コンパイラ　最終レポート

東京理科大学 工学部 情報工学科

4617041 白江 涼雅

1. 拡張した機能の説明

1.1 余剰計算

"a % b" (aとbは変数または定数)とすることでaをbで割ったあまりを計算できるようにした。 優先度は*や/と同様。

1.2 累乗計算

"a ^ b" (aは変数または定数,bは定数)とすることでaをb乗する計算ができるようにした。

1.3 for文

"for st1; cond; st2; do stmts endfor"

- st1はcondで処理する変数の初期化する式、
- condはループするかを判断する条件式、
- st2はループしたときどんな処理をするか、
- stmtsはループ内での処理、
- endforはforの終わりを示す。

condの条件を満たす限りfor からendfor までのループできるようにする。また、ループ処理が行われるたびにst2の処理を行うことができるようにした。

1.4 (前置、後置)インクリメント、デクリメント

"++i","i++"とすることでiをプラス1,"--i","i--"とすることでiをマイナス1できるようにした。 左辺値がなくても動き、式の中にも組み込むことができる。 変数の前に記号がきてるときは、計算している行でも影響がでるようにした。 記号が後ろに記号がきてるときは、計算している行では影響がでない。

2.実現方法

2.1 余剰計算

最終的にしたい処理を擬似コードを書くと

```
return a-b *( a / b)
```

しかしはじめにa/bをするために "mergecode(mergecode(a,b),makecode(O_OPR,0,5));" をするが、code.cをみてみるとmergecodeをすると2つめの引数側がfreeされてしまう。 そのため、cptr*の中身のみを保存する関数をcode.cを理解してからつくった。 code.cとcode.hに書き加えた。

```
cptr* copy_cptr(cptr* cpt){ //構造体cptrをコピーする関数
  return makecode(cpt->h->f,cpt->h->l,cpt->h->a);
}
```

あとは初めに述べたコードをDIVやMULTと同じ高さに言語化するだけ。

```
| T MOD  F
{
>---cptr *tmp,*tmp1,*tmp2,*tmp3,*tmp4;
>---/* したいこと: mod = a - b * (a / b) であまりが出る */
>---/* mergecodeすると$1.code,$3.codeが消えるからできない */
>---/* copy_cptr作成で解決 */
>---tmp1 = copy_cptr($1.code); tmp2 = copy_cptr($3.code);
>---tmp3 = copy_cptr($1.code); tmp4 = copy_cptr($3.code);
>---tmp = mergecode(mergecode(tmp1,tmp2),makecode(O_OPR,0,5));
>---/* b*tmp */
>---tmp = mergecode(mergecode(tmp4,tmp),makecode(O_OPR,0,4));
>---/* a-tmp */
>---tmp = mergecode(mergecode(tmp3,tmp),makecode(O_OPR,0,3));
>---$$.code = tmp;
}
```

2.2 累乗計算

最終的にしたい処理を疑似コードを書くと

```
sum = 1
ROOP b回繰り返す
    sum *= a
ENDROOP
```

ここで難しい作業はROOPをforのint型で書きたいが、cptr*型である bを引数としたときのループをどう表現するかである。 cmm.yのコードを見ると、yylval.valでNUMBERの値をint型で取得している旨のコードがあった。それを使いfor文を書いて処理した。 また、優先度はMULTやMODよりも 1 つ高くするために非終端を一つ増やした。

```
pow : F POW NUMBER
{
>---int right = yylval.val;
>---cptr* sum = makecode(O_LIT,0,1);
>---for(int i=0;i<right;i++){
>--->---sum =
```

```
mergecode(mergecode(sum,copy_cptr($1.code)),makecode(O_OPR,0,4));
>---}
>---$$.code = sum;
}
```

2.3 for文

したいことを擬似コードで書くと、

```
初期化する
もどってくるためのラベル1設置
もし、条件が偽ならループを抜ける（ラベル2に飛ぶ）
    forの中身の計算
    forをひとつ進める
無条件でラベル1に飛ぶ
ループを抜けるためのラベル2設置
```

である。while文がもともとあったので参考にしたのと、引数を気をつけて書いた。 また、優先度はwhileと同じ位置。

```
forstmt : FOR st cond SEMI st DO stmts ENDFOR
{
>---int for0,for1;
>---for0 = makelabel();for1 = makelabel();
>---cptr *tmp;
>---/* init をinitしつつ、ラベル設置 */
>---tmp = mergecode($2.code,makecode(O_LAB, 0, for0));
>---/* falseでroopしゅうりょう  */
>---tmp = mergecode(mergecode(tmp,$3.code),makecode(O_JPC,0,for1));
>---/* 満を持して中身計算   */
>---tmp = mergecode(tmp,$7.code);
>---/* forをひとつ進める */
>---tmp = mergecode(tmp,$5.code);
>---/* ループ処理 */
>---tmp = mergecode(tmp,makecode(O_JMP,0,for0));
>---tmp = mergecode(tmp,makecode(O_LAB,0,for1));
>---$$.code = tmp;
>---$$.val = 0;
}
```

2.4 インクリメント、デクリメント

4つ書くのは冗長なので、インクリメントついてのみ書く。 したいことを擬似コードでかくと

```
i =  i+1
```

こう書くとかんたんにみえるが、左辺値がなく、変数にそのまま数値をいれなければならない。 コードと教科書を見ているとsearch_all で位置を見つけて、その位置に対してLODしていけばいいとわかった。 また、左辺値がないときとあるときがあるので、優先度は pre_inc;のときの処理はforと同じ優先度、組み込まれている場合はpowよりも高い優先度にしたいので非終端記号をさらに一つ増やした。

```
pre_inc : PLPL ID
{
>---/*なぜかうまくいかない*/ /* 全体でlevelを管理してた */
>---/* iは+して,その行での評価はiのまま*/
>---/* NUMBERを直接よみこむ*/
>---list *tmpl = search_all($2.name);
>---if(tmpl == NULL) sem_error2("pre_inc");
>---cptr *tmp;
>---tmp = mergecode(makecode(O_LOD,level - tmpl->l,tmpl-
>a),makecode(O_LIT,0,1));
>---tmp = mergecode(tmp,makecode(O_OPR,0,2));
>---tmp = mergecode(tmp,makecode(O_STO,level-tmpl->l,tmpl->a));
>---tmp = mergecode(tmp,makecode(O_LOD,level-tmpl->l,tmpl->a));
>---$$.code = tmp;
}
;
```

++iは上のコードでいいが、i++はその行での処理はもともとのiの値でなければ行けないので、返す値が変わる。

```
post_inc : ID PLPL
{
>---/* 実際の変数の場所の数値はかえておいて、出力するのは普通の値*/
>---list *tmpl = search_all($1.name);
>---if(tmpl == NULL) sem_error2("pre_de");
>---cptr *tmp;
>---cptr *out;
>---tmp = mergecode(makecode(O_LOD,level - tmpl->l,tmpl-
>a),makecode(O_LIT,0,1));
>---tmp = mergecode(tmp,makecode(O_OPR,0,2));
>---tmp = mergecode(tmp,makecode(O_STO,level-tmpl->l,tmpl->a));
>---tmp = mergecode(tmp,makecode(O_LOD,level-tmpl->l,tmpl->a));
>---out = mergecode(mergecode(tmp,makecode(O_LIT,0,1))
>--->--->--->--->---,makecode(O_OPR,0,3));
>---$$.code = out;
}
;
```

考察

3.1 余剰計算

もともとのC言語にもいえることだが、aが負だったときや逆元処理ができない。 a<0については負のとき a= a+b をさせたり、ぎりぎりまで a += b * n(nは正数)することで解決できる。 しかしこれらはコーナーケースを考えだすと不毛なので実装しなかった。 逆元は フェルマーの小定理や拡張ユークリッドの互除法で可能になるが、実装力が追いつかなかった。

3.2 累乗計算

b側を変数も可能にすることができない。 理由はNUMBERのときはともかく、Eから読み込ませた時 yylval.valによってint型に変換することができない。 そのため、わけて実装するとして、変数の時sum *= aを繰り返す方法がわからなかった。 おそらく、for文の形を利用してうまく記述すればいけるはずだが、うまくいかなかった。

3.3 for文

for文でだめなケースは浮かばなかった。 C言語のコンパイラとしてfor文はうまく処理できるだろう。

3.4 インクリメント、デクリメント

インクリメント、デクリメントは本来の優先度と違う。 前置に関してはhoge(),hoge{},hoge[]と同等だが、同じにしてしまうとシンタクスエラーを吐く。 改善するポイントのひとつである。

その他の実装

残りの項目に関してはpl0i自体を変更する必要があるように感じた。

- gotoはgoto -> labelの順の時、読み込めていないままgoすることになるのでエラーを吐く。その処理を補うことがpl0iに必要だった。
- caseはswitchの数だけ分岐を付け加えなくてはならず、それを後読みで変更することができなかった。これもgotoと似たような問題点である。
- 配列 listやcptrはリストで管理されているため一見楽そうにみえるが、mergeした時にfreeされてしまう問題やbit管理を充分に行わなくてはならないため、楽に実装するには結局C言語の配列を頼りにする必要があった。

その他

%や^は+や*と同じ演算子であるため、O_OPRで扱えるようにしたほうが見栄えが良く、更に実装も楽になり穴も少なく済むように感じた。

感想

授業では実装についてはあまり触れてもらえず、コードに関しては冒頭の部分の説明だけだったため実装がかなりきつかった。 自分なりに考えつつ、コードを読み込むことでなんとか形になるものは作れた。 かなり時間をかけて作ったためにその分コンパイラに対して深い理解を得ることができ、構文解析のコードを読む力も養われたと思う。 課題に関しては、大方理解できたからといってすべてできるわけではなく、それぞれに難しいポイントが詰め込まれていて良い課題だった。