情報工学実験Cレポート
コンパイラ実験

氏名: 島谷　隼生 (Shimatani, Toshiki)
学生番号: 09428526

出題日: 2018 年 12 月 06 日
提出日: 2019 年 02 月 05 日
締切日: 2019 年 02 月 05 日

1　実験の目的

本実験では，情報系学科で学んだ 3 年間の総仕上げとしてコンパイラの作成を行うことを通じ，プログラミング言語で書かれたプログラムとアセンブリ言語の対応についてより深く理解し，木構造の取扱いについて習熟すること，および yacc.lex といったプログラムジェネレータを使用してプログラムを作成する経験を積むことを目的とする．

2　作成した言語定義

最終的に作成した言語の定義を BNF で記述する．以下がその言語定義である．

作成した言語定義 (1/3

$$
\begin{aligned}
< program > &::= < variable\_declarations >< function\_list > \\
&\mid < function\_list > \\
< function\_list > &::= < function >< function\_list > \\
&\mid < function > \\
< function\_list > &::= < function >< function\_list > \\
< function > &::= < pre\_func > (< argument\_list >) \\
&\quad < variable\_declarations >< statement\_list > \\
&\mid < pre\_func > () \\
&\quad \{< variable\_declarations >< statement\_list >\} \\
< pre\_func > &::= \texttt{func} < \texttt{IDENTIFIER} > \\
< argument\_list > &::= < argument > , < argument\_list > \\
&\mid < argument > \\
< argument > &::= \texttt{define} < \texttt{IDENTIFIER} > \\
&\mid \texttt{array} < \texttt{IDENTIFIER} > [] \\
&\mid \texttt{array} < \texttt{IDENTIFIER} > [< \texttt{NUMBER} >][] \\
< variable\_declarations > &::= < declaration >< variable\_declarations > \\
&\mid < declaration >
\end{aligned}
$$

作成した言語定義 (2/3)

$$
\begin{array}{rcl}
< declaration > & ::= & \texttt{define} < identifier\_list > \texttt{;} \\
& | & \texttt{array} < \texttt{IDENTIFIER} > [< \texttt{NUMBER} >]\texttt{;} \\
& | & \texttt{array} < \texttt{IDENTIFIER} > [< \texttt{NUMBER} >][< \texttt{NUMBER} >]\texttt{;} \\
< identifier\_list > & ::= & < \texttt{IDENTIFIER} > \texttt{,} < identifier\_list > \\
& | & < \texttt{IDENTIFIER} > \\
< statement\_list > & ::= & < statement >< statement\_list > \\
& | & < statement > \\
< statement > & ::= & < assignment\_statement > \\
& | & < loop\_statement > \\
& | & < selection\_statement > \\
& | & < function\_call > \\
& | & < break\_statement > \\
& | & < \texttt{IDENTIFIER} >< unary\_operator > \\
& | & < unary\_operator >< \texttt{IDENTIFIER} > \\
< assignment\_statement > & ::= & < \texttt{IDENTIFIER} > \texttt{=} < arithmetic\_expression > \texttt{;} \\
& | & < \texttt{IDENTIFIER} > \texttt{= -} < arithmetic\_expression > \texttt{;} \\
& | & < array\_reference > \texttt{=} < arithmetic\_expression > \texttt{;} \\
& | & < array\_reference > \texttt{= -} < arithmetic\_expression > \texttt{;} \\
< arithmetic\_expression > & ::= & < arithmetic\_expression >< additive\_operator > \\
& & < multiplicative\_expression > \\
& | & < multiplicative\_expression > \\
< multiplicative\_expression > & ::= & < multiplicative\_expression > \\
& & < multiplicative\_operator >< primary\_expression > \\
& | & < primary\_expression > \\
primary\_expression & ::= & < variable > \\
& | & (< arithmetic\_expression >) \\
< additive\_operator > & ::= & \texttt{+} \\
& | & \texttt{-} \\
< multiplicative\_operator > & ::= & \texttt{*} \\
& | & \texttt{/} \\
& | & \texttt{\%} \\
< unary\_operator > & ::= & \texttt{++} \\
& | & \texttt{--} \\
< variable > & ::= & \texttt{IDENTIFIER} \\
& | & \texttt{NUMBER} \\
& | & < array\_reference > \\
& | & \texttt{IDENTIFIER} < unary\_operator > \\
& | & < unary\_operator > \texttt{IDENTIFIER} \\
< array\_reference > & ::= & \texttt{IDENTIFIER}[< variable >] \\
& | & \texttt{IDENTIFIER}[< arithmetic\_expression >] \\
& | & \texttt{IDENTIFIER}[< variable >][< variable >] \\
& | & \texttt{IDENTIFIER}[< arithmetic\_expression >][< variable >] \\
& | & \texttt{IDENTIFIER}[< variable >][< arithmetic\_expression >] \\
& | & \texttt{IDENTIFIER}[< arithmetic\_expression >] \\
& & [< arithmetic\_expression >]
\end{array}
$$

作成した言語定義 (3/3)

$$
\begin{array}{rcl}
< loop\_statement > & ::= & \texttt{while}(< expression >)< statement\_list > \\
& | & \texttt{WHILE}(< expression >) < statement > \\
& | & \texttt{FOR} \\
& & (< for\_initial >< for\_expression >< for\_update >) \\
& & < statement\_list > \\
& | & \texttt{FOR} \\
& & (< for\_initial >< for\_expression >< for\_update >) \\
& & < statement > \\
< for\_initial > & ::= & < assignment\_statement > \\
& | & \texttt{;} \\
< for\_expression > & ::= & < expression > \texttt{;} \\
& | & \texttt{;} \\
< for\_update > & ::= & < \texttt{IDENTIFIER} > \texttt{=} < arithmetic\_expression > \\
& | & < array\_reference >=< arithmetic\_expression > \\
& | & \texttt{IDENTIFIER} < unary\_operator > \\
& | & < unary\_operator >< \texttt{IDENTIFIER} > \\
& | & \text{(何もないことを定義)} \\
< selection\_statement > & ::= & < if\_statement > \\
& | & < if\_statement >< else\_statement > \\
< if\_statement > & ::= & \texttt{IF}(< expression >)< statement\_list > \\
& | & \texttt{IF}(< expression >) < statement > \\
< else\_statement > & ::= & \texttt{ELSE}< statement\_list > \\
& | & \texttt{ELSE} < statement > \\
< break\_statement > & ::= & \texttt{BREAK;} \\
< expression > & ::= & < arithmetic\_expression >< comparison\_operator > \\
& & < arithmetic\_expression > \\
< comparison\_operator > & ::= & \texttt{==} \\
& | & \texttt{<} \\
& | & \texttt{>} \\
& | & \leqq \\
& | & \geqq \\
< function\_call > & ::= & \texttt{FUNCCALLIDENTIFIER}(); \\
& | & \texttt{FUNCCALLIDENTIFIER}(< parameter\_list >); \\
< parameter\_list > & ::= & < arithmetic\_expression > \texttt{,} < parameter\_list > \\
& | & < arithmetic\_expression >
\end{array}
$$

3 言語定義で受理されるプログラムの例

第 2 章で示した言語定義を用いて，第 10 章に示す 6 つの最終課題のプログラムを受理することができる．プログラムの詳細は 10 章に掲載するため，ここでは省略する．最終課題の 1 から 5 に関しては Web ページに掲載されていたプログラムが関数対応となっていなかったため，少し修正を加えている．

以下では受理できないものをいくつか示す．

3.1 受理できないプログラム

以下のようなプログラムは上記の言語定義では受理できない．

```
 ========================================================
| define a;          | define a[];       | while(1){      |
| a = 2;             |                   |                |
| define b;          |                   | }              |
|                    |                   |                |
 ========================================================
```

左から順に説明する．まず一つ目は宣言文と宣言文の間には文を入れることはできないということである．つまりプログラム内で使用する変数はすべて宣言部で宣言しきる必要が有る．

次に二つ目は，型制限である．この言語定義には array と define の二つの型しかないが，互換性はないため，配列であれば array を，それ以外であれば define を文頭に記述する必要が有る．

最後に三つめは，while 文による無限ループである．for 文において，第二式に記述をせずに無限ループさせることは可能だが，while 文においては対応していない．

上記のすべてにおいて，受理できない場合は syntax error となりプログラムが強制終了する．このほかにも対応していないプログラムは存在するが，ここでは課題に取りかかる上で，引っかかったものを抽出して掲載した．

4 コード生成の概略

4.1 コード生成とは

コード生成とは，プログラムを対象のアーキテクチャのサポートする命令に変換して出力することである．本実験では，対象とするプログラムを中間表現として抽象構文木に変換し，その後，コード生成を行っている．コード生成の結果は MIPS アセンブリ言語で出力される．これに対し，MIPS アセンブラ・シュミレータである MAPS を用いることで，コード生成が正しく行われたかどうかを判断する．

4.2 コード生成に向けて

コード生成は，主に以下の手順で行う．

- 事前に決めておくこと
  - レジスタ割り当て規則の決定
  - メモリの使用方法の決定
- 構文解析時にすること
  - 記号表の作成
- 抽象構文木の各ノードに対応するコードの出力
  - 変数領域の確保
  - 算術式
  - 代入文
  - ループ文
  - 条件分岐文
  - 関数呼び出し

最初にコード生成を伴わない “事前に決めておくこと” の二つの項目に関して説明する．

4.3 事前に決めておくこと

4.3.1 レジスタ割り当て規則の決定

今回は出力結果が MIPS アセンブリ言語となるため，概ね MIPS のレジスタ割り当てをそのまま流用することとした．しかし，実装を通じて$t7 や$t8 レジスタを一時保存先として使用したため，完全にレジスタ割り当てが MIPS のものと一致しているわけではない．また，今回はスタックを頻繁に利用したため，レジスタをあまり利用できていない．そのため，レジスタを用いた高速なデータのやり取りを生かすことができていない．また，そのため，レジスタ使用規約も$t0 から$7, および，$s0 から$s7 に対しては，特に制限がない．以下では，用途が明確に決まっているレジスタを説明していく．

最も重要なレジスタは$ra である．関数の呼び出し元に戻るためのレジスタであり，ここが書き換わるとプログラムカウンタが想定外の場所を指し，例外発生により，アセンブラ・シュミレータが停止してしまうので，関数呼び出し時には必ずスタック領域に退避させる必要がある．

$fp も$ra レジスタ同様，スタック領域への退避が必要なレジスタである．このレジスタは関数のスタック領域の端を指すものであり，書き換わる可能性のある$sp の保存先となっている．

$t8 に関しては配列のアドレスの引き渡し時に一時保存先として利用しているため，配列を読み取る部分でのコード生成時以外では使用してはならない．$t9 も同様の理由である．

$v0 と$v1 は最も汎用性が高いレジスタとなっている．二項演算の場合，多くの場合が$v0 と$v1 に値を格納して演算を行う．しかし，普通に利用すると上書きされてしまうので，スタックを利用している．他のレジスタを使用しないだけスタックへのロードストア命令が増えているため，性能は悪くなっている．実装上，多くの部分で流用ができたため，スタックを利用した手法を採用している．

その他の特殊なレジスタ ($k0 や$at など) は MIPS のレジスタ割り当ての通り，使用しないものとする．

4.3.2 メモリの使用方法の決定

メモリの使用方法はシステムプログラミングで取り扱った，手続き呼び出し規約をベースに考えた．

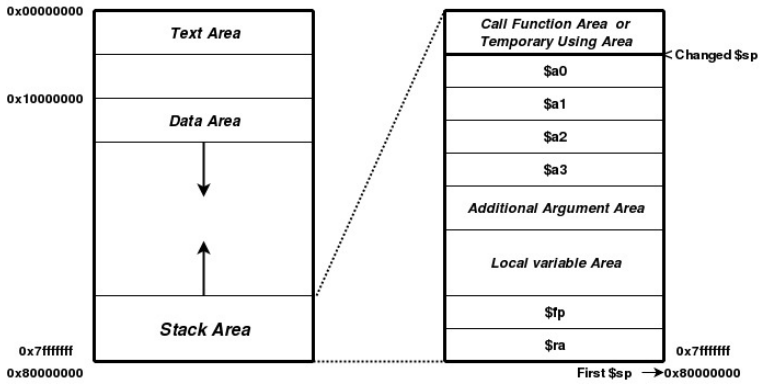

図 1: メモリの構成

図1から読み取れるように，メモリはテキスト部, データ部, スタック部の3つで構成している．0x80000000 から 0xffffffff までのアドレスはカーネルエリアとするため，本来アクセスはしない．main 関数に渡した引数を保存したり，ロードする場合にのみアクセスする．

また，図にある上下の矢印はデータ領域，スタック領域が広がっていく方向を示している．静的な領域を確保する場合は，スタック部から，動的な領域を確保する場合は，データ部からそれぞれ領域を伸ばしていく．

スタック部の詳細は図の右側に示す．スタックは関数が呼び出された時に確保され，最小を 24 バイトとし，以降は倍語長が確保されるように，8 バイト単位でスタックの確保量が変動する．関数中で宣言されたローカル変数は，図の “Local variable Area” に格納される．また，“Addtional Argument Area” は関数呼び出し時に引数が 5 つ以上だった場合に確保され，使用される領域である．そのため，引数が 4 つ以下の場合はこの領域のサイズは 0 となる．この部分に順に引数を並べることで被呼び出し側の関数は引数を順番通りにスタックに格納したり，取り出すことができる．

$a0 から$a3 の 4 つのレジスタが格納される領域は常に確保されるようにしている．このため，必ず格納される$ra と$fp の 8 バイトを合計した 24 バイトが最小スタックフレームサイズとなっている．

$a0 より上の領域は，被呼び出し関数の使用領域である．しかし，関数を呼び出すまでは使用されないため，算術式の計算結果を一時的に格納する場所として利用する．この仕様のため，被呼び出し関数は，自身

の変数領域にノイズが残ることがあるため，変数の初期化が必要となる．しかし，今回は言語定義に初期化が定義されていないため，プログラムを作成する際には，必ず値を代入した変数を使用することとなる．

4.4 構文解析時にすること

4.4.1 記号表の作成

コンパイラがコードを生成する中で，変数の情報を管理するためには記号表を用いるのがよい．これは，変数宣言時や，変数の参照の際に，名前が重複していないか，参照の際に変数がどちらの型であるか，確保するスタック領域のサイズはいくつかなどを確認するために使用する．
今回はリスト構造を用いて作成した．以下はその変数表構造体 Symbols のメンバである．

- int symno < 変数の番号 (Unique)
- char *symbolname < 変数の名前
- Stype type < 変数のタイプ (変数か引数かあるいは関数か)
- unsigned int size1 < 配列の場合，一次元のサイズ
- unsigned int size2 < 配列の場合，二次元のサイズ
- unsigned int address < 配列の場合，先頭アドレス
- struct symbols *next < 大域変数であれば，次は大域変数，局所変数であれば，同様である．
- struct symbols *branch < 関数である場合，この先が局所変数となる．

コード生成に取り組む上で段々と拡張したため，一つのメンバが複数の役割を果たしていることがある．そのため，拡張性の面から見るとあまり良くない構造体となっている．
抽象構文木を作成する上で作成したノード構造体 Node 内にしか含まれない情報もあり，コード生成に当たり，二つの構造体を対応させながらデータを取得する必要があり，複雑な動きとなっているため，可読性の観点からもよいとは言えない．今回は作成しなおす時間がなかったため，そのままとなっている．

4.5 抽象構文木の各ノードに対応するコードの出力

4.5.1 変数領域の確保

今回作成した言語定義では，受理する変数には大域変数と局所変数の 2 種類が存在する．今回作成したコンパイラでは，大域変数はメモリのデータ部に，局所変数はメモリのスタック部に格納される．今回は関数呼び出しへの対応を視野に入れて局所変数を軸にコード生成に取り組んだため，大域変数に対する挙動メモリの割り当て以外作成できていない．そのため，今回コード生成するプログラムでは，変数は局所変数のみとする．以下では，変数へのメモリ割り当てを説明する．
大域変数の場合は，配列であるかいなかで挙動が変わる．配列である場合，配列のサイズを要素数を掛けることで算出する．その求めた要素数の積に 4 バイトを掛けた値を用いて，アセンブラ指令の `.space` を用いてメモリを割り当てる．配列ではない場合，個別にアセンブラ指令の `.word` を用いてメモリ割り当てを行う．メモリの値を取り出すには，li 命令を用いてアドレスをレジスタに格納した後，lw 命令を用いてデータを取り出す．
局所変数の場合は，関数呼び出し時に確保された領域に宣言された順にメモリのアドレスが割り振られる．配列の場合は，4 バイト境界で，配列の要素数の積の分確保される．配列の格納方式には列優先方式を採用した．また，データの取り出しの際は，$fp から記号表を用いてアドレスを算出し，lw 命令を用いることでデータを取り出すことができる．データを格納する場合は，逆の手順となる．

4.5.2 算術式

算術式のコード生成の実現法には，スタックを用いる方法と 4 つ組中間表現を用いる方法の 2 通りの方法がある．スタックを用いる方法は，抽象構文木から直接コードへ変換する方法であり，4 つ組中間表現を用いる方法は抽象構文木から 4 つ組中間表現，コードの順に変換する．4 つ組中間表現には最適化に適しているという利点があるが，スタックを用いた方法の方が実装が用意であったため，今回はスタックを用いた方法を採用した．

図 1 での，被呼び出し関数の領域を一時保存領域として利用することでスタックを用いた算術式を実現している．プッシュ，ポップ操作をコンパイラプログラム中に大域変数として `stack_size` を定義し，$fp から $-4 * stack\_size$ した部分に中間データを格納することで実現している．4.3.2 項でも述べたが，この方式はメモリアクセスが増えるため性能が悪くなる．しかし，実装のしやすさと時間的都合からこちらを選択している．

4.5.3 代入文

代入文はまず左辺の変数のアドレスを得る作業を行い，その後，代入する式の部分の計算を行う．算術式と同様に，左辺の変数のアドレスはスタック領域に一時退避させておく．これは式が代入する式が複雑なものであった場合，上書きされてしまうからである．

両辺のどちらかでも配列が含まれる場合は挙動が少し複雑になる．配列の場合の処理は，配列の変数名をもとに記号表からアドレスを取り出す．このアドレスは配列の先頭アドレスであるため，このアドレスからのオフセットを計算する必要がある．ここでのオフセットとは配列の要素に割り当てられた値のことである．割り当てられた値が即値の場合，変数の場合，配列の場合でそれぞれ動作が分岐するので余計に複雑となる．

大域変数の場合は名前からアドレスの取得が容易であるため，比較的実装が簡単である．li 命令でアドレスを読み込み，オフセットを算出後，sw 命令を行えば代入文を実現することができる．

4.5.4 条件分岐

条件分岐は 3 部分からなる．式の条件判定，文集合，後続の条件節である．この内，文集合の後と，後続の条件節の末尾にジャンプ先となるラベルが必要となる．式の条件判定が真ならば文集合を実行し，偽ならば後続の条件節へと続いていく．また文集合を実行した場合は後続の条件節をスキップして末尾にジャンプする必要がある．またラベルが重複するとアセンブラ・シュミレータがエラーを出すため，ラベルは重複しないようにユニークな値としなければならない．

4.5.5 ループ文

ループ文には while 文と for 文の二種類がある．どちらも条件分岐と似たように式の条件判定部と文集合部からなる．同様にジャンプ先のラベルを適切に配置することでコード生成を行うことができる．ループ文には一つのループに三つのラベルが必要となる．先頭と条件部とループ文の外の 3 箇所である．

ループ文は部分の順番を入れ替えることでコードの実行数を減らすことができる．while 文を例に挙げると，まず条件式の部分にジャンプし，その後真ならば文集合にジャンプし，偽ならそのまま抜けるという順番である．これにより条件が偽になった場合のジャンプ命令を省略することができる．

また break 文を実装する上では，ラベル名を分けることはできない．これは break 文が一つのループを抜けるという特性をもつため，ラベル名を分けているとループの種類が増えることとなり，break 文の実行時に判定処理が必要となる．実装が困難となるため，while 文と for 文のループラベルをユニークにするための大域変数は共通としておく．

4.5.6 関数呼び出し

関数呼び出しでは，特に配列の取扱いに関して注意する必要がある．関数で配列を渡す場合，スタックを上手く利用すればまるごと引き渡すことも不可能ではないが，メモリ効率面からみてもあまり良い選択肢ではない．そのため，通常引数としての配列は先頭アドレスを渡すこととなる．これは関数呼び出しが一重の場合は問題とならないが，二重，あるいは再帰的な呼び出しをする際には，処理を工夫しなければ元の配列の先頭アドレスから，アドレスが格納されている$a レジスタのアドレスに置き換わってしまう．今回の実験ではこの処理が解決できず，関数を複数回呼び出したときのコード生成ができなかった．

5　コンパイラ作成過程で工夫した点

本実験におけるコンパイラ作成過程で工夫した点の一つはスタックを利用した局所変数の格納である．図 1 のような形をプロットとし，記号表と対応させたアドレスを用いることで実現することができた．これにより関数呼び出しの際に，引数をスタックフレームに格納するだけで受け渡しができるようになった．実装はしなかったが，return 文の実装も容易である．レジスタへの影響が小さいのであまり意識することなくレジスタを使用することができる．このスタックフレームを作成する過程で気づいたが，スタックフレーム長を調整することで返り値を複数可させることも可能であるように感じた．関数呼び出しの後，返り値用のスタック領域を確保しておくことで，2 つ目，3 つ目の返り値が実現できる．

また，今回は抽象構文木を作成していることから処理が段階的に実施される．そのため，各処理にデバッグ用の出力コードを埋め込み，`#ifdef` を利用することでデバックの切り替えをできるように工夫を行った．これによりアセンブラ出力時の関数の動きがわかりやすくなり，MAPS に通したいときには，コメントアウト一つで余分な出力が消え，シュミレートを行うことができた．これによりいくつかバグを発見することができた．このことから，デバックの大切さを再認識した．

また，抽象構文木生成関数において任意の子をもつノードを生成する `make_nchile_node` 関数を実装した．これは，第二引数のノードを親として，それ以降の引数のノードを子とする関数である．これによりノードを作成する関数の省略に成功した．

6　最終課題を解くために (定義した言語で) 書いたプログラムの概要

6.1　yacc プログラム: langspec.y

6.2　lex プログラム: langspec.l

6.3　抽象構文木作成およびコード 生成プログラム: ast.c

6.4　ast.c のヘッダ: ast.h

7　最終課題の実行結果

最終課題は 1 から 6 を行ったが，6 に関しては関数の再帰呼び出しに不具合があり，上手くいかなかった．代わりに別の関数の呼び出しを行うプログラムを作成し，単純な関数呼び出しは実装できたことが確認できた．以降に実行結果として，各最終課題のアセンブリコード 実行後のメモリの状態を記載する．

7.1　最終課題 1

ステップ数: 391

```
# tag=2047 index=255
7ffffd0: 00000000 00000000 00000001 0000000a
7ffffff0: 0000000b 00000037 00000000 00000018
###
# tag=2048 index=0
###
```

7.2　最終課題 2

ステップ数: 221

```
# tag=2047 index=255
7ffffd0: 00000000 00000000 00000001 00000005
7ffffff0: 00000006 00000078 00000000 00000018
###
# tag=2048 index=0
###
```

7.3 最終課題 3

ステップ数: 3249

```
# tag=2047 index=255
7fffffc0: 00000000 00000000 00000001 0000001e
7ffffff0: 00000008 00000004 00000002 00000010
7fffffe0: 0000001f 00000000 00000000 00000018
###
# tag=2048 index=0
###
```

7.4 最終課題 3 改良

ステップ数:　2301

```
# tag=2047 index=255
7fffffc0: 00000000 00000000 00000001 0000001e
7ffffff0: 00000008 00000004 00000002 00000010
7fffffe0: 0000001f 00000000 00000000 00000018
###
# tag=2048 index=0
###
```

7.5 最終課題 4

ステップ数: 391502 長いため，0 から 50 までとする．

```
# tag=2047 index=255
7ffff020: 00000000 00000000 00000002 7ffff050
7ffff040: 000003e8 000001f5 00000003 000003e8
7ffff050: 00000100 00000002 00000002 00000002
7ffff060: 00000001 00000002 00000001 00000002
7ffff070: 00000001 00000001 00000001 00000002
7ffff080: 00000001 00000002 00000001 00000001
7ffff090: 00000001 00000002 00000001 00000002
7ffff0a0: 00000001 00000001 00000001 00000002
7ffff0b0: 00000001 00000001 00000001 00000001
7ffff0c0: 00000001 00000002 00000001 00000002
7ffff0d0: 00000001 00000001 00000001 00000001
7ffff0e0: 00000001 00000002 00000001 00000001
7ffff0f0: 00000001 00000002 00000001 00000002
7ffff100: 00000001 00000001 00000001 00000002
```

7.6 最終課題 5

ステップ数: 1652

```
# tag=2047 index=255
7fffff90: 00000000 00000000 00000000 00000008
7fffffa0: 00000020 00000001 00000000 00000000
7fffffb0: 00000000 00000000 00000001 00000002
7fffffc0: 00000003 00000004 00000005 00000006
7fffffd0: 00000007 00000008 00000013 00000016
7fffffe0: 0000002b 00000032 00000002 00000002
7ffffff0: 00000002 00000000 00000000 00000018
###
```

7.7 最終課題 6

ステップ数: 4696

```
7ffffc0: 00000003 00000000 0000000a 00000000
7ffffd0: 00000004 00000002 00000009 00000007
7ffffe0: 7ffffd0 00000005 00000009 0000000a
7fffff0: 00000001 00000008 00000000 00000018
###
```

7.8 成功した関数呼び出しプログラム

ステップ数: 2301

```
# tag=2047 index=255
7ffffa0: 00000000 00000000 00000002 00000003
7ffffc0: 00000005 00000000 7ffffd0 000012dc
7ffffd0: 7ffffe0 7fffffec 00000000 00000000
7ffffe0: 00000003 00000002 00000001 00000003
7fffff0: 00000002 00000000 00000000 00000018
###
```

8 考察

ステップ数からわかるように非常に実行命令数が多い．これは値を取り出す一つ一つにスタックの出し入れ操作が含まれているからである．また，最終課題 3 の改良前と改良後でステップ数が 3 分の 1 程減少している．これは算術式の中に現れる変数や定数項の数が減ったためである．このことから，一つの演算子や項に対するステップ数の依存度の高さが読み取れる．そのため，最適化や適切なレジスタ操作を行わない，性能の低いコンパイラによって生成されたプログラムのコードはステップ数が増え，性能が悪化することが分かる．

9 ソースプログラムのある場所

作成したソースプログラ以下の場所に保存してある．一つ目はレポートに向けて途中で切り上げたコンパイラである．二つ目は最終課題 6 のコード生成に向けて改良を加え続けているものである．/home/users/ecs/09428526/git/C-exp/compiler/personal/ptype2

/home/users/ecs/09428526/git/C-exp/compiler/personal/sample

10 最終課題およびコード生成に成功したプログラム

10.1 最終課題 1

```
 1  func main(){
 2   define i;
 3   define sum;
 4
 5   sum = 0;
 6   i = 1;
 7   while(i < 11) {
 8      sum = sum + i;
 9      i = i + 1;
10   }
11  }
```

10.2 最終課題 2

```
 1  func main(){
 2    define i;
 3    define fact;
 4
 5    fact = 1;
 6    i = 1;
 7    while(i < 6) {
 8      fact = fact * i;
 9      i = i + 1;
10    }
11  }
```

10.3 最終課題 3

```
 1  func main(){
 2    define fizz;
 3    define buzz;
 4    define fizzbuzz;
 5    define others;
 6    define i;
 7
 8    fizz = 0;
 9    buzz = 0;
10    fizzbuzz = 0;
11    others = 0;
12    i = 1;
13    while(i < 31){
14      if ((i / 15) * 15 == i)
15        fizzbuzz = fizzbuzz + 1;
16      else if ((i / 3) * 3 == i){
17        fizz = fizz + 1;
18      }
19      else if ((i / 5) * 5 == i){
20        buzz = buzz + 1;
21      }else{
22        others = others + 1;
23      }
24
25      i = i + 1;
26    }
27  }b
```

10.4 最終課題 3(改良)

```
 1  func main(){
 2    define fizz;
 3    define buzz;
 4    define fizzbuzz;
 5    define others;
 6    define i;
 7
 8    fizz = 0;
 9    buzz = 0;
10    fizzbuzz = 0;
11    others = 0;
12    i = 1;
13    while(i < 31){
14      if (i%15 == 0)
```

```
15        fizzbuzz = fizzbuzz + 1;
16      else if (i%3 == 0){
17        fizz = fizz + 1;
18      }
19      else if (i%5 == 0){
20        buzz = buzz + 1;
21      }else{
22        others = others + 1;
23      }
24
25      i++;
26    }
27  }
```

10.5 最終課題 4

```
 1  func main(){
 2    define N;
 3    define i;
 4    define j;
 5    define k;
 6    array a[1001];
 7
 8    N = 1000;
 9    i = 1;
10    while (i <= N) {
11      a[i] = 2;
12      i = i + 1;
13    }
14
15    i = 2;
16    while( i <= N/2) {
17      j = 2;
18      while(j <= N/i){
19       k = i * j;
20       a[k] = 1;
21       j = j + 1;
22      }
23      i = i + 1;
24    }
25    a[0] = 256;
26  }
```

10.6 最終課題 5

```
 1  func main(){
 2    array matrix1[2][2];
 3    array matrix2[2][2];
 4    array matrix3[2][2];
 5
 6    define i;
 7    define j;
 8    define k;
 9
10    matrix1[0][0] = 1;
11    matrix1[0][1] = 2;
12    matrix1[1][0] = 3;
13    matrix1[1][1] = 4;
14
15    matrix2[0][0] = 5;
```

```
16    matrix2[0][1] = 6;
17    matrix2[1][0] = 7;
18    matrix2[1][1] = 8;
19
20    for(i=0;i<2;i++){
21      for(j=0;j<2;j++){
22        matrix3[i][j] = 0;
23      }
24    }
25
26    for(i=0;i<2;i++){
27      for(j=0;j<2;j++){
28        for(k=0;k<2;k++){
29          matrix3[i][j] = matrix3[i][j] + matrix1[i][k] * matrix2[k][j];
30        }
31      }
32    }
33  }
```

10.7 最終課題 6

```
 1  func quicksort(array a[], define l, define r){
 2      define v, i, j, t;
 3      define ii;
 4
 5      if (r > l){
 6          v = a[r]; i = l - 1; j = r;
 7          for(;;){
 8              while(a[++i] < v) ;
 9              while(a[--j] > v);
10              if (i >= j) break;
11              t = a[i]; a[i] = a[j]; a[j] = t;
12          }
13      t = a[i]; a[i] = a[r]; a[r] = t;
14
15      funccall quicksort(a, l, i-1);
16      funccall quicksort(a, i+1, r);
17      }
18  }
19
20  func main(){
21      array data[10];
22      data[0] = 10;
23      data[1] = 4;
24      data[2] = 2;
25      data[3] = 7;
26      data[4] = 3;
27      data[5] = 5;
28      data[6] = 9;
29      data[7] = 10;
30      data[8] = 1;
31      data[9] = 8;
32
33      funccall quicksort(data, 0, 9);
34  }
```

10.8 成功した関数呼び出しプログラム

```
1  func tmp(array a[], array b[]){
2       define d;
```

```
 3       b[0] = a[0];
 4       b[1] = a[1];
 5       d = b[0]+b[1];
 6
 7  }
 8
 9  func main(){
10       array a[3];
11       array b[2];
12       a[0] = 3;
13       a[1] = 2;
14       a[2] = 1;
15       b[0] = 0;
16       b[1] = 0;
17       funccall tmp(a,b);
18  }
```