情報科学演習Ｄ　課題4レポート

担当教員：松本　真佑
提出者：市川　達大
学籍番号：09B17010
メールアドレス：u243952e@ecs.osaka-u.ac.jp

提出年月日：2021年2月13日

1　課題の目的

第 1 回〜第 3 回までの課題の結果をもとに、該当コードの cas コードを生成するコンパイラを作成する。

2　課題の達成の方針と設計

2.1　外部仕様

課題 1 で字句解析した結果ファイル (ts ファイル) を読み込み、構文解析、意味解析を行いながら、cas コードを生成する。cas コードは BNF メソッドに一対一に対応する。また、構文的、意味的な誤りを見つけた場合、見つけた時点でエラーを出力しプログラムを終了する。

2.2　方針と設計

以上の課題を達成するために主に以下の方針で設計し実装する。今回の課題を実現するために以下の cas ファイルの生成方法を考え、その cas ファイルをしかるべきタイミングで生成していくことで課題を実現する。

1. 変数の参照
2. 演算
3. 値の代入
4. 文字列の出力
5. 分岐、制御文
6. 副プログラム宣言

上記に該当する cas ファイルを生成することで、全ての BNF メソッドを記述することができる。例えば「i<5 and true」という「式」も「変数, 定数の参照」と「演算」で表すことができる。

2.2.1　変数のメモリ割り当て

上記の cas ファイルの生成方法を考える前に、まずは変数のメモリ割り上げ方法を考える。今回は配布された cas ファイルをもとに、各変数はアドレス「変数の数＋何番目の変数 -1」の場所に格納し保存する。例えば、nomal04.pas の場合で考える。

```
1 program testBasic;
2 var i: integer;
3       a: array[1..10] of integer;
4
5 begin
6       i := 11;
7       a[5] := i;
8       writeln(a[5]);
9 end.
```

まず, 変数は int 型変数「i」と int 型配列「a」があり、配列は 1 つの要素を 1 つの変数と考えるので変数の総数は 11 となる。よって、変数 i の値は 1 番目の変数なので、アドレス「変数の数＋何番目の変数 -1」より、アドレス「11 + 1 - 1」の場所に格納する。同様にして、配列の要素 a[5] は 6 番目の変数なのでアドレス「11 + 6 -1」にデータが保存される。基本的な考えとしては、アドレス「変数の総数」を基準とし（今回の場合 11）, そのアドレスから何番先のアドレスかで何番目の変数の情報であるかを判別する。

2.2.2 変数の参照

上記 [2.2.1] の変数のメモリ割り当て法をもとに、変数が参照された時に該当メモリの値を取りにいくことで、変数の参照を実現でき、この変数の参照が全ての cas ファイル生成の基本となる。具体的に上記 [nomal04.pas] で i を参照する際、以下のコードで変数を参照する。

```
1 LD GR2, =0 ; assign var (i)
```

2.2.3 演算

演算はスタックを用いて、実現する。2 つの非演算子をスタックへプッシュし、演算命令を呼び出す直前で 2 数をポップして演算を実行する。3 項以上の演算の場合、一気に演算することはできないので、「乗法演算¿加法演算¿関係演算子」の規則にしたがって、前のこうから順番に push していき、かく演算子の非演算子が 2 項とも PUSH された段階で演算を実行する。ここで、問題になってくるのが「i ¡ a + b * 2」のような式の場合、非演算子が 2 項 PUSH されたのをどうのように判定するかであるが、この評価方法は下記「式の評価方法」で後述する。

2.2.4 値の代入

代入文 [左辺 := 右辺] はまず右辺を評価してスタックへプッシュし、その結果を左辺の変数に割り当てたメモリに格納することで実現する。

2.2.5 文字列の出力

文字列の出力は予め出力する文字列を DC 命令で保存しておき、出力命令が呼び出された時、保存したものを出力することで実現する。

2.2.6 分岐、制御文

分岐、制御分は条件の「式」を比較演算を行い比較して、フラグレジスタの結果をもとに分岐を生成することで実現する。各関係演算子における比較 cas コードはそれぞれ以下の方針で生成する。

1. = 「GR1,GR2 を比較後、その値が 0 の場合 true」
2. <> 「GR1,GR2 を比較後、その値が 0 でない場合 true」
3. < 「GR1,GR2 を比較後、その値が負であれば true」

4. <= 「GR1,GR2 を比較後、その値が正であれば false」

5. >= 「GR1,GR2 を比較後、その値が負であれば false」

6. > 「GR1,GR2 を比較後、その値が正であれば true」

2.2.7 副プログラム宣言

副プログラムの cas コードはプログラムの実行部分とは別に生成し、プログラムの最後で実行部分の後に付け加えることで実現する。仮引数がある場合、スタックから受け渡された引数を仮引数に設定したメモリ上に格納し、スタックで引き渡された引数はスタックから削除することで、引数の受け渡しも実現する。

3 プログラムの実装方法

上記の設計方針は主に pas プログラムをどのように cas ファイルに対応させていくかを記述したが、この章ではどのタイミングで cas ファイルを生成していくかについて説明する。

3.1 変数の参照

まず、変数の参照であるが、プログラム中で変数が参照された時点で、該当メモリの値を呼び出し、スタックへプッシュする cas コードを書き出す。これは変数を参照する場合というのは「代入分の右辺」か「式」または「入出力分」であるので、参照した後に何かしらの演算を行う場合が多く、演算はスタックを用いて行うのでスタックに PUSH しておく。具体的には以下のように「因子」内で変数が参照された時点で該当 cas コードを書き出す。

```
1  因子 (){
2  .
3  .
4  //変数出力の cas コードを書き込み
5  if(array_flag == 1) {
6          main_part.append("\t" + "POP" + "\t" +"␣GR2" + "\n");
7          if(num_variable > 1) {
8                  main_part.append("\t" + "ADDA" + "\t" +"␣GR2,␣="+ String.valueOf(
                       num_variable−2) + "\n");
9          }
10         else if(num_variable == 1) {
11                 main_part.append("\t" + "ADDA" + "\t" +"␣GR2,␣="+ String.valueOf(
                       num_variable−1) + "\n");
12                 main_part.append("\t" + "SUBA" + "\t" +"␣GR2,␣="+ 1 + "\n");
13         }
14 }
15 //配列じゃない
16 else if(array_flag == 0) {
17         main_part.append("\t" + "LD" + "\t" +"␣GR2,␣" + "=" + String.valueOf(
               num_variable−1) + "\n");
18 }
19 //変数の cas ファイル書き出し
20 set_variable();
21 .
22 .
23 }
```

3.2 演算

演算の cas コード生成は、非演算子が 2 つスタックへプッシュされた時点でその演算を実行する。具体的には「式」中で演算子が出現した際、フラッグを用いてどの演算子が出てきたかを管理し、同じ「式」中でもう一度その演算子の出現判定を行う時点で、該当演算子の非演算子が 2 つともスタックへ PUSH されたと判定し、演算を実行する。具体的には以下のような流れである。

```
式 {
while(){
        while(){
                while(){
                        (1)(3)因子
                        (2)(4)乗法演算子 ∗,div,/,and
                }
                加法演算子 +,−,or
        }
        関係演算子 =,<>,<=,<,>,>=
}
}
```

上記の式の流れから、例えば「a ∗ b」という演算を行う時、まずプログラムはまず a を読み込むので、a のメモリを参照し、a の値をスタックへプッシュする。次に ∗ を読み込むので、この時例えば ∗ が出現したことを管理する mul_flag の値 1 にすることで、演算子 ∗ が出現したという情報を保存する。そして次の b を参照した時、b の値をスタックへプッシュし、プログラムが「乗法演算子」があるかどうかの判定位置にきた時、非演算子が 2 つスタックへプッシュされたと判定した良いので、その時点で該当する演算の cas コードを書き出す。これをすることで「因子」内で「(式)」が呼び出された場合でも、同様の方法で問題なく cas ファイルの生成ができる。

3.3 値の代入

値の代入では、右辺を評価した値をスタックへプッシュし、左辺の変数のメモリ位置に評価値を格納することで、代入の cas コードを実現する。具体的には以下のようなコードの流れになる。

```
代入文

左辺 ←左辺のアドレス位置を保存
↓
:=
↓
右辺 ←右辺を評価して計算
//右辺を評価する cas コードを生成し結果をスタックへプッシュ
↓
//左辺のアドレス位置に右辺の結果を格納する cas コード生成
```

3.4 文字列の出力

文字列の出力は writeln 文から呼び出された「式」の時、式の中身が文字列である場合、DC 命令で文字列を格納し、該当箇所の cas コードは DC 命令でつけたラベルを出力することで、文字列の出力を実現する。具体的には以下のようになる。

```
基本文 |
.
.
入出力文
```

```
(1)writeln(式)←writeln から呼びださえたことをフラッグで管理。
.
(3)//出力 cas コード生成
.
.
|

式 |
.
.
(2)文字列 ←DC 命令で格納する cas コード生成
|
```

3.5 分岐、制御分

分岐、制御文は if や while からの呼び出しの際、条件式に当たる「式」を評価し、その値によって JZE や JPL などの分岐命令を生成することで、分岐を表現する。関係演算子ごとの分岐はそれぞれ [2.2.6] で示した、規則に基づき行う。ここでは、「¡=」や「¿=」の場合、比較結果が正か負かを一意に決定するために、「¡=」であれば「左辺が右辺より大きかったら false」、「¿=」であれば「右辺が左辺より大きかったら false」と判定することで、比較結果を一意に定めるようにしている。cas コード生成手順は以下のようになる。

```
文の並び |
.
.
if (1)式
↓
(2)//条件分岐 cas コードを追加
main_part.append("\t" + "POP" + "\t" + "GR1" + "\n");
main_part.append("\t" + "CPA" + "\t" + "GR1,␣=#FFFF" + "\n");
main_part.append("\t" + "JZE" + "\t" + "ELSE" + String.valueOf(while_index) + "\n");

then
↓
(3)複合分
.
while (1)式
↓
//while 文のラベルを追加
(2)main_part.append("LOOP" + String.valueOf(while_index) + "\t" + "NOP" + "\n");
↓
(3)式
↓
(4)ループ終了条件cas コード追加
main_part.append("\t" + "JUMP" + "\t" + "LOOP" + String.valueOf(while_index) + "\n");
main_part.append("ENDLP" + String.valueOf(while_index) + "\t" + "NOP" + "\n");
.
|
```

3.6 副プログラム宣言

副プログラム宣言分部分は、副プログラム宣言が宣言された時点でラベルを作成して、cas コードを生成するが、プログラムの実行部分と分けて cas ファイルを作成する必要があるので、予め cas ファイルの宣言部分を 1 次的に格納する StringBuffer クラスの変数を作成して、その変数に記述していきプログラムの実行部分の cas コードが全て生成できた時点で結合する。副プログラム宣言の cas コード生成の流れは以下のようになる。また副プログラムの仮引数は呼び出し前にスタッ

クへプッシュされており、呼び出し後、スタックポインタを用いて受け渡された引数を pop し、仮引数に割り当てたメモリへ格納することで仮引数への値の受け渡しを行う。受け渡した後は必要ないので、スタックから削除する。

```
副プログラム宣言頭部 |
.
//副プログラムの数カウント
procedure_num ++;
(1)//副プログラムのラベル cas コード生成
main_part.append("PROC" + String.valueOf(procedure_num) + "\t" + "NOP"+ "\n");
.
.
(2)//仮引数の cas コード生成
//スタックポインタの取り出し
main_part.append("\t" + "LD" + "\t" + "GR1,␣GR8" + "\n");
main_part.append("\t" + "ADDA" + "\t" + "GR1,␣=1" + "\n");
//引数を取得
main_part.append("\t" + "LD" + "\t" + "GR2,␣0,␣GR1" + "\n");
main_part.append("\t" + "LD" + "\t" + "GR4,␣=" + String.valueOf(num_of_var − 1 − i) +"\
    n");
main_part.append("\t" + "ST" + "\t" + "GR2,␣VAR,␣GR4" + "\n");
main_part.append("\t" + "SUBA" + "\t" + "GR1,␣=1" + "\n");
//スタック内の引数削除
main_part.append("\t" + "LD" + "\t" + "GR1,␣0,␣GR8" + "\n");
main_part.append("\t" + "ADDA" + "\t" + "GR8,␣=1" + "\n");
main_part.append("\t" + "ST" + "\t" + "GR1,␣0,␣GR8" + "\n");
.
.
|
(3)副プログラム動作部分のcas ファイル生成
変数宣言 |
.
|
複合文 |
.
|
```

4 本課題のポイント

ここからは本課題の要点となる部分について説明していく。

4.1 変数の代入方法

変数の代入は以下の 3 つのステップによって行う。

1. 左辺のメモリ割り当て位置の取得
2. 右辺の評価
3. 右辺の評価値をスタックへ PUSH
4. スタックから POP した値を 1 で取得したメモリ割り当て位置に ST

4.1.1 左辺のメモリ割り当て位置の取得

まず始めに左辺のメモリ割り当て位置を取得する必要があるが、これはメモリ割り当て規則にしたがって、左辺が何番目に宣言された変数であるかが分かればいい。よってここでは課題 3 で作成した記号表から該当変数が何番目にに宣言された変数であるかを numvariable に保存しておく。以下に該当箇所のプログラムを示す。

```
1  //まず変数名があるかどうかを確認する。
2                  if(is_block_has_var(index)) {
3                          Variable_table variable = code_block[variable_place].var_search(
                               get_name_list.get(index));
4                          left_var_name = variable.name;
5                          left_name_type = variable.name_type;
6                          left_data_type = variable.data_type;
7                          num_variable = variable.num_of_var;
8                          //配列の場合
9                          if(left_name_type.equals("array")) {
10                                 left_array_min = variable.array_min;
11                                 left_array_max = variable.array_max;
12                         }
13                 }
```

4.1.2 右辺の評価, 右辺の評価値をスタックへ PUSH

右辺の評価は次に説明する「式の処理方法」と同じなので説明は省略する。また評価後、評価値をスタックへ PUSH する必要があるので、代入分の右辺から「式」が呼ばれたことを管理するフラッグを用意して、そのフラッグが立っている時に式を呼び出した時のみスタックへ PUSH することで実現できる。

4.1.3 スタックから POP した値を 1 で取得したメモリ割り当て位置に ST

右辺の式を評価し終え、スタックへ値を保持したら最後に左辺のメモリへ割り当てる cas コードを生成する必要がある。これは上記で取得した、「何番目の変数であるか」をもとに cas コードを生成する。該当コードをいかに示す。

```
1  931  //代入分の左辺
2  932  //配列の場合
3  933  if(array_flag == 1) {
4  934   if(num_variable > 1) {
5  935    main_part.append("\t" + "ADDA" + "\t" +"␣GR3,␣="+ String.valueOf(num_variable−2)
       + "\n");
6  936   }
7  937   else if(num_variable == 1) {
8  938    main_part.append("\t" + "ADDA" + "\t" +"␣GR3,␣="+ String.valueOf(num_variable−1)
       + "\n");
9  939    main_part.append("\t" + "SUBA" + "\t" +"␣GR3,␣="+ 1 + "\n");
10 940   }
11 941   sub_left(array_flag);
12 942   array_flag = 0;
13 943
14 944  }
15 945  //配列じゃない場合
16 946  else {
17 947   main_part.append("\t" + "LD" + "\t" + "GR2,␣="+ String.valueOf(num_variable−1) + "
       \n");
18 948   sub_left(array_flag);
19 949  }
```

4.2 式の処理方法

1 つの式の中では基本的に「演算子」と「非演算子」が交互に登場する。例外として、「a - 2 * (-i * 2 + 3)　¡ 10 + a」のように「式」中の因子で「(式)」が呼び出された際、*と-が連続して登場することがあるが、これは一見演算子が連続しているように見えるがこの時の-は演算子ではなくi を修飾する符合のため、演算子が連続することはない。よってこの規則に基づき、非演算子を読み込み次第スタックへ PUSH していき 2 つ目の非演算子がスタックへ PUSH された時、四則演算の規則に基づき、演算子が乗法演算子であれば演算を実行、演算子が「加法演算子」であれば、次の演算子が「乗法演算子」以外の場合実行 (上記の例 「a - 2 *」の場合後ろの乗法を先に処理する必要があるのでまだ、前の演算子「-」はまだ実行しない。)、演算子が「関係演算子」の場合、次の演算子が「関係演算子」または存在しない場合のみ実行 (上記の例の場合,「..3) ¡ 10 + a」では、関係演算子＜の次の演算子が「+」なので先に後ろの演算を行ってから「＜」を評価する) する。 この規則に基づきながら、演算子の 2 つの非演算子が PUSH された時点で演算を実行することで式を評価していく。またこの規則は同じ式内でしか有効ではない。つまり、式の途中の「因子」で「(式)」が呼び出された時、この「(式)」は因子の呼び出しもとの式とは別の式だと考えるので、まずこの「(式)」を優先的に評価する。以上から、式の処理方法には以下 2 つのルールがある。

1. 「乗法演算子」＞「加法演算子」＞ 「関係演算子」の順に演算を実行
2. 生成された時間が遅い「式」から評価

4.3 手続きの呼び出し方法

手続きを呼び出す方法は、副プログラム宣言で生成した、cas コードに該当するラベルを CALL すれば呼び出すことができる。この時、引数がある場合、手続きを呼び出す前にスタックへ PUSH しておき、手続きないで現在のスタックポインタの値を取得して引数へ保存するように設定しておくことで引数の受け渡しを実現する。

4.4 分岐の処理方法

分岐の処理は 2.2.6、3.5 でも述べたように使われている関係演算子をもとに条件式の cas コードを生成し、その真偽によって分岐内の処理を行うかどうかを決定する cas コードを生成する。例として、下記コードに該当する cas コードを示す。

```
1  if f = true then
2      begin
3          writeln('then');
4      end;
5
6  cas コード
7          PUSH #0000
8          POP GR2
9          POP GR1
10         CPA GR1, GR2
11         JZE TRUE0
12         LD GR1, =#FFFF
13         JUMP BOTH0
14 TRUE0 LD GR1, =#0000
15 BOTH0 PUSH 0, GR1
16         POP GR1
17         CPA GR1, =#FFFF
18         JZE ELSE0
```

```
19          LD GR1, =4
20          PUSH 0, GR1
21          LAD GR2, CHAR0
22          PUSH 0, GR2
23          POP GR2
24          POP GR1
25          CALL WRTSTR
26          CALL WRTLN
27  ELSE0 NOP
```

上記のように条件式→分岐→分岐内処理の順番で cas コードを生成する。

4.5 レジスタやメモリの利用方法

レジスタの方法については, 以下のレジスタを使用した。

1. GR1,GR2 演算用
2. GR3 配列の添字の式の評価値を格納する
3. GR8 スタックポインタ

基本的には GR1,GR2 とスタックを用いることで演算を処理できるが、配列の添字の処理は GR3 に保存することでスタックの流れを無視でき、実装を容易にした。

4.6 ラベルの管理方法

ラベルの管理は「副プログラム」「if 文」「while 文」で管理する必要があるが、まず「副プログラム」のラベルはその副プログラムが何番目に宣言されたものかを意味解析時点で記号表に格納しておき、副プログラム宣言の cas コード生成時にその値を”PROC”+”何番目”といった名前をつけることで重複することなくラベルを一意に定めることができる。次に「if 文」「while 文」でのラベル作成は構文解析時に if 文と while 文が合計何回出現したかを記録するカウンタを用意してそのカウンタを使用してラベルをつけることで一意にラベルを指定できる。ここでは if 文と while 文の出現回数を区別しないので、ラベルが飛び飛びの値を刻むことがあるので、若干微妙な実装ではあるが、実装の簡単さを優先してこの方法をとった。

4.7 スコープの管理方法

スコープの管理方法は、そもそも変数の参照方法が意味解析で作成した記号表をもとに、「何番目に宣言された変数か」で変数を参照するので、cas コード生成時点で特に意識することはなかった。

4.8 課題１３の再利用

今回のコンパイラの作成を通して、基本的には課題１３を再利用して作成することができた。これは構文解析時点でしっかりと BNF メソッドを作成して、それに基づきながらコードを作成して行ったので、基本的には前課題に追加部分を実装する形でプログラムを作成することができた。しかしながら、コンパイラを作成する際、様々な動きを追いながら開発する必要があったので、構文解析時点絵で構文解析木を作成しておけば、もう少し簡単に実装できたと思う。

5　まとめ

今回で Pascal 風言語のコンパイラが完成したが、今回の課題では主にデバッグの方法について勉強になった。今回の課題ほど複雑で様々な処理を行うプログラムの場合、デバッグをいかに工夫するかが重要になってくる。今回は小さい機能を追加してはデバッグという方法をとり、できるだけ細かくテストを行うことで、エラーの原因を特定しやすいように工夫した。また、何度も使うコードや、似たような種類のコードをメソッド化、クラス化することの重大さを改めて実感した。このようにメソッドやクラスを適切に使用し、オブジェクト指向に基づくことで、プログラムの可読性、保守性はかなり上がるのでこれからも、適切な「オブジェクト指向」を心がけたい。

6　感想

今回の課題 4 は処理が非常に複雑で、かつ適切なタイミングで適切な cas ファイルを生成する必要があったので非常に難しく感じた。それに加えて、実行結果が cas ファイルであり、どの部分にエラーがあるかが非常に読みづらかったので課題 3 までよりてこづった。今回でデバッグやおる程度大きな規模のコードの書き方を学べたので課題全体を通して非常に学びのある講義だった。