

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
8. 制御文の翻訳
9. 関数呼び出しとメモリ管理
10. 流れ解析
11. 命令選択

参考書(理論より)

- ◆ 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.
- ◆ .V. ho, 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(実際の)

- ◆ 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る**, 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つの代表的な方法:
 - 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

高級 (high-level) 言語の誕生

- ◆ 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!

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)

FORTRAN I

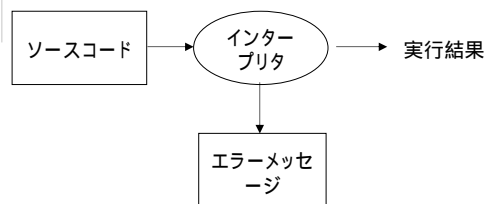
- ◆ 史上初のコンパイラ
 - 手で書いたものと殆どおなじくらい良いコード
 - 計算機科学に与えた影響はあまりに大きい
- ◆ 膨大な理論的研究を生み出すもとなった
- ◆ 現代のコンパイラはいずれも FORTRAN I の概要は持っている

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

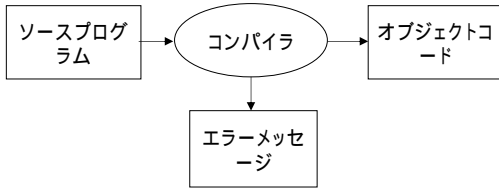
インタプリタ

ソースプログラムを解析して、即座に実行してしまう



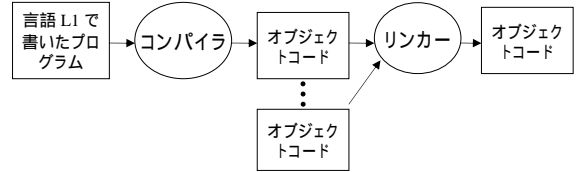
コンパイラ

- ◆ 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

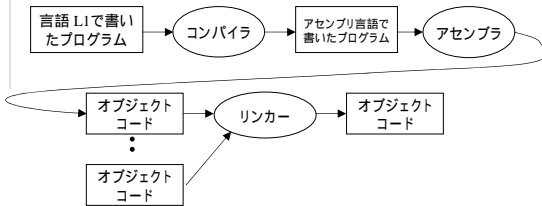


オブジェクトコード

- ◆ 絶対番地で書かれた機械語
- ◆ リロケートブルな機械語
- ◆ 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

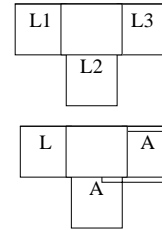


アセンブリ言語への翻訳



T図式

コンパイラ・トランスレータの機能の図式表現

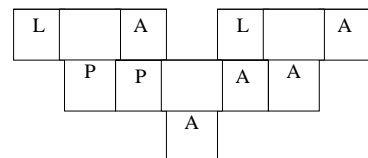


様々な技術

- ◆ 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

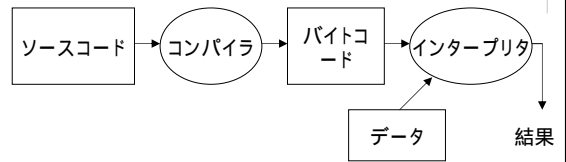
アセンブリ言語での実装を避けるには？



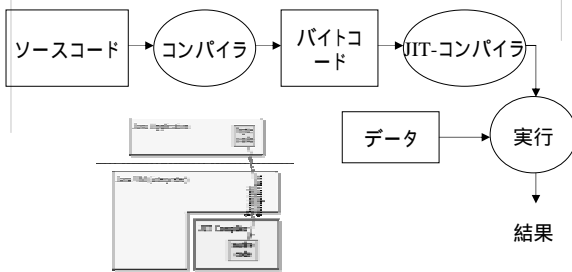
クロスコンパイラ、 機種非依存コンパイラ

- ◆あるプラットフォーム上で走って、他のプラットフォ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

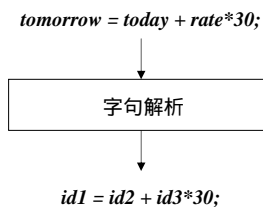


バイト・コードを実行時に動的に機械語に変換(コンパイル)する
<http://www.tri.ibm.com/projects/jit/jitanim.gif>

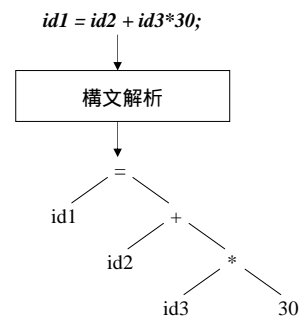
コンパイルのフェーズ

- ◆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

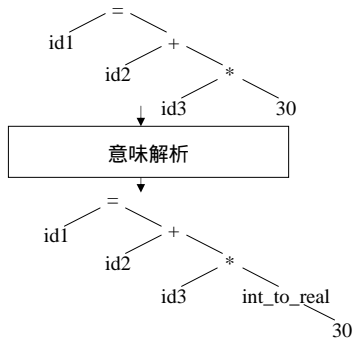
字句解析



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
```

最適化

```
temp1 = id3* 30.0
id1 = id2 + temp1
```

コード生成

```
temp1 = id3* 60.0
id1 = id2 + temp1
```

コード生成

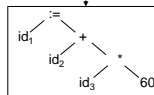
```
loada    id3
loadbi   60.
mul
store   temp
loada   id2
loaddb  temp
add
store  id1
```

position := initial + rate * 60

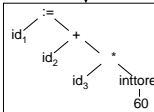
字句解析

$id_1 := id_2 + id_3 * 60$

構文解析



意味解析



コンパイラのフェーズ

中間コード生成

```
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

コード最適化

```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

コード生成

```
MOVf id3, R2
MULF #60.0, R2
MOVf id2, R1
ADDf R2, R1
MOVf R1, id1
```

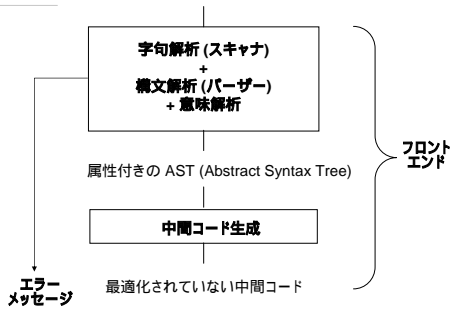
フロントエンドとバックエンド

- ◆ 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

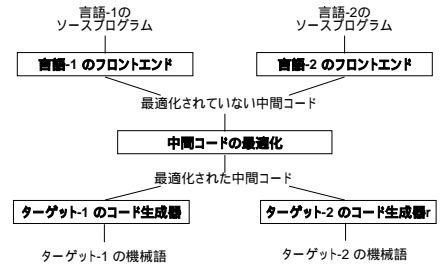
パス

- ◆ 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. **リターゲットング** – 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。