計算機言語と処理系

久野　靖*

1998.8.3〜6

0　はじめに

この科目は 4 日間の集中講義で、プログラミング言語で書かれたものがどうやって計算機のハードウェア上で動くようになるのかを学ぶものです。もっとありていにいえばコンパイラの仕組み、という感じですが、コンパイラそのものだけでなくその周辺の話題も取り上げることになります。

内容的には、プログラミングに関する知識はあまり必要としません。アルゴリズムは沢山出て来ますが、新たに学ぶのであって、前提として何かを必要とするということはあまりないでしょう。

それで、4 日間に渡ってお話ばかり聞かされても身につかないでしょうから、4 日間それぞれを午前と午後の合計 8 セクションに分け、各セクションで 1 つずつテーマを決め取り上げ、講義と併せて簡単な演習をやります。その演習の結果をすべて提出していただき、出席点+内容の採点に基づき成績をつけます。だから試験やレポートはなしです。

参考書「言語プロセッサ」を購入してくださった方もいらっしゃると思いますが、各回の内容はその本とだいたい対応していて、一応次のように予定しています。

- □ 1AM. 言語処理系の位置づけと概観 (1 章)
- □ 1PM. 形式言語 (2 章)
- □ 2AM. 字句解析 (3 章)
- □ 2PM. 構文解析 (4 章)
- □ 3AM. 意味解析 (5〜7 章)
- □ 3PM. 実行時環境 (8 章)
- □ 4AM. コード生成 (9、11 章)
- □ 4PM. 最適化 (10 章)

ただし 4PM については多少「総合問題」ぽくやるかも知れません。いずれにせよ、参考書を購入して頂いた方は、目を通してから来てくださるとスムーズに理解できると思います。(精読する必要はありません。)

*筑波大学大学院経営システム科学専攻

1　言語処理系の位置づけと概観 (1AM)

1.1　プログラミング言語と言語処理系

□ プログラミング言語とは?　言語処理系とは?

□ 計算機による情報処理を簡略化すると:

□ 入力は普通「文字」を使うのが多い

□ では日本語や英語?　→ 自然言語は処理するのがいろいろと難しい

□ ではどうするか?　→ 計算機言語

- 計算機言語:　情報交換のため人工的に構成
- 特定目的言語 vs プログラミング言語 (汎用)
- 言語処理系:　言語の動作を計算機で実行させるためのソフト

□ プログラミング言語にもいろいろある

- 機械語、アセンブリ言語:　低水準言語 (機械依存)
- 高水準言語 (機械独立) → 言語記述の各要素が機械の動作を抽象的に記述

```
x = i = 0;
while(++i <= 100)
    x += i;
```

- 各種の命令…増やす、減らす、ジャンプ
- 各種の場所…レジスタ、主記憶

1.2　計算機ハードウェアと機械語

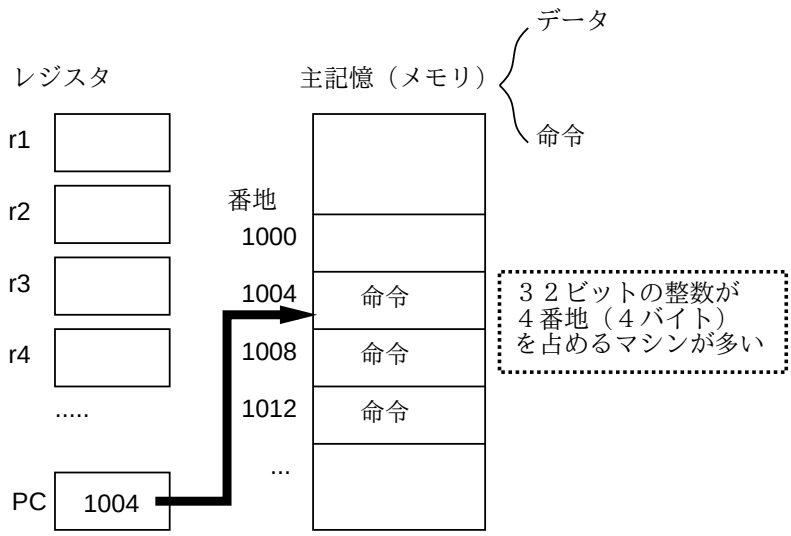
□ 計算機の心臓部:　CPU(中央処理装置) とメモリ (主記憶)

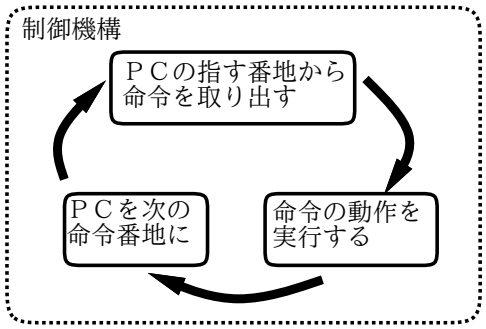
□ CPU: レジスタ群、演算装置、制御装置

□ 主記憶:　プログラム (命令のならび) とデータを保持

□ CPU の動作:　次の繰り返し

- 1.　PC(プログラムカウンタ) の指す番地から命令を持って来る
- 2.　持って来た命令に応じて動作を実行
- 3.　PC を次の命令の番地に進める (ジャンプ命令では飛び先に設定する)





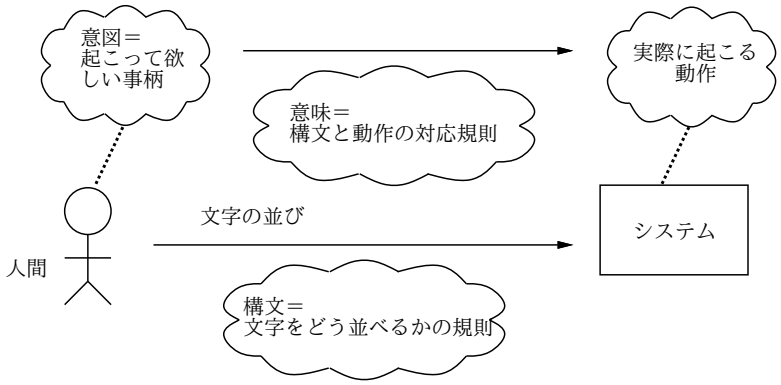

```
x = i = 0;
while(++i <= 100)
    x += i;
```

というコードを「長崎 1 号」という (空想上の) マシンのコードにすると:

```
3000:  loadi  r1,0        ←レジスタ 1 を 0 に
   4:  store  r1,1000     ← 1000 番地 (x に対応) を 0 に
   8:  store  r1,1004     ← 1004 番地 (i に対応) を 0 に
  12:  loadi  r3,1        ←レジスタ 3(定数 1 に対応) を 1 に
  16:  loadi  r4,100      ←レジスタ 4(定数 100 に対応) を 100 に
  20:  load   r1,1004     ← i をレジスタ 1 に
  24:  add    r1,r3       ← 1 を足し込む
  28:  store  r1,1004     ←元の i に格納
  32:  bgt    r1,r4,1032  ← i > 100 なら 3032 番地へ
  36:  load   r2,1000     ← x をレジスタ 2 に
  40:  add    r2,r1       ← x に i を足し込む
  44:  store  r2,1000     ←元の x に格納
  48:  b      3020        ← 3020 番地へ飛ぶ
3032:
```

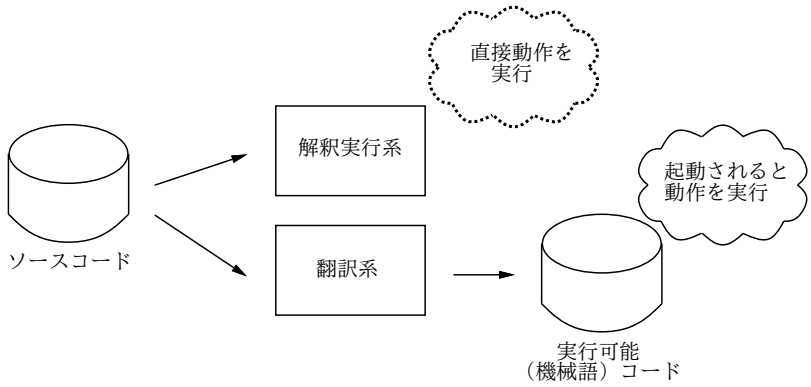
1.3 言語の構文と意味

- □ 構文: 書き方の規則（形式言語）
- □ 意味: 構文と動作の対応（自然言語）



1.4　プログラミング言語処理系

- □ 解釈実行系（インタプリタ）←実行系
- □ 翻訳系（コンパイラ）←変換系

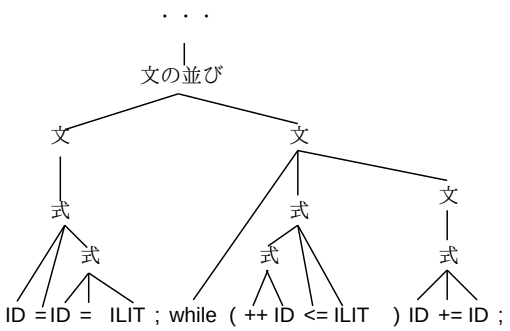


1.5　コンパイラの諸成分

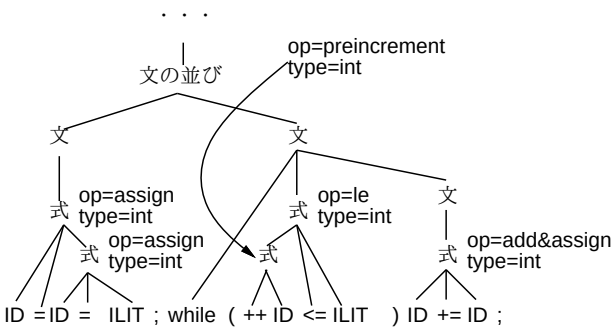
- □ コンパイラ → いくつもの成分に分けることができる
- □ コンパイラの成分 →字句解析、構文解析、意味解析、コード生成
- □ 字句解析 → ソースファイルを「かたまり」に分解

```
<keyword, "int">            <delimiter, "(">
<identifier, id# = 1>       <delimiter, "++">
<delimiter, "=">            <identifier, id# = 2>
<intliteral, value = 0>     <delimiter, "<=">
<delimitar, ",">            <intliteral, value = 100>
<identifier, id# = 2>       <delimiter, ")">
<delimitar, "=">            <identifier, id# = 1>
<intliteral, value = 0>     <delimitar, "+=">
<delimiter, ";">            <identifier, id# = 2>
<keyword, "while">          <delimiter, ";">
```

- □ 構文解析 → 「かたまり」の構造を決める



□ 意味解析 → 構文で表せない情報を抽出する



□ コード生成 → 目的コードを作り出す

- 目的コード (機械語/アセンブリ言語) の例については前述。

□ 記号表 → 各種の名前に関する情報を保持

名前	ID#	型	種別	番地
"x"	1	integer	局所変数	1000
"i"	2	integer	局所変数	1004

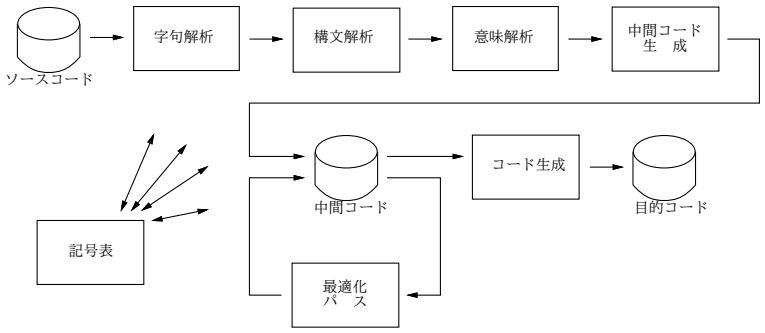
1.6 コンパイラのフェーズと分類

□ これらが順番に実行されるわけだが…

- まず最初のを全部やってしまって、次に 2 番目の処理を全部やってしまって…、というわけじゃない。
- フェーズ (段): 各構成成分
- パス: いくつかのフェーズのまとまり (プログラム)
- コンパイラ: パスの集まり

□ また別の分類方法として…

- 最適化コンパイラ: より高度な処理



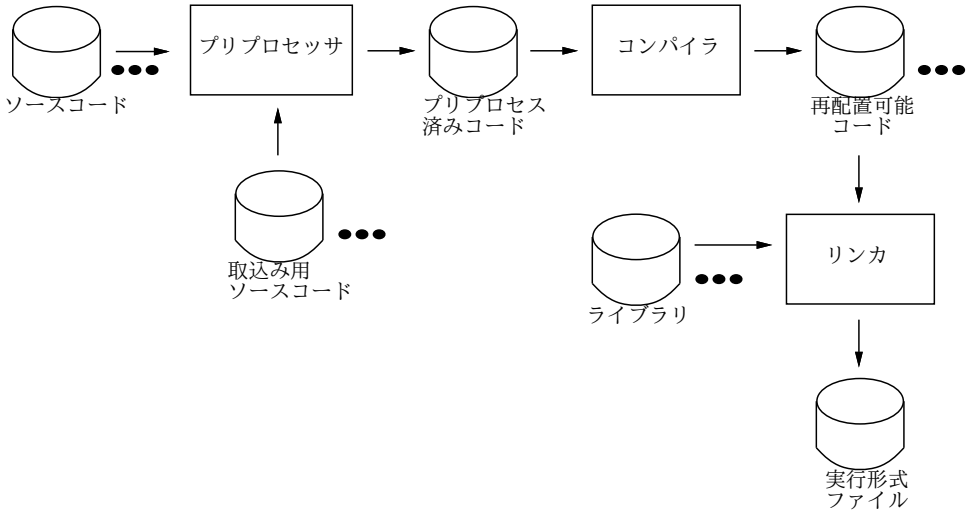
- フロントエンド (解析部) vs バックエンド (生成部)

1.7　コンパイラの周辺

□ プリプロセサ → 定数やマクロといった前段階の置き換えをやる

□ リンカ、ローダ → 別々に翻訳したものをまとめる

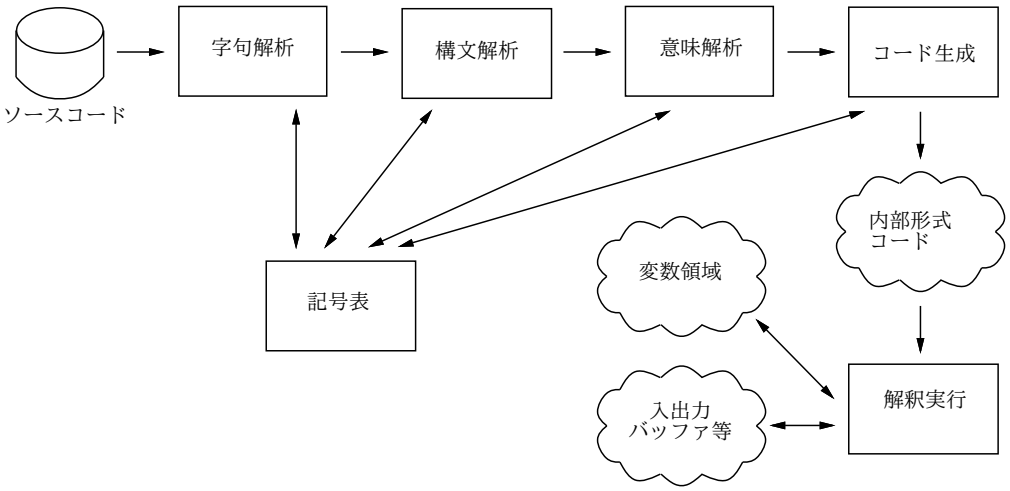
□ コンパイル&ゴー処理系 → コンパイルしたらすぐに実行



1.8　インタプリタの諸成分

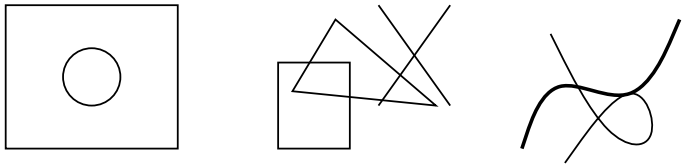
□ インタプリタ:　解析部は同じ

□ コンパイラインタプリタ、仮想機械



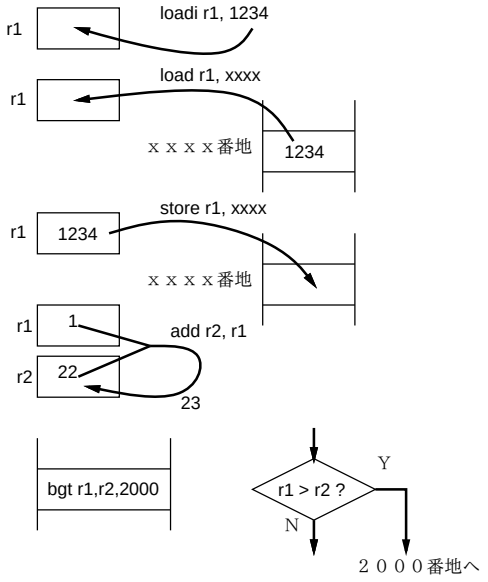
1.9 演習問題 (1AM)

□ [1-1.] あなたは、英語のアルファベットと数字だけを使って図を伝達しなければならない羽目に陥ったものとする。伝達のための言語を設計せよ。伝達すべき図形の例をあげておく (難しいと思う人は、3番目のは伝達できなくてもよい)。



□ [1-2.] 仮想的な計算機「長崎 1 号」は、レジスタ r1〜r15 とメモリ (1 語は 4 番地ぶん) を持ち、機械語命令は次のものだけであるような簡潔な計算機である。

```
load   rx, 番地   : レジスタ rx(x は 1〜15) に指定番地の内容を転送
loadi  rx, 定数   : レジスタ rx に定数を転送
store  rx, 番地   : レジスタ rx の内容を指定番地に格納
add    rx, ry     : レジスタ rx の内容に ry の内容を足し込む
sub    rx, ry     : レジスタ rx の内容から ry の内容を引く
mul    rx, ry     : レジスタ rx の内容を ry の内容倍する
b      番地           : 指定した番地に飛ぶ
beq    rx, ry, 番地   : rx==ry なら指定番地に飛ぶ
bne    rx, ry, 番地   : rx!=ry なら指定番地に飛ぶ
bgt    rx, ry, 番地   : rx>ry なら指定番地に飛ぶ
bge    rx, ry, 番地   : rx>=ry なら指定番地に飛ぶ
blt    rx, ry, 番地   : rx<ry なら指定番地に飛ぶ
ble    rx, ry, 番地   : rx<=ry なら指定番地に飛ぶ
```



□ 次の命令列を、1000 番地と 1004 番地に適当な数値を入れてから「長崎 1 号になって」実行してみよ。また、この命令列は何をしているのかも述べよ。

```
3000:  load   r1, 1000
3004:  store  r1, 1008
3008:  load   r2, 1004
3012:  bge    r1, r2, 3020
3016:  store  r2, 1008
3020:
```

□ [1-3.]　次の C 言語の断片

```
x = i = 1;
while(++i <= 5)
    x *= i;
```

を長崎 1 号の機械語 (のアセンブラ表記) に変換するとしたら、どのようになるか。ただし x は 1000 番地、i は 1004 番地に割り当てるものとし、プログラムは 3000 番地から入れるものとする。

□ [1-4.]　できたコードを「長崎 1 号になって」実行してみよ。

2 形式言語 (1PM)

プログラミング言語の「構文」(許される形) をきちんと定めるための道具が「形式言語」である。以下ではその概念を 1 つずつ学んでいく。

2.1 形式言語の諸定義

- □ V: アルファベット (有限個の記号から成る空でない集合)
- □ 語: V の要素を並べて得られる記号列
- □ V*, V+ … 0 個以上並んだもの、1 個以上の並んだもの
- □ V* = V+ ∪ { ε }
- □ 例えば V = {a, b} のとき, V* = { ε , a, b, aa, ab, ba, bb, aaa, aab, abb, baa, … }
- □ 言語 L : V* の任意の空でない部分集合
- □ 文: L の要素

2.2 生成文法

- □ T, N: 有限な記号の集まり
- □ P を: 「$\alpha \rightarrow \beta$」 ($\alpha \in$ (T ∪ N)+, $\beta \in$ (T ∪ N)*)
- □ S ∈ N : 出発記号 (「プログラム」)
- □ G = (T,N,P,S) を生成文法とよぶ
- □ T の要素: 終端記号 (言語の要素)
- □ N の要素: 非終端記号 (説明のための概念)
- □ u = x α y, v = x β y かつ $\alpha \rightarrow \beta \in$ P のとき u ⇒ v と書く
- □ u ⇒ v ⇒ w ⇒ … ⇒ z のとき u ⇒+ z と書く。0 回なら ⇒*
- □ L(G) = { x ∈ T* | S ⇒* x }

2.3 生成文法のいくつかの例

- □ 以下しばらく、英大文字∈ N、英小文字∈ T、出発記号は S とする。
- □ 例: P = { S → a S, S → ε } だと、生成される言語は?
- □ S ⇒ ε , S ⇒ a S ⇒ a、S ⇒ a S ⇒ a a S ⇒ a a、…だから
- □ L(G) = { ε , a, aa, aaa, aaaa, ... }

2.4 生成文法のタイプ

- □ タイプ 0 文法: もっとも一般的な場合.
- □ タイプ 1 文法: 左辺の記号列の長さは右辺の記号列の長さ以下
  - m A n → m t n, A ∈ N, t ∈ (N ∪ T)+, m,n ∈ (N ∪ T)*
- □ タイプ 2 文法: 各生成規則の左辺が 1 個の非端記号

- A → t, A ∈ N, t ∈ (N ∪ T)*

□ タイプ3文法: 各生成規則が次の形

- A → a または A → aB, A,B ∈ N, a ∈ T

□ タイプ1: 文脈依存文法、タイプ2: 文脈自由文法、タイプ3: 正規文法

□ タイプ0→1→2→3と進むにつれ、制約は強くなるが、計算機で取り扱いやくすなる。

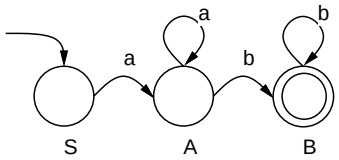
2.5 文法と言語の認識/解析

□ T+ の要素 t が L(G) に属すかどうか判別 (認識器)

□ さらに構造の決定 (解析器、パーザ)

□ たとえば、次の文法を認識する認識機を作ってみる

- P = {S → a A, A → a A, A → b B, B → b B, B → ε }



- このグラフの使い方… 記号に対応した矢印をたどって行き、最後に◎で終れば「OK」

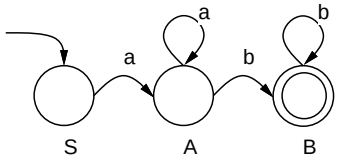
□ これをCのプログラムにするのは簡単。

```
main() {
  int c = getchar();
S:
  if(c == 'a') { c = getchar(); goto A; }
  goto error;
A:
  if(c == 'a') { c = getchar(); goto A; }
  if(c == 'b') { c = getchar(); goto B; }
  goto error;
B:
  if(c == 'b') { c = getchar(); goto B; }
  if(c == '\n') { goto success; }
  goto error;
error:
  printf("error!\n"); exit(1);
success:
  printf("success!\n"); exit(0);
}
```

- 「次の文字」を常に変数 c に入れておく (1文字の先読み)。
- すべての○、◎にラベルをつける。
- それぞれの箇所で、次の文字に応じて1文字読み進んで矢印の行き先に対応するラベルへ goto。
- ◎で改行が来たら成功。
- それ以外はすべて失敗。

2.6 文法と言語の認識/解析

□ P = {S → a A, A → a A, A → b B, B → b B, B → ε }



□ このグラフの作り方…(任意の正規文法に対してこれを作ることができる)

- 大文字 (非端記号) ごとに○を用意。
- X → ε であるような X は○のかわりに◎。
- X → a Y という規則があったら、X の○から Y の○へ「a」の矢線。
- X → a という規則があったら、X の○から無名の◎へ「a」の矢線。

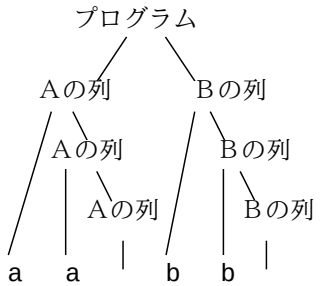
2.7 文脈自由文法とその記法

プログラム言語の構文定義には、文脈自由文法がもっとも多く使われる (理由: 記述能力と処理の簡単さのバランスがちょうどよい)。

□ 文脈自由文法→ BNF(Backus Naur Form ないし Backus Normal Form) で表記。BNF では「→」の代わりに「::=」というのを使う。また左辺が共通の場合に「|」(または) でくくり出して書く。例:

```
プログラム ::= A の列 B の列
A の列 ::= a A の列 | ε
B の列 ::= b B の列 | ε
```

□ 文脈自由文法で、ある記号列と文法との対応を表すのには、構文木 (parse tree) を使う。構文木では、1 つの規則適用ごとに 1 段ずつ (右辺の記号の数ずつ) 枝を延ばして行く。たとえば a a b b という列と文法の対応を示す構文木は次の通り:



構文木を書く代わりに、導出列を書くことでも構造を表せる。ただし、2 つ以上の置き換え箇所があったとき、どれを先に置き換えるかによって列が変わって来る。次の 2 つの書き方なら一意的に決まる。

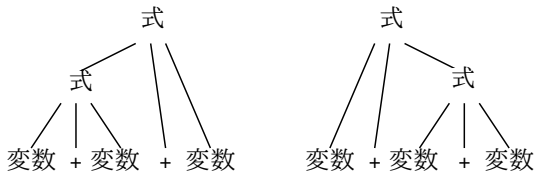
□ 最左導出 --- 常に左のものを先に置き換える

- プログラム ⇒ A の列 B の列 ⇒ a A の列 B の列⇒ a a A の列 B の列 ⇒ a a B の列 ⇒ a a b B の列⇒ a a b b B の列 ⇒ a a b b

□ 最右導出 --- 常に右のものを先に置き換える

- プログラム ⇒ A の列 B の列 ⇒ A の列 b B の列⇒ A の列 b b B の列 ⇒ A の列 b b ⇒ a A の列 b b ⇒ a a A の列 b b ⇒ a a b b

□ しかし、場合によっては構文木が 2 通り以上できてしまうような文法になることがある: 「あいまいな文法」という。たとえば「式 ::= 式 + 式 | 変数」という規則を「変数 + 変数 + 変数」にあてはめると次の 2 とおりの木ができる (どちらがより適切か?)。



2.8 演習問題 (1PM)

□ [2-1.] 次の生成文法により定義される言語はどんなものか。いくつか実際に生成してみると分かる。

- a. P = {S → a S, S → b S, S → c }
- b. P = {S → A c B, A → a A, A → ε , B → b B, B → b}
- c. P = {S → a S B C, S → a B C, C B → B C, a B → a b, b B → b b, b C → b c, c C → c c }

□ [2-2.] 次のような言語を定義する生成文法 (規則群) を書け。

- a. 「abc」が 1 回以上繰り返し並んだもの。i.e. abcabcabc
- b. a がいくつか並び、続いて b が a と同じ個数並んだもの。i.e. aaabbb
- c. 「1」と四則演算の記号 + - * / が交互に並んだもの。i.e. 1 + 1 * 1
- d. 上記に (余分でもいいから) 正しくかっこを挿入したもの。i.e. (1 + (1)) * (1 / 1)

□ [2-3.] 次の正規言語に対してまずどんな言語か考え、次に○と◎のグラフを作って、うまくその言語に対する認識器になっていることを確認せよ。

- a. P = {S → a S, S → b S, S → c }
- b. p = {S → a A, A → b B, B → c S, B → c }

□ [2-4.] 次の文法を考える

```
文 ::= if 文 | 代入文
if 文 ::= if ( 条件 ) 文
条件 ::= 式 == 式
代入文 ::= 変数 = 式 ;
式 ::= 変数 | 定数 | 式 + 式 | ( 式 )
```

ここで次のような文に対する構文木を描け

- a. x = y + 3 ;
- b. x = x + y + 3;
- c. if (x == 0) y = 1 ;

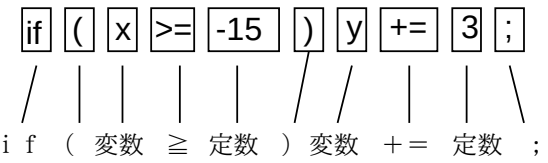
□ [2-5.] 変数と定数と足し算と掛け算とかっこから成る算術式を表す、あいまいでない文法を書け。べき乗 (演算子は「**」とする) を入れるとどうか?

3　字句解析 (2AM)

3.1　字句解析とは

□ 字句解析とは、

- ソースプログラムを「名前」「定数」などの「かたまり」(字句) に切り分け、
- 各かたまりの種別を区別する処理 (そうしておいた方が構文解析が楽)。



□ 字句の定義はタイプ 3 文法 (正規文法) で十分。

□ 字句解析器 (プログラムの作り方) → 手作り、オートマトン

3.2　字句定義の例

□ 字句の定義 → 正規文法/正規表現/オートマトン によることが多い (その程度の記述力で十分、記述がコンパクトで済む)

□ 例:　整数定数と実数定数

```
整定数 ::= 符号部 数字の並び
実定数 ::= 符号部 数字の並び 指数部
         | 符号部 数字の並び 少数点 数字の並び
         | 符号部 数字の並び 少数点 数字の並び 指数部
符号部 ::= + | - | ε
数字の並び ::= 数字 数字の並び | 数字
数字   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
指数部 ::= e 符号部 数字の並び
少数点 ::= .
```

これは正規文法になっていないが、正規文法に書き換えられる。

3.3　正規文法と正規表現

正規文法の言語を表すときには、「正規表現」と呼ばれる書き方を使うことが多い。この方がコンパクトに書けるし読みやすい。

□ 1.　「ε」は正規表現。

□ 2.　a が任意の終端記号のとき,「a」も正規表現 (a そのものを表す)。

□ 3.　α, β が正規表現であれば,「$\alpha\ \beta$」も正規表現 (連結)。

□ 4.　α, β が正規表現であれば,「$\alpha \mid \beta$」も正規表現 (または)。

□ 5.　α が正規表現であれば,「$\alpha*$」も正規表現 (0 回以上反復)。

□ 6.　グループ化のため適宜かっこを使う

あと、本当は不要なのだが、Unix の正規表現ツールでは次のものも使える。

□ 7.　α+ は　$\alpha\ \alpha*$ の意味 (1 回以上反復)。

□ 8.　α?　は（$\alpha \mid \varepsilon$）の意味 (あってもなくてもよい)。

□ 9.　[abc] は (a|b|c) の意味。[0-9] は (0|1|2|…|9) の意味。

3.4 正規表現の例

□ Unix のツール egrep は正規表現を与えて、その正規表現にマッチする (その正規表現が表す L(G) を含む) 行を選び出してくれる。

□ egrep の使い方: 「egrep ’正規表現’ ファイル」

□ たとえば次の正規表現だと:

- a
- abc
- (abc|def)
- te*t
- te+t
- te?t
- [aeiou][aeiou][aeiou]

3.5 正規表現による字句の記述

□ たとえば、「名前」は BNF だと

```
名前 ::= 英字 英数字の並び
英数字の並び ::= 英字 英数字の並び | 数字 英数字の並び | ε
英字 ::= a | b | ... | z
数字 ::= 0 | 1 | ... | 9
```

□ これを正規表現にすると

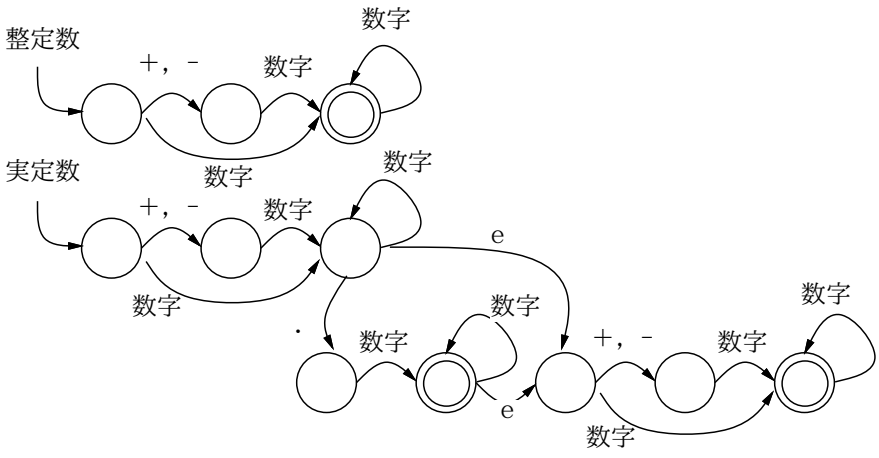
```
[a-z][a-z0-9]*
```

たったこれだけ! 短くなる。

3.6 字句定義の正規表現と有限オートマトン

□ 正規文法/正規表現の認識機 → 有限オートマトンが効率よい

□ オートマトンは手で作ることもできる

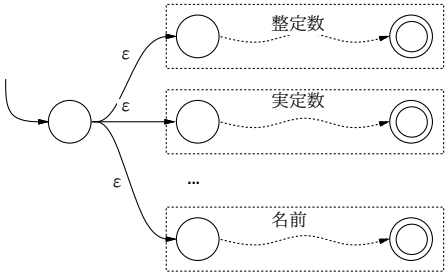


- 決定性オートマトン → 行き先が 1 通りに決まっている
- 非決定性オートマトン → 行き先が 1 通りに決まらない

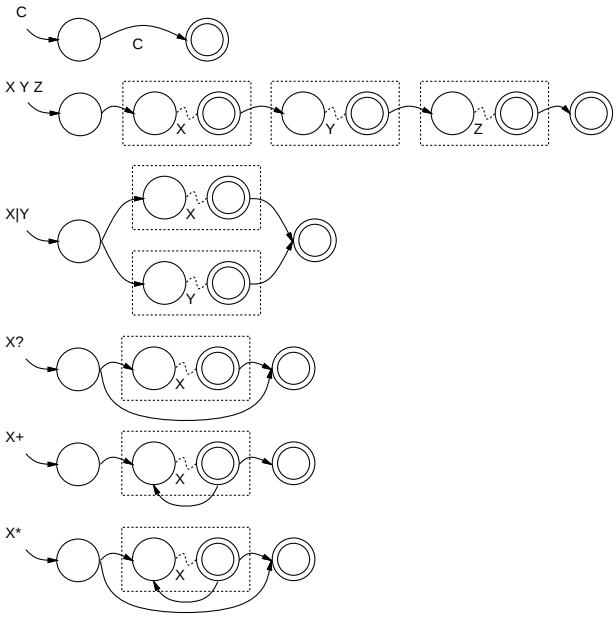
3.7　有限オートマトンの機械的な生成

□ 手で作るより、機械で (計算機で) 作った方が間違いがない

□ 単純に作ると非決定性、あとで決定性に直す



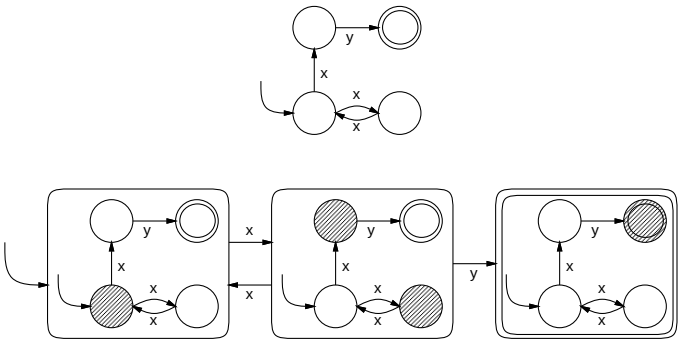
□ 正規表現を機械的に非決定性オートマトンに直す規則がある



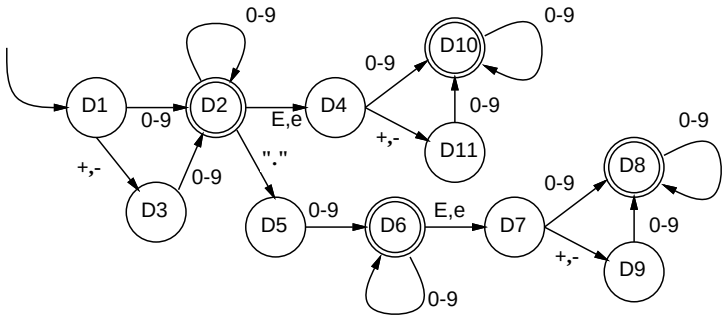
3.8　決定性有限オートマトンへの変換

□ 非決定性オートマトンを決定性オートマトンに直す方法がある

□ それには、「状態の集合」を 1 つの状態だと思えばよい



□ さらに「最小化」をする余地がある。次の例を見よう。



□ これは先に出て来た手で作ったものより状態数が多い。これを減らすには:

- ◎どうしで、重ね合わせても結果が変わらない (◎に到達するときはする、行き止まりになるときは行き止まり) なら、それらを重ね合わせる。
- そこにたどり着く○や◎について遡りながら同様にやる。
- これ以上重ね合わせられなくなったらおしまい。

3.9 字句解析器生成系

□ Lex ... Unix に備わっている、字句解析器生成系。これもオートマトンの原理を使用している。
□ たとえば整数、実数、文字列を認識する字句解析器の定義を示す。

```
L       [A-Za-z_]
D       [0-9]
DQuote  \"
Escape  \\
Dot     \.
StrChar [^"\\]
AnyChar .
Ident   {L}({L}|{D})*
IConst  [+-]?{D}+
RConst  [+-]?{D}+([Ee][+-]?{D}+|{Dot}{D}+([Ee][+-]?{D}+)?)
SConst  {DQuote}({StrChar}|{Escape}{AnyChar})*{DQuote}
%%
{Ident}         printf("<ident,%s>\n", yytext);
{IConst}        printf("<iconst,%s>\n", yytext);
{RConst}        printf("<rconst,%s>\n", yytext);
{SConst}        printf("<sconst,%s>\n", yytext);
[\n\t ]         ;
.               printf("invalid char: %c\n", yytext[0]);
```

□ これを動かしてみよう。

```
% lex sample.lex
% cc lex.yy.c -ll
% a.out
"this is a pen."
<sconst,"this is a pen.">
abc
<ident,abc>
123
<iconst,123>
123e10
<rconst,123e10>
123efg
<iconst,123>
```

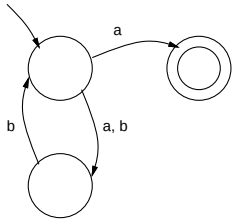
```
<ident,efg>
^D
%
```

3.10 字句解析の実際

- □ 字句解析器として使うには… 状態の記録、エラー対処など
- □ 予約語の扱い… オートマトンとして/名前の一部として
- □ 日本語などの問題
- □ 実際には字句解析は手で書くことが多い

3.11 演習問題 (2AM)

- □ [3-1.] 次のものを表す正規表現を書け。
  - a. 小数点つきの定数。e.g. 12.0, -23.45
  - b. 指数つきの定数。e.g. 1e0 -12.3e-5
  - c. 西暦の日付け。e.g. 1995.8.2 (ないし自分の好みの書式)
  - d. 元号の日付 (明治以後)。e.g. 平成７年８月２日
- □ [3-2.] 上のそれぞれに対応するオートマトンを書け。
- □ [3-3.] 次の図の非決定性有限オートマトンを決定性有限オートマトンに変換せよ。また、これらのオートマトンが認識する言語はどんなものか説明せよ。

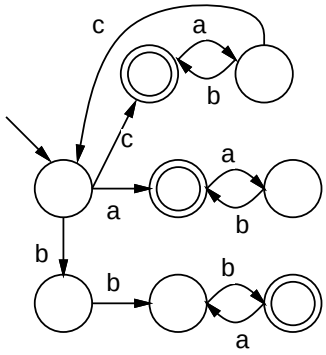


- □ [3-4.] 次の正規表現を機械的にオートマトンに直してみよ。

```
a ( b* | cc+ ) d
```

もし時間があればこれを決定性オートマトンに直せ。

- □ [3-5.] 次の図の決定性有限オートマトンを最小化してみよ。また、どのような言語を表すオートマトンか記せ。



- □ [3-6.] ここまでに出て来たどの (決定性) オートマトンでもよいから、前章の方法で C 言語のプログラムに直して動かしてみよ。

4 構文解析 (2PM)

4.1 構文解析と構文木

□ 構文解析: 入力プログラムを文法にあてはめること。

- 構文木を作ることにほぼ相当

□ たとえば次の文法に対し

```
プログラム ::= 文の並び
文の並び ::= 文 文の並び | ε
文 ::= 名前 = 式 ; | read 名前 ; | print 式 ;
     | if ( 条件 ) 文 | { 文の並び }
条件 ::= 式 < 式 | 式 > 式
式 ::= 名前 | 整定数
```

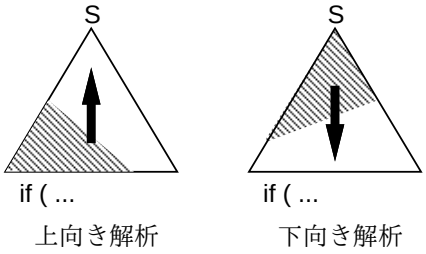
□ 次のプログラムは、どんな構文木になるだろう?

```
read x; read y; if(x>y) { z=x; x=y; y=z; } print x; print y;
```

4.2 上向き解析と下向き解析

□ 構文木を作るとき、

- 下から上へ向かって作る流儀 (上向き解析) と、
- 上から下へ向かって作る流儀 (下向き解析) とがある。



上向き解析　　下向き解析

4.3 下向き解析

4.3.1 First/Follow

□ 下向き解析: 出発記号から初めて、あてはまる規則で順に置き換えていく

□ 選択肢のところで正しく決めるには: First(A) と Follow(A) を使う

□ Follow(A) --- 記号 A の直後に来ることのできる端記号の集合 ($ は「おしまい」を表す特別な記号)

- A が「おしまい」に来るなら $ ∈ Follow(A)

□ First(A) --- 記号 A の先頭に来ることのできる端記号の集合

- A ⇒ε になり得るなら Follow(A) の要素も全部加える

□ 先の文法に対する First/Follow を考えてみる

```
Follow(プログラム) = $
Follow(文の並び) = } $
Follow(文) = } $ { IF PRINT READ 名前
Follow(条件) = )
Follow(EXPR) = < > ) ;

First(プログラム) = { IF PRINT READ 名前 $
First(文の並び) = 名前 READ PRINT IF { } $
First(文) = { IF PRINT READ 名前
First(条件) = 名前 整定数
First(式) = 名前 整定数
```

4.3.2 LL(1) 解析器と LL(1) 文法

☐ LL(1) 解析器: 下向きに 1 記号の先読みで解析

☐ 1 記号の先読みでわからない文法は LL(1) でない

- その場合には「くくり出し」の変形で対処 (例: 「<」と「>」)

☐ 左再帰があると LL(1) でない

- その場合には空置換を使った変形
- A → A β
- A → γ

☐ これは要するに「γ β β β β…」ですよね

- A → γ A'
- A' → β A'
- A' → ε

☐ たとえばさっきの文法を LL(1) に直すと:

```
プログラム ::= 文の並び
文の並び ::= 文 文の並び | ε
文 ::= 名前 = 式 ; | read 名前 ; | print 式 ;
     | if ( 条件 ) 文 | { 文の並び }
条件 ::= 式 比較演算子 式
比較演算子 ::= < | >
式 ::= 名前 | 整定数
```

4.3.3 再帰下降解析器

☐ 記号の展開〜手続き呼び出し

☐ 非端記号を手続きに対応させる

☐ ます字句解析から

```
%%
read            return READ;
print           return PRINT;
if              return IF;
[;(){}<>]       return yytext[0];
[0-9]+          return ICONST;
[a-z][a-z0-9]*  return IDENT;
[. \t\n]                ;
```

□ 続いてこれを取り込む C のソース

```
#define READ   256
#define PRINT  257
#define IF     258
#define IDENT  259
#define ICONST 260
#define yywrap() 1
#include "lex.yy.c"
#include <stdio.h>

int next;
error() { fprintf(stderr, "syntax error.\n"); exit(1); }
match(int x) { if(x != next) error(); next = yylex(); }

main() { next = yylex(); statlist(); }

statlist() { switch(next) {
  case '{': case IF: case PRINT: case READ: case IDENT:
    stat(); statlist(); break;
  case '}': case 0:
    break;
  default: error(); }
}
stat() { switch(next) {
  case IDENT:
    match(IDENT); match('='); expr(); match(';'); break;
  case PRINT:
    match(PRINT); expr(); match(';'); break;
  case READ:
    match(READ); match(IDENT); match(';'); break;
  case IF:
    match(IF); match('('); cond(); match(')'); stat(); break;
  case '{':
    match('{'); statlist(); match('}'); break;
  default: error(); }
}
cond() { switch(next) {
  case IDENT: case ICONST:
    expr(); relop(); expr(); break;
  default: error(); }
}
expr() { switch(next) {
  case IDENT:
    match(IDENT); break;
  case ICONST:
    match(ICONST); break;
  default: error(); }
}
relop() { switch(next) {
  case '<':
    match('<'); break;
  case '>':
    match('>'); break;
  default: error(); }
}
```

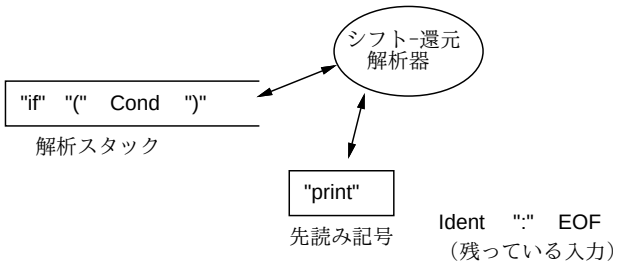
4.4 上向き解析

4.4.1 シフト-還元解析器

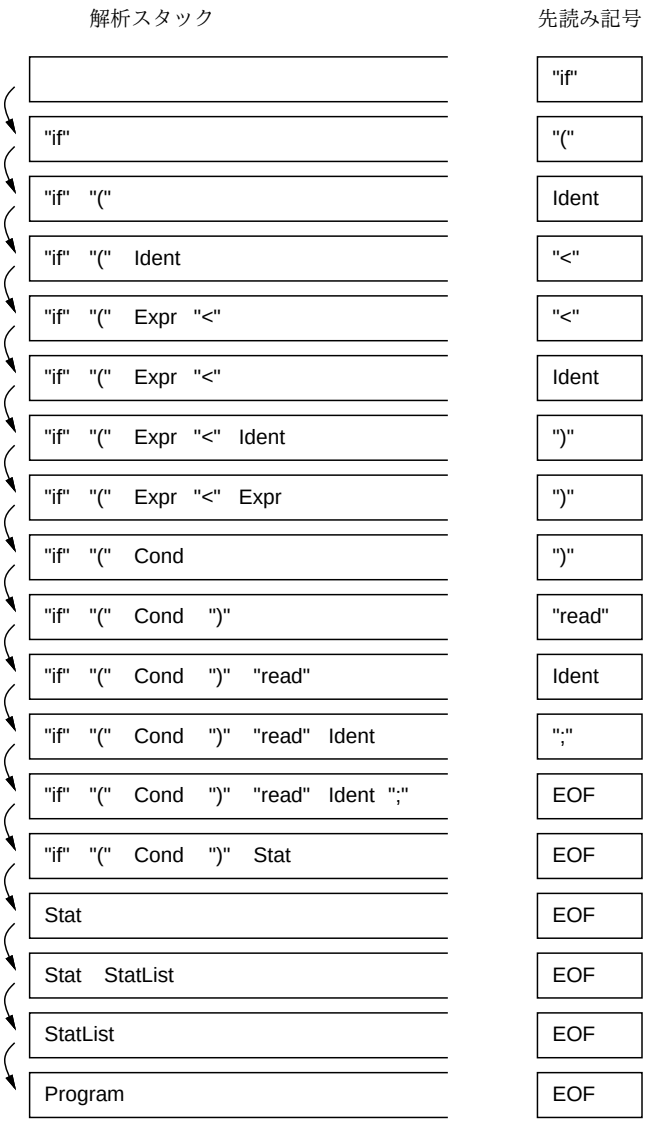
□ 解析スタックを使って、生成規則の逆向きに置き換えて行く

□ シフト --- 先読み記号をスタックに積み，入力を進める．

□ 還元 --- 生成規則「A → α」に対応して，記号列αの長さ分スタックを取り降ろし，代りに A を積む．



□ 先の文法を S-R で解析してみると…



4.4.2 LR オートマトンと項

□ 項: 文法規則+現在位置。全体を [] で囲み、現在位置を「・」で表す。

□ 「Stat → "if" "(" Cond ")" Stat」から作られる項:

- [Stat → ・ "if" "(" Cond ")" Stat]
- [Stat → "if" ・ "(" Cond ")" Stat]
- [Stat → "if" "(" ・ Cond ")" Stat]
- [Stat → "if" "(" Cond ・ ")" Stat]
- [Stat → "if" "(" Cond ")" ・ Stat]
- [Stat → "if" "(" Cond ")" Stat ・]

4.4.3 LR(0) オートマトンの作成

□ 閉包: 複数の場所に「並行している」ことを計算

□ [Stat →"if" "(" ・ Cond ")" Stat] にいるなら、

```
Cond → Expr "<" Expr
Cond → Expr ">" Expr
```

から作られる次の項にもいることになる

```
[ Cond → ・ Expr "<" Expr ]
[ Cond → ・ Expr ">" Expr ]
```

□ ということはさらに、

```
Expr → Ident
Expr → Const
```

から作られる次の項にもいることになる

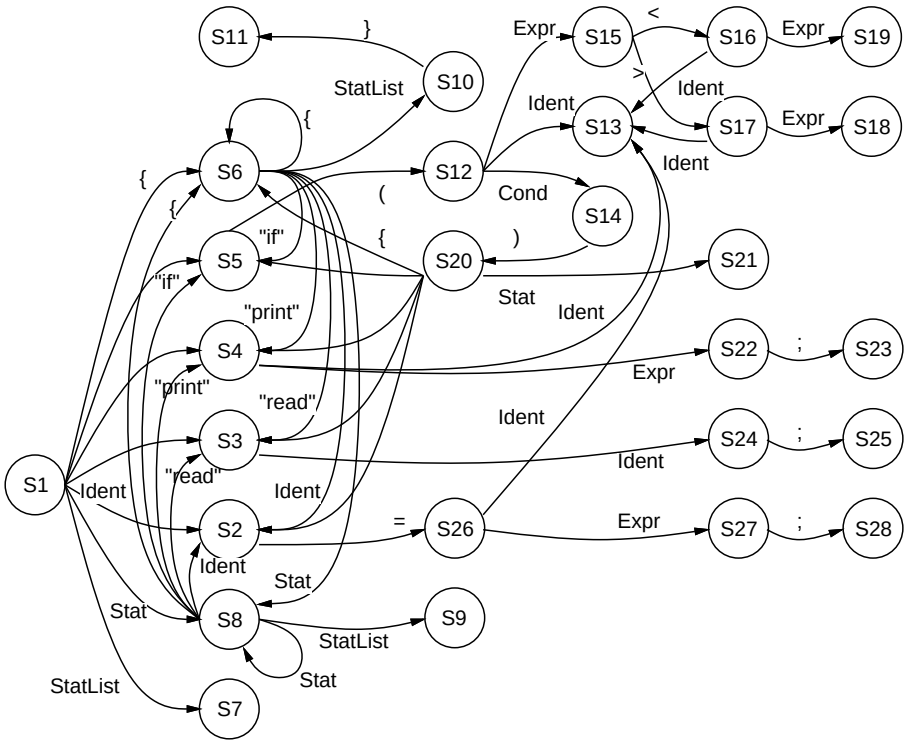
```
[ Expr → ・ Ident ]
[ Expr → ・ Const ]
```

□ 閉包がオートマトンの状態になる

□ 次の手順でオートマトンを作る

- まず、「開始記号の前にいる」項の閉包を作る。これが初期状態
- ここから、各記号のついた矢線を出し、「それを読んだ後の」項の閉包を作ってれを
- これをもはや新しい矢線や状態ができなくなるまでやる
- 出発記号の後にいる状態の閉包が最終状態

□ 先の文法からできたオートマトンを示す



□ 使い方: 入力をたどっていって、行き止まったら還元し、状態は 1 つ前に戻る

□ これを使うと「やまかん」でなくても先の S-R パーザの解析動作が行える

4.4.4 SLR(1) 解析器

□ 実は、「行き止まりで還元」だけでは済まない→還元するかしないか、するとしたらどの規則でするかで迷う場合がある (シフト-還元衝突、還元-還元衝突という)

□ ある箇所で、「A → ○」(もしかしたら複数) による還元とシフトとがともに可能なとき、次の規則でどうするか決めるやり方がある:

- 次の入力記号が Follow(A) に入っている → その規則で還元
- それ以外→シフト

□ これを「SLR(1) 解析器」と呼んでいる。上の規則でも衝突が解消されないときは「文法が SLR(1) でない」という。

□ 解析時には、スタックに状態と記号を交互に積む。

- [1.] まず S0 を積む
- [2.] 先頭の状態から次の入力記号への遷移があれば shift
- [3.] 先頭の状態に左端が・の項があり、その項の左辺の記号の Follow に次の入力記号があれば、reduce(右辺の長さの倍だけスタックから取り除き、代わりに左辺を積む)
- [4.] 先頭の状態と次の記号に応じて次の状態を積み、2 へ戻る
- [5.] 開始記号への還元で終る

□ 普通は、構文解析表の形で表現する

	ident	=	;	(	)	{	}	<	>	eof	if	read	print	stlist	stat	cond	expr
S1	s2					s6	r3			r3	s5	s3	s4	7	8		
S2		s26															
S3	s24																
S4	s13																22
S5				s12													
S6	s2					s6	r3			r3	s5	s3	s4	9	8		
S7										accept							
S8	s2					s6	r3			r3	s5	s3	s4	9	8		
S9							r2			r2							
S10							s11										
S11	r8					r8	r8			r8	r8	r8	r8				
S12	s13															14	15
S13			r11		r11			r11	r11								
S14					s20												
S15								s16	s17								
S16	s13																18
S17	s13																18
S18					r10												
S19					r9												
S20	s2					s6					s5	s3	s4		21		
S21	r7					r7	r7			r7	r7	r7	r7				
S22			s23														
S23	r6					r6	r6			r6	r6	r6	r6				
S24			s25														
S25	r5					r5	r5			r5	r5	r5	r5				
S26	s13																27
S27			s28														
S28	r4					r4	r4			r4	r4	r4	r4				

動作表　　行先表

4.4.5　正準 LR(1) 解析器

☐ SLR(1) でない文法:　Follow による区別では不十分な場合

☐ より厳密に:　項に先読み記号を含める→正準 LR(1) 解析器

- [Program → ・ Stat ; EOF]
- [Right → Left ・ ; "="]

☐ 文法クラスは右から左へ後戻りなく解析できる最大範囲

☐ 状態数が非常に多くなる

4.4.6　LALR(1) 解析器

☐ LR(1) の状態のうち、先読み記号が同じものを併合→状態数が減少

☐ 文法クラスはやや小さくなるが実用上十分

☐ Yacc を始め多くのツールで使われている

4.5　再帰上昇解析器

☐ オートマトンの状態移行を観察すると:

- [1.]　最初にある状態に来たときは，入力記号を参照してシフトか還元かを決める．
- [2.]　シフトの場合は，入力記号に応じた後続状態に移るが，やがて還元が起きて戻ってくる．そのときこの状態に対して起きることは次のいずれか
- [2a.]　規則の右辺の数だけ状態を取り降ろす際にスタックから取り降ろされてなくなる
- [2b.]　還元の結果この状態に戻ってきて，規則の左辺に基づき次の状態へ移る．
- [3.]　還元の場合は，対応する還元動作を行う．すなわち，生成規則 1 の還元なら解析終了だが，そうでなければ右辺の数 n だけ状態を取り降ろす．これはすなわち，この状態へやってきた n 個前の状態への戻りを意味する (2a と対応)．

□ これと同じ動作を手続き群で行なうようにする。

□ 各状態が手続きに対応

□ 再帰下降パーザと同様の特徴

4.6 実用のための構文解析

4.6.1 曖昧な文法

□ 次の文法を考えて見る

- Stat → "if" Cond "then" Stat "else" Stat
- Stat → "if" Cond "then" Stat

□ 次のようなあいまいさ

```
if C1 then ( if C2 then A else B )  -- (1)
if C1 then ( if C2 then A ) else B  -- (2)
```

□ 次のように書き換えればよいが…

- BalStat → "if" Cond "then" BalStat "else" Stat | 各種の文
- Stat → "if" Cond "then" Stat | BalStat

□ あいまいな文法の方が使いやすい場合もある

□ 動作はあいまいでないようにうまく制御する

□ 演算子の順位や結合性などもこれでうまく扱える

4.6.2 誤りからの回復

□ 構文エラーですぐ止まるのは不親切 (とされている…)

□ とりあえずそこをパスして先の解析を続行: エラー回復

□ そこが本来どうなっているべきか推察して直す: エラー修復

4.6.3 解析表の圧縮

□ 構文解析表は「白いところ」が多い

□ →うまく圧縮してコンパクトにしたい (今もそうか?)

4.7 構文解析器生成系

□ 解析用の表を生成してくれるツール。Yacc が代表的

```
%start program
%token IF ELSE READ PRINT IDENT ICONST
%nonassoc '>' '<'
%left '+' '-'
%left '*','/'
%right '^'
%%
program  : statlist
```

```
        ;
statlist :
        | statlist stat
        ;
stat    : IDENT '=' expr ';'
        | READ IDENT ';'
        | PRINT expr ';'
        | IF '(' expr ')' stat
        | IF '(' expr ')' stat ELSE stat
        | '{' statlist '}'
        ;
expr    : expr '<' expr
        | expr '>' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | expr '^' expr
        | IDENT
        | ICONST
        ;

% yacc -v yacsam.yacc
conflicts: 1 shift/reduce

% cat y.output
...
37: shift/reduce conflict (shift 38, red'n 7) on ELSE
state 37
        stat :  IF ( expr ) stat_    (7)
        stat :  IF ( expr ) stat_ELSE stat
        ELSE  shift 38
        .  reduce 7
...

alpha   [A-Za-z]
digit   [0-9]
%%
if                              return IF;
else                            return ELSE;
read                            return READ;
print                           return PRINT;
{alpha}({alpha}|{digit})*       return IDENT;
{digit}+                        return ICONST;
[-+=;(){}<>*/^]                 return yytext[0];
[\n\t ]                         ;
.                               ;
%%

#include <stdio.h>
main() { yyparse(); }
yyerror(s) char *s; { fprintf(stderr, "%s\n", s); }
#define yywrap() 1
#include "y.tab.c"
#include "lex.yy.c"

% lex yacsam.lex
% cc yacsam.c
% a.out
if(a > 1) { read a; print a; }
^D%
```

```
% a.out
if(a > ) { read a; print a; }
syntax error
%

% a.out
if(a + 1 > b - 1) print 1;
if(a > b + 1 > c) print 1;
syntax error
%

         | error ';'

% a.out
if(a > ) print a;
syntax error
b = b +; c = c -;
syntax error
syntax error

b = b + - c -;
syntax error
```

4.8 演習問題 (2PM)

□ [4-1.] 次の文法は LL(1) ではない。同等な LL(1) 文法に変形してみよ。

- 式 ::= 式 + 名前 | 式 - 名前 | 名前

□ [4-2.] 次の文法について、下記の入力に対する構文木を描け。

```
文の並び = ε | 文の並び 文
文 ::= if 文 | 代入文 | \{ 文の並び \}
if 文 ::= if ( 条件 ) 文 | if ( 条件 ) 文 else 文
代入文 ::= 名前 = 式 ;
式 ::= 項 | 式 + 項
項 ::= 因子 | 項 * 因子
因子 ::= 名前 | 整定数
条件 ::= 式 == 式 | 式 != 式
```

- a. 2 * x + 3 (式)
- b. x = y + 1; (代入文)
- c. if(x != 0) x = 1; (if 文)
- d.

```
if(x != 0) x = 1;
    if(y != 0) x = 0;
    else       x = -1;   (文)
```

- e. x = 1; y = 2; z = 3; (文の並び)

□ [4-3.] 上のどれかを S-R パーザで解析する様子を図示せよ。

□ [4-4.] 次の文法を考え、この文法の各記号に対する First、Follow を示せ。

- S → C
- C → a C b

- C → ε

□ [4-5.] この文法に対する LR(0) 項をすべて列挙せよ。

□ [4-6.] 上の文法に対する LR オートマトンを構成せよ。

□ [4-7.] この文法に対して、入力列 aabb を SLR(1) 解析器で解析するときの様子を示せ。

5 意味解析 (3AM)

5.1 意味解析の役割と位置づけ

□ 意味解析の役割:

- 名前
- 型情報
- 制御情報
- 生成

□ 受渡し形式: 記号表、修飾された構文木

5.2 属性文法

□ 記号に属性値を持たせる: `Expr.type`

□ 構文規則と対で計算規則を記述

□ 意味解析の定式化手段として有効

□ 単一代入、副作用なし→実用的には?

```
プログラム ::= 宣言の並び 文の並び
  { プログラム.mesg = 宣言の並び.mesg ∪ 文の並び.mesg
    プログラム.code = 文の並び.code
    文の並び.symtab = 宣言の並び.symtab }
```

5.3 構文指示翻訳

□ 意味規則を普通のプログラミング言語で記述

□ 記述は自由になったが、実行順序を考慮する必要

□ 実用的には多く使われている

```
プログラム ::= { start(); } 宣言の並び 文の並び { finish(); }
```

5.4 構文指示翻訳の実装

5.4.1 下向き解析器における構文指示翻訳

□ 再帰下降解析の場合、解析手続きに動作を組み込む

□ 属性値は引数として受け渡す

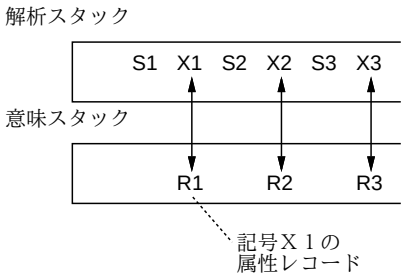
```
プログラム ::= 二進数
  { printf("\n%d\n", 二進数.val); }
二進数 ::= ε ;
  { 二進数.val = 0; 二進数.len = 0;
二進数 ::= 数字 二進数1
  { 二進数.val = 二進数1.val + 数字.val * 2**二進数1.len;
    二進数.len = 二進数1.len + 1; }
数字 ::= 0
   { 数字.val = 0; }
数字 ::= 1
   { 数字.val = 1; }
```

これを C 言語にすると次のように…

```
error() {
  printf("error!\n"); exit(1);
}
power(int x, int n) {
  int v = 1; while(--n >= 0) v *= x; return v;
}
typedef struct { int val, len; } attr;
int c;
attr digit() { attr d;
  switch(c) {
case '0':
    d.val = 0; c = getchar(); return d;
case '1':
    d.val = 1; c = getchar(); return d;
default: error();
  }
}
attr binnum() { attr b, b1, d;
  switch(c) {
case '\n':
    b.val = 0; b.len = 0; return b;
case '0': case '1':
    d = digit(); b1 = binnum();
    b.val = b1.val + d.val * power(2, b1.len);
    b.len = b1.len + 1; return b;
default: error();
  }
}
main() { attr b; c = getchar();
  b = binnum(); printf("%d\n", b.val);
}
```

5.4.2 上向き解析器における構文指示翻訳

□ シフト還元解析器の場合→各記号インスタンスに対応させた属性レコード→これを入れるスタック→意味スタック



□ 規則の途中での動作→２通りの実現方法

```
Stat → "if" Cond "then" { condjump(); } Stat { endif(); }

Stat → IfHead Stat { endif(); }
IfHead → "if" Cond "then" { condjump(); }

Stat → "if" Cond "then" Marker-1 Stat { endif(); }
Maker-1 → ε { condjump(); }
```

□ 先の 2 進数の例を Yacc にすると…

□ lex

```
%%
[01]  return yytext[0];
"\n"  return '\n';
```

□ yacc

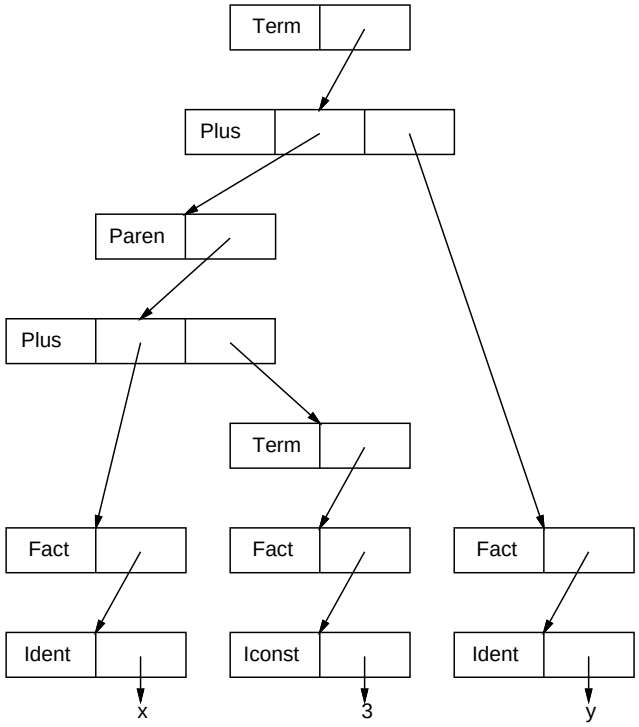
```
%type <attr> binnum, digit;
%union {
struct { int len, val; } attr;
}
%%
program: binnum '\n' { printf("\n%d\n", $1.val); }
       ;
binnum :             { $$.val = 0; $$.len = 0; }
       | digit binnum{ $$.len = $2.len + 1;
                       $$.val = $2.val + $1.val * power(2, $2.len); }
       ;
digit  : '0'         { $$.val = 0; }
       | '1'         { $$.val = 1; }
       ;
%%
```

□ C

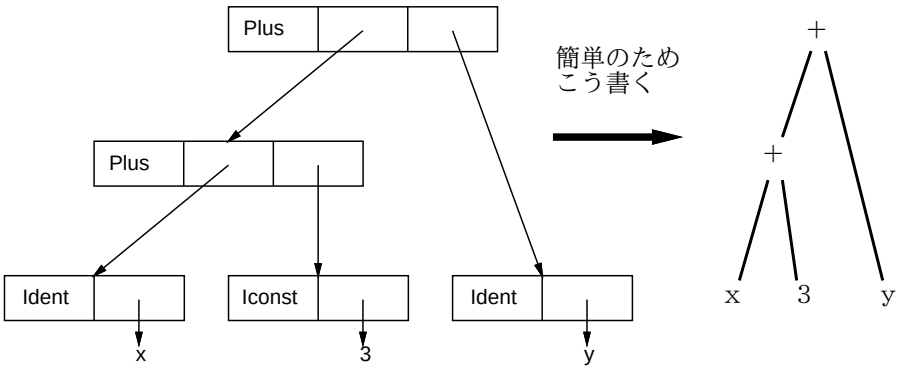
```
#include <stdio.h>
main() { yyparse(); }
yyerror(s) char *s; { fprintf(stderr, "%s\n", s); }
power(int x, int n) {
  int v = 1;
  while(--n >= 0) v *= x;
  return v;
}
#define yywrap() 1
#include "y.tab.c"
#include "lex.yy.c"
```

5.5 構文木の生成

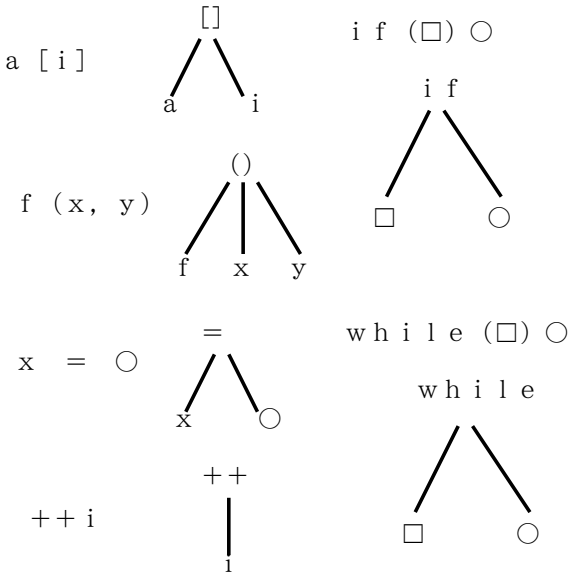
```
Expr = Term | Term "+" Expr | Term "-" Expr
Term = Fact | Fact "*" Term | Fact "/" Term
Fact = Ident | Iconst | "(" Expr ")"
```



□ 抽象構文木→余計な枝を省略



□ さまざまな機能も何らかの木に



```
%type <expr_attr> expr;
%type <term_attr> term;
%type <fact_attr> fact;
%token ICONST;
%token IDENT;
%{
typedef struct node {
    int kind;
    struct node *left, *right;
    char *str; } *nodep;
char *copystr();
nodep mknode();
#define T_Plus 1
#define T_Minus 2
#define T_Mult 3
#define T_Divd 4
#define T_Ident 5
#define T_Const 6
%}
%union {
struct { nodep tree; } expr_attr;
struct { nodep tree; } term_attr;
struct { nodep tree; } fact_attr;
}

%%
program :
        | program expr ';' { ptree($2.tree); printf("\n"); }
        ;
expr    : term          { $$.tree = $1.tree; }
        | term '+' expr { $$.tree = mknode(T_Plus, $1.tree, $3.tree, 0); }
        | term '-' expr { $$.tree = mknode(T_Minus, $1.tree, $3.tree, 0); }
        ;
term    : fact          { $$.tree = $1.tree; }
        | fact '*' term { $$.tree = mknode(T_Mult, $1.tree, $3.tree, 0); }
        | fact '/' term { $$.tree = mknode(T_Divd, $1.tree, $3.tree, 0); }
        ;
fact    : IDENT         { $$.tree = mknode(T_Ident, 0, 0, copystr()); }
        | ICONST        { $$.tree = mknode(T_Iconst, 0, 0, copystr()); }
        | '(' expr ')'  { $$.tree = $2.tree; }
        ;

#include <stdio.h>
main() { yyparse(); }
#define yywrap() 1
yyerror(s) char *s; { fprintf(stderr, "%s\n", s); exit(1); }
#include "y.tab.c"
#include "lex.yy.c"
char *copystr() {
  char *p = (char*)malloc(strlen(yytext)+1); strcpy(p, yytext); return p; }
nodep mknode(k, l, r, s)
  int k; nodep l, r; char *s; {
  nodep p = (nodep)malloc(sizeof(struct node));
  p->kind = k; p->left = l; p->right = r; p->str = s; return p; }
ptree(x)
  nodep x; {
  switch(x->kind) {
case T_Plus:  printf(" (+");ptree(x->left);ptree(x->right);printf(")");break;
case T_Minus: printf(" (-");ptree(x->left);ptree(x->right);printf(")");break;
case T_Mult:  printf(" (*");ptree(x->left);ptree(x->right);printf(")");break;
```

```
case T_Divd:  printf(" (/");ptree(x->left);ptree(x->right);printf(")");break;
case T_Ident: case T_Iconst: printf(" %s", x->str); } }
```

```
% a.out
1 + 2 ;
 (+ 1 2)
1 + 2 + 3 + 4 ;
 (+ 1 (+ 2 (+ 3 4)))
2 * 3 + 4 * 5 ;
 (+ (* 2 3) (* 4 5))
(((((((a)))))));
 a
```

6 記号表 (3AM cont.)

6.1 記号表の位置づけ

□ 記号表の情報→名前の情報

- (a) 名前の文字列
- (b) その名前のスコープ
- (c) その名前の用途
- (d) その名前に付随する属性

6.2 表の概念と実現技法

6.2.1 表の概念

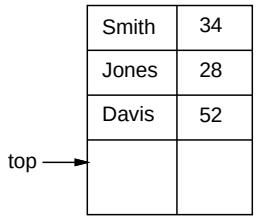
□ 表には<キー，情報>の組を複数個格納

□ キーの値→同じキーをもつ組を取り出す (検索)

6.2.2 1 次元配列による表

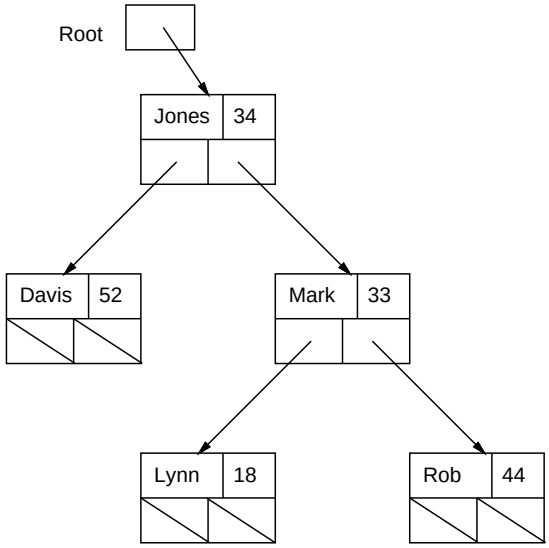
□ 単に配列に順番に入れて行く。

□ 単純だが検索は遅い



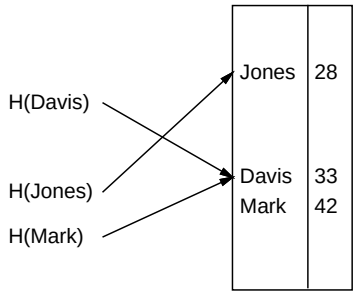
6.2.3 2 分木による表

□ 2 分探索が可能な動的データ構造

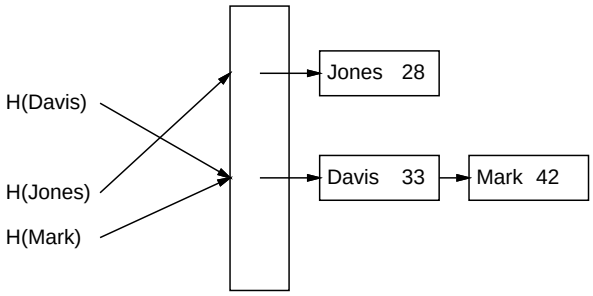


6.2.4 ハッシュ表

□ 項目数に関係なく一定時間で検索



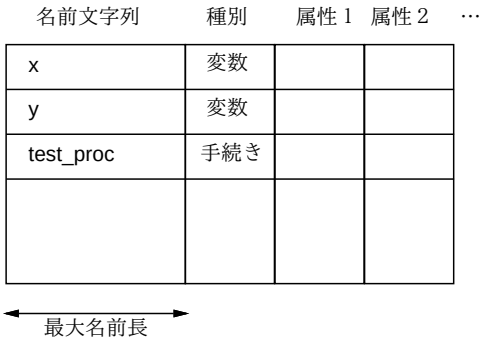
□ 衝突の処理→ランダムハッシュ、チェインリハッシュ



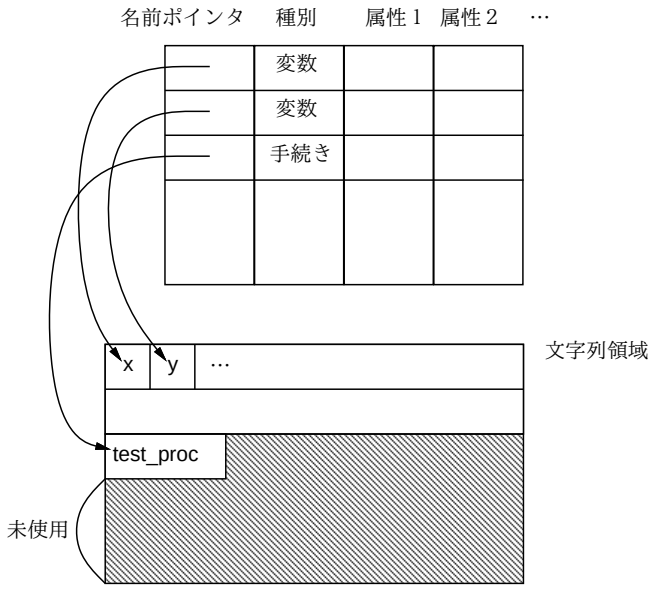
6.3 記号表の特質

6.3.1 文字列領域

□ 名前の文字列→長いものも短いものもある→管理?



□ 可変長文字列の効率的な扱い→文字列領域を別に取る

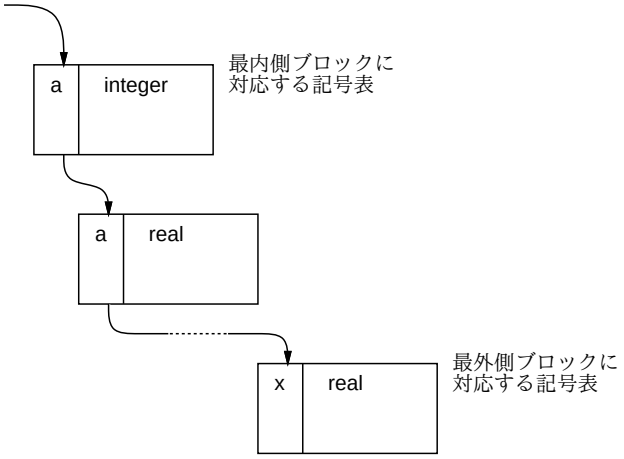


6.3.2 ブロック型スコープの処理

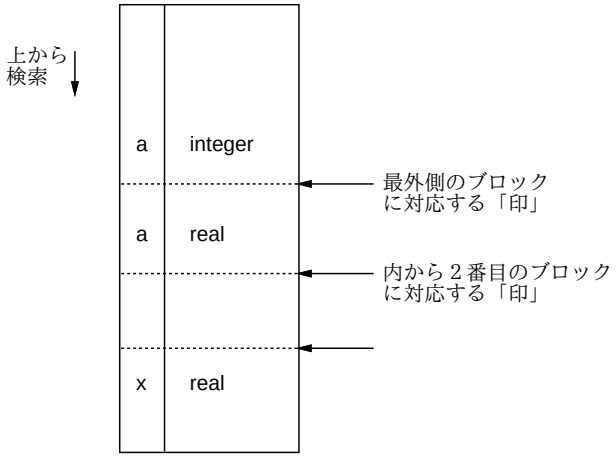
□ 言語のスコープ規則→それなりの扱い必要

```
begin
    real a;
        ...
        a := 1.0;
        begin
            integer a;
            ...
            a := 0;
            ...
        end;
        x := a;
       ...
end
```

□ スコープごとの表の連鎖

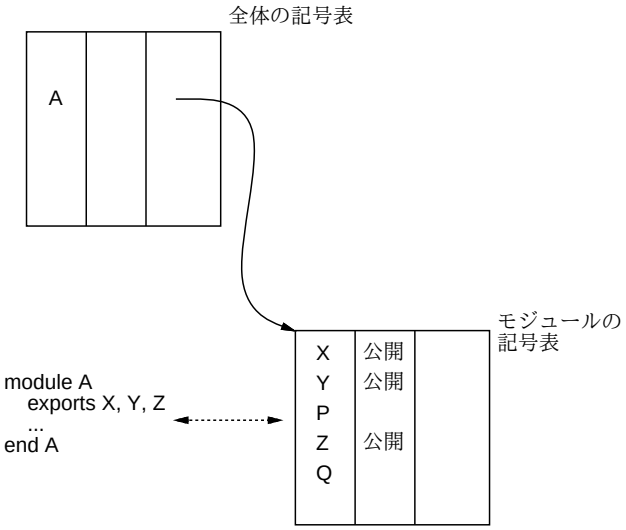


□ 1 つの表で済ませる工夫も…



6.3.3 スコープ規則の拡張

□ モジュール→閉じたあとでも参照できるスコープ



6.4 記号表に関連する言語処理系の問題

6.4.1 前方参照の問題

□ 前方参照: 定義より前にその記号を参照すること

```
        goto L;
        ...
L:      ...

type list = ^cell;
     cell = record
                car, cdr: list
            end;
```

□ 場合によってはかなり面倒

```
type a = real;
...
procedure p;
type b = a;
     a = integer;
...
```

6.4.2 分割翻訳

□ 独立翻訳→プログラマが正しいことをしてると仮定

□ 分割翻訳→翻訳単位間での整合性を検査

□ インタフェース用ファイル、モジュールデータベースなど使用

7 型検査 (3AM Cont.)

7.1 型の役割りと位置づけ

□ 型の役割り…

- (a) 複数のデータの種類を使い分ける.
- (b) 様々なデータ構造の構成を可能にする.
- (c) プログラムの安全性を高める.
- (d) プログラムの効率を高める.
- (e) プログラムの書きやすさ/読みやすさを高める.

7.2 型の同一性

□ 名前による同一性

```
type point = record x, y: real end;
var  a: point;
     b: point;
     c: record x, y: real end;

var  a, b: record x, y: real end;
     c: record x, y: real end;
```

□ 構造による同一性→単一化の問題

```
type p1 = ^r1;
     r1 = record r:real; next: p1 end;
     p2 = ^r2;
     r2 = record r:real; next: p2 end;

type p3 = ^r3;
     p4 = ^r4;
     r3 = record r:real; next: p4 end;
     r4 = record r:real; next: p3 end;
     p5 = ^r5;
     p6 = ^r6;
     r5 = record r:real; next: p6 end;
     r6 = record r:real; next: p5 end;
```

7.3 宣言の処理

7.3.1 型指定の処理

□ 型表を作る→1つの型は1つのエントリ

7.3.2 宣言部の処理

□ 名前を同定しながら型を決定する

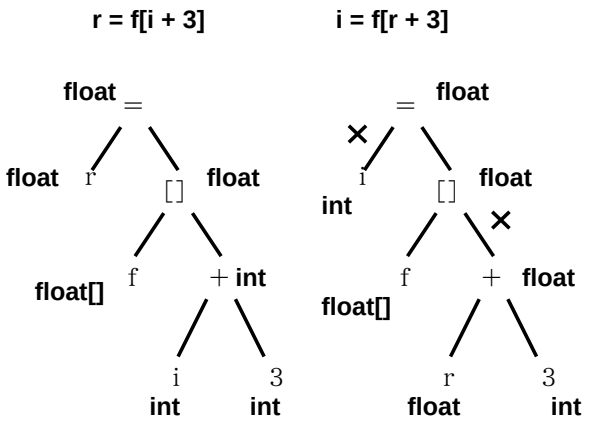
□ 多重定義を許す場合、1つの名前に複数の定義が対応

7.4 式における型の同定

□ 通常は宣言から一意に決まる

- 変数、定数など葉っぱのところに型を記入
- 下から順に途中の節を記入
- 正しく割り当てられない箇所→型エラー

float f[10]; float r; int i; とする



□ 多重定義がある場合→上から伝播/下から伝播

7.5 演習問題 (3AM)

□ [5-1.] 次の構文を考える

```
符号つき数値 = "+" 整数 | "-" 整数 | 整数
整数 = 数字 | 数字 整数
数字 = "0" | "1" | "2" | "3" | "4" |
       "5" | "6" | "7" | "8" | "9"
```

- a. 符号つき整数の値を求める属性文法をつけよ
- b. さらに少数点つき数を扱えるように拡張せよ
- c. どちらでも好きな方を再帰下降解析と組み合わせて動かせ

□ [5-2.] 次のプログラムに現われるすべての名前に名前ごとの固有番号をつけよ。また、それぞれの名前の種別 (広域、局所、引数)、および型 (整数、整数を返す関数、…) を表にまとめよ。

```
int c;
main() {
  int n, r; real c;
  scanf("%d, %d", &n, &r);
  c = comb(n, r);
  printf("%8.0lf\n", c); }
real comb(int n, int r) {
  c = fact(r) * fact(n - r);
  return fact(n) / c; }
fact(int n) {
  if(n <= 1) return 1; else return n * fact(n - 1); }
```

固有番号	名前	種別	型
1	c		

□ [5-3.] 次の各式の抽象構文木を描き、各部分木の型を記入せよ。ただし変数の型は i:int、r:real、s:char* とする。

- a. r = 3.1416 * i * i
- b. r = i * i * 3.1416
- c. i = trunc(r) * strlen(s)
- d. r = i = 0

8　実行時環境 (3PM)

8.1　実行時環境と名前の束縛

□ 実行時規約とは…

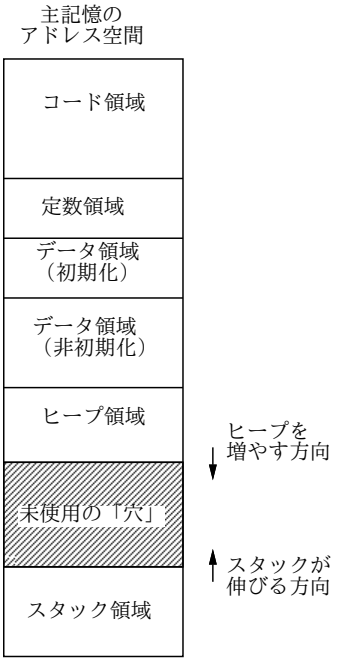
- 記憶領域の配置や割り当て方
- 変数の種類ごとのアクセス方法
- 関数の呼び出し方/呼び出され方や引数の渡し方/渡され方
- 生成コードと実行時ライブラリの分担

□ 静的束縛と動的束縛

8.2　記憶領域の種別と割当て

□ 記憶領域の種別…

- コード領域 --- プログラムの命令を保持する.
- 定数領域 --- 定数 (初期設定され，書き換えられないデータ) を保持する.
- 初期設定データ領域 --- 初期設定され，書き換えられるデータの領域.
- 非初期設定データ領域 --- 初期設定されないデータの領域.
- ヒープ領域 --- 実行時に動的に割り当てられるデータの領域.
- スタック領域 --- 局所変数など手続き呼出しに付随して割り当てられるデータ，戻り番地，その他管理情報の領域.



8.3　スタックとスタックフレーム

8.3.1　スタックの用途

□ (a) 手続きに局所的な作業領域

□ (b) 戻り番地の情報

□ (c) 引数の情報/戻り値の情報

□ (d) 呼出し元の領域を示す情報

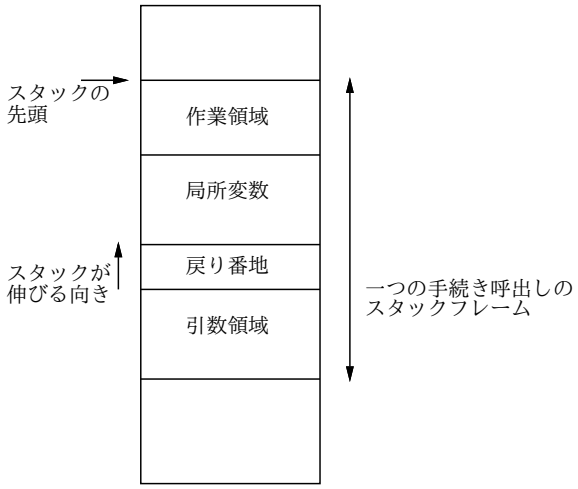
□ (e) 外側のスコープを示す情報

□ (f) 呼びに伴って壊れると困るレジスタ内容の写し

8.3.2 スタックフレームとフレームポインタ

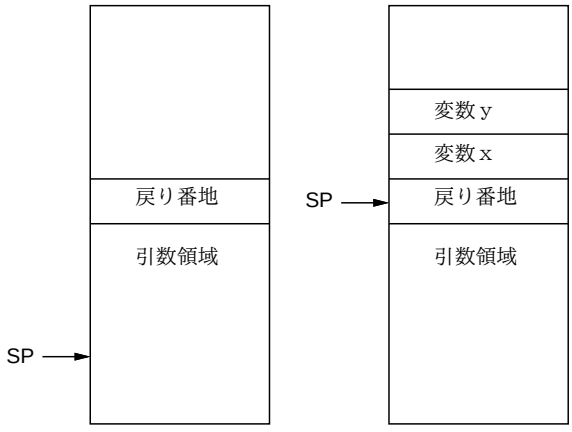
□ 実際の言語では「特定の○○番地」という番地指定はあまり使わない

□ 代わりに、スタックフレーム上の領域を多く使う



□ スタックフレーム上の起点を sp (stack pointer) というレジスタで指しておいて、「そのどれだけ上/下」という番地指定をすればよい

```
y = x  → load r1,4(sp)   ←変数 x のありかを指定している
         store r1,8(sp)  ←変数 y のありかを指定している
```

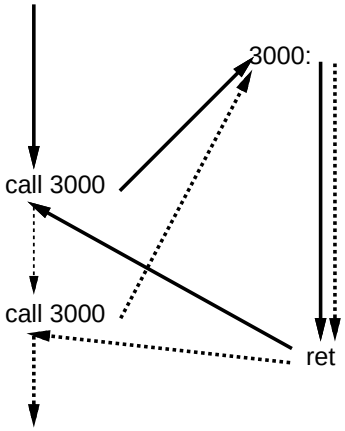


(a) 呼ばれる前　　(b) 呼ばれた後

□ なお、sp と fp (frame pointer) という 2 つのレジスタを用いる流儀もあるが、ここでは簡単のため sp のみで説明する (一部の図では fp も使っている)。

8.4 サブルーチンリンケージ

□ サブルーチンとは要するに、ある場所へ行ってそこからの命令を実行し、終わったら「元の場所 (の次)」へ戻ってくること。



□ どうやって行き、戻って来るかの約束==サブルーチンリンケージ (コンパイラを作る人が決めてよい。システムで標準的なものが決まっていることも)。

□ たとえば:

- `call` 番地 --- 呼出し。`rp` (`r12` とか決まったレジスタ) に「この命令の次の番地」を入れて、指定された番地へジャンプ。
- `ret` --- 戻り。`rp` に入っている番地へジャンプ。

□ 簡単な例題

```
1000:   0
2000:   call  rp,3000(r0)  ← call 命令の正確な形
        call  rp,3000(r0)
        call  rp,3000(r0)
3000:   load  r1,1000(r0)
        loadi r2,1(r0)
        add   r1,r2
        store r1,1000(r0)
        call  r0,0(rp)     ← ret 命令の正確な形
```

□ しかしこれだと、あるサブルーチンの中から別のサブルーチンが呼べない (なぜか?) し、ローカル変数も使えない。 →そこでスタックを使うようにする。引数の受渡しや局所変数もスタックを使う。

□ これらをすべてまとめて、関数の作り方:

- 関数の入口と出口:

```
store rp,0(sp)  ← sp の場所に戻り番地を入れる
...
load  rp,0(sp)  ←入れておいた戻り番地を戻す
call  r0,0(rp)  ← ret
```

- 関数を呼ぶところ:

```
loadi sp,-XXX(sp)  ← sp を空いている領域まで減らす
call  rp,YYYY(r0)  ←呼ぶ
loadi sp,XXX(sp)   ←戻って来たら sp を元に戻す
```

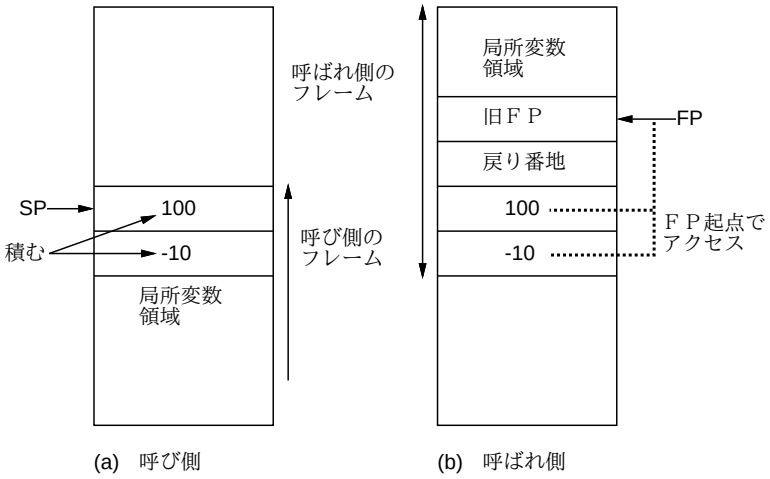
- 変数と引数と返値

```
-4(sp), -8(sp),... → ローカル変数、実引数
4(sp), 8(sp), ...  →仮引数 (受け取る引数)
r1                 →返値のうけわたし
```

8.5 引数と返値の受け渡し

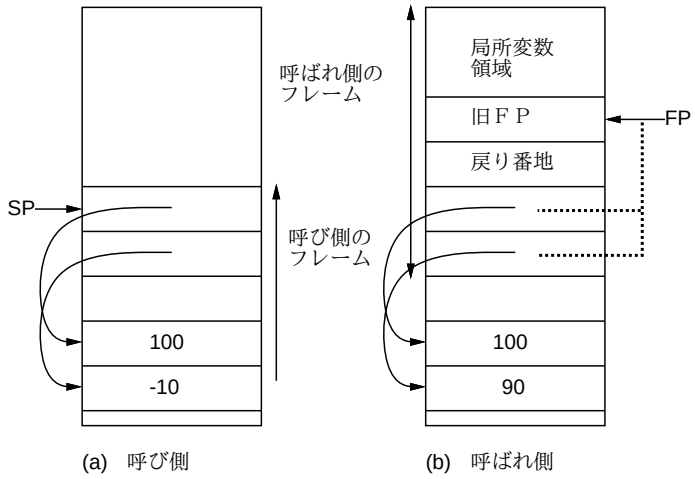
8.5.1 値呼び

□ 仮引数は「値を受け取った局所変数」。C で使われている。



8.5.2 参照呼び

□ 仮引数は「渡された変数のアドレス」。Fortran で使われている。



8.5.3 名前呼び

□ 仮引数は「渡されたものをそこに埋め込む印」。Algol などで使われていた。また、C などのマクロもこれになる。

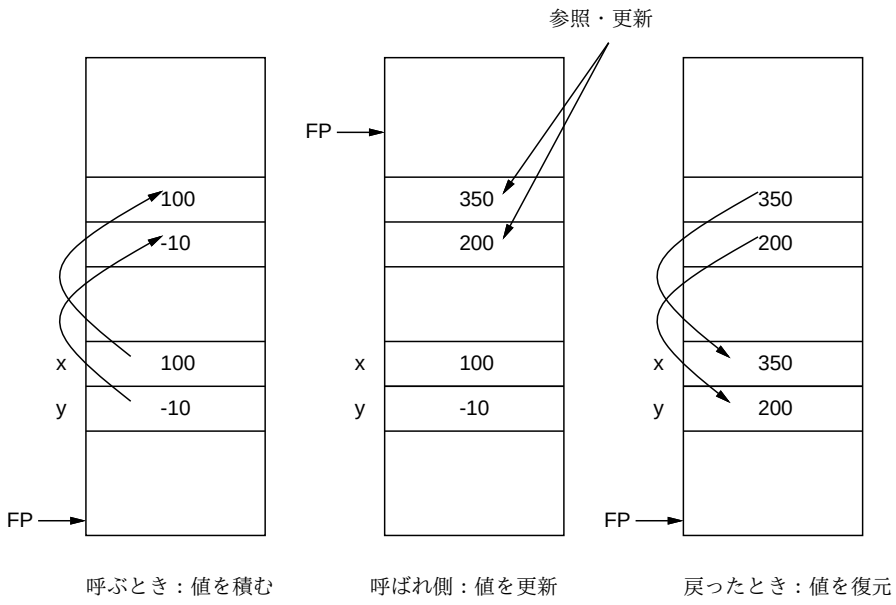
```
procedure sums(int k, a, b; real x, result)
begin
    result := 0.0;
    for k := a to b do result := result + x;
end;

sums(i, 1, 10, a[i,i], r);

r := 0.0;
for i := 1 to 10 do r := r + a[i,i];
```

8.5.4 複写復元呼び

□ 値呼びで戻って来たら値を書き戻す。参照呼びの代替になる。




```
procedure sub(integer i, j) begin i := i + 1; j := j + 1 end;
```

□ ただし次のような場合は結果が参照呼びと違って来る。

```
x := 10; sub(x, x);
```

8.5.5 レジスタ渡し

□ スタックにかえてレジスタで渡す→高速

□ 数に限り、大きさの制約→規約が複雑に

8.5.6 返値

□ レジスタで返すことが多い

□ 大きな値は?

8.5.7 レジスタの退避回復

□ 呼び側での保存 (caller-save)

□ 呼ばれ側での保存 (callee-save)

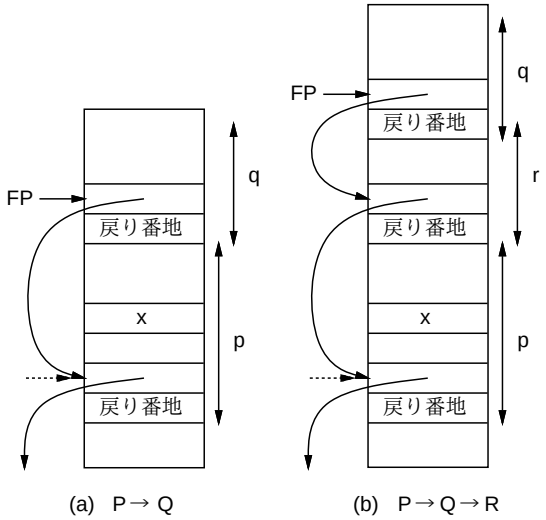
□ (OS としての規約などに依存。混ぜる場合も。)

8.6 環境の切換え

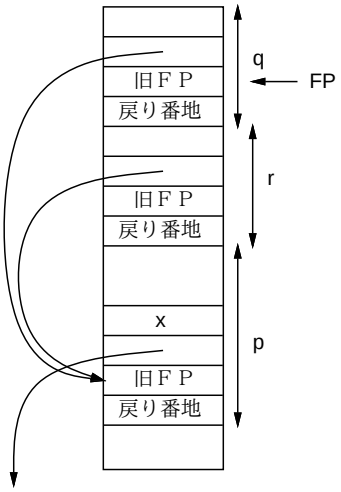
8.6.1 静的チェイン

```
procedure p;
var x: integer;
    procedure q; begin x := 1 end;
begin q end;

procedure p;
var x: integer;
    procedure q; begin x := 1 end;
    procedure r; begin q end;
begin q; r end;
```



(a) P→Q　　(b) P→Q→R

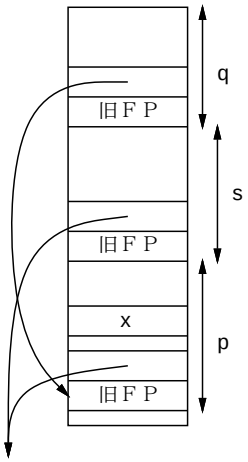


8.6.2 ディスプレイ

- □ 各レベルの環境ポインタを束にしたもの
- □ 専用レジスタにいれる場合も
- □ 必ずしも得になるとは限らない

8.6.3 手続き引数

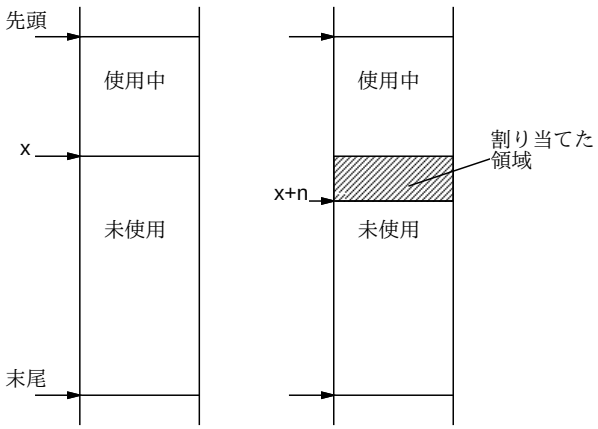
```
procedure s(px: procedure); begin px; px end;
procedure p;
var x: integer;
    procedure q; begin x := x + 1 end;
begin x := 0; s(q) end;
```



8.7 ヒープとその管理

8.7.1 ヒープの位置付けと機能

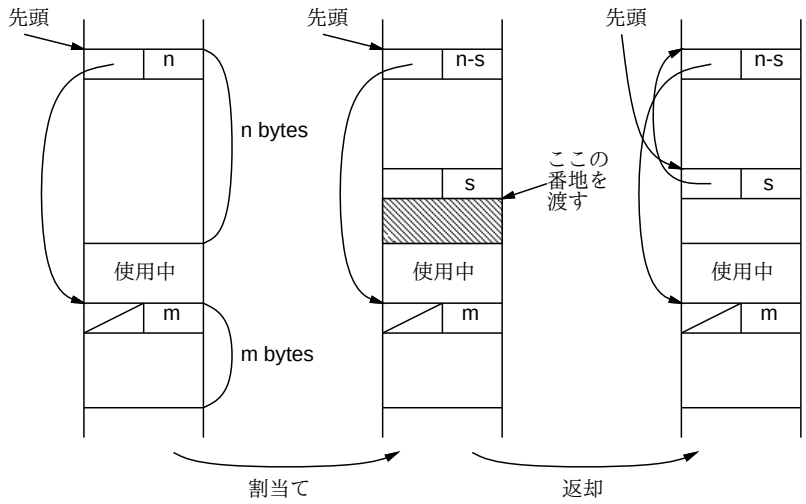
□ ヒープ→自前で割り当て (と返却) をする領域



8.7.2 手動による領域回復

□ 返却されたものを取っておいて再利用→工夫の余地

□ 断片化と漏洩の問題

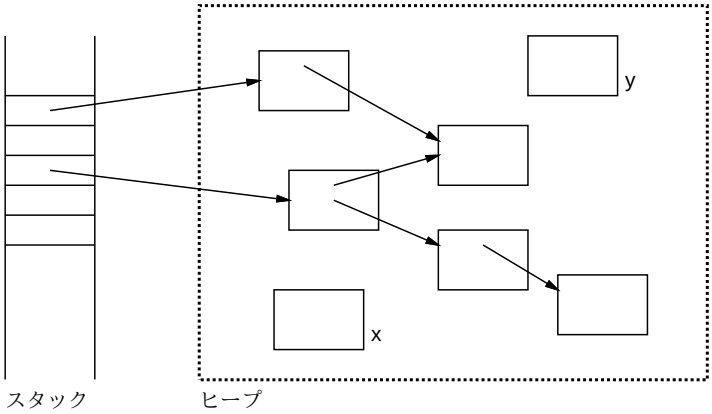


8.7.3 ごみ集め

□ 「自動的な返却」を実現する。

□ ごみ→「もはやアクセスする可能性のない領域」

□ 条件: (1) 根のアクセス、(2) スキャン、(3) ポインタの区別



8.7.4 マークスイープ型ごみ集め

□ [1.] 全ての領域に「ごみである」という印をつける.

□ [2.] たどったものについて「ごみである」という印を消す.

□ [3.] 領域を順に調べて印がついたままのものを回収

8.7.5 つめ合わせ

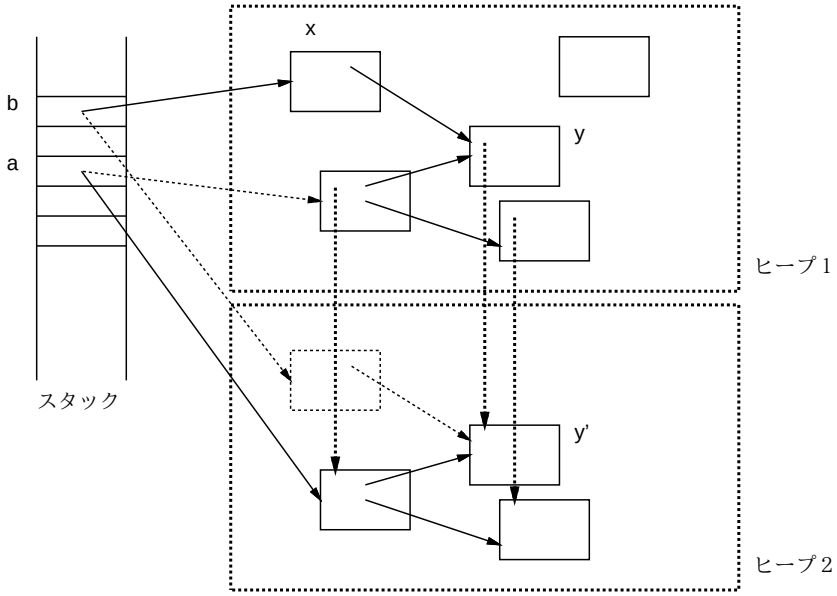
□ [1.] 「ごみは取り除い頭から詰め合せたらこの領域は何番地になるか」をヘッダに記録

□ [2.] 全ポインタを新しい番地を指すように修正

□ [3.] オブジェクトを実際に移動して詰め合せ

8.7.6 複写型ごみ集め

□ たどりながらコピーする。コピーしたものは新番地をヘッダに記録。

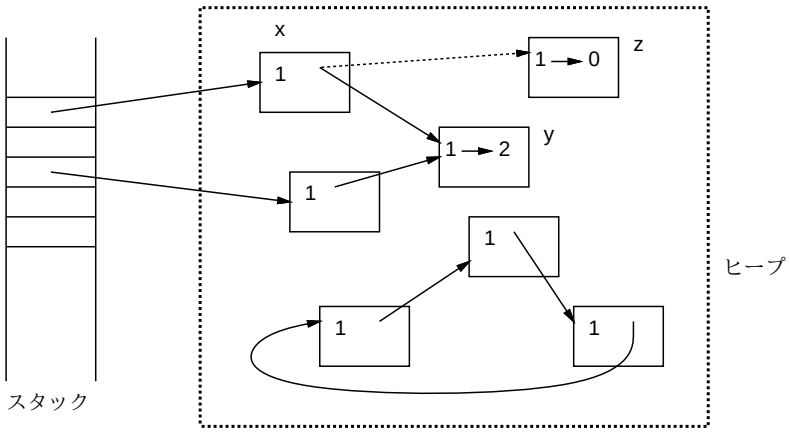


8.7.7 参照計数型ごみ集め

□ [1.] ヘッダにその領域を指すポインタ数を記憶

□ [2.] ポインタ値の複写や書換えが起こるつど計数値を更新

□ [3.] 計数値が 0 になったとき回収



8.7.8 演習問題 (3PM)

□ 「長崎 1 号」は使いにくいという評判で売れ行きが悪かったので、改良版の「長崎 2 号」を作った。

```
load   rx, 番地(ry)  : レジスタ rx に指定番地+ry の内容を転送
loadi  rx, 定数(ry)  : レジスタ rx に定数+ry を転送
store  rx, 番地(ry)  : レジスタ rx の内容を指定番地+ry に格納
add    rx, ry        : レジスタ rx の内容に ry の内容を足し込む
sub    rx, ry        : レジスタ rx の内容から ry の内容を引く
mul    rx, ry        : レジスタ rx の内容を ry の内容倍する
call   rx, 番地(ry)  : この命令の次の番地を rx に入れた後指定番地+ry へ飛ぶ
b      番地          : 指定番地に飛ぶ
beq    rx, ry, 番地  : rx==ry なら指定番地に飛ぶ(bne/bgt/bge/blt/ble もある)
```

今度はレジスタは 0〜15 まで 16 個あるが、r0 は常に内容が 0 になっている (何か転送してもよいが、読み出すと値は 0)。

□ [6-1.] この機械の呼び出し規約を定め、とあるコンパイラでコンパイルしたら

```
int func(int x, int y) { int z = x + y + y; return z; }
```

これの翻訳結果は次のようになった。ただし sp=r15、fp=r14、rp=r13、rr=r12 のことである。

```
func: store rp, 0(sp)
      load  r1, 8(sp)
      load  r2, 4(fp)
      add   r1, r2
      add   r1, r2
      store r1, -4(sp)  ※2
      load  rr, -4(sp)
      load  rp, 0(sp)
      call  r0, 0(rp)
```

□ これを主プログラムから次のように呼び出したとする。

```
      loadi sp, 1008(r0)  ← sp を適当に設定しないといけない
      loadi r1, 7(r0)
      store r1, -4(sp)
      loadi r1, 5(r0)
      store r1, -8(sp)
      loadi sp, -12(sp)
      call  rp, func(r0)  ※1
      loadi sp, 12(sp)
```

このコードを長崎 2 号になって実行したときの、※1 の直前、※2 の直後、※1 の直後のスタックとレジスタ (関係するものだけでよい) の内容を図示せよ。

□ [6-2.] 次の関数をコンパイルして fact(5) を動かしてみよ。

```
int fact(n) { if(n <= 0) return 1; else return n * fact(n-1); }
```

9　中間コード生成 (4AM)

9.1　目的コードと中間コード

□ 目的コードとは?

- 機械語やアセンブリ言語であってもよい
- それ以外の目的コードもある
- たとえばコンパイラインタプリタ、仮想機械

□ 中間コードとは?

- 目的コードにしてしまうとやりにくい最適化
- 機械依存部分の切り離し

9.2　様々な中間コード

9.2.1　木構造の中間コード

□ [○] 抽象構文木そのまま→生成は容易

□ [○] 情報を付随させやすい

□ [×] 込み入った最適化に向かない

□ [→] フロントエンドの分離によく使われる

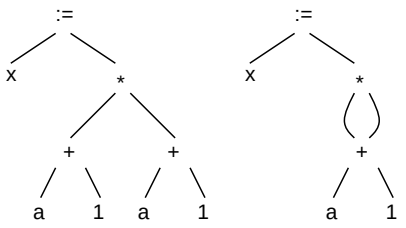
□ [→] プログラミング環境のプラットフォーム (DIANA)

9.2.2　非循環有向グラフ (DAG)

□ 部分木の共有→ DAG

- ただし、値が途中で変わるときは共有してはいけない

```
x := (a + 1) * (a + 1)
```



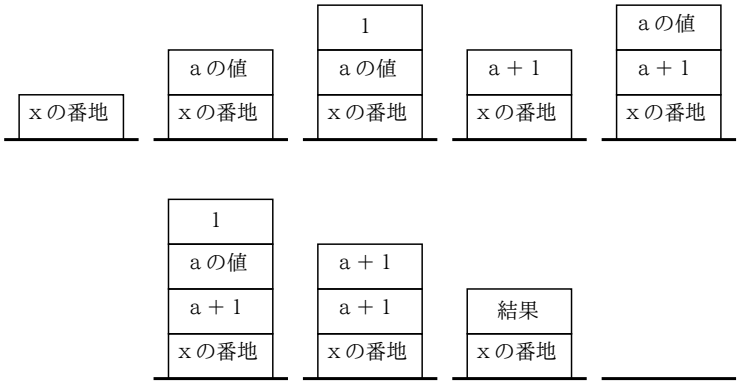
9.2.3　後置コード

□ [○] コンパクト

□ [○] 木からのコード生成が容易である.

```
x := (a + 1) * (a + 1)
```

```
lda  x  -- x の番地をスタックに積む
ld   a  -- a の値をスタックに積む
ldc  1  -- 定数 1 をスタックに積む
add     -- a + 1 の計算
ld   a  -- 再び a の値を積む
ldc  1  -- 再び 1 を積む
add     -- a + 1 の計算
mul     -- (a + 1) * (a + 1) の計算
st      -- 値を x に格納
```



□ P コードなど…移植性にすぐれる

□ コンパイラインタプリタ用

9.2.4 3 つ組

□ 命令に順に番号をつけ、ある命令の結果を参照したければその命令の番号で参照する。

- 「命令、オペランド 1、オペランド 2」から成るので「3 つ組」という。

```
(1)  add  a,1
(2)  add  a,2
(3)  mul  (1),(2)
(4)  st   (3),x
```

9.2.5 4 つ組

□ 命令番号で参照するかわりに、コンパイラが作業変数 (テンポラリ) を割り当てる。

- 「命令、オペランド 1、オペランド 2、結果」で「4 つ組」
- 読みやすく、分かりやすい
- DAG や構文木からすぐ作れる

```
x := (a + 1) * (a + 1)

t1 ← a + 1
t2 ← a + 1
t3 ← t1 * t2
x ← t3
```

9.2.6　4 つ組による中間コード生成

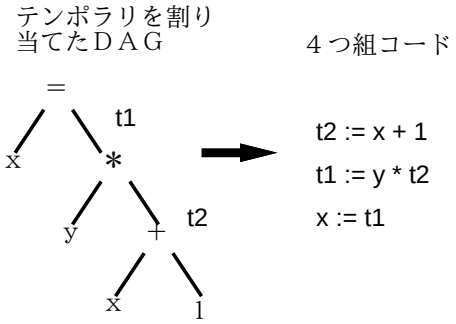
□ 以下では次の命令から成る 4 つ組を使う

```
    ○ ← ○
    ○ ← ○ + ○  (+、-、*、/ あり)
    goto Label
    if ○ = ○ goto Label (=、≠、≧、≦、＞、＜ あり)
Li:
```

なお「○」のところには次のどれかが入る
- ・変数 (a, b, c, ...)
- ・テンポラリ (t1, t2, t3, ...)
- ・定数 (100, -20, ...)

□ 式の抽象構文木や DAG から 4 つ組を生成するには、次の手順による

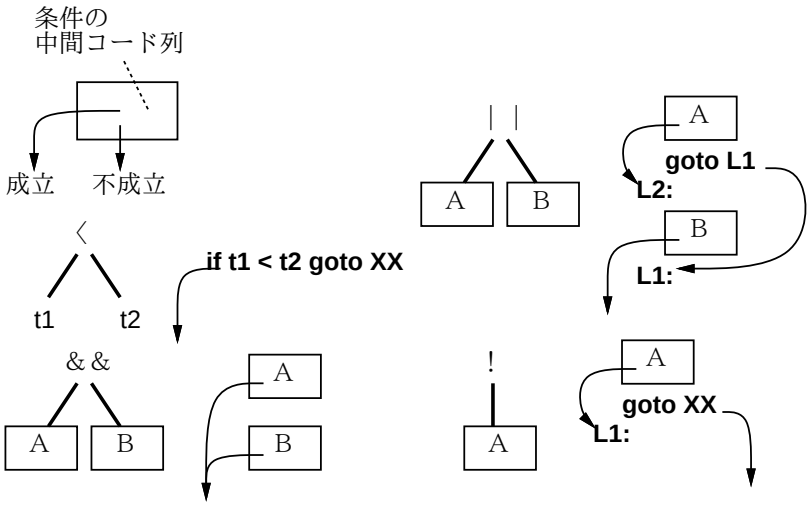
- 1.　すべての中間ノードにテンポラリを割り当てる
- 2.　木の下の方から順にノードを対応する 4 つ組に直していく



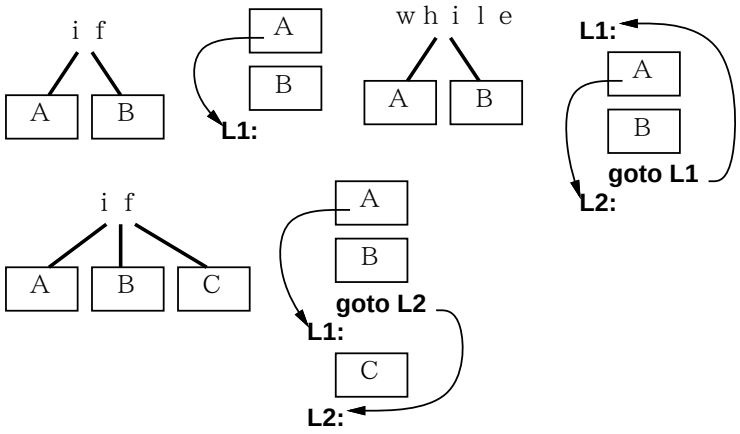
□ 条件や制御構造については、if-goto を使う。

□ 条件は、「成立」なら飛び出し、「不成立」ならそのまま下へ行くコード

- 各種の条件演算はその「配線」をつなぎかえる



□ 制御構造は「本体」のコードと条件のコードをつないで配線する



10 目的コード生成 (4AM cont.)

10.1 組単位の展開によるコード生成

□ 中間コード 1 命令ごとに対応してコード生成

- たとえば次の規則を用いる

```
a ← b
      load  r1,b (または loadi r1,b)
      store r1,a       ... a, b 等は実際は「4(sp)」等

a ← b + c
      load  r1,b
      load  r2,c
      add   r1,r2
      store r1,a

goto Li
      b Li

if a = b goto Li
      load r1,a
      load r2,b
      beq r1,r2,Li
```

□ とても簡単だが非能率的なコード

10.2 解釈実行型コード生成

□ インタプリタで記号実行しながら生成

□ 中間命令間の情報流通が可能

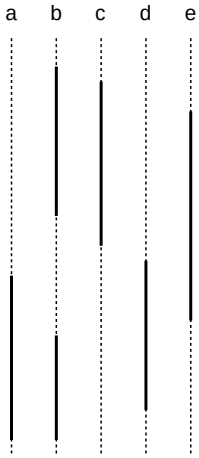
□ ブロックの境界をまたがる流通は無理

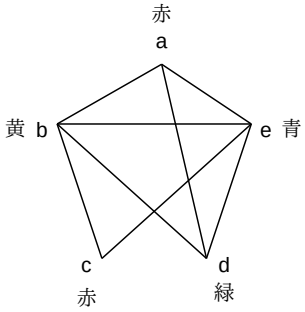
10.3 レジスタ割当てとレジスタ割付け

□ レジスタ割り当て→どのレジスタを使うか決める

□ レジスタ割り付け→変数をレジスタに置いてしまう

□ うまく割り付けないとレジスタが足りなくなる→割り付け問題





10.4　目的コードの改良とのぞき穴最適化

□ 生成されたコードを「のぞき穴」から見て改良

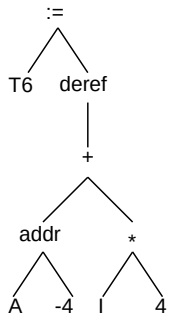
- 無効果命令の削除
- ロードストア最適化
- 定数畳み込み
- 代数的等価
- 命令選択
- アドレシングモード選択
- 不到達コードの削除
- 分岐最適化

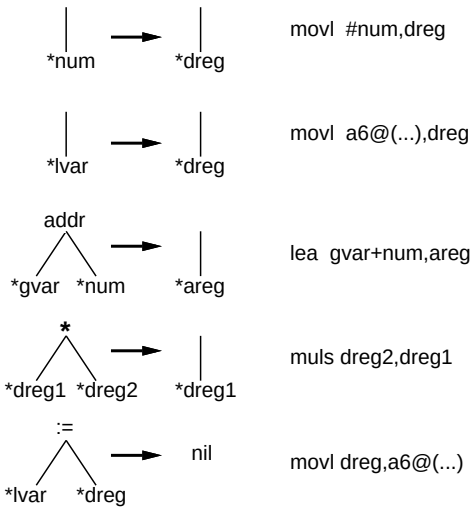
10.5　コード生成器生成系

10.5.1　記述主導型コード生成

□ 場合分けコードは大変→もっと「宣言的」にしたい

10.5.2　木のパターンマッチによるコード生成





□ マッチするパターンがなくて，書換えが止まってしまう可能性.

□ 書換えのループが起きて止まらなくなる可能性.

□ 複数のパターンがマッチしたときの選択方式.

10.5.3 Graham-Granville コード生成器

□ 木を前置記法で表し、LR パーザで解析→生成規則を実行

```
:= t6 deref + addr a -4 * i 4

dreg → num               { " movl #num,dreg" }           ; R1
areg → addr gvar num     { " lea gvar+num,areg" }        ; R2
dreg → * lvar dreg       { " muls a6@(off[lvar]),dreg" } ; R3
dreg → deref + areg dreg { " movl areg@(dreg),dreg" }    ; R4
t → := lvar dreg         { " movl dreg,a6@(off[lvar])" } ; R5

:= t6 deref + addr a -4 * i 4
→ := t6 deref A0 * i -4        (R2) lea A+-4,A0
→ := t6 deref A0 * i D0        (R1) movl #4,D0
→ := t6 deref A0 D0            (R3) muls a6@(-4),D0
→ := t6 D0                     (R4) movl A0@(D0),D0
→ T                            (R5) movl D0,a6@(-20)
```

10.5.4 Davidson-Fraser コード生成器

□ 命令を RTL で表す→ RTL の上で接合変形し機械語命令の形に

```
  r[y] := m[r[x] + 8] + r[y]

*r[16] := r[14] + -4     -- 局所変数 i の番地
*r[17] := m[r[16]]       -- i の値
*r[18] := r[17] * 4
 r[19] := a - 4          -- 配列 a の先頭マイナス 4 番地
*r[20] := r[18] + r[19]  -- a[i] の番地
 r[21] := m[r[20]]       -- a[i] の値

 r[17] := m[r[14] + -4]        -- movl a6@(-4),rx に相当
 r[19] := a - 4
 r[20] := (r[17] * 4) + r[19]  -- lea ax@(0,rx:1:4),rx に相当
 r[21] := m[r[22]]
```

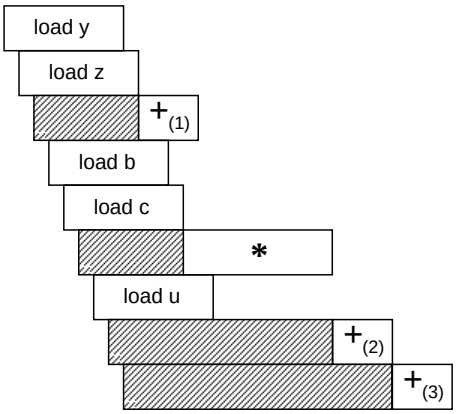
```
r[17] := m[r[14] + -4]
r[19] := a - 4
r[21] := m[(r[17] * 4) + r[19]] -- movl ax@(0,rx:1:4),rx に相当
```

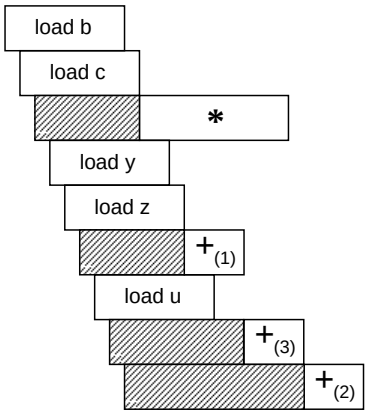
10.6 様々な目的機械の特性への対処

10.6.1 命令スケジューリング

□ 命令の実行時間特性は機械によって違う

```
x := y + z; a := b * c; t := a + u + x;
```





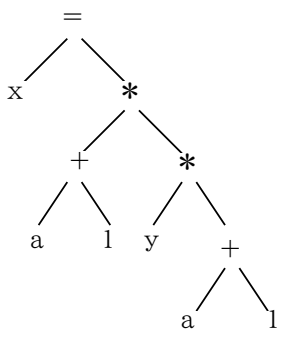
10.6.2 RISC アーキテクチャ

10.6.3 ベクトルプロセッサとマルチプロセッサ

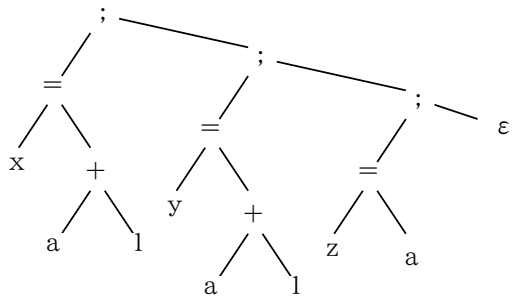
10.7 練習問題 (4PM)

□ [7-1.] 次の抽象構文木を DAG に変換せよ。

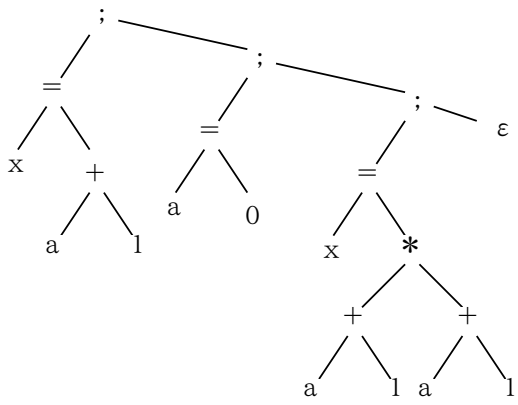
（a）



（b）



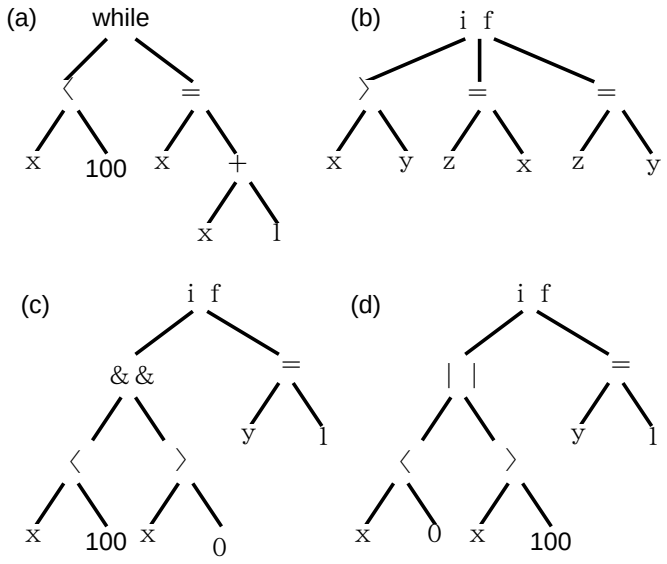
（c）



□ [7-2.] 上の (a)〜(c) を 4 つ組コードにせよ

□ [7-3.] 上の (a)〜(c) の [7-1.] で作った DAG 版から 4 つ組コードにせよ

□ [7-4.] 次の抽象構文木を 4 つ組コードにせよ



11 最適化 (4PM)

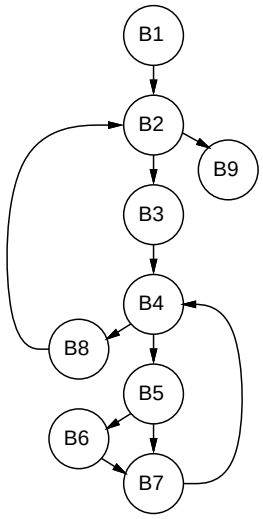
11.1 最適化の原理と分類

- □ [●] 最適化 (コードの改良) とは具体的には?
- □ [(a)] 実行しなくてもよい命令列を発見し，取り除く
- □ [(b)] 複数回実行されるがその間で結果が変らない命令列を発見し 1 回の実行ですませる
- □ [(c)] 複数回実行される命令列を，より少ない実行回数ですませる
- □ [(d)] ある命令列をそれと同じ結果をもたらすより高速な命令
- □ [●] 手続き内最適化/手続き間最適化

11.2 最適化のためのコード解析

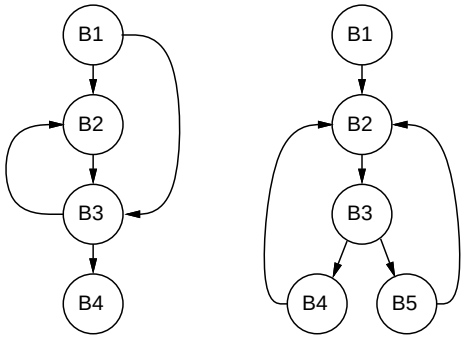
11.2.1 基本ブロックとフローグラフ

- □ 基本ブロック…途中での出入りのない一連の命令列
- □ ブロック内 (局所) 最適化 vs 広域最適化
- □ 広域最適化→フローグラフを作成し情報の抽出



11.2.2 制御フロー解析

- □ おもに、ループをみつけるのが目的
- □ フローグラフ G に含まれる節 d が節 n を「支配する」(dominate) とは，グラフの出発点から n に至る全ての経路が d を通ること
- □ フローグラフ G に含まれる辺 b → h が「帰辺」であるとは，h が b を支配する節である場合
- □ フローグラフ G に含まれる帰辺 b → h に関する「自然ループ」(natural loop) とは，n から b へ h を通らずに行ける経路があるような節 n(b 自身を含む) と h を合せたもの．h をループの「ヘッダ」(header) とよぶ
- □ 自然でないループもあるが、扱わない
- □ 帰辺を 1 つに、入口も 1 つに変換しておく



11.2.3 データフロー解析

□ フローグラフに即して最適化に必要な各種情報を得ること

```
    B1                      B5                      B6
S1:  (MOV 1 NOTYET)     S7:  (SUB I 1 T2)       S18: (MOV T6 X)
    B10                 S8:  (MUL T2 4 T3)      S19: (MOV* T11 T5)
    B2                  S9:  (ADDR A 0 T4)      S20: (MOV* X T10)
S2:  (IFEQ NOTYET 0 B9) S10: (ADD T4 T3 T5)     S21: (MOV 1 NOTYET)
    B3                  S11: (DEREF T5 T6)          B7
S3:  (MOV 0 NOTYET)     S12: (ADD I 1 T7)       S22: (ADD I 1 T12)
S4:  (MOV 1 I)          S13: (SUB T7 1 T8)      S23: (MOV T12 I)
    B11                 S14: (MUL T8 4 T9)      S24: (JMP B4)
    B4                  S15: (ADD T4 T9 T10)        B8
S5:  (SUB N 1 T1)       S16: (DEREF T10 T11)    S25: (JMP B2)
S6:  (IFGT I T1 B8)     S17: (IFLE T6 T11 B7)       B9
```

11.2.4 生きている変数の問題

□ ある変数が任意の位置で保存すべきかどうか調べる

```
LiveOut[b] = ∪_{b' ∈ Succ[b]} LiveIn[b']
LiveIn[b]  = Ref[b] ∪ (LiveOut[b] - Ass[b])
```

□ [1.] まず，各ブロックについて Ref[b]，Ass[b] を求める．

□ [2.] LiveIn[b]，LiveOut[b] についてはとりあえず空集合とする．

□ [3.] 各ブロックについて上の方程式に従って LiveIn[b]，LiveOut[b] を計算することを反復することをもはや各集合が変化しなくなるまで行う．

```
>(comp-liveness '(a n))
NIL
>(for b *blocks*
      (format t "~% ~A ~A ~A" b (block-livein b) (block-liveout b)))
 B1 (A N) (NOTYET N A)
 B10 (NOTYET N A) (NOTYET N A)
 B2 (NOTYET N A) (N A)
 B3 (N A) (N A I NOTYET)
 B11 (N A I NOTYET) (N A I NOTYET)
 B4 (N A I NOTYET) (A I N NOTYET)
 B5 (I N A NOTYET) (T10 T11 T5 T6 I N A NOTYET)
 B6 (T10 T11 T5 T6 I N A) (I N A NOTYET)
 B7 (I N A NOTYET) (N A I NOTYET)
 B8 (NOTYET N A) (NOTYET N A)
 B9 NIL NIL
```

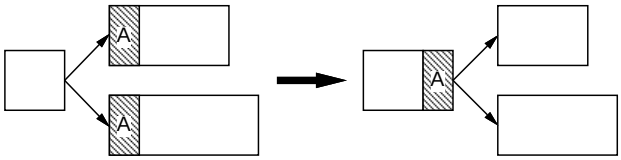
□ UD 連鎖、DU 連鎖、利用可能式、コピー文などの各種問題

11.2.5 記号実行と範囲解析

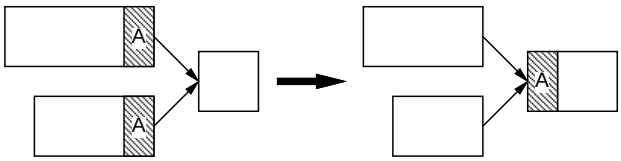
- □ 記号実行: 翻訳時にできる範囲で実行してみること
- □ 範囲解析: 変数の値の上限、下限を調べる

11.3 一般の各種最適化

- □ 定数畳み込み
- □ 数学的等価
- □ 共通式の除去
- □ コピー伝播
- □ 不要コード削除
- □ 不到達コード削除
- □ ホイスティング



左向きホイスティング



右向きホイスティング

11.4 ループ最適化

11.4.1 ループ不変文の移動

- □ [1.] オペランドが定数，または到達する定義が全て L 外にある変数のみから成る文に「ループ不変」の印をつける.
- □ [2.] オペランドが定数，または到達する定義が全て L 外にある変数，または到達する定義が 1 つだけであり，それが L に含まれていて，その定義に「ループ不変」の印がついているような変数のみから成る文にも「ループ不変」の印をつける.
- □ [(a)] s は必ず 1 回以上実行される.
- □ [(b)] x と同じ変数を定義する文は L 内では s だけである.
- □ [(c)] L 内の全ての x の使用について，そこに到達する定義は s だけである.
- □ [(a')] s は必ず 1 回以上実行されるか，または x は L 外では参照されない.

```
i := 0;
while(i < 10 && x > 0) {
    a[i] := y / x; i := i + 1;
```

```
i := 0;
t1 := y / x;
while(i < 10 && x > 0) {
    a[i] := t1; i := i + 1;

i := 0;
if(i < 10 && x > 0) {
    t1 := y / x;
    while(i < 10 && x > 0) {
        a[i] := t1; i := i + 1; }

i := 0;
if(x > 0) {
    t1 := y / x;
    while(i < 10) {
        a[i] := t1; i := i + 1; }
```

11.4.2 ループ内分岐の移動

```
t1 := ...
while(...) {
    C;
    if(t1)
        A;
    else
        B;
    D;

t1 := ...
if(t1)
    while(...) {
        C; A; D;
else
    while(...) {
        C; B; D;
```

11.4.3 帰納変数の最適化

□ 永続ループ変数: ループ内でまず参照、次に定義される

□ 帰納変数: ループ周回ごとに一定値ずつ増減

```
i := 0;                          p := top;
while i < x do begin             while p <> nil do begin
    ...                              ...
    i := i + 1                       p := p^.next
end                              end
```

□ 変数 i が永続ループ変数であり，かつループ内での i に対する定義 s が i := i ± C(ただし C はループ不変) の形であり，s がループ周回ごとに必ず 1 回実行されるならば i は帰納変数である.

□ 変数 j が永続ループ変数でなく，かつループ内の j に対する定義 s が j := i * c ± d(ただし i は帰納変数，c，d はループ不変) の形であり，s がループ周回ごとに必ず 1 回実行されるなら j は帰納変数である.

□ ここで i が基本帰納変数であるとき，j は i の「家族」(family) であるという. i が基本帰納変数でないとき，i は別の基本帰納変数 i0 の家族なので,

- [1.] i への定義と s の間に i0 への定義がはさまっていない.
- [2.] ループ外の i への定義が s に到達することがない.

□ この 2 条件が成り立てば j も i0 をもとに計算するように (つまり i0 の家族に) できる

```
i0 = 0;                          i0 = 0;
while(...) {                     while(...) {
    i = i0 * 2 + 1;                  j = i * 2 + 3;   /* s */
    ...                              i = i0 * 2 + 1;
    i0 = i0 + 1;                     ...
    j = i * 2 + 3;  /* s */          i0 = i0 + 1;
    ...
 }
```

□ [1.] 新しい変数 u を導入し，その初期値設定をループのプリヘッダに追加する.

□ [2.] 家族の基となっている変数の更新の直後に，u の増減を行う定義を追加する.

□ [3.] j の定義を j := u に取り替える.

```
i0 = 0; u = 1; v = 5;
while(...) {
    i = u;
    ...
    i0 = i0 + 1; u = u + 2; v = v + 4; j = v;
    ...
```

11.4.4 ループ展開

□ ループ内部を複数回反復すること

```
while(C) {              while(C) {
    B;                      B;
}                           if(!C) break;
                            B;
```

11.4.5 ループ融合

□ 複数ループを 1 つにまとめること

```
for i := 1 to n do a[i] := i;       for i := 1 to n do begin
for i := 1 to n do b[i] := n - i;       a[i] := i; b[i] := n - 1
                                    end;
```

11.4.6 線形化

□ 入れ子のループを 1 重に直すこと

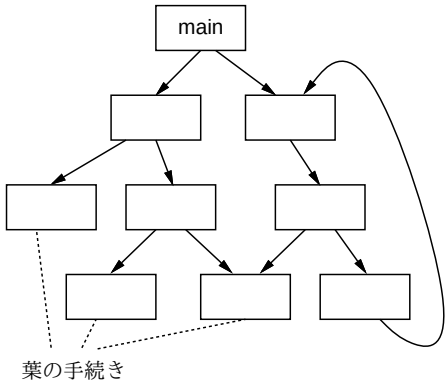
```
a: array[1..10][1..10] of ...;
...
for i := 1 to 10 do
    for j := 1 to 10 do a[i][j] := 0.0;

for i := 1 to 100 do a[i] := 0.0;
```

11.5 手続き間最適化

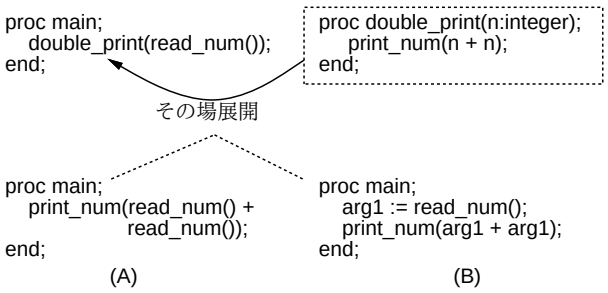
11.5.1 呼び出しグラフ

□ 呼び出しグラフを作成し、葉からやる。



11.5.2 その場展開

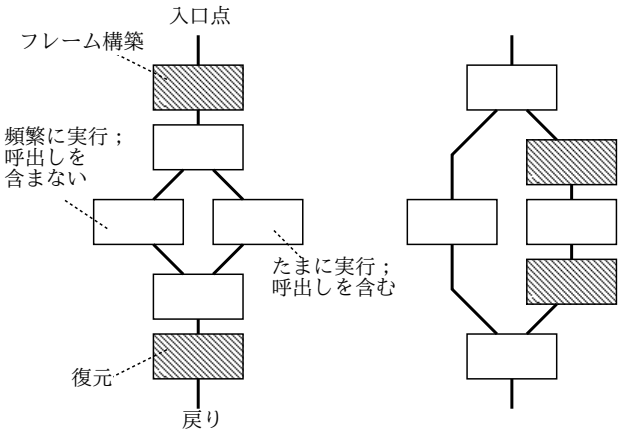
□ 呼び出す代わりにその場所に本体を埋め込む



11.5.3 入口と出口の簡略化

□ レジスタ退避回復をできるだけ避ける

□ 葉の手続きであればフレーム構築を略す



11.5.4 手続き間レジスタ割付け

□ 手続きごとに使うレジスタをよける

□ 引数や返値も転送が少なくなるように割り当てる

11.6　最後の演習

考え中…

12　最後に