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、実際には 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() が呼ばれた後になる。
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。

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、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。

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。

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

4.3 再帰下降型構文解析

算術式以外を解析するために用いる方法．言語の定義をそのままプログラムに出来るのでわかりやすい．
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...";
```

```
  }
  // printf("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、スタックに、行っている構文解析の経路を積んでいき記憶しておいて、parse error が起こったときにいくつかスタックから pop して表示するなどの方法だとプログラム作成者にとってエラーの原因を理解しやすくなるのではないだろうか。

5　コード生成の概略

5.1　メモリの使い方

5.2　レジスタの使い方

- $t0 〜 $t6 算術式の解析用
- $t7 la，li などのデータ読み込み用
- $t8 条件式の判定用
- $t9 代入文に置いて配列が代入先になったとき用

5.3　算術式のコード生成の方法

四つ組中間表現を用いている.

5.4　特に工夫した点についての説明

数値または識別子をを一つだけ代入する場合は，直で$v0 に値が代入されるようになっている.
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。

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の実行方法は以下のようになる。

```
$ 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

基本的な書き方は C 言語に近い．しかし，実装されていない機能が多く存在しているため，書き方に制限がかけられている．データの型は int 型しか存在せず，関数は関数名の前に fun とつける必要がある．配列の括弧の中には算術式が入るように再帰的に処理している．

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プログラムを分割する
- 関数を細分化する

まず、コンパイラの機能ごとにプログラムを分割する、ということについて。

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 }
```

11 参考・出典

https://github.com/rui314/8cc