システムソフトウェア特論'19 # 10 意味解析と記号表

久野 靖 (電気通信大学)

2019.1.9

意味解析の位置付け

プログラミング言語の定義は構文と意味の組み合わせによりなされるという説明をしてきました。それらのうち構文については、ここまでに文脈自由文法による形式的 (formal) な定義や、それを認識する解析器を扱って来ました。これはつまり、理論的な背景があり、それに基づいた厳密なやり方で扱う方法が確立している (やり方が分かっている)、ということです。

しかし逆にいえば、プログラミング言語という複雑な対象のうちで、理論的な扱いやそれに対応するツールがまだ確立していない「その他」の部分が「意味」として残されているとも取れます。そしてそれを引き受けるのが意味解析 (semantic analysis) です (図 1)。

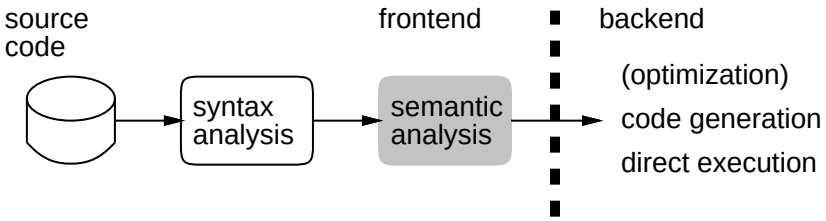

図 1: 意味解析の位置づけ

ただし、意味解析でも扱わない部分があります。それは「それぞれのコード記述がどのように動作するか」という部分です。言語の動作の実現は、インタプリタ (解釈系) であれば実行部分、コンパイラ (翻訳系) であれば出力されるコードによって定められますから、意味解析の中では扱わなくても済みます (検査や最適化などを目的に扱うこともあります)。

意味解析の位置付け (2)

意味解析は解析部 (フロントエンド) の一部ですから、プログラムコードに記述された情報を抽出して解析することが仕事であり、それら情報のうちで、構文では扱えない部分を取り扱います。具体的には、次のような事項です。

- ソースコード中で使われている名前の情報を抽出する
- それぞれの名前に付随すべき情報を整理し整合性を検査
- 目的コード生成で使用される情報を準備

具体的にはどういうことでしょうか。たとえばC言語をはじめ多くの強い型の言語 (型検査を行なう言語) では、変数は宣言してからでなければ使用できず、また使用するときに宣言と整合していない使い型をするとエラーになります。この「使用できない」「エラーになる」というのはすべてコンパイラが実現すべきことで、これが意味解析の仕事に含まれます。

前回までに扱ってきた「小さな言語」では意味解析がなく、構文解析が終わったら直ちに実行に入っていました。それは、「変数は宣言不要で、使ったら勝手に (その時点で) 用意される」「変数はすべて整数型」という設計になっていたからです。このような設計で、なおかつインタプリタであれば、実行を開始して最初に変数を読み書きした時にその場所を (実際にはハッシュ表の1エントリとしてですが) 用意すれば十分でした。

意味解析の位置付け (3)

しかし多くの言語ではもっと「きちんと」変数を扱う必要がありますし、変数以外に手続きの名前や型の名前もあります。手続きを呼ぶときには、その引数の個数や型が手続き定義と整合している必要もあります。そのような言語では、意味解析の仕事はそれなりに複雑です。[1]

以下ではまずソースコードに現れる名前の情報を収集する手段である記号表について、続いて強い型の言語でとくに重要になる型検査について取り上げていきます。

[1]そして、上記の 3 番目に「準備」とありましたが、変数であれば「どの場所にする」ということを決めて置かないと目的コードが生成できないので、それも担当します。

記号表

記号表に登録される情報

上で挙げたように、意味解析の仕事の多くは、変数名、手続き名などの「名前」に関することです。このため意味解析では、これらの名前の情報を記録した「表」を保持し、そこに情報を登録したり、その情報を参照してチェックするなどの形で多くの仕事をします。この表のことを伝統的に記号表 (**symbol table**) と呼んでいます。

記号表に登録される名前の情報としては、次のようなものがあります (もちろん言語によって違ってきます)。

- 名前の文字列 — 名前の文字列は生成コードに現れることもありますが、生成コードでは番地などの数値に変換されて消えるため、メッセージの表示にしか名前を使わない処理系もあります。
- 名前の種別 — 変数名、定数名、手続き名、モジュール名、型名、レコードのフィールド名、構造体のタグなど、言語により多様な種別の名前があります。
- 名前のスコープ — 変数でいえば、同じ変数でもローカル変数、グローバル変数、手続きのパラメタ、モジュール内の変数など、様々なスコープのものがあります。他の名前でも (変数とまったく同じではないにせよ) 同様です。
- 型情報 — 強い型の言語では、変数、定数、手続き (返値)、フィールドなど多くの名前に型 (**type**) が付随しています。この情報は型検査のために重要ですし、コード生成でも使われます。
- その他の属性 — 定数の値、変数の配置された位置、フィールドの位置など、名前ごとに固有のさまざまな属性があります。

記号表に登録される情報 (2)

これらの各種情報はどこから来るのでしょうか。名前の文字列は字句解析から来ます。また、名前の種別は「手続き定義の冒頭に現れた名前なら手続き名」「変数定義の箇所に現れた名前なら変数名」のように、多くは構文と対応しています。

そして、その後のスコープ、型情報、その他の属性については、記号表を解した処理、つまり意味解析の処理を通じて決まっていくことになります。具体例についてはこの後で取り上げます。

ブロックスコープとその実現

記号表も表なので、線形探索による表、2分探索木、ハッシュ表などさまざまな技法で実現できますが、それとは別にプログラミング言語を扱う上での考慮点が多くあります。ここではその中でも特徴的な、ブロック型スコープの実現について取り上げます。

ブロック型スコープとは、多くの言語に採用されている設計であり、1 つの名前をブロックの内側と外側で別の用途に使うことを許す設計です。次のC言語コードの例を見てください。

```
char x[100];
int test(int x) {
  ... /* (1) */
  while(...) {
    ... /* (2) */
    double x;
    ... /* (3) */
    if(...) { float x; ... /* (4) */ }
    else    { int x[20]; ... /* (5) */ }
    ... /* (6) */
  }
  ... /* (7) */
}
```

ここで、外側のグローバル変数 x は文字の配列ですが、関数 test では同名のパラメタ x が定義されています。このため、この関数内では外側の x は使えません。(1) の箇所で x という名前を使うとパラメタの x になります。

ブロックスコープとその実現 (2)

さて、**while** 文の本体はブロックになっていますが、その途中に `double` 型の変数の宣言があります (今日の **C** 言語では宣言はブロックの途中でも構いません)。ただし、**C** 言語ではその変数のスコープは宣言から後ろなので、(2) の箇所では `x` はパラメタの方を意味し、(3) の箇所では `double` の方を意味します。なお、言語によっては、宣言が後に出て来ても、ブロックの中で宣言があればそれを参照することになっているものもあり、その場合は (2) も `double` の方になります。

そして、**if** 文の中ではさらに別の `x` が宣言されていて、(4) ではこちらを指しますが、**else** の枝では配列の `x` が宣言されていてこちらを指します。もしこの宣言が無ければ、(5) では `double` の `x` を参照します。内側のスコープが閉じた (6)、(7) ではそれぞれ `double` のもの、パラメタのものというふうに参照先が戻ります。

このような意味に対応する記号表はどのように実現すればいいでしょうか。それは実は難しくありません。線形探索で表を実現するとして、図 **2** のように考えます。

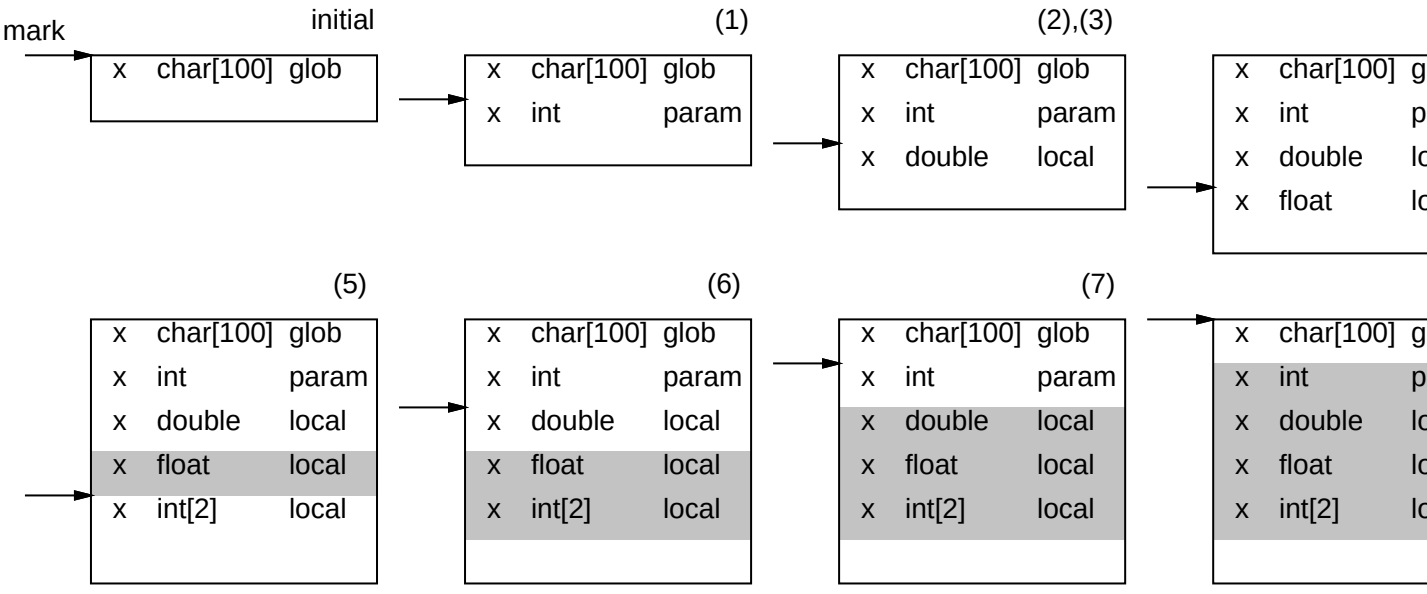

図 2: 線形探索で実現した記号表

ブロックスコープとその実現 (3)

表にはマーク (矢印) がつけてあり、そこが現在の最内側スコープのはじまりを意味します。最初はグローバルなスコープで、マークは先頭にあります。次に関数 `test` に入ると、その関数のトップレベルのスコープに入り、そこでパラメタを登録します。このほか、トップレベルで (つまり (1) や (7) で) 変数定義があったら、表の末尾に追加します。

検索については、表の末尾から上に向かって逆向きに検索します。こうすることによって、一番新しい (近くの) 宣言が見つかるので、(1) や (7) で `x` を参照するとパラメタが見つかります。マークは何のためにあるかというと、新しい変数定義を追加するときに、マークよりも下に同名の変数が定義されていたら、2 重定義としてエラーにするためです。マークより上は別の (外の) スコープなので、同じ名前でも内側スコープの別の変数になり、追加できます。

while のブロックに入った (2) や (3) のところでは、マークがパラメタより下に来ていて、同名の変数が定義できます。もちろん (2) のところではまだ `double x` に遭遇していないので、(3) まできたときにこの項目は入ります。同様にして **if** のブロックでもマークが進みます。

ブロックスコープとその実現 (4)

さて、次はスコープが閉じるところです。**if** の **then** 側のスコープが終わり、**else** で新しいスコープが始まると、マークは進みますが、同時にスコープが閉じたところでそのスコープ内の変数 (マークがあったところから現在位置まで) は探せない状態にします (図では網掛けしています)。閉じたスコープ内のローカル変数は以後参照できないわけですから。

同様に、**else** が閉じたところ、**while** が閉じたところ、関数が閉じたところでもそれぞれのスコープの変数を探せなくします。同時に、マークは「そのスコープに入る前の」位置に戻します。スコープは入れ子構造になっていますから、マークを保存しておいて戻すのにはスタック (**Last-In, First-Out** の記憶領域) を使うことができます。

例題: ブロックスコープの記号表

では、前節で述べた方式の記号表を作ってみます。まず最初に、エラーメッセージを出力するクラスを簡単に用意しておきます。メッセージを出してエラーをカウントし、後でカウントした数を取得できるようにしています。これは、エラーがあったら実行に進まないなどの動作を実現するためです。

```
public class Log {
  public static int err = 0;
  public static void pError(String s) { System.out.println(s)
  public static int getError() { return err; }
}
```

では次に記号表のクラス Symtab を見てみます。後の例題で同じクラスをそのまま使う都合上、今回の例題で不要な機能も少しくっついています。まず、型を扱うため、ITYPE、ATYPE という定数を定義しました。未定義のものを表すのは UNDEF です。

次に表の項目は Symtab.Ent という内側クラスのインスタンスで表します。その部分を先に見て頂きたいですが、1 つの項目の情報としては name(名前文字列)、alive(網掛けの場合は false)、type(上記の型情報)、pos(表の何番目の項目かを示す) があります。

記号表のインスタンス変数ですが、Ent の並びが表本体、あと前節で述べたマーク用の変数 mark と、マークを対比回復するためのスタック stk があります。

以下メソッドですが、duplCheck() は指定した変数を定義すると 2 重定義かどうかのチェックで、マークより下に網掛けでない同名の名前が定義済みなら 2 重定義です。

例題: ブロックスコープの記号表 (2)

addDef() は変数を定義しますが、まず 2 重定義ならエラーを表示した後、適当なエラー項目を返します。そうでない場合は与えられた名前で指定された型の項目を作って表に追加し、その項目を返します。

lookup は名前を指定して項目を探しますが、前に述べた理由で末尾から上に向かって探します。見つかればその項目を返し、見つからない場合は適当なエラー項目を返します。

enterScope() はスコープに入るところですが、マークをスタックに保存して表の末尾位置を新たなマーク位置とします。exitScope() はその逆ですが、ただしマークを戻す前にこれまでのマーク位置から末尾までを網掛けにします。

あとは getGsize() が表のサイズを返し、show() が表の内容を一覧表示するものです。

例題: ブロックスコープの記号表 (3)

```
import java.util.*;

public class Symtab {
  public static final int ITYPE = 1, ATYPE = 2, UNDEF = -1;
  List<Ent> tbl = new ArrayList<Ent>();
  Stack<Integer> stk = new Stack<Integer>();
  int mark;

  private boolean duplCheck(String n) {
    for(int i = mark; i < tbl.size(); ++i) {
      Ent e = tbl.get(i);
      if(e.alive && e.name.equals(n)) { return true; }
    }
    return false;
  }
  public Ent addDef(String n, int t) {
    if(duplCheck(n)) {
      Log.pError("dublicate: "+n); return new Ent(n, UNDEF, C
    }
    Ent e = new Ent(n, t, tbl.size()); tbl.add(e); return e;
  }
  public void enterScope() { stk.push(mark); mark = tbl.size(
  public void exitScope() {
    for(int i = mark; i < tbl.size(); ++i) { tbl.get(i).alive
    mark = stk.pop();
  }
  public Ent lookup(String n) {
```

```
    for(int i = tbl.size()-1; i >= 0; --i) {
      Ent e = tbl.get(i);
      if(e.alive && e.name.equals(n)) { return e; }
    }
    return new Ent(n, UNDEF, 0);
  }
  public int getGsize() { return tbl.size(); }
  public void show() { for(Ent e: tbl) { System.out.println('
  public static class Ent {
    public String name;
    public boolean alive = true;
    public int type, pos = 0;
    public Ent(String n, int t, int p) { name = n; type = t;
    public String toString() {
      return String.format("%s%s[%d %d]", name, alive?"_":"!"
    }
  }
}
```

例題: ブロックスコープの記号表 (4)

ではこの記号表に値を出し入れするドライバを使って試してみます。記号表を作った後、次のコマンドを入力できるようになっています。

- `quit` — 終わる
- `def` 名前 — 指定した名前の変数を定義
- `ref` 名前 — 指定した名前の変数を参照
- `enter` — 新しいスコープに入る
- `exit` — 最内側のスコープを閉じる
- `show` — 表全体を表示[2]

[2]簡単のため上記のコマンドどれかでなければ `show` として扱います。

例題: ブロックスコープの記号表 (5)

```
import java.util.*;

public class SamA1 {
  public static void main(String[] args) throws Exception {
    Symtab st = new Symtab();
    Scanner sc = new Scanner(System.in);
    while(true) {
      System.out.print("> ");
      String[] cmd = sc.nextLine().split(" +");
      if(cmd[0].equals("quit")) {
        System.exit(0);
      } else if(cmd[0].equals("def")) {
        System.out.println(" +> " + st.addDef(cmd[1], Symtab.
      } else if(cmd[0].equals("ref")) {
        System.out.println(" -> " + st.lookup(cmd[1]));
      } else if(cmd[0].equals("enter")) {
        st.enterScope();
      } else if(cmd[0].equals("exit")) {
        st.exitScope();
      } else {
        st.show();
      }
    }
  }
}
```

例題: ブロックスコープの記号表 (6)

では、動かしているところを見てみましょう。

```
% java SamA1
> def i          ←i を定義。
 +> i_[1 0]      ←i のオフセットは0
> def i          ←もういちど定義。
dublicate: i     ←重複定義エラー
 +> i_[-1 0]
> def j          ←j を定義
 +> j_[1 1]      ←j のオフセットは1
> enter          ←新しいスコープに入る
> def i          ←i を定義
 +> i_[1 2]      ←今度は新しい i が作れてオフセットは2
> ref i          ←ここで参照すると
 -> i_[1 2]      ←オフセット2の i が取れる
> exit           ←スコープを出ると
> ref i          ←再度 i を参照すると
 -> i_[1 0]      ←外側のオフセット0の i が取れる
> show           ←記号表表示
 i_[1 0]
 j_[1 1]
 i![1 2]         ←内側の i は網掛けになっている
> quit
```

演習 10-1 上の例をそのまま動かしてみなさい。さらに、より複雑なC言語のプログラム例でやるとどうなるかも、確認してみなさい。

より高度な記号表の構成

前節で示した記号表は極めてシンプルなものでした。とくに、線形探索で「網掛け」のところは単に飛ばすという実装だと、大きなプログラムになったときに速度が問題になる可能性があります。

1つのすぐ考え付く改良としては、記号表本体とは別に、現在見える名前のリストを (前節の例題のように) 並べた表を作り、その上で探索を行なう方法があります (図3)。

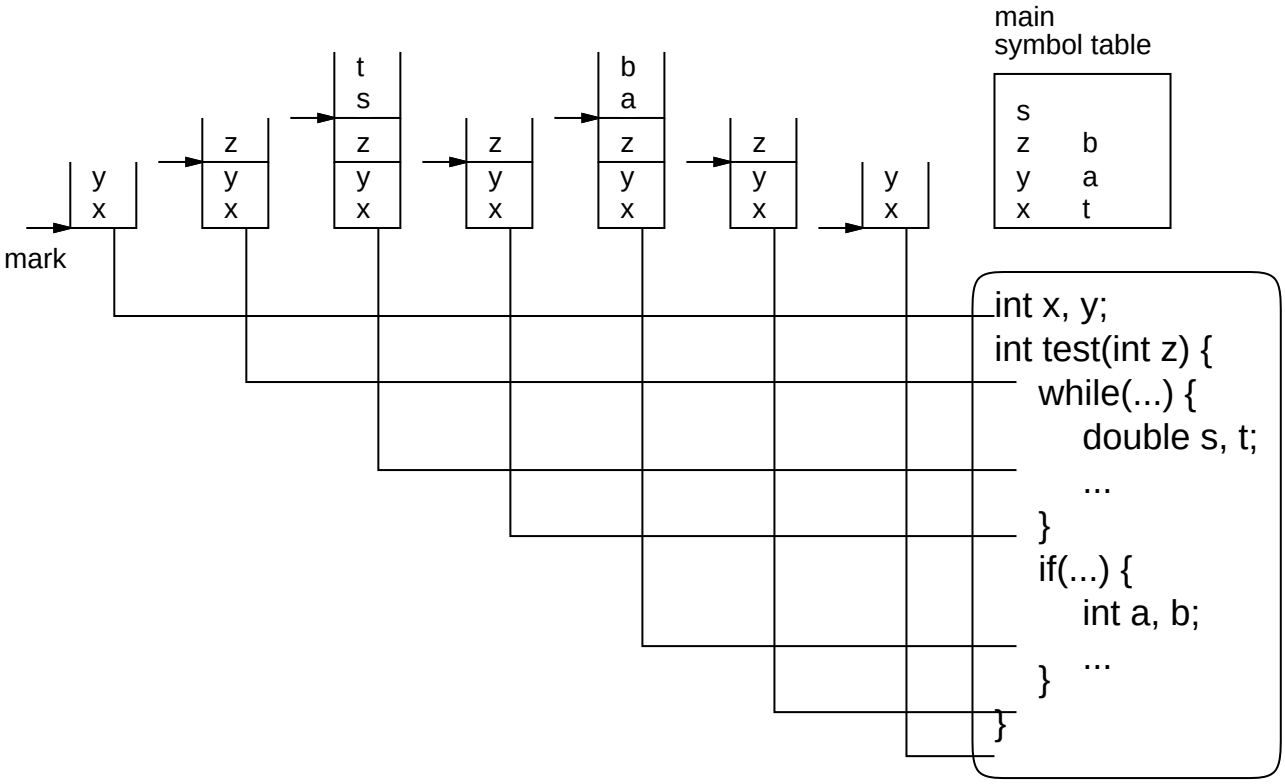

図 3: スコープをスタックで管理する記号表

より高度な記号表の構成 (2)

図で左上の名前が記されている部分は、実際にはテーブル本体の項目へのポインタなどで表現できます。この方法では、スコープに入るごとにマークをセットしてその上にそのスコープで定義された名前を積んで行き、スコープから出るときにそのスコープのぶんは消去してマークも 1 つ外側のスコープに対応する位置まで戻せばよいので、分かりやすくなります。また、線形探索する長さも現在の位置から見える名前のぶんだけになるので、先の方法よりも効率が良くなります。

ここまででは、記号表に名前だけを入れていましたが「どの関数の中でどの変数が定義されているか」などの情報を保持しようと思うと、むしろ関数ごとに 1 つの記号表を作るようにする方が自然です。

また、モジュールやクラスなどの機能のある言語では (Java もそうです)、変数や手続きがモジュールやクラスに所属することになるので、それぞれのモジュールごとに表があり、その中に手続きの表がある、など複雑な構造が必要になります。基本的に記号表は、ソースプログラムにある名前の情報を何らかの形で表現したデータ構造になっている必要があるわけです。

演習 10-2 先の例題の記号表を図 3 のような内部構造で作り直してみなさい。機能は同じままにすること。

演習 10-3 Java 言語のさまざまな名前の情報をすべて保持できるような記号表のデータ構造を設計してみなさい。

前方参照と分割翻訳

前方参照 (**forward reference**) とは一般に、ソースコード上で先 (下) のにある情報を参照することを言います。これに対して後方参照 (**backward reference**) は、前 (上) にある情報を参照することです (図 4)。

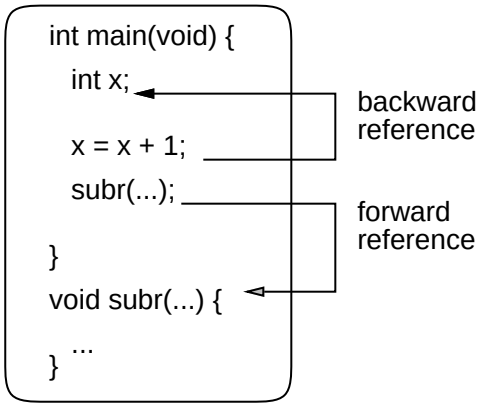

図 4: 前方参照・後方参照

後方参照は処理系にとっては「既に出て来たものの情報を取得」するだけなので、ソースコードを順に処理しながら記号表に登録し、参照するときは記号表を検索すればよいわけです。では前方参照はどうやって扱えばいいでしょうか。

基本的には、まずソースコードを処理しながら名前の情報を記号表に登録する処理だけをおこない、その後で改めて名前の参照が必要な処理をおこなう、というふうにコンパイラの処理を複数回に分けます。このような「ひとまとまりの処理」をコンパイラのパス (**pass**) といい、複数回のパスを持つコンパイラをマルチパス (**multi-pass**) コンパイラと呼びます。[3]

[3] マルチパスでも構文解析を 2 回やる必要は無く、2 パス目は 1 パス目で作った抽象構文木をだどりながら処理するなどでもよいわけです。

前方参照と分割翻訳 (2)

これに対し、1回のパスだけですべての処理を済ませるものはワンパス (single-pass) コンパイラです。過去においてはコンピュータのCPU能力やメモリ容量が限られていたのでパス数を減らすことがしばしば目標となりましたが、今は抽象構文木を保持して複数パスするやりかたが主流です。ただ、プログラミング言語の仕様については、上記の理由からワンパスで処理できるように作られている言語がまだ使われています。実はCやC++がそうです。

CやC++では関数呼出はそれに先だって (ヘッダファイルを使うなどして) プロトタイプ宣言を読ませておく必要がありますね。それは、呼び出しのところで引数や返値の型検査を行なうためにプロトタイプの情報を使うからですが、マルチパスで前方にある関数定義の情報が取れればわざわざプロトタイプを書かせる必要はないはずです。

Javaではプロトタイプ宣言は不要で呼び出し箇所より下にあるメソッドのチェックも問題なくできますが、これはマルチパスを前提とした言語仕様だからです。

ところで、C言語でも前方にある関数が呼べるではないか、と思ったかも知れませんが、トップレベルにある関数や変数の最終的な番地や位置はコンパイラの最終段階でライブラリ内の関数や別ファイルにある関数と合わせて正しい番地が埋め込まれるようになっています。この作業を行なうプログラムのことをリンケージエディタ (linkage editor) ないしリンカ (linker) と呼びます。

前方参照と分割翻訳 (3)

そういうわけで、話題が前後しますが、リンカのおかげでプログラムを複数のファイルに分割して作成し、最後にひとまとめにして実行することができるのです。これを分割コンパイル (separate compilation) と呼び、大規模なプログラム作成には不可欠な機能です。

ただし、多くのリンカは番地を埋めるだけで型検査はやらないので、分割コンパイル時にファイル間でのチェックを行なうには別の方法が必要です。C や C++では、分割コンパイルする各ファイルで共通のヘッダファイルを使うことで、型検査の情報を流通させています。

Java では、クラスファイルに型情報も記載されていて、必要に応じてクラスファイルを読み込む (無ければそのクラスも一緒にコンパイルしてクラスファイルを作る) ことにより、分割コンパイル時の型検査を可能としています。

演習 10-4 C 言語で `main.c` と `sub.c` という 2 つのソースファイルを作り、`sub.c` ではパラメタを 1 個以上受け取る関数 `sub()` を定義するが、`main.c` 側では「本物とは違う」プロトタイプ宣言を与え、両者をコンパイルして結合し、実行してみよ。さまざまな違え方でどのような値が渡るかを検討すること。

型と型検査

型の存在意義・強い型と弱い型

型 (type) とは「データの種別」を表すプログラミング言語の用語である。型がプログラミング言語に取り入れられた最初の理由は、コード生成時に適切な演算命令を出力するためです。たとえば、「1 + 2」も「1.0 + 2.0」も同じ足し算だから演算記号「+」を使用したいけれど、コンピュータのハードウェア上では整数の加算命令と実数の加算命令は違う命令であるので、どちらを出力するかを決める必要があります。型がある言語であれば、被演算子の型が整数なら整数用、実数なら実数用の加算命令を出力すればよいわけです。

型を扱うもう 1 つの理由は、適切でないソースコード記述を検出してエラーにすることです。たとえば a が配列の名前であるとき、a * 5 という式は意味を持ちません。今日ではどちらかというと、こちらの方 (適切でない用途を検出すること) が型の主要な役割になってきています。

上記の説明はいずれも、ソースコードにおいて変数等の型宣言を行ない、翻訳時にチェックや適切な命令の生成を行なう、という文脈で説明していました。このような、翻訳時に型検査を行なう言語を、強い型 (strongly typed) の言語と呼びます。C や Java は強い型の言語の例です。

型の存在意義・強い型と弱い型 (2)

これと対照的に、ソースコード上では型宣言を行なわず、翻訳時の型検査をおこなわない (ないし行なうこともあるが強制しない) 言語を、弱い型 (**weakly typed**) の言語と呼びます。**JavaScript**、**Python**、**Ruby** など多くのスクリプト言語はこちらに属します。

弱い型の言語でも、型は存在する。これらの言語でも、整数、実数、配列など多様なデータを扱い、それらの種類が型に対応するからです。ただし、ソースコード上では型を宣言しないので、翻訳時にはそれぞれの箇所で扱っているデータの型は分かりません。

ではどうするのかというと、これらの言語では実行時に値に型の情報が付随していて (**RTTI — runtime type information**)、それを参照して適切な動作を行なうようになっています。たとえば、次の **Ruby** の実行例を見てみましょう。

```
irb> def addorconcat(x, y) return x + y end
=> :addorconcat
irb> addorconcat 1, 2
=> 3
irb> addorconcat "a", "b"
=> "ab"
```

メソッド addorconcat は、**2** つ値を受け取り、それに対して「+」演算を施して返しますが、データが数値なのか (足し算になる)、文字列なのか (文字列連結になる) かは、実行時まで分かりません。「+」が実際に実行されるときに、**RTTI** を参照して数値なら加算、文字列が含まれていれば連結をおこないます (片方が文字列でないなら文字列への変換も合わせておこないます)。

型の存在意義・強い型と弱い型 (3)

この方法は型宣言が不要なのでコードは簡潔になります、そのかわり翻訳時に型をチェックしないため、翻訳時には型の間違いが検出できなません。不適切な操作はそこを実行たときにエラーとして検出されますが、あらゆる経路の実行を行なうテストというのは実質的には不可能です。また、実行時に **RTTI** を用いて判定し、適切な操作を行なうように切替えるため、処理時間が長くなるという弱点もあります。[4]

強い型の言語の得失はこの裏返しで、宣言が繁雑になりがちですが、翻訳時に型の間違いまでチェックでき、実行が高速にしやすい (たとえば整数どうしの加算と分かっていれば加算命令を **1** つ実行するだけですむ)、という特徴があります。また、プログラムを書いたり読んだりする際に、「このデータはこのような種類のものである」ということが明示され意識される、ということも利点の **1** つだと言えます。

[4] ただし今日では CPU が高速になったことと、実行時の最適化技術が進歩したことから、速度のハンディキャップはかなり小さくなっています。

型検査のアルゴリズム

では実際に強い型の言語における型検査はどのようにすればいいのでしょう。それは基本的には簡単です。式を表す抽象構文木の葉のところには変数や定数が来ますが、変数の型は型宣言により定まりますし、定数はその形から型が分かります。そして、各ノードはその種類ごとに子ノードの型から自分の型を決めることができます。図 5 を見てください。

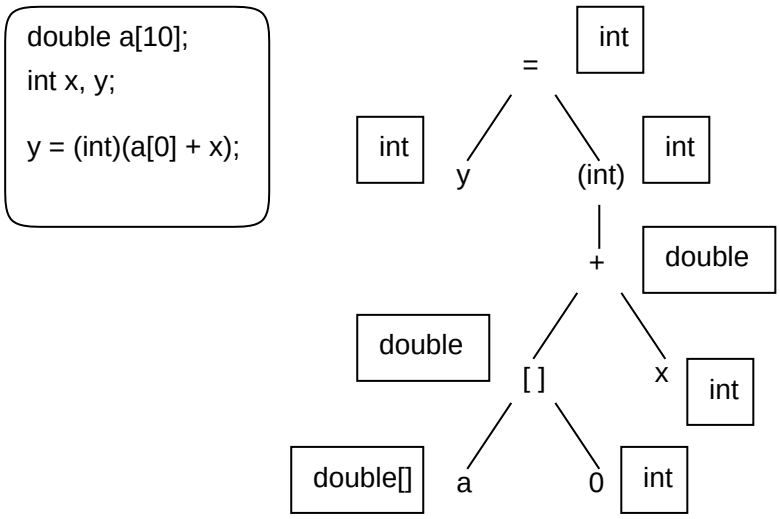

図 5: 抽象構文木の上での型割り当て

配列 a は double の配列であり、定数 0 は int ですから、添字演算は **OK** で (添字の型は整数である必要があります)、その結果は double です。それと int の変数 x を足すことは **OK** で[5] その結果は double です。キャストで double の値を int にすることは **OK** で、代入の左辺 y も int 型ですから、代入も **OK** でその結果は int になります。

[5] このとき int を double に変換してから足す必要があるので、構文木上で変換のノードを挿入するのが普通でしょう。

型検査のアルゴリズム (2)

このように、式のすべての箇所に型が規則通り割り当てられるなら型検査は **OK** になりますし、型が未定だったり矛盾が生じるようならエラーです。どのような演算や代入でどのような条件を満たすべき、ということはそれぞれの言語仕様により定められています (ここの例よりもかなり処理が複雑になる言語もあります)。

例題: 配列型を持つ強い型の小さな言語

では前節の内容を実践する「強い型の小さな言語」を実装してみます。文法は次の通りです。基本的にはこれまでにやってきた「小さな言語」と同じですが、変数宣言があり、配列も宣言できます。

$$
\begin{aligned}
&\mathit{statlist} ::= \mathit{statlist}\ \mathit{stat} \mid \epsilon \\
&\mathit{stat} ::= \textbf{int}\ \mathit{ident}\ \textbf{;} \mid \textbf{int}\ \mathit{ident}\ \textbf{[}\ \mathit{iconst}\ \textbf{]}\ \textbf{;} \mid \mathit{ident} = \mathit{expr}\ \textbf{;} \\
&\qquad \mid \mathit{ident}\ \textbf{[}\ \mathit{expr}\ \textbf{]} = \mathit{expr}\ \textbf{;} \mid \textbf{read}\ \mathit{ident}\ \textbf{;} \mid \textbf{print}\ \mathit{expr}\ \textbf{;} \\
&\qquad \mid \textbf{if}\ \textbf{(}\ \mathit{expr}\ \textbf{)}\ \mathit{stat} \mid \textbf{while}\ \textbf{(}\ \mathit{expr}\ \textbf{)}\ \mathit{stat} \mid \textbf{\{}\ \mathit{statlist}\ \textbf{\}} \\
&\mathit{expr} ::= \mathit{expr}\ \textbf{¿}\ \mathit{expr} \mid \mathit{expr}\ \textbf{¡}\ \mathit{expr} \mid \mathit{nexp} \\
&\mathit{nexp} ::= \mathit{nexp} + \mathit{nexp} \mid \mathit{nexp}\ \textbf{-}\ \mathit{nexp} \mid \mathit{term} \\
&\mathit{term} ::= \mathit{term}\ \textbf{*}\ \mathit{fact} \mid \mathit{term}\ /\ \mathit{fact} \mid \mathit{fact} \\
&\mathit{fact} ::= \mathit{iconst} \mid \mathit{ident} \mid \mathit{ident}\ \textbf{[}\ \mathit{expr}\ \textbf{]} \mid \textbf{(}\ \mathit{expr}\ \textbf{)}
\end{aligned}
$$

では、これに対応するSableCCのソースを見てみましょう。

例題: 配列型を持つ強い型の小さな言語 (2)

```
Package sama2;

Helpers
  digit = ['0'..'9'] ;
  lcase = ['a'..'z'] ;
  ucase = ['A'..'Z'] ;
  letter = lcase | ucase ;

Tokens
  iconst = digit+ ;
  blank = (' '|13|10)+ ;
  int = 'int' ;
  if = 'if' ;
  while = 'while' ;
  read = 'read' ;
  print = 'print' ;
  semi = ';' ;
  assign = '=' ;
  add = '+' ;
  sub = '-' ;
  aster = '*' ;
  slash = '/' ;
  lt = '<' ;
  gt = '>' ;
  lbra = '{' ;
  rbra = '}' ;
  lpar = '(' ;
```

```
  rpar = ')' ;
  lsbr = '[' ;
  rsbr = ']' ;
  ident = letter (letter|digit)* ;

Ignored Tokens
  blank;

Productions
  prog = {stlist} stlist
       ;
  stlist = {stat} stlist stat
         | {empty}
         ;
  stat = {idcl}    int ident semi
       | {adcl}    int ident lsbr iconst rsbr semi
       | {assign}  ident assign expr semi
       | {aassign} ident lsbr [idx]:expr rsbr assign expr sem
       | {read}    read ident semi
       | {print}   print expr semi
       | {if}      if lpar expr rpar stat
       | {while}   while lpar expr rpar stat
       | {block}   lbra stlist rbra
       ;
  expr = {gt}  [left]:nexp gt [right]:nexp
       | {lt}  [left]:nexp lt [right]:nexp
       | {one} nexp
       ;
```

```
nexp = {add}  nexp add term
     | {sub}  nexp sub term
     | {one}  term
     ;
term = {mul}  term aster fact
     | {div}  term slash fact
     | {one}  fact
     ;
fact = {iconst} iconst
     | {ident}  ident
     | {aref}   ident lsbr expr rsbr
     | {one}    lpar expr rpar
     ;
```

例題: 配列型を持つ強い型の小さな言語 (3)

```
package sama2;
import sama2.analysis.*;
import sama2.node.*;
import java.io.*;
import java.util.*;

class TypeChecker extends DepthFirstAdapter {
  Symtab st;
  HashMap<Node,Integer> vtbl;
  public TypeChecker(Symtab tb, HashMap<Node,Integer> vt) { s
  private void ckv(Node n, int t, int l, String s) {
    if(getOut(n) instanceof Integer && (Integer)getOut(n) ==
    Log.pError(l+": "+s);
  }
  private int cki(String i, int t, int l) {
    Symtab.Ent e = st.lookup(i);
    if(e.type != t) { Log.pError(l+": identyfier is not of ex
    return e.pos;
  }
  @Override
  public void outAIdclStat(AIdclStat node) {
    st.addDef(node.getIdent().getText(), Symtab.ITYPE);
  }
  @Override
  public void outAAdclStat(AAdclStat node) {
    Symtab.Ent e = st.addDef(node.getIdent().getText(), Symta
    vtbl.put(node, e.pos);
```

```
}
@Override
public void outAAssignStat(AAssignStat node) {
  int p = cki(node.getIdent().getText(), Symtab.ITYPE, node
  ckv(node.getExpr(), Symtab.ITYPE, node.getAssign().getLin
  vtbl.put(node, p);
}
@Override
public void outAAassignStat(AAassignStat node) {
  int p = cki(node.getIdent().getText(), Symtab.ATYPE, node
  ckv(node.getIdx(), Symtab.ITYPE, node.getLsbr().getLine()
  ckv(node.getExpr(), Symtab.ITYPE, node.getAssign().getLin
  vtbl.put(node, p);
}
@Override
public void outAReadStat(AReadStat node) {
  int p = cki(node.getIdent().getText(), Symtab.ITYPE, node
  vtbl.put(node, p);
}
@Override
public void outAPrintStat(APrintStat node) {
  ckv(node.getExpr(), Symtab.ITYPE, node.getPrint().getLine
}
@Override
public void outAIfStat(AIfStat node) {
  ckv(node.getExpr(), Symtab.ITYPE, node.getLpar().getLine(
}
@Override
```

```
public void outAWhileStat(AWhileStat node) {
  ckv(node.getExpr(), Symtab.ITYPE, node.getLpar().getLine(
}
@Override
public void inABlockStat(ABlockStat node) { st.enterScope()
@Override
public void outABlockStat(ABlockStat node) { st.exitScope()
@Override
public void outAGtExpr(AGtExpr node) {
  ckv(node.getLeft(), Symtab.ITYPE, node.getGt().getLine(),
  ckv(node.getRight(), Symtab.ITYPE, node.getGt().getLine()
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
public void outALtExpr(ALtExpr node) {
  ckv(node.getLeft(), Symtab.ITYPE, node.getLt().getLine(),
  ckv(node.getRight(), Symtab.ITYPE, node.getLt().getLine()
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
public void outAOneExpr(AOneExpr node) { setOut(node, getOu
@Override
public void outAAddNexp(AAddNexp node) {
  ckv(node.getNexp(), Symtab.ITYPE, node.getAdd().getLine()
  ckv(node.getTerm(), Symtab.ITYPE, node.getAdd().getLine()
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
```

```
public void outASubNexp(ASubNexp node) {
  ckv(node.getNexp(), Symtab.ITYPE, node.getSub().getLine()
  ckv(node.getTerm(), Symtab.ITYPE, node.getSub().getLine()
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
public void outAOneNexp(AOneNexp node) { setOut(node, getOu
@Override
public void outAMulTerm(AMulTerm node) {
  ckv(node.getTerm(), Symtab.ITYPE, node.getAster().getLine
  ckv(node.getFact(), Symtab.ITYPE, node.getAster().getLine
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
public void outADivTerm(ADivTerm node) {
  ckv(node.getTerm(), Symtab.ITYPE, node.getSlash().getLine
  ckv(node.getFact(), Symtab.ITYPE, node.getSlash().getLine
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
public void outAOneTerm(AOneTerm node) { setOut(node, getOu
@Override
public void outAIconstFact(AIconstFact node) {
  setOut(node, new Integer(Symtab.ITYPE));
}
@Override
public void outAIdentFact(AIdentFact node) {
  Symtab.Ent e = st.lookup(node.getIdent().getText());
```

```
    setOut(node, new Integer(e.type)); vtbl.put(node, e.pos);
  }
  @Override
  public void outAArefFact(AArefFact node) {
    int p = cki(node.getIdent().getText(), Symtab.ATYPE, node
    ckv(node.getExpr(), Symtab.ITYPE, node.getLsbr().getLine(
    setOut(node, new Integer(Symtab.ITYPE)); vtbl.put(node, p
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOu
}
```

例題: 配列型を持つ強い型の小さな言語 (4)

ではこれを動かすmain()を見てみます。記号表を作り、パーサを作り、解析し、記号表を表示します。この後の実行部分はコメントアウトしてあります。

```
package sama2;
import sama2.parser.*;
import sama2.lexer.*;
import sama2.node.*;
import java.io.*;
import java.util.*;

public class SamA2 {
  public static void main(String[] args) throws Exception {
    Parser p = new Parser(new Lexer(new PushbackReader(
      new InputStreamReader(new FileInputStream(args[0]), "JI
        1024)));
    Start tree = p.parse();
    Symtab st = new Symtab();
    HashMap<Node,Integer> vtbl = new HashMap<Node,Integer>();
    TypeChecker tck = new TypeChecker(st, vtbl); tree.apply(t
//    if(Log.getError() > 0) { return; }
//    Executor exec = new Executor(vtbl, st.getGsize()); tree
  }
}
```

例題: 配列型を持つ強い型の小さな言語 (5)

演習 10-5 「型のある小さな言語」で簡単なプログラムを書いてみなさい。変数の未定義や配列と変数の間違いなどを入れてみて、確かに型検査されていることを確認しなさい。

演習 10-6 「型のある小さな言語」を次のように拡張してみなさい。もちろん、型検査はきちんと行なわれること。

a. 変数に初期値が書けるようにする。

b. 配列の変数どうしの代入もできるようにする。

c. 配列リテラル (`[1，2，3]` のようなもの) を入れる。

d. その他、やってみたいと思う拡張。

課題 10A

今回の演習問題から (小問を)1 つ以上選び、プログラムを作成しなさい。作成したプログラムについてレポートを作成し、久野 (y-kuno@uec.ac.jp) まで PDF を送付してください。LaTeX の使用を強く希望します。レポートは次の内容を含むこと。期限は次回授業前日一杯。レポートおよびその評点はクラス内で公開します。

- タイトル – 「システムソフトウェア特論 課題＃ 9」、学籍番号、氏名、提出日付。
- 課題の再掲 — レポートを読む人がどの課題をやったのか分かる程度にやった課題を要約して説明してください。
- 方針 — その課題をどのような方針でやろうと考えたか。
- 成果物 — プログラムとその説明および実行例。
- 考察 — 課題をやってみて分かったこと、気付いたことなど。
- 以下のアンケートの解答。

  Q1. 強い型の言語と弱い型の言語のどちらが好みですか。またそれはなぜ。

  Q2. 記号表と型検査の実装について学んでみて、どのように思いましたか。

  Q3. リフレクション (課題をやってみて気付いたこと)、感想、要望など。

実行するためのExectutorのコード

```
package sama2;
import sama2.analysis.*;
import sama2.node.*;
import java.io.*;
import java.util.*;

class Executor extends DepthFirstAdapter {
  Scanner sc = new Scanner(System.in);
  PrintStream pr = System.out;
  HashMap<Node,Integer> pos;
  Object[] vars;
  public Executor(HashMap<Node,Integer> p, int s) { pos = p;
  @Override
  public void outAAdclStat(AAdclStat node) {
    vars[pos.get(node)] = new Object[Integer.parseInt(node.ge
  }
  @Override
  public void outAAssignStat(AAssignStat node) {
    vars[pos.get(node)] = getOut(node.getExpr());
  }
  @Override
  public void outAAassignStat(AAassignStat node) {
    Object[] a = (Object[])vars[pos.get(node)];
    a[(Integer)getOut(node.getIdx())] = getOut(node.getExpr()
  }
  @Override
  public void outAReadStat(AReadStat node) {
```

```
  String s = node.getIdent().getText().intern();
  pr.print(s + "> "); vars[pos.get(node)] = sc.nextInt(); s
}
@Override
public void outAPrintStat(APrintStat node) {
  pr.println(getOut(node.getExpr()).toString());
}
@Override
public void caseAIfStat(AIfStat node) {
  node.getExpr().apply(this);
  if((Integer)getOut(node.getExpr()) != 0) { node.getStat()
}
@Override
public void caseAWhileStat(AWhileStat node) {
  while(true) {
    node.getExpr().apply(this);
    if((Integer)getOut(node.getExpr()) == 0) { return; }
    node.getStat().apply(this);
  }
}
@Override
public void outAGtExpr(AGtExpr node) {
  if((Integer)getOut(node.getLeft()) > (Integer)getOut(node
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
```

```
@Override
public void outALtExpr(ALtExpr node) {
  if((Integer)getOut(node.getLeft()) < (Integer)getOut(node
    setOut(node, new Integer(1));
  } else {
    setOut(node, new Integer(0));
  }
}
@Override
public void outAOneExpr(AOneExpr node) { setOut(node, getOu
@Override
public void outAAddNexp(AAddNexp node) {
  int v = (Integer)getOut(node.getNexp()) + (Integer)getOut
  setOut(node, new Integer(v));
}
@Override
public void outASubNexp(ASubNexp node) {
  int v = (Integer)getOut(node.getNexp()) - (Integer)getOut
  setOut(node, new Integer(v));
}
@Override
public void outAOneNexp(AOneNexp node) { setOut(node, getOu
@Override
public void outAMulTerm(AMulTerm node) {
  int v = (Integer)getOut(node.getTerm()) * (Integer)getOut
  setOut(node, new Integer(v));
}
@Override
public void outADivTerm(ADivTerm node) {
```

```
    int v = (Integer)getOut(node.getTerm()) / (Integer)getOut
    setOut(node, new Integer(v));
  }
  @Override
  public void outAOneTerm(AOneTerm node) { setOut(node, getOu
  @Override
  public void outAIconstFact(AIconstFact node) {
    setOut(node, new Integer(node.getIconst().getText()));
  }
  @Override
  public void outAIdentFact(AIdentFact node) {
    setOut(node, vars[pos.get(node)]);
  }
  @Override
  public void outAArefFact(AArefFact node) {
    Object[] a = (Object[])vars[pos.get(node)];
    setOut(node, a[(Integer)getOut(node.getExpr())]);
  }
  @Override
  public void outAOneFact(AOneFact node) { setOut(node, getOu
}
```

実行するためのExectutorのコード (2)

では、次のような簡単なプログラムを実行してみましょう。nを入力し、配列にn個の値を入力し、逆順に出力しています。なお「-1」がうまく扱えないので、「0-1」と演算しています。

```
int a[100]; int n; int i; int d;
read n;
i = 0; while(i < n) { read d; a[i] = d; i = i + 1; }
i = n - 1; while(i > 0-1) { print a[i]; i = i - 1; }
```

実行のようすは次の通り。

```
% java Sam92 test.min
 a_[2 0]
 n_[1 1]
 i_[1 2]
 d_[1 3]
n? 3
d? 8
d? 9
d? 10
10
9
8
%
```