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`, `boolean`, `break`, `do`, `else`, `end`, `enddo`, `endif`, `false`, `float`, `for`, `func`, `if`, `in`, `int`, `let`, `of`, `return`, `then`, `to`, `true`, `type`, `unit`, `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`, `floatlit`, and `comment`. 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.
- `comment` is the language of comment. A comment is `/*` followed by a sequence of characters that do not contain `*/`, followed by `*/`.

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, a language of expressions expr 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 12–Equation 13), followed by a sequence of function declarations (Equation 19–Equation 20).

A type declaration is a type assigned to an identifier (Equation 5). A type is either the fixed types for Booleans (Equation 6), integers (Equation 7), floats (Equation 8), or `unit` (Equation 9), a type identifier (Equation 10), or an array type declaration defined using a base type (Equation 11).

A variable declaration (Equation 14) is a sequence of identifiers (Equation 15–Equation 16), a type, and an optional initialization. An optional initialization is either the empty string (Equation 17) or an assignment from a constant (Equation 18).

A function declaration (Equation 21) is an identifier, a sequence of parameters separated by commas (Equation 22–Equation 25), a return type, and a sequence of statements. A parameter is an identifier annotated with a type (Equation 26). 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

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 $\mathrm{T_N}$ for numeric binary operations.

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 in each iteration, and a sequence of statements executed in each iteration of the loop (Equation 34); a `break` keyword (Equation 35); or a return statement constructed from a numeric expression (Equation 36). An lvalue is an identifier followed by an optional array offset constructed from a numeric expression (Equation 37–Equation 39).

The language of expression is defined in Figure 2. A Boolean expression is a sequence of *clauses* separated by disjunction symbols (Equation 44–Equation 45). A clause is a sequence of *predicates* separated by conjunction symbols (Equation 46–Equation 47). A predicate may be either a numeric expression (Equation 48) or two numeric expressions separated by a comparator (Equation 49, Equation 50–Equation 55).

A numeric expression is a sequence of terms separated by sum and subtraction operators (Equation 56–Equation 59). A term is a sequence of factors separated by multiplication and division operators (Equation 60–Equation 63). A factor is an integer literal (Equation 64), a float literal (Equation 70), the unit value (Equation 71), an identifier (Equation 65), an offset into an identifier (Equation 66), a function call (Equation 67), or a parenthesized expression (Equation 68).

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 containing only the unit value be denoted Unit. 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).

Let the space of *first-order* types (denoted $\mathcal{T}_1$) satisfy the following inductive definition:

- Unit is a first-order type.
- $\mathbb{B}$ is a first-order type.
- $\mathbb{Z}$ is a first-order type.
- F is a first-order type.
- For each type T, if T is a first-order type, then T array is a first-order type.

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 Unit, $\mathbb{B}$, 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:

- Let the space of Boolean literals be $\{\texttt{true}, \texttt{false}\}$. If $e \equiv \texttt{c}$ where $\texttt{c}$ is a Boolean literal, then under γ, e has type $\mathbb{B}$.
- If $e \equiv \texttt{c}$ is an Integer literal $\texttt{c}$, then under Γ, e has type $\mathbb{Z}$.
- If $e \equiv \texttt{c}$ is a Float literal $\texttt{c}$, then under Γ, e has type F.
- If $e \equiv \_$ is the unit value, the under Γ, _ has type Unit.
- If $e \equiv \texttt{x}$ is an identifier $\texttt{x}$, then under Γ, e has type $\Gamma(\texttt{x})$.
- If $e \equiv \texttt{x[}e_0\texttt{]}$ is an offset expression e_0 into identifier $\texttt{x}$, under Γ, e_0 has type $\mathbb{Z}$ and $\texttt{x}$ has type type T $\textsf{array}$, then under Γ, e has type T.
- If $e \equiv \texttt{(}e_0\texttt{)}$ 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 $\mathrm{T_N}(T_0, T_1)$.
- If $e \equiv e_0 \oplus e_1$ is a comparitor applied to numeric expression e_0 and numeric expression e_1, under Γ, e_0 and e_1 have numeric types T_0 and T_1, then under Γ, e has type $\mathbb{B}$.
- If $e \equiv c \,\texttt{\&}\, p$ is a conjunction of a clause c and predicate p, under Γ, c and p have type $\mathbb{B}$, then under Γ, e has type $\mathbb{B}$.
- If $e \equiv e_0 \;\texttt{|}\; c$ is a disjunction of a Boolean expression e_0 and a clause c, under Γ, e_0 and c have type $\mathbb{B}$, then under Γ, e has type $\mathbb{B}$.

3.2 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 $l \equiv \texttt{x}$ is an identifier $\texttt{x}$, then the type of l under Γ is $\Gamma(\texttt{x})$.
- If $l \equiv \texttt{x[}e\texttt{]}$ is an offset expression e into identifier $\texttt{x}$, $\Gamma(\texttt{x}) = T$ $\textsf{array}$, 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 $s \equiv \texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{endif}$ has guard Boolean expression e and then-branch s_0, under Γ, e has type $\mathbb{B}$, and under Γ and ρ, s_0 is well-typed, then under Γ and ρ, s is well-typed.
- If $s \equiv \texttt{if}\ e\ \texttt{then}\ s_0\ \texttt{else}\ s_1\ \texttt{endif}$ has guard Boolean expression e, then-branch s_0, and else branch s_1, under Γ, e has type $\mathbb{B}$, and under Γ and ρ, s_0, and s_1 are well-typed, then under Γ and ρ, s is well-typed.
- If $s \equiv \texttt{while}\ e\ \texttt{do}\ s_0\ \texttt{enddo}$ has guard Boolean expression e and body s_0, and under Γ, e has type $\mathbb{B}$ and s_0 is well-typed, then under Γ, s is well-typed.
- If $s \equiv \texttt{for x :=}\ e_0\ \texttt{to}\ e_1\ \texttt{do}\ s_0\ \texttt{enddo}$ 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`, ... ,`e_n`)` is a call that runs procedure `f` on argument expressions $e_0, \ldots, e_n$, if $\Gamma(\mathtt{f}) = T_0 \times \ldots \times T_n \to T'$, 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.

3.3 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 $\mathsf{alias} = \mathtt{ids} \to \mathsf{Types}$. 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 $D \equiv$ `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 `x` to T (i.e., the alias map $A[\mathtt{t} \mapsto T]$).

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 $E \equiv$ `int`, then the actual type of E under A is $\mathbb{Z}$.

- If $E \equiv$ `float`, then the actual type of E under A is F.

- If $E \equiv$ `x`, then the actual type of E under A is $A(\mathtt{x})$.

- If $E \equiv$ `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 $D \equiv$ `var x : T ;`D' is a sequence of a variable declaration with a sequence of declarations, type expression `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 $\Gamma[\mathtt{x} \mapsto T]$.

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 $D \equiv$ `func f(x0 :` E_0`, ... , xn :` E_n`)` $:E'; D'$ 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 $\Gamma'[\mathtt{f} \mapsto T_0 \times \ldots \times T_n \to T']$.

Well-typedness of a function declaration A function declaration `func f(x0 :` E_0`, ... , xn :` E_n`) :` $E' s'$ `:` 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 $P \equiv$ `let` $D_T\ D_V\ D_F$ `in` s `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.

$$
\begin{aligned}
\text{program} &\rightarrow \texttt{let}\ \text{declseg}\ \texttt{in}\ \text{stmts}\ \texttt{end} && (1)\\
\text{declseg} &\rightarrow \text{typedecls vardecls funcdecls} && (2)\\
\text{typedecls} &\rightarrow \epsilon && (3)\\
\text{typedecls} &\rightarrow \text{typedecl typedecls} && (4)\\
\text{typedecl} &\rightarrow \texttt{type}\ \texttt{id}\ \texttt{:=}\ \text{type}\ \texttt{;} && (5)\\
\text{type} &\rightarrow \texttt{boolean} && (6)\\
\text{type} &\rightarrow \texttt{int} && (7)\\
\text{type} &\rightarrow \texttt{float} && (8)\\
\text{type} &\rightarrow \texttt{unit} && (9)\\
\text{type} &\rightarrow \texttt{id} && (10)\\
\text{type} &\rightarrow \texttt{array}\ \texttt{[}\ \texttt{intlit}\ \texttt{]}\ \texttt{of}\ \text{type} && (11)\\
\text{vardecls} &\rightarrow \epsilon && (12)\\
\text{vardecls} &\rightarrow \text{vardecl vardecls} && (13)\\
\text{vardecl} &\rightarrow \texttt{var}\ \text{ids}\ \texttt{:}\ \text{type optinit}\texttt{;} && (14)\\
\text{ids} &\rightarrow \texttt{id} && (15)\\
\text{ids} &\rightarrow \texttt{id}\ \texttt{,}\ \text{ids} && (16)\\
\text{optinit} &\rightarrow \epsilon && (17)\\
\text{optinit} &\rightarrow \texttt{:=}\ \text{const} && (18)\\
\text{funcdecls} &\rightarrow \epsilon && (19)\\
\text{funcdecls} &\rightarrow \text{funcdecl funcdecls} && (20)\\
\text{funcdecl} &\rightarrow \texttt{func}\ \texttt{id}\ \texttt{(}\ \text{params}\ \texttt{)}\ \texttt{:}\ \text{type}\ \texttt{begin}\ \text{stmts}\ \texttt{end}\ \texttt{;} && (21)\\
\text{params} &\rightarrow \epsilon && (22)\\
\text{params} &\rightarrow \text{neparams} && (23)\\
\text{neparams} &\rightarrow \text{param} && (24)\\
\text{neparams} &\rightarrow \text{param}\ \texttt{,}\ \text{neparams} && (25)\\
\text{param} &\rightarrow \texttt{id}\ \texttt{:}\ \text{type} && (26)\\
\text{stmts} &\rightarrow \text{fullstmt} && (27)\\
\text{stmts} &\rightarrow \text{fullstmt stmts} && (28)\\
\text{fullstmt} &\rightarrow \text{stmt}\ \texttt{;} && (29)\\
\text{stmt} &\rightarrow \text{lvalue}\ \texttt{:=}\ \text{expr} && (30)\\
\text{stmt} &\rightarrow \texttt{if}\ \text{expr}\ \texttt{then}\ \text{stmts}\ \texttt{endif} && (31)\\
\text{stmt} &\rightarrow \texttt{if}\ \text{expr}\ \texttt{then}\ \text{stmts}\ \texttt{else}\ \text{stmts}\ \texttt{endif} && (32)\\
\text{stmt} &\rightarrow \texttt{while}\ \text{expr}\ \texttt{do}\ \text{stmts}\ \texttt{enddo} && (33)\\
\text{stmt} &\rightarrow \texttt{for}\ \texttt{id}\ \texttt{:=}\ \text{expr}\ \texttt{to}\ \text{expr}\ \texttt{do}\ \text{stmts}\ \texttt{enddo} && (34)\\
\text{stmt} &\rightarrow \texttt{break} && (35)\\
\text{stmt} &\rightarrow \texttt{return}\ \text{expr} && (36)\\
\text{lvalue} &\rightarrow \texttt{id}\ \text{optoffset} && (37)\\
\text{optoffset} &\rightarrow \epsilon && (38)\\
\text{optoffset} &\rightarrow \texttt{[}\ \text{expr}\ \texttt{]} && (39)\\
\text{exprs} &\rightarrow \epsilon && (40)\\
\text{exprs} &\rightarrow \text{neexprs} && (41)\\
\text{neexprs} &\rightarrow \text{expr} && (42)\\
\text{neexprs} &\rightarrow \text{expr}\ \texttt{,}\ \text{neexprs} && (43)
\end{aligned}
$$

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

$$\text{expr} \rightarrow \text{clause} \tag{44}$$
$$\text{expr} \rightarrow \text{expr}\ \texttt{|}\ \text{clause} \tag{45}$$
$$\text{clause} \rightarrow \text{pred} \tag{46}$$
$$\text{clause} \rightarrow \text{clause}\ \texttt{\&}\ \text{pred} \tag{47}$$
$$\text{pred} \rightarrow \text{expr} \tag{48}$$
$$\text{pred} \rightarrow \text{expr boolop expr} \tag{49}$$
$$\text{boolop} \rightarrow \texttt{=} \tag{50}$$
$$\text{boolop} \rightarrow \texttt{<>} \tag{51}$$
$$\text{boolop} \rightarrow \texttt{<=} \tag{52}$$
$$\text{boolop} \rightarrow \texttt{>=} \tag{53}$$
$$\text{boolop} \rightarrow \texttt{<} \tag{54}$$
$$\text{boolop} \rightarrow \texttt{>} \tag{55}$$
$$\text{expr} \rightarrow \text{term} \tag{56}$$
$$\text{expr} \rightarrow \text{expr linop term} \tag{57}$$
$$\text{linop} \rightarrow \texttt{+} \tag{58}$$
$$\text{linop} \rightarrow \texttt{-} \tag{59}$$
$$\text{term} \rightarrow \text{factor} \tag{60}$$
$$\text{term} \rightarrow \text{term nonlinop factor} \tag{61}$$
$$\text{nonlinop} \rightarrow \texttt{*} \tag{62}$$
$$\text{nonlinop} \rightarrow \texttt{/} \tag{63}$$
$$\text{factor} \rightarrow \text{const} \tag{64}$$
$$\text{factor} \rightarrow \texttt{id} \tag{65}$$
$$\text{factor} \rightarrow \texttt{id}\ \texttt{[}\ \text{expr}\ \texttt{]} \tag{66}$$
$$\text{factor} \rightarrow \texttt{id}\ \texttt{(}\ \text{exprs}\ \texttt{)} \tag{67}$$
$$\text{factor} \rightarrow \texttt{(}\ \text{expr}\ \texttt{)} \tag{68}$$
$$\text{const} \rightarrow \texttt{intlit} \tag{69}$$
$$\text{const} \rightarrow \texttt{floatlit} \tag{70}$$
$$\text{const} \rightarrow \texttt{\_} \tag{71}$$

Figure 2: Grammar for Tiger expressions.