Code Generation for a Register Machine

The goal of this project is to implement a basic machine code compiler for small programming language and target a register machine of your choice. In addition to this brief, basic code can be found at

https://github.com/smarr/acc-codegen-project

This project is optional and the first of two possible options. If you chose not to do a project, the best possible mark will be a 2 (good/“gut” in German). If you chose to do a project (at most one will be considered), submit it, successfully defend it (at a level of good or better), and you achieve at least 40 points in the exam, you overall mark will be the exam mark improved by 1 grade.

The project has to be submitted on Moodle and presented. Up-to-date details will be made available here:

https://ssw.jku.at/Teaching/Lectures/ACC/

Please notify me of issues with the project description, ask questions, or request clarifications via email (stefan.marr@jku.at).

All projects are to be done individually. Teamwork is not permitted.

Submission Deadline on Moodle: June 22nd, 2026, 23:59 Linz time.

Project Presentation Date: June 23rd, 2026

1. The Small Programming Language: Simple Language 2026

Designing programming language is a great hobby for anyone who likes to fully understand how things work. Unfortunately, languages get complex very quickly.

We use the language specified by the following grammar for this project:

```
SimpleLang  = Declaration {Declaration}.
Declaration = VarDecl | FnDecl.
VarDecl     = "var" ident ":" Type ";".
Type        = ident.
FnDecl      = "fn" ident Parameters "{" {VarDecl} StatSeq "}".
Parameters  = "(" [Param {"," Param}] ")" [":" Type].
Param       = ident ":" Type.
StatSeq     = Statement {Statement}.

Statement =
      ident ( "=" Expression | ActParameters ) ";"
    | "if" "(" Condition ")" "{" StatSeq "}"
      {"elseif" "(" Condition ")" "{" StatSeq "}" }
      ["else" "{" StatSeq "}"]
    | "while" "(" Condition ")" "{" StatSeq "}"
    | "return" [Expression] ";"
.
Condition  = Expression Relop Expression.
Expression = [Addop] Term {Addop Term}.
Term = Factor {Mulop Factor}.
Factor = ident [ActParameters] | number | charCon | "("
Expression ")".
ActParameters = "(" [Expression {"," Expression}] ")".
Relop = "==" | "!=" | "<" | ">" | ">=" | "<=".
Addop = "+" | "-".
Mulop = "*" | "/" | "%".
```

The language is designed to be easy to parse and has only integer and char values. Integers are of machine-word size, and characters are 1-byte values.

The full grammar based on Coco is available here: https://github.com/smarr/acc-codegen-project/blob/main/codegen/SimpleLang.ATG

Built-in functions

put(e)	prints the expression e which is of type char to the console
putLn	starts a new line
ORD(ch)	converts the character ch to an integer
CHR(i)	converts the integer i to a character

Here a sample program in our language:
also available here: https://github.com/smarr/acc-codegen-project/blob/main/tests/test.sl

```
var i: int;

fn putInt(x: int): void {
    /* largest printable number = 9999 */
    var c0: char;
    var c1: char;
    var c2: char;
    var c3: char;

    c3 = CHR(48 + x % 10); x = x / 10;
    c2 = CHR(48 + x % 10); x = x / 10;
    c1 = CHR(48 + x % 10); x = x / 10;
    c0 = CHR(48 + x % 10);

    if (c0 > '0') { put(c0); put(c1); put(c2); }
    elseif (c1 > '0') { put(c1); put(c2); }
    elseif (c2 > '0') { put(c2); }
    put(c3);
}

fn main(): void { /* print odd numbers */
    i = 1;
    while (i < 100) {
        putInt(i);
        putLn();
        i = i + 2;
    }
}
```

2. Recommended Approach to the Project

The focus of the project should be building up the necessary data structures to represent the program's information during compilation and then generate machine code for it.

This means, the compiler can be a simple single-pass compiler that generates code during the parsing process. The grammar is kept deliberately simple to allow for a classic recursive descent parser. Parser generators such as Coco or ANTLR can be used: https://ssw.jku.at/Research/Projects/Coco/

Therefore, I'd recommend the following steps:

1. Implement a parser based on the provided grammar, manually, or with a parser generator.
2. Implement a symbol table to capture the necessary information about functions, types, locals, globals, etc.
3. Add code generation to target a register-based instruction set architecture.

Below, I'll provide more guidance on these three steps.

3. Implementing a Parser

The provided grammar can be used to generate a parser with Coco. The repository with example code also contains a `SimpleLang.java` that accepts the included `tests/test.sl` file. `SimpleLang.ATG` defines the basics required for parsing but does not include any further functionality.

I'd suggest that you use the programming language you are most familiar with for this project and use Coco to generate the basic parser as an initial template to work with. C# and Java would be the classic candidates for which Coco can generate code.

If you chose to use Coco or ANTLR, you could implement most of the functionality as part of the attributed grammar, but since it likely will require debugging, I'd recommend using an approach that makes debugging easy for you.

4. Implementing a Symbol Table

For the symbol table, I'd suggest that you use the equivalent of a Java LinkedHashMap to represent the information within a scope and use `Obj` and `Struct` nodes to represent the definitions/names and the corresponding type information.

The implementation should at least do the following:

- Build up a symbol table with `Obj` and `Struct` nodes

- Initialize the universe with the nodes for the int and char types, as well as the built-in functions put, putLn, ORD, and CHR.
- The parser should check the context conditions for type correctness and unique declarations. Thus, it should not allow us for example to define multiple variables, parameters, or functions with the same name, and should also ensure that the program is type correct.

5. Implementing Code Generation

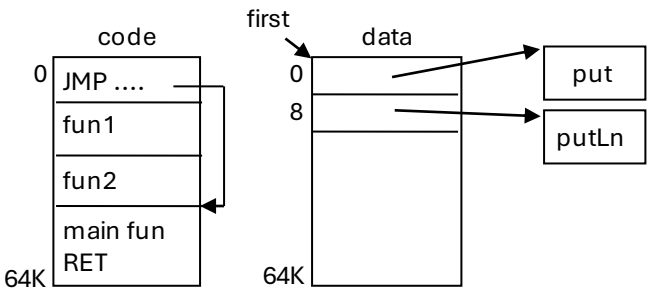
The choice of target instruction set architecture is yours. I'd recommend one for which you have a machine to run the program on, which you can also access when demonstrating the project as part of the project presentation.

The recommended choices would be x86-64 or AArch64. For both, the repository contains a code generator for a Hello World example.

The compiler is expected to generate a plain file with the binary code. Though, this can be a multi-step process. The learning goal is to map our Simple Language to a machine language, which could also be a textual assembly form for the chosen architecture. This text file could then be compiled to an object file, from which on can extract the binary code. The provided assembly.sh script does this for the Linux/x86-64 and macOS/AArch64.

The resulting file of machine code can be run with the launcher program (codegen/launcher.c). The loader allocates memory for the code and the data according to the figure below. After loading the code, it does a CALL to address 0 of the loaded code. At the end of the program's body there should be a return instruction that causes the program to return to the loader.

Note, the calling convention is architecture specific. On AArch64, the pointer to the data is in register 0. On Linux/x86-64, the pointer is in the `RDI` register. The provided examples show how to use it and how to extract the pointers to `put` and `putLn`.



Global data and variables are expected to be stored in that data array after the pointers to the two built-in functions.

6. Testing

It is highly recommended to implement basic tests to check on the key functionality of the symbol table and code generator.

Start with a simple Hello World-style program, just the main function, containing calls to the built-in functions.

Then, step-wise built up from there, and add support for variables, control structures, and globals step by step.

The provided GitHub repository has a basic GitHub Action Setup, which can be used as a starting point to have automatic tests.

7. Basic Marking Rubric

As a very rough guide to the marking, here a basic rubric. This is also an indication of what we will roughly ask during the project presentations. To raise the exam mark by 1, only the elements categorized as good are strictly required.

	good	very good
Code Generation	Generate code as textual assembler, which is compiled in a separate step to binary machine code	Generate binary machine code directly
Language Completeness	Support all language features used by test.sl	Support all language features indicated in the grammar.
Language Correctness	Support all correct programs.	Give useful errors for incorrect programs.

Plagiarism and Duplication of Material

The work you submit must be your own. We will run checks on all submitted work to identify possible plagiarism or other academic misconduct and take disciplinary action against anyone found to have broken the University's rules on academic integrity.