

コード生成

概要

- コード生成
式のコード生成、文、文の列のコード生成

演習で作るコンパイラの例

test.hcc

```
Int main()
{  int i j;
   i = 3;
   j = 4;
   putint(i+j);
}
```

test.hsm

PUSH	0	2
LDC	0	3
STV	0	0
LDC	0	4
STV	0	1
LDV	0	0
LDV	0	1
AD	0	0
WRI	0	0
POP	0	2
HLT	0	0

Test.classを逆アセンブル
javap -c Testで表示

```
0:      iconst_3
1:      istore_1
2:      iconst_4
3:      istore_2
4:      getstatic    #2;
7:      iload_1
8:      iload_2
9:      iadd
10:     invokevirtual    #3;
13:     return
```

Hsm (HiStackMachine)の概要(1)

- 演習で用いる仮想機械(スタックマシン)

- 構成

プログラムP

- 命令列の置き場

プログラムカウンタ(pc)

- 次に実行する命令を指示

スタック(S)

- 演算対象(被演算数、演算結果)を置く
- 記憶域

スタックポインタ(sp)

- スタックトップを指す

フレームポインタ(fp)

- 関数(手続き)のフレームの開始アドレス(後の講義で説明)

Hsm (HiStackMachine)の概要(2)

- 命令セット

- (1) ロード・ストア命令

- ロード命令:スタックトップに値を置く。
 - ストア命令:記憶域として確保した所に値を保存する。
 - 記憶域の確保、開放の命令

- (2) 演算命令

- 算術演算、関係演算。(論理演算はない)

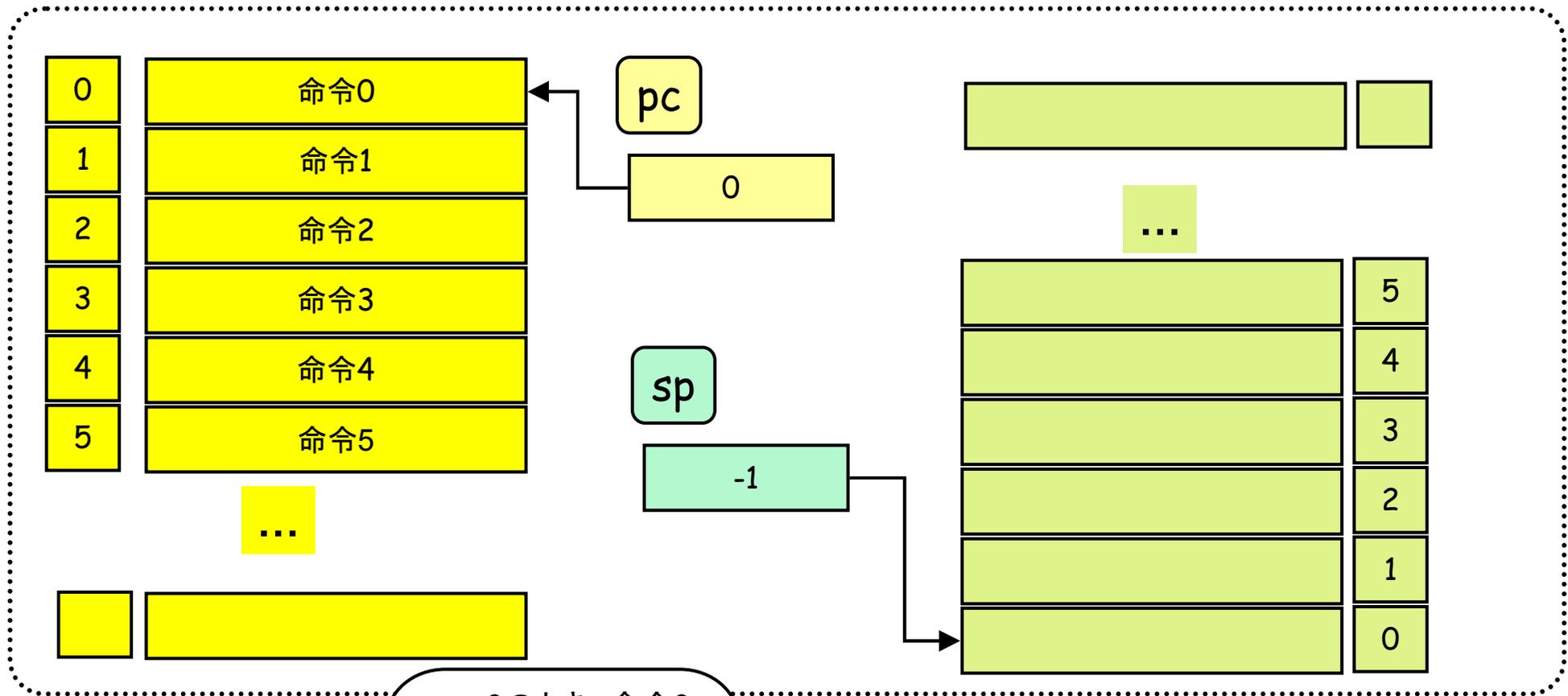
- (3) ジャンプ命令、制御命令

- 無条件ジャンプ、条件ジャンプ、停止命令

- (4) 入出力命令

- 入力、出力

hsmの構成図



P

pc = 0のとき、命令0
を実行する。
pc = 1のとき、命令1
を実行する
...
pc = nのとき命令nを
実行する

S

値をロードする場合には、 $sp \leftarrow sp + 1$ として
 $S[sp]$ に置く。

出力命令

WRI 0 0 整数表示:
S[sp]を表示; sp--; pc++;

WRC 0 0 文字表示:
文字コードS[sp]に対する文字の表示;
sp--; pc++;

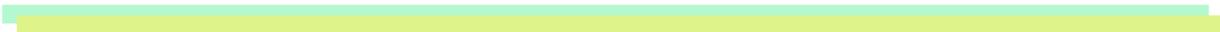
WNL 0 0 改行表示:
改行; pc++;

LDC 0 1
WRI 0 0

整数1を表示

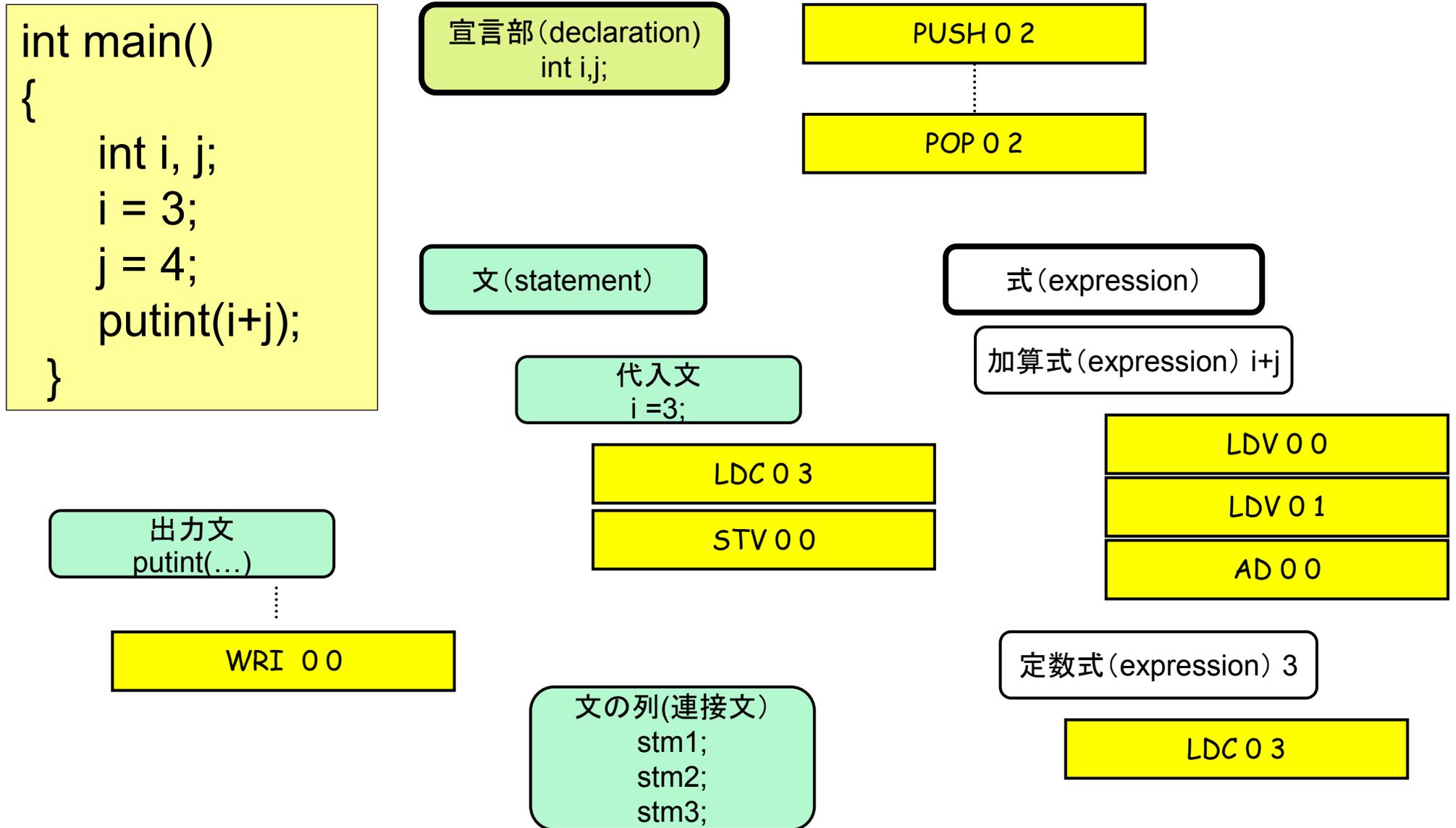
LDC 0 70
WRC 0 0

文字コード70番の文字(F)を表示

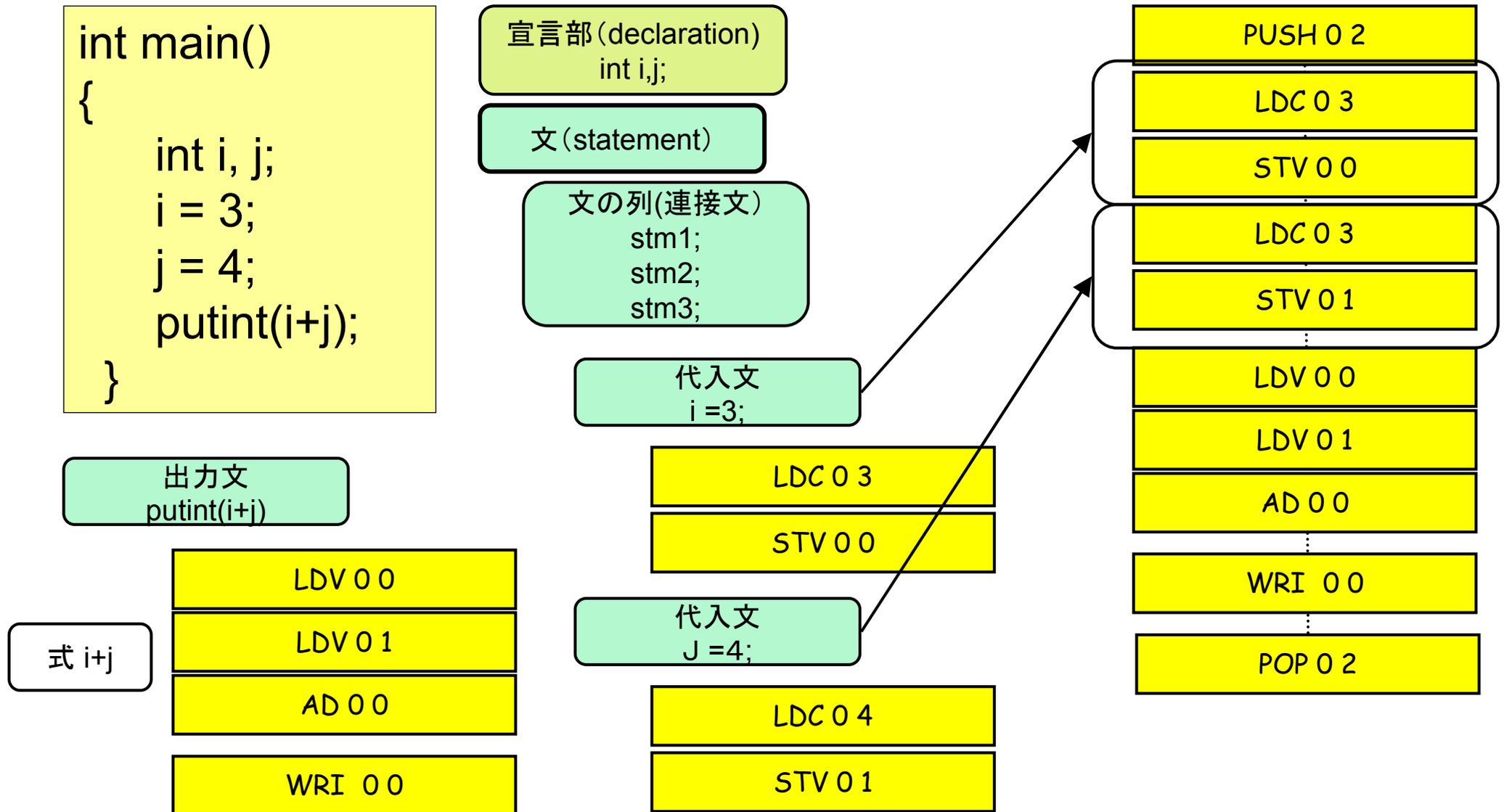


コード生成

プログラムの構成要素(パーツ)とコード生成

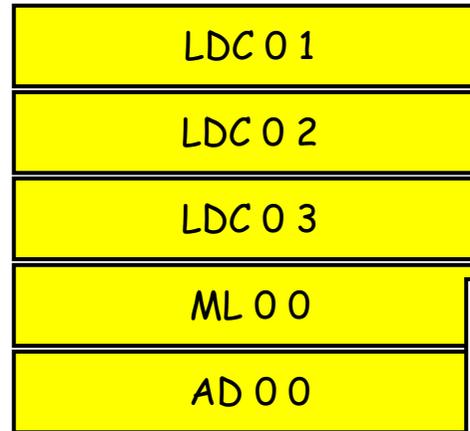


コード生成=各パーツのコードをつなぎ合わせる



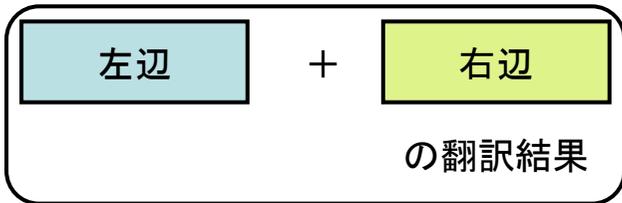
コード生成(式)

1 + 2 * 3



定数式(3つ)、加算式、
乗算式の各パーツのコード
を組み合わせる

加算式のコード生成ルール
(パーツの組み合わせ方)



左辺式の翻訳結果(コード生成の結果)の後に、
右辺式の翻訳結果を並べて、
最後に命令"AD 00"をくっつける

原始言語(Source Language):式の文法

```
<EXPRESSION> ::= <TERM>
                | <EXPRESSION> '+' <TERM>
                | <EXPRESSION> '-' <TERM>
<TERM> ::= <UNARY>
          | <TERM> '*' <UNARY>
          | <TERM> '/' <UNARY>
<UNARY> ::= <FACTOR>
          | '-' <UNARY>
<FACTOR> ::= <IDENT>
            | <NUMBER>
            | '(' <EXPRESSION> ')'
<IDENT> ::= a~zの英字
<NUMBER> ::= 数字の1回以上の繰り返し文字列
```

「式」のコード生成(1)

式 の翻訳結果

左辺 + 右辺
の翻訳結果

=

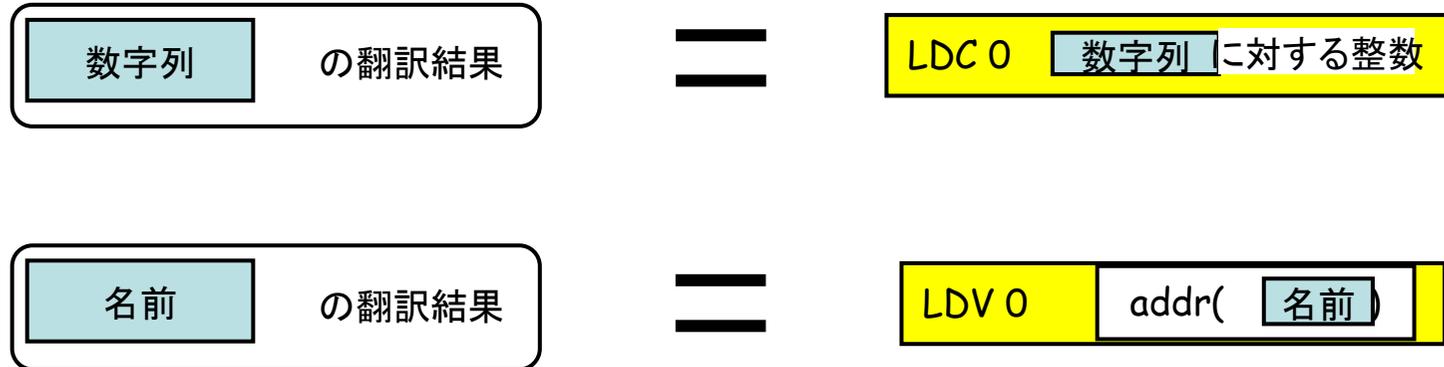
左辺 の翻訳結果
右辺 の翻訳結果
AD 0 0

- 式 の翻訳結果

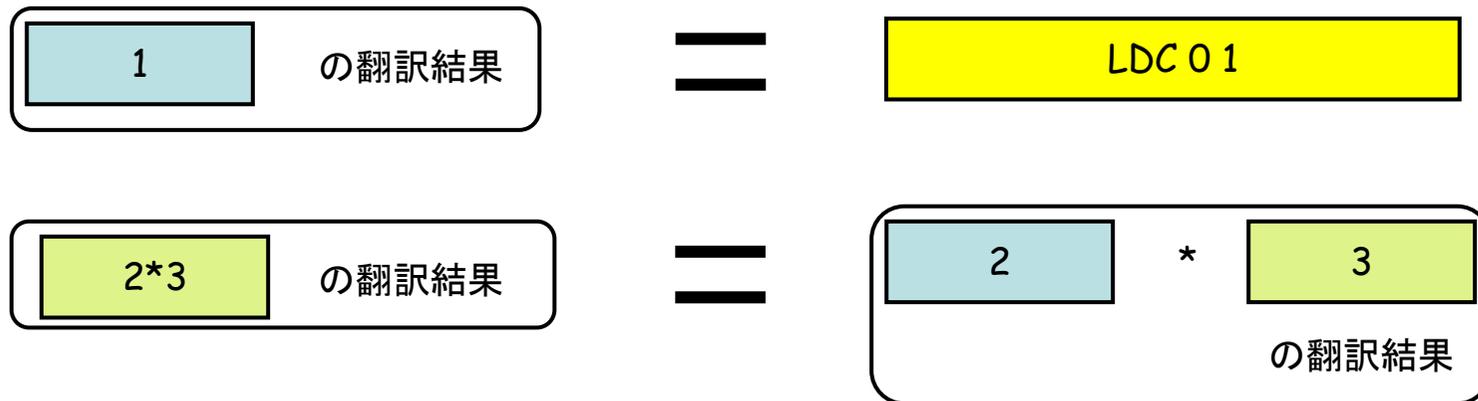
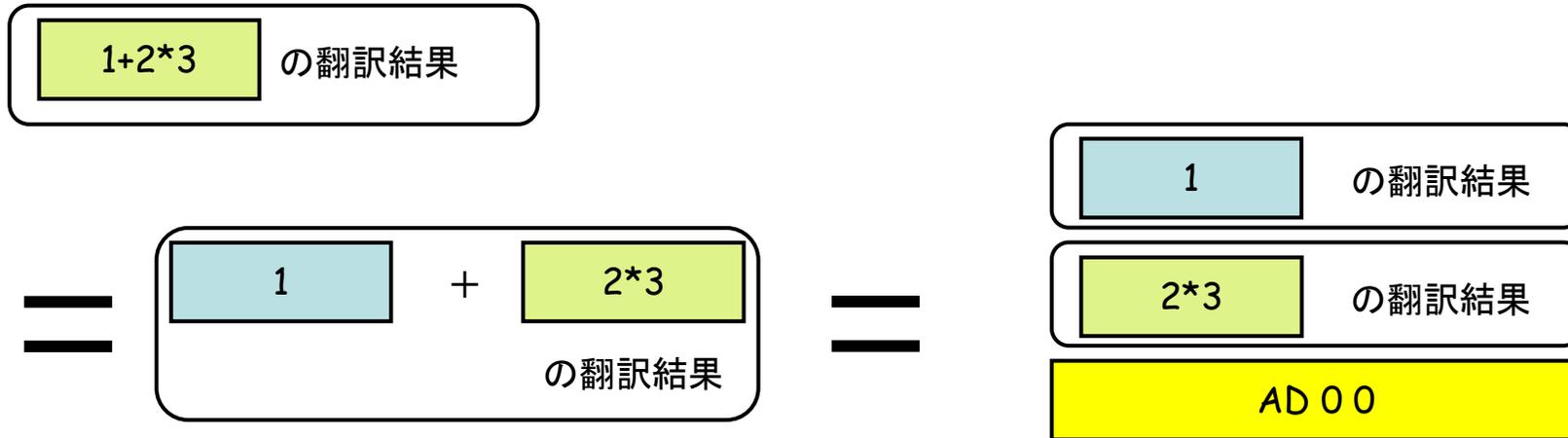
=

式 の翻訳結果
NEG 0 0

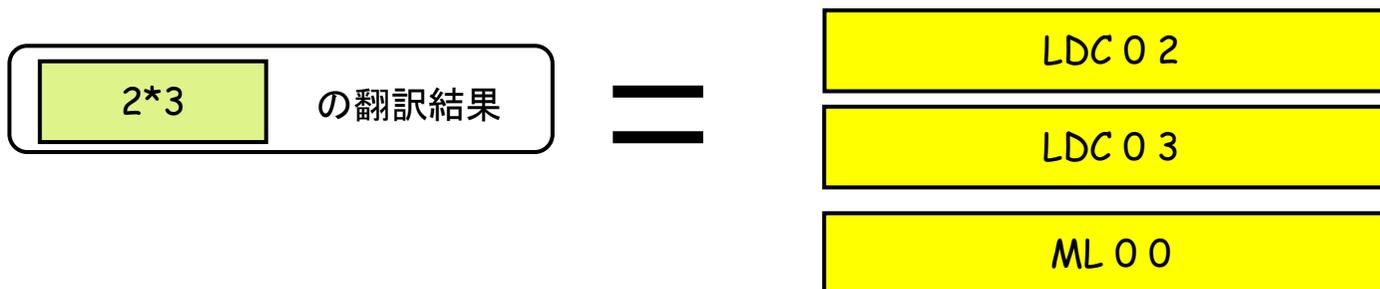
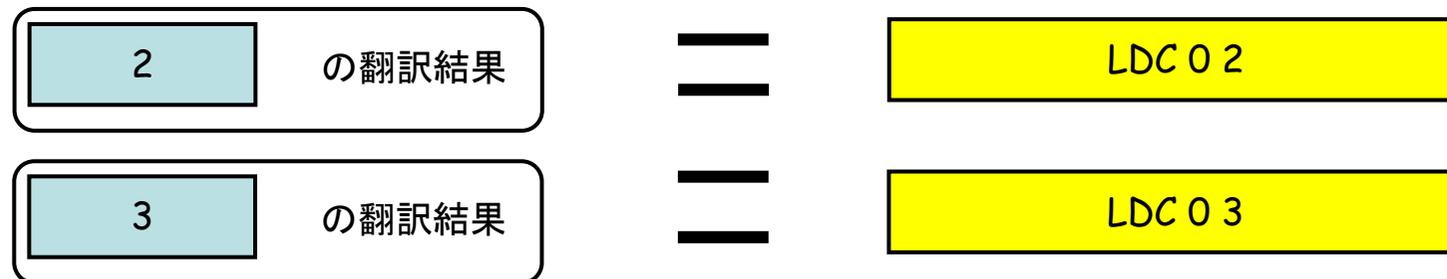
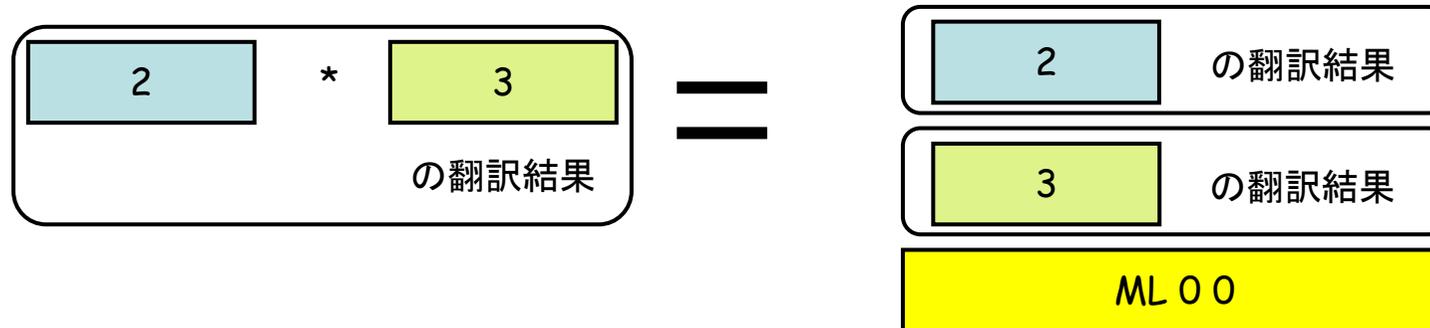
「式」のコード生成(2)



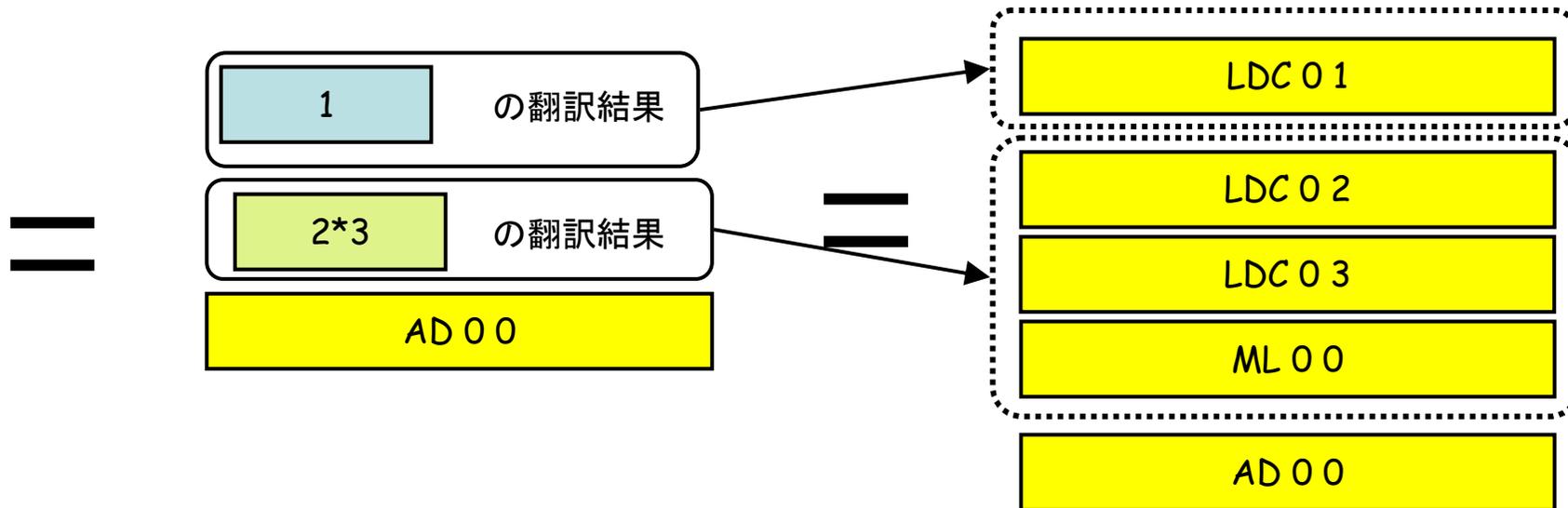
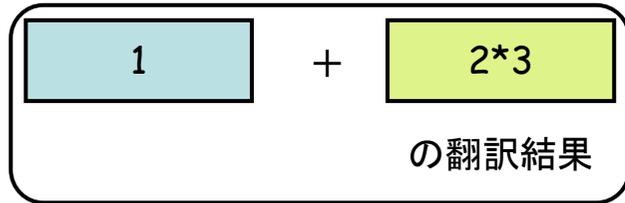
例: $1+2*3$ (1)



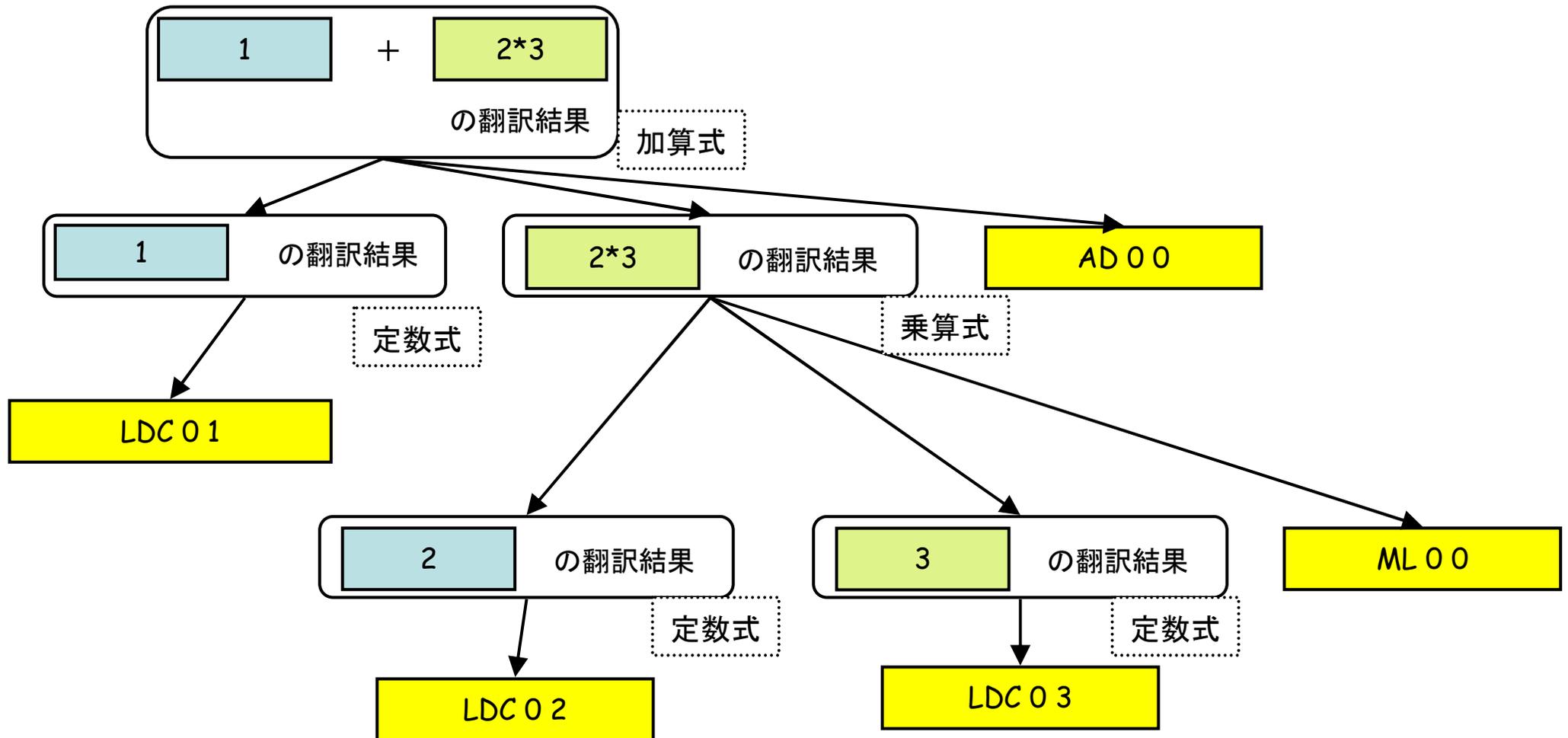
例: $1+2*3$ (2)



例: $1+2*3$ (3)

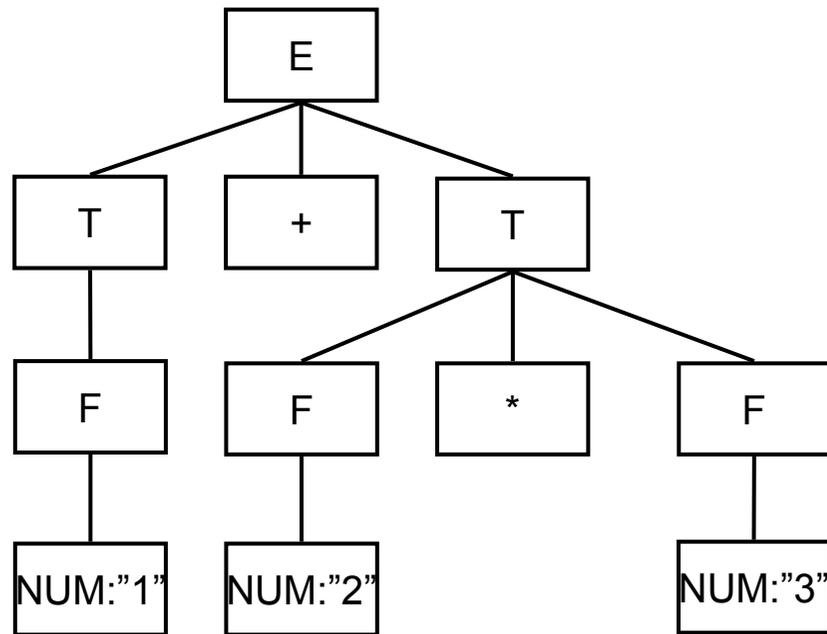


例: $1+2*3$ (4)



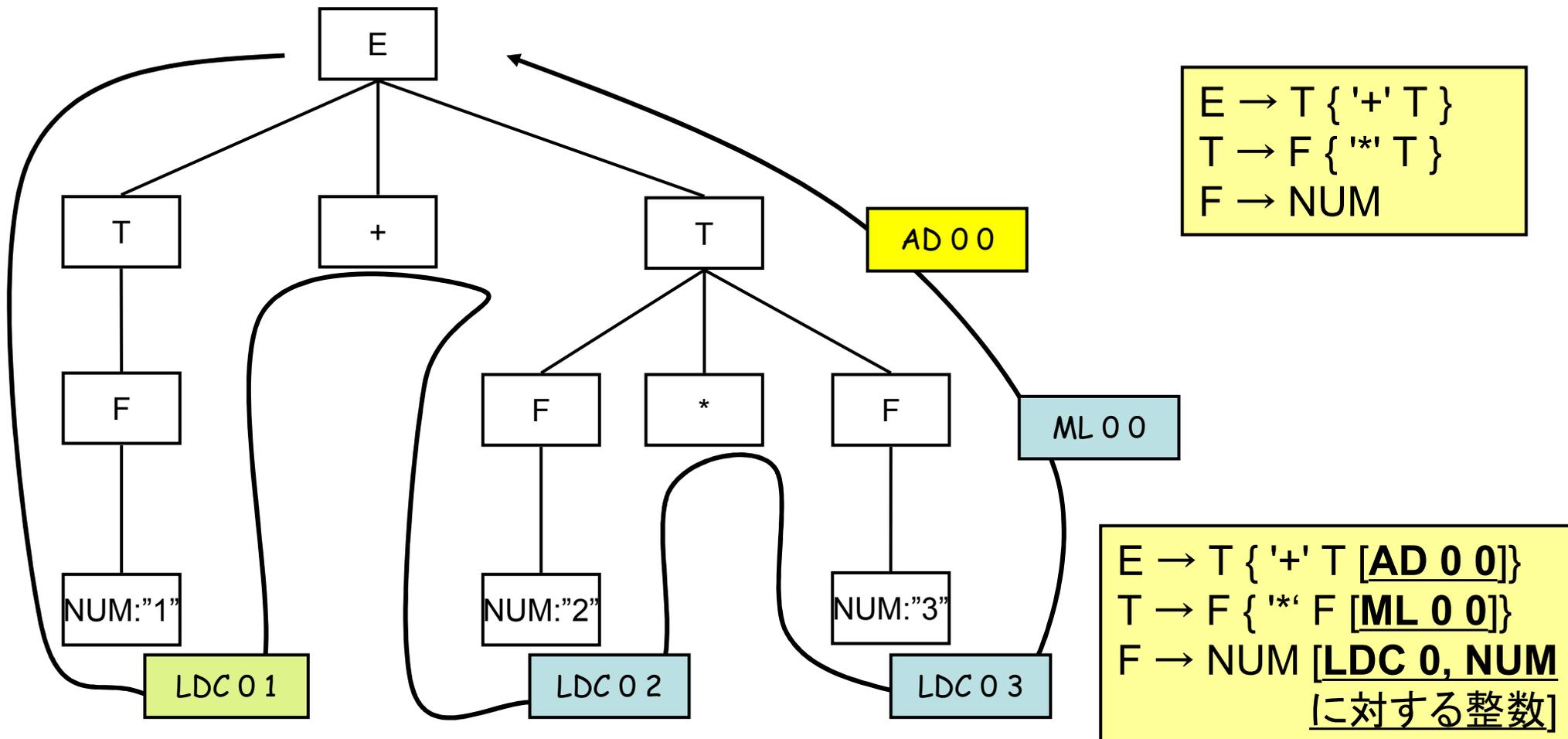
構文解析と同時に行うコード生成

1 + 2*3

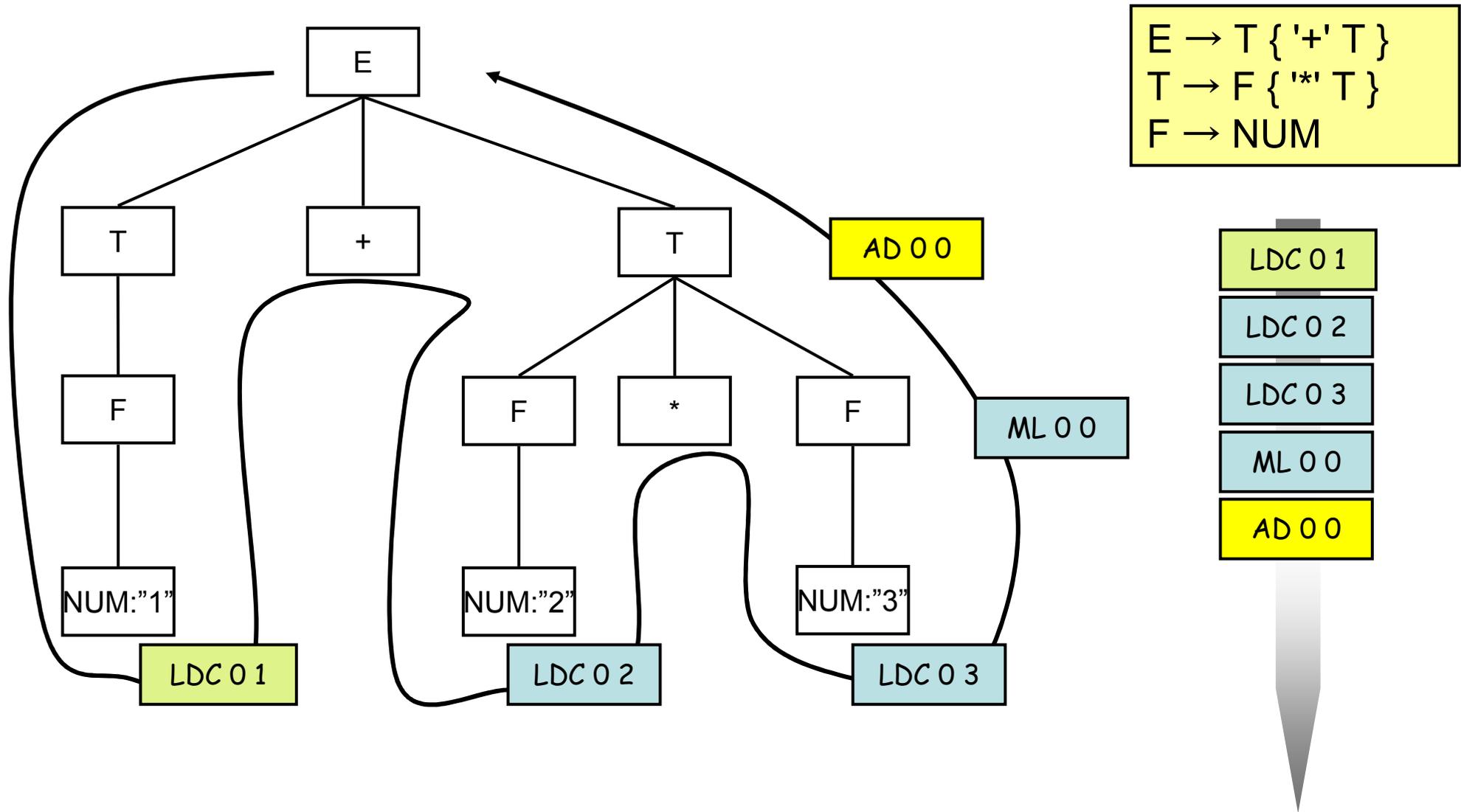


$E \rightarrow T \{ '+' T \}$
 $T \rightarrow F \{ '*' F \}$
 $F \rightarrow \text{NUM}$

構文解析と同時に行うコード生成



構文解析と同時に行うコード生成



練習問題(2)

- $1+2*3+4*5$ に対するコード生成を前頁のスライドを参考に、構文木に沿った形で行え。
- $(1+2)*3$ の場合は、どうか？

原始言語(Source Language): 文、ブロック等

```
<PROGRAM> ::= <MAIN>
<MAIN> ::= 'int' 'main' '(' ')' <BLOCK>
<BLOCK> ::= '{' <STATEMENTLIST> '}'
<STATEMENTLIST> ::= empty
                | <STATEMENTLIST> <STATEMENT>
<STATEMENT> ::= <SUBSTITUTION> '=' <EXPRESSION> ';'
                | '{' <STATEMENTLIST> '}'
                | 'putint' '(' <EXPRESSION> ')' ';'
                | 'putnl';
<SUBSTITUTION> ::= <IDENT>
```

putint <EXPRESSION>の値を整数で出力
putnl 改行コードをコンソールに出力する

出力文のコード生成

```
putint(123);
```

```
LDC 0 123
```

```
WRI 0 0
```

```
putint(1+2*3);
```

```
LDC 0 1
```

```
LDC 0 2
```

```
LDC 0 3
```

```
ML 0 0
```

```
AD 0 0
```

```
WRI 0 0
```

```
WRI 0 0 整数表示: S[sp]を表示; sp--; pc++;
```

```
putint( 式 );
```

の翻訳結果

=

```
式
```

の翻訳結果

```
WRI 0 0
```

文の列(statementlist)のコード生成

```
putint(123);
```

```
LDC 0 123
```

```
WRI 0 0
```

```
putint(123);  
putint(1+2*3);
```

```
putint(1+2*3);
```

```
LDC 0 1
```

```
LDC 0 2
```

```
LDC 0 3
```

```
ML 0 0
```

```
AD 0 0
```

```
WRI 0 0
```

```
LDC 0 123
```

```
WRI 0 0
```

```
LDC 0 1
```

```
LDC 0 2
```

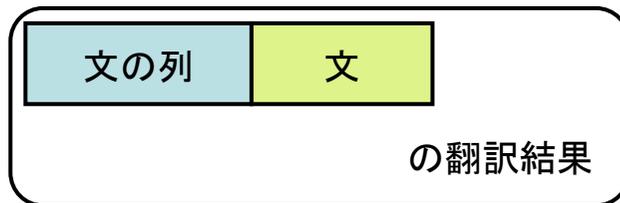
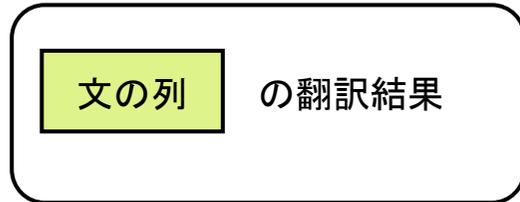
```
LDC 0 3
```

```
ML 0 0
```

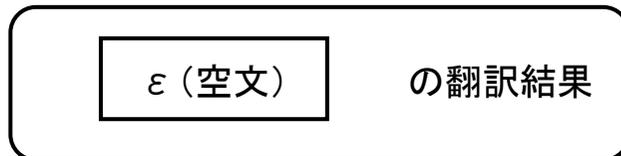
```
AD 0 0
```

```
WRI 0 0
```

「文の列」(statementlist)のコード生成



=



=

ϕ

「文」(statement)のコード生成

文 の翻訳結果

名前 = 式 ;
の翻訳結果

=

式 の翻訳結果
STV 0 addr(名前)

{ 文の列 }
の翻訳結果

=

文の列 の翻訳結果

putint(式);
の翻訳結果

=

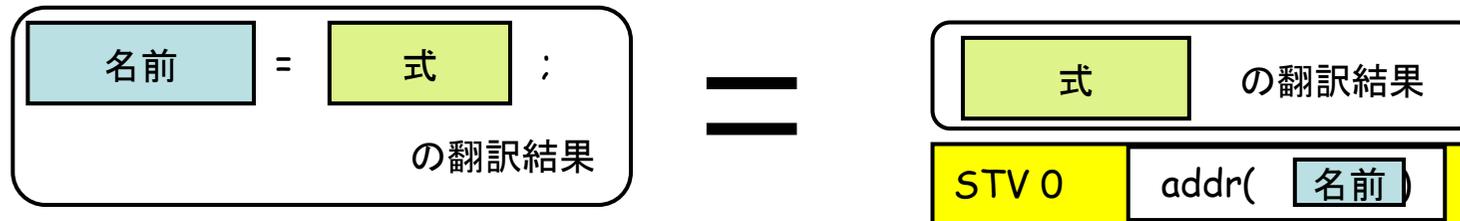
式 の翻訳結果
WRI 0 0

putnl; の翻訳結果

=

WNL 0 0

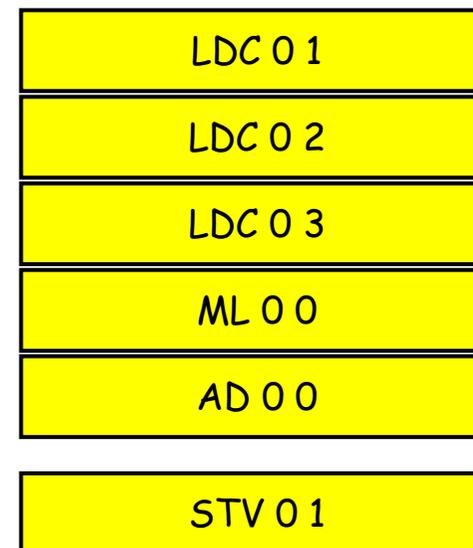
代入文のコード生成



addr(名前)は、名前に対して割り当てた記憶域のアドレス

a = 1+2*3 ;

今、変数aの値を保持するアドレスが1 (addr('a')=1) だとすれば、、、



練習問題(3)

- 下記のプログラムを、構文木に沿ってコード生成することで翻訳せよ。ただし”int a; int b;”の宣言部は、変数a,bの値を保持する領域を確保するコード(PUSH 0 2)に翻訳されるものとする。また、addr(a)=0, addr(b)=1とする。

```
int a;  
int b;  
a = 1+2*3;  
b = a * 4;  
putint(a);
```

記号表(変数表)を使った記憶域の管理

- 宣言された変数の記憶場所などを管理するために、記号表が必要となる。

```
int main()
{
  int abc;
  int cd;
  int efg;
  cd = 10;
  putint(cd);
}
```

記号表

ident="abc"	address=0
ident="cd"	address=1
ident="efg"	address=2

名前(エントリ)..

記号表に登録する内容

ident="abc"	address=0
-------------	-----------

つづり

値を保持する
アドレス

記号表(変数表)を使った記憶域の管理

- 必要なメモリの領域確保。(下記では3変数分必要)
- `cd = 10, putint(cd)`の各文では、`addr(cd)`を知る必要がある。
記号表から`ident`が"`cd`"である名前(エントリ)を検索する。

```
int main()
{
  int abc;
  int cd;
  int efg;
  cd = 10;
  putint(cd);
}
```

記号表

ident="abc"	address=0
ident="cd"	address=1
ident="efg"	address=2

コード

PUSH 0 3
LDC 0 10
STV 0 1
LDV 0 1
WRI 0 0
POP 0 3
HLT 0 0

コンパイラ作成の流れと、プログラムの実行

