記号処理・第 1 回

2018 年 4 月 10 日

今回の内容

シラバス抜粋

サブタイトル	コンパイラ構築の実際
講義概要	簡単な手続き型プログラミング言語を例として、コンパイラ構築の基本的事項について解説し、C を用いたコンパイラの構築をしていきます。その中で、プログラムがどのように計算機アーキテクチャと関わっているのかについての理解を深めます。この講義のねらいは、1. コンパイラの内部構造を知ること、2. ある程度大きなソフトウエアの構成を知ること、3. 種々のプログラミング技法に触れることの 3 つです。受講者が C によるプログラミングに習熟していることを前提としています。
到達目標	日頃お世話になっているコンパイラが、どのような仕組みで動いているのかを理解し説明できるようになる。計算機アーキテクチャに関する理解をプログラミングに応用できるようになる。
講義方法	配布したプリントに沿って講義を行います。概念的な話だけではなく、具体的なプログラムを提示しますので、受講者は自習時間に C を用いて実際にコンパイラを構築して行くことができます。
授業時間外における予･復習等の指示	授業時間以外の学習を前提に授業を進めます。第 2 回以降の授業の資料は事前に配布しますので、十分予習してください。ほぼ毎回演習課題が宿題に出ます。
系統的履修	「計算機システム II」を履修していると内容の理解の助けになるはずです。
成績評価	期末試験 (100 点満点) と提出された課題等で評価します。期末試験が x 点、課題の得点率が $y\%$ の場合、総合的な成績は $x+(100-x)y/500$ 点 (端数切り捨て) となります。
授業計画	(1) コンパイラの内部構成 (2) BNF 記法と構文木 (3) 構文図 (4) 構文図からのパーザの生成 (5) プログラミング言語 Minimum C (6) Minimum C の字句解析 (7) Minimum C の構文解析 (8) 仮想計算機 MVM (9) MVM におけるメモリ割り当て (10) オブジェクトの管理 1 (11) オブジェクトの管理 2 (12) 式に対するコード生成 1 (13) 式に対するコード生成 2 (14) 文に対するコード生成 (15) コンパイラの完成
参考文献	1. 湯浅太一『コンパイラ』(昭晃堂) 3,564 円 (ISBN 9784785620509) 2. R.ハンター『コンパイラ構成論』(近代科学社) 3,456 円 (ISBN 9784764901889)
履修上の注意	集中力をもって授業に望む態度が必要になると思います。それを覚悟して受講して下さい。
オフィスアワー	月曜 3 講時 (1-514) と木曜昼休み (1-614)

1.1 プログラミング言語とは

いろいろな用途にコンピュータが使われています。複雑な計算を行ったり、家電製品や携帯電話を制御したり、ゲームを楽しませてくれたりします。しかし、単にコンピュータという物 (ハードウエア) があっただけでは、それは何の役にも立ちません。コンピュータが役に立つのは、そのコンピュータがどのように働くべきかを指示したプログラム (ソフトウエア) が用意されているからです。コンピュータの行うべき仕事を事細く記述したものを「コンピュータプログラム」あるいは単に「プログラム」と呼びます。このとき、どのようなプログラムを書けばどのようにコンピュータが働くかについての約束事が必要になります。プログラムの書き方とその解釈のされ方があらかじめ決まっていなければ、コンピュータに意図した通りの働きをさせることはできません。その「約束事」を提供してくれるものを「プログラミング言語」と呼びます。日本語が、日本語を話す人の間での意思疎通を可能にしてくれるのと同様に、1 つのプログラミング言語は人間の意図をコンピュータに伝えることを可能にしてくれます。

現在広く利用されているコンピュータは、その中に「メモリ」と呼ばれる記憶装置を備えており、ここにいろいろな情報を保存することができるようになっています。見掛けの上では、メモリに蓄えられる情報は 0 と 1 の二種類の数だけからなる長い長い数列のように見えます。人間が扱う普通の文章や、数値などの情報は、適当な規則にしたがって、0 と 1 からなる数列として表現されてメモリ中に記憶されます。コンピュータ自身の働きを指定するためのプログラムもやはりこのメモリ中に置かれます。0 と 1 の数個から数十個の並びを 1 つの単位として、コンピュータが行うべき作業を指示します。どのような並びでどのような作業を行うかについての約束事は、そのコンピュータによってあらかじめ決められています。コンピュータはこの約束事にしたがって、順に指示されたとおりの作業を行っていきます。

この指示の単位となる 0 と 1 の並びを「機械語命令」と呼びます。1 つ 1 つの機械語命令は、非常に単純な作業しか行わせることはできませんが、この単純な作業を何千、何万と組み合わせることにより、コンピュータに非常に複雑な作業を行わせることができます。機械語命令をたくさん組み合わせて、コンピュータの行うべき一連の作業を指定したものを「機械語プログラム」と呼びます。

メモ

コンピュータに一連のまとまった作業をさせるには、この機械語プログラムを与えてやらなければなりませんが、1つ1つの機械語命令がどのような作業に対応するかについての約束事は一般に非常に複雑であり、また、1つ1つの機械語命令は非常に単純な作業しか行わせることができないため、ある程度のまとまった作業をコンピュータに行わせるために必要な機械語プログラムを人間が直接作るのは相当に困難です。そこで、もっと人間が理解しやすい書き方でコンピュータの行うべき仕事を記述し、それを機械語プログラムに機械的に変換することでコンピュータに仕事をさせるという方法がよく用いられます。この、より人間が理解しやすい書き方も、一定の約束事 (どう書けば、どう働くか) に基づいて書かれますが、ちょうど、人間の日常の言語に英語や日本語、中国語の違いがあるように、この「より人間が理解しやすい書き方」にも、その用途に応じていろいろなものがあります。機械語に対して、より人間が理解しやすいプログラミング言語を総称して、一般に「高級 (プログラミング) 言語」と呼びます[1]。

メモ

1.2 CPU の内部構成

コンピュータの演算・制御装置である **CPU** は、メモリに記憶されている機械語プログラムを読み取り、その指示に従って、四則演算などの計算や、各装置の制御、装置間でのデータの転送を行います。パソコン等で使用されている CPU の内部は、おおよそ次の図のような構成になっています。

キャッシュメモリ　メモリに記憶されているデータの内、CPU が頻繁に読み書きする部分のコピーを記憶しておく装置です。主記憶装置のメモリは、通常 DRAM (ダイナミック RAM) で構成されますが、キャッシュメモリは、より高速な SRAM (スタティック RAM) で構成されます。キャッシュメモリの記憶容量は主記憶装置に比べると僅かなものです[2]が、アクセスタイムがより小さく (高速に) なっています。CPU がメモリ中のデータを必要とする場合、まず、このキャッシュメモリに読み込まれてから使用されます。また、メモリにデータを格納する場合、とりあえずキャッシュメモリに記憶して、その後メモリに格納されます。しかし、これらの作業は、自動的に行われますので、機械語プログラムがキャッシュメモリの存在を意識する必要は特にありません。

[1] ここでの「高級」という語は「優れている」という意味ではないので誤解しないでください。

[2] 通常、主記憶は数百 MB から数十 GB、キャシュメモリは数百 KB から大きくても十数 MB 程度の大きさです。

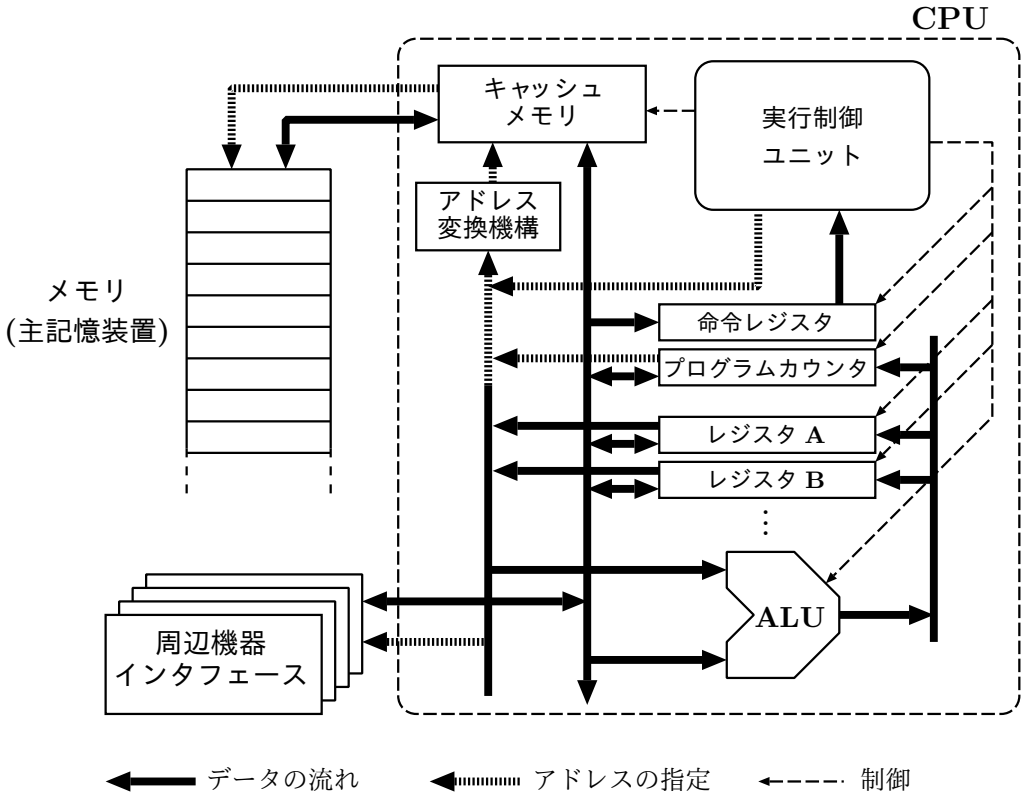


図 1: CPU の内部構成 (模式図)

メモ

実行制御ユニット　メモリから (キャッシュメモリを介して)「命令レジスタ」に読み込まれた機械語命令を解読して、そこに指示されている作業を実行します。1 つの機械語命令の実行が終ると (通常は) 次のアドレスに格納されている機械語命令を「命令レジスタ」に読み込み、それをまた実行するといった処理を繰り返していきます。CPU が実行できる機械語命令として、最低でも次のような機能を持つ命令群が用意されているのが普通です。

- ロード命令 — 定数データあるいはメモリ中の特定のアドレスのデータを (キャッシュメモリを介して) CPU の特定のレジスタにコピーします。
- ストア命令 — CPU の特定のレジスタに記憶されているデータを (キャッシュメモリを介して) メモリ中の特定のアドレスにコピーします。
- 演算命令 — レジスタに記憶されているデータの間で四則演算などの計算を (ALU を使って) 行い、その計算結果を特定のレジスタに格納します。

- 分岐命令 — 今実行した機械語命令の次のアドレスに格納されている命令に進むのではなく、別の離れたアドレスから機械語命令の実行を続けます。無条件に分岐を行う命令だけではなく、演算命令の結果に基づいて、特定の条件 (先に実行した計算の結果が 0 であった、負であったなど) が成り立ったときだけ分岐する命令も用意されています。

メモ

命令レジスタ　次に実行する機械語命令 (0/1 の列) を保持する働きを持ちます。「プログラムカウンタ」で指示されたアドレスのデータが、メモリから (キャッシュメモリを介して) この「命令レジスタ」に読み込まれ、「実行制御ユニット」によって機械語命令として解釈されて実行されます。

メモ

プログラムカウンタ　次に実行する機械語命令のアドレスを記憶します。たとえば、Intel 社の Core i3 プロセッサはアドレスを 64bit (または 32bit[3]) の符号なし整数で表現しますので、このような CPU の「プログラムカウンタ」は 64bit (または 32bit 長) のデータを記憶します。

メモ

レジスタ　計算に使うためにメモリから読み込まれたデータや、複雑な計算の途中結果などを記憶します。通常、1 つの CPU に、数個から数百個のレジスタが用意されています。一般的な用途に使用される**汎用レジスタ**と呼ばれるレジスタもあれば、浮動小数点データの処理専用のレジスタや、アドレスの処理専用のレジスタなど、特定の用途専用のレジスタもあります。64bit (32bit) CPU と呼ばれる CPU の汎用レジスタやアドレス処理用のレジスタは、各レジスタが 64bit (32bit) 長のデータを記憶することができるのが普通です。

メモ

[3] この CPU は、32bit CPU としても 64bit CPU としても動作することができます。

ALU　2つのデータ間の四則演算などの計算を行います。ALU は「算術・論理ユニット[4]」とともに呼ばれ、符号付き、あるいは符号なし整数の四則演算、0/1 の列のビット毎の論理演算[5]、浮動小数点データの数値の四則演算[6]などを行うことができます。

メモ

1.3　コンパイラとは

高級言語はコンピュータが直接理解することはできませんから、何らかの方法でコンピュータが実行できるようにしてやらなければなりません。これには大きく分けて 2 つの方法があります。その 1 つは、「**インタプリタ** (通訳)」と呼ばれる機械語プログラムによって、高級言語で書かれたプログラムを少しずつ解釈しながら、そこで指示されている通りの作業を行っていく方法です。インタプリタは、高級言語で書かれたプログラムを実行することのできる仮想的なコンピュータとして働きます。

メモ

もう 1 つの方法は、コンピュータがそのプログラムの実行を始める前に、高級言語で書かれているプログラムをあらかじめ機械語にすべて翻訳 (変換) しておく方法です。一旦翻訳が済んでしまえば、翻訳済の機械語プログラムは何度でも利用することができます。この翻訳作業のことを「**コンパイル**」と呼び、翻訳を行ってくれる仕組みを「**コンパイラ**」と呼びます。コンパイルする前の高級言語で書かれたプログラムを「**ソース (原始) プログラム**」あるいは「**ソースコード**」と呼び、コンパイルして得られた機械語プログラムのことを「**オブジェクト (目的) プログラム**」あるいは「**オブジェクトコード**」と呼びます。コンパイラ自身もやはり 1 つの機械語プログラムとして実現されます。

メモ

[4]ALU は Arithmetic and Logic Unit の略です。

[5]NOT、AND、OR、XOR などの演算を (たとえば 32bit の) 0/1 の列に対して bit 毎に行います。

[6]平方根の計算ができるような ALU もあります。

1.4 コンパイラの具体例

1 号館 542 実習室や 609 実習室のパソコンでは、Intel 社の Core i3 シリーズの CPU が使われています。この CPU には、64bit 長 (または 32bit 長) のレジスタがいくつか用意されており、ALU を使って、レジスタ間で四則演算などの計算を行ったり、メモリとレジスタの間でデータを転送したりすることができます。これらの実習室で利用している C コンパイラは、ソース言語を C として、この CPU の機械語をオブジェクト言語としたコンパイルを行うわけです。次は、これらの実習室の Linux 環境の C コンパイラ (cc) を使って、ある C プログラム (ソースプログラム) をコンパイルしてみた結果です。

メモ

ソースコード	オブジェクトコード (アセンブリ言語で書いたもの)	機械語の意味
`int fact (int n)`	`fact:`	関数 fact の機械語命令の始まり
`{`		
`  int i, x;`		
`  x = 1;`		
`  for (i = 2;`	`  cmpl $1, %edi`	レジスタ edi (引数 n) を定数 1 と比較
`       i <= n;`	`  jle .L4`	edi が 1 以下ならば .L4 にジャンプ
`       i ++)`	`  movl $1, %eax`	定数 1 をレジスタ eax (変数 x) に格納
	`  movl $2, %edx`	定数 2 をレジスタ edx (変数 i) に格納
	`.L3:`	
`    x = x * i;`	`  imull %edx, %eax`	edx (変数 i) を eax (変数 x) に掛け込む
	`  incl %edx`	edx (変数 i) を 1 増やす
	`  cmpl %edx, %edi`	edi (引数 n) を edx (変数 i) と比較
	`  jge .L3`	edi が edx 以上ならば .L3 へジャンプ
`  return x;`	`  rep ret`	呼び出し元へリターンする
	`.L4:`	
	`  movl $1, eax`	定数 1 を eax (戻り値) に格納
	`  ret`	呼び出し元へリターンする
`}`		

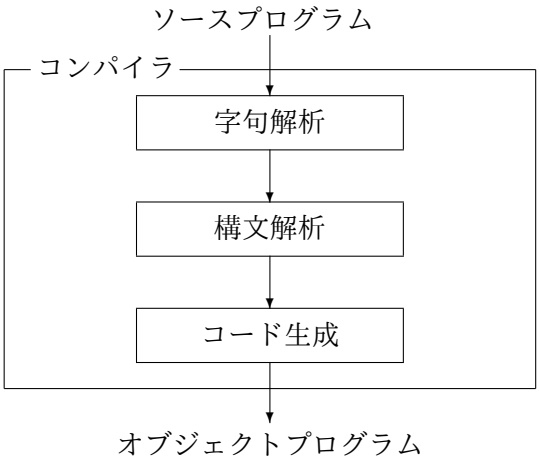
ただし、

- オブジェクトコードの欄の 1 行 1 行が、1 つの機械語命令を表しています。
- `fact` や `.L3`、`.L4` はラベルと呼ばれ、そのラベルの置かれた場所のアドレスを表しています。ラベルは機械語の命令を読みやすくするためのもので、それ自身は命令ではありません。
- `eax`、`edx`、`edi` は 32bit 長のレジスタの名前です。
- 関数が呼ばれる時には、その第 1 引数がレジスタ `%edi` に置かれる取り決めになっています。
- 関数の戻り値は、レジスタ `eax` に格納して呼び出し元に戻ることになっています。

メモ

1.5 コンパイラの内部構成

最も基本的なコンパイラは、下図のように、字句解析部 (lexical analyzer あるいは scanner)、構文解析部 (syntax analyzer あるいは parser)、コード生成部 (code generator) から構成されます。



字句解析部　字句解析部は、入力として与えられたソースコード (ファイルに格納された単なる文字列) を、**トークン (token)** と呼ばれるソース言語の基本的な字句に分解します。(変数名などの) 識別子、`if` や `while` などの予約語、`3.1459` のような数値を表す文字の列などがトークンとなり、トークンの列としてソースコードが認識されます。

ソース言語やコンパイラの実現方法によっては、字句解析部でトークンへの分解を行うとともに、高水準言語レベルでのプログラムの変換が行われることもあります。C プログラムにおける `#include` や `#define` などの処理がその一例です。この処理はコンパイラとは独立したプログラムで行われる場合もあり、このような前処理を行うプログラムを**プリプロセッサ**と呼びます。

メモ

構文解析部　構文解析部は、字句解析部で得られたトークンの列から、ソース言語の文法にしたがって、入力されたソースコードの構文を認識します。ほとんどの実用的なコンパイラでは、この作業を行った後、次のコード生成に移る前に、制約検査部と呼ばれる部分で、変数の宣言などに整合性が取れているかどうかなど、文法のみでは規定されていない部分について、入力されたソースコードの妥当性の検査を行います。

メモ

コード生成部　コード生成部は、構文解析部で得られた結果を解釈し、目的のオブジェクト言語で記述されたオブジェクトコードを生成します。実用的なコンパイラのほとんどは、この段階で、まず**中間コード**と呼ばれる仮想的な機械語のプログラムを生成し、それに対して**最適化**と呼ばれる生成されるオブジェクトコードの実行速度や大きさの面での効率化を行い、この最適化後の中間コードから、最終的なオブジェクトコードの生成を行います。また、オブジェクトコードそのものではなくアセンブリ言語のプログラムを生成し、これをアセンブラ (アセンブリ言語で記述されたプログラムを機械語へ変換するプログラム) を別に起動することで機械語へ変換するコンパイラも多くあります。

メモ

記号処理・第1回・終わり