プログラミング言語 10－1
（1987. 2. 13）

OS／omicron におけるシステム記述言語C処理系の設計思想とその最適化

田中泰夫，森岳志，屋代寛，並木美太郎[+]，中川正樹，高橋延匡
東京農工大学　工学部　数理情報工学科，+) 現在　日立製作所　基礎研究所

高橋研究室では，明析で手の入れられる研究用計算機環境を自前で構築しており，その記述言語Cの処理系“cat”も自作した．catは，その目的から効率の良いオブジェクト・コードが求められる．本稿では，ソフトウェア・ツールとしての応用，拡張性，保守性を考慮してフェーズ分割されたcat第2版の構成とその最適化について述べる．第2版catは，プリプロセッサ，パーザ，構文解析木レベルでのプロセッサ独立の最適化部，コード生成部，ターゲット・マシンであるMC68000を対象にしたプロセッサ依存の最適化部，アセンブラからなる．catの最適化の方針として，言語Cはソースレベルでかなりの最適化が可能なこと，及びシステム記述言語としてはプログラムの構造は保存すべきであるとの立場から，共通部分式の削除などは行なわないこととした．しかしながら，プロセッサ独立部での定数演算のまとめ上げと演算の強さの軽減，プロセッサ依存部でのアドレシング・モードの有効活用などは十分効果的であることが分かった．

Design Philosophy of the OS/omicron C compiler, CAT and Its Optimization Strategy

Yasuo TANAKA, Takeshi MORI, Hiroshi YASHIRO, Mitarou NAMIKI, Masaki NAKAGAWA, Nobumasa TAKAHASHI
Dept. of Information Science, Tokyo Univ. of Agriculture and Technology
(The fourth author currently at Advanced Research Loboratory, Hitachi Ltd.)

This paper describes the architecture of a language C compiler CAT for OS/omicron and its optimization strategy. As OS/omicron is programmed in C, CAT must produce efficient codes. But, CAT should not change the structure of programs, since a system programmer do not want his programs to be compiled in a way he would not expect. C itself allows him to control generated codes to be efficient. This leads to the optimization strategy that CAT must optimize the codes a programmer cannot optimize, but yet preserve the structure of programs intended by him. Nevertheless, folding and operator strength reduction in parse trees, full utilization of addressing modes in code generation and shortened branches in generated codes have turned effective.

1. はじめに

最近流行のワークステーションは，オペレーティングシステム（以下OSと略す）にUNIXを搭載しているものが多い．パーソナルコンピュータから移行したユーザは，満足している者がいるかもしれないが，VAXのようなUNIX本来のマシンから移った者には，本当に満足であろうか．確かに，MULTICSの反省としてUNIX本来の「Simple is beautiful」という設計方針は素晴らしく，実際に高い評価をすでに得ている．しかしながら日本語について考えた場合，多くの問題を露呈するUNIXは，元来アメリカで作られたものであり，日本語については当然，何等考えられていないため，簡単な事務文書を打つならワープロ専用機の方がはるかに優れている．つまり、日本文化を反映しない計算機はその価値が薄れる．

当研究室では，OSを含む全てのシステムプログラムを自作している．それは，特に以下の研究のために，明析で手の入れられるシステムソフトウェアを必要としたからである．

(1) 日本語情報処理の研究
(2) パタン認識（特に手書き文字認識）
(3) 文書出力の研究
(4) 浮動小数点の研究
(5) 計算機における教育（特に言語C−CAI）
(6) 人工知能の研究
(7) 並列処理の研究（以上のアプリケーションの処理速度向上のための並列計算機アーキテクチャの研究）

そのため，当研究室ではマイクロプロセッサにアドレス空間の広いMC68000を，基底言語に言語Cを採用し，そのコンパイラcatも自作した．

本稿では，catの開発の経緯およびその構成，そして最適化処理について述べる．

2. OS／π，OS／o，そしてcat

2. 1 設計方針

当学科では，パタン認識，人工知能のための並列処理の研究を行なうために，1980年度にプロジェクトPIEを発足させた．そこでは，問題解決向きのマトリックス型のマルチプロセッサシステムの開発を目指している．このマルチマイクロプロセッサシステムをSystem／π，そのOSをOS／πと呼んでいる．OS／oはその前段階としてのシングルプロセッサ用OSであり，OS／πを開発するためのOSとして位置付けられる．したがって，OS／oは開発環境の整備とシングルプロセッサ・マルチタスクによる評価データの収集を行なうことを一つの目的としている．しかし，OS／πは専用処理バックエンドプロセッサのOSであるため，システム資源全体の統括，ファイルシステム，マンマシンインタフェースなどは，OS／oが基本となる．そのため，OS／oは並列処理をOS／πに委託するとしても，その他の面では大学の研究室レベルの研究をサポートできる研究用スーパーパーソナル計算機のOSとして機能する必要がある．従って，以下の項目を設計方針とした．

(1) マルチタスクの機能を実現する

OS／πにおける完全な並列処理への移行を考慮し，またOS／oにおけるプログラミング効率の向上のためにマルチタスクの機能を実現する．プログラミング環境としてマルチタスクの有用性はいうまではない．また，人工知能の問題としては，問題を解決する場合に，木構造の探索問題に帰結できるものが多い．この問題解決をマルチタスクとして記述し，OS／oで擬似的な並列処理として実現し，プロセスの生成・実行状況をモニタする．

(2) システム記述言語として言語Cを用い，これを基底言語とする

ソフトウェア開発を行なう際，生産性や保守性，移植性を向上するためにも，アセンブリ言語でなく高水準言語で記述するのが望ましい．特に，大学の研究室では数年で担当者が卒業してしまうため，ソフトウェアの保守性は死活問題である．OS／oでは記述能力の高さからシステム記述言語に言語Cを採用した．そして，アプリケーション言語の処理系は言語Cで記述し，複数の言語で記述されたプログラムの相互利用を図る．

(3) 日本語情報処理の研究のために2バイトの漢字コードを標準とする

既存のOSで用いられている文字コードは，通常1バイトコードを採用している．これは，既存のOSが欧米からの輸入品であり，文化の基盤となる文字も輸入してしまったことに他ならない．しかし，日本語情報処理を行う場合，1バイトコードだけでは不十分であることは明らかである．そこで，モードを切り替えたり，エスケープ文字を使用して1バイトコードと2バイトコードを混在させるなどして漢字を表現しているが，そのためにアプリケーションプログラムの作成に厄介な問題が生じている．小さなプログラムをつくってみたら，エスケープ文字に対する状態遷移の処理が半分以上を占めていた，などということは日常茶飯時である．OS／oでは文字コードに2バイトを標準とし，アプリケーションプログラムの問題をこの種の非本質的な問題から開放する．

(4) オブジェクトモジュールはOS／o，OS／πで共通とし，動的なリロケータビリティとリエンタラビリティを保証する

マルチプロセッサシステムでは，ベースレジスタの動的変更が起きることが予想されるため，実行中にオブジェクトモジュールをリロケーションすることを予想してかなければならない．このため，動的なリロケータビリティが要求される．また，並列処理においては，同一の手続き部を複数実行することが考えられる．複数実行する手続きをメモリ空間上で一つにまとめるには，プログラムをリエントラントなものにすればよい．一方，単一プロセッサのOS／oにおいても実記憶方式でマルチタスクを実現するには，これらの保証があることが望ましい．タスク管理，メモリ管理のために柔軟性と可能性が得られる．言語Cコンパイラ cat は，このためにリエントラントでリロケータブルなコードを生成する．

(5) プログラムをROM化する

頻繁に使用されるコンパイラなどのシステムプログラムやOSの核をROM化することによりディスクからのロード時間を省き，プログラムの暴走から保護できる．この先は，ヒューマンフレンドリな設計の観点からもスーパーパーソナル計算機として特に重要なことと考える．

2. 2 実行環境

OS／oでは第2．1節の理由により，図1のような実行環境を設定した．

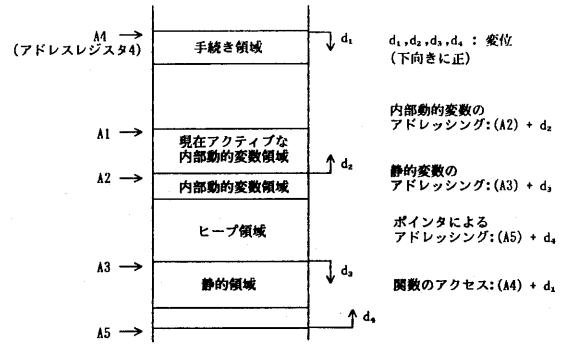

図1 OS／oの実行環境

リロケータブルなコードを実現するために

(a) 分岐命令は，PC相対

(b) データ参照はアドレスレジスタ相対

として絶対アドレスを一切用いないことにした．次にリエンタラブルなコードを実現するために，手続きとデータは分離した．データ領域としては，外部変数，外部静的変数，内部静的変数を一つにまとめ(静的データ領域)，内部動的変数はスタック上に割り当てることにした．

そこで，OS／oではMC68000の豊富なアドレスレジスタをベースレジスタとして使用し，図1のようにメモリ領域を構成している．各ベースレジスタの役割は，次のようになっている．

(a) 手続きの先頭を指すベースレジスタ（A4）

(b) 静的データ領域の基底を示すベースレジスタ（A3）

(c) 現在アクティブな動的変数領域の基底を示すベースレジスタ（A2）

(d) スタックポインタ（A1）

(e) ユーザ基底を示すベースレジスタ（A5）

静的データ領域の基底とは別にユーザ空間の基底を設け，データへのポインタはA5からの相対となっている．これは，ポインタによるタスク間コミュニケーションの可能性を残したためである．

(f) アドレスレジスタA5の補数値を保持するレジスタ（A6）

アドレスレジスタA5の補数値を保持するためにアドレスレジスタA6を設けた理由は，

変数のポインタ値：データ領域の基底（A2またはA3）

＋変位（d）－ユーザ空間の基底

を次の一命令で算出するためである．

lea d（A2またはA3，A6．L），ワークレジスタ

このようにすることにより，実行中どこで中断されてリロケーションが起きても，再開可能なコードとなった．

3. catの再構成

3. 1 cat第一版

言語Cコンパイラ cat は，OS／oソフトウエアの中核をなす重要なシステムプログラムである．cat は1982年より，その設計・開発が始まった．cat 自身は生産性・保守性を考慮して，言語Cで記述されている．開発方法を図2に示す．cat の開発は，当時研究室に言語C処理系が存在しなかったことから，東京大学大型計算機センターのVAX／UNIXを用いて行なわれた．これはUNIX上に信頼性の高い言語C処理系が存在したこと，豊富なツール群が利用可能であることによる．この cat の構成は図3のとおりである．内部は2パス構成を取っているが，明確なフェーズ分割はなされておらず，言語Cソースファイルから直接リロケー

タブルオブジェクトファイルを生成する方式を採った．また，構文解析は文レベルまでに対しては再帰下向き構文解析法，式に対しては演算子優先順位法を用いている．

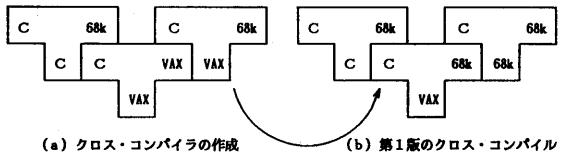

図2 cat の開発方法

開発の日程は以下のとおりであった．

(1) 1982 年春～ 1983 年春
cat の設計およびOS／o 実行環境の設定
(2) 1983 年春～ 1985 年春
VAX／UNIX 上でのクロスコンパイラ生成
(3) 1985 年春～ 1985 年冬
VAX／UNIXからターゲットシステムへの移植

移植に半年以上の歳月を費やしたのは
(1) 担当者が卒業してしまった
(2) 文書化が不十分であった
(3) テストが不十分であった
そして特に，
(4) 移植のためのツールが不十分であった
ことによる．この移植された cat（第一版 cat と呼ぶ）は言語Cで 20000 行におよび，プロシジャ・データを合わせて約 400KB のサイズとなった．

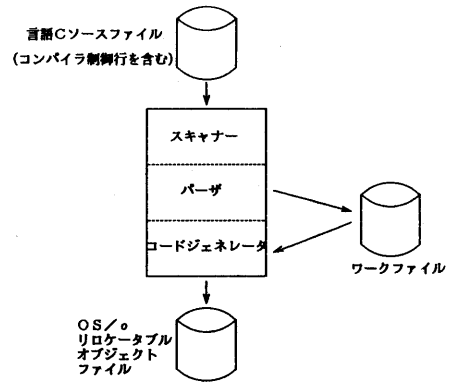

図3 cat 第一版の構成

なお，UNIXのYACC，PCC（Portable C Compiler）は使用しなかった．これは，全て自作するという方針のほかに，自前でパーザを用意したほうが浮動小数点方式の組込み，日本語化の変更など，将来の拡張・発展に有利であると判断したからであった．また，開発システムとしてUNIXは多いに利用すべきであるが，最後にOS／o，あるいはそのユティリティとして残る部分には，ブラックボックスを作らないという前提があるからである．

3. 2 cat 第一版の問題

第一版 cat は，当研究室初の大規模プログラムであり，担当の修士学生がいかに努力したかがうかがわれる．しかし，ソフトウエア工学の見地から眺めると，問題をいくつか抱えている．

(1) cat の構造は先にも触れたように，各フェーズ毎にプログラムが明確に分割されていない．これは，開発当初にセルフアセンブラがなかったこと，また，大容量の主記憶を仮定しオンメモリのコンパイルを目指したためである．このような一体構造のコンパイラは，各パス毎のデータ転送が内部で行なわれるためコンパイル速度の点では有利であるが，拡張・保守の点では全く不利である．この設計をした時点では，言語Cの処理系の規模が，実現されたものより，小さいと考えていたことによる．大学のように担当者が数年で入れ代わる環境で，拡張・保守が行ないがたいというのは致命的である．実際，第一版にオプティマイザを挿入するのは困難であると思われた．また，開発システム（VAX／UNIX）からターゲットシステムへの移植は，担当者が卒業したことも重なり難行した．
(2) 開発人員が少ないこと，また，開発期間を短縮するため，コード生成の段階でオプティマイズを行なっていない．例えば，定数同士の計算もコードを生成している上に，レジスタへの無駄なロードが非常に多い．また，MC68000 の命令を有効に使っておらず，非常に効率の悪いコードが出力されている．基底言語であるべきはずの言語Cコンパイラのコード生成効率が悪いということは，システム全体の実行効率を下げることにつながり，大問題である．

この２つの問題点のほかに，cat 第一版のマクロプロセッサは定数置換の機能しかなく，大規模プログラムの開発に必要なファイルのインクルード機能を持っていない．コンパイラはシステム開発の根底である．極端な言葉で表せば，システム記述言語として，位置付けられたコンパイラの欠点は，その言語を用いてシステムを記述するため，システム全体にその欠点が伝播すると言え

る．

3. 3 cat第二版の構成と特徴

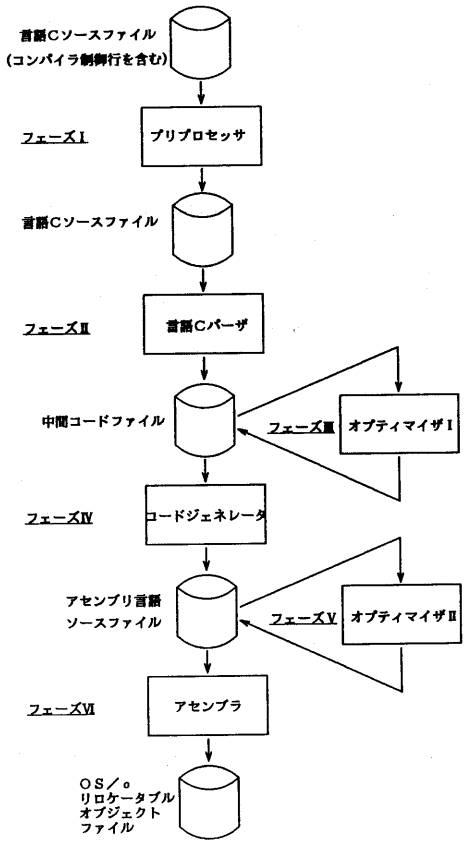

図4 cat第二版の構成

cat第二版ではフェーズ分割を行ない，上記の問題の解決を目指した．つまり，フェーズ分割でモジュラリティを高め，以下の効果を期待した．

(1) 信頼性，保守性の向上
(2) オプティマイザの追加，インタプリティブ・デバッガの開発 [8]，文書化ツールへの応用，などへの拡張性の向上
(3) 他の68000ファミリーチップへの対応

高橋研究室のここ5年間の経験で，プログラムを（中間ファイルへの出力などによって）完全に分割する目安を5000行，スーパプログラムでも10000行程度とした．

フェーズ分割により，パーザは中間コードと呼ばれるプロセッサに独立したコードを出力する，そしてコンパイラはアセンブラソースを出力する．新しいcatは次のフェーズから構成される．

1	プリプロセッサ	マクロ展開
2	スキャナ	字句解析
	パーザ	構文解析
3	オプティマイザ1	プロセッサ独立部分の最適化
4	コードジェネレータ	プロセッサ依存部の最適化と共にコード生成
5	オプティマイザ2	フェーズ2，3でもれた部分の最適化
6	アセンブラ	

これを，図4に示す．各フェーズ間のデータ転送はファイルを用いて行なう．また，各フェーズ毎のプログラムは言語Cで記述され，30000行におよぶ．このように分割することでコンパイル速度を犠牲にすることが懸念されるが，プロジェクトの寿命，大学という環境を考慮して保守性や拡張性の方を重視した．さらに，実行環境としてLSIファイルを実現して，ファイルI／Oの高速化で対処した．

これらのプログラムのうち，パーザは第一版のものをそのまま切り出し，若干手直しをした．アセンブラはすでに作成されていたので，オプティマイザ，コードジェネレータは新たに作成した．

4 オプティマイザの設計方針

4. 1 言語Cに適した最適化

高級言語処理系における目的プログラムの最適化処理を最初に行なったのは，FORTRANコンパイラである．それ以降，FORTRANコンパイラの歴史は，コンパイラの最適化の歴史そのものといえよう．最適化処理の対象も，もっぱら共通部分式の削除，不動変数のループ外への移動などである．しかし，言語CにとってFORTRAN流の最適化がそのままあてはまるかどうかは疑問である．言語Cにおいては，ソースプログラムのレベルで生成コードの効率を考えたプログラミングが可能である．言語Cの設計思想からいえば，システム記述言語という性格上，プログラマはせいぜいコードを見通していたい．コンパイラが最適化のために，プログラム構造を変えてしまうのは必ずしも好ましいことでない．したがって，言語Cにおける最適化処理の対象は，プログラムの構造は変えずにソースプログラムのレベルで考慮できないところにするべきである．つまり，プログラマの意図を尊重して，彼の手が届かないところの最適化を行なうべきである．

4. 2 プロセッサ独立部とプロセッサ依存部

第一版catのの開発で学んだ指針により，オプティマイザも保守性，拡張性に優れたものが望まれた．

目的プログラムの最適化処理の種類を大きく分けると，プロセッサのアーキテクチャに依存しないプロセッサ独立部と，プロセ

ッサ依存部がある．プロセッサ独立部の最適化処理とは，定数演算の最適化など，ターゲットプロセッサのアーキテクチャに関係無く，必ず効果が現れるところである最適化処理を指す．一方，プロセッサ依存部の最適化処理とは，ターゲットプロセッサのパフォーマンスを引き出すための最適化処理であって，他のプロセッサ上ではその効果が保証されないものである．

例えば，MC68000では定数を含む算術演算において，定数が1～8である場合，普通のadd，subからaddq，subqのような実行時間の短い命令に変換する最適化が考えられる．しかし，8086にはこのような命令が用意されていないため，MC68000のような最適化処理を施しても意味がない．

オプティマイザを作成する際，これらの部分を分離してインプリメントをした方が，後で他のプロセッサに移植する際，プロセッサに依存しない部分を再構成しなくてすむ．つまり，コンパイラの移植性が高くなる．

4．3 時間軸と空間軸

目的プログラムの最適化の狙いは，オブジェクトコードの実行時間の短縮，およびその占める領域の縮少である．前者を時間軸の最適化，後者を空間軸の最適化と呼ぶことにする．最適化においては，時間軸の最適化が空間軸の最適化につながることが理想である．しかし，最適化の種類と程度によっては両者が矛盾することも多い．catではオブジェクトコードを完全なリロケータブル，かつリエントラントなものにするため，サブルーチンからのリターン操作を行う際，MC68000のリターン命令（1ワード）を使用できない．そのかわりに4ワードのコードを出力している．そうすると，プログラムの中にreturn文が多数個ある場合，時間軸重視のオブジェクトコードと，空間軸重視のものとは違ってくる．一般にはプログラムの性質，および実行するマシンの規模によって，そのどちらかを選択する．大型計算機上の科学計算プログラムであれば，時間軸重視の方を選ぶが，パソコン上で走るプログラムなら，空間軸の方も考慮せざるをえないだろう．

5．オプティマイザ1

オプティマイザ1の最適化は，構文解析木レベルの最適化である．そして，その主たる効果は定数関係に表れる．この定数は，ユーザによって明示的に書かれたもの，配列のインデックス計算，構造体メンバへのアクセス，ポインタに対する加減算等のために出力されるものである．具体的には，次のことを行っている．

(1) 定数式の計算

図5．1に示すように定数同士の計算を行なってしまうことである．マクロで定数が書かれている場合，配列のインデックスが定数の場合に有効である

(2) 無駄な命令の削除

例えば，図5．2にのように明らかに結果がわかる場合，または演算をしなくてもよいものを削除する．オフセットが0Lである構造体，配列のインデックスが0である場合に有効である．

(3) 演算の強さの軽減

例えば，図5．3に示すように被演算子の一方が2の巾乗の定数である乗除余算にをシフト，算術ANDなどにおきかえることである．，ポインタに対する加減算，配列のインデックス計算に有効である．特に，ポインタの加減算に関しては，4バイト長の乗除算が必要となるが，MC68000が4バイト長の乗除余算を備えていないので有効である．

(4) 定数の畳み込み　例えば，図5．4に示すように離れた定数同士をまとめる演算である．構造体の多重アクセスに有効である．

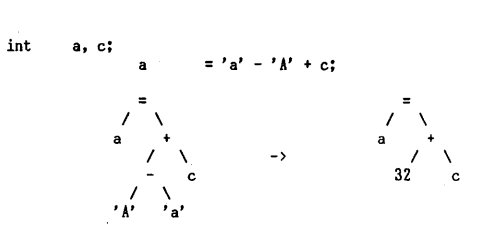

図5．1　定数式の計算

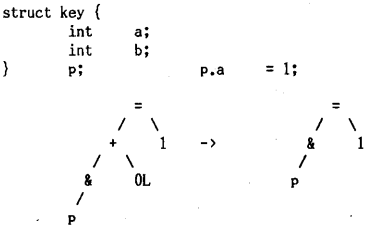

図5．2　無駄な命令の削除

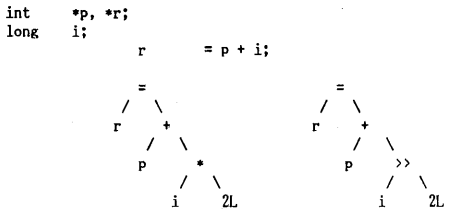

図５．３　演算の強さの軽減

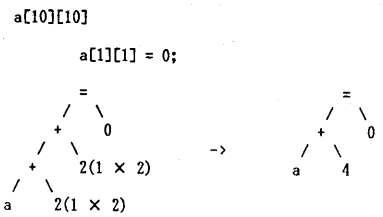

図５．４　定数の畳み込み

6. コードジェネレータの最適化およびオプティマイザ２

コードジェネレータは，パーザやオプティマイザ１の出力する中間コードファイルを入力とし，アセンブラ言語のソースファイルを出力する．出力ソースファイルをOS／ｏアセンブラやオプティマイザ２の入力として用いる．以下に，コードジェネレータの設計方針を述べる．

6. 1 アドレッシングモード

表１　アドレッシングモード　　(d は変位を表す)

アドレッシングモード	表記法	変位のビット長
データ・レジスタ直接	Dn	-------
アドレス・レジスタ直接	An	-------
アドレス・レジスタ間接	(An)	-------
ポストインクリメント付アドレス・レジスタ間接	(An)+	-------
プリデクリメント付アドレス・レジスタ間接	-(An)	-------
ディスプレースメント付アドレス・レジスタ間接	d(An)	16
インデックス付アドレス・レジスタ間接	d(An, Xn, {L, W})	8
アブソリュート・ショート	d.s	16
アブソリュート・ロング	d	32
ディスプレースメント付プログラムカウンタ間接	d(PC)	16
インデックス付プログラムカウンタ間接	d(PC, Xn, {L, W})	8
イミディエート・データ	#d	サイズ指定による
クイック・イミディエート	#d	命令に固有
インプランド・データ	{SR, USP, SP, PC}	-------

MC68000は，14種類のアドレッシングモードを持っている（表１参照）．コードジェネレータでは，これらのアドレッシングモードを有効利用して，MC68000のアーキテクチャと上記のプログラム実行環境に適したコードを生成する．そのためには，主にインデックス付アドレスレジスタ間接のアドレッシグモードを活用することによって，命令数の削減や実行時間の短縮を図っている．特にOS／ｏでは，データのアクセス方法として，間接アドレッシングモードが非常に有効である．

6. 2 コードジェネレータの最適化

コードジェネレータの生成するコードは，MC68000のアーキテクチャやOS／ｏの実行環境に適したコードになっている．しかし，ある特定の場合には，より一層適したコード生成が可能である．コードジェネレータでは，そのような場合に関して以下のような最適化を行なう．

(1) レジスタへの無駄なロードの削除

関数値を関数の引数とする場合などがこれに相当する．図６．１の例のような場合である．この例では，関数ｇを評価し，その関数値を関数ｆの引数とする．OS／ｏでは，関数値，引数値ともにスタック上に積まれる．単純にコード生成すると，一旦関数値を作業用レジスタにロードし，再びスタックに積むという操作を行なう．この無駄なレジスタへのロードを行なわない．

(2) 速くて短い命令

MC68000には，小さい定数の操作を行なうための，速くて短い命令がある．例えば，move命令では，ディスティネーションがレジスタで定数の値が小さい場合（−128 ～ 127）には，moveq命令を用いることにより，３倍速くなり，４バイト短くなる．(図６．２)

(3) アドレッシングモードの切替

OS／ｏでは，MC68000の全アクセス空間をアクセスする場合には，プロシジャ，データとも間接アドレッシングでアクセスする．

ところが，ここで問題となるのは，OS／oのリロケータブルな環境で24／32ビット範囲のアクセスを許すためには，数命令を要してしまうことである．一方，もしアクセス範囲が16ビットでおさまるならば1命令で実現することができる．そこで，オプションの切替により，領域が32KB以下のものにたいしては，効率の良いコードを生成するようにした．現実のプログラムでは，数KB程度のものがほとんであるために，この方法は有効である．

(4) リターンのまとめあげ

OS／oは，動的なリロケータビリティを保証するため，絶対アドレスを用いる命令（jsr, rts）を使用できない．このため，関数呼び出しに要する手続きが多少ふえており，リターン処理に関しても同様なことがいえる．そこで，関数内に複数のリターンがあった場合にはこれを一つにまとめる．これは，空間軸に対する最適化で時間軸に対してはむしろ悪くなるため，オプションによって切替可能にしている．

(5) 4バイト乗除算のインライン展開と関数コールとの選択

MC68000には，4バイト長の乗除算命令がないため，4バイト長の乗除算命令に対しては，ソフトウェアによって実現している．実現の方法としては，インライン展開を用いる方法と関数呼び出しを用いる方法がある．前者が，時間軸にたいして有効であるのに対し，後者は，空間軸にたいして有効である．そこで，前者の方法と後者の方法を，オプションによって切替ることができるようにした．

```
extern  int     f(), g();
auto    int     i;
i       = f( g())
```

```
<関数 fの呼び出し準備>                          <関数 fの呼び出し準備>
<関数 gの呼び出し準備>          ==>             <関数 gの呼び出し準備>
<関数 gの呼び出し>                              <関数 gの呼び出し>
move    (a1)+, d0       ; 関数結果の取りだし    <関数 fの呼び出し>
move    d0, -(a1)       ; 引数を積む            move    (a1)+, i(a2)    ; iに代入
<関数 fの呼び出し>
move    (a1)+, i(a2)    ; iに代入
```

図6．1　レジスタへの無駄なロードの禁止

```
auto    long    i, j;
i       = j     = 64;
```

```
        move.l  #64, d0                         moveq.l #64, d0
        move.l  d0, j(a2)       ==>             move.l  d0, j(a2)
        move.l  d0, i(a2)                       move.l  d0, i(a2)
```

図6．2　速くて短い命令

```
static  int     s;
f(s);
```

```
        subq    dv + 4, a1                      subq #dv + 4, a1
        lea     0(a6, a3.l), a0                 move    s(a3), -(a1)
        add.l   #s, a0
        move 0(a5, a0.l), -(a1) ==>
        move.l  #f, d0
        suba.l  #df + dv + 4, a2                suba.l  #df + dv + 4, a2
        move.l  #ret, (a2)                              move.l  #ret, (a2)
        jmp     0(a4, d0.l)                     jmp     f(a4)
        adda.s  #df + dr + 4, a2                adda.l  #df + dv + 4, a2
        addq    #2, a1                          addq    #2, a1
```

図6．3　アドレッシングモードの切り替え

```
auto    int     i, j;
if (…)
        return (j);
else
        return (i);
```

```
        move    i(a2), 4(a2)                    move    i(a2), d0
        movea   a2, a1                          bra     returnラベル
        move.l  (a1)+, d0
        jmp     0(a4, d0.l)

        move    j(a2), 4(a2)    ==>             move    j(a2), d0
        movea   a2, a1
        move.l  (a1)+, d0
        jmp     0(a4, d0.l)
                                return ラベル:
                                        move    d0, 4(a2)
                                        movea   a2, a1
                                        move.l  (a1)+, d0
                                        jmp     0(a4, d0.l)
```

図6．4　リターンのまとめあげ

6．3 オプティマイザ2

オプティマイザ2はオプティマイザ1とコードジェネレータで洩れた，機械依存部分の最適化を行う．コードジェネレータの生成するアセンブリ言語ソースファイルを入力とし，最適化されたアセンブリ言語ソースファイルを出力する．ここで，コードジェネレータから出力されるアセンブリ言語ソースファイルでは，命令語長をタブ以前の空白数で示しているのでオプティマイザ2は，番地をインクリメントする手間が軽減されている．

オプティマイザ2が行なう最適化処理は，現時点ではブランチ命令の最適化のみである．

具体的には，ブランチ命令は高水準言語のオブジェクトでは頻出するので，これだけでもかなりの効果がある．

(1) ブランチ命令において，オフセットが128バイト以下の場合にはショートブランチに変更する

(2) ブランチ命令において，飛び先が次アドレスの場合はそのブランチ命令を削除する

MC68000のすべてのブランチ命令の命令長と実行時間を表2に示す．

表2 ブランチコードのクロック数

	ブランチ	ショートブランチ
バイト数	4	2
クロック数（分岐あり）	10	10
（分岐なし）	12	8

例えば，catのfor文に対するコードは図7のようになっており，最適化を行えば，4×（ループ回数）クロック分，実行時間が短縮される．

また，オプティマイザ2は，一つのブランチ命令を最適化することによって新たに波及する最適化も行っている．

```
for (式1 ;式2 ;式3)              〈式1のコード〉
      〈文〉        ---〉 ラベル1
                              〈式2のコード〉
                              bcc ラベル2
                              〈文のコード〉
                              〈式3のコード〉
                              bra ラベル1
                        ラベル2
```

図7 for文に対するコード

7 オプティマイズの結果

表3 最適化による結果

第一版	149.6KB
第二版 コードジェネレータのみ	103.9KB
オプティマイザIを使用	102.4KB
オプティマイザIとIIを使用	98.7KB

現段階では，オプティマイズの効果を定量的に示すためのデータが不足している．そこで，catのパーザの部分（言語Cで約10000行）について，cat第一版の処理結果と第二版のそれを比較する．

第3章でも述べたとおり，cat第一版では，開発の性格上からまったく最適化を行っていない．そこで，第一版の生成したコードと，第二版で生成したコードを比較し，最適化による効果を検討した．その結果を表3に示す．

7. 1 オプティマイザ1

表4 オプティマイザ1の最適化項目

モジュール名	最適化のパターン						
	無駄定数の削除	定数式の演算	演算強さを減じる	型変換演算	畳込み演算	その他	合 計
argname.c	9	6	2	12	0	0	29
message.c	11	0	+2[2]	6	3	2[2]	24
pmessage.c	0	0	0	0	0	0	0
c.c	0	0	0	0	0	8[2]	8
type.c	25	7	21+1	40	6	0	100
tag.c	8	5	11+1	12	0	0	37
argdcl.c	5	2	0	3	0	0	10
case.c	18	6	20	22	11	0	77
dcltype.c	7	4	0	19	0	0	30
etc.c	60	0	6+1	25	2	0	14
exname.c	10	5	+1	11	4	0	31
exp.c	267	8	7	74	0	0	356
gettoken.c	20	0	6+1	3	0	0	30
iconst.c	7	0	2	0	0	0	9
init.c	18	14	0	37	0	0	69
label.c	15	6	+1	17	8	0	47
name.c	11	5	12+1	15	8	0	52
opstack.c	18	7	18	14	0	0	57
parser.c	0	0	0	0	0	0	0
program.c	20	8	0	137	0	0	165
pexp.c	58	1	2+2	37	0	0	100
合 計	587	84	107+11	484	42	10	1325

注） 1: 正符号(+)が付いている数字は論理演算を示す(変数!=0 --〉 変数)
2: 最適化の結果が定数1になるもの
3: 定数の符号反転演算

オプティマイザ1では，定数計算の最適化を主におこなった．表4に，オプティマイザ1を挿入したときの最適化項目を示す．

これをみると，主な最適化の項目は，無駄定数の削除，型変換の演算，演算の強さを減じることであることがわかる．また，表3より，空間軸に対してはそれほど大きな効果がないことがわかる．ただし，このパーザが非常に入念に設計，コーディングされており，研究室のプログラミング規範として使われているものであることを考慮する必要がある．つまり，オプティマイザ1の効果が少なかったのは，コーディング段階での最適化が非常に細かくなされていたことにもよる．別のソースプログラム（手書き文字認識のもの）では，6%ぐらい向上がなされたものもある．

しかしながら，時間軸上ではかなりの効果が出ている．というのは，配列演算の場合には，そのオフセットを求めるために4バイト長の乗算が必要になるが，これをシフト演算に置き換えるだけでもかなりの効果が期待できる．現在，時間軸上の効果を計測するハードウェアツールをOS／9上に構築中である．

7. 2 コードジェネレータ

表3から空間軸の最適化による効果によってプログラムが約3分の2に減少することがわかる．

時間軸に対しては，例えば文字列の長さを数えるstrlen ()のような関数では，ポインタのインクリメントをadd. 1からaddqにするだけで8×ループ回数のクロック分だけ実行時間が減る．このようのに，コードジェネレータでもかなりの時間軸の効果が

期待できる．

7. 3 オプティマイザ2

表5 オプティマイザ2による最適化

	ブランチの数	ショートブランチの数	合計
オプティマイズ前	2346	0	2346
オプティマイズ後	583	1590	2173

表5より全体の約4文の3のブランチがショートブランチに置き換わることがわかる．これは，テストプログラムに使用したパーザが比較的小さなモジュールで構成されているためである．

8. 終わりに

catの開発に際し，我々はMC68000について以下のことに気付いた．特に(1)は，24ビットのアドレス空間をもちながら，本来は16ビットマシンだと気付かされるものである．

(1) アドレスレジスタ間接のオフセットが16ビットしか取れない．
(2) 命令セットの対象性に欠ける．

このうち，(1)については，32ビット版であるMC68020では，解決されているが(2)は解決されていない．このようにCPUのアーキテクチャが統一されていないとコンパイラ，特にオプティマイザ，コードジェネレータに負担がかかる．なお，残された問題としてcatの8ビット（ASCIIコード）版と16ビット（全2バイトコード）版との性能評価は今後の課題としたい．

最後に，本コンパイラの開発に携わった藤森英明（現：日本電気），篠田佳博（現：日立製作所），施清池（現：野村コンピュータシステム）の各氏に感謝する．

9. 参考文献

[1] 高橋延匡，他：“OS／oのアーキテクチャと第一版の実現”，情報処理学会オペレーティング・システム研究会資料24－11，1984．9．
[2] 中川正樹，他：“MC68000ユニ&マルチ・プロセッサ・システム用システム記述言語C処理系の開発”，情報処理学会計算機システムの制御と評価研究会資料21－7，1983．12．
[3] 施清池，他：“catのオプティマイザ(1)－プロセッサ独立部分での最適化”，情報処理学会第32回全国大会，1986．3．
[4] 屋代寛，他：“catのオプティマイザ(2)－プロセッサ依存部分での最適化”，情報処理学会第32回全国大会，1986．3．
[5] 屋代寛，他：“OS／omicron 用言語Cコンパイラcatの開発－VAX／UNIXクロスシステムからの移行－”，情報処理学会ソフトウェア工学研究会48－1，1986．6．
[6] 並木美太郎，他：“言語Cコンパイラcatの方式設計”，情報処理学会ソフトウェア工学研究会48－2，1986．6．
[7] 中田育男著：“コンパイラ”，産業図書
[8] 田中泰夫，他：“言語Cインタプリティブデバッガ”，情報処理学会ソフトウェア工学研究会47－2，1986．5．