Tiger is simple, statically-typed programming language first introduced in Andrew Appel's book *Modern Compiler Design in Java* (Cambridge University Press, 1998). It is a hybrid of a simple imperative language and a simple functional language, so it includes both mutable state, and also things like "let" bindings. It does not have advanced features, such as an object oriented classes and first-class functions, but these could easily be added.

The following specification starts with the grammar of the language, then explains the semantic rules (including the type system), followed by the dynamic semantics (operational semantics). At the end of this document you will find a few simple examples.

```
program          →  exp
exp              →  letExp | seqExp | ifThenElse | whileExp | forExp
                   | assign | lValue | callExp | infixExp | neg | arrCreate | recCreate
                   | nil | INT | STRING | break
letExp           →  let dec+ in exp∗i end
seqExp           →  ( exp∗i )
ifThenElse       →  if exp then exp [ else exp ]
whileExp         →  while exp do exp
forExp           →  for ID := exp to exp do exp

dec              →  funDec | varDec | tyDec
funDec           →  function ID ( fieldDec∗ ) [ : TYID ] = exp
varDec           →  var ID [ : TYID ] := exp
tyDec            →  type TYID = ty

ty              →  TYID | arrayTy | recTy
arrayTy          →  array of ID
recTy            →  { fieldDec∗ }
fieldDec         →  ID : TYID

assign           →  lValue := exp
lValue           →  ID | subscript | fieldExp
subscript        →  lValue [ exp ]
fieldExp         →  lValue . ID
callExp          →  ID ( exp∗i )
infixExp         →  exp OPER exp
neg              →  - exp
arrCreate        →  TYID [ exp ] of exp
recCreate        →  TYID { fieldInit∗ }
2 Syntax

Tiger syntax is shown in Figure 1. The syntax is specified as an extended BNF – a way of writing down a context-free grammar, augmented with the following convenient short-hands:

- \( X^+ \) denotes one or more repetitions of \( X \)
- \( X^* \) denotes zero or more repetitions of \( X \) separated by commas
- \( X^*; \) denotes zero or more repetitions of \( X \) separated by semicolons
- \([X]\) denotes \( X \) is optional (note the difference with literal \([\) and \)] (e.g., for array sub-scripting).

At the lexical level, Tiger has the following kinds of tokens:

- Operators: ( ) [ ] { } : := . , / + - = <> < >= <= & |
- Keywords: array break do else end for function if in let nil of then to type var while
- Identifiers ID and TYID, which have the same lexical structure: starts with a letter, followed by zero of more letters, digits, or underscores. Identifiers are case sensitive. Keywords cannot be used as identifiers.
- Integer literals INT consist of a sequence of one or more decimal digits
- String literals STRING start and end with double quotes. A string can contain printable characters and/or special escape characters. Escape characters start with a backslash:
  \n newline
  \t tab
  \^c control character c
  \ddd ASCII character ddd (decimal)
  \ double quote
  \\ backslash
  \space*\ ignored region

  The last form makes it possible to have multi-line string literals by ending a line with a backslash, then restarting the literal on the next line starting with a backslash.

- Whitespace consists of spaces, tabs, and newlines, and it is only important insomuch as it separates identifiers and keywords. It is otherwise ignored. Whitespace inside quotes is preserved.

- Comments start with /* and end with */ (like C comments). Unlike C, however, comments in Tiger can be nested, and the comment delimiters must match up (the way matching parentheses match). From a lexical standpoint, comments are treated like whitespace.

All operators in Tiger are left associative, except for the comparison operators, which do no associate. The precedence of the operators is shown below, from highest precedence to lowest precedence:
3 Scope rules

Tiger uses lexical scoping and has four kinds of identifiers: type names, function names, variable names, and field names. Every use of a name must match an appropriate declaration in its scope, according to the following rules:

- **Type identifiers** are introduced by a `tyDec`. The scope of a type declaration starts at the beginning of a consecutive sequence of type declarations that include it, and ends at the last enclosing let expression. This rule makes it possible to have recursive types or even mutually recursive types.

- **Function identifiers** are introduced by a `funDec`. The scope of a function declaration starts at the beginning of a consecutive sequence of function declarations that include it, and ends at the last enclosing let expression. As with type declarations, this rule allows functions to be recursive without introducing special syntax (e.g., `letrec`).

- **Variable identifiers** are introduced by either a `varDec`, a `fieldDec` used as a formal parameter to a function, and the index variable in a `forExp`. The scope of a `varDec` starts at the declaration and extends to the end of the enclosing let expression. The scope of a formal parameter is the body of the function. The scope of the for loop index variable is the loop body.

- **Field identifiers** are introduced by a `fieldDec` inside a record type definition (`tyRec`). The scope is within the record only, or whenever a record of this type is accessed by a `fieldExp` or `fieldInit`.

There are two namespaces for identifiers: one for types and one for variables and functions. An identifier can be used simultaneously in both namespaces. Scopes can be nested, and identifiers in inner scopes hide identifiers in outer scopes that have the same name. It is an error to have two identifiers with the same name in the same scope.

4 Type system

4.1 Type structure

Tiger has the following types:

- **int** is a signed integer.

- **string** is an immutable character string.
• Arrays are references to mutable collections of elements.

• Records are references to mutable structures with fields. Each field has a type and unique name within the record.

• `void` is the type given to expressions that produce no value. Expressions of type `void` must not appear where a value is expected.

The type identifiers `int` and `string` are pre-defined at the top-level scope of the program.

Each recursive cycle of types must pass through at least one array or record. For example, the sequence of type declarations `type a=b` and `type b=a` is illegal.

Each declaration of an array or record type introduces a new type. For example, types `a` and `b` declared by `type a={f:int}` and `type b={f:int}` are incompatible, even though they have the same structure. On the other hand, after the declaration `type c=d`, types `c` and `d` are aliases referring to the same type.

The `nil` value does not have a type by itself; instead, `nil` belongs to all record types.

Assignment, parameter passing, and comparison operates on the value for `string` and `int`, but operates on the reference for arrays and records.

### 4.2 Type rules

The type rules for declarations are:

- **funDec**: If the declaration does not specify a return type, the return type is `void`. Either way, the return type must match the type of the body.

- **varDec**: If the declaration explicitly specifies a type, it must match the type of the initializer. The type of the variable is the explicitly specified type, or, if missing, the initializer type.

The type rules for l-values are:

- **ID**: The identifier must refer to a variable. The result type is the type of the variable.

- **subscript**: The base expression must have an array type, and the index must be of type `int`. The result type is the element type of the array.

- **fieldExp**: The base expression must have a record type, and the identifier must name a field of the record. The result type is the type of the field.

The type rules for expressions are:

- **nil**: Can only be used in a context where the specific record type can be determined (initializer of typed varDec, assignment, comparison using `<>` or `=` where the other operand has a known type, or actual parameter to a function call).

- **INT**: Has type `int`.

- **STRING**: Has type `string`.


• \textit{seqExp}: If the sequence is empty, the type is \texttt{void}, otherwise, the type is that of the last expression.

• \textit{neg}: Both the operand and the result are \texttt{int}.

• \textit{callExp}: The identifier must refer to a function. The number and types of actual and formal parameters must be the same. The type of the call is the return type of the function.

• \textit{infixExp}: The rules depend on the operator:
  - $+$, $-$, $*$, $/$: The operands must be of type \texttt{int} and the result type is \texttt{int}.
  - $=$, $<>$: The operand types must match and the result type is \texttt{int}.
  - $>$, $<$, $\geq$, $\leq$: The operand types must match and must be \texttt{int} or \texttt{string}. The result type is \texttt{int}.
  - $\&$, $|$: The operands must be \texttt{int} and the result type is \texttt{int}.

• \textit{arrCreate}: The \texttt{TYID} must refer to an array type. The expression in square brackets must be \texttt{int}, and the expression after of must match the element type of the array. The result type is the array type.

• \textit{recCreate}: The \texttt{TYID} must refer to a record type, and the order, names, and types of fields must match. The result type is the record type.

• \textit{assign}: The type of the \textit{lValue} and the \textit{exp} must match. The result type is \texttt{void}.

• \textit{ifThenElse}: The condition type must be \texttt{int}, and the then-clause and else-clause (if present) must have the same type, which becomes the result type.

• \textit{whileExp}: The condition type must be \texttt{int}, and the body type must be \texttt{void}. The result type is \texttt{void}.

• \textit{forExp}: The start and end index must be of type \texttt{int}. The variable is of type \texttt{int} and must not be assigned to in the body. The body must be of type \texttt{void}. The result type is \texttt{void}.

• \textit{break}: Can only be used in a \textit{whileExp} or \textit{forExp}. The result type is \texttt{void}.

• \textit{letExp}: If the body is empty, the type is \texttt{void}, otherwise, the type is that of the last body expression.

5 Dynamic semantics

The runtime behaviors of variable declarations are:

• \textit{varDec}: Evaluate the expression, and initialize the variable to that value.

The runtime behaviors of l-values are:

• \textit{ID}: The result is the current value of the variable.
• subscript: Evaluate the base expression to obtain a reference to an array. Evaluate the index expression to obtain an index. Indexing is zero-based. The result is the element at that index.

• fieldExp: Evaluate the base expression to obtain a reference to a record. The result is the value of the field in the record.

The runtime behaviors of expressions are:

• nil: The result is a null-reference to a record.

• INT: The result is the integer value.

• STRING: The result is the string value.

• seqExp: Evaluate each exp in order. If the sequence is empty, there is no result, otherwise, the result of the last exp is the result of the seqExp.

• neg: Signed integer negation.

• callExp: Evaluate each parameter exp in order. Copy the actual parameters to the formals. Run the body of the callee. The result is the return value from the callee.

• infixExp: The behaviors depend on the operator:
  – + - * /: Add, subtract, multiply, or divide the two integer operands.
  – = <>: Equality and inequality are by-value for int and string, and by-reference for records and arrays. The result is 1 for true or 0 for false.
  – > < >= <=: Magnitude comparison of int values, or lexicographic comparison of string values. The result is 1 for true or 0 for false.
  – & |: Logical boolean conjunction and disjunction using short-circuit semantics. In other words, if the value is already known after evaluating the left operand, do not evaluate the right operand. Any non-zero integer is considered true, and 0 is false.

• arrCreate: Evaluate the size expression. Allocate a new array of the appropriate size. Evaluate the initializer expression. Copy its value into all elements. The result is the reference to the new array.

• recCreate: Allocate a new record, and initialize its fields using the field expressions. The result is the reference to the new record.

• assign: Evaluate the lValue to a location and the exp to a value. Copy the value to the location.

• ifThenElse: Evaluate the condition. If it is nonzero, evaluate the then-clause and use its result, else evaluate the else-clause and use its result.

• whileExp: Evaluate the condition. If it is non-zero, evaluate the loop body and start over.

• forExp: Evaluate the lower and upper bound (only once before entering the loop). If the upper bound is less than the lower bound, the body is not executed. Otherwise, the body is executed once for every value between the lower and upper bound inclusive, with the iteration variable set accordingly.
- **break**: Terminate evaluation of the immediately enclosing `whileExp` or `forExp`.
- **letExp**: Evaluate each `dec` that is a `varDec` in order, then evaluate each `exp` in order. The result of the last `exp` is the result of the `letExp`. The runtime behavior in the following situations is unspecified:
  - Using a subscript with an out-of-bounds index.
  - Using a `fieldExp` on `nil`.
  - Overflowing the stack or running out of heap space.

6 Intrinsic functions

The following functions are pre-defined at the top-level scope of the program. They are part of the runtime system and form the standard library for Tiger.

- **print(s : string)** – Prints `s` to standard output.
- **flush()** – Flushes standard output.
- **getchar() : string** – One character from standard input, or empty string for end-of-file.
- **ord(s : string) : int** – ASCII value of first character of `s`, or -1 for empty string.
- **chr(i : int) : string** – Single-character string for ASCII value `i`, or halt program if out-of-range.
- **size(s : string) : int** – Length of `s`.
- **substring(s : string, first : int, n : int) : string** – Substring from `s[first]` to `s[first+n-1]` inclusive (zero-based indexing), or empty string if `n<0` or `first` or `first+n` are out of range.
- **concat(s1 : string, s2 : string) : string** – Concatenation of `s1` and `s2`.
- **not(i : int) : int** – if `i=0` then 1 else 0.
- **exit(i : int)** – Halt the program with code `i`. 