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
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;

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;-</td>
<td>Right to Left</td>
<td>Assignment. Requires identical type operands (no implicit conversion).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Logical or</td>
</tr>
<tr>
<td></td>
<td>Left to Right</td>
<td>Logical or</td>
</tr>
<tr>
<td>== ! =</td>
<td>Left to Right</td>
<td>Structural equality == and inequality !=.</td>
</tr>
<tr>
<td>&lt; &gt; &lt;= &gt;=</td>
<td>Left to Right</td>
<td>Numeric comparison for numbers, lexicographic comparison for strings.</td>
</tr>
<tr>
<td>+ -</td>
<td>Left to Right</td>
<td>Addition + and subtraction – for numbers, concatenation + and substring – for strings, deletion – for maps.</td>
</tr>
<tr>
<td>* / %</td>
<td>Left to Right</td>
<td>Multiplication *, division / and modulus % for numbers, match / and index % for strings.</td>
</tr>
<tr>
<td>~ ~</td>
<td>Left to Right</td>
<td>Replacement for strings (ternary operator).</td>
</tr>
<tr>
<td>- ! ^</td>
<td>Left to Right</td>
<td>Arithmetic – and logical negation ! for numbers, length ^ for strings and maps.</td>
</tr>
<tr>
<td>[]</td>
<td>Left to Right</td>
<td>Accessor for maps.</td>
</tr>
<tr>
<td>@% @@</td>
<td>Right to Left</td>
<td>Keys @ or values @@ for maps.</td>
</tr>
</tbody>
</table>
Language Tutorial

- **Functions**
  - name, list of parameters, return type, block (containing function's code)
    - `name(type_1 name_1; type_2 name_2 ... -> type_ret
      { 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
    - `{ decl_1 decl_2 ... stmt_1 stmt_2 ... }
  - 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 ] block_{if-true} ![ ] block_{if-false}
    - `[ expr ] block_{if-true}
  - Loops – test expression must be numeric
    - `< expr > block`
  - Return – expression may be empty
    - `-> expr_{opt};`
Example – source code

// hello.str - comment
main(^) -> #
{
    $name;
    write("Enter your name:\t");
    name <- read();
    print_banner("Hello "+name + "!"; 10);
    -> 0;
}
print_banner($ msg; # max) -> ^
{
    #i;
    i <- 0;
    <i < max>
    {
        write(msg + "\n");
        msg <- " " + msg;
        i <- i + 1;
    }
    <i > 0
    {
        write(msg + "\n");
        msg <- msg - 1;
        i <- i - 1;
    }
    write(msg + "\n");
int main(void)
{
  string name_1(""), __reg_str_25_(""), __reg_str_24_(""), __reg_num_23_(0), __reg_str_22_(""), __reg_str_21_(""), __reg_str_20_(""), __reg_str_19_(""), __reg_num_18_(0), __reg_str_26_(""), __LABEL_0: ;
  __reg_num_16_ = __str_concat(msg_4, __reg_str_15_);
  write(__reg_str_16_);
  __reg_str_17_ = __str_substr(msg_4, __reg_num_15_);
  msg_4 = __reg_str_17_;
  __LABEL_1: ;
  __REG_4_ = i_4 - __REG_2_;
  i_4 = __REG_4_;
  __LABEL_0: ;
  __REG_4_ = __REG_1_ + __REG_4_;
  __LABEL_1: ;
  __REG_3_ = i_4 > __REG_2_;
  if(__REG_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!
aiguille:strlang dara$
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
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 feasibility 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!
So long and thanks for all the strings!