Introduction to Compilers and Translation Engineering

ECE 46800/57300/59500, Fall 2023

Step 6, Pointers and Arrays

last update: 31 August at 11:36pm

Table of contents

Step 6: Pointers and Arrays

Suggested date of completion: Nov 17th 2023, 5PM ET

Due: Dec 1st 2023, 5 pm ET

Setting up your repository

Set up your GitHub classroom repository for step 6 using the link on Brightspace.

Set up your project development and testing environment as directed in the project environment documentations.

Then clone your repository as directed in the cloning and submitting instructions.

Background

In this step, we will add support in our language for several new features:

1. Pointers
2. Arrays
3. Memory allocation on the heap

Important note This step builds on *Step 4*, not *Step 5*. We *do not* expect or require register allocation in this step.

Pointers

The primary background you need for adding pointers to the language is the lecture notes and videos on Pointers. The changes you will need to make to your compiler include:

1. Modifying the grammar to support pointer dereference expressions (`* expr`) and address-of expressions (`& lval`). **We have done this step for you.** Note a few key changes we made:
    1. The left-hand side of assignment statements is no longer just a variable, but can be any lval.
    2. lvals can be variables, array expressions (see below), or deref expressions. The grammar for these is a little bit weird to accommodate ANTLR's rules about left recursion (array expressions are split out, instead of being part of the lval rule).
    3. There are two new types of expressions, dereferences and address-of.
2. Building your AST for dereference expressions and address of expressions. We have defined two new AST nodes (`PtrDerefNode` and `AddrOfNode` , respectively) to let you build your AST. **You will have to do this by adding actions in your grammar file.**
3. Adding support for pointer types to your compiler. **We have done this for you** (by abusing Java enums a little bit, allowing a `Type` to be a `PTR` type that includes an extra `Type` field specifying what it's a pointer to.)
4. Performing code generation for these new types of expressions. You will need to make sure to track types properly for your generated code objects, so pay careful attention to how types are managed in the associated AST nodes. **You will have to do this.**

An important note about pointer arithmetic: uC supports pointer arithmetic, as the dereference operator can be applied to arbitrary expressions. In a language like C, pointer arithmetic is tricky. An expression like:

```
*(p + 4)
```

gets converted by the compiler into:

```
*(p + sizeof(p) * 4)
```

In other words, integer values that show up in expressions are treated as "x elements of whatever the pointer is pointing to". This is fussy to deal with, and is part and parcel of the type promotion/conversion that C does that we *do not* do.

Hence, in uC, the semantics of pointer arithmetic is simpler. In expressions, addresses are treated as simple `int` s, and arithmetic is simple integer arithmetic. Those `int` s are treated as pointers in a dereference expression. Thus, if `p` is an integer pointer pointing to address `0x400` , then:

```
*(p + 4)
```

will dereference address `0x404` . (In C, that expression would dereference `0x410` .)

Arrays

Our language does *not* support array types (meaning you cannot declare a variable like `int x[10]`). However, our language *does* support array expressions for indexing into arrays pointed to by pointers:

```
 int * p;
p = ...
...
x = p[3];
```

Array expressions are lvals, and can hence appear as the left-hand-side of assignment expressions:

```
p[4] = 7;
```

(Note that our grammar uses a different rule for array expressions than it does for lvals, due to ANTLR's rules about left-recursion, as discussed above.)

In the lecture notes, we discussed how to implement array expressions using *syntactic sugar*: translating array expressions into corresponding pointer expressions. The notes talk about two ways you can do this: an AST rewrite pass that rewrites array expressions, or by implementing pointer arithmetic semantics during code generation. In this step, we will explore a third option: *desugaring during semantic actions*.

An array expression desugars into an equivalent pointer expression: `p[3]` is the same as `*(p + 3 * 4)` .

> Note that we multiply 3 by 4 in the expression. This is because of uC's pointer arithmetic semantics, as described above. You will always multiple by 4 because all three "base" types in uC (ints, floats, pointers) are 4 bytes.

When you encounter an array expression in your code (see the rules for `array_expr`), you can directly construct the AST subtree that represents your desired pointer expression. In other words, build an `PtrDerefNode` and an `ExpressionNode` that together implement the AST structure you would get *after* desugaring.

> Hint: Think about what a generic pointer expression for `lval[expr]` would look like: `*(lval + 4 * expr)` and build exactly that tree.

A fun consequence of syntactic sugar is that if you impement the desugaring right (i.e., if you build the AST right), you should not need to do any extra work for generating code for arrays – your code generation for pointers will suffice.

Memory allocation

Finally, this step introduces the use of the *heap* to our language. Programs can *dynamically allocate* memory not as global variables or local variables, but on the heap. See the lecture notes on malloc and free.

You will need to add code generation for malloc and free. This involves a few changes to the grammar:

1. We added support for *void* functions (functions that do not return a value), as `free` is such a function. You will need to alter your code generation for functions to correctly handle scenarios where a function has a void return type.
2. We added `malloc` as a new expression that can appear in programs. Malloc looks like a function that takes a single expression as an argument that specifies the number of bytes to allocate, and returns the address of the beginning of that allocation. We added `MallocNode` as a new AST Node corresponding to this expression.
3. We added `free` as a new statement that can appear in programs. Free looks like a (void) function that takes a single expresison as an argument that contains the starting address of the allocation to free. Be careful with `free` : `free(x)` does not say free the allocation at the address of `x` , it says free the allocation at the *address stored in* `x` (in other words, `x` is a pointer).

To support malloc and free (i.e., memory allocation and de-allocation, respectively), we augmented RiscSim with magic instructions:

1. `MALLOC dest, src` allocates a number of bytes in the heap equal to the value stored in `src` , and assigns the *address* of the allocation in `dest` .
2. `FREE src` frees the allocation whose address is stored in `src` .

These magic instructions are backed by a memory allocator provided by RiscSim. RiscSim will throw errors if you attempt to free an allocation that has not previously been allocated.

573 only Malloc's type signature is tricky. In C, malloc has the following type signature: `void * malloc(size_t)` – it returns a *void pointer* – a pointer whose target type is unspecified. We do not have void pointers in our language but malloc needs to be typechecked carefully, as both of these invocations need to typecheck:

```
 int * p;
int * * q;
...
p = malloc(10);
q = malloc(20);
```

In other words, when generating code for malloc, we do not necessarily know what type the returned address holds. To manage this, we have added a special type called `INFER` . Malloc returns a type called `INFER` , and you need to modify your type checking routines to make sure that `INFER` types propagate correctly through expressions, and do not cause type checking to fail.

What you need to do

We want you to add support for pointers, arrays, and memory allocation, as described above.

Running your code

To run your code, you can use the RISC simulator we have provided for you. When using the container environment, this is present in ~/RiscSim. When using the ecegrid environment script, the path to the driver script for it is set in $RISCSIM.

You can run an assembly file by running:

```
> python3 RiscSim/driver.py [assembly file]
```

Sample inputs and outputs

The inputs and outputs we will test your program on can be found in the directories `tests` and `outputs` . Note the tests are ordered by "difficulty." We *strongly* suggest adding enough functionality to pass a given test before moving on to the next test.

What you need to submit

- All of the necessary code for building your compiler.
- A Makefile with the following targets:
  1. `compiler` : this target will build your compiler
  2. `clean` : this target will remove any intermediate files that were created to build the compiler
- A shell script (this *must* be written in bash, which is located at `/bin/bash` on the ecegrid machines) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the compiler and second, the filename where you want to put the compiler's output. You can assume that we will have run `make clean; make compiler` before running this script, and that we will invoke the script from the root directory of your compiler.

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation.

See the submission instructions document for instructions on how to submit. You should tag your step 6 submission as `submission`

Most course announcements will be posted on Piazza and/or the corresponding webpage. Only a few critical announcements will be posted on main page.