IsoSec: Un lenguaje de programación aislado y seguro

Matías Nicolás Balbuena, Francisco Javier Hillebrand, and Oscar Andrés Wohlfarhrt

Universidad Gaston Dachary

Resumen Con el fin de permitir mas flexibilidad en la configuración de las aplicaciones se propone el lenguaje IsoSec. El objetivo de este lenguaje de programación es dar comportamiento personalizado para los usuarios de un sistema a través de scripts, evitando tener que modificar el software base desde su código fuente para dar soporte a estos comportamientos personalizados.
El objetivo de este lenguaje es ser simple y comprensible para la mayoría de usuarios del software en el que se implemente IsoSec, permitiendo que estos mismos usuarios puedan editar los scripts personalmente.
Además este lenguaje no permitirá declaraciones de nuevas clases, con el fin de evitar que los usuarios o hackers puedan ejecutar código dañino no contemplado por el software que utiliza este lenguaje.

Keywords: Lenguaje de programación · Compiladores · Interpretes · Parsers · Lexers.

1. Compiladores e Interpretes

Para comprender como se llevó a cabo el desarrollo de IsoSec, primero debemos comprender algunas nociones básicas de como funcionan los compiladores.

Estos están divididos en dos partes, Frontend y Backend, los cuales a su ves se poseen divisiones internas como se verá a continuación.

1.1. Frontend

Durante esta etapa es donde se analiza y procesa el código fuente del lenguaje para reconocer los tokens, o palabras claves, y la sintaxis del mismo. Esto se hace con el fin de obtener un código intermedio que se pueda utilizar en etapas posteriores.

Lexer En este punto se deben definir los tokens que se usarán para el código fuente del lenguaje, estos tokens pueden ser desde números hasta palabras completas, por ejemplo en el lenguaje C las palabras “if”, “while”, “for”, “struct”, etc. son tokens del lenguaje.

Dado que los números o nombres de variables también pueden ser tokens, se puede requerir una forma general para describir estos tokens. Esto normalmente se logra utilizando expresiones regulares o algún otro método que permita identificar conjuntos de caracteres, palabras o frases dentro del código fuente.

Parser Una ves definidos los tokens, se procede a definir como se deben estructurar los mismos para obtener la sintaxis del lenguaje. Esto se logra a través del parser, el cual debe poder identificar estos tokens y las estructuras que estos forman. Esto se hace con el objetivo de lograr un árbol, donde cada nodo es un token dentro del código fuente.

En este punto es donde existen multitud de métodos y formas para el análisis de la sintaxis. Para el caso particular de IsoSec se utilizó un parser

Código intermedio Una ves que se obtuvo un árbol que describe la sintaxis del lenguaje, este se debe traducir a un código intermedio, el cual esta completamente desligado del entorno en el que se implementará el lenguaje. Un ejemplo de código intermedio es el ByteCode de las maquinas virtuales de Java.

1.2. Backend

Finalmente, una ves que se posee un código intermedio, este se debe vincular con el entorno en el que se va a ejecutar, obteniendo así el código objeto que luego se vinculará con el sistema operativo sobre el que se ejecutará el programa creado con nuestro lenguaje.

Para nuestro caso particular, IsoSec utiliza como código intermedio al lenguaje CSharp, por lo que para compilarlo se utiliza el compilador de este lenguaje.

2. Desarrollo de IsoSec

2.1. Motivación

La motivación para crear este nuevo lenguaje surgió de ofrecer al usuario una forma de editar scripts o pequeños fragmentos de código que puedan alterar el comportamiento de un programa, pero sin darle el control completo que ofrecen algunos lenguajes de programación como Lua.

Lua Según [1], Lua se define como un lenguaje de scripting poderoso, eficiente, ligero e integrable. Soportando multiples paradigmas como programación procedural, orientada a objetos, funcional y orientada a datos.

Al ver la sintaxis de este lenguaje es imposible no notar el parecido que tiene con un pseudo-código en ingles, por lo que también resulta un lenguaje simple comprender y leer.

El problema que presenta este lenguaje en nuestro caso, es que pese a ser integrable, tiene soporte completo de operaciones que pueden crear, alterar e

incluso eliminar archivos, además de poseer la capacidad de realizar solicitudes HTTP.

En un entorno donde se cargan fragmentos de código constantemente, alguien podría introducir un virus fácilmente utilizando estas operaciones, o incluso poner en riesgo la integridad del sistema sobre el que se estén ejecutando estos scripts.

2.2. IsoSec como lenguaje de programación

Como se discutió antes, IsoSec surgió con la idea de ser un lenguaje aislado (Isolated) y seguro (Secure), de ahí el nombre.

IsoSec es un lenguaje **compilado**, **de alto nivel**, **eficiente**, **integrable** y **seguro**, dado que tiene restringido el acceso a cualquier recurso que no se le haya facilitado al momento de compilarse.

Como lexer y parser se utilizó ANTLR (en su versión 4 - ANTLR4), el cual según [2] se define como una generador de parsers poderoso para leer, procesar, ejecutar o traducir texto estructurado o archivos binarios. ANTLR4 utiliza la notación de Backus-Naur extendida (EBNF - Extended Backus-Naur Form), por lo que a continuación se presentará la gramática del lenguaje en este formato.

2.3. Gramática Formal

Como se mencionó anteriormente, un lenguaje de programación tiene un lexer y un parser, el conjunto de las definiciones de ambos resulta en una gramática formal que describe al lenguaje, en este caso a IsoSec.

Las gramáticas formales tienen dos tipos de tokens, los terminales y los no-terminales. Los terminales son partes de código fuente que se pueden encontrar en el archivo que contenga dicho código fuente, mientras que los no-terminales son conjuntos de tokens (terminales y no-terminales) que definen estructuras de tokens y dan origen a la sintaxis del lenguaje.

A continuación se listarán los tokens terminales y no-terminales del lenguaje IsoSec.

Lexer A continuación se listarán los tokens terminales de IsoSec y las definiciones de los mismos.

```
WS : [ \t\f\r\n] -> skip;

EndLine : ';';

LPar : '(';
RPar : ')';
LSBra : '[';
RSBra : ']';

RetOp : 'return';
```

```
NewOp : 'create';

IncOp : '++';
DecOp : '--';

PowerOp : 'pow';
PlusOp : '+';
MinusOp : '-';
ModOp : 'mod';
TimesOp : '*';
DivOp : '/';

LBSOp : '<<';
RBSOp : '>>';

LEOp : '<=';
GEOp : '>=';
LessOp : '<';
GreatOp : '>';
EqualOp : '==';
NEqualOp : '!=';
AssignOp : '=';

LAndOp : '&&';
LOrOp : '||';
LNotOp : '!';

BAndOp : '&';
BXOrOp : '^';
BOrOp : '|';
BNotOp : '~';

DotOp : '.';
Comma : ',';

Float : ([0-9]+'.'[0-9]+'f' | [0-9]+'f');
Double : [0-9]+'.'[0-9]+;
Int : [0-9]+;
Bool : 'true' | 'false';
Char : '\'' '\\'?. '\'';
String : '$'?'"' .*? '"';

Name : [_a-zA-Z][_a-zA-Z0-9]*;

BlockComment : '/*' .*? '*/' -> skip;
LineComment : '//' ~[\r\n]* -> skip;
```

Parser A continuación se listarán los tokens no-terminales de IsoSec y las definiones de estos.

```
program : (funcDeclaration | decLine)*;

typeName : compName ('<' typeName (Comma typeName)* '>')?
    (LSBra RSBra)? '?'?;
funcDeclaration : type=typeName 'function' name=Name LPar
    declararionArgs? RPar line* 'end';

declararionArgs : declaration (Comma declaration)?;

line : lstat=stat EndLine?;
decLine : dec=declaration EndLine?;

stat : RetOp exp #returnStat
     | declaration #decStat
     | ctrlStruct #ctrlStat
     | exp #expStat;

compName : compName DotOp Name
         | Name;

declarationBase : type=typeName name=Name;
declaration : baseDec=declarationBase (AssignOp value=exp)?;

ctrlStruct : 'if' cond=exp 'then' line* ifElse* else? 'end'
    #if
            | 'while' cond=exp 'do' line* 'end' #while
            | 'do' line* 'while' cond=exp #doWhile
            | 'for' (dvar=declaration | ivar=exp) ';'
    cond=exp ';' iexp=exp 'do' line* 'end' #for;

ifElse : 'else' 'if' cond=exp 'then' line*;
else : 'else' line*;

exp : LPar type=typeName RPar right=exp
    | left=exp op=( IncOp | DecOp )
    | op=( IncOp | DecOp | LNotOp | BNotOp | MinusOp )
    right=exp
    | left=exp op=PowerOp right=exp
    | left=exp op=( TimesOp | DivOp | ModOp ) right=exp
    | left=exp op=( PlusOp | MinusOp ) right=exp
    | left=exp op=( LBSOp | RBSOp ) right=exp
    | left=exp op=( GreatOp | LessOp | GEOp | LEOp )
    right=exp
    | left=exp op=( EqualOp | NEqualOp ) right=exp
    | left=exp op=BAndOp right=exp
    | left=exp op=BXOrOp right=exp
    | left=exp op=BOrOp right=exp
    | left=exp op=LAndOp right=exp
    | left=exp op=LOrOp right=exp
    | left=exp op=AssignOp right=exp
```

```
    | LPar left=exp RPar
    | compVar;

compVar : left=compVar DotOp right=atomVar
        | at=atom;

atom : newObj
     | atomVar
     | const;

atomVar : func
        | var;

var : name=Name LSBra args RSBra
    | name=Name;

func : name=Name LPar RPar
     | name=Name LPar args RPar;

args : exp (Comma exp)*;

const : Bool
      | Char
      | String
      | Float
      | Double
      | Int;

newObj : NewOp typeName LPar args? RPar;
```

2.4. Sintaxis

Se intentó mantener la sintaxis de IsoSec lo mas simple posible. Se puede añadir finales de linea (';'), pero es opcional y el lenguaje debería interpretar correctamente cualquier código fuente sin importar si existen varias instrucciones en una misma linea aunque no haya separadores.

Algunos ejemplos de la sintaxis de las lineas son:

```
ValidLine()
ValidLine();

ValidLine1() ValidLine2()
ValidLine1(); ValidLine2()
ValidLine1(); ValidLine2();
```

Comentarios Para los comentarios se decidió usar la misma sintaxis que utilizan lenguajes como C/C++, CSharp, Java, etc.

```
// Single line comment

/*
    Multiline comment
*/
```

Funciones IsoSec permite la declaración de funciones, en caso de que el valor devuelto por estas sea 'void', se debe omitir la linea de return.

```
Type function Name(arguments)
    //code ..
    return value //if the function Type is not void
end
```

Condicionales La definición de condicionales se baso casi por completo en Lua, manteniendo una sintaxis similar al lenguaje natural y simple de comprender. La unica diferencia a destacar con Lua es que lua posee el token 'elseif', en lugar de separar ambas palabras.

```
// Simple If statment
if condition then
    //code ...
end

// If Else statment
if condition then
    //code ...
else
    //code ...
end

// Else If statment
if condition1 then
    //code ...
else if condition2 then
    //code ...
else if condition3 then
    //code ...
else // optional
    //code ...
end
```

No se consideró la implementación de una estructura 'switch', habitual en muchos lenguajes modernos. Esta decisión se debió a la inspiración por parte de Lua [1], el cual también carece de este tipo de estructura.

Bucles Finalmente, la sintaxis de los bucles también se basó parcialmente en la de Lua. Dado que IsoSec es fuertemente tipado, se modificó un poco la sintaxis del bucle 'for', pareciéndose también a la sintaxis de C/C++.

```
// While loop
while condition do
    //code ...
end

// Do While loop
do
    //code ...
while condition

// For loop
for variable; condition; increment do
    //code ...
end
```

2.5. Transpilador a CSharp

Inicialmente se consideró la posibilidad de crear un interprete propio, pero dada la complejidad del desarrollo del mismo, se optó por hacer un transpilador a otro lenguaje.

El lenguaje intermedio que se decidió usar fue CSharp [3], dado que este es el lenguaje sobre el que se estaba desarrollando el lexer y el parser. Esto tambien hizo que el lenguaje pasara de ser compilado, permitiendo tambien aprobechar el rendimiento que esto conlleva.

Dada esta decisión, IsoSec puede utilizar todas las bibliotecas, objetos y funciones que ofrece CSharp, lo que también permitió agilizar el desarrollo de IsoSec al no tener que implementar desde 0 muchas funciones básicas.

2.6. Ejecución de Scripts

Finalmente, para ejecutar los scripts desarrollados con IsoSec, se desarrolló una aplicación de consola, aunque este proceso también se puede incrustar en cualquier programa desarrollado en CSharp de forma fácil.

La aplicación de consola se puede descargar desde el repositorio de IsoSec, junto con todo el código fuente del mismo.

3. Conclusión

Finalmente, se concluye que este proyecto tubo éxito. Si bien originalmente se consideró que el lenguaje fuera interpretado, y luego pasó a ser compilado, se logró el objetivo que se proponía de un lenguaje aislado y seguro.

Al utilizar el compilador de CSharp, se requiere incluir manualmente las bibliotecas a las que tendrá acceso el script. En caso de que este requiera algo a lo que no se le permitió acceso, este dará error y no se podrá ejecutar.

Por lo tanto, se puede decir que se a desarrollado un lenguaje que puede ser incrustado en casi cualquier aplicación (siempre que esté desarrollada con

.NET), y al cual se le puede restringir el acceso a los recursos del sistema, evitando cualquier posibilidad de un ataque al programa o sistema operativo.

Bibliografía

[1] R. Ierusalimschy, W. Celes y L. H. de Figueiredo. «The Programming Language Lua.» (), dirección: https://www.lua.org/ (visitado 25-10-2024).
[2] T. Parr. «ANTLR.» (), dirección: https://www.antlr.org/ (visitado 25-10-2024).
[3] BillWagner. «C# Guide - .NET managed language.» (), dirección: https://learn.microsoft.com/en-us/dotnet/csharp/ (visitado 25-10-2024).