SOCKit Reference Manual

Quentin Autry	`System Architect`
Hakim El Ghazi	`Float Goat`
Ryan Najac	`Manager`
Misha Smirnov	`Language Guru`
Stacey Yao	`Tester`

Columbia University

COMS W4115

Programming Languages and Translators

March 22, 2024

Contents

1 Overview

Sockit integrates the structured syntax of C with scripting conveniences, emphasizing Unix-like operations, concurrency, and streamlined data handling. It is optimized for system-level tasks, offering an efficient pathway for both compiled and script-like programming paradigms.

2 Lexical Conventions

The lexical components of Sockit include identifiers, keywords, constants, string literals, various operators, and separators. Spaces, tabs, and newlines serve to delineate tokens, except when their presence is integral to token formation.

2.1 Comments

Sockit supports block (`/* ... */`) and line (`//`) comments, akin to C. Unlike C, Sockit allows comments to nest, facilitating more flexible code annotation.

```
/* Block comment example */
// Line comment example
/* A comment /* nested within */ another */
```

2.2 Identifiers

Identifiers begin with a letter or underscore, followed by any combination of letters, digits, and underscores. Sockit treats uppercase and lowercase letters as distinct.

```
int identifier1;
data _dataIdentifier;
```

3 Keywords and Constants

Reserved keywords in Sockit are used to denote control structures, data types, and other language constructs. Sockit enhances type simplicity and introduces scripting-like conveniences.

Sockit supports integral constants in various bases: decimal, hexadecimal (prefixed by `0x`), and binary (prefixed by `0b`).

```
int dec = 10;
int hex = 0x1A;
int bin = 0b1010;
```

3.1 Special Keywords

`data` serves as a generic pointer type, akin to void pointers but with an emphasis on data-agnostic operations. `null` represents the absence of data.

```
data genericData = null;
```

4 Concurrency and Atomic Operations

4.1 Threads

Sockit introduces constructs for managing concurrent execution flows, simplifying the use of POSIX threads.

```
thread t = thread_create(function, arg);
thread_join(t);
```

4.2 Atomics

Atomic variables and operations ensure data integrity across concurrent executions.

```
@int atomicCounter = 0;
atomic_increment(&atomicCounter);
```

Example:

```
statement ::= if_statement | while_statement | for_statement;
if_statement ::= 'if' '(' expression ')' statement ['else' statement];
```

5 Syntax Summary

The syntax of Sockit is designed to support both the structured programming model of C and the dynamic, scripting capabilities familiar to Unix shell users. This summary provides an overview of the syntactic elements that constitute the Sockit language.

5.1 Basic Syntax

- `statements` include declarations, assignments, control flow statements (`if`, `for`, `while`, `switch`), and function calls.
- `expression` encompasses operations among variables, literals, and function calls that return values. Expressions are used for calculation, logic operations, and to determine flow control.

5.2 Function Declaration and Invocation

```
return_type function_name(parameters) {
    // function body
}

function_name(arguments);
```

5.3 Control Structures

- `if (condition)` : Executes a block of statements if the condition is true.
- `for (initialization; condition; increment)` : Executes a loop that continues as long as the condition evaluates to true.
- `while (condition)` : Continues to execute a block of statements as long as the condition remains true.

5.4 Concurrent Execution and Atomics

```
thread t = thread_create(function, arg);
thread_join(t);

@int atomicVar;
atomic_increment(&atomicVar);
```

5.5 Data Types and Variables

```
int main() {
    int integerVar = 10;
    data ptr = null;
    @int atomicInt = 0;
}
```

APPENDIX

Unix-like Scripting Features

Sockit's syntax extends traditional C by incorporating Unix-like scripting features for more dynamic programming:

```
command1 | command2; // Pipes the output of command1 to command2
data result = `command`; // Executes a shell command and captures its output
```

Background Process Execution

```
run_background_process(command);
```

Enables execution of long-running or daemon processes in the background, without blocking the main execution thread.

Advanced Atomic Operations

```
@int atomicVar;
atomic_compare_exchange(&atomicVar, expected, desired);
```

Provides mechanisms for compare-and-swap operations, crucial for lock-free concurrent algorithms.

This appendix highlights Sockit's unique features, designed to bridge the gap between structured and script-based programming within system-level applications.

Syntax Summary

Expressions and Operations

```
expression:
    primary
    * expression                    // multiplication
    / expression                    // division
    + expression                    // addition
    - expression                    // subtraction
    expression ? expression : expression // conditional
    expression == expression        // equality
    expression != expression        // inequality
    expression >= expression        // greater than or equal
    expression <= expression        // less than or equal
    expression > expression         // greater than
    expression < expression         // less than
    expression && expression        // logical AND
    expression || expression        // logical OR
    !expression                     // logical NOT
    expression & expression         // bitwise AND
    expression | expression         // bitwise OR
    expression ^ expression         // bitwise XOR
    ~expression                     // bitwise NOT
    expression << expression        // left shift
    expression >> expression        // right shift
```

Data Types and Declarations

```
types:
    int
    float
    bool
    char
    data // Special Sockit type for generic data handling

declarations:
    type specifier [list of variables];
    data variable_name; // Generic data variable declaration
```

Unix-like Features

```
// Data pipelining
command1 | command2; // Pipe output of command1 to command2

// Redirection
command > file;    // Redirect standard output to file
command < file;    // Redirect file to standard input of command
command >> file;   // Append standard output to file

// Background process
command &;         // Execute command in background
```

Concurrency and Atomics

```
// Atomic operations (indicated by @ prefix)
@int atomic_var;                // Declaration of atomic integer
@atomic_var++;                  // Atomic increment
@atomic_var--;                  // Atomic decrement
@atomic_compare_exchange(&atomic_var, expected, desired);
```

Control Structures

```
/* Standard C control structures are supported,
```

```
 * including if-else, while, switch-case
 */

if (expression) {
    // branch 1
} else {
    // branch 2
}

while (condition) {
    // loop body
}

switch (expression) {
    case value1:
        // case 1 actions
        break;
    case value2:
        // case 2 actions
        break;
    default:
        // default actions
}
```