

# アジェンダ

## 1. 改造部分の解説

### 1.1 パージング処理

### 1.2 アクセッサの導入

### 1.3 バインド処理

## 2. 言語仕様検討事項

## 1. 改造部分の解説

### **1.1 パージング処理**

#### 1.2 アクセッサの導入

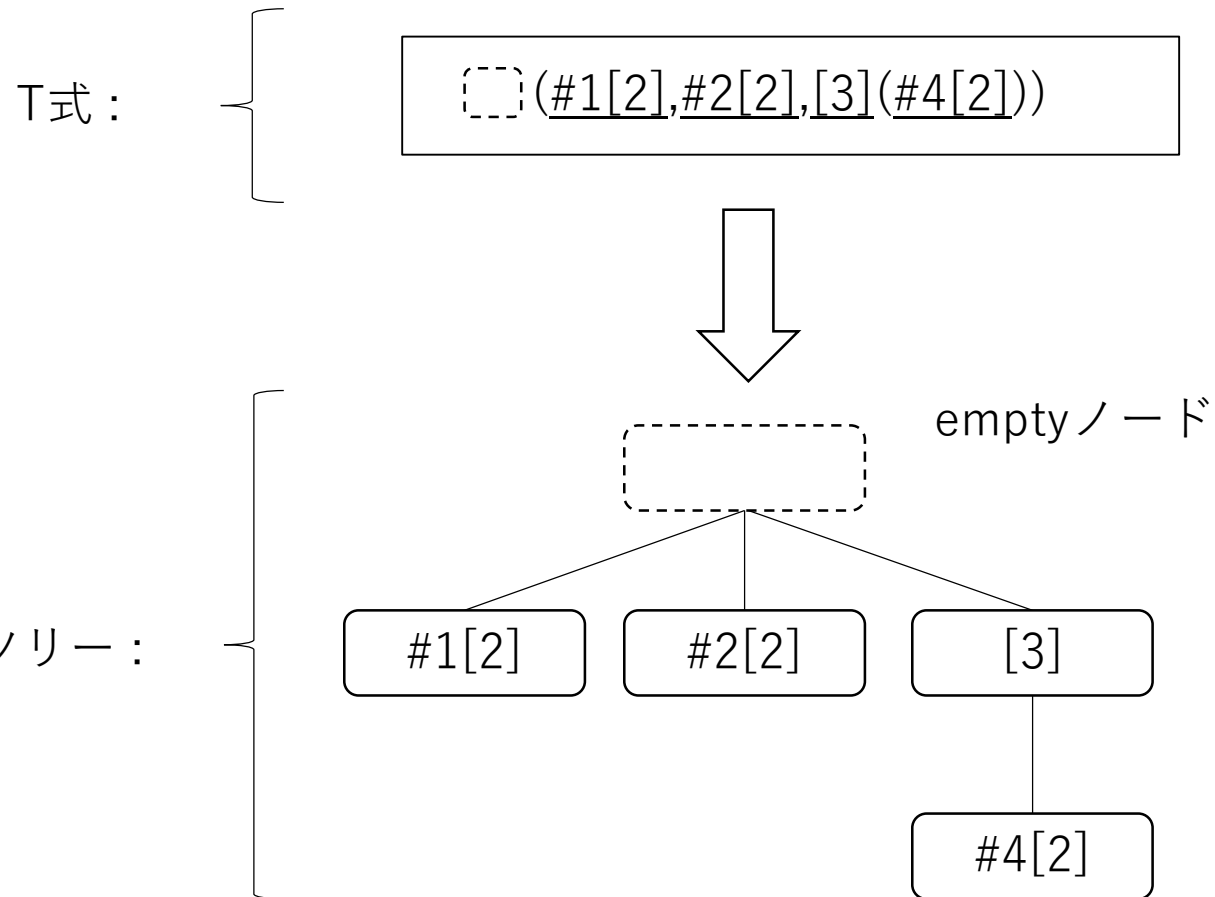
#### 1.3 バインド処理

## 2. 言語仕様検討事項

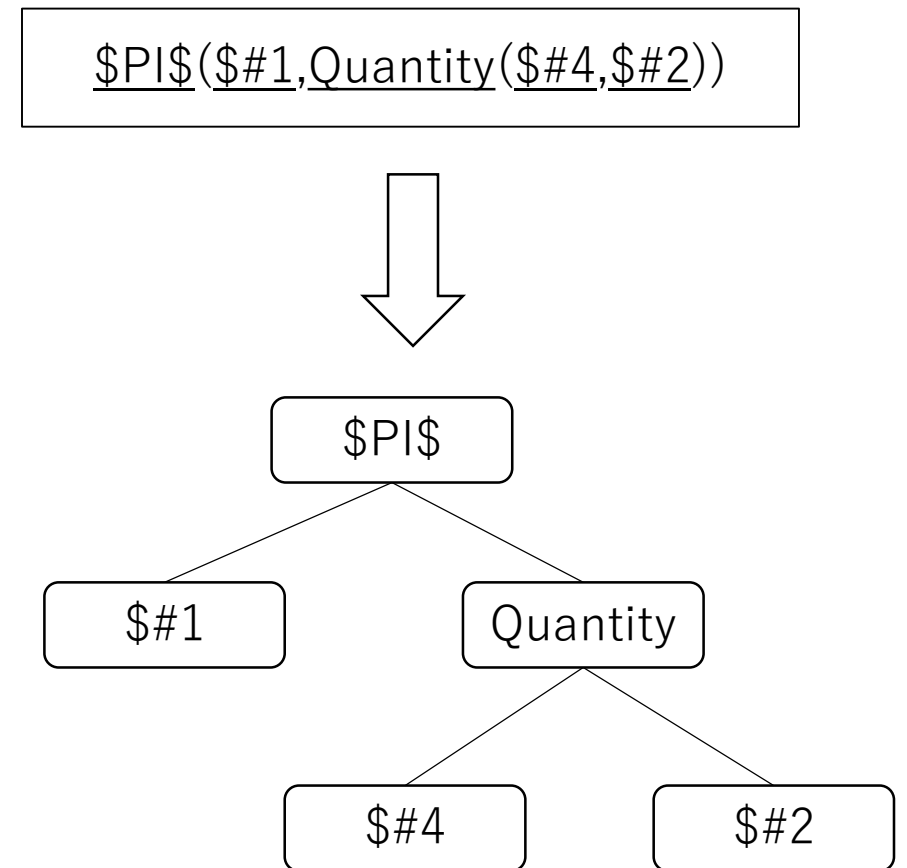
## ◎ パージングとは

- ・ T式 : tq言語による記述
- ・ パージング : T式からツリーを作成

(例1)



(例2)



## ◎ T式の文法(BNF記法)

- (1)  $\langle \text{header} \rangle ::= \langle \text{empty} \rangle \mid \langle \text{name-str} \rangle$   
(2)  $\langle \text{T-exp} \rangle ::= \langle \text{header} \rangle \{ "(" [ \langle \text{T-exp} \rangle \{ "," \langle \text{T-exp} \rangle \} ] ")" \}$

$\langle \text{empty} \rangle ::=$  空文字列

$\langle \text{name-str} \rangle ::=$  デリミタ以外の文字列

(凡例)

“...”: terminal

$\langle \rangle$  : nonterminal

$\{ \}$  : 0回以上

$[ ]$  : 0回または1回

例

- ① A // header
- ② A(B,C(D)) // T-expの入れ子構造
- ③ (A,B,C) // 先頭にempty header
- ④ A(,D) // ,の前に //
- ⑤ 2(A,3) // 数字のみのheaderも可
- ⑥ A(B,C)(D) // Aの後に()の繰り返し

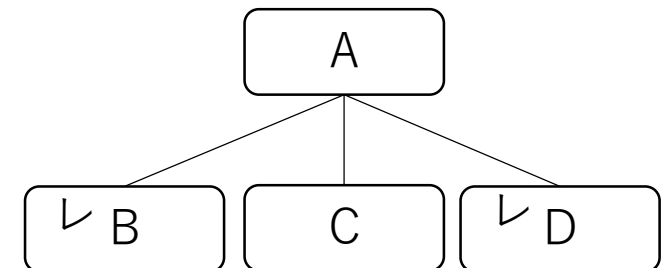
### ※⑥について

fは2変数関数であり、2つのパラメータ値を与えることで1変数関数になるように定義されているとする(例えば、 $f(x, y)(z) = x+y+z$ )。

fをコールする記法は次のようになる。

$f(2,3)(3)$

レ: 先頭子ノード(conjugate flag)

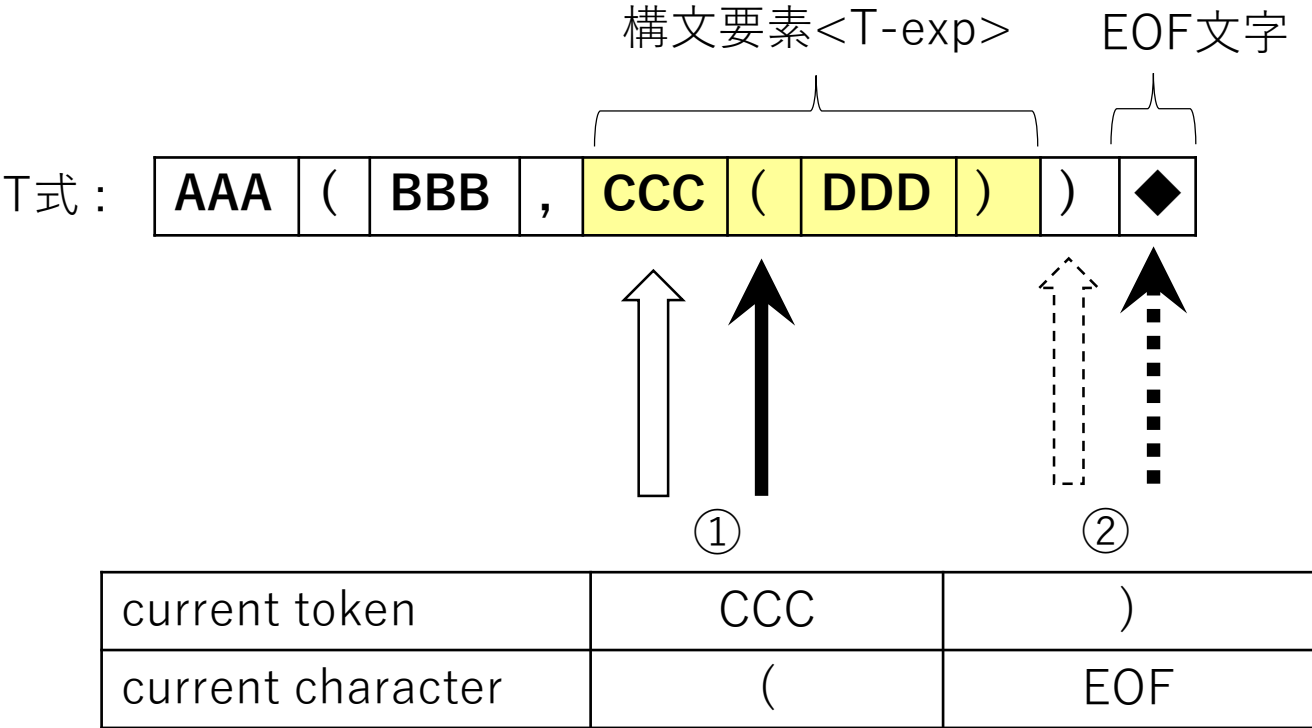


# ◎ パーシング処理

- (1) <header> ::= <empty> | <name-str>  
(2) <T-exp> ::= <header> { "(" [ <T-exp> { "," <T-exp> } ] ")" }

基本: 1 token 1 character look ahead

・ 処理関数との対応	
<header>	⇔ parse_header()
<T-exp>	⇔ parse_T()
token切出	⇔ next_token() skip()



- current token
  - ① 構文要素の解析開始時: 構文要素の先頭
  - ②       "                                  終了時: 構文要素の次
- current char  
current tokenの次の文字

token一覧	
"I"	名前文字列(identifier)
"(	
)"	
" ,	
"E"	EOF

## ◎ parse\_header()

(1) **<header> ::= <empty> | <name-str>**

(2) **<T-exp> ::= <header> { "(" [ <T-exp> { "," <T-exp> } ] ")" }**

```
typedef char    TOKEN;
```

```
char    ch;           // next char of current token
```

```
TOKEN token;         // current token
```

```
char*    BUFF;        // string for current token
```

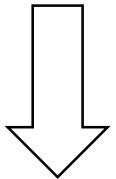
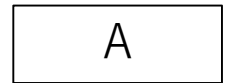
```
// return tree for <header>
```

```
NODE parse_header()
{
    NODE node = alloc_node();    // allocate node for this <header>

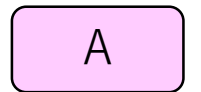
    if(token != 'I') {
        set_head(node, "");
    } else {
        set_head(node, BUFF);    // set buff to <header> node
        skip('I');              // skip identifier token
    }
    return node;
}
```

current token: "I" / "A"

T式:



ツリー:



node

## ◎ skip()

```
void skip(TOKEN tk)
{
    if(tk == token) {
        next_token();    // 次のトークンをカレント
    } else {
        error("syntax error");
    }
}
```

# ◎ parse\_T()

// return tree for <T-exp>

NODE parseT()

{

    NODE root;     // root node of tree for this <T-exp>

    NODE child;    // child node of root

    root = parse\_header();     // node for <header>

    // child nodes

    while(token == '(') {

        skip('(');                     // skip '('

        child = parseT();             // 1st child after '('

        add\_child(root, child);       // add child to root

        set\_conjugate(child, OFF);    // mark as 1st child

        while(token == ',') {

            skip(',');                 // skip ','

            child = parseT();         // 2nd or later child

            add\_child(root, child);    // add child to root

            set\_conjugate(child, ON);    // mark as 2nd or later

        }

        skip(')');                     // skip ')'

    }

    return root;                        // root node of this T-exp

}

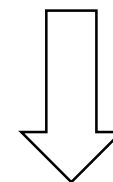
(1) <header> ::= <empty> | <name-str>

(2) <T-exp> ::= <header> { "(" [ <T-exp> { "," <T-exp> } ] ")" }

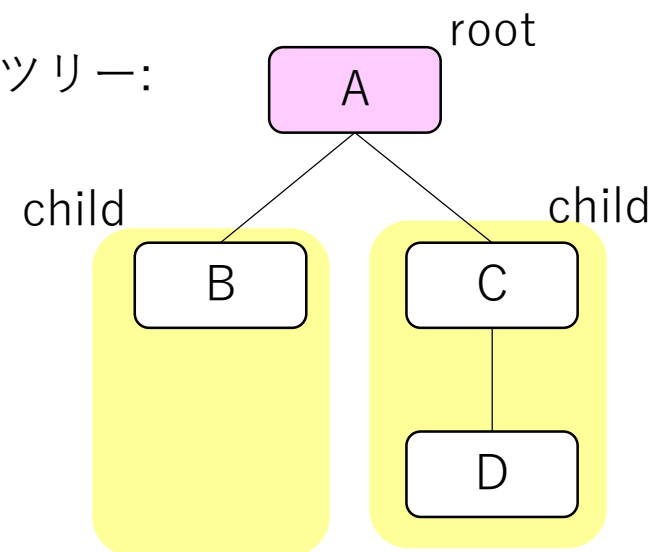
current token: "I" / "A"

T式:

A(B,C(D))



ツリー:





## 1. 改造部分の解説

### 1.1 パージング処理

### **1.2 アクセッサの導入**

### 1.3 バインド処理

## 2. 言語仕様検討事項

# ◎ ツリーの内部構造(1/2)

メモリ削減対応  
=> ツリー表現を変更

ポインタ表現



配列表現

構造体定義

ツリー全体

各ノード

```
typedef int      NODE; // node type
#define NO_NODE -1     // no node

struct LinkTable {
  int offset;
  int node_count; // number of nodes

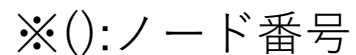
  // int *ser;
  NODE *parent; // parent
  // int *level;
  // int *child_no;
  char **head; // head string
  int *conjugate; // 1st children / 2nd or later
  // char *label_type;
  // int *label;
  int *indicator_pos;
  char **dimension_str;
  int *value_count;
  int **value_poses;
  char **values_str; // bounded values
  int *child_count; // number of childrens
  NODE **children; // children
  NODE *ref_node; // referenced node
  int *extra_stat;
};
```

※ノード0はnull node

ノード番号:  
(配列添字)

T式:

ツリー:



```

(*LT).head)[n]
(*LT).parent)[n]
((*LT).children)[n])[i]
(*LT).conjugate)[n] = ON

```

# ◎ アクセッサ

LT: LinkTableのアドレスを保持

要求	直コーディング	アクセッサ使用
ノードnのhead文字列	((*LT).head)[n]	head(n)
ノードnのparent	((*LT).parent)[n]	parent(n)
ノードnのi番目のchild	(((*LT).children)[n])[i]	child(n, i)
ノードnのconjugateをON	((*LT).conjugate)[n] = ON	set_conjugate(n, ON)

<定義> (C-functions.h内)

```
static struct LinkTable* LT;

// ith child of node n
NODE child(NODE n, int i)
{
    return (LT->children[n])[i];
}

// set conj flag of node n
void set_conjugate(NODE n, int conj)
{
    (LT->conjugate[n]) = conj;
}
```

<使用例>

```
ノードnをルートとするツリーの全ノードについて
head stringを表示する

int ptint_heads(NODE n)
{
    print_string(head(n));           // print self

    for(i=0; i<child count(n); i++) {
        print_heads(child(n, i));
    }
}
```

## 1. 改造部分の解説

### 1.1 パージング処理

### 1.2 アクセッサの導入

### **1.3 バインド処理**

## 2. 言語仕様検討事項

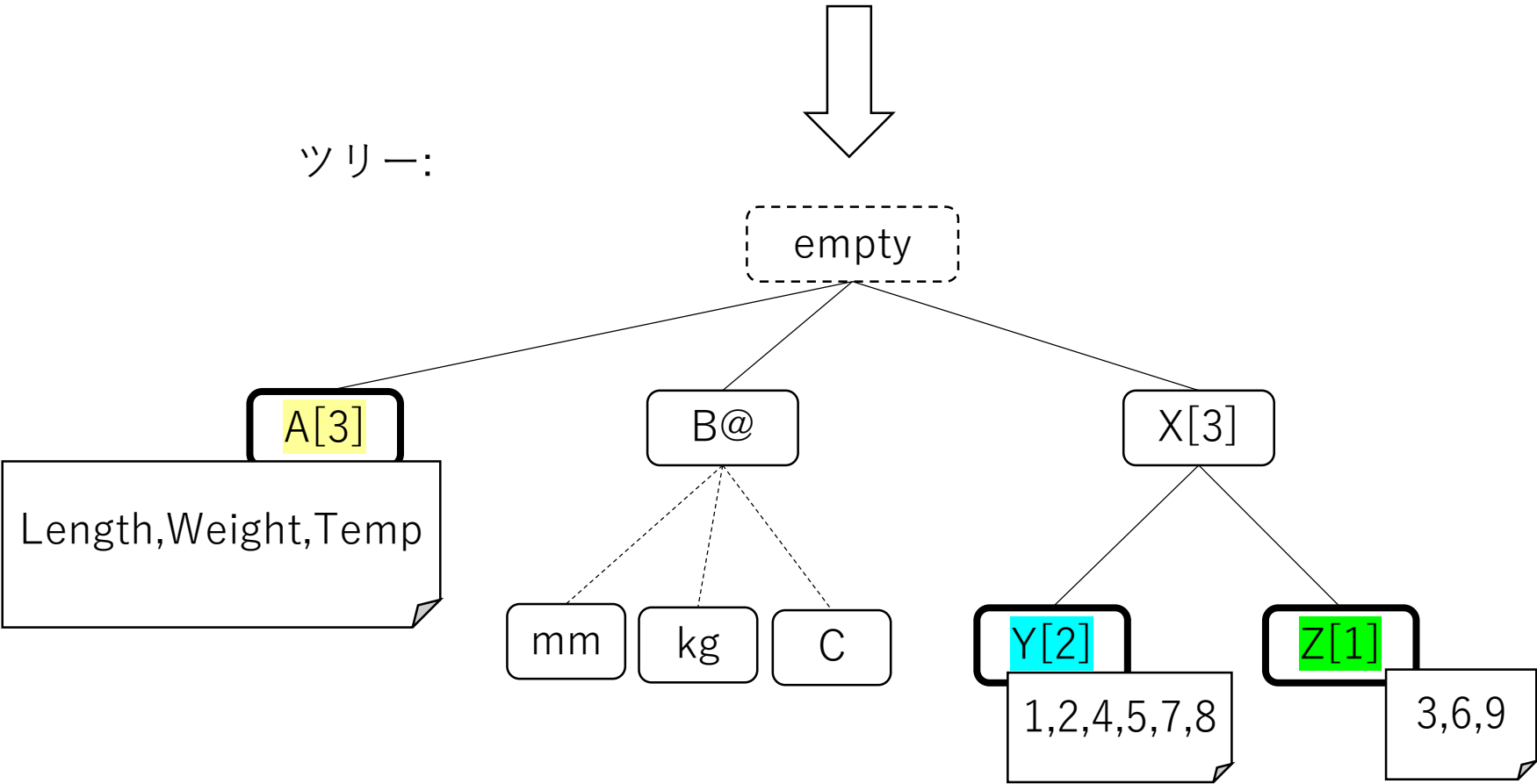
◎ バインド処理とは

in(T式): (A[3], B@(mm,kg,C), X[3](Y[2], Z[1]))

data(CSV): Length,Weight,Temp,1,2,3,4,5,6,7,8,9

※2次元以上や任意回繰返も指定可  
・ X[2,3,5]      3次元  
・ X[]            任意回繰返  
                  (現状1か所のみ可)



ツリー:



<値の割当順序>

ノード		値
X[3] (1回目)	Y[2]	1,2
	Z[1]	3
X[3] (2回目)	Y[2]	4,5
	Z[1]	6
X[3] (3回目)	Y[2]	7,8
	Z[1]	9

※ノード間リンク2種類  
・ \_\_\_\_\_ : 親子関係  
・ ..... : 属性関係?

 : [~]付leaf(CSVデータのバインド対象)       : ノードにバインドされた値

## ◎ バインド処理のロジック

```
void bind_node(Node node, Stream* in)           // in: CSVファイル
{
    int num = array_size(node);                 // ノードnodeの配列サイズ、配列指定なしは-1;
                                                // (例) A[3] -> 3、B -> -1
    if(is_leaf(node) && num == -1) {
        return;                                // leafであって配列指定なしの場合読み込みなし
    } else {
        for(int i=0; i<num; i++) {              // 配列サイズ分繰返す
            bind_primitive_node(node, in);
        }
    }
}
```

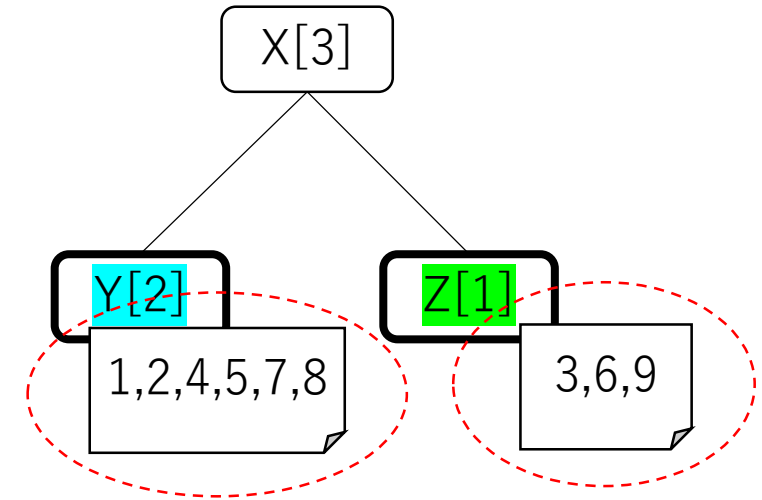
```
void bind_primitive_node(Node node, Stream* in) // in: CSVファイル
{
    if(is_leaf(node)) {
        bind(node, in, dim);                    // inからdim個のデータを読み込んでnodeにバインド
    } else {
        for(int i=0; i<child_count(node); i++) {
            bind_node(child(node, i), in);
        }
    }
}
```

※2次元以上や任意回繰返の処理は省略

## ◎ バインドの2パス処理

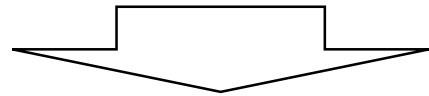
data(CSV): Length,Weight,Temp,1,2,3,4,5,6,7,8,9

値保持用のメモリ領域



(課題)メモリ領域のサイズが処理終了まで決定できない

- ・最初に大きめな領域確保 => メモリ量が無駄
- ・CSV値読み込みごとにメモリ領域割り当て => 性能上インパクト大



2パス方式

- ・1パス目: CSV値を読み込んで必要なメモリサイズを決定し、領域確保
- ・2パス目: CSV値を読み込んで、確保した領域に順次格納



## 1. 改造部分の解説

### 1.1 パージング処理

### 1.2 アクセッサの導入

### 1.3 バインド処理

## 2. 言語仕様検討事項