

演習7-1: step0: 実装の雛型

■ 構文解析器の雛型

- "compiler07_1_step0.c"

■ 実装されている機能

- 構文解析

- void copyToken(Token *, Token*);
- bool checkToken(Token *t, Kind);

- 字句解析

- 記号表

- スタック

- データ表示用の関数

演習7-1: step0: 実装の雛型

- "compiler07_1_step0.c"の前提
 - 記号表
 - ・ グローバルとして定義
 - スタック
 - ・ グローバルとして定義

演習7-1: step1: 記号表の実装

- 構造体を用いた記号表と操作関数
 - 記号 (Token) を格納する1次元配列
 - 配列の末尾を示すポインタ

```
typedef struct {  
    Token table[TABLE_SIZE];  
    Token *tail;  
} SymbolTable;
```



Step0に含まれています



この講義の課題のレベルでは記号表を用いなくても実装可能ですが、記号表は今後必要となります。

演習7-1: step1: 記号表を操作する関数

■ 記号表の初期化

関数名	initializeTable	
引数	SymbolTable *	初期化する記号表
戻り値	void	
機能	引数で与えられる記号表を初期化する. • SymbolTable.tailをSymbolTable.tableの先頭要素とする	



Step0に含まれています



この課題ではグローバルに宣言された記号表1つしか使用しないため、引数はvoidとする実装も可能.

演習7-1: step1:記号表を操作する関数

■ 記号表に要素(トークン)を追加する関数

関数名	addTable	
引数	SymbolTable * Token	初期化する記号表 追加するトークン
戻り値	void	
機能	引数で与えられるトークンを記号表の末尾に追加する.	

演習7-1: step1:記号表を操作する関数

■ 記号表の内容を表示する関数

関数名	printTable	
引数	SymbolTable *	表示する記号表
戻り値	void	
機能	引数で与えられる記号表の内容を標準出力に出力する.	



Step0に含まれています

演習7-1: step1:記号表を操作する関数

■ 作ると便利な関数

□ 記号表の要素を取得する関数

関数名	searchTable	
引数	SymbolTable * char *	要素を探索する記号表 記号表から取得したいトークンの名前
戻り値	Token	記号表に格納されていたトークン
機能	引数で与えられた名前を持つトークンが記号表に登録されている場合, そのトークンを返す. 登録されていない場合, NULLTokenを返す.	



この関数は実装しなくても, この課題は実装可能.

演習7-1: step1:記号表を操作する関数

■ 作ると便利な関数

□ 記号表の要素を入れ替える関数

関数名	replaceElementOfTable	
引数	SymbolTable * Token	要素を入れ替える記号表 入れ替えたいトークン
戻り値	bool	
機能	引数で与えられたトークンと同じ名前を持つトークンが記号表に登録されている場合, 引数で与えられたトークンで記号表を更新する(入れ替える). この場合, trueを返す. 入れ替え候補のトークンが記号表に登録されていない場合, falseを返す.	



この関数は実装しなくても, この課題は実装可能.

演習7-1: step1:記号表を操作する関数

■ 作ると便利な関数

□ 記号表の要素を確認する関数

関数名	containTable	
引数	SymbolTable * char *	要素を確認する記号表 有無を確認したトークンの名前
戻り値	bool	
機能	引数で与えられた名前と同じ名前を持つトークンが記号表に登録されている場合, trueを返す. 登録されていない場合, falseを返す.	



この関数は実装しなくても, この課題は実装可能.

演習7-1: step2: 構文解析

■ 構文解析器を実装する

□ 各構文規則を関数化する

- void statement(void);
 - 〈statement〉の処理
- void expression(void);
 - 〈expression〉の処理
- void factor(void);
 - 〈factor〉の処理
- void evaluate(Kind kind);
 - 〈factor〉 '+' 〈factor〉で表現される式の評価
 - 演習6-1のevaluate関数とは異なることに注意

演習7-1: step2: 構文解析

□ 作ると便利な関数

- `bool checkToken(Token *t, Kind kind);`
 - トークンの種類を確認する
- `copyToken(Token *to, Token *from);`
 - トークンをコピーする

演習7-1: step2: 構文解析

■ 各構文規則を処理する関数

□ 仕様は略(講義資料を参照のこと)

- void statement(void);
- void expression(void);
- void factor(void);

演習7-1: step2: 構文解析

■ 式の評価関数

□ 本演習では $\langle \text{factor} \rangle \text{ '+' } \langle \text{factor} \rangle$

関数名	evaluate	
引数	Kind	演算子の種類
戻り値	void	
機能	<p>引数 Kind op で指定される演算を行なう. opが二項演算子の場合, グローバルのスタックから2つのトークン(t2, t1)をpopし, t1 op t2 の演算結果を求め, その演算結果をスタックにpushする.</p> <p>引数 Kind op が処理できないトークンの種類であった場合, "error: invalid Kind"と表示してプログラムを終了する.</p>	

演習7-1: step3: 動作確認

- 変数と数値, 演算子を利用できる構文解析の実装
 - これまでのStepでの実装を統合し, 変数と数値, 演算子を利用できる構文解析器を実現する
 - 実装上の仕様
 - 本スライドおよび演習課題提出システムを参照
 - エラーメッセージなど

演習7-1: step3: テストケース

■ ノーマルケース

□ "a = 123"

- "compiler07_1_normalcase1.txt"
- VARIABLE = NUMBER

□ "a = 123 + 45"

- "compiler07_1_normalcase2.txt"
- VARIABLE = NUMBER + NUMBER

演習7-1: step3: テストケース

■ エラーケース

- "a"
 - "compiler07_1_errorcase1.txt"
 - 変数VARIABLEし記述されていない
- "a ="
 - "compiler07_1_errorcase2.txt"
 - 代入の右辺がない
- "a = bc"
 - "compiler07_1_errorcase3.txt"
 - 代入の右辺が変数
- "a = 123 + 45 + 56"
 - "compiler07_1_errorcase4.txt"
 - 代入の右辺に, 文法で許されない式が記述されている

演習7-1: step3: 動作確認

■ "compiler07_1_normalcase1.txt"

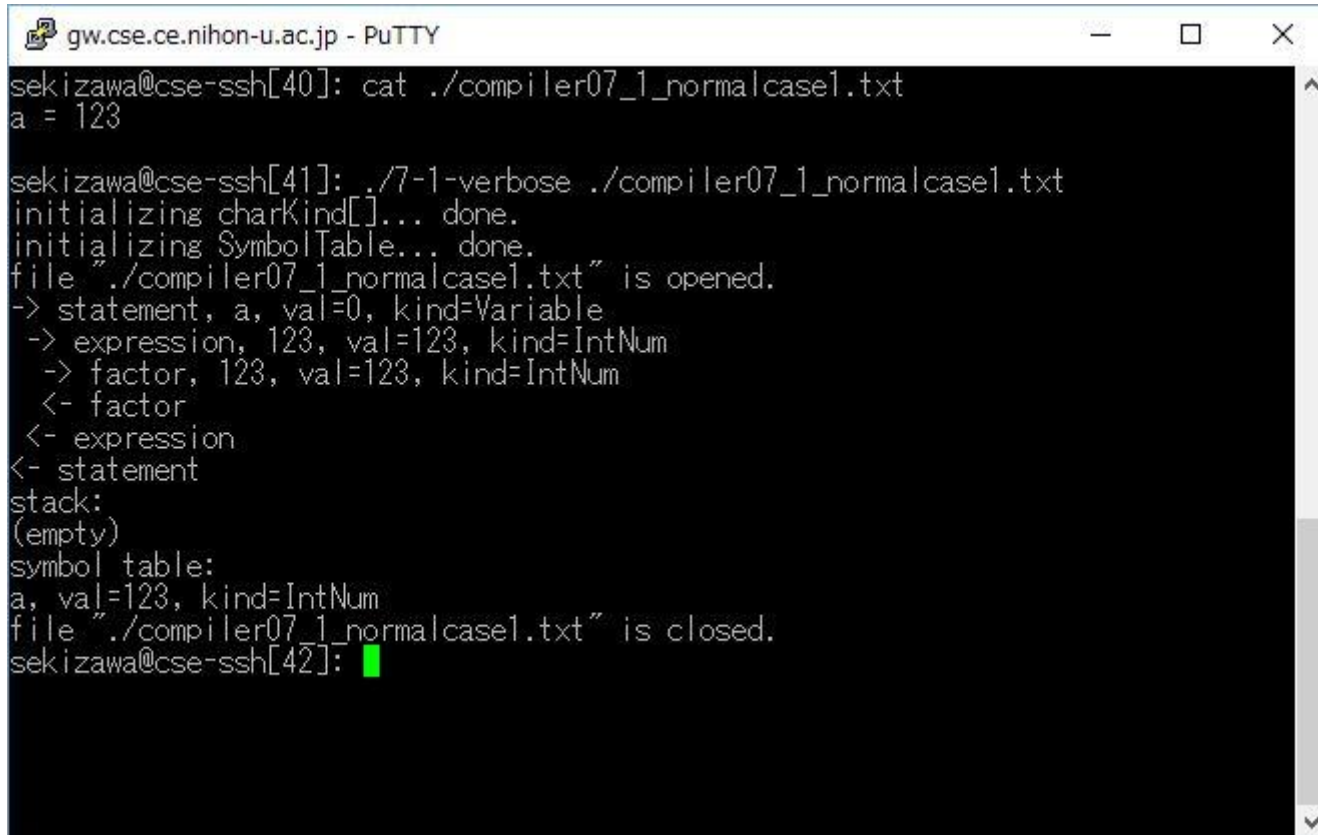


```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[37]: cat ./compiler07_1_normalcase1.txt
a = 123

sekizawa@cse-ssh[38]: ./7-1 ./compiler07_1_normalcase1.txt
stack:
(empty)
symbol table:
a, val=123, kind=IntNum
sekizawa@cse-ssh[39]:
```

演習7-1: step3: 動作確認

- "compiler07_1_normalcase1.txt"
 - VERBOSEあり

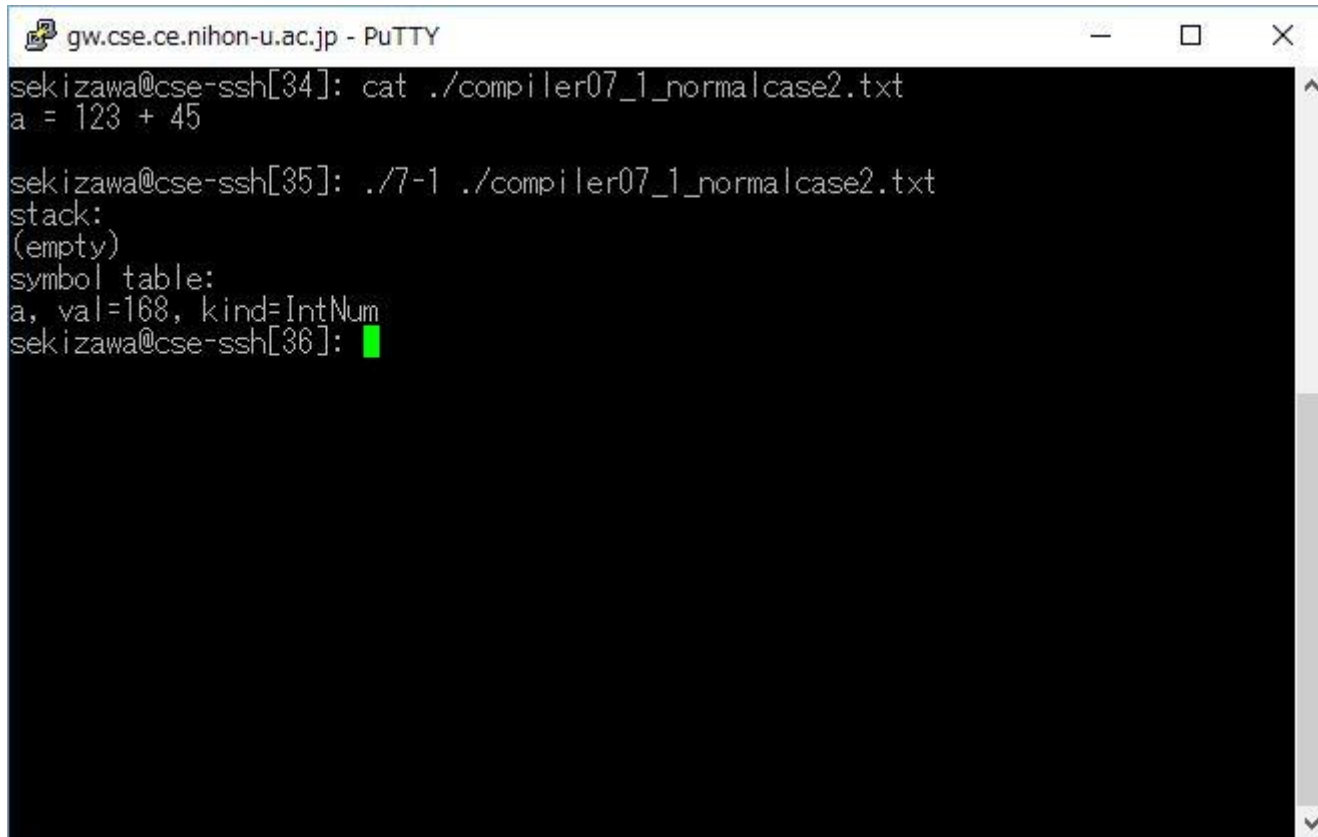


```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[40]: cat ./compiler07_1_normalcase1.txt
a = 123

sekizawa@cse-ssh[41]: ./7-1-verbose ./compiler07_1_normalcase1.txt
initializing charKind[]... done.
initializing SymbolTable... done.
file "./compiler07_1_normalcase1.txt" is opened.
-> statement, a, val=0, kind=Variable
  -> expression, 123, val=123, kind=IntNum
    -> factor, 123, val=123, kind=IntNum
      <- factor
    <- expression
  <- statement
stack:
(empty)
symbol table:
a, val=123, kind=IntNum
file "./compiler07_1_normalcase1.txt" is closed.
sekizawa@cse-ssh[42]:
```

演習7-1: step3: 動作確認

■ "compiler07_2_normalcase1.txt"

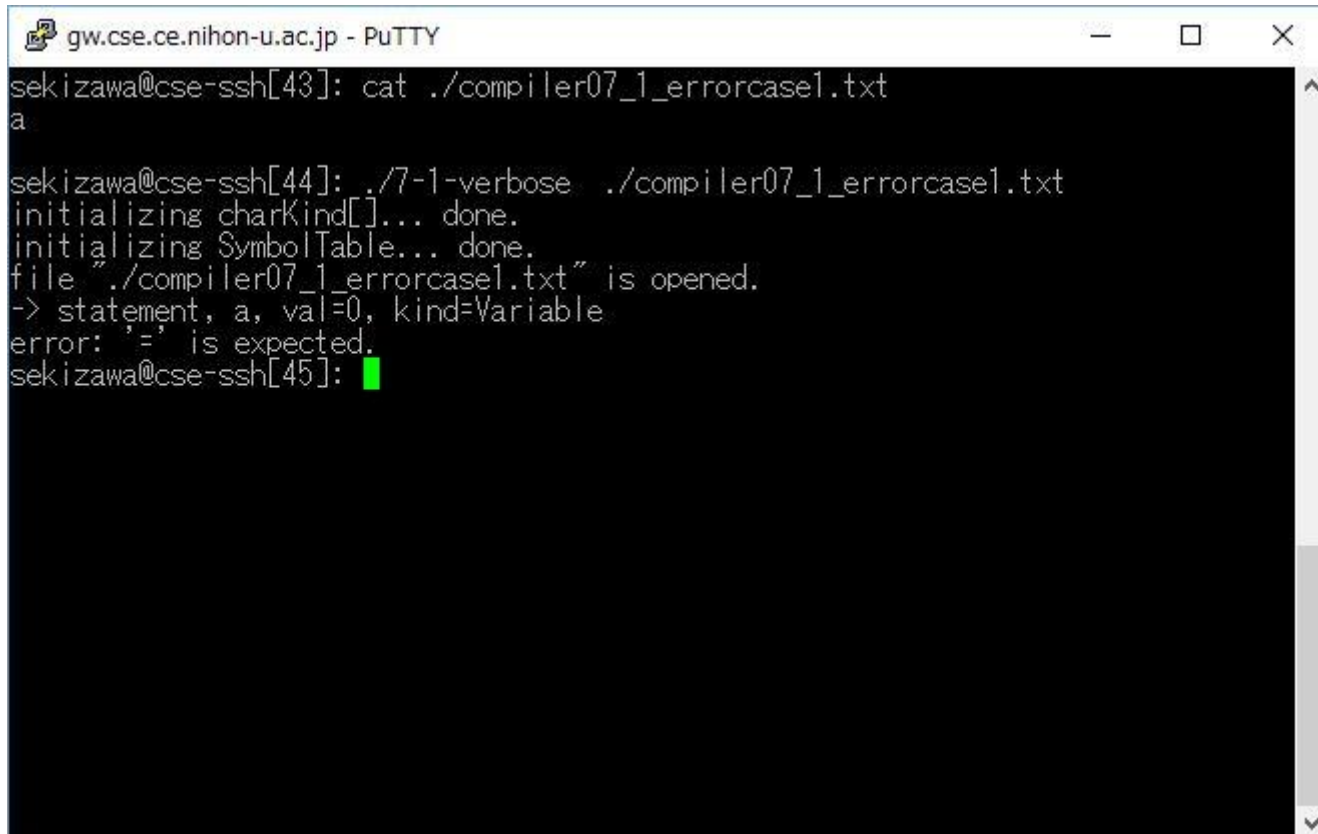


```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[34]: cat ./compiler07_1_normalcase2.txt
a = 123 + 45

sekizawa@cse-ssh[35]: ./7-1 ./compiler07_1_normalcase2.txt
stack:
(empty)
symbol table:
a, val=168, kind=IntNum
sekizawa@cse-ssh[36]:
```

演習7-1: step3: 動作確認

- "compiler07_1_errorcase1.txt"
 - VERBOSEあり

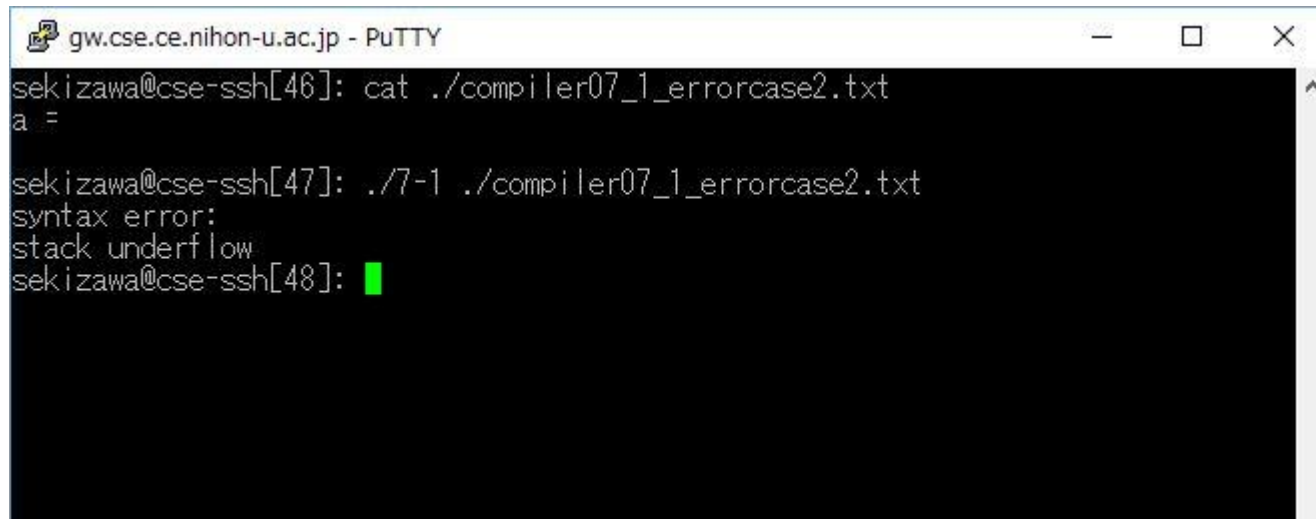


```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[43]: cat ./compiler07_1_errorcase1.txt
a

sekizawa@cse-ssh[44]: ./7-1-verbose ./compiler07_1_errorcase1.txt
initializing charKind[]... done.
initializing SymbolTable... done.
file "./compiler07_1_errorcase1.txt" is opened.
-> statement, a, val=0, kind=Variable
error: '=' is expected.
sekizawa@cse-ssh[45]:
```


演習7-1: step3: 動作確認

- "compiler07_1_errorcase2.txt"
 - 代入の右辺がない



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[46]: cat ./compiler07_1_errorcase2.txt
a =

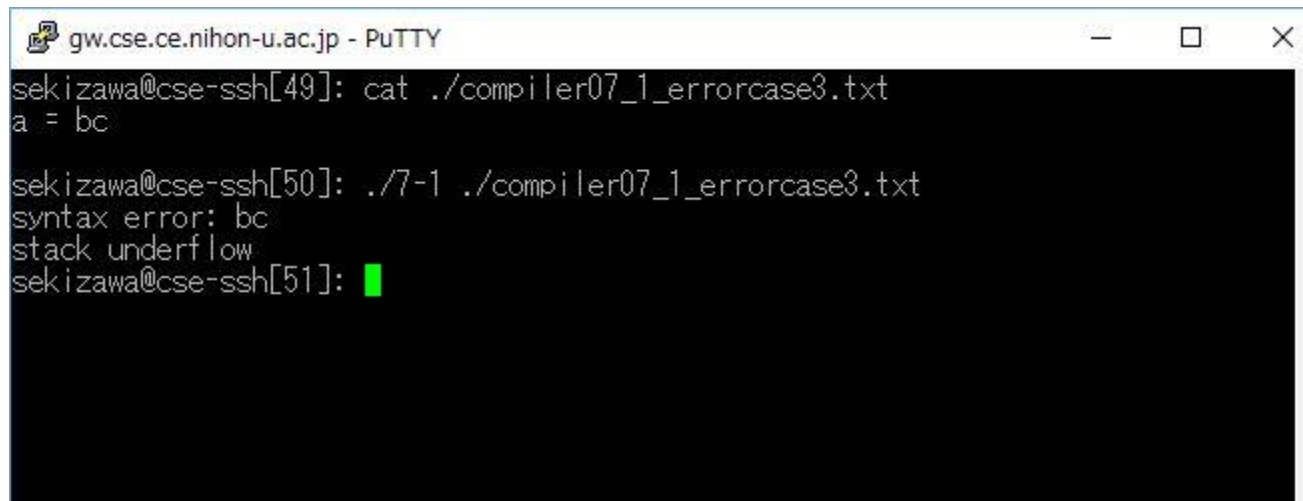
sekizawa@cse-ssh[47]: ./7-1 ./compiler07_1_errorcase2.txt
syntax error:
stack underflow
sekizawa@cse-ssh[48]: █
```



expressionでエラー判定をしていないため、evaluateでstack underflowが発生しています。エラー判定も可能です。

演習7-1: step3: 動作確認

- "compiler07_1_errorcase3.txt"
 - 代入の右辺が変数



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[49]: cat ./compiler07_1_errorcase3.txt
a = bc

sekizawa@cse-ssh[50]: ./7-1 ./compiler07_1_errorcase3.txt
syntax error: bc
stack underflow
sekizawa@cse-ssh[51]: █
```



expressionでエラー判定をしていないため、evaluateでstack underflowが発生していますがエラー判定も可能です。

演習7-1: step3: 動作確認

■ "compiler07_1_errorcase4.txt"

- 代入の右辺に文法から導出できない式が記述されている
 - EOFの前にトークンが残ってしまう



```
gw.cse.ce.nihon-u.ac.jp - PuTTY
sekizawa@cse-ssh[58]: cat ./compiler07_1_errorcase4.txt
a = 123 + 45 + 56

sekizawa@cse-ssh[59]: ./7-1-verbose ./compiler07_1_errorcase4.txt
initializing charKind[]... done.
initializing SymbolTable... done.
file "./compiler07_1_errorcase4.txt" is opened.
-> statement, a, val=0, kind=Variable
-> expression, 123, val=123, kind=IntNum
  -> factor, 123, val=123, kind=IntNum
    <- factor
  -> factor, 45, val=45, kind=IntNum
    <- factor
  <- expression
<- statement
error: token(s) remaining
sekizawa@cse-ssh[60]:
```