Matt Farkas

Michael Yahner

Optimization for Parallel Compilers Final Project

Professor Dr. O’Neil

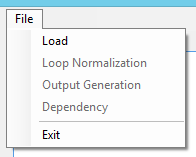
Due: April 18, 2013

**Program Overview**

The final project was to implement loop normalization if required based off the by clause of a for loop. From the normalized loop, we were to accomplish the generating the same output format as the output that was on the midterm. Starting with arrays, they are displayed by the first letter of their name in alphabetic order starting from zero, displaying their array type. This is followed by the upper bounds on new lines in order from outer most to inner with a zero delimiter after them. Next, comes all of the writes with line number, array name (position in alphabet), loop depth, and then constant with coefficient per each subscript. This is followed by another delimited zero and the reads with the same information.

**Instructions**

You will see below the File menu to operate this program. (In the README is the instructions on how to build and run this program.) From the file menu you can start off by selecting either “Load” or “Exit”. “Load” will open a dialog box to select the O’Neil code to work with. You then select “Loop Normalization” to see the normalized code in addition to being able to generate the output. Finally, click “Output Generation” to get the output that can be used in the midterm program.



To accomplish Three-Op conversion, a class called “ThreeOpConverter” was created that transforms O’Neil code into 3OP O’Neil code. A method called “Transform” is called to start the conversion process. The header is below:

public static string[] Transform(List<string> code, List<ThreeOPCode>intStatements, List<ThreeOPCode>letStatements)

Note the following input parameters:

List<string> code: The list of strings which contain the untransformed O’Neil Code

List<ThreeOpCode>intStatements: An empty list of ThreeOpCreation type for int’s

List<ThreeOpCode>letStatements: An empty list of ThreeOpCreation type for let statements

You might be wondering what the ThreeOpCreation object is used for. This is simply a class that was created to get and set the index and statements of the temporary variables used in the “Transform” method.

The following tranformation process is fairly complex, which includes a large amount of parsing based on different words, operations, and their relative locations in the statement. The following is a summarized flow of the process. For details, please review the ThreeOpConverter.cs class in the submitted code.

1. For Each Line of (O’Neil) code:
   1. Make a copy of the original statement.
   2. Trim the statement.
2. If the first word is “let”, “print”, or “if”, then the line is a candidate for transformation.
3. Determine if the candidate statements have one or more arrays and run the statement through one or more of the following functions:
   1. OneArrayTransformation: This function transforms ifs, lets, and prints/prompts one array statement into Three-Op.
   2. AfterEqualTransformation: Transforms statements that have multiple arrays.
   3. ParenthesisTransformation: Transforms statements with parenthesis into Three-Op.
   4. ValueExtraction: Creates a list of variables/constants in a non-Three-Op statement.
   5. OderOfOperations: Performs order of operations for all statements to make them Three-Op.
   6. LogicalOperator: Finds the logical operator in a statement.
4. Finally, the results (the code that has been transformed into Three-Op) is returned and printed to the user in the GUI.

To get an idea of what type of line parsing is used in ThreeOpCreation, consider the following code snip-it in the OneArrayTransformation method that parses text information from single-array statements (where “statement” is the current line of code being considered). Notice that positions of variable names, control flow syntax, and operations must be in their exact expected positions, or code will fail:

var indexBracketFront = statement.IndexOf("[", 0, StringComparison.Ordinal);

var indexBracketEnd = statement.IndexOf("]", 0, StringComparison.Ordinal);

. . .

if (int.TryParse(statement.Substring(indexBracketFront + 1, (indexBracketEnd - 1) - indexBracketFront), out result) == false)  
{  
 intStatements.Add(newThreeOPCode { Index = 2, Statement = "int t\_" + counter });  
 letStatements.Add(newThreeOPCode  
 {

Index = index,

Statement = original.Substring(0, original.IndexOf(type))   
 + "let " + "t\_" + counter + " = " +

statement.Substring(indexBracketFront + 1,

(indexBracketEnd - 1) - indexBracketFront)

});

Etc…

This next section will describe our attempt at constant propagation.

To assist in constant propagation, a class called “ConstantPropagation” was created. The method “Constants” is called to perform the propagation. Below is the function header, which simply takes the original code, and returns the code post constant propagation:

public static string[] Constants(List<string> code)

The following is a summarized flow of the constant propagation process. For details, please review the ConstantPropagation.cs class in the submitted code.

1. For each line of code:
   1. If the statement starts with “if” and contains “then” AND the statements contains a “let”, “goto”, “print”, or “prompt”, then continue.
   2. Find all variables created in the code and create a list by looking for int statements
2. Perform parsing on the lets statements to create the list of just variables
3. Remove variables from the list that have more than one let statement or they are used with an input.
4. For each variable (left in the list) :
   1. These are the values that are constants.
   2. For their let statement we take the value and add it to the list of that structure type to replace later.
   3. The int statement is removed because the variable is a constant.
5. For each variable, check the variables values against all the other variables to see if they are contained within. This means that if constant “c” is equal to constant “b + 10” then the constant “b” in that statement needs to be replaced with its value.
6. For each variable replace the constants where they are in the code with its value.

**Tests Performed**

A variety of tests were performed, including using all of the O’Neil test cases, and custom test cases to run our code. Unit testing was conducted when the code was being developed.

**Unexpected Behavior**

Automation - has a slight bug in that when it runs, anything other than 0 and 1 exit the program

Jacobi - still does not run because we still have not had to implement multi-dimensional arrays and Dr. O'Neil had talked about changing it just a one dimensional array

Sort Insertion - has a slight bug in, that when it runs, it does not swap variables correctly

Tax - crashes the program because a couple of the variables have not been initialized

All other cases worked fine

Three OP converter

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace OptimizingParallelCompiler

{

internal class ThreeOPConverter

{

/// <summary>

/// Transform's O'Neil code into 3OP O'Neil code

/// </summary>

/// <param name="code">the untransformed O'Neil Code</param>

/// <param name="intStatements">an empty list of ThreeOPCreation type for int's that may or may not be created</param>

/// <param name="letStatements">an empty list of ThreeOPCreation type for let statements that may or may not be created</param>

/// <returns>the code that has been 3OP transformed</returns>

public static string[] Transform(List<string> code, List<ThreeOPCode> intStatements, List<ThreeOPCode> letStatements)

{

var counter = 0;

//goes through each line of code

code.ForEach(x =>

{

//unmodified statement

var original = x;

x = x.Trim(' ', '\t');

var afterEqual = x;

//index in the code list

var index = code.IndexOf(original);

if (x.StartsWith("let", StringComparison.Ordinal))

{

//checks to see if there is a multidimensional array

if (Regex.Matches(x, ",").Count <= 0)

{

afterEqual = x.Substring(x.IndexOf("=") + 1, x.Length - (x.IndexOf("=") + 1));

var beforeEqual = x.Substring(0, x.IndexOf("="));

OneArrayTransformation(ref original, beforeEqual, ref counter, intStatements, letStatements, index, "let");

code[index] = original;

AfterEqualTransformation(ref original, ref afterEqual, ref counter, intStatements, letStatements, index);

code[index] = original;

ParenthesisTransformation(ref original, ref afterEqual, ref counter, intStatements, letStatements, index);

code[index] = original;

var elements = new List<string>();

afterEqual = afterEqual.Trim(' ');

ValueExtration(elements, afterEqual);

if (Regex.Matches(afterEqual, "[-+\*/%]").Count <= 2)

{

elements.Clear();

}

OderOfOperations(ref original, afterEqual, elements, ref counter, intStatements, letStatements, index);

code[index] = original;

}

}

else if (x.StartsWith("print", StringComparison.Ordinal) && x.Contains("["))

{

var array = x.Substring(0, x.IndexOf("["));

array = array.Substring(array.LastIndexOf(" ") + 1);

array += x.Substring(x.IndexOf("["), x.IndexOf("]") - x.IndexOf("[")) + "]";

OneArrayTransformation(ref original, array, ref counter, intStatements, letStatements, index, "print");

code[index] = original;

}

else if (x.StartsWith("if", StringComparison.Ordinal))

{

if (x.Contains("let") || x.Contains("goto") || x.Contains("print") || x.Contains("prompt"))

{

code.Insert(index + 1, original.Substring(0, original.IndexOf("if")) + " " + x.Substring(x.IndexOf("then") + "then".Length));

original = original.Replace(x.Substring(x.IndexOf("then") + "then".Length), "");

x = x.Substring(0, x.IndexOf("then") + "then".Length);

code[index] = original;

}

var equator = EquatorTypeAmount(afterEqual.Substring(afterEqual.IndexOf("(") + 1, afterEqual.IndexOf(")") - afterEqual.IndexOf("(") - 1));

OneArrayTransformation(ref original, original.Substring(original.IndexOf("(") + 1, original.IndexOf(equator.Type) - original.IndexOf("(") - 1), ref counter, intStatements, letStatements, index, "if");

OneArrayTransformation(ref original, original.Substring(original.IndexOf(equator.Type) + equator.Type.Length, original.IndexOf(")") - original.IndexOf(equator.Type) - equator.Type.Length), ref counter, intStatements, letStatements, index, "if");

equator = EquatorTypeAmount(original.Substring(original.IndexOf("(") + 1, original.IndexOf(")") - original.IndexOf("(") - 1));

if (equator.SpacesBefore > 1 || equator.SpacesAfter > 1)

{

var between = equator.SpacesBefore > 1 ? original.Substring(original.IndexOf("(") + 1, original.IndexOf(equator.Type) - original.IndexOf("(") - 1) : original.Substring(original.IndexOf(equator.Type) + equator.Type.Length, original.IndexOf(")") - original.IndexOf(equator.Type) - equator.Type.Length);

var elements = new List<string>();

between = between.Trim(' ');

ValueExtration(elements, between);

OderOfOperations(ref original, between, elements, ref counter, intStatements, letStatements, index, "if");

}

code[index] = original;

}

});

return code.ToArray();

}

/// <summary>

/// transforms ifs, lets, and prints/prompts arrays into threeOP code

/// </summary>

/// <param name="original">the original statement from the code</param>

/// <param name="statement">the statement to be checked/changed</param>

/// <param name="counter">counter for temp variable creation</param>

/// <param name="intStatements">list of temp variables creation</param>

/// <param name="letStatements">list of temp variables value set</param>

/// <param name="index">where the specific line of code resides in the program</param>

/// <param name="type">the type of statement being sent in</param>

private static void OneArrayTransformation(ref string original, string statement, ref int counter, List<ThreeOPCode> intStatements, List<ThreeOPCode> letStatements, int index, string type)

{

if (Regex.Matches(statement, "[[]").Count > 0)

{

var indexBracketFront = statement.IndexOf("[", 0, StringComparison.Ordinal);

var indexBracketEnd = statement.IndexOf("]", 0, StringComparison.Ordinal);

var result = 0;

var replace = string.Empty;

if (

int.TryParse(

statement.Substring(indexBracketFront + 1, (indexBracketEnd - 1) - indexBracketFront),

out result) == false)

{

intStatements.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

letStatements.Add(new ThreeOPCode

{

Index = index,

Statement = original.Substring(0, original.IndexOf(type)) + "let " + "t\_" + counter + " = " +

statement.Substring(indexBracketFront + 1,

(indexBracketEnd - 1) - indexBracketFront)

});

var previous = "t\_" + counter;

replace = statement.Substring(0, indexBracketFront) + "[t\_" + counter++ + "]";

if (type.Contains("print") || type.Contains("if"))

{

original = original.Replace(statement, replace);

statement = statement.Replace(statement, replace);

indexBracketEnd = statement.IndexOf("]", 0, StringComparison.Ordinal);

intStatements.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

var arrayName = statement.Substring(0, indexBracketFront);

letStatements.Add(new ThreeOPCode

{

Index = index,

Statement = original.Substring(0, original.IndexOf(type)) + "let " + "t\_" + counter + " = " +

arrayName + "[" + previous + "]"

});

replace = " t\_" + counter++ + " ";

}

original = original.Replace(statement, replace);

}

}

}

/// <summary>

/// transform's statements that have multiple arrays

/// </summary>

/// <param name="original">the original statement from the code</param>

/// <param name="statement">the statement to be checked/changed</param>

/// <param name="counter">counter for temp variable creation</param>

/// <param name="intStatements">list of temp variables creation</param>

/// <param name="letStatements">list of temp variables value set</param>

/// <param name="index">where the specific line of code resides in the program</param>

private static void AfterEqualTransformation(ref string original, ref string statement, ref int counter, List<ThreeOPCode> ints, List<ThreeOPCode> letStatements, int index)

{

var letStatement = string.Empty;

//checks to see if there is an array if not, it returns without doing anything

if (Regex.Matches(statement, "[[]").Count > 0)

{

//counts the number of arrays in the statement

var countBracket = Regex.Matches(statement, "[[]").Count;

var result = 0;

var replace = string.Empty;

// turns the array accesses into 3OP

while (countBracket > 0)

{

if (Regex.Matches(statement.Substring(0, statement.IndexOf("[")), " ").Count > 1)

{

var arrayname = statement.Substring(0, statement.IndexOf("["));

var spaceindex = arrayname.LastIndexOf(" ");

statement = statement.Substring(spaceindex);

}

var indexBracketFront = statement.IndexOf("[", 0, StringComparison.Ordinal);

var indexBracketEnd = statement.IndexOf("]", 0, StringComparison.Ordinal);

var arrayIndex = statement.Substring(indexBracketFront + 1,

(indexBracketEnd - 1) - indexBracketFront);

// dont remember why i did this

if (int.TryParse(arrayIndex, out result))

{

ints.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

var arrayName = statement.Substring(statement.IndexOf(" ", StringComparison.Ordinal),

indexBracketFront - statement.IndexOf(" ", StringComparison.Ordinal));

letStatement = original.Substring(0, original.IndexOf("let")) + "let t\_" + counter + " = " + arrayName + "[" + arrayIndex + "]";

original = original.Replace(arrayName + "[" + arrayIndex + "]", " t\_" + counter++);

}

else

{

ints.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

var previousInt = "t\_" + counter;

letStatements.Add(new ThreeOPCode

{

Index = index,

Statement = original.Substring(0, original.IndexOf("let")) + "let " + "t\_" + counter++ + " = " +

arrayIndex

});

ints.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

letStatement = original.Substring(0, original.IndexOf("let")) + "let t\_" + counter + " = " + statement.Substring(statement.IndexOf(" ", StringComparison.Ordinal) + 1,

indexBracketFront - (statement.IndexOf(" ", StringComparison.Ordinal) + 1)) + "[" + previousInt + "]";

var arrayname = statement.Substring(0, indexBracketFront);

if (Regex.IsMatch(arrayname, " "))

{

arrayname = arrayname.Substring(arrayname.IndexOf(" "));

}

original = original.Replace(arrayname + "[" + arrayIndex + "]", " t\_" + counter++);

}

if (statement.Length > indexBracketEnd + 2)

{

statement = statement.Substring(indexBracketEnd + 2);

}

letStatements.Add(new ThreeOPCode { Index = index, Statement = letStatement });

--countBracket;

}

}

statement = original.Substring(original.IndexOf("=") + 1);

}

/// <summary>

/// transforms parenthesis statements into 3OP code

/// </summary>

/// <param name="original">the original statement from the code</param>

/// <param name="statement">the statement to be checked/changed</param>

/// <param name="counter">counter for temp variable creation</param>

/// <param name="intStatements">list of temp variables creation</param>

/// <param name="letStatements">list of temp variables value set</param>

/// <param name="index">where the specific line of code resides in the program</param>

private static void ParenthesisTransformation(ref string original, ref string statement, ref int counter, List<ThreeOPCode> intStatements, List<ThreeOPCode> letStatements, int index)

{

if (Regex.Matches(statement, "[(]").Count > 0)

{

var isNested = statement.LastIndexOf("(") - statement.IndexOf("(") == 1 ? true : false;

var elements = new List<string>();

if (isNested)

{

while (isNested)

{

ValueExtration(elements, statement.Substring(statement.LastIndexOf("(") + 1, statement.IndexOf(")") - statement.LastIndexOf("(") - 1));

OderOfOperations(ref original, statement.Substring(statement.LastIndexOf("("), statement.IndexOf(")") - statement.LastIndexOf("(") + 1), elements, ref counter, intStatements, letStatements, index);

statement = original.Substring(original.IndexOf("=") + 1);

isNested = statement.LastIndexOf("(") - statement.IndexOf("(") == 1 ? true : false;

}

ValueExtration(elements, statement.Substring(statement.IndexOf("(") + 1, statement.IndexOf(")") - statement.IndexOf("(") - 1));

OderOfOperations(ref original, statement.Substring(statement.IndexOf("("), statement.IndexOf(")") - statement.IndexOf("(") + 1), elements, ref counter, intStatements, letStatements, index);

statement = original.Substring(original.IndexOf("=") + 1);

}

else

{

for (int i = 0; i <= Regex.Matches(statement, "[(]").Count; i++)

{

ValueExtration(elements, statement.Substring(statement.IndexOf("(") + 1, statement.IndexOf(")") - statement.IndexOf("(") - 1));

OderOfOperations(ref original, statement.Substring(statement.IndexOf("("), statement.IndexOf(")") - statement.IndexOf("(") + 1), elements, ref counter, intStatements, letStatements, index);

statement = original.Substring(original.IndexOf("=") + 1);

}

}

}

}

/// <summary>

/// does order of operations for all statements that are not 3OP already

/// </summary>

/// <param name="original">the original statement from the code</param>

/// <param name="statement">the statement to be checked/changed</param>

/// <param name="elements">the list of variables/constants</param>

/// <param name="counter">counter for temp variable creation</param>

/// <param name="intStatements">list of temp variables creation</param>

/// <param name="letStatements">list of temp variables value set</param>

/// <param name="index">where the specific line of code resides in the program</param>

/// <param name="type">the type of statement being sent in</param>

private static void OderOfOperations(ref string original, string statement, List<string> elements, ref int counter, List<ThreeOPCode> intStatements, List<ThreeOPCode> letStatements, int index, string type = "")

{

var changedString = statement;

if (elements.Count <= 2 || elements == null)

{

elements.Clear();

return;

}

var previousInt = "t\_" + counter;

var i = 0;

while (elements.Count > 1 && elements.Count > i)

{

original = original.Replace(statement, changedString);

statement = statement.Replace(statement, changedString);

if (Regex.Matches(original.Substring(original.IndexOf("=") + 1), "[-+/\*%]").Count <= 1 && type != "if")

{

changedString = changedString.Replace("(", "");

changedString = changedString.Replace(")", "");

original = original.Replace(statement, changedString);

elements.Clear();

return;

}

if (elements[i] == "\*" || elements[i] == "/" || elements[i] == "%")

{

intStatements.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

letStatements.Add(new ThreeOPCode { Index = index, Statement = original.Substring(0, original.IndexOf(type)) + "let t\_" + counter + " = " + ((i - 1) < 0 ? previousInt : elements[i - 1]) + elements[i] + elements[i + 1] });

changedString = changedString.Replace(((i - 1) < 0 ? previousInt : elements[i - 1]) + " " + elements[i] + " " + elements[i + 1], "t\_" + counter);

elements.RemoveRange(((i - 1) < 0 ? 0 : i - 1), ((i - 1) < 0 ? 2 : 3));

previousInt = "t\_" + counter;

elements.Insert(i - 1, previousInt);

counter++;

i = 0;

}

else

{

++i;

}

}

original = original.Replace(statement, changedString);

statement = statement.Replace(statement, changedString);

if (Regex.Matches(original.Substring(original.IndexOf("=") + 1), "[-+/\*%]").Count <= 1 && type != "if")

{

changedString = changedString.Replace("(", "");

changedString = changedString.Replace(")", "");

original = original.Replace(statement, changedString);

elements.Clear();

return;

}

i = 0;

while (elements.Count > 1 && elements.Count > i)

{

original = original.Replace(statement, changedString);

statement = statement.Replace(statement, changedString);

if (Regex.Matches(original.Substring(original.IndexOf("=") + 1), "[-+/\*%]").Count <= 1 && type != "if")

{

changedString = changedString.Replace("(", "");

changedString = changedString.Replace(")", "");

original = original.Replace(statement, changedString);

elements.Clear();

return;

}

if (elements[i] == "+" || elements[i] == "-")

{

intStatements.Add(new ThreeOPCode { Index = 2, Statement = "int t\_" + counter });

letStatements.Add(new ThreeOPCode { Index = index, Statement = original.Substring(0, original.IndexOf(type)) + "let t\_" + counter + " = " + ((i - 1) < 0 ? previousInt : elements[i - 1]) + elements[i] + elements[i + 1] });

changedString = changedString.Replace(((i - 1) < 0 ? previousInt : elements[i - 1]) + " " + elements[i] + " " + elements[i + 1], "t\_" + counter);

elements.RemoveRange(((i - 1) < 0 ? 0 : i - 1), ((i - 1) < 0 ? 2 : 3));

previousInt = "t\_" + counter;

elements.Insert(i - 1, previousInt);

counter++;

i = 0;

}

else

{

++i;

}

}

elements.Clear();

original = original.Replace(statement, previousInt);

}

/// <summary>

/// creates a list of the variables/constants in a non 3OP statement

/// </summary>

/// <param name="words">returned list of the statement</param>

/// <param name="statment">statement of variables/constants to be separated</param>

private static void ValueExtration(List<string> words, string statment)

{

while (statment.Length > 0)

{

if (statment.IndexOf(" ", StringComparison.Ordinal) < 0)

{

words.Add(statment);

statment = "";

}

else

{

words.Add(statment.Substring(0, statment.IndexOf(" ")));

statment = statment.Substring(statment.IndexOf(" ") + 1);

}

}

}

/// <summary>

/// finds the logical operator and returns all the information about it

/// </summary>

/// <param name="statement">the statement that has a logical operator</param>

/// <returns>returns a filled logical operator structure</returns>

private static LogicalOperator EquatorTypeAmount(string statement)

{

var type = "";

var amount = 0;

var s = statement;

var spacesBefore = 0;

var spacesAfter = 0;

if (statement.IndexOf("!=") > 0)

{

type = "!=";

amount = 2;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type)+ 1), " ").Count;

}

else if (statement.IndexOf(">") > 0)

{

statement = statement.Substring(statement.IndexOf(">"), 2);

if (statement == ">=")

{

type = ">=";

amount = 2;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type) + 1), " ").Count;

}

else

{

type = ">";

amount = 1;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type) + 1), " ").Count;

}

}

else if (statement.IndexOf("<") > 0)

{

statement = statement.Substring(statement.IndexOf("<"), 2);

if (statement == "<=")

{

type = "<=";

amount = 2;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type) + 1), " ").Count;

}

else

{

type = "<";

amount = 1;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type) + 1), " ").Count;

}

}

else if (statement.IndexOf("=") > 0)

{

statement = statement.Substring(statement.IndexOf("="), 2);

if (statement == "==")

{

type = "==";

amount = 2;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type) + 1), " ").Count;

}

else

{

type = "=";

amount = 1;

spacesBefore = Regex.Matches(s.Substring(0, s.IndexOf(type)), " ").Count;

spacesAfter = Regex.Matches(s.Substring(s.IndexOf(type) + 1), " ").Count;

}

}

return new LogicalOperator { Type = type, Amount = amount, SpacesBefore = spacesBefore, SpacesAfter = spacesAfter };

}

/// <summary>

/// holds information about a logical operator

/// </summary>

private struct LogicalOperator

{

/// <summary>

/// the actual representation of the logical operator

/// </summary>

public string Type { get; set; }

public int Amount { get; set; }

/// <summary>

/// the amount of spaces that come before it from (

/// </summary>

public int SpacesBefore { get; set; }

/// <summary>

/// the amount of spaces after operator but before )

/// </summary>

public int SpacesAfter { get; set; }

}

}

/// <summary>

/// class that holds index and statement of the temp variables

/// </summary>

public class ThreeOPCode

{

/// <summary>

/// Where the index in the program where the new statement will go

/// </summary>

public int Index { get; set; }

/// <summary>

/// the three op statement

/// </summary>

public string Statement { get; set; }

}

}

Constant Propagation

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace OptimizingParallelCompiler

{

public static class ConstantPropagation

{

public static string[] Constants(List<string> code)

{

var codeVariables = new List<Variables>();

code.ForEach(x =>

{

var index = code.IndexOf(x);

var original = x;

x = x.Trim(' ', '\t');

if (x.StartsWith("if", StringComparison.Ordinal) && x.Contains("then"))

{

if (x.Contains("let") || x.Contains("goto") || x.Contains("print") || x.Contains("prompt"))

{

code.Insert(index + 1, original.Substring(0, original.IndexOf("if")) + " " + x.Substring(x.IndexOf("then") + "then".Length));

original = original.Replace(x.Substring(x.IndexOf("then") + "then".Length), "");

x = x.Substring(0, x.IndexOf("then") + "then".Length);

code[index] = original;

}

}

});

var varabiles = code.FindAll(x =>

{

x = x.Trim(' ', '\t');

if (x.StartsWith("int", StringComparison.Ordinal) || x.StartsWith("list", StringComparison.Ordinal))

{

return true;

}

return false;

});

foreach (var item in varabiles)

{

codeVariables.Add(new Variables { Name = item });

}

var lets = code.FindAll(x =>

{

x = x.Trim(' ', '\t');

if (x.StartsWith("let", StringComparison.Ordinal) || x.StartsWith("input", StringComparison.Ordinal))

{

return true;

}

return false;

});

codeVariables.RemoveAll(x => x.Name.Contains("list"));

for (var j = 0; j < lets.Count; j++)

{

var s = lets[j];

if (lets[j].Contains("let"))

{

lets[j] = lets[j].Substring(0, lets[j].IndexOf("="));

}

}

var sentence = string.Join(string.Empty, lets);

for (int j = 0; j < codeVariables.Count; j++)

{

codeVariables[j].Name = codeVariables[j].Name.Substring(codeVariables[j].Name.LastIndexOf(" "));

}

var i = 0;

while(i < codeVariables.Count)

{

var pattern = @"\b" + codeVariables[i].Name;

var count = Regex.Matches(sentence, pattern).Count;

if (count > 1 || count <= 0)

{

codeVariables.RemoveAt(i);

--i;

}

else if (count == 1)

{

var index = sentence.IndexOf(codeVariables[i].Name);

var statement = sentence.Substring(index - 1 - "input".Length <= 0 ? 0 : index - 1 - "input".Length, "input".Length);

if (statement == "input")

{

codeVariables.RemoveAt(i);

--i;

}

}

++i;

}

for (i = 0; i < codeVariables.Count; i++)

{

//InformationOutput.InformationPrint(code.IndexOf(codeVariables[i].Name).ToString() + Environment.NewLine);

code.ForEach(x =>

{

if (x.Contains("let"))

{

var beforeEquals = x.Substring(0, x.IndexOf("="));

if (beforeEquals.Contains(codeVariables[i].Name))

{

codeVariables[i].Value = code[code.IndexOf(x)].Substring(code[code.IndexOf(x)].IndexOf("=") + 2);

code.Remove(x);

}

}

else if (Regex.Match(x,@"\b"+codeVariables[i].Name+@"\b").Success && Regex.Match(x,@"\bint\b").Success) //(x.Contains(codeVariables[i].Name) && x.Contains("int"))

{

code.Remove(x);

}

});

}

i = 0;

while (i <= codeVariables.Count - 1)

{

var j = 0;

while (j <= codeVariables.Count - 1)

{

var s = codeVariables[j].Name;

s = s.Trim(' ');

if (Regex.Matches(codeVariables[i].Value, @"\b"+s+@"\b").Count > 0)

{

codeVariables[i].Value = Regex.Replace(codeVariables[i].Value, s, codeVariables[j].Value, RegexOptions.IgnorePatternWhitespace | RegexOptions.Singleline | RegexOptions.IgnoreCase);

}

++j;

}

++i;

}

foreach (var item in codeVariables)

{

code.ForEach(x =>

{

var index = code.IndexOf(x);

//x = x.Replace(item.Name, item.value);

var pattern = @"\b" + item.Name + @"\b";

x = Regex.Replace(x, pattern, item.Value, RegexOptions.IgnoreCase | RegexOptions.Singleline | RegexOptions.IgnorePatternWhitespace);

code[index] = x;

});

}

return code.ToArray();

}

private class Variables

{

public string Name { get; set; }

public string Value { get; set; }

}

}

}