プログラミング言語

名倉 正剛
日本大学 工学部 情報工学科
70号館7044号室

授業の日程　※変更する可能性あり

9/17	ガイダンス・復習
9/24	プログラム言語の歴史
10/1	構文，意味，BNF
10/8	識別子，変数
10/15	ステートメント，文，式
10/22	スコープ
10/29	手続き，制御構造
11/5	中間試験

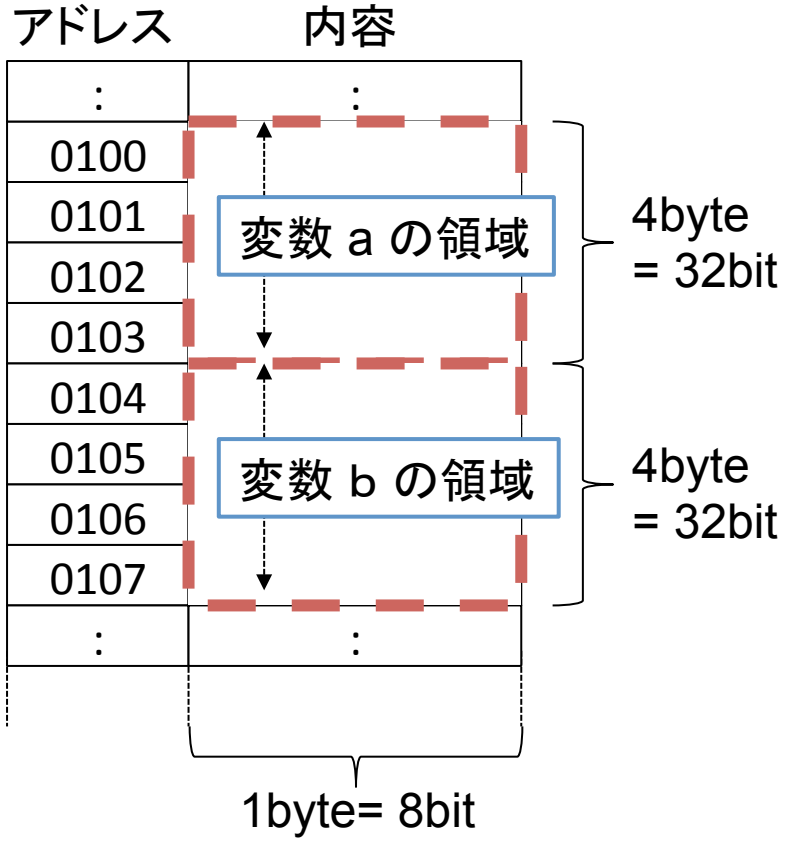
11/12	モジュール
11/19	データ抽象化
11/26	（月曜授業日）
12/3	（土曜授業日）
12/10	例外処理
12/17	並行，排他制御
12/24	関数型言語，OO言語，言語の分類
1/14	まとめ
1/21	期末試験

プログラミング言語　第5回

- ステートメント，文，式
  - と，シラバスに書いていますが，ステートメント＝文 ですので，今日は 文 と 式ということになります．
- その前に，前回の残りから．
  - 変数の部分
- 最後に，研究内容の紹介をさせてください．

復習：変数の概念

- 32bit 処理系で2変数を定義した場合 `int a, b;`
  - メモリ上のどこかの領域に，32bit 分の領域を2つ確保.
  - 確保した領域を，それぞれ a, b という名前で参照できるようにする.
  - 参照した領域に，int (32bit 整数) の値を入出力できるようにする.



復習：変数の4要素

- 変数を定義した場合，次の4要素を定義したことになる．
  - 名前：変数を示す識別子（a とか，b とか）．
  - 属性：参照で示される場所に記憶されている内容(0, 1 のビット列) を，我々の世界の概念 (整数型とか) にどうマッピングするか，どういう演算の種類が許されるのかを規定するもの．
  - 参照：記憶装置上の先頭のアドレス
  - 値： 上記の属性で示された範囲内の値

- たとえば，int a = 100 の場合：

  名前："a"，属性：32ビット整数型，参照：0x0100，値：100

復習：変数と型

- 一般的な言語では，変数はその宣言時に，型が規定される．
  - 例：C言語
  - （最適化の存在を無視すれば）宣言された時点で，記憶装置上に，型に応じた領域を確保される，と説明できる．
- 言語によっては，型を規定しない場合もある．
  - 例：Perl 言語や，BASIC では，明示的な変数宣言を必要としない．
  - 変数に最初に代入され時点で，代入される値に応じて，型が決定される（型推論という）
    - 整数が代入されれば，整数型
    - 小数が代入されれば，浮動点小数型

変数と有限性

- 普通の人間は，有限なものしか認識できない（はず）
  - コンピュータで扱える情報も，有限なもののみ．

- 無限に桁が存在する数値を表現できない．
  - Q1：循環小数はどうなる？
  - Q2：無理数はどうなる？？
  - Q3：非常に大きい数はどうなる？？？

Q1：循環小数はどうなる？

- そもそも，循環小数ってなに？
  - ある桁から先で，同じ数字の列が無限に繰り返される小数
    - 例：　1/3 = 0.33333…（3の繰り返し）
      1/7=0.142857142857…（142857 の繰り返し）
- どのように表現されるか？
  - 当然，分数では表現されない．
  - データのビット数に合わせて表現できるうちで一番近い浮動小数点に近似して表される．
  - 浮動小数点はあとで．

Q2：無理数はどうなる？？

- そもそも，無理数ってなに？
  - 有理数ではない実数．分子・分母ともに，整数であるような分数で表せない実数．
  - 小数部分が循環しない無限小数．
    - 例：　$\sqrt{2} = 1.41413562373095\ldots$

      $\pi = 3.141592653589793\ldots$

- どのように表現されるか？
  - データのビット数に合わせて表現できるうちで．．（以下略）．

Q3 : 非常に大きい数はどうなる？？？

- そもそも，非常に大きい数ってどのくらい？
  - 言語処理系によっては，64 ビット整数型が用意されているものもある．
    - long long int 型 (C99 仕様)
    - 符号なしならば，18446744073709551616 まで表現可能 (1844京6744兆737億955万1616) まで表現可能
      - ちなみに，全世界の預金額は3000兆円
      - アメリカの国家予算は，209兆5250億円
      - 日本の国家予算は，172兆1250億円
      - 全世界の国家予算ならば足りそう？
- それより大きい数を表したいとき，どのように表現されるか？
  - データのビット数に合わせ．．（以下略）．．．．

浮動小数点数型による表現

- 循環小数や，無理数や，大きな数を表現したいとき，浮動小数点数型で近似して表す．
  - 言語によっては，用意されていないものもある．
  - C 言語，Java 言語：float (32bit 単精度)，double (64bit 倍精度)
  - BASIC：Double (64bit 倍精度)
  - 単精度? 倍精度??
- 丸め誤差が生じる

浮動小数点数型

- コンピュータ内で，整数以外の有限桁の数を表現するための形式
- 物理や化学で，非常に大きな数や，非常に小さな数を表現する場合と同じ方法.

  例）1mol は，$6.022140857 \times 10^{23}$

- コンピュータ内部では，$A \times 2^B$ で表現する.
  - A を仮数部，B を指数部という.
  - 単精度の場合，仮数部と指数部の合計が 32bit
    - 符号部 1 ビット・指数部 8 ビット・仮数部 23 ビット
  - 倍精度の場合，仮数部と指数部の合計が 64bit
    - 符号部 1 ビット・指数部 11 ビット・仮数部 52 ビット
  - いずれにせよ，仮数部の桁数分の精度しか表現できない ⇒ 丸め誤差が発生

丸め誤差

- 数値を，どこかの桁で端数処理した場合に生じる誤差のこと.

- 整数型の場合
  - int 型の変数に，小数を代入した時に，小数点以下の桁のデータが切り捨てられる.
- 浮動小数点数型の場合
  - 仮数部で表すことのできる桁数分だけの桁を超えてしまう部分のデータが切り捨てられる.

浮動小数点での丸め誤差

- コンピュータ内では2進数表現：思わぬ部分で丸め誤差が影響する場合あり.

- 0.1 (10進数) を $A \times 2^B$ で表現するために，2 進数に変換すると，
  - 0 . 0001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 1001 100
    (1001 が循環，有効数字 52 桁)
- これを，再度 10 進数で表現すると，
  - 0.09999999999999964472863211994990706443786621093750
  - 0.1 にならない.
- 演算結果に誤差が生じることになる.

浮動小数点での丸め誤差を生じるプログラム例

- C 言語で次のようなプログラムを記述する

```
#include<stdio.h>

int main(void){
  double d = 0.1 + 0.2;
  if (d == 0.3) printf("Variable d is 0.3.\n");
  else printf("Variable d is not 0.3.\n");
}
```

- 実行結果は，どちらになるか?

A)

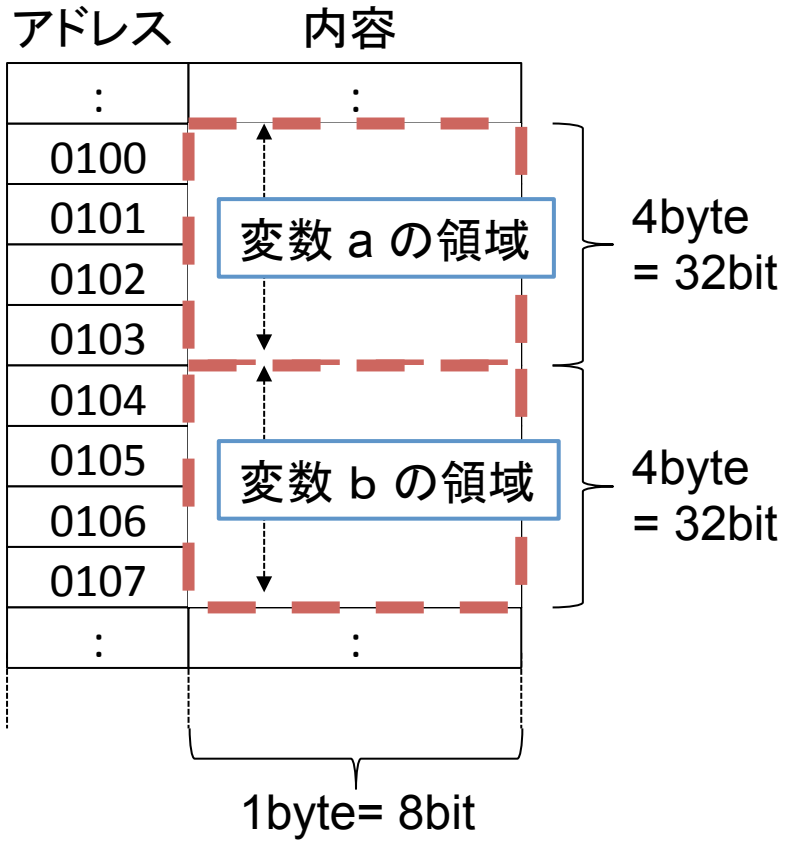
```
Variable d is 0.3.
```

B)

```
Variable d is not 0.3.
```

変数に対する参照方法

- 変数を読み出す場合，次の2通りの参照方法がある．
  - 値を直接参照
  - 変数領域のアドレスから領域内の変数を参照
- 値を代入していると思って代入先を書き換えた場合に，同一の領域を書き換える場合あり．



アドレスの参照

- C 言語で次のようなプログラムを記述する

```
#include<stdio.h>

int main(void){
  int a =1;
  int *b = &a;
  a = 50;
  *b = 100;
  printf(" Variable a is %d\n", a);
}
```

- 実行結果は，どうなるか?

ポインタとアドレス

- C 言語のポインタの記法は，変数のアドレスを自由に設定するための仕掛け．
  - & を変数名の前に付加することで，アドレス参照
  - * を設定した変数では，変数の参照先のアドレス (これをポインタという) を自由に変更できる．

  ⇒ アドレスを代入すると，参照を代入することになるため，同じ領域を参照する．

- Java ではポインタがない．
  - が，しかし，オブジェクトの参照が同じ意味を持つ．

Java の場合

- Java 言語で次のようなプログラムを記述する

```
public class Main {
  public static void main(String args[]){
    Test t1 = new Test();
    Test t2 = t1;
    t1.str = "String1";
    t2.str = "String2";
    System.out.println("t1 is " + t1.str);
    System.out.println("t2 is " + t2.str);
  }
}
```

```
public class Test {
  public String str;
}
```

- 実行結果は，どうなるか?

ヒント

- 第2回のスライド（後半部分に注意）

Java 言語

- オブジェクト指向プログラミングの考え方に基づいて設計された言語(1991，米サン・マイクロシステムズ *James Gosling*)
  - 完全にクラスベースなオブジェクト指向プログラミング
  - 高水準言語（C など）では，機種間でのプログラムの互換性はあったが，機械語プログラムに変換すると互換性がなかった
    → 機種間でバイナリ互換性を保つために，仮想マシン（VM）上で動作する．
- C/C++ の文法から多くを引き継いでいる．
  - 文法的には，ポインタはない．
  - しかし，オブジェクトインスタンスの代入は，参照により行われる．

```
List list1 = new ArrayList();
list1.add("A");
list1.add("B");
List list2 = list1;
list2.clear();
if (list1.isEmpty())
  System.out.println("empty");
```

真になる
（empty が表示）

文(Statement)と式(Expression)

- プログラミング言語の構文の要素:
  文と式に分類

- 文:単独で完結する言語要素.
  (例:代入文, if 文, for 文, while 文, ...)

- 式:単独では完結せず, 文や他の式の一部として利用される言語要素.
  (例:条件式, 演算式, . . .)

式

- 値を算出する目的で記述

- 構成要素
  - 定数
  - 変数
  - 演算子
  - 関数呼び出し

- 式の評価は，値の代入や，演算子の処理により実施される.
- 式の実行前後で変数の値が変わる場合も，変わら無い場合もある.

参照透過性 (Referential transparency)

- 式の実行前後で変数の値を変更しない場合の性質
- 文脈によらず式の値が構成要素によってのみ定まる
  - 当然，代入文は参照透過性を満たさない

- 参照透過性を満たす式は，プログラム内のどこで実行しても，何回実行しても同じ処理を実施可能.
  - 同じ処理演算結果を得ることができる.
- 参照透過性を満たす関数は，同じ引数を与えた場合に実行のたびに同じ値を返却し，関数外の変数の状態を変更しない.
  - 関数内の変数は実行終了とともに消えるので，どのように変更されたとしても，開始時と終了時では状態が変わらないことに注意

参照透過性を満たさない場合

- 式の場合
  - 前後の式との実行順序に依存する.
- 関数の場合
  - 同じ入力を与えた場合でも，実行回数により異なる結果が得られる.
  - 関数外の変数の状態を変更する場合があるから，他の関数の実行に影響を与える.

演算子の評価順序

- 演算子の評価順序は，言語によって規定されている.
  数学，工学，科学全般での優先順位と共通.
  1. かっこ内の項
  2. べき乗，累乗根
  3. 乗法，除法
  4. 加法，減法

- 式の評価は，左から右に行う.

A+B+C*D？

- 例えば，A+B+C*D はどうなるか？
  - 方法1）まず，優先度の高い演算子から評価し，優先度に差がない状態になったら，左から評価.
    1. C*D を評価（E とおく）
    2. A+B+E を評価
  - 方法2）左右に演算子が存在するオペランドについて，演算子の優先度に差異のないオペランドまでを部分評価する．そのあとで優先度の高い演算子から評価する.
    1. B は差異がないが，C は左右で差異があるため，A+B を評価（F とおく）
    2. C*D を評価（E とおく）
    3. F+E を評価

C 言語での優先順位（その1）

1	() [] -> . ++ --	関数呼び出し演算子，配列要素，メンバへのアクセスなどの後置演算子
2	! ~ - + * & sizeof *type cast* ++ --	前置の単項演算子など
3	* / %	乗法，除法，剰余
4	+ -	加法，減法
5	<< >>	ビット単位のシフト
6	< <= > >=	大小比較
7	== !=	等価/非等価比較
8	&	ビット単位のAND (例：a & b)
9	^	ビット単位の排他的 OR (例：a ^ b)
10	|	ビット単位の通常 OR (例：a | b)

C 言語での優先順位（その2）

11	&&	論理 AND
12	||	論理 OR
13	?: = += -= *= /= %= &= |= ^= <<= >>=	条件式と代入
14	,	コンマ演算子 ※下を参照

- コンマ演算子
  - 演算子の左オペランドを評価し，その値を捨て，そのあとで右オペランドを評価する．

例：

```
for (i = 0, j = 0; i + j < 100; i++, j++){
        …..
}
```

演習問題

- 左右の式のうち，評価が異なるものは？

1	A + B * C	(A + B) * C
2	!A + !B	(!A) + (!B)
3	!(A && B)	!A && !B　（ヒント:ド・モルガンの法則）
4	++B + A	A + (B++)
5	A || B && C	A || (B && C)
6	A && B == C	(A && B) == C
7	A == B || C	(A == B) || C
8	A += B + C	A += (B + C)
9	A = (B, C)	A = B, C
10	A << B + C	A << (B + C)

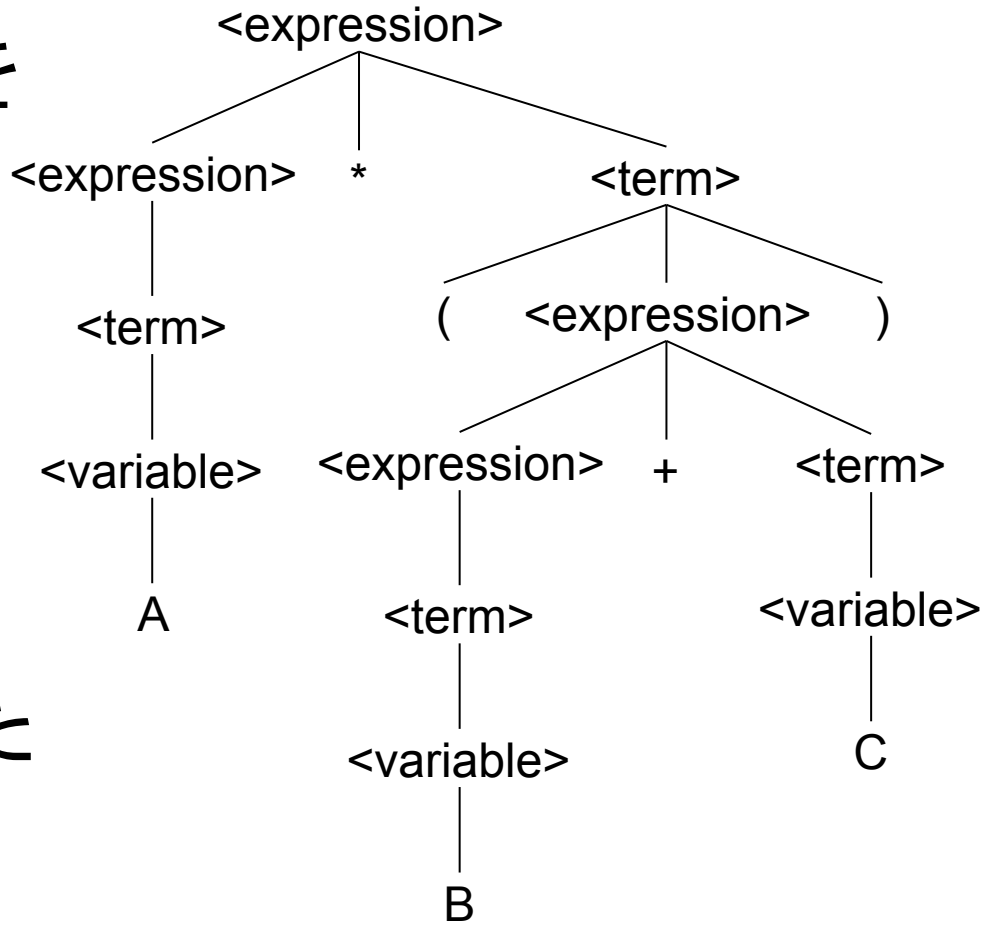
式評価の BNF 表現

- A*(B+C) を受理できるBNF

```
<expression> ::= <term> | <expression> ‘+’ <term>
                                | <expression>  ‘*’ <term>
<term> ::= <variable> | ( <expression> )
<variable> ::= 'A' | 'B' | 'C’
```

構文木

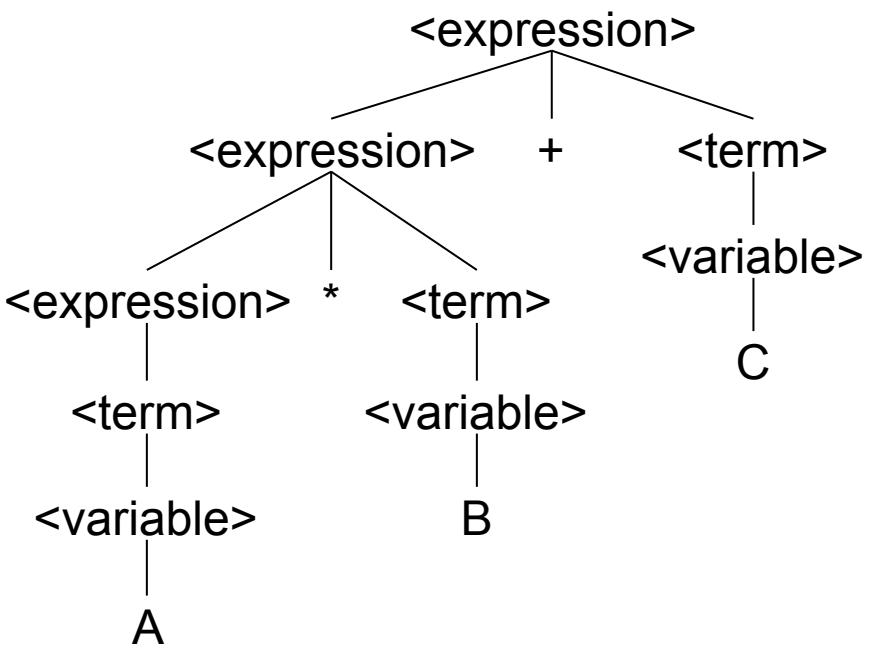
- 特定の言語定義を使って実際の言語を解析した結果を表すための木構造図



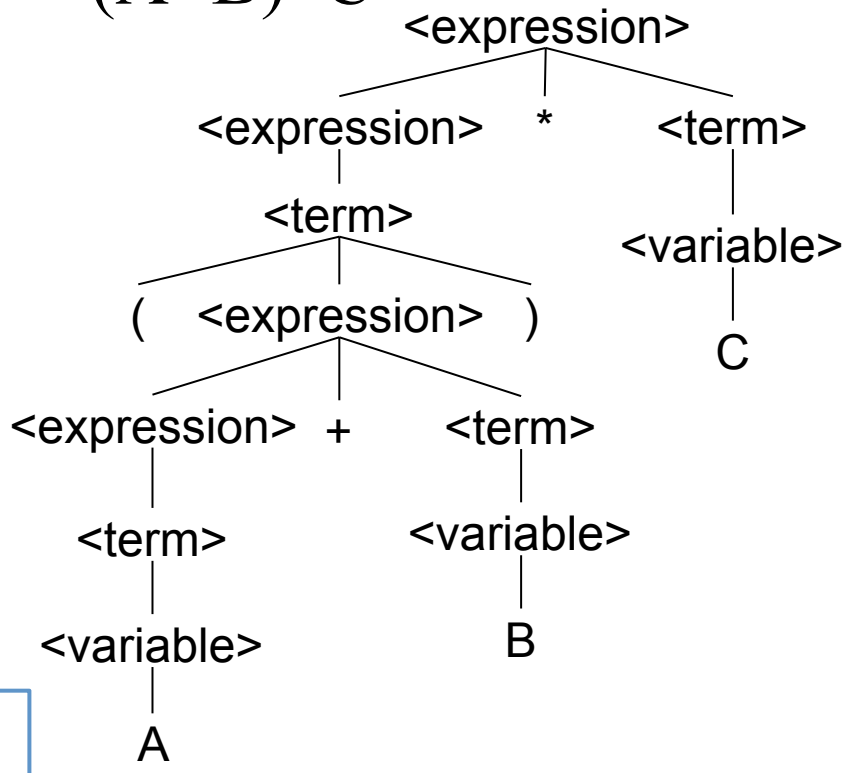
- 前の BNF で A*(B+C) を解析した場合:

構文木と演算順序

- 解析対象により，構文木の形状は異なる．
  - A*B+C



  - (A+B)*C



この BNF では，
<expression> '+' <term>
<expression> '*' <term>
の順序で表さないと受理しないことに注意

曖昧な文法

- 次の文法定義があったとする

```
<expression> ::= <variable> | <expression> ‘+’ <expression>
                             | <expression>  ‘*’ <expression>
<expression> ::= <expression> | ( <expression> )
<variable> ::= 'A' | 'B' | 'C’
```

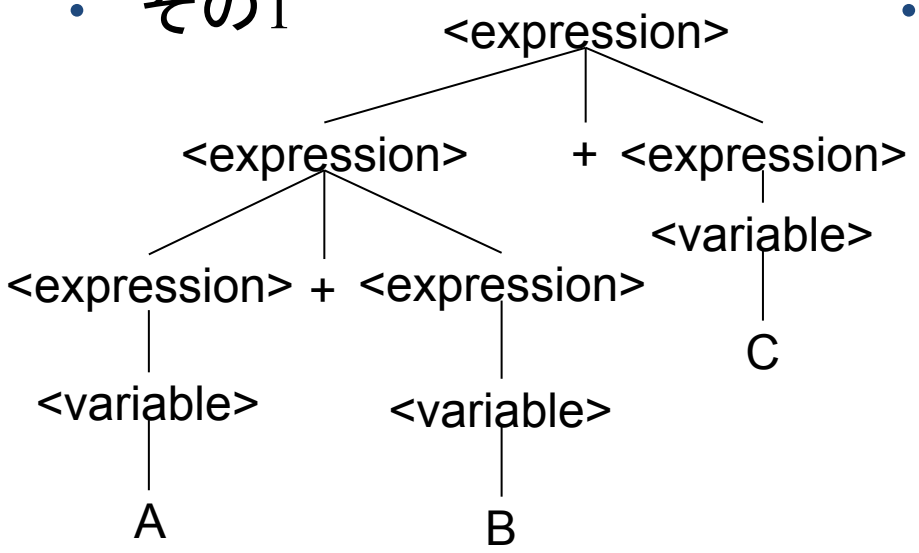
- 上の BNF で A+B+C を受理することは「曖昧」さを含んでいる.
- 先ほどの BNF（下記）と比較してみてください

```
<expression> ::= <term> | <expression> ‘+’ <term>
                        | <expression>  ‘*’ <term>
<term> ::= <variable> | ( <expression> )
<variable> ::= 'A' | 'B' | 'C’
```

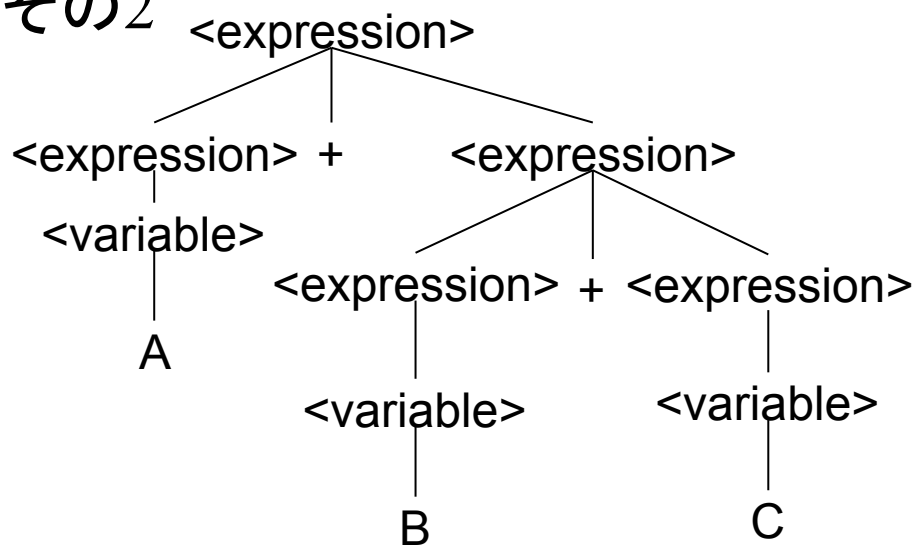
構文の曖昧さ

- 同じA+B+Cの受理に対して，構文木が複数かける.
  - 同じ言語の受理に対して，解析処理が定まらない→ 曖昧な文法
    → 構文解析の効率が大幅に低下する

- その1



- その2



<expression> ::= <variable> | <expression> ‘+’ <expression> | <expression> ‘*’ <expression>
<expression> ::= <expression> | (<expression>)
<variable> ::= 'A' | 'B' | 'C’

文の構造の種類

- 分岐構造
  - IF-THEN-ELSE 構造
  - CASE-WHEN（SWITCH-CASE) 構造
- 繰り返し（反復）構造
  - WHILE-DO-END 構造
  - DO-WHILE (REPEAT-UNTIL) 構造
  - LOOP-REPEAT 構造
- GOTO 文とラベル

分岐構造

※ Ada の場合

IF-THEN-ELSE 構造

```
if C1 then S1
elsif C2 then S2
elsif C3 then S3
……
elsif Cn then Sn
else Sn+1
endif;
```

- C 言語では，一部をブロック構造で表現.
- elsif を示す専用キーワードが存在する言語もある.

CASE-WHEN 構造

```
case V is
  when C1 S1
  when C2 S2
……
  when Cn Sn
  when others Sn+1
end case;
```

- C 言語では，一部をブロック構造で表現.
  - switch-case 構造

繰り返し（反復）構造

※ Pascal の場合

- WHILE-DO-END 構造

```
while COND do
  STATEMENT
end
```

- REPEAT-UNTIL 構造

```
repeat
  STATEMENT
until COND
```

- それぞれ，前判定，後判定
  - C 言語だと，後判定は DO-WHILE.

- LOOP-REPEAT 構造

```
loop
  if not COND then exit
  STATEMENT
repeat
```

```
loop
  STATEMENT
  if COND then exit
repeat
```

- 処理実行前に，条件を満たさない場合に終了 or処理実行後に，条件を満たす場合に終了.
- C 言語だと，無限ループと if + break で書く.

GOTO 文とラベル

- ラベルを定義して，goto 文で自由にジャンプできる
  （右はC言語の場合．
  "a\n"は表示されない）

```
int main(void){
        goto test;
        printf("a\n");
    test:
        printf("b\n");
        return 0;
}
```

- 前ページまでにあげた
  制御構造内から，自由
  に外に飛ぶことが可能
- 逆に，制御構造内に入ることも可能．

→ 便利ではあるが，制御構造を崩してしまう．
（スパゲッティプログラム）