

Compilers Project

Maximiliano Sá
Rita Moreira

Faculdade de Ciências da Universidade do Porto

December 2025

Contents

1	Introduction	2
2	Tools	3
2.1	Requirements	3
2.2	Build	3
2.3	Run	3
2.4	Directory Tree	4
2.5	Frontend	5
2.6	Lexical and Syntax Analysis	5
2.6.1	Grammar	5
2.6.2	Operator Precedence	5
2.7	Semantic Analysis	6
2.7.1	Abstract Syntax Trees - AST	6
2.7.2	Symbol Table	6
3	Intermediate Representation	7
3.1	Code Generator	7
3.1.1	Three Address Code TAC)	7
4	Backend	8
4.1	MIPS generator	8
5	Conclusion	9
6	Webgraphy	10

1 Introduction

This project consists of an Ada language compiler for a small subset, which includes statements, conditions, while loops, I/O operations, and expressions that range from types of value to actual operators. All of this inside one main procedure.

In general, the parser builds an Abstract Syntax Tree (AST) for a block of code and supports pretty-printing and a structural debug dump of the AST. From there, a symbol table is built. Also, with the help of the AST, its three address code is generated, printed, and sent to be transformed into MIPS. Last but not least, the MIPS generated code, when run on <https://dpetersanderson.github.io/features.html>, gives the result of the initial code we sent out as input. This code was made in C.

Note: All of the examples below are from input3.adb.

2 Tools

2.1 Requirements

To run this compiler on your machine, you need to install **make**, **flex**, **bison**, and **gcc**.

2.2 Build

To compile the code for personal use, it is recommended to use Makefile, because it is simpler. The executables presented in the document (printAST_MAC and printAST_Linux) were compiled using the command `make && mv printAST printAST_{your OS}`, which regenerates `parser.tab.c/.h` and `lex.yy.c` when needed, and builds `printAST_{your OS}`. Another way of compiling is by doing it manually:

```
bison -d parser.y  
flex lexer.x  
gcc -Wall -g -o (nameOfExecutable) parser.tab.c lex.yy.c  
ast.c ast_debug.c symbolTable.c codeGenerator.c  
mips_backend.c main.c -lm'
```

2.3 Run

There are two different executables in this repository: printAST_MAC and printAST_Linux. The user must choose the one with the OS they want. If the user wants to create another one, go to 3. To run it with pretty-printing of the AST, use this:

```
./printAST_{your OS}  
or  
./printAST_{your OS} path/to/file.ada
```

To run the debug AST dump instead, use:

```
./printAST_{your OS} -- debug  
or  
./printAST_{your OS} -- debug path/to/file.ada
```

Here, the path/to/file.ada is optional, and, if omitted, uses standard input as the input.

2.4 Directory Tree

```
|-- code/
|-- ast.c
|-- ast.h
|-- lexer.x
|-- main.c
|-- Makefile
|-- parser.y
|-- codeGenerator.c
|-- codeGenerator.h
|-- mips_backend.c
|-- mips_backend.h
|-- symbolTable.c
|-- symbolTable.h
|-- printAST_Mac
|-- printAST_Linux
|-- test_inputs/
    |-- input1.ada
    |-- input2.ada
    |-- input3.ada
    |-- input4.ada
    |-- input5.ada
    |-- input6.ada
    |-- input7.ada
    |-- input8.ada
|-- test_outputs/
    |-- output1MIPS.s
    |-- output2MIPS.s
    |-- output3MIPS.s
    |-- output4MIPS.s
    |-- output5MIPS.s
    |-- output6MIPS.s
    |-- output7MIPS.s
    |-- output8MIPS.s
|-- README.pdf
```

2.5 Frontend

2.6 Lexical and Syntax Analysis

2.6.1 Grammar

This subset's grammar consists of:

- Procedure: procedure {name of procedure} is begin {stm list} end {name of procedure};
- Statement list: could be zero or more **statements**
- Statements:
 - Assignment: {ID} := {expression};
 - If then condition: if {expression} then {stm list} end if;
 - If then else condition: if {expression} then {stm list} else {stm list} end if;
 - While loop: while {expression} loop {stm list} end loop;
 - Put (print): Put_line({expression});
 - Get (read/scan): Get_line({expression});
- Expressions:
 - Literals
 - ID
 - unary minus (-)
 - unary not (!)
 - binary operations
 - logical operations
 - relational operations
 - (expression)

2.6.2 Operator Precedence

To be able to read expressions in the correct way mathematically, it is necessary to implement precedence (left, right, or non-associative). From highest to lowest, the order of precedence in expressions is:

- Unary not and unary minus
- Power
- Multiplicative, division, modulus and remainder

- Additive (add and subtract)
- Relational/equality (equal, not equal, less than, greater than, less than or equal to, greater than or equal to)
- Boolean AND
- Booleans OR and XOR

2.7 Semantic Analysis

2.7.1 Abstract Syntax Trees - AST

This AST implementation contains the expressions NUM, FLOAT, ID, STRING, BOOL, OP(op, left, right), UNARY(op, child), and the statements ASSIGN(id, expr), PUT_LINE(expr), GET_LINE(id), IF(cond, then, else = optional), WHILE(cond, body), COMPOUND(left, right), PROC(name, body). From here, there are two types of AST printing:

- Pretty printing

```
PROCEDURE main IS
BEGIN x := 5 ; y := 3.14 ;
IF x > 3 THEN PUT_LINE("ok");
ELSE PUT_LINE("no"); END IF;
END main;
```

- Debug printing

```
PROCEDURE(main)
body:
ASSIGN(x, NUM(5))
ASSIGN(y, FLOAT(3.14))
IF
  cond: OP(GT, ID(x), NUM(3))
  then:
    PUT_LINE(STRING("ok"))
  else:
    PUT_LINE(STRING("no"))
```

2.7.2 Symbol Table

To define each symbol presented and to pass that definition to the Three Address Code, a symbol table is essential. It includes, for each symbol that appears, its type, kind, size, offset, canonical and original name, and scope (which is always the name of the procedure, except the actual procedure, since functions are not implemented).

Symbol Table					
CANONICAL NAME	NAME	KIND	SCOPE	SIZE	OFFSET
y	y	VAR	main	4	4
x	x	VAR	main	4	0
main	main	PROC	Global	0	0

In a more technical way, the symbol table header defines kinds of symbols (VAR, CONST, TYPE, FIELD, PROC), kinds of types (Integer, Float, Boolean, String, Void), the backbone of entries and symbols, as mentioned above, and different functions, such as creating the table, adding and removing entries, looking up for values and even checking semantics, to deal with the table while reading the AST.

3 Intermediate Representation

3.1 Code Generator

3.1.1 Three Address Code TAC

To succeed at this translation, we need to define instruction types (LABEL, JUMP, JUMP FALSE, PRINT, READ, ASSIGN, COND), binary, unary, and relational operations, an atom (operand of TAC), and its kind, an instruction, an instructions node and list, and the state of the code generator.

We also need to create functions to initialize the generator, construct the different types of atoms, manage temporary values, construct instructions (emitters), print an instruction and a list of them, manage the memory and the list, and the most important, translating functions from AST to TAC (transExp, transCond, transStm).

```

x := 5
y := 3.14
t0 := x
t1 := 3
COND t0 > t1 L0 L1
LABEL L0
PUT "ok"
JUMP L2
LABEL L1
PUT "no"
LABEL L2

```

4 Backend

4.1 MIPS generator

MIPS or assembly is a text representation of machine code (0's and 1's), more readable than binary code. It uses symbolic labels for addresses and constants. It can be transformed into executable machine code by an assembler. The MIPS architecture uses 32 registers (\$0-\$31) of 32 bits, has a set of load/store instructions, and the instructions in general have a fixed size of 32 bits. In addition to that, the operations can be between 3 registers or registers and constants.

Our MIPS code generates an archive MIPS from a list of instructions, and the content inside the archive is the actual code of those instructions. Because the MIPS architecture has a finite number of registers, something that TAC does not have. The implementation uses a cyclic mapping (round-robin) that defines 16 available registers (8 temporary registers and 8 saved registers), for when the TAC asks for \$t20, the backend calculates $20\%16 = 4$, and the compiler reuses the physical address that corresponds to index 4.

Another important aspect of this implementation is that the reading of complex types, such as float or string, demands the use of specific syscalls and a different memory management compared to simple integers.

For floats, the MIPS architecture uses a specific coprocessor for the floating point. The compiler emits `syscall 6`, and the value read by the system is stored automatically in the float register \$f0. Then, that value is moved from \$f0 to the memory position that was reserved in the stack for that variable, using the instruction `swc1` (Store Word Coprocessor 1).

For strings, this process requires a pre-allocated buffer. The generator detects when the string reading is necessary, and allocates a global space for `str_buf` in `.data` section. The `syscall 8` is used, which needs two arguments: the buffer address \$a0 and the reading maximum size \$a1. After the scanning, the buffer's address is stored in a local variable in the stack. So, the variable points to the address where the text was actually written.

```
.data
str_1: .asciiz "no"
str_0: .asciiz "ok"
flt_0: .float 3.140000
.text
.globl main
main:
    addi $sp, $sp, -16
    sw $ra, 12($sp)
    sw $fp, 8($sp)
    addi $fp, $sp, 16
    addi $t0, $zero, 5
    sw $t0, 0($fp)
```

```

la $t0, flt_0
lwcl $f12, 0($t0)
swcl $f12, 4($fp)
lw $t1, 0($fp)
move $t2, $t1
addi $t0, $zero, 3
move $t3, $t0
slt $1, $t3, $t2
bne $1, $zero, L0
beq $1, $zero, L1
L0:
la $a0, str_0
li $v0, 4
syscall
j L2
L1:
la $a0, str_1
li $v0, 4
syscall
L2:
li $v0, 10
syscall

```

The MARS (MIPS Assembler and Runtime Simulator) prints, for this list of instructions, the string "ok".

5 Conclusion

In this project, we successfully developed a functional compiler for a specific subset of the Ada language. By leveraging tools such as Flex and Bison, we implemented a robust frontend capable of performing lexical and syntactic analysis, resulting in the construction of an Abstract Syntax Tree (AST) that faithfully represents the program's structure.

The semantic analysis phase was supported by a Symbol Table, which allowed for efficient scope management and type checking during the traversal of the AST. A significant part of the development was focused on the backend, where we implemented an Intermediate Representation (Three Address Code) to bridge the gap between high-level Ada constructs and low-level machine instructions.

Ultimately, this project provided deep insight into the translation process of software, demonstrating how high-level logic is transformed into executable machine code through a structured and automated pipeline.

6 Webgraphy

- Teacher's slides and exercise sheets
- <https://github.com/marcauberer/compiler-design-series>
- <https://www.geeksforgeeks.org/compiler-design/introduction-of-compiler-design/>
- <https://medium.com/@nevo.krien/from-zero-to-building-my-own-compiler-ed0fcec9970d>
- <https://ada-lang.io>
- http://www.ada-auth.org/standards/aarm12_w_tc1/html/AA-A-10-1.html