情報科学演習Ｄ

第４回レポート課題

提出年月日：平成 30 年 12 月 9 日
電子メール：u839996e@ecs.osaka-u.ac.jp
学籍番号　：09B16084
氏名　　：山野広大

1　課題 4 の目的

Pascal 風言語で記述されたプログラムを，アセンブラ言語 CASL II で記述されたプログラムに翻訳（変換）するコンパイラの作成。

2　プログラムの動作原理

2.1　プログラムで利用したデータの説明

ここでは本プログラムで使用したデータについての説明を行う。
課題 2、課題 3 で作成したデータはそのまま使用したためここでは課題 4 で追加したもののみ記載する。

名前	型	用途
variableList	List<Variable>	全ての変数を保存する
subproIndexList	List<Integer>	副プログラムの行数番号を保存する
outputStringList	List<String>	文字定数の出力用文字列を保存する
forDS	int	変数用のバッファの数を保存する
subprogramCount	int	副プログラムの数を保存する
callCount	int	手続き呼出し文が出現した数を保存する
whileCount	int	while 文が出現した数を保存する
ifCount	int	if 文が出現した数を保存する
elseCount	int	else が出現した数を保存する
boolCount	int	関係演算子が出現した式が出現した数を保存する
share	int	メソッド間で数値を共有するのに使用する
shareString	String	メソッド間で文字列を共有するのに使用する
subproIdx	int	現在読んでいるトークンが何個目の副プログラムかを保存する
isTrue	boolean	関係演算子が出現した式が真か偽かを保存する
isError	boolean	checker を通した後エラーがあったかどうかを保存する
notCalled	boolean	現在読んでいるトークンが呼び出し文で呼び出されたかを保存する
isSubpro	boolean	現在読んでいるトークンが副プログラム内かを保存する
subprogramList	List<String>	全ての副プログラムの出力用文字列を保存する
sb	StringBuilder	1 つの副プログラムの出力用文字列を保存する

次に Variable クラスに追加した変数の説明を行う。

名前	型	用途
small	int	変数が配列型の場合、添え字の最小値を保存する
big	int	変数が配列型の場合、添え字の最大値を保存する
num	int	変数が何番目に宣言されたかを保存する
number	int	変数が int 型の場合、代入された数値を保存する
bool	int	変数が boolean 型の場合、代入された真偽値を保存する (true=1,false=0)
string	String	変数が String 型の場合、代入された文字列を保存する
subproNum	int	変数がローカル変数、仮パラメータの場合、 どの副プログラムで宣言されたかを保存する
intArray	int[]	変数が配列型の int 型の場合、代入された数値を保存する
boolArray	int[]	変数が配列型の boolean 型の場合、代入された真偽値を保存する
stringArray	String[]	変数が配列型の String 型の場合、代入された文字列を保存する
isPara	boolean	変数が仮パラメータかどうかを保存する

2.2 プログラムの流れ

2.2.1 プログラム全体の流れ

1. 課題 3 で作成した checker プログラムを通し、同時に宣言された全変数（仮パラメータ含む）を variableList に追加する
2. checker プログラムで構文的、意味的エラーを発見した場合は最初に発見した誤りの行数番号とエラーメッセージを標準エラーに出力し、isError の値を true にする。誤りを発見しなかった場合は isError の値は false にしたままにする。※ 1, ※ 2
3. isError が false の場合のみ出力用のファイルを作成する
4. トークンを読みつつコンパイルを行い、コンパイル結果の CASL II プログラムを出力ファイルに書き出す

※ 1　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、終了する
※ 2　構文的な誤りを発見した場合は”Syntax error: line” という文字列と共に最初に発見した誤りの行数番号を標準エラーに出力する。意味的な誤りを発見した場合は”Semantic error: line” という文字列と共に最初に発見した誤りの行数番号を標準エラーに出力する。

2.2.2 コンパイラプログラムの流れ

上記プログラム全体の流れの 4 番目のコンパイル部分についてより詳細に説明を行う

1. START 文やサブルーチン用の CASL 文を出力ファイルに書き出す
2. 指導書記載の EBNF に対応する形でコンパイルを行い、pas ファイルの main 部分にあたるプログラム (ブロックの次に現れる複合文) の CASL 文を出力ファイルに書き出す
3. 副プログラムが宣言された場合はその部分の CASL 文を出力ファイルに書き出す
4. 変数が宣言された場合はその分の領域を確保する CASL 文を出力ファイルに書き出す

5. 文字定数が出現した場合はその文字列を定義する CASL 文を出力ファイルに書き出す

6. サブルーチンの CASL 文を出力ファイルに書き出す

3　プログラムの実装方針

課題 2 同様、指導書記載の EBNF に対応するメソッドを作成し、その中でトークンを 1 つずつ読み、コンパイルを行う。各メソッドには引数として subproIdx を与える。これは読んでいるトークンが副プログラム内であるか否かを判断する必要があるためだ。(副プログラム外の場合は subproIdx の値は 0 である)

3.1　副プログラムの扱い

pas プログラム内に副プログラムが存在した場合は、副プログラム内の CASL 文は pas ファイルの main 部分にあたるプログラム終了直後に出力ファイルに書き出す必要がある。そのために、副プログラム内のトークンをコンパイルしているときはコンパイル結果を出力ファイルに書き出すのではなく、sb に書き込み、その副プログラムメソッドが終了したら subprogramList に sb を追加する。そして pas ファイルの main 部分にあたるプログラムのコンパイルが終了した直後に subprogramList の中身を全て順番に出力ファイルに書き出す。

3.2　スコープ管理

副プログラム内でグローバル変数と同一名のローカル変数またはパラメータが使用された場合はスコープ管理が必須となる。そこで副プログラム内で変数が利用されたら for 文を使い、variableList 内に同一名の変数があるかをチェックする。同一名の変数が複数存在した場合はその変数の subproNum を調べ、現在読んでいる副プログラムの subproIdx と比較し、一致したものがある場合はその変数をローカル変数またはパラメータ、一致するものがない場合はグローバル変数として扱うことでスコープ管理を実装した。

3.3　レジスタの使い方

名前	用途
GR1, GR2	演算や出力時の一時保存用として使用
GR3	仮パラメータ出現時に仮パラメータの番地保存用として使用
GR4, GR5	未使用
GR6, GR7	サブルーチンで使用
GR8	スタックポインタで使用

3.4　ラベル名や変数名の付与ルール

副プログラム、ループ、分岐などのラベル名の付与方法として、私は pas ファイル内に出現した順番で付与した。具体的には while 文や if 文を読んだときにそのカウント変数を 1 増やし、その値をラベル名に使用した。その結果、副プログラム内に if 文が出現した場合など、出力ファイルを読むとラベルの順番が昇順にならないことがあるが動作には何の影響もないためそのような仕様にした。
変数名の付与方法も同様に pas ファイルで宣言された順番で付与した。つまり、変数が宣言され、VariableList に追加されるときにその変数オブジェクトの num を 1 ずつ増やし、セットする。また、配列型変数に関しては (添え字の最大値-添え字の最小値+1) の領域が必要なため、配列型変数の次に宣言された変数の num の値は、配列型変数にセットした num の値に (添え字の最大値-添え字の最小値+1) を加えた値をセットする。

4　コード生成の説明

この章ではコード生成の具体例について説明する

4.1　式に関するコード生成

私は式に関するコード生成として、式の型 3 種類についてメソッドを作成した。また、どのメソッドについても式の結果を share や shareString に格納することで式メソッド終了直後に他のメソッドで使用できるようにした。また、式のメソッドが呼び出されるときは index の値が式の先頭のトークンを指すようになっている。

4.1.1　char 型

char 型の式には演算子が出現しないため変数か文字定数かの判定が行えればよい。そのため IDList の index 番目が名前か文字定数かで分岐させ、文字定数の場合はその文字列を GR1 にロードし、PUSH する。また、shareString には文字定数からを取り除いたものを格納する。変数の場合はその変数の num を GR2 にロードし、GR1 に VAR から GR2 番目をロードし、PUSH する。また、shareString には変数の getString() を使って得た文字列を格納する。

4.1.2　int 型

int 型の式には加法、乗法演算子が出現するため、EBNF に対応するメソッドを作成した。単純式、項のメソッドでは、EBNF の初めの要素をコンパイルし終えたら、そのあとに演算子が出現するかを調べ、演算子が出現した場合、被演算子の要素をコンパイルし、その後に演算子の種類に応じて演算子をコンパイルした。int 型変数の因子では変数、定数、括弧が存在するが、変数の場合はその変数の num を GR2 にロードし、GR1 に VAR から GR2 番目をロードし、PUSH する。また、share には変数の getNumber() を使って得た数値を格納する。変数が配列型だった場合は、添え字の式をコンパイルするメソッドを呼び出し、それを GR2 に POP する。その後 GR2 に配列型変数の (num-添え字の最小値-1) の値を加える。その後 GR1 に VAR から GR2 番地の値をロードして PUSH する。定数の場合は数値を PUSH する。また、share には定数の数値を格納する。括

弧の場合は式をコンパイルするメソッドを呼び出す。また、int 型式の単純式には選択任意の符号という要素が存在する。そのため、単純式のメソッドが呼び出されたら先頭のトークンが符号に対応するメソッドかどうか評価し、トークンが符号かつ'-' の場合のみ、符号の直後の項のコンパイルが終わったらそれを POP して GR2 に格納し、GR1 に 0 を格納し、GR1-GR2 を行い、PUSH することでマイナスの処理を行っている。また、マイナスの処理を行った場合、項の share に (-1) をかけたものを share とする。

4.1.3 boolean 型

boolean 型の式には加法、乗法演算子に加え、関係演算子が出現する。そのため int 型同様 EBNF に対応するメソッドを作成した。加法、乗法演算子については基本的に int 型と同じ処理を行っているため説明は省略する。関係演算子が出現した場合、boolCount をインクリメントし、演算子の左辺を shareTemp というローカル変数に保存し、右辺のコンパイルを行う。両辺のコンパイルが終了したら、演算子の種類に対応する CASL 文を出力する。また、右辺のコンパイル結果の share と shareTemp を比較することで share の値を決定している。さらに、boolean 型の因子には not が出現する可能性がある。not が出現したら、その後ろの項をコンパイルし、その後 POP して GR1 に格納し、XOR で # FFFF の排他的論理和をとり、それを PUSH する。

4.2 変数への値の代入に関するコード生成

初めに代入文の流れを説明する。

1. 左辺の変数の型を getType() を使って調べる
2. 左辺の型に合わせて右辺の式をコンパイルするメソッドを呼び出す
3. PUSH されている右辺の式の結果を GR1 に POP する
4. 左辺の変数が宣言されている番号 (VAR からの番号) を getNum() を使って調べる
5. 調べた番号を GR2 にロードする
6. GR1 の内容を VAR から GR2 番地に保存する

このプログラムでは右辺の式の結果を左辺の変数に代入する CASL 文を出力しているが、java 上でも左辺の変数オブジェクトに右辺の値を share や shareString を使ってセットしている。しかしここで問題になるのが、コンパイルと同時に変数に値をセットすると、本来セットすべきでない時 (呼び出されていない副プログラム内や条件式の値が fals e の if 文内など) にもセットしてしまい、以降その変数が利用されるときに不適切な値を返してしまうということだ。この問題を解決するためにグローバル変数である isTrue や notCalled を使い、pas プログラム同様、代入すべき時のみ値をセットするようにしている。(値をセットすべきでない時でもコンパイルは行い、出力ファイルに書きでしている)

4.3　手続き呼び出し文に関するコード生成

手続き呼び出し文のメソッドではまず呼び出された手続きが何番目に宣言されたものか調べ、ローカル変数に保存する。次に引数があるかを調べ、あったらその型を調べ、その型に応じて式メソッドを呼び出し PUSH する。そのあと出力ファイルに CALL と、手続き宣言番号を書き込む。その後、その時点での index の値をローカル変数に保持しておき、グローバル変数である isSubpro を true、notCalled を false に変えて複合文メソッドを呼ぶ。この時、複合文メソッドの引数 subproIdx の値は手続き宣言番号である。なお、isSubpro が true、notCalled が false の場合、代入文や if 文などは実行されるが出力ファイルには書き込まれない。つまり、java 上でも呼ばれた副プログラムを実行しているようなイメージである。複合文メソッドが終了したら変更した isSubpro と notCalled と index の値を元に戻す。

5　サブルーチンについて

本プログラムでは指導書記載のサブルーチンライブラリをそのまま利用した。

6　課題 1〜3 で作成したプログラムをうまく再利用できたかどうか, およびその理由

本プログラムを作成するにあたって過去の課題は、考え方や方針を立てるという点では大いに利用できたといえるが、実際にプログラムとして再利用できたかといえばあまりうまく再利用できなかったように感じる。だが、再利用できた部分も存在する。例えば、課題 2 で作成した式をチェックするメソッドは、チェックした式の型を返すように作成した。そのメソッドを本プログラムから呼び出して使用した箇所がいくつかある。また、課題 3 のプログラムをほとんどそのまま本課題で用い、構文的または意味的エラーを発見した場合に isError の値を書き換えることで、エラーがあった場合に出力ファイルを作成することを防いだ。しかし、いま述べた部分以外のコンパイラプログラムは全て新しく作成したものなのでもう少しうまく再利用できた部分があったかもしれない。

7　まとめ

課題 4 では Pascal 風言語で記述されたプログラムを、アセンブラ言語 CASL II で記述されたプログラムに翻訳するコンパイラを作成した。作成したプログラムは入力ファイルのトークンを読みながらコンパイルをしていく。また、java 上でも入力ファイルのプログラムの動作と同じ動作を行っているため、条件分岐や副プログラムの扱い、手続き呼び出しなどがやや煩雑になったが、一貫した方針でプログラムを作成することができた。

8　感想

課題の初めのうちは答えの cas ファイルと一致させるようにプログラムを作成していた。そのためトークンを解析した直後に出力ファイルに書き出すようにしていたが、副プログラムが出現して、その方法ではうまくいかなくなった。そのためトークンを解析したら、コンパイルした CASL

文をどこかに格納しておき、副プログラムでなかったら出力ファイルに書き出すなどの工夫をすればより良いプログラムになったと考える。
また、式に関するメソッドを各型ごとに作成したが、すべての型を解析できるメソッドを 1 つ作成できればよりよかった。
入力ファイルのプログラムと同じ動作を java 上でもしているためデバッグがやりやすかったことは非常に良かった。しかし、逆に代入文や副プログラムの扱いがやや難しかった。