# Bali++ Specifications

CS212

Fall 2005

## 1 Introduction

### 1.1 Bali Specifications

As discussed in lecture, we can specify a language in terms of its syntax (spelling and structural rules) and semantics (meaning). Although formal description methods exist for both kinds of specifications, we will only formalize the Bali syntax, as developed in Section 2. Bali's semantics are informally defined in Section 3.

### 1.2 Notation

We use the following notation throughout this document:

- Specific language elements, like keywords, operators, and punctuation, are shown as **`bold`**.
- Other terminals are shown as ***`bolditalic`***, such as names and integers, because they can have different values.
- Non-terminals in production rules are shown as plain red.
- Square brackets are used to denote optional items. [ **`int`** ] indicates that the keyword **`int`** can be present but is not required.
- A single | denotes that either the item on the left or the right can be present. So **`a`** | **`b`** indicates that either **`a`** or **`b`** must be present, but not both.
- Parentheses are used to group options. Thus, ( **`a`** | **`b`** ) **`c`** indicates that **`c`** must be prefixed by either **`a`** or **`b`**. Square brackets can also be used for grouping with the difference that the group would be optional.
- An asterik (*) is used to indicate zero or more occurrences of this item and a plus (+) is used to indicate one or more occurrences of this item.
- An arrow (→) indicates a production rule, which means *can be expressed as*.

# 2 Bali Syntax

## 2.1 Functions

| | | |
|---|---|---|
| program | → [ globalVars ] ( function \| prototype )* | program has globals and functions/prototypes |
| globalVars | → **`global`** varBlock | global variables |
| prototype | → functionHeader **`;`** | function prototype |
| function | → functionHeader varBlock statementBlock | function has header, variables and statements |
| functionHeader | → type name **`(`** [ params ] **`)`** | header |
| type | → ( **`int`** \| **`boolean`** \| **`char`** )[ **`*`** ]* | primitive/pointer types |
| type | → **`void`** ( **`*`** )+ | void pointer type |
| params | → type name ( **`,`** type name )* | one or more parameters |
| varBlock | → **`{`** varDecl* **`}`** | block of variable declarations |
| statementBlock | → **`{`** statement* **`}`** | block of statements |
| varDecl | → type name ( **`,`** name )* **`;`** | can declare more than one variable |

## 2.2 Statements

| | | |
|---|---|---|
| statement | → statementBlock | block statement |
| statement | → **`if (`** expression **`)`** statement [ **`else`** statement ] | conditional statement |
| statement | → **`do`** statement **`until (`** expression **`);`** | do..until loop |
| statement | → **`for (`** [ expression ] **`;`** [ expression ] **`;`** [ expression ] **`)`** statement | for loop |
| statement | → expression **`;`** | expression statement |
| statement | → **`print`** expression **`;`** | output |
| statement | → **`return`** expression **`;`** | return statement |
| statement | → **`;`** | empty statement |

## 2.3 Expressions

| | | |
|---|---|---|
| expression | → expPart [ binop expPart ] | basic expression |
| expression | → expPart **=** expression | assignment |
| expression | → expPart **[** expression **]** | array element |
| expPart | → unaryop expPart | unary operation |
| expPart | → ***int*** \| ***boolean*** \| ***'char'*** \| **null** | constant |
| expPart | → **readInt()** | integer input |
| expPart | → **readChar()** | character input |
| expPart | → **malloc (** expression **)** | memory allocation |
| expPart | → **free (** expression **)** | memory deallocation |
| expPart | → **<** type **>** expPart | type cast |
| expPart | → name | variable |
| expPart | → name args | function call |
| expPart | → **(** expression **)** | nested expression |
| binop | → arithmeticOp \| comparisonOp \| booleanOp | operators |
| arithmeticOp | → **+** \|**–**\| ***** \| **/** \| **%** | arithmetic operators |
| comparisonOp | → **==** \| **!=** \| **>** \| **<** \| **>=** \| **<=** | comparison operators |
| booleanOp | → **&&** \| **\|\|** \| **^** | boolean operators (**&&** and **\|\|** are short-circuiting) |
| unaryOp | → **–** \| **!** \| pointerOp | unary operators |
| pointerOp | → **&** \| ***** | pointer operators |
| args | → **(** [ expression ( **,** expression )* ] **)** | function call arguments |
| name | → (**a–z**\|**A–Z**) (**a–z**\|**A**–**Z**\|_\|**0–9**)* | names must begin with a letter |

# 3 Bali Semantics

## 3.1 Reserved Keywords

The following keywords may not be used as variable or function names: `global`, `int`, `boolean`, `char`, `void`, `true`, `false`, `if`, `else`, `do`, `until`, `for`, `print`, `null`, `return`, `free`, `malloc`, `readInt`, and `readChar`.

## 3.2 Variable Scope

The scope of a variable depends on where it is defined. Variables defined in the global block are visible throughout the program. Function arguments and function variables are visible in the function in which they are defined. No two variables in the same scope can have the same name (i.e. no two global variables can have the same name and for a given function no argument or function variable can have the same name as another argument or function variable in that same function). Global variables can be redefined inside a function (variable shadowing). Variables can have the same name as functions.

## 3.3 Expressions

### Operators

Variable types can only be mixed in expressions under certain circumstances. Logical operators accept and produce booleans. The equality and inequality operators accept all types and produce a boolean result. However, the types of the two operands must be the same. All other comparison operators accept only integers and produce a boolean result. Division is integral for integers. Arithmetic operators accept either two integers or one pointer and one integer. If an arithmetic operator acts on an integer and a pointer, then it produces a pointer of the same type as the input pointer. Only the addition and subtraction operations can be used for this type of mixed expression. The `&` reference operator works on any value with type $T$ and produces a pointer to that value with type $T$-pointer. This operator only accepts variables. The `*` dereference operator works on a value with type $T$-pointer and returns the value pointed at as type $T$. Void-pointer cannot be dereferenced.

The assignment operator must have a valid variable on the left that is the same type as the output of the expression on the right. It assigns the value of the expression on the right to the variable on the left and produces a result of the same type and value as the value assigned. The assignment operator can also accept as a left value a dereferenced pointer (a pointer being operated on by `*`). In this case, the right side of the expression is assigned to the address the pointer is pointing to. A pointer of type $T$-pointer on the left side of the expression must be accompanied by an expression of type $T$ on the right side.

The `[ ]` array access operator requires a dereferencable (non-void) pointer variable on the left side, and an integer expression inside it. It adds the integer on the inside to the pointer on the left side, and produces a pointer of the same type. Then it dereferences that pointer, and returns the result. `Arr[k]` is semantically equivalent to `*(Arr + k)`, and the same restrictions apply.

### Execution Order

Binary operations should be evaluated starting with the expression on the left and then the expression on the right. So for `a + b`, `a` should be evaluated first, then `b`, and then their results should be added. Likewise, function arguments should be evaluated from left to right. Assignment expressions should be evaluated from right to left.

### Casts

The type of expressions that can be cast is restricted to pointers. The cast expression part changes the type of the input to the one specified in the cast. Any pointer type can be cast to any other pointer type. However, casts between non-pointers and pointers (as well as two non-pointer types) are not allowed.

### Constants

Any Java integer is a constant of type `int`. The keywords `true` and `false` are constants of type `boolean`. Any Java character delimited by single quotes (') is a constant of type `char`. The `null` keyword is a constant of type `void*`, which represents the memory address `0`.

## 3.4 Statements

Each statement type that requires expressions requries a particular type of expression. `if` and `do..until` statements require an expression returning a boolean value. The `for` statement's second expression must return a boolean value. A `print` statement requires an expression returning an integer, character, or character pointer. It uses the appropriate instructions to generate integer, character, or string output, respectively. A `return` statement requires an expression returning a type identical to that of the function return type. Finally an expression statement can have an expression with any return type, since the return value is eliminated.

Bali supports two kinds of loop statements. The `do..until` loop executes the statement, and loops if its condition evaluates to false. The `for` loop executes the first expression (if present) before entering the loop cycle. Then it checks the second expression, and if true, executes the statement. If there is no second expression, true is assumed. Then the third expression is executed (if present), and the code loops.

## 3.5 Functions

- All functions contain an implicit return statement at the end of the function that returns `0` for integers, `'\0'` for characters, `null` for pointers, and `false` for booleans. Thus a function will return something even if no specific return statement is encountered.
- There must be one function called `main` with a return type of `int` and no parameters. This function will be called on startup.
- There are four built-in functions. The `readInt and readChar` functions request an integer or a character from the user, respectively. The `malloc` function allocates a block of memory of the provided size (which must be an integer) and returns a pointer of type `void*`. The `free` function deallocates a block of memory, determined by the argument passed. The argument passed to `free` should be a `void*` pointer to the start of a memory block that has been previously allocated by `malloc`. The return value of `free` is always the integer `0`.
- A function is considered declared after the prototype grammar structure occurs and that same function is considered defined after the function grammar structure with an identical name occurs.
- With the exception of main and built-in functions, functions must be declared before they are called or defined. Furthermore, these functions should be defined exactly once and declared exactly once. A function declaration and definition must match in name, return type, and parameter number and types (i.e. only the names of the parameters can differ). Note: these rules therefore forbid function overloading.

- Built-in functions are implicitly declared and defined and may be called anywhere. Therefore they should not be explicitly declared or defined anywhere. Main on the otherhand is only implicitly declared and thus may be called anywhere but needs to be defined exactly once.