言語処理系論

小林　直樹

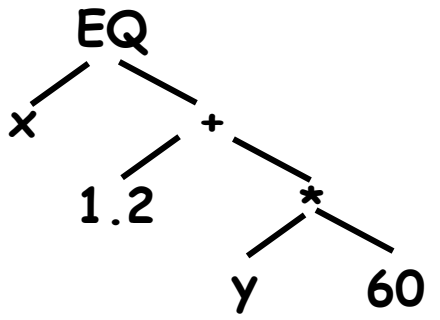
コンパイラの構造

x = 1.2+y*60

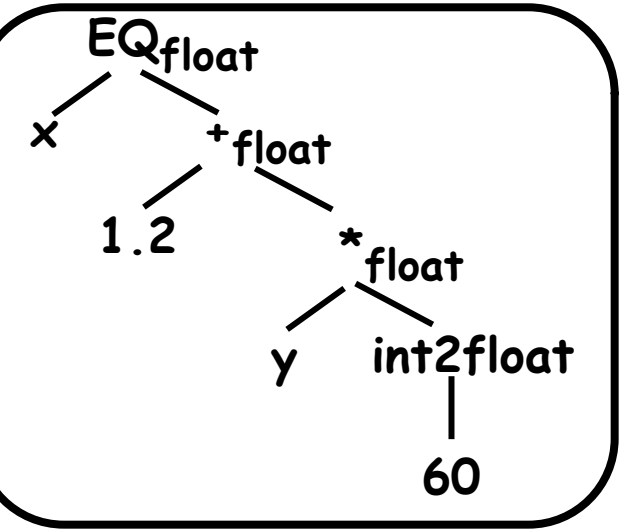
字句解析

ID(“x”) EQ Float(1.2)
PLUS ID(“y”) TIMES Int(60)

構文解析



意味解析



中間コード生成

```
t1 = int2float(60)
t2 = y * t1
t3 = 1.2 + t2
x = t3
```

最適化

```
t2 = y * 60.0
x = 1.2 + t2
```

機械語コード生成

```
FR1 <- y
FR1 <- FR1 * 60.0
FR1 <- FR1 + 1.2
x <- FR1
```

本日の内容

- 字句解析
  - 文字列をトークンの列に変換

アウトライン

- **基本原則**（復習）
- 字句解析の実際
- 具体例
- 字句解析生成器lex

字句解析器の構成の原則

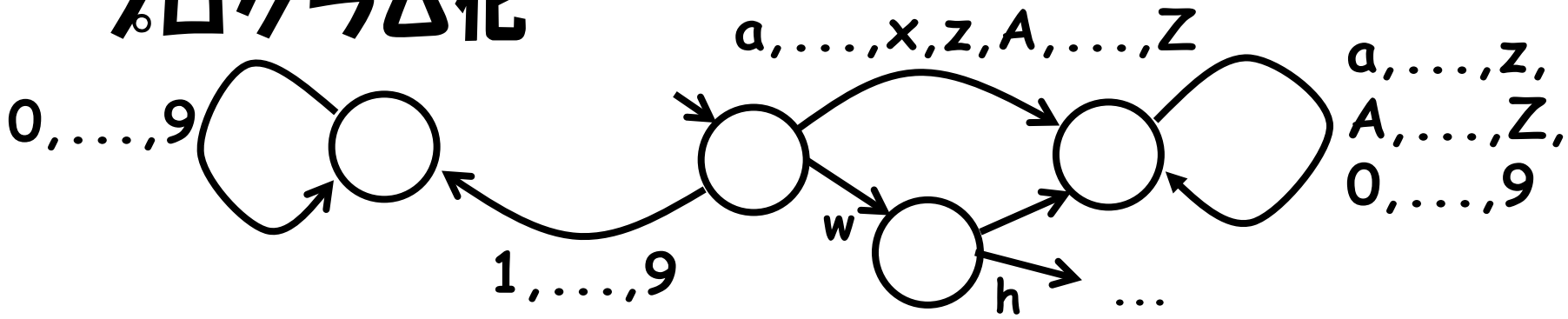
- トークンの仕様を正規表現で記述

  WHILE: “while”

  ID: (a|...|z|A|...|Z)(a|...|z|A|...|Z|0|..|9)*

  NUM: (1|...|9)(0|...|9)*

- トークンに相当する文字列（lexeme, 語彙素）を受理するオートマトンを構築、プログラム化



アウトライン

- 基本原則
- 字句解析の実際（原則の例外と対処法）
- 具体例
- 字句解析生成器lex

原則の例外

(1) 文字列を受理して終わりではなく、
トークンとその属性を出力する必要

- 例：”234”

  オートマトン: “yes”

  字句解析器: Int(234)

(2)一つのトークンを認識したら次のトークンの
認識へ

- 例：”123>2” → Int(123)　GT　Int(2)

原則の例外

(3) 曖昧性

例： EQ: =　　LT: <　　LEQ: <=　　IF: if

INT: [1-9][0-9]*　ID:　[a-z]([a-z]|[0-9])*

- "<=" は LEQか、LT　EQか？
- "if123" はID("if123")か、IF INT(123)か？
- "if" は IFか、ID("if")か？

曖昧性解消の約束事

1. longest match: 複数の部分文字列がトークン定義にマッチする場合、それらの中で最長のものを選ぶ。

原則の例外

(3) 曖昧性

例： EQ: =　　LT: <　　LEQ: <=　　IF: if

INT: [1-9][0-9]*　ID:　[a-z]([a-z]|[0-9])*

- "<=" は LEQか、LT　EQか？
- "if123" はID("if123")か、IF INT(123)か？
- "if" は IFか、ID("if")か？

曖昧性解消の約束事

1. longest match: 複数の部分文字列がトークン定義にマッチする場合、それらの中で最長のものを選ぶ。
2. first match: 最長のマッチが複数ある場合、先に定義されているトークンを選択

原則の例外

(3) 曖昧性

例： EQ: =　　LT: <　　LEQ: <=　　IF: if

INT: [1-9][0-9]*　ID:　[a-z]([a-z]|[0-9])*

- "<=" は LEQか、LT　EQか？
- "if123" はID("if123")か、IF INT(123)か？
- "if" は IFか、ID("if")か？

曖昧性解消の約束事

1. longest match: 複数の部分文字列がトークン定義にマッチする場合、それらの中で最長のものを選ぶ。
2. first match: 最長のマッチが複数ある場合、先に定義されているトークンを選択

原則の例外

(4) **正規表現で表現できないものあり**

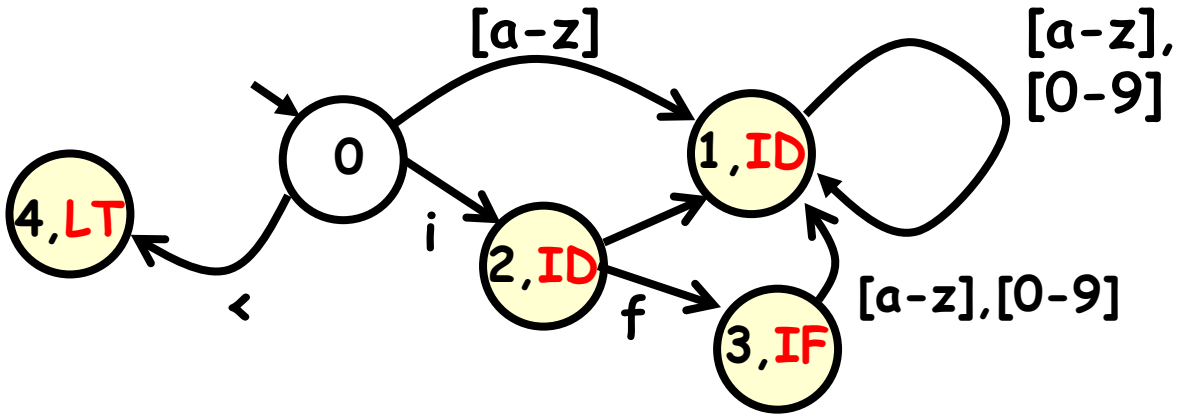
例：nested comments

(* これは (* 正しいコメント *) です *)

(* これは (* コメントが閉じていません *)

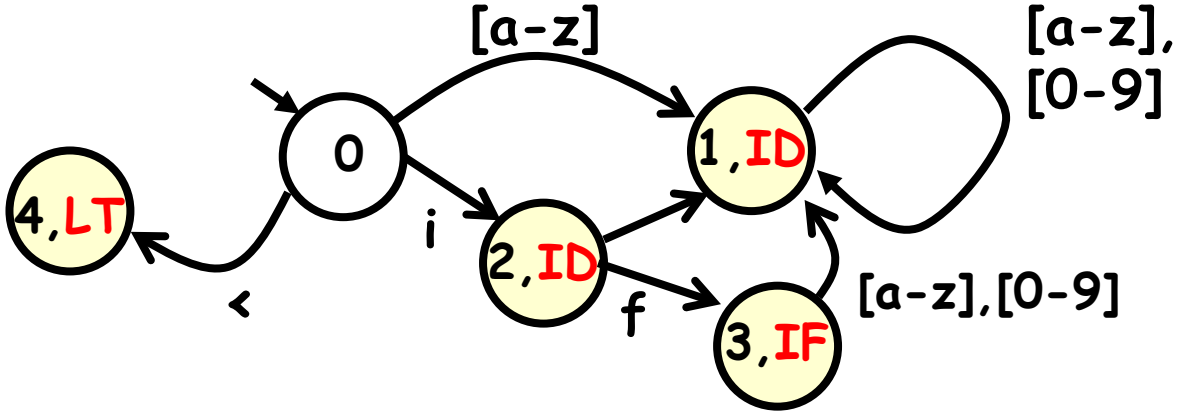
対処法
(1. トークンとその属性の出力)

- 「オートマトン」（トランスデューサ）の受理状態に、受理するトークンの種類を付加



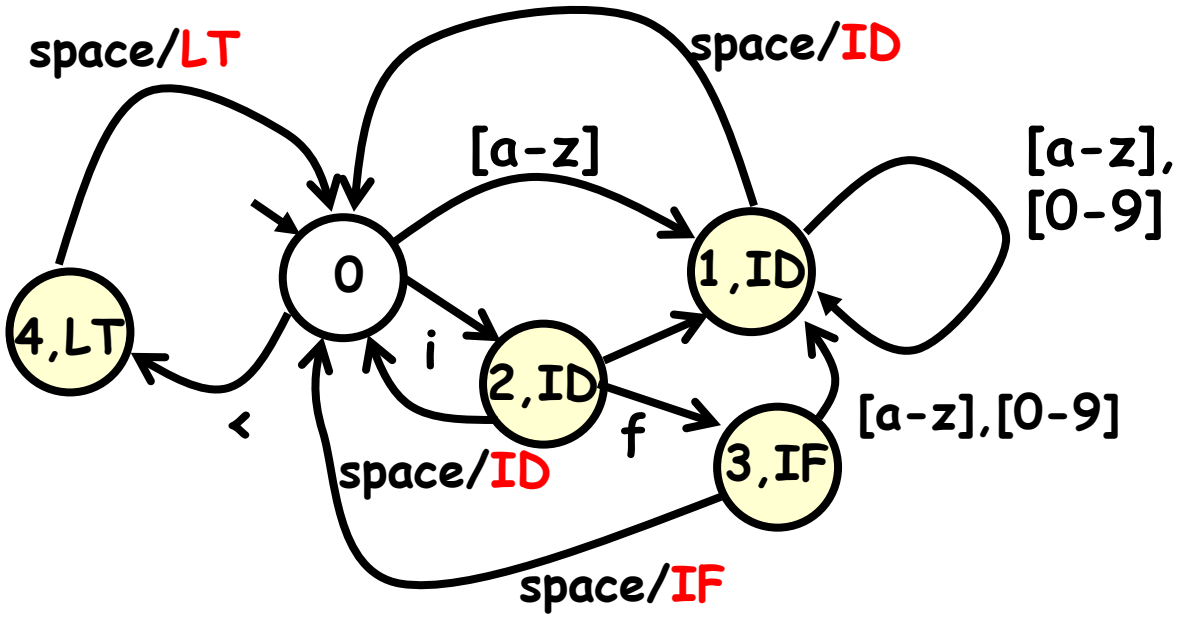
対処法
(1. トークンとその属性の出力)

- 「オートマトン」（トランスデューサ）の受理状態に受理するトークンの種類を付加
- トークンの区切り記号（スペースなど）を読んだらトークンの属性を出力



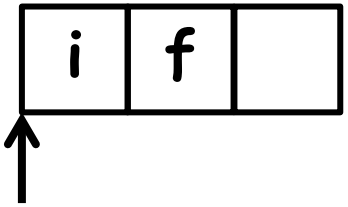
対処法
(1. トークンとその属性の出力)

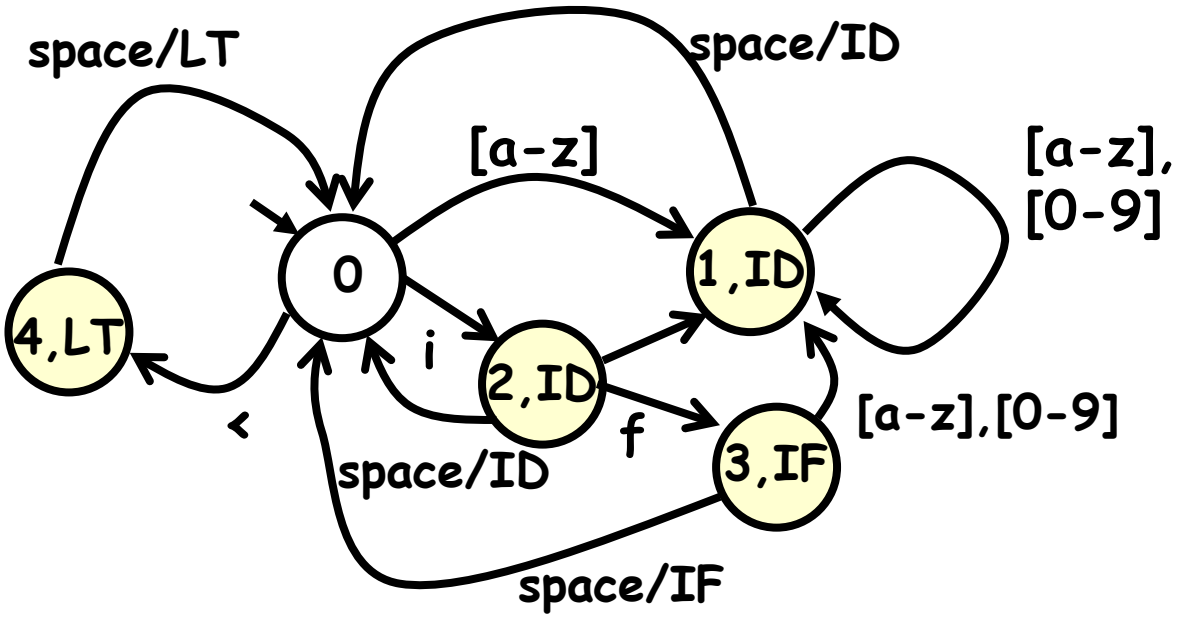
- 「オートマトン」（トランスデューサ）の受理状態に受理するトークンの種類を付加
- トークンの区切り記号（スペースなど）を読んだらトークンの属性を出力



対処法
(1. トークンとその属性の出力)

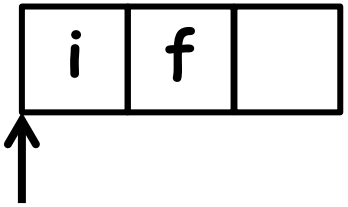
入力例：

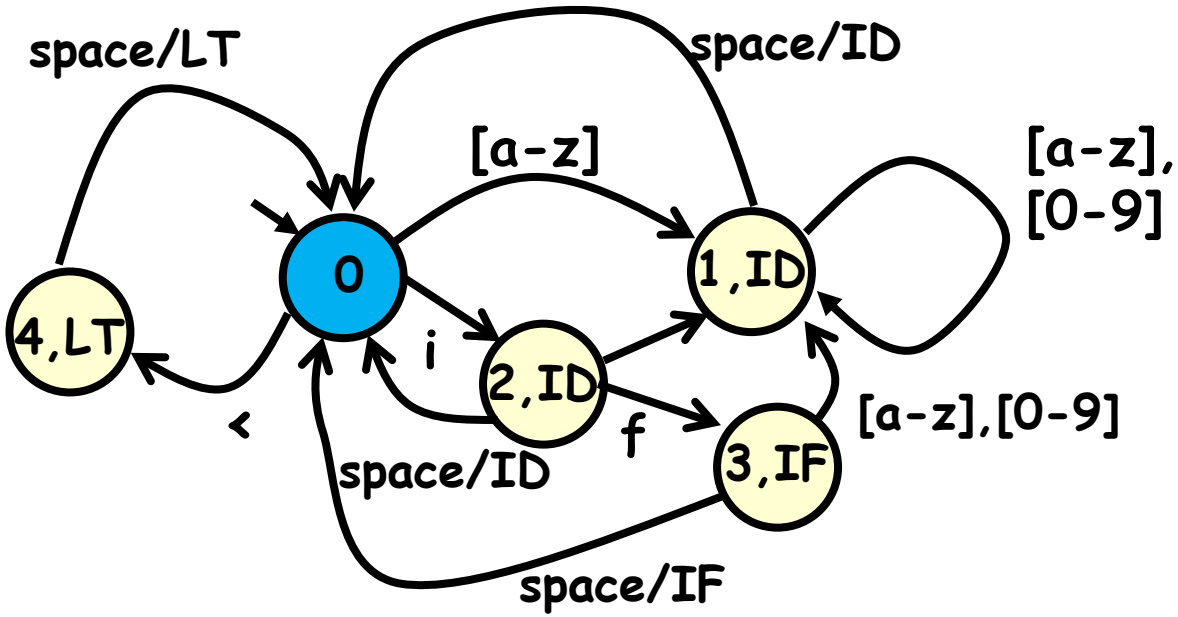




対処法
(1. トークンとその属性の出力)

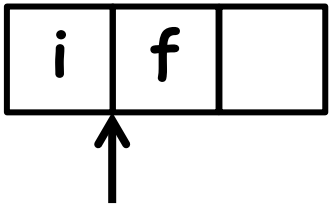
入力例：

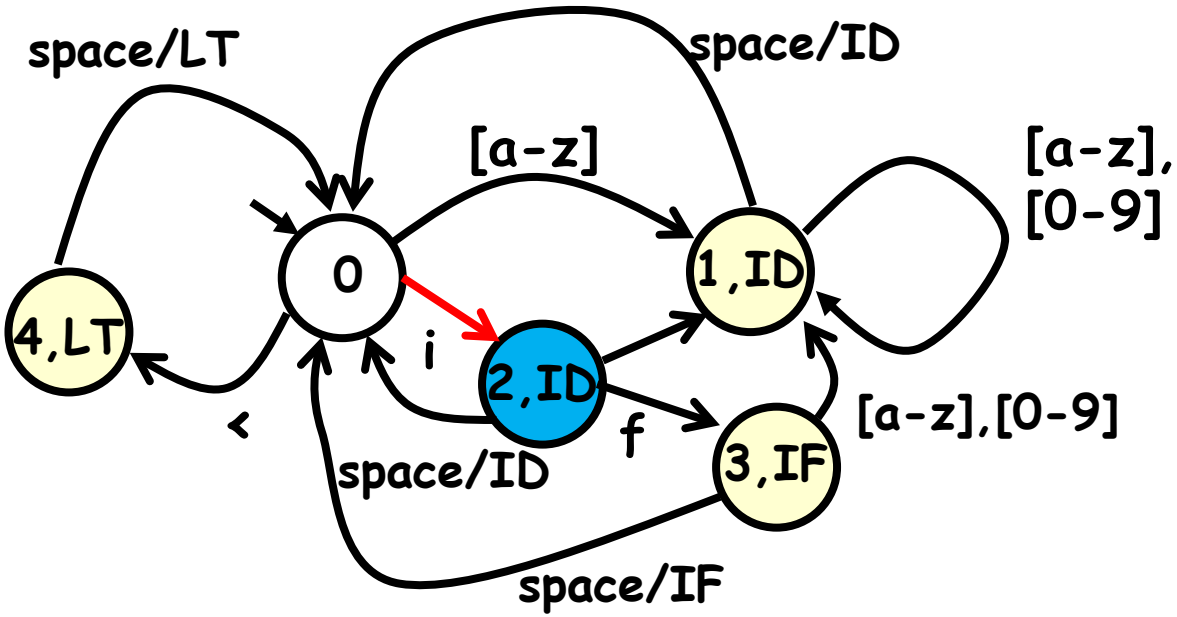




対処法
(1. トークンとその属性の出力)

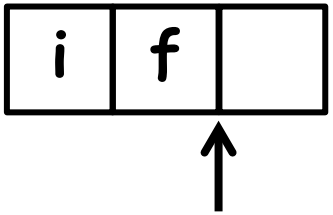
入力例：

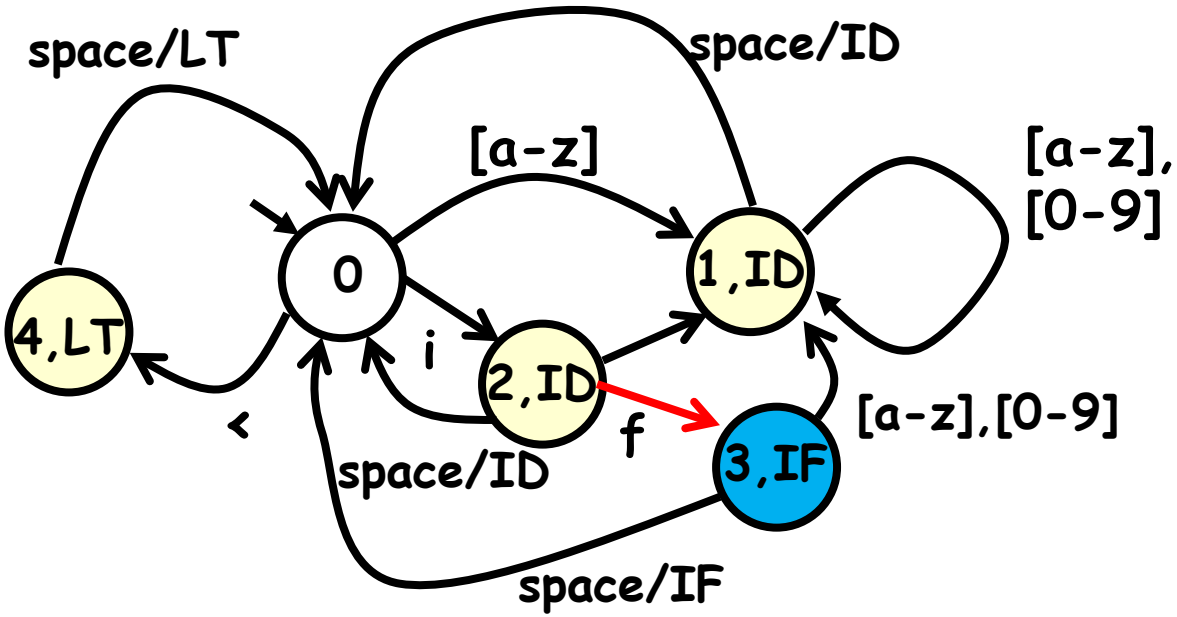




対処法
(1. トークンとその属性の出力)

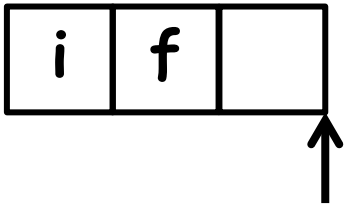
入力例：

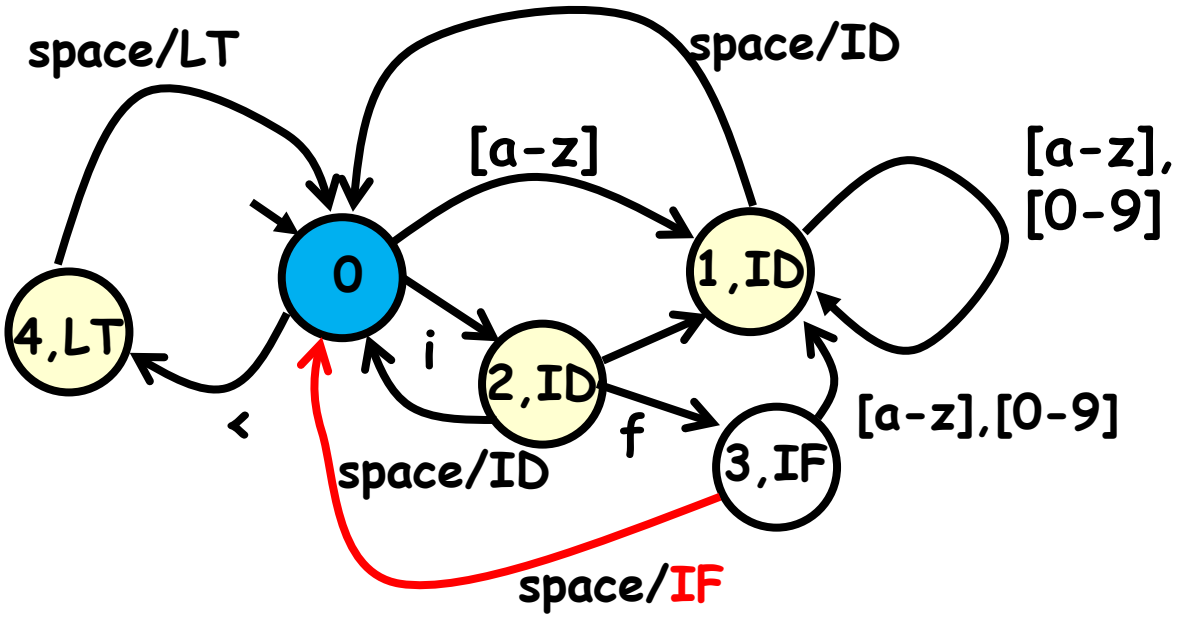




対処法
(1. トークンとその属性の出力)

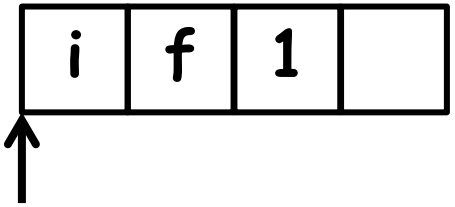
入力例：

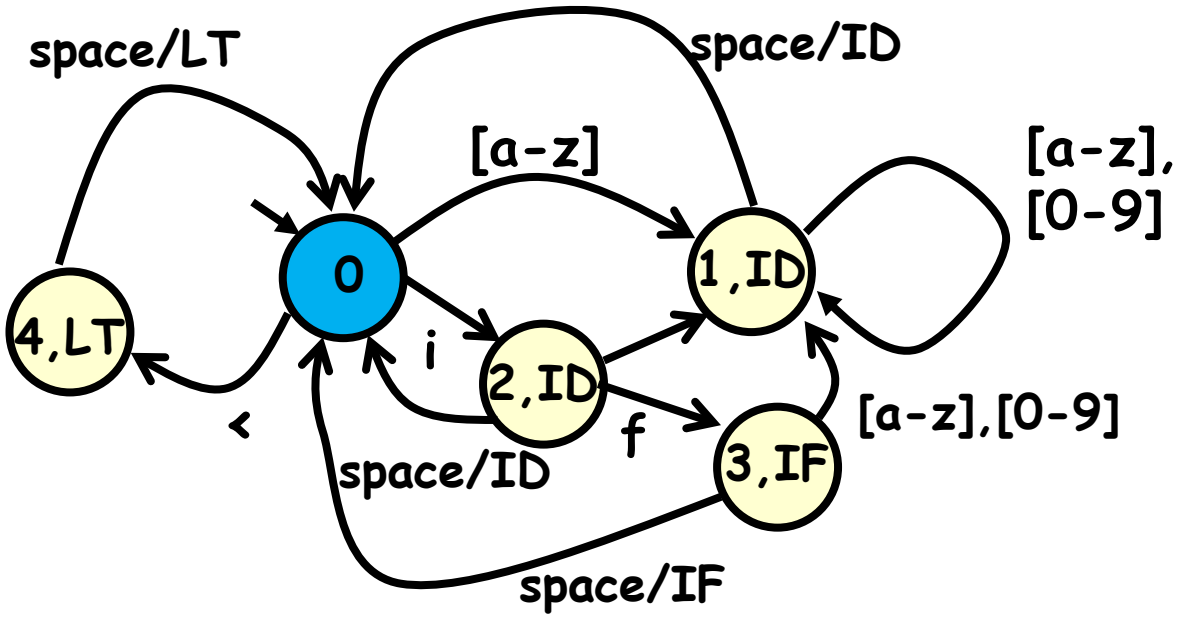




対処法
(1. トークンとその属性の出力)

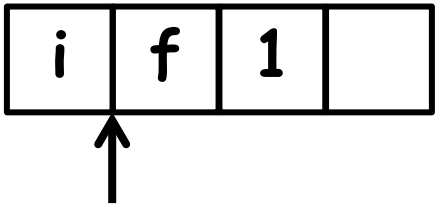
入力例：

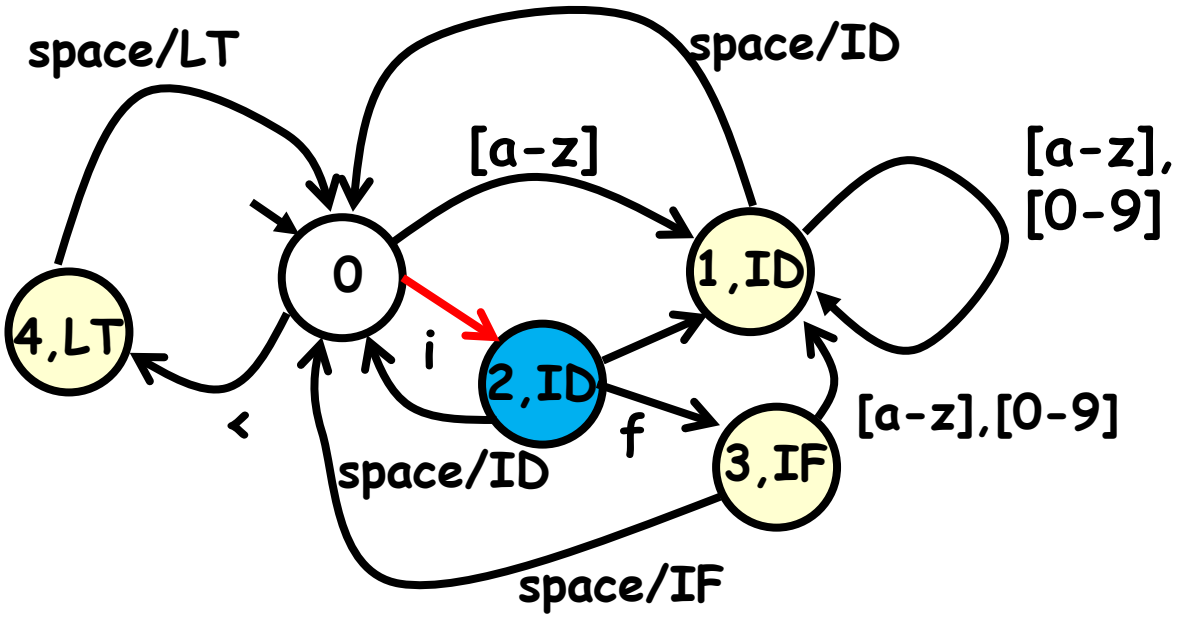




対処法
(1. トークンとその属性の出力)

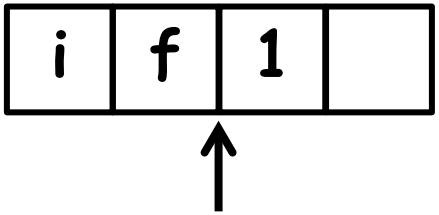
入力例：

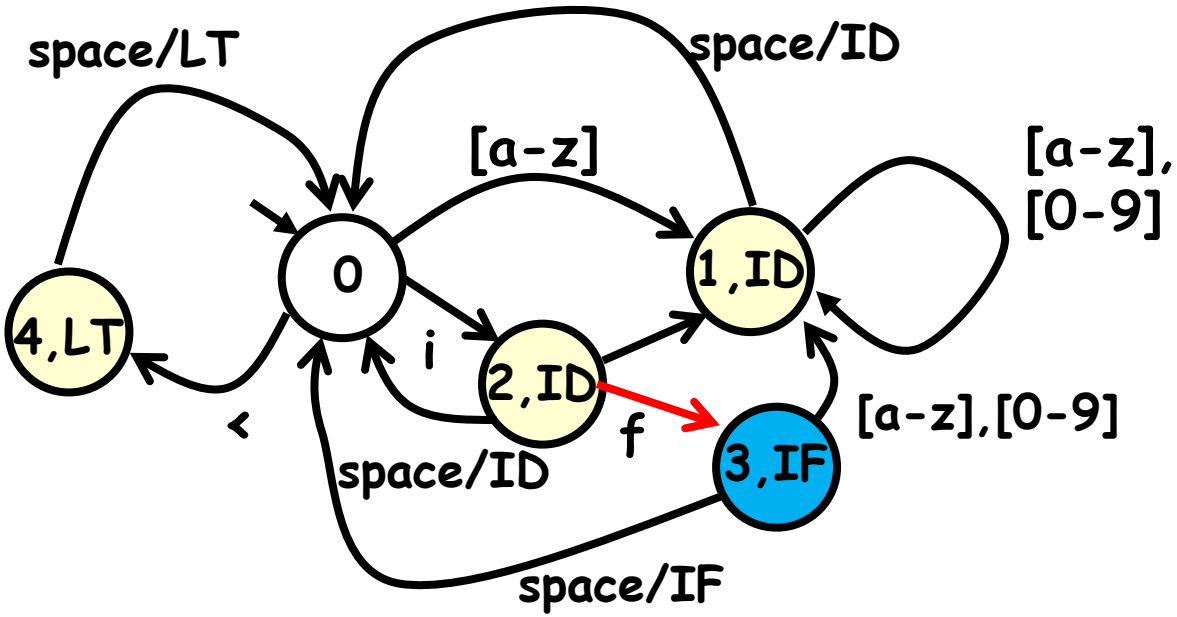




対処法
(1. トークンとその属性の出力)

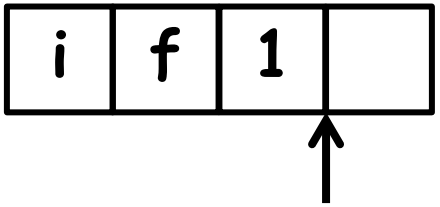
入力例：

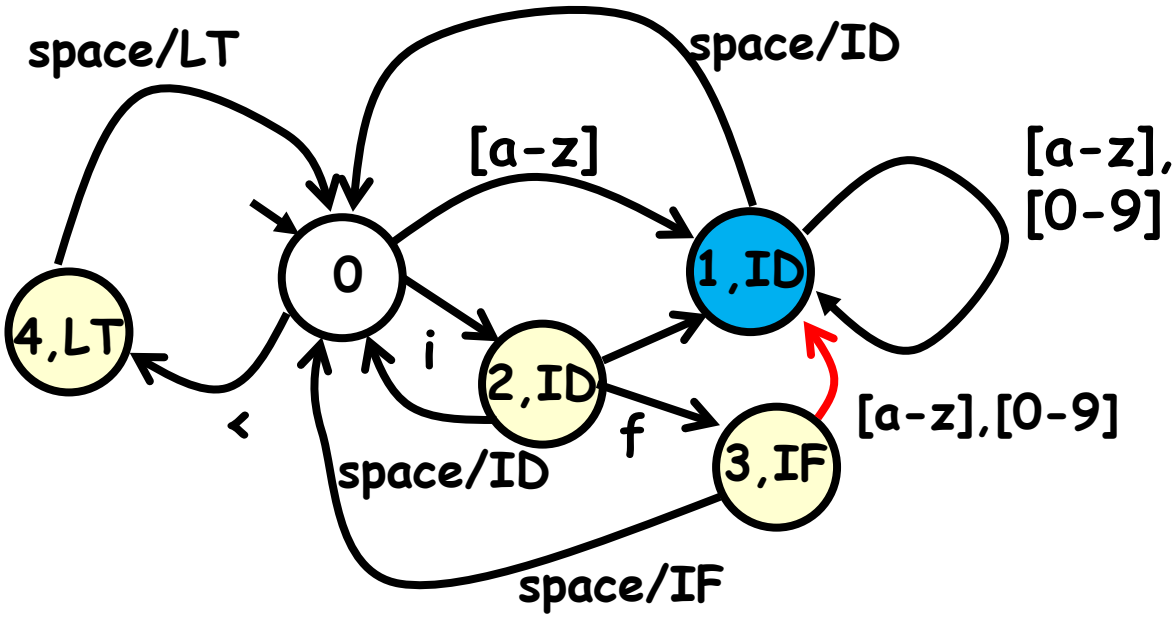




対処法
(1. トークンとその属性の出力)

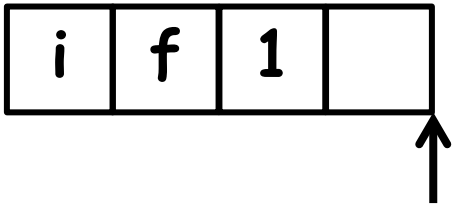
入力例：

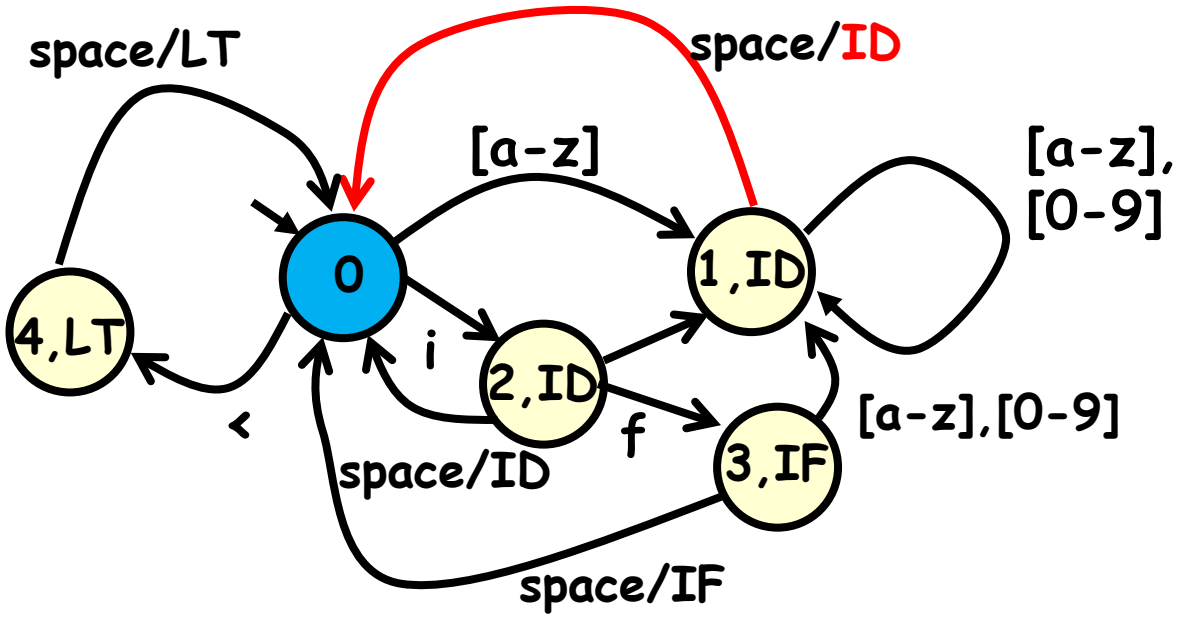




対処法
(1. トークンとその属性の出力)

入力例：





原則の例外

(1) 文字列を受理して終わりではなく、
トークンとその属性を出力する必要

- 例：”234”

  オートマトン: “yes”

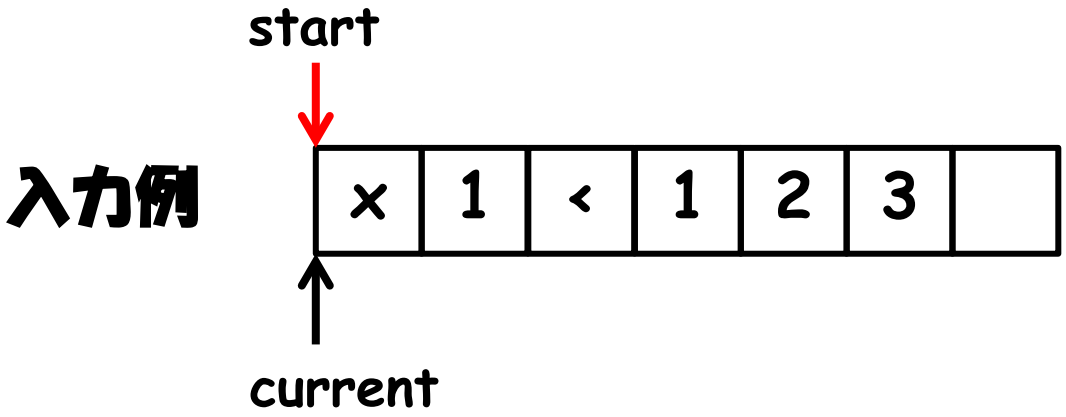
  字句解析器: Int(234)

(2)一つのトークンを認識したら次のトークンの
認識へ

- 例：”123>2” → Int(123)　GT　Int(2)

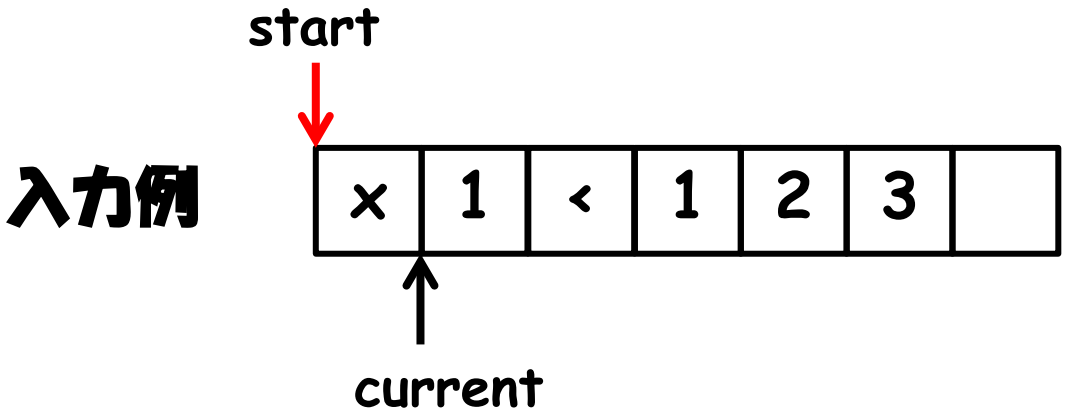
対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート



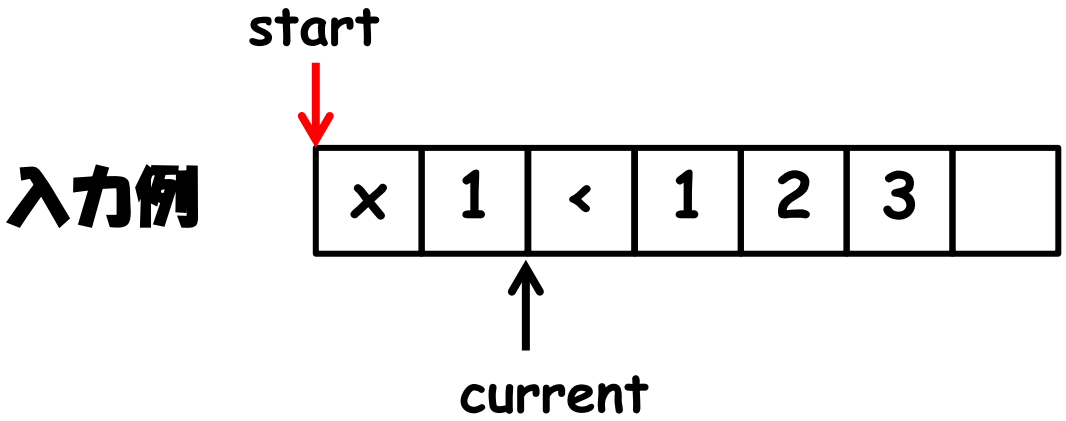
対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート



対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

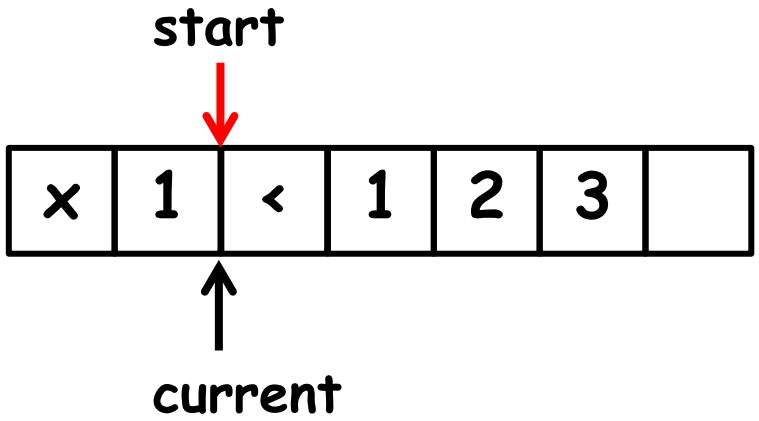


ID("x1")を出力

対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

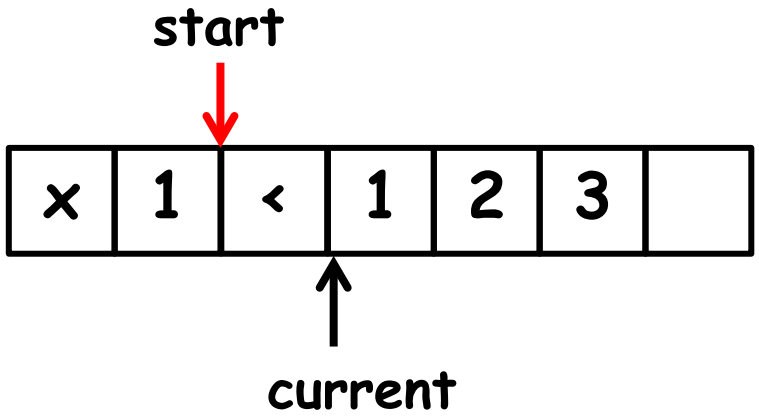
入力例



対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

入力例

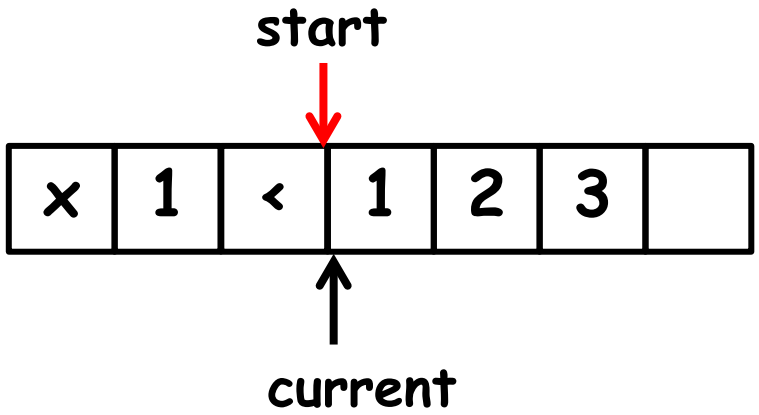


LTを出力

対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

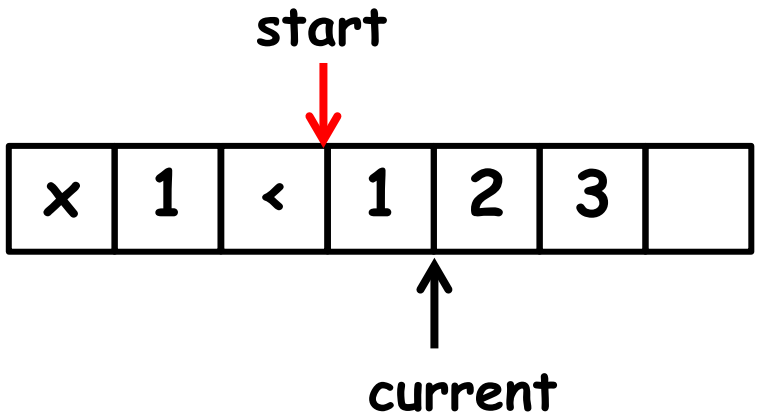
入力例



対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

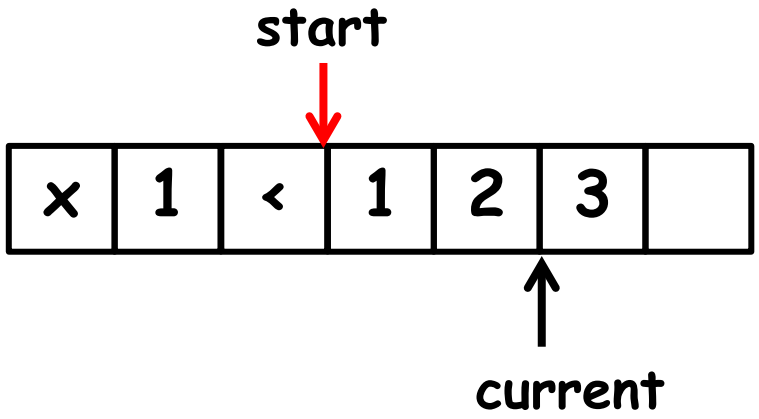
入力例



対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

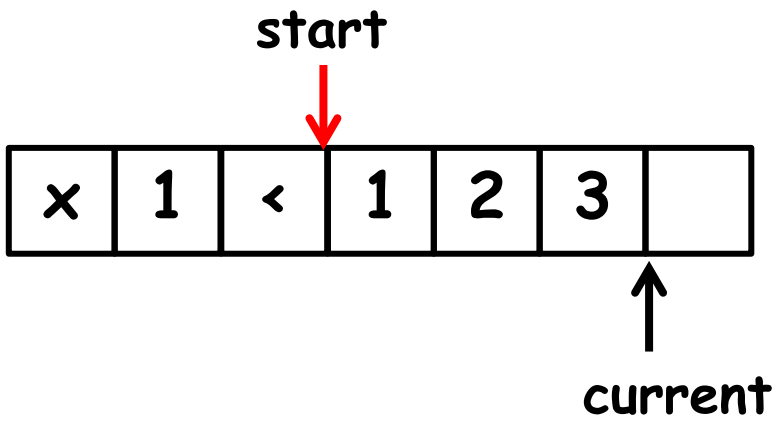
入力例



対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

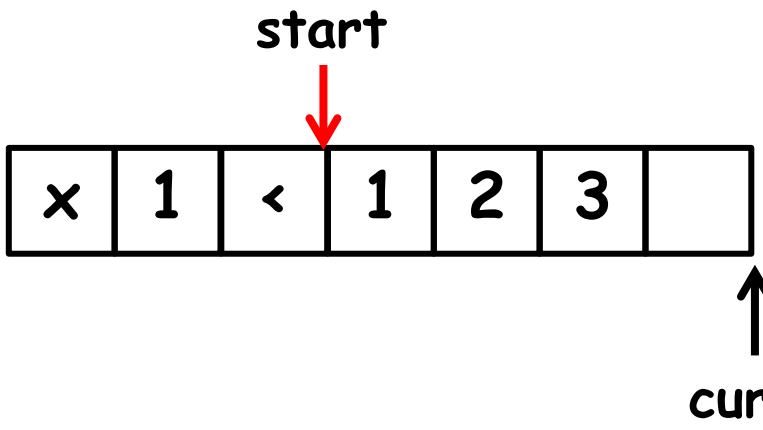
入力例



対処法
(2. トークンの列の認識)

- 現在読んでいるトークンの開始位置を状態として保持
- トークンを読み終えたらアップデート

入力例



Int(123)を出力

原則の例外

(3) 曖昧性

例： EQ: =　　LT: <　　LEQ: <=　　IF: if

INT: [1-9][0-9]*　ID:　[a-z]([a-z]|[0-9])*

- "<=" は LEQか、LT　EQか？
- "if123" はID("if123")か、IF INT(123)か？
- "if" は IFか、ID("if")か？

曖昧性解消の約束事

1. longest match: 複数の部分文字列がトークン定義にマッチする場合、それらの中で最長のものを選ぶ。
2. first match: 最長のマッチが複数ある場合、先に定義されているトークンを選択

対処法
(3. longest/first match)

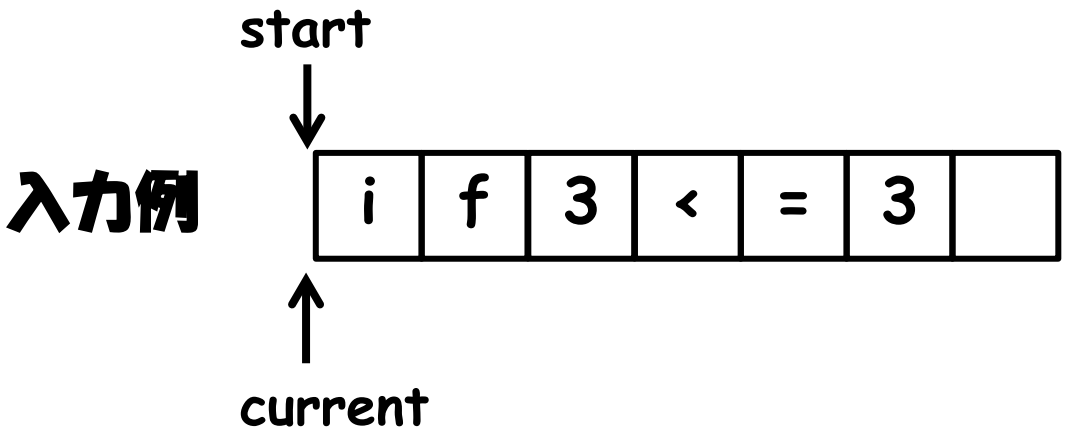
- “first match”の対処
  - 正規表現の解釈時に対処

    IF: “if”　　ID:　[a-z]([a-z]|[0-9])*

  → IF: “if”　　ID:　[a-z]([a-z]|[0-9])*＼”if”
- “longest match”の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に、記憶したトークンを出力

対処法
(3. longest/first match)

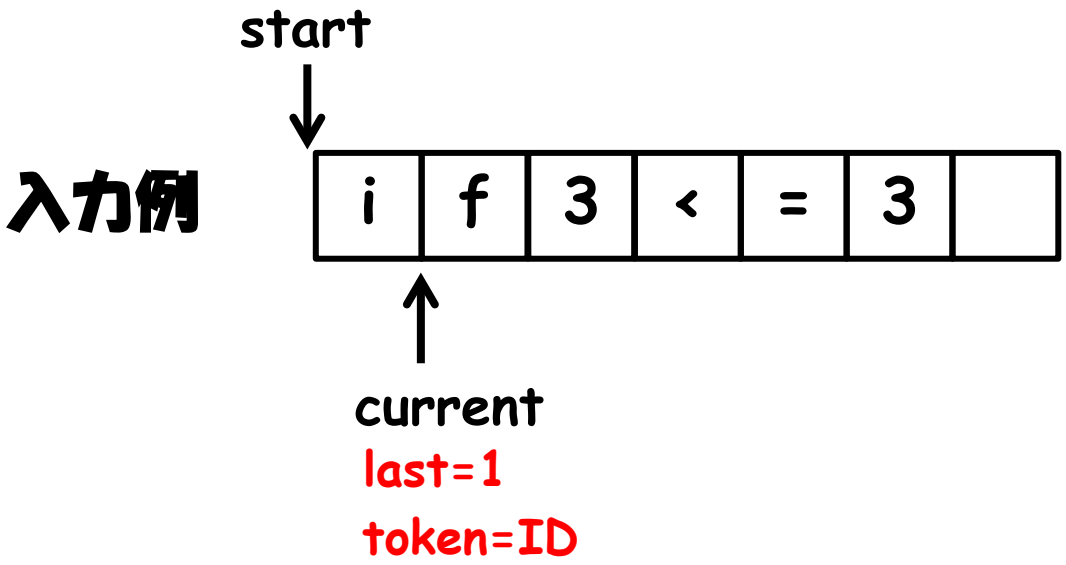
- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力




```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力



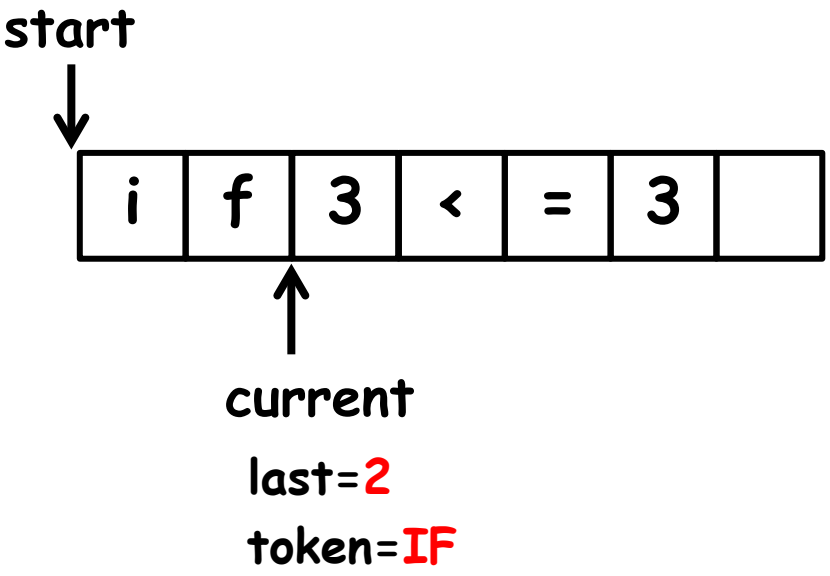

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

対処法
(3. longest/first match)

- “longest match”の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

入力例

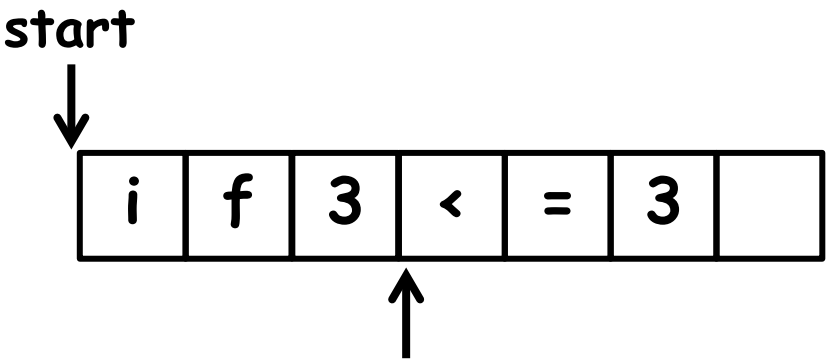


対処法
(3. longest/first match)

- “longest match”の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

入力例



last=3

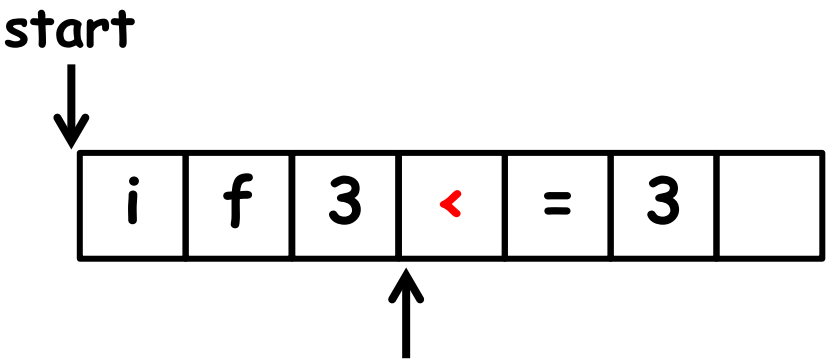
token=ID

対処法
(3. longest/first match)

- “longest match”の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

入力例



last=3

token=ID

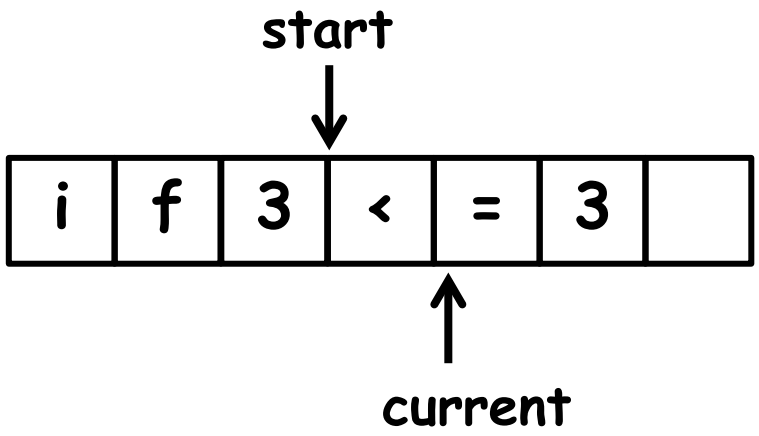
ID(“if3”)を出力

対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

入力例



last=4

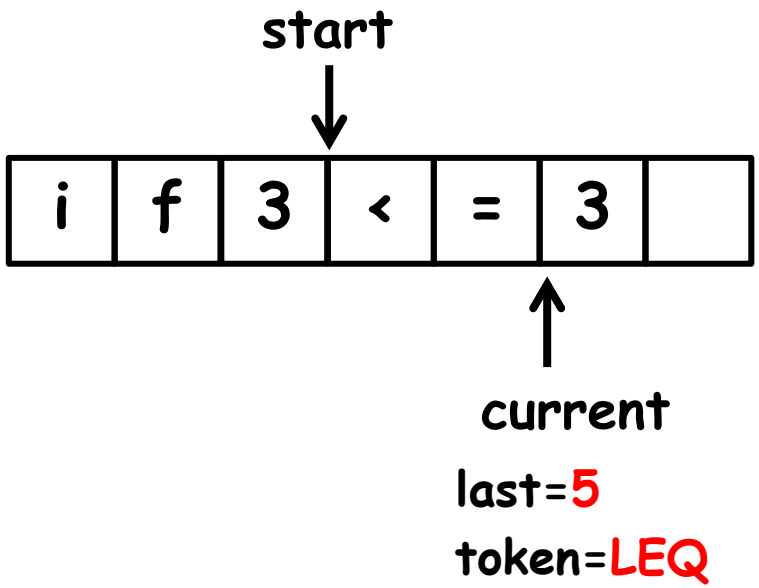
token=LT

対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

入力例

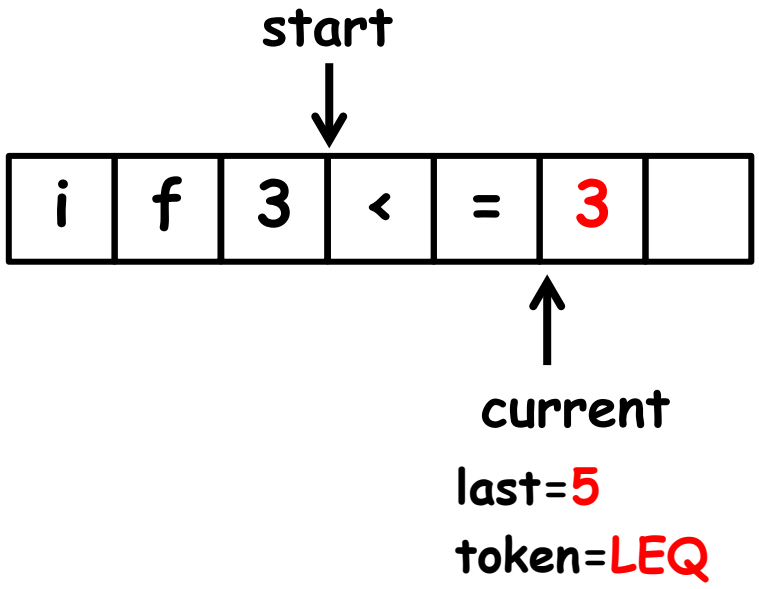


対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

```
LT: <
LEQ: <=
IF: if
INT: [1-9][0-9]*
ID:  [a-z]([a-z]|[0-9])*
```

入力例

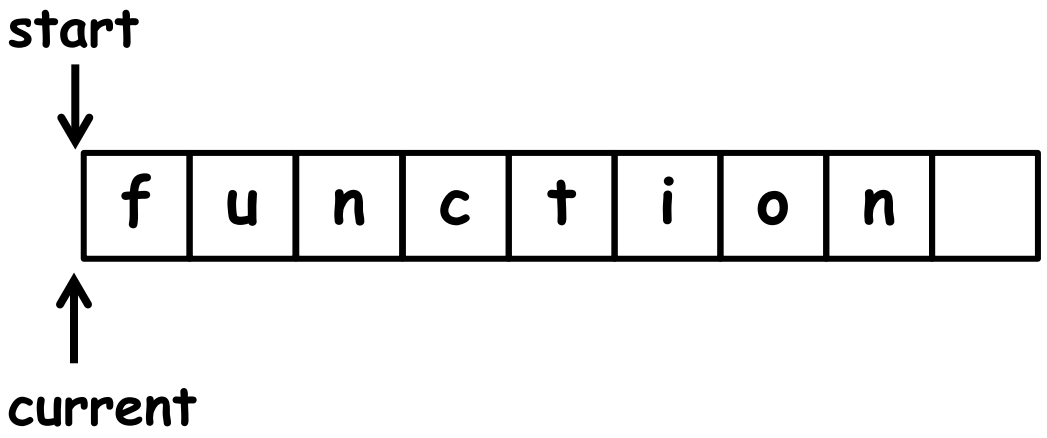


LEQを出力

対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

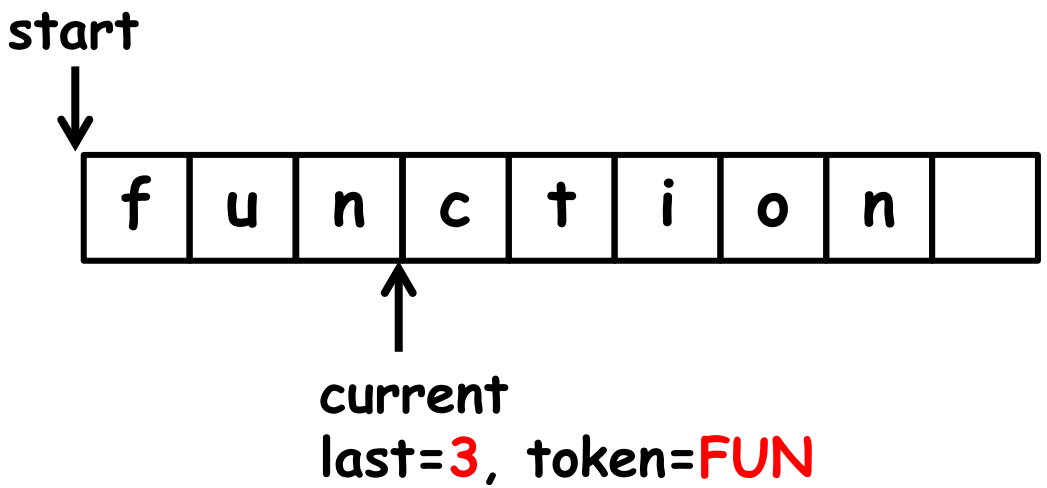
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

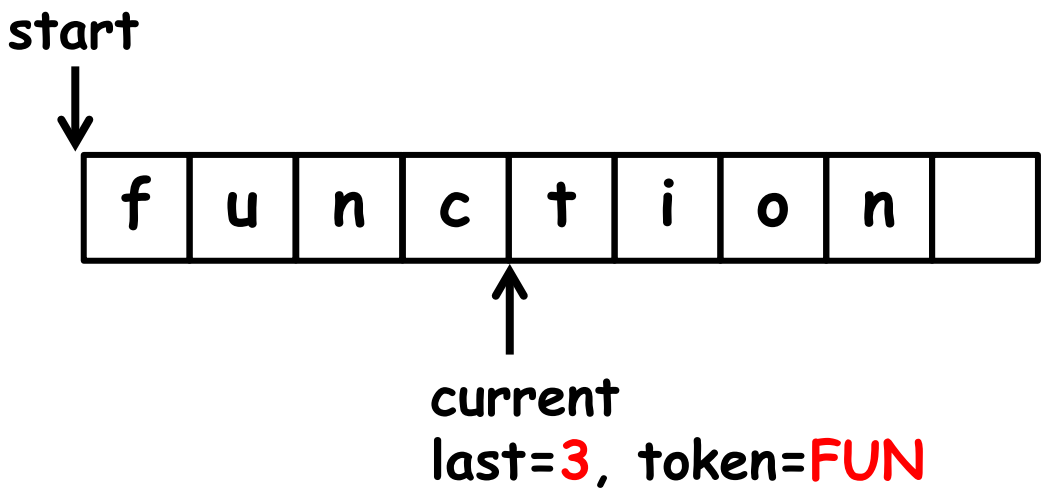
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

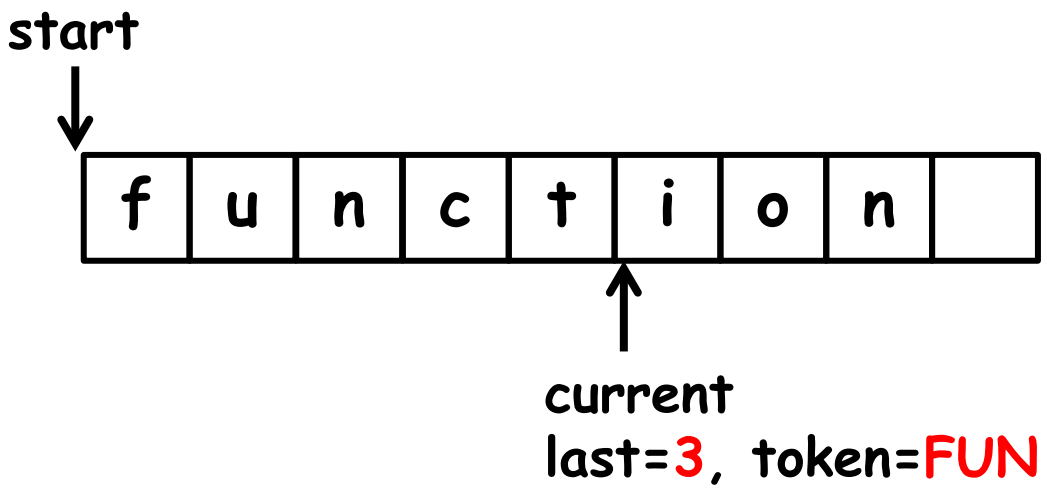
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

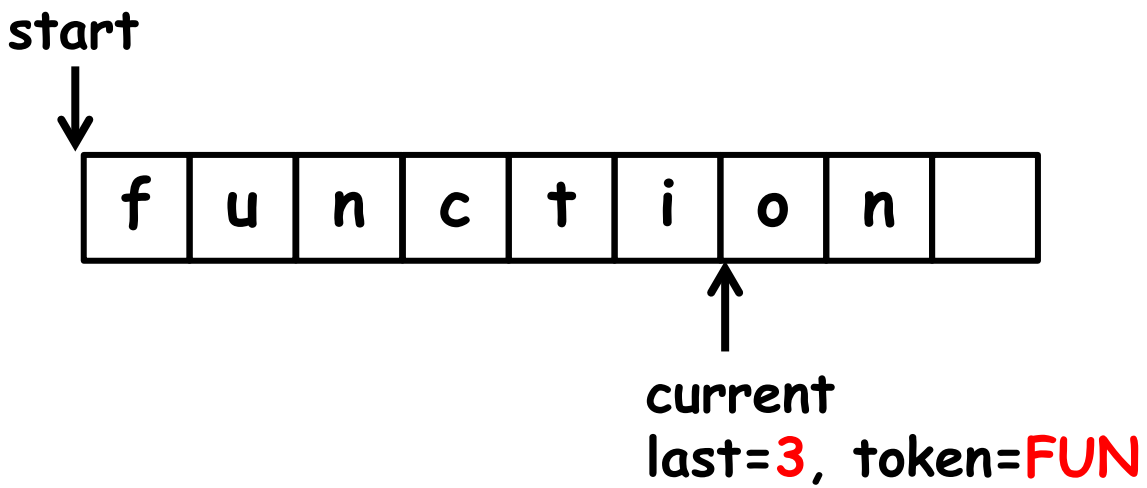
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- “longest match”の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

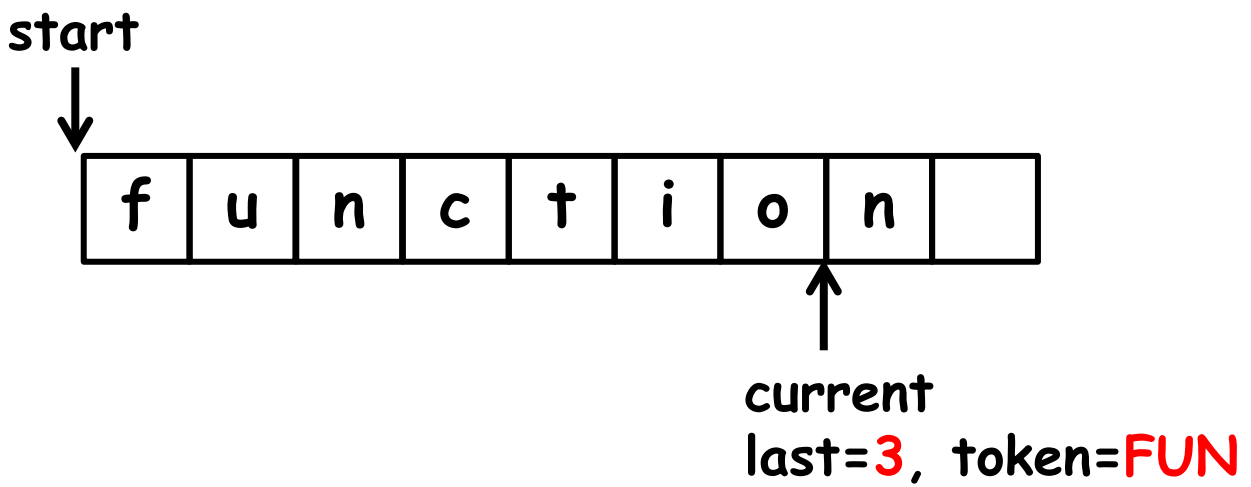
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

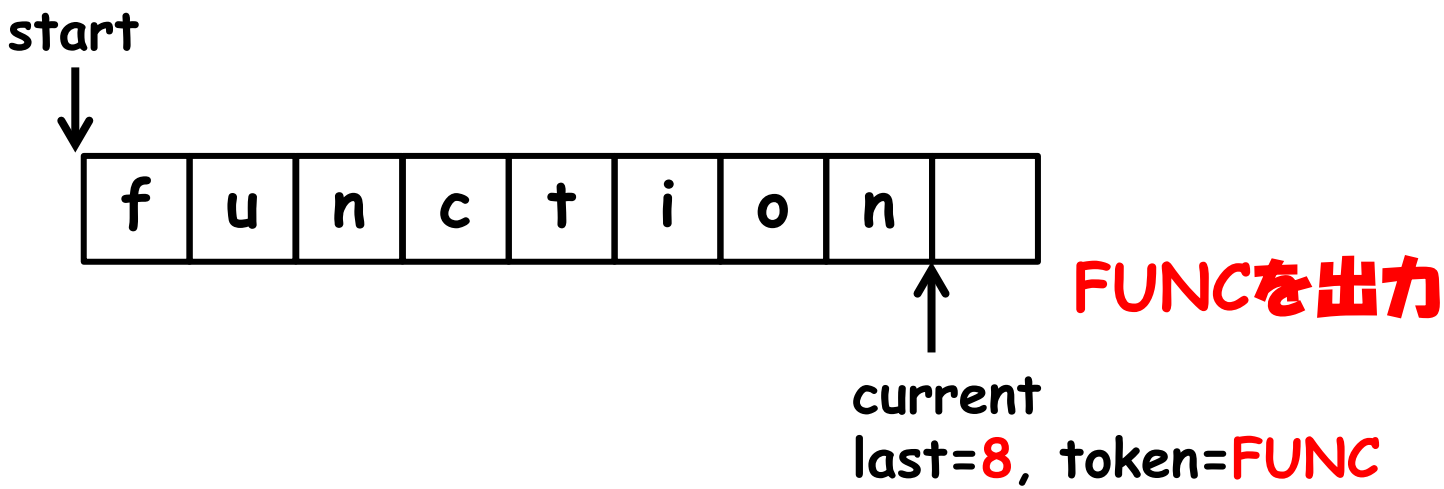
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

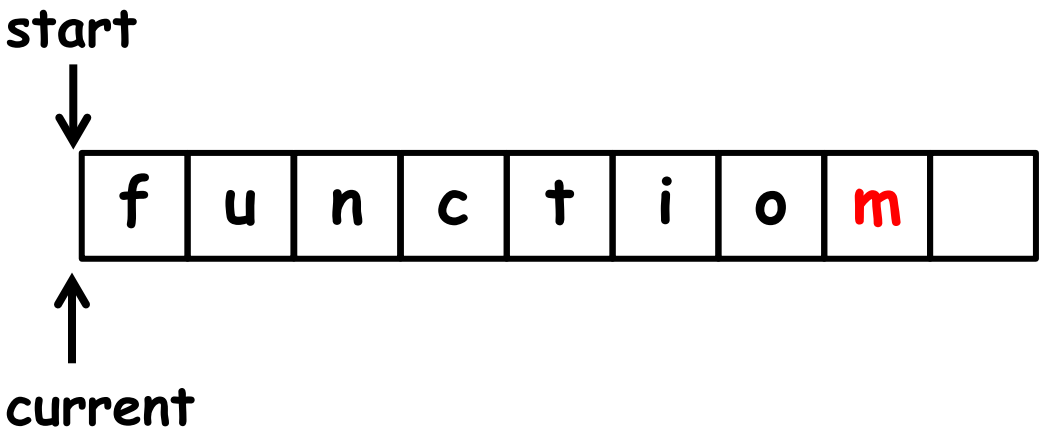
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

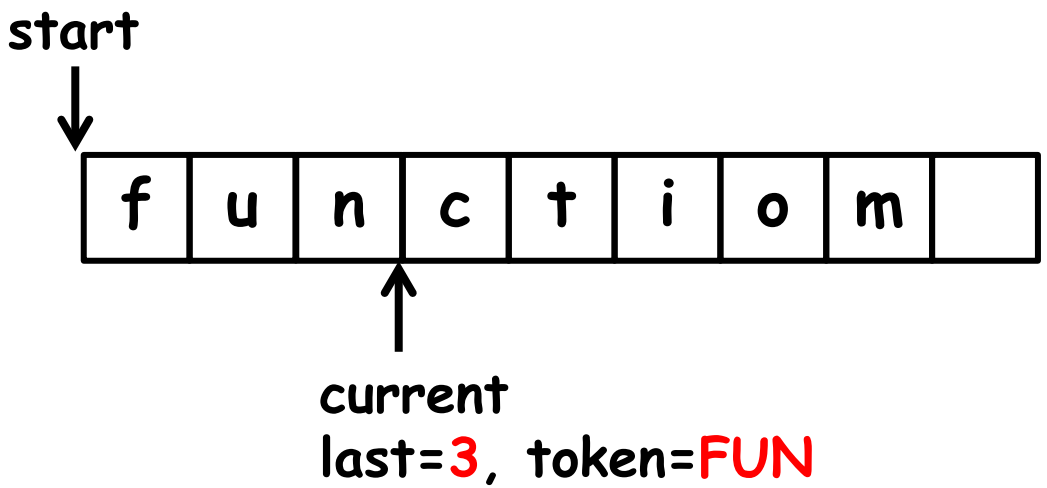
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

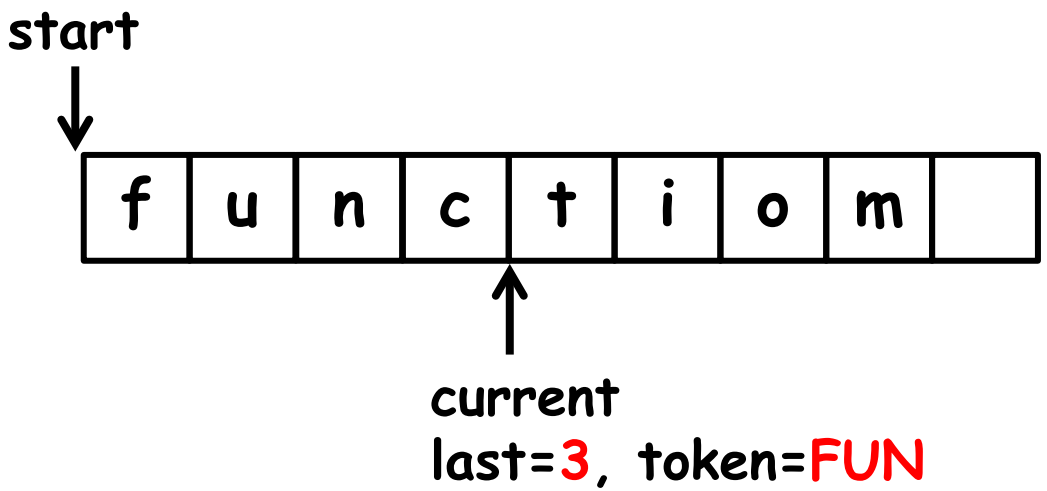
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

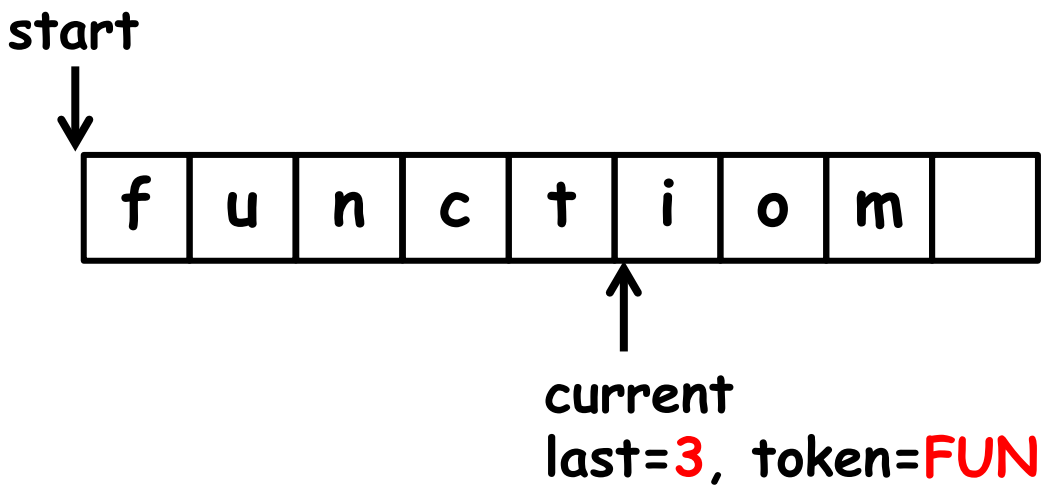
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

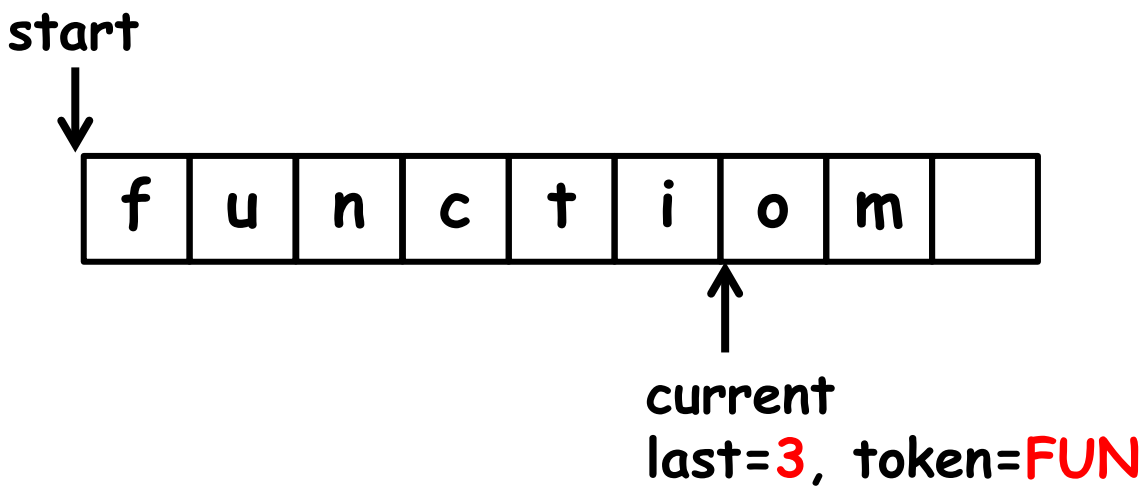
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

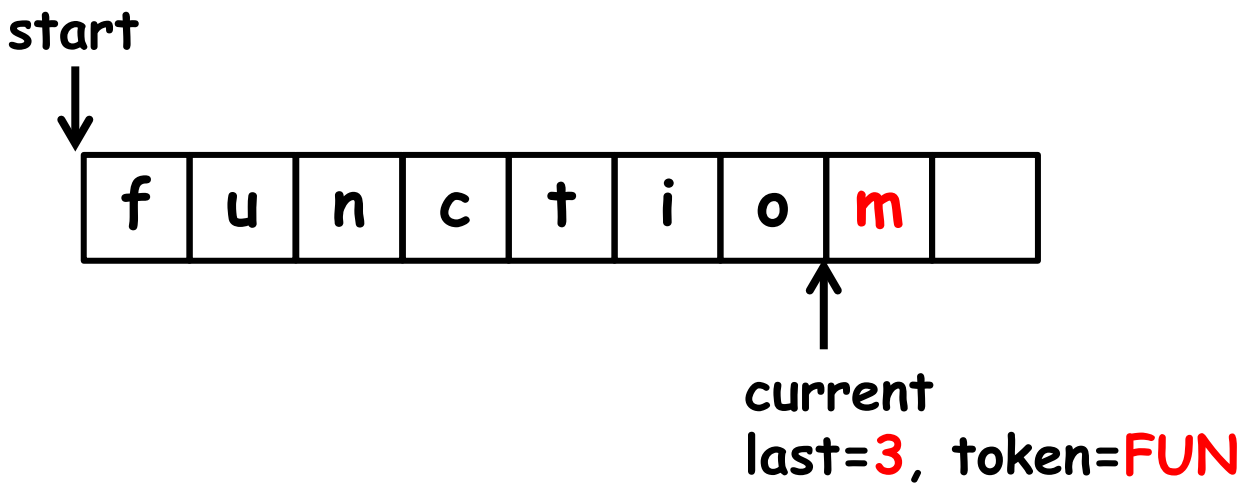
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

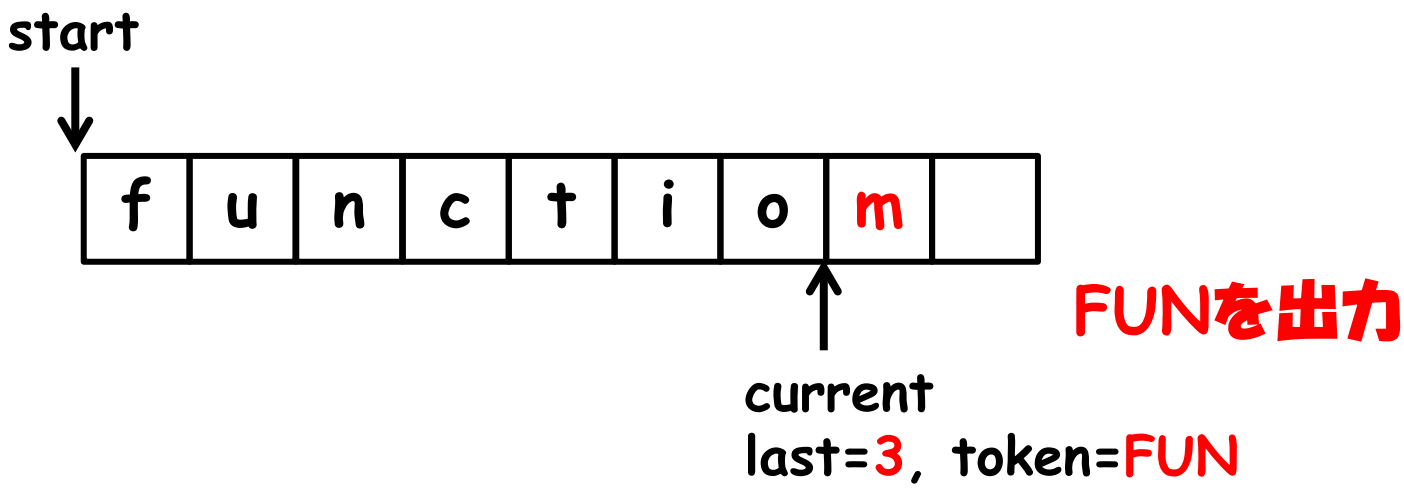
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

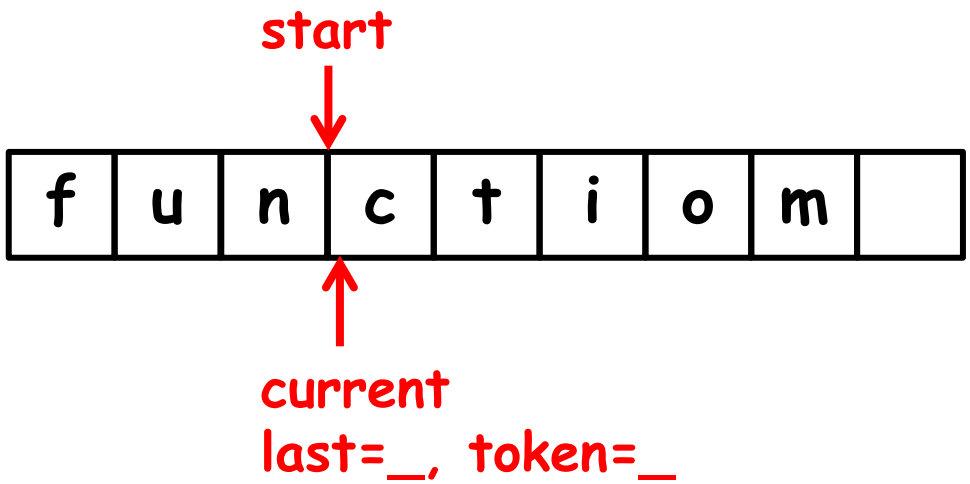
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

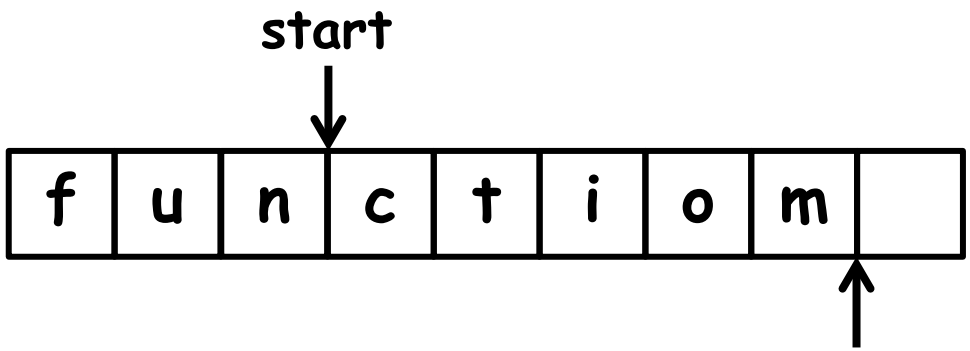
極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



対処法
(3. longest/first match)

- "longest match"の対処
  - 最後に受理状態に入った位置とトークンを覚え先読み
  - マッチする文字がない時に記憶したトークンを出力

極端な場合：
FUN: fun　FUNC: function
ID: f以外から始まる文字列



ID("ctiom")
を出力

原則の例外と対処

(4) 正規表現で表現できないものあり

例：nested comments

(* これは (* 正しいコメント *) です *)

(* これは (* コメントが閉じていません *)

⇒

- 個別にプログラミング
- トークンの定義を見直して構文解析に先送り
  (e.g. 「変数名」と「型の名前」は
  字句解析段階では区別不能)

通常のプログラミング言語では
"nested comments"以外はまれ

アウトライン

- 基本原則
- 字句解析の実際（原則の例外と対処法）
- 字句解析プログラムの具体例
  - 講義ホームページを参照
- 字句解析生成器lex

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

トークンの開始位置の情報などを初期化

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

トークンを認識する
オートマトンの初期状態へ

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

オートマトンの各状態に対応する関数を用意

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

次の文字に応じて状態遷移

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

トークンの終わりの文字だったら、次のトークンの処理へ

手書きの字句解析プログラム例

```
let rec main (input: string) =
   ...; pos_start := 0; pos_current := 0; last_token := INVALID;
   q0()
and q0 () = (* 初期状態 *)
  match readc() with
    ' ' -> (pos_start := !pos_start+1; q0())
  | '=' -> (save EQ; next())
  | '<' -> (save LT; q_lt())
  | '+' -> (save PLUS; next())
  | 'i' -> (save ID; q_i())
  | '0' -> (save INT; next())
  | c -> if '1'<=c && c<='9' then (save INT; q_num())
         else if 'a'<= c && c<='z' then (save ID; q_sym())
        else if c='¥000' then () (* 文字列の最後なら終了 *)
         else report_error(!pos_current)
and q_lt() = ...
```

最後に認識したトークンの種類を記憶

アウトライン

- 基本原則
- 字句解析の実際（原則の例外と対処法）
- 字句解析プログラムの具体例
- 字句解析生成器lex

lex (lexical analyzer generator)

- トークンの仕様記述ファイルから字句解析プログラムを自動生成
  - 正規表現からオートマトンへの変換などの煩わしい作業からコンパイラ製作者を解放
  - トークンの仕様の理解・変更が容易に
- 多くの標準的なプログラミング言語用に開発ずみ（C, Java, ML, ...）

lexの入力ファイルの形式(OCaml用)

```
{
  .... (*   ヘッダ部：
        生成されたコードの先頭に追加されるOCamlの
        プログラム。トークンのデータ型の定義や
        字句解析に用いる変数の宣言等 *)
}

... (* 本体：
      正規表現のエイリアス、
      トークンの定義とそれに対する処理 *)
{
 ... (* 生成されたコードの後ろに追加されるOCamlの
        プログラム *)
}
```

lexの入力ファイルの本体の記述例

```
let space = [' '  '¥t' '¥r']
let digit = ['0' - '9]
let digitnz = ['1'-'9']
...
rule token = parse          (* ここからトークンの定義 *)
| space+  {token lexbuf}
| newline
   {line_no := !line_no+1; token lexbuf}
| "="    {EQ}
| digitnz digit*
   {let s = Lexing.lexeme lexbuf in INT(int_of_string s)}
| ...
```

正規表現のエイリアス（別名）の宣言

残りの文字列からトークンを探す

ヘッダ部で宣言した行番号を表す変数をアップデート

トークンEQを返す

マッチした文字列を抽出

対応する整数を属性とするトークンINTを返す

レポート課題2

- lexer-by-hand.mlまたはlexer-tab.mlに次の拡張を施せ（テスト結果も添えること）

  2-1[必須] 以下のトークンを追加せよ

  GT: ">"　GEQ ">="　ELSE: "else"　TIMES: "*"

  2-2[余力があれば]

  コメント（"/*"で始まり、"*/"が含まない文字列が続いて"*/"でおわる文字列）をスペースと同一視して扱えるようにせよ。できればnested commentに対応

  例："if /* this is a comment */ x<y"

  -> IF ID("x") LT ID("y")

2-3[必須]lexer.mll を拡張し、以下のトークンを追加せよ

GT: ">"　GEQ: ">="　ELSE: "else"