コンパイラ

第6回 構文解析
― 構文解析プログラムの作成 ―

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

E館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

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

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

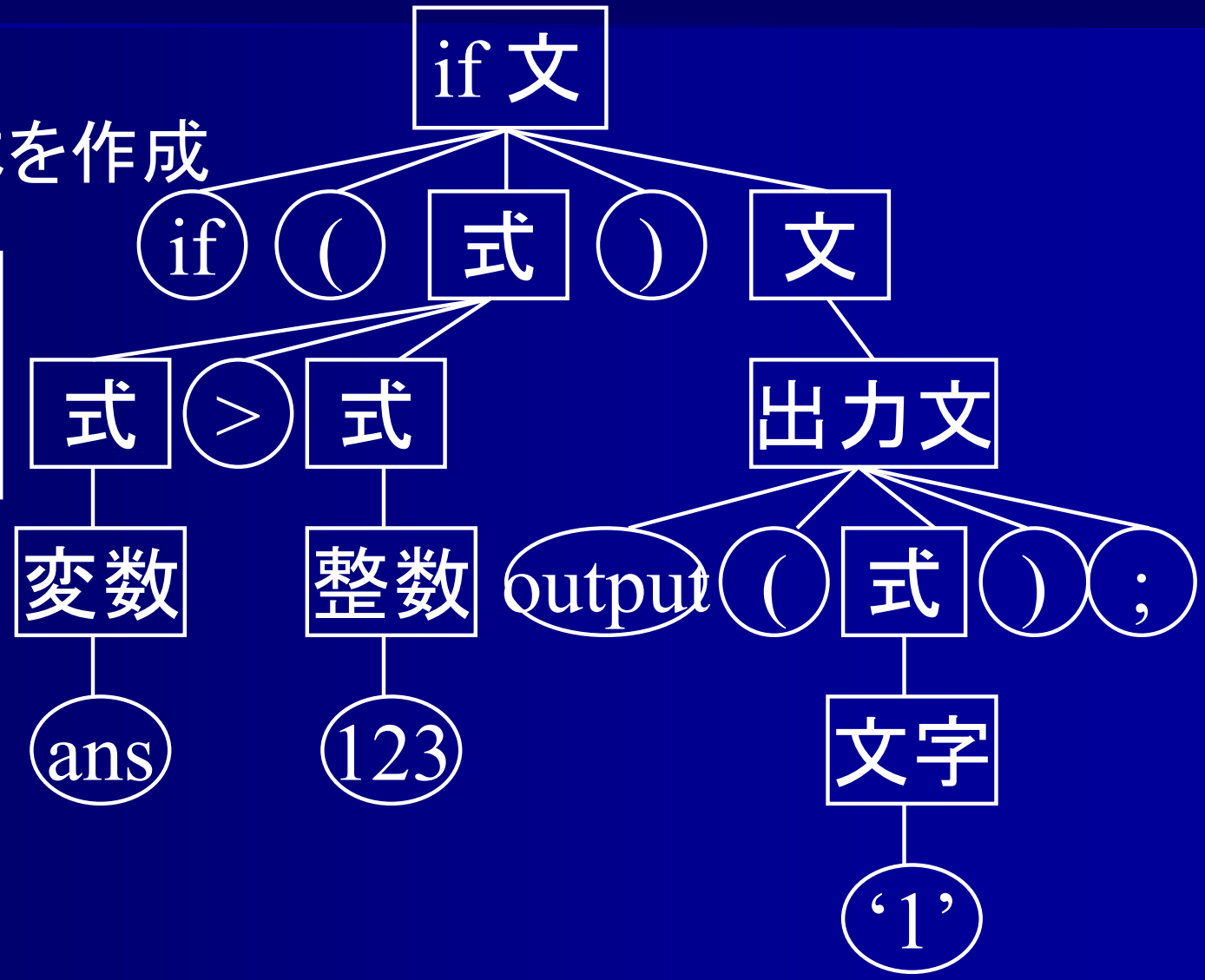
↓

1. PUSH &ab　　2. OUTPUT

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’) ;
```

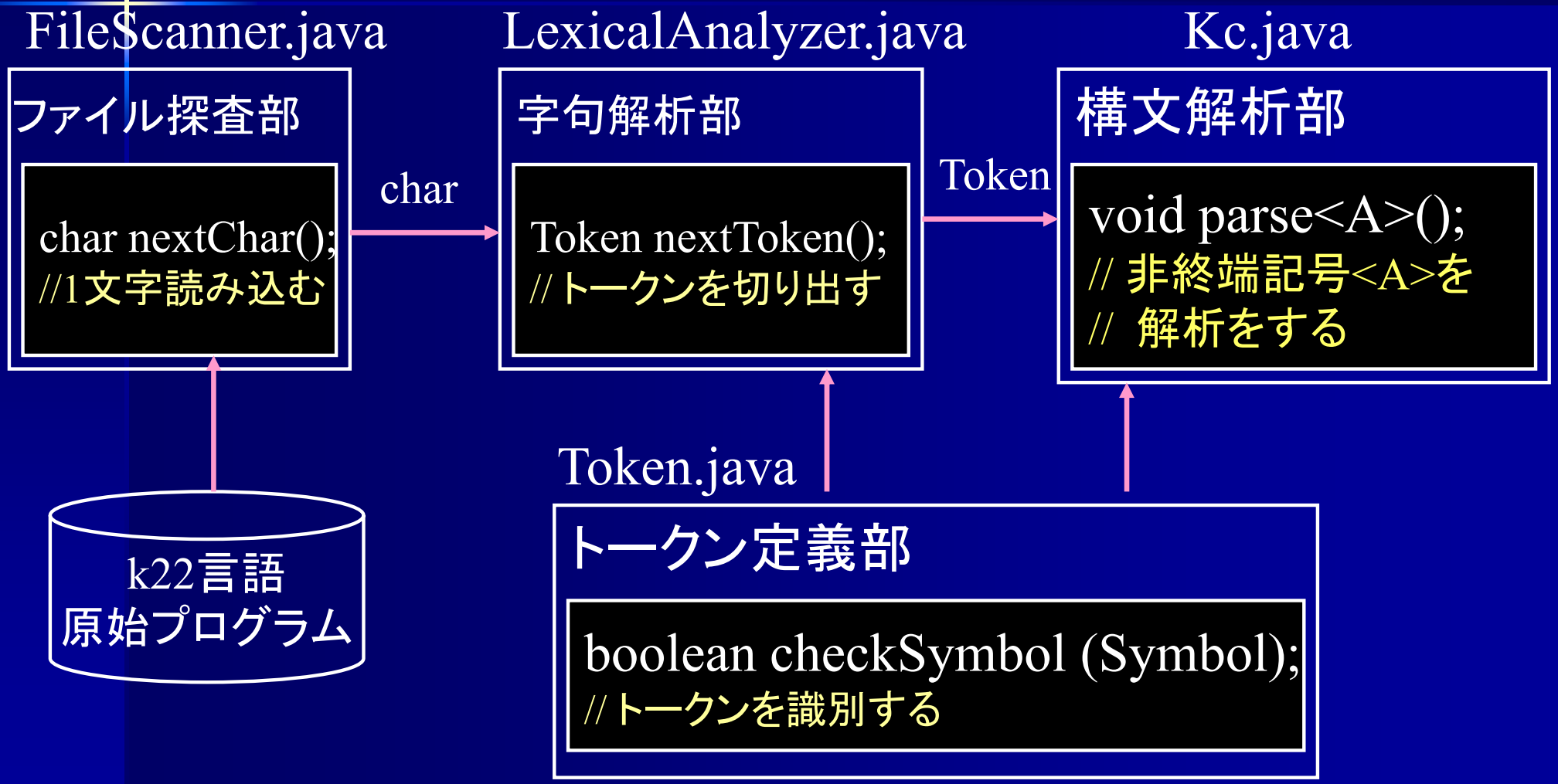


構文解析

情報システムプロジェクトI の構文解析

下降型解析 (top-down parsing)	再帰下降解析 (recursive descent parsing)
	LL解析 (Left to right scan & Left most derivation)
上昇型解析 (bottom-up parsing)	演算子順位構文解析 (operator precedence parsing)
	LR解析 (Left to right scan & Right most derivation)

プログラムの構造(構文解析系)



Kc クラス

Kc		# 構文解析部
- lexer	: LexicalAnalyzer	# 字句解析器
- token	: Token	# 読み取りトークン
Kc (sourceFileName : String)		# コンストラクタ
parseProgram()	: void	# <Program>の解析
parseMainFunction()	: void	# <MainFuntion>の解析
parseBlock()	: void	# <Block>の解析
parseVar_decl()	: void	# <Var_decl>の解析
:	:	
closeFile ()	: void	# 入力ファイルを閉じる
- syntaxError (message : String)	: void	# エラー検出時の処理
+ main (args : String[])	: void	# メイン

構文解析プログラム

- 非終端記号ごとに解析用メソッドを作成
  - 例 :非終端記号 <A> の解析

```
void parse<A> () {
  if (トークン列が<A>のマクロ構文と合致) {
      <A>のコード生成;
  } else syntaxError();
    /* マクロ構文と一致しなかった場合はエラー */
}
```

構文解析部

- 構文解析部のプログラム

```
Token token;                // 現在読み込み中のトークン
Token nextToken();          // 次のトークンを読み込む
```

例 <decl> ::= “int” NAME “;” の解析

```
void parseDecl () {
  if (token == “int”) token = nextToken();
    else syntaxError();
  if (token == NAME ) token = nextToken();
    else syntaxError();
  if (token == “;”) token = nextToken();
    else syntaxError();
}
```

マクロ構文と合致しているか？

合致すれば次のトークンを読み込む

合致しなければ構文エラー

Token クラス

Token		# トークン定義部
- symbol	: Symbol	# トークンの種類
- intValue	: int	# トークンの値
- strValue	: String	# トークンの名前
Token (symbol : Symbol)		# コンストラクタ
Token (symbol : Symbol, intValue : int)		# コンストラクタ
Token (symbol : Symbol, StrValue : String)		# コンストラクタ
checkSymbol (symbol : Symbol)	: boolean	# トークンの種類を比較
getSymbol ()	: Symbol	# トークンの種類を返す
getIntValue ()	: int	# トークンの値を返す
getStrValue ()	: String	# トークンの名前を返す

Token クラス

```
class Token {
    Symbol symbol; /* トークンの種類 */
    int intValue;        /* 整数値 または 文字コード */
    String strValue;    /* 変数名 または 文字列 */
}
```

トークン	symbol	intValue	strValue
main	MAIN		
==	EQUAL		
123	INTEGER	123	
‘a’	CHARACTER	97 (‘a’の文字コード)	
time	NAME		“time”

トークンの判定

- トークンの一致判定は
  Token.checkSymbol (Symbol) を利用

```
boolean checkSymbol (Symbol symbol)
```

例：トークンが “+”か？

```
token.checkSymbol (Symbol.ADD)
```

値を持つトークン

- 値を持つトークン
  - 整数(整数値)
  - 文字(文字コード)
  - 変数名(文字列)

intValue フィールドの値を得る

```
int getIntValue()
```

strValue フィールドの値を得る

```
String getStrValue()
```

```
int val = token.getIntValue();
```

トークンの値は制約検査部・コード生成部で必要

構文解析部

例 <decl> ::= “int” NAME “;” の解析

```
void parseDecl () {
   if (token.checkSymbol (Symbol.INT))
      token = nextToken();
      else syntaxError();
   if (token.checkSymbol (Symbol.NAME))
      token = nextToken();
      else syntaxError();
   if (token.checkSymbol (Symbol.SEMICOLON))
     token = nextToken();
     else syntaxError();
}
```

非終端記号 <A> の解析

- <A> ::= α (∈ (N∪T)*) の解析
  1. <A> ::= ε のとき
     何もしない
  2. <A> ::= “a” (∈ T) のとき

```
if (token == “a”) token = nextToken();
    else syntaxError();
```

非終端記号 <A> の解析

- <A> ::= α (∈ (N∪T)*) の解析

この判定には<B>のFirst集合が必要

3. <A> ::= <B> (∈ N) のとき
   1. ε ∉ First (<B>) のとき

```
if (token ∈ First (<B>)) parse<B>();
   else syntaxError();
```

<B>解析メソッドへ

   2. ε ∈ First (<B>) のとき

```
if (token ∈ (First (<B>)-ε)) parse<B>();
```

else 節無し

非終端記号 <A> (分岐) の解析

- <A> ::= α (∈ (N∪T)*) の解析

4. <A> ::= $\beta_1 \mid \beta_2 \mid \beta_3 \mid \varepsilon$ のとき

ε あり

```
if (token ∈ First (β1)) {
    β1の解析;
} else if (token ∈ First (β2)) {
    β2の解析;
} else if (token ∈ First (β3)) {
    β3の解析;
} else ;
```

ε に対応

非終端記号 <A> (分岐) の解析

- <A> ::= α (∈ (N∪T)*) の解析

  4. <A> ::= $\beta_1 \mid \beta_2 \mid \beta_3$ のとき

ε 無し

```
if (token ∈ First (β1)) {
    β1の解析;
} else if (token ∈ First (β2)) {
    β2の解析;
} else if (token ∈ First (β3)) {
    β3の解析;
} else syntaxError();
```

合致しなければ
構文エラー

非終端記号 <A> (分岐) の解析

- <A> ::= α (∈ (N∪T)*) の解析

4. <A> ::= $\beta_1 \mid \beta_2 \mid \beta_3 \mid \varepsilon$ のとき

```
switch (token) {
    case First (β1) : β1の解析; break;
    case First (β2) : β2の解析; break;
    case First (β3) : β3の解析; break;
    default : break;
}
```

ε に対応
ε が無い場合は default : syntaxError();

非終端記号 <A> (連接) の解析

- <A> ::= α (∈ (N∪T)*) の解析

  5. <A> ::= $\beta_1\beta_2\beta_3$ のとき

```
β1の解析;
β2の解析;
β3の解析;
```

非終端記号 <A> (閉包) の解析

- <A> ::= α (∈ (N∪T)*) の解析

6. <A> ::= {β} のとき

```
while (token ∈ First (β)) {
    βの解析;
}
```

非終端記号 <A> (省略) の解析

- <A> ::= α (∈ (N∪T)*) の解析

7. <A> ::= [β] のとき

```
if (token ∈ First (β)) {
    βの解析;
}
```

else syntaxError(); は付けない

非終端記号 <A> (括弧) の解析

- <A> ::= α (∈ (N∪T)*) の解析

8. <A> ::= (β) のとき

βの解析;

非終端記号解析の例

- 例 : <MainFunction> ::= “main” “(” “)” <Block>

```
parseMainFunction() {
    if (token == “main”) token = nextToken();
        else syntaxError ();
    if (token == “(”) token = nextToken();
        else syntaxError ();
    if (token == “)”) token = nextToken();
        else syntaxError ();
    if (token ∈ First (<Block>)) parseBlock();
        else syntaxError ();
}
```

終端記号なら次のトークンを読む

非終端記号なら対応するメソッドへ

この判定には<Block>のFirst集合が必要

非終端記号解析の例 (分岐)

- 例 : <Factor> ::= NAME | INTEGER
  | CHARACTER | “input”

```
parseFactor() {
    switch (token) {
        case NAME :           token = nextToken(); break;
        case INTEGER :        token = nextToken(); break;
        case CHARACTER : token = nextToken(); break;
        case “input” :          token = nextToken(); break;
        default :                syntaxError ();
    }
}
```

εに対応

非終端記号解析の例 (閉包)

- 例 : <Block> ::= “{” { <Decl> } { <St> } “}”

```
parseBlock() {
    if (token == “{”) token = nextToken();
        else syntaxError();
    while (token ∈ First (<Decl>)) parseDecl(); // { <Decl> }
    while (token ∈ First (<St>)) parseSt();          // { <St> }
    if (token == “}”) token = nextToken();
        else syntaxError();
}
```

この判定には <Decl>, <St> のFirst 集合が必要

非終端記号解析の例 (閉包)

- 例 : <Block> ::= “{” { <Decl> } { <St> } “}”

First (<Decl>) = {“int”}

First (<St>) = {“if”, “while”, NAME, INTEGER, “output”, …}

```
parseBlock() {
    if (token == “{”) token = nextToken(); else syntaxError();
    while (token == “int”) parseDecl();
    while (token == “if” || token == “while”
            || token == NAME || token == INTEGER
            || token == “output” || … ) parseSt();
    if (token == “}”) token = nextToken(); else syntaxError();
}
```

First 集合の判定

First (<St>)
= {“if”, “while”, “break”, “output”, …}
First (<Exp>)
= {INTEGER, NAME, “input”, “++”, …}

要素数が多い

要素数が多い場合は判定用メソッドを作っておくと便利

```
/* token が <St> のFirst 集合なら true を返す */
boolean isStFirst (Token token) {
    return (token == “if”  || token == “while”
            || token == “break” || token == “output”
            || … );
}
```

非終端記号解析の例 (省略)

- 例 : <Decl> ::= “int” NAME [“=” <Const>] “;”

```
parseDecl() {
    if (token == “int”) token = nextToken();
        else syntaxError ();
    if (token == NAME) token = nextToken();
        else syntaxError ();
    if (token == “=”) {
        token = nextToken();
        if (token ∈ First (<Const>)) parseConst();
            else syntaxError ();
    }
    if (token == “;”) token = nextToken(); else syntaxError();
}
```

[] 内の解析

else syntaxError() は付けない

左再帰性のある場合の解析

- 例 : <Term> ::= <Tetm> “+” <Factor > | <Factor>
  このままプログラムすると…

左再帰

```
parseTerm() {
    if (token ∈ First (<Term>)) {
        parseTerm();
        if (token == “+”) token = nextToken();
            else syntaxError();
        if (token ∈ First (<Factor>)) parseFactor();
            else syntaxError();
    } else if (token ∈ First (<Factor>)) parseFactor();
    else syntaxError();
}
```

左再帰性があると
無限ループに陥る

左再帰性のある場合の解析

- 例 : <Term> ::= <Tetm> “+” <Factor > | <Factor>

左再帰性の除去を行う

右再帰に

<Term> ::= <Factor> <T’>

<T’> ::= “+” <Factor > <T’> | ε

EBNF記法に

<Term> ::= <Factor> { “+” <Factor > }

左再帰性のある場合の解析

右再帰に : <Term> ::= <Factor> <T'>

<T'> ::= "+" <Factor > <T'> | ε

```
parseTerm() {
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    if (token == "+") parseT'();
}
parseT'() {
    if (token == "+" ) {
        token = nextToken();
        if (token ∈ First (<Factor>)) parseFactor();
            else syntaxError();
        if (token == "+") parseT'();
    }
}
```

左再帰性のある場合の解析

EBNF記法に: <Term> ::= <Factor > { “+” <Factor> }

```
parseTerm() {
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    while (token == “+”) {
        token = nextToken();
        if (token ∈ First (<Factor>))
            parseFactor();
            else syntaxError();
    }
}
```

{} 内の解析

同一の接頭部を持つ場合の解析

- 例 : <Term> ::= <Factor > “+” <Term> | <Factor>

同一の接頭部

このままプログラムすると…

```
parseTerm() {
    if (token ∈ First (<Factor>)) {
        parseFactor();
        if (token == “+”) token = nextToken();
            else syntaxError();
        if (token ∈ First (<Term>)) parseTerm();
            else syntaxError();
    } else if (token ∈ First (<Factor>)) parseFactor();
    else syntaxError();
}
```

同一の条件式

絶対にこの分岐には入らない

同一の接頭部を持つ場合の解析

- 例 : <Term> ::= <Factor> “+” <Term> | <Factor>

左括り出しを行う

<Term> ::= <Factor> [“+” <Term>]

同一の接頭部を持つ場合の解析

左括り出し: <Term> ::= <Factor > [“+” <Term>]

```
parseTerm() {
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    if (token == “+”) {
        token = nextToken();
        if (token ∈ First (<Term>))
            parseTerm();
            else syntaxError();
    }
}
```

[] 内の解析

同一の接頭部を持つ場合の解析

- 例 : <Term> ::= <Factor> “+” <Term>
  | <Factor> “-” <Term>
  | <Factor>

左括り出し

<Term> ::= <Factor> [“+” <Term> | “-” <Term>]

ここもまとめる

<Term> ::= <Factor> [(“+” | “-”) <Term>]

同一の接頭部を持つ場合の解析

<Term> ::= <Factor > [(“+” | “-”) <Term>]

```
parseTerm() {
    if (token ∈ First (<Factor>)) parseFactor();
        else syntaxError();
    if (token == “+” || token == “-”) {
        token = nextToken();
        if (token ∈ First (<Term>))
            parseTerm();
            else syntaxError();
    }
}
```

[] 内の解析

構文解析時のエラー処理

- 入力がマクロ構文と一致しなかった
  ⇒構文解析エラー

```
parseMainFunction() {
    if (token == “main”) token = nextToken();
      else syntaxError (“main がありません”);
    if (token == “(”) token = nextToken();
      else syntaxError (“( がありません”);
    if (token == “)”) token = nextToken();
      else syntaxError (“) がありません”);
    if (token ∈ First (<Block>)) parseBlock();
      else syntaxError (First (<Block>) + “がありません”);
}
```

エラー検出時はエラーメッセージを表示して停止

構文解析時のエラー処理

エラー検出時はエラーメッセージを表示して停止

```
private void syntaxError (String err_mes) {
    System.out.println (analyzeAt() + “でエラー検出”);
      /* LexicalAnalyzer の analyzeAt() を用いてエラー位置表示 */
    System.out.println (err_mes);  /* エラーメッセージ表示 */
    closeFile ();                  /* 入力ファイルを閉じる */
    System.exit (0);               /* プロクラム停止 */
}
```

エラー箇所およびエラー原因がユーザに
分かり易いエラーメッセージを作成する

プログラム末到達時の処理

- プログラム末到達時にファイル末ならば
  コンパイル完了

```
void parseProgram () {
    if (token ∈ First (<MainFunction>))
        parseMainFunction();
        else syntaxError ();
    if (token == “$”) {
       コンパイル完了処理
    } else syntaxError (“ファイル末ではありません”);
}
```

ファイル末を示すトークン

デバグ用に...

```
static final boolean DEBUG = false;
```

```
void parse<A> () {
        :
   if (DEBUG)
     System.out.println ( /* デバグ情報を出力 */ );
        :
}
```

DEBUG = true; として実行するとデバグ情報出力

(デバガがあるなら不要)

トレース機能

```
static final boolean TRACE = false;
int level = 0;
```

```
void parse<A> () {
    ++level;
    if (TRACE) {
      for (int i=0; i<level; ++i) System.out.print (“ ”);
      System.out.println (“<A>開始”);
    }
    // <A> 解析
    if (TRACE) {
      for (int i=0; i<level; ++i) System.out.print (“ ”);
      System.out.println (“<A>終了”);
    }
   --level;
}
```

トレース機能

```
% java kc.Kc foo.k
 program 開始
   main 開始
     ver_decl 開始
     ver_decl 終了
     statement 開始
       if_statement 開始
         exp 開始
                                  :
```

トレース機能があると(ユーザにとって)便利

LL(1)文法

- LL(1)文法
  - 1個のトークン(直後に来るトークン)の先読みで構文解析可能な文法
    - 左辺が同じ生成規則が複数あるとき、トークンを1個先読みすればどの右辺を選択するかわかる
    - 同一の左辺に対して、右辺の先頭トークン（終端記号）が全て異なる

次に来るトークンを先読みする
メソッドがあれば解析可能

構文解析不能な文法

例 : First (α) = {“x”, “a”}

First (β) = {“x”, “b”}

Firsr (γ) = {“x”, “c”}

<A> ::= α|β|γ

<B> ::= {α}β

<C> ::= [α] β

<A> <B> <C> 共に
先頭の終端記号が “x” だと
どの分岐か判定できない

左括り出しも難しい

LL(1) 文法でないとバックトラック無しでは
構文解析不能

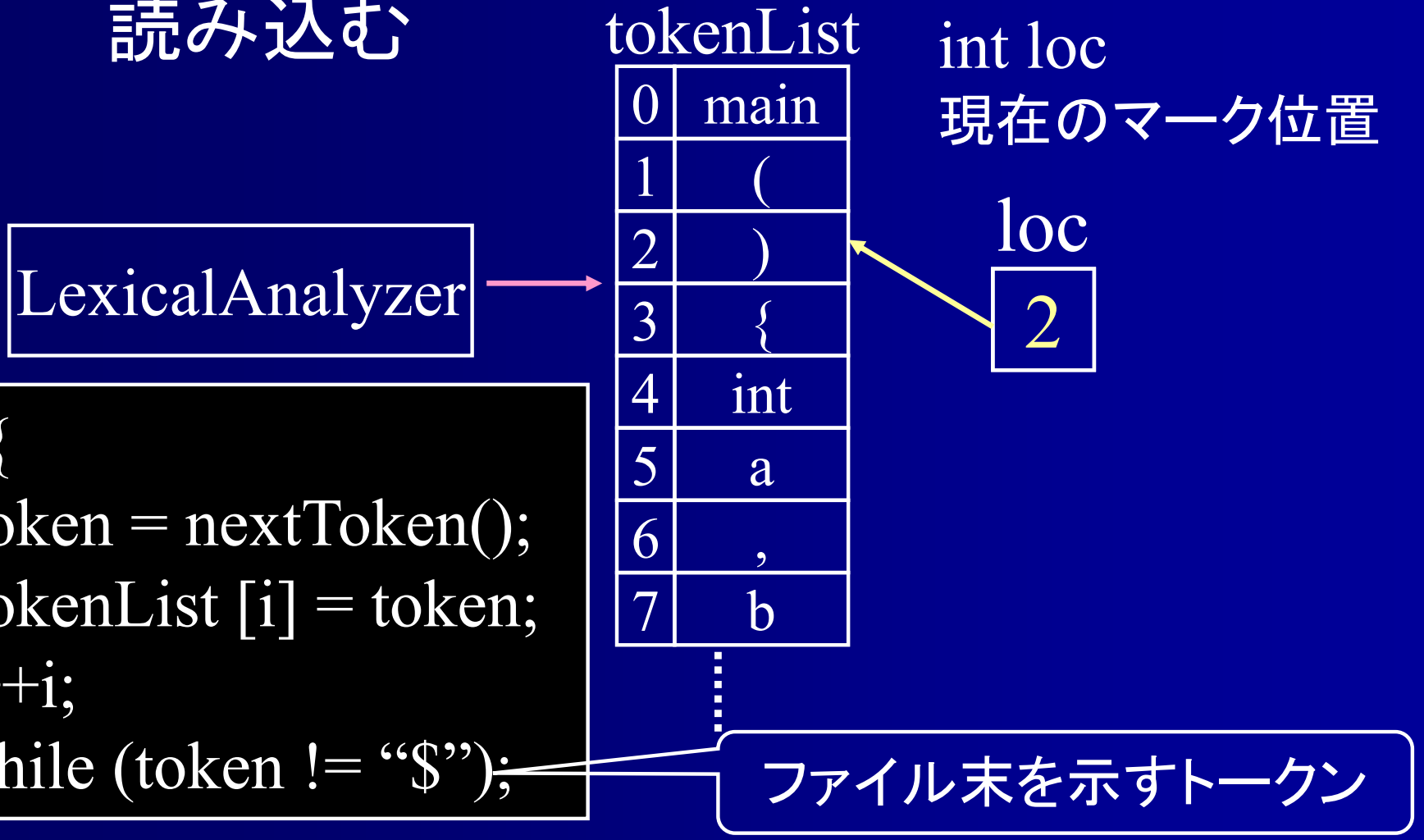
バックトラックありの構文解析

- 構文解析メソッドを boolean 型に
  - 解析完了 ⇒ true 解析失敗 ⇒ false
  - 返り値が false ならば次の導出パターンへ

```
/* <A>の構文解析を行う
   構文解析を完了できれば true を返す */
boolean parse<A> () {
    if (トークン列が<A>のマクロ構文と合致) {
        return true;    // 構文解析完了
    } else {
        return false;   // 構文解析失敗
    }
}
```

バックトラックありの構文解析

- 字句解析器から1度に全てのトークンを読み込む



バックトラックありの構文解析

```
int loc;                    // 現在のマーク位置

/* マーク位置を1つ進める */
void proceed() {
    ++loc;
    token = tokenList [loc];
}

/* マーク位置を指定の位置に戻し、生成したコードを削除する */
void backtrack (int backPoint) {
    tokenList [backPoint +1 ] ～ tokenList [loc] のコードを削除
    loc = backPoint;
    token = tokenList [loc];
}
```

バックトラックありの構文解析部

- 構文解析部のプログラム

```
int loc;                    // 現在のマーク位置
void proceed();             // マーク位置を1つ進める
void backtrack();           // マークを指定の位置に戻し、コードを削除する
```

```
/* <A>の構文解析を行う 構文解析を完了できれば true を返す */
boolean parse<A> () {
   int backPoint = loc;             // 開始位置を記憶
            :
   proceed(); return true;          // 構文解析完了
            :
   backtrack (backPoint); return false;   // 構文解析失敗
}
```

開始位置まで戻る

バックトラックありの構文解析の例

文末を示すトークン

- 生成規則
  - <E> ::= <T> “$” | <F> “$”
  - <T> ::= <F> “+” <F>
  - <F> ::= “a” | “b” | “c”

First (<T>) = {“a”, “b”, “c”}
First (<F>) = {“a”, “b”, “c”}

First 集合で判定できない

⇒バックトラック無しには構文解析不能

バックトラックありの構文解析の例

- 生成規則
  - <E> ::= <T> “$” | <F> “$”
  - <T> ::= <F> “+” <F>
  - <F> ::= “a” | “b” | “c”

```
boolean parseF(){
    if (token == “a” | token == “b” | token == “c”) {
        proceed();          // マーク位置を1つ先へ
        return true;        // 解析完了 <F> ::= “a” | “b”
    } else  return false; // 解析失敗
}
```

バックトラックありの構文解析

– <T> ::= <F> “+” <F>

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
             proceed();
             if (parseF()) {
                  return true; // 解析完了 <T> ::= <F> “+” <F>
             } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
    } else { backtrack (backPoint);  return false; }
}
```

マークを初期位置に戻す

バックトラックありの構文解析

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {   // <E> ::= <T> “$” の解析
        if (token == “$”) {
             proceed();  return true;       // 解析完了 <E> ::= <T> “$”
        } else backtrack (backPoint); // 解析失敗, バックトラック
   }
   if (parseF()) {   // <E> ::= <F> “$” の解析
         if (token == “$”) {
             proceed();  return true;       // 解析完了 <E> ::= <F> “$”
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

マークを初期位置に戻す

構文解析の例

入力列 : “a” “+” “b” “$”

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
         if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

構文解析の例

入力列：“a” “+” “b” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
            proceed();
            if (parseF()) {
                return true; // 解析完了
            } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
    } else { backtrack (backPoint);  return false; }
```

構文解析の例

入力列：“a” “+” “b” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
            proceed();
            if (parseF()) {
                return true; // 解析完了
            } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
   } else { backtrack (backPoint);  return false; }
```

構文解析の例

入力列：“a” “+” “b” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
            proceed();
            if (parseF()) {
                return true; // 解析完了
            } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
    } else { backtrack (backPoint);  return false; }
```

構文解析の例

入力列：“a” “+” “b” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
             proceed();
             if (parseF()) {
                  return true; // 解析完了
             } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
   } else { backtrack (backPoint);  return false; }
}
```

構文解析の例

入力列 : “a” “+” “b” “$”

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
         if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

構文解析の例

入力列 : “a” “+” “b” “$”　構文解析完了

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
         if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

構文解析の例

入力列：“c” “$”

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
         if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

構文解析の例

入力列：“c” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
            proceed();
            if (parseF()) {
                return true; // 解析完了
            } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
    } else { backtrack (backPoint);  return false; }
}
```

構文解析の例

入力列：“c” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
             proceed();
             if (parseF()) {
                  return true; // 解析完了
             } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
   } else { backtrack (backPoint);  return false; }
```

開始位置に戻る

構文解析の例

入力列：“c” “$”

```
boolean parseF(){
   int backPoin
   if (parseT())
        if (token
            proce
        } else bac
   }
   if (parseF())
        if (token
            proce
        } else { b
   } else { back }
}
```

```
boolean parseT(){
   int backPoint = loc; // 開始位置を記憶
   if (parseF()) {
        if (token == “+”) {
             proceed();
             if (parseF()) {
                  return true; // 解析完了
             } else { backtrack (backPoint);  return false; }
        } else { backtrack (backPoint);  return false; }
   } else { backtrack (backPoint);  return false; }
}
```

構文解析の例

入力列 : “c” “$”

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
        if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

構文解析の例

入力列：“c” “$”

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
        if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```

構文解析の例

入力列：“c” “$”　　構文解析完了

```
boolean parseE(){
   int backPoint = loc;   // 開始位置を記憶
   if (parseT()) {
        if (token == “$”)
             proceed();  return true;      // 解析完了
        } else backtrack (backPoint); // 解析失敗
   }
   if (parseF()) {
         if (token == “$”) {
             proceed();  return true;      // 解析完了
        } else { backtrack (backPoint); return false; }
   } else { backtrack (backPoint); return false; }
}
```