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

   入力
   ```
   a := a * 2;
   ```
   出力
   ```
   rvalue a
   push   2
   *
   lpush  a
   ```

2. begin-end 文

   入力
   ```
   begin a := a * i; i := i + 1; end;
   ```
   出力
   ```
   rvalue a
   rvalue i
   *
   lpush  a
   rvalue i
   push   1
   lpush  i
   ```

3. while 文

   入力
   ```
   while i < 10 do i := i + 1;
   ```
   出力
   ```
   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

4.2.1 OCaml のインストール

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。

■macOS　macOS ではターミナルを使用します。

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

■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

4.2.2 Visual Studio Code のインストール

Visual Studio Code (VS code) とは、2024 年現在最も人気のエディタです。様々な開発支援ツールをプラグインとしてインストールできるため、生産性が高いです。VS code 以外のエディタも使用して OK ですが、本演習は VS code をお勧めします。

ダウンロードページから適切なインストーラをダウンロードし展開してください。

4.3 OCaml の実行方法

OCaml には様々な処理系が存在します。まず 1 番簡単な方法がインタプリタ実行する方法です。やり方は簡単で、プログラムを ocaml コマンドで実行する方法です。

```
$ ocaml <your program name>.ml # <your program name> は任意の文字列に変更
```

すると、実行結果が表示されます。

また、OCaml にはバイトコードコンパイラとネイティブコードコンパイラも用意されています。これらを使う場合は以下のコマンドを用います。

```
# バイトコードコンパイラを用いる場合
$ ocamlc -o <outputname> <your program name>.ml

# ネイティブコードコンパイラを用いる場合
$ ocamlopt -o <outputname> <your program name>.ml
```

その後、<outputname> という名前で実行バイナリが生成されますので、./<outputname> というコマンドで実行してください。

5　演習

5.1　予習: OCaml 入門

演習に臨む前に、OCaml に入門しましょう。「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。プログラムは、`ocaml` コマンドで REPL を起動した後その中に打ち込むか、`turorial.ml` というファイルに記述し、`ocaml` コマンドを実行して結果を確認してください。

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「ヴァリアント」、「再履ヴァリアント」
   - OCaml 爆速入門 (その 3) – 変更可能データ構造 の「制御構造」

5.2　1 日目: レキサ・パーサジェネレータによる電卓アプリケーションの作成

1 日目では、まず OCaml の解説を行います。次に、今回のコンパイラ実装で核となるレキサ・パーサジェネレータについて学びます。OCaml コンパイラには、ocamllex/ocamlyacc というレキサ・パーサジェネレータが付属します。今回はこれらを利用し、簡単な電卓アプリケーションを作成します。雛形は `lexer.mll`、`parser.mly`、そして `calc.ml` に定義されています。

実装を開始する前に、まず、ocamllex/ocamlyacc 入門 (6 節) を通読してください。通読が終わったら、雛形を拡張して電卓アプリケーションを実装してください。雛形には足し算とかけ算が定義されていますが、引き算と割り算が定義されていません。他の実装を参考に実装してみましょう。

5.2.1　レポート

課題 1　「OCaml 爆速入門 (その 1、2、3)」の中で面白いと思ったプログラムを 3 つ選び、それぞれコード例を示しながらなぜ面白いと思ったか言葉で説明してください。

課題 2　作成した電卓アプリケーションの実装と動作例をまとめてレポートで提出してください。

5.3　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. sytax.ml に Sub, Div, Mul, EQ などを Sytax.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.mll 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 virual_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

5.3.1 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。

5.4 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。

5.4.1 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。

5.4.2 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ーに 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。

5.4.3 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 ocamllex/ocamlyacc 入門

6.1 ocamllex について

ocamllex は、正規表現の集合と、それに対応するセマンティクスから字句解析器を生成します。入力ファイルが lexer.mll であるとき、以下のようにして字句解析器の OCaml コードファ イル lexer.ml を生成することができます。

```
ocamllex lexer.mll
```

このファイルでは字句解析器の定義で、エントリーポイントあたり 1 つの関数が定義されています。それぞれの関数名はエントリーポイントの名前と同じです。それぞれの字句解析関数は字句解析バッファを引数に取り、それぞれ関連付けられたエントリーポインントの文法属性を返します。

字句解析バッファは標準ライブラリ Lexing モジュール内で抽象データ型 (abstract data type) として実装されています。Lexing.from_channel 、Lexing.from_string 、Lexing.from_function はそれぞれ入力チャンネル、文字列、読み込み関数を読んで、字句解析バッファを返します

使用する際は ocamlyacc で生成されるパーザと結合して、構文解析器で定義されている型 token に属する値をセマンティクスに基づいて計算します (ocamlyacc については後述) 。

6.1.1 字句解析の定義の記述法

字句解析の定義の記述法は以下の通りです。

```
{ header }
let ident = regexp ...
rule entrypoint =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint =
  parse ...
and ...
{ trailer }
```

header と trailer は省略可能です。

6.1.2 正規表現の記述法

regext (正規表現は) 次の文法で定義します。

’ char ’ 文字定数。OCaml の文字定数と同じ文法です。その文字とマッチします。

_ (アンダースコア) どんな文字とでもマッチします。

`eof`　入力終端にマッチします。警告: システムによっては対話式入力において、end-of-file のあとにさらに文字列が続く場合がありますが、ocamllex では eof の後に何かが続く正規表現を正しく扱うことは出来ません。

`" string "`　: 文字列定数。OCaml の文字列定数と同じ文法です。文字列にマッチします。

`[ character-set ]`　指定された文字集合に属する 1 文字とマッチします。有効な文字集合は、文字定数 1 つの ' c '、文字幅 ' c1 ' - ' c2 ' (c1 と c2 自身を含むその間の文字すべて)、または 2 つ以上の文字集合を結合したもののどれかです。

`[ ^character-set ]`　指定された文字集合に属さない 1 文字とマッチします。

`regexp *`　(反復) regexp とマッチする文字列が 0 個以上連なった文字列にマッチします。

`regexp +`　(厳しい反復) regexp とマッチする文字列が 1 個以上連なった文字列にマッチします。

`regexp ?`　(オプション) 空文字列か、regexp とマッチする文字列にマッチします。

`regexp1 | regexp2`　(どちらか) regexp1 にマッチする文字列か、 regexp2 にマッチする文字列のどちらかにマッチします。

`regexp1 regexp2`　(結合) regexp1 にマッチする文字列のあとに regexp2 にマッチする文字列を結合した文字列にマッチします。

`（regexp）`　regexp と同じです。

`ident`　前もって `let ident = regexp` で定義された ident に割り当てられている正規表現になります。演算子の優先順位は、* と + が最優先、次に ？、次に結合、最後に — になります。

6.1.3 Action の記述法 (2 日目以降に使用します)

action は OCaml の任意な式です。これらの式は lexbuf 識別子に現在の字句解析バッファが割り当てられた環境で評価されます。lexbuf の主な利用法を、Lexing 標準ライブラリモジュールの字句解析バッファ処理関数と関連付けながら以下に示します。

`Lexing.lexeme lexbuf`　マッチした文字列を返します。

`Lexing.lexeme_char lexbuf n`　マッチした文字列の n 番目の文字を返します。最初の文字は n = 0 になります。

`Lexing.lexeme_start lexbuf`　マッチした文字列の先頭文字が入力文字列全体において何番目の文字であるかを返します。入力文字列の先頭は 0 です。

`Lexing.lexeme_end lexbuf`　マッチした文字列の終端文字が入力文字列全体において何番目の文字であるかを返します。入力文字列の先頭は 0 です。

`entrypoint lexbuf`　(entrypoint には、同じ字句解析器の定義内にある別のエントリーポイント名が入ります) 字句解析器の指定されたエントリーポイントを再帰的に呼びます。入れ子のコメントなどの字句解析に便利です。

6.2 ocamlyacc について

ocamlyacc は、文脈自由文法の記述とそれに対応するセマンティクスから構文解析器を生成します。入力ファイルが `parser.mly` であるとき、以下のようにして構文解析器の Caml コードファイル `parser.ml` とそのインターフェイスファイル `parser.mli` を生成することができます (今回はインターフェイスファイルについての説明は省略する)。

```
ocamlyacc parser.mly
```

生成されたモジュールには文法で、エントリーポイントあたり 1 つの関数が定義されています。それぞれの関数名はエントリーポイントの名前と同じです。構文解析関数は字句解析器と字句解析バッファをとり、対応するエントリーポイントの意味属性を返します。字句解析関数は普通、字句解析記述から `ocamllex` プログラムによって作られたものを用います。字句解析バッファは標準ライブラリ `Lexing` モジュール内で抽象データ型 (abstract data type) として実装されています。トークンは `ocamlyacc` が生成するインターフェイスファイル `parser.mli` 内で定義されている型 `token` の値です。

6.2.1 構文解析定義の記述法

文法定義は以下のようなフォーマットになります。

```
%{
  header
%}
  declarations
%%
  rules
%%
  trailer
```

header と trailer は省略可能です。

6.2.2 declaration (宣言) の記述法

宣言は行単位で与えます。すべて % で始めます。

`%token symbol ... symbol` 指定されたシンボルをトークン (終端シンボル) として定義します。これらのシンボルは型 token に定数コンストラクタとして追加されます。

`%token < type > symbol ... symbol` 指定されたシンボルを、指定された型の属性を持つトークンとして定義します。これらのシンボルは、指定された型を引数に取るコンストラクタとして型 token に追加されます。type は OCaml の任意な型を置くことが出来ます。

`%start symbol ... symbol` 指定されたシンボルを文法のエントリーポイントとして定義します。それぞれのエントリーポイントは、同名の構文解析関数として出力モジュールに定義

されます。エントリーポイントとして定義されない非終端記号は構文解析関数を持ちません。初期シンボルは以下の %type 指示語を用いて型を指定する必要があります。

`%left symbol ... symbol`

`%right symbol ... symbol`

`%nonassoc symbol ... symbol` 指定されたシンボルの優先度や関連性を指定します。同じ行にあるシンボルはすべて同じ優先度になります。この行は、すでに現れていた %left 、%right 、%nonassoc 行のシンボルより高い優先度を、この行より後に現れた %left 、%right 、%nonassoc 行のシンボルより低い優先度を持ちます。シンボルは %left では左に、%right では右に関連付けられます。%nonassoc では関連付けられません。主にシンボルはトークンですが、ダミーの非終端記号を指定することも出来ます。これは rules 内において %prec 指示語を用いることで指定できます。

6.2.3 rules (ルール) の記述法

rules の文法は次のようになります。

```
nonterminal :
    symbol ... symbol { semantic-action }
  | ...
  | symbol ... symbol { semantic-action }
;
```

rules では %prec symbol 指示語を置くことで、デフォルトの優先度と関連性を、指定されたシンボルのものに上書き出来ます。

semantic-action は任意の OCaml の式です。この式は定義される非終端記号に対応するセマンティクス属性を生成するために評価されます。semantic-action ではシンボルのセマンティクス属性に $ (数字) でアクセス出来ます。$1 で第一シンボル (もっとも左) の属性に、$2 で第二シンボルに、という具合です。

7 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

8 参考文献

1. 「OCaml 爆速入門 (2021 年度「プログラミング言語」配布資料 (3))」、五十嵐淳、2021 年
2. 「Objective Caml 入門」、五十嵐淳、2007 年
3. 「Chapter 12: Lexer and parser generators (ocamllex, ocamlyacc)」、OCaml 公式ドキュメント、2024 年 8 月閲覧