CS316 Compilers Lab

Prof. Uday Khedker, Spring 2024

~~Area~~ Group 51

Govind Kumar (210050058)
Krishna N Agaram (210051003)

May 15, 2024

Abstract

We detail here our implementations of some basic language functionality added over the required functionality of the sclp compiler. To be precise, we explain our implementation of anywhere-declaration of variables (incl. declaring variables in nested scopes), the use of function calls in expressions, and finally the implementation of arrays. We describe extensions to implementations of pointers and structures that we have thought about. We conclude by explaining how our program generator that we use for testing and timing analysis works.

Contents

1 Anywhere-declaration of variables and nested scoping

The problem of declaring variables within the full function's scope is easy to do - we assign offsets (extending the size of the stack) just after the new declaration is processed.

To support nested scoping, as is standard, a stack of symtabs is maintained during parsing. Checking for variable names is done backwards starting at the top of the stack. Once a scope is left, the top of the symtab stack is popped (the symtab is persistent, only pointers to symtabs are stored on the symtab stack).

To allocate offsets for variables in nested scopes, we make use of the following fact: once a scope is left, the space used for its variables may be re-used. With this in mind, we do the following during parsing:

- On entering a scope, we design a control stack for it by maintaining two numbers, `size` and `cap`.
- When variables are declared, the `size` increases to accommodate the variables

```
size += new_var->get_size();
```

- When a scope ends, update `cap` as

```
cap = max(cap, size);
```

  Also, update the parent's `cap` as

```
parent->cap = max(parent->cap, parent->size + cap);
```

Essentially, we use the tightest stack size we can to accommodate all nested scopes, re-using stack space as and when scopes are closed and opened.

2 Usage of function calls in expressions

The key detail in expression evaluation is that of temporaries. Temporary variables are used to evaluate parts of expressions. When using a function call in an expression, we need to make sure that the function does not overwrite registers that store intermediate expression computations, or overwrites them after saving them elsewhere, and upon returning restores their values.

Our implementation saves registers that have already been allocated for the evaluation of the expression on the stack before the parameters for the function call (the reason for putting it before is so that the function call offsets remain the same as before - in particular, it is a caller-saved kind of operation). Once the call has returned, the values are popped back from the stack into the registers, yielding the abstraction of having not being touched.

We find the registers that need to be saved by maintaining at all times a list of registers in use (which is anyways done) and examining that list at the right time. An important point to notice here is that examining the list before the TAC statements that push the arguments of the function to the stack is sub-optimal; some registers of that list were used to compute expressions needed for a/some argument(s). All these registers will be freed after the push statements; those that remain will be used after the call. Our implementation picks on this idea, saving (and restoring) exactly those registers that are used after the push statements and before the call statement. Every register saved will thus be used after the call, and vice versa.

Note: (Further optimization) sclp does not differentiate caller and callee saved registers. In our implementation it is almost immediate to do so, since every call to the register allocator takes as argument a bit-vector of allowed register indices. One could then optimize by saving only those used registers that are caller-saved before each function call. Of course, this involves the additional saving required by each function to save callee-saved registers before using them, and restore them before returning (one can optimize leaf functions by using caller-saved registers instead - no saves required - since there will be no further calls from here).

3 Arrays and pointers

The non-trivial elements of the implementation of arrays and pointers are (a) the declaration and access processing, and (b) the dereferencing required while assigning a value to an array element or the element pointed to by a pointer. We address these issues for the case of arrays. We have not implemented pointers, but we describe how we may do so.

3.1 Declaration processing

We implement a `Type` class with the following attributes:

```
class Type {
    int info; // size (of array) or -1 (base type)
    Type *basetype; // the basetype of this object
};
```

An array `int a[10][20]` is then said to have type `Type(10, new Type(20, INT))`. To construct the type from the declaration, we must process the last index of the array first. This prompts the grammar statements (key-value syntax used on argument names for clarity)

```
array := name dims {
    $$ = new Array_Ast(name_ast=$1, type=$2);
}
dims := [ int_const ] {
    $$ = new Type (info=$2, base=nullptr); // we backpatch the base type later
}   |   [ int_const ] dims {
    $$ = new Type(info=$2, base=$4);
}
/* backpatch */
var_decl_stmt := basetype var_decl_list semicolon {
    for ( auto it: *($2) ) {
        it->type.set_base_type($1); /* sets base type recursively */
        curr_symtab->add_var(it);
    }
}
```

3.2 Dereferencing

We first need to generate code to compute offsets; we do this via a simple aggregating function that takes as argument the name of the array and a list of `Expr_Ast`'s (corresponding to the offsets) and uses the general rule - $am_2m_3 + bm_3 + c = ((a) * m_2 + b) * m_3 + c$ for three-dimensional arrays - to aggregate the offset. The generated code is returned.

Now, to generate code to access arrays using an offset, we simply compute the offset and return the address of the element to be accessed. Then the `Assn_Ast::gen_tac` (Assn for assignment) function is given two booleans corresponding to whether dereferencing is required or not of the left and right hand sides respectively. When dereferencing is required on the right hand side, it simply adds an `load_addr` statement (converted to `la` in asm):

```
load_addr $t1, 0($t0)
```

On the left hand side, we need instead to add a store operation (converted to `sw` in asm):

```
store_addr $t1 0($t0)
```

The situation for single-level pointers is identical (indeed, the `Assn_Ast` did not need to know whether it is an array access or not, it simply needed to know whether a dereference is required or not.

For situations where multiple dereferences are allowed, we implement a `Dereference_Ast` that indicates one dereference operation, and to be consistent with the above `Assn_Ast::gen_tac` function, the yacc code generates one less dereference operation in total. We have not done this yet.

4 Structures and methods

We propose how to extend our implementation to support structures, and then optionally classes (adding methods). We do not discuss inheritance here. There are four problems to tackle: (a) declaration processing (b) access (c) visibility checks (d) methods.

4.1 Type checking and declaration processing

The `Type` struct from before is modified in the following way:

```
class Type {
```

```
    int info;
    /* empty array (for base/ptrs), 1 element (for arrays), multiple elements (structs) */
    std::vector<std::pair<std::string, Type *>> children; /* name and type */
    std::vector<bool> visib; /* visibility of each child */
};
```

The `yacc` code for processing struct declarations (not including methods) is simple:

```
struct := name '{' child_or_visibility_list '}' ; | name '{' '}' ;
child_or_visibility_list
    := var_decl_list child_or_visibility_list
    |  visibility    child_or_visibility_list
    |  var_decl_list
    |  visibility
visibility := public: | private:
```

The actions are also simple enough: `visibility` simply toggles a global bit in the code (as $0/1$ depending on whether it evaluated to `private` or `public`). Encountering `var_decl_list` pushes a child back to the current struct's `Type` along with the current visibility bit.

4.2 Handling access

Like for array access, an address-style mindset is useful here. To access a struct attribute (using the `.` or `->` operators), we compute only the address of the required attribute. This is done to allow for assignment of a value to the attribute, as well as using the attribute as part of an expression. (The extra store operation to be generated to store the updated value of the attribute back on the stack is handled by the `Assn_Ast`, as described in the section on arrays). The addresses of attributes w.r.t the address of the top of the struct (i.e. the offsets of the attributes) are computed at compile time; the address of the struct is then added to this offset (a code statement is generated for this) to return the address of the attribute `<struct>.<name>` for further use.

The SDTS given in the slides for the theory course describes code to compute the TAC for accesses in this way. It also describes yacc code to process struct accesses, so we shall not do that here.

4.3 Visibility Checking

This is an operation that is completed by the time the AST for the program has been built. On each struct access, we look for the struct name in the symbol table, if exists, fetch its `Name_Ast` from the table, check the type stored for a variable with the name corresponding to the accessed attribute, and if exists, check the visibility bit corresponding to the variable. If all checks pass, we continue, else flag an error. This ensures visibility checks.

4.4 Implementing methods on objects

We have not thought as much about a concrete implementation of methods, so we may not be complete in our analysis here (for example, we do not cover operator overloading and constructor initializer lists here).

Methods are declared as functions inside the struct code; in the yacc action, we append to the name of the function the struct name and `::`. For example, the method `drive` on `class Car` is stored as a free function with name `Car::drive` with an extra argument `Car * const` (with name `this`). This name will not conflict, since `::` is not allowed in function names and so the name essentially serves as `(Car,drive)`. We will need special yacc nonterminals (and a lex statement for the token) for the constructor, copy constructor and destructor.

For method calls, say `name.foo` or `name->foo`, we look up the type `T` (or `*(T.basetype)` if `T` is a pointer) of `name`, and register a call to `T::foo`, prepending the `Name_Ast` corresponding to `name` to the argument list, as a `Func_Call_Ast`.

For declarations, we must call the constructor after making space for the object; we do this in the yacc code whenever a variable of that type is created. If no constructor is defined, we do nothing (imitating the default constructor).

5 Testing and speed analysis: The Program Generator

To test our compiler's correctness (in particular, we only check for acceptance and correct assembly output of `valid` programs), we wrote a small program generator that generates as large programs as we wish, adhering to the sclp grammar.

The implementation is done in Python, and is a top-down conversion of the sclp grammar. For example, an expression is generated like so:

```
def expr(depth: int=3, return_type: str):
    // a constant/name from the symtab with the correct return type
    if (depth == 0) return operand(return_type)
    toss = randint(0, 1 + (return_type == 'bool'))
    match toss:
        // new arithmetic expression of random smaller depth
        case 0:
            depth_rec = randint(0, depth-1)
            return arith(depth=depth_rec, return_type=return_type)
        // ternary
        case 1:
            depth_rec = randint(0, depth-1)
            return ternary(depth=depth_rec, return_type=return_type)
        // special for return type bool - comparisons
        case 2:
            depth_rec = randint(0, depth-1)
            return comparison(depth=depth_rec)
```

Each called function requires its own expressions and recursively asks for them with whatever type is needed. Random statements are similarly generated:

```
def stmt(depth: int):
    toss = randint(0, 6)
    match toss:
        // returns a random declaration
        case 0:
            typ = choice('bool', 'int', 'float', 'string')
            return decl(typ)
        // gets random name from the symbol table, random expression of same type and assign
        case 1: return assign()
        // random io
        case 2: return io()
        // function calls
        case 3:
            func = choice(self.avail_decls)
            return call(func) /* call with random expressions as args (correct type)
        // function return
        case 4:
            return f'return {expr(depth=3, return_type=self.return_type)}
        // if (else) statement (with a random number of inner statements with their own depths)
        case 5:
            return ifelse (depth, num_stmts=5)
        // (do) while statement
        case 6:
            return while  (depth, num_stmts=5)
```

We implemented a function definition generator taking its parameters and a list of global arguments that can be referenced as arguments. A function symbol table is maintained and a list of available functions (for calling) at that point is also given as an argument.

```
def func_def(args, globl_args, decls):
    signature = <create fn signature as a string>
    ans = f'{signature} {\n'
    for i in range(10): ans += decl()
    for i in range(20):
        ans += stmt()
    return ans + '}'
```

A program is then a global section - a bunch of random declarations and random function declarations, and then a list of function definitions. We tested our compiler by `diff`ing the output of our compiler and the reference implementation.

This also has the added benefit of allowing an analysis of the compilation speed. We conducted a preliminary speed analysis on our compiler; some rough times are below (run with 2GB of RAM).

Approximate program size (lines)	sclp runtime (s)	A5-sclp runtime (s)
1,000	0.1	0.45
2,000	0.2	0.73
3,000	0.3	1.33
4,000	0.44	1.82
10,000	1.49	11.20
25,000	3.14	24.23
36,000	5.16	37.76
75,000	9.74	out of memory

Regressing with more testcases gives us a speedup of at least **4x** over the reference implementation.

6 Acknowledgement

Thank you for a really fun course. We learnt lots of C++ stuff, good practices, and of course how to write a compiler.