情報科学科演習 D
課題 4 + 発展課題

基礎工学部情報科学科ソフトウェア科学コース
学籍番号: 09B20001

イグレシウスエドゥアルド

2022 年 01 月 26 日

1　システムの仕様

1. コンパイラは Compiler.run(String, String) から実行される。
2. プログラムの入力としてはトークンごとに分割されたデータパス（ts ファイル）と CASL2 でプログラム記述したいプログラム（CAS ファイル）のデータパスである。
3. プログラムの出力としては、一つ目のデータパスに記述される ts ファイルから構成される CASLII のプログラムである。そのプログラムが二番目の引数のデータパスに書き出す。
4. 与えられた ts ファイルのデータパスがない場合は”File not found”というエラーメッセージを出力して、プログラムを終了させる。
5. 与えられた ts ファイルのデータパスを見つけた場合、
   (a) 構文的にかつ意味的に正しい場合、標準出力に”OK”を出力して、二番目のパスにその CAS ファイルを上書きする。
   (b) 構文的なエラーがある場合、そのエラーは行の番号と共に、”Syntax error : line { LineNumber } ”というエラーメッセージを標準エラーに出力する。
   (c) 意味的なエラーがある場合、そのエラーは行の番号と共に、”Semantic error : line { LineNumber } ”というエラーメッセージを標準エラーに出力する。
   (d) 与えられたファイル内に複数の構文的か意味的なエラーがある場合、その最初のエラーのみは出力される。また、途中でエラーを見つけた場合は二番目の引数の CAS ファイルのデータパスに出力しない。

2　プログラムの方針・設計

本課題のプログラムの設計は全体的に図 1 のようになる。

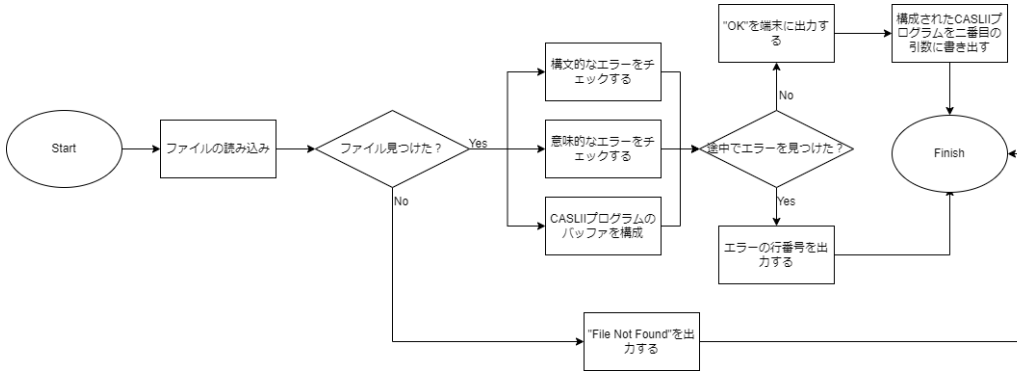

図 1　全体的なプログラムの流れ

2.1　コンパイラの全体フローチャート

本課題のメイン処理となる。コンパイラのアルゴリズムは図 2 のようになる。

2.2　過去の課題プログラムの再利用

コンパイラは課題 3（意味的解析器）の続きとなる。課題 3 の追加点は以下に説明する。

1. 課題 3 までは変数ごとに変数表に追加したが、変数タイプのオブジェクトはその変数の名前、型名しか持たないが、本課題はその変数の番地も格納しなければならないので、変数型オブジェクトは変数の名前、型、最初番地、最後番地、参照状態、代入状態に拡張された。また、配列型か標準型かを区別できるように、タイプ型オブジェクトはそのタイプの名前だけではなく、その変数の最初添え字と最後添え字を格納する。標準型であれば、最初添え字と最後添え字両方は 0 にする。

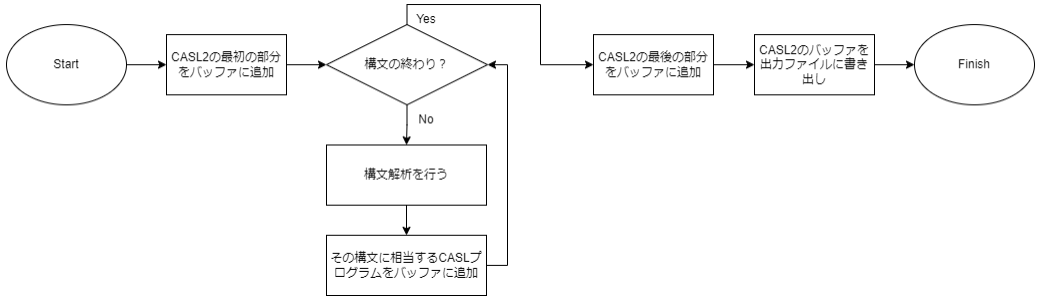

図 2　コンパイラの流れ

2. 課題 3 までは使用可能なスコープのみを格納する形にしたが、変数の番地も考慮しなければならないので、使用可能なスコープの他に、全てのスコープを格納する表も用意した。その表はページ（スコープ）ごとに分割して、現スコープの番号を格納するために、別の変数に格納する。

2.3　レジスタやメモリの利用方法

CASLII のプログラムは各構文に分割して、またレジスタの数が 8 個（使用できるのは 5 個）しか使わず、構文の数より少ないので、各構文に異なるレジスタに割り当てることができない。よって、スタックを使って、変数や定数や関数を呼出したいときには、PUSH 命令を使い、その変数や定数を使いたい時には、POP 命令を使う。また、メモリについては三つの役割に使う。

- サブルーチンのライブラリメモリ領域、事前に定義されたサブルーチンをライブラリとして格納されて、本課題の計算などを楽にするために使われる。サブルーチンのライブラリは 256 語長のバッファが必要になり、また、GR6 は 0、GR7 はそのバッファの先頭番地を指定するように CASL2 のプログラムの最初に初期化しなければならない。そのライブラリは LIBBUF として定義される。よって、毎回 CASLII のコードを生成するために以下のコードは先に書かないといけないことになる。

```
CASL START BEGIN
BEGIN LAD GR6, 0;
      LAD GR7, LIBBUF;
```

- グローバル変数とサブルーチンの引数とローカル変数のメモリ領域。その領域は VAR として定義されて、領域の大きさはプログラム内のグローバル変数とサブルーチンの引数とローカル変数の大きさによる（2.2 章参照）。変数や引数の大きさは以下のように定義できる。

```
var_size = var.endAddress - var.startAddress + 1
```

- 各文字列の格納領域。CASLII は文字列が定数として扱うことができないので、CASL2 のプログラムのメモリ領域に格納しなければいけない。

以上を踏まえて、CASLII のコードの最後のところは以下のコードが必要になる。

```
VAR DS [変数の必要番地];
[文字列]
LIBBUF DS 256;
[ライブラリ]
```

2.4　式の処理方法

課題 3 までは「式」構文以降（単純式、項、因子、変数、純変数、添え字付き変数、添え字）はその構文の型だけを返すことにしたが、本課題ではその構文に相当する CASLII コードも返す必要があるが、。それで、構文のコードの区切り（発展課題）も楽にできるからである。実際には以下のようになる。

- 添字構文、添字構文は式構文と一致するので、式構文が返す型と CASLII コードをそのまま返す。
- 添え字付き変数、変数名と添え字で構成される。CASLII プログラムは以下のようになる。

```
[添え字からの CASLII コード]
POP GR2; 添え字で計算された値を取り出す
ADDA GR2, =[変数名の番地]; VAR からの変数の先頭番地 + 添え字と一致
LD GR1, VAR, GR2; その変数の実番地に格納されたデータを GR1 に格納
PUSH 0, GR1; その値をスタックに PUSH する
```

- 純変数、添え字付き変数と似ているが、添え字の値とプラスする必要がない。つまり、CASLII コードは以下のようになる。

```
LD GR2, =[変数名の番地]; VAR からその変数の番地
LD GR1, VAR, GR2; その変数の実番地に格納されたデータを GR1 に格納
PUSH 0, GR1; その値をスタックに PUSH する
```

- 変数、変数は純変数か添え字付き変数かであるが、その構文に相当する CASLII コードを返す。
- 定数、符号なし整数、文字列、”false”、”true”の選択で、それぞれのコードは以下のようになる。
  - 符号なし整数、

```
PUSH [符号なし整数]; その整数をそのままスタックに PUSH する
```

  - 文字列
    - 文字、

```
LD GR1, =[文字]; その文字を GR1 に格納
PUSH 0, GR1; スタックに文字をプッシュする
```

    - 文字列

```
LD GR1, =[文字列の長さ];
PUSH 0, GR1;
LAD GR2, CHAR[文字列の番号]; その文字列のメモリ領域の番地を GR2 に格納
PUSH 0, GR2;
```

  - ”true”、

```
PUSH =#0000;
```

  - ”false”、

```
PUSH =#FFFF;
```

- 因子構文、変数、定数、(式)、not 因子の選択になり、相当された構文によって CASLII のコードを返す。ただし、not 因子の方は、boolean の値を逆にする必要があるので、以下のコードを追加する。

```
PUSH =#FFFF;
```

項構文、単純式構文と式構文はこれから演算子を持つ場合のみを説明する。演算子がないと、その構文の中にただ一つの構文しか持たなく、そのまま返すだけである。

最初は三つの構文同じく、二つの「子」構文の CASL コードを追加した後、PUSH された値を POP する。

```
[子構文 1 の CASL バッファ]
[子構文 2 の CASL バッファ]
POP GR2;
POP GR1;
```

その後は演算子によって、CASLII のコードを生成する。追加コードは表 1 に示される。

ただし、単純式の方は最初に「-」があるので、演算子を使う前に、最初に以下のコードを追加しなければならない。

表 1　演算子による追加コード

演算子	CASLII コード	演算子	CASLII コード
*	CALL MULT; PUSH 0, GR2;	+	ADDA GR1, GR2; PUSH 0, GR1;
div 又は/	CALL DIV; PUSH 0, GR2;	-	SUBA GR1, GR2; PUSH 0, GR1;
mod	CALL DIV; PUSH 0, GR1;	or	OR GR1, GR2; PUSH 0, GR1;
and	AND GR1, GR2; PUSH 0, GR1;		
=	CPA GR1, GR2; JZE TRUEX; LD GR1, =#FFFF; JUMP BOTHX; TRUEX LD GR1, =#0000 BOTHX PUSH 0, GR1;	<>	CPA GR1, GR2; JNZ TRUEX; LD GR1, =#FFFF; JUMP BOTHX; TRUEX LD GR1, =#0000 BOTHX PUSH 0, GR1;
<	CPA GR1, GR2; JMI TRUEX; LD GR1, =#FFFF; JUMP BOTHX; TRUEX LD GR1, =#0000 BOTHX PUSH 0, GR1;	<=	CPA GR1, GR2; JPL TRUEX; LD GR1, =#0000; JUMP BOTHX; TRUEX LD GR1, =#FFFF BOTHX PUSH 0, GR1;
>	CPA GR1, GR2; JPL TRUEX; LD GR1, =#FFFF; JUMP BOTHX; TRUEX LD GR1, =#0000 BOTHX PUSH 0, GR1;	>=	CPA GR1, GR2; JMI TRUEX; LD GR1, =#0000; JUMP BOTHX; TRUEX LD GR1, =#FFFF BOTHX PUSH 0, GR1;

```
POP GR2; その構文の値を GR2 に格納
LD GR1, =0; GR1 を 0 にする
SUBA GR1, GR2; -GR2 を得たい
PUSH 0, GR1;  -GR2 を PUSH
```

2.5　変数の代入の方法

変数の代入は「代入文」構文に相当する。代入文構文は「左辺」と「右辺」構文から構成される。右辺は式と一致するので 2.4 章と同じである。左辺は「変数」のみで構成されるが、2.4 章に説明された「変数」と異なる。2.4 章の「変数」はその変数の番地に格納される値をスタックに PUSH するが、「左辺」の方が VAR からの番地までだけをスタックに PUSH する。つまり、左辺の「変数」構文が以下のようになる。

```
LD GR2, =[変数名の番地]; VAR からその変数の番地
PUSH 0, GR2; その値をスタックに PUSH する
```

以上を使って、左辺と右辺の CASL バッファを得たら、以下のように、変数代入文構文の CASLBuffer を構成する。

```
[右辺の CASL バッファ]
[左辺の CASL バッファ]
POP GR2;  左辺の値を GR2 に格納
POP GR1;  右辺の値を GR1 に格納
ST GR1, VAR, GR2;
```

2.6　出力の処理

出力の処理は入出力文の writeln の方で処理を行い、式の並びからの値のリストを一個ずつ読み取り、それぞれ、以下のコードを実行する。

```
[式のバッファ]
POP GR2; 式の値を GR2 に POP する
POP GR1: 文字列の場合のみ使う
CALL WRT[値の型]; 型によって、WRTCH、WRTSTR、WRTINT かを呼び出す
```

すべての式を処理した後、最後に、「CALL WRTLN」を追加して、改行をつける。

2.7 分岐の処理方法

2.7.1 if 文

if 文は「式」構文からの値が偽であれば、ELSE の複合文の CASLII コードを実行するが、真であれば、IF 文の複合文を実行する。なので、「式」の値が=#FFFF と比較して、ゼロ（同じ）であれば、ELSE の複合文を、そうでは無ければ、IF の複合文を実行するという風に分岐される。つまり、ELSE 文がなければ、CASL バッファが以下のようになる。

```
[式の CASL バッファ]
    POP GR1;  左辺の値を GR2 に格納
    CPL GR1, =#FFFF;  右辺の値を GR1 に格納
    JZE ELSEX;
[IF の複合文のバッファ]
ELSEX NOP;
```

また、ELSE 文があれば、以下のようになる。

```
[式の CASL バッファ]
    POP GR1;  左辺の値を GR2 に格納
    CPL GR1, =#FFFF;  右辺の値を GR1 に格納
    JZE ELSEX;
[IF の複合文のバッファ]
    JUMP ENDIFX;
ELSEX NOP;
[ELSE の複合文のバッファ]
ENDIFX NOP;
```

2.7.2 while 文

while 文は ELSE 文なしの IF 文と似ているが、複合文を実行した後、その while 文の最初のコードに移る。つまり、以下のようになる。

```
LOOPX   NOP;
[式の CASL バッファ]
        POP GR1;  左辺の値を GR2 に格納
        CPL GR1, =#FFFF;  右辺の値を GR1 に格納
        JZE ENDLPX;
[IF の複合文のバッファ]
        JUMP LOOPX;
ENDLPX NOP;
```

2.8 手続きの呼出し方法

手続き呼び出し文に相当する。手続き呼び出しは大体四つのプロセスに分けられる。

1. 実引数の最初の番地の指定、GR8 から引数の数によって他のレジスタに一番下の実引数を指定させる。

   ```
   LD GR1, GR8; 副プログラムの戻り番地を格納する GR8 を GR1 にコピー
   ADDA GR1, =[引数の数];GR1 を一番下の実引数に指定させる
   ```

2. 実引数を仮引数への割り当て、副プログラムを呼び出すために、副プログラムの実引数の数とその副プログラムの仮引数の数が同じではなければならない。本プログラムでは、副プログラムの仮引数とローカル変数の値がメモリ領域 (VAR) に格納するので、以下のように、スタックフレームにある実引数の値を仮引数の番地へ割り当てられる。

   ```
   LD GR2, 0, GR1; 現実引数の番地を格納する GR1 を GR2 にコピー
   LD GR3, =(仮引数の番地); VAR から仮引数の番地を GR3 に格納
   ST GR2, VAR, GR3; 実引数を仮引数の番地へ割り当て
   SUBA GR1, =1; 実引数のポインターを持つ GR1 を次の実引数に移動
   ```

   以上のコードは副プログラムの仮引数の数で繰り返す。
3. 複合文、メインプロセスの複合文と同じである。
4. GR8 を元のところに置く。GR8 は実引数の数によって、スタックフレームから上がるが、元のところに戻さないと、スタックオーバーとなる可能性がある。以下のコードで GR8 の値を元に戻すことができる。

```
LD GR1, 0, GR8; 副プログラムの戻り番地を GR1 に格納
ADDA GR8, =(実引数の数); GR8 を元のスタック番地に戻す。
ST GR1, 0, GR8; GR1 に格納された副プログラムの戻り番地をまた、GR8 に格納
```

2.9 ラベルの管理方法

「式」や分岐の処理や手続きの呼び出しに使う。それぞれは以下のようにラベルをつける。

- 「式」= TRUEX と BOTHX
- if 文 = ELSEX と ENDIFX
- while 文化 = LOOPX と ENDLPX
- 手続き = PROCX

それぞれ異なるラベルをつけるが、そのラベルの後ろに”X”という文字が入っている。その”X”はカウンターを示して、本課題では 4 つの static カウンターを使って、ラベルを管理する。それぞれは「式」のカウンター、「if 文」のカウンター、「while 文」のカウンター、「手続きのカウンター」である。毎回相当する構文が実行されたら、そのカウンターをインクリメントすることで、if 文を何回も呼び出すと、同じラベルのをつけないように設定する。また、if 文の中で if 文を使うと、最初の if 文と次の if 文のラベルが重複になる可能性があるので、カウンターをインクリメントする前に、ローカル変数に格納して、そのローカル変数をその構文のカウンターとして扱う。

2.10 スコープの管理方法

2.2 章で示した通り、本プログラムでは、使用可能変数表だけではなく、全体的な変数表も用意して、変数の番地を求めるときに、その表を使う。また、現スコープの番号を好きに変更できるように、もう一つのカウンターを用意して、block 構文で毎回手続きを定義する度に、インクリメントするが、block 構文が終わった後、そのカウンターを 0 にすることで、スコープを管理することができる。

2.11 CASLII の最初部分と最後部分

3 実装プログラム

JAVA では、複数の返却値を返すのができないので、最初に、「式」構文以降の返却値を元々その型のみであるが、以下のように変更する。

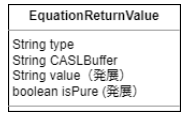

図 3　式の返却オブジェクト

また、関数型オブジェクトや変数型オブジェクトは番地も考慮しなければいけないので以下のように変更する。

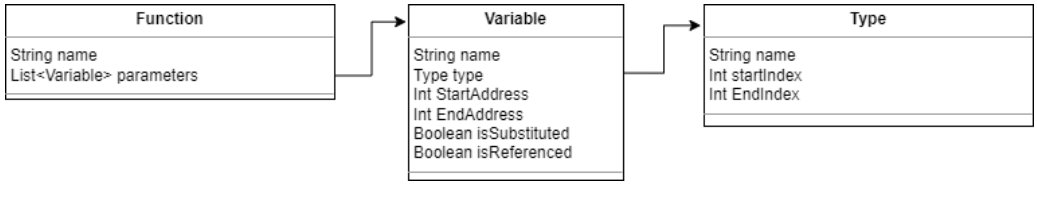

図 4　変更されたオブジェクト

各構文に相当する関数は「式」構文以降が EquationReturnValue 型の値を返却するが、その残りの関数は String 型の値（CASL バッファ）を返す。

4 「発展」最適化

最適化は「式」構文以降の構文だけに適応して、図 3 の value と isPure を使う。

- value, value はその構文の値となる。例えば、i+2 という式があるとしたら、以下のようになる。
  - 式の value は”i+2”
  - 一つ目の単純式の value は”i”、二つ目の単純式の value は”2”となる。
  - 単純式 1 の項の value は”i”、単純式 2 の項の value は”2”となる。
  - ...

  つまり、以下の方法で式の値が変更できる場合、その value を変更するが、そうではない場合は、そのままその構文の value に追加していく。
- isPure、isPure は因子の「(式)」に使い、もし、演算子がない場合は isPure を True にして、そうでなければ False にする。そうすると「(式)」を判断するときに、isPure が true であれば、カッコを追加しなくてもよい。それで value の一致判定をもっと簡単に判断できて、もっと効果的に最適化できる。

4.1 定数計算

定数計算は両方の演算数が定数のときにのみ、演算子によって実施する。定数かどうかは JAVA の Integer の ParseInt メソッドを使う。ただし、value”i+1”などの int に変更できない値がある場合は、NumberExceptionError という例外が発生されるので、Try-Catch 文を使って、boolean のローカル変数の値を変える。その value が定数に変えられるとしたら、その boolean 変数を True にして、その値を他の変数に格納する。そうでなければ、その boolean 変数を false にする。その二つの変数用の boolean 変数が true だとしたら（つまり、両方の演算数が定数に変えられるとしたら)、その演算子によってコンパイラ内で定数の演算を行う。最後に、その構文の value は今まで登録された値を消して、その演算結果に変える。

4.2 代入的簡約化

簡約可能な演算命令は表 2 に示される。

表 2　簡約可能演算命令表

命令	結果	命令	結果	命令	結果	命令	結果
-0	0	x%x	0	0/x	0	x/1	x
0*x	0	x*0	0	x*1	x	1*x	x
0%x	0	x%1	0	x+0	0	0+x	x
x-0	x	x-x	0	!F	T	!T	F
x and F	F	F and x	F	x and T	x	T and x	x
X or F	x	F or x	x	x or T	T	T or x	T
x==T	x	T==x	x	x!=F	x	F!=x	x
x and x	x	x or x	x	x==x	T	x!=x	F

以上の代入的簡約化の設定方法は構文の value の判定からである。また、value で二つの値が判定できるので、「式」以降の構文だけではなく、代入文にも使われる。例えば、以下のテストケースである。

```
program testBasic;
var i: integer;

begin
    i := 5;
    i := i;
    writeln(i);
end.
```

”i := i”という文はそもそも意味がない（自己代入）ので、その文を CASL2 プログラムに書かなくてもよい。

4.3 最適化の評価

以下のテストファイルで最適化のプログラムを判断する。

```
program testBasic;
var i: integer;
    x : boolean;

begin
    i := 2*5;
    i := (i/1*1*1+0+0+i mod i + (i - i))+0*i+0 mod i+i*0+i mod 1+(2+6-4-4); (* i := i と一致する*)
    x := (x=x) and (x or true) and (true or x) and (x or x or x or true); (* x := true と一致する*)
    if x = true then
    begin
        writeln(i);
    end;
end.
```

以上のテストケースを使って、最適化機能付きコンパイラが出力する CASL ファイルの行数が **32** 行が最適化機能なしのコンパイラが出力ファイルの行数が **232** 行になる。これはもちろん大幅な最適化になるが、以上のケースが最悪の場合だけである。例えば、以下のテストケースの場合、

```
program testBasic;
var i: integer;

begin
    i := 5;
    writeln(i);
end.
```

両方のコンパイラが **17** 行の CASLII コードを出力して、同じ行数になる。よって、以上のケースを追加することで、ある程度（場合によるが)、最適化できるという結論が得られた。

5 「発展」テストの改善

- 再帰関数、今までのテストケースは再帰関数を使っても、その関数を呼び出した後のローカル変数のアクセスはないので、6.2 章のバグがカバーしていない。なので 6.2 章のテストケースも必要ではないかと思う。
- 副プログラムの仮引数の数と実引数の数の判定、副プログラムを確認するときに、引数の数を確認するためのテストケースも必要だと思う。例えば、proc0 は 4 つの引数があるが、副プログラム呼出時に、3 つや 5 つの場合も必要だと思う。
- 配列型変数の添え字問題、添え字についての意味的なエラーがないので、それについてのテストケースも必要だと思う。例えば、配列型変数は a[11.....0] のような降順の配列変数宣言も必要だと思う。また、定義された配列の添え字範囲外のテストケースも必要だと思う。例えば、a[0...10] と定義したが、a[11] をアクセスするとかのテストケースである。
- データの読み込みテストケース、つまり、”readln”を使い、ユーザーからの入力を求める。その値をちゃんと読み込みかどうかを確認するために、writeln を使い、その読み込んだ値を出力するというテストケースがあれば、readln の方も確認できる。

6 考察

6.1 最適化の限界

4 章で示された最適化はある程度実行できるが、限界がある。例えば、以下のようなテストケースである。

```
program testBasic;
var i: integer;

begin
    i := 5;
    i :=  (i*i+2)/((i*i)+2);
    writeln(i);
end.
```

以上のテストケースが人間の目であれば、簡単に i が 1 という結果が得られるが、4 章まで作られたコードが手動的に実行

されてしまう。なぜかというと、”i*i”と”(i*i)”は別のものとして見なすからである。()”を持つ二つの演算数の一致を判定するには 4 章のコードで難しい。i*i+2 を計算するときに全ての演算数が項の構文に混ぜてしまうので、その構文を構成する構文の独立性が失うことが分かった。

6.2 手続きの問題

2.8 章で示された手続きの呼び出し処理は実はバグが入ってしまう。例えば、以下のようなテストケースである。

```
program countToN;

procedure countton(n: integer);
var resultTMP: integer;
begin
    if n<>0 then
    begin
       resultTMP := n;
       countton(n-1);
       writeln(resultTMP);
    end;
end;

begin
    countton(5);
end.
```

直感的に見れば、答えとしては”1,2,3,4,5”という順番に出力するはずであるが、結果としては”1,1,1,1,1”、つまり、全て 1 になってしまう。これはなぜかというと、手続き内のローカル変数が動的ではなく、静的に格納するからである。変数 resultTMP はその変数を呼び出す度に、異なる値を格納するが、同じ所に格納するので、全ての writeln のところは最後に、呼び出す値 (1) になってしまうことが分かった。

以上の問題を解決するためにスタックフレームを使って、図 5 のように設計する。

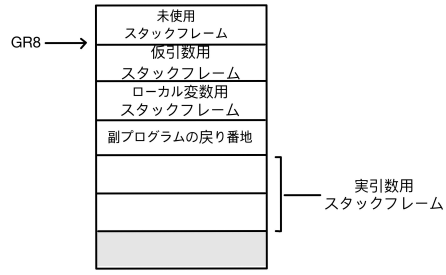

図 5　スタックフレーム

スタックフレームを使って、メモリ領域ではなく、副プログラムの仮引数とローカル変数はスタック上に格納する。また、副プログラム内の複合分は PUSH と POP も使うので、GR8 の位置は仮引数用スタックフレームの上に置いておいたら、バグが発生しない。また、2.8 章で示したように、先に実引数を仮引数用の領域に格納するが、メモリ領域ではなく、その仮引数用のスタックフレームに格納すればよい。また、副プログラム内で仮引数やローカル変数をアクセスしたいときにも、メモリ領域ではなく、スタックフレームからそのデータを受け取る。つまり、「変数」構文で副プログラムかメインプログラムかを区別する必要がある。最後に、スタックオーバーしないようにちゃんと GR8 を元のところにしなければならない。

7 感想

本課題は先生がもうすでに注意されたが、一番難しいというよりめんどくさい課題になると思う。どの CASLII コードがどの構文に相当するかが最初にわからないので、たくさんのバグが入ってしまった。そこで、各々の構文の役割をちゃんと考えてみて、どのように CASL コードに影響を及ぼすかを理解してから、CASlII コードの生成が楽になった。