SportS Programming Language Documentation

Laura Coroniti

February 2024

1 Introduction

SportS is a general-purpose programming language with a sports theme, aimed at making programming engaging and straightforward. It introduces an innovative approach to coding by integrating sports-related concepts, targeting both experienced programmers and beginners. The language simplifies learning programming through relatable sports analogies, offering a unique coding experience.

2 Running a Program

This section delves into the optimization process of the abstract syntax tree (AST), how to visualize your program's AST for better understanding and debugging, and the unique auditory feedback provided by SportS for an engaging programming experience.

2.1 AST Optimization

- **Constant Folding**: This optimization evaluates expressions at compile time rather than at runtime when all the leaves of an AST node are constants. For example, an expression like `3 SCORES 4` would be simplified to `7` during the compilation process, thereby reducing the computation required at runtime.
- **Type Conversion**: SportS automatically performs type conversion in mixed-type operations. If one operand is a float and the other an integer, the integer is converted to a float. For instance, in the operation `5 LOSES 2:0`, the integer `5` is treated as a float `5:0`.

These optimizations enhance the efficiency of SportS programs by minimizing runtime and conserving computational resources.

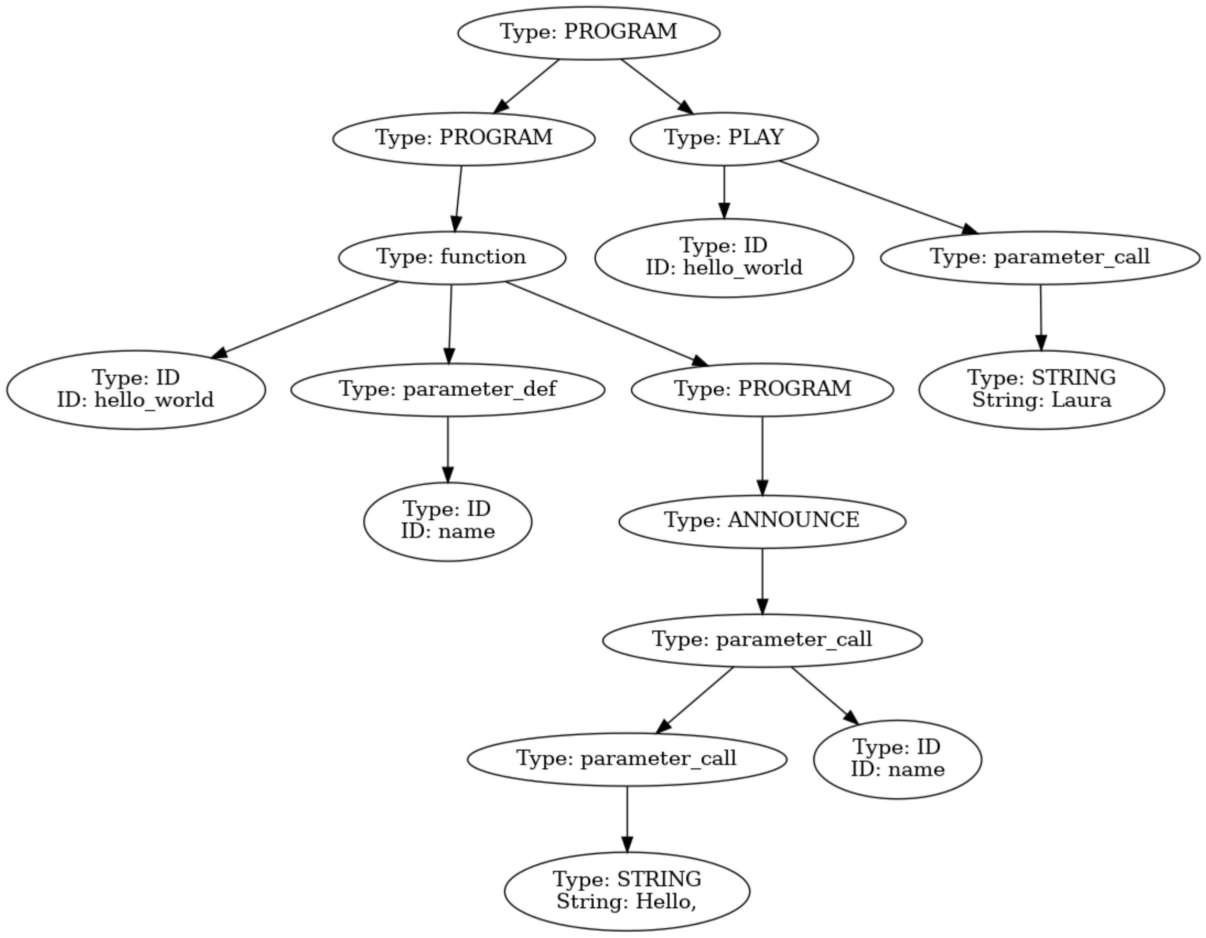
2.2 Visualizing the AST

Understanding the structure of your program is made easier with SportS's ability to visualize the Abstract Syntax Tree (AST). This feature is invaluable for debugging and comprehending the complex code.

To visualize your program's AST, SportS generates a file named `ast.gv`. You can convert this file into a visual representation using the Graphviz tool with the following command:

```
dot -Tpng ast.gv -o ast.png
```

Replace `ast.png` with the desired file name. Executing this command produces a PNG image that illustrates the hierarchical structure of your code, as shown below:



2.3 Auditory Feedback: The Starting Whistle

Embracing its sports theme, SportS introduces an element of fun to the programming experience by playing a whistle sound at the start of program execution.

3 Language Syntax

SportS's syntax is themed around sports terminology, making for an intuitive and engaging programming experience. The language features include:

3.1 Assignment

In SportS, the assignment of values to variables is performed using terminology that resonates with the sports theme, making the process straightforward and memorable.

- **IS**: The keyword `IS` is used for assigning values to variables. This keyword simplifies the syntax, making it clear and concise. For example, setting a variable to 5 would be written as `a IS 5`, emphasizing the action of assignment.
- **ALLSTAR**: To declare global variables, which are accessible throughout the entire program, SportS introduces the `ALLSTAR` keyword. This term suggests a variable of significant importance or utility, akin to an all-star player in sports. Variables declared as `ALLSTAR` are prefixed with an asterisk (*) to signify their global status. For instance, declaring a global variable `*teamScore` could be done with `ALLSTAR SCORE *teamScore IS 10`.

3.2 Data Types

In SportS, data types are dynamically allocated, allowing for a flexible and intuitive programming approach. This dynamic allocation means the explicit declaration of data types using keywords is optional, serving more for readability and documentation purposes. The same philosophy applies to the `ALLSTAR` keyword for global variables, where the asterisk `*` preceding a variable name is the critical indicator of its global scope.

The language introduces unique keywords for data types, each inspired by sports terminology, further enriching the programming experience with its thematic consistency:

- **SCORE**: Used for numerical values, `SCORE` accommodates both integers and floats. Integers are represented as whole numbers (e.g., `0`, `5`), while floats are depicted in a format akin to sports match results (e.g., `5:8`), where the colon (`:`) serves the role of a decimal point.
- **PLAYER**: Designates string data types, which are enclosed within pointy brackets `< >`.
- **TEAM**: Used for arrays, which are initialized with a parameter list enclosed in vertical bars `| |`. This keyword suggests a collection of elements working together, akin to a sports team. SportS allows for dynamic index assignment within arrays, where specifying an index out of bounds simply increases the array's size to accommodate the new element. Unassigned indices return `UNDEF`, indicating an undefined value.

To illustrate, consider the following example, which demonstrates the assignment of different data types in SportS:

```
SCORE a IS 1
SCORE b IS 2:7
PLAYER c IS <Hello World!>
TEAM d IS | 3 4 5 |
d # 3 IS 6
```

3.3 Control Structures

Control structures in SportS are designed to manage the flow of a program using terminology that resonates with its distinctive theme. These structures enable the execution of code blocks based on certain conditions or repeated execution, akin to conventional control flow structures found in other programming languages but with a unique twist.

- **WIN**, **TIE**, and **LOSE**: These keywords are used to implement conditional logic, similar to the `if`, `else if`, and `else` statements, respectively. A `WIN` block executes when its condition is true, a `TIE` block for subsequent conditions, and a `LOSE` block runs when none of the preceding conditions are met. SportS allows for an arbitrary number of `TIE` blocks to handle multiple conditions and an optional `LOSE` block for cases where no conditions are satisfied.

```
WIN a TRAILS 2
START_WHISTLE
    ANNOUNCE | <a TRAILS 2> |
END_WHISTLE
TIE a TRAILS_OR_TIES 3
START_WHISTLE
    ANNOUNCE | <a TRAILS_OR_TIES 3 > |
END_WHISTLE
LOSE
START_WHISTLE
    ANNOUNCE | <a LEADS 3> |
END_WHISTLE
```

- **PENALTY_SHOOTOUT**: This keyword introduces a loop structure that emulates the repetition of a penalty shootout in sports, functioning similarly to a `while` loop. The `PENALTY_SHOOTOUT` loop repeats a block of code as long as its condition remains true, providing a mechanism for executing repetitive tasks until a specific condition is met.

```
PENALTY_SHOOTOUT i TRAILS size
START_WHISTLE
    *a IS *a SCORES 1
    i IS i SCORES 1
END_WHISTLE
ANNOUNCE | *a |
```

3.4 Functions

- **STRATEGY** for defining functions.
- **USING** followed by a parameterlist for defining function parameters. parameters can be any amount. If there are more parameters given then in the definition the parameters that are too much are not used or used for the first undefined variable in the function scope. If there are less parameters given then defined the last parameters do not have a value or better said 'UNDEF'
- **START_WHISTLE** and **END_WHISTLE** to denote the beginning and end of block scopes. local variables are in the whole function scope visible
- **RESULT** for specifying the return value of a function.
- **PLAY** for invoking functions.

3.5 Functions

Functions in SportS are defined and controlled through a set of specialized keywords that facilitate the creation, parameterization, execution, and return of values from reusable code blocks.

- **STRATEGY**: This keyword is used for defining functions.
- **USING**: Followed by a parameter list, `USING` specifies the parameters that a function accepts. SportS is flexible with the number of parameters passed to a function; if more parameters are provided than defined, the excess parameters are ignored or assigned to the first undefined variable within the function's scope. Conversely, if fewer parameters are provided, the undefined parameters default to 'UNDEF', indicating an uninitialized state.
- **RESULT**: Specifies the return value of a function. This keyword is used to define what a function will output or return after its execution, allowing the function to provide a specific outcome based on its operations and inputs. But it is optional if a value is returned.
- **RESULT**: This keyword indicates the return value of a function, determining the output produced upon the function's completion. It allows functions to yield a defined outcome derived from their internal operations and the provided inputs. Returning a value is optional, making it flexible for a variety of use cases where a return value might not be necessary.
- **PLAY**: Invokes or calls a function, executing the block of code defined within the function scope. This keyword emphasizes the action of putting a strategy into play, mirroring the execution of a planned move or play in sports.

```
STRATEGY demo USING | x y |
START_WHISTLE
    a IS x SCORES y
RESULT a
END_WHISTLE
b IS PLAY demo USING | 3 4:0 |
```

3.6 Logical Operations

Logical operations in SportS utilize sports-themed keywords to perform boolean logic, enabling conditions to be evaluated in a manner consistent with typical programming languages but with a unique syntactic flavor:

- **AND_GOAL**, **OR_GOAL**, **NOT_GOAL** correspond to the logical AND, OR, and NOT operations, respectively, allowing for the combination and negation of boolean expressions.

- **INBOUNDS** and **OUTBOUNDS** are designed to determine the equivalence or difference between two values, respectively.
- **LEADS**, **TRAILS**, **LEADS_OR_TIES**, **TRAILS_OR_TIES** enable comparison operations, assessing the relationship between two values in terms of leading, trailing, or being equivalent.

Values are interpreted in a context-sensitive manner: `0` and `0:0` are considered false, while all other numerical values are true. Strings are considered true if their length is greater than 0, providing a straightforward mechanism for evaluating truthiness.

3.7 Arithmetic Operations

- The standard operations are represented by **SCORES** (addition), **LOSES** (subtraction), **MULTIPLIES** (multiplication), and **TACKLES** (division), offering a familiar set of tools for performing basic mathematical operations.
- **REMAINDER** denotes the modulo operation, which calculates the remainder of a division. However, it's important to note that modulo operations are restricted to integer operands, as they do not apply to floats or strings.
- When performing addition with at least one string operand, SportS treats the operation as concatenation, merging the string with the other operand's string representation. Other operations with a string will result in `UNDEF`, indicating an undefined or invalid operation.
- In operations involving both integers and floats, the result is automatically promoted to a float to preserve precision. This type constraint ensures that arithmetic expressions yield the most accurate results possible.

3.8 Miscellaneous

SportS includes a variety of additional features that enhance the coding experience, offering utilities for documentation, output, value manipulation, and more. These features contribute to the language's flexibility and ease of use.

- **;COACH**: Comments in SportS are introduced with the `;COACH` prefix, serving as annotations within the code to provide guidance or explanations. This notation is inspired by the role of a coach in sports, offering insights and strategies to improve understanding and performance.
- **ANNOUNCE**: Used for printing output to the console, `ANNOUNCE` supports an arbitrary number of values in a single line, facilitating versatile display options. This feature allows for concise and flexible output formatting, as demonstrated in the following example:

  ```
  ANNOUNCE | <a:> a <b:b> b |
  ```

- **SUBSTITUTE**: Facilitates the swapping of values, mirroring the action of substituting players in a sports game. This operation can be applied to variables or array elements, providing a straightforward method for exchanging values, as illustrated below:

  ```
  SUBSTITUTE d # 0 - d # 2   ;COACH Swap first and third value
  SUBSTITUTE a - b           ;COACH Swap the values of a and b
  ```

- **SIZE**: Retrieves the size of an array. The usage of `SIZE` is exemplified in the following code snippet:

  ```
  size IS PLAY SIZE USING | d |
  ```

- The **START_WHISTLE** and **END_WHISTLE** keywords are crucial for defining the start and end of code blocks across various structures, such as functions, loops, and conditional constructs. By marking these boundaries, these keywords ensure that local variables declared within a function are visible and accessible throughout the entirety of that function's scope, regardless of where they are defined. Furthermore, `START_WHISTLE` and `END_WHISTLE` are also necessary for delimitation of loops and conditional blocks, contributing to a well-structured and clear organization of code. This approach guarantees that the behavior and visibility of local variables are consistent and predictable, enhancing the readability and maintainability of the code.

4 Error handling

Warning (Yellow card) when undefined result from operation e.g. Modulo with a string or float
Error (Red card) when dividing by zero

5 Error Handling

5.1 Warnings: Yellow Card

In SportS, operations that result in undefined or unexpected outcomes issue a ”Yellow Card” warning. This mechanism is akin to cautioning in sports, indicating that while the operation could be completed, it might not have produced the intended result. An example of such a warning would be attempting a modulo (REAMAINDER) operation with a non-integer operand, like a string or float. The language alerts the programmer to potential misuse or misinterpretation of operations without halting execution, encouraging a review and correction of the code.

5.2 Errors: Red Card

Errors, on the other hand, result in a ”Red Card” and are issued for operations that cannot be executed due to critical issues, such as dividing by zero. In these cases, SportS will halt execution, signaling that immediate correction is necessary to proceed. This mirrors the issuing of a red card in sports, which signifies a serious infraction that removes the player from the game.