IIT CS443: Compiler Construction

Project 3: MiniC to LLVM Compiler

Prof. Stefan Muller

Out: Thursday, Sep. 26
Due: Thursday, Oct. 10, 11:59pm CDT

Total: 50 points (Programming: 45 points, Test case: 5 points)

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- The starter code for Project 3 has been distributed through a pull request to your Project 2 GitHub repo. Merge this pull request into your main branch to start working. (This shouldn't cause merge conflicts, but if it does and you aren't sure how to handle them, ask.)
- **Important new submission instructions!** When you want to submit, commit the latest changes to your GitHub repo **with a commit message that clearly indicates this is your Project 3 submission** (e.g., "Project 3 Submission" would be a good commit message). This is so I know which commit to grade. If you resubmit later, just use a similar commit message. I'll grade the last commit that clearly indicates it's a Project 3 submission (and count your late days based on the time stamp of this 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 MiniC Language Specification

The MiniC language spec is below. I suggest reading it carefully even (perhaps especially) if you're comfortable with C, because there are some important differences. Please feel free to ask any clarification questions at all on Discord to make sure you understand the spec before starting the compiler.

1.1 Syntax

The abstract syntax of Mini-C is below.

Types	*typ*	::=	void | bool | char | int | *typ*[] | *id* | *typ*((*typ id*,)*)
Constants	*c*	::=	$'alpha'$ | *num*
Binary Operators	*bop*	::=	+ | − | ∗ | / | AND | OR |<|≤|>|≥|! =|=
Unary Operators	*unop*	::=	− |!
L − values	*lhs*	::=	*id* | *id*[*e*] | *id.id*
Exp.Lists	*elist*	::=	ϵ | *e*(, *e*)∗
Expressions	*e*	::=	*c* | *id* | *e bop e* | *lhs* = *e* | new(*typ*) | *unop e* | *e*(*elist*) | *e*[*e*] | *e.id* | (*typ*)*e*
Statements	*s*	::=	*e*; | *typ id* [= *e*] | {s^*} | if (*e*) *s* else *s* | for (*e*; *e*; *e*) *s*
		|	break; | continue; | return *e*; | return;
Declaration	*d*	::=	*typ id*((*typ id*)*)*s* | struct *id* {(*typ id*)*};
Program	*p*	::=	d^*

Comments. Comments start with `//` and go to the end of the line, or block comments can go between `/*` and `*/`.

1.2 Semantics

Types

Base Types are int, bool, int and void.

Array Types *typ*[]
A **pointer to** an array of *typ* values (of any length). As discussed in class, all arrays are treated as pointers and are allocated on the heap—there is no way to allocate a local array.

Structure Types
Structure types may be declared by name (see Declarations). A structure has a specified set of fields of specified types. Fields can be retrieved using $s.f$ and assigned using $s.f = e$. As with arrays, a value of named structure type is a **pointer** to a location in the heap containing the values for the fields (it is unspecified how exactly these values are stored in memory, so you can implement that how you want, within reason.)

Function Types $t(t_1\ x_1, \ldots, t_n\ x_n)$
In MiniC, this represents a **pointer** to a function that accepts arguments $x_1, \ldots, x_n$ of types $t_1, \ldots, t_n$ and returns a value of type t.

Expressions

Expressions in general work like MiniIITRAN expressions, with the following additions/changes:

Assignment *lhs* = *e*
lhs refers to anything that can appear on the left hand side of an assignment. In MiniC, this can be a variable, an array access of a variable, or a field access of a variable. The type of the *lhs* must match the type of *e*. Assignments perform the assignment and return the assigned value.

Allocation new(*typ*)
Allocates memory **in the heap** for an object of type *typ* (which should be either a named structure type or an array type, in which case it must include the length of the array, e.g. new(int[10])) and returns a pointer to it.

Function calls $e(e_1, \ldots, e_n)$.
e is evaluated to a function pointer, then e_1 through e_n are evaluated (left to right) to values, at which point the function is called. This returns the return value of the expression.
Types: e is a *pointer* to a function that accepts arguments of types $t_1, \ldots, t_n$ and returns type t. Expression e_i has type t_i and the entire call expression has type t.

Array access $e_1[e_2]$.
e_1 is evaluated to an array (a pointer of type t^*) and e_2 is evaluated to an index. The value at index e_2 of the array is returned (if out of bounds, this has undefined behavior).
Types: $e_1 : t^*$, $e_2 :$ int, Result: t.

Field access $e.id$.
e is evaluated to a pointer to a structure; the value of field id is returned.
Types: e is a named structure type with a field id.

Type casts $(t)e$.
e is evaluated and cast to type t. Characters are interpreted as ASCII values. For conversions out of bool, true is interpreted as 1 and false as 0. For conversions to bool, 0 is interpreted as false and all other values are interpreted as true. Casts to and from void are not allowed.

Statements

Like in MiniIITRAN, expressions can be statements. A block of statements is written $\{s_1; \ldots s_n\}$. The block may be empty. The form for if statements requires both an if and else branch. (An if without an else is valid syntax; this will parse to an if statement with an else branch of {}). New or changed statements are described below.

If
Unlike in C, the expressions are required to be of type bool.

For
For loops for $(e_1; e_2; e_3)$ s evaluate the expression e_1 once at the start of the loop, then evaluate the test e_2 (which must be of type bool). If the test is true, the body s is evaluated followed by e_3, before returning to the test. In the abstract syntax, e_1 must be an expression. In the parser, this is allowed to be a variable declaration, which is desugared into a declaration followed by the loop.

Break and continue
break exits the current for loop and continues executing the next statement. continue jumps to the end of the body of the current for loop, and executes the "next" expression (e_3 above). **Note:** Both statements are invalid outside a while loop—this should be a compile error!

Return
The statement return e evaluates e (whose type must be the return type of the current function) and exits the function immediately, returning the value. The statement return (with no expression) exits a function with return type void.

Variable declarations *typ id* [= e]. Declares a variable *id* of type *typ*. If an expression (which must be of type *typ*) is given, the variable is initialized to its value. Otherwise, the variable is uninitialized. Use of an uninitialized variable results in undefined behavior (this may be a runtime error or return an arbitrary value). Variable declarations may be mixed with statements in function bodies, but may not appear at the top level (because MiniC does not support global variables).

1.2.1 Declarations

Function Declarations *typ id*((*typ id*)*)*s*.
Declares a function with return type *typ* and body *s*. This binds a global variable of function pointer type with name *id*. Function declarations may only appear at top level (i.e., not inside functions).

Structure Type Declarations struct *id* {(*typ id*)*}
Declares a structure type *id* with the given fields. The types of the fields may include *id*, allowing recursive structures. Stucture declarations may only appear at top level.

1.3 Differences with C

This is an incomplete list of differences between MiniC and C, other than that only a subset of features are supported.

- `bool` is a completely supported type (as in some versions of C). Like in MiniIITRAN, it has no literals, but you can cast 0 and 1 to `bool`.
- Arithmetic operators operate only on integers. You cannot, e.g., subtract `'z' - 'a'` (but you can cast other values to integers).
- The expressions in `if` and `while` must be of type `bool`. e.g., no `while 1...`
- There is no explicit pointer type, though arrays and structs are pointers (in this way, MiniC is a bit more like OO languages such as Java). They are allocated using `new` rather than by explicit memory allocation. For this reason, the operators `*` and `&` don't exist, and there's no pointer arithmetic (this is a feature, not a bug).
- There is also no `null` pointer, but you can cast 0 to a struct or array[1] if you want one.
- There are no global variables (other than function names).

2 Module Structure and Helpful Functions

The definition of LLVM ASTs is in `llvm/llvm_ast.ml`, and is the same as you had for Project 2. The definition of MiniC ASTs is in `c/c_ast.ml`, and the module is `C.Ast`. C is a considerably more involved language to parse than IITRAN, so we are using an off-the-shelf C parser called FrontC, modified for MiniC syntax. An advantage of this is that the parser will accept very close to actual C syntax, including some features that aren't directly supported by MiniC (e.g., the `++` operator and `while` loops). These are "desugared" into MiniC ASTs by the desugaring pass in `c/c_desugar.ml` (you don't need to look at or understand this file). The compiler will raise an error if you use features of C that cannot be desugared into MiniC.

FrontC also comes with a pretty-printer for C, which I've heavily modified. You can access the printing functions in the module `Cprint`, whose interface is given in `c/frontc/cprint.mli`. You don't *need* to use this, but may find it helpful in debugging (as always, please remove or comment out any printing/debugging code before you submit). Otherwise, you don't need to touch FrontC at all.

The only file you'll be editing is `cllvm.ml`. As in Project 2, this file `opens` `C.Ast` and binds `LLVM.Ast` as `L`. The LLVM interpreter can be run from your code or the command line, as in Project 2, and the LLVM pretty-printer interface is the same.

[1] Yes, I know this is awful. But is it really more awful than having a null pointer in the first place? Something to think about.

C AST

The C AST looks a lot like the IITRAN AST, including the use of mutual recursion between, e.g., `'a exp_` and `'a exp`, where the latter is a structure with fields `edesc`, which gives the underlying type of expression, `eloc` which gives the location in the source file (locations are now just a pair of a file name and a line number, because that's what FrontC does), and `einfo` which stores extra information of type `'a` (unit for `p_exp`s and `typ` for `t_exp`s).

The main new AST nodes are the "Lvalues" `LHVar`, `LHArr`, and `LHField` (constructors of type `'a lhs`), which are the expressions that can appear on the left-hand side of an assignment. MiniC allows these to be only

1. Variables (x)
2. Array accesses where the array expression is a variable ($x[e]$)
3. Field accesses where the structure expression is a variable ($x.s$)

The `LHField` constructor also has an extra component of type `'a` which, in a `t_lhs`, is the type of x (i.e., the type of the structure, e.g., `TStruct s`).

The functions `lhs_to_exp` and `exp_to_lhs` convert back and forth between expressions and lvalues, where `lhs_to_exp` always succeeds and `exp_to_lhs` returns an option which is `None` if the expression is not a valid lvalue.

Variable expressions (`EVar`) also now take a *pair* of a `var`, which is just an alias of `string`, and a `var_scope`, which is either `Local` or `Global`.

Other helpful functions

The table below lists some other functions/definitions you may find helpful.

Function	Location	Type	Description
`get_field_i_and_typ`	`c_typecheck.ml`	`ctx -> string -> string -> (int * typ) option`	The first string argument is the name of a structure type, the second is a field. Returns the index of that field within the struct and its type, or `None` if the field doesn't exist.
`get_field_i`	`c_typecheck.ml`	`ctx -> string -> string -> int option`	Same as `get_field_i_and_typ`, but returns only the index.
`word_size`	`cllvm.ml`	`int`	The size of a word in bytes. Is an alias for `Config.word_size`.
`malloc`	`cllvm.ml`	`L.var`	A pointer to the malloc function.
`L.sizeof`	`llvm_ast.ml`	`L.typdefs -> L.typ -> int`	Returns the size of an LLVM type in memory *in words* (not bytes)

Arguments to the compile functions

The `compile_*` functions all take two extra arguments: `ctx`, of type `ctx`, and `tds` of type `L.typdefs`. Mainly, these contain information about the types and structures defined in the C code. See above ("Other helpful functions") for how you can use values of these two types.

3 Programming Task: MiniC Compiler (45 points)

Your task is to implement the function `compile_stmt` which compiles a MiniC statement to a list of LLVM instructions. In addition to the `ctx` and `tds` arguments described above, `compile_stmt` takes two arguments `break_lbl` and `cont_lbl` of type `L.label option`. These are the LLVM code labels of the end and "next" expression of the innermost currently active `for` loop, if there is one (i.e., they are the jump targets of `break` and `continue`, respectively). If we are not in a `for` loop, both arguments should be `None`.

You will, of course, need to define several other (mutually recursive) functions to compile expressions, etc. Much of this code is the same as for IITRAN, with one notable difference: while both integers and characters compiled to `i64` for IITRAN, we will now use `i32` for `int`, `i8` for `char` and `i1` for `bool`. This behavior is implemented in `compile_typ` and `itype`, `ctype`, and `btype` are defined for your convenience. This also means you need to pay more attention to types when you compile casts.

I suggest reusing your code from Project 2, modifying as necessary for the AST constructor names and the difference above. If you prefer, you're welcome to use my Project 2 solutions here instead, once they're posted.

Obviously, you also need to handle the new forms of expression and statement that didn't exist in MiniIITRAN (arrays, structs, `new`, functions, `break`, `continue`, etc.). That's essentially all there is to the programming task of this project.

Some (I hope) useful notes and/or hints (mostly copied from Project 2):

- I've already done some of the work of compiling casts for you, in the function `compile_cast`.
- Also see the function `compile_var`, which handles the difference between local and global variables.
- Don't forget that the AND and OR operators should short-circuit, wherever they're used in a program.
- 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 MiniC types to LLVM types, for convenience.
- Note, once again, that array and struct types compile to **a pointer to** either the array element or the LLVM struct type. For example, the MiniC type `int[]` compiles to `L.TPointer (L.TInteger 32)` and, if `my_struct` is declared as a struct type, the type `my_struct` compiles to `L.TPointer (L.TStruct "my_struct")`. The function `compile_typ` (which you will find helpful!) implements this correctly. When you call `L.sizeof`, be careful what you call it on! (That is, if you call it on the result of calling `compile_typ` on an array or struct type, you'll just get the size of a pointer, which is likely not what you wanted.)
- As with Projects 1 and 2, 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.

4 Test Cases (5 points)

Each group should write (at least) one MiniC program that is not substantially the same as the test cases I've given you or others written by other students/groups (this is good incentive to write your tests early and post them before other groups!) You're encouraged to try and find corner cases and explore code paths other tests might have missed. Don't be too adversarial though; reasonable student solutions should pass your test.

Some other rules (the test cases I use for grading will follow these rules as well):

- Your test case should be syntactically valid and desugar to MiniC (you are welcome to use features that are not directly in the MiniC syntax, like while loops). That is, your code should not raise a syntax or "unsupported" error.
- Your test case should also be type-correct. That is, it should not raise a type error.
- The intended behavior of the test case *may* be to raise an error in your compiler code *during compilation*, or to raise a runtime error.
- Your test case should not trigger any unspecified behavior (e.g., reading an uninitialized variable, or an out-of-bounds array access).

Post your test case as a new thread on the "Project 3 Test Cases" discussion board on Blackboard. You can (and should!) test your compiler on other students' test cases. I may do so as well during grading. Feel free to ask clarification questions, note issues, etc., as replies in the threads created by other students.

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 proj3-tests/<file>.c
```

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 LLVM, you can interpret or compile this file (the easiest thing would be to interpret it using `lli tests/<file>.ll`. Remember 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>.c
```

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

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. 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`.

Running `make test` will run all of the tests listed in the `p3tests` file. If you add another test case, it won't automatically be run by `make test` unless you add it to `p3tests` following the existing examples. **Note:** This test script is at best barely production-ready. Feel free to modify it how you want (let me know if you make useful modifications I can push out in future assignments!). It may not be a substitute for running tests individually, as described above (in particular, if you get an error like "Expected: 42, Got:", this probably means that the test raised an exception; you should re-run that test manually to debug it.)