コンパイラ

第5回 下降型構文解析

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

1

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

2

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

3

マクロ構文
(情報システムプロジェクトIの場合)

- マクロ構文 (EBNF記法で定義)

<Program> ::= <Main_function> EOF　（ファイル末）

<Main_function> ::= “main” “(” “)” <Block>

<Block> ::= “{” { <Var_decl> } { <Statement> } “}”

（0回以上の繰り返し／変数宣言／文）

4

マクロ構文(変数宣言部)

<Var_decl>(変数宣言) ::= “int” <Name_list> “;”

<Name_list>(変数名列) ::= <Name_list> “,” <Name> | <Name>

<Name>(変数名) ::= NAME　（再帰）
| (NAME “=” <Constant>)
| (NAME “[” INTEGER “]”)
| (NAME “[” “]” “=” “{” <Constant_list> “}”)
| (NAME “[” “]” “=” STRING
| (NAME “[” INTEGER “]” “[” INTEGER “]”) (拡張課題)

<Constant_list>(定数列) ::= <Constant_list> “,” <Constant>
| <Constant>　（再帰）

<Constant>(定数) ::= ([“-”] INTEGER) | CHARACTER

例：int i, n=-5, a[10], b[]={‘a’, ‘b’, ‘c’};

5

マクロ構文(文)

<Statement>(文) ::= <If_statement>
| <While_statement>
| <For_statement>
| <Exp_statement>
| <Outputint_statement>
| <Outoutchar_statement>
| <Break_statement>
| “{” { <Statement> } “}”
| “;”
| <Do-while_statement> (拡張課題)
| <Continue_statement> (拡張課題)
| <Switch_statement> (拡張課題)

6

マクロ構文(文)

<If_statement>(if文)
::= “if” “(” <Expression> “)” <Statement>

<While_statement>(while文)
::= “while” “(” <Expression> “)” <Statement>

<If_Statement> (if文) (拡張課題)
::= “if” “(” <Expression> “)” <Statement>
[“else” <Statement>]

7

マクロ構文(文)

<For_statement>(for文))
::= “for” “(” <Expression> “;”
<Expression> “;”
<Expresson> “)” <Statement>

<For_statement>(for文) (拡張課題)
::= “for” “(” [<Expression> { “,” <Expresson> }] “;”
[<Expression>] “;”
[<Expresson> { “,” <Expresson> }] “)” <Statement>

例 : for (; ;) outputchar (‘!’);
for (i=0, j=1, k=2; i<10; ++i, ++j, ++k);

8

マクロ構文(文)

<Exp_statement>(式文) ::= <Expression> “;”
<Outputchar_statement>(出力文)
::= “outputchar” “(” <Expression> “)” “;”
<Outputint_statement>(出力文)
::= “outputint” “(” <Expression> “)” “;”

<Break_statement>(break文) ::= “break” “;”

9

マクロ構文(文:拡張課題)

<Do-while_statement>(do-while文)
::= “do” <Statement>
“while” “(” <Expression> “)” “;”
<Continue_statement>(continue文) ::= “continue” “;”

10

マクロ構文(論理式)

<Expression>(式) ::= <Exp>
[(“=” | “+=” | “-=” | “*=” | “/=”) <Expression>]
<Exp>(論理式) ::= <Exp> “||” <Logical_term>
| <Logical_term> 再帰
<Logical_term>(論理項)
::= <Logical_term> “&&” <Logical_factor>
| <Logical_factor>
<Logical_factor>(論理因子) ::= <Arithmetic_expression>)
[(“==” | “!=” | “<” | “>” | “<=” | “>=”)
<Arithmetic_expression>]

11

マクロ構文(算術式)

<Arithmetic_expression>(算術式)
::= <Arithmetic_expression> 再帰
(“+” | “-”) <Arithmetic_term>
| <Arithmetic_term>
<Arithmetic_term> (算術項)
::= <Arithmetic_term>
(“*” | “/” | “%”) <Arithmetic_factor>
| <Arithmetic_factor>
<Arithmetic_factor>(算術因子) ::= <Unsigned_factor>
| “-” <Arithmetic _factor>
| “!” <Arithmetic _factor>

12

マクロ構文(符号無し因子)

<Unsigned_factor>(符号無し因子) ::= NAME
| NAME “[” <Expression> “]”
| NAME (“++” | “--”)
| (“++” | “--”) NAME
| (“++” | “--”) NAME “[” <Expression> “]”
| INTEGER | CHARACTER
| “inputchar” | “inputint”
| “(” <Expression> “)”
| <Sum_function> | <Product_function>
| NAME “[” <Expresson> “]” (“++” | “--”) (拡張課題)
| NAME “[” <Expresson> “]” “[” <Expresson> “]”(拡張課題)

13

マクロ構文 (和関数・積関数)

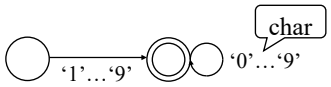
<Sum_function> (和関数)
::= “+” “(” <Expression_list> “)”
<Product_function> (積関数)
::= “*” “(” <Expression_list> “)”
<Expression_list> (式の並び) ::=
::= <Expression>
| <Expression_list> “,” <Expression>

再帰

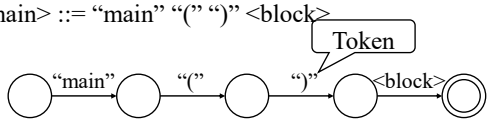
14

字句解析と構文解析

字句解析系 : マイクロ構文の解析
INTEGER ::= ‘0’| Pdec { Dec }



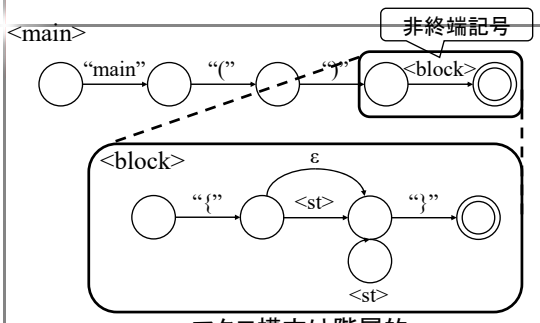
構文解析系 : マクロ構文の解析
<main> ::= “main” “(” “)” <block>



解析対象が char か Token かが違うだけ？

15

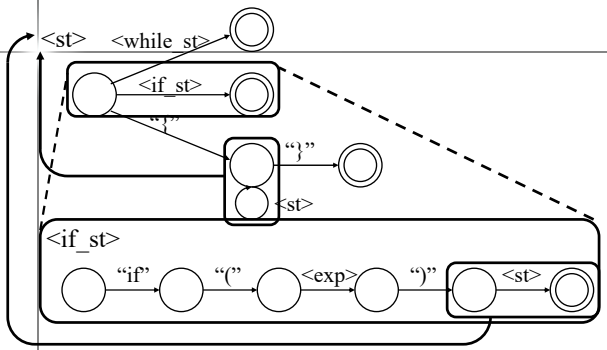
マクロ構文の解析



マクロ構文は階層的

16

マクロ構文の解析



マクロ構文は再帰的

17

字句解析と構文解析

- 字句解析系 : マイクロ構文を解析
  - 文字列 ⇒ トークン
- 構文解析系 : マクロ構文を解析
  - トークン列 ⇒ ？？？

⇒ 字句解析系と同じ処理で一応解析可能

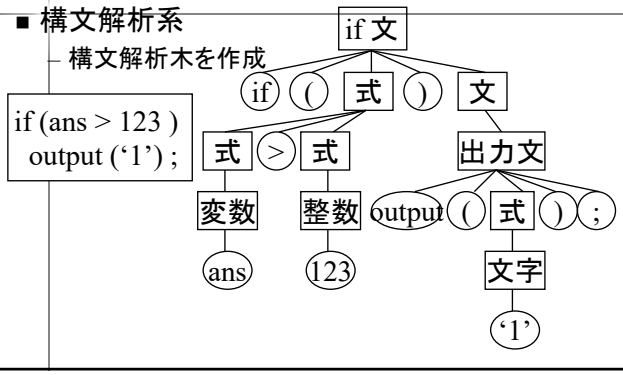
しかしマクロ構文はマイクロ構文よりも複雑
(階層的, 再帰的)

構文解析木を生成する
トークン列 ⇒ 構文解析木

18

構文解析系 (syntax analizer, parser)

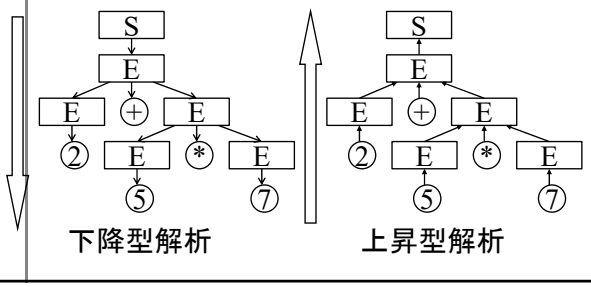
- 構文解析系
  - 構文解析木を作成



19

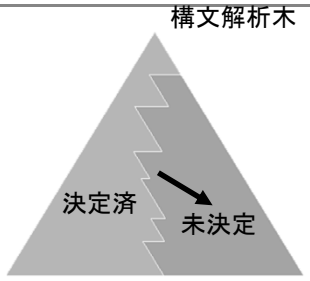
構文解析

- 文法 G = {**N**, **T**, S, **P**} が与えられたとき、
  ω∈**T*** に対してS⇒ω であるか判定,
  その導出木を得る



20

下降型解析(top-down parsing)



入力記号列 | 既読 | 未読

21

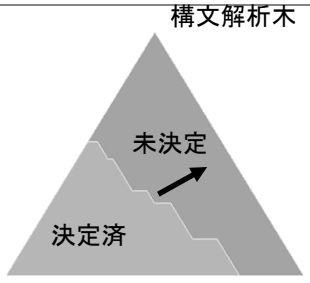
下降型解析の例

<namelist> ::= <name> | <name> “,” <namelist>
<name> ::= “a” | “b” | “c”

a, b, c <namelist>
→ <name> “,” <namelist>
→ “a” “,” <namelist>
→ “a” “,” <name> “,” <namelist>
→ “a” “,” “b” “,” <namelist>
→ “a” “,” “b” “,” <name>
→ “a” “,” “b” “,” “c”
<namelist> ⇒ a,b,c

22

上昇型解析(bottom-up parsing)



入力記号列 | 既読 | 未読

23

上昇型解析の例

<namelist> ::= <name> | <namelist> “,” <name>
<name> ::= “a” | “b” | “c”

a, b, c “a” “,” “b” “,” “c”
→ <name> “,” “b” “,” “c”
→ <namelist> “,” “b” “,” “c”
→ <namelist> “,” <name> “,” “c”
→ <namelist> “,” “c”
→ <namelist> “,” <name>
→ <namelist>
<namelist> ⇒ a,b,c

24

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)

25

最左導出(left most derivation)

■ 一番左にある非終端記号から置き換える

例 : **N**={S,A,B,C,D}
T={a,b,c,d}
P={S→ABC, A→a, B→bD, C→c, D→d}

S→ABC
ABC→**a**BC
a**B**C→a**bD**C
ab**D**C→ab**d**C
abd**C**→abd**c**

⇔最右導出(right most derivation)

26

最左導出の利点

■ 左から右に順に置き換えていけばいい
– 左に戻る必要が無い

abcd **E**FGhIjK
↑これを変換
abcd**e** FGhIjK
←
これより前は変換済
⇒変換場所を左に戻す必要が無い

27

最左導出の例

例 : **N**={E,T,F}
T={a,b,c,d,*,+}
P={E→T+T | T, T→F*F | F, F→a | b | c | d}

a*b+c*d

E→T+T →F*F+T (T→F*F)
→a*F+T (F→a)
→a*b+T (F→b)
→a*b+F*F (T→F*F)
→a*b+c*F (F→c)
→a*b+c*d (F→d)

28

最左導出の例

if (ans >= 123) output ('a');

<st> → **<if_st>**
→ "if" "(" **<exp>** ")" <st>
→ "if" "(" **<factor>** ">=" <factor> ")" <st>
→ "if" "(" "ans" ">=" **<factor>** ")" <st>
→ "if" "(" "ans" ">=" "123" ")" **<st>**
→ "if" "(" "ans" ">=" "123" ")" **<output_st>**
→ "if" "(" "ans" ">=" "123" ")" "output" "(" **<exp>** ")" ";"
→ "if" "(" "ans" ">=" "123" ")" "output" "(" "'a'" ")" ";"

29

再帰性(recurtion)

■ マクロ構文は再帰的に定義

例 : <st> ::= <if_st>
| <while_st>
| "{" {<st>} "}"
<if_st> ::= "if" "(" <exp> ")"<st>
<while_st> ::= "while" "(" <exp> ")"<st>

ある非終端記号からの導出時に
同一の非終端記号に戻ってくる可能性がある

30

左再帰性(left recurtion)

- 左再帰性
  - 右辺の左端に自分自身への再帰がある

直接左再帰 : <A> → <A>α | β

間接左再帰 : <A> → <B>α | β
<B> → <A>γ | δ

31

右再帰性(right recurtion)

- 右再帰性
  - 右辺の右端に自分自身への再帰がある

直接右再帰 : <A> → α<A> | β

間接右再帰 : <A> → α<B> | β
<B> → γ<A> | δ

32

構文解析の問題点
左再帰性

- 左再帰性
  - 最左の非終端記号が自分自身だと停止しない

例 : <E> ::= <E> “+” <T> | <T>

E→**E**+**T**
E+T→**E**+**T**+T
E+T+T→**E**+**T**+T+T
E+T+T+T→**E**+**T**+T+T+T
永久に停止しない

33

左再帰性の除去

- 左再帰の対処法1 :
  左再帰⇒右再帰に変形
  - <E> ::= <E> “+” <T> | <T>

右再帰に

<E> ::= <T> <E’>
<E’> ::= “+” <T> <E’> | ε

再帰しない部分を先頭に移動させる

新たな非終端記号を加える

34

構文解析の問題点
左再帰性

- 左再帰の対処法2 :
  BNF記法 ⇒ EBNF記法 に変形
  - <E> ::= <E> “+” <T> | <T>

ENBF記法に

<E> ::= <T> { “+” <T> }

再帰しない部分を先頭に移動させる

{} で囲む

35

左再帰性の除去

再帰しない部分を先頭に移動させる

<A> ::= <A>α|β

右再帰に

<A> ::= β<A’>
<A’> ::= α<A’> | ε

EBNF記法に

<A> ::= β{α}

L(G) = {β, βα, βαα, βααα, …}

36

左再帰性の除去

<A> ::= <A>α_1|<A>α_2|β_1|β_2

右再帰に　　EBNF記法に

<A> ::= (β_1|β_2) <A'>
<A'> ::= (α_1|α_2) <A'> | ε

<A> ::= (β_1|β_2){α_1|α_2}

L(G) = {β_1, $\beta_1\alpha_1$, $\beta_1\alpha_2$, $\beta_1\alpha_1\alpha_1$, $\beta_1\alpha_1\alpha_2$, …,
β_2, $\beta_2\alpha_1$, $\beta_2\alpha_2$, $\beta_2\alpha_1\alpha_1$, $\beta_2\alpha_1\alpha_2$, …,}

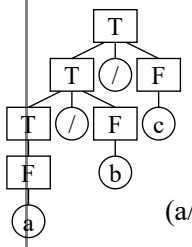
上記の手法で左再帰性は除去可能
しかし演算子の結合性が失われてしまう

37

左再帰性の除去の問題点

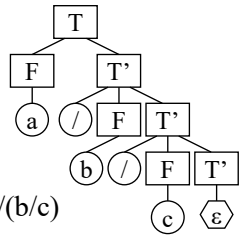
a/b/c

<T> ::= <T> "/" <F> | <F>
<F> ::= "a" | "b" | "c"



(a/b)/c

<T> ::= <F> <T'>
<T'> ::= "/" <F> <T'> | ε
<F> ::= "a" | "b" | "c"



a/(b/c)

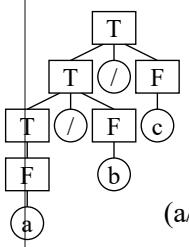
右再帰にすると構文木の構造が変わってしまう

38

左再帰性の除去の問題点

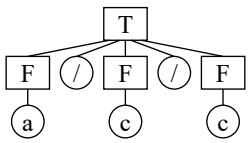
a/b/c

<T> ::= <T> "/" <F> | <F>
<F> ::= "a" | "b" | "c"



(a/b)/c

<T> ::= <F> { "/" <F>}
<F> ::= "a" | "b" | "c"



a/b/c ← (a/b)/c? a/(b/c)?

EBNFにすると結合性に関する情報が消えてしまう

39

演算子の結合性

- 左再帰性の除去
  ⇒演算子の結合性の情報が失われる

コード生成時に演算子の結合性を考慮する

右側 / 左側	=	&&	||	==	+, -	*, /
=	<	<	<	<	<	<
&&	>	>	<	<	<	<
||	>	>	>	<	<	<
==	>	>	>	E	<	<
+, -	>	>	>	>	>	<
*, /	>	>	>	>	>	>

>：左側の演算子を優先
<：右側の演算子を優先
E：文法に不適合(エラー)

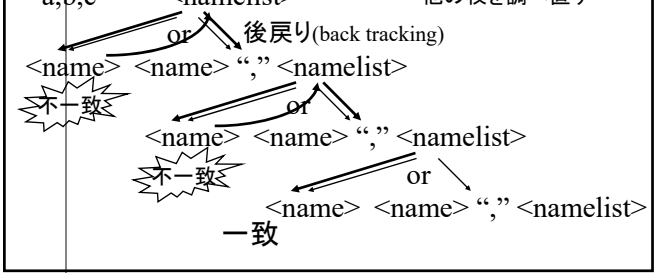
40

下降型解析の問題点

<namelist> ::= <name> | <name> "," <namelist>
<name> ::= "a" | "b" | "c"

不一致の場合は
他の枝を調べ直す

a,b,c　　<namelist>



41

下降型構文解析の問題点
バックトラック(back tracking)

- バックトラック
  - 文法に一致しない場合後戻りする

例 : <E> ::= <T> "+" <E> | <T> "-" <E> | <T>
<T> ::= <F> "*" <T> | <F> "/" <T> | <F>
<F> ::= "a" | "b" | "c" | "d"

a*b - c*d　E→T+E
T+E→F*T+E
F*T+E→a*T+E
a*T+E→a*F+E
a*F+E→a*b+E　不一致

ここまで戻ってやり直し

42

左括り出し(left factoring)

■ バックトラックの対処

⇒バックトラックが起こらないように文法を変形

左括り出し

— 右辺の先頭の共通部分を括り出す

例 : <E> ::=<T>“+” <E> |<T>“-” <E> |<T>

⇩

<E> ::= <T> <E’>

<E’> ::= “+” <E> | “-” <E> | ε

43

左括り出し(left factoring)

■ バックトラックの対処

⇒バックトラックが起こらないように文法を変形

左括り出し

— 右辺の先頭の共通部分を括り出す

例 : <E> ::=<T>“+” <E> |<T>“-” <E> |<T>

⇩

<E> ::= <T> [“+” <E> | “-” <E>]

共通部分から後ろを括弧でまとめても良い

44

左括り出し

共通部分　　共通部分無し

<A> ::= $\alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

共通部分から後ろを表す新たな非終端記号を導入

<A> ::= α <A’> | $\gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

<A’> ::= $\beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

例 : <if_st> ::=“if” “(” <exp> “)” <st>

| “if” “(” <exp> “)” <st> “else” <st>

⇩ 左括り出し

<if_st> ::= “if” “(” <exp> “)” <st> <else>

<else> ::= ε | “else” <st>

45

左括り出し

共通部分　　共通部分無し

<A> ::= $\alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_m \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

共通部分から後ろを括弧でまとめてもよい

<A> ::= $\alpha (\beta_1 \mid \beta_2 \mid \ldots \mid \beta_m) \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$

例 : <if_st> ::=“if” “(” <exp> “)” <st>

| “if” “(” <exp> “)” <st> “else” <st>

⇩ 左括り出し

<if_st> ::= “if” “(” <exp> “)” <st> [“else” <st>]

46

左括り出し

例 : <E> ::= <T> “+” <E> | <T> “-” <E> | <T>

<T> ::= <F> “*” <T> | <F> “/” <T> | <F>

<F> ::= “a” | “b” | “c” | “d”

⇩

<E> ::= <T> <E’>

<E’> ::= “+” <E> | “-” <E> | ε

<T> ::= <F> <T’>

<T’> ::= “*” <T> | “/” <T> | ε

<F> ::= “a” | “b” | “c” | “d”

47

左括り出し

例 : <E> ::= <T> “+” <E> | <T> “-” <E> | <T>

<T> ::= <F> “*” <T> | <F> “/” <T> | <F>

<F> ::= “a” | “b” | “c” | “d”

⇩

<E> ::= <T> [(“+” | “-”) <E>]

<T> ::= <F> [(“*” | “/”) <T>]

<F> ::= “a” | “b” | “c” | “d”

48

左括り出し

例 : <E> ::= <T> <E'>
<E'> ::= "+" <E> | "-" <E> | ε
<T> ::= <F> <T'>
<T'> ::= "*" <T> | "/" <T> | ε
<F> ::= "a" | "b" | "c" | "d"

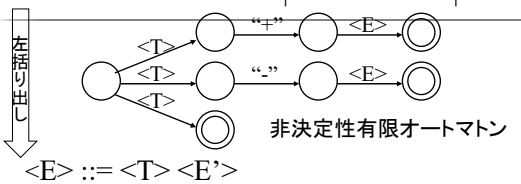
a*b - c*d　E→TE'
TE'→FT'E'
FT'E'→aT'E'
aT'E'→a*bE'
a*bE'→a*b-T
a*b-T→a*b-FT'

バックトラック無しに解析可能

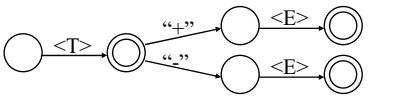
49

左括り出し

<E> ::= <T> "+" <E> | <T> "-" <E> | <T>



<E> ::= <T> <E'>
<E'> ::= "+" <E> | "-" <E> | ε



50

左括り出しとバックトラック

- バックトラックが無くせるとは限らない
  - 本質的に後戻りが必要な文法が存在

例 S→aBcd, B→bc | b

⇩ 左括り出し

S→aBcd, B→bB' B'→c | ε

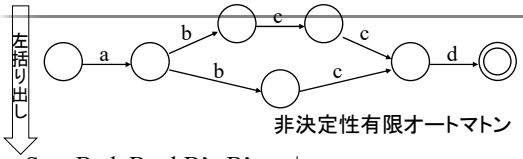
abcd　S→aBcd
aBcd→abB'cd
abB'cd→abccd

バックトラック
ボーー致

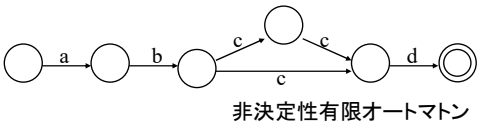
51

左括り出し

S→aBcd, B→bc | b



S→aBcd, B→bB' , B'→c | ε



左括り出しをしても NFA のまま

52

トークンの先読み

<E> ::= <T> { ("*" <T>) | ("/" <T>) }

- a*b
- a/b
- a

どれで解析する？

⇩

トークンを先読みする

- LL (k) 文法
  k 個先のトークンを先読みすれば解析可能な文法

53

LL(*k*) 文法

例 : <factor> ::= NAME "==" NAME
| NAME ">" NAME
| NAME "<" NAME

どのルールを適用する？

a == b

ここまで読めば
どれか判別できる

2つめのトークンまで読めば解析可能
⇒ LL(2) 文法

54

LL(*k*) 文法

例 : <unsigned> ::= NAME “[” <exp> “]”
| NAME “[” <exp> “]” “++”
| NAME “[” <exp> “]” “--”

a [1+2+3+4+5+6+7] ++

ここまで読めば
どれか判別できるが…

<exp>は無限にトークンが来る可能性あり

⇒ これは LL(*k*) 文法では無い

55

LL(*k*)文法⇒LL(1)文法

■ LL(*k*) 文法
⇒左括り出しで LL(1)文法に

例 : <factor> ::= NAME “==” NAME
| NAME “>” NAME
| NAME “<” NAME

⇩ 左括り出し

<factor> ::= NAME (“==” | “>” | “<”) NAME

56

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

57

LL(1)文法構文の解析

	先頭に来るトークン
<unsigned> ::= NAME	NAME
| INTEGER	INTEGER
| “(” <exp> “)”	“(”
| “input”	“input”

```
<unsigned>の解析() {
 swich (nextToken ) {
  case NAME :      { 変数の解析 }            break;
  case INTEGER :   { 整数の解析 }            break;
  case “(” :       {“(” <exp> “)” の解析 } break;
  case “input” :   { 入力の解析 }            break;
 }
}
```

次のトークンを見れば
どの処理をするか分かる

58

LL(1)文法構文の解析

	先頭に来るトークン
<st> ::= “{” { <st> } “}”	“{”
| “;”	“;”
| <while_st>	これらの先頭に来るトークンは？
| <if_st>	
| <write_st>	
| <exp_st>	

各非終端記号の First 集合 を求める

59

First 集合
(先頭終端記号集合)

■ First (α) = {“a” | α⇒aβ}
– ただし、α⇒εのときはεを含む
記号列 αの先頭に来る終端記号の集合

<if_st> ::= “if” “(” <expression> “)” <statement>
<while_st> ::= “while” “(” <expression> “)” <statement>

if 文の先頭のトークンは “if”
while 文の先頭のトークンは “while”

First (<if_st>) = { “if” }
First (<while_st> = { “while” }

60

First集合

例 : <A> ::= “a” | “b” | “c”
<B> ::= “dd” “ee” | “ff” “gg”
<C> ::= <A> | <B>
First (<A>) = { “a”, “b”, “c” }
First (<B>) = { “dd”, “ff” }
First (<C>) = { “a”, “b”, “c”, “dd”, “ff” }
First (<A>) First (<B>)

61

First 集合の求め方

- First (α) の求め方 (α∈(**N**∪**T**)*)
  - 初期状態では First (α) = φ(空集合)

1. α :: = ε のとき
   First (α) += ε;
2. α :: = “a” (∈**T**) のとき
   First (α) += “a”;
3. α ::= <A> (∈**N**) のとき
   First (α) += First (<A>);

<A> ∈**N**, “a”∈ **T**, α∈ (**N**∪**T**)*

62

First 集合の求め方

4. α ::= <X> β のとき
   1. ε∉ First (<X>) のとき
      First (α) += First (<X>);
   2. ε∈First (<X>) のとき
      First (α) += (First (<X>) -ε) + First (β);
5. α :: = β | γ のとき
   First (α) += First (β) + First (γ);

<X> ∈**N**, α, β, γ ∈ (**N** ∪ **T**)*

63

First 集合の求め方

6. α ::= {β} のとき
   First (A) += First (β) + ε;
7. α ::= [β] のとき
   First (A) += First (β) + ε;
8. α ::= (β) のとき
   First (A) += First (β);

α, β∈(**N** ∪ **T**)*

64

First集合の求め方
4. ε-ルール

4. α ::= <X> βのとき
1. ε∉ First(<X>) のとき First (α) += First (<X>)
2. ε∈First(<X>) のとき First (α) += (First (<X>) -ε) + First (β);

例 : <A> ::= <B> “a”
<B> ::= “b” | ε

A ⇒ ba , A⇒εa = a

First (<A>) = {“b”, “a”}
First (<B>) - ε　First (“a”)

65

First 集合

例 : <S> ::= “a” | <B> <C>
<B> ::= “b” | ε
<C> ::= “c”

First (ε) = {ε} (ルール1.)
First (“a”) = {“a”} (ルール2.)
First (“b”) = {“b”} (ルール2.)
First (“c”) = {“c”} (ルール2.)
First (<C>) = First (“c”) = {“c”} (ルール2.)
First (<B>) = First (“b”)+First (ε) = {“b”, ε} (ルール5.)
First (<B><C>) = (First (<B>)-ε)+First (<C>)
= {“b”, “c”} (ルール4-2.)
First (<S>) = First (“a”) + First (<B><C>)
= {“a”, “b”, “c”} (ルール5.)

66

First 集合を用いた構文解析

nextToken と First 集合を比較

```
<st>の解析() {
  if (nextToken ∈ First (<if_st>)) {
      <if_st> の解析;
  } else if (nextToken ∈ First (<while_st>)) {
      <while_st> の解析;
  } else if (nextToken == "{" ) {
      "{" { <st> } "}" の解析;
  } else if (nextToken == ";") {
      ";" の解析;
  } else syntaxError();
}
```

67

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

68