実験 B 言語処理系　補足資料

金子研究室 TA 衆

1. はじめに

この配布資料は，実験を進めるにあたり,「何から手をつけたらよいのかわからない」という人のためのものです．UNIX 系 OS 環境下でのターミナルの扱いに習熟している人は，特にこの先を読む必要はありません．実験用ページ

http://www.tuat.ac.jp/~k1kaneko/gengo/

を参照し，自力で課題を進めたほうがスムースであると思われます．いずれにしても，まずテキストの実験課題のところをよく読んで，目的をはっきりさせることから始めてください．

2. OS の選択

本実験は基本的に Linux 環境下でおこなうことを前提にしています．ですが，学科計算機室のマシンには Cygwin がインストールされていますので，Windows 上で実験を完遂することも可能です．おそらく大半の人にとっては Windows のほうがやりやすいでしょうから，これ以降の説明では Cygwin を利用した実験の手順を示します．Linux でも多少の違いを除けばそのまま通用すると思います．トラブルが起きたら TA にでも聞いてください．

3. まずは実験道具の準備です

本実験は「言語処理系を 1 から作れ」というものではありません．あらかじめ以下の実験用ディレクトリ

http://www.tuat.ac.jp/~k1kaneko/gengo/

上にコンパイラ等のソースコードが用意されているので，それらに対し，改良をくわえる形でおこないます．

3.1 ソースコードの入手と展開

まずはこれらソースコードを手に入れることからはじめましょう．ブラウザ（IE でも Firefox でもなんでもいいです）で上記ディレクトリを開き，79.5KB の”gengo2002.tar.gz”というファイルを見つけてダウンロードしてください．ダウンロード先は計算機室の環境だと ”Z:\Cygwin” にするのが良いでしょう（Cygwin のホームディレクトリになっているので）．もちろん，任意の場所にダウンロード後，　”Z:\Cygwin”　にファイルを移動させても構いません．また，以降の説明中でのコマンドライン操作のやり方が気に入らない人は無理に従う必要はまったくありません．

ここまでできたら，Cygwin を起動してください．まずは,

```
$ ls
```

と打ち込んでみましょう．gengo2002.tar.gz が見つかるはずです．このファイルは圧縮されているので，展開して中身を取り出す必要があります．

```
$ tar xvzf gengo2002.tar.gz                    ←tar.gz ファイルの展開
```

すると，新しく gengo2002 というディレクトリができ，その下にファイルが展開されます．

```
$ cd gengo2002                           ←gengo2002 ディレクトリに移動
$ ls                               ←gengo2002 ディレクトリ下のファイル表示
```

”p386”，”zinc-0”，”znas”という 3 つのディレクトリが見えているはずです．これらのなかに実験で使用する道具（のもと）が入っています．それぞれの詳細はあとで説明するとして，とりあえずは道具を完成させてしまいましょう．まずは znc(本日みなさんがいじるコンパイラです) からいきましょう．

3.2 znc のビルド

```
$ cd zinc-0                                  ←zinc-0 ディィレクトリに移動
$ make clean                                 ← まずは掃除から
$ make znc                     ← コンパイラ znc の作成（警告が出ますが気にしない）
```

ここで,「make」というコマンドが登場してきました．zinc-0 ディィレクトリ内で make コマンドを実行すると，VisualC++でいうところの「ビルド」のようなことを make が勝手にやってくれます．make で何をしているのかを知りたい人は,「Makefile」というファイルをみつけてテキストエディタで開いてみてください．面倒な人は次にいきましょう．続いて計算機シミュレータ p386 を znc のときと同じ要領でビルドします.

3.3 p386 のビルド

```
$ cd ../p386                                 ←
".."は現在ディレクトリの親ディレクトリ（1 つ上，つまり gengo2002 ディィレクトリ．その下の p386 に移動）
$ make clean                                 ← ここでもファイルの掃除から
$ make p386                    ← 仮想機械 p386 の作成（警告が出ますが気にしない）
```

さて，これで znc(コンパイラ)・p386(計算機シミュレータ) は完成しました．残るは znas(アセンブラ) です．ところが，実のところ znas は awk スクリプトなので，make（コンパイル・リンク・ビルド）する必要はありません．ですから，これで実験で使う道具はとりあえず揃ったことになります.

3.4 テキストエディタの選択

実験道具は揃いました．次からいよいよプログラムを書いたり，処理系のコードを改良していく作業に入りますが，その際にくれぐれも Notepad(メモ帳) や Wordpad(ワードパッド) などを使わないようにご注意ください．本実験で扱うソースコードは元は Unix 系用なので，文字コードおよび改行コードが違います (Shift-JIS と EUC, CR+LF と LF)．Notepad だと読むことすらできませんし，Wordpad だと読むことはできますが，保存時に問題が発生する場合があります.「秀丸」「Terapad」「サクラエディタ」「xyzzy」などを用いれば変換作業を自動でやってくれるので（設定はしないとダメですが），これらを使うのが便利です.「Terapad」の場合を例にとってみます.

http://www5f.biglobe.ne.jp/~t-susumu/library/tpad.html

にアクセスし,「tpad093.zip (527KB)」というのを落としましょう．これをデスクトップなどの適当なフォルダに展開し,「Terapad.exe」を起動してください．起動したらメニューの「表示」⇒「オプション」をクリックします (図 1 左)．オプション設定のウインドウが出てくるので,「文字コード」タブを選択し，図 1 右のように設定して「OK」してください．EUC と LF の組合せです．このあとは編集したいソースコードを常に Terapad で開くようにすれば，文字コードに関する問題で悩むことはなくなります.

4. znc,znas,p386 について

ここでは先ほど作った 3 つの道具（znc，znas，p386）を実際に使っていきます.

4.1 znc(コンパイラ)

ZINC という言語 (文法は実験テキストを参照) のプログラム（実験テキストの課題例参照）をアセンブリ（アセンブラ）コードに変換します．早速使ってみましょう．ZINC のサンプルプログラムは実験ページに用意されていますので,

http://www.tuat.ac.jp/~k1kaneko/gengo/gcd.Zn

から取得してください．ダウンロードができたら，内容を確認してみましょう．拡張子は Zn となっていますが，実体は単なるテキストファイルです．gcd.Zn を右クリックして”アプリケーションから開く”を選択し，テキストエディタで開いてみてください．メモ帳（Notepad）では文字コードの違いで正しく読めないはずです．Terapad, 秀丸（2 つのうちどちらかは学科端末にインストールされていると思います）などを使ってください．確認がで

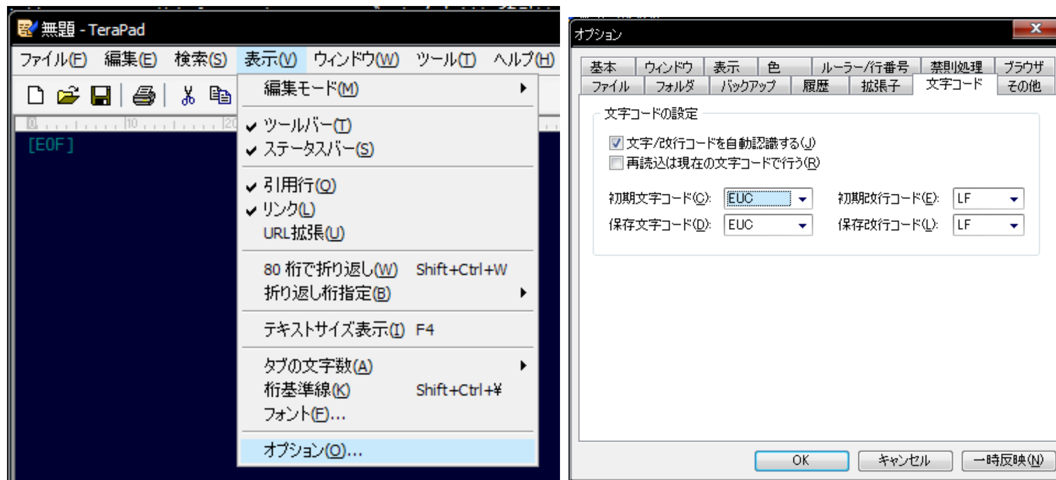

図 1: Terapad の設定

きたら，実際にコンパイルしてみましょう．gcd.Zn を znc のあるディレクトリ（Z:\Cygwin\gengo2002\zinc-0）においてください．そして Cygwin コマンドライン上においても znc のあるディレクトリに移り，コンパイルします．

```
$  cd ~/gengo2002/zinc-0            ← ディレクトリ移動（チルダはホームディレクトリをあらわす）
$ ./znc < gcd.Zn > gcd.s                ←"gcd.Zn"というソースファイルを znc でコンパイル，
コンパイラ znc の出力を"gcd.s"ファイルにリダイレクト
```

（補足：znc はデフォルトで標準入力から ZINC プログラムを読み，標準出力に結果を出力します．普通，標準入力はキーボード，標準出力は端末に割当てられていますので，リダイレクトを使って znc への入力を"gcd.Zn"に、znc からの出力（コンパイル結果）を"gcd.s"というファイルにしたわけです．なお，"gcd.Zn"も"gcd.s"もテキストファイルなので，拡張子含め，ファイル名は任意のものに変更することが可能です.）
ここで，Cygwin 上から ls するか，エクスプローラからファイルを見ると，"gcd.s"というファイルができていると思います．これが znc コンパイラの出力したアセンブリ・コードになります．このアセンブリ・コードではまだ実行ができないので，今度は"gcd.s"を znas(アセンブラ) にかけることを考えましょう．そのためにはコンパイルが終わったら，アセンブリコード"gcd.s"をアセンブラ znas のあるディレクトリに移動，またはコピーします．Cygwin 上でファイル移動・コピーをおこなう場合は，次の 2 種類のうちどちらかの方法を使ってください．

```
$ mv gcd.s ../znas/                          ← 移動する場合は mv で
$ cp gcd.s ../znas/                          ← コピーする場合は cp で
```

また，図 2 のようにエクスプローラを 2 つ開いて D&D でもかまいません．

4.2 znas(アセンブラ)

znas は awk 言語で書かれたアセンブラです.（詳細は http://www.tuat.ac.jp/~k1kaneko/gengo/append.html および http://www.tuat.ac.jp/~k1kaneko/gengo/append2.html 参照）．znas ディレクトリに移動し，"gcd.s"をアセンブラにかけ，ファイル出力します．やりかたは先ほどの znc のときと同じです．"error in 2"などのエラーが出る場合は，TA を呼んでください．

```
$ cd ../znas/                              ←znas のあるディレクトリに移動
$ ./znas < gcd.s > gcd.p3                  ←"gcd.s"をアセンブルし，p386 用機械語コー
ド"gcd.p3"に出力
```

これで，機械語に変換できたことになります．"gcd.p3"の内容を確認してみてください (テキストエディタで見ることができます)．16 進数 2 桁（8 ビット）の羅列を見ることができればアセンブルは（多分）成功です．ここまでやって，ようやく ZINC 言語で書いたプログラムを実行することができます．しかし，先ほど znc でコンパイル後，出力の"gcd.s"を znas ディレクトリに移したのと同様に，やはりここでも出力"gcd.p3"を p386 ディレクト

図 2: エクスプローラから移動させる

リに移動させなければなりません．

```
$ mv gcd.p3 ../p386/                    ← 移動する場合は mv で
$ cp gcd.p3 ../p386/                    ← コピーする場合は cp で
```

4.3 p386(計算機シミュレータ)

p386 はシミュレータです．ZINC 言語で書いたプログラムは最終的にここでその動作を確認することができます．(詳細は http://www.tuat.ac.jp/~k1kaneko/gengo/append.html 参照)

```
$ cd ../p386/              ← シミュレータのあるディレクトリに移動
$ ./p386 gcd.p3             ← 実行（リダイレクトしなくてよい）
```

p386 が動き出したら，とりあえず Enter キーを押してみてください．可能な操作が表示されます．ここでは g を選んでみましょう．すると”13 HALT” と表示されたはずです（最初の gcd.Zn の数値をいじっていなければ）．この実験で使う znc，znas，p386 の説明は以上です．

4.4 自動化スクリプト

コマンドラインからのディレクトリ移動・ファイル操作に慣れていない人はここまでの手順が頭の中でうまく整理できないかもしれません．また，実験途中で何度も同じようなコマンドを打たされるのは苦痛であるかとも思います．そこで，リスト 1 のようなシェルスクリプトを用意しておくと便利です．jikkou.sh という名前のテキストファイルを新しく zinc-0 ディレクトリに作成し，内容をリスト 1 のようにします．

リスト 1: jikkou.sh

```
1  #!/bin/sh
2  FILENAME=$1
3  SPLITNAME=${FILENAME%\.*}
4  ./znc < $FILENAME > $SPLITNAME.s
5  cp $SPLITNAME.s ../znas
6  cd ../znas
7  ./znas < $SPLITNAME.s > $SPLITNAME.p3
8  cp $SPLITNAME.p3 ../p386
9  cd ../p386
10 ./p386 $SPLITNAME.p3
11 cd ../zinc-0
```

この jikkou.sh を zinc-0 ディレクトリで以下のように呼び出すと，

```
$ make clean
$ make znc              ←znc コンパイラをビルド (これはコンパイラのソースを変更したときのみおこなう)
$ ./jikkou.sh test.zn             ←ZINC コンパイル・znas アセンブル・p386 実行のバッチ処理
```

ZINC 言語プログラム test.zn をコンパイラ znc にかけて test.s を得，test.s を znas アセンブラにかけて test.p3 を得，test.p3 を p386 シミュレータ上で実行するという一連の動作をわずかなコマンドタイプで行うことができるようになります．また，課題 2 で znc コンパイラのファイル lex.c，parse.c，codegen.c などを修正した直後は make znc によってコンパイラをビルドしなおす必要がありますが，それ以外は毎回ビルドを行う必要はありません．

5. 実験，その後

本実験は結構重めなので，時間内に課題が終わらないことも考えられます．また，特に年末年始は端末室が閉鎖されていることが多いです．そこでレポートを書く際に，自宅など端末室以外の環境で追加実験をしたくなることがあるかもしれません．ここでは参考に情報をいくつかのせておきます（注：ここに書いてあることについては内容の保証はできません．実験はなるべく端末室で終わらせるようにしてください．きついですが）．

- 実験ページを参照したい：学外からでも閲覧可能です．適宜オンラインマニュアルとしてご利用ください．
- 家に Linux 環境がない ：Cygwin をインストールしてしまうのが手っ取り早いでしょう．自宅でインターネットにアクセスできる環境にある人は，http://www.cygwin.com/setup.exe (図 3 参照) をダウンロード，実行すればネットワークインストールできます．サイズがかなり大きいので日本のミラーサイトから落とすのが良いでしょう．

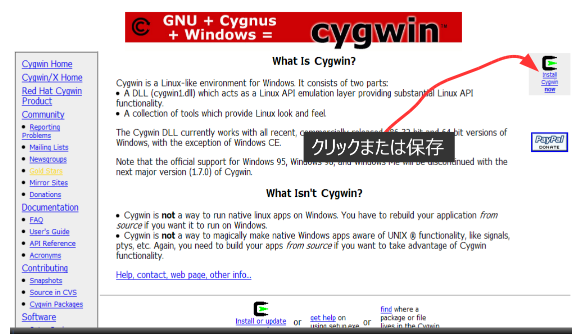

図 3: www.cygwin.com トップページ

setup.exe を実行して，Install from Internet→ Defult Text File Type を”Unix /binary”にする → プロクシなどは環境に合わせて適当に設定 → ミラーは http://ftp.jaist.ac.jp/あたりがおすすめ → devel カテゴリの gcc と make にチェックを入れる（gawk は勝手に入ると思われる）→ インストール終了まで待つ → 終了という流れで OK だと思います．gcc と make を入れないと意味がないので注意してください．なお，Cygwin のインストールが不可能な場合は USB ディスクなどを用意して，”cygwin”というフォルダを作り，端末室の PC 上の Cygwin 環境で作った”znc.exe”，”znas”，”p386.exe”ファイルを退避します．さらに端末室マシン上の Cygwin のインストールディレクトリをさがし，/bin と/lib 以下を退避します．こうすることで cygwin1.dll や bash.exe，make.exe などの必要プログラムを USB ディスク上に移してしまいます．これらのファイルを自分の PC に写し，cygwin/bin と cygwin/lib にパスを通し，cygwin/bin にある”bash.exe”を起動します．うまくいけば実験を続行できる可能性があります．その際，znas をテキストエディタで開き，冒頭の`#!/usr/bin/awk -f` の`/usr/bin` の部分を環境にあわせて書換える必要があります．

6. その他重要と思われる事項

本実験では，

http://www.tuat.ac.jp/~k1kaneko/gengo/append.html

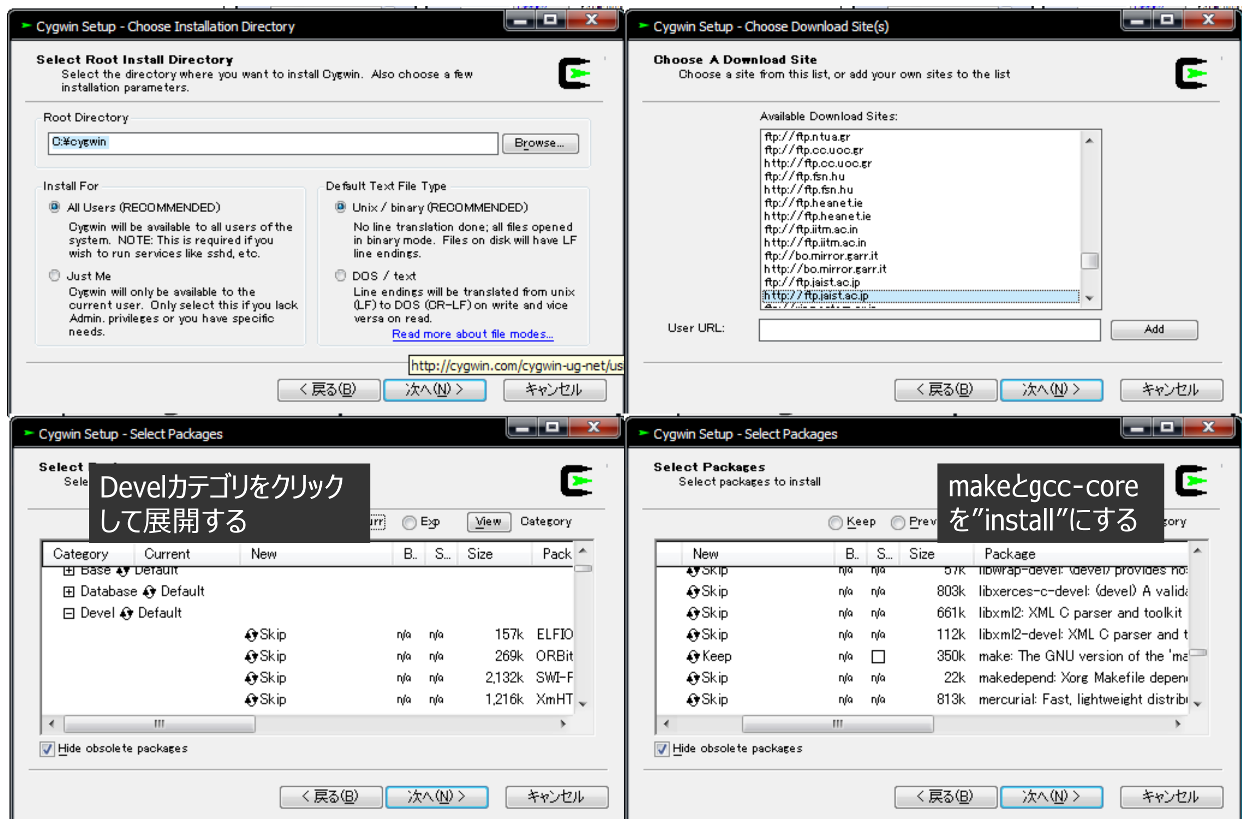

図 4: cygwin インストール手順

を読むことがひじょうに重要となってきます．ここには p386 やアセンブリ言語の仕様が書かれていますのでよく読まないと課題をこなすことができません．しかし，このページ自体結構なボリュームなので，重要なポイントをいくつか紹介します．

6.1 ZINC プログラム

毎年質問の多い，はまりやすいポイントをいくつか．

- ZINC コンパイラ znc はアセンブリコードを出力しますが，現在の仕様だと，ZINC 言語のプログラムの中で，必ず"_main"（アンダースコアが入っていることに注意）という手続きが必要です．
- アンダースコア（アンダーバー）をはずして"main"にしやすいので気をつけてください．
- 代入文を書くとき，set を忘れないようにしてください．
- 実験テキストの gcd.zn 中の putchar の引数の 45 とか 48 の意味がわからない人は，ascii コード表を検索してください．

6.2 覗穴的最適化

ここでは覗穴的最適化のイメージをつかむために，いくつか例を示します．

（注）これらはあくまでも「例」です．znc がここにあげたパターンのコードを生成するとは限りません．

冗長なデータ転送

```
movl $10, %eax
movl %eax, i
```

を見てください．この場合，わざわざ eax レジスタを介する意味はありません（eax に値をいれておかなければならない事情がなければ）．したがって，

```
movl $10, i
```

というふうに命令数を減らすことができます．このとき，テキストの表 1,2,3 をしっかりみて，movl という命令がオペランドにどんなアドレスをとることができるのかを調べなくてはなりません．今の場合だと，表 1 より，movl

imm，mem という形をしていることがわかります．また，表 3 より movl は番号 8 ですから，表 2 の番号 8 に該当する欄を探すと，imm，mem というパタンが許されていますので，これでよいということになります．

到達できないコード

```
...
jmp L.4
L.3:
...
L.4:
...
```

無条件分岐 jmp L.4 で L.3 を飛び越してしまうので，プログラムの他の場所に L.3 に分岐する命令がなければ L.3 は絶対に実行されませんから，L.3:から L.4 の直前までは除去することができます．

多重分岐　分岐命令でジャンプした先が，ふたたび分岐命令であるような場合を考えてみましょう．

```
...
jmp L.4
...
L.4:
jmp L.5
...
```

この場合，最初の jmp L.4 で L.4 に分岐してきても，結局 L.5 に飛ばされますから，

```
...
jmp L.5
...
L.4:
jmp L.5
...
```

とすることができます．

代数的単純化　ごく単純な例ですが，

```
addl $0, %eax
```

のように 0 を足すという演算命令は削除できる可能性があります．

演算の高速化　たとえば 2 の倍数で乗除算をおこなう場合，それらをシフト命令におきかえることにより，高速な演算が期待できます．

6.3　ZINC のデータ型

ZINC 言語において，データ型は ”word” の 1 種類だけである．386 のサブセットを ターゲットにしている現在のコンパイラでは，word は 32 bit の固定長の整数で，2 の 補数表現である．従って，表現可能な値の範囲は $-2^{31} \leqq x < 2^{31}$ である．(http://www.tuat.ac.jp/~k1kaneko/gengo/append.html より）

7.　Commentary on znc

ここからは znc コンパイラのソースコードに対して，簡単なコメントをつけていきます．「とにかく課題を早く終わらせたい」という人はとくに読む必要はないと思います．また，プログラムを書いた人とこの解説を書いている人は別人なので多分に誤った解釈もあると思います．解析の参考程度に眺めてください．

7.1　main.c

C で書かれたプログラムのエントリポイント (開始地点) は通常 main 関数なので，トップダウン方式で，まずはこの関数を含むファイルを見ていきましょう．

リスト 2: main.c

```
1  /*
2
3  Tiny ZiNC Compiler --- main
4
5
6  Copyright (C) 2000  KISHIMOTO, Makoto
7
8  This program is free software; you can redistribute it and/or modify
9  it under the terms of the GNU General Public License as published by
10 the Free Software Foundation; either version 2 of the License, or
11 (at your option) any later version.
12
13 This program is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU General Public License for more details.
17
18 You should have received a copy of the GNU General Public License
19 along with this program; if not, write to the Free Software
20 Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
21
22
23   KISHIMOTO, Makoto <cs96068@cs.tuat.ac.jp>
24
25 */
26
27 #include "token.h"
28 #include "parse.h"
29
30 static void init_all(void);
31
32 static void
33 init_all(void)
34 {
35   init_token();
36   init_parse();
37 }
38
39 int
40 main(void)
41 {
42   init_all();
43
44   parse();
45
46   exit(0);
47 }
```

39〜47 行目が main 関数です.

```
int
main(void)
{
  init_all();

  parse();

  exit(0);
}
```

init_all という関数を呼び出して parse() をよび，exit(0) で正常脱出 (終了) です．ちょっと拍子抜けしたかもしれないですが，main() は短いですね．さて，init という語は initialize の略で，プログラム中で見かけたらほぼ間違いなく初期化ルーチンです．parse(パーズ？パース？) というのは，ここでは構文解析のルーチンの

ことを指します．要するにこのプログラム (コンパイラ) は，大きく見て「初期化して，構文解析器 (字句解析・コード生成器は副次的に呼び出される) を起動する」という 2 つのことしかしていないことがわかります．そして 30 行目には init_all() の (プロトタイプ)「宣言」が，32〜37 行目には同じ init_all() の「定義」があります．

```
static void init_all(void);


static void
init_all(void)
{
  init_token();
  init_parse();
}
```

30 行目の init_all() のように，static を関数の宣言や定義の頭につけると，他のファイルからその関数を呼び出すことが出来なくなって，ある種の隠蔽をすることができます．java などとは違い，C では public とか private によってアクセス指定をすることが出来ないので，こういう書き方をすることが多いのです．ちょっと前置きが長くなりましたが，init_all() は内部でさらに 2 つの初期化ルーチン init_token() と init_parse() を呼び出しています．token はあまり日本語に訳さずにトークンとそのまま書かれることが多く，字句解析ルーチンで生成され，主に構文解析を行う際にその値が使われます．トークンはプログラム中の意味をなす最小単位を表します．字句解析ルーチンはプログラムを 1 文字づつ読み込み，意味をなす文字列の並びができたら，それをトークンという単位になおして (文法的な構成を調べる) 構文解析ルーチンに渡します．init_token と init_parse の定義を探しておきましょう．このファイル中には定義が見あたらないので 27,28 行目の

```
#include "token.h"
#include "parse.h"
```

でインクルードされる 2 つのファイル内を探せば良さそうです．あと，インクルードするファイルの指定方法ですが，たとえば stdio.h(標準入出力ライブラリ) のようなシステムに用意されているファイルを読むときは

```
#include <stdio.h>
```

のように'<' と'>' でファイル名を囲みます．通常，コンパイラにはインクルードパスやライブラリパスといったシステムヘッダやライブラリのありかを指定するようになっていますので，勝手に適当な場所を探してくれます．それに対してユーザが独自に定義したファイルなどは

```
#include "token.h"
```

のように'"'2 つで囲む方法をとります．

7.2 token.h

リスト 3: token.h

```
1  #if !defined _TOKEN_H_
2  #define _TOKEN_H_
3
4  #include "lang_dep.h"
5
6  typedef struct token {
7    int lex;  /* 字句値 : 1 バイトからなるトークン ( 名前を除く ) の */
8              /* 場合は文字コードが直接入る */
9    union {
10     void *p;
11     zinc_word w;
12   } sem;  /* 意味値 : 適当な情報へのポインタ */
13 } token;
14
15 /*
16 複数バイトからなるトークン ( 名前は 1 バイトの場合も含む ) などの字句値。
17 1 バイトからなるトークンと衝突する 0 ? 255 は空けておかねばならない。
```

```
18 */
19 typedef enum lexval {
20   LEXVAL_eof = 257,  /* End Of File */
21   LEXVAL_name,       /* 名前 (予約語除く) */
22   LEXVAL_literal,    /* (字句的に) リテラル */
23   LEXVAL_coloneq,    /* := */
24   LEXVAL_eqeq,       /* == */
25   LEXVAL_bangeq,     /* != */
26   LEXVAL_lt_or_eq,   /* <= */
27   LEXVAL_gt_or_eq,   /* >= */
28   LEXVAL_lsl,        /* << */
29   LEXVAL_asr,        /* >> */
30   LEXVAL_lsr,        /* >>> */
31
32   LEXVAL_IF = 513,  /* これより予約語 */
33   LEXVAL_WHILE,
34   LEXVAL_PROC,
35   LEXVAL_WORD,
36   LEXVAL_GETCHAR,
37   LEXVAL_PUTCHAR,
38   LEXVAL_SET,
39   LEXVAL_DEFPROC,
40   LEXVAL_CALL
41 } lexval;
42
43 void init_token(void);
44 token const get_token(lexval lv);
45 token const get_token_literal(zinc_word w);
46 token const get_token_p(lexval lv, void *p);
47
48 #endif /* _TOKEN_H_ */
```

token.h は 48 行です.

7.2.1 インクルードガード

まずこのファイル全体が，冒頭の 1,2 行目と最後の 48 行目の

```
#if !defined _TOKEN_H_
#define _TOKEN_H_
〜略〜
#endif /* _TOKEN_H_ */
```

という指令で囲まれているのに気づくと思います．これは「インクルードガード」という手法で,「いったん token.h をインクルードしたら，その後は token.h はインクルードされないようにする」という目的で使います．C プログラムはコンパイラにかける前に，プリプロセッサ (preprocessor) というプログラムに通され，ここでマクロ展開やインクルードファイルの展開などが行われます．プリプロセッサへの指令はおおざっぱにいって，#が行頭にくるものだと考えてください．C ではマクロも#define で使いますし，インクルードも#include というように#がついていますよね．そして，たとえば先の main.c 内の#include "token.h" という指令に対して，プリプロセッサは token.h の内容を展開します．しかし，その次の指令でインクルードする parse.h が token.h をインクルードしていたとしたらどうでしょう．以下のような状況になると思われます．まず，この定義

```
#include "token.h"
#include "parse.h"
```

の token.h の内容を展開して,

```
〜token.h の本体内容〜
#include "parse.h"
```

となり，次に parse.h を展開すると，parse.h では token.h をインクルードしていますから,

```
〜token.h の本体内容〜
〜parse.h の本体内容〜(token.h の内容もインクルードしているので二重定義になる!)〜
```

したがって，こういった状況になるのを避けるためにインクルードガードを使うわけです.

#if !defined _TOKEN_H_

は,「_TOKEN_H_が定義されていなければ」の意味で，条件が成立しなければ次に出現する

#endif

までを丸ごと無視するようになっています．最初に token.h が読み込まれるときは，当然_TOKEN_H_は定義されていませんから，最初の!defined _TOKEN_H_の値は 1 になります．なので次の行の

#define _TOKEN_H_

は無視せず読まれ， _TOKEN_H_が定義されることになります (これ以降 defined _TOKEN_H_は 1，!defined _TOKEN_H_は 0 になる)．次以降の token.h の読み込みでは，すでにプリプロセッサは!defined _TOKEN_H_の値が 0 になることを知っているので endif までを無視する（つまり token.h は展開されない）というわけです.

7.2.2 lang_dep.h, confdef.h

ここでひじょうに短いながら重要な役割をしている 2 つのヘッダファイルを見ていきましょう.

リスト 4: confdef.h

```
1  /*
2
3    zinc コンパイラを生成する環境に合わせて変更するべき設定
4
5  */
6
7  #if !defined _CONFDEF_H_
8  #define _CONFDEF_H_
9
10 #include <sys/types.h>
11 #include <stdio.h>
12
13 #define EMSTOP(mes, num) \
14 fprintf(stderr, "EMSTOP %s:%d " mes "(input line : %d)\n", \
15         __FILE__, __LINE__, num);
16
17 typedef int zinc_int32;
18 typedef unsigned int zinc_u_int32;
19
20 #endif /* _CONFDEF_H_ */
21 #include <stdio.h>
```

リスト 5: lang_dep.h

```
1  /*
2
3    zinc の処理系に依存する仕様に関する設定
4
5  */
6
7  #if !defined _LANG_DEP_H_
8  #define _LANG_DEP_H_
9
10 #include "confdef.h"
11
12 typedef zinc_int32 zinc_word;  /* word の型 */
13 typedef zinc_u_int32 zinc_u_word;  /* word の型の符号無しの型 */
14
15 #define ZINC_WORD_MAX 0x7fffffff  /* zinc_word のとりうる最大の値 */
16 #define ZINC_SIZEOF_WORD 4
17
18 #define ZINC_C_NAME_MAX_LEN 511  /* 名前の最大バイト数 */
19
20 #endif /* _LANG_DEP_H_ */
```

両ファイルともほぼコメント中に全貌が表れているのですが，ZINC 言語の唯一の型である zinc_word 型は zinc_int32 型であり，zinc_int32 型は int 型，つまり 32 ビット (4 バイト) であることを押さえてください．課題 1(2) のヒントにもなっていると思います．また，confdef.h 中の 13〜15 行目,

```
#define EMSTOP(mes, num) \
fprintf(stderr, "EMSTOP %s:%d " mes "(input line : %d)\n", \
        __FILE__, __LINE__, num);
```

のように，複数行にわたるマクロを書きたい場合は,

```
\
```

を行末に書いて区切ればいいです．ちなみに EMSTOP はコンパイルエラーの起こったファイルと行番号，エラーメッセージを表示してコンパイル作業を止める関数 (マクロです)．これは znc のソース中あちこちに出てきますから，いちおう頭に入れておいてください．

7.2.3 トークンの内部表現

さて，token.h に話を戻しましょう．トークンを生成するのは字句解析器です．図 5 を見てください．

図 5: 字句解析器のはたらき

字句解析器はプログラム中にたとえば"if"という文字の並びを見つけると，それに対応したトークンを生成して返します．そしてトークンの構造は 6〜13 行目で記述されています．

```
typedef struct token {
  int lex;  /* 字句値 : 1 バイトからなるトークン ( 名前を除く ) の */
            /* 場合は文字コードが直接入る */
  union {
    void *p;
    zinc_word w;
  } sem;  /* 意味値 : 適当な情報へのポインタ */
} token;
```

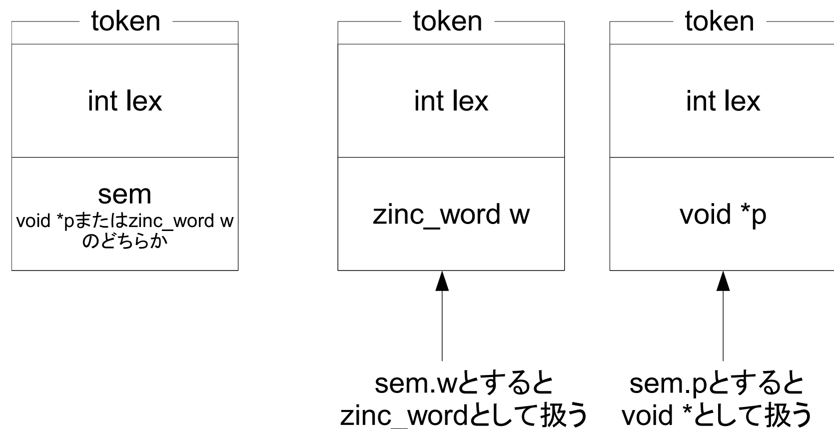

図 6: データ構造 token

この token データ構造について見ていきます．図 6 をご覧ください．token 型は int 型の lex というフィールドを持っていて，ここには各トークンの字句値 (構文解析器はこの「字句値」という単位にもとづいて解析を行います) が入ります．また，もう 1 つのフィールドとして union(共用体) によって表現された sem を持ちます．union にはなじみのない人もいるかもしれません．仮に sem が union ではなく struct で宣言されていたとしたら，sem には 8 バイトの領域が用意されます．しかし，union の場合，struct と異なりメンバ全体の合計が確保されるのではなく，union のメンバとして宣言されたもののうち，最大のサイズにあわせて sem フィールドがとられます．今の場合だと zinc_word 型 (4 バイト) と void *型 (4 バイト) は両方 4 バイトなので，sem のためには 4 バイトが確保されるわけです．実際に token 型の内容にアクセスする場合には，sem フィールドを zinc_word として扱いたいときは sem.w とすればいいですし，void *として扱いたいときは sem.p とすればよいです．union をつかうと記憶領域が少なくてすむ利点があります．今の場合「token データ構造内で，意味値 sem はポインタか zinc_word であり，両方が同時に使われることがない」とわかっているので union を使った方がいいわけです．

また,「何で字句値を返すだけじゃだめなのか？」という疑問もあるかと思います．これは後ほど出てくる構文解析に関連しています．たとえば x と y という変数があった場合，字句解析器は両方に対して同じ値を token.lex に入れて返します (この時点で x とか y とかの「変数の名前」による区別はなくなります!)．つまり構文解析器は「変数」という構文的なくくりでトークンを受け取るので，構文解析中に「変数の名前を知りたい」とか「定数の値を知りたい」などの要求に応えるためには sem という意味値を用意してそこに情報を入れておく必要があるのです．ちなみに sem.p は文字列へのポインタとして使われますし，定数ならば sem.w に値がそのまま入ります．

19〜41 行目では字句値が宣言されています．enum(列挙体) の開始値は 257 と 513 ですから，LEXVAL_name の値は 258 ですし，LEXVAL_PROC の値は 515 です．

```
typedef enum lexval {
  LEXVAL_eof = 257,  /* End Of File */
  LEXVAL_name,       /* 名前 ( 予約語除く ) */
  LEXVAL_literal,    /* ( 字句的に ) リテラル */
  LEXVAL_coloneq,    /* := */
  LEXVAL_eqeq,       /* == */
  LEXVAL_bangeq,     /* != */
  LEXVAL_lt_or_eq,   /* <= */
  LEXVAL_gt_or_eq,   /* >= */
  LEXVAL_lsl,        /* << */
  LEXVAL_asr,        /* >> */
  LEXVAL_lsr,        /* >>> */

  LEXVAL_IF = 513,  /* これより予約語 */
  LEXVAL_WHILE,
  LEXVAL_PROC,
  LEXVAL_WORD,
  LEXVAL_GETCHAR,
  LEXVAL_PUTCHAR,
  LEXVAL_SET,
  LEXVAL_DEFPROC,
  LEXVAL_CALL
} lexval;
```

7.3 token.c

token.c はいままでの内容が把握できていれば比較的わかりやすい内容です．

リスト 6: token.c

```
1 #include "confdef.h"
2
3 #include "token.h"
```

```
4
5  /* 1 バイトのトークン */
6  static token one_byte_token[256];
7
8  /* その他のスタティックなトークン */
9  static token const token_eof        = {LEXVAL_eof,        {0}};
10 static token const token_coloneq    = {LEXVAL_coloneq,    {0}};
11 static token const token_eqeq       = {LEXVAL_eqeq,       {0}};
12 static token const token_bangeq     = {LEXVAL_bangeq,     {0}};
13 static token const token_lt_or_eq   = {LEXVAL_lt_or_eq,   {0}};
14 static token const token_gt_or_eq   = {LEXVAL_gt_or_eq,   {0}};
15 static token const token_lsl        = {LEXVAL_lsl,        {0}};
16 static token const token_lsr        = {LEXVAL_lsr,        {0}};
17 static token const token_asr        = {LEXVAL_asr,        {0}};
18
19 static token const token_IF         = {LEXVAL_IF,         {0}};
20 static token const token_WHILE      = {LEXVAL_WHILE,      {0}};
21 static token const token_PROC       = {LEXVAL_PROC,       {0}};
22 static token const token_WORD       = {LEXVAL_WORD,       {0}};
23 static token const token_GETCHAR    = {LEXVAL_GETCHAR,    {0}};
24 static token const token_PUTCHAR    = {LEXVAL_PUTCHAR,    {0}};
25 static token const token_SET        = {LEXVAL_SET,        {0}};
26 static token const token_DEFPROC    = {LEXVAL_DEFPROC,    {0}};
27 static token const token_CALL       = {LEXVAL_CALL,       {0}};
28
29 /* 初期化 */
30 void
31 init_token(void)
32 {
33   int i;
34
35   for (i = 0; i < 256; i++)
36     {
37       one_byte_token[i].lex = i;
38       one_byte_token[i].sem.w = 0;
39     }
40 }
41
42 token const
43 get_token(lexval lv)
44 {
45   if (lv < 256)
46     {
47       return one_byte_token[lv];
48     }
49   else
50     {
51       switch (lv)
52         {
53         case LEXVAL_eof:
54           return token_eof;
55         case LEXVAL_coloneq:
56           return token_coloneq;
57         case LEXVAL_eqeq:
58           return token_eqeq;
59         case LEXVAL_bangeq:
60           return token_bangeq;
61         case LEXVAL_lt_or_eq:
62           return token_lt_or_eq;
63         case LEXVAL_gt_or_eq:
64           return token_gt_or_eq;
65         case LEXVAL_lsl:
66           return token_lsl;
67         case LEXVAL_lsr:
68           return token_lsr;
69         case LEXVAL_asr:
70           return token_asr;
```

```
71
72          case LEXVAL_IF:
73            return token_IF;
74          case LEXVAL_WHILE:
75            return token_WHILE;
76          case LEXVAL_PROC:
77            return token_PROC;
78          case LEXVAL_WORD:
79            return token_WORD;
80          case LEXVAL_GETCHAR:
81            return token_GETCHAR;
82          case LEXVAL_PUTCHAR:
83            return token_PUTCHAR;
84          case LEXVAL_SET:
85            return token_SET;
86          case LEXVAL_DEFPROC:
87            return token_DEFPROC;
88          case LEXVAL_CALL:
89            return token_CALL;
90
91          default:
92            EMSTOP("unknown token", 0);
93            exit(1);
94          }
95      }
96  }
97
98  token const
99  get_token_literal(zinc_word w)
100 {
101   token newtoken;
102
103   newtoken.lex = LEXVAL_literal;
104   newtoken.sem.w = w;
105
106   return newtoken;
107 }
108
109 token const
110 get_token_p(lexval lv, void *p)
111 {
112   token newtoken;
113
114   newtoken.lex = lv;
115   newtoken.sem.p = p;
116
117   return newtoken;
118 }
```

9〜27 行目で予約語の token 構造体を定義しています．eof の場合を例にとると，LEXVAL_eof という「字句値」と 0 という「意味値」をもつことになります．ここにならんだものに関しては，意味値はとくに必要とされないので該当フィールドは 0 で初期化されています．

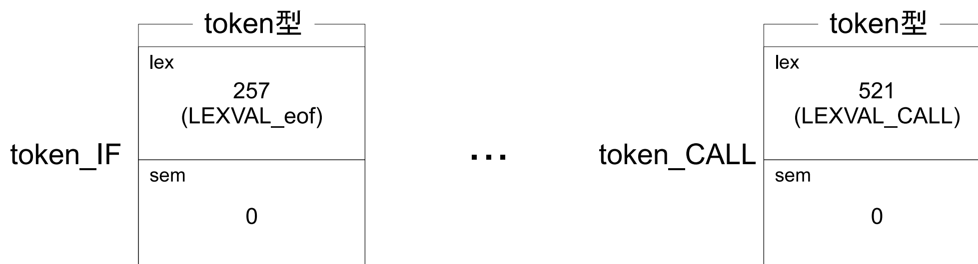

図 7: スタティックなトークン

29〜40 行目は init_token 関数です．main 関数内で最初に呼び出される init_all() 内でこれまた最初に呼び出される init_token() にようやくたどり着きました．今回のようにトップダウンで見ていくやり方だと，途中の変数名などを覚えておくのが結構大変だったと思います．本解説ではこのやり方を続けますが,「ひらメソッド」などで検索するといろいろソースコードの読み方の例が出てくると思います．これは結構奥深いテーマなので 1 度調べてみるのも面白いかもしれません．さて，init_token() の本体では，1 文字で決まるトークン用に字句値をセットしているだけです．意味値も一応 0 で初期化しています．42〜96 行目にある get_token は字句解析器で検出した各字句について，構文解析器が受け付けてくれるようなデータ構造を返します．ここで返すデータ構造とはさっき定義したスタティックなトークンです．ちなみに，1 文字で決まるようなものには

’%’,’&’,’(’,’)’,’*’,’+’,’,’,’-’,’;’,’{’,’|’,’}’,’~’

があります．98〜107 行目の get_token_literal と 109〜118 行目の get_token_p は，字句値をセットして token 型のトークンを返すという点は get_token と同じですが,「意味値があとで必要な場合」に使う関数です．具体的には前者は数値リテラル，後者は名前（変数名を表す文字列へのポインタ）を sem フィールドにセットして token 構造体を返します.

7.4 lex.h, lex.c

ここまでで，main 関数からはじまり，init_all→init_token とトップダウンでたどってきた流れは最後まで眺めたことになります．次はもう 1 つの流れである init_all→init_parse をたどっていきたいと思います. init_parse 関数の宣言は parse.h に，定義本体は parse.c にあるのですが，parse.c はこのコンパイラの本丸とも言えるべきファイルなので，init_parse が登場するところだけをざっと見て，先に字句解析ルーチンである lex.h，lex.c を見ていくことにします．では，parse.h と parse.c の冒頭部を見ていきましょう.

7.4.1 static 関数

リスト 7: parse.h

```
1  #if !defined _PARSE_H_
2  #define _PARSE_H_
3
4  void init_parse(void);
5  void parse(void);
6
7  #endif /* _PARSE_H_ */
```

実質 2 つの関数の宣言だけですね．これはかんたんに読めるかも？と思って期待すると …

リスト 8: parse.c(の冒頭部)

```
1  #include <stddef.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  #include "confdef.h"
6  #include "lang_dep.h"
7
8  #include "token.h"
9  #include "lex.h"
10 #include "name.h"
11 #include "express.h"
12 #include "codegen.h"
13 #include "labelgen.h"
14
15 #include "parse.h"
16
17 static void read_next(void);
18
19 static void module(void);
20 static void decl_proc(void);
21 static void define_glo_var(void);
22 static void define_proc(void);
23 static void define_loc_var(void);
24
```

```
25 static void statement(void);
26 static void comp_statement(void);
27 static void if_statement(void);
28 static void while_statement(void);
29 static void set_statement(void);
30 static void call_statement(void);
31 static void putchar_statement(void);
32
33 static exp_node *kakko_exp(void);
34 static exp_node *int_exp(void);
35 static exp_node *add_exp(void);
36 static exp_node *mul_exp(void);
37 static exp_node *unary_exp(void);
38 static exp_node *int_prim(void);
39
40 static token next;
41
42 static name_list *glo_name_list = 0;
43 static name_list *loc_name_list = 0;
44
45 void
46 init_parse(void)
47 {
48   init_lex(stdin);
49   init_codegen(stdout);
50 }
```

c ファイルのほうにはたくさん関数の宣言が書かれていました (17〜38 行目)．これらの関数宣言にはいずれも static 指定がついています．一方，ヘッダファイルにあった 2 つの関数 init_parse と parse には static はついていませんでした．一般に，インクルードされるファイルは*.h ファイルです (もちろん*.c だろうがファイル名を書けばインクルードはできてしまいます)．だから，parse.h の方は他のファイル上に展開される可能性があります (つまり，他のファイルから init_parse と parse は呼び出すことが出来る)．ところが，17〜38 行目のように static をつけて宣言するとその関数はそのファイル内でのみ使えることになります．*.c ファイルは基本的に他のファイルからインクルードされることはありませんから，他ファイルに対して関数実装を隠蔽できるというわけです．static をつけないと extern 宣言さえすればリンカが見つけてしまうので，このような手法をもちいてプログラムの堅牢性を高める工夫がなされています．

さて，ようやく肝心の init_parse ですが，中身はまたも初期化関数 init_lex と init_codegen ですね．init_lex の引数 stdin は「標準入力」を表します．今回の実験環境ではコンソール端末です．init_codegen の引数 stdout は「標準出力」で，これもコンソール端末です．なのでリダイレクトやパイプを使ってコンソール上からファイルを入力したり，逆に出力先をファイルにしたりできます．つぎに，init_lex の宣言と定義を探しに lex.h，lex.c を見ていきましょう．

リスト 9: lex.h

```
1  #if !defined _LEX_H_
2  #define _LEX_H_
3
4  #include "token.h"
5
6  void init_lex(FILE *fp);
7  token const lex(void);
8  int get_linenum(void);
9
10 #endif /* _LEX_H_ */
```

init_lex は字句解析のための初期化関数，lex は字句解析ルーチンの本体関数です．get_linenum は読み込み中ファイルの現在の行を返します．エラーメッセージなどを出す際などにつかいます．

お次は lex.c ですが，字句解析というコンパイラの核を担うファイルのひとつだけあって結構長いです (ほかに構文解析・コード生成が核です)．

リスト 10: lex.c

```
1  #include <stddef.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <ctype.h>
5
6  #include "confdef.h"
7  #include "lang_dep.h"
8
9  #include "token.h"
10
11 #include "lex.h"
12
13 /* 文字列と字句値の対応のための構造体 */
14 typedef struct words_tbl {
15   char const *str;
16   lexval const lexval;
17 } words_tbl;
18
19 static int get_character(void);
20 static void unget_character(int c);
21
22 static token const lexbang(int c);
23 static token const lexsl(int c);
24 static token const lexdec(int c);
25 static token const lexcolon(int c);
26 static token const lexlt(int c);
27 static token const lexeq(int c);
28 static token const lexgt(int c);
29 static token const lexname(int c);
30
31 static int line_count;
32 static FILE *in_file;
33
34 /* 予約語の文字列と字句値の対応、二分検索のため辞書順を保つこと */
35 static words_tbl const keywords_tbl[] = {
36   {"call",     LEXVAL_CALL},
37   {"defproc",  LEXVAL_DEFPROC},
38   {"getchar",  LEXVAL_GETCHAR},
39   {"if",       LEXVAL_IF},
40   {"proc",     LEXVAL_PROC},
41   {"putchar",  LEXVAL_PUTCHAR},
42   {"set",      LEXVAL_SET},
43   {"while",    LEXVAL_WHILE},
44   {"word",     LEXVAL_WORD}
45 };
46
47 void
48 init_lex(FILE *fp)
49 {
50   line_count = 1;
51   in_file = fp;
52 }
53
54 static int
55 get_character(void)
56 {
57   int c;
58
59   c = getc(in_file);
60
61   if (c == '\n')
62     line_count++;
63
64   c = ((c == EOF) ? (-1) : (c));
65
66   return c;
67 }
```

```
68
69  static void
70  unget_character(int c)
71  {
72    if (c == EOF)
73      return;
74
75    if (c == '\n')
76      line_count--;
77
78    ungetc(c, in_file);
79  }
80
81  int
82  get_linenum(void)
83  {
84    return line_count;
85  }
86
87  token const
88  lex(void)
89  {
90    int c;
91
92    for (;;)  /* トークンの 1 文字目 (の候補) を見る */
93      {
94        c = get_character();
95        switch (c)
96          {
97          /* End Of File */
98          case -1:
99            return get_token(LEXVAL_eof);
100
101         /* 空白は読み飛ばす */
102
103         case '\t':  /* FALLTHRU */  /* 「落ち抜け」を積極的に使う case */
104         case '\n':  /* FALLTHRU */
105         case '\f':  /* FALLTHRU */
106         case '\r':  /* FALLTHRU */
107         case '\x1a':  /* FALLTHRU */
108         case ' ':  /* FALLTHRU */
109           break;
110
111         /* 以下、キャラクタを分類 */
112         /* 使われないキャラクタについては case を書かずに default に落とす */
113         /* '_' をアルファベットの一種とみなすことと */
114         /* '/' の特殊な扱い ("//" によるコメントを読み飛ばす) に注意  */
115
116         /* 1 文字で決定する記号類 (ASCII 順) */
117         case '%':  /* FALLTHRU */
118         case '&':  /* FALLTHRU */
119         case '(':  /* FALLTHRU */
120         case ')':  /* FALLTHRU */
121         case '*':  /* FALLTHRU */
122         case '+':  /* FALLTHRU */
123         case ',':  /* FALLTHRU */
124         case '-':  /* FALLTHRU */
125         case ';':  /* FALLTHRU */
126         case '{':  /* FALLTHRU */
127         case '|':  /* FALLTHRU */
128         case '}':  /* FALLTHRU */
129         case '~':  /* FALLTHRU */
130           return get_token(c);
131
132         /* 複数文字の記号類 (ASCII 順) */
133         /* (c を渡しているが、ほぼ無意味である) */
134         case '!':
```

```
135             return lexbang(c);
136
137           case '/':
138             return lexsl(c);
139
140           case ':':
141             return lexcolon(c);
142
143           case '<':
144             return lexlt(c);
145
146           case '=':
147             return lexeq(c);
148
149           case '>':
150             return lexgt(c);
151
152           /* 残りはアルファベットと数字。さらにそれ以外ならエラー */
153           default:
154             if (isalpha(c) || (c == '_'))
155               {
156                 return lexname(c);
157               }
158             else if (c == '0')
159               {
160                 int next;
161                 next = get_character();
162                 if (isdigit(next))
163                   {
164                     EMSTOP("decimal literal cannot begin with \'0\' except for 0 itself", line_co
165                     exit(1);
166                   }
167                 else
168                   {
169                     unget_character(next);
170                     return lexdec(c);
171                   }
172               }
173             else if (isdigit(c))
174               {
175                 return lexdec(c);
176               }
177             else
178               {
179                 EMSTOP("unknown character", line_count);
180                 exit(1);
181               }
182           }
183       }
184   }
185
186   static token const
187   lexbang(int c)
188   {
189     int next;
190
191     next = get_character();
192     if (next == '=')
193       {
194         return get_token(LEXVAL_bangeq);
195       }
196     else
197       {
198         EMSTOP("unknown operator \'!\'", line_count);
199         exit(1);
200       }
201   }
```

```
202
203 static token const
204 lexsl(int c)
205 {
206   int next;
207
208   next = get_character();
209   if (next == '/')
210     {
211       do
212         next = get_character();
213       while (next != '\n');
214       return lex();
215     }
216   else
217     {
218       unget_character(next);
219       return get_token(c);
220     }
221 }
222
223 static token const
224 lexdec(int c)
225 {
226   char buf[2];
227   zinc_u_word d;
228
229   d = 0;
230
231   for (;;)
232     {
233       buf[0] = c;
234       buf[1] = '\0';
235       d += strtoul(buf, 0, 10);
236
237       c = get_character();
238
239       if (!isdigit(c))
240         {
241           unget_character(c);
242           break;
243         }
244       if (d > (ZINC_WORD_MAX / 5))  /* XXX この判定は厳密ではない */
245         {
246           fprintf(stderr, "lex.c : lexdec() : "
247                           "overflow (literal of decimal) in line %d\n", line_count);
248           exit(1);
249         }
250       d *= 10;
251     }
252
253   /* printf("%d\n", d); */
254
255   return get_token_literal(d);
256 }
257
258 static token const
259 lexcolon(int c)
260 {
261   int next;
262
263   next = get_character();
264   if (next == '=')
265     {
266       return get_token(LEXVAL_coloneq);
267     }
268   else
```

```
269        {
270          unget_character(next);
271          return get_token(c);
272        }
273  }
274
275  static token const
276  lexlt(int c)
277  {
278    int next;
279
280    next = get_character();
281    if (next == '=')
282      {
283        return get_token(LEXVAL_lt_or_eq);
284      }
285    else if (next == '<')
286      {
287        return get_token(LEXVAL_lsl);
288      }
289    else
290      {
291        unget_character(next);
292        return get_token(c);
293      }
294  }
295
296  static token const
297  lexeq(int c)
298  {
299    int next;
300
301    next = get_character();
302    if (next == '=')
303      {
304        return get_token(LEXVAL_eqeq);
305      }
306    else
307      {
308        EMSTOP("unknown operator \'=\'", line_count);
309        exit(1);
310      }
311  }
312
313  static token const
314  lexgt(int c)
315  {
316    int next;
317
318    next = get_character();
319    if (next == '=')
320      {
321        return get_token(LEXVAL_gt_or_eq);
322      }
323    else if (next == '>')
324      {
325        next = get_character();
326        if (next == '>')
327          {
328            return get_token(LEXVAL_lsr);
329          }
330        else
331          {
332            unget_character(next);
333            return get_token(LEXVAL_asr);
334          }
335      }
```

```
336     else
337       {
338         unget_character(next);
339         return get_token(c);
340       }
341 }
342
343 static token const
344 lexname(int c)
345 {
346   char buf[ZINC_C_NAME_MAX_LEN + 1];
347
348   /* 名前を全部読み込む */
349   {
350     int i;
351
352     i = 0;
353     do
354       {
355         buf[i++] = c;
356         if (i >= ZINC_C_NAME_MAX_LEN)
357           {
358             EMSTOP("too long name", line_count);
359             exit(1);
360           }
361         c = get_character();
362       }
363     while (isalnum(c) || (c == '_'));
364
365     buf[i] = '\0';
366     unget_character(c);
367   }
368
369   /* 予約語かどうかチェック */
370   {
371     int dn = (sizeof(keywords_tbl)) / (sizeof(keywords_tbl[0])) - 1;
372     int up = 0;  /* 2 分検索の上界と下界 */
373     int mid;
374     int r;
375
376     for(;;)
377       {
378         mid = (dn + up) / 2;
379         r = strcmp(buf, keywords_tbl[mid].str);
380
381         if (r == 0)  /* 予約語が見つかった */
382           return get_token(keywords_tbl[mid].lexval);
383         else if (dn <= up)  /* 見つからない */
384           {
385             char *name;
386
387             name = malloc(1 + strlen(buf));
388             strcpy(name, buf);
389             return get_token_p(LEXVAL_name, name);
390           }
391         else if (r < 0)
392           dn = mid - 1;
393         else if (r > 0)
394           up = mid + 1;
395       }
396
397     /* NOTREACHED */
398   }
399 }
```

というわけで，399 行あります．もしかするとこの長さのプログラムは実験でははじめて出てくるものかもしれません．この長さになってくると頭からひととおり眺めて一発で理解するというのは困難になってきます．そこ

で 22〜29 行目の関数宣言がすべて static であり，かつ名前に lex がついている点に注目しましょう．すなわち，この関数は外部ファイルから呼び出されることはなく，lex.c 内で定義されている関数の下請けとして使われている可能性が高いということと，その関数とは名前から lex() であることが推測できます．実際，lex 以下には，図 8 に示すような呼出し関係があります．

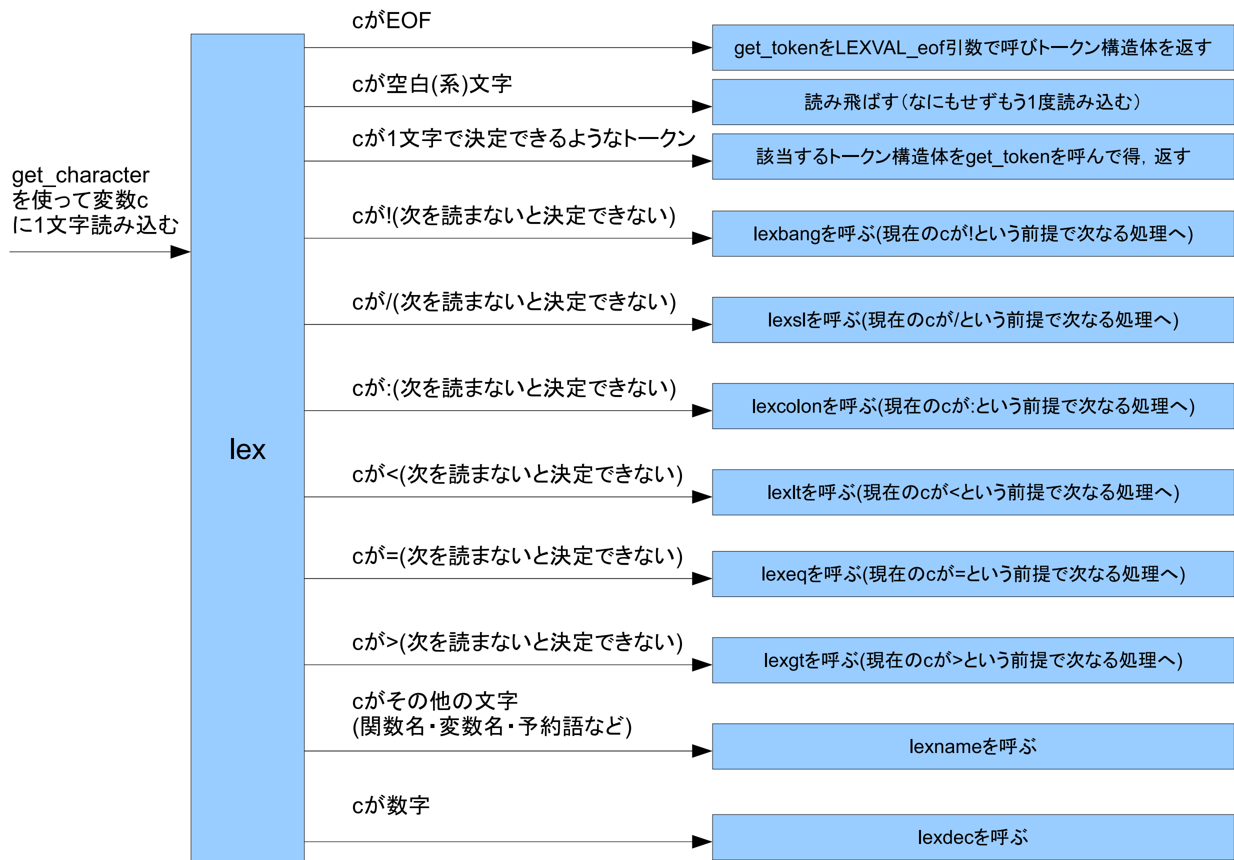

図 8: 字句解析器 lex の構造

多少は構造をつかむ助けになったでしょうか？これら lex 関連をすべて lex() の下請けと考えてしまえば，

- init_lex()
- get_character()
- unget_character()
- get_linenum()
- lex()

の 5 つの関数だけとなり，見通しがよくなると思います．それでは，これら 5 関数の本体に入る前に，データ構造を見ていきます．14〜17 行目では

```
typedef struct words_tbl {
  char const *str;
  lexval const lexval;
} words_tbl;
```

が定義されています．コメントによれば，/* 文字列と字句値の対応のための構造体 */とのことです．重要なことですが，この words_tbl は予約語にしか使われません．つまり，通常の変数や手続き名は別の場所で管理されるということです．また，予約語を表すデータ構造は，35〜45 行目で定義されています．この表はプログラム中に出現する予約語と，その「字句値」とを対応させるはたらきをします．47〜52 行目 init_lex についてはとくに解説の必要もないと思いますが，

```
void
init_lex(FILE *fp)
```

```
{
  line_count = 1;
  in_file = fp;
}
```

lex(字句解析) を行う前に行数を 1 にセットし，入力ファイルにファイルポインタをセットします．この関数は parse.c 内で init_lex(stdin); という形で呼び出されていた事を思い出してください．標準入力にポインタがあっていますので，fopen してファイルをオープンする必要がなくなっています．続いて 1 文字読み込む関数 get_character です．

```
static int
get_character(void)
{
  int c;

  c = getc(in_file);

  if (c == '\n')
    line_count++;

  c = ((c == EOF) ? (-1) : (c));

  return c;
}
```

これも特別難しいことをやっているわけではありません．getc(in_file) という呼出しで，で in_file(ここではつまり標準入力) から 1 文字読むことが出来ます．getc をつかえば勝手にストリームポインタを移動してくれますので標準入力内のバッファがどうなっているのかなどは気にする必要がありません．そして，読んだ文字が改行文字ならば行数をインクリメントします．また，

```
c = ((c == EOF) ? (-1) : (c));
```

という表現は，

```
if (c == EOF) {
  c = -1;
} else {
  c = c;
}
```

と同じ意味です．つまり EOF(ファイルの終端) が来たら c に-1 を代入しなさい，そうでなければそのまま c に c の値を代入しなさい (この場合ちょっと妙な表現になりますが)，ということです．

```
static void
unget_character(int c)
{
  if (c == EOF)
    return;

  if (c == '\n')
    line_count--;

  ungetc(c, in_file);
}
```

この unget_character は c に読み込んだ値を，in_file に読み戻すはたらきをします.「読み戻す」というのはちょっとピンと来ないかもしれないので例を挙げると，先ほどの図 8 で見たように，1 文字読んだだけではで決まらないトークンというのがいくつかありました．たとえば，'>' という文字を読んだとして，この時点では以下の可能性があります.

1. 次に'=' が来て，>=というトークンであることが判明する
2. 次に'>' が来て，>>というトークンであることが判明する
3. 次に来るのは'=''>' 以外で，>というトークンであることが判明する

ここで，最後の 3. の場合，最初の'>' を読み込んだあとに次が'=','>' 以外であると判定するためにその先の文字を読み込んでしまっています．したがって，読み戻さないと次の lex() 呼出しにおいてさらに次の文字が読み込まれてしまうことになります．このような場合に対処するために，unget_character 関数も不可欠なものになります．読み戻しには実質的には ungetc 関数を用います．C の場合には標準ライブラリに ungetc があるので問題ないのですが，Pascal などの言語で実装する場合は別の方法を模索しなければならなくなります (別に難しいというわけではありません).

```
int
get_linenum(void)
{
  return line_count;
}
```

この関数は先ほど述べたように現在の読み込み中の行番号を返すだけです．static 変数 line_count に直接他のファイルからアクセスさせるのではなく，必ず get_linenum を使ってアクセスさせるようになっています (get_linenum は static 関数でないことにご注意ください).

それではいよいよ lex() 本体を見ていきましょう．まず全体が，

```
for(;;) {
～略～
}
```

という無限ループの形をしていますね．つまり基本は get_character() を呼び出して変数 c に値を入れてトークンの種類が判別できるまで次々に読み込んでいきます．判別が成功し，結果に応じたトークンを返したら抜ける，という流れです．98,99 行目の

```
case -1:
  return get_token(LEXVAL_eof);
```

は読み込んだ文字が EOF(ファイル終端，値は 0) のときです．このときは get_token(LEXVAL_eof); によって EOF トークンを表す token 構造体を返します.

7.4.2 C における switch, case と break

C の switch 文はちょっとはまりやすいポイントがあります．それは case 項を実行し終わった時点で，自動的に switch 文から脱出するようにはなっていないということです．たとえば以下のようなコード

```
switch(2) {
  case 1:  printf("case 1 \n");
  case 2:  printf("case 2 \n");
  case 3:  printf("case 3 \n");
  case 4:  printf("case 4 \n");
  default:  printf("case default \n");
}
```

を実行すると，出力は,

```
  case 2
  case 3
  case 4
  default
```

となります．予想と違っていた人もいるのではないでしょうか．もし case 2 だけを表示したいならば，

```
case 2:  printf("case 2 \n"); break;
```

のように break; をつけなければなりません．この動作は仕様ミスであるともいわれていますが，fall through(下まで落ちる）といわれています．なので break; のついていない case 項に関しては警告を出すコンパイラもあります．こうした C の仕様を頭に入れたうえで 103〜109 行目を読んでみましょう．

```
case '\t':  /* FALLTHRU */  /* 「落ち抜け」を積極的に使う case */
case '\n':  /* FALLTHRU */
case '\f':  /* FALLTHRU */
case '\r':  /* FALLTHRU */
case '\x1a':  /* FALLTHRU */
case ' ':  /* FALLTHRU */
break;
```

まことにややこしいですが，ここでは落ち抜けを積極的に使うと書いてあります．ですが，言いたいこと自体は単純で，

```
case '\t':  break;
case '\n':  break;
case '\f':  break;
case '\r':  break;
case '\x1a': break;
case ' ':  break;
```

と書くより短いしすっきりする，という程度のものです．空白文字に対してはやることが特にないので，処理を一番下でまとめてもよい，というわけです．続いて 117〜130 行目も同様に落ち抜けを使っています．

```
case '%':  /* FALLTHRU */
case '&':  /* FALLTHRU */
case '(':  /* FALLTHRU */
case ')':  /* FALLTHRU */
case '*':  /* FALLTHRU */
case '+':  /* FALLTHRU */
case ',':  /* FALLTHRU */
case '-':  /* FALLTHRU */
case ';':  /* FALLTHRU */
case '{':  /* FALLTHRU */
case '|':  /* FALLTHRU */
case '}':  /* FALLTHRU */
case '~':  /* FALLTHRU */
  return get_token(c);
```

さっきの落ち抜け利用と違うのは，return get_token(c); で token 型構造体を返す必要があるところです．しかしながら，1 文字で判断できる記号の場合，現在 c に入っている ascii 値と「字句値」が同じなので，get_token(c) を呼び出すことで適当なトークン構造体を得，返すことができます．この処理は，1 文字で判断できる記号に共通なのでやはり落ち抜けを使って処理をまとめています．

ところでコメント中の FALLTHRU が気になったかもしれません．実はこのコメントは lint という C プログラムを検査するプログラムに「私，落ち抜けを意識的に使っているから警告出さないで」という指示を出すものです．ちょっとマニアックになってくるのでこの話はここまでにします．また，EOF も 1 文字で判断できるにもかかわらず，特別扱いしたのは，EOF のを読み込んだ時点で c の値は-1 ですが，トークンの字句値としては 257 を割り当てなければならないからです．

この先は 1 文字で判断できない記号の扱いになります．図 8 にあるような lex*という副次的な関数群が登場してきます．134〜181 行目が該当します．

```
case '!':
  return lexbang(c);

case '/':
  return lexsl(c);

case ':':
  return lexcolon(c);

case '<':
  return lexlt(c);

case '=':
  return lexeq(c);

case '>':
  return lexgt(c);

/* 残りはアルファベットと数字。さらにそれ以外ならエラー */
default:
  if (isalpha(c) || (c == '_'))
    {
      return lexname(c);
    }
  else if (c == '0')
    {
      int next;
      next = get_character();
      if (isdigit(next))
        {
          EMSTOP("decimal literal cannot begin with \'0\' except for 0 itself", line_count);
          exit(1);
        }
      else
        {
          unget_character(next);
          return lexdec(c);
        }
    }
  else if (isdigit(c))
    {
      return lexdec(c);
```

```
            }
          else
            {
              EMSTOP("unknown character", line_count);
              exit(1);
            }
```

133 行目のコメントに/* (c を渡しているが、ほぼ無意味である) */というのは，該当する lex*関数を呼び出す時点でその c が何であるかは自明だからです．1 文字目が'!' だった場合，lexbang を呼び出します (bang はエクスクラメーションマーク，ビックリマークのこと)．

```
static token const
lexbang(int c)
{
  int next;

  next = get_character();
  if (next == '=')
    {
      return get_token(LEXVAL_bangeq);
    }
  else
    {
      EMSTOP("unknown operator \'!\'", line_count);
      exit(1);
    }
}
```

このとき 2 文字目は必ず'=' となり，!=というトークンにならなければなりません (単独の'!' というトークンはありません)．そうでない場合はエラーなので行を表示して解析を中止します．

```
static token const
lexsl(int c)
{
  int next;

  next = get_character();
  if (next == '/')
    {
      do
        next = get_character();
      while (next != '\n');
      return lex();
    }
  else
    {
      unget_character(next);
      return get_token(c);
    }
}
```

1 文字目が'/' だった場合，lexsl() が呼び出されます (lex スラッシュ)．ZINC では//がコメントなので，まず 2 文字目も'/' が続くかをチェックし，そうであった場合は改行文字がくるまで，つぎの記号を読み続けます (こ

れにより行末までのコメントを処理する)．2 文字目が'/' でない場合は，読み戻して，その文字から解析を再開します (1 文字の'/' は割り算の演算子として扱われます)．

```
static token const
lexcolon(int c)
{
  int next;

  next = get_character();
  if (next == '=')
    {
      return get_token(LEXVAL_coloneq);
    }
  else
    {
      unget_character(next);
      return get_token(c);
    }
}
```

1 文字目が':' だった場合，lexcolon() が呼び出されます (lex コロン)．ZINC では:=が代入演算子を表すので 2 文字目が'=' なら字句値 LEXVAL_coloneq に対応したトークンを，2 文字目がそれ以外なら単なる':' なので読み戻して解析を続けます．

```
static token const
lexlt(int c)
{
  int next;

  next = get_character();
  if (next == '=')
    {
      return get_token(LEXVAL_lt_or_eq);
    }
  else if (next == '<')
    {
      return get_token(LEXVAL_lsl);
    }
  else
    {
      unget_character(next);
      return get_token(c);
    }
}
```

1 文字目が'<' だった場合，lexlt() が呼び出されます（lex less than）．ここでは 2 文字目にくる文字が'=' ならば比較演算子<=を表すトークンを返し，2 文字目が'<' ならばシフト演算子<<を表すトークンを返します．2 文字目がそれらのどちらでもなければ単なる'<' なので，読み戻して解析を続けます．

```
static token const
lexeq(int c)
{
```

```
  int next;

  next = get_character();
  if (next == '=')
    {
      return get_token(LEXVAL_eqeq);
    }
  else
    {
      EMSTOP("unknown operator \'=\'", line_count);
      exit(1);
    }
}
```

1 文字目が'=' だった場合，lexeq を呼び出します (lex equal)．このとき 2 文字目は必ず'=' となり，==というトークンにならなければなりません．意外かもしれませんが，ZINC には単独の'=' という演算子はないのです．代入は:=ですし，比較は=です．したがって 2 文字目が'=' 以外の場合はエラーなので行を表示して解析を中止します．

```
static token const
lexgt(int c)
{
  int next;

  next = get_character();
  if (next == '=')
    {
      return get_token(LEXVAL_gt_or_eq);
    }
  else if (next == '>')
    {
      next = get_character();
      if (next == '>')
        {
          return get_token(LEXVAL_lsr);
        }
      else
        {
          unget_character(next);
          return get_token(LEXVAL_asr);
        }
    }
  else
    {
      unget_character(next);
      return get_token(c);
    }
}
```

1 文字目が'>' だった場合，lexgt() が呼び出されます (lex greater than)．ZINC では>=が比較演算子を表すので 2 文字目が'=' なら字句値 LEXVAL_gt_or_eq に対応したトークンを，2 文字目が'>' であって 3 文字目が'>'

以外なら算術シフト演算子>>に対応したトークンを，1〜3 文字目が’>’ ならば論理シフト演算子>>>に対応したトークンを返します．この関数では 2 文字先読みする場合がありますが，’>’ が途中でこなかった場合は，もちろんこれまでと同様に読み戻して以降の解析を続けます.

```
static token const
lexdec(int c)
{
  char buf[2];
  zinc_u_word d;

  d = 0;

  for (;;)
    {
      buf[0] = c;
      buf[1] = '\0';
      d += strtoul(buf, 0, 10);

      c = get_character();

      if (!isdigit(c))
        {
          unget_character(c);
          break;
        }
      if (d > (ZINC_WORD_MAX / 5))  /* XXX この判定は厳密ではない */
        {
          fprintf(stderr, "lex.c : lexdec() : "
                          "overflow (literal of decimal) in line %d\n", line_count);
          exit(1);
        }
      d *= 10;
    }

  /* printf("%d\n", d); */

  return get_token_literal(d);
}
```

lexdec 関数は数値を解析するのに使います．基本的には引数として与えられる c の値，および 237 行目の `c = get_character();` で上位桁から 1 桁づつ読んで数値に変換していきます．まず配列 `buf[0]` に読み込んだ 1 文字を入れます．`buf[1]` には’\0’ を入れておきます．こうしないと strtoul が buf を文字列と見なしてくれません．srttoul は `string to unsigned long` と考えてください．この場合だと，buf で表される文字列を基数 10 で `unsigned long` 型に変換し，値を返します．ここで第 2 引数の 0 は変換できない文字列が出現した場合のみに必要となるので，ここでは無視してかまいません．あとは返却値を d に足し込んで，最後に 10 倍して桁をずらしながら文字列で表現された値を数値に変換します．また数値の場合はトークン構造体の中に「意味値」である数値を入れなければならないので，専用の `get_token_literal()` を呼びます.

```
token const
get_token_literal(zinc_word w)
{
```

```
  token newtoken;

  newtoken.lex = LEXVAL_literal;
  newtoken.sem.w = w;

  return newtoken;
}
```

ちゃんと newtoken.sem.w に数値が代入されているのがわかります．newtoken.lex は数値の場合，一律で LEXVAL_literal なので，構文解析フェーズに進んだときに数値の具体的な値を知るために必要なのでした．最後に，lexname を見ていきます．

```
static token const
lexname(int c)
{
  char buf[ZINC_C_NAME_MAX_LEN + 1];

  /* 名前を全部読み込む */
  {
    int i;

    i = 0;
    do
      {
        buf[i++] = c;
        if (i >= ZINC_C_NAME_MAX_LEN)
          {
            EMSTOP("too long name", line_count);
            exit(1);
          }
        c = get_character();
      }
    while (isalnum(c) || (c == '_'));

    buf[i] = '\0';
    unget_character(c);
  }
```

とりあえずここまでの処理を説明します．まず，先ほどと同じように読み込み中の文字列を保持するバッファ buf[ZINC_C_NAME_MAX_LEN + 1] に数値またはアルファベット (isalnum 関数で判断できる)，アンダースコアが続く限り読み込み，最後に'\0' を入れて文字列を完成させます．次に，この読み込んだ文字列 (名前) が予約語かどうか調べ，予約されていたらその名前のトークンを返し，予約されていない名前だったらその名前用に文字列領域を確保して，トークン化します．ちなみに予約語でない名前が複数回出現する場合は，そのたびに 383〜389 行目の処理は行われます (つまり複数 malloc されます)．

```
  /* 予約語かどうかチェック */
  {
    int dn = (sizeof(keywords_tbl)) / (sizeof(keywords_tbl[0])) - 1;
    int up = 0;  /* 2 分検索の上界と下界 */
    int mid;
    int r;
```

```
    for(;;)
      {
        mid = (dn + up) / 2;
        r = strcmp(buf, keywords_tbl[mid].str);

        if (r == 0)  /* 予約語が見つかった */
          return get_token(keywords_tbl[mid].lexval);
        else if (dn <= up)  /* 見つからない */
          {
            char *name;

            name = malloc(1 + strlen(buf));
            strcpy(name, buf);
            return get_token_p(LEXVAL_name, name);
          }
        else if (r < 0)
          dn = mid - 1;
        else if (r > 0)
          up = mid + 1;
      }

    /* NOTREACHED */
  }
```

予約語というのは，当たり前ですがコンパイル前にすべてわかっているので，並びを strcmp 的に（？）辞書順にしておけば，二分検索という高速なサーチ法が使えます．lex.c の 34〜45 行目で

```
/* 予約語の文字列と字句値の対応、二分検索のため辞書順を保つこと */
static words_tbl const keywords_tbl[] = {
  {"call",     LEXVAL_CALL},
  {"defproc",  LEXVAL_DEFPROC},
  {"getchar",  LEXVAL_GETCHAR},
  {"if",       LEXVAL_IF},
  {"proc",     LEXVAL_PROC},
  {"putchar",  LEXVAL_PUTCHAR},
  {"set",      LEXVAL_SET},
  {"while",    LEXVAL_WHILE},
  {"word",     LEXVAL_WORD}
};
```

のように，予約語テーブルを辞書順に並べたのはこの理由からです．いま，予約語の数はそんなに多くないので，「そんな大げさなことをしなくても線形検索で十分ではないか」と思われるかもしれませんが，名前が出現した場合はいつもこのルーチンを通ることが予想されるので，今後，予約語を増やすことをあらかじめ見越した設計にしておくのはよいことです．なにより，これは実習用・教育用のコンパイラなのでその辺が重要なのです．

7.5 codegen.h, codegen.c

ここまで読んでくださった方はだんだん解説が適当になってきていることにお気づきのはずです．実はもう実験開始まで数時間しかありません．これからさらにいいかげんになると思います．が，課題と関連が深いのはここからの内容なのです．ごめんなさい．まずは codegen.h です．コード生成器のヘッダファイルです．

リスト 11: codegen.h

```
1 #if !defined _CODEGEN_H_
```

```
#define _CODEGEN_H_

#include <stdio.h>

#include "express.h"
#include "name.h"

void init_codegen(FILE *fp);
void gen_verb(char s[]);
void gen_make_frame(int size);
void gen_exp(exp_node *p);
void gen_label(int n);
void gen_goto(int n);
void gen_jz(int n);
void gen_set(name_info *store_to);

#endif /* _CODEGEN_H_ */
```

まず関数宣言から見ていくと，init_codegen はもうおなじみ初期化関数ですね．残りは gen*で，これは generate という意味でしょう．後に続く語もコード生成ルーチンらしく，アセンブリ言語っぽいものが多いですね．また，express.h と name.h という２つのインクルードファイルがありますが，これらはそれぞれ「式関連」「名前処理関連」のデータ構造が宣言されています．この構造を理解しないとこの先がおぼつかないので，codegen.c より前に先に見ていくことにします．

リスト 12: express.h

```
#if !defined _EXPRESS_H_
#define _EXPRESS_H_

#include "lang_dep.h"
#include "name.h"

typedef struct exp_node_ exp_node;

typedef enum exp_type_ {
  EXP_ISEQ,
  EXP_ISNOTEQ,
  EXP_ISLT,
  EXP_ISGT,
  EXP_ISLTEQ,
  EXP_ISGTEQ,
  EXP_ADD,
  EXP_SUB,
  EXP_OR,
  EXP_MUL,
  EXP_DIV,
  EXP_MOD,
  EXP_LSL,
  EXP_ASR,
  EXP_LSR,
  EXP_AND,
  EXP_NOT,
  EXP_PLUS,
  EXP_MINUS,
  EXP_CONST,
  EXP_VAR
} exp_type;

struct exp_node_ {
  exp_type type;
  union {
    struct {
      exp_node *_0;
      exp_node *_1;
    } _2;
    struct {
```

```
41        exp_node *_0;
42      } _1;
43      zinc_word constval;
44      name_info *var;
45    } val;
46  };
47
48  #endif /* _EXPRESS_H_ */
```

33〜46 行目で宣言されている exp_node 構造体がすごく重要です．字句解析器は，文字を読み込んでトークン構造体を返しました．構文解析器が受け取るのは字句解析が出力したトークンですが，構文解析器が出力するのは exp_node 構造体で表される木構造なのです．また，7 行目の typedef は,「型 exp_node は型 struct exp_node_ である」という意味です (アンダースコアの有無に注意)．C では本来構造体を使うときには struct キーワードをつけなくてはなりません．ですが頻繁に出てくる型名だと struct をいちいち書くのは面倒です．そこで typedef しまうと，ノードを表す型を使いたいときには exp_node と書けばよいことになります．しかしながら struct を書かないということは，名前を見ただけでは構造体だとわからなくなるデメリットもあるので，場合によって使い分ける事が大事です．9〜31 行目は exp_type の宣言です．見た感じでは演算子系が多いですね.「式って何？」という疑問に文法記号を使わないで答えるのは難しいですが,「計算して値を求めることができるもの」とでも便宜的に理解しておけばいいかと思います．exp_node のメンバは,

- 型 exp_type で表される式の種類 type
- 共用体 val(以下のうちいずれか)
  - 子ノードを 2 つもつ場合 (_2)
  - 子ノードを 1 つもつ場合 (_1)
  - zinc_word 型の定数値 constval
  - name_info 型のへのポインタ var

です．たとえば式の種類 type が，EXP_ADD ならば 2 項の足し算なので，子ノードを 2 つもつと考えられます．EXP_CONST ならば定数 constval ですし，EXP_VAR なら変数なのでポインタ var を val メンバとしてもちます．

リスト 13: name.h

```
1   #if !defined _NAME_H_
2   #define _NAME_H_
3
4   typedef enum name_stat {
5     NAME_UNKNOWN,
6     NAME_DECLARED,
7     NAME_DEFINED
8   } name_stat;
9
10  typedef enum name_type {
11    NAME_PROC,
12    NAME_GLO_VAR,
13    NAME_LOC_VAR
14  } name_type;
15
16  typedef struct name_info_ name_info;
17  typedef struct name_list_ name_list;
18
19  name_info *name_find(name_list *p, char const s[]);
20  name_list *name_append(name_list *p, char *s);
21  void name_list_free(name_list *p);
22  void name_set_decl(name_info *p);
23  void name_set_def(name_info *p);
24  void name_set_proc(name_info *p);
25  void name_set_glo_var(name_info *p);
26  void name_set_loc_var(name_info *p);
27  name_stat name_get_stat(name_info const *p);
28  name_type name_get_type(name_info const *p);
29  char const *name_get_str(name_info const *p);
30  void name_set_id(name_info *p, int id);
31  int name_get_id(name_info const *p);
```

```
32
33 #endif /* _NAME_H_ */
```

つづいて名前関連の構造を見ていきましょう．name.h です．まず，name_stat というのは名前の状態で，

- NAME_DECLARED ... 宣言されている
- NAME_DEFINED ... 定義されている
- NAME_UNKNOWN ... 便宜的に用意されている (名前表を追加してから宣言・定義のいずれかの値を設定するまでの間しか使われない)

の 3 種です.「定義」は実際に領域が確保された名前です.「宣言」というのは手続きにしかありません．手続きの中で手続きを呼ぶ場合などに，手続きの本体を書かずに，その存在だけを知らせておくことができます．また，name_type は名前の種類を表していて，

- NAME_PROC ... 手続き
- NAME_GLO_VAR ... グローバル (大域) 変数
- NAME_LOC_VAR ... ローカル (局所) 変数

です．また，先ほどの木構造を表すノード構造体の時と同じように

```
typedef struct name_info_ name_info;
typedef struct name_list_ name_list;
```

という typedef により名前表を struct キーワードを書かずに使えるようにしています．name_info_，name_list_ の本体は name.c 内にあって，

```
struct name_info_ {
  char *str;
  name_stat stat;
  name_type type;
  int idnum;
};

struct name_list_ {
  name_info info;
  name_list *next;
};
```

です．1 つの名前に関する情報を保持するのが name_info_型で，

- char *str ... 綴りへのポインタ
- name_stat stat ... 名前の状態
- name_type type ... 名前の種類
- int idnum ... 何番目の名前か (アセンブリコードに落としたあとに，ベースポインタからオフセットでアクセスするために必要)

name.c の本体部も続けてみていきます.

リスト 14: name.c

```
1  #include <stddef.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  #include "name.h"
6
7  struct name_info_ {
8    char *str;
9    name_stat stat;
10   name_type type;
11   int idnum;
12 };
13
```

```
14 struct name_list_ {
15   name_info info;
16   name_list *next;
17 };
18
19 name_info *
20 name_find(name_list *p, char const s[])
21 {
22   name_list *index;
23
24   index = p;
25
26   while (index != 0)
27     {
28       if (strcmp(s, index->info.str) == 0)
29         break;
30
31       index = index->next;
32     }
33
34   return (index == 0) ? 0 : (&(index->info));
35 }
36
37 name_list *
38 name_append(name_list *p, char *s)
39 {
40   name_list *newname;
41
42   newname = malloc(sizeof(name_list));
43   newname->info.str = s;
44   newname->info.stat = NAME_UNKNOWN;
45   newname->info.idnum = 0;
46   newname->next = p;
47
48   return newname;
49 }
50
51 void
52 name_list_free(name_list *p)
53 {
54   name_list *next;
55
56   while (p != 0)
57     {
58       next = p->next;;
59       free(p->info.str);
60       free(p);
61       p = next;
62     }
63 }
64
65 void
66 name_set_decl(name_info *p)
67 {
68   p->stat = NAME_DECLARED;
69 }
70
71 void
72 name_set_def(name_info *p)
73 {
74   p->stat = NAME_DEFINED;
75 }
76
77 void
78 name_set_proc(name_info *p)
79 {
80   p->type = NAME_PROC;
```

```
81  }
82
83  void
84  name_set_glo_var(name_info *p)
85  {
86    p->type = NAME_GLO_VAR;
87  }
88
89  void
90  name_set_loc_var(name_info *p)
91  {
92    p->type = NAME_LOC_VAR;
93  }
94
95  name_stat
96  name_get_stat(name_info const *p)
97  {
98    return p->stat;
99  }
100
101 name_type
102 name_get_type(name_info const *p)
103 {
104   return p->type;
105 }
106
107 void
108 name_set_id(name_info *p, int id)
109 {
110   p->idnum = id;
111 }
112
113 int
114 name_get_id(name_info const *p)
115 {
116   return p->idnum;
117 }
118
119 char const *
120 name_get_str(name_info const *p)
121 {
122   return p->str;
123 }
```

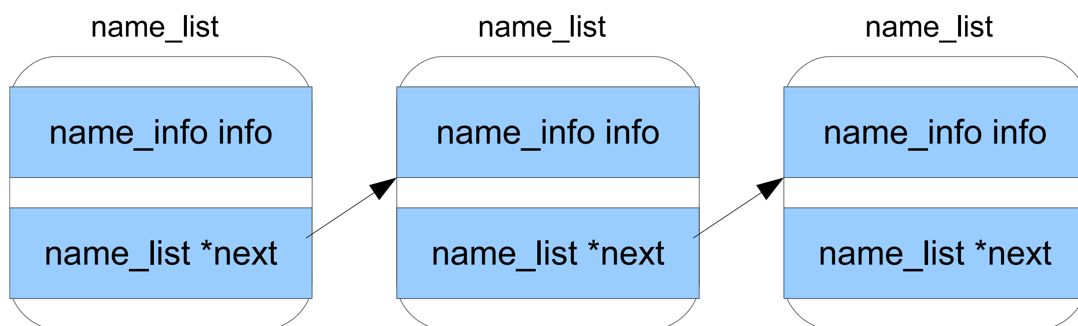

図 9: 名前表の構造

結局のところ，このファイルの関数は，図 9 のようなリスト構造で表される名前表を操作するものに過ぎないのです．リスト構造といえば，必要な (ありそうな) 関数は想像がつきます．すなわち，探索・要素追加・要素削除です．そしてこれらに相当するのが，

- `name_info *name_find(name_list *p, char const s[])`
  起点のポインタ p(リストの末尾) から開始して，名前 s をリストをたどって探し，見つかったら

name_info 型の情報を返す

- name_list *name_append(name_list *p, char *s)
  p が指しているリストの末尾に名前 s の項目を追加する（info.stat，info.idnum はあとで別関数で設定される）
- void name_list_free(name_list *p)
  p が指しているリストを解放する（ローカル変数は手続きを脱出すると必要なくなるので捨ててもよい）

のこりの name_set_*関数群は該当項目を name_info フィールドのメンバにセットするだけです.

リスト 15: codegen.c

```
1  /*
2
3  Tiny ZiNC Compiler --- code generater
4
5
6  Copyright (C) 2000  KISHIMOTO, Makoto
7
8  This program is free software; you can redistribute it and/or modify
9  it under the terms of the GNU General Public License as published by
10 the Free Software Foundation; either version 2 of the License, or
11 (at your option) any later version.
12
13 This program is distributed in the hope that it will be useful,
14 but WITHOUT ANY WARRANTY; without even the implied warranty of
15 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
16 GNU General Public License for more details.
17
18 You should have received a copy of the GNU General Public License
19 along with this program; if not, write to the Free Software
20 Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
21
22
23   KISHIMOTO, Makoto <cs96068@cs.tuat.ac.jp>
24
25 */
26
27 /*
28
29   コードジェネレータ
30
31 */
32
33 #include <stddef.h>
34 #include <stdlib.h>
35 #include <stdio.h>
36
37 #include "confdef.h"
38
39 #include "express.h"
40 #include "name.h"
41 #include "labelgen.h"
42
43 #include "codegen.h"
44
45 static void gen_add_grp(exp_node *p, char const op[]);
46 static void gen_cmp_grp(exp_node *p, char const op[]);
47
48 static FILE *out_file;
49
50 void
51 init_codegen(FILE *fp)
52 {
53   out_file = fp;
54 }
55
```

```
56  void
57  gen_verb(char s[])
58  {
59    fprintf(out_file, "%s", s);
60  }
61
62  void
63  gen_make_frame(int size)
64  {
65    fprintf(out_file, "\tsubl $%d, %%esp\n", size);
66  }
67
68  void
69  gen_exp(exp_node *p)
70  {
71    switch (p->type)
72      {
73      case EXP_ISEQ:
74        gen_cmp_grp(p, "jne");  /* 条件の反転に注意 */
75        break;
76      case EXP_ISNOTEQ:
77        gen_cmp_grp(p, "je");  /* 条件の反転に注意 */
78        break;
79      case EXP_ISLT:
80        gen_cmp_grp(p, "jnl");  /* 条件の反転に注意 */
81        break;
82      case EXP_ISGT:
83        gen_cmp_grp(p, "jng");  /* 条件の反転に注意 */
84        break;
85      case EXP_ISLTEQ:
86        gen_cmp_grp(p, "jnle");  /* 条件の反転に注意 */
87        break;
88      case EXP_ISGTEQ:
89        gen_cmp_grp(p, "jnge");  /* 条件の反転に注意 */
90        break;
91
92      case EXP_ADD:
93        gen_add_grp(p, "addl");
94        break;
95      case EXP_SUB:
96        gen_add_grp(p, "subl");
97        break;
98      case EXP_OR:
99        gen_add_grp(p, "orl");
100       break;
101
102     case EXP_MUL:
103       gen_exp(p->val._2._1);
104       fprintf(out_file, "\tpushl %%eax\n");
105       gen_exp(p->val._2._0);
106       fprintf(out_file, "\tpopl %%ecx\n");
107       fprintf(out_file, "\timull %%ecx, %%eax\n");
108       break;
109     case EXP_DIV:
110       gen_exp(p->val._2._1);
111       fprintf(out_file, "\tpushl %%eax\n");
112       gen_exp(p->val._2._0);
113       fprintf(out_file, "\tpopl %%ecx\n");
114       fprintf(out_file, "\tcltd\n");
115       fprintf(out_file, "\tidivl %%ecx, %%eax\n");
116       break;
117     case EXP_MOD:
118       gen_exp(p->val._2._1);
119       fprintf(out_file, "\tpushl %%eax\n");
120       gen_exp(p->val._2._0);
121       fprintf(out_file, "\tpopl %%ecx\n");
122       fprintf(out_file, "\tcltd\n");
```

```
123          fprintf(out_file, "\tidivl %%ecx, %%eax\n");
124          fprintf(out_file, "\tmovl %%edx, %%eax\n");
125          break;
126        case EXP_LSL:
127          gen_exp(p->val._2._1);
128          fprintf(out_file, "\tpushl %%eax\n");
129          gen_exp(p->val._2._0);
130          fprintf(out_file, "\tpopl %%ecx\n");
131          fprintf(out_file, "\tshll %%cl, %%eax\n");
132          break;
133        case EXP_ASR:
134          gen_exp(p->val._2._1);
135          fprintf(out_file, "\tpushl %%eax\n");
136          gen_exp(p->val._2._0);
137          fprintf(out_file, "\tpopl %%ecx\n");
138          fprintf(out_file, "\tsarl %%cl, %%eax\n");
139          break;
140        case EXP_LSR:
141          gen_exp(p->val._2._1);
142          fprintf(out_file, "\tpushl %%eax\n");
143          gen_exp(p->val._2._0);
144          fprintf(out_file, "\tpopl %%ecx\n");
145          fprintf(out_file, "\tshrl %%cl, %%eax\n");
146          break;
147
148        case EXP_AND:
149          gen_add_grp(p, "andl");
150          break;
151        case EXP_NOT:
152          gen_exp(p->val._1._0);
153          fprintf(out_file, "\tnotl %%eax\n");
154          break;
155        case EXP_PLUS:
156          gen_exp(p->val._1._0);
157          break;
158        case EXP_MINUS:
159          gen_exp(p->val._1._0);
160          fprintf(out_file, "\tnegl %%eax\n");
161          break;
162
163        case EXP_CONST:
164          fprintf(out_file, "\tmovl $%d, %%eax\n", p->val.constval);
165          break;
166        case EXP_VAR:
167          {
168            name_type var_scope;
169            var_scope = name_get_type(p->val.var);
170            if (var_scope == NAME_GLO_VAR)
171              {
172                fprintf(out_file, "\tmovl %s, %%eax\n", name_get_str(p->val.var));
173              }
174            else if (var_scope == NAME_LOC_VAR)
175              {
176                fprintf(out_file, "\tmovl -%d(%%ebp), %%eax\n", name_get_id(p->val.var));
177              }
178            else
179              {
180                EMSTOP("internal conflict", 0);
181                exit(1);
182              }
183            break;
184          }
185        }
186    }
187
188    static void
189    gen_add_grp(exp_node *p, char const op[])
```

```
190 {
191   gen_exp(p->val._2._1);
192   fprintf(out_file, "\tpushl %%eax\n");
193   gen_exp(p->val._2._0);
194   fprintf(out_file, "\tpopl %%ecx\n");
195   fprintf(out_file, "\t%s %%ecx, %%eax\n", op);
196 }
197
198 static void
199 gen_cmp_grp(exp_node *p, char const op[])
200 {
201   int label;
202
203   gen_add_grp(p, "cmpl");
204   fprintf(out_file, "\tmovl $0, %%eax\n");  /* 注 : ←ここを xorl %eax, %eax に */
205   label = getnewnum();                      /*        しては **いけない** */
206   fprintf(out_file, "\t%s L.%d\n", op, label);  /* なぜなら←ここで見ている */
207   fprintf(out_file, "\tnotl %%eax\n");  /* フラグを破壊してしまうからである */
208   gen_label(label);
209 }
210
211 void
212 gen_label(int n)
213 {
214   fprintf(out_file, "L.%d:\n", n);
215 }
216
217 void
218 gen_goto(int n)
219 {
220   fprintf(out_file, "\tjmp L.%d\n", n);
221 }
222
223 void
224 gen_jz(int n)
225 {
226   fprintf(out_file, "\tjz L.%d\n", n);
227 }
228
229 void
230 gen_set(name_info *store_to)
231 {
232   name_type var_scope;
233
234   var_scope = name_get_type(store_to);
235   if (var_scope == NAME_GLO_VAR)
236     {
237       fprintf(out_file, "\tmovl %%eax, %s\n", name_get_str(store_to));
238     }
239   else if (var_scope == NAME_LOC_VAR)
240     {
241       fprintf(out_file, "\tmovl %%eax, -%d(%%ebp)\n", name_get_id(store_to));
242     }
243   else
244     {
245       EMSTOP("internal conflict", 0);
246       exit(1);
247     }
248 }
```

codegen.c における最重要関数は，やはり最も長い gen_exp 関数です．基本は構文解析器の出力した構文木のノードへのポインタを受け取って，そのノードの種類によって処理を分けています．

```
static void gen_add_grp(exp_node *p, char const op[]);
static void gen_cmp_grp(exp_node *p, char const op[]);
```

は，前者は EXP_ADD，EXP_SUB，EXP_OR にノードに行う処理が演算子以外に違いがないため，add_grp として処理をまとめてしまい，引数の op で演算子だけ調整するためにあります．後者も同様で関係演算子のコード生成が，演算子の差異をのぞけば同一であるため，cmp_grp としてまとめています．その他の式（代入以外）は gen_exp 内に書かれた処理を行いますが，読み解くポイントは「gen_exp は式の計算を eax レジスタに保持する」という基本原則です．たとえば，かけ算の式 EXP_MUL の場合を考えてみましょう．

```
case EXP_MUL:
  gen_exp(p->val._2._1);
  fprintf(out_file, "\tpushl %%eax\n");
  gen_exp(p->val._2._0);
  fprintf(out_file, "\tpopl %%ecx\n");
  fprintf(out_file, "\timull %%ecx, %%eax\n");
    break;
```

この EXP_MUL の構造をごく単純化して表すと図 10 のようになります．まず gen_exp(p->val._2._1); で右側の

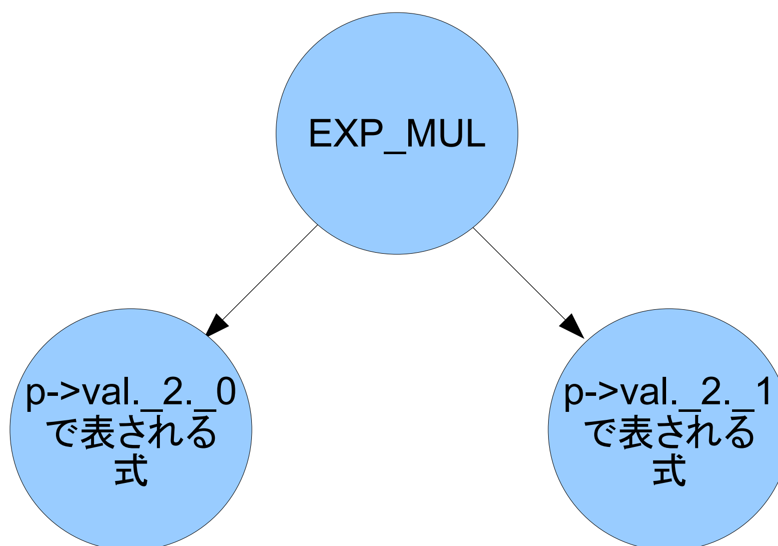

図 10: EXP_MUL の構造

ノード（さらに子供をもっているかもしれません）を計算するようなコードを生成します．すると結果はレジスタ eax に入っています．ですから次の

```
fprintf(out_file, "\tpushl %%eax\n");
```

で eax レジスタの内容をいったんスタックにプッシュして退避します．その次に gen_exp(p->val._2._0); を呼び出して左側のノードを計算するようなコードを生成します．結果はやはり eax レジスタに入っています．右側のノードを計算した値はスタックに退避してあるので，

```
fprintf(out_file, "\tpopl %%ecx\n");
```

で ecx レジスタに持ってきて，計算の準備を整えます．最後に

```
fprintf(out_file, "\timull %%ecx, %%eax\n");
```

でかけ算完了です．もちろん結果は eax レジスタに入っています．この流れがだいたいつかめれば，あとのコード生成ルーチンに関しても理解できるかと思われます．あとは代入のコードを処理する gen_set を見ておきましょう．

```
void
gen_set(name_info *store_to)
{
```

```
  name_type var_scope;

  var_scope = name_get_type(store_to);
  if (var_scope == NAME_GLO_VAR)
    {
      fprintf(out_file, "\tmovl %%eax, %s\n", name_get_str(store_to));
    }
  else if (var_scope == NAME_LOC_VAR)
    {
      fprintf(out_file, "\tmovl %%eax, -%d(%%ebp)\n", name_get_id(store_to));
    }
  else
    {
      EMSTOP("internal conflict", 0);
      exit(1);
    }
}
```

グローバル変数ローカル変数とで出力するコードが違います．ローカル変数はアセンブリ言語に直すときに，ebp からの相対位置でアクセスするようにします (手続きから抜けたら使わないので，ebp のずらしだけで領域確保・解放を行えるようにするため)．

7.6 parse.c

ここから znc コンパイラ最重要部である構文解析器 (パーザ) を実装している parse.c を見ていきます。

リスト 16: parse.c

```
1  #include <stddef.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  #include "confdef.h"
6  #include "lang_dep.h"
7
8  #include "token.h"
9  #include "lex.h"
10 #include "name.h"
11 #include "express.h"
12 #include "codegen.h"
13 #include "labelgen.h"
14
15 #include "parse.h"
16
17 static void read_next(void);
18
19 static void module(void);
20 static void decl_proc(void);
21 static void define_glo_var(void);
22 static void define_proc(void);
23 static void define_loc_var(void);
24
25 static void statement(void);
26 static void comp_statement(void);
27 static void if_statement(void);
28 static void while_statement(void);
29 static void set_statement(void);
30 static void call_statement(void);
31 static void putchar_statement(void);
32
33 static exp_node *kakko_exp(void);
34 static exp_node *int_exp(void);
```

```
35  static exp_node *add_exp(void);
36  static exp_node *mul_exp(void);
37  static exp_node *unary_exp(void);
38  static exp_node *int_prim(void);
39
40  static token next;
41
42  static name_list *glo_name_list = 0;
43  static name_list *loc_name_list = 0;
44
45  void
46  init_parse(void)
47  {
48    init_lex(stdin);
49    init_codegen(stdout);
50  }
51
52  void
53  parse(void)
54  {
55    read_next();  /* まず先読み */
56    module();  /* プログラム全体は 1 個のモジュール定義 */
57  }
58
59  static void
60  read_next(void)
61  {
62    next = lex();
63  }
64
65  static void
66  module(void)
67  {
68    gen_verb(".globl _main\n\n");
69    for (;;)
70      {
71        switch (next.lex)
72          {
73          case LEXVAL_PROC:
74            decl_proc();
75            break;
76          case LEXVAL_WORD:
77            define_glo_var();
78            break;
79          default:
80            goto end_decl_g;  /* for (;;) を抜ける */
81          }
82      }
83  end_decl_g:
84    while (next.lex == LEXVAL_DEFPROC)
85      define_proc();
86
87    if (next.lex != LEXVAL_eof)
88      {
89        EMSTOP("unknown token", get_linenum());
90        exit(1);
91      }
92  }
93
94  static void
95  decl_proc(void)
96  {
97    name_info *thisname;
98
99    do
100     {
101       read_next();
```

```
102       if (next.lex != LEXVAL_name)
103         {
104           EMSTOP("proc name must be name", get_linenum());
105           exit(1);
106         }
107       if (name_find(glo_name_list, (char *)next.sem.p) != 0)
108         {
109           EMSTOP("same (proc) name used twice", get_linenum());
110           exit(1);
111         }
112       glo_name_list = name_append(glo_name_list, (char *)next.sem.p);
113       thisname = name_find(glo_name_list, (char *)next.sem.p);
114       name_set_decl(thisname);
115       name_set_proc(thisname);
116       read_next();
117     }
118   while (next.lex == ',');
119   if (next.lex != ';')
120     {
121       EMSTOP("compiler expected \';\'", get_linenum());
122       exit(1);
123     }
124   read_next();
125 }
126
127 static void
128 define_glo_var(void)
129 {
130   name_info *thisname;
131
132   do
133     {
134       read_next();
135       if (next.lex != LEXVAL_name)
136         {
137           EMSTOP("global variable name must be name", get_linenum());
138           exit(1);
139         }
140       if (name_find(glo_name_list, (char *)next.sem.p) != 0)
141         {
142           EMSTOP("same (global variable) name used twice", get_linenum());
143           exit(1);
144         }
145       glo_name_list = name_append(glo_name_list, (char *)next.sem.p);
146       thisname = name_find(glo_name_list, (char *)next.sem.p);
147       name_set_def(thisname);
148       name_set_glo_var(thisname);
149       gen_verb(".lcomm ");
150       gen_verb((char *)next.sem.p);
151       gen_verb(", 4\n");
152       read_next();
153     }
154   while (next.lex == ',');
155   if (next.lex != ';')
156     {
157       EMSTOP("compiler expected \';\'", get_linenum());
158       exit(1);
159     }
160   read_next();
161 }
162
163 static void
164 define_proc(void)
165 {
166   name_info *thisname;
167
168   read_next();
```

```
169   if (next.lex != LEXVAL_name)
170     {
171       EMSTOP("proc name must be name", get_linenum());
172       exit(1);
173     }
174   if ((thisname = name_find(glo_name_list, (char *)next.sem.p)) == 0)
175     {
176       EMSTOP("proc name is not declared", get_linenum());
177       exit(1);
178     }
179   if (name_get_type(thisname) != NAME_PROC)
180     {
181       EMSTOP("name is not declared for procedure", get_linenum());
182       exit(1);
183     }
184   if (name_get_stat(thisname) == NAME_DEFINED)
185     {
186       EMSTOP("same name proc defined twice", get_linenum());
187       exit(1);
188     }
189   if (name_get_stat(thisname) != NAME_DECLARED)
190     {
191       EMSTOP("name is not declared", get_linenum());
192       exit(1);
193     }
194   name_set_def(thisname);
195   gen_verb("\n.text\n\t.align 2\n\t.type\t");
196   gen_verb((char *)next.sem.p);
197   gen_verb(", @function\n");
198   gen_verb((char *)next.sem.p);
199   gen_verb(":\n\tpushl %ebp\n\tmovl %esp, %ebp\n");
200   read_next();
201   if (next.lex != ':')
202     {
203       EMSTOP("compiler expected \':\'", get_linenum());
204       exit(1);
205     }
206   read_next();
207   define_loc_var();
208   comp_statement();
209   gen_verb("\tleave\n\tret\n");
210   name_list_free(loc_name_list);
211   loc_name_list = 0;
212 }
213
214 static void
215 define_loc_var(void)
216 {
217   int frame_size;
218   name_info *thisname;
219
220   frame_size = 0;
221   while (next.lex == LEXVAL_WORD)
222     {
223       do
224         {
225           read_next();
226           if (next.lex != LEXVAL_name)
227             {
228               EMSTOP("local variable name must be name", get_linenum());
229               exit(1);
230             }
231           if (name_find(loc_name_list, (char *)next.sem.p) != 0)
232             {
233               EMSTOP("same name local variable defined twice", get_linenum());
234               exit(1);
235             }
```

```
236             frame_size += ZINC_SIZEOF_WORD;
237             loc_name_list = name_append(loc_name_list, (char *)next.sem.p);
238             thisname = name_find(loc_name_list, (char *)next.sem.p);
239             name_set_def(thisname);
240             name_set_loc_var(thisname);
241             name_set_id(thisname, frame_size);
242             read_next();
243           }
244         while (next.lex == ',');
245         if (next.lex != ';')
246           {
247             EMSTOP("compiler expected \';\'", get_linenum());
248             exit(1);
249           }
250         read_next();
251       }
252     if (frame_size > 0)
253       gen_make_frame(frame_size);
254   }
255
256   static void
257   comp_statement(void)
258   {
259     if (next.lex != '{')  /* このコメントは } エディタの括弧対応のためのおまじない */
260       {
261         EMSTOP("compiler expected \'{\'", get_linenum());   /* } */
262         exit(1);
263       }
264     read_next();
265     do
266       statement();   /* { */
267     while (next.lex != '}');
268     read_next();
269   }
270
271   static void
272   statement(void)
273   {
274     switch (next.lex)
275       {
276       case LEXVAL_IF:
277         if_statement();
278         break;
279       case LEXVAL_WHILE:
280         while_statement();
281         break;
282       case LEXVAL_SET:
283         set_statement();
284         break;
285       case LEXVAL_CALL:
286         call_statement();
287         break;
288       case LEXVAL_PUTCHAR:
289         putchar_statement();
290         break;
291       default:
292         EMSTOP("compiler expected statement", get_linenum());
293         exit(1);
294       }
295   }
296
297   static void
298   if_statement(void)
299   {
300     int fi_label;
301
302     fi_label = getnewnum();
```

```
303     read_next();
304     gen_exp(kakko_exp());
305     gen_verb("\tandl %eax, %eax\n");
306     gen_jz(fi_label);
307     comp_statement();
308     gen_label(fi_label);
309   }
310
311   static void
312   while_statement(void)
313   {
314     int loop_label, quit_label;
315
316     loop_label = getnewnum();
317     quit_label = getnewnum();
318     read_next();
319     gen_label(loop_label);
320     gen_exp(kakko_exp());
321     gen_verb("\tandl %eax, %eax\n");
322     gen_jz(quit_label);
323     comp_statement();
324     gen_goto(loop_label);
325     gen_label(quit_label);
326   }
327
328   static void
329   set_statement(void)
330   {
331     name_info *store_to;
332
333     read_next();
334     if (next.lex != LEXVAL_name)
335       {
336         EMSTOP("left of \":=\" must be name", get_linenum());
337         exit(1);
338       }
339     store_to = name_find(loc_name_list, next.sem.p);
340     if (store_to == 0)
341       store_to = name_find(glo_name_list, next.sem.p);
342     if (store_to == 0)
343       {
344         EMSTOP("left of \":=\" is unknown name",  get_linenum());
345         exit(1);
346       }
347     if (name_get_type(store_to) == NAME_PROC)
348       {
349         EMSTOP("cannot assignment for procedure", get_linenum());
350         exit(1);
351       }
352     read_next();
353     if (next.lex != LEXVAL_coloneq)
354       {
355         EMSTOP("compiler expected \":=\"", get_linenum());
356         exit(1);
357       }
358     read_next();
359     if (next.lex == LEXVAL_GETCHAR)
360       {
361   #if defined ZINC_FreeBSD
362         gen_verb("\tpushl $___sF\n\tcall _fgetc\n\taddl $4, %esp\n");
363   #else /* ZINC_FreeBSD */
364         gen_verb("\tint $192\n");
365   #endif /* ZINC_FreeBSD */
366         read_next();
367       }
368     else
369       {
```

```
370        gen_exp(int_exp());
371      }
372    gen_set(store_to);
373    if (next.lex != ';')
374      {
375        EMSTOP("compiler expected \';\'", get_linenum());
376        exit(1);
377      }
378    read_next();
379  }
380
381  static void
382  call_statement(void)
383  {
384    read_next();
385    if (next.lex != LEXVAL_name)
386      {
387        EMSTOP("to call is must be name", get_linenum());
388        exit(1);
389      }
390    gen_verb("\tcall ");
391    gen_verb(next.sem.p);
392    gen_verb("\n");
393    read_next();
394    if (next.lex != ';')
395      {
396        EMSTOP("compiler expected \';\'", get_linenum());
397        exit(1);
398      }
399    read_next();
400  }
401
402  static void
403  putchar_statement(void)
404  {
405    read_next();
406    gen_exp(int_exp());
407  #if defined ZINC_FreeBSD
408    gen_verb("\tpushl $___sF+88\n\tpushl %eax\n\tcall _fputc\n\taddl $8, %esp\n");
409  #else /* ZINC_FreeBSD */
410    gen_verb("\tint $193\n");
411  #endif /* ZINC_FreeBSD */
412    if (next.lex != ';')
413      {
414        EMSTOP("compiler expected \';\'", get_linenum());
415        exit(1);
416      }
417    read_next();
418  }
419
420  static exp_node *
421  kakko_exp(void)
422  {
423    exp_node *p;
424
425    if (next.lex != '(')  /* このコメントは ) エディタの括弧対応のためのおまじない */
426      {
427        EMSTOP("compiler expected \'(\'", get_linenum());   /* ) */
428        exit(1);
429      }
430    read_next();
431    p = int_exp();   /* ( */
432    if (next.lex != ')')
433      {
434        EMSTOP("compiler expected \')\'", get_linenum());   /* ) */
435        exit(1);
436      }
```

```
437     read_next();
438 
439     return p;
440 }
441 
442 static exp_node *
443 int_exp()
444 {
445     exp_node *thisexp;
446 
447     thisexp = add_exp();
448     while ((next.lex == LEXVAL_eqeq)
449            || (next.lex == LEXVAL_bangeq)
450            || (next.lex == '<')
451            || (next.lex == '>')
452            || (next.lex == LEXVAL_lt_or_eq)
453            || (next.lex == LEXVAL_gt_or_eq))
454       {
455         exp_node *newexp;
456         newexp = malloc(sizeof(exp_node));
457         newexp->val._2._0 = thisexp;
458         thisexp = newexp;
459         switch (next.lex)
460           {
461           case LEXVAL_eqeq:
462             newexp->type = EXP_ISEQ;
463             break;
464           case LEXVAL_bangeq:
465             newexp->type = EXP_ISNOTEQ;
466             break;
467           case '<':
468             newexp->type = EXP_ISLT;
469             break;
470           case '>':
471             newexp->type = EXP_ISGT;
472             break;
473           case LEXVAL_lt_or_eq:
474             newexp->type = EXP_ISLTEQ;
475             break;
476           case LEXVAL_gt_or_eq:
477             newexp->type = EXP_ISGTEQ;
478             break;
479           default:
480             EMSTOP("internal conflict", get_linenum());
481             exit(1);
482           }
483         read_next();
484         newexp->val._2._1 = add_exp();
485       }
486 
487     return thisexp;
488 }
489 
490 static exp_node *
491 add_exp(void)
492 {
493     exp_node *thisexp;
494 
495     thisexp = mul_exp();
496     while ((next.lex == '+')
497            || (next.lex == '-')
498            || (next.lex == '|'))
499       {
500         exp_node *newexp;
501         newexp = malloc(sizeof(exp_node));
502         newexp->val._2._0 = thisexp;
503         thisexp = newexp;
```

```
504         switch (next.lex)
505           {
506           case '+':
507             newexp->type = EXP_ADD;
508             break;
509           case '-':
510             newexp->type = EXP_SUB;
511             break;
512           case '|':
513             newexp->type = EXP_OR;
514             break;
515           default:
516             EMSTOP("internal conflict", get_linenum());
517             exit(1);
518           }
519         read_next();
520         newexp->val._2._1 = mul_exp();
521       }
522 
523     return thisexp;
524   }
525 
526   static exp_node *
527   mul_exp()
528   {
529     exp_node *thisexp;
530 
531     thisexp = unary_exp();
532     while ((next.lex == '*')
533         || (next.lex == '/')
534         || (next.lex == '%')
535         || (next.lex == LEXVAL_lsl)
536         || (next.lex == LEXVAL_asr)
537         || (next.lex == LEXVAL_lsr)
538         || (next.lex == '&'))
539       {
540         exp_node *newexp;
541         newexp = malloc(sizeof(exp_node));
542         newexp->val._2._0 = thisexp;
543         thisexp = newexp;
544         switch (next.lex)
545           {
546           case '*':
547             newexp->type = EXP_MUL;
548             break;
549           case '/':
550             newexp->type = EXP_DIV;
551             break;
552           case '%':
553             newexp->type = EXP_MOD;
554             break;
555           case LEXVAL_lsl:
556             newexp->type = EXP_LSL;
557             break;
558           case LEXVAL_asr:
559             newexp->type = EXP_ASR;
560             break;
561           case LEXVAL_lsr:
562             newexp->type = EXP_LSR;
563             break;
564           case '&':
565             newexp->type = EXP_AND;
566             break;
567           default:
568             EMSTOP("internal conflict", get_linenum());
569             exit(1);
570           }
```

```
571         read_next();
572         newexp->val._2._1 = unary_exp();
573       }
574 
575     return thisexp;
576   }
577 
578   static exp_node *
579   unary_exp()
580   {
581     exp_node *op, *e, *thisexp;
582 
583     thisexp = 0;
584     op = 0;
585 
586     while ((next.lex == '~')
587         || (next.lex == '+')
588         || (next.lex == '-'))
589       {
590         exp_node *old_op;
591         old_op = op;
592         op = malloc(sizeof(exp_node));
593         if (thisexp == 0)
594           thisexp = op;
595         switch (next.lex)
596           {
597           case '~':
598             op->type = EXP_NOT;
599             break;
600           case '+':
601             op->type = EXP_PLUS;
602             break;
603           case '-':
604             op->type = EXP_MINUS;
605             break;
606           default:
607             EMSTOP("internal conflict", get_linenum());
608             exit(1);
609           }
610         if (old_op != 0)
611           old_op->val._1._0 = op;
612         read_next();
613       }
614     e = int_prim();
615     if (thisexp == 0)
616       thisexp = e;
617     else
618       op->val._1._0 = e;
619 
620     return thisexp;
621   }
622 
623   static exp_node *
624   int_prim()
625   {
626     exp_node *thisexp;
627     name_info *v;
628 
629     switch (next.lex)
630       {
631       case '(':   /* このコメントは ) エディタの括弧対応のためのおまじない */
632         read_next();
633         thisexp = int_exp();    /* ( */
634         if (next.lex != ')')
635           {   /* ( */
636             EMSTOP("compiler expected \')\'", get_linenum());
637             exit(1);
```

```
638          }
639        read_next();
640        break;
641      case LEXVAL_name:
642        thisexp = malloc(sizeof(exp_node));
643        thisexp->type = EXP_VAR;
644        v = name_find(loc_name_list, next.sem.p);
645        if (v == 0)
646          v = name_find(glo_name_list, next.sem.p);
647        if (v == 0)
648          {
649            EMSTOP("unknown name", get_linenum());
650            exit(1);
651          }
652        if (name_get_type(v) == NAME_PROC)
653          {
654            EMSTOP("name of procedure cannot be refereed", get_linenum());
655            exit(1);
656          }
657        thisexp->val.var = v;
658        read_next();
659        break;
660      case LEXVAL_literal:
661        thisexp = malloc(sizeof(exp_node));
662        thisexp->type = EXP_CONST;
663        thisexp->val.constval = next.sem.w;
664        read_next();
665        break;
666      default:
667        EMSTOP("expression is not correct", get_linenum());
668        exit(1);
669      }
670
671    return thisexp;
672  }
```

672 行...というわけでうんざりするかもしれませんが，類似している定型的な処理が多いので，ポイントを早めにつかめば解読は不可能ではありません．まずは 52 行目からいきましょう．

```
void
parse(void)
{
  read_next();  /* まず先読み */
  module();  /* プログラム全体は 1 個のモジュール定義 */
}
```

という関数が構文解析のトップです．read_next() は内部で字句解析関数 lex() を呼び出し，トークン値を得ます (next という変数に格納されます)．そして next に入っているトークン値をもとに文法のトップである module() を呼び出します．ここでテキストの図 2「プログラミング言語 ZINC の文法」をご覧ください．一番上の規則である，

<モジュールソース> ::= <モジュール内宣言>* <手続き定義>*

に注目しましょう．BNF 記法のままだと最初は読みづらいと思いますので，日本語で読み方を考えていきます．この規則は，
<モジュールソース>とは，0 個以上の<モジュール内宣言>の次に 0 個以上の<手続き定義>で構成される．
というように読むことが出来ます．なお，<モジュールソース>はプログラム内では module() 関数に相当します (65 行目)．

```
static void
module(void)
{
```

```
  gen_verb(".globl _main\n\n");
  for (;;)
    {
      switch (next.lex)
        {
        case LEXVAL_PROC:
          decl_proc();
          break;
        case LEXVAL_WORD:
          define_glo_var();
          break;
        default:
          goto end_decl_g;  /* for (;;) を抜ける */
        }
    }
end_decl_g:
  while (next.lex == LEXVAL_DEFPROC)
    define_proc();

  if (next.lex != LEXVAL_eof)
    {
      EMSTOP("unknown token", get_linenum());
      exit(1);
    }
}
```

さて，もう一度実験テキストの図 2 を調べると、‘<’ ‘>’ で囲まれた要素と，defproc のようにそのままの要素があることに気づくと思います．前者を非終端記号，後者を終端記号といいます．終端記号は ZINC プログラムの中にそのままの形で登場します．すなわちこれ以上展開できない終端の記号ということです．また，左辺に非終端記号 ‘<’ ‘>’ で囲まれた要素しかないことに気づくと思います．これはプログラムのトップレベル，すなわち<モジュールソース>からはじまって図 2 の規則を用いて右辺の形に展開していき，すべての要素が展開されて非終端記号になったところで解析が終了することを表しています．また，プログラムとの関連で考えると，非終端記号は parse.c 内でそれぞれ関数として登場します（実行効率の関係上，1 対 1 の関係にはなっていない）．さきほどすでに述べたように，<モジュールソース>は parse.c 内で module() に相当します．<モジュールソース>は 0 個以上の<モジュール内宣言>の次に 0 個以上の<手続き定義>が続くはずである，ということがテキストの図 2 からわかっているので，それを頭に入れながら module() 関数を見ていきましょう．最初にある

```
gen_verb(".globl _main\n\n");
```

は_main 関数用のラベルを設定しているだけです．つぎに for(;;) による無限ループがあります．この次につづく switch で next.lex の値（つまりトークン値）を調べ，

```
  for (;;)
    {
      switch (next.lex)
        {
        case LEXVAL_PROC:
          decl_proc();
          break;
        case LEXVAL_WORD:
          define_glo_var();
          break;
```

```
        default:
          goto end_decl_g;  /* for (;;) を抜ける */
        }
    }
```

その値が，LEXVAL_PROC(すなわち proc) であって手続き宣言と判断できる場合は decl_proc() を呼出し，LEXVAL_WORD(すなわち word) でグローバル変数定義と判断できる場合は define_glo_var() を呼び出します．for 文による無限ループによって，0 個以上の非終端記号の処理を表現しています（最初から LEXVAL_PROC でも LEXVAL_WORD でもなく，default に落ちる場合で 0 個の並びも表現できます）．このことを少し不思議に感じた方もいるかもしれません．先ほど各非終端記号に対し，parse.c 内で相当する関数が存在すると書きましたが，いまの場合，<モジュール内宣言>に相当する関数が見当たりません．実はこの関数に相当するのは上記の for ループなのです．
<モジュール内宣言>は<手続き宣言>または<グローバル変数定義>である
というだけの意味しかないので（トークンをみて選択するだけ），1 つ上の module() 内で展開してしまっているわけです．

　手続き宣言またはグローバル変数定義以外のものがきたら，for(;;) を goto で脱出し，今度は<手続き定義>に相当する 84 行目以降の，

```
end_decl_g:
  while (next.lex == LEXVAL_DEFPROC)
    define_proc();
```

という処理に移ります．このように，トップレベル規則<モジュールソース>を展開するところからはじめ，基本的に図 2 の文法に従って非終端記号（の展開）に相当する関数を呼び出して構文解析をおこなっていきます．展開候補が複数ある場合は，lex() を呼び出して得たトークン値によってどの関数を呼び出すのか決定します．ここでいったん図 11 に示すコールグラフを見てください．これは parse 関数から下位の関数がどのように呼び出されるかを示したものです．

```
switch (next.lex)
```

以上まだ書きかけです

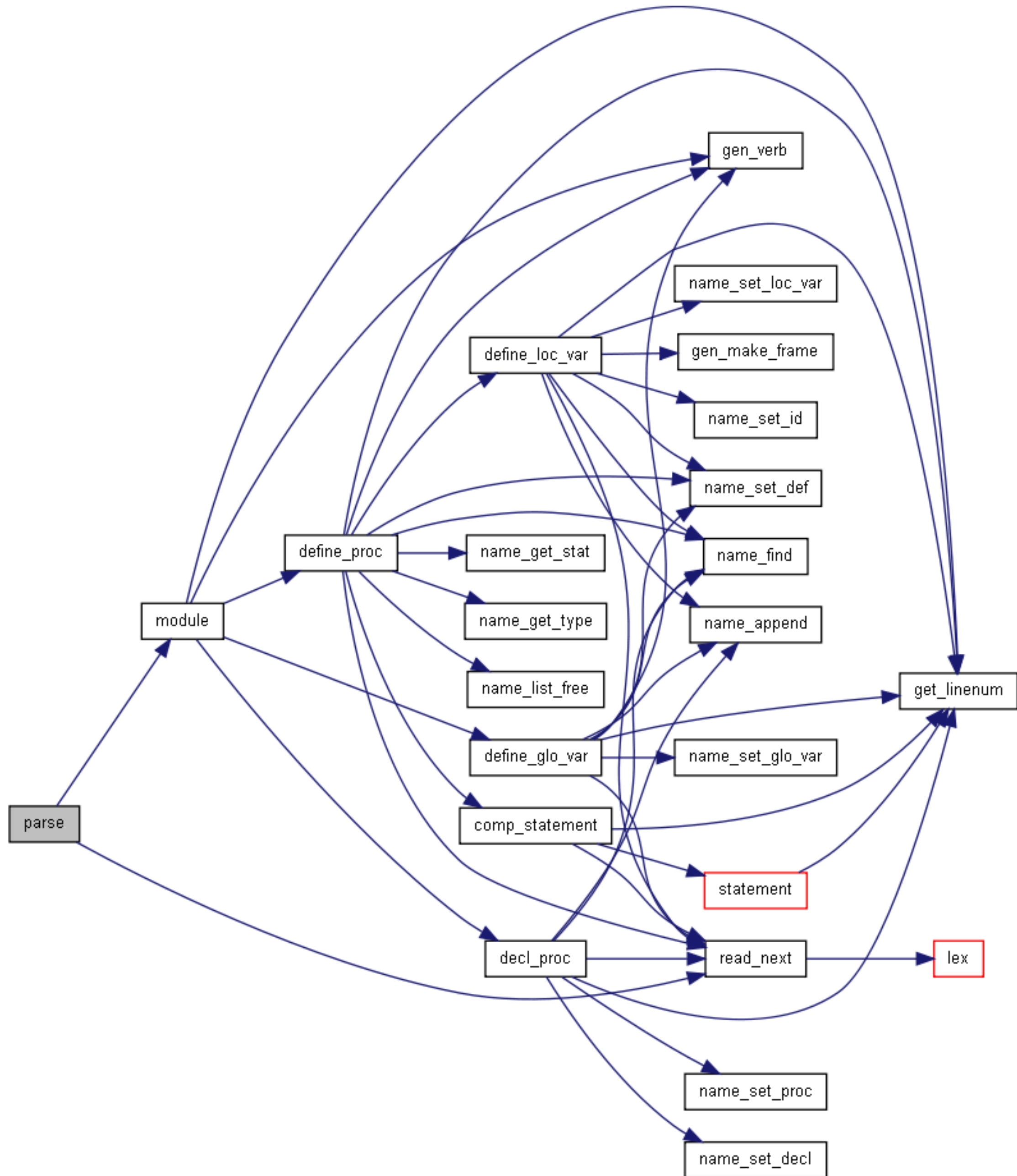

図 11: parse() のコールグラフ