CHOMP 🍪

A Language for Bit Manipulation

Evelyn Vu (evelyn.vu@tufts.edu)
Nicole Ogen (nicole.ogen@tufts.edu)
Luella Sugiman (luella.sugiman@tufts.edu)

Table of Contents

Introduction

CHOMP is an imperative procedural language with a static type system, supporting lexical variable scope, lists and higher order functions. The language supports special data types, the Bin types, that abstractly represents bits, nibbles, bytes, and words. It's designed to allow users to operate on a bit level, and help the user work with and visualize bits and bitwise functions.

Notation

In the syntax notation used in this manual, syntactic categories are indicated by *italic* type, and literal code in `Courier New`. Alternatives are listed on separate lines. Syntactic categories are optional when they contain the subscript opt. For example, $\textit{expression}_{opt}$ is an optional expression. This means either nothing or an expression can be there.

Lexical conventions

Comments

The characters `/*` introduce a comment, which terminates with the characters `*/`.

Identifiers

Identifiers must start with a lowercase or uppercase letter, then it can be followed by one or more combinations of lowercase letters, uppercase letters, numbers, or underscores (_).

Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:
`list int bool void char bit nibble byte word if else for while return true false null cons cdr car print`

Chars

A char is one character surrounded by single quotes (`'`).

Types

CHOMP is a statically typed language, so each variable must be declared with a type, and it should remain the same type throughout its scope.

- ➢ `int` values are integer values in the range of -2^{31} to 2^{31} - 1.
- ➢ `bool` values are either `true` or `false`.
- ➢ `void` type is used to represent the absence of a value.
- ➢ `char` values are any single ASCII characters.
- ➢ `list` values are a mutable list of elements where all elements are of the same type.

- Bin types: a “bin” type is a general term that applies to any of the types below, it represents signed numerals (note that two’s complement is applied)
  - `bit` value is a binary value that can fit in a bit
  - `nibble` value is a binary value that can fit in 4 bits
  - `byte` value is a binary value that can fit in 8 bits
  - `word` value is a binary value that can fit in 16 bits

Statements

Expression statement

expression`;`

The *expression* is evaluated.

Compound statement

`{` *stmt_list* `}`

stmt_list:
/* nothing */
stmt_list stmt

Conditional statements

There are two forms of conditional statements:
`if (` *expression* `) {` *statement* `}`
`if (` *expression* `) {` *statement-1* `} else {` *statement-2* `}`

If the evaluated *expression* is non-zero, the first *statement* is executed. In the 2nd form of conditional statement, if the evaluated *expression* is zero, the second *statement* is executed. The else and subsequent statement is attached to the last encountered if.

While

`while (` *expression* `)` *statement*

If the *expression* evaluates to non-zero, the *statement* will be executed. The *expression* test takes place before each execution of the *statement.*

For

`for (` *expression-1*$_{opt}$*; expression-2; expression-3*$_{opt}$ `) {` *statement* `}`

Expression-1 initializes the loop. *Expression-2* specifies a test that's made before each iteration, and the loop is executed until *expression-2* evaluates to 0. *Expression-3* performs incrementation after each iteration of the loop.

Expression-1 and *expression-3* are optional, so initialization & incrementation isn't necessary in this statement.

Return

`return (` *expression*$_{opt}$ `);`

The return statement lets a function return to the function call. It returns the result of the evaluated *expression*, if provided.

Expressions

Precedence & Associativity

Operators have the following precedence:

1. `=`
2. `||, |, ^`
3. `&&, &`
4. `==, !=`
5. `< > <= >=`
6. `+, -, <<, >>`
7. `*, /`
8. `!, ~`
9. `::`
10. `car, cdr`

All operators are left-associative, except for assignment (`=`), not (`!`), and bin-not (`~`).

Primary Expressions

Primary expressions involve parenthesized expressions, function calls, and list initialization groups left to right.

- `(`*expression* `)` : a parenthesized expression
- `{` *expression* `}` : used to encapsulate functions, also used to indicate bin-type when printed
- `[` *expression* `]` : used to indicate a list type

Operators

It is the responsibility of the user to call the operators with the allowed types and values as arguments. If the user fails to abide by the operators' allowable types, the program will result in an error or thrown exception.

Unary Operators

Expressions with unary operators group right-to-left.

- `-` *expression*: evaluates an expression to the negative value of the same type. The type of the expression must be an `int` or Bin type.
- `!` *expression*: logical negation for booleans or conditionals

Multiplicative Operators

The multiplicative operations group left-to-right.

- *expression* `*` *expression*: the binary `*` operator indicates multiplication. Both operands must be of the same type, whether that be an `int` type or a Bin type.
- *expression* `/` *expression*: the binary `*` operator indicates multiplication. Both operands must be of the same type, whether that be an `int` type or a Bin type.

Additive Operators

The additive operators group left-to-right.

- *expression* `+` *expression*: the binary `+` operator indicates addition. Both operands must be of the same type, whether that be an `int` type or a Bin type. The result is the sum of the operands.
- *expression* `-` *expression*: the binary `-` operator indicates subtraction. Both operands must be of the same type, whether that be an `int` type or a Bin type. The result is the difference of the operands.

Relational Operators

The relational operators group left-to-right.

- *expression* < *expression*
- *expression* <= *expression*
- *expression* > *expression*
- *expression* >= *expression*

The relational operators are as follows: < (less than), <= (less than or equal to), and > (greater than), >= (greater than or equal to). The relational operators all return a boolean upon completion, returning `false` if the specified relation is false and `true` if the specified relation is true. Both operands must be of the same type, whether that be an `int` type, `char` type or a Bin type.

Equality Operators

- ➢ *expression* `==` *expression*: the `==` operator indicates “equal to” or equality. The operator will return `true` if the two expressions have equal value, and `false` otherwise. This operator can be applied to `int, bool, char,` Bin type and any expression that evaluates to one of these four types.
- ➢ *expression* `!=` *expression:* the `!=` operator indicates “not equal to” or inequality. The operator will return `true` if the two expressions *do not have* equal value, and `false` otherwise. This operator can be applied to `int, bool, char,` Bin type and any expression that evaluates to one of these four types.
- ➢ *expression* `&&` *expression:* the `&&` operator indicates “logical AND”. The operator will return `true` if the two operands are `true` and `false` otherwise. This operator can be applied to `int, bool, char,` Bin type and any expression that evaluates to one of these four types.
- ➢ *expression* `||` *expression:* the `||` operator indicates “logical OR”. The operator will return `true` if one of the two operands is `true` and `false` otherwise. This operator can be applied to `int, bool, char,` Bin type and any expression that evaluates to one of these four types.

Assignment Operators

The assignment operator groups right-to-left.

- ➢ *id* `=` *expression*: the value of the expression replaces that of the value referred to by the id. The left operand must be a variable and the right operand must be an expression of the same type. This operator can be applied to `int, bool, char,` Bin type and any expression that evaluates to one of these four types.

Bin Operators

All of the bin operators accept only a Bin type as an expression.

Bin-Shift Operators

- ➢ `<<` *expression*: the `<<` operator indicates a left shift. When shifting left, the most-significant bit is lost and a 0 bit is inserted on the other end. It returns a Bin type of the operand Bin type.
- ➢ `>>` *expression*: the `>>` operator indicates a right shift. When shifting right, the least-significant bit is lost and a 0 bit is inserted on the other end. It returns a Bin type of the operand Bin type.

Bin-Binary Operators

- ➢ *expression* `|` *expression*: the `|` operator indicates a bitwise OR. The bitwise OR operator takes two operands and returns a Bin type of the operand Bin type. The returned Bin type is the result of performing a bitwise OR on each bit at the same index in both expressions. At every index, the bitwise OR returns a 1 if either of the bits at the same index are 1. Otherwise it returns 0 at that index. Both expressions must be of the same Bin type.

- ➢ *expression* `&` *expression*: the `&` operator indicates a bitwise AND. The bitwise AND operator takes two operands and returns a Bin type of the operand Bin type. The returned Bin type is the result of performing a bitwise AND on each bit at the same index in both expressions. At every index, the bitwise AND returns a 1 if both of the bits at the same index are 1. Otherwise it returns 0 at that index. Both expressions must be of the same Bin type.
- ➢ *expression* `^` *expression*: the & operator indicates a bitwise XOR. The bitwise XOR operator takes two operands and returns a Bin type of the operand Bin type. The returned Bin type is the result of performing a bitwise XOR on each bit at the same index in both expressions. At every index, the bitwise XOR returns a 1 if exactly one of the bits at the same index is 1. Otherwise it returns 0 at that index. Both expressions must be of the same Bin type.
- ➢ `~` *expression*: the `~` operator indicates a bitwise NOT. The bitwise NOT operator takes one operand and returns a Bin type of the operand Bin type. The returned Bin type is the result of performing a bitwise NOT on the expression. The bitwise NOT inverts every bit in the expression. Every 0 becomes a 1 and every 1 becomes a 0.

Bin-Concat Operator:

- ➢ *expression* >< *expression*: the >< operator indicates a bitwise concatenation. The bitwise concatenation operator takes two operands and returns a Bin type. The two expressions can be different Bin types. The returned Bin type is the result of concatenating both expressions and the type returned is the Bin type that holds 2x as many bits as the biggest of the two Bin types given as expressions. It's the user's responsibility to properly type the variable that stored the result of the concatenation. If a word is one of the expressions provided, a word will be returned, and overflowing bits will be truncated.

List Operators

Lists in CHOMP are a built-in data structure. The types that can be stored in a list are as follows: `int`, `bool`, `char`, `list`, and any Bin type.

- ➢ *expression* `::` *list* : Takes an expression and a list. The `::` (pronounced "cons") operation returns a new cons cell containing the two arguments as a list. If the second type is not a list, and if the *expression* is not the same type as what's contained in the *list*, then the program will result in an error or thrown exception.
- ➢ `car` *list*: Takes a list. The `car` operation is used to split lists by returning the first item in the list. If the second type is not a list, then the program will result in an error or thrown exception.
- ➢ `cdr` *list*: Takes a list. The `cdr` operation is used to split lists by returning the rest of the list besides the first item in the list. If the second type is not a list, then the program will result in an error or thrown exception.

Print Operators

The following print operators serve to output expressions to standard output. It is important to note that any expression that the user wishes to output to standard output must be enclosed in parentheses before it is passed to the print operator.

- ➢ `Print (`*expression*`):` Takes an expression enclosed in parenthesis and prints it out to the terminal.
- ➢ `Println (`*expression*`):` Takes an expression enclosed in parenthesis and prints it out to the terminal followed by a newline.

Literals

`int`: any continuous string of digits

```
123
```

`char`: one ASCII character within single quotes around it.

```
'A'
```

`bool`: is either the word true or false

```
true
false
```

`null`: just the keyword `null`

```
null
```

`list`: a sequence of comma separated values, all of the same type, within brackets

```
[1, 2, 3]
['a', 'b', 'c']
```

Bin Literals

A bin literal is any string of 0s and 1s contained within `{{` and `}}`. If a bin literal has less digits then the maximum number of digits that can fit in that type, leading zeros will be added to the literal so that it has that maximum number of digits. If a bin literal is given an incorrect bin type that does not correspond to the length of the bin literal, this will result in a compiler error. If the user places any character that is not a 0 or 1 or is not the appropriate number of binary numbers, then the compiler will result in an error or exception.

`bit`: contains 1 digit.

```
{{0}}
```

`nibble`: contains 2 - 4 digits.

```
{{10}} /* would be stored as {{0010}}*/
{{0101}}
```

`byte`: contains 5 - 8 digits.

```
{{101011}} /* would be stored as {{00101011}}*/
{{01010011}}
```

`word`: contains 9 - 16 digits.

```
{{101011001}} /* would be stored as {{0000000101011001}}*/
```

```
{{0101001101010010}}
```

Variable Declarations

Declarations are used to specify the type which CHOMP gives to each identifier.
Any variable declaration that is declared outside of a function is a global declaration which means it declares a global variable. Any variable declaration that is declared inside of a function is a local variable, local to the function where it is declared.

Variable declarations have the forms below:

type-specifier identifier `=` *expression*;
type-specifier identifier;

The type-specifiers are:

`list` *type-specifier*
`int`
`bool`
`void`
`char`
`bit`
`nibble`
`byte`
`word`

Variable Assignment

Variable assignments are used to assign a value to an identifier.

Assignments have the forms below:

type-specifier identifier `=` *expression*;
identifier `=` *expression*;

The second form shown above can only be used if the identifier was previously declared with a type-specifier.

Functions

Function declarations have the form:

type-specifier identifier `(`*formals*$_{opt}$`)` `{`*statement*$_{opt}$`}`;

Formals refers to a comma separated sequence of 0 or more formals of the form:

type-specifier identifier

CHOMP supports higher order functions which means that the functions can take a function as an argument. The syntax for a formal that is a function is the same as for any other formal where the type-specifier is followed by the identifier associated with the function.

A function must return an expression with the type specified by the type-specifier, unless the return type is void. If the return type is void, there can be no return call or a return call followed by no expression. Example functions are shown below:

```
int add (int a, int b) {
     int c = a + b;
     return c;
}

void foo (char a) {
     print (a);
}

void faa (void foo) {
     foo('b');
     return;
}
```

Functions are called using the following form:

  identifier `();`

Program Structure

CHOMP expects files that are a sequence of variable and function declarations. These declarations & assignments will be completed sequentially. Once the program compiles successfully, the driver function named `main` will be run. A runtime error will be thrown if the program doesn't have a `main` function. Any declared variables or functions that's not used or called in `main` will be redundant.

Scope

Variables and functions must be declared before they are used. The scope of a variable is the location that is declared until the end of the block that it was declared in. If more than one variable shares the same name, the value used will be the one declared in the closest block it's used in.

Variables declared and/or assigned outside of functions are globally-scoped. Global variables will retain the value that is assigned to it in the global scope. It's accessible within a non-global scope, and its value is modifiable. However, once we exit that local scope (i.e. exit the block), its value reverts to its original global value.

Variables declared and/or assigned within functions are locally-scoped. They are accessible and modifiable within the function they're declared in, and once we exit that local scope (i.e. exit the block), it's no longer accessible unless declared once again.

Standard Library Functions

The following functions belong to the Bin Library, a collection of built-in functions that are
Note: remember that {{ }} indicates that the argument is a Bin Type. If the user passes an argument that is not of Bin Type, the program will result in an error or thrown exception.

Name	**Function Type**	**Function Contract**
`To{Bin Type}` `collection:` `ToBit(b)` `ToCrumb(b)` `ToNibble(b)` `ToByte(b)` `ToChar(b)` `ToWord(b)` `ToInt(b)`	`{{`Bin Type`}} -> {{`Bin Type`}}`	A collection of functions that convert a Bin Type *b* into a different Bin Type specified by the specific name of the `To{type}` function. Converting to a larger type fills in the additional space with zeros on the left. Converting to a smaller type truncates from the right (the least significant bits).
`set(v, i, b)`	`{{`Bin Type`}}, int, {{bit}} -> {{`Bin Type`}}`	Takes a Bin Type *v* and sets the `bit` at index of `int` *i* in *v* to `bit` value *b*, returns *v* with altered value. The value of *i* must be in the range: [0, |v| - 1], else the program will result in an error or thrown exception.
`flipBit(v, i)`	`{{`Bin Type`}}, int -> {{`Bin Type`}}`	Takes a Bin Type *v* and flips the `bit` at index of `int` *i*, returns *v* with altered value. The value of *i* must be in the range: [0, |v| - 1], else the program will result in an error or thrown exception.
`getBit(v, i)`	`{{`Bin Type`}}, int -> {{bit}}`	Takes a Bin Type *v* and returns the `bit` at index of `int` *i*.. The value of *i* must be in the range: [0, |v| - 1], else the program will result in an error or thrown exception.

Selected Examples

Examples of using the functions in the Bin Library are shown below:

To{Bin Type} Collection

```
byte b = {{10001011}}
nibble n = toNibble(b)
print(b)
/* Output: {{1000}}*/

byte c = {{10001011}}
word w = toWord(c)
print(c)
/* Output: {{0000000010001011}}*/
```

set(v, i, b)

```
byte b = {{10001011}}
set(b, 4, {{0}})
print(b)
/* Output: {{10000011}}*/
```

flipBit(v, i)

```
byte b = {{10001011}}
flipBit(b, 6)
print(b)
/* Output: {{10001001}}*/
```

getBit(v, i)

```
byte b = {{10001001}}
bit a = getBit(b, 4)
print(a)
/* Output: {{1}}*/
```

References

Ritchie, Dennis M. “C Reference Manual - Bell Labs.” *C Reference Manual*, www.bell-labs.com/usr/dmr/www/cman.pdf. Accessed 18 Oct. 2023.