SLD-Explorer: Guía Completa de Implementación

📚 Índice

🎯 Visión General {#vision-general}

SLD-Explorer es un visualizador interactivo de árboles de resolución SLD en Prolog. El proyecto permite:

- Cargar programas Prolog
- Ejecutar consultas
- Visualizar el proceso de resolución paso a paso
- Entender cómo funciona el backtracking

Conceptos Clave

SLD Resolution (Selective Linear Definite):

- Sistema de demostración usado por Prolog
- Estrategia: depth-first, left-to-right
- Cada nodo = estado de la computación
- Cada rama = aplicación de una regla

Estructura del Árbol:

```
Nodo = (Goals, Substitution, RuleApplied, Status, Children)
```

🏗️ Arquitectura del Sistema {#arquitectura}

Módulos Prolog (Backend)

```
prolog/
├── core/
│   ├── unification.pl      ✅ Implementado
│   ├── sld_engine.pl       ✅ Implementado
│   └── substitution.pl     (parte de unification.pl)
│
├── tree/
│   ├── tree_builder.pl     (parte de sld_engine.pl)
│   └── tree_export.pl      ✅ Implementado
│
├── extensions/
│   ├── cut_handler.pl      📋 Por implementar
│   └── strategies.pl       📋 Por implementar
│
└── main.pl                 ✅ Implementado
```

Módulos Web (Frontend)

```
web/
├── index.html
├── css/
│   └── styles.css
└── js/
    ├── main.js
    ├── tree-visualizer.js  (D3.js)
    └── api-client.js
```

🚀 Fase 1: MVP (Core) {#fase-1-mvp}

✅ Completado

1. Módulo de Unificación (`unification.pl`)

Funcionalidades:

- Algoritmo de unificación de Robinson
- Occurs check para evitar ciclos
- Aplicación de sustituciones
- Composición de sustituciones

Uso:

```prolog
?- unify(f(X, b), f(a, Y), Subst).
Subst = [X=a, Y=b].

?- apply_substitution(g(X, Y), [X=a, Y=b], Result).
Result = g(a, b).
```

2. Motor SLD (`sld_engine.pl`)

Funcionalidades:

- Construcción completa del árbol SLD
- Detección de éxito/fallo
- Límite de profundidad (evita bucles infinitos)
- Análisis del árbol (contar nodos, profundidad, soluciones)

Uso:

```prolog
?- sld_resolution(Program, [grandparent(tom, X)], Tree).
```

3. Exportación (`tree_export.pl`)

Formatos soportados:

- JSON (para visualización web)
- DOT/Graphviz (alternativa)

Uso:

```prolog
?- export_tree_file(Tree, 'output.json').
```

4. Sistema Principal (`main.pl`)

Comandos disponibles:

```prolog
```

```
run_examples.          % Todos los ejemplos
example_family.        % Relaciones familiares
example_member.        % Member de listas
example_append.        % Append
example_backtracking.  % Ejemplo con backtracking
explore_query(File, Q). % Consulta personalizada
```

📝 Pruebas del MVP

Test 1: Instalación

```
bash

# Desde el directorio del proyecto
swipl -s main.pl
```

Deberías ver:

```
=== SLD Explorer Loaded ===
Quick commands:
  run_examples.          - Run all built-in examples
  ...
```

Test 2: Ejemplo Simple

```
prolog

?- example_family.
```

Output esperado:

```
=== Example: Family Relations ===
=== Tree Statistics ===
Total nodes: 7
Tree depth: 4
Solutions found: 2

=== Solutions ===
Solution 1: {W=ann}
Solution 2: {W=pat}
```

Test 3: Exportación

```
prolog
```

```
?- example_family,
   sld_resolution([...], [grandparent(tom,W)], Tree),
   export_tree_file(Tree, 'family.json').
```

🌐 Fase 2: Interfaz Web {#fase-2-web}

Objetivos

1. Visualización interactiva del árbol SLD
2. Editor de programas Prolog
3. Input de consultas
4. Animación paso a paso
5. Exportación de imágenes

Tecnologías

- **D3.js**: Visualización de grafos
- **CodeMirror**: Editor de código Prolog
- **SWI-Prolog HTTP Server**: Bridge Prolog-Web

Estructura HTML Base

```
html
```

```
<!DOCTYPE html>
<html>
<head>
  <title>SLD Explorer</title>
  <link rel="stylesheet" href="css/styles.css">
</head>
<body>
  <div id="app">
    <div id="editor-panel">
      <h2>Prolog Program</h2>
      <textarea id="program-editor"></textarea>
      <h2>Query</h2>
      <input id="query-input" type="text" />
      <button id="run-btn">Generate SLD Tree</button>
    </div>

    <div id="visualization-panel">
      <div id="controls">
        <button id="step-back">◀ Step Back</button>
        <button id="step-forward">Step Forward ▶</button>
        <button id="auto-play">▶️ Auto Play</button>
      </div>
      <svg id="tree-canvas"></svg>
    </div>

    <div id="info-panel">
      <h3>Current Node</h3>
      <div id="node-details"></div>
    </div>
  </div>

  <script src="https://d3js.org/d3.v7.min.js"></script>
  <script src="js/api-client.js"></script>
  <script src="js/tree-visualizer.js"></script>
  <script src="js/main.js"></script>
</body>
</html>
```

Bridge Prolog-Web

Opciones:

Opción A: SWI-Prolog HTTP Server

```
prolog
```

```
:- use_module(library(http/thread_httpd)).
:- use_module(library(http/http_dispatch)).
:- use_module(library(http/http_json)).

:- http_handler('/api/explore', handle_explore, []).

server(Port) :-
    http_server(http_dispatch, [port(Port)]).

handle_explore(Request) :-
    http_read_json_dict(Request, Dict),
    % Procesar programa y consulta
    Program = Dict.program,
    Query = Dict.query,
    % Generar árbol
    sld_resolution(Program, Query, Tree),
    tree_to_json(Tree, JSON),
    reply_json_dict(JSON).
```

Opción B: Script Python intermediario

python

```
from pyswip import Prolog
from flask import Flask, request, jsonify

app = Flask(__name__)
prolog = Prolog()
prolog.consult("main.pl")

@app.route('/api/explore', methods=['POST'])
def explore():
    data = request.json
    program = data['program']
    query = data['query']

    # Ejecutar consulta en Prolog
    result = list(prolog.query(f"explore_query_json('{program}', {query}, Tree)"))

    return jsonify(result[0]['Tree'])
```

Visualización D3.js

javascript

```
class SLDTreeVisualizer {
    constructor(svgElement) {
        this.svg = d3.select(svgElement);
        this.width = 1200;
        this.height = 800;
        this.tree = null;
    }

    loadTree(treeData) {
        this.tree = treeData;
        this.render();
    }

    render() {
        // Crear layout jerárquico
        const treeLayout = d3.tree()
            .size([this.width - 100, this.height - 100]);

        // Convertir a jerarquía D3
        const root = d3.hierarchy(this.tree, d => d.children);

        // Calcular posiciones
        treeLayout(root);

        // Dibujar enlaces
        this.svg.selectAll('.link')
            .data(root.links())
            .enter()
            .append('path')
            .attr('class', 'link')
            .attr('d', d3.linkVertical()
                .x(d => d.x)
                .y(d => d.y));

        // Dibujar nodos
        const nodes = this.svg.selectAll('.node')
            .data(root.descendants())
            .enter()
            .append('g')
            .attr('class', 'node')
            .attr('transform', d => `translate(${d.x},${d.y})`);

        nodes.append('circle')
            .attr('r', 8)
            .attr('fill', d => this.getNodeColor(d.data.status));
```

```
        nodes.append('text')
            .attr('dy', -10)
            .text(d => d.data.goals);
    }

    getNodeColor(status) {
        const colors = {
            'success': '#90EE90',
            'failure': '#FFB6C1',
            'has_solution': '#FFFFE0',
            'pending': '#ADD8E6'
        };
        return colors[status] || '#CCCCCC';
    }

    animateStep(nodeId) {
        // Highlight del nodo actual
        this.svg.selectAll('.node')
            .filter(d => d.data.id === nodeId)
            .select('circle')
            .transition()
            .duration(500)
            .attr('r', 12)
            .attr('stroke', 'orange')
            .attr('stroke-width', 3);
    }
}
```

Cliente API

javascript

```
class SLDAPIClient {
    constructor(baseURL) {
        this.baseURL = baseURL;
    }

    async exploreQuery(program, query) {
        const response = await fetch(`${this.baseURL}/api/explore`, {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({program, query})
        });
        return response.json();
    }
}
```

🔧 Fase 3: Extensiones {#fase-3-extensiones}

1. Manejo de Cortes (`cut_handler.pl`)

Objetivo: Visualizar el efecto del operador de corte `!`

```prolog
% Modificar sld_engine.pl para detectar cortes
expand_sld_tree_with_cut(Program, [!|RestGoals], Subst, Rule, Depth, Node) :-
    % Marcar que se ha encontrado un corte
    Node = node([!|RestGoals], Subst, Rule, cut_encountered, [
        % Continuar con RestGoals sin explorar alternativas
        ChildNode
    ]),
    expand_sld_tree_with_cut(Program, RestGoals, Subst, cut_rule, Depth, ChildNode).
```

Visualización:

- Nodos de corte en color especial (rojo/naranja)
- Mostrar ramas "podadas"
- Explicación textual del efecto

2. Estrategias Alternativas (`strategies.pl`)

Breadth-First:

```prolog
sld_breadth_first(Program, Query, Tree) :-
    % Cola de nodos por explorar
    Queue = [initial_node(Query)],
    expand_bfs(Program, Queue, [], Tree).

expand_bfs(_, [], Explored, Tree) :-
    % Reconstruir árbol desde nodos explorados
    build_tree_from_explored(Explored, Tree).

expand_bfs(Program, [Node|Rest], Explored, Tree) :-
    expand_node(Program, Node, Children),
    append(Rest, Children, NewQueue),
    expand_bfs(Program, NewQueue, [Node|Explored], Tree).
```

Comparador de estrategias:

```prolog
compare_strategies(Program, Query) :-
    format('~n=== Depth-First ===~n'),
    sld_resolution(Program, Query, DFTree),
    print_tree_stats(DFTree),

    format('~n=== Breadth-First ===~n'),
    sld_breadth_first(Program, Query, BFTree),
    print_tree_stats(BFTree).
```

3. Debugging Features

Step-by-step execution:

```prolog
sld_step(Program, CurrentState, NextState) :-
    CurrentState = state(Goals, Subst, Depth),
    % Aplicar una sola regla
    Goals = [Goal|Rest],
    member(clause(Head, Body), Program),
    unify(Goal, Head, Subst, NewSubst),
    append(Body, Rest, NewGoals),
    NextState = state(NewGoals, NewSubst, Depth+1).
```

Breakpoints:

```prolog
% Pausar cuando se alcanza cierto goal
sld_with_breakpoint(Program, Query, Breakpoint, Tree) :-
    sld_resolution_breakpoint(Program, Query, Breakpoint, Tree).
```

🧪 Testing {#testing}

Tests Unitarios

```prolog
```

```
:- begin_tests(unification).

test(unify_variables) :-
    unify(X, a, [], Subst),
    Subst = [X=a].

test(unify_compound) :-
    unify(f(X, b), f(a, Y), [], Subst),
    member(X=a, Subst),
    member(Y=b, Subst).

test(occurs_check_fail) :-
    \+ unify(X, f(X), [], _).

:- end_tests(unification).

:- begin_tests(sld_engine).

test(simple_success) :-
    Program = [clause(p(a), [])],
    Query = [p(a)],
    sld_resolution(Program, Query, Tree),
    Tree = node([], _, _, success, _).

test(simple_failure) :-
    Program = [clause(p(a), [])],
    Query = [p(b)],
    sld_resolution(Program, Query, Tree),
    Tree = node([p(b)], _, _, failure, []).

test(backtracking) :-
    Program = [
        clause(p(a), []),
        clause(p(b), [])
    ],
    Query = [p(X)],
    sld_resolution(Program, Query, Tree),
    find_all_solutions(Tree, Solutions),
    length(Solutions, 2).

:- end_tests(sld_engine).
```

Ejecutar tests

```
bash
```

```
swipl -g "run_tests." -t halt main.pl
```

Tests de Integración

```prolog
integration_test_1 :-
    % Test: cargar programa, ejecutar consulta, exportar JSON
    load_program('examples/family.pl', Program),
    Query = [grandparent(X, Y)],
    sld_resolution(Program, Query, Tree),
    tree_to_json(Tree, JSON),
    JSON \= null.
```

🗺️ Roadmap {#roadmap}

✅ Semana 1-2: MVP

- [x] Módulo de unificación
- [x] Motor SLD básico
- [x] Exportación JSON
- [x] Sistema principal con ejemplos

📋 Semana 3: Interfaz Web

- [ ] HTML/CSS básico
- [ ] Integración D3.js
- [ ] Bridge Prolog-Web
- [ ] Editor de código

📋 Semana 4: Features Avanzadas

- [ ] Animación paso a paso
- [ ] Manejo de cortes
- [ ] Estrategias alternativas
- [ ] Exportación de imágenes

📋 Semana 5: Pulido

- [ ] Testing exhaustivo
- [ ] Documentación
- [ ] Optimizaciones
- [ ] Deploy

📖 Recursos Adicionales

Bibliografía

1. **"The Art of Prolog"** - Sterling & Shapiro (Cap. 15: SLD Resolution)
2. **"Programming in Prolog"** - Clocksin & Mellish
3. **SWI-Prolog Documentation**: https://www.swi-prolog.org/

Papers

- Robinson, J.A. (1965). "A Machine-Oriented Logic Based on the Resolution Principle"
- Apt, K.R. (1997). "From Logic Programming to Prolog"

Herramientas

- **Graphviz**: https://graphviz.org/
- **D3.js Gallery**: https://observablehq.com/@d3/gallery
- **CodeMirror**: https://codemirror.net/

🎓 Valor Académico

Este proyecto demuestra comprensión profunda de:

1. **Semántica operacional de Prolog**
2. **Meta-programación** (meta-intérprete)
3. **Algoritmos clásicos de IA** (unificación, resolución)
4. **Estructuras de datos funcionales** (árboles inmutables)
5. **Visualización de conceptos abstractos**

Perfecto para: Proyecto final de programación declarativa, lógica computacional, o IA simbólica.

🚀 Siguientes Pasos

1. **Ejecutar MVP:**

```
bash
```

```
swipl -s main.pl
?- run_examples.
```

2. **Experimentar con ejemplos propios:**

```prolog
?- explore_query('mi_programa.pl', [mi_consulta(X)]).
```

3. **Comenzar interfaz web** (siguiente fase)
4. **Probar exportaciones:**

```prolog
?- example_family,
   sld_resolution(..., Tree),
   tree_to_dot(Tree, DotCode),
   open('tree.dot', write, S),
   write(S, DotCode),
   close(S).
```

Luego: `dot -Tpng tree.dot -o tree.png`

¡Éxito con tu proyecto! 🎉