编译原理-语法分析 MIDL

2018303252 邹荧双

MIDL 词法规则

关键字

struct float boolean short long double int8 int16 int32 int64 uint8 int16 int32 int64 char unsigned

专用符号

{ } ; [ ] * + – ~ / % >> << & ^ | ,

标识符 ID

ID = LETTER (UNDERLINE?(LETTER | DIGIT))*

整数 INTEGER

INTEGER = (0 | [1–9] [0–9]*) INTEGER_TYPE_SUFFIX?

字符串 STRING（可以转义）

STRING = " (ESCAPE_SEQUENCE | (~\ | ~"))* "

BOOLEAN

BOOLEAN = TRUE | FALSE

其他

LETTER = [a–z] | [A– Z]

DIGIT = [0–9]

UNDERLINE= _

INTEGER_TYPE_SUFFIX = I | L

ESCAPE_SEQUENCE = \ (b | t | n | f | r | " | \ )

MIDL 语法规则

struct_type->“struct” ID “{” member_list “}”EOF

member_list-> { type_spec declarators “;” }

type_spec -> base_type_spec | struct_type

base_type_spec->floating_pt_type|integer_type| “char” | “boolean”

floating_pt_type -> “float” | “double” | “long double”

integer_type -> signed_int | unsigned_int

signed_int->(“short”|“int16”)

|(“long”|“int32”)

|(“long” “long”|“int64”)

|“int8”

unsigned_int -> (“unsigned”“short”| “unit16”)

| (“unsigned”“long”| “unit32”)

| (“unsigned” “long” “long” | “unit64”)

| “unit8”

declarators -> declarator {“,” declarator }

declarator -> ID [exp_list]

exp_list -> “[” or_expr { “,”or_expr } “]”

or_expr -> xor_expr {“|” xor_expr }

xor_expr -> and_expr {“^” and_expr }

and_expr -> shift_expr {“&”shift_expr }

shift_expr –> add_expr { (“>>” | “<<”) add_expr }

add_expr –> mult_expr { (“+” | “–”) mult_expr }

mult_expr –> unary_expr { (“*” |“/”|“%”) unary_expr }

unary_expr –> [“–”| “+” | “~”] (INTEGER | STRING | BOOLEAN)

① struct_type -> "struct" ID "{" member_list "}" EOF

② member_list -> { type_spec declarators ";" }

③ type_spec -> base_type_spec | struct_type

④ base_type_spec -> floating_pt_type | integer_type | "char" | "boolean"

⑤ integer_type -> signed_int | unsigned_int

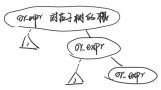
⑥ signed_int -> ("short" | "int16") | ("long" | "int32") | ("long" "long" | "int64") | "int8"

⑦ unsigned_int -> ("unsigned" "short" | "uint16") | ("unsigned" "long" | "uint32") | ("unsigned" "long" "long" | "uint64") | "uint8"

⑧ declarators -> declarator { "," declarator }

⑨ declarator -> ID [exp_list]

⑩ exp_list -> "[" or_expr { "," or_expr } "]"



⑪ xor_expr -> and_expr { "^" and_expr }

⑫ and_expr -> shift_expr { "&" shift_expr }

⑬ shift_expr -> add_expr { (">>" | "<<") add_expr }

⑭ add_expr -> mult_expr { ("+" | "-") mult_expr }

⑮ mult_expr -> unary_expr { ("*" | "/" | "%") unary_expr }

⑯ unary_expr -> ["-" | "+" | "~"] (INTEGER | STRING | Boolean)

测试

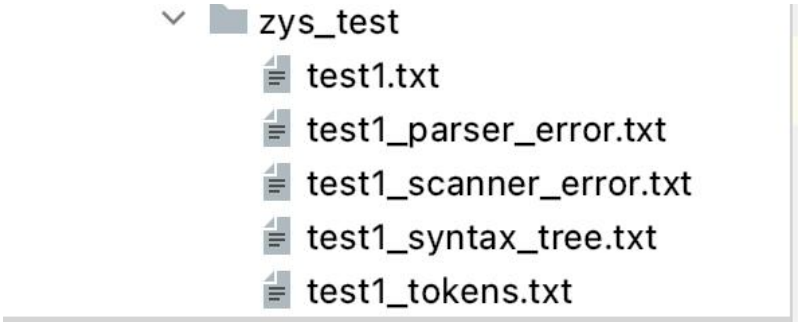
1. 只需调用 Parser 类的 Run 函数，函数内为输入文件的名（去掉.txt），如下图所示：

```
parser.Run( input_file_name: "../test/zys_test/test1");
parser.Run( input_file_name: "../test/zys_test/test2");
parser.Run( input_file_name: "../test/zys_test/test3");
parser.Run( input_file_name: "../test/zys_test/test4");
```

Screen Shot 2021-05-13 at 12.19.25 AM

注：因为使用 clion 可执行文件位于 cmake-build-debug 文件夹，因此此时的 test 文件夹在上级目录，也可以使用绝对路径。

2. 测试文件的输出，如下图所示：



Screen Shot 2021-05-13 at 12.22.13 AM

注：其中后缀为 paser_error.txt 的文件为语法分析的错误输出，scanner_error.txt 文件为词法分析的输出，syntax_tree.txt 文件为抽象语法树的输出，tokens.txt 的文件为词法分析 tokens 的输出。

3. 该语法分析使用鸵鸟政策，如遇到了语法错误将会继续执行，直到文件截止。