Tiger Language Specification

The following is an informal specification of a dialect of Tiger that will act as the target language for the semester project.

1 Lexicon

The lexemes of Tiger consist of keywords and classes. The keywords consist of the following: `array`, `begin`, `break`, `do`, `else`, `end`, `enddo`, `endif`, `float`, `for`, `func`, `if`, `in`, `int`, `let`, `of`, `return`, `then`, `to`, `type`, `var`, `while`, `,`, `:`, `;`, `(`, `)`, `[`, `]`, `{`, `}`, `.`, `+`, `-`, `*`, `/`, `=`, `<>`, `<`, `>`, `<=`, `>=`, `&`, `|`, and `:=`. Each keyword token contains as its language exactly its literal string. I.e., the language of `func` is the character string `func`.

The classes consist of `id`, `intlit`, and `floatlit`. The language of each class is as follows:

- `id` is the language of program identifiers. A program identifier is a sequence of letters, numbers, and the underscore character. An identifier must begin with either a letter or underscore, and must contain at least one letter or number.
- `intlit` is the language of integer literals. An integer literal is a non-empty sequence of digits.
- `floatlit` is the language of floating-point literals. A floating-point literal must consist of a non-empty sequence of digits, a radix (i.e., a decimal point), and a (possibly empty) sequence of digits. The literal cannot contain any leading zeroes not required to ensure that the sequence of digits before the radix is non-empty. E.g., `0.123` is a float literal, but `00.123` is not.

2 Syntax

The syntax of Tiger is given as a BNF in Figure 1; the syntax is defined over terminals consisting of the lexemes defined in §1 and languages of Boolean boolexpr numeric numexpr, and constant expressions const, which are defined in Figure 2. A Tiger program (Equation 1) is a *declaration segment* followed by a sequence of statements each ending with a semicolon (Equation 27–Equation 28), within a let binding. A declaration segment (Equation 2) is a sequence of type declarations (Equation 3–Equation 4), followed by a sequence of variable declarations (Equation 10–Equation 11), followed by a sequence of function declarations (Equation 17–Equation 18).

A type declaration is a type assigned to an identifier (Equation 5). A type is either the fixed types for integers (Equation 6) or floats (Equation 7), a type identifier (Equation 8), or an array type declaration defined using a base type (Equation 9).

A variable declaration (Equation 12) is a sequence of identifiers (Equation 13–Equation 14), a type, and an optional initialization. An optional initialization is either the empty string (Equation 15) or an assignment from a constant (Equation 16).

A function declaration (Equation 19) is an identifier, a sequence of parameters separated by commas (Equation 20–Equation 23), an optional return type, and a sequence of statements. A parameter is an identifier annotated with a type (Equation 24). A statement may be an assignment from a data expression to a variable (Equation 30); an `if` statement without an `else` branch (Equation 31); an `if` statement with an `else` branch (Equation 32); a `while` loop with a Boolean expression as loop guard and sequence of statements of as body (Equation 33); a `for` loop that consists of a data expression for initializing an identifier, a data expression that the value stored in the identifier is checked against

Argument 0	Argument 1	Result
$\mathbb{Z}$	$\mathbb{Z}$	$\mathbb{Z}$
$\mathbb{Z}$	F	F
F	$\mathbb{Z}$	F
F	F	F

Table 1: Type map for all numeric binary operators, $\mathrm{T_N}$.

in each iteration, and a sequence of statements executed in each iteration of the loop (Equation 34); a call statement consisting of an optional store to a variable, the identifier of a callee, and arguments represented as a comma-separated sequence of expressions (Equation 43–Equation 46); a `break` keyword (Equation 36); or a return statement constructed from a numeric expression (Equation 37). An optional store is either an empty string (Equation 41) or an assignment to an lvalue (Equation 42). An lvalue is an identifier followed by an optional array offset constructed from a numeric expression (Equation 38–Equation 40).

The languages of Boolean, numeric, and constant expressions are defined in Figure 2. A Boolean expression is a sequence of *clauses* separated by disjunction symbols (Equation 47–Equation 48). A clause is a sequence of *predicates* separated by conjunction symbols (Equation 49–Equation 50). A predicate may be either two numeric expressions separated by a binary predicate (Equation 51, Equation 53–Equation 58) or a parenthesized Boolean expression (Equation 52).

A numeric expression is a sequence of terms separated by sum and subtraction operators (Equation 59–Equation 62). A term is a sequence of factors separated by multiplication and division operators (Equation 63–Equation 66). A factor is an integer literal or float literal (Equation 67, Equation 71–Equation 72), an identifier (Equation 68), an offset into an identifier (Equation 69), or a parenthesized numeric expression (Equation 70).

3 Semantics

In this section, we give a semantics for Tiger by defining conditions under which a syntactically valid Tiger program is well-typed. Let the type of Booleans be denoted $\mathbb{B}$. Let the space of *numeric* types include the type *integer* (denoted $\mathbb{Z}$) and the type *float* (denoted F). For each numeric type T, let there be a type of *arrays* of T (denoted T array). Let the numeric types and array types be the class of *first-order* types (denoted $\mathcal{T}_1$). For each positive natural number $i \in \mathbb{N}$ and first-order types $T_0 \times \ldots \times T_n, T'$, let $T_0 \times \ldots \times T_n \to T'$ be the *function type* from $T_0 \times \ldots \times T_n$ to T'. Let the space of all types (denoted types) be $\mathbb{B}$, the first-order types, and the function types.

Let a *type context* be a partial function from program identifiers to types. I.e., the space of typing contexts is denoted $\mathsf{contexts} = \mathsf{ids}_P \to \mathsf{Types}$. For each type context $\Gamma \in \mathsf{contexts}$, program identifier $\mathtt{x} \in \mathsf{ids}_P$, and type $T \in \mathsf{Types}$, let Γ updated to bind $\mathtt{x}$ to T be denoted $\Gamma[\mathtt{x} \mapsto T]$. For all type contexts Γ_0 and Γ_1 over distinct sets of identifiers, let $\Gamma_0 \cup \Gamma_1$ denote the type context with all bindings in Γ_0 and Γ_1.

3.1 Types of expressions

The type of each expression is determined by the subexpressions from which it is constructed. For each context $\Gamma \in \mathsf{contexts}$, factor, term, or numeric expression e, and type T, we denote that under Γ, e has type T as $e : T$.

In particular:

- If $e \equiv \mathtt{c}$ is an Integer literal $\mathtt{c}$, then under Γ, e has type $\mathbb{Z}$.
- If $e \equiv \mathtt{c}$ is a Float literal $\mathtt{c}$, then under Γ, e has type F.
- If $e \equiv \mathtt{x}$ is an identifier $\mathtt{x}$, then under Γ, e has type $\Gamma(\mathtt{x})$.
- If $e \equiv \mathtt{x}[e_0]$ is an offset expression e_0 into identifier $\mathtt{x}$, under Γ, e_0 has type $\mathbb{Z}$ and $\mathtt{x}$ has type type T array, then under Γ, e has type T.

- If $e \equiv$ `(`e_0`)` is a parenthesized expression e_0 which under Γ has type T, then under Γ, e has type T.
- If $e \equiv t \otimes f$ is a non-linear operator $\otimes$ applied to term t and factor f, under Γ, t has numeric type T_0 and f has numeric type T_1, then under Γ, e has type $\mathrm{T_N}(T_0, T_1)$.
- If $e \equiv e_0 \oplus t$ is a linear operator $\oplus$ applied to numeric expression e_0 and term t, under Γ, e_0 has numeric type T_0 and t has numeric type T_1, then under Γ, e has type $\mathrm{T_N}(T_0, T_1)$.

3.2 Well-typedness of Boolean expressions

Under a given typing-context Γ, a Boolean expression is well-typed if it is constructed from predicates over expressions with numeric types. In particular, for each Boolean expression e,

- If $e \equiv e_0 \oplus e_1$ is a Boolean operation applied to numeric expression e_0 and numeric expression e_1, under Γ, e_0 and e_1 has numeric types, then under Γ, e is well-typed.
- If $e \equiv c$ `&` p is a conjunction of a clause c and predicate p, under Γ, c and p are well-typed, then under Γ, e is a well-typed Boolean expression.
- If $e \equiv e_0$ `|` c is a disjunction of a Boolean expression e_0 and a clause c, under Γ, e_0 and c are well-typed, then under Γ, e is well-typed.

3.3 Well-typedness of statements

A statement s is well-typed if it defines a feasible sequence of instructions that can be executed. For each typing context Γ and lvalue l, the type of l under Γ is defined casewise on l as follows:

- If $l \equiv$ `x` is an identifier `x`, then the type of l under Γ is $\Gamma($`x`$)$.
- If $l \equiv$ `x[`e`]` is an offset expression e into identifier `x`, $\Gamma($`x`$) = T$ `array`, and under Γ, e has type $\mathbb{Z}$, then under Γ has type T.

For each typing context Γ and return type ρ,

- If $s \equiv l := e$ assigns expression e to lvalue l, under Γ, l and e have type T, then under Γ and ρ, s is well-typed.
- If $s \equiv$ `if` e `then` s_0 `endif` has guard Boolean expression e and then-branch s_0, under Γ, e is well-typed, and under Γ and ρ, s_0 is well-typed, then under Γ and ρ, s is well-typed.
- If $s \equiv$ `if` e `then` s_0 `else` s_1 `endif` has guard Boolean expression e, then-branch s_0, and else branch s_1, under Γ, e is well-typed, and under Γ and ρ, s_0, and s_1 are well-typed, then under Γ and ρ, s is well-typed.
- If $s \equiv$ `while` e `do` s_0 `enddo` has guard Boolean expression e and body s_0, and under Γ, e and s_0 are well-typed, then under Γ, s is well-typed.
- If $s \equiv$ `for x :=` e_0 `to` e_1 `do` s_0 `enddo` has initializer expression e_0, final expression e_1, and body s_0, under Γ, e_0 and e_1 have type $\mathbb{Z}$, and under Γ and ρ, s_0 is well-typed, then under Γ and ρ, s is well-typed.
- if $s \equiv$ `l := f(`e_0`,` $\ldots$ `,`e_n`)` is a call that runs procedure `f` on argument expressions $e_0, \ldots, e_n$, if $\Gamma($`f`$) = T_0 \times \ldots \times T_n \to T'$, and **(1)** for each $0 \leq i \leq n$, under Γ, e_i has type T_i; **(2)** under Γ, l has type T'; then under Γ and ρ, s is well-typed.
- If $s \equiv$ `break`, then under Γ and ρ, s is well-typed.
- If $s \equiv$ `return` e returns expression e and under Γ, e has type ρ, then under Γ and ρ, s is well-typed.

A sequence of statements is well-typed under Γ and ρ if each statement in the sequence is well-typed under Γ and ρ.

3.4 Well-typedness of programs

A program is well-typed if the statements in each declared function and within the overall let-binding of the program are well-typed.

Type alias maps In particular, let a *type alias map* be a map from each type identifier to a type; i.e., the space of type alias maps is denoted $\mathsf{alias} = \texttt{ids} \to \mathsf{Types}$. The type-alias map of a sequence of type declarations D under a type-alias map A is defined as follows:

- If $D \equiv \epsilon$, then the alias map of D under A is A.
- If $D \equiv \texttt{t := } E; D'$ is a sequence of type declarations and type expression E has actual type T under A, then the alias map of D under A is the alias map of D' under an alias map that extends A to bind $\texttt{x}$ to T (i.e., the alias map $A[\texttt{t} \mapsto T]$).

The type alias map of a type-declaration segment D is the alias map of D under an empty alias map.

For each type expression E and type-alias map A, E has actual type T under A under the following conditions:

- If $E \equiv \texttt{int}$, then the actual type of E under A is $\mathbb{Z}$.
- If $E \equiv \texttt{float}$, then the actual type of E under A is F.
- If $E \equiv \texttt{x}$, then the actual type of E under A is $A(\texttt{x})$.
- If $E \equiv \texttt{array[intlit] of } E_0$ and E_0 has an actual type T_0 under A that is a numeric type, then E has actual type T_0 array under A.

Type context of variable declarations The typing context of a sequence of variable declarations D under a type-alias map A is defined as follows:

- If $D \equiv \epsilon$ is the empty sequence of variable declarations, then the typing context of D under A is the empty context.
- If $D \equiv \texttt{var x : T ;} D'$ is a sequence of a variable declaration with a sequence of declarations, type expression $\texttt{T}$ has actual type T under A, and the typing context of D' under A is Γ', then the typing context of D under A is $\Gamma[\texttt{x} \mapsto T]$.

Type context of function declarations The typing context of a sequence of function declarations D under a type-alias map A is defined as follows:

- If $D \equiv \epsilon$ is the empty sequence of function declarations, then the typing context of D under A is the empty context.
- If $D \equiv \texttt{func f(x0 : } E_0\texttt{, ..., xn : } E_n\texttt{) :}E'; D'$ is a function declaration followed by a sequence of declarations, the type expressions $E_0, \ldots, E_n, E'$ have actual types $T_0, \ldots, T_n, T'$ under A, and D' has type context Γ' under A, then the type context of D under A is $\Gamma'[\texttt{f} \mapsto T_0 \times \ldots \times T_n \to T']$.

Well-typedness of a function declaration A function declaration $\texttt{func f(x0 : } E_0\texttt{, ..., xn : } E_n\texttt{) : } E's'$: is well-typed under typing context Γ and type-alias map A if **(1)** $E_0, \ldots, E_n, E'$ have actual types $T_0, \ldots, T_n, T'$ under A and **(2)** under type context $\Gamma[\texttt{x0} \mapsto T_0] \ldots [\texttt{xn} \mapsto T_n]$ and return type T', s' is well-typed.

Program well-typedness A program is well-typed if under the typing context defined by its type, variable, and function declarations, each declared function and its main statement are well-typed. In particular, for program $P \equiv \texttt{let } D_T\ D_V\ D_F \texttt{ in } s \texttt{ end}$, if **(1)** D_T has type-alias map A, **(2)** D_V has type context Γ_V under A, **(3)** D_F has type context Γ_F under A, **(4)** D_F is well-typed under typing context $\Gamma' = \Gamma_V \cup \Gamma_F$ and type-alias map A, and **(5)** s is well-typed under Γ', then P is well-typed.

program → `let` declseg `in` stmts `end` (1)
declseg → typedecls vardecls funcdecls (2)
typedecls → ϵ (3)
typedecls → typedecl typedecls (4)
typedecl → `type` `id` `:=` type `;` (5)
type → `int` (6)
type → `float` (7)
type → `id` (8)
type → `array` `[` `intlit` `]` `of` type (9)
vardecls → ϵ (10)
vardecls → vardecl vardecls (11)
vardecl → `var` ids `:` type optinit`;` (12)
ids → `id` (13)
ids → `id` `,` ids (14)
optinit → ϵ (15)
optinit → `:=` const (16)
funcdecls → ϵ (17)
funcdecls → funcdecl funcdecls (18)
funcdecl → `func` `id` `(` params `)` optrettype `begin` stmts `end` `;` (19)
params → ϵ (20)
params → neparams (21)
neparams → param (22)
neparams → param `,` neparams (23)
param → `id` `:` type (24)
optrettype → ϵ (25)
optrettype → `:` type (26)
stmts → fullstmt (27)
stmts → fullstmt stmts (28)
fullstmt → stmt `;` (29)
stmt → lvalue `:=` numexpr (30)
stmt → `if` boolexpr `then` stmts `endif` (31)
stmt → `if` boolexpr `then` stmts `else` stmts `endif` (32)
stmt → `while` boolexpr `do` stmts `enddo` (33)
stmt → `for` `id` `:=` numexpr `to` numexpr `do` stmts `enddo` (34)
stmt → optstore `id` `(`numexprs`)` (35)
stmt → `break` (36)
stmt → `return` numexpr (37)
lvalue → `id` optoffset (38)
optoffset → ϵ (39)
optoffset → `[` numexpr `]` (40)
optstore →ϵ (41)
optstore → lvalue `:=` (42)
numexprs → ϵ (43)
numexprs → neexprs (44)
neexprs → numexpr (45)
neexprs → numexpr `,` neexprs (46)

Figure 1: Grammar for Tiger top-level constructs, represented in BNF.

boolexpr → clause (47)
boolexpr → boolexpr | clause (48)
clause → pred (49)
clause → clause & pred (50)
pred → numexpr boolop numexpr (51)
pred → (boolexpr) (52)
boolop → = (53)
boolop → <> (54)
boolop → <= (55)
boolop → >= (56)
boolop → < (57)
boolop → > (58)
numexpr → term (59)
numexpr → numexpr linop term (60)
linop → + (61)
linop → - (62)
term → factor (63)
term → term nonlinop factor (64)
nonlinop → * (65)
nonlinop → / (66)
factor → `const` (67)
factor → `id` (68)
factor → `id[` numexpr`]` (69)
factor → (numexpr) (70)
const → `intlit` (71)
const → `floatlit` (72)

Figure 2: Grammar for Tiger expressions.