Function Calls and Return Statements

Lecture #09

Charles Averill

Introduction to Compiler Design
The University of Texas at Dallas

Spring 2023

The Current State of the Compiler

Previously, we added in support for one function declaration into our language:

```
void main() {
    int x;
    x = 5;
    print x;
}
```

Now, let's put them to use.

The Goal

We'd like to be able to

1. Define multiple functions
2. Call functions
3. Return values from functions
4. Pass a single (temporarily unusable) argument to functions

The LLVM

This is mostly recap, as we've generated code like this before. We're just generalizing it now.

```
define dso_local i32 @fred() {
    ...
    ; ret <return type> <constant | register>
    ret i32 0
}

define dso_local void @main() {
    ...
; call <return type> <parameter types> @<function name>(<parameters>)
    call i32 () @fred()
    ...
    ret void
}
```

Declaring Multiple Functions

We've actually already almost accomplished this in the previous lecture thanks to the generality of our code generator updates.

Just one small change left: I defined a `translate_reinit` function that resets the set of free allocated registers, loaded registers, and next possible register numbers. This function gets called inside of our function parsing loop in `generate_llvm`.

Calling Functions

We're going to temporarily require that function calls always have one argument that will go unused. Eventually we'll work on passing this argument into the body of the called function, and allowing for more parameters per function.

We run into an interesting problem when parsing function calls, can anyone think of why?

Calling Functions

The issue is that function names are identifiers just like variable names. Therefore, everywhere we dealt with parsing variable name instances (declarations, assignments, rvalues), we now have to check ahead for a left parenthesis to see if we're actually looking at a function call.

Function calls are actually expressions, not statements. So we've added a custom parser inside of `parsing/expression.py`.

We're going to require that an identifier has been matched by the time we're inside of `function_call_expression`, but sometimes it's more advantageous to not consume the token, so we have an override variable to bypass this requirement should we need it.

Return Statements

Parsing return statements is pretty straightforward. We've added a new token for the `return` keyword. We'll ensure that void functions don't try to return expressions, and that non-void functions have to return expressions.

Return Value Code Generation

Generating return code is pretty easy. We have to do some type checking to determine whether we're returning `void` or some data.

It's important to realize that `ret` calls should consume a virtual register index, clang will whine at you if you don't increment the counter for those statements. But you can't assign return statements to a virtual register. E.g. `%9 = ret void` is an illegal statement, but if you don't increment the VR count to 9 here, you'll get a misnumbering error from clang. I can't find any documentation on this, weird!

Function Call Code Generation

Our function call code is pretty familiar as well. Do some type checking to make sure the passed identifier is a function, make sure that you call it with the right type, and assign it to a virtual register.

Bug Fixes

There are a lot of bug fixes in this update:

- Changed all `LLVM_LOADED_REGISTERS.append` calls to calls to new function `add_loaded_register` that's defined in `translate.py`. This prevents loaded registers from one function interfering with the list of loaded registers in another
- Added a recursive call to `root.middle` in `determine_binary_expression_stack_allocation`. For the past few updates, we haven't been allocating space for code in those branches! This is a failure on my part to write proper unit tests
- Added a forgotten case in `Type.llvm_repr` to handle functions that have non-void return types
- Added a new `factorial` unit test to handle some more complex programs. New class requirement: I want each of you to write and push 3 complex programs using the current language features

Optional Homework

This week's optional homework is one of my favorite features in Purple: N-base integer literals.

In most large languages, you can define literals in base 10 (standard notation), base 16 (`0xABCD`), base 2 (`0b1001`), and base 8 (`0o1234`).

In Purple, I added the ability to define N-base literals, for $N <= 36$. The format specifies that the number literal is followed by a new token, #, and then a single character, [1-9 |A-Z].

For example, `LMNOP#R (base 27) = 11610727 (base 10)`.

This isn't a standard C feature, but is a lot of fun and it's a challenging parsing problem. Can anyone think of why?

Example N-Base Literal Program

```
/**
 * @file base_test.prp
 * @author Charles Averill
 * @brief Test number literal declarations
 * @date 27-Sep-2022
 */

int main(void) {
    int F00F;
    F00F  = F00F#G;
    print F00F;   // 61455
    print 1001#2; // 9
    print 0b1001; // 9
    print 1234#8; // 668
    print 0o1234; // 668
    print f00F#G; // 61455
    print 0xf00f; // 61455
    print 1234#5; // 194
}
```