Racecar Language

Language Reference Manual

Team 19

Samuel Kohn	Project Manager
Alexander Fields	Language Guru
Colfax Selby	Verification and Validation
Jeremy Spencer	System Architect
Mason Silber	System Integrator

Due: 27 March 2013
COMS W4115: Programming Languages and Translators
Professor: Alfred Aho
Mentor: Melanie Kambadur

Introduction

This manual contains the specification of the Racecar language. It is split into 9 sections: identifiers and scope, data types, statements, expressions, comments, control flow, built-in functions, user-defined functions, and the formal grammar. The notation used is as follows: text in `typewriter` represents code. *`Italics`* and `<bracketed>` text represent placeholders for code. The actual code fragments to replace the placeholders are specified either before or after the code block with the placeholder.

Identifiers and Scope

Identifiers are function or variable names. Syntactically, identifiers are limited to alphanumeric strings beginning with a letter, and they are case-sensitive. There is no limit to the length of an identifier, and all identifiers must be unique in their scope. Declaring an identifier which is already declared in the same scope is an error.

There are three kinds of scopes a variable can have in Racecar: global, function, and control-flow. The actual commands executed by Racecar are not contained in any function—they are "global." Any variables declared outside of all curly braces are said to have global scope. They can be accessed everywhere except for inside all function definitions. There is no main method required, but it is recommended that in all but the most trivial programs there is a single main-like function which drives the program. This function would be called in the global scope, e.g. on the first line of the program after any comments.

A variable declared inside a function definition, but outside of any other curly braces, has function scope. So does a variable listed as a function parameter. These variables can have the same names as variables declared outside the function, since variables with function scope are isolated from variables whose scope is outside of the function (i.e. global scope). Function scope extends into any curly braces inside the function definition.

Control-flow statements, namely repeat-if, repeat-times, and if-else, define the boundaries of control-flow scope. The scope of variables declared inside curly braces of a control-flow statement extends everywhere inside those curly braces. Additionally, variables whose scope extends directly up to a control-flow statement are also accessible inside that control-flow statement. Consequently, it is illegal to declare a variable inside a control-flow statement if a variable with the same name is accessible just outside that statement.

Identifiers representing functions have a scope which extends throughout the entire source file—into other functions and outside of all functions (i.e. global scope). Functions can be recursive, i.e. the scope of a function name identifier extends into the function definition itself. It is illegal to declare a function inside another function, so all functions must be declared as global functions.

Data Types

There are three data types in Racecar: `word`, `number`, and boolean. A literal word is an opening double-quote, followed by any string of characters that is not a double quote, followed by another (closing) double-quote. The empty word is notated as `""`. There is no way to have a double-quote as a character in a word. Words can be concatenated with the concatenation operator `++`, which is left-associative, and their equality can be tested with the operators `is` and `is not`. Numbers are positive or negative integers. They can be combined using the arithmetic binary operators `*`, `/` (integer division), `+`, and `-`, under the standard arithmetic operator precedences and left associativity. They can be compared using any of the comparison operators: `is`, `is not`, `>`, `<`, `>=`, and `<=`. Arithmetic operations are evaluated before comparisons. Numbers can be concatenated into strings, but not into other numbers. The left-associativity of the concatenation operator means that the following is still valid: `"hello" ++ 5 ++ 2 ++ 3`, and the result of the expression is the word `"hello523"`. To "cast" a number to a word, simply concatenate it with the empty word, as in `"" ++ 5`, which evaluates to the word `"5"`. There is no way to cast a word to a number. Boolean values are *not* allowed to be stored in variables, and hence the word "boolean" is not a reserved word (although its use as an identifier is strongly discouraged!). The boolean literals are `true` and `false`. Booleans are used internally to evaluate the result of comparisons and boolean operations (`and`, `or`, and `not`) in if-else and repeat-if statements.

Statements

All Racecar programs consist of zero or more statements. There are 10 types of statements: drive, steer, function definition, function invocation, repeat-if, repeat-times, if-else, print, declaration, and assignment. Drive and steer statements are covered in the build-in functions section, and repeat-if, repeat-times, and if-else are covered in the control flow section. Function definition and function invocation statements are specified in the user-defined functions section. The remaining statements are print, declaration, and assignment.

Print statements cause the specified text to appear in the graphical "console" in the Racecar application. The syntax for print statements is: `print` *`word_expression`* (word expressions are covered in the expressions section). This command causes the value of the word expression to appear in the console. Print statements are newline-terminated automatically.

All variables must be explicitly declared before they can be used. The syntax for declarations is: `IDENTIFIER is a` *`type`*, where type is either `number` or `word`. Before a variable has been assigned a value, using it (in an assignment or an expression) is an error.

Assignment statements store a value in a previously-declared variable. The syntax for variable assignments is: `set IDENTIFIER to` *`expression`*, where expression is either a numeric or word expression. Attempting to assign a word expression to a numeric identifier or vice versa will generate an error, as will assigning a boolean expression to any identifier or any expression to an undeclared identifier or to a function.

Statements end at the end of the line. There is no way to put two statements on a line, or to split a statement up over multiple lines.

Comments

There are both single-line and multi-line comments in Racecar. Single line comments begin with `:)` (smiley) and end at the end of the line. They do not need to begin at the beginning of the line. Multiline comments begin with a hyphenated frowny face `:-(` and end with a hyphenated smiley face `:-)`. They can be nested.

Expressions

An expression is a sequence of one or more identifiers, delimiters, and operators. Expressions can be word expressions, numeric expressions, or boolean expressions. Word expressions only contain string literals, identifiers of type word, and concatenation operators. Numeric expressions only contain arithmetic expressions, namely integer constants, identifiers of type number, the `+`, `-`, `*`, and `/` operators, and parentheses. Boolean expressions contain the constants `true` and `false`, comparisons of numeric and word expressions to similarly-typed expressions, and the logical `and`, `or`, and `not` operators. Valid comparison operators are `is`, `is not`, `>`, `<`, `>=`, and `<=`. There is no == or `!=` (use `is` and `is not` instead). Words can only be compared using `is` and `is not`. Delimiters are whitespace and parentheses.

Control Flow

There are two types of loops in Racecar: repeat-if (similar to while in other programming languages) and repeat-times (similar to for). The repeat-if statement looks like the following:

```
repeat if <boolean expression>
{
    <code block>
}
```

This loop checks the value of the boolean expression and executes the code block if it evaluates to `true`. It then checks the value of the boolean expression again and executes code block. This process repeats until the boolean expression evaluates to `false`. Any boolean expression can be used in these loops.

The second type of loop is the repeat-times loop. It looks like the following:

```
repeat <number expression> times
{
    <code block>
}
```

This loop will execute the code in the code block repeatedly the specified number of times. You can specify this number by any numerical expression.

Built-in Functions

There are 5 built in functions: `drive`, `steer`, `canMove`, `getWheelDirection`, and `getLocation`. The first two are complete statements when they are invoked correctly (similar to all user-defined functions), but the latter three "return" values and hence are expressions (unlike user-defined functions). The direction arguments to the first three functions are not strings, but rather language keywords. The drive function is defined by the following:

```
drive <direction> <number expression> <optional step(s)>
```

`drive` tells the program to move the car `<number expression>` steps in the `<direction>` direction. The possible values for `<direction>` are: `forward`, `forwards`, `backward`, and `backwards` (all of these are reserved words). The user can either type `step` or `steps`, or omit the term entirely. Racecar does not check the English grammar of using `step` for only moving 1 step and `steps` for 0 steps or 2 or more steps.

The `steer` function is the following:

```
steer <direction>
```

The `steer` function turns the front wheels of the car 45 degrees in the direction specified. The possible values for `<direction>` in this function are: `left`, `right`, and `straight`. The `<direction>` is the new desired state of the wheels, not the action that the wheels will take. Consequently, if the `steer` function is called with `<direction>` equal to the current direction of the wheels, then the wheels are not turned. Also, if the wheels are already turned to the right, then calling `steer left` will make the wheels turn all the way to the left (as opposed to moving 45 degrees to the left of their current position, which would put them pointing straight ahead).

The `canMove` function is the following:

```
canMove <direction>
```

The `canMove` function returns a boolean (`true` or `false`) representing whether the car can move one step in the given direction without hitting into an obstacle or the edge of the map. If `<direction>` is a possible argument to the `drive` command, then the result will reflect whether the car would crash if the following code were immediately executed:

```
drive <direction> 1 step
```

Note that the current direction of the wheels is taken into account. If `<direction>` is a possible argument to the `steer` command, then the result will reflect whether the car will hit an obstacle if the following code were immediately executed:

```
steer <direction>
drive forwards 1 step
```

In this case, the current direction of the wheels is irrelevant (but is not altered by this function call), and the it is assumed that the car will try to move forwards after turning its wheels. In order to check if the car would hit anything by moving backwards with the wheels pointed in the given direction, it is necessary to explicitly steer the wheels and then test using `canMove backwards`.

The `getWheelDirection` function takes no arguments and returns a `word` indicating the direction in which the wheels are oriented. The possible return values are: `"left"`, `"right"`, or `"straight"`.

The `getLocation` function takes no arguments and returns a `number` corresponding to the unique position of the car on the map. The exact representation of the car's coordinates are implementation-specific and subject to change, and hence should not be used other than for equality comparison with some previously stored location.

User-defined Functions

Users can define their own functions in Racecar. Function definition follows the signature:

```
define <identifier> <optional parameter list>
{
    <code block>
}
```

The function name must be a valid identifier and cannot conflict with any variable names that have global scope. The `<optional parameter list>` is defined by the following:

```
using <identifier> (type) and ... and <identifier> (type)
```

The keyword `using` is required if there is at least one parameter, and the keyword `and` is required whenever there are two or more parameters, between the type of one parameter and the name of the next. Parameter names cannot conflict with function names (even other functions). Note that the parameter list is in fact optional—it is omitted completely if there are no parameters to the function. Functions cannot return values.

User-defined functions are invoked using the same syntax as built-in functions, but unlike some built-in functions, user-defined functions are always complete statements on their own, and never expressions:

```
function_name parameter_1 ... parameter_n
```

To use a non-trivial expression as a parameter, surround it with parentheses. For example, `myFunction (2 + 3) "hello" ("John" ++ " Doe")` is a function call where `myFunction` is called with three parameters.

Formal Grammar

The grammar productions are formatted as follows: nonterminals are in `lowercase`; terminals are in `UPPERCASE`; literals are in `'quotes'` and can only be one character long, as defined by PLY. The empty string is represented by `epsilon`.
The start symbol for all Racecar programs is `statements`. Comments are treated as whitespace by the lexer and hence are not covered by the grammar.

```
statements
    : statements statement
    | epsilon

statement_block
    : '{' statements '}'

statement
    : statement_contents NEWLINE
    | NEWLINE

statement_contents
    : drive_command
    | steer_command
    | define_command
    | repeat_if_command
    | repeat_times_command
    | if_command
    | print_command
    | declaration_command
    | assignment_command
    | function_command

expression
    : expression OR and_expression
    | and_expression

and_expression
    : and_expression AND not_expression
    | not_expression

not_expression
    : NOT not_expression
```

```
    | TRUE
    | FALSE
    | CAN_MOVE can_move_direction
    | comparison

can_move_direction
    : drive_direction
    | steer_direction

comparison
    : comparison bool_operator plus_expression
    | plus_expression

bool_operator
    : IS | IS_NOT | GT | LT | GEQ | LEQ

plus_expression
    : plus_expression '+' times_expression
    | plus_expression '-' times_expression
    | times_expression

times_expression
    : times_expression '*' word_expression
    | times_expression '/' word_expression
    | word_expression

word_expression
    : word_expression CONCAT primary_expression
    | primary_expression

primary_expression
    : WORD
    | NUMBER
    | ID
    | GET_CAR_POSITION
    | GET_WHEEL_DIRECTION
    | '(' expression ')'

function_command
    : function_command primary_expression
    | primary_expression
```

```
drive_command
    : DRIVE drive_direction plus_expression optional_steps

drive_direction
    : FORWARD
    | FORWARDS
    | BACKWARD
    | BACKWARDS


optional_steps
    : STEP
    | STEPS
    | epsilon

steer_command
    : STEER steer_direction

steer_direction
    : LEFT
    | RIGHT
    | STRAIGHT

define_command
    : DEFINE ID optional_param_list NEWLINE statement_block

optional_param_list
    : epsilon
    | USING ID '(' type ')' optional_extra_params

optional_extra_params
    : epsilon
    | AND ID '(' type ') optional_extra_params

type
    : NUMBER_TYPE
    | WORD_TYPE

repeat_if_command
    : REPEAT IF expression NEWLINE statement_block

repeat_times_command
```

```
    : REPEAT plus_expression TIMES NEWLINE statement_block

if_command
    : IF expression NEWLINE statement_block optional_else_if
optional_else

optional_else_if
    : optional_else_if NEWLINE ELSE IF expression NEWLINE
statement_block
    | epsilon

optional_else
    : NEWLINE ELSE NEWLINE statement_block
    | epsilon

print_command
    : PRINT word_expression

declaration_command
    : ID IS_A type

assignment_command
    : SET ID TO expression
```