1 言語処理

- ♣ 「言語処理」に関する用語や基礎知識
- ♣ コンパイラはどんな処理を行っているのか?

1.1 プログラミング言語

FORTRAN: 科学技術 計算用

COBOL: 事務 計算用

PL/I: 汎用 言語 (巨大)

BASIC, C: 汎用言語

C++: C に オブジェクト指向 の考えを導入

Java: オブジェクト指向, ネットワーク指向

LISP: 関数 型言語, リスト 処理言語

Prolog: 論理 型言語

Perl, Ruby, Python, JavaScript: スクリプト言語

Scala, Erlang: 並行処理 指向言語

...

1.2 言語処理系とその種類

1.2.1 言語処理系

言語処理系 (言語プロセッサ, language processor)

- 計算機用言語で書かれた プログラム や データ を処理するプログラムの総称
- 1) 読み込んで, 2) 解析 し, 3) 他の形式に 変換 したり, 書かれていることに従って 計算 を行なったりする.

1.2.2 種々の言語処理系

1. トランスレータ (translator)

   プログラムやデータを他の形式に 変換 するプログラムの総称

2. コンパイラ (compiler)

   プログラミング 言語で書かれたプログラムを, 計算機の 機械語 やそれに近い形式に変換 (翻訳) するプログラム (トランスレータの一種)

3. アセンブラ (assembler)

アセンブリ言語 (assembly language)

`lw`, `addu` 等の機械語の表記法 (ニモニック (mnemonic)) を用いたプログラミング言語

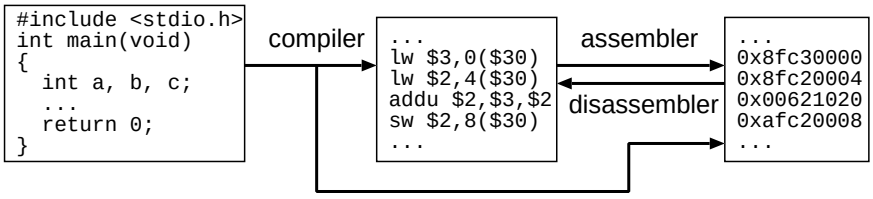
アセンブラ

アセンブリ言語を機械語に変換するトランスレータ

逆アセンブラ (disassembler)

機械語をアセンブラに逆変換するトランスレータ (機械語の解析に用いる)

コンパイラには, アセンブリ言語を生成し, アセンブラを起動して機械語に変換するものが多い [1]



4. プリプロセッサ (前 処理系, preprocessor)

コンパイルの前処理としての 小規模 な変換を行なう

例) C の '#' で始まる文[2]

```
#define MAX 1024
#define ax(k,i) (a[(k)*2+(i)])
    ...
int a[MAX], b, c;
b = MAX + 1;
c = ax(3,5);
```

⇒

```
    ...
int a[ 1024 ], b, c;
b =  1024  + 1;
c =  (a[(3)*2+(5)]) ;
```

C++ をプリプロセッサで C に変換してから, C コンパイラでコンパイルするものもある[3]

5. インタプリタ (interpreter)

プログラミング言語で書かれたプログラムを機械語に変換しないで直接 実行 するプログラム

(例) 「`int a, b;`」を読んだら, 変数 `a` を 16 番地, `b` を 20 番地等のように決める

「`a = b+1;`」を読んだら, 20 番地の値に 1 を足して, 結果を 16 番地に格納する

(例) Java は, コンパイラで中間言語を生成し, それをインタプリタで実行する[4]

長所

– OS や CPU に依存せずにプログラムを配布可能

短所

– 実行が 遅い

---

[1] GCC も内部で一旦アセンブリ言語を生成している. `-S` オプションを用いて `gcc -S test.c` でコンパイルすると, アセンブリプログラムが `test.s` に生成される.

[2] GCC のプロプロセッサは `cpp` である. 例えば `test.c` という C プログラムがあるとき, `cpp test.c` でプリプロセッサの出力を見ることができる.

[3] G++ はこの処理方式をとっている

[4] `javac` はコンパイラで, Java VM という仮想計算機の機械語 (バイトコード) が生成する. `java` はインタプリタで, この機械語をソフトウェア実行する.

6. JIT コンパイラ (Just In Time compiler)

   コンパイルを 実行 時に行う

   見掛け上は インタプリタ として動作するが, 内部で変換した機械語を実行する

   長所

   – OS や CPU に依存しないソースプログラム (や中間言語) を配布可能
   – 実行が 高速 (インタプリタよりは)

   短所

   – 実行時に コンパイル のオーバヘッドが生じる
   – コンパイルに時間がかけられないため 最適化 の強度が限られる

7. バイナリ トランスレータ (binary translator)

   ある CPU 用の機械語を 別の CPU 用の機械語に変換

   (例 1) Java バイトコード → x86 機械語 (高速 化のため)

   (例 2) x86 機械語 → ARM 機械語 (エミュレーション のため)

   (いずれも just in time 方式)

1.3 言語処理系の開発

☆ C や Java のプログラムを, (コンパイラがない) 新しい計算機/プロセッサ上で実行したいとき, どうするか?

1. ハンド・コンパイルする (人手で 機械語に直す) → 大変
2. コンパイラを アセンブリ 言語で書く → 超大変
3. クロスコンパイラ (cross compiler) を開発する

   クロスコンパイラとは

   – 通常のコンパイラは, ある計算機上で動いて, その 計算機用の機械語を生成する (native compiler と呼ばれる)
   – クロスコンパイラは, ある計算機上で動いて, 別の 計算機用の機械語を生成する

     (例) PC (Core i7) 上で動作し, スマホ用 ARM プロセッサの機械語を生成する

     (例) PC (Core i7) 上で動作し, 炊飯器制御用マイコンの機械語を生成する

   native コンパイラがあれば, クロスコンパイラの開発はその プログラミング 言語で行える

   (例) PC 上で GCC が動けば, スマホ用 ARM のコードを生成する C 言語のクロスコンパイラは C 言語 で開発できる

4. コンパイラ・コンパイラ (compiler-compiler) を用いる

   言語の 文法 と 意味 の記述を入力 ⇒ コンパイラを 自動生成

   Unix の yacc (GNU の bison) が有名

5. コンパイラのリターゲッティング (retargeting)

   ターゲット = コンパイラがコード生成の 対象 とする計算機

   リターゲッティング = ターゲットの 変更

   コンパイラのターゲット 依存 部分だけを書き換える

   GCC 等多くのコンパイラでは, ターゲット依存部分とそうでない部分が明確に分離されている

6. コンパイラとインタプリタの組合せ

   コンパイラで 中間言語 にまで変換し, これをインタプリタで実行する (Java の方式)

   中間言語は (インタプリタ さえ作れば) どの計算機上でも実行できる

   – 中間言語のインタプリタはコンパイラよりも開発が 容易 [5]

   – ただし, 実行は若干 遅い [6]

1.4 コンパイラの処理

1. 字句 解析 (lexical analysis)

   プログラム (= 文字 の列) を 単語 の列に変換

2. 構文 解析 (parsing), 文法 解析 (syntax analysis)

   プログラムの文法構造を解析し, 抽象構文木 (abstract syntax tree) を生成

3. 意味 解析 (semantic analysis)

   プログラムの意味を分析し, 意味を表す木や 中間 言語を生成

4. 最適 化 (optimization)

   プログラムの 実行速度 や コードサイズ を改善

5. コード 生成 (code generation)

   機械語やアセンブリ言語等を生成

Nagisa ISHIURA

[5] インタプリタの実装は比較的容易だが, 実行環境 (クラスライブラリ等) の開発は非常に大変, というのが実際的には問題である.

[6] Java ではこの解決のために, JIT (just-in-time) コンパイラや, Java チップ (バイトコードを実行する LSI) などの技術が用いられる.