CHIL

CSS HTML Integrated Language

Programming Languages and Translators Fall 2013

Authors:
Gil Chen-Zion gc2466
Ami Kumar ak3284
Annania Melaku amm2324
Isaac White   iaw2105

Professor:
Prof. Stephen A. Edwards

Contents

Chapter 1

An Introduction to CHIL

The CHIL language is framed to help developers swiftly generate websites. CHIL allows the developer to create functional, stylistic, and dynamic web pages through the mastery of only one language. This language will combine both content and styling into a single language.

CHIL is designed to be a simple, efficient, abstracted mark-up language that is specifically aimed toward the novice developer. The efficient aspect allows developers to simultaneously define structures and styling, where styling is scoped within the structure. Furthermore, CHIL provides the ability to programmatically loop through different types of element creation, or define more complex elements made up of basic types provided by the language.

1.1 Background

Web design and programming are currently severely separated from one another by a divide in both skill set and interests. Web design methods, particularly HTML and CSS, are only capable of producing custom written pages according to the current web standards, which constrains programmers that are accustomed to writing extensible code which can be reused. Though there are some current solutions to solve this problem (php for example) most solutions don't really solve the problem of reconciling the duality of css with more standard programming naming conventions or design patters (ie, css properties use dashes, which wouldn't be allowed in most programming languages for variable names).

CHIL was created to try to bridge this gap. CHIL tries to simply model all possible html elements and styles by defining them in general element objects that can be nested to emulate HTML divs with greater abstraction. CSS properties can be applied to these abstracted elements either through pure css strings which will be placed into a style section verbatim, or through user defined properties that have definitions with greater abstraction to benefit the programmer.

The program aims to be different from PHP in the way that it both allows the programmer to pair and define CSS styles programmatically, but also to have more abstract control over the rendering of the page. CHIL is designed to allow for the abstraction away from specific tags that would be used in HTML instead to a single general type, element, that has properties of type style, to represent applied CSS.

1.2 Goals of CHIL

CHIL allows a programmer with little knowledge of HTML or CSS to create webpages with advanced features the would otherwise not access. It enables programmers to approach web design from a programming mentality and create reusable programs to design websites. The language aims to simplify HTML programming by replacing the use of tags by using general element type. In addition, programs in the language can include functions, types, and looping, all impossible in HTML.

Chapter 2

Language Tutorial

CHIL essentially consists of `element`s with properties defined by the programmer. These `element`s are added to the global `Page` object via the `add` function and rendered to an html file with the built in `toHtml` function, which can then be displayed on a browser. Other variables and functions can be declared and used to assist in building the web page.

2.1 How To

2.1.1 Primitve Types

For how to implement primitive types, see Chapter 3 Language Reference Manual

2.2.2 Syntax

2.2.2.1 Type Casting

Datatypes:

int	*integer*
string	*string*
el	*element*
float	*float*
style	*style*

All datatypes need to be type casted:

```
int i = 3

string s = “hello”

float f = 3.0

el e = {
style: ${ @ … css style properties … will be ouputed as css
}
}
```

*See Sample Code

2.2.2.2 Features

Loops
Comments
Whitespace

Loops

While Loops

```
while (expr)
      @ body of while loop
endwhile
```

*See Chapter 3

For Loops

```
int i = 0;

for (i = 0; i < 4; i++)
      @ body of loop
endfor
```

Comments

Singleline
```
      @ A singleline comment
```
Multiline
```
      @> A multiline
            comment <@
```

Whitespace
*See Chapter 3 for more information

2.2.2.3 Built-In Functions

toHTML
stf
sti
fti
its
fts

toHTML
A function automatically utilized during compile which assigns elements their tag type, the CHIL programmer can

override the method to create a Page (tag types) that they want.

stf
```
        float x = stf("3")
```
sti
```
        int x = sti("8")
```
fti
```
        float x = fti(3.0)
```
its
```
        string x = its(3)
```

fts
```
        string x = fts(3.0)
```

*See Chapter 3 for function definitions

2.2.2.4 Conditionals
If
Else

```
if (condition)
      @ body of if
      else
endif
```

2.2.2.5 Reserved Words
Page.add

```
Page.add({
      @ body of Page.add
})
```

2.2 First Example

The first program we wrote and tested in our language was Hello World. This consisted of adding an `element` to the global `Page` type. Each `element` has properties `contents` and `style`. In this case, the `contents` property includes the `string` "Hello, World!" which is printed out on the webpage. Below is code for the "Hello, World!" program.

```
Page.add({
      contents: "Hello, World!"
})

@ Prints some htmlWrapper + <p>Hello, World!</p>
```

2.2 Compiling and Running

In order to compile this code, you would write:

```
$ chil test.ch
```

This creates an `html` file with the code for the website you want to create. You can view this website by opening the file in a browser.

2.3 More Examples

CHIL also supports arithmetic operations. Programmers can define variables of type `int`, `string`, and `float` and manipulate them by concatenation and other arithmetic operators. In the following example, an integer `totalValue` is declared and set to 42. Then, an `element`, `theAnswer`, is defined with `contents totalValue` and some `style` specifications, like having a bold font or a black background.

```

int totalValue = 20 + 22

el theAnswer = {
      contents: totalValue,
      style: ${
            css: "font-weight: bold; font-family: arial; font-size: 2rem;
color: white; background-color: black; display: block; box-sizing:
border-box; padding: .5rem; border: 1px dotted white; margin: 1rem;"
      }
}

Page.add(theAnswer)
```

The program produces the following html code:

```
<html>
<head>
<title>Addition</title>
<style type="text/css">
.theAnswer {font-weight: bold; font-family: arial; font-size: 2rem; color:
white; background-color: black; display: block; box-sizing: border-box;
padding: .5rem; border: 1px dotted white; margin: 1rem;}
</style>
</head>
<body>
<p class="theAnswer">42</p>
</body>

</html>
```

The webpage produced has a “42” on the top left of the screen, surrounded by a black rectangle.

function.ch

This program prints three paragraphs on an HTML page.

```
fn testFunc(string param)

      el someElement = {

                  contents: param

      }

      rtn someElement

endfn

el x = testFunc("element 1")

el y = testFunc("element 2")

el z = testFunc("element 3")

Page.add(x)

Page.add(y)

Page.add(z)
```

This program produces the following code:

```
<html>
<head>
<title>Function</title>
</head>
<body>
<p class="someElement">”element 1”</p>
<p class="someElement">”element 2”</p>
<p class="someElement">”element 3”</p>
</body>

</html>
```

Chapter 3

Language Reference Manual

3.1 LEXICAL CONVENTIONS

3.1.1 Comments

Multi-line comments begin with @> and end with <@. A single-line comment is preceded by @. The nesting of comments is not allowed.

3.1.2 Primitive Types

Key Word	**Description**
`int`	non-decimal number
`string`	sequence of characters
`float`	decimal number
`boolean`	true or false value
`element`	fundamental container that can contain text or images

`style`	characteristics of an element
`array`	list of primitives of a single type

3.1.2.1 Integers

A data type consisting of whole number values.

Examples:
```
int x = 5
int y = 0
int z = -20
```

3.1.2.2 Strings

A constant representing character strings.

Examples:
```
string x = "Hello World"
string y = "abc"
```

String Concatenation

Strings are combined using + operator.

Examples:
```
x = "Hello"
y = "World"
z = x+y            ("Hello World")
or
z = "Hello" + "World"
```

3.1.2.3 Floats

A numeric data type containing a decimal. Examples of floats are:

```
125.3
3.0
3.
```

3.1.2.4 Booleans

Booleans are incorporated in the use of boolean operators. The use of a boolean operator in comparison expressions return one of two predefined constants: true or false.

Boolean operators used in comparisons:

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

Examples:

```
x = 3
y = 3
z = 1

x == y       (evaluates to true)
x > y        (evaluates to false)
x > z        (evaluates to true)
z != y       (evaluates to true)
y => x       (evaluates to true)
```

3.1.2.5 Elements

Elements are the fundamental container for any object which is intended to eventually be rendered as HTML. An element's attributes determine which tag will eventually be associated with it.

Example:

```
someElementName = {
      contents: "I'm an element.",
      style: ${
      nameDeclaredByProgram: someValue
      },
      children:
      @> children not provided here, since it often makes sense to
      define them as another variable or add them on to defined
      elements later via the someElementName.children[] = operator. <@
}
```

3.1.2.6 Styles

All elements have a property called style, which is provided without any allowed values. Page.addStyle() can be used to add possible values to the list for allowed properties, which will tell the compiler that the keywords are permitted.

Styles are processed by by the Page.toHtml() method, which the programmer is responsible for providing. If no Page.toHtml() method is set, then the compiler will use a provided fallback that is not accessible to the programmer. If Page.toHtml() produces invalid output, the compiler will also fallback to the built in, inaccessible method. Style is defined at the time of Element declaration, but styles may be added by referencing Element.style.someStyleName. Style elements are declared with ${} syntax, using the equals operator.

Examples:

```
@>In this example, keywords have been previously provided by a standard
library<@

3pxBorderTop = ${
      border: "top,3px,solid",
      innerSpacing: "top, .25",
      outerSpacing: "top ,.25"
}
```

3.1.2.7 Arrays

An array is a list of objects or primitive types. All elements of an array must be of the same type. Arrays can be concatenated by using the + operator. Any array that is used in an addition operation will take precedence for returning the output as an array. However, since arrays may contain only a single type, this operation may fail if not used with matching types. Values can also be added to the end of the array using `varName[] = someVal`

Examples:

```
r = []                          (initialize an empty array)
x = ["hello", "world"]          (array of Strings)
y = [42,34,42]                  (array of type int)
r[] = 34 @add 34 to the end of the r array
s = [42] @make a new array called s
z = r + s

@> z = [34,42] <@
```

3.1.3 Identifiers

Identifiers consist of uppercase and lowercase letters, numbers, and underscore (_). The first letter cannot be a digit or an underscore (_). Two different identifiers cannot have the same name.

3.1.4 Keywords

```
element
style
page
if
endif
else
elseif
for
endfor
rtn
fn
endfn
Page
```

3.2 EXPRESSIONS

3.2.1 Operators

Function Name	**Description**
`=`	assignment operator
`++,--`	increment and decrement
`+, -, *, /`	basic arithmetic operators
`+=, -=, *=, /=`	other arithmetic operations
`~`	power
`!`	factorial
`==, !=`	comparison operators, by value
`<, <=, >, >=`	inequality operators, by value
`%`	modulus
`“”`	denotes a string
`(,)`	contains an expression
`&&, ||`	logical operators

3.2.2 Increment Operators

Increment and decrement are unary operators that work on a single integer. The increment operators follow an integer value expression, as follows:

```
expr++
expr--
```

`++`	increment
`--`	decrement

3.2.3 Multiplicative Operators

The multiplicative operators are `*` and `/`. Both can be used on integer or float value expressions. Multiplication and division operators are formatted as follows:

```
expr * expr
expr *= expr
expr / expr
expr /= expr
```

`*`	multiplicaton
`*=`	multiplication
`/`	divison
`/=`	division

3.2.4 Additive Operators

The additive operators are + and `-`, and are also used on integer or float value expressions. The format of addition and subtraction is similar to that of multiplication and division.

```
expr + expr
expr - expr
expr += expr
expr -= expr
```

`+`	addition
`+=`	addition
`-`	subtraction
`-=`	subtraction

3.2.5 Inequality Operators

Inequality operators compare two integer or two float expressions. The inequality operators are `<, >, <=, and` >=. An expression with an equality has the following format:

```
expr < expr
```

`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

3.2.6 Comparison Operators

Two integer or float operands can be compared by the operators == and `!=`. For example:

```
expr == expr
expr != expr
```

==	equal to
`!=`	not equal to

3.2.7 Logical Operators

The two logical operators are `&&` and `||`. Two boolean value expressions can be combined by either AND or OR as follows:

```
(3 < 4) && (4 < 5)            evaluates to true
(3 > 5) || (6 != 7)           evaluates to false
```

`&&`	and
`||`	or

3.2.8 Precedence

The following list shows the precedence of all the operators, from highest to lowest precedence:

```
Parenthesis
Increment operators
Multiplicative operators
Additive operators
Inequality operators
Comparison operators
Logical operators
```

3.3 FUNCTIONS

3.3.1 Function Call

Functions will be called by their identifier, with arguments specified in parenthesis and separated by commas. Member functions will be called after an object and separated by a period. For example, for the function “func”:

```
func(1, 2)
a.func(1, 2)
```

3.3.2 Recursion

Recursive functions cannot be defined. No function can call itself recursively.

3.3.3 Built-in Functions

`toHtml()` converts the program into HTML. If no `toHtml()` is defined, a default `toHtml()` is used. Otherwise, the new `toHtml()` is used for the conversion.

*for functions to convert types see Section 7. Type Conversions

3.4 DECLARATIONS

3.4.1 Variable Declaration

Variables are declared by their type and identifier and assigned values using =. Integers just include digits and floats contain a single decimal point. Strings are identified by opening and closing quotes and arrays are denoted by opening and closing square brackets. The following are examples of variable declarations:

```
string a = “Hello World”
int b = 42
string[] c = [“hello”, “world”]
int[] d = [42,34,42]
string[] e = [a]
```

3.4.2 Function Declaration

Functions are declared by the keyword `fn` and an identifier, which is used to refer to the function throughout the program. Parameters are passed to the function in parenthesis and separated by commas. The body of a function is closed by the `endfn` keyword. The format of a function declaration is as follows:

```
fn identifier(parameter list)
      @ body of function
endfn
```

3.4.3 Scoping

Variables declared within the program scope can be accessed by all functions in the file. Variables declared within a statement (`if` or `for`) or function have a scope that lasts within the body of the statement or function, starting at the variable declaration and ending at `endif`, `endfor`, or `endfn`.

3.5 STATEMENTS

3.5.1 Conditional Statement

An if statement begins with the `if` keyword and ends with `endif`. If is followed by a condition contained in parenthesis. The body of the if statement is followed by endif, as follows:

```
if (condition)
      @ body of if
endif
```

3.5.2 For Loop

A for loop starts with `for` and ends in `endfor`. `for` is followed by three expressions enclosed in parenthesis and separated by semicolons. The first expression is evaluated before the body of the for loop. The body of the loop is executed whenever the second expression evaluates to true. The last expression is evaluated after each iteration of the loop. The format of the for loop is as follows:

```
for (expr; expr; expr)
      @ body of loop
endfor
```

3.5.3 While Loops

The while loop starts with the reserved word `while` and ends with `endwhile`. An `expr` enclosed in parenthesis follows the `while`. This `expr` is evaluated before the start of the loop and before every iteration of the loop. The body of the loop is only executed if the `expr` evaluates to `true`.

```
while (expr)
      @ body of while loop
endwhile
```

3.5.4 Return Statements

All functions return a data type using the `rtn` keyword. If the keyword is not included the function will return `undefined.`

```
fn identifier(parameter list)
      @ body of function
      @ optional return statement
      rtn type
endfn
```

3.6 TYPE CONVERSIONS

The following functions take one argument of the type they are converting from and return a value of the type they are converting to:

`stf`	string to float
`itf`	int to float
`fti`	float to int
`sti`	string to int
`its`	int to string
`fts`	float to string

3.7 WHITESPACE

Whitespace in CHIL includes tabs, spaces, and comments.

Chapter 4

Project Plan

4.1 Team Responsibilities

All members of the team helped with coding and debugging. Annania and Ami also worked on the report, while Gil and Isaac took care of testing.

Annania	**coding, debugging, report, powerpoint**
Ami	**coding, debugging, report**
Gil	**coding, debugging, testing, powerpoint**
Isaac	**coding, debugging, testing**

4.2 Project Timeline

9-16	Preliminary idea and basic functionality of language decided
9-20	Code conventions and language properties defined
9-25	Complete proposal
10-28	Complete Language Reference Manual
11-17	Complete Scanner & Parser
12-1	Complete Ast, Interpreter
12-9	Complete Bytecode, Makefile
12-13	Write Test Cases
12-18	Complete Final Report

4.3 Software Development Environment

Different team members used different software environments to develop this project. Some of us used sublime for text editing, while others used vim. Similarly, we ran our programs either in Unix or Ubuntu. Also, we used the git version control system to share our files and keep track of modifications.

4.4 Project Log

9-16	Preliminary idea and basic functionality of language decided
9-20	Code conventions and language properties defined
9-25	Project Proposal completed
10-28	Language Reference Manual completed
11-3	Project initiated, syntax & types
11-10	Parser & Scanner & understanding microc
11-17	Ast, Interpreter, started
11-24	sample Test Cases written
12-1	Execute, Compile, Bytecode started
12-10	Testing & type casting
12-17	Final Report, Parser, Interpreter, Ast, Execute, Compile, Bytecode
12-18	Final Report, Parser, Interpreter, Ast, Execute, Compile, Bytecode

12-19	Final Report, Parser, Interpreter, Ast, Execute, Compile, Bytecode
12-20	PowerPoint, Final Report, Parser, Interpreter, Ast, Execute, Compile, Bytecode

4.6 Style Guide

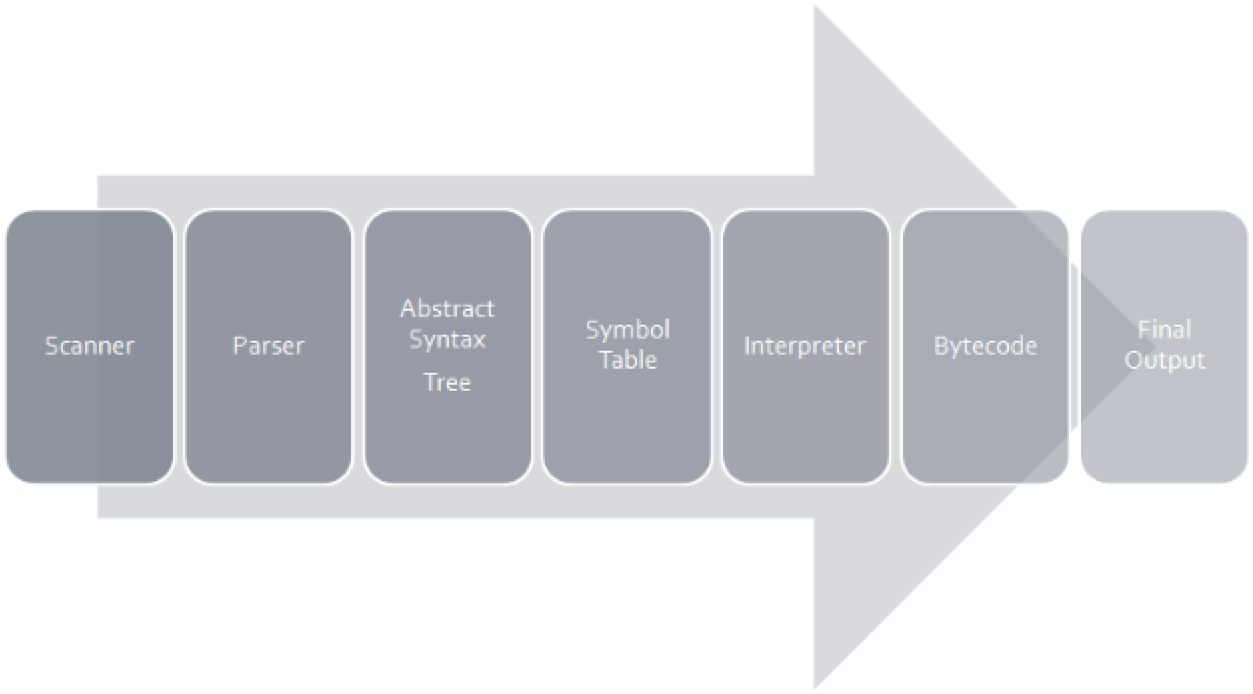
Guidelines followed as closely as possible while programming our compiler:
• Formatting and indents: tabs were used for scoping, methods were spaced out, within files names were defined before use
• Comments and documentation: We preceded each method ofwith Ocaml comments describing the code below.

Chapter 5

Architectural Design

5.1 Architecture

The architecture of our compiler includes a scanner, parser, abstract syntax tree, symbol table, interpreter, intermediate bytecode, and final output. The scanner is a OCamllex file, the parser is a OCamlyacc file, and the rest of the files are OCaml files. The scanner is in charge of lexical analysis of the file and separates the input stream from the program into tokens. The parser then looks for semantic meaning in the tokens. Meanwhile, it continually updates the symbol table. The abstract syntax tree contains the grammar of the language, which is unambiguous. The interpreter defines how to evaluate statements and keeps track of the environment and side-effects while doing so. The bytecode file takes the code and puts it in an intermediate bytecode. This bytecode eventually leads to the final output of our compiler: an html file.



5.2 The Runtime Environment

Once the a CHL program is written and set to compile, the file is initially loaded. The compiler starts form the first available statements and traverses the file sequentially rendering to a global `Page` state every time the `add` method is called. The `Page` is finally output as an html file that can be viewed via a web browser.

5.3 Error Recovery

Error recovery in CHIL involves basic syntactical error handling. If a programmer's code has any semantic or syntactic inconsistencies with our language definitions, the compiler will throw an error and give the programmer a brief insight on where the error could be or logically what type of error it is. For example, if a function is called but not defined, our compiler gives a message saying, "undefined function," followed by the name of the function. This not only tells the programmer the nature of the problem but also the exact location of the mistake.

Chapter 6

Test Plan

6.1 Goals

Our testing plan, after having a code that compiled in OCaml, was to test as many aspects of our language as possible in the smallest units we could in order to isolate what worked and what did not. By testing one thing at a time, we could isolate errors and pinpoint specific incorrect parts in our program. Our goal was to have as much properly working functionality as we could in our language in order to achieve its specific purpose.

6.2 Hypothesis

We hypothesized the most difficult part of the project would be parsing the `string`s passed to the `style` properties and types and converting them to html tags for our final output. However, most of our difficulties came from adding types, other than `int`s, to our compiler.

6.3 Methods

Our first phase of testing occurred concurrent to our coding phase. After implementing each additional functionality in our code, we ran test cases specific to that aspect of our language. At first, we focused on getting our test cases to compile.

6.3.1 Phase I

We first implemented our style of variable and function declaration, in addition to a basic integer type and basic arithmetic operators. Our tests consisted of basic variable declarations within a function, then calling the function after.

6.3.2 Phase II

Then, we added more operators, in addition to the basic four: add, subtract, multiply, and divide. We first defined an increment and decrement, and then included a modulus. We implemented conditionals and loops according to our syntax. We tested increment in a for loop and tested the basic arithmetic operators within variable assignment.

6.3.3 Phase III

In phase III, we attempted to implement additional types in our language, like strings, floats, booleans, and elements.

6.4 Tools

In order to run our tests, we had a testall file. We also individually ran tests when we were checking for specific functionality.

6.5 Implementation

Our implementation phase consisted of our html code generating phase.

6.5.1 Phase I

Our first goal with testing was to get “Hello, World!” to compile and send to an html file to be viewed on a browser. Getting a basic program running would mean our functionality to generate html files was coded properly.

6.5.2 Phase II

Phase II consisted of displaying other variable types on a webpage, in addition to adding style elements, like background color and font.

6.5.3 Phase III

Our final development phase was to create more complicated test cases incorporating all of the elements described above.

6.5 Sample Program

*See Chapter 2

6.6 Automation

We used a Makefile and a testall.sh that allowed us to simulanteously test all cases based on microc but adapted for CHIL testing.

Makefile

```
OBJS = ast.cmo parser.cmo scanner.cmo interpret.cmo bytecode.cmo compile.cmo
execute.cmo chil.cmo

TESTS = \
arith1 \
arith2 \
arith3 \
arith4 \
cool \
cooler \
fib \
for1 \
func1 \
func2 \
func3 \
func4 \
mod \
gcd \
global1 \
hello \
if1 \
incdec \
if2 \
if3 \
if4 \
ops1 \
var1 \
var2 \
var3 \
stmts1 \
while1

TARFILES = Makefile testall.sh scanner.mll parser.mly \
        ast.ml bytecode.ml interpret.ml compile.ml execute.ml chil.ml \
        $(TESTS:%=tests/test-%.mc) \
        $(TESTS:%=tests/test-%.out)

chil : $(OBJS)
        ocamlc -o chil $(OBJS)

.PHONY : test
test : chil testall.sh
        ./testall.sh

scanner.ml : scanner.mll
        ocamllex scanner.mll

parser.ml parser.mli : parser.mly
        ocamlyacc parser.mly

%.cmo : %.ml
        ocamlc -c $<

%.cmi : %.mli
        ocamlc -c $<

chil.tar.gz : $(TARFILES)
        cd .. && tar czf chil/chil.tar.gz $(TARFILES:%=chil/%)

.PHONY : clean
clean :
        rm -f chil parser.ml parser.mli scanner.ml testall.log \
        *.cmo *.cmi *.out *.diff

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
bytecode.cmo: ast.cmo
bytecode.cmx: ast.cmx
compile.cmo: bytecode.cmo ast.cmo
compile.cmx: bytecode.cmx ast.cmx
execute.cmo: bytecode.cmo ast.cmo
execute.cmx: bytecode.cmx ast.cmx
interpret.cmo: ast.cmo
interpret.cmx: ast.cmx
chil.cmo: scanner.cmo parser.cmi interpret.cmo execute.cmo compile.cmo \
    bytecode.cmo ast.cmo
chil.cmx: scanner.cmx parser.cmx interpret.cmx execute.cmx compile.cmx \
    bytecode.cmx ast.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```

To run make:

```
make
```

or

```
make test
```

testall.sh

```
#!/bin/sh

CHIL="./chil"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.mc files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
        echo "FAILED"
        error=1
    fi
    echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to
difffile
Compare() {
    generatedfiles="$generatedfiles $3"
    echo diff -b $1 $2 ">" $3 1>&2
    diff -b "$1" "$2" > "$3" 2>&1 || {
        SignalError "$1 differs"
        echo "FAILED $1 differs from $2" 1>&2
    }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
    echo $* 1>&2
    eval $* || {
        SignalError "$1 failed on $*"
        return 1
    }
}

Check() {
    error=0
    basename=`echo $1 | sed 's/.*\\///
                             s/.ch//'`
    reffile=`echo $1 | sed 's/.ch$//'`
    basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

    echo -n "$basename..."

    echo 1>&2
    echo "###### Testing $basename" 1>&2

    generatedfiles=""

    generatedfiles="$generatedfiles ${basename}.i.out" &&
    Run "$CHIL" "-i" "<" $1 ">" ${basename}.i.out &&
    Compare ${basename}.i.out ${reffile}.out ${basename}.i.diff

    generatedfiles="$generatedfiles ${basename}.c.out" &&
    Run "$CHIL" "-c" "<" $1 ">" ${basename}.c.out &&
    Compare ${basename}.c.out ${reffile}.out ${basename}.c.diff

    # Report the status and clean up the generated files

    if [ $error -eq 0 ] ; then
        if [ $keep -eq 0 ] ; then
```

```
            rm -f $generatedfiles
        fi
        echo "OK"
        echo "###### SUCCESS" 1>&2
    else
        echo "###### FAILED" 1>&2
        globalerror=$error
    fi
}

while getopts kdpsh c; do
    case $c in
        k) # Keep intermediate files
            keep=1
            ;;
        h) # Help
            Usage
            ;;
    esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/fail-*.ch tests/test-*.ch"
fi

for file in $files
do
    case $file in
        *test-*)
            Check $file 2>> $globallog
            ;;
        *fail-*)
            CheckFail $file 2>> $globallog
            ;;
        *)
            echo "unknown file type $file"
            globalerror=1
            ;;
    esac
done

exit $globalerror
```

Chapter 7

Lessons Learned

7.1 Ami Kumar

By working on this project for the semester, I learned many helpful lessons. In terms of the coding aspect, I learned the difficulty of seemingly simple tasks in terms of creating a compiler, such as adding types. In addition, I learned the strengths and weaknesses of certain languages for different implementations. OCaml was difficult to pick up but is much more concise than other languages in which I have coded, such as Java and C++.

For groups taking this course in the future, I would tell them to start as early as possible, and get help whenever they spend too long trying to figure out one aspect of their compilers. The earlier they start, the earlier they will run into problems and be able to ask someone for assistance instead of trying to figure everything out themselves. Over the course of writing the program, I would suggest to test as often possible. Groups should tackle one function at a time and test as soon as they are finished to see if they have implemented it correctly. Finally, if something is not working in their code and they have spent much time and energy trying to fix it already, move on to something else and either edit the vision for your language or come back to the error later.

7.2 Gil Chen-Zion

As I reflect on the work I have done for this assignment I really that I have learned a lot about language design as well as group project organization. First, one concrete thing that I learned that I did not know before this project was how to use a Makefile in order to compile my code. That process allowed for quicker speed when running and testing the program. Furthermore, I learned how the greater process of language design all works into each other. The process for making even a single change within the language requires edits within each and every file. One wrong edit or change can cause errors. This sensitivity was difficult to troubleshoot because it wasn't always easy to pinpoint the problem and the hints were not always effective. Although we did not accomplish all that we had hoped, we were still able to get a grasp and enhance the language to add key functionalities. The learning curve for Ocaml was especially high. Consequently, we do understand that time management in this process was a crucial error that we had. Not only was it time consuming to get an understanding of the larger picture of the language we wanted to create but the actual code generation caused many more issues. This necessity for better understanding of the depth of a project is crucial for me to utilize going forward at Columbia. Furthermore, as the group leader, learning how to delegate tasks and build a language through a decisive "dictorship" is a crucial take away for me from this project.

7.3 Annania Melaku

Although we did not accomplish all that we set out to accomplish, there were several lessons that I will take away from this experience. First and foremost, I realized that the most essential task is fully realizing what the project is in terms of requirements and implementation. By that I mean that it is important that every member is on the same page in how the language should work. Scheduling and time management are also necessary processes for the execution of the project to be smooth and efficient. I learned it is better to allocate more time to each task than to underestimate the amount of time needed for each step. For example, our biggest issue was type declarations which we assumed would be a simple task and therefore did not foresee the intricacies of implementation. In relation to that, it is also beneficial to find dependencies, meaning: which part of the project needs to be completed before another part can be started. Finally, I learned that Ocaml, although powerful, as a language is an acquired taste.

For future projects the necessary steps:

- Figure out what the project is
- set realistic goals and deadlines
    - figure out dependencies
- test & debug as you go along, it is difficult to debug later, especially in Ocaml
- keep track of everything that is going on

7.4 Isaac White

I can't even begin to cover all the lessons I learned from working on this project, but the most important one was the importance of starting early and not underestimating the complexity of the task of building the language. Specifically, every group mentions in their lessons learned that you should start early, but I think that should be expanded upon to discuss not just when to start but how to start.

The most helpful thing to do first is probably to get a thorough understanding of MicroC, the simplified language provided by Professor Edwards. Not only will understanding this language help you understand how the general pieces of the OCaml build system work, but it will also impress upon you how much functionality is left for you to build since MicroC is so limited. Here I think it's important to note that MicroC is probably unlike the language you want to build in one very important way: it only allows for one user defined variable type, Int, and the code provided for it is written accordingly. Figuring out the handling for multiple types effectively was our greatest challenge (other simple interfaces like custom functions, syntax, etc. were deceptively easy to implement), and early on fooled us into believing that we were closer to completion than we actually were.

To avoid this, it is imperative that as soon as the language ideas are defined (when the language reference manual is submitted should be fine), the group should work together to articulate all the steps that need to be taken to create the final deliverable and divide up tasks. Although the files are highly interrelated, a testing architecture must be defined so that members can verify the code they are adding works as they commit it for later use. Otherwise, they are likely to run into the problem that we had where members wrote code without running tests on it as it was being developed, which resulted in overly complex resolutions to get to a final project which we didn't have time for.

In short, I hope future groups will use our lessons as a warning when planning the development of their language. It's not just about your intentions or how much time you think you'll be able to complete the assignment in; you need to write out a plan to determine how much needs to be done each week with a buffer time left at the end so you can evaluate every week if the group is on track. The project is simply too involved when done correctly to recover from failing to plan in this way.

Appendix

A CHIL Grammar

* is 0 or more
+ is one or more
? is 0 or 1 (at most one)
NEWLINE is “\n”

program → *declaration* | *program declaration*
declaration → *fdecl* | *vdecl*
stmt-list → *stmt-list stmt* | *stmt-list*
fdecl → NEWLINE `fn` ID (*formals-list*) NEWLINE *expr-list* `endfn NEWLINE`
expr-list → *expr-list expr* | *expr*
expr → *dataType*
| ID
| *expr* + *expr*
| *expr* ^ *expr*
| *expr* - *expr*
| *expr* * *expr*
| *expr* / *expr*
| *expr* == *expr*
| *expr* != *expr*
| *expr* < *expr*
| *expr* <= *expr*
| *expr* > *expr*
| *expr* >= *expr*
| *expr* && *expr*
| *expr* || *expr*
| *expr* % *expr*
| ID = *expr*
| ID (*actuals-list*)
| ID ++
| ID --
| (*expr*)
formals-list → *dataType* | *formals-list*, *dataType*
vdecl → *dataType* ID = *value* newline
actuals-list → *expr* | *actuals-list*, *expr*
dataType → `boolean | int | float | string | element | style`
variable → ID
value → *true-or-false*
| *integer*
| *number*
| *string*
| *element*
| *style*
true-or-false → `true | false`
integer → [0 - 9]+

float → [0 - 9]*'.'[0-9]+ | [0 - 9]+'.'[0-9]* | 0
string → [a - zA - Z0 - 9]*
element → contents: *string* NEWLINE style: $ { css: *string* }
style → css: *string*

B. Code Style Conventions

B.1 General Principles

In general, CHIL code should be neat, easy to read, consistent, and well-commented. Programmers should always adhere to all conventions of the language. CHIL does not end lines with semicolons, and uses newlines to denote the end of a line instead. Every block, including functions, if/else statements, and loops, start with their keyword identifier and end with the reserved word prefixed by "end." For example, any code within a for loop must be between the `for` and `endfor`. In addition, there must be a newline before and after each function declaration and after every line of code. This is not just for clean code but is a convention of the language and necessary for proper program compilation.

B.2 Documentation Comments

Programmers should comment often to explain code that is ambiguous and to keep their files readable and easy to understand. There is no set documentation pattern, though frequent commenting is suggested. New lines are required after all comments, and is enforced by the compiler in order to ensure a standard format for CHIL programs.

C. Code documentation of final submission

all files heavily based on Professor Edwards' MicroC

ast.ml

```
(* Language interpretation rules *)
(* Operation keys *)
type op = Add | Sub | Mult | Div | Mod | Equal | Neq | Less | Leq | Greater
| Geq | Pow | Fact

(* Possibly types of expressions *)
type expr =
    Literal of int
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

(* Possible types of statments *)
type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

(* Function declaration types *)
type func_decl = {
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
  }

type program = string list * func_decl list

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
      Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/" | Mod -> "%"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">=" | Pow ->
"~" | Fact -> "!") ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ "\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ "\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl id = "int " ^ id ^ "\n"

let string_of_fdecl fdecl =
  fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ") {\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"
```

```
let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

bytecode.ml

```
type bstmt =
    Lit of int    (* Push a literal *)
  | Drp           (* Discard a value *)
  | Bin of Ast.op (* Perform arithmetic on top of stack *)
  | Lod of int    (* Fetch global variable *)
  | Str of int    (* Store global variable *)
  | Lfp of int    (* Load frame pointer relative *)
  | Sfp of int    (* Store frame pointer relative *)
  | Jsr of int    (* Call function by absolute address *)
  | Ent of int    (* Push FP, FP -> SP, SP += i *)
  | Rts of int    (* Restore FP, SP, consume formals, push result *)
  | Beq of int    (* Branch relative if top-of-stack is zero *)
  | Bne of int    (* Branch relative if top-of-stack is non-zero *)
  | Bra of int    (* Branch relative *)
  | Hlt           (* Terminate *)

type prog = {
    num_globals : int;   (* Number of global variables *)
    text : bstmt array; (* Code for all the functions *)
  }

let string_of_stmt = function
    Lit(i) -> "Lit " ^ string_of_int i
  | Drp -> "Drp"
  | Bin(Ast.Add) -> "Add"
  | Bin(Ast.Sub) -> "Sub"
  | Bin(Ast.Mult) -> "Mul"
  | Bin(Ast.Div) -> "Div"
  | Bin(Ast.Equal) -> "Eql"
  | Bin(Ast.Neq) -> "Neq"
  | Bin(Ast.Less) -> "Lt"
  | Bin(Ast.Mod) -> "Mod"
  | Bin(Ast.Leq) -> "Leq"
  | Bin(Ast.Greater) -> "Gt"
  | Bin(Ast.Geq) -> "Geq"
  | Bin(Ast.Pow) -> "Pow"
  | Bin(Ast.Fact) -> "Fact"
  | Lod(i) -> "Lod " ^ string_of_int i
  | Str(i) -> "Str " ^ string_of_int i
  | Lfp(i) -> "Lfp " ^ string_of_int i
  | Sfp(i) -> "Sfp " ^ string_of_int i
  | Jsr(i) -> "Jsr " ^ string_of_int i
  | Ent(i) -> "Ent " ^ string_of_int i
  | Rts(i) -> "Rts " ^ string_of_int i
  | Bne(i) -> "Bne " ^ string_of_int i
  | Beq(i) -> "Beq " ^ string_of_int i
  | Bra(i) -> "Bra " ^ string_of_int i
  | Hlt    -> "Hlt"

let string_of_prog p =
  string_of_int p.num_globals ^ " global variables\n" ^
  let funca = Array.mapi
      (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
  in String.concat "\n" (Array.to_list funca)
```

chil.ml

```
(* Rules for final output file *)
type action = Ast | Interpret | Bytecode | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                          ("-i", Interpret);
                          ("-b", Bytecode);
                          ("-c", Compile) ]
  else Compile in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
    Ast -> let listing = Ast.string_of_program program
           in print_string listing
  | Interpret -> ignore (Interpret.run program)
  | Bytecode -> let listing =
      Bytecode.string_of_prog (Compile.translate program)
    in print_endline listing
  | Compile -> Execute.execute_prog (Compile.translate program)
```

compile.ml

```
(* Rules for compiling code into executable *)
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index   : int StringMap.t; (* "Address" for global variables *)
    local_index    : int StringMap.t; (* FP offset for args, locals *)
  }

(* val enum : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* val string_map_pairs StringMap 'a -> (int * 'a) list -> StringMap 'a *)
let string_map_pairs map pairs =
  List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(** Translate a program in AST form into a bytecode program.  Throw an
    exception if something is wrong, e.g., a reference to an unknown
    variable or function *)
let translate (globals, functions) =

  (* Allocate "addresses" for each global variable *)
  let global_indexes = string_map_pairs StringMap.empty (enum 1 0 globals)
in

  (* Assign indexes to function names; built-in "print" is special *)
  let built_in_functions = StringMap.add "max" (-1) StringMap.empty in
  let built_in_functions = StringMap.add "print" (-1) built_in_functions in
  let function_indexes = string_map_pairs built_in_functions
      (enum 1 1 (List.map (fun f -> f.fname) functions)) in

  (* Translate a function in AST form into a list of bytecode statements *)
  let translate env fdecl =
     (* Bookkeeping: FP offsets for locals and arguments *)
    let num_formals = List.length fdecl.formals
    and num_locals = List.length fdecl.locals
    and local_offsets = enum 1 1 fdecl.locals
    and formal_offsets = enum (-1) (-2) fdecl.formals in
    let env = { env with local_index = string_map_pairs
              StringMap.empty (local_offsets @ formal_offsets) } in

    let rec expr = function
      Literal i -> [Lit i]
      | Id s ->
        (try [Lfp (StringMap.find s env.local_index)]
          with Not_found -> try [Lod (StringMap.find s env.global_index)]
          with Not_found -> raise (Failure ("undeclared variable " ^ s)))
      | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]
      | Assign (s, e) -> expr e @
        (try [Sfp (StringMap.find s env.local_index)]
        with Not_found -> try [Str (StringMap.find s env.global_index)]
        with Not_found -> raise (Failure ("undeclared variable " ^ s)))
      | Call (fname, actuals) -> (try
        (List.concat (List.map expr (List.rev actuals))) @
        [Jsr (StringMap.find fname env.function_index) ]
        with Not_found -> raise (Failure ("undefined function " ^ fname)))
      | Noexpr -> []

    in let rec stmt = function
      Block sl     ->  List.concat (List.map stmt sl)
      | Expr e       -> expr e @ [Drp]
      | Return e     -> expr e @ [Rts num_formals]
      | If (p, t, f) -> let t' = stmt t and f' = stmt f in
      expr p @ [Beq(2 + List.length t')] @
      t' @ [Bra(1 + List.length f')] @ f'
      | For (e1, e2, e3, b) ->
        stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))
      | While (e, b) ->
        let b' = stmt b and e' = expr e in
        [Bra (1+ List.length b')] @ b' @ e' @
        [Bne (-(List.length b' + List.length e'))]

    in [Ent num_locals] @      (* Entry: allocate space for locals *)
    stmt (Block fdecl.body) @  (* Body *)
    [Lit 0; Rts num_formals]   (* Default = return 0 *)

  in let env = { function_index = function_indexes;
               global_index = global_indexes;
               local_index = StringMap.empty } in

  (* Code executed to start the program: Jsr main; halt *)
  let entry_function = try
    [Jsr (StringMap.find "build" function_indexes); Hlt]
  with Not_found -> raise (Failure ("no \"build\" function"))
  in

  (* Compile the functions *)
  let func_bodies = entry_function :: List.map (translate env) functions in

  (* Calculate function entry points by adding their lengths *)
  let (fun_offset_list, _) = List.fold_left
      (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0) func_bodies in
  let func_offset = Array.of_list (List.rev fun_offset_list) in

  { num_globals = List.length globals;
    (* Concatenate the compiled functions and replace the function
       indexes in Jsr statements with PC values *)
    text = Array.of_list (List.map (function
      Jsr i when i > 0 -> Jsr func_offset.(i)
      | _ as s -> s) (List.concat func_bodies))
  }
```

execute.ml

```
open Ast
open Bytecode

(* Stack layout just after "Ent":

              <-- SP
   Local n
   ...
   Local 0
   Saved FP   <-- FP
   Saved PC
   Arg 0
   ...
   Arg n *)

let execute_prog prog =
  let stack = Array.make 1024 0
  and globals = Array.make prog.num_globals 0 in

  let rec exec fp sp pc = match prog.text.(pc) with
    Lit i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)
  | Drp    -> exec fp (sp-1) (pc+1)
  | Bin op -> let op1 = stack.(sp-2) and op2 = stack.(sp-1) in
      stack.(sp-2) <- (let boolean i = if i then 1 else 0 in
      match op with
      Add     -> op1 + op2
      | Sub     -> op1 - op2
      | Mult    -> op1 * op2
      | Div     -> op1 / op2
      | Equal   -> boolean (op1 =  op2)
      | Mod     -> op1 mod op2
      | Pow     -> (*New operation definition for power*)
         let rec powerfun v1 v2 =
        if v2 = 0 then 1
        else v1 * (powerfun v1 (v2 - 1)) in
       (fun v1 v2 ->
         if v2 = 0 then 1
         else v1 * (powerfun v1 (v2 - 1))) op1 op2
       | Fact ->
       let rec factfun v1 = (*New operation definition for factorial*)
        if v1 = 0 then 1
        else v1 * (factfun (v1 - 1)) in
       (fun v1 ->
         if v1 = 0 then 1
         else v1 * (factfun (v1 - 1))) op1
      | Neq     -> boolean (op1 != op2)
      | Less    -> boolean (op1 <  op2)
      | Leq     -> boolean (op1 <= op2)
      | Greater -> boolean (op1 >  op2)
      | Geq     -> boolean (op1 >= op2)) ;
      exec fp (sp-1) (pc+1)
  | Lod i   -> stack.(sp)   <- globals.(i)  ; exec fp (sp+1) (pc+1)
  | Str i   -> globals.(i)  <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Lfp i   -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
  | Sfp i   -> stack.(fp+i) <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Jsr(-1) -> print_endline (string_of_int stack.(sp-1)) ; exec fp sp
(pc+1)
  | Jsr i   -> stack.(sp)   <- pc + 1       ; exec fp (sp+1) i
  | Ent i   -> stack.(sp)   <- fp           ; exec sp (sp+i+1) (pc+1)
  | Rts i   -> let new_fp = stack.(fp) and new_pc = stack.(fp-1) in
               stack.(fp-i-1) <- stack.(sp-1) ; exec new_fp (fp-i) new_pc
  | Beq i   -> exec fp (sp-1) (pc + if stack.(sp-1) =  0 then i else 1)
  | Bne i   -> exec fp (sp-1) (pc + if stack.(sp-1) != 0 then i else 1)
  | Bra i   -> exec fp sp (pc+i)
  | Hlt     -> ()

  in exec 0 0 0
```

interpret.ml

```
open Ast

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
end)

exception ReturnException of int * int NameMap.t

(* Main entry point: run a program *)

let run (vars, funcs) =
  (* Put function declarations in a symbol table *)
  let func_decls = List.fold_left
      (fun funcs fdecl -> NameMap.add fdecl.fname fdecl funcs)
      NameMap.empty funcs
  in

  (* Invoke a function and return an updated global symbol table *)
  let rec call fdecl actuals globals =

    (* Evaluate an expression and return (value, updated environment) *)
    let rec eval env = function
      Literal(i) -> i, env
      | Noexpr -> 1, env (* must be non-zero for the for loop predicate *)
      | Id(var) ->
        let locals, globals = env in
        if NameMap.mem var locals then
          (NameMap.find var locals), env
        else if NameMap.mem var globals then
          (NameMap.find var globals), env
        else raise (Failure ("undeclared identifier " ^ var))
```

```
| Binop(e1, op, e2) ->
  let v1, env = eval env e1 in
    let v2, env = eval env e2 in
  let boolean i = if i then 1 else 0 in
  (match op with
    Add -> v1 + v2
  | Sub -> v1 - v2
  | Mult -> v1 * v2
  | Div -> v1 / v2
  | Pow ->
       let rec powerfun v1 v2 =
            if v2 = 0 then 1
            else v1 * (powerfun v1 (v2 - 1)) in
       (fun v1 v2 ->
   if v2 = 0 then 1
   else v1 * (powerfun v1 (v2 - 1))) v1 v2
  (* define power function *)
   | Fact ->
       let rec factfun v1 =
            if v1 = 0 then 1
            else v1 * (factfun (v1 - 1)) in
       (fun v1 ->
   if v1 = 0 then 1
   else v1 * (factfun (v1 - 1))) v1
  | Equal -> boolean (v1 = v2)
  | Mod -> v1 mod v2
  | Neq -> boolean (v1 != v2)
  | Less -> boolean (v1 < v2)
  | Leq -> boolean (v1 <= v2)
  | Greater -> boolean (v1 > v2)
  | Geq -> boolean (v1 >= v2)), env
| Assign(var, e) ->
  let v, (locals, globals) = eval env e in
  if NameMap.mem var locals then
    v, (NameMap.add var v locals, globals)
  else if NameMap.mem var globals then
    v, (locals, NameMap.add var v globals)
  else raise (Failure ("undeclared identifier " ^ var))
| Call("print", [e]) ->
  let v, env = eval env e in
  print_endline (string_of_int v);
  0, env
 (* | Call("max", [e]) ->
      let v, env = eval env e in
      v, env *)
| Call(f, actuals) ->
  let fdecl =
    try NameMap.find f func_decls
    with Not_found -> raise (Failure ("undefined function " ^ f))
  in
  let actuals, env = List.fold_left
      (fun (actuals, env) actual ->
      let v, env = eval env actual in v :: actuals, env)
```

```
            ([], env) (List.rev actuals)
        in
        let (locals, globals) = env in
        try
          let globals = call fdecl actuals globals
          in 0, (locals, globals)
        with ReturnException(v, globals) -> v, (locals, globals)
    in


    (* Execute a statement and return an updated environment *)
    let rec exec env = function
      Block(stmts) -> List.fold_left exec env stmts
      | Expr(e) -> let _, env = eval env e in env
      | If(e, s1, s2) ->
        let v, env = eval env e in
        exec env (if v != 0 then s1 else s2)
      | While(e, s) ->
        let rec loop env =
          let v, env = eval env e in
          if v != 0 then loop (exec env s) else env
        in loop env
      | For(e1, e2, e3, s) ->
        let _, env = eval env e1 in
        let rec loop env =
          let v, env = eval env e2 in
          if v != 0 then
            let _, env = eval (exec env s) e3 in
            loop env
          else
            env
        in loop env
      | Return(e) ->
        let v, (locals, globals) = eval env e in
        raise (ReturnException(v, globals))
    in

    (* Enter the function: bind actual values to formal arguments *)
    let locals =
      try List.fold_left2
        (fun locals formal actual -> NameMap.add formal actual locals)
        NameMap.empty fdecl.formals actuals
      with Invalid_argument(_) ->
      raise (Failure ("wrong number of arguments passed to " ^ fdecl.fname))
    in
    (* Initialize local variables to 0 *)
    let locals = List.fold_left
      (fun locals local -> NameMap.add local 0 locals) locals fdecl.locals
    in
    (* Execute each statement in sequence, return updated global symbol
table *)
    snd (List.fold_left exec (locals, globals) fdecl.body)
```

```
  (* Run a program: initialize global variables to 0, find and run "main" *)
  in let globals = List.fold_left
      (fun globals vdecl -> NameMap.add vdecl 0 globals) NameMap.empty vars
  in try
    call (NameMap.find "build" func_decls) [] globals
  with Not_found -> raise (Failure ("did not find the build() function"))
```

parser.mly

```
%{ open Ast %}
/*Token keys as defined in the AST*/
%token SEMI LPAR RPAR LBRACE RBRACE COMMA NEWLINE
%token ADD SUB MULT DIV ADDASSN MINUSASSN MULTASSN DIVASSN ASSIGN SQUARE
POWER FACTORIAL MODULUS INCREMENT DECREMENT
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR INT ENDIF ENDFOR ENDWHILE WHILE
%token FN ENDFN
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%right ADDASSN
%right MINUSASSN
%right DIVASSN
%right MULTASSN
%left EQ NEQ
%left LT GT LEQ GEQ
%left ADD SUB
%left MULT DIV MODULUS
%left SQUARE
%left POWER FACTORIAL

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [] }
 | program vdecl { ($2 :: fst $1), snd $1 }
 | program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
   NEWLINE FN ID LPAR formals_opt RPAR NEWLINE vdecl_list stmt_list ENDFN
NEWLINE
```

```
     { { fname = $3;
       formals = $5;
       locals = List.rev $8;
       body = List.rev $9 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { List.rev $1 }

formal_list:
    ID                   { [$1] }
  | formal_list COMMA ID { $3 :: $1 }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   INT ID NEWLINE { $2 }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
  expr NEWLINE { Expr($1) }
  | RETURN expr NEWLINE { Return($2) }
  | IF LPAR expr RPAR NEWLINE stmt_list %prec NOELSE ENDIF NEWLINE { If($3,
Block(List.rev $6), Block([])) }
  | IF LPAR expr RPAR NEWLINE stmt_list ELSE NEWLINE stmt_list ENDIF NEWLINE
{ If($3, Block(List.rev $6), Block(List.rev $9)) }
  | FOR LPAR expr SEMI expr SEMI expr RPAR  NEWLINE stmt_list ENDFOR NEWLINE
     { For($3, $5, $7, Block(List.rev $10)) }
  | WHILE LPAR expr RPAR NEWLINE stmt_list ENDWHILE NEWLINE { While($3,
Block(List.rev $6)) }

/*MATCHING RULES FOR EXPRESSIONS*/
expr:
    LITERAL          { Literal($1) }
  | ID               { Id($1) }
  | expr ADD  expr { Binop($1, Add,   $3) }
  | expr SUB  expr { Binop($1, Sub,   $3) }
  | expr MULT  expr { Binop($1, Mult,  $3) }
  | expr DIV expr { Binop($1, Div,   $3) }
  | expr EQ     expr { Binop($1, Equal, $3) }
  | expr NEQ    expr { Binop($1, Neq,   $3) }
  | expr LT     expr { Binop($1, Less,  $3) }
  | expr LEQ    expr { Binop($1, Leq,   $3) }
  | expr GT     expr { Binop($1, Greater,  $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3) }
  | expr MODULUS expr {Binop($1, Mod, $3)}
  | expr SQUARE        { Binop($1, Mult, $1) }
  | expr FACTORIAL       { Binop($1, Fact, Literal(1)) }
  | expr POWER   expr { Binop($1, Pow, $3) }
  | ID INCREMENT       { Assign($1, Binop(Id($1), Add, Literal(1))) }
  | ID DECREMENT      { Assign($1, Binop(Id($1), Sub, Literal(1))) }
  | ID ADDASSN expr  { Assign($1, Binop(Id($1), Add, $3))}
  | ID MINUSASSN expr  { Assign($1, Binop(Id($1), Sub, $3))}
  | ID MULTASSN expr  { Assign($1, Binop(Id($1), Mult, $3))}
  | ID DIVASSN expr  { Assign($1, Binop(Id($1), Div, $3))}
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID LPAR actuals_opt RPAR { Call($1, $3) }
  | LPAR expr RPAR { $2 }

actuals_opt:
    /* nothing */ { [] }
  | actuals_list  { List.rev $1 }

actuals_list:
    expr                    { [$1] }
  | actuals_list COMMA expr { $3 :: $1 }
```

scanner.mll

```
{ open Parser }
(* TOKEN RULE MAPPING FOR STRINGS *)
rule token = parse
  [' ' '\t' '\r'] { token lexbuf } (* Whitespace *)
| "@>"     { comment lexbuf }
| "@"    { singcomment lexbuf }       (* Comments *)
| '('      { LPAR }
| ')'      { RPAR }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '\n'     { NEWLINE }
| ';'      { SEMI }
| ','      { COMMA }
| '*'      { MULT }
| '+'      { ADD }
| "+="         { ADDASSN }
| "-="         { MINUSASSN }
| "/="         { DIVASSN }
| "*="         { MULTASSN }
| "++"         { INCREMENT }
| "--"         { DECREMENT}
| '-'      { SUB }
| '/'      { DIV }
| "^^"         { SQUARE }
| '~'      { POWER }
| '!'    { FACTORIAL }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| '%'    { MODULUS }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "endfor" { ENDFOR }
| "endif"  { ENDIF }
| "return" { RETURN }
| "int"    { INT }
| "while" { WHILE }
| "endwhile" { ENDWHILE}
| "fn" { FN }
| "endfn" { ENDFN }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "<@\n" { token lexbuf }
| _    { comment lexbuf }

and singcomment = parse
      "\n" {token lexbuf}
| _ { singcomment lexbuf }
```

Chapter 9

New Project

9.1 Example Program

```
fn fib(x)
     if (x < 2)
          return 1
     endif
     return fib(x - 1) + fib(x - 2)
endfn

fn build9)
     print(fib(0))
     print(fib(1))
     print(fib(2))
     print(fib(3))
     print(fib(4))
     print(fib(5))
```

9.2 Tests

```
fn build()
     int b
     b += 10
     print(b)
     b -= 2
     print(b)
     b /= 4
     print(b)
     b *= 2
     print(b)
     b = b ^^
     print(b)
     b -= 14
     print(b)
     b = b - 3
     print(b)
     b = 3!
     print(b)
endfn
```

This program prints:

```
10
8
2
4
16
2
8
```

```
6
```

Example 2:

```
fn space()
        int a
        a = 10000 ^^
        a *= 10
        return a
endfn

fn eyes()
        int b
        b = space()
        b += 11001100
        return b
endfn

fn build()
        print(space())
        print(space())
        print(eyes())
        print(eyes())
        print(eyes())
        print(space() + 1)
        print(space() + 100000010)
        print(space() + 110000110)
        print(space() + 1111100)
        @>
        1100000010
        1110000110
        1001111100
        <@
        print(space())
endfn
```

This program outputs:

```
1000000000
1000000000
1011001100
1011001100
1011001100
1000000001
1100000010
1110000110
1001111100
```

```
1000000000
```

9.3 Tutorial

Build

All code to be run during program execution must be placed into a function with the name "build." The compiler finds this function during processing of the source and uses it to begin execution.

```
fn build()

@>Code to get run during program execution goes here<@

endfn
```

Syntax

Primitive Types

*See Chapter 3

Type Declarations

Integer

int x = 3;

Features

Loops
Comments
Whitespace
Conditionals

*See Chapter 2.2.2.2

A CHIL Grammar

* is 0 or more
+ is one or more
? is 0 or 1 (at most one)
NEWLINE is “\n”

program → declaration | program declaration
declaration → fdecl | vdecl
stmt-list → stmt-list stmt | stmt-list
fdecl → NEWLINE `fn` ID (*formals-list*) NEWLINE *expr-list* `endfn NEWLINE`
expr-list → expr-list expr | expr
expr → dataType
| ID
| *expr* + *expr*
| *expr* ^ *expr*
| *expr* - *expr*
| *expr* * *expr*
| *expr* / *expr*
| *expr* == *expr*
| *expr* != *expr*
| *expr* < *expr*
| *expr* <= *expr*
| *expr* > *expr*
| *expr* >= *expr*
| *expr* && *expr*
| *expr* || *expr*
| *expr* % *expr*
| ID = *expr*
| ID (*actuals-list*)
| ID ++
| ID --
| (*expr*)
formals-list → dataType | formals-list, dataType
vdecl → dataType ID = *value* newline
actuals-list → expr | actuals-list, expr
dataType → `boolean | int | float | string | element | style`
variable → ID
value → true-or-false
| *integer*
| *number*
| *string*
| *element*
| *style*
true-or-false → `true | false`
integer → [0 - 9]+
float → [0 - 9]*’.’[0-9]+ | [0 - 9]+’.’[0-9]* | 0
string → [a - zA - Z0 - 9]*