

Estudio de Sentencias Complejas Secuenciales en DAAD Moderno

Resumen Ejecutivo

Este documento analiza el estado actual de la implementación de **sentencias complejas secuenciales** en el transpilador DAAD Moderno, identifica las limitaciones actuales y propone soluciones para manejar cadenas de comandos complejas que requieren ejecución secuencial.

Análisis del Estado Actual

Implementaciones Existentes

1. Patrones Complejos ([daad-transpiler-complex-patterns.cs](#))

- ☒ Manejo de patrones con múltiples parámetros
- ☒ Expansión automática de combinaciones
- ☒ Generación de múltiples entradas DAAD clásicas
- ☒ NO maneja secuencias temporales

2. Contactos Básicos ([CompleteDaadTranspiler.cs](#))

- ☒ 82 contactos DAAD implementados
- ☒ Mapeo directo moderno → clásico
- ☒ NO maneja secuencias complejas

3. Análisis de Flujo ([Services.Ant.cs](#))

- ☒ Análisis básico de flujo del juego
- ☒ Métricas de complejidad
- ☒ NO analiza secuencias de comandos

Limitaciones Identificadas

1. Falta de Secuenciación Temporal

- No hay soporte para **then**, **next**, **after**
- No hay manejo de delays o pausas
- No hay control de flujo secuencial

2. Ausencia de Cadenas de Comandos

- No se pueden crear secuencias como: **accion1() then accion2() then accion3()**
- No hay soporte para callbacks o continuaciones
- No hay manejo de estados intermedios

3. Falta de Condicionales Secuenciales

- No hay **if...then...else** complejos

- No hay soporte para condiciones encadenadas
- No hay manejo de fallos en secuencias

🔧 Propuesta de Implementación

1. Gramática Extendida para Secuencias

```
(* Extensión para sentencias secuenciales *)
SequentialStatement ::= SimpleStatement (SequenceOperator SimpleStatement)*

SequenceOperator ::= "then" | "after" | "next" | "finally" | "on_error"

SimpleStatement ::= Action | ConditionalAction | LoopAction

ConditionalAction ::= "if" Condition "{" ActionBlock "}"
                    ("else" "{" ActionBlock "}")?

LoopAction ::= "while" Condition "{" ActionBlock "}"
              | "for" Identifier "in" Range "{" ActionBlock "}"
              | "repeat" Integer "{" ActionBlock "}"

DelayedAction ::= "after" Duration ActionBlock

Duration ::= Integer ("ms" | "s" | "turns")

AsyncAction ::= "async" ActionBlock

ErrorHandling ::= "on_error" "{" ActionBlock "}"
```

2. Ejemplos de Uso

```
responses {
  on ["usar llave dorada"] {
    require: carried(llave_dorada) && at(puerta_secreta)
    do: {
      message("Insertas la llave dorada...")
      then pause(2s)
      then {
        if chance(80) {
          message("¡La puerta se abre lentamente!")
          then sound("puerta_abriendo")
          then set(puerta_abierta, 1)
          then after(1s) {
            message("Un pasaje secreto se revela.")
            goto(pasaje_secreto)
          }
        } else {
          message("La llave no encaja correctamente.")
          then vibrate(500ms)
          then message("Quizás necesites encontrar la llave correcta.")
        }
      }
    }
  }
}
```

```

        }
    }
    finally {
        add_score(10)
        set(llave_usada, 1)
    }
    on_error {
        message("Algo salió mal con la llave.")
        restart()
    }
}

on ["ritual complejo"] {
    require: carried(vela) && carried(libro) && at(altar)
    do: {
        message("Comienzas el ritual místico...")
        then light_candle()
        then after(3s) read_incantation()
        then {
            for i in 1..3 {
                message("Repites la incantación... (${i}/3)")
                then pause(1s)
            }
        }
        then {
            while not_zero(energia_magica) {
                dec(energia_magica)
                then increase_power()
                then if lt(energia_magica, 10) {
                    message("La energía se debilita...")
                    break
                }
            }
        }
    }
    finally {
        message("El ritual está completo.")
        set(ritual_completado, 1)
    }
}
}
}

```

3. Implementación Técnica

```

// =====
// IMPLEMENTACIÓN DE SENTENCIAS SECUENCIALES
// =====

public abstract record SequentialStatement;

```

```
public record SimpleSequentialStatement(Action Action) : SequentialStatement;

public record ChainedSequentialStatement(
    SequentialStatement First,
    SequenceOperator Operator,
    SequentialStatement Second
) : SequentialStatement;

public record ConditionalSequentialStatement(
    Condition Condition,
    SequentialStatement ThenBranch,
    SequentialStatement? ElseBranch = null
) : SequentialStatement;

public record LoopSequentialStatement(
    LoopType Type,
    Condition? Condition,
    SequentialStatement Body
) : SequentialStatement;

public record DelayedSequentialStatement(
    Duration Delay,
    SequentialStatement Statement
) : SequentialStatement;

public record AsyncSequentialStatement(
    SequentialStatement Statement
) : SequentialStatement;

public record ErrorHandlingSequentialStatement(
    SequentialStatement TryStatement,
    SequentialStatement ErrorHandler
) : SequentialStatement;

public enum SequenceOperator
{
    Then,          // Ejecutar después
    After,         // Ejecutar después de delay
    Next,          // Siguiente en secuencia
    Finally,       // Ejecutar al final
    OnError        // Ejecutar si hay error
}

public enum LoopType
{
    While,
    For,
    Repeat
}

public record Duration(int Value, TimeUnit Unit);

public enum TimeUnit
{

```

```
    Milliseconds,
    Seconds,
    Turns
}

// =====
// TRANSPILADOR DE SECUENCIAS
// =====

public class SequentialTranspiler
{
    private readonly ILogger<SequentialTranspiler> _logger;
    private readonly Dictionary<string, int> _processNumbers = new();
    private int _nextProcessNumber = 10; // Empezar desde proceso 10

    public SequentialTranspiler(ILogger<SequentialTranspiler> logger)
    {
        _logger = logger;
    }

    public List<ClassicProcess> TranspileSequential(SequentialStatement statement)
    {
        var processes = new List<ClassicProcess>();
        var context = new SequentialContext();

        TranspileStatement(statement, processes, context);

        return processes;
    }

    private void TranspileStatement(SerialStatement statement,
                                    List<ClassicProcess> processes,
                                    SequentialContext context)
    {
        switch (statement)
        {
            case SimpleSequentialStatement simple:
                TranspileSimple(simple, processes, context);
                break;

            case ChainedSequentialStatement chained:
                TranspileChained(chained, processes, context);
                break;

            case ConditionalSequentialStatement conditional:
                TranspileConditional(conditional, processes, context);
                break;

            case LoopSequentialStatement loop:
                TranspileLoop(loop, processes, context);
                break;

            case DelayedSequentialStatement delayed:
                TranspileDelayed(delayed, processes, context);
        }
    }
}
```

```
        break;

    case AsyncSequentialStatement async:
        TranspileAsync(async, processes, context);
        break;

    case ErrorHandlingSequentialStatement error:
        TranspileErrorHandling(error, processes, context);
        break;
    }
}

private void TranspileChained(ChainedSequentialStatement chained,
                             List<ClassicProcess> processes,
                             SequentialContext context)
{
    // Transpile first statement
    TranspileStatement(chained.First, processes, context);

    // Handle sequence operator
    switch (chained.Operator)
    {
        case SequenceOperator.Then:
            // Crear proceso para segunda parte
            var nextProcess = CreateNextProcess(context);
            processes.Add(nextProcess);

            // En el proceso actual, llamar al siguiente
            context.CurrentProcess.Conducts.Add(new Conduct("PROCESS", new[] {
nextProcess.Number }));

            // Cambiar contexto al nuevo proceso
            context.CurrentProcess = nextProcess;
            break;

        case SequenceOperator.After:
            // Implementar delay usando flags y procesos automáticos
            TranspileWithDelay(chained.Second, processes, context);
            break;

        case SequenceOperator.Finally:
            // Agregar al final del proceso actual
            context.FinallyStatements.Add(chained.Second);
            break;

        case SequenceOperator.OnError:
            // Implementar manejo de errores
            context.ErrorHandler = chained.Second;
            break;
    }

    // Transpile second statement
    TranspileStatement(chained.Second, processes, context);
}
```

```

private void TranspileConditional(ConditionalSequentialStatement conditional,
                                List<ClassicProcess> processes,
                                SequentialContext context)
{
    // Transpile condition
    var conditionConducts = TranspileCondition(conditional.Condition);
    context.CurrentProcess.Conducts.AddRange(conditionConducts);

    // Create branches
    var thenProcess = CreateNextProcess(context);
    var elseProcess = conditional.ElseBranch != null ?
CreateNextProcess(context) : null;

    // Add conditional jump
    context.CurrentProcess.Conducts.Add(new Conduct("PROCESS", new[] {
thenProcess.Number }));

    if (elseProcess != null)
    {
        context.CurrentProcess.Conducts.Add(new Conduct("PROCESS", new[] {
elseProcess.Number }));
    }

    // Transpile branches
    context.CurrentProcess = thenProcess;
    TranspileStatement(conditional.ThenBranch, processes, context);
    processes.Add(thenProcess);

    if (conditional.ElseBranch != null && elseProcess != null)
    {
        context.CurrentProcess = elseProcess;
        TranspileStatement(conditional.ElseBranch, processes, context);
        processes.Add(elseProcess);
    }
}

private void TranspileLoop(LoopSequentialStatement loop,
                           List<ClassicProcess> processes,
                           SequentialContext context)
{
    switch (loop.Type)
    {
        case LoopType.While:
            TranspileWhileLoop(loop, processes, context);
            break;

        case LoopType.For:
            TranspileForLoop(loop, processes, context);
            break;

        case LoopType.Repeat:
            TranspileRepeatLoop(loop, processes, context);
            break;
    }
}

```

```

    }
}

private void TranspileWhileLoop(LoopSequentialStatement loop,
                                List<ClassicProcess> processes,
                                SequentialContext context)
{
    // Crear proceso para el loop
    var loopProcess = CreateNextProcess(context);
    var bodyProcess = CreateNextProcess(context);

    // Proceso del loop: verificar condición
    if (loop.Condition != null)
    {
        var conditionConducts = TranspileCondition(loop.Condition);
        loopProcess.Conducts.AddRange(conditionConducts);
    }

    // Si condición es true, ejecutar body
    loopProcess.Conducts.Add(new Conduct("PROCESS", new[] { bodyProcess.Number
}));

    // Body process
    context.CurrentProcess = bodyProcess;
    TranspileStatement(loop.Body, processes, context);

    // Al final del body, volver al loop
    bodyProcess.Conducts.Add(new Conduct("PROCESS", new[] { loopProcess.Number
}));

    processes.Add(loopProcess);
    processes.Add(bodyProcess);
}

private void TranspileDelayed(DelayedSequentialStatement delayed,
                                List<ClassicProcess> processes,
                                SequentialContext context)
{
    // Implementar delay usando flags temporales y procesos automáticos
    var delayFlag = GetNextDelayFlag();
    var delayProcess = CreateNextProcess(context);

    // Establecer flag de delay
    context.CurrentProcess.Conducts.Add(new Conduct("LET", new[] { delayFlag,
delayed.Delay.Value }));

    // Proceso automático que decreuenta el delay
    var autoProcess = new ClassicProcess
    {
        Number = 1, // Proceso automático
        Conducts = new List<Conduct>
        {
            new("GT", new[] { delayFlag, 0 }),
            new("MINUS", new[] { delayFlag, 1 }),

```



```
        new("ZERO", new[] { delayFlag }),
        new("PROCESS", new[] { delayProcess.Number })
    }
};

processes.Add(autoProcess);

// Proceso que se ejecuta después del delay
context.CurrentProcess = delayProcess;
TranspileStatement(delayed.Statement, processes, context);
processes.Add(delayProcess);
}

private void TranspileAsync(AsyncSequentialStatement async,
                           List<ClassicProcess> processes,
                           SequentialContext context)
{
    // Crear proceso separado para ejecución asíncrona
    var asyncProcess = CreateNextProcess(context);

    // Llamar al proceso asíncrono sin esperar
    context.CurrentProcess.Conducts.Add(new Conduct("PROCESS", new[] {
asyncProcess.Number }));

    // Transpile async statement
    context.CurrentProcess = asyncProcess;
    TranspileStatement(async.Statement, processes, context);
    processes.Add(asyncProcess);
}

private void TranspileErrorHandling(ErrorHandlingSequentialStatement error,
                                    List<ClassicProcess> processes,
                                    SequentialContext context)
{
    // Implementar try-catch usando flags de error
    var errorFlag = GetNextErrorFlag();
    var errorProcess = CreateNextProcess(context);

    // Establecer handler de error
    context.ErrorHandler = error.ErrorHandler;
    context.ErrorFlag = errorFlag;

    // Transpile try statement
    TranspileStatement(error.TryStatement, processes, context);

    // Error handler process
    var errorHandlerProcess = new ClassicProcess
    {
        Number = errorProcess.Number,
        Conducts = new List<Conduct>
        {
            new("EQ", new[] { errorFlag, 1 }),
            new("CLEAR", new[] { errorFlag })
        }
    }
```

```

    };

    context.CurrentProcess = errorHandlerProcess;
    TranspileStatement(error.ErrorHandler, processes, context);
    processes.Add(errorHandlerProcess);
}

// Métodos de utilidad
private ClassicProcess CreateNextProcess(SequentialContext context)
{
    var process = new ClassicProcess
    {
        Number = _nextProcessNumber++,
        Condatcs = new List<Condatc>()
    };
    return process;
}

private List<Condatc> TranspileCondition(Condition condition)
{
    // Implementar transpilación de condiciones
    return new List<Condatc>();
}

private int GetNextDelayFlag() => 100 + (_nextProcessNumber % 50);
private int GetNextErrorFlag() => 150 + (_nextProcessNumber % 50);
}

// =====
// CONTEXTO DE TRANSPILACIÓN SECUENCIAL
// =====

public class SequentialContext
{
    public ClassicProcess CurrentProcess { get; set; }
    public List<SequentialStatement> FinallyStatements { get; set; } = new();
    public SequentialStatement? ErrorHandler { get; set; }
    public int ErrorFlag { get; set; }
    public Dictionary<string, int> Variables { get; set; } = new();
}

public class ClassicProcess
{
    public int Number { get; set; }
    public List<Condatc> Condatcs { get; set; } = new();
}

public record Condatc(string Name, int[] Parameters);

```

Integración con el Transpilador Existente

1. Modificaciones Requeridas

```
// En CompleteDaadTranspiler.cs
public partial class CompleteDaadTranspiler : ICompleteDaadTranspiler
{
    private readonly SequentialTranspiler _sequentialTranspiler;

    // Agregar soporte para secuencias
    public Task<TranspileResult> TranspileSequentialAsync(SequentialStatement
statement)
    {
        var processes = _sequentialTranspiler.TranspileSequential(statement);

        // Convertir a formato DAAD clásico
        var classicCode = GenerateClassicProcesses(processes);

        return Task.FromResult(new TranspileResult
        {
            Success = true,
            Output = classicCode,
            Statistics = new TranspileStatistics
            {
                ProcessingTime = TimeSpan.FromMilliseconds(100),
                ConductosUsed = processes.Sum(p => p.Conducts.Count)
            }
        });
    }
}
```

2. Extensión del Parser

```
// En DaadParser.cs
public static class SequentialParser
{
    public static Parser<char, SequentialStatement> SequentialStatement =>
        ChainedStatement.Or(SimpleStatement.Cast<SequentialStatement>());

    private static Parser<char, SequentialStatement> ChainedStatement =>
        SimpleStatement
            .Then(SequenceOperator, (first, op) =>
                SequentialStatement.Select(second =>
                    new ChainedSequentialStatement(first, op, second)))
            .Cast<SequentialStatement>();

    private static Parser<char, SequenceOperator> SequenceOperator =>
        OneOf(
            String("then").ThenReturn(SequenceOperator.Then),
            String("after").ThenReturn(SequenceOperator.After),
            String("next").ThenReturn(SequenceOperator.Next),
            String("finally").ThenReturn(SequenceOperator.Finally),
            String("on_error").ThenReturn(SequenceOperator.OnError)
        )
    }
}
```




```
}  
    );
```

Análisis de Impacto

Ventajas de la Implementación

1. ☒ **Expresividad Mejorada**
 - Código más legible y mantenible
 - Secuencias complejas naturales
 - Manejo de errores robusto
2. ☒ **Compatibilidad Total**
 - Transpilación a DAAD clásico
 - Sin cambios en el runtime
 - Mantiene todas las características existentes
3. ☒ **Extensibilidad**
 - Fácil agregar nuevos operadores
 - Soporte para patrones complejos
 - Integración con sistemas existentes

Desafíos y Limitaciones

1.  **Complejidad del Código Generado**
 - Múltiples procesos DAAD clásicos
 - Uso intensivo de flags temporales
 - Debugging más complejo
2.  **Limitaciones de DAAD Clásico**
 - Número limitado de procesos
 - Flags limitados
 - No hay true async support
3.  **Performance**
 - Overhead de múltiples procesos
 - Uso de memoria adicional
 - Complejidad de transpilación

Recomendaciones

Fase 1: Implementación Básica (Inmediata)

1. Implementar **then** y **finally**

- Operadores secuenciales básicos
- Transpilación a procesos DAAD clásicos
- Testing básico

2. Soporte para Condicionales Secuenciales

- `if...then...else` mejorado
- Cadenas de condiciones
- Manejo básico de errores

Fase 2: Características Avanzadas (Futuro)

1. Loops y Iteración

- `while, for, repeat`
- Control de flujo avanzado
- Optimizaciones de rendimiento

2. Manejo de Errores Robusto

- `try...catch` completo
- Recovery automático
- Logging de errores

3. Características Asíncronas

- `async` statements
- Timeouts y delays
- Callbacks y eventos

Conclusión

La implementación de sentencias complejas secuenciales en DAAD Moderno es **técnicamente viable** y **altamente beneficiosa** para la expresividad del lenguaje.

Estado actual: ✗ NO IMPLEMENTADO

Prioridad: ● ALTA

Complejidad: ● MEDIA-ALTA

Impacto: ● ALTO

La propuesta técnica muestra un camino claro hacia la implementación completa, manteniendo compatibilidad total con DAAD clásico mientras añade capacidades modernas de programación secuencial.

Estudio generado: 17 de Julio de 2025

Autor: Sistema de Análisis DAAD Moderno

Versión: 1.0.0