CS-1319-1
Monsoon 2023
Assignment 4

Group 7
Saptarishi Dhanuka
Shivam Kedia

PLDI

Assignment 4

The Translator

Note: We have experimented quite a bit with design choices in the translator. Consequently our code may contain some dead code. Our PDF only mentions the important aspects of our design choices.

Globals

1. `global_symtab`: Pointer to the global symbol table data structure.
2. `current_symtab`: Pointer to the current symbol table (could be that of a function's or main(), helps keep track of where we are.)
3. `quad_array`: A global array that stores the quads as they are emitted.

Structures

Symbol Table

1. `symtab`: A dynamic array
   - size; n_elems; is_global; name; parent: `metadata` for the dynamic array
   - symboltable; pointers to all the entries (basically an array of pointers, each elements is a pointer to a symtab_entry)
2. `symtab_entry`:
   - name: string, name of the "symbol"
   - type: string, because there could be multiple derivatives of primitive types
   - size: int
   - initial_val: string because the type wasn't fixed and our `union` wasn't the best
   - nested table: pointer to another symtab
   - isptr which lets us know if it's a ptr

TAC

1. `quad`: A generalised structure where some values may be null depending on the kind of instruction it contains.

- opcode: an instance of an `enum` where we defined all the opcodes based on our convenience
- result, arg1, arg2: parameters for each instruction. These have been deliberately made `char *`s to enable uniformity and easy printing

2. `quad_array`: A simple, static, global array of pointers to the `quad` struct, of a fixed size `#defined` as `QUAD_ARRAY_LEN`
3. `list`:
   - the vanilla linked lists from Data Structures 1.0 containing an integer which would index into the quad array.
   - these indexed quads are the ones that need to backpatched at some point
4. `opcodetype`:
   - an enum of all the opcodes in the struct where the same character "operand" like a '*' may have multiple contexts and to enable ease of use has multiple enumerations according the context. For e.g op_MULT and op_DEREF.

Parsing and Semantic Actions:

1. `expression`:
   - loc: location of the related symbol table entry
   - truelist, falselist, nextlist for conditionals and jumps, etc.
   - arraybase for keeping track of the base address of the array
   - array lenght
   - size of each array element
   -
2. `statement`:
   - `nextlist:` Contains a pointer to the next list (implemented as a linked list)
3. functions (`parameter lists`):
   - linked list to store the symbol table entries of the arguments (which are just identifiers) and their types (these are `_arg_expr_list`)

Functions

For the `symtab`:

1. Functions to make the dynamic array:
   - new_symtab: constructor
   - resize_symtab: makes the array dunamic
   - insert_entry, search_symtab, update_entry: insert, search, update

- free_symtab: destructor

For the `quad`

Broke all the quad related functionality into 3 types of quads (comments in the `.c` file further mention which types of instructions have been assigned to which type of quad.):

New Quads are of the following types:

- `binary`: any quad which could be broken down `arg2` operating on `arg2` by means of `opcode` leading to `result`:
  1. binary `op`: y = a ± b etc.
  2. `res`(y) = `op`(call) `arg1`(proc) `arg2`(N_params)
  3. indexd copy: `result` = `arg1`[`arg2`], where `op` = `op_INDEX`
  4. indexd result: `result`[`arg1`] = `arg2`, where `op` = `op_INDEX_res`
- `unary`: any quad which could be broken down as `opcode` operating on `arg1` leading to `result`:
  1. copy: `result` = `arg1`, `op` is copy itself
  2. unary: `result` = `op1` `arg1`
  3. conditional jump: `arg`(condition) `op`(goto) `result`(label)
  4. pointer/addr assign: `op`(op_res_STAR or op_res_AND) `result` = `arg1`
- `instr`: any quad which could be broken down as `opcode` leading to `result`:
  1. unconditional jump: `op`(goto) `result`(label)
  2. param: `op`(param) `result`(arg)
  3. return: `op`(return) `result`(return val)

Print Quad: Based on the different types of instructions, it hardcodes how to print a quad. Segregating into the above 3 types makes it easy to organise.

Garbage collection has been on a best-effort basis in the semantic actions.

Functions (parameter lists)

- Simple linked list functions to create, append nodes to a list.
- Using those functions, create, merge lists and count elements in a list.

Other misc

- makelist, mergelist and backpatch were following the descriptions given in the slides

- print symtba with the fields mentioned. We have printed temps also we were told print whole symtab. We could have easily not printed them by simply checking the category for “temp”
- Augmented with function guard grammar rule for scoping related issues
- for a lot of the grammar rules I translated the slides into C code and augmented similarly
- A lot of explanation is given in the comments of the code
- We changed the lexer to have the attributes of the tokens like intval and loc
- int2bool for handling implicit boolean expressions in if statements and for loops

Citation: Referred to this repo to get an idea what exactly to do in this huge assignment: https://github.com/Fronsto/Compiler-for-NanoC/tree/main