

# コンパイラ

## 第14回 コンパイラコンパイラ

<http://www.info.kindai.ac.jp/compiler>

E館3階E-331 内線5459

[takasi-i@info.kindai.ac.jp](mailto:takasi-i@info.kindai.ac.jp)

# 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



# コンパイラの特徴

- 作成は規則的
  - 構文規則に従って規則的に作られる
- 作成作業が膨大 (特にLR構文解析)
  - LL構文解析：非終端記号ごとに解析が必要
  - LR構文解析：状態から解析表を作成

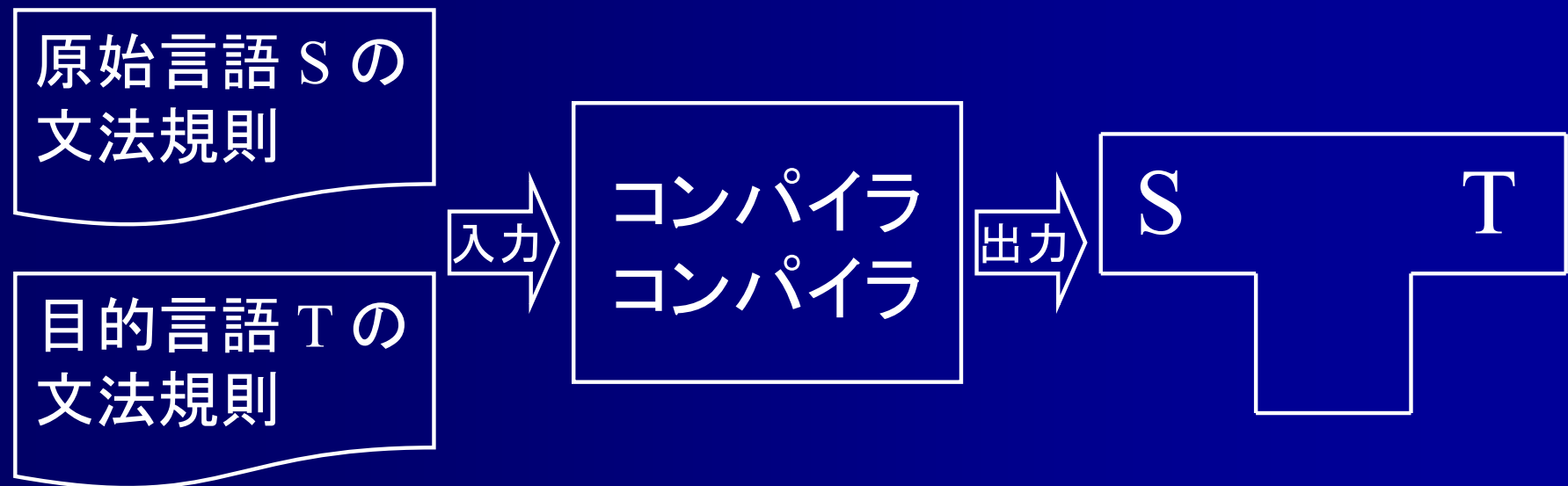
人間が作成するよりも  
計算機に任せてはどうか？

⇒ コンパイラコンパイラを利用

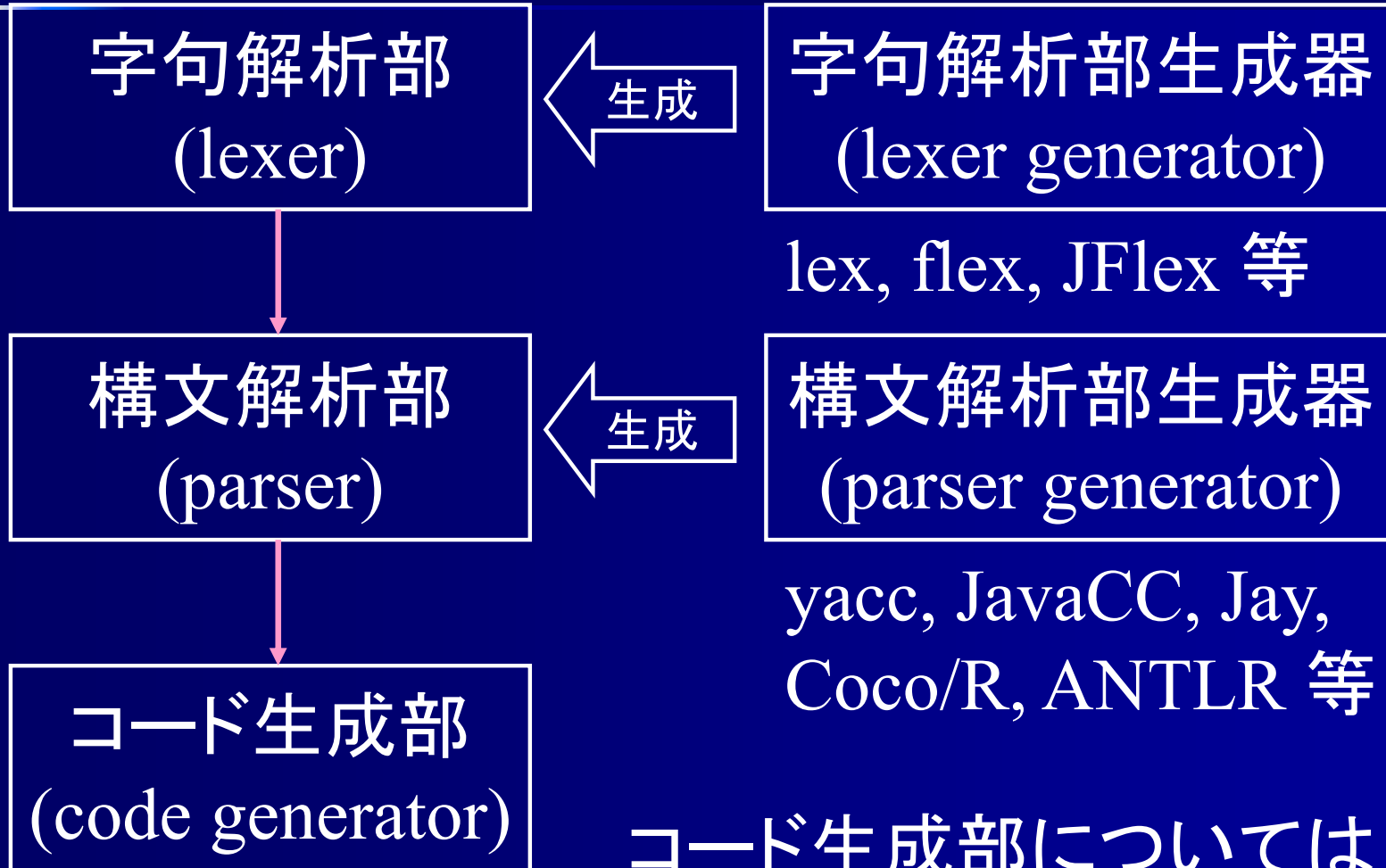
# コンパイラコンパイラ

## ■ コンパイラコンパイラ

- コンパイラを自動生成するためのプログラム



# 生成器 (generator)



字句解析部生成器  
(lexer generator)

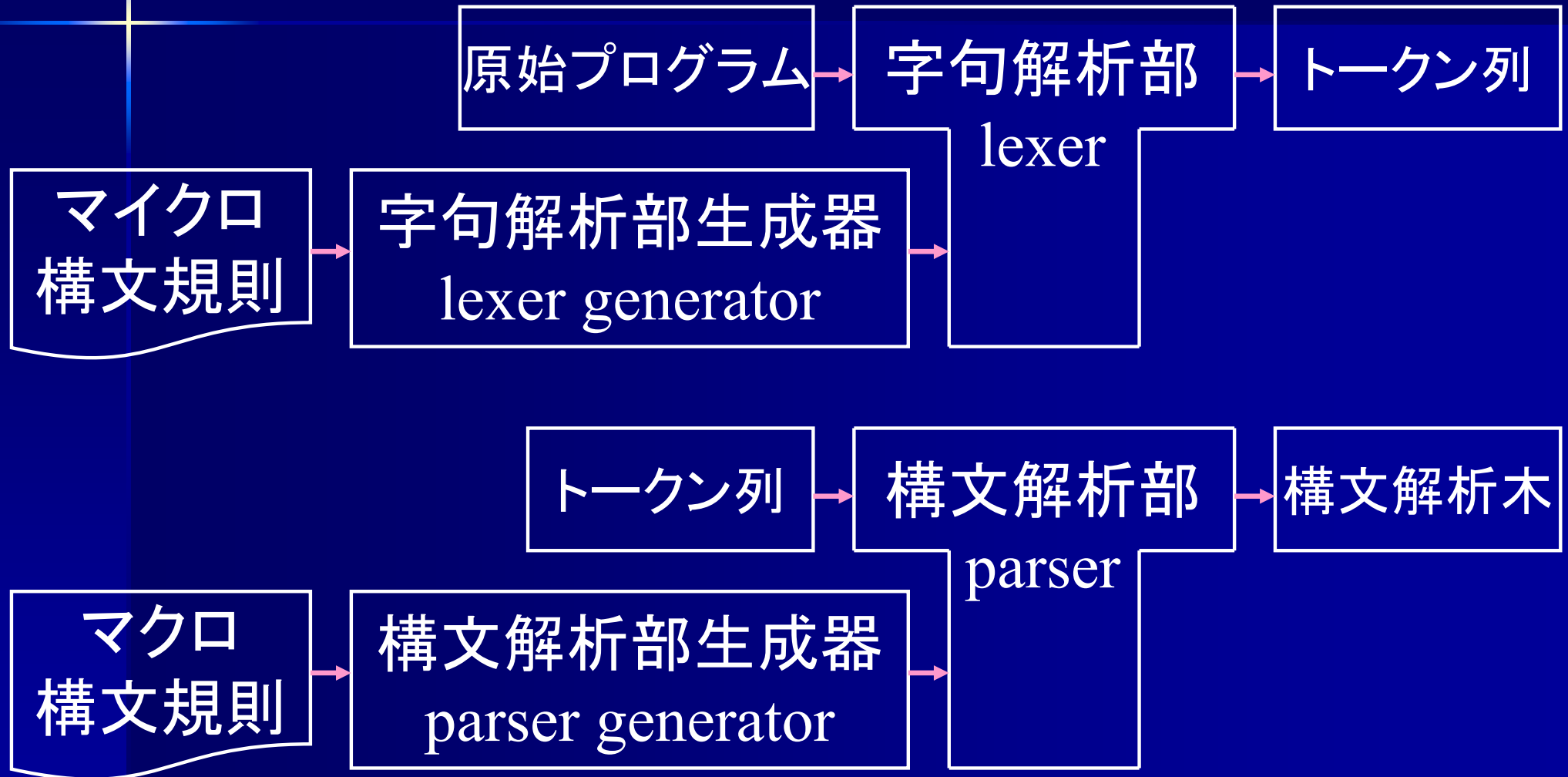
lex, flex, JFlex 等

構文解析部生成器  
(parser generator)

yacc, JavaCC, Jay,  
Coco/R, ANTLR 等

コード生成部については  
今のところ有望な生成器は無い

# 生成器



# 代表的なコンパイラコンパイラ

コンパイラ コンパイラ	生成するプログラムの 記述言語	解析法
lex + yacc	C言語	LALR(1)
flex + Bison	C言語	LALR(1)
JavaCC	Java	LL( $k$ )
JFlex + Jay	Java	LALR(1)
JFlex + CUP	Java	LALR(1)
Coco/R	Java, C#	LL( $k$ )
JS/CC	Java script	LALR(1)
ANTLR	C, C#, Java, ruby 等	LL(*)

# lex と yacc

## ■ lex

- 字句解析器(記述言語C)を生成
- 後継 : flex

## ■ yacc (yet another compiler compiler)

- 構文解析器(記述言語C)を生成
- LALR(1) 構文解析
- 後継 : Bison, kmyacc 等
  - ※ lex と yacc は基本的にセットで使用する
  - ※ Linux, MacOS では基本ソフトとしてインストール済



# lex と yacc

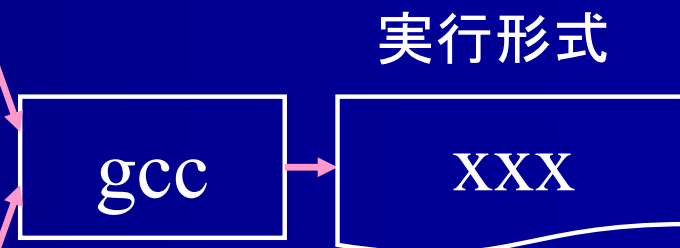
マイクロ構文定義



マクロ構文定義



#include



\$ xxx inputfile

# JFlex と Jay

## ■ JFlex

- Flex の Java 版
- 字句解析器(記述言語Java)を生成
- URL : <http://www.jflex.de/>

## ■ Jay

- yacc の Java 版
- 構文解析器(記述言語Java)を生成
- LALR(1) 構文解析
- URL : <http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html>

※ JFlex と Jay は基本的にセットで使用する

# JFlex と Jay

マイクロ構文定義



マクロ構文定義



```
$ java zzz inputfile
```

# JavaCC

## ■ JavaCC

- 字句・構文解析器(記述言語Java)を生成
- LL( $k$ ) 構文解析
- URL : <https://javacc.github.io/javacc/>

マイクロ構文定義  
マクロ構文定義



```
$ java yyy inputfile
```

# JavaCC で省略できる作業

## ■ 字句解析系

- マイクロ構文から有限オートマトンを求める
- `nextToken()` メソッドの作成

## ■ 構文解析系

- マクロ構文が LL(1) 文法か否かの判定
- マクロ構文から First 集合を求める
- `nextToken()` メソッドの呼び出し
- トークンの一致判定

# JavaCC で省略できない作業

- 構文解析系
  - 左再帰性の除去
  - 左括り出し
- コード生成系
  - 全て
- 最適化系
  - 全て

# JavaCC のインストール(Mac)

- MacPorts を使うのが簡単

- Mac OS X のパッケージ管理ツール

- インストール後は port コマンドで様々なパッケージをインストール可能

- URL:<http://www.macports.org/>

# MacPorts のインストール

1. <http://www.macports.org/> から pkg ファイルをダウンロード
2. pkg ファイルをクリックしてインストール
3. `/opt/local/etc/macports/` に移動
4. エディタで `sources.conf` を編集
  - i. `rsync:` で始まる行をコメントアウト
  - ii. その下に以下の一行を加える  
`http://www.macports.org/files/ports.tar.gz [default]`

(※) 3., 4. はプロキシ内からインストールする場合



Available Downloads

MacPorts FAQ

Report a Bug

Bug reporting Guidelines

Git Repository

MacPorts Team

Becoming a Member

## The MacPorts Project Official Homepage

The MacPorts Project is an open-source community initiative to design an easy-to-use system for compiling, installing, and upgrading either command-line, X11 or Aqua based open-source software on the [Mac operating system](#). To that end we provide the command-line driven MacPorts software package under a [3-Clause BSD License](#), and through it easy access to thousands of ports that [greatly simplify](#) the task of [compiling and installing](#) open-source software on your Mac.

We provide a single software tree that attempts to track the latest release of every software title (port) we distribute, without splitting them into "stable" Vs. "unstable" branches, targeting mainly macOS Sierra v10.12 and later (including macOS Catalina v10.15). There are [thousands of ports](#) in our tree, distributed among different categories, and more are being added on a regular basis.

### Getting started

For information on installing MacPorts please see the [installation](#) section of this site and explore the myriad of download options we provide and our base system requirements.

If you run into any problems installing and/or using MacPorts we also have many options to help you, depending on how you wish to get [get in touch with us](#). Other important help resources are our online documentation, A.K.A [The MacPorts Guide](#), and our Trac [Wiki server & bug tracker](#).

Latest MacPorts [release](#): 2.6.3

## Available Downloads

[MacPorts FAQ](#)[Report a Bug](#)[Bug reporting Guidelines](#)[Git Repository](#)[MacPorts Team](#)[Becoming a Member](#)

## Quickstart

1. Install [Xcode and the Xcode Command Line Tools](#)
2. Agree to Xcode license in Terminal: `sudo xcodebuild -license`
3. Install MacPorts for your version of the Mac operating system:

- [macOS Catalina v10.15](#)
- [macOS Mojave v10.14](#)
- [macOS High Sierra v10.13](#)
- [macOS Sierra v10.12](#)
- [Older OS? See here.](#)

## Installing MacPorts

MacPorts version 2.6.3 is available. If you are upgrading to a new major version, please read the [release notes](#).

- “pkg” installers for [Catalina](#), [Mojave](#), [High Sierra](#), [Sierra](#), [Mavericks](#), [Mountain Lion](#), [Lion](#), [Snow Leopard](#), [Leopard](#) and [Tiger](#) are also available.
- In [source form](#) as either a [tar.bz2](#) package or a [tar.gz](#) one for manual compilation, if you

OS のバージョンに応じた pkg ファイルをダウンロード



Finder で開く



MacPorts-2.1.3-10.6-SnowLeopard.pk





## ようこそ MacPorts インストーラへ

- はじめに
- 大切な情報
- 使用許諾契約
- インストール先の選択
- インストールの種類
- インストール
- 概要

### Welcome to the MacPorts for Mac OS X Installer

MacPorts provides the infrastructure that allows easy installation and management of freely available software on Mac OS X 10.6 systems.

<http://www.macports.org/>

This installer guides you through the steps necessary to install MacPorts 2.1.3 for Mac OS X. To get started, click Continue.

This package was made with:

**Mac Ports**

<http://www.macports.org/>

戻る

続ける



```
$ su
# cd /opt/local/etc/macports
# /usr/bin/emacs1 sources.conf
```

プロキシ内で  
ダウンロード  
する場合は  
書き換える

```
sources.conf
# configuration file
# 9 21:18:18Z jmr@macports.org $
# , insert a "file://" entry following
# your local ports directory:
# /opt/local/etc/macports/ports
# should always be tagged [default]
# for proper functionality of various resources (port groups, mirror
# sites, etc). If you switch it from the rsync:// URL, be sure to keep
# it tagged [default].
# To prevent a source from synchronizing when `port sync` is used,
# append [nosync] at the end as shown in this example:
# Example: file:///Users/landonf/misc/MacPorts/ports [nosync]
# NOTE: The port command parses source URLs in order and installs the
# first occurrence when a port appears in multiple repositories.
# So keep "file://" URLs above other URL types.
# To get the ports tree from the master MacPorts server in California, USA use:
# rsync://rsync.macports.org/release/ports/
# To get it from the mirror in Trondheim, Norway use:
# rsync://trd.no.rsync.macports.org/release/ports/
# A current list of mirrors is available at https://trac.macports.org/wiki/Mirrors
# If an rsync URL points to a .tar file, a signed .rmd160 must exist next to
# it on the server and will be used to verify its integrity.
#rsync://rsync.macports.org/release/tarballs/ports.tar [default]
http://www.macports.org/files/ports.tar.gz [default]
: sources.conf All 114 (Conf[Color]) 5:08PM
```

```
#rsync://rsync.macports.org/release/tarballs/ports.tar [default]
http://www.macports.org/files/ports.tar.gz [default]
```

# JavaCC のインストール

1. port sync コマンドでパッケージリスト取得
2. port install コマンドでインストール

```
$ su
# cd /opt/local/etc/macports
# /usr/bin/emacs1 sources.conf
# port -d sync
# port install javacc
# exit
$ javacc
```

```
# port -d sync
```

```
info-no-MacBook-Air-42:~ info$ su
Password:
sh-3.2# cd /opt/local/etc/macports/
sh-3.2# emacs sources.conf
sh-3.2# port -d sync
---> Updating the ports tree
Synchronizing local ports tree from http://www.macports.org/files/port
% Total    % Received % Xferd  Average Speed   Time    Time     Time
   Dload  Upload    Total   Spent    Left
49 21.5M    49 10.7M    0
```

```
# port install javacc
```

```
info-no-MacBook-Air-42:~ info$ bash
sh-3.2# port install javacc
Warning: The Command Line Tools for Xcode don't appear to be installed; most por
ts will likely fail to build.
Warning: See http://guide.macports.org/chunked/installing.xcode.html for more in
formation.
---> Fetching archive for javacc
---> Attempting to fetch javacc-5.0_2.darwin_11.x86_64.tbz2 from http://package
s.macports.org/javacc
---> Attempting to fetch javacc-5.0_2.darwin_11.x86_64.tbz2.rmd160 from http://
packages.macports.org/javacc
---> Installing javacc @5.0_2
---> Activating javacc @5.0_2
---> Cleaning javacc
---> Updating database of binaries: 100.0%
---> Scanning binaries for linking errors
sh-3.2# exit
exit
info-no-MacBook-Air-42:~ info$
```

# JavaCC のインストールの確認

- ls コマンドで javacc があるか確認

```
$ ls -l /opt/local/bin/javacc  
-rwxr-xr-x 1 root admin 163 8 2 2013 /opt/local/bin/javacc
```

- javacc コマンドで Usage メッセージを確認

```
$ javacc  
Java Compiler Compiler Version 5.0 (Parser Generator)  
Usage: javacc option-settings input file  
:
```



# JavaCCの使い方

## ■ JavaCCの使い方

- jj ファイルにマイクロ構文定義, マクロ構文定義を記述する

マイクロ構文定義  
マクロ構文定義



info.kindai.ac.jp  
<https://www.info.kindai.ac.jp/compiler/>

このページは2022年度の「コンパイラ」の公式ホームページです。ここに講義録、課題、レポートの提出方法他の情報を掲載します。

## 連絡

- 第14回について

第14回では [JavaCC](#) について説明します。必須ではありませんが、理解を深めるために、[こちらのページ](#)を参考に JavaCC のインストールを行い、[こちらのページ](#)にも目を通して置くことをお勧めします。また、javacc のサンプルプログラム集 [javacc.zip](#) もダウンロードしておくといいいでしょう。

- [第14回 JavaCC実習について](#)
- [javacc.zip](#)
- [Mac へのインストール, Linux へのインストール](#)
- [JavaCCの使い方](#)

- 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翌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扱いにします。

オンライン授業では、当日 [GoogleClassroom](#) から出席カードが提出が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。

# サンプルプログラム

javacc/

sample0.k, sample1.k, sampleLL2.k :

k言語の例プログラム

sample0.asm, sample1.asm, sampleLL2.asm :

\*.kのコンパイル後のアセンブラコード

vsm : vsmインタプリタ

sampleExp.txt : calc.jj用の数式例

kc/ : 構文解析器生成用ファイル一式

parser.jj, parserCode.jj, parserCodeLL2.jj : .jjファイル

Parser.java, ParserCode.java, ParserCodeLL2.java :

生成されたJavaプログラム

Instruction.java, Operator.java, PseudoIseg.java,

Type.java, Var.java, VatTable.java

calc/ : 計算機生成用ファイル一式

calc.jj : .jjファイル

Cals.java : 生成されたJavaプログラム

# jj ファイルのコンパイル

```
$ javacc jj ファイル名
```

```
$ javac 生成された Java ファイル名
```

```
$ java Java クラス名
```

```
$ cd ~/javacc/kc
```

```
$ javacc parse.jj
```

```
$ cd ..
```

```
$ javac kc/Parse.java
```

```
$ java kc.Parse sample0.k
```

# jj ファイルのコンパイル例

sample0.k

```
main () {  
    int i;  
    i = inputint;  
    if (i) outputint ( i*2 );  
}
```

```
$ cd ~javacc/kc  
$ javacc parserCode.jj  
$ cd ..  
$ javac kc/ParserCode.java  
$ java kc.ParserCode sample0.k
```

OpCode.asm

```
PUSHI      0  
INPUT  
ASSGN  
REMOVE  
PUSH       0  
BEQ        10  
PUSH       0  
PUSHI      2  
MUL  
OUTPUT  
HALT
```

# JavaCC により 生成される Java プログラム

生成されるプログラム	役割
Xxx.java	メインクラス(構文解析部)
XxxConstants.java	トークンID, 定数等を定義
XxxTokenManager.java	トークン管理(字句解析部)
ParseException.java	構文解析エラー時の処理
Token.java	トークン型を定義
TokenMgrError.java	字句解析エラー時の処理

# jj ファイルの記述

- 構文解析クラス記述部
  - 生成する構文解析クラスのメソッドを定義  
(main メソッドを含む)
- マイクロ構文定義部
  - トークン, 空白を定義
- マクロ構文定義部
  - 各非終端記号の生成規則を定義

# サンプル jj ファイル

- parser.jj, parserCode.jj
  - 以下のマクロ構文を定義

```
<Main> ::= "main" "(" ")" "{" { <Decl> | <State> } "}" EOF
<Decl> ::= "int" <NAME> { "," <NAME> } ";"
<Name> ::= NAME
<State> ::= <If> | <Output> | <Assgn> | "{" { <State> } "}"
<If> ::= "if" "(" <Exp> ")" <State>
<Output> ::= "outputint" "(" <Exp> ")" ";"
<Assgn> ::= Name "=" <Exp> ";"
<Exp> ::= <Term> { ( "+" | "-" ) <Term> }
<Term> ::= <Factor> { ( "*" | "/" ) <Factor> }
<Factor> ::= NAME | INTEGER | "inputint"
```



# 構文解析クラス記述部

```
javacc_options // オプション指定

PARSER_BEGIN ( <IDENTIFIER> ) // 生成するクラス
java_compilation_unit // 生成するクラスに置くメソッド
PARSER_END ( <IDENTIFIER> )

( production ) * // マイクロ構文, マクロ構文の定義
```

これを JavaCC でコンパイルすると  
<IDENTIFIER>.java が生成される

# parser.jj のクラス記述部

生成されるファイルは  
Parser.java

```
PARSER_BEGIN (Parser)
import java.util.*;
import java.io.*;
public class Parser{
    public static void main (String[] args) { // main メソッド
        try {
            Parser parser = new Parser (new FileReader (args[0]));
                                                    // 構文解析器生成
            parser.Main();                          // 構文解析メソッド呼出
        } catch (Exception err_mes) {
            System.out.println (err_mes);          // エラー出力
        }
    }
}
PARSER_END (Parser)
```

PARSER\_BEGIN () から  
PARSER\_END () までが  
そのまま生成ファイルに出力される

# Parser.java の冒頭部

```
/* Generated By:JavaCC: Do not edit this line. Parser.java */
import java.util.*;
import java.io.*;
public class Parser {
    public static void main (String[] args) { // main メソッド
        try {
            Parser parser = new Parser (new FileReader (args[0]));
                                                    // 構文解析器生成
            parser.State();                          // 構文解析メソッド呼出
        } catch (Exception err_mes) {
            System.out.println (err_mes);           // エラー出力
        }
    }
}
```

# マイクロ構文の記述

## 空白の定義

```
SKIP : {  
    <パターン>  
}
```

```
SKIP : {  
    <“ ” | “\n” | “\t” | “\r” >  
}
```

## トークンの定義

```
TOKEN : {  
    <トークン名:パターン>  
}
```

```
TOKEN : {  
    <ASSGN: “=” >  
    | <ADD: “+” >  
    | <SUB: “-” >  
    | <MUL: “*” >  
    | <DIV: “/” >  
}
```

パターンは正規表現で記述

# 表記例

INTEGER ::= '0' | Pdec {Dec}

0 回以上の繰り返し

TOKEN : { <INTEGER: "0" | ["1"-“9”] ([“0”-“9”])\*> }

NAME ::= Alpha { Alpha | Dec }

0~9 の数字

TOKEN : { <NAME: [“a”-“z”][“A”-“Z”]  
([“a”-“z”][“A”-“Z”][“0”-“9”])\*> }

LINECOMMENT ::= '/' '/' {任意の文字} (改行)

SKIP : { <“//” (~[“¥n”, “¥r”])\* [“¥n”, “¥r”]> }

表記法	意味	注記
“abc”	文字列 abc	
$\alpha \mid \beta \mid \gamma$	$\alpha$ または $\beta$ または $\gamma$	$\alpha\beta\gamma$ は文字列
[“a”]	a ([ ] は文字クラス)	[ ] 内は文字のみ
[“a”, “b”, “c”]	a または b または c	, は [ ] 内のみ
~[“a”, “b”, “c”]	a, b, c 以外の文字	
~[ ]	任意の文字	
[“a” - “z”]	小文字	- は [ ] 内のみ
[“0” - “9”]	数字	- は [ ] 内のみ
( $\alpha$ )?	$\alpha$ が 0 回または1回	() は省略不可
( $\alpha$ )*	$\alpha$ が 0 回以上	() は省略不可
( $\alpha$ )+	$\alpha$ が 1 回以上	() は省略不可
( $\alpha$ ) {n}	$\alpha$ が n 回	() は省略不可
( $\alpha$ ) {m, n}	$\alpha$ が m 回以上 n 回以下	() は省略不可

# トークンのマッチング

- 長さの異なる規則：長い方を優先 (最長一致)
- 同じ長さの規則：先に書かれた方を優先

```
TOKEN : {  
  <ASSGNADD: “+=”>  
  | <ADD: “+”>  
  | <INC: “++”>  
  | <IF: “if”>  
  | <WHILE: “while”>  
  | <NAME: ([“a”-“z”][“A”-“Z”]  
            [“0”-“9”][“a”-“z”][“A”-“Z”])*>  
}
```

どの順番で書いても  
++, += は + より優先

予約語は変数名より  
先に定義

# parser.jj のマイクロ構文の記述

```
SKIP : {  
  < “ ” | “\n” | “\t” | “\r” >  
  < “//” (~ [ “\n”, “\r” ] ) * [ “\n”, “\r” ] >  
}
```

0 回以上の繰り返し

```
TOKEN : {  
  < LPAREN: “(” > | < RPAREN: “)” >  
  | < ASSGN: “=” >  
  | < ADD: “+” > | < SUB: “-” > | < MUL: “*” > | < DIV: “/” >  
  | < INTEGER: ([ “0” - “9” ] ) + >  
  | < IF: “if” >  
  | < NAME: ([ “a” - “z” ] | [ “A” - “Z” ]  
    [ “0” - “9” ] | [ “a” - “z” ] | [ “A” - “Z” ] ) * >  
}
```

1 回以上の繰り返し



# コメント

ラインコメント // ... (改行)

```
SKIP : {  
  <“//” (~[“\n”, “\r”])* [“\n”, “\r”]>  
}
```

改行以外

改行

ブロックコメント /\* ... \*/

```
SKIP : {  
  <“/*” (~[])* “*/”>  
}
```

任意の1文字

最長一致なので  
“\*/” はここにマッチ

これで OK ?

# コメント

任意の1文字

```
SKIP : {  
    <“/*” (~[])* “*/” >  
}
```

最長一致の原則に従うと...

```
main () {  
    :  
    /* コメント1 */  
    :  
    /* コメント2 */  
    :  
    /* コメント3 */  
    :  
}
```

ここまでマッチしてしまう

# 状態付トークン・スキップ

## ■ 状態付トークン・スキップ

- 特定の状態でのみ解析されるトークン・スキップ

### 状態付トークンの定義

```
<状態1> TOKEN : {  
    <トークン名:パターン> : 状態2  
}
```

状態1 でトークンを読めば状態2 へ移行

<状態1>を省略した場合は <DEFAULT>  
状態2を省略した場合は状態はそのまま

# 状態付トークン・スキップ

```
<EN> TOKEN : {  
  <HELLO: “hello”>  
  <THANKYOU: “thankyou”>  
  <BYE: “bye”>  
}  
<EN> SKIP : {  
  <“jp”> : JP  
}
```

状態 EN で “jp” を  
読んだら状態 JP へ

```
<JP> TOKEN : {  
  <OHAYOU: “おはよう”>  
  <ARIGATOU: “ありがとう”>  
  <SAYOUNARA: “さようなら”>  
}  
<JP> SKIP : {  
  <“en”> : EN  
}
```

状態 JP で “en” を  
読んだら状態 EN へ

hello thankyou jp ありがとう さようなら en bye ⇒ 受理

hello jp おはよう thankyou en bye ⇒ thankyou で不受理

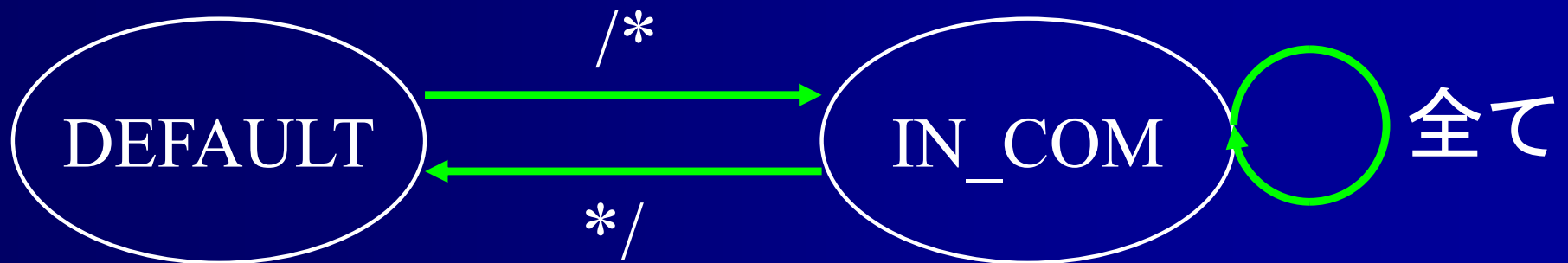
# 状態付スキップの表記例

BLOCKCOMMENT ::= ‘/’ ‘\*’ {任意の文字} ‘\*’ ‘/’

SKIP : { <“/\*”> : IN\_COM }

<IN\_COM> SKIP : { <~[]> | <“\*/”> : DEFAULT }

任意の1文字



# ブロックコメント

状態付きSKIPを使用

```
SKIP : { <“/*”> : IN_COM }  
<IN_COM> SKIP : { <~[]>  
                  | <“*/”> : DEFAULT }
```

状態付きSKIP無しでも一応可能

```
SKIP : { <“/*” (~[“*”])* (“*”)+  
        (~[“/”, “*”] (~[“*”])* (“*”)+ )*  
        “/” > }
```

# マクロ構文の記述 (コード生成無し)

非終端記号  $\langle A \rangle$  に対するマクロ構文の定義  
 $\langle A \rangle ::= \text{“token1” } \langle B \rangle \text{ “token2” } \langle C \rangle \text{ “token3”}$

```
void A() :  
{  
{  
  <token1> B() <token2> C() <token3>  
}  
}
```

マクロ構文に従いトークンを並べるだけ

# 表記例

$\langle \text{Stlist} \rangle ::= \{ \langle \text{St} \rangle \}$

```
void Stlist() :
```

```
{
```

```
{
```

```
  <LBRACE> ( St() ) * <RBRACE>
```

```
}
```

0 回以上の繰り返し

$\langle \text{Var} \rangle ::= \text{NAME} [ \langle \text{Exp} \rangle ]$

```
void Var() :
```

```
{
```

```
{
```

```
  <NAME> [ <LRRACKET> Exp() <RBLACKET> ]
```

```
}
```

0 回または 1 回



表記法	意味	注記
<ID>	終端記号 <ID>	字句解析部で定義
“abc”	終端記号 “abc”	字句解析部で定義
name()	非終端記号	
$\alpha \beta$	$\alpha\beta$ の接続	
$[\alpha]$	$\alpha$ が 0 回または1回	字句解析と異なる
$(\alpha)?$	$\alpha$ が 0 回または1回	() は省略不可
$(\alpha)^*$	$\alpha$ が 0 回以上	() は省略不可
$(\alpha)^+$	$\alpha$ が 1 回以上	() は省略不可
$(\alpha) \{n\}$	$\alpha$ が $n$ 回	() は省略不可
$(\alpha) \{m, n\}$	$\alpha$ が $m$ 回以上 $n$ 回以下	() は省略不可

# 構文解析系の作成

```
<If> ::= “if” “(” <Exp> “)” <State>
```

自力で書くと

```
void If() {  
    if (token == “if”) nextToken(); else SyntaxError();  
    if (token == “(”) nextToken(); else SyntaxError();  
    if (token ∈ First (<Exp>)) Exp(); else SyntaxError();  
    if (token == “)”) nextToken(); else SyntaxError();  
    if (token ∈ First (<State>)) State(); else SyntaxError();  
}
```

トークンの一致判定, nextToken()呼出, エラー処理等が必要

# 構文解析系の作成

```
<If> ::= “if” “(” <Exp> “)” <State>
```

JavaCC では

```
void If() :  
{  
{  
  <IF> <LPAREN> Exp() <RPAREN> State()  
}  
}
```

構文規則を並べるだけでいい

終端記号は文字列を直接書いても OK

```
“if” “(” Exp() “)” State()
```

# Parser.java の If()

構文エラーがあった場合は  
上位メソッドに例外を投げる

```
final public void If() throws ParseException {  
    jj_consume_token(IF);  
    jj_consume_token(LPAREN);  
    Exp();  
    jj_consume_token(RPAREN);  
    State();  
}
```

if (token == IF)  
 nextToken();  
else syntaxError();

# 構文解析系の作成

```
<State> ::= <If> | <While> | <Output> | <Assgn>
```

自力で書くと

```
void State() {  
    if (token ∈ First (<If>)) If();  
    else if (token ∈ First (<While>)) While();  
    else if (token ∈ First (<Output>)) Output();  
    else if (token ∈ First (<Assgn>)) Assgn();  
    else syntaxError();  
}
```

各非終端記号の First 集合を求めておく必要がある

# 構文解析系の作成

```
<State> ::= <If> | <While> | <Output> | <Assgn>
```

JavaCC では

```
void State() :  
{  
  {  
    If() | While() | Output() | Assgn()  
  }  
}
```

各非終端記号の First 集合を  
javacc が自動的に求めてくれる

# 構文解析系の作成

```
<Exp> ::= <Term> { ( “+” | “-” ) <Term> }
```

自力で書くと

```
void Exp() {  
    if (token ∈ First (<Term>)) Term(); else syntaxError();  
    while (token == “+” || token == “-”) {  
        nextToken;  
        if (token ∈ First (<Term>)) Term(); else syntaxError();  
    }  
}
```

# 構文解析系の作成

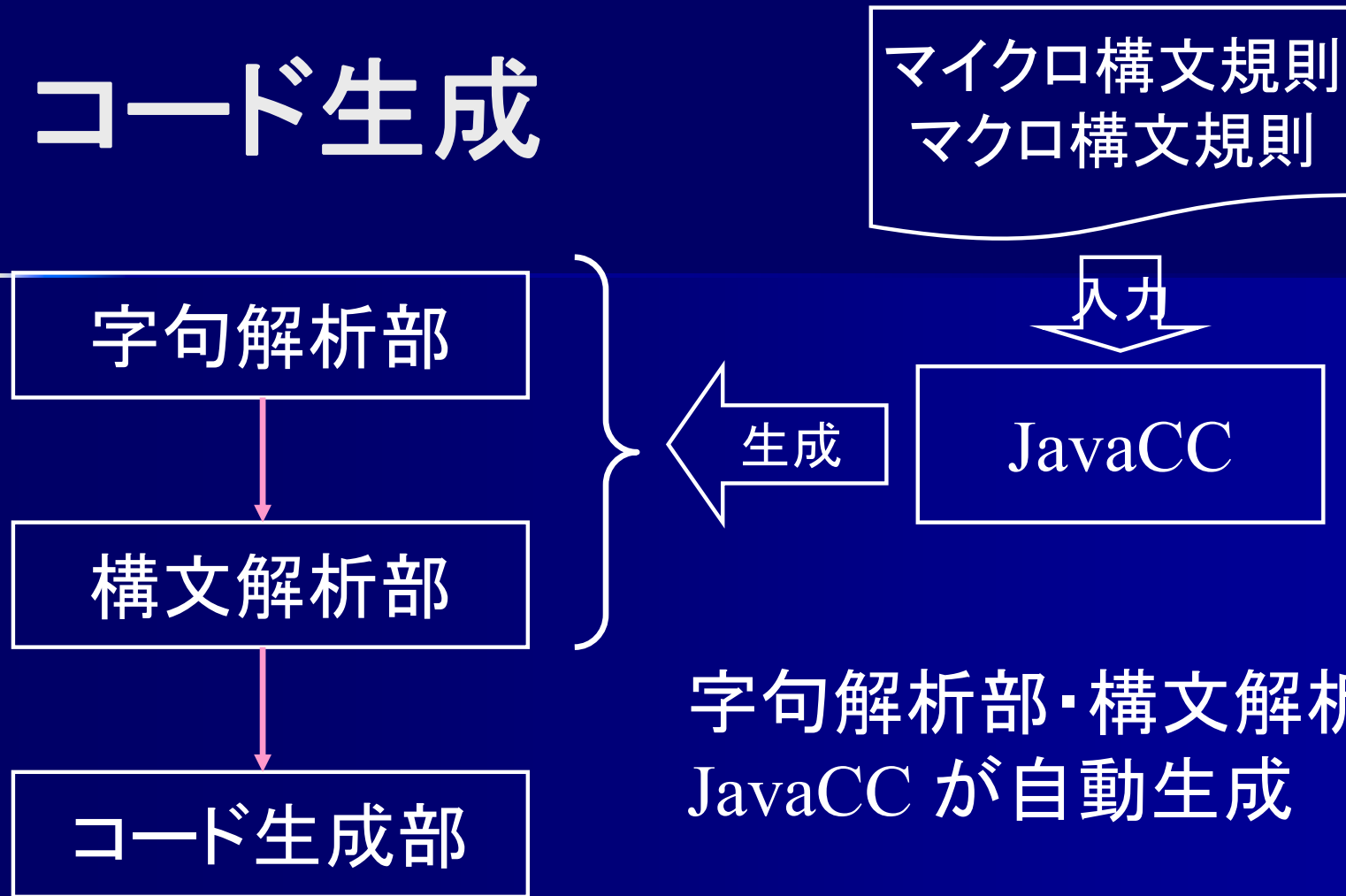
```
<Exp> ::= <Term> { ( “+” | “-” ) <Term> }
```

JavaCC では

```
void Exp() :  
{  
{  
    Term() ( ( “+” | “-” ) Term() ) *  
}  
}
```



# コード生成



字句解析部・構文解析部は  
JavaCC が自動生成

コード生成部は jj ファイルに  
手書きで埋め込む必要あり

# parserCode.jj のクラス記述部

```
public class ParserCode {
    static VarTable varTable;    // 変数表
    static PseudoIseg iseg;      // 疑似ISEG
    public static void main (String[] args) { // main メソッド
        try {
            ParserCode parser = new ParserCode (new FileReader (args[0]));
            parser.Main();        // 構文解析メソッド呼出
        } catch (Exception err_mes) {
            System.out.println (err_mes); // エラーメッセージ出力
        }
        if (args.length == 1) iseg.dump2file(); // アセンブラコード出力
        else iseg.dump2file (args[1]);
    }
}
```

# マクロ構文の記述 (コード生成あり)

非終端記号 <A> に対するマクロ構文の定義

<A> ::= “token1” <B> “token2” <C> “token3”

```
void A() :  
{ <A> に関する最初の処理を行う Java 命令 }  
{  
  <token1> {token1 を処理する Java 命令}  
  B()      {<B> を処理する Java 命令}  
  <token2> {token2 を処理する Java 命令}  
  C()      {<C> を処理する Java 命令}  
  <token3> {token3 を処理する Java 命令}  
}
```

# 構文解析系(コード無し)の作成

```
<Factor> ::= NAME | INTEGER
```

```
void Factor() :  
{  
{  
    <NAME>  
    | <INTEGER>  
}  
}
```

ここに生成するコードを埋め込む

# 構文解析系(コード有り)の作成

```
<Factor> ::= NAME | INTEGER
```

```
void Factor() :  
{ /* ここにメソッドの開始時に行う処理を記述 */ }  
{  
    <NAME> {  
        /* ここに NAME を読んだ場合の処理を記述 */  
    }  
    | <INTEGER> {  
        /* ここに INTEGER を読んだ場合の処理を記述 */  
    }  
}
```

# JavaCCのTokenクラス

Token		#トークン管理部
+ beginColumn	: int	#トークン先頭文字の列番号
+ beginLine	: int	#トークン先頭文字の行番号
+ endColumn	: int	#トークン末尾文字の列番号
+ endLine	: int	#トークン末尾文字の行番号
+ image	: String	#トークンの文字列表現
+ kind	: int	#トークンの種類
+ next	: Token	# 次のトークン
+ specialToken	: Token	# 次の特殊トークン
+ Token ()		# コンストラクタ
+ newToken (ofKind : int)	: static Token	# 新規トークン生成
+ toString ()	: String	# image を返す

# トークンデータの取り出し

Token型変数 token を用いて  
トークンデータを取り出す

```
token = <NAME>
```

読み込み中のトークンが<NAME>に  
マッチした場合代入される

トークンの文字列表現は token.image で参照

# トークンデータの取り出し

## 変数名の取り出し

```
token = <NAME> {  
    String name = token.image;  
}
```

token の文字列表現

## 整数値の取り出し

```
token = <INTEGER> {  
    int value = Integer.parseInt (token.image);  
}
```

文字列を整数値に変換



# 構文解析系(コードあり)の作成

```
void Factor() :  
{ int val, addr; String name; }  
{  
  token = <NAME> {  
    name = token.image;  
    addr = varTable.getAddress (name);  
    iseg.appendCode (Operator.PUSH, addr);  
  }  
  | token = <INTEGER> {  
    val = Integer.parseInt (token.image);  
    iseg.appendCode (Operator.PUSHI, val);  
  }  
}
```

読み込んだトークンは  
Token 型変数 token に  
代入可能

token の文字列表現

生成するプログラムに埋め込まれる

# 構文解析系(コードあり)の作成

```
<Exp> ::= <Term> { ( “+” | “-” ) <Term> }
```

```
void Exp() :  
{ /* ここにメソッドの開始時に行う処理を記述 */  
  Operator opcode ; // 演算子を記憶するための局所変数  
}  
{  
  Term() (  
    ( “+” { opcode = Operator.ADD; }  
    | “-” { opcode = Operator.SUB; } )  
    Term() { iseg.appendCode (opcode); }  
  )*  
}
```

```
<Assgn> ::= NAME “=” <Exp> “;”
```

```
void Assgn() :  
{ String name; int addr; }  
{  
    token = <NAME> {  
        name = token.image;  
        addr = varTable.getAddress (name);  
        iseg.appendCode (Operator.PUSHI, addr);  
    }  
    “=”  
    Exp() { iseg.appendCode (Operator.ASSGN); }  
    “;” { iseg.appendCode (Operator.REMOVE); }  
}
```

```
<If> ::= “if” “(” <Exp> “)” <State>
```

```
void If() :
```

```
{ int beqAddr, stLastAddr; // ジャンプ先のアドレス }
```

```
{
```

```
“if”
```

```
“(”
```

iseg 上のBEQ の  
アドレスを記憶

```
Exp() { beqAddr = iseg.appendCode (Operator.BEQ); }
```

```
“)”
```

```
State() {
```

```
    stLastAddr = iseg.getLastCodeAddress() ;
```

```
    iseg.replaceCode (beqAddr, stLastAddr+1);
```

```
}
```

```
}
```

アドレス付け変え

# 字句解析時のコード生成

字句解析時にもコードを埋め込み可能

```
TOKEN : { <トークン名: パターン> {コード} }
```

```
TOKEN_MGR_DECLS : { // 字句解析時用の変数宣言  
    static int paren_ctr = 0; // 括弧数カウント用  
}
```

```
TOKEN : {  
    <LPAREN: “(”> { ++paren_ctr; }  
    <RPAREN: “)”> { --paren_ctr;  
        if (paren_ctr < 0) syntaxError (“ ) が多過ぎです”); }  
}
```

# トークンの先読み

```
<F> ::= NAME “[” <Exp> “]” | NAME | INTEGER
```

```
void F() :
```

```
{
```

```
{
```

```
  <NAME> “[” <Exp> “]”
```

```
  | <NAME>
```

```
  | <INTEGER>
```

```
}
```

NAME を読んだ場合  
どちらか区別できない

1個のトークン先読みでは区別ができない  
⇒ LL(1) 文法ではない

# LOOKAHEAD オプション

- 先読みトークン数の指定
  - デフォルトでは  $1 \Rightarrow$  LL(1) 解析

全体を LL(2) 解析する場合は

```
options {  
    LOOKAHEAD = 2;  
}
```

ただし先読み数を多くすると処理が遅くなる

# LOOKAHEAD オプション

一部を LL(2) 解析する場合は

```
void F() :  
{  
  {  
    LOOKAHEAD(2) (<NAME> “[” <Exp> “]” )  
    | <NAME>  
    | <INTEGER>  
  }  
}
```

この部分のみ  
2個を先読み



# サンプル jj ファイル

## ■ parserCodeLL2.jj

– 以下のマクロ構文(LL2文法)を定義

```
<Main> ::= "main" "(" ")" "{" { <Decl> | <State> } "}" EOF
<Decl> ::= "int" <NAME> { "," <NAME> } ";"
<Name> ::= NAME "[" INTEGER "]" | NAME
<State> ::= <If> | <While> | <Output> | <Assgn> | "{" { <State> } "}"
<If> ::= "if" "(" <Exp> ")" <State>
<While> ::= "while" "(" <Exp> ")" <State>
<Output> ::= "outputint" "(" <Exp> ")" ";"
<Assgn> ::= Name [ "[" <Exp> "]" ] "=" <Exp> ";"
<Exp> ::= <Term> { ( "+" | "-" ) <Term> }
<Term> ::= <Factor> { ( "*" | "/" ) <Factor> }
<Factor> ::= NAME "[" <Exp> "]" | NAME | INTEGER | "inputint"
```

# DEBUG\_PARSER オプション

- true にすると構文解析のトレース出力

```
options {  
    DEBUG_PARSER = true;  
}
```

```
$ java kc.Parser sample0.k
```

```
Call: Main
```

```
Consumed token: <“main” at line 1 column 1>
```

```
Consumed token: <“(” at line 1 column 6>
```

```
Consumed token: <“)” at line 1 column 7>
```

```
Consumed token: <“{” at line 1 column 9>
```

```
Call: Decl
```

```
Consumed token: <“int” at line 2 column 9>
```

解析中の  
非終端記号

# JavaCC のオプション(一部)

オプション		デフォルト
LOOKAHEAD	先読みトークン数	1
STATIC	static メソッドを生成	true
UNICODE_INPUT	入力としてUnicode を扱う	false
IGNORE_CASE	大文字小文字を無視	false
OUTPUT_DIRECTORY	出力ディレクトリ	.
DEBUG_PARSER	構文解析をトレース出力	false
DEBUG_LOOKAHEAD	先読みをトレース出力	false
DEBUG_TOKEN_MANAGER	字句解析をトレース出力	false
BUILD_PARSER	構文解析部を生成	true
BUILD_TOKEN_MANAGER	字句解析部を生成	true
JDK_VERSION	JDK のバージョン	1.5

# JavaCC の活用

- JavaCC はコンパイラ作成以外にも活用可能

例：電卓の作成

calc.jj：以下の構文規則に従う式の値を計算

```
<List> ::= { <E> “=” }
```

```
<E> ::= <T> { ( “+” <T> ) | ( “-” <T> ) }
```

```
<T> ::= <F> { ( “*” <F> ) | ( “/” <F> ) | ( “%” <F> ) }
```

```
<F> ::= ( “(” <E> “)” ) | INTEGER
```

# サンプル jj ファイル calc.jj

sampleExp.txt

```
11 + 22 + 33 + 44 =
```

```
55 - 66 + 77 - 88 =
```

```
4 * ( 7 / 4 ) / 2 =
```

```
$ javacc calc.jj
```

```
$ cd ..
```

```
$ javac calc/Calc.java
```

```
$ java calc.Calc sampleExp.txt
```

```
110
```

```
22
```

```
2
```

# コンパイラコンパイラの入手先

lex, Flex	Linux, MacOS の基本ソフトとしてインストール済
yacc, Bison	Linux, MacOS の基本ソフトとしてインストール済
JavaCC	<a href="https://javacc.github.io/javacc/">https://javacc.github.io/javacc/</a>
JFlex	<a href="http://www.jflex.de/">http://www.jflex.de/</a>
Jay	<a href="http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html">http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html</a>
CUP	<a href="http://www2.cs.tum.edu/projects/cup/">http://www2.cs.tum.edu/projects/cup/</a>
Coco/R	<a href="http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/">http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/</a>
JS/CC	<a href="http://jscc.phorward-software.com/">http://jscc.phorward-software.com/</a>
ANTLR	<a href="http://www.antlr.org/">http://www.antlr.org/</a>