1 コンパイラ

コンパイラ (compiler) とは:
compile: (他動詞) まとめる，編集する．

1. 狭義には: あるプログラム言語で書かれたソース・ファイルを入力して，これに対応する実行可能なプログラムを出力するソフトウエア．

例 1

```
#include "stdio.h"
main(){printf("Hallo World!\n");}
```

という `C` のソースプログラムが，`example.c` というファイルとしてある時，`C` のコンパイラを例えば，

```
> cc example.c
```

としてこのソースプログラム名をパラメタとして与えて呼ぶと，これに対応する，`a.out` というような名前のついた実行可能なプログラム(目的プログラム)が出力される．このプログラムを実行すると `‘‘Hallo World!’’` という文字列が画面に出力される．

2. 1. の補足: 多くのコンパイラはソースファイルがプログラム言語の文法にそって書かれていなかったり，データタイプに整合性がないような場合，エラーメッセージを出したり，ソースコードを適当に解釈しなおして作業を先に進めたりすることを試みる，というような機能があり，これがコンパイラの使用いやすさを決定する大きなファクターとなっていることが多い．

例 2

```
#include "studio.h"
main(){printf("Hallo World!\n");}
```

というソースプログラムをコンパイラにかけると:

```
> cc example.c
example.c:1:  studio.h:  No such file or directory
>
```

というエラーメッセージが出て目的プログラムは作成されない．

3. より一般的には，特定の決まった書式で書かれた(テキスト)ファイルを読み込んで(それに相当する)ファイルを出力するようなソフトウエアをすべてコンパイラと呼ぶことにする．

例 3

```
$Y=X_{a,b}$ とする．
```

という内容のファイルが *test.tex* としてあるとするとき

```
> tex test.tex
```

として $\TeX$ のコンパイラを呼ぶと，$test.dvi$ というファイルが作られる，このファイル名をパラメタとして適当なプリンタドライバーを起動すると，

$Y = X_{a,b}$ とする．

が印刷される．ちなみにこのテキスト自体も $\TeX$ を用いて書かれている．

4. たとえば，C のプログラムを入力すると，対応するアセンブリ言語のプログラムを出力するもの，modula のプログラムを C のプログラムに翻訳するプログラムなども，3. の意味でのコンパイラと考えられる．

本講義の理想的な目標としては，受講者が自分で高性能なコンパイラが作れるようになることであるが，これには，一学期で講義しきれないような雑多なテクニックを知る必要があり，実際にはむずかしいであろう．ただし，プログラミングの経験があれば，講義で得た知識を活用してコンパイラのトイプログラムくらいは書けるようになるのではないかと思う．コンパイラの勉強をすることの，それ以外のメリットとして考えられることは:

1. 大きくて複雑なプログラムを造るときの方法論を学ぶことができる: コンパイラはコンピュータの歴史の初期から，そのプログラミングの仕方について沢山の研究がなされているので，大規模なプログラムの構成法の例として十分に洗練されたものとなっている．

2. コンパイラを理解していることは，プログラムを書くときにも助けになる．これの例としては，コンパイラの仕組みを頭におくとコンパイラの出したエラーメッセージから，その原因となっているプログラムのエラーが何であるかを推測することが容易になるであろう．

3. コンパイラを実現するために使われている手法はコンパイラ以外のソフトウエアを作るときにも，部品として有用であるものが多い．

参考文献

コンパイラ: 原理・技法・ツール; A.V. エイホ，R. セシィ，J.D. ウルマン共著，原田賢一 訳，サイエンス社，1990. (原著: **Compilers – Principles, Techniques, and Tools**; Alfred V.Aho, Ravi Sethi, Jeffery D.Ullman, Addison-Wesley Publishing Company, 1986).

コンパイラの理論と実現; 疋田 輝雄，石畑 清 共著，共立出版，1988.

2 コンパイラの大まかな構造

コンパイラは大まかに言うと以下のような部分に分割された構造になっていることが多い:

原始プログラム
(source file)

↓

字句解析
(lexical analysis)

↓

構文解析
(syntax analysis)

↓

意味解析
(semantics analysis)

↓

中間コード作成 (generation of intermediate codes)

↓

コードの最適化
(code optimization)

↓

コードの生成
(code generation)

↓

目的コード
(target(object) code)

上の図での

$$\begin{array}{c}\boxed{\quad\quad\quad A\quad\quad\quad}\\ \downarrow \\ \boxed{\quad\quad\quad B\quad\quad\quad}\end{array}$$

は，(a) B は，A の出力を一つ一つひろって，それに対して処理をほどこしてゆくようになっている，という場合と，(b) A が入力に対して処理をほどこして，その結果を中間ファイルとしてディスクにはき出し，B はその中間ファイルを頭から読みなおして処理を続行する場合とあがある．(a) の場合は，A は B の subroutine として実現され，(b) の場合は，A が呼びだされた後で B が呼びだされる形をとる．中間ファイルを作る回数により，**1 パス・コンパイラ**，**2 パス・コンパイラ**，**多パス・コンパイラ**などと呼ぶ．

上のようなコンパイラの構造はもっと大まかに，プログラムの解析部，とコードの生成部というふうに二分して見ることもできる．入力出力の処理に着目して，フロントエンド，バックエンドというような呼びかたで前半後半を分類することもある．

コンパイラが上のようなブロック構造になっていることの利点は，コンパイラを製作するときに見通しがよくなる，ということ以外にも，たとえば，良い設計をしておくと，フロントエンドの一部とバックエンドに変更を加えるだけで，異なるマシンに移植をすることが容易にできるようになる，といったことが可能となる．これの良い例としては gcc がある（演習: gcc で移植を容易に実現させるためにどんな設計がとられているかを調べてみなさい）

コンパイラの生成するコードはそのコンパイラの稼働している計算機用のものである必要はない．たとえば，プログラミングのツールの完備している UNIX 上で PC 用のプログラムを開発する，など．出力コードを走らせるための目的マシン (target machine) がコンパイラの動いているものと同じでない，あるいは目的マシンの切りかえができるようになっているようなコンパイラをクロス・コンパイラ (cross-compiler) とよぶ．

第 1節での意味で，広義にはコンパイラと考えることのできるようなコンパイラの “仲間” に，マクロ・プロセッサやプレ・プロセッサなどがある．

マクロ・プロセッサは，ファイルの中に書き込まれたマクロ定義を読み，これにしたがって，ファイルの中のマクロの呼び出しを置き換えてゆく（これをマクロを展開するという）ようなプログラムである．C のコンパイラには，通常はマクロプロセッサがコンパイラに組みこまれていて，たとえば，

```
#define baka() kaba=chindonya()
```

というマクロ定義がソース・プログラムにあると，これ以降の部分で

```
...  baka() ...
```

が出てきたときには，これはマクロ・プロセッサによって

```
...  kaba()=chindonya() ...
```

に置き換えられた後に処理される．

プレ・プロセッサは入力ファイルを他の (多くの場合にはよく知られた言語の) ファイルに翻訳するプログラムで，このようなものを作ることで，新しい言語のコンパイラを比較的簡単に実現することができる．最終的には本物のコンパイラを製作するつもりのときでも，まずプレ・プロセッサを作ることでプロトタイプを実現してみる，という方針が有効なことも多い．
たとえば，Pascal のプログラムを C のプログラムに翻訳するプログラムは C のコンパイラのプレ・プロセッサとして用いることにより，全体として，Pascal のコンパイラを実現することができる．

3 字句解析

字句解析は構文解析のサブルーチンとして実現されることが多い

↓

字句解析 (lexical analysis)

トークン↓ ↑トークンの要求　　| 記号表 |

構文解析 (syntax analysis)

字句解析は記号表を拡張/参照しながら行われ，構文解析でも記号表がさらに拡張/参照される．
字句解析部は入力をトークンに分解して，このデータを記号表に記入する．さらにこのトークンが構文解析部に渡される．これ以外にも字句解析部は入力まわりの副次的な仕事をいくつかすることが多い．たとえば，

1. 入力コードからコメントを取り去ったり言語によっては機能を持たないような空白文字などを取り去る．
2. 入力の行数や入力行の内容を記録して，エラーメッセージなどを出力するときの必要にそなえる．
3. マクロの展開がすでにここで行われる場合もある．

上のような仕事のために字句解析がさらに部分に分割されていることもある．
字句解析と構文解析を二つのユニットとして分けることの利点としては，以下のような点が考えられる:

1. デザインが簡単明瞭になる．
2. 効率を上げるための改良をほどこしやすくなる: たとえば，ファイルの読み込みのためにバッファを設定して処理測度を上げるというようなことの実

現を字句解析部で行うことができる．
3. コンパイラの移植性が上げられる: 入力の処理に関する変更を字句解析部で局所的に実行することができるようになる．

各々のトークン対し具体的な語彙とそれをあらわすパターンが対応づけされる: できる．

トークン	$\boxed{\text{if}}$	$\boxed{\text{関係}}$	$\boxed{\text{数}}$	$\boxed{\text{記号列}}$	$\boxed{\text{識別子}}$
語彙 (lexem)	*if*	$<, >=, \ldots$	75, 3.14, ...	"Hallo world"	**pi, x2, count,** ...
パターン	*if*	$<, \ldots$	数表現	"と" に囲まれた文字列	文字とそれに続く文字か数字

パターンは後出の正規表現を用いて与えることができる．上の例では $\boxed{\text{数}}$ や $\boxed{\text{識別子}}$ といったトークンはそれだけでは必要な情報をすべて含まないので，このトークンを形容する属性 (attributes) を附帯させる必要がある．

例 4 E=M*C**2 は字句解析により，例えば

$\langle id,$ E に対応する記号表の記載へのポインタ $\rangle$
$\langle$ 代入演算子 $\rangle$
$\langle id,$ M に対応する記号表の記載へのポインタ $\rangle$
$\langle$ 積演算子 $\rangle$
$\langle id,$ C に対応する記号表の記載へのポインタ $\rangle$
$\langle$ 巾演算子 $\rangle$
$\langle$ 数，$2\rangle$

というように，トークンと *attributes* の組に翻訳されて，構文解析部に送られることになる．

字句解析部でエラーが発生したときの，それからの回復についても考慮しなくてはいけないが，ここで検出できるようなエラーはあまり多くない，例えば C のプログラムで，

```
fi(a==f(x))
```

とあっても，`fi` は関数の識別子であるかもしれないので，ここではまだエラーかどうかは判定できない．ただしパターンのどれにもマッチしない記号列が出てくることはあり得る．この際の対処法としては，次にパターンにマッチするところが出てくるまで入力を読みとばす "パニック・モード" や，記号を自動的につけ加えたり，消去したり，転置したりする，というような方法が考えられる．二番目の方法での判定条件としては，一番少ない変更でエラーの回復ができるものを採用するという方針などが考えられる．

記号列に関する記法を復習しておく．L と M を記号列の集合とする．このとき，

$$L \cup M = \{s \mid s \text{ は } L \text{ の元かまたは } M \text{ の元}\}$$
$$LM = \{st \mid s \in L\, t \in M\}$$

とする．ここに，st は記号列 s に記号列 t を繋げたものをあらわす．また LL を L^2 と書き，$\underbrace{L \cdots L}_{\text{n 個}}$ を L^n と書く．ここで，

$$L^* = \bigcup_{i=0}^{\infty} L^i = \{t \mid t \text{ は } L \text{ の元を幾つか（}0\text{ 個も含む）つなげたもの}\}$$

とする．ε で空の文字列をあらわすことにすると，$\varepsilon \subset L^*$ である．また

$$L^+ \bigcup_{i=1}^{\infty} L^i$$

とする．$L^+ = L^* \setminus \{\varepsilon\}$ である．

例 5 $L = \{A, B, \ldots, Z, a, b, \ldots z\}$ $D = \{0, \ldots, 9\}$ とする．このとき，$L \cup D$ はアルファベットと数字の集合となり，LD はアルファベットに数字が続いている文字列の全体の集合となる．

以下では a を記号とするとき，a と長さが 長さが 1 で一番目の記号が a であるような記号列とを同一視することにする．したがって，Σ を記号の集合とするとき，Σ は長さが 1 の Σ 上の記号列の集合であるとも見なす．
Σ を記号の集合とするとき，Σ 上の正規表現を以下により帰納的に定義する:

1. ε は正規表現;
2. $a \in \Sigma$ なら，a は正規表現;
3. r と s が正規表現なら，$(r|s)$, $(r)(s)$, $(r)^*$ も正規表現である;
4. 正規表現は以上の規則の繰返しの適用で得られるもののみとする．

ただし括弧は明らかな場合には適当に省略することにする．正規表現 r にマッチする記号列の全体の集合 $I(r)$ は Σ^* の部分集合として，次のようにして与えられる:

1. $I(\varepsilon) = \emptyset$;
2. $a \in \Sigma$ なら，$I(a) = \{a\}$;

r と s を正規表現として，

3. $I((r|s)) = I(r) \cup I(s)$;
4. $I((r)(s)) = I(r)I(s)$;
5. $I((r)^*) = I(r)^*$.

正規表現は次のような形の定義式の連立表現に分割することができる．

$$\begin{gathered} d_1 \to r_1 \\ d_2 \to r_2 \\ \cdot \\ \cdot \\ \cdot \\ d_n \to r_n \end{gathered}$$

ここに d_i はそれぞれ異なる（記号列の集合に対応する）名称で，r_i は $\Sigma \cup \{d_1, \ldots, d_{i-1}\}$ 上の正規表現である．
これは後に述べるバックス・ナウアー表現 (BNF) の特殊なものとなっている．一般的なバックス・ナウアー表現では $d_i \to r_i$ の r_i の中に d_i が入っていてもよいとする．

例 **6** 次の連立表現により，C の識別子にマッチする正規表現を定義することができる．

$$\begin{aligned} \text{文字} &\to A|B|\cdots|Z|a|b|\cdots|z| \\ \text{数字} &\to 0|1|\cdots|9| \\ \text{識別子} &\to \text{文字 (数字 | 文字)}^* \end{aligned}$$

次のような略号が用いられることが多い:

1. “r を一つ以上ならべたもの” は正規表現 rr^* によってあらわされるが，これを r^+ と略記する．
2. “r をゼロ個あるいは一個ならべたもの” は正規表現 $(r|\varepsilon)$ であらわせるが，これを $r?$ と略記する．
3. $A|B|\cdots|Z$ を $[A{-}Z]$ ，$A|B|\cdots|Z|a|b|\cdots|z|$ を $[A{-}Za{-}z]$, $0|1|\cdots|9|$ を $[0{-}9]$ etc. と略記する．

特に例 6の正規表現は

$$\text{識別子} \to [A{-}Za{-}z][A{-}za{-}z0{-}9]^*$$

と書くことができる．
正規表現の認識は遷移図式で表現することができる．
たとえばトークン $\boxed{\text{大小関係}}$ が

記号列	属性	
$<$	LT	(less than)
$<=$	LE	(less or equal to)
$<>$	NE	(not equal to)
$=$	EQ	(equal)
$>$	GT	(greater than)
$>=$	GE	(greater or equal to)

として与えられているとき，これを認識するための遷移図は

$\xrightarrow{start} \bigcirc \quad \xrightarrow{<} \bigcirc \quad \xrightarrow{=} \circledcirc$ $\boxed{\text{大小関係}}$(LE) を返す

$\xrightarrow{>} \circledcirc$ $\boxed{\text{大小関係}}$(NE) を返す

$\xrightarrow{\text{その他}} \circledcirc\ *$ $\boxed{\text{大小関係}}$(LT) を返す

$\xrightarrow{=} \circledcirc$ $\boxed{\text{大小関係}}$(EQ) を返す

$\xrightarrow{>} \bigcirc \quad \xrightarrow{=} \circledcirc$ $\boxed{\text{大小関係}}$(GE) を返す

$\xrightarrow{\text{その他}} \circledcirc\ *$

と描ける．ここで $\circledcirc$ は結果を，$*$ は読み込んだ記号を読み戻す（unread する）ことが必要な場合を示す．

前の例 6での識別子の認識のための遷移図式は，

文字
あるいは
数字

$\xrightarrow{start} \bigcirc \quad \xrightarrow{\text{文字}} \bigcirc \quad \xrightarrow{\text{その他}} \circledcirc\ *$

となる．ここで，$\circledcirc\ *$ での動作は，対応するトークンがすでに記号表に登録されているかを調べて，もしあればその値を返し，そうでなければ，記号表にこの識別子を加えた後その結果を返す，というようなものとなる．

遷移図式は対応する場合わけプログラムに容易に書きなおせる（演習: 上の遷移図に対応する C プログラムがどんなものになるか考えてみよ .)

遷移図式からただちに対応する非決定性有限オートマトンを求められるが，非決定性有限オートマトンから決定性有限オートマトンを自動的に生成するアルゴリズムが存在するから，これを用いると，与えられた正規表現を認識するプログラムの自動生成が可能となる．実際 UNIX の lex はそのような tools の一つとなっている．

4　構文解析

構文解析はパーシング（parsing）とも呼ばれ，構文解析部はパーサー（parser）とも呼ばれる．

まず計算機言語のほとんどすべてを記述することが可能な BNF（Backus-Nauer Form）について復習する．BNF で記述できる文法は正規表現であらわされる文法を真に含む文脈自由文法というクラスとなっている．

例 7 $\quad if\ \boxed{expression}\ then\ \boxed{statement_1}\ else\ \boxed{statement_2}$
は正規表現ではうまくあらわせない．これはしかし，

$$statement \to if\ expression\ then\ statement\ else\ statement$$

という形に表記できる．

正規表現の表記では，$d \to r$ という形の定義式では r には d は含まれないのだった．この制限を除いた連立表現はバックス・ナウアー表現 (BNF) とよばれ，BNF であらわされる文法は文脈自由文法とよばれる．つまり，BNF は

$$\begin{array}{c} d_1 \to r_1 \\ d_2 \to r_2 \\ \cdot \\ \cdot \\ \cdot \\ d_n \to r_n \end{array}$$

という形をしている．ここに d_i はそれぞれ異なる（記号列の集合に対応する）名称で，r_i は $\Sigma \cup \{d_1, \ldots, d_{i-1}, d_i\}$ 上の正規表現である．
正規表現にマッチする記号列の集合（文法）は BNF によっても表現できるから，最初から，BNF のみを用いるようにした方が一般的で議論もすっきりすると思うかもしれない．しかし，

1. 正規表現は文脈依存文法の表現に比べてコンパクトで分かりやすく，
2. 正規表現で扱える範囲のものは，それによった方が効率良く扱える．

BNF G で与えられた文法にマッチする表現 α^* はその導出により与えられる．ここに，G における導出は

$$\alpha_1 \Rightarrow \aleph_2 \Rightarrow \alpha_3 \Rightarrow \cdots \Rightarrow \alpha^*$$

という形のものとする．ここに，α_1 は一つの非終端記号，α^* は終端記号のみからなる記号列で，各 $\alpha_i \Rightarrow \alpha_{i+1}$ は，G のある元 $A \to \gamma$ に対し，γ にマッチする記号列 γ^* があって，α_i と α_{i+1} がそれぞれ $\alpha A \beta, \alpha\gamma^*\beta$ という形になっているものとする．
BNF G が与えられたとき，G により規定される文法 $\mathcal{G} \subseteq \Sigma^*$ は

$$\mathcal{G} = \{\alpha \in \Sigma^* \mid \alpha \text{ の } G \text{ における導出が存在する}\}$$

として定義する．
BNF による導出に対応する構文木を考えることができる．導出が線型なものだったのに対し，構文木は木の形に分岐するものとなる．

例 8 算術表現（の一部）が以下のような *BNF* で与えられているとする:

$$\begin{aligned} &E \to EAE|(E)| - E|[a-z] \\ &A \to +| - | * |/ \end{aligned}$$

このとき，`((a+b)*c)/d` の導出は

$$E \Rightarrow EAE \Rightarrow E/E \Rightarrow E/d \Rightarrow \cdots$$

と与えられるが，これに対応して，

E

$E \quad A \quad E$

$/ \quad$ d

という木の構造が得られる．

$(\quad E \quad)$

$E \quad A \quad E$

$* \quad$ c

BNF によって与えられた文法は構文木の構造に関する曖昧さを残してしまうことがありえる．例えば上の例での BNF による文法には `a+b*c` という表現が含まれるが，これに対応して，

E

$E \quad A \quad E$

$* \quad$ c

$E \quad A \quad E$

a $\quad$ + $\quad$ b

E

$E \quad A \quad E$

a $\quad$ +

$E \quad A \quad E$

$*$

の両方の構造木が得られてしまう．このような曖昧さを回避する方法としては，

1. 括弧をつけた表現のみを許すように文法を書きなおす
2. 演算の hierarchy を導入して default の括弧のつき方を決めてしまう

などが考えられる．

構文解析は，構造木を根から下方にむかってトップダウンに決定してゆく方法と，構造木の葉から上方向にボトムアップに決定してゆく方法がある．トップダウン解析は C などの再帰的関数を許す言語で実現することが容易である．これに対し，ボトムアップ解析は人手で実現するのは難しいが，ボトムアップ解析の一つである Knuth による LR 解析やその改良である SLR, LALR などは，文法構造解析プログラムの自動生成に応用できる．UNIX 上の構文解析プログラムの自動作成システム yacc は LALR を用いている．

5 記号表の管理

字句解析の際に読み込まれた，変数名，関数名，手続き名などの識別子は記号表に登録され必要に応じて参照される．特に記号表へ参照は頻繁に行われるので，この部分の速さがコンパイラーの効率に大きく作用することになる．
記号表の管理の手法としては，記号表をどの枝も大体同じくらいの長さになるような二分木のとして構成する *balanced binary tree* の手法やハッシュ法 (hashing) によるものなどがある．
ハッシュ法の基本的な考え方は次のように述べられる: 記号列に記号表のアドレスを対応させる適当な関数（ハッシュ関数）を用意しておき，記号の登録時にはこの関数の値のアドレスから順に探して最初の空いた場所にこの記号を登録する．識別子の記号表での検索のときには，ふたたびハッシュ関数でその識別子に対応するアドレスを計算して，このアドレスから検索することで時間を短縮する．ハッシュ関数としては，計算がやさしく，違う記号列を与えたときに，できるだけ違うアドレスを返すようなものが望ましい．

6 意味解析

構文解析でパースされたプログラムは，型の整合性などセマンティックスに関するチェックをする必要がある．型は C では配列や構造体などがあるが，これらは，

1. 基本的な型（`int`, `float` などあらかじめすでに用意されているもの）
2. すでにある型から新しい型を構成する方法（C では配列，構造体，ポインタなど）
3. 型名の宣言

により構成される．たとえば C では型の宣言は，

```
typedef struct{
                char name[MAXNAME];
                int age;
                float weight;
               } person;
```

などとして行う．また C では再帰的な型もゆるされることに注意する．例えば:

```
typedef struct{
                .
                .
                .
                struct person mother;
                struct person father;
               } person;
```

型の整合性のチェック (type checking) としては，演算での被演算子の間の型の整合性，代入式の左辺と右辺の型の整合性などを調べる必要がある．たとえば C ではポインタに整数を足したり引いたりすることができるが，ポインタに整数を掛けることは許されていない．言語によっては，関数に渡す引数の型のチェックも行う．
型は同じ構造をしていれば同一のものと見なして扱うやり方（C など — これを構造同値という）と，違う名前がついていれば，別のものとして扱うやり方（pascal など — これを名前同値という）がある．
C^{++} や C の新しい規格では，方の多重定義 (overload) がありこれもうまく扱う必要がある．

7　中間コード

コンパイラーによっては，中間段階で（後の処理に都合のよいような形式の）言語のコードを生成するようになっている．
三つ組による中間コードでは

行番号 (演算子，被演算子 1，被演算子 2)

という形のコードの列に構造木をばらす．

例 9

```
a=100*b - c/d;
```

というソースコードが，木の構造:

```
        =
  a            -
        *           /
 100        b   c       d
```

という構造木にパースされたとき，これを，

```
1.  (*, 100, b)
2.  (/, c, d)
3.  (-, 1, 2)
4.  (=, a, 3)
```

という中間コードに変換する．

これの変形で，

(演算子，被演算子 1, 被演算子 2, 結果を収納する変数)

という四つ組による中間コードの形式もある．上の例の中間コードは，この形式では

```
(*, 100, b, temp1)
(/, c, d, temp2)
(-, temp1, temp2, a)
```

となる．後で述べるコードの最適化のときには，こちらの形式の方が都合がよいことが多い．上の中間コードでは，テンポラリーな変数を導入することにより，ソースコードでのように演算が入れ子になっていないことに注意．

8 ブール表現の評価

ブール表現はブール地あるいは関係式をブール演算子 and, or, not でつなげたものである．ここでは，

$$E \to E \text{ or } E|E \text{ and } E| \text{ not } E|(E)|\underline{id}\ \underline{resp}\ \underline{id}|true|false$$

として与えられているものとする．ここに，$\underline{resp}$ は<, =<,... などの関係記号とする．
ブール表現の解釈はブール値によるものと制御の流れ (flow of control) の方法によるものとがある．
ブール値による解釈では真偽を 1 と 0 で表現して，これにブール演算を他の数の演算と同様にほどこすものとして扱う．
たとえば，

```
a not b and not c
```

を，

```
t1 = not c
t2 = b and t1
t3 = a or t2
```

に相当する中間コードに翻訳する．関係式については，例えば "a < b" は "if a < b then 1 else 0" というような条件文と解釈して，たとえばこれを前の三つ組の書きかたでは:

```
0. if a < b goto 3
1. t = 0
2. goto 4
3. t=1
4.
```

のように解釈する．
ブール表現は主に(プログラム制御の)流れの管理に使われるので，制御の流れの手法では，ブール表現の解釈自身を最初からそのようなものとして実現する．このやり方ではブール表現が制御の流れとして直接翻訳されるため出力コードの効率は良いものとなる．

流れの制御文は

$$S \to \quad \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S_1 \\ \mid \underline{\texttt{if}}\ E\ \underline{\texttt{then}}\ S_1\ \underline{\texttt{else}}\ S_2 \\ \mid \underline{\texttt{while}}\ E\ \underline{\texttt{do}}\ S_1$$

というような形で与えられる．ここで E がブール表現でこれを翻訳する必要がある．
まずブール表現 E ごとに `E.true`, `E.false` というラベルを用意しておく．ここで，E が “`a < b`” のときには

```
if a < b goto E.true
goto E.false
```

が生成されるコードになる．また例えば E が “$E1$ `or` $E2$” のときには，`E1.true` と `E.true` は同じラベルとなり，$E1$ が真でないときには，$E2$ の評価をする．また，`E2.true` は `E.true` と同じラベルとなり，`E2.false` は E.false と同じラベルとして，

```
if E1 goto E1.true
goto E1.false
      .
      .
      .
E1.false: if E2 goto E2.true
      .
      .
      .
```

が生成されるコードとなる．C では制御文はこのような方式で解釈されるが，たとえば，

```
if (a==b || (c=d)!=e)
```

で `a` と `b` が等しいときには，`||` の右側は評価されず，したがって，特にこのときには `d` の値は `c` に代入されない．このようなクセを逆手にとるとトリッキーなプログラミングができるようになるが，プログラムの可読性を保持するためには，これは避けるべきであろう．

9 コードの最適化

出力コードの最適化の方向としては，

1. 出力プログラムの小型化
2. 実行時の効率の向上
3. 実行時の使用メモリーの少量化

といったことが考えられるが，1., 2., 3. は必ずしも共存できるとは限らない．

コンパイラによっては，1., 2., 3. のうちどれを主眼にして最適化を行うかをオープションで指定できるものもある．
最適化には，出力プログラムの走るチップに依存するもの (machine dependent) とそうでないもの (machine independent) がある．後者は中間コードの変換の形で実現できる．以下に最適化のいくつかの手法について述べる:

a. 共通な部分表現の除去．

例 **10** `a = b/(c + d) -(c + d)` は，

```
(+, c, d, t1)
(/, b, t1, t2)
(+, c, d, t3)
(-, t2, t3, a)
```

というような四つ組の表現に変換されるが，これをさらに

```
(+, c, d, t1)
(/, b, t1, t2)
(- t2, t1, a)
```

と変換することで改良できる．

このような改良では実行時の効率にはそれほど大きな寄与をしないように見えるかもしれないが，たとえば，この代入式がループの内側にある場合には，ここでの足し算が 1 回減らせたことが多大な影響力を持つ可能性もある．

b. 複写の伝搬の廃止と不要なコード の除去.

例 **11** `x = y;`
```
z = x-1;
u = x;
```
となっているとき，`y` を `x` に代入して，`x` を `y` に代入しているが，これを，

```
x = y;
z = y - 1;
u = y;
```

とする．この変換で，`x = y` という代入が不必要になり，これを取りさることができる．

c. コード の移動

ループの中で同じ計算を繰返しているとき，これをループの外に置くことで効率を上げることができる．

例 **12**
```
for ( i = 0; i < 20 * k; i++ )
     x[ i ] = 10 * a[ j ] + y[ i ];
```
は，

```
l = 20 * k;
m = 10 * a[ j ]
for ( i = 0; i < l; i++ )
     x[ i ] = m * a[ j ] + y[ i ];
```

とすることで効率をあげられる．

d. ループの制御変数の除去．

例 **13** `for ( i = 0; i < 100; i++ ) c[ i ] = a[ i ] * b[ i ];`
は，

```
1. i = 0
2. goto 4
3. i = i + 1
4. if i > 100 goto 8
5. t = 4 * i
6. c[ t ] = a[ t ] * b[ t ]
7. goto 3
8.
```

というような三つ組表現にあわされる．ここに $5.$ で “4” となっているのは配列の要素の大きさである．これは，

```
1. t = 0
2. goto 4
3. t = t + 4
4. if t > 400 goto 7
5. c[ t ] = a[ t ] * b[ t ]
6. goto 3
7.
```

とすることで改良できる．

e. 演算の強さの軽減．

たとえば，`x = 2 * y` は 2 をかけるという演算のかわりに（2 進法であらわされている）`y` の値の入ったレジスタの内容を左にシフトする演算で置き換えることで高速化できる．同様に，`x = y +1` はレジスタの内容をひとつ上げるという演算を用いることで改良できる．

以上で見た例はコードの局所的な改良法だったが，プログラムの構成単位の何ブロックにもわたるような大域的な最適化についても色々と研究されている．

10　コンパイラの移植

コンパイラを移植が容易な形に設計するための手段の一つとして，ある仮想機械の機械語コードを出力するようにコンパイラを書いておく，ということが考えられる．このような設計にしておけば，システムに依存するのは，コンパイラの入力部分をのぞくと，この仮想機械の機械語コードを個々のシステムのための機械語のコードに変換する部分のみとなり，主にここを書きかえればよいことになる．

例 **14** コンピュータ M と N があるとする．M で動いている，C のソースコードを M の機械語に変換するるコンパイラ C_0 があるとする．このとき N で動くコンパイラを作ることを考える．まず M 上で，C でを使って N のコードを出力するコンパイラを製作する．このソースコードを S として',これを C_0 でコンパイルしたものを C_1 とする．つぎに S を C_1 を用いてコンパイルすると，N で動く S で記述したプログラムで，N 用のコードを出すものが作成される．

上のような手法は boot strapping と呼ばれている，