コンパイラ

湯淺太一

講義資料のダウンロード

URL:
www.yuasa.kuis.kyoto-u.ac.jp/~yuasa/index_J.html

「コンパイラ」講義資料（パスワードが必要）

ユーザー名：compiler
パスワード：konpaira

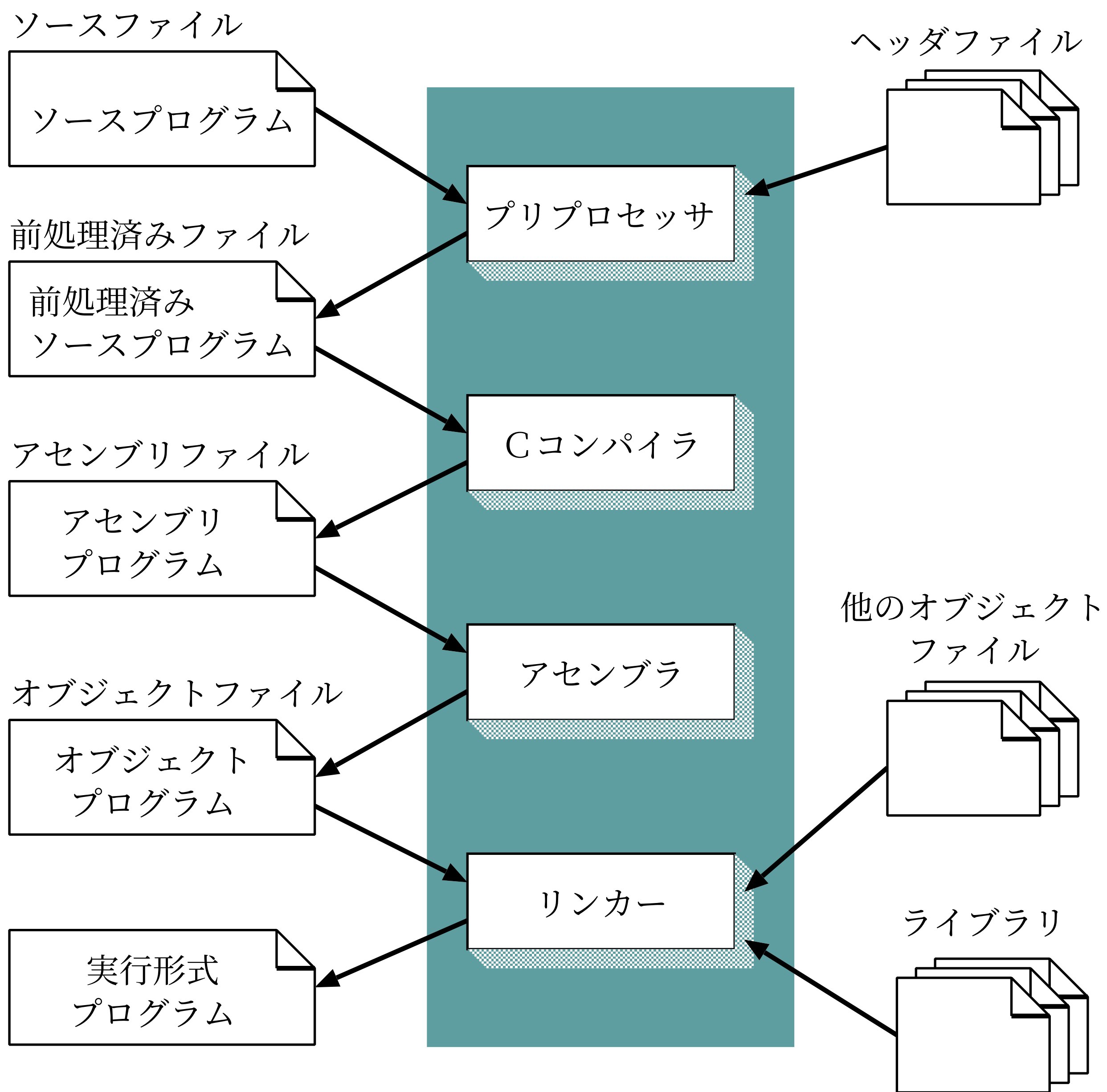
「コンパイラ」講義資料

PostScript版
1up 2up 4up
PDF版
1up 2up 4up

第１章

コンパイラの概要

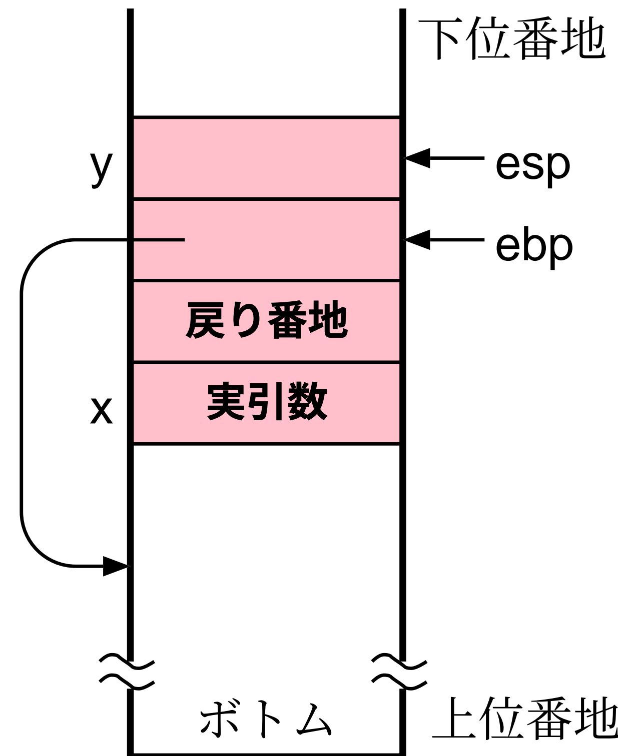
Cコンパイラ



コード生成例

```
int foo(int x) {
    int y = x*x;
    return y+2;
}
```

```
 1  _foo: push  ebp
 2        mov   ebp,esp
 3        sub   esp,4
 4        mov   eax,8[ebp]
 5        imul  eax,8[ebp]
 6        mov   -4[ebp],eax
 7        mov   eax,-4[ebp]
 8        add   eax,2
 9        mov   esp,ebp
10        pop   ebp
11        ret
```



機械語コード

```
0101 0101 1000 1001 1110 0101 1000 0011
1110 1100 0000 0100 1000 1011 0100 1101
0000 1000 0000 1111 1010 1111 0100 1101
0000 1000 1000 1001 0100 1101 1111 1100
1000 1011 0101 0101 1111 1100 1000 0011
1100 0010 0000 0010 1000 1001 1101 0000
1100 1001 1100 0011
```

```
 1  0101 0101
 2  1000 1001 1110 0101
 3  1000 0011 1110 1100 0000 0100
 4  1000 1011 0100 1101 0000 1000
 5  0000 1111 1010 1111 0100 1101 0000 1000
 6  1000 1001 0100 1101 1111 1100
 7  1000 1011 0101 0101 1111 1100
 8  1000 0011 1100 0010 0000 0010
 9  1000 1001 1101 0000
10  1100 1001
11  1100 0011
```

`mov`命令

1000 1001 11x_1x_2 $x_3y_1y_2y_3$

レジスタ$x_1x_2x_3$からレジスタ$y_1y_2y_3$へ移動する．

1000 1011 01x_1x_2 $x_3y_1y_2y_3$ $i_1i_2i_3i_4$ $i_5i_6i_7i_8$

レジスタ$y_1y_2y_3$の指す位置から相対的に
$i_1i_2i_3i_4i_5i_6i_7i_8$番地の位置にあるデータを，レジスタ$x_1x_2x_3$へ移動する．

1000 1001 01x_1x_2 $x_3y_1y_2y_3$ $i_1i_2i_3i_4$ $i_5i_6i_7i_8$

レジスタ$y_1y_2y_3$ の指す位置から相対的に
$i_1i_2i_3i_4i_5i_6i_7i_8$番地の位置へ，
レジスタ$x_1x_2x_3$のデータを移動する．

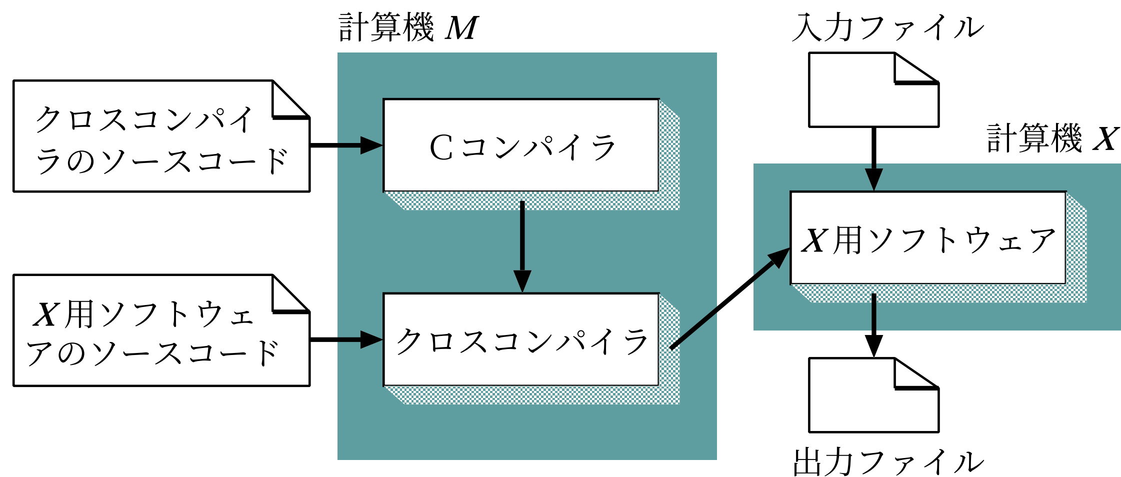
AT&T形式

```
 1  _foo: pushl %ebp
 2        movl  %esp,%ebp
 3        subl  $4,%esp
 4        movl  8(%ebp),%eax
 5        imull 8(%ebp),%eax
 6        movl  %eax,-4(%ebp)
 7        movl  -4(%ebp),%eax
 8        addl  $2,%eax
 9        movl  %ebp,%esp
10        popl  %ebp
11        ret
```

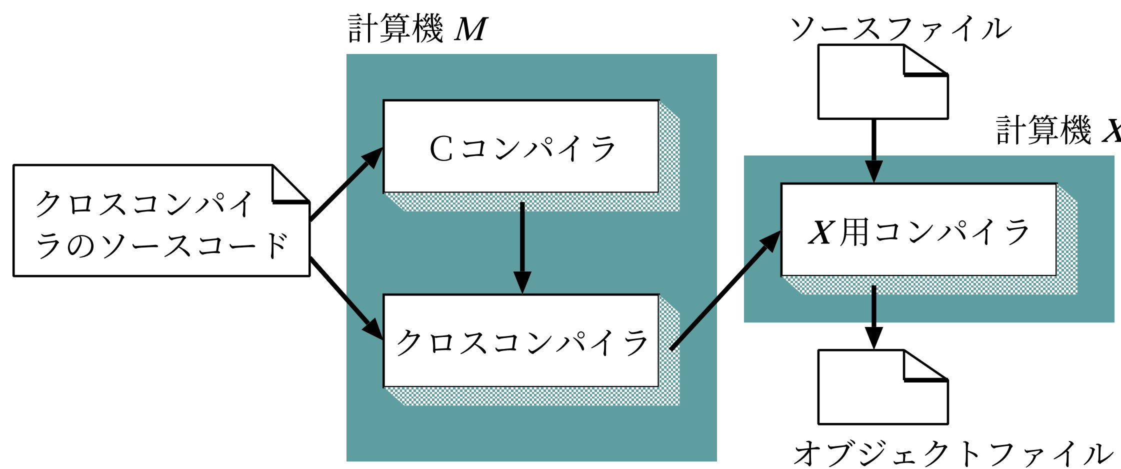
コンパイラとは

ソース言語から目的言語への変換

クロスコンパイル



ブートストラップ



コンパイラの構造

1. 字句解析
2. 構文解析
3. 意味解析
4. コード生成

字句解析

字句要素を切り出して，トークン列を生成.

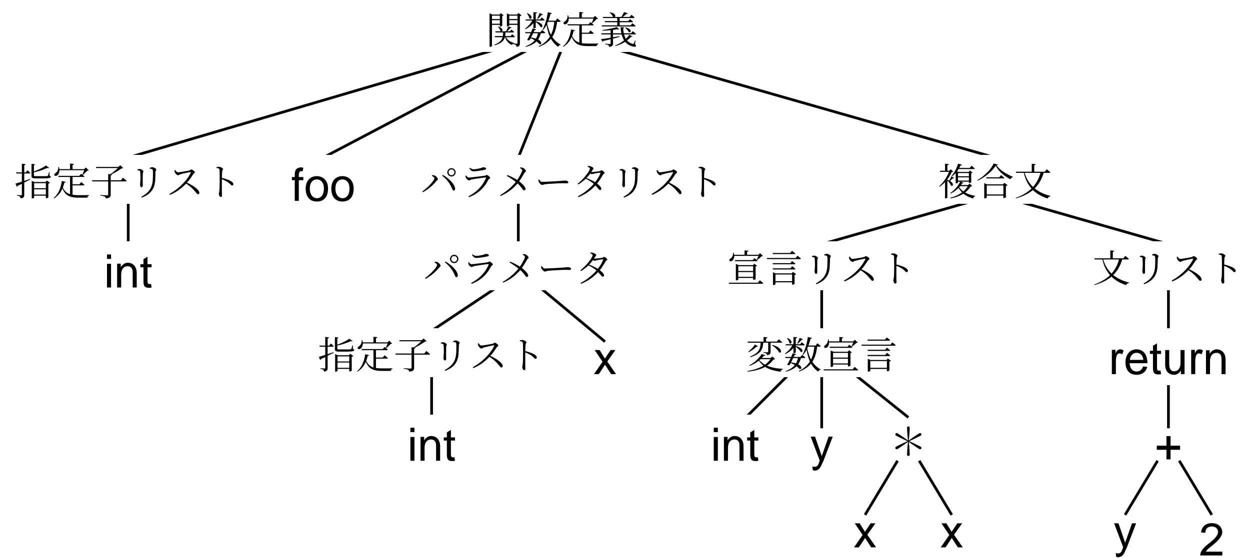
C 言語の字句要素

1. 識別子
2. キーワード
3. 定数（整数定数，浮動小数点定数，文字定数）
4. 文字列リテラル
5. 演算子
6. 区切り記号

int	foo	(	int	x	)	{	int	y	=	x	*	x	;	return	y	+	2	;	}

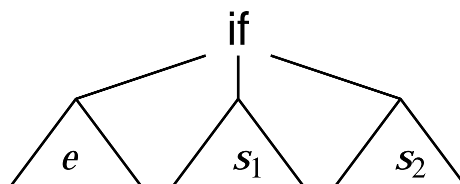
構文解析

文法をチェックし，構文木を生成.

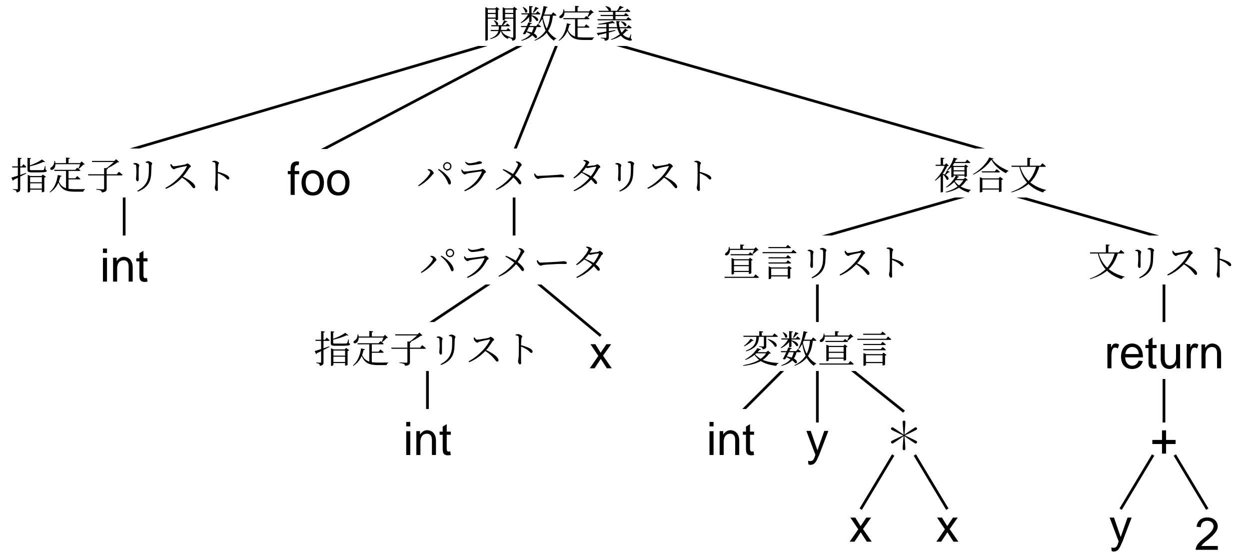


必要な情報のみ保持

`if ( 条件 ) 文1 else 文2`



意味解析



コード生成

局所変数等の割り付け

最適化

```
1   _foo: mov  eax,4[esp]
2         imul eax,eax
3         add  eax,2
4         ret
```

第 2 章

字　句　解　析

文字列の計算

アルファベット：言語を構成する文字の（有限）集合

文字列: 文字の並び

空の文字列: 文字を一つも含まない文字列ε

文字列の連結$s \cdot t$

例：$\mathtt{aa} \cdot \mathtt{bc} = \mathtt{aabc}$

結合則$s \cdot (t \cdot u) = (s \cdot t) \cdot u$が成り立つ

$\rightarrow s \cdot t \cdot u$と書いてよい

$\rightarrow stu$と略記することもある

文字列のべき乗

$s^0 = \varepsilon$

$s^n = s \cdot s^{n-1} \quad (n = 1, 2, 3, \ldots)$

公式

$s^1 = s \cdot s^0 = s \cdot \varepsilon = s$

$s^{i+j} = s^i \cdot s^j$

文字列集合の連結

$A \cdot B = \{s \cdot t \mid s \in A,\ t \in B\}$

例：$\{\mathsf{a}, \mathsf{bc}\} \cdot \{\mathsf{a}, \mathsf{c}\} = \{\mathsf{aa}, \mathsf{ac}, \mathsf{bca}, \mathsf{bcc}\}$

結合則 $A \cdot (B \cdot C) = (A \cdot B) \cdot C$ が成り立つ

→ $A \cdot B \cdot C$ と書いてよい

→ ABC と略記することがある

文字列集合のべき乗

$A^0 = \{\varepsilon\}$

$A^n = A \cdot A^{n-1} = \underbrace{A \cdot A \cdot \cdots \cdot A}_{n} \quad (n = 1, 2, 3, \ldots)$

例：$A = \{\mathsf{a}, \mathsf{bc}\}$ のとき

$A^3 = A \cdot A \cdot A = \{\mathsf{aaa}, \mathsf{aabc}, \mathsf{abca}, \mathsf{abcbc}, \mathsf{bcaa}, \mathsf{bcabc}, \mathsf{bcbca}, \mathsf{bcbcbc}\}$

公式

$A^1 = A \cdot A^0 = A \cdot \{\varepsilon\} = A$

$A^{i+j} = A^i \cdot A^j = \underbrace{\overbrace{A \cdot A \cdot \cdots \cdot A}^{i} \cdot \overbrace{A \cdot A \cdot \cdots \cdot A}^{j}}_{i+j}$

$\{\varepsilon\}$ と空集合

$\{\varepsilon\} \cdot A = A \cdot \{\varepsilon\} = A$

$\phi \cdot A = A \cdot \phi = \phi$

文字列集合の正の閉包(positive closure)

$$A^+ = A^1 \cup A^2 \cup A^3 \cup \cdots = \bigcup_{i=1}^{\infty} A^i$$

例：

$$\{\mathtt{a}\}^+ = \{\mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \ldots\}$$

$$\{\mathtt{a}, \mathtt{b}\}^+ = \{\mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \ldots\}$$

文字列集合の閉包(closure)

$$A^* = \{\varepsilon\} \cup A^+ = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots = \bigcup_{i=0}^{\infty} A^i$$

例：

$$\{\mathtt{a}\}^* = \{\varepsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \mathtt{aaaa}, \ldots\}$$

公式

$$A^* \cdot A = A \cdot A^* = A^+$$

$$A^* \cdot A^+ = A^+ \cdot A^* = A^+$$

$$A^* \cdot A^* = A^*$$

例：C言語における10進定数

$$\{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\} \cdot \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}, \mathtt{5}, \mathtt{6}, \mathtt{7}, \mathtt{8}, \mathtt{9}\}^*$$

正規表現

アルファベット $\Sigma = \{a_1, \cdots, a_n\}$ 上の正規表現

1. ε
2. $a_i \quad (a_i \in \Sigma)$
3. $r_1 \,|\, r_2$ （r_1, r_2 は正規表現）
4. $r_1 \cdot r_2$ （r_1, r_2 は正規表現）
5. r^* （r は正規表現）

$L(r)$：正規表現 r が表す文字列集合

$L(\varepsilon) = \{\varepsilon\}$

$L(a_i) = \{a_i\}$

$L(r_1 \,|\, r_2) = L(r_1) \cup L(r_2)$

$L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$

$L(r^*) = L(r)^*$

例：$\mathtt{a} \cdot ((\mathtt{a} \,|\, \mathtt{b})^*)$ は，$\{\mathtt{a}, \mathtt{b}\}$ 上の正規表現． $\mathtt{a}\,(\mathtt{a} \,|\, \mathtt{b})^*$ と略記

$$
\begin{aligned}
&L(\mathtt{a}(\mathtt{a} \,|\, \mathtt{b})^*) \\
&\quad = L(\mathtt{a}) \cdot L((\mathtt{a} \,|\, \mathtt{b})^*) = L(\mathtt{a}) \cdot (L(\mathtt{a} \,|\, \mathtt{b}))^* = L(\mathtt{a}) \cdot (L(\mathtt{a}) \cup L(\mathtt{b}))^* = \{\mathtt{a}\} \cdot (\{\mathtt{a}\} \cup \{\mathtt{b}\})^* \\
&\quad = \{\mathtt{a}\} \cdot \{\mathtt{a}, \mathtt{b}\}^* = \{\mathtt{a}\} \cdot \{\varepsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \mathtt{aaa}, \ldots\} \\
&\quad = \{\mathtt{a}, \mathtt{aa}, \mathtt{ab}, \mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \mathtt{abb}, \mathtt{aaaa}, \ldots\}
\end{aligned}
$$

正規表現rの解釈

- $\varepsilon \mid r$：rが省略可能
- a_i：a_i自体
- $r_1 \mid r_2$：r_1またはr_2
- $r_1 \cdot r_2$：r_1のあとにr_2
- r^*：rが0個以上

略記法

- $r_{opt} = \varepsilon \mid r$
- $r^+ = r \cdot r^*$

C言語の字句要素

- 識別子: $\alpha\,(\alpha \mid \delta)^*$
- キーワード: $\texttt{auto} \mid \texttt{break} \mid \texttt{case} \mid \texttt{char} \mid \cdots$
- 演算子: $\texttt{+} \mid \texttt{\&} \mid \texttt{=} \mid \texttt{+=} \mid \cdots$
- 区切り記号: $\texttt{,} \mid \texttt{;} \mid \texttt{(} \mid \texttt{)} \mid \cdots$
- 10進定数: $(\texttt{1} \mid \cdots \mid \texttt{9})\,\delta^*\langle$オプション$\rangle$
- 浮動小数点定数: $((\delta^+\,\texttt{.}\,\delta^* \mid \texttt{.}\,\delta^+)\,\langle$指数部$\rangle_{opt} \mid \delta^+\langle$指数部$\rangle)\,(\texttt{F} \mid \texttt{f} \mid \texttt{L} \mid \texttt{l})_{opt}$
- 文字定数: $\texttt{L}_{opt}$ ’ (\\⟨文字⟩ | ⟨\\と’と改行以外⟩)$^+$ ’

ここで,

- $\alpha = \texttt{_} \mid \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z}$
- $\delta = \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \cdots \mid \texttt{9}$
- $\langle$オプション$\rangle = ((\texttt{U} \mid \texttt{u})\,(\texttt{L} \mid \texttt{l})_{opt} \mid (\texttt{L} \mid \texttt{l})\,(\texttt{U} \mid \texttt{u})_{opt})_{opt}$
- $\langle$指数部$\rangle = (\texttt{E} \mid \texttt{e})\,\texttt{-}_{opt}\,\delta^+$

有限オートマトン (finite automaton)

有限個の状態を持ち，文字を 1 文字ずつ読み込むことによって，自分自身の状態を変化させる，一種の仮想計算機

FA は五つ組 $\langle \Sigma, Q, \Delta, s, F \rangle$ で定義

Σ: アルファベット

Q: 状態全体の集合

Δ: 状態遷移 $\langle q_1, a, q_2 \rangle$ の集合

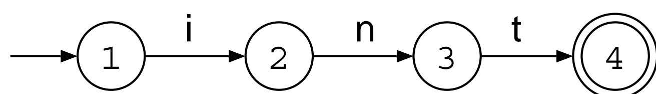
状態 q_1 で a を読んだら状態 q_2 に遷移

s: 初期状態 ($s \in Q$)

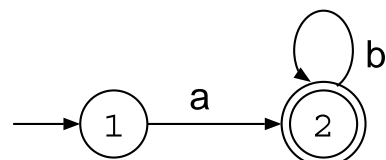
F: 受理状態の集合 ($F \subset Q$)

状態遷移図

$\mathrm{M1} = \langle \Sigma, \{1,2,3,4\}, \{\langle 1, \mathtt{i}, 2\rangle, \langle 2, \mathtt{n}, 3\rangle, \langle 3, \mathtt{t}, 4\rangle\}, 1, \{4\}\rangle$



$\mathrm{M2} = \langle \Sigma, \{1,2\}, \{\langle 1, \mathtt{a}, 2\rangle, \langle 2, \mathtt{b}, 2\rangle\}, 1, \{2\}\rangle$

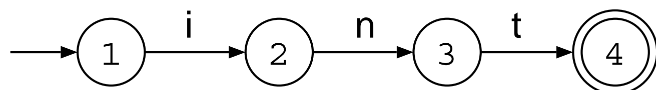


$M = \langle \Sigma, Q, \Delta, s, F \rangle$: FA
$\alpha = a_1 a_2 \cdots a_n$: 入力文字列
次の条件を満たす状態列$q_0, q_1, \ldots, q_n$が存在するとき，Mはαを受理(accept)する．

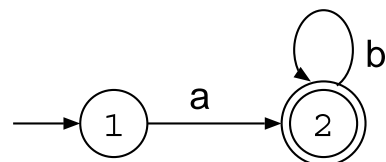
$$s = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_n \in F$$

$L(M)$: Mが受理する文字列全体
→ Mが受理する言語(language)

例：
$L(\mathrm{M1}) = \{\texttt{int}\}$



$L(\mathrm{M2}) = \{\mathtt{a}, \mathtt{ab}, \mathtt{abb}, \mathtt{abbb}, \ldots\} = L(\mathtt{ab}^*)$



$L(M) = L(M')$のとき，MとM'は等価(equivalent)

決定性(deterministic)有限オートマトン

任意の$q \in Q$と$a \in \Sigma$に対して，$\langle q, a, q' \rangle$の形の状態遷移が高々一つ

DFA $\langle \Sigma, Q, \Delta, s, F \rangle$の状態遷移関数

$$\delta(q,a) = \begin{cases} q' & (\langle q, a, q' \rangle \in \Delta \text{のとき}) \\ \bot & (\text{その他の場合}) \end{cases}$$

δが与えられれば，

$$\Delta = \{\langle q, a, \delta(q,a) \rangle \mid q \in Q, a \in \Sigma, \delta(q,a) \neq \bot\}$$

DFAを$\langle \Sigma, Q, \delta, s, F \rangle$と表記することがある．

適用例(1)：受理するかどうかの判定

入力文字列: $a_1 a_2 \cdots a_n$

1. $i \leftarrow 1$，$q \leftarrow s$
2. $i \leq n$かつ$\delta(q, a_i) \neq \bot$ならば，

   $q \leftarrow \delta(q, a_i)$，$i \leftarrow i + 1$

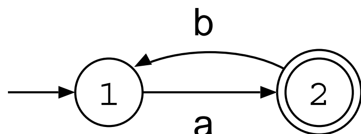
   とし，このステップ2を繰り返す．
3. $i > n$かつ$q \in F$であればyesと表示し，そうでなければnoと表示する．

適用例(2)：文字列の切り出し

入力文字列: $a_1a_2\cdots a_n$

1. $k \leftarrow 0$，$i \leftarrow 0$，$q \leftarrow s$（初期状態）
2. $q \in F$であれば，$k \leftarrow i$
3. $i \leftarrow i+1$
4. $\delta(q, a_i) \neq \perp$ なら，$q \leftarrow \delta(q, a_i)$ とし，ステップ2へ戻る．そうでなければ，k の値を返す.

例：



入力文字列がababbのとき

$$1 \xrightarrow{\texttt{a}} 2_{(k=1)} \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{a}} 2_{(k=3)} \xrightarrow{\texttt{b}} 1 \xrightarrow{\texttt{b}} \perp$$

非決定性有限オートマトン

ε遷移　$\langle q, \varepsilon, q' \rangle$

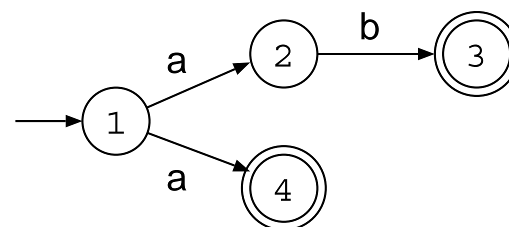
状態qに達したときに，入力文字列と無関係に，
q'へ遷移してよい．qに留まってもよい．

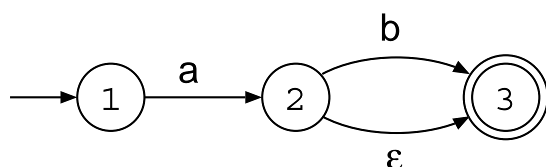
非決定性(nondeterministic)有限オートマトン(NFA)
次のいずれかを満たすFA $\langle \Sigma, Q, \Delta, s, F \rangle$

1. Δがε遷移を含む．
2. ある$q \in Q$と$a \in \Sigma$に対して，$\langle q, a, q' \rangle$の形の状態遷移が，Δの中に複数存在する．

例

M4 $= \langle \Sigma, \{1,2,3\}, \{\langle 1, \mathtt{a}, 2\rangle, \langle 2, \mathtt{b}, 3\rangle, \langle 2, \varepsilon, 3\rangle\}, 1, \{3\}\rangle$
M5 $= \langle \Sigma, \{1,2,3,4\}, \{\langle 1, \mathtt{a}, 2\rangle, \langle 2, \mathtt{b}, 3\rangle, \langle 1, \mathtt{a}, 4\rangle\}, 1, \{3,4\}\rangle$





次の条件をすべて満たす経路

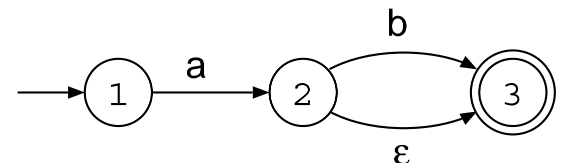
$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \cdots \xrightarrow{x_m} q_m$$

が存在するとき，NFA $\langle \Sigma, Q, \Delta, s, F \rangle$ は，文字列 $\alpha = a_1 a_2 \cdots a_n$ を受理する．

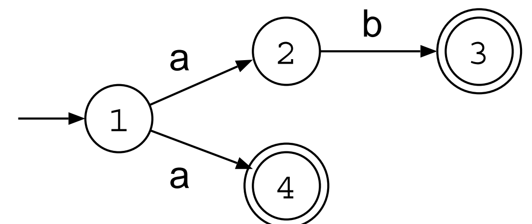
1. $q_0 = s$
2. $\langle q_{i-1}, x_i, q_i \rangle \in \Delta \ (i = 1, 2, \ldots, m)$
3. $x_1, x_2, \ldots, x_m$ から ε を除去すると α と一致する．
4. $q_m \in F$

例:

$L(\mathtt{M4}) = \{\mathtt{a}, \mathtt{ab}\}$
a に対して: $1 \xrightarrow{\mathtt{a}} 2 \xrightarrow{\varepsilon} 3$
ab に対して: $1 \xrightarrow{\mathtt{a}} 2 \xrightarrow{\mathtt{b}} 3$



$L(\mathtt{M5}) = \{\mathtt{a}, \mathtt{ab}\}$
a に対して: $1 \xrightarrow{\mathtt{a}} 4$
ab に対して: $1 \xrightarrow{\mathtt{a}} 2 \xrightarrow{\mathtt{b}} 3$



正規表現からDFAへの変換

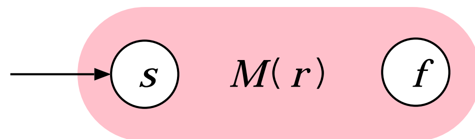
r：正規表現
M：FA
$L(r) = L(M)$のとき，rとMは等価

正規表現rを，等価なDFAに変換

1. rを，等価なNFAに変換
2. NFAを，等価なDFAに変換
3. DFAを，状態数最小の等価なDFAに変換

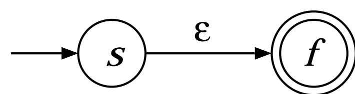
正規表現からNFAへの変換

$M(r)$: 正規表現rに対する標準形NFA

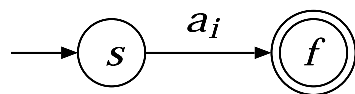


1. 初期状態への遷移は存在しない
2. 受理状態は一つだけ
3. 受理状態からの遷移は存在しない

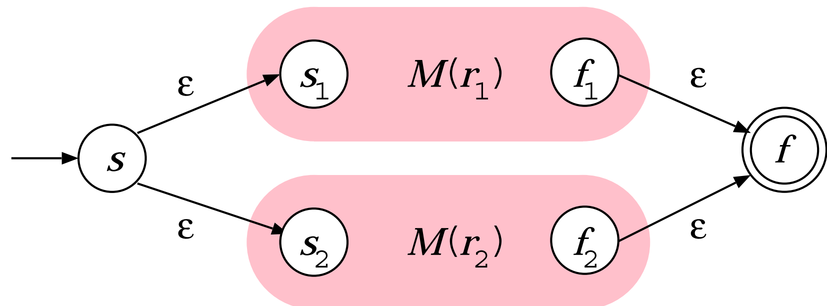
(1) $M(\varepsilon)$



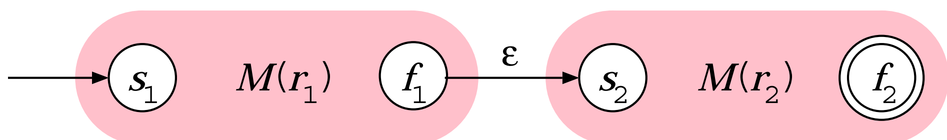
(2) $M(a_i)$



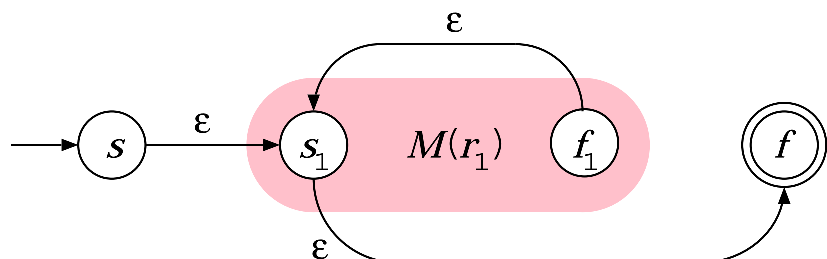
(3) $M(r_1 \,|\, r_2)$



(4) $M(r_1 \cdot r_2)$



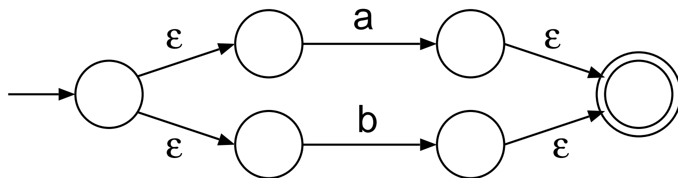
(5) $M(r_1^*)$



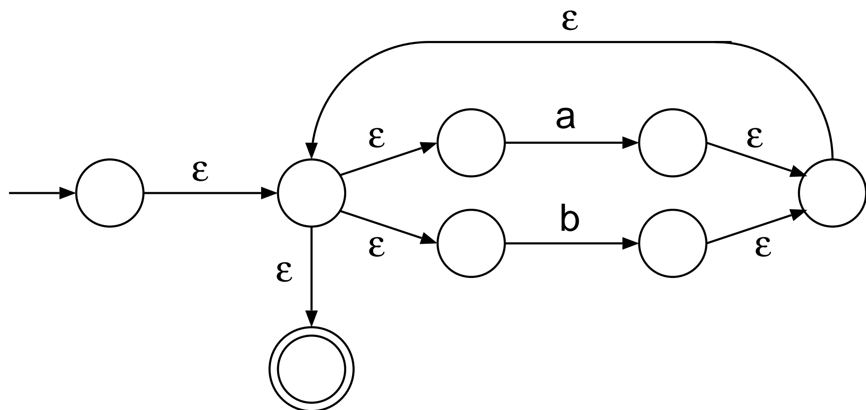
例：$M((\texttt{a}\,|\,\texttt{b})^*\texttt{ab})$

(1) $M(\texttt{a})$ と $M(\texttt{b})$

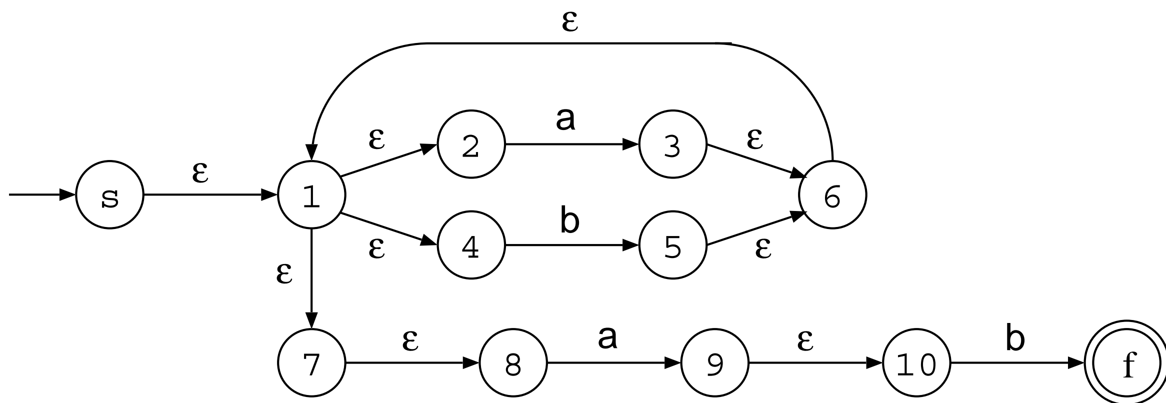
(2) $M(\texttt{a}\,|\,\texttt{b})$



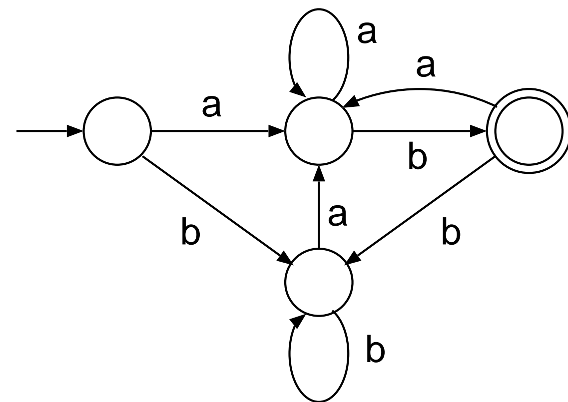
(3) $M((\texttt{a}\,|\,\texttt{b})^*)$



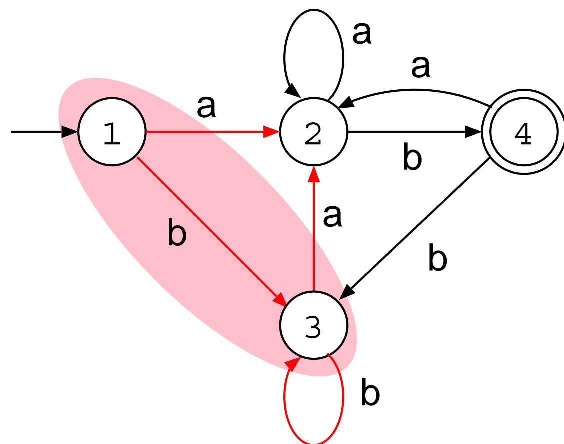
(4) $M((\texttt{a}\,|\,\texttt{b})^*\texttt{ab})$

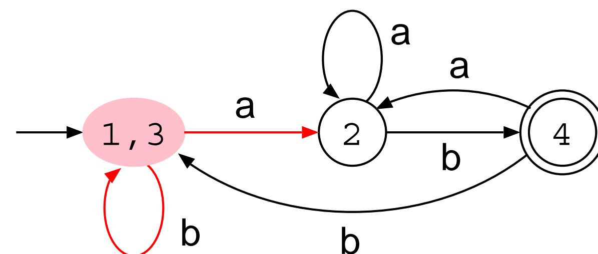


NFAからDFAへの変換



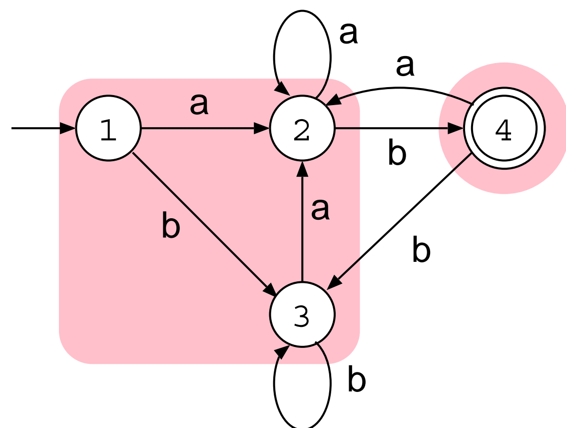
状態数最小のDFA

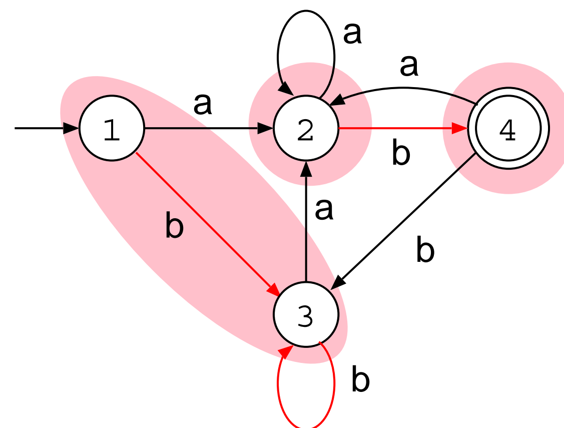




アルゴリズム

1. 受理状態とその他の２グループに分ける．
2. 同一文字に対する遷移先が異なる状態を，別グループに分ける．





字句解析プログラム

正規表現以外の規則

1. 最も長い字句要素を切り出す.
   例: `abc` = `a` + `bc` = `ab` + `c`
2. 字句要素の種類ごとに優先度を設ける.
   例: `if` は識別子ではなく，キーワード

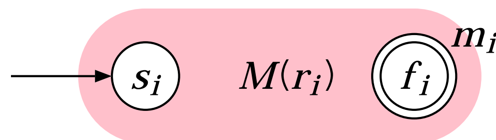
字句構造全体のDFA

$r_1, r_2, \ldots, r_n$: 字句要素の種類ごとの正規表現

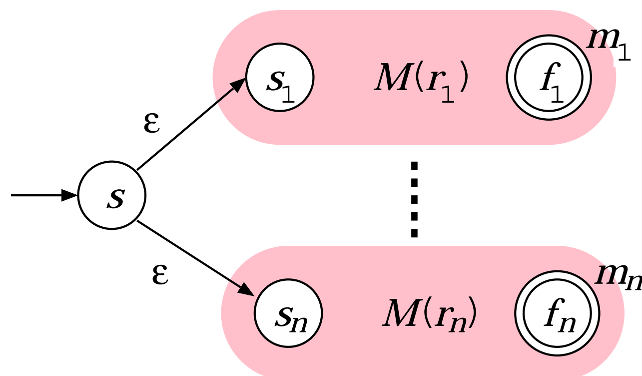
$r_1 \mid r_2 \mid \cdots \mid r_n$ のDFAでは種類がわからない

$m_1, m_2, \ldots, m_n$: 種類ごとのマーカ → 各受理状態に適切なマーカをつける

1. 種類ごとにNFA $M(r_i)$を求め，受理状態にマーカをつける



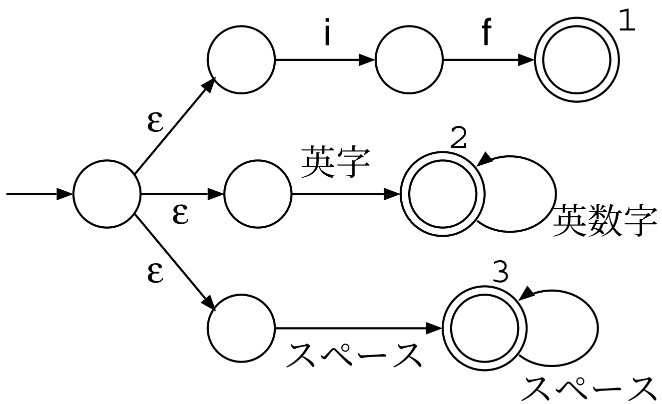
2. 新しい初期状態sを用意し，全体のNFA M_{all}を作成.

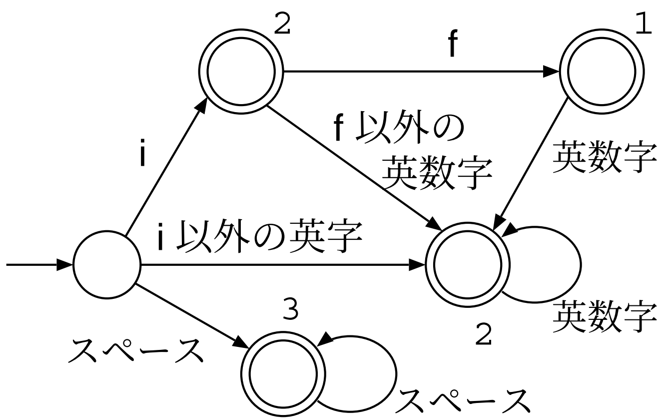


3. M_{all}をDFA $\widehat{M}_{all}$に変換.
   個々の受理状態には，優先度最大のマーカを残す.
4. 状態数最小のDFAを作成.
   ただし，マーカの異なる受理状態は別グループ.

例

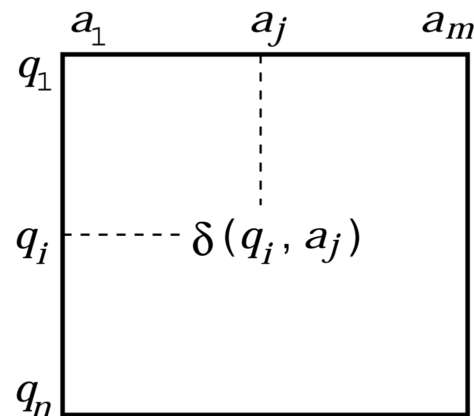
1. キーワード: `if`
2. 識別子: $\langle$英字$\rangle\langle$英数字$\rangle^*$
3. 空白: $\langle$スペース$\rangle^+$





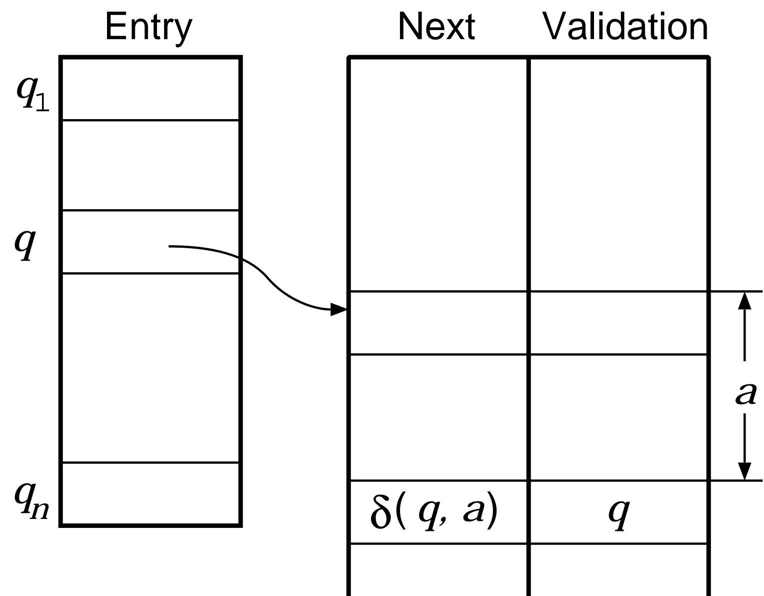
状態遷移表

状態遷移関数 $\delta : Q \times \Sigma \to Q \cup \{\bot\}$ を表にしたもの



ほとんどの要素が $\bot$（未定義）

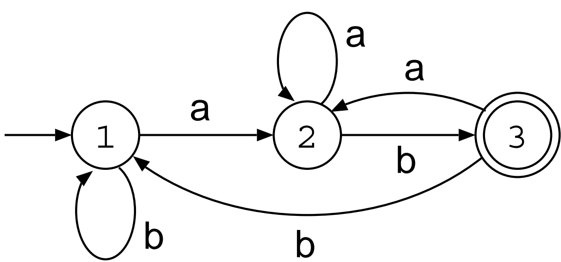
状態遷移表の圧縮



$$\texttt{Validation[Entry[}q\texttt{]+}a\texttt{]} = \begin{cases} q & (\delta(q,a) \neq \perp) \\ q \text{ 以外} & (\delta(q,a) = \perp) \end{cases}$$

```
next_state(q, a) {
  int i = Entry[q] + a;
  if (Validation[i] == q)
    return Next[i];
  else
    return UNDEFINED;
}
```

各ベクタの設定方法



State 1 (0–127):

Index	Value
97	2
98	1

State 2 (2–129):

Index	Value
99	2
100	3

State 3 (4–131):

Index	Value
101	2
102	1

	Next	Validation
0		
2		
4		
97	2	1
98	1	1
99	2	2
100	3	2
101	2	3
102	1	3
131		

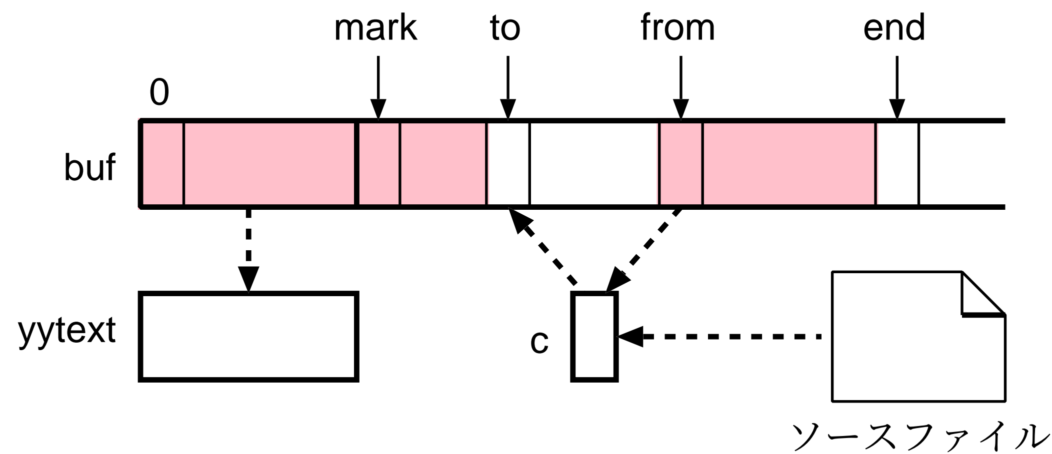
	Entry
1	0
2	2
3	4

ASCIIコードでは，’a’ = 97,’b’ = 98

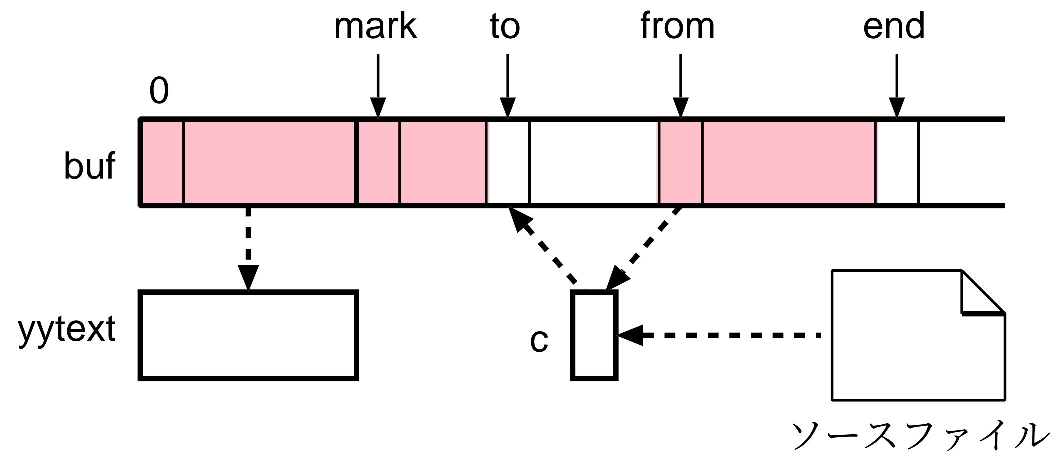
字句要素の切り出し

関数`scan`:

最も長い字句要素を切り出して`yytext`に格納し，マーカの値を返す．入力の終りに達していれば，`ENDOFFILE`を返す．



- `buf[0]`〜`buf[to-1]`:
  これまでに読み込んだ文字列
- `buf[0]`〜`buf[mark-1]`:
  最も長い字句要素
- `buf[from]`〜`buf[end-1]`:
  前回読み込みすぎた文字列



cへの文字設定

```
advance_c() {
  buf[to++] = c;
  if (from < end)
    c = buf[from++];
  else
    c = getc(in);
}
```

バッファの初期化

```
init_buf() {
  mark = to = from = end = 0;
  c = getc(in);
}
```

字句要素の切り出し

```
cutout_lexeme() {
  for (i = 0; i < mark; i++)
    yytext[i] = buf[i];
  yytext[i] = '\0';
  if (mark < to) {
    buf[to++] = c;
    while (from < end)
      buf[to++] = buf[from++];
    c = buf[mark++];
    from = mark;
    end = to;
  }
  mark = to = 0;
}
```

その他のサブ関数

next_state(q,a): 状態遷移関数$\delta(q, a)$の値

final(q): 状態qが受理状態かどうか

marker(q): 受理状態qのマーカ

mark_buf(): toの値をmarkに保存

```
scan() {
  if (c == EOF) return ENDOFFILE;
  q = 1;  qf = UNDEFINED;
  for (;;) {
    if (final(q)) {
      mark_buf();  qf = q;
    }
    if ((x = next_state(q,c)) != UNDEFINED) {
      q = x;  advance_c();
    } else if (qf != UNDEFINED) {
      cutout_lexeme();  return marker(qf);
    } else {
      lexical_error();
      if (c == EOF) return ENDOFFILE;
      q = 1;
    }
  }
}
```

エラーリカバリ（次の空白まで読み飛ばす）

```
lexical_error() {
  while (c != ' ' && c != '\n' && c != EOF)
    advance_c();
  mark_buf();
  cutout_lexeme();
  printf("lexical error: '%s'\n", yytext);
}
```

字句解析の自動化

lex と flex

```
% cat scan.l
%%
[\t ]+ {}
(a|b)*ab { printf("%s: OK.\n", yytext);}
. { printf("%s: wrong.\n", yytext);}
%%
% lex scan.l
% cc lex.yy.c -ll
% a.out
aabbab
aabbab: OK.

aba
ab: OK.
a: wrong.

%
```

```
% cat scan1.l
%%
[\t ]+ {}
"+"|"-"|"*"|"/"|"="
    printf("%s (operator) ", yytext);
";"  printf("%s (separator) ", yytext);
[a-z][a-z0-9]*
    printf("%s (identifier) ", yytext);
[0-9\.]+ {printf("%s (number) ", yytext);}
.   printf("%s (don't know) ", yytext);
%%
% lex scan1.l
% cc lex.yy.c -ll
% a.out
abc=e*3.14+abc/e
abc (identifier) = (operator) e (identifier)
* (operator) 3.14 (number) + (operator)
abc (identifier) / (operator) e (identifier)
```

DFAから正規表現への変換

DFA $M = \langle \Sigma, \{q_1, q_2, \ldots, q_n\}, \delta, q_1, F \rangle$

R_{ij}^k: q_iからq_jへ，$q_1, q_2, \ldots, q_k$だけを通過して遷移させる文字列全体の集合

$$q_i \rightarrow \boxed{q_1 \sim q_k} \rightarrow q_j$$

1. $k > 0$のとき

$$R_{ij}^k = R_{ik}^{k-1}(R_{kk}^{k-1})^* R_{kj}^{k-1} \cup R_{ij}^{k-1}$$

$$q_i \rightarrow \boxed{q_1 \sim q_{k-1}} \rightarrow q_k \rightarrow \cdots \rightarrow q_k \rightarrow \boxed{q_1 \sim q_{k-1}} \rightarrow q_j$$

$$q_i \rightarrow \boxed{q_1 \sim q_{k-1}} \rightarrow q_j$$

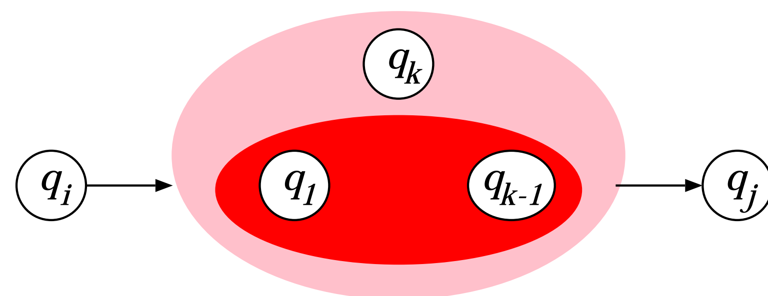
2. $k = 0$のとき

$$R_{ij}^0 = \begin{cases} \{a \,|\, \delta(q_i, a) = q_j\} & (i \neq j \text{のとき}) \\ \{a \,|\, \delta(q_i, a) = q_j\} \cup \{\varepsilon\} & (i = j \text{のとき}) \end{cases}$$

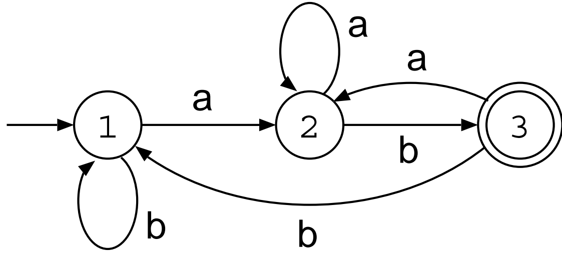
n: Mの状態数

$$L(M) = \bigcup_{q \in F} R_{sq}^n$$

この集合式を正規表現に変換する



例



$$R^3_{13} = R^2_{13}(R^2_{33})^* R^2_{33} \cup R^2_{13} = R^2_{13}(R^2_{33})^+ \cup R^2_{13} = R^2_{13}(R^2_{33})^*$$

$$R^2_{13} = R^1_{12}(R^1_{22})^* R^1_{23} \cup R^1_{13}$$

$$R^2_{33} = R^1_{32}(R^1_{22})^* R^1_{23} \cup R^1_{33}$$

$$R^1_{12} = R^0_{11}(R^0_{11})^* R^0_{12} \cup R^0_{12} = (R^0_{11})^+ R^0_{12} \cup R^0_{12}$$

$$= (R^0_{11})^* R^0_{12} = \{\mathtt{b}, \varepsilon\}^* \{\mathtt{a}\} = \{\mathtt{b}\}^* \{\mathtt{a}\}$$

$$R^1_{22} = R^0_{21}(R^0_{11})^* R^0_{12} \cup R^0_{22} = \phi \cup \{\mathtt{a}, \varepsilon\} = \{\mathtt{a}, \varepsilon\}$$

$$R^1_{23} = R^0_{21}(R^0_{11})^* R^0_{13} \cup R^0_{23} = \phi \cup \{\mathtt{b}\} = \{\mathtt{b}\}$$

$$R^1_{13} = R^0_{11}(R^0_{11})^* R^0_{13} \cup R^0_{13} = \phi$$

$$R^1_{32} = R^0_{31}(R^0_{11})^* R^0_{12} \cup R^0_{32} = \{\mathtt{b}\}\{\mathtt{b}\}^* \{\mathtt{a}\} \cup \{\mathtt{a}\} = \{\mathtt{b}\}^* \{\mathtt{a}\}$$

$$R^1_{33} = R^0_{31}(R^0_{11})^* R^0_{13} \cup R^0_{33} = \{\mathtt{b}\}\{\mathtt{b}\}^* \phi \cup \{\varepsilon\} = \{\varepsilon\}$$

$$R^3_{13} = R^2_{13}(R^2_{33})^*$$

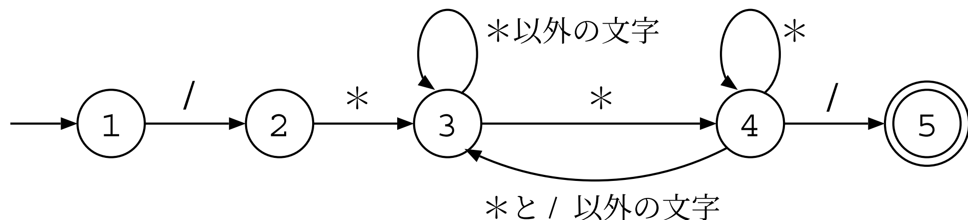
$$= (\{\mathtt{b}\}^* \{\mathtt{a}\}\{\mathtt{a}, \varepsilon\}^* \{\mathtt{b}\})(\{\mathtt{b}\}^* \{\mathtt{a}\}\{\mathtt{a}, \varepsilon\}^* \{\mathtt{b}\})^*$$

$$= (\{\mathtt{b}\}^* \{\mathtt{a}\}^+ \{\mathtt{b}\})(\{\mathtt{b}\}^* \{\mathtt{a}\}^+ \{\mathtt{b}\})^*$$

$$= (\{\mathtt{b}\}^* \{\mathtt{a}\}^+ \{\mathtt{b}\})^+$$

DFAと等価な正規表現は，$(\mathtt{b}^* \mathtt{a}^+ \mathtt{b})^+$

例：コメントの正規表現



$\mathtt{/*}\langle \mathtt{*}$以外の文字$\rangle^* \mathtt{*}$
　$(\langle \mathtt{*}$と$\mathtt{/}$以外の文字$\rangle\langle \mathtt{*}$以外の文字$\rangle^* \mathtt{*} \mid \mathtt{*})^* \mathtt{/}$

状態番号の3と4を入れ替えると

$\mathtt{/*}\,(\langle \mathtt{*}$以外の文字$\rangle \mid (\mathtt{*})^+\langle \mathtt{*}$と$\mathtt{/}$以外の文字$\rangle)^*\,(\mathtt{*})^*\,\mathtt{*/}$

正規表現の限界

正規表現では表せない文字列集合の例

$$X = \{a^i c b^i \mid i \geq 0\}$$

正規表現rで表せたと仮定する

n: $M(r)$の状態数

$a^n c b^n$ $(\in X)$を受理する$M(r)$の経路

$$q_0 \underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{n} q_n \xrightarrow{c} q_{n+1} \underbrace{\xrightarrow{b} \cdots \xrightarrow{b}}_{n} q_{2n+1}$$

$q_i = q_j$ なる i, j $(0 \leq i < j \leq n)$が存在する

$$q_0 \underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{i} q_i \underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{j-i} q_j \underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{n-j} q_n \xrightarrow{c} q_{n+1} \underbrace{\xrightarrow{b} \cdots \xrightarrow{b}}_{n} q_{2n+1}$$

q_iからq_jへの経路をカットすると

$$q_0 \underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{i} q_i (= q_j) \underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{n-j} q_n \xrightarrow{c} q_{n+1} \underbrace{\xrightarrow{b} \cdots \xrightarrow{b}}_{n} q_{2n+1}$$

$M(r)$は$a^{n-(j-i)} c b^n$ $(\notin X)$も受理する$\rightarrow$ 矛盾

文脈自由文法なら

$$A \rightarrow aAb \mid c$$

第３章

文　　法

構文，制約，意味

構文(syntax)：プログラムの構造を定義
制約(constraint)：構文以外の規則
意味(semantics)：実行を定義

例：if文

構文：`if` （ $expr$ ） $stat_1$ `else` $stat_2$
制約：$expr$は真偽値を表す型
意味：$expr$を計算し，その値が真なら $stat_1$ を，
偽なら $stat_2$ を実行する

文法：構文＋制約
構文解析は「文法解析」ではない．

構文解析の話をするときは，制約には言及しない．
→ 構文＝文法

構文の記法

ALGOL 60のBNF

⟨program⟩ `::=`
  `program` ⟨identifier⟩ `(` ⟨identifier-list⟩ `)` `;` ⟨block⟩ `.`
  | `program` ⟨identifier⟩ `;` ⟨block⟩ `.`

⟨identifier⟩は非終端記号(nonterminal symbol)あるいは構文変数(syntax variable)

`program`は，終端記号(terminal symbol)

⟨identifier-list⟩ `::=` ⟨identifier⟩ | ⟨identifier-list⟩ `,` ⟨identifier⟩

識別子リストとは:

1. 識別子であるか，
2. 識別子リストと識別子をカンマで区切ったもの

例：“`x, y, z`”

1. `x`は識別子なので識別子リスト
2. `x`が識別子リストなので“`x, y`”も識別子リスト
3. “`x, y`”が識別子リストなので，“`x, y, z`”は識別子リスト

生成規則(production)

`::=`と縦棒は，メタ記号(meta-symbol)

拡張BNFの例：

$\langle$identifier-list$\rangle$ `::=` { $\langle$identifier$\rangle$ `,` } $\langle$identifier$\rangle$

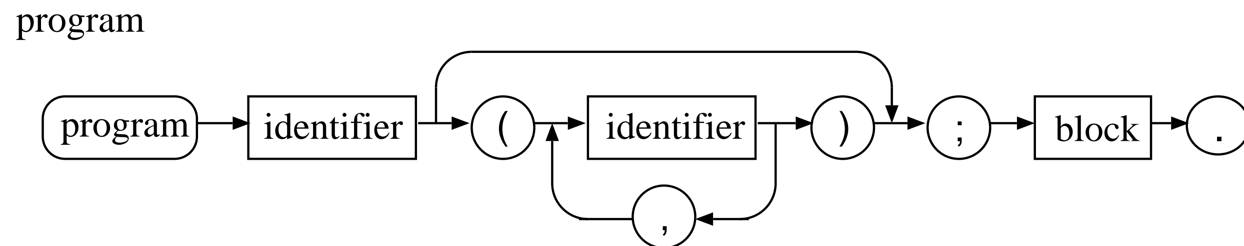
C言語の構文記法：

identifier-list:
  identifier
  identifier-list , *identifier*

`for` ($expr_{opt}$; $expr_{opt}$; $expr_{opt}$) $stat$

Pascalの構文図式(syntax diagram)の例：

program



文法

生成規則：$A \to \alpha$　(Aは記号，αは記号列)

ε規則：$A \to \varepsilon$

文法：組$\langle P, S \rangle$

　P：生成規則の集合

　S：出発記号(start symbol)

語彙(vocabulary) V：Pに現れる記号全体の集合

非終端記号V_N：生成規則の左辺に現れる記号

終端記号V_T：非終端記号以外の記号

$$V = V_N \cup V_T$$
$$V_N \cap V_T = \phi$$
$$S \in V_N$$

例：文法G1 $= \langle \mathrm{P1}, E \rangle = \langle V_N, V_T, \mathrm{P1}, E \rangle$

$$\begin{aligned} \mathrm{P1} = \{\ & E \to E + T \\ & E \to T \\ & T \to T * F \\ & T \to F \\ & F \to (E) \\ & F \to i\ \} \end{aligned}$$

$$V_N = \{E, F, T\}$$
$$V_T = \{+, *, (,), i\}$$
$$\begin{aligned} V &= V_N \cup V_T \\ &= \{E, F, T, +, *, (,), i\} \end{aligned}$$

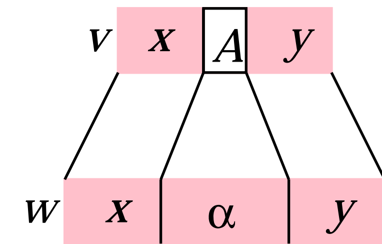
$$\begin{aligned} \mathrm{P1} = \{\ & E \to E + T \mid T \\ & T \to T * F \mid F \\ & F \to (E) \mid i\ \} \end{aligned}$$

$v = xAy$, $w = x\alpha y$,

$A \to \alpha \in P$のとき，$v \Rightarrow w$

vはwを直接生成する．

wはvに直接還元される．

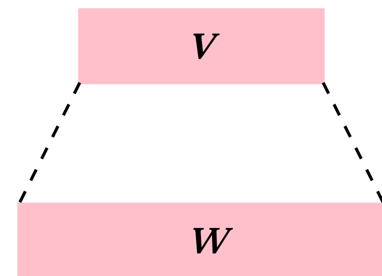


$v \Rightarrow u_1 \Rightarrow u_2 \cdots \Rightarrow u_n \Rightarrow w$ のとき，$v \overset{+}{\Rightarrow} w$

$v \overset{+}{\Rightarrow} w$または$v = w$のとき，$v \overset{*}{\Rightarrow} w$

vはwを生成(produce)する．

wはvに還元(reduce)される．



$S \overset{*}{\Rightarrow} x$のとき，$x$は$G$の文形式(sentential form)

特にxが終端記号列ならxはGの文(sentence)

$L(G)$：Gの文全体の集合，Gの定義する言語

例：$\mathrm{P1} = \{\, E \to E+T \mid T$

$T \to T*F \mid F$

$F \to (\, E\,) \mid i \,\}$

$E \Rightarrow E+T \Rightarrow T+T \Rightarrow T+T*F \Rightarrow F+T*F$

$\Rightarrow F+T*i \Rightarrow F+F*i \Rightarrow F+i*i \Rightarrow i+i*i$

$i+i*i$だけが文

導出(derivation)：

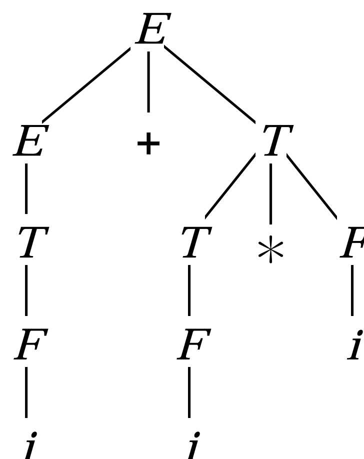
vから$v \stackrel{*}{\Rightarrow} w$である記号列$w$を求めること

最左(left-most)導出

$$E \underset{\text{lm}}{\Longrightarrow} E+T \underset{\text{lm}}{\Longrightarrow} T+T \underset{\text{lm}}{\Longrightarrow} F+T \underset{\text{lm}}{\Longrightarrow} i+T \underset{\text{lm}}{\Longrightarrow} i+T*F$$
$$\underset{\text{lm}}{\Longrightarrow} i+F*F \underset{\text{lm}}{\Longrightarrow} i+i*F \underset{\text{lm}}{\Longrightarrow} i+i*i$$

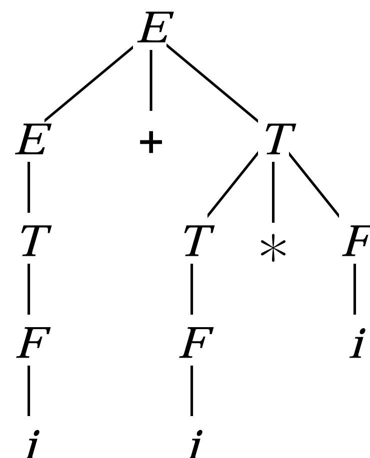
最右(right-most)導出

$$E \underset{\text{rm}}{\Longrightarrow} E+T \underset{\text{rm}}{\Longrightarrow} E+T*F \underset{\text{rm}}{\Longrightarrow} E+T*i \underset{\text{rm}}{\Longrightarrow} E+F*i$$
$$\underset{\text{rm}}{\Longrightarrow} E+i*i \underset{\text{rm}}{\Longrightarrow} T+i*i \underset{\text{rm}}{\Longrightarrow} F+i*i \underset{\text{rm}}{\Longrightarrow} i+i*i$$

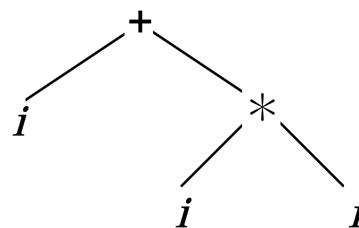


解析木とあいまい性

解析木



構文木

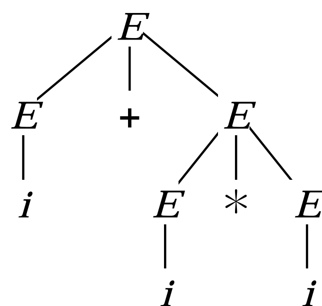


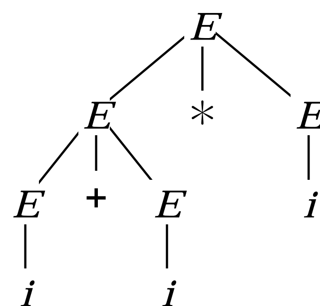
あいまいな文：二つの異なる解析木が考えられる文

あいまいな文法：あいまいな文を生成する文法

例：文法 $\mathrm{G2} = \langle \mathrm{P2}, E \rangle$，$\mathrm{P2} = \{ E \rightarrow E + E \mid E * E \mid (E) \mid i \}$

文 $i + i * i$ に対する二つの解析木





あいまいな英文

Time flies.

「主語＋動詞」→「光陰矢の如し」
「動詞＋目的語」→「蝿の速度計測をしろ」

あいまいな日本語文

ここではきものをぬぎなさい

「ここで履物を脱ぎなさい」
or「ここでは着物を脱ぎなさい」?

if文の構文

stat:

`if` (*expr*) *stat*

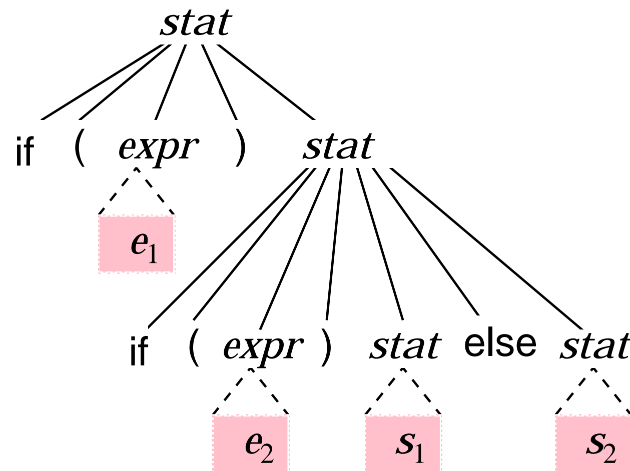
`if` (*expr*) *stat* `else` *stat*

other

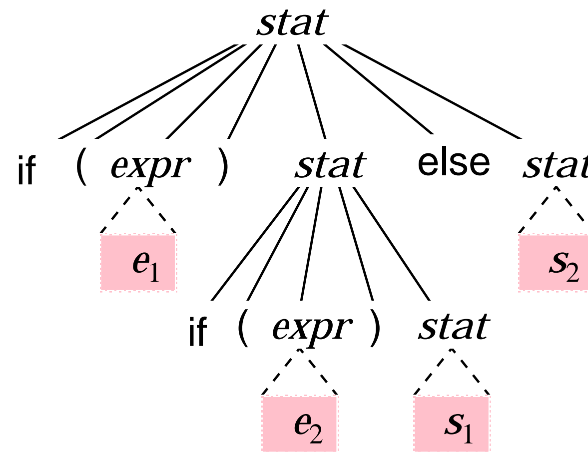
あいまいなif文

`if` (e_1) `if` (e_2) s_1 `else` s_2

`if` (e_1) { `if` (e_2) s_1 `else` s_2 }



`if` (e_1) { `if` (e_2) s_1 } `else` s_2



C言語の場合は左側

if文のあいまい性除去

開いた（open）文：直後に「`else`＋文」をつけても
　文になり得る文
閉じた（closed）文：open でない文

stat:
　closed-stat
　open-stat
closed-stat:
　`if` (*expr*) *closed-stat* `else` *closed-stat*
　other
open-stat:
　`if` (*expr*) *stat*
　`if` (*expr*) *closed-stat* `else` *open-stat*

演算子の優先順位と結合性

演算子の優先順位

$$x + y * z = x + (y * z)$$

演算子の結合性

$$x - y - z = (x - y) - z$$

なので左結合的(left associative)

$$x = y = z = x = (y = z)$$

なので右結合的

優先順位と結合性

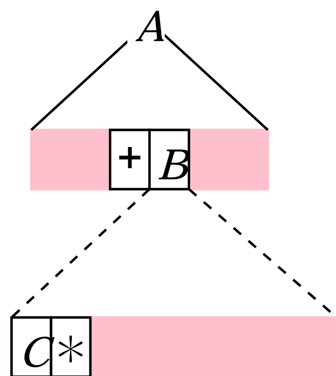
$$x + y - z = (x + y) - z$$

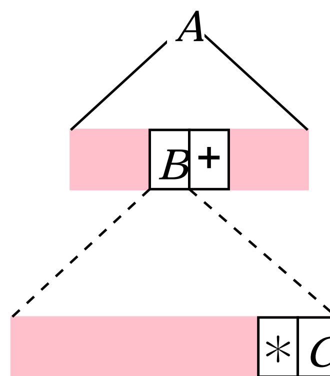
$op_1 \lessdot op_2$：$\cdots op_1\ expr\ op_2 \cdots$のとき$op_2$を先に計算

1. 生成規則$A \rightarrow \cdots op_1\, B \cdots$が存在
2. ある非終端記号Cに対して$B \stackrel{+}{\Rightarrow} C\, op_2 \cdots$

$op_1 \gtrdot op_2$：$\cdots op_1\ expr\ op_2 \cdots$のとき$op_1$を先に計算

1. 生成規則$A \rightarrow \cdots B\, op_2 \cdots$が存在
2. ある非終端記号Cに対して$B \stackrel{+}{\Rightarrow} \cdots op_1\, C$





例：文法G1

1. $E \rightarrow E+T$が存在し，

2. $E \Rightarrow E+T$

$\rightarrow$ $+ \gtrdot +$

1. $E \rightarrow E+T$が存在し

2. $T \Rightarrow T*F$

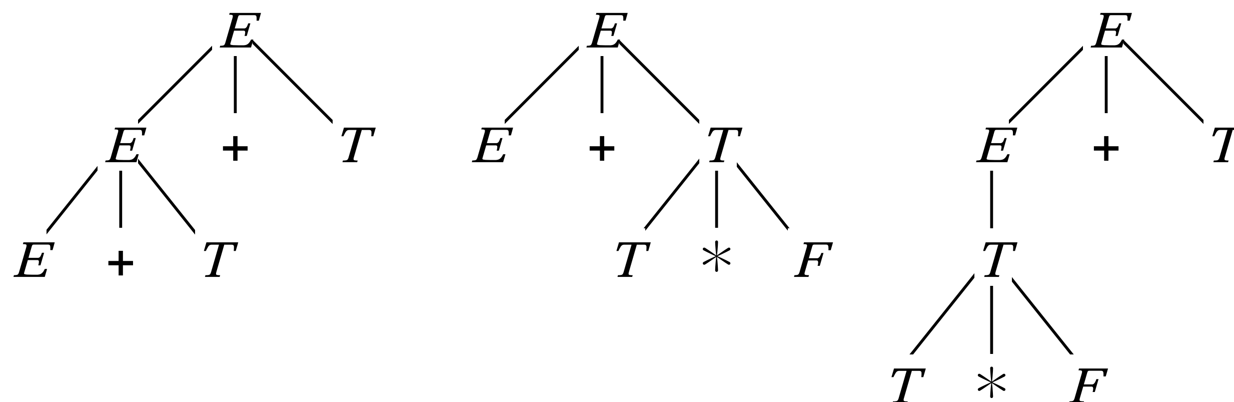
$\rightarrow$ $+ \lessdot *$

1. $E \rightarrow E+T$が存在し，

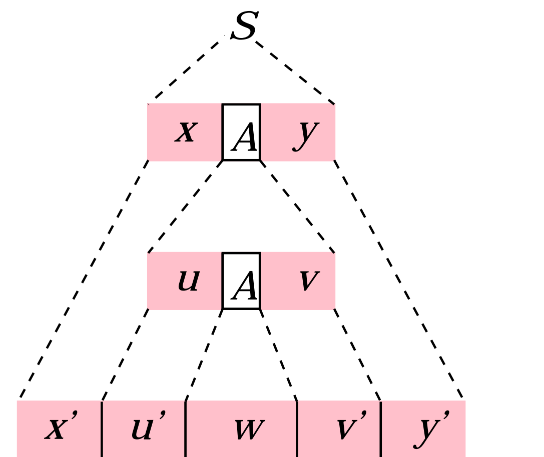
2. $E \Rightarrow T \Rightarrow T*F$

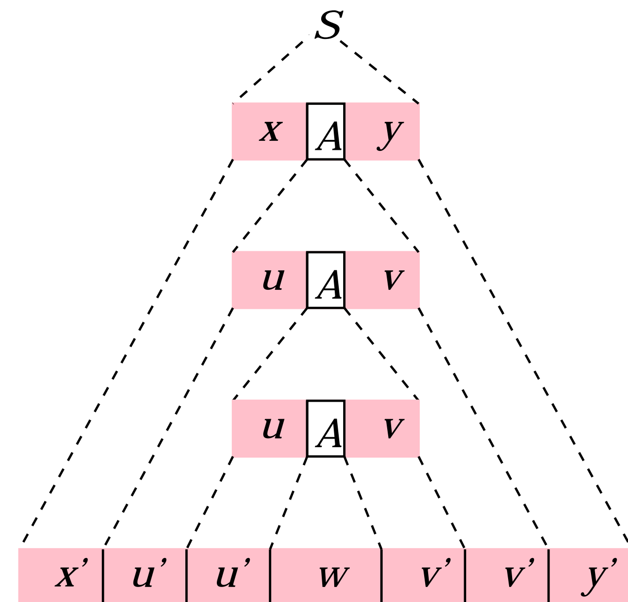
$\rightarrow$ $* \gtrdot +$

$+ \lessdot *$かつ$* \gtrdot +$なので，$*$は$+$より優先順位が高い．



文脈自由文法とその限界





Aが文形式に現れていれば，Aの前後（文脈）に関係なく，Aを左辺とする生成規則を適用可能

文脈自由文法で表現できない記号列集合

$$Y = \{a^i b^i c^i \mid i \geq 1\} = \{abc, aabbcc, aaabbbccc, \ldots\}$$

$L(G) = Y$なる文脈自由文法Gが存在したと仮定

次の$a^j b^j c^j$が存在する．

$$S \overset{*}{\Rightarrow} xAy \overset{+}{\Rightarrow} xuAvy \overset{+}{\Rightarrow} a^j b^j c^j$$

Gが生成する記号列

$$S \overset{*}{\Rightarrow} xAy \overset{+}{\Rightarrow} xuAvy \overset{+}{\Rightarrow} xuuAvvy \overset{+}{\Rightarrow} x'u'u'wv'v'y'$$

は，Yの要素でない．→矛盾

Y を生成する文脈自由でない文法

$$
\begin{aligned}
S &\rightarrow aB\,b\,c \\
B &\rightarrow aB\,b\,C \mid \varepsilon \\
Cb &\rightarrow b\,C \\
Cc &\rightarrow c\,c
\end{aligned}
$$

$$
\begin{aligned}
S &\Rightarrow aBbc \Rightarrow aaBbCbc \Rightarrow aaaBbCbCbc \\
&\Rightarrow aaabCbCbc \Rightarrow aaabCbbCc \Rightarrow aaabCbbcc \\
&\Rightarrow aaabbCbcc \Rightarrow aaabbbCcc \Rightarrow aaabbbccc
\end{aligned}
$$

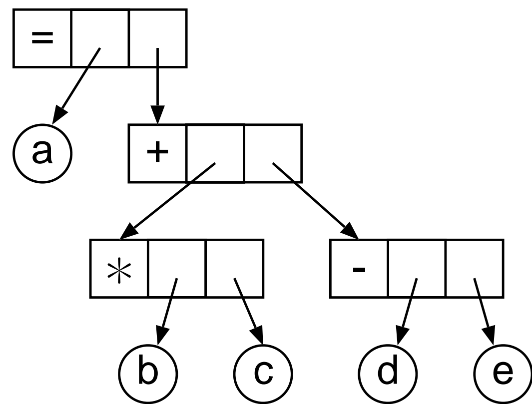
第4章

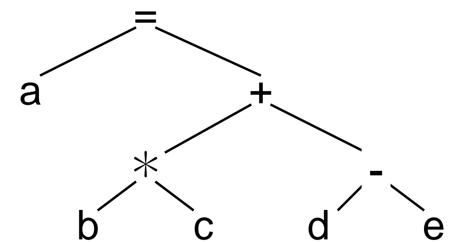
構 文 解 析

構文木とその表現

N組 (N-tuple)

a = b * c + (d - e)





tuple：N組へのポインタの型

token：トークンへのポインタの型

tuple型は，token型を含む.

N組の生成

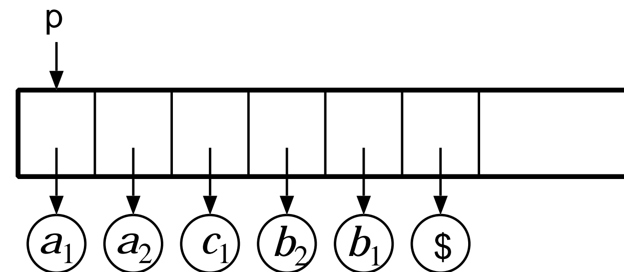
```
x1 = make_tuple("*", b, c);
x2 = make_tuple("-", d, e);
x3 = make_tuple("+", x1, x2);
x4 = make_tuple("=", a, x3);
```

トークンの種類判定

```
int is_a(x)
```

再帰的下向き (recursive descent) 構文解析法

解析関数 (parsing function)



$A \to aAb \mid c$

```
tuple parse_A() {
  token *prev = p, x, z; tuple y;
  if (is_a(x = *p++) && (y = parse_A())
                     && is_b(z = *p++))
    return make_tuple("A", x, y, z);
  p = prev;
  if (is_c(x = *p++)) return x;
  p = prev;
  return NULL;
}
```

$は特殊な終端記号と見なす.

生成規則は再帰的

→ 再帰的な解析関数群

入力トークン列全体の解析

```
tuple parse() {
  tuple x = parse_A();
  if (x && is_eof(*p))
    return x;
  else
    syntax_error();
}
```

再帰的下向き構文解析法の問題点

左再帰 (left recursion)

バックトラック (backtracking)

左再帰

文法が左再帰を含む$\rightarrow$ 解析関数の再帰呼出しが止らない.

G1の生成規則：

$$E \rightarrow E \texttt{+} T \mid T$$

直接の左再帰 (immediate left recursion)

```
tuple parse_E() {
  token *prev = p; tuple x, y;
  if ((x = parse_E()) && is_plus(*p++)
                      && (y = parse_T()))
    return make_tuple("+", x, y);
  p = prev;
  if (x = parse_T()) return x;
  p = prev;
  return NULL;
}
```

間接的な左再帰

$$A \rightarrow Ba \mid c$$
$$B \rightarrow Ab \mid d$$

C 言語における左再帰

identifier-list:
  identifier
  identifier-list , *identifier*

右再帰に置き換え可能.

identifier-list:
  identifier
  identifier , *identifier-list*

単純に右再帰に置き換えられない例

additive-expression:
  multiplicative-expression
  additive-expression + *multiplicative-expression*
  additive-expression - *multiplicative-expression*

左結合的が右結合的になってしまう.

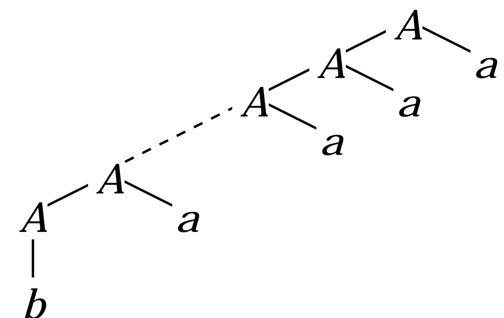
`a - b + c` $=$ `(a - b) + c` $\neq$ `a - (b + c)` $=$ `a - b - c`

C 言語の文法には，間接的な左再帰は存在しない.

「直接の左再帰」の除去

$A \to Aa \mid b$

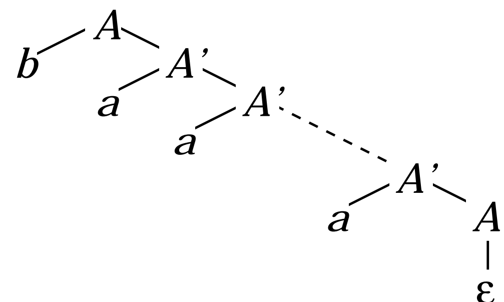
$$A \underset{\mathrm{lm}}{\Longrightarrow} Aa \underset{\mathrm{lm}}{\Longrightarrow} Aaa \underset{\mathrm{lm}}{\Longrightarrow} Aaaa \underset{\mathrm{lm}}{\Longrightarrow} \cdots \underset{\mathrm{lm}}{\Longrightarrow} Aa^n \underset{\mathrm{lm}}{\Longrightarrow} ba^n$$



$A \to bA'$

$A' \to aA' \mid \varepsilon$

$$A \underset{\mathrm{rm}}{\Longrightarrow} bA' \underset{\mathrm{rm}}{\Longrightarrow} baA' \underset{\mathrm{rm}}{\Longrightarrow} baaA' \underset{\mathrm{rm}}{\Longrightarrow} \cdots \underset{\mathrm{rm}}{\Longrightarrow} ba^nA' \underset{\mathrm{rm}}{\Longrightarrow} ba^n$$




```
tuple parse_A() {
  token *prev = p, y;
  if (is_b(y = *p++)) return parse_A1(y);
  p = prev;
  return NULL;
}
tuple parse_A1(tuple x) {
  token *prev = p, y;
  if (is_a(y = *p++))
    return parse_A1(make_tuple("A", x, y));
  p = prev;
  return x;
}
```

一般の場合

$$A \to A\alpha_1 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \cdots \mid \beta_m$$

ここで

$\beta_j \in V^*$，β_jはAで始まらない.

$\alpha_i \in V^+$

$$A \to \beta_1 A' \mid \cdots \mid \beta_m A'$$
$$A' \to \alpha_1 A' \mid \cdots \mid \alpha_n A' \mid \varepsilon$$

例：G1の生成規則

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid i$$

から左再帰を除去した文法G3

$$E \to TE'$$
$$E' \to +TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \varepsilon$$
$$F \to (E) \mid i$$

バックトラック

複数の選択肢があるとき，その一つを試してみて，
失敗だったらもとの状態に戻してやり直す．

例

$A \to aBc$

$B \to b \mid bd$

```
tuple parse_A() {
  token *prev = p, x, z; tuple y;
  if (is_a(x = *p++) && (y = parse_B())
                     && is_c(z = *p++))
    return make_tuple("A", x, y, z);
  p = prev;
  return NULL;
}
tuple parse_B() {
  token *prev = p, x, y;
  if (is_b(x = *p++)) return x;
  p = prev;
  if (is_b(x = *p++) && is_d(y = *p++))
    return make_tuple("B", x, y);
  p = prev;
  return NULL;
}
```

正しく動作しない（例：$abdc$）→ 一般的には複雑な機構が必要

くくり出し (factoring) によるバックトラックの回避

数学では「$ax + ay$ の a をくくり出して $a(x + y)$」

$B \to b \mid bd$ の b をくくり出す.

$B \to b\,(d \mid \varepsilon)$

```
tuple parse_B() {
  token *prev = p, x;
  if (is_b(x = *p++)) {
    token *prev1 = p, y;
    if (is_d(y = *p++))
      return make_tuple("B", x, y);
    p = prev1;
    return x;
  }
  p = prev;
  return NULL;
}
```

生成規則が終端記号で始まらない場合は適用が困難.

$A \to aB\,c \quad B \to C \mid D \quad C \to \cdots \quad D \to \cdots$

バックトラックのもう一つの大きな問題：

適切なエラーメッセージ出力が難しい

LL(k)構文解析法

k 個のトークンを先読みし，どの生成規則を適用するかを決定する

$B \to C \mid D$ の解析関数で，

次の入力トークン x を先読みし，

1. もし $C \overset{+}{\Rightarrow} x \cdots$ なら `parse_C` を呼び出す.
2. もし $D \overset{+}{\Rightarrow} x \cdots$ なら `parse_D` を呼び出す.
3. どちらでもなければ構文エラー

LL(1) 文法

文法 $G = \langle V_N, V_T, P, S \rangle$

$\mathrm{First}(\alpha)$：記号列 α の first 集合

$$\mathrm{First}(\alpha) = \{a \mid a \in V_T,\ \alpha \stackrel{*}{\Rightarrow} a \cdots\}$$

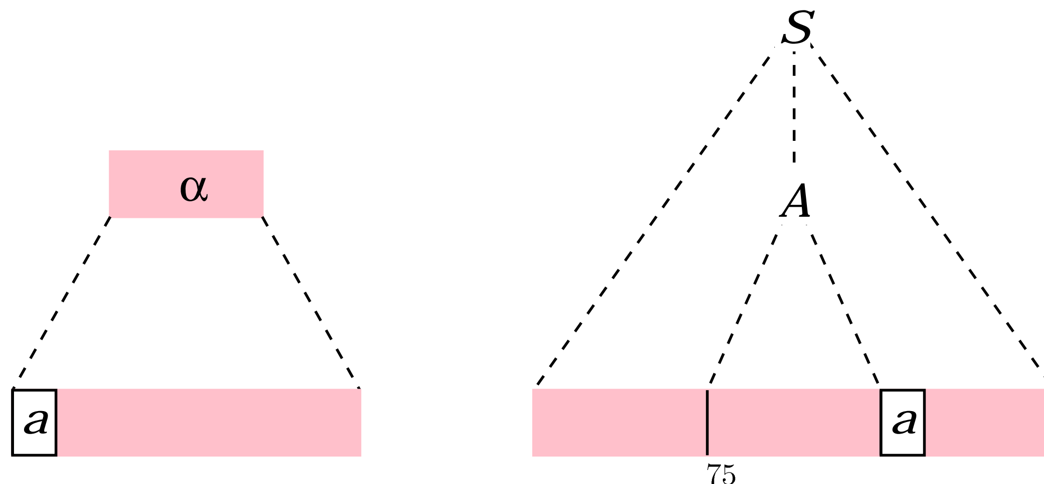
$\mathrm{Follow}(A)$：非終端記号 A の follow 集合

$$\mathrm{Follow}(A) = \{a \mid a \in V_T,\ S\,\$ \stackrel{*}{\Rightarrow} \cdots Aa \cdots\}$$

$\$ \in \mathrm{Follow}(A)$ なら，

$$S \stackrel{*}{\Rightarrow} \cdots A$$

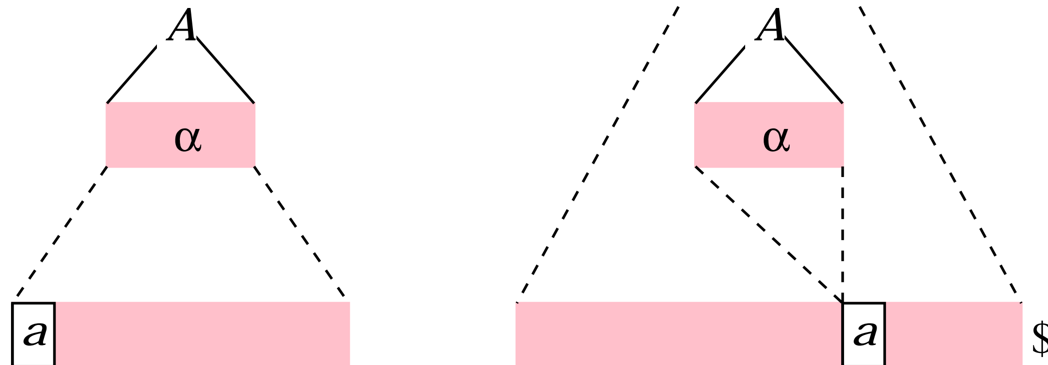
常に，$\$ \in \mathrm{Follow}(S)$



$\mathrm{Director}(A,\alpha)$：生成規則 $A \rightarrow \alpha$ の director 集合

$$\mathrm{Director}(A,\alpha) = \begin{cases} \mathrm{First}(\alpha) & (\alpha \overset{*}{\Rightarrow} \varepsilon \text{でない}) \\ \mathrm{First}(\alpha) \cup \mathrm{Follow}(A) & (\alpha \overset{*}{\Rightarrow} \varepsilon \text{のとき}) \end{cases}$$

$A \rightarrow \alpha$ で還元するときに，入力トークン列の先頭に現れる可能性のある終端記号の全体



LL(1) 文法：右辺だけが異なる任意の生成規則

$A \rightarrow \alpha$ と

$A \rightarrow \beta$ に対して

必ず

$\mathrm{Director}(A,\alpha) \cap \mathrm{Director}(A,\beta) = \phi$

が成り立つ文法

LL(1) 文法のための計算

記号列αがεを生成する（$\alpha \overset{*}{\Rightarrow} \varepsilon$）かどうかの判定

$\alpha = \varepsilon$なら$\alpha \overset{*}{\Rightarrow} \varepsilon$

$\alpha = a_1 a_2 \cdots a_n \neq \varepsilon$なら$\alpha \overset{*}{\Rightarrow} \varepsilon$は，

すべてのa_iに対して$a_i \overset{*}{\Rightarrow} \varepsilon$

と同値

Null(a)：記号aに対して$a \overset{*}{\Rightarrow} \varepsilon$かどうか

$$\mathrm{Null}(a) = \begin{cases} \mathrm{true} & (a \overset{*}{\Rightarrow} \varepsilon \text{のとき}) \\ \mathrm{false} & (a \overset{*}{\Rightarrow} \varepsilon \text{でないとき}) \end{cases}$$

1. すべての記号aに対して$\mathrm{Null}(a) \leftarrow \mathrm{false}$
2. ε規則$A \rightarrow \varepsilon$に対して，$\mathrm{Null}(A) \leftarrow \mathrm{true}$
3. ε規則以外の生成規則$A \rightarrow a_1 \cdots a_n$に対して，
   すべてのa_iについて$\mathrm{Null}(a_i) = \mathrm{true}$ならば，
   $\mathrm{Null}(A) \leftarrow \mathrm{true}$
4. 変更がなくなるまで，ステップ3を繰り返す．

例：G1から左再帰を除去した文法G3

$$
\begin{aligned}
&E \to TE' \\
&E' \to \texttt{+}TE' \mid \varepsilon \\
&T \to FT' \\
&T' \to \texttt{*}FT' \mid \varepsilon \\
&F \to \texttt{(}\,E\,\texttt{)} \mid i
\end{aligned}
$$

$$\mathrm{Null}(E') = \mathrm{Null}(T') = \mathrm{true}$$

$$\mathrm{Null}(E) = \mathrm{Null}(T) = \mathrm{Null}(F) = \mathrm{false}$$

first 集合の計算

$\alpha = \varepsilon$ なら $\mathrm{First}(\alpha) = \phi$

$\alpha = a_1 a_2 \cdots a_n \neq \varepsilon$ に対しては，

$$\mathrm{First}(\alpha) = \begin{cases} \mathrm{First}(a_1) & (a_1 \overset{*}{\Rightarrow} \varepsilon \text{でない}) \\ \mathrm{First}(a_1) \cup \mathrm{First}(a_2 \cdots a_n) & (a_1 \overset{*}{\Rightarrow} \varepsilon \text{のとき}) \end{cases}$$

$\rightarrow$ 記号についての first 集合から求められる．

1. 終端記号 x に対して $\mathrm{First}(x) \leftarrow \{x\}$
   非終端記号 A に対して，$\mathrm{First}(A) \leftarrow \phi$
2. 生成規則 $A \rightarrow \alpha_1 a \alpha_2$ $(a \in (V_N \cup V_T))$ に対して，
   $\alpha_1 \overset{*}{\Rightarrow} \varepsilon$ なら $\mathrm{First}(A) \leftarrow \mathrm{First}(A) \cup \mathrm{First}(a)$
3. 変更がなくなるまで，ステップ 2 を繰り返す．

例：文法G3のfirst集合

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid i$$

生成規則$F \rightarrow (E)$と$F \rightarrow i$から

$$\mathrm{First}(F) = \{(, i\}$$

生成規則$T' \rightarrow *FT'$から

$$\mathrm{First}(T') = \{*\}$$

$T \rightarrow FT'$から（$\mathrm{Null}(F) = \mathrm{false}$なので）

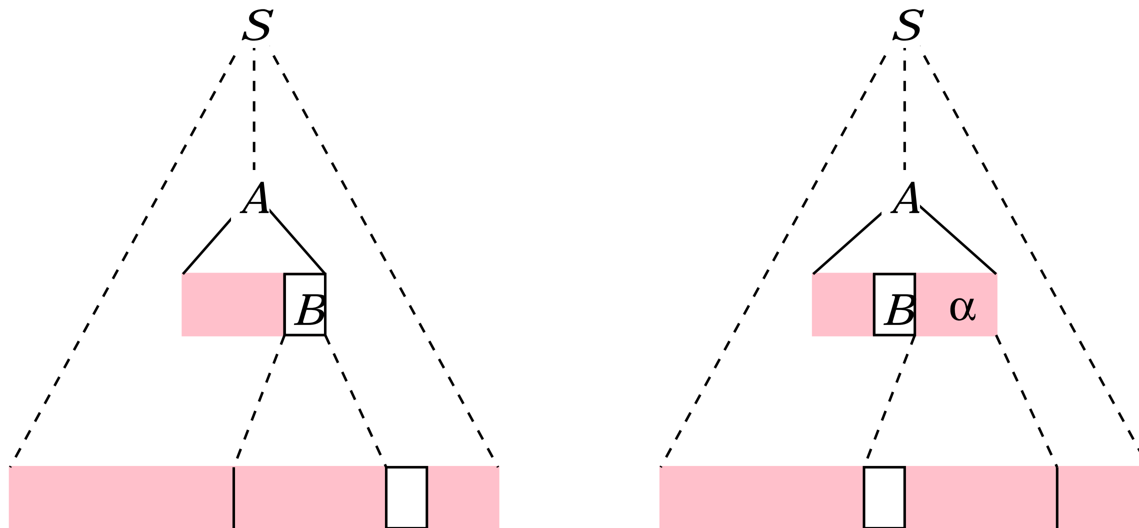
$$\mathrm{First}(T) = \mathrm{First}(F) = \{(, i\}$$

$\mathrm{First}(T')$，$\mathrm{First}(T)$と同様に，

$$\mathrm{First}(E') = \{+\}$$
$$\mathrm{First}(E) = \mathrm{First}(T) = \{(, i\}$$

follow集合の計算

1. 出発記号 S に対して $\mathrm{Follow}(S) \leftarrow \{\$\}$
   S 以外の非終端記号 A に対して $\mathrm{Follow}(A) \leftarrow \phi$
2. 生成規則 $A \rightarrow \cdots B\ (B \in V_N)$ に対して,
   $\mathrm{Follow}(B) \leftarrow \mathrm{Follow}(B) \cup \mathrm{Follow}(A)$
3. 生成規則 $A \rightarrow \cdots B\alpha\ (B \in V_N, \alpha \neq \varepsilon)$ に対して,
   $\mathrm{Follow}(B) \leftarrow \mathrm{Follow}(B) \cup \mathrm{First}(\alpha)$
   もし $\alpha \stackrel{*}{\Rightarrow} \varepsilon$ なら，さらに
   $\mathrm{Follow}(B) \leftarrow \mathrm{Follow}(B) \cup \mathrm{Follow}(A)$
4. 変更がなくなるまでステップ2と3を繰り返す.



例：文法G3のfollow集合

$$
\begin{aligned}
&E \to TE' \\
&E' \to +TE' \mid \varepsilon \\
&T \to FT' \\
&T' \to *FT' \mid \varepsilon \\
&F \to (\,E\,) \mid i
\end{aligned}
$$

出発記号Eが右辺に現れる規則は$F \to (\,E\,)$だけ

$$\mathrm{Follow}(E) = \{\,\$,)\}$$

E'が右辺に現れる生成規則は$E \to TE'$と$E' \to +TE'$

$$\mathrm{Follow}(E') = \mathrm{Follow}(E) = \{\,\$,)\}$$

Tが右辺に現れる生成規則も$E \to TE'$と$E' \to +TE'$
Tの直後はE'，$\mathrm{Null}(E') = \mathrm{true}$なので，

$$\mathrm{Follow}(T) = \mathrm{First}(E') \cup \mathrm{Follow}(E') = \{+, \$,)\}$$

$\mathrm{Follow}(E')$，$\mathrm{Follow}(T)$と同様に，

$$\mathrm{Follow}(T') = \mathrm{Follow}(T) = \{+, \$,)\}$$
$$\mathrm{Follow}(F) = \mathrm{First}(T') \cup \mathrm{Follow}(T') = \{*, +, \$,)\}$$

文法G3におけるdirector集合

$$E \to TE'$$
$$E' \to +TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \varepsilon$$
$$F \to (E) \mid i$$

$$\mathrm{Null}(E') = \mathrm{Null}(T') = \mathrm{true}$$
$$\mathrm{Null}(E) = \mathrm{Null}(T) = \mathrm{Null}(F) = \mathrm{false}$$

$$\mathrm{First}(E) = \mathrm{First}(T) = \mathrm{First}(F) = \{(, i\}$$
$$\mathrm{First}(T') = \{*\}$$
$$\mathrm{First}(E') = \{+\}$$

$$\mathrm{Follow}(E) = \mathrm{Follow}(E') = \{\$,)\}$$
$$\mathrm{Follow}(T) = \mathrm{Follow}(T') = \{+, \$,)\}$$
$$\mathrm{Follow}(F) = \mathrm{First}(T') \cup \mathrm{Follow}(T') = \{*, +, \$,)\}$$

$$\mathrm{Director}(E',\ +TE') = \{+\}$$
$$\mathrm{Director}(E',\ \varepsilon) = \mathrm{First}(\varepsilon) \cup \mathrm{Follow}(E') = \{\$,)\}$$
$$\mathrm{Director}(T',\ *FT') = \{*\}$$
$$\mathrm{Director}(T',\ \varepsilon) = \mathrm{First}(\varepsilon) \cup \mathrm{Follow}(T') = \{+, \$,)\}$$
$$\mathrm{Director}(F,\ (E)) = \{(\}$$
$$\mathrm{Director}(F,\ i) = \{i\}$$

$$\mathrm{Director}(E',\ +TE') \cap \mathrm{Director}(E',\ \varepsilon) = \phi$$
$$\mathrm{Director}(T',\ *FT') \cap \mathrm{Director}(T',\ \varepsilon) = \phi$$
$$\mathrm{Director}(F,\ (E)) \cap \mathrm{Director}(F,\ i) = \phi$$

G3はLL(1)文法である.

LL(1)構文解析法 (Left-to-right scanning, Left-most derivation)

バックトラックのない再帰的下向き構文解析

G3のための解析関数

```
tuple parse_E() {
  if (is_lpar(*p)
       || is_i(*p)) {
    tuple x = parse_T();
    return parse_E1(x);
  } else
    syntax_error();
}
```

```
tuple parse_E1(tuple x) {
  if (is_plus(*p)) {
    tuple y;
    p++;
    y = parse_T();
    return parse_E1(make_tuple("+", x, y));
  } else if (is_eof(*p) || is_rpar(*p))
    return x;
  else
    syntax_error();
}
```

バックトラックがなくなったので

1. 適切な時点で構文エラーを検出できる
2. pの値を保存しておく必要がない.

LR構文解析 (Left-to-right scanning, Right-most derivation in reverse)

文法 $\langle\{E \to E+i \mid i\}, E\rangle$ における $i+i+i$ の最右導出

$$E \underset{\rm rm}{\Longrightarrow} E+i \underset{\rm rm}{\Longrightarrow} E+i+i \underset{\rm rm}{\Longrightarrow} i+i+i$$

注目点挿入

$$\begin{aligned} E\bullet &\underset{\rm rm}{\Longrightarrow} E+i\bullet \rightharpoondown E+\bullet i \rightharpoondown E\bullet +i \\ &\underset{\rm rm}{\Longrightarrow} E+i\bullet +i \rightharpoondown E+\bullet i+i \rightharpoondown E\bullet +i+i \\ &\underset{\rm rm}{\Longrightarrow} i\bullet +i+i \rightharpoondown \bullet i+i+i \end{aligned}$$

$i+i+i\,\$$ の LR 構文解析

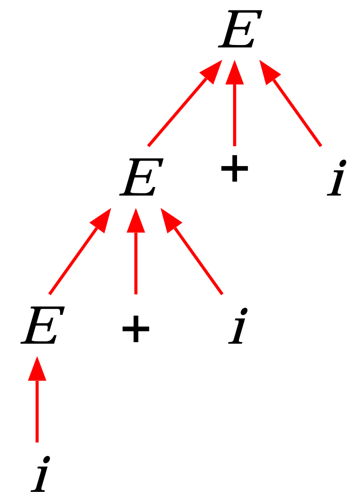
$$\begin{aligned} &\bullet i+i+i\,\$ \to i\bullet +i+i\,\$ \to E\bullet +i+i\,\$ \to E+\bullet i+i\,\$ \to E+i\bullet +i\,\$ \to E\bullet +i\,\$ \\ &\to E+\bullet i\,\$ \to E+i\bullet \$ \to E\bullet \$ \end{aligned}$$

各ステップでは,

還元（reduce）：注目点直前の記号列を還元

シフト（shift）：注目点を一つ右に移動

$\bullet i + i + i\,\$ \to i \bullet + i + i\,\$ \to E \bullet + i + i\,\$ \to E + \bullet i + i\,\$ \to E + i \bullet + i\,\$ \to E \bullet + i\,\$$
$\to E + \bullet i\,\$ \to E + i \bullet \$ \to E \bullet \$$



$A \to \alpha$で還元するときに
Aの節を用意し，αの各記号の節へ枝を張る．

上向き構文解析法(bottom-up parsing)

初期状態：$\bullet x\,\$$
x：入力トークン列

解析中の状態：$u \bullet v\,\$$
$u \in (V_T \cup V_N)^*$：構築した部分解析木の記号列
$v \in V_T^*$：未処理の入力トークン列

最終状態：$S \bullet \$$
S：出発記号

最終状態に変換できれなければ，構文エラー．

LR(0) 項

生成規則の右辺にドットをつけたもの

例：$E \to E + i$ の LR(0) 項：

$E \to \bullet E + i$　　（導入項）

$E + i$ を読み込めば E に還元できる．

$E \to E \bullet + i$

E を読み込んだ．$+ i$ を読み込めば E に還元できる．

$E \to E + \bullet i$

$E +$ を読み込んだ．i を読み込めば E に還元できる．

$E \to E + i \bullet$　　（完全項）

$E + i$ を読み込んだ．E に還元できる．

E: 文法上の出発記号

S: 新しい出発記号

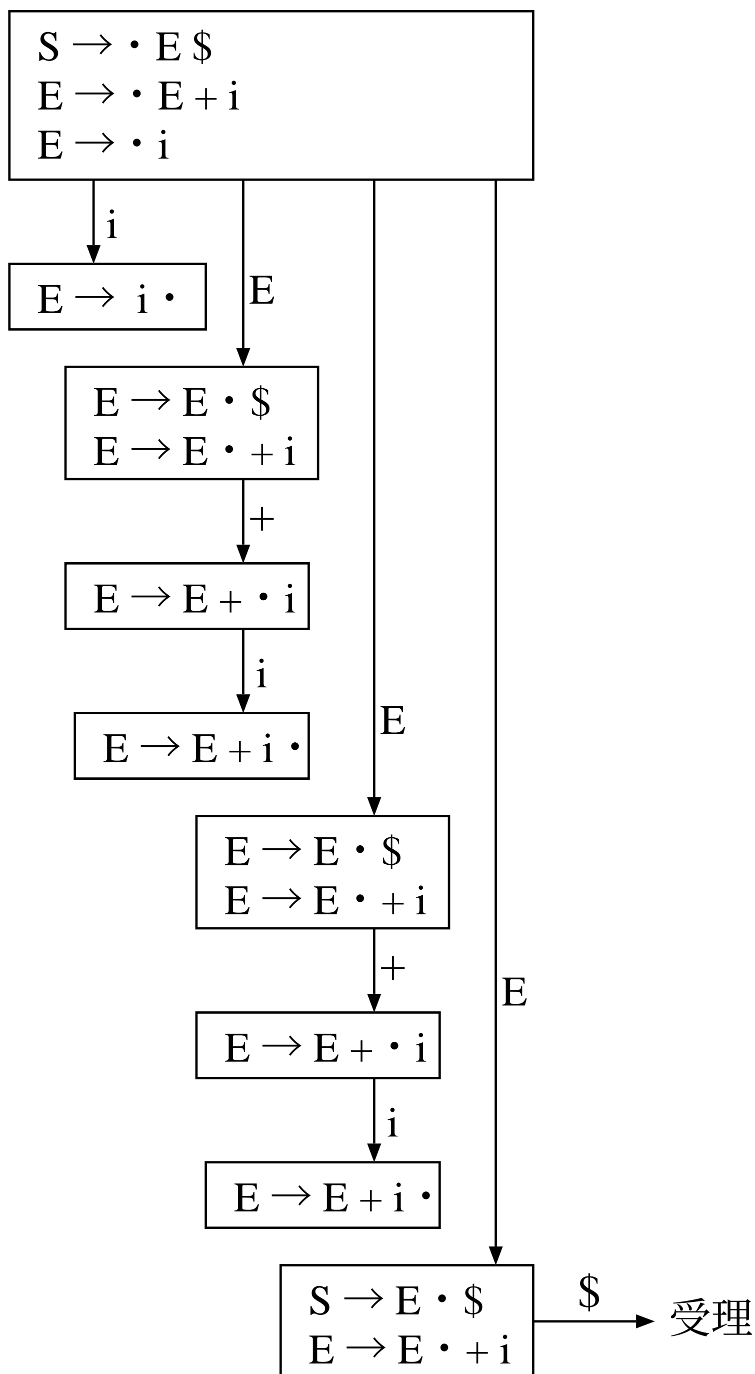
$S \to \bullet E \$$　　（出発項）

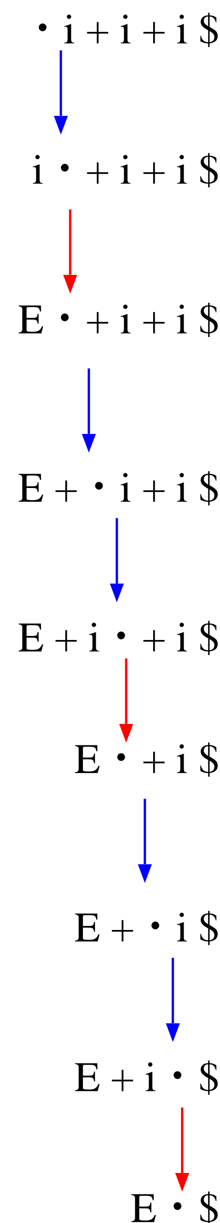
E を読み込んで $\$$ に達すれば受理できる．

$S \to E \bullet \$$　　（受理項）

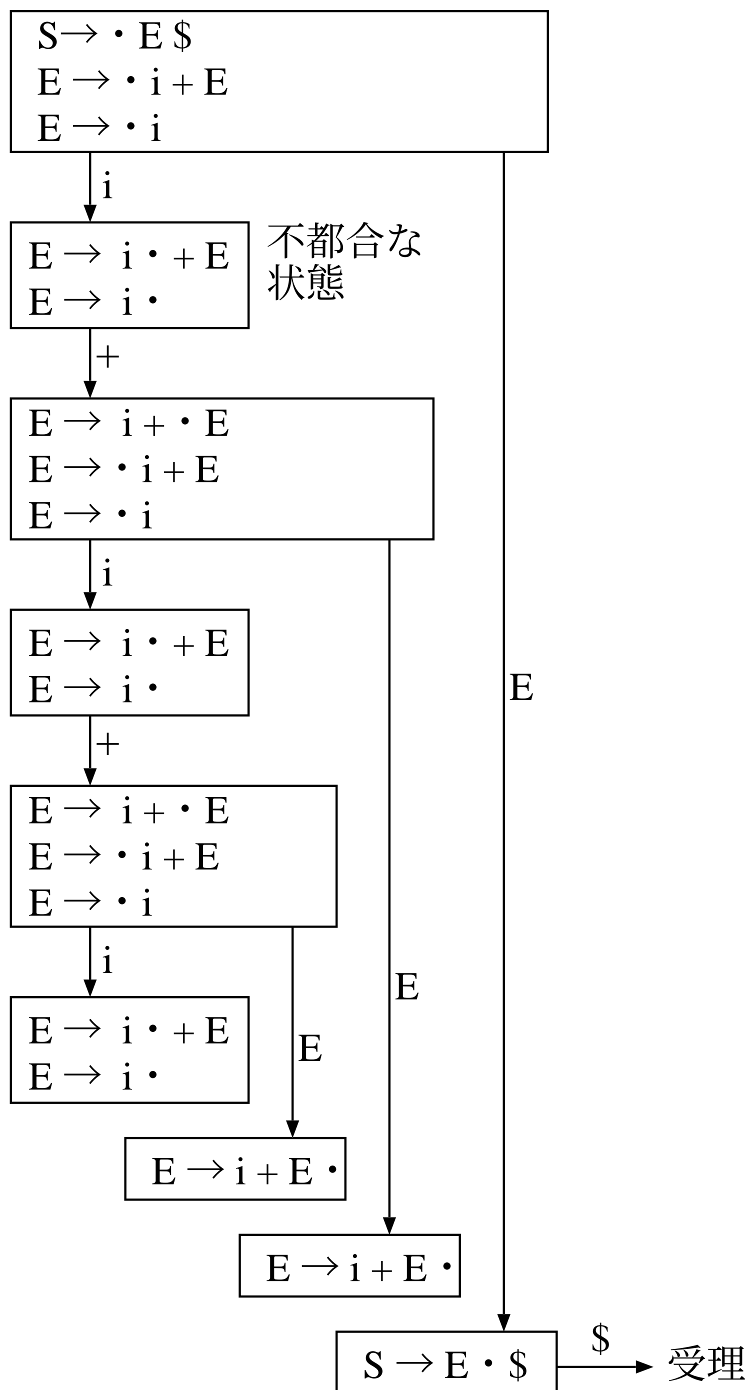
E を読み込んだ．$\$$ に達していれば受理できる．

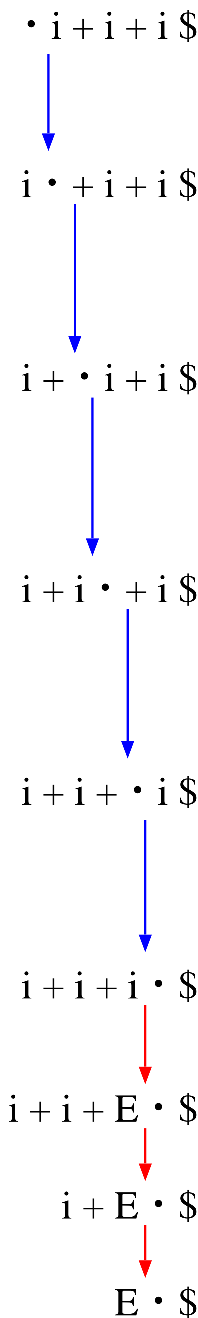
文法 $\langle \{E \to E + i \mid i\}, E\rangle$ におけるLR構文解析





文法 $\langle\{E \to i + E \mid i\}, E\rangle$ における LR 構文解析





I: LR(0) 項集合

$\mathrm{Closure}(I)$：I の閉包 (closure)

- $\mathrm{Closure}(I) \supseteq I$
- $A \to \alpha_1 \bullet B\, \alpha_2$ が $\mathrm{Closure}(I)$ の要素なら，
  任意の導入項 $B \to \bullet\, \beta$ も，$\mathrm{Closure}(I)$ の要素

例：文法 $\langle \{E \to i + E \mid i\}, E \rangle$

$$\mathrm{Closure}(S \to \bullet\, E\,\$)$$
$$= \{S \to \bullet\, E\,\$,\ \ E \to \bullet\, i + E,\ \ E \to \bullet\, i\}$$

$\mathrm{Goto}(I, a)$

$= \mathrm{Closure}(\{A \to \alpha_1\, a \bullet \alpha_2 \mid (A \to \alpha_1 \bullet a\, \alpha_2) \in I\})$

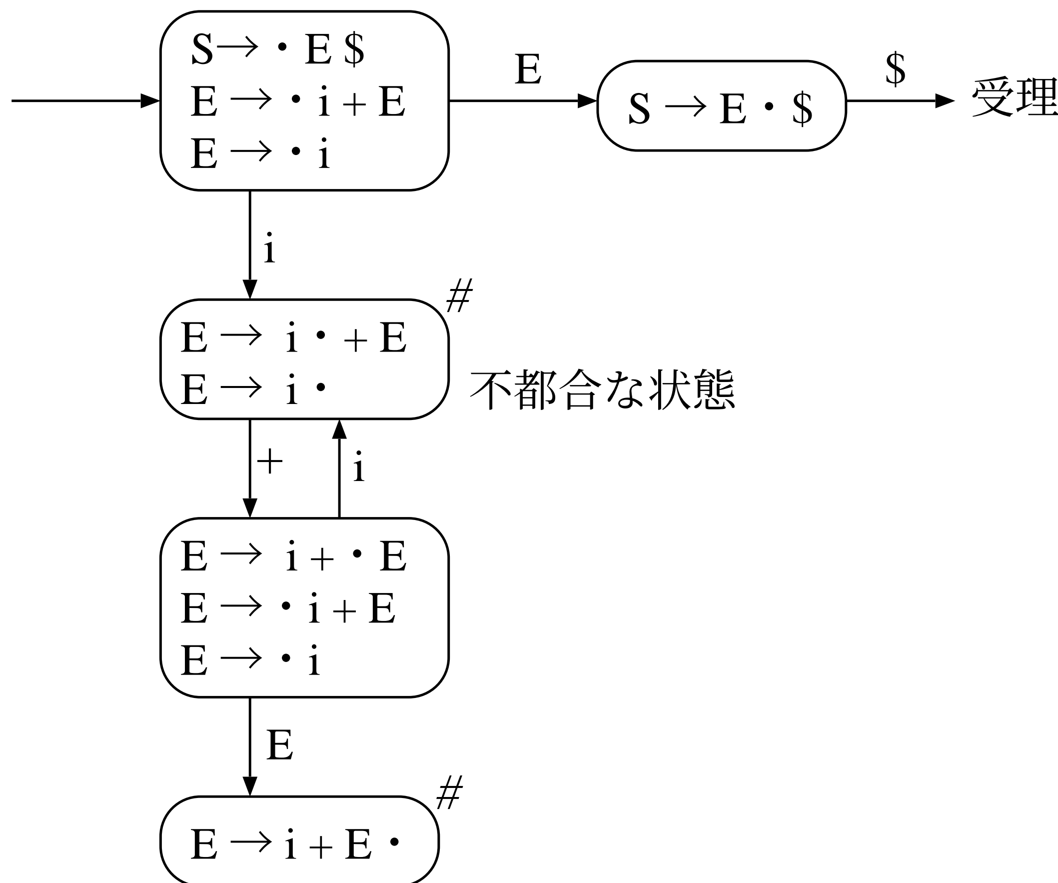
例：文法 $\langle \{E \to i + E \mid i\}, E \rangle$

$$\mathrm{Goto}(\{E \to i \bullet + E,\ \ E \to i \bullet\},\ +)$$
$$= \mathrm{Closure}(\{E \to i + \bullet\, E\})$$
$$= \{E \to i + \bullet\, E,\ \ E \to \bullet\, i + E,\ \ E \to \bullet\, i\}$$

正準オートマトン (canonical automaton)

初期状態 Closure($\{S \to \cdot E\,\$\}$) から始め，可能な遷移をすべて求める

例：文法 $\langle\{E \to i + E \mid i\}, E\rangle$ の正準オートマトン



#: 還元状態（完全項を含む状態）

正準集合：正準オートマトンの状態集合

配置(configuration)

LR構文解析のある時点

$$a_1 a_2 \cdots a_n{}^\bullet x_k x_{k+1} \cdots x_m \$$$

における配置は,

$$(q_0\, a_1\, q_1\, a_2\, q_2 \cdots a_n\, q_n,\quad x_k x_{k+1} \cdots x_m \$)$$

q_0：初期状態，$q_i = \mathrm{Goto}(q_{i-1}, a_i) = \mathrm{Goto}(q_0, a_1 a_2 \cdots a_n)$

LR構文解析開始時点の配置：

$$(q_0,\quad x_1 x_2 \cdots x_m \$)$$

$A \to a_i \cdots a_n$による還元後：

$$(q_0\, a_1\, q_1 \cdots a_{i-1}\, q_{i-1}\, A\, q,\quad x_k x_{k+1} \cdots x_m \$)$$

$$q = \mathrm{Goto}(q_{i-1}, A)$$

x_kを読み込んでシフト後：

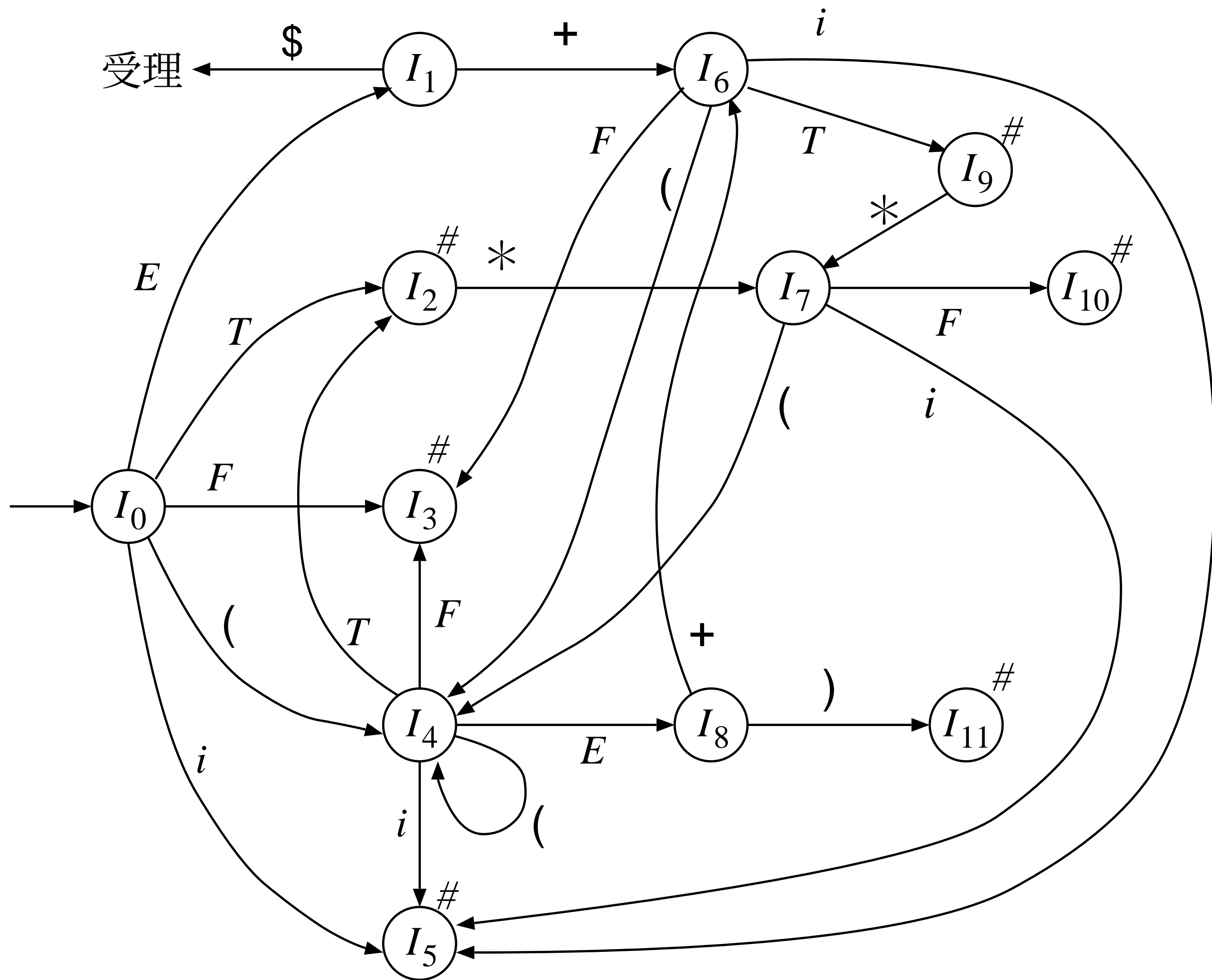
$$(q_0\, a_1\, q_1 \cdots a_n\, q_n\, x_k\, q,\quad x_{k+1} \cdots x_m \$)$$

$$q = \mathrm{Goto}(q_n, x_k)$$

最終的な配置：

$$(q_0\, S\, q,\quad \$)$$

例：文法G1 = $\langle \mathrm{P1}, E \rangle$ の正準オートマトン



$$\mathrm{P1} = \{\ E \to E+T \mid T \quad T \to T*F \mid F \quad F \to (\ E\ ) \mid i\ \}$$

初期状態 I_0

$= \mathrm{Closure}(\{S \to \cdot E\,\$\})$

#: 還元状態

不都合な状態（とりあえずシフトを優先）：

$I_2 = \{\ E \to T\cdot,\ \ T \to T\cdot *F\ \}$

$I_9 = \{\ E \to E+T\cdot,\ \ T \to T\cdot *F\ \}$

G1の正準集合

$$
\begin{aligned}
I_0 = \{\ & S \to \cdot E\,\$, \\
& E \to \cdot E + T, \\
& E \to \cdot T, \\
& T \to \cdot T * F, \\
& T \to \cdot F, \\
& F \to \cdot (E), \\
& F \to \cdot i\ \}
\end{aligned}
$$

$$
\begin{aligned}
I_1 = \{\ & S \to E \cdot \$, \\
& E \to E \cdot + T\ \}
\end{aligned}
$$

$$
\begin{aligned}
I_2 = \{\ & E \to T \cdot, \\
& T \to T \cdot * F\ \}
\end{aligned}
$$

$$
I_3 = \{\ T \to F \cdot\ \}
$$

$$
\begin{aligned}
I_4 = \{\ & F \to (\cdot E), \\
& E \to \cdot E + T, \\
& E \to \cdot T \\
& T \to \cdot T * F, \\
& T \to \cdot F, \\
& F \to \cdot (E), \\
& F \to \cdot i\ \}
\end{aligned}
$$

$$
I_5 = \{\ F \to i \cdot\ \}
$$

$$
\begin{aligned}
I_6 = \{\ & E \to E + \cdot T, \\
& T \to \cdot T * F, \\
& T \to \cdot F, \\
& F \to \cdot (E), \\
& F \to \cdot i\ \}
\end{aligned}
$$

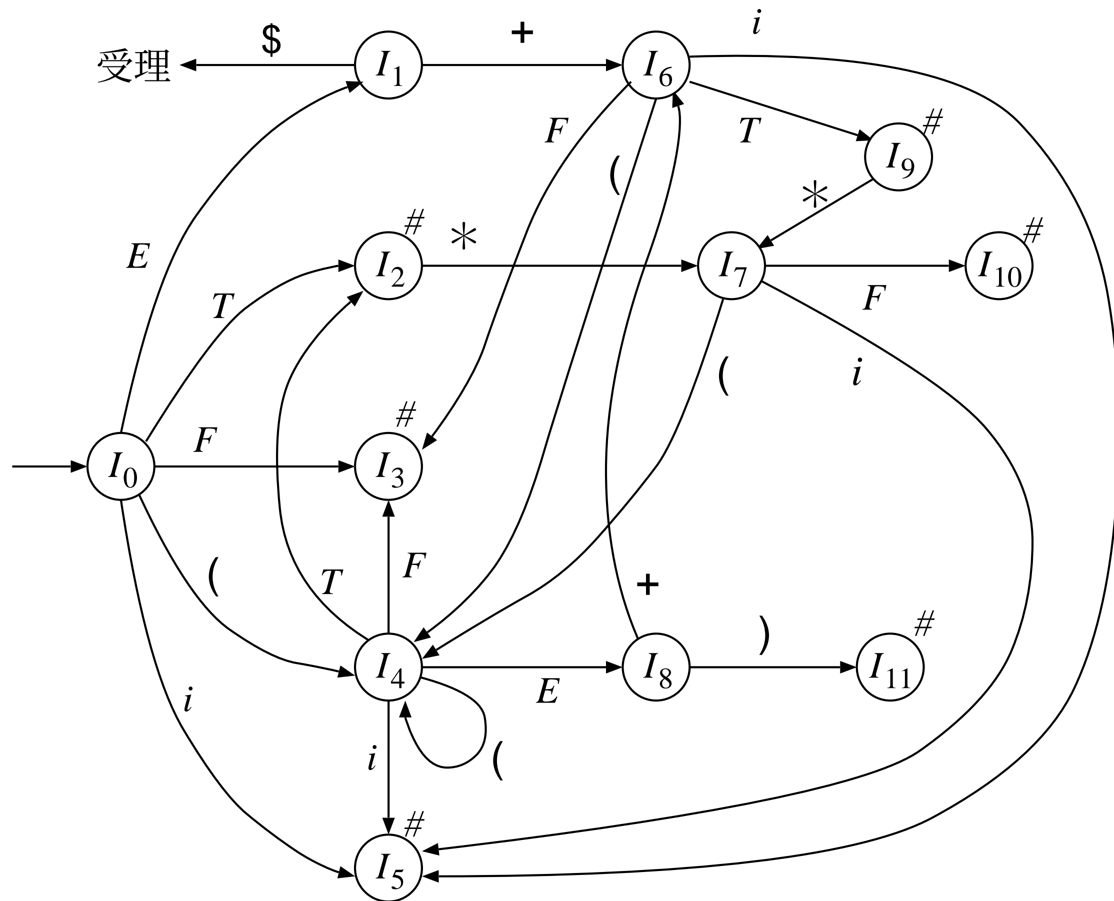
$$
\begin{aligned}
I_7 = \{\ & T \to T * \cdot F, \\
& F \to \cdot (E), \\
& F \to \cdot i\ \}
\end{aligned}
$$

$$
\begin{aligned}
I_8 = \{\ & F \to (E \cdot), \\
& E \to E \cdot + T\ \}
\end{aligned}
$$

$$
\begin{aligned}
I_9 = \{\ & E \to E + T \cdot, \\
& T \to T \cdot * F\ \}
\end{aligned}
$$

$$
I_{10} = \{\ T \to T * F \cdot\ \}
$$

$$
I_{11} = \{\ F \to (E) \cdot\ \}
$$



スタック	入力	動作	スタック	入力	動作
$(I_0,$	$i + i * i\,\$)$	iの読込み	$(I_0\,E\,I_1 + I_6\,F\,I_3,$	$* i\,\$)$	$T \to F$で還元
$(I_0\,i\,I_5,$	$+ i * i\,\$)$	$F \to i$で還元	$(I_0\,E\,I_1 + I_6\,T\,I_9,$	$* i\,\$)$	‘$*$’の読込み
$(I_0\,F\,I_3,$	$+ i * i\,\$)$	$T \to F$で還元	$(I_0\,E\,I_1 + I_6\,T\,I_9 * I_7,$	$i\,\$)$	iの読込み
$(I_0\,T\,I_2,$	$+ i * i\,\$)$	$E \to T$で還元	$(I_0\,E\,I_1 + I_6\,T\,I_9 * I_7\,i\,I_5,$	$\$)$	$F \to i$で還元
$(I_0\,E\,I_1,$	$+ i * i\,\$)$	‘+’の読込み	$(I_0\,E\,I_1 + I_6\,T\,I_9 * I_7\,F\,I_{10},$	$\$)$	$T \to T * F$で還元
$(I_0\,E\,I_1 + I_6,$	$i * i\,\$)$	iの読込み	$(I_0\,E\,I_1 + I_6\,T\,I_9,$	$\$)$	$E \to E + T$で還元
$(I_0\,E\,I_1 + I_6\,i\,I_5,$	$* i\,\$)$	$F \to i$で還元	$(I_0\,E\,I_1,$	$\$)$	受理

SLR(1) 構文解析

Simple LR,
１文字先読み (lookahead) し，
follow 集合で不都合な状態を解消

SLR(1) 文法：SLR(1) 構文解析が可能な文法

例：G1 の不都合な状態 I_2

$$I_2 = \{ E \rightarrow T\bullet,\ \ T \rightarrow T\bullet * F \}$$

次の入力トークンが

1. $\mathrm{Follow}(E) = \{+,), \$\}$ の要素なら，$E \rightarrow T$ で還元
2. ‘*’ なら，‘*’ を読み込んでシフト
3. どちらでもなければ，構文エラー

$I_9 = \{ E \rightarrow E + T\bullet,\ \ T \rightarrow T\bullet * F \}$
$\mathrm{Follow}(E) = \{+,), \$\} \not\ni *$

$\rightarrow$ G1 は SLR(1) 文法

	i	+	*	(	)	$
I_0	s5			s4		
I_1		s6				受理
I_2		r2	s7		r2	r2
I_3		r4	r4		r4	r4
I_4	s5			s4		
I_5		r6	r6		r6	r6
I_6	s5			s4		
I_7	s5			s4		
I_8		s6			s11	
I_9		r1	s7		r1	r1
I_{10}		r3	r3		r3	r3
I_{11}		r5	r5		r5	r5

動作表

	E	T	F
I_0	I_1	I_2	I_3
I_4	I_8	I_2	I_3
I_6		I_9	I_3
I_7			I_{10}

行先表

1: $E \rightarrow E + T$
2: $E \rightarrow T$
3: $T \rightarrow T * F$
4: $T \rightarrow F$
5: $F \rightarrow (E)$
6: $F \rightarrow i$

動作表(action table)
- sj：状態I_jにシフト
- rj：j番目の生成規則で還元
- 受理：構文解析終了
- 空欄：動作未定義 → 構文エラー

行先表(goto table)
- 還元後の状態（Goto(I, a)）

SLR(1)構文解析プログラム（文法に依存せず）

```
tuple parse() {
  int state = 0;
  for (;;) {
    action act = lookup_action(state, *p);
    if (act.todo == SHIFT) {
      push_state(state);
      push_value(*p++);
      state = act.num;
    } else if (act.todo == REDUCE) {
      int nt = reduce(act.num);
      state = lookup_goto(top_state(), nt);
    } else if (act.todo == ACCEPT)
      return pop_value();
    else  /* if (act.todo == UNDEFINED) */
      syntax_error();
  }}
```

配置 $(q_0\, a_1\, q_1 \cdots q_{n-1}\, a_n\, q_n,\ \ x_k x_{k+1} \cdots x_m\$)$ の表現：

変数 `state`：q_n

state スタック：$q_0, q_1, \cdots, q_{n-1}$　　value スタック：$a_1, \cdots, a_n$

還元用関数（文法に依存）

```
int reduce(int production) {
  tuple x, y;
  switch (production) {
  case 1: /* E -> E + T */
    x = pop_value(); pop_value();
    y = pop_value();
    pop_state(); pop_state();
    push_value(make_tuple("+", y, x));
  case 2: /* E -> T */
    return E;
  case 3: /* T -> T * F */ ...
  case 4: /* T -> F */     ...
  case 5: /* F -> ( E ) */ ...
  case 6: /* F -> i */     ...
  }}
```

$E \to E + T$ による還元：

$$(q_0\, a_1\, q_1 \cdots a_{n-3}\, q_{n-3}\, E_1\, q_{n-2} + q_{n-1}\, T_1\, q_n,\ \ \cdots) \to (q_0\, a_1\, q_1 \cdots a_{n-3}\, q_{n-3}\, E_2\, q\,,\ \ \cdots)$$

E_1 の構文木は T_1 と同じ

LR(1) や LALR(1) でも，プログラムは同じ．動作表と行先表が異なる．

LR(1)構文解析

follow集合より正確な先読み記号を使用

$A \to \alpha$による還元

$$S \bullet \$ \overset{*}{\underset{\mathrm{rm}}{\Longrightarrow}} uA \bullet v\,\$ \underset{\mathrm{rm}}{\Longrightarrow} u\alpha \bullet v\,\$$$

次の入力xは，$x \in \mathrm{First}(v\,\$)$のはず．

LR(1)項：次の入力がxなら$A \to \alpha$で還元可能

$$[A \to \alpha \bullet,\ x]$$

LR(1)項の一般形：

$$[A \to \alpha_1 \bullet \alpha_2,\ x]$$

完全項を核(core)とするLR(1)項

$$[A \to \alpha_1\alpha_2 \bullet,\ x]$$

になって次の入力がxなら，$A \to \alpha_1\alpha_2$で還元可能

LR(1)構文解析の初期状態：

$$I_0 = \mathrm{Closure}(\{\,[S' \to \bullet S,\ \$]\,\})$$

$$\mathrm{Goto}(I, a) = \mathrm{Closure}(\{[A \to \alpha_1 a \bullet \alpha_2,\ c] \mid [A \to \alpha_1 \bullet a\,\alpha_2,\ c] \in I\})$$

LR(1) 項集合 I の閉包 $I' = \mathrm{Closure}(I)$

1. $I' \leftarrow I$
2. I' の要素 $[A \rightarrow \alpha_1 \cdot B\,\alpha_2,\ x]$,
   生成規則 $B \rightarrow \beta$,
   $x' \in \mathrm{First}(\alpha_2\, x)$ であるすべての x' について,
   $[B \rightarrow \cdot\,\beta,\ x']$ を I' に追加.
3. I' に追加がなくなるまで，ステップ 2 を繰り返す.

例：G1 を LR(1) 構文解析するときの初期状態

$$
\begin{aligned}
I_0 &= \mathrm{Closure}(\{\ [S \rightarrow \cdot E,\ \$]\ \}) \\
&= \{\ [S \rightarrow \cdot E,\ \$], \\
&\qquad [E \rightarrow \cdot E + T,\ \$],\ [E \rightarrow \cdot T,\ \$],\ [E \rightarrow \cdot E + T,\ +],\ [E \rightarrow \cdot T,\ +], \\
&\qquad [T \rightarrow \cdot T * F,\ \$],\ [T \rightarrow \cdot F,\ \$],\ [T \rightarrow \cdot T * F,\ +],\ [T \rightarrow \cdot F,\ +], \\
&\qquad [T \rightarrow \cdot T * F,\ *],\ [T \rightarrow \cdot F,\ *],\ [F \rightarrow \cdot (\ E\ ),\ \$],\ [F \rightarrow \cdot i,\ \$], \\
&\qquad [F \rightarrow \cdot (\ E\ ),\ +],\ [F \rightarrow \cdot i,\ +],\ [F \rightarrow \cdot (\ E\ ),\ *],\ [F \rightarrow \cdot i,\ *]\ \} \\
&= \{\ [S \rightarrow \cdot E,\ \$], \\
&\qquad [E \rightarrow \cdot E + T,\ \$/+],\ [E \rightarrow \cdot T,\ \$/+], \\
&\qquad [T \rightarrow \cdot T * F,\ \$/+/*],\ [T \rightarrow \cdot F,\ \$/+/*], \\
&\qquad [F \rightarrow \cdot (\ E\ ),\ \$/+/*],\ [F \rightarrow \cdot i,\ \$/+/*]\ \}
\end{aligned}
$$

LR(1) 構文解析のための正準オートマトンの作り方：

SLR(1) の場合とまったく同じ．

例：G1 を LR(1) 構文解析するためのオートマトン

$$
\begin{aligned}
I_2 &= \mathrm{Goto}(I_0,\ T) = \mathrm{Closure}(\{\ [E \to T\bullet,\ \ \$/+],\ \ [T \to T\bullet * F,\ \ \$/+/*]\ \})\\
&= \{\ [E \to T\bullet,\ \ \$/+],\ [T \to T\bullet * F,\ \ \$/+/*]\ \}\\
I_4 &= \mathrm{Goto}(I_0,\ (\,) = \mathrm{Closure}(\{\ [F \to (\bullet E\,),\ \ \$/+/*]\ \})\\
&= \{\ [F \to\ (\bullet E\,),\ \ \$/+/*],\\
&\qquad [E \to \bullet E + T,\ \ )/+],\ \ [E \to \bullet T,\ \ )/+],\\
&\qquad [T \to \bullet T * F,\ \ )/+/*],\ \ [T \to \bullet F,\ \ )/+/*],\\
&\qquad [F \to \bullet (\,E\,),\ \ )/+/*],\ \ [F \to \bullet i,\ \ )/+/*]\ \}\\
I_2' &= \mathrm{Goto}(I_4,\ T) = \mathrm{Closure}(\{\ [E \to T\bullet,\ \ )/+],\ \ [T \to T\bullet * F,\ \ )/+/*]\ \})\\
&= \{\ [E \to T\bullet,\ \ )/+],\ \ [T \to T\bullet * F,\ \ )/+/*]\ \}\\
&\neq I_2
\end{aligned}
$$

SLR(1) のときは $\mathrm{Goto}(I_4, T) = I_2$ だった．
→ LR(1) は，より詳細な構文解析が可能．

SLR(1) 文法はすべて LR(1) 文法．
LR(1) 文法で，SLR(1) 文法でないものが存在する．

LALR(1) 構文解析

LR(1) 構文解析法は，状態数が多過ぎて実用的でない．

核が同じである LR(1) 項を同一視して状態数を減らす．

例：

$$I_2 = \{\, [E \to T\cdot,\ \$/+],\ [T \to T\cdot * F,\ \$/+/*] \,\}$$
$$I_2' = \{\, [E \to T\cdot,\ )/+],\ [T \to T\cdot * F,\ )/+/*] \,\}$$

を融合し，

$$I_2 = \{\, [E \to T\cdot,\ \$/+/)],\ [T \to T\cdot * F,\ \$/+/*/)] \,\}$$

さらに

$$\mathrm{Goto}(I_4,\ T) = I_2$$

LALR(1) は，正確さは LR(1) より劣るが，
SLR(1) よりは正確．
状態数は SLR(1) と同じ．

LALR(1) 文法で，SLR(1) 文法でない例：

$$\mathrm{P4} = \{\ S \to E + E \mid i \\ E \to i\ \}$$

$I_0 = \{\ [S' \to \cdot S,\ \$],\ [S \to \cdot E + E,\ \$],\ [S \to \cdot i,\ \$],\ [E \to \cdot i,\ +]\ \}$

$I_1 = \{\ [S' \to S\cdot,\ \$]\ \}$

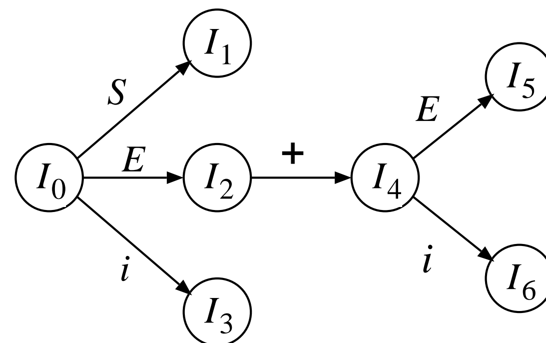
$I_2 = \{\ [S \to E\cdot + E,\ \$]\ \}$

$I_3 = \{\ [S \to i\cdot,\ \$],\ [E \to i\cdot,\ +]\ \}$

$I_4 = \{\ [S \to E\cdot E,\ \$],\ [E \to \cdot i,\ \$]\ \}$

$I_5 = \{\ [S \to E + E\cdot,\ \$]\ \}$

$I_6 = \{\ [E \to i\cdot,\ \$]\ \}$



この文法はLALR(1) 文法（LR(1) 文法でもある）

SLR(1) 用の $I_3 = \{S \to i\cdot,\ \ E \to i\cdot\}$ について

$\mathrm{Follow}(S) = \{\$\}$，$\mathrm{Follow}(E) = \{\$, +\}$

なので，SLR(1) 文法でない．

ポイント：

I_3は，I_0からiを読み込んだ直後の状態.

$E \to i$で還元すれば，$S \to E + E$の最初のE.

だから，次の入力は‘+’

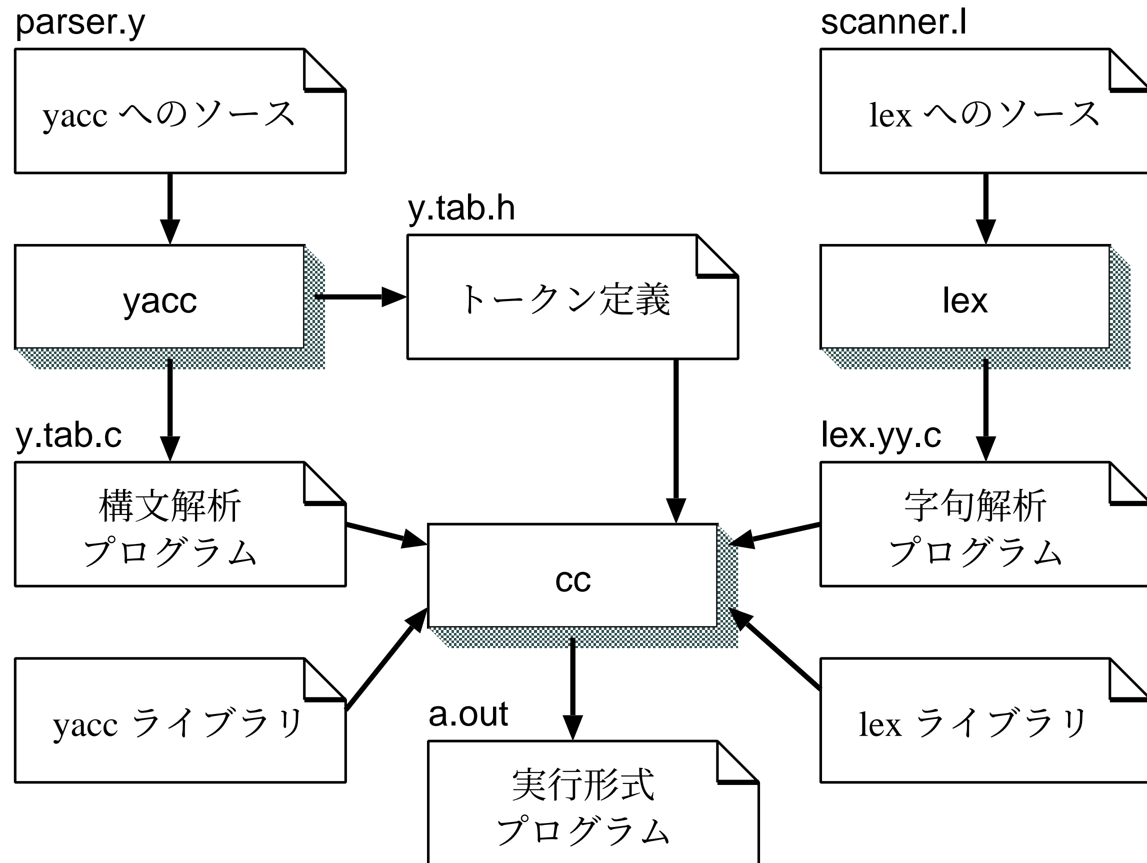
二つ目のEのために，Follow(E)が\$を含む.

構文解析の自動化(yacc – Yet Another Compiler Compiler)

```
% cat parser.y
%token PLUS TIMES LPAR RPAR IDENTIFIER
%%
E:  E PLUS T {$$=make_tuple("+",$1,$3);}
  | T        {$$=$1;}
T:  T TIMES F {$$=make_tuple("*",$1,$3);}
  | F         {$$=$1;}
F:  LPAR E RPAR {$$=$2;}
  | IDENTIFIER  {$$=$1;}
%%
  ··· ここにmake_tuple等の定義 ···
% yacc -d parser.y
% ls -l
...    565 Aug  2 23:36 parser.y
...  11838 Aug  2 23:36 y.tab.c
...     92 Aug  2 23:36 y.tab.h
% cat y.tab.h
#define PLUS 257
#define TIMES 258
...
%
```

yacc用にlexを使う例：

```
% cat scanner.l
%{
#include "y.tab.h"
extern int yylval;
%}
%%
[\t ]+ {}
"+"     {return PLUS;}
"*"     {return TIMES;}
"("     {return LPAR;}
")"     {return RPAR;}
[a-z][a-z0-9]* {yylval = copy_lexeme(); return IDENTIFIER;}
\n      {return 0;}
.       {printf("lexical error: '%c'\n", yytext[0]); return 0;}
%%
char StringTable[1000]; char *STp = StringTable;
int copy_lexeme() {
  int p = (int) STp; strcpy(STp, yytext);
  STp += strlen(yytext)+1; return p;
}
%
```



make_tupleに構文木を表示させる：

```
int prev_tuple = 0;
int make_tuple(char *op, int arg1, int arg2) {
  printf("T%d: (%s, ", ++prev_tuple, op);
  if (arg1 > 0) printf("%s, ", arg1); else printf("T%d, ", -arg1);
  if (arg2 > 0) printf("%s", arg2); else printf("T%d", -arg2);
  printf(")\n"); return -prev_tuple;
}
```

実行例：

```
% lex scanner.l
% cc y.tab.c lex.yy.c -ly -ll
% a.out
x*y+z*w
T1: (*, x, y)
T2: (*, z, w)
T3: (+, T1, T2)
% a.out
x+y+
T1: (+, x, y)
syntax error
%
```

yaccを使って，動作表と行先表の情報を調べる：

```
% yacc -v parser.y
% cat y.output
state 0
    $accept : . E $end  (0)
    IDENTIFIER  shift 5
    LPAR  shift 4
    .  error
    E  goto 1
    T  goto 2
    F  goto 3
state 1
  ... 中略 ...
state 11
    F : LPAR E RPAR .  (5)
    .  reduce 5
7 terminals, 4 nonterminals
7 grammar rules, 12 states
%
```

あいまいな文法への対処

文法 $\mathrm{Gif} = \langle \mathrm{Pif}, S \rangle$ を使用する.

$$
\begin{aligned}
\mathrm{Pif} = \{\ & S \rightarrow \texttt{if (} E \texttt{)} S \mid \texttt{if (} E \texttt{)} S \texttt{ else } S \mid i \texttt{ = } E \texttt{ ;} \\
& E \rightarrow i\ \}
\end{aligned}
$$

LL(1)構文解析の場合

くくり出しの結果Gif' $= \langle \mathrm{Pif'}, S \rangle$

$$\mathrm{Pif'} = \{\ S \rightarrow \texttt{if}\ \texttt{(}\ E\ \texttt{)}\ S\ A \mid i\ \texttt{=}\ E\ \texttt{;}$$
$$A \rightarrow \texttt{else}\ S \mid \varepsilon$$
$$E \rightarrow i\ \}$$

$\mathrm{First}(E) = \{i\}$
$\mathrm{First}(A) = \{\texttt{else}\} \cup \mathrm{Follow}(A)$
$\quad = \{\texttt{else}\} \cup \mathrm{Follow}(S) = \{\texttt{else}, \$\}$
$\mathrm{First}(S) = \{\texttt{if}, i\}$
$\mathrm{Follow}(S) = \{\$\} \cup \mathrm{First}(A) = \{\$, \texttt{else}\}$
$\mathrm{Follow}(A) = \mathrm{Follow}(S) = \{\$, \texttt{else}\}$
$\mathrm{Follow}(E) = \{\texttt{)}, \texttt{;}\}$

$\mathrm{Director}(S,\ \texttt{if}\ \texttt{(}\ E\ \texttt{)}\ S\ A) = \{\texttt{if}\}$
$\mathrm{Director}(S,\ i\ \texttt{=}\ E\ \texttt{;}) = \{i\}$
$\mathrm{Director}(A,\ \texttt{else}\ S) = \{\texttt{else}\}$
$\mathrm{Director}(A,\ \varepsilon) = \mathrm{Follow}(A) = \{\$, \texttt{else}\}$
$\mathrm{Director}(E,\ i) = \{i\}$

$\mathrm{Director}(A,\ \texttt{else}\ S) \cap \mathrm{Director}(A,\ \varepsilon) = \{\texttt{else}\} \neq \phi$
なので，Gif'は，LL(1)文法でない．

次が`else`なら，$A \to$ `else` Sを採用することにする．

```
tuple parse_A() {
  if (is_else(*p)) {
    p++;
    return parse_S();
  } else if (is_eof(*p))
    return EMPTY;
  else
    syntax_error();
}
```

```
tuple parse_S() {
  tuple x, y, z;
  if (is_if(*p)) {
    p++;
    if (!is_lpar(*p++)) syntax_error();
    x = parse_E();
    if (!is_rpar(*p++)) syntax_error();
    y = parse_S(); z = parse_A();
    return make_tuple("if", x, y, z);
  } else if (is_i(*p)) {
    x = *p++;
    if (!is_eq(*p++)) syntax_error();
    y = parse_E();
    if (!is_semicolon(*p++)) syntax_error();
    return make_tuple("=", x, y);
  } else
    syntax_error();
}
```

LR構文解析の場合

Gifの正準集合（抜粋）:

$$
\begin{aligned}
I_0 &= \mathrm{Closure}(\{S' \to \cdot S\,\$\}) \\
&= \{S' \to \cdot S\,\$,\ \ S \to \cdot\,\texttt{if}\ (\ E\ )\ S,\ \ S \to \cdot\,\texttt{if}\ (\ E\ )\ S\ \texttt{else}\ S,\ \ S \to \cdot\, i\ \texttt{=}\ E\ ;\} \\
I_1 &= \mathrm{Goto}(I_0,\ \texttt{if}) = \{S \to \texttt{if}\cdot(\ E\ )\ S,\ \ S \to \texttt{if}\cdot(\ E\ )\ S\ \texttt{else}\ S\} \\
I_2 &= \mathrm{Goto}(I_1,\ (\,) = \mathrm{Closure}(\{S \to \texttt{if}\ (\cdot E\ )\ S,\ \ S \to \texttt{if}\ (\cdot E\ )\ S\ \texttt{else}\ S\}) \\
&= \{S \to \texttt{if}\ (\cdot E\ )\ S,\ \ S \to \texttt{if}\ (\cdot E\ )\ S\ \texttt{else}\ S,\ \ E \to \cdot\, i\} \\
I_3 &= \mathrm{Goto}(I_2,\ E) = \{S \to \texttt{if}\ (\ E\cdot)\ S,\ \ S \to \texttt{if}\ (\ E\cdot)\ S\ \texttt{else}\ S\} \\
I_4 &= \mathrm{Goto}(I_3,\ )\,) = \mathrm{Closure}(\{S \to \texttt{if}\ (\ E\ )\cdot S,\ \ S \to \texttt{if}\ (\ E\ )\cdot S\ \texttt{else}\ S\}) \\
&= \{S \to \texttt{if}\ (\ E\ )\cdot S,\ \ S \to \texttt{if}\ (\ E\ )\cdot S\ \texttt{else}\ S, \\
&\qquad S \to \cdot\,\texttt{if}\ (\ E\ )\ S,\ \ S \to \cdot\,\texttt{if}\ (\ E\ )\ S\ \texttt{else}\ S,\ \ S \to \cdot\, i\ \texttt{=}\ E\ ;\} \\
I_5 &= \mathrm{Goto}(I_4,\ S) = \{S \to \texttt{if}\ (\ E\ )\ S\cdot,\ \ S \to \texttt{if}\ (\ E\ )\ S\cdot\texttt{else}\ S\}
\end{aligned}
$$

不都合な状態I_5について

$$\mathrm{Follow}(S) = \{\$,\ \texttt{else}\}$$

なので，GifはSLR(1)文法ではない．

シフトすると決めてしまえばSLR(1)構文解析は可能．
$\mathrm{Action}(I_5, \texttt{else}) = \mathrm{s}6$として動作表を作る．

GifはLR(1)文法でもLALR(1)文法でもない．
同様に対処すれば，LR(1)・LALR(1)構文解析も可能．

yaccの場合

yaccのルール

1. 還元よりもシフトを優先する．
2. 先に与えた生成規則による還元を優先する．

```
% cat Gif.y
  ... 中略 ...
S:  IF LPAR E RPAR S {$$=make_tuple("if",$3,$5,EMPTY);}
  | IF LPAR E RPAR S ELSE S {$$=make_tuple("if",$3,$5,$7);}
  | IDENTIFIER EQ E SEMICOLON {$$=make_tuple("=",$1,$3);}
E:  IDENTIFIER {$$=$1;}
  ... 中略 ...
% yacc -d Gif.y
conflicts: 1 shift/reduce
% cc y.tab.c lex.yy.c -ly -ll
% a.out
if (x) if (y) a=b; else b=a;
T1: (=, a, b)
T2: (=, b, a)
T3: (if, y, T1, T2)
T4: (if, x, T3, -)
%
```

エラーリカバリ

1回のコンパイルで，できるだけ多くのエラーを検出

エラー検出で構文解析をただちに終了しないで，
なんらかの処置を行って実行を継続.

LL(1)構文解析のエラーリカバリ

`parse_`Aがエラーを検出するのは：

1. 次の入力が，どのDirector(A, α)にも属さない.
2. $A \to \alpha$の解析中に，次の入力が，期待した終端記号と異なる.

例：Gif'の`parse_S`

Aから生成されたであろうトークン列を予測し，それらを読み飛ばす.

1の場合：

$A \overset{*}{\Rightarrow} \cdots a$である終端記号$a$まで
または，Follow(A)の要素の直前まで読み飛ばす.

2の場合：

期待する終端記号の自動挿入（読み飛ばしなし）
または，期待する終端記号まで読み飛ばす.

LR構文解析のエラーリカバリ

動作表の値が「未定義」のときにエラー検出

非終端記号ごとのエラーリカバリは困難
→大きな単位（文，宣言，関数定義など）について

非終端記号Aについてのエラーリカバリ：

$$(q_0\, a_1\, q_1 \cdots a_n\, q_n,\ \ x_k x_{k+1} \cdots x_m \$)$$

の配置で，$\mathrm{Action}(q_n, x_k)$が未定義

エラーがなくAへの還元が成功していれば，

$$(q_0\, a_1\, q_1 \cdots q_i\, A\, q,\ \ x_j x_{j+1} \cdots x_m \$) \qquad 0 \le i < n,\ k \le j \le m,\ q = \mathrm{Goto}(q_i, A)$$

スタックの巻き戻し：$\mathrm{Goto}(q_i, A)$が定義されているq_iが見つかるまで

入力トークンの読み飛ばし：$\mathrm{Follow}(A)$の要素x_jが見つかるまで

GifのSについてのエラーリカバリ：

```
while ((state = lookup_goto(top_state(), S)) < 0) {
  pop_state(); pop_value();
}
pop_value(); push_value(ERROR);
while (!is_eof(*p) && !is_else(*p)) p++;
```

yaccのエラーリカバリ: エラー規則を使用

```
S:  IF LPAR E RPAR S { ... }
  | IF LPAR E RPAR S ELSE S { ... }
  | IDENTIFIER EQ E SEMICOLON { ... }
  | error EQ E SEMICOLON {
      printf("Error: bad lvar.\n"); $$=$3;}
  | IDENTIFIER error E SEMICOLON {
      printf("Error: '=' expected.\n");
      $$=make_tuple("=",$1,$3);}
```

「$S \to i$ = E ;」に対応するエラー規則

$S \to$ error = E ;

$S \to i$ error E ;

```
% a.out
if (x) =b; else b=a;
Error: bad lvar.
T1: (=, b, a)
T2: (if, x, b, T1)
% a.out
if (x) a b; else b=a;
Error: '=' expected.
T1: (=, a, b)
T2: (=, b, a)
T3: (if, x, T1, T2)
%
```

「$A \rightarrow \alpha_1$ `error` α_2」の`error`の位置で構文エラー：

「$A \rightarrow \alpha_1 \cdot$ `error` α_2」を含む状態で検出

1. エラートークンを生成して，

   $A \rightarrow \alpha_1$ `error` $\cdot\, \alpha_2$

   にシフト．
2. α_2に還元できるトークン列を探し，Aへ還元．
   （トークンを読み飛ばしながら，通常の解析）
3. 通常の構文解析を再開．

第5章

意　味　解　析

Cコンパイラの主要な意味解析処理

1. 名前とオブジェクトとの対応づけ
2. 型チェック

オブジェクト:名前をつけられる構成要素

変数，関数，ラベル，構造体，共用体，
構造体/共用体のメンバ，enum 定数，型

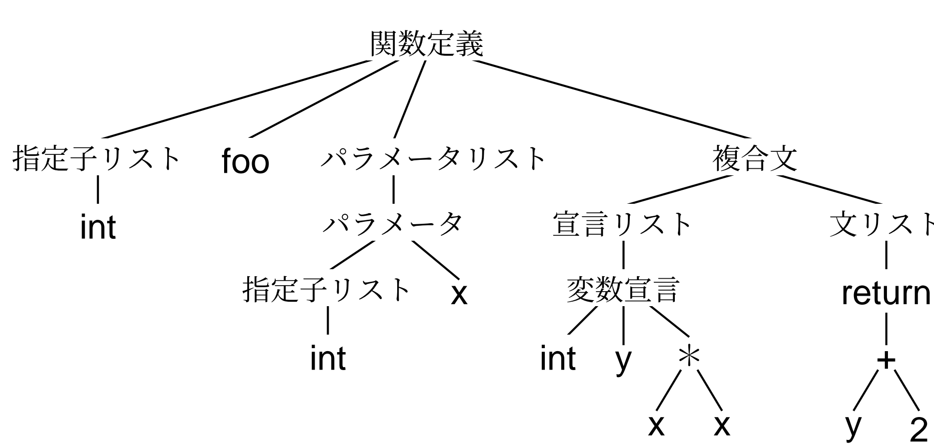
例：

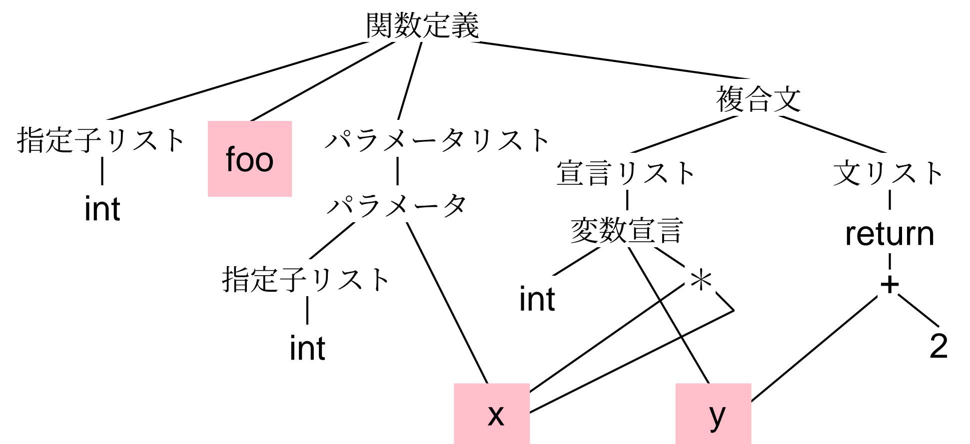
```
int foo(int x) {
    int y = x*x;
    return y+2;
}

double bar(int x) {
    double y = x*x;
    return y+2;
}
```

オブジェクト構造体

オブジェクトを表現する構造体

- オブジェクトの種類(変数，関数，ラベルなど)
- オブジェクトの名前
- 型情報
- 属性(auto，register，staticなど)
- 最適化のための情報
- オブジェクトを割り当てる場所





名前空間とスコープ

名前空間：名前が表すオブジェクトの検索空間

C 言語の主要な名前空間：

- 変数，関数，typedef 名，enum 定数の名前空間
- ラベルの名前空間
- 構造体と共用体の名前空間

例：

```
struct tarao {
    int tarao;
} tarao;

struct tarao x = tarao;
```

スコープ：オブジェクトを参照できる範囲

```
int foo(int x) {
    int y = x*x;
    return y+2;
}
```

```
int foo(int x) {
    int y = x*x;
    return y+2;
}
```

スコープ間の関係

- 二つのスコープには共通部分がない．
- 二つのスコープはまったく一致する．
- 片方のスコープが，もう一方に含まれる．

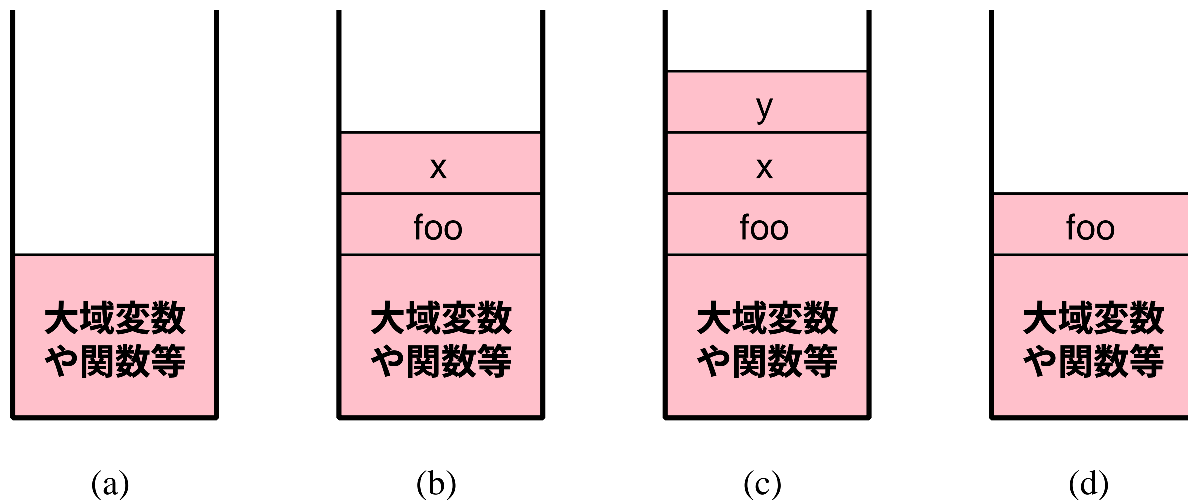
同じ名前空間に属する二つのオブジェクトのスコープが一致するときは，異なる名前をつけなければならない．

例：C 言語の関数パラメータ

名前とオブジェクトの対応づけ

1. まず，xの現れているプログラム中の位置によって，xが属する名前空間を特定する．
2. その名前空間に登録されているオブジェクトのうち，xの現れている位置をスコープに含むものを，内側のスコープを持つものから順に調べる．
3. xを名前とするオブジェクトが見つかれば，最初に見つかったオブジェクトを，xは表す．そのようなオブジェクトがなければ，xの表すオブジェクトは未定義である．

スタックを使った実装



(a) (b) (c) (d)

大域変数，関数，typedef 名などは，ハッシュ表等に登録するほうが検索効率が良い．

前方参照

```
int sigma(int n) {
    int i = 0, s = 0;
    goto test;
loop:
    s += i;
test:
    if (++i <= n) goto loop;
    return s;
}
```

1. “goto test;” で，未定義ラベル test をラベル表に登録.
   {⟨test，未定義，参照済⟩}
2. “loop:” で，ラベル loop を登録.
   {⟨test，未定義，参照済⟩, ⟨loop，定義済，未参照⟩}
3. “test:” で，ラベル test が「定義済」に.
   {⟨test，定義済，参照済⟩, ⟨loop，定義済，未参照⟩}
4. “goto loop;” で，ラベル loop が「参照済」に.
   {⟨test，定義済，参照済⟩, ⟨loop，定義済，参照済⟩}
5. 関数の意味解析終了時に，ラベル表を調べ，
   「未定義」のラベルがあればエラー.「未参照」のラベルがあれば警告.

型チェックと型変換

型表現：型を表すデータ

```
⟨char, (signed|unsigned|-)⟩
⟨int, (signed|unsigned|-), (long|short|-)⟩
⟨float⟩
⟨double, (long|-)⟩
⟨enum, ...⟩
```

多くのプログラミング言語（C 言語を含む）では，構文木を葉から根の方向に向かって調べれば，式の型が決定できる．

例：e_1 * e_2の型チェック

A：e_1の型（型表現）

B：e_2の型（型表現）

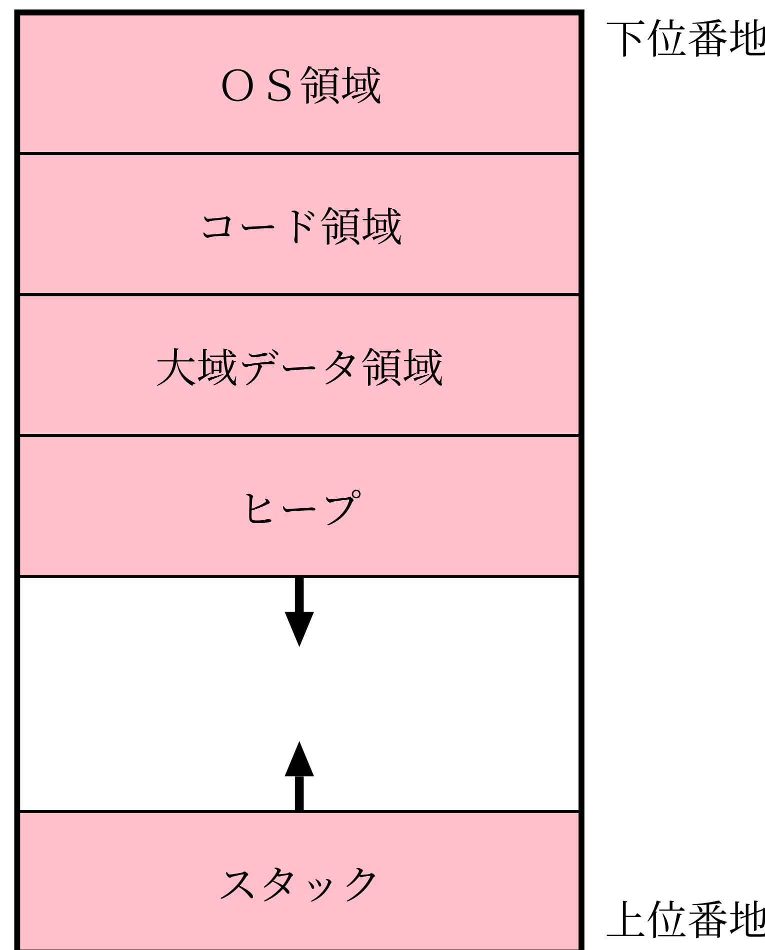
```
if (A==Tdouble) {
  check_double(B); return A;
} else if (B==Tdouble) {
  check_double(A); return B;
} else if (A==Tfloat) {
  check_float(B); return A;
} else if (B==Tfloat) {
  check_float(A); return B;
} else {
  check_int(A); check_int(B); return
Tint; }
```

第6章

コード生成

実行環境のモデル

メモリ領域



- OS領域（場所・サイズは固定）
- ユーザ領域
  - コード領域（ロード時に確定）
  - データ領域
    - 大域データ領域（ロード時に確定）
    - スタック（実行時に拡大・縮小）
    - ヒープ（実行時に拡大）

CPUとレジスタ

中央処理装置 (Central Processing Unit, CPU):
コード領域から機械語命令を一つずつ取り出し実行

算術論理装置 (Arithmetic Logic Unit, ALU):
算術演算や比較演算を実行

Pentium のレジスタ

1. **汎用レジスタ** (general-purpose register):
   オペランドとして使用できる32ビット長レジスタ
   ベースポインタ`ebp`，スタックポインタ`esp`，`eax`，`ebx`，`ecx`，`edx`など
2. **条件フラグ** (condition flag):
   比較命令の結果を格納する1ビット長レジスタ
   ゼロフラグ`zf`，符号フラグ`sf`など
3. **命令ポインタ** (プログラムカウンタ，instruction pointer):
   次の命令の場所を指す32ビット長レジスタ`eip`
   CPUは次の動作を繰り返す:
   1. `eip`の指す位置から命令を一つ取り出す
   2. 次の命令を指すように`eip`を更新する
   3. 取り出した命令を実行する

アセンブリ命令

ラベル宣言

$\langle$ラベル$\rangle$:

アセンブリ命令

$\langle$命令名$\rangle$$\langle$オペランド$_1\rangle$, ... , $\langle$オペランド$_n\rangle$

まとめて

$\langle$ラベル$\rangle$: $\langle$命令名$\rangle$$\langle$オペランド$_1\rangle$, ...

オペランド:

- 汎用レジスタ
- メモリ番地
  - ラベル
  - 相対番地: $n[R]$
- 整数定数

算術命令:

```
add   eax,ebx      ; eaxにebxの値を足す
sub   esp,4        ; espから4を引く
imul  ebx,_x       ; ebxにxの値を掛ける
```

移動命令:

```
mov   eax,4        ; eaxに整数4を格納する
mov   ebp,esp      ; espの値をebpに格納する
```

無条件ジャンプ命令:

```
jmp   ラベル
```

比較命令:

cmp x,y

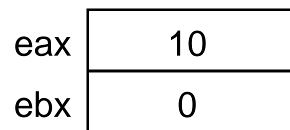
条件付きジャンプ命令:

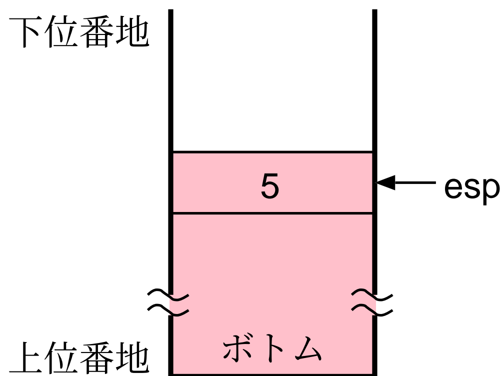
```
j…   ラベル
```

命令	条件	意味
jg	$x > y$	Jump if Greater
jge	$x \geq y$	Jump if Greater or Equal
je	$x = y$	Jump if Equal
jne	$x \neq y$	Jump if Not Equal
jl	$x < y$	Jump if Less
jle	$x \leq y$	Jump if Less or Equal

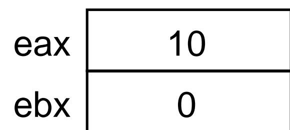
スタック操作命令:

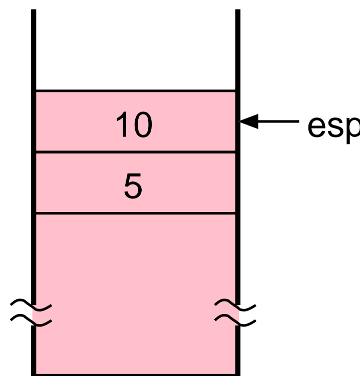
```
push eax
pop  ebx
```



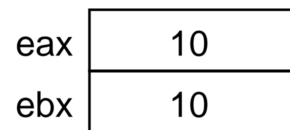


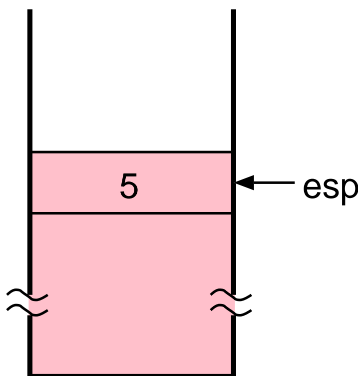
(a) 実行前





(b) push eax 実行後

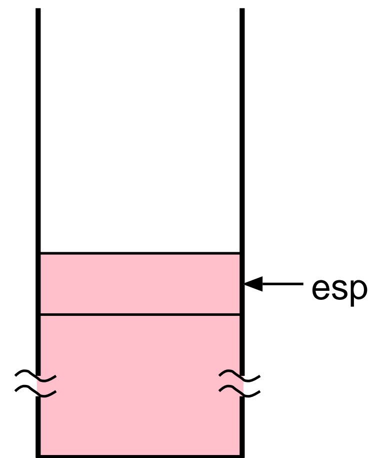




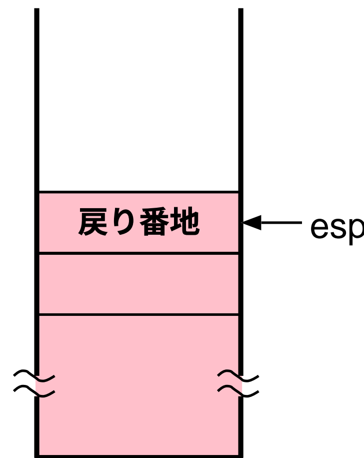
(c) pop ebx 実行後

関数呼出し命令: `call ラベル`

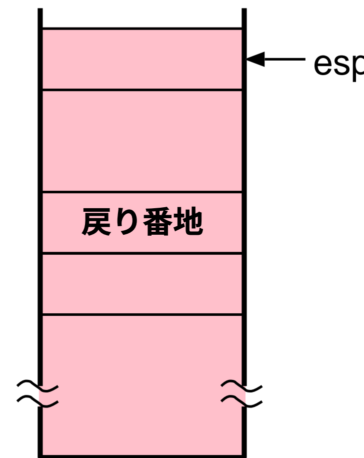
リターン命令: `ret`



(a) 呼出し直前
(e)リターン直後



(b) 呼出し直後
(d) リターン直前

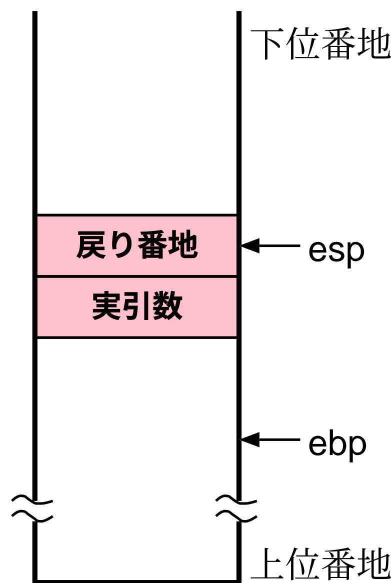


(c) 関数実行中

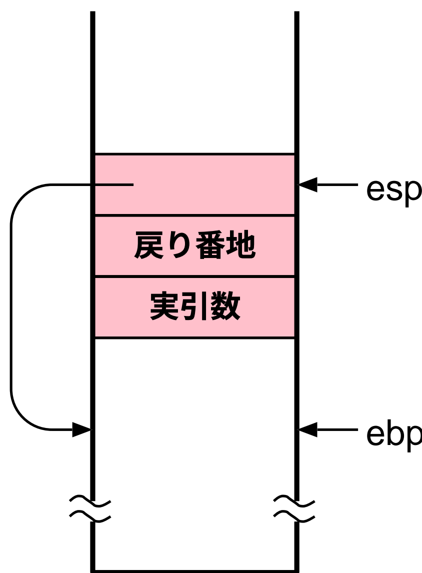
関数呼出し

```
int foo(int x) {
    int y = x*x;
    return y+2;
}
```

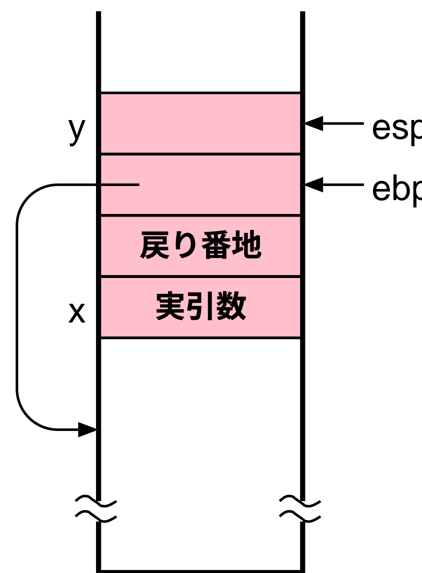
```
 1   _foo: push  ebp
 2         mov   ebp,esp
 3         sub   esp,4
...        ...
 9         mov   esp,ebp
10         pop   ebp
11         ret
```



(a) 呼出し直後



(b) ebp の値の保存



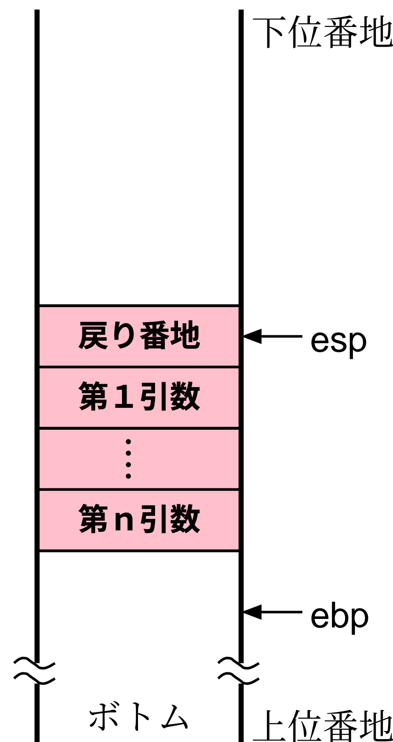
(c) 本体実行中

fooの呼出し

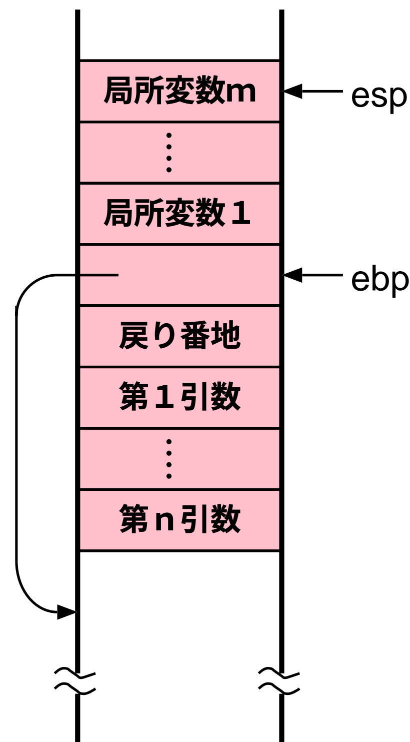
```
push   5
call   _foo
add    esp,4
```

一般の関数 f のコード

```
_f:    push ebp
       mov  ebp,esp
       sub  esp, Nlocal
       本体の実行
Lret:  mov  esp,ebp
       pop  ebp
       ret
```



(a) 呼出し直後



(b) 本体実行中

関数呼出し式 $f(e_1, e_2, \ldots, e_n)$ のコード

```
[e_n を計算し，結果をプッシュする]
...
[e_2 を計算し，結果をプッシュする]
[e_1 を計算し，結果をプッシュする]
call _f
add  esp,n×4
```

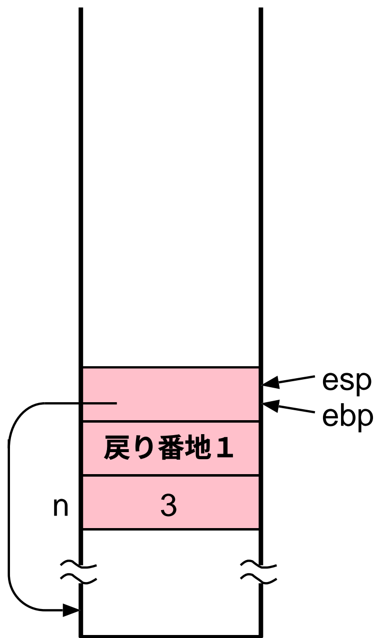
例: f(1,g(2,3),4) のコード

```
push 4          ; fへの第3引数
push 3          ; gへの第2引数
push 2          ; gへの第1引数
call _g
add  esp,8
push eax        ; fへの第2引数
push 1          ; fへの第1引数
call _f
add  esp,12
```

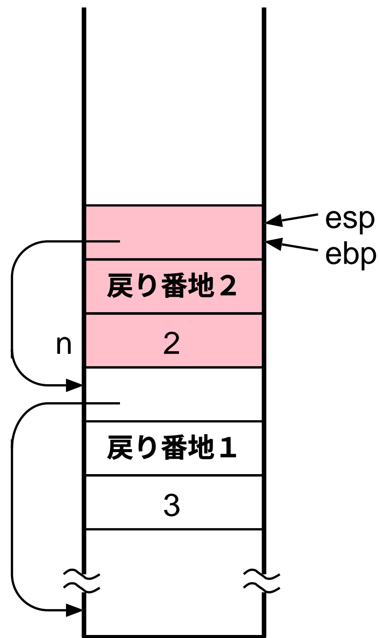
例: 階乗を求める関数`fact`のコードとその実行

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

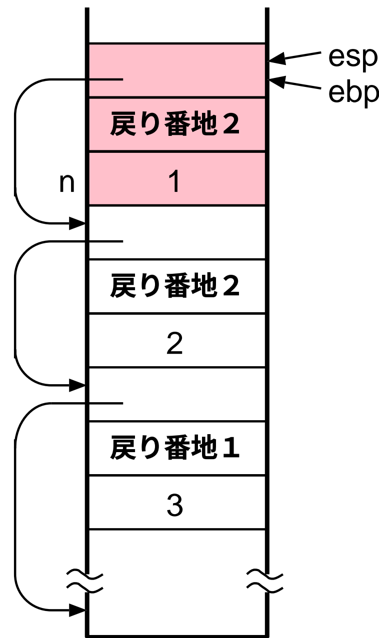
```
_fact: push ebp
       mov  ebp,esp
       cmp  8[ebp],1     ; n=1かどうか
       jne  L2           ; n≠1ならL2へ
       mov  eax,1        ; n=1なら戻り値は1
       jmp  L1
L2:    mov  eax,8[ebp]   ; n−1の計算
       sub  eax,1
       push eax
       call _fact        ; 再帰呼出しで(n−1)!の計算
       add  esp,4
       imul eax,8[ebp]   ; n×(n−1)!の計算
L1:    pop  ebp
       ret
```



(a) fact(3) 実行中

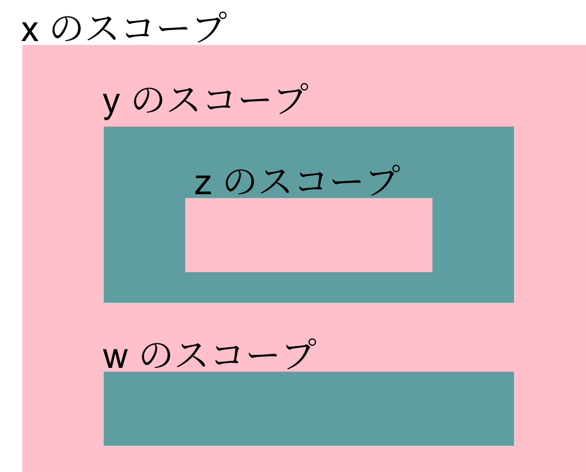


(b) fact(2) 実行中



(c) fact(1) 実行中

局所変数等の割当て



1. 局所変数領域のサイズ $Nlocal$ をできるだけ小さく
   → 同じ位置に複数の局所変数を割り当てる
2. スコープが重なる局所変数は，同じ位置に割り当てない

順序	動作	相対番地	last_alloc	max_alloc
0	初期化		0	0
1	xの割当て	−4	1	1
2	yの割当て	−8	2	2
3	zの割当て	−12	3	3
4	zの解放		2	3
5	yの解放		1	3
6	wの割当て	−8	2	3
7	wの解放		1	3
8	xの解放		0	3

$Nlocal = 3 \times 4 = 12$ バイト使用

レジスタの退避

Pentiumの汎用レジスタは，
　`esp`と`ebp`の他に6個だけ

1. **呼出し後保存レジスタ** (callee-saved register)
   呼び出す側が使用中かもしれない汎用レジスタ．
   呼び出される関数は，使用前に退避
2. **呼出し前保存レジスタ** (caller-saved register)
   上記以外の汎用レジスタ．
   呼び出される関数は，退避せずに使用してよい
   別の関数を呼び出すときは，退避しておく

少なくとも`eax`レジスタは呼出し前保存
→ 全レジスタが呼出し前保存と仮定

文のコード生成

複合文のコード

$\{d_1 \ \cdots \ d_n \ s_1 \ \cdots \ s_m\}$

d_1 の初期化

…

d_n の初期化

s_1 の実行

…

s_m の実行

宣言 `int x = 10;` の初期化

```
mov n[ebp],10
```

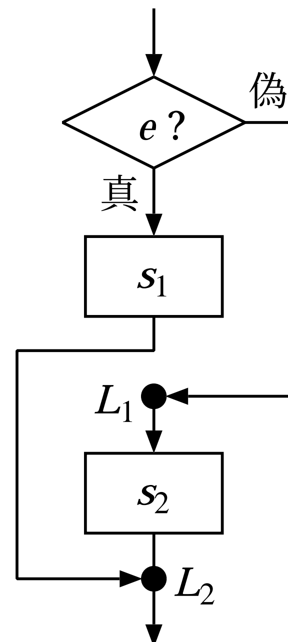
if文のコード

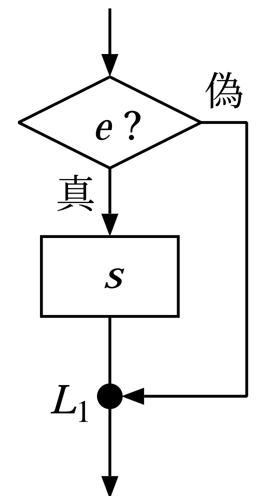
`if` (e) s_1 `else` s_2

　　e を計算し，結果が偽ならば L_1 へジャンプ
　　s_1 の実行
　　`jmp` L_2
L_1: s_2 の実行
L_2:

`if` (e) s

　　e を計算し，結果が偽ならば L_1 へジャンプ
　　s の実行
L_1:

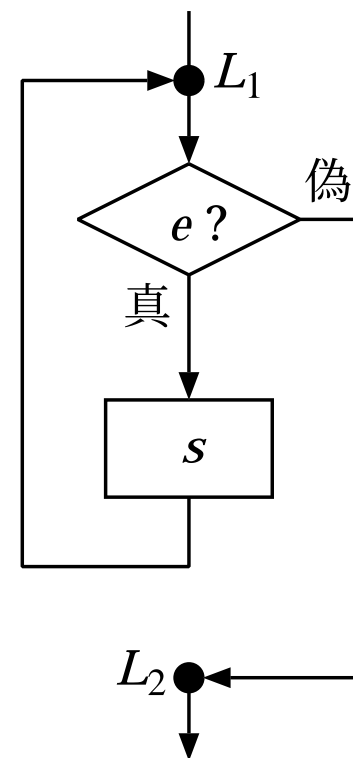




while文のコード

`while (`e`)` s

L_1: | eを計算し，結果が偽ならばL_2へジャンプ |
| sの実行 |
`jmp` L_1
L_2:



break文のコード

`jmp` L_2

continue文のコード

`jmp` L_1

式文と代入式のコード

式文 e;

eの計算（結果は破棄）

変数vへの代入式v = e'

e'の計算（Rに結果）

```
mov   loc(v),R
```

例: `x = y = z+1;`

```
mov   R,loc(z)
add   R,1
mov   loc(y),R
mov   loc(x),R
```

ラベルとジャンプ命令の抑制

```
if (e1) {
  if (e2) s1 else s2
}
```

	e_1を計算し，結果が偽ならL_1へジャンプ
	e_2を計算し，結果が偽ならL_2へジャンプ
	s_1の実行
	jmp L_3
L_2:	s_2の実行
L_3:	
L_1:	

L_1とL_3は融合可能

抑制の方法

1. まったく参照されないラベルを抑制する
2. 直後に来るラベル（もしあれば）を再利用する
3. 直後が無条件ジャンプなら，そのラベルへジャンプする

算術式のコード生成

演算式$e_1 \circ e_2$:

1. e_1の値をあるレジスタRにロードし，
2. Rを第1オペランド，e_2の値を第2オペランドとして，
   '$\circ$'に対応する命令$inst$を実行する．

例: `x+y`

```
mov   R,loc(x)
add   R,loc(y)
```

例: `a*b+y`

```
mov   R,loc(a)
imul  R,loc(b)
add   R,loc(y)
```

例: `a*b+x*y`

```
mov   R1,loc(a)
imul  R1,loc(b)
mov   R2,loc(x)
imul  R2,loc(y)
add   R1,R2
```

例: R_1だけを使って`a*b+x*y`を計算

```
mov   R1,loc(a)
imul  R1,loc(b)
mov   temp,R1
mov   R1,loc(x)
imul  R1,loc(y)
add   R1,temp
```

可換演算と非可換演算

例: R_1だけを使って`a*b-x*y`を計算

```
mov   R1,loc(a)
imul  R1,loc(b)
mov   temp,R1
mov   R1,loc(x)
imul  R1,loc(y)
sub   R1,temp
```

ではなく

```
mov   R1,loc(x)
imul  R1,loc(y)
mov   temp,R1
mov   R1,loc(a)
imul  R1,loc(b)
sub   R1,temp
```

利用可能なレジスタ数が N,
e_1 と e_2 の計算に必要なレジスタ数も N のとき,
e_1 - e_2 は

e_1 の計算（R_1 に結果）

mov　$temp, R_1$

e_2 の計算（R_2 に結果）

mov　$R_3, temp$
sub　R_3, R_2

よりも

e_2 の計算（R_2 に結果）

mov　$temp, R_2$

e_1 の計算（R_1 に結果）

sub　$R_1, temp$

右辺の計算を先に！

算術式のコード生成アルゴリズム

N: 式の計算に利用できるレジスタ数（$N \geq 2$）

$\rho(e)$: eの計算に必要なレジスタ数

$N \geq \rho(e) \geq 0$
$\rho(e) = 0 \leftrightarrow e$は変数か定数

アルゴリズム:

算術式$e_1 \circ e_2$に対して，
まず$\rho(e_1)$と$\rho(e_2)$を求め
各e_iのコード生成開始時点で，
$\rho(e_i)$個以上のレジスタを空ける

RSL (Right-Save-Left) 型

$\rho(e_1) = \rho(e_2) = N$のとき

```
e2の計算（R2に結果）
mov    temp, R2
e1の計算（R1に結果）
inst   R1, temp
```

RL (Right-Left) 型

$\rho(e_1) < N$のとき

```
e2の計算（R2に結果）
e1の計算（R1に結果）
inst   R1, R2
```

R (Right) 型

e_1が変数/定数vで，演算が可換のとき

```
e2の計算（R2に結果）
inst   R2, loc(v)
```

LR (Left-Right) 型

$\rho(e_2) < N$のとき

```
e1の計算（R1に結果）
e2の計算（R2に結果）
inst   R1, R2
```

L (Left) 型

e_2が変数/定数vのとき

```
e1の計算（R1に結果）
inst   R1, loc(v)
```

	$\rho(e_2) = N$	その他	$\rho(e_2) = 0$
$\rho(e_1) = N$	RSL	LR	L
$N > \rho(e_1) > 0$	RL	RL/LR	L
$\rho(e_1) = 0$（非可換）	RL	RL	L
$\rho(e_1) = 0$（可換）	R	R	L

RSL型コード生成ルーチン

```
reg emit_RSL_code(char *inst, tree e1, tree e2) {
    reg R1, R2 = emit_expr(e2);
    loc temp = allocate_temp();
    emit("mov", temp, R2);
    release_register(R2);
    R1 = emit_expr(e1);
    emit(inst, R1, temp);
    release_temp(temp);
    return R1;
}
```

RL型コード生成ルーチン

```
reg emit_RL_code(char *inst, tree e1, tree e2) {
    reg R2 = emit_expr(e2);
    reg R1 = emit_expr(e1);
    emit(inst, R1, R2);
    release_register(R2);
    return R1;
}
```

使用レジスタ数の計算

算術式 $e_1 \circ e_2$ の使用レジスタ数

$\rho(e_2)$ の値	N	その他	0
$\rho(e_1) = N$	N	N	N
その他	N	RL 型: $\max(\rho(e_1)+1, \rho(e_2))$ LR 型: $\max(\rho(e_1), \rho(e_2)+1)$	$\rho(e_1)$
$\rho(e_1) = 0$（非可換）	N	$\max(2, \rho(e_2))$	1
$\rho(e_1) = 0$（可換）	N	$\rho(e_2)$	1

$\rho(e_1) = N$ または $\rho(e_2) = N$ なら $\rho(e_1 \circ e_2) = N$

RL 型なら $\rho(e_1 \circ e_2) = \max(\rho(e_1)+1, \rho(e_2))$

e_2 の計算（R_2 に結果）　$\rho(e_2)$ 個使用

e_1 の計算（R_1 に結果）　$\rho(e_1)+1$ 個使用

$inst \quad R_1, R_2$

例: `a*b-x*y`

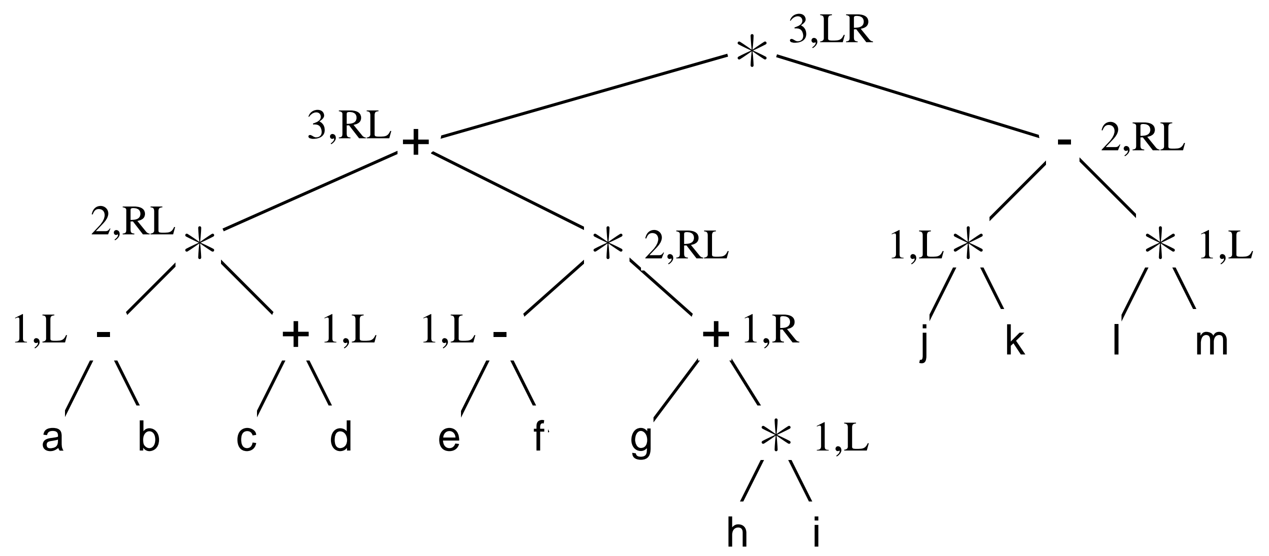
$\rho(\texttt{a}) = \rho(\texttt{b}) = 0$ より $\rho(\texttt{a*b}) = 1$，同様に $\rho(\texttt{x*y}) = 1$

したがって，`a*b-x*y` は RL 型

```
mov   R,loc(x)
imul  R,loc(y)
mov   R',loc(a)
imul  R',loc(b)
sub   R',R
```

例: $N = 3$のとき,

((a-b)*(c+d)+(e-f)*(g+h*i))*(j*k-l*m)




```
mov  eax,h      mov  eax,c      mov  eax,l
imul eax,i      add  eax,d      imul eax,m
add  eax,g      mov  ecx,a      mov  ebx,j
mov  ebx,e      sub  ecx,b      imul ebx,k
sub  ebx,f      imul ecx,eax    sub  ebx,eax
imul ebx,eax    add  ecx,ebx    imul ecx,ebx
```

関数呼出し式の使用レジスタ数

$f(e_1, \ldots, e_n)$

各e_iの値をスタックにプッシュ

```
ei の計算（R に結果）
push R
```

$\rho(e_i)$ 個のレジスタを使用

すべての実引数をスタックにプッシュ

使用レジスタ数は$\max(m, \rho(e_1), \ldots, \rho(e_n))$

f の使用レジスタ数は，分からない．$\rightarrow$ 最大の N 個と仮定

$$\rho(f(e_1, \ldots, e_n)) = \max(N, \rho(e_1), \ldots, \rho(e_n)) = N$$

例: `f()-x*y`

$\rho(\texttt{f()}) = N$，$\rho(\texttt{x*y}) = 1$だからLR型

```
call _f
mov  R,loc(x)
imul R,loc(y)
sub  eax,R
```

条件ジャンプのコード生成

「eを計算し，結果が偽ならLへジャンプ」

```
[eの計算（Rに結果）]
cmp   R,0            ; 0と比較
je    L              ; 等しければジャンプ
```

「真なら」の場合は，`je`を`jne`で置き換える．

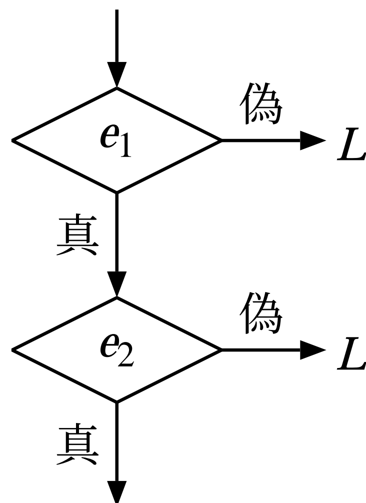
例: `if (f()) x = 10;`のコード

```
        call _f
        cmp  eax,0
        je   L
        mov  loc(x),10
    L:
```

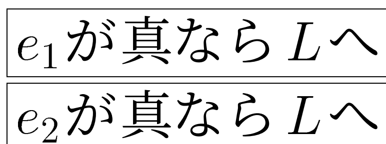
「e_1 >= e_2が真ならLへ」（e_1 >= e_2がRL型のとき）

```
[e2の計算（R2に結果）]
[e1の計算（R1に結果）]
cmp   R1,R2
jge   L
```

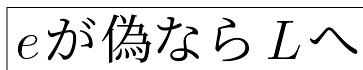
「e_1 && e_2が偽なら Lへ」



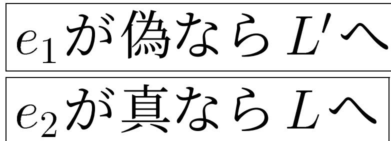
「e_1||e_2が真なら Lへ」



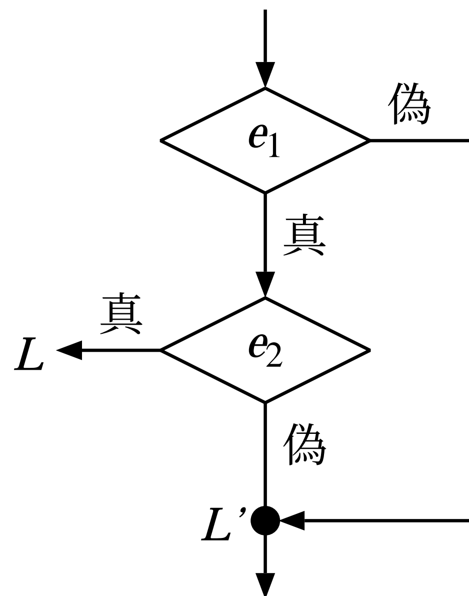
「!eが真なら Lへ」



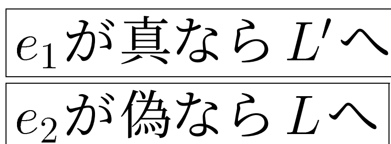
「e_1 && e_2が真なら Lへ」



L':

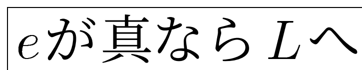


「e_1||e_2が偽なら Lへ」



L':

「!eが偽なら Lへ」



例: 「(a && b) || !(c || d)が真ならLへ」

```
(a && b) || !(c || d)が真ならLへ
  a && bが真ならLへ
    aが偽ならL1へ          cmp loc(a),0
                           je L1
    bが真ならLへ           cmp loc(b),0
                           jne L
    L1:
  !(c || d)が真ならLへ
    c || dが偽ならLへ
      cが真ならL2へ        cmp loc(c),0
                           jne L2
      dが偽ならLへ         cmp loc(d),0
                           je L
      L2:
```

戻り値の計算コード

`return` e;

eの計算（Rに結果）

```
mov   eax,R
jmp   Lret
```

モード指定で移動命令を削除

eaxモード: 結果をできるだけ`eax`に

no-eaxモード: 結果をできるだけ`eax`以外に

freeモード: どのレジスタでもよい

例: `return a*b-x*y;`のコード

`a*b-x*y`全体はeaxモード，`a*b-x*y`はRL型

`x*y`の計算（R_2に結果）	no-eaxモード
`a*b`の計算（R_1に結果）	eaxモード

$inst$ R_1,R_2

生成結果（移動命令削除）

```
mov   ebx,loc(x)
imul  ebx,loc(y)
mov   eax,loc(a)
imul  eax,loc(b)
sub   eax,ebx
jmp   Lret
```

関数コードの生成例

```
_fact: push ebp
       mov  ebp,esp
       [本体 { if ... }の実行]
       [if文if(n==1) ... else ...の実行]
       [n==1が偽ならL2へ]
       cmp  8[ebp],1
       jne  L2
       [return 1;の実行]
       mov  eax,1
       jmp  L1
L2:    [return n*fact(n-1);の実行]
       [n*fact(n-1)の計算]
       [fact(n-1)の計算]
       [n-1を計算し，結果をプッシュ]
       mov  eax,8[ebp]
       sub  eax,1
       push eax
       call _fact
       add  esp,4
       imul eax,8[ebp]
L1:    pop  ebp
       ret
```

その他のトピック

1. 局所関数
    - 定義を包含する他の局所関数・最上位関数のフレーム参照
    - 静的リンクまたはディスプレイを使用
2. ３オペランド命令と１オペランド命令
    - ２オペランド命令と同様に考察
3. 呼出し後保存レジスタ
    - 呼出し前保存レジスタを優先的に割り当てる
    - 関数呼出しを含む式は，呼出し後保存レジスタを使って高速化