Project 4

Due Nov 2, 2016 11:59:59pm

Introduction

In this project, you will write a compiler for a programming language called Simpl, which is a simple imperative language with arithmetic, strings, and function calls. Your compiler will translate Simpl source code into RubeVM byte code (from project 3).

Project Structure

The project skeleton code is divided up into the following files:

`Makefile`	Makefile
`lexer.mll`	Simpl lexer
`parser.mly`	Simpl parser
`ast.mli`	Abstract syntax tree type
`disassembler.ml`	RubeVM bytecode disassembler
`main.ml`	The main compiler logic
`s{1--4}.si`	Small sample Simpl programs

You will only change `main.ml`; you should not edit any of the other files. The file `main.ml` includes code to run the parser and then compile the input file to `a.out`. Right now, the generated output file always contains code that prints `Fix me!`:

```
$ make
$ ./main.byte s1.si
$ rubevm a.out  # renamed main.byte from p3 to rubevm
Fix me!
$ ...
```

You'll need to modify the implementation of `compile_prog` in `main.ml` to perform actual compilation.

You can use your own `rubevm` from project 3, or you can run as

```
/afs/glue.umd.edu/class/fall2016/cmsc/430/0101/public/bin/rubevm
```

(So if you've already added that directory to your path, you can just run `rubevm`.)

In Simpl, the program executes by running a top-level expression. To make it easier to debug and grade your compiled programs, the programs you generate should take that expression, turn it into a string, and print it out; more details below.

Simpl Syntax

The formal syntax for Simpl programs is shown in Figure 1. A Simpl program P consists of a sequence of function definitions, each of which may take any number of arguments. To execute a program, we invoke its function `main`, which must take no arguments.

The body of a function consists of a single expression, which is evaluated and returned from the function call. Simpl expression include integers n and strings `"str"`. Local variables are identifiers *id*, which are made up of upper and lower case letters or underscore. Simpl includes read and writing local variables; conditionals and while loop; sequencing; standard binary operations; table reads and writes; and function calls. Note that there is no built-in mechanism to create an empty table. Instead, this is done with a special function call, similar to calling `malloc` in C.

P	::=	F^*	Simpl program
F	::=	def id $(id, \ldots, id)$ E end	Function definition
E	::=	v	Values
	|	id	Local variable read
	|	$id = E$	Local variable write
	|	`if` E `then` E `else` E `end`	Conditional
	|	`while` E `do` E `end`	While loop
	|	E; E	Sequencing
	|	$E \oplus E$	Binary operation
	|	$E[E]$	Table read
	|	$E[E] = E$	Table write
	|	$id(E, \ldots, E)$	Function call
v	::=	n	Integers
	|	`"`str`"`	Strings
$\oplus$	::=	`+` | `-` | `*` | `/` | `==` | `<` | `<=`	Binary operators

Figure 1: Rube syntax

```
type bop =
  | BPlus | BMinus | BTimes
  | BDiv | BEq | BLt | BLeq

type expr =
  | EInt of int
  | EString of string
  | ELocRd of string
  | ELocWr of string * expr
  | EIf of expr * expr * expr
  | EWhile of expr * expr
  | ESeq of expr * expr
  | EBinOp of expr * bop * expr
  | ETabRd of expr * expr
  | ETabWr of expr * expr * expr
  | ECall of string * (expr list)

(* name * arg list * body *)
type simpl_fn = { fn_name : string;
                  fn_args : string list;
                  fn_body : expr }

(* program is a list of functions *)
type simpl_prog = simpl_fn list
```

Figure 2: Abstract syntax tree for Simpl

Abstract syntax trees We've provided you with a parser that translates Simpl source code into an abstract syntax tree. Figure 2 shows the OCaml AST data types.

The first few lines define binary operators `bop`, which are used in expressions.

The type `expr` is for expressions. Expression `EInt n` represents the integer `n`, and `EString s` represents the string `s`. Expression `ELocRd s` represents (reading) the local variable `s`. Notice in our abstract syntax tree, we use strings for the names of local variables. Expression `ELocWr(s,e)` corresponds to `s=e`, where `s` is a local variable to be assigned the value of expression `e`. Expression `EIf(e1,e2,e3)` corresponds to `if e1 then e2 else e3 end`. Expression `EWhile(e1, e2)` executes body `e2` as long as guard `e1` is true, and the whole expression evaluates to 0. Expression `ESeq(e1,e2)` corresponds to `e1;e2`. Expression `EBinOp(e1,b,e2)` corresponds to `e1 b e2` with binary operator `b`. Expression `ETabRd(e1, e2)` corresponds to `e1[e2]`, and expression `ETabWr(e1, e2, e3)` corresponds to `e1[e2] = e3`. Finally, expression `ECall(s,el)` corresponds to calling function `s` with the arguments given in `el`. (The arguments are in the same order in the list as in the program text, and may be empty.)

A function `simpl_fn` is a record containing the function name, arguments, and body. Finally, a program `simpl_prog` is a list of functions.

$$\textsc{Int}\ \frac{}{P \vdash \langle A, H, n\rangle \to \langle A, H, \mathtt{n}\rangle} \qquad \textsc{Str}\ \frac{}{P \vdash \langle A, H, \texttt{"}str\texttt{"}\rangle \to \langle A, H, \text{“}\mathtt{str}\text{”}\rangle}$$

$$\textsc{Id}\ \frac{id \in dom(A)}{P \vdash \langle A, H, id\rangle \to \langle A, H, A(id)\rangle} \qquad \textsc{Id-W}\ \frac{P \vdash \langle A, H, E\rangle \to \langle A', H', v\rangle \qquad A'' = A'[id \mapsto v]}{P \vdash \langle A, H, id = E\rangle \to \langle A'', H', v\rangle}$$

$$\textsc{If-T}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, v_1\rangle \qquad v_1 \neq 0 \qquad P \vdash \langle A_1, H_1, E_2\rangle \to \langle A_2, H_2, v_2\rangle}{P \vdash \langle A, H, \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{end}\rangle \to \langle A_2, H_2, v_2\rangle}$$

$$\textsc{If-F}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, 0\rangle \qquad P \vdash \langle A_1, H_1, E_3\rangle \to \langle A_3, H_3, v_3\rangle}{P \vdash \langle A, H, \mathtt{if}\ E_1\ \mathtt{then}\ E_2\ \mathtt{else}\ E_3\ \mathtt{end}\rangle \to \langle A_3, H_3, v_3\rangle}$$

$$\textsc{While-T}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, v_1\rangle \qquad v_1 \neq 0 \qquad P \vdash \langle A_1, H_1, E_2; \mathtt{while}\ E_1\ \mathtt{do}\ E_2\ \mathtt{end}\rangle \to \langle A_2, H_2, v_2\rangle}{P \vdash \langle A, H, \mathtt{while}\ E_1\ \mathtt{do}\ E_2\ \mathtt{end}\rangle \to \langle A_2, H_2, v_2\rangle}$$

$$\textsc{While-F}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, 0\rangle}{P \vdash \langle A, H, \mathtt{while}\ E_1\ \mathtt{do}\ E_2\ \mathtt{end}\rangle \to \langle A_1, H_1, 0\rangle}$$

$$\textsc{Seq}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, v_1\rangle \qquad P \vdash \langle A_1, H_1, E_2\rangle \to \langle A_2, H_2, v_2\rangle}{P \vdash \langle A, H, (E_1;\ E_2)\rangle \to \langle A_2, H_2, v_2\rangle}$$

$$\textsc{BinOp}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, v_1\rangle \qquad P \vdash \langle A_1, H_1, E_2\rangle \to \langle A_2, H_2, v_2\rangle \qquad v = v_1 \oplus v_2}{P \vdash \langle A, H, E_1 \oplus E_2\rangle \to \langle A_2, H_2, v\rangle}$$

$$\textsc{TabRd}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, \ell\rangle \qquad P \vdash \langle A_1, H_1, E_2\rangle \to \langle A_2, H_2, v_2\rangle \qquad v = H_2(\ell)(v_2)}{P \vdash \langle A, H, E_1[E_2]\rangle \to \langle A_2, H_2, v\rangle}$$

$$\textsc{TabWr}\ \frac{P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, \ell\rangle \qquad P \vdash \langle A_1, H_1, E_2\rangle \to \langle A_2, H_2, v_2\rangle \qquad P \vdash \langle A_2, H_2, E_3\rangle \to \langle A_3, H_3, v_3\rangle \qquad H' = H_3[\ell(v_2) \mapsto v_3]}{P \vdash \langle A, H, E_1[E_2] = E_3\rangle \to \langle A_3, H', v_3\rangle}$$

$$\textsc{Call}\ \frac{\begin{array}{c}P \vdash \langle A, H, E_1\rangle \to \langle A_1, H_1, v_1\rangle \qquad \ldots \qquad P \vdash \langle A_{n-1}, H_{n-1}, E_n\rangle \to \langle A_n, H_n, v_n\rangle \\ P(id) = (\text{def } id\ (id_1, \ldots, id_n)\ E \text{ end}) \\ A' = [id_1 : v_1, \ldots, id_n : v_n] \qquad P \vdash \langle A', H_n, E\rangle \to \langle A'', H'', v\rangle\end{array}}{P \vdash \langle A, H, id(E_1, \ldots, E_n)\rangle \to \langle A_n, H'', v\rangle}$$

$$\textsc{Program}\ \frac{P = F^* \qquad P \vdash \langle \emptyset, \emptyset, \mathtt{main()}\rangle \to \langle A', H', v\rangle}{\vdash P \Rightarrow v}$$

Figure 3: Simpl Operational Semantics

Simpl Semantics

Figure 3 gives the formal, big-step operational semantics for evaluating Simpl expressions (we will discuss relating these rules to compilation next). Note that you are **not** implementing these rules directly. Rather, you are building a compiler such that the output program, when run on the Rube VM, will behave according to the rules.

These rules show reductions of the form $P \vdash \langle A, H, E\rangle \to \langle A', H', v\rangle$, meaning that in program P, and with local variables environment A and *heap* H, expression E reduces to the value v, producing a new local variable assignment A' and a new heap H'. As usual, we extend the set of values v with *locations* ℓ, which are pointers used for tables. The program P is there so we can look up functions. We've labeled the rules so we can refer to them in the discussion:

- The rules INT and STR say that an integer or string evaluates to the expected value, in any environment and heap, and returning the same environment and heap. In the syntax of Simpl, strings begin and end with double quotes `"`, and may not contain double quotes inside them.
- Like Ruby, a local variable can be created by writing to it. The rule ID says that the identifier *id* evaluates to whatever value it has in the environment A. If *id* is not bound in the environment, then this rule doesn't apply—and hence your compiled code would signal an error. Reading a local variable does not change the local variable environment or the heap.
- The rule ID-W says that to write to a local variable *id*, we evaluate the E to a value v, and we return a configuration with a new environment A'' that is the same as A', except now *id* is bound to v. The value v is returned by the assignment expression itself. As just mentioned, it is possible to create a new local variable by writing to it. (So, you'll need to do a little extra work to figure out what locals are used in a function body, hence what registers to allocate for those locals.)
- The rules IF-T and IF-F say that to evaluate an if-then-else expression, we evaluate the guard, and depending on whether it evaluates to a non-0 value or 0, we evaluate the then or else branch and return that. (Notice that *any* non-0 value, including all strings and all table pointers, are treated as "true.") Also notice the order of evaluation here: we evaluate the guard E_1, which produces a configuration $\langle A_1, H_1, v_1\rangle$, and then we evaluate the then or else branch with that local variable environment and heap.
- The rules WHILE-T and WHILE-F evaluate a while loop, using the same rule as IF-* that 0 is false and any other value is true. A while loop always evaluates to 0.
- The rule SEQ says that to evaluate E_1; E_2, we evaluate E_1 and then evaluate E_2, whose value we return. Note that in the syntax, semicolon is a *separator*, and does not occur after the last expression. Thus, for example, `1; 2` is an expression, but `1; 2;` is not (and will not parse). Notice again the order of evaluation between E_1 and E_2.
- The rule BINOP evaluates a binary operation by evaluating the left side, the right side, and then returning the result of applying the binary operation. We haven't formally listed the rules for what $v_1 \oplus v_2$ means, but they are as follows: `+`, `-`, `*`, `/`, `<`, and `<=` operate only on integers. It is an error to apply them if one of the arguments is not an integer. Operator `==` can apply to any operands, though it's guaranteed to return false if the two operands are not of the same type. `==` on integers and strings should do a "deep" comparison of those quantities, just like in RubeVM. `==` on tables should be a "shallow" comparison of the tables, returning true only if both sides point to the same table.
- The rule TABRD evaluates E_1, which must resolve to a location ℓ. It next evaluates E_2, which resolves to some value v_2. Then it returns the mapping for v_2 in the table at location ℓ. It is an error if E_1 does not resolve to a location or v_2 is not in the table at location ℓ.

- The rule TabWr is similar, except it updates the heap so that the table at location ℓ maps the key v_2 to the value v_3. It returns the value v_3.

- The rule Call describes a function call. We evaluate the arguments E_1 through E_n, in order from 1 to n, to produce values. (Notice here the "threading" of the location variable environment and heap through the evaluation of E_1 through E_n.) Next, we look up the function id in the program; it is an error if the function does not exist. The function must take the same number of arguments as the number supplied in the call; however, we will not test that you detect this error. (If you do check for it, it's best done at compile time in this language.)

  We create a new environment A' in which each of the formal arguments id_i is bound to the actual arguments v_i. (Don't worry about duplicate parameter names; we won't test that case.) We evaluate the body of the function in this new environment A', and whatever is returned is the value of the function invocation.

  Notice that Simpl has no nested scopes. Thus when you call a function, the environment A' you evaluate the function body in is not connected to the environment A from the caller. This makes these semantics simpler than a language with closures.

- Finally, rule Program explains how to evaluate a Simpl program. We simply evaluate a function call to `main` with no arguments, starting in the empty environment and the empty heap.

Errors

The rules above describe how to run a program that behaves correctly. They do not say what to do when there is an error. This is fairly typical of language definitions, which leave it up to the language implementor to decide what to do for errors. However, for grading purposes, we do need to specify some of the ways you should handle errors:

- If a program tries to read a table value that does not exist, it should print `halt: Key does not exist` and then exit immediately.

- For any error not on this list, your implementation may report the error in whatever way you prefer; we will not test these cases.

- Recall that we will not write any test cases that have the following two issues: (1) calling a function with the wrong number of arguments and (2) having duplicate parameter names.

For all error cases, you should generate a program that reports the error at run time, even if you could imagine detecting the error at compile time. For example, just by looking at it, we can see the Simpl expression `0+"foo"` will always fail at run-time. But for this project, the code `0+"foo"` should compile, and only when we run it will we get the error.

Note that the way we've set up this project, most errors can be left to RubeVM to signal.

Built-in functions

In addition to the core language, Simpl also includes several built-in functions. Your compiler should behave as if the built-in functions exist at the start of the program. Here are the functions:

- `print_string`(x) – Print string x. Returns x.

- `print_int`(x) – Print integer x. Returns x (the integer, not its string representation).

- `to_s`(x) – If x is a string, return it. If x is an integer, return the integer as a string.

- `to_i`(x) – If x is an integer, return it. If x is a string, convert the string to an integer and return it. If x contains characters other than digits and possibly an initial minus, this may behave arbitrarily.
- `concat`(x_1, x_2) – Return concatenation of strings x_1 and x_2
- `length`(x) – return length of string x
- `size`(x) – return number of key-value pairs in table x
- `mktab()` – return a new, empty table
- `is_i`(x) – return 1 if x is an integer and 0 otherwise
- `is_s`(x) – return 1 if x is a string and 0 otherwise
- `is_t`(x) – return 1 if x is a table and 0 otherwise

Notice these are suspiciously similar to the foreign functions supported in RubeVM, except for the last four. Thus, you probably don't need to do much work at all to support these. Also notice that we didn't include an `iter` function; why not?

As in Project 3, if the user defines their own function with the same name as a built-in function, the user-defined function should take precedence.

Compilation

As mentioned in the introduction, your compiled program should begin by calling `main()`, which will yield a value v. The return value should be an integer or string, which should then be printed to standard out. (We won't test the case where the return value is not an integer or a string.) Recall from project 3 that the interpreter already prints the returned integer or string to standard output, so you probably don't need to do much to achieve this behavior.

Here are some suggestions for doing the project:

- Notice that Simpl is pretty close in many ways to RubeVM. Thus, if you implement things in the natural way, the project will be a lot easier, i.e., use RubeVM integers, strings, and tables for Simpl integers, strings, and tables.
- You don't have to implement function calls right away, since the program always begins by calling `main`; thus you can just hard code that into your compiler.
- To implement local variables, you'll need to walk through a function's code and create a mapping from variables to RubeVM register numbers. Notice that the RubeVM semantics automagically implement the right behavior for variables (referring to a variable/register that has not been written is an error). To rephrase in a slightly different way: the generated RubeVM code will refer directly to registers, and so you won't actually manipulate local variable names at run time.
- The code for `mktab()` and `is_*(`x`)` will be the same for all programs. You can thus hard-code it into your compiler.

Academic Integrity

The Campus Senate has adopted a policy asking students to include the following statement on each assignment in every course: "I pledge on my honor that I have not given or received any unauthorized assistance on this assignment." Consequently your program is requested to contain this pledge in a comment near the top.

Please **carefully read** the academic honesty section of the course syllabus. **Any evidence** of impermissible cooperation on projects, use of disallowed materials or resources, or unauthorized use of computer accounts, **will be submitted** to the Student Honor Council, which could result in an XF for the course, or suspension or expulsion from the University. Be sure you understand what you are and what you are not permitted to do in regards to academic integrity when it comes to project assignments. These policies apply to all students, and the Student Honor Council does not consider lack of knowledge of the policies to be a defense for violating them. Full information is found in the course syllabus—please review it at this time.