コンパイラ理論

櫻井彰人

目的

- コンパイラの基礎（理論と実際）を、ツールを使って、小さいコンパイラを作りながら、学ぶ

講義内容

1. コンパイラの基礎
2. 言語理論から
3. 構文解析とYacc
4. 再帰下降型構文解析とLR構文解析
5. 演算子優先順位と結合性
6. 字句解析とlex
7. 意味解析と記号表

参考書（理論寄り）

- 原田賢一, **コンパイラ構成法,** 共立出版, 1999
- 中田育男, **コンパイラ,** オーム社, 1995.
- A.V. Aho, R. Sethi, J. D. Ullman. “Compilers: Principles, Techniques and Tools", Addison-Wesley, 1985
- A. V. エイホ, R. セシィ, J. D. ウルマン. "**コンパイラ I , II** －原理・技法・ツール－." サイエンス社, 1990.
- A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman. "Compilers: Principles, Techniques, & Tools," Addison-Wesley, 2006.

参考書（実際的）

- 石田綾, スモールコンパイラの製作で学ぶプログラムのしくみ, 技術評論社, 2004.
- 日向俊二, やさしいコンパイラの作り方入門, カットシステム, 2009.
- 前橋和弥, プログラミング言語を作る, 技術評論社, 2009.
- 青木峰郎, ふつうのコンパイラをつくろう, ソフトバンククリエイティブ, 2009.
- 原悠, Rubyで作る奇妙なプログラミング言語, 毎日コミュニケーションズ, 2008.
- Terence Parr, Language Implementation Patterns, Pragmatic Bookshelf, 2010.

採点

- レポート2回～
- 内容・方法は未定

プログラムはどう処理されるか?

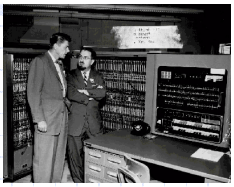
- 2つの代表的方法:
  - インタプリタ (より古い, 研究は少ない)
  - コンパイラ (より新しい, かなり広く研究されている)
- インタプリタはプログラムを「そのまま」実行する
  - 前処理はほんの少しか殆ど行わない
- コンパイラは徹底した前処理といえる
  - 非常に多くの場合、コンパイラ

スクリプト言語として蘇った

語源

- interpreter: 翻訳者
  - Interpret: 翻訳する
- compiler: まとめる人
  - Compile: 重ねる、まとめる
- assembler: 組み立てる人
  - Assemble: 組み立てる

高級(high-level)言語の誕生



Ronald Reagan and Watson Laboratory's Herb Grosch at an IBM 701 in 1954
http://www.columbia.edu/cu/computinghistory/701.html

- 1953年 IBM は 701 を作る
- プログラミングはすべて、アセンブラで
- 問題: ソフトウェアコストは、ハードウェアコスト以上
- John Backus: "Speedcoding"
  - インタプリタ
  - 手で書いたアセンブラより 10-20 倍遅い!

プログラムの始まり

- von Neuman
  - Stored program のアイデアを出した人

FORTRAN I

- 1954年 IBM は 704 を開発
- John Backus
  - アイデア: 高級コードをアセンブラに翻訳しよう!
  - 不可能だと考えた人は多い
- 1954年～7年 FORTRAN I プロジェクト
- 1958年には, ソフトウェアの 50% 以上が FORTRAN で書かれる
- 開発期間の大幅短縮
  - (2 週間 ⇒ 2 時間)

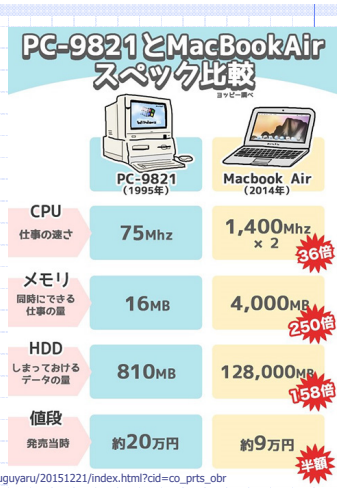
メモリ容量

- PB = 1024 TB , 1000TB
- TB = 1024 GB , 1000GB
- GB = 1024 MB , 1000MB
- MB = 1024 KB, 1000KB
- KB = 1024 B, 1000B

- GB = 10^9 B, 1024^3 B

脱線

- PC-9801VM (1985ごろ)
  - V30/10MHz
  - 640KB
  - 5インチFDD
  - 30~40万円



http://time-space.kddi.com/digicul-column/suguyaru/20151221/index.html?cid=co_prts_obr

FORTRAN I

- 史上初のコンパイラ
  - 手で書いたものと殆どおなじくらい良いコード
  - 計算機科学に与えた影響はあまりに大きい
- 膨大な理論的研究を生み出すもととなった
- 現代のコンパイラはいずれも FORTRAN I の概要は持っている

FORTRAN II

```
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
      READ INPUT TAPE 5, 501, IA, IB, IC
  501 FORMAT (3I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C IS GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
      IF (IA) 777, 777, 701
  701 IF (IB) 777, 777, 702
  702 IF (IC) 777, 777, 703
  703 IF (IA+IB-IC) 777,777,704
  704 IF (IA+IC-IB) 777,777,705
  705 IF (IB+IC-IA) 777,777,799
  777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
  799 S = FLOATF (IA + IB + IC) / 2.0
      AREA = SQRT( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) *
     +     (S - FLOATF(IC)))
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
  601 FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2,
     +        13H SQUARE UNITS)
      STOP
      END
```

注: 等幅フォントで表示する必要があります

http://en.wikipedia.org/wiki/Fortran

コンパイラの目的

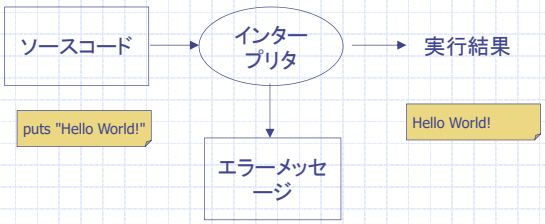
- 必要性は，現代では，自明．高級言語(C, Java, ...)
- コンパイルの過程は、大きく、2つに分かれる：ソースプログラムの解析とオブジェクトコードの生成

```
int main(int argc, char** argv) {
  puts("Hello, world!");
}
```

```
.LC0:
        .string "Hello, world!"
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        subl    $28, %esp
        pushl   $.LC0
        call    puts
        leave
        ret
```

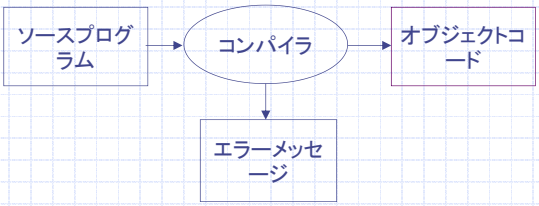
インタープリタ

ソースプログラムを解析して、即座に実行してしまう



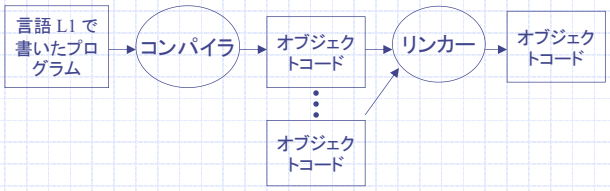
コンパイラ

- ソースプログラムを解析して、オブジェクトコードを生成する

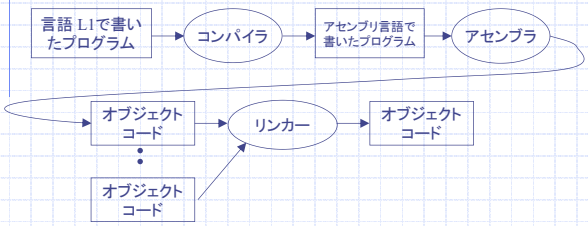


オブジェクトコード

- 絶対番地で書かれた機械語
- リロケータブルな機械語
- アセンブリ言語で書かれたプログラム
- 他のプログラム言語で書かれたプログラム

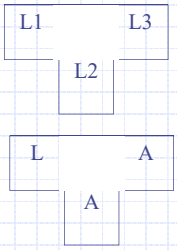


アセンブリ言語への翻訳



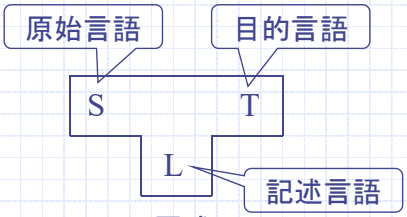
T図式

コンパイラ･トランスレータの機能の図式表現



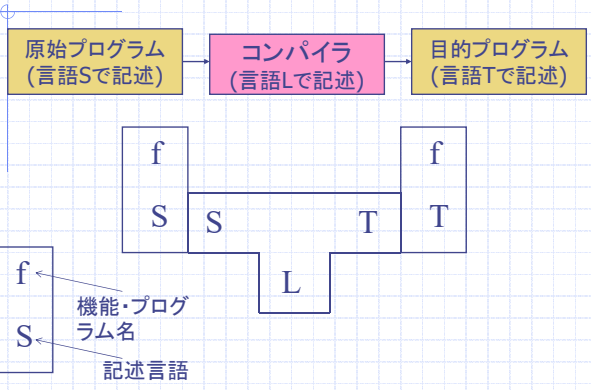
T図式

- 原始言語 S で書いたプログラムを目的言語 T で書かれたプログラムに変換する、言語 L で記述されたコンパイラ



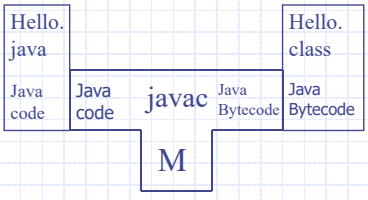
T図式

T図式

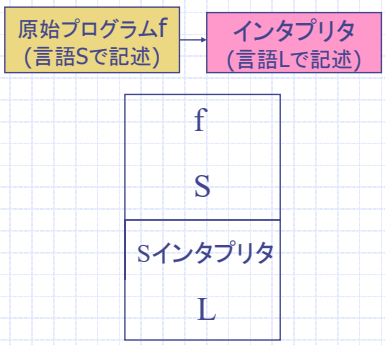


T図式

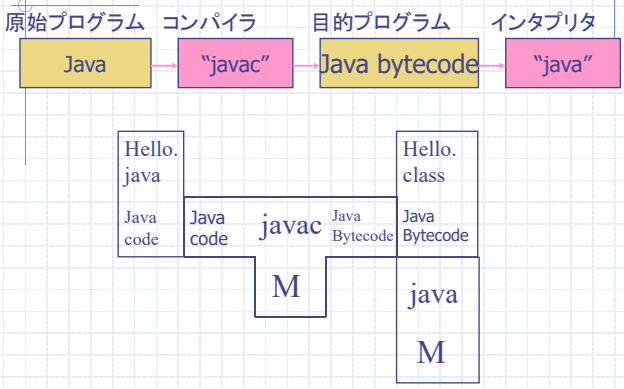
例 : Java を JBC(Java byte code) に変換する機械語 M で記述された javac コンパイラ



T図式(インタプリタの場合)



T図式(Javaの場合)

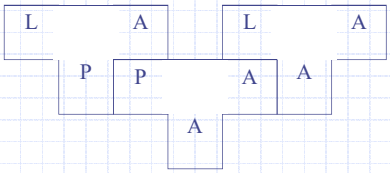


様々な技術

- 直接開発
- ブートストラップ
- クロスコンパイラ
- 仮想マシン
- Just-in-time コンパイラ

ブートストラップ

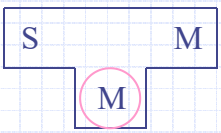
アセンブリ言語での実装を避けるには？



コンパイラの作成

計算機 M 上で動く高水準言語 S のコンパイラが欲しい

必要なコンパイラ



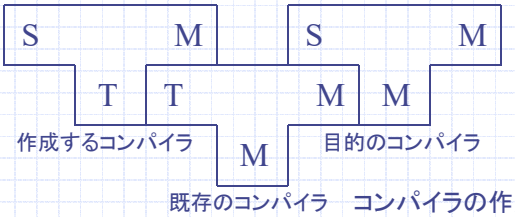
しかし機械語 M で
プログラムは難しい

既存の高水準言語
コンパイラを利用

コンパイラの作成

計算機 M 上で動く高水準言語 S のコンパイラが欲しい

計算機 M 上で動く高水準言語 T のコンパイラを利用

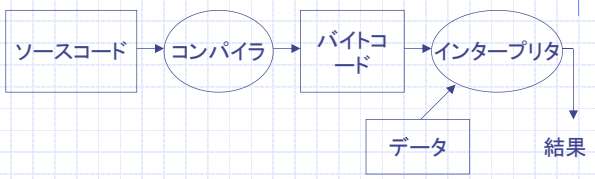


コンパイラの作成は
高水準言語で行える

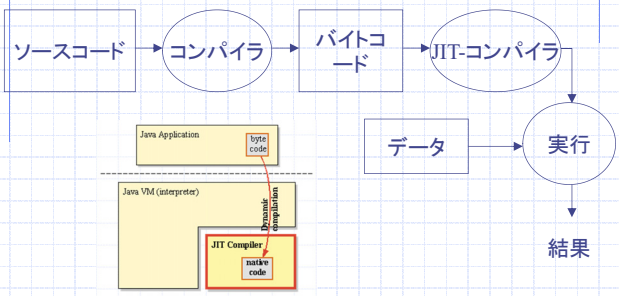
クロスコンパイラ、機種非依存コンパイラ

- あるプラットホーム上で走って、他のプラットホーム用のコードを生成するコンパイラ
- 機種非依存、可搬型コンパイラ

仮想マシン



Just-in-time コンパイラ

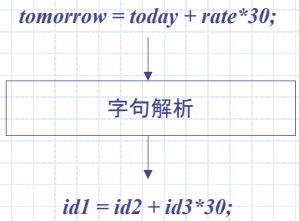


バイト・コードを実行時に動的に機械語に変換（コンパイル）する
http://www.trl.ibm.com/projects/jit/jitanim.gif

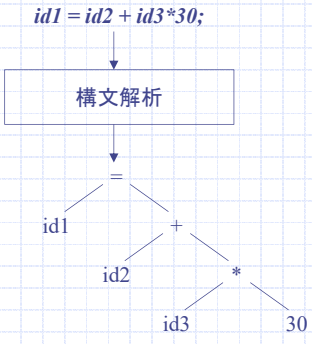
コンパイルのフェーズ

- コンパイルのフェーズ（おおまか）:
  - 字句解析 lexical analysis
  - 構文解析 syntax analysis
  - 意味解析 semantic analysis
  - 最適化 optimization
  - コード生成 code generation

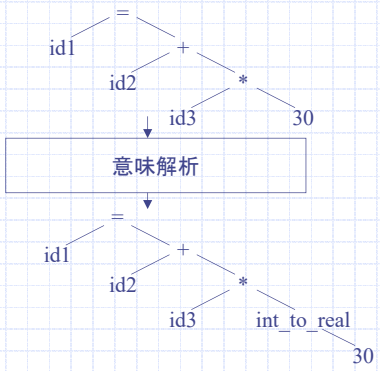
字句解析



構文解析



意味解析



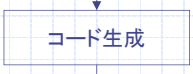
コード最適化

temp1 = int_to_real(30)
temp2 = id3*temp1
temp3 = id2 + temp2
id1 = temp3

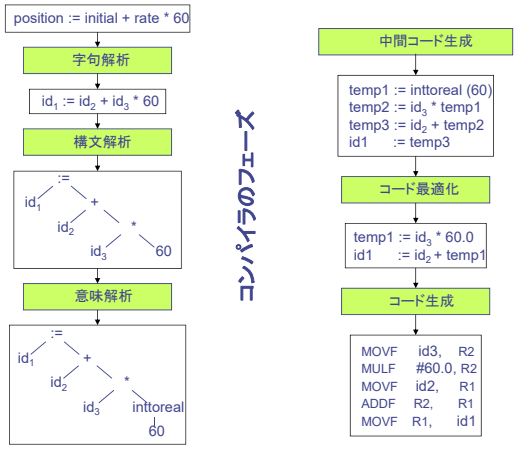
temp1 = id3* 30.0
id1 = id2 + temp1

コード生成

temp1 = id3* 60.0
id1 = id2 + temp1




```
loada     id3
loadbi    60.
mul
store     temp
loada     id2
loaddb    temp
add
store     id1
```



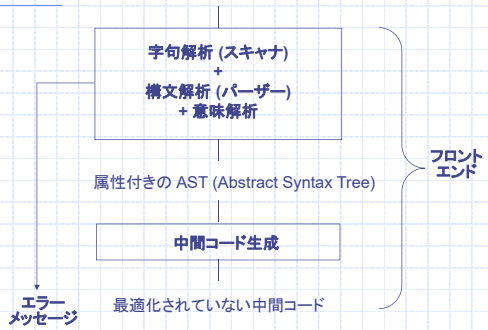
フロントエンドとバックエンド

- コンパイルのフェーズで、ソース言語の方に(ターゲット言語へと比べて)より近いフェーズをフロントエンド(front-end)と呼ぶ
- コンパイルのフェーズで、ターゲット言語の方に(ソース言語へと比べて)より近いフェーズをバックエンド(back-end)と呼ぶ

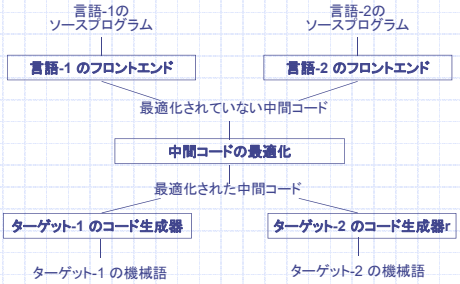
パス

- 一回のパスというのは、コンパイラの動作で(多くの場合ソース)コード全部を対象に処理すること

コンパイラフロントエンド



コンパイラのコンポーネント化



中間言語を用いることのよさ

1. **リターゲッティング** – 新規の機械用のコンパイラを作るとき、既存のフロントエンドに新規のコード生成器を作る.
2. **最適化** – コード最適化部分を再利用することにより、様々な言語や機械に対してコンパイラを作ることができる.

注: “中間コード”, “中間言語”, and “中間表現” はいずれも区別なく用いられる.