Wellesley College ◊ CS301 Compiler Design ◊ November 9, 2000
Handout #10 (Revised)

THE BOBCAT REFERENCE MANUAL

Important changes from the initial draft:

- In addition to being well-typed, a legal Bobcat program must not define the same global variable or function name twice (see Section 3). This constraint considerably simplifies the interpretation and compilation of Bobcat programs.
- `error` expressions now carry an explicit type. This significantly simplifies type checking. See the modified form for error in the grammar of Fig. 1 and the typing rules of Fig.2.
- The [relop] rule in Figure 2 has changed to allow comparison of two values at any base type (see Section 3 for a discussion). This obviates previously presented standard library functions for comparing characters (`charLt`, `charLeq`, `charGeq`, `charGt`).
- The return type of a function can be any expression type. This includes base types (as before) but also includes the new `void` type keyword (new). Function declarations not declaring an explicit return type effectively desugar into ones with a `void` return type (see Section 2.3).
- The previous typing rules were missing a rule for a top level program. This has now been added (the [globalLet] rule of Fig. 3), along with some text in Section 3 explaining it.
- The grammar for the top-level (program) `let` construct has been extended to allow multiple semi-colon separated expressions in the body. The previously posted grammar was also missing an `end` keyword.
- An ambiguity in the specification of the parsing of `if`, `while`, `for`, and assignment expressions has been resolved in Section 2.2.
- A new desugaring for top-level programs is given in Section 2.3. This supersedes the desugaring presented in the previous version of the Bobcat reference manual.

1. Overview

Bobcat is an extension to Kitty that supports global, first-order, mutually recursive functions and three primitive data types (booleans, characters, and integers). The key differences between Bobcat and Kitty are as follows:

1. Whereas all Kitty values are integers, Bobcat supports three types of values: booleans, characters, and integers. It is illegal to use values of one type in a context where a value of another type is expected. This property is enforced by static type checking rules (see Section 3), which can be used to reject ill-typed programs at compile-time. Only the semantics of well-typed programs are defined.

2. Bobcat allows the definition of collections of mutually recursive functions that can refer to global variables. Function parameters and results have explicitly declared types.

3. Bobcat supports a library of standard functions that subsume and extend Kitty's nullary and unary operators.

4. Bobcat has a simple construct for aborting the program upon encountering an error.

Like Kitty, Bobcat is essentially a subset of Appel's Tiger language.[1] The key features of Tiger not supported by Kitty are (1) record and array datatypes and (2) block structured function declarations.

[1] We say "essentially" because while Bobcat's functions are take from Tiger, Bobcat has a few features that Tiger does not have. In particular: Bobcat supports characters as a disintct datatype, which Tiger does not; (2) Bobcat has

We will use Bobcat as our source language in our second pass at exploring the stages of compilation. Once we complete this second pass, we will consider extensions to Bobcat that support additional Tiger features.

2. Syntax

2.1 Lexical Conventions

The lexical conventions for Bobcat are exactly the same as those of the Kitty language. See the Kitty Reference Manual for details.

2.2. Grammar

Syntactically well-formed core Kitty programs are those derivable from the grammar in Figure 1. We use the following conventions in the grammar:

- Bold unitalicized names are terminal symbols that stand for classes of tokens.
- Plain unitalicized names are terminal symbols that stand for single tokens with those names.
- Italicized names are non-terminal symbols in the grammar, whose meanings are defined by productions.
- Italicized annotations following the hash mark (#) are comments and not part of the grammar.
- If *toks* is a sequence of tokens, then *{toks}* stands for zero or more repetitions of *toks*.
- If *toka1* and *toka2* are sequences of tokens, then *{toks1 [toks2]}* stands for zero or more repetitions of *toks1* that are separated by *toka2*.

Note that Bobcat variable and function declarations are *not* separate by semi-colons.[2]

The grammar in Figure 1 is ambiguous. The ambiguities are removed by the following rules, which are the same as those in Kitty[3]:

> *Precedence:* The precedence of operators from highest to lowest is as follows (operators on the same line have the same precedence):
>
> unary minus (negation)
> `*, /, %`
> `+, -`
> `<, <=, =, <>, >=, >`
> `&`
> `|`
>
> *Associativity:* The operators `*, \, %, +, -, &`, and `|` are all *left-associative*. E.g., 1 – 2 + 3 is parsed as if it were written (1 – 2) + 3. The relational <, <=, =, <>, >=, and > are all *non-associative*. E.g., 1 < 2 = 3 is not a legal expression, even though the explicitly grouped versions (1 < 2) = 3 and 1 < (2 = 3) are legal expressions.

---

several standard library functions not suported by Tiger; (3) Bobcat's error construct does not appear in Tiger; (4) Bobcat's program syntax is more lenient than Kitty's; and (5) Bobcat inherits from Kitty a few non-Tiger features (constants, writes).

[2] Kitty declarations *were* separated by semi-colons, but this was a mistake on my part based on a misreading of the Tiger grammar. Tiger declarations are *not* separated by any symbols, either.

[3] The Kitty reference manual did not list the weak precedence of keywords rule, but it should have!

```
# Prog derives Bobcat expressions
Prog → let {GlobalDecl} in {Exp[;]} end # top-level program

# The following expxressions are inherited from Kitty
# Exp derives Bobcat expressions
Exp → () # the literal for "no value"
Exp → intlit # as specified by the lexical conventions for integer literals
Exp → charlit # as specified by the lexical conventions for character literals
Exp → id # as specified by the lexical conventions for identifiers
Exp → Const
Exp → -Exp # unary minus operator
Exp → writes (stringlit)
Exp → Exp Binop Exp
Exp → id := Exp # assignment
Exp → if Exp then Exp else Exp
Exp → if Exp then Exp
Exp → let {VarDecl} in {Exp [;]} end
Exp → while Exp do Exp
Exp → for ident := Exp to Exp do Exp
Exp → (Exp ; {Exp[;]}) # sequence expression, parentheses required
Exp → (Exp) # grouping via optional parentheses

# The following expressions are new to Bobcat
Exp → id ({Exp [,]}) function application
Exp → error ExpTy stringlit

# VarDecl derives variable declarations
VarDecl → var id := Exp
VarDecl → var id : BaseType := Exp

# VarDecl derives variable declarations
FunDecl → function id ({Formal[,]}) : ExpType = Exp
FunDecl → function id ({Formal[,]}) = Exp

# Formal derives formal
# argument declarations
Formal → id : BaseType

# GlobalDecl derives global
# variable or function decls
GlobalDecl → VarDecl
GlobalDecl → FunDecl

# ExpType derives expr. types
ExpType → BaseType
ExpType → void

# BaseType derives base types
BaseType → bool
BaseType → char
BaseType → int

# Const derives constants
Const → minint
Const → maxint
Const → true
Const → false

# Binop derives binary operators
Binop → Arithop
Binop → Relop
Binop → Logop

Arithop → +
Arithop → -
Arithop → *
Arithop → /  # integer division
Arithop → %  # integer modulus

Relop → <
Relop → <=
Binop → =
Binop → <> # not equals
Relop → >=
Relop → >

Logop → &  # short circuit and
Logop → |  # short circuit or
```

Figure 1: Bobcat Grammar

Dangling Else: The presence of both if-then and if-then-else expressions in a language introduces an ambiguity as to which `if` expression an `else` clause belongs. The Kitty convention (as in many other languages) is that an else clause belongs to the innermost if expression enclosing it. Thus, the expression

`if` E_1 `then if` E_2 `then` E_3 `else` E_4

is parsed as if it were written

`if` E_1 `then (if` E_2 `then` E_3 `else` E_4`)`

Weak precedence of keyword expressions: It is assumed that keyword expressions (`if`, `while`, `for`) bind less tightly than binary operator expressions. In other words, the final expression of a keyword expression should be interpreted as extending as far right as possible. For these purposes, assignment is considered a keyword expression. For example:

Expression	**parses as**	**and not as**
`if` *E1* `then` *E2* `+` *E3*	`if` *E1* `then (`*E2* `+` *E3*`)`	`(if` *E1* `then` *E2*`) +` *E3*
`if` *E1* `then` *E2* `else` *E3* `+` *E4*	`if` *E1* `then` *E2* `else (`*E3* `+` *E4*`)`	`(if` *E1* `then` *E2* `else` *E3*`) +` *E4*
x `:=` *E1* `+` *E2*	*x* `:= (`*E1* `+` *E2*`)`	`(`*x* `:=` *E1*`) +` *E2*
`while` *E1* `do` *x* `:=` *E2* `+` *E3*	`while` *E1* `do` *x* `:= (`*E2*`+`*E3*`)`	`(while` *E1* `do` *x* `:=` *E2*`)+`*E3*
`for` *i* `:=` *E1* `to` *E2* `do` *x* `:=` *E3* `+`*E4*	`for` *i* `:=` *E1* `to` *E2* `do` *x* `:= (`*E3* `+`*E4*`)`	`(for` *i* `:=` *E1* `to` *E2* `do` *x* `:=` *E3*`) +`*E4*

2.3. Desugaring

It is inconvenient to write an explicit `let` … `in` … `end` for every top-level Bobcat program. Following the conventions of the C programming language, Bobcat supports the following desugaring for Bobcat programs:

GlobalDecl$_1$ `...` *GlobalDecl*$_n$ `let` *GlobalDecl*$_1$ `...` *GlobalDecl*$_n$ `in main() end`

That is, a Bobcat program can be written as a *non-empty* sequence of declarations, one of which is assumed to be a declaration of a function `main` of type `() → void`. Executing the program is equivalent to invoking `main` with no arguments. It is also convenient to be able to process single Kitty expressions as if they were Bobcat programs. For this reason, Bobcat also supports the following desugaring for top-level expressions into Bobcat programs: *Exp* `let in` *Exp* `end`

Several of the Bobcat expressions in Figure 1 can be viewed as desugaring into other expressions, as illustrated in the following table:

Source language expression	**Expression after desugaring**

`for` I_1 `:=` E_1 `to` E_2 `in` E_3	`let var` I_1 `:=` E_1
	`var` I_2 `:=` E_2 `(* Assume` I_2 `fresh *)`
	`in while (`I_1 `<=` I_2`) do`
	`(`E_3`;` I_1 `:=` I_1 `+ 1)`
	`end`

`if` E_1 `then` E_2	`if` E_1 `then` E_2 `else ()`
E_1 `&` E_2	`if` E_1 `then` E_2 `else false`
E_1 `|` E_2	`if` E_1 `then true else` E_2
`-` E_1	`0 -` E_1

`writes("abc...")`	`(writec('\a');`
	`writec('\b');`
	`writec('\c');`
	`...)`

Note that the desugaring for E_1 | E_2 is different than that used in Kitty. This is because the operands of | in Kitty are integers, while the operands of | in Bobcat must be boolean. For example, `2 | 3` is a legal Kitty program that should evaluate to 2, but it is an illegal Bobcat program.

Unlike Tiger, Bobcat allows the `void` expression type to be explicitly written in a program. Because of this, the function declarations that do not list an explicit return type, e.g., those of the form

id (**id**$_1$: *BaseType*$_1$, … , **id**$_n$: *BaseType*$_n$),

effectively desugar into declarations with `void` as an explicit return type:

id (**id**$_1$: *BaseType*$_1$, … , **id**$_n$: *BaseType*$_n$) : void

3. Static Semantics

The *static semantics* of a language describes those properties that can be determined at compile-time – i.e., without executing the program. A Bobcat program has to satisfy two static properties in order to be considered *valid*:

- The global variable and function names in a program must be pairwise distinct and also distinct from any standard library function name. For instance, the name `f` cannot be declared twice as a global variable and/or function, nor can the name `abs` be declared as a global variable or function (because it is already in the standard library).

- A program must be well-typed according to the typing rules shown in Figures 2 and 3. These are explained further below.

The typing rules for Bobcat refer to types in the following domains:

BT *BaseTy* = {`bool`, `int`, `char`}
ET *ExpTy* = *BaseTy* {`void`}
FT *FunTy* = *BaseTy* * × *ExpTy*

The metavariables *BT*, *ET*, and *FT* will be used to range over elements of these types. The notation (BT_1, … BT_n) → *ET* will be used to stand for a function type in *FunTy*.

The typing rules use type environments, which are finite partial mappings from variable names to *BaseType* *FunctionType*. If *A* is a type environment, then the notation *A(x)* refers to the base type or function type associated with variable *x* in the environment (if there is such a type). The notation*A*[x_1 := T_1, …, x_n := T_n] denotes a type environment that, for all *i* in the range [1..n] binds x_i to T_i (a base type or function type) and binds every *y* not in the set { x_1 ,… , x_n }to *A(y)*.

The typing rules involve four different kinds of judgements:

- The judgement ◊ *Prog* claims that program *Prog* is statically valid in terms of its types.

- The judgement *A* ▷ *Exp* : *ET* claims that expression *Exp* has type *ET* with respect to type environment *A*..

- The judgement $A \clubsuit Decl_1 \ldots Decl_n \blacklozenge A'$ claims that type environment A can be extended to type environment A' by processing a sequence of declarations, where each $Decl_I$ is either a single variable declaration or a mutually recursive set of function declarations.

- The judgement $A \quad Decl \spadesuit A'$ claims that A can be extended to A' by processing a "single" declaration $Decl$ (which may be a single variable declaration or a mutually recursive set of function declarations).

The "claims" made by judgements are not necessarily true. For instance, the following is a well-formed judgement that is false: `{} ▷ 1 : bool.` (Here, {} stands for the empty type environment.) A judgement is considered true if and only if there is derivation (i.e., a finite proof tree) consisting of instantiations of the typing rules in which every hypothesis of a rule instantiation is the conclusion of some other rule instantiation. An expression *Exp* is said to *be well-typed with respect to a type environment A* if there exists an expression type *ET* such that the judgement $A \triangleright E : ET$ can be derived from instantiations of the typing rules. A program *Prog* is said to be *well-typed* if the judgement $\Diamond Prog$ can be derived from instantiations of the typing rules.

The complete typing rules for Bobcat appear in Figures 2 and 3. The figures show typing rules only for "core" expressions; expressions that can be derived by desugaring are first assigned to be desugared before they are type-checked. The [group] rule is included for completeness only because all the other rules are expressed in terms of the concrete syntax of Bobcat. There would be no need for the [group] rule in the abstract syntax for Bobcat.

The [relop] rule specifies that the Bobcat relational operators can be used to compare any two values that have the same base type. For instance, all of the following expressions are legal in a well-typed Bobcat program: `5 < 3`, `'h' < 'q'`, `true < false`. However, any attempt to compare values of different types, as in `5 < 'q'`, `false < 'q'`, or `true < 3`, is a type error. Such operators, which have different meanings at different types, are said to be *overloaded*.

Some of the typing rules have side conditions, which are conditions that must be true in order for the rule to be applicable. The use of a metavariable in a side condition implies that the side condition is only true if the metavariable can be properly instantiated. For example, the following side condition appears in the [varRef] and [assign] rules:

where $A(\mathbf{id}) = BT$

This side condition means that the type associated with **id** in the type environment A must be a base type, and that this base type will be named BT when used elsewhere in the rule.

$$\text{[nothing]}\ \frac{}{A \rhd () : \text{void}} \qquad \text{[intLit]}\ \frac{}{A \rhd \textbf{intlit} : \text{int}} \qquad \text{[charLit]}\ \frac{}{A \rhd \textbf{charlit} : \text{char}}$$

$$\text{[trueConst]}\ \frac{}{A \rhd \text{true} : \text{bool}} \qquad \text{[falseConst]}\ \frac{}{A \rhd \text{false} : \text{bool}}$$

$$\text{[maxintConst]}\ \frac{}{A \rhd \text{maxint} : \text{int}} \qquad \text{[minintConst]}\ \frac{}{A \rhd \text{minit} : \text{int}}$$

$$\text{[arithop]}\ \frac{A \rhd Exp_1 : \text{int} \ ; \ A \rhd Exp_2 : \text{int}}{A \rhd Exp_1\ Arithop\ Exp_2 : \text{int}}$$

$$\text{[relop]}\ \frac{A \rhd Exp_1 : BT;\ A \rhd Exp_2 : BT}{A \rhd Exp_1\ Relop\ Exp_2 : \text{bool}}$$

$$\text{[varRef]}\ \frac{}{A \rhd \textbf{id} : BT} \ \text{where } A(\textbf{id}) = BT$$

$$\text{[assign]}\ \frac{A \rhd Exp : BT}{A \rhd \textbf{id} := Exp : \text{void}} \ \text{where } A(\textbf{id}) = BT$$

$$\text{[if]}\ \frac{A \rhd Exp_1 : \text{bool} \ ; \ A \rhd Exp_2 : ET \ ; \ A \rhd Exp_3 : ET}{A \rhd \text{if } Exp_1 \text{ then } Exp_2 \text{ else } Exp_3 : ET}$$

$$\text{[while]}\ \frac{A \rhd Exp_1 : \text{bool} \ ; \ A \rhd Exp_2 : \textit{void}}{A \rhd \text{while } Exp_1 \text{ do } Exp_2 : \textit{void}}$$

$$\text{[seq]}\ \frac{\text{i } [1..n] \ .\ A \rhd Exp_i : ET_i}{A \rhd (Exp_1; \ldots; Exp_n) : ET_n}$$

$$\text{[localLet]}\ \frac{A \clubsuit VarDecl_1 \ldots VarDecl_k \blacklozenge A';\quad \text{i } [1..n] \ .\ A' \rhd Exp_i : ET_i}{A \rhd \text{let } VarDecl_1 \ldots VarDecl_k \text{ in } Exp_1, \ldots, Exp_n \text{ end} : ET_n}$$

$$\text{[funApp]}\ \frac{\text{i } [1..n] \ .\ A \rhd Exp_i : BT_i}{A \rhd \textbf{id}\ (Exp_1, \ldots, Exp_n) : ET} \ \text{where } A(\textbf{id}) = (BT_1, \ldots, BT_n) \rightarrow ET$$

$$\text{[group]}\ \frac{A \rhd Exp : ET}{A \rhd (Exp) : ET} \qquad \text{[error]}\ \frac{}{A \rhd \text{error } ET\ \ (\textbf{stringlit}) : ET}$$

Figure 2: Typing Rules, Part 1

$$\text{[decls]}\ \frac{i\ [1..n]\ .\ A_{i-1}\ \ GlobalDecl_i \spadesuit A_i}{A_0 \clubsuit GlobalDecl_1\ \ldots\ GlobalDecl_n \diamondsuit A_n}$$

$$\text{[varDeclUntyped]}\ \frac{A \rhd Exp\ :\ BT}{A\ \ \texttt{var}\ \mathbf{id}\ :=\ Exp \spadesuit A[\mathbf{id} := BT]}$$

$$\text{[varDeclTyped]}\ \frac{A \rhd Exp\ :\ BT}{A\ \ \texttt{var}\ \mathbf{id}\ :\ BT\ :=\ Exp \spadesuit A[\mathbf{id} := BT]}$$

$$\text{[funDecls]}\ \frac{i\ [1..n]\ .\ A' \rhd Exp_i\ :\ ET_i}{\begin{array}{l} A\ \ \texttt{function id}_1\texttt(\texttt{id}_{(1,1)}\texttt:BT_{(1,1)}\texttt,\ldots\texttt,\texttt{id}_{(1,k1)}\texttt:BT_{(1,k1)}\ \texttt):ET_1\ =\ Exp_1 \\ \quad\ldots \\ \quad\texttt{function id}_n\texttt(\texttt{id}_{(n,1)}\texttt:BT_{(n,1)}\texttt,\ldots\texttt,\texttt{id}_{(n,k1)}\texttt:BT_{(n,kn)}\ \texttt):ET_n\ =\ Exp_n \spadesuit A' \\ \text{where } A' = A[\texttt{id}_1\ :=\ (BT_{(1,1)},\ldots,BT_{(1,k1)})\ \rightarrow\ ET_1, \\ \qquad\qquad \ldots, \\ \qquad\qquad \texttt{id}_n\ :=\ (BT_{(n,1)},\ldots,BT_{(n,kn)})\ \rightarrow\ ET_n] \end{array}}$$

$$\text{[gloabalLet]}\ \frac{A_{\text{stdlib}} \clubsuit GlobalDecl_1\ \ldots GlobalDecl_k \diamondsuit A';\ A' \rhd Exp\ :\ \text{void}}{\lozenge\ \texttt{let}\ GlobalDecl_1\ \ldots GlobalDecl_k\ \texttt{in}\ Exp\ \texttt{end}}$$

where A_{stdlib} is a type environment for the Bobcat standard library

Figure 3: Typing Rules, Part 2

4. Dynamic Semantics

The *dynamic semantics* of a language describes the run-time meaning of phrases in the language. The dynamic semantics of Bobcat is similar to that of Kitty, except for the following differences:

- Meaning is only ascribed to valid Bobcat programs. In contrast, every Kitty expression has a meaning. This simplifies the implementation of Bobcat interpreters and compilers, which may be predicated on the assumption that only valid programs will be interpreted/compiled. For example, when adding two values, a Bobcat interpreter need not check that both are integers, since this fact has already been proven by the type checker.

- A Bobcat expression can denote one of four kinds of values: a boolean, a character, an integer, or “no value”. In contrast, a Kitty program can only denote two kinds of values: an integer or “no value”.

- Bobcat supports the declaration and application of global, mutually recursive functions. The meaning of a function application **id**(*E*$_1$, ..., *E*$_n$) is determined via the following steps:

    1. Evaluate the actual parameters *E*$_1$, . . . , *E*$_n$ from left to right into values *V*$_1$, . . . , *V*$_n$. (In a valid program, it is guaranteed all such values will be of base type.)

    2. Suppose that there is a corresponding global function declaration

        function **id**(**id**$_1$:*E*$_1$, ..., **id**$_n$:*E*$_n$): *T* = *E*$_{body}$

        (In a valid program, it is guaranteed that there is such a declaration, and it is unique.) Evaluate the body expression *E*$_{body}$. in a context where the formal parameters **id**$_1$, ..., **id**$_n$ stand for the actual parameter values *V*$_1$, . . . , *V*$_n$ and any non-shadowed global variables stand for their global value. In the case where T is a base type, the result of the evaluation of body will be a value of type *T*, and this value should be returned as the value of the function application. In the case where *T* is `void`, both the body and the function application will have no value.

- The Bobcat standard I/O library `readc` function (see Section 5) always returns a character. In contrast, the Kitty `readc` unary operator returns an integer, where an integer in the range 0-255 is interpreted as a character with that ASCII value and –1 is interpreted as indicating the end-of-file. Since there is no end-of-file character, Bobcat needs a separate standard I/O library function `eof()` to detect the end-of-file condition.
- For any Kitty expression *E*, if the Bobcat program `let in` *E* `end` is valid, then this program has the same behavior as the Kitty expression as long as *E* does not contain an invocation of `readc()`.

5. Standard Library Functions

Top-level Bobcat programs are assumed to be type-checked and executed relative to an "top-level environment" that includes bindings for functions in a standard library. Separating the specification for the standard library from that of the core language simplifies both the description and implementation of the language. It also provides a modular way to extend the language with new library functions without having to change the specification or implementation of the core language.

This section documents the functions in the Bobcat standard library. They are loosely organized into categories of related functions:

Mathematical Functions

`abs (x: int): int`
Returns the absolute value of `x`.

`expt (base: int, power: int): int`
Returns the result of raising `base` to `power`. Signals an error if power < 0.

`sqr (x: int): int`
Returns the result of squaring `x`.

Logic Functions

`not (b: bool): bool`
Returns the logical negation of `b`.

`and (b1: bool, b2: bool): bool`
Returns the logical conjunction of `b1` and `b2`. Unlike the short-circuit conjunction operator `&`, `and` always evaluates both arguments.

`or (b1: bool, b2: bool): bool`
Returns the logical disjunction of `b1` and `b2`. Unlike the short-circuit disjunction operator `|`, `or` always evaluates both arguments.

`boolToInt (b: bool): int`
Returns `0` for `true` and `1` for `false`.

`intToBool (i: int): bool`
Returns `true` for `0` and `false` for any other integer.

Character Functions

`chr (i: int): char`
Returns the character corresponding to ASCII value `i`.

`ord (c: char): int`
Returns the ASCII value of character `c`.

`digitToInt (d: char): int`
If `d` is a digit character, returns the numerical value of the digit; otherwise signals an error.

`intToDigit (i: int): char`
If `i` is a number in the range [0..9], returns the digit character corresponding to the number; otherwise, signals an error.

`isDigit (c: char): bool`
Returns `true` if `c` is a digit, and `false` otherwise.

`isLetter (c: char): bool`
Returns `true` if `c` is a letter, and `false` otherwise.

`isLowercase (c: char): bool`
Returns `true` if `c` is a lowercase letter, and `false` otherwise.

`isUppercase (c: char): bool`
Returns `true` if `c` is an uppercase letter, and `false` otherwise.

`isWhitespace (c: char): bool`
Returns `true` if `c` is a whitespace character, and `false` otherwise.

`lowercase (c: char): char`
If `c` is an uppercase character returns the corresponding lowercase character; otherwise returns `c`.

`uppercase (c: char): char`
If `c` is a lowercase character returns the corresponding uppercase character; otherwise returns `c`.

`eof : () → bool`
Returns `true` if the standard input stream is empty (i.e., if performing `readc` would signal an error) and returns `false` otherwise.

`readc : () → char`
Consumes and returns the next character from standard input stream. Signals an error if standard input is empty.

`readi : (int) → int`

`readi(default)` first consumes any whitespace characters (spaces, tabs, and newlines) in the standard input stream. If the first non-whitespace character is a digit, or a minus sign followed by a digit, the maximal sequence of characters interpretable as an integer is consumed, and the integer represented by these characters is returned. However, if the first non-whitespace character is not the first character of an integer representation, it is not consumed, and the value *`default`* is returned. Thus, *`default`* serves as an indication of the failure of `readi` to read an integer. It is helpful to parameterize over this value because different failure values are suitable in different contexts.[4] For instance, 0, -1, `minint`, and `maxint` are all typical values of the argument to `readi`.

`writec : (char) → void`
`writec(c)` writes character c to standard output stream.

`writei : (int) → void`
`writei(i)` writes a string representation of integer I to standard output stream.

6. Examples

6.1 Count

Here is a Bobcat program that counts the number of characters in the standard input stream:

```
let
  var count := 0
  function inc() = count := count + 1
in
  while not(eof()) do (readc(); inc());
  writei(count)
end
```

By the program desugaring in Section 2.3, this can also be written as:

```
var count := 0

function inc() = count := count + 1

function main () : void =
  (while not(eof()) do (readc(); inc());
   writei(count))
```

[4] Of course, any attempt to encode a failure value as an integer necessarily means that there can be an ambiguity between actually reading that integer and reading no integer. In more advanced languages, this problem can be dealt with by either returning a compound data structure (such as ML's int option) that can represent failure as a value distinct from the integers, or by raising an exception when no integer is read.

Since mutually recursive functions can be in any order, this can be rewritten as:

```
var count := 0

function main () : void =
  (while not(eof()) do (readc(); inc());
   writei(count))

function inc() = count := count + 1
```

However, both of the following are invalid: the first because `inc` is unbound in the body of `main`, due to the fact that the `count` variable declaration breaks up the mutual recursion between `main` and `inc`; and the second because the variable `count` is unbound in the bound of `inc`.

```
/* Invalid program 1 */
function main () : void =
  (while not(eof()) do (readc(); inc());
   writei(count))

var count := 0

function inc() = count := count + 1


/* Invalid program 2 */
function main () : void =
  (while not(eof()) do (readc(); inc());
   writei(count))

function inc() = count := count + 1

var count := 0
```

6.2 Even/Odd

Here is a program that uses the classic mutually recursive definitions of functions that compute the evenness and oddness of an integer:

```
function main() = writei(test())

function test (): int =
  let var sum := 0
   in for i := 0 to 3 do
          sum := sum + expt(10,2*i) * boolToInt(even(2*i))
                     + expt(10,2*i+1) * boolToInt(even(2*i+1));
     sum
  end

function even (n: int): bool =
  if n = 0 then
    true
  else
    odd(n-1)

function odd (n: int): bool =
  if n = 0 then
    false
  else
    even(n-1)
```

6.3 Chartable

Below is a program that prints a table of the characters whose ASCII values are between 1 and 128.

```
function isSpecial(c: char): bool =
  c = '\t' | c = '\n' | c = '\'' | c = '\\'

function specialToChar (c: char): char =
  if c = '\t' then 't'
  else if c = '\n' then 'n'
  else if c = '\'' then '\''
  else if c = '\\' then '\\'
  else (writec(c); error char "is not a special")

function writeChar (c: char): void =
  (writec('\'');
   if isSpecial(c) then
      (writec('\\'); writec(specialToChar(c)))
   else
      writec(c);
   writec('\'')
   )

function main () =
  for i := 0 to 127 do
    (writeChar(chr(i));
     if (i % 8) = 7 then writec('\n') else writec('\t')
     )
```