情報システム実験 I
コンパイラ

伊澤侑祐

1　実験目的

本実験では、コンパイラの実装を通してある程度複雑なソフトウェアの実装手法を体験します。コンパイラは愚直に実装しようとすると大変ですが、これまで多くの実装手法が研究・開発されてきました。具体的には、While 言語という、制御構造として繰り返し構造のみを持った言語のコンパイラを実装し、その開発手法の一旦を垣間見ることがきます。

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

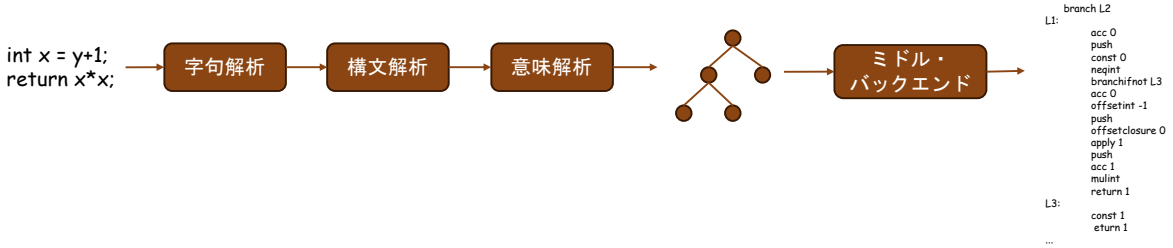

図 1: 一般的なコンパイラのコンパイルフロー図

3 コンパイラ

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。

具体的には、コンパイラはユーザによって書かれたプログラムから動作させたい機械 (CPU などの計算装置) が実行できるプログラムへ変換します。コンパイラの入力言語は**ソース言語**、ソース言語で書かれたプログラムを**ソースプログラム**と呼びます。一方、動作させたい機械のことを**ターゲット機械**、最終的に変換されるプログラムを**ターゲットプログラム**などと呼びます。

図 1 は、一般的なコンパイラのコンパイルフローを表しています。コンパイラはソースプログラムを一気にターゲットプログラムへ変換するのではなく、何段階かのコード変換フェーズを経ることによって変換しています。

次に、コンパイラのコンパイルフローの概要を説明します。コンパイラは、まず、ユーザが記述した While 言語プログラム (ソースプログラム) を入力として受けとります。次に、**字句解析器**がソースプログラムを**字句** (**トークン**、**token**) へ分割し、字句の種類を識別する**字句解析 (Lexical Analysis)** を行います。次に、**構文解析器**が字句の列を整理し文構造を把握する**構文解析 (Syntax Analysis)** を行い**構文木 (シンタックス・ツリー、syntax tree)** へ変換します。次に、変換された構文木に意味的な誤りがないかチェックする**意味解析 (Semantic Analysis)** を行います (本実験では意味解析の実装は時間の都合上スキップします)。意味解析が完了し誤りがないことが分かったら、構文木を**中間表現 (Intermediate Representation)** へ変換します。中間表現を対象に、プログラムを高速に動かすため、未使用定義の削除や関数呼出の展開といった**最適化 (Optimization)** を行います。最適化が終わったら、最後にターゲットプログラムを生成 (**コード生成、Code Generation**) します。各フェーズの詳しい説明は**コンパイラ**第 2 章あるいは第 3 章以降の該当ページを参照してください。

4 While 言語と While 言語コンパイラ

4.1 While 言語の仕様

本実験では、図 2 の仕様を持つ While 言語のコンパイラを実装します。仕様の中にある変数の意味は、図 2 右側の定義一覧を参照してください。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 3a）。また、図 3b に 9 の階乗を計算するプログラムの例を示します。

文	S	::=	$x :- a$
		|	skip
		|	S_1; S_2
		|	if P then S_1 else S_2
		|	while P begin S end
算術式	a	::=	x
		|	n
		|	(a)
		|	a_1 op_a a_2
算術演算	op_a	::=	$+$
		|	$-$
		|	$\star$
		|	$/$
真偽値	P	::=	true
		|	false
		|	not P
		|	P_1 op_b P_2
		|	a_1 op_r a_2
真偽演算	op_b	::=	and
		|	or
比較演算	op_r	::=	$<$
		|	$>$
		|	$\leq$
		|	$\geq$
		|	$==$

S	文 (statements)
a	算術式 (arithmetic expressions, arith)
x,y	変数 (program variables)
n	数値 (number literals)
P	真偽値 (boolean variables)

図 2: ソース言語の仕様

$$
\begin{aligned}
S &\rightarrow x := a \\
&\rightarrow \mathsf{i} := a \\
&\rightarrow \mathsf{i} := a + a \\
&\rightarrow \mathsf{i} := x + \mathsf{a} \\
&\rightarrow \mathsf{i} := \mathsf{i} + a \\
&\rightarrow \mathsf{i} := \mathsf{i} + n \\
&\rightarrow \mathsf{i} := \mathsf{i} + 1
\end{aligned}
$$

(a) 文の生成例

```
1        a := 1;
2        i := 2;
3        while i < 10 do
4        begin
5          a := a * 1;
6          i := i + 1;
7        end
```

(b) プログラム例

4.2 While 言語コンパイラの構造

本実験で実装する While 言語 (詳しい解説は第 4 節を参照) コンパイラのコンパイルフロー (コンパイルの段階を段階ごとに示したもの) です。

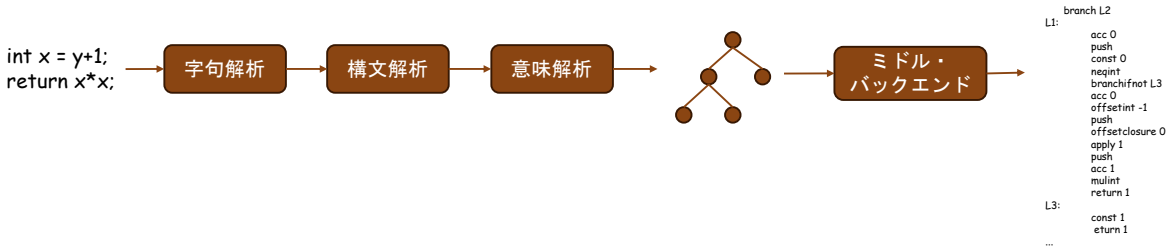

図 4: 本実験で実装するコンパイラのコンパイルフロー図

5 仮想スタック機械の仕様

本実験では中間表現としては仮想的なスタック機械 (仮想機械) の仮想命令を利用します。仮想機械とは、Intel CPU や Ryzen CPU といった具体的な機械ではなく、それらを抽象化した機械のことです。VirtualBox のような仮想環境ではなく、Java や Python などの処理系を思い浮かべてください。それらがいわゆる仮想機械を用いて実現されています。仮想的なスタック機械の詳細は**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。

5.1 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 3b 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
   ```
   1 rvalue a
   2 push   2
   3 *
   ```

```
4 lpush  a
```

2. begin-end 文

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

出力

```
1 rvalue a
2 rvalue i
3 *
4 lpush  a
5 rvalue i
6 push   1
7 lpush  i
```

3. while 文

入力 `while i < 10 do i := i + 1;`

出力

```
1  label   L.0
2  rvalue  i
3  push    10
4  <
5  gofalse L.1
6  rvalue  i
7  push    1
8  +
9  lpush   i
10 goto    L.0
11 label   L.1
```

6 準備

6.1 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。

6.2 環境構築

6.2.1 仮想開発環境のインストール

本演習では、OCaml やエディタが既にインストールされた仮想開発環境を用意しています。OS は Ubuntu 24.04 LTS (Long-Term Support) です。CPU は x86 ベースのもの (例えば、Intel の Core シリーズや AMD の Ryzen シリーズ) を対象としているので、ARM ベースの CPU を使用している場合 (例えば、macOS の M1 チップ) は仮想開発環境を利用せず手元で環境構築をしてください。

まず、Oracle VirtualBox を手元のマシンにインストールします。このリンクからダウンロードページへ行き、手元のマシンの OS (macOS や Windowd) に対応するバージョンをインストールしてください。

インストールが終わったら、Box から Ubuntu.ova をダウンロードしてください。起動が終わったら、VirtualBox に Ubuntu.ova を読み込みます。「仮想アプライアンスのインポート」を押し、Ubuntu.ova を選択したあと、「開く」、「次へ」をクリックしてください。案内に従っていけば仮想マシンを構築できます。

仮想開発環境は以下の ID とパスワードを設定しています：

- User: eecs-compiler
- Password: eecs-compiler

6.2.2 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　演習室では特別な設定は必要ありません。

お手元の Windows マシンでは第 6.2.1 章で説明した VirtualBox を利用した方法を活用してください。

6.2.3　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。

6.3　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。プログラムを `ocaml` コマンドで実行します。

■仮想開発環境、macOS の場合　仮想のコマンドをターミナルに入力することで OCaml プログラムを実行できます.

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

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。

■演習室の場合　コマンドプロンプトを起動し，`compiler-dayX` (X は 1,2,3 のどれか) に移動したあと，まず `setup.bat` を実行してください．実行は 1 度だけで OK です．`win64ocaml` というリポジトリをダウンロードします.

```
> .\bin\setup.bat
```

実行が終了したら，次のコマンドを実行してください.

```
> .\bin\run.bat .\win64ocaml\bin\ocaml.exe <your program name>.ml
```

バイトコードコンパラとネイティブコードコンパイラは次のコマンドで実行してください.

```
> .\bin\run.bat .\win64ocaml\bin\ocamlc.byte.exe <your program name>.ml

> .\bin\run.bat .\win64ocaml\bin\ocamlopt.byte.exe <your program name>.ml
```

7 演習

7.1 予習: OCaml 入門

演習に臨む前に、OCaml に入門しましょう。まず，授業配布スライド「OCaml Crash Course」を読み，内容を一通りターミナルに打ち込んで感覚を掴んでください．余裕があれば，「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。

具体的には、以下の内容に取り組んでください。

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

7.2 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。「OCaml Crash Course」と「OCaml 爆速入門」を実演しながら解説します．次に、簡単にコンパイラの仕組みについて解説します．その後，今回のコンパイラ実装演習で核となるレキサ・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、GitHub リポジトリからソースコードをダウンロードしてください。次に、ocamllex/ocamlyacc 入門 (8 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。

7.2.1 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。

7.2.2 演習室での実行方法

演習室の環境では、一般的な実行方法とはやり方が異なります。以下に簡単に整理します。コマンドプロンプト上で実行してください。

```
cd compiler-day1

# プロジェクトのビルド
.\bin\build.bat

# 中間ファイルの消去
.\bin\clean.bat

# バイナリの実行
.\bin\run.bat .\calc
```

7.3 2 日目: 仮想スタック機械への簡易コンパイラの作成

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。今回実装の対象とする制御構造は、`begin – end` 文、 `while` 文です。`if` 式の実装は時間が余った場合に取り組みますので、課題には組み込んでいません。

課題に取り組む前に、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。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 次の算術二項演算子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にしてください．

- 引き算 (Subtract, Sub)
- かけ算 (Multiply, Mul)
- 割り算 (Division, Div)

課題 2 次の比較二項演算子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にしてください．

- > (Greater Than, GT)
- >= (Greater or Equal, GE)
- <= (Less or Equal, LE)
- == (Equal, EQ)

課題 3 `begin – end` 文 と `while` 文を実装してください。`begin – end` は `Block` 型、`while` 文は `While` 型という名前を用い文法上で表現します。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。場合によっては示された補足資料を参考に実装してください。

課題 4 (時間があれば) 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。場合によっては示された補足資料を参考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・2

1. sytax.ml に Sub, Div, Mul, EQ などを定義
   - Add や Sub などの算術演算子は type a = ... に追加する形で実装
   - EQ や LT などの比較演算子は type p = ... に追加する形で実装
   - デバッグ用の補助関数である string_of_arith, string_of_predicate 関数のパターンマッチに、構文木から文字列の変換パターンを追加
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 3

1. syntax.mli に Seq 型、Block 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.mli で定義した Assign, Seq, While へ翻訳するルールを定義

■ **begin – end 文の定義の仕方** begin – end 文は、i := 1; j := 2 といった文の連続を表します。今回、 begin – end は Block 型という名前にコンパイルするとします。したがって、文の連続を Seq 型で表し、Seq 型を Block 型で包めば実現できます。Seq 型は文と文の連続なので文 (s, statement) を要素に持たせた形 (Seq of s * s) で表現し、Block 型は文である Seq 型を要素に持つため Block of s と定義すればよいでしょう。

では、文の連続はどのように表現すればよいでしょうか。文の連続は、パーサジェネレータのレベルで補足することができます。具体的には、parser.mly に文の連続をパースできるルールを追加すれば実現できます。では、どのように文の “連続” を表現すればよいでしょうか？ ocamlyacc のテクニックを用い、次のような定義を記述することで連続した文を Seq 型へ変換することができます。

```
1 statement:
2 | BEGIN statements END { ... }
3
4 /* 連続する statement 文() をパースするための記号 */
5 statements:
```

```
6 | statement SEMICOLON { ... }
7 | statement SEMICOLON statements { ... }
```

statments ルールは次のように読みます。まず、statements にマッチしたとき、一文しかなかったら statement を評価する (評価する所は ... になっています)。一方、文の終わりを表す SEMICOLON の後ろに連続する statements が来た場合、Seq 型に対応させる (対応の仕方は自分で考えてみてください)。

■ while 文の定義の仕方 While 型は条件式 (predicate) と文 (statement) のペアを要素に持つため、syntax.ml に定義する While 型は `While of p * s` となります。

while 文のパースは、while 文の構成要素を洗い出すことができれば簡単です。while 文は、次のトークン列から構成されます：

```
1 WHILE predicate DO
2   statement
```

これは、次のように読むことができます：WHILE が来たら predicate をパースする。その次に DO が来て、最後に statement が来る。この読み方に従ってパースし While 型に変換する定義を parser.mly の statement のルールに追加します。

課題 4

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

7.3.1 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。

■2 日目のレポートに関する注意 2 日目のレポートですが、最終的な評価は

- 1 日目のレポート
- 2 日目のレポート (課題 1〜4、途中までで OK)
- 3 日目のレポート (課題 1〜4、実装課題または調査課題)

に対して行うため、2 日目の課題が最後まで終わらない状態で、つまり、出来たところまで含めた形で提出して OK です。できれば課題 3 まで解けていることが望ましいですが、課題 3 が途中の状態でも評価します。

2 日目と 3 日目は地続きになっているので、3 日目は 2 日目に引き続き同じ内容に取り組みます。2 日目の課題で出来なかった・分からなかった所を明らかにして、3 日目に質問してください。ちなみに、3 日目に示している課題は発展課題の想定です。演習時間内に発展課題の説明を行いますが、取り組むかは自由となっています。

7.3.2 演習室での実行方法

演習室の環境では、一般的な実行方法とはやり方が異なります。以下に簡単に整理します。コマンドプロンプト上で実行してください。

```
cd compiler-day2

# プロジェクトのビルド
.\bin\build.bat

# 中間ファイルの消去
.\bin\clean.bat

# バイナリの実行
.\bin\run.bat .\while_lang
```

7.3.3 UNIX 環境 (macOS や仮想環境の Linux) での実行方法

```
cd compiler-day2

# プロジェクトのビルド
make

# 中間ファイルの削除
make clean

# バイナリの実行
./while_lang
```

7.4 3 日目: 2 日目の続き + 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。

7.4.1 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。予め用意された `assign.while` という While 言語プログラムを例としたときの、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。

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。`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` 文が実装されていないので、それらを実装してください。実装した内容をレポートにまとめて提出してください。

7.4.2 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) 一対一に対応付けることができます。具体的に、どの命令からどの命令へ翻訳すればよいか、次の 3 つに対し While 言語の仮想スタックマシン命令と 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。
  - `BINARY` で始まる二項演算命令
  - `STORE` あるいは `LOAD` で始まる値の格納・読込命令
  - `SETUP_LOOP`、`JUMP_ABSOLUTE`、`POP_JUMP_IF_FALSE`、`COMPARE_OP`、`POP_BLOCK` 命令を用いるループ構造
- 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。

7.4.3 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。

7.4.4 演習室での実行方法

演習室の環境では、一般的な実行方法とはやり方が異なります。以下に簡単に整理します。コマンドプロンプト上で実行してください。

```
cd compiler-day3

# プロジェクトのセットアップ
.\bin\setup.bat

# プロジェクトのビルド
.\bin\build.bat

# ビルドファイルの消去成果物も削除される ()
.\bin\clean.bat

# バイナリの実行
.\bin\run.bat .\while_lang <path-to-while-lang-file>
```

```
cd compiler-day3

# プロジェクトのビルド
make

# 中間ファイルの削除
make clean

# バイナリの実行
./while_lang <path-to-while-lang-file> # while 言語プログラムファイルへのパス
```

8 ocamllex/ocamlyacc 入門

8.1 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。

```
1    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) 。

8.1.1 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。

```
1  { header }
2  let ident = regexp ...
3  rule entrypoint =
4    parse regexp { action }
5        | ...
6        | regexp { action }
7  and entrypoint =
8    parse ...
9  and ...
10 { trailer }
```

header と trailer は省略可能です。

8.1.2 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。

8.1.3 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。

8.2 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)。

```
1    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。

8.2.1 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。

```
1  %{
2    header
3  %}
4    declarations
5  %%
6    rules
7  %%
8    trailer
```

header と trailer は省略可能です。

8.2.2 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。

8.2.3 rules (ルール) の記述法

rules の文法は次のようになります。

```
1 nonterminal :
2     symbol ... symbol { semantic-action }
3   | ...
4   | symbol ... symbol { semantic-action }
5 ;
```

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。

9 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」、Python 公式ドキュメント
- 「opcode_ids.h」、CPython

10 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