情報工学実験コンパイラ レポート

氏名：佐藤佑太
学生番号：09425566

出題日：平成 27 年 4 月 13 日
提出日：平成 27 年 7 月 28 日
提出期限：平成 27 年 7 月 28 日

1　実験の目的

1. 言語処理系がどのように実装されるのか学ぶ
2. 大きなプログラムにおける関数の切り出しなどの設計・モジュール化
3. コンピュータにとって効率のよい計算方法、アルゴリズムの設計

これらのの三つ事柄について学習することが主な目的である.

2　言語の定義

作成した言語定義表 (BNF 記法) を、資料 1 に添付する。

私たちのチームでは、「できるだけコンパイラに優しい言語定義にする」ことについて特に意識して意識して言語定義を行った。
これは単にコンパイラの設計しやすさやコンパイラ自体をリダーブルなものにするだけでなく、そこからの機能拡張を行いやすいという柔軟性の面でのメリットもあると考えたからだ。
だが実際にはコンパイラ設計の実態を知らず、「コンパイラに優しい」設計にはほど遠かったが現実がで苦労した部分も多く、配列や関数のコード生成部分は、今回の実験では実装し切ることができなかった。

3　定義した言語で受理されるプログラムの例

定義した言語によって受理されるプログラムの例を、巻末の資料 2 に添付する。
本言語によって実行されるプログラムは、変数宣言部、関数宣言部、実行部の 3 つに分かれて記述される。
変数宣言部では、以下の構文により全ての変数の宣言を行う。

```
define <変数名>;
```

関数宣言部では、以下の構文にしたがって関数宣言を行う。

```
func <関数名>() {
  <処理内容>
}
```

これら 2 つの部分の後に、実際に手続き的に実行される処理を記述する実行部が続く。
このようにプログラムを大きく 3 つに分けることによって、コンパイラにとってメモリの確保などのメモリ管理がしやすくなると考えた。また文法的に、先読みする深さを浅くすることができるため、構文解析器の設計がしやすくなった。

4　字句解析，演算子順位構文解析・再帰下降型構文解析のそれぞれの方法の概略

4.1　字句解析

lex.c に定義されている nextToken() 関数で主要部分となる字句解析の処理を行っている。
字句解析では、大まかに以下のような流れでプログラムをトークン列に分解している。

1. ファイルから次の 1 文字を読み込み、遷移表をもとに状態を遷移する

2. 状態を確認して終了状態なら 3 の処理へ、終了状態でなければ 1 を繰り返す

3. それまでに読み込んだ文字列と、得られた文字列に合致する型情報を格納する

4. そのトークンを返り値として関数を終了する

また今回作成した lex.c に構文解析で先読みを行った場合に、nextToken() 関数によって得られるトークンを一つ戻すために ungetToken() という関数を定義している。
ungetToken() 関数が行う処理は実際には integer 型の変数を 0 から 1 にするだけで、nextToken() 関数内でこの flag の値を見て 1 なら前のトークンをそのまま return するような仕組みになっている。
この方法には、実装が簡易に行えるというメリットがあるが、デメリットも多い。例えば構文解析器で 2 つトークンを先読みする必要がある場合などには、直感的に ungetToken() を 2 つ呼ぶことでは解決することができない。
またトークンの巻き戻しが起こるのは実際には ungetToken() 関数が呼ばれたときではなく、ungetToken() 関数が呼ばれた後に nextToken() が呼ばれたときとなる。
実際に構文解析器を実装するときにはこれらの仕様を意識することは避けることができず、lex.c の独立性を下げている要因になっている。
これには、例えばある程度のトークンは構文解析中にスタックに貯めておいて、あとでそこを参照できるようにしておく、つまり呼んだ関数内で参照しているスタックポインタを一つ下げるような働きをする関数を作れば解決出来るのではないかと思った。

また今回作成した言語は、ソースコード中の空白やインデントなどは全く無視されるように設計している。
正確には、以下のようにトークンを読み込んだ際に先頭の文字が空白、改行、TAB である場合はそれらでなくなるまでスキップするような仕様になっている。

```
// トークンの先頭の文字が空白、改行、タブじゃなくなるまでスキップ！
while (i==0 && (c==' ' || c=='\n' || c=='\t')) {
  c = getc(fp);
}
```

よってインデントを用いた言語仕様にしたり、C のように if 分の一行の処理などの改行などを言語の機能として入れることは現状できなくなっている。

4.2 演算子順位構文解析

演算子順位構文解析器のプログラム oparser.c は、構文解析器でトークン列が算術式であると判断されたときに呼び出されるようになっている。
Oparser() 関数でトークンの型を解析しながら逆ポーランド記法による算術式を生成し、その node を return する仕組みになっている。なお、ここで行っているのは下向き構文解析である。

このコンパイラで用いる演算子順行列は、以下のように定義した。
この行列に従って Oparser() 関数で参照する構造体を define.h に定義している。
なお、算術式は$で挟まれていると考えて算術式の構文解析を行う。これは、算術式の先頭で$を呼んだことにして、また最後に$を読み込んだ算術式の解析を終了するというものである。
このためにトークンの種類として DOLLAR というのもを定義しておいて、それを用いることによって演算子の解析を行う。

左＼右	+,-	*,/	(	)	$
+,-	>	<	<	>	>
*,/	>	>	<	>	>
(	<	<	<	=	×
)	>	>	×	>	>
$	<	<	<	×	終了

4.3 再帰下降型構文解析

本実験で作成したコンパイラの構文解析器では、再帰下降型の構文解析を行う。
再帰下降構文解析は、解析木を根から下向きに作っていく解析方法で、言語の定義がそのままプログラムになるので実装が簡単であるというメリットがある。
ただし、算術式はこの方法で簡単に解析することはできないため、さきほど説明を行った演算子順位構文解析器を別に作成した。
再帰下降型構文解析は、字句解析で行うことができている部分を除いて言語定義に厳密に従って行われる。
文法ごとにそれぞれ関数として切り出し、そこからネストが深くなっていくように文法に従って解析関数を呼び出していく。

このとき、現在読み込んでいるトークンがどの文法に適用されるかを最初のトークンだけでは判断できない場合がある。
この問題は Back Tracking と呼ばれ、いくつかの解決方法があるが、今回は次のトークンを読み込んでから文法を判断し、ungetToken() を読んでから対応する解析関数を呼び出すことで読み込むトークンを一つ後戻りする方法をとった。

また、コンパイラにとって「エラーが起こった場所を特定する」ということが非常に重要だと考え、エラーが発生した関数と、そのときのエラーの内容を説明するエラーメッセージを表示するようにした。
このエラー処理を行う parse_error() 関数は以下のように定義した。

```
// エラー処理
void parse_error(char *error_func_name, char *error_message) {
  if (error_message == NULL) {
    error_message = "sorry, an unknow error occurs...";
  }
  fprintf(stderr, "error in %s: %s\n", error_func_name, error_message);
  exit(1);
}
```

この関数は一つ目にエラーが起こった関数名、二つ目にそのときのエラー内容を格納した文字列を引数に取り、その内容を表示してプログラムを終了する。
ただこのエラー処理には問題点があり、字句解析中のエラー処理とは全く別のものになってしまうということや、新たに小さな単位での構文解析処理を行う関数が呼ばれた場合にはエラーを起こした関数名が上書きされて親となる関数を特定することができないなどの問題がある。

前者の問題は、新たに字句解析器用のエラー処理関数を作成するか、または構文解析と連動してエラーメッセージを表示するために共通するエラー処理関数を作成するなどの方法で解決することができると考えられる。

後者の問題は、parse error が起こったときに逐一それが起こった関数名を渡せば表面的には解決する。だが

DRY 的にソフトウェアを設計するならより柔軟性のある関数を実装するべきだろう。例えば、スタックを用いて構文解析の経路を記憶しておいて、parse error が起こったときにスタックのデータをいくつか pop して表示することで、プログラム作成者はどのように解析が行われ、どこで syntax エラーとなっているか把握しやすくなると思われる。

5　コード生成の概略

5.1　メモリの使い方

本言語ではグローバル変数のみを用いるようにしている。グローバル変数はデータセグメントに領域を確保し、変数名の前にラベルをつけるようにしている。
parser.c 内に定義している gen_code_data_segment() 関数を、プログラム全体を解析するプログラムの最後に呼ぶことにより以下のコードをアセンブリ言語プログラムを生成する。
サンプルプログラムのひとつである、1 から 10 までの数値を足すプログラムだと、データセグメント領域に変数用のメモリを確保する処理は以下のように生成される。

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

5.2　レジスタの使い方

算術式の計算には、破壊してもよい t 系レジスタを用いている。今回は$t0〜$t6 までを実際の計算用に用いることとしている。
また、$t7 はデータを読み込むためのレジスタとして、$t8 を条件式の判定用の変数として確保している。

5.3　算術式のコード生成の方法

算術式のコード生成は、gen.c 内の関数で行うように統一している
形式としては四つ組中間表現の方法に近く、gen_code_operation() 関数を算術式の解析器 oparser.c から呼び出し引数として使用するべき値や変数の文字列と、token の演算子の型を情報として渡す。
それを受け取った gen_code_operation() 関数は使うべき$t レジスタの探索を行う。それに従って gen.c 内の gen_operation() を呼び出し、gen_operation() 関数が token の演算子の型で条件分岐してソース言語の算術式に対応するアセンブリ言語を標準出力に渡すような処理の流れになっている。

5.4　特に工夫した点についての説明

ループ文でのラベルの使い方に苦労した。プログラム中でループ文のネストなどがある場合に、ラベルの名前を各ループ命令に合わせて生成しなければいけない。
今回のコンパイラでは、L1,L2,、トップのループ文の始まりが L1 に対応し、そのループ文が終了する場所が L2 に対応するようになっている。
以降はこのルールに従い、L3 は L4 に、L4 は L5 に対応するようにラベルを生成している。

また、Linux における標準出力と標準エラー出力を使い分けることも重視した。

今回の私のコンパイラでは、外部ファイルにアセンブリ言語を書き込むわけではなく、生成したターゲットコードは全て標準出力に渡している。ファイルに保存したい場合などはこれをリダイレクトすればよいのだが、この方法では最善ではないように思われる。
この仕様のため、エラーや wargning のメッセージは、標準出力とはきちんと切り離して標準エラー出力に渡す必要がある。
また、デバッグメッセージも表示できるようにするには、$./esp$ コマンドにオプションとして $-d$ や $--debug$ などでデバッグメッセージを表示しながらコンパイルを行えるようにしたいと思った。これから取り組んでいきたい。

6 ソースプログラムについて

6.1 リポジトリ

以下のレポジトリで、本実験で作成したコンパイラのソースコードを管理している。
http://jikken1.arc.cs.okayama-u.ac.jp/gitbucket/09425566/compiler

6.2 実行方法

サンプルのプログラムである、1 から 10 の数値を足すプログラムは以下のようにすれば実行できる。

```
$ git clone http://jikken1.arc.cs.okayama-u.ac.jp/gitbucket/09425566/compiler
$ cd ./compiler
$ make
$ ./esp sample-programs/one2ten.esp
```

7 最終課題を解くために書いたプログラムの概要

本実験で作成した言語で実装した、1 から 10 までの数を足すプログラムは以下のようになる。

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

1,2 行目で本プログラムで使う変数 i と sum を定義し、6〜9 行目で while 文を用いて 1〜10 の和を計算している。
文法は C 言語を基本としたが、変数宣言に define を使い、変数宣言を行うことのできる場所はプログラムの冒頭のみとした。これによりコンパイラが構文解析しやすくなるのと、メモリの確保も行いやすくなる。

今回関数を実装することはできなかったが変数宣言部の後に関数の宣言部が続き、func ¡関数名¿とすることで宣言できるようにした。
また、関数を呼び出す処理を、もともとは ¡関数名¿(¡引数¿) と定義していたのだが、構文解析で算術式との判断をつけることが難しいことなどから構文解析中に問題が発生することに気づき、関数呼び出しの構文ををcall ¡関数名¿(¡引数¿) に変更した。

8　最終課題の実行結果

1. 1 から 10 までの数の和
    - プログラム：one2ten.esp
    - 実行結果：55〜(0x10004004)
    - ステップ数：360 instructions
2. 階乗の計算
    - プログラム：fact.esp
    - 実行結果：120〜(0x10004004)
    - ステップ数：195 instructions
3. エラトステネスのふるい関数のコード生成部分を作成できなかったため実現できなかった。
4. 行列積の計算配列を扱う部分のコード生成処理を作成できなかったため実現できなかった。
5. クイックソート
   ローカル変数を作成出来なかったため，作成出来なかった.

9　考察

本実験を通して、コンパイラの設計において、またソフトウェアの設計において特に重要なのが、モジュールごとの疎結合性だということを学んだ。疎結合性の高い設計をすることによって、不具合を起こしている場所の特定や、モジュールや関数の再利用を容易に行うことができ、信頼性も生産性も高まる。
私がコンパイラを設計する上でもこの疎結合性をできるだけ保てるように設計をした。
特に、以下の点に気をつけた。

- コンパイラの機能ごとにファイルを分割する
- 関数を細分化する

まず、コンパイラの機能ごとにファイルを分割する、ということについて。
関数をどこで分けるかという設計も難しいのだが、ファイルをどのように分けて一つのプログラムとするかの境界も案外難しい。
コンパイラを授業で学習した時には、字句解析、構文解析、コード生成、最適化という順番でターゲット言語のプログラムを得ることができると学んだが、今回作成したコンパイラや、また多くの言語はどの処理もタイミングが入り組んでいることばかりだ。
今回作成したコンパイラは、構文解析のプログムを軸として、構文解析器が新しいトークンが必要になった時に字句解析器を呼び、一通りの構文解析が終わったらコードを生成する、といった処理の流れになっている。このように処理が入り混じる中でそれぞれの処理を独立させるために、ファイルを分割することが役にたつ。
今回作成したコンパイラも、字句解析器をまず完全に完成させてから、構文解析器の実装に移るような流れになっていたため、字句解析器の独立性はかなり高いものになっていると感じる。
だが現状私のプログラムでは、構文解析器がコード生成を行う役割も持っていて、かなり疎結合性が低くなってしまっている。
これを解決するために、コード生成を行うパーツの独立性をあげようと試みた。算術式の構文解析器については oparser.c の中で行っていたコード生成を gen.c というファイルとして切り出す作業を行った。parser.c につ

いては十分な時間を設けることができなかったが、こちらのコード生成の処理も gen.c で全て定義できるようにしたいと思っている。

次に、関数を細分化するということについて。
Google の LVM コミッタである植山氏の開発した C コンパイラの 8cc のソースコードを見ると、ひとつひとつの関数が非常に小さい単位で細分化されていることがわかる。
ただ、関数をまとめることは処理の流れをわかりにくくすることにもつながるのではないかという疑問も残る。だからこそ手続き型言語での大きいプログラムの設計に慣れていないと、一つの関数に大きな処理を任せがちなのではないかと推測できる。
だが、大きな関数を作ることよりも小さな関数が疎結合され大きな処理を行うようなプログラムを設計することの方がメリットが大きいため、私たちがソフトウェアの設計・開発を、特にチームや OSS のプロジェクトとして行う上では、後者を意識する必要があるだろう。

10 巻末資料

10.1 資料 1: 言語定義表 (BNF)

```
<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>;| define <配列宣言>;
<配列宣言>::= <識別子>[<整数>]|<配列宣言>[<整数>]
<文集合> ::=  <文> <文集合>| <文>
<文> ::= <代入文> | <配列代入文> | <ループ文> | <条件分岐文> | <関数宣言文> | <関数>;
<代入文> ::= <識別子> = <算術式>; | <識別子> = call <関数>;
<配列代入文> ::= <配列> = <算術式>; | <配列> = call <関数>;
<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> | <配列>
<引数> ::= <変数>,<引数> | <変数>
<関数> ::= <識別子>(<引数>) |  <識別子>()
<関数宣言文> ::= func <識別子>(<引数>) { <文集合> } | func <識別子>(){ <文集合> }
|func <識別子>(<引数>) {<文集合> return <変数>;}|func <識別子>(){<文集合> return <変数>;}
<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><数字> | <0 以外>
<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
<整数> ::= <数>|0
<数字> ::= 0|1|2|3|4|5|6|7|8|9
<0 以外> ::= 1|2|3|4|5|6|7|8|9
<配列> ::= <識別子>[<変数>]|<配列>[<変数>]
```

10.2 資料 2: 本言語で実行されるプログラムの例

```
1  define array[3];
2  define array2[2][4];
3  define a;
4  define i;
5  define flag;
6
7  func resetArray() {
8    i = 0;
9    while (i<3) {
10     array[i] = 0;
11     i = i + 1;
12   }
13   return 1;
14 }
15
16 flag = call resetArray();
17 array2[0][flag] = 100;
18
19 if (flag == 1) {
20   array[1] = 1;
21 }else {
22   array[2] = 1;
23 }
```