

情報システム実験 I

コンパイラ

伊澤侑祐

1 実験目的

コンパイラとは、ある言語で書かれたソースコードを読み込んで、コンピュータが実行できる機械語に翻訳するプログラムです。多くの場合、高級言語と呼ばれる C 言語や Pascal など人間が理解しやすい言語で記述されたソースコードを CPU（中央演算処理装置）が実行できる命令コードに変換するプログラムを指します。コンパイラの開発はむづかしいですが、これまでの研究開発の中から多くの体系的な開発手法が提案されています。本実験では、簡単なコンパイラを作成して、開発手法の基本を学びます。

本実験の目的は以下の通りです。

1. Pscal 風の高級言語 (While 言語) を仮想的なスタック機械の命令コードに翻訳する仕組みを学ぶ。
2. 仮想的なスタック機械の命令コードを経由して Python 命令コードへ翻訳する仕組みを学ぶ。同時に、実際に Python インタプリタで While 言語コンパイラを動作させてみる。
3. スタック機械である Python インタプリタの仕組みの概観を理解する。
4. Visual Studio Code に慣れる。

2 テキスト

以下の本 (**Pascal** 以外) をテキストとして、それぞれを 1 人 1 冊貸し出します。必要なら各自の責任の元に、貸し出し帳に記入の上持ち帰っても構いません。テキストの参照箇所を示す時、テキスト名を**コンパイラ I**のように略して表記しています。OCaml については、五十嵐淳教授 (京都大学) の「OCaml 爆速入門」を「手を動かしながら」読んでください。最初の授業でも解説します。

OCaml 爆速入門 「[OCaml 爆速入門](#)」、五十嵐淳、2019 年

コンパイラ I 「コンパイラ I 言語・技法・ツール」、A.V. エイホ他著、サイエンス社、1990 年

コンパイラ II 「コンパイラ II 言語・技法・ツール」, A.V. エイホ他著, サイエンス社, 1990 年
Pascal 「Pascal」, K. イェンゼン他著, 培風館, 1993 年

3 ソース言語とスタック機械の仕様

3.1 ソース言語の仕様

ソースコードを記述する言語をソース (原始) 言語と呼びます。本実験では、図 1 の仕様を持つ Pascal 風のソース言語のコンパイラを作成します。仕様の見方については、コンパイラ I の 2 章 2.2 節および 3 章 3.3 節を参照して下さい。この書き換え規則により、例えば代入文「 $i:=i+1$ 」が生成できます (図 2a)。また、図 2b に 9 の階乗を計算するプログラムの例を示します。

stmt_list	:=	stmt_list stmt; stmt;
stmt	:=	id := expr while cond do stmt begin stmt_list end
cond	:=	expr > expr expr < expr expr == expr
expr	:=	expr + term expr - term term
term	:=	term * factor factor
factor	:=	id num (expr)
id	:=	letter (letter digit)*
num	:=	digit digit*
letter	:=	a b ... z A B ... Z
digit	:=	0 1 2 ... 9

図 1: ソース言語の仕様

```
stmt  → id := expr
      → letter := expr
      → i := expr
      → i := expr + term
      → i := term + term
      → i := factor + term
      → i := id + term
      → i := letter + term
      → i := i + term
      → i := i + factor
      → i := i + num
      → i := i + digit
      → i := i + 1
```

(a) 文の生成例

```
a := 1;
i := 2;
while i < 10 do
begin
  a := a * 1;
  i := i + 1;
end
```

(b) プログラム例

3.2 スタック機械の仕様

オブジェクトコードはターゲット機械に依存します。本実験のターゲット機械は仮想的なスタック機械です。詳細はコンパイラ I の 2 章 2.8 節を参照して下さい。表 1 にスタック機械の命令セットを示します。なお、Python バイトコードへの変換のため、教科書にある命令セットを一部変更しています。

表 1: スタック機械の命令セット

push v	v をスタックに積む。
pop	スタックの最上段の要素を取り去る。
+	最上段とその下にある要素を取り出して加算し、結果をスタックに積む。
−	最上段とその下にある要素を取り出して下の値から上の値を引き、結果をスタックに積む。
*	最上段とその下にある要素を取り出して掛け算し、結果をスタックに積む。
>	最上段とその下にある要素を取り出し、下の値が上の値より大きい場合は 1(true)、そうでない場合は 0(false) をスタックに積む。
<	最上段とその下にある要素を取り出し、下の値が上の値より小さい場合は 1(true)、そうでない場合は 0(false) をスタックに積む。
==	最上段とその下にある要素を取り出し、下の値が上の値と等しい場合は 1(true)、そうでない場合は 0(false) をスタックに積む。
rvalue l	データの格納場所 l の内容をスタックに積む。
lpush l	最上段の要素を取り出しデータの格納場所 l に代入する。
copy	最上段の値を複写してスタックに積む。
label l	飛先 l を示す。それ以外の効果はない。
goto l	次はラベル l をもつ命令から実行を続ける。
gofalse l	スタックの最上段から値を取り去り、その値が 0 なら飛越しをする。
gotrue l	スタックの最上段から値を取り去り、その値が 1 なら飛越しをする。

3.3 具体例

目的のコンパイラはソース言語で書かれたプログラムをスタック機械の命令コードに変換します。代入文、begin-end 文、while 文の具体例と図 2b のプログラムの翻訳結果を以下に示します。

1. 代入文

a := a * 2;

```
rvalue a
push 2
*
lpush a
```

2. begin-end 文

begin a := a * i; i := i + 1; end;

```
rvalue a
rvalue i
*
lpush a
rvalue i
push 1
lpush i
```

3. while 文

while i < 10 do i := i + 1;

```
label L.0
rvalue i
push 10
<
gofalse L.1
rvalue i
push 1
+
lpush i
goto L.0
label L.1
```

4 準備

4.1 資料

本演習では、[GitHub Organization](#) を通じて資料を提供します。[GitHub リポジトリ](#)からファイルをダウンロードできるか確認してください。

4.2 環境構築

プログラミング言語に OCaml を用いるため、OCaml のインストールが必要です。各プラットフォームごとにインストール方法を説明します。

4.2.1 macOS

1. [Homebrew](#) をインストールする
 - 出来なかったら [初心者向けガイド](#) も確認する
2. `brew install ocaml` を実行する
3. ターミナルで `ocaml` と打ち REPL が立ち上がるか確認

4.2.2 Windows

Windows では cygwin を使います。

1. [Cygwin](#) にアクセスして Cygwin の `setup-x86_64.exe` をダウンロードしインストール
 - 分からなかった場合は [Qiita](#) を参照
2. [OCaml for Windows](#) にアクセス
3. 64-bit をクリックし OCaml64.exe をダウンロードしインストール
 - 分からなかった場合は [Haskell 勉強会のページ](#) などを参照すること
4. Cygwin で `ocaml` と打ち REPL が立ち上がるか確認

5 演習

5.1 1日目: OCaml 入門

1 日目では、まず OCaml に入門します。「[OCaml 爆速入門](#)」を読みながら OCaml プログラムの書き方を学びましょう。余った時間で「コンパイラ I」を読み、コンパイラの仕組みについて理解しましょう。

1. 「OCaml 爆速入門」を読みながら OCaml について勉強する。特に、以下の章を読み練習問

題を解く。

- [OCaml 爆速入門 \(その 1\)](#)
 - [OCaml 爆速入門 \(その 2\) – データ構造の基本](#) の「ヴァリエント」、「再履ヴァリエント」
 - [OCaml 爆速入門 \(その 3\) – 変更可能データ構造](#) の「制御構造」
2. コンパイラ I (以下、テキスト) の 2 章「簡単な 1 パス コンパイラ」の 2.1 節から 2.5 節を読んで、コンパイラの仕組みを理解する。

さらに時間が余った場合は、2 日目の内容を進めてください。

5.1.1 レポート

まず、自分が勉強のために記述した OCaml プログラムとその実行結果のうち、面白いと思ったプログラムを 3 個選んでレポートにまとめてください。まとめるとき、なぜ面白いと思ったのか文章で説明してください。さらに、以下の OCaml 爆速入門の練習問題を解きその解答をレポートにまとめてください。

1. 与えられた正整数 n に対、 $1^2 + 2^2 + \dots + n^2$ を計算する再帰関数 `square: int -> int` を定義せよ。(引数 n が負である場合の動作はどう定義してもよい。)
2. 与えられた正整数 n に対し、 n 番目のフィボナッチ数を計算する再帰関数 `fib: int -> int` を定義せよ。(引数 n が負である場合の動作はどう定義してもよい。)
3. 一品が菜食主義者でも食べられるかどうかを判定する関数 `isVeggieDish: dish -> bool` を定義せよ。
4. 定食が菜食主義者でも食べられるかどうかを判定する再帰関数 `isVeggieMenu: menu -> bool` を定義せよ。

5.2 2 日目: 仮想スタック機械への簡易コンパイラの作成

まず、コンパイラの雛形となるソースコードを [GitHub](#) からダウンロードできるか確認してください。

2 日目は、While 言語から仮想スタック機械へのコンパイラを製作します。配布するソースコードは大部分が実装されていますが、パーサーが足し算と `<` 以外の演算を認識しません。また、`begin - end` 文 と `while` 文を認識しません。

課題 1 引き算、かけ算、割り算、`>`、`<=`、`>=`、`==` などの演算を実装してください。具体的には、`syntax.ml`、`parser.mly` と `lexer.mll` を改造し、上記の演算を認識できるよう改良してください。

課題 2 `begin - end` 文 と `while` 文を実装してください。具体的には、`syntax.ml`、`parser.mly` と `lexer.mll` を改造し、上記の文を認識できるよう改良してください。

課題 3 追加した演算や文を、対応する仮想スタック命令へ翻訳するプログラムを `virtual_stack.ml` に実装してください。

具体的には、以下の手順で取り組んでください。

課題 1

1. `syntax.ml` に `Sub`, `Div`, `Mul`, `EQ` などを `Syntax.t` に定義
 - `Add` や `LT` の定義を真似して実装する
 - 適宜 `print_t` 関数のパターンマッチに 文字列の変換ルールを追加
2. `parser.mly` で引き算、割り算、等号などのトークンを定義する
3. `lexer.mly` で定義した文字列からトークンへ翻訳するルールを定義する
4. `parser.mly` で引き算、割り算、等号などのトークンから `Syntax.t` への翻訳を定義する

課題 2

1. `syntax.ml` に `Seq` 型、そして `While` 型を追加する
2. `parser.mly` で `begin - end`、そして `while` を認識するためのトークンを定義
3. `lexer.mly` で文字列から定義したトークンへ翻訳するためのルールを定義
4. `parser.mly` でトークンから `syntax.ml` で定義した `Assign`, `Seq`, `While` へ翻訳するルールを定義

課題 3

認識するための改良が終わったら、仮想スタック機械へのコンパイラを実装してください。大部分が実装されていますが、パーサと同様に、仮想命令生成器 (`virtual_stack.ml`) が代入文、`begin - end` 文、そして `while` 文を認識しません。これらを正しく仮想スタック機械の命令へ翻訳できるように `virtual_stack.ml` を改良してください。

具体的には、以下の手順で取り組んでください。

1. `virtual_stack.ml` で引き算、割り算、等号などを適切な仮想機械命令へ翻訳する
 - `Add` や `LT` のパターンを真似して実装する
2. `virtual_stack.ml` で `Seq`, `While` を適切な仮想機械命令へ翻訳する
 - `Assign` のパターンを真似して実装する

5.2.1 レポート

自分が改良した内容をレポートにまとめて提出してください。レポートにまとめる際、実行した `While` 言語ソースコードと実行結果を含めることが望ましい。

5.3 3日目: Python バイトコードコンパイラの作成

まず、コンパイラの雛形となるソースコードを [GitHub](#) からダウンロードできるか確認してください。

3 日目は、実装課題と調査課題の二種類があります。2 日目の実装が 8 割以上完成している人は実装課題に取り組んでください。2 日目の実装が間に合わなかった人は 2 日目の実装の続きに取り組んだ上で調査課題を遂行してください。どちらも取り組んだ場合は加点します。

5.3.1 実装課題

2 日目までに実装した仮想スタック機械コンパイラを Python バイトコードへコンパイルします。まず、簡単なプログラムを利用して Python バイトコードへ変換してみましょう。emit_pyc.ml と assemble_pyc.ml を用います。assign.while という代入を行う While 言語プログラムが用意されています。このファイルを例としたときの、Python バイトコードのコンパイル手順は以下の通りです。

```
$ ./while_lang test/assign.while
```

すると、test/assign.pyc というファイルが生成されます。この中には Python バイトコードのバイナリが含まれています。assemble_pyc.py で Python インタプリタで読み取れる表現に変換しているので、Python 2 インタプリタ (3 ではないので注意!) で実行できます。次のように実行してください。

```
$ ./interpret.py test/assign.pyc
```

実行に成功すると、何らかの答えが返ってきます。

emit_pyc.py は簡単な演算や文ならコンパイルできますが、相変わらず演算をや begin – end 文が実装されていませんね。

実装課題 では、emit_pyc.ml の実装済みのコードを参考に、未実装の演算や文の Python バイトコード命令への翻訳を行ってください。実装した内容をレポートにまとめて提出してください。

5.3.2 調査課題

- 仮想スタックマシンの命令と Python バイトコードの命令は、(ほぼ) 一対一に対応できます。具体的に、どの命令からどの命令へ翻訳すればよいか、仮想スタック機械命令を自分で 3 個選び、Python バイトコード命令への対応の仕方を説明してください。[Python バイトコードの逆アセンブラ](#)に Python の命令セットが書かれているので、よく読んで対応を探してください。
- やり残した 2 日目の課題に取り組んで 3 日目のレポートに含めて報告してください。

5.3.3 レポート

実装課題か調査課題に取り組み、レポートにまとめて提出してください。レポートにまとめる際、特に実装課題の場合、実行した While 言語ソースコードと実行結果を含めることが望ましい。

6 Python バイトコードを調べる上での資料

- 「[Python バイトコードの逆アセンブラー](#)」
- 「[opcode_ids.h](#)」、CPython