SNUPLC: SNUPL/2 Compiler

Yongun Seong

2023-12-12

Contents

1 Introduction

As a part of the Compilers course, we were tasked with building a compiler for the SnuPL/2 language. We worked our way up through the various steps that a real-world compiler would perform, each building upon the last. Starting from the program source, we convered it into a token stream, then an abstract syntax tree, then an internal representation in the form of a three-address code, and finally assembly code that could be fed into an assembler to produce a working executable.

2 Implementation

We implemented our compiler in phases, building upon the provided skeleton code. We describe each phase in detail below.

2.1 Lexical Analysis

For the first phase, we implemented a scanner that tranforms the program source into a token stream. We were provided with a scanner for a simpler SnuPL/-1 language, which we modified to match the SnuPL/2 specification. We started from the simplest tokens, the keywords and single-character tokens. We then implemented the more complex tokens, such as numeric literals and strings. Thankfully, the provided skeleton code was very helpful and handled most of the complexity in handling strings and characters, such as escape sequences. For numbers, however, we had to write our implementation from scratch, building up a number from its digits.

2.2 Syntax Analysis

In the next phase, we implemented a parser, converting the token stream into an abstract syntax tree. This phase was the most complex, simply by the number of lines we ended up having to write. We built up the AST by writing a method for each language construct, which would recursively call other methods to parse its subparts.

To make the process of implementation bearable, allowing us to check our work as we went along, we worked on the parser in a bottom-up fashion. We started by implementing the simplest, smallest constructs such as literals and identifiers, working through the various expressions and statements, and finally the top-level module.

Our parser performs slightly more checking than possible in a pure, context-free parser. For example, we check that the closing `end` keyword is followed by the matching identifier for the scope being closed. Also, we ensure that each identifier is declared before being used. We implemented the latter check by creating two separate methods to parse identifiers, one where the identifier is being newly declared, in a `var` or `const` decl for example, and one where the identifer is being used.

2.3 Semantic Analysis

For the semantic analysis phase, we implemented a type checker as well as any other checks that cannot be easily performed by a parser.

Before implementing the type checker proper, we implemented a type system, adding a `Match` and `Compare` method to each `CType` class. These methods checked whether `this` matched the given type, usually requiring them to be exactly equal. The only exceptions were numeric types (integer and long integers) which were allowed to be implicitly converted to each other, and open arrays. We could have implemented explicit type conversions instead, but it was more convenient to allow them to be handled silently, implicitly, and it even resulted in simpler code down the line. Finally, each array argument was implicitly converted to accept a pointer to the array instead.

Once we had our type system in place, our type checker used it by traversing the AST, checking the type of each node, as well as computing the type of each node. For example, a binary operator node would check both its operands, ensure they are compatible with the given operator, based on the comparison methods we implemented above, and then compute its own type based on the types of its operands.

We also implemented the following checks:

- Ensure constant assignments are valid, and can be fully evaluated at compile time.
- Ensure the number of arguments passed to a function matches the number of parameters declared by the function.
- Ensure array index operators are valid, with the correct number of indices of integer type.

We are aware of a few limitations of our implementation, that result in our compiler accepting invalid programs that a fully-featured SnuPL/2 compiler should reject. Notably, we do not check whether all possible branches of a function actually return a value. We could have implemented this in multiple ways. For example, we could have required the last statement of a function to be a return statement, though would reject arguably valid code like `if true then return 0 else return 1 end`. Another option would have been to actually check each branch, which would have been complex and time-consuming to implement.

2.4 Intermediate Code Generation

During the intermediate code generation phase, our compiler transforms the abstract syntax tree into three-address code. From the provided skeleton code, we added this phase by implementing the `ToTac` function for each AST node type.

Roughly speaking, by the amount and complexity of the code, this phase was the simplest phase of them all. Many node types had straightforward, natural translations to a TAC, most of them clocking in at less than 10 lines of code. There were a few exceptions, however, which we discuss below.

2.4.1 Boolean Expressions

Boolean expressions required special care, as they were defined to be short-circuiting. This meant that the `&&` and `||` operators were actually control flow statements, and could not be translated directly into `and dst <- src1, src2` or similar.

Instead, we wrote two `ToTac` methods for each AST node type that could appear in a boolean expression: one that simply computed and returned its value, and another that would jump to one of the given labels, depending on whether the value was true or false.

2.4.2 Symbolic Constants

We also implemented symbolic constant "inlining" in this phase. Before translating a `CAstDesignator`, if the matching symbol was a compile-time constant, it was fully evaluated and replaced with its value instead.

Notably, for strings, we had to look up the symbol table to find a global symbol that matched the value of the given string. This was because we had lost the original reference to the string constant during parsing. We expect that there is a better, intended way to implement this with the given skeleton code, but our implementation worked well enough.

```
// only strings can be const arrays
if (GetType()->IsArray()) {
  auto *data = (const CDataInitString *) Evaluate();
  string str = data->GetData();
  CSymtab *st = GetSymbol()->GetSymbolTable();
  for (auto sym : st->GetSymbols()) {
    if (sym->GetSymbolType() != ESymbolType::stGlobal)
      continue;
    if (sym->GetData() == data)
    return new CTacName(sym);
  }
}
```

2.5 Code Generation

For this final phase, our compiler tranforms the intermediate three-address code into assembly code. In this case, we only targeted the x86_64 (AMD64) architecture.

We start by translating the top-level module, which recursively translates any subscopes such as functions and procedures. For each scope being translated, we generated its data section, containing any global variables or string constants. Then, we generated its text section, starting with the prologue, the body, and finally the epilogue.

For the data section, we did not modify the provided skeleton code at all, as it was already sufficient for generating both global variables and string constants. It took care of allocating space for each variable by generating `.align` and `.skip` directives, and handled string constants by generating `.asciiz` directives with aprpropriate labels.

For the text section, before we could emit any code, we had to first calculate the size and structure of our stack frame (discussed later), and assign each symbol to a memory location. Global variables, strings, and procedures were assigned a `rip`-relative addresses, resolved by the assembler. Parameters were assigned to fixed locations in the stack relative to `rbp`, and local variables were assigned in whichever order they were encountered as we iterated over the symbol table, being assigned to incrementing offets from `rsp`, plus whatever offset required to skip over the argument build area, aligned to 8 bytes.

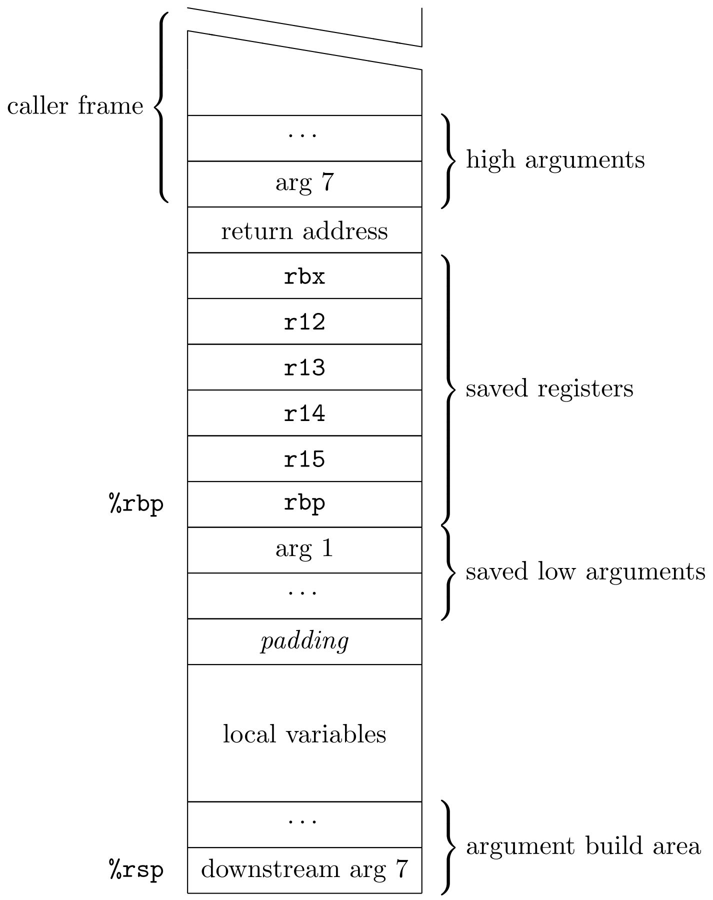
Once we had resolved each symbol to a location, we emitted the prologue, and then translated each TAC instruction in the current scope, one by one. TAC instructions of them of the form `dst <- src1 op src2` into `LOAD src1, %rax`, `LOAD src2, %rbx`, `op %rbx, %rax`, and finally `STORE(%rax, dst)`. Here, `LOAD` and `STORE` referenced the memory locations that we resolved earlier.

Interestingly, this simple instruction translation scheme allowed us to handle implicit widening and narrowing of numeric operands elegantly. The `LOAD` instructions took care of emitting the correctly-sized instruction to load the operand with sign-extension, but the actual operation was always performed on the full 64-bit register. Finally, the `STORE` instruction correctly truncated any result to the correct size.

The difficult part of this phase was not translating each instruction, but the mapping of each operand into a memory location, and calculating the appropriate sizes of each stack frame in order to correctly generate the prologue and epilogue.

2.5.1 Stack Frame

Our compiler produces a stack frame as follows:



We use `rbp` when accessing saved arguments, regardless of whether they were passed through the stack or through the registers. In fact, the position of every argument depends only on its argument index, with low (1 through 6) arguments at $\texttt{rbp} - 8 \times \texttt{index}$ and high (7 and above) arguments at $\texttt{rbp} + 8 \times \texttt{index}$. Similarly, we use `rsp` when accessing local variables or when building the argument list.

In the end, the procedure prologue looks like this:

```
# save callee saved registers
pushq   %rbx
pushq   %r12
pushq   %r13
pushq   %r14
pushq   %r15
pushq   %rbp
# set rbp to the current stack pointer
movq    %rsp, %rbp
# allocate space for locals and arguments
```

```
subq    $[calculated size], %rsp
# align to 16 bytes
andq    $-16, %rsp
# save parameters to stack
movl    %edi, -8(%rbp)
```

In particular, we align the stack pointer to 16 bytes using the `andq $-16, %rsp` instruction. This way, we do not have to calculate the exact size of the padding, nor care whether our caller has already aligned the stack pointer. Although we lose track of the exact position of the stack pointer, we can easily restore it in the epilogue by by copying the value from `rbp`.

The epilogue is quite simpler:

```
leave
popq    %r15
popq    %r14
popq    %r13
popq    %r12
popq    %rbx
ret
```

Notably, the epilogue is exactly the same regardless of the number of arguments, the number or size of the local variables, the size of the padding, or anything else.

3 Discussion

Before starting this project, we did not know any C++, though we did know C. In fact, we still cannot say we understand C++ that well, and any C++-specific features we did use were guessed from the context. Thankfully, not understanding C++ did not cause any major problems, and we were glad that we did not have to learn C++ just for this project. We were, however, hindered by the fact that we did not know how to write and run unit tests in C++, as it would have been extremely useful to have a robust test suite for each phase of the compiler, with coverage reports. Indeed, given the nature of compiler code, it would have been very easy divide the compiler into units to test separately. Instead, we had to test everything manually, which was very time-consuming and resulted in many missed bugs.

On bugs, on every phase after the first, we discovered significant bugs we had missed in the previous phases, which we discovered only because they surfaced while testing the then-current phase. Even now, we are not sure whether we found all the bugs. Luckily, none of the bugs we found required significant refactoring after the fact, and were mostly self-contained.

4 Conclusion

We successfully built a working compiler for the SnuPL/2 language, implementing each step performed by a real-world compiler: lexing, parsing, type checking, intermediate code generation, and finally target code generation. It generates assembly for the x86_64 architecture, which can be assmbled to produce a working binary. The only major phase we did not handle was optimization, though realistically, the optimization would be one of the important and complex jobs of a real-world compiler.