Abstract

TACK is a simple statically-typed programming language invented for a Compiler Construction course at NYU. This document specifies the language. The semester-long project is to implement a compiler for TACK, one milestone at a time. See the class webpage for more information:

http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/tack-spec/HelloWorld_tack.txt

1 Example: Hello, World!

The following example TACK program prints a greeting to standard output. (See also:

http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/tack-spec/HelloWorld_tack.txt)

```tack
# Hello-world
main = fun () -> int {
  print("Hello, World!"); #call print function
  -> 0; #return 0
}
#end of main function
```

2 Syntax

Figure 1 shows the grammar. The notation follows the conventions in the Dragon book: the arrow ‘→’ separates the head and body of a production; non-terminals are in italics; tokens are in bold; and the vertical bar ‘|’ separates choices. In addition, a variety of superscript notations indicate repetition of the preceding item: \(X^+\) repeats \(X\) one or more times, \(X^*\) repeats \(X\) zero or more times, and \(X^*\) repeats \(X\) zero or more times separated by commas. The start symbol of the TACK grammar is `program`. A TACK program consists of a single file.

At the lexical level, TACK has the following tokens:

1. A tack is a course of action, especially one differing from some preceding course.
• Punctuation and operators: ( ), [ ], { }, ; , ->, =, :=, , : , !, /, %, +, -, <=, <, >=, >, ==, !=, & &, ||.

• Keywords: bool, else, false, for, fun, if, in, int, null, string, true, type, void, while.

• Identifiers (id): An identifier starts with a letter or underscore, followed by zero or more letters, underscores, or digits. Keywords cannot be used as identifiers. Identifiers are case-sensitive.

• Boolean literals (boolLit): true or false.

• Integer literals (intLit): An integer literal is either 0, or a digit from 1-9, followed by zero or more digits from 0-9.

• String literals (stringLit): A string literal starts and ends with double-quotes " ". A string can contain non-newline characters or escapes, consisting of a backslash \ followed by a non-newline character.

• Whitespace: Any whitespace outside of strings is ignored. Whitespace consists of space, tab, newline, or comments. A comment starts with a hash # and ends at the next newline or the end of the file, whichever comes first.

Figure 2: Tack operator precedence.

Figure 2 shows the operator precedence, in order from highest at the top to lowest at the bottom. In other words, parentheses and literals bind strongest, whereas || binds weakest. All postfix and infix operators are left-associative.

Figures 3 and 4 show additional example programs.

3 Scope Rules

Tack has three kinds of identifiers: functions, fields, and variables. Functions are defined by funDef, which can only appear at the top-level of a file; fields can be defined by a fieldType or a fieldLit; and variables can be defined by a varDef or a forStmt. Formal parameters are defined in a funType (each fieldType in the recordType defines a parameter), but they are treated as variables, not fields.

Tack associates scopes with the program, funDef, recordType, recordLit, forStmt, and blockStmt. If a blockStmt is used as the body of a funDef or forStmt, then they share a single scope. In other words, the formal parameters of a function are in the same scope as the function body, and the iterator variable of a for-loop is in the same scope as the loop body.

Tack is a block-structured language with lexical scoping. In other words, scopes nest, with identifiers in inner scopes hiding identifiers in outer scopes. It is a compiler error to define the same identifier in the same scope more than once. Identifiers can be used before they are defined, which is important to support recursive functions.

4 Types and their Relations

Tack has the following types:

• bool is a truth value (true or false).

• int is a 64-bit signed two’s-complement integer.

• string is a mutable reference to an immutable character string. That is, a string variable can be modified to refer to a different value, but the string value it refers to cannot itself be modified.

• Arrays are references to mutable collections of elements. For example, [int] is an array of integers. The empty array [] has an unknown element type.

• Records are references to mutable structures with fields. Each field has a unique name and a type. For example, (x: int, y: string) is a record with two fields.

• Function types can only appear as part of a function definition, and indicate the formal parameter types and the return type.

• void can only be used as a function return type, and indicates that the function returns no value.

• The null value has a type by itself.

Tack defines three relations on types:
• $T_1 \equiv T_2$: Type $T_1$ is the same type as $T_2$ if they have the same structure. If they are records, they must have the same field names and field types in the same order.

• $T_1 \leq T_2$: Type $T_1$ is a subtype of $T_2$ if a value of $T_1$ can be used wherever a value of $T_2$ is expected. The type of null is a subtype of all record types. If $T_1$ and $T_2$ are record types, then $T_1 \leq T_2$ if the list of fields of $T_2$ is a prefix of the list of fields of $T_1$. Otherwise, $T_1 \leq T_2$ only if $T_1 \equiv T_2$. In particular, array types are only subtypes if they have the same element type.

• $T_1 \approx T_2$: Type $T_1$ is castable to $T_2$ if they are both primitive types (\texttt{bool}, \texttt{int}, or \texttt{string}), or if $T_1 \leq T_2$ or $T_2 \leq T_1$.

5 Type Rules

The type rules for statements are:

• \textit{varDef}: The variable gets its type from the expression. The expression must have a known type: if part of its type is unknown due to an empty array, it must be made known with a cast.

• \textit{assignStmt}: The left-hand-side must be an l-value, i.e., a variable identifier, subscript expression, or field expression. The type of the right-hand-side must be a subtype of the type of the left-hand-side.

• \textit{blockStmt}: No rules.

• \textit{callStmt}: No rules.

• \textit{forStmt}: The expression after the \texttt{in} must refer to an array. The element type must be known. The type of the variable is the element type of the array.

• \textit{ifStmt}: The expression must be of type \texttt{bool}.

• \textit{returnStmt}: If the enclosing function returns \texttt{void}, the return statement must omit the expression. If the enclosing function is non-\texttt{void}, the return statement must specify an expression belonging to a subtype of the return type.

• \textit{whileStmt}: The expression must be of type \texttt{bool}.

The type rules for expressions are:

• \textit{id}: The identifier must refer to a variable. The result type is the type of the variable.

• \textit{callExpr}: The base expression must be an identifier that refers to a function. The number of actual parameters in the \textit{callExpr} must equal the number of formal parameters in the function definition. Each actual must be a subtype of the corresponding formal. The result type is the return type of the function.

• \textit{castExpr}: The type of the expression must be castable to the target type, which becomes the result type.

• \textit{fieldExpr}: The base expression must have a record type. The identifier must be the name of a field, whose type becomes the result type.

• \textit{infixExpr}: The rules depend on the operator:

  • $\&\&$, $\&$: Both operands must be \texttt{bool}, and the result type is \texttt{bool}.

  • $==$, $!=$: If the operands are null or have record type, then the operand types must be castable to each other. Otherwise, the operand types must be the same. The result type is \texttt{bool}.

  • $\leq$, $<$, $\geq$, $>$: Both operand types must be \texttt{int}, and the result type is \texttt{bool}.

  • $+$: If at least one of the operand types is \texttt{string}, then the other operand must be castable to \texttt{string}, and the result type is \texttt{string}. Otherwise, both operands must be \texttt{int}, and the result type is \texttt{int}.

  • $-\cdot$, $\div$, $\%$: Both operand types must be \texttt{int}, and the result type is \texttt{int}.

• \textit{parenExpr}: The result type is the type of the base expression.

• \textit{prefixExpr}: If the operator is $\sim$, the base type must be \texttt{int}, and the result type is \texttt{int}. If the operator is $!$, the base type must be \texttt{bool}, and the result type is \texttt{bool}.

• \textit{subscriptExpr}: The base expression must have an array type, and the subscript expression must have type \texttt{int}. The result type is the element type of the array type.
• boolLit, intLit, stringLit: The type of primitive literals is bool, int, and string, respectively.
• arrayLit: All elements must have the same type, and the result type is the array type for that element type.
• recordLit: Each field gets its type from its initializing expression. The result type is the record type for the given field names and types.
• The type of the null literal is the null type.

6 Dynamic Semantics

The runtime behaviors of statements are:
• varDef: Evaluate the expression, and copy its value into the variable.
• assignStmt: Evaluate the right-hand side expression to an r-value, evaluate the left-hand side expression to an l-value, and copy the r-value to the l-value.
• blockStmt: Execute the statements in the order they appear.
• callStmt: Evaluate the call expression, and discard the result, if any.
• forStmt: At the start of the loop, evaluate the expression to obtain a reference to an array a, and determine the array size n. Iterate over the indices i from 0 to n − 1. For each index, assign array element a[i] to the variable, then execute the loop body.
• ifStmt: Evaluate the expression. If the result is true, execute the first statement, otherwise, execute the second statement, if any.
• returnStmt: Evaluate the expression, if any, to obtain the return value. Pop the activation record from the stack, and transfer control back to the previous activation, or to the operating system if the current activation is at the top of the stack.
• whileStmt: Evaluate the expression. If it is true, execute the loop body and start over.

The runtime behaviors of expressions are:
• id: The result is the current value of the variable.
• callExpr: Evaluate each of the parameter expressions to obtain the actual parameters. Push an activation record for the callee, and initialize its formal parameters to the actuals. Transfer control to the callee. After the callee returns, its return value becomes the result of the callExpr.
• castExpr: Evaluate the expression to obtain its value. If the cast is between two primitive types (bool, int, string), pass the value to the corresponding conversion function (see Section 7). Otherwise, the original value is a reference, and the result is that same reference.
• fieldExpr: Evaluate the base expression to obtain a reference to a record. The result is the at the base reference plus the offset corresponding to the field name.
• infixExpr: The behaviors depend on the operator:
  • ||, &&: Logical or (||) and and (&&) have short-circuit semantics. In other words, if the value is already known after evaluating the left operand, they do not evaluate the right operand.
  • ==, !=: Equality and inequality are by-value for bool and int, and by-reference for all other types.
  • <=, <, >=, >: Compare two signed integers.
  • +: If one of the operands is a string, convert the other operand to string using the corresponding function (see Section 7), and the result is the concatenated string. Otherwise, both operands are int, and the result is their sum.
  • -, *, /, %: Subtract, multiply, divide, or compute the modulo (division remainder) of the two int operands.
• parenExpr: The result is the value of the base expression.
• prefixExpr: Evaluate the base expression. If the operator is -, the result is signed int negation. If the operator is !, the result is logical negation.
• subscriptExpr: Evaluate the base expression to obtain a reference to an array. Evaluate the subscript expression to obtain an index. The result is at the base reference plus the offset corresponding to the index. Indexing is zero-based.
The runtime behaviors of literals are:

- **boolLit, intLit, stringLit**: The result of a primitive literal is its value.
- **arrayLit**: Allocate memory for the element size times the number of elements. Evaluate each element expression, and assign its value to the offset corresponding to its index. The result is the reference to the new array.
- **recordLit**: Allocate memory for the record size. Evaluate each field expression, and assign its value to the offset corresponding to its field name. The result is the reference to the new record.
- **null**: The result is a null-pointer.

The runtime behavior in the following situations is unspecified:

- Using a variable before initializing it.
- Reaching the end of a non-void function without returning a value.
- Using an out-of-bounds index for an array.
- Casting a record reference to a type that is not a supertype.
- Dereferencing null.
- Overflowing the stack.
- Running out of heap memory.
- Using a null-character in a string.

### 7 Intrinsic Functions

The following functions are in scope for every program. It is a compiler error to try to define a function of the same name in the program. These functions are part of the runtime system.

```plaintext
append: (lhs: string, rhs: string)-> string
Return the concatenation of lhs and rhs.

bool2int: (b: bool)-> int
If b is true return 1 else 0.

bool2string: (b: bool)-> string
If b is true return "true" else "false".

int2bool: (i: int)-> bool
If i is 0 return false else true.

int2string: (i: int)-> string
Return the human-readable base-10 string representation of the signed integer. For example, if i is -12, return "-12".

length: (s: string)-> int
Return the number of characters in the string.

newArray: (eSize: int, aSize: int)-> [()]
This function is intended for internal compiler use only, and should not be called directly from TACK code. It allocates an array for the specified element size and array size. The return type (array of empty records) is a place-holder for the actual array type.

newRecord: (rSize: int)-> ()
This function is intended for internal compiler use only, and should not be called directly from TACK code. It allocates a record for the specified record size. The return type (empty record) is a place-holder for the actual record type.

print: (s: string)-> void
Print the string to standard output.

range: (start: int, end: int)-> [int]
If start ≥ end, return an empty array. Otherwise, return a new int array with end − start elements, initialized with the consecutive integers from start to end − 1.

size: (a: [()])-> int
The parameter type (array of empty records) is a place-holder. This function can be called on arrays of any type, and returns the number of elements.

string2bool: (s: string)-> bool
If the string is "" or "0" or "false" return false else true.

string2int: (s: string)-> int
Skip any leading whitespace. Parse leading characters to an integer, and ignore the remainder, if any. For example, if s is " -12 C", return -12.

stringEqual: (lhs:string, rhs:string)-> bool
If lhs and rhs contain the same characters return true else false.
```
# Insertion sort

```haskell
sort = fun (a : [int]) -> void {
    n = size(a);
    for iOuter in range(0, n-1) {
        iSmallest = iOuter;
        for iInner in range(iOuter, n) {
            if a[iInner] < a[iSmallest] {
                iSmallest := iInner;
            }
        }
        eSmallest = a[iSmallest];
        a[iSmallest] := a[iOuter];
        a[iOuter] := eSmallest;
    }
}
main = fun () -> int {
    a = [2, 0, 1, 2];
    sort(a);
    for e in a {
        print(" "+ e);
    }
    print("\n");
    -> 0;
}
```

Figure 3: TACK program to sort an array of integers; for the input array [2, 0, 1, 2], this prints 0 1 2 2. See also: [http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/tack-spec/InsertionSort_tack.txt](http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/tack-spec/InsertionSort_tack.txt)

# Tree interpreter

```haskell
eval = fun (expr: (o: string)) -> int {
    if stringEqual("+", expr.o) {
        e = expr : (o: string, l: (o: string), r: (o: string));
        -> eval(e.l) + eval(e.r);
    }
    if stringEqual("-", expr.o) {
        e = expr : (o: string, l: (o: string), r: (o: string));
        -> eval(e.l) - eval(e.r);
    }
    -> expr.o: int;
}
main = fun () -> int {
    tree = (o="+", l=(o="-", l=(o="5"), r=(o="3")), r=(o="2"));
    result = eval(tree);
    print(result + "\n");
    -> 0;
}
```

Figure 4: TACK program to evaluate a tree representation of the expression (5 - 3) + 2; prints 4. See also: [http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/tack-spec/TreeInterpreter_tack.txt](http://cs.nyu.edu/courses/fall11/CSCI-GA.2130-001/tack-spec/TreeInterpreter_tack.txt)