Tau Language Reference

[Initial draft: August 17, 2024]

Program

Tau programs consist of a sequence of one or more function declarations.

The function named, "main", will be the function automatically invoked to initiate the program.

Functions

Functions declarations include a name, zero or more typed parameters, a return type, and a function body. A function body is a compound statement.

Function names and parameter names are simple identifiers.

Example:

```
func fib(N : int) : int {
    if N < 2 {
        return 1
    } else {
        return fib(N-1) + fib(N-2)
    }
}
```

Types

Tau programs manipulate data of three given types: integer, Boolean, and single-dimensional arrays of either integer or Boolean values.

Integer and Boolean types are denoted by the reserved words, `int` and `bool`.

Array types are expressed with square brackets before their type:

```
func dummy(a : []int) {
    var b : [5] int

    b[2] = a[3]
}
```

Boolean Type

Boolean typed values represent `true` or `false`. Tau programs refer to the Boolean type as `bool`.

`true`, `false`, and `bool` are all reserved words.

Integer Type

Integer typed values represent 64-bit 2's-complement integer values. Tau program refer to the integer type as (keyword) `int`.

Array Type

Array typed values represent a sequence of either integer or boolean values. Tau programs refer to array types `[` *INT* `]` prior to the base type. The INT denotes the original size to be allocated as an integer literal.

Void Type

Tau programs may declare functions that do not return a value. Such functions are declared with the type `void`.

Statements

If Statement

If statements are composed of a test expression that guards a compound statement, and an optional alternative (aka “else”) compound statement. Unlike some terrible programming languages, Tau does not require that the test expression be wrapped in parentheses.

The test expression must evaluate to a Boolean value.

See the example above for an `if` statement.

While Statement

While statements are composed of a test expression that guards a compound statement.

The test expression must evaluate to a Boolean value.

Example:

```
func sum(len : int, a:[]int) : int {
    var i:int
    var sum:int

    sum = 0
    i = 0
    while i < len {
        sum = sum + a[i]
        i = i + 1
    }
    return sum
}
```

Assignment Statement

An assignment statement overwrites the previous value held in storage with a computed value. The left-hand side of the assignment denotes the storage to be overwritten, and the right-hand side denotes the expression that computes the new value.

There are examples above.

Return Statement

Return statements terminate the execution of the currently executing function and returns control to the calling function. Return statements optionally include an expression that represents the return value of the function.

Syntactically, return statements may only appear as the last statement in a compound statement.

Compound Statement

Compound statements consist of an optional sequence of variable declarations followed by an optional sequence of statements. Compound statements are delimited by curly braces.

A variable declaration begins with a the `var` keyword followed by an identifer, a colon, and a type.

Compound statements represent new lexical scopes for additional local variables. Those variables have a lifetime of the compound statement.

Statements and declarations are not separated by semicolons (or any other punctuation).

Any arrays allocated by local variable declarations are automatically freed when the control exits the block.

Call Statement

Call statements invoke a function and discard that invocations return value, if any. The function invocation in a call statement follows the same syntax and semantics as a function call expression, which is described more fully below.

A *statement* that calls a function must begin with the keyword `call`:

```
func foo() {
    print 123
}

func main() {
    call foo()
}
```

Print Statement

Tau supports a `print` statement that prints a single value on a line. The value does not need to be enclosed in parentheses.

Expressions

Tau expressions represent run-time values. They use syntax that should be very reminiscent of most modern programming languages.

Integer Literal Expressions

Integer literals are textual representations of integer values. They are all base 10.

Boolean Literal Expressions

Boolean literals are `true` and `false`.

Arithmetic Expressions

Arithmetic expressions compute integer values from subexpression(s).

Tau supports binary expressions for addition, subtraction, multiplication, and division. Tau further supports unary negation.

All arithmetic expressions consume integer values and produce integer results.

Comparison Expressions

Tau supports binary expressions for comparing integer or Boolean values. It supports less-than, less-than-or-equal, greater-than, greater-than-or-equal, equals, and not-equals.

Boolean values may be compared to Boolean values, and integer values may be compared to integer values. `False` is less than `True`.

All comparison expressions produce Boolean results.

Logical Expressions

Tau supports short-circuit `and`, and short-circuit `or` as binary expressions. It also supports unary `not`.

All logical expressions consume Boolean values and produce Boolean results.

Array Index Expressions

Arrays are indexed using square brackets to enclose an index. Arrays are indexed from 0.

It is impossible to determine an array’s size at runtime.

Function Call Expressions

Functions are called by providing the function's name followed by its run-time arguments enclosed in parentheses. The parentheses are required.

Function arguments are separated by commas.

The function returns a value of the type given in the function's declaration.

Operator Precedence and Associativity in Expressions

Tau expressions have the typical precedence in most programming languages. From highest precedence to lowest:

- Function invocation and array subscripting
- Unary negation ("`-`") and `not`
- Multiplication and division ("`*`" and "`/`")
- Addition and subtraction ("`+`" and "`-`")
- Comparison operations ("`<`", "`<=`", "`==`", "`!=`", "`>`", and "`>=`")
- Logical `and`
- Logical `or`

(All binary operators are left associative.)