IIT CS443: Compiler Construction

Project 2: MiniIITRAN to LLVM Compiler

Prof. Stefan Muller

Out: Tuesday, Sep. 17
Due: Thursday, Sep. 26, 11:59pm CDT

Total: 50 points

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- Download the starter code by cloning the Github repo for the assignment. 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.
- Note that the Github repo (and thus, by default, teams) will be the same for the remaining project. Changes to teams will be possible in extenuating circumstances after this, but do think carefully about your partner (or lack thereof) for this assignment.
- 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 MiniIITRAN Language Specification

Recall the language specification of MiniIITRAN from last time, reproduced below.

1.1 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.

1.2 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).

2 LLVM AST

The definition of LLVM ASTs is in `llvm/llvm_ast.ml`. From your code, you can reference this module as `LLVM.Ast`. The type `var` of variables is no longer just `string` but uses constructors to distinguish between Local and Global variables. The only global variables will be function names, which we don't even have in this project, so you will only have to worry about local variables for now. Much later in the course, we'll need to swap out a different type for variables, so the types of values and instructions are parameterized by the type `'var` of variables. For this (and the next few) project(s), we will only instantiate it with `var`. We'll conflate IITRAN variables and LLVM locals, so you can turn an IITRAN variable `s` into `L.Local s` (don't worry about the `%` at the beginning of LLVM locals: the LLVM pretty-printer will add this). You can assume (probably incorrectly) that the variable names generated by `L.new_temp` (see below) are distinct from any variables in the program[1].

This file contains the full AST for the subset of LLVM we will eventually be using, but you don't need all of it for this assignment. As usual, the type of types is `typ`. The only type you'll need for this project is `TInteger n` ($i n$). We've defined binary arithmetic operators `bop` and comparison operators `cmp`. A `value` is either a constant (`Const`) integer or a variable (`Var`). The instructions you'll need are `ILabel`, `ISet`, `IBinop`, `ICmp`, `IBr`, `ICondBr`, and `IRet`. The return instruction `IRet` can return a value or not (not returning a value corresponds to a C function returning `void`.) Since we have no functions in MiniIITRAN, `IRet` will only be used to end the program and return the value of the variable RESULT.

Our LLVM definition also has functions, but you won't need these for now. For this project, an LLVM program is a list of `inst`s.

3 Module Structure and Helpful Functions

The (only) file you'll be editing is `iitllvm.ml`. The definitions of the MiniIITRAN language from Project 1 are in the folder `iitran` and are collected in the module `IITRAN`. So, for example, you can refer to the IITRAN AST module as `IITRAN.Ast`. Same for the LLVM definitions in the folder `llvm`, which are collected in the module `LLVM`. At the top of `iitllvm.ml`, I've `open`ed `IITRAN.AST`, so you can refer to IITRAN AST definitions without any module name, and rebound `LLVM.Ast` to `L`, so you can refer to, e.g., `L.TInteger`. We can't open both because they have some of the same names, so this is the next most convenient thing.

Speaking of convenience, there are a few other definitions in both folders you might want to know about. You'll probably use the functions `new_temp ()` and `new_label ()` a lot. Both take a unit (`()`) as argument and return a new temporary LLVM variable and new label, respectively.

There's an LLVM interpreter (`LLVM.Interp`), which you can call with your programs to test them out. You can also run this from the command line on the LLVM programs output by your compiler (see the section on Testing), but maybe you will want to call it from inside your compiler for debugging purposes (please make sure to not do this in the code you hand in though!) The interface for the interpreter code is in `llvm/llvm_interp.mli`. There are also pretty-printers for both MiniIITRAN (`IITRAN.Print`) and LLVM (`LLVM.Print`). The printing functions take a `Format.formatter` as their first argument. Use `Format.std_formatter` to print to standard output. The printing functionality for IITRAN is the same as from Project 1. The interface for the LLVM pretty-printer is in `llvm/llvm_print.mli`.

4 Task: MiniIITRAN Compiler

Your task is to implement the function `compile_stmt : IITRAN.Ast.t_stmt -> LLVM.Ast.inst list` which compiles a MiniIITRAN statement to a list of LLVM instructions. You'll almost certainly want to

[1] Just don't use variable names starting with "temp" in your test cases!

define some (possibly mutually recursive) helper functions, like `compile_exp`. My solution uses the following, all mutually recursive (in addition to `compile_stmt`):

- `compile_binop (dest: L.var) (b: bop) (e1: t_exp) (e2: t_exp) : L.inst list`
- `compile_unop (dest: L.var) (u: unop) (e: t_exp) : L.inst list`
- `compile_branch_exp (e: t_exp) (tlabel: label) (flabel: label) : L.inst list`
- `compile_exp (dest: L.var) (e: t_exp) : L.inst list`

but you're free to do this differently if you want. Just don't change the type signature of `compile_stmt`, because the top-level compile function, `compile_prog`, uses that to compile the entire IITRAN program into one big "main" function by just calling `compile_stmt` on each statement in the body of the program.

Some (I hope) useful notes and/or hints:

- Don't forget that the AND and OR operators should short-circuit, wherever they're used in a program. In class, we saw a few ways to do this. Pick one (or your own solution, as long as it works!)
- Because LLVM is typed, your generated code will need to have types. We'll use the type `i64` (bound as `itype` for convenience) for integers and characters and `i1` (`btype`) for logicals/Booleans. Usually, the types will be clear based on the operations, e.g., arithmetic operations will produce and consume integers. There are one or two cases where you'll need to do something different based on the type of expressions. Because type checking has already run, you're using `t_exp`s, and so recall that you can get the type of an expression `e` with `e.einfo`. We've provided the function `compile_typ`, which compiles MiniIITRAN types to LLVM types, for convenience.
- As with Project 0, you don't need to worry about cases that would be type errors (e.g., if a character is used as an operand to +), because the program has already passed type-checking.
- Your generated LLVM code *does not* need to be in SSA form; you're free to assign to variables multiple times. We run an algorithm to convert the code to SSA (if you're interested, it's in `llvm/ssa.ml`) before outputting it.

5 Testing

Compile your code using `make` (in the top level of the source tree). This will produce the binary `./main`, which you can use as follows to compile test programs:

`./main tests/<file>.iit`

This will parse and compile the file, and then type-check the resulting LLVM code. By default, it will output human-readable LLVM code in `tests/<file>.ll`. If you have the LLVM toolchain installed, you can interpret or compile this file (the easiest thing would be to interpret it using `lli tests/<file>.ll`. Note that the program doesn't print anything, but just returns the value as the program's exit code. In Bash on Linux, you can print the exit code of the last command using `echo $?` (you can Google for how to do it on other platforms if you don't know.)

Even easier (especially if you don't have LLVM) is to use the CS443 LLVM interpreter, which is already built in to `main`. To do this, run

`./main -interpllvm tests/<file>.iit`

After type-checking the LLVM output by your compiler, this will run the interpreter on it and print the result.

For this project, you aren't required to write new test cases (since you already wrote IITRAN test cases for the last project), though you are certainly encouraged to write new test cases to check extra cases in your compiler (and are welcome to share them with the class if you do).

If you're finding the output LLVM code hard to read, you can turn off the conversion to SSA using the `-nossa` flag to `./main`, so the output will be exactly what was produced by your compiler (wrapped in an LLVM main function). Of course, this means you won't be able to run LLVM tools on the output code, but the built-in LLVM interpreter will still work if you also use `-interpllvm`.