Compiler Project Overview

The project implements the core of a simple programming language called “Wabbit.” Wabbit supports the following features:

* Evaluation of expressions (math)
* Integers, floats, characters, and bools.
* Assignment to variables
* Printing
* Basic control flow (if, while)
* User-defined functions

Although the language is simple, the project still has to implement all of the core components of an actual compiler, including all of the parsing, type checking, control-flow analysis, intermediate code generation, and output code.

The implementation for the project was incremental. The first five projects go through all of the core stages of compilation which are lexing, parsing, type checking, and code generation for a small subset of the language. The last three projects involve expanding the compiler to include more advanced features.

Ultimately, the final output of the compiler will be Web Assembly, LLVM, or transpiled Python. These three targets give a rich flavor of what a compiler can do and how different things work.

A Taste of Wabbit

Here is a sample of a simple Wabbit program that computes the ever-so-useful Fibonacci numbers:

/\* fib.wb - Compute fibonacci numbers \*/

const LAST = 30; // A constant declaration

// A function declaration

func fibonacci(n int) int {

**if** n > 1 { // Conditionals

**return** fibonacci(n-1) + fibonacci(n-2);

} **else** {

**return** 1;

}

}

func main() int {

var n int = 0; // Variable declaration

**while** n < LAST { // Looping (**while**)

print fibonacci(n); // Printing

n = n + 1; // Assignment

}

**return** 0;

}

The fib.wb program above can be found in the directory Programs/fib.wb.

Running and Compiling Programs

Ultimately the wabbit compiler will allow programs to be compiled and executed in a variety of ways. First, you’ll be able to transpile Wabbit source to a Python script.

bash % python3 -m wabbit.python Programs/fib.wb bash % python3 out.py 1 1 2 3 5 8 13 21 34 55 … bash %

You should see output similar to the above being generated, albeit very slowly. This is the most portable technique for running Wabbit code as it involves nothing but pure Python code.

The fib.wb program can also be compiled to a stand-alone executable:

bash % python3 -m wabbit.compile Programs/fib.wb

bash % ./a.out

1

1

2

3

5

...

bash %

This creates a program called a.out. If you run it, you should see the same output as the Python code, but substantially faster. This is executing native machine code on your system. Creating this executable requires the clang C/C++ compiler. If you don’t have it installed correctly, compilation will probably fail.

If you don’t have clang installed, you can also run the program as a just-in-time compiled binary inside Python. To do this, you first need to build a run-time library:

bash % cd wabbit

bash % make mac *# Change to make linux on Linux*

Next, you can try running the program:

bash % cd ..

bash % python3 -m wabbit.run Programs/fib.wb

1

1

2

3

5

...

bash %

Both the *compile* and *run* options use LLVM to generate native machine code.

The final target for Wabbit is Web Assembly. You can create a *.wasm* file as follows:

bash % python3 -m wabbit.wasm Programs/fib.wb

This creates a file *out.wasm*. To run this program in the browser, launch a web server:

bash % python3 -m http.server

Next, got to your browser and load <http://localhost:8000/test.html>. You should see the output the program appearing on a web page. Refer to the file *test.html* to see how it’s done.

Playground

The SillyWabbit/ directory contains a full implementation of the language interpreter in a form where you can experiment with Wabbit programs and see what they do. See the README file in that directory for more information.

Wabbit Language Specification[¶](file:///C:\work\compilers_old\doc\html\Wabbit.html#wabbit-language-specification)

This document serves as a reference to the Wabbit language which you will be compiling.

1. Lexical Conventions and Syntax

Each statement is terminated by a semicolon. For example:

print 3;

var a int = 4;

A single-line comment is denoted by //. For example:

var a int = 4; // This **is** a comment

Multiline comments can be written using /\* ... \*/. For example:

/\*

This **is** a multiline

comment.

\*/

An identifier is a name used to identify variables, constants, and functions. Identifiers can include letters, numbers, and the underscore (\_), but must always start with a non-numeric character. Wabbit follows the same rules as Python. The following reserved words may not be used as an identifier:

const **else** **import** **false** func **if** print **return** true **while** var

A numeric literal such as 12345 is intepreted as an integer. A numeric literal involving a decimal point or scientific notation such as 1.2345 or 123e2 is interpreted as a floating point number. The literals true and false are interpreted as booleans.

A character literal such as *‘h’* is interpreted as a single text character. Escape codes such as \', \n, \\, and \xhh are to be interpreted in the usual way as well.

The following operators are recognized:

+ - \* / < <= > >= == != ! && || ++, --

The following tokens serve as delimiters in expressions, function calls, and function definitions:

( ) , { }

Curly braces are used to enclose blocks of statements. For example:

**if** (a < b) {

statement1;

statement2;

} **else** {

statement3;

statement4;

}

2. Types

There are four built-in datatypes; int, float, byte, and bool.

int is a signed 32-bit integer. float is a 64-bit double precision floating point number. char is a single character, represented as a byte. bool represents the boolean values true and false.

Variables are always declared with an explicit type and may include and optional initializer. For example:

var a int;

var b float = 3.14159;

var c bool;

var d char = 'h';

Constants may be declared without a type, in which case the type is inferred from the value:

const e = 4; // Integer constant

const f = 2.71828; // Float constant

const g = true; // Bool constant

const h = 'h'; // Char constant

3. Operators and Expressions

3.1 Numeric operators

Numeric types support the binary operators +, -, \*, and / with their standard mathematical meaning. Operators require both operands to be of the same type. For example, x / y is only legal if x and y are the same type. The result type is always the same type as the operands. Note: for integer division, the result is an integer and is truncated.

Numeric types also support the unary operators of + and -. For example:

z = -y;

z = x \* -y;

No automatic type coercion is performed. Thus, integers and floats can not be mixed together in an operator. If this is desired, one of the values needs to be explicitly converted to the other using a conversion function (unspecified) of some kind.

3.2 Character operations

Character literals support no operations whatever. A character is simply a “character” and that’s it.

3.3 Relations

The operators <, <=, >, >=, ==, and != can be used to compare two values and have the standard meaning found in Python. The two operands must be of the same type.

The logical operators &&, ||, and ! implement the logical and, logical-or, and logical negation operations. These operators only work on boolean values. Thus, the following expressions are legal:

(a < 3) && (a > 0)

!(a == 0)

Expressions such as the following are illegal unless a and b are of type bool:

a && b; // Illegal unless a,b are bools

Although character literals support no mathematical operators, they are comparable.

3.4 Boolean types and operators

Boolean types only support the operators ==, !=, &&, ||, and !. In particular, boolean values are not equivalent to integers and can not be used in mathematical operators involving numbers.

3.5 Associativity and precedence rules

All operators are left-associative. The following chart shows the precedence rules from highest to lowest precedence:

+, -, ! (unary) // Highest precedence

\*, /

+, -

<, <=, >, >=, ==, !=

&&

|| // Lowest precedence

Relational operators may NOT be chained or associate together. For example:

a < b && b < c; // OK

a < b < c; // Illegal

3.6 Short-circuit evaluation

The logical operators && and || should implement short-circuit behavior in evaluation. That is, in the expression a && b, if a evaluates to false, then b is not evaluated. Similarly, if a evaluates to true, then a || b does not evaluate b.

4. Control Flow

The if statement is used for conditions. For example:

**if** (a < b) {

statements;

...

} **else** {

statements;

...

}

The conditional expression used to test must evaluate to a bool. Code such as the following is an error unless a has type bool:

**if** (a) { // Illegal unless a **is** type bool

...

}

The else clause is optional.

The while statement can be used to execute a loop. For example:

**while** (n < 10) {

statements;

...

}

This executes the enclosed statements as long as the associated condition is true. Again, the conditional expression must evaluate to type bool.

The break statement can be used to break out of a loop early. For example, this code only prints the numbers 0, 1, …, 4:

var n int = 0;

**while** (n < 10) {

statements;

**if** (n == 5) {

**break**;

}

print n;

n = n + 1;

}

The continue statement can be used to jump back to the top of a loop, ignoring the remainder of the loop body.

5. Functions

Functions can be defined using the func keyword as follows:

func fib(n int) int {

**if** (n <= 2) {

**return** 1;

} **else** {

**return** fib(n-1) + fib(n-2);

}

}

Functions must supply types for the input parameters and return value as shown.

External functions can be declared using import as follows:

**import** **func** sin(x float) float;

These functions must already exist somewhere in the runtime environment. How that actually happens might be resolved by the linker or loader. The exact details are not our concern.

When calling a function, all function arguments are fully evaluated prior to making the associated function call. That is, in a call such as foo(a,b,c), the arguments a, b, and c are fully evaluated to a value first.

6. Scoping rules

Declarations are placed into one of two scopes. Declarations defined outside of a function are global. Declarations inside a function are local. Local declarations are not visible to any other part of a program except for code in the same function. Statements inside a function can access declarations in local or global scope. For example:

var a int; // Global variable

func foo(b int) int {

var c int; // Local variable

...

}

Nested function definitions and closures are not supported. For example:

func foo(b int) int {

func bar(c int) int { // Illegal. Nested functions **not** allowed

...

}

...

}

7. Main entry point and initialization

Programs always begin execution in a function main() which takes no arguments and returns an integer result. For example:

func main() int {

var i int = 0;

**while** (i < N) {

print fib(i);

i = i + 1;

}

**return** 0;

}

Any initialization steps related to global variables must execute prior to the invocation of main(). For example:

var a int = 4;

var b int = 5;

var c int = a + b; // Evaluates prior to main()

...

func main() int {

...

}

If there is no main() function, any kind of “scripting” statements will still execute as part of the initialization step. Your compiler should emit a dummy main() function in this case.

8. Printing

The built-in print value operation can be used for debugging output. It prints the value of any type given to it. Values are normally printed on separate lines. However, if you print a single character value, it is printed with no line break.

9. Memory Access

No programming language is complete without some kind of crazy feature and Wabbit has just that in the form of direct memory access. Memory is an array of bytes and memory addresses are integers. To access memory, use the backtick (`) operator on any integer value. For example:

`128 = 45; // Save the int 45 at memory address 128

`128 = 4.5; // Save the float 4.5 at memory address 128

`128 = 'x'; // Save the byte 'x' at memory address 128

This also works in loops and with expressions. This is how you’d write out some array values into memory starting at address 1000:

var addr int = 1000;

var n int = 0;

while n < 100 {

`(addr + n\*4) = n;

n = n + 1;

}

If you want to read from memory, use the backtick as an expression. Here’s how you could total up the above numbers:

var total int = 0;

var addr int = 0;

var n int = 0;

while n < 100 {

total = total + `(addr + n\*4);

n = n + 1;

}

print total;