Compiler

学生番号 09B20083 吉崎　響

2023 年 1 月 27 日

1　外部仕様

Complier は与えられた Pascal 風言語の ts ファイルを CASL II に翻訳するプログラムである。ts ファイルの内容は課題 1 に準ずるものであり、作成する CASL II の仕様は課題の仕様に従っている。メソッドは Compiler.run(String, String) である。第一引数には翻訳したい ts ファイルのパスを第二引数には翻訳した CASL II の内容を書き出したいファイルのパスを入力する。Complier の仕様は以下である。
・入力ファイルが見つからない場合は標準エラーに”File not found”と出力して終了するこの際 cas ファイルは作成されない
・ts ファイルに構文的もしくは意味的な誤りを発見した場合は標準エラーにエラーメッセージを出力する. エラーメッセージは課題 2、課題 3 と同様である。
・第一引数で指定された ts ファイルを読み込み，CASL II プログラムにコンパイルを行い第二引数で指定されたファイルに書き出す

2　設計方針

2.1　全体

課題 4 でも課題 3 同様に課題 2 で作成した木を再利用している。木の構造については指導書の EBNF に沿って作成している。詳しくは Checker レポーと Paster レポートに記載している。また課題 4 では課題 3 で作成した記号表を再利用している。課題 4 では追加の情報が必要になるため、記号表に対して情報の追加を行っている。今回も課題 3 同様の方法で Visitor モデルを利用して実装した。課題 4 のために新たに利用した Visitor モデルは 3 つである。
1. 記号表に追加の情報を入れ、文字列をラベル付けする Visitor (Visitor A　とする)
2. メインの複合文に関して CASL II を記述する Visitor　　(Visitor B　とする)
3. 副プログラムに関して CASL II を記述する VIsitor (Visitor C　とする)
VisitorA =make_chart_visit(),VisitorB=write_code_visit(),Visitor C=subpro_write_visit()
この 3 つに分けた理由としては、1 の機能は、2.3 が成立する上で必要条件であるため、独立させることで 1 の機能が正常に動作することを保証できるからである。また 2.3 を分けた理由としては 2.3 では主に変数に対して処理後大幅に異なるからである。

2.2　記号表追加と文字列のラベル

2.2.1　CASL II の変数と文字列の扱い方

CASL II では変数はメインプログラムでは VAR 領域、副プログラムではスタックで管理する方法をとった。メインプログラムの場合、メインプログラムの変数の数だけ VAR ラベルで DS で領域を確保する。VAR 領域にどの順番で値を保存するかは任意に決めることができる。そこで変数が領域のどこに保存されてい

るかに関する項目を記号表に追加することで参照の際に VAR から何番目の領域にどの変数が格納されているかを確認して呼び出だせるようにした。表については後述の記号表追加で述べる
副プログラムの場合はスタックでローカル変数を管理する。これは副プログラムは再帰的に呼び出される可能性があり、その場合、VAR のような固定の領域で管理するよりもスタックで管理するほうが楽だからである。副プログラムの場合は、**副プログラム開始時のスタックの先頭から何番目に変数が格納されているか**を記号表に追加することで参照を可能にした。記号表の番号はプログラムごとに独立である。(すべてのプログラムの記号表は 0 から始まる)

2.2.2 記号表追加

課題 4 で必要となる記号表の情報は以下のものである。1. どのプログラムまたは副プログラムの変数であるか、2. 変数の名前、3. 配列の最小の添え字 (デフォルトは 0)、4. 配列のサイズ (デフォルトは 1)、5. 仮パラメーターであるかどうか、6.VAR またはスタックでの位置番号。このうち 1.2.5 は課題 3 の表にすでに含まれるため、足りない情報を記号表に追加する形で実装した。
6 の情報について詳しく説明する。CASL II では　メインプログラムの場合
VAR DS 3 のような場合
番地 VAR[変数 1] VAR+1[変数 2] VAR+2[変数 3]
のように VAR を基準としていくつ加算するかで管理されている。
副プログラムの場合
ローカル変数 3 つの場合スタックは以下のようになる
==========
(変数 1)　←副プログラム開始時のスタックの先頭
——-
(変数 2)
——-
(変数 3)
==========
番号の振り方は以下のように振る。
メインプログラムの場合、番号の順番は変数名でアルファベット順に振った。
配列の場合はその配列のサイズ分の領域を確保するが、記録しているのは先頭の配列の番号だけである。
副プログラムの場合
仮パラメーターは定義されている順番に 0 から数を振っていき、残りのローカル変数をアルファベット順に振った。(理由は副プログラムの説明で触れる)

また以下の表も同時に作成した。副プログラムの名前と副プログラムに振られた番号の表 (副プログラム呼び出し用ラベルのため)
副プログラムでは仮パラメーターの数も必要となるので仮パラメーター表を作成した。
仮パラメーター表にはプログラムの名前と仮パラメーターの数が格納されている。

例
pro(X,Y)
gar(X Y Z)
c interger a[1..10]
変数表の番号
pro X-0 ,Y-1 gar X-0 ,Y-1,Z-2,a-3,c-13
仮パラメーター表
pro -2 gar -3

2.2.3 文字列定数のラベル表

文字列は、DC 領域で確保するため、ラベルを設定する必要がある。文字列の管理は文字列表で行う。文字列表は実際の文字列と対応するラベル名の 2 つが入った表である。ラベル名は CHART+順番の番号とする。つまり　1 番最初に出てきた文字列には CHART0 であり、5 番目であれば CHART4 というラベル名になる。ただし同じ文字列が来た場合は表には登録しない。同じ文字列ならば同じ参照ラベルでも問題ないからである。また文字数が 1 文字の場合登録しない。(文字定数として扱うため)

2.3 メインプログラム CASLL II 翻訳

メインプログラムの始まりでは初期処理を行っている (START など)

2.3.1 式の処理方法

式の評価はノードの末端から始まるため、因子→項→式の順で説明する。因子ノードより下の定数、変数は内容をスタックに PUSH する。ただし、定数が 2 文字以上の文字列定数の場合は PUSH しない。2 文字以上の文字列は演算の対象にならないからである. 因子は下のノードから戻ってきた際に、下のノードが not　因子であればスタックから値を POP して not 処理を行って結果を PUSH それ以外の場合はすでに結果が PUSH されているので何も行わない。 項のノードでは下のノードを左から訪問するのではなく右から訪問する。そうすることで、スタックに入る因子の結果が右のものほど奥になるため、POP すると並び順で左から順に取り出せるからである。乗算は左から順に計算を行わないと結果が異なる場合があるためこのようのした。例 因子 A*因子 B*因子 C div 因子 D
項ノード帰還時のスタック
======
因子 D
因子 C
因子 B
因子 A
======
項の処理は、項の下のノードである因子ですでに結果が入っているため乗法演算子に合わせてスタックから因子を 2 つずつ POP して、計算し結果をスタックに PUSH すればよい。乗法演算子の数だけ処理を繰り返した後そのまま上のノードへ戻る。単純式も項と同様の理由で右のノードから訪問する。単純式では符号が存在する可能性がある。負の符号が存在する場合は、スタックから POP して正負を反転させてスタックに PUSH する。それ以外の処理は項と同様である。ただし演算の種類は変わる。最後に式の処理では、関係演算子がある場合は、因子を 2 つ POP して計算して結果を PUSH する。**式の結果は PUSH するような仕様になっている。**また式の結果がどの型 (String,char,int) であるかは出力文の際に必要になる。そこで式の型を保存するためのスタックを用意した。式の結果を PUSH する際に式の型を PUSH するようにした。ただし boolean は検出できない。boolean は出力文の対象には選ばれないためである。式の型を判断する具体的な方法としては、式の最後に出てくる変数または定数の型を式の型としている。常に変数または定数の型をノードを訪れるたびに更新し、最終的に式が定まった時に型を POP する。(関係演算子で boolean に変わった場合でも出力には関係ないので影響が出ない) また式が利用された後には式の型のスタックを利用された式の数だけ POP するようにしている。具体的には if.while 文の条件、write の呼び出し、代入文、添え字の後である。

2.3.2 変数の代入の方法

変数の代入では 左辺 ":=" 式. である。そこで式ノードから先の訪問しすることで結果をスタックに保存させる。そして、左辺ノードでは変数の VAR における位置を調べ、その番地に値を保存する。

2.3.3 write の処理

write では、出力するものが文字であるか、数字であるか、文字列であるかがわからないと出力できない。また、出力の順番は、式の順である。そこで write ノードの下の式の並びのノードを右から訪問することでスタックの順序を後に出力するものほど奥にした。また式の型をスタックから POP してその型に応じて処理を進めた。

2.3.4 分岐の処理方法

分岐が登場するのは、関係演算子の計算、if 文、while 文である。いずれの場合もラベルの管理が必要である。関係演算子の場合は、正誤判定が登場した回数を記録することで、1 回目の正誤判定であれば TURE0,2 回目であれば TRUE1 のように回数で区別してラベル付けするだけでよい。これは正誤判定中に別の正誤判定が登場することがないからである。一方で while 文と if 文では、if 文中に別の if 文が登場することがある。そのような場合では、戻るべきラベルの番号がわからなくなってしまう。そこでスタックを利用した。スタックを利用すれば、取り出すべきラベルがわかる。if 文の始まりで PUSH し、終わりで POP すれば、必ず、if 文終了時には自身のラベルが POP できるからである。

2.3.5 レジスタ利用方法

演算で使用するのは GR1 と GR2 のみである。GR1 と GR2 レジスターはすべての処理において独立に機能するので GR1 と GR2 の値は必要であれば初期化するようになっている

2.3.6 副プログラム呼び出し

副プログラムの呼び出しを行う前に引数をスタックに積む必要がある CASL II の変数と文字の扱いで述べたように、スタックには定義された順と逆に積む。またスタックポインタは呼び出しが終了すると同時に修正する必要がある。これは呼び出し終了時にはスタックには代入した引数が詰まれいているままなのでスタックポインタを減じることでスタックを整理する必要があるからである。呼び出しの際は作成した副プログラムラベル表から呼び出す副プログラムの名前と一致するラベルを呼び出せばよい。

2.4 副プログラム CASLL II 翻訳

メインプログラムと異なる点は、変数の扱いのみである。また副プログラムの先頭ラベルは副プログラムラベル表から参照して貼る。

2.4.1 ローカル変数用の領域をスタックに用意

スタックを用意するために、スタックポインタをローカル変数と仮パラメーターの数だけ減じておく。そうすることでローカル変数を格納する領域を確保する。またこの領域は副プログラム終了時にスタックポインタを加算して消さなければならない。そうでないと戻り番地を参照できなくなる。

2.4.2 仮引数に実数代入

副プログラム開始時に仮引数に引数をコピーする必要がある。副プログラム開始時のノードは副プログラム宣言なので副プログラム宣言で処理を行う。スタックには引数と戻り番地が積まれており、また変数表でローカル変数のスタックポインタ先頭からの位置が決まっている状態である。例
pro(X,Y,Z)

c :char d :integer
の場合に pro(99,100,101) で呼び出された場合
==========
X　(1 番目に定義された仮パラメータ) ← (スタックの先頭)
Y (2 番目に定義された仮パラメータ)
Z (3 番目に定義された仮パラメータ)
c (ローカル変数)
d　(ローカル変数)
戻り番地
99　(1 番目の引数)
100 (2 番目の引数)
101 (3 番目の引数)
==========
そこで、引数を対応する仮パラメータの領域に代入する必要がある。仮パラメータと対応する引数をスタックに積む順は同じにしているため、仮パラメータと引数の間には副プログラムの仮パラメータとローカル変数の数の合計になっている。その性質を利用して、仮パラメータの位置からその数だけ離れたスタックの内容を代入すればよい

2.4.3　変数の呼び出し方

副プログラムの場合、変数の呼び出しにスコープ管理が必要になってくる。方法としては自身の副プログラム変数表に該当する変数があれば、その番号を利用して、スタックから値を取り出す。一方で存在しなければ、メインプログラムの変数表からスタック番号を取り出し、VAR 領域から取り出す。代入の際も同様である。

2.4.4　副プログラム呼び出し

メインプログラムと同様に対応する引数をスタックに格納すればよい

3　実装

3.1　全体

作成した表一覧
変数表 variable_list
副プログラムのラベルと仮パラメータの数の表
paranum_list
while と if のラベルスタック while_log if_log
その他 true などのラベルはクラスで管理

3.2　記号表追加と文字列のラベル Visitor A

3.2.1　記号表追加

配列の先頭の添え字と配列のサイズを表に追加する方法は以下のものである。すべての変数の定義は変数宣言の並びノード下で行われる。そのため、このノードに帰ってくるまでに処理を終わらせる。変数宣言

の並びでは　(変数名の並び　型) の順番で繰り返しノードを訪れる。そこで同じ (変数の並び) ノードに含まれている変数はすべて同じ型の変数である。変数名の並びに含まれている変数の名前をすべて獲得する。次に型ノードでは下のノードが標準の場合はデフォルトが適切な値となっているため処理は必要ない。配列型ノードである場合、配列型の下のノードである添字の最小値と添字の最大値の値を取得するし、そして前の (変数名の並び) ノード獲得した変数と一致する記号表に対して 3 と 4 の情報を追加した。

また記号表に番号を振る方法はメインプログラムでは記号表を取り出し加算する。記号表はハッシュマップであるため、アルファベット順に番号が振られる。

副プログラムの場合は、ローカル変数に番号を振る前に、仮パラメーターに対して定義されている順に番号を振る。仮パラメータ名の並びでは前から順に子供のノードを訪れるために、(仮パラメーター名) ノードを定義された順に訪れる。そこで仮パラメーターが追加されるたびに番号を振ることで定義順に番号を振った。また仮パラメーターの定義がすべて終わると仮パラメータ ノードに帰るためそのタイミングで仮パラメーターが終わったことを記録する。この時の数を記録することで副プログラムの仮パラメータの数を表に追加した。その他ローカル変数ではメインプログラム同様に記号表を取り出し加算するこの時初期値を、仮パラメータの数とすることで重複が生じないようにした。

3.2.2　文字ラベル

文字列は必ず、文字列ノードを通る。そこで文字列ノードで文字列の内容を獲得する。獲得した文字列に対してラベル名を CHART+数の形式でつける。数は文字列を獲得した順に 0 から降っていく。同じ内容の文字列が来た場合には登録は行わない仕様になっている。

3.3　メインプログラム CASLL II 翻訳　　Visitor B

3.3.1　初期 終了処理

メインプログラムのでは、副プログラムノードの複合文などの翻訳を行わないように、副プログラム宣言群を通らないようにした。またプログラムの最初の処理である START の呼び出しや、GR7 に LIBBUF の番地を格納するなどの初期処理は最初に訪れるノードであるプログラム名に記述した。また RET によるメイン文の終了の処理はすべてが終わった後に訪れる (プログラム) ノードで記述した。

3.3.2　レジスタ管理

GR8 はスタックポインタ　GR7 は LIBBF GR1,2 は演算に使用する

3.3.3　式の処理方法

変数ノードを POP する方法は、標準型の場合
1. 変数の名前を利用して、変数表からスタック番号を得る
2.GR2 にスタック番号を LD する
3.VAR から GR2 番目の変数を GR1 に LD する
4.GR1 の内容を PUSH する
配列型の場合は　変数名　添え字　の順で　ノードが並んでいるので添え字のノードから先に訪問する。すると添え字は式ノードで計算されるため、添え字の値がスタックに PUSH されている状態になる。1.GR1 にスタックから添え字の値を POP する
2. 変数表から配列の最小の添え字を獲得し、GR2 に LD する
3.GR1 の内容から GR2 の内容を引く (これで配列において先頭から何番目であるかが GR1 に入る)
4. 変数の名前を利用して、変数表から配列の先頭のスタック番号を得る

5.4 の内容を GR2 に LD する
6.GR2 に GR1 を加算する (スタックにおける要素の位置がわかる)
7.VAR から GR2 番目の変数を GR1 に LD する
8.GR1 の内容を PUSH する
定数の場合は、文字列の長さが 1 文字の場合は、文字列を PUSH、符号なし整数の場合はそのまま PUSH true の場合は#0000 を PUSH、false の場合は#FFFF を PUSH する。次に因子ノードでは、not の場合は true か false がスタックに PUSH されている状態である。そこで内容を POP して、#FFFF と排他的論理和を取ることで not を求め結果を PUSH した。項ノードでは GR1、GR2 の順で POP する (GR1 には左 GR2 には右の非演算子が入る) そして乗法演算子に対応した計算を GR1 と GR2 に対して実行し、結果をスタックに POP する。例 5+6
POP GR1 (5 が入る)
POP GR2 (6 が入る)
ADDA GR1 GR2(GR1 を必ず被乗数にする)
PUSH 0,GR1
単純式ノードでは基本的に項と同様であるが、負符号がある場合のみ、スタックから GR2 へ POP し、GR1 に 0 を LD し、SUBA GR1,GR2 を実行し、GR1 を POP することで負の数へ変換しスタックへ保存した。式ノードでは関係演算子がない場合は何も行わない。関係演算子がある場合は以下のように処理した。(式は逆からの訪問を行っていないためスタックでは奥に左の値が来る) 1.POP GR1
2.POP GR2
3.CPA GR1,GR2
この結果に応じて判定を行った。判定結果によってラベル TRUE に移動するか、移動せず FALSE の処理を行うかに分かれる。
分岐条件 TRUE
false の処理
JUMP END
TRUE true の処理
END
ここで関係演算子の種類に応じて分岐条件が異なる。'=' は JZE '¡¿' は JNZ '¡' は JMI '¡=' は JMI と JZE の両方で TRUE へ分岐する。'¿' は JPL '¿=' は JPL と JZE の両方で TRUE へ分岐する
TRUE の場合は GR1 に#0000 が、FALSE の場合は GR1 に#FFFF が入る。そして END で GR1 の内容をスタックに PUSH する。

3.3.4 変数の代入の方法

代入文は代入する値である式の処理から行うため、すでに答えがスタックに PUSH されている。標準型の場合
1. 変数の名前を利用して、変数表からスタック番号を得る
2.GR2 にスタック番号を LD する
3. 式の結果を GR1 に POP する
3.VAR から GR2 番目の変数に GR1 を ST する
配列型の場合は式の結果、添え字の結果の順で PUSH されている
1.GR1 にスタックから添え字の値を POP する
2. 変数表から配列の最小の添え字を獲得し、GR2 に LD する
3.GR1 の内容から GR2 の内容を引く (これで配列において先頭から何番目であるかが GR1 に入る)
4. 変数の名前を利用して、変数表から配列の先頭のスタック番号を得る
5.4 の内容を GR2 に LD する
6.GR2 に GR1 を加算する (スタックにおける要素の位置がわかる)

7.GR1 に式の答えをスタックから POP
8.VAR から GR2 番目の領域に GR1 を ST する

3.3.5 write の処理

write では出力する内容によって呼び出すサブルーチンが異なる。よって write では出力する式が String か char か interger かを判断しなければ出力できない write ノードでは出力すべき内容がスタックに順に保存されている (1 番手前に最初に出力するものがある) 方針でも述べたようにそれに対応する式の型をスタック保存している。よって write では以下を式の個数だけ繰り返す。1. 式の型のスタックを 1 つ POP する 2. 型にあった方法で出力行うそしてすべての出力が終わったのちに改行を出力する。

3.3.6 if と while 分岐の処理方法

if,while 文では式ノードの処理を行った後、スタックに入っているのが true か false で処理を行う。またラベルは if、while のラベルスタックで管理する以下のようになる
POP GR1
CPL GR1 , =#FFFF”
JZE ELSE (if 文のラベルをスタックに POP)
if 文が true であった時の処理
JUMP END
ELSE NOP (if 文のラベルをスタックの値を獲得 POP ではない)
if 文が false であった時の処理
END (if 文のラベルを POP)
while の場合は
LOOP NOP (ラベルをスタックに POP)
条件分岐 END
while 文の処理
JUMP LOOP
END

3.3.7 副プログラム呼び出し

副プログラムを呼び出す前に引数を POP する。実際は式の処理を行うと PUSH されるので意識しなくとも実行できる。ただし、PUSH の順は定義されている順と逆の順で PUSH するため、式の並びでは右の式ノードから訪問する。CALL の際に呼び出すラベルは副プログラムのラベル表を参照する。また CALL 終了時には仮パラメータの数つまり POP した引数ぶん GR8 を加算する。(スタック整理)

3.4 副プログラム CASLL II 翻訳 Visitor C

3.4.1 レジスタ管理

GR8 はスタックポインタ、GR5 は変数領域の先頭スタックポインタ、GR1,GR2,GR3,GR4 は演算用とする。(副プログラム特有の動作には GR3,GR4 を利用している)

3.4.2　ローカル変数用の領域をスタックに用意

ローカル変数と仮パラメーターの数を変数表の変数の数をカウントすることで獲得する。そしてその数だけスタックポインタを減じる (SUBL GR8 , ローカル変数と仮パラメーターの数) また終了時にはの数だけスタックポインタを加算する (ADDLGR8 , ローカル変数と仮パラメーターの数) この時 GR8 の値を GR5 にコピーしておくことで GR5 がローカル変数領域の先頭の番地となる。

3.4.3　仮引数に実数代入

以下のようなコードで実装した
for(i=0;i¡仮パラメータの数;i++){
引数を取り出す
LD GR3,=(ローカル変数と仮パラメーターの数の合計+1+i)　1 を足したのはスタックに戻り番地があるため、
ADDL GR3,GR5 (対応する引数が入ったスタックの位置を GR3 へ)
LD GR3, 0, GR3 (GR3 にスタックポインタ内の引数をコピー)
仮パラメータのスタックに代入
LD GR4, =i
ADDL GR4, GR5 (i 番目の仮パラメータの番地を獲得)
ST GR3, 0, GR4(仮パラメータに引数を代入)
}

3.4.4　代入方法

変数の呼び出しの際は変数表から副プログラムの変数であるかメインプログラムの変数であるか判断する。
メインプログラムの変数の場合はメインプログラムと同様に処理する。副プログラムの変数である場合 スタックに式の値が保存されているので
1.GR1 にスタックから式の値を POP する
2. スタックにおける位置を獲得し GR4 に保存する
3.GR4 に ADDL により GR5 の値を加算
4.GR4 番地に GR1 の値を ST する

4　考察工夫

再帰に対してスタックが非常に有用であることがわかった。再帰関数における呼び出しがスタックの PUSH に再帰関数の終了がスタックの POP と性質が一致しているからである。今回の場合であれば、if,while のラベル管理にはスタックが最適であった。
では副プログラムのローカル変数をスタックではなく VAR 領域で管理するにはどうするか。再帰に対応するのであれば、VAR 変数の内容を退避する必要がある。というのは再帰によってローカル変数が書き換えられてしまう恐れがあるためである。これを解決するために変数をスタックに退避する必要がある。この手法は、仮パラメータへの値の代入が容易になる一方ですべての変数をスタックに退避した上で終了時にはスタックから取り出し VAR に代入しなければならないため CASL II としては動作が重くなってしまう。よって最適化の観点からも副プログラムにはスタックを使うことが望ましい。

5　感想

pascal は非常に翻訳しやすい言語に感じた。pascal は EBNF からわかる通り、式の評価さえ行えば、残りの機能は基本的に独立している。独立性が強いおかげで、スタックの管理が簡潔になったため、CASL II に翻訳しやすかった。また、CASL II 翻訳を通して、どのような行為がどうしてプログラムを助長にするのかという理屈がつかめた気がした。CASL II を翻訳していると、明らかな式に対して、if 文を使用したことで、必要ない else 文まで記述する必要があるなどである。

6　発展課題

6.1　デザインパターンの導入

課題 3,4 を通して Visitor パターンを実装した。課題 4 では Visitor パターンを利用することで以下の機能を独立させることができた。1. 記号表の追加と文字ラベル表の作成
2. メインプログラムの CASL II 翻訳
3. 副プログラムの CASL II 翻訳
この 3 つを独立させることで、処理別にプログラムが分けられるため、可読性が上がる。またエラーの原因の特定が容易になる。また記号表に追加する情報を変更したい時、副プログラムの機能の解釈を間違え変更する際でも独立させることで変更していない他の機能は正常に動くことが担保できる。また課題 3 では記号表作成のみの機能である Visitor が存在したため、その Visitor を再利用するだけで記号表の作成ができた。このように異なる課題に一部の機能だけを再利用することも可能になる。

6.2　最適化

メインプログラムに対して最適化を行った。メインプログラム記述の visitor を write_code_visit() → Opt_main() に変更
定数を直接評価することで、1+3 や 2*5 などの計算を省略した。定数による*2*4,/2,/4 を算術シフトに変更し重みを減らした。方法としては、casl II のスタックを想定したスタック (ただし　変数は値ではなく変数であることのみ保存する　) と casl II のスタックを想定した型 (定数なら int や boolean、変数は一律に VAR) のスタックを用意した。このスタックから型を取り出すことで定数かどうかがわかる。すべての加法演算と乗法演算において定数と定数の計算であった場合に、CASL II ではなく java 上で計算を行い、直接結果を PUSH する仕様に変更した。関係演算子も同様に java 上で計算し、定数同士の演算の場合は、条件分岐を利用せず、直接 true,false をスタックに代入した。また if,while 文でも式が定数のみで構成されている場合　例 if true then
if,else は用いず、true の場合は、if の複合文を、false の場合は else の複合文のみを翻訳するように変更した。while も同様に while が false の場合は、複合文を翻訳しないようにした。これは実行ステップの上では変化がないが、コードが短くなるという利点がある。
結果
normal03 に対して最適化前 Step count: 141
normal03 に対して最適化後 Step count: 140 normal12 に対して最適化前 Step count: 4741
normal12 に対して最適化後 Step count: 4735
コードに定数同士の計算があまり含まれていないため大きな変化はなかった。
*2 と*4 そして/2 と/4 を算術シフトに変更
normal20 に対して最適化前 Step count: 30330 normal20 に対して最適化後 Step count: 30294 2 倍などは for 文中などで比較的よくでる計算であるため先ほどの最適化より変化が大きい