**Lenguaje de Condiciones — Manual de referencia**

Fecha: 2025-08-27 Autor: Gustavo E. Enríquez

**1. Propósito**

Documento que define el lenguaje de condiciones usado por el diseñador visual y el motor (engine). Está diseñado para ser simple, legible por humanos y por IA, y fácil de evaluar en Delphi.

**2. Resumen rápido**

* Sintaxis inspirada en SQL / JavaScript: operadores =, <>, >, <, >=, <= y lógicos AND, OR, NOT.
* Valores: números, strings (comillas simples o dobles), booleanos (TRUE, FALSE).
* Variables: identificadores simples como score, status o la forma indexada Context["key"].
* Se serializa como texto en el campo expression dentro del contrato JSON.

**3. Léxico**

* Identificadores (variables): [\_A-Za-z][\_A-Za-z0-9]\*.
* Números: secuencias de dígitos, opcional punto decimal: 123, 45.67.
* Strings: entre comillas dobles "texto" o simples 'texto'. Para incluir la comilla dentro del string, escapar con \ o usar la otra comilla (p. ej. "He said ""hi""" o 'He said ''hi''').
* Booleanos: TRUE, FALSE (no sensibles a mayúsculas).
* Operadores: =, <>, >, <, >=, <=, AND, OR, NOT.
* Paréntesis: ( y ) para agrupar.

**4. Gramática (EBNF simplificada)**

<expression> ::= <or\_expr>

<or\_expr> ::= <and\_expr> { 'OR' <and\_expr> }

<and\_expr> ::= <not\_expr> { 'AND' <not\_expr> }

<not\_expr> ::= [ 'NOT' ] <cmp\_expr>

<cmp\_expr> ::= <term> [ <cmp\_op> <term> ]

<cmp\_op> ::= '=' | '<>' | '>' | '<' | '>=' | '<='

<term> ::= <identifier> | <number> | <string> | 'TRUE' | 'FALSE' | '(' <expression> ')'

identifier ::= letter { letter | digit | '\_' }

number ::= digit { digit } [ '.' digit { digit } ]

string ::= '...' | "..."

Notas:

* El operador = realiza comparación de igualdad entre tipos compatibles. Para strings es comparación literal.
* <> es desigualdad.
* Precedencia: NOT > AND > OR.

**5. Ejemplos válidos**

* score >= 80 AND passed = TRUE
* region = "EU" OR region = 'US'
* NOT error
* (age > 18 AND country = "CO") OR admin = TRUE
* name = "O\'Reilly" (ejemplo con comilla escapada)

**6. Representación JSON (contrato)**

Una condición se incluye en el contrato del enlace como un objeto:

{

"condition": {

"expression": "score >= 80 AND passed = TRUE",

"variables": ["score","passed"]

}

}

* expression: la expresión en texto.
* variables: (opcional) lista de variables usadas — útil para UIs y para prevalidar.

**7. Evaluación en el engine (Delphi)**

**7.1 Requisitos del motor**

* El motor recibe la expresión y un diccionario Variables: map<string, TValue> (valores en runtime).
* Debe evaluar la expresión y devolver true o false.
* Debe manejar tipos: Integer, Float, String, Boolean.

**7.2 Implementación sugerida (inicial)**

* Usar System.Bindings.Eval (si está disponible en la plataforma objetivo) para evaluar expresiones simples.
* Alternativa: implementar un parser/árbol AST usando la gramática EBNF y evaluar recorriendo el AST.

Ejemplo de función en Delphi (esquema):

function EvalCondition(const Expr: string; Vars: TDictionary<string, TValue>): Boolean;

var

Context: TBindings;

Pair: TPair<string, TValue>;

begin

Context := TBindings.Create;

try

for Pair in Vars do

Context.AddVariable(Pair.Key, Pair.Value);

Result := TValueData.Evaluate(Expr, Context).AsBoolean;

finally

Context.Free;

end;

end;

Nota: TBindings y TValueData.Evaluate son aproximaciones; adaptar según la librería real disponible.

**8. Extensiones futuras (opcional)**

* Funciones estándar: LEN(s), IN(x, a, b), STARTSWITH(s, prefix), CONTAINS(s, part).
* Comparaciones más ricas: LIKE con patrones simples.
* Soporte para objetos anidados: user.age, order.total.
* Evaluación segura: controlar uso de funciones que puedan ejecutar código nativo.

**9. Seguridad y validación**

* El engine debe validar la expresión al compilar el grafo y rechazar expresiones sintácticamente inválidas.
* Cuando se use un evaluador que ejecute código, debe garantizarse sandboxing para evitar ejecución arbitraria.
* Validar tipos: comparar string con string, number con number, etc., o documentar conversión implícita.

**10. Checklist para integración Designer ↔ Engine**

**11. Preguntas frecuentes (FAQ)**

**Q:** ¿Se permiten funciones en la primera versión?  
**A:** No; arrancamos sin funciones. Solo operaciones aritméticas y lógicas.

**Q:** ¿Las comparaciones entre tipos diferentes se convierten implícitamente?  
**A:** No por defecto. El engine puede intentar coerción controlada (ej. "123" -> número) o reportar error.

**Documentación: Registro y Uso de Manejadores de Eventos**

El motor de ejecución de grafos proporciona dos mecanismos para definir la lógica de un nodo:

1. **Herramientas (Tools):** Clases que heredan de TAiToolBase. Son la opción preferida para lógica compleja, reutilizable y desacoplada.
2. **Manejadores de Eventos (Event Handlers):** Métodos asignados a la propiedad OnExecute de un nodo. Son ideales para prototipado rápido, lógica simple o tareas que requieren una fuerte integración con la interfaz de usuario de la aplicación.

Esta guía explica cómo registrar y utilizar los **Manejadores de Eventos**.

**Concepto Clave: El Registro de Handlers**

El sistema de guardado/carga del grafo se basa en JSON, un formato de texto que no puede almacenar punteros a métodos. Para solucionar esto, el motor utiliza un registro centralizado: TAgentHandlerRegistry.

El TAgentHandlerRegistry es un Singleton (una única instancia global) que actúa como un diccionario. Su función es mapear un **identificador de texto simple** (un string) a un **puntero de método real** en tu código.

**Flujo de trabajo:**

1. **En la carga de la aplicación:** Registras tus métodos en el TAgentHandlerRegistry, asociando cada uno a un nombre único (ej. 'SumaSimple', 'ActualizarLog').
2. **Al guardar el grafo:** El motor guarda este nombre de texto en el archivo JSON del nodo (ej. "onExecuteHandler": "SumaSimple").
3. **Al cargar el grafo:** El motor lee el nombre del JSON, lo busca en el TAgentHandlerRegistry, encuentra el puntero al método correspondiente y lo asigna al evento OnExecute del nodo.

**Guía de Implementación (Paso a Paso)**

Sigue estos tres pasos para implementar un manejador de eventos personalizado.

**Paso 1: Crear el Método Manejador**

Primero, crea un método en una de tus clases (normalmente un formulario o una clase de lógica de aplicación) que coincida con la firma del evento TAIAgentsNodeOnExecute.

La firma requerida es:

codeDelphi

procedure(Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var Output: String) of object;

**Ejemplo:**  
Supongamos que tenemos un formulario TMyMainForm. Podemos añadirle el siguiente método:

codeDelphi

// En la declaración de TMyMainForm

type

TMyMainForm = class(TForm)

private

procedure HandleNodeLogMessage(Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var Output: String);

// ...

end;

// En la implementación

procedure TMyMainForm.HandleNodeLogMessage(Node, BeforeNode: TAIAgentsNode; Link: TAIAgentsLink; Input: String; var Output: String);

begin

// Lógica del evento: Escribir en un TMemo y pasar el input sin cambios.

MemoLog.Lines.Add(Format('Nodo "%s" ejecutado con entrada: %s', [Node.Name, Input]));

Output := Input; // El output es el mismo que el input en este caso.

end;

**Paso 2: Registrar el Método en el Inicio de la Aplicación**

Este es el paso crucial. Debes "enseñarle" al motor sobre tu nuevo método. El mejor lugar para hacerlo es en un punto de inicialización de tu aplicación, como el evento OnCreate del formulario principal.

Para ello, utiliza el TAgentHandlerRegistry.

codeDelphi

// No olvides añadir la unidad del registro a tu cláusula 'uses'

uses uHandlerRegistry;

procedure TMyMainForm.FormCreate(Sender: TObject);

begin

// Registrar el manejador con un nombre único

TAgentHandlerRegistry.Instance.RegisterNodeHandler(

'LogMessageToMemo', // 1. El nombre de texto para el JSON

Self.HandleNodeLogMessage // 2. El puntero al método real

);

// Puedes registrar tantos handlers como necesites

// TAgentHandlerRegistry.Instance.RegisterNodeHandler('OtroHandler', Self.OtroMetodo);

end;

**Paso 3: Asignar el Handler en el Grafo (JSON)**

Ahora que el motor conoce el identificador 'LogMessageToMemo', puedes usarlo en la definición de tus grafos en JSON. Simplemente asigna ese string a la propiedad onExecuteHandler del nodo deseado.

**Ejemplo de JSON:**

codeJSON

{

"graph": { ... },

"nodes": [

{

"name": "NodoInicial",

"nextLink": "Link\_A\_Logger"

},

{

"name": "NodoLogger",

"description": "Este nodo usa un evento para escribir en el log de la UI.",

"tool": null,

"onExecuteHandler": "LogMessageToMemo" // <-- Aquí se usa el identificador

}

],

"links": [ ... ]

}

¡Eso es todo! Cuando cargues este grafo usando LoadFromStream, el "NodoLogger" tendrá su propiedad OnExecute automáticamente conectada a tu método TMyMainForm.HandleNodeLogMessage.

**Cuándo Usar Eventos vs. Herramientas (Tools)**

|  |  |  |
| --- | --- | --- |
| **Característica** | **Manejador de Evento (OnExecute)** | **Herramienta (TAiToolBase)** |
| **Reusabilidad** | Baja. La lógica está atada a la clase que la define. | **Alta.** La herramienta es una clase independiente y autocontenida. |
| **Complejidad** | Ideal para lógica **simple** y tareas cortas. | Ideal para lógica **compleja**, con múltiples propiedades y estado interno. |
| **Acoplamiento** | **Alto.** Fuerte acoplamiento con la clase contenedora (ej. TForm). | **Bajo.** Totalmente desacoplada del resto de la aplicación. |
| **Casos de Uso** | <ul><li>Prototipado rápido</li><li>Integración con UI (logs, barras de progreso)</li><li>Operaciones triviales</li></ul> | <ul><li>Lógica de negocio principal</li><li>Interacción con APIs externas</li><li>Algoritmos complejos</li><li>Funcionalidad para ser compartida entre proyectos</li></ul> |

**Contrato 1: datosinterfaces.json (Catálogo de Herramientas / Blueprint)**

**Propósito:**  
Este archivo es un **catálogo de todas las herramientas disponibles** que se pueden usar en el diseñador. Es leído **una sola vez** por el diseñador visual al arrancar para poblar la caja de herramientas (ToolBox). **El motor de ejecución (TGraphBuilder) NUNCA lee este archivo.**

**Estructura General:**  
Un Array de Objetos JSON. Cada objeto es un "Blueprint" que describe una herramienta.

**Estructura de un Objeto "Blueprint"**

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| displayName | String | Sí | El nombre legible que se muestra en el botón de la ToolBox. |
| description | String | Sí | Texto de ayuda (Hint) que aparece al pasar el ratón sobre el botón. |
| toolClassName | String | Sí | El nombre de la clase Delphi de la herramienta (ej. TFileWatcherTool). Es el identificador único que el diseñador guardará en el grafo. |
| schema | Objeto | Sí | Define la estructura de los parámetros de la herramienta para generar el diálogo de configuración. |
| metadata | Objeto | Sí | Contiene datos no funcionales, principalmente para la visualización del nodo. |
| orderPosition | Number | No | Entero para ordenar los botones en la ToolBox (de menor a mayor). Si no se provee, aparece al final. |

**Detalles de los Sub-Objetos**

**1. El Objeto schema**  
Define los campos del diálogo de configuración.

|  |  |  |
| --- | --- | --- |
| **Clave** | **Tipo** | **Descripción** |
| type | String | Debe ser "object". |
| title | String | Título para la ventana de configuración (ej. "Configuración de File Watcher"). |
| properties | Objeto | Objeto donde cada clave es el nombre de una propiedad de la herramienta (ej. FolderPath). El valor es un objeto que la describe:<ul><li>title (String): Etiqueta para el campo.</li><li>description (String): Texto de ayuda para el campo.</li><li>type (String): Tipo de dato JSON (string, integer, boolean, number).</li><li>default (Varios): Valor por defecto para el campo.</li><li>enum (Array de Strings): (Opcional) Opciones para un ComboBox.</li></ul> |

**2. El Objeto metadata.visualize**  
Define la apariencia por defecto del nodo en el lienzo.

|  |  |  |
| --- | --- | --- |
| **Clave** | **Tipo** | **Descripción** |
| width, height | Number | Dimensiones del nodo. |
| nodeColor | String | Color de relleno en formato #AARRGGBB. |
| borderColor | String | Color del borde. |
| textColor | String | Color del texto. |
| cornerRadius | Number | Radio de las esquinas. |
| svgIcon | String | **CRÍTICO:** Solo la data del path SVG (el contenido del atributo d). |
| iconColor | String | Color para el ícono SVG. |
| iconAlignment | String | Posición del ícono (left, right, center). |
| ports | Array | Array de objetos que definen los puntos de conexión. |

**3. El Objeto port (dentro de metadata.visualize.ports)**  
Define la lógica y apariencia de cada punto de conexión.

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| id | String | Sí | Identificador único y lógico del puerto (ej. out\_success, in\_data). |
| label | String | No | Texto que se muestra junto al puerto en el diseñador. |
| direction | String | Sí | input o output. |
| type | String | Sí | Propósito lógico: success, failure, fanout (para output); default, data (para input). |
| category | String | No | tool (defecto) o accessory. |
| maxConnections | Number/String | No | Regla de cardinalidad. Ej: 1 para salidas, "unlimited" para entradas. |
| alignment | String | Sí | Dónde se dibuja el puerto: left, right, top, bottom. |
| offsetPercent | Number | Sí | Posición a lo largo del borde (de 0.0 a 1.0). |

**Ejemplo Completo (datosinterfaces.json)**

codeJSON

[

{

"displayName": "File Exists",

"description": "Verifica si un archivo o carpeta existe en la ruta especificada.",

"toolClassName": "TFileExistsTool",

"orderPosition": 101,

"schema": {

"type": "object",

"title": "Configuración de File Exists",

"properties": {

"FilePath": {

"title": "Ruta del Archivo/Carpeta",

"description": "Ruta completa a verificar.",

"type": "string",

"default": "C:\\"

}

}

},

"metadata": {

"visualize": {

"width": 220,

"height": 100,

"nodeColor": "#FF8A2BE2",

"ports": [

{

"id": "in\_flow",

"label": "Entrada",

"direction": "input",

"type": "default",

"category": "tool",

"maxConnections": "unlimited",

"alignment": "left",

"offsetPercent": 0.5

},

{

"id": "out\_success",

"label": "Existe (Sí)",

"direction": "output",

"type": "success",

"category": "tool",

"maxConnections": 1,

"alignment": "right",

"offsetPercent": 0.3

},

{

"id": "out\_failure",

"label": "No Existe (No)",

"direction": "output",

"type": "failure",

"category": "tool",

"maxConnections": 1,

"alignment": "right",

"offsetPercent": 0.7

}

]

}

}

}

]

**Contrato 2: grafo\_guardado.json (Instancia de Grafo / Diseño de Usuario)**

**Propósito:**  
Este archivo representa **un grafo específico creado y guardado por un usuario**. Contiene las instancias de los nodos, sus parámetros configurados, sus posiciones en el lienzo y los enlaces que los conectan. **Este es el archivo que TGraphBuilder lee para construir el grafo ejecutable.**

**Estructura General:**  
Un Objeto JSON con tres claves principales: Nodos, Enlaces y opcionalmente Meta.

**Estructura de un Objeto Nodo (en el array Nodos)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| id | String | Sí | ID único de la **instancia** del nodo (ej. un GUID). |
| label | String | Sí | Nombre que el usuario le dio al nodo en el lienzo. |
| properties | Objeto | Sí | Contenedor para todos los datos de configuración del nodo. |
| ports | Array | **Recomendado** | Una copia de los objetos port del blueprint. Necesario para que el builder relacione un internalId con un id lógico. |
| position | Objeto | Sí (p/ Diseñador) | Coordenadas x e y del nodo en el lienzo. |

**Detalles del Objeto properties (de un Nodo)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| toolClassName | String | Sí | El nombre de la clase Delphi (ej. TFileExistsTool). Le dice al builder qué Tool instanciar. |
| description | String | No | Un comentario o descripción que el usuario asignó a esta instancia específica del nodo. |
| parameters | Objeto | No | Un objeto donde cada clave es el nombre de una propiedad de la Tool y el valor es lo que el usuario configuró (ej. { "FilePath": "C:\\datos.txt" }). |
| engine | Objeto | No | Contiene configuraciones específicas del motor para este nodo. Por ahora, solo joinMode (ej. { "joinMode": "jmAll" }). |

**Detalles del Objeto port (en el array ports de un Nodo)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| id | String | Sí | El ID lógico del puerto (ej. out\_success). |
| internalId | String | Sí | Un ID único (ej. GUID) que el diseñador asigna a este terminal físico. Es lo que se usa en los campos sourceTerminal y targetTerminal del enlace. |

**Estructura de un Objeto Enlace (en el array Enlaces)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| id | String | Sí | ID único de la **instancia** del enlace (ej. un GUID). |
| sourceNodeId | String | Sí | El id del nodo de origen. |
| targetNodeId | String | Sí | El id del nodo de destino. |
| sourceTerminal | String | Sí | El internalId del puerto de origen. |
| targetTerminal | String | Sí | El internalId del puerto de destino. |
| properties | Objeto | No | Contenedor para los datos de configuración del enlace. |

**Detalles del Objeto properties (de un Enlace)**

|  |  |  |  |
| --- | --- | --- | --- |
| **Clave** | **Tipo** | **Obligatorio** | **Descripción** |
| description | String | No | Un comentario o descripción que el usuario asignó a este enlace. |
| engine | Objeto | No | Contiene configuraciones específicas del motor para este enlace. Ejemplos: { "linkMode": "lmConditional", "maxCycles": 5 }. |

**Ejemplo Completo (grafo\_guardado.json)**

codeJSON

{

"Nodos": [

{

"id": "a1b2c3d4-start",

"label": "Inicio",

"properties": {

"description": "Punto de entrada del proceso.",

"toolClassName": "TStartTool",

"parameters": {

"StartMessage": "Proceso de verificación de archivo iniciado"

}

},

"ports": [

{ "id": "out\_flow", "internalId": "term-start-out" }

],

"position": { "x": 100, "y": 200 }

},

{

"id": "e5f6g7h8-check",

"label": "Verificar si existe reporte.txt",

"properties": {

"description": "Comprueba la existencia del reporte diario.",

"toolClassName": "TFileExistsTool",

"parameters": {

"FilePath": "C:\\reportes\\diario.txt"

},

"engine": {

"joinMode": "jmAny"

}

},

"ports": [

{ "id": "in\_flow", "internalId": "term-check-in" },

{ "id": "out\_success", "internalId": "term-check-success" },

{ "id": "out\_failure", "internalId": "term-check-failure" }

],

"position": { "x": 350, "y": 200 }

}

],

"Enlaces": [

{

"id": "link-1",

"sourceNodeId": "a1b2c3d4-start",

"targetNodeId": "e5f6g7h8-check",

"sourceTerminal": "term-start-out",

"targetTerminal": "term-check-in",

"properties": {

"description": "Conexión inicial hacia el verificador.",

"engine": {

"linkMode": "lmFanout"

}

}

}

]

}