bada, my ADA

bada is a compiler for a minimalist ADA implementation. It has just enough features to write interesting programs in ADA syntax. It supports terminal I/O, math for boolean, integer, and single-precision floats, arrays, and procedures -- all with limitations and exclusions.

To build bada requires Visual Studio 2017[[1]](#footnote-2) and the source code[[2]](#footnote-3). After building run the traditional hello world:

hello.adb

procedure main is

i : integer;

begin

i := 3 + 2 + 1;

put\_line("hello world");

put\_line(i);

end main;

C:\Users\pablo\Documents\compiler\parser>..\debug\**bada.exe hello.adb**

compiling hello.adb

Success is indicated by the lack of error messages. The output file produced is called hello.adb.asm. Load that into SPIM[[3]](#footnote-4) to see the expected output. The symbol table is in the oddly named hello.adb.tree.

If you have errors in your source there will be helpful messages and no asm output created. It's fine to compile multiple files at once as show below.

C:\Users\pablo\Documents\compiler\parser>..\debug\bada.exe testfloat.adb testint.adb qs.adb

compiling testfloat.adb

compiling testint.adb

compiling qs.adb

The interesting program qs.adb shows off the capabilities of bada. It is an implementation of quicksort. The user enters any number (up to 99) of floating point numbers. QS sorts them using quicksort, and prints the results. It requires procedures and arrays. Here's the source:

qs.adb

procedure main is

item\_count : integer;

swap\_count : integer;

compare\_count : integer;

values : array(100) of float;

procedure get\_user\_input is

exit : boolean;

item : float;

begin

put\_line("enter some positive float numbers terminated by -1");

item\_count := 0;

exit := false;

while not exit loop

get(item);

if (item = (0.0-1.0)) then

exit := true;

end if;

if not (item = (0.0-1.0)) then

values(item\_count) := item;

item\_count := item\_count + 1;

end if;

end loop;

end get\_user\_input;

procedure print\_list is

ix : integer;

begin

ix := 0;

while(ix < item\_count) loop

put\_line(values(ix));

ix := ix + 1;

end loop;

end print\_list;

procedure swap(in left : integer; in right : integer) is

temp : float;

begin

temp := values(left);

values(left) := values(right);

values(right) := temp;

swap\_count := swap\_count + 1;

end swap;

procedure quicksort(in lower : integer; in upper : integer) is

pivot : float;

left : integer;

right : integer;

is\_less : boolean;

continue : boolean;

begin

pivot := values(lower);

values(item\_count) := pivot; -- add a sentinel so we can't run off end

left := lower + 1;

right := upper;

continue := true; -- get do ... while() effect

while(continue) loop

compare\_count := compare\_count + 1;

while(values(right) > pivot) loop

compare\_count := compare\_count + 1;

right := right - 1; -- find a right value <= pivot

end loop;

compare\_count := compare\_count + 1;

while(values(left) < pivot) loop

compare\_count := compare\_count + 1;

left := left + 1;

end loop;

compare\_count := compare\_count + 1;

continue := false;

if(left < right) then

continue := true;

call swap(left, right);

left := left + 1;

right := right - 1;

end if;

end loop;

-- left and right have crossed, so move pivot into place

-- and recurse.

call swap(lower, right); -- move pivot.

if((right - 1) > lower) then

call quicksort(lower, right - 1);

end if;

if((right + 1) < upper) then

call quicksort(right + 1, upper);

end if;

end quicksort;

begin

compare\_count := 0;

swap\_count := 0;

call get\_user\_input;

call quicksort(0, item\_count - 1);

call print\_list;

put("quicksorted in ");

put(compare\_count);

put(" comparisons and ");

put(swap\_count);

put\_line(" swaps.");

end main;

This program is big enough to uncover some annoyances: the lack of <=, the lack of unary minus for constants, no ELSE for ifs, no do...while. There were obvious workarounds though, only the lack of 'else' wasn't easy to adapt to. Here's the output from a test run using data from random.org

begin program

enter some positive float numbers terminated by -1

9.25

0.74

5.76

5.22

9.49

9.47

4.01

7.66

1.13

5.92

6.53

1.71

4.56

3.84

4.68

5.92

6.49

4.23

6.56

7.54

-1

0.74000001

1.13000000

1.71000004

3.83999991

4.01000023

4.23000002

4.55999994

4.67999983

5.21999979

5.76000023

5.92000008

5.92000008

6.48999977

6.53000021

6.55999994

7.53999996

7.65999985

9.25000000

9.47000027

9.48999977

quicksorted in 121 comparisons and 24 swaps.

end program

Quicksort doesn't exercise the math so I made another program that does. gcd.adb uses Euclid's method to find the greatest common divisor of two positive integers.

gcd.adb

procedure main is

a : integer;

b : integer;

remainder : integer;

temp : integer;

begin

put\_line("This is a greatest common divisor program.");

put("Enter a positive integer. ");

get(a);

put("Enter a positive integer. ");

get(b);

if(a < b) then

temp := a;

a := b;

b := temp;

end if;

remainder := 1;

while(remainder > 0) loop

remainder := a mod b;

a := b;

b := remainder;

end loop;

put("The greatest common divisor is ");

put\_line(a);

end main;

The output from a sample run:

begin program

This is a greatest common divisor program.

Enter a positive integer. 15

Enter a positive integer. 9

The greatest common divisor is 3

end program

begin program

This is a greatest common divisor program.

Enter a positive integer. 13

Enter a positive integer. 97

The greatest common divisor is 1

end program

begin program

This is a greatest common divisor program.

Enter a positive integer. 24

Enter a positive integer. 8

The greatest common divisor is 8

end program

This short program is a good choice to show what the generated code looks like.

# MIPS assembly generated from BADA source.

.data

enter\_msg : .asciiz "\nbegin program\n"

exit\_msg : .asciiz "\nend program\n"

true\_msg : .asciiz "true"

false\_msg : .asciiz "false"

# user string literals:

literal\_1: .asciiz "This is a greatest common divisor program." # a string

literal\_2: .asciiz "Enter a positive integer. " # a string

literal\_3: .asciiz "Enter a positive integer. " # a string

literal\_7: .asciiz "The greatest common divisor is " # a string

LOCALS\_0 = -36

PARAM\_SIZE\_0 = 16

.text

.globl main

# begin user procedures

# source line #1

# source line #2

# source line #3

# source line #4

# source line #5

# source line #6

# begin procedure main\_0

main\_0:

addiu $sp, -8 # space for saved registers

sw $ra, 8($sp) # save return so we can make calls

sw $fp, 4($sp) # preserve caller frame

add $fp, $sp, $0 # create our frame

addiu $sp, LOCALS\_0 # space for local vars

# source line #7

li $v0, 4 # write string function

la $a0, literal\_1 # load string literal

syscall # do the write string

li $v0, 11 # write char function

li $a0, 10 # ascii char

syscall # do the write char

# source line #8

li $v0, 4 # write string function

la $a0, literal\_2 # load string literal

syscall # do the write string

# source line #9

li $v0, 5 # read integer function

syscall # do the read

sw $v0, 0($fp) # store result

# source line #10

li $v0, 4 # write string function

la $a0, literal\_3 # load string literal

syscall # do the write string

# source line #11

li $v0, 5 # read integer function

syscall # do the read

sw $v0, -4($fp) # store result

# source line #13

li $t0, 0 # place a literal in register

sw $t0, -16($fp) # move literal to memory

lw $t0, 0($fp) # load left op

lw $t1, -4($fp) # load right op

slt $t2, $t0, $t1 # binary op

sw $t2, -16($fp) # store result

lw $t0, -16($fp) # load if expression

beq $t0, $0, if\_4 # jump past when false

# source line #14

lw $t0, 0($fp) # load op

sw $t0, -12($fp) # assignment

# source line #15

lw $t0, -4($fp) # load op

sw $t0, 0($fp) # assignment

# source line #16

lw $t0, -12($fp) # load op

sw $t0, -4($fp) # assignment

# source line #17

if\_4: # after the if block

# source line #19

li $t0, 1 # place a literal in register

sw $t0, -20($fp) # move literal to memory

lw $t0, -20($fp) # load op

sw $t0, -8($fp) # assignment

# source line #20

while\_5: # before the while expression

li $t0, 0 # place a literal in register

sw $t0, -24($fp) # move literal to memory

li $t0, 0 # place a literal in register

sw $t0, -28($fp) # move literal to memory

lw $t0, -24($fp) # load left op

lw $t1, -8($fp) # load right op

slt $t2, $t0, $t1 # binary op

sw $t2, -28($fp) # store result

lw $t0, -28($fp) # load if expression

beq $t0, $0, if\_6 # jump past when false

# source line #21

li $t0, 0 # place a literal in register

sw $t0, -32($fp) # move literal to memory

lw $t0, 0($fp) # load left op

lw $t1, -4($fp) # load right op

div $t0, $t1 # mult op

mfhi $t2 # get result from special reg

sw $t2, -32($fp) # store result

lw $t0, -32($fp) # load op

sw $t0, -8($fp) # assignment

# source line #22

lw $t0, -4($fp) # load op

sw $t0, 0($fp) # assignment

# source line #23

lw $t0, -8($fp) # load op

sw $t0, -4($fp) # assignment

# source line #24

j while\_5 # jump to start of while

if\_6: # after the if block

# source line #25

li $v0, 4 # write string function

la $a0, literal\_7 # load string literal

syscall # do the write string

# source line #26

li $v0, 1 # write integer function

lw $a0, 0($fp) # load the integer

syscall # do the write integer

li $v0, 11 # write char function

li $a0, 10 # ascii char

syscall # do the write char

# source line #27

lw $ra, 8($fp) # restore our return addr

lw $fp, 4($fp) # restore caller frame

addiu $sp, 44 # restore locals space

jr $ra #

# end procedure main\_0

# end user procedures.

main:

# program entry point

addiu $sp, $sp, -12 # space for stack frame

addu $fp, $sp, $0 # init frame

sw $0 12($fp) # initial pfp is null

li $v0, 4

la $a0, enter\_msg

syscall #print string

addiu $sp, $sp, -4 # space for parent fp

sw $fp, 4($sp) # push parent frame

jal main\_0 # call user procedure

addiu $sp, $sp, 4 # pop params

li $v0, 4

la $a0, exit\_msg

syscall # print string

li $v0, 10

syscall # exit program

There are test programs that exercise each of the primitive types included in the source. Here's the output from running them.

**testint.adb**

begin program

proving that assert(false) is detected:

0 assert failed

now on to the tests...

writing integers

0

-1

21

16

30 tests taken.

end program

**testfloat.adb**

begin program

proving that assert(false) is detected:

0 assert failed

now on to the tests...

writing floats

0.00000000

-1.00000000

21.00000000

16.00000000

24 tests taken.

end program

**testbool.adb**

begin program

proving that assert(false) is detected:

0 assert failed

now on to the tests...

writing alternating values

true

false

true

false

true

false

17 tests taken.

end program

Finally, I have a program that generates every non-fatal error in the source. The error messages show where the error is in the source code and where it was found in the parser code.

**errors.adb**

C:\Users\pablo\Documents\compiler\parser>..\debug\bada.exe errors.adb

compiling errors.adb

\*\*\* error source line 7 parser line 438 type mismatch';'

error source line 18 parser line 231 duplicate procedure definition.'is'

error source line 20 parser line 261 Procedure end wrong name.'not\_duplicate'

error source line 26 parser line 417 duplicate definition 'i'

error source line 28 parser line 451 not a valid array size.'0'

error source line 30 parser line 521 changing a constant':='

error source line 31 parser line 526 type mismatch';'

error source line 32 parser line 539 type mismatch, expected boolean'then'

error source line 33 parser line 559 changing a constant')'

error source line 34 parser line 561 can't read a boolean')'

error source line 36 parser line 593 type mismatch, expected boolean'loop'

error source line 38 parser line 616 undefined procedure'undef'

error source line 39 parser line 619 expected a procedure'i'

error source line 40 parser line 630 wrong number of parameters for procedure';'

error source line 41 parser line 630 wrong number of parameters for procedure';'

error source line 42 parser line 663 changing a constant')'

error source line 43 parser line 667 not an out param: #0','

error source line 44 parser line 683 type mismatch parameter 0','

error source line 45 parser line 669 type mismatch parameter 2')'

error source line 46 parser line 681 missing 'out' param: #2')'

error source line 47 parser line 741 type mismatch';'

error source line 48 parser line 779 type mismatch';'

error source line 49 parser line 818 type mismatch';'

error source line 50 parser line 842 not requires boolean';'

error source line 50 parser line 526 type mismatch';'

error source line 51 parser line 884 undefined variable 'ii'

error source line 51 parser line 526 type mismatch';'

error source line 52 parser line 783 invalid operation on float. ';'

error source line 53 parser line 779 type mismatch';'

error source line 53 parser line 783 invalid operation on float. ';'

error source line 54 parser line 745 invalid operation on float. ';'

31 errors.

## Some Limitations Apply

Procedures are implemented, but not functions. The word 'call' was added to make it easier for the parser to identify procedure calls. By not having functions I don't have to worry about mixing calls inside of expressions. Return values are essential though, so that's done with 'out' parameters. I require every parameter to include a direction prefix of 'in' or 'out' (both is not supported). Also when making the call you have to include 'out' (but not in). This is for the benefit of the programmer, out parameters should be highly visible.

Arrays are implemented, but the lower bound is always 0 like C and not arbitrary like ADA. The syntax in ADA doesn't support having arrays as parameters directly. Instead you have to do a typedef of the array and that typedef can be a parameter. I chose not to do that so arrays can't be parameters in bada. Arbitrary nesting of procedures and data does work though so arrays can be shared without being full globals like they would be in C. Also the upper bound has to be a positive integer literal and there's no bounds checking.

## Improvements

* Not having dynamic memory is a big loss. Add that and bada is far more useful.
* 'else' is important. Having to repeat 'if' expressions feels bad.
* '=' not having the very lowest precedence caught me more than once. It's not natural, and hard to debug.
* I want structs.
* Running on x86 is more practical than using SPIM.

## Reliability

There are certainly bugs. Every new program I wrote uncovered more bugs. There are no *known bugs* however.

## About The Code

A few words about the purpose of each file. First the headers:

|  |  |
| --- | --- |
| codegen.h | Contains the methods called by the parser at points where code needs to be generated. |
| location.h | Defines a Location struct with all type & location info. |
| parser.h | Declares a parse function for main to use. |
| scanner.h | Declares a CreateScanner function for main to use. |
| scanner\_p.h | Declares a scanner class that is not exposed outside the scanner (p means private). |
| ScopeTable.h | Declares a class that manages one scope of symbols. |
| Symbol.h | Declares the symbol types: VariableSymbol, ProcedureSymbol, param\_info. |
| SymbolTable.h | Declares a class that manages the stack of symbol scopes. This is what symbol table users see. |
| token.h | Declares the enum TokenType and Token class. |

The cpp files:

|  |  |
| --- | --- |
| genmips.cpp | Implements all the mips specific code. Writes the output asm file. |
| location.cpp | Adds a constructor and ToString to the Location struct. |
| main.cpp | Runs the scanner and parser on each filename passed on the command line. |
| parser.cpp | Implements the grammar as a recursive descent parser. Uses GetToken() to pull tokens in, calls into genmips to write out. Also tracks errors. |
| scanner.cpp | Does file handling for the scanner. Actual scanning is in tokenizer.cpp. |
| ScopeTable.cpp | Implements managing one scope's symbols. |
| Symbol.cpp | Implements the Symbol type. |
| SymbolTable.cpp | Implements managing all symbols. |
| token.cpp | Implements the token class. |
| tokenizer.cpp | Implements the DFA for the bada language, produces tokens. |

## The Flow

At the top level, main gets a filename from the command line and makes a scanner for it. That GetToken function is given to the parser. The parser does recursive descent, matching tokens according to the grammar and calling into the code generator when appropriate. The code generator accumulates output into two stringstreams: one for code and one for data. When the parsing completes successfully, the code generator writes prolog code, the data string, the code string, and epilog code. The parser writes the symbol table and returns errors. Main gets control back and displays the errors.

The biggest design flaw in the code is the symbol table being distinct from the code generator. Both the symbol table and the codegen pieces have similar methods for scopes opening and closing and the like.

## Implementation Details

The code generator has a function NextCodeLocation which is a counter returning unique numbers. Those get appended to code labels to ensure uniqueness. For example, with nested procedures you could have more than one function with the same name. NextCodeLocation fixes that.

bada allows procedure calls, so I have to take care of all the details of an activation record. One of the problems is the start of the function should reserve all the stack space that the function will use but it doesn't know how much that will be. Fortunately mips allows #defines, so the function entry code uses LOCALS\_XX to reserve the stack space. Later, when the function is completed, the value of LOCALS\_XX is known and gets written into the .data section.

The generated code is gloriously unoptimized. It takes the source one operation at a time.

## The Code Flaws

Logically I think the codegen should own the symbol table. Codegen creates locations and symbols need to have locations (and should be invariant). To further justify it, imagine a compiler that is producing output for two different targets at the same time. This should be possible but if the symbol table is independent of codegen then it is impossible.

The other sin I have to confess about the code is many of the comments are out of date. This is the danger of writing comments, especially when I am writing the comments before I understand what I'm doing. Any readers who have other thoughts, suggestions, feedback, tell me what you think: pwoodward@horizon.csueastbay.edu

1. https://www.visualstudio.com/downloads/ [↑](#footnote-ref-2)
2. https://github.com/chipsahoy/bada/tree/master/parser [↑](#footnote-ref-3)
3. http://spimsimulator.sourceforge.net/ [↑](#footnote-ref-4)