Strlang

String Manipulation Language

Dara Hazeghi
22 December, 2011
COMS 4115

Table of Contents

Chapter 1

Introduction to strlang

strlang is a simple imperative programming language, designed specifically to make manipulating text strings easy. The language features a minimalistic syntax and includes support for a variety of string-oriented operations including matching and substituting regular expressions. While taking syntactic cues from Perl and other more traditional text processing languages, strlang is a compiled language and offers native performance and full static semantic checking.

1.1 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 quickly and easily. As a compiled language though, it features all of the advantages of static type-checking and outputs C++ code as which can readily be built and optimized by standard tools.

1.2 Overview

strlang is an imperative language with a C-like structure. It incorporates variables, expressions, blocks, conditionals, loops and functions. The native data types for strlang are strings and numbers. String operations include concatenation, substring, matching and replacing. Matching and replacing allows the use of powerful Perl-style regular expressions. Numbers allow for standard arithmetic operations. Additionally, the map data type provides a container to hold key-value pairs of basic data types (strings or numbers).

strlang is a strongly typed language. The compiler provides type-checking and other error-detection, and performs lowering of expressions and blocks. It outputs a particular subset of C++ code that is similar to three-address code.

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

Chapter 2

Tutorial

The strlang language features a C-like structure wrapped in a minimalistic syntax. Its focus is on making manipulation of text strings easy and efficient.

2.1 Basics

strlang has three types of variables: strings (`$`), numbers (`#`) and maps (`%[k;v]`). Strings are used to store text, number for integer values, and maps to hold key-value pairs. Keys and values in a map may be strings or numbers.

Comments

Comments are notes by the programmer in the code. They are ignored by the compiler. Comments begin with `//` and run through the end of the line. For readability, comments will be shown in red, code in green.

```
... // this is a comment and will be ignored
```

Declarations

Variables must be declared before they are used. Declarations specify the variable's type.

```
$ str1;             // declaring a string called 'str'
# num1;             // declaring a number called 'num'
%[$;#] city_temps;  // declaring a map with string keys and number values
```

Assignment

The most common operation on variables is to assign them the value of some expression using the assignment operator (`<-`). The value of the expression on the right side is stored in the variable on the left side. The variable and expression must have the same type (i.e. both string or number or...)

```
str2 <- str1;       // value of the variable str1 stored in str2
num2 <- num1;       // value of num1 stored in num2
num2 <- str1;       // ERROR: this won't work (different variable types)
```

Literals

Strings and numbers that are known may be used directly in place of variables.

```
num1 <- 0;          // num1 assigned the value 0
str1 <- "hello";    // str1 assigned the value "hello"
```

Operators

Variables and constants can be transformed using unary and binary operators. Each operator has rules regarding the types of its inputs and outputs. Here are some of the basic operators:

```
num1 <- num1 + 1 * 3 - 7 / 5;     // standard arithmetic operations
str1 <- str1 + " world";          // concatenating two strings
str1 <- "hello world" - 5;        // removing the last 5 characters
str1 <- "hello" / "l";            // search one string for the other
num1 <- city_temps[str1];         // get value associated with key str1
city_temps["Ojai"] <- 44;         // associate value 44 with key 'Ojai'
```

Expressions

An expression is simply some operation that can be executed and evaluated. Literals, variable names and binary and unary operations are expressions, among others.

```
3*7                 // arithmetic expression
"hello " + str1     // string expression
num1                // arithmetic expression
```

Functions

Functions are the logical programming unit for strlang. Each function is a semi-autonomous chunk of code which receives certain input at the beginning from whomever invokes it, and when it finishes, it may choose to send back certain output.

The function header defines the name the function is invoked by, as well as the types and numbers of the inputs to the function, and how they will referred to inside the function. The header also defines the type of value the function will output. The function's body contains the actual expressions and statements executed when the function is invoked.

Syntax

```
name(type input1; type input2; ... ) -> output_type // header
{
        // body
}
```

Functions that do not take input from their caller, or do not return output use `^` in place of those parts of the header.

```
empty_func(^) -> ^ // no input, no output
...
no_input_func(^) -> # // no input, number output
...
```

To get values out of a function, a return statement is used. The statement consists of the return operator (`->`) followed by an expression (or no expression if the function is not defined as outputting a value).

```
add(#a; #b) -> #    // add takes in two numbers, outputs a number
{
      -> a + b;     // output sum of num1 and num2 to the caller
}
```

Invoking Functions

Defining functions isn't much use if they can't invoked. Invoking a function requires specifying the values or variables that will be given as input. These values are mapped to the input variables defined in the function header in the order given. For functions that output a value, the result can be assigned using the standard assignment operator.

```
num1 <- add(num1; num2);   // function 'add' receives num1 and num2
                           // as input and its output is saves in num
                           // in this case, 'a' gets num1, 'b' gets num2
empty_func();              // a function with no inputs
```

Every program must define a function named 'main' which takes no inputs and outputs a number. This function is where program execution begins.

Built-in functions

A few functions are built in to the language to make input/output and other lower-level processes easier. The most important are:

```
read(^) -> $        // read in a line of text (by default from
                    // the keyboard) and return it as a string.
write($msg) -> ^    // output the string 'msg' (by default to the screen)
```

2.2 Example

Hello World code

```
main(^) -> #        // function header
{                   // mark beginning of function body
      $ name;       // variable declaration

      write("Enter your name: "); // call to function, with parameter
      name <- read(); // call to function, assigning result to 'name'

      write("Hello " + name + "!\n");  // call to another function
      // expression is evaluated before function is called

      -> 0;  // leaving the function, giving back the value '0'
}            // end of function body
```

Running the program

The compiler is invoked using the strlang executable. The compiler expects the name of the source file and the name to save the executable output as. Saving the above file to hello.str, the process is:

```
$ ./strlang
usage:
./strlang
        -a file.str                 (dump AST of source)
        -c file.str [file.cpp]      (compile to C++)
        -e file.str file.exe        (compile to executable)
        -h                          (display this message)
        -v                          (display version number)

$ ./strlang -e hello.str hello
$ ./hello
Enter your name: Dara H
Hello Dara H!
$
```

2.3 Other language features

Conditionals

strlang includes the standard if-else construct. The test expression, which must be of type number, is evaluated first. If the result is non-zero, the block directly after the expression is executed. If not, the block associated with the else-clause is executed. The else clause is optional.

```
[num1 < 0]                          // test clause: num1 < 0 is test expr
{ write("num1 is negative"); }      // if num1 < 0, this executes
![]                                 // otherwise (else)...
{ write("num1 is nonnegative"); }// this block executes
```

Loops

strlang's loops behave like while loops in C. The expression in the loop header is evaluated. If the result is non-zero, the loop block is executed. If not, control jumps to the first statement after the loop block. When the last statement of the loop block executes, the program returns to the loop header.

```
<num1>                      // loop header: num1 (!= 0) is test expr
{
       write(to_string(num1) +    // print message...
              "bottles of beer on the wall.\n");
       num1 <- num1 - 1;          // decrement loop counter
}      // end of loop
```

Operators

strlang has many operators for manipulating strings, and several for manipulating maps. See the Language Reference Manual for further details.

2.4 Invoking the Compiler

The strlang has 3 modes: one for generating code, one for generating an executable and one for displaying a program's internal representation.

In compilation mode, the compiler will either display or save to a file the C++ code it has generated for the given program.

```
$ ./strlang -c hello.str hello.cpp
```

In executable mode, the compiler will first generate C++ code and then invoke the C++ compiler to build an executable that can be run directly.

```
$ ./strlang -e hello.str hello
```

In display mode, the compiler will dump to the screen the abstract syntax tree representation of the program after parsing it.

```
$ ./strlang -a hello.str
```

2.5 Installation

strlang assumes a UNIX-like environment. It requires make, bash, diff and several other standard UNIX utilities to build and execute. It also requires a C++ compiler, including the C++ standard library (STL).

1). Obtain and install the ocaml 3.1 distribution, hosted at http://caml.inria.fr/. The ocaml binaries must be added to the $PATH environment variable.

2). Obtain and install the pcre regular expression library, located at http://www.pcre.org/. If building from source, make sure to enable the C++ wrapper: (./configure --enable-cpp). The libraries and headers need to be installed in directories in the standard search path (/usr/local/include and /usr/local/lib are usually fine).

3). Obtain and decompress the strlang source distribution.

4). Adjust strlang.ml's cpp_compiler variable if you are using a C++ compiler other than g++. If PCRE is not installed in the default search path, you may need to change the pcre_* variables as well.

5). From the top-level of the source distribution, type 'make'.

Chapter 3

Language Reference Manual

strlang is a simple imperative programming language, designed specifically to make manipulating text strings easy. The language features a minimalistic syntax and includes support for a variety of string-oriented operations including matching and substituting regular expressions. While taking syntactic cues from Perl and other more traditional text processing languages, strlang is a compiled language and offers native performance and full static semantic checking.

3.1 Lexical conventions

strlang has five types of tokens: names, number constants, string constants, operators, and punctuation. The language does not have any keywords.

Names are any sequence of one or more alphabetic (upper and lower-case), numeric or _ characters. The first character must be alphabetic. Names are case sensitive. A small number of names are reserved for use as built-in functions and may not be used for other purposes: `open`, `read`, `end_input`, `write`, `to_num`, `to_string` and `exit`.

A number constant is any sequence of one or more integer characters. As a practical matter, integer constants are limited to 2^31.

A string constant is a sequence of characters surrounded by double quotes. C-style escape characters are used.

Operators and punctuation include: `( ) { } [ ] + - * / % | & < > <= >= == != ! ^ ~ @@ @% <- -> $ # %` and `;`.

Whitespace, meaning spaces, tabs, carriage-returns and newlines, is ignored.

Comments are begun with the characters // and run to the end of the line. Comments are ignored.

3.2 Types

strlang has two fundamental types of variables: strings and numbers. Strings contain text, whereas numbers contain integer quantities. In addition to the basic types, the language includes the map type for associating key-value pairs. strlang is a strongly typed language, and all conversions between different types must be made explicitly.

Strings `$`

Text strings are the bread and butter of strlang. Strings are sequences of ASCII character values. Regular expressions are no different than other strings, and are interpreted only in the context of searching and replacement. They use the extended Perl syntax. String operators include concatenation, substring creation, searching, matching and replacing.

Numbers `#`

Number variables are included to allow for integer and boolean arithmetic. Numbers are signed 32-bit integer quantities. They support the five standard integer arithmetic operators, as well as comparison, and boolean connectors. In a boolean context, the value zero is treated as false, and all other values are treated as true.

Maps `%[t1 ; t2]`

Maps are the only aggregate type in strlang. Maps are sets of key-value pairs. Only a single value may be associated with each key. When a map variable is defined, the types of its keys and values must also be defined. Keys and values may be strings or numbers, but must be homogeneous. Accordingly, there are four types of maps: number-to-number, number-to-string, string-to-number and string-to-string.

Map operations include insertion of a key-value pair, lookup of a value by key, deletion of a key-value pair by key and two special operators used to extract all the values or keys in a map to be used as values in a new map with number keys. Note that maps with differing key or value types are considered to be of different types.

None `^`

Certain expressions have no value associated with them. Their type is consequently considered to be 'none'. Variables of this type cannot be explicitly created.

Type	Name	Notes
`$`	String	Text strings.
`#`	Number	Signed integer quantities.
`%[k;v]`	Map (k-to-v)	Set of key value pairs. k and v can be either `$` or `#`. The map has keys of type k and values of type v.
`^`	None	Used to indicate functions with no parameters, or no return value.

3.3 Expressions

Operator Precedence and Associativity

Precedence and associativity of the various operators in strlang is given below, ordered from lowest to highest precedence.

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.

3.3.1 String Expressions

String Constants `"[^\"]*"`

Returns the string containing the text between the double-quote symbols.

```
a <- "str const"; // string variable 'a' now contains 'str const'
```

Concatenation `$ '+' $`

Returns the concatenation of the two string operands.

```
a <- "hello " + "world"; // 'a' now contains 'hello world'
```

Substring `$ '-' #`

Returns a substring of the first operand. If the second operand is non-negative, these are the characters of the first operand starting at position n (n being the second operand). Otherwise, these are the characters of the first operand except for the last |n|.

```
a <- "hello world" - 6; // 'a' now contains 'world'
a <- "hello world" - -6; // 'a' now contains 'hello'
```

Match `$ '/' $`

Return the first substring in the first operand that matches the second operand. If there is no match, an empty string is returned. The second operand will be interpreted as a regular expression, so special symbols will need to be escaped if they are to be interpreted as literals.

```
a <- "hello world." / "world"; // 'a' now contains 'world'
a <- "hello world." / "h[e-l]*o"; // 'a' now contains 'hello'
a <- "hello world." / "."; // 'a' now contains 'h'
a <- "hello world." / "\\."; // 'a' now contains '.'
```

Index `$ '%' $`

Return the position in the first operand of the first match for the second operand. If there is no match, the value -1 is returned. As with match, the second operand will be interpreted as a regular expression.

```
i <- "hello world" % "world"; // 'i' now contains '6'
i <- "hello world" % "h[e-l]*o"; // 'i' now contains '0'
i <- "hello world" % "world."; // 'i' now contains '-1'
```

Replace `$ '~' $ '~' $`

Return a string consisting of the first operand, with all occurrences of the second operand replaced by the third operand. The second operand will be interpreted as a regular expression.

```
a <- "hello world" ~ "[eo]" ~ "x"; // 'a' now contains 'hxllx wxrld'
a <- "hello world" ~ "world" ~ ""; // 'a' now contains 'hello '
```

Length `'^' $`

Return the length of the operand - the number of ASCII characters in the string.

```
i <- ^ "hello world"; // 'i' now contains '11'
```

Comparison `$ '<' $` or `$ '>' $` or `$ '<=' $` or `$ '>' $`

Return 1 if the first operand is lexicographically less than, greater than, less than/equal or greater than/equal to the second operand. Otherwise return 0.

```
i <- "hello" < "helloo"; // 'i' now contains '1'
i <- "hello" > "helloo"; // 'i' now contains '0'
i <- "hello" >= "hello"; // 'i' now contains '1'
```

3.3.2 Number Expressions

Number Constants `[0-9]+`

Returns a number containing the integer value in the expression.

```
i <- 7; // 'i' now contains '7'
```

Addition `# '+' #`

Returns the sum of the two operands.

```
    i <- 7 + 4; // 'i' now contains '10'
```

Subtraction `# '-' #`

Returns the difference of the two numbers.

```
    i <- 7 - 3; // 'i' now contains '4'
```

Multiplication `# '*' #`

Returns the product of the two numbers.

```
    i <- 7 * 3; // 'i' now contains '21'
```

Division `# '/' #`

Returns the whole-number quotient of the two numbers. If the value of the second operand is 0, a runtime error will occur.

```
    i <- 7 / 3; // 'i' now contains '2'
    i <- 7 / 0; // runtime error
```

Modulus `# '%' #`

Returns the remainder of the two numbers. If the value of the second operand is 0, a runtime error will occur.

```
    i <- 7 % 3; // 'i' now contains '1'
    i <- 7 % 0; // runtime error
```

Boolean Connectors `# '|' #` or `# '&' #`

Returns the logical disjunction or conjunction of the two operands. Evaluation is not short-circuited, meaning both operands are always evaluated.

```
    i <- 0 | 1; // 'i' now contains '1'
    i <- 1 & 0; // 'i' now contains '0'
    i <- 0 & (0 / 0); // runtime error, even though 0 is first operand
```

Comparison `# '<' #` or `# '>' #` or `# '<=' #` or `# '>=' #`

3.3.3 Map Expressions

Accessor `% '[' $ ']'` or `% '[' # ']'`

Return the value for the key given as the second operand, in the map given as the first operand. The type of the second operand must match the type of the map's keys. If the key is not found, zero or the empty string are returned, depending upon whether the map's values are numbers or strings.

```
s <- ms["hi"]; // 's' now contains the value associated with 'hi' in 'ms'
i <- mn[3]; // 'i' now contains the value associated with '3' in 'mn'
```

Deletion `% '-' $` or `% '-' #`

Delete the key-value pair associated with the key given as the second operand and the map given as the first operand. Return that value.

```
s <- ms - "hi";
// 's' has same value as above example, but 'ms' no longer contains
// the given pair
```

Emptying `% '<-' '0'`

Empty the map given as operand of all key-value pairs, and return the same empty map.

```
empty <- ms <- 0; // 'ms' and 'empty' are now both maps with no contents
```

Length `'^' %`

Return the number of key-value pairs in the map.

```
i <- ^m; // 'i' contains the number of keys in m
```

Keys `'@%' %`

Return a new map containing the keys of the map given as the operand. The new map's values are the keys of the old map. The new map's keys are numbers, zero through the size of the map.

```
keys <- @% m; // 'keys' contains the keys of 'm' as its values
```

Values `'@@' %`

Return a new map containing the values of the map given as the operand. The new map's values are the values of the old map. The new map's keys are numbers, zero through the size of the map.

```
vals <- %% m; // 'vals' contains the values of 'm' as its values
```

3.3.4 General Expressions

Lvalues `name`

Lvalues are simply variables. They can both be read from and assigned to. They return the value stored in the given variable. As there are no implicit type conversions, the type of the return value is simply the same as the type of the variable.

Equality/Inequality `expr '==' expr` or `expr '!=' expr`

Return the number 1 if the two expressions are structurally equal, and 0 otherwise. Strings, numbers and maps may be compared, but both operands must be of the same type.

```
        i <- 1 == 1; // 'i' contains 1
        i <- "s" == "S"; // 'i' contains 0
```

Assignment `lvalue '<-' expr`

Returns the value of the second operand, and also stores that value into the first operand. Accordingly, the types of the first and second operands must be the same.

```
        j <- i <- 1 == 1; // 'i' and 'j' contains 1, if both i and j are # vars
```

Function Calls `name '(' expr1 ';' expr2 ';' ... ')'`

Function calls evaluated as follows: first all of the arguments (the expressions within parenthesis) are evaluated, left-to-right. Then the function code is called and executed. The return value is precisely the value returned by the function. In the case that the function's return type is none, no value is returned.

Unlike variables, functions need not necessarily be declared prior to use, but the name must have a matching declaration somewhere in the program. Moreover, the number and types of the expressions given as arguments to the function must match with the number and types of the parameters given in that function's declaration.

Parameters are passed by reference.

3.4 Statements

Statements comprise the bulk of a strlang program. Statements are executed in sequence. In addition to simple expressions, statements include loop and conditional control structures, logical code blocks, and return statements. Each type of statement is described in detail below.

Variable Declarations `type name ';'`

All variables must be declared prior to use. The declaration associates a type to a given name. The type may be any of the types given above, save for `^` (none). Unlike other types of statements, variable declarations may only appear at the beginning of a block, or at the beginning of a program before any other code. Declarations not within any block are considered global. Declarations within a block are considered local to that block.

All variables are initialized automatically. Number variables are set to zero by default, strings to the empty string, and maps to a map containing no keys.

Expressions `expr ';'`

Expression statements are evaluated according to the rules described in the previous section.

Return **`'->'`** **expr**$_{opt}$ **`';'`**

Return statements are used to stop execution of the current function and return control to the calling function, at the place where the current function was called. The associated expression is evaluated, and its value is given back to the caller. The type of the expression must match the return type of the function. Functions with return type of 'none' should omit the expression portion of the statement.

Blocks **`'{'`** **decl_list**$_{opt}$ **stmt_list**$_{opt}$ **`'}'`**

Blocks are logical units of code. Blocks consist of grouping markers (curly braces) around an optional list of variable declarations followed by an optional list of statements. Variables declared within a block are not valid outside that block. Two variables of the same name may not be declared in the same scope. Within a block, references to a name are bound to the variable declared in the nearest enclosing scope.

Conditionals `'[' expr ']' block` or `'[' expr ']' block '![]' block`

In a conditional statement, the expression is first evaluated and if the result is non-zero, the block immediately after is executed. The expression must be of type number. If the result of evaluating the expression is zero and the conditional uses the second form, the second block is executed.

Loops `'<' expr '>' block`

In a loop statement, the expression is first evaluated. If the result is non-zero, the following block is executed and then control moves back to the top of the loop, where the process repeats. The expression for a loop must be of type number.

3.5 Functions

Functions `name '(' decl_list ')' '->' type block` or
`name '(' '^' ')' '->' type block`

Functions in strlang consist of a signature and a block. The signature is the name, declaration list, and type. The declaration list indicates the names and types of the input parameters for the function (or `^` for none). The type indicates the kind of value that will be returned to the caller. Finally, the block contains the actual code of the function.

Function names must be unique within a program. Variables also may not use the same names as functions. The scope of a function is the entire program.

Variables declared in the input parameter section have the same scope as variables declared at the top of the function's code block.

A functions with a return type other than `^` and lacking an explicit return statement will automatically return zero, the empty string, or an empty map, in accordance with the return type given in its signature.

3.6 Program Structure

Program **decl_list**$_{opt}$ **func_list**

A strlang program consists of an optional list of variable declarations, followed by a list functions. Variables declared at the program level have program-wide scope.

Functions need not be declared in any particular order in a strlang program. However, every strlang program is required to contain a function by the name of main with the following signature:

```
main(^) -> ^
```

Execution of a strlang program begins with the first statement in the code block of this main function. The program is then executed statement-by-statement, as described above.

3.7 Built-in Functions

strlang provides a few built-in functions to facilitate input and output and conversion between different types.

Signature	Description

`open($ io_type;`
`     $ filename) -> ^`
	open() opens an input or output stream for subsequent reads or writes.
	
	If io_type matches "in", it will attempt to open filename for reading and use that file for subsequent read() calls. If filename is "stdin", it will use standard input.
	
	If io_type match "out", it will attempt to open filename for writing and use it for subsequent write() calls. If filename is "stdout", it will use standard output.

`read(^) -> $`	read() gets the next line of input from the current input

	stream, and returns it as a string.
`end_input(^) -> #`	end_input returns 1 if the input stream has reached the end of input, and 0 otherwise.
`write($ outstr) -> ^`	write() prints the string outstr to the current output stream.
`to_num($ str) -> #`	to_num() converts a string to a number, which it returns.
`to_string(# num) -> $`	to_string() converts a number to string, which it returns.
`exit(^) -> #`	exit() terminates the current program, returning the given number value to the calling environment.

3.8 Syntax Summary

```
program:
        decl_list func_list func

decl_list:
        /* empty */
        decl_list decl ;

func_list:
        /* empty */
        func_list func

func:
        name ( formals_list_opt ) -> ret_type block



decl:
        type name

type:
        $              /* string */
        #              /* number */
        %[$; $]        /* map from string to string */
        %[$; #]        /* map from string to number */
        %[#; $]        /* map from number to string */
        %[#; #]        /* map from number to number */

formals_list_opt:
        ^              /* void - empty argument list */
        formals_list

formals_list:
        decl
        decl ; formals_list

ret_type:
        ^              /* void - no return value */
        type

block:
        { decl_list stmt_list }

stmt_list:
        /* empty */
        stmt stmt_list

stmt:
        block                      /* code block */
        expr ;                     /* single expression */
        < expr > block             /* while(expr) block */
        [ expr ] block             /* if(expr) block
        [ expr ] block ![] block   /* if(expr) block else block */
        -> ;                       /* return; */
        -> expr;                   /* return expr; */

expr:
        expr binop expr            /* binary operation */
        unop expr                  /* unary operation */
        expr ~ expr ~ expr         /* search/replace */
        lvalue <- expr             /* assignment */
        ( expr )                   /* grouping */
        name ( actuals_list_opt )  /* function call */
        lvalue                     /* variable */
        number_literal             /* number literal: [0-9]+ */
        string_literal             /* string literal: "[^\"]*" */

lvalue:
        name                       /* variable */
        name [ expr ]              /* map accessor variable */

binop:        /* any of the following */
        + - * / % == != < > <= >= | &

unop:         /* any of the following */
        - ! ^ @@ @%
```

Chapter 4

Project Plan

4.1 Team Responsibilities

The strlang project will be a one-person project. Due to the late start-date, the emphasis will be on quickly settling on a feasible set of features to implement, carefully defining the basic infrastructure so that a skeleton compiler can be rapidly written and then filling out the individual features one at a time. To verify functionality, I will be adding tests as I add features with the intent of maximizing coverage.

4.2 Style Guide

To minimize time spent rereading code and debugging, I will be trying to adhere to the following style recommendations:

1) Modular design. The compiler will be split logically into different passes, with each one in its own file. Each piece should be relatively self-contained.
2) Self-documenting code. In particular, function names should be chosen for descriptiveness, and arguments whose types are not obvious will be annotated explicitly with their types
3) Design for the general. Avoid special cases and duplication, even at the cost of efficiency.
4) Make use of the library. Before implementing anything generic, verify in the OCaml library documentation that there isn't a built-in function that will do the job.
5) Avoid external dependencies. The C++ code generated by the strlang compiler should be as generic as possible, and avoid relying on specific compiler and language features where possible.

4.3 Project Timeline

This following timeline indicates the various milestones for this project, and the dates by which they should be finished.

11/7	Project begins
11/9	Project proposal - defined primary language features
11/14	Language Reference Manual, Parser and AST
11/21	Static Semantic analysis and symbol table
11/28	Simplification pass and basic code-generation
12/5	Testsuite and second-tier feature support
12/12	Sample applications and Design documentation

4.4 Development Environment

The strlang project will be built in a UNIX-compatible environment. The project's build infrastructure will include Makefiles to automate the build process and bourne shell scripts to run and verify testcases. The compiler will be written in the OCaml language, relying on the ocamlc, ocamllex and ocamlyacc tools from the OCaml 3.13 distribution available from inria.fr. Additionally, to convert the C++ code produced by the strlang compiler to executable code, the g++ 4 compiler, including STL library is required, and for regular expression support, the pcrecpp library must also be installed.

4.5 Project Development Log

These are the dates of major project developments.

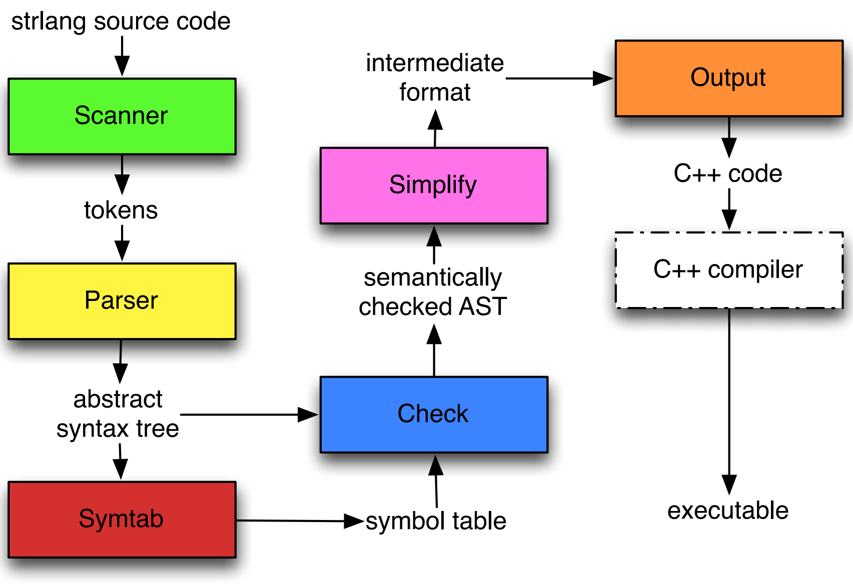
11/7	Project begun
11/9	Language whitepaper completed
11/11	Parser completed, no conflicts
11/13	AST generation and dumping completed
11/14	Symbol table completed, rough semantic checking implemented
11/16	Basic simplification pass and code generation completed, first testcase compiles
11/18	Reworked semantic checking pass to generate clean intermediate code
11/19	Reworked code-generation pass
11/20	Testsuite for semantic checks completed
11/22	Runtime testsuite covers basic features
11/27	Reworked simplification pass to further simplify intermediate format
11/30	Runtime testsuite for advanced features completed
12/4	Language manual brought up to date with final feature list
12/12	Primary example programs completed and verified
12/14	Code cleanup
12/17	Report writeup
12/20	Project presentation slides
12/21	Clean up compiler driver
12/22	Final testcases and examples

Chapter 6

Architectural Design

The strlang compiler takes as input a single strlang program in source form, and outputs that same program as C++ source code after having checked it for errors. The strlang compiler consists of essentially 6 passes. These are the scanner, parser, symbol table generator, static semantic checker, simplifier, and output passes. Each pass transforms or annotates its input in preparation for the following pass.

Strlang Compiler Architecture



6.1 Scanner

The scanner receives the original source code and transforms it into a stream of tokens. It breaks up the input stream using an ocamllex generated automata.

5.2 Parser

strlang uses a bottom-up yacc-generated parser. The parser receives tokens from the scanner and generates an abstract syntax tree from those tokens. Syntactic errors are detected by the parser. The current implementation does not make any attempt at error recovery or diagnosis for parse errors.

5.3 Symbol Table

To avoid the need for forward declarations, strlang relies a two-pass process to check the AST generated by the parser. The symtab pass builds a table of all the identifiers (names) used in the source, their type, and the scope in which they were declared.

5.4 Static Semantic Checker

The 'check' pass essentially performs two related goals - it verifies that the operands in all expressions are compatible, and creates a new abstract syntax tree in which the expressions also include type information.

For unary and binary operators, the process of checking the types of the operands is complicated by the fact that in strlang most operators are overloaded to handle more than one type of operand. For example, the '+' sign means addition if the operands are both of type number, but concatenation if the operands are both of type string. All operators on non-number types are converted into function calls to stubs which are defined by in the strlang library. This means that `"a" + "b"` is converted to `__str_concat(a; b);` .

For variables and function calls, all names are checked via the symbol table, so the names in the AST (basically just strings) are replaced by the variable or function signatures, as defined in that table. For function calls, the number and type of the arguments are all verified against the function's signature. For assignment, the types of the rvalue and lvalue are compared.

In addition to expressions, the checking pass verifies certain attributes of statements. Return statements are checked against the signature of the enclosing function, to make sure the right type of value is returned. The expressions in loop headers and conditionals are also verified to be of number type.

Finally, this pass confirms that the program does in fact have a main function defined.

5.5 Simplification

The 'simple' pass turns the checked AST into a simplified intermediate representation. This simplification flattens functions to a single scope/block, unwinds complex expressions and converts loops and conditional to jumps and labels.

The difference between simplified and complex expressions is that in simplified expressions, operands must be names, they cannot be other expressions. As a result, the compiler is responsible for creating a number of temporary storage units, to hold intermediate results in complex expressions. Although this seems quite inefficient, it is actually analogous to what many real compilers do, prior to optimization.

Removing conditionals and loops is simply a matter of putting labels and jumps in the appropriate spots. For instance, with a loop, the test is done at the end of the body, and if it succeeds, you must jump to the top of the loop, just before the first statement.

Once the conditionals and loops are gone, the other blocks can be safely flattened. The only caveat is that if a variable in an inner block has the same name as a variable in an outer block, there could be a conflict. This is addressed by including the scope id as part of the variable's original data, allowing the two to be differentiated. However, because declarations must all occur prior to regular statements, all declarations must then be moved to the beginning of the function.

Simplification Examples

Before	After

`a <- a + 3 * 4;`	`tmp1 <- 3;`
	`tmp2 <- 4;`
	`tmp3 <- tmp1 * tmp2;`
	`a <- a + tmp3;`

`f(3; 4; c);`	`tmp1 <- 3;`
	`tmp2 <- 4;`
	`f(tmp1; tmp2; c);`

`< a != 2 > { stmt_list }`	`goto endloop;`
	`startloop:`
	`stmt_list`
	`endloop:`
	`tmp1 <- 2;`
	`tmp2 <- a != tmp1;`
	`cond_goto tmp2 startloop;`

`[a != 2] { stmt_list } ![] {`
`stmt_list2 }`
	`tmp1 <- 2;`
	`tmp2 <- a != tmp1;`
	`tmp3 <- !tmp2;`
	`cond_goto tmp3 else;`
	`stmt_list`
	`goto endif;`
	`else:`
	`stmt_list2`
	`endif:`

`f(^) -> ^`
`{`
  `#a;`
  `{`
    `#a;`
    `a <- 2;`
  `}`
  `a <- 1;`
`}`
	`f(^) -> ^`
	`{`
	  `#a_scope1;`
	  `#a_scope2;`
	  `#tmp1;`
	  `#tmp2;`
	
	  `tmp1 <- 2;`
	  `a_scope2 <- tmp1;`
	
	  `tmp2 <- 1;`
	  `a_scope1 <- tmp2;`
	`}`

5.6 Output

The 'output' pass is responsible for turning the intermediate format into C++ code. The intermediate format is designed to map well onto a subset of C++, so this pass functions basically like a pretty printer.

5.7 Dependencies

The C++ code that is generated by strlang is reasonably self-contained and largely avoids using features beyond those present in traditional C. It does however rely on the presence of the strlib.h header file and the STL and PCRE libraries.

The STL library is used primarily for the <string> and <map> classes which are used to implement the string and map types in the strlang language. These could be replaced readily enough with simpler self-contained replacements, but that seemed beyond the scope of the original project. The PCRE library is used to provide regular expression processing for certain string operations.

The compiler relies a number of built-in OCaml libraries. The driver in particular uses the 'Unix' OCaml module in order to directly exec the C++ compiler when requested.

The strlib.h header file was created in conjunction with the compiler and contains the wrappers used for the various string and map operations that strlang supports. This code could simply be emitted in-line in the output pass, obviating the need for a separate header file. In practice though, the separate header solution proved significantly cleaner, since emitting non-trivial blocks of literal C++ code via OCaml is actually fairly ugly and inconvenient. A few of the wrappers in that header use templates to avoid having to rewrite the same wrapper four times using different input parameter types. To avoid messing around with pointers, pass-by-reference is done using C++ reference variables. Other than that, no C++-specific features are used.

Chapter 6

Test Plan

Because thorough testing is vitally important for a sophisticated piece of software like a compiler, the test-suite was largely built concurrently with the compiler. The goal was to maximize code coverage and feature coverage, ensuring that as many of the compiler's features were tested as possible.

The test-suite is sub-divided into two categories: tests that cover the semantic verifier and tests that cover code generation. Most tests are be unit tests, but a few larger programs are tested for code-generation too.

For the semantic verifier, the goal was to ensure that no invalid program was accepted. Accordingly, for each type of error that the verifier explicitly handled, I built a testcase. These testcases are thus designed to fail in the compilation phase.

For the code-generation part of the equation, I built a variety of small programs, each intended to cover one particular type of code that the compiler must be able to produce correctly. These are both compile and runtime tests in that they are expected to compile, and to run and produce a particular output.

The test-suite was run regularly, via script. For the semantic tests, the error messages output are compared against the expected errors for that input. In the code-generation case, the output of running the program is compared against known-good output from running the same program.

6.1 The Testsuite

The test-suite for strlang consists of 39 tests.

```
$ ./test.sh
tests/check/bad-assign1.str      // invalid assignments
tests/check/bad-assign2.str
tests/check/bad-assign3.str
tests/check/bad-call1.str        // invalid function calls
tests/check/bad-call2.str
tests/check/bad-call3.str
tests/check/bad-if.str           // invalid conditional
tests/check/bad-main.str         // main given bad arguments
tests/check/bad-operands-add.str // invalid operands for operators
tests/check/bad-operands-cmp.str
tests/check/bad-operands-div.str
tests/check/bad-operands-len.str
tests/check/bad-operands-mod.str
tests/check/bad-operands-mul.str
tests/check/bad-operands-mul2.str
tests/check/bad-operands-neg.str
tests/check/bad-operands-neg2.str
tests/check/bad-operands-sub.str
tests/check/bad-ret1.str         // invalid return statement
```

```
tests/check/bad-var.str           // invalid variable declaration
tests/check/bad-while.str         // invalid loops
tests/check/empty.str             // missing main function
tests/check/redeclared1.str       // redeclaring the same identifier
tests/check/redeclared2.str
tests/check/undeclared-var1.str   // using an undeclared identifier
tests/check/undeclared-var2.str
tests/run/arithmetic1.str         // functionality tests
tests/run/arithmetic2.str         // arithmetic operators
tests/run/builtin.str             // built-in functions
tests/run/fib.str                 // compute the fibonacci sequence
tests/run/hello-long.str          // extended 'hello world'
tests/run/hello.str               // simple hello world
tests/run/loop1.str               // testing that loops behave
tests/run/map1.str                // testing map operations
tests/run/modinv.str              // compute modular inverses and gcds
tests/run/ret.str                 // test different sort of return stmts
tests/run/sort.str                // sort entries from a file
tests/run/stringfun.str           // string banner fun
tests/run/strings1.str            // test string operators
SUCCEEDED
$
```

6.2 Example: fibonacci

This simple program computes the 17th number in the Fibonacci sequence.

```
// fib.str
fib(#i) -> #
{
	[(i == 1) | (i == 2)] {-> 1;}
	![] { -> (fib(i - 1) + fib(i - 2)); }
}

main(^) -> #
{
	write("fib(17) = " + to_string(fib(17)) + "\n");
}
```

The compiler produces the following C++ code from this source code.

```
// fib.cpp
 #include "strlib.h"

int fib(int&);

int main(void);


int fib(int& i_3)
{
  int __reg_num_12_(0);
  int __reg_num_9_(0);
  int __reg_num_8_(0);
  int __reg_num_11_(0);
  int __reg_num_10_(0);
  int __reg_num_20_(0);
  int __reg_num_16_(0);
  int __reg_num_15_(0);
  int __reg_num_14_(0);
  int __reg_num_19_(0);
  int __reg_num_18_(0);
  int __reg_num_17_(0);
  int __reg_num_13_(0);
  int __reg_num_21_(0);
  __reg_num_8_ = 1;
  __reg_num_9_ = i_3 == __reg_num_8_;
  __reg_num_10_ = 2;
  __reg_num_11_ = i_3 == __reg_num_10_;
  __reg_num_12_ = __reg_num_9_ || __reg_num_11_;
  if(__reg_num_12_) goto __LABEL_0;
  __reg_num_14_ = 1;
  __reg_num_15_ = i_3 - __reg_num_14_;
  __reg_num_16_ = fib(__reg_num_15_);
  __reg_num_17_ = 2;
  __reg_num_18_ = i_3 - __reg_num_17_;
  __reg_num_19_ = fib(__reg_num_18_);
  __reg_num_20_ = __reg_num_16_ + __reg_num_19_;
  return __reg_num_20_;
  goto __LABEL_1;
__LABEL_0: ;
  __reg_num_13_ = 1;
  return __reg_num_13_;
__LABEL_1: ;
  return __reg_num_21_;
}

int main(void)
{
  string __reg_str_6_("");
  string __reg_str_5_("");
  string __reg_str_4_("");
  string __reg_str_3_("");
  int __reg_num_2_(0);
  int __reg_num_1_(0);
  string __reg_str_0_("");
  int __reg_num_7_(0);
  __reg_str_5_ = "\n";
  __reg_num_1_ = 17;
  __reg_num_2_ = fib(__reg_num_1_);
  __reg_str_3_ = to_string(__reg_num_2_);
  __reg_str_0_ = "fib(17) = ";
  __reg_str_4_ = __str_concat(__reg_str_0_, __reg_str_3_);
  __reg_str_6_ = __str_concat(__reg_str_4_, __reg_str_5_);
  write(__reg_str_6_);
  return __reg_num_7_;
}
```

6.3 Example: stringfun

This program manipulates a given string, splitting it up and replacing certain characters with other ones, to a good end. It also demonstrates how regular expressions fit into strlang.

```
// stringsfun.str

main(^) -> #
{
	$str;
	$vowelly;
	$vowelless;
	#i;

	str <- "Clinton deploys vowels to Bosnia.  Cities of Sjlbvdnzv, Grzny to
Be First Recipients.";

	write("Headline:\n" + str + "\n");

	i <- str % "\\.  ";
	vowelless <- str - (i + 1);
	vowelly <- munge(str - -(i + 1));

	write("\nAfter operation vowel storm:\n" + vowelly + vowelless + "\n");

}

munge($str) -> $
{
	$tmp;
	tmp <- str ~ "[ln]" ~ "o";
	tmp <- tmp ~ "[rs]" ~ "e";
	tmp <- tmp ~ "[t]" ~ "a";
	tmp <- tmp ~ "[zjv]" ~ "i";
	-> tmp;
}
```

This is the C++ code produced by the compiler for the stringsfun program.

```
// stringsfun.cpp
#include "strlib.h"

int main(void);

string munge(string&);


int main(void)
{
  string str_1("");
  string vowelly_1("");
  string vowelless_1("");
  int i_1(0);
  string __reg_str_32_("");
  string __reg_str_31_("");
  string __reg_str_30_("");
  string __reg_str_29_("");
  string __reg_str_28_("");
  int __reg_num_27_(0);
  string __reg_str_26_("");
  string __reg_str_25_("");
  int __reg_num_24_(0);
  int __reg_num_23_(0);
  string __reg_str_22_("");
  string __reg_str_21_("");
  int __reg_num_20_(0);
  int __reg_num_19_(0);
  int __reg_num_18_(0);
  string __reg_str_17_("");
  string __reg_str_16_("");
  string __reg_str_15_("");
  string __reg_str_14_("");
  string __reg_str_13_("");
  int __reg_num_33_(0);
  __reg_str_32_ = "Clinton deploys vowels to Bosnia.  Cities of Sjlbvdnzv,
Grzny to Be First Recipients.";
  str_1 = __reg_str_32_;
  __reg_str_30_ = "\n";
  __reg_str_28_ = "Headline:\n";
  __reg_str_29_ = __str_concat(__reg_str_28_, str_1);
  __reg_str_31_ = __str_concat(__reg_str_29_, __reg_str_30_);
  write(__reg_str_31_);
  __reg_str_26_ = "\\.  ";
  __reg_num_27_ = __str_index(str_1, __reg_str_26_);
  i_1 = __reg_num_27_;
  __reg_num_23_ = 1;
  __reg_num_24_ = i_1 + __reg_num_23_;
  __reg_str_25_ = __str_substr(str_1, __reg_num_24_);
  vowelless_1 = __reg_str_25_;
  __reg_num_18_ = 1;
  __reg_num_19_ = i_1 + __reg_num_18_;
  __reg_num_20_ = - __reg_num_19_;
  __reg_str_21_ = __str_substr(str_1, __reg_num_20_);
  __reg_str_22_ = munge(__reg_str_21_);
  vowelly_1 = __reg_str_22_;
  __reg_str_16_ = "\n";
  __reg_str_13_ = "\nAfter operation vowel storm:\n";
  __reg_str_14_ = __str_concat(__reg_str_13_, vowelly_1);
  __reg_str_15_ = __str_concat(__reg_str_14_, vowelless_1);
  __reg_str_17_ = __str_concat(__reg_str_15_, __reg_str_16_);
  write(__reg_str_17_);
  return __reg_num_33_;
}

string munge(string& str_2)
{
  string tmp_2("");
  string __reg_str_11_("");
  string __reg_str_10_("");
  string __reg_str_9_("");
  string __reg_str_8_("");
  string __reg_str_7_("");
  string __reg_str_6_("");
  string __reg_str_5_("");
  string __reg_str_4_("");
  string __reg_str_3_("");
  string __reg_str_2_("");
  string __reg_str_1_("");
  string __reg_str_0_("");
  string __reg_str_12_("");
  __reg_str_10_ = "o";
  __reg_str_9_ = "[ln]";
  __reg_str_11_ = __str_replace(str_2, __reg_str_9_, __reg_str_10_);
  tmp_2 = __reg_str_11_;
  __reg_str_7_ = "e";
  __reg_str_6_ = "[rs]";
  __reg_str_8_ = __str_replace(tmp_2, __reg_str_6_, __reg_str_7_);
  tmp_2 = __reg_str_8_;
  __reg_str_4_ = "a";
  __reg_str_3_ = "[t]";
  __reg_str_5_ = __str_replace(tmp_2, __reg_str_3_, __reg_str_4_);
  tmp_2 = __reg_str_5_;
  __reg_str_1_ = "i";
  __reg_str_0_ = "[zjv]";
  __reg_str_2_ = __str_replace(tmp_2, __reg_str_0_, __reg_str_1_);
  tmp_2 = __reg_str_2_;
  return tmp_2;
  return __reg_str_12_;
}
```

Chapter 7

Lessons Learned

The advantage of being a one-person team is that you are the one determining the objectives and how to get there. That said, designing a language and a compiler is not a trivial process. I left my original group due to differences in goals and approach. As a result, the project started effectively a month behind schedule. Even so, the experience with the original group was actually valuable. The debates and discussions we had helped me to get a good sense of the issues to consider when designing my language. Moreover, I learned to prioritize clarity and simplicity above all else, and to keep practical considerations uppermost.

The most difficult part of doing a project on your own is the lack of external feedback. While I was able to get lots of feedback when designing the language, I could definitely have benefited from somebody looking at my code and questioning my approach during the implementation phase. At several points, I was forced to essentially rewrite one of the compiler passes from scratch because my design had neglected to take into account some important consideration.

Another challenge with the project was pacing. Concerned about time constraints, I wound up implementing the bulk of the compiler in around 4 days. This was good in terms of time, but it meant that after rushing the initial implementation, I then spent several weeks beating out bugs and cleaning up hacks. A more balanced schedule would likely have meant saving time in the long-term.

One surprise was the difficulty in writing good testcases. While I attempted to be exhaustive, writing simple testcases for each and every feature and implementation detail, crafting more sophisticated testcases was not easy. I also discovered on numerous occasions that features I thought I had tested were actually buggy, but I had simply failed to be comprehensive in my testcase.

Regarding the OCaml language, I was a skeptic at the beginning. However, it quickly grew on me. Debugging in a strongly typed language like OCaml is a dream compared to in C or C++. I simply spent a lot less time hunting for the source of a bug, and a lot more time writing code. While the omnipresent type errors were annoying, they certainly proved less difficult and more informative than the segmentation faults I would have encountered attempting the project in a more conventional language.

All in all, I enjoyed working on this project a great deal and I hope to be doing further work or research on compilers and programming languages in the future.

Appendix

Source Code Listing

Ast.ml

```
(* abstract syntax tree
       -defines the AST produces by the parser
       -includes functions to dump the AST to screen
*)

type bop = Plus | Minus | Times | Divide | Mod | Less | Greater | Leq | Geq |
       Eq | Neq | Or | And

type uop = Neg | Not | Len | Keys | Vals

let depth = ref 0

type simple_type =
       Str
       | Num
       | None (* expressions with no type - void functions *)

type var_type =
       Map of simple_type * simple_type
       | Simple of simple_type

type var = string * var_type

type var_decl = string * var_type * int
type func_decl = string * var_type * var_type list * int

type decl =
       FuncDecl of func_decl
       | VarDecl of var_decl

type expr =
       StrLiteral of string
       | NumLiteral of int
       | Replace of expr * expr * expr (* string matching *)
       | Binop of expr * bop * expr
       | Unop of expr * uop
       | Assign of lvalue * expr (* lvalue and right side *)
       | FuncCall of string * expr list
       | Rvalue of lvalue (* variable (on the right side) *)
       | NoExpr

and lvalue = string * expr (* expression is optional - map accessor *)

type stmt =
       CodeBlock of block
       | Loop of expr * block
       | Conditional of expr * block * block
       | Return of expr
       | Expression of expr

and block = {
       locals: var list;
       statements: stmt list;
       block_id: int;
}

and func = {
       name : string;
       ret_type: var_type;
       body: block;
```

```
        formals : var list
}

type program = {
        globals: var list;
        functions: func list;
        block_count: int
}

let rec tabs_helper = function
        0 -> ""
        | x -> "  " ^ tabs_helper (x-1)

let rec tabs = function
        0 -> tabs_helper depth.contents
        | x -> ""

let string_of_binop = function
        Plus -> "+" | Minus -> "-" | Times -> "*" | Divide -> "/" | Mod -> "%"
        | Less -> "<" | Greater -> ">" | Leq -> "<=" | Geq -> ">=" | Eq -> "=="
        | Neq -> "!=" | Or -> "|" | And -> "&"

let string_of_unop = function
        Neg -> "-" | Not -> "!" | Len -> "^" | Keys -> "@%" | Vals -> "@@"

let rec string_of_lval = function
        (name, expr) -> name ^ if expr <> NoExpr then "[" ^ string_of_expr expr ^ "]" else
""

and string_of_expr = function
        StrLiteral(s)  -> s
        | NumLiteral(l)                -> string_of_int l
        | Replace(e1, e2, e3) -> string_of_expr e1 ^ " ~ " ^ string_of_expr e2 ^ " ~ " ^
string_of_expr e3
        | Binop(e1, op, e2) -> string_of_expr e1 ^ " " ^ string_of_binop op ^ " " ^
string_of_expr e2
        | Unop(e, o) -> string_of_unop o ^ string_of_expr e
        | Assign(lv, e) -> string_of_lval lv ^ " <- " ^ string_of_expr e
        | FuncCall(f, v) -> f ^ "(" ^ String.concat "; " (List.map string_of_expr v) ^ ")"
        | Rvalue(lv) -> string_of_lval lv
        | NoExpr -> ""

let rec string_of_stmt = function
        CodeBlock(b) -> string_of_block b
        | Loop(e, b) ->
                "\n" ^ tabs 0 ^ "< " ^ string_of_expr e ^ " >\n" ^ string_of_block b
        | Conditional(e, iftrue, iffalse) ->
                "\n" ^ tabs 0 ^ "[" ^ string_of_expr e ^ "]\n" ^ string_of_block iftrue ^
                if iffalse.block_id != -1 then tabs 0 ^ "![]\n" ^    string_of_block
iffalse
                else ""
        | Return(e) -> tabs 0 ^ "-> " ^ string_of_expr e ^ ";\n"
        | Expression(e) -> tabs 0 ^ string_of_expr e ^ ";\n"

and string_of_block (b:block) =
        let s = tabs 0 ^ "{\n" in
        depth := depth.contents + 1;
        let s = s ^ string_of_vars b.locals ^ (String.concat "" (List.map string_of_stmt
b.statements)) in
        depth := depth.contents - 1; let s = s ^ "" in s ^ tabs 0 ^ "}\n\n"

and string_of_formals = function
                []  -> "^"
                | formals -> String.concat "; " (List.map string_of_var formals)

and string_of_func (f:func) =
        let formals = string_of_formals f.formals in
        let ret_type = string_of_type f.ret_type in
        let ret_type = if ret_type = "" then "^" else ret_type in
        tabs 0 ^ f.name ^ "(" ^ formals ^ ")"
        ^ " -> " ^ ret_type ^ "\n" ^ string_of_block f.body

and string_of_funclist = function
        [] -> ""
        | flist -> String.concat "\n" (List.map string_of_func flist)

and string_of_var (v:var) =
        (string_of_type (snd v)) ^ " " ^ fst v

and string_of_vars = function
        [] -> ""
        | v -> tabs 0 ^ String.concat (";\n" ^ tabs 0) (List.map string_of_var v) ^
";\n\n"

and string_of_type = function
        Map(k,v)       -> "%[" ^ (string_of_simple_type k) ^ ";" ^ (string_of_simple_type
v) ^ "]"
        | Simple(t) -> string_of_simple_type t

and string_of_simple_type = function
        Str -> "$"
        | Num -> "#"
        | None  -> "^"

let string_of_decl = function
        VarDecl(n, t, id) -> (string_of_int id) ^ " " ^ n ^ " " ^ string_of_type t
        | FuncDecl(n, t, f, id) ->
                (string_of_int id) ^ " " ^ n ^ " (" ^ String.concat "; " (List.map
string_of_type f) ^
                ") " ^ string_of_type t

let string_of_program (p:program) =
        depth := 0; string_of_vars p.globals ^ string_of_funclist p.functions
```

Check.ml

```
(* checking pass
       -perform semantic checks on all expressions
       -annotate expressions in AST with type information
       -resolve identifier names
*)

open Ast

(* checked c_expression *)
type c_expr =
       StrLiteral of string
       | NumLiteral of int
       | Binop of var_type * c_expr * bop * c_expr
       | Unop of var_type * c_expr * uop
       | Assign of var_type * c_lvalue * c_expr (* lvalue and right side *)
       | FuncCall of func_decl * c_expr list
       | Rvalue of var_type * c_lvalue (* variable (on the right side) *)
       | NoExpr

and c_lvalue = var_decl * c_expr

type c_stmt =
       CodeBlock of c_block
       | Loop of c_expr * c_block
       | Conditional of c_expr * c_block * c_block
       | Return of c_expr
       | Expression of c_expr

and c_block = {
       c_locals: var_decl list;
       c_statements: c_stmt list;
       c_block_id: int;
}

and c_func = {
       c_formals: var_decl list;
       c_header: func_decl;
       c_body: c_block;
}

type c_program = {
       c_globals: var_decl list;
       c_functions: c_func list;
       c_block_count: int
}

let build_fdecl (name:string) (ret:var_type) (args:var_type list) =
       (name, ret, args, 0)

let get_ret_of_fdecl (f:func_decl) =
       let (_,t,_,_) = f in
       t

let main_fdecl (f:c_func) =
       let fdecl = f.c_header in
       let (name, t, formals, _) = fdecl in
       if name = "main" && t = Simple(Num) && formals = [] then true
       else false

let type_of_expr = function
       StrLiteral(s) -> Simple(Str)
       | NumLiteral(n) -> Simple(Num)
       | NoExpr -> Simple(None)
       | Binop(t,_,_,_) -> t
       | Unop(t,_,_) -> t
       | Rvalue(t,_) -> t
```

```
	| Assign(t,_,_) -> t
	| FuncCall(fdecl,_) -> let (_,t,_,_) = fdecl in t

let binop_error (t1:var_type) (t2:var_type) (op:Ast.bop) =
	raise(Failure("operator " ^ (string_of_binop op) ^ " not compatible with 
expressions of type " ^
		(string_of_type t1) ^ " and " ^ (string_of_type t2)))

let unop_error (t:var_type) (op:Ast.uop) =
	raise(Failure("operator " ^ (string_of_unop op) ^ " not compatible with expression 
of type " ^
		(string_of_type t)))

let check_binop (c1:c_expr) (c2:c_expr) (op:Ast.bop) =
	let (t1, t2) = (type_of_expr c1, type_of_expr c2) in
	match(t1, t2) with
		(Simple(Num), Simple(Num)) -> Binop(Simple(Num), c1, op, c2) (* standard 
arithmetic binops *)
		| (Simple(Str), Simple(Str)) -> (* string binops with 2 string operands *)
			let f = (match op with
			Less -> build_fdecl "__str_less" (Simple(Num)) [t1; t2]
			| Leq -> build_fdecl "__str_lessequal" (Simple(Num)) [t1; t2]
			| Greater -> build_fdecl "__str_greater" (Simple(Num)) [t1; t2]
			| Geq -> build_fdecl "__str_greaterequal" (Simple(Num)) [t1; t2]
			| Eq -> build_fdecl "__str_equal" (Simple(Num)) [t1; t2]
			| Neq -> build_fdecl "__str_notequal" (Simple(Num)) [t1; t2]
			| Plus -> build_fdecl "__str_concat" t1 [t1; t2]
			| Divide ->build_fdecl "__str_match" t1 [t1; t2]
			| Mod -> build_fdecl "__str_index" (Simple(Num)) [t1; t2]
			| _ -> binop_error t1 t2 op) in
			FuncCall(f, [c1; c2])
		| (Simple(Str), Simple(Num)) -> (* string binops with 1 string operand *)
			(match op with
			Minus -> let f = build_fdecl "__str_substr" t1 [t1; t2] in
				FuncCall(f, [c1; c2])
			| _ -> binop_error t1 t2 op)
		| (Map(k1,v1), Map(k2,v2)) -> (* map binops *)
			if k1 = k2 && v1 = v2 then
			let f = (match op with
				Eq -> build_fdecl "__map_equal" (Simple(Num)) [t1; t2]
				| Neq -> build_fdecl "__map_notequal" (Simple(Num)) [t1; 
t2]
				| _ -> binop_error t1 t2 op) in
			FuncCall(f, [c1; c2])
			else binop_error t1 t2 op
		| (Map(k, v), Simple(l)) ->
			if k = l then
				let f = (match op with
					Minus -> build_fdecl "__map_remove" (Simple(v)) [t1; 
t2]
					| Mod -> build_fdecl "__map_exists" (Simple(Num)) 
[t1; t2]
					| _ -> binop_error t1 t2 op) in
				FuncCall(f, [c1; c2])
			else binop_error t1 t2 op
		| _ -> binop_error t1 t2 op

let check_unop (c:c_expr) (op:Ast.uop) =
	let t = type_of_expr c in
	match t with
		Simple(Num) ->
			(match op with
				(Neg | Not) -> Unop(Simple(Num), c, op)
				| _ -> unop_error t op)
		| Simple(Str) ->
			(match op with
				Len -> let f = build_fdecl "__str_len" (Simple(Num)) [t] in
					FuncCall(f, [c])
				| _ -> unop_error t op)
			| Map(k, v) ->
				let f = (match op with
					Len -> build_fdecl "__map_len" (Simple(Num)) [t]
					| Keys -> build_fdecl "__map_keys" (Map(Num, k)) [t]
					| Vals -> build_fdecl "__map_vals" (Map(Num, v)) [t]
					| _ -> unop_error t op) in
					FuncCall(f, [c])
			| _ -> unop_error t op

let rec compare_arglists formals actuals =
	match (formals,actuals) with
	([],[]) -> true
	| (head1::tail1, head2::tail2)
		-> (head1 = head2) && compare_arglists tail1 tail2
	| _ -> false

and check_func (name:string) (cl:c_expr list) env =
	let decl = Symtab.symtab_find name env in
	let func = (match decl with FuncDecl(f) -> f
		| _ -> raise(Failure("symbol " ^ name ^ " is not a function"))) in
	let (_,t,formals,_) = func in
	let actuals = List.map type_of_expr cl in
	if (List.length formals) = (List.length actuals) then
		if compare_arglists formals actuals then
			FuncCall(func, cl)
		else
			raise(Failure("function " ^ name ^ "'s argument types don't match 
its formals"))
	else raise(Failure("function " ^ name ^ " expected " ^ (string_of_int (List.length 
actuals)) ^
		" arguments but called with " ^ (string_of_int (List.length formals))))

and check_lvalue (lv:lvalue) (checked:c_expr) env =
	let (name,e) = lv in
	let decl = Symtab.symtab_find name env in
	let var = (match decl with VarDecl(v) -> v
			| _ -> raise(Failure("symbol " ^ name ^ " is not a variable"))) in
	let (_,base_t,_) = var in
	let t = type_of_expr checked in
	match t with
		Simple(None) -> (base_t, (var, checked))
		| _ ->
			match base_t with
				Map(k, v) ->
					if t = Simple(k) then (Simple(v), (var, checked))
					else raise(Failure("map type does not match accessor 
type"))
				| _ -> raise(Failure("cannot apply accessor to non-map"))

and check_expr (e:expr) env =
	match e with
	Ast.StrLiteral(s) -> StrLiteral(s)
	| Ast.NumLiteral(n) -> NumLiteral(n)
	| Ast.NoExpr -> NoExpr
	| Ast.Replace(e1, e2, e3) ->
		let c1, c2 = (check_expr e1 env, check_expr e2 env) in
		let c3 = check_expr e3 env in
		let (t1, t2, t3) = (type_of_expr c1, type_of_expr c2, type_of_expr c3) in
		if(t1 = t2 && t2 = t3 && t3 = Simple(Str)) then
			let f = build_fdecl "__str_replace" t1 [t1; t2; t3] in
			FuncCall(f, [c1; c2; c3])
		else raise(Failure("operator ~ requires 3 string expressions"))
	| Ast.Binop(e1, op, e2) ->
		let (c1, c2) = (check_expr e1 env, check_expr e2 env) in
		check_binop c1 c2 op
	| Ast.Unop(e1, op) ->
		let checked = check_expr e1 env in
		check_unop checked op
	| Ast.Rvalue(l) ->
```

```
		let checked = check_expr (snd l) env in
		let (result_t, lv) = check_lvalue l checked env in
		Rvalue(result_t, lv)
	| Ast.Assign(l, ae) ->
		let checked = check_expr ae env in
		let deref = check_expr (snd l) env in
		let (result_t, lv) = check_lvalue l deref env in
		let t = type_of_expr checked in
		if t = result_t then Assign(t, lv, checked)
		else
			(match (checked, result_t) with
				(NumLiteral(0), Map(_,_)) -> let f = build_fdecl
"__map_empty" result_t [result_t] in
				FuncCall(f, [Rvalue(result_t, lv)])
			| _ -> raise(Failure("assignment not compatible with expressions of
type " ^
			string_of_type result_t ^ " and " ^ string_of_type t)))
	| Ast.FuncCall(name, el) ->
		let checked = check_exprlist el env in
		check_func name checked env

and check_exprlist (el:expr list) env =
	match el with
	[] -> []
	| head :: tail -> (check_expr head env) :: (check_exprlist tail env)

(* check a single statement *)
let rec check_stmt (s:stmt) ret_type env =
	match s with
	Ast.CodeBlock(b) -> CodeBlock(check_block b ret_type env)
	| Ast.Loop(e, b) -> let checked = check_expr e env in
		if type_of_expr checked = Simple(Num) then Loop(checked, check_block b
ret_type env)
		else raise(Failure("loop condition expression must be numeric"))
	| Ast.Conditional(e, b1, b2) -> let checked = check_expr e env in
		if type_of_expr checked = Simple(Num) then Conditional(checked,
check_block b1 ret_type env, check_block b2 ret_type env)
		else raise(Failure("if condition expression must be numeric"))
	| Ast.Return(e) -> let checked = check_expr e env in
		let t = type_of_expr checked in
		if t = ret_type then Return(checked) else
			raise(Failure("function return type " ^ string_of_type ret_type ^ "
not compatible with expression of type " ^ string_of_type t))
	| Ast.Expression(e) -> Expression(check_expr e env)

(* check a list of statements *)
and check_stmtlist (s:stmt list) (ret_type:var_type) env =
	match s with
	[] -> []
	| head :: tail -> check_stmt head ret_type env :: check_stmtlist tail ret_type env

and check_vdecllist (v:var list) env =
	match v with
	[] -> []
	| head :: tail ->
		let decl = Symtab.symtab_find (fst head) env in
		match decl with
			FuncDecl(f) -> raise(Failure("symbol is not a variable"))
			| VarDecl(v) -> v :: check_vdecllist tail env

and check_fdecl (f:string) env =
	let decl = Symtab.symtab_find f env in
	match decl with
		VarDecl(v) -> raise(Failure("symbol is not a function"))
		| FuncDecl(f) -> f

(* check a block *)
and check_block (b:block) (ret_type:var_type) env =
	let vars = check_vdecllist b.locals (fst env, b.block_id) in
```

```
        { c_locals = vars;
                c_statements = check_stmtlist b.statements ret_type (fst env, b.block_id);
                c_block_id = b.block_id}

(* check a function *)
and check_func (f:func) env =
        let checked = check_block f.body f.ret_type env in
        let formals = check_vdecllist f.formals (fst env, f.body.block_id) in
        { c_header = check_fdecl f.name env; c_body = checked; c_formals = formals }

(* check a function list *)
and check_funclist (funcs:func list) env =
        match funcs with
        [] -> []
        | head :: tail -> check_func head env :: check_funclist tail env

and check_main (f:c_func list) =
        if (List.filter main_fdecl f) = [] then false
        else true

let check_program (p:program) env =
        let vars = check_vdecllist p.globals env in
        let checked = check_funclist p.functions env in
        if (check_main checked) then {c_globals = vars; c_functions = checked;
c_block_count = p.block_count}
        else raise(Failure("function main(^) -> # not found"))
```

Output.ml

```
(* output
	-takes simple IR produced by the simlification pass
	-outputs C++ code
*)

open Ast
open Check
open Simple

let type_of_var (v:simple_var) =
	let (_,t,_) = v in
	t

let c_of_bop = function
	Plus -> "+" | Minus -> "-" | Times -> "*" | Divide -> "/" | Mod -> "%"
	| Less -> "<" | Greater -> ">" | Leq -> "<=" | Geq -> ">=" | Eq -> "=="
	| Neq -> "!=" | Or -> "||" | And -> "&&"

let c_of_uop = function
	Neg -> "-" | Not -> "!" | _ -> raise(Failure("internal error"))

let c_of_var_type = function
	Simple(Str) -> "string"
	| Simple(Num) -> "int"
	| Simple(None) -> "void"
	| Map(k,v) ->
		"map <" ^ (match (k,v) with
			(Str,Str) -> "string,string"
			| (Str,Num) -> "string,int"
			| (Num,Num) -> "int,int"
			| (Num,Str) -> "int,string"
			| _ -> raise(Failure("internal error"))) ^ ">"

let c_of_param_type t =
	if t = Simple(None) then "void"
	else (c_of_var_type t) ^ "&"

let c_of_var_name (decl:simple_var) =
	let (name,_,id) = decl in
	let suffix = if id = -1 then "" else string_of_int id in
	name ^ "_" ^ suffix

let c_of_var_init = function
	Simple(Num) -> "(0)"
	| Simple(Str) -> "(\"\")"
	| Map(_,_) -> ""
	| _ -> raise(Failure("internal error"))

let c_of_var_decl (v:simple_var) =
	let (_, t, _) = v in
	(c_of_var_type t) ^ " " ^ c_of_var_name v

let c_of_param (p:simple_var) =
	let (_, t, _) = p in
	(c_of_param_type t) ^ " " ^ c_of_var_name p

let c_of_formals_list = function
	[] -> "void"
	| formals -> String.concat ", " (List.map c_of_param formals)

let c_of_actual_list = function
	[] -> ""
	| actuals -> String.concat ", " (List.map c_of_var_name actuals)

let c_of_var_decl_def (v:simple_var) =
	let (_,t,_) = v in
	(c_of_var_decl v) ^ (c_of_var_init t)
```

```
let c_of_var_decl_list = function
        [] -> ""
        | vars-> (tabs 0 ^ (String.concat (";\n" ^ tabs 0) (List.map c_of_var_decl_def
vars)) ^ ";\n")

let c_of_func_decl_formals = function
        [] -> "void"
        | vars -> String.concat (", ") (List.map c_of_param_type vars)

let c_of_func_decl (f:simple_fdecl) =
        let (name, ret, formals, id) = f in
        c_of_var_type ret ^ " " ^ name ^ "(" ^ c_of_func_decl_formals formals ^ ");\n"

let c_of_func_decl_list = function
        [] -> ""
        | fdecls -> String.concat ("\n") (List.map c_of_func_decl fdecls) ^ "\n\n"

let c_of_method_call (result:simple_var) (fname:string) (receiver:simple_var) args =
        let(_,t,_) = result in
        (if t <> Simple(None) then (c_of_var_name result) ^ " = " else "") ^
(c_of_var_name receiver) ^ "." ^
                fname ^ "(" ^ (c_of_actual_list args) ^ ")"

let c_of_func_call (result:simple_var) (fname:string) (args:simple_var list) =
        let (_,t,_) = result in
        (if t <> Simple(None) then (c_of_var_name result) ^ " = " else "") ^ fname ^ "(" ^
                (c_of_actual_list args) ^ ")"

let c_of_func_name = function
        (name,_,_,_) -> name

let c_of_unop (result:simple_var) (op:Ast.uop) (expr:simple_var) =
                (c_of_var_name result) ^ " = " ^ c_of_uop op ^ " " ^ (c_of_var_name expr)

let c_of_binop (result:simple_var) (op:Ast.bop) (e1:simple_var) (e2:simple_var) =
        (c_of_var_name result) ^ " = " ^ (c_of_var_name e1) ^ " " ^
                c_of_bop op ^ " " ^ (c_of_var_name e2)

let c_of_expr = function
        Bin(r, op, e1, e2) -> (c_of_var_name r) ^ " = " ^ (c_of_var_name e1) ^ " " ^
                c_of_bop op ^ " " ^ (c_of_var_name e2)
        | Un(r, op, e) -> c_of_unop r op e
        | Lit(r, v) -> (c_of_var_name r) ^ " = " ^
                (match v with
                StrLit(s) -> s
                | NumLit(n) -> string_of_int n)
        | Deref(result, base, accessor) ->
                (c_of_var_name result) ^ " = " ^ (c_of_var_name base) ^ "[" ^
c_of_var_name accessor ^ "]"
        | Alias(base, accessor, input) ->
                let (_,t,_) = accessor in
                let extra = if t <> Simple(None) then "[" ^ (c_of_var_name accessor) ^ "]"
else "" in
                (c_of_var_name base) ^ extra ^ " = " ^ (c_of_var_name input)
        | Call(r, f, el) ->
                c_of_func_call r (c_of_func_name f) el

let rec c_of_stmt = function
        Decl(v) -> if type_of_var v <> Simple(None) then tabs 0 ^ (c_of_var_decl_def v) ^
";\n"
        else ""
        | If(r, l) -> tabs 0 ^ "if(" ^ (c_of_var_name r) ^ ") goto " ^ l ^ ";\n"
        | Label(s) -> s ^ ": ;\n"
        | Jmp(s) -> tabs 0 ^ "goto " ^ s ^";\n"
        | Ret(e) -> let (_,t,_) = e in tabs 0 ^ "return" ^ (if t <> Simple(None) then " "
^ c_of_var_name e else "") ^ ";\n"
        | Expr(e) -> tabs 0 ^ c_of_expr e ^ ";\n"
```

```
and c_of_stmt_list = function
	[] -> ""
	| head::tail -> (c_of_stmt head) ^ (c_of_stmt_list tail)

and c_of_block (sl:simple_stmt list) =
	let s = "\n" ^ tabs 0 ^ "{\n" in
	depth := depth.contents + 1;
	let s = s ^ String.concat "" (List.map c_of_stmt sl) in
	depth := depth.contents - 1; s ^ tabs 0 ^ "}\n\n"


and c_of_header (header:simple_fdecl) (args:simple_var list) =
	let (name, ret, _, id) = header in
	tabs 0 ^ c_of_var_type ret ^ " " ^ name ^ "(" ^ c_of_formals_list args ^ ")"

and c_of_funclist = function
	[] -> ""
	| head::tail -> c_of_header head.header head.args ^ (c_of_block head.code) ^
		(c_of_funclist tail)

let rec c_of_simple (p:simple_program) =
	depth := 0; "#include \"strlib.h\"\n\n" ^ (c_of_func_decl_list p.fdecls) ^
	c_of_var_decl_list p.gvars ^ c_of_funclist p.funcs
```

Parser.mly

```
%{ open Ast

let block_id = ref 1

let gen_block_id (u:unit) =
       let x = block_id.contents in
              block_id := x + 1; x

%}

%token LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET EOF
%token PLUS MINUS TIMES DIVIDE MOD
%token OR AND
%token LESS GREATER LESSEQUAL GREATEREQUAL EQUAL UNEQUAL NOT
%token CARET AT TILDE
%token LARROW RARROW

%token SEMI

%token STR NUM

%token <string> STR_LITERAL
%token <int> NUM_LITERAL

%token <string> ID

%nonassoc NOELSE

%right LARROW

%left OR
%left AND

/* theoretically EQUAL and UNEQUAL should have different precedence, but this breaks
other things */
%left LESS LESSEQUAL GREATER GREATEREQUAL EQUAL UNEQUAL

%left PLUS MINUS
%left TIMES DIVIDE MOD
%left TILDE

%left NEG NOT CARET
%left LBRACKET

%right AT

%start program
%type <Ast.program> program

%%

program:
       var_list func_list { { globals = List.rev $1; functions = List.rev $2; block_count
= gen_block_id ()} }

func_list:
       { [] }
       | func_list func { $2 :: $1 }

func:
       ID LPAREN formals_list_opt RPAREN RARROW var_type_opt block
              { { name = $1; ret_type = $6; body = $7; formals = $3} }

block:
       LBRACE var_list stmt_list RBRACE
              { {locals = List.rev $2; statements = $3; block_id = gen_block_id ()} }
```

```
stmt_list:
        { [] }
        | stmt stmt_list { $1 :: $2 }

stmt:
        block { CodeBlock($1) }
        | expr SEMI { Expression($1) }
        | LESS expr GREATER block { Loop($2, $4) }
        | LBRACKET expr RBRACKET block NOT LBRACKET RBRACKET block { Conditional($2, $4,
$8) }
        | LBRACKET expr RBRACKET block %prec NOELSE { Conditional($2, $4, {locals = [];
statements = []; block_id = -1}) }
        | RARROW expr SEMI { Return($2) }
        | RARROW SEMI { Return(NoExpr) }

expr:
        /* binary arithmetic operators */
        expr PLUS expr { Binop($1, Plus, $3) }
        | expr MINUS expr { Binop($1, Minus, $3) }
        | expr TIMES expr { Binop($1, Times, $3) }
        | expr DIVIDE expr { Binop($1, Divide, $3) }
        | expr MOD expr { Binop($1, Mod, $3) }

        /* comparison */
        | expr LESS expr { Binop($1, Less, $3) }
        | expr GREATER expr { Binop($1, Greater, $3) }
        | expr LESSEQUAL expr { Binop($1, Leq, $3) }
        | expr GREATEREQUAL expr { Binop($1, Geq, $3) }

        | expr TILDE expr TILDE expr { Replace($1, $3, $5) }

        /* equality */
        | expr EQUAL expr { Binop($1, Eq, $3) }
        | expr UNEQUAL expr { Binop($1, Neq, $3) }

        /* logical or/and */
        | expr OR expr { Binop($1, Or, $3) }
        | expr AND expr { Binop($1, And, $3) }

        /* assignment */
        | lvalue LARROW expr { Assign($1, $3) }

        /* arithmetic and logical negation */
        | MINUS expr %prec NEG { Unop($2, Neg) }
        | NOT expr { Unop($2, Not) }

        /* length operator */
        | CARET expr { Unop($2, Len) }

        /* keys operator */
        | AT MOD expr { Unop($3, Keys) }

        /* values operator */
        | AT AT expr { Unop($3, Vals) }

        /* grouping */
        | LPAREN expr RPAREN { $2 }

        /* function call */
        | ID LPAREN actuals_list_opt RPAREN { FuncCall($1, $3) }

        /* variable */
        | lvalue { Rvalue($1) }

        /* literals */
        | STR_LITERAL { StrLiteral($1) }
        | NUM_LITERAL { NumLiteral($1) }

formals_list_opt:
        formals_list { $1 }
        | CARET { [] }

formals_list:
        var { [$1] }
        | var SEMI formals_list { $1 :: $3 }

actuals_list_opt:
        { [] }
        | actuals_list { $1 }

actuals_list:
        expr { [$1] }
        | expr SEMI actuals_list { $1 :: $3 }

var_list:
        { [] }
        | var_list var SEMI { $2 :: $1 }

var:
        var_type ID { ($2, $1) }

var_type_opt:
        CARET { Simple(None) }
        | var_type {$1}

var_type:
        STR { Simple(Str) }
        | NUM { Simple(Num) }
        | MOD LBRACKET STR SEMI STR RBRACKET { Map(Str, Str) }
        | MOD LBRACKET NUM SEMI STR RBRACKET { Map(Num, Str) }
        | MOD LBRACKET STR SEMI NUM RBRACKET { Map(Str, Num) }
        | MOD LBRACKET NUM SEMI NUM RBRACKET { Map(Num, Num) }

lvalue:
        ID { ($1, NoExpr) }
        | ID LBRACKET expr RBRACKET { ($1, $3) }
```

Scanner.mll

```
(* tokenize input *)

{ open Parser }

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf }
| "//"  { comment lexbuf }
| '('    { LPAREN }            | ')'    { RPAREN }
| '{'    { LBRACE }            | '}'    { RBRACE }
| '['    { LBRACKET }  | ']'    { RBRACKET }

| '+'    { PLUS }              | '-'    { MINUS }
| '*'    { TIMES }             | '/'    { DIVIDE }
| '%'   { MOD }

| '|'    { OR }                | '&'   { AND }

| '<'   { LESS }               | '>'   { GREATER }
| "<="  { LESSEQUAL }  | ">="  { GREATEREQUAL }
| "=="  { EQUAL }              | "!="  { UNEQUAL }

| '!'   { NOT }                | '^'   { CARET }
| '~'   { TILDE }              | '@'   { AT}

| "<-"  { LARROW }             | "->"  { RARROW }
| eof   { EOF }

| '#'   { NUM }                | '$'   { STR }

| ';'   { SEMI }

| ['0'-'9']+ as lxm { NUM_LITERAL(int_of_string lxm) }
| '\"'[^'\"']*'\"' as lxm { STR_LITERAL(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }

| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
 "\n" { token lexbuf }
| _    { comment lexbuf }
```

Simple.ml

```
(* simplification pass
  convert checked AST to flat intermediate representation
        -simplify expressions
        -flatten blocks
        -replace loops with labels and branches
*)

open Ast
open Check

type simple_type = Ast.var_type
type simple_var = Ast.var_decl
type simple_fdecl = Ast.func_decl

type simple_lit =
        StrLit of string
        | NumLit of int

type simple_expr =
        Bin of simple_var * Ast.bop * simple_var * simple_var (* a = b op c *)
        | Un of simple_var * Ast.uop * simple_var (* a = op b *)
        | Call of simple_var * simple_fdecl * simple_var list (* a = b(c..d..) *)
        | Lit of simple_var * simple_lit (* a = "b" *)
        | Deref of simple_var * simple_var * simple_var (* a = b[c] *)
        | Alias of simple_var * simple_var * simple_var (* a[b] = c *)

type simple_stmt =
        | Decl of simple_var
        | If of simple_var * string
        | Jmp of string
        | Label of string
        | Ret of simple_var
        | Expr of simple_expr

and simple_func = {
        args: simple_var list;
        header: simple_fdecl;
        code: simple_stmt list;
}

type simple_program = {
        gvars: simple_var list;
        fdecls: simple_fdecl list;
        funcs: simple_func list;
        blocks: int;
}

let tmp_reg_id = ref 0
let label_id = ref 0

let gen_tmp_var t =
        let x = tmp_reg_id.contents in
        let prefix = (match t with
                Simple(Str) -> "__reg_str_" | Simple(Num) -> "__reg_num_" |
                Map(_,_) -> "__reg_map_" | _ -> raise(Failure("unsupported type"))) in
        tmp_reg_id := x + 1; (prefix ^ (string_of_int x), t, -1)

let gen_tmp_label (s:unit) =
        let x = label_id.contents in
        label_id := x + 1; "__LABEL_" ^ (string_of_int x)

let is_vdecl (s:simple_stmt) =
        match s with
        Decl(_) -> true
        | _ -> false

let is_not_vdecl (s:simple_stmt) =
        not (is_vdecl s)

let rec simplify_rvalue (t:simple_type) (l:c_lvalue) =
        let(decl, e) = l in
        if e = NoExpr then
                ([], decl)
        else
                let (se, r) = simplify_expr e in
                let tmp = gen_tmp_var t in
                ([Decl(tmp)] @ se @ [Expr(Deref(tmp, decl, r))], tmp)
                (* side-effects are that passing map expression always is done using a
temporary *)

and simplify_binop (t:simple_type) (e1:c_expr) (e2:c_expr) (op:bop) =
        let (se1, r1) = simplify_expr e1 in
        let (se2, r2) = simplify_expr e2 in
        let tmp = gen_tmp_var t in
        ([Decl(tmp)] @ se1 @ se2 @ [Expr(Bin(tmp, op, r1, r2))], tmp)

and simplify_unop (t:simple_type) (e1:c_expr) (op:uop) =
        let (se, r) = simplify_expr e1 in
        let tmp = gen_tmp_var t in
        ([Decl(tmp)] @ se @ [Expr(Un(tmp, op, r))], tmp)

and simplify_assign (t:simple_type) (l:c_lvalue) (e:c_expr) =
        let (se, r) = simplify_expr e in
        let (decl, l_expr) = l in
        if l_expr = NoExpr
                then (se @ [Expr(Alias(decl, ("", Simple(None), 1), r))], r)
        else
                let (le, lr) = simplify_expr l_expr in
                (se @ le @ [Expr(Alias(decl,lr,r))], r)

and simplify_call (fdecl:simple_fdecl) (el:c_expr list)
        (rl:simple_var list) (sl:simple_stmt list) =
        match el with
        [] ->
                let (_,t,_,_) = fdecl in
                let tmp = (match t with
                        Simple(None) -> ("__none", t, -1) | _ -> gen_tmp_var t) in
                        let c = Call(tmp, fdecl, (List.rev rl)) in (* reverse the list of
results as it was constructed right-to-left *)
                        ([Decl(tmp)] @ sl @ [Expr(c)], tmp)
        | head :: tail ->
                let (se, r) = simplify_expr head in
                (* tack on the result to the list of results, the intermediate statements
to the list of statements *)
                simplify_call fdecl tail (r :: rl) (se @ sl)

and simplify_expr (e:c_expr) =
        match e with
        StrLiteral(s) ->
                let tmp = gen_tmp_var (Simple(Str)) in
                ([Decl(tmp); Expr(Lit(tmp, StrLit(s)))], tmp)
        | NumLiteral(n) ->
                let tmp = gen_tmp_var (Simple(Num)) in
                ([Decl(tmp); Expr(Lit(tmp, NumLit(n)))], tmp)
        | NoExpr ->
                ([], ("none", Simple(None), -1))
        | Rvalue(t,l) ->
                simplify_rvalue t l
        | Binop(t, e1, op, e2) ->
                simplify_binop t e1 e2 op
        | Unop(t, e1, op) ->
                simplify_unop t e1 op
        | Assign(t, l, e1) ->
                simplify_assign t l e1
        | FuncCall(fdecl, el) ->
                simplify_call fdecl el [] []
```

```
let gen_default_ret (t:simple_type) =
        if t = Simple(None) then [Ret(("none", t, -1))]
        else let tmp = gen_tmp_var t in
                Decl(tmp) :: [Ret(tmp)]

let rec simplify_stmt (s:c_stmt) =
        match s with
        CodeBlock(b) -> simplify_block b
        | Conditional(e, b1, b2) ->
                let (se, r) = simplify_expr e in
                let sb1 = simplify_block b1 in
                let sb2 = simplify_block b2 in
                let startlabel = gen_tmp_label () in
                let endlabel = gen_tmp_label () in
                se @ [If(r, startlabel)] @ sb2 @ [Jmp(endlabel); Label(startlabel)] @ sb1
@ [Label(endlabel)]
        | Loop(e, b) ->
                let (se, r) = simplify_expr e in
                let sb = simplify_block b in
                let startlabel = gen_tmp_label () in
                let endlabel = gen_tmp_label () in
                [Jmp(endlabel); Label(startlabel)] @ sb @ [Label(endlabel)] @ se @ [If(r,
startlabel)]
        | Return(e) ->
                let (se, r) = simplify_expr e in
                se @ [Ret(r)]
        | Expression(e) -> (* only need simplified statements, not final tmp register *)
                let (se, r) = simplify_expr e in
                se

and simplify_stmtlist (slist:c_stmt list) =
        match slist with
        [] -> []
        | head :: tail -> simplify_stmt head @ simplify_stmtlist tail

and simplify_block (b:c_block) =
        let decls = List.map (fun e -> Decl(e)) b.c_locals in
        decls @ (simplify_stmtlist b.c_statements)

and simplify_fdecls = function
        [] -> []
        | head :: tail ->
                head.c_header :: simplify_fdecls tail

and simplify_func (f:c_func) =
        let body = simplify_block f.c_body in
        let ret_type = Check.get_ret_of_fdecl f.c_header in
        let body = body @ (gen_default_ret ret_type) in
        let vdecls = List.filter is_vdecl body in
        let stmts = List.filter is_not_vdecl body in
        {header = f.c_header; args = f.c_formals; code = vdecls @ stmts}

and simplify_funclist (flist:c_func list) =
        match flist with
        [] -> []
        | head :: tail -> simplify_func head :: simplify_funclist tail

let rec simplify_program (p:c_program) =
        { gvars = p.c_globals; fdecls = simplify_fdecls p.c_functions; funcs =
simplify_funclist p.c_functions; blocks = p.c_block_count}
```

Strlang.ml

```
(* compiler driver
  - invokes each compiler pass in sequence *)

open Unix

let cpp_compiler = "g++"
let pcre_include = ""
let pcre_lib = ""
let pcre_name = "-lpcrecpp"

type action = Ast | Compile | Assemble | Help | Version

let version (n:unit) =
	"strlang version 0.1 (Green Eggs 'n Ham) 12/22/11"

let usage (name:string) =
	"usage:\n" ^ name ^ "\n" ^
		"	-a file.str			(dump AST of source)\n" ^
		"	-c file.str [file.cpp]	(compile to C++)\n" ^
		"	-e file.str file.exe		(compile to executable)\n" ^
		"	-h				(display this message)\n" ^
		"	-v				(display version number)\n"

let get_compiler_path (path:string) =
	try
		let i = String.rindex path '/' in
		String.sub path 0 i
	with _ -> "."

let _ =
	let action =
		if Array.length Sys.argv > 1 then
			(match Sys.argv.(1) with
				"-a" -> if Array.length Sys.argv == 3 then Ast else Help
				| "-c" ->
					if (Array.length Sys.argv == 3) ||
						(Array.length Sys.argv == 4) then Compile
else Help
				| "-e" -> if Array.length Sys.argv == 4 then Assemble else
Help
				| "-v" -> Version
				| _ -> Help)
		else Help in
	match action with
		Help -> print_endline (usage Sys.argv.(0))
		| Version -> print_endline (version ())
		| (Ast | Compile | Assemble) ->
			let input = open_in Sys.argv.(2) in
			let lexbuf = Lexing.from_channel input in
			let ast = Parser.program Scanner.token lexbuf in
			if action = Ast then print_endline (Ast.string_of_program ast)
			else
				let env = Symtab.symtab_of_program ast in
				let checked = Check.check_program ast env in
				let simple = Simple.simplify_program checked in
				let program = Output.c_of_simple simple in
				if action = Compile then
					if Array.length Sys.argv == 3 then print_endline
program
					else
						let out = open_out Sys.argv.(3) in
						output_string out program; close_out out
				else
					let outfilename = Sys.argv.(3) ^ "_strlang.cpp" in
					let out = open_out outfilename in
					output_string out program; close_out out;
```

```
                                      execvp cpp_compiler [|cpp_compiler; pcre_include; "-
I" ^ (get_compiler_path Sys.argv.(0));
                                              outfilename; "-o"; Sys.argv.(3); pcre_lib;
pcre_name|]
```

Symtab.ml

```
(* symbol table
	-construct symbol table from AST
	-provide functions to look up symbols in the table
*)

open Ast

module SymMap = Map.Make(String)
let scope_parents = Array.create 1000 0

let string_of_symtab env =
	let symlist = SymMap.fold
		(fun s t prefix -> (string_of_decl t) :: prefix) (fst env) [] in
	let sorted = List.sort Pervasives.compare symlist in
	String.concat "\n" sorted

let rec symtab_find (name:string) env =
	let(tab, scope) = env in
	let to_find = name ^ "_" ^ (string_of_int scope) in
	if SymMap.mem to_find tab then SymMap.find to_find tab
	else
		if scope = 0 then raise (Failure("symbol " ^ name ^ " not declared in
current scope"))
		else symtab_find name (tab, scope_parents.(scope))

let rec symtab_add_decl (name:string) (decl:decl) env =
	let (tab, scope) = env in
	let to_find = name ^ "_" ^ (string_of_int scope) in
	if SymMap.mem to_find tab
		then raise(Failure("symbol " ^ name ^ " declared twice in same scope"))
	else ((SymMap.add to_find decl tab), scope )

(* add list of variables to the symbol table *)
let rec symtab_add_vars (vars:var list) env =
	match vars with
	[] -> env
	| (name,t) :: tail -> let env = symtab_add_decl name (VarDecl(name, t, snd env))
env in
		symtab_add_vars tail env

(* add declarations inside statements to the symbol table *)
let rec symtab_add_stmts (stmts:stmt list) env =
	match stmts with
	[] -> env
	| head :: tail -> let env = (match head with
		CodeBlock(b) -> symtab_add_block b env
		| Loop(e, b) -> symtab_add_block b env
		| Conditional(e, b1, b2) -> let env = symtab_add_block b1 env in
			symtab_add_block b2 env
		| _ -> env) in symtab_add_stmts tail env

and symtab_add_block (b:block) env =
	if(b.block_id != -1) then
		let (tab, scope) = env in
		let env = symtab_add_vars b.locals (tab, b.block_id) in
		let env = symtab_add_stmts b.statements env in
		scope_parents.(b.block_id) <- scope; ((fst env), scope)
	else env

and symtab_add_func (f:func) env =
	let scope = snd env in
	let args = List.map snd f.formals in
	let env = symtab_add_decl f.name (FuncDecl(f.name, f.ret_type, args, scope)) env
in
	let env = symtab_add_vars f.formals ((fst env), f.body.block_id) in
	symtab_add_block f.body ((fst env), scope)
```

```
(* add list of functions to the symbol table *)
and symtab_add_funcs (funcs:func list) env =
        match funcs with
        [] -> env
        | head :: tail -> let env = symtab_add_func head env in
                symtab_add_funcs tail env

(* add builtin functions to the symbol table *)
let add_builtins env =
        let env = symtab_add_decl "read" (FuncDecl("read", Simple(Str), [], 0)) env in
        let env = symtab_add_decl "end_input" (FuncDecl("end_input", Simple(Num), [], 0))
env in
        let env = symtab_add_decl "write" (FuncDecl("write", Simple(None), [Simple(Str)],
0)) env in
        let env = symtab_add_decl "to_string" (FuncDecl("to_string", Simple(Str),
[Simple(Num)], 0)) env in
        let env = symtab_add_decl "to_num" (FuncDecl("to_num", Simple(Num), [Simple(Str)],
0)) env in
        let env = symtab_add_decl "open" (FuncDecl("open", Simple(None), [Simple(Str);
Simple(Str)], 0)) env in
        symtab_add_decl "exit" (FuncDecl("exit", Simple(None), [Simple(Num)], 0)) env

let symtab_of_program (p:Ast.program) =
        let env = add_builtins(SymMap.empty, 0) in
        let env = symtab_add_vars p.globals env in
        symtab_add_funcs p.functions env
```

Strlib.h

```
/*
** wrappers used by strlang for manipulating string and map data-types
** implementations for builtin functions
*/

#ifndef __strlib_h
#define __strlib_h

#include <cstdio>
#include <string>
#include <map>
#include <pcrecpp.h>

using namespace std;

FILE *__in = stdin;
FILE *__out = stdout;

void write(const string &s) { fprintf(__out, "%s", s.c_str()); }

string read(void)
{
	char buffer[512];
	int len;

	fgets(buffer, 512, __in);
	len = strlen(buffer);

	if(buffer[len - 1] == '\n') buffer[len - 1] = '\0';

	else fprintf(stderr, "library error: read\n");

	return string(buffer);
}

int end_input(void) { return (__in == stdin) ? 0 : feof(__in); }

string to_string(const int n)
{
	char buf[20];
	sprintf(buf, "%d", n);
	return string(buf);
}

int to_num(const string &s)
{
	char *end;
	int i = strtol(s.c_str(), &end, 10);

	if(*end) fprintf(stderr, "library error: to_int\n");

	return i;
}

void open(const string &inout, const string &file)
{
	if(strcmp(inout.c_str(), "in") == 0)
	{
		if(__in != stdin) fclose(__in);

		if(file == "stdin") __in = stdin;

		else if(!(__in = fopen(file.c_str(), "r")))
		{
			fprintf(stderr, "library error: open(\"in\", %s)\n", file.c_str());
			__in = stdin;
		}
```

```
        }

        else if(strcmp(inout.c_str(), "out") == 0)
        {
                if(__out != stderr && __out != stdout) fclose(__out);

                if(file == "stdout") __out = stdout;

                else if(!(__out = fopen(file.c_str(), "a")))
                {
                        fprintf(stderr, "library error: open(\"out\", %s)\n",
file.c_str());
                        __out = stdout;
                }
        }

        else fprintf(stderr, "library error: invalid argument to open: %s\n",
inout.c_str());
}

int __str_len(const string &s)
{
        return s.size();
}

string __str_substr(const string &s, const int i)
{
        if(i >= 0) return s.substr(i);
        else return s.substr(0, -i);
}

string __str_replace(const string orig, const string &search, const string &replace)
{
        string out(orig);
        pcrecpp::RE(search).GlobalReplace(replace, &out);
        return out;
}

string __str_match(const string orig, const string &search)
{
        string out;
        pcrecpp::RE("(" + search + ")").PartialMatch(orig, &out);
        return out;
}

int __str_index(const string &orig, string &search)
{
        string tmp = __str_match(orig, search);
        return ((tmp != "") ? orig.find(tmp) : -1);
}

int __str_less(const string &l, const string &r) { return (l < r) ? 1 : 0; }
int __str_greater(const string &l, const string &r) { return (l > r) ? 1 : 0; }
int __str_lessequal(const string &l, const string &r) { return (l <= r) ? 1 : 0; }
int __str_greaterequal(const string &l, const string &r) { return (l >= r) ? 1 : 0; }
int __str_equal(const string &l, const string &r) { return (l == r) ? 1 : 0; }
int __str_notequal(const string &l, const string &r) { return (l != r) ? 1 : 0; }

template <class key, class val>
map<key, val> __map_empty(map<key, val> &m) { m.clear(); return m; }

template <class key, class val>
int __map_equal(const map<key, val> &l, const map<key, val> &r) { return (l == r) ? 1 :
0; }

template <class key, class val>
int __map_notequal(const map<key, val> &l, const map<key, val> &r) { return (l != r) ? 1
: 0; }
```

```
string __str_concat(const string &l, string &r) { return l + r; }

template <class key, class val>
val __map_remove(map<key, val> &m, const key &k) { val v = m[k]; m.erase(k); return v; }

template <class key, class val>
int __map_exists(map<key, val> &m, const key &k)
{
        typename map<key, val>::iterator it = m.find(k);
        return (it != m.end());
}

template <class key, class val>
map<int, key> __map_keys(map<key, val> &m)
{
        int i = 0;
        map<int, key> k;
        for(typename map<key, val>::const_iterator it = m.begin(); it != m.end(); ++it)
        {
                k[i] = it->first;
                i++;
        }
        return k;
}

template <class key, class val>
map<int, val> __map_vals(map<key, val> &m)
{
        int i = 0;
        map<int, val> v;
        for(typename map<key, val>::const_iterator it = m.begin(); it != m.end(); ++it)
        {
                v[i] = it->second;
                i++;
        }
        return v;
}

template <class key, class val>
int __map_len(const map<key, val> &m)
{
        return m.size();
}

#endif
```