Dara Hazeghi
dmh2186
11/6/11
COMS 4115

Project Proposal

strlang - A simple string processing language

strlang is an imperative language designed specifically with text processing in mind. It is built around a string data type with built-in operators for manipulating text. Strlang includes standard imperative features such as variables, procedures, conditionals and loops. It is intended to be to be both simple and expressive, with a reduced set of built-in operators.

Motivation

Textual data is omnipresent. A vast number of programming tasks boil down to processing this data in some fashion, from simple searches to complex transformations across formats. Traditional compiled programming languages like C and Java don't make this particularly easy, either requiring special libraries or special extensions to accomplish such tasks. As a result, much text processing is done with scripting languages like Perl and Python, which while easier to program, tend to be much slower and not to provide the extensive error-checking of compiled languages.

Strlang is intended to combine the best of both these approaches. Its sparse syntax and built-in string operations will allow the programs to be written easily and quickly with little delving into special libraries. As a compiled language though, it features all of the advantages of static type-checking and will output an intermediate format that can be easily optimized by standard tools.

Description

Strlang supports two data primitives: strings and numbers. Strings support standard operations including concatenation, searching, replacement, splitting into multiple substrings and comparison. Regular expressions can be used in search/replace operations. Numbers support standard arithmetic expressions.

To aid with processing these, strlang also includes two collection types: lists and maps. Lists allow the storage and retrieval of an arbitrary number of (homogeneous) primitive data elements. Maps store associations between data keys and data values.

Strlang includes constructs for procedures, looping, branching and expressions. It also includes a small set of built-in procedures to handle input/output operations and to access system functions.

The basic structure is not dissimilar from C. A program consists of variable declarations and procedures. A procedure is a signature followed by a code block. A code block is a set of variable declarations, statements, where each statement can be an expression, a language construct, or another code block.

The strlang compiler's job is to perform basic semantic checking, and output Java code that can perform the correct computation. In particular, the compiler needs to verify that all variables and procedures referenced exist in the relevant scope, and that the types of the various operands in each expression is compatible.

Possible uses for strlang might include performing mass edits on text documents, compiling statistics on word frequencies, reformatting text or translating between different document and format types or extracting specific columns or entries from large files. The

Example

Suppose one wanted to get a count of the number of citations in a text file. Assuming the citation format were of the form ($Name_{opt}$ pagenum), the following program would print out the number of strings of that form found.

```
\start() // procedure definition
{
	$citation;	// define a string var
	$line;
	#count;		// define a num var

	citation <- "\(([A-Z][a-zA-Z]* )?[0-9]+\)"; // assigning a string

	[!open("read"; "input.txt")] { -> 1; }
	// read is a procedure call, [condition] { code } is a conditional
	// and -> expr; is a return statement

	line <- read();			// open() means read() is from input.txt
	<<line != ?>>			// loop - while(line != NULL)
	{
		count <- count + ^(line * citation);
		// a * b returns list of string in a matching b
		// ^a returns the number of elements in list a

		line <- read();
	}

	write("Number of citations found: " + tostr(count));
	-> 0;
}
```

Language Definition

```
program:
	variable_list_opt procedure_list

variable_list:
	variable
	| variable variable_list
```

```
variable:
       type Id;

type:
       // num, string, list(num | string) or map (num->num | num->string or...)
       # | $ | @[#] | @[$] | %[#;#] | %[#;$] | %[$;#] | %[$;$]

procedure_list
       procedure
       | procedure procedure_list

procedure:
       \Id (variable_list_opt) type_opt block

block:
       { variable_list_opt statement_list_opt }

statement_list:
       statement
       | statement statement_list

statement:
       block
       | expr;
       | -> expr_opt;                      // return
       | [expression] block                // if
       | [expression] block [!] block      // if else
       | << expression >> block            // while

expr:
       // numeric operands can be used with all these operators
       // definitions provided for string operands (where applicable)

       expr + expr          // concatenate two strings
       | expr - expr        // remove the first or last op2 characters from op1
       | expr * expr        // list of matches, where op2 is regexp string
       | expr / expr        // search/replace strings of op2 (regexp) in op1
       | expr % expr        //list of strings in op1, split based on (regexp) op2

       | expr < expr        // lexicographic string comparisons
       | expr <= expr
       | expr > expr
       | expr >= expr
       | expr == expr
       | expr != expr

       | expr & expr        // logical and - only for numbers
       | expr | expr        // logical or - only for numbers

       | - expr
       | ! expr

       | ^ expr             // length of list/map/string/num
       | (expr)

       | Id(expr_list_opt)  // call of procedure with parameters

       | lvalue <- expr     // assignment, same types, or null for map/list
       | lvalue

       | StringLit
       | NumLit
       | ?                  // null
```

```
expr_list:
       expr
       | expr expr_list

lvalue:
       Id
       | Id[expr]          // access list/map value
```

Builtins

`open($;$)` - Set the input or output stream to the specified file or keyboard/screen.
`$ <- read($)` - Read and return one line from the input stream.
`write($)` - Write the specified string to the output stream.
`$ <- tostr(#)` - Convert a number to a string.
`# <- tonum($)` - Convert a string to a number.
`@$ <- keystrs(%($;?)` - Get a list of all the string keys in a map.
`@# <- keynums(%(#;?)` - Get a list of all the number keys in a map.

Miscellaneous

Initialization: all variables are initialized to empty (strings, lists or maps) or 0 (numbers) when they are defined.

Scope: all variables have block scope. They can be accessed only from blocks inside and including the current block.