LazyComp Documentation

# What it is

LazyComp is a compiler from lazytongue to zASM.

# Architecture

LazyComp is written in C[[1]](#footnote-2). It consists of (f)Lex-generated tokenizer, Yacc/Bison-generated parser, own semantic analyzer and own code generator. Between analyzer and code generator lies intermediate representation (IR) code.

# Language

LazyTongue (LZ) is a natively compiled C-like language with classes. See Appendix A for a more detailed language description.

LZ source code is parsed in two passes: a declarative pass, and an imperative pass. First all the declarations and types are parsed, and then commands.

# Usage

LazyComp is called with the path to the main source file: “LazyComp.exe main.lz”. It will produce a zASM file “aout.txt”. The file can then be used in-game with the zCPU tool to compile the assembly and upload it to zCPU[[2]](#footnote-3).

# Compiling

LazyComp is compiled with Visual Studio 2019. There are also some old cmake/gcc files, but those are not maintained.

Code Documentation

# Main function

Processing fundamentally happens in three functions:

|  |
| --- |
| int main(){  yyparse(); //calls Bison parser, which calls Yacc lexer.  semantic\_analyze(); //builds a symbol table and outputs IR.  codegen\_generate(); //consumes IR and outputs zASM.  } |

# Tokenization

Tokenization happens inside the Flex lexer on-demand, controlled by the Bison parser. The blueprint for the lexer is in the file “yaccin.l”. The blueprint contains the definitions of tokens (their regex pattern and integer code), as well as extra code to save the character locations within the source file. The external program “Flex” is used to build the generated file “lexx.yy.c”.

# Syntax analysis

The Bison parses the tokens that it requests from Bison and builds a parse tree. The blueprint for the parser is in the file “yaccin.y”. The blueprint contains the grammar rules of the LZ language and the semantic actions to be done as soon as each production is fired. These semantic actions build the parse tree. The external program “Bison” is used to build the generated files “yaccin.tab.h” and “yaccin.tab.c”.

# Semantic analysis

The semantic analyzer visits every node of the parse tree and dispatches the appropriate sub-analyzer depending on the symbol at the parse tree node.

The semantic\_analyze() function is first called on the node that represents the entire program (entire source file) which calls semantic\_finalize() as the last action.

As nodes are walked, expression results and arguments are pushed through an external stack using push\_expr() and pop\_expr(). Values are stored in “temporary registers”, which are considered infinite in the IR.

The analysis function is dispatched based on node type, and then separates into “declarative pass / imperative pass” cases and then further by production number.

Some data that is relevant to specific node type analyzers is passed through the stack (e.g. the jump labels for the ‘if-block’)[[3]](#footnote-4).

There is also a stack for symbol tables / scopes, which are entered and left when going into functions and such.

IR code is emitted into “code sections” which are small code sections that can later be re-arranged and inserted into each other[[4]](#footnote-5).

Expressions – AST nodes that compute values – communicate the flow of values between them using value handles. Value handles describe how values are exchanged in the IR code. Value handles carry the IR name of a value (IR register, possibly with an accessor (& or \*)), the data type of the value (int, char, struct, etc), and the expression type – variable expressions (R-values or pointers) vs number expressions (L-values). Pointers carry the memory location which is permitted to be dereferenced to read or write to it. R-values are IR-expressions that are simply a dereferenced pointer (\*x). R-values cannot be stored in an IR-register as they stop being an R-value and decay into an L-value.

An expr node receives information about the value type they should produce (R/L/P). The node itself may perform the operations needed to obtain the needed value (emit conversion code or simply reinterpret the value), and then an “outputRes” function which can a) convert between R- and P- representation of a variable expression, b) decay a variable expression to a number expression (R- or P- to L) or c) pass the variable as is.

An expr node has two output channels: one of them is the IR-code emitted to manipulate the IR-values, and the other is the value handles given to the parent nodes so they node how to refer to the result in IR.

Semantic analysis creates several “code segments” that have IR code, and symbol tables. Before the analysis exits, it creates an IR code file by “flattening” the code segments and symbol tables, i.e. joining them into a single CS and a single ST.

# Code generation

Code generator reads IR code commands line-by-line from the one remaining code segment (flattened), which should be identical to what’s in the IR code file. It might as well be reading the “aout\_IR” file, but technically not.

Codegen looks up variables (by global IR name?) and then checks, which frame to look in. Frames are not actual function frames but rather scopes of visibility. The frame of a method should be inside the frame of a struct.

# IR code reference

Commands:

|  |  |  |
| --- | --- | --- |
| **Declarations** | | |
| SYMBOL | name VAR | PARAM | LABEL [ARRAY num] | **Defines** a **variable** or forward-declares a **label**. |
| FUNCTION | name | Begins a **function definition** |
| STRUCT | *name* | Begins a **structure definition**. |
| LABEL | name | **Defines** a **label** (jump destination) |
| BEGIN |  | Beginning of a code block for a function or struct definition (pairs with END) |
| END |  | End of a code block (pairs with BEGIN) |
| **Control flow** | | |
| RET | *value* | **Exits** the current function and **returns** *value* |
| JMP | *label* | Unconditional **jump** to a *label* |
| JNE | *val1* *val2* *label* | **Jump** to *label* if *value1* **!=** *value2* |
| JE | *val1* *val2* *label* | **Jump** if value1 **==** value2 |
| CALL | *res* *func* [*args*…] | **Calls** the function ***func*** with zero or more *args* and puts the result into *res*. |
| FRAME | ENTER | LEAVE | Enter or exit a new function stack frame. |
| **Operators** | | |
| MOV | *dest* *src* | **Copies** the value *src* into the value *dest* |
| ADD | *res* *val1* *val2* | **Adds** *val1* and *val2* and puts the result in *res*. |
| SUB | *res* *val1* *val2* | **Subtracts** *val2 from val1* and puts the result in *res*. |
| DIV | *res* *val1* *val2* | **Divides** *val1* by *val2* and puts the result in *res*. |
| MUL | *res* *val1* *val2* | **Multiplies** *val1* by *val2* and puts the result in *res*. |
| MOD | *res* *val1* *val2* | Computes the **remainder of division** of *val1* by *val2* and puts the result in *res*. |
| OR | *res* *val1* *val2* | Computes the **logical OR** between *val1* and *val2* and puts the result in *res*. |
| NOT | *res* *val1* | Computes the **logical NEGATION** of *val1* and puts the result in *res*. |
| NEG | *res* *val1* | **Negates** the **sign** of *val1* and puts the result in *res* |
| EQUAL | *res* *val1* *val2* | Checks if (*val1* **==** *val2*) and puts the result in *res*. |
| NOTEQUAL | *res* *val1* *val2* | Checks if (*val1* **=/=** *val2*) and puts the result in *res*. |
| GREATER | *res* *val1* *val2* | Checks if (*val1* **>** *val2*) and puts the result in *res*. |
| LESS | *res* *val1* *val2* | Checks if (*val1* **<** *val2*) and puts the result in *res*. |
| FLOOR | res val1 | Floors the value towards -inf, resulting in an integer |
| MIN | *res val1 val2* | Returns the smaller of two values |
| MAX | *res val1 val2* | Returns the larger of two values |
| **Special** | | |
| INSERT | *num* | generate an include instruction for another assembly file |
| COMMENT | GENERAL ...  SOURCE *str*  LINE *file:line* | A comment to be passed on to the assembly.  GENERAL: random stuff  SOURCE: contains source code line or expression  LINE: source file and line position |
| **Unused** | | |
| DEREF | *res* *val1* | (deprecated) dereferences val1 and puts the result in res (use \*val instead) |
| ALLOC | *num* | (deprecated) allocates *num* empty bytes of memory at the current code location. |
| USING | *name* | (deprecated) ?? consider symbols from *name* visible ? |
| DEBUG |  | (unused) Output debug information to stdout |

Values:

A value (formally IR\_symbol) can be of type: VAR, STRING, FUNC, LABEL, and STRUCT.

|  |  |
| --- | --- |
| VAR | A location in memory where values can be stored – this can be a local (stack) or global variable, a function argument or a struct member. |
| STRING | A pointer to a null-terminated string. The string itself is stored in code memory. |
| LABEL | A pointer to a code location. The value of a label need not be known when it is used. Label value becomes known when a LABEL command is used. |
| FUNC | A pointer to a function. Essentially a label, but with some more associated data. |
| STRUCT | A structure that contains variables and functions. |

Valid value-expressions (IR expressions) are “val”, “\*val”, and “&val”. An expression may have at most one accessor (\* or &). They determine what kind of memory access is used when reading and writing to these values.

|  |  |
| --- | --- |
| val1 = val2 | mem[val1] = mem[val2] |
| val1 = \*val2 | mem[val1] = mem[ mem[val2] ] |
| val1 = &val2 | mem[val1] = val2 |
| \*val1 = val2 | mem[ mem[val1] ] = mem[val2] |
| \*val1 = \*val2 | mem[ mem[val1] ] = mem[ mem[val2] ] |
| \*val1 = &val2 | mem[ mem[val1] ] = val2 |
| &val1 = anything | ERROR |

Appendix A. LazyTongue description

# Example code

|  |
| --- |
| int import(int arg) end  int export(int arg)  return arg;  end  int print(string str) end  class horse  float fluff = 5;  int hooves;  int boop()  fluff = fluff + 1;  end  end  int main(int argc, char\*\* argv)  char C;  int A;  class horse H;  if (argc == 1)  print("hello, I am %s"/\*, argv[0]\*/);  A = 5;  \*argv = "beep";  return 1;  else  float A = import(argc);  A = A \* argv;  export(A);  H.fluff = 10;  H.boop();  while (A)  print(A, argc);  A = A / 1;  if (A == 1) return 1; end  if (A < 3)  return 2;  elseif(A > 2)  return 3;  end  end  return 0;  end  string boop;  end |

# Grammar

|  |
| --- |
| program : decl\_stmt\_list ;  stmt : imp\_stmt | decl\_stmt ;  stmt\_list : stmt\_list\_ne | ;  stmt\_list\_ne : stmt\_list\_ne stmt | stmt ;  decl\_stmt : class\_def  | func\_def  | var\_decl ';'  ;  decl\_stmt\_list : decl\_stmt\_list\_ne | ;  decl\_stmt\_list\_ne : decl\_stmt\_list\_ne decl\_stmt | decl\_stmt ;  imp\_stmt : if\_block  | while\_loop  | expr ';'  | RETURN ';'  | RETURN expr ';'  | for\_loop  ;  class\_def : CLASS ID decl\_stmt\_list END ;  STAR : '\*' ;  ptr\_stars : ptr\_stars\_ne | ;  ptr\_stars\_ne: ptr\_stars\_ne STAR | STAR ;  typename : TYPE ptr\_stars | CLASS ID ptr\_stars ;  func\_def : typename ID '(' var\_decl\_list ')' stmt\_list END ;  var\_decl : typename ID  | typename ID '[' expr ']'  | typename ID '=' expr  ;  var\_decl\_list : var\_decl\_list\_ne | ;  var\_decl\_list\_ne : var\_decl\_list\_ne ',' var\_decl | var\_decl ;  if\_block : if\_then END | if\_then ELSE stmt\_list END ;  if\_then : IF '(' expr ')' stmt\_list  | if\_then ELSEIF '(' expr ')' stmt\_list  ;  /\*  if\_block : if\_start ELSE stmt\_list END  // | if\_start ELSE if\_block  | if\_start END  ;  if\_start : IF '(' expr ')' THEN stmt\_list ;  \*/  while\_loop : WHILE '(' expr ')' stmt\_list END ;  for\_loop : FOR '(' stmt expr ';' expr ')' stmt\_list END ;  expr\_list : expr\_list\_ne | ;  expr\_list\_ne : expr\_list\_ne ',' expr | expr ;  expr : ID  | INTEGER  | INTEGERX  | INTEGERB  | FLOATING  | CHARACTER  | STRING  | expr '[' expr ']'  | '(' expr ')'  | expr '(' expr\_list ')'  | expr '.' expr  | expr '^' expr  | expr '/' expr  | expr '\*' expr  | expr '%' expr  | expr '-' expr  | expr '+' expr  | '!' expr  | expr '&' expr  | expr '|' expr  | expr EQUAL expr  | expr NOTEQUAL expr  | expr '>' expr  | expr '<' expr  | expr '=' expr  | expr INC  | INC expr %prec PREINC  | expr DEC  | DEC expr %prec PREDEC  | '-' expr %prec PRENEG  | '\*' expr %prec PREDEREF  | '&' expr %prec PREREF  ; |

Note: grammar in the newest compiler version may be slightly different.

# C-like features of the language

## Statements

A statement can be a declaration/definition, a control block, or an expression. Statements end with either ‘;’ or ‘end’.

## Comments

C-style comments are supported. Comments are ignored by the compiler.

|  |
| --- |
| // this is a comment.  /\*  this is also a comment  \*/ |

## Literals

Code can contain literal values – integers, floating point, string or single-character literals.

|  |
| --- |
| 5  2.5  ‘c’  “string” |

## Arithmetic

Expressions with infix math operators can be written. Supported operators are the same as in C.

|  |
| --- |
| A = 2 + 5\*C; |

## Function calls

Expressions can contain function calls. Function signatures are strongly typed.

|  |
| --- |
| foo(); A = bar(2, 5, C) |

## Code blocks and scope

A block of code, that in C begins and ends with curly braces “{ … }”, in LZ begins implicitly at appropriate time (beginning of an if-block, for-loop, function definition or class definition) and ends with an end. There is no dedicated “begin block” command.

## Function declarations

Functions are declared as “type, name, list of typed arguments, body”:

|  |
| --- |
| float dot(vec2 A, vec2 B) return A.x \* B.x + A.y \* B.y end |

## Variable declarations

Variables are declared as “type, name” or “type, name, initializer”.

|  |
| --- |
| int A;  float B = 2.5; |

## Arrays

A variable can be an array.

|  |
| --- |
| char str[10]; |

## Pointers

A variable can be a pointer to another variable.

|  |
| --- |
| int \*p;  p = &A;  \*p = 5; |

## If-block

“if”, “if-else”, “if-elseif-else” blocks are supported.

|  |
| --- |
| if(A)  …  elseif(B > 10)  …  else  …  end |

## While loop

While loops are supported.

|  |
| --- |
| while(A)  …  end |

## For loop

For loops are supported.

|  |
| --- |
| for( int i = 0; i < 10; i++)  …  end |

## Return statement

A return statement can be used to end the execution of a function, and return a value to the place where the function was called from.

|  |
| --- |
| return 1; |

# Missing C features

## Continue, break stataments

Not yet implemented

## Struct definitions

Structs are currently not used as LZ has classes instead.

# Extra features

## Class definitions

It is possible to define classes and then use them as variables, arguments and return types:

|  |
| --- |
| class vec  float X  float Y  float len() return sqrt(X\*X+Y\*Y) end  end |

Classes can include variable and function declarations.

1. It is hoped that LazyComp will eventually be able to parse C and compile itself. [↑](#footnote-ref-2)
2. If external symbols are inserted into the assembly, then an external linker will be needed before the file is digestible by the zCPU tool. Some versions of Cyclone include a linker. CycloneBuilder might be extended to do this, though it expects HL-ZASM source code and doesn’t do much with raw zASM. [↑](#footnote-ref-3)
3. It may be a good idea to change this into actual, typed function arguments, but that would change the dispatch function api and necessitate the rewrite of how data flows between semantic analysis functions in general. [↑](#footnote-ref-4)
4. There are two modes of operation: with all code going into a single section, or into several ones. One or the other may be not working atm. [↑](#footnote-ref-5)