

2 字句解析

♣ 「字句解析」とは **どんな処理** か.

♣ 字句解析の **方法** は.

2.1 字句解析とは

- 字句解析の仕事

1. 文字列中の字句 (単語, トークン) を認識する
2. それぞれの字句の **種類** がわかるようにする
3. 字句を必要な **内部表現** に変換する

- 字句の種類 (C 言語の場合 6 種類)

1. **識別子** (**identifiers**)

変数や関数などの **名前** を表す

main, a123, Max.Value のように, **英文字** と **数字** からなり, **英文字** で始まる.

大 文字と **小** 文字は区別される. **_** (アンダーライン) は英文字とみなされる.

2. **キーワード** (keywords)

決まった意味で用いる語

C 言語では, 識別子として用いることはできない. このような語を特に **予約** 語 (**reserved word**) という.

C 言語 (ISO-C90) では **32** 個¹

auto, break, case, char, const, continue, default, do, double,
else, enum, extern, float, for, goto, if, int, long, register,
return, short, signed, sizeof, static, struct, switch,
typedef, union, unsigned, void, volatile, while

3. **定数** (constants)

- ある特定の型の定数を表すもの

- **整数** 定数: 123 (**10 進** 定数), 0123 (**8 進** 定数), 0xae86 (**16 進** 定数)

- **文字** 定数: 'a', '0', '\n' (**改行**), '\t' (**タブ**) など

- **浮動小数点数** 定数: 1.23e6 (1.23×10^6), 2.34e-12 (2.34×10^{-12}) など

¹C 言語の新しい規格 C99 では, inline, restrict, _Bool, _Complex, _Imaginary の 5 つが追加されている.

4. **文字列** 定数 (**string literals**)
2 重引用符 (") で囲まれた記号列 ("This is a pen." など)
5. **演算子** (**operators**)
+, =, >>, ++, += など演算を表す記号
6. **区切り子** (**delimiters** , **punctuators**)
;, ,, (などの区切り記号
7. (番外) **空白**
スペース , **タブ** , **改行** , **コメント** など, 識別子を区切る以外は無視される
コメント
/* と ***/** で区切られた文字列で, ***/** を含んではならない
// から改行記号までの文字列

- 字句解析ルーチン

- ー 字句解析の 2 形態

1. プログラム **全体** に対して字句解析を行なう.

これに続いて **構文** 解析を行なう

2. **一字句** づつ解析しながら **構文** 解析を行なう

構文 解析から一々 **字句** 解析を呼び出す

- ー いずれにしても, 一字句を切り出す「字句解析ルーチン」を考えるとよい

1. 呼び出される毎に次の **一字句** を切り出す
2. 指定された変数に, その字句の種類, 値などがセットされる

2.2 字句解析の方法

- プログラムは **文字列**

```
int fx()
{
    int ab=23;
}
```

というプログラムは, コンパイラから見ると

i	n	t		f	x	(	)	改	{	改			i	n	t		a	b	=	2	3	;	改	}	改
---	---	---	--	---	---	---	---	---	---	---	--	--	---	---	---	--	---	---	---	---	---	---	---	---	---

という文字列に過ぎない (「改」は改行文字).

ここから字句を **抽出** するのが字句解析の役割

- 具体的な字句解析ルーチンの仕事

- ー 呼び出されると 1 つのトークンを切り出す.

- 次の **1 文字** だけが見えている (これを c とする).
- **空白** は読み飛ばす.
- 切り出したトークンを **文字配列** (token とする) に格納する.
- トークンの **種類** を指定された変数 (type とする) にセットする.

● 解析例

★ 1 回目の呼び出し

c	i
token	\0
type	?

1. 'i' (**アルファベット**) で始まる字句は **識別子** か **キーワード** → 'i' を token に格納.

c	n
token	i \0
type	?

2. 次の文字が 'n' ということは, これはトークンの続き. → 'n' を token に追加

c	t
token	i n \0
type	?

3. 同様に次の文字 't' を token に追加

c	
token	i n t \0
type	?

4. 次の文字 = ' ' なので トークンは終りとわかる.

現在の token "int" は **キーワード** である.

c	
token	i n t \0
type	キーワード

これで 1 トークンの切り出しが完了

★ 2 回目の呼び出し

c	
token	\0
type	?

1. 空白を読み飛ばす

c	f
token	\0
type	?

2. c = 'f' (アルファベット) で始まるので, 識別子かキーワード → 'f' を token に格納.

c	x								
token	f	\0							
type	?								

3. 続く文字 'x' を token に追加

c	(								
token	f	x	\0						
type	?								

4. 次の文字 = ')' なので トークンは終り.

現在の token "fx" は **キーワード** ではないので, **識別子** と判定.

c	(								
token	f	x	\0						
type	識別子								

これで 1 トークンの切り出しが完了

★ 3 回目の呼び出し

c	(								
token	\0								
type	?								

1. c = ')' を見た時点で, これが一つのトークンであると判定できる. ')' をトークンに入れ, タイプをセットして終了.

c	)								
token	(	\0							
type	左括弧								

★ 4 回目の呼び出し

c	)								
token	\0								
type	?								

1. c = ')' を見た時点で, これが一つのトークンであると判定できる. ')' をトークンに入れ, タイプをセットして終了.

c	改								
token	)	\0							
type	右括弧								

★ 5 回目の呼び出し

c	改								
token	\0								
type	?								

1. 改行記号を読み飛ばす

c	{								
token	\0								
type	?								

2. c = '{' を見た時点で、これが一つのトークンであると判定できる. '{' をトークンに入れ、タイプをセットして終了.

c	改								
token	{	\0							
type	左ブレース								

● 字句解析ルーチンのおおよその構造

[List 2.1]

```
void 字句解析 () {
    空白の読み飛ばし;
    if      (先頭がアルファベット) { 識別子 , キーワード の処理; }
    else if (先頭が数字) { 整数 定数, 浮動小数点 定数の処理; }
    else if (先頭が ' ) { 文字 定数の処理; }
    else if (先頭が " ) { 文字列 定数の処理; }
    else if (先頭が記号) { 演算子 および 区切り記号 の処理; }
    else { エラー; }
}
```



Nagisa ISHIURA