Friday, August 19th 2016

COMS4115: Final Project Report

Kyra F. Lee, kfl2120@columbia.edu

Introduction

In order to compress the instructions for the thousands of individual stitches that comprise a knitted project, traditional knitting patterns are written in a terse, almost algebraic style. Unfortunately, any arithmetic errors in this notation can make a pattern impossible to follow. This is a problem crying out for a computer-based solution. This project constructs a compiler whose input is very similar to traditional knitting patterns and whose output verifies that a pattern is free of errors and can be executed by a knitter without errors.

A knitted object consists of a series of **rows**. Each row of knitting consists of sequences of two basic stitches, **knit** and **purl**. Because the structure of these stitches, a stitch worked as a knit on the right-side of the fabric produces a purl stitch on the reverse, wrong-side of the fabric, and vice versa. Each row is worked on the foundation of the one preceding it, in alternating directions (considered facing the right-side of the finished fabric, the first, right-side row is worked from right to left, then the second, wrong-side row is worked from left to right, etc.). Each row must consume exactly as many stitches as were produced in the row before it. Additionally, it is possible to **increase** (producing two new stitches while only consuming one from the preceding row) and to **decrease** (consuming two stitches from the preceding row while only producing one). Each stitch can be represented by a node with outgoing pointers to the other stitches that surround it, and a complete knitted project is thus a grid of such interconnected nodes. If a complete data structure has been built, it can be proven to be valid, with each row consuming and producing appropriate numbers of stitches.

Language Tutorial

A traditional knitting pattern (such as the one given for a dishcloth at http://homespunliving.blogspot.com/2007/11/waffle-knit-dishcloth-pattern.html) can be converted into the stricter syntax required by this language, following the Language Reference Manual. For example, the instruction to "Cast on 38 stitches." turns into `CO 38`. The completely converted example is given in the test plan under dishcloth.knit.

Once the .knit program is written, it can be compiled by piping it to the knitting compiler, as `cat dishcloth.knit | ./knit > dishcloth.c` to produce a C language program. The C program can be compiled, for example with `gcc dishcloth.c -o dishcloth`. The executable program can then be run with one of three options.

1. --row_count, which will print a statement of how many rows the pattern contains
2. --stitch_count [row number], which will print a statement of how many stitches the specified row of the pattern contains. If the given row number is greater than the number of rows in the pattern, the number of stitches in the last pattern row will be given instead.
3. --print_chart, which will print a schematic representation of the entire knitted pattern, where each knit stich is represented with a `.` each purl stitch with a `-` each increase with a `v` and each decrease with a `/`
   Examples of all three of these execution modes are given in the test plan examples.

Language Reference Manual

A valid program consists of a series of **row definitions**, a series of **stitch pattern definitions**, and a knitting **pattern**. In this LRM, we define the lexical conventions of the language, the syntax of each of these three components, and the constraints governing what constitutes a valid program. In this manual, underline denotes definitions, *italics* represent non-terminal variables, and `monospace font` represents strings to be included literally in .knit programs.

Lexical Conventions

- Comments begin on their own line with the keyword `Note:` and continue until the end of that line, consuming the next newline character.
- Newlines (`'\n'`) are used to mark the ends of some syntactic elements, as explained below. All other whitespace (`'\t', '\r', ' '`) is ignored, except as necessary to separate tokens.
- Identifiers consist of an upper- or lower-case letter that is not a lower-case `'k'` or `'p'`, followed by zero or more letters, digits, and underscores (`'_'`). The regular expression matcher for these identifiers is thus `['a'-'j' 'l'-'o' 'q'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*`. Lowercase `'k'` and `'p'` cannot begin an identifier, to avoid ambiguity with the knit and purl stitch keywords `'k'` and `'p'`. We denote an identifier in the rules that follow by *Id.*
- Literals consist of a series of ASCII digits representing a base-10 integer. Denoted by *Lit.*

- The symbols `'('`, `')'`, `':'`, `'*'`, and `','` are used as part of program control. Use of any other symbol is an error.
- Reserved keywords which may not be used for any other purpose are `"BO"`, `"CO"`, `"dec"`, `"inc"`, `'k'`, `'p'`, `"repeat"`, `"Row"`, `"Stitch Pattern"`, `"stitches remain"`, `"times"`, `"until"`, and `"Work"`. Additionally, no reserved keyword of the C programming language may be used as an identifier.

Syntax

- Variables are either integer literals or string identifiers (constrained as defined above).
  var = Lit | Id
- A stitch instruction is a single increase or decrease, or a variable representing a number of knit or purl stitches. *Stitch* = `inc | dec | k` *var* `| p` *var*. Whitespace between the `k` or `p` and the variable is optional.
-
- A stitch list is a comma-separated series of some number of such stitch instructions.
  stitch_list = Stitch (`,` *Stitch*)*
- Formal argument lists are series of identifiers, also comma-separated.
  formal_args = Id (`,` *Id*)*
- A repeat condition is either an instruction to repeat a certain number of times, or until a certain number of stitches remain in the previous row.
  repeat_condition = var `times` | `until` *var* `stitches remain`
- A row definition is an identifier, optional formal arguments set off by parentheses, a colon, an optional beginning stitch list, a non-empty repeat stitch list set off by *s, a repeat condition, and an optional ending stitch list, and a newline.

row = Id `Row(` *formal_args*? `):` *stitch_list*? `*` *stitch_list* `* repeat` *repeat_condition stitch_list?* \n

- An actual arguments list is a series of comma-separated variables.
  actual_args = var (`,` *var*)*
- A call references a previously defined row or stitch pattern name, followed by a list of actual arguments corresponding to the pattern definition's formal arguments.
  call = Id`(`*actual_args*?`)`
- A stitch pattern is an identifier, a colon, a newline, and a series of newline-separated calls.
  stitch_pattern = Id `Stitch Pattern:`\n (*call* \n)*
- A pattern is a cast-on statement, a series of newline-separated calls paired with repeat conditions, and a bind-off statement.
  pattern = `CO` *var* \n (`Work` *call repeat_condition* \n)* `BO`
- A complete program is a series of rows, a series of stitch patterns, and a single pattern.
  *Program = row** \n *stitch_pattern** \n *pattern*

Program Validity

Identifiers used as variables may only be declared in formal argument lists. It is an error to reference a variable that has not been declared in this way. It is also an error to use the same identifier to refer to more than one row or stitch pattern, or to call an identifier that has not been defined as a row or stitch pattern. It is an error to call a row or stitch pattern with a different number of actual arguments than the definition has formal arguments.

The program executes by creating the stitches specified by each row called by the pattern or stitch pattern in order. To be valid, each row worked in the knitting pattern must consume exactly as many stitches as were produced by the row before it. Each knit or purl consumes and produces one stitch. An increase consumes one stitch and produces two stitches. A decrease consumes two stitches and produces one stitch. If while the program executes a row ever attempts to consume more stitches than the previous row produced, the program will immediately terminate with a failure message. If at the end of any row there are still unconsumed stitches from the previous row remaining, the program will immediately terminate with a failure message.

Project Plan

Programming style and process

Since this project was developed individually, formal communication, strict code check-in requirements, and separately defined roles were not necessary. A stricter project timeline would have been a helpful tool, but one was not strictly followed (see Lessons Learned). OCaml source files are restricted to an 80-character line length.

Development environment

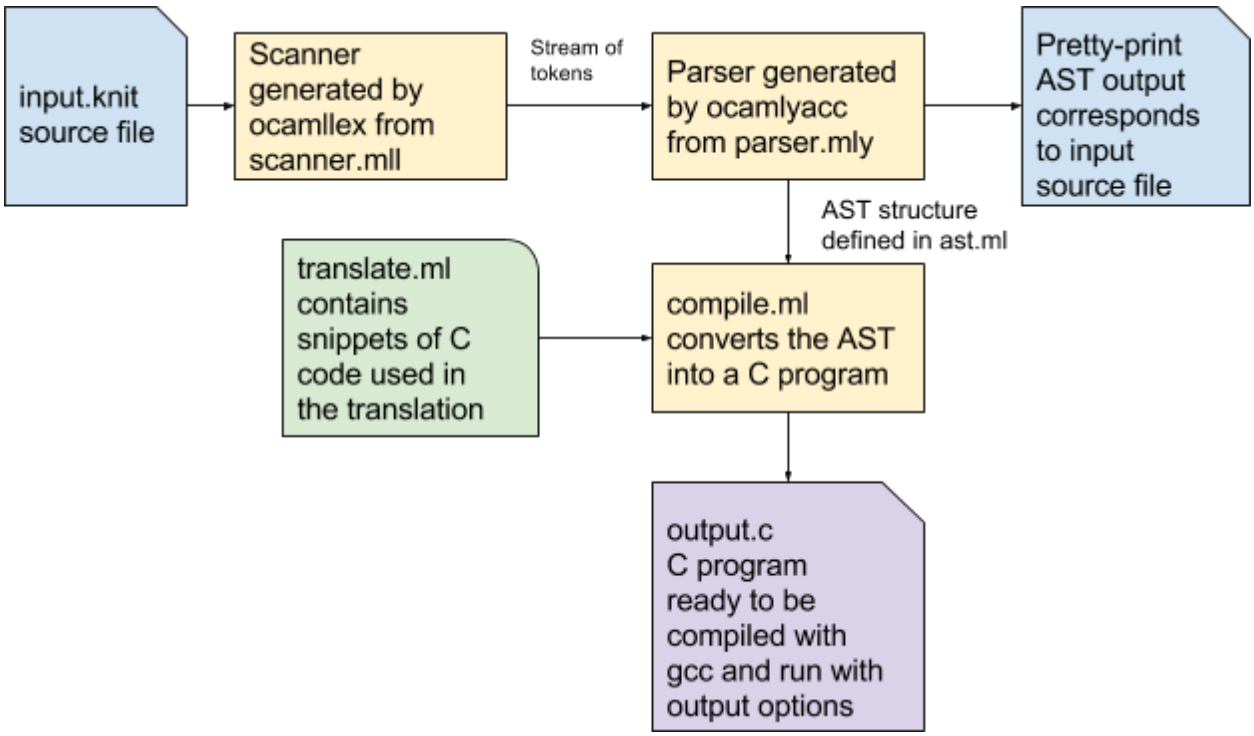
The OCaml compiler, version 4.01.0
gcc version 4.8.4 (Ubuntu 4.8.4-2ubuntu1~14.04.3)
vim, gedit, make, local git repository

Project Log

Date	Log
6/8	Initial language proposal submitted
6/16	Proposal comments received
6/21	Proposal comments response
6/29	First draft of translator frontend (Scanner, Parser, AST)
6/29	Initial LRM submitted
7/12	LRM comments received
8/7	Translator frontend rewrite completed (simplified AST definition from 62 to 32 LOC).
8/8	AST pretty-print output working end-to-end (program can be parsed, output, and reparsed into the same AST).
8/12	Output program structure finalized, including stitch struct.

8/15	Compilation flow working end-to-end
8/18	Additional testing and output tweaks
8/19	Final report submitted

Architectural Design Diagram



Test Plan

The test plan for this compiler consists of several small programs, compiled and executed, with their translations and execution output examined. Below, the details of some representative test programs are given in full. Other test programs in the full testing suite include programs to test small, plain squares of just knit stitches, purl stitches, knits and purls in combination, programs to test increases and decreases on their own, and programs to test error conditions for too many and not enough stitches consumed. All test programs can be run and analyzed with a quick series of shell commands.

incdec.knit

This program tests the increase and decrease stitch functionality, as well as providing interesting examples for the --row_count and --stitch_count executable options. These queries make it possible to use knit program outputs with increases and decreases to implicitly calculate values based on the requested start and end stitch count. For example, the increase portion of this program shows that by increasing 2 stitches on every row, it will take 5 rows to increase from 10 to 20 stitches, or (20 - 10) / 2 = 5.

- Input file

```
increase Row(): k1, inc *k1* repeat until 2 stitches remain inc, k1
decrease Row(): k1, dec *k1* repeat until 3 stitches remain dec, k1

CO 10
Work increase() until 20 stitches remain
Work decrease() until 10 stitches remain
BO
```

- Translated output C file

```
#include <stdio.h>
#include <stdlib.h>

enum stitch_type { KNIT, PURL, INCREASE, DECREASE, CO };

struct stitch {
  int right_side;
  int count;
  enum stitch_type type;
  struct stitch* next;
  struct stitch* prev;
  struct stitch* next_row;
  struct stitch* prev_row;
};

struct stitch* curr;
struct stitch* prev_row;
int i, repeat;
void print_symbol(enum stitch_type type, int rs) {
  char symbol = 'x';
  if ((type == KNIT && rs) || (type == PURL && !rs)) { symbol = '.';
  if ((type == PURL && rs) || (type == KNIT && !rs)) { symbol = '-';
  if (type == INCREASE) { symbol = 'v'; }
  if (type == DECREASE) { symbol = '/'; }
  if (type == CO) { symbol = '_'; }
  printf("%c ", symbol);
}void make_stitch(enum stitch_type type) {
  struct stitch* new_stitch =
      (struct stitch*) malloc(sizeof(struct stitch));
  new_stitch->type = type;
  new_stitch->next_row = NULL;
  new_stitch->next = NULL;
  new_stitch->prev = curr;
```

```
  int count = 1;
  if (curr != NULL) {
    curr->next = new_stitch;
    count = curr->count + 1;
  }
  new_stitch->count = count;
  new_stitch->prev_row = prev_row;
  if (prev_row != NULL) {
    prev_row->next_row = new_stitch;
    prev_row = prev_row->prev;
  } else if (type != CO) {    printf("ERROR! No stitch available to
work type %d\n", type);    exit(0);
  }  curr = new_stitch;
}

void decrease() {
for (i=0;i<1;i++){make_stitch(KNIT);}
make_stitch(DECREASE);prev_row->next_row=curr;prev_row=prev_row->prev
do {for (i=0;i<1;i++){make_stitch(KNIT);}}while(prev_row!=NULL &&
prev_row->count!=3);
make_stitch(DECREASE);prev_row->next_row=curr;prev_row=prev_row->prev
for (i=0;i<1;i++){make_stitch(KNIT);}
if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "decrease");
  exit(0);
}
}

void increase() {
for (i=0;i<1;i++){make_stitch(KNIT);}
make_stitch(INCREASE);prev_row=prev_row->next;make_stitch(INCREASE);
do {for (i=0;i<1;i++){make_stitch(KNIT);}}while(prev_row!=NULL &&
prev_row->count!=2);
make_stitch(INCREASE);prev_row=prev_row->next;make_stitch(INCREASE);
for (i=0;i<1;i++){make_stitch(KNIT);}
if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "increase");
  exit(0);
}
}
int main(int argc, char* argv[]) {
int repeat;
struct stitch* first_stitch;
make_stitch(CO);
int i;
```

```
for (i = 1; i < 10; i++) {
make_stitch(CO);
}
first_stitch=curr;
do {
prev_row=curr;curr=NULL;
increase();prev_row=curr;}while(prev_row!=NULL &&
prev_row->count!=20);
do {
prev_row=curr;curr=NULL;
decrease();prev_row=curr;}while(prev_row!=NULL &&
prev_row->count!=10);if(first_stitch->next_row!=NULL){first_stitch=fi
st_stitch->next_row;}
if (argc > 1) {
if (!strcmp(argv[1], "--row_count")) {curr=first_stitch;i=0;
while(curr!=NULL){i++; curr=curr->next_row;}
printf("This pattern has %d rows.\n",i);
}
if (!strcmp(argv[1], "--stitch_count")) {curr=first_stitch;
if(argc > 2){
  repeat=atoi(argv[2]);i=1;
  while(curr->next_row!=NULL && i <
repeat){i++;curr=curr->next_row;}}
  repeat=i;i=0;
  if(curr->next!=NULL){while(curr!=NULL){i++;curr=curr->next;}}
  else{i=curr->count;}
  printf("Row %d of this pattern has %d stitches.\n",repeat,i);
}
if (!strcmp(argv[1], "--print_chart")) {curr=first_stitch;
while(curr->next_row!=NULL){curr=curr->next_row;}
first_stitch=curr;
while(first_stitch!=NULL){
  if(curr->next!=NULL){
    while(curr!=NULL){print_symbol(curr->type,1);curr=curr->next;}
  } else {
    while(curr!=NULL){print_symbol(curr->type,0);curr=curr->prev;}
  }
  printf("\n");first_stitch=first_stitch->prev_row;curr=first_stitch;
}
}
}
}
```

- Sample execution output

```
knit$ ./incdec --row_count
This pattern has 10 rows.
knit$ ./incdec --stitch_count 5
Row 5 of this pattern has 20 stitches.
knit$ ./incdec --stitch_count 6
Row 6 of this pattern has 18 stitches.
knit$ ./incdec --stitch_count 9
Row 9 of this pattern has 12 stitches.
knit$ ./incdec --stitch_count 2
Row 2 of this pattern has 14 stitches.
```

vars.knit

This program tests the use of variables in knitting programs, and the output displays the --print_chart execution option, with knit and purl stitches shown as they appear from the right side of the knitted work.

- Input file

```
split Row(a, b): pa *k1* repeat until b stitches remain p b

variable Stitch Pattern(x):
  split(x, 7)
  split(5, x)

CO 15
Work variable(3) 10 times
BO
```

- Translated output C file

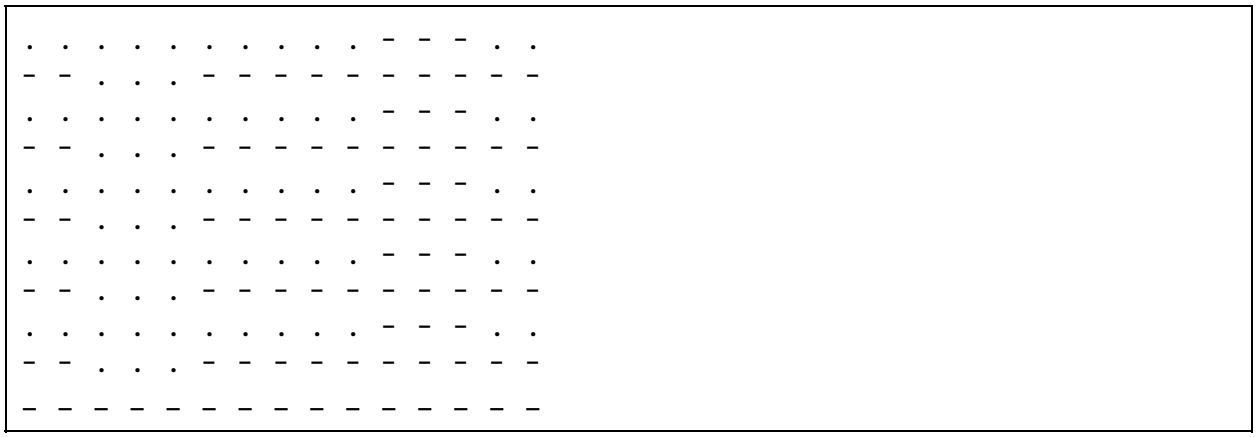
```
#include <stdio.h>
#include <stdlib.h>

enum stitch_type { KNIT, PURL, INCREASE, DECREASE, CO };

struct stitch {
  int right_side;
  int count;
  enum stitch_type type;
  struct stitch* next;
  struct stitch* prev;
  struct stitch* next_row;
  struct stitch* prev_row;
};
```

```
struct stitch* curr;
struct stitch* prev_row;
int i, repeat;
void print_symbol(enum stitch_type type, int rs) {
  char symbol = 'x';
  if ((type == KNIT && rs) || (type == PURL && !rs)) { symbol = '.';
  if ((type == PURL && rs) || (type == KNIT && !rs)) { symbol = '-';
  if (type == INCREASE) { symbol = 'v'; }
  if (type == DECREASE) { symbol = '/'; }
  if (type == CO) { symbol = '_'; }
  printf("%c ", symbol);
}void make_stitch(enum stitch_type type) {
  struct stitch* new_stitch =
      (struct stitch*) malloc(sizeof(struct stitch));
  new_stitch->type = type;
  new_stitch->next_row = NULL;
  new_stitch->next = NULL;
  new_stitch->prev = curr;
  int count = 1;
  if (curr != NULL) {
    curr->next = new_stitch;
    count = curr->count + 1;
  }
  new_stitch->count = count;
  new_stitch->prev_row = prev_row;
  if (prev_row != NULL) {
    prev_row->next_row = new_stitch;
    prev_row = prev_row->prev;
  } else if (type != CO) {    printf("ERROR! No stitch available to
work type %d\n", type);    exit(0);
  }  curr = new_stitch;
}

void split(int a,int b) {
for (i=0;i<a;i++){make_stitch(PURL);}
do {for (i=0;i<1;i++){make_stitch(KNIT);}}while(prev_row!=NULL &&
prev_row->count!=b);
for (i=0;i<b;i++){make_stitch(PURL);}
if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "split");
  exit(0);
}
}
int main(int argc, char* argv[]) {
int repeat;
struct stitch* first_stitch;
```

```
make_stitch(CO);
int i;
for (i = 1; i < 15; i++) {
make_stitch(CO);
}
first_stitch=curr;
for(repeat=0;repeat<10;repeat++){

prev_row=curr;curr=NULL;
split(2,10);prev_row=curr;
}if(first_stitch->next_row!=NULL){first_stitch=first_stitch->next_row
}
if (argc > 1) {
if (!strcmp(argv[1], "--row_count")) {curr=first_stitch;i=0;
while(curr!=NULL){i++; curr=curr->next_row;}
printf("This pattern has %d rows.\n",i);
}
if (!strcmp(argv[1], "--stitch_count")) {curr=first_stitch;
if(argc > 2){
  repeat=atoi(argv[2]);i=1;
  while(curr->next_row!=NULL && i <
repeat){i++;curr=curr->next_row;}}
  repeat=i;i=0;
  if(curr->next!=NULL){while(curr!=NULL){i++;curr=curr->next;}}
  else{i=curr->count;}
  printf("Row %d of this pattern has %d stitches.\n",repeat,i);
}
if (!strcmp(argv[1], "--print_chart")) {curr=first_stitch;
while(curr->next_row!=NULL){curr=curr->next_row;}
first_stitch=curr;
while(first_stitch!=NULL){
  if(curr->next!=NULL){
    while(curr!=NULL){print_symbol(curr->type,1);curr=curr->next;}
  } else {
    while(curr!=NULL){print_symbol(curr->type,0);curr=curr->prev;}
  }
  printf("\n");first_stitch=first_stitch->prev_row;curr=first_stitch;
}
}
}
}
```

- Sample execution output

```
knit$ ./vars --print_chart
```



dishcloth.knit

This program, based on the example given in the tutorial, shows how closely the programs of this knitting correspond to real, traditional knitting patterns and tests the basics of row and stitch pattern functionality.

- Input file

```
Note: See
http://homespunliving.blogspot.com/2007/11/waffle-knit-dishcloth-patt
rn.html

First Row(): *k1* repeat until 0 stitches remain

Second Row(): k3 *p1* repeat until 3 stitches remain k3

Third Row(): k3 *p2, k1* repeat 10 times p2, k3

Fourth Row(): k3 *k2, p1* repeat 10 times k5

WaffleKnit Stitch Pattern():
  First()
  Second()
  Third()
  Fourth()


CO 38
Note: The first row of the stitch pattern is also a plain knit garter
row
Work First() 3 times
Work WaffleKnit() 14 times
Work First() 4 times
BO
```

- Translated output C file

```
#include <stdio.h>
#include <stdlib.h>

enum stitch_type { KNIT, PURL, INCREASE, DECREASE, CO };

struct stitch {
  int right_side;
  int count;
  enum stitch_type type;
  struct stitch* next;
  struct stitch* prev;
  struct stitch* next_row;
  struct stitch* prev_row;
};

struct stitch* curr;
struct stitch* prev_row;
int i, repeat;
void print_symbol(enum stitch_type type, int rs) {
  char symbol = 'x';
  if ((type == KNIT && rs) || (type == PURL && !rs)) { symbol = '.';
  if ((type == PURL && rs) || (type == KNIT && !rs)) { symbol = '-';
  if (type == INCREASE) { symbol = 'v'; }
  if (type == DECREASE) { symbol = '/'; }
  if (type == CO) { symbol = '_'; }
  printf("%c ", symbol);
}void make_stitch(enum stitch_type type) {
  struct stitch* new_stitch =
      (struct stitch*) malloc(sizeof(struct stitch));
  new_stitch->type = type;
  new_stitch->next_row = NULL;
  new_stitch->next = NULL;
  new_stitch->prev = curr;
  int count = 1;
  if (curr != NULL) {
    curr->next = new_stitch;
    count = curr->count + 1;
  }
  new_stitch->count = count;
  new_stitch->prev_row = prev_row;
  if (prev_row != NULL) {
    prev_row->next_row = new_stitch;
    prev_row = prev_row->prev;
  } else if (type != CO) {    printf("ERROR! No stitch available to
```

```
work type %d\n", type);    exit(0);
  }  curr = new_stitch;
}

void Fourth() {
for (i=0;i<3;i++){make_stitch(KNIT);}
for(repeat=0;repeat<10;repeat++){
for (i=0;i<2;i++){make_stitch(KNIT);}
for (i=0;i<1;i++){make_stitch(PURL);}
}
for (i=0;i<5;i++){make_stitch(KNIT);}
if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "Fourth");
  exit(0);
}
}

void Third() {
for (i=0;i<3;i++){make_stitch(KNIT);}
for(repeat=0;repeat<10;repeat++){
for (i=0;i<2;i++){make_stitch(PURL);}
for (i=0;i<1;i++){make_stitch(KNIT);}
}
for (i=0;i<2;i++){make_stitch(PURL);}
for (i=0;i<3;i++){make_stitch(KNIT);}
if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "Third");
  exit(0);
}
}

void Second() {
for (i=0;i<3;i++){make_stitch(KNIT);}
do {for (i=0;i<1;i++){make_stitch(PURL);}}while(prev_row!=NULL &&
prev_row->count!=3);
for (i=0;i<3;i++){make_stitch(KNIT);}
if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "Second");
  exit(0);
}
}

void First() {
```

```
do {for (i=0;i<1;i++){make_stitch(KNIT);}}while(prev_row!=NULL &&
prev_row->count!=0);

if (prev_row != NULL) {
  printf("ERROR! %d stitches remain after working %s Row.\n",
prev_row->count, "First");
  exit(0);
}
}
void WaffleKnit() {
First();
prev_row=curr;curr=NULL;
Second();
prev_row=curr;curr=NULL;
Third();
prev_row=curr;curr=NULL;
Fourth();
}


int main(int argc, char* argv[]) {
int repeat;
struct stitch* first_stitch;
make_stitch(CO);
int i;
for (i = 1; i < 38; i++) {
make_stitch(CO);
}
first_stitch=curr;
for(repeat=0;repeat<3;repeat++){

prev_row=curr;curr=NULL;
First();prev_row=curr;
}
for(repeat=0;repeat<14;repeat++){

prev_row=curr;curr=NULL;
WaffleKnit();prev_row=curr;
}
for(repeat=0;repeat<4;repeat++){

prev_row=curr;curr=NULL;
First();prev_row=curr;
}if(first_stitch->next_row!=NULL){first_stitch=first_stitch->next_row
}
if (argc > 1) {
if (!strcmp(argv[1], "--row_count")) {curr=first_stitch;i=0;
```

```
while(curr!=NULL){i++; curr=curr->next_row;}
printf("This pattern has %d rows.\n",i);
}
if (!strcmp(argv[1], "--stitch_count")) {curr=first_stitch;
if(argc > 2){
  repeat=atoi(argv[2]);i=1;
  while(curr->next_row!=NULL && i <
repeat){i++;curr=curr->next_row;}}
  repeat=i;i=0;
  if(curr->next!=NULL){while(curr!=NULL){i++;curr=curr->next;}}
  else{i=curr->count;}
  printf("Row %d of this pattern has %d stitches.\n",repeat,i);
}
if (!strcmp(argv[1], "--print_chart")) {curr=first_stitch;
while(curr->next_row!=NULL){curr=curr->next_row;}
first_stitch=curr;
while(first_stitch!=NULL){
  if(curr->next!=NULL){
    while(curr!=NULL){print_symbol(curr->type,1);curr=curr->next;}
  } else {
    while(curr!=NULL){print_symbol(curr->type,0);curr=curr->prev;}
  }
  printf("\n");first_stitch=first_stitch->prev_row;curr=first_stitch;
}
}
}
}
```

Lessons Learned

The correct approach to the AST is the most important part of a compiler project. Earlier, failed AST strategies led to overly complex code and very slow progress. Once the most necessary parts of the AST became clear and the clean rewrite was fully completed, the rest of the project fell into place in less than two weeks. Setting and achieving an earlier deadline for this milestone would have allowed more time for documentation, automated testing, and error handling.

Appendix: Code Listing

ast.ml

```
type var = Literal of int | Id of string

type stitch = Knit of var | Purl of var | Increase | Decrease

type repeat = Times of var | Until of var

type row = {
    row_name : string;
    row_formals : string list;
    row_start : stitch list;
    row_repeat : stitch list;
    repeat_condition : repeat;
    row_end : stitch list;
  }

type call = {
    called_name : string;
    actuals : var list;
  }

type stitch_pattern = {
    stitch_pattern_name : string;
    stitch_pattern_formals : string list;
    rows : call list;
  }

type pattern = {
    cast_on : int;
    work : (call * repeat) list;
  }

type program = row list * stitch_pattern list * pattern option

let string_of_var = function
    Literal(l) -> string_of_int l
  | Id(name)   -> name

let string_of_stitch = function
    Knit(count) -> "k" ^ string_of_var count
  | Purl(count) -> "p" ^ string_of_var count
  | Increase    -> "inc"
  | Decrease    -> "dec"
```

```
let string_of_repeat = function
    Times(count)  -> string_of_var count ^ " times"
  | Until(remain) -> "until " ^ string_of_var remain ^ " stitches
remain"

let string_of_row row =
  row.row_name ^ " Row (" ^ String.concat ", " row.row_formals ^ "):
^
  String.concat "," (List.map string_of_stitch row.row_start) ^
  " * " ^  String.concat "," (List.map string_of_stitch
row.row_repeat) ^
  " * repeat from * to * " ^ string_of_repeat row.repeat_condition ^
  " " ^ String.concat "," (List.map string_of_stitch row.row_end) ^
"\n"

let string_of_call call = call.called_name ^ "(" ^
  String.concat ", " (List.map string_of_var call.actuals) ^ ")"

let string_of_stitch_pattern stitch_pattern =
  stitch_pattern.stitch_pattern_name ^ " Stitch Pattern (" ^
  String.concat ", " stitch_pattern.stitch_pattern_formals ^ "):\n" ^
  String.concat "\n" (List.map string_of_call stitch_pattern.rows) ^
"\n"

let string_of_work (call, repeat) =
  "Work " ^ string_of_call call ^ " " ^ string_of_repeat repeat

let string_of_pattern = function
    None -> "WHOOPS!!! No pattern?!?!?!"
  | Some(pattern) -> "CO " ^ string_of_int pattern.cast_on ^ "\n" ^
    String.concat "\n" (List.map string_of_work pattern.work) ^
    "\nBO"

let string_of_program (rows, stitch_patterns, pattern) =
  String.concat "\n" (List.map string_of_row rows) ^ "\n" ^
  String.concat "\n" (List.map string_of_stitch_pattern
stitch_patterns) ^
  "\n" ^ string_of_pattern pattern ^ "\n"
```

compile.ml

```
open Ast
open Translate

(** Translate a program in AST form into the text of a C program. *)
```

```
let translate (rows, stitch_patterns, pattern) =
  let translate_stitch = function
      Knit(var) -> for_loop (string_of_var var) "make_stitch(KNIT);"
    | Purl(var) -> for_loop (string_of_var var) "make_stitch(PURL);"
    | Increase ->

"make_stitch(INCREASE);prev_row=prev_row->next;make_stitch(INCREASE)
    | Decrease ->

"make_stitch(DECREASE);prev_row->next_row=curr;prev_row=prev_row->pre
;"
  in
  let translate_repeat body = function
      Times(var) -> "for(repeat=0;repeat<" ^ (string_of_var var) ^
";repeat++)"
      ^ "{\n" ^ body ^ "\n}"
    | Until(var) -> "do {" ^ body ^ "}"
      ^ "while(prev_row!=NULL && prev_row->count!=" ^ string_of_var
var ^ ");"
  in
  let translate_row row =
    "\nvoid " ^ row.row_name ^ "("
    ^ (String.concat "," (List.map (fun x -> ("int " ^ x) )
row.row_formals))
    ^ ") {"
    ^ debug ("printf(\"" ^ row.row_name ^ "\\n\");\n") ^ "\n"
    ^ (String.concat "\n" (List.map translate_stitch row.row_start))
^ "\n"
    ^ translate_repeat
      (String.concat "\n" (List.map translate_stitch row.row_repeat))
      row.repeat_condition
    ^ "\n" ^ (String.concat "\n" (List.map translate_stitch
row.row_end))
    ^ "\nif (prev_row != NULL) {\n"
    ^ "  printf(\"ERROR! %d stitches remain after working %s
Row.\\n\","
    ^ " prev_row->count, \"" ^ row.row_name ^ "\");\n"
    ^ "  exit(0);\n"
    ^ "}\n}"
  in
  let translate_call call =
    call.called_name ^ "("
    ^ (String.concat "," (List.map string_of_var call.actuals)) ^
");" in
  let translate_stitch_pattern stitch_pattern =
    "\nvoid " ^ stitch_pattern.stitch_pattern_name ^ "("
    ^ String.concat ", " stitch_pattern.stitch_pattern_formals ^ ")
```

```
{\n"
    ^ debug ("printf(\"" ^ stitch_pattern.stitch_pattern_name ^
"\\n\");\n")
    ^ String.concat new_row (List.map translate_call
stitch_pattern.rows)
    ^ "\n}\n\n" in
  let translate_pattern_repeat (call, repeat) =
    translate_repeat (new_row ^ translate_call call ^
"prev_row=curr;") repeat
  in
  match pattern with
    None -> ":("
  | Some(pattern) ->
    preamble
    ^ String.concat "\n" (List.map translate_row rows)
    ^ String.concat "\n" (List.map translate_stitch_pattern
stitch_patterns)
    ^ "\nint main(int argc, char* argv[]) {\n"
    ^ "int repeat;\n"
    ^ "struct stitch* first_stitch;\n"
    ^ cast_on pattern.cast_on
    ^ "first_stitch=curr;\n"
    ^ String.concat "\n" (List.map translate_pattern_repeat
pattern.work)
    ^ final_output
```

knit.ml

```
type action = Ast | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
          List.assoc Sys.argv.(1) [ ("-a", Ast); ("-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
  | Compile -> print_string (Compile.translate program)
```

Makefile

```
OBJS = ast.cmo parser.cmo scanner.cmo translate.cmo compile.cmo
knit.cmo
```

```
knit : $(OBJS)
     ocamlc -o knit $(OBJS)

scanner.ml : scanner.mll
     ocamllex scanner.mll

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

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

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

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

# Generated by ocamldep *.ml *.mli
ast.cmo :
ast.cmx :
compile.cmo : translate.cmo ast.cmo
compile.cmx : translate.cmx ast.cmx
knit.cmo : scanner.cmo parser.cmi compile.cmo ast.cmo
knit.cmx : scanner.cmx parser.cmx compile.cmx ast.cmx
parser.cmo : ast.cmo parser.cmi
parser.cmx : ast.cmx parser.cmi
scanner.cmo : parser.cmi
scanner.cmx : parser.cmx
translate.cmo :
translate.cmx :
parser.cmi : ast.cmo
```

parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN COMMA COLON STAR
%token KNIT PURL INC DEC CO BO
%token WORK REPEAT TIMES UNTIL REMAIN
%token ROW STPATT
%token <int> LITERAL
%token <string> ID
```

```
%token EOL EOF

%start program
%type <Ast.program> program

%%

program:
   /* nothing */ { [], [], None }
 | program EOL { $1 }
 | program row { let (rows, stitch_patterns, pattern) = $1 in
                 $2 :: rows, stitch_patterns, pattern }
 | program stitch_pattern { let (rows, stitch_patterns, pattern) = $1
in
                            rows, $2 :: stitch_patterns, pattern }
 | program pattern { let (rows, stitch_patterns, _ )  = $1 in
                            rows, stitch_patterns, $2 }
 | program error { print_endline "KYRA :("; ([], [], None) }

row:
  ID ROW LPAREN formals_opt RPAREN COLON
    stitch_list_opt STAR stitch_list STAR REPEAT repeat
stitch_list_opt EOL
    { { row_name = $1;
        row_formals = $4;
        row_start = $7;
        row_repeat = List.rev $9;
        repeat_condition = $12;
        row_end = $13 } }

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

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

stitch_pattern:
  ID STPATT LPAREN formals_opt RPAREN COLON EOL call_list EOL { {
    stitch_pattern_name = $1;
    stitch_pattern_formals = $4;
    rows = List.rev $8; } }

pattern:
  CO LITERAL EOL work_list BO EOL { Some({ cast_on = $2; work =
List.rev $4; }) }
```

```
call:
  ID LPAREN vars_opt RPAREN { { called_name = $1; actuals = $3; } }

call_list:
    /* nothing */  { [] }
  | call_list call EOL { $2 :: $1 }

work_list:
    /* nothing */         { [] }
  | work_list WORK call repeat EOL { ($3, $4) :: $1 }

repeat:
    var TIMES        { Times($1) }
  | UNTIL var REMAIN { Until($2) }

stitch_list_opt:
    /* nothing */ { [] }
  | stitch_list   { List.rev $1 }

stitch_list:
    stitch { [$1] }
  | stitch_list COMMA stitch { $3 :: $1 }

stitch:
    KNIT var { Knit($2) }
  | PURL var { Purl($2) }
  | INC      { Increase }
  | DEC      { Decrease }

vars_opt:
    /* nothing */ { [] }
  | var_list      { List.rev $1 }

var_list:
    var                { [$1] }
  | var_list COMMA var { $3 :: $1 }

var:
    LITERAL { Literal($1) }
  | ID      { Id($1) }
```

scanner.mll

```
{ open Parser }
```

```
rule token = parse
  [' ' '\t' '\r' ] { token lexbuf } (* Whitespace *)
| "Note: "     { comment lexbuf }   (* Comments *)
| '\n'      { EOL }
| '('       { LPAREN }
| ')'       { RPAREN }
| ','       { COMMA }
| ':'       { COLON }
| '*'       { STAR }
| 'k'       { KNIT }
| 'p'       { PURL }
| "inc"     { INC }
| "dec"     { DEC }
| "CO"      { CO }
| "BO"      { BO }
| "Row"     { ROW }
| "Work"    { WORK }
| "times"   { TIMES }
| "until"   { UNTIL }
| "stitches remain" { REMAIN }
| "Stitch Pattern" { STPATT }
| "repeat" { REPEAT }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'j' 'l'-'o' 'q'-'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 }
```

translate.ml

```
(* To turn on debug output in the C program generated by the compiler
  change this debug function to return the log statement instead of "
*)
let debug log = ""
(* Code snippets to be included in the generated C program. *)
let includes =
    "#include <stdio.h>\n" ^
    "#include <stdlib.h>\n\n"
let stitch_enum =
    "enum stitch_type { KNIT, PURL, INCREASE, DECREASE, CO };\n\n"
```

```
let stitch_struct =
    "struct stitch {\n" ^
    "  int right_side;\n" ^
    "  int count;\n" ^
    "  enum stitch_type type;\n" ^
    "  struct stitch* next;\n" ^
    "  struct stitch* prev;\n" ^
    "  struct stitch* next_row;\n" ^
    "  struct stitch* prev_row;\n" ^
    "};\n\n"
let globals =
    "struct stitch* curr;\nstruct stitch* prev_row;\n" ^
    "int i, repeat;\n"
let print_stitch_symbol =
    "void print_symbol(enum stitch_type type, int rs) {\n" ^
    "  char symbol = 'x';\n" ^
    "  if ((type == KNIT && rs) || (type == PURL && !rs)) { symbol =
'.'; }\n" ^
    "  if ((type == PURL && rs) || (type == KNIT && !rs)) { symbol =
'-'; }\n" ^
    "  if (type == INCREASE) { symbol = 'v'; }\n" ^
    "  if (type == DECREASE) { symbol = '/'; }\n" ^
    "  if (type == CO) { symbol = '_'; }\n" ^
    "  printf(\"%c \", symbol);\n" ^
    "}"
let make_stitch =
    "void make_stitch(enum stitch_type type) {\n" ^
    debug "  printf(\"type %d\t\", type);\n" ^
    "  struct stitch* new_stitch =\n" ^
    "      (struct stitch*) malloc(sizeof(struct stitch));\n" ^
    "  new_stitch->type = type;\n" ^
    "  new_stitch->next_row = NULL;\n" ^
    "  new_stitch->next = NULL;\n" ^
    "  new_stitch->prev = curr;\n" ^
    "  int count = 1;\n" ^
    "  if (curr != NULL) {\n" ^
    "    curr->next = new_stitch;\n" ^
    "    count = curr->count + 1;\n" ^
    "  }\n" ^
    "  new_stitch->count = count;\n" ^
    debug "  printf(\"stitch count: %d\\t\", count);\n" ^
    "  new_stitch->prev_row = prev_row;\n" ^
    "  if (prev_row != NULL) {\n" ^
    debug "    printf(\"prev row count: %d\\n\", prev_row->count);\n"
^
    "    prev_row->next_row = new_stitch;\n" ^
    "    prev_row = prev_row->prev;\n" ^
```

```
    "  } else if (type != CO) {" ^
    "    printf(\"ERROR! No stitch available to work type %d\\n\",
type);" ^
    "    exit(0);\n" ^
    "  }" ^
    "  curr = new_stitch;\n" ^
    "}\n"
(* Functions and definitions to be included at the beginning of the
generated
  program text. *)
let preamble = includes ^ stitch_enum ^ stitch_struct ^ globals
    ^ print_stitch_symbol ^ make_stitch
let for_loop times body = "for (i=0;i<" ^ times ^ ";i++){" ^ body ^
"}"
let new_row = "\nprev_row=curr;curr=NULL;\n"
let cast_on count =
    "make_stitch(CO);\n" ^
    "int i;\n" ^
    "for (i = 1; i < " ^ string_of_int count ^ "; i++) {\n" ^
    "make_stitch(CO);\n" ^
    "}\n"
(* Final program output options. *)
let print_row_count =
  debug "printf(\"row_count\\n\");"
  ^ "curr=first_stitch;i=0;\nwhile(curr!=NULL){i++;
curr=curr->next_row;}"
  ^ "\nprintf(\"This pattern has %d rows.\\n\",i);\n"
let print_stitch_count =
  debug "printf(\"stitch_count\\n\");" ^ "curr=first_stitch;\n"
  ^ "if(argc > 2){\n"
  ^ "  repeat=atoi(argv[2]);i=1;\n"
  ^ "  while(curr->next_row!=NULL && i <
repeat){i++;curr=curr->next_row;}}\n"
  ^ "  repeat=i;i=0;\n"
  ^ "
if(curr->next!=NULL){while(curr!=NULL){i++;curr=curr->next;}}\n"
  ^ "  else{i=curr->count;}\n"
  ^ "  printf(\"Row %d of this pattern has %d
stitches.\\n\",repeat,i);\n"
let print_chart =
  debug "printf(\"print_chart\\n\");"
  ^ "curr=first_stitch;\n"
  ^ "while(curr->next_row!=NULL){curr=curr->next_row;}\n"
  ^ "first_stitch=curr;\n"
  ^ "while(first_stitch!=NULL){\n"
  ^ "  if(curr->next!=NULL){\n"
  ^ "
```

```
while(curr!=NULL){print_symbol(curr->type,1);curr=curr->next;}\n"
  ^ "  } else {\n"
  ^ "
while(curr!=NULL){print_symbol(curr->type,0);curr=curr->prev;}\n
}\n"
  ^ "
printf(\"\\n\");first_stitch=first_stitch->prev_row;curr=first_stitch
"
  ^ "\n}\n"
let final_output =

"if(first_stitch->next_row!=NULL){first_stitch=first_stitch->next_row
}\n"
  ^ "if (argc > 1) {\n"
  ^ "if (!strcmp(argv[1], \"--row_count\")) {" ^ print_row_count ^
"}\n"
  ^ "if (!strcmp(argv[1], \"--stitch_count\")) {" ^
print_stitch_count ^ "}\n"
  ^ "if (!strcmp(argv[1], \"--print_chart\")) {" ^ print_chart ^
"}\n"
  ^ "}\n}\n"
```