# SPL Static Semantic Checker - Technical Specification

## Table of Contents



## System Overview

### Purpose

Build a static semantic checker that validates SPL (Simple Programming Language) programs against name-scope rules by traversing syntax trees and detecting naming conflicts.

### Core Functionality

- Parse SPL source code into syntax trees
- Assign unique identifiers to all tree nodes
- Traverse trees systematically to identify scopes
- Enforce two primary semantic rules:
    1. No name conflicts between variables, functions, and procedures
    2. No duplicate variable declarations within same scope
- Generate comprehensive error reports for violations

### Technology Stack

- **Primary Language**: [To be determined by team]
- **Symbol Table**: Hash Table or Relational Database
- **Data Structures**: Trees, Hash Maps, Sets
- **Testing Framework**: [To be determined by team]

# Architecture Design

## System Flow


## Component Interaction Model

- **Layered Architecture**: Infrastructure → Core Logic → Presentation
- **Observer Pattern**: Error reporting observes rule violations
- **Visitor Pattern**: Tree traversal with pluggable node processors
- **Repository Pattern**: Symbol table abstraction

# Core Infrastructure Components

## 1. Syntax Tree Components

### AST Node Structure

Class: ASTNode
Attributes:
- nodeId: UniqueIdentifier
- nodeType: NodeType (enum)
- parent: ASTNode (nullable)
- children: List<ASTNode>
- sourcePosition: SourceLocation
- scopeInfo: ScopeInformation

Methods:
- getId(): UniqueIdentifier
- getType(): NodeType
- addChild(node: ASTNode): void
- getChildren(): List<ASTNode>
- accept(visitor: NodeVisitor): void

### Node Types Enum

Enum: NodeType
Values:
- SPL_PROG
- VARIABLES
- VAR
- PROCDEFS
- FUNCDEFS
- MAINPROG
- USER_DEFINED_NAME

### Node ID Generator

Class: NodeIdGenerator
Attributes:
- currentId: Integer (static)

Methods:
- generateId(): UniqueIdentifier
- reset(): void (for testing)

### Tree Builder

Class: SyntaxTreeBuilder
Methods:
- buildTree(source: String): ASTNode
- parseNode(tokenStream: TokenStream): ASTNode
- linkParentChild(parent: ASTNode, child: ASTNode): void

## 2. Symbol Table System

### Symbol Entry Record

Class: SymbolEntry
Attributes:
- name: String
- symbolType: SymbolType
- scopeType: ScopeType
- nodeId: UniqueIdentifier
- declarationLocation: SourceLocation
- isUsed: Boolean

Methods:
- getName(): String
- getType(): SymbolType
- getScope(): ScopeType
- equals(other: SymbolEntry): Boolean

### Symbol Types

Enum: SymbolType
Values:
- VARIABLE
- FUNCTION
- PROCEDURE

### Scope Types

Enum: ScopeType
Values:
- EVERYWHERE
- GLOBAL
- PROCEDURE_SCOPE
- FUNCTION_SCOPE
- MAIN_SCOPE

### Symbol Table Implementation

Class: SymbolTable
Attributes:
- symbolMap: HashMap<String, List<SymbolEntry>>
- nodeToSymbol: HashMap<UniqueIdentifier, SymbolEntry>
- scopeToSymbols: HashMap<ScopeType, Set<SymbolEntry>>

Methods:
- insert(entry: SymbolEntry): Boolean
- lookup(name: String): List<SymbolEntry>
- lookupByScope(name: String, scope: ScopeType): SymbolEntry
- getSymbolsInScope(scope: ScopeType): Set<SymbolEntry>
- removeSymbol(nodeId: UniqueIdentifier): Boolean
- clear(): void

## 3. Tree Traversal Engine

### Tree Walker Interface

Interface: TreeWalker
Methods:
- traverse(root: ASTNode, visitor: NodeVisitor): void
- setTraversalOrder(order: TraversalOrder): void

### Node Visitor Interface

Interface: NodeVisitor
Methods:
- visitNode(node: ASTNode): VisitResult
- enterScope(node: ASTNode): void
- exitScope(node: ASTNode): void

### Traversal Implementation

Class: DepthFirstWalker implements TreeWalker
Methods:
- traverse(root: ASTNode, visitor: NodeVisitor): void
- traverseRecursive(node: ASTNode, visitor: NodeVisitor): void

# Scope Management System

## 4. Scope Detection System

### Scope Detector

Class: ScopeDetector implements NodeVisitor
Attributes:
- currentScope: ScopeStack
- symbolTable: SymbolTable

Methods:
- detectScope(node: ASTNode): ScopeType
- visitNode(node: ASTNode): VisitResult
- enterScope(node: ASTNode): void
- exitScope(node: ASTNode): void

### Scope Stack

Class: ScopeStack
Attributes:
- scopes: Stack<ScopeContext>

Methods:
- push(scope: ScopeContext): void
- pop(): ScopeContext
- peek(): ScopeContext
- getCurrentScope(): ScopeType
- getDepth(): Integer

### Scope Context

Class: ScopeContext
Attributes:
- scopeType: ScopeType
- scopeNode: ASTNode
- symbols: Set<String>
- parentScope: ScopeContext

Methods:
- addSymbol(name: String): void
- hasSymbol(name: String): Boolean
- getSymbolCount(): Integer

## 5. Scope Hierarchy Management

### Scope Rules Matrix

Scope Relationships:
EVERYWHERE
├── GLOBAL (variables)
├── PROCEDURE_SCOPE (procedures)
├── FUNCTION_SCOPE (functions)
└── MAIN_SCOPE (main program)

Inheritance Rules:
- All scopes inherit from EVERYWHERE
- No cross-inheritance between GLOBAL, PROCEDURE_SCOPE, FUNCTION_SCOPE, MAIN_SCOPE

# Rule Enforcement Engine

## 6. Semantic Rule Checker

### Rule Validator Interface

Interface: RuleValidator
Methods:
- validate(symbolTable: SymbolTable): List<RuleViolation>
- getRuleName(): String
- getRulePriority(): Integer

### Everywhere Scope Rule

Class: EverywhereConflictRule implements RuleValidator
Methods:
- validate(symbolTable: SymbolTable): List<RuleViolation>
- checkVariableFunctionConflict(): List<RuleViolation>
- checkVariableProcedureConflict(): List<RuleViolation>
- checkFunctionProcedureConflict(): List<RuleViolation>

### Variables Scope Rule

Class: VariableDuplicateRule implements RuleValidator
Methods:
- validate(symbolTable: SymbolTable): List<RuleViolation>
- findDuplicatesInScope(scope: ScopeType): List<RuleViolation>

### Rule Violation Record

Class: RuleViolation
Attributes:
- ruleType: RuleType
- conflictingSymbols: List<SymbolEntry>
- errorMessage: String
- severity: ErrorSeverity
- sourceLocation: SourceLocation

Methods:
- getDescription(): String
- getSeverity(): ErrorSeverity
- getLocation(): SourceLocation

## 7. Conflict Detection Algorithms

### Name Conflict Detector

Class: ConflictDetector
Methods:
- findConflicts(symbols: List<SymbolEntry>): List<Conflict>
- groupByName(symbols: List<SymbolEntry>): Map<String, List<SymbolEntry>>
- analyzeGroup(group: List<SymbolEntry>): List<Conflict>

# Error Management System

## 8. Error Types and Classification

### Error Severity

Enum: ErrorSeverity
Values:
- ERROR (blocks compilation)
- WARNING (compilation continues)
- INFO (informational only)

### Rule Types

Enum: RuleType
Values:
- VARIABLE_FUNCTION_CONFLICT
- VARIABLE_PROCEDURE_CONFLICT
- FUNCTION_PROCEDURE_CONFLICT
- DUPLICATE_VARIABLE_DECLARATION

### Error Reporter

Class: ErrorReporter
Attributes:
- violations: List<RuleViolation>
- errorCount: Integer
- warningCount: Integer

Methods:
- addViolation(violation: RuleViolation): void
- generateReport(): ErrorReport
- hasErrors(): Boolean
- getErrorCount(): Integer
- clearErrors(): void

## 9. Error Message Generation

### Message Templates

Templates:
- Variable-Function Conflict: "Variable '{name}' conflicts with function '{name}' in {scope} scope"
- Variable-Procedure Conflict: "Variable '{name}' conflicts with procedure '{name}' in {scope} scope"
- Function-Procedure Conflict: "Function '{name}' conflicts with procedure '{name}' in {scope} scope"
- Duplicate Variable: "Variable '{name}' declared multiple times in {scope} scope"

### Error Formatter

Class: ErrorFormatter
Methods:
- formatViolation(violation: RuleViolation): String
- formatSummary(violations: List<RuleViolation>): String
- formatLocation(location: SourceLocation): String

# Utility Components

## 10. Supporting Data Structures

### Source Location

Class: SourceLocation
Attributes:
- line: Integer
- column: Integer
- filename: String

Methods:
- toString(): String
- compareTo(other: SourceLocation): Integer

### Configuration Manager

Class: Configuration
Attributes:
- caseSensitive: Boolean
- stopOnFirstError: Boolean
- maxErrorCount: Integer
- verboseOutput: Boolean

Methods:
- loadConfig(file: String): void
- setProperty(key: String, value: Object): void
- getProperty(key: String): Object

## 11. Helper Utilities

### String Utilities

Class: StringUtils
Methods:
- normalize(name: String): String
- compareNames(name1: String, name2: String): Boolean
- isValidIdentifier(name: String): Boolean

### Collection Utilities

Class: CollectionUtils
Methods:
- findDuplicates<T>(list: List<T>): List<T>
- groupBy<T, K>(list: List<T>, keyExtractor: Function<T, K>): Map<K, List<T>>
- intersection<T>(set1: Set<T>, set2: Set<T>): Set<T>

# Integration Interfaces

## 12. External Integration Points

### Parser Integration

```
Interface: ParserIntegration
Methods:
- parseToAST(source: String): ASTNode
- getParseErrors(): List<ParseError>
```

### Output Integration

```
Interface: OutputHandler
Methods:
- outputResults(report: ValidationReport): void
- setOutputFormat(format: OutputFormat): void
```

### Testing Integration

```
Interface: TestHarness
Methods:
- runTestCase(testCase: TestCase): TestResult
- validateExpectedErrors(expected: List<RuleViolation>, actual: List<RuleViolation>): Boolean
```

# Implementation Guidelines

## Design Patterns to Use

1. **Visitor Pattern**: For tree traversal and node processing
2. **Strategy Pattern**: For different rule validation strategies
3. **Observer Pattern**: For error reporting and logging
4. **Factory Pattern**: For creating different node types
5. **Repository Pattern**: For symbol table abstraction

## Performance Considerations

- **Symbol Table**: Use hash-based lookups for O(1) average case
- **Tree Traversal**: Single-pass traversal to minimize time complexity
- **Memory Management**: Clean up temporary objects during traversal
- **Caching**: Cache scope detection results for frequently accessed nodes

### Error Handling Strategy

- **Graceful Degradation**: Continue checking after finding errors
- **Error Limits**: Stop after maximum error count to prevent flooding
- **Recovery**: Attempt to recover from parse errors where possible
- **Logging**: Comprehensive logging for debugging

## Testing Strategy

### Unit Testing

- **Symbol Table Operations**: Insert, lookup, delete functionality
- **Tree Traversal**: Verify all nodes are visited exactly once
- **Rule Validation**: Test each rule with known valid/invalid cases
- **Error Reporting**: Verify error messages and formatting

### Integration Testing

- **Component Interaction**: Test data flow between components
- **End-to-End**: Complete SPL programs through entire pipeline
- **Error Scenarios**: Programs with multiple types of violations

### Test Data Categories

1. **Valid Programs**: No violations, should pass all checks
2. **Single Violations**: One type of error each
3. **Multiple Violations**: Complex programs with various errors
4. **Edge Cases**: Empty programs, minimal programs, deeply nested scopes

### Sample Test Cases

Test Case 1: Variable-Function Conflict
SPL Code:
glob { var x; }
func { function x; }

Expected: Variable 'x' conflicts with function 'x' in Everywhere scope

Test Case 2: Duplicate Variables
SPL Code:
glob { var x; var x; }

Expected: Variable 'x' declared multiple times in Global scope

Test Case 3: Valid Program
SPL Code:
glob { var x; }
func { function y; }
proc { procedure z; }
main { /* main code */ }

Expected: No violations

## Team Task Assignment Matrix

### Phase 1: Foundation (Days 1-3)

| Component | Primary Owner | Support |
|---|---|---|
| AST Node Structure | Member 2 | Member 1 |
| Symbol Table Core | Member 1 | Member 3 |
| Scope Detection | Member 3 | Member 2 |
| Error Framework | Member 4 | All |

### Phase 2: Core Logic (Days 4-7)

| Component | Primary Owner | Support |
|---|---|---|
| Tree Traversal | Member 2 | Member 3 |
| Symbol Table Operations | Member 1 | Member 4 |
| Rule Validation | Member 3 | Member 1 |
| Error Reporting | Member 4 | Member 2 |

### Phase 3: Integration (Days 8-10)

| Task | Primary Owner | Support |
|---|---|---|
| Component Integration | Member 2 | All |
| Rule Engine Integration | Member 3 | Member 1 |
| Error System Integration | Member 4 | Member 2 |
| Symbol Table Integration | Member 1 | Member 3 |

### Phase 4: Testing & Finalization (Days 11-14)

| Task | Primary Owner | Support |
|---|---|---|
| Unit Testing | Member 4 | All |
| Integration Testing | All | Member 4 |
| Performance Testing | Member 1 | Member 2 |

| Task | Primary Owner | Support |
|---|---|---|
| Documentation | All | Member 4 |

### Daily Coordination Points

- **Daily Standup**: 15 minutes, progress and blockers
- **Integration Sessions**: 1 hour every other day
- **Code Reviews**: Peer review all commits
- **Testing**: Continuous testing throughout development

## Success Criteria

### Functional Requirements

- ✅ Correctly identifies all name conflicts per SPL rules
- ✅ Generates accurate error messages with source locations
- ✅ Handles all valid SPL programs without false positives
- ✅ Processes syntax trees efficiently

### Non-Functional Requirements

- ✅ Processes programs up to 1000 lines in under 5 seconds
- ✅ Memory usage scales linearly with program size
- ✅ Modular design allows easy rule additions
- ✅ Comprehensive test coverage (>90%)

### Deliverables

1. **Working semantic checker** with all required functionality
2. **Comprehensive test suite** with >50 test cases
3. **Technical documentation** for future maintenance
4. **Demo program** showing all major features