低レイヤを知りたい人のための Cコンパイラ作成入門

Rui Ueyama <ruiu@cs.stanford.edu>

2019/01/15

はじめに

このオンラインブックは執筆中です。完成版ではありません。フィードバックフォーム

この本には一冊の本に盛り込むにはやや欲張りな内容を詰め込みました。本書では、C言語で書かれたソースコードをアセンブリにコンパイルするコンパイラ、つまりCコンパイラを作成します。コンパイラそのものもCを使って開発します。当面の目標はセルフホスト —— すなわち自作コンパイラでそれ自身のソースコードをコンパイルできるようにすることです。

この本では、コンパイラの理論の学習曲線が急峻になりすぎないように、様々なトピックを本書全体を通じて次第に掘り下げていくという形で説明することにしました。その理由は次のとおりです。

コンパイラは、構文解析、意味解析、コード生成といった複数のステージに概念的に分割することができます。よくある教科書的アプローチでは、それぞれのトピックについて章を立てて解説を行うことになります。そのようなアプローチの本は、話が途中で高度になりすぎて読み

通すのが辛いことがあります。その結果として、本の最初に書いてあった構文解析の、最初のほうに書いてあったテクニックだけはわかるけど……といった状態になることがよくあります。

また、こういった開発手法では、全てのステージが完成するまでコンパイラを動かしてみることができないので、全体が動き始めるまで自分の理解やコードが決定的に間違っていても気づくことができません。そもそも次のステージの入力としてなにが期待されているのか、自分で作ってみるまでよくわからないので、あるステージで何を出力すればよいのかもよくわからないのです。また、完成するまで何のコードもまったくコンパイルできないのでモチベーションを保つのが困難という問題もあります。

本書ではその罠を避けるために別のアプローチをとることにしました。この本の最初のほうで、読者はごく単純な言語仕様の「独自言語」を実装することになります。その言語はあまりにも単純なので、それを実装する時点では、コンパイラの作り方について詳しく知っている必要はありません。その後、読者は本書を通じて「独自言語」に機能を追加していって、最終的にそれをC言語と一致するものに育て上げることになります。

そのようなインクリメンタルな開発手法では、細かいコミットを刻みつつ、ステップ・バイ・ステップでコンパイラを作っていくことになります。この開発手法では、どのコミットにおいてもコンパイラはある意味常に「完成形」です。ある段階ではただの電卓レベルのことしかできないかもしれないし、ある段階では相当限定されたCのサブセットかもしれないし、ある段階ではほぼCと言える言語かもしれない、というようになります。ポイントは、どの段階でも、その時点の完成度に合わせたリーズナブルな仕様の言語を目指すという点です。別の言い方をすれば、ものすごく簡単な自作言語のコンパイラを作り、それに次第にCぽい方向に向かって少しづつ進化させていくことで、最終的に自作言語をCと同じものにする、ということを行います。開発中に一部の機能だけを突出してC言語ぽくすることは行いません。

インクリメンタルな開発では、本書を読んでいるどの時点においても、読者はそこまでのレベルにおいてのリーズナブルな言語の作り方の知識をまんべんなく持っている、ということが達成されることになります。これはコンパイラ作成の一部のトピックだけ極端に偏って詳しい状態よりずっとよい状態です。そして、本書を読み終わるころには、すべてのトピックについてまんべんなく知識が得られていることでしょう。

また、この本は、大きなプログラムを1から書くにはどうすればよいのかということを説明している本でもあります。大きなプログラムを作るスキルというのは、データ構造やアルゴリズムを学ぶのとはまた違った一種独特のスキルなのですが、そういったものを解説している本はあまりないように思います。また、仮に解説してもらっても、実際に体験してみなければピンとこないでしょう。本書は、自作言語をC言語に育てていくプロセスがその実体験になりうるようにデザインされています。

筆者の目論見が成功していれば、この本を読むことで、読者はコンパイラ作成のテクニックやCPU命令セットの知識だけではなく、大きなプログラムを小さなステップにわけて少しづつ作っていく方法や、ソフトウェアテストの手法、バージョン管理の手法、そしてコンパイラ作成のような野心的なプロジェクトに取り組むときの心構えすら学ぶことができるはずです。

この本の想定読者は普通のCプログラマです。Cの言語仕様を熟知しているスーパーCプログラマである必要はありません。ポインタや配列が理解できており、他人の書いた小規模なCプログラムを、少なくとも時間をかければ読める、というレベルであれば十分です。

この本の執筆にあたって、言語仕様やCPUの仕様については、単に仕様を説明するだけではなく、なぜそのようなデザインが選ばれたのかについての解説をできる限り行うようにしました。また、読者の興味を引くようなコンパイラやCPU、コンピュータ業界や歴史についてのコラムを散りばめて、楽しく読み進められるように心がけました。

コンパイラ作成は大変楽しい作業です。最初のころはバカバカしいくらい単純なことしかできなかった「自作言語」のためのコンパイラが、開発を続けていくとたちまちのうちに自分でも驚くくらいC言語っぽく成長していって、まるで魔法のようにうまく動くようになります。実際に開発をしてみると、その時点でうまくコンパイルできるとは思えない大きめのテストコードがエラーなしにコンパイルできて、完全に正しく動くことに驚くことがよくあります。そういうコードはコンパイル結果のアセンブリを見ても自分ではほとんど理解できません。時折、自作のコンパイラが作者である自分を超える知性を持っているように感じることすらあります。コンパイラは仕組みがわかっていても、どことなく、なぜここまでうまく動くのか不思議な感じがするプログラムです。きっとあなたもその魅力に夢中になることでしょう。

さて、前置きはこれくらいにして、さっそく筆者と一緒にコンパイラ開発の世界に飛び込んでみましょう！

コラム: なぜC言語なのか

数多くあるプログラミング言語の中で、この本ではなぜCを選んだのでしょうか？ あるいはなぜ自作言語ではないのでしょうか？ この点については、絶対にCでなければならない理由はないのですが、ネイティブコードを出力するコンパイラの制作手法を学ぶために何らかの言語を選ばなければいけないとしたら、Cはさほど多くないリーズナブルな選択肢のうちの一つだと思います。

インタプリタ方式の言語では低レイヤについて学ぶことができません。一方でCでは普通はアセンブリにコンパイルするので、C言語そのものと同時に、CPUの命令セットやメモリレイアウトなどについて学ぶことができます。

Cは広く使われているので、コンパイラがきちんと動くようになったら、ネットからダウンロードしてきた第三者のソースコードをコンパイルして遊ぶことができます。例えばミニUnixのxv6をビルドして遊ぶことができるでしょう。コンパイラの完成度が十分に高ければ、Linuxカーネルすらコンパイルすることが可能なはずです。こういった楽しみ方はマイナーな言語や自作言語ではできません。

Cのようなネイティブな機械語にコンパイルされる静的型付け言語で、Cと少なくとも同じくらい広く使われているものとして、C++があります。しかしC++は、言語仕様があまりにも巨大で、気軽に自作コンパイラを作るというのは不可能で、現実的に選択肢に入りません。

オリジナルの言語をデザインして実装するのは、言語のデザインセンスを磨くという意味ではよいのですが、落とし穴もあります。実装が面倒なところは、言語仕様でそれを避けることにより実装しないで済ませてしまうことができるのです。言語仕様が標準として与えられているCのような言語ではそうはいきません。その縛りは学習という意味ではわりとよいものだと思います。

本書の想定する開発環境

本書では開発環境として、IntelやAMDなどのいわゆる普通のPCで動く64ビットのLinux環境を想定します。読者がお使いのディストリビューションに合わせてgccやmakeといった開発ツールをあらかじめインストールしておいてください。

Intel CPU上のmacOSは、Linuxとアセンブリのソースレベルでかなり互換性がありますが、完全互換ではありません。この本の内容に従ってmacOS対応のCコンパイラを作成するのは不可能ではないものの、実際に試してみると、細かな点でいろいろな非互換性に悩まされることになるでしょう。Cコンパイラ作成のテクニックと、macOSとLinuxの差異を同時に学ぶというのは、あまりお勧めできることではありません。何かがうまく動かない場合、どちらの理解が間違っているのかよくわからなくなってしまうからです。したがって現時点では、本書ではmacOSは対象外とします。macOSではVirtualBoxなどの仮想マシンを使ってLinuxを使うようにしてください。

WindowsはLinuxとはアセンブリのソースレベルで互換性がありません。ただし、Windows 10ではLinuxを1つのアプリケーションのようにWindows上で動作させることが可能で、それを使うことでWindows上で開発を進めていくことができます。Windows Subsystem for Linux（WSL）というアプリケーションがそのLinux互換環境の名前です。本書の内容を

Windowsで実践するときは、WSLをインストールして、その中で開発を進めるようにしてください。

本書の表記法

関数や式、コマンドなどは本文中で`main`や`foo=3`、`make`のように等幅フォントで表示します。

複数行に渡るコードは、次のように等幅フォントを使って枠の中に表示します。

```
int main() {
  printf("Hello world!\n");
  return 0;
}
```

ユーザがそのまま入力することを想定しているシェルコマンドの場合、`$`から始まる行はプロンプトを意味しています。その行の`$`以降をシェルに入力してください（`$`そのものは入力しないようにしてください）。`$`以外の行は、入力したコマンドからの出力を表しています。例えば下のブロックは、ユーザが`make`という文字列を入力してエンターを押した場合の実行例です。`make`コマンドからの出力は`make: Nothing to be done for `all'.`です。

```
$ make
make: Nothing to be done for `all'.
```

コラム: コンパイラをコンパイルするコンパイラ

CコンパイラがCで書かれているといった自己参照的な状況は珍しくありません。C以外でも、数多くの言語実装がその言語自体を使って書かれています。

すでに言語Xの実装がある場合、その言語自身を使って新たなXコンパイラを作ることに論理的な矛盾はありません。セルフホストするためには、単に既存のコンパイラで開発を進めていって、自作のものが完成したらスイッチすればよいだけです。この本で我々が行おうとしているのはまさにその方法です。

しかし既存のコンパイラがない場合はどうすればよいのでしょうか？ そのときには別の言語を使って書くしかありません。セルフホストを目的としてX言語の最初のコンパイラを書くときには、Xと異なる既存のY言語を使って書き、コンパイラの完成度が高まったところで、コンパイラ自身をY言語からX言語に書き直す必要があります。

現代の複雑なプログラミング言語のコンパイラも、その言語の最初のコンパイラをコンパイルするために使った別のコンパイラ、というように系譜をさかのぼっていくと、最終的に、コンピュータの黎明期に誰かが機械語で直接書いた単純なアセンブラにたどりつくはずです。現存するコンパイラのある意味の究極の祖先にあたるそのアセンブラが、単一なのか複数あったのかはわかりませんが、現在のコンパイラがごく少数の祖先から出発しているのは間違いないでしょう。コンパイラではない実行ファイルもコンパイラが生成したファイルですから、現存するほぼすべての実行ファイルのバイナリは、その原始のアセンブラの間接的な子孫にあたるわけです。これは生命の起源のような面白い話ですね。

変更履歴

- 2019/01/13: 関数呼び出しのサポート以降の内容を追記
- 2019/01/02: 再帰下降構文解析で説明に使っている文法を右再帰から左再帰に変更
- 2018/10/31: 初期版を公開

機械語とアセンブラ

この章では、コンピュータを構成するコンポーネントと、我々が作成するCコンパイラからどのようなコードを出力すればよいのかということについて、大雑把なイメージをつかむことを目標とします。具体的なCPUの命令などについてはまだ深入りはしません。まずは概念を把握することが重要です。

CPUとメモリ

コンピュータを構成するコンポーネントは、大まかにいうとCPUとメモリにわけることができます。メモリはデータを保持できるデバイスで、CPUは、そのメモリを読み書きしながら何らかの処理を行なっていくデバイスです。概念的に、CPUにとってはメモリはランダムアクセス可能な巨大なバイトの配列のように見えます。CPUがメモリにアクセスするときは、メモリの何バイト目にアクセスしたいのかという情報を数値で指定するわけですが、その数値のことを「アドレス」といいます。例えば「アドレス16から8バイトのデータを読む」というのは、バイトの配列のように見えているメモリの16バイト目から8バイト分のデータを読む、という意味です。同じことを「16番地から8バイトのデータを読む」ということもあります。

CPUが実行するプログラムと、そのプログラムが読み書きするデータは、どちらもメモリに入っています。CPUは「現在実行中の命令のアドレス」をCPU内部に保持していて、そのアドレスから命令を読み出して、そこに書かれていることを行い、そして次の命令を読み出して実行する、ということを行なっています。その現在実行中の命令のアドレスのことを「プログラムカウンタ」（PC）や「インストラクションポインタ」（IP）といいます。CPUが実行するプログラムの形式そのもののことを「機械語」といいます。

プログラムカウンタは必ずしも直線的に次の命令だけに進んでいくわけではありません。CPUの「分岐命令」と呼ばれる命令を使うと、プログラムカウンタを、次の命令以外の任意のアドレスに設定することができます。この機能によってif文などが実現されています。プログラムカウンタを次の命令以外の場所に設定することを「ジャンプする」あるいは「分岐する」といいます。

CPUはプログラムカウンタのほかにも、少数のデータ保存領域を持っています。例えばIntelやAMDのプロセッサには、64ビット整数が保持できる領域が16個あります。この領域のことを「レジスタ」と呼びます。メモリはCPUからみて外部の装置で、それを読み書きするには多少の時間がかかりますが、レジスタはCPU内部に存在していて、遅延なしにアクセスすることができます。

多くのCPU命令は、2つのレジスタの値を使って何らかの演算を行なって、その結果をレジスタに書き戻すというフォーマットになっています。したがってプログラムの実行というものは、CPUがメモリからレジスタにデータを読み込んできて、レジスタとレジスタの間でなんらかの演算を行い、その結果をメモリに書き戻す、ということで実行が進んでいくことになります。

特定の機械語の命令を総称として「命令セット」といいます。命令セットは一種類というわけではなく、CPUごとに好きにデザインしてかまいません。とはいえ、機械語レベルの互換性がないと同じプログラムを動かせないので、命令セットのバリエーションはそれほど多くありません。PCでは、Intelやその互換チップメーカーであるAMDの、x86-64と呼ばれる命令セットが使われています。x86-64は主要な命令セットの1つですが、x86-64だけが市場を独占して

いるというわけではありません。例えばiPhoneやAndroidではARMという命令セットが使われてます。

アセンブラとは

機械語はCPUが直接読んでいくものですから、CPUの都合だけが考慮されていて、人間にとっての扱いやすさというものは考慮されていません。こういった機械語をバイナリエディタで書いていくのは、不可能というわけではないものの、とても辛い作業です。そこで発明されたのがアセンブラです。アセンブリは機械語にほぼそのまま1対1で対応するような言語なのですが、機械語よりもはるかに人間にとって読みやすいものになっています。

アセンブリのコードを機械語に変換するのは「コンパイルする」ということもありますが、入力がアセンブリであることを強調して特別に「アセンブルする」ということもあります。

仮想マシンやインタープリタではなくネイティブなバイナリを出力するコンパイラの場合、アセンブリを出力することが目標になります。本書で作るCコンパイラもアセンブリを出力します。

読者はアセンブリをいままでにどこかで見たことがあるかもしれません。もしアセンブリを見たことがなければ、今が見てみるよい機会です。`objdump`コマンドを使って、適当な実行ファイルを逆アセンブルして、そのファイルの中に入っている機械語をアセンブリとして表示してみましょう。以下は`ls`コマンドを逆アセンブルしてみた結果です。

```
$ objdump -d -M intel /bin/ls
/bin/ls:     file format elf64-x86-64

Disassembly of section .init:

0000000000003d58 <_init@@Base>:
  3d58:  48 83 ec 08           sub    rsp,0x8
  3d5c:  48 8b 05 7d b9 21 00  mov    rax,QWORD PTR [rip+0x21b97d]
  3d63:  48 85 c0              test   rax,rax
  3d66:  74 02                 je     366a <_init@@Base+0x12>
  3d68:  ff d0                 call   rax
  3d6a:  48 83 c4 08           add    rsp,0x8
  3d6e:  c3                    ret
...
```

筆者の環境では`ls`コマンドには2万個ほどの機械語命令が含まれているので、逆アセンブルした結果も2万行近い長大なものになります。ここでは最初のごく一部だけを掲載しました。

アセンブリでは、基本的に機械語1個につき1行という構成になっています。例として次の行に着目してみましょう。

```
  3d58:  48 83 ec 08           sub    rsp,0x8
```

この行の意味は何でしょうか？ 3d58というのは、機械語が入っているメモリのアドレスです。つまり、`ls`コマンドが実行されるとき、この行の命令はメモリの0x3d58番地に置かれるようになっていて、プログラムカウンタが0x3d58のときにこの命令が実行されることになります。その次に続いている4つの16進数の数値は実際の機械語です。CPUはこのデータを読んで、それを命令として実行します。`sub rsp,0x8`というのは、その機械語命令に対応するアセンブリです。CPUの命令セットについては章を分けて説明しますが、この命令は、RSPというレジスタから8を引く（subtract = 引く）という命令です。

Cとそれに対応するアセンブラ

簡単な例

Cコンパイラがどのような出力を生成しているのかというイメージを掴むために、Cコンパイラが出力するアセンブリコードを少し見てみましょう。最も簡単な例として次のプログラムを考えてみます。

```
int main() {
  return 42;
}
```

このプログラムが書かれているファイルを`test1.c`とすると、次のようにしてコンパイルして、`main`が実際に42を返していることを確認することができます。

```
$ gcc -o test1 test1.c
$ ./test1
$ echo $?
42
```

Cではmain関数が返した値はプログラム全体としての終了コードになります。プログラムの終了コードは画面に表示されることはありませんが、暗黙のうちにシェルの$?という変数にセットされているので、コマンド終了直後に$?をechoで表示することで、そのコマンドの終了コードを見ることができます。ここでは正しく42が返されていることがわかります。

さて、このCプログラムに対応するアセンブリプログラムは次の通りです。

```
.intel_syntax noprefix
.global main
main:
        mov rax, 42
        ret
```

このアセンブリでは、グローバルなラベルmainが定義されていて、ラベルのあとにmain関数のコードが続いています。ここでは42という値を、RAXというレジスタにセットし、mainからリターンしています。整数を入れられるレジスタはRAXを含めて合計で16個あるのですが、関数からリターンしたときにRAXに入っている値が関数の返り値という約束になっているので、ここではRAXに値をセットしています。

このアセンブリプログラムを実際にアセンブルして動かしてみましょう。アセンブリファイルの拡張子は.sなので、上のアセンブリコードをtest2.sに記述して、次のコマンドを実行してみてください。

```
$ gcc -o test2 test2.s
$ ./test2
$ echo $?
42
```

Cのときと同じように42が終了コードになりました。

大雑把にいうと、Cコンパイラは、test1.cのようなCコードを読み込んで、test2.sのようなアセンブリを出力するプログラムということになります。

関数呼び出しを含む例

もう少し複雑な例として、関数呼び出しのあるコードがどのようなアセンブリに変換されるのかを見てみましょう。

関数呼び出しは単なるジャンプとは異なり、呼び出した関数が終了した後に、元々実行していた場所に戻ってこなければいけません。元々実行していたアドレスのことを「リターンアドレス」といいます。仮に関数呼び出しが1段しかなければ、リターンアドレスはCPUの適当なレジスタに保存しておけばよいのですが、実際には関数呼び出しはいくらでも深くできるので、リターンアドレスはメモリに保存することになっています。従ってリターンアドレスはスタックになっています。

スタックは、スタックの一番上のアドレスを保持する1つの変数のみを使って実装することができます。そのスタックトップを保持している記憶領域のことを「スタックポインタ」といいます。x86-64には、関数を使ったプログラミングをサポートするために、スタックポインタ専用のレジスタと、そのレジスタを利用する命令をサポートしています。スタックにデータを積むことを「プッシュ」、スタックに積まれたデータを取り出すことを「ポップ」といいます。

さて、関数呼び出しの実例を見てみましょう。次のCコードを考えてみてください。

```
int plus(int x, int y) {
  return x + y;
}

int main() {
  return plus(3, 4);
}
```

このCコードに対応するアセンブリは次のようになります。

```
.intel_syntax noprefix
.global plus, main

plus:
        add rsi, rdi
        mov rax, rsi
        ret

main:
        mov rdi, 3
        mov rsi, 4
```

```
        call plus
        ret
```

1行目はアセンブリの文法を指定する命令です。2行目の.globalから始まる行は、plusとmainという2つの関数がファイルスコープではなくプログラム全体から見える関数だということをアセンブリに指示しています（関数定義にstaticをつけるとグローバルではなくなります）。これはさしあたり無視してかまいません。

まずmainに着目してみてください。Cではmainからplusを引数つきで呼び出しています。アセンブラにおいては、第一引数はRDIレジスタ、第二引数はRSIレジスタに入れるという約束になっているので、mainの最初の2行でそのとおりに値をセットしています。

callというのは関数を呼び出す命令です。具体的にcallが行うのは次のことです。

- callの次の命令のアドレスをスタックにプッシュ
- callの引数として与えられたアドレスにジャンプ

したがってcall命令が実行されると、CPUはplus関数を実行し始めることになります。plus関数に着目してください。plus関数には3つの命令があります。

addは足し算を行う命令です。この場合には、RDIレジスタとRSIレジスタを足した結果がRSIレジスタに書き込まれます。x86-64の整数演算命令は、通常2つのレジスタしか受け取らないので、どちらかのレジスタの値を上書きする形で結果が書き込まれることになります。

関数からの返り値はRAXに入れるということになってました。したがって足し算の結果はRAXに入れておきたいので、RSIからRAXに値をコピーする必要があります。ここではmov命令を使ってそれを行なっています。movはmoveの省略形ですが、実際にはデータを移動するのではなくコピーする命令です。

plus関数の最後では、retを呼んで関数からリターンしています。具体的にretが行うのは次のことです。

- スタックからアドレスを1つポップ
- そのアドレスにジャンプ

つまりretは、callが行なったことを元に戻して、呼び出し元の関数の実行を再開する命令です。このようにcallとretは対になる命令として定義されています。

plusからリターンしたところにあるのはmainのret命令です。元のCコードではplusの返り値をそのままmainから返すということになっていました。ここではplusの返り値がRAXに入

った状態になっているので、そのまま`main`からリターンすることで、それをそのまま`main`からの返り値にすることができます。

本章のまとめ

本章ではコンピュータが内部でどのように動いているのかということと、Cコンパイラが何をすればよいのかということについて、概要を説明しました。機械語やアセンブリを見ると、Cとはかけ離れた、ごちゃっとしたデータの塊のように見えますが、実際は意外とCの構造を素直に反映していると思った読者も多いのではないでしょうか。

まだ本書では具体的なCPU命令についてほとんど説明していないので、`objdump`で表示されたアセンブリコードについて1つ1つの命令はわからないと思いますが、何をしていそうかということについて、なんとなく勘が効くようになったのではないでしょうか。本章の段階ではそういう勘が効くようになれば十分です。

本章のポイントを箇条書きで下にまとめます。

- CPUはメモリを読み書きすることでプログラムの実行を進めていく
- CPUが実行するプログラムと、そのプログラムが扱うデータは、どちらもメモリに入っていて、CPUはメモリから順に機械語命令を読み、その命令を実行する
- CPUにはレジスタという小さな記憶領域があり、多くのCPU命令はレジスタ間での操作として定義されている
- アセンブリは機械語を人間にとって読みやすくした言語で、Cコンパイラはアセンブリを出力する
- 関数呼び出しはスタックを使って実装されている

ステップ1：整数1個をコンパイルする言語の作成

最もシンプルなC言語のサブセットを考えてみてください。読者の皆さんはどういう言語を想像するでしょうか？ `main`関数しかない言語でしょうか。あるいは式1つだけからなる言語でしょうか。

突き詰めて考えると、整数1つだけからなる言語というものが、考えうる限り最も簡単なサブセットだといってよいと思います。したがってこのステップではまずその言語を実装します。

このステップで作成するプログラムは、1個の数を入力から読んで、その数をプログラムの終了コードとして終了するアセンブリを出力するコンパイラです。つまり入力は単に42のような文字列で、それを読むと次のようなアセンブリを出力するコンパイラです。

```
.intel_syntax noprefix
.global main

main:
        mov rax, 42
        ret
```

`.intel_syntax noprefix`というのは、複数あるアセンブリの書き方のなかで、本書で使っているIntel記法という記法を選ぶためのアセンブラコマンドです。今回作成するコンパイラでは必ず冒頭にこの行をお約束として入れるようにしてください。それ以外の行は、[第1章]で説明した通りです。

読者はここで、「こんなプログラムはコンパイラとは言えない」と思うかもしれません。筆者も正直そう思います。しかし、このプログラムは、数値1つからなる言語を入力として受け付けて、出力としてその数値に対応したコードを出力するというもので、それはコンパイラの定義から言うと立派なコンパイラです。このような簡単なプログラムも、改造していくとすぐにかなり難しいことができるようになるので、まずはこのステップを完了してみましょう。

実はこのステップは、開発全体の手順からみてみるととても重要です。このステップで作るものをスケルトンとして使って今後開発を進めていくからです。このステップでは、コンパイラ本体の作成に加えて、ビルドファイル（Makefile）、ユニットテストの作成、gitリポジトリのセットアップも行います。それらの作業について1つ1つ見ていきましょう。

なお、本書で作るCコンパイラは9ccという名前です。ccというのはC compilerの略称です。9という数字に特に意味はないのですが、筆者の以前につくったCコンパイラが8ccという名前なので、それの次の作品ということで9ccという名前にしました。もちろんみなさんは好きな名前をつけてもらってかまいません。ただし、事前に名前を考えすぎてコンパイラ作成が始められないということはないようにしましょう。GitHubのリポジトリも含め、名前は後から変えられるので、適当な名前で始めて問題ありません。

コラム: Intel記法とAT&T記法

Intel記法の他にAT&T記法というアセンブラの記法もUnixを中心に広く使われています。gccやobjdumpはデフォルトではAT&T記法でアセンブリを出力します。

AT&T記法では結果レジスタが第2引数に来ます。したがって2引数の命令では引数を逆順に書くことになります。レジスタ名には`%`プレフィックスをつけて`%rax`というように書きます。数値には`$`プレフィックスをつけて`$42`というように記述します。

また、メモリを参照する場合、`[]`の代わりに`()`を使って、独特の記法で式を記述します。以下にいくつか対比のために例を示します。

```
mov rbp, rsp   // Intel
mov %rsp, %rbp // AT&T

mov [rbp + rcx * 4 - 8], rax // Intel
mov %rax, -8(rbp, rcx, 4)    // AT&T
```

今回作るコンパイラでは読みやすさを考慮してIntel記法を使うことにしました。Intelの命令セットマニュアルではIntel記法が使われているので、マニュアルの記述をそのままコードに書けるという利点もあります。表現力はAT&T記法もIntel記法も同じです。どちらの記法を使っても、生成される機械語命令列は同一です。

コンパイラ本体の作成

コンパイラには通常はファイルとして入力を与えますが、ここではファイルIOをするのが面倒なので、コマンドの第1引数に直接コードを与えることにします。第1引数を数値として読み込んで、定型文のアセンブリの中に埋め込むCプログラムは、次のように簡単に書くことができます。

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

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "引数の個数が正しくありません\n");
```

```
    return 1;
  }

  printf(".intel_syntax noprefix\n");
  printf(".global main\n");
  printf("main:\n");
  printf("  mov rax, %d\n", atoi(argv[1]));
  printf("  ret\n");
  return 0;
}
```

9ccという空のディレクトリを作って、その中に9cc.cというファイルを上記の内容で作成します。そのあと下のように9ccを実行して動作を確認してみましょう。

```
$ gcc -o 9cc 9cc.c
$ ./9cc 123 > tmp.s
```

1行目で9cc.cをコンパイルして9ccという実行ファイルを作成しています。2行目では123という入力を9ccに渡してアセンブリを生成し、それをtmp.sというファイルに書き込んでいます。tmp.sの内容を確認してみましょう。

```
.intel_syntax noprefix
.global main
main:
  mov rax, 123
  ret
```

見ての通りうまく生成されていますね。こうしてできたアセンブリファイルをアセンブラに渡すと実行ファイルを作成することができます。

Unixにおいてはcc（あるいはgcc）は、CやC++だけではなく多くの言語のフロントエンドということになっていて、与えられたファイルの拡張子で言語を判定してコンパイラやアセンブラを起動するということになっています。したがってここでは9ccをコンパイルしたときと同じように、.sという拡張子のアセンブラファイルをgccに渡すと、アセンブルをすることができます。以下はアセンブルを行い、生成された実行ファイルを実行してみた例です。

```
$ gcc -o tmp tmp.s
$ ./tmp
$ echo $?
123
```

シェルでは直前のコードの終了コードが$?という変数でアクセスできるのでした。上の例では、9ccに与えた引数と同じ123という数字が表示されています。つまりうまく動いているということです。0〜255の範囲の123以外の数を与えてみて（Unixの終了コードは0〜255ということになっています）、実際に9ccがうまく動くことを確認してみてください。

ユニットテストの作成

趣味のプログラミングでテストを書いたことがない読者も多いと思いますが、本書ではコンパイラを拡張するたびに、新しいコードをテストするコードを書くことにします。テストを書くのは最初は面倒に感じるかもしれませんが、すぐにテストのありがたみがわかるようになるはずです。テストコードを書かなかった場合、結局は同じテストを手で毎回走らせて動作確認をするしかないわけですが、そちらのほうが圧倒的に面倒なのです。

テストが面倒だという印象の多くの部分は、テストフレームワークが大げさであったり、テストの思想が時に教条的であるところからきていると思います。例えばJUnitのようなテストのフレームワークはいろいろな便利な機能を持っていますが、導入するのも使い方を覚えるの面倒です。したがってこの章ではそういったテストフレームワークを導入することはしません。インクリメンタルに、手書きのとても簡単な「テストフレームワーク」をシェルスクリプトで書いて、それを使ってテストを書くことにします。

以下にテスト用のシェルスクリプト`test.sh`を示します。シェル関数`try`は、引数を入力の値と、期待される出力の値という2つの引数を受け取って、実際に9ccの結果をアセンブルし、実際の結果を期待されている値と比較するということを行います。シェルスクリプトでは、`try`関数を定義した後に、それを使って0と42がどちらも正しくコンパイルできることを確認しています。

```
#!/bin/bash
try() {
  expected="$1"
  input="$2"

  ./9cc "$input" > tmp.s
```

```
  gcc -o tmp tmp.s
  ./tmp
  actual="$?"

  if [ "$actual" = "$expected" ]; then
    echo "$input => $actual"
  else
    echo "$expected expected, but got $actual"
    exit 1
  fi
}

try 0 0
try 42 42

echo OK
```

上記の内容で`test.sh`を作成し、`chmod a+x test.sh`を実行して実行可能にしてください。実際に`test.sh`を走らせてみましょう。何もエラーが起きなければ、以下のように`test.sh`は最後に`OK`を表示して終了します。

```
$ ./test.sh
0 => 0
42 => 42
OK
```

もしエラーが起きれば、`test.sh`は`OK`を表示しません。その代わりに`test.sh`は、失敗したテストで想定されていた値と実際の値を以下のように表示します。

```
$ ./test.sh
0 => 0
42 expected, but got 123
```

テストスクリプトをデバグしたいときは、bashに`-x`というオプションを与えてスクリプトを実行してください。`-x`オプションをつけると、bashは以下のように実行のトレースを表示します。

```
$ bash -x test.sh
+ try 0 0
+ expected=0
+ input=0
+ gcc -o 9cc 9cc.c
+ ./9cc 0
+ gcc -o tmp tmp.s
+ ./tmp
+ actual=0
+ '[' 0 '!=' 0 ']'
+ try 42 42
+ expected=42
+ input=42
+ gcc -o 9cc 9cc.c
+ ./9cc 42
+ gcc -o tmp tmp.s
+ ./tmp
+ actual=42
+ '[' 42 '!=' 42 ']'
+ echo OK
OK
```

我々が本書を通して使う「テストフレームワーク」は、単なる上記のようなシェルスクリプトです。このスクリプトはJUnitなどの本格的なテストフレームワークとくらべて簡単すぎるように見えるかもしれませんが、このシェルスクリプトの簡単さは、9cc自身の簡単さとバランスが取れているので、これくらい簡単なほうが望ましいのです。ユニットテストというものは、要は自分の書いたコードを一発で動かして結果を機械的に比較できればよいだけなので、難しく考えすぎず、まずはテストを行うことが大切なのです。

makeによるビルド

本書を通して読者のみなさんは9ccを何百回、あるいは何千回もビルドすることになるでしょう。9ccの実行ファイルを作成して、その後にテストスクリプトを走らせる作業は毎回同じなので、ツールに任せると便利です。こうした用途で標準的に使われているのが`make`コマンドです。

makeは、実行されるとカレントディレクトリの`Makefile`という名前のファイルを読み込んで、そこに書かれているコマンドを実行します。`Makefile`は、コロンで終わるルールと、そのルールのためのコマンドの列という構成になっています。次のMakefileはこのステップで実行したいコマンドを自動化するためのものです。

```
9cc: 9cc.c

test: 9cc
	./test.sh

clean:
	rm -f 9cc *.o *~ tmp*
```

上記のファイルを、`9cc.c`があるのと同じディレクトリに`Makefile`というファイル名で作成してください。そうすると、`make`を実行するだけで9ccが作成され、`make test`を実行するとテストを実行する、ということができるようになります。makeはファイルの依存関係を理解できるので、`9cc.c`を変更した後、`make test`を実行する前に、`make`を実行する必要はありません。9ccという実行ファイルが9cc.cより古い場合に限り、makeは、テストを実行するより前に9ccをビルドしてくれます。

`make clean`というのはテンポラリなファイルを消すルールです。テンポラリファイルは手で`rm`してもよいのですが、消したくないファイルを誤って消してしまうと面倒なので、こういったユーティリティ的なものも`Makefile`に書くことにしています。

なお、`Makefile`を記述する際の注意点ですが、`Makefile`のインデントはタブ文字でなければいけません。スペース4個や8個ではエラーになります。これは単に使い勝手の悪い文法なだけなのですが、makeは1970年代に開発された古いツールで、伝統的にこうなってしまっています。

gitによるバージョン管理

本書ではバージョン管理システムとしてgitを使います。本書を通してコンパイラをステップ・バイ・ステップで作っていくわけですが、そのステップごとに、gitのコミットを作って、コミットメッセージを書くようにしてください。コミットメッセージは日本語で構わないので、実際に何を変更したのかを1行サマリーとしてまとめるようにしてください。1行以上の詳細な説明を書きたいときは、最初の行の次に1行空行を開けて、そのあとに説明を書くようにします。

gitでバージョン管理を行うのはみなさんが手で生成したファイルだけです。9ccを動かした結果として生成されるファイルなどは、同じコマンドを実行すればもう一度生成できるので、バージョン管理対象には入れる必要はありません。むしろ、こういったファイルを入れてしまうとコミットごとの変更点が不必要に長くなるので、バージョン管理から外して、リポジトリに入れないようにする必要があります。

gitでは`.gitignore`というファイルに、バージョン管理から外すファイルのパターンを書くことができます。`9cc.c`があるのと同じディレクトリに、以下の内容で`.gitignore`を作成して、テンポラリファイルやエディタのバックアップファイルなどをgitが無視するように設定しておきましょう。

```
*~
*.o
tmp*
9cc
```

gitを使うのが初めてという人は、gitに名前とメールアドレスを教えておきましょう。ここでgitに教えた名前とメールアドレスがコミットログに記録されます。下は筆者の名前とメールアドレスを設定する例です。読者の皆さんは自分の名前とメールアドレスを設定してください。

```
$ git config --global user.name "Rui Ueyama"
$ git config --global user.email "rui314@gmail.com"
```

gitでコミットを作るためには、まず変更があったファイルを`git add`で追加する必要があります。今回は初回のコミットなので、まず`git init`でgitリポジトリを作成し、その後に、ここまでで作成したすべてのファイルを`git add`で追加します。

```
$ git init
Initialized empty Git repository in /home/ruiu/9cc
$ git add 9cc.c test.sh Makefile .gitignore
```

そのあと`git commit`でコミットします。

```
$ git commit
```

`git commit`を実行するとエディタが立ち上がるので、「整数1つをコンパイルするコンパイラを作成」と1行で書いて、エディタを終了してください。コミットがうまくいったことは以下のように`git log -p`を実行すると確認することができます。

```
$ git log -p
commit 0942e68a98a048503eadfee46add3b8b9c7ae8b1 (HEAD -> master)
Author: Rui Ueyama <rui314@gmail.com>
Date:   Sat Aug 4 23:12:31 2018 +0000

    整数1つをコンパイルするコンパイラを作成

diff --git a/9cc.c b/9cc.c
new file mode 100644
index 0000000..e6e4599
--- /dev/null
+++ b/9cc.c
@@ -0,0 +1,16 @@
+#include <stdio.h>
+#include <stdlib.h>
+
+int main(int argc, char **argv) {
+  if (argc != 2) {
...
```

最後に、ここまでで作成したgitリポジトリをGitHubにアップロードしておきましょう。特にGitHubにアップロードする積極的な理由はないのですが、アップロードしない理由もないですし、GitHubはコードのバックアップとしても役に立ちます。GitHubにアップロードするためには、新規のリポジトリを作って（この例では`rui314`というユーザを使って`9cc`というリポジトリを作成しました）、次のコマンドでそのリポジトリをリモートリポジトリとして追加します。

```
$ git remote add origin git@github.com:rui314/9cc.git
```

その後、`git push`を実行すると、手元のリポジトリの内容がGitHubにプッシュされます。`git push`を実行した後、GitHubをブラウザで開いて、自分のソースコードがアップロードされていることを確認してみてください。

これで第1ステップのコンパイラの作成は完了です。このステップのコンパイラは、コンパイラと呼ぶには簡単すぎるようなプログラムですが、コンパイラに必要な要素をすべて含んだ立派なプログラムです。これから我々はこのコンパイラをひたすら機能拡張していって、まだ信じられないかもしれませんが、立派なCコンパイラに育て上げることになります。まずは最初のステップが完成したことを味わってください。

ステップ2：加減算のできるコンパイラの作成

このステップでは、前のステップで作成したコンパイラを拡張して、`42`といった値だけではなく、`2+11`や`5+20-4`やのような加減算を含む式を受け取れるようにします。

`5+20-4`のような式は、コンパイルする前に計算して、その結果の21をアセンブリに埋め込むこともできますが、それだとコンパイラではなくインタープリタになってしまうので、加減算を実行時に行うアセンブリを出力する必要があります。加算と減算を行うアセンブリ命令は`add`と`sub`です。`add`は、2つのレジスタを受け取って、その内容を加算し、結果を第1引数のレジスタに書き込みます。`sub`は`add`と基本的に同じですが、減算を行います。これの命令を使うと、`5+20-4`は次のようにコンパイルすることができます。

```
.intel_syntax noprefix
.global main

main:
        mov rax, 5
        add rax, 20
        sub rax, 4
        ret
```

上記のアセンブリでは、`mov`でRAXに5をセットし、そのあとRAXに20を足して、そして4を引いています。retが実行される時点でのRAXの値は `5+20-4`すなわち21になるはずです。実行して確認してみましょう。上記のファイルを`tmp.s`に保存してアセンブルし、実行してみます。

```
$ gcc -o tmp tmp.s
$ ./tmp
$ echo $?
21
```

上記のように正しく21が表示されました。

さて、このアセンブリファイルはどのように作成すればいいのでしょうか？ この加減算のある式を「言語」として考えてみると、この言語は次のように定義することができます。

- 最初に数字が1つある
- そのあとに0個以上の「項」が続いている
- 項というのは、+の後に数字が来ているものか、-の後に数字が来ているものである

この定義を素直にCのコードに落としてみると、次のようなプログラムになります。

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

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "引数の個数が正しくありません\n");
    return 1;
  }

  char *p = argv[1];

  printf(".intel_syntax noprefix\n");
  printf(".global main\n");
  printf("main:\n");
  printf("  mov rax, %ld\n", strtol(p, &p, 10));

  while (*p) {
    if (*p == '+') {
      p++;
      printf("  add rax, %ld\n", strtol(p, &p, 10));
      continue;
    }
```

```
    if (*p == '-') {
      p++;
      printf("  sub rax, %ld\n", strtol(p, &p, 10));
      continue;
    }

    fprintf(stderr, "予期しない文字です: '%c'\n", *p);
    return 1;
  }

  printf("  ret\n");
  return 0;
}
```

ちょっと長いプログラムになっていますが、前半部分とretの行は以前と同じです。中間に項を読み込むためのコードが足されています。今回は数字1つを読むだけのプログラムではないので、数字を読み込んだあとに、どこまで読み込んだのかがわからないといけません。atoiでは読み込んだ文字の文字数は返してくれないので、atoiでは項をどこから読めばよいのかわからなくなってしまいます。したがってここでは、C標準ライブラリのstrtol関数を使いました。

strtolは数値を読み込んだ後、第2引数のポインタをアップデートして、読み込んだ最後の文字の次の文字を指すように値を更新します。したがって、数値を1つ読み込んだ後、もしその次の文字が+や-ならば、pはその文字を指しているはずです。上のプログラムではその事実を利用して、whileループの中で次々と項を読んで、1つ項を読むたびにアセンブリを1行出力するということを行なっています。

さて、さっそくこの改造版コンパイラを実行してみましょう。9cc.cファイルを更新したら、makeを実行するだけで新しい9ccファイルを作ることができるのでした。実行例を以下に示します。

```
$ make
$ ./9cc '5+20-4'
.intel_syntax noprefix
.global main
main:
  mov rax, 5
```

```
  add rax, 20
  sub rax, 4
  ret
```

どうやらうまくアセンブリが出力されているようですね。この新しい機能をテストするために、test.shに次のようにテストを1行追加しておきましょう。

```
try 21 '5+20-4'
```

ここまでできたら、ここまでの変更点をgitにコミットしておきましょう。そのためには以下のコマンドを実行します。

```
$ git add test.sh 9cc.c
$ git commit
```

git commitを実行するとエディタが起動するので「足し算と引き算を追加」と書いて保存し、エディタを終了します。git log -pコマンドを使ってコミットが期待した通りに行われていることを確認してみてください。最後にgit pushを実行してGitHubにコミットをプッシュしたら、このステップは完了です！

ステップ3：トークナイザを導入

前のステップで作成したコンパイラには1つ欠点があります。もし入力に空白文字が含まれていたら、その時点でエラーになってしまうのです。例えば以下のように5 - 3という空白の入った文字列を与えると、+あるいは-を読もうとしているところで空白文字を見つけることになり、コンパイルに失敗してしまいます。

```
$ ./9cc '5 - 3' > tmp.s
予期しない文字です: ' '
```

この問題を解決する方法はいくつかあります。1つの自明な方法は、+や-を読もうとする前に空白文字を読み飛ばすことでしょう。このやり方には特に問題があるというわけはないのです

が、このステップでは別の方法で問題を解決することにします。その方法というのは、式を読む前に入力を単語に分割してしまうという方法です。

日本語や英語と同じように、算数の式やプログラミング言語も、単語の列から成り立っていると考えることができます。例えば`5+20-4`は`5`, `+`, `20`, `-`, `4`という5つの単語でできていると考えることができます。この「単語」のことを「トークン」といいます。トークンの間にある空白文字というのは、トークンを区切るために存在しているだけで、単語を構成する一部分ではありません。したがって、文字列をトークン列に分割するときに空白文字を取り除くのは自然なことです。文字列をトークン列に分割することを「トークナイズする」といいます。

文字列をトークン列に分けることには他のメリットもあります。式をトークンに分けるときにそのトークンを分類して型をつけることができるのです。例えば`+`や`-`は、見ての通りの`+`や`-`といった記号ですし、一方で`123`という文字列は123という数値を意味しています。トークナイズするときに、入力を単なる文字列に分割するだけではなく、その1つ1つのトークンを解釈することで、トークン列を消費するときに考えなければならないことが減るのです。

現在の加減算ができる式の文法の場合、トークンの型は、`+`、`-`、数値の3つです。さらにコンパイラの実装の都合上、トークン列の終わりを表す特殊な型を1つ定義しておくとプログラムが簡潔になります（文字列が`'\0'`で終わっているのと同じですね）。

コードを読みやすくするために、1文字のトークンについては、そのトークンのASCIIコードをそのトークンの型とすることにします。したがって、`+`を表すトークンの型は`'+'`、`-`を表すトークンの型は`'-'`です。2文字以上のトークンについてはこのような方法は使えないですし、数値といったトークンもこれで表すことはできないですのですが、そういったトークンはASCIIコードの範囲より大きな数値（256以上）でその型を表すことにします。

やや長くなりますが、トークナイザを導入して改良したバージョンのコンパイラを下に掲載します。

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// トークンの型を表す値
enum {
  TK_NUM = 256, // 整数トークン
  TK_EOF,       // 入力の終わりを表すトークン
};
```

```
// トークンの型
typedef struct {
  int ty;       // トークンの型
  int val;      // tyがTK_NUMの場合、その数値
  char *input; // トークン文字列（エラーメッセージ用）
} Token;

// トークナイズした結果のトークン列はこの配列に保存する
// 100個以上のトークンは来ないものとする
Token tokens[100];

// pが指している文字列をトークンに分割してtokensに保存する
void tokenize(char *p) {
  int i = 0;
  while (*p) {
    // 空白文字をスキップ
    if (isspace(*p)) {
      p++;
      continue;
    }

    if (*p == '+' || *p == '-') {
      tokens[i].ty = *p;
      tokens[i].input = p;
      i++;
      p++;
      continue;
    }

    if (isdigit(*p)) {
      tokens[i].ty = TK_NUM;
      tokens[i].input = p;
      tokens[i].val = strtol(p, &p, 10);
      i++;
      continue;
    }
```

```
    fprintf(stderr, "トークナイズできません: %s\n", p);
    exit(1);
  }

  tokens[i].ty = TK_EOF;
  tokens[i].input = p;
}

// エラーを報告するための関数
void error(int i) {
  fprintf(stderr, "予期しないトークンです: %s\n",
          tokens[i].input);
  exit(1);
}

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "引数の個数が正しくありません\n");
    return 1;
  }

  // トークナイズする
  tokenize(argv[1]);

  // アセンブリの前半部分を出力
  printf(".intel_syntax noprefix\n");
  printf(".global main\n");
  printf("main:\n");

  // 式の最初は数でなければならないので、それをチェックして
  // 最初のmov命令を出力
  if (tokens[0].ty != TK_NUM)
    error(0);
  printf("  mov rax, %d\n", tokens[0].val);

  // `+ <数>`あるいは`- <数>`というトークンの並びを消費しつつ
  // アセンブリを出力
  int i = 1;
```

```
  while (tokens[i].ty != TK_EOF) {
    if (tokens[i].ty == '+') {
      i++;
      if (tokens[i].ty != TK_NUM)
        error(i);
      printf("  add rax, %d\n", tokens[i].val);
      i++;
      continue;
    }

    if (tokens[i].ty == '-') {
      i++;
      if (tokens[i].ty != TK_NUM)
        error(i);
      printf("  sub rax, %d\n", tokens[i].val);
      i++;
      continue;
    }

    error(i);
  }

  printf("  ret\n");
  return 0;
}
```

100行程度のあまり短いとはいえないコードですが、行なっていることにトリッキーなことはないので、上から読んでいけば読めるはずです。コードを読んでみるとわかるように、このコンパイラはエラーチェックが非常に弱くて、長い入力などを与えると簡単にクラッシュしてしまいますが、現在のところはそれで構いません。この程度の完成度のときに最初から頑強に作っても仕方がないので、むしろ意図的にエラーチェックは省いています。読者の皆さんもこの段階ではあまり防御的になりすぎないように気をつけてください。

この改良版では空白文字がスキップできるようになったはずなので、次のようなテストを1行`test.sh`に追加しておきましょう。

```
try 41 " 12 + 34 - 5 "
```

そのあとgitにコミットしてみてください。これでこのステップは完了です。

コラム: ソースコードのフォーマッタ

日本語でも句読点など正書法のレベルで誤りの多い文章が読むに耐えないのと同じように、ソースコードも、インデントがおかしかったり空白の有無などが一貫していなかったりすると、ソースコードの中身以前のレベルできれいなコードとは言えません。コードのフォーマッティングといったいわばどうでもいい部分では、機械的に一定のルールを適用して、気が散らずに読めるコードを書くように気をつけてください。

複数人で開発するときにはどういったフォーマットにするか相談して決めなければいけませんが、この本では一人で開発しているので、ある程度メジャーなフォーマットのなかから自分で好きなフォーマットを選んで構いません。

最近開発された言語では、どういうフォーマットを選ぶかという、好みはわかれるけど本質的ではない議論の必要性そのものをなくすために、言語公式のフォーマッタを提供しているものがあります。たとえばGo言語ではgofmtというコマンドがあり、それを使うとソースコードをきれいに整形してくれます。gofmtはフォーマットのスタイルを選ぶためのオプションがなく、いわば唯一の「Go公式のフォーマット」にしか整形することができません。あえて選択肢を与えないことにより、フォーマットをどうするかという問題をGoは完全に解決しているわけです。

CやC++ではclang-formatというフォーマッタがありますが、本書では特にこういったツールを使うことを推奨したいわけではありません。フォーマットのおかしなコードを書いて後から整形するのではなく、最初から一貫した見た目のコードを書くように気をつけてみてください。

文法の記述方法と再帰下降構文解析

さて、次は乗除算や優先順位のカッコ、すなわち`*`、`/`、`(`)を言語に追加したいのですが、それをするためには1つ大きな技術的チャレンジがあります。掛け算や割り算は式の中で最初に計算しなければいけないというルールがあるからです。例えば`1+2*3`という式は`1+(2*3)`というように解釈しなければいけないのであって、`(1+2)*3`というように解釈することはでき

ません。こういった、どの演算子が最初に「くっつく」のかというルールを「演算子の優先順位」といいます。

演算子の優先順位はどのように処理すればよいのでしょうか？ ここまで作ってきたコンパイラでは、先頭からトークン列を読んでアセンブリを出力していくだけなので、素直にそのまま拡張して*と/を追加すると、1+2*3を(1+2)*3としてコンパイルすることになってしまいます。

既存のコンパイラは当然こういったものをうまく扱えています。コンパイラの構文解析は非常に強力で、どのような複雑なコードでも、文法にそっている限りは正しく解釈することができます。このコンパイラの振る舞いには人間を超える知的な能力すら感じることがありますが、実際には、コンピュータには人間のような文章読解能力はないので、構文解析はなんらかの機械的メカニズムのみによって行われているはずです。具体的にはどういう仕組みで動いているのでしょうか？

この章では、コーディングは一休みにして、構文解析のテクニックについて学んでいきましょう。

文脈自由文法

プログラミング言語の構文の大部分は「生成規則」というものを使って定義されています。生成規則は文法を再帰的に定義するルールです。

自然言語について少し考えてみましょう。日本語において文法は入れ子構造になっています。例えば「花がきれいだ」という文の「花」という名詞を「赤い花」という名詞句に置き換えても正しい文になりますし、「赤い」というのを「少し赤い」というようにさらに展開してもやはり正しい文になっています。「少し赤い花がきれいだと私は思った」というように別の文章の中に入れることもできます。

こういった文法を、「文とは主語と述語からなる」とか「名詞句は名詞か、あるいは形容詞の後に名詞句が続くものからなる」といったようなルールとして定義されているものと考えてみましょう。そうすると「文」を出発点にして、ルールに従って展開していくことで、定義された文法における妥当な文というものをどのようなものでも作り出すことができます。

あるいは逆に、すでに存在している文について、それにマッチする展開手順を考えることで、その文字列がどのような構造を持っているのかどうかを考えることもできます。

元々上記のようなアイデアは自然言語のために考案されたのですが、コンピュータで扱うデータとの親和性がとても高いため、生成規則はプログラミング言語を始めとしてコンピュータの

様々なところで利用されています。

生成規則は「A→α」という形式で書くことができます。この規則は、Aをαに展開できるという意味です。αは0個以上の記号の列で、それ以上展開できない記号と、さらに展開される（いずれかの生成規則で左辺に来ている）記号の両方を複数含むことができます。それ以上展開できない記号を「終端記号」、展開できる記号を「非終端記号」といいます。このような生成規則で定義される文法のことを「文脈自由文法」といいます。

非終端記号は複数の生成規則にマッチしてかまいません。例えばA→α_1とA→α_2の両方の規則があった場合、Aはα_1かα_2のどちらに展開することもできる、という意味になります。A→α_1とA→α_2は、A→α_1 | α_2と省略して書くこともできます。

例として次の生成規則を考えてみてください。

```
add: num
add: add "+" num
add: add "-" num
num: digit
num: num digit
digit: "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

ここでは矢印の代わりにコロンを使いました。ダブルクオートでくくられた文字はその文字自身を表しています。この規則では、digitというのは0〜9の一桁の数字、numは一桁以上の数字の列、addというのはそれと+あるいは-を組み合わせたものということになります。

実は上の生成規則は、加減算の式の文法を定義しています。addから出発して展開していくと、任意の加減算の文字列、例えば`1`や`10+5`や`42-30+2`のような文字列を作り出すことができます。以下の展開結果を確認してみてください。

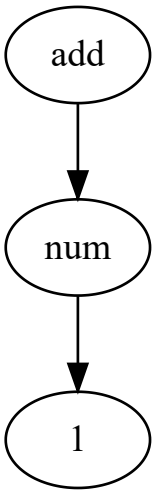
```
add → num → digit → "1"
```

```
add → add "+" num
    → num "+" num
    → num digit "+" num
    → digit digit "+" num
    → digit digit "+" digit
    → "1" digit "+" digit
```

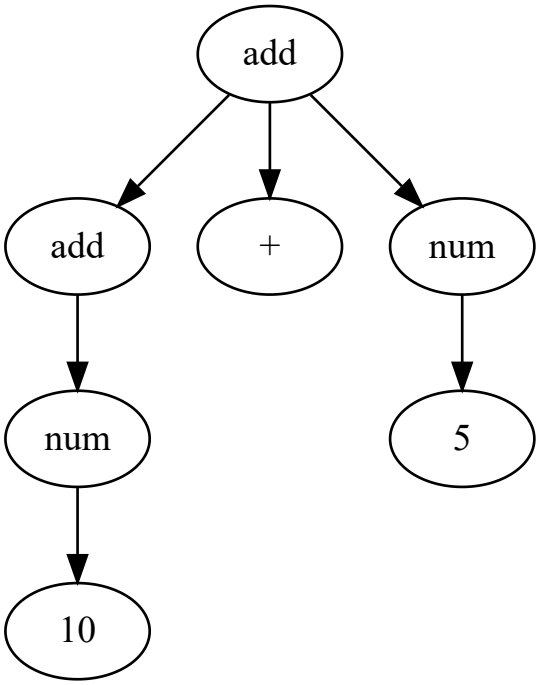
```
    → "1" "0" "+" digit
    → "1" "0" "+" "5"
```

```
add → add "+" num
    → add "-" num "+" num
    → num "-" num "+" num
    → num digit "-" num "+" num
    → digit digit "-" num "+" num
    → digit digit "-" num digit "+" num
    → digit digit "-" digit digit "+" num
    → digit digit "-" digit digit "+" digit
    → "4" "2" "-" "3" "0" "+" "2"  （digitをすべて展開）
```

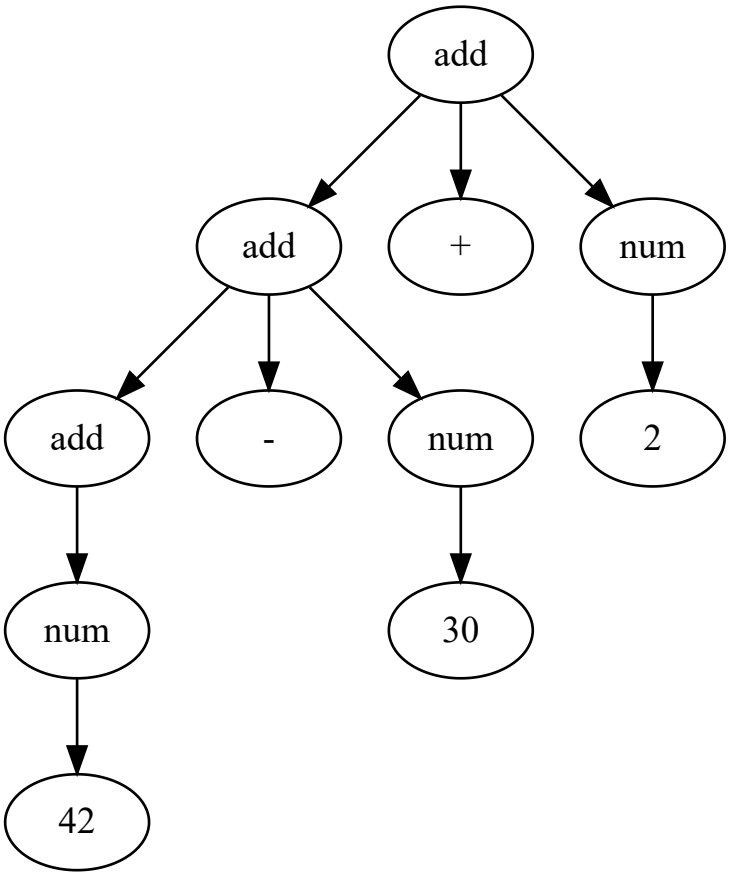
このような展開の手順を、矢印を使ってステップごとに表すだけではなく、木で表すこともできます。そのような木を構文木（syntax tree）といいます。上の式の構文木を以下に示します。なお、numより下のノードを描くと冗長になりすぎるので、以下の図では省略しました。



1の構文木



10+5の構文木



42-30+2の構文木

上の図でわかるように、この文法では、木を深くすることでいくらでも長い加減算の式を作り出すことができます。したがってこの文法は任意の長さの加減算の式を作り出すことができます。

生成規則による演算子の優先順位の表現

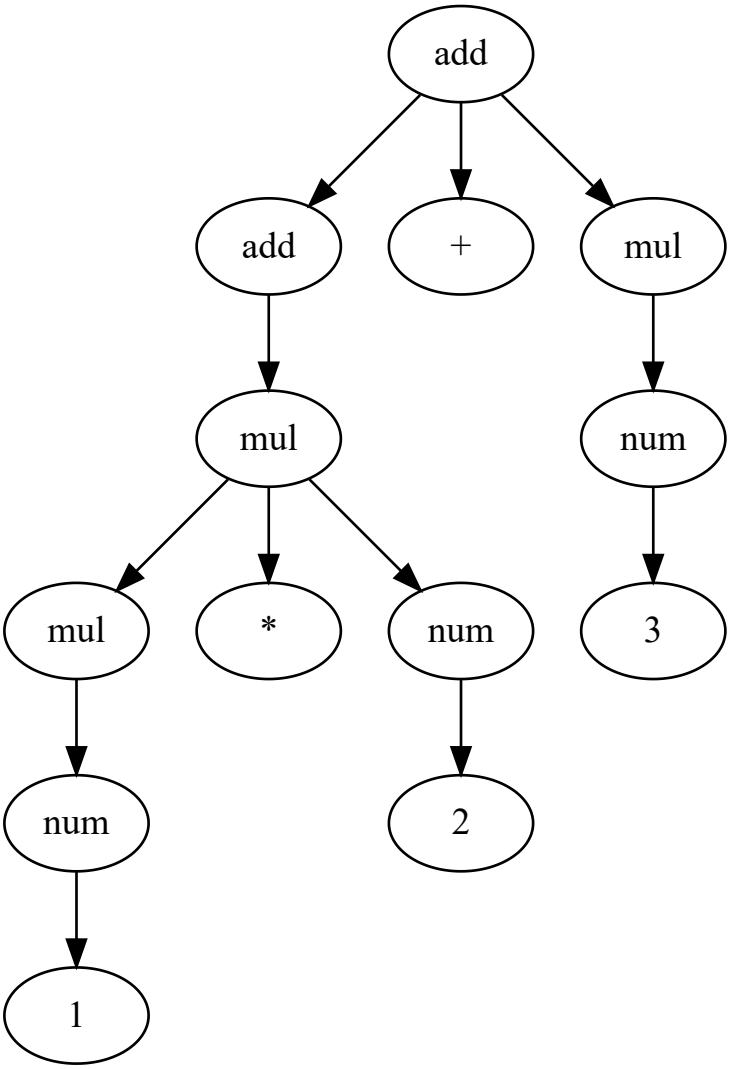
生成規則は文法を表現するための大変強力なツールです。演算子の優先順位（掛け算や割り算は足し算や引き算よりも先に行うといったルール）も、文法を工夫すると、生成規則の中で表すことができます。

日本語でも少し考えてみましょう。「きれいな花がプランターに咲いている」という文では、「きれいな花」がひとかたまりになっていて、プランターがきれいだという誤解が生じることはありません。この文章の構文木を組み立てると、「きれいな」と「花」というのが枝としてひとまとまりになっていて、「プランター」というのは木の中で離れた場所にあるようにできるはずです。

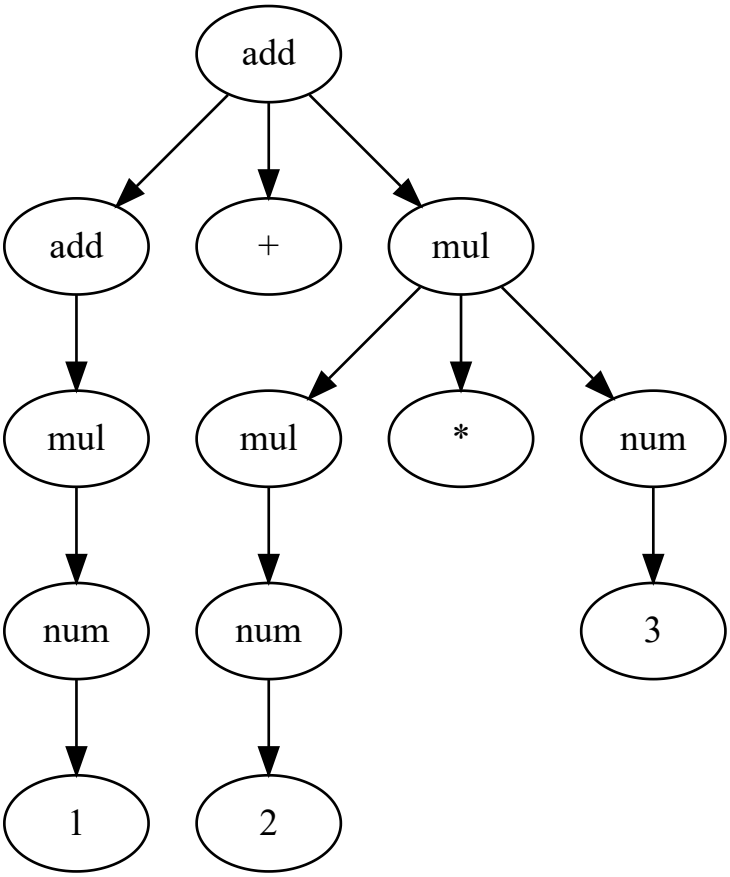
このような考え方を応用して、乗除算を加減算よりも優先する文法などを考案することができます。その文法を以下に示します。

```
add: mul
add: add "+" mul
add: add "-" mul
mul: num
mul: mul "*" num
mul: mul "/" num
```

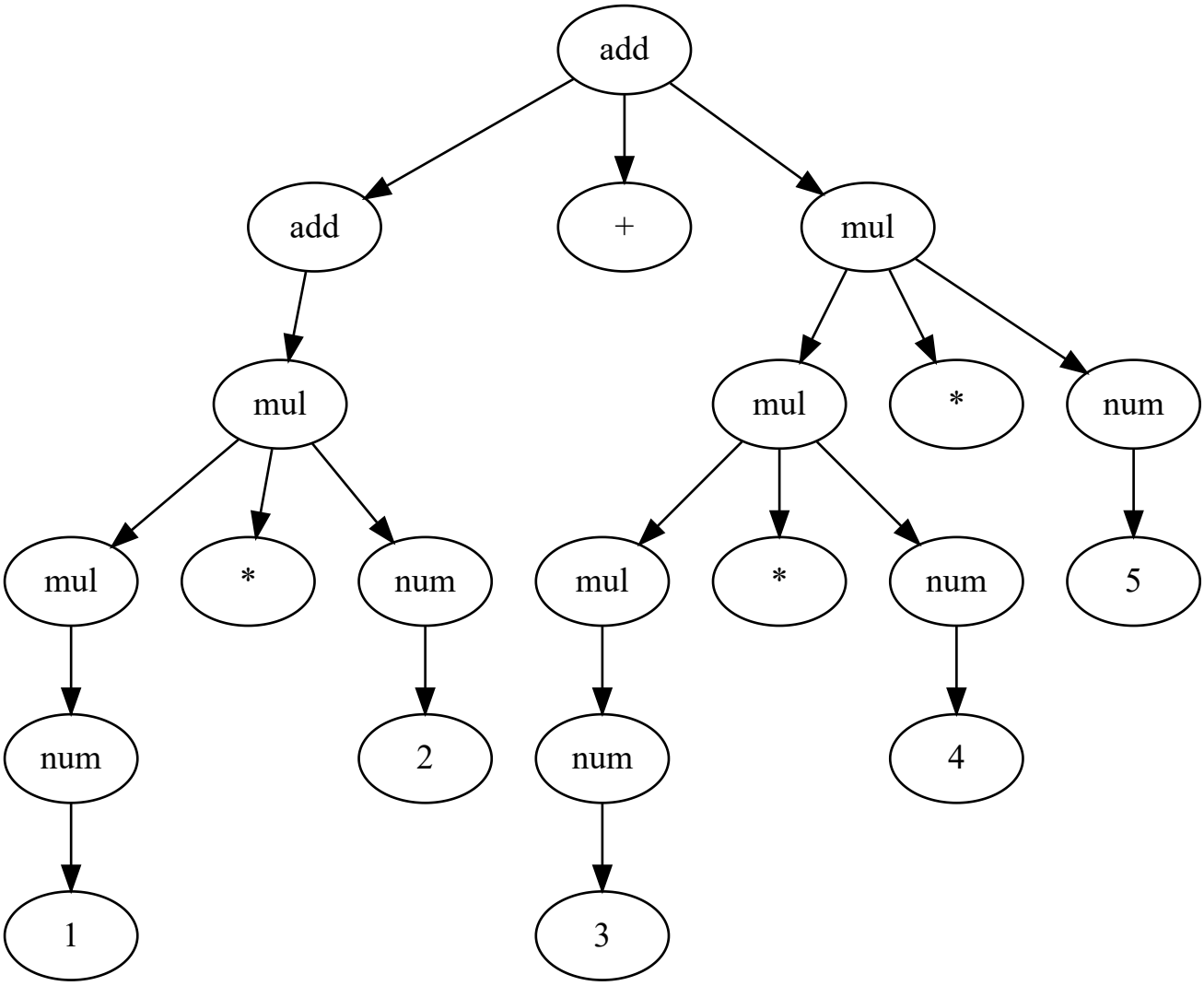
以前はaddが直接numに展開されていたのですが、今回はaddはmulを経由してnumに展開されるルールになりました。mulというのが乗除算の生成規則で、加減算を行うaddは、mulをいわば一つのパーツとして使っています。木の末端から先に計算するというルールを採用した場合、この文法では加減算が先にくっつくというルールが構文木の中で自然と表現されることになります。具体的にいくつか例を見てみましょう。



1*2+3の構文木



1+2*3の構文木



1*2+3*4*5の構文木

上の木構造では、掛け算が足し算より常に木の末端方向に現れるようになっています。実際のところ、mulからaddに戻るルールがないので、掛け算の下に足し算がある木は作りようがないのですが、そうはいってもこのような単純なルールで優先順位が木構造としてうまく表現できるのはかなり不思議に感じます。読者の皆さんも実際に生成規則と構文木を付き合わせて、構文木が正しいことを確認してみてください。

実は上の文法には曖昧性がないので、四則演算の式が与えられるとそれに対応する構文木は一意に決まります。例えば1*2+3*4*5は(1*2)+((3*4)*5)という解釈しか不可能です。この文法では、それ以外の展開順で1*2+3*4*5という式を生成することはできません。

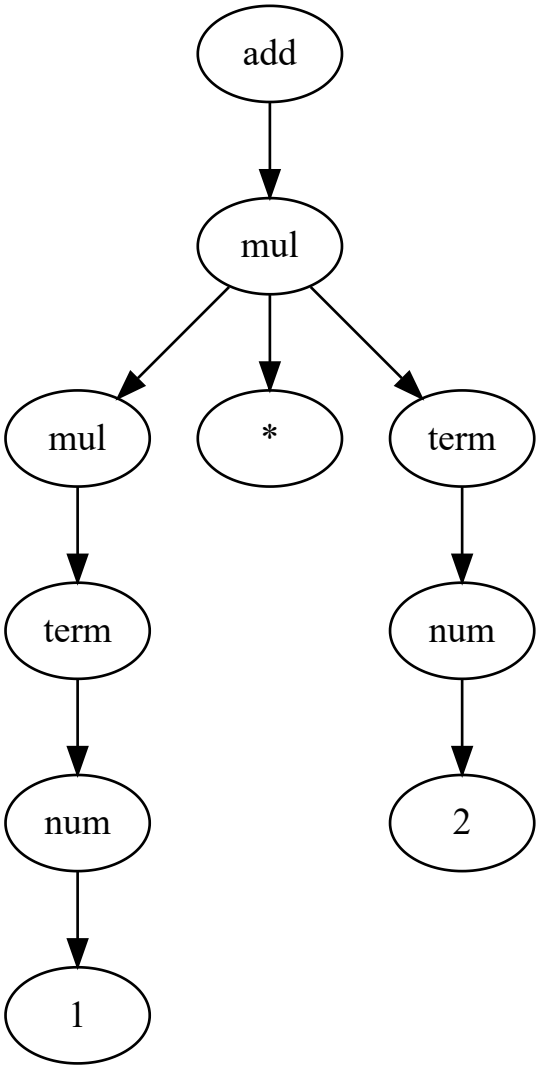
間接的な再帰を含む生成規則

addやmulは右辺にそれ自身が来ていて直接再帰するように定義されていました。生成文法では直接再帰だけではなく間接的に再帰する文法も普通に書くことができます。下は、優先順位のカッコを四則演算に追加した文法の生成規則です。

```
add: mul
add: add "+" mul
add: add "-" mul
mul: term
mul: mul "*" term
mul: mul "/" term
term: num
term: "(" add ")"
```

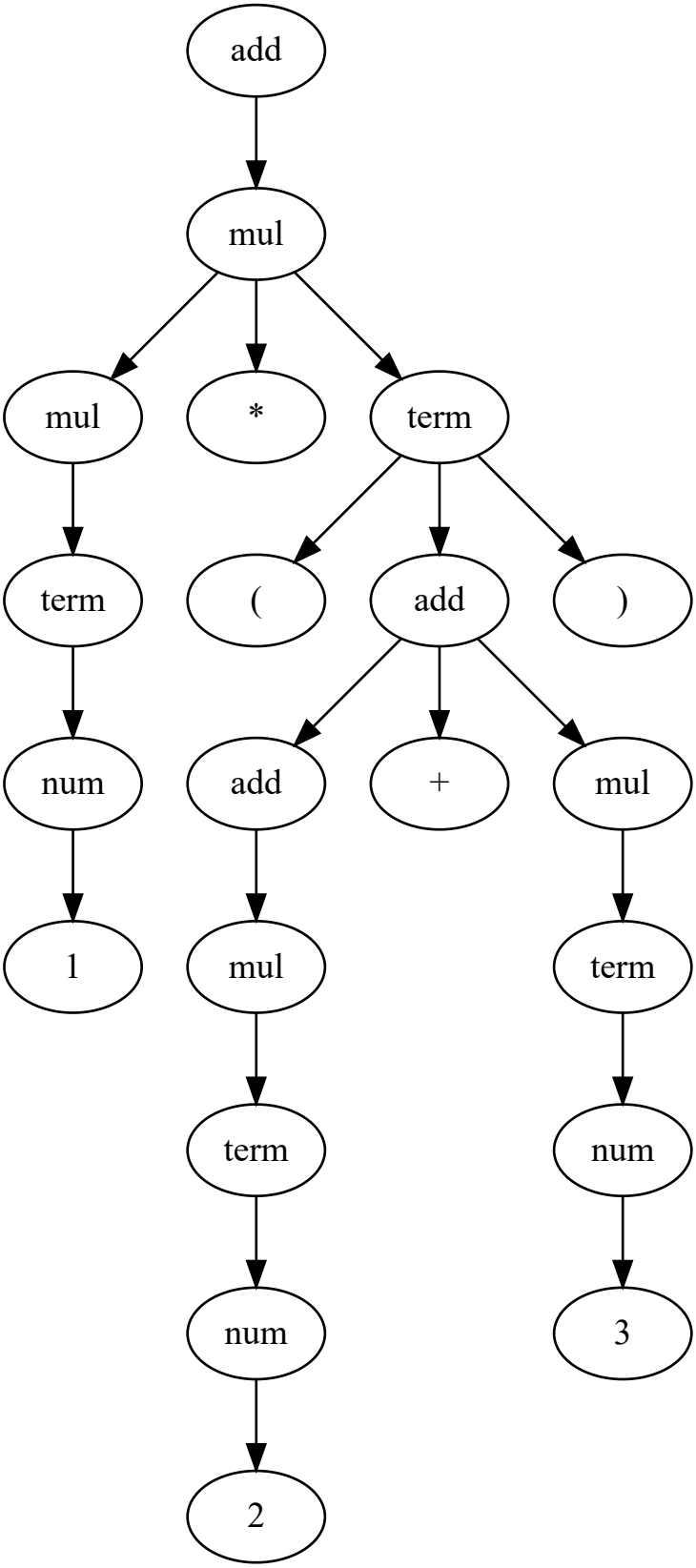
上記の文法を以前の文法と比べてみると、今までnumが許されていたところに、numあるいは"(" add ")"が来てよいことになっています。つまりこの新しい文法では、丸カッコでくくられた式というものは、いままでの単一の数と同じ「くっつき具合」で扱われることになります。一つ例を見てみましょう。

次の木は1*2の構文木です。



1*2の構文木

次の木は1*(2+3)の構文木です。



1*(2+3)の構文木

2つの木を比べてみると、termの展開結果だけが異なることがわかります。展開結果の末端に現れるtermというのは、1つの数字に展開してもよいし、カッコでくくられた任意の式に展開

してもよい、というルールが、木構造の中にきちんと反映されています。このように簡単な生成規則でカッコの優先順位も扱えるというのは少し感動的ではないでしょうか。

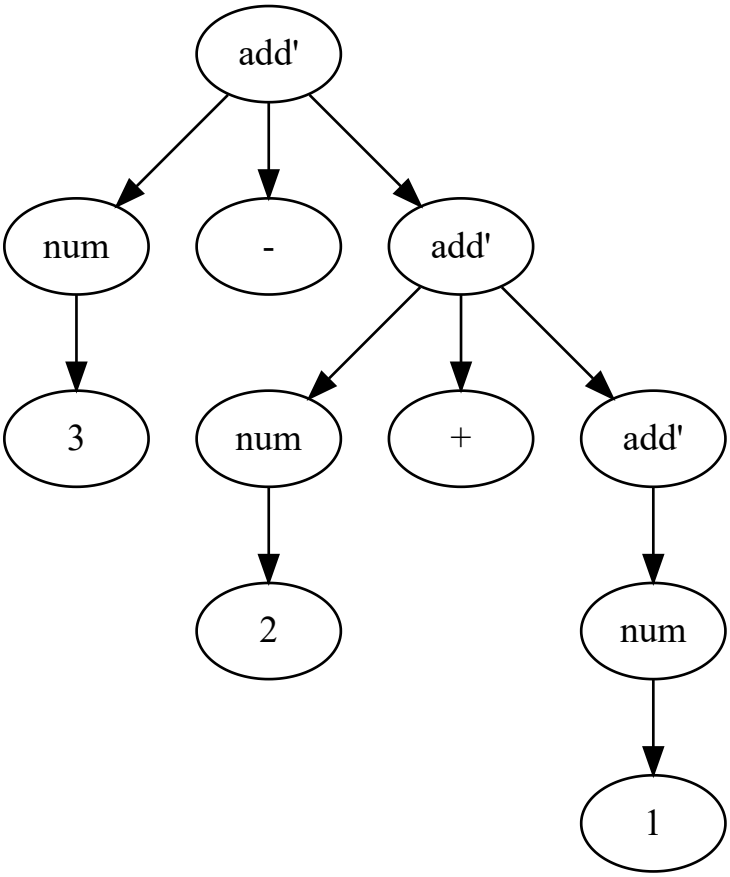
演算子の結合規則

今まで見てきた文法では、同じ優先順位の演算子が複数連続して使われている場合、構文木が左に深くなるようになっていました。解析木がそのような形になる演算子のことを「左結合」の演算子といいます。

今までの例には出てきていませんが、同じ優先順位の演算子が複数連続して使われているときに、構文木が右に深くなるような生成規則を考えることもできます。そのような演算子のことを「右結合」の演算子といいます。例えば次の文法における+と-演算子は右結合になるように定義されています。

```
add': num
add': num "+" add'
add': num "-" add'
```

この文法で実際に構文木を組み立てて、木が右に深くなっていることを確認してみましょう。上記の文法に従って3-2+1の構文木を組み立てると下のようになります。



右結合の3-2+1の構文木

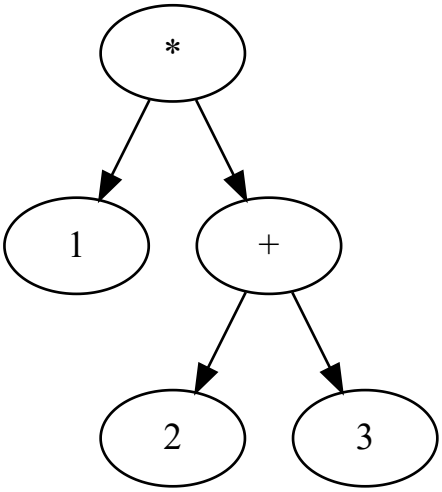
この文法では、2+1がまずグループ化されていて、その結果が3から引かれるということになります。すなわちこの文法では3-2+1は3-(2+1)=0として解釈されます。これは左結合で解釈した場合の計算結果、すなわち(3-2)+1=2とは結果が異なっています。このように、生成規則から生成される言語は完全に同一でも、生成規則の定義によって文の意味が異なってくるということがありえるのです。

左結合と右結合のどちらが正しいのかという問題はただの定義の問題ですが、我々が普通の数式やC言語の式で期待しているのは左結合です。従ってCでは+や-を含むほとんどの演算子は左結合としてパースする必要があります。

Cでは一部、右結合として定義されている演算子も存在します。代入の=演算子がそれです。Cの式a=b=1はa=(b=1)として解釈されるので、まず1がbにセットされ、次にその返り値の1がaにセットされます。左結合として解釈したときの式(a=b)=1はそもそもCとして正当な代入式になっていません。このように、Cでは=演算子を右結合として定義することにより、余分なカッコを書かなくてすむようにしています。

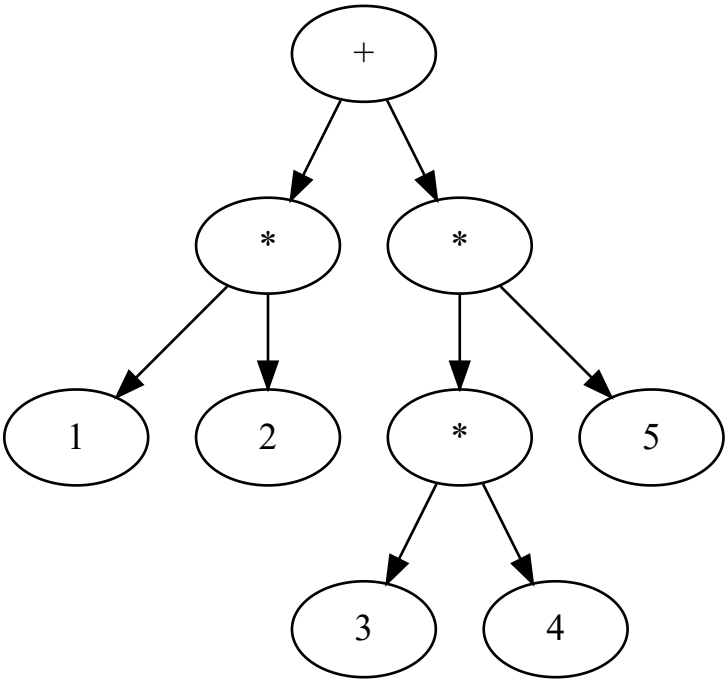
抽象構文木

今まで見てきた木は生成規則をそのまま表しているのでかなり冗長な表現になっています。こういった木は多少切り詰めたり変形しても曖昧にはなりません。例えば1*(2+3)は次の木で簡潔に表すことができます。



簡潔に表現した1*(2+3)の構文木

もう一つ例を挙げてみましょう。以下に示すのは1*2+3*4*5を簡潔に表した木です。



簡潔に表現した1*2+3*4*5の構文木

このようにコンパクトに表した構文木のことを抽象構文木（abstract syntax tree; AST）といいます。コンパイラで構文木を扱うときは、文法に一対一で対応している木よりも、冗長なノードを省いた木のほうが扱いやすいので、通常は構文解析をしつつ抽象構文木を構築していく

ことになります。抽象構文木はコンパイラの内部表現なので実装の都合で適当に定義してかまいません。ここでは算術演算子については二分木になるような抽象構文木を採用しました。

コラム: 算術式のいろいろな記述方法

我々が小学校で習った算術式の書き方のルールは、ある一つのルールにすぎません。別のルールを考えることもできますし、実際に、通常とは異なるルールを導入しているプログラミング言語も存在します。そういった言語をここではいくつか紹介してみましょう。

Forthという言語では、2項演算子を、2つの項の真ん中ではなく後ろに書くというルールを採用しています。そのような記法を「後置記法」といいます。後置記法に対して普通の演算子の書き方を「中置記法」といいます。後置記法では、例えば2+3は2 3 +、2+3*4は2 3 4 * +と記述することになります。後置記法では、中置記法で優先順位のカッコをが必要な式を、カッコを使わずに書けることがあります。例えば(2+3)*4は2 3 + 4 *と記述できます。そのような利点がありつつも、後置記法では、被演算子と演算子を一つの式の中で遠く離れて書くことになりがちで、対応関係がわかりにくく、あまり広く使われていません。

Lispという言語では、後置記法とは逆に演算子を被演算子の前に置いて、すべてを明示的にカッコでくくって記述します。例えば2+3は(+ 2 3)、2+3+4は(+ 2 3 4)と書くことになります。このような記法では、演算子の優先順位といったルールを導入する必要がなく、3項以上を取る演算子を自然に記述することもできますが、式がカッコだらけになりがちで、中置記法ほどの人気はありません。

APLという言語では、すべての演算子の優先順位が同じで、右結合するというルールになっています。例えばAPLでは、2×3+4は2×(3+4)、2×3÷4+5は2×(3÷(4+5))として解釈されます。APLのルールは単純で、それなりの良さがありますが、普通の記法とはかなり異なるルールで、読みにくいと思うひとも多いと思います。（なお、APLは×や÷などのASCII文字セットにはない記号を普通に使っていることも特徴の一つです。）

我々が小学校で習った演算子の優先順位のルールは、算術式の記法が発達してきた1600年代から、数式を扱う人々の間で自然と成立してきたルールです。その時代に+、-、×、÷といった記号が発明されたわけですが、当時の人々にも、乗除算を加減算より先に行うというルールは自然と受け入れられていたようです。そのようなルールがない場合、例えばax^2+bx+cのような多項式は、カッコをたくさん使って$(a(x^2))+(bx)+c$と書かなくてはいけなくなるので、これはリーズナブルなルールと言えるでしょう。C言語では、式の記法については特にチャレンジングなことをすることはなく、「普通の式の書き方」で算術式を書けるようにしています。

再帰下降構文解析

C言語の生成規則が与えられれば、それをどんどん展開していくことで、生成規則の観点からみて正しい任意のCプログラムを機械的に生成することができます。しかし9ccにおいて我々が行いたいことは、むしろ逆のことです。外部から文字列としてCプログラムが与えられていて、展開すると入力の文字列になる展開手順、すなわち入力と同じ文字列になる構文木の構造を知りたいのです。

実はある種の生成規則については、規則が与えられれば、その規則から生成される文にマッチする構文木を求めるコードを機械的に書いていくことができます。例として四則演算の文法を考えてみましょう。四則演算の文法を再掲します。

```
add: mul
add: add "+" mul
add: add "-" mul
mul: term
mul: mul "*" term
mul: mul "/" term
term: num
term: "(" add ")"
```

パーサを書くときの基本的な戦略は、これらの項一つ一つをそのまま関数一つ一つにマップするというものです。したがってパーサは`add`、`mul`、`term`という3つの関数を持つことになります。それぞれの関数は、その名前のとおりの記号列をパースします。

具体的にコードで考えてみましょう。2桁以上のnumは複数の文字からできているわけですが、我々のコンパイラにはすでにトークナイザが導入されているので、複数の数字をまとめて1つの数として扱うことはすでに達成されています。そこで、入力は文字列ではなくトークンの列と見なすことにします。トークンの型を再掲します。

```
typedef struct {
  int ty;   // トークンの型
  int val;  // tyがTK_NUMの場合、その数値
} Token;
```

パーサからは抽象構文木を作って返したいので、抽象構文木のノードの型を定義しておきましょう。

```
enum {
  ND_NUM = 256,      // 整数のノードの型
};

typedef struct Node {
  int ty;            // 演算子かND_NUM
  struct Node *lhs; // 左辺
  struct Node *rhs; // 右辺
  int val;           // tyがND_NUMの場合のみ使う
} Node;
```

lhsとrhsいうのはそれぞれleft-hand sideとright-hand side、すなわち左辺と右辺という意味です。

新しいノードを作成する関数も定義しておきます。この文法における四則演算では、左辺と右辺を受け取る2項演算子と、数値の2種類があるので、その2種類に合わせて関数を2つ用意します。

```
Node *new_node(int ty, Node *lhs, Node *rhs) {
  Node *node = malloc(sizeof(Node));
  node->ty = ty;
  node->lhs = lhs;
  node->rhs = rhs;
  return node;
}

Node *new_node_num(int val) {
  Node *node = malloc(sizeof(Node));
  node->ty = ND_NUM;
  node->val = val;
  return node;
}
```

トークン列はtokensというToken型のグローバル変数に入っているものとします。現在着目しているトークンのインデックスは、int型のposという変数で表します。posの初期値は0、すなわち最初のトークンです。

最後に、便利な関数として、次のトークンが期待した型かどうかをチェックして、期待した型の場合だけ入力を1トークン読み進めて真を返す関数consumeを定義しておきます。

```
int consume(int ty) {
  if (tokens[pos].ty != ty)
    return 0;
  pos++;
  return 1;
}
```

これらの関数とデータ型を使ってパーサを書いていきましょう。addは左結合の演算子として定義されています。左結合の演算子をパーズする関数は、パターンとして次のように書きます。

```
Node *add() {
  Node *node = mul();

  for (;;) {
    if (consume('+'))
      node = new_node('+', node, mul());
    else if (consume('-'))
      node = new_node('-', node, mul());
    else
      return node;
  }
}
```

add関数をよく読んでみてください。addの生成規則、より正確にはそれを変形した次の生成規則が、そのままコードになっていることがわかると思います。

```
add: mul add'
add': ε
```

```
add': "+" mul add'
add': "-" mul add'
```

ここでε（イプシロン）は何もないことを表す記号です。つまりadd'は、空の記号列か、+ mul add'か、- mul add'に展開されるということになります。結果として、add'は、+ mulか- mulが0個以上続いたものということになります。上記のadd関数から返される抽象構文木では、演算子は左結合になっています。

add関数が使っているmul関数も定義してみましょう。mulも左結合の演算子なので、同じパターンで記述することができます。その関数を下に示します。

```
Node *mul() {
  Node *node = term();

  for (;;) {
    if (consume('*'))
      node = new_node('*', node, term());
    else if (consume('/'))
      node = new_node('/', node, term());
    else
      return node;
  }
}
```

最後にterm関数を定義してみましょう。termが読み込むのは左結合の演算子ではないので、上記のパターンのコードにはなりませんが、term → "(" add ")" | numという生成規則をそのまま関数呼び出しに対応させることで、term関数は以下のように記述することができます。

```
Node *term() {
  if (consume('(')) {
    Node *node = add();
    if (!consume(')'))
      error("開きカッコに対応する閉じカッコがありません: %s",
            tokens[pos].input);
    return node;
  }
```

```
  if (tokens[pos].ty == TK_NUM)
    return new_node_num(tokens[pos++].val);

  error("数値でも開きカッコでもないトークンです: %s"
        tokens[pos].input);
}
```

さて、これで全ての関数が揃ったわけですが、これで本当に文字列をパースできるのでしょうか？ 一見よくわからないかもしれませんが、この関数群を使うときちんと文字列がパースできます。例として1+2*3という式を考えてみましょう。

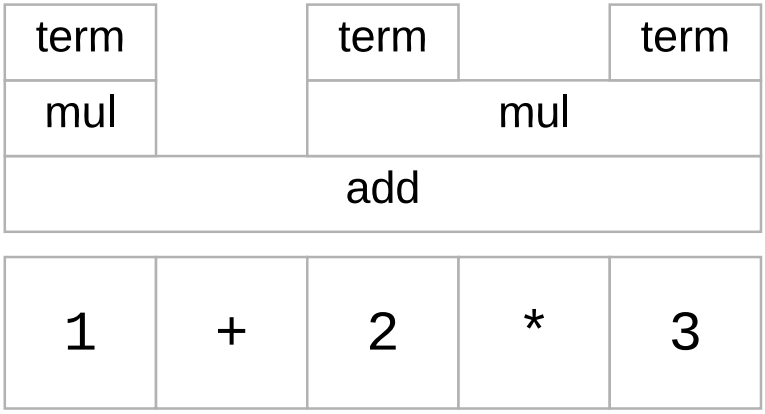
最初に呼ばれるのはaddです。式というのは全体としてaddであると決めつけて（実際にそうなわけですが）入力を読み始めるわけです。そうすると、add→mul→termというように関数呼び出しが行われて、文字列"1"が読み込まれ、addには、返り値として1を表す構文木が返ってきます。

次に、addの中のconsume('+')という式が真になるので、+という文字が消費され、mulが再度呼び出されます。この段階での残りの文字列は2*3です。

mulからは前回と同様にtermが呼び出されて、文字列"2"が読み込まれますが、今回はmulはすぐにはリターンしません。mulの中のconsume('*')という式が真になるので、mulは再度termを呼び出して、文字列"3"を読み込みます。結果としてmulからは2*3を表す構文木が返ることになります。

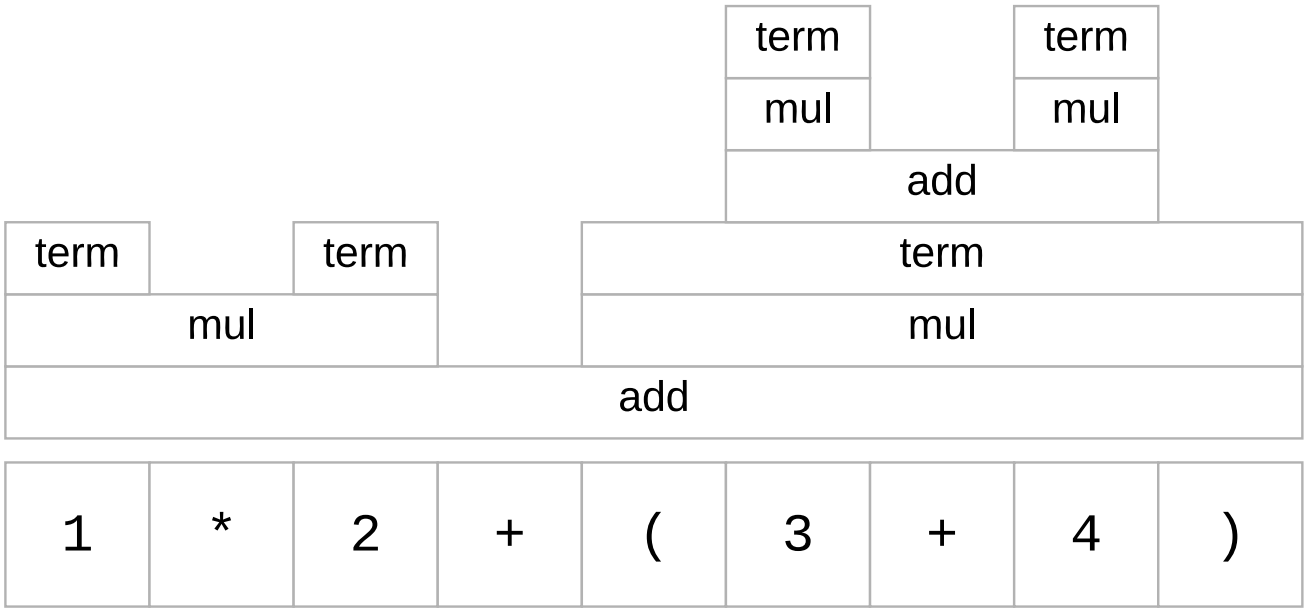
リターンした先のaddでは、1を表す構文木と2*3を表す構文木が組み合わされて、1+2*3を表す構文木が構築され、それがaddの返り値になります。つまり正しく1+2*3がパースできたというわけです。

関数の呼び出し関係とそれぞれの関数が読み込む文字を図に示すと次のようになります。下の図では、1+2*3全体に対応したaddの層がありますが、これが入力全体を読み込むaddの呼び出しを表しています。addの上に2つのmulがありますが、それらは1と2*3を読み込む別のmulの呼び出しを表しています。



1+2*3のコールグラフ

もう少し複雑な例を下に示します。下の図は、1*2+(3+4)をパースしているときの関数の呼び出し関係を表しています。



1*2+(3+4)のコールグラフ

再帰に慣れていないプログラマの場合、上のような再帰的な関数はわかりづらく感じるかもしれません。正直、再帰には非常に慣れているはずの筆者ですら、こういったコードが動くのは一種の魔法のように感じます。再帰的なコードは、仕組みがわかっていてもどこか不思議な感じがするのですが、それはおそらくそういうものなのでしょう。何度もよく頭の中でコードをトレースしてみて、きちんとコードが動作することを確認してみてください。

上記のような1つの生成規則を1つの関数にマップするという構文解析の手法を「再帰下降構文解析」といいます。上記のパーサではトークンを1つだけ先読みして、どの関数を呼び出すか、あるいはリターンするか、ということを決めていましたが、そのようにトークンを1つだ

け先読みする再帰下降パーサのことをLL(1)パーサといいます。また、LL(1)パーサが書ける文法のことをLL(1)文法といいます。我々の元の文法はLL(1)文法ではなかったので、`add`や`mul`では文法をLL(1)になるように変形して、その変形した文法に対して再帰下降パーサを書きました。

スタックマシン

前章ではトークン列を抽象構文木に変換するアルゴリズムについて説明しました。演算子の優先順位を考慮した文法を選ぶことによって、*や/が、+や-に比べて、常に枝の先の方に来ている抽象構文木を作ることができるようになったわけですが、この木をどのようにアセンブリに変換すればよいのでしょうか？ この章ではその方法を説明します。

まずは、加減算と同じ方法ではなぜアセンブリに変換できないのかを説明します。加減算のできるコンパイラでは、RAXを結果のレジスタとして、そこに加算や減算を行っていました。つまりコンパイルされたプログラムでは中間的な計算結果を1つだけ保持していて、それがRAXに入っていました。

しかし、乗除算が含まれる場合は中間的な計算結果が1つだけになるとは限りません。例として2*3+4*5を考えてみてください。足し算を行うためには両辺が計算済みでなければいけないので、足し算の前に2*3と、4*5を計算する必要があります。つまりこの場合は途中の計算結果を2つ保持できなければ全体の計算ができないのです。

こういったものの計算が簡単に行えるのが「スタックマシン」というコンピュータです。ここではいったんパーサの作った抽象構文木から離れて、スタックマシンについて学んでみましょう。

スタックマシンの概念

スタックマシンは、スタックをデータ保存領域として持っているコンピュータのことです。したがってスタックマシンでは「スタックにプッシュする」と「スタックからポップする」という2つ操作が基本操作になります。プッシュでは、スタックの一番上に新しい要素が積まれます。ポップでは、スタックの一番上から要素が取り除かれます。

スタックマシンにおける演算命令は、スタックトップの要素に作用します。例えばスタックマシンのADD命令は、スタックトップから2つ要素をポップしてきて、それらを加算し、その結

果をスタックにプッシュします（x86-64命令との混同を避けるために、仮想スタックマシンの命令はすべて大文字で表記することにします）。別の言い方をすると、ADDは、スタックトップの2つの要素を、それらを足した結果の1つの要素で置き換える命令です。

SUB、MUL、DIV命令は、ADDと同じように、スタックトップの2つの要素を、それらを減算・乗算、除算した1つの要素で置き換える命令ということになります。

PUSH命令は引数の要素をスタックトップに積むものとします。ここでは使用しませんが、スタックトップから要素を1つ取り除くPOPという命令も考えることができます。

さて、これらの命令を使って、2*3+4*5を計算することを考えてみましょう。上のように定義したスタックマシンを使うと、次のようなコードで2*3+4*5を計算することができるはずです。

```
// 2*3を計算
PUSH 2
PUSH 3
MUL

// 4*5を計算
PUSH 4
PUSH 5
MUL

// 2*3 + 4*5を計算
ADD
```

このコードについて少し詳しくみていきましょう。スタックにはあらかじめ何らかの値が入っているものとします。ここではその値は重要ではないので、「...」で表示します。スタックは図において上から下に伸びるものとします。

最初の2つのPUSHが2と3をスタックにプッシュするので、その直後のMULが実行される時点ではスタックの状態は次のようになっています。

...
2
3

MULはスタックトップの2つの値、すなわち3と2を取り除いて、それを掛けた結果、つまり6をスタックにプッシュします。したがってMULの実行後にはスタックの状態は次のようになります。

...
6

次にPUSHが4と5をプッシュするので、2番目のMULが実行される直前にはスタックは次のようになっているはずです。

...
6
4
5

ここでMULを実行すると、5と4が取り除かれて、それを掛けた結果の20に置き換えられます。したがってMULの実行後には次のようになります。

...
6
20

2*3と4*5の計算結果がうまくスタックに入っていることに着目してください。この状態でADDを実行すると、20+6が計算され、その結果がスタックにプッシュされるので、最終的にスタックは次の状態になるはずです。

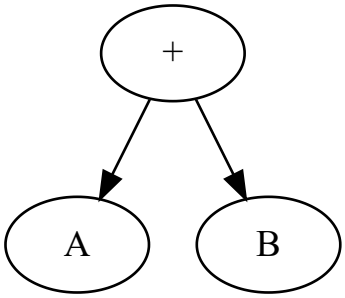
...
26

スタックマシンの計算結果はスタックトップに残っている値ということにすると、26は2*3+4*5の結果ですから、きちんとその式が計算できたことになるわけです。スタックマシンではこの式に限らず、複数の途中結果を持つどのような式でも計算することができます。スタックマシンを使うと、どのような部分式も、それを実行した結果として1つの要素をスタックに結果として残すという約束を守っている限り、上記の方法でうまくコンパイルできるのです。

スタックマシンへのコンパイル

さて、最後に、抽象構文木をスタックマシンのコードに変換する方法について説明しましょう。それができるようになれば、四則演算からなる式をパースして抽象構文木を組み立て、それをx86-64命令を使ったスタックマシンにコンパイルして実行することができるようになります。つまり四則演算のできるコンパイラが書けるようになるというわけです。

スタックマシンでは、部分式を計算すると、それが何であれその結果の1つの値がスタックトップに残るということになっていました。例えば下のような木を考えてください。



加算を表す抽象構文木

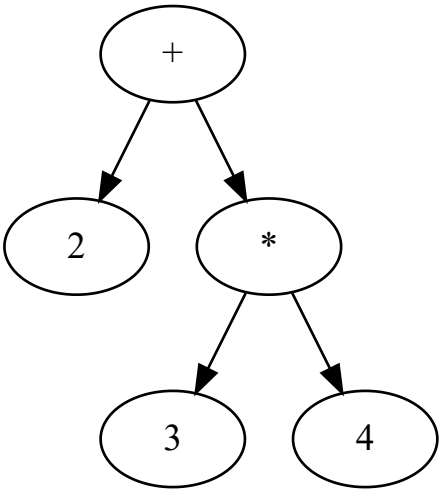
AやBというのは部分木を抽象化して表したもので、実際にはなんらかの型のノードを意味しています。しかしその具体的な型や木の形は、この木全体をコンパイルするときには重要ではありません。この木をコンパイルするときは次のようにすればよいのです。

1. 左の部分木をコンパイルする
2. 右の部分木をコンパイルする
3. スタックの2つの値を、それらを加算した結果で置き換えるコードを出力

1のコードを実行した後には、その具体的なコードが何であれ、左の部分木の結果を表す1つの値がスタックトップに置かれているはずです。同様に、2のコードを実行した後には、右の部分木の結果を表す1つの値がスタックトップに置かれているはずです。したがって、木全体の値を計算するためには、その2つの値を、その合計値で置き換えればよいというわけです。

このように、抽象構文木をスタックマシンにコンパイルするときは、再帰的に考えて、木を下りながらどんどんアセンブリを出力していくことになります。再帰の考え方に慣れていない読者にとってはやや難しく思えるかもしれませんが、木構造を扱う時には再帰を使うのはセオリーです。木のように自己相似形のデータ構造を扱う時には、部分も全体も同じように扱う必要があるので、同じ関数を全体に対しても、部分に対しても、部分のさらに一部分についても、同じように使うということになるのです。

以下の例に具体的に考えてみましょう。



加算を表す抽象構文木

コード生成を行う関数は木のルートのノードを受け取ります。

上記の手順に従うと、その関数がまず行うのは左の部分木をコンパイルすることです。つまり数値の2をコンパイルすることになります。2を計算した結果はそのまま2ですから、その部分木のコンパイル結果はPUSH 2です。

次にコード生成関数は右の部分木をコンパイルしようとします。そうすると再帰的に部分木の左側をコンパイルすることになり、結果としてPUSH 3が出力されます。次は部分木の右側をコンパイルすることになり、PUSH 4が出力されます。

そのあとコード生成関数は再帰呼び出しを元に戻りながら、部分木の演算子の型に合わせたコードを出力していきます。最初に出力されるのは、スタックトップの2つの要素を、それらを掛けたもので置き換えるコードです。その次にスタックトップの2つの要素を、それらを足したもので置き換えるコードが出力されます。結果として下のアセンブリが出力されることになります。

```
PUSH 2
PUSH 3
PUSH 4
MUL
ADD
```

このような手法を使うと、抽象構文木を機械的にアセンブリに落としていけるのです。

x86-64におけるスタックマシンの実現方法

ここまでは仮想的なスタックマシンの話でした。実際のx86-64はスタックマシンではなくレジスタマシンです。x86-64の演算は通常2つのレジスタ間に対して定義されており、スタックトップの2つの値に対して動作するように定義されているわけではありません。したがって、スタックマシンのテクニックをx86-64で使うためには、レジスタマシンでスタックマシンをある意味でエミュレートする必要があります。

レジスタマシンでスタックマシンをエミュレートするのは比較的簡単です。スタックマシンで1命令になっているものを複数の命令を使って実装すればよいのです。

そのための具体的な手法を説明しましょう。

まずスタックの先頭の要素を指すレジスタを1つ用意しておきます。そのレジスタのことをスタックポインタといいます。スタックトップの2つの値をポップしてきたいのであれば、スタックポインタの指す要素を2つ取り出して、スタックポインタを取り出した要素のぶんだけ変更しておきます。同じように、プッシュするときは、スタックポインタの値を変更しつつそれが指しているメモリ領域に書き込めばよいというわけです。

x86-64のRSPレジスタはスタックポインタとして使うことを念頭に置いて設計されています。x86-64の`push`や`pop`といった命令は、暗黙のうちにRSPをスタックポインタとして使って、その値を変更しつつ、RSPが指しているメモリにアクセスする命令です。したがって、x86-64命令セットをスタックマシンのように使うときは、RSPをスタックポインタとして使うのが素直です。では早速、1+2という式を、x86-64をスタックマシンと見立ててコンパイルしてみましょう。以下にx86-64のアセンブリを示します。

```
# 左辺と右辺をプッシュ
push 1
push 2

# 左辺と右辺をRAXとRDIにポップして足す
pop rdi
pop rax
add rax, rdi

# 足した結果をスタックにプッシュ
push rax
```

x86-64には「RSPが指している2つの要素を足す」という命令はないので、いったんレジスタにロードして加算を行い、その結果をスタックにプッシュし直す必要があります。上記の`add`命令で行っているのはそういう操作です。

同様に2*3+4*5をx86-64で実装してみると次のようになります。

```
# 2*3を計算して結果をスタックにプッシュ
push 2
push 3

pop rdi
pop rax
mul rax, rdi
push rax

# 4*5を計算して結果をスタックにプッシュ
push 4
push 5

pop rdi
pop rax
mul rax, rdi
push rax

# スタックトップの2つの値を足す
# つまり2*3+4*5を計算する
pop rdi
pop rax
add rax, rdi
push rax
```

このように、x86-64のスタック操作命令を使うと、x86-64であってもかなりスタックマシンに近いコードを動かすことができます。

次のgen関数はこの手法をそのままCの関数で実装したものです。

```
void gen(Node *node) {
  if (node->ty == ND_NUM) {
    printf("  push %d\n", node->val);
    return;
  }
```

```
  gen(node->lhs);
  gen(node->rhs);

  printf("  pop rdi\n");
  printf("  pop rax\n");

  switch (node->ty)
  case '+':
    printf("  add rax, rdi\n");
    break;
  case '-':
    printf("  sub rax, rdi\n");
    break;
  case '*':
    printf("  mul rdi\n");
    break;
  case '/':
    printf("  mov rdx, 0\n");
    printf("  div rdi\n");
  }

  printf("  push rax\n");
}
```

特にパースやコード生成において重要なポイントではないのですが、トリッキーな仕様の`mul`と`div`命令が上のコードでは使われているので、それについて説明しておきましょう。

x86-64の`mul`が素直な仕様になっていれば、上のコードでは本来`mul rax, rdi`のように書きたかったところですが、そのような2つのレジスタをとる乗算命令はx86-64には存在しません。その代わりに、x86-64の`mul`は、暗黙のうちにRAXを取って、それを引数のレジスタの値に掛けて、その結果をRAXにセットする、という仕様になっています。従って上のコードでは`mul`には1つのレジスタしか渡していません。

`mul`と同様に、`div`は暗黙のうちにRDXとRAXを取って、それを連結したものを128ビット整数とみなして、それを引数のレジスタの64ビットの値で割り、その結果をRAXにセットする、という仕様になっています。RDXをまずゼロクリアしているのはそのためです。

さて、これでスタックマシンの説明は終わりです。ここまで読み進めたことによって、読者のみなさんは複雑な構文解析と、その構文解析の結果得られた抽象構文木をマシンコードに落と

すことができるようになったはずです。その知識を活用するために、コンパイラ作成の作業に戻ってみましょう！

コラム: 最適化コンパイラ

この章で筆者が説明に使ったx86-64のアセンブリはかなり非効率的に見えるかもしれません。例えばスタックに数値を`push`してそれを`pop`する命令は、直接レジスタにその値を`mov`する命令で書けば1命令で済むはずです。読者の中には、そういったアセンブリから冗長さを取り除いて最適化したいという気持ちが湧き上がってきている人もいることでしょう。しかし、その誘惑には決して負けないようにしてください。一番最初のコード生成では、コンパイラの実装の容易さを優先して冗長なコードを出力するのは、望ましいことなのです。

9ccには必要ならば後から最適化パスを付け足すことができます。生成されたアセンブリを再度スキャンして、特定のパターンで現れている命令列を別の命令列で置き換えることは難しくありません。例えば「`push`直後の`pop`は`mov`に置き換える」とか「連続している`add`が、即値を同じレジスタに足している場合、その即値を合計した値を足す1つの`add`に置き換える」といったルールを作って、それを機械的に適用すれば、冗長なコードを、意味を変えることなくより効率的なコードに置き換えることができます。

コード生成と最適化を混ぜてしまうとコンパイラが複雑になってしまいます。最初から難しいコードになってしまうと、後から最適化パスを足すのはむしろ困難です。Donald Knuthが言っていたように「早すぎる最適化は全ての悪の元凶」なのです。読者の皆さんが作成するコンパイラでも、実装の簡単さだけを考慮するようにしてください。出力に含まれる明白な冗長さは後から取り除けるので心配する必要はありません。

ステップ4：四則演算のできる言語の作成

この章では、前章までに作ってきたコンパイラを変更して、優先順位のカッコを含む四則演算の式を扱えるように拡張します。必要なパーツは揃っているので、新たに書くコードはほんのわずかです。コンパイラの`main`関数を変更して、新しく作成したパーサとコードジェネレータを使うようにしてみてください。下のようなコードになるはずです。

```
int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "引数の個数が正しくありません\n");
    return 1;
  }

  // トークナイズしてパースする
  tokenize(argv[1]);
  Node *node = add();

  // アセンブリの前半部分を出力
  printf(".intel_syntax noprefix\n");
  printf(".global main\n");
  printf("main:\n");

  // 抽象構文木を下りながらコード生成
  gen(node);

  // スタックトップに式全体の値が残っているはずなので
  // それをRAXにロードして関数からの返り値とする
  printf("  pop rax\n");
  printf("  ret\n");
  return 0;
}
```

この段階まで進んだことで、加減乗除と優先順位のカッコからなる式が正しくコンパイルできるようになっているはずです。いくつかテストを追加しておきましょう。

```
try 47 "5+6*7"
try 15 "5*(9-6)"
try 4 "(3+5)/2"
```

なお、ここまでは説明の都合上、一気に*、/、()を実装しているような話の流れになっていますが、実際には、一気に実装することは避けてください。あくまでインクリメンタルに、1コミットで1つの機能を、その実装をテストするテストコード付きで追加するようにしてください。元々は加減算ができる機能があったわけですから、まずはその機能を壊さずに、抽象構文木とそれを使ったコードジェネレータを導入するようにしてみてください。そのときには新

たな機能を足すわけではないので、新しいテストは必要ありません。その後に、*、/、( ) を、1コミットごとに、テスト込みで実装していってください。

コラム: 9ccにおけるメモリ管理

読者は本書をここまで読んだところで、このコンパイラにおけるメモリ管理がどうなっているのか不思議に思っているかもしれません。ここまでに出てきたコードでは、mallocは使っていますが、freeは呼んでいません。つまりアロケートしたメモリは解放されません。これは、いくらなんでも手抜きではないでしょうか？

実際にはこの「メモリ管理を行わないことをメモリ管理ポリシーとする」という設計は、いろいろなトレードオフを考慮した上で、筆者が意図的に選択したデザインです。このデザインチョイスには多くのメリットがあります。

まず、メモリを解放しないことによって、まるでガベージコレクタがある言語のようにコードを書けるという点があります。これにより、メモリ管理を行うコードを書かなくてよくなるだけではなく、手動メモリ管理にまつわる不可解なバグを元から断つことができます。

次に、freeをしないことによって発生する問題というのは、普通のPCのようなコンピュータで動かすことを考えると、実質的にあまり存在しません。コンパイラは1つのCファイルを読み込んでアセンブリを出力するだけの短命なプログラムです。プログラム終了時に確保されているメモリはOSによってすべて自動的に解放されます。したがって、トータルでどれくらいメモリを割り当てるかということだけが問題になるわけですが、筆者の実測ではかなり大きなCファイルをコンパイルしたときでもメモリ使用量は100 MiB程度にすぎません。したがってfreeしないというのは現実的に有効な戦略なのです。例えばD言語のコンパイラDMDも、同じ考えから、mallocだけを行いfreeはしないというポリシーを採用しています。

ステップ5：任意長の入力のサポート

いままではトークンの個数が100個固定になっていて、それを超える長いプログラムを与えるとコンパイラがクラッシュするという仕様になっていました。最初から細部まで作り込み過ぎないほうがよいという意味で、手抜きなのはむしろ望ましい仕様だったのですが、そろそろそ

ういう制限を直してもよい段階に達したと言えるでしょう。したがってこの章ではまず可変長ベクタを実装し、その後それを使って100トークン以上の入力をサポートします。

可変長ベクタ

ここで作る可変長ベクタには`void`のポインタを入れるものとします。`void`のポインタは他の型のポインタをなんでも保持することができます。`int`も、`void *`にキャストしてに`int`にキャストし直すと元の整数に戻るので、やや無理やりですが、ここで作るベクタには整数も入れることができます。

可変長ベクタは実際のデータを保持するバッファに加えて、バッファの大きさと、そのうち何この要素を使っているのかという情報を保持する必要があります。可変長ベクタの構造体を以下に示します。

```
typedef struct {
  void **data;
  int capacity;
  int len;
} Vector;
```

`data`はデータ本体で、`capacity`というのはバッファの大きさです。`data[0]`から`data[capacity - 1]`までがバッファの領域ということになります。lenというのはベクタに追加済みの要素の個数を表しています。`len == capacity`のときにバッファは一杯で、それ以上要素を足すには、新たにバッファを確保して既存の要素をコピーし、dataポインタをすげ替える必要があります。

ベクタのために`9cc.c`に書くべき関数は以下の2つだけです。

```
Vector *new_vector() {
  Vector *vec = malloc(sizeof(Vector));
  vec->data = malloc(sizeof(void *) * 16);
  vec->capacity = 16;
  vec->len = 0;
  return vec;
}
```

```
void vec_push(Vector *vec, void *elem) {
  if (vec->capacity == vec->len) {
    vec->capacity *= 2;
    vec->data = realloc(vec->data, sizeof(void *) * vec-
  >capacity);
  }
  vec->data[vec->len++] = elem;
}
```

`data`や`len`は構造体のメンバに直接アクセスすればよいので、それらに対するアクセッサは特に定義する必要はありません。

データ構造のユニットテスト

この節で作成したベクタについても当然テストを書きたいところですが、これらの機能は外部に直接アクセスできる形で実装されていないので、単にコマンドを実行してみるという方法ではテストできません。従ってこれらのテストはCで書くことにします。具体的には、9ccの一部分としてテストコードを書いて、`./9cc -test`と実行したときに限り、コンパイラ本体ではなくそのテスト関数を呼ぶことにします。もう一つテスト用に`main`関数を作ってバイナリを分ける方法もあるのですが、それは現時点ではやや大げさなので、ここでは同じバイナリに入れる方法を採用しました。

テストコードを以下に示します。このコードを9cc.cに追加してください。

```
#include "9cc.h"
#include <stdio.h>
#include <stdlib.h>

int expect(int line, int expected, int actual) {
  if (expected == actual)
    return;
  fprintf(stderr, "%d: %d expected, but got %d\n",
          line, expected, actual);
  exit(1);
}

void runtest() {
```

```
  Vector *vec = new_vector();
  expect(__LINE__, 0, vec->len);

  for (int i = 0; i < 100; i++)
    vec_push(vec, (void *)i);

  expect(__LINE__, 100, vec->len);
  expect(__LINE__, 0, (int)vec->data[0]);
  expect(__LINE__, 50, (int)vec->data[50]);
  expect(__LINE__, 99, (int)vec->data[99]);

  printf("OK\n");
}
```

テストコードはこれくらいの簡単さで十分です。上記のコードで使っている`__LINE__`という識別子は、Cプリプロセッサのマクロで、その識別子の現れている行番号を表します。これにより、テストが失敗した時、上のテストコードの何行目のテストが失敗しているのかすぐにわかるので便利というわけです。

上のコードを9cc.cに加えたら、`argv[1]`が`-test`の時に`runtest`関数を呼ぶように`main`関数に変更を加えてください。最後に`Makefile`に以下の変更を加えて、`make test`としたときに`./test.sh`だけではなく`./9cc -test`も走らせるようにすれば、テストは完成です。

```
test: 9cc
	./9cc -test
	./test.sh
```

ベクタを使う

上記で定義した可変長ベクタを使って任意の長さの入力をサポートできるようにしてください。

[要加筆]

ステップ6：1文字のローカル変数

前章までで、四則演算ができる言語のコンパイラを作ることができました。この章では、その言語に機能を追加して、変数を使えるようにします。具体的には次のように変数を含む複数の文をコンパイルできるようになることが目標です。

```
a = 3;
b = 5 * 6 - 8;
a + b / 2;
```

一番最後の式の結果をプログラム全体の計算結果とすることにします。これの言語は、四則演算だけの言語に比べると、かなり「本物の言語」のような雰囲気が出てきていると言えるのではないでしょうか？

この章では、まず変数をどのように実装すればよいのかについて説明を行い、その後、インクリメンタルに変数を実装していくことにします。

スタック上の変数領域

Cにおける変数はメモリ上に存在します。変数はメモリのアドレスに名前をつけたものと言ってもよいでしょう。メモリアドレスに名前をつけることにで、「メモリの0x60a0番地にアクセスする」というように表現するのではなく、「変数aにアクセスする」というように表現することができるようになります。

関数のローカル変数は、関数呼び出しごとに別々に存在しなければいけません。実装の都合だけを考えると、例えば「関数fのローカル変数aは0x60a0番地に置く」というようにアドレスを決め打ちにしてしまうのが簡単そうですが、それだとfを再帰的に呼び出した場合にうまく動きません。従って、ローカル変数を関数呼び出しごとに別々に持たせるために、Cではローカル変数はスタックに置くことになっています。

スタックの内容を具体的な例を挙げて考えてみましょう。ローカル変数aとbを持つ関数fがあり、別の何らかの関数がfを`call`命令で呼び出したとします。`call`はリターンアドレスをスタックに積むので、fが呼ばれた時点のスタックトップは、そのリターンアドレスが入っていることになります。それ以外にも、元々スタックには何らかの値が入っているものとします。ここでは具体的な値は重要ではないので「...」で表すことにします。図にすると次のようになります。

…	
リターンアドレス	← RSP

ここでは「←RSP」という表記で、現在のRSPレジスタの値がこのアドレスを指しているということを表すことにします。aとbのサイズはそれぞれ8バイトとします。

この状態からaとbの領域を確保するためには、変数2個分、つまり合計で16バイトRSPを押し下げる必要があります。それを行うと次のようになります。

…	
リターンアドレス	
a	
b	← RSP

上記のようなレイアウトにすると、RSP+8の値を使うとaに、RSPの値を使うとbに、それぞれアクセスできるということになります。このように関数呼び出しごとに確保されるメモリ領域のことを「関数フレーム」や「アクティベーションレコード」といいます。

基本的にローカル変数というのは、このような単純なものとして実装されています。

ただし、この方法は一つ落とし穴があるので、実際の実装にはもう一つレジスタを使うことになります。我々のコンパイラでは（そしてそのほかのコンパイラでも）、関数を実行している間にRSPが変更されることがあることを思い出してください。9ccは式の途中の計算結果をRSPを使ったスタックにプッシュ／ポップしているので、RSPの値は頻繁に変更されます。したがって、aやbにはRSPからの固定のオフセットでアクセスすることができません。

これを解決するためには、RSPとは別に、現在の関数フレームの開始位置を常に指しているレジスタを用意します。そのようなレジスタを「ベースレジスタ」、そこに入っている値のことを「ベースポインタ」と呼びます。x86-64では慣習としてRBPレジスタをベースレジスタとして使用します。

関数実行中にはベースポインタは変化してはいけません（それこそがベースポインタを用意する理由です）。関数から別の関数を呼び出して、戻ってきたら別の値になっていた、というのではダメですから、関数呼び出しごとに元のベースポインタを保存しておいて、リターンする前に書き戻す必要があります。

ベースポインタを使った関数呼び出しにおけるスタックの状態を示したのが以下の図です。この図では、関数gがf呼んだものと仮定しています。

gのリターンアドレス	
gの呼び出し時点のRBP	
gのローカル変数	
...	
fのリターンアドレス	
fの呼び出し時点のRBP	← RBP
a	
b	← RSP

このようにすると、aにはRBP-8、bにはRBP-16というアドレスで常にアクセスすることができます。具体的にこのようなスタックの状態を作るアセンブリを考えてみると、それぞれの関数の冒頭に、以下のようなアセンブリをコンパイラが出力すればよいということになります。

```
push rbp
mov rbp, rsp
sub rsp, 16
```

このような、コンパイラが関数の先頭に出力する定型の命令のことを「プロローグ」といいます。

RSPがリターンアドレスを指している状態から上のコードを実行すると、期待している通りの関数フレームができあがることを確認してみてください。なお、16というのは、実際には関数ごとに変数の個数やサイズに合わせた値にする必要があります。

関数からリターンするときには、RBPに元の値を書き戻して、RSPがリターンアドレスを指している状態にして、`ret`命令を呼びます（`ret`命令はスタックからアドレスをポップして、そこにジャンプする命令です）。それを行うコードは以下のように簡潔に書くことができます。

```
mov rsp, rbp
pop rbp
ret
```

このような、コンパイラが関数の末尾に出力する定型の命令のことを「エピローグ」といいます。

上のコードの最初の2つの命令を実行したときのスタックの状態を以下に示します。RSPが指しているアドレスより下のスタック領域は、もはや無効なデータとみなしてよいので、図では省略しました。

1. `mov rsp, rbp`を実行した後のスタック

gのリターンアドレス	
gの呼び出し時点のRBP	
gのローカル変数	
...	
fのリターンアドレス	
fの呼び出し時点のRBP	← RSP, RBP

2. `pop rbp`を実行した後のスタック

gのリターンアドレス	
gの呼び出し時点のRBP	← RBP
gのローカル変数	
...	
fのリターンアドレス	← RSP

このようにしてローカル変数というものは実現されているのです。

トークナイザの変更

変数をどのように実装すればよいのかがわかったところで、早速実装してみましょう。ただし任意の個数の変数をサポートするのは急に難しくなりすぎるので、このステップにおける変数は小文字1文字に限定することにして、変数aはRBP-8、変数bはRBP-16、変数cはRBP-24、というように、すべての変数が常に存在するものとします。アルファベットは26文字あるので、関数を呼び出すときに26×8すなわち208バイト分RSPを押し下げることにすると、すべての1文字変数の領域を確保できることになります。

では早速実装してみましょう。まずはトークナイザに手を加えて、今までの文法要素の他に、一文字の変数をトークナイズできるようにします。そのためには新たなトークンの型を追加す

る必要があります。変数名はinputメンバーから読むことができるので、特にToken型に新たにメンバーを足す必要はありません。結果として、トークンの型は次のようになります。

```
enum {
  TK_NUM = 256, // 整数トークン
  TK_IDENT,     // 識別子
  TK_EOF,       // 入力の終わり表すトークン
};
```

トークナイザに変更を加えて、アルファベットの小文字ならば、TK_IDENT型のトークンを作成するようにしてください。次のようなif文をトークナイザに加えれば良いはずです。

```
if ('a' <= *p && *p <= 'z') {
  tokens[i].ty = TK_IDENT;
  tokens[i].input = p;
  i++;
  p++;
  continue;
}
```

パーサの変更

再帰下降構文解析では文法さえわかれば機械的に関数呼び出しにマップできるのでした。したがって、パーサに加えるべき変更を考えるためには、変数名（識別子）を加えた新たな文法がどうなっているのかを考えてみる必要があります。

変数名（すなわち識別子）をidentとしましょう。これはnumと同じように終端記号です。変数というのは数値が使えるところではどこでも使えるので、numだったところをnum | identというようにすると、数値の代わりに変数が使える文法になります。

それに加えて、文法に代入式を足す必要があります。変数は代入できないと仕方がないので、a=1のような式を許す文法にしたいというわけです。ここではCにあわせて、a=b=1のように書ける文法にしておきましょう。

さらに、セミコロン区切りで複数の文（ステートメント）を書けるようにしたいので、結果として新しい文法は以下のようになります。

```
program: stmt program
program: ε

stmt: assign ";"

assign: add
assign: add "=" assign

add: mul
add: add "+" mul
add: add "-" mul

mul: term
mul: mul "*" term
mul: mul "/" term

term: num
term: ident
term: "(" add ")"
```

まずは`42;`や`a=b=2;a+b;`のようなプログラムがこの文法に合致していることを確認してみてください。そのあと、ここまでで作成したパーサに手を入れて、上記の文法をパースできるようにしてください。この段階では`a+1=5`のような式もパースできてしまいますが、それは正しい動作です。そのような意味的に不正な式の排除は次のパスで行います。パーサを改造することについては、特にトリッキーなところはなく、いままでと同じように文法要素をそのまま関数呼び出しにマップしていけばできるはずです。

セミコロン区切りで複数の式をかけるようにしたので、パースの結果として複数のノードをどこかに保存する必要があります。いまのところは次のグローバルな配列を用意して、そこにパース結果のノードを順にストアするようにしてください。最後のノードはNULLで埋めておくと、どこが末尾かわかるようになります。codeの変数定義と、プログラム全体を読み込む再帰下降パーサのコードを以下に示します。

```
Node *code[100];

void program() {
  int i = 0;
```

```
  while (tokens[pos].ty != TK_EOF)
    code[i++] = stmt();
  code[i] = NULL;
}

Node *stmt() {
  Node *node = assign();
  if (!consume(';'))
    error("';'ではないトークンです: %s", tokens[pos].input);
}
```

抽象構文木では新たに「識別子を表すノード」を表現できるようになる必要があります。そのために、識別子の新しい型と、ノードの新しいメンバーを追加しましょう。例えば次のようになるはずです。パーサでは、識別子トークンに対してND_IDENT型のノードを作成して返すことになります。

```
enum {
  ND_NUM = 256,      // 整数のノードの型
  ND_IDENT,          // 識別子のノードの型
};

typedef struct Node {
  int ty;            // 演算子かND_NUM
  struct Node *lhs; // 左辺
  struct Node *rhs; // 右辺
  int val;           // tyがND_NUMの場合のみ使う
  char name;         // tyがND_IDENTの場合のみ使う
} Node;
```

lvalueとrvalue

代入式はそれ以外の二項演算子とは違って、左辺の値を特別に扱う必要があるので、それについてここで説明しておきましょう。

代入式の左辺はどのような式でも許されているというわけではありません。例えば1=2というように1という数に2を代入することはできません。a=2のような代入は許されていますが、(a+1)=2のような文は不正です。9ccにはまだポインタや構造体は存在していませんが、もし存在しているとしたら、*p=2やa.b=2のような代入は正当なものとして許さなければいけません。このような正当な式と不正な式の区別はどのようにつければよいのでしょうか？

その問いには単純な答えがあります。Cにおいて代入式の左辺にくることができるのは、メモリのアドレスを指定する式だけです。

変数というのはメモリに存在していてアドレスを持っているので、変数は代入の左辺に書くことができます。同様に、*pのようなポインタ参照も、pの値をアドレスとみなすという話なので、これも左辺に書くことができます。a.bのような構造体のメンバアクセスも、メモリ上に存在する構造体aの開始位置からbというメンバのオフセット分進んだメモリアドレスを指しているので、左辺に書くことができます。

左辺に書くことができる値のことをlvalue（left value、左辺値）、そうではない値のことをrvalue（right value、右辺値）といいます。現在の我々の言語では、変数のみがlvalueで、それ以外の値はすべてrvalueです。

変数のコード生成を行う際はlvalueを起点に考えることができます。代入の左辺として変数が現れている場合は、左辺の値として変数のアドレスを計算するようにして、そのアドレスに対して右辺の評価結果をストアします。それ以外のコンテキストで変数が現れている場合は、同じように変数のアドレスを計算したあとに、そのアドレスから値をロードすることにより、lvalueをrvalueに変換します。

任意のアドレスから値をロードする方法

ここまでのコード生成ではスタックトップのメモリにしかアクセスしていませんでしたが、ローカル変数ではスタック上の任意の位置にアクセスする必要があります。ここではメモリアクセスの方法について説明します。

CPUはスタックトップだけではなくメモリの任意のアドレスから値をロードしたりストアすることができます。

メモリから値をロードするときは、`mov dst, [src]`という構文を使います。この命令は「srcレジスタの値をアドレスとみなしてそこから値をロードしdstに保存する」という意味です。例えば`mov rdi, [rax]`ならば、RAXに入っているアドレスから値をロードしてRDIにセットするということになります。

ストアするときは、`mov [dst], src`という構文を使います。この命令は「dstレジスタの値をアドレスとみなして、srcレジスタの値をそこにストアする」という意味です。例えば`mov [rdi], rax`ならば、RAXの値を、RDIに入っているアドレスにストアするということになります。

`push`や`pop`は暗黙のうちにRSPをアドレスとみなしてメモリアクセスをする命令なので、実はこれらは普通のメモリアクセス命令を使って複数の命令で書き直すことができます。つまり、例えば`pop rax`は

```
mov rax, [rsp]
add rsp, 8
```

という2つの命令と同じですし、`push rax`は

```
sub rsp, 8
mov [rsp], rax
```

という2つの命令と同じです。

コードジェネレータの変更

ここまでの知識を使って、変数を含む式を扱えるようにコードジェネレータに変更を加えてみましょう。今回の変更では式をlvalueとして評価するという関数を追加することになります。下のコードにおける`gen_lval`という関数はそれを行なっています。`gen_lval`は、与えられたノードが変数を指しているときに、その変数のアドレスを計算して、それをスタックにプッシュします。それ以外の場合にはエラーを表示します。これにより`(a+1)=2`のような式が排除されることになります。

変数をrvalueとして使う場合は、まずlvalueとして評価したあと、スタックトップにある計算結果をアドレスとみなして、そのアドレスから値をロードします。コードを下に示します。

```
void gen_lval(Node *node) {
  if (node->ty != ND_IDENT)
    error("代入の左辺値が変数ではありません");
```

```
  int offset = ('z' - node->name + 1) * 8;
  printf("  mov rax, rbp\n");
  printf("  sub rax, %d\n", offset);
  printf("  push rax\n");
}

void gen(Node *node) {
  if (node->ty == ND_NUM) {
    printf("  push %d\n", node->val);
    return;
  }

  if (node->ty == ND_IDENT) {
    gen_lval(node);
    printf("  pop rax\n");
    printf("  mov rax, [rax]\n");
    printf("  push rax\n");
    return;
  }

  if (node->ty == '=') {
    gen_lval(node->lhs);
    gen(node->rhs);

    printf("  pop rdi\n");
    printf("  pop rax\n");
    printf("  mov [rax], rdi\n");
    printf("  push rdi\n");
    return;
  }

  gen(node->lhs);
  gen(node->rhs);

  printf("  pop rdi\n");
  printf("  pop rax\n");

  switch (node->ty)
```

```
  case '+':
    printf("  add rax, rdi\n");
    break;
  case '-':
    printf("  sub rax, rdi\n");
    break;
  case '*':
    printf("  mul rax, rdi\n");
    break;
  case '/':
    printf("  mov rdx, 0\n");
    printf("  div rdi\n");
  }

  printf("  push rax\n");
}
```

メイン関数の変更

さて、すべてのパーツが揃ったところでmain関数も変更して、コンパイラを実際に動かしてみましょう。

```
int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "引数の個数が正しくありません\n");
    return 1;
  }

  // トークナイズしてパースする
  // 結果はcodeに保存される
  tokenize(argv[1]);
  program()

  // アセンブリの前半部分を出力
  printf(".intel_syntax noprefix\n");
```

```
  printf(".global main\n");
  printf("main:\n");

  // プロローグ
  // 変数26個分の領域を確保する
  printf("  push rbp\n");
  printf("  mov rbp, rsp\n");
  printf("  sub rsp, 208\n");

  // 先頭の式から順にコード生成
  for (int i = 0; code[i]; i++) {
    gen(code[i]);

    // 式の評価結果としてスタックに一つの値が残っている
    // はずなので、スタックが溢れないようにポップしておく
    printf("  pop rax\n");
  }

  // エピローグ
  // 最後の式の結果がRAXに残っているのでそれが返り値になる
  printf("  mov rsp, rbp\n");
  printf("  pop rbp\n");
  printf("  ret\n");
  return 0;
}
```

ステップ7：複数文字のローカル変数

前章では変数名を1文字に決め打ちにして、aからzまでの26個のローカル変数が常に存在しているものとして扱いました。この章では、任意の長さの変数名を使えるようにします。変数の個数も任意の個数ということにします。

マップ

変数名を管理するために、まずは文字列をキーにできるマップを実装しましょう。普通のプログラミング言語やライブラリではマップはハッシュテーブルとして実装されています。ハッシュテーブルは高速にルックアップできる優れたデータ構造ですが、実装が大規模になるので、9ccで使うのには適していません。したがってここでは、連想配列によってマップを実装することにします。

連想配列というのはキーと値のペアが入ったベクタのことです。マップのデータ構造を以下に示します。

```
typedef struct {
  Vector *keys;
  Vector *vals;
} Map;
```

マップのために9cc.cに追加すべき関数は次の3つです。

```
Map *new_map() {
  Map *map = malloc(sizeof(Map));
  map->keys = new_vector();
  map->vals = new_vector();
  return map;
}

void map_put(Map *map, char *key, void *val) {
  vec_push(map->keys, key);
  vec_push(map->vals, val);
}

void *map_get(Map *map, char *key) {
  for (int i = map->keys->len - 1; i >= 0; i--)
    if (strcmp(map->keys->data[i], key) == 0)
      return map->vals->data[i];
  return NULL;
}
```

ルックアップするときは、キーを順番にスキャンして、一致するものを探します。マップに値を追加するときは、既存の同じキーが存在している場合はそれを上書きし、そうでない場合は新規にキーを追加するというのが普通ですが、連想配列の場合は特にそうする必要はありませ

ん。新しいキーから古いキーに向かってスキャンするようにすれば（リストの末尾から先頭に向かって一致するかどうかを試していけば）、同じキーの新しい値は、自然と古い値を隠すようになります。

マップのテストは、ベクタのテストと同じように行うことができます。マップのテストを行う関数`test_map`を以下に示します。

```
void test_map() {
  Map *map = new_map();
  expect(__LINE__, 0, (int)map_get(map, "foo"));

  map_put(map, "foo", (void *)2);
  expect(__LINE__, 2, (int)map_get(map, "foo"));

  map_put(map, "bar", (void *)4);
  expect(__LINE__, 4, (int)map_get(map, "bar"));

  map_put(map, "foo", (void *)6);
  expect(__LINE__, 6, (int)map_get(map, "foo"));
}
```

現在は`runtest`関数に直接ベクタのテストコードが書かれていると思いますが、そのコードを`test_vector`関数に移して、`runtest`から`test_vector`と`test_map`を呼ぶようにしてください。最後にテストがうまく動いていることを`make test`を実行して、このステップは完了です。

この節で定義したマップはかなり素朴な実装ですが、今後のコンパイラ開発を進めていく上で十分な機能を持っています。

複数文字のローカル変数をサポート

[要加筆]

分割コンパイルとリンク

この段階までは、Cファイルとテストのシェルスクリプトがそれぞれ1つだけというファイル構成で開発を進めてきました。この構成に問題があるというわけではないのですが、だんだんソースが長くなってきているので、このあたりで複数のCファイルに分割して見通しをよくすることにしましょう。このステップでは、9cc.cという1つのファイルを、以下の5つのファイルに分割します。

- `9cc.h`: ヘッダファイル
- `main.c`: `main`関数
- `parse.c`: パーサ
- `codegen.c`: コードジェネレータ
- `container.c`: ベクタ、マップ、およびそのテストコード

`main`関数は小さいので他のCファイルに入れてもよかったのですが、意味的に`parse.c`と`codegen.c`のどちらにも属さないので、別のファイルに分けることにしました。

`9cc.h`というのはヘッダファイルです。プログラムの構成によっては1つの`.c`ファイルごとに1つの`.h`ファイルを用意することもありますが、余分な宣言があっても特に害をなすことはないので、ここではそこまで細かな依存関係の管理をする必要はありません。`9cc.h`というファイルを一つ用意して、すべてのCファイルで`#include "9cc.h"`というようにインクルードしてください。

リンク

分割コンパイルにおいて複数のCファイルがどのように実行ファイルに変換されるのかを理解しておきましょう。

CコンパイラはCファイルをアセンブリファイルにコンパイルします。アセンブラはアセンブリファイルを読んで、`.o`という拡張子のファイルを出力します。拡張子が`.o`のファイルのことを「オブジェクトファイル」といいます。

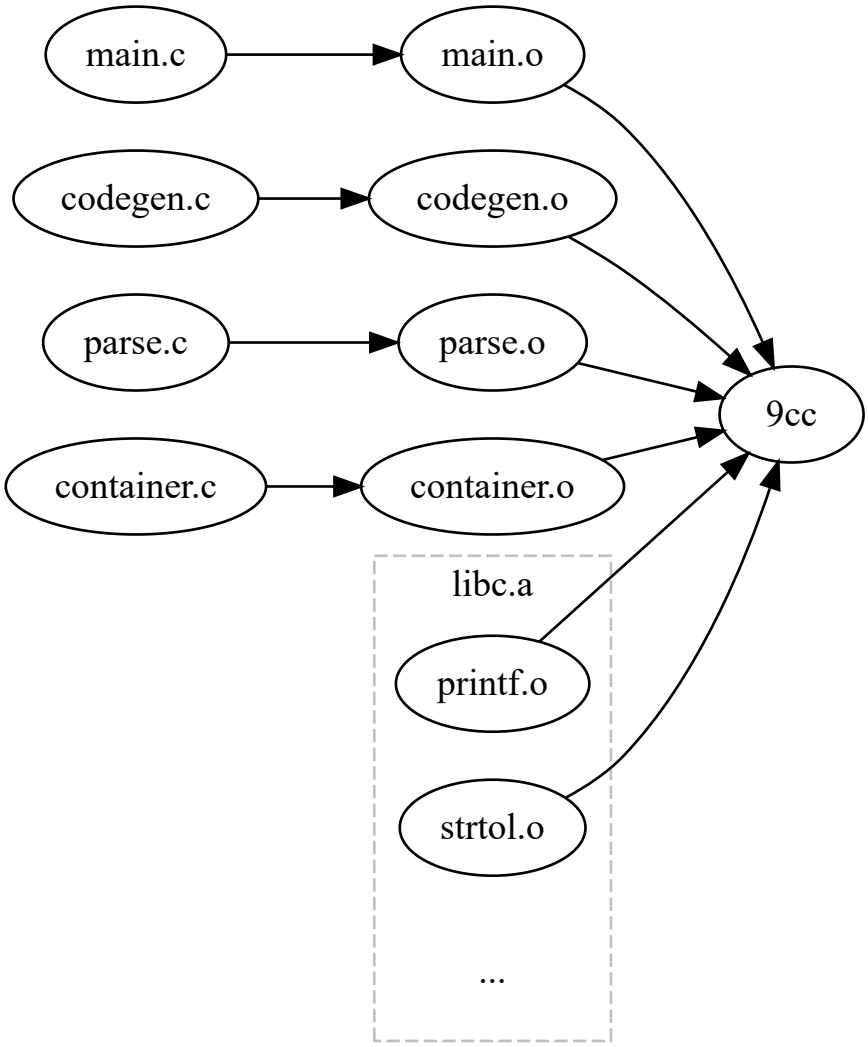
オブジェクトファイルにはコンパイラがコード生成を行なった結果が含まれています。例えば、元々のCファイルで定義されていた関数や変数はオブジェクトファイルにも含まれています。一方で、別のCファイルで定義されていて、元々のCファイルでは単に使っているだけの関数や変数のコードは、オブジェクトファイルには入っていません（Cコンパイラはその関数や変数の名前だけ知っていて、実際のコードのことは知らないので、コードを出力しようがあ

りません）。そういう情報は、コンパイル時には解決できないので、アセンブラは出力ファイルを生成するときに、未解決の名前とそのオブジェクトファイル上での位置をオブジェクトファイルに記録しておいて、後で修正してもらうようにします。

複数のオブジェクトファイルをまとめて実行ファイルを作成するのが「リンカ」と呼ばれるプログラムです。リンカは、オブジェクトファイルを連結して、未解決の名前を解決し、実行ファイルを作成します。リンカが行う動作のことを「リンク」するといいます。

リンカは未解決の名前が最後まで残ってしまったり、逆に複数のオブジェクトファイルが同じ名前を定義しているときに、それをエラーとして扱います。リンクに失敗することを「リンクエラー」といいます。

下の図は、9ccという実行ファイルが作成されるまでの流れを表したものです。



オブジェクトファイルへのコンパイルとリンク

.cを.oに変換するのはコンパイラとアセンブラの役目です。普通のコンパイラではバックグラウンドでアセンブラを起動していたり、アセンブラを内蔵していたりするので、gccコマン

ド1つで.cから.oへの変換ができているように見えますが、概念的にはCからアセンブリへのコンパイルとアセンブルとは分割することができます。

複数の.oファイルをまとめて9ccという実行ファイルにするのはリンカの役目です。リンカはユーザが明示的に与えたオブジェクトファイルをリンクするだけではありません。実はリンカには、暗黙のうちにlibc、すなわちC標準ライブラリが渡されていて、printfなどの関数はそこからリンクされることになります。printfやstrtolなどの関数は、標準で配布されているライブラリに含まれているというだけが特別なのであって、その実態は誰かが普通にCで書いただけのコードです。したがって、ユーザが書いたコードのオブジェクトファイルと同じように、最終的なプログラムには、標準ライブラリに含まれるそれらのオブジェクトファイルもリンクされなければいけないのです。

C標準ライブラリのオブジェクトファイルは、基本的に1つの.oファイルに1つの関数が入っているという形になっています。こうすることによって、不要なコードはファイルごとリンカによって無視されるので、使っていない関数が出力されるファイルに入ってしまうということを避けることができます。ファイルがたくさんあると冗長なので、ライブラリでは、.aという形式のアーカイブファイル（.zipや.tarのようなものです）に複数のオブジェクトファイルがまとめて入れられているのが普通です。

何をヘッダファイルに書けばよいのか

Cファイルに書くべきものと、ヘッダファイルに書くべきものの区別というのは、Cの初心者や中級者にとってはややわかりづらいようです。したがってここではそれについて説明をします。

まず端的に例を示しましょう。ヘッダファイルに書いてよいのは次のようなものです。

```
int plus(int x, int y);
extern int var;
struct StructTag { ... };
typedef struct { ... } FooType;
enum { ... };
```

逆に次のようなものはヘッダファイルに書いてはいけません。

```
int plus(int x, int y) {
  ...
```

```
}

int var;
struct StructTag foo;
FooType bar;
```

次の2行のコードを比べてみてください。

```
int one();
int two() { return 2; }
```

1行目も2行目も文法的に正しいCのコードですが、前者は関数本体が書かれていません。前者のような実体を含んでいないものを「宣言」、それに対して2行目のようなものを「定義」といいます。Cコンパイラは宣言をみることで、oneがintを返す関数だということを知ることができます。それによりoneを呼ぶことができるようになります。しかし、oneそのもののコードを出力することはできません。宣言しかみていない場合、関数本体については何も知らないのでこれは当然です。

グローバル変数についても同じように宣言と定義の区別があります。グローバル変数ではexternをつけると宣言、それがない場合に定義になります。下に例を示します。

```
extern int var; // 宣言
int var;        // 定義
```

structやunion、typedefなども宣言です。そういった型を定義すること自体はアセンブリを出力する対象にはならないからです。ただしその型の変数などを定義すると、当然ながらそれは定義になります。下に例を示します。

```
typdef struct { int foo; } FooType; // 宣言
FooType var;                        // varの定義

struct FooType { int foo; };        // 宣言
struct FooType var;                 // varの定義
```

ヘッダファイルには複数のCファイルが必要とする宣言だけを書いてください。そうすることにより、ヘッダファイルをインクルードすることでコンパイラに型情報を教えることができます。ヘッダファイルはそれ単体ではコンパイル対象にはなりません。仮にヘッダファイルをコ

ンパイルしたら、何も定義は含まれていないはずなので、空のアセンブリができあがるはずです。

ヘッダファイルに定義を書いてしまった場合はどうなるでしょうか？ 例えばヘッダファイルに関数を本体付きで書いてしまったとしましょう。そうすると、それをインクルードしているすべてのCファイルにその関数が書いてあるかのようなコンパイル結果になるので、複数のオブジェクトファイルに関数定義が含まれるようになります。結果として、名前が重複して定義されていることになるので、リンクエラーになります

1つのCファイルの中からしか使われていない関数などはヘッダファイルに書く必要はありません。そういった関数の情報は、他のCファイルをコンパイルするときに必要ないからです。

ヘッダファイルに宣言を書いたグローバル変数や関数は、どれか1つのCファイルにだけ定義を書いておきます。例えばヘッダファイルに以下の宣言が書かれていたとすると

```
extern int foo;
int bar();
```

どれか1つのCファイルには次のような定義が含まれていなければいけません。

```
int foo;
int bar() { ... }
```

コラム: 分割コンパイルとビルドのスピード

そもそもなぜファイルを分割する必要があるのでしょうか？ 技術的に見るとソースを分割しなければならない必然性というものはありません。コンパイラにソースコード全体が一度に渡された場合、コンパイラは完全なプログラムを出力することができます。

ただしこのようなやり方の場合、コンパイラは、プログラムが使っているコードを本当にすべて知っている必要があります。例えば`printf`というものは単に誰かがC言語で書いた関数なわけですが、そういったものもコンパイラの入力に与える必要が出てきてしまいます。何度も同じ関数をコンパイルするのは、多くの場合、単なる時間の無駄です。

また、このようなやり方では、1行変更しただけでもすべてのコードをコンパイルし直すことが必要です。大きなプロジェクトではソースコードは1000万行以上あったりするので、1つの単位としてソースコード全体を一気にコンパイルするやり方は現実的に無理があるのです。

Makefileの変更

さて、プログラムを複数のファイルに変更したところで、Makefileも更新しておきましょう。下のMakefileは、カレントディレクトリに置かれているすべての.cファイルをコンパイル＆リンクして、9ccという実行ファイルを作成するためのものです。プロジェクトのヘッダファイルとしては、9cc.hという一つのファイルだけが存在して、そのヘッダファイルをすべての.cファイルでインクルードしているものと仮定しています。

```
CFLAGS=-Wall -std=c11
SRCS=$(wildcard *.c)
OBJS=$(SRCS:.c=.o)

9cc: $(OBJS)
	$(CC) -o 9cc $(OBJS) $(LDFLAGS)

$(OBJS): 9cc.h

test: 9cc
	./9cc -test
	./test.sh

clean:
	rm -f 9cc *.o *~
```

Makefileのインデントはタブ文字でなければいけないことに注意してください。

makeは高機能なツールで、必ずしも使いこなす必要はないのですが、上のMakefileくらいは読めるようになっているといろいろな場面で役に立ちます。そこで、この節では上のMakefileの説明を行います。

Makefileでは、コロンで区切られた行と、それに続くタブでインデントされた0行以上のコマンドの行が、1つのルールを構成します。コロンの前の名前のことを「ターゲット」といいます。コロンの後ろの0個以上のファイル名のことを依存ファイルといいます。

CFLAGSやSRCS、OBJSは変数です。

CFLAGSはmakeの組み込みルールによって認識される変数で、Cコンパイラに渡すコマンドラインオプションを書いておきます。ここでは、積極的に警告をだすための-Wallというフラグと、Cの最新規格であるC11で書かれたソースコードということを伝えるための-std=c11という2つのフラグを渡しています。

SRCSの右辺で使われているwildcardというのはmakeが提供している関数で、関数の引数にマッチするファイル名に展開されます。$(wildcard *.c)は、現在のところmain.c parse.c codegen.c container.cに展開されるというわけです。

OJBSの右辺では変数の置換ルールを使っていて、それによりSRCの中の.cを.oに置き換えた値を生成しています。SRCSはmain.c parse.c codegen.c container.cなので、OBJSはmain.o parse.o codegen.o container.oになります。

これらを踏まえた上で、make 9ccと実行したときに何が起こるのかをトレースしてみましょう。makeは引数として指定されたターゲットを生成しようとするので、9ccファイルを作ることがコマンドの最終的な目標になります（引数がない場合は最初のルールが選ばれるので、この場合は9ccは指定しなくてもよい）。makeはそのために依存関係をたどっていって、欠けている、あるいは古くなっているファイルをビルドしようとします。

9ccの依存ファイルは、カレントディレクトリにある.cファイルに対応する.oファイルです。もし前回makeを実行したときの.oファイルが残っていて、それが対応する.cファイルより新しいタイムスタンプであるときは、makeはわざわざ同じコマンドを再実行したりはしません。.oファイルが存在しないか、.cファイルのほうが新しい場合にのみ、コンパイラを実行して.oファイルを生成します。

$(OBJS): 9cc.hというルールは、すべての.oファイルが9cc.hに依存していることを表しています。したがって9cc.hを変更した場合、すべての.oファイルが再コンパイルされることになります。

ステップ8: ==と!=を追加する

==というのは普通の2項演算子です。+が両辺を足した結果を返すように、==は両辺が同じ場合は1を、違う場合は0を返します。x86-64では、比較はcmp命令を使って行います。スタックから2つの整数をポップして比較を行い、同一の場合に1、そうでなければ0をRAXにセットするコードは次のようになります。

```
pop rax
pop rdi
cmp rdi, rax
sete al
movzb rax, al
```

このコードは、短いアセンブリながらやや盛りだくさんなので、ステップバイステップでコードを見ていきましょう。

最初の2行では値をスタックからポップしています。3行目では、それらのポップしてきた値を比較しています。比較結果はどこにいくのでしょうか？ x86-64では、比較命令の結果は特別な「フラグレジスタ」というものにセットされます。フラグレジスタは加算や比較演算命令が実行されるたびに更新されるレジスタで、結果が0かどうかといったビットや、桁あふれが発生したかどうかというビット、結果が0未満かどうかといったビットなどを持っています。

フラグレジスタは普通の整数レジスタではないので、RAXに結果をセットしたい場合、フラグレジスタからRAXに値をコピーしてくる必要があります。それがsete命令です。sete命令は、直前のcmp命令で調べた2つのレジスタの値が同じだった場合に、指定されたレジスタ（ここではAL）に1をセットします。それ以外の場合は0をセットします。!=の場合はsetneを使ってください。

ALというのは本書のここまでに登場していない新しいレジスタ名ですが、実はALはRAXの下位8ビットを指す別名レジスタにすぎません。従ってseteがALに値をセットすると、自動的にRAXも更新されることになります。ただし、RAXをAL経由で更新するときに上位56ビットは元の値のままになるので、RAX全体を0か1にセットしたい場合、上位56ビットはゼロクリアする必要があります。それを行うのがmovzb命令です。sete命令が直接RAXに書き込めればよいのですが、seteは8ビットレジスタしか引数に取れない仕様になっているので、比較命令では、このように2つの命令を使ってRAXに値をセットすることになります。

コラム: フラグレジスタとハードウェア

値の比較結果が、普通の整数レジスタと異なる特別なレジスタに暗黙のうちに保存されるというこれはx86-64の仕様は、最初はわかりにくく感じるかもしれません。実際、RISCプロセッサでは、フラグレジスタを持つのを嫌って、値の比較結果を普通のレジスタにセットするという命令セットを持っているものがあります。たとえばRISC-Vはそのような命令セットです。

しかし、ハードウェアを実装する立場からすると、素朴な実装であれば、フラグレジスタを作るのはとても簡単です。整数演算の結果をレジスタに書き戻すときに、そこから線を分岐して別のロジックにつないで、そちらでその結果がゼロかどうか（すべての線が0かどうか）とか、結果がマイナスかどうか（最上位ビットの線が1かどうか）などを見て、その結果をフラグレジスタにセットしてしまえばよいのです。フラグレジスタを持つCPUはまさにそのように実装されていて、整数演算を行うたびにフラグレジスタもついでに更新されることになります。

ソフトウェアの場合、「ついでに何かを計算する」ということをすると必ず余分な時間がかかってしまいますが、ハードウェアでは、線を分岐して余分にトランジスタを使うこと自体に時間的ペナルティは発生しないので、フラグレジスタを毎回更新するコストは、少なくとも素朴な実装の場合には存在しないのです。

1972年のCコンパイラ

ここまで我々はインクリメンタルにコンパイラを作ってきました。この開発プロセスはある意味でCの歴史をそのままなぞっていると言うことができます。

現在のCを見てみると意味のよくわからない部分や不必要に複雑な部分が散見されますが、そういったものは歴史を抜きにして理解することはできません。現在のCの不可解なところも、初期のCのコードを読んで、初期のCの形とその後の言語とコンパイラの発展の様子をみてみると、いろいろ腑に落ちるところがあります。

CはUnixのための言語として1972年に開発が始まりました。1972年か1973年当時の、つまりCの歴史の中での極めて初期のソースコードがテープに残されていて、そこから読み出したファイルがインターネットに公開されています。当時のCコンパイラのコードを少し覗いてみましょう。以下に示すのは、`printf`フォーマットでメッセージを受け取って、それをコンパイルのエラーメッセージとして表示する関数です。

```
error(s, p1, p2) {
  extern printf, line, fout, flush, putchar, nerror;
  int f;

  nerror++;
  flush();
```

```
    f = fout;
    fout = 1;
    printf("%d: ", line);
    printf(s, p1, p2);
    putchar('\n');
    fout = f;
}
```

どことなく奇妙な、CのようなCではないような言語に見えます。当時のCはこういう言語でした。このコードを読んでまず気がつくのは、我々が作ってきたコンパイラの初期の段階と同じように、関数の返り値や引数に型がないことです。ここではsは文字列へのポインタ、p1やp2は整数のはずなのですが、当時のマシンではすべてが同じ大きさだったので、このように変数は型なしになっています。

2行目には、errorが参照しているグローバル変数と関数の宣言が書かれています。当時のCコンパイラにはヘッダファイルもCプリプロセッサもなかったので、このようにしてプログラマはコンパイラに変数や関数の存在を教えてやる必要がありました。

現在の我々のコンパイラと同じように、関数は名前が存在するかどうかがチェックされるだけで、引数の型や個数が一致しているかどうかはチェックされません。想定している個数の引数をスタックに積んだあとに、おもむろに関数本体にジャンプすれば関数呼び出しが成功するので、それでよしとしていたのでしょう。

foutというのは出力先のファイルディスクリプタの番号を持っているグローバル変数です。この頃にはまだfprintfが存在しておらず、標準出力ではなく標準エラー出力に文字列を書き出すためには、グローバル変数経由で出力先をスイッチする必要がありました。

errorの中ではprintfが2回呼ばれています。2回目のprintfではフォーマット文字列に加えて2つの値が渡されています。では、1つの値だけを取るようなエラーメッセージを表示するときにはどうしていたのでしょうか？

実はこのerror関数は、単に無理やり少ない引数で読んでも正しく動作します。関数の引数チェックがこの時点では存在しなかったことを思い出してください。s、p1、p2といった引数は単にスタックポインタから1、2、3番目のワードを指しているだけで、実際にp2に相当する値が渡されているかどうかはコンパイラは気にしません。printfは、第一引数の文字列に含まれる%dや%sの個数ぶんだけ余分な引数にアクセスするので、%dをひとつだけ含むメッセージの場合、p2はまったくアクセスされません。したがって引数の個数が一致していなくても問題ないのです。

このように初期のCコンパイラには、現時点での9ccと類似した点がたくさんあります。

もう1つコードの例を見てみましょう。下のコードは、渡された文字列を静的に確保された領域にコピーして、その領域の先頭を指すポインタを返す関数です。つまりこれは静的な領域を使うstrdupのような関数です。

```
copy(s)
char s[]; {
  extern tsp;
  char tsp[], otsp[];

  otsp = tsp;
  while(*tsp++ = *s++);
  return(otsp);
}
```

この当時はint *pという形式の宣言の構文が考案されていませんでした。そのかわりにポインタ型はint p[]というように宣言します。関数の引数リストと関数本体との間に変数定義のようなものが入っていますが、これはsをポインタ型として宣言するためのものです。

この初期のCコンパイラには特筆するべきことが他にもあります。

- この時点では構造体は存在していませんでした。
- &&や||といった演算子もまだありません。この頃は&や|がifなどの条件式の中でだけ論理演算子になるという文脈依存の動作になっていました。
- +=といった演算子は=+というように書いていました。この文法には、iに-1を代入するつもりで、空白を入れずにi=-1と書いてしまうと、i =- 1と見なされてiがデクリメントされるという意図しない動作になってしまう問題がありました。
- 整数型はcharとintだけで、shortやlongは存在しませんでした。「関数ポインタの配列」といった型を宣言する構文は存在せず、複雑な型を記述することができませんでした。

上記のほかにも70年代初期のCにはいろいろな機能が欠けていました。とはいえ、このCコンパイラは、上のソースコードからわかるようにCで書かれていました。構造体すらない時代にすでにCはセルフホストしていたのです。

古いソースコードを見ると、Cの一部のわかりにくい文法がなぜ現在の形になってしまったのかを推測することもできます。externかautoかintかcharの後ろに必ず変数名が来る、という文法なら変数定義のパースは簡単です。ポインタを表す[]も単に変数名の直後に来るだけならパースするのは簡単です。ただし、この文法を、この初期のコンパイラで見えている方向性に沿って発展させていくと、現在の不必要に複雑な形になってしまうのもわかるような気がします。

さて、1972年にUnixとCの共同開発者のDennis Ritchieが行っていたのは、まさにインクリメンタルな開発でした。彼は、Cそのものを発展させるのと平行して、Cを使ってそのコンパイラを書いていたのです。現在のCは、言語の機能追加を続ける中で特別なポイントに達した何らかの完成形というわけではなく、単にDennis Ritchieがある時点で、これで言語の機能は十分、と思ったところで言語として完成ということになっただけです。

我々のコンパイラでも最初から完成形を追い求めることはしませんでした。Cの完成形は特別な意味があるものではないので、それを特別に追い求めることにそこまで意味はないでしょう。どの時点でもリーズナブルな機能のセットを持った言語として開発を続けていって、最終的にCにする、というのは、最初のCコンパイラがそうしていた由緒正しい開発手法なのです。自信を持って開発を進めていきましょう！

ステップ9: 関数の呼び出しに対応する

これ以降の章は執筆中です。ここまでの章は丁寧に書いたつもりですが、ここからの章は正直まだ公開するレベルには達していないと思います。ただし、ここまで読み進めてきた人ならば自分で必要なことを補完して読めないこともないでしょうし、どのような手順で進めるのがよいのか道標が欲しい人もいるでしょうから、そういう意味で公開しておきます。

このステップでは`foo()`のような引数なしの関数呼び出しを認識できるようにして、これを`call foo`にコンパイルするということを目標にします。

テストでは`int foo() { printf("OK\n"); }`のような内容のCファイルを用意しておいて、それを`cc -c`でオブジェクトファイルにコンパイルして、自分のコンパイラの出力とリンクします。そうすると全体としてきちんとリンクできて、自分の呼び出したい関数がきちんと呼ばれていることも確認できるはずです。

それが動いたら、次は`foo(3, 4)`のような関数呼び出しを書けるようにしてください。引数の個数や型のチェックはいりません。単に引数を順番に評価すると、スタック上に関数に渡すべき引数ができあがるので、それをx86-64のABIで規定されている順番でレジスタにコピーして、関数をcallします。

テストでは上と同じように、`int foo(int x, int y) { printf("%d\n", x + y); }`のような関数を適当に用意しておいて、それをリンクすれば動作確認できるはずです。

その後`return`文も追加してください。

x86-64の関数呼び出しのABIは（上のようなやり方をしている限りは）簡単ですが、注意点が一つあります。関数呼び出しをする前にRSPが16バイトにアラインされている必要があります。アラインメントがずれている（たとえばRSPが4バイトにしかアラインされていない）場合、だいたいうまく動くけど、16バイトアラインメントを前提にしている関数が（たとえば半分の確率で）落ちる謎のコードになります。関数を呼ぶ前にRSPを調整するようにして、16バイトアラインメントを保証するようにしましょう。

ステップ10: 関数の定義に対応する

ここまでが終わったら次は関数定義をできるようにします。とはいえCの関数定義は構文解析が面倒なのでいきなり全部を実装したりはしません。現在のところ我々の言語にはint型しか存在しないので、`int foo(int x, int y) { ... }`という構文ではなく 型名を省略した`foo(x, y) { ... }`という構文を実装します。

呼び出された側ではxやyといった名前で引数にアクセスできる必要があるわけですが、レジスタで渡された値にそのまま名前でアクセスすることは現状できません。ではどうするかというと、xやyといったローカル変数が存在するものとしてコンパイルして、関数のプロローグの中で、レジスタの値をそのローカル変数のためのスタック上の領域に書き出してください。そうすれば、その後は特に引数とローカル変数を区別することなく扱えるはずです。

今までは暗黙のうちに全体が`main() { ... }`で囲まれているのと同じ動作になっていましたが、それは廃止して、全部のコードを何らかの関数の中に書くようにします。そうするとトップレベルをパースしているときは、まずトークンを読むとそれは必ず関数名のはずで、その後に続くのは引数リストのはずで、そのあとは関数本体が続いているはず、となるので、簡単に読めます。

このステップが終わるとフィボナッチ数列を再帰で計算しつつ表示したりできるようになるのでグッと面白くなるはずです。

ステップ11: 制御構文を足す

`if`、`while`、`for`を追加してください。制御構文には他にも`do ... while`、`goto`、`continue`、`break`など様々な構文が存在しますが、それらはこの時点ではまだ実装する必要

はありません。

`if (A) B`は次のようなアセンブリにコンパイルします。

```
  Aをコンパイルしたコード // スタックトップに結果が入っているはず
  pop rax
  cmp rax, 0
  jne .Lend
  Bをコンパイルしたコード
.Lend:
```

つまり`if (A) B`は、

```
if (A == 0)
  goto end;
B;
end:
```

と同じように展開されるというわけです。

同様に`while (A) B`は次のようにコンパイルします。

```
.Lbegin:
  Aをコンパイルしたコード
  pop rax
  cmp rax, 0
  jne .Lend
  Bをコンパイルしたコード
  jmp .Lbegin
.Lend:
```

`for`も同様です。gccで小さいループをコンパイルしてそのアセンブリを参考にして作ってください。

ステップ12: 暗黙の変数定義を廃止して、intというキーワードを導入する

次にそろそろ配列を扱いたいのですが、Cでは配列はポインタと密接に結びついているので、まずポインタを実装する必要があります。例によって一気にポインタと配列を足すことはせず、最も簡単なポインタの実装をまず行い、それをだんだん改造していくことにします。

いままでは変数や関数の返り値はすべて暗黙のうちにintということになっていました。したがってわざわざ`int x;`というように変数をその型名と一緒に定義することはせず、新しい識別子はすべて新しい変数名だとみなしていました。今後はそのような仮定を置くことはできなくなります。そこで、まずその点を改造します。以下の機能を実装してください。

- 新しい識別子を変数名とみなすのはやめて、定義されていない変数が現れたらエラーにしてください。
- `int x;`という形で変数を定義するようにしてください。`int x = 3;`といった初期化式などはサポートする必要はありません。同様に`int x, y;`といったものも必要ありません。なるべく単純なものだけを実装します。
- 関数はいままで`foo(x, y)`といった形で書いていましたが、これを`int foo(int x, int y)`といった形になるように改造します。現状、トップレベルは関数定義しかないはずなので、パーザはまず`int`を読み、そのあとは必ず関数名のはずなのでそれを読み、次に`int <引数の名前>`という列を読む、ということになります。これ以上難しい構文には対応する必要はないですし、「将来の拡張にそなえて」といった念のためになにかする必要もありません。単純に"int <関数名>(<int <変数名>の繰り返しからなる引数リスト>)"を読むために十分なだけのコードを書いてください。

ステップ13: ポインタ型を導入する

ポインタを表す型を定義する

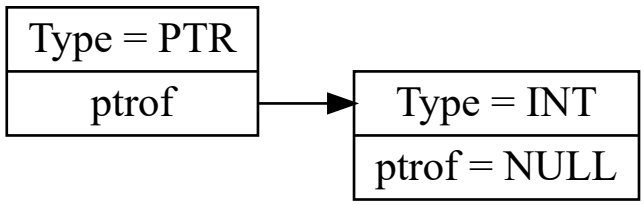
このステップでは、いままでは型名に`int`しか許していなかったのを、`int`のあとに`*`が0個以上続く、というものを型名として許すことにします。すなわち`int *x`や`int ***x`といった定義を構文解析できるようにします。

「intへのポインタ」といった型はコンパイラの中で無論扱える必要があります。たとえば変数xがintへのポインタだとしたら、コンパイラは式*xはint型だとわからなければいけないわけです。「intへのポインタへのポインタへのポインタ」というように型はいくらでも複雑にできるので、これは固定のサイズの型だけで表すことはできません。

ではどうするかというと、ポインタを使います。今まで変数に対してマップを通して紐付けられている情報は、スタック上のベースポインタ（RBP）からのオフセットだけでした。これに変更を加えて、変数の型を持てるようにしてください。変数の型というのは、大雑把にいうと、次のような構造体になるはずです。

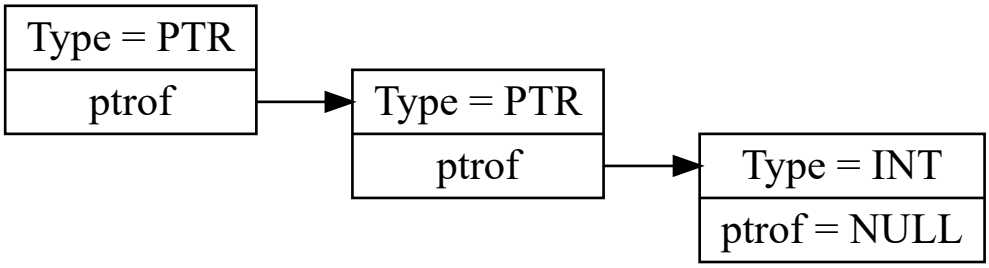
```
struct Type {
  enum { INT, PTR } ty;
  struct Type *ptrof;
};
```

ここで`ty`はint型か「〜へのポインタ」型かという2つの値のどちらかを持つことができます`ptrof`は`ty`が「〜へのポインタ」型であるときのみに意味のあるメンバーで、そのときには、「〜」が指すTypeオブジェクトへのポインタを入れておきます。たとえば「intへのポインタ」なら、その型を表すデータ構造は内部的に次のようになるわけです。



intへのポインタを表すデータ構造

「intへのポインタへのポインタ」なら次のようになります。



intへのポインタのポインタを表すデータ構造

このようにすればコンパイラ内部でいくらでも難しい型を表すことができるというわけです。

ポインタが指している値に代入する

代入式の左辺が単純な変数名ではない式、たとえば*p=3のような式はどのようにコンパイルすればよいのでしょうか？ こういった式も、左辺が単純な変数のときと基本的な概念は変わりません。この場合には、pのアドレスが生成されるように、*pを左辺値としてコンパイルすればよいのです。

*p=3を表す構文木をコンパイルするときは、再帰的にツリーを下りながらコード生成していくわけですが、まず最初に呼ばれるのは*pを左辺値としてコンパイルするためのコードジェネレータです。

そのコードジェネレータでは与えられた構文木の型に応じて分岐することになります。単純な変数では前述のとおりその変数のアドレスを出力するコードを出力することになるわけですが、ここではデリファレンス演算子が与えられているので、違った動作をする必要があります。デリファレンス演算子が与えられている場合、その中の構文木を「右辺値」としてコンパイルしてください。そうすると、それは何らかのアドレスを計算するコードにコンパイルされるはずです（そうでなければその結果をデリファレンスすることはできません）。そしてそのアドレスをそのままスタックに残しておけばよいというわけです。

この段階までが完成したら、次のような文をコンパイルできるようになるはずです。

```
int x;
x = 3;
int *y;
y = &x;
*y; // → 3
```

ステップ14: ポインタの加算と減算を実装する

このステップでは、ポインタ型の値pに対してp+1やp-5のような式を書けるようにします。これはただの整数の加算と同じように見えますが、実際には結構異なる演算です。p+1は、pが持っているアドレスに1を足す、という意味ではなくて、pの次の要素を指すポインタにする、という意味なので、ポインタが指しているデータ型の幅をpに足してやらなければいけないわけです。たとえばpがintを指している場合、p+1はアドレスのバイト数としては4を足すことになります（intが4バイトの場合）。一方でpがintへのポインタへのポインタである場合、p+1は8を足すことになります（ポインタが8バイトの場合）。

したがってポインタの加減算では、型のサイズを知る方法が必要になりますが、現状ではintなら4、ポインタなら8なので、そのように決め打ちでコードを書いてください。

この段階ではまだ連続してメモリをアロケートする方法がないので（我々のコンパイラにはまだ配列がない）、テストを書くのはちょっと大変です。ここは単に外部のコンパイラの助けを借りて、そちらのほうでmallocすることにして、自分のコンパイラの出力ではそのヘルパー関数を使ってテストを書くようにしてみてください。例えばこんな感じでテストできるでしょう。

```
int *p;
alloc4(&p, 1, 2, 4, 8);
int *q;
q = p + 2;
*q;  // → 4
q = p + 3;
*q;  // → 8
```

ステップ15: 配列を実装する

配列型を定義する

このステップでは配列を実装します。この段階まではレジスタに入る大きさのデータしか扱ってきていませんでしたが、今回初めてそれより大きなデータが登場します。

とはいえCの文法は配列については抑制的です。関数の引数として配列を渡したり、関数の返り値として配列を返したりすることはできません。そういう意図のコードを書くと、配列そのものが値渡しされるのではなく、その配列を指すポインタが自動的に作られてそれが渡されることになります。配列に配列を直接代入してコピーする、といったこともサポートされていません（memcpyを使わないといけない）。

したがって、レジスタに入らないデータを関数や変数の間でやり取りする必要はありません。1ワードより大きいメモリ領域をスタック上に割り当てる機能があれば十分です。

次のような変数定義を読み込めるようにしてください。

```
int a[10];
```

上記のaの型は配列であり、その配列は長さが10で、要素の型はintです。ポインタ型と同様、配列の型もいくらでも複雑にできるので、ステップ7と同じように、ptrofで配列の要素の型は指し示すようにします。型を表す構造体は次のようになるはずです。

```
struct Type {
  enum { INT, PTR, ARRAY } ty;
  struct Type *ptrof;
  size_t array_size;
};
```

ここで`array_size`は配列型のときにのみ意味のあるフィールドで、配列の長さを持つ変数です。

ここまでできれば配列のための領域をスタックにアロケートするのは簡単にできるはずです。配列のバイト単位での大きさを求めるためには、配列の要素のバイト単位での大きさと配列の長さをかければいいだけです。いままではすべての変数を1ワードとしてスタック領域を確保していたはずですが、それを変更して、配列は配列に必要な大きさを確保するようにしてください。

配列からポインタへの暗黙の型変換を実装する

配列とポインタはよく組み合わせて使われるので、Cでは構文上、ポインタと配列をあまり区別せずともなんとなく動くようになっているのですが、それが裏目に出て、配列とポインタの

関係がどうなっているのか、プログラマにとってわかりにくくなってしまっているようです。そこでここでは配列とポインタの関係について説明をします。

まずCにおいては配列とポインタは完全に別々の型です。

ポインタは（x86-64では）8バイトの値の型です。intに対して+や-といった演算子が定義されているように、ポインタに対しても+や-が（やや異なる形で）定義されています。ポインタにはそれに加えて、単項*演算子が定義されていて、それを使うことでポインタの指している先を参照することができます。単項*を除けばポインタにはそれほど特別なことはないといってよいでしょう。言ってみればポインタはintのような普通の型です。

一方で配列は何バイトにでもなりうる型です。ポインタとは異なり、配列に対しては演算子がほとんど定義されていません。定義されている演算子は、配列のサイズを返す`sizeof`演算子（記号ではないのですが`sizeof`は単項演算子扱いです）と、配列の先頭の要素のポインタを返す&演算子だけです。それ以外、配列に対して適用できる演算子はありません。

ではなぜ`a[3]`のような式がコンパイルできるのでしょうか？ Cでは、`a[3]`は`*(a+3)`と等価であるものとして定義されています。配列に対して+演算子は定義されていないのではなかったのでしょうか？

ここで配列が暗黙のうちにポインタに変換される、という文法が効いてくることになります。`sizeof`か単項&のオペランドとして使われるとき以外、配列は、その配列の先頭要素を指すポインタに暗黙のうちに変換されるということになっているのです。したがって、`*(a+3)`は、配列aの先頭要素を指すポインタに3を足したものをデリファレンスする、という式になり、それは結果的に配列の3番目の要素をアクセスするのと同じ意味になります。

Cにおいては配列アクセスのための`[]`演算子というものはありません。Cの`[]`は、ポインタ経由で配列の要素にアクセスするための簡便な記法にすぎないのです。

同様に、関数引数として配列を渡すとその配列の先頭要素へのポインタになったり、ポインタに対して配列を直接代入しているかのような書き方ができたりしますが、それも上記のような理由によります。

というわけで、コンパイラは、ほとんどの演算子の実装において、配列をポインタに型変換するということを行わなければなりません。これは実装するのはさほど難しくはないでしょう。`sizeof`と単項&を実装している場合を除き、演算子のオペランドをパーズしたら、その型がTの配列だったらTへのポインタということにしてしまう、とすればよいはずです。コードジェネレータでは、配列型の値は、その値のアドレスをスタックにプッシュするというコードを生成すればよいはずです。

ここまで完成すれば、次のようなコードが動くようになるはずです。

```
int a[2];
*a = 1;
*(a + 1) = 2;
int *p;
p = a;
*p + *(p + 1)  // → 3
```

ステップ16: 配列の添字を実装する

Cでは x[y]は*(x+y)と等価であるものとして定義されています。したがって添字の実装は比較的簡単です。単純にx[y]をパーザの中で*(x+y)として読み換えるようにしてください。たとえばa[3]は*(a+3)になります。

この文法では、3[a]は*(3+a)に展開されるので、a[3]が動くなら3[a]も動くはずですが、なんとCでは3[a]のような式は実際に合法です。試してみてください。

ステップ17: グローバル変数を実装する

そろそろリテラルの文字列をプログラムに書けるようにしたいところです。Cではリテラルの文字列はcharの配列です。すでに配列は実装しているからよいのですが、リテラルの文字列はスタック上に存在している値ではない、という点が違います。文字列リテラルは、スタック上ではなく、メモリ上の固定の位置に存在しているわけです。したがって、文字列リテラルを実装するために、まずグローバル変数を足すことにします。

いままではトップレベルには関数定義しか許していなかったはずです。その文法を変更して、トップレベルにグローバル変数を書けるようにします。

変数定義は関数定義と見た目が似ているので構文解析はややトリッキーです。たとえば次の4つの定義を比べてみてください。

```
int *foo;
int foo[10];
int *foo() {}
int foo() {}
```

上の2つのfooは変数定義で、下の2つは関数定義ですが、その2つは、関数名あるいは変数名になる識別子まで到達して、その次のトークンを読んでみるまで区別がつきません。したがって、まず「型名の前半を読む」関数を呼び、そのあとに識別子が来ているはずなのでそれを読み、それからトークンを1つ先読みしてみる必要があります。先読みしたトークンが"("なら関数定義を読んでいた、ということになりますし、そうでなければ変数定義を読んでいた、ということになります。

パースしたグローバル変数の名前はマップにいれて、名前でルックアップできるようにしてください。変数名がローカル変数として解決できなかった場合に限り、グローバル変数として解決を試みます。これによりローカル変数が同名のグローバル変数を隠すという動きが自然に実装できます。

パーザでは、ローカル変数の参照とグローバル変数の参照は抽象構文木の別のノードに変換します。パースの段階で名前が解決できるので、その段階でタイプも分けてしまうというわけです。

いままではすべての変数がスタックにあったはずなので、変数の読み書きはRBP（ベースポインタ）からの相対で行なっていました。グローバル変数はスタック上にある値ではなく、メモリ上の固定の位置にある値なので、そのアドレスに直接アクセスするようにコンパイルします。実際のgccの出力を参考にしてみてください。

実装してみると、ローカル変数とグローバル変数がかなり異なるものであることにびっくりするはずです。見た目上区別することなく書くことができるのは、C言語がうまく抽象化しているからですね。ローカル変数とグローバル変数は、実は内部ではかなり違うように実装されているのです。

ステップ18: 文字型を実装する

配列は1ワードより大きくなりうる型でしたが、文字は1ワードより小さな型です。みなさんは、このステップに至るまでに、型を表すオブジェクトを受け取って、その型のサイズのバイ

ト数を返す関数を書く必要があったと思います。まずは文字という型を足し、そのあとにその関数に変更を加え、文字型に対して1を返すようにしてください。

このステップではリテラルの文字（シングルクオートでくくられた文字）を実装する必要はありません。一気に実装したくなる気持ちは抑えてなるべく小さな変更にとどめます。

したがってこのステップでは文字というのは本当にただの小さな整数型です。movsbl命令を使うと1バイトを読み込むことができます。

書き出す時にはmovb命令を使います。EAXがRAXの下位32ビットであるように、x86-64では下位8ビットのエイリアスされたレジスタというのがあります。例えばRAXの下位8ビットはALとしてアクセスすることができます。したがってRAXに文字が入っているとして、それを書き出したい場合、ALをメモリにmovすればよいということになります。実際のコンパイラの出力を参考にしてください。

このステップが実装できれば、次のようなコードが動くようになるはずです。

```
char x[3];
x[0] = -1;
x[1] = 2;
int y;
y = 4;
x[0] + y;  // → 3
```

ステップ19: 文字列リテラルを実装する

このステップではダブルクオートでくくられた文字列をパースしてコンパイルできるようにします。配列とグローバル変数、文字型という必要なパーツが揃ったので、比較的簡単に実装できると思います。

まずはトークナイザに手を入れて、ダブルクオートを見つけたら、次のダブルクオートまで読んで文字列トークンを作成するようにしてください。このステップではバックスラッシュによるエスケープなどは実装する必要はありません。ステップバイステップで行くのが重要なので、簡単に実装できそうに思えても、しないようにしてみてください。

文字列リテラルのデータを表すアセンブリのコードは、CPUに実行されるマシンのコードを生成している途中に出力することはできません。出力されるアセンブリでは、グローバルなデータと、コードは、混ぜずに書く必要があります。つまり、コードを出力するときには、コード中に出現していたすべての文字列リテラルをまず出力してしまいたいわけですが、そのために構文木をくだるのは面倒です。これをするためには、今までに見た文字列リテラルがすべて入っているベクタというのを用意して、パーザが文字列をみるたびにそれに単に足して行くようにするのが簡単でしょう。

実際のコンパイラの出力を参考にしてください。

ここまでくると`printf`で文字列を出力することも可能になっているはずです。自分で作ったプログラミング言語を使って、テストコードのような自明なものではなく、もうちょっと凝ったプログラムを書いてみるよい機会です。たとえば8クイーン問題のソルバーなどは自作言語で書けるのではないでしょうか？ 人類はデジタルコンピュータの発明から、このレベルで簡単にコードが書けるプログラミング言語の開発まで、何十年もかかりました。それが数週間で実装できるのは人類の、そしてあなたの素晴らしい進歩です。

（可変長引数を取る関数を呼ぶときは、浮動小数点数の引数の個数をALに入れておく、ということになっています。我々のコンパイラにはまだ浮動小数点数がありません。したがって、関数を呼ぶ前に常にALに0をセットするようにしましょう。）

ステップ20: テストをCで書き直す

このステップではテストを書き直して`make test`を高速化します。このステップに至ったころにはすでにシェルスクリプトに100個以上のテストが書かれていることでしょう。シェルスクリプトのテストではテスト1つにつき何個ものプロセスが起動されます。つまりテスト1つごとに自作コンパイラ、アセンブラ、リンカ、テストそのものを起動しています。

プロセスの起動というものは小さなプログラムでもそこまで速くありません。したがってそれを何百回も行うとなると、トータルでは無視できない時間がかかるようになってしまいます。おそらくみなさんのテストスクリプトも実行に数秒かかるようになっていると思います。

そもそもシェルスクリプトでテストを書いていたのは、そうしなければまともなテストができなかったからでした。電卓レベルの言語の段階では、`if`や==などがなかったので、計算結果が正しいかどうかその言語の中では検証できなかったのです。しかし今は検証できるようになりました。結果が正しいかどうかを比較して、間違っている場合は（文字列の）エラーメッセージを表示してexitする、ということが可能になっているのです。

そこで、このステップでは、シェルスクリプトで書いていたテストをCファイルに書き直してください。

ステップ21以降: [要加筆]

付録：参考資料

- Compiler Explorer: 便利なオンラインコンパイラ
- N1548: C11言語仕様の最終ドラフト（公式の規格書と内容は同一）
- 8cc: 筆者によるCコンパイラ
- 9cc: 筆者によるCコンパイラ
- Cコンパイラをスクラッチから開発してみた（日記）

付録：x86-64命令セット チートシート

この章では、このコースで製作するコンパイラで利用するx86-64命令セットの機能をまとめました。この章では簡潔に表現するために以下の省略記法を使いました。

- `src`, `dst`: 同じサイズの2つの任意のレジスタ
- `r8`, `r16`, `r32`, `r64`: それぞれ8ビット、16ビット、32ビット、64ビットのレジスタ
- `imm`: 即値
- `reg1:r[src]` 2つのレジスタreg1、reg2`をそれぞれ上位ビットと下位ビットとして、128ビットのような1つのレジスタに入らない大きな数を表す場合の記法

整数レジスタの一覧

関数から戻る際に元の値に戻さなくてよいレジスタには✔をつけました。RAXは関数からの返り値です。可変長引数を呼び出す場合は、RAXに浮動小数点数の引数の個数をセットします。

64ビット	32ビット	16ビット	8ビット	ABIにおける使い方	
RAX	EAX	AX	AL	返り値 / 引数の数	✔
RDI	EDI	DI	DIL	第1引数	✔
RSI	ESI	SI	SIL	第2引数	✔
RDX	EDX	DX	DL	第3引数	✔
RCX	ECX	CX	CL	第4引数	✔
R8	R8D	R8W	R8B	第5引数	✔
R9	R9D	R9W	R9B	第6引数	✔
R10	R10D	R10W	R10B		✔
R11	R11D	R11W	R11B		✔
RBP	EBP	BP	BPL	ベースポインタ	
RSP	ESP	SP	SPL	スタックポインタ	
RBX	EBX	BX	BL		
R12	R12D	R12W	R12B		
R13	R13D	R13W	R13B		
R14	R14D	R14W	R14B		
R15	R15D	R15W	R15B		

メモリアクセス

mov dst, [src]	srcのアドレスからdstに値をロード
mov [dst], src	srcの値をdstのアドレスにストア
push r64/imm	RSPを8減らして、r64/immをRSPにストア
pop r64	RSPからr64にロードして、RSPを8増やす

関数呼び出し

call label	RIPをスタックにプッシュしてlabelにジャンプ
call *r64	RIPをスタックにプッシュしてr64のアドレスにジャンプ
ret	スタックをポップしてそのアドレスにジャンプ
leave	`mov rsp, rbp`の後`pop rbp`と同等

条件分岐

cmp reg1, reg2/imm je label	reg1 == reg2/immならlabelにジャンプ
cmp reg1, reg2/imm jne label	reg1 != reg2/immならlabelにジャンプ
cmp reg1, reg2/imm jl label	op1 < op2ならlabelにジャンプ （符号ありでの比較）
cmp reg1, reg2/imm jle label	op1 <= op2ならlabelにジャンプ （符号ありでの比較）

条件代入

cmp reg1, reg2/imm
sete %al
movzb %al, %eax
	RAX = (op1 == op2) ? 1 : 0

cmp reg1, reg2/imm
setne %al
movzb %al, %eax
	RAX = (op1 != op2) ? 1 : 0

cmp reg1, reg2/imm
setl %al
movzb %al, %eax
	RAX = (op1 > op2) ? 1 : 0
	（符号ありでの比較）

cmp reg1, reg2/imm
setle %al
movzb %al, %eax
	RAX = (op1 >= op2) ? 1 : 0
	（符号ありでの比較）

整数・論理演算

add dst, src/imm	dst = dst + src/imm
sub dst, src/imm	dst = dst - src/imm
mul dst, src	dst = src * dst
imul dst, src	mulの符号つきバージョン
div r32	EAX = EDX:EAX / r32 EDX = EDX:EAX % r32
div r64	RAX = RDX:RAX / r64 RDX = RDX:RAX % r64
idiv r32/r64	divの符号つきバージョン
cqto	RAXを128ビットに符号拡張して RDX:RAXにストア
and dst, src	dst = src & dst
or dst, src	dst = src | dst
xor dst, src	dst = src ^ dst
neg dst	dst = -dst
not dst	dst = ~dst
shl dst, imm/CL	immかCLレジスタの値だけdstを左シフトする（レジスタでシフト量を指定する場合、CLしか使えない）
shr dst, imm/CL	immかCLレジスタの値だけdstを論理右シフトする シフトインされてきた上位ビットはゼロクリアされる
sar dst, imm/CL	immかCLレジスタの値だけdstを算術右シフトする シフトインされてきた上位ビットは、もともとのdstの符号ビットと同じになる
lea dst, [src]	[src]のアドレス計算を行うが、メモリアクセスは行わずアドレス計算の結果そのものをdstにストア
movsb dst, r8	r8を符号拡張してdstにストア
movzb dst, r8	r8を符号拡張せずにdstにストア
movsw dst, r16	r16を符号拡張してdstにストア
movzw dst, r16	r16を符号拡張せずにdstにストア

付録：Gitによるバージョン管理

GitはもともとLinuxカーネルのバージョン管理のために開発されました。Linuxカーネルは数千人の開発者からなる巨大なプロジェクトなので、そのための複雑なワークフローを満たすべく、Gitは豊富な機能を持っています。これらの機能は便利ではあるのですが、自分しかコミッターがいない個人開発ではGitの機能を使いこなす必要はありません。Gitをマスターするとなるとかなりたくさんのことを学ぶ必要がありますが、このコースで覚えるべき事項はごくわずかです。Gitを使った開発が初めてだという人のために以下にチートシートを用意しました。

- `git add ファイル名`

  新規作成したファイルをリポジトリに加える

- `git commit -A`

  作業ツリーに行ったすべての変更をまとめてコミットする（エディタが立ち上がるのでコミットメッセージを入力する）

- `git reset --hard`

  前回のコミット以降に作業ツリーに行った変更を全て取り消す

- `git log -p`

  過去のコミットを見る

- `git push`

  Githubなどアップストリームにリポジトリをプッシュする

バージョン管理システムを初めて使う人のために、Gitとバージョン管理システムの概念について少し説明しておきましょう。

Gitは、ファイルの変更履歴が入ったデータベースの管理ツールです。そのデータベースのことをリポジトリといいます。Githubなどからリポジトリをクローンすると、リポジトリがダウンロードされて、そのあとリポジトリから、デフォルトの最新の状態のディレクトリツリーが、あるディレクトリ以下にすべて展開されます。

リポジトリから展開されたディレクトリツリーのことを「作業ツリー」といいます。皆さんには作業ツリーにあるソースファイルをエディタで編集したりコンパイルしてもらうわけですが、作業ツリー自体はリポジトリの一部分というわけではありません。作業ツリーはいわば

zipファイルから展開されたファイルのようなもので、そちらにいくら変更を加えても、元々のリポジトリはそのままの状態で残っています。

自分が作業ツリーに対して行った変更は、キリの良いところで「コミット」という単位にまとめてリポジトリに書き戻します。これによりデータベースが更新され、そのあとまた別の変更点を作っていくことができるわけです。Gitを使う場合は、ファイルを変更してコミット、という繰り返しで開発を進めていくことになります。

コミットするときの注意点

コミットメッセージは日本語でよいのできちんと書いてください。例えば「*と/を演算子として追加、ただし演算子の優先順位は処理しない」といった一行メッセージでよいです。

コミット単位はなるべく小さく分けてください。例えばコードを変更していると、ちょっとしたリファクタリングがしたくなるものですが、そういう場合はリファクタリングは別のコミットとしてコミットしてください。2つ以上の別々の機能を1つのコミットにまとめるのは望ましくありません。

Gitの高度な機能を使う必要はありません。たとえばブランチを使う必要はないはずです。

機能追加のコードは、その機能をテストするコードと必ず一緒にコミットするようにしてください。また、コミットするときは、まずテストを走らせてみて、既存機能が壊れていなくて、新しい機能もきちんと動いていることを確認してからコミットするようにしてください。別の言い方をすると、どの時点でのリポジトリをチェックアウトしてきても、コンパイルとテストは通る、ということを目指してもらいます。ただしうっかりテストが通らないものをコミットしてしまったときは、gitのコミットログを改変してまで修正する必要はありません。単に次のコミットで修正を入れてください。

Gitの内部構造

Gitのドキュメントを読んでいるとトリッキーな機能がたくさんあるのですが、Gitが原理的にどのようにデータを保存しているかというモデルを自分の中に構築しておくと、機能が理解しやすくなります。そこでここではGitの内部構造について解説します。

Gitは、ユーザープログラムとして実装されたファイルシステムの一種です。Gitのデータベースの構造はファイルシステムと大変よく似ているのです。ただし、通常のファイルシステムは

ファイル名を使ってファイルにアクセスするのに対して、Gitではファイルのハッシュ値を名前として使うところが異なります。

ファイルの内容によって名前が決まるこういった仕組みのことを、content-addressableな仕組みといいます。content-addressableなファイルシステムでは、名前が同じであれば内容は同一です。また、内容が異なっているファイルが同じ名前（同じハッシュ値）を持つことはできません。これは暗号的にセキュアなハッシュ関数を使うことにより保証されています。こういったファイルシステムでは、ファイルに別個に名前をつける必要がなく、名前が決まればファイルの内容も一意に決まるという性質があります。

コミットというのもGitの内部ではファイルになっています。そのファイルには、コミットメッセージのほかに、そのコミットに属するファイルのハッシュ値や、その一つ前のコミットのハッシュ値が書かれています。

Gitのリポジトリからファイルを取り出すためには、自分の欲しいファイルのハッシュ値がわかっていなければなりません。

コミットファイルのハッシュ値がわからなければコミットファイルが得られない、というのは鶏と卵の問題のような感じがしますが、実際には、リポジトリにはコミットファイルのハッシュ値とそれに対する名前の目録が入っているので、それを使ってコミットを見つけることができます。たとえばリポジトリには、"master"という名前（デフォルトで作業ツリーに展開される履歴）のコミットのハッシュ値は`da39a3ee5e...`である、といった情報が入っています。この情報を使うことで、Gitはmasterに属するファイルを作業ツリーに展開することができます。

「コミットを行う」というのが内部的にどうなっているかというと、変更があったファイルをGit内部のファイルシステムに追加し、それらのファイルのハッシュ値や一つ前のコミットのハッシュ値を含んだコミットファイルを同様に追加し、最後にそのコミットファイルのハッシュ値で目録を更新する、となっているわけです。

例えばmasterの最後のコミットをなかったことにするためには（やらないほうがよいですが）、masterが指しているコミットファイルを見て一つ前のコミットのハッシュ値を得て、そのハッシュ値でmasterを上書きすればよいわけです。「ブランチ」というのも、あるコミットを一つ前のコミットとして持っているコミットが2つ以上あって、その2つのコミットが目録に載っている、という話にすぎません。

こういったcontent-addressableなバージョン管理システムにはセキュリティ上の利点もあります。あるコミットの名前（コミットファイルのハッシュ値）には、そのコミットに属するすべてのファイルのハッシュ値と、その前のコミットファイルのハッシュ値が含まれています。その前のコミットファイルにはさらにその前のコミットファイルのハッシュが含まれているので、結局のところ最新のコミットにたどり着くすべてのコミットのハッシュ値が、最新のコミ

ットのハッシュ値の計算に含まれていることになります。したがって、ハッシュ値を変えないままコミットの内容や履歴をこっそり改変することが原理的に不可能になっているのです。面白い性質ですね。

Gitの機能を学ぶ時には、このcontent-addressableなファイルシステムを常に念頭に置いてみてください。きっといろいろなことがわかりやすくなるはずです。