IIT CS443: Compiler Construction

Project 0: IITRAN Interpreter

Prof. Stefan Muller

Out: Thursday, Aug. 22
Due: Tuesday, Sep. 3, 11:59pm CDT

Updated Aug. 28 (language spec clarifications)

This assignment contains 1 task(s) for a total of 15 points.

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- Download the starter code by cloning the Github repo for the assignment: `https://classroom.github.com/a/PmLB5yVX`. When you first go to the link, you'll be asked to create/join a team (if you're working by yourself, just create a team with just you). Github will create one repo per team.
- If you're not familiar with using Git, see the "Learn the Basics of Git in Under 10 Minutes" link under Resources on the course website. (I haven't verified that it takes under 10 minutes.)
- Submit your homework by pushing your changes to Github by the deadline (or the extended deadline if taking late days). You can, of course, push non-finished code to your repo (e.g., while collaborating). I'll grade the last commit before the deadline. If you push to the repo one or two days after the deadline, I'll consider that a submission using late days and grade the last submission from the last day.
- If you accidentally push to your repo after the deadline and didn't intend to take late days, email me ASAP. Otherwise, you do not need to let me know if you're using late days; I'll count them based on the date of your last submission.
- Compile (by running `make`) before submitting. **Submissions that don't compile will not get credit.**

Collaboration and Academic Honesty

You may work in groups of at most 2 on this project. Read the policy on the website and be sure you understand it.

1 OCaml

1.1 Set Up OCaml

The main purpose of this project is setting up OCaml (if you don't have it set up already; if you do, lucky you, skip to Section 2) and getting acquainted (or reacquainted) with it, so you can dive right in to the other

projects.

The instructions at `https://ocaml.org/docs/installing-ocaml` work well for Mac and Linux. The best way I've found to get OCaml running on Windows is to set up WSL (instructions are at `https://docs.microsoft.com/en-us/windows/wsl/install`) and then follow the Linux instructions. The instructions above also describe how to set up support in your favorite editor (as long as it's Emacs, Vim or VSCode.)

Finally, **you will need to install Dune**, a build system for OCaml (instructions also at the link above, under Install Platform Tools).

Please start early so I and other students can help you out if you run into problems. Post on Discord with questions/trouble/tips.

1.2 OCaml Basics

A Quick Guide to the OCaml Module System. Like many languages, OCaml uses modules for abstraction and namespaces. Module names in OCaml begin with an uppercase letter (this isn't just a convention; it's actually enforced by the language). You can access a definition (function or variable) `bar` from a module `Foo` using the syntax `Foo.bar`. If you don't want to keep typing `Foo.`, there are two options:

- (Preferred) Bind it to a shorter name using, e.g.,

  ```
  module F = Foo
  ```

  Afterward, you can refer to `F.bar`.

- Open the module using `open Foo`, after which you can just refer to `bar`. This is fine to do sometimes, but be careful that you're not introducing conflicting names (e.g., if the file you're working in already contains a definition of something called `bar`; if it does, the newer definition will shadow the older one and the compiler won't warn you about this, potentially causing confusing errors!)

When working in multiple files, each file becomes a module. The name of the module is the name of the file (without .ml) with the first letter capitalized. For example, in `iit_interp.ml`, you can use the definitions in `varmap.ml` by `Varmap.`... The project is already set up so that Dune, the build system, will find and link the files correctly.

The OCaml Standard Library. OCaml comes with an extensive standard library, and there are many, many 3rd party packages and libraries. That's one of the reasons we're using it. Keep the API reference (`https://v2.ocaml.org/api/index.html`) handy while you're working on projects for the course. That page lists the API by module. As with any module, refer to standard library functions by, e.g., `List.map`. It's a good idea to spend a bit of time looking through the standard library just to get a sense of what's there, so you don't spend a lot of time re-implementing something that's just in the library. The tasks in this project will get you familiar with a couple of the libraries you'll be using a lot in the course.

2 MiniIITRAN Language Specification

In this project, you'll be working with the language MiniIITRAN, a well-behaved, strongly typed subset of IITRAN. Here, we give a specification of this language.

2.1 Syntax

	digit	::=	$0-9$
	alpha	::=	$a-z, A-Z$
	alphanum	::=	*alpha* | *digit* | _
Identifiers	*id*	::=	*alpha* *alphanum**
Ident.Lists	*idlist*	::=	*id* | *id*, *idlist*
Numbers	*num*	::=	$-?$ *digit*+
Types	*typ*	::=	INTEGER | CHARACTER | LOGICAL
Constants	*c*	::=	*alpha* | *num*
Binary Operators	*bop*	::=	$+ \mid - \mid * \mid / \mid$ AND | OR $\mid < \mid \leq \mid > \mid \geq \mid \# \mid =$
Unary Operators	*unop*	::=	$\sim \mid$ NOT | CHAR | LG | INT
Expressions	*e*	::=	*c* | *id* | *e bop e* | $e <- e$ | *unop e*
Statements	*s*	::=	*e* | STOP | DO s^* END | IF *e s* | IF *e s* ELSE *s* | WHILE *e s*
Declaration	*d*	::=	*typ* *idlist*
Program	*p*	::=	d^* s^*

Operator Precedence and Associativity.

Operator(s)	Precedence	Associativity
All unary operators	6	—
*, /	5	Left
+, -	4	Left
Comparison operators	3	Left
AND	2	Left
OR	1	Left
$<-$	0	Right

(Higher numbers indicate that operators have higher precedence, i.e., "bind tighter"). All operators are left-associative except assignment. So, $x <- y <- 2 + 5$ should parse as $x <- (y <- (2 + 5))$.

Comments. Comments start with `$` and go to the end of the line.

2.2 Semantics

Constants Numeric constants are specified as integers, possibly preceded by a $-$ sign. Character constants are specified as `'c'`. There is no direct way to specify logical constants, but logical constants can be introduced using LG n, which becomes logical true if $n \geq 0$ and logical false if $n < 0$.

Variables Variables are declared with declarations, which declare one or more variables with a given type. A variable may not be used without a declaration (this is a change from real IITRAN, but makes compilation easier). Later declarations of variables take precedence over older declarations (the older declarations become useless, as declarations must precede all statements.) Variable names are case-insensitive. Variables are initialized to 0 (for logical variables, this means false, and for character variables, it means ASCII 0).

Binary Operations e_1 *bop* e_2
Operations are evaluated left-to-right (with short-circuiting as described below under "logical operators"): e_1 is evaluated before e_2. This mostly matters when expressions contain assignments. For example, if `B` is initially 0, then `(B <- 1) + (B <- B + 1)` should evaluate to 5.

Individual categories of binary operators are described below.

Arithmetic Operators (+, -, *, /) e_1 *bop* e_2
Types: e_1 : INTEGER, e_2 : INTEGER, result: INTEGER
Note: Integer overflows and division by zero result in runtime errors.

Comparison Operators ($<, \leq, >, \geq, \#, =$) e_1 *bop* e_2
Types: e_1 : INTEGER, e_2 : INTEGER, result: LOGICAL
Note: # is "not equal to."

Logical Operators (AND, OR) e_1 *bop* e_2
Types: e_1 : LOGICAL, e_2 : LOGICAL, result: LOGICAL
Important Note: Both AND and OR should *short circuit*, that is: if e_1 evaluates to false, e_2 should not be evaluated at all in e_1 AND e_2 (and similar for e_1 OR e_2 if e_1 evaluates to true). In MiniIITRAN, this is mainly relevant if e_2 might divide by zero. For example, `1 > 0 OR 1 / 0 > 0` should never raise a divide-by-zero exception.

Assignment $x <- e$
Types: x and e must have the same type. The result is of that same type.
Result: Compute e, assign its value to x and return the value.
Note: $e_1 <- e_2$ where e_1 is some expression other than a variable is a runtime error. For the purposes of this class, it is unspecified whether or not such assignments are syntactically valid (so your parser may accept them or not).

Integer Negation $\sim e$
Types: e : INTEGER, result: INTEGER
Result: $0 - e$

Logical Negation NOT e
Types: e : LOGICAL, result: LOGICAL

Type Conversions INT, LG, CHAR.
The result type is as specified (INTEGER, LOGICAL, CHARACTER, respectively). The argument can have any type.
LG n becomes logical true if $n > 0$ and logical false if $n \leq 0$.
LG c, where c is a character is always logical true unless c is ASCII 0.
INT c returns the ASCII code of c.
INT l of a logical constant l returns 0 for false and 1 for true.
CHAR n returns the character with ASCII code n.
CHAR l returns the character with ASCII code 0 or 1.

Expression statements
Types: The expression must be well-typed with any type.
Result: The expression is computed. Any assignments are performed, but otherwise the value is ignored.

If statements IF e s_1 ELSE s_2
Types: e : LOGICAL.
Result: If e evaluates to true, performs s_1, otherwise s_2. If s_2 is absent, control continues to the next statement.

While statements WHILE e s
Types: e : LOGICAL.
Result: If e evaluates to true, performs s and then evaluates e again and loops. When e evaluates to false, control continues to the next statement.

Stop *Result:* Ends execution of the program.

Do DO $s_1 s_2 \ldots s_n$ END
Types: All substatements must be well-typed. *Result:* Substatements are executed in order.

Program results. The program returns the value in the designated variable RESULT when execution ends, either by control reaching the end of the programming or encountering a STOP. By convention, RESULT should be declared to be an integer (programs should return integer values, and your compiler may assume this is the case).

3 MiniIITRAN AST

The AST definition is in `iit_ast.ml`. You can refer to any definition in this file using `IITRAN.Ast`. (This is an exception to the usual way of turning file names into module names, but it's a convention we'll use throughout the course.) However, we've already `open`ed the module for you at the top of the file you'll be working in. There is one important thing to note about the AST definition. The definitions of expressions (`exp`) and statements (`stmt`) carry additional information. An `exp` is a record containing three fields:

- `edesc` is the algebraic data type for expressions. This looks just like what we've seen in class; its type is now `exp_` instead of `exp`. Note that subexpressions (e.g., of `EBinop`) have type `exp`, so the subexpressions also carry the extra information. The types `exp` and `exp_` are mutually recursive.
- `eloc`, of type `loc`, is the location in the source code of the expression. A `loc` is a pair of two `Lexing.position`s, giving the start position and end position of the expression. You can look up the definition of `Lexing.position` in the standard library API, and an example of how to use it at the bottom of `ast.ml`.
- `einfo`, which gives additional information of type `'a`. The types `'a exp_` and `'a exp` are parameterized by this type. In general, this can be anything you want: the type `'a exp` carries information of type `'a`. We'll use two types: `unit exp`, which we'll call `p_exp`, carries no information, and will be the type for expressions that come right out of the parser, and `typ exp`, which we'll call `t_exp`, tags each expression with each type. The type checker will turn a `p_exp` into a `t_exp`, and then you'll compile `t_exp`s in the next project (the type information will come in handy!)

You can access fields of a record using `record.field` so, for example, instead of pattern matching on an expression `e`, you'd pattern match on `e.edesc` (and you can use `e.eloc` and `e.einfo` to get the location and info). You can build a record with the syntax `{ field1 = val1; field2 = val2; ... }`. We've also provided functions `mk_exp : p_exp_ -> loc -> p_exp` and `mk_t_exp : 'a exp_ -> loc -> 'a -> 'a exp`. For this assignment, you're only really working with `exp_s` (and shouldn't need to build them), so don't worry too much about this now.

The definitions of `'a stmt_`, `'a stmt`, `p_stmt` and `t_stmt` are similar, except that statements don't have types, so there's no info field, just `sdesc` and `sloc`. The types for statements still have the `'a` parameter because they need to pass it on to expressions that are part of the statements.

4 Codebase and Convenience Functions

4.1 Printing

The file `iit_print.ml` contains pretty-printing functions for IITRAN, which you might want to use during debugging. You can access these functions using `IITRAN.Print.f`, where `f` is the name of the function or definition. The interface for the `Print` module is in `iit_print.mli`. Some of these functions return a string, which you can use, e.g., with `Printf.printf`. Many take a `Format.formatter` as their first argument and return unit. Use `Format.std_formatter` to print to standard output. (You can also use other formatters to print to standard error, build strings, etc.; see the documentation for the `Format` library for information.)

Please comment out or remove any code that prints to the console before submitting your solution.

4.2 Varmap

The module `Varmap` provides a type (`'a Varmap.t`) and operations on finite maps whose keys are strings and whose values are of type `'a`, which is the way of writing a type variable of polymorphic types in OCaml[1]. For example, an `int Varmap.t` maps strings to integers. The operations provided by `Varmap` are listed at `https://v2.ocaml.org/api/Map.S.html`.

5 Task: MiniIITRAN Interpreter

As an introduction to OCaml (and a refresher on building interpreters), you'll write part of the MiniIITRAN interpreter, in `iit_interp.ml`. The interpreter represents MiniIITRAN values as integers, together with their MiniIITRAN type (e.g., `(1, TInteger)` represents the integer 1, but `(1, TLogical)` represents the logical value true. The interpreter also maintains an environment of type `env = (int * typ) Varmap.t`. Its keys are variables and the values are the current value (int, typ pair) of each variable.

Task 1 (Programming, 15 points).

Fill in the remaining cases in `interp_exp`, which interprets an expression to produce the resulting value and environment (remember, assignment is an expression, so evaluating expressions can change the environment) as a triple of type `int * typ * env`.

- You don't need to handle the case where the left-hand side of an assignment is some expression other than a variable; this is not valid MiniIITRAN. This case is already filled in to raise a runtime error. You can leave that there.
- You don't need to worry about what to do in cases that would be a runtime type error (e.g., if you're assigning an integer to a variable that was already declared as some other type, or reading or assigning to an undeclared variable). Your interpreter may do anything you want in such cases. Type errors will already have been ruled out by the type checker, which runs before the interpreter, so they will never come up. Static typing is great!
- Your interpreter **should** properly handle the short-circuiting behavior described in the language specification above for AND and OR operations (e.g., when evaluating `1 > 0 OR 1 / 0 > 0`, do not evaluate `1 / 0`).
- You need not catch runtime errors like integer overflows, divide-by-zero, etc. If you just use the OCaml integer operations, they'll raise exceptions in these cases and it's fine to just propagate that exception.

6 Testing

When you're done, run `make` (in the top level directory of the repo, not in `src`) to compile your interpreter. This produces a binary `iit`, which you can run on the command line like so:

```
./iit file.iit
```

where `file.iit` is an IITRAN source code file. This parses the source code, type-checks it[2] and runs the interpreter on it.

We've given you a number of test files in `tests`. Each program begins with a comment indicating the expected result (recall that this is the value of variable RESULT), if the program is syntactically correct. Files of the form `tests/synerrN.iit` should not parse and should result in a syntax error (for now, this is just a test of our parser, but on the next project, it will be a test of *your* parser, which you'll be writing). You can run all of the tests by running (on the command line, still in the top level of the repo) `make test`.

[1] We could call it `Stringmap` but you'll be using it a lot in this course to make maps of variables.

[2] The type-checker is defined in `iit_typecheck.ml`. You're welcome to take a look at it, though you shouldn't need to do so, and definitely shouldn't need to modify it.