University of Southampton

Programming Language Concepts

April 2017

Bark of the Hound Language

Authors:
Fraser Crossman *(fc4g15)*
Anish Katariya *(ak7n14)*

Contents

1 Introduction

The Bark of the Hound Language is a domain specific programming language capable of manipulating sets through the use of the Insert and SetMinus operators. Using this language were able to take finite languages as input and manipulate them to create new languages. The language also includes features found in general purpose programming languages such as printing, looping, conditionals, mathematical operators, and boolean operators that can be used to solve a wide variety of problems. All inputs are assumed to be a sequence of finite sets followed by a positive integer, which represents the maximum size of the output set.

```
 {a, bc, de}
3
```

2 Syntax

2.1 Basics

A well structured program written in the language starts with a begin token and ends with an end token. Comments in the language are written between two backslash star sequences (/* comment */)

2.2 Variable Types

The language has support for three primitive types; Integer, String and Boolean in addition to the Set type. Each variable declaration is preceded by the keyword let and followed by a semicolon.

```
let int  w;      /* Integer named w */      let str  x;      /* String named x */
let bool y;      /* Boolean named y */      let set  z;      /* Set named z */
```

2.3 Operations

2.3.1 Integers

The language has support for the following mathematical operators: addition (+), subtraction (-), division (/), multiplication (*), and modulus (%). All mathematical operators are left associative. Brackets can be used to override their precedence.

```
let int  x = 4;      int  x = int  x + 1 * 5;      /* x = 9 */
```

2.3.2 Strings

For string manipulation the language has support for string concatenation (^) operator.

```
let str  x = "Julian" ;      let str  y = "Rathke" ;
let str  z = str  x ^ " "  ^ str  y;      /* z = "Julian Rathke" */
```

2.3.3 Booleans

The language has support for boolean operators such as equals (==), not equal (! =), less than (<), greater than (>), less than or equal to (<=), greater than or equal to (>=) , and (&&) , or (||), and not (!)

```
let bool  a = (3 > 4);      /* a = false*/
```

2.3.4 Sets

The language has support for both set minus (SetMinus) and set insert (Insert) operations. Sets also maintain alphabetical order when stored and printed to standard output.

```
 let set  x;
set  x = set  x Insert "A" ;       /* x = {A} */
set  x = set  x Insert "C" ;       /* x = {A, C} */
set  x = set  x Insert "B" ;       /* x = {A, B, C} */
```

2.3.5 Print

The language provides the ability to print **any** data type to standard output using the print statement.

3 Control Flow

The language has support for conditional if else statements and for loops which may be nested.

3.1 If- Then - Else

If then else statements use the following syntax:
if <condition> then <statement> fi
or
if <condition> then <statement> else <statement> fi

```
 begin
      if (3 < 4) then
            print "GREEN";
      else
            print "RED";
      fi       /* "GREEN" is printed to standard output */
end
```

3.2 For Loop

The language has support for regular for loop and enhanced for loops. The for loops can be described written as either:
for <condition> do <statement> rof
or
for <string> in <set> do <statement> rof

```
 begin
      let set  s1us2;
      let str  temp;
      for str  inp1 in args0  do
            str  temp = "a"  ^ str  inp1;
            set  s1us2 = set  s1us2 Insert str  temp;
      rof
end
```

4 Additional features

4.1 Programmer Convenience

This language provides many programmer conveniences to aid the writing of simple programs. As previously noted, all sets store their strings alphabetically which is advantageous when printing only a subset of a language stored in a set variable. Variables are also initialised on declaration to prevent the need to initialise a variable which will later be reassigned.

Type	*Initialised Value*
Integer	0
String	""
Boolean	false
Set	{:}

Table 1: Depicts data types and automatically initialised values

In order to ensure that the output of a program does not exceed the value specified by the trailing integer in input the binding OUTPUT_COUNT is added so that it can be referenced and used in programs. This is especially useful when working with infinite sets.

```
begin
      let set  aStar;       let str  aString;       let int  count;
      for int  count < int  OUTPUT_COUNT  do
            int  count = int  count + 1;
            set  aStar = set  aStar Insert str  aString;
            str  aString = str  aString ^ "a" ;
      rof
      print set  aStar;       /* a* is printed - up to the length specified */
end
```

For the purposes of readability, it is also possible to use a predefined constant to represent the empty sting named __empty_string .

4.2 Type Checking

The symbols + , - , / , and * must be used with integer values only. Similarly, the string concatenation operator ^ must be used with only strings. The boolean operators > , >= , < , and <= must also be used with integer values only however, || , && , and ! must only be used with Boolean type expressions.
The == and ! = operators may be used by integer, string, or Boolean values. Finally the operators Insert and SetMinus must only be used with a set as the first argument and a string as the second. This is because, in this language, sets only contain strings. It is convenient for the programmer that these rules are enforced by the language interpreter as it means that unintentional problems with a program can be found early in development.

4.3 Error Messages

The error messages displayed are informative and indicate to the programmer the type of error that has occurred. This proves to be very useful when debugging user written programs.

4.3.1 Syntax Errors

If any of the aforementioned type checks fail or a general syntax error is found then the following error is displayed: `Fatal error: exception Parsing.Parse_error`

4.3.2 Invalid Variable References

If a variable is not present in the current environment, an error specific to the specified type is displayed. The general form of this error is:
`Fatal error: exception Failure("Variable [type] [var name] Not Declared as a [type name] type. Hint: Maybe try - let [type] [var name] = [default value];")`
An example of this type of error where x is an integer referenced, but not present, in the current environment is shown below:
`Fatal error: exception Failure("Variable int x Not Declared as an integer type. Hint: Maybe try - let int x = 0;")`
If the token denoting the type of a variable is missing the following error is displayed: `Fatal error: exception Failure("lexing: empty token")`
If a variable is reused in the subject of a for loop then an error is thrown. `Variable [var name] is already in use. Try changing variable name.`
Our language allows reference to arguments. If an undefined argument is specified then the following error is displayed: `Invalid Input [arg ref].Enter valid args#.`

4.3.3 Division by Zero Error

If a division by 0 is performed this error is displayed: `Fatal division by 0 error.`

4.3.4 Print Error

If a set, for whatever reason, fails to print correctly when using the print keyword then the following error is displayed: `Set undefined. Ensure the set is correctly defined.`