P1: コンパイラの作成

津邑 公暁

Dec. 6, 2005

1 実験用ツールとシミュレータ

compkit.tar[*1] を展開すると，compkit というディレクトリが作成される。その中の README に書いてある install 方法に従って，install する。

```
[tsumura@edupc00 ~]$ cp ~tsumura/compkit.tar . ↵
[tsumura@edupc00 ~]$ tar xf compkit.tar ↵
[tsumura@edupc00 ~]$ cd compkit ↵
[tsumura@edupc00 ~]$ ls -F ↵
ChangeLog    README       getsym.c       sample.asm
Makefile     asm.pl       getsym.h       sr/
```

必要であれば，Makefile 中の *PREFIX* を変更し，make を実行すればよい。デフォルトでは*$HOME/p1* 以下に install される。

本キットには，以下のものが含まれる

1. getsym: 作成するコンパイラ用の字句解析ルーチン
2. asm: 目的計算機用アセンブラ
3. sr: 目的計算機用シミュレータ

コンパイラの作成実験では，対象となる**高級言語**（今回は Pascal サブセット）から目的計算機用の**機械語コード**を生成し，得られたコードが目的計算機上で期待通りの動作をするか確認する必要がある。

しかし今回の実験では，アセンブリ言語から機械語コードを生成するアセンブラが 2. に用意されているので，この入力となるアセンブラファイルを最終出力とするコンパイラを作成すれば良い。また，コンパイラの出力を 2. に掛けて得られた機械語コードは，テキスト付録 A で示されたアーキテクチャ上で動作する。この計算機は仮想的なもので実在しないので，3. のシミュレータ上で実行のシミュレーションを行う。

また，コンパイラについても**字句解析**部分は，1. で getsym() という関数が用意されているので，これを利用すればよい。使い方についてはテキスト付録 C を参照。

2 高級言語とアセンブラ

アセンブリ言語によるプログラミングについては H2 実験で行っているはずなので，メモリからレジスタにロードして演算しストアという計算の流れ，条件分岐命令による流れの制御，といった基本的な内容については既知とする。ただし，計算機のアーキテクチャや機械命令セットについては H2 のものと異なるので，テキストの付録 A，B を熟読すること。

また，基本的なアーキテクチャは単純なレジスタマシンであるが，多少アドレッシングに癖があるので注意

[*1] http://www.para.tutics.tut.ac.jp/~tsumura/compkit.tar から入手可能

が必要である。たとえば即値命令のオペランドは 16bit であるので，数値定数を単純に即値命令にコンパイルしてしまうと，図 14 のプログラムのように 16bit を超えた定数が扱えない。

次に，高級言語の命令をアセンブリ命令に変換する際の指針について述べる。

2.1 制御構文

アセンブリ言語には，各種フラグの状態により指定アドレスに分岐する命令しか存在しない。このため，`if` や `while` は条件式の計算 + 計算結果のフラグによりループ先頭や `else` 節に該当するコードへ分岐，という形になる (テキスト p.81 図 13 参照)。

2.2 変数とレジスタ

高級言語では，任意個数の変数を宣言でき，それらは値を格納する箱として自由に定義・参照できる。しかし，アセンブリ言語では決まった数のレジスタしか持たないため，元プログラムの変数をそのままレジスタに対応させることはできない。このため，大域変数は固定アドレス，局所変数はスタック上にアドレスを割り当て，変数 x の参照は x に割り当てられたアドレスの内容のロード，定義は同アドレスへのストアというコードになる。

大域変数については単に名前とアドレスの対応表を作って変換するだけだが，作成実験 4 で登場する局所変数については，同じ名前の変数が何度も出現するため，コンパイル時の表管理，および実行時の処理に工夫が要る。

3 ファイルの分割と make によるコンパイル

今回の実験のように大規模なプログラムを作成する場合，ソースファイルを複数に分割した方が良い。これは以下の理由による。

- ソースが 1 ファイルになっていると，一箇所を修正しただけで全体を再コンパイルする必要があり，時間が掛かる。これに対しソースを分割した場合，`make` を利用することにより，修正したソースファイルだけを再コンパイルして他のオブジェクトファイルとリンクすることができる。
- 適度な大きさに分割することによりソースの編集作業がやりやすくなる。

3.1 ファイル分割の方針

まずファイル分割以前の大前提として，大規模なプログラムは全体の流れをいくつかの機能に分割し，それぞれを別個の関数とする。各機能がまだ大きすぎるようなら，さらに細かい関数に分割する。一般に，一つの関数はエディタの一画面に収まる長さ〜せいぜい 100 行程度までに納めるのがよいとされている。

各関数は，関連性の深いもの同士をひと固まりとして，いくつかのファイルに分けて格納する。指針としては，最初に大きく分けた機能ごとに 1 ファイルとする，あるいはあるデータ構造を扱う関数群を 1 ファイルとする (たとえば変数テーブル操作に関する関数群を `vartable.c` とする)，といった方法が考えられる。

また，`#define` によるマクロ定義や `typedef` による型宣言などは，関数本体とは別にヘッダファイル (`*.h`) として独立させ，`#include` により取り込む形にするのがよい。

3.2 make

コマンド make は，Makefile という名前でファイル間の依存関係を記述しておくと，各ファイルの日付から必要なファイルだけ再コンパイルし，高速に実行ファイルを作成してくれる。

たとえば，プログラム compiler のソースが次のファイル群からなるとする。

```
main.c  parse.c  generate.c  table.h  command.h
```

ここで，各*.c の内容はそれぞれメインルーチン，構文解析処理，コード生成処理であり，テーブル定義ファイル table.h を parse.c, generate.c が，機械命令定義ファイル command.h を generate.c が include しているとする。この場合，Makefile の内容は次のようになる。

```
CC      = gcc
OPTS    = -O2 -Wall
OBJS    = main.o parse.o generate.o

target: compiler

compiler: $(OBJS)
    $(CC) $(OBJS) -o compiler $(OPTS)
main.o: main.c
    $(CC) main.c -c $(OPTS)
parse.o: parse.c table.h
    $(CC) parse.c -c $(OPTS)
generate.o: generate.c table.h command.h
    $(CC) generate.c -c $(OPTS)
#※上の字下げ部分は，かならず TAB を使うこと。スペースではエラーになる
```

まず，最初の XX=YYY の形式はマクロ定義であり，あとで$(XX) と書かれている場所が YYY に置換される。この機能は例の CC のようにコンパイラ名をマクロ化して，あとから cc のような別のコンパイラへの置換えを容易にしたり，OBJS のようにファイルの羅列をマクロ化して，何度も書く手間を省いたりするのに用いる。

コマンド make を実行すると，Makefile の内容を取り込み，まず target:と書かれている行を探し，指定されているファイル (この場合は compiler) を完成させようとする。

最初に実行した時には compiler が存在しないので，次に compiler:という行を探す。以下の行は，

$$\begin{aligned} &\mathit{file}:\ \mathit{file}_1\,\mathit{file}_2\ldots\mathit{file}_n \\ &\qquad \mathit{command} \end{aligned}$$

という形式をしており，*file* を作るためには $\mathit{file}_1, \ldots, \mathit{file}_n$ を使ってコマンド *command* を実行すればよいことを表している[*2]。

ここで，compiler を作るために必要な main.o, parse.o, generate.o が存在しないので，さらにそれぞ

[*2] 複数のコマンドを実行したい場合は，*command1*; *command2*; ... のようにセミコロンで区切って列挙する。

れの作成方法を指定した行を見に行き，各行で gcc が実行されて*.o ができ，最終的にこれらがリンクされて compiler が完成する。

次に，generate.c だけを変更し，再度 make したとする。今度は compiler を含めすべてのファイルが存在しているが，make は各ファイルのタイムスタンプをチェックし，ターゲットが依存しているファイルがターゲットより新しい場合は，依存をたどって必要な部分だけを作り直す。この場合，generate.c が新しくなっているので再コンパイルされて generate.o が作り直され，先ほど作った main.o, parse.o とリンクされて，新しい compiler が生成される。この過程では main.c, parse.c は再コンパイルされないため，単に gcc main.c parse.c generate.c -o compiler とするより高速である。

今度は，table.h を変更した場合を考える。この場合，*.c は一切変更されていないが，table.h を include した*.c はすべて再コンパイルしなければ，正しく結果が反映されない。この場合にも，make を実行することにより，タイムスタンプの新しくなった table.h に parse.o, generate.o が依存すると書いてあるため，この 2 つのオブジェクトが作り直され，元からあった main.o とリンクされる。

4 可読性・保守性について

ソフトウェア実験でも何度か触れたが，とくに今回のように大規模なプログラムでは，可読性・保守性に充分注意して設計やプログラム記述を行うこと。いき当たりばったりのコーディングをしていると，デバッグの際に，自分が苦労することになる。

参考までに，昨年度までの提出プログラムを見て気のついた点をいくつか挙げる。まず，同じエラーメッセージを何度も記述しているものが多かった。これは当然，

```
#define ERR_SYNTAX    "文法に誤りがあります\n"
```

などと定義しておくべきである (でないと，エラーメッセージを変えるのが大変)。

また，コード生成も

```
fprintf(outfile, "jnz    %d", label_num++);
```

のような文を毎回書かずに，これを関数／マクロ化しておいて

```
gen_jump("jnz");
```

などとしたほうが，ミスを犯しにくい。とくに副作用を伴うもの (上記の例では出力と同時に label_num の値を変えている) は，関数かマクロとして一箇所にまとめておかないと，あとで変更の必要が出てきたときに苦労する。

その他，ポインタ参照もうっかりミスではまりやすいので，複雑かつ同じパターンが繰り返し出るようなら，こまめにマクロ化した方がよい。

5 プログラムのドキュメントについて

最終レポートには完成したプログラムのリストの他，プログラムのドキュメント，実行例，および考察を書くこと。

ここでいうプログラムのドキュメントは，外部仕様書および内部仕様書の両方を含むものとする。前者は作

成したコンパイラのユーザ向けの使用説明書であり，使い方 (オプションや出力メッセージの説明) の他，コンパイラの機能や制限事項などについて明確に記述する [*3]。後者は今後，他人がこのコンパイラを保守・改良する (場合を仮定して) 際にコードを理解するのに必要な情報を記述する。具体的には各関数と機能の一覧，テーブルなどデータ構造の説明，とくにテクニックを用いた箇所 (要するにソースを眺めただけでは何をしているかわかりにくい箇所) があればその説明，などを書くとよい。

6 余力がある人は

余力がある場合は，以下のような発展課題を試みるとよい。

6.1 対象言語の拡張

最終的に提出するコンパイラは，テキスト p.78 図 10 および p.85 図 19 に与えられた言語仕様のプログラムをコンパイルできることを条件とする。しかし，可能であればより Pascal に近い構文を使えるよう拡張し，上位互換のコンパイラとしてもよい。

具体的には，for 文，case 文などは比較的容易に実現できる。また，引数の参照渡しや配列型の実現は多少困難であるが，意欲のある人はテキストに示された参考文献などを参照して挑戦してみるとよい。

6.2 生成コードの最適化

作成するコンパイラは，与えられたプログラムの動作を正しく実現するコードを生成できれば，生成されたコードの実行速度に条件は設けない。しかし実用的なコンパイラでは，高速，あるいはサイズの小さいコードを生成するためにさまざまな工夫をしており，可能であればこうした最適化技術についても実験を行っておくことが望ましい。

以下，具体例をいくつか挙げる。

1. 同じ定数値が複数回出た場合には値を一つにまとめる。また，宣言や定義のみ行われて参照されない変数を削除する。[簡単]
2. 演算命令を生成する前に計算式を簡略化しておくことで，冗長な計算や命令を省く。
   たとえば，定数式 2*3 はコンパイル時に計算してしまって定数値 6 に置き換えられる。また，(x*y)+(x*y) のような式は単純にコンパイルすると x*y を 2 回計算してしまうが，「x*y の値二つの和」という形をコンパイラが認識できれば，「x*y の結果をレジスタ R0 に入れて，R0+R0 を計算」というコードを生成できる。[やや複雑]
3. 変数へのアクセスは一般に，毎回レジスタへのロードやレジスタからのストア，というコードにコンパイルされる。しかし，ループ変数のように頻繁に定義・参照を繰り返す変数については，レジスタの一つをこの変数専用にしてしまうことで，毎回ロード・ストアする手間を省くことができる。これはループ回数の多いプログラムでは高速化の効果が高いが，レジスタをどの変数専用に使うかの判断が難しい。[複雑]

[*3] たとえば，`man gcc` などを参考にするとよい。