情報科学演習 D
課題 4

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: 09B20033

城間大幹

2023 年 1 月 26 日

1　仕様

コンパイラへの入力は Pascal 風プログラムを字句解析した結果 (ts ファイル) であり，出力は CASL II で記述したプログラム (cas ファイル) である．構文的もしくは意味的な誤りを検出した場合は，標準エラーにその誤りの内容を出力して終了する．エラーメッセージの仕様は課題 3 に準じる. 開発対象となるコンパイラ のメソッドは Compiler.run(String, String) である. 当該メソッドの仕様は以下の通り．

- 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する．
- コンパイル結果の CASL II プログラムは第二引数で指定された cas ファイルに書き出す．
- 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．
  (エラーメッセージの内容は Checker.run() のエラー出力に準じる)
- 構文的な誤りが含まれる場合もエラーメッセージを表示する．
- 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．

なお，誤りを発見した場合には cas ファイルを生成してはならない．

2　方針と設計

課題 3 までのプログラムに CASL の文を生成するメソッドを追加し, それらを適切な位置で呼び出すことにより実装した．CASL 文は Stringbuilder に append し続ける形で生成した．

3　実装プログラム

3.1　追加したメソッド

ここでは課題 4 で新たに追加したメソッドについて, 説明する

3.1.1　arrayPointer

配列 arraypointstack に引数の arraypoint の内容を追加したり, 逆に配列 arraypointstack から値を取り出したりするなどして, 配列の添字を扱うためのメソッドである．引数 mode で push か pop を指定することで, 両者を切り分けて実行できる．

3.1.2　arrayPointerPush

変数リストにある配列を参照するために用いるメソッドである．ある配列要素を参照するには, 配列の先頭要素の番地 + インデックスを番地として指定しなければならない．したがって, 配列の先頭要素の番地を探すのがこのメソッドの役割である．

3.1.3　whichProcedureCall

CASL で呼び出す副プログラムの番号を指定するメソッドである．CALL PROC に続く番号となる．

3.1.4 variableSameName

変数リスト variable に scope に関わらず, 同一の名前を持つ変数が何個あるかをカウントするメソッドである.

3.1.5 push0ForMinus

CASL で負の数を扱うためのメソッドである. 例えば, -2 は 0-2 の演算結果の数であることから, 本プログラムでは 0 と 2 を PUSH して SUBA することにより, -2 を実装している. 負の数を扱うために必要な 0 を PUSH する.

3.1.6 addAndRenewOperandStack

被演算子のスタックへの格納や, 演算を行った値をスタックへ再度格納するメソッドである.

3.1.7 stringbuilderForOperatortype

演算の種類を示す変数 operatortype に応じて ADD, SUB, MULT, DIV, MOD の CASL 文を実装するメソッドである.

3.1.8 selectComparisonOperator

比較演算子を示す変数 comparisonOperator に応じて if 文, while 文の条件式の CASL 文を実装するメソッドである.

3.1.9 selectLogicalOperator

AND と OR 演算の CASL 文を実装するメソッドである.

3.1.10 notOperator

対象の数と#FFFF を XOR 演算することにより, NOT の CASL 文を実装するメソッドである.

3.1.11 conditionalFinish

AND や OR により入れ子や複数になった条件式の最後の判定に用いるメソッドである. #FFFF となれば ELSE を, そうでなければ TRUE の文を実行する.

3.1.12 changeArgument

副プログラムを呼び出す前に, 引数の値を変数領域に格納するためのメソッドである.

3.1.13 selectStringbuilder

メインプロセス時は sb に, 副プログラム実行中は sbForProc に sbForScope の内容を加えるメソッドである.

3.1.14 selectsbForScopeORsbsub

代入式の右辺や条件式などでは sbForScope を, 代入式の右辺などでは sbsub を用いることを決めるメソッドである.

3.1.15 inputsbsubTosbForScope

主に代入式の左辺で用いる, sbsub を sbForScope に加えるメソッドである.

3.1.16 inputcomparisonOperator

比較演算子それぞれに応じた数を変数 comparisonOperator に格納するメソッドである. comparisonOperator は if 文や while 文の条件式で用いるグローバル変数である.

3.2 CASL への変換

3.2.1 定数, 変数, 配列の参照方法

基本的にはそれぞれの値をスタックに PUSH し, その後の状況に応じて ST や LD 命令を実行するため, ここでは参照してスタックに PUSH するための方法について述べる.

まず定数は値をそのままスタックに PUSH するだけで良い.

次に配列でない変数は, メソッド addAndRenewOperandStack で variable を探索し, その変数が変数領域 VAR の何番目であるかを取得し GR2 に格納する. そして VAR と GR2 のアドレスを元にその変数の中身を GR1 に格納し, スタックに PUSH する.

最後に配列については, まず添字の値を GR2 に格納する. 添字には定数, 変数, 式といったパターンが考えられるが, いずれの場合であれ添字の値を求めその内容を GR2 に格納すれば良い. 式の扱い方については 3.2.3 に記載した. 次に配列の先頭要素のアドレスをメソッド arrayPointer で取得し, その値と GR2 を加算した値を再度 GR2 に代入する. そして VAR と GR2 のアドレスを元にその変数の中身を GR1 に格納する.

3.2.2 変数の代入方法

CASL の代入文は, 右辺の値をスタックに PUSH した後, 左辺の変数のアドレス箇所を割り出した後に POP で取り出し, ST 命令で格納するのが一連の流れである. したがって, 使用する Stringbuilder は主プログラムでは sb と sbsub, 副プログラムでは sbForProc と sbsub を用意し, 左辺は sbsub に, 右辺は sb, sbForProc に格納し, 代入文が終了した後に, メソッド inputsbsubTosbForScope により sb, sbForProc に sbsub を append することで実装した.

sbsub を使用するかの判定は, メソッド selectsbForScopeORsbsub により行った. selectsbForScopeORsbsub では, 代入文の右辺を示す sassignFlag, 副プログラム呼び出し文を示す procFlag, if 文, while 文の条件式を示す conditionalFlag のいずれかが 1 の場合は sbForScope を, そうでない場合は sbsub を選択する. なお sb と sbForProc の選択については, 3.2.8 章で述べる.

3.2.3 式の処理方法

演算の優先順位を考慮するために, 算術演算子を示す formerOperatortype と operatortype の値を元に, 演算の CASL 文を生成するメソッド addAndRenewOperandStack を実行した. formerOperatortype は一つ前の演算子, operatortype は formerOperatortype の次の演算子を示す.

まず二つ目の演算子が現れるまで, 被演算子をスタックに PUSH, 演算子を formerOperatortype に格納する. 二つ目の演算子が加算または減算であった場合は, formerOperatortype を引数に addAndRenewOperandStack を実行し, 演算子二つを POP したのちに, 演算結果をスタックに PUSH する. 一方二つ目の演算子が乗算, 除算であった場合は, 次に現れる被演算子をスタックに PUSH した後に, operatortype を引数に addAndRenew-OperandStack を実行し, 演算結果をスタックに PUSH する.

また括弧を含む演算については, 括弧内の処理を行うメソッド bracket でメソッド calculation を呼び出していること, 演算子を示す formerOperatortype と operatortype がローカル変数であることから, 括弧が含む演算も前述と同様に行えば良い.

このようにして演算の優先順位を考慮した計算を可能にした.

なお論理演算子については, 演算子を Operatorlist に格納し, 2 項そろったタイミングでリストから呼び出すことで実装した. 括弧の処理については算術演算子と同様である.

3.2.4 副プログラムの呼び出し方法

副プログラムの処理は procFlag を 1 とした時に実行する.

本プログラムでは, 課題 3 以前に作成した変数リスト variable を再利用するために, 主プログラム, 副プログラムに関わらず登場する変数全てに変数領域を割り当てている. したがって引数があれば, その内容を PUSH した後, それぞれの引数に割り当てた領域を探して ST で格納し, 更新する. そしてメソッド whichProcedureCall により副プログラムの場所を探し, CALL する.

3.2.5 分岐の処理方法

分岐先のラベルについては 3.2.7 章で説明する,

- 副プログラムの分岐
  3.2.4, 3.2.7 に記載した通りである.
- while 文と if 文の分岐
  メソッド selectComparisonOperator による各条件式に応じた分岐命令からなる. 比較演算子に対し左辺を GR1, 右辺を GR2 として CPA 命令を実行することから,
  - SEQUAL(=) の結果が真の時は JZE
  - SNOTEQUAL(<>) の結果が真の時は JNZ
  - SLESS(<) の結果が真の時は JMI
  - SLESSEQUAL(<=) の結果が真の時は JMI と JZE
  - SGREATEQUAL(>=) の結果が真の時は JPL と JZE
  - SGREAT(>) の結果が真の時は JPL

  を実行して WTRUE または ITRUE に飛び, 比較の結果が負の時は JUMP を実行して ENDLP または ELSE に飛ぶ.

3.2.6 レジスタやメモリの利用方法

使用したレジスタは GR1, GR2 のだけであり主に GR2 が変数などのアドレスを示し, GR 1 が変数の中身を示すことが多い. 続いてメモリについては, 変数リスト variable に格納された分だけ変数領域 VAR を確保し, write で使用するシングルクォーテーションマークで囲まれた文字列は CHAR として確保し, 使用した.

3.2.7 ラベルの管理方法

今回使用したラベルの名称と, 添字の管理方法について示す.

- 副プログラム文を示す PROC
  procCounter により副プログラムの数をカウントし, PROC の添字にする. なお副プログラムを呼び出す際は, 変数リスト variable から名称を探索し, 何番目に登場したかを返り値として示すメソッド whichProcedureCall を用いてラベルを指定する.

- while 文の条件式を示す LOOP

whileCounter により while 文の数をカウントする．while 文が入れ子になる場合などに添字がずれないよう, メソッド SWHILE 実行時の whileCounter の値をローカル変数の initwhile に保存し, LOOP の添字とする．

- while 文中のループ処理を示す WTRUE
  while 文が入れ子になる場合などに添字がずれないよう, メソッド SWHILE 実行時の whileCounter の値をローカル変数の wtrue に保存し, WTRUE の添字とする．

- while 文の終了を示す ENDLP
  while 文が入れ子になる場合などに添字がずれないよう, メソッド SWHILE 実行時の whileCounter の値をローカル変数の endwhile に保存し, ENDLP の添字とする．

- if 文の条件式が真の時を示す ITRUE
  ifCounter により if 文の数をカウントする．if 文が入れ子になる場合などに添字がずれないよう, メソッド SIF 実行時の ifCounter の値をローカル変数の itrueif に保存し, ITRUE の添字とする．

- if 文の条件式が偽の時を示す ELSE
  if 文が入れ子になる場合などに添字がずれないよう, メソッド SIF 実行時の ifCounter の値をローカル変数の elseif に保存し, ELSE の添字とする．

- if 文の終了を示す ENDIF
  if 文が入れ子になる場合などに添字がずれないよう, メソッド SIF 実行時の ifCounter の値をローカル変数の endif に保存し, ENDIF の添字とする．

- 条件式が複数項ある際, ある一つの項が真の時を示す TRUE
  条件式の項数を数える andCounter を TRUE の添字とする．

- 条件式が複数項ある際, ある一つの項が偽の時を示す FALSE
  条件式の項数を数える andCounter を FALSE の添字とする．

- 条件式が複数項ある際, 最終的な真偽の値を示す INPUT
  条件式の項数を数える andCounter を INPUT の添字とする．

3.2.8 スコープの管理方法

本プログラムでは scope という変数によりスコープを管理している．副プログラムを定義する begin から end の間は scope は副プログラムの名前とし, それ以外は global という名称にしている．CASL 文については, 副プログラムの内容は sbForProc に, 主プログラムの内容は sb に格納し, Pascal 文終了時 sbForProc を sb に結合する．それぞれの scope に応じて sb と sbForScope を選択する役割を, メソッド selectStringbuilder が担っている．

3.2.9 過去の課題プログラムの再利用

課題 3 以前のコードの 9 割を変更せずに実装した．新たなメソッドを追加したことにより行間が長くなったり, リファクタリングにより不要なコードが削除されたりしたがベースはそのままである．

4　工夫点

工夫点は二つある．まず一つ目は, CASL 文を記述する工程では, 多くの場合 Stringbuilder に append するたびに sb, sbForScope, sbsub のいずれかを判断する必要があり, 必然的にコードが長くなってしまう．そこで sbForScope を使用するメソッド selectStringbuilder と sbTemp を使用するメソッド selectsbForScopeORsbsub により, 記述量を削減した点である．

続いて二つ目は, 引数がある副プログラムを呼び出す CASL 文を簡略化したことである．言語処理工学の資料などサンプルの CASL 文では, スタックの戻り番地の前に引数の内容を PUSH して副プログラムを実行し, その後引数を GR8 などを用いて副プログラム内のローカル変数に格納している．しかし, 変数 variable リストを用いた探索により引数の変数領域でのアドレスを特定し, CALL 命令前に内容を ST で格納することで, 副プログラム実行後はすぐに begin 以下の文から始めることができる．以下に normal10 の CASL 文を比較することで, 具体的に示す. 配布された CASL 文では,

```
PUSH 99 ; proc-call const-uint (99)
CALL PROC0 ; proc-call

PROC0 NOP ; proc
LD GR1, GR8 ; proc local-var
ADDA GR1, =1 ; proc local-var
LD GR2, 0, GR1 ; proc fparam (x)
LD GR3, =2 ; proc fparam (x)
ST GR2, VAR, GR3 ; proc fparam (x)

SUBA GR1, =1 ; proc fparam (x)
LD GR1, 0, GR8 ; proc fparam
ADDA GR8, =1 ; proc fparam
ST GR1, 0, GR8 ; proc fparam
; L8 assign ; proc assign
LD GR1, ='s'; proc assign const-str ('s')
```

となっていたが,

```
PUSH 99
LD GR2,  =2;LD4
POP GR1
ST GR1,  VAR,  GR2
CALL PROC0

PROC0 NOP
LD GR1,  ='s'
```

上記のように, 副プログラム内の begin 以下を実行するまでの CASL 文を短縮できた.

5　感想

コードを書き始めるまでの準備と定期的にリファクタリングすることの重要性を痛感した．まずコードを書き始めるまでの準備とは, そもそもどういうアルゴリズムで問題を解くか, どういった設計・構造にするか, CASL 文はどのような文を書くかなどを決める, 大枠を定めるということである．当たり前な事ではあるが, 開発過程の中で明確にそうした時間を確保し, そしてそこで決まった内容を明確に output するまで行う必要がある．なぜなら方針ややるべきことが明確でないと, 開発が進んでいく過程で迷走してしまうからである．コードを書くという作業につい逃げてしまうが, 開発全体の効率や後々の実装のことを考慮し, コードを書き始めるまでの準備を決して軽視しないようにしたい．

また定期的にリファクタリングすることにより, コードの行数が減ることや見栄えが良くなるだけでなく, コードを何度も見返すことにより理解が深まることや, 最適化したコードであればデバッグが楽になり, 開発が効率よく進むことを感じた．今回の開発はこれまでの開発方法に倣い, 要件を全て満たすプログラムを作成した後, リファクタリングする流れで進めたが, この方法では手当たり次第にコードを書くことになるため必然的にコードが長くなってしまい, 複雑で難易度が高い今回のプログラムだと一時は 3,500 行ほどのコードになってしまった．そのためコードの概要理解が追いつかない, 単純に忘れてしまうといった理由から重複する箇所やメソッドが多々出たり, 一つのエラーに対する原因特定の時間が日々増えたりするという問題が生じた．開発の後半ではリファクタリングと並行して行うことで 1000 行以上のコードの削減し, 同時に開発効率も大幅に改善させた．開発初期から行なっていれば更なるコードの最適化, 開発期間の短縮が実現できたと思う．今後は機能追加とリファクタリングの両輪で, 開発を初期から進めていきたい．

6　謝辞

終始熱心なご指導を頂いた情報科学演習 D の松本 真佑，桝井 晃基担当教員, また TA の方々には心より感謝の意を表します．