ASSIGNMENT 4

CS 473: Compiler Design / Fall 2024

Description

In this assignment, your compiler will traverse the AST from previous assignments and translate it to two Intermediate Representation (IR) forms for a stack machine: First a tree, and then a list. Your submission will pass the list IR to the provided interpreter, which can print and execute your programs. You should use the lexical parser that you implemented in Assignment 1, the syntactic parser that you implemented for Assignment 2, and the AST and semantic analysis that you implemented for Assignment 3.

The Bluejay Language

The Bluejay language is defined in a separate document.

Aves Intermediate Representation

The provided IR, called Aves (the name for scientific class of all birds), is implemented in files `ir.h` and `ir.c`. It is a stack-based IR, in which all operands are taken from the stack, and all results are left on the stack for the next operands. Some IR operations use immediate values to specify exactly what to do. Below is a list of all possible operations, and how they manipulate the stack (S denotes "the rest of the stack"):

Operation	Immediates	Stack before	Stack after	Description
nop		S	S	No-operation, does perform any computation, does not modify the stack
iconst	int	S	<int> S	Pushes the integer specified as an immediate
sconst	string	S	<address> S	Pushes the string specified as an immediate, leaves its address on top of the stack.
ops		<int2> <int1> S	<int3> S	Performs a binary operation on the two values on top of the stack, leaves the result on top of the stack <int3> = <int1> OP <int2>
ops		<int1> S	<int2> S	Performs a unary operation on the value on top of the stack, leaves the result on top of the stack <int2> = OP <int1>

intrinsic	enum intrinsic	<arg> S	S	Performs the intrinsic operation specified in the immediate. Takes one argument from the top of the stack
reserve	int size char * name char * value	S	S	Reserves memory for a global variable. Leaves the stack unchanged. The immediate values specify the name, the size, and the initial value of the global variable
read	char * name	S	<int> S	Reads the value of the global variable specified in the immediate, leaves it on top of the stack. Reading a string leaves its address on top of the stack.
write	char * name	<val> S	S	Writes the value on top of the stack to the global variable specified in the immediate
arg_local_read	int	S	<int> S	Reads the value of a local variable or argument specified by its index as an immediate, leaves it on top of the stack
arg_local_write	int	<val> S	S	Writes the value on top of the stack to the local variable or argument specified by its index as an immediate
label	ir_label *	S	S	Adds a label, specified in the immediate, that other IR instructions can jump to. Leaves the stack unchanged
jump	ir_label *	S	S	Jumps to the label specified in the immediate. Leaves the stack unchanged
branch	ir_label *	<int> S	S	Jumps to the label specified in the immediate if the value on top of the stack is zero
call	ir_label * int	<argN> … <arg2> <arg1> <int1> S	<int2> ret	Calls the function specified by the label in the immediates, consuming N+1 elements from the stack where N is the number of arguments provided to the instruction. Leaves a return integer on top of the stack.
ret		<int> S	S	Returns from the given function with the value on top of the stack. It is not possible to return from top-level code.
pop		<int> S	S	Discards the value on top of the stack.

Operations

Aves supports binary and unary operations. Both types of operations leave the result on top of the stack. Binary operations take two operands from the stack, and unary operations take their single argument from the stack. All operations are signed unless told otherwise. Overflow behavior is undefined.

Binary operations

Name	Stack before	Stack after	Notes
ir_add	<S> <A> <B>	<S> <A+B>	
ir_sub	<S> <A> <B>	<S> <A-B>	
ir_mul	<S> <A> <B>	<S> <A*B>	
ir_div	<S> <A> <B>	<S> <A/B>	
ir_mod	<S> <A> <B>	<S> <A%B>	
ir_bor	<S> <A> <B>	<S> <A|B>	Bitwise OR
ir_band	<S> <A> <B>	<S> <A&B>	Bitwise AND
ir_xor	<S> <A> <B>	<S> <A^B>	
ir_or	<S> <A> <B>	<S> <A||B>	Logical OR
ir_and	<S> <A> <B>	<S> <A&&B>	Logical AND
ir_eq	<S> <A> <B>	<S> <A==B>	1 if A has the same value as B, 0 otherwise
ir_lt	<S> <A> <B>	<S> <A<B>	1 if A is less than B, 0 otherwise
ir_gt	<S> <A> <B>	<S> <A>B>	1 if A is greater than B, 0 otherwise

Unary operations

Name	Stack before	Stack after	Notes
ir_not	<S> <A>	<S> <!A>	0 if A is not zero, 1 if A is zero

Reserving memory

Aves has a dedicated operation `reserve` that reserves memory. It takes the size of memory to reserve, the initial value of the reserved memory, and a name to refer to that memory later as a variable.

- Reserve an integer: All integers are initialized as zero. To reserve an integer:
    1. size must be 4
    2. val must be NULL
- Reserve a string:
    1. size must be the number of characters in the string, including the NULL terminator
    2. val contains the initial contents of the string

Intrinsics

Aves supports 3 types of intrinsic (built-in) functions:

Name	Stack before	Stack after	Notes
intrinsic_exit	<S> <int>		Exits with the exit code on top of the stack
intrinsic_print_int	<S> <int>	<S>	Prints the int on top of the stack
Intrinsic_print_string	<S> <address>	<S>	Prints the zero-terminated string which address is on top of the stack

Calling functions

Function arguments are passed via the stack. The first argument on the stack should be an arbitrary number (to reserve stack space for the return value). Then, the arguments must be on the stack, in the same order in which the function declares them. Finally, a CALL instruction specifies how many arguments to pass to the function, and which function to call. Upon return from a CALL instruction, the return is left on top of the stack. In the IR, all functions return an integer.

Inside a function, each argument can be accessed by their index. The first argument is at index 0, the second argument is at index 1, etc. Reading an argument pushes it to the top of the stack. The return instruction returns the value on top of the stack, and stops executing the current function.

Example:

Bluejay	**Function IR**	**Call IR**

`int add3(int a, int b, int c)`
`{`
`  return a+b+c;`
`}`
 	
`print_int(add3(400,70,3));`
	`Function("add3", 0);`
	`ArgLocal(0);`
	`ArgLocal(1);`
	`Op(ir_add);`
	`ArgLocal(2);`
	`Op(ir_add);`
	`Return();`
		`Iconst(<any number>);`
		`Iconst(400);`
		`Iconst(70);`
		`Iconst(3);`
		`Call("add3",3);`
		`Intrinsic(intrinsic_print_int);`

Tree IR

Each IR node has three pointers you can use to translate the AST into a tree form: `tree_ir_1`, `tree_ir_2`, and `tree_ir_3`. For instance, when translating an AST that adds two numbers together, you can use `tree_ir_1` for the left-hand side and `tree_ir_2` for the right-hand side:

AST	`BinOpNode(plus_op, <left>, <right>);`

Tree IR	`ir_node * ir = Ops(ir_add);`
	`ir->tree_ir_1 = /*translate AST <left> to IR */`
	`ir->tree_ir_1 = /*translate AST <right> to IR */`

List IR

Each IR node has a `next` pointer that you can use to turn a tree representation into a list representation. The interpreter provided will expect a list representation. You can turn the example above into a list representation as follows

List IR	

```
ir_node * left  = /*turn ir->tree_ir_1 into a list*/
ir_node * right = /*turn ir->tree_ir_2 into a list*/
// We want to generate the list [left] -> [right] -> Op(ir_add)
// [left] and [right] may be more than one ir instruction
findLast(left)->next = right;  // Generates [left] -> [right]
findLast(right)->next = ir; // Generates [right] -> Op(ir_add)
// We return the list we created above
// starting with the first instruction on the left-hand side
return left;
```

Entry Point

You will reuse files `lexer.lex`, `parser.y` , `semantic_analysis_(symbols|types).[ch]`, and `frame.[ch]` from Assignment 3.

You will modify file `ast_to_ir.c` with a traversal of the AST that transforms it into a Tree IR. You will modify file `ir_tree_to_list.c` with a traversal of the Tree IR that transforms it into a List IR. You can also modify files `transform.c` and `main.c` as you see fit.

You may need to modify your semantic analyses from previous assignments to fully support instrinsics and default implicits.

Due Date and Resubmission Policy

This assignment is due on **November 16th 2024** (Saturday) at **5pm CST**. There is no late policy.

The code and date used for your submission is defined by the last commit to your Git repository.

To resubmit this assignment, your **original grade** (as defined by the autograder) should be **equal to or higher than 30%** for undergraduate students. You can resubmit your assignment until **November 23rd 2024** (following Saturday) at **5pm CST**. Together with your resubmission, you will have to submit a written description of what you changed from the original submission (on Gradescope).

Bonus Points

This assignment has a total of **20% bonus points**.

You can earn 10% extra points by using Piazza as described in the syllabus. Your posts should be public, tagged with the `assignment4` label, and non-anonymous to the instructors to count towards the bonus. You can claim bonus points through **a Gradescope quiz**.

You can earn 10% extra points by fixing all memory leaks in your program. Each passing test without memory leaks is worth 1% bonus points.

Submission and Grading

This assignment is submitted through Github, and has an automatic grade component of 100%. You can check your current grade at any point by submitting your code and checking the autograder. The automatic grade is determined by 10 tests, that will check if your submission outputs the expected result. Each test is worth 10%.

Graduate students

On top of the behavior described above, graduate students have to add support for reading/writing arrays . Graduate students need to design new Aves IR instructions to read/write arrays, but do not have to implement them. They should modify the utilities that print an IR program to also print the new array IR operations, and use that to demonstrate the correctness of their compiler.

I will consider assigning **bonus points** to graduate students that implement the new array operations on the Aves interpreter, on a case-by-case basis. Please contact Prof Pina if you want to do this.

Each regular test is only worth 8% for graduate students. The remaining 20% will be graded by providing a 5 minute video showing how you added support for arrays and answering the following questions:

1. How does your compiler reserve room for arrays as global variables?
2. How does your compiler read from arrays?
3. How does your compiler write to arrays?
4. Show one limitation of your array implementation.

You can record such a video using Zoom, which you may already have installed to attend lectures remotely. Simply start a meeting (without any other participants), share your screen, and start recording. Note that Zoom requires some time to process your video after you record it, so plan accordingly. Extension requests to upload videos after the due time and date because Zoom is still processing them will be denied.

The maximum length for the video is 5 minutes, instructors will stop watching at the 5 minute mark (nothing past that point in the video will be graded). This video should be a screencast of your IDE open

on the code submitted, and you should highlight the code. Note that longer videos are not better videos, and you should record a video as short as needed to show all the expressions and answer the questions above.

The final grade for the assignment will be the grade of the original submission, for assignments without a resubmission; or the average between the original grade and the resubmission grade, for assignments with a resubmission. The grade of the original submission includes any bonus points.

Graduate student video resubmission

This assignment requires graduate students to provide full array support to Bluejay, which should build on top of the previous assignments. As such, graduate students can resubmit the graduate portion of all previous assignments with the original submission of Assignment 4. The grade of the graduate portion of each previous assignment (1, 2, and 3) will then be the average of the original grade and the resubmission.

Errors and Omissions

If you find an error or an omission, please post it on Piazza as soon as you find it.

Hardcoding and Academic Integrity

Any hardcoding will result in a 0% grade. Hardcoding is when you submit code that detects which test is being run, and simply outputs the expected result. For instance, detecting that test 22 is running, and replacing the usual execution of your submission with `printf("expected result").`

The academic integrity policy described in the syllabus applies to this assignment. You are responsible for writing all the code that you submit. We will use an automatic tool that detects plagiarism on all submitted code, and we will investigate all instances where plagiarism is more than likely.

Please refer to the syllabus for the full academic integrity policy.