情報工学実験C コンパイラ

氏名:大西　隼也
学籍番号:09427510

出題日: 2017 年 12 月 5 日
提出日:2018 年 2 月 6 日
締切日:2018 年 2 月 6 日

1　実験の目的

本実験では，プログラミング言語（高級言語）で書かれたプログラムを入力し，コンピュータが実行できる言語（低級言語）に変換するプログラムであるコンパイラの簡易版を作成する．

実験の目的として，yacc，lex といったプログラムジェネレータ（プログラムを作成するプログラム）を用いる経験をすること，またコンパイルの作成を通じて，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との対応について理解を深めるというものがある．

2　今回作成した言語の定義

今回，作成したコンパイラでは，受理するプログラムの関係上，基本言語仕様の拡張が必要なかったため，そのまま用いている．

```
<プログラム> ::= <変数宣言部> <文集合>
<変数宣言部> ::= <宣言文> <変数宣言部> | <宣言文>
<宣言文> ::= define <識別子>; | array <識別子> [ <数> ];
<文集合> ::=   <文> <文集合>| <文>
<文> ::= <代入文> | <ループ文> | <条件分岐文>
<代入文> ::= <識別子> = <算術式>;
<算術式> ::= <算術式> <加減演算子> <項> | <項>
<項> ::= <項> <乗除演算子> <因子> | <因子>
<因子> ::= <変数> | (<算術式>)
<加減演算子> ::= + | -
<乗除演算子> ::= * | /
<変数> ::= <識別子> | <数> | <識別子> [ < <数> ]
<ループ文> ::= while (<条件式>) { <文集合> }
<条件分岐文> ::= if (<条件式>) { <文集合> }
                          | if (<条件式>) { <文集合> } else { <文集合> }
<条件式> ::= <算術式> <比較演算子> <算術式>
<比較演算子> ::= == | '<' | '>'
```

<識別子> ::= <英字> <英数字列> | <英字>
<英数字列> ::= <英数字> <英数字列>| <英数字>
<英数字> ::= <英字> | <数字>
<数> ::= <数字> <数> | <数字>
<英字> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<数字> ::= 0|1|2|3|4|5|6|7|8|9

3 定義した言語で受理されるプログラムの例

今回定義した言語では以下のような仕様を満たすプログラムを受理可能である.

- define を用いて変数の定義が可能である.
- 定義した変数に適切な値を代入可能である.
- 変数と数字を用いた単純な四則演算の算術式を使用できる.
- while を用いたループ文を使用できる.

以下, 受理されるプログラムの例を示す.

- 最終課題 1

  1 から 10 までの数の和

```
define i;
define sum;

sum = 0;
i = 1;
while(i < 11) {
   sum = sum + i;
   i = i + 1;
}
```

- 最終課題 2

  5 の階乗の計算

```
define i;
define fact;

fact = 1;
i = 1;
while(i < 6) {
   fact = fact * i;
```

```
    i = i + 1;
}
```

4 コード生成の概略

作成した抽象構文木（AST）を元に，コード生成を行う．コード生成は，主に次のような段階で行われる．

- レジスタ割り当て規則の決定
- メモリの使用方法の決定
- 記号表の作成
- AST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

以下，各段階について詳細に述べる．

4.1 レジスタ割り当て規則

プログラムのコンパイルを行う上で，計算の際にどのレジスタを割り当てるか規則を決定する必要がある．規則を決めずにレジスタを使用すると，計算結果が途中で失われたり，場合によっては特殊な用途でのみ用いられるレジスタもあるため，不正なメモリアクセスをしてしまう恐れもあるので注意が必要である．

今回，関数の実装は行っておらず，単純な算術計算のプログラムのコンパイルだけを考えたため，使用する数種類のレジスタに対する規則を決定した．

レジスタ名	用途
$zero	定数の 0
$ra	戻りアドレス
$gp	グローバル領域へのポインタ
$sp	スタックポインタ
$t(*)	計算用に使用する一時変数 (t0-t7)
$v(*)	式の評価に用いる．システムコールに利用

4.2 メモリの使用方法

メモリの仕様についても，出力するコードのどの位置に，テキストセグメントとデータセグメントを配置するか決定する必要がある．

今回は，テキストセグメント生成後にコード末尾にデータセグメントを確保し，確保したデータセグメントに変数領域を記述する．

- 生成コードの冒頭，テキストセグメントの指定

```
INITIAL_GP = 0x10008000         # initial value of global pointer
INITIAL_SP = 0x7ffffffc         # initial value of stack pointer
stop_service = 99

.text   0x00001000
```

- 生成コード末尾のデータセグメント

```
.data   0x10004000
_i:     .word   0x0000
_fact:  .word   0x0000
```

4.3 算術式のコード生成について

算術式のコード生成については，スタックを用いた実装法と 4 つ組中間表現を用いた方法があるが，今回は最終課題に対してのみ適切なコードが出力されるよう，算術式のコード生成部分のレジスタ割り当てなどを実装したため，どちらの方法も用いていない．

5 最終課題を解くために書いたプログラムの概要

5.1 抽象構文木の作成に関して

抽象構文木の作成に関しては，基本言語定義に応じて適切なノードを生成する以下のような関数を複数個実装した．

引数として，ノードのタイプと作成するノードの数に対応するトークンをとる．戻り値として，構造体ノードを返す．

```
typedef struct node{
  NType type;
  int ivalue; // 整数の場合の値を入れる
  char* variable; // 変数の場合、変数名の文字列を入れる
  struct node *child;
  struct node *brother;
} Node;
```

```
Node* build_node1(NType t, Node* p1){
  Node *p;
  p = (Node *)malloc(sizeof(Node));
  if(p == NULL){
    yyerror("out of memory");
  }
  p->type=t;
  p->child = p1;
  p->brother = NULL;

  return p;
};
```

5.2 コード生成に関して

作成された抽象構文木に対して，以下のような関数を実装した．

```
void printstate(Node *p){
  switch(p->type){
  case PROGRAM:
    /* printf("プログラム\n"); */
    //初期化の宣言

    省略

    break;
  case UNIONS:
    // printf("変数宣言部\n");
    break;
  case DECSEN:
    // printf("宣言文\n");
    if(ma_flag == 1){
      printf("main:\n");
    ma_flag--;
    }
    //printf("\tla\t$t%d, _%s\n", count, p->child->variable);
    //printf("\tli\t$t%d,%d\n", count+1, p->child->ivalue);
    //printf("\tsw\t$t%d, 0($t%d)\n", count, count+1);
    //printf("\tnop\n");

    hensu[valnum] = p->child->variable;
    valnum++;
    break;
```

```
   中略

  default:
    printf("その他\n");
  }
}
```

6 最終課題の実行結果

最終課題の実行結果を以下に示す.
最終課題 1 は 1 から 10 までの和であるため，10 進数 55 が出力されれば良い.
最終課題 2 は 5 の階乗を計算するため，10 進数 120 が出力されれば良い.

- 最終課題 1

```
maps -i kadai1.s

loading "kadai1.s" ...
kadai1.s: No such file or directory
done.
 > run

Stop at 0xfffffffc
1327 instructions

 > memdump 0x10004000
0x10004000: 00 00 00 0b 00 00 00 37 - 00 00 00 00 00 00 00 00  .......7 ........
```

これは 16 進数の 37 であり，つまり 10 進数の 55 である.

- 最終課題 2

```
maps -i kadai1.s
loading "kadai1.s" ...
kadai1.s: No such file or directory
done.
 > run

Stop at 0xfffffffc
1197 instructions

 > memdump 0x10004000
0x10004000: 00 00 00 06 00 00 00 78 - 00 00 00 00 00 00 00 00  .......x ........
```

これは 16 進数の 78 であり，つまり 10 進数の 120 である．

7 特に工夫した点についての説明

7.1 printstate 関数について

5.2 章で一部コードを記載した printstate 関数に関して，直感的に理解しやすいように，抽象構文木のノードタイプに応じて適切な処理を行うよう，スイッチ文を用いた．

8 考察

8.1 算術式のコード生成の難しさ

今回は最終課題に対するコードを生成するために，多少強引にレジスタやメモリを実装したが，普段使っている gcc のように，文法規則を守っていれば適切なコードが生成される，更にはその最適化を行うには，考慮すべき点が非常に多いと感じた．

算術式のコード生成に焦点をあてて考えても，例えば四則演算の計算順序や前置後置演算子の処理の問題であったり，変数と数値の組み合わせのパターンを網羅する必要がある．

今回作成したコンパイラは，レジスタの割り当てをほぼ人力で行ったため，汎用性が低くとても実用に耐えるものではない．より精度の高いコンパイラの作成には，適切にレジスタやメモリの規則を決定しコーディングを行う必要があるだろう．

8.2 ペアプログラミングの有用性について

今回の実験で初めて，ペアプログラミングをしてみたが，難しい課題に対しても相談相手がいることで諦めること無く相談しながらコーディングが出来る点がよいと感じた．実際のコーディングの際も自分では気づかないスペルミスに即座に気づくことができたり，エラーチェックも多角的な視点から行うことで，スムーズに進んだ．

欠点としては，ペア間の知識の差が大きすぎると，理解の共有に時間がかかり，双方苦労することが挙げられる．

9 コンパイラのソースプログラム

今回，作成したプログラムのソースコードについて，主に基本言語仕様について記述した yacc と lex のプログラム 2 つと抽象構文木及びコード生成を行う関数を記述した c 言語のプログラム 1 つの計 3 つに分かれており，非常に膨大なページ数となるため github へのリンクと，プログラム名を記載することで割愛する．

`https://github.com/Shunya-Onishi/compila`

- C 言語のプログラム:makeast2.c
- Yacc のプログラム:edu4.y
- Lex のプログラム:edu4.l