課題名　：　情報科学演習 D　課題 4 レポート

氏名　：　ブアマニー　タンピモン
学籍番号　：　09B20065
メール　：　u946641i@ecs.osaka-u.ac.jp

提出年月日　：　2023 年 1 月 27 日

1　システムの仕様

第 1 課題には字句解析で Pascal プログラムを単語に区切り、第 2 課題に渡して構文解析によりプログラムが構文的に合ってるかどうかを確認する。また、第 3 課題では制約検査を行い、変数が未定義、二重定義、また型の不一致か確認する。そのため、この課題は最後のフェーズ、CASL-II で記述された目的コードのコンパイラを作成する。このコンパイラは、課題 1 から字句解析した Pascal 風プログラムを入力され、ファイルが見つからない場合は「File not found」を標準エラーに出力し、構文解析で間違いがあるなら見つけた行 X を「Syntax error: line X」で標準エラーに出力し、意味解析で間違いがあるなら見つけた行 X を「Semantic error: line X」で標準エラーに出力する。しかし、エラーがない場合はは CASL II で記述したプログラム (cas ファイル) を出力する。開発対象となるコンパイラのメソッドは Compiler.run(String,String) で一つ目の String は入力ファイル名で、2 つ目の String は出力のファイル名である。

2　課題達成の方針と設計

2.1　変数の代入の方法

CASL II は変数の記憶領域にスタックという駆動レコードが存在する。そこで実行中に次のトークンに繋がるように保存される。確保 ST で変数の代入ができるために、代入する番地が GR2、代入情報が GR1 に保存して「ST GR1,VAR,GR2」をコマンドする。代入文は 2 つの部分があり、「左辺 := 式」である。トークンの読み込むは左から右で、まずは左辺を処理する。左辺は変数であり、変数宣言の時に変数名、変数番地 (star index)、変数の型とサイズなど保存される。Integer,Char,Boolean という標準型ならサイズが 1 であるが、Array はその長さのサイズが持ち、変数が定義されるたびに Start index の数が上がる。

図 1 の変数表で見れる通り、例として a が 10 語であり、a[1]-a[10] は番地 0-9 に確保する。そのため、番地 1-9 が埋まってるため次の変数 b が番地 10 から始まる。そのため、代入文が来ると、どの番地に代入するかは左辺の変数から変数表で変数番地を確認して GR2 に保存する。標準型の変数は Starting index そのまま使うことができるが、配列型の場合は番地が数個あるため、何個目に確保するかが計算できるため starting no. という添字の最小値を保存し、一個目の番地が配列の添字の最小値が保存する。そのため、左辺の変数がどの番地に保存するのかは、見つけたい変数の添字と最小値の差を計算し、GR2 という番地を決める値に足す。例として、「b[7]:=25」の左辺は b[7] であり、まずは添字 7 は添字の最小値 4 から 3 番地差があり、3 を GR2 にいれる。それで b の変数は番地 10 から始まるため、１０を足して、b[7] は番地 13 にあるとわかる。また、右の式について説明する。式であることで、結果は 3 つの種類に表すことができる、これは「integer,char,boolean」である。式がどれだけ複雑であっても計算が終わるとスタックに保存する。そのため、POP GR1 で結果が GR1 に格納するようになる。しかし、それは ST の前に行わないといけないため、左辺の処理はすべて他のリストに保存し、GR1 に式の結果が保存してから左辺の処理が出力させる。

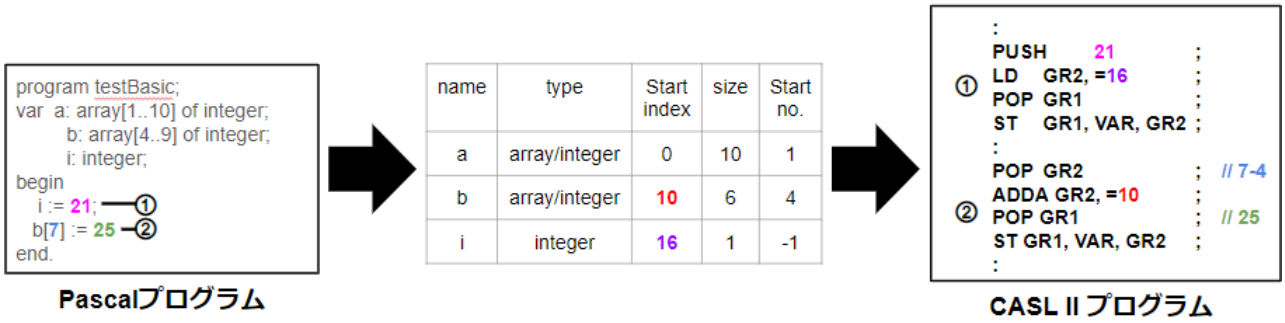

図 1: 変数の代入例

2.2　式の処理方法

図 2 は式の構文定義を説明する。式が処理終わるとスタックに保存するのが目標である。それだけでなく、小分ごとの計算や、変数定数定義で新しい結果がでるたび汎用レジスタ GR は算術、論理、比較ようであるためにどのレジスタが使われるまま他の情報を上に書かれる可能性があるため、データを保存し次のトークンに行く前はスタックに格納する。

```
式 = 単純式 [ 関係演算子 単純式 ].
単純式 = [ 符号 ] 項 { 加法演算子 項 }.
項 = 因子 { 乗法演算子 因子 }.
因子 = 変数 | 定数 | "(" 式 ")" | "not" 因子.
```

図 2: 式の構文定義

2.2.1　因子の対応

まず、式の一番小さい構成要素「因子」の変数または定数を見る。変数は前に定義したと考えるため、変数表により変数の番地を見つけて、値をスタックに保存する。定数の場合、スタックにプッシュする前の処理は定数の種類により対応が異なる。定数が数字の場合は「PUSH ＜数字＞;」などその値のままプッシュすることができる。定数が文字の場合も、「LD GR1, =＜文字＞」で一旦 GR1 に保存し、GR1 の状態でプッシュした。しかし、文字列の場合は直接レジスタに入れれないため、定数として定義しないといけない。そのためまず、constant リストに存在するのか確認して、まだ存在しないなら新しく入れる。それで、その文字列の長さを先にいれて、それで「LAD GR2, CHAR<定数番号>;」でその定数の番地を GR2 にいれて、「PUSH 0,GR2;」で中にある値をプッシュする。最後にすべての命令が CASLII に変更し終わると、「CHAR ＜リストの順番＞ DC <文字列>;」で constant にある定数を定義するべきである。しかし、注意点として文字列が「"true","false"」の場合それは文字列でなく、boolean である。これは数字と同じく boolean の状態「#0000,#FFFF」をスタックにいれる。それでそれぞれの型を「integer,char,boolean」リターンする。

しかし、因子でも一番小さい構成要素でなく、「(<式>)、not<因子>」でも行えるため、「(<式>)」は結果が変わらないためそのままスタックに保存する。しかし「not<因子>」は<因子>の結果が逆にするため、結果を False と XOR して、状態が逆してからまたスタックに保存する。

2.2.2　項の対応

項は乗算演算子に対応するため、まずはスタックに保存した 2 つのデータが順番に GR2 と GR1 にポップし、演算子により「*」なら「CALL MULT;」で結果は GR1 にある。しかし、「div , / , mod」なら「CALL DIV;」で、商は GR2, 余りは GR1 に保存する。また、演算子が「and」なら「AND GR1, Gr2;」で計算し、結果は GR1 に保存する。それでまた結果がスタックにプッシュする。

項の構文は「項 = 因子　乗法演算子　因子」であり、項の計算は左から右の順番で計算できるため、乗算演算子の命令が次の因子が処理した後に行う。例えば、「因項 1 = 因子 1　乗法演算子 1 因子 2 乗法演算子 2 因子 3 」の場合、順番で因子 1, 因子 2 が処理してから上に説明した乗法演算子 1 の処理が行う。それで「因子 1　乗法演算子 1 因子 2」の処理結果がスタックの最後に保存されるため、因子 3 が処理されて結果がまたスタックにプッシュしてから乗法演算子 2 の処理が行う。

2.2.3　単純式の対応

次は単純式、または加法演算子の処理について説明する。単純式の構文は「単純式 = [符号]　項　加法演算子 項」である。まず、符号は（+/-）で、符号がないまたは+なら結果が変わらないで何もしなくていいが、-(負)の場合、項の結果が逆になる。これはまず、項を処理して結果がスタックに入れると、結果が GR2 にプッシュし、GR1 が 0 に入れれば「SUB GR1,GR2;」で項の負の結果が GR1 に保存される。これでまた GR1 をプッシュする。

単純式の加法演算子の処理は項の対応と似ていて、左から右の対応であるため、最初の２つの項が処理されてから加法演算子を行って結果がスタックにいれる。これで３つ目の項が処理され、スタックに入れるとまた加法演算子を行ってはじめに計算した結果と３つ目の項が演算されて、次々の項と加法演算子の処理が行う。また、加法演算子は３つの種類があり「+,-,or」でそれぞれの CASL コマンドが「ADDA/SUBA/OR GR1,GR2」で、加法演算の結果が GR1 に保存するためそれをスタックにプッシュする。

2.2.4　式の対応

式の対応は関係演算子を処理する。式の構文は「式 = 単純式 [関係演算子　単純式]」であり、[関係演算子　単純式] がない場合式は単純式の結果を表示する。これは「boolean,integer,char」が可能である。しかし [関係演算子　単純式] がある場合、結果が２つの単純式の関係演算であり、True か False という boolean のみが出力できる。図 3 で見れる通り、まずはポップでスタックにある２つの単純式の結果が順番に GR2,GR1 にいれて CMP で比較する。CMP の比較には GR1-GR2 にして、結果に基づき FR に変更する。結果が 0 なら ZF が 1 にして負数なら SF が 0 から 1 に変更する。それでその結果により分岐ジャンプで結果を決める。関係演算子は「=,<>,<,<=,>,>=」6 つの種類があり、それらの演算子を持つ式が特殊の番号 X(例:1) が順番に定義されて、分岐するときに使われる。CASL のプログラムは条件分岐命令を使って、条件が満たす場合は TRUEX(例:TRUE1) にジャンプし GR1 に#0000(<=,>=の場合は#FFFF) を入れる。満たさない場合はジャンプしないで、GR1 に #FFFF(<=,>=の場合は#0000) を入れる。それで２つともは BOTHX(例:BOTH1) に繋がり、GR 1 にある結果をスタックにプッシュする。「単純式 = 単純式」の場合は JZE で２つの単純式が同じか確認する。「単純式<>単純式」は JNZ で２つの単純式が同じではないことを確認する。「「単純式</>=単純式」」は JMI で左は右の単純式より小さいを確認するが、>=の場合は GR1 に入力される値が>と逆である。同じく、「「単純式>/<=単純式」」は JPL で左は右の単純式より大きいのを確認するが、<=の場合は GR1 に入力される値が>と逆である。

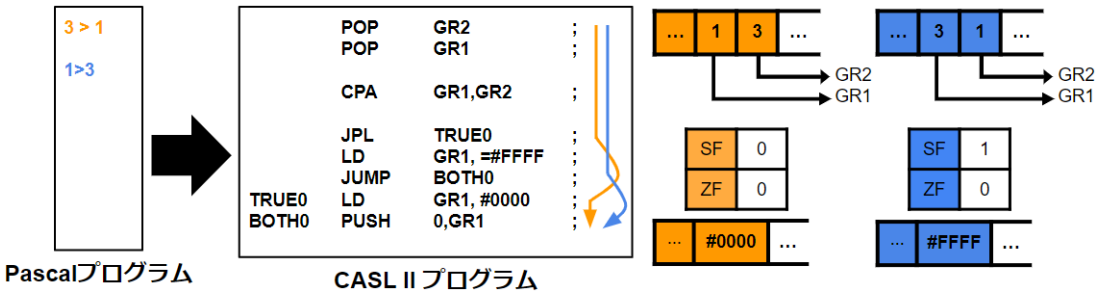

図 3: 式の対応例

2.2.5　手続きの呼び出し方法

手続きに必要な命令は CALL と RET である。CALL は RET が来るまでオペランドに記入した番地にコマンドを読み込むということで、最後の番地をスタックに保存する。RET は前に CALL した番地に戻るという意味である。それにレジスタは２つありスタックの最上段のアドレスを保持しているレジスタ SP と次に実行するべき命令語の先頭アドレスを保持するプログラムレジスタ PR である。図 4 で見れる通り、手続きが呼び出されるとまず定義される引数を順番にプッシュし、CALL を呼び出すと RET がスタックにプッシュされて、PR が実行アドレス行 (2) になる。また、行 (2) には引数から仮パラメータにいれる必要がある。それができるように、まず手続き内の変数表を作って、それで GR1 が GR8-(変数の数) にすると、GR1 が引数の 1 個目に指定する。それで順番で読み取り、スタックにプッシュすると、仮パラメータが入力させる。これは、手続き内に変数が変更されても本プログラムに影響しないようになる。また、手続き内が実行終わると、本プログラムに戻る前にまず仮パラメータをすべての削除するようにすると GR8 が RET アドレスが保存する番地に戻る。最後に RET 命令を使うと RET アドレスが PR に入れられ、スタックから抜き出される。

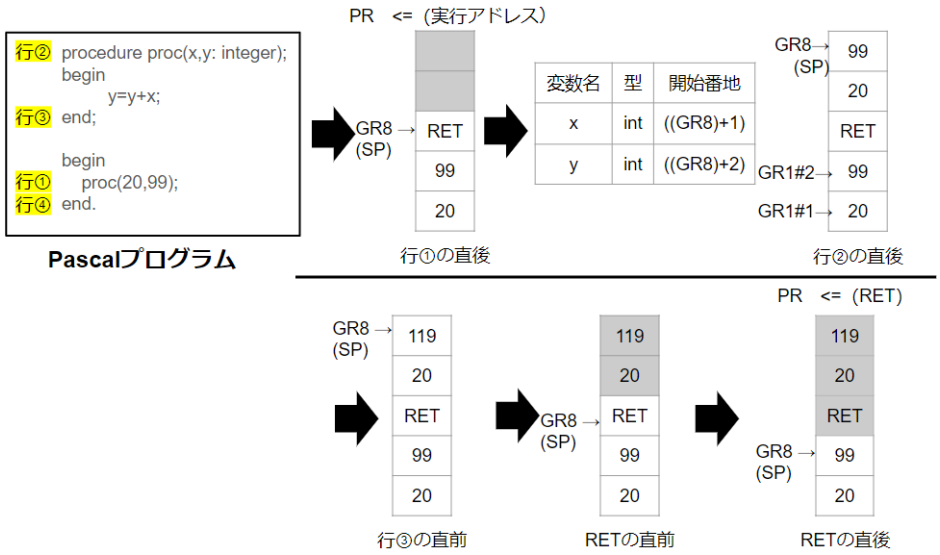

図 4: 手続きの呼び出し方法

2.3 分岐の処理方法

分岐は 2 つあり、「IF 文と WHILE 文」である。その 2 つの文の似ている部分としては Boolean が出す式が定義され、それが True なら一つの分岐に行って、False ならもう一つの分岐に行くことである。違いは IF は一回式を確認し、満たしたら一つの命令に行き、満たしてないならもう一つの命令に行くが、WHILE はその条件を満たすまで命令を繰り返すことである。

2.3.1 IF 文

まず、IF 文の CAS プログラムを説明する。式の処理により、結果がスタックの最新にある、その値を GR1 に POP し False である#FFFF と比較する。それで「JZE　ELSEX」で同じであると else の構文にジャンプする。それでジャンプしないなら True であるため、if 内の複合文を翻訳する。else がない場合、複合文が翻訳し終わると「ELSEX NOP」があるとそこでまた繋がるが、else がある場合、ちゃんと if と else の部分を分けないといけないため、ENDIFX という繋がる行を整って、IF の複合文が終わると「JUMP ENDIFX」で ENDIFX 行に行く。JUMP 行の後に True ならもう入れないためそこれ「ELSEX　NOP」で else 複合文を翻訳はじまって、それを終わってから「ENDIF NOP」行を作成し、 2 つの分岐が繋がる。

2.3.2 WHILE 文

次は WHILE 文を説明する。WHILE 文はループに入るたびに式の状態を確認しないといけないため、まず「LOOPX NOP」でループを始まり、式の処理し、結果がスタックの最後にある。そのため、GR1 にポップして#FFFF(False) と比較すると同じなら「JZE ENDLPX」で ENDLPX 行にジャンプする。しかし、まだ条件内なら結果がが#0000(True) なので、#FFFF と異なり、ジャンプしないで次の行に行く。そこで While の複合文を翻訳する。まだループに出れないためそのあと「JUMP　LOOP」で強制で WHILE ループに戻る。そのため、ループがまだ終わらないと次の行にいけないため、次の行が「ENDLPX NOP」にして、WHILE 分岐の集合する場所になる。

2.4 レジスタやメモリの利用方法

2.4.1 レジスタの利用

レジスタは 3 つの種類があり、GR、PR と FR である。GR（General Register）で、GR0-GR8 の 9 個が利用できる。この課題には基本 GR1,GR2 が計算や、出力利用される。GR3 は臨時変数として、仮パラメータの挿入

するときに GR1 が番地を保存するためレジスタが足りない時のみ使う。GR6 と GR7 はサブルーチンライブラリに利用する。そのため、GR6 は 0，GR7 はバッファ先頭番地を示すように初期化した。また、GR8 は SP（スタックポインター）を兼ねて、スタックの最上段のアドレスを保持しているようになる。PR(プログラムレジスタ) は次に実行すべき命令語の先頭アドレスを保持する。基本はコマンドの後で次の行に動き出すが、手続きに行きたい時は次の行ではないため、CALL 命令で指定アドレスに行き、また利用しおわると RET で GR8 にあるリターン番号にもどる。FR（フラグレジスタ）は OF（オーバーフローフラグ)、SF(サインフラグ)、ZR(ゼロフラグ) であり、順番に計算結果がオーバーフローか、負数がでるのかと、ゼロになるのかを表示する。基本はそのフラグにより分岐命令で条件によりジャンプをする。

2.4.2　メモリの利用方法

プログラムのメモリの配置は図 5 で見れる通り、一つのプログラムは主記憶の容量が 65536 語であり、アドレスは 0-65535 である。大域変数はデータ領域に保存しコマンドの最後表に「VAR　　DS X;」で X 語の領域確保する。また、サブルーチンライブラリを利用するため「LIBBUF　　DS 256」でサブルーチンライブラリ用のバッファ領域も定義しないといけない。また、局所変数はスタックに保存し、ブロックが終了時に解放する。しかし、この課題には任意に解放する局所変数がないためヒープは使われてない。データ領域について、コードに左から読み込むと、変数が入るたびに変数表にその変数の VAR から離れた先頭番地を保存する。変数のメモリ管理は変数表に行う。基準型の変数はそのものであって、1 個は 1 語であるが、配列変数の場合は先頭のみ保存され、本当のサイズは配列の個数である。配列変数の定義するときに、添字の最小値と最大値が定義されて、「最大値-最小値＋１」が配列のサイズになる。また、X 添字の配列の番地を見つかるように変数表から最小値の添え字の番号が見つかり、それで X は最小値から「X-最小値」が離れるためそれで配列変数の X 添字の番地を計算できる。また、手続きを呼び出すときに、手続き内にある変数が定義されるが、手続きに出ると書き込まれる。

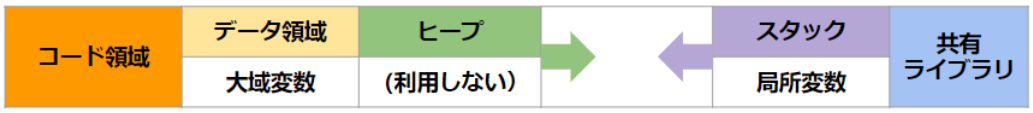

図 5: 変数の記憶領域

2.5　ラベルの管理方法

CASL プログラムは複数のラベルが必要である。コマンド番地が順番に読み込むでなく「If-else 文、手続き文、while 文、関係演算子あり式」が使われるとばジャンプを利用し、ラベルに行く必要がある。それぞれのラベルの名前が異なるように、名前を指定する変数が以下のように利用される。

利用する文	ラベル名 (X は数字)	X を管理する変数
If-else 文	IF(X) ENDIF(X) ELSE(X)	countifelse
手続き文	PROC(X)	countproc
while 文	LOOP(X) ENDLP(X)	countloop
関係演算子あり式	TRUE(X) BOTH(X)	counteq

表 1: ラベルの管理方法

但し、If-else 文と while 文は複数の同じ文であると、重なることで最後の変数だけじゃ足りないので、スタック ifstack,loopstack を定義し、新しい if/while ブロックが入ると新しい変数をプッシュするし、出るとポップする。そのため、現在の文がスタックの最新の変数を使える。

2.6　スコープの管理方法

課題３で説明した通り、このプログラムはスコープルールに基づいて変数表を作る。そのため、変数表は動的に管理して、変数表がスタックに保存され、新しい手続きに入ると新しい変数表を作成しスタックに入れる。その手続きから出ると、最新の変数表をスタックから削除する。そのため、次の手続きが来ると変数の番地が再利用して、メモリーが上手く管理することができる。また、名前の参照は最も新しい変数表から順に検索する。

2.7　過去の課題プログラムの再利用

課題４は課題３のプログラムを再利用し、その２つの課題の違いは課題３が構文解析、意味解析でエラーを検出し、エラーがない場合はＯＫのみ出力する。しかし、課題４は課題３のエラー検出だけでなく、エラーがない場合、プログラムを Pascal から Casl プログラムを作成しないといけない。課題３にはプログラムが Pascal 風言語の構文定義により、ENBF 式の要素で色々なメソッドに分けれた。課題３の検出と課題４のコンパイルは同じ構文を確認するため、はじめから新しいファイルを作成し、構文解析、意味解析しながら、行ごとに CASLII に翻訳する。しかし、途中でエラーを見つかる場合新しいプログラムを消すべきである。

2.7.1　変数表の利用

課題３には変数未定義、変数二重定義の意味解析器のために変数表が作られた。課題４には変数の番地がわかるように課題３の変数表から「開始番地」,「サイズ」,「開始番号」を追加した。詳しくは「2.1 変数の代入の方法」で参考することができる。

3　実装プログラム

この課題には配布データに実行して、20 個のファイルすべて通るようになった。次は課題３から追加した変数や関数の説明である。但し、方針や設計、または考察に説明した変数ー関数は無視する。

3.1　出力の処理: 変数 iswriteln と constant

まず、プログラムが出力できるように、Pascal には「writeln(P1,P2,P3...)」のように writeln メソッドで渡した P1,P2,P3.. という変数、文字列、または数字の複数のパラメータを順番に出力してから最後に \n を出力する。しかし、図 6 で見れる通り casl には複数のパラメータの同時に出力ができなく、一つずつのパラメータがどの種類か確認してから対応する。'test' のような文字列の場合、文字列の長さが GR1 にポップし、文字列が GR2 に保存してから WRTSTR で書き出す。変わりに、数字や文字の場合はその文字を GR2 に保存し、順番に WRTINT か WRTCH で書き出す。Pascal ですべてのパラメータを書き出すと改行を出力装置に書き出し、CASL II も同じく WRTLN で処理する。

また、数字や文字は CASL 内に定義できるが、文字列はできないため、定数を定義する必要がある。定数はすべて CHARX で定義され、X は定義の順番番号である。例えば、文字列'test' を定義する場合は「CHAR0 DC 'test';」 で記入される。それぞれの定数は string のリスト「constant」に保存され、コードがすべて casl に変更されてから最後に入れる。

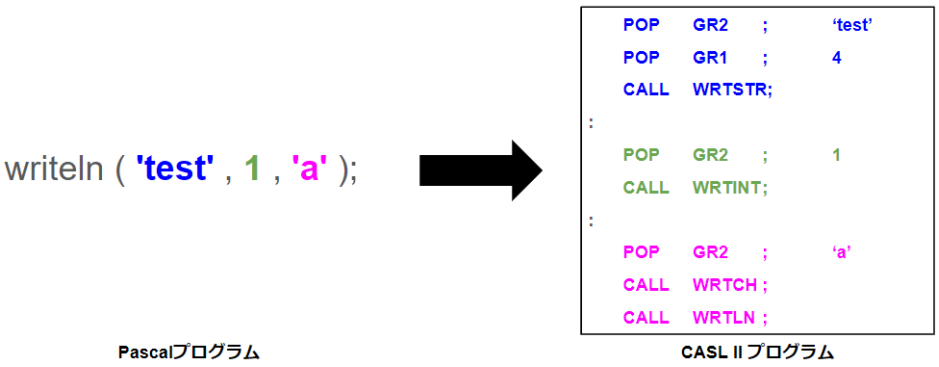

図 6: 課題 3 の仕組み

3.2 その他の変数説明

変数	型	説明
lastmem	int	変数が最後に定義されるメモリ番地
countproc	int	翻訳した手続きの数
counteq	int	翻訳した関係演算子あり式の数
countif	int	翻訳した if-else 文の数
countloop	int	翻訳した while 文の数
iswriteln	bool	writeln の構文内かを指定する
isreplace	bool	replace 内の構文内の指定
issubdecg	bool	手続き内の指定
isreadln	bool	readln 内の指定
isreplaceleft	bool	配列の変数が代入さることの指定

表 2: 変数説明

4 考察と工夫点

4.1 サブルーチンライブラリ (lib.cas)

今回の課題 4 は定義されたサブルーチンを利用する。Pascal から CAS プログラムに翻訳した後、data/cas/lib.cas を利用して、CAS プログラムの最後の部分に打つ。利用方法としては「CALL　<サブルーチン>」でサブルーチンは 10 個あり「MULT,DIV,RDINT,RDCH,RDSTR,RDLN,WRTINT, WRTCG,WRTSTR,WRTLN」である。

4.2 CAS プログラムの順番

Pascal プログラムから CAS プログラムに翻訳すると、「<本プログラム> {<副プログラム>} <変数領域確保><定数定義>」という構成ができるように色々な工夫した。まず、Pascal のプログラムの順番は「<本プログラムの変数定義> <副プログラムの変数定義> <副プログラム> <本プログラム>」であり、プログラムは構文としてトークンを左から右に読み取り、希望の順番にプログラムを書けるように一部のプログラムをどこかに保存する必要がある。

4.2.1 副プログラムの順番処理

副プログラムは本プログラムの後に出力できるように工夫をした。この問題も代入文に似ているような問題があって、「a:=b」は右の式 b の結果を先に処理してから a の番地に保存する。基準型の変数は後で処理してもいいが、添字に式がある可能性がある配列変数は前もってコードの場所を整るのができない。そのため、write 関数を作成し、それぞれの構文の要素の関数に CASL プログラムを書き込むことでなく、その関数で処理する。write 関数は普段に CAS ファイルに 1 行ずつ記入するが、条件により代わりに String の List にいれる。isreplaceleft と issubdec という boolean 変数を確認し現在代理文の左返信、または Pascal の副プログラムに入れるのかを確認する。isreplaceleft が true ならコマンドが rpleftcommand リストに保存し、issubdec が true なら subdeccommand リストに保存する。代入文の場合、後で書き出すため、右の式がわからなくても左辺に適切な代入文を書き込める。それで式が書き出すと printrpleft 関数を呼び出し、rpleftcommand の中身を書き出す。同じく、副プログラムが先に Pascal として読み込まれるが、翻訳されたコマンドがすべて subdeccommand に保存され、本プログラムが実行されてから printsubdec 関数を呼び出し、中の副プログラムを出力する。

4.2.2 定数、変数の順番処理

Pascal プログラムには変数は色々な場所に定数され、変数名を呼び出して色々な場所に使われるが、CAS プログラムは変数名を使わなく番地を呼び出す一つの場所にする必要がある。そのため変数表に変数のサイズ、最初番地を保存する。また、配列変数は最初番地を使うと最小値の添字の番地になるため、最小値も保存して（基準型なら-1)、希望の添字から引算すれば最小値の添字の番地からどれだけ離れるか計算できる。プログラムが終わると printvariable 関数でまとめの確保番地を出力する。また、文字列の定数は Pascal ではコード内でそのまま記入できるが、CAS プログラムでは下に定義しないといけない。そのため、constant リストで文字列を保存し、リストの添字は定数名 (CHARX) の番号 X となる。同じ文字列が複数の行に定義しないように毎回新しい文字列を定義した文字列と確認し、既に存在する場合は存在する定数名を使う。

4.3 発展課題ーテスト改善

4.3.1 配列の確認テスト

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

テストケース１：配列の確認テスト

これは配列の確認するテストケースで、実行する場合意味的解析のエラー部分が行 3,5,6,7 で、4 つが出るべきである。それぞれのエラー部分を説明する。行 3 には変数宣言に配列変数 b を定義して、添字は最小値 10、最大値 2 である。これは添字の順番が逆で、使えない。プログラムがエラーを表示しない場合、代入時に他の変数に代入して影響する恐れがある。解決方法としてはエラー検査するとき最大値は最小値より大きいことをちゃんと確認する。行 5 ,6 は a[1] と a[11] に代入するが、本当は 1 と 11 は a の添字範囲外なので、代入できる場合、エラーが出さないまま他の変数のメモリーに影響する恐れがある。解決方法は最小値と最大値を保存し、毎回配列変数を呼び出すと添え字は範囲内か確認する。また、行 7 のエラーは未代入変数を出力する。このテストケースを防ぐように、変数表に「未代入」という科目を追加し、未代入変数を出力しようとするとエラーを出す。

4.3.2 手続きの確認テスト

```
1  program testBasic;
2  var i: integer;
3      c: char;
4
5  procedure subproc(x: integer);
6  begin
7      writeln(x);
8  end;
9
10 begin
11     subproc(1,2);
12     subproc;
13 end.
```

テストケース 2：手続きの確認テスト

これは手続きの確認テストで、実行する場合意味的解析のエラー部分が行 11,12 に出るべきである。それぞれのエラー部分を説明する。行 11 には手続きの仮パラメータより大くデータを提出する。エラーがない場合、データがすべてスタックにプッシュし、最後に入れるデータのみが仮パラメータに入れられる。また、リターンするとき、追加データがあるためリターン番地が保存する場所とずれているため、エラーが出る、または間違った場所に行くようになる。行 12 のエラーも同じく手続きの仮パラメータより少なく提出する。そのため、リターンアドレスが一つの変数として仮パラメータにいれてしまう恐れがある。解決方法としては関数表に変数の数と型の順番を保存し、手続きを CALL するときにちゃんとプッシュされるか確認する。

5 感想

このプログラムを書く時、途中で色々なエラーを手間取られた。先生が整って下さったテストケースファイルにより、非難でプログラムをデバッグすることができた。しかし、卒業して実際会社で働くときはそれを整えることがなく、自分のコードを確認するように、自分で色々なケースを作成し、エラーが行える範囲をうまく把握しないといけない。これは授業のプログラミングと実際のプログラミングの違いと自分が実感できた。また、レポートを書くことは難しかったとは感じた。この課題は Pascal から casl に変更するため色々な部分が正しい順番に出力するという感じで、自分は何をやってるのかは分かるが、説明するのは中々できないという問題があった。そのため、もう一回言語 A の授業を繰り返して、プログラムの構成を画像化にしてからその画像を説明した。これからももっとプログラムを説明できるように、ノートやプログラム内のコメントをよく使って、うまく理解できるプログラムを作成できるように頑張ります。