# コンパイラ講義資料

末永幸平

# 謝辞


- この講義資料へのコメントをくださった以下の方々に感謝します: 馬谷誠二氏, 石崎一明氏

# 注意

- この講義資料は学期途中でアップデートする可能性があります
  - 特に最適化のところはもっと拡充したいなあとか考えています
- 最新版は講義の Web ページをチェックしてください

# コンパイラって何だ?

実験1でわけも分からず
やっていたことを分かるようになろう

```
> cat kadai1.c
itn main(int argc, char **argv)
{
  return 0;
}
> gcc kadai1.c
kadai1.c:2: error: expected '=',
',', ';', 'asm' or
'__attribute__' before 'main'
>
```

```
> cat kadai1.c

int main(int argc, char **argv)
{
  return x;
}
> gcc kadai1.c
kadai1.c: In function 'main':
kadai1.c:4: error: 'x' undeclared
(first use in this function)
kadai1.c:4: error: (Each undeclared
identifier is reported only once
kadai1.c:4: error: for each function it
appears in.)
>
```

```
> cat kadai1.c
int main(int argc, char **argv)
{
  return 0;
}
> gcc kadai1.c
> ./a.out
>
```

```
> cat kadai1.c

#include <stdio.h>

int main(int argc, char **argv)
{
  printf("Hello!¥n");
  return 0;
}
> gcc kadai1.c
> ./a.out
Hello!
>
```

# ちょっと考えてみましょう

- さっきの gcc ってコマンドは何をしてるのか
- さっきの gcc ってコマンドはなぜ必要なのか

# gcc がしてること

- C のソースコードを実行可能ファイルに変換

```
#include <stdio.h>

int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

```
0000000 facf feed 0007 0100 0003 8000 0002 0000
0000020 0010 0000 05b0 0000 0085 0020 0000 0000
0000040 0019 0000 0048 0000 5f5f 4150 4547 455a
0000060 4f52 0000 0000 0000 0000 0000 0000 0000
0000100 0000 0000 0001 0000 0000 0000 0000 0000
0000120 0000 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0019 0000 0228 0000
0000160 5f5f 4554 5458 0000 0000 0000 0000 0000
0000200 0000 0000 0001 0000 1000 0000 0000 0000
0000220 0000 0000 0000 0000 1000 0000 0000 0000
0000240 0007 0000 0005 0000 0006 0000 0000 0000
0000260 5f5f 6574 7478 0000 0000 0000 0000 0000
0000300 5f5f 4554 5458 0000 0000 0000 0000 0000
0000320 0eb0 0000 0001 0000 0076 0000 0000 0000
0000340 0eb0 0000 0004 0000 0000 0000 0000 0000
0000360 0400 8000 0000 0000 0000 0000 0000 0000
0000400 5f5f 7473 6275 0073 0000 0000 0000 0000
0000420 5f5f 4554 5458 0000 0000 0000 0000 0000
0000440 0f26 0000 0001 0000 000c 0000 0000 0000
0000460 0f26 0000 0001 0000 0000 0000 0000 0000
0000500 0408 8000 0000 0000 0006 0000 0000 0000
```

# C プログラムを実行するために gcc kadai1.c が必要な理由

- コンピュータは C 言語のソースコードを
  そのまま実行できないから

# コンパイラとは

- 高級言語を低級言語に変換するソフトウェア
  - 高級言語: C とか Java とか Scheme とかの人間にとって比較的わかりやすい言語
  - 低級言語: 機械語, アセンブリ言語等のコンピュータにとって比較的わかりやすい言語
- gcc はコンパイラの一種

# この科目について

# コンパイラについて学ぶ科目です

- 2回生後期配当専門科目「コンパイラ」
- 開講時限: 月曜3限
- 担当: 末永幸平
- 到達目標
  - コンパイラに関する知識を身につける
  - 簡単なコンパイラを必要な参考文献を参照しつつ作れる
- 評価や履修上の注意は講義ホームページを参照のこと

# この科目で直接的に学べること

- コンパイラの一般的な動作と作り方
- コンパイラ実装に使われるいろいろなアルゴリズム
  - 字句解析
  - 構文解析
  - 意味解析
  - 中間コード生成
  - 最適化
  - アセンブリ生成
  - レジスタ割り付け
  - …

# この科目で間接的に学べること

- オレオレ言語 (i.e., DSL) の実装方法
  - 自分で言語を作って，その言語のための
    コンパイラを作れるようになるかもしれない
- ある程度大きいソフトウェアを作るときに
  問題を分割するための戦略
- MIPS アセンブリ言語
- Racket 言語
  - Scheme の一種
- (英語の読解力?)

# レポート課題に関する注意

- 締切厳守
  - 締切に遅れた課題は一切採点しません
    - 遅延を認めてもあまりクオリティは上がらないようなので
    - TA にも自分の研究をする時間が必要
- 剽窃は厳しくカウント
  - 去年以前に提出された課題や Web 上の情報も含めてコピペのチェックをしているので注意
  - 発覚した場合はこの科目は 0 点
    - 悪質な場合はカンニングと同様の扱い

# コンパイラ入門の入門

コンパイラはどのように
構成されているか

# コンパイラがやること

- 高級言語→低級言語

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

```
0000000 facf feed 0007 0100 0003 8000 0002 0000
0000020 0010 0000 05b0 0000 0085 0020 0000 0000
0000040 0019 0000 0048 0000 5f5f 4150 4547 455a
0000060 4f52 0000 0000 0000 0000 0000 0000 0000
0000100 0000 0000 0001 0000 0000 0000 0000 0000
0000120 0000 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0019 0000 0228 0000
0000160 5f5f 4554 5458 0000 0000 0000 0000 0000
0000200 0000 0000 0001 0000 1000 0000 0000 0000
0000220 0000 0000 0000 0000 1000 0000 0000 0000
0000240 0007 0000 0005 0000 0006 0000 0000 0000
0000260 5f5f 6574 7478 0000 0000 0000 0000 0000
0000300 5f5f 4554 5458 0000 0000 0000 0000 0000
0000320 0eb0 0000 0001 0000 0076 0000 0000 0000
0000340 0eb0 0000 0004 0000 0000 0000 0000 0000
0000360 0400 8000 0000 0000 0000 0000 0000 0000
0000400 5f5f 7473 6275 0073 0000 0000 0000 0000
0000420 5f5f 4554 5458 0000 0000 0000 0000 0000
0000440 0f26 0000 0001 0000 000c 0000 0000 0000
0000460 0f26 0000 0001 0000 0000 0000 0000 0000
0000500 0408 8000 0000 0000 0006 0000 0000 0000
```

# コンパイラの難しさ

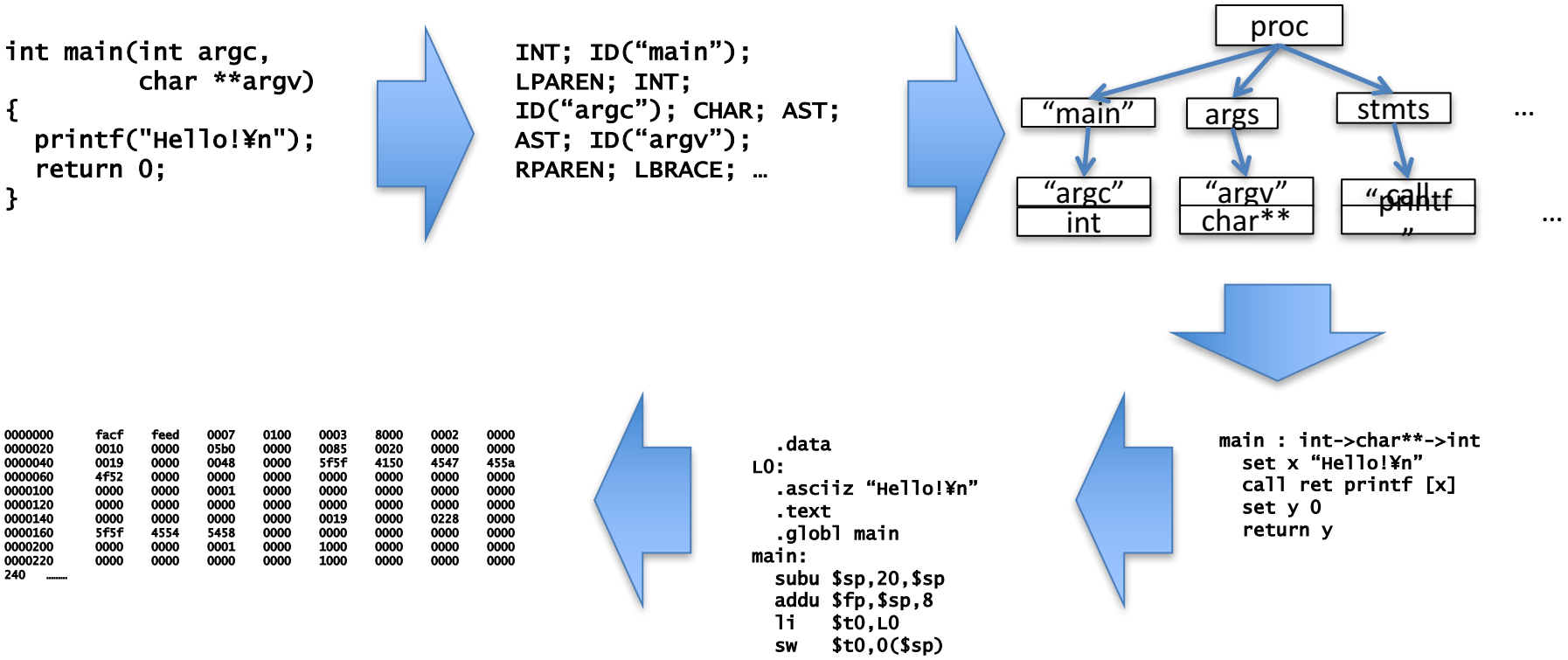
- 全然見かけの違うものに変換する必要

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

```
0000000 facf feed 0007 0100 0003 8000 0002 0000
0000020 0010 0000 05b0 0000 0085 0020 0000 0000
0000040 0019 0000 0048 0000 5f5f 4150 4547 455a
0000060 4f52 0000 0000 0000 0000 0000 0000 0000
0000100 0000 0000 0001 0000 0000 0000 0000 0000
0000120 0000 0000 0000 0000 0000 0000 0000 0000
0000140 0000 0000 0000 0000 0019 0000 0228 0000
0000160 5f5f 4554 5458 0000 0000 0000 0000 0000
0000200 0000 0000 0001 0000 1000 0000 0000 0000
0000220 0000 0000 0000 0000 1000 0000 0000 0000
0000240 0007 0000 0005 0000 0006 0000 0000 0000
0000260 5f5f 6574 7478 0000 0000 0000 0000 0000
0000300 5f5f 4554 5458 0000 0000 0000 0000 0000
0000320 0eb0 0000 0001 0000 0076 0000 0000 0000
0000340 0eb0 0000 0004 0000 0000 0000 0000 0000
0000360 0400 8000 0000 0000 0000 0000 0000 0000
0000400 5f5f 7473 6275 0073 0000 0000 0000 0000
0000420 5f5f 4554 5458 0000 0000 0000 0000 0000
0000440 0f26 0000 0001 0000 000c 0000 0000 0000
0000460 0f26 0000 0001 0000 0000 0000 0000 0000
0000500 0408 8000 0000 0000 0006 0000 0000 0000
```

# ではどうするか

- 間にいくつもフェーズを挟んで少しずつ変換
  - 少しずつコンピュータが読みやすい形にしていく




- この科目で学ぶこと ≒
  各フェーズで使うデータ構造と各フェーズ間の変換アルゴリズム

# コンピュータはどういう形が読みやすいか

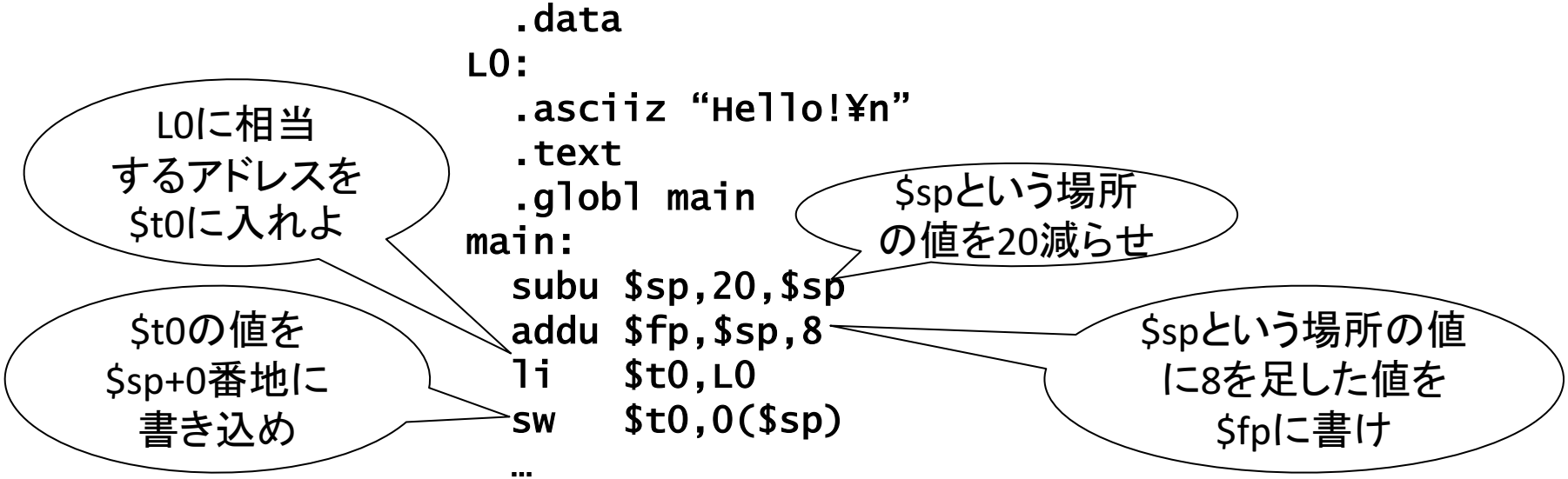
- CPU が次にどのような命令を実行しなければならないかが明確な形

```
0000000    facf    feed    0007    0100    0003    8000    0002    0000
0000020    0010    0000    05b0    0000    0085    0020    0000    0000
0000040    0019    0000    0048    0000    5f5f    4150    4547    455a
0000060    4f52    0000    0000    0000    0000    0000    0000    0000
0000100    0000    0000    0001    0000    0000    0000    0000    0000
0000120    0000    0000    0000    0000    0000    0000    0000    0000
0000140    0000    0000    0000    0000    0019    0000    0228    0000
0000160    5f5f    4554    5458    0000    0000    0000    0000    0000
0000200    0000    0000    0001    0000    1000    0000    0000    0000
0000220    0000    0000    0000    0000    1000    0000    0000    0000
240.....
```

- コンパイラはバイト列であるプログラムを命令の列に変換

# コンピュータはどういう形が読みやすいか

- CPU が次にどのような命令を実行しなければならないかを細かく書いてある形




- コンパイラはバイト列であるプログラムを命令の列に変換

# 各フェーズで使われるデータ構造と変換アルゴリズム

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

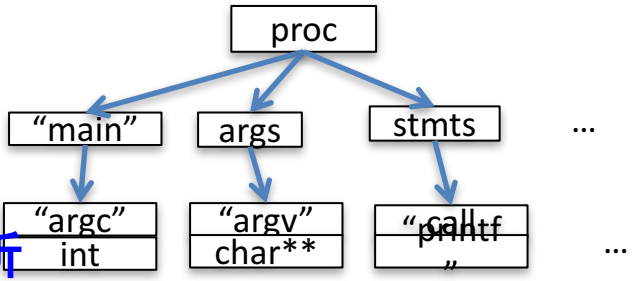
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ生成

中間命令列

最適化

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブル

アセンブリ

最適化

```
0000000   facf   feed   0007   0100   0003   8000   0002   0000
0000020   0010   0000   05b0   0000   0085   0020   0000   0000
0000040   0019   0000   0048   0000   5f5f   4150   4547   455a
0000060   4f52   0000   0000   0000   0000   0000   0000   0000
0000100   0000   0000   0001   0000   0000   0000   0000   0000
0000120   0000   0000   0000   0000   0000   0000   0000   0000
0000140   0000   0000   0000   0000   0019   0000   0228   0000
0000160   5f5f   4554   5458   0000   0000   0000   0000   0000
0000200   0000   0000   0001   0000   1000   0000   0000   0000
0000220   0000   0000   0000   0000   1000   0000   0000   0000
240   ……
```

機械語

# 各フェーズの説明

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

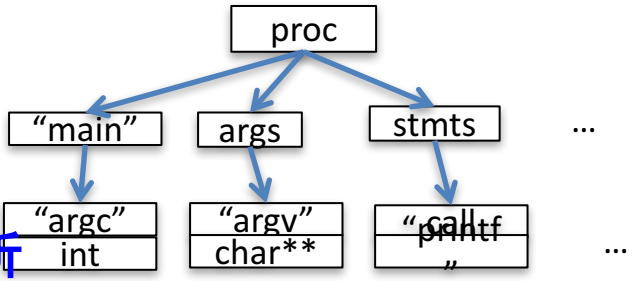
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ生成

中間命令列

最適化

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブル

アセンブリ

最適化

```
0000000  facf  feed  0007  0100  0003  8000  0002  0000
0000020  0010  0000  05b0  0000  0085  0020  0000  0000
0000040  0019  0000  0048  0000  5f5f  4150  4547  455a
0000060  4f52  0000  0000  0000  0000  0000  0000  0000
0000100  0000  0000  0001  0000  0000  0000  0000  0000
0000120  0000  0000  0000  0000  0000  0000  0000  0000
0000140  0000  0000  0000  0000  0019  0000  0228  0000
0000160  5f5f  4554  5458  0000  0000  0000  0000  0000
0000200  0000  0000  0001  0000  1000  0000  0000  0000
0000220  0000  0000  0000  0000  1000  0000  0000  0000
240    ……
```

機械語

# 字句解析

- バイト列であるテキストを「単語」(トークン) の列として切り出すフェーズ

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

トークン列

```
INT; ID(“main”); LPAREN; INT;
ID(“argc”); CHAR; AST; AST;
ID(“argv”); RPAREN; LBRACE; …
```

- 解析時にオートマトンや正則言語の理論が用いられる
  - どのようなバイト列が単語として認識されるべきか
  - バイト列を前からスキャンして単語を切り出す方法

# 字句解析に相当することは
# 自然言語を使うときにやっている

テキスト

We have a
green apple
on the table.

トークン列

We（指示詞）;
have（動詞）;
a（冠詞）;
green（形容詞）;
apple（名詞）;
on（前置詞）;
the（冠詞）;
table（名詞）

# 各フェーズの説明

意味解析・
最適化

抽象構文木

プログラムテキスト

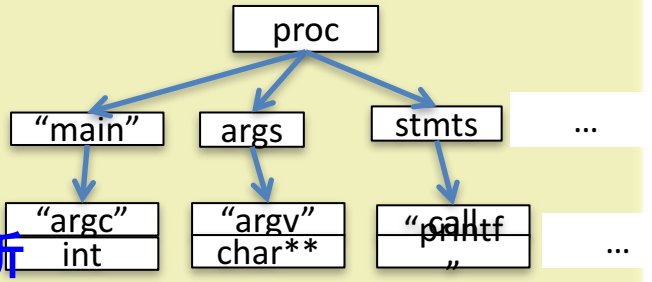
```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析




```
0000000    facf    feed    0007    0100    0003    8000    0002    0000
0000020    0010    0000    05b0    0000    0085    0020    0000    0000
0000040    0019    0000    0048    0000    5f5f    4150    4547    455a
0000060    4f52    0000    0000    0000    0000    0000    0000    0000
0000100    0000    0000    0001    0000    0000    0000    0000    0000
0000120    0000    0000    0000    0000    0000    0000    0000    0000
0000140    0000    0000    0000    0000    0019    0000    0228    0000
0000160    5f5f    4554    5458    0000    0000    0000    0000    0000
0000200    0000    0000    0001    0000    1000    0000    0000    0000
0000220    0000    0000    0000    0000    1000    0000    0000    0000
240    ……
```

機械語

アセンブル

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

アセンブリ
生成

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

最適化

最適化

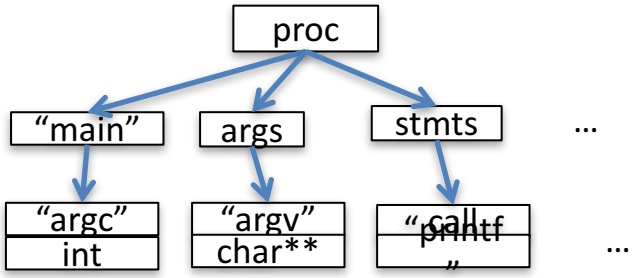
# 構文解析フェーズ

- トークン列を文法構造にしたがって木構造で表現し直すフェーズ

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

抽象構文木



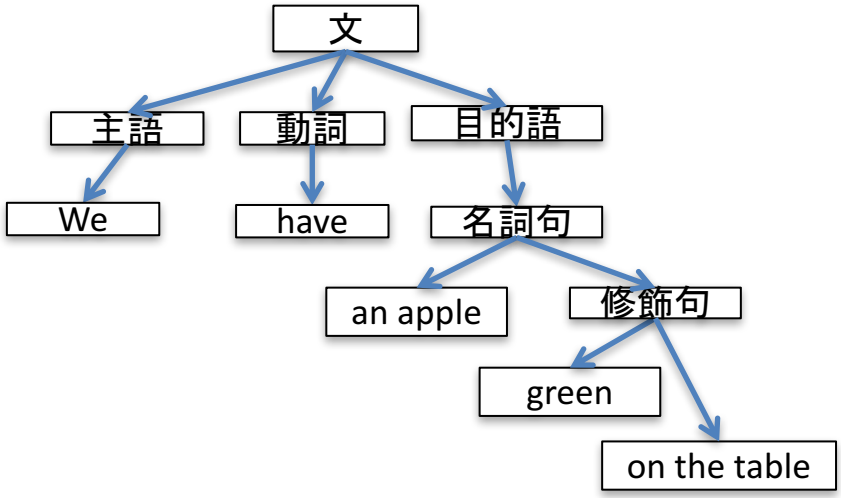

- 解析時に文脈自由言語の理論をベースにした手法が用いられる
  - 文法の定義の仕方
  - 文法構造の見つけ方

# 構文解析に相当することも
# 自然言語を使うときにやっている

トークン列

```
We （指示詞）;
have （動詞）;
a （冠詞）;
green （形容詞）;
apple （名詞）;
on （前置詞）;
the （冠詞）;
table （名詞）
```

抽象構文木



# 各フェーズの説明

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

字句解析

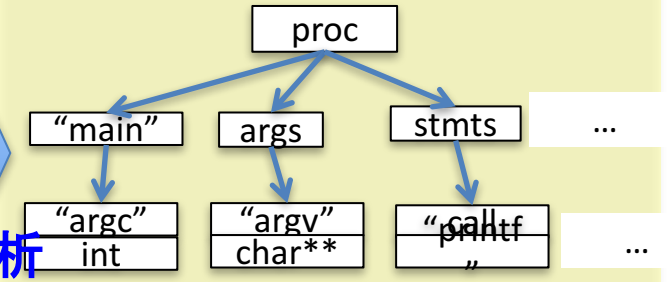
トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

意味解析・最適化

抽象構文木




```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ生成

中間命令列

最適化

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブル

アセンブリ

最適化

```
0000000  facf  feed  0007  0100  0003  8000  0002  0000
0000020  0010  0000  05b0  0000  0085  0020  0000  0000
0000040  0019  0000  0048  0000  5f5f  4150  4547  455a
0000060  4f52  0000  0000  0000  0000  0000  0000  0000
0000100  0000  0000  0001  0000  0000  0000  0000  0000
0000120  0000  0000  0000  0000  0000  0000  0000  0000
0000140  0000  0000  0000  0000  0019  0000  0228  0000
0000160  5f5f  4554  5458  0000  0000  0000  0000  0000
0000200  0000  0000  0001  0000  1000  0000  0000  0000
0000220  0000  0000  0000  0000  1000  0000  0000  0000
240   ……
```

機械語

# 意味解析

- プログラムの意味論に従って必要な情報を収集するフェーズ
  - 型検査や型推論等
- 詳しくは後日

# 最適化

- プログラムの実行効率を上げるための変換をほどこすフェーズ
  - 無駄な代入文を除去したり
  - 定数だとわかりきっている変数を展開したり
  - よく使う変数を高速にアクセスできる記憶領域に割り当てたり
- 抽象構文木での最適化，中間命令列での最適化等何度も行われることが多い
- 詳しくは後日

# 各フェーズの説明

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

字句解析

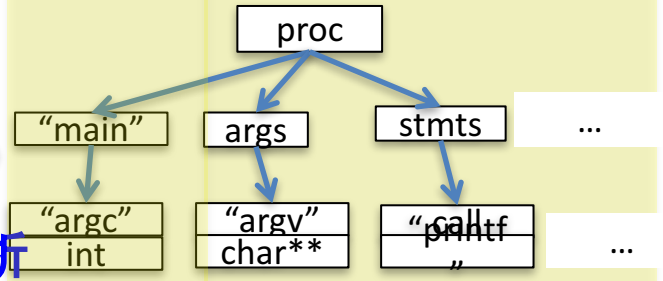
トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

意味解析・最適化

抽象構文木




```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

最適化

アセンブリ生成

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

最適化

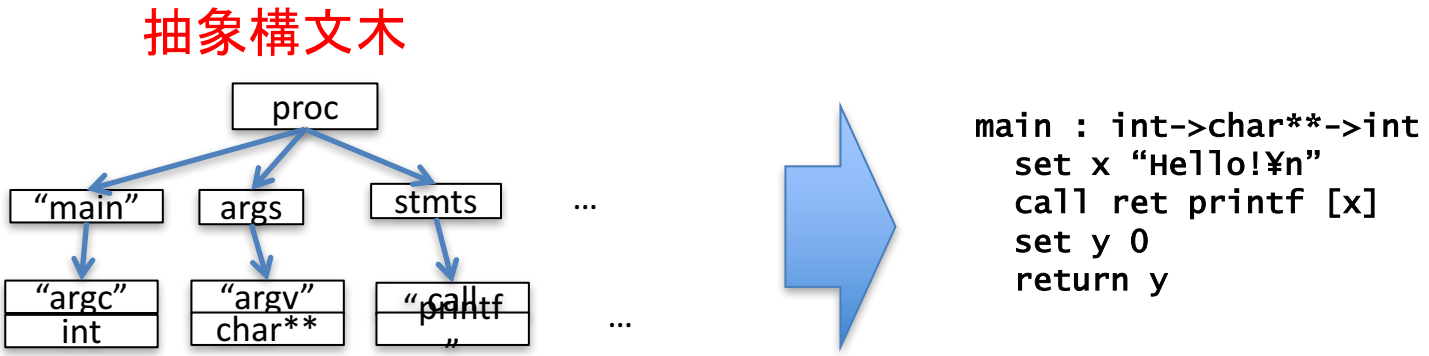
アセンブル

```
0000000  facf  feed  0007  0100  0003  8000  0002  0000
0000020  0010  0000  05b0  0000  0085  0020  0000  0000
0000040  0019  0000  0048  0000  5f5f  4150  4547  455a
0000060  4f52  0000  0000  0000  0000  0000  0000  0000
0000100  0000  0000  0001  0000  0000  0000  0000  0000
0000120  0000  0000  0000  0000  0000  0000  0000  0000
0000140  0000  0000  0000  0000  0019  0000  0228  0000
0000160  5f5f  4554  5458  0000  0000  0000  0000  0000
0000200  0000  0000  0001  0000  1000  0000  0000  0000
0000220  0000  0000  0000  0000  1000  0000  0000  0000
240  ……
```

機械語

# 中間命令生成

- 機械語に少し近い言語 (中間言語) への変換




- 中間言語の設計は目的に合わせて変わる

# 各フェーズの説明

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

字句解析

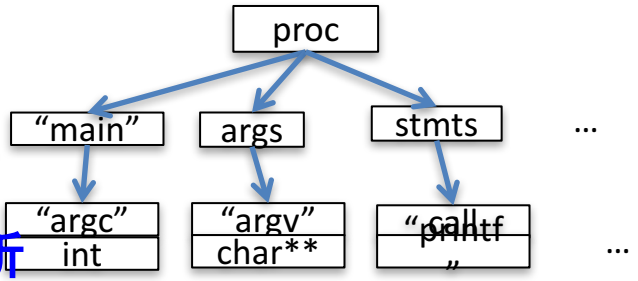
トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

意味解析・最適化

抽象構文木




アセンブリ生成

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

最適化

アセンブル

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

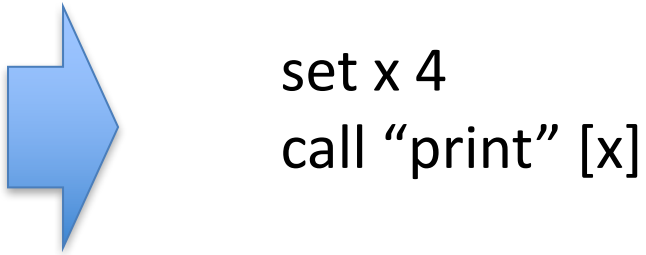
最適化

機械語

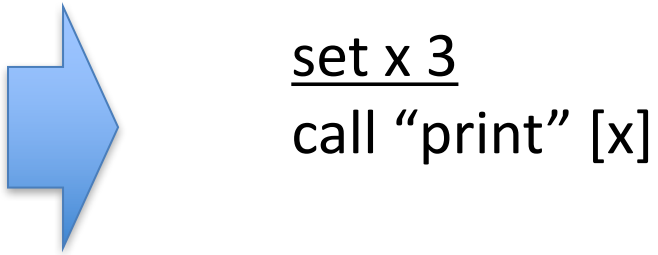
```
0000000  facf  feed  0007  0100  0003  8000  0002  0000
0000020  0010  0000  05b0  0000  0085  0020  0000  0000
0000040  0019  0000  0048  0000  5f5f  4150  4547  455a
0000060  4f52  0000  0000  0000  0000  0000  0000  0000
0000100  0000  0000  0001  0000  0000  0000  0000  0000
0000120  0000  0000  0000  0000  0000  0000  0000  0000
0000140  0000  0000  0000  0000  0019  0000  0228  0000
0000160  5f5f  4554  5458  0000  0000  0000  0000  0000
0000200  0000  0000  0001  0000  1000  0000  0000  0000
0000220  0000  0000  0000  0000  1000  0000  0000  0000
240   ……
```

# 中間命令レベルでの最適化

- いろいろあります
  - 無駄な命令の除去




  - 無駄な代入文の除去




- 詳細は後日

# 各フェーズの説明

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

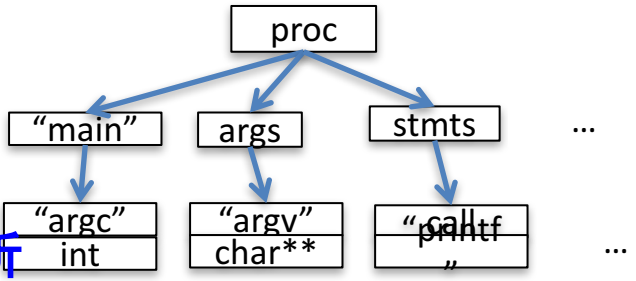
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

中間命令列

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ生成

```
  .data
LO:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,LO
  sw   $t0,0($sp)
  …
```

アセンブリ

最適化

アセンブル

```
0000000   facf   feed   0007   0100   0003   8000   0002   0000
0000020   0010   0000   05b0   0000   0085   0020   0000   0000
0000040   0019   0000   0048   0000   5f5f   4150   4547   455a
0000060   4f52   0000   0000   0000   0000   0000   0000   0000
0000100   0000   0000   0001   0000   0000   0000   0000   0000
0000120   0000   0000   0000   0000   0000   0000   0000   0000
0000140   0000   0000   0000   0000   0019   0000   0228   0000
0000160   5f5f   4554   5458   0000   0000   0000   0000   0000
0000200   0000   0000   0001   0000   1000   0000   0000   0000
0000220   0000   0000   0000   0000   1000   0000   0000   0000
240   ……
```

機械語

最適化

# アセンブリ生成

中間命令列

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

# アセンブリ言語とは?

- コンピュータが扱える機械語を人間が読みやすいように文字列で表現したもの

```
0000000    facf    feed    0007    0100    0003    8000    0002    0000
0000020    0010    0000    05b0    0000    0085    0020    0000    0000
0000040    0019    0000    0048    0000    5f5f    4150    4547    455a
0000060    4f52    0000    0000    0000    0000    0000    0000    0000
0000100    0000    0000    0001    0000    0000    0000    0000    0000
0000120    0000    0000    0000    0000    0000    0000    0000    0000
0000140    0000    0000    0000    0000    0019    0000    0228    0000
0000160    5f5f    4554    5458    0000    0000    0000    0000    0000
0000200    0000    0000    0001    0000    1000    0000    0000    0000
0000220    0000    0000    0000    0000    1000    0000    0000    0000
240 ……
```

は

アセンブリ言語では

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

と表現

# アセンブリ生成

- 中間命令列をアセンブリ言語に変換するフェーズ

中間命令列

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

- それぞれの中間命令をどの機械命令を使えば効率よく実現できるか等の難しさ

# 各フェーズの説明

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

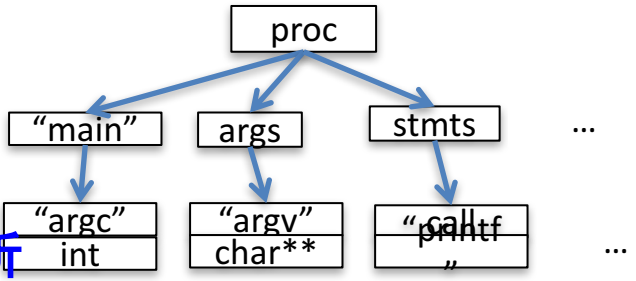
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

アセンブリ生成

中間命令列

最適化

```
  .data
LO:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,LO
  sw   $t0,0($sp)
  …
```

アセンブル

アセンブリ

最適化

```
0000000  facf  feed  0007  0100  0003  8000  0002  0000
0000020  0010  0000  05b0  0000  0085  0020  0000  0000
0000040  0019  0000  0048  0000  5f5f  4150  4547  455a
0000060  4f52  0000  0000  0000  0000  0000  0000  0000
0000100  0000  0000  0001  0000  0000  0000  0000  0000
0000120  0000  0000  0000  0000  0000  0000  0000  0000
0000140  0000  0000  0000  0000  0019  0000  0228  0000
0000160  5f5f  4554  5458  0000  0000  0000  0000  0000
0000200  0000  0000  0001  0000  1000  0000  0000  0000
0000220  0000  0000  0000  0000  1000  0000  0000  0000
240  ……
```

機械語

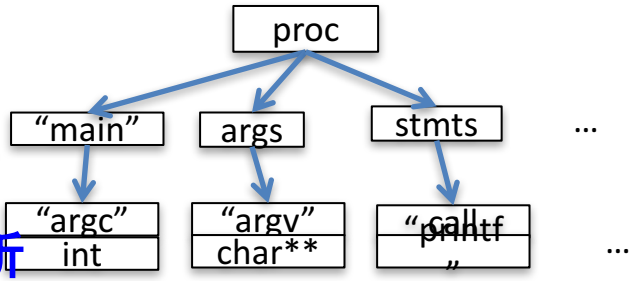
# この講義の進み方

# この順番でやります

4

意味解析・
最適化

抽象構文木




プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

字句解析

5

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

6

中間命令
生成

1

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

3

最適化

アセンブリ
生成

2

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

最適化

アセンブル

```
0000000   facf   feed   0007   0100   0003   8000   0002   0000
0000020   0010   0000   05b0   0000   0085   0020   0000   0000
0000040   0019   0000   0048   0000   5f5f   4150   4547   455a
0000060   4f52   0000   0000   0000   0000   0000   0000   0000
0000100   0000   0000   0001   0000   0000   0000   0000   0000
0000120   0000   0000   0000   0000   0000   0000   0000   0000
0000140   0000   0000   0000   0000   0019   0000   0228   0000
0000160   5f5f   4554   5458   0000   0000   0000   0000   0000
0000200   0000   0000   0001   0000   1000   0000   0000   0000
0000220   0000   0000   0000   0000   1000   0000   0000   0000
240   ……
```

機械語

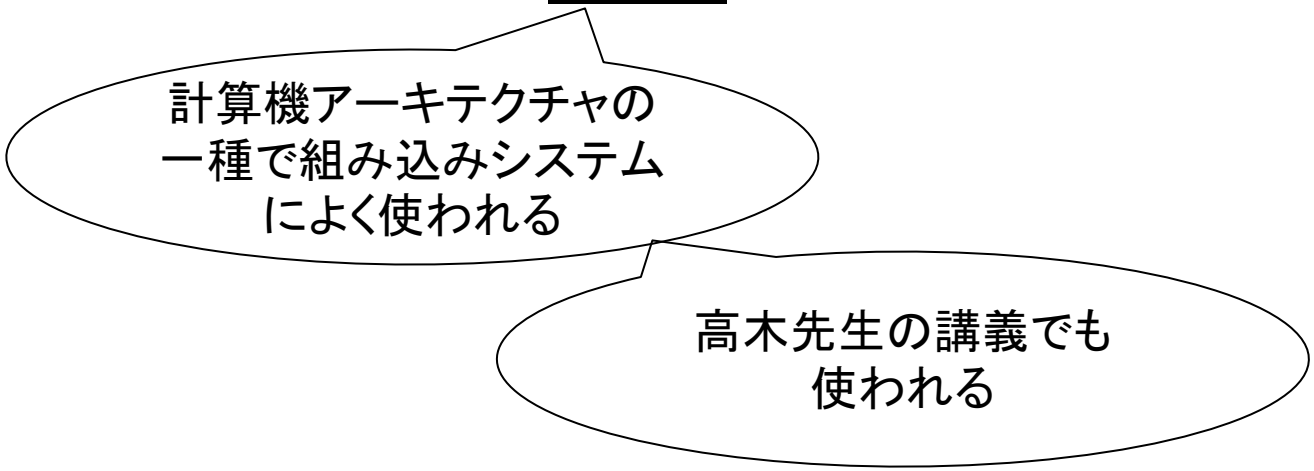
# なぜ中間命令生成からやるか

- 字句解析・構文解析はややとっつきにくい
  - 「言語・オートマトン」で学ぶ正則言語や文脈自由言語の知識が必要
  - 使用するアルゴリズムもやや大変
- 中間命令生成以降の話は結構つぶしがきく
  - やる気のあるうちに役に立つ話をやっちまおう

# この講義で作るコンパイラ

# この講義で作るコンパイラ

- ソース言語: C のサブセット (後述)
- ターゲット言語: MIPS アセンブリ言語



# アセンブリ言語にまつわる問題

- MIPS の命令列は MIPS アーキテクチャ以外では実行できない
  - 演習室のマシンや配布している MBA では実行できない

# じゃあこの講義で作るコンパイラが生成するコードは実行できないの?

- 否! MIPS の動作をシミュレートするシミュレータを用いれば動く!

# SPIM シミュレータ

- James Larus が作成している
  MIPS シミュレータ
- http://spimsimulator.sourceforge.net/
- Windows, MacOS, Linux で動作
- QtSpim という GUI もある
- デモ

# ここまでのまとめ

- コンパイラは高級言語プログラムを低級言語プログラムに変換するソフトウェアである
- コンパイラは間にいくつかフェーズを挟みながら変換を進める
- この講義では C のサブセットから MIPS アセンブリを生成するコンパイラを作る
- この講義は中間命令生成から解説をする

# ソース言語の定義

「言語を定義する」とは
この講義で扱うソース言語の定義

# ここを決めよう

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

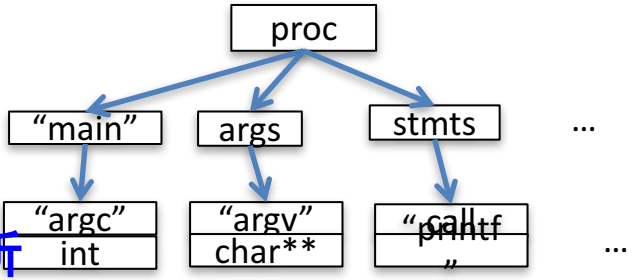
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

アセンブリ生成

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

最適化

アセンブル

```
0000000   facf   feed   0007   0100   0003   8000   0002   0000
0000020   0010   0000   05b0   0000   0085   0020   0000   0000
0000040   0019   0000   0048   0000   5f5f   4150   4547   455a
0000060   4f52   0000   0000   0000   0000   0000   0000   0000
0000100   0000   0000   0001   0000   0000   0000   0000   0000
0000120   0000   0000   0000   0000   0000   0000   0000   0000
0000140   0000   0000   0000   0000   0019   0000   0228   0000
0000160   5f5f   4554   5458   0000   0000   0000   0000   0000
0000200   0000   0000   0001   0000   1000   0000   0000   0000
0000220   0000   0000   0000   0000   1000   0000   0000   0000
240   ……
```

機械語

最適化

# この講義のソース言語

- Tiny-C
  - C言語のサブセット
  - 変数への代入，ポインタ，条件分岐，ループ，関数呼び出し，関数定義
  - 配列，構造体，メモリの動的確保は扱わない予定
    - 実験3SWでやるかも

# ちょっとまった!

- Tiny-C
  - C言語のサブセット
  - 変数への代入，ポインタ，条件分岐，ループ，関数呼び出し，関数定義
  - 配列，構造体，メモリの動的確保は扱わない予定
    - 実験3SWでやるかも

**これで「Tiny-C は定義できた」と言える?**

# プログラミング言語を定義するとはどういうことか

**プログラムの文法**

- 構文 (syntax) の定義
- 意味論 (semantics) の定義

**(文法的に正しい)**
**プログラムの意味**

# 構文の定義

- どのようなプログラムが文法的に正しいかを決めること

✔
```
int f(int x) {
  int y = 0;
  y = x + 1;
  printf(“%d¥n”, y);
  return y;
}
```

✗
```
int f<int #) {
  int y (^^) 0;
  y = x (`Δ’) 1;
  printf “%d¥n” y
  りたーん y;
]
```

# 意味論の定義

- (文法的に正しい) プログラムの動作を決めること

```
int f(int x) {
  int y = 0;
  y = x + 1;
  printf(“%d¥n”, y);
  return y;
}
```

プログラム

f は整数型の引数 x を受け取り，局所変数 y に x + 1 を代入し，x の値と改行文字を表示して y を返す関数である

動作

# 構文を定義する方法

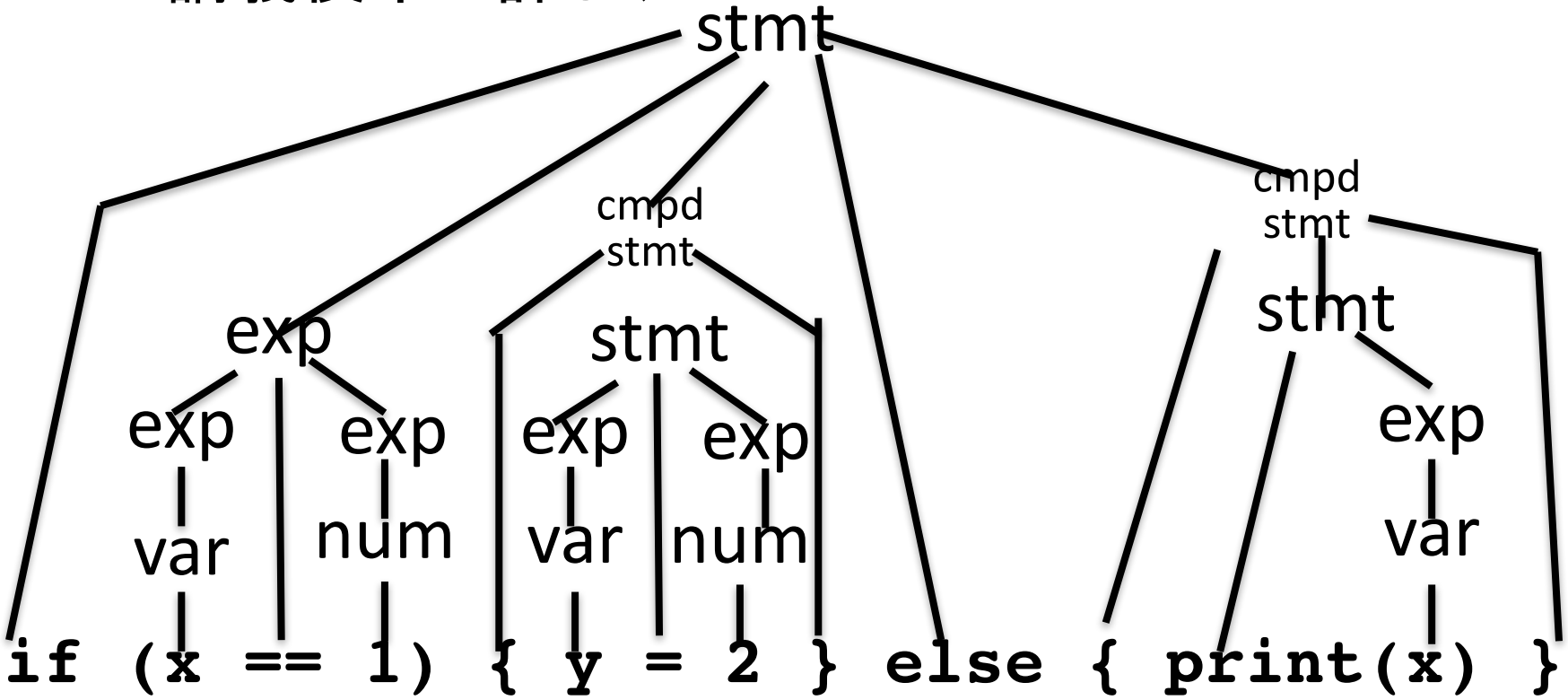
- BNF (Backus-Naur form)
  - 正しいプログラムを生成する文脈自由文法を指定する方法
  - 今回は感じをつかむだけで OK
    - 詳しくは「言語・オートマトン」と本講義終盤の構文解析の解説で

# BNF による Tiny-C 構文の定義

- 別紙参照
  - <statement> や <exp> などの「木」を表す記号を非終端記号と言う
  - true や false などの「葉」を表す記号を終端記号と言う

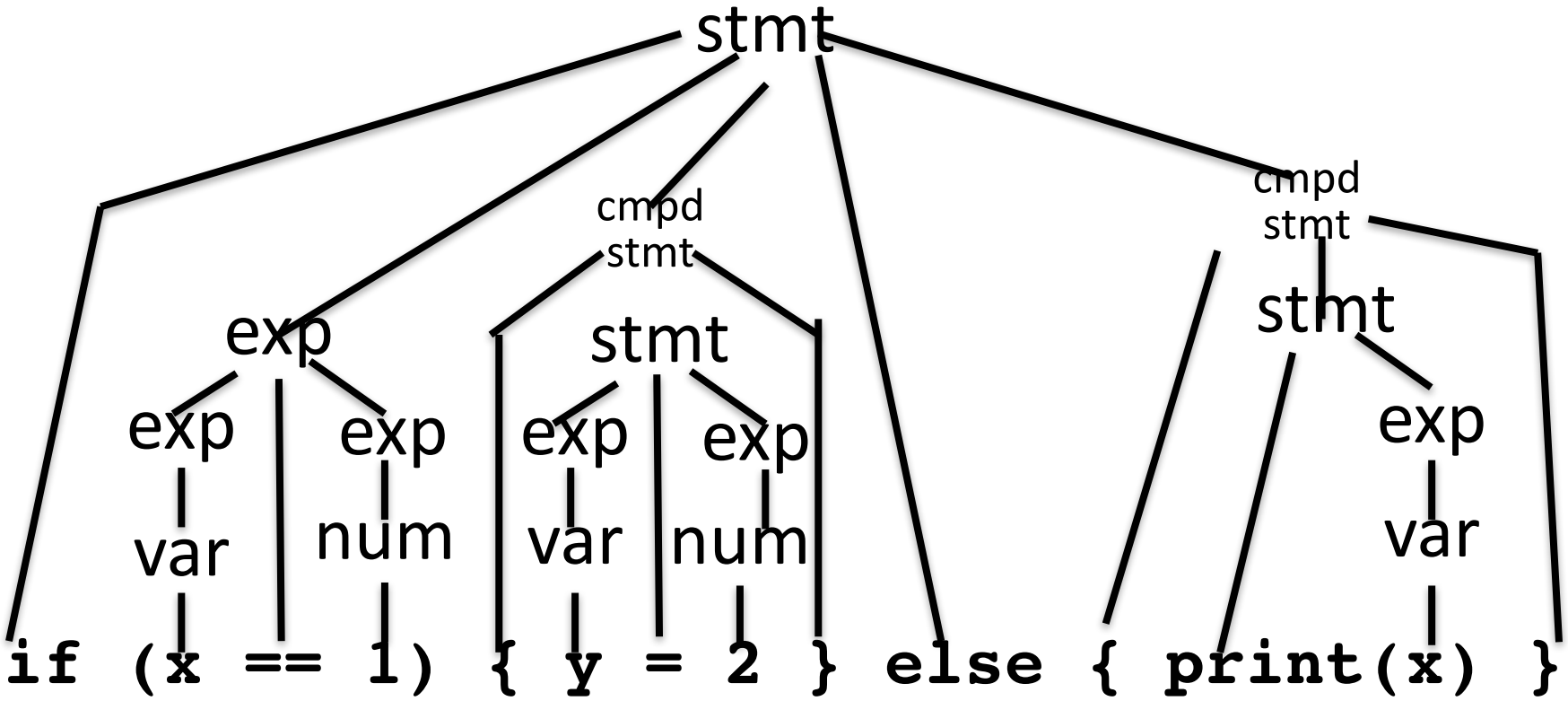
# 構文解析とは

- プログラムが文法でどのように導出されているかを解析する方法
  - 講義後半で詳しく



# 抽象構文木とは

- 導出過程を表現した木
  - 構文解析はこの木を出力する



この講義ではこっち

# 意味論を定義する方法

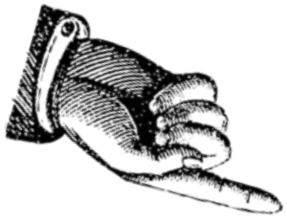
- 自然言語でプログラムの動きを説明
  - 例: 文 x = *exp は exp を評価して，その結果をメモリアドレスとしてメモリから値を読み，その値を x に代入する
  - 分かりやすいが曖昧なことが多い
    - 評価ってなに? メモリってなに? アドレスってなに? 読むってなに? x ってなに? 代入ってなに?
- 数学でプログラムの動きを定義
  - 例: (長くなるので略)
  - 曖昧さはなくなる (が，慣れてないと読めない)

末永の研究ではこっち

# 中間命令の定義

「言語を定義する」とは
この講義で扱うソース言語の定義

# ここは後日



プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

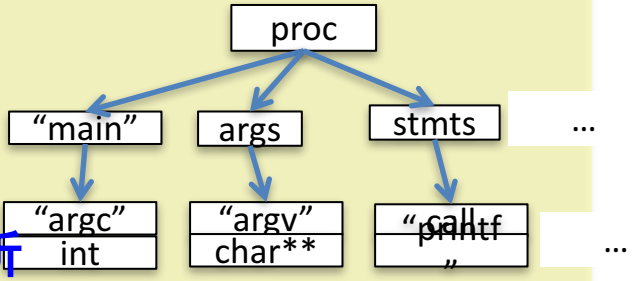
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

最適化

アセンブリ生成

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

最適化

アセンブル

```
0000000   facf   feed   0007   0100   0003   8000   0002   0000
0000020   0010   0000   05b0   0000   0085   0020   0000   0000
0000040   0019   0000   0048   0000   5f5f   4150   4547   455a
0000060   4f52   0000   0000   0000   0000   0000   0000   0000
0000100   0000   0000   0001   0000   0000   0000   0000   0000
0000120   0000   0000   0000   0000   0000   0000   0000   0000
0000140   0000   0000   0000   0000   0019   0000   0228   0000
0000160   5f5f   4554   5458   0000   0000   0000   0000   0000
0000200   0000   0000   0001   0000   1000   0000   0000   0000
0000220   0000   0000   0000   0000   1000   0000   0000   0000
240   ……
```

機械語

# ここを決めよう

プログラムテキスト

```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

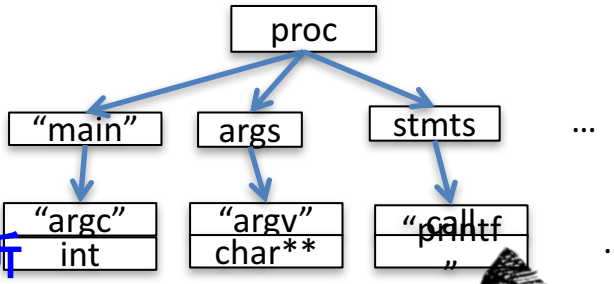
字句解析

トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析

抽象構文木




意味解析・最適化

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

最適化

アセンブリ生成

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ

最適化

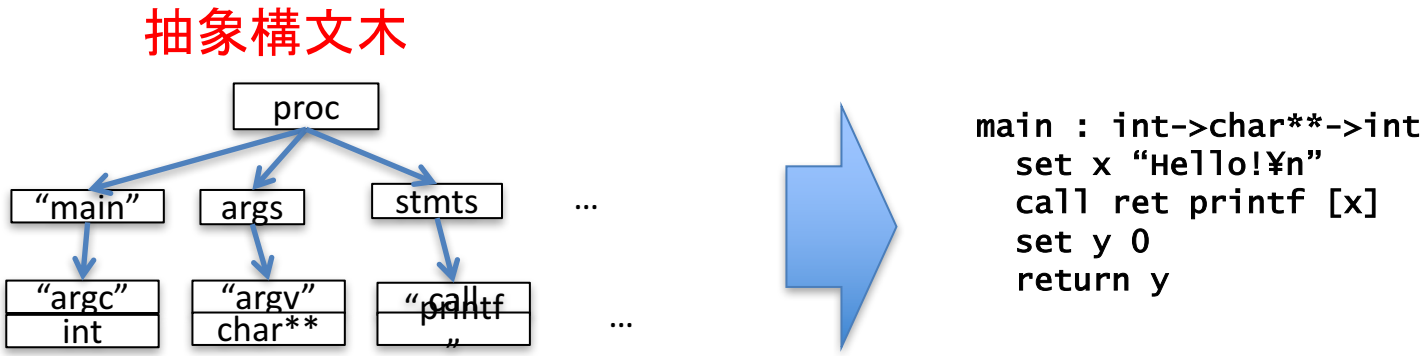
アセンブル

```
0000000   facf   feed   0007   0100   0003   8000   0002   0000
0000020   0010   0000   05b0   0000   0085   0020   0000   0000
0000040   0019   0000   0048   0000   5f5f   4150   4547   455a
0000060   4f52   0000   0000   0000   0000   0000   0000   0000
0000100   0000   0000   0001   0000   0000   0000   0000   0000
0000120   0000   0000   0000   0000   0000   0000   0000   0000
0000140   0000   0000   0000   0000   0019   0000   0228   0000
0000160   5f5f   4554   5458   0000   0000   0000   0000   0000
0000200   0000   0000   0001   0000   1000   0000   0000   0000
0000220   0000   0000   0000   0000   1000   0000   0000   0000
240   ……
```

機械語

# 中間命令とは (前回の復習)

- 機械語に少し近い命令セット



# この講義で使う中間命令の定義

- 別紙参照

# なぜそういう設計になっているか

- CPU はあまり複雑な処理を一度にできない
  - x = (y + z) * (x + 2): y + z を計算して，x + 2 を計算して，両方の結果を掛けて，x に格納する必要

# Tiny-C 抽象構文木から中間命令への変換

構文主導翻訳

# やるべきこと

- 抽象構文木を受け取って等価な中間命令列を返す関数を実装

**x = 3 + (y * 7)** ➔

**t1 = y**
**t2 = 7**
**t3 = t1 * t2**
**t4 = 3**
**t5 = t4 + t3**
**x = t5**

# やるべきこと

- 抽象構文木を受け取って等価な中間命令列を返す関数を実装

***x = *p + *q** ➔

**t1 = p**
**t2 = *t1**
**t3 = q**
**t4 = *t3**
**t5 = t2 + t4**
***x = t5**

# このような関数を実装するのに必要なこと

- 抽象構文木のデータ型の定義
- 中間命令 (列) のデータ型の定義
- 変換アルゴリズムの定義

- ここから先は Racket で説明
  - 勉強してきたね?
  - でも復習するね

# Racket の構造体

- ラベルがついたデータの組
  - struct キーワードで定義
    - (struct posn (x y)): ラベル x とラベル y を持つ posn という名前の構造体
  - 定義した構造体の型を持つ値の作り方
    - (posn 2 3): ラベル x を 2 に, ラベル y を 3 とする posn 構造体
  - 構造体に名前をつけられる
    - (define p (posn 2 3)): 構造体 (posn 2 3) に p という名前をつける
      - 値 v に名前 x をつけることを一般に「x を v に束縛する」という

# Racket の構造体

- ラベルがついたデータの組
  - 構造体から値を取り出す方法: セレクタ
    - (posn-x p): posn 構造体 p の x フィールドを取り出す
    - (posn-y p): posn 構造体 p の y フィールドを取り出す

posn 構造体を定義したときに posn-x と posn-y は自動的に作られる

  - 実行時型検査
    - (posn? p): p が posn 構造体であるならば #t

posn? は posn 構造体を定義した時に自動的に作られる

# 抽象構文木のデータ構造

- syntax.rkt を見てください
  - 式や文それぞれの場合について構造体を一つ定義
  - 構造体が作る木構造で抽象構文木を表現
    - 変数 a は (varexp ‘a)
    - 算術式 3 + 5 は (aopexp ‘plus (intexp 3) (intexp 5)) のように
  - プログラム全体は fundef 構造体のリストと main 文の組からなる構造体
    - fundef 構造体は後日使うので今は気にしないで OK

# 中間命令文のデータ構造

- intermedCode.rkt を見てください
  - 式や文それぞれの場合について構造体を一つ定義
  - おおむね抽象構文木と同様
  - 中間命令列は，中間命令文のリストで表現
    - (list (skipstmt) (writestmt (varexp ‘a) (varexp ‘b))) みたいに

# Tiny-C 文の抽象構文木から中間命令列への変換

- <stmt> を “<stmt> に相当する動作” をする命令列に
  - 構文が再帰的に定義されているので, 変換を再帰関数で自然に表現可能
    - 正確な定義は別紙参照
  - <var> = <exp> 等のケースで次ページの「式の変換」を使用

# Tiny-C 式の抽象構文木から中間命令列への変換

- <exp> を “<exp> を評価して x に代入する” という動作をする命令列に
  - x は前ページの「文の変換」中に与えられる
  - 正確な定義は別紙参照

# プログラム変換 (program transformation)

- プログラムを別のプログラムに変換するアルゴリズムのこと
  - 例: アルゴリズム I は Tiny-C プログラムを中間命令という別のプログラムに変換
- コンパイラはプログラム変換を何度も繰り返して高級言語プログラムを低級言語に変換
- 変化の前後の等価性が問題になることが多い

# 構文主導翻訳 (syntax-directed translation)

- 文法規則に付随する形で変換を与えるプログラム変換の方法
  - 例: <stmt> ::= <var> = <exp> という規則にそって I(<var>=<exp>) の定義が与えられている
  - 例: <exp> ::= $\langle exp_1 \rangle$ <aop> $\langle exp_2 \rangle$ という規則にそって Ix($\langle exp_1 \rangle$ <aop> $\langle exp_2 \rangle$) の定義が与えられている

- 再帰関数や属性文法 (attribute grammar) で定義される

# MIPS アセンブリプログラミング入門

コンパイラ作成に必要な知識だけ

# MIPS とは

- 計算機アーキテクチャの一種
- 主に組み込みで使われる
  - 組み込み: 家電等に「組み込んで」使われる計算機のこと

# ここからやること

- コンパイラを書く前に，まず自分で MIPS アセンブリを書けるようになる
  - http://pages.cs.wisc.edu/~larus/HP_AppA.pdf の A.1, A.2, A.5, A.6, A.10 あたりを読めば独学できる
  - ヘネパタの和訳本にも対応する章があるが，英語の勉強だと思って読むと良い
  - リファレンスの読み方を解説するので命令等は自分で調べてほしい

# 単語集

- Register: レジスタ．CPU 中の記憶領域．アセンブリ中では $ で始まる記号で表す．
- Immediate: 即値．定数のこと．
- Load: ロード．レジスタに値を読み込むこと．
- Store: ストア．レジスタからメモリに値を書き込むこと

# 最初の例: 計算と出力

- example01.s というファイルを作って，以下のコードを入力して QtSpim で実行してください

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $v0,1
        li      $a0,20
        syscall
        addiu   $sp,$sp,20
        jr      $ra
```

# 解説

```
    .text
    .globl    main
main:
    addiu     $sp,$sp,-20
    li        $v0,1
    li        $a0,20
    syscall
    addiu     $sp,$sp,20
    jr        $ra
```

「ここからプログラムが書いてあるよ!」と
アセンブラに伝える text ディレクティブ

# 解説

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $v0,1
    li      $a0,20
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

「main はグローバル関数のラベルだよ」
とアセンブラに伝える globl ディレクティブ

# 解説

```
    .text
    .globl    main
main:
    addiu     $sp,$sp,-20
    li        $v0,1
    li        $a0,20
    syscall
    addiu     $sp,$sp,20
    jr        $ra
```

main ラベル: プログラムのこの場所を main という名前にする

# 解説

```
    .text
    .globl    main
main:
    addiu     $sp,$sp,-20
    li        $v0,1
    li        $a0,20
    syscall
    addiu     $sp,$sp,20
    jr        $ra
```

**addiu 命令を $sp, $sp, -20 という引数で実行**

- sp レジスタの値と -20 の和を sp レジスタに
  格納
  - ヘネパタの PDF A-51 ページ参照

# 解説

```
    .text
    .globl    main
main:
    addiu     $sp,$sp,-20
    li        $v0,1
    li        $a0,20
    syscall
    addiu     $sp,$sp,20
    jr        $ra
```

li: 即値をレジスタに
読み込む命令

# 解説

```
    .text
    .globl    main
main:
    addiu     $sp,$sp,-20
    li        $v0,1
    li        $a0,20
    syscall
    addiu     $sp,$sp,20
    jr        $ra
```

syscall: システムコールを実行する命令

# システムコール

- 入出力等を OS に依頼するために出す割り込みのこと
  - 割り込みや例外やシステムコールの詳細は他講義で
- とりあえずこの講義では「$v0 に 1 をロードして syscall 命令を実行すると $a0 の値が整数としてコンソールに出力される」という理解で OK
  - syscall 命令の詳しい説明は A-43 参照

# 解説

```
    .text
    .globl    main
main:
    addiu     $sp,$sp,-20
    li        $v0,1
    li        $a0,20
    syscall
    addiu     $sp,$sp,20
    jr        $ra
```

jr: $ra の表す命令アドレスにリターン (後述)

# 命令リファレンス (英語版) の構成

- A-49 の “Encoding MIPS Instructions” のところにカテゴリ別に命令が解説されている
  - 算術命令・論理命令・ビット操作命令 (A-51～)
  - 定数をロードする命令 (A-57～)
  - 比較命令 (A-57～)
  - 分岐命令 (A-59～)
  - 無条件ジャンプ命令 (A-63～)
  - トラップ命令 (A-64～)
    - この講義では不要
  - メモリ内容をレジスタにロードする命令 (A-66～)
  - レジスタ内容をメモリにストアする命令 (A-68～)
  - レジスタ間で値を転送する擬似命令 (A-70～)
  - 浮動小数命令 (A-73～)
    - この講義では不要
  - 割り込みとかする命令 (A-80～）
- 正直僕も全部は把握してないし，必要なところを必要に応じて読めば良い

# リファレンスを読むときに注意すること

- 引数名には意味がある
  - r で始まる引数: $ で始まるレジスタ名
  - imm: 定数
  - label, target: ラベル名
  - address: アドレス指定
    - とりあえず 4($sp) のように「定数(レジスタ名)」と思っておけば良い
    - 詳細は A-45 の “Addressing modes” 参照

# 2つ目の例: メモリアクセス

- example02.s というファイルを作って，以下のコードを入力して QtSpim で実行してください

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $t0,5
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        li      $v0,1
        move    $a0,$t1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
```

# 解説

```
    .text
    .globl main
main:
    addiu $sp,$sp,-20
    li    $t0,5
    sw    $t0, 4($sp)
    lw    $t1, 4($sp)
    li    $v0,1
    move $a0,$t1
    syscall
    addiu $sp,$sp,20
    jr    $ra
```

sw: $t0 の値を
メモリアドレス $sp + 4 に
書き込む

# 解説

```
    .text
    .globl main
main:
    addiu $sp,$sp,-20
    li    $t0,5
    sw    $t0, 4($sp)
    lw    $t1, 4($sp)
    li    $v0,1
    move $a0,$t1
    syscall
    addiu $sp,$sp,20
    jr    $ra
```

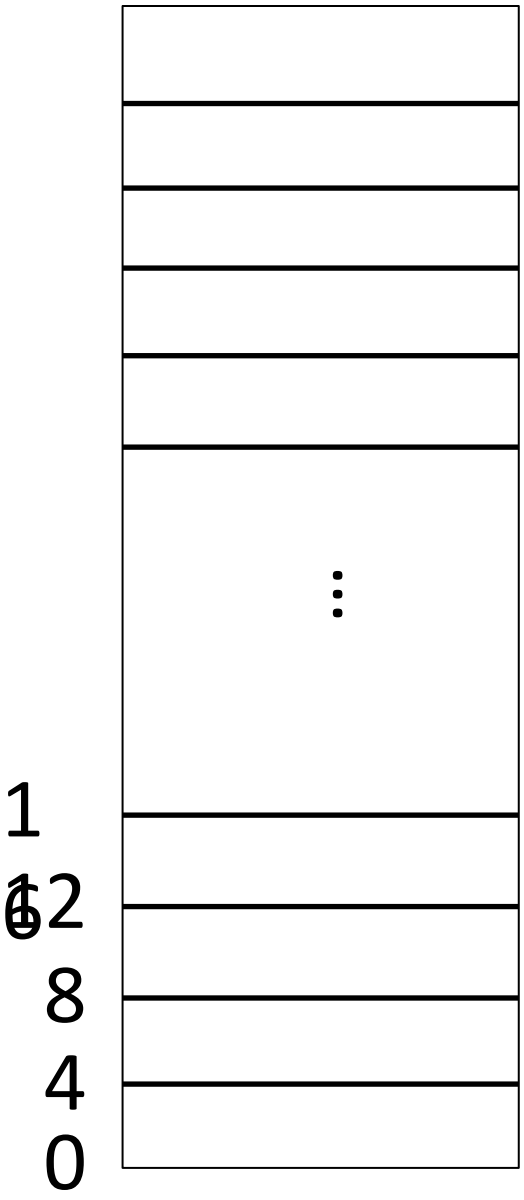
lw: メモリアドレス $sp + 4 の場所から $t1 に値を読み込む

# 解説

```
    .text
    .globl main
main:
    addiu $sp,$sp,-20
    li    $t0,5
    sw    $t0, 4($sp)
    lw    $t1, 4($sp)
    li    $v0,1
    move $a0,$t1
    syscall
    addiu $sp,$sp,20
    jr    $ra
```

move: $t1 の値を $a0 に書き込む

# メモリ




- コンピュータが持っている記憶領域
- 1バイトごとにメモリアドレスがついている
- 書き込み，読み込みの際はアドレスを指定する
- 絵で描くときは上がアドレスの大きい側になるように描く

# 今から $sp の説明をします

```
    .text
    .globl main
main:
    addiu $sp,$sp,-20
    li    $t0,5
    sw    $t0, 4($sp)
    lw    $t1, 4($sp)
    li    $v0,1
    move $a0,$t1
    syscall
    addiu $sp,$sp,20
    jr    $ra
```

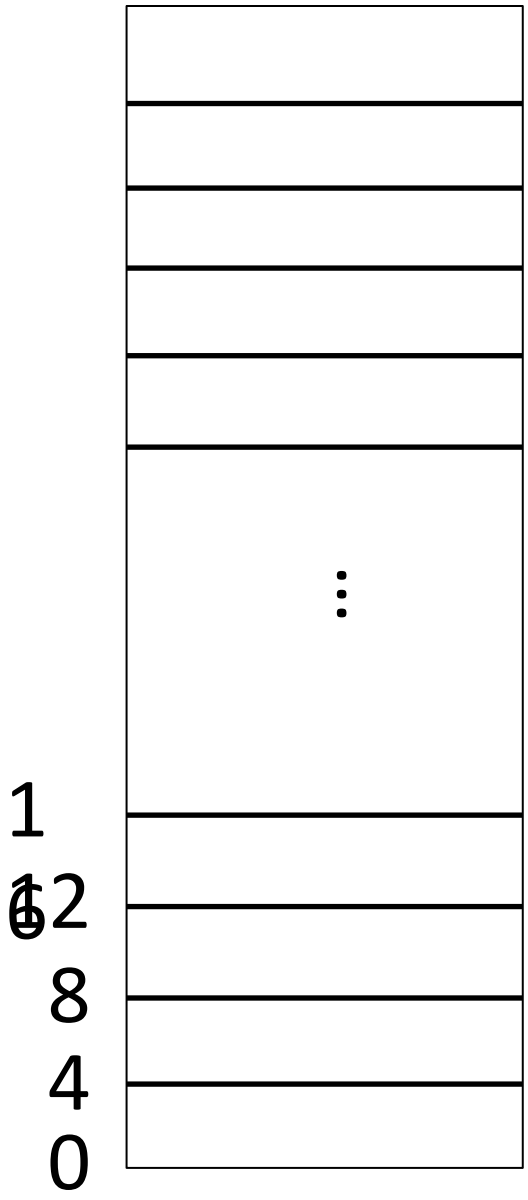
なぜ最初に $sp を -20 減らしているのか

なぜ $sp + 4 に値を書くのか

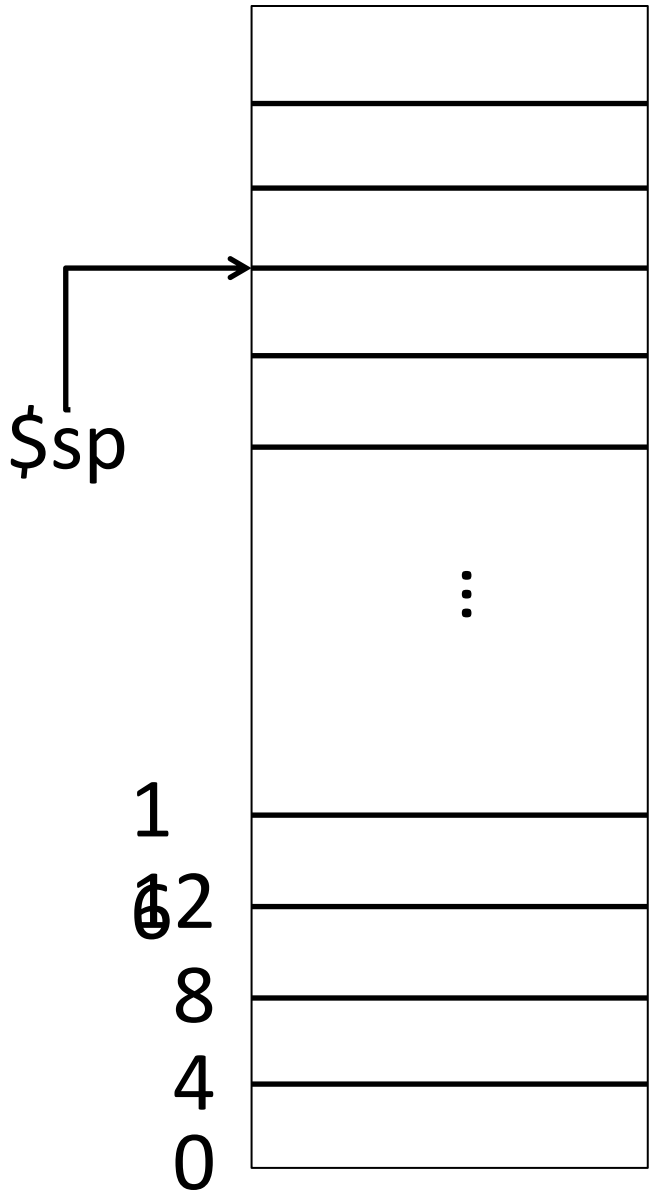
なぜ最後に $sp を 20 増やしているのか

# ローカル領域

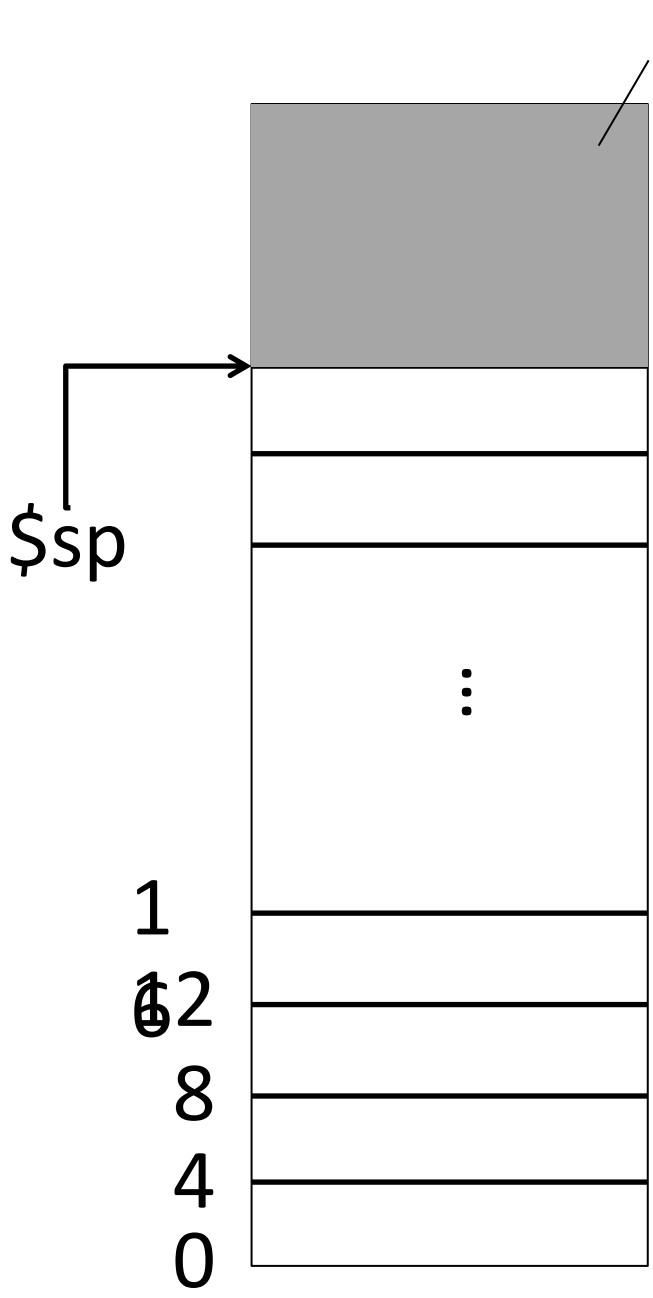
- 関数が自分の中だけで使える領域



# ローカル領域



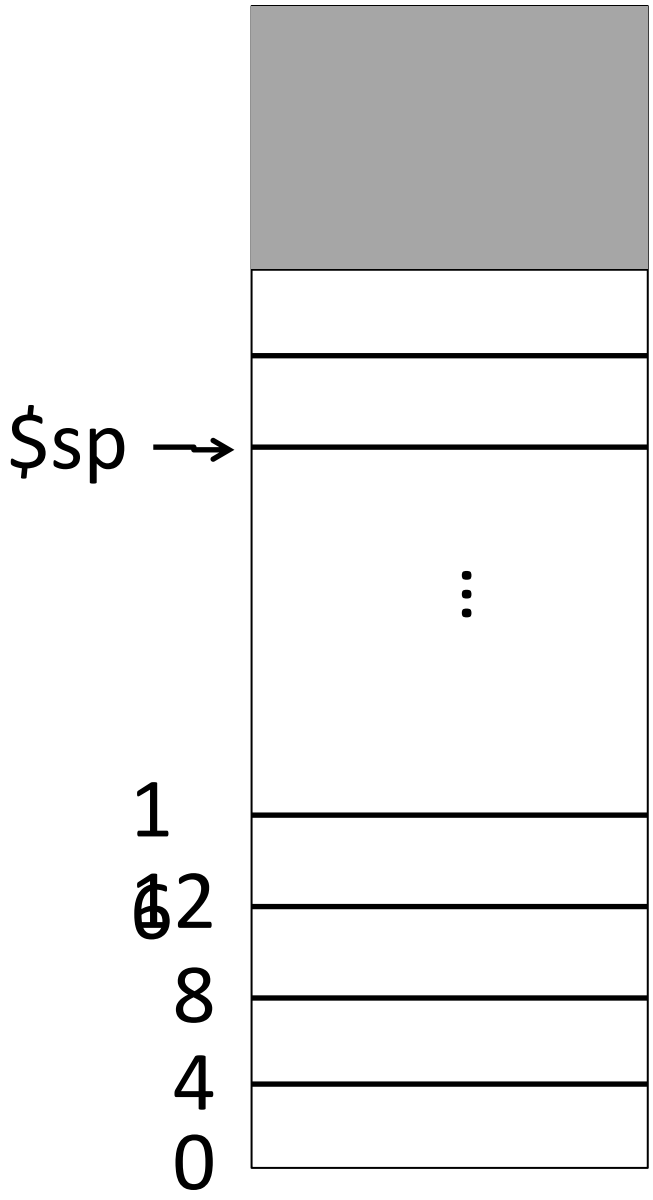

- 関数が自分の中だけで使える領域
- 使っていい領域の天井のアドレスが $sp に入っている




# 領域

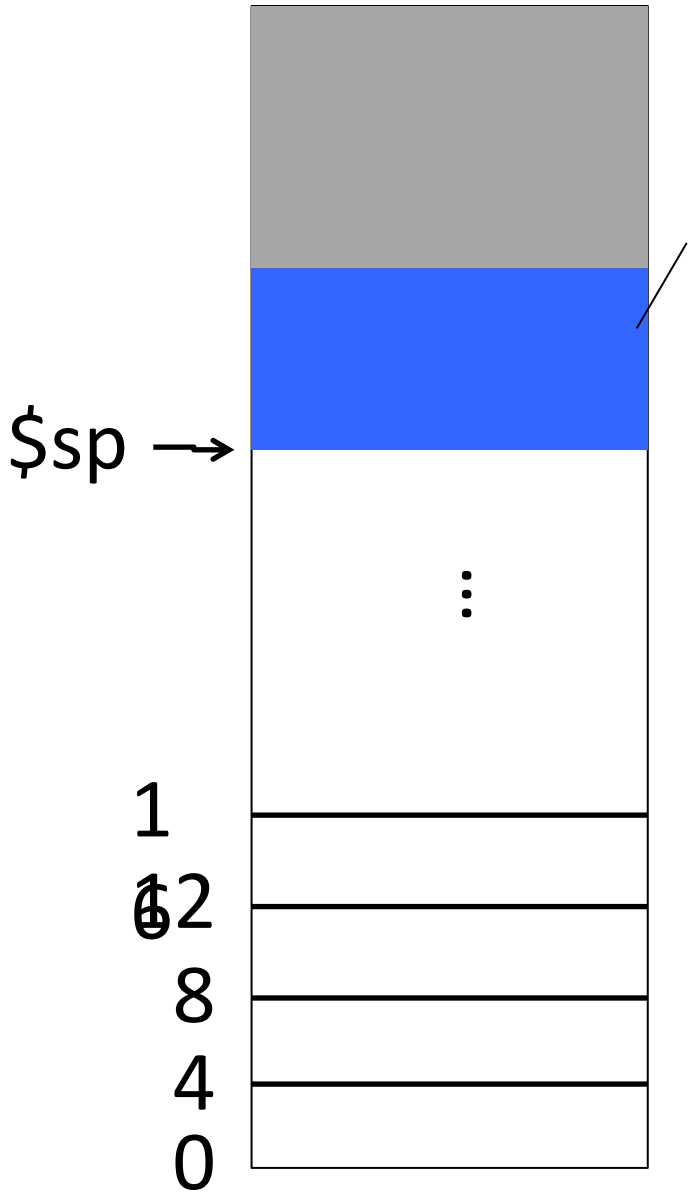
- 関数が自分の中だけで使える領域
- 使っていい領域の天井のアドレスが $sp に入っている

# ローカル領域




- 関数が自分の中だけで使える領域
- 使っていい領域の天井のアドレスが $sp に入っている
- 関数の頭で自分が使う分だけ $sp を減らす

# ローカル領域

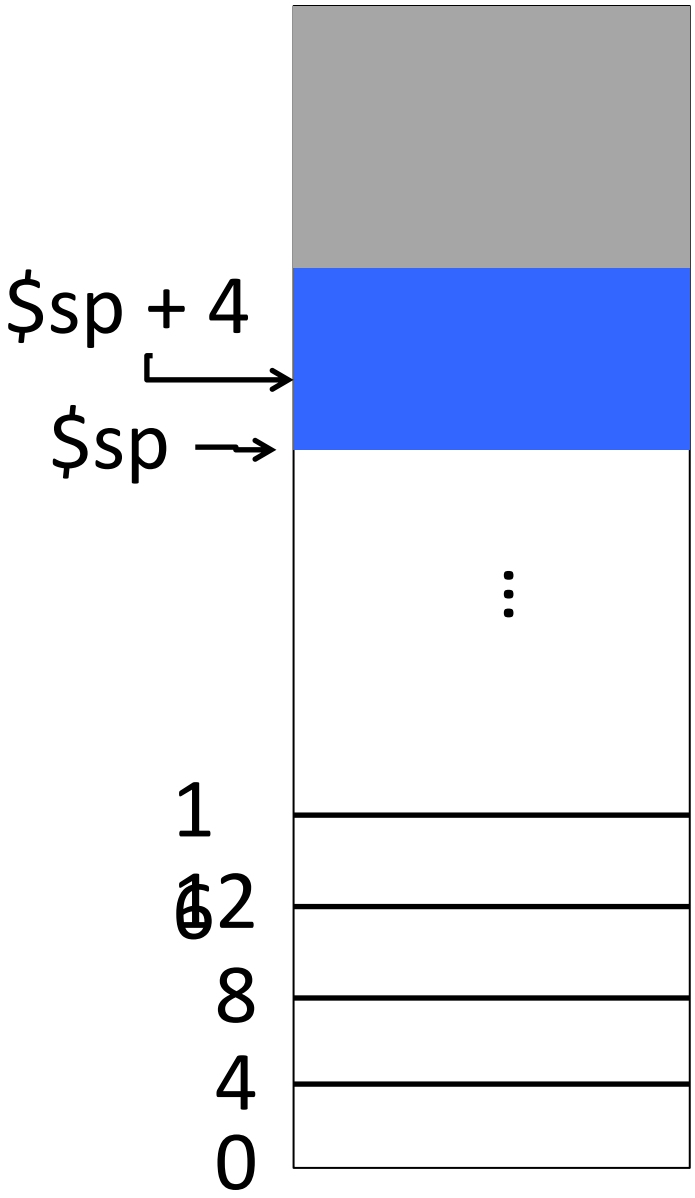



- 関数が自分の中だけで

**ここが今から使える**
**ローカル領域**

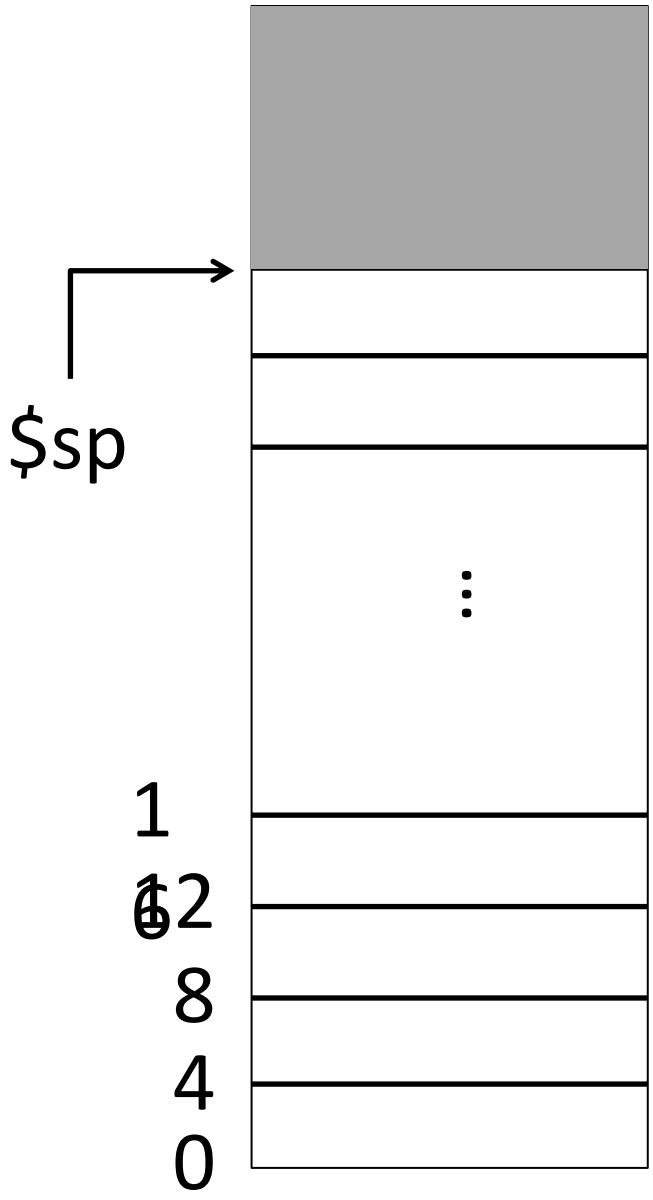
- 使っていい領域の天井のアドレスが $sp に入っている
- 関数の頭で自分が使う分だけ $sp を減らす

# ローカル領域




- 関数が自分の中だけで使える領域
- 使っていい領域の天井のアドレスが $sp に入っている
- 関数の頭で自分が使う分だけ $sp を減らす
- ローカル領域を使うときは $sp からの差分でアクセス
  - 4($sp) とか書く

# ローカル領域




- 関数が自分の中だけで使える領域
- 使っていい領域の天井のアドレスが $sp に入っている
- 関数の頭で自分が使う分だけ $sp を減らす
- ローカル領域を使うときは $sp からの差分でアクセス
  - 4($sp) とか書く
- 関数から返るときに $sp を行儀よく戻す

# したがってそれぞれの命令の意味は

```
    .text
    .globl main
main:
    addiu $sp,$sp,-20
    li    $t0,5
    sw    $t0, 4($sp)
    lw    $t1, 4($sp)
    li    $v0,1
    move $a0,$t1
    syscall
    addiu $sp,$sp,20
    jr    $ra
```

なぜ最初に $sp を -20 減らしているのか

なぜ $sp + 4 に値を書くのか

なぜ最後に $sp を 20 増やしているのか

# したがってそれぞれの命令の意味は

```
    .text
    .globl main
main:
    addiu $sp,$sp,-20
    li    $t0,5
    sw    $t0, 4($sp)
    lw    $t1, 4($sp)
    li    $v0,1
    move $a0,$t1
    syscall
    addiu $sp,$sp,20
    jr    $ra
```

main 関数の頭でローカル領域を 20 バイト確保

ローカル領域である
アドレス $sp + 4 に書き込み

main 関数から返るときに
20 バイト戻す

# 3つ目の例: 関数呼び出し

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

- example03.s というファイルを作って，左のコードを入力して QtSpim で実行してください

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

**jal f: f という関数を**
**呼び出している**

# jal 命令

- 関数呼び出し命令
  - $ra に「関数の実行が終わったら帰って来るべき場所」が格納される
    - したがって，今の $ra の値を，必要ならばローカル領域に退避する必要がある (後述)
  - 退避した $ra は関数から帰ってきた後に戻す必要がある

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

**$ra をローカル領域 $sp+0 に退避 (後述)**

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

退避した値を戻す (後述)

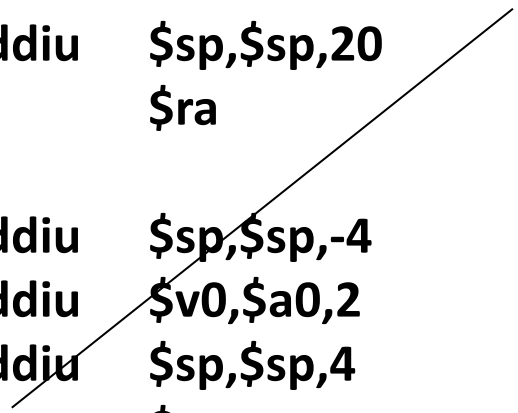
# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

関数内のローカル領域の確保

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```



$ra に格納されている
「終わった後帰ってくるべき場所」
にジャンプ

# 引数・返り値の受け渡し方法

- 最初の 4 つの引数はレジスタで
  - $a0,…,$a3 に最初の 4 つの引数
- 残りの引数はメモリ経由で
  - ローカル領域の高アドレス側にあらかじめ置いておく
  - ヘネパタ A-25 参照
- 返り値は $v0, $v1 で

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

一つ目の引数を $a0 にセット

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

渡された引数 ($a0) を使って
返り値 ($v0) を計算

# 解説

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $a0,5
        sw      $ra,0($sp)
        jal     f
        lw      $ra,0($sp)
        move    $a0,$v0
        li      $v0,1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
f:
        addiu   $sp,$sp,-4
        addiu   $v0,$a0,2
        addiu   $sp,$sp,4
        jr      $ra
```

f からの返り値 ($v0) を
呼び出し側で使う

# 呼び出し規約

- 関数間での引数・返り値の受け渡し方法やレジスタの使い方に関する規則 (A-22 参照)
  - 引数: $a0,…,$a3 に最初の 4 つ，残りはメモリ経由で渡す
  - 返り値: $v0, $v1 で返す
  - $t0,…,$t9 は関数側で上書きして良い
  - $s0,…,$s7 は関数側は上書きしてはいけない
    - 上書きする場合は，関数から返るときに元に戻さないといけない

# なぜ呼び出し規約が必要なのか

- ライブラリ関数などをどこからでも呼び出せるように
  - もし呼び出し規約がなかったら: 関数がどのレジスタを上書きするかとかをいちいち気にしなければならなくて不便

# 4つ目の例: 再帰関数呼び出し

- example04.s というファイルを作って，次のページのコードを入力して QtSpim で実行してください

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

f は 1 + 2 + … + $a0 を計算して
$v0 に返す関数

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

# f の C 言語での実装

- こんな感じ

```
int f(int x) {
  if (x <= 0) {
    return 0;
  } else {
    return (x + f(x - 1));
  }
}
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

最初の引数セット

```
f:
        addiu   $a0,$a0,-1
        sw      $ra,4($sp)
        jal     f
        lw      $ra,4($sp)
        lw      $a0,8($sp)
        addu    $v0,$v0,$a0
        addiu   $sp,$sp,12
        jr      $ra
end:
        li      $v0,0
        addiu   $sp,$sp,12
        jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

```
f:
    addiu   $sp,$sp,-12
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

今の $ra の値を 0($sp) に
退避しておく(後述)

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

f の呼び出し

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
                        p)
                        -1
                        p)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
                        1
                        )
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

退避しておいた $ra を
戻す (後述)

```
    .text
    .globl  main
main:
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

f のこの呼び出しで使う
ローカル領域を確保

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

$a0 が 0 より小さかったら end に飛ぶ (リファレンス参照)

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

$a0 が >0 の場合は
$a0 を一旦退避しておいて
$a0 を 1 減らす

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    jr      $ra
```

$ra を退避, f を呼び出す, $ra を戻す ($v0 に 1 + 2 + … + ($a0 - 1) がセットされる)

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
```

退避した $a0 を戻す

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
```

さっき呼び出した f の返り値に $a0 を足したものが全体の返り値

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
```

$sp を戻す

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
```

$a0 が <= 0 だった場合は
返り値は 0

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
```

$sp を元に戻して
呼び出し元に返る

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

# jal 命令 (再掲)

- 関数呼び出し命令
  - $ra に「関数の実行が終わったら帰って来るべき場所」が格納される
    - したがって，今の $ra の値を，必要ならばローカル領域に退避する必要がある (後述)
  - 退避した $ra は関数から帰ってきた後に戻す必要がある

**なぜこれが必要か**
**今から説明します**

# 5つ目の例: $ra を退避しなかったら?

- example04.s をコピーして example05.s というファイルを作り,
- $ra を退避・回復している命令 (次のページで灰色にしてあります) を削除して
- QtSpim で実行してください

```
    .text
    .globl  main
main:
    addiu   $sp,$sp,-20
    li      $a0,10
    sw      $ra,0($sp)
    jal     f
    lw      $ra,0($sp)
    move    $a0,$v0
    li      $v0,1
    syscall
    addiu   $sp,$sp,20
    jr      $ra
```

```
f:
    addiu   $sp,$sp,-12
    ble     $a0,0,end
    sw      $a0,8($sp)
    addiu   $a0,$a0,-1
    sw      $ra,4($sp)
    jal     f
    lw      $ra,4($sp)
    lw      $a0,8($sp)
    addu    $v0,$v0,$a0
    addiu   $sp,$sp,12
    jr      $ra
end:
    li      $v0,0
    addiu   $sp,$sp,12
    jr      $ra
```

最初に覚えておいた $ra が消えた!
余計なことを! (余計じゃないけど)

CPU の気持ち:「f を実行した後は $ra に書いてあるところに戻ればいいのねん」

CPU の気持ち:「f を呼び出せばいいのねん」

CPU の気持ち:「社長さんのために $ra にこの jal 命令の直後のアドレスをセットしてあげたのねん」

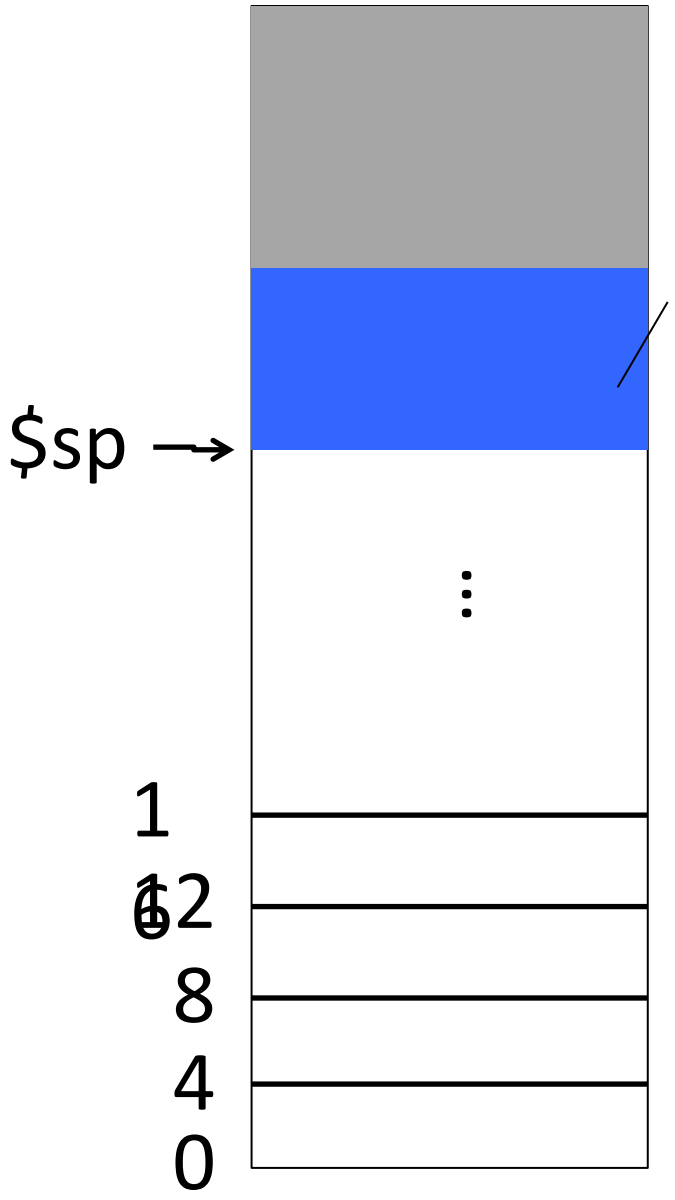
```
f:
        addiu   $sp,$sp,-12
        ble     $a0,0,end
        sw      $a0,8($sp)
        addiu   $a0,$a0,-1
        sw      $ra,4($sp)
        jal     f
        lw      $ra,4($sp)
        lw      $a0,8($sp)
        addu    $v0,$v0,$a0
        addiu   $sp,$sp,12
        jr      $ra

end:
        li      $v0,0
        addiu   $sp,$sp,12
        jr      $ra
```

# $ra とネストした関数呼び出し

- 関数の中から関数を呼び出すと $ra の内容が上書きされる
- なので関数から返ってきた後に $ra の内容を元に戻す必要がある場合は $ra をスタックに退避する必要がある
  - フレーム内に $ra 退避用の領域を作っておくことが多い

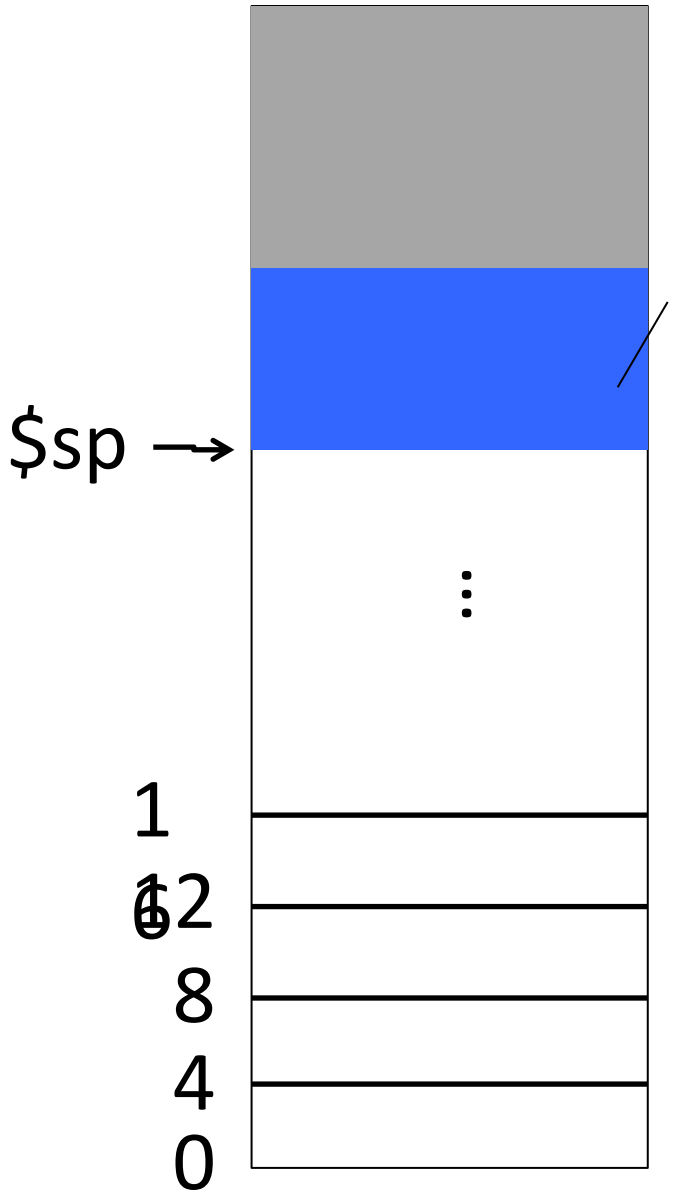
# 再帰関数呼び出しでのローカル領域

- 再帰呼び出しごとにローカル領域が確保される



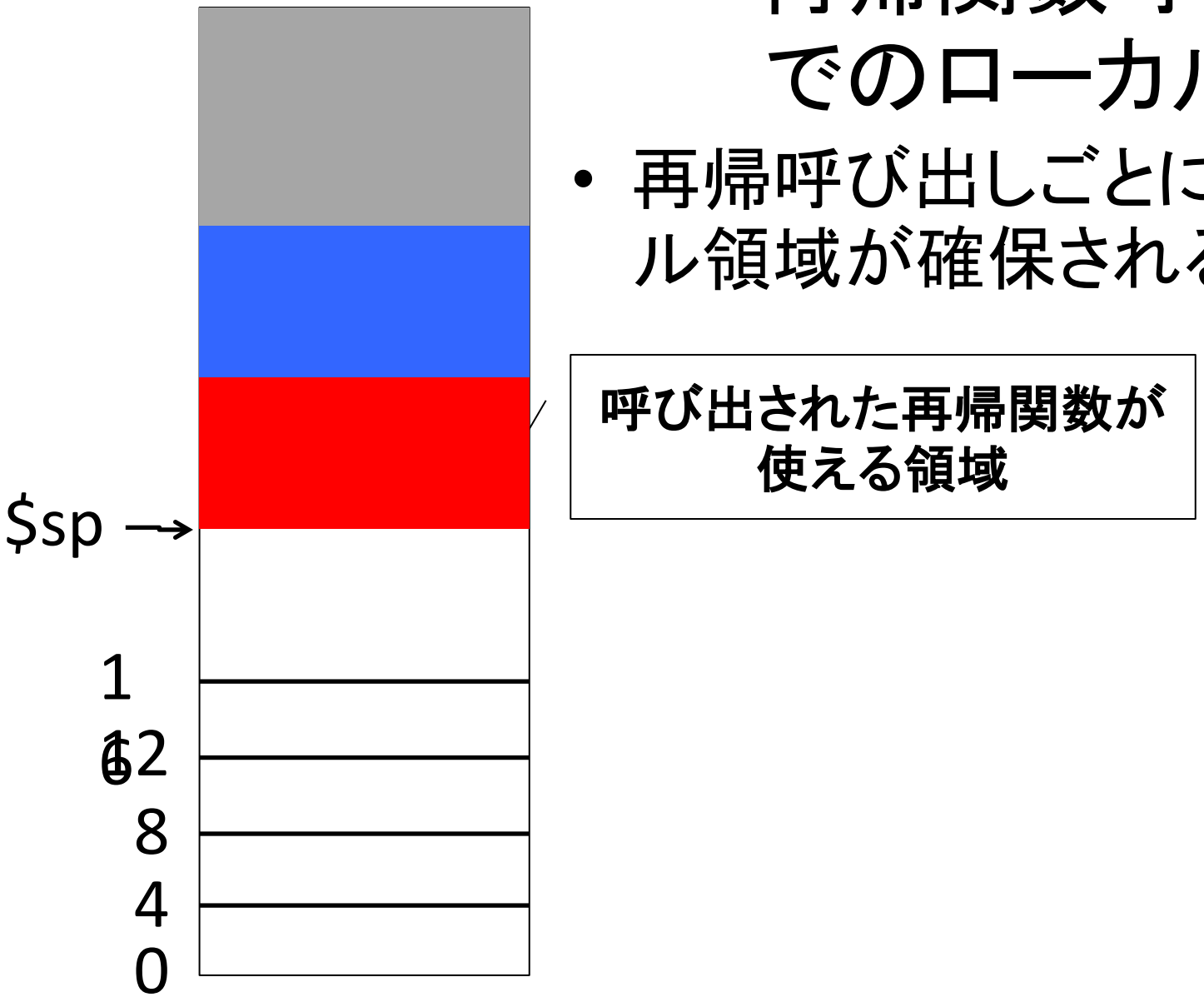
# 再帰関数呼び出しでのローカル領域

- 再帰呼び出しごとにローカ
  る



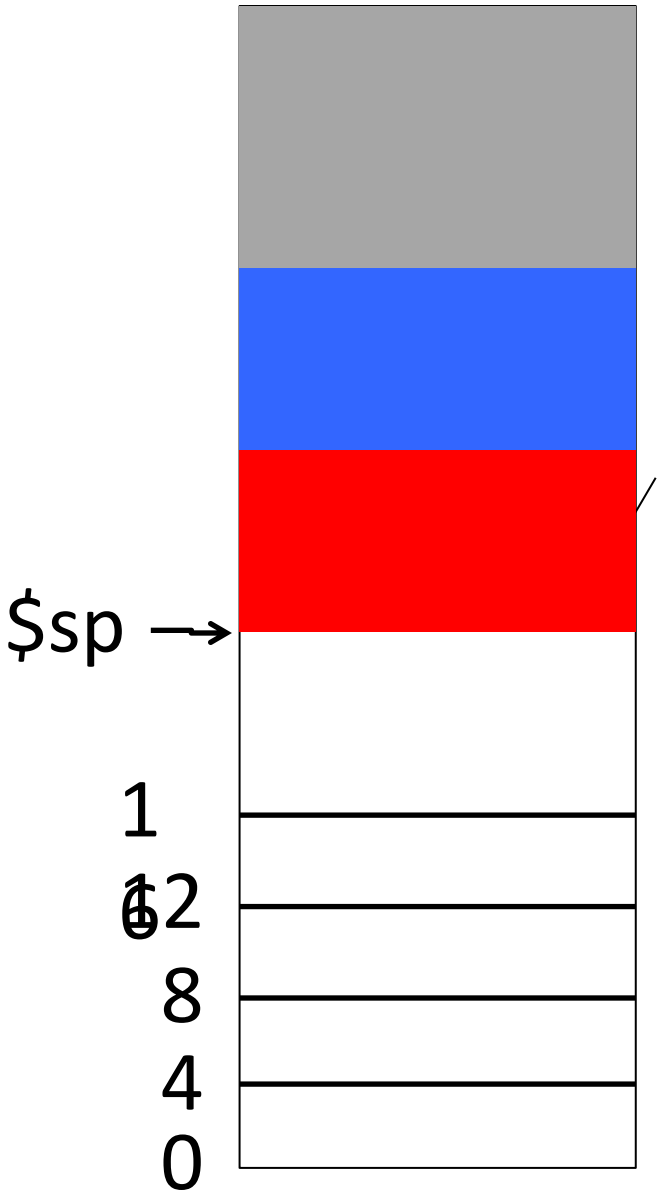
# 再帰関数呼び出しでのローカル領域

- 再帰呼び出しごとにローカル領域が確保される



# 再帰関数呼び出しでのローカル領域

- 再帰呼び出しごとにローカル領域が確保される

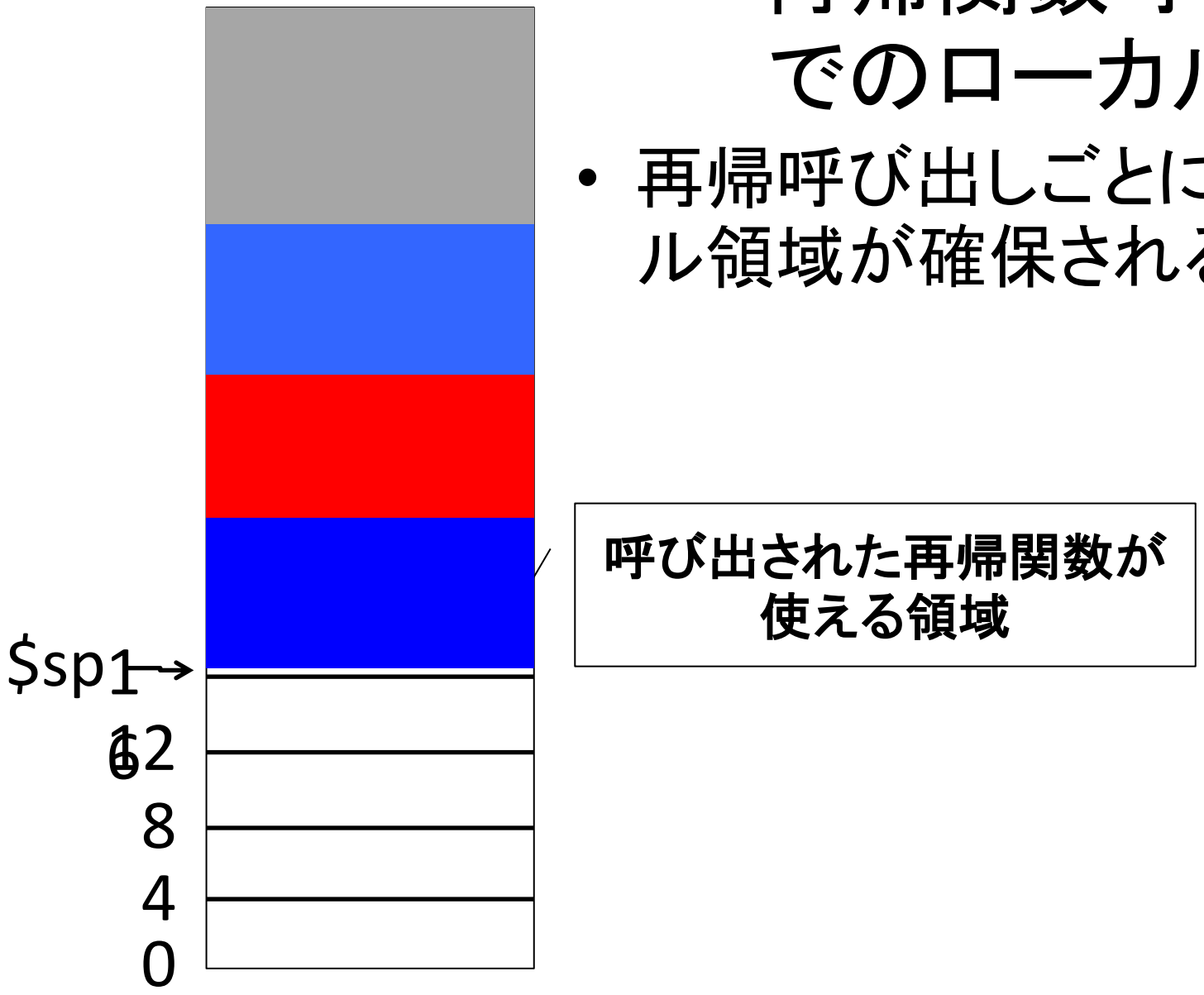



呼び出された再帰関数が使える領域

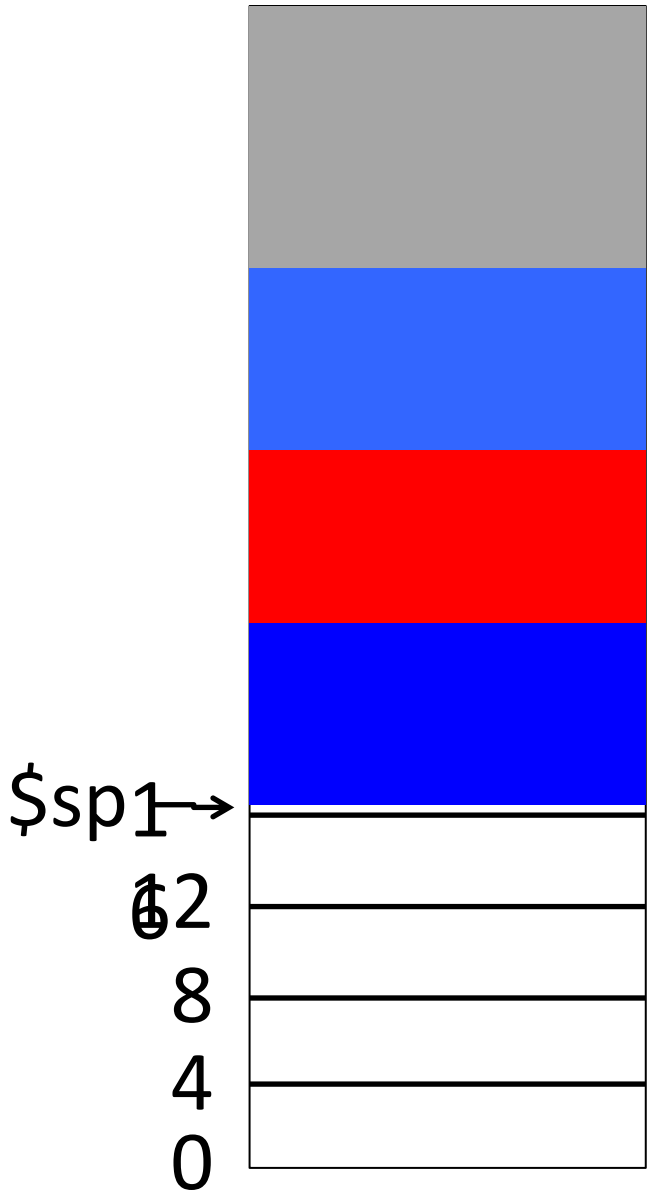
再帰関数呼び出し

# 再帰関数呼び出しでのローカル領域

- 再帰呼び出しごとにローカル領域が確保される



# 再帰関数呼び出しでのローカル領域




- 再帰呼び出しごとにローカル領域が確保される
- ローカル領域にはローカル変数のみならず 関数実行に必要な情報全般を保存
  - $ra の値や引数など
  - 箱一つを「スタックフレーム」と呼ぶ
- ローカル変数として使われる領域を総称して「スタック」と呼ぶ

# ローカル領域以外のメモリ領域

- ヒープ
  - malloc で確保されるメモリ領域
- 静的データ領域
  - グローバル変数や定数などを配置する領域
- コード領域
  - プログラムを配置する領域

# 記憶領域割り当て

コード生成

# 中間命令とアセンブリのギャップ

- アセンブリでは変数は使えない
  - すべての変数にメモリかレジスタ (記憶領域) を割り当てる必要

```
x = 5;
*p = x;
y = *p;
print(y);
```

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $t0,5
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        li      $v0,1
        move    $a0,$t1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
```

# 中間命令とアセンブリのギャップ

- アセンブリでは if とか while とか使えない
  - 条件判定/ジャンプを明示的に行う必要

```
if (x) {
  y = 1;
  print(y);
} else {
 z = 2;
 print(z);
}
```

```
        .text
        .globl  main
main:
        beqz    $t0,false
        li      $a0,1
        li      $v0,1
        syscall
        j       end
false:
        li      $a0,2
        li      $v0,1
        syscall
end:
```

# 記憶領域割り当て

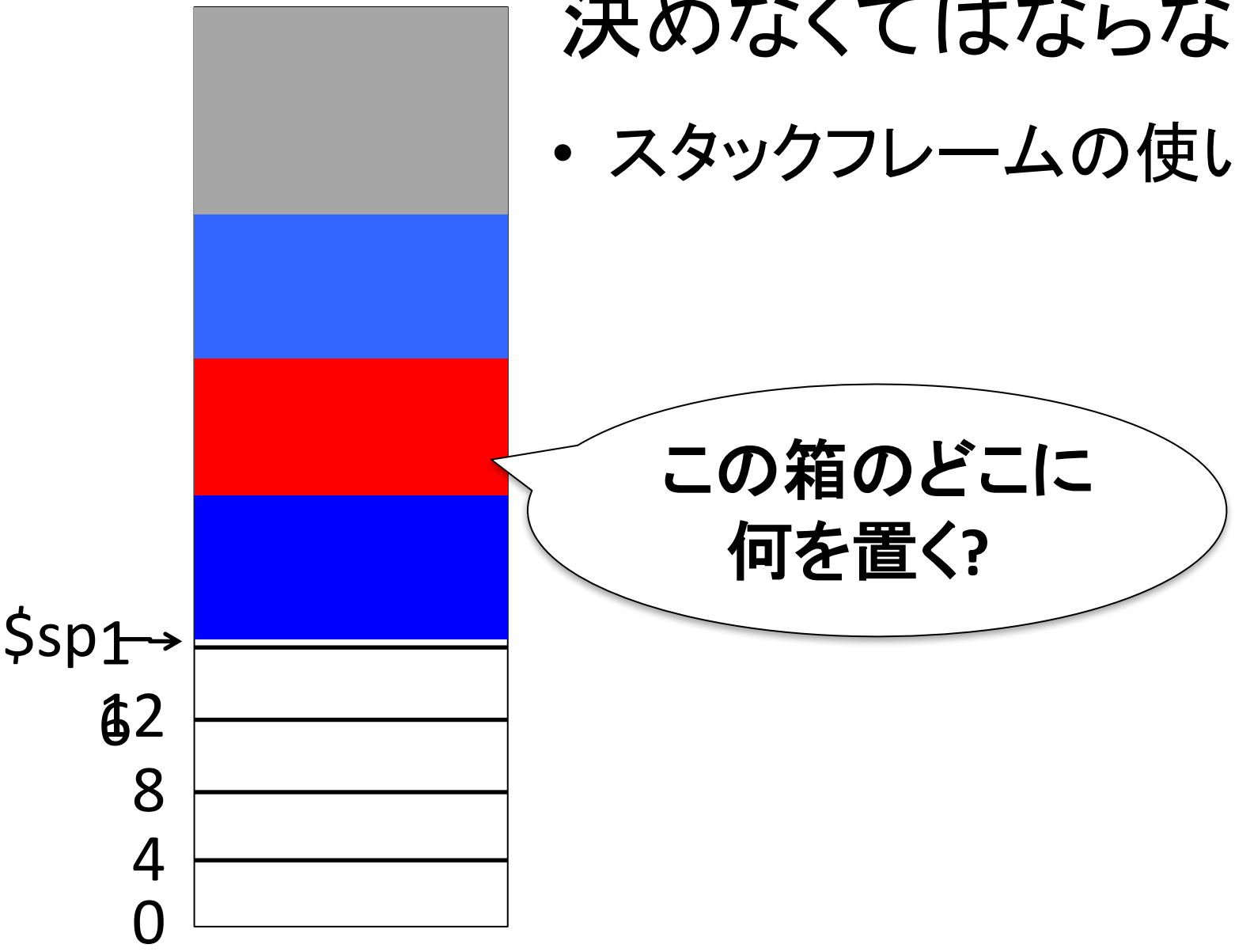
- 各変数にローカル領域上の場所を割り当てる

```
x = 5;
*p = x;
y = *p;
print(y);
```

```
        .text
        .globl  main
main:
        addiu   $sp,$sp,-20
        li      $t0,5
        sw      $t0, 4($sp)
        lw      $t1, 4($sp)
        li      $v0,1
        move    $a0,$t1
        syscall
        addiu   $sp,$sp,20
        jr      $ra
```

# 決めなくてはならないこと

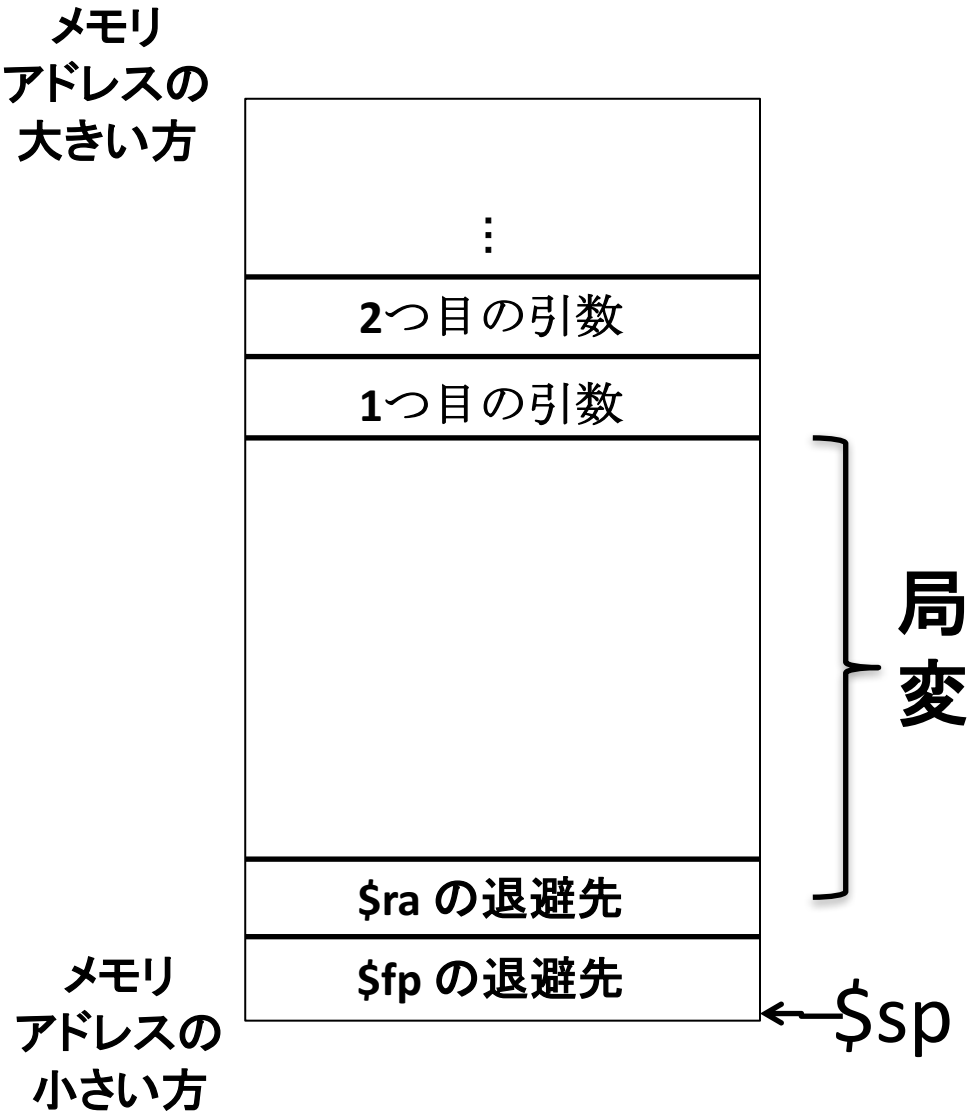
- スタックフレームの使い方



# フレームに入れなくてはならない情報は何か

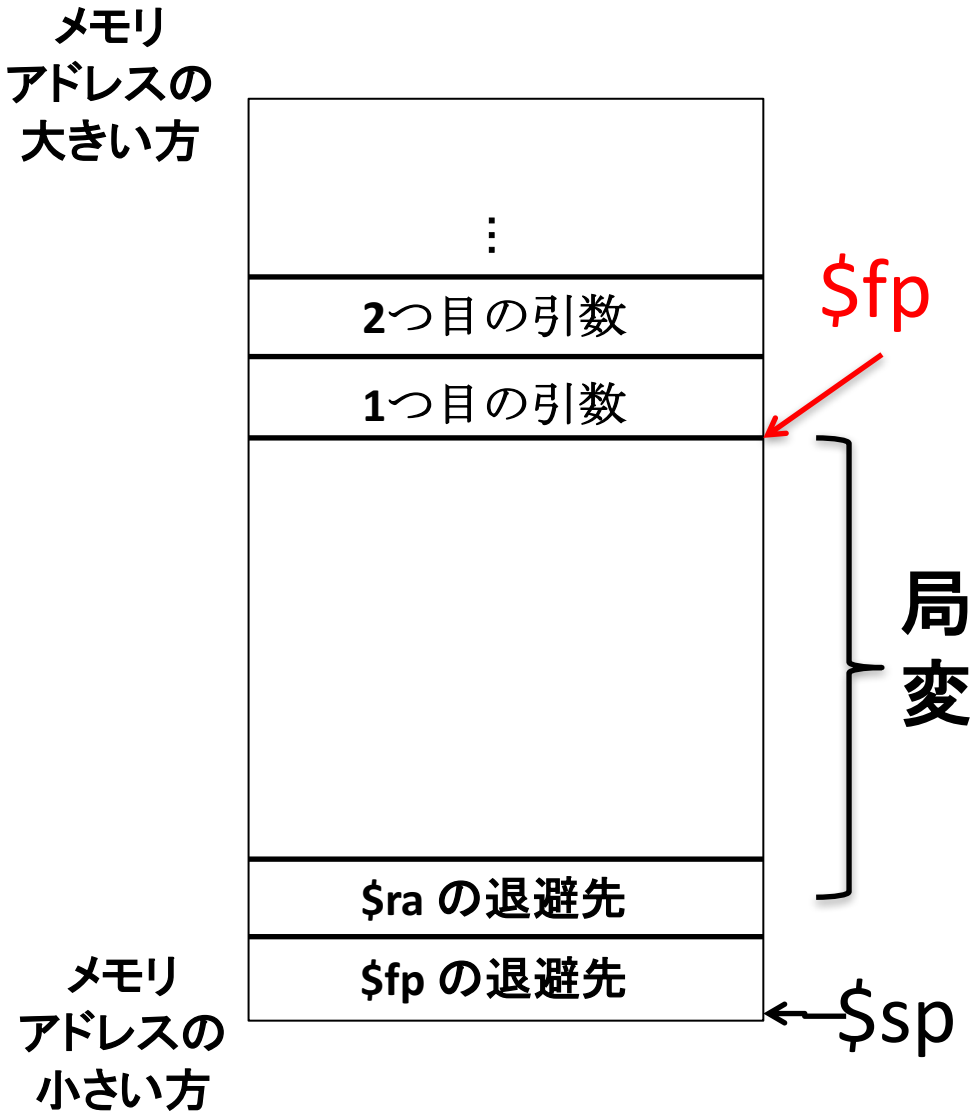
- 関数に渡される引数
  - MIPS の呼び出し規約では 5 つ目以降の引数
  - この講義では差し当たって全引数をメモリにおいて渡すことにする (レジスタを使わない)
    - そっちの方が簡単なので
- 局所変数
- 再帰関数呼び出しをする際の $ra
- 再帰関数呼び出しをする際の $fp
  - あとで説明

# (この講義での) 関数フレームの構造




- $sp はフレームの先頭を指す
- 関数の引数は (当面) スタック経由で渡す
- $ra と $fp の退避場所は決まっている

# フレームポインタ $fp




- フレームの途中を指すレジスタを用意することが多い
  - フレームポインタと呼ぶ
- 本講義では１つ目の引数の場所
- フレーム中の情報にアクセスしやすくなるというご利益
  - 引数がどのようなサブルーチンも 0($fp), 4($fp), …
  - $sp しかないと局所変数領域のサイズによって引数のオフセットが変わる

# 記憶領域割り当ての方法

- 構文主導翻訳
  - 変数と記憶領域の対応関係を記録するリストを保持
  - 構文木中で変数 x に遭遇したら以下を実行
    - x がすでにリストに入っていたら，すでに割り当てたアドレスを使う
    - x がリストに入っていなかったら，新しく記憶領域を割り当て，リストを更新し，割り当てたアドレスを使う

- 詳細は assignAddr.rkt を参照

# 中間命令とアセンブリのギャップ (再掲)

- アセンブリでは if とか while とか使えない
  - 条件判定/ジャンプを明示的に行う必要

```
if (x) {
  y = 1;
  print(y);
} else {
 z = 2;
 print(z);
}
```

```
        .text
        .globl  main
main:
        beqz    $t0,false
        li      $a0,1
        li      $v0,1
        syscall
        j       end
false:
        li      $a0,2
        li      $v0,1
        syscall
end:
```

# if とか while とかどうするの?

- がんばってアセンブリで対応するコードを作る
  - 詳しくは次のセクションで
  - アドレスを割り当てておくと後はだいぶラク

# アセンブリ生成

# ここです

プログラムテキスト

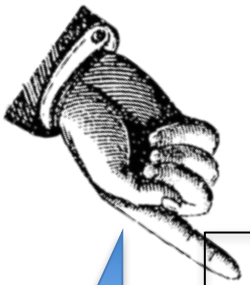
```
int main(int argc,
         char **argv)
{
  printf("Hello!¥n");
  return 0;
}
```

字句解析

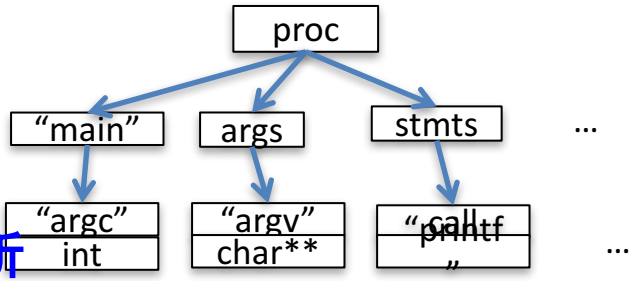
トークン列

```
INT; ID(“main”);
LPAREN; INT;
ID(“argc”); CHAR; AST;
AST; ID(“argv”);
RPAREN; LBRACE; …
```

構文解析



抽象構文木




意味解析・最適化

```
  .data
L0:
  .asciiz “Hello!¥n”
  .text
  .globl main
main:
  subu $sp,20,$sp
  addu $fp,$sp,8
  li   $t0,L0
  sw   $t0,0($sp)
  …
```

アセンブリ生成

```
main : int->char**->int
  set x “Hello!¥n”
  call ret printf [x]
  set y 0
  return y
```

中間命令列

最適化

アセンブル

```
0000000  facf  feed  0007  0100  0003  8000  0002  0000
0000020  0010  0000  05b0  0000  0085  0020  0000  0000
0000040  0019  0000  0048  0000  5f5f  4150  4547  455a
0000060  4f52  0000  0000  0000  0000  0000  0000  0000
0000100  0000  0000  0001  0000  0000  0000  0000  0000
0000120  0000  0000  0000  0000  0000  0000  0000  0000
0000140  0000  0000  0000  0000  0019  0000  0228  0000
0000160  5f5f  4554  5458  0000  0000  0000  0000  0000
0000200  0000  0000  0001  0000  1000  0000  0000  0000
0000220  0000  0000  0000  0000  1000  0000  0000  0000
240  ……
```

機械語

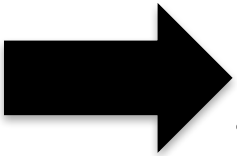
アセンブリ

最適化

# アセンブリ生成の方法

- 構文主導翻訳
  - 中間命令の各構成子をアセンブリに変換
  - MIPS アセンブリはマスターしていることを前提に進めます

**x = y + 2**



```
lw $t0,-4($fp)
li $t1,2
addu $t0,$t0,$t1
sw $t0,-8($fp)
```

**(x がオフセット -8 に,
y がオフセット -4 に
割り当てられているなら)**

# アセンブリ生成に使うプログラム

- codegen.rkt
  - アセンブリ生成用
- printcode.rkt
  - MIPS 命令列の内部表現 (あとで説明) を文字列に変換

- 両方とも PandA からダウンロードすること

# MIPS アセンブリの内部表現

- アセンブリの構成要素に対応する構造体
  - instr: 命令行を表す構造体
    - op: ‘li や ‘addu 等 MIPS の命令に対応するシンボル
    - args: ‘$ra 等の引数のシンボルのリスト
  - label: ラベル行を表す構造体
    - name: ラベル名のシンボル
  - dir: ディクレティブを表す構造体
    - label: ‘text や ‘globl のようなディレクティブ名のシンボル
    - args: 引数のシンボル
  - comment: コメント行を表す構造体
    - 生成するアセンブリにデバッグ用にマークを付けたいときに使用

# (この後必要になる知識) quote

- シンボルのリストを作る方法
  - ‘(li 3 4) は (‘li ‘3 ‘4) と同じ
  - ‘(+ 3 4) は (‘+ ‘3 ‘4) と同じ

# (この後必要になる知識) quasiquote

- quote で作った式の中に値を埋め込む方法

バッククオート

**–`(+ 3 4) → (‘+ ‘3 ‘4)**

**–(let ([x 3]) `(+ ,x 4)) → (‘+ ‘3 ‘4)**

**quote の中でコンマを使うと, その直後の式が評価されて埋め込まれる**

# 式をアセンブリに変換する関数

- intermed-exp->code
  - 引数: dest, e
  - 中間命令の式 e を評価してアドレス dest に書き込むアセンブリを返す

# 例えば変数式 (i.e., (varexp x)) は

- x がオフセット -4 に，dest がオフセット -8 に割り当てられてれば…

```
lw   $t0,-4($fp)
sw   $t0,-8($fp)
```

# 変数式 (varexp x) の変換の仕方

- addr->sym 関数を使って x のアドレスと dest のアドレスの表現をシンボルの形で取り出す
  - symsrc, symdest とする
- symsrc のアドレスから値を読んで symdest のアドレスに値を書き込むコードを生成する
  - lw でメモリの symsrc 番地からレジスタに値をロード
  - sw でメモリの symdest 番地に値をストア
  - 具体的にどのようになるかはソースコード参照

# 例えば算術式
# (i.e., (aopexp addu x1 x2)) は

- x1 がオフセット -4 に，x2 が オフセット -8 に，dest がオフセット -12 に割り当てられてれば…

```
lw    $t0,-4($fp)
lw    $t1,-8($fp)
addu  $t0,$t0,$t1
sw    $t0,-12($fp)
```

# (aopexp op x y) の変換方法

- addr->sym 関数を使って x と y と dest のアドレスをシンボルとして取り出す
  - sym1, sym2, symdest とする
- メモリから sym1 番地と sym2 番地の値をレジスタにロードする命令を生成
  - lw を使う
- レジスタを用いて op に相当する算術演算
  - 結果をどちらかのレジスタに保存
- 演算結果をメモリの symdest 番地にストア

# 文をアセンブリに変換する関数

- intermed-stmt->code
  - 引数 localvarsinbytes と argsinbytes は今のところは無視して OK
  - 引数 s が変換しようとする文

# 例えば代入文
# (i.e., (letstmt to exp)) は

- to がオフセット -4 に割り当てられていれば

さっきの関数 intermed-exp->code を使って
オフセット -4 に exp の評価結果を
代入するコードを生成すればよい

# 例えばメモリへの書き込み文 (i.e., (writestmt to from)) は

- to がオフセット -4 に，from が オフセット -8 に，割り当てられてれば…

```
lw    $t0,-4($fp)
lw    $t1,-8($fp)
sw    $t1,0($t0)
```

# メモリへの書き込み文 (i.e., (writestmt to from)) は

- addr->sym 関数を使って to と from のアドレスをシンボルにしておく
  - symdest, symsrc とする
- メモリの symdest と symsrc の値をレジスタに読み込む
  - symdest は -8 等のオフセットで表現されるアドレスが入っているはず
- symdest の値が表すアドレスに，symsrc の値を読み込む

# 条件分岐文
# (ifstmt x stmts1 stmts2) は

- x がオフセット -4 に割り当てられてれば…

```
    lw      $t0,-4($fp)
    beqz    $t0,falselabel
    [stmts1 に対応するコード]
    j       end
falselabel:
    [stmts2 に対応するコード]
end:
```

他のラベルとかぶらないラベル名に

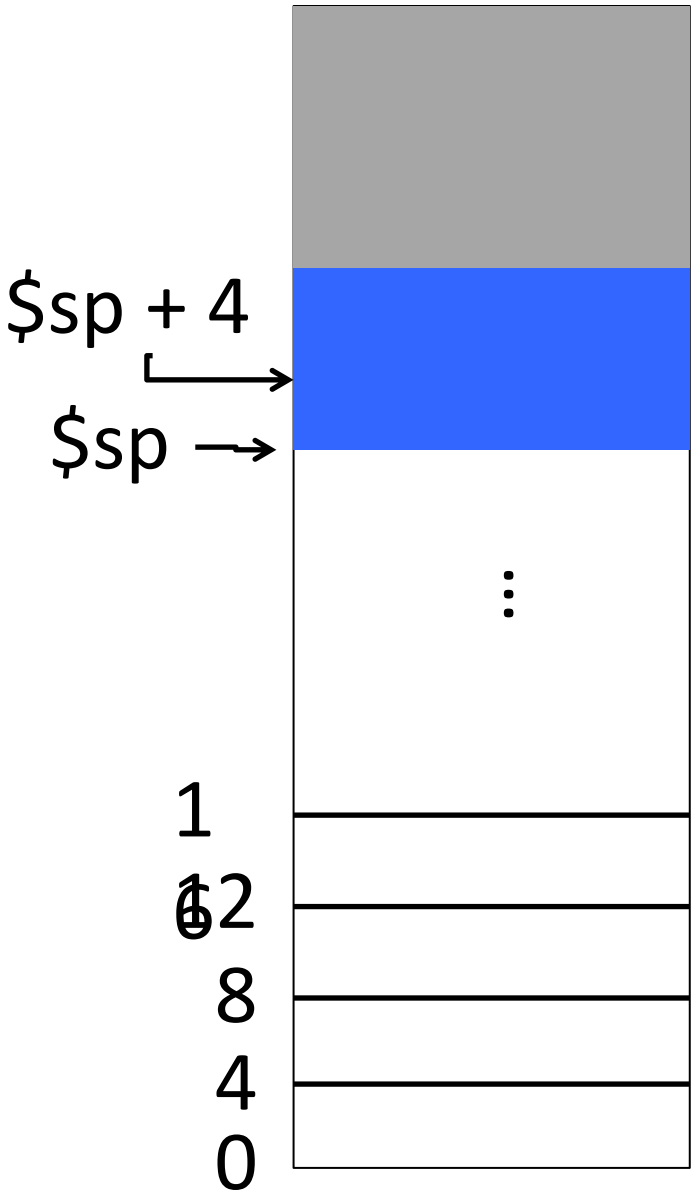
再帰的に生成

他のラベルとかぶらないラベル名に

# (ifstmt x stmts1 stmts2) の生成の仕方

- addr->sym 関数で x のアドレスをシンボルに
  - sym とする
- stmts1 と stmts2 に対応するコードを
  生成しておく
  - それぞれ中間命令文のリストなので，map 関数で intermed-stmt->code をそれぞれに適用し，flatten 関数で入れ子リストを平たいリストにする
  - code1, code2 とする
- 他とかぶらないラベルを nextlabel 関数で 2 つ生成
- 前ページの方法にしたがってコードを作る

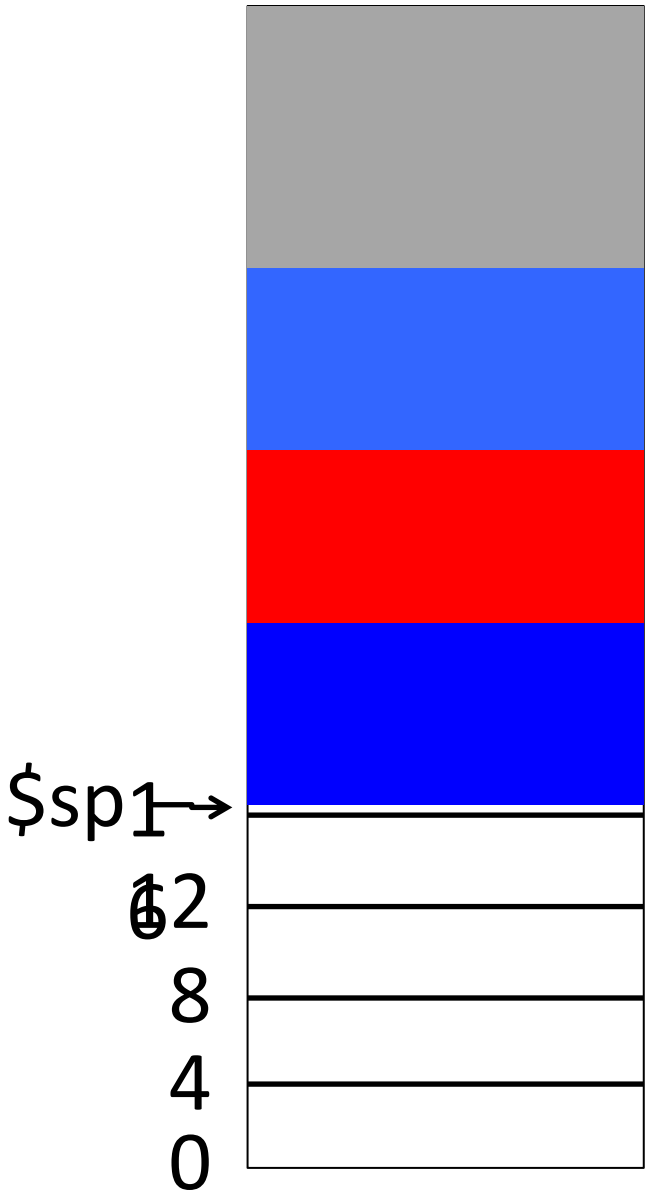
# 関数定義・関数呼び出しのサポート




# ローカル領域 (復習)

- 関数が自分の中だけで使える領域
- 使っていい領域の天井のアドレスが $sp に入っている
- 関数の頭で自分が使う分だけ $sp を減らす
- ローカル領域を使うときは $sp からの差分でアクセス
  - 4($sp) とか書く

# 再帰関数呼び出しでのローカル領域 (復習)




- 再帰呼び出しごとにローカル領域が確保される
- ローカル領域にはローカル変数のみならず 関数実行に必要な情報全般を保存
  - $ra の値や引数など
  - 箱一つを「**スタックフレーム**」と呼ぶ
- ローカル変数として使われる領域を総称して「スタック」と呼ぶ

# (関数定義・呼び出しを含む)プログラムの中間命令への変換

- functions.pdf を参照
  - Tiny-C と中間命令の構文を関数定義・関数呼び出しで拡張
    - すでに実験用コンパイラには入っている
  - convToIntermed.rkt を拡張
  - assignAddr.rkt を拡張
  - codegen.rkt を拡張

# 構文の拡張

- functions.pdf の通り
  - <statement> 等を Tiny-C の文と中間命令の文と両方を表すように定義している

# 中間命令への変換の拡張

- ほとんど再帰的に変換を適用するだけ
- 関数呼び出しのみちょっとややこしい

$$\mathcal{I}(\langle var\rangle = \langle var\rangle_f(\langle exp\rangle_1, \ldots, \langle exp\rangle_n)) =$$
$$\mathcal{I}_{x_1}(\langle exp\rangle_1);$$
$$\ldots$$
$$\mathcal{I}_{x_n}(\langle exp\rangle_n);$$
$$\langle var\rangle = \langle var\rangle_f(x_1, \ldots, x_n)$$
$$\text{where } x_1, \ldots, x_n \text{ are fresh variables.}$$

実引数一つ一つについて

新しい変数を作って

その変数に結果を代入する中間命令を作る

# Racket ではどう実装するか (ヒント)

- まず map 関数で fresh な変数のリスト vars を作る
- vars と引数の式のリストを受け取って syntax-expr->intermed-stmts を順番に適用する再帰関数を定義
  - foldl 関数を使っても良い
- 全中間命令と関数呼び出し命令をリストにして，flatten で潰す

# アドレス割り当ての拡張 (関数呼び出し･return)

- 再帰的にアドレス割り当て関数を適用するだけ

# アドレス割り当ての拡張 (関数定義)

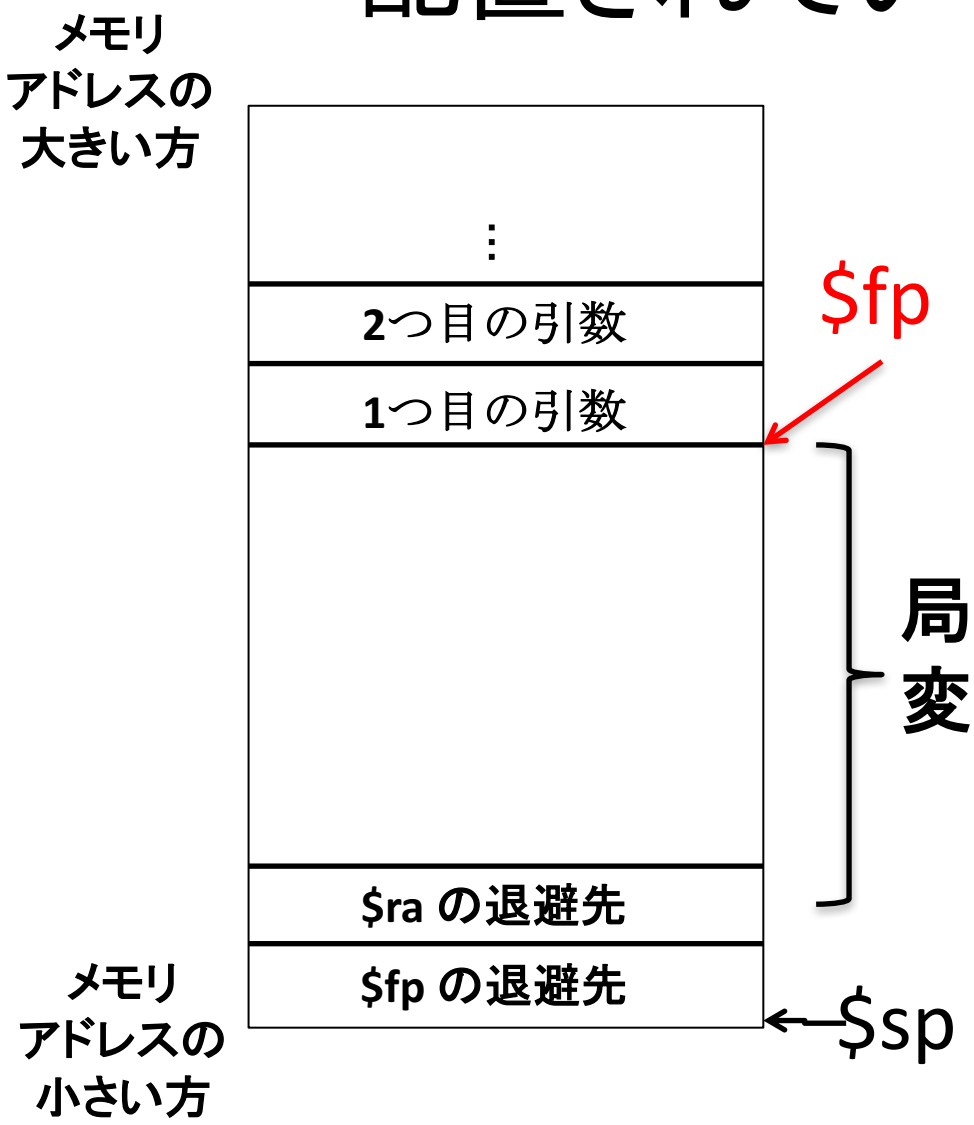
- 仮引数とオフセットの対応関係をまず作る

各仮引数が置かれる
オフセット

$$\textbf{let } \delta_0 = \{\langle var \rangle_1 \mapsto m_1, \ldots, \langle var \rangle_n \mapsto m_n\} \textbf{ in}$$

# 引数は関数本体ではどこに配置されているように見える?

- 1個目の引数: 0($fp)
- 2個目: 4($fp)
- …



# アドレス割り当ての拡張 (関数定義)

- 仮引数とオフセットの対応関係をまず作る

各仮引数が置かれる
オフセット

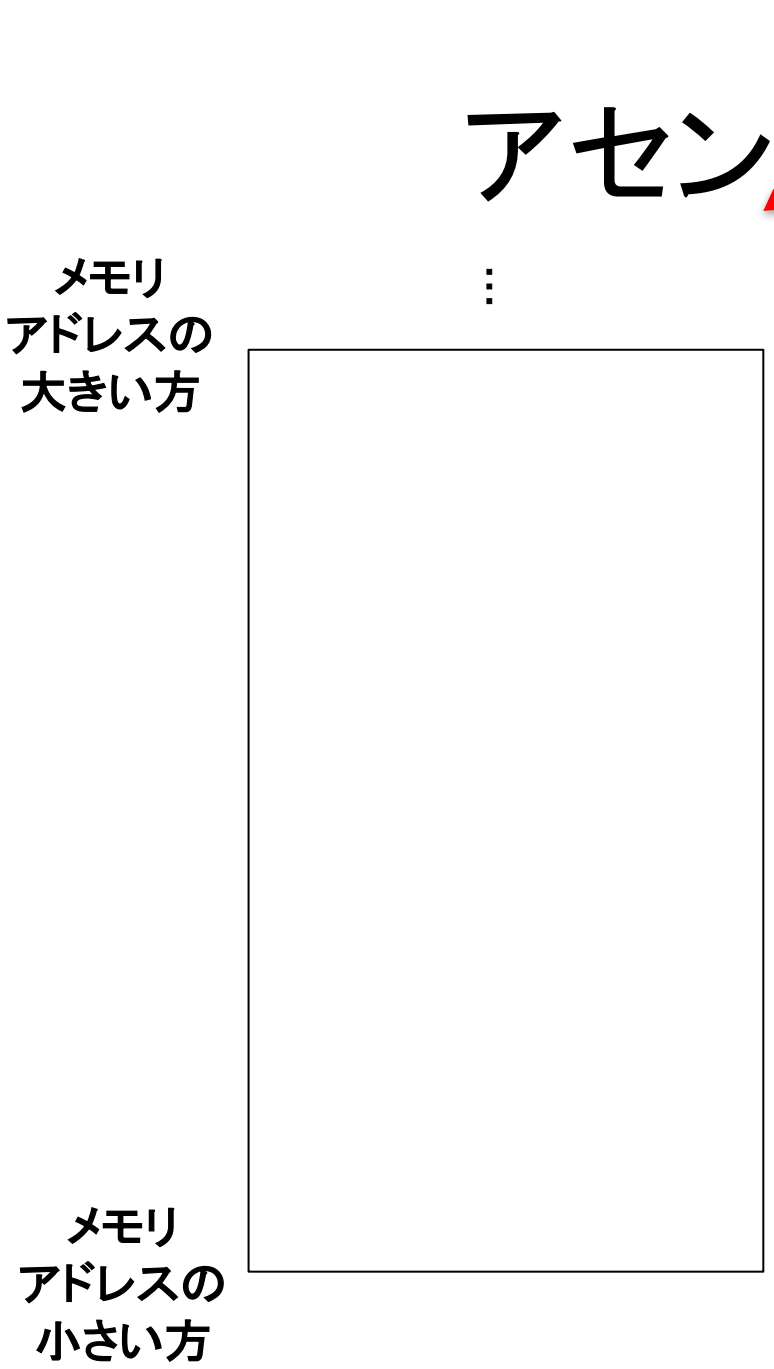
$$\mathbf{let}\ \delta_0 = \{\langle var\rangle_1 \mapsto m_1, \ldots, \langle var\rangle_n \mapsto m_n\}\ \mathbf{in}$$

- 関数本体にアドレス割り当て
- 局所変数のサイズを記録しておく
  - あとで関数フレームを確保するコードを生成するのに必要

# アドレス割り当ての拡張 (プログラム全体)

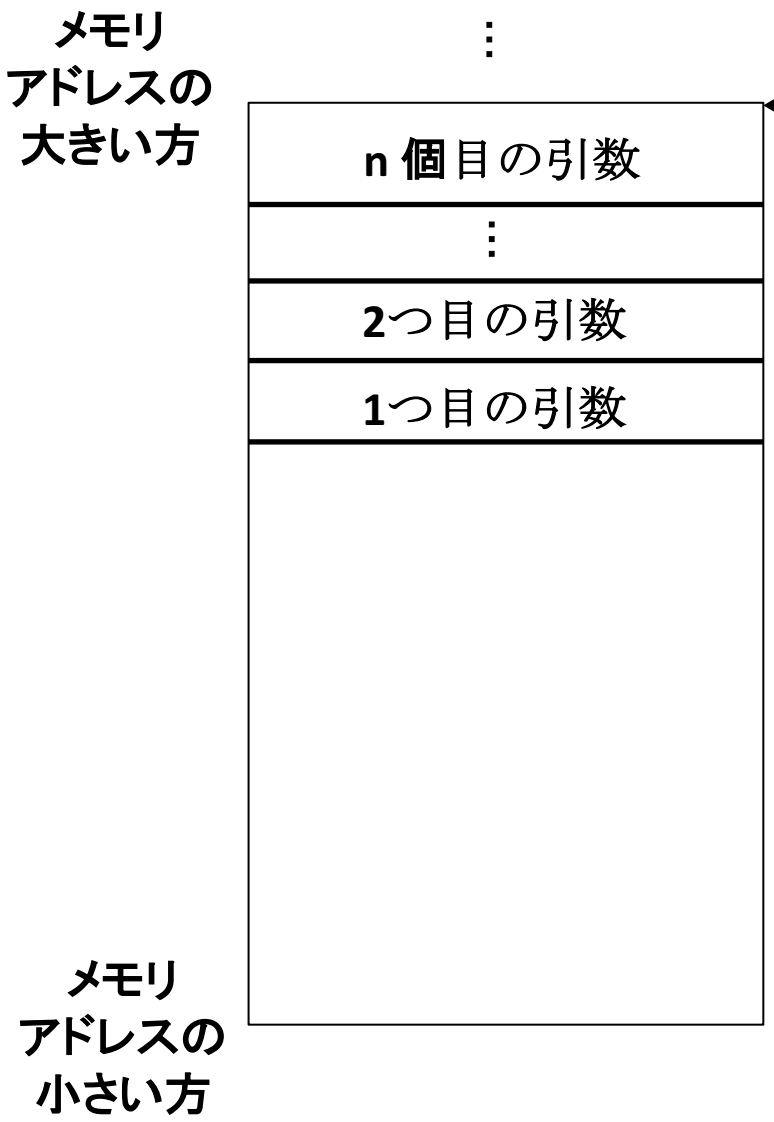
- 各関数定義と main について
  アドレス割り当てを再帰的に適用

# アセンブリ生成の拡張



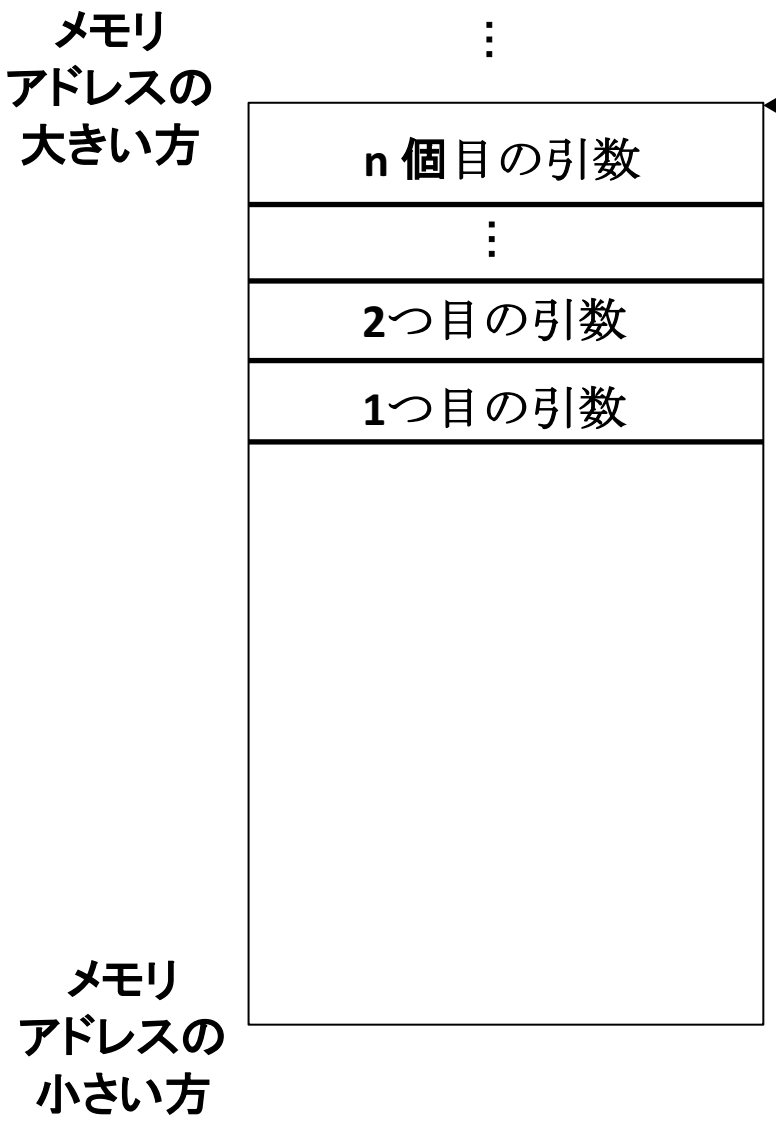

- $sp
- 関数呼び出し時

# アセンブリ生成の拡張



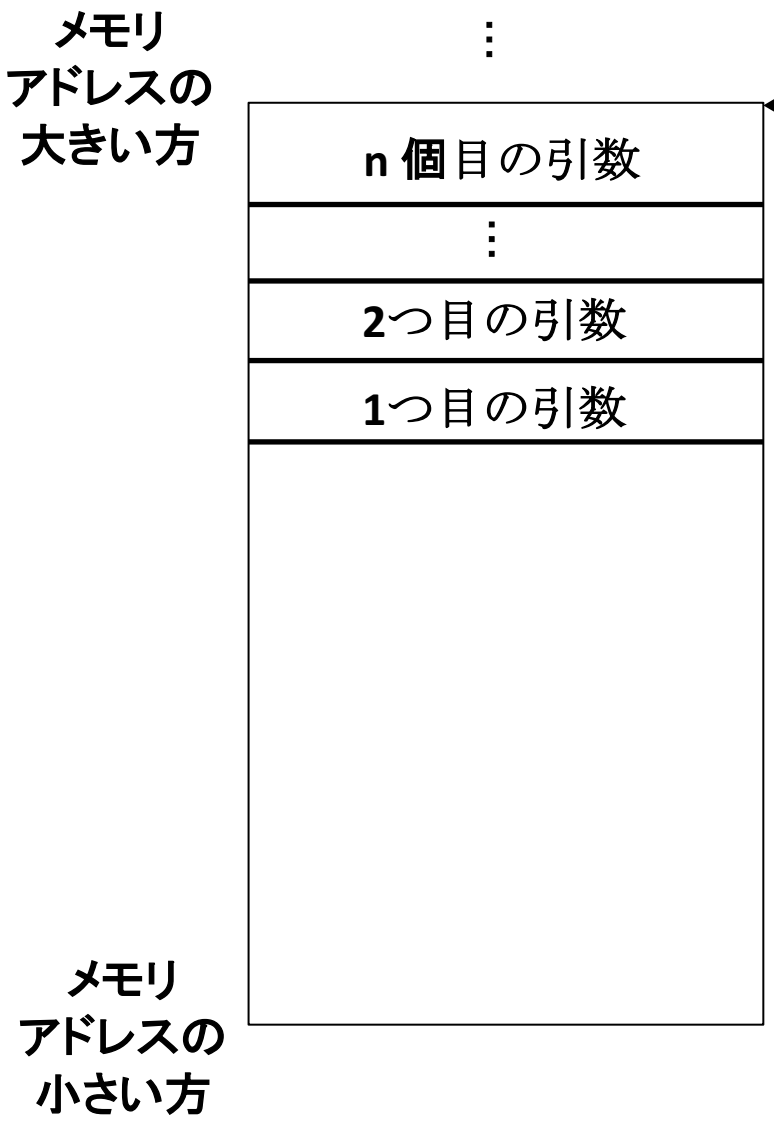

- 関数呼び出し時
  - 実引数を逆順で $sp の下にストア

# アセンブリ生成の拡張



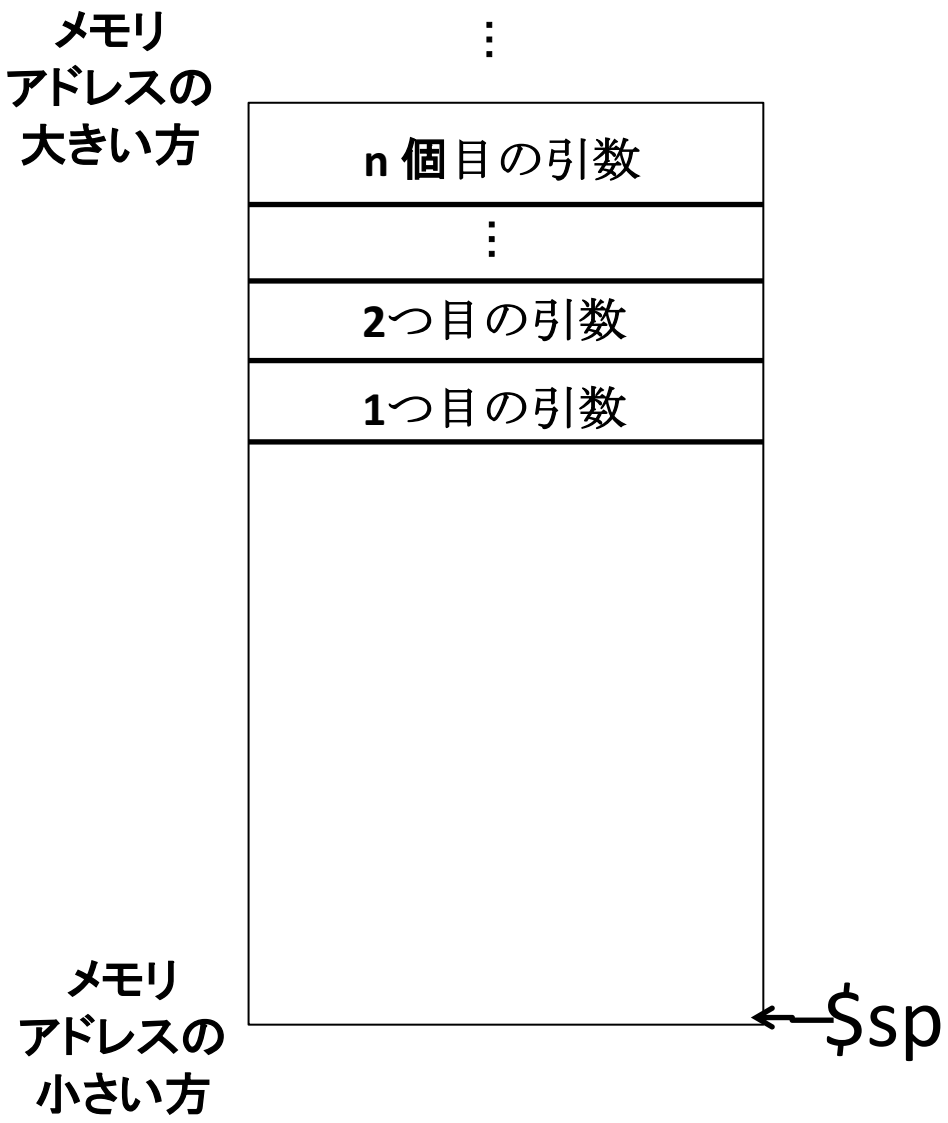

- 関数呼び出し時
  - 実引数を逆順で $sp の下にストア
  - jal 命令で関数を呼ぶ
  - jal から戻った直後に返り値を $v0 から指定された場所にストア

# アセンブリ生成の拡張



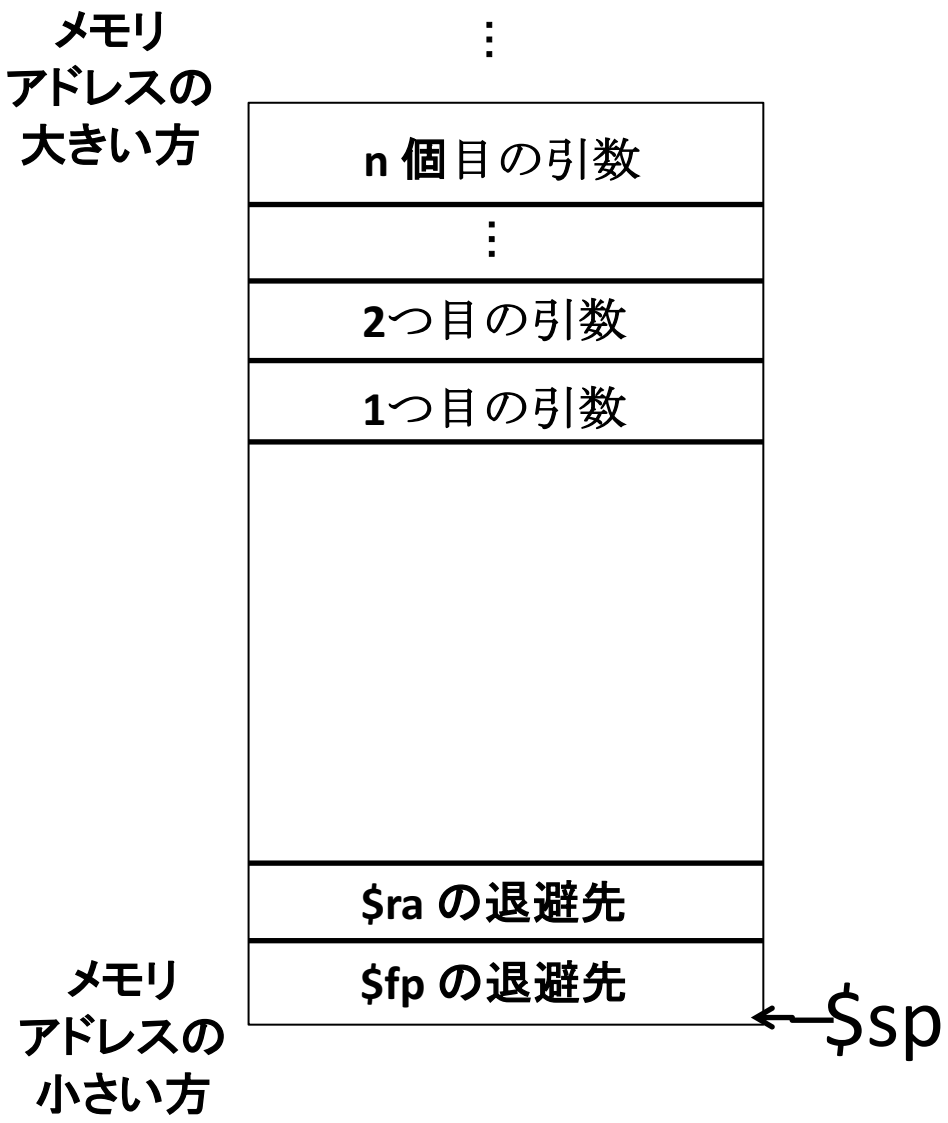

- 関数定義本体の頭

# アセンブリ生成の拡張



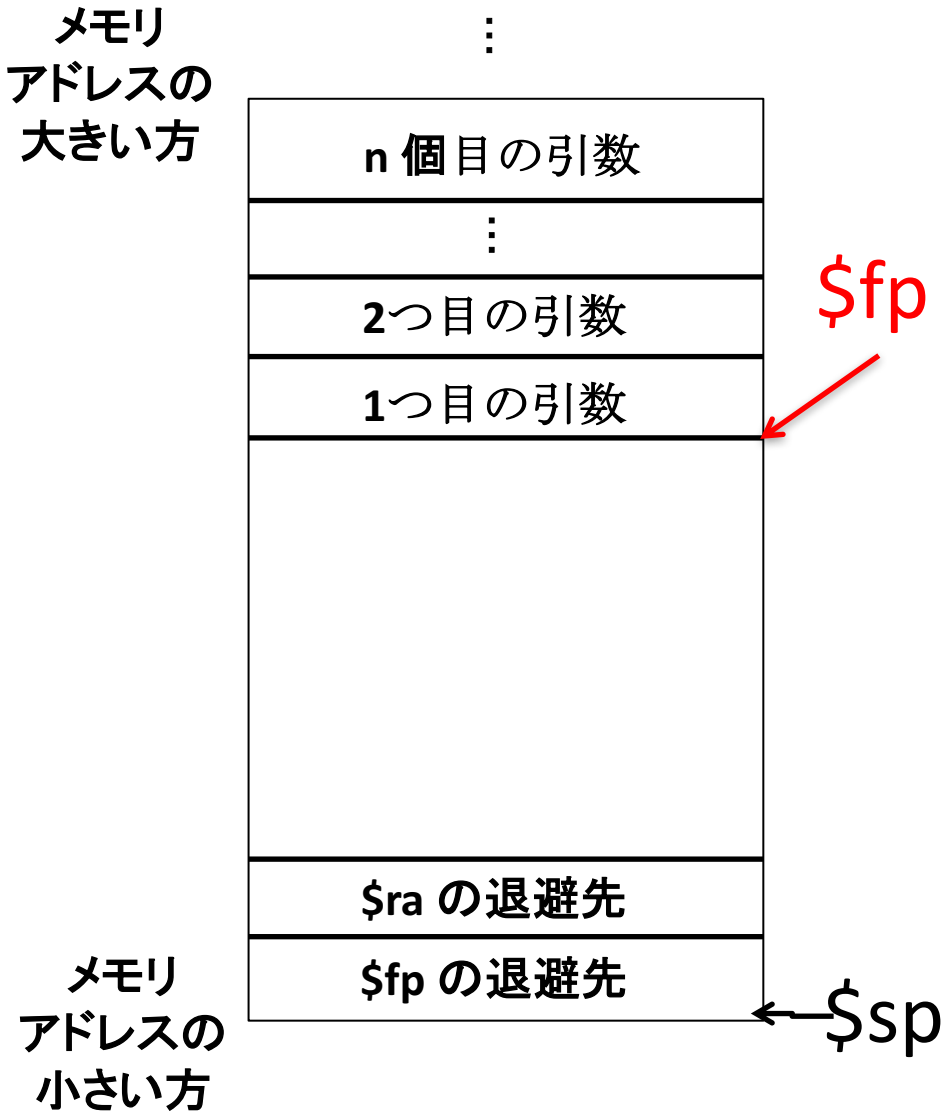

- 関数定義本体の頭
  - $sp を動かしてフレームを確保

# アセンブリ生成の拡張



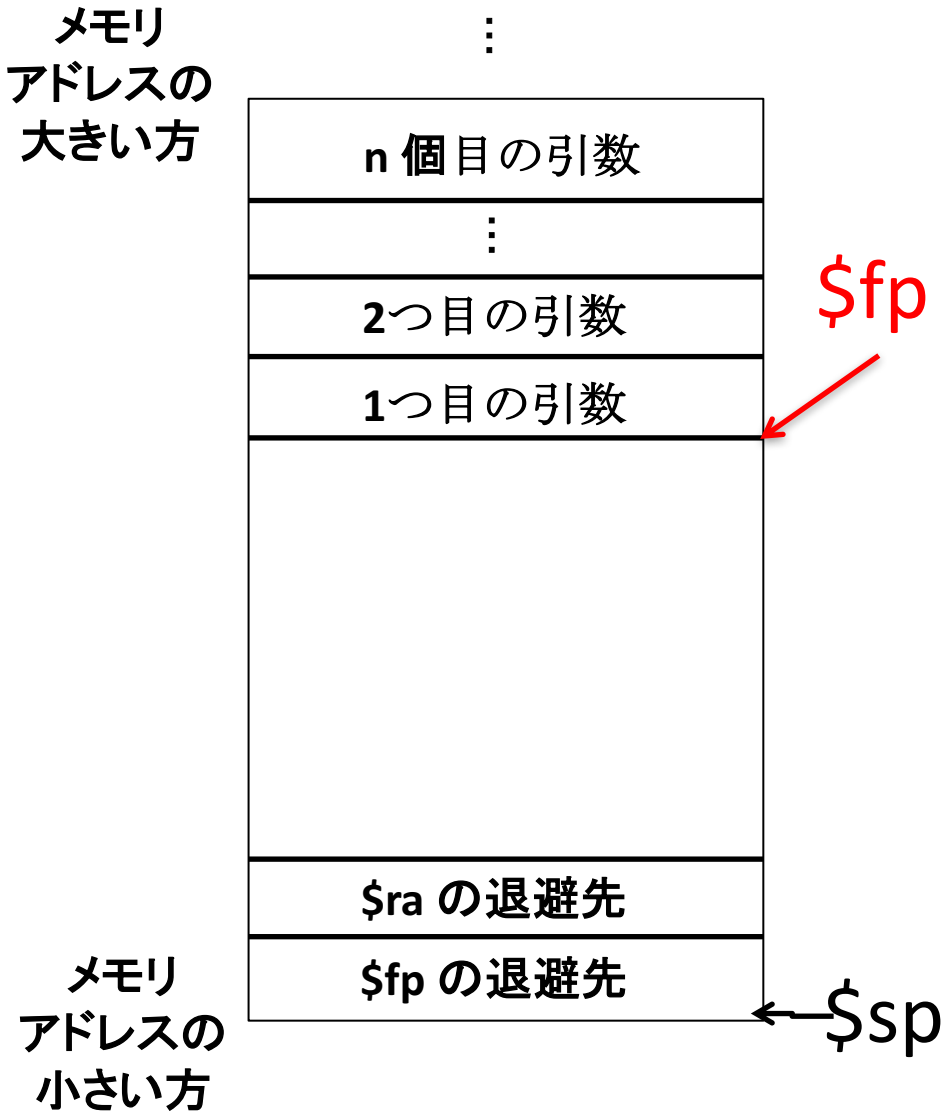

- 関数定義本体の頭
  - $sp を動かしてフレームを確保
  - $fp と $ra を退避

# アセンブリ生成の拡張



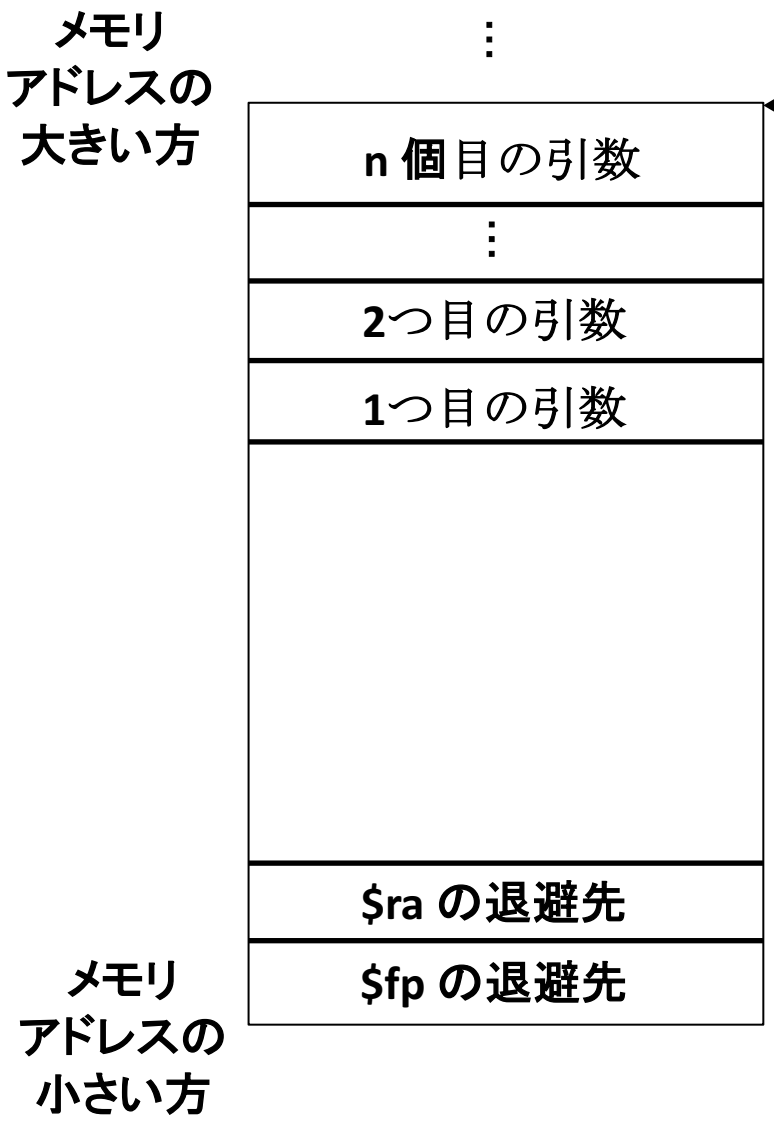

- 関数定義本体の頭
  - $sp を動かしてフレームを確保
  - $fp と $ra を退避
  - $fp をセット
    - $sp + (局所変数サイズ) + $ra のサイズ + $fp のサイズに

# アセンブリ生成の拡張



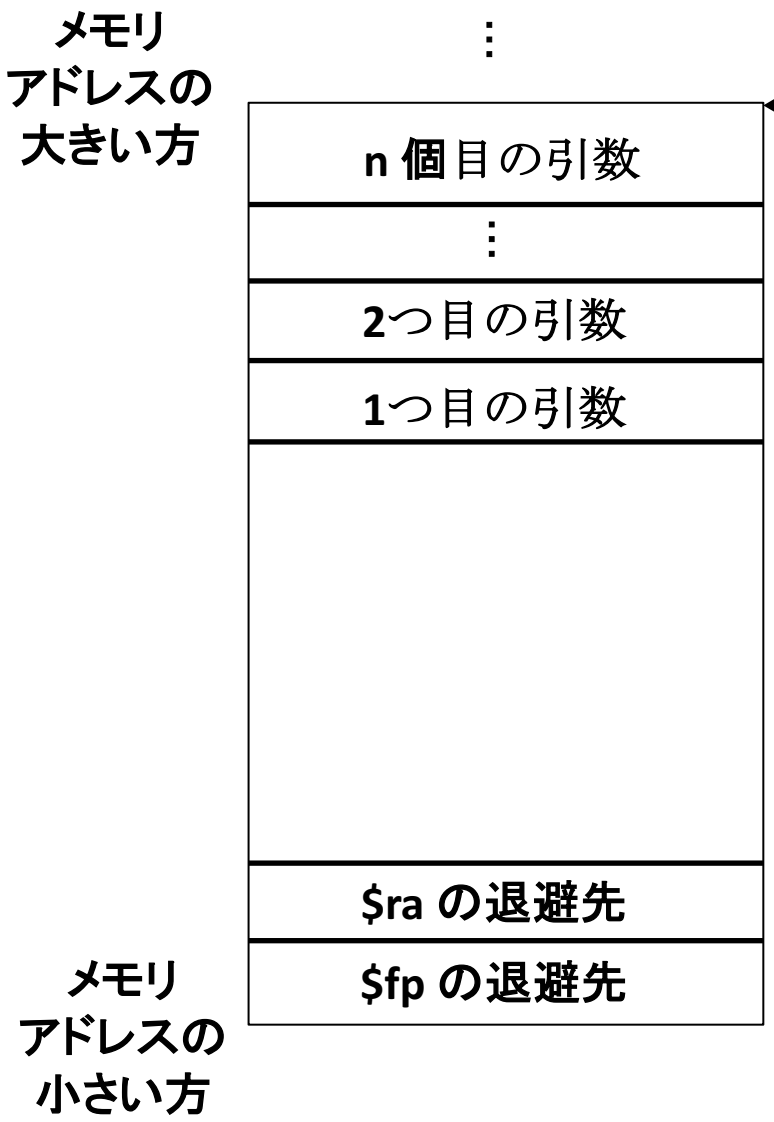

- return の直前

# アセンブリ生成の拡張




- return の直前
  - 退避しておいた $fp と $ra を復帰
  - $sp の値を元に戻す

# アセンブリ生成の拡張




- return の直前
  - 退避しておいた $fp と $ra を復帰
  - $sp の値を元に戻す
  - 返り値を $v0 にセット
  - jr 命令で $ra に設定されている場所に戻る
    - フレームの内容をクリアする必要はない

# Racket ではどう実装するか (ヒント)

- 関数定義先頭の操作は実装済
- return 直前の操作は restorecode 関数に実装されている
- callstmt
  - 引数を逆順で $sp の下にストアするコードを生成する再帰関数を実装

# 今まで説明したことに加えて普通のコンパイラで必要なこと

- 関数呼び出し前後
  - 呼び出し側で退避しなければならないレジスタ (caller-save registers) の退避と復帰
- 関数定義の先頭と return 直前
  - 呼び出され側で退避しなければならないレジスタ (callee-save registers) の退避と復帰

- どのレジスタが caller-save でどのレジスタが callee-save かは呼び出し規約で決まっている

# 最適化の入門の入門

# 最適化とは

- プログラムを**その意味を変えずに**より**効率のよい**プログラムに**変換**すること
  - 「その意味を変えずに」: いくら効率がよくなっても，コードの意味が変わるのは (普通は) ダメ
  - 効率のよい: 様々な効率のよさがありうる
    - 実行速度
    - メモリ効率
    - コードサイズ
    - …
  - 変換: 最適化は通常プログラム変換で行う

# 最適化はどう難しいか

- **プログラムの実行前**に**プログラムの実行に関する情報**を集めて最適化可能な場所を見つける必要

この情報が完璧に集められることはありえない

```
y = p;
if (y == 0) {
  x = 0;
} else {
  x = y;
}
print (x);
```

print(x) の直前ではx,yの初期値によらずx==pということが推論できれば

```
print (p);
```

# 最適化はどう難しいか

- **プログラムの意味を変えない**ことを保証するのは大変

```
x = y / z;
x = x * z;
print(x);
```

?

```
print (y);
```

# 最適化はどう難しいか

- **プログラムの意味を変えない**ことを保証するのは大変

```
x = y / z;
x = x * z;
print(x);
```

z == 0 だったら
変換元は
divide-by-zero エラーが起きるが
変換後は起きない

```
print (y);
```

# 最適化はどう難しいか

- **プログラムの意味を変えない**ことを保証するのは大変

```
y = p;
if (y == 0) {
  x = 0;
} else {
  x = y;
}
print (x);
```

もしxに勝手に別の値を代入するスレッドが並行に走っていたら

```
print (p);
```

# 最適化はどう難しいか

- すごい解析はコストがかかる

スパコン128台を1ヶ月回して解析した結果ここの命令が無駄なことが分かりました! といわれましても …

```
x = y;
sugoikansuu(&x);
choufukuzatsunakansuu(&y);
chottodakesugoikansuu(&x,&y);
print(y);
```

# 最適化はどう難しいか

- しかもあまり見合わないことが多い

こんな代入のコストは結局ハードウェアで行われる並列化とかで隠されちゃうから消えても意味ないよwwざまあwww

```
x = y;
sugoikansuu(&x);
choufukuzatsunakansuu(&y);
chottodakesugoikansuu(&x,&y);
print(y);
```

# これからやる最適化は…

- 難しいところには目をつぶる
  - マルチスレッドとかないからいろいろ変換しても安全
- どのくらい効くのかは保証しない
  - 字面上無駄そうな命令を削除したりする程度
  - 本当は実験して効果を確認する必要
- でも，もっとすごい最適化を勉強するときのベースにはなる (と思う)
  - なーんも知らんかったら論文は読めない
- 「プログラム解析」の一例として聞いてくれるのが良いかもしれない

# 取り扱う最適化

- 定数畳み込み (constant folding)
  - x = 3; y = x + 2; print(y); ➔ x = 3; y = 5; print(5);
- コピー伝播 (copy propagation)
  - y = z; x = y; print(x); print(z); ➔
    y = z; x = y; print(z); print(z);
- 無駄な代入命令の除去 (an instance of dead code elimination)
  - x = 3; y = 5; print(5); ➔ print(5);
  - x = y; y = z; print(z); print(z); ➔ print(z); print(z);
- レジスタ割り付け (register allocation)
  - (うまく例がかけなかった)

# 最適化は組み合わせるとすごい

- コピー伝播
  - y = z; x = y; print(x); print(z); ➔
    y = z; x = y; print(z); print(z);
- 無駄な代入命令の除去
  - y = z; x = y; print(z); print(z); ➔ print(z); print(z);

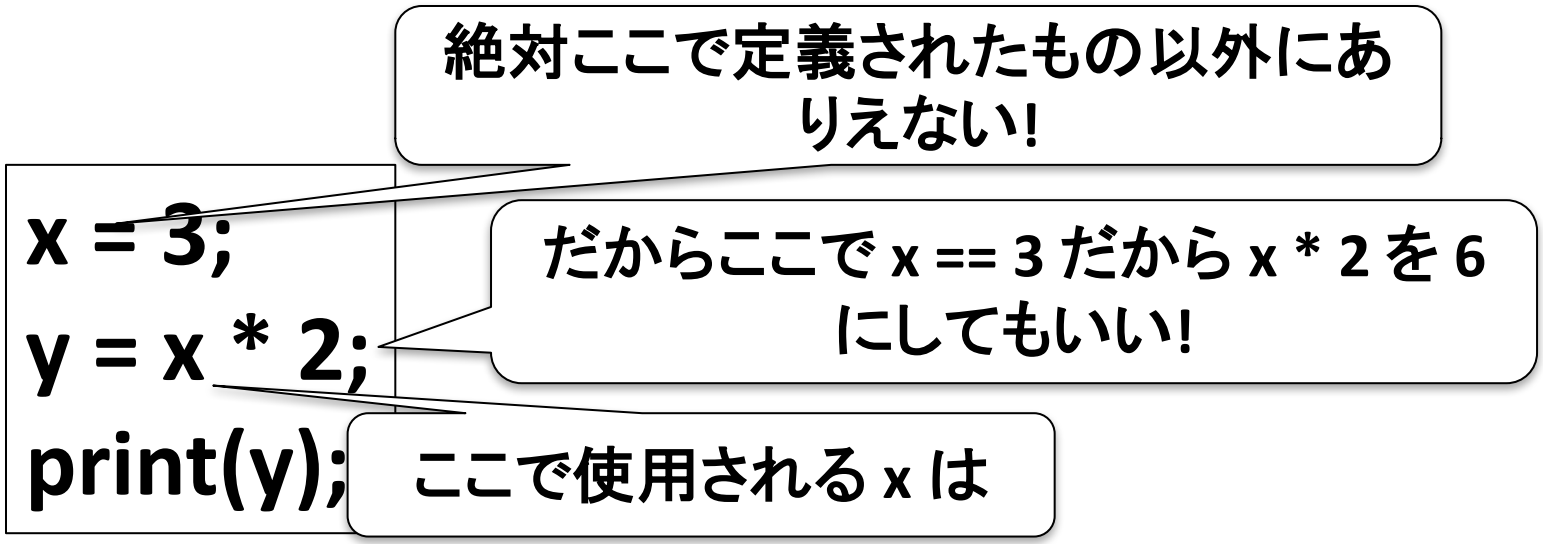
- いろんな最適化を繰り返すことですごくなる!!!!

# データフロー解析

最適化に必要な情報を集めよう

# 最適化に有用な情報

- データの流れに関する情報 (データフロー)



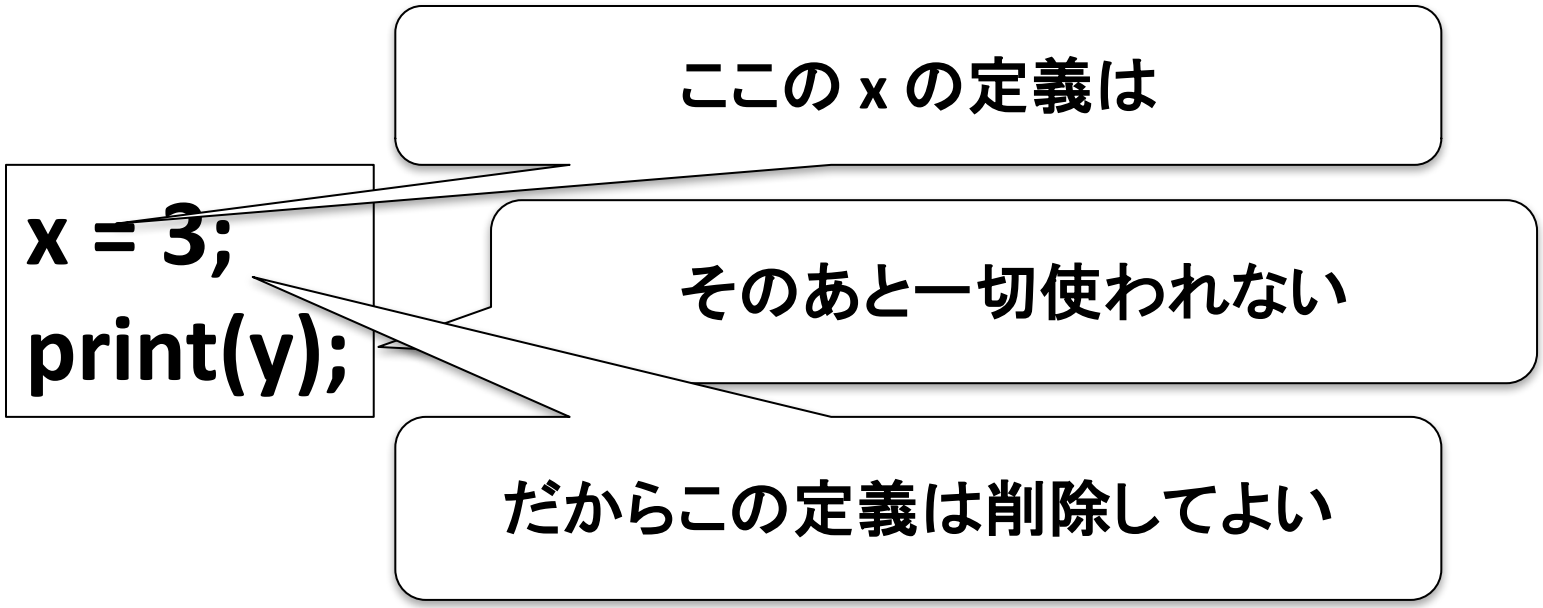
# 「定義」と「使用」

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
```

- x の「定義」≒xへの代入
  - 複数回代入文があると，複数の定義があらわれる
  - 本講義では異なる定義を行番号で表す (e.g., 1行目の x の定義)
- x の「使用」≒式での x への参照
  - 各々の使用がどの定義から来ているかが問題となる
    - 例: 2行目のxの使用は1行目の定義から来ている

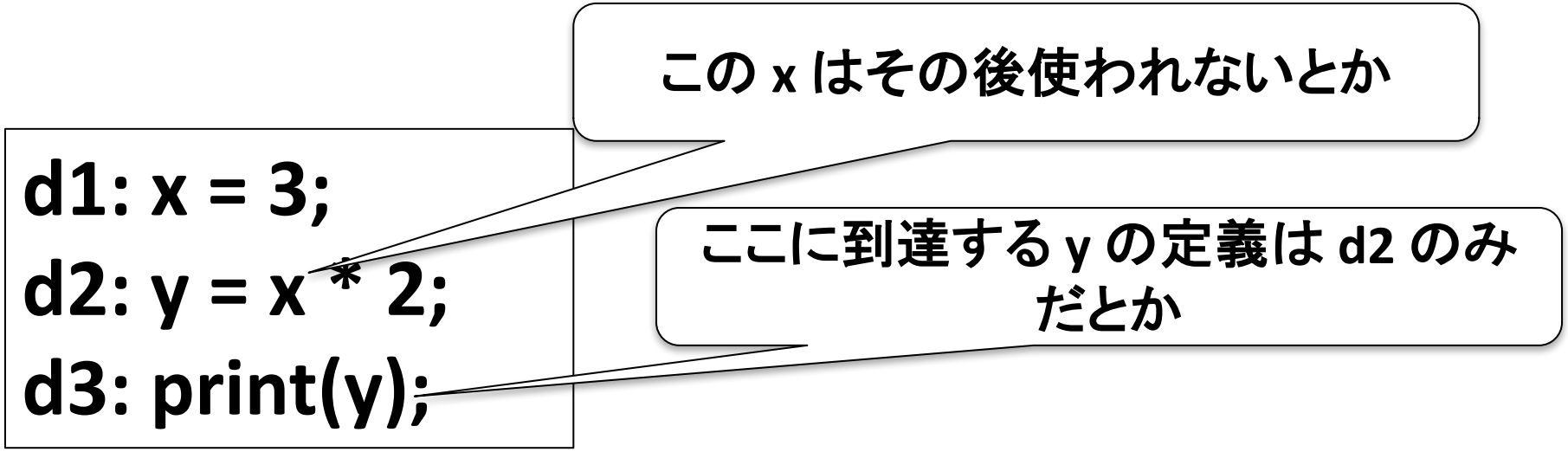
# 最適化に有用な情報

- データの流れに関する情報 (データフロー)



# データフロー解析

- データフローを解析する手法 (そのまんまや)




- このプログラムだけだと簡単そうに見えるが，プログラムが複雑になると意外と大変

# 典型的なデータフロー解析

- 到達可能定義解析 (reachable definition analysis)
  - プログラムのある地点で有効でありうる代入文の集合を求める
- 到達コピー解析 (reachable copy analysis)
  - プログラムのある地点で必ず有効なコピー文 x = y の集合を求める
- 生存変数解析 (live variable analysis)
  - 変数がどの区間で「生きている」かを求める

# 到達可能定義解析

- あるプログラムの行に到達しうる**定義** (i.e., **代入文**) の集合を求める解析

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
```

4行目に1行目の x の定義は到達しないが2行目の定義は到達しうる

```
1: x = 3;
2: y = x * 2;
3: if (x>=4) then goto 4 else goto 5
4: print(y); goto 6
5: x = 6
6:
```

6行目には1行目の x の定義は到達するかもしれないししないかもしれない (5 行目を通ったら 1 行目の x の定義は 6 行目に到達しない)

# プログラム解析とapproximation

- 実行前に行う解析では「かもしれない」状況を近似して扱う必要

```
1: x = 3;
2: y = x * 2;
3: if (x>=4) then goto 4
   else goto 5
4: print(y); goto 6
5: x = 6
6:
```

6行目には1行目の定義は到達するかもしれないししないかもしれない ➔

- 到達「し得る」定義を求めたいなら「到達する」と扱う
- 「絶対」到達する定義を求めたいなら「到達しない」と扱う

- 到達可能定義の解析では前者

# プログラム解析と approximation

- コンパイル時の解析では近似が必要なので不正確さが避けられない

```
1: x = 3;
2: y = x * 2;
3: if (x>=4) then goto 4
   else goto 5
4: print(y); goto 6
5: x = 6
6:
```

実際は else 節しか実行されないので 6 行目に 1 行目の定義は到達しないが，今日やる解析では「到達し得る」と解析される

# 典型的なデータフロー解析手法

- データフロー方程式に基づく手法
  1. 各文が**データフロー値 (dataflow value)** をどのように発生・消滅させるかを**伝達関数 (transfer function)** で表現
  2. 制御の流れにしたがってデータフロー値の間の制約を**データフロー方程式 (dataflow equation)** で表現
  3. データフロー方程式の解を**不動点反復 (fixed-point iteration)** で計算

# データフロー値

- 解析しようとする情報を表すデータ

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

- 例: 到達可能定義解析
  - データフロー値 (x, d): 変数 x に関する d 行目の定義
    - 左の例では (x, 1), (y, 2), (x, 3) の3つがありうる

# TODO: From here

# 到達可能定義問題の定式化

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

- INPUT: 行番号のついた中間命令で書かれたプログラム
- OUTPUT: 以下を満たす行番号からデータフロー値の集合への写像 R
  - プログラムの実行中に d' 行目の x の定義が d 行目に到達したならば $(x, d') \in R(d)$
  - 逆は成り立たなくてよい (一般には逆も成り立つデータフロー解析を作るのは不可能)

# 到達可能定義問題の定式化

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

- データフロー解析の解は複数ありうる
  - R(1) = R(2) = R(3) = R(4) = R(5) = {(x,1), (y,2), (x, 3)} も一応解の一つ
    - 「プログラムの実行中に d’ 行目の x の定義が d 行目に到達したならば…」なので到達しないのに (x, d’) ∈ R(d) でも良い
    - でも，この解からは有用な情報は得られない
  - できるだけ各 R(d) が小さくなる情報が有用
    - 「解析の精度が高い」と言う

# 典型的なデータフロー解析手法

- データフロー方程式に基づく手法
  1. 各文が**データフロー値 (dataflow value)** をどのように発生・消滅させるかを**伝達関数 (transfer function)** で表現
  2. 制御の流れにしたがってデータフロー値の間の制約を**データフロー方程式 (dataflow equation)** で表現
  3. データフロー方程式の解を**不動点反復 (fixed-point iteration)** で計算

# データフローの発生と消滅

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

- 到達定義解析では gen と kill という形で定義される
  - gen(d): d 行目で新たに生成される定義の集合
    - 1行目では {(x, 1)}
  - kill(d): d 行目で消去される定義の集合
    - 1行目では第一要素が x であるデータフロー値: {(x,1), (x, 3)}

# 伝達関数 (transfer function)

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

- 各文の直前と直後のデータフロー値の関係を示す関数
  - 各行 j で fj(X) = gen(j) ∪ (X - kill(j))
  - 前から後ろに計算するので前向き (forward) と言う

# 典型的なデータフロー解析手法

- データフロー方程式に基づく手法
  1. 各文が**データフロー値 (dataflow value)** をどのように発生・消滅させるかを**伝達関数 (transfer function)** で表現
  2. 制御の流れにしたがってデータフロー値の間の制約を**データフロー方程式 (dataflow equation)** で表現
  3. データフロー方程式の解を**不動点反復 (fixed-point iteration)** で計算

# データフロー方程式

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

- 文の間のデータフローの制約を定める式
  - d 行目直前のデータフロー値 IN(d), 直後のデータフロー値 OUT(d) の間の制約
    - IN(1) = ∅, OUT(1) = f(1)
    - IN(2) = OUT(1), OUT(2) = f(2)
  - 到達可能定義解析では一般に
    IN(d) = ∪_{d'はdの先行文} OUT(d')
    OUT(d) = gen(d) ∪ (IN(d) - kill(d))

# 典型的なデータフロー解析手法

- データフロー方程式に基づく手法
  1. 各文が**データフロー値 (dataflow value)** をどのように発生・消滅させるかを**伝達関数 (transfer function)** で表現
  2. 制御の流れにしたがってデータフロー値の間の制約を**データフロー方程式 (dataflow equation)** で表現
  3. データフロー方程式の解を**不動点反復 (fixed-point iteration)** で計算

# 不動点反復による データフロー方程式の解法

- IN(d), OUT(d) をそれぞれ $\varnothing$ とおく
- データフロー方程式にしたがって IN(d), OUT(d) を繰り返し更新する
- 値が変わらなくなったらそれが解

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = ∅
IN(2) = ∅
OUT(2) = ∅
IN(3) = ∅
OUT(3) = ∅
IN(4) = ∅
OUT(4) = ∅
IN(5) = ∅
OUT(5) = ∅

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = ∅
IN(2) = ∅
OUT(2) = ∅
IN(3) = ∅
OUT(3) = ∅
IN(4) = ∅
OUT(4) = ∅
IN(5) = ∅
OUT(5) = ∅

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**
**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = ∅
OUT(2) = {(y, 2)}
IN(3) = ∅
OUT(3) = {(x, 3)}
IN(4) = ∅
OUT(4) = ∅
IN(5) = ∅
OUT(5) = ∅

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**
**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = {(x, 1)}
OUT(2) = {(x, 1), (y, 2)}
IN(3) = {(y, 2)}
OUT(3) = {(x, 3)}
IN(4) = {(x, 3)}
OUT(4) = ∅
IN(5) = ∅
OUT(5) = ∅

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**
**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = {(x, 1)}
OUT(2) = {(x, 1), (y, 2)}
IN(3) = {(x, 1), (y, 2)}
OUT(3) = {(x, 3), (y, 2)}
IN(4) = {(x, 3)}
OUT(4) = {(x, 3)}
IN(5) = ∅
OUT(5) = ∅

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**
**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = {(x, 1)}
OUT(2) = {(x, 1), (y, 2)}
IN(3) = {(x, 1), (y, 2)}
OUT(3) = {(x, 3), (y, 2)}
IN(4) = {(x, 3), (y, 2)}
OUT(4) = {(x, 3)}
IN(5) = {(x, 3)}
OUT(5) = ∅

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**
**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = {(x, 1)}
OUT(2) = {(x, 1), (y, 2)}
IN(3) = {(x, 1), (y, 2)}
OUT(3) = {(x, 3), (y, 2)}
IN(4) = {(x, 3), (y, 2)}
OUT(4) = {(x, 3), (y, 2)}
IN(5) = {(x, 3)}
OUT(5) = {(x, 3)}

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**

**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = {(x, 1)}
OUT(2) = {(x, 1), (y, 2)}
IN(3) = {(x, 1), (y, 2)}
OUT(3) = {(x, 3), (y, 2)}
IN(4) = {(x, 3), (y, 2)}
OUT(4) = {(x, 3), (y, 2)}
IN(5) = {(x, 3), (y, 2)}
OUT(5) = {(x, 3)}

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**

**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の実行例

```
1: x = 3;
2: y = x * 2;
3: x = 4;
4: print(y);
5:
```

IN(1) = ∅
OUT(1) = {(x, 1)}
IN(2) = {(x, 1)}
OUT(2) = {(x, 1), (y, 2)}
IN(3) = {(x, 1), (y, 2)}
OUT(3) = {(x, 3), (y, 2)}
IN(4) = {(x, 3), (y, 2)}
OUT(4) = {(x, 3), (y, 2)}
IN(5) = {(x, 3), (y, 2)}
OUT(5) = {(x, 3), (y, 2)}

**IN(d) = ∪ _{d'はdの先行文} OUT(d')**

**OUT(d) = gen(d) ∪ (IN(d) - kill(d))**

# 不動点反復の性質

- 不動点反復が止まったら，必ずフロー方程式の解が得られる
  - 停止条件は「フロー方程式の解が見つかったとき」に等価
- 不動点反復は有限ステップで止まる
  - 一度 IN(d), OUT(d) に加わったデータフロー値が消えることはない
  - 可能なデータフロー値は有限個
- 不動点反復で得られる解は，フロー方程式を満たす解の中で最小
  - 完備半順序 (cpo) の理論を使うと出てくる

# 様々なデータフロー解析

- データフロー値，伝達関数，フロー方程式を変えることにより，異なるデータフロー解析が得られる
  - どの解析でも解は不動点反復で得られる

# 到達コピー解析

- 求める情報: 各文の直前で必ず有効なコピー文の集合
- データフロー値: x = y という形の文の集合
- 伝達関数
  - gen(d : x = y) = { x = y }
  - gen(d : x = e) = {} (e is not a variable)
  - kill(d : x = e) = { z = w | x is not x nor w }
  - fd(X) = gen(d) ∪ (X - kill(d))
- フロー方程式
  - IN(d) = ∩_{d' は d の先行文} OUT(d')
    - 「必ず」到達するコピー文を求めるので∩になる
  - OUT(d) = fd(IN(d))

# 生存変数解析

- 求める情報: 各文の直前で「これ以降定義される前に使用される可能性がある変数」の集合
- データフロー値: 変数の集合
- 伝達関数
  - def(d : x = e) = {x}
  - use(d : x = e) = e に含まれる変数の集合
  - fd(X) = use(d) ∪ (X - def(d))
- フロー方程式
  - IN(d) = fd(OUT(d))
    - 他の解析とはフローが逆であることに注意
    - 「これ以降…」を求めるために逆向きになる
  - OUT(d) = ∪_{d' は d の後続文} IN(d')

# 基本ブロック

- 制御の分岐や合流がない文の塊
  - 伝達関数やデータフロー方程式は文単位ではなく基本ブロック単位で与えることが多い

基本ブロック B

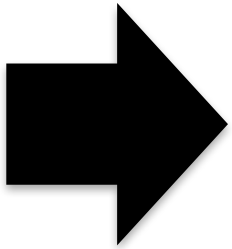
**1: x = y + 3**
**2: z = 2**
**3: x = p + 4**
**4: k = f(x, y)**

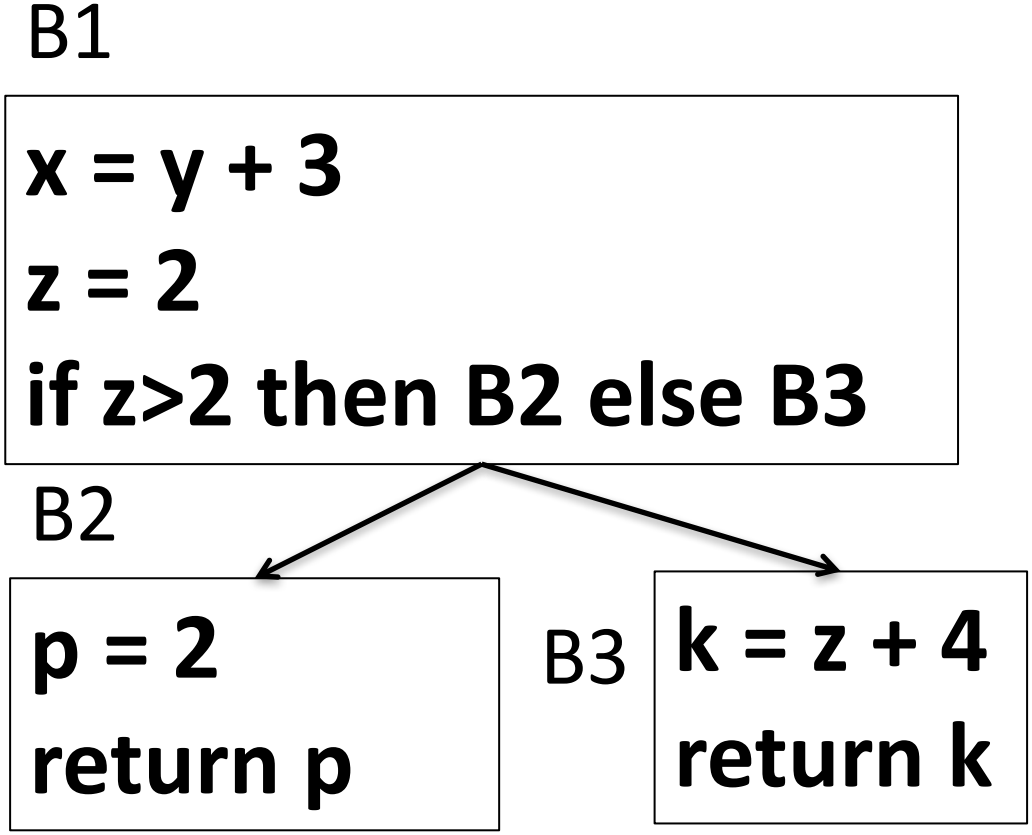
- gen(B) = {(z, 2), (x, 3), (k, 4)}
- kill(B) =
  (x, 3) 以外の (x, _) の形 ∪
  (z, 2) 以外の (z, _) の形 ∪
  (k, 4) 以外の (k, _) の形

# データフローグラフ

- 基本ブロックを節点，制御の流れを辺とするグラフ
  - フロー方程式を立てる際に使用

```
x = y + 3;
z = 2;
if z > 2 then
  p = 2;
  return p;
else
  k = z + 4;
  return k;
```





# データフロー解析と最適化

集めた情報をどう最適化に活かすか

# 取り扱う最適化

- 定数畳み込み (constant folding)
  - x = 3; y = x + 2; print(y); → x = 3; y = 5; print(5);
- コピー伝播 (copy propagation)
  - x = y; y = z; print(x); print(z); →
    x = y; y = z; print(z); print(z);
- 無駄な代入命令の除去 (an instance of dead code elimination)
  - x = 3; y = 5; print(5); → print(5);
  - x = y; y = z; print(z); print(z); → print(z); print(z);
- レジスタ割り付け (register allocation)
  - (うまく例がかけなかった)

# 定数畳み込み

- コンパイル時に分かっている定数をあらかじめ計算する最適化
  - 実行時に行う計算を減らせる

```
x = y + 3;
z = 2;
if z > 2 then
  p = 2;
  return p;
else
  k = z + 4;
  return k;
```

**ここの z は絶対に 2 だから**
**この命令は k = 6 と置き換えて良い**

# 定数畳み込み

- コンパイル時に分かっている定数をあらかじめ計算する最適化
  - 実行時に行う計算を減らせる

```
x = y + 3;
z = 2;
if z > 2 then
  p = 2;
  return p;
else
  k = z + 4;
  return k;
```

**ここの条件は常に false になるので then 節を除去可能**

# 到達定義解析を用いた定数畳み込み

- d : x = e において以下の条件が成り立てば e を計算してしまって良い
  - e 中の変数が { x1, …, xn }
  - d の直前に到達する xi の定義がそれぞれ di : xi = <定数> のみ
    - d に到達する xi の定義が2つ以上ある場合は, どの定義が到達するかわからない

# さっきの例では…

```
x = y + 3;
d1 : z = 2;
if z > 2 then
  p = 2;
  return p;
else
  k = z + 4;
  return k;
```

**ここに到達する z の定義は**
**d1 : z = 2 のみ**

# データフロー解析に基づく不要代入文除去

- d : x = e において以下の条件が成り立てば d を除去してよい
  - e が関数呼び出しでない
    - 関数呼び出しが print 等を含むときは除去すると振る舞いが変わってしまう
    - 一般には「代入文が副作用を発生させないとき」
  - d の直後で x が生存していないとき
    - x は結局使われることがないので代入文を除去して良い

# 到達コピー解析に基づくコピー伝播

- d : x = e において，次の条件が成り立てば e 中の y を z で置き換えて良い
  - d の直前に到達する y を左辺に持つコピー文が y = z のみであるとき

# レジスタ割り当て

# レジスタ割り当て (register allocation)

- レジスタを記憶領域の一部として使うこと
  - レジスタは主記憶よりずっと速くアクセスできる

# レジスタ割り当ての難しさ

- レジスタの数は限られている
  - 速い記憶領域は値段が高い

# 基本的なレジスタ割り当ての方法

- 同時に生存している変数には異なるレジスタを割り当てる
- レジスタの数が足りない場合にはスタック上に記憶領域を確保 (spill)

# 干渉グラフ (interference graph)

- 変数を節点とし，同時に生存する変数間を辺で結んだ**無向**グラフ
  - 生存変数解析から作ることができる

- k 個のレジスタを変数に割り当てる問題は干渉グラフのk-彩色問題 (graph coloring problem) と等価

# グラフ彩色問題 (graph coloring problem)

- 入力: 無向グラフ (V, E), 自然数 k
- 出力: ρ(n) ≠ ρ(m) if n ≠ mを満たす V から {0, 1, …, k-1} への写像 ρ が存在すれば YES, 存在しなければ NO
  - 辺で結ばれた2つの節点が異なる色になるように各節点を k 色で塗れるか? という問題
  - NP 完全問題の一つ
    - P ≠ NP ならば多項式時間アルゴリズムが存在しない
  - 平面グラフのときはいわゆる四色問題

# これ以降の講義

- 板書で授業を進めるのでノートを持参のこと
- 内容: 字句解析と構文解析
  - 期末試験はここから出題