コンパイラ

第1回 コンパイラの概要

http://www.info.kindai.ac.jp/compiler
E館3階E-331 内線5459
takasi-i@info.kindai.ac.jp

1

本科目の内容

- コンパイラ(compiler)とは何か
- コンパイラの構成
- コンパイラの作成方法
  - 字句解析
  - 構文解析
  - 制約検査
  - コード生成
  - 最適化

情報システムプロジェクトIと連携

2

成績について

評価基準	
各週課題	30%
定期試験	70%

- 無届欠席禁止
  - やむを得ず欠席した場合は翌週までに連絡すること
  - 無届欠席が複数回ある場合は試験の点数に関わりなく不受となる

オンライン授業では GoogleClassroom から出席カードが提出されれば出席とします

3

2021年度

学年	コース	受講者数	合格	不可	不受	合格率
3	システム	72	67	2	3	97%
4	メディア	1	1	0	0	100%

2022年度

学年	コース	受講者数	合格	不可	不受	合格率
3	システム	87	83	2	2	97%
3	メディア	1	1	0	0	100%
4	メディア	2	1	1	0	50%

※全出席し、全レポートを〆切までに提出して不可になった受講生はいない

4

5

6


classroom.google.com
クラスに参加
参加
現在、次のメールアドレスでログインしています
@info.kindai.ac.jp
アカウントを切り替える
クラスコード
教師にクラスコードを聞いてこちらに入力してください。
クラスコード
クラスコードを使用してログインするには
承認済みアカウントを使用します
コンパイラ zokgxoo
シスプロ1 cr7xwqo

7


classroom.google.com
Google Classroom
ToDo
チェックが必要な課題
カレンダー
2023-基礎線形代数学...
情報学科情報学部1年生
2023-コンパイラ
理工学部情報学科情報システムコース...
2023-基礎微分積分学...
情報学部情報学科1年生
2023-情報システムプ...
理工学部情報学科情報システムコース...
プログラミング基礎1
2023年度
2023-社会情報学実習...

8


classroom.google.com
2023-コンパイラ
理工学部情報学科情報システムコース3年生
ストリーム
授業
メンバー
採点
カスタマイズ
2023-コンパイラ
理工学部情報学科情報システムコース3年生
Meet
リンクを生成
クラスコード
zokgxoo
期限間近
クラスへの連絡事項を入力
石水隆 さんが新しい資料を投稿しました: Slack について
石水隆 さんが新しい資料を投稿しました: 第4回 講義資料
石水隆 さんが新しい資料を投稿しました: 第3回 講義資料

9


info.kindai.ac.jp
◆コンパイラ◆
https://www.info.kindai.ac.jp/compiler/
連絡
出席について
単位取得には原則として全ての授業に出席する必要があります。 やむを得ず欠席する場合はその要週までに必ず欠席届を出してください。 欠席届無しの欠席が複数回ある場合は履修の意思無しと見做して不受験にします。
オンライン授業では、当日 GoogleClassrooom から出席カードが提出されていれば出席扱いにします。
課題について
単位取得には原則として全ての課題テストを受講する必要があります。 正当な理由のある場合を除いて原則として締切を過ぎた受講は認めません。
講義資料
第1回 コンパイラの概要 (4/12) パワーポイント PDF ノート用PDF (4/3 update)
第2回 形式言語と形式文法 (4/19) パワーポイント PDF ノート用PDF (4/3 update)
第3回 字句解析(1) (4/26) パワーポイント PDF ノート用PDF (4/3 update)
第4回 字句解析(2) (5/10) パワーポイント PDF ノート用PDF (4/3 update)
課題テスト

10


classroom.google.com
2023-コンパイラ
理工学部情報学科情報システムコース3年生
ストリーム
授業
メンバー
採点
作成
Google カレンダー
クラスのドライブ フォルダ
すべてのトピック
第4回 : 字句解析(2)
第3回 : 字句解析(1)
第2回 : 形式言語と...
第1回 : コンパイラ...
Slack について
第4回 : 字句解析(2)
第4回 講義資料
投稿日: 3月24日
第4回 課題
各週課題
下書き
第3回 : 字句解析(1)
第3回 講義資料
投稿日: 3月24日
第3回 課題
各週課題
投稿予定: 4月20日 8:00
第2回 : 形式言語と形式文法

11


classroom.google.com
2023-コンパイラ
理工学部情報学科情報システムコース3年生
ストリーム
授業
メンバー
採点
第2回 講義資料
投稿日: 3月23日
第2回 課題
各週課題
投稿予定: 4月13日 8:00
第1回 : コンパイラの概要
第1回 講義資料
投稿日: 3月23日
第1回 課題
各週課題
期限: 4月19日
Slack について
Slack について
投稿日: 昨日

12

13

導入

Javaプログラムの実行

Hello.java

```
public class Hello {
   public static void main (String args[]) {
      System.out.print("Hello! World!¥n");
   }
}
```

```
$ javac Hello.java
$ java Hello
Hello! World!
```

←これは？→

←実行→

Cプログラムの実行

Hello.c

```
#include <stdio.h>
int main () {
   printf ("Hello! World!¥n");
}
```

```
$ gcc -o Hello Hello.c
$ Hello
Hello! World!
```

実行の前にコンパイル(compile)を行う

14

機械語(machine language)

- 1,0 の並び
- 計算機で実行可能
- レジスタ, ビット操作が必要
- ハードウェアに依存

```
0001 0000 0101
0010 0000 1010
0000 1100 1110
0100 1111 0011
0101 0000 0001
```

⇩

- プログラムの作成が困難
- プログラムの理解が困難
- プログラムのデバグが困難

人間が機械語を直接操作するのは効率が悪い

15

アセンブリ言語 (assembly language)

- 機械語命令を簡略名で記述
  - レジスタ, ビット操作が必要
  - ハードウェア依存
- 番地・レジスタ等に名前
- 実行は機械語変換が必要

```
A       DC   5
B       DC   10
START LD   GR0, A
        ADD  GR0, B
        ST   GR0, A
```

⇩

- 機械語よりはプログラムの作成・理解・デバグが容易

しかしまだ人間がアセンブリ言語を直接操作するのは効率が悪い

16

高水準言語 (high level language)

- 命令が基本的に英語
- ハードウェアに依存しない
- 変数名、メソッド名等を付けられる
- メソッド、関数等を定義できる
  - C, Java 等

```
public class Sample {
   public static void main
         (String  args[])  {
      int n;
      int a[n] = new int[8];
      for (int i=0; i<n; ++i) {
         a[i] = i*2;
      }
      int x, y, z;
      if (x == 1) {
         System.out.
               print (y) :
      } else {
```

⇩

- 人間にとって理解し易い

しかし計算機はそのままでは高水準言語を理解できない

17

プログラミング言語の翻訳

- プログラミング言語は文法が明確
  ⇒計算機で“翻訳”可能

高水準言語のプログラム → 翻訳 → 低水準言語のプログラム

⇔自然言語は文法に曖昧性
⇒計算機での“翻訳”は難しい

18

プログラミング言語の文法

<文> ::= <if 文>
<while 文>
<for 文>
<式文>
⋮
“{” <文の並び> “}”
“;” (空文)

文として定義されている
もの以外はエラー

19

プログラミング言語の文法

<if文> ::= “if” “(” <式> “)” <文>
または
“if” “(” <式> “)” <文> “else” <文>

<式> ::= <項> “+” <項>

<項> ::= <因子> “*” <因子>

<因子> ::= <整数>
<変数>
“(” <式> “)”

全て厳密に
定義されている

20

コンパイラ (compiler)

- コンパイラ
  - 原始プログラム(source program)を
    目的プログラム(object program)に
    変換(翻訳)するプログラム

原始プログラム (source program) → 入力 → コンパイラ (compiler) → 出力 → 目的プログラム (object program)

21

原始プログラム (source program)

- 原始プログラム(source program)
  - 高水準言語(high level language)で記述
  - 人間がエディタで作成
  - そのままでは実行不可
  - C, Java 等

```
public class Sample {
    public static void main (String  args[])  {
        int n;
        int a[n] = new int[8];
        for (int i=0; i<n; ++i) {
            a[i] = i*2;
        }
        int x, y, z;
        if (x == 1) {
            System.out.print (y) :
        } else {
```

22

目的プログラム (object program)

- 目的プログラム(object program)
  - 低水準言語(low level language)で記述
    (高水準言語を出力するコンパイラもある)
  - 高水準言語からコンパイラが変換
  - 実行可能なプログラムもある
  - 機械語, アセンブリ言語

```
0  PUSHI 0
1  POP    5
2  PUSH 5
3  PUSH 1
4  COMP
5  BGE   20
6  JUMP 11
7  PUSHI 5
8  PUSH  5
9  INC
```

23

原始プログラムと目的プログラム

原始プログラム　コンパイラ　目的プログラム

Hello.java → javac → Hello.class

```
public class Hello {
    public static void main (String args[]) {
        System.out.print(“Hello! World!¥n”)
    }
}
```

人間が読み書き可能

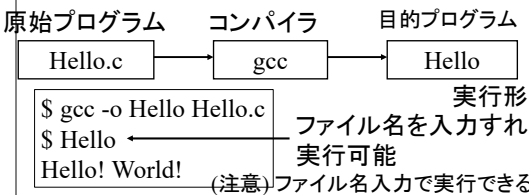
```
ﾊ・ｺﾓ????
??          ???
?? ? ??<init>?()V?Code?
LineNumberTable?main?([Ljava/lang/String;)V?
SourceFile?
Hello.java? ? ????
Hello! World! ????Hello?java/lang/Object
?java/lang/System?out?Ljava/io/PrintStream;
?java/io/PrintStream? println
?(Ljava/lang/String;)V?!????????? ??
?          ????????*+??????
????????   ?
???        ???%?????  ｲﾌｶ??????
???
????????
????
```

人間には理解不能

24

実行形式プログラム (executable program)

- 実行形式プログラム(executable program)
  - 実行可能なプログラム
  - 機械語で記述
  - 高水準言語からコンパイラが変換




```
$ gcc -o Hello Hello.c
$ Hello
Hello! World!
```

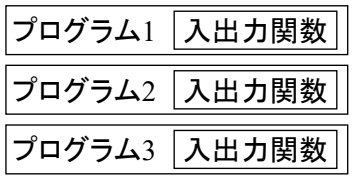
実行形式ファイル名を入力すれば実行可能

(注意)ファイル名入力で実行できるもの全てが実行形式プログラムではない

25

ライブラリ(library)

- 多くのプログラムに共通して使われる機能
  - 入出力関数, 数学関数(三角, 指数対数等)等



個別に作るのは無駄

⇒予め作成しておけばいい

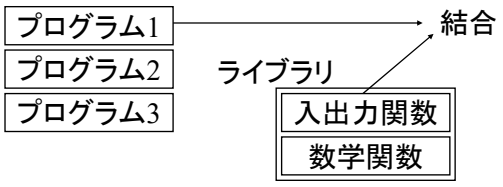
26

ライブラリ(library)

- 多くのプログラムに共通して使われる機能
  = プログラムごとに作成するのは無駄

⇩

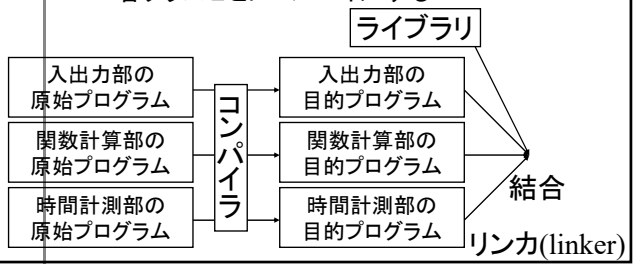
ライブラリ(library)を用いる



27

分割コンパイル (separate compile)

- 分割コンパイル(separate compile)
  - 原始プログラムをクラス、メソッドごとに分割
  - 各クラスごとにコンパイルする

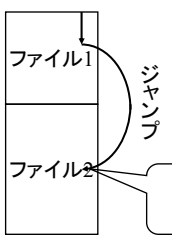


28

分割コンパイルの問題点

複数のファイルを別々にコンパイル

⇒他のファイルのサイズ、番地が分からない



⇒番地を後から決定できるようにする

再配置可能プログラム (relocatable program)

飛び先の番地は？

29

再配置可能プログラム (relocatable program)

- 再配置可能プログラム
  - プログラム先頭を0番地として相対的に記述
  - 他のプログラムと結合時に番地を再計算

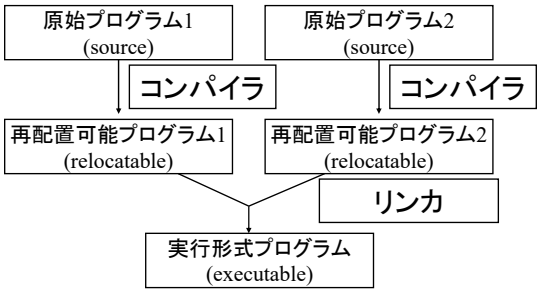
```
0 LOAD  1000
1 LOAD  L1:
2 ADD
3 BEQ       10
4 INPUT
5 STORE 1002
      :
```

先頭を0番地とした番地

他のプログラムの番地には仮のラベル

30

分割コンパイル



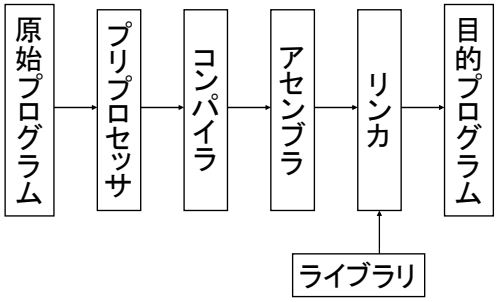
31

プリプロセッサ(preprocessor)

- プリプロセッサ
  - 目的プログラムが高水準言語のコンパイラ
  - コンパイラの前処理として行う

32

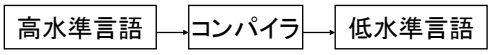
コンパイルシステム例



33

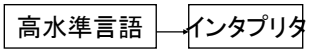
インタプリタ(interpreter)

- コンパイラ



実行

- インタプリタ(interpreter)



実行

高水準言語を解釈して処理

BASIC, perl, ruby 等

34

コンパイラとインタプリタ

- コンパイラ
  - 一旦コンパイルすれば高速で実行可能
    (インタプリタの数十～数百倍)
  - ⇒繰り返し実行するときに有効
- インタプリタ
  - コンパイルすることなく実行可能
  - ⇒1回だけ実行するときに有効
  - ⇒作成→実行を繰り返すときに有効

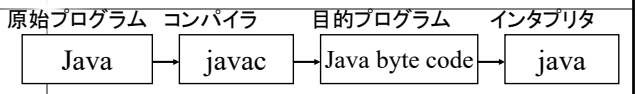
35

コンパイラとインタプリタ

	コンパイラ	インタプリタ
処理	低水準言語に変換	そのまま実行
プログラム作成+実行	コンパイルが必要	そのまま実行可能
実行速度	速	遅
処理系の多機種への移植	難	易
作成し易さ	難	易

36

Javaの場合



Java byte code は
中間コード(intermidiate code)
実行形式ではない

```
$ javac Hello.java
$ java Hello
Hello! World!
```

← インタプリタ “java” を使用

コンパイラ＋インタプリタ

37

コンパイラの記述言語

- コンパイラ
  - 原始プログラム(source program)を目的プログラム(object program)に変換(翻訳)するプログラム

コンパイラもプログラム

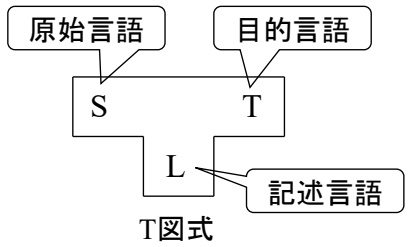
その言語は？

高水準言語？ 低水準言語？

38

T図式

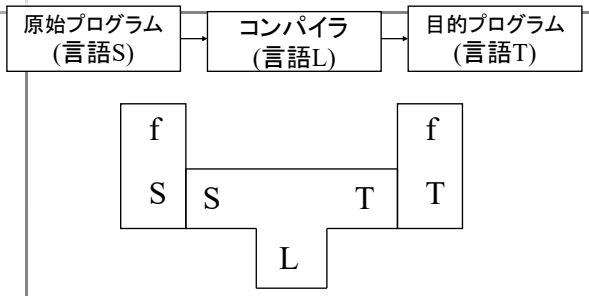
- 原始言語 S を目的言語 T に変換する言語 L で記述されたコンパイラ



T図式

39

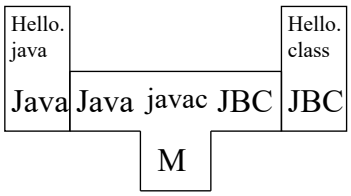
T図式



40

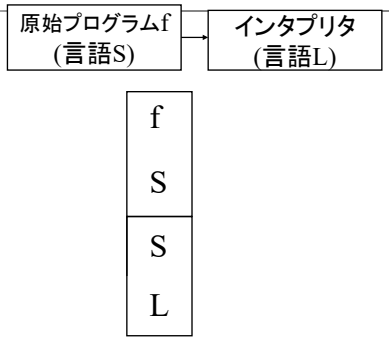
T図式

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



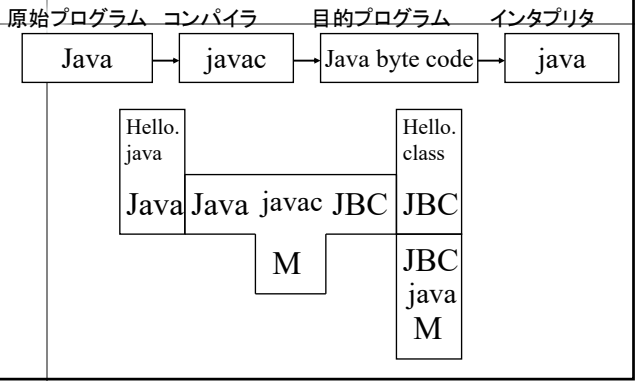
41

T図式(インタプリタの場合)



42

T図式(Javaの場合)



43

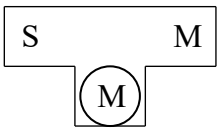
コンパイラの作成

機械語 M のみ実行可能

計算機 M

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

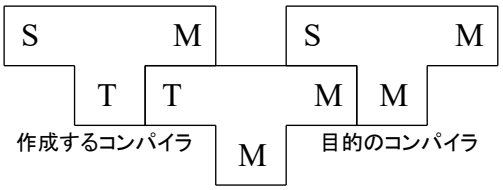
44

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

45

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

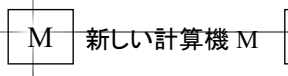
- ✓ ではTのコンパイラはどうやって作る？
- ✓ M上で動く既存の高水準言語コンパイラが無い場合は？

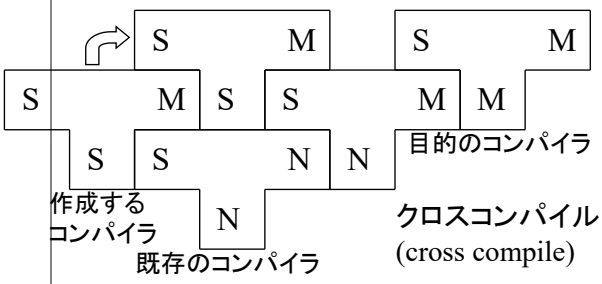
⇩

別の計算機 N 上で動くコンパイラを利用

46

コンパイラの作成



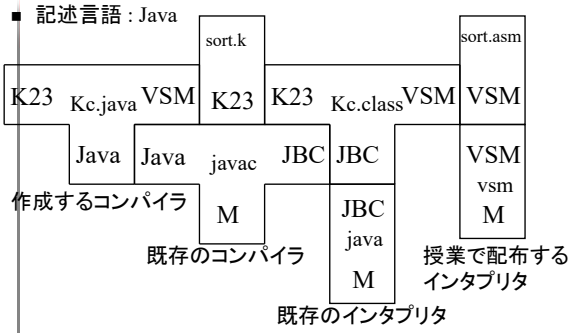


クロスコンパイル
(cross compile)

47

情報システムプロジェクトIの場合

- 原始言語 : K23言語(C風言語)
- 目的言語 : VSM(Virtual Stack Machine)アセンブラ言語
- 記述言語 : Java



48

コンパイラの構造

- 字句解析系
- 構文解析系
- 制約検査系
- 中間コード生成系
- 最適化系
- 目的コード生成系

49

字句解析系
(lexical analyzer, scanner)

- 字句解析系
  - 空白、コメントを読み飛ばす
  - 単語(token)に区切る

```
if (ans >= 123 ) /* ansの値で分岐 */ (改行)
  (空白)output ('1') ;
```

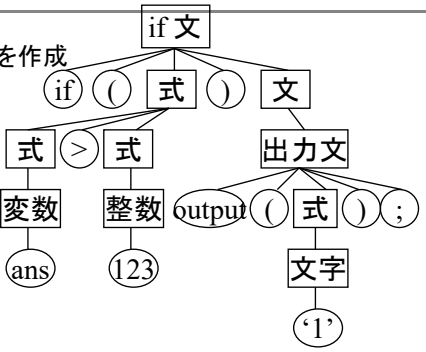
```
予約語 "if"
左括弧 "("
変数   "ans"
不等号 ">="
整数   "123"
右括弧 ")"
予約語 "output"
   :
```

50

構文解析系
(syntax analizer, parser)

- 構文解析系
  - 構文解析木を作成

```
if (ans > 123 )
  output ('1') ;
```



51

制約検査系
(constraint checker)

- 制約検査系
  - 変数の未定義・二重定義・型の不一致などを検査

```
int i, j;
x = 0;
i[10] = 5;
0 = 10;
```

変数 x は未定義

変数 i は配列ではない

代入の左辺が変数ではない

52

中間コード生成系
(semantics analyzer, intermediate code generator)

- 意味解析系
  - 単純な命令の列(中間コード)を生成する
- 中間コード(intermediate code)
  - ハードウェアには依存しない
  - 3番地コード(three address code)が多用される

```
A := B op C
```

```
if (a>0) b:=2*a+b;
```

⇨

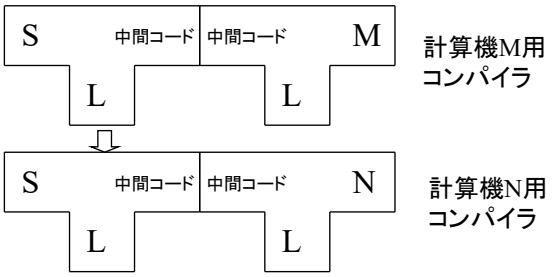
```
  if (a≦0) goto L:
  t := 2 * a
  b := t + b
L:
```

53

中間コードを用いる利点

中間コードはハードに依存しない

⇒異なるハードで共通で使用可能



54

最適化系
(optimizer)

- 最適化系
  - 中間コードを改良
    - 実行速度を速く
    - メモリ使用領域を小さく

```
if (a≦0) goto L:
t := 2 * a
b := t + b
L:
```

⇨

```
if (a≦0) goto L:
t := a + a
b := t + b
L:
```

掛け算より足し算の方が速い

55

目的コード生成系
(object code generator)

- 目的コード生成系
  - 変数の記憶位置決定
  - レジスタの割付

```
if (a≦0) goto L:
t := a + a
b := t + b
L:
```

⇨

```
LD   GR1, a
LEA GR1, 0, GR1
JMI  L:
JZE  L:
ADD GR1, a
ADD GR1, b
ST   GR1, b
L:
```

56

表管理
(table manegement, bookkeeping)

- 表管理
  - 原始プログラム中の変数,関数等の名前,型情報等を記憶

```
int i, j;
char ch;
int a[10];
```

変数名	型	サイズ	番地
i	int	1	0
j	int	1	1
ch	char	1	2
a	int[]	10	3〜12

57

誤り処理(error handling)

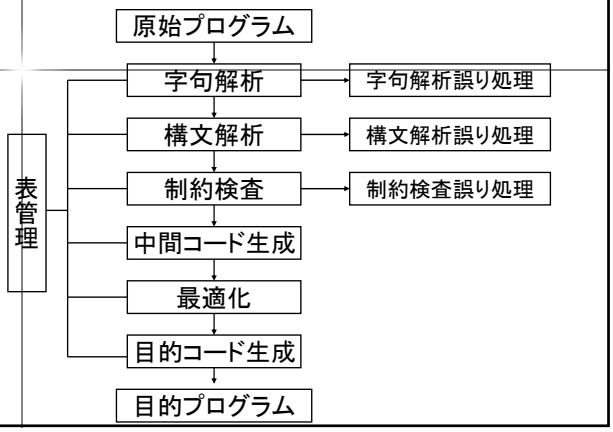
- 誤り処理
  - 原始プログラムが言語の制約を満たしていない場合にエラーメッセージを出す

```
int あ, い, う;
if () output (1);
5 = a;
```

1行目で字句解析エラー:
変数名に日本語は使えません
2行目で構文解析エラー:
if文の条件式がありません
3行目で制約検査エラー:
代入の左辺が変数ではありません

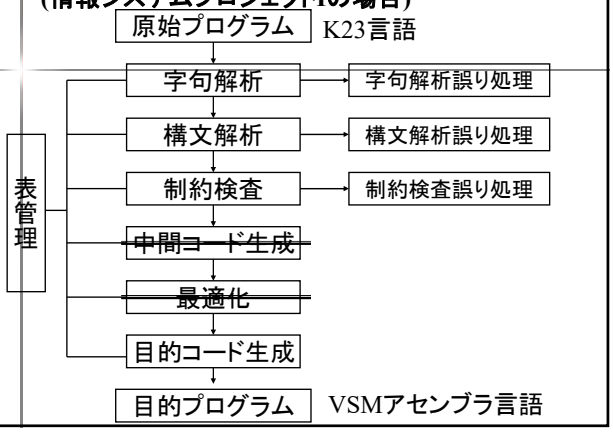
58

コンパイラの構成



59

コンパイラの構成
(情報システムプロジェクトIの場合)



60

処理の流れ
(情報システムプロジェクトIの場合)

output (ab);

↓

字句解析系　マイクロ構文の文法に従い解析

output (変数名) ;

↓

構文解析系　マクロ構文の文法に従い解析

<output_statement> ::= “output” “(” <exp> “)” “;”

↓

コード生成系　VSMアセンブラの文法に従い生成

1. PUSH ab　　2. OUTPUT

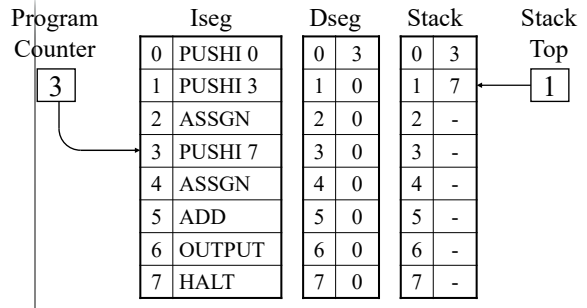
61

スタックマシン (stack machine)

- スタックマシン
  - Instruction Iseg[] : アセンブラプログラムを格納
  - int Dseg[] : 実行中の変数値を格納
  - int Stack[] : スタック(作業場所)
  - int Program Counter : 現在の Iseg の実行位置
  - int Stack Top : 現在のスタックの操作位置

62

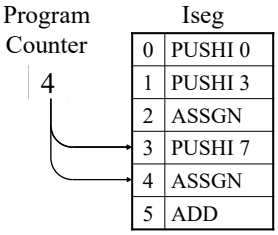
スタックマシン (stack machine)



63

Iseg と Program Counter

- VSM の動作
  1. Iseg の PC 番地の命令を実行
  2. PC := PC+1 or ジャンプ命令で指定した先



64

Dseg

- 実行中の変数値を格納

```
int i, j, x=2, y=3;
char c = ‘a’;
int a[5];
```

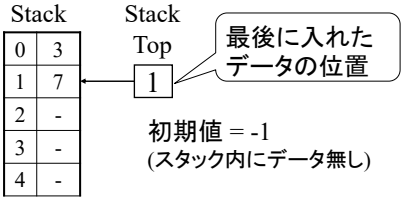
Dseg

0	0	i
1	0	j
2	2	x
3	3	y
4	‘a’	c
5	0	a[0]
6	0	a[1]
7	0	a[2]
8	0	a[3]
9	0	a[4]

65

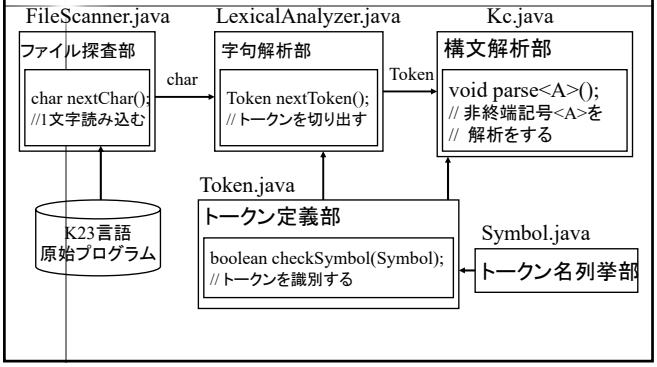
Stack

- Stack
  - 作業場所，処理中のデータの一時置き場
  - Last In First Out



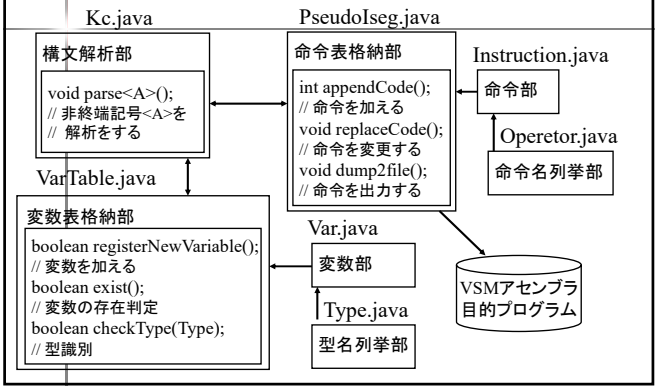
66

プログラムの構造
(字句解析系・構文解析系)



67

プログラムの構造(コード生成系)



68

宿題

- 「言語理論とオートマトン」の復習をする
  - 有限オートマトン
  - 正則表現
  - 正則文法
  - BNF記法, EBNF記法

69

課題テスト

- 毎週 GoogleClassroom上で課題テストを行う
  - 授業後～翌週の授業開始まで
- GoogleClassroomで

  コンパイラ

  ⇒授業

  ⇒その回の課題

  と辿る

70

71

72

73

74

質問

石水隆
E館　3F E-331

オフィスアワー
金曜3,4時限

takasi-i@info.kindai.ac.jp

75