Compiladores I (2011-2012)

Práctica 2.1 – Compilador: Analizador Sintáctico

Ejercicio

Implementar un analizador sintáctico y léxico del lenguaje de programación ANSI C (versión reducida).

En el caso del analizador léxico sólo deberéis adaptar el analizador ya implementado en la práctica 1.

En el *Anexo* encontrareis la gramática mínima que debe soportar el analizador sintáctico.

El fichero de entrada y salida se pasarán como parámetros al analizador desde línea de comandos.

Salida del analizador

La salida del analizador se dará por fichero de texto y pantalla. La información que aparecerá en dicho salida será referente a:

- *Árbol análisis sintáctico*: reducción de las reglas a medida que se genera el árbol. Hay que indicar cada regla que se reduce de forma que se pueda seguir el recorrido del árbol generado. También se debe mostrar el atributo del Token reducido.
- *Errores:* si se encuentra un error léxico o sintáctico hay que indicar en qué línea y columna ha ocurrido.

Para facilitar la lectura del fichero de salida, se indicará en qué línea del código analizado ocurren las acciones y eventos anteriores. Por ejemplo, "int a;" generaría algo similar a:

```
Línea 12.1:     (Regla) type_specifier   <- INT
Línea 12.1:     (Regla) declaration_specifiers   <- type_specifier
Línea 12.5:     (Regla) declarator <- IDENTIFIER a
Línea 12.5:     (Regla) init_declarator  <- declarator
Línea 12.5:     (Regla) init_declarator_list <- init_declarator
Línea 12.6:     (Regla) declaration <- declaration_specifiers init_declarator_list ';'
Línea 12.6:     (Regla) external_declaration <- declaration
Línea 12.6:     (Regla) translation_unit <- external_declaration
```

Nota: la salida anterior sólo es un ejemplo, puede ser incorrecta o inexacta.

Anexo

```
%token BREAK CASE CHAR CONTINUE DEFAULT DO DOUBLE ELSE FLOAT FOR IF INT LONG
%token RETURN SHORT SIZEOF STRUCT SWITCH TYPEDEF UNION VOID WHILE
%token IDENTIFIER CHARACTER_CONSTANT INTEGER_CONSTANT FLOATING_CONSTANT STRING
%token INC_OP DEC_OP LE_OP GE_OP EQ_OP NE_OP AND_OP OR_OP
%token MUL_ASSIG DIV_ASSIG MOD_ASSIG ADD_ASSIG SUB_ASSIG
%token TYPEDEF_IDENTIFIER

%right <no_definit> ’=’
%left <no_definit> ’+’ ’-’
%left <no_definit> ’*’ ’/’

%start translation_unit

%%

primary_expression : IDENTIFIER {}
        | constant {}
        | STRING {}
        | ’(’ expression ’)’ {}
        ;

constant : INTEGER_CONSTANT {}
        | CHARACTER_CONSTANT {}
        | FLOATING_CONSTANT {}
        ;

postfix_expression : primary_expression {}
        | postfix_expression ’[’ expression ’]’ {}
        | postfix_expression ’(’ ’)’ {}
        | postfix_expression ’(’ argument_expression_list ’)’ {}
        | postfix_expression ’.’ IDENTIFIER {}
        | postfix_expression INC_OP {}
        | postfix_expression DEC_OP {}
        ;

argument_expression_list : assignment_expression {}
        | argument_expression_list ’,’ assignment_expression {}
        ;


unary_expression : postfix_expression {}
        | INC_OP unary_expression {}
        | DEC_OP unary_expression {}
        | unary_operator cast_expression {}
        | SIZEOF unary_expression {}
        | SIZEOF ’(’ type_name ’)’ {}
        ;

unary_operator : ’+’ {}
        | ’-’ {}
        | ’!’ {}
        ;

cast_expression : unary_expression {}
        | ’(’ type_name ’)’ cast_expression
        ;

multiplicative_expression : cast_expression {}
        | multiplicative_expression ’*’ cast_expression {}
        | multiplicative_expression ’/’ cast_expression {}
        | multiplicative_expression ’%’ cast_expression {}
        ;
```

```
additive_expression : multiplicative_expression {}
            | additive_expression ’+’ multiplicative_expression {}
            | additive_expression ’-’ multiplicative_expression {}
            ;

relational_expression : additive_expression {}
            | relational_expression ’<’ additive_expression {}
            | relational_expression ’>’ additive_expression {}
            | relational_expression LE_OP additive_expression {}
            | relational_expression GE_OP additive_expression {}
            ;

equality_expression : relational_expression {}
            | equality_expression EQ_OP relational_expression {}
            | equality_expression NE_OP relational_expression {}
            ;

logical_AND_expression : equality_expression {}
            | logical_AND_expression AND_OP equality_expression {}
            ;

logical_OR_expression : logical_AND_expression {}
            | logical_OR_expression OR_OP logical_AND_expression {}
            ;

conditional_expression : logical_OR_expression {}
            | logical_OR_expression ’?’ expression ’:’ conditional_expression {}
            ;

constant_expression : conditional_expression {}
            ;

assignment_expression : conditional_expression {}
            | unary_expression assignment_operator assignment_expression {}
            ;

assignment_operator : ’=’ {}
            | MUL_ASSIG {}
            | DIV_ASSIG {}
            | MOD_ASSIG {}
            | ADD_ASSIG {}
            | SUB_ASSIG {}
            ;
expression : assignment_expression {}
            | expression ’,’ assignment_expression {}
            ;

declaration : declaration_specifiers ’;’ {}
            | declaration_specifiers init_declarator_list ’;’ {}
            | TYPEDEF declaration_specifiers declarator ’;’ {}
            ;

declaration_specifiers : type_specifier {}
            ;

init_declarator_list : init_declarator {}
            | init_declarator_list ’,’ init_declarator {}
            ;

init_declarator : declarator {}
            | declarator ’=’ initializer {}
            ;
```

```
type_specifier : VOID {}
        | CHAR {}
        | SHORT {}
        | INT {}
        | LONG {}
        | FLOAT {}
        | DOUBLE {}
        | struct_or_union_specifier {}
        | typedef_name {}
        ;

typedef_name : TYPEDEF_IDENTIFIER {}
        ;

struct_or_union_specifier : struct_or_union IDENTIFIER '{' struct_declaration_list '}' {}
        | struct_or_union '{' struct_declaration_list '}' {}
        ;

struct_or_union : STRUCT {}
        | UNION {}
        ;

struct_declaration_list : struct_declaration {}
        | struct_declaration_list struct_declaration {}
        ;

struct_declaration : specifier_qualifier_list struct_declarator_list ';' {}
        ;

struct_declarator_list : struct_declarator {}
        | struct_declarator_list ',' struct_declarator {}
        ;

struct_declarator : declarator {}
        ;

declarator : IDENTIFIER {}
        | '(' declarator ')' {}
        | declarator '[' INTEGER_CONSTANT ']' {}
        | declarator '(' parameter_list ')' {}
        | declarator '(' identifier_list ')'
        | declarator '(' ')' {}
        ;

parameter_list : parameter_declaration {}
        | parameter_list ',' parameter_declaration {}
        ;

parameter_declaration : declaration_specifiers declarator {}
        | declaration_specifiers abstract_declarator {}
        | declaration_specifiers {}
        ;

identifier_list : IDENTIFIER {}
        | identifier_list ',' IDENTIFIER {}
        ;

type_name : specifier_qualifier_list {}
        | specifier_qualifier_list abstract_declarator {}
        ;

specifier_qualifier_list : type_specifier specifier_qualifier_list {}
        | type_specifier {}
        ;
```

```
abstract_declarator : ’(’ abstract_declarator ’)’ {}
            | ’[’ INTEGER_CONSTANT ’]’ {}
            | abstract_declarator ’[’ INTEGER_CONSTANT ’]’ {}
            | ’(’ ’)’ {}
            | ’(’ parameter_list ’)’ {}
            | abstract_declarator ’(’ ’)’ {}
            | abstract_declarator ’(’ parameter_list ’)’ {}
            ;

initializer : assignment_expression {}
            | ’{’ initializer_list ’}’ {}
            | ’{’ initializer_list ’,’ ’}’ {}
            ;

initializer_list : initializer {}
            | initializer_list ’,’ initializer {}
            ;

statement : labeled_statement {}
            | compound_statement {}
            | expression_statement {}
            | selection_statement {}
            | iteration_statement {}
            | jump_statement {}
            ;

labeled_statement : CASE constant_expression ’:’ statement {}
            | DEFAULT ’:’ statement
            ;

compound_statement : ’{’ ’}’ {}
            | ’{’ declaration_list ’}’ {}
            | ’{’ statement_list ’}’ {}
            | ’{’ declaration_list statement_list ’}’ {}
            ;

statement_list : statement {}
            | statement_list statement {}
            ;

expression_statement : ’;’ {}
            | expression ’;’ {}
            ;


selection_statement : IF ’(’ expression ’)’ statement {}
            | IF ’(’ expression ’)’ statement ELSE statement {}
            | SWITCH ’(’ expression ’)’ statement {}
            ;

iteration_statement : WHILE ’(’ expression ’)’ statement {}
            | DO statement WHILE ’(’ expression ’)’ ’;’ {}
            | FOR ’(’ expression ’;’ expression ’;’ expression ’)’ statement {}
            | FOR ’(’ ’;’ expression ’;’ expression ’)’ statement {}
            | FOR ’(’ ’;’ ’;’ expression ’)’ statement {}
            | FOR ’(’ ’;’ ’;’ ’)’ statement {}
            | FOR ’(’ expression ’;’ ’;’ expression ’)’ statement {}
            | FOR ’(’ expression ’;’ ’;’ ’)’ statement {}
            | FOR ’(’ ’;’ expression ’;’ ’)’ statement {}
            | FOR ’(’ expression ’;’ expression ’;’ ’)’ statement {}
            ;
```

```
jump_statement : CONTINUE ';' {}
        | BREAK ';' {}
        | RETURN ';' {}
        | RETURN expression ';' {}
        ;

translation_unit : external_declaration {}
        | translation_unit external_declaration {}
        ;

external_declaration : function_definition {}
        | declaration {}
        ;

function_definition : declarator compound_statement {}
        | declaration_specifiers declarator declaration_list compound_statement {}
        | declaration_specifiers declarator compound_statement {}
        | declarator declaration_list compound_statement {}
        ;

declaration_list : declaration {}
        | declaration_list declaration {}
        ;

%%
```