教育用計算機の言語処理系の省メモリ化に関する検討

坂上 駿仁[*1]　重村 哲至[*2]

Consideration about a Memory Saving Method of the Compiler of the Educational Computer

Hayato SAKAGAMI [*1], Tetsuji SHIGEMURA[*2]

Abstract

It is important for students of computer science cource at the universities to understand the operation principle of the computer systems. In order to understand them, it is preferable to perform practical learning with the system that actually works. So we developed an educational computer of simple hardware configuration called TaC. In this paper, we describe a method for operating the language processor on the small main memory of TaC. We divided the compiler into six modules. Thereby, we describe a reduction method for main memory size consuming at a compile module-divided. Object size of each module is expected to be within 15 kiB.

Key Words : Compiler, Memory Saving, Educational Computer

1　はじめに

高専や大学の情報工学系学科の学生は，コンピュータ・システム全体の動作原理を理解することが重要である．これらを真に理解するためには，実際に動作するシステムを教材として実践的な学習を行うことが望ましい．しかし，実用的なシステムはもちろん，既存の教育用システムでも大規模すぎて限られた授業時間内で扱うことは難しい．そこで，徳山高専情報電子工学科では，コンピュータ・アーキテクチャ，OS，言語処理系等を横断的に学ぶことのできる教育用コンピュータ TaC[1] (Tokuyama Advanced Educational Computer)を開発した．TaC の外観を写真 1 に示す.

TaC の設計は，学生が TaC 自身の OS や言語処理系を実装例として参照しながら学習できるように，小規模で分かりやすいことを重視している．また，一般的なパソコンと基本的な構成が同じであることを学習者が実感できるように，TaC 上で高級言語を用いたプログラムの作成と実行が可能なセルフ開発環境の構築を目標としている.

写真 1　TaC の外観

本稿では，セルフ開発環境を実現するために，UNIX 上で動作している TaC 言語処理系を TaC の OS である TacOS[2]に移植する方法をコンパイラを中心に述べる.

[*1] 情報電子工学専攻

[*2] 情報電子工学科

2 C--言語

C--言語[3]は，TaC のシステム記述言語として設計された高級言語である．TacOS を記述するために使用される実用的な言語であり，言語処理系の記述も想定されている．学習者が C--コンパイラをコンパイラの実装例として参照しながら学習できるように，また，TaC の限られたハードウェア資源で言語処理系を動かせるように，小規模な言語として設計されている．さらに，初学者が容易に習得できるよう，C 言語の文法を簡略化し，Java 言語の特徴を取り入れている．

本研究では C--言語を入力とする TaC 言語処理系を TacOS へ移植する．

3 言語処理系の TacOS への移植

従来の TaC 言語処理系は図 1 に示すように，プリプロセッサ，C--コンパイラ，アセンブラ，リンカ，ローダからなる．これらは C 言語で記述され UNIX 上で動作していた．TacOS に移植するために，まずセルフホスティング化を行う．さらにメモリ制約問題を解決する．

3.1 セルフホスティング

セルフホスティングとは，コンパイラでそのコンパイラ自身のソースコードをコンパイルすることである．言語処理系全体を C--言語で書き換え，図 2 に示すように既存の言語処理系を用いることで，言語処理系を TacOS 上で実行できるようにする．

3.2 メモリの制約

TaC の主記憶サイズは 64 [kiB]であり，TacOS 実行時の空き容量は 29 [kiB]しかない．メモリが小さいため，言語処理系の移植は難しい．特に，言語処理系の中でプログラムの規模が最大のコンパイラは，C--言語に書き換えた時点で実行ファイルサイズが 52 [kiB]になる．C--コンパイラが小さなメモリで動作できるように工夫する必要がある．

4 コンパイラの省メモリ化手法

図 3 のように C--コンパイラを 6 つのモジュールに分割することで，プログラムが一時に消費するメモリの量を削減する．各モジュールは中間ファイルを出力し，それを次のモジュールの入力とすることで分割を実現する．

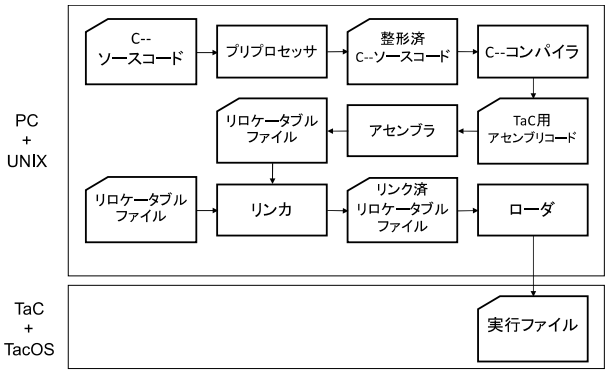

図 1 従来の TaC 言語処理系の構成

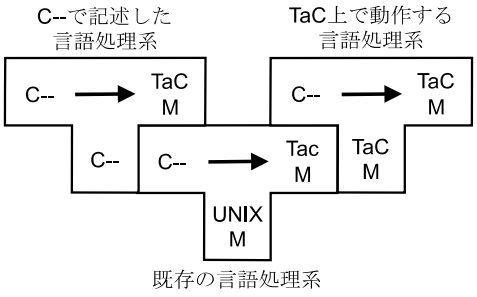

図 2 TaC 上で動作する言語処理系の生成

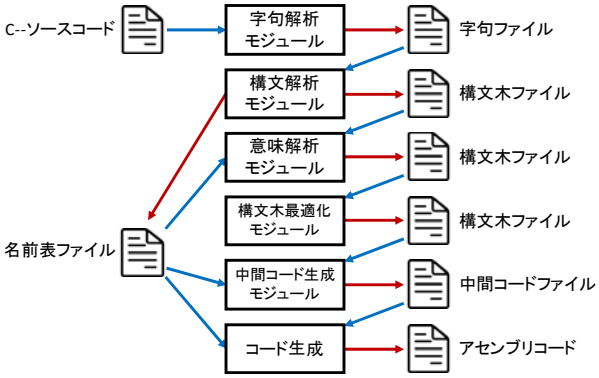

図 3 コンパイラの分割

4.1 従来のコンパイラの構成

図 4 に従来の C--コンパイラのクラス図を示す．コンパイラは次の順に動作する．

(1) 字句解析クラスは C--言語のソースコードを読み，構文・意味解析クラスへ字句トークンを出力する．

(2) 構文・意味解析クラスは，字句トークンを入力し，構文木クラス，名前表クラスに構文木と名前表を作る．

(3) 構文木最適化クラスは，関数等の入力が終了する毎に構文・意味解析クラスから呼ばれ，構文木の最適化を行う．

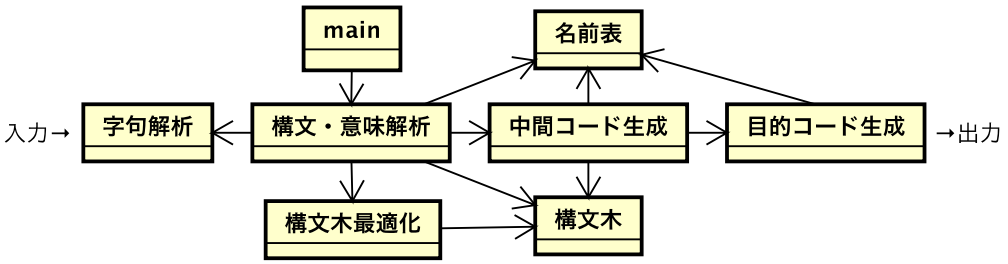

図4　C--コンパイラの構成

(4) 中間コード生成クラスは, 構文木最適化クラスに続いて呼び出され，構文木と名前表を利用して中間コードを目的コード生成クラスに出力する.
(5) 目的コード生成クラスは，中間コードを入力し，名前表を利用して目的コード(TaC のアセンブリ言語)を出力する.

このように，従来の C--コンパイラでは構文解析と意味解析を同時に行っていたが，これらの境界が曖昧な状態では学習教材として不適切であること，構文・意味解析部は全モジュールの中で最大規模であり多大なメモリ消費が予想されることから，これらは分離すべきである．そこで，まず，構文解析と意味解析が一体となった状態で5つのモジュールに分割する. 次に, 構文解析と意味解析を分割する.

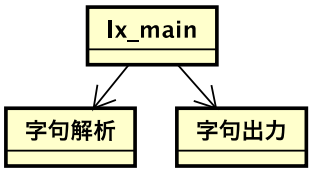

図5　字句解析モジュールの構成

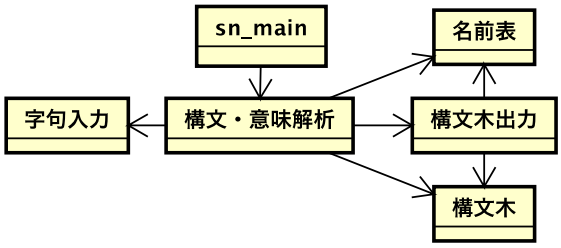

図6　構文・意味解析モジュールの構成

4.2　モジュール分割

3.1 節の各ステップをそれぞれが独立したプログラムにする．ステップ間では，ファイルを用いて構文木等を引き継ぐ．そのため，各ステップに main クラスと中間ファイル入出力クラスを新しく追加する.

例えば，構文・意味解析クラスから見た従来の字句解析クラスと，新規の字句入力クラスのインタフェースを同じにしておくことで，構文・意味解析クラスを変更することなく構文・意味解析モジュールを独立したプログラムにすることができる．このような手法を用いて，従来のクラスを変更することなく分割を実現する.

(1) 字句解析

構成を図5に示す. lx_main クラスはソースファイルをオープンした後，ソースファイルが EOF になるまで繰り返し字句解析クラスと字句出力クラスを交互に呼び出す．字句解析クラスはソースファイルから読み込んだ字句トークンを返す．字句出力クラスは字句ファイルに字句トークンを書き込む.

(2) 構文・意味解析

構成を図6に示す. sn_main クラスは, 字句ファイル, 構文木ファイル，名前表ファイルをオープンした後，構文・意味解析クラスを呼び出す．構文・意味解析クラスは字句入力クラスを用いて字句トークン入力し，名前表クラスに名前表を，構文木クラスに構文木を組み立てる.

構文・意味解析クラスは関数定義，大域変数宣言が終了する毎に構文木出力クラスを呼び出し構文木を構文木ファイルに出力する．構文木はファイルに出力されると消去される．構文・意味解析クラスの処理が全て終了した後，名前表クラスは名前表の最終状態を名前表ファイルに出力する.

(3) 構文木最適化

構成を図7に示す. op_main クラスは，入力用構文木ファイルと出力用構文木ファイルをオープンした後，構文木入力クラスを呼び出し，入力用構文木ファイルから構文木を生成する. op_main クラスは関数定義，大域変数宣言が終了する毎に構文木最適化クラスを呼び出し，構文木を辿りながら，構文木の最適化を行う．最適化が完了すると，構文木出力クラスを呼び出し，構文木を出力用構文木ファイルに出力する.

(4) 中間コード生成

構成を図 8 に示す．vm_main クラスは，構文木ファイル，名前表ファイル，中間コードファイルをオープンした後，名前表クラスを呼び出し名前表を生成する．さらに，構文木入力クラスを呼び出し構文木を生成する．vm_main クラスは関数定義，大域変数宣言が終了する毎に中間コード生成クラスを呼び出す．中間コード生成クラスは構文木を辿りながら中間コードを生成し，中間コードファイルに出力する．

(5) 目的コード生成

構成を図 9 に示す．tac_main クラスは中間コードファイル，名前表ファイルをオープンした後，名前表クラスを呼び出し，名前表を生成する．その後，中間コード入力クラスを呼び出す．中間コード入力クラスは中間コードを読みながら目的コード生成クラスを呼び出す．目的コード生成クラスはアセンブリコードを生成・出力する．

4.3 構文解析と意味解析の分離

従来の C--コンパイラは構文解析部で構文木を組み立てるのと同時に型チェック等（意味解析）を行っていた．本研究では，構文解析部は構文木を組み立てるだけで終了し，意味解析部が名前表と構文木を利用して意味解析を行うように変更した．

図 10, 11 に変更後の構文解析および意味解析モジュールを示す．構文解析モジュールは図 6 の構文・意味解析モジュールと，意味解析モジュールは図 7 の構文木最適化モジュールと，それぞれ同様の手順でファイルの入出力を行う．

この変更に伴い，意味解析のために型情報の一部を構文木に記録する等の変更を行った．変更点は以下の通りである．

(1) 型を変換をするだけの演算子は型チェックを行ってから読み捨てていたが，意味解析部で型チェックを行うために構文木に登録する．

(2) sizeof 演算子は構文解析部でサイズを計算し，結果を定数として構文木に登録していたが，意味解析部で計算するために演算子を構文木に登録する．

(3) 単項演算子'+'は型チェックを行ってから読み捨てていたが，意味解析部で型チェックを行うために，また，付属変数が代入不可になることを検出するために構文木に登録する．

(4) 初期化済み大域配列宣言と構造体宣言の区別は，意味解析部で行う．

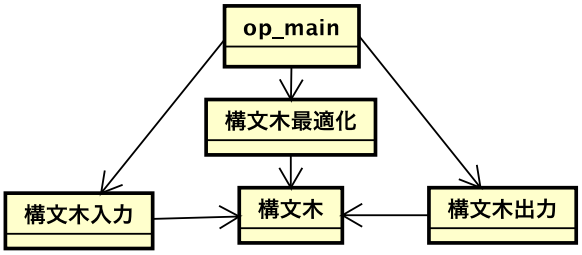

図 7　構文木最適化モジュールの構成

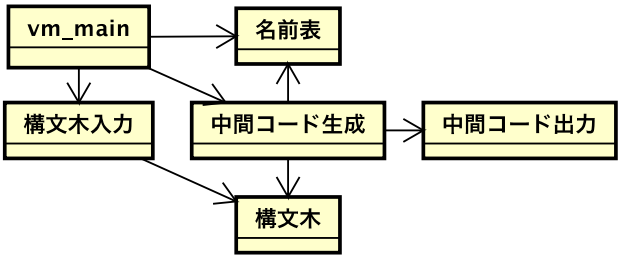

図 8　中間コード生成モジュールの構成

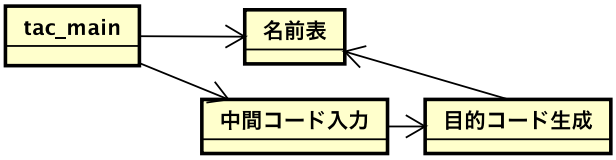

図 9　目的コード生成モジュールの構成

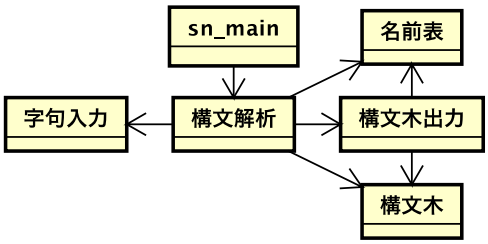

図 10　構文解析モジュールの構成

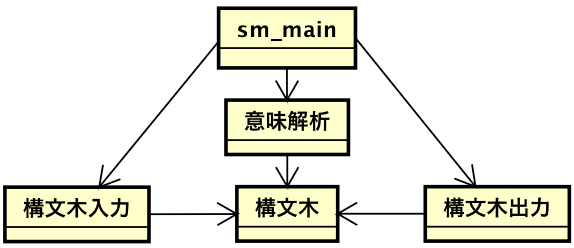

図 11　意味解析モジュールの構成

(5) 構造体フィールドのアクセスでは，意味解析部でフィールド番号の特定と型チェックを行う．構文木にフィールド番号の代わりにフィールド名を登録する．

(6) 意味解析部で型チェックを行うため，定数の型情報を構文木に残す．
(7) 基本型の「初期化済み大域変数」と「非初期化大域変数」を意味解析部で判別する．意味解析時に構文木があれば初期化済み，なければ非初期化である．
(8) 意味解析でのエラーメッセージに行番号を表示するため，構文木ノードにソースコードの行番号を残す．

5 中間ファイルの形式

分割した各モジュール間で受け渡す必要がある情報をまとめた結果から，表 1 のような中間ファイル形式を考案した．以下で各ファイルについて説明する．

(1) 字句ファイル

字句解析モジュールが出力するファイルである．1 行で 1 つの行番号付きトークンを表現する．定数トークン，名前トークンや文字列トークン等，特別に値が必要な場合に限りオプション値が出力される．

例として，リスト 1 のソースコードを字句解析した結果をリスト 2 に示す．なお，LxINT 等のトークンの種類を表す記号は，実際のファイルではトークンの種類を表す番号になる．

(2) 名前表ファイル

構文解析モジュールが出力するファイルである．1 行で名前表の 1 項目を表現する．各項目は，名前の綴り，関数や変数等の種類，データ型，配列の次元，"カウント"，大域フラグの 6 つの要素で表現される．"カウント"には種類によって必要な値（例えば関数なら引数の数）が格納される．

C--コンパイラでは，名前表に登録された局所変数はその有効範囲を過ぎると名前表から削除される．これにより，局所変数のスコープを管理している．名前表ファイルは構文解析完了時のものである．そのため，局所変数が格納されていない．局所変数の情報は，構文木を用いて意味解析部に引き継がれる．

(3) 構文木ファイル

構文解析部，意味解析部，構文木最適化部が出力するファイルである．意味解析部は部分木の型を追跡しながら，構文木の妥当性を検証する．その際，必要に応じて構文木を更新することがある．構文木最適化部では最適化を行い，構文木を更新する．そのため，各モジュールが出力する構文木ファイルの内容は異なる．

各行の第２列で，"N"は構文木の新ノード生成を表す．構文木は二分木で表現され，基本的には，ノードには型，左右の値には子ノードの番号が格納される．また，"F"は関数生成，"G"は大域変数生成，"D"は初期化済み大域変数生成，"B"は非初期化大域変数生成，"S"は文字列を表す．C--コンパイラでは関数や大域変数宣言毎にコード生成を行う．コード生成開始の目印として"F"，"G"，"D"，"B"の生成操作が構文木ファイルに含まれる．生成操作のオペランドは，生成される項目の名前表インデクス等である．

大域変数が初期化済みか否かの判別は意味解析部が

表１　中間ファイル形式

ファイル	形式
字句ファイル	"行番号　トークン　[値]"
名前表ファイル	"名前 種類 型 次元 カウント 大域フラグ"
構文木ファイル	"行番号 N 型 左の値 右の値" "行番号 F インデクス 深さ" "行番号 G インデクス" "行番号 D インデクス" "行番号 B インデクス" "行番号 S 文字列"
中間コードファイル	"命令 [引数]..."

リスト１　ソースコードの例１

```
1: int a;
2: return 0;
```

リスト２　字句ファイルの例

```
1  LxINT                  // int
1  LxNAME    a            // a
1  ;                      // ;
2  LxRETURN               // return
2  LxINTEGER 0            // 0
2  ;                      // ;
```

リスト３　ソースコードの例２

```
1: int i = ord('a') + 3;
```

リスト４　構文解析部出力の構文木ファイルの例

```
1 N SyCNST   97   TyCHAR // 文字定数'a'      (0)
1 N SyORD    0    SyNULL // ordに'a'が付属   (1)
1 N SyCNST   3    TyINT  //整数定数 3        (2)
1 N SyADD    1    2      //ordと3の加算      (3)
1 G 1                    // 大域変数iを生成
```

リスト５　意味解析部出力の構文木ファイルの例

```
1 N SyCNST   97   TyINT  // 整数定数 97      (0)
1 N SyCNST   3    TyINT  //整数定数 3        (1)
1 N SyADD    0    1      //97と3の加算       (2)
1 D 1                    // 大域変数iを生成
```

リスト６　構文木最適化部出力の構文木ファイルの例

```
1 N SyCNST   100  TyINT  // 整数定数 100     (0)
1 D 1                    // 大域変数iを生成
```

行うべきである．構文解析部の出力する構文木ファイルでは大域変数生成を操作"G"としているが，意味解析部では操作"D"と操作"B"に分けている．例として，リスト 3 のソースコードを構文解析した結果の構文木ファイルの内容をリスト4, 5, 6 に示す. リスト4, 5, 6 の"//"から右は説明用のコメントであり，右端括弧内の数字はノード番号を表す.

(4) 中間コードファイル

中間コード生成部が出力するファイルである．中間コードとしてスタックマシン命令をファイルに出力する．命令の後に，1 つ以上の引数が続く場合もある．命令の形式は, C--言語マニュアル [4]の付録 C に準拠する.

6 結果

以上の検討をもとに，C 言語で記述した従来のコンパイラをモジュールに分割した．セルフホスティングは後に行う．各モジュールの主要クラスのコメント行を含めた行数を表 2 に示す. 全クラスを 1,000 行以内で記述できた.

同じ動作をする C--プログラムと C プログラムは，ほぼ同じ行数で実現できる．また, C--プログラムの実行ファイルサイズは 100 行につき約 1 [kiB]の規模になる [2]．さらに，各モジュールの主要クラス以外の行数はコメント行を含めて合計600行以内である．これらから，各モジュールの実行ファイルサイズは最大でも約 15 [kiB]以下の規模になる見通しである．TacOS 実行時の TaC の空きメモリは 29 [kiB]であることから, 本コンパイラを TacOS 上で動作させた際，最低でも約 15 [kiB]のデータ領域（BSS，ヒープ，スタック領域）が確保されると予想できる.

一体版と分割版で大域名の前方参照ルールに違いが生じることがわかった．例えば，リスト 7 のようなソースプログラムをコンパイルすると，一体版では変数 i の未定義エラーになる．一方，分割版では正常にコンパイルできる．これは，分割版では意味解析を行う前に，構文解析部によって作られた最終的な名前表を読み込んでいることから，大域名の宣言と参照の順序に制約がなくなるためである．これを仕様の改善として許容するか否かは検討が必要である.

リスト 7　前方参照の例

```
1: void f() {
2:   i = 1;
3: }
4: int i;
```

表 2　主要クラスの行数

クラス	行数
字句解析	390
構文解析	924
意味解析	654
構文木最適化	624
中間コード生成	818
目的コード生成	924

7 まとめ

本研究では，TaC の主記憶サイズ制約の中で C--コンパイラを実行可能にするため，コンパイラを機能毎のモジュールに分割し，省メモリ化する方法を検討した．C--コンパイラを 6 つのモジュールに分割し，それらを順次実行し，同時に使用するメモリの量を少なくする．各モジュールは中間ファイルを出力し，それを次のモジュールの入力とする.

従来のコンパイラを構成するクラスに，main クラスと中間ファイルの入出力クラスを追加することで，従来のクラスを変更することなく，コンパイラを分割できることがわかった．また，構文解析と意味解析を分割するために構文木の仕様を変更した．構文木上に，意味解析で必要な型情報を記録するようにした.

モジュールを分割した状態のコンパイラを C 言語で記述した結果，TacOS 上で動作可能な規模に分割できることが確認できた．今後 TacOS 上でコンパイラが実行可能になる見通しを得た.

今後の課題として，以下が挙げられる.

(1) 分割した C--コンパイラのセルフホスティングを行う.

(2) 言語処理系全体を TacOS へ移植する.

文献

1) 重村他, 機械語教育用マイコン TeC とシリーズ化した教育用パソコン TaC のアーキテクチャ，情報処理学会研究報告書，Vol.2012-CE-117, No.9 (2012)，pp.1-7.

2) 村田他，教育用コンピュータの OS の設計，徳山工業高等専門学校, 研究紀要 第 39 号(2015 年), pp.57-62.

3) 重村他，教育用システム記述言語 C--，教育システム情報学会研究報告，Vol.22/No.4 (2007)，pp.75-80.

4) 重村哲至，プログラミング言語 C--Ver.3.1.2，https://github.com/tctsigemura/C--/blob/master/doc/cmm.pdf , (2016/9/7 アクセス)

（2016.10.13 受理）