

Reporte

Matías Greco, Javier Reyes

2 de Octubre, 2018

Introducción

El presente reporte explica el trabajo realizado para el desarrollo de un analizador léxico y sintáctico para el lenguaje de programación C+-.

El lenguaje C+- corresponde a un subconjunto del lenguaje C, con la adición de algunas características de C++, como la posibilidad de definir una función con paso por valor o paso por referencia.

El analizador léxico y sintáctico fue desarrollado en la herramienta ANTLR4 (ANother Tool for Language Recognition) con lenguaje de salida C++.

Características del lenguaje

El lenguaje C+- incluye las siguientes características, propias del standard C89

- Funciones.
- Declaraciones.
- Asignaciones.
- Expresiones lógicas y de operaciones.
- If.
- Switch (case y default).
- While.
- For.
- Do While.
- Struct.

Algunas características interesantes:

- Comma expression

Características no incluidas

- **Macros:**

No se incluyó debido a que complica todo el proceso. Requeriría una precompilación y la capacidad de incluir otros archivos.

- **Punteros:**

No se incluyó debido a que complica la gramática, por la aparición de una infinidad de tipos distintos (del estilo `int***`). También requiere una administración de memoria que se escapa un poco de nuestro objetivo.

- **Typedef:**

No se incluyó debido a que complica la gramática. Eso permitiría utilizar una expresión de tipo `VAR` como una definición de tipo. Al no incluirlo, el lenguaje no pierde capacidades, ya que una estructura personalizada `S` tiene tipo `struct S`.

Definiciones léxicas

La definición de los componentes léxicos del lenguaje `C+-` es similar al lenguaje `C`, y se define de la siguiente forma:

- **Keywords:** `int`, `char`, `double`, `float`, `long`, `short`, `unsigned`, `sizeof`, `if`, `else`, `while`, `for`, `break`, `continue`, `true`, `false`, `struct`, `void`, `return`, `switch`, `case`, `default`, `do`. Tienen el mismo uso que en `C`.
- **Identificadores:** Puede componerse de letras, números y guiones bajos, pero no pueden empezar con un número.
- **Valores constantes:** Pueden ser números enteros con o sin signo (expresables en base 8, 10 y 16), números de punto flotante, caracteres y strings.
- **Operadores aritméticos:** `+` para suma, `-` para resta, `*` para multiplicación / para división y `%` para el resto de la división.
- **Operadores de comparación:** `==`, `!=`, `<=`, `>=`, `<`, `>`.
- **Operadores unarios:** `++`, `--`, `+`, `-`, `!`, `~`.
- **Operadores de shift:** `<<`, `>>`.
- **Operadores bitwise:** `&`, `^`, `|`.
- **Operadores lógicos:** `&&`, `||`.
- **Operador ternario:** `? :`
- **Operador coma:** `exp1, exp2` ejecuta `exp1`, luego `exp2` y retorna `exp2`.
- **Operadores varios:** `sizeof` retorna el tamaño en bytes de una expresión o tipo; llamadas a métodos (`f(exp1, exp2)`); acceso a miembros (`estructura.miembro`); y acceso a elementos de un array (`arr[i]`).

Resolución de ambigüedades y precedencias

Las únicas ambigüedades presentes en el lenguaje son

- Problema If/If/Else: En este caso, si se da

```
if(exp1) if(exp2) st1; else st2;
```

el `else` corresponde siempre al `if` más al interior, es decir, es equivalente a

```
if(exp1)
{
    if(exp2)
    {
        st1;
    }
    else
    {
        st2;
    }
}
```

- Problemas del estilo

```
a = b ? c : d = e;
```

En este caso, el standard no determina si el orden debe ser

```
a = (b ? c : (d = e));
```

o bien

```
a = ((b ? c : d) = e);
```

Por lo tanto, dejamos este problema como *undefined behavior*.

Las precedencias son las standard para C y para lenguajes de programación similares.

Apéndice 1: Toda la gramática

Antes que todo, cabe resaltar que si bien esta gramática no es LL(*) dado que tiene recursiones por la izquierda, ANTLR4 es capaz de reconocer algunas de estas recursiones y por detras las modifica para que la gramática resultante sea efectivamente LL(*). Por ejemplo, para la variable **expression**, la modificación es esencialmente la misma que agregar tantos niveles asociados a expresiones como prioridades para operaciones hay.

```
grammar Cmm2;

build:
    (
        declare_statement
        | forward_function_definition
        | function_definition
        | struct_definition
        | ';'
    )*
;

declare_statement:
    declare_expression ';'
;

declare_expression:
    type VAR ('=' expression)? (',' VAR ('=' expression)?)*
    | type VAR '[' comma_expression ']'
;

compare_op:
    '=='
    | '<='
    | '>='
    | '<'
    | '>'
;

assign_op:
    '='
    | '+='
    | '-='
    | '*='
    | '/='
    | '%='
    | '<<='
```

```

    | '>>='
    | '&='
    | '^='
    | '|='
;

unary_left_op:
    '++'
    | '--'
    | '+'
    | '-'
    | '!'
    | '~'
;

statement:
    comma_expression? ';'
    | declare_statement
    | Break ';'
    | Continue ';'
    | Return comma_expression? ';'
    | Case INT_NUMBER ':'
    | Default ':'
    | if_statement
    | while_statement
    | for_statement
    | switch_statement
    | do_statement
    | '{' statement* '}'
;

if_statement:
    If '(' comma_expression ')' statement (Else statement)?
;

switch_statement:
    Switch '(' comma_expression ')' '{' statement* '}'
;

while_statement:
    While '(' comma_expression ')' statement
;

for_statement:
    For '('

```

```

        (comma_expression | declare_expression)? ';'
        comma_expression? ';'
        comma_expression?
        ')' statement
;

do_statement:
    Do statement While '(' comma_expression ')' ';'
;

function_call_expression :
    VAR '(' (expression (',' expression)*)? ')'
;

function_definition :
    (type | 'void') VAR '('
    ((type '&'? VAR (',' type '&'? VAR)*)? | 'void')
    ')' '{' statement* '}'
;

forward_function_definition:
    (type | 'void') VAR '('
    ((type '&'? VAR? (',' type '&'? VAR?)*)? | 'void')
    ')' ';'
;

struct_definition:
    'struct' VAR '{'
    declare_statement*
    '}' ';'
;

FLOAT_NUMBER :
    [0-9]* '.' [0-9]+
    | [0-9]+ '.' [0-9]*
;

INT_NUMBER:
    DEC_NUMBER ('u' | 'U')? ('ll' | 'LL')?
    | OCT_NUMBER ('u' | 'U')? ('ll' | 'LL')?
    | HEX_NUMBER ('u' | 'U')? ('ll' | 'LL')?
    | CHAR_CONSTANT
;

```

```

STRING_CONSTANT :
    '""' ~( '""' )* '""'
;

CHAR_CONSTANT:
    '\'' ~( '\'' )* '\''
;

DEC_NUMBER:
    '0'
    | [1-9] [0-9]*
;

OCT_NUMBER:
    '0' [0-7]+
;

HEX_NUMBER:
    ('0x' | '0X') [0-9a-fA-F]+
;

type:
    Unsigned? Int
    | Unsigned? Char
    | Double
    | Unsigned? Long
    | Unsigned? Short
    | Float
    | 'struct' VAR
;

//KEYWORDS
Int: 'int';
Char: 'char';
If: 'if';
Else: 'else';
While: 'while';
For: 'for';
Break: 'break';
Continue: 'continue';
True: 'true';
False: 'false';
Struct: 'struct';
Void: 'void';

```

```

Return: 'return';

Switch: 'switch';
Case: 'case';
Default: 'default';
Do: 'do';

Double: 'double';
Long: 'long';
Short: 'short';
Float: 'float';
Unsigned: 'unsigned';
Sizeof: 'sizeof';

VAR:
    [a-zA-Z_] [a-zA-Z0-9_]*
;

WS
    : [ \t\u000C\r\n]+ -> skip
;

COMMENT:
    '//' ~[\n]* -> skip
;

MULTILINE_COMMENT:
    '/*' .*? '*/' -> skip
;

comma_expression:
    expression
    | comma_expression ',' expression
;

expression:
    '(' comma_expression ')'
    | INT_NUMBER
    | STRING_CONSTANT
    | CHAR_CONSTANT
    | VAR
    | expression '.' VAR
    | 'sizeof' '(' (expression | type) ')'
    | function_call_expression

```



```

| expression '[' expression ']'
| expression ('++' | '--')
| unary_left_op expression
| expression ('*' | '/' | '%') expression
| expression ('+' | '-') expression
| expression ('<<' | '>>') expression
| expression ('<' | '<=' | '>' | '>=') expression
| expression ('==' | '!=') expression
| expression '&' expression
| expression '^' expression
| expression '|' expression
| expression '&&' expression
| expression '||' expression
| <assoc=right> expression '?' comma_expression ':' expression
| <assoc=right> expression assign_op expression
;

```