strlang

Dara Hazeghi
12/22/11
COMS 4115 – Fall 2011

Introduction to strlang

- Simple static imperative language for text processing
  - Sparse, minimalist syntax
  - C-like structure
- Allow programmer to easily and efficiently manipulate strings
  - Strongly-typed to catch errors at compile-time
  - Produce code that can be optimized and executed quickly

Features

- String as a primary data type
  - Full set of operators for building, searching and transforming strings
  - Maps for associating key-value pairs
- Procedural structure
  - Functions, blocks, loops, conditionals
  - All computation performed in expressions
- Generates linearized (low-level) C++ code as output
  - Simplified expressions, no blocks, no loops

Language Tutorial

- Variables and types
  - Declaration: type name;
    - String (text) - $ - `$ str;`
    - Number (integral) - # - `# num;`
    - Map (aggregate) – %[k;v] - `%[$;#] map;`
- Expressions
  - Literals
    - String: `"str_literal"`
    - Number: `12345`
  - Assignment
    - `name <- expression`
  - Unary and binary operators
    - `expr + expr` or `expr % expr` or `^expr` or …
    - See table
  - Function calls
    - `name(expr1; expr2; expr3…)`
  - Rvalues (variables)
    - Name
  - Example: `a <- b <- 3 + 5 / 4 | 3;`

Operator	Associativity	Notes
`<-`	Right to Left	Assignment. Requires identical type operands (no implicit conversion).
`|`	Left to Right	Logical or `|`. No short-circuit evaluation.
`&`	Left to Right	Logical and `&`. No short-circuit evaluation.
`== !=`	Left to Right	Structural equality `==` and inequality `!=`.
`< > <= >=`	Left to Right	Numeric comparison for numbers, lexicographic comparison for strings.
`+ -`	Left to Right	Addition `+` and subtraction `-` for numbers, concatenation `+` and substring `-` for strings, deletion `-` for maps.
`* / %`	Left to Right	Multiplication `*`, division `/` and modulus `%` for numbers, match `/` and index `%` for strings.
`~~`	Left to Right	Replacement for strings (ternary operator).
`- ! ^`	Left to Right	Arithmetic `-` and logical negation `!` for numbers, length `^` for strings and maps.
`[]`	Left to Right	Accessor for maps.
`@% @@`	Right to Left	Keys `@%` or values `@@` for maps.

Language Tutorial

- Functions
  - name, list of parameters, return type, block (containing function's code)
    - `name(type1 name1; type2 name2 … -> typeret { code block }`
  - No return value, or no parameters (void): `^`
  - Parameters passed by reference
  - Program control starts in (required) main function
    - `main(^)  -> # { code block }`
- Blocks
  - List of variable declarations, followed by list of statements
    - `{ decl1 decl2 … stmt1 stmt2 … }`
  - Variables declared in block only valid in that block (scope rules)
- Statements
  - Expressions – see above
    - `expression;`
  - Blocks – same syntax as above
  - Conditionals – test expression must be numeric, second clause optional
    - `[ expr ] blockif-true ![ ] blockif-false`
    - `[ expr ] blockif-true`
  - Loops – test expression must be numeric
    - `< expr > block`
  - Return – expression may be empty
    - `-> exproptional;`

Example – source code

```
// hello.str – comment

main(^) –> #                          // main take no input, returns a number
{
    $name;                            // string variable

    write("Enter your name:\t");      // write string to the output stream
    name <– read();                   // read string from the input stream, store in variable

    print_banner("Hello " +
          name + "!"; 10);            // call print_banner function with 2 parameters

    –> 0;                             // return the value ‘0’ to the calling environment
}

print_banner($ msg; # max) –> ^       // print_banner takes string and number, returns nothing
{
    #i;                               // number variable

    i <– 0;                           // assignment: i set to ‘0’
    <i < max>                         // loop: while(i < max)
    {                                 // begin loop block
          write(msg + "\n");          // + concatenates two strings, which are then written out
          msg <– " " + msg;
          i <– i + 1;                 // + adds two numbers
    }                                 // end loop block

    <i > 0>                           // another loop
    {
          write(msg + "\n");
          msg <– msg – 1;
          i <– i – 1;
    }
    write(msg + "\n");
}
```

Example – compiled code

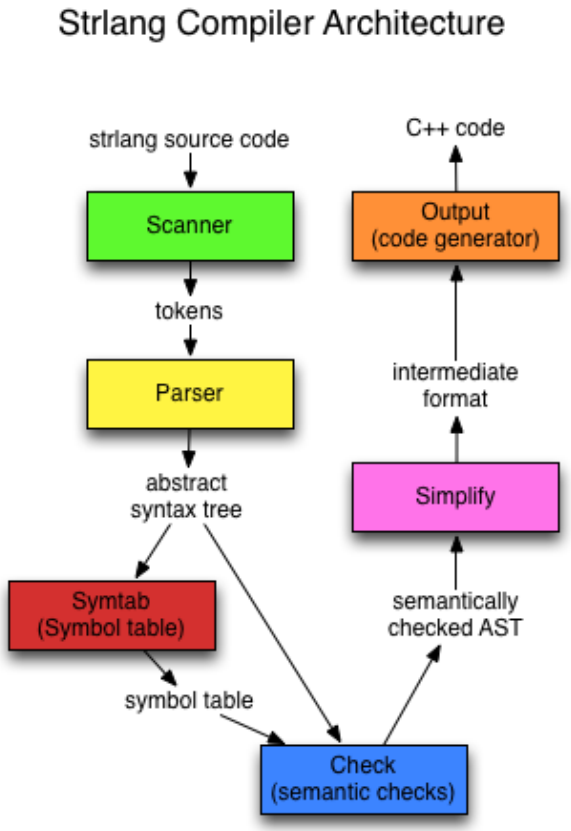
```
$ ./strlang –c hello.str

#include "strlib.h"

int main(void);

void print_banner(string&, int&);


int main(void)
{
  string name_1("");
  string __reg_str_25_("");
  string __reg_str_24_("");
  int __reg_num_23_(0);
  string __reg_str_22_("");
  string __reg_str_21_("");
  string __reg_str_20_("");
  string __reg_str_19_("");
  int __reg_num_18_(0);
  int __reg_num_26_(0);
  __reg_str_25_ = "Enter your name:\t";
  write(__reg_str_25_);
  __reg_str_24_ = read();
  name_1 = __reg_str_24_;
  __reg_num_23_ = 10;
  __reg_str_21_ = "!";
  __reg_str_19_ = "Hello ";
  __reg_str_20_ = __str_concat(__reg_str_19_, name_1);
  __reg_str_22_ = __str_concat(__reg_str_20_, __reg_str_21_);
  print_banner(__reg_str_22_, __reg_num_23_);
  __reg_num_18_ = 0;
  return __reg_num_18_;
  return __reg_num_26_;
}

void print_banner(string& msg_4, int& max_4)
{
  int i_4(0);
  int __reg_num_17_(0);
  string __reg_str_16_("");
  string __reg_str_15_("");
  string __reg_str_14_("");
  string __reg_str_13_("");
  int __reg_num_12_(0);
  int __reg_num_11_(0);
  int __reg_num_10_(0);
  string __reg_str_9_("");
  string __reg_str_8_("");
  string __reg_str_7_("");
  int __reg_num_6_(0);
  int __reg_num_5_(0);
  int __reg_num_4_(0);
  int __reg_num_3_(0);
  int __reg_num_2_(0);
  string __reg_str_1_("");
  string __reg_str_0_("");
  __reg_num_17_ = 0;
  i_4 = __reg_num_17_;
  goto __LABEL_3;
__LABEL_2: ;
  __reg_str_15_ = "\n";
  __reg_str_16_ = __str_concat(msg_4, __reg_str_15_);
  write(__reg_str_16_);
  __reg_str_13_ = " ";
  __reg_str_14_ = __str_concat(__reg_str_13_, msg_4);
  msg_4 = __reg_str_14_;
  __reg_num_11_ = 1;
  __reg_num_12_ = i_4 + __reg_num_11_;
  i_4 = __reg_num_12_;
__LABEL_3: ;
  __reg_num_10_ = i_4 < max_4;
  if(__reg_num_10_) goto __LABEL_2;
  goto __LABEL_1;
__LABEL_0: ;
  __reg_str_8_ = "\n";
  __reg_str_9_ = __str_concat(msg_4, __reg_str_8_);
  write(__reg_str_9_);
  __reg_num_6_ = 1;
  __reg_str_7_ = __str_substr(msg_4, __reg_num_6_);
  msg_4 = __reg_str_7_;
  __reg_num_4_ = 1;
  __reg_num_5_ = i_4 - __reg_num_4_;
  i_4 = __reg_num_5_;
__LABEL_1: ;
  __reg_num_2_ = 0;
  __reg_num_3_ = i_4 > __reg_num_2_;
  if(__reg_num_3_) goto __LABEL_0;
  __reg_str_0_ = "\n";
  __reg_str_1_ = __str_concat(msg_4, __reg_str_0_);
  write(__reg_str_1_);
  return;
}
```

Example – running compiled code

```
aiguille:strlang dara$ ./strlang -e hello.str hello
aiguille:strlang dara$ ./hello
Enter your name:       strlang
Hello strlang!
 Hello strlang!
  Hello strlang!
   Hello strlang!
    Hello strlang!
     Hello strlang!
      Hello strlang!
       Hello strlang!
        Hello strlang!
         Hello strlang!
          Hello strlang!
           Hello strlang!
          Hello strlang!
         Hello strlang!
        Hello strlang!
       Hello strlang!
      Hello strlang!
     Hello strlang!
    Hello strlang!
   Hello strlang!
  Hello strlang!
 Hello strlang!
Hello strlang!
aiguille:strlang dara$
```

Design

- 6 step compilation process
  - scanner – split source input into stream of tokens
  - parser – parse tokens to generate abstract syntax tree
  - symtab – build symbol table for all identifiers in the AST
  - check – validate AST and annotate it with type information
  - simple – simplify AST by converting expressions to SSA-like form, flattening blocks and replacing loops with gotos
  - output – dump simple IR as C++ code (pretty-printer)
- Final step – C++ compiler generates executable from code output by strlang compiler



Conclusion

- Major goals
  - 0) Gain experience in language design
  - 1) Come up with a coherent design
  - 2) Implement it cleanly and correctly
  - 3) Make the language/compiler useful
  - 4) Complete deliverables by deadline
- Success?
  - strlang design is reasonably clear, comprehensible
  - Compiler meets the design spec, finished by deadline
  - Code is generally clean
  - Testsuite passes, no major known defects
  - But… not quite as useful as hoped for
    - Missing split operator for strings
    - Syntax can be restrictive

Lessons Learned

- Working as 1-person group has pluses and minuses
  - + having control of design allows focus
    - Able to emphasize simplicity and feasability in design
    - No issues with integration, coding could be done rapidly and efficiently
  - - could have used some feedback in coding phase
    - Easy to get tunnel vision, miss important design considerations
    - Not infrequently thinking, “there must be a better way to do this”
- Overall, did benefit from earlier group participation
  - Design phase was simplified - had already gone over many of the major issues
- Planning is key – deadlines, well-defined milestones, building the testsuite as you go
- Writing a compiler is fun – everybody should do it at least once!

The End

- So long and thanks for all the strings!