システムプログラミング実験
コンパイラの作成

担当教員：藤田

1　実験の目的

コンパイラとは，ある言語で書かれたプログラムを読み込んで，それを等価な別の言語に翻訳するプログラムです．多くの場合，C や Pascal など人にとって理解しやすい言語 (高級言語と呼ばれます) で書かれたプログラムを，機械 (CPU) の命令コードを使ったプログラムに変換するプログラムを意味します．コンパイラの開発は難しいことですが，これまでの研究開発の中からたくさんの系統的な開発技法が見出されています．本実験では，簡単なコンパイラを作成して，開発技法の基本的な部分を学びます．図 1 に，コンパイル処理の流れを示します．

ソースコード $\Longrightarrow$ [字句解析] $\rightarrow$ [構文解析] $\rightarrow$

$\rightarrow$ [意味解析] $\rightarrow$ [中間コード生成] $\rightarrow$ [コード最適化] $\rightarrow$

$\rightarrow$ [コード生成] $\Longrightarrow$ オブジェクトコード

図 1: コンパイル処理の流れ

本実験の目的は以下の通りです．

1. Pascal 風の簡単な仕様の高級言語で書かれたプログラムを，仮想的なスタック機械の命令コードに翻訳するコンパイラの作成を通して，その仕組みと作り方の基本を理解する（図 2）．
2. スタック機械の仕組みを理解する．

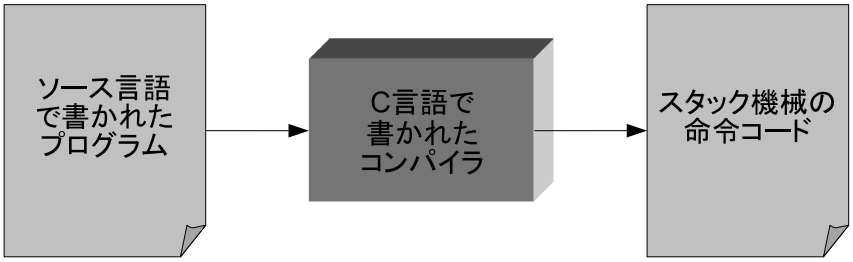

図 2: 作成するコンパイラ

2　ソース言語とスタック機械の仕様

2.1　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 3 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 4）．また，図 5 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 3: 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

図 4: 文の生成例

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

図 5: プログラム例

2.2 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．図 6 にスタック機械の命令セットを示します．

`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．
`lvalue` l	データの格納場所 l の番地をスタックに積む．
$:=$	最上段にある右辺値をその下の左辺値の示す場所に入れ，両方の値を取り去る．
`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．
`halt`	実行を停止する．

図 6: スタック機械の命令セット

2.3 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，while 文，begin-end 文の具体例と図 5 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
```
lvalue  a
rvalue  a
push    2
*
:=
```

2. while 文

入力
```
while i < 10 do i:=i+1;
```
出力
```
label   test
rvalue  i
push    10
```

```
<
gofalse out
lvalue  i
rvalue  i
push    1
+
:=
goto    test
label   out
```

3. begin-end 文

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

4. 図 5 のプログラム

入力
```
a := 1; i := 2; while i < 10 do begin a := a * i; i := i + 1; end;
```
出力
```
lvalue  a             lvalue  a
push    1             rvalue  a
:=                    rvalue  i
lvalue  i             *
push    2             :=
:=                    lvalue  i
label   test          rvalue  i
rvalue  i             push    1
push    10            +
<                     :=
gofalse out           goto    test
                      label   out
```

3 実験内容

本実験の課題は，前節で示した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翻訳するコンパイラ (C 言語で記述) を作成することです (図 2 参照)．以下に示す手順で進めてください．

1. **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〜2.7 節を読んで，コンパイラの仕組みを理解する．
2. 次に 2.8 節を読み，スタック機械の仕組みについて理解する．本実験で使用する命令コードは，テキストに記載されているものと同じです．
3. 次に 2.9 節の C プログラム (pp. 87–91) の構成と動作を理解する．テキストの C プログラムは，数字 (整数)，変数と演算子＋，－，∗, (,) によって記述できる中置形の算術式を後置形に変換して出力する機能を持ちます．プログラムを渡しますので，実際にコンパイルして実行してみてください．
4. 与えられた C プログラムを適宜修正し，スタック機械の命令コードを生成するプログラム (コンパイラ) を作成する．
5. 作成したコンパイラを用いてソース言語のプログラム (例えば図 5 のプログラム) の命令コードを生成し，動作を確認する．

上の手順にこだわる必要はありません．テキストの 2.5 節〜2.7 節は 2.9 節の C プログラムの各モジュールの説明にもなっていますので，プログラムに `printf` 文などを挿入して動作を確認しながら読むと効果的でしょう．

余力のある人は，以下の追加課題に取り組んでください．

6. while 文の翻訳ではラベルを生成しますが，次ページのヒントに沿った翻訳方法ではあらかじめ設定したラベルしか生成できません．例えば下の例のように，while 文の入れ子構造を持つプログラムでは多数のラベルを生成する必要があります (**コンパイラ I**，p. 80)．翻訳方法を見直し，ラベルを自動生成できるようにコンパイラを拡張してください．

```
sum := 0;
i := 1;
while i < 10 do
begin
     j := 1;
     while j < 10 do
     begin
          sum := sum + i * i + j * j;
          j := j + 1;
     end;
     i := i + 1;
end;
```

図 7: while 文の入れ子構造を持つプログラム例

3.1 コンパイラ作成のヒント

元の C プログラム (**コンパイラ I**，pp. 87–91) は中置形の算術式を後置形に変換します．このプログラムに必要な機能を追加していけば目的のコンパイラが作れます.

3.1.1 代入文

まず代入文の翻訳から始めましょう．代入文を翻訳するための手順を説明します.

1. 代入演算子 ":=" の認識

   元のプログラムは ":=" をトークンとして認識しません．関数 `lexan()` の `else if` の並びに ":=" を認識するための条件文を加えます（図 8）．適当な定数，例えば ASSIGN を定義して返します.

```
else if (t==':'){
   t=getchar();
   if (t=='='){
      return ASSIGN;
   }
}
```

図 8: ":=" の認識

2. 関数 `stmt()` の作成

   非終端記号 stmt に対応する関数 `stmt()` を作成し，関数 `parse()` から呼出します (元の `expr()` を `stmt()` に書きかえる)．図 9 に関数 `stmt()` の概略を示します．ただし，このままでは `rvalue` と `push` は出力されません．この 2 つの命令はどこで出力すればよいか各自で考えてみてください.

   プログラミングの基本的な姿勢ですが，一気に作らず，少しずつ作っては「セーブ，コンパイル，デバッグ」する習慣を身につけてください．大きなプログラムを一気に作ると，バグが出たときに問題箇所の特定が難しくなります．代入文単体で動くことを確認してから次の while 文に進みましょう.

3.1.2 while 文

次に while 文の翻訳について説明します．while 文の解析は代入文と比べて多少複雑です．`lookahead`（先読み記号）に何が入っているかを常に意識しながら翻訳を進めましょう.

翻訳を始める前にトークン `while` と `do` をキーワードとして登録します.（`while` と `do` を配列 `keywords[]` に追加します．その際，`while` には定数 `WHILE` を新たに定義して対応させます．`do` も同様.）

以下に翻訳手順の概略を示します.

1. `label test` を出力
2. 関数 `cond()` を呼出す

   `cond()` は元のプログラムには存在しません．非終端記号 cond に対応する関数 `cond()` を以下の手順で作成します.

```
stmt(){
   if (lookahead==ID){//代入文の翻訳
      printf("lvalue "); //"lvalue"を出力
      emit(ID, tokenval); //出力関数を呼び出す
      match(lookahead); //先読み
      if (lookahead==ASSIGN){//ASSIGN を確認
         match(lookahead); //先読み
         expr(); //expr() を呼び出す
         printf(":=¥n"); //":="を出力
      }
   }
   else if (lookahead==WHILE){//while 文の翻訳
      3.1.2 節参照
   }
   else if (lookahead==BEGIN){//begin-end 文の翻訳
      3.1.3 節参照
   }
}
```

図 9: 関数 stmt() の雛形と代入文の翻訳

(a) 関数 term() をコピーし，関数名を cond に書きかえる

(b) 関数 expr() を呼出す（元の factor() を expr() に置き換える）

(c) トークン <，>，= のいずれかを確認

(d) 関数 expr() を呼出す（元の factor() を expr() に置き換える）

(e) (c) で確認したトークン（<，>，= のいずれか）を出力

3. gofalse out を出力

4. トークン do を確認

5. 関数 stmt() を呼出す（再帰呼出しになっている点に注意）

6. goto test を出力

7. label out を出力

3.1.3 begin-end 文

begin-end 文の翻訳ができれば，図 5 のプログラムを翻訳することができます．各自で考えてみてください (加点あり).

ヒント：end が見つかるまで，(C 言語の)while 文を使って関数 stmt() を再帰的に呼び出す．その際，lookahead（先読み記号）に何が入っているかに注意し，適切な箇所で先読み関数 match(lookahead) を呼び出す.

4 実験レポート

以下に示す構成でレポートを作成してください．なお，作成ツールは Word, LaTeX など各自の好きなものを使って構いませんが，提出ファイルの形式は PDF としてください.

1. 表紙

   氏名，学修番号，提出日を書いてください.

2. 実験の目的と内容

   この資料を参考に簡単にまとめて下さい.

3. 開発環境

   実験で利用したコンパイラや OS の名前．個人で所有する PC を利用した時は，その環境も記述する．提出されたプログラムの動作を確認できるよう，コンパイルで使用したコマンド (gcc ∗.c など) を書くこと．コンパイルオプションが必要な場合は、忘れずに書いて下さい.

4. 作成したコンパイラの機能の説明

   (a) 各モジュールの機能

   モジュール（.c ファイル）単位ではなく，関数毎にその機能を説明してください. 既存の関数については**コンパイラ I** の 2.9 節を参考に簡単な説明をつけてください．その際，テキストを丸写しするのではなく自分の言葉で説明してください．新たに作成した関数についてはその構成と動作について詳しく述べてください．該当部分のプログラムを張り付けただけのレポートは認めません.

   (b) 関数間の呼び出し関係

   関数の呼び出し関係を図などを使って説明する (**コンパイラ I** の図 2.36(p. 83) を参照).

   ソースコードの中でコメントで説明することは認めません. ソースコードを読まなくても作業内容が分かるようにレポートの本体できちんと説明すること．なお，本体でソースコードの一部をコピペすることはかまいませんが，必要な説明を必ず加えてください.

5. 言語仕様に合わせて作成した適当なプログラムとそのコンパイル結果

   代入文と while 文についてはそれぞれ 1 例ずつ（例えば 3.3 節の例の）翻訳結果を示してください. begin-end 文ができた人は図 5 のプログラムの翻訳結果を示してください．シミュレータ上の実行結果は、画面のコピーをレポートに貼り付けてください。画面は、キーボード上の “Print Screen” キーを押すと画像としてクリップボードに取り込むことができます。Windows に付属するペイントツール等のソフトウエアを使うと、取り込んだ画像を必要な部分だけ切り取ることができます. プログラム作成の証拠になるので自分の名前の入ったタイトルバーは切り取らずに残すこと.

   また，追加課題ができた人は図 7 のプログラムの翻訳結果を示してください.

6. まとめ

   作成したコンパイラに関して面白い/難しいと思った点があれば述べてください．また，実験の内容や進め方について意見や感想などがあれば自由に述べてください.

7. 作成したコンパイラのソースコードをレポートの最後に付録として添付すること

   自分で作成，変更したファイルだけでなく，コンパイルに必要な全てのファイルを提出してください. 提出したファイルが，エラー無くコンパイルできることを必ず確認すること.