情報科学演習Ｄ　課題4レポート

氏名　山久保孝亮
所属　大阪大学基礎工学部情報科学科ソフトウェア科学コース
メールアドレス　u327468b@ecs.osaka-u.ac.jp
学籍番号　09B22084

提出日　2025 年 1 月 23 日
担当教員　桝井晃基　松本真佑

1　システムの仕様

課題 4 の外部仕様は以下のようになる．

- 第一引数で指定された ts ファイルを読み込んで構文解析，意味解析，CASL のコード生成の順に処理を行う．コンパイル結果は第二引数で指定された cas ファイルに書き込む．
- コンパイルが成功した場合は文字列”OK”を返し，構文的，意味的誤りを検出した場合は各誤りの内容を返す．課題 3 のように複数の誤りが pas プログラムに含まれる場合は，一番最初に検出した誤りの内容を返す．また，誤りを最初に発見すると cas ファイルを生成せず，入力ファイルが見つからない場合は文字列”File not found”を返す．

2　課題達成の方針と設計

今回の課題では，構文解析と意味解析を終了した際にグローバル変数として定義したリスト resultData に CASL のコードを add() によって追加していく．この resultData は”String 型のリスト”のリストであり，例えば”CALL PROC0”という CASL コードの場合は [”CALL”,”PROC0”] という二つの文字列の要素を持つリストを追加していた．つまり，resultData の各要素は CASL コード一行分を表す．
以下では，各処理における CASL コードの流れを記述する．

2.1　変数の代入方法

変数の代入は，純変数への代入と添字付き変数への代入に分類できる．

- 純変数への代入の場合，

  1. 指定する変数のアドレスと VAR のアドレスの差を計算し GR2 に格納する．
  2. スタックから右辺の値を POP し，GR1 に格納する．
  3. VAR と GR2 から実行アドレスを計算し，GR1 に格納している値をストアする．

  という流れで CASL コードを追加する．1 の差の計算方法は 3.1 の実装プログラムで記述する．また，VAR は 2.4 で記述するようにメモリの先頭の番地である．

- 添字付き変数への代入の場合，

  1. 右辺の構文定義”式”内で処理した CASL コードを追加する．
  2. 添え字を表す構文定義”式”内で処理した CASL コードを追加する．
  3. スタックから添え字の値を POP し，GR2 に格納する．
  4. 指定する配列のアドレスと VAR のアドレスの差を計算し，GR2 の値と加算し，GR2 に格納する．これにより，指定したい配列の指定したいインデックスのアドレスを指定することができる．
  5. VAR と GR2 から実行アドレスを計算し，GR1 に格納している値をストアする．

  という流れで CASL コードを追加する．

2.2 式の処理方法

式の処理によって生成される CASL は，式に関係演算子が含まれる時と含まれない時に分類される．

- 関係演算子が含まれない時，構文定義”単純式”内で処理された CASL コードが追加される．このとき，単純式によって得られる値はスタックに PUSH されている．
- 関係演算子が含まれる時，一つ目の単純式と二つ目の単純式内で CASL コードが追加された後に，関係演算子の処理を実行する CASL コードを追加する．関係演算子を処理する CASL コードは以下のようになる．

  1. スタックから GR1 と GR2 にそれぞれ一つ目の単純式と二つ目の単純式の結果を POP する．
  2. GR1 と GR2 の値を比較する．このとき，CPL または CPA を使用する．
  3. それぞれの関係演算子によって，TRUE X ラベルにジャンプする．X はラベルの番号を表す.
  4. 比較結果が false の時の処理を記述する．具体的には，GR1 に#0000 を格納して BOTH Y ラベルにジャンプする．Y もラベルの番号を表す
  5. 比較結果が true の時の処理を記述する．TRUEX ラベルを付けた命令 NOP の後に，GR1 に #FFFF を格納する．
  6. BOTH ラベルを付けた命令 NOP の後に，GR1 の内容を PUSH する．

  という流れで処理が行われる．それぞれの処理を分岐させる方法については実装方法で記述する．また，TRUEX の X と BOTHY の Y はラベルの管理方法で記述する．

単純式によって生成される CASL は加法演算子が含まれる時と含まれない時に分類できる．

- 加法演算子が含まれない時,

  1. 構文定義”項”内で処理された CASL コードが追加される．このとき，項によって得られる値はスタックに PUSH されている．
  2. 符号が「-」のとき，スタックから単純式の値を GR2 に POP し，GR1 に 0 を格納する．そして，GR1 から GR2 の値を減算し，GR1 に格納する．これにより，0-G R2 という計算が実行され，負の数が実現できる．そして計算後の負の数を PUSH する．

- 加法演算子が含まれる時，加法演算子が含まれるときの処理の後に

  1. 再び構文定義”項”内で処理された CASL コードが追加される．
  2. スタックから加法演算子の第二項を POP する．
  3. スタックから加法演算子の第一項を POP する．
  4. 各加法演算子に対応する命令 (ADDA,SUBA,OR) を実行し，結果を GR1 に格納する．
  5. GR1 の内容をスタックに PUSH する．

  という流れで CASL コードが追加される．この処理は構文定義より，0 回以上繰り返される．

項内では，単純式内の”項”で追加される CASL コードが”因子”で追加されるようになり，各加法演算子に対応する命令が各乗法演算子に対応する命令に変更される．変更後の CASL コードは以下の表 1 のようになる．

”*”	CALL MULT の後，GR2 を PUSH
”/”または”div”	CALL DIV の後，GR2 を PUSH
”mod”	CALL DIV の後，GR1 を PUSH
”and”	AND の後，GR1 を PUSH

表 1: 単純式から変更された各乗法演算子に対応する処理

因子では,「変数」,「定数」,「(式)」,「not 因子」の 4 種類に処理が分かれる．

- 変数の時,

  1. VAR からのアドレスの差を計算し，VAR とアドレスの差を足し合わせた番地の値を GR1 に格納する．
  2. GR1 の値をスタックに PUSH する．

  という流れで CASL コードが追加される．

- 定数の時は，構文定義”定数”内でスタックに値が PUSH される．PUSH される内容は以下の表 2 のようになる．

数値	その数値の値
長さが 1 の文字列	GR1 に格納された 1 文字の文字列
true	#FFFF
false	#0000

表 2: PUSH される内容

- (式) の時は，構文定義”式”内で処理された CASL コードが追加される．
- not 因子のとき,

  1. 因子内で処理された CASL コードが追加される．
  2. スタックから因子の値を GR1 に POP する．
  3. #FFFF と GR1 の値を XOR し，その結果を GR1 に格納する．これにより，GR1 の各ビットを反転させることができる．
  4. GR1 の値をスタックに PUSH する．

  という流れで CASL コードが追加される．

2.3 手続きの呼び出し方法

手続きの呼び出しは,

1. 実パラメータをスタックに PUSH する．実パラメータが無ければ何も PUSH せずに次の処理を行う．
2. 各手続きに対応するラベルを CALL する．CALL をするとスタックにプログラムレジスタの値を，つまり次に実行すべき命令語の先頭アドレスを PUSH する．
3. GR1 にスタックポインタを代入し，仮パラメータの個数を加算する．これにより，GR1 が CALL を実行する前に PUSH した最初の実パラメータの値が格納されている番地を指すようになる．実パラメータがなければ 0 を加算し，以下の 4 からの処理は実行しない．

という流れで CASL コードを追加する．

2.4　ラベルの管理方法

CASL コード内で使用されるラベルは以下の表のようになっている．

CASL	CASL 文の記述の開始を表す
BEGIN	CASL 文のプログラムの先頭を表す
LOOP	while の条件式についての処理の先頭を表す
TRUE	while，if の条件式が true の場合の処理を記述
BOTH	条件式の結果が true であればその後の構文定義”複合文”， false であれば ELSE にジャンプする処理の先頭を表す
ENDLP	while の終了を表す．その後は while 文以降の処理が記述される
ELSE	if-else でない場合は if の終了を表す．その後は if 文以降の処理が記述される if-else である場合は else 内の構文定義”複合文”の処理が記述される
ENDIF	if-else 文の終了を表す．その後は if-else 文以降の処理を記述
PROC	各副プログラムの先頭を表す
VAR	メモリの番地の先頭を表す
CHAR	二文字以上の文字列の情報を格納

表 3: ラベルの種類と内容

LOOP，TRUE，BOTH，ENDLP，ELSE，ENDIF，PROC，CHAR はその後，構文定義内で処理された順に 0 から 1 ずつ数値が増やされ分類される．即ち，if 文が入れ子構造になっている場合は，ELSE や ENDIF が登場するより前に複合文内での if が処理されることになるため，CASL コード内で登場するラベルの順番は，TRUE0，BOTH0，TRUE1，BOTH1，ELSE1，ELSE0 のようになる．

2.5　分岐の処理方法

分岐の処理においては，構文定義”文”における if 文，if-else 文，while 文内で処理が行われる．

- while 文のとき，

  1. LOOP NOP を追加し while 文の開始位置を決める
  2. 2.2 の構文定義”式”における，関係演算子が含まれる場合の処理により CASL コードが追加される．
  3. スタックから条件式の結果 (true または false) を GR1 に POP する．
  4. GR1 と#0000(false) が一致しているかを CPL により確認し，一致してればゼロフラグが立つ．
  5. JZE により，ゼロフラグが立っていれば ENDLP にジャンプする．
  6. 構文定義”複合文”内で処理された CASL コードが追加される．
  7. LOOP にジャンプする. 即ち，1 からこの処理を再び行う．
  8. ENDLP NOP を追加し while 文の処理が終了する

  という流れで CASL コードが追加される．

- if 文の場合，

  1. 2.2 の構文定義”式”における，関係演算子が含まれる場合の処理により CASL コードが追加される．

2. スタックから条件式の結果 (true または false) を GR1 に POP する．
3. GR1 と#0000(false) が一致しているかを CPA により確認し，一致してればゼロフラグが立つ．
4. JZE により，ゼロフラグが立っていれば ELSE にジャンプする．
5. 構文定義”複合文”内で処理された CASL コードが追加される．
6. ELSE NOP を追加し if 文の処理が終了する．

という流れで CASL コードが追加される．

- if-else 文の場合，

1. if 文の 1 から 5 と同様の処理が行われる．
2. ENDIF にジャンプする．
3. ELSE NOP を追加し，その後 else の際の構文定義”複合文”内で処理された CASL コードが追加される．
4. ENDIF NOP を追加し if-else 文の処理が終了する．

という流れで CASL コードが追加される．

2.6 レジスタやメモリの利用方法

各レジスタの利用方法は以下の表のようになる．

GR1	様々な用途に用いられるバッファ
GR2	様々な用途に用いられるバッファ
GR3	スタックに PUSH されている実パラメータのアドレスの内容を格納
GR6	0 番地のアドレスを格納
GR7	LIBBUF のアドレスを格納
GR8	スタックポインタ

表 4: 各レジスタとその利用方法

また，メモリの仕様及び利用方法は以下のようになる．

- メモリの仕様は VAR がメモリの先頭を表し，まずグローバル変数の領域が DS により確保される．以降，各副プログラムごとに仮パラメータ，ローカル変数の領域が格納される．また，変数の領域の直後のアドレスから，二文字以上の文字列を DC により格納される．各文字列は CHAR とその後に 0 から始まる数値を結合したものをラベルとして持つ．最後に，LIBBUF というラベルとともに DS により 256 個の領域が確保される．
- メモリの利用方法は 2.1 のように参照する変数とメモリの先頭である VAR のアドレスの差を計算してその差を VAR と加算することで実行アドレスを計算する．具体的な処理方法は 3.2 で記述する．

2.7 スコープの管理方法

各仮パラメータのスコープの管理は，2.3 の副プログラムが呼び出された後に処理される．具体的には，

1. 「PUSH された実パラメータのアドレス」の中身即ち実パラメータの値を GR2 に格納し，DS で確保した領域にその値を格納する．

2. GR1 の値を 1 減らす．これは，4 で指定した実パラメータの次のアドレスを表す．3 の時点で最後の実パラメータでなければ GR1 は次の実パラータのアドレスを，3 の時点で最後の実パラメータの値であれば GR1 は 2 のプログラムレジスタの値を指す．

3. 1,2 を繰り返す．繰り返しが終了すると GR1 にプログラムレジスタの値を格納する．この後は複合文の CASL を記述する．

4. スタックポインタに実パラメータのサイズを加算した値を GR8 に格納する．これにより，1 で PUSH する前のスタックのアドレスをスタックポインタが指すようになる．最後に，GR1 をスタックポインタに格納し，RET する．これにより，プログラムレジスタの値に格納されているアドレスの処理が実行されるようになる．

という流れで CASL コードが追加される．

2.8　過去の課題プログラムの再利用

手続き内で定義された変数に関する情報 (識別子名，型，サイズ) はリスト型のグローバル変数として定義している．具体的には，グローバル変数について格納する global_variable_table とローカル変数について格納する local_variable_table を定義している．ローカル変数表は直前に定義した副プログラムのローカル変数の情報のみを格納するため，複数の副プログラムを定義した場合，以前の変数の情報が失われてしまっていた．そこで，新たにローカル変数表を格納するリスト local _variable_table_list を作成することで複数のローカル変数表の情報を記憶できるようになった．

3　実装プログラム

ここでは，2 で記述した CASL コードを記述するためのプログラム内での処理について記述する．

3.1　各ラベルの数字の管理方法

2.4 で記述した LOOP，TRUE，BOTH，ENDLP，ELSE，ENDIF，PROC はそれぞれ int 型のグローバル変数として X_count として定義した．X は前述の各ラベルが入る．また，各構文定義を処理するメソッド内ではローカル変数 this_X_count に X_count を代入し，処理が終わると使用したグローバル変数を 1 インクリメントする．これにより，入れ子構造の際のラベルの管理は以下の図のようになる．

```
1   if a > 1 then  //条件式内でTRUE0,BOTH0が確定
2   begin  //ELSE0が確定
3       if b > 2 then  //TRUE1,BOTH1が条件式内で確定
4       begin  //ELSE1が確定
5           //処理
6       end
7       else
8       begin  //ENDIFが確定
9           //処理
10      end;
11  end;
```

図 1: 入れ子構造におけるラベルの管理

上図のラベルが確定しているのは，this_X_count に代入されたことを表す．11 行目の「end;」で ELSE0 に関する処理を記述するが，ローカル変数として定義しているため，内側の if 文の影響を受けて ELSE2 となるのではなく，ELSE0 として記述することができる．

3.2 変数の参照の際の VAR とのアドレスの差の計算方法

2.6 より，メモリにはグローバル変数，仮パラメータ，ローカル変数の順で領域が確保されている．また，仮パラメータとローカル変数は一つの副プログラムに対して 1 セットで確保されるため，二つ以上の副プログラムが定義されている場合は，グローバル変数，仮パラメータ，ローカル変数，仮パラメータ，ローカル変数… という順番で領域が確保される．また，参照する変数は,「グローバル変数」,「ローカル変数または仮パラメータ」,「副プログラム内で参照されるグローバル変数」,「添字付き変数」の 4 つに分類できる．それぞれの処理は，以下のように行う．

- グローバル変数の場合は，グローバル変数表から順に参照していき，対象の識別子と変数表内の識別子が一致するまで各変数のサイズを加えることで計算する．具体的には，以下のようなプログラムで実行される．

```
1 for(variable list : global_variable_table) {
2   if(list.getName().equals(name)) {
3     break;
4   }
5 sum = sum + list.getSize();
6 }
```

  これにより，対象の識別子と変数表中の識別子が一致するまでのサイズを計算できる．

- ローカル変数または仮パラメータの場合は，各ローカル変数表において，対象の変数までのサイズとグローバル変数表の全ての変数のサイズを足し合わせる．このとき，各副プログラムのローカル変数を格納した表は定義された副プログラムの順に参照される．つまり，副プログラム f1 と f2 が存在し，f2 中の変数を参照したい場合はまず f1 のローカル変数表のサイズを全て足してから f2 の変数表を参照し，対象の変数までのサイズを計算する．その後，グローバル変数表の全変数のサイズを足し合わせる．具体的には，以下のようなプログラムで記述される．

```
1  if(local_variable_table_list.size() != 0) {
2    for(ArrayList<variable> table : local_variable_table_list) {
3      for(variable list : table) {
4        if(list.getName().equals(name) && (count == proc_count)) {
5            local = true;
6            for(variable list1 : global_variable_table) {
7                sum = sum + list1.getSize();
8              }
9            break;
10         }
11         sum = sum + list.getSize();
12     }
13       count++;
14     }
15 }
```

  local は，false に初期化されている boolean 型の変数であり，count は 0 に初期化されている int 型の変数である．count は現在何番目に定義したローカル変数表を参照しているのかを表しており，4 行目の条件式より，proc_count と一致した時のみグローバル変数の全サイズを足し合わせる．local は前述の処理が起こったとき，即ち参照する変数がローカル変数であったときに true となる．

- 副プログラム内でグローバル変数を参照する場合は，まず上記のローカル変数の参照と同様の処理を行う．そして，以下の処理を追加で行う．

```
1 if(local == false) {
2   sum = 0;
3   //グローバル変数と同じ処理を行う
4 }
```

ローカル変数表に対する処理を事前に行い，sum の値が変更されているのでまず sum を 0 に初期化している．そして，グローバル変数のときと同様の処理を行っている．

- 添字付き変数の時は，上記と同様の方法で計算してから 1 を引く．

のように分けることができる．

3.3 副プログラムの CASL コードの処理順序

副プログラムの処理は，構文解析や意味解析を行う際は，

構文定義”副プログラム”内の処理→構文定義”プログラム”内の複合文の処理

という順で処理が実行される．しかし，CASL コードの場合は，

構文定義”プログラム”内の処理→構文定義”副プログラム”内の複合文の処理

という順でコードが追加される．
この違いを解決するために，グローバル変数として”String 型のリスト”のリスト proc_casl_list を定義した．これには，構文定義”副プログラム宣言群”の一回の繰り返しの中で追加された CASL コードが格納される．また，このとき resultData の CASL コードは削除されている．
CASL コードの追加方法は，各副プログラム宣言の前と後に resultData のサイズを size() により取得し，subList() を使って抜き出した．また，resultData に proc_casl _list の内容を追加するのは構文定義”プログラム”内の複合文の後である．これにより，上記の CASL コードの順番で追加することができる．

3.4 添字付き変数への代入の際の処理順序

添字付き変数への代入は，構文解析や意味解析を行う際は，

変数名 (配列名) →式 (添字) →右辺の式

という順で処理が実行される．しかし，CASL コードの場合は，

右辺の式→式 (添字) →変数名 (配列名のアドレス計算)

の順に処理が実行される．
この処理の違いを解決するために，グローバル変数として”String 型のリスト”のリスト left_index_casl_list を定義した．これには，構文解析における”式”内で追加される CASL コードを格納しておく．そして，CASL コードの”右辺の式”の処理が行われてから resultData に追加する．
具体的な追加の方法としては，3.3 で記述した proc_casl_list への追加方法と基本的には同じである．ただ，添え字付き変数への代入における添え字の”式”のみに対して処理を行いたいため，添え字付き変数への代入であることを示すフラグをグローバル変数として定義した．そして 2.1 のように添え字付き変数への代入の処理を追加していく．
ところで，添字を表す構文定義”式”内で”(式)”が含まれる時，即ち”式”の中に入れ子構造で再び”式”があ

る場合，上述の通り添字を表す”式”で追加される CASL コードは一旦別のリストに格納される．このとき，どの”式”の処理が終了した段階で CASL コードを別のリストに格納するかを判定する必要がある．そこで，first_index_flag というグローバル変数と，first_flag というローカル変数を定義する．そして，添字を表す構文定義”式”の処理メソッドを呼び出す直前で前者を true に設定する．メソッド内では，後者に前者の値を格納し，前者を false とする．これにより，このメソッド内では first_flag は true となり，今後入れ子構造になっている”式”が呼び出されたときはグローバル変数が false となっているため first_flag は false となる．これにより，first_flag を使うことでこの問題を解決できる．

3.5　スコープ管理の処理

スコープ管理は，2.7 より，実パラメータのサイズと，参照する実パラメータのアドレスを計算する必要がある．

- 実パラメータのサイズは，3.2 と同様にしてローカル変数表のサイズを足し合わせ parameter_size に格納して実現した．このとき，VAR からのアドレスの差ではなくサイズの合計のみ知ることができればよいので直前のローカル変数表のみを参照すればよい．
- 3.2 の仮パラメータの処理を行い，sum に格納した．

また，2.7 の 1 と 2 は仮パラメータごとに繰り返して処理されるため，上記で計算した sum の値を繰り返しごとに 1 ずつ増やしていくことで実現した．

4　考察

今回の課題を通して，特定の場合の処理を行う場合はフラグを使用することで処理を分岐させた．フラグを用いて処理を分岐させること欠点は,「フラグの数が増えると，プログラムの処理の分岐が増加し可読性が下がってしまうこと」が挙げられる．この欠点を解決するための工夫として，以下のようなものが考えられる．

- フラグの名称だけで何を目的としたフラグなのかが分かるようにする．例えば，flag1 のような名称にしてしまうと，何のためのフラグなのかが分からなくなってしまうが,「isGrobalVariable」等の名称にすれば，true と false がこの変数名への返答となり，名称を見るだけで true の場合はグローバル変数について記述するということが分かるようになり，可読性が向上する．
- 各フラグを構造体として定義する．これによりフラグの数が増加した際も管理しやすくなる．また，新しいフラグを追加する際の修正箇所が限定され，可読性が向上する．

5　感想

この講義全体を通して，計画的に課題を進めることができたと感じた．今後の学習及び研究においても長期的に計画を立てて学習を進めていきたいと思う．

参考文献

[1] 言語処理工学 A 講義スライド

[2] 情報科学演習 D 指導書