ソフトウェアアーキテクチャ
第5回　コンパイラ

環境情報学部
萩野　達也

スライドURL

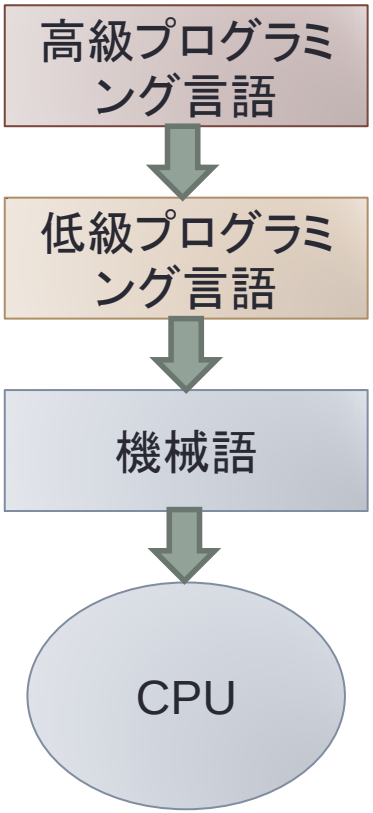
https://vu5.sfc.keio.ac.jp/slide/

プログラミング言語

- プログラミング言語
  - コンピュータに処理を伝えるために作られた人工的な言語

- 低級言語（low-level programming language）
  - コンピュータが直接実行できる機械語と対応
  - アセンブリ言語
  - CPUの種類ごとに異なる

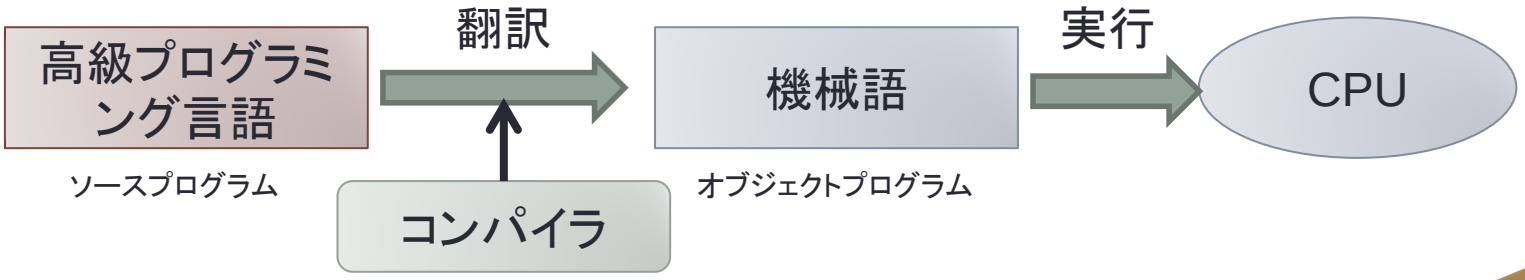
- 高級言語（high-level programming language）
  - CPUやアーキテクチャに非依存
  - 人が理解しやすい言語
  - コンピュータが直接実行することはできない
  - 低級言語と対応を取る必要がある



プログラミング言語の実行方法

翻訳

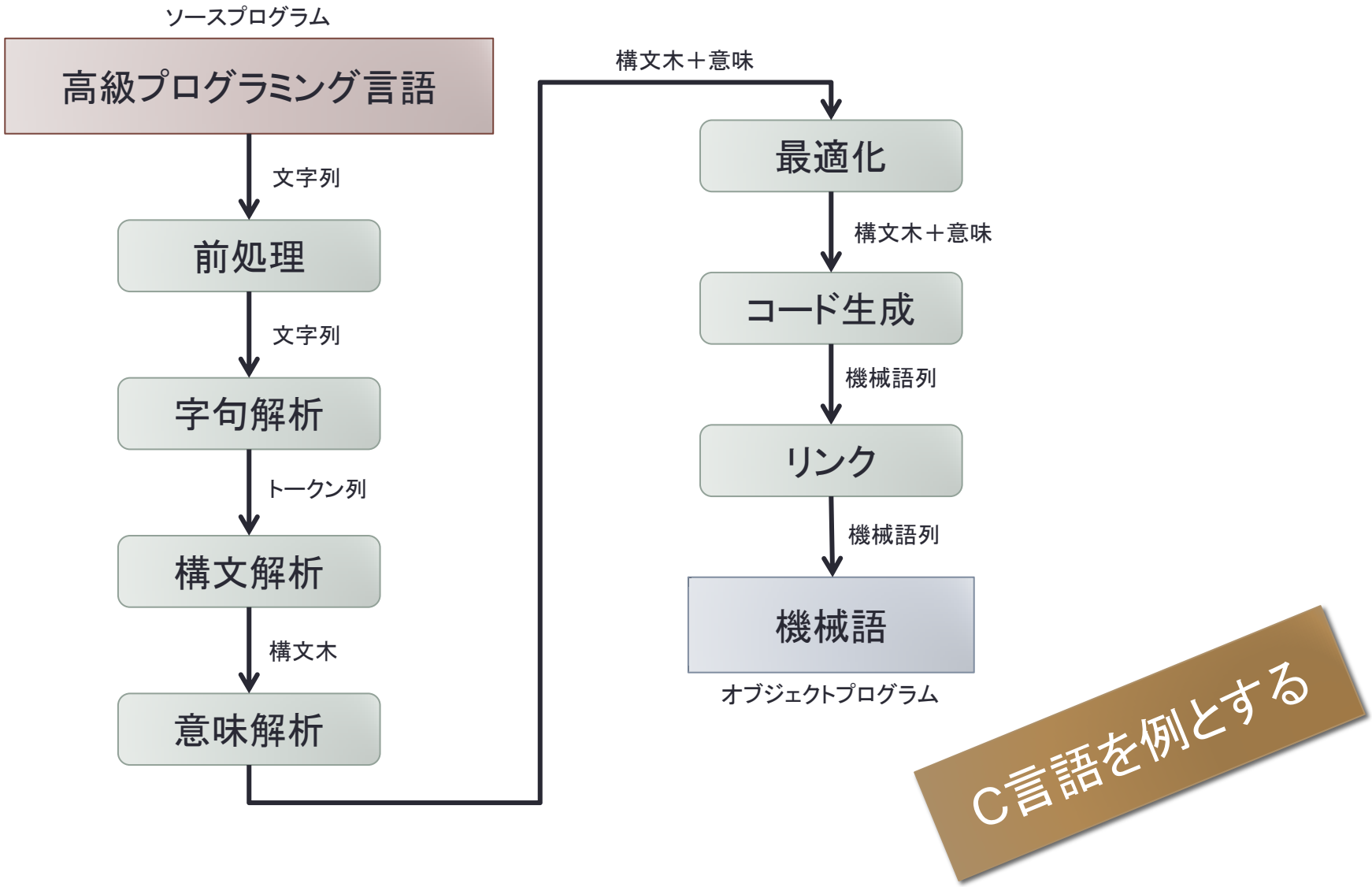
- コンパイラ方式（compiler）
  - プログラムを機械語に変換して実行
  - プログラムを機械語に変換（翻訳）するプログラムをコンパイラ



通訳

- インタープリタ方式（interpreter）
  - プログラムを変換や翻訳せずに，そのまま直接実行
  - コンパイラに比べて実行速度は落ちる
  - プログラムを変更してから実行までは早い

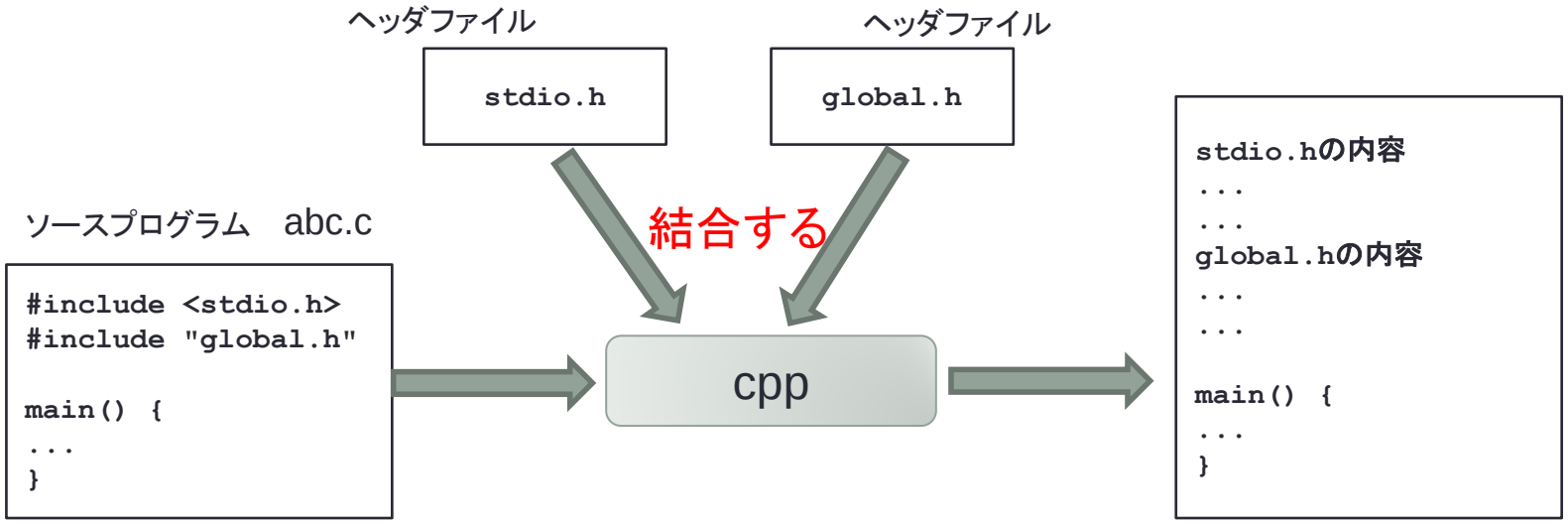
コンパイラの構成


ソースプログラム
高級プログラミング言語
文字列
前処理
文字列
字句解析
トークン列
構文解析
構文木
意味解析
構文木＋意味
最適化
構文木＋意味
コード生成
機械語列
リンク
機械語列
機械語
オブジェクトプログラム
C言語を例とする

前処理(プリプロセス)

- C言語コンパイラのプリプロセッサ
  - cpp（C PreProcessor）
  - #で始まる行がプリプロセッサへの指示

- 役割
  - ヘッダなどの他のファイルを読み込みソースプログラムを一つにする.
  - マクロの定義に従いソース中のマクロの展開を行う.
  - 条件に従いソースの一部分の選択的コンパイルを可能にする.



cppのマクロ処理

- 定数に名前を付ける
  - **#define MAX 100**

```
for (i = 0; i < MAX; i++) {
  a[i] = b[MAX - i - 1];
}
```


```
for (i = 0; i < 100; i++) {
  a[i] = b[100 - i - 1];
}
```

- プログラムが読みやすくなる
- 変更が簡単

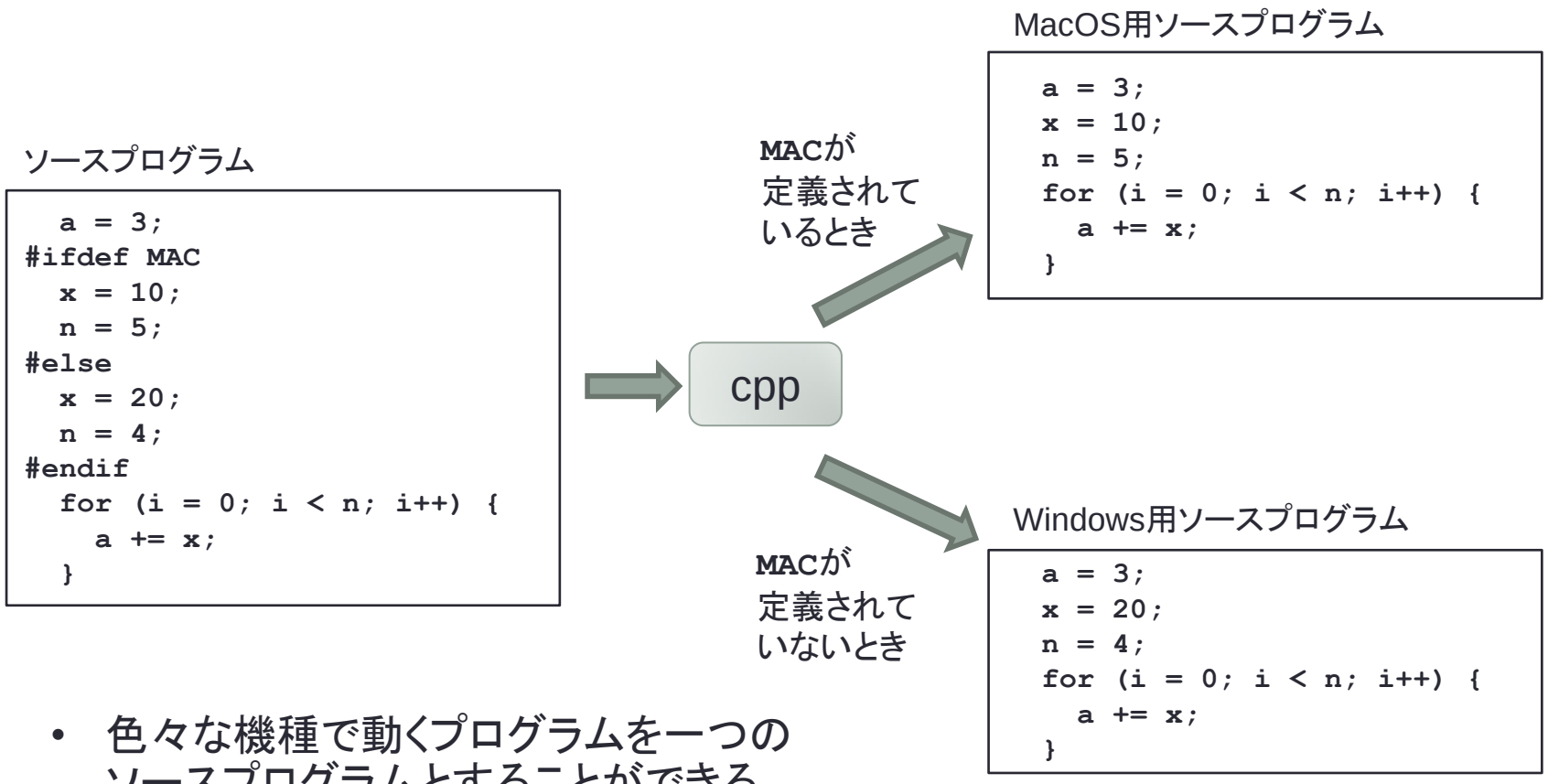
- 引数を取って，マクロ展開の時に引数を使うことができる
  - 関数呼び出しではなく，単純な文字列置換であることに注意
  - **#define DOUBLE(x) (x)*(x)**

```
a = DOUBLE(5);
b = DOUBLE(a+1) + DOUBLE(c);
c = DOUBLE(DOUBLE(a+b));
```


```
a = (5)*(5);
b = (a+1)*(a+1) + (c)*(c);
c = ((a+b)*(a+b))*((a+b)*(a+b));
```

条件付きコンパイル

- 条件にしたがってコンパイルするコードを選択する



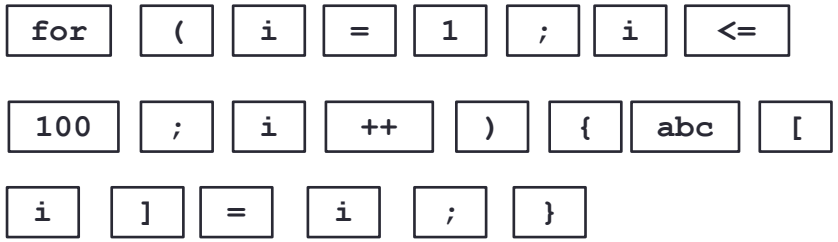
- 色々な機種で動くプログラムを一つの
  ソースプログラムとすることができる

字句解析

- 字句解析（lexical analysis）
  - 文字列を意味のある最小単位にまず分ける
  - 変数名や関数名，数字，記号などの単位に分ける処理

```
for (i = 1; i <= 100; i++) {
  abc[i] = i;
}
```

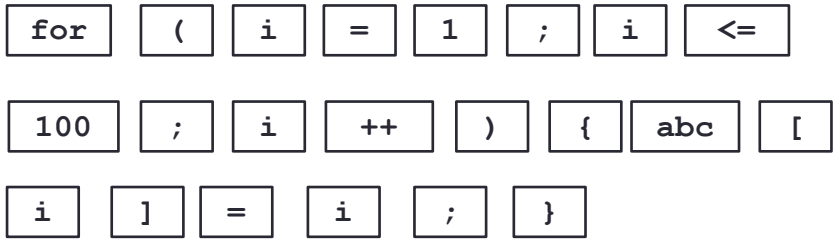
字句解析



- 字句
  - 数字：数字の列
  - 名前：英文字で始まる英数字文字列
  - 記号：1文字あるいは2文字の記号列
- 字句解析では字句の種類と値に分けられる

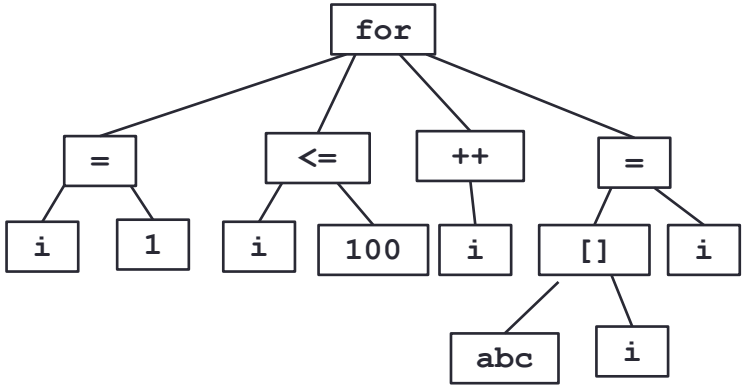
構文解析

- 構文解析（syntax analysis）
  - 意味となるかたまりに分ける
  - 式や文のかたまりに分ける
  - 字句の列から構文木を作る



構文解析

構文木
(parse tree)



- 構文
  - 正しい文や式として解釈するための規則
  - 文法規則で書かれることが多い
  - 文脈自由文法（context free grammar）
  - 構文解析は文法に基づいて解析を行う

構文規則

- for文の構文規則

- 構文規則は文脈自由文法として与えられることが多い
  - BNF（Bacus Naur Form）による定義

```
<for文> ::= 'for' '(' <式> ';' <式> ';' <式> ')' <文>
```

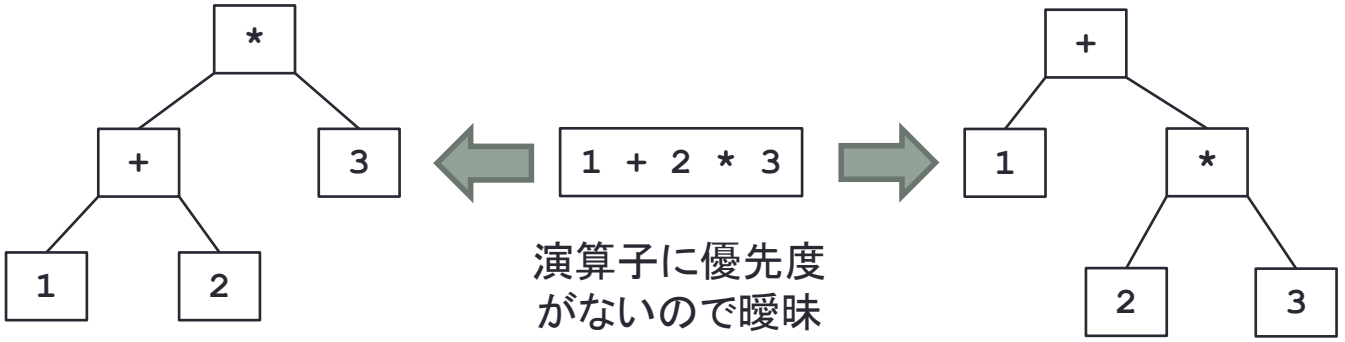
- C言語の構文規則は大きく5つに分けられる
  - 式
  - 文
  - 関数
  - 変数宣言
  - 型宣言

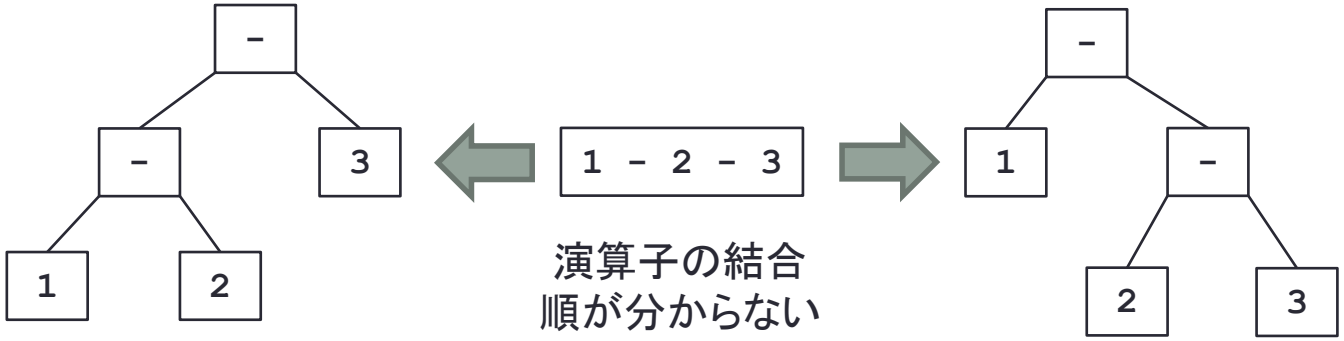
英語の5文型
- S V
- S V C
- S V O
- S V O O
- S V O C

式の構文規則

- 四則演算の構文規則

```
<式> ::= <数> | <式> '+' <式> | <式> '-' <式> | <式> '*' <式> | <式> '/' <式>
```

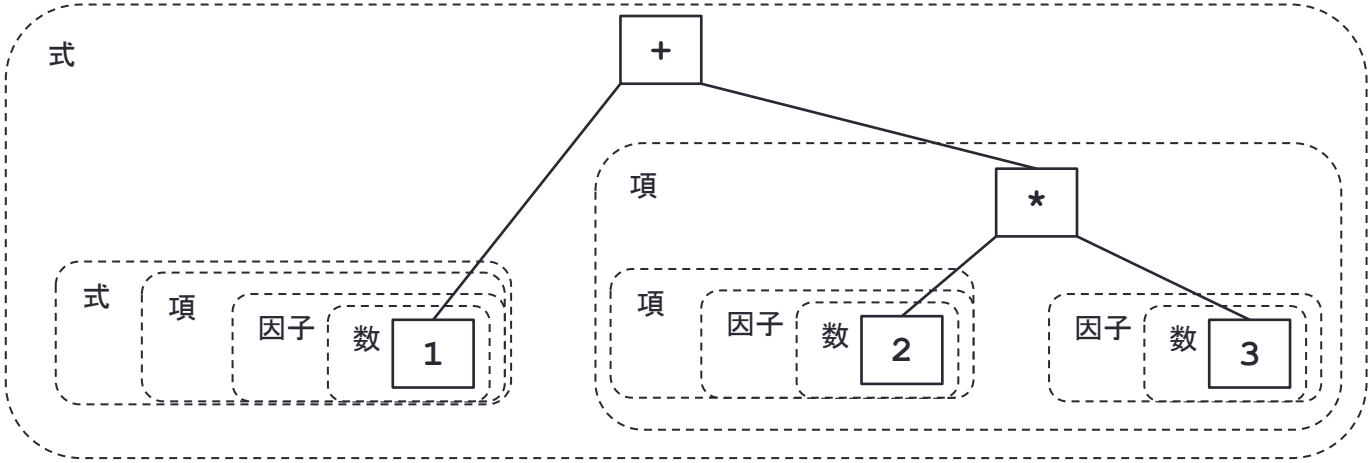




曖昧性のない式の構文規則

- 優先度別に式の規則を分ける

```
<式> ::= <式> '+' <項> | <式> '-' <項> | <項>

<項> ::= <項> '*' <因子> | <項> '/' <因子> | <因子>

<因子> ::= <数> | '-' <因子> | '(' <式> ')'
```

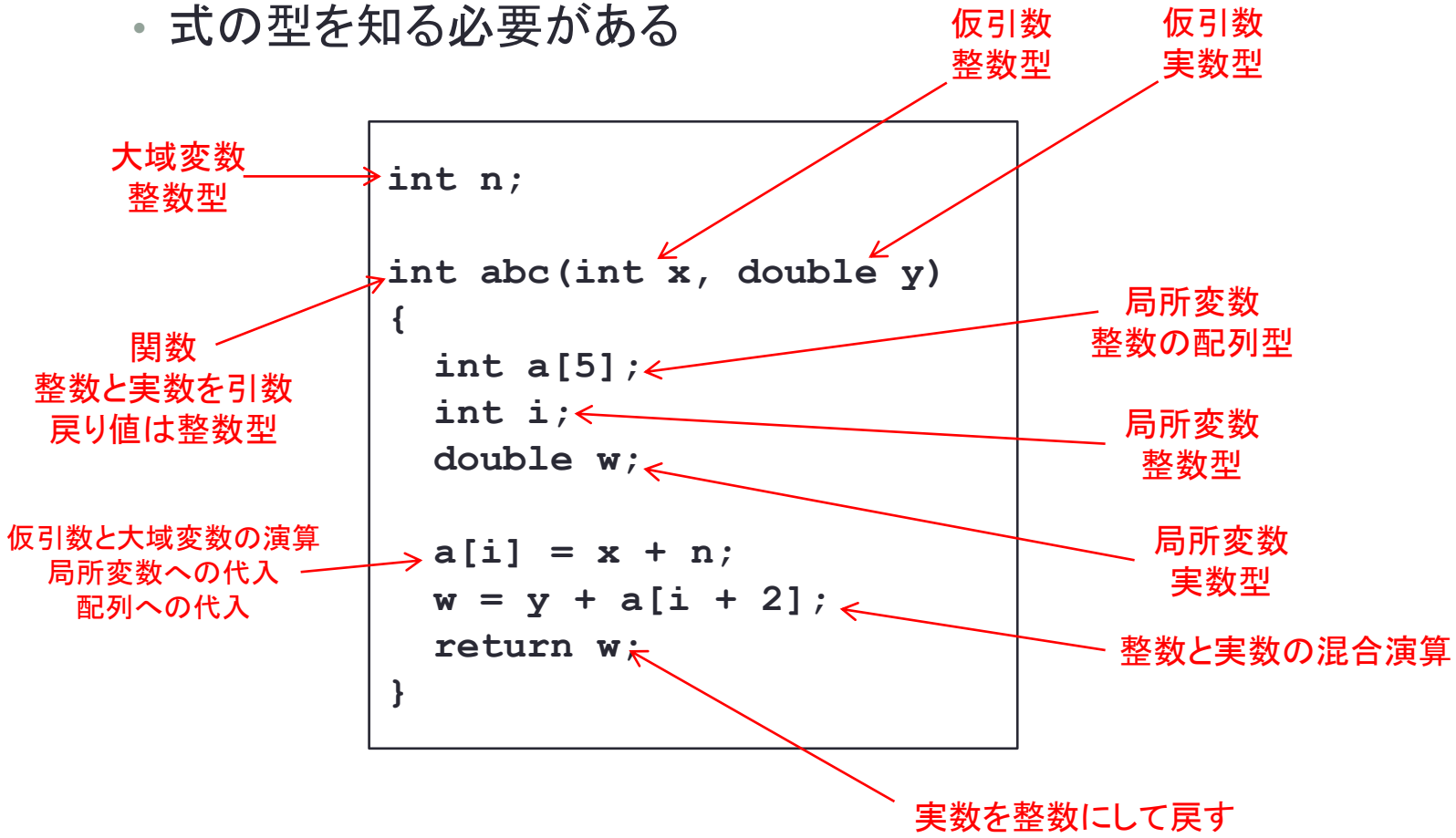


C言語の文のBNF構文

```
<文> ::= <式> ';' |

         'if' '(' <式> ')' <文> ( 'else' <文> )? |

         'while' '(' <式> ')' <文> |

         'for' '(' <式> ';' <式> ';' <式> ')' <文> |

         'switch' '(' <式> ')' '{'
           (('case' <定数> | 'default') ':' <文>*)* '}' |

         'do' <文> 'while' '(' <式> ')' |

         '{' <文>* '}'
```

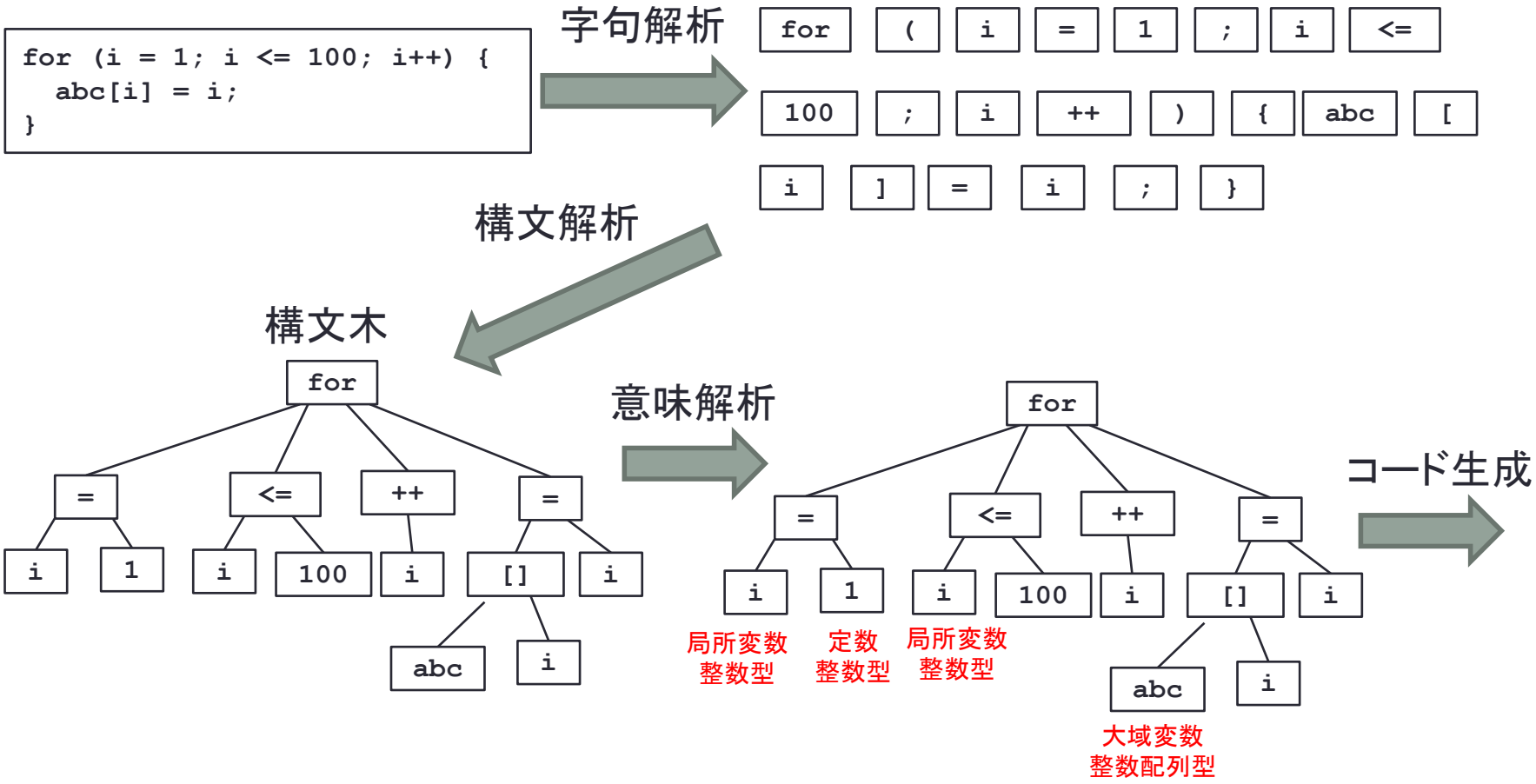
意味解析

- 構文が正しいだけでは翻訳できない
  - 変数は前もって宣言しておく必要がある
  - 式の型を知る必要がある



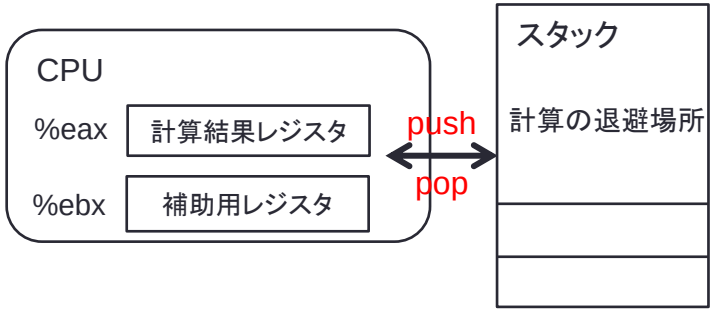
コード生成

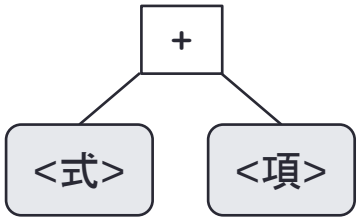
- 字句解析，構文解析，意味解析が終わるとコードを生成することができる



コード生成の方法

- コード生成（code generation）
  - 構文木からターゲットとするCPUのアーキテクチャにあったコードを生成
  - 変数や引数に対する参照の解決
- コード生成の方法
  - 構文木ごとにコード生成を決めればよい.
- 例
  - 式の足し算に関するコード生成
  - i386のコードを生成





コード生成

計算結果は%eaxに入れる

<式>に対するコード

<式>を計算するコード

```
pushl %eax
```

<式>の計算結果をスタックに退避

<項>に対するコード

<項>を計算するコード

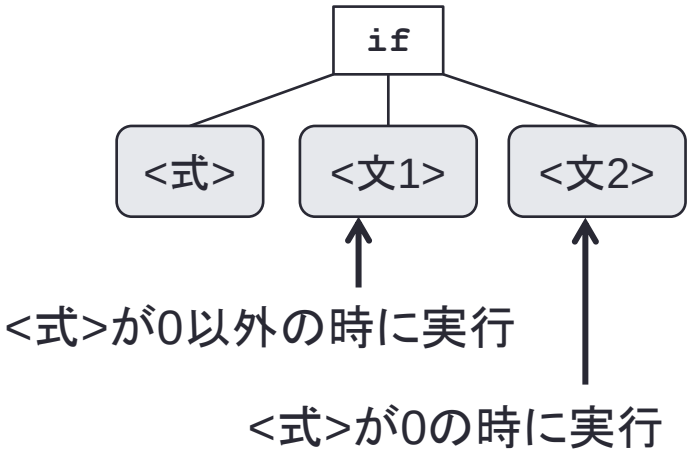
```
movl %eax,%ebx
popl %eax
addl %ebx,%eax
```

<式>の計算結果をスタックから戻し<項>計算結果を加える

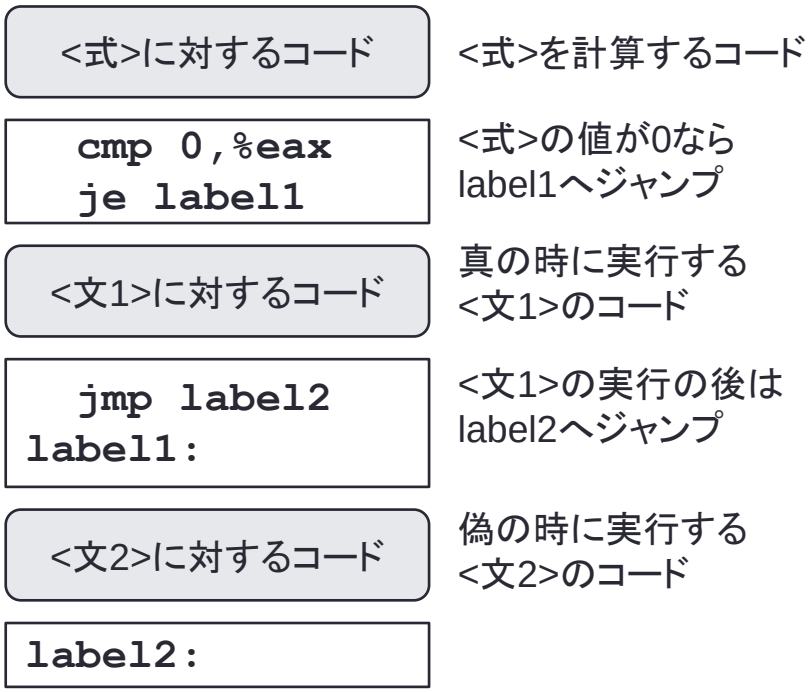
<式> ::= <式> '+' <項>

if文のコード生成

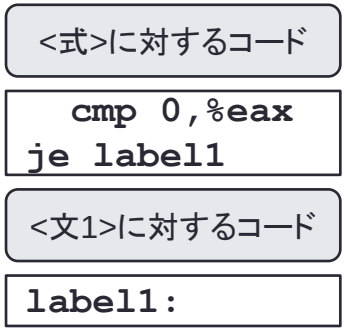
```
<if文> ::= 'if' '(' <式> ')' <文1> ( 'else' <文2> )?
```



コード生成

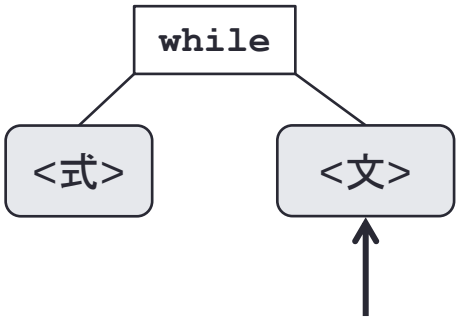


<文2>がない時の
コード生成



while文のコード生成

<while文> ::= 'while' '(' <式> ')' <文>



<式>が0になるまで実行

コード生成

```
label1:
```
<式>に対するコード　　<式>を計算するコード

```
  cmp 0,%eax
je label2
```
<式>の値が0なら label2へジャンプして終了

<文>に対するコード　　<文>のコード

```
  jmp label1
label2:
```
もう一度<式>をチェック

より良い コード生成

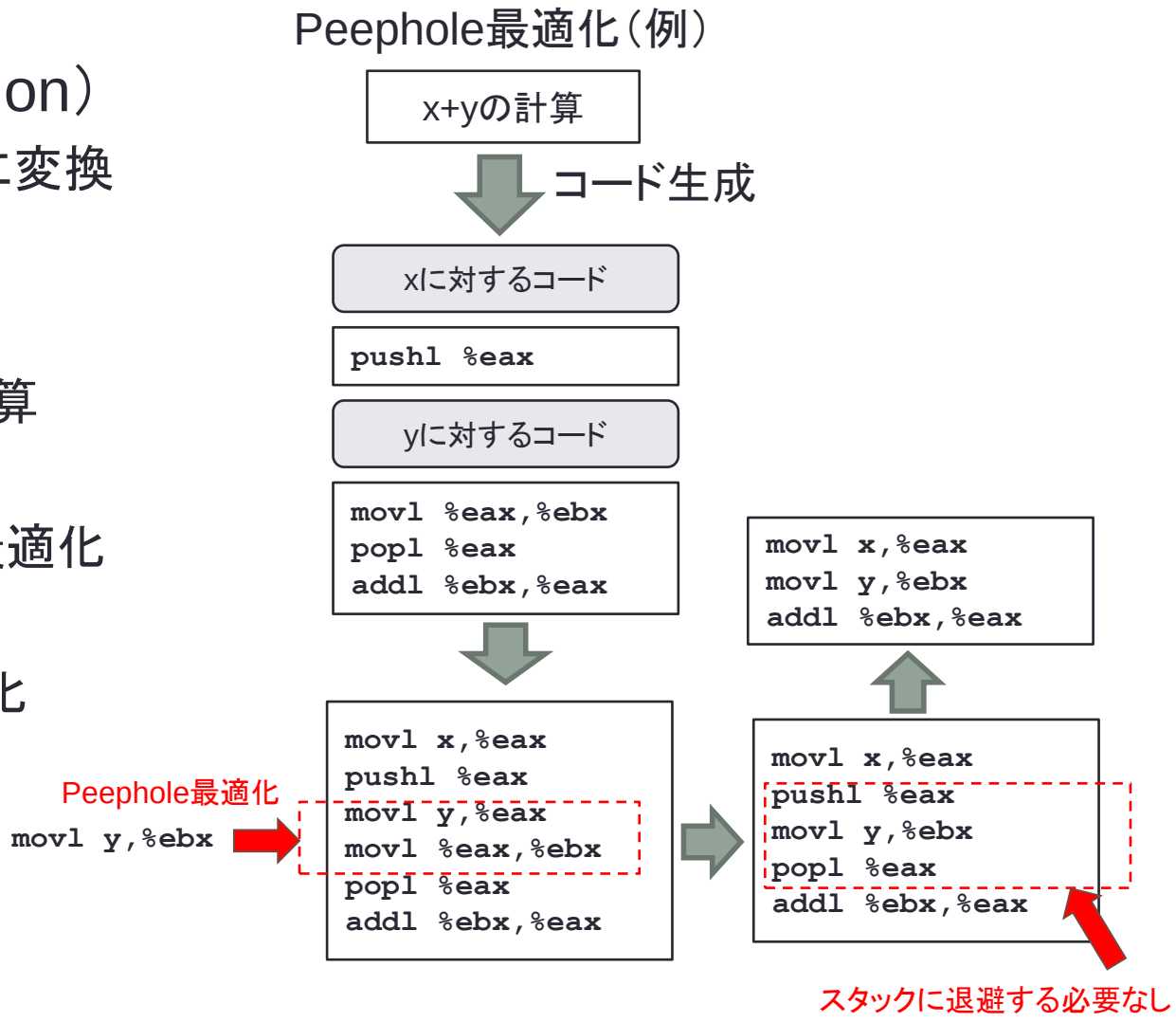
```
  jmp label2
label1:
```
<文>に対するコード

```
label2:
```
<式>に対するコード

```
  cmp 0,%eax
  jne label1
```

最適化

- 最適化（optimization）
  - 効率の良いコードに変換
- 最適化の手法
  - 定数だけの式の計算
  - Peephole最適化
  - 繰り返しにおける最適化
  - レジスタの利用
  - グローバルな最適化



繰り返しの最適化

- 繰り返しで変化しないものは外に出す
- 掛け算は足し算に置き換える

```
for (i = 0; i < 1000; i++) {
  b = x * x + 3;
  a = a + b + i * 4;
}
```

変化しないものは外に出す

```
b = x * x + 3;
for (i = 0; i < 1000; i++) {
  a = a + b + i * 4;
}
```

掛け算を
足し算に

```
b = x * x + 3;
j = 0;
for (i = 0; i < 1000; i++) {
  a = a + b + j;
  j = j + 4;
}
```

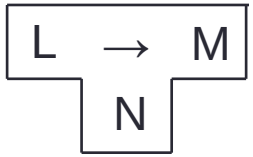
計算コスト

足し算
引き算 < 掛け算 < 割り算

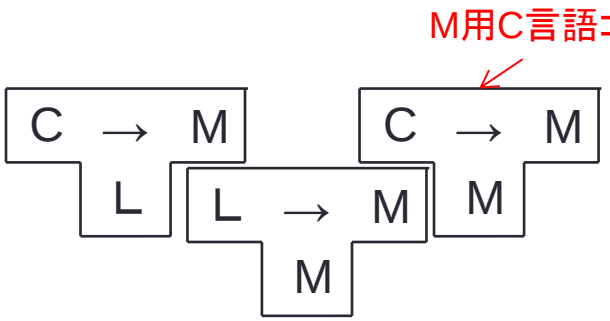
コンパイラは何言語で書く？

- C言語のコンパイラも高級言語で書かれている
  - コンパイラをコンパイルしなくてはいけない
  - 最初のC言語コンパイラは別言語で書く必要がある
  - 一度できるとC言語自身でC言語のコンパイラを書いても良い

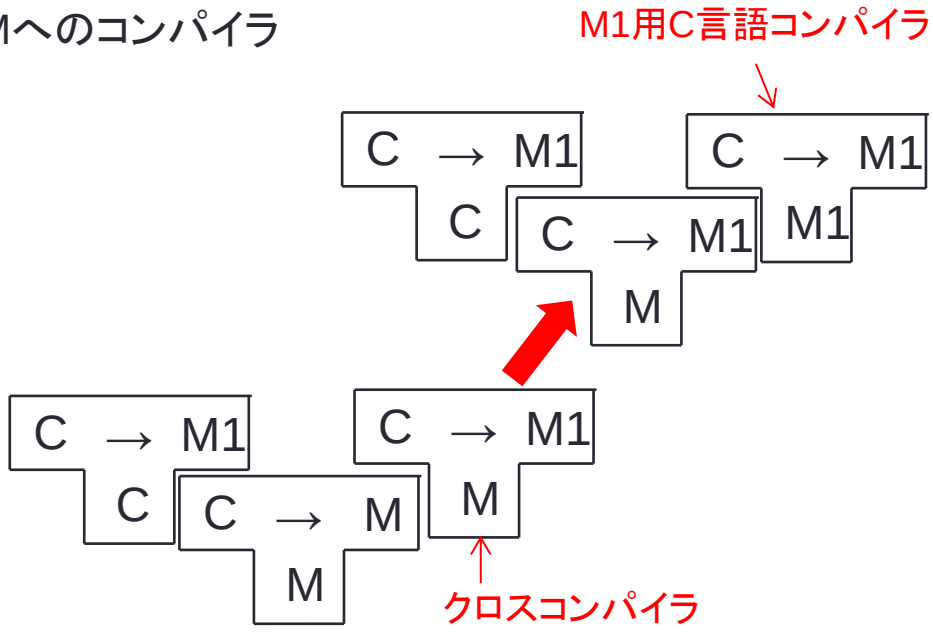
- T図式



N言語で書かれた
言語Lから言語Mへのコンパイラ



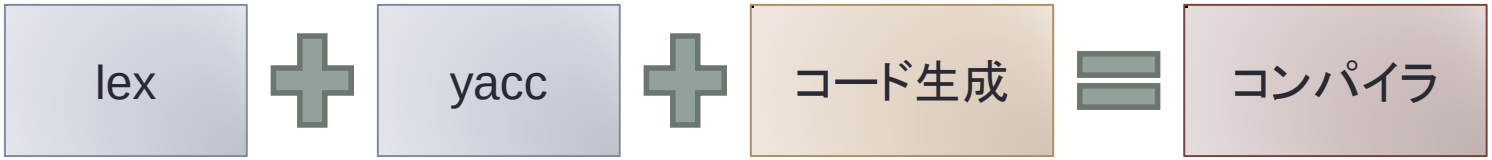
L言語で書かれたC言語コンパイラを
L言語コンパイラでコンパイル



C言語コンパイラのM1への移植

コンパイラコンパイラ

- コンパイラを最初からすべて書くのは面倒
  - 字句解析を自動生成
    - lex
    - 正規表現を使って字句を定義
  - 構文解析を自動生成
    - yacc
    - Yet Another Compiler Compiler
    - 文脈自由文法から構文解析を自動生成



まとめ

- 言語処理系
  - インタープリタ
  - コンパイラ

- コンパイラの処理
  - 前処理（preprocess）
  - 字句解析（lexical analysis）
  - 構文解析（syntax analysis）
  - コード生成（code generation）
  - 最適化（optimization）