情報科学演習Ｄ　課題3レポート

担当教員：松本　真佑
提出者：市川　達大
学籍番号：09B17010
メールアドレス：u243952e@ecs.osaka-u.ac.jp

提出年月日：2021 年 2 月 12 日

1　課題の目的

コンパイラを作成する第 3 段階で、第 1 回の字句解析をもとにした、第 2 回の構文解析につづき「意味解析」を行う。

2　課題の達成の方針と設計

2.1　外部仕様

課題 1 で字句解析した結果ファイル (ts ファイル) を読み込み、意味的に誤りがあるかどうかを判定する。正しい場合は標準出力に”OK”を出力し、構文が正しくない場合は”Sematic error: line”という文字列とともに最初の誤り部分の行番号を出力する。

2.2　方針と設計

以上の課題を達成するために主に以下の方針で設計し実装する。今回の課題で検出する意味的な誤りとして主に以下 3 つの課題がある。

1. 変数の重複宣言の判定
2. 参照された変数が宣言されているかどうかの判定
3. 添字の型が integer であるかどうかの判定
4. 代入分の左辺に配列型の変数名が指定されている.
5. if 文や while 文の式の型が boolean でない.
6. 演算子と被演算子の間で型の不整合が発生している.
7. 代入文の左辺と右辺の式の間で型の不整合が発生している.
8. 手続き名が，副プログラム宣言において宣言されていない.

上記の問題を解決するために、宣言された変数の状態、有効範囲を管理する記号表を実装することで今回の課題を達成する。今回は C 言語でいう構造体を [LinkedList.java] というクラスで実現する。LinkedList クラスには以下のフィールドを保持する。

```
1        String name; //名前
2        String name_type; //種類
3        String data_type; //データ型
4        int array_min,array_max; //配列の最小値、最大値
5        Variable_table next;//次のノードへの参照
```

この構造体データを線形リストとして保存することで 1 つの記号表を表現する。さらに、この構造体クラスを以下のように宣言し、複数個インスタンス化することで変数の有効範囲を管理する。また、最大ブロック数は 100 とする。

```
1  LinkedList[] code_block = new LinkedList[100];//記号表
       最大 100個程度のブロック数のコードであると想定する
```

2.2.1 記号表の基本方針

次にどのように記号表を実装していくかを説明する。記号表は変数宣言がある時、インスタンス化した構造体クラスに先ほど記した「名前」「種類」「データ型」配列の場合「添字の最大値」と「添字の最小値」を登録して行けば良い。この時、同ブロック内を同じ有効範囲とし、ブロックが変われば新たな別の記号表し、記号表へ情報を格納していき、そのブロックから抜ければ記号表を捨てることで変数の有効範囲を実現する。具体的には構造体クラス [LinkedList.java] 内の addNode メソッドで登録していく。

```
1  //ノードの追加
2          public void addNode(String name, String name_type, String data_type, int num_var) {
3                  Variable_table node = new Variable_table();
4                  node.name= name;
5                  node.name_type = name_type;
6                  node.data_type = data_type;
7                  node.num_of_var = num_var;
8                  node.next = first;
9                  first = node;
10         }
```

2.2.2 変数の重複宣言の判定

変数の重複宣言の判定は、記号表に変数や副プログラム名を登録する際に、同ブロック内に同じ名前の変数があるかどうかを判定し、あればエラーを出すことで判定できる。

2.2.3 参照された変数が宣言されているかどうかの判定

参照された変数が宣言されているかどうかの判定は、代入分や式にて変数が参照された際、参照された変数が、記号表の現在有効な範囲に登録されているかどうかを判定し、なければエラーを出力することで実現する。

2.2.4 添字の型が integer であるかどうかの判定

添字の型が integer であるかどうかの判定は、配列の添字が参照された際、その変数や定数が integer 型であるかどうかを判定する。定数の場合それが int であるかどうかを判定し、変数の場合はまずその変数が現在有効な範囲の記号表に登録されているかどうかを判定し、されていれば記号表に登録されている「型」を参照することでその変数が integer であるかどうかを判定する。添字が変数や定数でなく何かしらの演算を含む「式」である場合は演算子が「+,-,*,/」のいずれか、かつ非演算子が integer 型であるかどうかを判定することで実現する。

2.2.5 代入分の左辺に配列型の変数名が指定されている

代入分の左辺に配列型の変数名が指定されているかの判定は、添字を持たない変数が参照された時、記号表からその変数の「種類」が array だった時、配列が添字なしで参照されたとしてエラーを出力することで実現する。

2.2.6 if 文や while 文の式の型が boolean でない.

if 文や while 文の式の型が boolean でないかどうかの判定は、現在有効な記号表への参照をもとに、式の非演算子が boolean でなく、かつ関係演算子も持たない場合式が boolean 型でないとしてエラーを出力することで実現する。

2.2.7 演算子と被演算子の間で型の不整合が発生している.

記号表の「型」をもとに、非演算子どうしが常に同じ型であるかを判定し、違う型の場合エラーを出力することで実現する。

2.2.8 代入文の左辺と右辺の式の間で型の不整合が発生している

代入文の左辺と右辺の式の間で型の不整合が発生しているかどうかの判定は、記号表をもとに式を評価した後の「型」が左辺と右辺で一致してるかを判定し、一致していなかった時エラーを出力する。式の評価は演算子と被演算子の間で型の整合性が取れていることを前提に、関係演算子があれば「boolean」それ以外は非演算子の方に一致するので、これをもとに判定できる。

3 プログラムの実装方法

上記の設計方針をもとに具体的にプログラムを実装していく。

3.1 記号表の作成

記号表の作成は「変数宣言」の BNF メソッド- `is_variable_dec` メソッド内で行い、予めインスタンス化しておいた構造体を連結リストでつなげたものに登録していく。ブロックが変わった時、別の連結リストへ登録することで変数のスコープを表現する。具体的流れとして、まず変数が重複していないかを調べ、重複していない場合 上記した「LinekedList.java」内の addnode メソッドにて記号表へ登録していく。

3.2 変数が存在するかの探索

変数か存在するかは、現在有効な記号表を全て調べ、参照されている変数名があるかどうかを調べる。有効な記号表であるかは、記号表の状態を管理する 1 次元配列 $code_block_state$ にて管理する。

```
1  //変数が存在するか全てのブロックで調べる
2          public boolean is_block_has_var(int index) {
3                  //コードブロック全てを調べる
4                  for(int i= code_block_num; i >= 0; i--) {
5
6                          if(code_block_state[i] == 1 && code_block[i].search(get_name_list.get(
                               index))) {
7                              variable_place = i;
8                              return true;
9                          }
10                 }
11                 return false;
12         }
```

また、記号表に登録されている情報は「LinkedList.java」内の「var_search」メソッドにて取得する。

```
LinkedList.java

        //検索した型名の情報をとってくる
        public Variable_table var_search(String search_name) {
                Variable_table current = first;
                while(current != null) {
                        if(current.name.equals(search_name)){
                                return current;
                        }
                        current = current.next;
                }
                return current;

        }
```

3.3 式の評価

この課題において、様々な意味的な誤りを検出する必要があるが、「変数の重複宣言の判定」や「参照された変数が宣言されているかどうかの判定」や「手続き名が，副プログラム宣言において宣言されていない」は記号表を見て、参照されている変数が存在するかどうかを判定すればいいだけなので比較的簡単であるが、残りの意味的な誤りを判定するには「式」を評価する必要があり、この課題ではこの「式の評価」に難しさを感じた。というのも「式」では様々な演算子や「因子」が呼び出されプログラムの動作を追いかけにくく、同時に常に演算子と非演算子の整合性を気にしながら「i+2*(-j) +g < g * i and i <= 10」みたいな式を最終的に boolean 形であると評価する必要がある。今回のプログラムでは「常に演算子と非演算子の整合性を取りながら、関係演算子がくれば boolean 型になる」と判定することで、式の評価を実現した。また、演算子と非演算子どうしの整合性は非演算子の方を管理するグローバル変数で管理することでメソッド間の受け渡しも実現した。グローバル変数を使うと予期しないバグが起こりそうであまり使いたくなかったが、仮引数を用意して受け渡しをするのもナンセンスかと思ったので今回はグローバル変数を用いた。

4 まとめ

今回の課題 3 では、記号表などのデータ構造の管理方法やグローバル変数の危うさを学んだ。複数のメソッドを往復する状態では変数の管理を工夫する必要がある。

5 感想

今回の課題 3 では、式の評価にだいぶ苦戦した。前後の状態を管理しながらも、複数のメソッドに跨ってプログラムが動作するので、非常にわかりずらかった。また、演算子と非演算子の状態の管理をグローバル変数で行ったので、因子内で [(式)] が呼び出された時などに予期しないエラーが起こったので、グローバル変数を一度ローカル変数に格納し直す必要があったので非常に難しく感じた。